PORTING THE *CHOICES* OBJECT-ORIENTED OPERATING SYSTEM TO THE MOTOROLA 68030

BY

BJORN ANDREW HELGAAS

B.A., Augustana College, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

TABLE OF CONTENTS

Chapter 1

Introduction

One of the goals of the *Choices*[**?**, **?**, **?**] project is to apply object-oriented design techniques to an operating system. This thesis evaluates the impact of these techniques on the process of porting the system from one machine to another. The object-oriented design of *Choices* should make it easier to port the system to new machines by providing a large body of reusable code and isolating machine-dependencies from the rest of the system.

The object-oriented approach to building software seems to offer advantages over traditional approaches because it emphasizes data encapsulation and code reuse through inheritance and polymorphism[**?**]. Most object-oriented languages support *classes*, abstractions for describing several objects with the same behavior. A class may have many *instances*, each of which is a distinct object with its own protected state, but which share the same procedures. Using *inheritance*, new classes can be built by specifying how they differ from previously built classes, allowing the new class to reuse some of the procedures of the original class. *Polymorphism* allows a procedure to use objects of different types, as long as they support a required set of operations.

Even more important than the reuse of procedures through inheritance is reuse of design, through object-oriented abstract designs, or *frameworks*[**?**]. Such a design consists of several classes and defines the interaction between them. The classes in the framework are usually abstract, so concrete subclasses must be supplied to fill in details left unspecified in the design.

Choices consists of several frameworks for the various parts of the operating system. Several of these abstract designs are incomplete; details that depend on a particular processor or machine architecture are intentionally left unspecified. Implementations of *Choices* for a particular machine complete the frameworks by supplying concrete subclasses that encapsulate those low-level details.

Each framework embodies a set of assumptions made about the underlying hardware. If these assumptions are sufficiently general, the framework can be reused without change for many different processors and machines. If they are not, the interfaces of the framework will require changes to accommodate unforeseen hardware characteristics.

Choices was originally written to run on the Encore Multimax, a symmetric shared memory multiprocessor based on the National Semiconductor 32332 processor. The port of *Choices* from this processor to the Motorola 68030 required no changes to processor-related interfaces, thus validating the design of these frameworks. During the port of *Choices* to the 68030, some "machine independent" code was changed and reorganized. In general, the affected code was related to the machine architecture, I/O devices, and other details outside the processor itself.

The frameworks enforce a separation of non-portable details from the main body of *Choices* code. This separation should make the system easier to maintain and easier to port to new architectures because only that code that explicitly depends on the processor and the machine needs to be modified. The fraction of the total amount of code dependent on the 32332 and

68030 processor architectures is very low, indicating that the non-portable parts of *Choices* are well-isolated.

The remainder of this thesis examines several *Choices* frameworks and how their design affects the process of porting them to a new processor. Chapter 2 is a high-level overview of the parts of *Choices* involved in porting *Choices* to a new processor, the process scheduling, exception handling, and virtual memory frameworks. Subsequent chapters examine these frameworks in turn, detailing the parts of each that must be modified when porting *Choices*. Finally, on the basis of the amount of code that is processor-dependent, we conclude that the object-oriented design of *Choices* has succeeded in isolating non-portable code from the rest of the system and that this isolation makes the system more modular and portable.

Chapter 2

Choices Overview

Choices code is organized into a hierarchy of C++ classes representing several frameworks, each one a subsystem of the kernel. This chapter is an overview of the major processor-dependent frameworks, including those for process scheduling, exception handling, and virtual memory.

High-level interfaces between and within each framework are specified by abstract classes. Concrete subclasses of each abstract class "flesh out" the system by providing implementations of the interface. These subclasses can provide different implementations of the same interface. For example, subclasses of the **MemoryObjectCache** class in the virtual memory framework can implement caching based on random, least-recently-used, first-in-first-out, or other strategies.

By a similar mechanism the system is customized to a particular type of machine and processor. Abstract classes such as **CPU** and **AddressTranslator** specify an interface that each implementation must support. A concrete subclass is then written for each processor and machine to which *Choices* is ported. The subclass hides idiosyncrasies of the machine and provides a uniform interface to the rest of the system.

Choices enforces a rigid separation between code that is intended to be portable across all architectures, code that depends on the processor type but should work on different machines using the same processor (processor-dependent code), and code that depends on details of the machine besides the processor type, *i.e.*, bus architecture and I/O devices (machine-dependent code).

Under this design philosophy, the process of porting *Choices* to a new machine translates into defining new concrete subclasses for several subsystems. Besides the areas discussed in this thesis, this includes writing device drivers for terminals, disks, network controllers, and other peripherals. The **IOController** framework, of which device drivers are a part, is an area of recent work in *Choices*, and is discussed in detail in [**?**]. Finally, each machine requires miscellaneous pieces of code for booting, mutual exclusion, and so on, many of which are discussed in [**?**].

2.1 Processes

The concept of a *process* as the execution of a computer program is among the most fundamental abstractions supported by operating systems. The process abstraction has a tremendous influence on the structure of applications. An application may be composed of a single process, several communicating processes in distinct address spaces, or even many cooperating processes in a single address space, depending on the type of computation and the process support provided by the operating system.

Most operating systems, including UNIX, support multiple processes executing in independent address spaces[**?**, **?**]. These processes interact only through services provided by the kernel, such as the file system, semaphores, or explicit interprocess communication primitives. Application processes execute in a protected environment; they cannot access memory used by the kernel or by other processes.

To avoid the overhead of using operating system services for all interprocess communication, some systems support *threads,* multiple processes running in a single address space[**?**]. While threads give up the protection afforded by separate address spaces, they can communicate via shared memory without involving the operating system at all. This type of communication can be very efficient, particularly if the machine has multiple processors and the threads are running simultaneously. The *Choices* kernel is implemented as a collection of threads running in a single address space, which contains all the objects that make up the kernel.

The process scheduling model implemented in *Choices* is simple and general. The major classes involved are **Process**, **ProcessContainer**, and **CPU**. Instances of **Process** manage all the information associated with a process, such as its address space and scheduling priority. An instance of **ProcessContainer** is simply a queue or other data structure that maintains an ordered group of Processes. ProcessContainers manage queues of "ready" and "blocked" processes[**?**]. The **CPU** class contains a method called `idleLoop()`, which is executed whenever there is no other process to run. This method waits until the CPU's ProcessContainer contains a Process, then removes and dispatches it.

Each Process is associated with a ProcessContainer where it waits when it is ready for execution. Similarly, each CPU is associated with a ProcessContainer where it gets the next process to execute. Note that there may be one or several ProcessContainers in the system. In a multiprocessor system, CPUs may be associated with different ProcessContainers to dedicate certain processors to real-time processes, to partition the machine between several groups of users, or to reduce contention at a central scheduler.

Each Process has an associated ProcessorContext, an object saves register contents when the process is not executing. A concrete subclass of **ProcessorContext** is all that is required to port the process scheduling framework to a new processor. The rest of the framework contains no processor-dependent code.

2.2 Exceptions

Exceptions are synchronous or asynchronous events that cause the normal execution of a process to be temporarily suspended while the exception is processed. Some exceptions (interrupts) are caused by I/O devices that require service; others are caused by software events such as attempts to execute illegal instructions.

Application processes and kernel processes execute in different protection domains, so there must be controlled ways to transferring into the kernel from an application and vice versa. Most processors support at least two privilege levels: a supervisor mode and a user mode. Application processes are executed with the processor in user mode, while kernel processes are executed in supervisor mode.

It is straightforward to change from the privileged supervisor mode to the unprivileged user mode, but the only way to change from user mode to supervisor mode is via an exception. Thus, the exception mechanism is used to provide a controlled entry into the kernel. This entry is used by applications to request system services as well as by I/O devices that require service.

The *Choices* exception handling model is based on the **CPU** and **Exception** classes. A CPU associates each hardware exception with an instance of the **Exception** class. The exception handler is written as a method in **Exception**. When an exception occurs, the processor arranges to temporarily suspend the current process and call the exception-handling method of the appropriate Exception.

To customize the exception handling framework to a particular processor, a concrete subclass of **CPU** is required. This class handles initialization details and provides an operating system entry point to receive control when exceptions occur. New subclasses of **Exception** are also required to handle processor-specific exceptions such as page faults and system calls.

2.3 Virtual Memory

Virtual memory presents a process with the illusion of an address space larger than the size of physical memory and also prevents a process from accessing memory used by the kernel or by other processes. The memory management hardware and the exception-handling mechanism mentioned above are used to maintain the integrity of the kernel and to protect application processes from each other. Processors support virtual memory in various ways, including using a simple TLB (translation lookaside buffer), a TLB with a hierarchical page table, and using variable-size segments.

The low-level virtual memory framework is defined using two abstract classes, **AddressTranslator** and **AddressTranslation**, that specify the interface that the physical processor must support. Subclasses of these abstract classes implement the interface on top of the facilities provided by the processor.

In the *Choices* virtual memory system, an instance of the **Domain** class maintains machine-independent information about the virtual address space of a process. This consists of a list of MemoryObjectCaches and the virtual address at which each one is mapped. Each Domain also maintains a pointer to an AddressTranslation, which manages the low-level processor-dependent information. A method in **Domain** is called to fix page faults and protection violations; this method uses machine-independent information to call **AddressTranslation** methods, which update the page tables or other processor data structures.

The virtual memory system is ported to a new processor type by supplying concrete subclasses of **AddressTranslator** and **AddressTranslation**. The **AddressTranslation** subclass maintains page tables or other processor-specific data structures, while the **AddressTranslator** subclass provides an interface to the hardware translation mechanism. Methods in this class are called to enable virtual address translation, to switch from one virtual address space to another, and to flush TLB caches.

Chapter 3

Process Scheduling

In order to make efficient use of the central processors, most operating systems interleave the execution of many processes. When a process is unable to continue execution (because it requires I/O or other services), it is temporarily suspended and another process is run in its place. The processors of the system are shared among all processes that are ready to execute[**?**, **?**, **?**]. When the execution of a process is suspended to execute another process, the state of the first process must be preserved so that it can be resumed at a later time. The activity of suspending one process and resuming another is called *process scheduling* or *context switching*. This chapter examines the portability of the *Choices* process scheduling framework by detailing those parts that are processor-dependent.

Processes in many traditional operating systems, including UNIX, execute in separate address spaces[**?**]. The address space of each process is protected from access by other processes using virtual memory hardware. A process can interact with another process only by using services provided by the operating system.

In these systems, the state of a process includes not only the state of the processor itself, *i.e.*, the contents of processor registers, but also the contents of the entire address space in which the process executes. To switch between processes, both the register set and the address space of each process must be saved and restored. Usually the address space is saved by using the virtual memory mechanism to map the address space of a process into parts of the physical address space reserved for the process.

The time spent by the operating system in switching between processes is purely overhead; it does not advance the state of any computation being performed by the processes. Therefore, minimizing the time required for a context switch is an important concern in the design of operating systems.

The execution time cost of a context switch can be considered in two parts: the time required to save and restore register contents, and the time required to switch from one virtual address space to another. The cost of saving register contents is largely determined by the size of the register set of the processor, which is not under the control of the operating system. In many systems, the bulk of the cost of a context switch is due to switching virtual address spaces, because this can require flushing processor caches. The resulting reduction in cache performance often outweighs the cost of other housekeeping performed by the kernel[**?**].

Some systems, including *Choices*, provide a special type of process, sometimes called a lightweight *thread*, that can share its address space with other threads, eliminating the need to flush processor caches when switching between threads in the same address space. The thread model foregoes the protection advantages of separate address spaces but can lead to effective and efficient decompositions of many types of computations.

3.1 *Choices* Process Scheduling

The *Choices* process model supports both lightweight threads and traditional heavyweight processes. The notion of a virtual address space is represented by the class **Domain**. The process concept is represented separately by the **Process** class. Each process, of course, must have an associated Domain in which it runs, but the Domain may be partially or completely shared with other processes. Each Process is also associated with an instance of the **ProcessorContext** class. These objects store the contents of processor registers when the process is not executing on a physical processor. Unlike Domains, ProcessorContexts cannot be shared with other Processes.

The **Process** class is entirely portable and does not need to be changed when porting *Choices* to a new machine. Part of the declaration of the **Process** class is shown in Figure 3.1. Processor-dependent details, such as the set of registers that must be saved when a process is suspended, are isolated into subclasses of the **ProcessorContext** class.

```
class Process {
protected:
    Domain * _domain;
    ProcessorContext * _context;

    virtual void save();
    virtual void restore( Process * oldProcess );

public:
    void giveProcessorTo( Process * newProcess );
    ProcessorContext * context();
};
```

Figure 3.1: The **Process** Class

The abstract **ProcessorContext** class, shown in Figure 3.2, declares an interface consisting of operations that manipulate the set of processor registers. The actual set of registers and the implementations of the operations for a particular processor are defined in concrete subclasses of **ProcessorContext**. When porting *Choices* to a new processor type, only a new subclass of **ProcessorContext** needs to be written.

```
class ProcessorContext {
public:
    virtual Process * checkpoint() = 0;
    virtual void restore( Process * oldProcess ) = 0;
};
```

Figure 3.2: The **ProcessorContext** Class

The `checkpoint()` and `restore()` methods of **ProcessorContext** are called by the `give-ProcessorTo()` method in class **Process**. The `giveProcessorTo()` is the crucial method for context switching. An outline of this method is given in Figure 3.3. A process calls this method to give up the processor and allow another process to resume execution. The first process's call to `giveProcessorTo()` does not return until some other process has called `giveProcessorTo()` to resume execution of the first process.

Choices supports several different types of processes, including kernel processes that run in supervisor mode with interrupts either enabled or disabled, and application processes that run in user mode with interrupts enabled. The different types of processes have slightly different characteristics, a fact that *Choices* uses for two context-switching optimizations.

The most important optimization is to avoid switching virtual address spaces (Domains) unless it is necessary, thus avoiding the penalty of flushing any processor caches. Every Domain in the system (including those of application processes) includes the kernel Domain, so kernel

```
void
Process::giveProcessorTo( Process * newProcess )
{
    save();
    Process * oldProcess = _context->checkpoint();
    if( oldProcess == 0 ) {
        newProcess->context()->restore( this );
        Assert( NOTREACHED );      /* We never get here */
    } else {
        restore( oldProcess );
    }
}
```

Figure 3.3: The `Process::giveProcessorTo()` Method

processes can run in any Domain, making it unnecessary to switch Domains when switching to a kernel process. It is also unnecessary to switch Domains when switching to an application process that happens to run in the current Domain.

This optimization is implemented by redefining the **restore()** method in subclasses of **Process**. The `restore()` method in **SystemProcess**, the class that describes kernel processes, does nothing. The **restore()** method of **ApplicationProcess** checks the current Domain of the processor and activates the Domain of the new process only if it is different.

The second context-switching optimization used in *Choices* takes advantage of the differing register usage of various processes. Processes that do not use floating-point arithmetic never use any floating-point registers, so it is unnecessary to save and restore them. One of the assumptions made in the kernel is that no system processes perform floating-point math, so **ProcessorContext** subclasses associated with SystemProcesses need not store floating-point registers. Often it is possible to use a similar optimization for application processes as well, if the processor can trap on the first floating-point operation.

3.2 The MC68030SystemContext Class

```
class MC68030SystemContext : public ProcessorContext {
protected:
    int _d2;
    int _d3;
    int _d4;
    int _d5;
    int _d6;
    int _d7;
    int _pc;                            /* stashed in %a1 */
    int _a2;
    int _a3;
    int _a4;
    int _a5;
    int _a6;
    char * _supervisorStackPointer;  /* %a7 */

public:
    virtual Process * checkpoint();
    virtual void restore( Process * oldProcess );
};
```

Figure 3.4: The **MC68030SystemContext** Class

The **MC68030SystemContext** class, shown in Figure 3.4, is a concrete subclass of **ProcessorContext**. The instance variables declared in the class correspond to the registers of the MC68030.

The `checkpoint()` and `restore()` methods are essentially the same as the UNIX library functions `setjmp()` and `longjmp()`[?]. Like `setjmp()`, `checkpoint()` appears to return twice, while the `restore()` method, like `longjmp()`, does not itself return, but appears to cause `checkpoint()` to return the second time. The code for `checkpoint()`, given in Figure 3.5, simply saves the contents of most of the processor registers in a MC68030SystemContext, clears register `%d0` (which holds the return value of a function), and returns.

The restore() function, shown in Figure 3.6, is the one that performs the "magic." It restores the register contents from a MC68030SystemContext object and returns to the program counter saved by a previous invocation of checkpoint().

```
    .globl  checkpoint__20MC68030SystemContext
checkpoint__20MC68030SystemContext:
    movel   %sp@(4),%a0             /* get "this" pointer */
    movel   %sp@,%a1                /* save return address in %a1 (_pc) */
    moveml  %d2-%d7/%a1-%a7,%a0@(_d2_MC68030SystemContext)
    clrl    %d0                     /* return 0 */
    rts
```

Figure 3.5: The MC68030SystemContext::checkpoint() Method

```
    .globl  basicRestore__20MC68030SystemContextP7Process
basicRestore__20MC68030SystemContextP7Process
    movel   %sp@(4),%a0           /* get "this" pointer */
    movel   %sp@(8),%d0           /* return Process * in %d0 */
                                  /* (guaranteed to be non-zero) */
    moveml  %a0@(_d2_MC68030SystemContext),%d2-%d7/%a1-%a7
                                  /* NB: we just switched stacks */
    movel   %a1,%sp@              /* stash %pc in return address */
    rts                           /* checkpoint() "returns" again! */
```

Figure 3.6: The MC68030SystemContext::basicRestore() Method

The instruction in basicRestore() that restores the stack pointer from the value saved in the MC68030SystemContext is important because the supervisor stack pointer determines which process is currently executing. Before the new stack pointer is loaded, the current process is the one that called giveProcessorTo(). After the new stack pointer is loaded, the current process is the one passed as the argument to giveProcessorTo().

The ten instructions in checkpoint() and basicRestore() are the only processor-dependent code that is critical to the performance of the process scheduling system. Though there are no actual performance statistics for this code, the number of instructions compares favorably with the code for other versions of *Choices*. For example, the corresponding methods in the Multimax implementation contain twice as many instructions (mainly because the NS32332 lacks a "move multiple" instruction).

Just after a ProcessorContext is restored, the `Process::restore()` method is called (see Figure 3.3. This method takes care of any housekeeping associated with the old process (usually this means adding it to a ProcessContainer), and switches to the Domain of the new process. The `Process::restore()` method manipulates the *old* process in some way, but is executed by the *new* process, *i.e.*, on the supervisor stack of the new process. Having the new process dispose of the old process avoids a race condition in the kernel. If the old process were to add itself to a ProcessContainer, another processor might remove and begin executing it while the process was still executing on the first processor. If this were to occur, both processors would be executing on the same supervisor stack, with totally unpredictable results.

3.3 Summary

With a good understanding of the `setjmp()` and `longjmp()` functions and the compiler calling convention, porting the process scheduling code to a new machine is straightforward. Only a concrete subclass of **ProcessorContext** needs to be written, and this class is nothing more than a slight adaptation of `setjmp()/longjmp()`.

The process of porting *Choices* to the MC68030 involved writing a **MC68030ApplicationContext** class in addition to the **MC68030SystemContext** presented above. The **MC68030ApplicationContext** class redefines `checkpoint()` to save floating-point registers and the user stack pointer in addition to those saved by the `checkpoint()` method in

MC68030SystemContext. The `checkpoint()` and `basicRestore()` methods for these two classes are written in assembly language and comprise a total of about 20 assembly instructions.

The files `MC68030Context.h` and `MC68030Context.cc` contain about 240 lines of C++ code, not including comments and white space. Most of the code is "boilerplate," copied from corresponding files for the NS32332 with minor changes for register names. The NS32332 files contain almost exactly the same amount of code.

Chapter 4

Exception Handling

An *exception* is a condition that causes a processor to stop normal program execution and perform special processing to handle the condition[**?**]. The *Choices* exception framework handles both *interrupts* and *traps*, the two types of exceptions recognized by most processors. This chapter includes details about the MC68030 implementation as an example of the portability of the framework.

Exceptions caused by external events, such as a hardware clock tick, the arrival of a packet on a network, or a keypress on a terminal, are called interrupts. Interrupts are unrelated to the current state of the processor and can occur at any time, though most processors can delay or disable recognition of interrupts.

Interrupts are most often used to interface with peripherals outside the processor itself. Peripherals such as disks, printers, and terminals are much slower than typical processors, so it would be inefficient for the processor to simply wait for the peripheral to complete an operation before proceeding. The delay between the request for a disk block and its arrival may be many milliseconds, enough time for the processor to execute tens of thousands of instructions. Rather than waste this time, the processor can continue with other tasks, returning to service the disk operation when an interrupt signals that the transfer has completed.

Other exceptions originate within the processor itself. All processors have one or more special instructions that are used to cause exceptions, and most processors generate exceptions when an illegal or undefined instruction is executed. These exceptions are known as traps. Unlike interrupts, traps are caused by the execution of a specific instruction and cannot be ignored.

Processors must support at least two privilege levels to run *Choices*. The kernel executes at the most privileged level, while applications run at a lower privilege level. The hardware support for multiple privilege levels allows the kernel to protect itself from malicious applications. The exception-handling mechanism provides a controlled entry into the kernel from applications.

If the processor is in user mode when an interrupt or trap is recognized, the processor automatically switches to supervisor mode, so the special processing to handle exception conditions is always performed by the kernel. In most systems, applications request operating system services by executing trap instructions to enter the kernel. The kernel performs the requested service and returns to user mode to continue execution of the application.

4.1 *Choices* Exception Handling

A *Choices* kernel includes objects that represent intangible concepts such as exceptions as well as physical entities such as processors and disk drives. An Exception object encapsulates the special processing to be performed in response to a trap or interrupt. Exception handlers are written as virtual functions in subclasses of the abstract class **Exception**, shown in Figure 4.1.

```
class Exception {
protected:
    virtual void basicRaise( char * exceptionStackFrame ) = 0;

public:
    void raise( char * exceptionStackFrame );
};
```

Figure 4.1: The **Exception** Class

The virtual function `basicRaise()` is the exception handler and must be implemented by each concrete subclass of **Exception**. The function `raise()`, shown in Figure 4.2, is an "assist" function that simply performs the virtual function call to `basicRaise()`; this is useful if the exception handler is to be called from an assembler-language routine, because the exact mechanism for calling a virtual function is compiler-dependent.

```
void
Exception::raise( char * exceptionStackFrame )
{
    basicRaise( exceptionStackFrame );
}
```

Figure 4.2: The `Exception::raise()` Method

The argument passed to `raise()` and `basicRaise()` is a pointer to a stack frame that contains the information needed to resume normal processing after the exception handler has finished. Often part of the frame is constructed automatically by the processor, so its details are processor-dependent. Interrupt handlers generally ignore the stack frame, but some trap handlers need information from it to process the trap. For example, if the trap is a virtual memory fault, the relevant virtual address may be stored in the stack frame. If the trap was caused by an unimplemented instruction, the frame may contain the address of the instruction, which may be used by a trap handler to emulate the instruction.

Each physical processor in a *Choices* system is represented by an instance of the **CPU** class. When a CPU is initialized, it assigns an Exception object to each source of hardware exceptions. The CPU object also ensures that the `raise()` method of the appropriate Exception object is called when a hardware exception occurs. The functions that must be implemented by concrete subclasses of **CPU** are shown in Figure 4.3.

```
class CPU {
protected:
    virtual void chipInitialize() = 0;
    virtual void installExceptions() = 0;

public:
    virtual Exception * setException( int vector,
        Exception * exception ) = 0;
    virtual Exception * exception( int vector ) = 0;
};
```

Figure 4.3: Exception-related Parts of the **CPU** Class

Most processors identify hardware exceptions by integer *vector numbers*, so the `setException()` method associates an Exception object with a vector number. This method is mainly used by the `installExceptions()` method, called when the CPU is constructed, but it can also be used to temporarily change the exception handler for an exception. The `exception()` method returns the Exception currently associated with a vector number.

The `chipInitialize()` method performs any required initialization of a processor, such as loading vector table registers, enabling interrupts, starting hardware timers, and so on. After this method is called, the processor should be able to respond to exceptions.

In addition to the functions shown in Figure 4.3, subclasses of **CPU** must arrange to translate a hardware exception into a call of the `raise()` method of the appropriate Exception object. The response of most processors to an exception includes an unconditional branch to an operating system entry point, but the processor usually does not do any of the normal housekeeping associated with calling a C++ method, such as saving register contents and passing parameters. A subclass of **CPU** must provide code at the entry point, usually written in assembly language, that takes care of these details.

4.2 MC68030 Exception Handling

Part of the exception-handling process is carried out automatically by the MC68030 processor when it recognizes an exception condition. The details of the response to an exception vary, but the processor always saves some processor state, switches to supervisor mode, and branches to an operating system entry point. There is no code that performs these steps; they are built into the processor itself. The MC68030 processor performs the following steps when it recognizes an exception[**?**]:

- After making an internal copy of the `Status Register` (`%sr`), the processor sets the `S` bit in the `%sr` and (for interrupts only) updates the interrupt priority mask. After the `S` bit is set, the processor is at the supervisor privilege level and uses the supervisor stack.[1]

- An interrupt causes the CPU to perform an interrupt acknowledge cycle to read a *vector number* from external circuitry. For all other exceptions, the vector number is generated internally by the processor itself. This number identifies the type of exception that has occurred. For example, the vector number distinguishes among reset, illegal instruction, privilege violation, trap, and other exceptions.

[1]The MC68030 has provision for *two* supervisor stacks, the master stack and the interrupt stack, and the `M` bit of the `%sr` determines which is used when the `S` bit is set. *Choices* needs only one supervisor stack, so it always uses the interrupt stack and ignores the the master stack.

- The processor saves part of its state in an *exception stack frame* on the supervisor stack. The information stored in the frame depends on the type of exception, but always includes the internal copy of **%sr**, the current value of the **Program Counter** (**%pc**), and an offset into an exception vector table.

- Finally, the processor jumps to an operating system entry point. The entry point is found by using the vector offset into the exception vector table addressed by the **Vector Base Register** (**%vbr**).

The layout of the information saved automatically by the MC68030 is shown in Figure 4.4. This exception stack frame contains all the information needed by the processor to resume what it was doing before the exception was recognized. The **Vector Offset** stored in the stack frame is the offset into the exception vector table for the current exception. The **Format** field identifies the format of the hardware stack frame. The MC68030 can generate six different stack frames depending on the type of exception being processed; the information shown in Figure 4.4 is common to all frame formats. The exception stack frame is built on the supervisor stack. Figure 4.4 shows only part of the entire exception stack frame; the code at the operating system entry point saves some additional information.

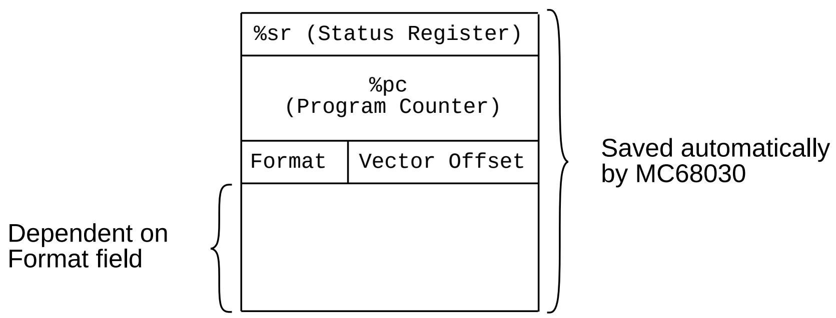

Figure 4.4: Format of MC68030 Hardware Exception Stack Frame

At this stage in the processing of an exception, the operating system entry point determined by the exception vector table receives control. This code, which is discussed in greater detail below, is responsible for calling the `raise()` method of the Exception associated with the current exception vector.

After the `raise()` method returns, the MC68030 executes a `rte`, or *return from exception*, instruction. This instruction restores the processor state from the exception stack frame, removes the entire frame from the supervisor stack, and resumes normal processing at the program counter saved in the stack frame. Note that since the `%sr` is part of the saved processor state, execution of the `rte` instruction may cause the processor to switch from supervisor mode back to user mode.

4.3 The MC68030CPU Class

The process of handling hardware exceptions is highly dependent on the CPU. The **MC68030-CPU** class contains most of the exception-handling code.

Instances of **MC68030CPU** maintain a table of Exceptions used by `setException()` and `exception()`. Each MC68030CPU also maintains an exception vector table. The address of this table is loaded into the `Vector Base Register` (`%vbr`). As mentioned above, the processor uses this table to find the operating system entry point for each type of hardware exception. These tables are depicted in Figure 4.5. The MC68030 allows for 256 exception vectors, so each table contains 256 entries. Figure 4.6 shows the parts of the **MC68030CPU** class definition involved in exception processing, including declarations for these tables.

The `installExceptions()`, `setException()`, and `exception()` methods need no explanation. The `chipInitialize()` method is called before interrupts are enabled and is responsible

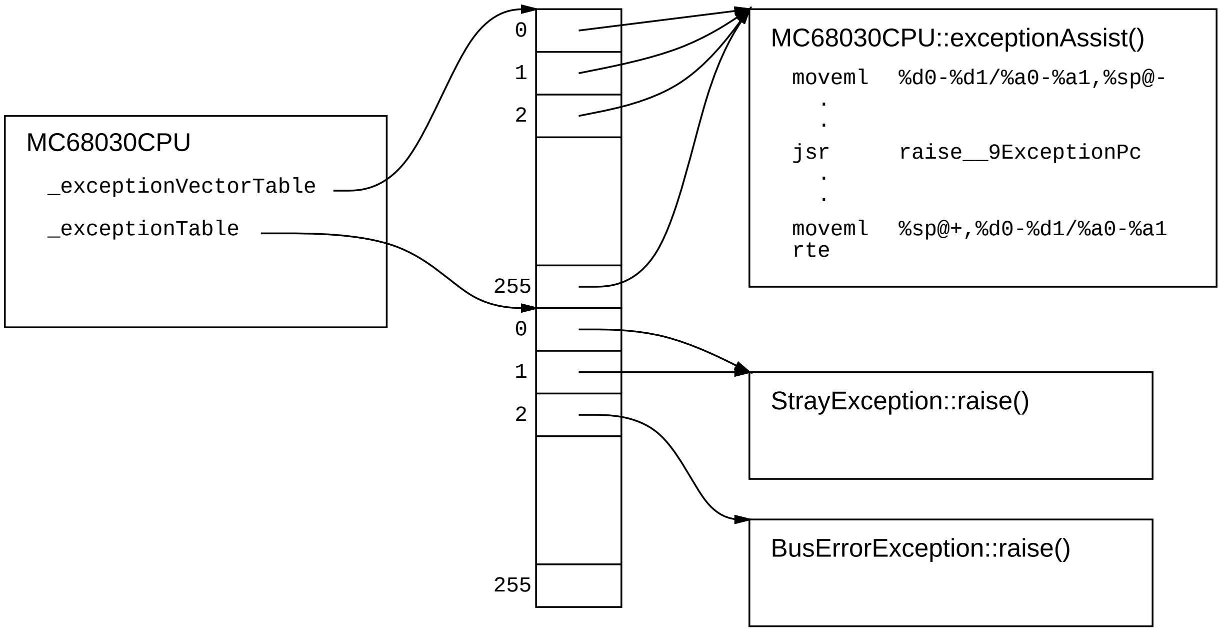

Figure 4.5: Exception-related Data of Class **MC68030CPU**

```
typedef void (*PFV)();      /* ptr to function of no args, returns void */

class MC68030CPU : public CPU {
protected:
    PFV * _exceptionVectorTable;     /* %vbr points to this table */
    Exception ** _exceptionTable;

    static void exceptionAssist();   /* calls Exception::raise() */

    virtual void chipInitialize();
    virtual void installExceptions();

public:
    virtual Exception * setException( int vector, Exception * exception );
    virtual Exception * exception( int vector );
};
```

Figure 4.6: Exception-related Parts of the **MC68030CPU** Class

for loading any processor registers needed to handle exceptions. The implementation in the **MC68030CPU** class, shown in Figure 4.7, simply loads the address of the exception vector table into the `%vbr` register. When an exception occurs, the MC68030 automatically jumps to one of the addresses in this table.

```
void
MC68030CPU::chipInitialize()
{
    asm volatile( "movec %0,%%vbr" : : "r" (_exceptionVectorTable) );
}
```

Figure 4.7: The `MC68030CPU::chipInitialize()` Method

The most interesting of the methods defined in **MC68030CPU** is `exceptionAssist()`. This method is written in assembly language and contains the first instructions executed after an exception is recognized. For MC68030CPU objects, each entry in the exception vector table contains the address of the same operating system entry point, `exceptionAssist()`. Because the exception stack frame contains the vector offset, `exceptionAssist()` can use it to find the proper Exception object. If the processor did not save the vector offset, a separate entry point would have to be used for each possible hardware exception.

The `exceptionAssist()` function is important because the exception must be completely transparent to the process that was interrupted. When normal processing is resumed, there must be no net change in the state of the processor. In particular, the contents of all registers, including the supervisor stack pointer, must be exactly the same as before the exception.[2] This method performs the following actions:

[2] Some traps are used to request operating system services; these may place return values in registers. Also, certain exceptions, such as divide-by-zero and illegal instruction exceptions, may cause the process to be terminated.

- Completes the exception stack frame by saving the contents of any registers that might be changed by `exceptionAssist()` itself or by the `raise()` method of an Exception.
- Determines the Exception object corresponding to the hardware exception currently being processed.
- Calls the `raise()` method of the Exception.
- Restores the register contents saved previously and executes a `rte` (*return from exception*) instruction to clean up the rest of the exception stack frame and resume normal processing.

The complete implementation of the `exceptionAssist()` method is shown in Figure 4.8.

```
    .globl  exceptionAssist__10MC68030CPU
exceptionAssist__10MC68030CPU:
    moveml  %d0-%d1/%a0-%a1,%sp@-     /* save volatile registers */

    movew   %sp@(vector_exceptionStackFrame),%d0   /* get vector offset */
    andiw   #0x0fff,%d0              /* mask off format identifier */
    movec   %vbr,%a0                 /* Exception at %vbr + 1024 */
    movel   %a0@(1024,%d0:w:1),%d0   /* get pointer to Exception */

    movel   %sp,%sp@-                /* push pointer to stack frame */
    movel   %d0,%sp@-                /* push Exception "this" pointer */
    jsr     raise__9ExceptionPc
    addql   #8,%sp                   /* discard raise() parameters */

    moveml  %sp@+,%d0-%d1/%a0-%a1    /* restore volatile registers */
    rte
```

Figure 4.8: The `MC68030CPU::exceptionAssist()` Method

If `raise()` were arbitrary assembly code, `exceptionAssist()` would have to save and restore the contents of *all* user-visible registers to ensure their preservation. However, in *Choices*, these methods have been compiled with a known calling convention. Under this convention, a function may destroy the contents of registers `%a0`, `%a1`, `%d0`, and `%d1`, but must preserve the contents of all other registers. Therefore, **exceptionAssist()** is required to save only the contents of registers **%a0**, **%a1**, **%d0**, and **%d1**. The **raise()** methods themselves are responsible for preserving all other registers. Of course, it is always safe to save *more* than the minimal set of registers if the calling convention is unknown.

After saving the volatile registers, **exceptionAssist()** maps the exception vector into the corresponding Exception. The **MC68030CPU** class allocates the Exception table immediately after the exception vector table, so the **%vbr** register can be used to locate both tables. The offset into the exception table is the same as the offset into the exception vector table, so the offset stored in the hardware exception stack frame can be used directly.

After locating the appropriate Exception, **exceptionAssist()** calls the **raise()** method. It is complicated to call a virtual function such as **basicRaise()** directly from assembly language because the function address must be looked up in table of virtual functions. Furthermore, the index in the table may change if virtual functions are added to or removed from the class. To sidestep these problems, **exceptionAssist()** calls **raise()**, a non-virtual C++ function that simply calls **basicRaise()**, the virtual function that implements the exception handler.

As Figure 4.8 shows, eight instructions are executed between after the exception is recognized and before **raise()** is called. The virtual function call performed by **raise()** adds another seven, for a total of fifteen instructions executed before the first instruction of the exception handler proper. The corresponding path in the Multimax version of *Choices* includes thirteen instructions, so the two implementations should have similar performance.

The exception stack frame contains the information necessary to resume normal processing after handling the exception. For traps, the stack frame may contain additional information about the cause of the trap. For example, the stack frame generated when the processor takes a privilege violation exception includes the memory address that caused the fault and the type of access that was attempted. A pointer to the stack frame is passed to the **`raise()`** method so it can use this information. The layout of a complete MC68030 exception stack frame is shown in Figure 4.9. Part of the stack frame is generated automatically by the processor, and the contents of registers **`%a0`**, **`%a1`**, **`%d0`**, and **`%d1`** are pushed by **`exceptionAssist()`**.

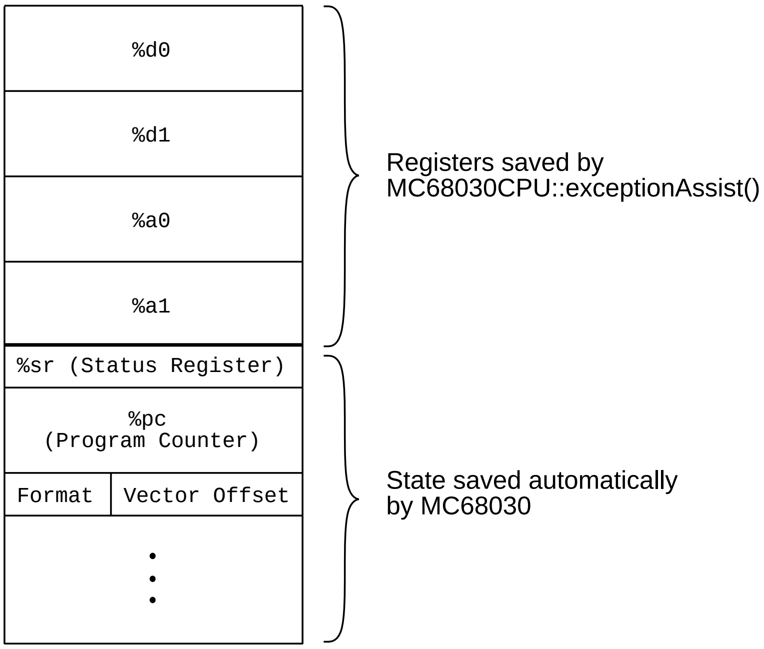

Figure 4.9: Format of MC68030 Exception Stack Frame

The format of the exception stack frame varies widely between processor architectures, so **`basicRaise()`** methods that use information from the frame are processor-dependent. Many **`basicRaise()`** methods, such as those for handling interrupts, don't need any information from the frame, so they are processor-independent.

4.4 Summary

The exception-handling system is perhaps the easiest part of *Choices* to port. A concrete subclass of **CPU** must be written to implement the `chipInitialize()`, `installExceptions()`, `setException()`, `exception()`, and `exceptionAssist()` methods. Most of these are trivial, and only `exceptionAssist()` must be written in assembly language.

The compiler calling convention determines the registers that `exceptionAssist()` must save, just as it determines those that must be saved by `ProcessorContext::checkpoint()`. The two sets are disjoint; registers saved by `exceptionAssist()` need not be saved by `checkpoint()`, but every register must appear in one set or the other.

The `MC68030CPU.h` and `MC68030CPU.cc` files contain about 250 lines of C++ code. In addition, `MC68030ContextSwitching.s` has about 10 assembler instructions related to exception handling. The amount of code required for the NS32332 is similar. Both processors have quite traditional exception-handling schemes so the *Choices* framework maps onto the processor architecture in an obvious way.

Chapter 5

Virtual Memory

Modern operating systems use virtual memory hardware for two purposes[**?**, **?**, **?**]. First, virtual memory provides the illusion that the system contains more memory than it really does, allowing the system to run larger programs than would otherwise be possible. Second, virtual memory hardware prevents processes from interfering with other processes or the operating system itself.

In a virtual memory system, addresses generated during the execution of a process, such as data and instruction references, are termed *virtual* addresses. These addresses are converted by a translation function into *physical* or *real* addresses, which are used to address the memory in the machine. The set of valid physical addresses is determined by the amount and configuration of the memory installed in the machine, while the set of valid virtual addresses is determined by the function that translates virtual to physical addresses.

Because the translation function can be different for each process in the system, the set of valid virtual addresses can also be different for each process and the same virtual address can be translated into a different physical address for each process. This feature keeps one process from accessing the memory of other processes.

The virtual-to-physical address translation function is expressed by a data structure maintained by the operating system. By modifying the data structure, the operating system can use *backing storage* such as magnetic disks to store infrequently-used portions of main memory while presenting the illusion that everything is in real memory.

It would be possible to implement such a scheme entirely in software, but since the translation occurs for memory reference made by a process, the overhead would be enormous. Therefore, most machines provide a hardware *memory management unit*, or MMU, that performs the translation from virtual to physical addresses with the aid of data structures maintained by the operating system. The memory management hardware generates an exception if an attempt is made to reference an invalid or protected virtual memory address.

Even with an MMU, each virtual address reference may require two or more additional memory references to perform the translation, so MMUs maintain a cache of recently-used translations, called a *translation lookaside buffer* (TLB). The MMU searches the TLB first, and if it does not contain the desired virtual address, the MMU searches the translation data structure in main memory. Since the entries in the TLB are valid only for a single translation function, it must be flushed when the processor switches to a different translation function.

This chapter covers the *Choices* framework for low-level virtual memory and its machine-independent interface. We present an overview of the two-level page table translation scheme, a system used by many common processors, as one possible implementation of the interface. Examples from the port to the MC68030 show how the framework is customized to a particular processor.

5.1 *Choices* Virtual Memory

In many operating systems, such as UNIX, each process has its own independent virtual address space[**?**, **?**]. A *Choices* process, on the other hand, runs in a virtual memory environment called a Domain, parts of which may be shared by several processes. All SystemProcesses share a single Domain known as the KernelDomain. The KernelDomain includes all the instructions and data that make up the operating system.

Another feature of *Choices* memory management is that the virtual memory regions that make up the instructions and data of the operating system are included in the Domain of *every* process in the system, including all ApplicationProcesses. These operating system areas are protected from access by applications by the virtual memory protection mechanism, but can be freely accessed by kernel code running in supervisor mode.

Choices takes advantage of the fact that all SystemProcesses run in the KernelDomain and the fact that all Domains include the KernelDomain to avoid changing the translation function and flushing the TLB when switching to the Domain of a SystemProcess.

```
class AddressTranslation {
public:
    virtual void addMapping( const char * virtualAddress,
        PhysicalMemoryChain * chain, ProtectionLevel protection ) = 0;
    virtual int basicChangeProtection( const char * virtualAddress,
        unsigned int length, ProtectionLevel protection ) = 0;
    virtual void basicRemoveMapping( const char * virtualAddress,
        unsigned int length ) = 0;
};
```

Figure 5.1: The **AddressTranslation** Class

The **Domain** class, which provides high-level operations such as adding and removing MemoryObjects and repairing page faults, is implemented using the abstract **AddressTranslation** class. A concrete subclass of **AddressTranslation** class implements the interface shown in Figure 5.1 by manipulating low-level machine-dependent data structures. For example, one subclass might implement the operations using page tables while another might directly modify TLB entries. Subclasses of **AddressTranslation** can often take advantage of the sharing of kernel MemoryObjects to reduce the size of these data structures.

The interface to the memory management hardware provided by the machine is defined by the class **AddressTranslator**, shown in Figure 5.2. The operations defined on an AddressTranslator include enabling address translation, flushing entries from the TLB, and inquiring about the cause of a virtual memory exception.

```
class AddressTranslator {
public:
    virtual void basicActivate( AddressTranslation * translation ) = 0;
    virtual void enable() = 0;
    virtual AddressTranslationFailureCode faultType( const char * address,
        unsigned int statusWord, AddressTranslation * translation,
        AccessType & accessType ) = 0;
    virtual void flushCache( const char * virtualAddress,
        unsigned int length ) = 0;
};
```

Figure 5.2: The **AddressTranslator** Class

5.2 MC68030 Virtual Memory

The MC68030 processor has an integral MMU[**?**] that implements a paged virtual memory system[**?**, **?**]. In such a system, physical memory is divided into equal-sized chunks called *frames*. The MC68030 MMU allows various frame sizes from 256 to 32,768 bytes, but the entire system must use the same frame size. The virtual address space is divided into chunks called *pages*, which are the same size as the frames.

The function that relates virtual to physical addresses is expressed as a tree of translation tables. These tables are stored in main memory and are automatically searched by the MMU as needed. The MC68030 MMU can search translation trees involving as many as five levels of tables, but the current *Choices* implementation uses only two-level trees. Figure 5.3 is a picture of such a tree. The physical address of the top-level table is maintained in an MMU register called a *root pointer*. Entries in the top-level table, or *pointer table*, point to second-level tables, or *page tables*. Page table entries contain physical addresses of frames.

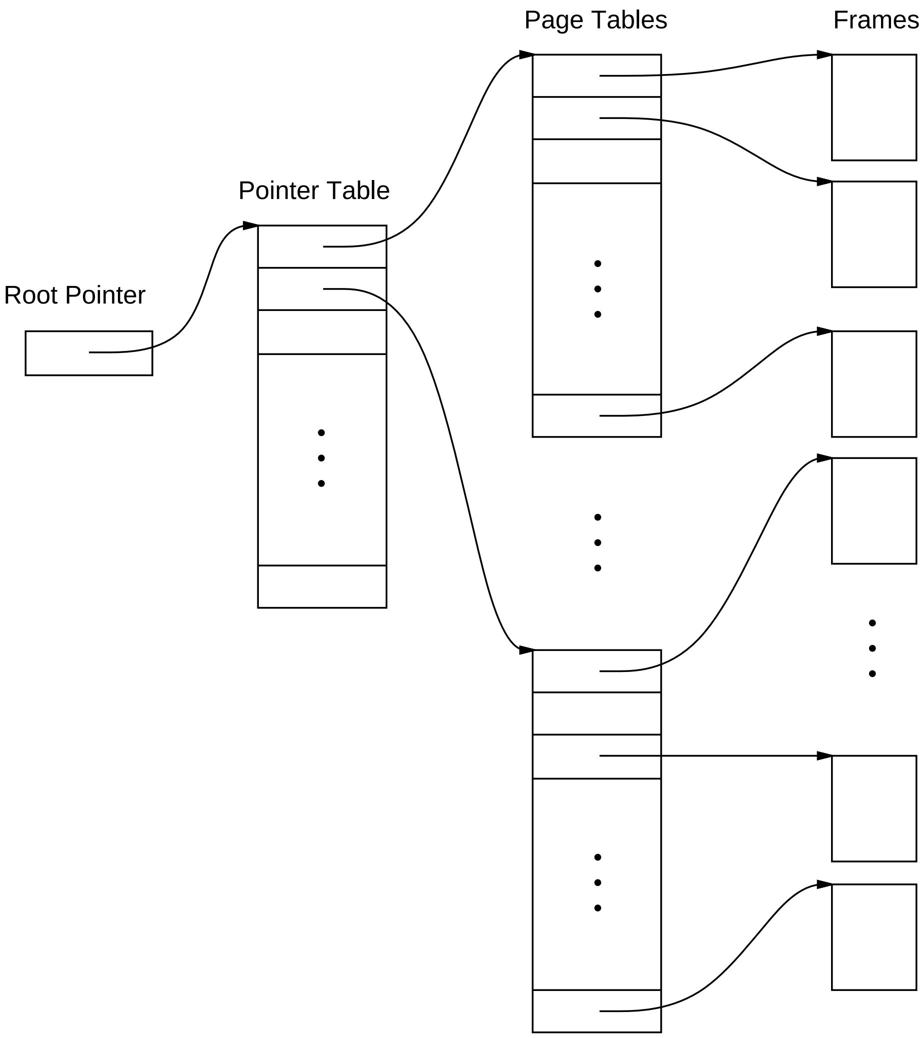

Figure 5.3: A Two-Level Translation Tree

For translation using a two-level page table, a virtual address is broken into three parts: an index into the pointer table, an index into a page table, and an offset into a page. The sizes of these parts are determined by the page size and the sizes of the pointer and page tables. The MC68030 version of *Choices* uses 4096-byte pages, and the pointer and page tables each contain 1024 entries. Therefore, a virtual address is divided into a 10-bit pointer table index, a 10-bit page table index, and a 12-bit offset into a page. The entries in the pointer and page tables are all 4 bytes long.

The following is an outline of the steps performed by the MMU to translate a virtual to a physical address:

- The MMU retrieves an entry from the pointer table. The address of this entry is computed by multiplying the high-order 10 bits of the virtual address by 4 (the size of pointer table entries) and adding the result to the contents of the root pointer.

- If the pointer table entry is invalid, the virtual address is invalid and the MMU aborts the current instruction by generating a Bus Error (page fault) exception.

- If the pointer table entry is valid, it contains the address of a page table. The MMU retrieves an entry from the page table. The address of the page table entry is computed by multiplying the second 10 bits of the virtual address by 4 (the size of page table entries) and adding the result to the base of the page table.

- If the page table entry is invalid, the MMU signals a Bus Error exception.

- If the page table entry is valid, it contains the physical address of a frame of main memory. The final physical address is computed by adding the low-order 12 bits of the virtual address to the frame address.

Two-level page tables are used in the virtual memory systems of many microprocessors, including the Motorola 68030, the National Semiconductor 32332 using the NS32382 MMU[**?**], and the Intel 80386[**?**]. The only differences are in the format of the pointer table entries and the page table entries. In *Choices*, the **TwoLevelPageTable** class, a concrete subclass of the **AddressTranslation** class, is used for all of these processors. Trivial classes are defined to represent the pointer table entries and page table entries of each processor, and instances of these classes are used by the **TwoLevelPageTable** class.

The **TwoLevelPageTable** class takes advantage of the fact that the virtual address space of the kernel is mapped into *every* Domain at the same address. Therefore, the (second-level) page tables for the kernel's address space are identical for every AddressTranslation. By keeping track of the areas of virtual memory used by the kernel, the **TwoLevelPageTable** class can allocate a single set of these page tables and share them among all instances.

Until now, we have not mentioned any aspects of protection. The page table entries usually contain several bits of information in addition to the physical address of a frame of memory. This information includes one or more *protection* bits, a *referenced* bit, and a *modified* bit.

The protection bits are checked by the MMU as it performs an address translation and can allow or disallow a memory access based on the current privilege level of the processor and the type of access being attempted. For example, the bits might allow access only while the processor is in supervisor mode, or might allow only read accesses. Attempts to perform a disallowed access cause the MMU to generate a Protection Violation exception. Protection bits are maintained by the operating system; they are never changed by the MMU itself.

The referenced and modified bits, on the other hand, are maintained by the MMU. The referenced bit in a page table entry is set by the MMU whenever the processor references a location in the frame addressed by the entry. Similarly, the modified bit in a page table entry is set whenever the processor writes to a location in the frame addressed by the entry.

5.3 The MC68030Translation Class

The **TwoLevelPageTable** class implements two-level page tables that are general enough to be used with several processors, including the NS32332, the 80386, and MC68030. In addition to the **TwoLevelPageTable** class itself, two small classes are defined for each processor. These classes implement the details of manipulating entries in the pointer and page tables.

For the MC68030, the classes defined are **MC68030PointerTableEntry** and **MC68030-PageTableEntry**. Conceptually, these classes should be subclasses of an abstract **PageTable-Entry** class with operations implemented as virtual functions, but this is not done in this case. The reason is that the page table format must correspond *exactly* to that expected by the MMU, since it searches the page tables directly. Instances of classes with virtual functions need an additional pointer for the run-time virtual function lookup, and there is no room for this pointer in the page table structure expected by the processor.

```
class MC68030PointerTableEntry {
public:
    void mapToPageTable( MC68030PageTableEntry * pageTable );
    MC68030PageTableEntry * pageTable();
    unsigned int valid();
    void setProtection( ProtectionLevel protection );
};
```

Figure 5.4: The **MC68030PointerTableEntry** Class

Methods of the **TwoLevelPageTable** class take care of most of the details of maintaining page tables, so classes for pointer- and page-table entries are simple. The **MC68030Pointer-TableEntry** and **MC68030PageTableEntry** classes are shown in Figures 5.4 and 5.5.

```
class MC68030PageTableEntry {
public:
    void mapToAddress( const char * address, ProtectionLevel protection );
    const char * addressMappedTo();
    void unmap();
    unsigned int valid();
    unsigned int referenced();
    unsigned int modified();
    void setProtection( ProtectionLevel protection );
    int okToPerform( AccessType attemptedAccess );
};
```

Figure 5.5: The **MC68030PageTableEntry** Class

The methods defined by these classes are small; as examples, the implementations of `mapToPageTable()` and `mapToAddress()` are shown in Figures 5.6 and 5.7. Given the details of the page table format for a particular processor, writing these functions is usually a trivial task.

The `mapToAddress()` method and a few others are complicated by use of protection bits. *Choices* assumes three machine-independent levels of protection, must be implemented using whatever levels of protection are supported by the memory-management hardware. The machine-independent protection levels are from the perspective of *user-mode* processes; code running in supervisor-mode can always perform any memory operations. The machine-independent protection levels are summarized in Table 5.1.

Many memory-management units can enforce the machine-independent protection levels directly. Unfortunately, the MC68030 MMU does not support the **ReadOnly** level. MC68030

```
inline void
MC68030PointerTableEntry::mapToPageTable( MC68030PageTableEntry * table )
{
    entry = (unsigned int) table | 0x2;
}
```

Figure 5.6: The `MC68030PointerTable::mapToPageTable()` Method

```
inline void
MC68030PageTableEntry::mapToAddress( const char * addr,
    ProtectionLevel protection )
{
    MC68030Protection level = (protection == ReadWrite) ? READWRITE
                                                        : READONLY;
    entry = (unsigned int) addr | (level << 2) | 0x1;
}
```

Figure 5.7: The `MC68030PageTableEntry::mapToAddress()` Method

page table entries have two protection bits. One is a *write-protect* bit, which is effective for both supervisor and user mode. The other is a *supervisor-only* bit, which protects memory from any access by user-mode code. There is no combination of these two bits corresponding to the **ReadOnly** protection level.

Because the protection levels supported by the MC68030 MMU do not correspond directly to those assumed by *Choices*, the **TwoLevelPageTable** class cannot be used directly as the processor-dependent **AddressTranslation** class. The MC68030 MMU does, however, support the use of separate translation trees for user and supervisor mode. The **MC68030Translation**

	User mode	Supervisor mode
NoAccess	none	read, execute, write
ReadOnly	read, execute	read, execute, write
ReadWrite	read, execute, write	read, execute, write

Table 5.1: Access Allowed by Protection Level and Privilege Level

class makes use of this feature by using *two* instances of **TwoLevelPageTable**, one for each privilege level, and adding mappings to one or both tables as necessary. Every `addMapping()` call adds a **ReadWrite** mapping to the supervisor-mode table. **ReadOnly** and **ReadWrite** mappings are also added to the user-mode table with the write-protect bit set appropriately.

5.4 The MC68030MMU Class

The **MC68030MMU** class is an implementation of the interface defined by the **AddressTranslator** class. The `basicActivate()` method, shown in Figure 5.8, loads both the user-mode and the supervisor-mode root pointers.

```
void
MC68030MMU::basicActivate( AddressTranslation * translation )
{
    MC68030Translation * tables = (MC68030Translation *) translation;
    const POINTER_TABLE_ENTRY * system = tables->systemPointerTable();
    const POINTER_TABLE_ENTRY * user = tables->userPointerTable();

    int systemRoot[2];
    systemRoot[0] = RP_NO_LIMIT | RP_4_BYTE;
    systemRoot[1] = (int) system;

    int userRoot[2];
    userRoot[0] = RP_NO_LIMIT | RP_4_BYTE;
    userRoot[1] = (int) user;

    asm volatile( "pmove %0@,%%srp" : : "a" (&systemRoot) );
    asm volatile( "pmove %0@,%%crp" : : "a" (&userRoot) );
}
```

Figure 5.8: The `MC68030MMU::basicActivate()` Method

The `faultType()` method, shown in Figure 5.9, is the most complicated of the class. This method is called by the page fault exception handler to find out why the page fault occurred. Information from the exception stack frame is used to determine at what privilege level the processor attempted the faulting access and whether the attempted access was a read or write. The method uses this information to ask the AddressTranslation why such an access should cause a fault. The attempted access and reason for the fault are returned to the page-fault hander.

5.5 Summary

The virtual memory system is probably the most difficult part of *Choices* to move to a new machine. Much of the work is involved in determining exactly what caused a page fault or protection violation. Including the class declaration, the **MC68030MMU** class totals about 140 lines of C++ code.

Before the port to the MC68030, each processor required its own concrete subclass of **AddressTranslation**. For the NS32332, this class totaled about 650 lines of code. In the process of porting this code to the MC68030, which uses a similar translation scheme, the code was factored into the **TwoLevelPageTable** class and companion pointer table and page table entry classes. Only the table entry classes are dependent on the processor, so the number of NS32332-dependent lines of code dropped from 650 to about 200. Because the MC68030 MMU does not support the **ReadOnly** protection level, two parallel page tables are required. These are maintained by another 140 lines of code in **MC68030Translation**.

Processors using different translation mechanisms will require a completely different subclass of **AddressTranslation**, which would probably also require on the order of 500-1000 lines of C++ code.

```
AddressTranslationFailureCode
MC68030MMU::faultType( const char * address, unsigned int statusWord,
    AddressTranslation * at, AccessType & attemptedAccess )
{
    ADDRESS_TRANSLATION * translation = (ADDRESS_TRANSLATION *) at;

    int addressSpace = statusWord & 0x7;
    int dataFault = statusWord & 0x100;
    int instructionFault = statusWord & 0xc000;
    int readWrite = statusWord & 0x40;
    int user = ((addressSpace == 1) || (addressSpace == 2) ||
                (addressSpace == 3));
    int write = (dataFault && (readWrite == 0));

    attemptedAccess = user ? (write ? ApplicationWrite : ApplicationRead)
                           : (write ? SystemWrite      : SystemRead);

    if( dataFault && instructionFault ) {
        Console() << this << "::faultType(): Simultaneous data "
            << "and instruction faults!\n" << eor;
        Assert( NOTREACHED );
    }

    AddressTranslationFailureCode error = translation->
        determineError( address, attemptedAccess );

    if( error == NonResidentMemory ) {
        return( error );
    } else if( error == ProtectionViolation ) {
        return( error );
    } else if( error == MMUCacheMiss ) {
        Console() << this << "::faultType: A protection level "
            << "MMU cache miss occurred!\n" << eor;
        flushCache( address, 1 );
        return( error );
    } else {
        Assert( NOTREACHED );
    }
    return( error );
}
```

Figure 5.9: The `MC68030MMU::faultType()` Method

Chapter 6

Conclusions

In summary, the major parts of *Choices* that required porting effort included the process scheduling, exception handling, and low-level virtual memory subsystems. The object-oriented design of these areas eased the porting process by minimizing changes and maximizing code reuse. We review the changes required to port *Choices* from the NS32332 processor to the MC68030 and show that the processor-dependent code in *Choices* is isolated in a small part of the system.

The process scheduling framework requires a concrete subclass of **ProcessorContext**. The instance data of the subclass includes the set of registers that must be preserved across a procedure call (determined by the compiler calling convention), and the methods include `checkpoint()` and `basicRestore()`. These methods are slightly modified versions of the standard `setjmp()` and `longjmp()` UNIX library functions and are written in assembly language.

To port the exception handling framework to a new architecture, one must supply a concrete subclass of **CPU**. The instance data of the new **CPU** class includes a table of Exceptions, one for each hardware exception type. The method of interest is **`exceptionAssist()`**, the lowest-level operating system entry point, which must be written in assembly language.

The low-level virtual memory framework is probably the most difficult to move to a new machine. Concrete subclasses of **AddressTranslation** and **AddressTranslator** must be written. The **AddressTranslation** subclass must implement the **`addMapping()`**, **`basicChangeProtection()`**, and **`basicRemoveMapping()`** methods. These methods can all be written in C++. If the machine can use a translation scheme based on two-level page tables, the **TwoLevelPageTable** class eliminates the need to write an entire **AddressTranslation** subclass. To use **TwoLevelPageTable**, classes that represent the formats of pointer table and of page table entries are required. These classes are usually trivial; most methods are only one or two lines of C++ code.

The **AddressTranslator** subclass handles all operations directly involving the MMU of the machine, including the **`basicActivate()`**, **`enable()`**, **`faultType()`**, and **`flushCache()`** methods. All of these methods can be written in C++ with a few instructions of inline assembly code to reference MMU registers. The **`faultType()`** method is the most complicated because it determines exactly what caused a page fault or protection violation. This requires interpreting the exception stack frame or interrogating the MMU itself.

Although this version of *Choices* still lacks timing facilities, the number of instructions on the critical execution paths through the process scheduling and exception handling code indicates that the performance of those subsystems should be comparable to that measured on the Multimax[**?**]. Performance of the low-level virtual memory system is expected to be slightly poorer because two parallel page tables are used on the MC68030 to provide the functionality required by *Choices*.

Choices Subsystem	Lines of C++ Code Interface	Implementation
High-level Virtual Memory	5,300	19,000
Networking	5,100	9,000
File Systems	2,300	10,900
Kernel	1,900	5,800
Distributed Virtual Memory	800	3,600
Input/Output Devices	1,100	2,900
Multimax-Dependent	1,200	3,300
Macintosh-Dependent	800	3,400
NS32332-Dependent	600	1,300
MC68030-Dependent	500	2,100

Table 6.1: Lines of *Choices* Code by Subsystem

The small number of processor-dependent instructions in the frequently-executed process scheduling and exception handling code means that any significant optimizations in these areas will be done in the machine-independent code. This is important because such optimizations automatically benefit all platforms.

No changes were required to the interfaces of the **ProcessorContext**, **CPU**, **Exception**, **AddressTranslation**, or **AddressTranslator** classes. This indicates that the assumptions made in the process scheduling, exception handling, and low-level virtual memory frameworks are valid and portable, at least to common general-purpose processors such as the NS32332, MC68030, and Intel 80386. During the porting process, some similarities between processors, such as the use of page tables for virtual memory, were abstracted into processor-independent classes. These abstractions moved code into the machine-independent portion of *Choices*, where it can be easily reused for other processors.

Table 6.1 summarizes the relative sizes of *Choices* subsystems in terms of the number of lines of C++ code in header (interface) and other files. These figures do not include comments or other white space. Of the approximately 74,000 lines of code in each version of *Choices*, 67,600, or over 90%, are entirely machine-independent. The bulk of the remaining code is machine-dependent code that deals with bus architecture, I/O devices, etc. Only about 3% of the code is dependent on the processor itself, and nearly a quarter of this consists of class declarations, which are essentially identical for each version of *Choices*. The MC68030-dependent category in Table 6.1 is somewhat larger than the NS32332-dependent category; most of this difference is due to remote debugging code that has not yet been ported to the NS32332.

Although there are many parts of *Choices* that could benefit from reorganization, the already small amount of non-portable code is one indication that the system is well-factored and that processor and machine dependencies are well-isolated from the rest of the system.

Bibliography

[1] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Inc., 1986.

[2] Roy Campbell, Gary Johnston, and Vincent Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[3] Roy Campbell, Vicent Russo, and Gary Johnston. The design of a multiprocessor operating system. Technical Report UIUCDCS-R-87-1388, University of Illinois at Urbana-Champaign, Department of Computer Science, December 1987.

[4] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley Publishing Company, Inc., revised first edition, 1984.

[5] Intel Corporation, P.O. Box 58130, Santa Clara, California. *80386 Programmer's Reference Manual*, 1986.

[6] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, June 1988.

[7] Panagiotis Kougiouris. The I/O subsystem of an object-oriented operating system. Master's thesis, University of Illinois at Urbana-Champaign, 1991. Forthcoming.

[8] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, Inc., 1989.

[9] Jeffrey Franklin Mantei. Porting Choices to the HP 9000 series 800 workstation. Master's thesis, University of Illinois at Urbana-Champaign, 1991.

[10] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[11] Motorola, Inc. *MC68030 Enhanced 32-Bit Microprocessor User's Manual.* Prentice-Hall, Inc., third edition, 1990.

[12] National Semiconductor Corporation, 2900 Semiconductor Dr., P.O. Box 58090, Santa Clara, California 95052–8090. *Series 32000 Databook*, 1986.

[13] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, Inc., second edition, 1985.

[14] Marc J. Rochkind. *Advanced UNIX Programming.* Prentice-Hall, Inc., 1985.

[15] Vincent Frank Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[16] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, Inc., 1987.

[17] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach threads and the UNIX kernel: The battle for control. In *Proceedings of Summer Usenix*, June 1987.

[18] Ann L. Winblad, Samuel D. Edwards, and David R. King. *Object-Oriented Software*. Addison-Wesley Publishing Company, Inc., 1990.