Process Synchronization – Part I

CS 1550 – Introduction to Operating Systems

Prof. Taieb Znati
Department Computer Science
University of Pittsbrugh

InterProcess Synchronization (IPC)

- Cooperating processes share data in different ways
  - By directly sharing a logical address space – code and data
  - By sharing data through files and messages
- Situations where the correctness of the computation performed by cooperating processes can be affected by the **relative timing** of the processes' execution is called a **race condition**
  - A process can be interrupted at any point in its instruction stream, and only partially completes its execution, resulting in data inconsistency of shared data
    - The outcome of the computation depends on the particular order in which the access to shared data takes place.
- To be considered correct, cooperating processes may **NOT** be subject to race conditions

Race Condition – Bounded Buffer

Producer

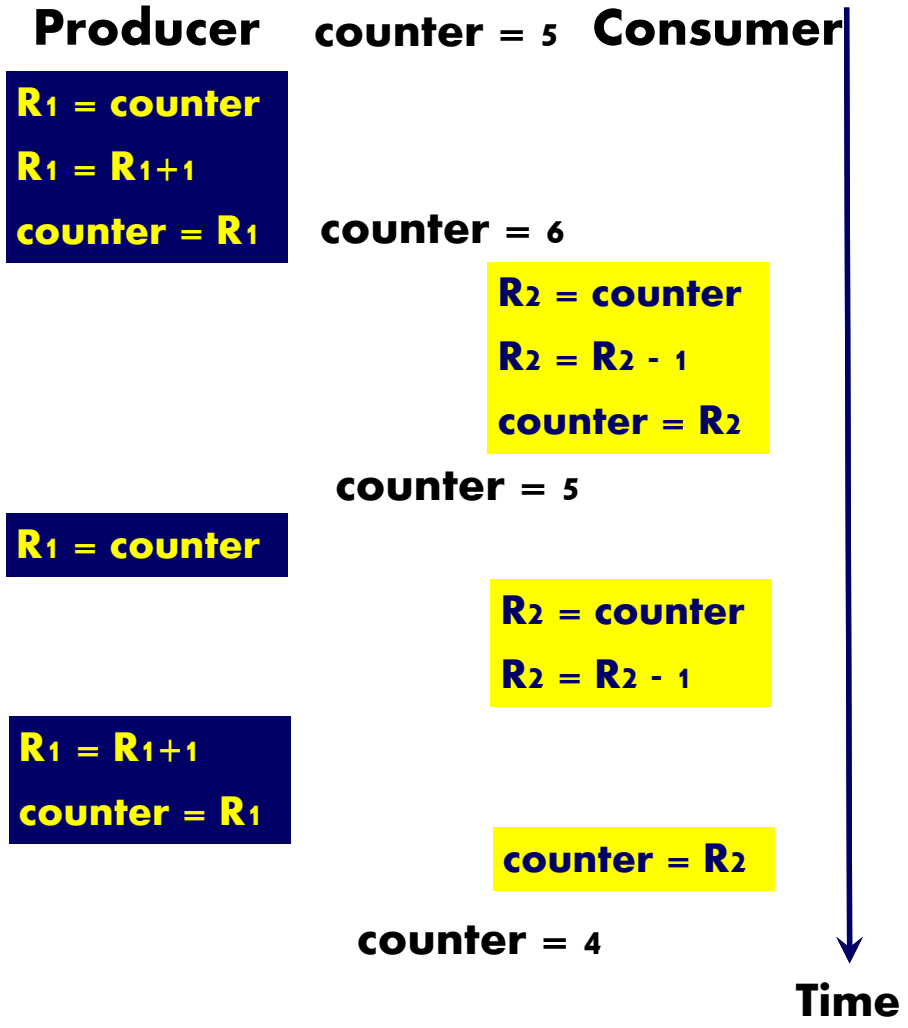
```
While (true)
{
    next_item = produce();
    while (counter == BSIZE);
  /*Buffer Full – Do Nothing*/
    buffer[in] = next_item;
    in = (in + 1) % BUFFSIZE;
    counter++;
}
```

Consumer

```
While (true)
{
   while (counter == 0);
   /*Buffer Empty – Do Nothing*/
   next_item = buffer[out];
   out = (out + 1) % BUFFSIZE;
   counter--;
   consume(next_item);
}
```

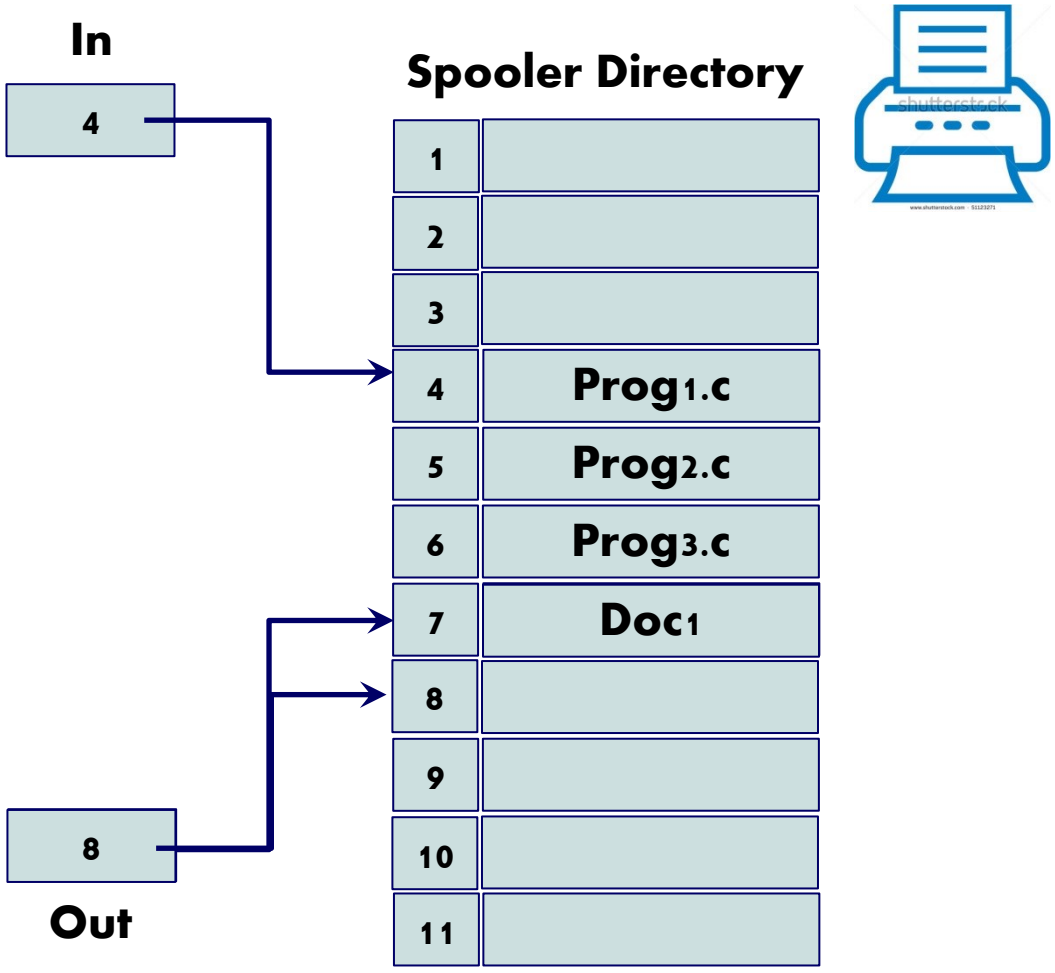
Race Condition – Inconsistent Data

- The "**counter**" increment and decrement can be implemented as:
  - **R = counter;**
  - **R = R ± 1;**
  - **counter = R;**



Race Condition – Printer Daemon

1. **Proc1 reads Out**
   - **Out = 7**
2. **Proc2 reads Out**
   - **Out = 7**
3. **Proc2 stores Doc2**
   - **Out++**
   - **Store Out**
4. **Proc1 stores Doc1**
   - **Out++**
   - **Store Out**



Critical Section Problem

- Race condition occurs frequently in OS, as different parts of the system manipulate resources
  - Growing importance of multicore architectures has increased emphasis on developing multithreaded applications
  - Multiple threads, running in parallel on different cores, can be sharing data – potential for race condition
- To guard against race conditions, the OS must ensure that only one process at a time can be manipulating a shared variable
  - Process synchronization is required

Critical Section Problem – General Process Structure

Critical Section Problem

- **P** = {P_1, P_2, …, P_N}, a set of processes
  - Each process has a **critical section**, in which it can manipulate common variable
- The critical section problem is to design a protocol the processes can use to cooperate, while guaranteeing data consistency

Process General Structure

Do {

Initial Section

No Access to Shared Variables

Entry Section

Critical Section

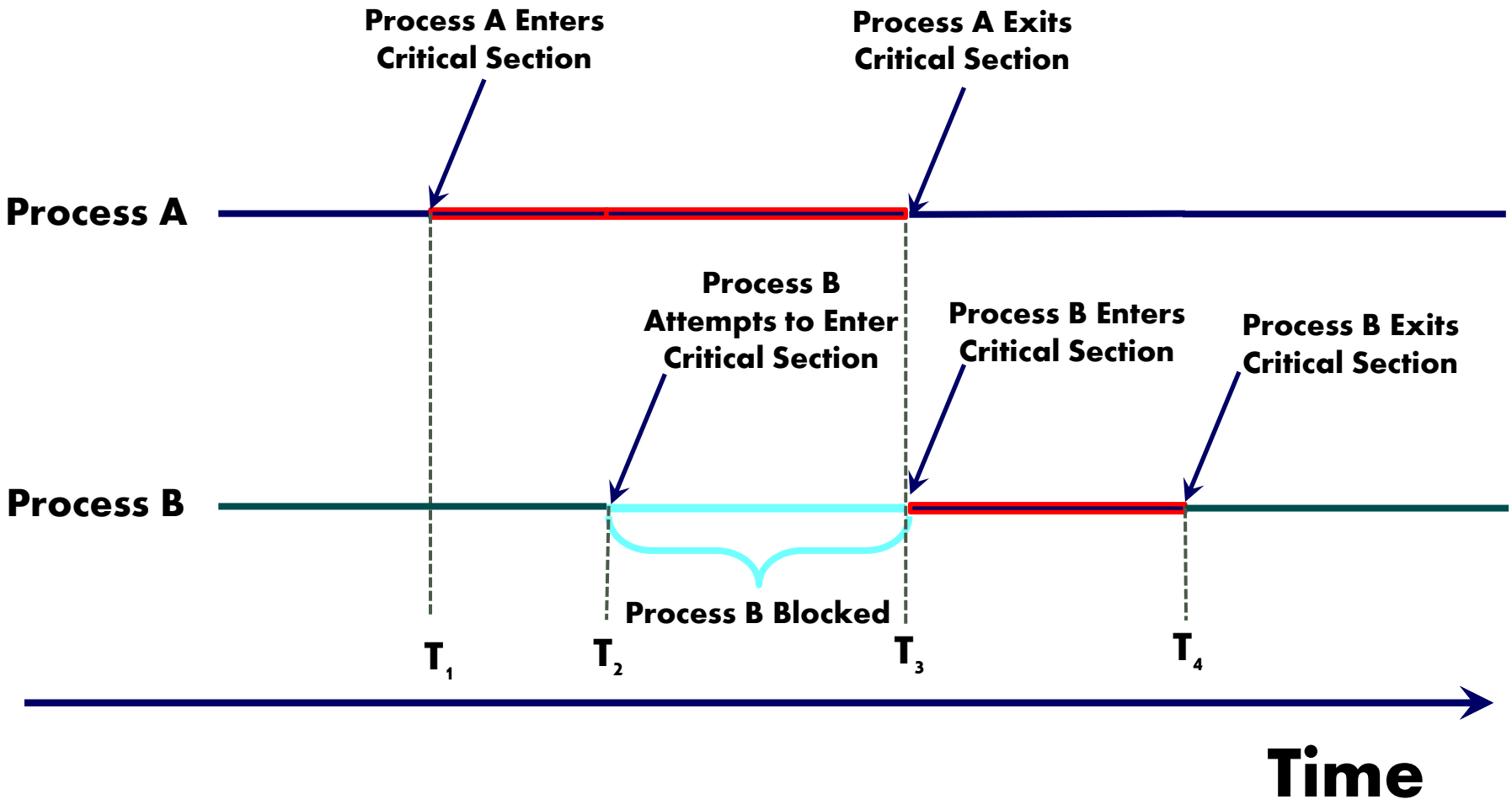
Exit Section

Remainder Section

No Access to Shared Variables

} **while (true)**

Critical Section – Execution



Critical-Section Conditions

- Mechanisms to control access to critical sections will be assumed to operate under the following conditions
  - **Relative Speed** : No assumptions may be made about the relative execution speeds of the cooperating processes except execution of each process proceeds at some nonzero speed.
    - No assumptions about the number of processors in the systems
  - **No Indivisibility**: No assumptions may be made about the indivisibility of machine instructions except that which is induced as part of the definition of a synchronizing primitives
    - Instructions may broken up into a number of atomic actions
    - Execution of other processes may be interleaved among those atomic actions
  - **Bounded Use**: A process only executes in its critical section for some finite time and may not terminate while in its critical section

Critical-Section Solution – Requirements

- A CS solution must satisfy the following:
  - **Mutual Exclusion:** No two processes can be in their critical section simultaneously, accessing shared resources
  - **Progress**: When no process is in its CS, and one or more processes are waiting to enter their CS, decision as to which one enters its CS involves only those processes and must be made within a finite amount of time
    - A process operating outside of its CS cannot prevent another process from entering its CS
  - **Bounded Waiting:** When a process requests access to ac critical section, a decision that grants it access may not be delayed indefinitely
    - No one process can repeatedly enter its CS while other processes never get a chance to enter theirs
    - A process cannot be denied access to its CS, because of starvation or deadlock

CS Solution – Interrupt Disabling

- On a single-CPU system, mutual exclusion can be achieved using OS primitives to disable and enable interrupts
  - Concurrency is achieved by interleaving execution of ready processes – CPU switch from process to process occurs as a result of **clock** or other **interrupts**
- Critical Section Access

DisableInterrupt()

Critical Section

EnableInterrupt()

Limitation of Disabling Interrupts

- It only works in a single-processor environment
  - Disabling interrupts only affects the CPU that executes the ***InterrruptDisable()*** primitive – other processes running on different processors can still access the shared memory
  - Disabling interrupts in all CPUs is not feasible – delay required is prohibitive and may lead to a significant performance degradation
- Interrupts can be lost if not serviced promptly
  - When interrupts are disabled, a process executing in its CS for any longer than a short time would disrupt the proper execution of I/O operations of other processes
- It is risky to grant user processes the power to disable interrupts – misbehavior leads to undesirable outcome
    - A process waiting to enter its CS could be starved
    - Exclusive access to CPU inhibits scheduling, for example

Mutual Exclusion

SOFTWARE SOLUTION

Strict Alternation – Version 1

Shared Variables – int turn; initially turn = 0

Turn == i ⇒ P_i ready to enter its critical section

Process P_0

```
do {
while (turn != 0) ; /*Spin*/

    Critical Section

turn = 1;
  Reminder Section
} while (1);
```

Process P_1

```
do {
while (turn != 1) ; /*Spin*/

    Critical Section

turn = 0;
  Reminder Section
} while (1);
```

Satisfies Mutual Exclusion, but NO Progress

Strict Alternation – Version 2

Shared variables – bool flag[2]; initially flag [0] = flag [1] = false.
flag [i] = true ⇒ P_i ready to enter its critical section

Process P_0

```
do {
 flag[0] := true;
 while (flag[1]) ;        /*Spin*/

   Critical Section

 flag [0] = false;
 Reminder Section
} while (1);
```

Process P_1

```
do {
 flag[1] := true;
 while (flag[0]) ;        /*Spin*/

   Critical Section

 flag [1] = false;
 Reminder Section
} while (1);
```

Satisfies Mutual Exclusion, but NO Progress

Peterson's Algorithm

Shared variables – bool flag[2]; int turn.

flag[i] == true and turn == i ⇒ P_i ready to enter its critical section

Process P_0

```
do {
 flag[0] := true; turn = 1;
 while (turn == 1 && flag[1]);
```

Critical Section

```
 flag [0] = false;
 Reminder Section
} while (1);
```

Process P_1

```
do {
 flag[1] := true; turn = 0;
 while (turn == 0 && flag[0]);
```

Critical Section

```
 flag [1] = false;
 Reminder Section
} while (1);
```

Satisfies Mutual Exclusion and Progress

Mutual Exclusion

HARDWARE SYNCHRONIZATION

ATOMIC INSTRUCTION

Synchronization Hardware Instruction

- Software-based solutions are not guaranteed to work on modern multi-core architectures
- Other solutions to the CS problem rely on techniques ranging from hardware to software-based API
  - The premise of these solutions is **locking**, whereby access to critical sections is protected through the use of **locks**
    - They differ on their design and issue of the locks
  - Hardware instructions that support atomic operations
    - Test-and-Set() and Compare-and-Swap()
    - Mutex Locks
    - Semphores

Test-and_Set() Instruction Atomic Operation

- **Test_and_Set** is a special hardware instruction to test and modify the content of a word, atomically

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

- Mutual exclusion

```
do{ while (test_and_set)(&lock);
        Critical Section
  lock = false;} while (true)
```

Test_and_Set()

Enter_Region

```
TSL Register, Lock
CMP Register, #0
JNE Enter_Region
RET()
```

Critical Section

Exit_Region

```
MOVE Lock, #0
RET()
```

Compare_and_Swap() Instruction Atomic Operation

- Compare_and_Swap() operates on three operands
  - It exchange the contents of two locations atomically

```
int Compare_and_Swap(int *vl, int xpctd, int   n_vl){
 int tmp =*vl;
  if (*vl == xpctd)
     *vl == n_val;
  return(tmp); }
```

- Mutual exclusion

```
Do{while (Compare_and_Swap(&lock, 0, 1) !=0) ;
```

Critical Section

```
 lock = 0;
} while (true);
```

Compare_and_Swap()

Enter_Region

```
MOVE Register, #1
XCHG Register, Lock
CMP Register, #0
JNE Enter_Region
RET()
```

Critical Section

Exit_Region

```
MOVE Lock, #0
RET()
```

Limitations of Hardware-Based Solutions

- Mutual exclusion is preserved
  - If process P_i enter CS, other processes, P_j, are busy waiting
  - The "generic" solutions, however, do not satisfy the bounded-waiting requirement
    - Selection of which process enters next the CS, after the current process leaves its CS, is arbitrary
    - Starvation is possible
  - Solutions that can satisfy bounded-waiting requirements are possible using additional data structure
- Busy waiting is an undesirable property of the solution

Mutex Locks

- Simplest and most efficient thread synchronization mechanism
  - Used to protocol critical sections and prevent race conditions
  - A process must acquire the lock, prior to entering its CS
  - A process releases the lock upon exiting its CS

```
Do{
  acquire (lock)

    Critical Section

  release(lock)
}while (true);
```

```
acquire (lock) {
  while (!available);
   available = false;}

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

Mutex Locks Semantics

- A mutex lock has a boolean variable whose value indicates one of two possible states of the lock
  - **Locked**: the lock is not available
    - An attempt to acquire() the lock fails
      - The process is blocked until the lock is released
  - **Unlocked**: the lock is available
    - An attempt to acquire() the lock succeeds
    - The lock becomes unavailable – locked

Mutex Locks Limitation

- A mutex lock requires busy waiting
  - A process waiting for the lock to become available loops continuously in the acquire() call
    - Also referred to as **spinlock** – process "spins" as it is waiting
  - Busy waiting wastes CPU cycles and prevent other processes from doing useful work
- Spinlock has the advantage that no context switch is required when a process waits on a lock
  - Context switching takes considerable amount of time
- Spinlocks are useful if the locks are held only for short period of times
- Spinlocks are often used in multiprocessor environements
  - One thread spins on one processor, while the other executes its CS on another processor

Semaphores

- Semaphores are a more robust synchronization tool than mutex locks
- A semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 atomic and mutually exclusive operations:
  - **wait(S)** – originally termed P() operation
  - **signal(S)** – originally termed V() operation

Wait and Signal Operation

Wait Operation

```
wait(S) {
   while (S<=0);
   S--;
}
```

Signal Operation

```
signal(S) {
   S++;
}
```

- All modifications to the integer, S, in wait() and signal() must be executed **indivisibly**
  - **When one process modifies S, no other process can simultaneously modify S**
  - The testing of S in wait() must be executed with no interruption

Critical Section of n Processes

- Shared data:

  semaphore mutex; //initially mutex = 1

- Process Pi:

```
do {
    wait(mutex);
        critical section
    signal(mutex);
    remainder section
} while (1);
```

Semaphore Usage

- Two Types of Semaphores
  - Counting semaphore – the integer value can range over an unrestricted domain.
  - Counting semaphores can be used to control access to resource with a finite number of instances
    - The wait() operation decrements the counter
    - A signal() operation increments the counter
      - When the counter reaches 0, all resources are being used
  - Binary semaphore – the integer value can range only between 0 and 1
    - Binary semaphores behave similarly to mutex locsk
    - They can be simpler to implement.

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**.

Implementation

- Semaphore operations now defined as

wait(S):

```
S.value--;
if (S.value < 0) {
        add this process to S.L;
        block;
}
```

signal(S):

```
S.value++;
if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
}
```

Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

Implementing *S* as a Binary Semaphore

- Data structures:

  binary-semaphore S1, S2;

  int C:

- Initialization:

  S1 = 1

  S2 = 0

  C = initial value of semaphore **S**

Implementing *S*

- **Wait Operation**

```
wait(S1);
C--;
if (C < 0) {
        signal(S1);
        wait(S2);
}
signal(S1);
```

- **Signal Operation**

```
wait(S1);
C ++;
if (C <= 0)
    signal(S2);
else
    signal(S1);
```

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

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

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

Priority Inversion Problem

- Unexpected consequence of busy wait
- Given H (a high priority job) and L (low priority job)
- Scheduling Rule – Whenever H is ready to run, L is preempted and H is run
- H executes and I/O and blocks
- At a certain time, t, H becomes ready to run, while L is in its critical section
  - H begins busy waiting
  - Since L is never scheduled when H is running, L never gets a chance to leave its critical section, so H loops forever

Priority Inversion Problem

- H executes…
- H blocks on I/O
- I/O completes
- H runs…
- H attempts to enter C.S.
- H busy waits forever
- L is ready to run
- L runs…
- L enters C.S.…
- …
- L is preempted

Using mutex (provided by OS)

- Simpler than semaphore
- Two states: locked or unlocked
- Functions:
  - Declare mutex variable
  - Initialize mutex variable (just once)
  - Lock, C.S., unlock

Conclusion

- Process Concept, Management and Operation
  - Process state, PCB, and Context
    - Context switching
  - Process scheduling
  - Process Operations
- Cooperating Processes
  - Interprocess Communications
  - Sockets, RPC, and pipes