A Framework for Building Resilient Operating Systems

自己回復型オペレーティングシステム構築フレームワーク

February 2009

Hiroo Ishikawa

A Framework for Building Resilient Operating Systems

Hiroo Ishikawa

A Framework for Building Resilient Operating Systems

自己回復型オペレーティングシステム構築フレームワーク

A DISSERTATION
SUBMITTED TO THE GRADUATE SCHOOL OF SCIENCE AND ENGINEERING
OF WASEDA UNIVERSITY
IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF ENGINEERING

Hiroo Ishikawa

February 2009

Acknowledgements

This is a great ooportunity to express my respect to my thesis advisor, Prof. Tatsuo Nakajima. My research and this dissertation would not exist without his support. I would like to thank my colleague Dr. Alexandre Courbot for encouraging me in this project, giving me useful advice, and reading and commenting on most of the manuscript. I would also like to thank all the members at Distributed Computing Laboratory in Waseda University for encouraging me.

Abstract

Dependability has been becoming the largest concern as many consumer products contain computers with a large amount of software. The complexity in such software cases the increase of the visibility of failures. On one hand, software testing and verification have tried to produce bug-free software. On the other hand, software infrastructures such as operating systems are required to manage and control errors and failures by themselves. Resiliency means self-managing and self-control of errors and failures.

The resiliency of an operating system by restart recovery is studied. Restart recovery of a system, namely system reboot is commonly used to recover or sometimes avoid failures of systems. System reboot is much simpler and less resource consuming recovery method than the others such as primary-backup, data structure repairing, etc. However, the downtime due to the system reboot is generally unacceptable because it may end up with terminating services such as Web servers, media players, automatic ticket gates, robots, and so forth. Restart recovery is classified in terms of timing against failure. The classification characterizes the advantages and the disadvantages of restart recovery.

This dissertation proposes a framework for developing a resilient operating system, which is based on microkernel architecture. The framework enhances the dependability of an operating system by dividing it into fine-grain components and applying restart recovery to individual components upon a failure. Since restart recovery is applied at the component level, it is quicker than that of a whole system. This enhances the availability of the operating system, and the continuity of user applications.

A prototype operating system called Arc is developed under the framework and is evaluated by conducting a fault injection test. The results show that restart recovery successfully handles errors in inputs or the execution environment without sacrificing performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our life and work have depended on various kinds of computers including mobile phones, toys and home appliances. This makes us consider the dependability of computer systems. We need to provide administration-free computer systems because end users never expect to handle troubles on such devices. A dependable system is a system that never fails visibly, even though the system may internally undergo failure.

This research aims at increasing dependability of computer systems with operating system technology. Nowadays, computers are required to be as small as possible so that a manufacturer can embedded them into many different shapes. This limits hardware functionalities. Instead, software technology is often expected to take over this shortage. In particular, operating systems are a platform for technologies because they are the lowest layer of software systems.

Traditionally, the dependability of an operating system relies on hardware redundancy(Bartlett, 1981)(Patterson et al., 1988)(Gray and Reuter, 1993)(Siewiorek and Swarz, 1998)(Oppenheimer et al., 2002). A redundant system has extra hardware so that it takes over the main parts upon failure. Indeed, the original purpose of hardware redundancy is to make a system survive over hardware faults(Siewiorek and Swarz, 1998). However, it is also used for dealing with software faults(Pullum, 2001). Then, the dependability of a traditional operating system depends on redundant hardware that is overwhelming operating systems in mobile devices. This is because mobile devices are inherently limited in their size and power. Inevitably, there is no space for hardware redundancy in most of these devices. This means that an operating system for mobile devices cannot rely on hardware redundancy.

Resiliency is one of the important features of dependable systems. As opposed to dependability, the term resiliency has narrower meaning; in this dissertation a resilient system is defined as a system which involves a mechanism to recover its failure by itself. The more complex an operating system has become, the more difficult its users maintain the entire system. An operating system is required to have resiliency.

1.1 Motivation

In this dissertation, we propose a framework for developing a resilient operating system. There are some ways to develop a resilient software system in general, such as checkpoint-resume(Gray and Reuter, 1993)(Lowell et al., 2000), system reboot(Baker and Sullivan, 1992)(Chen et al., 1996), microreboot(Candea et al., 2004), data structure repairing(Demsky and Rinard, 2003), N-versioning(Randell, 1975)(Pullum, 2001), etc. We adopt microreboot as the basis of our framework, but modify its requirements for operating system components. The recent trend of application development separates an application into multiple processes to increase its stability and safety. We think this trend will spread to the construction of operating systems. To make an operating system more reliable, it should be separated into independent components. The philosophy of microreboot basically fits to this idea.

However, some kinds of operating system components, in particular, device drivers, do not fit in the philosophy of microreboot. Microreboot requires a component to be *crash-only*. A normal component can follow several different execution path which start with initialization, then provide a service, and finally exit or crash in the middle of the service. A crash-only component must be able to follow only one execution path which starts with initialization, then provides service, and crashes. The main idea of this crash-only software is that eliminating failure recovery by letting every component exit by crash. In case a component crashes but other interacting components are alive, it is just restarted.

The problem of microreboot and crash-only software is that the component has only one initialization procedure. For example, since some peripheral devices hold state which should not be discarded by the driver initialization, the policy of microreboot is not always acceptable. This research is motivated by this shortcoming of microreboot.

The framework proposed by this dissertation distinguishes the procedure for restart and the one for initialization(Ishikawa et al., 2008). The framework can deal with the operating system components that microreboot cannot handle. Since each component is required to involve its own restart procedure, we can keep the mechanism of the framework as simple as microreboot.

1.2 Contributions

There are two contributions of this dissertation.

The first contribution is a classification of restart recovery in terms of timing against failure. Restart recovery makes a different effects to a system depending on when it is executed. We consider the advantages and the disadvantages of the restart recovery techniques in three cases: pre-failure, on-failure, and post-failure.

The second contribution is a framework for failure recovery and a prototype operating system implementation based on the framework. We demonstrate the feasibility of the framework with the implementation. The prototype operating system is a multi-server operating system, which consists of a microkernel and user-level servers. For the evaluation of the framework and the prototype operating system, we conduct case studies and fault injection test.

1.3 Organization of the Dissertation

In the following chapters, we discuss the resiliency of operating systems and applying restart recovery to operating system components.

Chapter 2 introduces related work. This may help the reader to understand the context of the research. The chapter first explains the basics of dependability in operating systems. The basic terms necessary to understand the following chapters. Then, it shows related research projects. Since the focus of the dissertation is the dependability of operating systems, the chapter introduces research projects in the context of dependability and operating systems.

Chapter 3 explains the concept of restart recovery, which is applied to the framework proposed in the dissertation. This chapter chronologically classifies failure recovery techniques. It then indicates that the restart recovery with persistent memory is the most efficient technique in terms of resource consumption.

Chapter 4 describes the design and the implementation of the framework and the prototype operating system. The prototype operating system, which is built on top of a microkernel, provides persistent memory for each operating system components such as device drivers and file systems. Upon failure recovery, a failed operating system component is safely restarted and resumed transparently. The chapter explains the mechanisms for and policies on restart recovery in the prototype operating system.

Chapter 5 describes the evaluation of the framework and the prototype operating system. We first conduct case studies on several operating system components. The series of case studies show the concrete usage of the prototype operating system, and the efficiency of its restart recovery mechanism. We then conduct fault injection tests, which intentionally inject a fault into a normal component, to evaluate the resiliency of the framework.

Chapter 6 concludes the dissertation. We also discuss future direction in this research field.

Chapter 2

Dependability in Operating Systems

This chapter describes the context of this research. This chapter first explains the basic terms and definitions in dependable computing, and then the state of the art in operating system dependability from several different viewpoints.

2.1 Basics of Dependability

This section first takes a look at the basics of dependability. The term dependability has many meanings such as reliability, availability, safety, accountability, security, etc. In this dissertation, we consider a narrow meaning of dependability, which is its fault tolerance aspect.

2.1.1 Definitions

This section introduces the basic terms that are necessary to understand the following chapters. We first need to understand what a dependable operating system is supposed to deal with. That is actually program execution state and events caused by defects in the program. This section first takes a look at the relationship of them, and then show how the defects eventually become visible to the users of the system.

Fault, Error, and Failure

The definitions of *fault*, *error*, and *failure* in this dissertation are based on (Avizienis et al., 2004). A module behaves in a specified way. A failure is the event that occurs when observed behavior deviates from the specified behavior. An error is the state of the deviation between the specified behavior and the actual behavior. A fault is the identified or hypothesized cause of an error (Avizienis et al., 2004). A fault of a module causes an error in that module, and the error then causes a failure of the module. A failure in the module then becomes a fault in another module where it is involved (Figure 2.1).

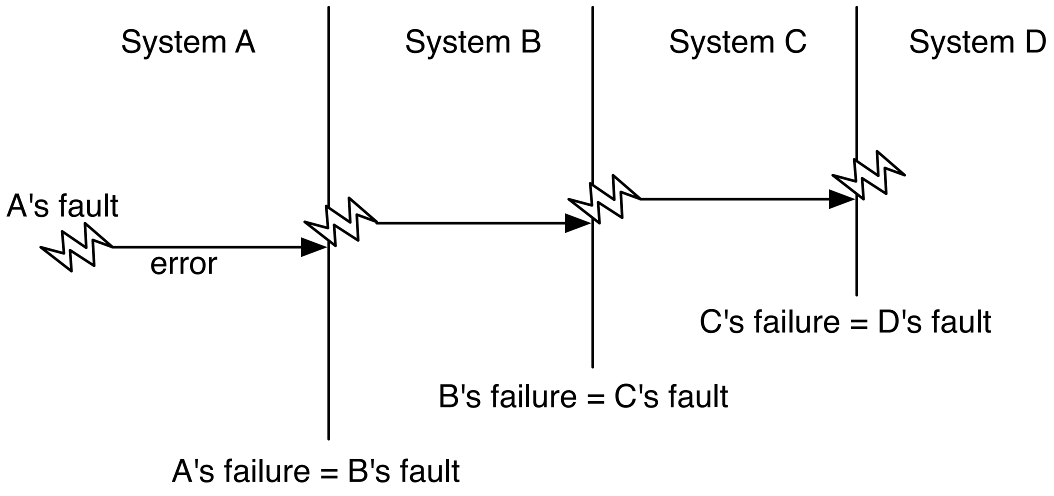


Figure 2.1: Failure propagation

The series of figures (Figure 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, and 2.8) illustrates the definitions. In these figures, the grid indicates all possible execution paths, including error paths, of a program. Each intersection indicates a state of a program, and each line indicates a state transition. Although a program always have three possible transitions at each state, we focus only on two possibilities, whether an execution path follows the specified behavior or not. In any way, an execution path progresses along the grid. The black or dotted black lines indicate the internal behavior of a program; the solid black lines indicate correct behavior, while the dotted black lines indicate incorrect behavior.

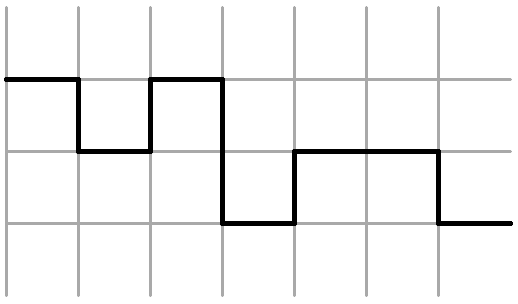


Figure 2.2: Specified behavior

Figure 2.2 shows the specified behavior of a program. The execution of the program is supposed to follow the same path as the specified behavior. If the execution of the program follows the same path as the specified behavior, it is referred to as legal (Figure 2.3). The legal behavior generates legal visible events (Figure 2.4). Visible events of a program are what an external entity of the program such as an end-user or another program can observe. For example, a user of a web browser observes the web page displayed by the browser changes to the next page when he clicks one of the links in the current page. The transition from the current page to the next page is a visible event. Visible events are indicated by black dots. Unlike the internal behavior of a program, state transitions in visible events may not follow the grid.

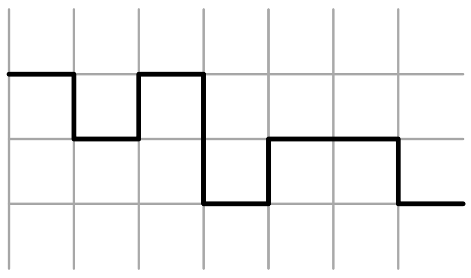


Figure 2.3: Legal behavior

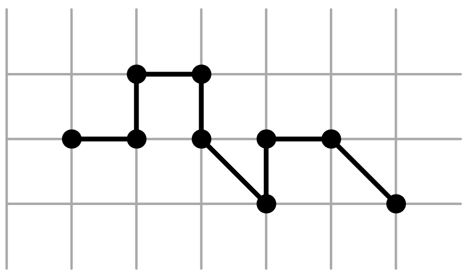


Figure 2.4: Legal visible events

Erroneous behavior deviates from the specified behavior because of a fault (Figure 2.5). As mentioned before in this section, the state of a program that deviates from the specified behavior is defined as error. The dotted line in Figure 2.5 indicates an error. If an error is exposed to outside a program, then the observer of the program observes a failure (Figure 2.6). Failure is also a deviation from the legal visible events (compare Figure 2.4 and 2.6).

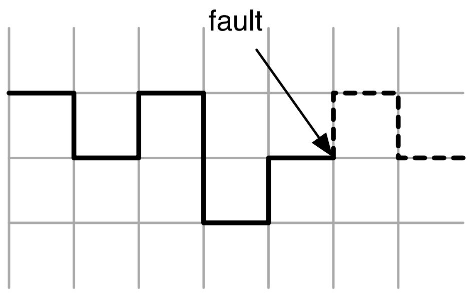


Figure 2.5: Erroneous behavior

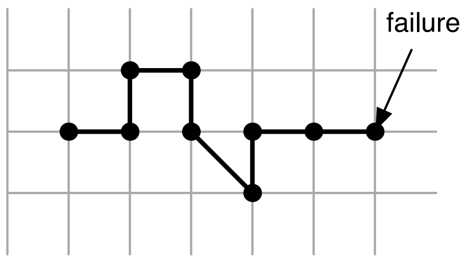


Figure 2.6: Visible events including failure

Recovery is to get the erroneous execution path back to the legal one by repairing (Figure 2.7). The recovered execution path may involve an error. However, the recovery is successful as long as the visible events are legal (Figure 2.8).

Crash, Hang, and Panic

We define the terms, *crash, hang and panic*. Crash, hang and panic are the types of failure.

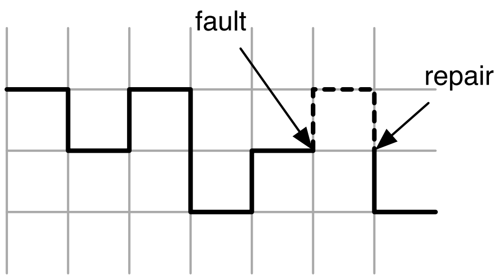


Figure 2.7: Recovered behavior

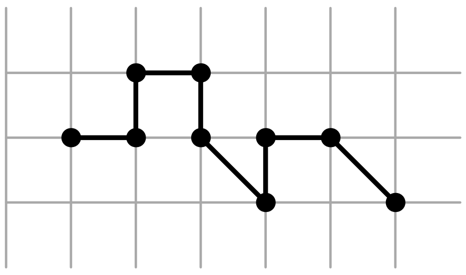


Figure 2.8: Recovered visible events

Crash *Crash* is a failure that violates the specification of the runtime system such as a CPU for operating systems, an operating system for applications, a Java virtual machine for Java applications. It is also a forceful termination of computation. A component is let crash mainly because the runtime system knows no way to recover the error inside the component or the runtime system itself involves a fault that drives it to recognize its own state in a wrong way. For example, in many operating systems, if an application attempts to access the kernel area, it is forced to crash by the operating system because the kernel area is specified as a forbidden area for applications.

Hang *Hang* is the state where the computation is never able to make a progress. Hang is observed outside the component as no response of the component to requests. This is caused by I/O blocking, scheduling starvation, dead lock, live lock, and so on. At a glance, some types of hang would not seem like an error state because the computation proceeds anyway. However, in that case, the execution repeat the same computation and it doesn't make any progress for its clients.

Panic Unlike the previous two types of failure, *panic* is reported by the failed component itself. A component can detect its erroneous state by checking invariants of itself. Checking invariants itself is a normal operation. However, the operations after an invariant is violated could be illegal unless the component repairs it. A component issues a panic when it cannot handle the violation of an invariant. While crash and hang are detected outside the component, panic is detected inside the component.

2.1.2 Fault Tolerance

Fault tolerance is a property of a system. A fault tolerant system can handle an error caused by a fault and take itself back to an error-free state. There are three approaches to realize fault tolerance.

Failure Avoidance

Failure avoidance is defined as preventing an error from being effective. This is done by verifying programs in order to ensure that they involve no faults. Compilers, verification tools, or sometimes program loaders are used in this case.

Failure oblivious computing is a variant of failure avoidance. Failure avoidance technologies generally warn if they find faults in a program. The main idea of failure oblivious computing is to append a functionality to a faulty program, functionality which modifies inputs to the program so that the bug doesn't become effective. For example, a long string that may expose a buffer over flow is shortened not to allow it. The result may return an error message of the system, however it is specified behavior.

Failure Ignorance

Failure ignorance is referred to masking a failure. Therefore, failure ignorance leaves the system in an erroneous state. An ignored failure is usually handle in a system at a higher level. For instance, the Java programming language provides an exception handling mechanism. Some Java programs ignore exceptions as long as the operating system does not detect its error. If the error is detected by the operating system, the process will be terminated.

Failure Recovery

Failure recovery is referred to returning to an error-free state through a certain repair process at runtime. Failure recovery is performed in cooperation with error detection and repairing. Restarting a system or a process is the most simplest repairing, however, this often causes the system down. In general, more detail error detection enables more complex repairing. Thus, if an error detection knows which part of a system is corrupted and how it is exactly corrupted, a system can repair only the corrupted data, keeping itself available.

2.1.3 Reliability

Reliability is referred to the continuity of specified behavior (Avizienis et al., 2004). Reliability is generally calculated by means of MTTF (*mean time to failure*). Suppose a system start up at t0. Then, the system suffers from a failure at t_f. The time from t_0 to t_f is the MTTF of the system. It is obvious that MTTF includes the time during which the system is under an erroneous state.

2.1.4 Availability

Availability is defined as the rate of the time during which the system operates normally to the entire time. The availability is calculated by means of MTTF and MTBF (*mean time between failure*). MTBF represents the time for repairing the failed system. Given total MTTF and total MTBF, the availability A of the system is calculated by the following formula.

$$A = MTTF/(MTTF + MTBF)$$

The total of MTTF and MTBF is the total operational time. If A is equal to 1, the system is a complete system and it is totally error-free. If A is equal to 0, that means that the system cannot serve in a correct way in any time. In terms of availability, failure avoidance and failure ignorance aim to increase MTTF, but failure recovery aims to decrease MTBF.

2.2 State of the Art in Operating System Fault Tolerance

This section explores the state of the art in operating system fault tolerance. We address software engineering approaches to construct a fault tolerant operating system, and runtime facility to make an operating system robust against failure. However, the runtime facility (failure recovery technology) is mainly described here because an operating system has to ensure its state is error-free and consistent in order to provide valid services to the user; other fault tolerant techniques such as fault avoidance or ignorance are very limited for the guarantee.

Fault tolerance has been implemented in many kinds of operating systems in a number of different ways. This section describes the related work in terms of failure transparency in operating systems. There is a large amount of implementations. They are roughly classified into two groups. The first group tries to achieve seamless failure recovery by means of checkpoint-restart. The last group puts weight on improving the restart process for the system crash recovery. This section first takes a look at checkpoint-restart technologies in operating systems, then restart technologies.

2.2.1 Checkpoint-Restart

A checkpoint is defined as a state that is necessary for resuming the execution of a system after failure. A system that supports checkpoint-restart needs to consider the range and frequency of checkpoints and storage for them. There are enormous research in this field. Hence, this section describes typical operating systems with checkpoint-restart facility.

NonStop kernel

Tandem's NonStop system aims at providing continuous operations in the presence of a single fault (Bartlett, 1981), with dedicated hardware such as redundant processors, buses, and power supplies. It incorporates *process-pairs* for its fault tolerance. A process-pair consists of the primary and the backup processes. The primary process tailors I/O devices or handles requests from clients, submitting a copy of each request to the backup process so that the backup process is able to take over the primary process upon a failure in the primary process. In order to use the facility, user programs must explicitly form checkpoints, that means the design of the user programs must be done carefully.

TARGON/32

TARGON/32 is a UNIX-based fault-tolerant system that relies on computers connected through a dedicated network (Borg et al., 1989). The kernel runs on every machine. The machine plays one of two roles: primary or backup. The primary serves regular operations, while the backup synchronizes its state with the primary in the background and takes over the operations of the primary upon a failure. The synchronization mechanism relies on *three-way atomic message transmission* with which a sender process delivers a message not only to the receiver, but also to the sender's backup and the receiver's backup simultaneously. TARGON/32 also provides the *sync* operation so that the primary process frequently synchronize its state to its backup. Frequent synchronization allows the backup to recover from a more recent point. Although user processes work under the default scheme, server processes need some customizations due to their performance optimizations.

Hypervisor-based Fault Tolerance

Bressoud and Schneider built a hypervisor that enables replicas of a guest operating system run without modifications to the hardware, the operating system, or application programs (Bressoud and Schneider, 1996). This is a primary-backup system; the replica takes over the failed primary. The main contribution of the project is to define a protocol to synchronize the state of the primary and the backup. One important feature of the protocol is to absorb the timing of interrupts, because interrupts are a non-deterministic event.

By means of hypervisor, they achieved building such a system with no modification to hardware, guest operating systems, or application programs. However, their implementation depends on some hardware specific functions to improve its performance. This hypervisor-based fault tolerant system runs programs about twice as slow as a bare machine does.

Rx

Rx is middleware that recovers processes from both deterministic and non-deterministic errors (Qin et al., 2007) by means of checkpoint-restart. The main idea of Rx is to checkpoint-restart with changing execution environment of a target process upon failure.

Changing the execution environment, some deterministic bugs can be eliminated, which a simple checkpoint-restart cannot eliminate. For example, some timing bugs (e.g. data race) may disappear if the process scheduling is changed. Or some memory-related bugs (e.g. buffer overflow and memory corruption) may disappear if the memory layout is changed. In this sense, Rx is a *environmental N-versioning* system.

Rx consists of five of user-level and kernel-level components: sensors for detecting and diagnosing failures, a checkpoint-and-rollback that forms and restores checkpoints of the target server application, environmental wrappers to change the execution environment during re-execution, a proxy that interposes between the server application and its clients and records and replays exchanging messages, a control unit for maintaining checkpoints during normal execution and determining the recovery strategy.

The environmental change is applied only when necessary. After the re-execution strategy is determined based on failure symptoms, the target server application is rolled back to a checkpoint and re-executed with the selected environmental wrapper. If the re-execution passes the point where the failure is observed, the new environment is disabled to avoid imposing time and space overhead.

2.2.2 Recovery with Persistent Memory

Persistent memory is a special kind of memory to store the data that is necessary to keep the continuity of the recovered service. Some implementations use secondary storage (i.e. hard disks) as the persistent memory. Some other implementations use primary storage (i.e. the main memory, or RAM). Programming interfaces for persistent memory are also varied. For example, an API for databases, raw memory access, or file operations can be provided.

Recovery Box

Recovery box is a stable area of memory in which the system stores carefully selected pieces of system, and from which the system can be regenerated quickly (Baker and Sullivan, 1992). Recovery box is implemented under Sprite, which is a UNIX-like distributed operating system (Ousterhout et al., 1988). It aims at designing for fast recovery rather than crash prevention to provide low-cost high availability. Recovery box is statically allocated; it is always located at a fixed address in an address space and its size is compiled in the operating system. Hence, the application recovery process can easily salvage the recovery box after system or process restart. Recovery box provides data structures for its own management, because it is separated into more granular chunks of memory. Recovery box is dynamic in this sense. Applications use (allocate or free) it through dedicated programming interface. This allows the accesses to recovery box to be atomic.

Chorus Hot Restart

Chorus Hot Restart technology is a facility for restart recovery of user processes, which are called actors (Abrossimov et al., 1996), under Chorus/ClassiX r3 operating system (Chorus systems, 1996b) (Chorus systems, 1996a). Hot restart is referred to restarting a failed actor to its initial entry point without reloading the program from stable storage such as hard disk. The Chorus Hot Restart technology also provides a special memory region to each actor, called *persistent regions* so that actors can form fine-grain checkpoints.

An actor is a unit of protection (protection domain). Thus, it is difficult for a failed actor to corrupt another actor. In order to keep consistency of cooperated actors, Hot Restart technology gives a high-level abstraction called *restart groups*. The actors in a restart group are restarted if one of its member actors fails. Restart groups form a hierarchy with the Chorus nucleus (kernel). The restart group to which the nucleus belong is the largest group. That is, if the nucleus is restarted, all actors are also restarted. A restart of a restart group can propagate to other groups.

The hot restart technology defines the level of restart, such as termination, reload, and hot-restart. The termination terminates the execution of a failed actor. The reload loads the program executed by a failed actor from stable storage and restart it, hence, the state of the actor is completely lost. On the other hand, the hot-restart only re-initializes and restarts the actor, hence the checkpoints in its persistent region are intact.

Any restart group except for the largest group can choose its restart policy whether it propagates its restart or doesn't, and changes the level of restart instead.

However, the authors only give the ideas of the mechanisms of Hot Restart technology. Hence, the interface and usage of the persistent regions are not addressed. For example, they don't answer how fine-grain the checkpoints are. This is less information than needed in order to able to determine the feasibility of the Hot Restart technology.

Rio File Cache and Rio Vista

Rio file cache provides a reliable storage with less disk traffic (Chen et al., 1996). To accomplish this, the Rio file cache provides a mechanism that protects main memory from corruption and preserves the contents of the main memory across systems reboot. In common case, applications write data synchronously through the disk in order to tolerate a system failure, thus, it makes a significant performance overhead. With the Rio file cache, since the data on the main memory is protected against corruption and system reboot, data doesn't have to be written to the disk synchronously.

Rio involves two main technologies. The first technology protects the main memory from wild write during a system crash. Rio adopts virtual memory protection and *code patching* for this purpose. With the virtual memory protection, the memory region for file cache is writable only when file cache procedures are executed. However, since the platform they use allows the kernel to access the region through physical addresses, the code patching technique is used to prevent it. The code patching modifies the kernel object code by inserting a check before every kernel operation that stores data, as proposed by Wahbe et al (Wahbe et al., 1993).

The second technology called *warm reboot* enables the file cache to survive a crash. Warm reboot consists of two steps. The first step is to maintain additional data to manage the file cache. The second step is to reorganize the file cache after reboot by means of this additional data. This dumps all of physical memory to the swap partition, and restores the file cache through a user-level process from the swap partition. This obviously implies that firmware doesn't cleans up the main memory.

Rio Vista is a recoverable virtual memory implementation based on Rio file cache (Lowell and Chen, 1997). Unlike the predecessor, recoverable virtual memory proposed by Satanarayanan et al., Rio Vista does not require to write log files immediately to the disk because Rio file cache provides persistency of file cache across system reboot. Additionally, Rio Vista does not require undo log. These improvements reduce the performance overhead of recoverable virtual memory.

2.2.3 Crash-Only Software

Crash-only software is first proposed by Candea, et al. Candea and Fox (2003). It is the idea that applies system reboot in fine-grain components. The time for restarting a fine-grain component is less than that of restarting a whole system. This improves availability of a system, because MTBF becomes small. There are some variants of the crash-only software in operating systems. The idea of our research is also based on the idea of the crash-only software.

Nooks

Nooks is an extension of the Linux operating system (Swift et al., 2004b) (Swift et al., 2004a). The purpose of Nooks is to make Linux tolerate its device driver crashes, which is a major cause of operating system crash. In the original Linux kernel, device drivers are part of the kernel; they reside in the kernel space, hence the kernel crashes if one of the device drivers crashes. Nooks isolates each of the device drivers into a separate protection domain for better fault containment.

Nook also recovers device drivers upon failure. It unloads a failed device driver and restart it from a safe initial state. In order to keep the consistency between the device drivers and applications, Nooks delivers error messages to the applications to notify the device drivers have been restarted. Nooks was able to detect about 75% of failures according to its evaluation result.

However, the simple way of device driver failure recovery described above caused applications crash because few of the applications deal with error notifications from the device drivers being recovered. Thus, the authors also proposed *shadow drivers* as an extension of Nooks (Swift et al., 2004a). When a failure occurs, the shadow driver inserts itself temporarily in place of the failed driver and offers the service on behalf of the failed driver. While a device driver offers normal operations, it shadow driver monitors all communication between the kernel and the device driver (*passive mode*). While the device driver is recovered, the shadow driver communicates with the kernel on behalf of the failed device driver, and intercepts and responds to the restarted device driver. Then, the shadow driver restores the state of the device driver so that it can process I/O requests made before the failure.

Shadow drivers are implemented for classes of device drivers (i.e. the interfaces to device drivers). Hence, a single shadow driver implementation can support any device driver in the class. However, this leads a limitation for the device drivers with non-standard interface.

Process Resurrection

Process resurrection is a recovery technique based on process restart for hard real-time systems (Lee and Sha, 2005). Using signal handler mechanism provided by Unix-like operating systems, the authors claim that it can recover the failure of hard real-time processes efficiently and quickly. However, this is not the case for interactive processes.

The mechanism provides a custom signal handler that replaces the crashed process image with a newly loaded one, retaining its process id. The state that is necessary across restart is passed via the arguments or the shared memory of Linux operating system. The persistent storage based on the shared memory is process-local and is managed through dedicated programming interface. Hence, cooperation with other processes is less considered. This is because the process resurrection targets real-time applications, which require simple state restoration.

Since the process resurrection relies on the crash detection mechanism originally provided by the operating system kernel, rather than a monitoring system, no runtime overhead is involved.

MINIX3

MINIX3 is the third version of UNIX-like microkernel operating system. Although it originally aimed at an educational use, the latest version (MINIX3) intends to provide a fault tolerance facility, called the *reincarnation server*. The reincarnation server is a single user-mode process that is in charge of restarting and reintegrating failed processes.

MINIX3 is a multi-server operating system, where some operating system services such as device drivers, file systems, and network protocol stack run in user-mode. These user mode servers are strongly isolated using protection domains provided by the MMU (Memory Management Unit).

Restart of a failed process is policy-driven. The default policy is null, that is, the reincarnation server simply restarts a failed process. The administrator can provide a specific failure recovery as a recovery policy of specific types of processes. The policy file is written in a shell script-like language.

The reincarnation server itself is insufficient to restart a process. It needs another server called the data store to cooperate with it. The data store plays two roles. Firstly, it offers naming service which translates a string to a process id. Secondly, it serves as a database. Processes store the data that they need to preserve across restart to the data store.

The reincarnation server has currently only been evaluated with device drivers. According to the result of its evaluation, network device drivers and block device drivers are transparently recovered from failure. However, character device drivers are not.

Choices

Choices is an object-oriented operating system developed at the University of Illinois at Urbana-Champaign. System resources, policies and mechanisms are represented by objects organized in class hierarchies.

Choices implements four types of error detection techniques (David and Campbell, 2007). First, it uses virtual memory protection. Like other microkernel operating systems do, Choices isolate some operating system services and user applications into user-mode protection domains. Some device drivers and file systems are implemented as isolated OS components. Second, processor exceptions are handled at language-level. Choices provides a C++ library to handle CPU exceptions by means of C++ exception handling mechanism (David et al., 2006). Third, code checksum is computed periodically in order to detect the corruption of critical kernel code on memory. Last, watchdog timers are used to detect infinite loops in the operating system code.

Choices implements six types of recovery techniques (David and Campbell, 2007). First, Choices supports code reloading to cope with transient memory faults such as bit-flips or memory corruption due to invalid instructions in system code. When the system detects this type of error through CPU exceptions, the handler simply reloads the failed instruction from a memory mapped flash storage. Second, Choices supports component-level micro reboot. A failed component is reinitialized, or destroyed and re-created. It reprocesses the request before the failure. Third, critical operating system services including the paging daemon and the process dispatcher are automatically restarted upon failure. Fourth, cooperating with the watchdog timer error detection, Choices recovers operating system and user state in volatile memory (David et al., 2007). Watchdog timers reset the processors, the memory management unit, and interrupt subsystem if the timers expire. In this case, volatile memory is still preserved. Hence the system can reconstruct its state from the memory. Fifth, software transactional memory is applied to Choices components so that they can rollback and retry the aborted operations. As the authors noted, this technique is different from micro-rebooting which re-initializes the failed component. Last, process-level restart is attempted if transparent recovery is not possible or if the recovery process itself encounters errors.

2.2.4 N-Versioning

Different versions of a program behave differently. Therefore, even if one version suffers from failure, another version may not suffer. N-Versioning is based on this idea.

Generally speaking, N-Versioning is utilized with checkpoints. For failure avoidance, a system runs N-Versions concurrently and checks the result at each checkpoint to find out errors. In this case, $2t+1$ versions of program are necessary, where t is the number of failed versions at the same time. The system tolerates at most t of failed versions at a checkpoint. The correctness of results at a checkpoint is calculated by voting algorithm. In order to build a system that tolerates four failed versions, at least nine different versions are necessary.

For failure recovery, a system holds $N-1$ versions as backups of the primary.

2.2.5 Language-based Fault Tolerance

This category produces a dependable system cooperating with programming languages and/or compilers. The purpose of language- or compiler-support is to find faults in a program before the execution. If a software fault is found before the module is installed and run, a system can avoid failure.

Singularity

Singularity is an operating system whose primary goal is dependability(Hunt et al., 2005). It has pursued the goal by building on advances in programming languages and tools.

A process in Singularity is called a software isolated process (SIP)(Aiken et al., 2006). SIPs run on a microkernel. Not only applications, but also file systems, network systems, and other operating system components run as individual processes. Hence, Singularity is a multiserver operating system. A SIP consists of a set of memory pages, a set of threads, and a set of channel endpoints. User programs are written in a dedicated programming language called Sing#(Fähndrich et al., 2006) and verified to ensure they are type and memory safe. Because user program is verified safe, several SIPs can share the same address space. Moreover, SIPS can safely execute at the same privileged level as the kernel.

The Singularity project indicates the construction of the future of operating systems. Current operating systems are written in low-level programming languages such as C, C++, and assembly. This causes programming flaws. Cooperating with programming languages and tools that ensure safety, one can eliminate the current shortcomings. However, failure recovery mechanisms are not installed in Singularity. If Singularity involves a failure recovery mechanism, it becomes more dependable.

2.3 Summary

This chapter first introduced the basics of dependability in operating systems. Then, the chapter defined and explained various terms related to fault tolerant computing that would be necessary to understand the discussions through out this dissertation.

Enormous amount of work has been done in the dependability of operating systems. Software engineering approaches have tried to offer the ways of programming dependable operating systems, or the frameworks of dependable operating systems. Most of the fault tolerant technologies applied to operating systems are based on *restart recovery.* The restart recovery is known as very effective way to bring a failed system to an error-free state, because it builds the state of the system up from scratch. Thus, the major issues of dependable operating system technologies have sought how to make an effective restart mechanism to reduce the recovery time and how to keep the consistency of the execution state of an operating system across the recovery so that the user doesn't notice or observe the failure.

Half of the research projects shown this chapter hide their recovery mechanisms from applications. Eventually, they have to keep most of application's state for recovery; this requires redundancy and/or a large amount of storage. The other half requires applications to cooperate with them. However, they perform system reboot to recover the error state. This obviously leads to significant down time.

Chapter 3

Restart Recovery

Restart recovery is a method of failure recovery that eliminates faults by restarting the failed process. Restart recovery may loose the latest execution state because it usually carries re-initialization of the failed process, which causes downtime.

This chapter classifies restart recovery techniques in terms of timing it is taken place and the quantity of a process's state that is preserved across restart. After defining a system model, we first characterize the restart recovery techniques in terms of the timing at which they are performed. Then, we discuss the quantity of state that each of the techniques has to preserve for consistent recovery. In summary, this classification tells us that using persistent memory as a non-volatile part of main memory is efficient.

3.1 Prior Research

The classification of failure recovery in general is not new. For example, Gray et al.(Gray and Reuter, 1993), Siewiorek et al.(Siewiorek and Swarz, 1998), or Pullum(Pullum, 2001) made classifications of failure recovery techniques. These precursors focused on failure recovery as reaction to a failure. In terms of failure transparency, the reactive failure recovery is not the only way. We should distinguish broader range of failure recovery techniques which carry out before failure, or ignore failure.

Moreover, the precursors have different assumptions. Hence, they apply different recovery techniques. Gray et al. discuss fault tolerance of database systems based on checkpoint-restart. Siewiorek et al. mainly address the reliability of hardware which requires hardware redundancy. Pullum focuses on pure software which is free from the restriction of computational resources unlike operating systems. Thus, voting algorithm is mainly discussed. This dissertation focuses on failure recovery in operating systems. We adopt process restart for failure recovery.

3.2 System Model

The first task of this chapter is to define a system model. Through this chapter, we assume that a system consists of a supervisor, servers and clients (Figure 3.1). The supervisor detects errors in the servers, and triggers recovery. A server is defined as an independent process that provides services to clients. A client is also an independent process that performs some tasks interacting with servers. By process, we mean a unit of fault containment or a protection domain; an error in a process cannot directly contaminate other processes (e.g. *wild writes*). In the context of operating systems, the kernel, services such as file systems and network protocol stacks, and user applications fall into the supervisor, the servers and the clients, respectively. For simplifying the discussion, none of the servers run as a client of another server.

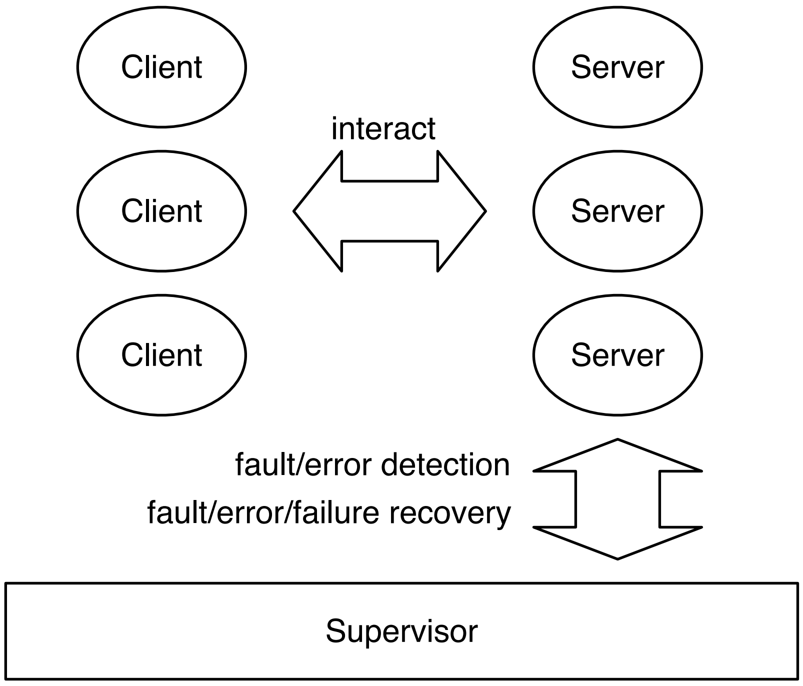


Figure 3.1: System Model

Clients and servers communicate synchronously. A client delivers a request to a server, the server processes the request, then replies to the client. The client is blocked during the request is processed by the server. A failure in a server aborts the communication between a client and the server. The client will then recognize the failure through the communication error thrown by the supervisor. A server may reply an error message due to an incorrect message from a client. This is not the case for failure.

The goal of each recovery technique described here is implemented in the supervisor to make the recovery of failed servers transparent from their clients. In other words, each of the recovery technique has to satisfy *consistent recovery* (Lowell, 1999).

3.3 Timing of Recovery

Restart recovery can be applied to three different periods of a process's lifetime (Figure 3.2). These three periods are divided by the time of error detection and the time of failure. The time of error detection, t_d, is defined as the time when the supervisor detects an error in a process. The time of failure, t_f, is defined as when an error in a server becomes visible to the clients.

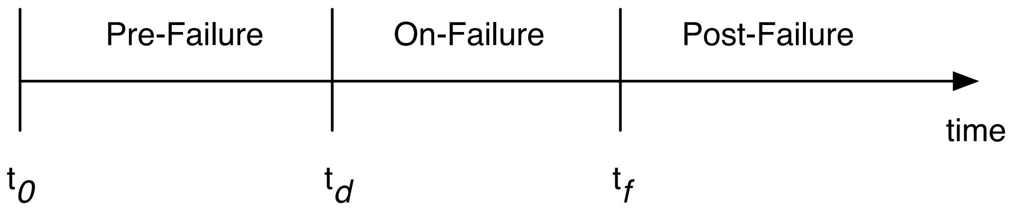


Figure 3.2: Pre-failure, on-failure, and post-failure

The period before t_d, between t_d and t_f, or after t_f is called *pre-failure*, *on-failure*, or *post-failure* respectively. A fault appears in the pre-failure state of a process. The fault causes an error which contaminates the process, while the process behaves legally from the client's viewpoint. If the error is detected at t_d, the process transits to the on-failure state. The supervisor tries to recover the process during this period. If the supervisor succeeds in the recovery, the process transits to the pre-failure state again. If the error is undetected or the recovery failed, the error becomes visible to, or the failure is observed by, the clients at t_f.

We refer the recovery techniques for on-failure, pre-failure, and post-failure to reactive recovery, proactive recovery and delayed recovery respectively. The following sections describes these recovery techniques one by one.

3.3.1 Proactive Recovery

Proactive recovery is a way to bring a system in an erroneous state back into a normal state before an error is detected (Huang et al., 1995)(Castelli et al., 2001). During the pre-failure period, a server process behaves correctly in terms of the interactions with its clients, but might behave incorrectly in terms of resource utilization. If the resource utilization of a process is incorrect, it will cause failure. For instance, memory leak causes an out-of-memory error at t_d and then the process is terminated at t_f. In the pre-failure state, a process or a system is contaminated by an error until it is detected. It is important for proactive recovery to detect an erroneous behavior of a server in terms of resource usage. The supervisor needs to detect the erroneous state of a server and observe how it behaves, even though the server correctly works for the clients.

Since proactive recovery assumes the interactions between a server and its clients, it can expect high probability of going forward to a quiescent state where restart recovery breaks no significant state of the server. In particular, a quiescent state is when the server has replied to a client. This is a characteristic only proactive recovery has.

Proactive recovery is mostly used for resource starvation problem such as memory leaks. This type of errors is either deterministic or non-deterministic but makes no effect to computation, although it eventually stops the computation if it is left unrecovered. Therefore, it is easy to bring the system to a quiescent state where the state that is necessary for the recovery can be safely captured.

Error Detection

Proactive recovery requires error detection in order to act while the error is latent. The error detectors find out an error through the observation of a server's behavior, in particular its usage of resources. This means that the observable error types are limited to errors wasting resources, such as memory leaks, CPU starvation, and dead locks. This type of error, particularly memory leaks, hardly has an effect on the visible behavior of a server. Since the error does not directly cause a failure, the recovery can be delayed. On the other hand, the detector takes time to analyze a server's behavior and determine its error state.

Example of Proactive Recovery

Software rejuvenation is an example of proactive recovery. Software rejuvenation requires computation models and/or statistics in order to watch resource utilization of a system. Based on the models and statistics, the time of failure is predicted (Huang et al., 1995)(Castelli et al., 2001)(Gross et al., 2002). When the monitored behavior breaks the models, or when the statistical result exceeds the threshold, the target system is restarted to bring the system to an error-free state. Recovering memory leak error is a typical example that software rejuvenation is applied to. The detector monitors the load of memory usage periodically. It restarts the system when the load exceeds the threshold.

Limitations

Proactive recovery depends on the precision of error detectors (i.e. the precision of behavior models, statistics, etc). This makes the construction of the supervisor complicated because the detectors have to be implemented in the supervisor. Since every component behaves differently, the detector may need to diverse for every type of components for its precision.

3.3.2 Reactive Recovery

Reactive recovery is the way of performing recovery in response to a result of error detection, and commonly used among many fault tolerant systems (Randell, 1975)(Bartlett, 1981)(Borg et al., 1989)(Siewiorek and Swarz, 1998). When the reactive recovery is performed, the error is already visible to the supervisor, but not visible to the clients. If no reactive recovery is performed, the failure becomes visible to the clients. When a failure occurs in a system, the error could be already propagated inside the system, hence, reactive recovery must ensure the propagated errors are removed. This is the main difference of reactive recovery from proactive recovery.

Error Detection

There are three types of error detectors. The first type is detectors with compilers or language runtime systems. A runtime type checker and software fault isolation are included in this category. A runtime type checker verifies if each operation breaks data types (Lindholm and Yellin, 1999),(ECMA International, 2005). For example, it checks a write operation on an array does not break its boundary. Software fault isolation is more coarser grain boundary checking (Wahbe et al., 1993). It ensures a component never make a direct access to another component.

The second type is a detector with hardware. Detectors with the MMU protection mechanism and CPU exception mechanism fall into this category. A result of these types of detectors are delivered to a software layer, normally the supervisor. The supervisor then verifies if the result violates the specification.

The last type is detection via calls to the supervisor. The supervisor determines a certain type of illegal arguments as a serious fault. Or, a server may report its failure through the *panic* call.

Examples of Reactive Recovery

There are many implementations of the reactive recovery. We mention here several popular technologies.

Checkpoint-restart recovery takes the failed process back to the previous checkpoint by means of snapshots or undo logs, then to the consistent state by means of redo logs. Checkpoint-restart recovery works transparently from a process so that the program is left unmodified.

Crash-only is another kind of reactive recovery (Candea and Fox, 2003). Instead of forming checkpoints, crash-only requires each component constructed in a way that it can safely crash anytime and it is always consistent. A crash-only component is recovered by restart; the supervisor cleans up the resources the crashed component used before failure, and reallocates and reintegrates the component to the system.

Limitations

Reactive recovery is limited in state management. When reactive recovery is applied, the state of a server might be contaminated by the error. The mission of reactive recovery is to remove this contamination while keeping consistent interactions with the clients. Reactive recovery obviously cannot go forward to a quiescent state. Thus, it has to go backward to a previous quiescent state. Since the last interaction is not always the same as the previous quiescent state, recovery has to fill the gap between them. For example, checkpoint restart records a quiescent state at each checkpoint, and bring a server to the consistent state with redo logs after restarting at the last checkpoint.

3.3.3 Delayed Recovery

Delayed recovery is a variation of reactive recovery that performs in the post-failure state. A failed system is not suspended at the moment the error became effective, but required to continue running as if it is in normal mode. According to the definition of reactive recovery, it is impossible to run a failed server. However, it is possible if the recovery platform can virtualize a failure to a normal event. In practice, this is implemented as degraded mode, or failure ignorance.

The idea of delayed recovery is to bring the failed system to the nearest quiescent state after the failure, in order to easily capture the consistent state or just avoid downtime upon the failure. While the recovery is postponed, the error may continue to contaminate the system. Thus, to deploy delayed recovery, the system must ensure that failure is not significant and error is not propagated. This is possible in some cases. For instance, failure in an optional component such as disk de-fragmentation, etc. can be ignored and the recovery can be delayed.

The difference of the delayed recovery from the other two recovery methods is the necessity of the failure virtualization. As describe above, the failure virtualization is referred to a technique for pretending normal mode in a failed system. The degradation of functionality is one of popular techniques of the failure virtualization. It is also possible to assign a page to an access violation. However, this is only the case for an allocation fault; the first access to an array that should have allocated in the heap but has not allocated, for instance.

Example of Delayed Recovery

Consider the example shown above. The allocation of an array is temporarily validated with an extra page mapping. The failed server replies to a client by means of the array and reaches a quiescent state. The supervisor then restarts the failed server for recovery.

There are two ways for recovery. The first way is to preserve only certified state that the server defines in advance. Thus, the array temporarily allocated is discarded. If the interactions with the client depends on the array, this breaks the consistency between the states of the client and the server, hence the server failed transparent recovery. The second way is to preserve all the state including the array, and let the server salvage the state after restart. The supervisor has to provide a set of abstractions and APIs to operate on the failed state. This would make a server programming more difficult.

Limitation

The purpose of delayed recovery is to bring a failed system to a quiescent state that facilitate to capture the state for the recovery. However, the recovery process becomes complicated.

3.3.4 Summary of the Timing of Recovery

There are three recovery techniques in terms of their timing against failure. The proactive recovery takes the advantage that an error has not been effective yet. Most of the existing recovery techniques belong the reactive recovery, which is applied when a fault or an error is exposed by the supervisor. The delayed recovery is the most difficult recovery technique of the three. This is because a failed server and/or the supervisor needs to salvage healthy state in the state contaminated by failure.

3.4 Quantity of State Necessary for Consistent Recovery

The previous section has discussed the timing of the restart recovery. This section then discusses the quantity of state for consistent restart recovery. As described in Chapter 2, the *consistent recovery* makes failures transparent. In other words it makes no effect on a sequence of visible event. To keep the consistency of visible events, some servers preserve their internal state across restart, and some others reproduce it.

The state preservation has a trade-off. If a large quantity of state is preserved, a server can be quickly restarted. However, a fault is included in a higher possibility than the case where a small quantity of state is preserved. On the contrary, if a small quantity of state is preserved a fault remains in a low possibility. However, a server may need to reconstruct its latest state before the crash, in order to make the recovery transparent. This lets the recovery of a server slow down or makes transparent recovery impossible.

3.4.1 Complete Restart

The first recovery method discussed here is complete restart of a server. This type of restart recovery forces a server to start over. Thus, a server with restart recovery completely looses its execution state. Depending on the quantity of state that a server holds, transparency of restart recovery changes. If a server holds the state of the clients, the transparency of recovery is very low because the state is lost by restart. On the other hand, if a server hold only its local state, a restart recovery is very efficient. The disappearance of a server's internal state must be taken into account during the construction of servers and clients.

In the case of a server holding the state of a client, the client should be notified of the recovery of the server to update its state. In terms of an operating system, an inter-server consistency can be maintained in this *para-transparent recovery*.

This category includes system reboot as recovery, which is a common recovery method among Internet server administrators (Candea et al., 2004). System reboot restarts all over the system including hardware. As pointed out by Candea et al. (Candea et al., 2004), this way takes a long down time that exposes the failure to end users.

Crash-only software is another example of the complete restart. As described in Chapter 2, the philosophy under the crash-only software is to construct a software component stateless so that the supervisor can restart them anytime and avoid recovery. In other words, crash-only components have no state to keep consistent with each other. This is useful for two reasons. First, the crash-only software assumes that a system is divided into fine-grain crash-only software components, where crash of a component does not require system reboot. Second, the crash-only software assumes Internet servers which users accept delays in responses of hundreds of millisecond order.

Timing of Complete Restart

Now, we would like to see the relationships between the complete restart and its timing. Since the complete restart completely discards the state of a server, the clients and the server are required to reach a quiescent state in advance, where the clients' state in the server can be lost. For instance, a file server reaches to a quiescent state where no files are opened.

Proactive recovery satisfies the requirement of the complete restart because a server can reach the nearest quiescent state during recovery. The complete restart on failure limits the design of systems like crash-only software. If the supervisor is able to guarantee the safety of delayed recovery, the complete restart in the post-failure phase is valid.

3.4.2 Restarting at a Checkpoint

Checkpointing is another method of preservation. A checkpoint is defined as a set of a server's state that the supervisor forms for recovery of the server. A server is generally required to cooperate to form checkpoints. However, the server never knows the contents of a checkpoint. Upon recovery, the supervisor resumes the server's state with one of the checkpoints (usually the latest one), then restarts the server. We call the checkpoint where a server is resumed, *the resuming point*. Checkpointing is popular among long running applications such as scientific simulators because they may loose the latest state of computation but not all of it.

Upon a failure, the supervisor restores the state of the failed server with one of the checkpoints that it has formed until then, and restarts the failed server. To synchronize the state of the server and its clients, checkpointing usually preserves not only checkpoints, but also redo logs. Replaying the interactions with the clients by means of the redo logs, the server is updated to the latest state 3.4.2.

The last checkpoint may involve an error state. If so, redo logs reproduce the same failure. In this case, the supervisor restores a older checkpoint and older redo logs.

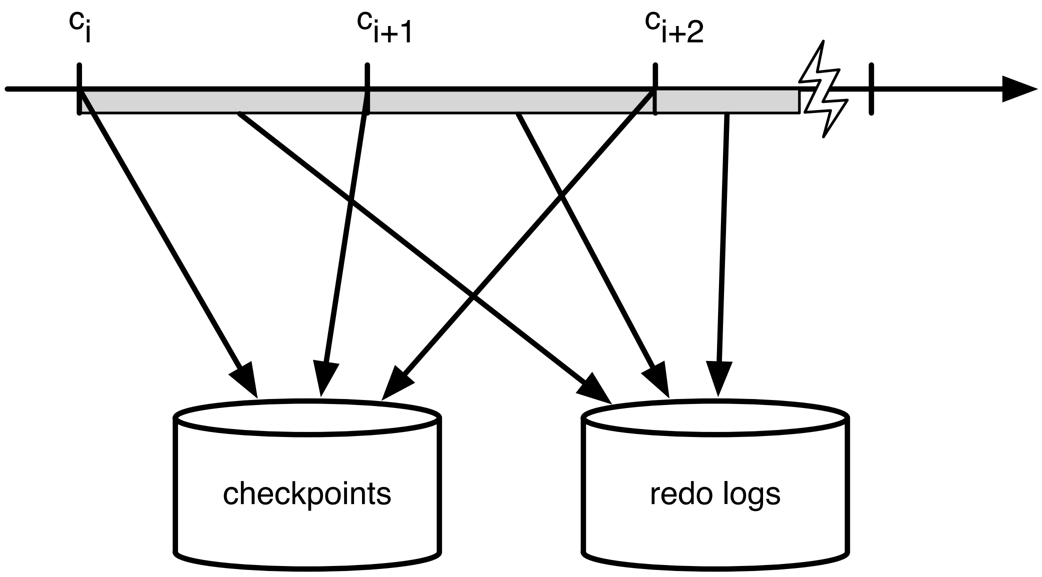


Figure 3.3: Checkpointing

Obviously, a drawback of checkpointing is the requirement for storage for checkpoints and logs. This involves two problems. The first problem is the size of storage. The size of storage increases by the number of servers and by the amount of checkpoints. The more checkpoints are taken, the more efficient it is, in particular, for long running servers. However, it is difficult to determine the number of latest checkpoints to be stored. The second problem is the overhead of runtime performance. To form a checkpoint, the supervisor suspends a server, forms a checkpoint, and resumes the server. In addition, the supervisor also intervenes in the communications of a server to generate redo logs. The frequency of checkpoints must be considered in terms of the overhead and the size of storage.

There are many fault tolerant technologies that adopt checkpointing. Many of the technologies try to recover failed applications. There are only several technologies for operating systems, as in Chapter 2.

Timing of Checkpoint-Restart

In terms of concept, the pre-failure checkpoint-restart works well. In this case, a quiescent state is as the same as a checkpoint. Since the recovery procedure forms a checkpoint then restarts and resume the server at the checkpoint, there is no gap between the checkpoint and the server's latest state. However, there's a problem from a technical viewpoint. Depending on the method of checkpointing, errors that are recoverable by the proactive recovery might be unrecovered. Several checkpointing mechanisms preserve a whole process image (Laadan and Nieh, 2007). In this case, the errors such as memory leaks are preserved in the checkpoint, too. To avoid this problem, a checkpointing mechanism choose significant states in a server.

Checkpoint-restart on failure is a general solution. The supervisor restarts a failed server at the latest safe checkpoint. Some checkpoint-restart mechanisms replay the messages that are sent from outside after the last checkpoint but before the server crashes, in order to reproduce the latest state (Lowell, 1999).

Checkpoint-restart in the delayed recovery reproduces the same failure as long as the resuming point is captured after a failure. However, a server might survive if the resuming point is formed before a failure. In this case, the server catches up the latest state with the redo logs.

3.4.3 Immediate Restart

The other type of restart is *immediate restart*. It is in the middle of the complete restart and the checkpointing. It is as light-weight as the complete restart, and as quick as checkpointing.

Immediate restart takes only the last state of a server including the last message the server received. In the recovery process, the supervisor restarts a failed server. It is however the server's responsibility to restore the latest state after the restart. Upon a completion of the restoration, the server reprocess the last message.

For immediate restart, it is important to identify significant and non-significant state of a server. The difficulty of discriminating between significant and non-significant data is what makes recovery of propagation errors difficult. If not enough state is dropped, the corruption of the server's state may survive the recovery; on the other hand, if too much state is reinitialized, complete recovery is not possible and the behavior of the recovered server may differ from its normal operation. As the supervisor cannot decide what differentiates useful from futile state, the immediate restart technique requires the programmer to directly specify which of its variables should survive a recovery. In general however, the following rules are applied.

Significant state The state that should be preserved is classified into the following categories. The significant state tends to be held across requests.

- Hardware configuration and status. Some of them are changed from the default values while a device driver (a kind of servers) is interacting with its clients. Some configuration or status registers hold values across server restart, however they should be reset to ensure the registers are in error-free state. Since the device driver cannot reproduce them by itself, the values should be held in persistent memory.

- Client status. A server involves client status to continue the operations across requests. The client status the server holds is created in the interactions with a client. To reproduce the client status, the server and the client must redo the same interactions, and this breaks the transparency requirement. A server is required to save the client status in the persistent memory.

- Other unreproduceable data such as the result of the calculation that takes very long time.

Non-significant state State that is local to the processing of a request and is not significant as long as a side-effect that depends on it has not been performed by the server. All the input data of a request is contained into the incoming message, that is replayed once the server is restarted. Therefore, it is safe and desirable to discard any local data created during this process, as it is likely that the corruption occurred there.

Timing of Restart

Immediate restart holds the same as the checkpointing. The significant state may be contaminated by error. Thus, immediate restart in the reactive or the delayed recovery may fail. While the has alternative preserved state to handle that situation, immediate recovery has only the latest significant state.

3.4.4 Summary of the Quantity of State for Consistent Recovery

This section has classified the restart recovery techniques in three types in terms the quantity of preserved state (Figure 3.4). Complete restart hold no state to be preserved. Hence, complete restart requires servers to reproduce the whole state or to be stateless. Checkpointing restarts a failed server with its previous state. While servers don't have to be aware of their state, they are required to determine the frequency of checkpoints. The frequency of checkpoints affects to storage consumption and performance overhead of checkpoints. Immediate restart offers an efficient recovery mechanism with the persistent memory. Immediate restart is a technique place in the middle of complete restart and checkpointing. A server may involve state that is necessary across restart, however, it is required to specify a recovery procedure for the state.

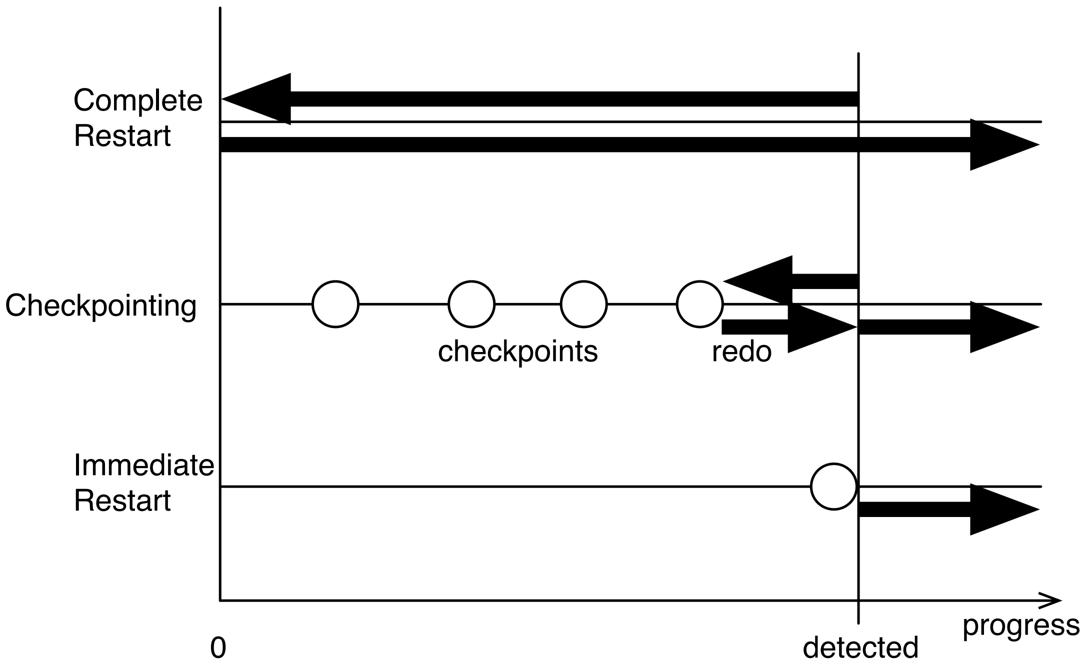


Figure 3.4: Comparison of restart recovery methods

3.5 Summary

Proactive recovery can be applied to a failed server with any type of restart. This is because the type of errors proactive recovery addresses affects late to a server. Thus, a server can reach a quiescent state, where the supervisor can safely restart the server, in a high possibility. Similar to proactive recovery, it is possible to apply reactive recovery to a failed server with any type of restart. On the contrary to the other two recovery methods, we can apply delayed recovery to limited cases. First, we need to ensure that preserved state is not contaminated by errors. Second, we need to ensure that a failed server can reach a quiescent state with effective errors.

The next chapter describes design and implementation of the Arc operating system, a failure recovery framework based on immediate restart and a prototype operating system under the framework.

Chapter 4

Design and Implementation of the Arc Operating System

This chapter describes the design and the implementation of the Arc operating system. Arc is a prototype operating system that aims at utilizing restart recovery for every operating system components in order to build a robust and reliable system. This chapter first explores Arc, which is a multi-server operating system built on top of L4 microkernel. The next section describes the assumptions of Arc failure recovery technologies, and the design of the recovery framework and machinery. Lastly, the chapter describes the implementation issues.

4.1 Introduction to Microkernels

The following sections explain why we adopted multi-server architecture with a microkernel comparing with monolithic kernel architecture, introduce the L4 microkernel on which Arc servers are built, and then describe some details about Arc.

4.1.1 Microkernel versus Monolithic Kernel

Arc adopts L4 microkernel as its basis. A microkernel brings a number of benefits in terms of reliability such as strong isolation of operating system subsystems or message passing mechanisms. Microkernels are often compared to *monolithic kernels*. This section tries to understand microkernels with a discussion of the difference between microkernels and monolithic kernels.

Traditionally, an operating system kernel involves thread scheduling, memory management, protection domains, device drivers, file systems, network protocol stacks, etc. In other words, these subsystems run inside the kernel to allow multiple users to share the underlying resources (e.g. processors, memory, and peripheral devices). We call this type of kernels *monolithic kernels*. A monolithic kernel such as Linux kernel offers reliability at some degree to prevent a user from accidentally or intentionally occupying the underlying resources, or invading the other users who concurrently run.

On the other hand, *microkernels* have tried to take non-essential subsystems such as memory management, device drivers, file systems, network protocol stacks, etc. out of the kernel. There is little consensus on the definition of microkernels. However, microkernels commonly provide thread scheduling, protection domains, and inter-process communication facilities. As opposed to monolithic kernels, microkernels are safe by design.

Since many subsystems run in non-privileged mode (user mode), one can say the microkernel architecture is more reliable than the monolithic architecture. Suppose a device driver fails. A monolithic kernel suffers from a failure then crashes, because the device driver runs in privileged mode, hence it can corrupt other code of the kernel such as thread schedulers or memory management. On the other hand, a microkernel never crashes because device drivers run in user mode, which means the microkernel is protected by hardware. The other processes hold the same. A process runs in a protection domain under a microkernel.

There are some drawbacks of microkernels against monolithic kernels. However, we think this is a trade-off to system reliability. A system running under a microkernel is a kind of distributed system. Generally speaking, a distributed system is more reliable than a monolithic system because risk is also distributed to subsystems. However, it has a problem of communications between the subsystem. Since a subsystem under a microkernel runs in a user mode protection domain, subsystems communicate via inter-process communication facility (IPC) provided by the microkernel, while subsystems can communicate via normal function calls under a monolithic kernel. Although IPC is costly at some degree mainly because the kernel has to switch protection domains, some efforts have been made in various microkernels (Bershad et al., 1990)(Liedtke, 1993). Despite the efforts, there have been still some issues pointed out, particularly on scalability (Lameter, 2007).

4.1.2 L4 Microkernel

This subsection briefly introduces some basic knowledge on the L4 microkernel (L4). For more detail, the readers would like to refer the L4 reference manual (Universität Karlsruhe, 2005).

L4 has originally been developed by the researchers of the University of Karlsruhe. Variants of L4 have been actively developed by some organizations. The main characteristics of L4 is minimalism and a fast IPC mechanism (Liedtke, 1993)(Liedtke, 1996) which are critical issues of microkernels. A microkernel should be designed as small as possible in order to delegate as many functions to user-level servers as they can. As discussed in Section 4.1.1, this enhances the safety and reliability of the system. A fast IPC mechanism is mandatory for frequent communication between the user-level servers isolated by the microkernel. In particular, the speed of IPC was a critical bottleneck of the Mach operating system, which is the first generation microkernel.

L4 virtualizes the processors and the main memory, and provides less than 20 system calls so that user tasks operate the virtualized resources. Some of the system calls are privileged ones. Only privileged tasks are allowed to invoke them. The privileged tasks are fixed, although their implementations are open to developers.

L4 defines two privileged tasks, Sigma0 and the root task. Sigma0 holds all memory pages that are accessible via CPU including physical memory pages. For instance, Sigma0 holds 4GB of memory pages on a certain 32-bit CPU, including some amount of physical memory (RAM). Sigma0 distributes its memory pages to other processes and pager processes.

The root task is the first user process. L4 launches the root task after Sigma0. The root task is allowed to control the life cycle and status of threads and processes.

Inter-Process Communication

L4 provides inter-process communication (IPC) to threads. A thread can synchronously or asynchronously communicate with another thread by issuing an IPC system call (Figure 4.1). L4 IPC is causally ordered. In other words, it keeps a *happens-before* relationship of messages delivered by a thread; a message delivered by a thread before another message delivered to the same destination by the same thread arrives at the destination first. In Figure 4.1, *Call* is synchronous communication, while *Send* is asynchronous communication. *Receive* waits for a message blocking the execution.

L4 maps every notification to an IPC message. That is, interrupts, page faults, or exceptions are delivered as IPC messages from the kernel to a thread that is registered as the receiver in advance. If no threads are interested in a notification, the notification is discarded.

Memory Mapping

Another important service provided by L4 is memory mapping. Memory mapping is used to assign some physical memory pages to an address space, or make some pages in an address space to be shared with another address space.

Mapping operation is an IPC message, while unmapping operation is a system call. A source thread, which belongs to the source address space where some pages are mapped from, delivers an IPC message for memory mapping to a destination thread, which belongs to the destination address space where the pages are mapped to. Mapping operation allows not only to map some memory pages, but also to give some memory pages to the destination, which is called *grant* operation.

Mapping operations create a hierarchy of memory mappings (Figure 4.2). A lower layer is always allowed to unmap its pages or change their status, pages which are mapped

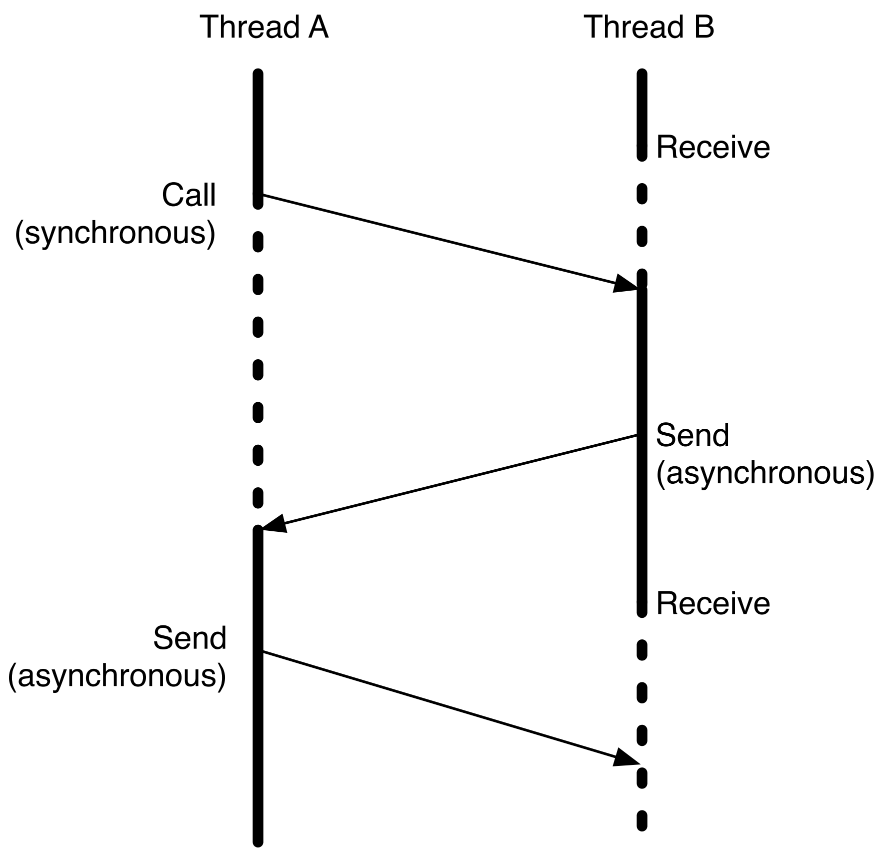


Figure 4.1: Examples of IPC

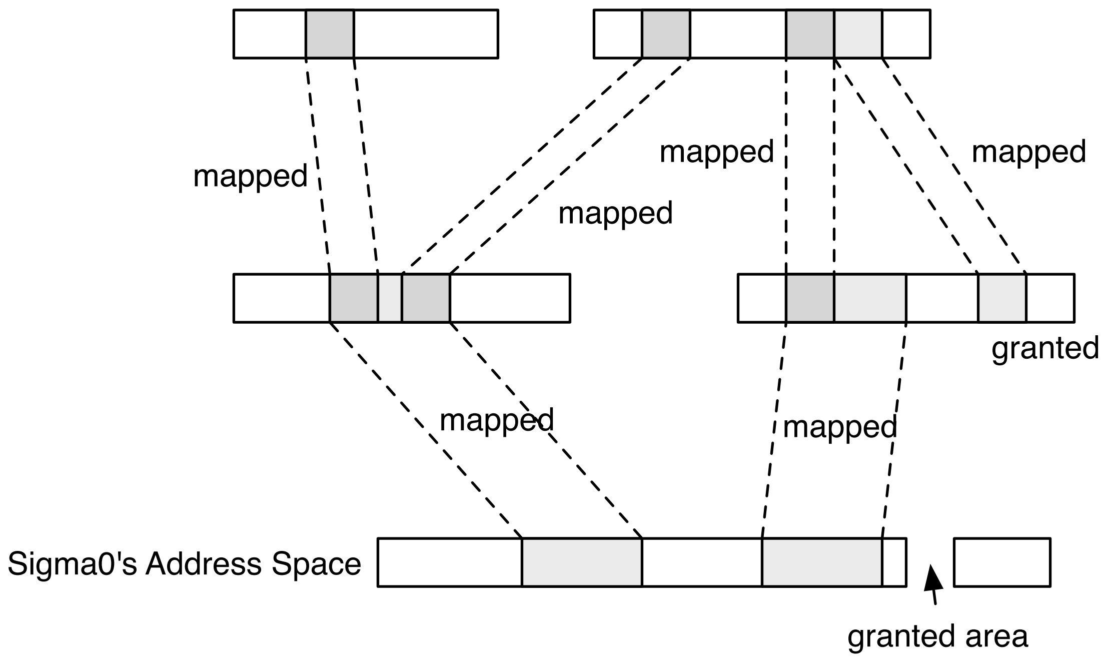


Figure 4.2: Memory mapping hierarchy

to other address spaces.

Sigma0

Sigma0 is a special server task that holds the entire physical address space including the main memory; Sigma0 is the root of memory mapping. The address space is divided into pages, which are usually 4096 bytes large. Sigma0 gives or rents memory pages to the other processes including the root task, by means of memory mapping protocol.

Sigma0 of Arc simply maps all addresses to the same addresses of the root task on demand due to the shortcoming of Sigma0 (see Section 4.2.2). Thus, it doesn't manage the memory.

4.2 Architecture of Arc

This section describes the architecture of Arc and the functions of main components. As mentioned before, Arc is a *multi-server* operating system, which consists of a microkernel and a collection of user-level servers (Golub et al., 1990).

4.2.1 Decoupling the Operating System Functionality

The design issue of Arc is fault containment and independence of subsystems. Major operating systems such Linux and Microsoft Windows crash because they are monolithic; an error can easily contaminate the kernel. Even though an error stays in a subsystem,

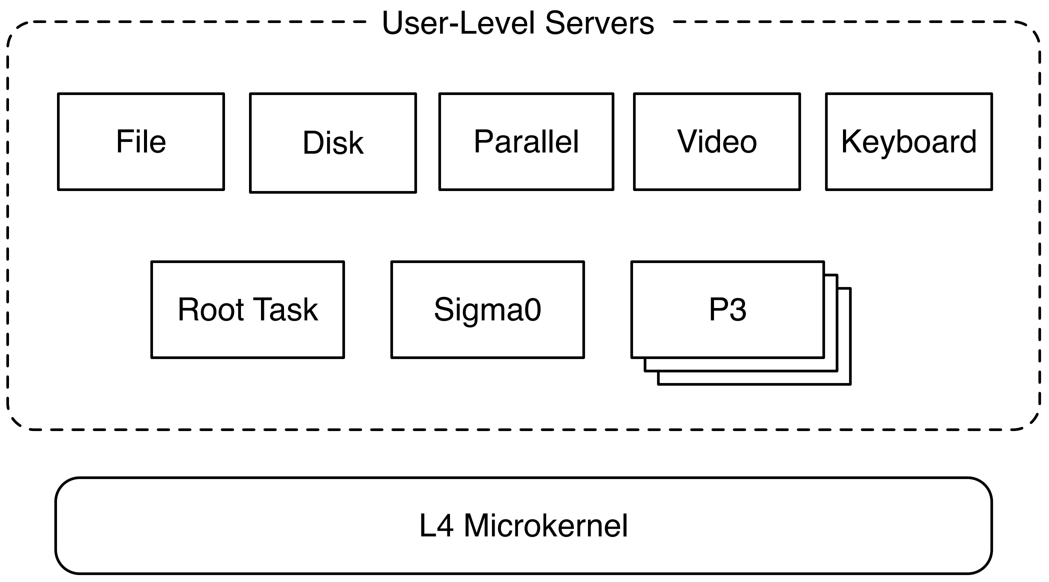


Figure 4.3: The structure of Arc

restart recovery does not fit to a monolithic kernel because a subsystem is accessed via regular function calls; a kernel thread enters the subsystem, hence the thread context is lost by a restart of the subsystem. As we discussed so far, we use microkernel architecture to decouple the operating system functionality such as file systems and device drivers from the kernel and put them into individual processes. Since every process on a microkernel communicates with another process via a message passing mechanism that the kernel provides, we can keep a restart mechanism simple.

4.2.2 Multiserver

In a multi-server operating system, the kernel or each server plays a different role. For instance, some servers are allowed to invoke privileged system calls to the kernel, but others are not. Some servers dedicate to control peripherals, others deal with virtual resources. The kernel and some servers provide a service to users by interacting with each other. The structure of Arc is depicted in Figure 4.3.

Single Pager Style versus Multiple Pager Style

There is an important notion about memory mapping, called *pagers*. A pager is a task that is responsible for memory mapping upon a page fault. Every thread is associated with a pager.

An address space at its initial state is invalid because any physical pages are not mapped yet. An access to such a region which is not mapped generates a *page fault*. It is a pager that receives a page fault. When a pager receives a page fault as an IPC message, it finds a valid page and map it to the requested address. If the page fault request itself is determined invalid, the task that generated the page fault may destroyed.

Arc adopts the multiple pager style. Single pager style and multiple pager style are brought by the L4 memory model. In the single pager style, a central pager is responsible for page fault handling of all address spaces. This facilitates memory management. Mapping database stores the record of memory mapping such as destination thread identifier, destination address, the current permission of a physical page. In addition, the scalability of the pager is poor. In the multiple pager style, mapping database can be distributed to pagers. Since a pager dedicates to a small set of tasks, scalability is better than the single pager style. However, page sharing is more complicated.

Arc Root Task (ART)

Arc Root Task (ART) is an independent task that is responsible for thread and address space management. Only ART can create or destroy threads and address spaces.

In terms of safety of memory management, ART holds all the physical pages. The reason comes from a shortcoming of Sigma0. Sigma0 can map or grant memory pages, but cannot revoke the memory pages mapped or granted. Suppose there is a user task that holds some memory pages granted. If the user task doesn't use the memory pages at all, it is waste of memory. If they were mapped to another task that required more memory, it would be efficient in terms of memory usage. As such, the shortcoming of Sigma0 may lead to inefficiency of memory usage including memory leak. As opposed to Sigma0, the L4 root task can revoke the pages mapped (not granted).

ART involves a memory mapping database. The memory mapping database includes the physical addresses, destination addresses, current owners and permissions, and state. The memory mapping database is updated when a page fault, a page mapping/unmapping request, or a page reservation request is processed, or a user process is created or terminated. Since only the dedicated thread can access the memory mapping database, no synchronization mechanism is required.

The life cycle of threads is also under the control of ART. Only ART is allowed to create or destroy threads. Therefore, when a thread in a process wants to create/destroy another thread inside it, it has to ask ART to create/destroy it by sending an IPC message.

The other privileged operation of ART is the configuration of interrupts. Interrupts are delivered as IPC messages under L4. A thread that is interested in an interrupt needs to associate its thread identifier (TID) to the interrupt request number (IRQ). Only privileged tasks such as ART are allowed to perform this TID-IRQ association.

Per-Process Pager (P3)

Per-process pager (P3) is a type of pager, which is responsible for page mapping upon a page fault. P3 realizes the multiple pager style; each process is associated with a different

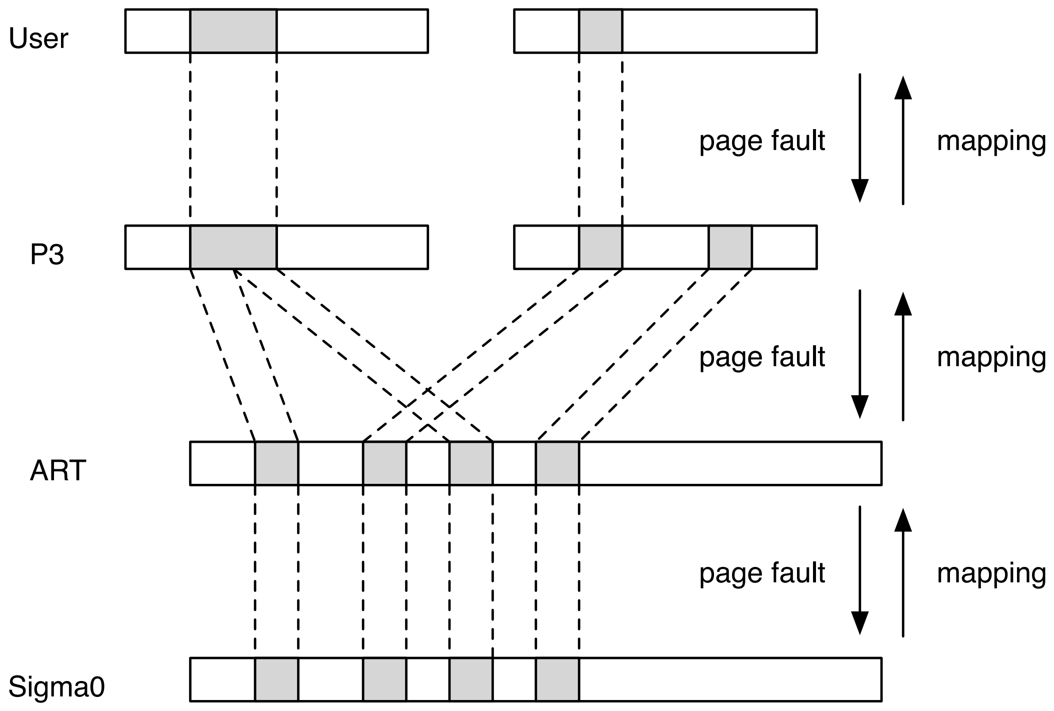


Figure 4.4: The root pager, per-process pagers and user processes

P3 (Figure 4.4). Since the P3 architecture does not allow user processes to directly (i.e. no P3 intervention) map their addresses with each other, the depth of memory mapping hierarchy is always at most four (i.e. Sigma0, ART, P3, and user).

4.2.3 Memory Mapping in Arc

Arc prohibits user tasks to directly share any pages in order to avoid inconsistency in memory mapping hierarchy. As depicted in Figure 4.5, user-level direct mapping enables direct mapping between the user process and ART. In the memory mapping model of Arc, memory for user process is allocated through P3, and this allows to offer persistency of the user's data. If a server that maps its own address directly to another process crashes, the mapping route is changed according to L4 memory mapping rule. Since the crashed user process forgets the address that was directly mapped, it is impossible for the user process to reproduce the same mapping.

Page sharing involves P3. As depicted in Figure 4.6, P3 of process P and P3 of process Q share a page so that process P and Q share the page. Page sharing is processed as follows: A requester (process Q) delivers an IPC message to a source (process P). The source marks a page as sharable, then replies to the requester. The requester maps the sharable page to its address space.

Unlike user-level direct mapping, memory sharing through P3 can reproduce the mapping hierarchy. P3's address space includes the same pages of the corresponding user

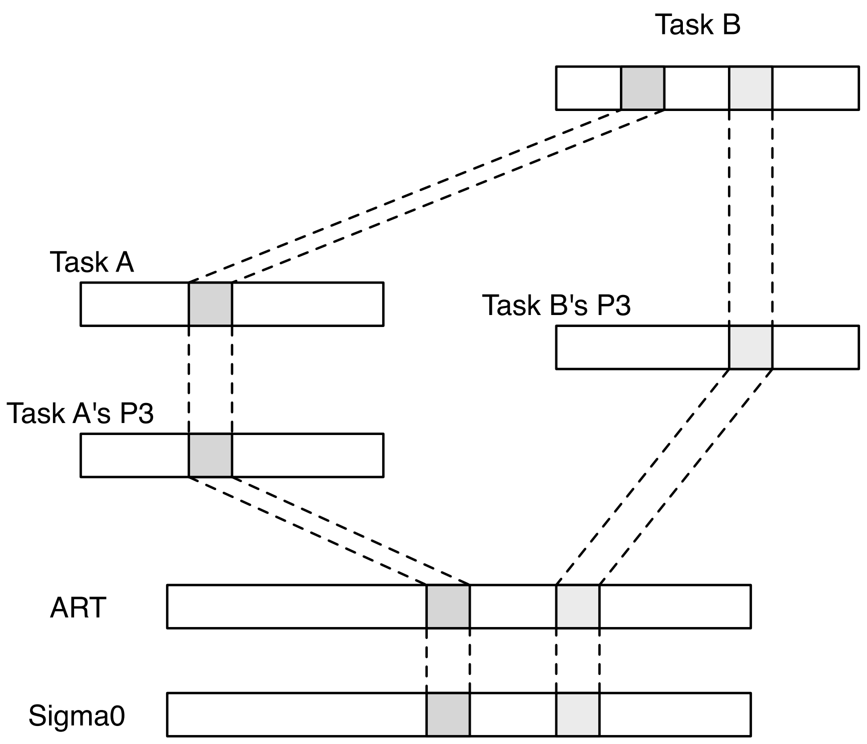


Figure 4.5: User-level direct memory mapping

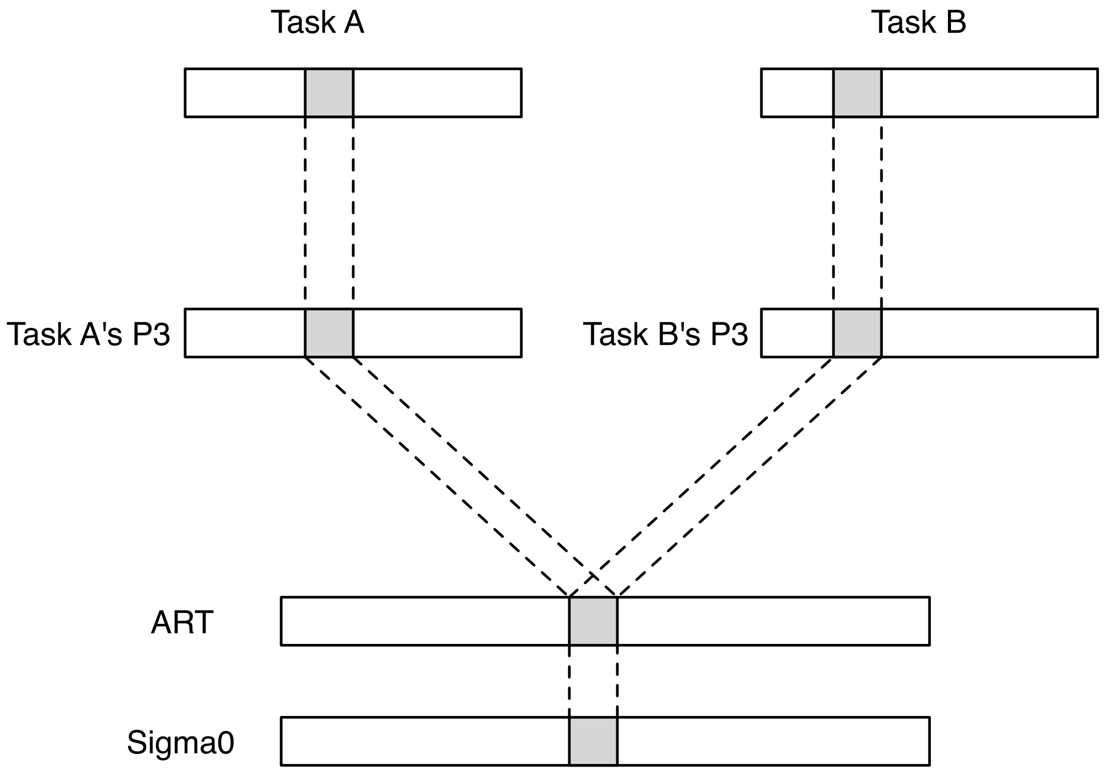


Figure 4.6: Page sharing via P3

address space at the same addresses. Even if the user process fails, the mapping tree itself is consistent. The drawback of this method is complexity of mapping procedure. In the procedure, the mapper process, which offers the sharing of memory, marks the address as sharable through P3, then gives the address to the mappee process, which receives the sharing of memory. The mappee process maps the given address to its own address.

Memory sharing is implemented by means of the state record of pages and the mapping destination record of pages in the memory mapping database of ART.

4.3 Design of the Restart Framework

The restart framework is one of the services of Arc that provides a way to build fault tolerant servers. The restart framework aims at making an operating system micro-rebootable(Candea et al., 2004). In other words, each server on this framework is able to restart as independently as possible upon failure.

4.3.1 Fault Model

A program fails for various reasons such as accessing uninitialized variables, typo, double-free, race conditions, etc. Some types of faults always cause the failure because they are on *deterministic* execution paths. We call this type of faults *deterministic faults* or *deterministic software bugs*. The other types of faults rarely become effective because they are on *non-deterministic* execution paths which the program execution enters according to non-deterministic events such as user inputs or scheduling policies. We call the latter type of faults *non-deterministic faults* or *non-deterministic software bugs*. Errors and failures caused by non-deterministic faults are called non-deterministic errors and non-deterministic failures, respectively.

Arc attempts to recover components that failed due to non-deterministic faults. Since deterministic faults always become effective, the Arc restart mechanism can not deal with them. On the other hand, non-deterministic faults hardly become effective again because they are related to environmental factor such as timing of execution or input data. Based on this fact, Arc components are restarted upon failure.

Either deterministic or non-deterministic software bugs are a logical fault because software is a logical artifact. *Transient bugs* or *Heisen-bugs*(Gray, 1985) are referred to bugs that are impossible to regenerate again or can't be observed when redoing the previously failed sequence of operations again.

4.3.2 Failure Mode

As described in Section 2.1.1 , we consider crash, hang, and panic. This totally depends on the ability of an error detector. If there is a light-weight byzantine failure detector, Arc is able to handle byzantine failure. Since the detectors is another research field and there is a lot of research projects going on, this research does not address it. On the other hand, crashes are easily detectable on the current IA32 platforms by means of MMU functionality, and hangs can be detected by means of operation timeouts. Panics are obviously detectable because a failed server issues them by itself.

Proactive recovery based on system's behavior is difficult under the current version of Arc, because it basically depends on error detectors. However, it is possible by periodic execution of proactive recovery, by which each server is forcefully restarted in a predefined period.

4.3.3 Recovery Model

Recovery proceeds in the way described as follows:

1. Detect an error, or observe a failure,
2. Restart the failed component, and
3. Integrate the restarted component

At the detection stage, the recovery platform finds the anomaly of servers. The detection would be performed by means of facilities provided by CPU and peripherals such as CPU exceptions, watchdog timers and MMU page fault, or software sensors such as timeout and statistical analysis.

When a failure occurs, the failed server is recovered by restart. The recovery platform forcefully terminates the failed server and then starts it over at its entry point. Since the initial state of a server might be inconsistent to the state of its clients, the integration phase takes care of the consistency so that the restarted server causes another error in a client.

4.3.4 Design Issues

There are two design issues on the restart framework. One of the issues is the method to manage the state of a process across restart. There is a trade-off. In order to keep the consistency of the system's state, the state of a process should be preserved across restart, however, the state should be discarded to bring it to error-free state. The other is usability of the framework. As discussed so far, complexity is one of the cause of non-deterministic errors. The framework should affect to the complexity of servers as little as possible.

State Management

Restarting a process normally discards its execution state. This is unacceptable in many cases in operating system subsystems. For instance, a server holds the state of peripheral as data and a restart may make this state inconsistent with the state of the peripheral.

That means the server is unable to work properly anymore and the operating system cannot provide any services in the worst case.

The restart framework is required to manage and preserve the execution state across restart so that a server or the entire operating system can consistently continue its operations. The restart framework solves this issue with the light-weight persistent memory.

Usability

A framework governs the way to develop software. Hence, its usability is an important factor. In terms of reliability, the framework should not make programming complicated because it causes the source of software faults (software bugs). However, the framework is required to integrate failure recovery into the development. Many of the existing failure recovery technologies have attempted to hide the recovery procedure from targets. As discussed in Chapter 3, this is costly in terms of computational resources. Combining the failure recovery procedure into a development process, a system can efficiently recover from a failure. The restart framework needs to solve the problem that seems compromising.

4.3.5 Event-Driven Programming versus Thread Programming

Both event-driven programming and thread programming are programming styles of servers (Ousterhout, 1996). A server is a system that receives requests from its clients, processes them, and returns the result to the clients. The two programming styles differs in how to deal with the requests from multiple clients.

Event-driven programming organizes a server as an event loop and event handlers (Figure 4.7). The event loop receives a request from a client, invokes the event handler that corresponds to the message, waits for the handler to finish processing the message, and replies to the request. Event handlers are normally registered to the event loop. A message that no event handler is interested in is never processed. An event handler is non-preemptive; hence, no synchronization mechanism is necessary for shared resources among handlers. On the other hand, event handlers are supposed to be short-lived. Otherwise, the response of a server will be slow.

A server under thread programming style can concurrently process requests. A thread is assigned to a message or a session. Unlike event handlers that dedicate to the type of messages, threads can process any types of messages. Thanks to concurrency, a received message is immediately processed no matter if the previous message is under processing. In addition, thread programming style is efficient in case an I/O bound request is being processed. However, exclusive access to shared resources has to be greatly taken into account. Synchronization among threads makes a system complicated (Ousterhout, 1996).

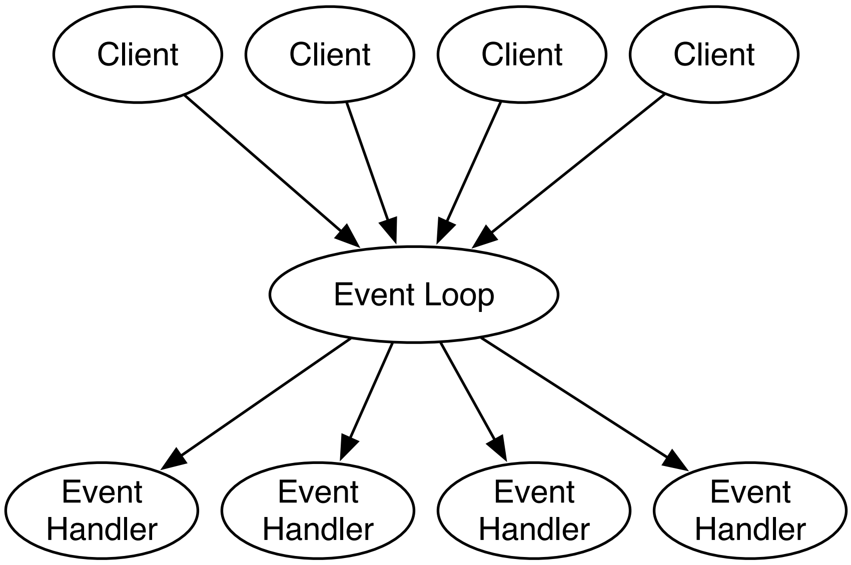


Figure 4.7: Event-driven programming

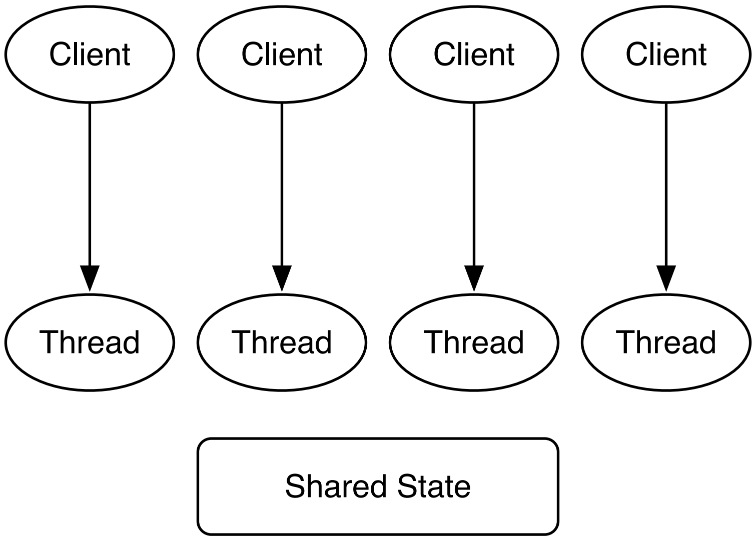


Figure 4.8: Thread programming

The restart framework supports event-handler programming style by default. The consistency of server state is critical for safe recovery. In terms of state management, a server under event-handler programming style is much simpler than a thread-based one because it is single-threaded, thus easy to capture a consistent state. However, in thread programming style, each thread has different execution state and some state are shared among threads and this may lead to inconsistency upon recovery.

4.3.6 Persistent Memory

The restart framework offers persistent memory to every server. Persistent memory is a special type of memory that is able to hold data across server restart. A server stores significant state that is impossible or difficult to reproduce for keeping state consistency with its clients and some other servers.

In prior research, explicit or implicit programming interface or language-based interface is used to access persistent memory (Baker and Sullivan, 1992)(Satyanarayanan et al., 1994)(Chen et al., 1996). Explicit programming interface commonly includes the functions for allocating and releasing a persistent memory area. While Baker and Sullivan proposed explicit programming interface in (Baker and Sullivan, 1992) to store and load data on the persistent memory, Satyanarayanan et al. allowed a program to implicitly access to the persistent memory in a region where transactional processing is guaranteed (Satyanarayanan et al., 1994).

The persistent memory offered by Arc requires explicit declaration of data persistency but no programming interface to access the persistent memory region. The set of programming interface is minimal, so that the implementation of the restart framework can be simplified. The determination of persistency of data depends on how significant the data is.

4.3.7 Independent Recovery Procedure

An Arc server includes an initialization and a recovery procedures. Recovery procedure is different from initialization procedure for Arc servers that involve non-volatile execution state. The non-volatile execution state must be reproduced to repair the consistency with clients or the resources under the server's management.

Crash-only software is another method of server construction that has a server be stateless so that the server can crash anytime and recover with no restoration(Candea and Fox, 2003). However, crash-only requires a trade off. It requires servers to store their state to the external storage so that they can be stateless. This leads to performance loss. In addition, crash-only method cannot handle the hardware state as it is. In the most extreme case, a device server has to copy the state of the hardware to the external storage every time the state of the hardware changes. Indeed, the philosophy of crash-only software came from Internet servers which are integrated with databases and do not directly control hardware peripherals. In terms of operating system implementations, Nooks can be seen as an implementation of crash-only software(Candea et al., 2004)(Swift et al., 2005). Nooks cannot keep consistency between drivers and applications. Although Nooks achieved to recover the kernel, many applications crash during recovery. This indicates that it is not easy to apply crash-only to operating system subsystems.

The restart framework forces each server to implement not only the initialization procedure, but also the recovery procedure. While the initialization procedure is invoked once at the first activation of the server, the recovery procedure is invoked when it is restarted due to failure recovery. The execution environment and conditions of the recovery procedure are the same as that of the initialization procedure. Separating the recovery procedure from the initialization procedure protects the recovery procedure from being contaminated by inconsistencies.

4.4 Design of Restart Mechanisms

The framework defines only the structure of the Arc servers. The restart mechanism actually recovers the failed servers. This section describes the mechanism of Arc restart recovery. The restart mechanism involves multiple levels of restart. The levels of restart are applied recursively from finer to coarser. There is a trade-off; the escalation of granularity looses transparency. This section focuses on each of the mechanisms. The next section discusses the escalation of the granularity of restart in detail.

4.4.1 Process Restart

This section considers restarting a single process. Process restart normally destroys all the contents of the process including threads, IPC state, stack etc. Thus, persistent memory is used to preserve some state across restart. The supervisor then recreates the process. There are several ways to recreate a process. We consider the ways in terms of urgency.

Design Issues

The design issues of process restart is transparency of restart recovery. In other words, clients should never know that a server restarts upon a failure. This design issue is divided into sub issues as follows:

- Preserving the communication state between a server and its clients
- Preserving the memory sharing state between a server and its clients

An IPC has state such as sending, sent, receiving, and received. A client can detect that a server does not exist in the sending or receiving phase. A server becomes unavailable

Table 4.1: Temperature of Restart

Temperature	**Process recreation**	**Recovery procedure**	**Memory persistency**
Hot		√	√
Warm	√	√	√
Cold	√		

by its elimination and creation for recovery. If a client issues an IPC, the server's failure will be exposed. In order to hide the down time of the server, the recovery platform keep the communication state.

Restart of failed servers has a problem on shared memory pages. To exchange a large amount of data, a server and its client may establish shared memory pages. Since a simple restart involves deletion of an address space, the shared pages need to be restored after the restart so that clients are unaware of a restart of the server.

Temperature of Restart

The solution for both issues is to keep the failed process alive. This is because IPC messages that already arrived at the failed process and are waiting for being processed are discarded due to the current specification of L4. Accordingly, the sender of IPC messages will receive error messages of IPC operations. Similarly, the memory sharing state will be discarded when a process is destroyed. The recovery platform needs to avoid destroying a failed process for transparency.

Hot restart is a method of restart recovery that keeps a failed process alive. Hot restart has a message handler of a server reprocess the last message. Since a failed server might hold a persistent memory region, the recovery platform restarts the server from its recovery procedure so that the server can reconstruct the state by means of the persistent memory. Then, the server reprocess the last message after the recovery procedure. Cooperating with the persistent memory, a server can also keep the shared pages.

There are two more options for process restart. *Warm process restart* recreates a failed process preserving the persistent memory. Unlike hot restart, the failed process is once deleted and then created with fresh program image. The recovery procedure is invoked to rebuild the process's state.

Cold process restart destroys an entire process including the persistent memory and recreates the process. It is impossible to save the persistent memory from any contamination. Cold process restart is used to remove faults in the persistent memory. However, since cold process restart discards the state of clients in the server, group restart described below must be considered for failure transparency.

Table 4.1 simply compares the temperature of restart.

4.4.2 Group Restart

The next granularity of restart is *group restart*. Although this type of restart already looses the failure transparency at process level, it is necessary to avoid system reboot. For example, a cold restart of a disk driver requires a file server, which is a client of the disk driver, to perform a hot or warm restart to re-establish their communications. The target of group restart is a collection of processes that are dynamically organized based on client-server relationships. A restart group is organized at runtime by the recovery platform. The processes in a restart group are restarted at the same time, but are allowed to take different temperature of restart.

Design Issues

The rule to form a restart group is necessary. The concept of groups includes a whole system. However, a group restart is required to organize the smallest set of processes that is sufficient for recovery. The following sections discuss the design issues.

Restart Propagation

If a recovery of server failed, the P3 of the server delivers an error message to its clients on behalf of it. Normally, the clients deal with the error message. However, if the clients are in a group, they are forcefully restarted by the P3s of the clients. Therefore, there must be an agreement on the force-restart between a client and a server. In Arc, an agreement is formed by a client implementing the recovery framework. The recovery framework has processes assume that a recovery is performed at anytime.

Dynamic organization of a restart group is recognized by the recovery platform. With the help of naming services, the recovery platform understands the communication network of processes.

The order of restart of the group members follows the age of process so that the root of restart propagation is recovered first. The age of a server is incremented when a client connects to the server.

Escalation and Propagation

A process have three choices of restart such as handler restart, hot restart, and cold restart. In addition, it has another choice, called group restart. A system can combine these restart strategies flexibly in order to increase its reliability.

The base strategy is recursive approach; each approach is applied one by one, from hot to cold and from process-level to system-level.

4.4.3 System Reboot

If any finer restart fails, system reboot is performed. The recovery platform issues the reset command of CPU, then the whole system is rebooted. Some data could be salvaged from the main memory after the reboot by the technique proposed by Baker and Sullivan (Baker and Sullivan, 1992), or Chen et al. (Chen et al., 1996), which this research doesn't address.

System reboot looses all the execution state. Therefore, the processes created after the system launched is not automatically recreated. In other words, only the initial processes are launched after the system reboot.

4.5 Implementation

The restart framework is implemented in P3 and a library. The library provides the basic structure of servers and the interface to the persistent memory. P3 implements the restart mechanism and the persistent memory. They are both implemented in C++.

4.5.1 Server Framework

The framework provides a C++ base class, called `ArcServer`, that enforces that all servers provide the mandatory operations. Listing 4.1 shows a simplified version of the declaration of this class.

```
class ArcServer
{
protected:
    virtual status_t Service(const L4_ThreadId_t& tid,
                            const L4_Msg_t& msg,
                            L4_Msg_t& retMsg) = 0;

public:
    virtual status_t Initialize() = 0;
    virtual status_t Exit() = 0;
    virtual status_t Recover();
    status_t MainLoop();
};
```

Listing 4.1: Simplified version of the ArcServer class declaration.

The `Initialize` and `Exit` methods are pure virtual, which enforces the server developer to implement them. `Recover` is virtual but the framework provides a default implementation that just returns an error message to P3, stating that the server cannot be recovered and causing a full-restart of the failed server. Therefore, it is possible and easy to write servers without caring about recovery.

The other public function, `MainLoop`, is already implemented and is called by P3 after the server is started or recovered by `Initialize` or `Recover`. It waits for incoming messages, and passes them to the `Service` method along with the identifier of the sender. It also replays the failed request in case of server recovery.

The `Service` method, although declared as pure virtual, is not to be implemented directly by the server's writer, contrary to the other methods. This method mainly consist of a `switch` statement that checks the label of the incoming message, calls the corresponding handler, and puts the message to return to the client into `retMsg`. As it must deal with error handling and is very repetitive, we wrote a set of macros to construct it using the message/handler paradigm to associate message labels to the corresponding method handler. An example usage of these methods is given by Fig. 4.2.

```
BEGIN_HANDLERS(AudioDriver)
CONNECT_HANDLER(AUDIO_SET_VOL, SetVolume)
CONNECT_HANDLER(AUDIO_FLUSH_BUF, Flush)
...
END_HANDLERS
```

Listing 4.2: Example usage of the message/handler macros.

The message labels are just integers that define the protocol used between the client and the server and may be build using an enumeration. Handler functions must follow the following prototype:

```
status_t HandlerFunction(const L4_Msg_t& msg,
                              L4_Msg_t& retMsg);
```

Listing 4.3: Handler function prototype.

They return the message that is to be sent back to the client in `retMsg`.

This framework provides all the necessary building blocks to write an Arc server. By just being required to write the initialization and finalization methods and to connect messages to their corresponding handlers, the server writer can concentrate on the functional aspect of the server without having to worry about inter-process communication. Once the server is written, he can then take care of the recovery aspect.

Making a server recoverable requires to write the corresponding recovery method, and to mark relevant data to be persistent across failures.

4.5.2 Light-weight Persistent Memory

The persistent memory area is designed to host data that is necessary to recover and must survive a failure. Contrary to what can be expected from a persistent memory, it does not rely on external storage and simply consists of a special part of the main memory[1]. Its features make it particularly suitable for embedded devices:

- Very lightweight: no special management is required for persistent data, which is accessed like any other data in the main memory.
- No data redundancy: persistent data is not a duplicate of existing data, but the actual runtime data mapped to a dedicated range of the address space.
- Easy to use: a single macro to append to a variable declaration makes it persistent.

Implementation of this persistent memory is made through a cooperation of the compiler and linker. The GCC compiler provides support for section attributes that define the section of an object file in which a declared variable should be located. By using this feature, we defined a C macro that can be appended to any declared variable in order to place it into a special *.pdata* section.

```
#define IS_PERSISTENT __attribute__ ((section(".pdata")))
```

At link time, the LD linker is given a special link script that relocates all data in the *.pdata* section together. When P3 loads the server program binary, it detects this *.pdata* section and automatically maps it into a dedicated area of the address space (Figure 4.9).

Each process possesses its own persistent memory area. P3 assigns physical memory pages to the persistent memory area at load time, just as it does with other memory sections. However, upon a failure of the server process, all the physical pages but the persistent memory area are unmapped. This design makes the persistent memory area absolutely free in terms of memory footprint, and virtually free in terms of performance, since only a few trivial operations must be performed by P3 at load and recovery time to support it.

We should note that this memory is not transactional. It is not possible to bring it back to a previous state: this means that if the failure entered the persistent memory area, it will most likely not be recoverable.

[1]Real persistent storage is not necessary here as the memory must only survive server failures, not system restart.

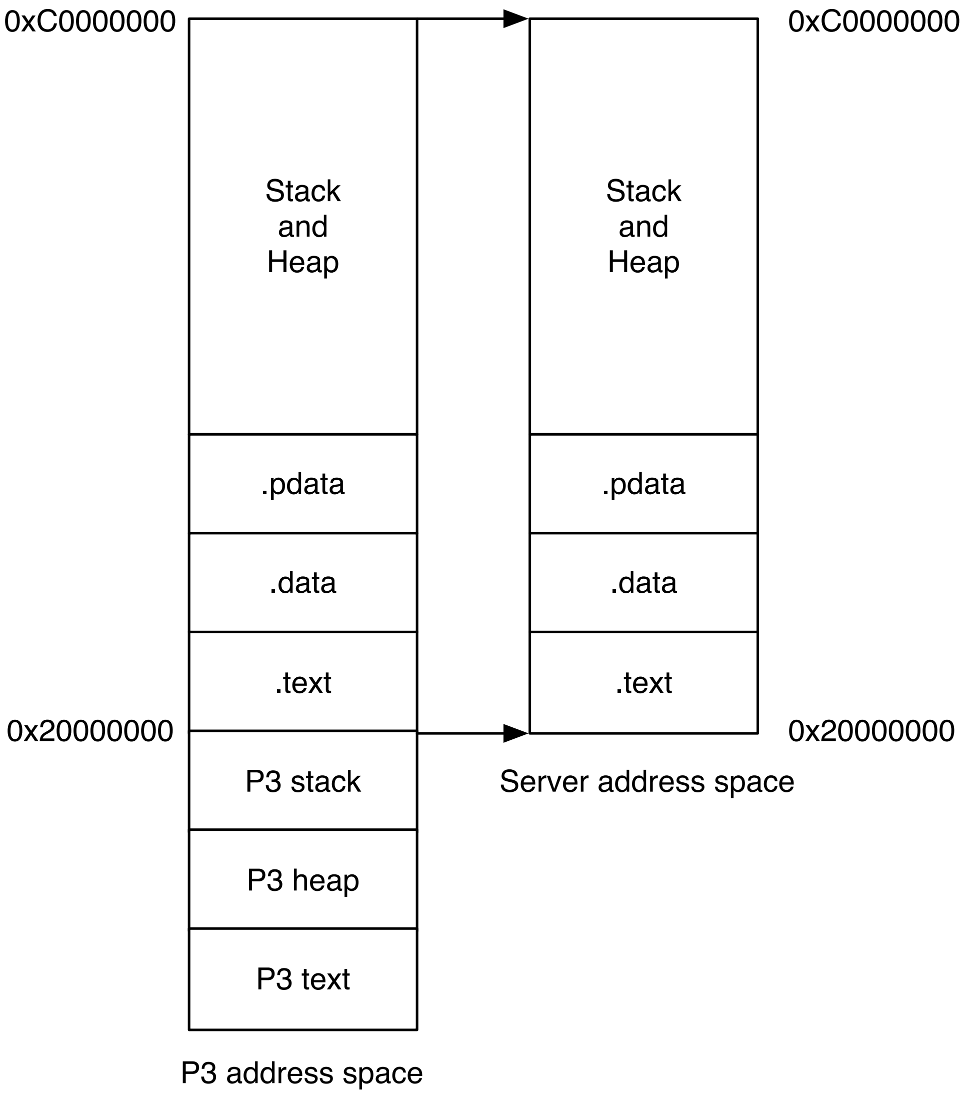


Figure 4.9: Memory layout of Arc servers

4.5.3 Session Library

For more reliable IPC, the restart framework provides the session library to clients, and a base class to servers. The session library provides a C++ class to create shared memory pages between processes. The base class for servers offers several facilities to manage clients. To preserve share memory pages across restart, they are allocated in a dedicated memory region that P3 manages. Listing 4.4 and Listing 4.5 depict the client side session library and the server side session library respectively.

```
class Session
{
public:
    Session(L4_ThreadId_t peer);
    virtual ~Session();
    virtual addr_t GetBaseAddress();
    virtual stat_t Begin(L4_Word_t* send_regs, size_t scount,
                         L4_Word_t* recv_regs, size_t rcount);
    virtual stat_t End(L4_Word_t* send_regs, size_t count);
    virtual stat_t Put(L4_Word_t* send_regs, size_t scount,
                       L4_Word_t* recv_regs, size_t rcount);
    virtual stat_t Get(L4_Word_t* send_regs, size_t scount,
                       L4_Word_t* recv_regs, size_t rcount);
};
```

Listing 4.4: The Session class (client side).

Receiving a request from a client, a server allocates some continuous pages from the dedicated area in the `HandleConnect` method. The server then reserves the pages for the client, and delivers the base address of the shared area back to the client.

The `HandleConnect` method also registers clients to the list `_clients` in order to release the shared pages when they disconnect from the server. Each `SessionControlBlock` object contains the thread ID of a client, and the base address and size of the shared area in the client's address space.

On the client side, the instantiation of a Session object establishes a connection to a server. After sending a request to a server, the constructor of the Session class allocates the pages from the dedicated region of memory, pages which the server has reserved in advance (Figure 4.10). The constructor takes a server's thread ID to find the reserved pages. The number of shared pages allocated in the constructor is one in the current implementation.

The base address of a shared area is obtained through `GetBaseAddress`. The base address in a client's address space is different from the one in a server's address space. The Session class converts the base address when the constructor receives the first reply

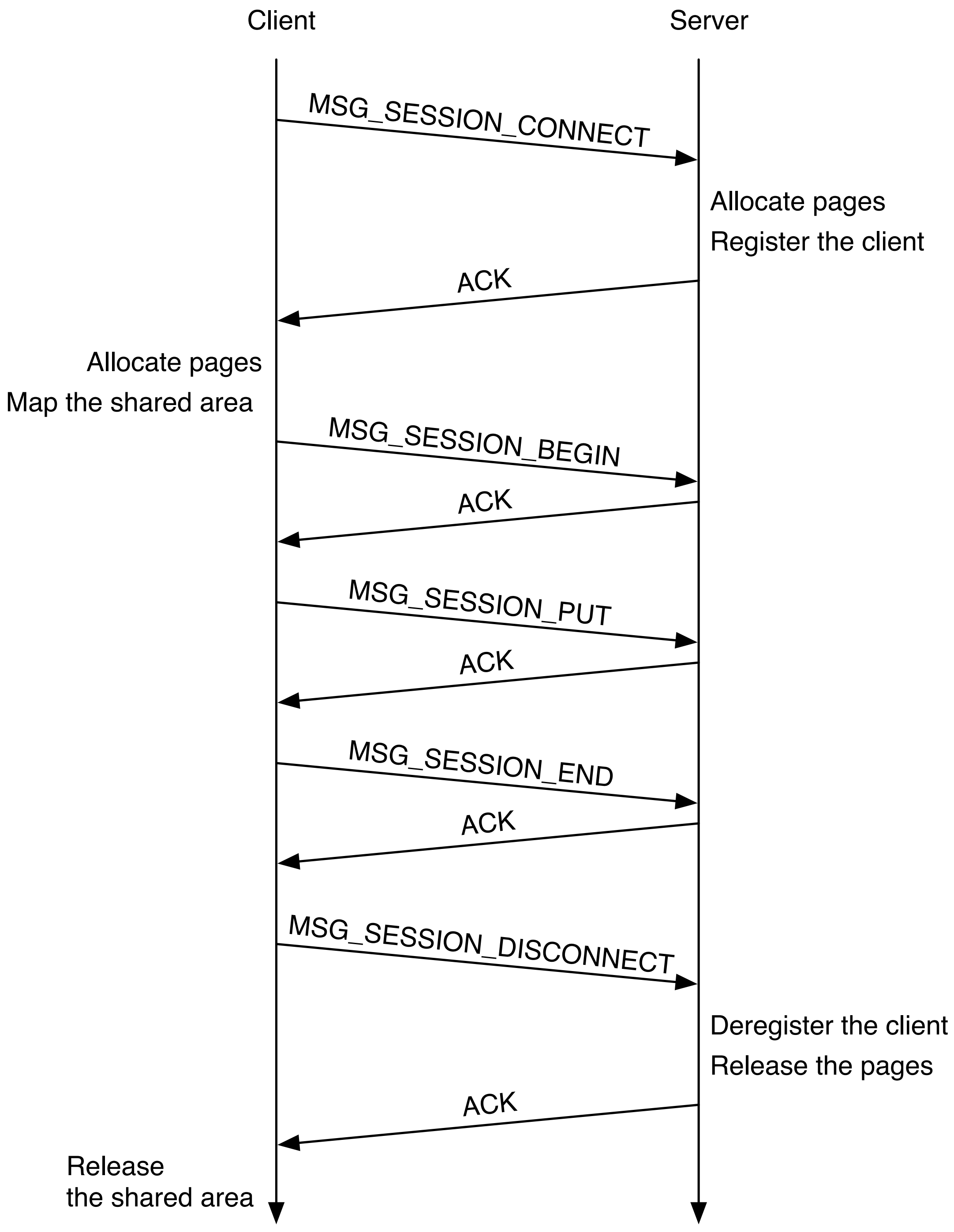


Figure 4.10: Page sharing protocol

```
class SessionServer : public ArcServer
{
protected:
    List<SessionControlBlock*>    _clients;

    virtual void Register(const L4_ThreadId_t& tid,
                          addr_t base, size_t size);
    virtual void Deregister(SessionClient* c);
    SessionClient* Search(const L4_ThreadId_t& tid,
                          addr_t base);

    virtual stat_t IpcHandler(const L4_ThreadId_t& tid,
                              L4_Msg_t& msg);
    virtual stat_t HandleConnect(const L4_ThreadId_t& tid,
                                 L4_Msg_t& msg);
    virtual stat_t HandleDisconnect(const L4_ThreadId_t& tid,
                                    L4_Msg_t& msg);
    virtual stat_t HandleBegin(const L4_ThreadId_t& tid,
                               L4_Msg_t& msg);
    virtual stat_t HandleEnd(const L4_ThreadId_t& tid,
                             L4_Msg_t& msg);
    virtual stat_t HandleGet(const L4_ThreadId_t& tid,
                             L4_Msg_t& msg);
    virtual stat_t HandlePut(const L4_ThreadId_t& tid,
                             L4_Msg_t& msg);
};
```

Listing 4.5: The SessionServer class (server side).

from a server.

A client transfers data to the server through the `Put` method. A client first needs to fill the shared area with data, then issue the `Put` method. The method is synchronous; it is blocked while the server is processing the data in the shared area. A client invokes the `Get` method to obtain data from the server. The `Get` method is also synchronous.

A client invokes the `Begin` and `End` methods to separate a session. For instance, a function that opens a file invokes the `Begin` method to inform a file server that the following `Put` and `Get` operate on the specified file. Some servers ignore `Begin` and `End` because they provide a single session per connection.

The `Begin`, `End`, `Get`, and `Put` methods take two or four arguments. In Listing 4.4, `send_regs` represents a reference to an array which is delivered via the IPC registers, `scount` indicates the size of the array, `recv_regs` represents a reference to an array which is filled with the IPC registers when the client receives a reply from a server, and `rcount` indicates the size the receiving array.

4.6 Performance Evaluation

We evaluate the overhead of P3. P3 redirects all page mapping requests to the root pager, which holds the physical memory pages. We measured the cycle count of handling a page fault and the subsequent page mapping. Page fault handling with P3 takes approximately 1.5 times as many cycles as the one without P3. Since page faults only occur the first time a user process accesses the page, we consider the resulting overhead as not critically harmful to performances. The evaluations have been performed on an Intel Pentium4 3.4GHz with 1GB of main memory.

Total down time of the framework upon a failure has been measured to 0.2 ms. To get this metric, we injected a simple fault that results in a memory access violation, and measured the time from the fault being triggered to the `Recover` method being called. Table 4.2 shows the detail of the down time. The *Unmap* operation unmaps all the pages mapped to the driver process excepted for pages of the persistent memory. *Initialize* re-initializes the counterpart of the process inside P3, which manages the base address and the size of each segment. *Load* reads the binary and extracts it to the main memory. *Start* triggers the driver process, however, the driver doesn't immediately run because none of its pages are currently mapped. P3 thus invokes the IPC handler which deals with page mapping requests. According to the table, program loading takes almost half of the total time, and the remaining time is consumed by page fault handling after the driver process is restarted.

Table 4.2: Measured recovery time

Operation	**Time**	**Rate**
Unmap	28.63us	14.33%
Initialize	0.22us	0.11%
Load	88.46us	44.27%
Start	8.89us	4.45%
Mapping	73.8us	36.9%

To these 0.2 ms of recovery time, must be added the time necessary to invoke the `Recover` method of the driver. The execution time of this method is obviously driver-dependent. For the drivers we evaluated, the parallel driver introduces a forced latency of around 10 ms because it waits the time necessary to ensure the last written data byte (if any) has been acknowledged by the printer. The video driver, on the other hand, just requires to map the hardware framebuffer, an operation that does not take more than a couple of milliseconds.

4.7 Summary

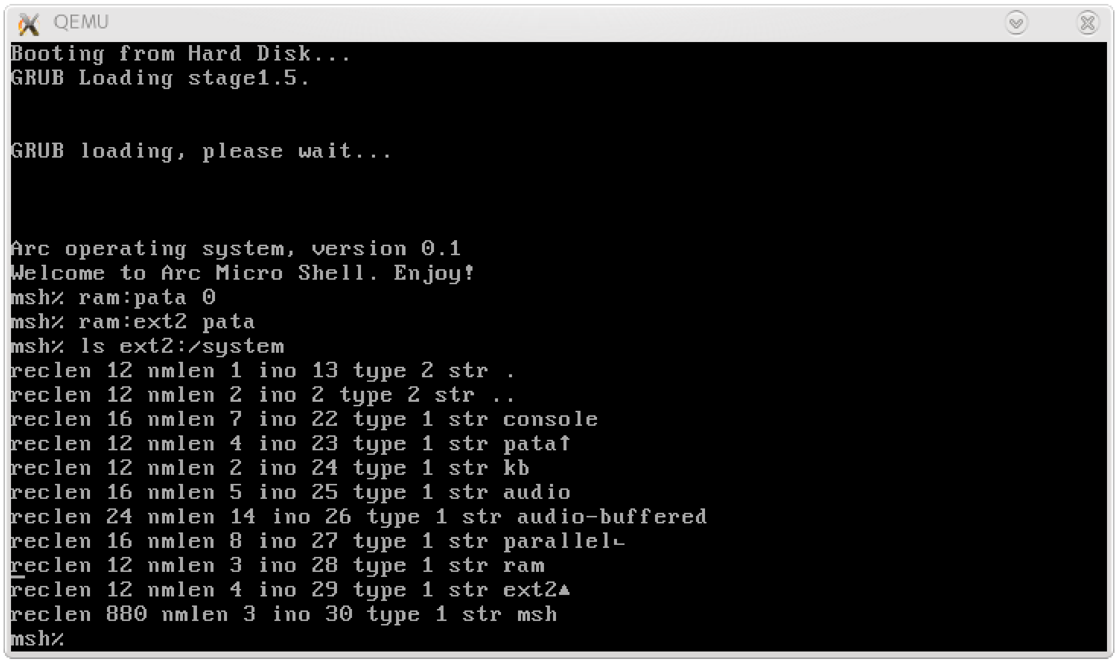
Arc is a multi-server operating system written from scratch on top of L4 microkernel. Since the server processes run in user-mode and isolated from each other, Arc is inherently tolerate to failure of a server process. Dividing memory management into P3s increases the independence of server processes and scalability of the system.

The restart framework provides the way of fault tolerant server construction in addition to the basic fault tolerance provided by L4. Since it is difficult to separate the significant and non-significant state of a program in general, the restart framework provides a dedicated method that is invoked when a server is restarted by the supervisor. The framework expects the developers to overwrite the recovery method for each server implementation.

The restart framework provides the persistent memory which data is preserved across restart. The recovery method uses the persistent memory to restore the state of a server. The persistent memory is implemented in P3. A server uses the persistent memory through C macros.

The framework provides a reliable session library based on the persistent memory. The session library allows two processes to exchange a large amount of data through a shared memory area.

In the next chapter, we will evaluate the feasibility of Arc and the restart framework.



```
Booting from Hard Disk...
GRUB Loading stage1.5.


GRUB loading, please wait...



Arc operating system, version 0.1
Welcome to Arc Micro Shell. Enjoy!
msh% ram:pata 0
msh% ram:ext2 pata
msh% ls ext2:/system
reclen 12 nmlen 1 ino 13 type 2 str .
reclen 12 nmlen 2 ino 2 type 2 str ..
reclen 16 nmlen 7 ino 22 type 1 str console
reclen 12 nmlen 4 ino 23 type 1 str pata↑
reclen 12 nmlen 2 ino 24 type 1 str kb
reclen 16 nmlen 5 ino 25 type 1 str audio
reclen 24 nmlen 14 ino 26 type 1 str audio-buffered
reclen 16 nmlen 8 ino 27 type 1 str parallel∟
reclen 12 nmlen 3 ino 28 type 1 str ram
reclen 12 nmlen 4 ino 29 type 1 str ext2▲
reclen 880 nmlen 3 ino 30 type 1 str msh
msh%
```

Figure 4.11: Screenshot of Arc running on QEMU

Chapter 5

Case Study and Discussion

This chapter evaluates the efficiency of our framework. Since the main point for a recovery mechanism is to effectively recover upon failures, this chapter first illustrates how we managed to recover on four different drivers instances: first, on a character device driver (parallel port driver), a class of devices that are reputed particularly difficult to recover because of the stream nature of the managed hardware. Then, we will consider recovering a VESA video driver and a sound blaster audio driver. This kind of driver must keep enough information about the current state of the managed hardware to ensure a complete and transparent recovery (i.e. without graphical glitches or noticeable delay in display).

5.1 Case Study

The case studies described in this section explores the usage and feasibility of Arc and the restart framework. We mainly developed device drivers on the restart framework, such as a parallel port driver, a VESA VBE graphic driver, a Sound Blaster audio driver and a PATA disk driver.

The case studies shown in this section are conducted on QEMU, an open source processor emulator. Through this experiment, QEMU runs on Linux version 2.6 on Intel Core2 processor 2.66GHz with 2GB RAM and emulates Intel IA32 instruction set. QEMU supports the peripherals used in the experiment, such as VESA VBE graphic device, Sound Blaster audio device, and hard disks based on PATA.

5.1.1 Design Guideline

As described in Chapter3, a developer has to distinguish significant and non-significant states in a server. Chapter3 showed the guideline to distinguish server state that should be preserved in the persistent memory area from the other non-persistent state:

Significant state The state that should be preserved is classified into the following categories. The significant state tends to be held across requests.

- Hardware configuration and status. Some of them are changed from the default values while a device driver (a kind of servers) is interacting with its clients. Some configuration or status registers hold values across server restart, however they should be reset to ensure the registers are in error-free state. Since the device driver cannot reproduce them by itself, the values should be held in persistent memory.
- Client status. A server involves client status to continue the operations across requests. The client status the server holds is created in the interactions with a client. To reproduce the client status, the server and the client must redo the same interactions, and this breaks the transparency requirement. A server is required to save the client status in the persistent memory.
- Other unreproduceable data such as the result of the calculation that takes very long time.

Non-significant state State that is local to the processing of a request and is not significant as long as a side-effect that depends on it has not been performed by the server. All the input data of a request is contained into the incoming message, that is replayed once the server is restarted. Therefore, it is safe and desirable to discard any local data created during this process, as it is likely that the corruption occurred there.

5.1.2 Parallel Port Driver

Our first attempt at recovery has been performed on a very simple SPP parallel port driver. The SPP protocol is an unidirectional protocol designed to send data to a printer(Peacock, 2007). Because of its simplicity, we chose to implement it in order to be able to easily demonstrate how recovery can be managed in this class of drivers.

Character devices drivers are known to be very difficult to recover; character devices deliver or receive data one unit of data at a time, and do not support random access. Hence, if a data is read by the driver and lost in the failure, failure transparency is broken. All the same, if a service fails while sending data to a character device, transparent recovery implies that the same data is not sent twice, and thus that the driver keeps track of its internal state and uses it upon failure. We will show how we implemented this feature using our framework.

Driver Implementation

The driver accepts `WRITE` requests that embed the data to be written to the port. This task is performed by the `writeData` function, which writes an array of bytes to the parallel port and which simplified code can be seen on Listing 5.1.

```
 1  status_t
 2  writeData(const char* const buffer, size_t count) {
 3    Byte control = inb(CONTROLPORT);
 4    for (int i = 0; i < count; i++) {
 5      control &= ~STROBE;
 6      outb(DATAPORT, buffer[i]);
 7      outb(CONTROLPORT, control);
 8      L4_Sleep(SEND_DELAY);
 9      control |= STROBE;
10      outb(CONTROLPORT, control);
11      L4_Sleep(SEND_DELAY);
12      ...
13    }
14    ...
```

Listing 5.1: Simplified version of the `writeData` function.

Writing a byte to the parallel port consists of setting the data port to the value of the byte to be written, and to lower the `STROBE` bit of the control port for at least 5 milliseconds so that the printer is aware that new data is written and readable. After that, `STROBE` is set again for 5 milliseconds before the next byte is written.

When receiving a `WRITE` request, the handler calls this function with the byte array and length of data as parameters. The rest of the driver is very simple: it has no initialization or exit function. Therefore, failures can only happen in the `WRITE` request handler and the `writeData` function, in which case P3 will restart the server and replay the last message received. If the request has not been processed at the time of failure, then recovery is completely transparent.

If the `writeData` function was in the midst of writing data from the buffer, then data may be written twice as the whole request is being reprocessed. Therefore, in order to ensure transparent recovery, we need to instrument the `writeData` function in such a way that does not repeat already sent data.

Recovery Handling

The parallel port driver is a typical example of driver that needs to keep some intra-request data persistent in order to achieve transparent recovery. The stream nature of the parallel port also requires to keep track of the amount of processed data, symbolized by the `i` variable.

This means that `i` must be made persistent. Moreover, we have to be very careful about its time of increment: in the implementation of Listing 5.1, `i` is only incremented at the end of the loop. If a failure occurs between the time `STROBE` is set down (line 9) and the end of the loop, the last character would be written twice when recovery takes place. Therefore, `i` must be updated right after the character is sent to the printer. Listing 5.2 gives the code of the recover-friendly version of this method.

```
static int i IS_PERSISTENT = 0;

status_t writeData(const char* const buffer, size_t count) {
  Byte control = inb(CONTROLPORT);
  if (!Recovered) i = 0;
  while (i < count) {
    control &= ~STROBE;
    outb(DATAPORT, buffer[i]);
    outb(CONTROLPORT, control);
    i++;
    L4_Sleep(SEND_DELAY);
    control |= STROBE;
    outb(CONTROLPORT, control);
    L4_Sleep(SEND_DELAY);
    ...
  }
  ...
```

Listing 5.2: Recovery-friendly version of the `writeData` function.

The first notable thing is that the `i` iterator is declared outside of the function to survive its local scope, and is made persistent. Then, it is reset to its initial (0) value only if the IPC being processed is not the replay of a recovery. Finally, `i` is incremented as soon as `STROBE` is set down, as from this time we can guarantee that the printer will receive the byte being issued if `STROBE` remains down at least 5 milliseconds.

The recovery function of the driver only has to ensure this last condition: it just waits for 5 milliseconds to ensure the current state of the `STROBE` bit is acknowledged, and then sets it high so that the port is ready to send data again.

Results

We tested the two versions of the parallel port driver by randomly setting the `buffer` variable and accessing it within the `writeData` function in order to make the driver fail. As expected, the non-instrumented version re-emitted the whole buffer to the parallel port, resulting in duplicated data being sent. On the other end, the recovery-friendly version never lost data, and thus recovered completely totally transparently.

This result is very significant regarding the capability of this framework to produce reliable drivers. Previous research on device drivers and software recovery essentially relied on simple driver restarting or resuming from a previously captured snapshot. Both methods would fail to recover this driver in most cases. Restarting the driver automatically implies that the failed IPC is entirely processed again, likely resulting in duplicate data being written. Snapshotting the entire process does not ensure that the latest snapshot is recent enough to have the data iterator up-to-date with the current state of sent data. Moreover, it is also likely that the corruption of the `buffer` variable resides in the snapshot, resulting in the same failure occurring after recovery.

By letting the programmer decide which data is relevant to recovery, and by discarding all other data within the process, we ensure that important data is always preserved while maximizing chances to erase the cause of failure. This allows even very state-dependent drivers like character devices drivers to recover successfully from failures.

5.1.3 VESA VBE Graphic Driver

The VESA BIOS Extension (VBE(Video Electronics Standard Association, 1998)) specifies a standard and unified way to use graphic hardware. A program using the VBE API can use any graphic board that supports the standard without regarding to the actual underlying hardware.

Driver Implementation

VBE provides APIs for getting information about the video card (total memory, supported modes and their properties, ...), setting a particular video mode, getting the hardware address of the video framebuffer, and controlling various other aspects of the graphic board. The driver that we implemented provides a view over the graphic board and its video modes to let the client programmer choose the desired video mode, as well as an access to the video framebuffer.

To allow this, the driver must first get all this information for itself. On initialization, the driver uses the VBE API to fill two kinds of structures: the `Screen` information (video board properties like vendor, amount of memory, ...) and a list of `VideoMode`, which gives information about the resolution, pixel properties, and access rules of a given video mode. In particular, every mode descriptor holds the hardware address of its framebuffer.

The video driver exposes the information about the video card to the client through a shared memory page that is mapped read-only. When the client requests a video mode, the driver performs two operations: it sets the video mode to the graphic board, and maps the memory framebuffer to the client so it can draw on the screen. To do this last operation, the driver makes a privileged request to obtain a mapping of the hardware address of the framebuffer, then maps it to the requesting client. It also keeps the process ID of the client in memory; as the screen is a common resource, only an exclusive access is allowed.

The video driver holds the following information across requests:

1. Properties and supported modes of the video board,
2. Current holder of the screen,
3. Current video mode,
4. A mapping of the framebuffer of the current mode.

On driver failure, the first information is easily retrievable by querying the VBE API again at initialization time. However, since the driver has been restarted, this would happen in another memory page, and this duplicated information will make that the mapping of the current client would not correspond to the mapping of the driver. More serious, the current holder of the screen and current video modes would be lost; this means that the driver does not know who is the current holder of the screen, or if there is even a holder. In these conditions, if another process requests access to the screen, and the driver gives it, two processes would access the screen concurrently, resulting in unpredictable results.

Recovery Handling

There are two problems to solve in order to make the VBE driver transparently recoverable. The first one is to ensure that holder and current mode information are preserved across failure. The second is to avoid querying again the video card for its capabilities and supported modes on recovery, in order to avoid wasting a memory page.

The first problem is easily solved by placing the variables for current holder and video mode into persistent memory. Since the framebuffer is recoverable from this information and its hardware address is fixed, we prefer to retrieve it again on recovery to comply with our "preserve only the necessary" policy.

The second problem requires that the memory page that holds card and modes information is also placed into persistent memory. As it is dynamically allocated by the driver at initialization time, this requires the use of a dedicated persistent memory allocator, which works the same way as a classical memory allocator (but on the persistent memory part of the address space). Being placed in a part of the address space that is not cleared on failure, the memory mapping of the card and mode information is preserved across failure and remain the same between the driver and its potential client.

Recovery is limited to restoring the mapping of the video framebuffer. We chose not to keep this mapping in persistent memory because it is easily retrievable, and in the case it is the source of the failure we prefer to request it again.

Results

Once the video mode is set and the framebuffer is mapped to the client process, the video driver does not interfere with the drawing operations themselves, which are directly performed by the client by writing to the video framebuffer. It may, however, be requested for tasks like changing the beginning address of the physical screen (to support double buffering) or updating the color palette. Also, another process can request access to the screen, in which case it should be denied.

We introduced a bug in this last event that makes the driver fail, ran a graphical program, and let another process try to get access to the screen at random times. As we could expect, the restoration was effective and complete, and the graphical client was not perturbed by the video driver crashing. Drawing operations are not conditioned by the driver being unavailable, and the only potential delays happen when the client asks the driver the change the displayed video page while the latter is being recovered, as the IPC is delayed. However, we have not perceived any visible slowdown.

5.1.4 Sound Blaster Audio Driver

The sound blaster test case provides us with a more complex driver features set, which includes interrupt handling, DMA programming, and deeper hardware interaction. We developed a driver that is able to control the sound blaster 16 DSP (revision 4) and the mixer. The driver does not support MIDI playback.

The goal through this experiment is to prevent any noticeable side-effect upon driver recovery. We need to avoid the termination of the sound caused by a crash and recovery of the audio driver. In addition, we need to ensure that the audio driver still performs correctly after its recovery.

Driver Implementation

The sound blaster 16 DSP works in collaboration with the DMA[1] to provide uninterrupted audio playback. First, a memory area is reserved as the audio buffer. The DMA chip is then programmed with the bounds of this buffer. Audio data is sent to the sound card through the audio buffer.

It is only when the sound card is then programmed that the memory transfer actually starts. It must be given the properties of the audio data (sample size (8 or 16 bits), playback rate, mono or stereo data, ...) and the number of audio samples to play before issuing an interrupt. The DMA then sends the requested amount of data, and the sound card raises the interrupt which is supposed to be caught by the client in order to fill the audio buffer again. To obtain uninterrupted audio playback, the audio buffer is divided in two, and the sound card is asked to only play half of its total size before issuing an interrupt. The idea being that when the audio card plays one half of the buffer, the application writes the other half (Fig. 5.1). The driver provides interfaces to setup the

[1]Direct Memory Access. The DMA is a programmable chip that performs transfers between the memory and a peripheral, without involving the processor in the process.

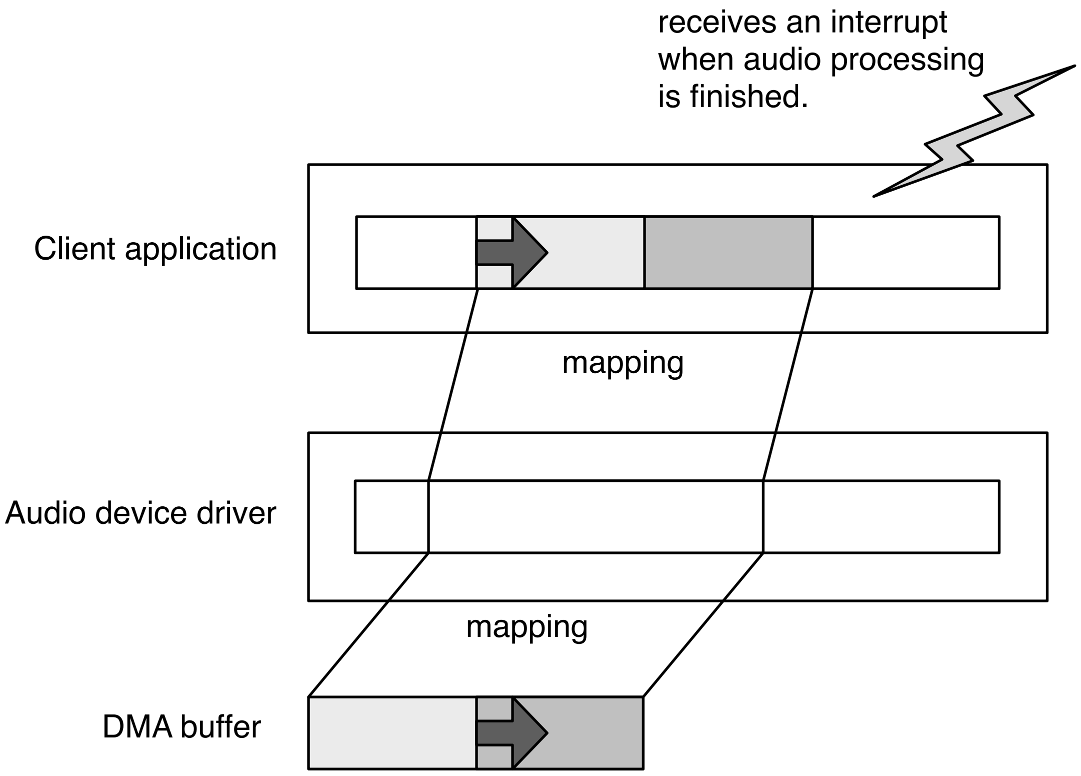


Figure 5.1: Audio playback mechanism

sound samples properties, playback rate, install the audio callback, and control playback. It also provides calls to control the audio mixer.

Recovery Handling & Results

Upon crash, the most important thing to ensure is that the audio playback in not interrupted, i.e. that the user perceives no gap in the audio playback while the driver recovers. This feature is easily supported by the DSP/DMA interaction and the fact that the application itself writes the audio data. DMA can be programmed such as it automatically restarts the same transfer again and again once it is finished. As the application directly writes audio data into the memory buffer transferred by the DMA to the card, this involves that once the card and DMA are programmed, audio playback continuity is handled by the application callback alone and do not rely on the audio driver being active.

Audio playback is therefore guaranteed as long as no system call is performed to stop the playback, change the card settings, or control the volume. Considering the frequency of these calls, and the speed at which the driver recovers, it is virtually impossible for the user or other applications to notice a crash and its successful recovery.

Some engineering still has to be performed on the driver to ensure successful recovery. The playback parameters must be kept for the eventuality the user wants to suspend and resume the playback. This is because the playback parameters are only given when the playback is initialized. Also, for security reasons, the current owner of the audio device must be preserved to avoid interception by another process during the recovery. Finally, the DMA buffer mapping of the driver should also be preserved, as it is shared with the client application. All these features are easily obtained by making the relevant variables persistent.

The mixer settings are directly queried and setup to the sound card, and thus require no particular care across recovery because they do not reside in the driver's address space.

Double-Buffered Variant

We developed an alternative version of the audio driver that uses a double-buffering mechanism for the DMA buffer. Instead of directly mapping the DMA buffer to the client application, it maps another area of the driver's address space that the client application fills upon a sound card interrupt. The audio library then sends a message to the audio driver that copies the client buffer into the DMA buffer and acknowledge the end of interrupt to the hardware. This design can be justified by safety reasons (in the former design, the audio library, and therefore the client application, directly acknowledges the hardware, a situation that not all operating systems may allow) or to support other, older models of sound cards.

The only situation where the user would experience a gap in the audio playback would be when the user callback function has not finished committing its part of the audio buffer, but another interrupt is raised because the sound card finished playing the other half of the buffer. This situation can happen if the driver is too slow to recover. However, even in the case of a buffer size of 4096 bytes and audio settings of 44100Hz for 16 bits stereo playback, this would only happen if the driver takes more than 23ms to recover. As section 4.6 shows, these delays should be respected on almost any machine.

5.1.5 Disk Driver

We developed a parallel ATA (PATA) disk driver (McLean, 1997). The purpose of the case study on the disk driver is to test the session library. The disk driver exchange data with a client through a shared memory region.

Driver Implementation

We first developed the disk driver without considering failure recovery. The driver implements the `HandlePut` and `HandleGet` methods. The `HandleBegin` and `HandleEnd` methods are empty. These methods are used to lock a resource that a server manages across multiple `Put` or `Get` requests. The disk driver does not manage the state that is required to be locked during the requests from a client, such as the current block number. Instead, a client specifies block number every time it request a transfer.

A process of the disk driver controls one disk. A different process controls another disk. Thus, a crash of the disk driver process has no effect on the other processes of the disk driver. The disk driver uses interrupts for efficiency of the communication with disks.

Recovery Handling

Upon recovery, the disk driver needs the information of the shared areas, such as a client's thread ID, the base address, and the size. Thus, the disk driver must declare that information persistent. However, the information of the shared areas is included in the `SessionServer` class, which is inherited by a server that offers shared memory areas to its clients. This means that the disk driver inherits the persistency of the information.

Results

To test the recovery of the disk driver, we implemented a workload that reads some data from a disk. We also inject a fault into the disk driver. The fault crashes the disk driver once every thousands of times of read access.

The framework transparently recovers the disk driver from the failure. The recovery process successfully handles the interrupt status, which could cause inconsistencies between the driver's and hardware states. This is because the restart of the disk driver re-initializes the interrupt status and the last request from a client is reprocessed.

5.1.6 File System

As an example of a server that holds clients' state, we developed an Ext2 file system server. Ext2 is a file system originally developed for the Linux operating system (Card et al., 1994)(Bovet and Cesati, 2005). The Ext2 file system manages files with *i-nodes* (Figure 5.2). A file or a directory is represented by an i-node. An i-node holds some pointers to data blocks which the contents of a corresponding file or directory is stored. Both i-nodes and data blocks are stored on a disk (Figure 5.3).

Server Implementation

Our implementation of the Ext2 file system creates a counterpart of a file called Ext2File when a client opens a file, and holds it until the file is closed. An Ext2File object holds i-node number, file operation mode, a pointer to i-node counterpart called Ext2Inode, and a pointer to the representation of a disk partition called Ext2Partition. The objects in the implementation read from or write to a disk through an Ext2Partition object. Open,

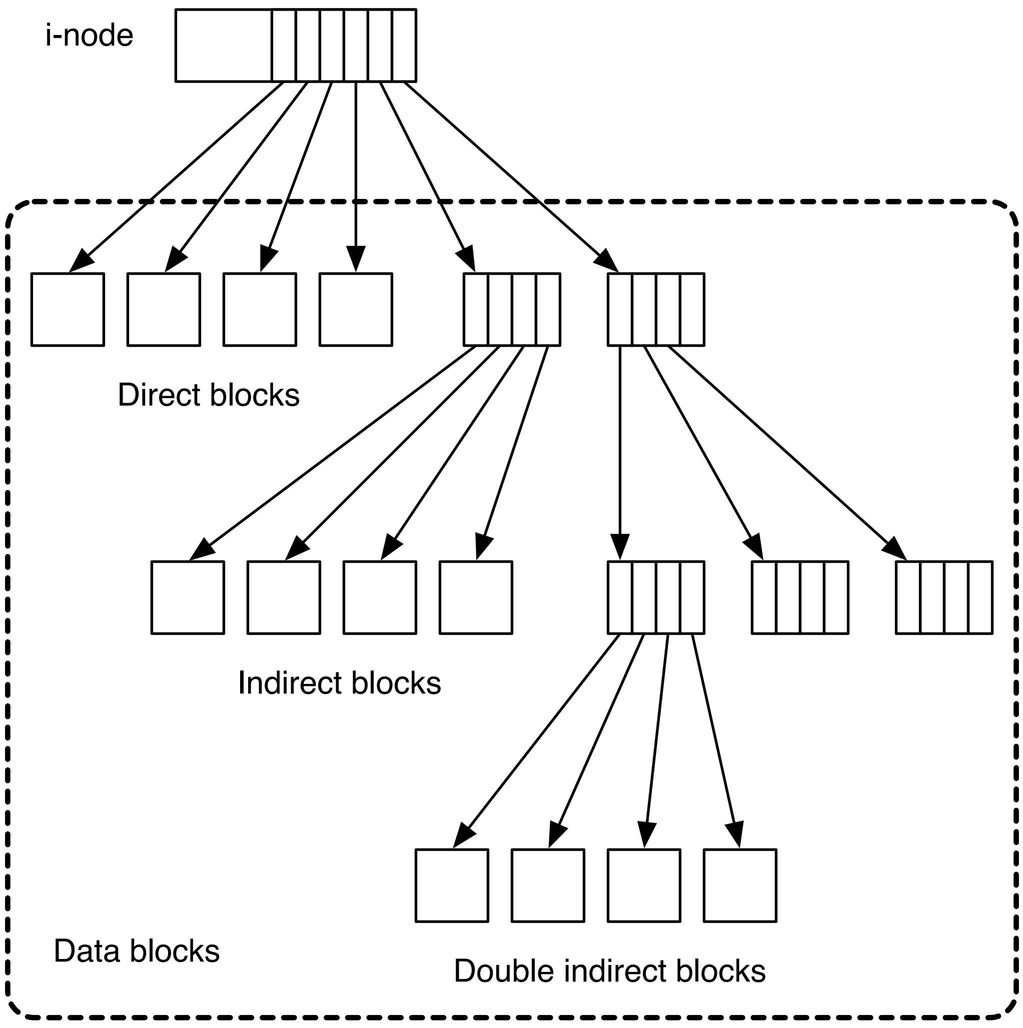


Figure 5.2: Ext2 file system

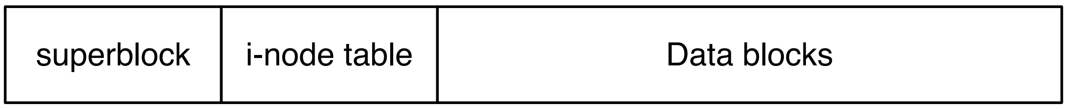


Figure 5.3: Simplified disk layout of the Ext2 file system

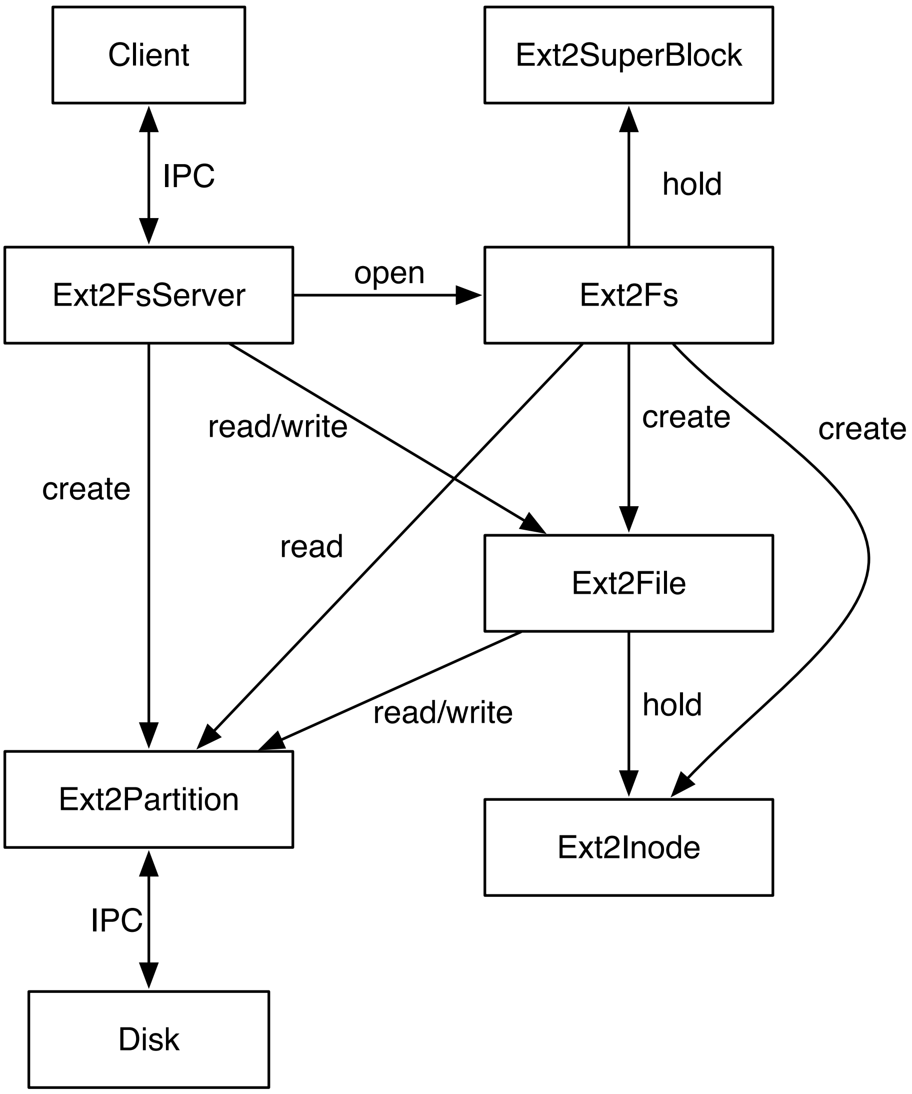


Figure 5.4: Design of the Ext2 file system server

close, read and write operations are implemented in HandleBegin, HandleEnd, HandleGet and HandlePut handlers respectively.

While Ext2File and Ext2Inode are local to a file, which means they are allocated when a file is opened and released when the file is closed, Ext2Fs, Ext2Partition, and Ext2FsServer are global objects which are dynamically allocated at the beginning and exist as long as the server is active. Ext2FsServer is the main part of the server. It derives SessionServer to exchange file data with a client through a shared memory area. Ext2FsServer initializes Ext2Partition and Ext2Fs. As described above, Ext2Partition is an interface to a disk partition. Ext2Fs provides functions to search and open files. It also holds an Ext2Superblock object which contains basic information about a file system on a disk such as the location of the i-node table and the data block area, etc.

When Ext2FsServer receives a request to open a file for a client, it forwards the request to Ext2Fs. Ext2Fs returns an Ext2File object that corresponds to the requested path if it is in the partition. The Ext2File object processes the following read/write requests on the file until the server receives a close request. Upon a close request, the server releases the Ext2File object to the heap.

Ext2FsServer can hold multiple sessions at once; a client can open multiple files at once, or some clients can open different files at the same time. However, file operations are serialized; Ext2FsServer processes at most one request at once.

Recovery Handling

The significant state in the Ext2 file system includes Ext2File objects, Ext2Inode objects, and shared memory areas. Since the framework preserves the shared memory areas, the Ext2 file system is required to preserve Ext2File and Ext2Inode objects.

We create dedicated persistent memory areas for these two types of objects (Listing 5.3). These persistent memory areas are an array of each object. The server manages the persistent memory area with a set of indexes. If the value of an index is non-zero, its corresponding elements are occupied.

```
static Byte         __index[NUM_CLIENTS] IS_PERSISTENT;
static Ext2File     __file[NUM_CLIENTS] IS_PERSISTENT;
static Ext2Inode    __inode[NUM_CLIENTS] IS_PERSISTENT;
```

Listing 5.3: The persistent memory areas for Ext2FsServer

The handler that opens a file uses the persistent memory area. It get an Ext2File object via the Open method of Ext2Fs. Since the Ext2File object is allocated in the heap, it will be discard upon restart recovery. Thus, the handler copies the object to the persistent area (Listing 5.4), and releases the original Ext2File object. Finally the handler copies the index to the SessionControlBlock.

Results

We conducted a fault injection test of the file system under continuous read operations. As a result, we confirmed that the persistent objects are properly handled and the file system is transparently recovered from a failure.

5.2 Reliability Evaluation

This section evaluates the reliability of operating system services on the framework through a fault injection test. A *fault injection test* intentionally injects one or more faults into a

```
stat_t
Ext2FsServer::HandleBegin(const L4_ThreadId_t& tid,
                          L4_Msg_t& msg)
{
    ...
    file = _e2fs->Open(path, mode);
    ...
    index = AllocateFileContainer();
    __file[index].Copy(file);
    __inode[index] = *file->Inode();
    __file[index]._inode = &__inode[index];
    delete file;
    session->data = index;
    ...
}

stat_t
Ext2FsServer::HandleEnd(const L4_ThreadId_t& tid,
                        L4_Msg_t& msg)
{
    ...
    ReleaseFileContainer(session->data);
    session->data = -1;
    ...
}
```

Listing 5.4: The open and close operations at the server side

target system in order to evaluate if the system is able to deal with the faults, or observe what effects the faults produce. This section first addresses the design and implementation of our fault injection testing platform, then shows the result of fault injection on Arc.

5.2.1 Fault Injection Testing Platform

The Fault Injection testing Platform (FIP) is built into Arc. Since the main purpose of the framework is to improve the reliability of operating system services apart from the core services, we regard L4 microkernel, Sigma0, ART, and P3 as sufficiently reliable systems. On the other hand, the other operating system services in Arc such as device drivers, file systems, etc. have comparably shorter life cycle than the core systems do. The purpose of the fault injection test is to evaluate the reliability of the restart framework. L4 microkernel, Sigma0, ART, and P3 are out of target.

There are two purposes of a fault injection test. One of the purposes is to observe the effect of a fault in a target system. For this purpose, a fault is injected to observe how it grows and becomes effective. The other purpose is to test the recovery mechanisms of the system. A fault is injected to test if a recovery mechanism recovers the error state in an anticipated way. Since we'd like to evaluate the recovery mechanism of Arc in this section, we construct FIP putting more weight on the last characteristic of a fault injection test than the first.

Fault injection is a typical way of evaluating the reliability of a system. There are various ways to perform fault injection tests. This dissertation builds a fault injection system borrowing the idea from the one built for evaluating the Rio file cache (Chen et al., 1996).

Corrupting servers from outside: A fault is required to be injected outside a target in order to simulate a Heisen-bug. It is possible to generate a fault inside a target. This is not a precise way to simulate a Heisen-bug. Firstly, a fault injection from inside tends to generate deterministic faults. To reduce the modification of a target, a fault injector should be as small as possible. However, a simple fault injector causes a deterministic fault. That means that that kind of fault injectors cannot simulate Heisen-bugs. Secondly, a complex fault injector changes the layout of a target on memory. A complex fault injector can generate some Heisen-bugs. However, it occupies some chunks of memory. This may change the way of error propagation inside a target.

Controlling the frequency of faults: In other words, it is necessary to control the number of faults injected at once. The ratio of fault effectiveness depends on the number of faults in a program.

Injecting different types of faults: Different types of faults should be generated to test various cases. Fault generators should be prepared as plug-ins for flexibility.

Design

The fault injection platform (FIP) consists of a user level and a system level components. The user level component controls the sequence of a fault injection test. An operator controls the targets, type of faults, number of faults, etc. through this component. The system level component implements fault injection. Receiving a request from the user level component, the system level component generates faults and injects them to the target.

FIP consists of Arc, a fault injector, and target components (Fig. 5.5). Target components are loaded from a disk and executed, by a command issued by an operator. Therefore, Target components are not launched when the system boots up. A fault is injected before target components start execution.

The main part of the fault injection platform is integrated in P3. FIP receives type and number of faults. A single type of fault can be injected at once; FIP cannot mix

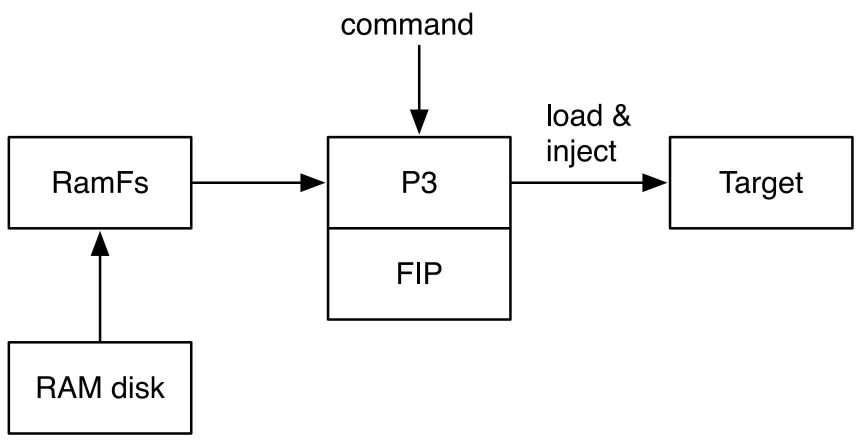


Figure 5.5: The architecture of FIP

multiple types of faults.

FIP provides two ways to inject a fault. The first way is to inject a fault at install time. A fault is injected to a target component by the fault injection tool before it is copied to the disk. The injection is performed outside Arc, such as a user's development platform. An experiment in this way will have a fault injected version of a component, in addition to the original. Therefore, a user is able to identify faults injected in advance, by comparing two versions of a component. This may help failure analysis.

The second way is to inject a fault at load time. A fault is injected to a target component when it is loaded from a disk. Since it is P3 that loads user programs, fault injection is performed by P3. In this case, the contaminated version of a component only exists on the memory. This means that failure analysis may be difficult, in particular, if the fault causes system crash. On the other hand, a user doesn't have to prepare different versions of a component at install time.

A single type of fault can be injected at once. If different types of faults are allowed to be injected at once, the evaluation will have to consider combination of faults. This is too complicated; Heisen-bugs rarely occur in a real system. Single fault injection is considered sufficient to simulate Heisen-bugs. From a practical point of view, this simplifies the user interface; A user simply specifies a type of fault and it frequency.

The fault injector is a kind of instruction interpreter. It interprets instructions one by one, and invokes a fault generator if the current instruction is the one the fault generator is interested in. The interpreter consists of a scanner and a parser. The scanner reads program by bytes. The parser interprets the stream of bytes, and invokes fault generators when the interpretation matches to some conditions.

Table 5.1: Injected Faults

Abbreviation	**Type of Fault**	**Method of Injection**
1	Assignment statements	Changing the source or destination register
2	Conditional constructs	Deleting branches
3	Initialization	Deleting instructions responsible for initializing a variable at the start of a procedure
4	Off-by-one errors	Changing conditions such as `>` to `>=`, `<` to `<=`, etc.

Implementation

As enumerated by Chen et al. (Chen et al., 1996), this fault injector generates the following types of faults.

5.2.2 Instruction interpreter

In order to inject a fault selectively to a target, we implemented an instruction interpreter for Intel IA32 instruction set in P3. The interpreter processes binary code when P3 loads the program into memory (Figure 5.5). A fault generator is invoked through the interpreter and injects a fault into the program. A single fault generator is invoked at a time to reduce the complexity of failure analysis. The interpreter adds extra time to program loading. However, it makes no effects to the other performance of P3. The instruction interpreter is also written in C++. We developed four types of fault generators.

5.2.3 Fault generators

Fault generators on Table5.1 are implemented. Four of them are implemented under the instruction interpreter in P3. The rest of them are implemented as a library and directly linked to a server program.

Assignment Statements

A fault in assignment statements is generated by changing the source or the destination register of the move instructions. This fault generator changes a register, not the value of a register. In IA32, A move instruction that takes the source and the destination registers as its operands takes MOD R/M byte to specify the registers. This fault generator changes the source or the destination register in the MOD R/M byte. A move instruction to be modified is randomly selected.

Conditional Constructs

The fault generator emulates a fault in a branch by deleting a branch instruction. It looks for a branch instruction such as `jcc` in IA32, and replace it with `nop` instructions. The fault generator is implemented under the instruction interpreter.

Initialization

This fault generator corrupts the initialization of local variables. The implementation focuses on a move instruction that accesses a stack pointer with an immediate value. The fault generator randomly removes this type of move instructions. The fault generator is implemented under the instruction interpreter.

Off-by-one Errors

An off-by-one error is a programming error in conditional statements. For instance, mistaking <for `<=` changes the behavior of a program at boundary condition. The fault generator inject this type of faults by changing the condition of a branch instruction.

5.2.4 Experiment

We chose a wave file player as the workload for the experiment. We developed a wave file player from scratch. The wave file player reads the wave data from a hard disk through a file system, process it, and write to the audio device (Figure 5.6). Under this workload, all the components are active during the experiment.

We inject some faults to a server at once. Although we can specify the frequency of fault injection, actual number of faults depends on the size of the target.

The experiment is conducted on the QEMU instruction emulator running on the Linux kernel version 2.6.24-21 on the Intel Core2 Duo 2.66GHz processor with 4MB of cache and 2GB of RAM.

5.2.5 Results

We first introduce the distribution of target instructions, which FIP modifies (Table 5.2). `mov/movsx/movzx` are the move instructions that take only registers as their operands. They moves the value of a register to another register. `move reg` are the move instructions that move an immediate value to a register. `move imm` are the move instructions that move an immediate value to a register or an address. The fault generator for assignment statements modifies `mov/movsx/movzx` and `mov reg`, while one for initialization modifies `mov reg` and `mov imm`. `jcc` are conditional jump instructions. The fault generators for conditional constructs and off-by-one errors modify these instructions. The bottom row indicates the total number of instructions including `nop`, in each binary file.

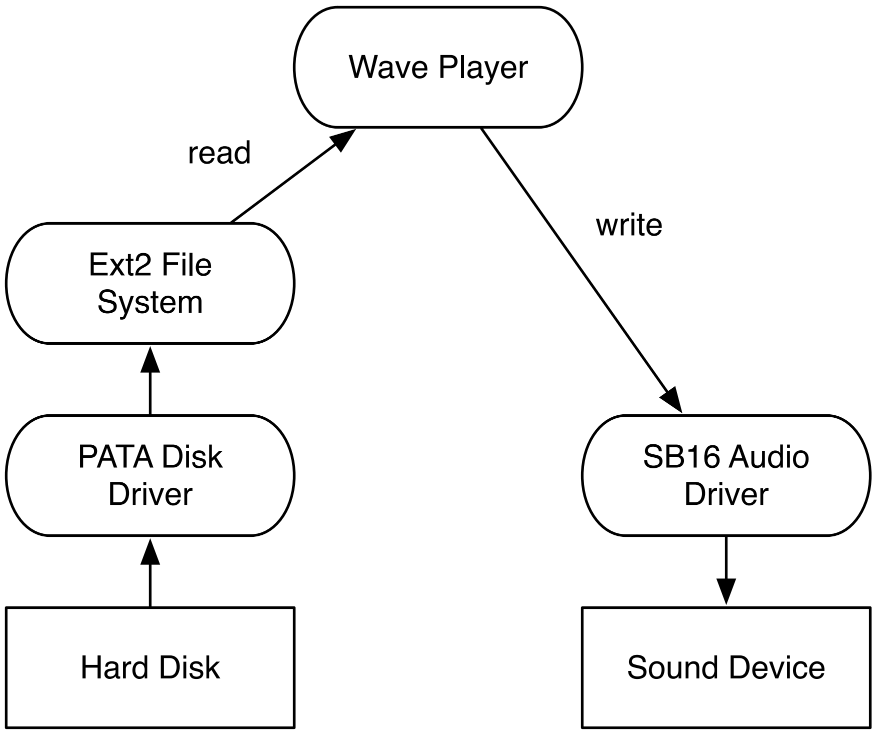


Figure 5.6: The experimental setting

Table 5.2: Number of target instructions

Instruction	`pata`	`ext2`	`audio`
`mov/movsx/movzx`	1515	2476	1054
`mov reg`	150	109	72
`mov imm`	451	554	332
`jcc`	443	683	306
Total	11789	15608	9459

Table 5.3: The result of fault injection (assignment statements)

Target	Effective Rate	Detection Rate	Success Rate
`pata`	72%	77.8%	50%
`ext2`	92%	95.7%	72.7%
`audio`	92%	100%	52.2%

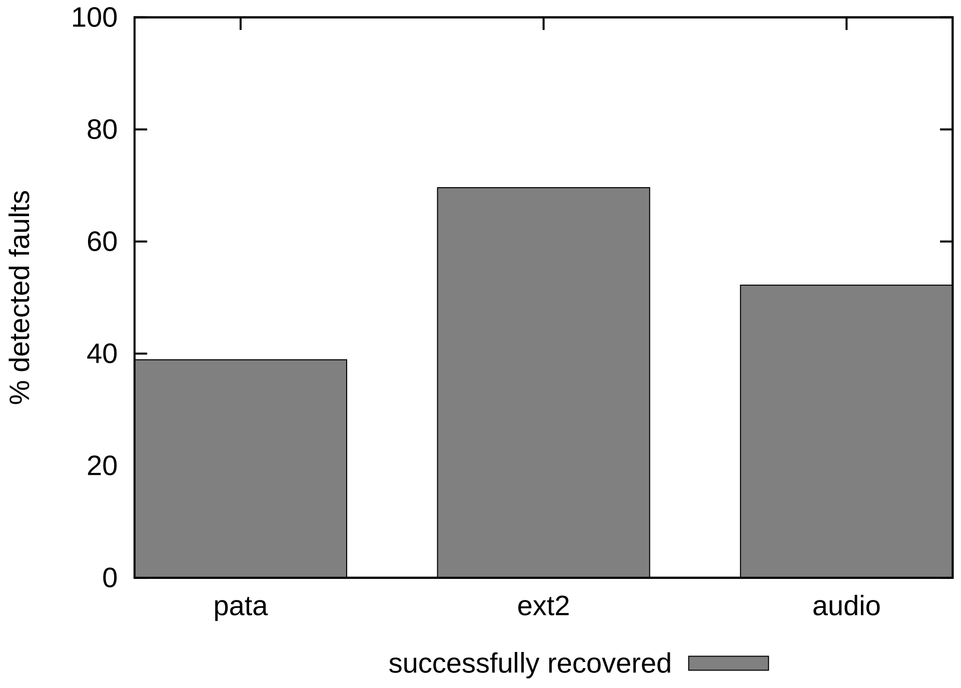


Figure 5.7: The result of fault injection (assignment statements)

Table 5.3 and Figure 5.7 show the result of fault injection to assignment statements. In the table, the effective rate is the percentage of the injected faults are manifested. The effect varies. Some faults cause crash, but some other cause freeze. The detection rate is the percentage that the supervisor (P3) can detect the error caused by an injected fault. P3 then restarts the target. The success rate indicates the rate of successful recovery. Unsuccessful recovery causes another failure. We conducted fifty times of the experiment for each target.

In the result of fault injection to assignment statements, P3 achieved to detect all the effective faults. In any case, the recovery sometimes failed and the targets freeze or exit. We consider this is because the persistent memory area was contaminated by the error.

Table 5.4: The result of fault injection (condition constructs)

Target	Effective Rate	Detection Rate	Success Rate
`pata`	26%	30.8%	75%
`ext2`	70%	74.3%	92.3%
`audio`	56%	67.9%	0%

Table 5.5: The result of fault injection (initialization)

Target	Effective Rate	Detection Rate	Success Rate
`pata`	58%	79.3%	95.7%
`ext2`	76%	52.6%	95%
`audio`	60%	67%	50%

While the previous result shows similarity among the targets, each of the target indicates different result in the result of fault injection to condition constructs (Table 5.4 and Figure 5.8).

The faults injected to `pata` generated no crash failure. Thus, P3 could not detect them. As described above, the current implementation detects only memory access violations. In cooperation with other types of error detectors, P3 would be able to recover them.

In the case of `ext2`, P3 detected approximately 80% of faults and successfully recovered half of them. Some failures could not be detectable by the current P3. For instance, the file system failed to match strings due to the injected fault, thus a client failed to open a file that was surely on the disk.

P3 failed to recovery `audio`. In any failed case, `audio` frozen. We consider the reason is corruption of the persistent memory area.

Next, we injected some faults to the initialization of some variables (Table 5.5 and Figure 5.9). In some cases, P3 failed to recover the failure. We consider this is also the case of the corruption of the persistent memory area because some persistent variables are initialized with immediate values.

The fault injection of the off-by-one errors shows a similar result to that of fault injection to condition constructs (Table 5.6 and Figure 5.10). Indeed, both of the test modify conditional jump instructions. This case also shows the vulnerability of the persistent memory area. We consider that a modification on a condition of a boundary causes

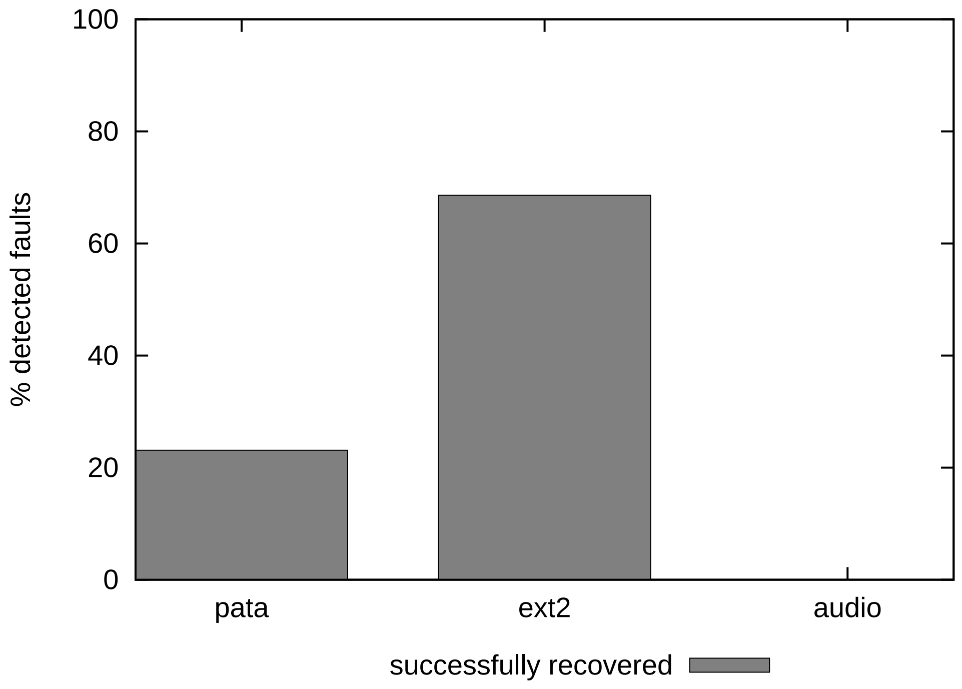


Figure 5.8: The result of fault injection (condition constructs)

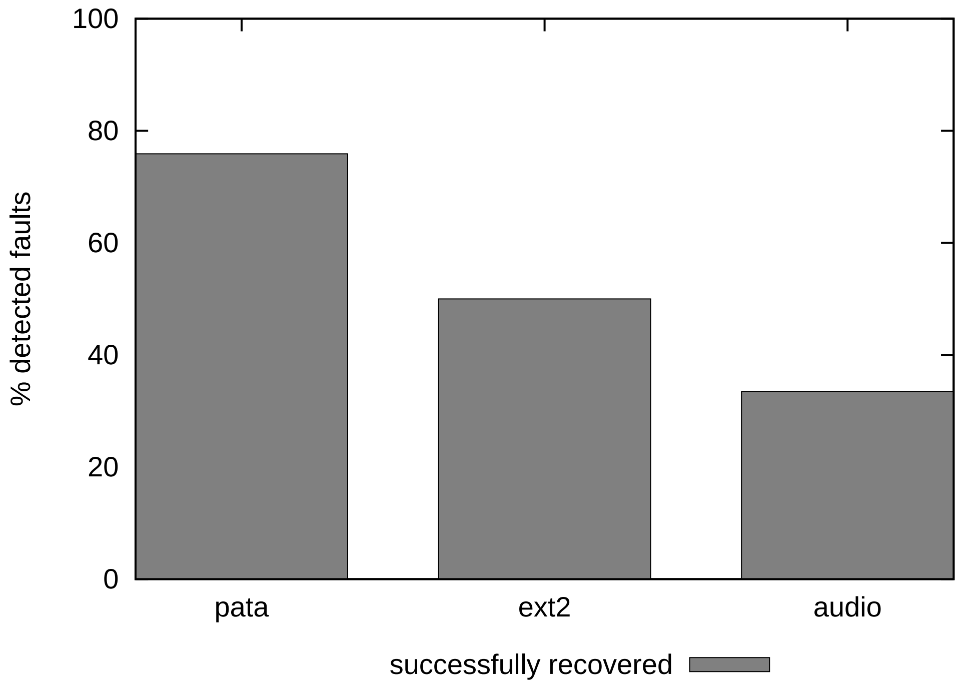


Figure 5.9: The result of fault injection (initialization)

Table 5.6: The result of fault injection (off-by-one errors)

Target	Effective Rate	Detection Rate	Success Rate
`pata`	44%	27.3%	66.7%
`ext2`	56%	96.4%	92.6%
`audio`	50%	80%	0%

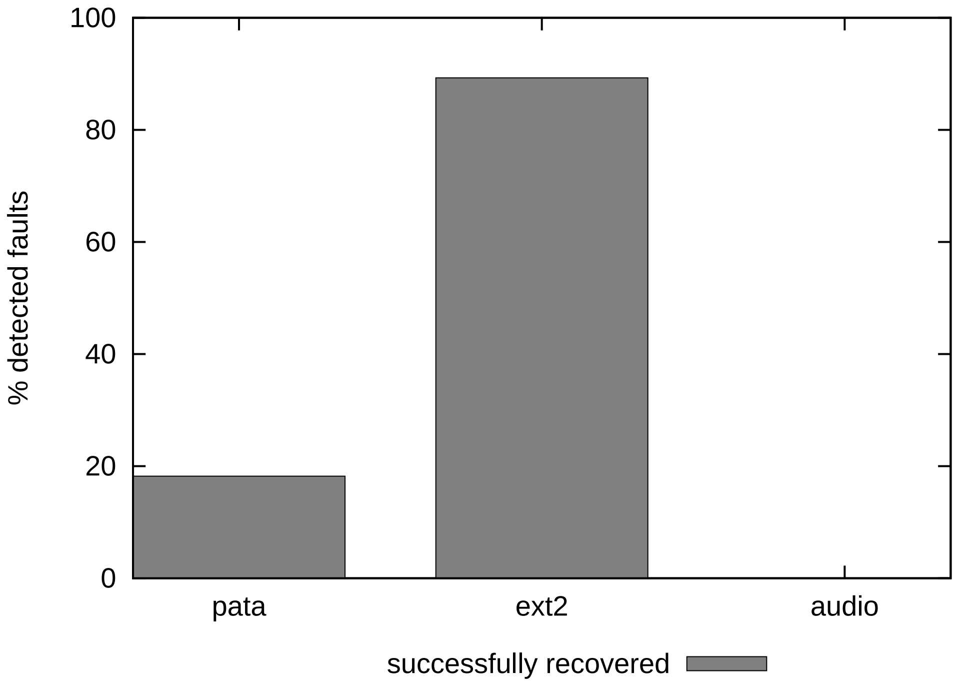


Figure 5.10: The result of fault injection (off-by-one errors)

writing a wrong value to the persistent memory area.

Most of the faults injected here were effective when a target started up, not when the workload was active. Since the persistent memory is frequently accessed at initialization time, we consider P3 often failed recovery. The result of case study supports this idea because the targets survive failures in their services.

5.3 Discussions

Through the evaluation, we assumed fail-stop failure. This is because our framework focuses on recovery, not detection of errors. As described in Chapter 3, for detecting byzantine or fail-stutter failure, the framework needs to cooperate with certain error detectors.

The case study has shown that P3 successfully recovers a crash of a target during its service, which are caused by an out-of-memory error or violation of invariants. If a developer has sufficient programming skill and knowledge on the server he develops, our framework is useful for these cases.

Servers with their internal state such as the parallel port device driver and the Ext2 file system server, are transparently recovered; failure and recovery are invisible from their clients. This is a successful result of our approach, which is in the middle of crash-only software and checkpoint-restart. Operating system components so far didn't care about their recovery, though they gave a large effort to check invariants. The result indicates that efficient recovery is possible if developers are aware of failure recovery of operating system components. The drawback of our approach is to force developers to have responsibility for tuning the detail of recovery. While this research gave a guideline to develop a server with failure recovery in mind, cooperation with compilers and programming tools would overcome the drawback.

Downtime in the case study was short enough. In particular, the case study on the audio device driver proved its sufficiency; there was no gap in the audio playback during the test. However, downtime depends on the amount of internal state that the recovery handler has to deal with. The servers developed in the case study directly used persistent memory areas to store data, because we designed to store all the significant data into the persistent memory area which was statically allocated. On the other hand, it is possible for a certain server to reproduce its internal state by means of the contents of persistent memory in order to reduce the size of persistent memory areas. Moreover, it is also possible to implement a dynamic memory allocator for persistent memory base on our framework. In these cases, the recovery handler is require to rebuild the server's internal state and it may cause long downtime. A server developer has to be aware of the trade-off between downtime and the size of persistent memory.

We succeeded to recover a single server, but not a group of servers. This is because our recovery mechanism is apparently able to confine failure in a process. However, considering the result of the fault injection test, group restart should be available in case a restart recovery fails.

The series of fault injection test have shown the vulnerability of the persistent memory areas. These tests have put more weight on programming errors than errors in inputs or execution environments. As described above, the current implementation does not protect persistent memory areas from wild writes. Instead, the group restart should be applied in case of failed recovery. Fortunately, when P3 failed recovery, the target always froze. A freeze detector must improve the situation. Since the current number of servers is too low to form a group, we could not study on the group restart. Hence, this is a part of future work.

The following sections discuss several further issues.

5.3.1 Applicability to a Monolithic Kernel

The framework is built on top of a microkernel which provides strong isolation of processes. If one would like to port the framework to a monolithic kernel in which all system components are melted together, it is necessary to solve two isolation issues.

The first issue is fault containment. A process on a microkernel is isolated from the kernel and other processes. This guarantees fault containment; a failed process cannot make a wild write to another process or the kernel. Under a monolithic kernel, isolation facilities like Nooks isolate a driver from the other parts of the kernel by means of the MMU isolation mechanism(Swift et al., 2006). This means a driver runs as a process under Nooks and this approach introduces a large performance penalty because of the switch between page tables. If another approach for fault containment, such as software fault isolation(Wahbe et al., 1993), can be applied, the framework would work with a monolithic kernel with little loss of performance.

The second issue is isolation for communication. Since a server interacts with a user process through IPC, an IPC message is isolated from its local stack by the IPC mechanism, which means the server is easily restarted without loosing the message from the user process. On the other hand, a component in a monolithic kernel communicates via some function calls. Hence a message from the kernel is placed in a server's local stack, which results in the message being lost on server restart. Thus, a special calling mechanism is necessary to preserve the message. Again, Nooks provides a mechanism to switch the kernel stack to a dedicated stack when the execution path moves to a device driver, and switch back it to the kernel stack when the call finishes. The mechanism proposed in Nooks is therefore also applicable to our framework.

5.3.2 Limitations

There are several limitations to our approach. First, a device driver or a subsystem in another operating system needs to be partially modified in order to work within Arc and the framework. Introducing the restart handler and the persistent memory requires the subsystem to change its data structures. More fundamentally, the libraries available in Arc may not be compatible with other operating systems. Some subsystems are probably required to undergo heavy modifications. However, logic to manipulate a peripheral or a resource is common as long as the used hardware or specification is the same. In case of device drivers, dissemination of IDL for device drivers such as Devil (Mérillon et al., 2000) could improve the situation.

Second, recovery of servers using shared memory areas is not fully supported. Some servers use shared memory pages to share data with their clients so that they can transfer a large amount of data. Those memory pages are directly mapped from the server's address space to the client's address space. According to the L4 specification, if the server's address space is accidentally removed (by crash or whatever), the mapping path is changed such that the client is mapped from the source (in our case, P3 of the server). After recovery of the server, it keeps the same memory page as before the crash from its P3. Hence, the server and the client can still have normal communications. However, the mapping path is incorrect: if the client doesn't release the mapped pages when it terminates the communication with the server, it can still hold the memory pages that might be mapped to another client. In the next release, we prohibit the direct memory mapping from a sever to a client, and involve P3 so that such memory mapping is performed indirectly.

Chapter 6

Conclusion and Future Directions

This chapter describes the conclusion and the future directions of the research.

6.1 Conclusion

Dependability should be considered through software construction, not only provided by runtime support. Previous approaches assumed software including operating systems is correct. Hence, they do not have recovery in mind. Indeed, hardware fault was traditionally considered a problem of dependability. However, software systems have been so complicated that their behavior seems non-deterministic despite the fact that they are logical artifacts. External recovery mechanisms such as checkpoint-restart are rather ad hoc approaches, because the purpose of such approaches is to leave existing software unmodified. Language support such as exception handling mechanisms allows programs to repair the data they manipulate on, however, it is insufficient to recover a crashed program.

The research has considered a framework for developing resilient operating systems. We first analyzed the characteristics of restart recovery, which we adopted as the basic mechanism of the framework. Restart recovery requires no hardware supports and complicated algorithms. Hence it is suitable for a system with poor resources, in particular, mobile devices.

The research proposed the classification of failure recovery techniques. The classification includes pre-failure and post-failure periods. The classification indicates that one should consider failure recovery not only reactive to failure, but also proactive to failure.

The research proposed a framework for developing resilient operating systems based on immediate restart and developed a prototype operating system implementation, called Arc, under the framework. The framework adopted microkernel architecture. Arc was constructed as a set of independent servers on top of the microkernel. The framework provides a set of class libraries to the servers. The servers under the framework are automatically recovered upon failure.

Although developers have to specify the data to be store on persistent memory, we are able to design light-weight failure recovery mechanism, which is efficient for operating systems, based on a result of the classification. Immediate restart is a middle solution between complete restart and checkpointing. It requires persistent memory which is a part of main memory with non-volatility. By means of persistent memory, a failed process can be immediately restarted.

The case studies on some device drivers and a file server indicated that immediate restart with persistent memory and our guideline was feasible; we first developed these components without recovery in mind, then appended the functionality of recovery. The significant result of the case study is we achieved to recover programs with their internal state transparently. For instance, the parallel port driver could keep the consistency of hardware state across recovery, the file system could manage the consistency of files.

6.2 Future Directions

The result of the fault injection test indicates that the protection of persistent memory areas was necessary. It showed us the vulnerability of persistent memory, particularly at its initialization time. Transaction processing of persistent memory areas will improve the robustness of persistent memory(Lowell and Chen, 1997)(Saito and Bershad, 1998). However, this approach generally introduces the overhead to access persistent memory.

Our restart recovery technique requires a short down time. It is necessary to identify how it affects to real-time computing. Many embedded systems requires some degree of real-time property of applications. Restart recovery generates a gap in an operation. This may break the requirements for real-time computing. *Process resurrection* is a technique to recover a hard real-time application by restart(Lee and Sha, 2005). However, the way of constructing an application is very limited, hence it is not applicable to operating system components. We need to find the balance between the performance of real-time and restart recovery.

Appendix A

Arc Design Note: Page Management

Physical page management is one the core functions of Arc root task (ART). The page management enables processes to efficiently use the main memory and share a page to exchange data. This document describes the design and implementation of the page management.

A.1 Requirements

This section enumerates the requirements to the Arc page management.

Access to the initial programs: The initial programs are loaded into the main memory by a boot loader such as GRUB. Since the initial programs are placed regardless to the page management policy of Arc, ART is required to keep the information of the initial programs such as their base addresses and sizes.

Reuse of unused pages: To share the main memory efficiently among processes, page allocation should be dynamic.

On-demand paging: This is also for the efficiency of memory usage. In particular, this is useful for dynamic data structures such as stacks, and heaps. In cooperation with a program loader, the text area of a process can be on-demand.

Copy-on-write: A copy-on-write page is a read-only page shared among some processes. If one of the processes tries a write access to the page, the page management allocates a new page, copies the contents of the copy-on-write page, and remaps it to the process. In particular, copy-on-write is used to create a process based on an existing process. This mechanism enhance the efficiency of memory usage.

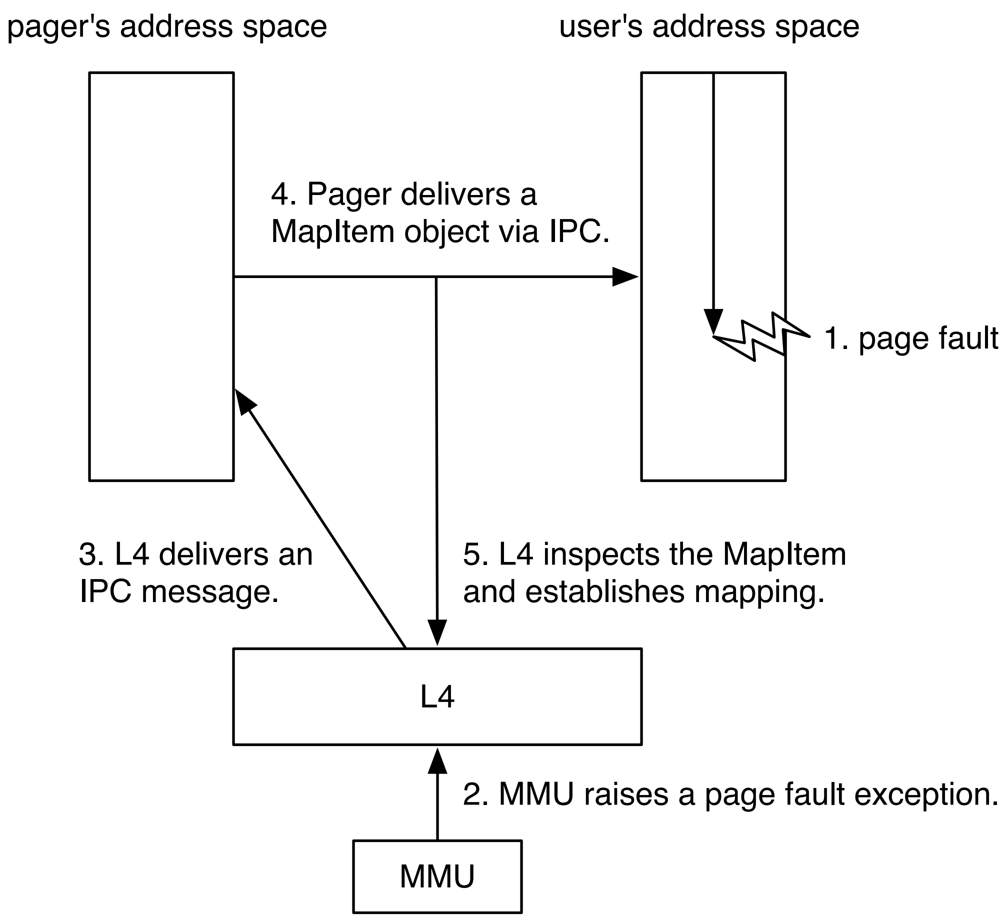

pager's address space
user's address space
4. Pager delivers a MapItem object via IPC.
1. page fault
3. L4 delivers an IPC message.
5. L4 inspects the MapItem and establishes mapping.
L4
2. MMU raises a page fault exception.
MMU

Figure A.1: Page mapping via a page fault

Page sharing between processes: Some processes require to share pages among them to exchange a large amount of data.

Physical pages: Some processes, in particular some device drivers, require to access physical addresses. For instance, a device driver needs a page in the first 1MB of memory. The page management has to authenticate the processes that require direct access.

There is another type process that needs to know the physical address of a virtual address. The page management should allow processes to look up the physical address of a virtual address.

A.1.1 L4 Mapping Protocol

L4 defines two ways of page mapping. One way is via a page fault (Figure A.1), and the other way is via a direct communication (Figure A.2).

L4 microkernel delivers a page fault as an IPC message to a pager that is responsible for the user process that triggered the page fault. A page fault message is delivered to a

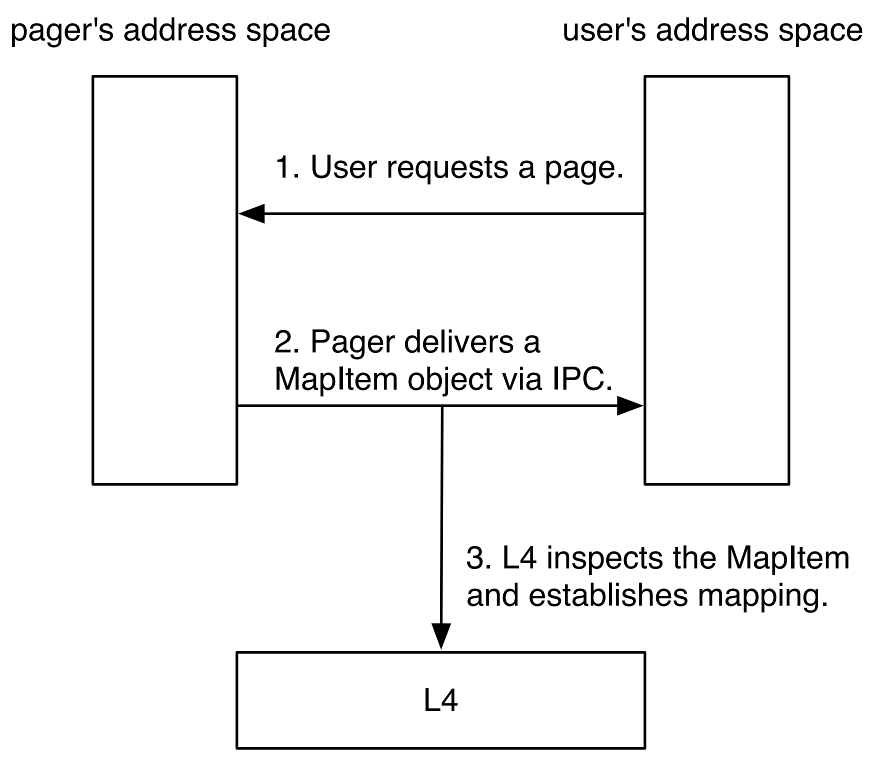

pager's address space
user's address space
1. User requests a page.
2. Pager delivers a MapItem object via IPC.
3. L4 inspects the MapItem and establishes mapping.
L4

Figure A.2: Page mapping via a direct communication

pager by the kernel as if it is delivered by the user process. Thus, it includes a special value in its header so that a pager can recognize if a message is a page fault or not. After receiving a page fault message, a pager checks if the address where the page fault occurred is valid. If it is valid, the pager maps a page from its own address space to the requesting address space. If it is invalid, the pager regards it as a failure, and carries out recovery.

For successful page fault handling, the pager puts the fault address and the source address in a data structure called MapItem, and send it back to the fault process. Dealing with the message, the microkernel establishes a mapping.

A process can also get mapped through a direct IPC. This allows a user process to map a page directly to another. In this case, the process first needs to send a mapping request to the source process. Although the format of a mapping request is free, it is supposed to include the destination address at least. The source process basically performs the same operations as a pager does in page fault handling. It verifies the destination and the source addresses, creates a MapItem object, and send it back to the destination process.

Processes including pagers are allowed to grant a page (Figure A.3). A granted page is moved from the source process to the destination process. In other words, if a process grants a page, the page is not shared, and the source process loose access to the page.

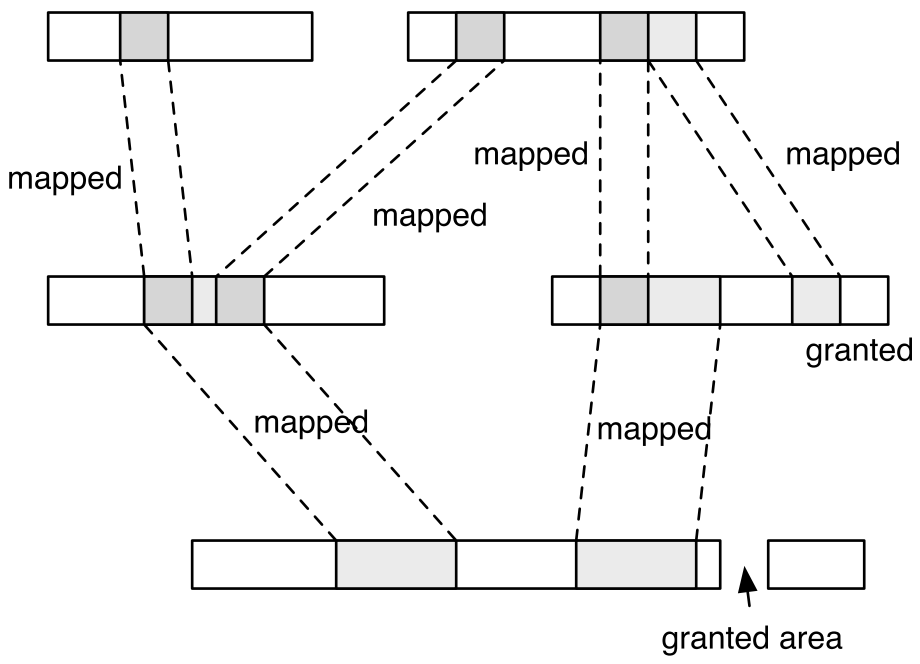


Figure A.3: The grant operation

A.2 Design

This section first addresses how ART manages physical memory pages. Then, we take a look at the mapping poilicy, and the API and the architecture of memory management.

A.2.1 Physical Page Management

ART manages the physical pages with its local page table. The page table is an array of the Page objects. A physical page is referenced through the relative position of a corresponding Page object. A Page object represents a 4K page on IA32 platform. Each of the Page objects hold the following information.

size The size of the page that the Page object represents. On IA32 platform, 4K or 4M is available. If 4M is specified, the subsequent 1023 elements are ignored by the page management.

type The type field holds *reserved*, or *conventional*. A reserved page is reserved for ART, sigma0, or the kernel, thus it is never mapped to any user process. A conventional page can be mapped to any process.

state It indicates the current state of a page. There are six states a page can take: free, allocated, reserved, mapped, unmapped, and immortal.

holders This is a list of processes. Each element contains the destination address which the page is mapped to, the ID of the main thread of a process, and access rights (readable, writable, and executable).

ART builds the page table at the beginning of its execution to instantiate a memory allocator (malloc and mfree) for itself. ART obtains the current memory layout through some L4 system calls. At this moment, the layout holds base addresses and sizes of reserved areas, the kernel, and initial programs including Sigma0 and ART itself, which are loaded by a boot loader such as GRUB. Base on this layout information, ART initializes the page table.

ART dynamically allocates and releases pages with a page allocator. A page allocator looks for a chunk of free pages, changes the state of the pages allocated, and gives them to a service routine in ART. When ART releases pages, the page allocator changes the state of the pages free.

ART needs to track down the destination of a physical page to release all the pages mapped to a terminated process, or to allow page sharing between processes. Hence, ART holds a list of counterparts of running processes called Space. ART creates a Space object when it creates a process. Thus, a user process and its corresponding P3 are managed through different Space objects.

A Space object includes a mapping database. The mapping database is a table that consists of key-value pairs. It holds a destination address in the corresponding process as the key, and the pointer to a Page object as the value.

A.2.2 Mapping Policy

ART performs three types of page mapping: anonymous page mapping, reserved page mapping, and specific page mapping.

Page Mapping Error

If a requested address is illegal, ART forces to terminate the process. An address is illegal if it is out of the user area, or if the process tries to share a private page.

Anonymous Page Mapping

Anonymous page mapping fetches a page through the page allocator, and maps it to the requested virtual address. For instance, the annoymous page mapping is applied to most heap and stack pages.

Reserved Page Mapping

Reserved page mapping deals with a reserved page that corresponds to the requested virtual address. ART searches a Page object in the mapping database of the requesting process with the requested virtual address. If found, it checks the ownership and permission of the page, then maps it to the destination. In the case a reserved page is copy-on-write, ART maps a copy of the original page to the destination.

Direct Page Mapping

Direct page mapping allows a process to get a page at the specified physical address. Since this mapping also allows a process to map the kernel area, the mapping procedure needs to verify the requesting physical address and authenticate every requesting process.

A.2.3 Controlling Page Status

It is sometimes necessary for a process to control page status. In particular, sharing a page that is already assigned to a process. In this case, the page is mapped as a private page. Thus, the owner process need to change its status to *shared*. ART provides a method to control the page status.

A *private* page is mapped to at most one process. This is the default status of a page. A *shared* page is mapped to at most two processes. One process is the owner of the page and able to control the status. The other is the partner and access the page following its access rights.

A.2.4 Architecture

The page management is handled by five components in ART (Figure A.4). The page table holds an array of page frames, each of which holds the information of a physical page. The page allocator manages free pages. The mapping management assigns physical pages to processes communicating through IPC. The mapping management fetches or releases pages through the page allocator, but directly operates on the page table to update the status of the pages. The process management creates or deletes space objects. A space object is a counterpart of a process. It involves a mapping database. A mapping database holds a table that maps a physical address to a virtual address. An entry is registered to a mapping database by the mapping management when a physical page is mapped to a process.

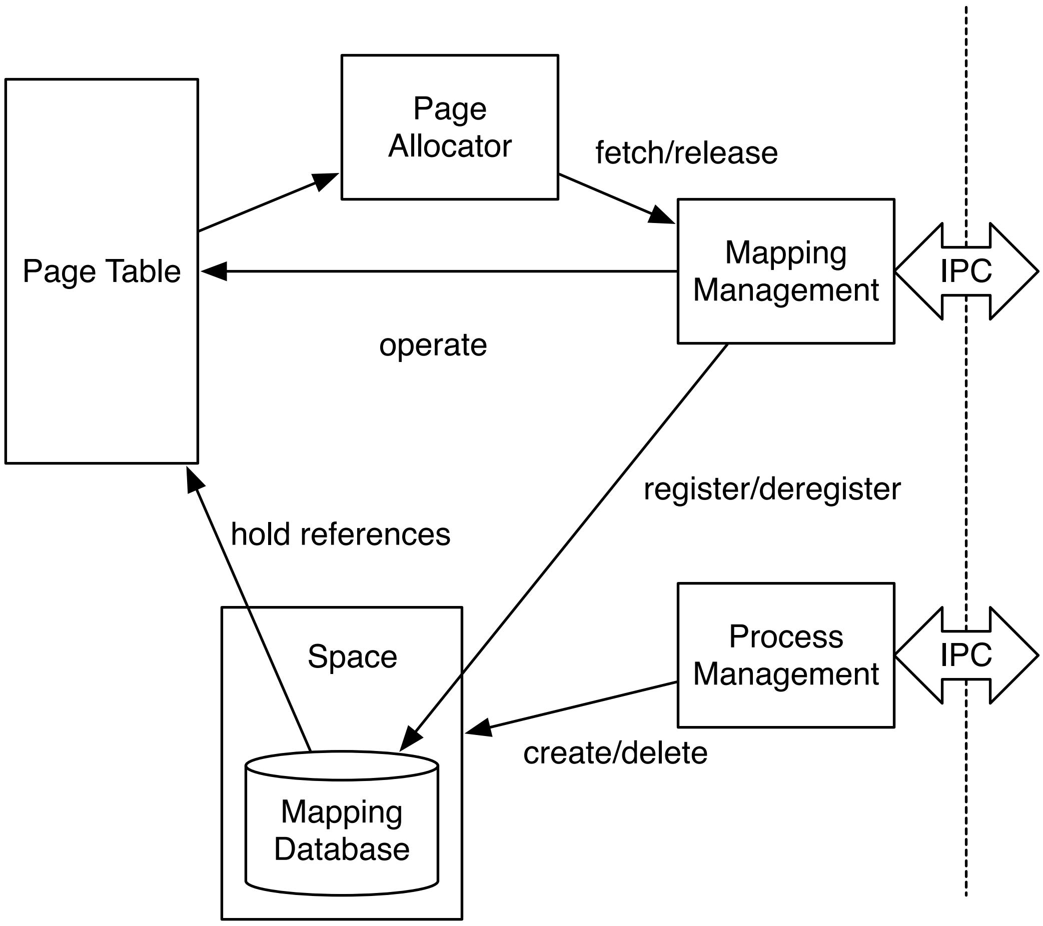


Figure A.4: Architecture of the physical page management

Appendix B

Arc Design Note: Arc Root Task

Root task is a privileged process that is allowed to issue all L4 system calls. Arc Root Task (ART) is the root task of Arc OS. ART is responsible for the privileged system calls and physical page management. In addition, it launches the initial servers such as MicroShell, a console driver and a keyboard driver.

B.1 Architecture

ART consists of two threads. One thread is responsible for user process life-cycle. The other is reponsible for physical page management. They share the page table and Space objects which are counterparts of user processes.

B.2 Space

ART manages address spaces and threads with *Space* objects. A Space involves a page mapping database, thread information, etc. ART creates or deletes a thread through Space.

Each thread is required to have a unique identifier (TID). ART maintains a table that holds the usage of TID.

B.3 Interrupts

An interrupt is delivered via IPC on L4. Hence, an interrupt source is regarded as a thread. In order to receive an interrupt, a thread needs to register its TID to a corresponding interrupt thread. This registration is a privileged system call; only ART is able to issue the call.

ART is responsible for the association of a user thread and an interrupt thread on behalf of the user thread.

B.4 Protocols

ART provides services to P3 (Per-Process Pager) via IPC. This section describes the IPC protocols of ART.

B.4.1 Thread Management

ART provides functions for managing thread and address space life cycles.

New Thread

To create a new thread in a user process, P3 needs to specify a space ID of the user process. P3 specifies its own address space when it needs a thread running inside P3. If a new thread is created, a TID is returned. The thread is activated by its pager. For instance, a thread in a user process is activated by P3, and a thread in P3 is activated by ART.

Register	Name	Type
reg_0	space ID	L4_ThreadId_t

New Task

To create a new process, ID of P3 must be given. ART internally creates two Space objects for P3 and the user process.

Register	Name	Type
reg_0	pager	L4_ThreadId_t

Priority

The protocol changes the priority of the given thread. The range of the priority follows the specifiecation of L4.

Register	Name	Type
reg_0	thread ID	L4_ThreadId_t
reg_1	priority	L4_Word_t

Delete Thread

The protocol deletes the specified thread in the specified address space. The speicified address space must corresponds to the P3 that is the sender of the message.

Register	Name	Type
reg_0	Thread ID	L4_ThreadId_t
reg_1	Space ID	L4_ThreadId_t

Delete Task

The protocol deletes the specified address space. All the threads in the address space are deleted. The sender of the message must be P3 that manages the given space.

Register	Name	Type
reg_0	Space ID	L4_ThreadId_t

Set Interrupt

The protocol associates the specified thread to the specified interrupt number. The upper 18 bits contain thread number and the remaining 14 bits contain interrupt number.

Register	Name	Type
reg_0	Thread number(18)/IRQ(14)	L4_Word_t

Unset Interrupt

The protocol detaches the thread from the specified interrupt number.

Register	Name	Type
reg_0	IRQ	L4_Word_t

Execute

The protocol creates a new process that executes the given program. `argv` represents the virtual address where the program path and its arguments are stored. `argc` represents the virtual address where the size of the array is stored.

Register	Name	Type
reg_0	argc	L4_Word_t
reg_1	argv	L4_Word_t

Exit

P3 sends this message when it exits. The status represents the exist status.

Register	Name	Type
reg_0	status	L4_Word_t

B.4.2 Memory Management

ART provides functions for memory mapping. Every L4 process requires a pager process that is mainly responsible for page fault handling. Sigma0 is the initial pager that holds all the physical memory pages and works as the pager for the root task. Sigma0 has a drawback that it cannot unmap a page once it is mapped to another process. This is why ART, the root task of Arc, owns all the physical pages on behalf of Sigma0.

Allocate

The protocol reserves the count of pages for the specified destination addresses. If a process plans to share the reserved page with another process, Peer field is used to specify the process ID and Attribute field is used to specify the access right.

Register	Name	Type
reg_0	Destination	L4_Word_t
reg_1	Count	L4_Word_t
reg_2	Peer/Attribute	L4_Word_t

Release

The protocol unmaps and releases the address.

Register	Name	Type
reg_0	Address	L4_Word_t

Map

The specified page is mapped to the destination. If the source process ID is *nil thread*, the source address represents a physical address. If the source process ID is the *any thread*, the source address is ignored and an anonymous page is immediately mapped to the destination.

Register	Name	Type
reg_0	Destination Address	L4_Word_t
reg_1	Destination Process	L4_ThreadId_t
reg_2	Source Address	L4_Word_t
reg_3	Source Process	L4_ThreadId_t
reg_4	Count	L4_Word_t

Unmap

The protocol unmaps the specified address.

Register	Name	Type
reg_0	Address	L4_Word_t

Physical Address

The protocol returns the physical address of the specified virtual address.

Register	Name	Type
reg_0	Address	L4_Word_t

Appendix C

Arc Design Note: Program Loading

Program loading is a core function of a multi-task operating system. Per-Process Pager (P3) is a memory manager and a program loader dedicated to only ne user process. It is placed between ART and the user process. This document describes how it manages programs and memory and how interact with the user process and ART.

C.1 Design

ART provides process and thread management and memory management. In order to work as a dynamic operating system, Arc is required to provide a program loading mechanism. There three choises of program loading mechanisms.

The first approach is that ART is responsible for program loading. ART loads a binary from a disk into reserved pages, set up the stack, and run the program to activate the address space (Figure C.1). ART runs as the pager for every process. In this approach, ART needs to check if a fault address is reserved for the text or stack area because ART cannot map the reserved pages to an inactive address space in advance. ART also needs to check which section a page fault occurs. For instance, ART has to protect a text area from being written.

The second aproach is to map a program loader into a user's address space. The program loader loads a binary from a disk into the address space, set up the process, and jumps to the entry point of the program (Figure C.2). ART needs to hold the executable image of the program loader. Like the first approach, all page allocation requests are handle by only ART.

The last approach is to create a pager for each process. A per-process pager loads a program and set up a stack for the user process in its own address space, then maps those areas upon page faults generated by the user process (Figure C.3). A page is mapped from

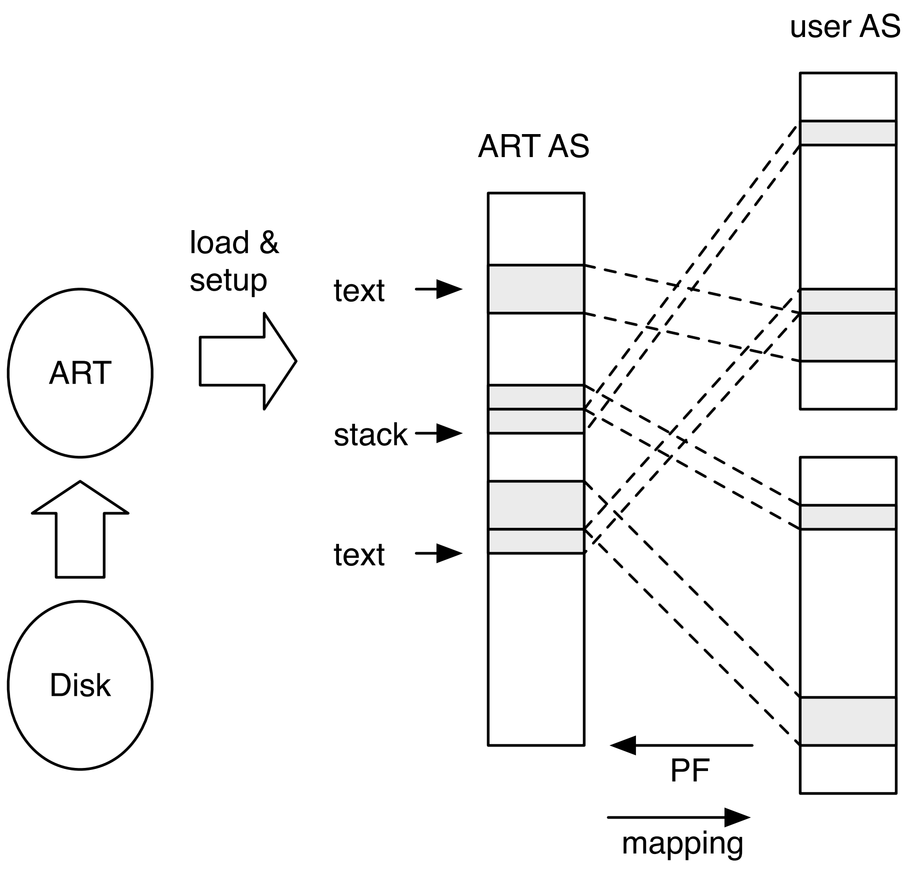


Figure C.1: The first approach of program loading

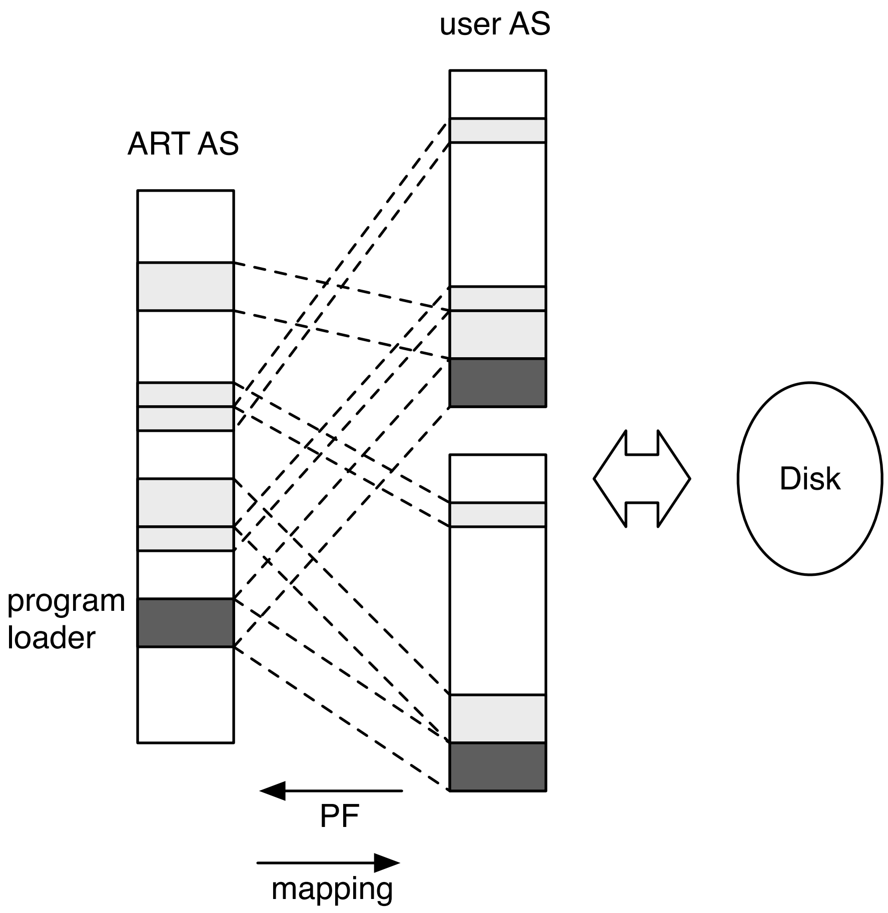


Figure C.2: The second approach of program loading

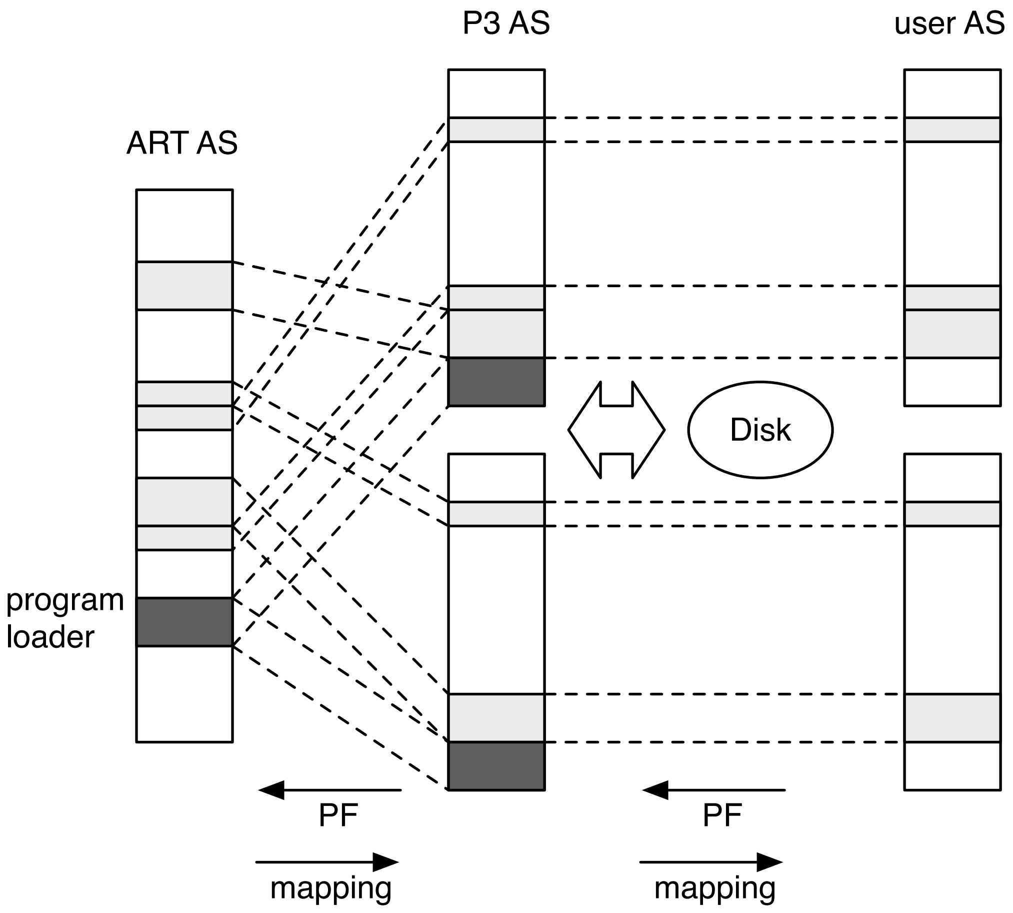


Figure C.3: The third approach of program loading

ART to a per-process pager, then mapped to the user process.

We chose the last approach to implement persistent memory. Persistent memory preserves data across process termination. For instantance, a program can fetch the latest data in the persistent memory after its restart. This is mandatory for our recovery framework. Since a per-process pager holds the persistent memory area of the user process, we can destroy a failed user process.

Appendix D

Arc Design Note: Quick Start Guide

Arc OS is a multi-server operating system built on top of L4 microkernel. This document offers an experience of Arc OS to the reader through hands-on practice. In this document, we run Arc OS in QEMU virtual machine. The following document assumes to use Linux operating system with the kernel version 2.6.

D.1 Getting the Source Code

The latest version of source code is found at `http://www.dcl.info.waseda.ac.jp/~ishikawa/arc/arc_latest.tar.bz2`.

Decompressing the tarball, `arc` directory is created.

D.2 Building Arc OS

To build Arc, the following tools are mandatory.

Table D.1: Requirements

Name	Version	Description
gcc	4.0 or later	GNU C compiler
g++	4.0 or later	GNU C++ compiler
cmake	2.6.0 or later	A build tool like GNU make
ruby	1.8.0 or later	Ruby interpreter
grub	0.9.7	A boot loader
e2fsprogs	1.40.8 or later	Tools for manipulating Ext2 file system
e2tools	0.0.16 or later	Tools for manipulating Ext2 file system
qemu	0.9.1 or later	An instruction emulator

Change the current directory to `arc`, then make a new directory, say `build`. Change the current directory to `build`.

```
% cd arc
% mkdir build
% cd build
```

Run `cmake`. Then, run `make`. This will build the Arc OS binary files.

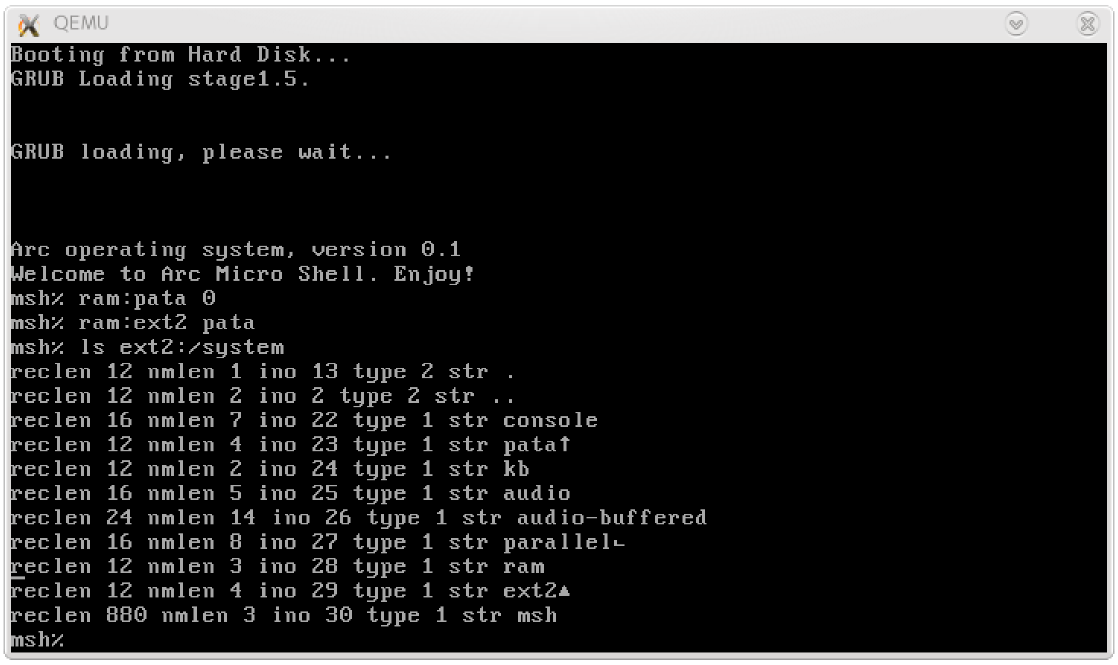
```
% cmake ..
% make
```

D.3 Playing with Arc OS

Run `make` again with the argument `qemu`. This will create a disk image and automatically launch QEMU with the image.

```
% make qemu
```

Arc MicroShell will appear on the QEMU console (Figure D.1).



```
Booting from Hard Disk...
GRUB Loading stage1.5.


GRUB loading, please wait...



Arc operating system, version 0.1
Welcome to Arc Micro Shell. Enjoy!
msh% ram:pata 0
msh% ram:ext2 pata
msh% ls ext2:/system
reclen 12 nmlen 1 ino 13 type 2 str .
reclen 12 nmlen 2 ino 2 type 2 str ..
reclen 16 nmlen 7 ino 22 type 1 str console
reclen 12 nmlen 4 ino 23 type 1 str pata↑
reclen 12 nmlen 2 ino 24 type 1 str kb
reclen 16 nmlen 5 ino 25 type 1 str audio
reclen 24 nmlen 14 ino 26 type 1 str audio-buffered
reclen 16 nmlen 8 ino 27 type 1 str parallel∟
reclen 12 nmlen 3 ino 28 type 1 str ram
reclen 12 nmlen 4 ino 29 type 1 str ext2▲
reclen 880 nmlen 3 ino 30 type 1 str msh
msh%
```

Figure D.1: Arc MicroShell

D.4 Directory Layout

On the top directory, there are eight directories and two files.

Applications: Contains test applications including the wave player.

Boot: Contains Kickstart that is the second bootloader for L4.

Documents: Contains a series of Arc Design Note, and some publications.

Include: This is obsolete directory. However it still contains some important header files.

Kernel: Contains the L4 microkernel implementations. The implementations are downloaded when Arc is built for the first time.

Libraries: Contains libraries.

Services: Contains servers.

Tools: Contains build tools.

CMakeLists.txt: This is the root of `cmake` file.

Doxyfile: A configuration file for Doxygen.

Bibliography

V. Abrossimov, F. Herrmann, J. C. Hugly, F. Ruget, E. Pouyoul, and M. Tombroff. Fast error recovery in CHORUS/OS: The hot-restart technology. Technical report, Chorus Systems, Inc., 1996.

M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *2006 ASM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC2006)*, pages 1–10, October 2006.

A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

M. Baker and M. Sullivan. The recovery box: Using fast recovery to provide high availability in the UNIX environment. In *Proceedings of Summer 1992 USENIX Conference*, pages 31–44, June 1992.

J. F. Bartlett. A NonStop kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, December 1981.

B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transaction on Computer Systems*, Vol. 8(No. 1):pp. 37–55, February 1990.

A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transaction on Computer Systems*, 7(1):1–24, February 1989.

D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd ed.* O'Reilly Media, Inc., 2005.

T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

G. Candea and A. Fox. Crash-only software. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HotOS IX)*, pages 67–72, May 2003.

G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, December 2004.

R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the 1st Dutch International Symposium on Linux*, 1994.

V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 45(2):311–332, March 2001.

P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 74–83, October 1996.

Chorus systems. Chorus/ClassiX r3.1b for ix86 - product description. Technical Report CS/TR-90-25.1, Chorus systems, 1996a.

Chorus systems. Chorus/ClassiX r3 - technical overview. Technical Report CS/TR-96-119.8, Chorus systems, 1996b.

F. M. David and R. H. Campbell. Building a self-healing operating system. In *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 3–10, September 2007.

F. M. David, J. C. Carlyle, E. M. Chan, D. K. Raila, and R. H. Campbell. Exception handling in the Choices operating system. *Advanced Topics in Exception Handling Techniques*, 4119:42–61, 2006.

F. M. David, J. C. Carlyle, and R. H. Campbell. Exploring recovery from operating system lockups. In *Proceedings of 2007 USENIX Annual Technical Conference*, pages 351–356, June 2007.

B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA2003)*, pages 78–95, October 2003.

ECMA International. Standard ECMA-335, Common Language Infrastructure (CLI), 2005.

M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language support for fast and reliable message based communication in Singularity OS. In *Proceedings of EuroSys 2006*, pages 177–190, March 2006.

D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of USENIX Summer 1990 Technical Conference*, June 1990.

J. Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandem Computers, 1985.

J. Gray and A. Reuter. *Transaction Processing.* Morgan Kaufmann Publishers, 1993.

K. C. Gross, V. Bhardwaj, and R. Bickford. Proactive detection of software aging mechanisms in performance critical computers. In *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 17–23, December 2002.

Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of The 25th International Symposium on Fault-Tolerant Computing*, pages 381–390, June 1995.

G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

H. Ishikawa, A. Courbot, and T. Nakajima. A framework for self-healing device drivers. In *Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO2008)*, pages 277–286, October 2008.

O. Laadan and J. Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 323–336, 2007.

C. Lameter. Extreme high performance computing and why microkernels suck. In *Proceedings of the Linux Symposium*, pages 251–262, June 2007.

K. Lee and L. Sha. Process resurrection: A fast recovery mechanism for real-time embedded systems. In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RSTAS'05)*, pages 292–301, March 2005.

J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, pages 175–188, December 1993.

J. Liedtke. Toward real μ-kernels. *Communications of the ACM*, 39(9):70–77, September 1996.

T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, 2nd ed.* Addison-Wesley, 1999.

D. E. Lowell. *Theory and Practice of Failure Transparency.* PhD thesis, University of Michigan, 1999.

D. E. Lowell and P. M. Chen. Free transaction with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, pages 92–101, October 1997.

D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI2000)*, pages 289–304, October 2000.

P. T. McLean. *AT Attachiment 3 Interface (ATA-3).* X3T13, 1997.

F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *The 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 17–30, October 2000.

D. Oppenheimer, A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, and D. A. Patterson. ROC-1: Hardware support for recovery-oriented computing. *IEEE Transactions on Computers*, 51(2):100–107, February 2002.

J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX Technical Conference, January 1996.

J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, September 1988.

C. Peacock. Interfacing the standard paralel port, 2007. URL \url{`http://www.beyondlogic.org/spp/parallel.htm`}.

L. Pullum. *Software Fault Tolerance Techniques and Implementation.* Artech House, inc., 2001.

F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies–a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3), August 2007.

B. Randell. System strcture for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, January 1975.

Y. Saito and B. Bershad. A transactional memory service in an extensible operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 53–64, June 1998.

M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ASM Transactions on Computer Systems*, 12(1):33–57, February 1994.

D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation.* A K Peters, Ltd., 3rd edition, 1998.

M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, February 2005.

M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ASM Transactions on Computer Systems*, 24(4), November 2006.

M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, December 2004a.

M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), November 2004b.

Universität Karlsruhe. L4 experimental kernel reference manual, version x.2. Technical report, System Architecture Group, Department of Computer Science, Universität Karlsruhe, April 2005.

Video Electronics Standard Association. *VESA BIOS Extention Core Function Standard, Version 3.0*, 1998.

R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, pages 203–216, December 1993.

Publication List

Hiroo Ishikawa and Tatsuo Nakajima. Construction and reconfiguration of a component-based embedded JVM. *International Journal of Computer Science and Network Security*, 8(6):29–35, June 2008.

Tatsuo Nakajima, Eiji Tokunaga, Hiroo Ishikawa, Daiki Ueno, Kaori Fujinami, Midori Sugaya, and Shuichi Oikawa. Software infrastructures for building ubiquitous computing environments. *International Journal of Computer Systems Science & Engineering*, 22 (3), May 2007.

Hiroo Ishikawa, Alexandre Courbot, and Tatsuo Nakajima. A framework for self-healing device drivers. In *Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008)*, pages 277–286, October 2008.

Yu Murata, Wataru Kanda, Kensuke Hanaoka, Hiroo Ishikawa, and Tatsuo Nakajima. A study on asymmetric operating systems on symmetric multiprocessors. In *Proceedings of the 2007 IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2007)*, pages 182–195, December 2007.

Hiroo Ishikawa and Tatsuo Nakajima. Micro-reboot support for multi-server operating systems. In *Proceedings of the 1st International Workshop on Dependable Ubiquitous Nodes*, November 2007.

Hiroo Ishikawa, Tatsuo Nakajima, Shuichi Oikawa, and Toshio Hirotsu. Proactive operating system recovery. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (Poster Session)*, October 2005.

Hiroo Ishikawa and Tatsuo Nakajima. EarlGray: A component-based Java virtual machine for embedded systems. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC 2005)*, pages 403–409, May 2005.

Shuichi Oikawa, Hiroo Ishikawa, Masatoshi Iwasaki, and Tatsuo Nakajima. Constructing secure operating environments by co-locating multiple embedded operating systems. In

Proceedings of the 2nd IEEE Consumer Communications and Networking Conference (CCNC 2005), January 2005.

Shuichi Oikawa, Hiroo Ishikawa, Masatoshi Iwasaki, and Tatsuo Nakajima. Providing protected execution environments for embedded operating systems using a μ-kernel. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing (EUC 2004), Springer-Verlag LNCS 3207*, pages 153–163, August 2004.

Tatsuo Nakajima, Kaori Fujinami, Eiji Tokunaga, and Hiroo Ishikawa. Middleware design issues for ubiquitous computing. In *Proceedings of the 3rd International Conference on Mobile and Ubiquitous Multimedia (MUM 2004)*, pages 55–62, 2004a.

Hiroo Ishikawa, Yuuki Ogata, Kazuto Adachi, and Tatsuo Nakajima. Building smart appliance integration middleware on the OSGi framework. In *Proceedings of the 7th International Conference on Object-oriented Real-time Distributed Computing (ISORC 2004)*, pages 139–146, May 2004.

Tatsuo Nakajima, Eiji Tokunaga, Hiroo Ishikawa, Kaori Fujinami, and Shuichi Oikawa. Human factor issues in building middleware for pervasive computing. In *Proceedings of the 2nd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (WSTFEUS 2004)*, pages 3–7, May 2004b.

Tatsuo Nakajima, Shuichi Oikawa, Hiroo Ishikawa, Masatoshi Iwasaki, and Midori Sugaya. Experiences with building distributed middleware for home computing on commodity software. In *Proceedings of the 4th International Workshop on Smart Appliances and Wearable Computing (IWSAWC 2004)*, pages 424–429, March 2004c.

Hiroo Ishikawa and Tatsuo Nakajima. A case study on a component-based system and its configuration. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2003), Springer-Verlag LNCS 2826*, pages 198–210, September 2003.

Hiroo Ishikawa, Yuuki Ogata, Kazuto Adachi, and Tatsuo Nakajima. Requirements for a component framework of future ubiquitous computing. In *Proceedings of the 2nd International Workshop on Software Technologies for Future Embedded Systems (WSTFES 2003)*, pages 9–12, May 2003.

Tatsuo Nakajima, Eiji Tokunaga, Hiroo Ishikawa, Daiki Ueno, Makoto Kurahashi, Masatoshi Iwasaki, Masahiro Nemoto, and Andrej van der Zee. Software infrastructure for building large-scaled smart environments. In *Proceedings of the 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, pages 82–89, January 2003.

Eiji Tokunaga, Hiroo Ishikawa, Makoto Kurahashi, Yasunobu Morimoto, and Tatsuo Nakajima. A framework for connecting home computing middleware. In *Proceedings of IEEE International Workshop on Smart Appliances and Wearable Computing (IW-SAWC 2002)*, pages 765–770, July 2003.

Daiki Ueno, Hiroo Ishikawa, Eiji Tokunaga, and Tatsuo Nakajima. Connecting object-oriented middleware for home computing with virtual overlay networks. In *Proceedings of International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, pages 163–168, May 2002.

Tatsuo Nakajima, Eiji Tokunaga, and Hiroo Ishikawa. Building middleware components for ubiquitous computing on commodity software. In *Proceedings of International Conference on Real-Time Computing, Systems and Applications (RTCSA 2002)*, March 2002a.

Tatsuo Nakajima, Hiroo Ishikawa, Eiji Tokunaga, and Frank Stajano. Technology challenges for building internet-scale ubiquitous computing. In *Proceedings of the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2002)*, pages 171–179, January 2002b.

Tatsuo Nakajima, Hiroyuki Aizu, Hiroo Ishikawa, Ichiro Sato, and Daiki Ueno. A virtual overlay network for integrating home appliances. In *Proceedings of the 2002 International Symposium on Applications and the Internet (SAINT 2002)*, pages 246–253, January 2002c.

Hiroo Ishikawa, Eiji Tokunaga, and Tatsuo Nakajima. A case study of implementing home appliance middleware on linux and java. In *Proceedings of the 2002 Symposium on Applications and the Internet Workshops*, pages 31–34, January 2002.

Tatsuo Nakajima, Hiroo Ishikawa, and Eiji Tokunaga. Software design issues for building internet-scale ubiquitous computing. In *Proceedings of Advanced Topic Workshop on Middleware for Mobile Computing*, November 2001.

Hiroo Ishikawa and Tatsuo Nakajima. Some software design issues for realizing internet-scale ubiquitous computing. In *Proceedings of Workshop on Experience with Reflective Systems*, September 2001.