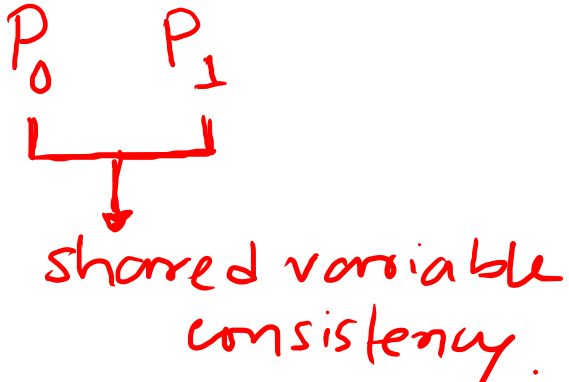
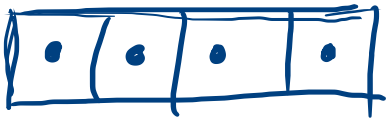


Chapter 5: Process Synchronization

Background

- **Concurrent access** to **shared data** may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the **orderly execution** of cooperating processes
- Suppose that we wanted to provide a solution to the **consumer-producer problem** that fills all the buffers(actually support bounded buffer). We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

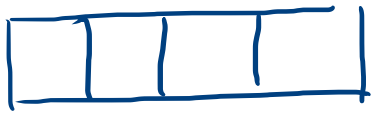
Producer




```
while (true)
   {
            /* produce an item and put in nextProduced
            while (count == BUFFER_SIZE);
                  // do nothing
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
   }
```

CS

Consumer



count = 0

```
while (1)
 {
        while (count == 0); // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        /*  consume the item in nextConsumed
 }
```

CS

Race Condition

- count++ could be implemented as

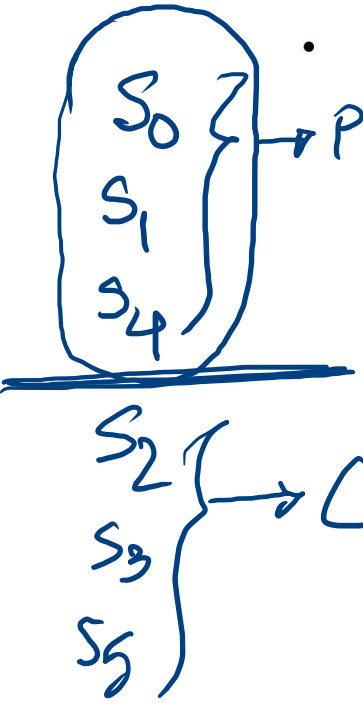
```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}



Race Condition

- A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called **race condition.**

- To guard against the race condition above, we need to ensure that only **one process at a time can be manipulating** the variable counter. To make such a guarantee , we require that the **processes be synchronized** in some way.

Critical Section Problem

Consider a system consisting of n processes $\{P_0, P_1, ..., P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.

1) entry section.
2) CS
3) exit section
4) remainder

Critical Section Problem

The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process P_i is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

Critical Section

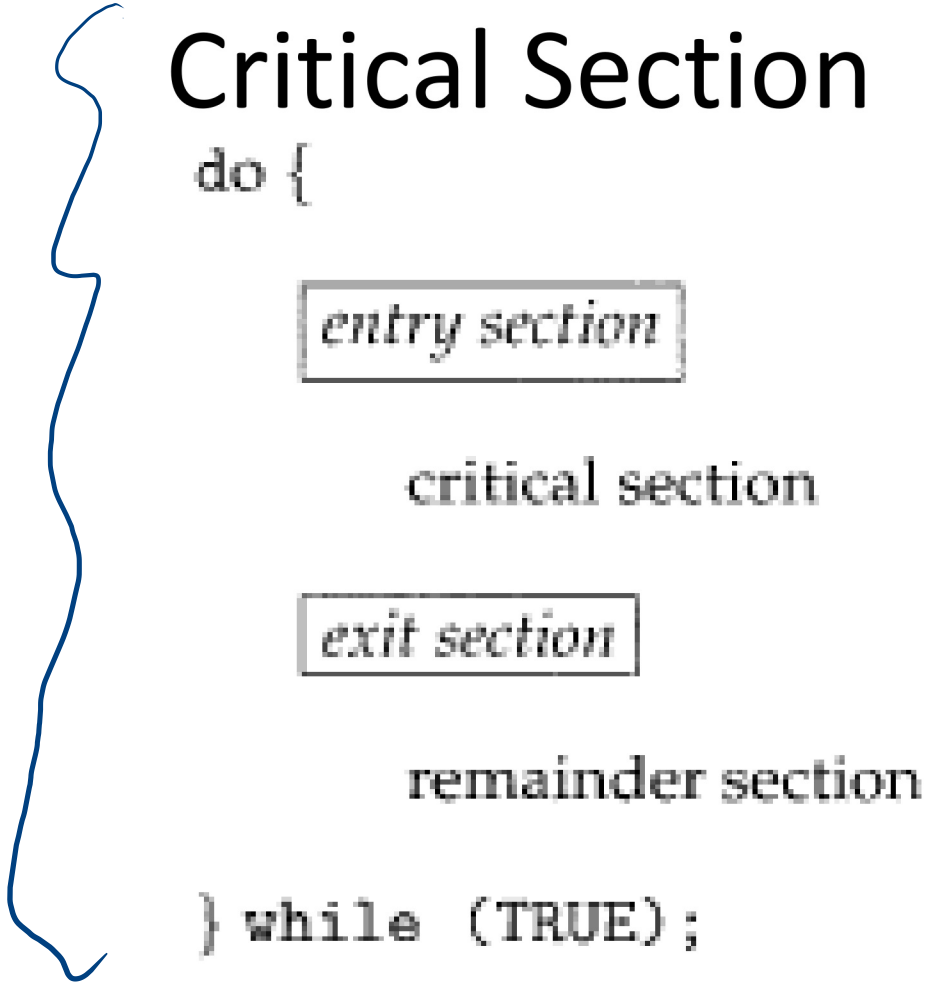


Figure 6.1 General structure of a typical process P_i.

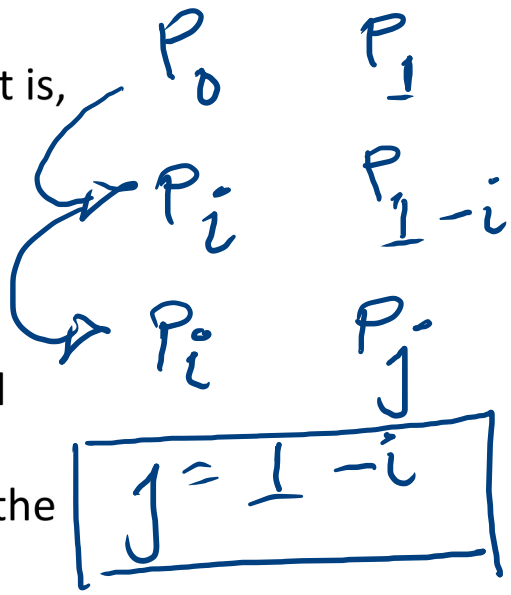
Solution to Critical-Section Problem

A solution to the critical-section problem must **satisfy** the following **three requirements:**

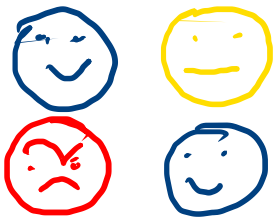
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no thread is executing in its critical section, and if there are some threads that wish to enter their critical sections, then one of these threads will get into the critical section.
2. **Bounded Waiting** - After a thread makes a request to enter its critical section, there is a bound on the number of times that other threads are allowed to enter their critical sections, before the request is granted.

Peterson's Solution

- A classic software based solution to the critical section problem known as Peterson's solution
- **Two process solution**
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two data items to be shared between the two process:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!



1. Mutual Exclusion
2. Progress.
3. Bounded waiting



Peterson's Solution for process P_i

i indicates the index of flag array
j indicates the value of the turn variable.

Process 0: i=0; j=1 – i = 1

```
do {
    S0  flag[0] = TRUE ;
    S1  turn = 1 ;
    S2  while ( flag[1] && turn == 1) ;
            T              1

        CRITICAL SECTION

        flag [0] = FALSE ;

        REMAINDER SECTION

} while (TRUE) ;
```

P0 S0
P1 S0
P0 S1
P1 S1
P0 S2

Process 1: i=1; j=1 – i = 0

```
do {
    S0  flag[1] = TRUE ;
    S1  turn = 0 ;
    S2  while ( flag[0] && turn == 0) ;
            T              0

        CRITICAL SECTION

        flag [ 1] = FALSE ;

        REMAINDER SECTION

} while (TRUE) ;
```

Assignment

- How to handle critical section problem in OS?
- Why would anyone favor a preemptive kernel over a non-preemptive kernel?

CS solution with locks using h/w

- A simple tool requires-lock
- Race conditions are prevented by requiring that critical section be protected by locks.

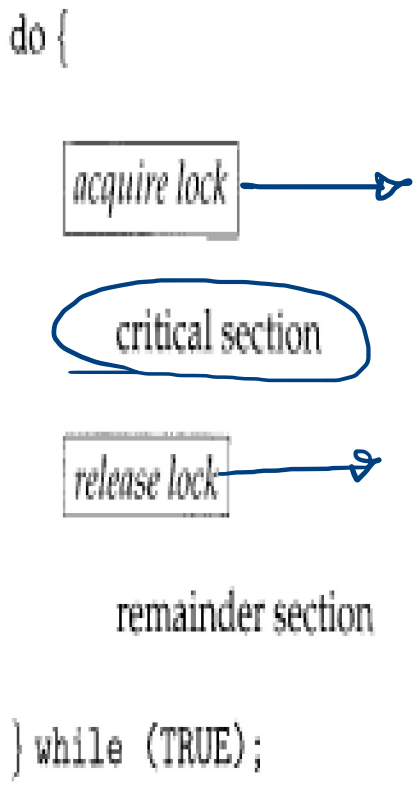
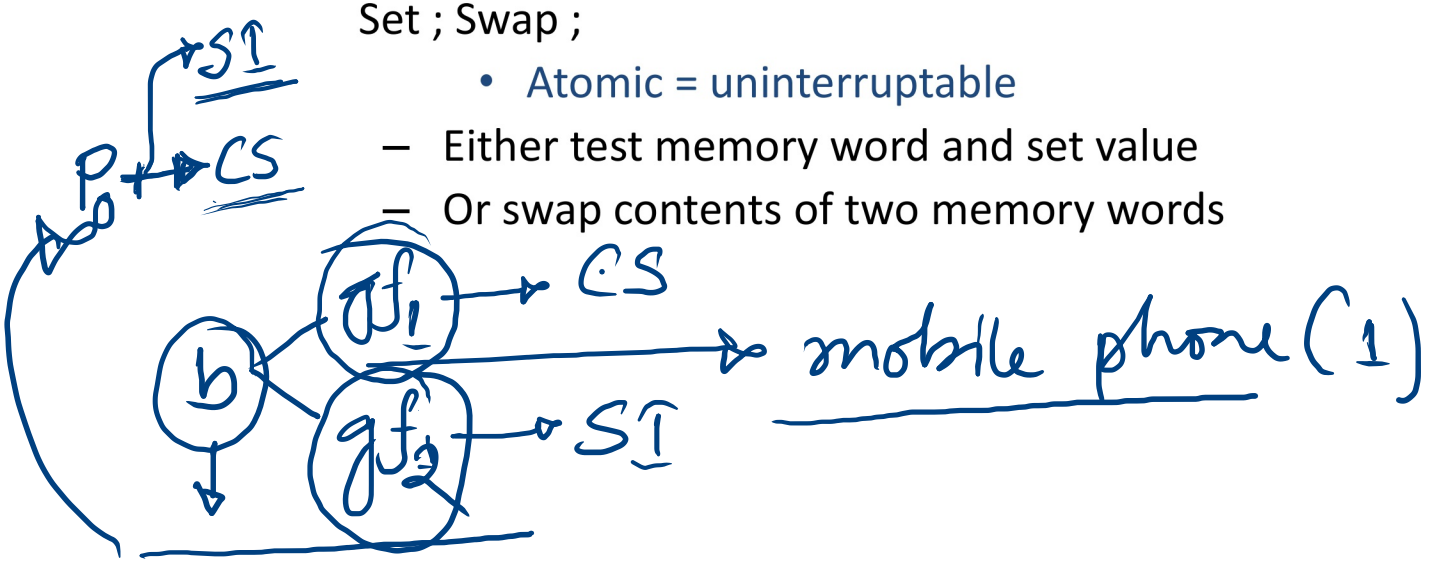


Figure 6.3 Solution to the critical-section problem using locks.

Synchronization Hardware

- Many systems provide **hardware support** for critical section code
- Uniprocessors – could disable interrupts while modified a shared variable
  - Currently running code would execute without preemption. So no unexpected modifications could be made to the shared variable.
  - This is the approach taken by non-preemptive kernels.
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable-time consuming.
- Modern machines provide special atomic hardware instructions: Test And Set ; Swap ;
    - Atomic = uninterruptable
  - Either test memory word and set value
  - Or swap contents of two memory words



Solution using TestAndSet

- Shared boolean variable **lock**, initialized to false.
- Solution:

```
do {
    while ( TestAndSet (&lock ));
        // critical section
    lock = false;
        // remainder section
} while ( true);
```

- Definition:

```
boolean TestAndSet (boolean *target){
    boolean rv = *target;
    *target = true;
    return rv;
}
```

P_0 : lock = F

Test+Set(F){

rv = F

lock = T

return rv;

}

P_1 : lock = T

P_0: ✓ ✓

P_1: X X

Compare And Swap Instruction

- Shared boolean variable **lock**., initialized to 0.
- Solution:

```
do {
   while ( compareAndswap(&lock, 0, 1 ));
      // critical section
   lock = 0;
      // remainder section
} while ( true);
```

- Definition:

```
int compareAndswap(int *v, int e int n){
    int temp = *value;
     if(*value == expected)
          *value = new_value;
    return temp;
}
```

Complete Solution using Test And Set

```
do{
waiting[i] = true;
key = true;
while(waiting[i] && key)
    key=testAndset(&lock);
waiting[i] = false;
        // critical section
j = (i  + 1) % n;
while((j!=i) && !waiting[j])
    j = (j+1)%n;
if (j==i)
    lock = false;
else
    waiting[j] = false;
        // remainder section
}while(true);
```

```
Boolean waiting[n];
Boolean lock;
```

False

n = 2

P_0, P_1

	i	waiting	lock	key	j
P_0	0	FTF	FTF	TF	1
P_1	1	FTTF	FTF	TT	0 1

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

Mutex Locks

- Stands for **Mutual Exclusion**.
- One of the simplest software tools for solving the critical section problem.
- Used to protect critical sections and prevent race condition.
- A process acquires the lock before entering the critical section and releases the lock after completing its critical section.
- A boolean variable **available** indicates if the lock is available or not.
- Two primitives acquire() and release() for acquiring and releasing locks, respectively are defined as follows.

```
acquire(){
    while(!available); // busy wait
    available = false;
}
```

```
release(){
    available =true;
}
```

Mutex Locks

- Calls to either acquire() or release() must be done atomically.
- Main disadvantage is busy waiting.
- When one process is in its critical section, others must loop continuously in the call to acquire().
- This type of mutex lock is also known as spin lock as the process literally spins while waiting for the lock to become available.
  - Spin lock is a problem in real multiprogramming situation, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other processes might be able to use productively.
  - One advantage of spin lock is that, no context switching is required when a process must wait on a lock. Thus, when locks are expected to be held for a short time, spin locks are useful.
  - They are often employed on multi-processor systems where one thread can spin on one processor while another thread performs its critical section on another processor.

Semaphore

- Hardware based solution to the "CS" problem are complicated for application programmer to use. To overcome this difficulty we can use a **synchronization tool** called a semaphore.

- A **semaphore**, in its most basic form, is a protected integer variable that can facilitate and restrict access to shared resources in a multi-processing environment.
- Semaphore *S* – integer variable ; can not modify directly.
- Using two standard operations we can modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated

Semaphore

- No other process can access the semaphore when P or V are executing.
- This is implemented with **atomic** hardware and code. An atomic operation is indivisible, that is, it can be considered to execute as a unit.

- S. wait ( ) //Wait until semaphore S is available

<Critical Section>

S. signal ( ) // Signal to other process that semaphore S is free

Semaphore Operations Meaning

while (S<=0); // no-op S++;
S--;

Wait (S) :

- If S is positive(S>0), decrement the value by 1 and return.
- If S is not positive, it just put the calling process into no-operation.

Signal (S) :

- If there is one or more process waiting on S, Then one process is selected and waken up, and signal returns.
- If there is no process waiting then S is simply incremented by 1 and signal returns.

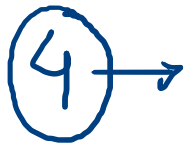
Semaphore

```
S. wait ( ) {
      while (S <= 0) ;    // no-op
         S--;
 }
S. signal ( ) {
   S++;
}
```

- If a semaphore executes S. wait( ) and semaphore S is free (non-zero), it continues executing. Is semaphore S is not free, the OS puts the process on the wait queue for semaphore S.
- A S. signal( ) unblock / wake up one process on semaphore S's wait on a queue.

Semaphore as General Synchronization Tool

- Two types of semaphore :
  - Binary semaphore – If there is a single resource(CS) only , integer value can range only between 0 and 1; can be simpler to implement
    - Also known as mutex locks
  - Binary semaphore initialized to 1
  - Provides mutual exclusion
  - Counting semaphore – If there are more than one but limited resources, integer value can range over an unrestricted domain



Mutual exclusion with semaphores

- Binary Semaphores (mutex) can be used to solve CS problem.
- A semaphore variable (say mutex) can be shared by n processes and initialized to 1.
- Each process is structured as follows :

```
do{
        waiting(mutex);
                //critical section
        signal(mutex);
                //remainder section
}while (TRUE);
```

Counting semaphore

- **wait()** is called when a process wants access to a resource.
- If there is a semaphore is greater than zero, then it can take that resource.
- If the semaphore is zero, that process must wait until it becomes available.
- **signal()** is called when a process is done using a resource.

```
wait(s){
    while (s<=0);    /* wait until s>0 */
    s--;
}

signal(s){
    s++;
}

Init(Semaphore s , Int v){
    s=v;
}
```

Semaphore Usage

- Consider two concurrently running processes
  - P_1 with statement S_1
  - P_2 with statement S_2
- Suppose we require S_2 to be executed only when S_1 has finished execution.
- To do this, we allow P_1 and P_2 to share a semaphore synch initialized to 0 and the processes P_1 and P_2 are defined as follows.

```
P1(){
    S1;
    signal(synch);
}
```

```
P2(){
    wait(synch);
    S2;
}
```

- Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after statement S_1 has been executed.

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

Self-Study

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

- WILL BE GIVEN AS A GROUP WORK MATERIAL.

End of Chapter 5