Chapter 12
Operating Systems

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

– D. Ritchie

Operating systems are like underwear - nobody really wants to look at them.

– B. Joy

Abstract Although programs written for embedded platforms, for example, are often said to execute on “bare metal” (i.e., the unmanaged processor), more usually we write programs that do not execute in isolation or without assistance. More often, they execute on a managed platform which is supported by an Operating System (OS). The OS is a software layer tasked with management of the computer as a whole: it ensures that resources are shared between many executing processes, and performs certain low-level tasks on their behalf. In this chapter the aim is to focus on aspects of OS design and implementation which most directly relate to the hardware and software components the OS manages. We concentrate on UNIX-style concepts and approaches, particularly those which can be supported by SPIM, but the discussion is more generally applicable.

12.1 Introduction

An Operating System (OS) is a layer of software that exists to perform various management tasks. At the most basic level, the OS is tasked with loading a program image into memory and providing a context in which it executes as a process; we can identify more advanced tasks using some examples:

• Whether internal or external, events will occur that require the attention of the OS. Examples include some external device that needs to be attended to (e.g., the keyboard has a key press ready to be read), or an executing process that
needs the OS to perform some function on behalf of it (e.g., write output to a display device). As such, the OS needs to provide suitable mechanisms so it can be invoked to deal with such events.

- It is unusual for one process to have dedicated access to the components that constitute a processor. More commonly we would like to have several processes resident in memory, but allow them to coexist by sharing the processor and associated resources having one process active at a given point in time. It is easy to see this in action: a single computer might allow several users to login and execute programs at the same time, or one user might execute a word processor and web-browser at the same time. The OS needs to provide the mechanism by which this is possible.

- Many peripheral devices can be attached to a computer; examples include resources such as display devices, storage devices and so on. Somehow, a process executing on the processor needs to access those devices. To permit some level of portability, access to the devices should be somewhat abstract. For example, if a program needs to know the exact storage device it is accessing, it cannot be executed on a computer with another device. A better approach would be to have the OS provide the program with a generic view of all storage devices and manage the details of access behind the scenes.

- Given that many processes can be sharing the processor, there is a danger that one will influence the execution of another or, worse still, somehow influence the OS itself. For example, you might imagine that if one process accidentally writes values into memory at addresses that hold the instructions for some second process, the corrupted second process probably will not execute as expected. The OS provides mechanisms that protect processes from each other in this respect.

- There is also a chance that two processes sharing the processor will need to access the same resource at the same time. Sometimes this is not possible, for example perhaps the resource is a storage device which cannot read and write data at the same time: while one process is using it to write, another cannot instruct it to read. In this case the OS needs to act as an arbitrator between the processes. That is, and more generally, it needs to manage the resource in order that usage is functionally valid (i.e., works correctly for each process) and fair (e.g., does not indefinitely prevent one process from performing an access).

The problem is, our typical view of an OS is much wider ranging than we would like to investigate here. An “operating system” such as Microsoft Windows, for example, includes a huge range of auxiliary software and features (e.g., any Graphical User Interface (GUI), or software tools such as calculators and text editors) that we would like to ignore.

Adopting a purist perspective, we focus specifically on topics that relate to an OS kernel: the central component of the OS that most closely interacts with the hardware devices investigated up until now. We cover roughly three core topics: event management which details how the kernel is invoked to deal with different events that can occur, memory management which details how the kernel manages memory, the main resource attached to the processor that we previously discussed, and process management which details how the kernel manages the execution of
many processes. Roughly speaking the first two topics define the hardware platform which the OS manages on behalf of software that executes according to the third topic. There are clearly many other topics missing from this list; we motivate our choices by the fact that event, process and memory management place direct constraints on and rely upon features in the processor design.

Since SPIM [35], outlined in Appendix A, uses a simulation-based approach to executing programs, it does not model some features of a MIPS32-based processor which are required to support a full kernel; for example, it does not fully model the memory hierarchy. As a result, we adopt a practical approach to the topics it can support (e.g., we develop a real exception handler) but overview those it does not at a higher-level (e.g., virtual memory).

12.2 The Hardware/Software Interface

If the OS is tasked with managing the interface between hardware and software, this interface needs to include the ability for the kernel to inspect and control some aspects of how the processor operates. Before we discuss how the kernel implements said interface, we need to investigate the mechanism by which that inspection and control is exercised. For MIPS32 at least, this is defined in the Privileged Resource Architecture (PRA) which in some sense defines how the processor operates in “managed” mode and contrasts with the ISA which defines how the processor operates in “bare metal” mode.

12.2.1 MIPS32 Co-processor Registers

The MIPS32 ISA includes an interface to co-processor devices. One might view these as physically separate from the processor; for our purposes, however, we can view them simply as collections of special-purpose registers. We denote these in a similar way to the general-purpose register file so that \( CP0[i] \) is the \( i \)-th register within co-processor zero, or \( CP1[j] \) is the \( j \)-th register in co-processor one. Access to the registers is achieved by copying values to and from the general-purpose register file using dedicated instructions. For example, the \texttt{mtc0} and \texttt{mfc0} copy values to and from co-processor zero. As a result, we might write the assembly language fragment

```assembly
mfc0 $t0, $0 ; move CP0[0] into GPR[8]
... ; perform some processing of GPR[8]
mtc0 $t0, $0 ; move GPR[8] into CP0[0]
```

to read the value of \( CP0[0] \), perform some change to the value and then write this back to update the co-processor. The idea is that in reality this would represent communication between the processor and co-processor, potentially prompting the co-processor to perform some action on behalf of the processor.
### Figure 12.1
The sub-set of MIPS32 co-processor zero registers from Table 12.1 described as bit-fields.
Although they take different approaches to control of their behaviour, processors need somewhere to store configuration and some mechanism to inspect and update it. Although it supports up to three co-processors, a MIPS32-based processor dedicates co-processor zero for this role. Of the thirty two or so registers in CP0, some are optionally defined and only a few are central to our discussion; many control advanced features of the processor, see [44, Chapter 6] for more detail. Table 12.1 shows a list of pertinent registers, each has a mnemonic which we will use instead of the register number for clarity and is composed of a number of fields shown by Figure 12.1. The BadVAddr, Count, Compare and EPC registers are not very interesting in that they each consist of a single 32-bit field; Status, Cause and Config consist of many fields, mainly 1-bit flags. We will expand on the meaning of relevant fields in subsequent sections. For now it suffices to understand that when we refer to the BE field in Config, for example, we mean the 15-th bit of CP0[16]; this bit indicates the endianness mode the processor is operating in. For more detail, again see [44][Chapter 6].

There are three additional points of information not captured by the bit-field diagrams. Firstly, some fields are read/write in the sense that we can read their value and write a new value to update the register content, others are read only in the sense that we can only read their value. Secondly, in Status the combination of UM and R0 fields has the alias KSU so in some sense you can think of KSU1 ≡ UM and KSU0 ≡ R0. Thirdly, just like it only supports some co-processor registers, SPIM does not support all fields in all registers. For example, in Status roughly only those fields related to interrupt handling are supported.

### 12.2.2 MIPS32 Processor Modes

In common with other processors, MIPS32 permits four **processor modes** which one can also think of as **privilege levels**. The current mode is dictated by the value of various status registers. The processor is in:

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>BadVAddr</td>
<td>Address for most recent address-related exception.</td>
</tr>
<tr>
<td>9</td>
<td>Count</td>
<td>Cycle accurate time-stamp counter.</td>
</tr>
<tr>
<td>11</td>
<td>Compare</td>
<td>Timer interrupt control.</td>
</tr>
<tr>
<td>12</td>
<td>Status</td>
<td>Processor status.</td>
</tr>
<tr>
<td>13</td>
<td>Cause</td>
<td>Details of exception meaning.</td>
</tr>
<tr>
<td>14</td>
<td>EPC</td>
<td>Exception Program Counter (EPC).</td>
</tr>
<tr>
<td>16</td>
<td>Config</td>
<td>Processor configuration.</td>
</tr>
</tbody>
</table>

*Table 12.1* A table describing a sub-set of MIPS32 co-processor zero registers.
The kernel “owns” some region of memory which contains data and instructions that allow it to operate; we call this **kernel memory** to contrast with **user memory** which is “owned” by user processes. The basic idea is that depending on the current mode when it is executed, a given instruction will be allowed or disallowed access to a certain processor feature or region of memory. Intuitively this is a useful feature since it enables the kernel to protect resources it manages against misuse by executing processes. For example, it might be reasonable for changes to the processor configuration to be restricted to kernel mode (i.e., performed by the kernel) since if this were allowed in user mode (i.e., performed by a user process), one process could incorrectly configure the processor and perhaps cause another process to execute incorrectly.

Two questions arise from this description. Firstly, what is the consequence of an instruction executing in the wrong mode ? Above we said it could be disallowed access to some resource. In reality we would like to catch the event and have the kernel deal with it gracefully. This is achieved via the exception and interrupt mechanism described in the next section. Secondly, how does the processor enter and exit kernel mode ? If kernel mode is intended to implement some protection mechanism, what is to stop a process executing in user mode from simply changing the processor state to kernel mode and subverting that protection ? And if this change is disallowed, how can we ever enter kernel mode ? Again we will see that this is achieved via the exception and interrupt mechanism.

### 12.2.3 MIPS32 Assembly Language

At a simple level the kernel is just a special piece of software and as such we implement it using assembly language just like we might implement a program intended
to be run in user mode. The only real difference is the level and detail at which
we need to interact with the resources the processor offers. To cope with this, it is
useful to extend the discussion in Chapter 10 with some special-purpose assembler
directives.

In particular, we need to use `.kdata` and `.ktext` which are analogies of the
`.data` and `.text` directives already described in Chapter 10. Using `.data` and
`.text` allowed the programmer to mark sections of data and instructions so the
assembler placed them in relevant places within the object or executable file, and
allowed it to prevent errors such as the use of invalid content within a particular
section. The `.kdata` and `.ktext` directives work in the same way except they
specify kernel data and instruction sections. Both directives accept an optional ad-
dress argument that allows placement of data or instructions at a particular address.
For example, we can use the directive

```plaintext
.ktext 0x80000180
```

to place code at address 80000180.

A second set of new directives reduces how pedantic the assembler is when pre-
venting access to reserved registers. Whereas normally `GPR[1]` (or `$1`, or `$at`) is
reserved for use by the assembler, by issuing the directive

```plaintext
.set noat
```

we are able to use it as normal, e.g., write to `GPR[1]` without provoking an error.
Using the directive

```plaintext
.set at
```

again makes it reserved; the assembler will again produce an error if `GPR[1]` is
written to.

### 12.3 Boot-Strapping

You probably know the “chicken or egg” problem: given a chick hatches from an
egg laid by another chicken, how did the first egg (or chicken) come about? There is
a similar problem in the context of programs and their execution. When a computer
is powered on, the memory content is essentially empty (or at least just random)
so although one can reset the processor state to start executing a program as some
fixed address, the program image needs to be there in the first place. The problem
is, we already described the kernel as being responsible for management of loading
and execution of programs: the kernel is also a program but clearly it cannot load
itself! To solve this problem, it is common for a short **boot-strap** program to be
stored in non-volatile ROM. Often this forms part of very low-level configuration
called the **Basic Input/Output System (BIOS)** which allows direct control of how
hardware devices should behave. For example, the BIOS often includes options that
control how the memory hierarchy behaves, allowing one to enable or disable cache
memories and so on.
The boot-strap program itself is tasked with **booting** the computer. The idea is that when the processor is powered on, it resets into kernel mode and is fed the boot-strap program which initialises the hardware components, and invokes a dedicated **boot-loader** program which loads the kernel and starts executing it. For example, a typical boot-loader might first check the floppy disk drive (if one exists) for presence of a kernel, then move onto any hard disk drive or CDROM and so on. The exact mechanism by which it does this is beyond our scope, but clearly the program can be fairly simple since it has a very specific task. In addition, the boot-strap and boot-loader programs can typically be kernel agnostic in the sense that they do not care too much about which kernel they are loading, just that the executable program starts at one of the preset locations. Since the processor starts in kernel mode, the boot-strap and kernel have full access to all resources. Once the kernel is running, it takes control of the management of those resources and executes other programs in user mode.

As an aside, the slightly odd terminology used is derived from the phrase “to pull one’s self up by one’s boot-straps”. The original source of this phrase is debatable, but the implication is the achievement of something seemingly impossible: in this case managing to get the computer into a working state from an uninitialised starting point.

### 12.4 Event Management

The terms **exception**, **interrupt** and **trap** are often used somewhat interchangeably, but in reality they are different and it can be important to see what the differences are. All basically describe an event occurring inside or outside the processor which changes the normal flow of execution in some way: usually normal processing is suspended when such an event occurs and control is passed to an event **handler** (an interrupt or exception handler) in the kernel which deals with it in some way. Roughly speaking, we can describe all three terms as follows:

**Exception** An exception is a condition that occurs within the processor itself. An example is an unaligned load or store, or division by zero. In these cases, something has happened that the processor cannot deal with correctly; typically this might terminate the active process, or mean that some flag is set within a status register so that the process can check whether the error occurred and continue normally.

**Interrupt** An interrupt is a signal from either an external hardware component that is attached to the processor (a hardware interrupt) or software component executing on it (a software interrupt, or trap), rather than the processor itself. Occurrence of an **interrupt request** from such a component indicates that normal execution should halt and some action should occur in order to manage the signalling hardware or software. Note that use of an interrupt mechanism removes the need for a **polling loop** whereby the processor would otherwise need to con-
continuously check the hardware or software to see if it requires management (and hence waste time as a result).

Whereas an interrupt occurs because we have explicitly set things up so it will be triggered, an exception happens because of an implicit side effect within execution. Put a different way, an interrupt may occur at any time: such an event is asynchronous with respect to processor execution. Exceptions on the other hand are usually synchronous in that they occur as a direct result of a particular instruction being executed. Two final concepts that relate to both exceptions and interrupts need to be described:

**Definition 48.** Events are termed **precise** if when they occur and control is passed to the appropriate handler the *PC* is saved somewhere, and the pipeline content is valid in the sense that all instructions before the event are fully executed and all after the event are not fully executed. If these conditions are not satisfied, the event is termed **imprecise**.

**Definition 49.** Events are either **maskable** or **non-maskable**, properties that relate to whether the processor can ignore them or not. Maskable interrupts, for example, can be ignored by the processor by setting an **interrupt mask**, typically within a status register. In contrast, non-maskable interrupts always require immediate attention by the processor and have no corresponding field in the status register.

One way to categorise exceptions and interrupts is by considering an exception to be a non-maskable interrupt that just happens to be generated internally by the processor rather than externally by something else. Another way, which is adopted by MIPS32 and documented in detail within [44, Chapter 5], is a two-stage approach which relates the two concepts more or less the other way around. The basic idea is that if the processor has a mechanism to cope with exceptions, then it can use this to service interrupt requests. That is, once it detects an interrupt request the processor raises a particular type of exception to deal with the request. In order to see the complete picture of how both are managed, we look in detail at the role of the processor and the kernel.

### 12.4.1 Handling Interrupts

MIPS32 can distinguish between interrupts generated by up to eight different components. Two of these are reserved for software interrupts (i.e., events relating to software components), six are reserved for hardware interrupts (i.e., events relating to hardware components). Each interrupt has a number and is associated with fields in both *Cause* and *Status* as detailed by Table 12.2. Hardware interrupt number five is a somewhat special case since it is tied to the **timer interrupt** which is generated periodically by the processor itself rather than another hardware component.

The *IM0*...*IM7* fields in the *Status* register represent the interrupt mask. The idea is that if, for example, *IM2* is set to zero, then the processor is masking (or
### Table 12.2
A table describing MIPS32 interrupt types and their mapping into *Cause* and *Status* registers.

<table>
<thead>
<tr>
<th>Interrupt Type</th>
<th>Interrupt Number</th>
<th>Field in <em>Cause</em></th>
<th>Field in <em>Status</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>0</td>
<td><em>IP0</em></td>
<td><em>IM0</em></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td><em>IP1</em></td>
<td><em>IM1</em></td>
</tr>
<tr>
<td>Hardware</td>
<td>0</td>
<td><em>IP2</em></td>
<td><em>IM2</em></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td><em>IP3</em></td>
<td><em>IM3</em></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td><em>IP4</em></td>
<td><em>IM4</em></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td><em>IP5</em></td>
<td><em>IM5</em></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td><em>IP6</em></td>
<td><em>IM6</em></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td><em>IP7</em></td>
<td><em>IM7</em></td>
</tr>
</tbody>
</table>

ignoring) hardware interrupt number zero. As such, the processor can be selective in which interrupts are dealt with in a particular context; while dealing with a critical task it might be prudent to ignore interrupts generated by the keyboard device for example since these are less important. In addition, the *IE* field in the *Status* register acts as a master interrupt enable switch: when set to one the interrupt system is enabled, when set to zero it is disabled. As such, one might include instructions at the beginning of a program such as

```
li $t0, 0xFF01 ; move 0xFF01 into GPR[8]
mtc0 $t0, $12 ; move GPR[8] into CP0[12]
```

which load the value $FF01_{(16)}$ into $CP0[12]$, i.e., the *Status* register, to enable interrupts via the master flag and mask none of them (i.e., allow all to occur).

The behaviour of the processor when it spots an **interrupt request** from some component, when we say an interrupt is **raised**, is conceptually simple although can be tricky to implement correctly. The component in question issues an interrupt request by setting one of the *IP0*...*IP7* fields in *Cause*. For software interrupts, the process can directly update the *Cause* register; for hardware interrupts the hardware component is reliant on some mechanism that sets the field for it. For example, the field could be directly connected to a pin so that an external hardware device can signal directly into the processor. An interrupt is raised when all of the following conditions are met:

1. One of the interrupt request fields in *Cause*, say *IPn*, is set to one.
2. The corresponding interrupt mask field *IMn* in *Status* is set to one, i.e., the interrupt is not being ignored.
3. The *IE* field in *Status* is set to one, i.e., interrupts have been enabled.

Once the conditions are met, the processor raises an exception which deals with the interrupt. Before it does this, however, the processor updates the *Status* and *Cause* registers to reflect the change in state. For example, it might set the *NMI* field in the *Status* register so the handler could tell the event was maskable or non-maskable.
12.4.2 Handling Exceptions

The occurrence of an exception, whether generated to manage an interrupt or a “real” exception condition such as an arithmetic error, means transfer of control to the exception handler within the kernel. Since the processor needs to know where the handler is, it must be placed at a fixed address in memory which is termed the exception vector. Where there is a single vector for all exception types, we term the system single vectored: a handler placed at a single address must work out which exception occurred by decoding the various status registers. In contrast, a fully vectored system is where dedicated addresses are used to invoke different handlers for each exception type. Trade-offs clearly exist for each choice. For example, with a fully vectored system the handler needs to do less work in decoding the exception type, but each handler is somewhat constrained in size since one handler cannot overlap into the space reserved for entry into another handler. MIPS32 implements what one might call a partly vectored approach. There is a single vector 80000180 through which “general” exceptions are passed; a handler placed there determines the exception type by looking at the Exc Code field in the Cause register. Additionally, there are other vectors through which more “special” exceptions are passed. For example, there are dedicated vectors for dealing with non-maskable interrupts, soft reset, cache errors and so on. We concentrate on the use of a “general” exception handler from here on.

To transfer control to the exception handler, the processor must first be set into a stable and known state. Depending on the pipeline state and the need for precise exceptions the processor might first squash or flush partly executed instructions, or stall the exception mechanism until partly executed instructions are completed. Some types of kernel are re-entrant in the sense that, for example, they themselves can be interrupted while handling an interrupt. This is a tricky condition to deal with, so for more simple kernels which are not re-entrant, the processor might additionally disable interrupts so the kernel cannot be interrupted. The crucial next step is to switch into kernel mode: the exception needs to be managed by the kernel so to do this, the handler needs to execute in kernel mode with full access to the resources available. The exception handler can now take control; this is achieved by first setting the Exc Code field in the Cause register, and the Exception Program Counter or EPC register that holds the PC value related to the exception that occurred. For precise exceptions, EPC is the instruction which caused the exception (for example a load instruction which uses an unaligned address). To continue execution, EPC should be incremented before being used as the new PC. For imprecise exceptions, EPC is the PC value where execution should resume; no increment is required. Finally the processor sets PC to the appropriate address, in this case $PC = 80000180_{(16)}$, i.e., the “general” exception vector. Restarting execution at the new PC effectively invokes the exception handler which can perform whatever tasks are required to manage the event that occurred. Once the exception handler has finished, control needs to be passed back again; typically this means restarting the process that was executing when the event occurred. The processor manages the switch from user mode into kernel mode automatically; however, a switch in
the other direction is as the result of executing the `eret` instruction which takes the content of $EPC$ and copies it into $PC$, switches into user mode and then allows execution to continue.

Note that the cost of an exception (and hence an interrupt since they are managed in essentially the same way) can be described as the sum of

- the cost of entering the kernel, i.e., transferring control from user to kernel mode,
- the cost of any activity required of the kernel, e.g., dealing with an exception,
- the cost of exiting the kernel, i.e., transferring control from kernel to user mode.

Clearly it is advantageous to keep this cost as low as possible: the faster the steps are completed, the less time is wasted as overhead and the more time can be spent performing useful computation. Also note that this description resolves a problem from earlier where we wondered if a process executing in user mode was prevented from entering kernel mode freely for reasons of protection, how could we ever enter kernel mode? The answer is now clear: the exception mechanism implements the switch from one to another. Either the kernel configures the processor so it is always invoked at some point (e.g., it configures the timer interrupt to be triggered every say $n$ seconds), or it is invoked when and only when some event occurs that requires it to intervene.

### 12.4.3 Handling Traps

At the start of the chapter we described one of the requirements on the kernel as provision of a core set of functions which can be called by and executed on behalf of a process running in user mode. The example we gave was that of a **device driver** which is the generic name we give to operations that relate to management of a specific peripheral device. However, the concept is more general than this: imagine there is a set of **kernel functions** within the kernel which need to be called in a similar way to **user functions** housed within the process itself. The problem is, the kernel functions cannot be linked to user functions otherwise we would effectively include the entire kernel in *every* object file produced by the linker! The typical approach to solving this general problem is the use of software interrupts as a means of implementing **system calls**, i.e., calls from a process executing in user mode into functionality within the kernel.

Since we have already explained how interrupts and exceptions work in MIPS32, the concept of a software interrupt or trap is easy to explain: essentially a process executes a dedicated instruction which raises an interrupt request that is eventually serviced as an exception by the exception handler. MIPS32 has three main classes of trap instruction:

- Execution of the `syscall` instruction is intended to represent a system call by the executing process; it results in the system call exception `Sys` (exception number 8).
• The break instruction is intended to cause a break-point exception which is required for single stepping through a program by a debugger; it results in the break-point exception $Bp$ (exception number 9).
• There are a range of trap instructions such as $teq$ and $tne$ which cause the generic trap exception $Tr$ (exception number 13).

We concentrate here on use of syscall to perform system calls. In concept, this is fairly simple. Recall that in Chapter 10 we used the jal instruction to implement function calls: a user mode caller executed jal to save the return address and then transfer control to a user mode callee function. One can consider execution of syscall as somewhat similar in the sense that a user mode caller executes it to transfer control to a kernel mode callee function that exists within the kernel. The open question, however, is how one passes arguments and return values between user process and kernel? Just like the calling convention between a caller and callee user function might be standardised but is essentially a decision made by a particular programmer (or compiler), the system calling convention is a decision made by a particular kernel. That is, each kernel might have a different scheme to achieve this. A typical approach is to adopt a similar scheme to that described in Chapter 10 whereby arguments and return values are passed in predetermined registers: the user process loads the registers with values which are passed as arguments to the system call; the system call loads the registers with values which are passed back again as return values.

In fact, this approach is adopted by a limited kernel implemented by SPIM that allows simulated programs to interact with the simulation environment. For example, it allows a simulated program to print output to the terminal in which SPIM is executed. Appendix A outlines the system calling convention used by SPIM; briefly one loads the system call number into $v0$ and any arguments into $a0, a1$ and $a2$ then uses the syscall instruction to provoke a system call trap. As an example, consider using the print_int system call (system call number 1) to print the integer 999 to the terminal. This can be achieved as follows:

```
l0: ...
  addi $a0, $0, 999  # set system call argument
  addi $v0, $0, 1    # set system call number
  syscall             # invoke system call
l1: ...
```

Instructions before the system call (denoted by the label l0) are executed, then we load the argument and system call number into the appropriate registers. Execution of syscall provokes the trap which is handled by the kernel; the integer 999 is printed to the terminal. When the kernel has finished management of the trap, control is returned and instructions after the system call (denoted by the label l1) are executed.
### Table 12.3 A table describing MIPS32 exception types.

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Exception Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Int</td>
<td>Interrupt.</td>
</tr>
<tr>
<td>1</td>
<td>Mod</td>
<td>TLB modification.</td>
</tr>
<tr>
<td>2</td>
<td>TLBL</td>
<td>TLB exception (load).</td>
</tr>
<tr>
<td>3</td>
<td>TLBS</td>
<td>TLB exception (store).</td>
</tr>
<tr>
<td>4</td>
<td>AdEL</td>
<td>Address exception (load or fetch).</td>
</tr>
<tr>
<td>5</td>
<td>AdES</td>
<td>Address exception (store).</td>
</tr>
<tr>
<td>6</td>
<td>IBE</td>
<td>Bus error (fetch).</td>
</tr>
<tr>
<td>7</td>
<td>DBE</td>
<td>Bus error (load or store).</td>
</tr>
<tr>
<td>8</td>
<td>Sys</td>
<td>System call.</td>
</tr>
<tr>
<td>9</td>
<td>Bp</td>
<td>Breakpoint exception.</td>
</tr>
<tr>
<td>10</td>
<td>RI</td>
<td>Reserved instruction.</td>
</tr>
<tr>
<td>11</td>
<td>CPu</td>
<td>Co-processor unusable.</td>
</tr>
<tr>
<td>12</td>
<td>Ov</td>
<td>Arithmetic overflow exception.</td>
</tr>
<tr>
<td>13</td>
<td>Tr</td>
<td>Trap exception.</td>
</tr>
<tr>
<td>15</td>
<td>FPE</td>
<td>Floating point exception.</td>
</tr>
</tbody>
</table>

### Listing 12.1 An example interrupt handler (user portion).

```
.globl __start
.data
A: .word 0
    .word 1
    .word 2
    .word 3
.text
__start:  li  $t0, 0xFF01 # enable interrupts
         mtc0  $t0, $12
         la  $t0, A
         addi $t1, $t0, 0 # try to load from address A+0
         lw  $t2, 0($t1)
         addi $t1, $t0, 4 # try to load from address A+4
         lw  $t2, 0($t1)
         addi $t1, $t0, 8 # try to load from address A+8
         lw  $t2, 0($t1)
         addi $t1, $t0, 12 # try to load from address A+12
         lw  $t2, 0($t1)
         addi $t1, $t0, 13 # try to load from address A+13
         lw  $t2, 0($t1)
         addi $v0, $0, 10 # exit
         syscall # exit
```
Listing 12.2 An example interrupt handler (kernel portion).
12.4.4 An Example Exception Handler

As an example, we will consider writing a short handler to catch unaligned load and store exceptions: recall from Chapter 8 that this means the effective address used in a load or store is not a multiple of the size of data one is accessing. For example, if we access 32-bit words via the `lw` and `sw` instructions, the effective address must be a multiple of four (since a 32-bit word is four 8-bit bytes). Using the SPIM mechanism for system calls described above, we will print an error message when such an event occurs.

Although this is a specific example, the key thing to take away is the general framework which can be adapted or extended to cope with other exception types. The implementation is split into three parts. Listing 12.1 details the user portion; this represents a process executing in user mode that is used to test the handler by intentionally performing a series of aligned loads then an unaligned load to an array labelled A. Note that since SPIM will need to be run in bare mode without the default handler, the entry point is labelled `__start` rather than `main`.

MIPS32 permits thirty-two exception types, some of which are currently unused or reserved for processor-specific behaviour. We list the first fifteen, most central types in Table 12.3; in common with co-processor register we refer to the exception name rather than number for clarity. When an exception occurs, the exception number is stored in the `Exc Code` field in the `Cause` register. The register design has the least-significant two bits set to zero. This means one can extract `Exc Code` and the least-significant eight bits of `Cause` and use them as an aligned offset into a branch table which dispatches control to a specific handler for each exception type. We use this fact to implement a fairly generic dispatcher in Listing 12.2. It is worth noting that registers `GPR[26]` and `GPR[27]`, or `$k0` and `$k1`, are reserved for use by the operating system kernel: it can overwrite these registers without regard for their content whereas it must preserve others. This is an important point, and one related to the concept of preserving registers in function calls; if the kernel corrupts the

Listing 12.3 An example interrupt handler (kernel portion continued).

```
Listing 12.3 An example interrupt handler (kernel portion continued).

1  id0:       ...
2  id4:       ...
3  id5:  la $a0, message  # print message
4       addi $v0, $0, 4
5       syscall
6       mfc0 $a0, $8
7       addi $v0, $0, 1
8       syscall
9       la $a0, nl
10      addi $v0, $0, 4
11      syscall
12      j    exit_exp
13  id6:       ...
```
process state, then the process will execute incorrectly just like if a callee function corrupts the state of the caller, the caller will potentially execute incorrectly when control returns to it. As such, the handler starts by saving the values of $at, $v0 and $a0 which may be overwritten. It then extracts the exception type and uses the value as an offset into a branch table (which accommodates exception numbers 0 to 15 only). The branch table branches to the dedicated handler for a given type; in our case we are interested in exceptions AdEL and AdES (exception numbers 4 and 5).

Listing 12.3 lists the dedicated interrupt handler branched to by the branch table. For exception types four and five the handler prints a message followed by the address that caused the exception followed by a new line. After the dedicated handler completes, it branches back to either exit_exp or exit_int depending on whether EPC needs to be incremented or not as a result of a precise or imprecise interrupt. Either way, the next steps clear the Cause register, re-enable interrupts which had been disabled. Saved registers are then restored (note the use of the .set noat and .set at directives that allow this) before the handler uses an eret instruction to return to the address held in EPC, i.e., resume normal execution in user mode.

12.5 Memory Management

So far we have considered that processes have access to the entire address space, i.e., the range of addresses that are valid for use and supported by memory. As we move toward a situation where the kernel shares the processor and associated resources between many resident processes, this starts to become an unattractive assumption. We can highlight why by investigating some examples:

1. Two processes cannot both occupy the same region of memory. For example, the instructions for only one process can be located at address zero within a given address space. On the other hand, the memory model outlined in Chapter 10 stipulates that some features should be located at fixed addresses. For example, the stack should start at address $FFFFFFF (16) and grow downward. Clearly there needs to be a solution to this contradiction.

2. In the chapter introduction, we discussed the issue of memory protection using a high-level example. To briefly recap, the problem is that if several processes have access to the same address space, there is the potential for one process $P$ to access the memory being used by another process $Q$. In some cases, this could be an advantage, for example $P$ and $Q$ can communicate with each other using the shared memory, but can also cause problems. Such problems typically take the form of accidental corruption, whereby $P$ alters memory belonging to $Q$ and causes it to execute incorrectly, but can also represent a security hole whereby $P$ can read data owned by $Q$ that it should not be able to.

3. Looking at the memory hierarchy in Chapter 8, the top levels which represent registers and memory implement an attractive trade-off: registers are fast whereas memory is large, so when we run out of registers we store (or spill) values into
Operating Systems

memory for a while and then load them again when we need them. At the levels below this, which represent memory and disks, we do not have the same trade-off. That is, it is obvious that when we run out of memory we could store values on disk and then load them again when we need them, we just have not discussed a mechanism to do it. Given the kernel is in control of the disks, it is reasonable to assume it should implement such a mechanism.

There are several different ways to address these problems, but we focus on a specific technique called virtual memory.

12.5.1 Basic Concept

Although a virtual memory system can offer some benefits where there is a single shared address space, we can solve all the problems above in one go by allocating each user process (and the kernel) separate address spaces. Each address space is identified with an Address Space Number (ASN) which could be, but does not have to be, the same as the process number. The basic idea is that instead of thinking about memory as a single large array called MEM, we split it into two concepts: an \( m \)-element physical memory which represents the hardware which a process can load data from and store data into, and an \( n \)-element virtual memory which is in some sense a process’s abstract view of the address space it is allocated. To bridge the gap between the abstract virtual memory and concrete physical memory, there needs to be some mechanism managed by the kernel (and preferably supported by hardware) which translates virtual addresses into physical addresses. For example, when a process loads from virtual address \( x \), the access is translated into a load from some physical address \( y \).

Of course, the devil is in the detail; with virtual memory, there is quite a lot of detail and therefore plenty of devil as well. Even so, there are two key advantages from this arrangement. The first advantage is that translation mechanism permits \( x \) and \( y \) to be different on a per-process basis. To see why this is useful consider an example where we translate virtual address \( x \) to physical address \( p \) for process \( P \) and virtual address \( x \) to physical address \( q \) for process \( Q \). The result is that both processes can access their own virtual address \( x \) (i.e., they are both fooled into thinking they have access to the full address space) but behind the scenes we are translating accesses to disjoint regions of physical memory so that both can coexist; this essentially solves the first problem from above. Even better, since each process has a separate virtual address space the second problem disappears as well: the translation process prevents one process from accessing physical memory which is used to house part of the virtual address space for another process. Essentially we get a form of memory protection for free. The second advantage is that the size of the virtual memory address space can be larger than the physical address space if we can cope with this in the translation mechanism, i.e., we can have \( n > m \). This is possible because at whatever level you look at, a process is accessing a working set of instructions and data, i.e., those which are currently being used. We only need to
keep the working set in physical memory and can switch regions in and out behind the scenes to ensure this; this essentially solves the final problem from above.

As one might expect, the sticking point is finding an efficient way to realise the translation mechanism: given the so-called von Neumann bottleneck already noted in Chapter 8, the mechanism should not impose a performance overhead and ideally not an area overhead either. There are a few different approaches but we will examine a standard approach termed paged virtual memory. Conceptually, there is a convenient similarity between how cache memory and paged virtual memory operate. At a high-level, therefore, paged virtual memory can be described by the following comparison:

**Figure 12.2** A diagram showing two options for placing the MMU and the implication for how cache memory operates.
A cached memory system:
- divides memory into chunks called cache lines,
- operates transparently; configuration (e.g., dirty and valid bits) for each cache line managed internally and automatically by cache,
- holds a sub-set of larger address space; cache lines not held in the cache are stored in memory
- uses address translation to map addresses to sub-words within cache lines,
- uses management policies control cache line replacement, eviction and so on.

A paged virtual memory system:
- divides memory into chunks called pages,
- operates with support from the kernel; configuration (e.g., dirty and valid bits) for each page held in a page table,
- holds a sub-set of larger address space; pages not held in physical memory are stored on disk in a dedicated swap file,
- uses address translation to map virtual address to frame index and offset within physical memory,
- uses management policies control page replacement, eviction and so on.

The virtual memory system is often supported by the addition of a dedicated Memory Management Unit (MMU) which works alongside the processor so when accesses to memory are made, the accesses pass through and are partly serviced by the MMU. The MMU, and the decision to use virtual memory at all, is closely integrated with the processor ISA and micro-architectural design. For example, exactly where the MMU is placed highlights implications of the difference between virtual and physical addresses. Previously, our processor and associated components dealt exclusively with physical addresses. Now that a process will access memory using virtual addresses that are translated into physical addresses by the MMU, changes need to be made: any component which previously stored or operated on physical addresses now potentially needs to cope with virtual addresses instead. For example, the PC associated with each process will now be a virtual address and instruction fetches will thus be to virtual addresses. Likewise, any cache memories or branch prediction hardware will potentially need to change: in the case of caches, this depends on where the MMU is placed. Figure 12.2 shows two options, one where the MMU is placed before the cache and one after. In the first case, the cache operates with virtual addresses. On one hand this seems reasonable, but a potential problem is that if the cache tags are based on virtual addresses and there are many virtual address spaces, there is a danger that the cache will incorrectly identify two identical addresses from different address spaces as the same: this effectively means the wrong cache line will be resident. To cope with this the cache tags could be extended to include an ASN, or the cache flushed each time a new process (and hence address space) becomes active. The second case places the MMU before the cache so that it operates on physical addresses so that things work roughly in the same way as before.
12.5 Memory Management

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
    printf("page size is %ld bytes\n", sysconf(_SC_PAGESIZE));
}
```

Listing 12.4 A short C program to test the page size used by the host operating system.

### 12.5.2 Pages and Frames

The basic idea is to divide physical and virtual address spaces into fixed size chunks. The chunks of physical address space are called *frames*, there are $VM_{frames}$ of them and each one is $VM_{framesize}$ in size so that the physical address space is $m = VM_{frames} \cdot VM_{framesize}$ bytes. The chunks of virtual address space are called *pages*, there are $VM_{pages}$ of them and each one is $VM_{pagesize}$ in size so that each virtual address space is $n = VM_{pages} \cdot VM_{pagesize}$ bytes. The size of frames and pages are fixed and the same, i.e., $VM_{framesize} = VM_{pagesize}$, but can be dictated by the kernel. One can query the page size, for example, using a program such as the one outlined in Listing 12.4. A representative size would be 4096 bytes, and one would expect the size always to be a power-of-two in order that other aspects of the system are made simpler.

Each user process (and the kernel) has a separate virtual address space identified by an ASN. At a given point in time, frames in physical memory will be populated by pages from these address spaces as detailed by the example in Figure 12.3. Imagine that $VM_{framesize} = VM_{pagesize} = 4096$ bytes and that physical memory can, somewhat oddly, hold only $VM_{frames} = 3$ frames. The physical address space is thus $m = VM_{frames} \cdot VM_{framesize} = 3 \cdot 4096 = 12288$ bytes so valid physical addresses are in the range $0...12287$. Two processes $P$ and $Q$ are resident, each of which has a virtual address space that consists of $VM_{pages} = 4$ pages. Each virtual address space is thus $n = VM_{pages} \cdot VM_{pagesize} = 4 \cdot 4096 = 16384$ bytes so valid virtual addresses are in the range $0...16383$. For clarity the $i$-th page in the virtual address space for $P$ or $Q$ is labelled $P-i$ or $Q-i$, for example in this case process $P$ has four pages labelled $P-0$, $P-1$, $P-2$ and $P-3$.

Frames in the physical memory $MEM$ are currently populated with the three pages $P-2$, $P-3$ and $Q-0$. The remaining five pages (i.e., pages $P-0$, $P-1$, $Q-1$, $Q-2$ and $Q-3$) are held on disk in the swap file $SWAP$. Based on this state, it is already important to re-enforce two facts. Firstly, frame number $i$ in physical memory does not necessarily hold page number $i$ from a particular virtual address space; the translation mechanism maps pages to arbitrary frames. Secondly, frame numbers $i$ and $j$ do not necessarily hold pages from the same address space; physical memory can be populated with pages with different owners but the translation mechanism gives the illusion that each has dedicated access.
To keep track of the virtual memory state, the kernel maintains an internal data structure for each ASN which describes the associated virtual address space. The organisation of this data structure is crucial, but in order to focus on the fundamental idea we take a (very) simple approach:

**Definition 50.** For each virtual address space there is a page table where we write $PT_i$ to describe the page table related to the ASN $i$. Each $PT_i$ has $VM_{pages}$ entries in

---

**Figure 12.3** A diagram showing pages from two processes variously resident in physical memory and swap, and the page tables which describe this example state.
it, one for each page in the associated virtual address space. The $j$-th entry in such a page table, which we write $PT_i[j]$, has the following fields:

1. $PT_i[j]_{location}$ which determines whether page $j$ resides in physical memory or the swap file.
2. If page $j$ is resident in the swap file:
   - $PT_i[j]_{swap}$ holds an offset into the swap file where the page content is stored.
3. If page $j$ is resident in physical memory:
   - $PT_i[j]_{frame}$ holds the frame number in physical memory in which the page is resident.
   - $PT_i[j]_{dirty}$ determines if the page has been altered or not (i.e., if the versions in memory and on disk are different).

Although this simple scheme will suffice for our description, several issues need to be at least mentioned if not explored further. Firstly, note that it can be attractive to split this description into separate parts; often one might find a dedicated frame table to manage frames in physical memory or a dedicated disk map to manage pages in the swap file. Secondly, note that the data structure used to realise the information held in a page table is crucial: many techniques for efficient page table data structures exist (e.g., so-called inverted or multi-level page tables) although we ignore them here. Thirdly, note that the kernel might reserve some of the frames in physical memory as special, and lock them so they cannot be used to hold pages from address spaces associated with user mode processes. The reason for this is simple: if the kernel itself does not exist in physical memory, it will be hard for it to manage the various data structures! To avoid this somewhat circular problem, we simplify matters and assume the address space associated with kernel memory is always resident in physical memory.

Based on this description, the bottom half of Figure 12.3 enhances the top half by describing two page tables $PT_0$ and $PT_1$, which relate to ASNs 0 and 1 assigned to processes $P$ and $Q$ and allow us to interpret the state of the virtual memory system. A demand paging system [25] would assume that when a process is initially allocated a page table no pages are resident in physical memory: all pages are transferred into physical memory or the swap file on demand, i.e., when their content is accessed. In this case, however, we are past this initial stage since frames in physical memory are populated with pages from $P$ and $Q$. In particular, looking at the $PT_0$ we can see that page 1 in this address space (i.e., page $P - 1$) is resident in the swap file at offset 1. On the other hand, looking at $PT_1$ we can see that page 0 in this address space (i.e., page $Q - 0$) is resident within frame 2 of physical memory and has been altered (i.e., any version of the page within the swap file is out of date).
12.5.3 Address Translation and Memory Access

Given suitable page tables that are configured by the kernel to describe the state of the virtual memory system, the next step is to utilise them so memory accesses can be satisfied: this is where the MMU starts to play a role. The MMU needs access to the page table for the process currently active (i.e., the process currently executing on the processor) which would have been configured by the kernel. Conceptually, and in MIPS32 at least, one can imagine the kernel setting the co-processor registers to reference the current ASN and include a pointer to the associated page table data structure. An important fact to re-enforce is that the MMU operates as autonomously as possible. We do not want every memory access by a process to require attention by the kernel since this would impose a massive performance overhead. Rather, the MMU is able to automatically satisfy memory accesses for virtual addresses whose corresponding page is in physical memory; the kernel is only invoked when there needs to be some management or re-configuration of the virtual memory system.

Put simply, every use of a virtual address VA must be translated into a physical address PA which can be serviced by the physical memory MEM. Examples where accesses using VA are performed are easy to imagine: our PC must now hold the virtual address of the next instruction to fetch from memory, and execution of a load or store instruction will require a virtual effective address. The translation process is similar to that we described while looking at cache memories: we use VA to calculate two components which are used to access the active page table:

\[
VA_{offset} = VA \mod VM_{pagesize} \\
VA_{index} = \lfloor VA / VM_{pagesize} \rfloor
\]

The value \( VA_{index} \) gives the page number for this virtual address, it identifies the page sized chunk of our virtual address space the virtual address lies within; the value \( VA_{offset} \) gives the offset within that page. Since the kernel would select \( VM_{pagesize} \) to be a power-of-two, computing the values \( VA_{index} \) and \( VA_{offset} \) is not costly: given a 32-bit address \( VA \), if we select \( VM_{pagesize} = 4096 \), then it amounts to taking the twelve least-significant bits as \( VA_{offset} \) and the remaining twenty bits as \( VA_{index} \). Returning to the example in Figure 12.3, one could imagine process number 0 (i.e., process \( P \)) performing a load from virtual address

\[
VA = 00003008_{(16)} = 12296_{(10)}.
\]

This is reasonable: the virtual address \( 12296_{(10)} \) is within the range of valid virtual addresses even though it is outside the range of valid physical addresses. Extracting the correct sections of \( VA \) as follows

\[
\begin{pmatrix}
00003 \\
v_{index} \\
v_{offset}
\end{pmatrix}_{(16)}
\]
we find that

\[ VA_{offset} = 008_{(16)} = 8_{(10)} \]
\[ VA_{index} = 00003_{(16)} = 3_{(10)} \]

or more simply that this virtual address is in page number 3 (i.e., page \( P - 3 \)) at offset 8 of this virtual address space.

The MMU can check if the page is resident in physical memory by inspecting

\[ PT_i[VA_{index}]_{location}. \]

That is, it looks in the page table for the active process and retrieves the field which tells us whether page \( VA_{index} \) is held in physical memory or the swap file. As is the case detailed in Figure 12.3, imagine the page is held in physical memory. In this case, the easier of the two, the MMU can translate the VA into a physical address \( PA \) as follows:

\[ PA = PT_i[VA_{index}]_{frame} \cdot VM_{framesize} + VA_{offset}. \]

Step-by-step, the idea is that the MMU looks in the page table for the active process and retrieves the frame number for page \( VA_{index} \) which is scaled to give a base address, and added to \( VA_{offset} \) to give the physical address. That is, it calculates

\[ PA = PT_0[VA_{index}]_{frame} \cdot VM_{framesize} + VA_{offset} \]
\[ = PT_0[3]_{frame} \cdot VM_{framesize} + 8 \]
\[ = 1 \cdot 4096 + 8 \]
\[ = 4104_{(10)} \]

The resulting address \( PA \) is within the range of valid physical addresses and can thus be used as the effective address passed to the physical memory in order to perform the load we started off with. Note that in this case the MMU has dealt with the load without any intervention of the kernel: there is essentially no performance overhead (bar the cost of actual translation). Where the memory access is a store rather than a load, the access mechanism works much the same way: if the page is in physical memory, we can simply translate the address and perform the store. However, one minor addition is required in order to update the value of \( PT_i[VA_{index}]_{dirty} \). This marks the page in the page table as dirty which means that the version held in physical memory is noted as differing from the version held in the swap file: essentially we know that the page content has been updated as a result of the store instruction being executed.

The more difficult case is where page \( VA_{index} \) is not held in physical memory: the value we want to load is currently resident in the swap file so clearly we cannot perform the load. Imagine that process number 0 (i.e., process \( P \)) performs another load, this time from virtual address

\[ VA = 000000008_{(16)} = 8_{(10)}. \]
This address gives $VA_{\text{index}} = 0$ and $VA_{\text{offset}} = 8$, i.e., we are trying to load from page number 0 (i.e., page $P-0$) that is currently held in the swap file. This sort of situation is termed a page fault and results in an exception being raised that invokes the kernel in order to resolve matters so the load can eventually be performed. You can think of this as being similar in concept to the exceptions caused by unaligned loads (or stores) we examined previously: in both cases a load (or store) instruction causes some problem with respect to the memory system that needs to be resolved by the kernel.

**12.5.4 Page Eviction and Replacement**

When a cache memory discovers that it does not hold the cache line associated to some address, it might first evict an old cache line to free some space and then load the required cache line from memory. One might say the old cache line is replaced by the new one; once this occurs, the cache can perform the access required of it. The kernel performs a similar task with pages. As above, imagine that process number 0 (i.e., process $P$) performs a load from virtual address $VA = 00000008_{(16)} = 8_{(10)}$

which gives $VA_{\text{index}} = 0$ and $VA_{\text{offset}} = 8$. To perform the load, page 0 (i.e., page $P-0$) must be held in physical memory so the kernel must first evict another page to free space and then load page 0 from the swap file. Once the page is loaded into physical memory and the page table updated to reflect the new state, the kernel can return control to the process which caused the page fault thereby restarting the original memory access: the second time around this will succeed because page 0 is now in physical memory. If one believes that access to the swap file on disk is possible, this should seem a fairly simple task for the kernel. However, there are at least two issues that need to be mentioned which can somewhat complicate matters.

Firstly, how does the kernel select a page for eviction? That is, if there are no empty pages (in which case the kernel would just select one of them to load the new page into) which page of those resident should be moved from physical memory into the swap file? Like a cache, the kernel needs to employ a replacement policy so those pages used most are likely to be retained in physical memory. Although the policy is analogous to that required in the context of caches, since the cost of accessing memory and disk is higher than the cache and memory the penalty for an ineffective policy is higher as well. The particular case where a page is repeatedly swapped into physical memory then swapped out again is often called thrashing. The prolonged spells of disk activity that result mean you can often hear this happening as well as noticing the performance degradation: the system seems to “freeze” as the kernel spends most of the time swapping pages in and out of physical memory and no time allowing processes to execute. When we examined caches, use of Least Recently Used (LRU) policy was described as an expensive option despite
providing an effective way to identify cache lines to evict. As a result of the higher penalties involved here, the overhead of operating LRU to identify pages to evict is typically worth it.

Secondly, what actually happens when a page is evicted? Like a cache, if the page resident in physical memory that we want to evict is dirty (i.e., the page content has been updated), we first need to copy back the content to the swap file. More simply, if we want to evict a dirty page, we first save the content into the swap file whereas if the page is not dirty we can simply discard it. In the context of our running example from above process number 0 (i.e., process \( P \)) needs page 0 (i.e., page \( P - 0 \)) in physical memory. If the kernel chooses to evict the page in frame 0 (i.e., page \( P - 2 \)), then it can simply overwrite the frame with new page content since the old page content is non-dirty. However, if the kernel chooses to evict the page in frame 2 (i.e., page \( Q - 0 \)), then it first needs to save the old page content into the swap file, since it is marked as dirty, and only then overwrite the frame with new page content. This difference in behaviour might influence the page replacement policy in the sense that if we select dirty pages for eviction, the cost is higher than if we select non-dirty pages. Specifically, the former case implies two accesses to the swap file (one page store and one load) whereas the latter implies one access (one page load).

### 12.5.5 Translation Look-aside Buffer (TLB)

So far we have painted a very positive picture of virtual memory: as long as the kernel can manage the page table configuration, the initial problems we outlined are more or less solved. That is, by the kernel and MMU implementing the translation mechanism behind the scenes and swapping pages in and out of physical memory, each resident process has access to a large protected address space. However, there is a nasty issue lurking behind this picture. At the moment, each access to memory performed by a process actually requires two accesses to memory, one to inspect the page table and one to perform the actual access. Clearly this is far from ideal: even though the MMU acts autonomously so as to minimise the kernel activity to those occasions where some management or re-configuration of the virtual memory system is required, the price of solving our problems has been a 2-fold increase in the cost of every memory access. Since this includes instruction fetches, we have essentially halved the performance of the processor!

Understandably, solutions to this issue are fundamental if virtual memory is to be feasible for use. Fortunately we have already examined techniques that can provide such a solution: the basic idea is to hold page table entries in what amounts to a fancy cache memory called a **Translation Look-aside Buffer (TLB)**. Although there are ways to optimise the performance of the resulting system, in a basic sense the TLB is just a fully-associative cache which maps page numbers to frame numbers. Essentially we try to use the TLB to eliminate memory accesses that inspect the page table in our translation mechanism, i.e., we want to have a situation where
\[ PA = PT_i[V_{index}]_{frame} \cdot VM_{framesize} + VA_{offset} \]

That is, we calculate \( VA_{index} \) and send this to the TLB which searches the entries and hopefully returns us the base address of the frame which holds that page. Clearly the TLB can only hold a sub-set of entries from a given page table. For some virtual address \( VA \), if there is an entry for \( VA_{index} \) in the TLB a TLB-hit occurs and we can quickly form the physical address \( PA \) without a second memory access. If there is no entry for \( VA_{index} \) in the TLB, a TLB-miss occurs and we are forced to access the page table in memory to form \( PA \). The principles of temporal and spatial locality hopefully mean that more TLB-hits occur than TLB-misses and hence on average the second accesses required to load a page table entry from memory are required infrequently.

Since the TLB is a cache, in the case of a TLB-miss one would expect the page table entry retrieved from memory to replace one currently resident in the TLB. This can be managed automatically in hardware like a normal cache, or provoke an interrupt which allows the kernel to dictate the replacement policy.

### 12.6 Process Management

Informally we have already said that a process is a run-time entity in the sense that a process context captures one executing instance of a program: the program is a static entity, the process is an active or dynamic entity. In the most general sense, we would like to define a process as the basic unit managed by an OS that supports multitasking. The basic idea is that the kernel permits many processes to be resident in memory at the same time, keeps track of their contexts and swapping between them to allow sharing of the processor and hence concurrent execution: if virtual memory provides an abstract view of memory, one could think of multitasking as providing each process a virtual processor which is an abstraction of the real processor.

The concept of a process versus that of a thread both have associated subtleties that we need to be careful about. As such, we will adopt a (fairly) standard way to categorise the two:

**Definition 51.** Processes:
- have independent goals; each is executed to perform a different task,
- have separate processor context (e.g., different \( PC \) and \( GPR \) state),
- have separate address space,
- interact with each other using relatively expensive mechanisms (e.g., system calls).

**Definition 52.** Threads:
- have a common goal; each is executed as part of a thread team which cooperate to perform a single task,
• have a somewhat separate processor context (certainly different \( PC \), probably shared \( GPR \) state),
• have a shared address space,
• interact with each other using relatively inexpensive mechanisms (e.g., via controlled access to their shared address space).

Again informally, one might describe a thread as a sub-process or lightweight process. Some kernels manage threads naively alongside processes, others do not recognise them and force threads to be supported by a user mode library. Here we are interested mainly in the mechanism that allows the processor to be shared, so we take the simplifying approach of assuming the two things are essentially the same and exclusively discuss processes.

### 12.6.1 Storing and Switching Process Context

For each process the kernel allocates a **Process Identifier (PID)**, which is simply an integer name for the process, and maintains a **Process Control Block (PCB)** that acts as an area where the process context can be stored:

**Definition 53.** The PCB for the \( i \)-th process, which we denote \( PCB[i] \), has (at least) the following fields:

1. \( PCB[i]_{PC} \), the saved program counter for the process which represents the address where execution should resume.
2. \( PCB[i]_{GPR} \), the saved general-purpose register file content; *all* registers in \( GPR \) are saved here, we use \( PCB[i]_{GPR[j]} \) to refer to the \( j \)-th such register where appropriate.

Of course, each PCB might include many more details that relate to how the kernel controls each process. For example, one might include a **priority level** whose use we will see later, and pointers to any virtual memory configuration (e.g., the page table) for the process.

With one processor, clearly only one process can actually be executing at a time: the context of that **active process** is held and used by the processor, the context of other **non-active processes** are stored in their respective PCB entries. In order for the kernel to share the processor between the processes, there needs to be a mechanism to swap the active executing process for another; this is called a **context switch**.

Imagine there are two resident processes called \( P \) and \( Q \) which have PIDs 0 and 1 respectively, and that \( P \) is the active process: the context of \( P \) is held by the processor since \( P \) is currently executing, the context of \( Q \) is held in \( PCB[1] \). At some point, the kernel steps in and decides that \( Q \) should execute instead: to do this, the context associated with \( P \) needs to be saved and then replaced by the context associated with \( Q \). More simply, the kernel saves the context held by the processor into \( PCB[0] \) then
loads the saved context from PCB[1]. Since this replaces the value of PC, when the kernel returns control to user mode process Q will start to execute; PCB[1]PC has become the new PC so this is where execution recommences. Now imagine at some point the kernel again steps in and decides Q has executed for long enough and that P should have another chance: to do this, the kernel saves the context held by the processor into PCB[1] then loads the saved context from PCB[0]. When the kernel returns control to user mode process P will resume execution; PCB[0]PC, where execution of P was halted the first time we switched context, has become the new PC so this is where execution recommences.

A context switch is conceptually simple but deceptively costly. As in the case of handling an exception, said cost can be described as the sum of transfer into kernel mode from user mode and vice versa, plus activity performed by the kernel. Focusing on the kernel activity, to switch from the process with PID 0 to the process with PID 1 we simply save the PC into PCB[0]PC and each GPR[j] into PCB[0]GPR[j], then we load PC from PCB[1]PC and each GPR[j] from PCB[1]GPR[j]. Even with this simple scheme where the context of a process is described by just the program counter and a register file of thirty two registers, we need a total of sixty six memory accesses! This is a lower bound on cost: we did not for example include the cost of the kernel deciding on which process it wants to execute or updates to state such as the virtual memory configuration. Keeping in mind that context switching is pure overhead with respect to useful computation, this is a high price and one which particular architectures attempt to solve with the assistance of features in the processor.

### 12.6.2 Process Scheduling

Although we have established a mechanism to store and switch process context to and from the processor and PCB table, there are three questions to answer that fall roughly under the umbrella of **process scheduling**. The goal is to have a **scheduler** (a particular algorithm or mechanism within the kernel) perform context switches in order to manage access to the processor according to some policy. Firstly we need to understand what states a process can be in. For example, how do we know some process Q is a candidate to be executed and more generally, how does the kernel keep track of this information? Secondly, given the scheduling mechanism needs to be executed in some way so it can perform the context switch, what options exist for provoking this execution? Thirdly, given some process P will be active, **which** process (i.e., which Q) should the kernel select to replace it?

#### 12.6.2.1 Process States and Queues

The starting point is the definition of numerous **process states** which describe what phase of execution a process is currently in. The specific states and their level of
detail depend on the kernel; in the most basic sense one can imagine a 2-state model whereby a process could either be executing or not executing. The idea would be for the kernel to keep track of which state each process is in by maintaining a process queue (or list) for each possibility and moving processes between queues as their state changes. In the 2-state model this is very simple since the executing queue can only ever contain one process (if there is one processor) and the not executing queue contains all other processes.

More generally, we can formulate the 5-state description in Figure 12.4 where the meaning and purpose of each state, and the transitions between them, is as follows:

**Created** When a process is initially created, it is placed in the created state. This represents a “foetal” process in the sense that it has been created but not yet initialised, and is hence not yet ready for execution. Once the kernel has initialised the process (e.g., allocated structures to keep track of it, initialised context such as the instructions or stack), it is moved into the ready state and thereafter becomes a candidate for execution.

**Ready** From the point of view of scheduling in the sense of the question above, the ready state (and associated ready queue) is perhaps most important. Essentially, all processes in the ready state are candidates for selection by the kernel to replace the active process via a context switch; a process on the ready queue is ready for execution. The policy that selects a process from the ready queue and schedules it for execution is discussed later.

**Executing** A process in the executing state is active: the associated context is being used by the processor to execute instructions. During execution, one of two events can cause the active process to be descheduled (i.e., switched for some other process). Firstly, the kernel might step in and suspend the active process because it wants to (or needs to) allow another process to execute. This sets the process state to ready, and hence moves it into the ready queue. Secondly, an operation performed by the process might cause it to become blocked; for example, it could try to access a resource which is not currently free. This sets the process state to blocked, and hence moves it into the blocked queue. One slight subtlety is that at least one process must be available for execution otherwise
it is unclear what the processor will execute. To cope with this, many kernels maintain a somewhat privileged special process called the idle process: this is simply a process which causes the process to do nothing if there are no useful processes to execute.

**Blocked** A process which is blocked is essentially waiting for some event to happen before it can continue execution; in some sense it can execute but cannot do anything useful but wait. Common reasons for a process to become blocked include waiting for some I/O operation (e.g., a slow disk access, or user input) or some resource to become available. Once the event occurs (e.g., the I/O operation completes or the resource becomes free), the process can return to the ready state since it is then ready for execution again.

**Terminated** During execution, a process can be terminated. Sometimes this relates to natural completion (i.e., the process exits gracefully), but sometimes can be due to action by the kernel (e.g., the kernel terminates a process that causes some exception). Either way, the idea is that the process lifetime is over and the associated context should be removed from any structure the kernel may have used to keep track of it. A process which has been terminated but whose context has not yet been removed from the kernel is termed a zombie process: it has been terminated but sort of still lives on!

### 12.6.2.2 Creating and Terminating Processes

One question that arises from the previous discussion of process states is how processes are created and terminated, i.e., how processes enter the “created” state initially and move into the “terminated” state at some point during execution. Both operations must be provided by the kernel as system calls: a process must be able to ask the kernel to create a new process or terminate itself. To illustrate how such system calls can work (a given kernel might take different approaches), we use the example of a UNIX-style kernel.

- A process can terminate itself by using the `exit` system call; the prime reason to do so would be normal completion of execution. It could also be the case that an external event triggers termination of a process, for example if the process causes an unrecoverable exception the kernel might opt to terminate it.
- A process can create a new process by using the `fork` or `exec` system calls: the original process is termed the parent process, the new process is the child process. Since the parent and child are separate processes, they are allocated separate address spaces. By using `fork`, the parent memory content is copied into the child context: the two processes execute the same program image. Using `exec` changes this behaviour by subsequently patching the child context with a new program image. Intuitively this can be more useful since it allows one program to initiate the execution of a totally different program. Given this model, UNIX traditionally has a root process called `init` which is the ancestor of every other process in the system: every other process has been executed as the result of a series of `fork` or `exec` started by `init`. 
• The parent can use the `wait` system call to suspend itself until the child has finished executing (i.e., is terminated), otherwise the parent and child processes can execute independently. If the parent does not wait for the child and the child terminates, this is one example of where a zombie process occurs: the child still has a PCB entry even though it is not an actively executing process. In some cases the system enforces the convention that when the parent process is terminated, all children are terminated automatically. Where this is not the case and the parent terminates before the child, the child becomes an orphan process.

### 12.6.2.3 Invoking the Scheduler

In reality, the kernel clearly cannot just “step in” as we described above: it must be invoked by some mechanism:

**Definition 54.** The concept of multiprogramming is where the kernel waits until the active process is blocked (e.g., waiting for a read from disk to complete) and then switches it for another process. The problem with this approach is neither the kernel nor the processes have any control of when context switches occur; this makes use of multiprogramming difficult in interactive systems.

**Definition 55.** The concept of cooperative multitasking places control in the hands of the processes themselves: a context switch occurs when the active process offers to be switched for another process; the active process is said to yield control of the processor. Although this can be effective when the processes behave properly, the kernel need to deal with the issue of uncooperative processes: if the active process never yields, then other processes are starved of execution time.

**Definition 56.** The concept of preemptive multitasking places control in the hands of the kernel: a context switch occurs when the active process is deemed, by the kernel, due to be switched for another process.

Although there are advantages and disadvantages to all three approaches, an obvious advantage of preemptive multitasking is the ease by which it can be realised. More specifically, it is quite easy to construct a system which decides when a context switch is due. The easiest way to achieve this is via the concept of time sharing or time slicing. The idea is that the kernel controls a timer which acts like an alarm clock: once reset the timer waits for a period of time (a so-called time slice or time quantum) and then sounds an alarm, which in this case means it triggers an interrupt that in turn invokes the kernel. In MIPS32, this behaviour is controlled by the `Count` and `Compare` registers (i.e., registers `CP0[9]` and `CP0[11]`). Roughly speaking, the `Count` register is incremented by the processor after completion of every cycle and when `Count = Compare`, a timer interrupt is triggered. Thus, by setting `Count = 0` and `Compare` equal to the time quantum, e.g., `Compare = 1000`, one can force an interrupt to occur 1000 cycles into the future. In turn, this allows the kernel to force a situation where the process scheduling mechanism is invoked every 1000 cycles: a process can execute for 1000 cycles after which the kernel “steps in” and performs a context switch so another process can execute.
12.6.2.4 Scheduling Policies

Finally we are in a position to discuss how the scheduler, which we assume has been invoked by some mechanism, selects a process from the ready queue in order to perform a context switch. That is, how the scheduler applies a policy which would typically attempt to be

- fair, i.e., it does not indefinitely prevent a (valid) process from executing even though it may favour some over others,
- efficient, i.e., the scheduling mechanism does not take a prohibitively long time to execute,
- robust, i.e., it cannot perform some action which causes a process (or the kernel) to malfunction.

Often, the scheduling policy is split into three levels which are applied with differing frequencies and have differing goals:

**Long-term** The long-term scheduler deals with the high-level or “big picture” issues; it is invoked infrequently and tasked with deciding which processes should inhabit the ready queue. The idea is that the long-term scheduler takes on the role of load balancing: it might try to maintain a mix of I/O bound (i.e., those that perform mainly I/O) and computationally bound processes for example, in order to give the best overall system performance.

**Mid-term** The mid-term scheduler is typically tasked with swapping processes into and out of physical memory based on their current state (or predicted future behaviour).

**Short-term** The short-term scheduler deals with the low-level detail of selecting a process from the read queue and performing a context switch so that it can execute. Since the short-term scheduler is invoked frequently (e.g., after each time slice), efficiency of this operation is important.

Even within these types, one might expect different algorithms to be deployed in different scenarios in order to cope with different demands. In short, a “one size fits all” approach is seldom appropriate. Investigation of these algorithms at a deep level is a little beyond the scope of this book; to give a flavor, some very simple short-term approaches to scheduling are overviewed below:

**Round Robin** The concept of a “round robin” allocation is more general than this context; it means giving an equal share of some resource, in turn, to some recipients. Here, the resource is execution time and processes are the recipients: a round robin scheduler assigns a fixed size time slice to all processes and iterates through them in turn. Each time it is invoked, it selects the next process and schedules this for execution. This approach is simple and so can be inexpensive to implement and execute; it is fair in the sense that every process is guaranteed to be executed at some point.

**Priority-based** One of the disadvantages of round robin scheduling is that it ignores priorities that might be assigned to processes. For example, processes that
deal with interaction with the user could be said to be higher priority: if they are not scheduled very soon after a user clicks the mouse or presses a key for example, the user will notice the delay which will degrade their experience. By assigning each process a priority, e.g., adding it to the PCB entry maintained by the kernel, one can imagine a scheduling scheme that alters round robin by ordering the iteration by priority. That is, instead of iterating through the processes in turn, it orders them so that the next process is more likely to be one of those with a high priority than one with a low priority. Of course the problem with this approach is fairness: one now cannot make the claim that every process will be executed at some point since a low priority process could be starved of execution time by higher-priority processes.

**Resource-based** If one could estimate the amount of resources that a process needs to execute, e.g., the length of time it is expected to execute, one might make scheduling decisions based on this. For example, one might use the so-called Shortest Job First (SJF) or Shortest Remaining Time (SRT) strategies whereby a process which is expected to take the least time to execute is scheduled before others. Intuitively this should make some sense; it roughly operates like the “express” lane at a supermarket checkout. The idea is to maximise throughput in terms of the number of processes that complete in a given time period. The disadvantages are that again fairness is hard to guarantee, and also that obtaining accurate information about execution time (or any resource requirement for that matter) can often be difficult.

### 12.6.3 An Example Scheduler

In an attempt to demonstrate the main concepts of process management, we will write a “process switcher” that one can think of as a (very) limited preemptive multitasking scheduler. The switcher is invoked by the timer interrupt and switches between two hard-coded processes (i.e., there is no fork system call), one which prints the character ‘A’ and one which prints the character ‘B’. There is a nice historical motivation to use of this particular example. Moody [45, Page 36] recounts the words of Linus Torvalds, creator of the Linux kernel, in describing his efforts on the comp.os.minix newsgroup:

> I was testing the task-switching capabilities [of the Intel 80386 processor], so what I did was I just made two processes and made them write to the screen and had a timer that switched tasks. One process wrote ‘A’, the other wrote ‘B’, and I saw “AAAAABBBB” and so on.

Like the example exception handler, this is a specific example but again the key thing to take away is the general framework which can be adapted or extended to cope with a more complete or flexible approach: just like Linux, with a few decades work by hundreds of committed programmers you too could transform the example into a real OS!
### Listing 12.5 An example process scheduler (user portion).

```
.globl __start
.data
.text
__start:  li $t0, 0xFF01    # enable interrupts
           mtc0 $t0, $12

     li $t0, 4        # set timer limit
     mtc0 $t0, $11
     li $t0, 0        # set timer count
     mtc0 $t0, $9

idle:    nop       # perform idle loop
             nop
             nop
             nop
             j idle

proc_0:  li $a0, 0x41 # process 0 (prints A)
             li $v0, 11
loop_0:  syscall
             j loop_0

proc_1:  li $a0, 0x42 # process 1 (prints B)
             li $v0, 11
loop_1:  syscall
             j loop_1
```

Implementation of the process switcher is again split into three parts. Listing 12.5 details the user code which includes the two processes. Note again that since SPIM will need to be run in bare mode (and with the `-mapped_io` option) without the default handler, the entry point is labelled `__start` rather than `main`. The code starts by enabling all interrupts and setting the `Count` and `Compare` registers so as to trigger a timer interrupt after a few cycles. It then drops into an infinite loop labelled `idle` which consists of `nop` instructions and in some sense relates to the idle process described previously. With no action by the kernel, we would expect execution to never exit from this loop. However, our scheduled timer interrupt will cause the process switcher to execute two hard-coded processes marked `proc_0` and `proc_1` which loop and use the SPIM mechanism for system calls to print ‘A’ and ‘B’ characters to the terminal. Note that we have intentionally initialised the arguments to the system call outside the loop. This acts as a rough test that the scheduler is context switching correctly: if it does not correctly save and restore the `$a0` and `$v0` registers for each process, they will not print the right thing.

Listing 12.6 presents a similar generic dispatcher as the one used in the exception handler example, i.e., Listing 12.2. It performs the same sequence of saving registers used by the kernel, extracting and using the exception number to invoke a dedicated handler, and management of EPC upon completion. One slight difference is the addition of code to reinitialise `Count` and `Compare` to ensure timer interrupts keep being triggered, but otherwise one would expect Listing 12.2 and Listing 12.6
12.6 Process Management

Listing 12.6 An example process scheduler (kernel portion).

```
.kdata
.id: .word id0
.id: .word id1
.id: .word id2
.id: .word id3
.id: .word id4
.id: .word id5
.id: .word id6
.id: .word id7
.id: .word id8
.id: .word id9
.id: .word idA
.id: .word idB
.id: .word idC
.id: .word idD
.id: .word idE
.id: .word idF
.save_at: .word 0
.pid: .word 1
pcb: .align 2
.space 164

.ktext 0x80008180
.set noat
.add $k0, $0, $at # save GPR[1]
.sw $k0, save_at($0)
.set at

.mfc0 $k0, $13 # get exception cause
.li $k1, 0x3F
.and $k0, $k0, $k1
.lw $k0, id($k0) # load handler address
.jr $k0 # jump to handler address

.exit_exp: mfc0 $k0, $14
.addi $k0, $k0, 4
.mtc0 $k0, $14

.exit_int: mtc0 $0, $13 # clear cause
.li $k0, 4 # set timer limit
.mtc0 $k0, $11
.li $k0, 0 # set timer count
.mtc0 $k0, $9
.li $k0, 0xFF01 # enable interrupts
.mtc0 $k0, $12

.set noat
 lw $k0, save_at($0) # restore GPR[1]
.add $at, $0, $k0
.set at

.eret # return from handler
```
Listing 12.7 An example process scheduler (kernel portion continued).
to be merged into one and roughly represent a single dispatcher capable of dealing with any exception. Listing 12.7 shows the major difference between the two examples, which is the length of the dedicated handler for Int (exception number 0) triggered by the timer interrupt. In Listing 12.6 space for two kernel data structures is allocated: pid holds the index of the currently executing PCB (0 for proc_0 and 1 for proc_1), pcb holds PCB data relating to each process. We take a very limited approach and assume there will only ever be two processes, i.e., those marked proc_0 and proc_1, and allocate space accordingly. Each entry has enough space to save the entire general-purpose register file as well as the PC value for each process; there are no priority levels or other meta-data. We initialise pid to the value 2 which essentially marks the system as uninitialised.

The code in Listing 12.7 for Int (exception number 0) starts by retrieving and testing the value of pid. If it finds the system uninitialised, it branches to the label id0_init and initialises it. To do this, it stores the addresses of proc_0 and proc_1 in the PC entries within their respective PCB space and then schedules execution of proc_0 by setting the pid value and updating the EPC value to match. If the system is already initialised, one of the processes must already be running and control drops through to the label id0_swap to perform a context switch. The handler first loads the current PBC index and scales it to point at the corresponding space within the pcb structure. It then saves the context of the current process, storing each general-purpose register and the PC value where execution should resume (which is loaded from EPC). It then needs to decide which other process should be scheduled for execution: in this example, since there are only two processes the choice is easy! The handler toggles the current PCB index to the other process before again manipulating the index into a pointer at pcb. It then restores the process context, including the PC value (which is stored in EPC), before finally branching back to the generic handler which returns control to the newly scheduled process.

12.7 Further Reading

  *Operating System Concepts.*
  *Operating Systems Design and Implementation.*
12.8 Example Questions

48. Imagine processes that execute on a MIPS32-based processor are scheduled by an operating system kernel. This platform is used in an application that gathers data from a number of sensors. Efficient context switching is important; if a device driver does not begin execution soon after an interrupt, important data could be lost from the associated sensor.

Describe one way the processor could be improved to cope better with this demand.

49. In the context of virtual memory as implemented by a given operating system:

a. Describe the resident set versus the working set of a process.

b. What is “thrashing” and how might one detect and resolve the problem?

c. List some advantages and disadvantages of increasing the page size (given that the physical and virtual address spaces are of fixed size).

50. a. List two reasons a process might be terminated.

b. List two reasons a process might be suspended.