

Lecture 2

Processes and Threads

Feb 02 2016

Learning Objectives

By the end of this lecture, you should be able to:

- Distinguish between programs, processes, and threads.
- Describe the data structures used by an OS to manage processes and threads.
- Identify different process and thread states.
- Identify race conditions.
- Identify conditions for mutual exclusion.
- Analyze some mutual exclusion strategies.

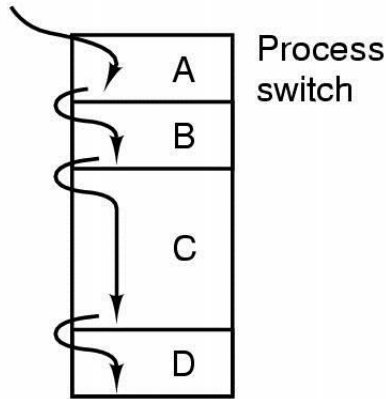
What is a process?

- A program is a string of bits interpreted by the processor as a sequence of executable instructions.
 - Unless explicitly modified by a programmer (or by some program), a program never changes.
- A **process** is a program in execution.
- It could be thought of as a dynamic data structure, one that possibly changes with each CPU cycle.
 - It includes program code, program data, values of registers, contents of stacks, etc.

Processes

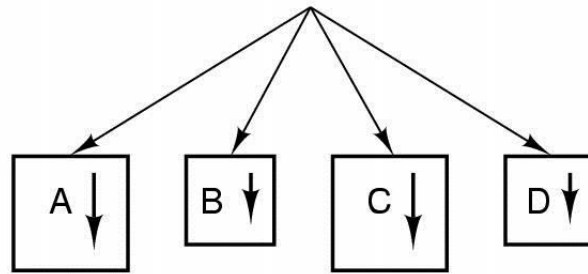
The Process Model

One program counter

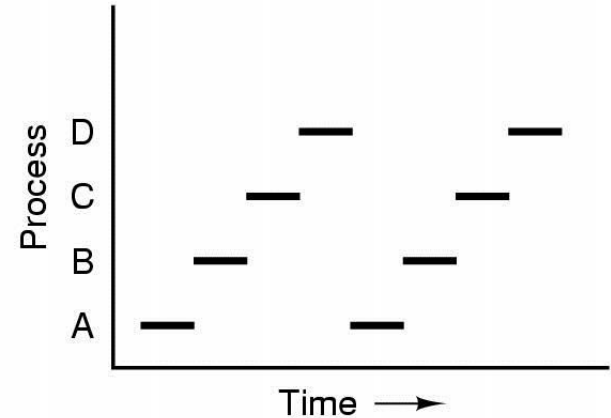


(a)

Four program counters



(b)



(c)

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Shared Resources

Processes may, and do, share several resources:

- They do share the same processor.
- They do share the same physical memory.
- They do share the same peripherals.
- They may share the same data.
- They may share the same program.

Process Creation

Principal events that cause process creation

1. New batch job
2. System initialization (terminal logon)
3. Created by OS to provide a service
4. Spawned by existing process

Process Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

Process Hierarchies

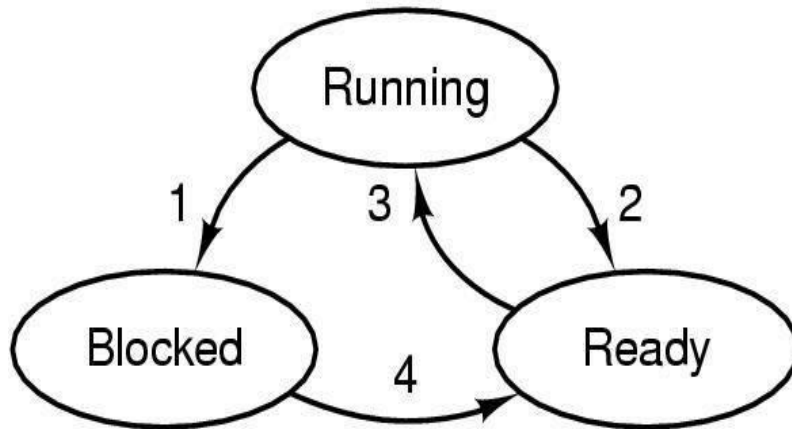
- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
 - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
 - all processes are created equal

Process Control Block

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

Fields of a process table entry

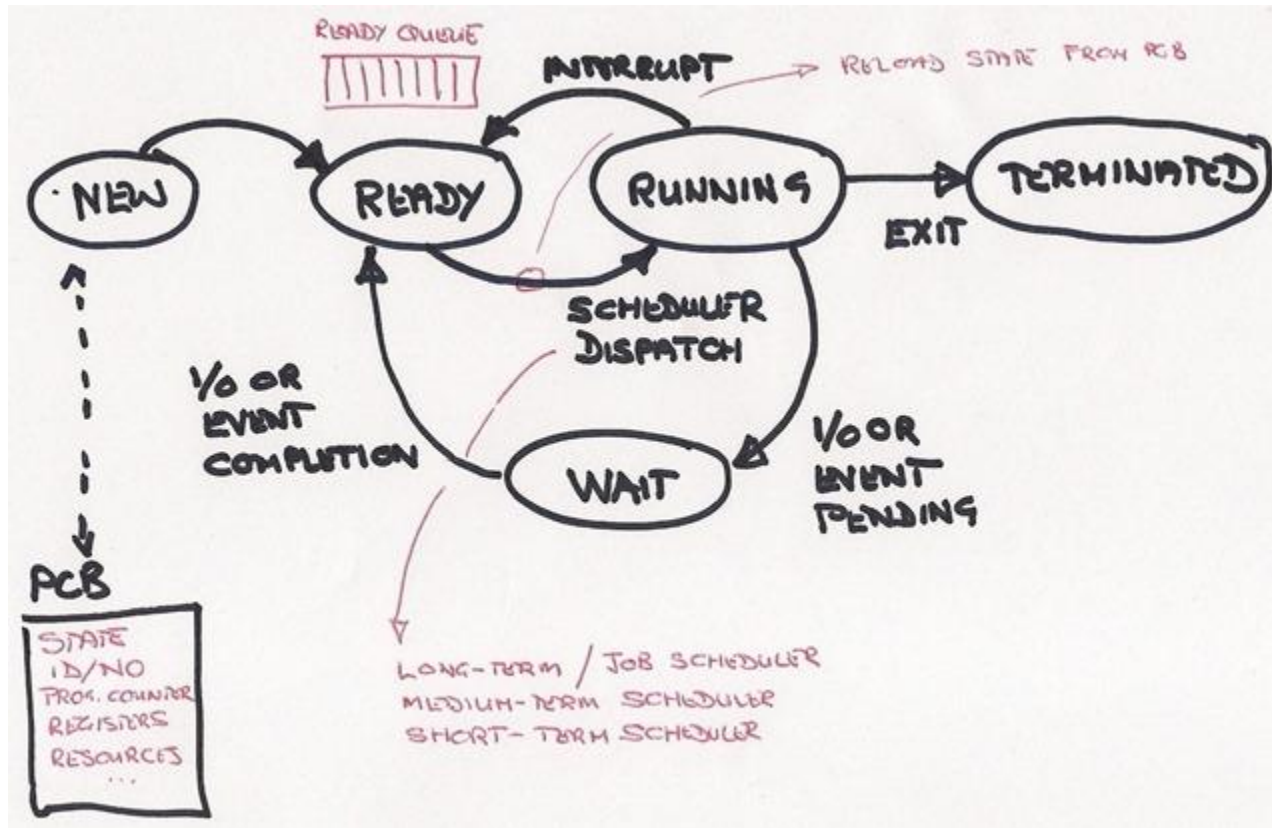
Process States (I)



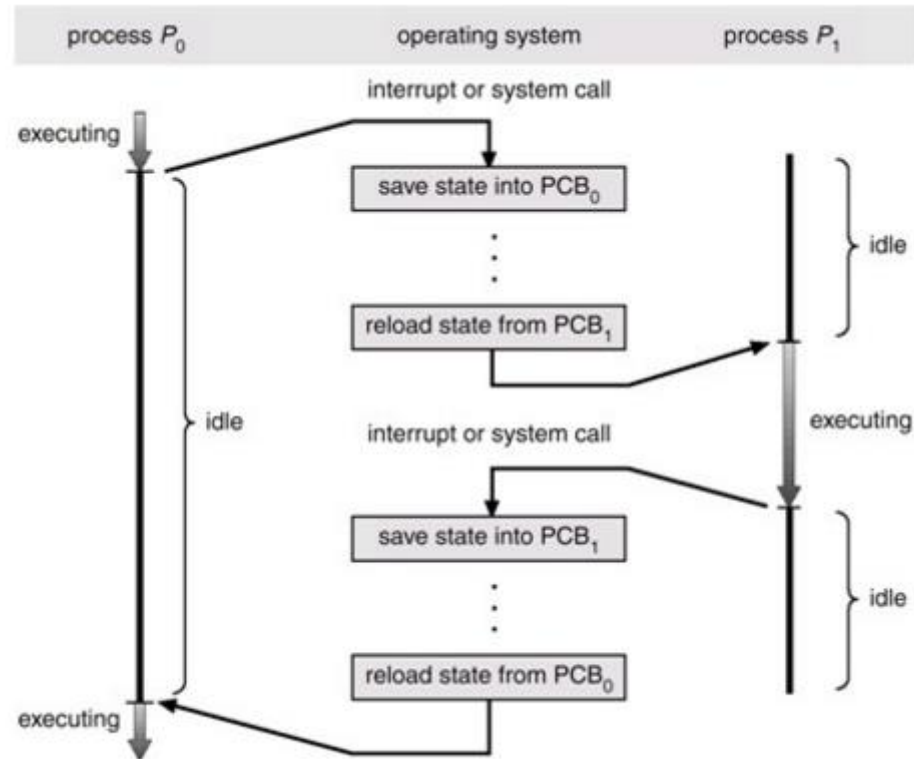
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
 - running
 - blocked
 - ready
- Transitions between states shown

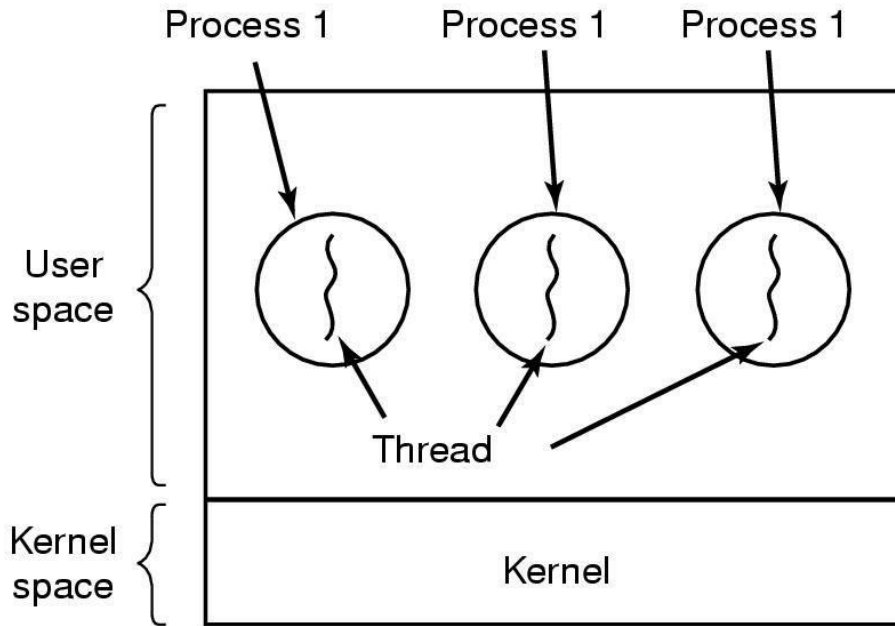
Process States (II)



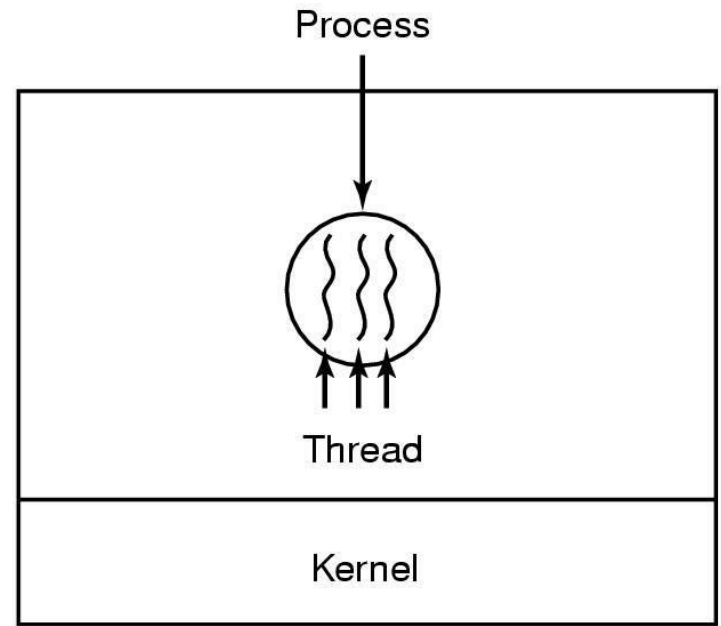
Process Execution



Threads



(a)



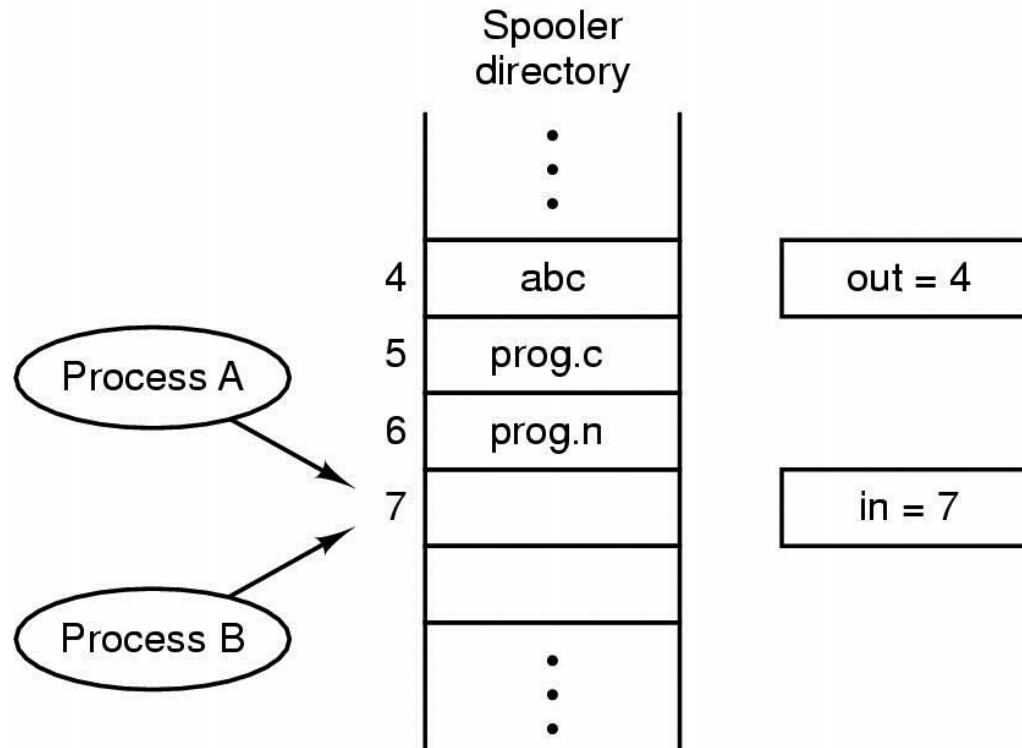
(b)

- A **thread** is a single flow of execution within a process.
- Threads of a single process share much information.
- Switching threads is much less expensive than switching processes.

Process Interaction

- In a multiprogramming environment, the OS has to manage the interaction among different processes.
- In particular, it should make sure that processes do not stand in each others' ways.
 - The same applies to threads. But threads belonging to the same process are expected to be cooperating rather than competing.

Race Conditions



Two processes want to access shared memory at same time

Mutual Exclusion

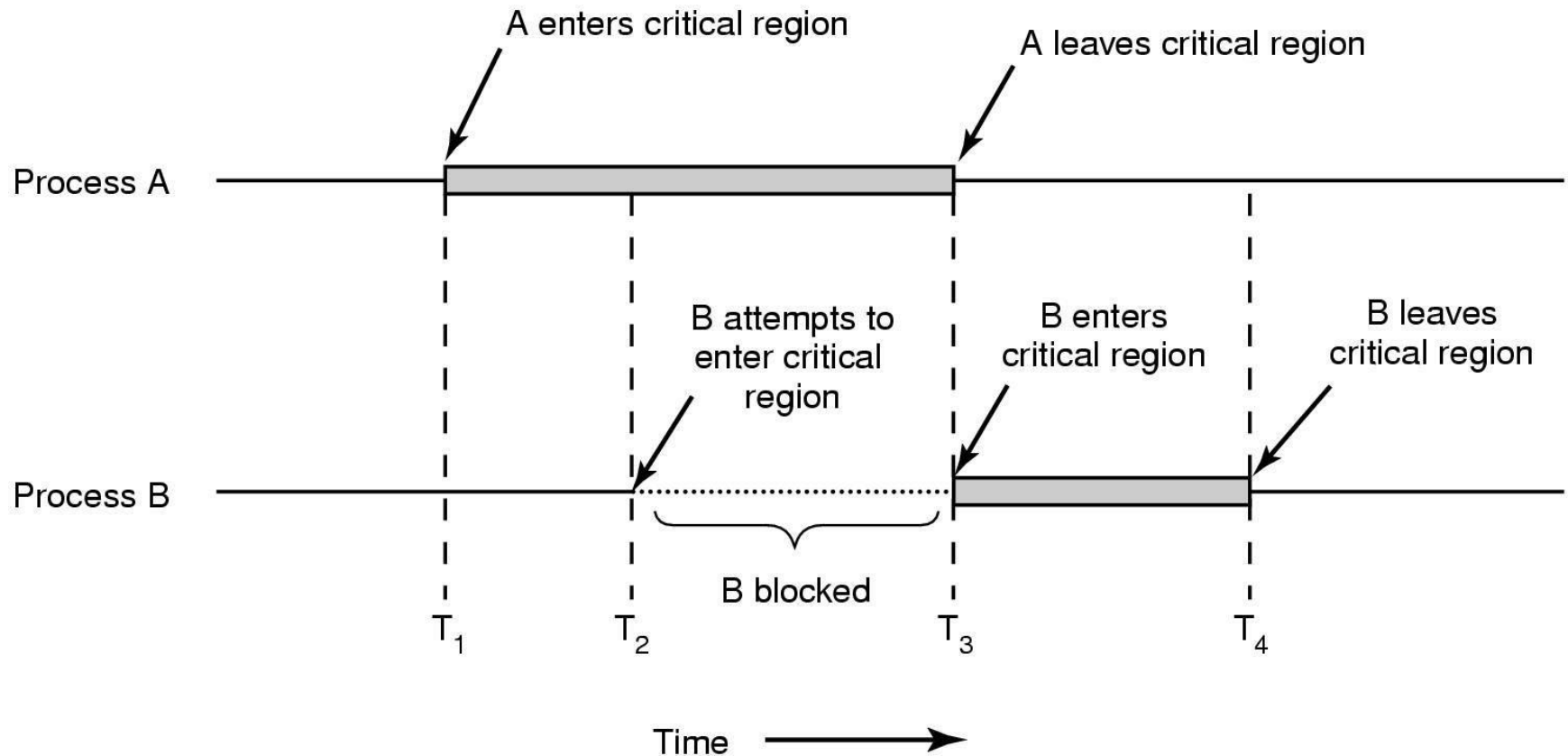
- **Race conditions** occur when two or more processes are using some shared resource whose state (from the point of view of each process) depends on who runs precisely when.
- Getting over race conditions lies in ensuring **mutual exclusion**:
 - Making sure that a process can use a shared resource only if it is not being used by another process.
- The part of a program that uses a shared resource is called a **critical region**.

Conditions for Mutual Exclusion

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region
2. No process must wait forever to enter its critical region
3. No process running outside its critical region may block another process
4. No assumptions made about speeds or numbers of CPUs

Critical Regions (2)



Mutual exclusion using critical regions

Implementing Mutual Exclusion

- Some initial ideas:
 - Disabling interrupts.
 - Lock variables.
 - Strict alternation.
- We'll consider each in detail.

Disabling Interrupts

- Once a process enters its critical region, it disables interrupts.
 - Thus, making sure that no other process will be scheduled.
- Re-enable interrupts just before leaving critical region.
- **Problem:** *User processes should not be entrusted with interrupts.*

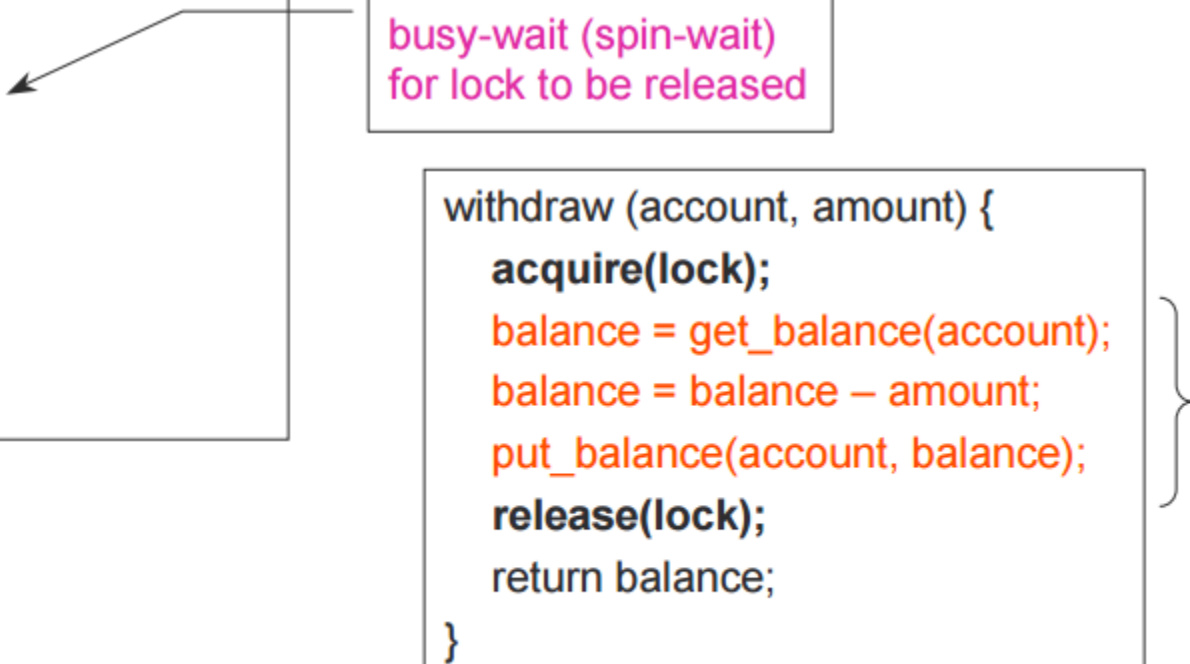
Lock Variables

- A shared Boolean variable indicates whether a process could enter its critical region.
 - Or, whether some other process is in its critical region.
- Prior to entering its critical region, process must check variable.
- If true, set it to false and enter critical region.
- Reset variable to true just before leaving critical region.

Lock Code

```
struct lock {  
    int held = 0;  
}  
void acquire (lock) {  
    while (lock->held);  
    lock->held = 1;  
}  
void release (lock) {  
    lock->held = 0;  
}
```

busy-wait (spin-wait)
for lock to be released



```
withdraw (account, amount) {  
    acquire(lock);  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    release(lock);  
    return balance;  
}
```

**Critical
Section**

- **Problem:** Race conditions may occur, with this variable as the shared resource.

Strict Alternation

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region( );  
    turn = 1;  
    noncritical_region( );  
}
```

(a)

(a) Process 0.

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region( );  
    turn = 0;  
    noncritical_region( );  
}
```

(b)

(b) Process 1.

Strict Alternation

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region( );  
    turn = 1;  
    noncritical_region( );  
}
```

(a)

(a) Process 0.

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region( );  
    turn = 0;  
    noncritical_region( );  
}
```

(b)

(b) Process 1

- **Problem:** *If one process is much faster than the other, condition 3 will be violated*

Next Time . . .

- Peterson's algorithm.
- The TSL instruction.
- Mutual exclusion primitives.

Points to Take Home

- Programs, processes, and threads.
- Process and thread management.
- Process and thread states.
- Race conditions.
- Conditions for mutual exclusion.
- Ensuring mutual exclusion.