CS 4500 - Operating Systems
Module 3: The Process Model, Threads, and Interprocess Communication

Stanley Wileman

Department of Computer Science
University of Nebraska at Omaha
Omaha, NE 68182-0500, USA

August 1, 2017
In This Module...

- The process model
- Process states
- Threads
- Interrupts, traps, and signals
- Interprocess communication
- Race conditions
- Solutions to the mutual exclusion problem
Even in single user systems, more than one potentially unrelated program may need to be active at the same time. This gives the effect of parallel execution of the programs, or pseudoparallelism.

A process is the encapsulation of all information about a running program, allowing the processor to be switched between them.
Why Use Multiple Processes?

A significant advantage of using multiple processes is that this may cause more processes to be available to use the processor(s). This increases the probability that the processor(s) will be active, yielding better utilization of the processor resource.

In general, choose to use multiple processes when

- A significant amount of overlap between the processes can be achieved; that is, one process is computing while another is doing input/output.
- An application is too complex to develop as a single process, or it would be too complex and difficult to maintain as a single process.
- There are multiple unrelated computations to be completed that do not interact with each other.
- The system has multiple processors. Indeed, the only way to make effective use of multiple processors (or cores) is to have multiple processes capable of using the processors.
There are, of course, disadvantages with multiple processes:

- With more processes there is likely more overhead time and more resource consumption by the operating system. This overhead does not directly contribute to the processing needed by the application.
- Increased programmer effort is also needed to coordinate the multiple processes and provide for communication between them.
- Additional processor features are required to ensure the security of the memory belonging to each process.
Process and Program Timing

- If a processor is dedicated to running a single program or process, then the process is run at the full speed of the processor; there is nothing except the process to capture the “attention” of the processor.
- If multiple processes are running, then the processor is continually being switched between the processes; each process gets only a fractional part of the total processor time.
- Since many programs are not time-critical, the process model is acceptable.


- **Program** is a static set of instructions and data for a processor.
- **Process** is the dynamic activity of a processor (CPU) in carrying out the instructions of a program.
- Input/output state, quotas, privileges, register contents, and so forth are all associated with a process, but not with a program.
Process Hierarchies

- A process can create additional processes using `fork()` or a similar system call.
- A process is called a child process of the parent process that created it.
- Each process can create an arbitrary number of their own child processes; the number of such child process is limited only by system resources (e.g. memory) and quotas.
- The result is that there is a tree-structured hierarchy of processes in the system to an arbitrary depth, again limited only by system resources and quotas.
System Services, or OS APIs

- System services are those functions provided by the operating system to applications.
- They are usually requested, or invoked, using instructions that cause a trap. For example, the Intel x86 processors provide the INT instruction that is often used for this purpose. Mainframes use the SVC instruction in a similar manner.
- Most modern operating systems are not written in assembly language. But some OS functions must utilize assembly language to some extent since it is the only language that is uniquely tailored to a unique processor.
Here’s a response to that question found on Quora:

- The Linux Kernel and most UNIX utilities, what you would call the operating system depending on how pedantic you want to be, are written in C.

- Windows (I am not authoritative so take this with a grain of salt) uses a kernel written in C, with most utilities and applications written in C++.

- Both iOS and OSX (the “Apple OS”), are written in a combination of C and Objective-C, C is used for the mach kernel and other lower levels. Objective-C is used for the higher levels.

- Android is built on top of Linux, and a large portion of it is written in C. There are swaths of code written in C++ though. Also, most of the application framework (the stuff you would call android, and almost everything you see on screen) is written in Java.

- Generally speaking, C was created alongside UNIX and is a fantastic language for writing things like operating systems. In all of my above examples, assembly is used for performance-critical situations. However, the entire operating system is not written in assembly because the resulting code would be unportable and hard to
Here’s a list of the languages used for major operating systems. It’s not completely verified, since some systems are proprietary. Also these are the languages used for the kernel, not necessarily all the applications.

- **UNIX** and Linux – C
- Microsoft Windows – C
- Apple’s iOS and OSX – C for the Mach \(^1\) kernel, and Objective-C
- Android – built on Linux, large parts written in C

In the “early days” operating systems were written entirely in assembler. Now, with the complexity of most systems, assembler coding is minimized to just those parts that are architecture dependent. Otherwise the resulting code would not be portable, and would be hard to maintain.

\(^1\)Mach is a UNIX-like system
Libraries and Direct System Calls

Virtually all high-level languages include libraries of functions (methods) to be used by applications written in that language.

Some of these functions are included to provide a convenient way to invoke operating system services. As we have seen, the C library includes functions for virtually every system call available on the host system. For example:

- the C library on Linux includes functions for open, close, read, write, fork, execve, ...
- the C library for Microsoft Windows includes functions for CreateProcess, CreateFile, ReadFile, WriteFile, ...

But most language libraries also include functions—that may also use system calls—to implement their own unique features, like the input/output functions used in standard C, Java, C++, and other languages. Using these libraries makes applications more portable, even though the language-specific libraries must be rewritten for each host system.
The Window’s CreateProcess system call requires the caller to give values for 10 parameters:

- **ImageName** - full path and file name of the module to execute
- **CommandLine** - string with command line (e.g. parameters)
- **ProcessSecurity** - security descriptor for the new process
- **ThreadSecurity** - security descriptor for the thread in the new process
- **IneritHandlesFlag** - Boolean: will child inherit handles from parent?
- **CreationFlags** - various flags, including process priority class
- **Environment** - as in **UNIX**, an array of “name=value” strings
- **CurrentDirectory** - as in **UNIX**, the current working directory
- **StartupInfo** - things related to a console or window for the process
- **ProcessInfo** - structure that receives process and thread identifiers

Wow! Compare that with the information needed for **UNIX** fork and exec system calls.
The Windows \_spawnvp Library Function

To contrast with the low-level CreateProcess system call for Windows, consider the C library function \_spawnvp that can also be used to create a new child process. It has three parameters:

- **mode** - if this is \_P\_WAIT the parent waits for the child to terminate. If it is \_P\_NOWAIT the parent proceeds without waiting for the child to terminate.

- **commandname** - the name of the file containing the executable code for the program to be executed by the child process

- **argv** - an array of pointers to parameters, exactly as would be provided to the **main** function in a C program

Notice that only three parameters are needed for this library function. It should be clear, however, that it will depend on the CreateProcess system call for its implementation.
During its lifetime, a process will always exist in one of three states: ready, running, or blocked. The state of a process is determined by the operating system, and is recorded in its process table entry.

Many factors can affect the state of a process, but there are some general characteristics that apply to each process in a given state.

\(^2\)Additional names for process states may be used in some systems, but these three are the fundamental ones.
A process in the **running** state is currently being executed by a processor. The process has all necessary resources required for execution. We’ll assume the green ellipse in the diagram below contains the set of all running processes.

![Running](image-url)
A process in the **ready** state has all the resources required for execution except a processor. We’ll assume all the ready processes are represented by the yellow ellipse in the diagram.
Process States: Blocked

A process in the **blocked** state is not capable of being executed because it lacks one or more required resources, like memory, completion of an I/O operation, or expiration of a timer on which it is waiting. The process is said to be **waiting** on a resource, **blocked**, or **sleeping**. We assume all blocked processes are represented by the red ellipse.
- **Running**: currently being executed by a processor, and has all necessary resources required for execution
- **Ready**: has all necessary resources required for execution except a processor
- **Blocked**: not capable of being executed because one or more required resources are unavailable. The process is said to be waiting on a resource, blocked, or sleeping.
In response to system conditions and process actions, the operating system will move processes between the running, ready, and blocked states. There are four transitions of primary interest to us:

- Ready to Running
- Running to Blocked
- Blocked to Ready
- Running to Ready
A process is moved from **Ready** to **Running** when a processor becomes available, and the **scheduler** component of the operating system selects the process for execution from the set of Ready processes.
A process is moved from **Running** to **Blocked** when it requests the use of a resource that is not currently available.
A process is moved from **Blocked** to **Ready** when the resource on which it was waiting is made available to it.
A process is moved from **Running** to **Ready**, or preempted when the operating system determines that the process should be replaced in the Running state by a more appropriate process. (We will later consider various reasons for process preemption.)
Process State Transition Summary

Process transitions we have considered:

- **Ready** to **Running**: the scheduler selects the process to run on an idle processor
- **Running** to **Blocked**: the process requests use of an unavailable resource
- **Blocked** to **Ready**: the resource on which the process was waiting becomes available
- **Running** to **Ready**: the operating system decides another process should be run
A process blocks when it cannot proceed because it needs the use of a resource that is not immediately available.

Examples:
- A `read` system call can result in a process blocking until the requested data is obtained.
- An explicit time delay blocks the process requesting the delay.

Resources that can be provided quickly don’t result in blocking (for example, getting additional memory—if it is immediately available, or getting the current time of day).

Other terms for blocking include **sleeping** or **waiting**.
A process can, in many operating systems, voluntarily relinquish its use of a processor.

Or the system may decide to forcibly remove it from the Running state for a variety of reasons. Here are the most common:

- A more appropriate process has entered the Ready state. Many systems give each process a processor scheduling priority, and a higher priority process may have become ready.
- A process in an interactive environment may have used a substantial amount of processor time (we'll see this is called a quantum later), and the processor should be used to execute another process for its “fair share” of the processor’s time.
A process is allowed to execute when the scheduler selects it from the set of processes in the Ready state.

There are many different scheduling algorithms used in operating systems, including round robin, priority, and shortest job first. We will examine some of these later.

Moving a process into or out of the Running state requires that all processor-specific registers, and perhaps other information about the process, be saved or restored. This action is called a context switch, and often accounts for the major time overhead associated with the operating system.
A process moves from the Blocked state to the Ready state when the resource(s) for which it was waiting becomes available.

No context switch is required for this transition, since there is no processor context to be saved or restored.

Note that there is no transition directly from the Blocked state to the Running state. If the unblocked process is to replace a running process, then the scheduler will make that decision and effect a separate Ready to Running transition for the process.
Process Tables

- The state of each process and other process-related information is stored in its entry in the process table.
- The process table entries are often elements of an array, but other data structures may be used for the process table.
- A static-sized process table obviously limits the number of processes that may exist in the system, but dynamically sized process tables are more complex.
- The contents of process table entries depend on the particular operating system.
struct proc {
    struct proc *p_forw;         /* blocked process queue */
    struct proc *p_back;         /* (doubly-linked) */
    struct proc *p_next;         /* list of active procs */
    struct proc **p_prev;        /* (ready) and zombies. */
    struct pcred *p_cred;        /* Process owner’s identity. */
    struct filedesc *p_fd;       /* open files structure. */
    struct pstats *p_stats;      /* Accounting/statistics*/
    struct plimit *p_limit;      /* Process limits. */
    struct vm-space *p_vmspace;  /* Address space. */
    struct sigacts *p_sigacts;   /* Signal actions, state */
    pid_t p_pid;                 /* Process identifier. */
    struct proc *p_pgrpnxt;      /* next proc in process grp. */
    struct proc *p_pptr;         /* proc struct of parent. */
    struct proc *p_osptr;        /* older sibling processes. */
    struct proc *p_ysptr;        /* younger siblings. */
    struct proc *p_cptr;         /* youngest living child. */
    ...
The term *interrupt* is used to describe the mechanism used by computer systems to indicate to a processor that an event requires attention and to communicate information about the event. In general, there are three stages in interrupt activity:

- **Interrupt Request Generation** – When a device determines that an event requires attention, it generates an interrupt request. For example, the completion of a disk read operation will usually cause the generation of an interrupt request.

- **Interrupt Recognition** – Before an interrupt request can be dealt with it must be recognized. There are mechanisms in most systems to disable interrupt recognition, either globally or for interrupt requests from specific sources.

- **Interrupt Handling** – Once an interrupt request has been generated and the request has been recognized, the last step is for the interrupt to be handled. That is, the system must take actions to deal with the event that caused the initial request.
Interrupt Request Recognition

- Interrupt request generation is often asynchronous with the processing of instructions. That is, an interrupt request may be generated at an arbitrary time, even during the processing of an instruction.

- The event causing an interrupt request may have nothing to do with the currently running process.

- In many systems, interrupt requests have associated priorities, with faster and more important devices generating higher-priority interrupt requests.

- Recognition of specific interrupt requests may be masked. This means that the recognition hardware has been told to ignore specific requests. This does not mean that a masked interrupt request is discarded; it is simply ignored, for now. That is, the recognition hardware’s “mask” prevents it from seeing the request.

- Certain classes of interrupt requests, usually those generated by devices other than the processor, can be globally disabled. This means no requests from those interrupt classes will be recognized.
An Interrupt Request Masking Analogy

Assume you are cooking something in your kitchen – that is, using the stove. Also assume your child is outside playing. Then consider the following sequence of events.

- Your child begins crying, shouting about a bee sting.
- Although you hear the crying, you don’t instantly go outside to deal with the event; instead, you carefully turn off the stove. (It wouldn’t be good for your child to be stung by a bee and for the house to burn down.)
- Once the “current instruction” has been completed, you rush outside to help.
- While looking for the bee sting, your telephone begins ringing. Assuming the bee sting “interrupt request” is of higher priority, you ignore the ringing telephone; you can later return the call.
- Once you child’s bee sting has been dealt with, you can choose to deal with the phone call (if it’s important), or you can return to the kitchen to continue your cooking.
Handling an Interrupt

- The term **interrupt handler** is used to describe the code module that is executed once an interrupt request has been recognized by the hardware.

- Interrupt handlers must obviously deal with the specific cause of an interrupt request, but some parts of the handler may be common to all types of interrupts. For example, it is usually required that processor context (e.g. register contents) that may be modified during interrupt handling be saved prior to executing the interrupt handler, and later restored before returning to the code for the interrupted process.

- Interrupt handlers, in general, may not block as they are **not** considered part of a process, including the process that was interrupted when the interrupt request was recognized.
Vectored Interrupts

Many systems associate a unique integer with each interrupt source. For example, Intel x86 hardware uses 0 for a divide error, 6 for an invalid operation code, and 12 for a stack segment violation.

Such systems often use the interrupt code as an index to an array of information about the interrupt type. This array is usually called an interrupt vector. The Intel x86 architecture uses the name interrupt descriptor table (IDT) for this array.

Each entry in an interrupt vector contains information about the processing to be performed for the interrupt request. In particular, it identifies the location of the interrupt handler code module.
Traps are very similar to interrupts. However traps occur synchronously with the processing of instructions. That is, traps are generated as a result of the processing of an instruction.

A trap can be generated as a result of a problem (like dividing by zero or attempting to access an unallowed memory location). For example, consider the Intel x86 DIV instruction. If the divisor is zero, then the instruction generates a divide error.

But traps may also be explicity generated by special instructions provided for that purpose. The Intel x86 INT instruction and the mainframe SVC instruction each result in a trap being generated. This is the mechanism often used by operating systems to make a system call.

Unlike an interrupt, traps are always generated by the currently running process, and cannot be blocked or masked (like interrupts).

UNIX systems usually translate traps into signals that are then delivered to the process that generated the trap.
Handling an Interrupt

1. The interrupt (if not blocked or masked) is reported to the processor, usually only after the completion of the current instruction.
2. The hardware saves some processor registers (typically on a stack).
3. The program counter (and perhaps other registers) are set from a predefined location in the interrupt vector.
4. The remaining volatile registers are saved by software.
5. A new stack is established.
6. The interrupt handler processes the interrupt (perhaps starting I/O) and possibly unblocks a process.
7. The scheduler may intervene to select a new process to execute.
8. A context switch may take place, and the new process begins/resumes execution.
Handling a Trap

1. The trap is reported to the processor, synchronously with the execution of the current instruction (that cased the trap).
2. The hardware saves some processor registers (typically on a stack).
3. The program counter (and perhaps other registers) are set from a predefined location determined by the trap type (e.g. divide by zero, illegal instruction).
4. The remaining volatile registers are saved by software.
5. A new stack might be established, if necessary.
6. The trap handler deals with the trap, often arranging for a signal to be sent to the process.
7. The process that generated the trap may continue (to handle the signal delivery), or a different process may be selected to run.
A signal is the user-level software equivalent of a hardware interrupt or trap.

As such, there are three stages in signal processing:

- **Signal generation** – this can be done explicitly in software, implicitly as a result of software-related exception (like a timer expiration), or implicitly as a result of a trap (like divide by zero).
- **Signal delivery** – this occurs when a signal is actually delivered to a process for handling. Like interrupts, signal delivery can be affected in several ways: signals may be ignored, blocked, or masked.
- **Signal handling** – as you’d expect, a signal handler is the code module used to deal with a signal. Each signal handler for a process must be explicitly registered with the operating system.
An Example of Signal Handling

- The simple program shown on the next two slides (and in the file `zdiv.c` in the class directory) illustrates some essentials of signals and signal handling in C.
- When executed, the process asks you if you want to use a signal handler; answer `y` or `n` and press the enter key.
- The program will then read an integer $k$, and the computation of $4/k$ will be attempted.
- If $k$ is not 0, then the computation proceeds as expected—no trap or signal is generated, and the program displays the appropriate result.
- But if $k$ is 0, the divide exception, or trap, occurs. How the program deals with this exception depends on the presence or absence of a signal handler:
  - With no signal handler, the message `Floating point exception (core dumped)` will be displayed and the program will be terminated.
  - With a signal handler, the output indicates that a signal was delivered to the process, and the process was still in control of the execution.
Example Signal-handling Program (Part 1)

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
/*---------------------------------------------------------*/
/* The function sigh is the signal handler. It should have */
/* void return type and a single integer parameter (which */
/* will receive the signal number. */
/*---------------------------------------------------------*/
void sigh (int signum)
{
    printf("Signal %d was delivered to us.\n", signum);
    printf("But we are still in control!\n");
    exit(1);
}
```
int main() {
    int k, ans, c;
    printf("Use a signal handler (y or n)? ")
    if ((c = getchar()) == 'y')
        signal (SIGFPE, sigh); /* setup handler for SIGFPE */
    else if (c != 'n') {
        printf ("You must answer y or n.\n");
        exit(1);
    }
    printf("Enter an integer k. We will compute 4 / k.\n");
    printf("k = ");
    scanf("%d", &k);
    ans = 4 / k; /* SIGFPE will occur if k == 0 */
    printf("4 / k = %d\n", ans);
    exit(0);
}
A thread, sometimes called a lightweight process, is conceptually similar to a process in that it represents a separately scheduled unit of work for a processor.

A thread has some private memory and its own set of processor register contents, including the program counter, but shares all other resources (memory, quotas, open files, etc.) with all other threads belonging to the same process.

In systems with threads, the process can be considered the owner of the resources (memory, etc.) while the threads represent the units of execution for the processor.
When To Use Threads

- Threads are appropriate tools to use when multiple independent execution paths (or computations) need access to the same address space and other resources.
- For example, consider a program such as a terminal emulator, like putty. It must, in effect, simultaneously do several things:
  - Wait for input from the keyboard. When it arrives, process it, perhaps encrypting it and send it to a remote system, or treating it as a command to be processed locally.
  - Wait for a network packet from a remote system. When it arrives, decrypt it and display it, taking into account all the various embedded commands that might affect how it is displayed.

Since we don’t know whether the next event will be input from the keyboard, or input from the network, we can use two threads, each of which is scheduled independently of the other.
Threads are commonly implemented either as integral parts of the operating system kernel or as user-level library functions.

In kernel implementations, kernel intervention is required on every switch between threads, even between those in the same process.

In user-level implementations, the kernel is not aware of threads, and may block an entire process even though other threads in the process could run.
Complications With Threads

- On a fork call, does the child get as many threads as the parent, including blocked threads?
- What happens if application library functions are not reentrant?
- How are detailed error reports provided? Recall a single errno variable is used for each UNIX process.
- To which thread is a signal delivered?
- To which thread is an input report delivered?
- How is the size of a thread’s stack determined?
Common Thread Implementations

- POSIX (IEEE standard Portable Operating System Interface)
  Pthreads available for numerous systems (including Linux, OS X, and Windows)
- Mach threads found in OSF/1 and Digital UNIX
- Win32 implemented in Microsoft Windows
Processes (and threads) need to be able to communicate with each other to achieve synchronization, share data, etc.

Many different techniques have been devised to effect this communication.

Even unrelated processes may need to synchronize their access to shared resources.
Race Conditions

- Processes that use shared resources must be careful to ensure that modifications to those resources are done in a consistent manner.
- We use the term **atomic** to refer to an operation on a shared resource that must be completely performed by a process before any other process can use the resource.
- Failing to guarantee atomic execution can result in inconsistent results. This potential is called a **race condition**.
Suppose we have two threads in a process, each of which has access to a global integer variable Z; Z is initially zero.

Each thread is very simple. It performs a fixed number of repetitions of a loop. Inside the loop, the variable Z is first incremented (by 1), then it is decremented (by 1).

When both threads have finished, what is the expected value of Z? (Compile/run race.c found in the class directory)
Depending on a processor's architecture, incrementing or decrementing a variable can be done in various ways.

Some processors may have specific instructions to increment or decrement a variable stored in primary memory (RAM).

With other processors it may be necessary to copy the variable to a register, increment or decrement the register, and then store the result back into primary memory.
Specific Code Example (Intel x86)

Here is specific code for increment and decrement generated by the C compiler on a Linux system. The `movl` instruction is used to move (copy) a value from one place to another. $Z(%rip)$ is a reference to a memory-based volatile variable $Z$, and $%eax$ is a processor register. $1$ is the constant 1. The `addl` and `subl` instructions should be obvious. (Use the `-S` option with gcc to generate assembler code.)

```
  movl  Z(%rip), %eax
  addl  $1, %eax
  movl  %eax, Z(%rip)
  movl  Z(%rip), %eax
  subl  $1, %eax
  movl  %eax, Z(%rip)
```
The result in $Z$ could be zero, or it could be non-zero. There is no guaranteed method to determine the answer that will be produced. Clearly the desired result should be zero. Can you see how these results could be obtained? Consider all of the possible execution sequences that could be used by a processor when it is executing the code shown in the example program.
The Moral of the Story

- Situations like this, involving unconstrained access by two or more processes to shared resources, result in race conditions.

- Race conditions are usually difficult to detect, since they may depend on execution timing (which may differ between executions of the same code with the same data).

- The only feasible solution to race conditions is to properly design the programs to synchronize access to the shared data resources.
The region of a program in which a shared resource is accessed is called a **critical section**.

Avoiding race conditions requires guaranteeing **mutual exclusion** between the critical sections referencing the same shared resource.

Other conditions must hold for a good solution to the race condition problem.

The program `nrace.c` in the class directory shows one solution to the race condition problem we’ve just seen.
1. No two processes may be simultaneously inside their critical sections.
2. No assumptions may be made about the speeds of, or the number of processors.
3. No process running outside its critical section may block other processes.
4. No process should have to wait forever to enter a critical section once it has indicated its desire to do so.
In the remainder of this module we will consider some early efforts made to solve the mutual exclusion problem, including some that are successful and some that are unsuccessful. Specifically, we will consider these:

- Busy Waiting
- Petersen’s Solution
- Hardware Instruction Assistance
- The Priority Inversion Problem
- Sleep and Wakeup
Q. How is it that a processor, executing the code for the critical section of some process, is “redirected” to execute code for some other process?

A. It requires an interrupt or trap to divert the processor from the code it is executing. This could be an interrupt from a timer (remember the process state transitions), or an interrupt from almost any source.
To prevent the processor from being “yanked away” from a process, interrupt recognition could be disabled just before entering a critical section, and then reenabled after the critical section was completed.

Unfortunately, this solution has several problems...
Problems with Disabling Interrupt Recognition

- The solution only works for systems with a single processor. If two processors sharing memory are executing two processes, then both processes can still access the shared memory location.

- It is a dangerous approach, since an simple coding error could result in failing to reenable the interrupts, which will prevent the processor from recognizing and handling any more interrupts!

- And, of course, disabling interrupt recognition is a privileged operation, only permitted in kernel mode.
The basic idea here is to use a variable (lock) that is either 0 (indicating the resource is free) or 1 (indicating the resource is in use).

To enter the critical section, execute code similar to this:

```c
while (lock == 1); /* wait until free */
lock = 1; /* set the lock */
```

To exit from the critical section:

```c
lock = 0; /* release the lock */
```

Unfortunately, there is a race condition here! Can you see it?
Suppose we have three processes, A, B and C using the resource. C is in its critical section, so lock = 1. A and B are “busy waiting” for lock to become 0.

Immediately after C sets lock = 0 (on exit from the critical section), both A and B might determine lock = 0 and then proceed to set lock = 1 and enter their critical sections, both at the same time.

Remember that we may make no assumptions about the number of processors, so A and B could each be executing on separate processors.

Even if they were sharing a single processor – that is, if their execution was being multiplexed on a single processor, it would still be possible for each of them to enter their critical sections at the same time. It would be useful for you to examine the execution sequence and process state transitions that could result in this behavior.
Suppose there are just two processes, A and B, with B in its critical section and A waiting to enter.

B finishes (setting lock = 0), then A tests lock and finds it to be 0. But before A has a chance to set lock = 1, B attempts to enter its critical section again, finds lock = 0, and proceeds to set lock = 1.

Later, A continues to execute. Having already tested lock and found it 0, it proceeds to set lock to 1 and enter its critical section.

The problem here is that the testing of lock by process A is not immediately—atomically—followed by the setting of the lock variable. This lack of an atomic test and set is common in failed solutions to the mutual exclusion problem.
Busy Waiting with Strict Alternation

Instead of using a lock variable, which leaves a window between testing the lock and setting it to 1, let’s use a “turn” variable that indicates which process is to be allowed entry to the critical section.

Again assume we have two processes, A and B.

Here’s the code for process A; B’s code is similar:

```c
while (TRUE) {
    while (turn == 'B') /* wait */ ;
    ... A’s critical section ...
    turn = 'B';
    ... A’s other code ...
}
```
Problems with Strict Alternation

- Strict alternation, although it avoids allowing multiple processes inside their critical sections at the same time, doesn’t adapt well to varying time requirements, and doesn’t meet our third solution requirement: “No process running outside its critical section may block other processes.”

- Busy waiting solutions, in general, also have the problem of wasting processor time while a process is waiting for entry to the critical section.

- But busy waiting may sometimes be unavoidable, so such solutions (when correctly implemented) still have application in operating systems.
T. Dekker, a Dutch mathematician, was the first person to devise a correct solution to the mutual exclusion problem without requiring strict alternation.

His solution only depends on atomic memory writes. That is, if two processors both attempt to write to the same memory location at the same time, the hardware guarantees that the two writes will be done atomically, one after the other.
G. L. Peterson, in 1981, discovered a simpler solution to the problem (Dekker’s solution is rather cryptic).

His solution utilizes a turn variable (as did the last solution), but only to resolve ties between processes wanting to enter the critical section at the same time.

It also uses an array, interested, to show which processes want to enter their critical sections.
Using Peterson’s Solution

- We consider only two processes, numbered 0 and 1.
- When a process wants to enter its critical section, it calls the function `enter_region` with its process number as an argument.
- When a process leaves its critical section, it calls the function `leave_region`, again with its process number as an argument.
- Each entry in the interested array is initially FALSE.
void enter_region(int process) {
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process &&
        interested[other] == TRUE) {
        /* wait */
        ;
    }
}
The “leave_region” Function

```c
void leave_region (int process)
{
    interested[process] = FALSE;
}
```
Historically, problems dealt with in software have often been addressed by later hardware enhancements.

Modern processors usually have hardware features to help solve the mutual exclusion problem. For example:

- IBM mainframe computers have the TS instruction (the acronym means Test and Set).
- The Intel x86 architecture has the XCHG instruction.

Each of these instructions operate on two data items, one in (possibly shared) memory, and the other in a register or processor flags unique to a processor.
The result of comparisons, arithmetic, and other operations are recorded in a set of flags called the *condition code*. The condition code is part of a processor register. It can be tested by conditional branch instructions.

The TS instruction is coded as

\[ \text{TS } M \]

where M is an integer operand in shared memory.

The instruction atomically does this:

1. The value of the memory operand (M) is used to set the condition code to indicate if it is zero or nonzero.
2. M is set to all ones.
Assume global variable M is used to indicate if a process is inside its critical section (1) or not (0).

The enter_region function (in assembler) might look like this:

```
ENTER_REGION:
WAIT: TS M TEST IF M == 0; SET IT TO ALL ONES
     BC 8,WAIT REPEAT IF M WAS NOT 0
     BR 15 RETURN FROM FUNCTION
```

The leave_region function only has to set M to 0:

```
LEAVE_REGION:
XR R SET REGISTER R = 0
ST R,M STORE 0 IN M
BR 15 RETURN FROM FUNCTION
```
The XCHG instruction has two operands, one in a processor register and the other in shared memory.

The instruction atomically exchanges the contents of the operands.

The Intel instruction description includes this note: “this instruction is useful for implementing semaphores or similar data structures for process synchronization.” We’ll study semaphores in the next module.
Again assume a global variable m is used to indicate if a process is inside its critical section (1) or not (0).

The enter_region function (in assembler) might look like this:

```assembly
text
enter_region:
wait:    mov r,1    set reg r to 1
    xchg r,m    swap r and m (m set to 1)
    cmp r,0     compare r and 0
    jnz wait    repeat if m was not 0
    ret
```

The leave_region function only has to set m to 0:

```assembly
text
leave_region:
    mov m,0
    ret
```
Operating systems frequently associate a priority with a process.

A priority is just an integer, with larger values indicating higher priority.

The system’s scheduler is designed to guarantee that the process(es) with the highest priority are being actively executing (that is, they’re in the running state).

Of course there must be some way of handling cases where there are multiple processes with the same priority, and more processes with the highest priority than there are processors.
The Priority Inversion Problem

- Assume we have two processes, H and L, with high and low priority, respectively.
- The scheduler guarantees that whenever H is in the ready state it will be run (moved to the running state).
- Assume L is running and enters its critical section, and in the meantime H becomes ready, and wants to enter its critical section.
- H is moved to the running state, but no work will ever get done; H will loop forever, waiting to enter its critical section.
One solution to the priority inversion problem is to use an aging algorithm.

In this scheme, a process that is ready to run, but doesn’t get to run because of its low priority, is periodically given a priority “boost.”

Eventually its priority will be high enough to allow it to run, finish using the shared resource, and exit its critical section.
Each of the solutions thus far has the unfortunate problem of requiring a process to loop, continually testing some condition, until it can enter its critical section.

This busy waiting wastes enormous amounts of CPU time that could be better expended on other ready processes.
Sleep and Wakeup

- One approach to eliminate the busy waiting problem is to cause the process that would normally perform a busy loop to yield the processor, go to sleep (that is, move to the blocked state), and have some other process awaken it when the resource is available.

- One of the first approaches along these lines used a pair of functions called sleep and wakeup.

- The sleep function causes the calling process to block until it is explicitly awakened by some other process.

- The wakeup function causes the process identified by the argument to be awakened, that is, moved from the blocked to the ready state.
Using Sleep and Wakeup

- When a process discovers that it cannot enter its critical section it calls sleep, relinquishing the processor to other processes, one of which will eventually exit its critical section.
- When a process completes its critical section, it calls wakeup to awaken a process that is sleeping.
- We will see use of these functions in a proposed solution to the classic producer-consumer problem in the next module.
In the next module we will look at contemporary solutions to the mutual exclusion problem.

We will use the producer-consumer problem as the target problem for our discussion.

This problem appears in many places in concurrent programs, including the operating system itself.