Inter-Process Communications

Using Tanenbaum's
Modern Operating Systems (3rd edition)

Inter-Process Communications (IPC)

- IPC is related to three issues:
  - Make sure two (or more) processes do not get in each other's way (grab the last seat on a plane…)
  - Proper sequencing
    - spinlock, semaphore, mutex
  - Pass information between processes
    - Signal, pipe, message queue, socket, shared memory, etc …
- Same also for threads

Locking & Synchronization

Race Conditions (I)

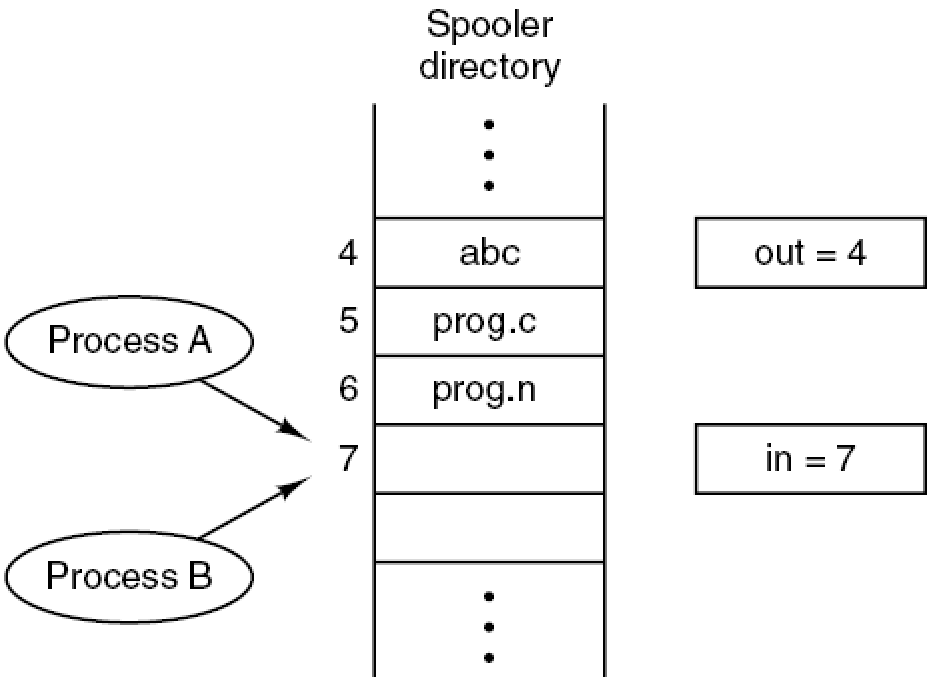


Figure 2-21. Two processes want to access shared memory at the same time

Race Conditions (II)

- R/W access to a shared resource
- **Race condition**:
  - The final result of a number of processes that run on shared resource, depends on who runs when

- **Critical section**:
  - The part of the program where the shared resource is modified

Critical Section

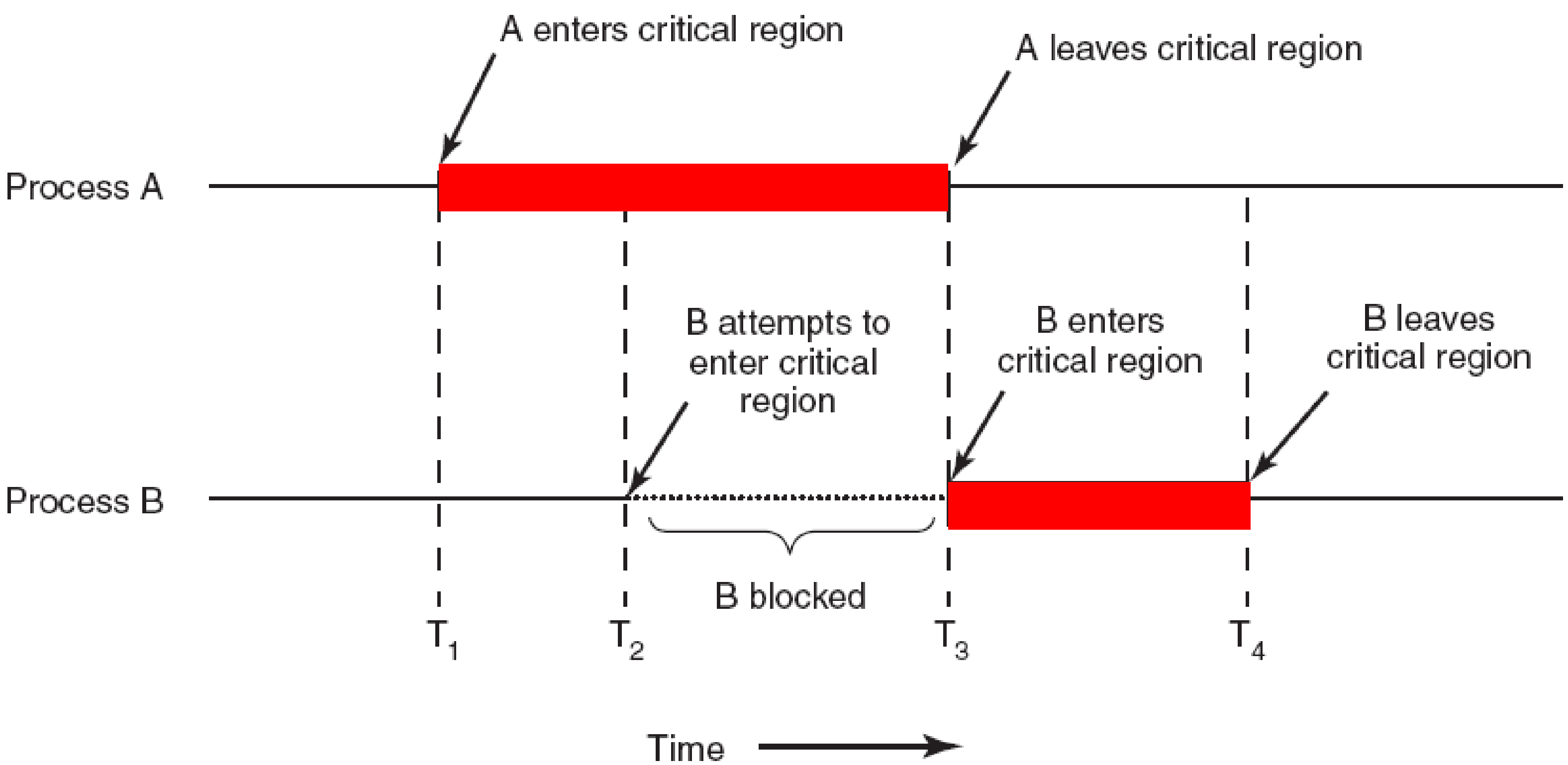


Figure 2-22. Mutual exclusion using critical regions

Mutual Exclusion – Definition

- Mutual Exclusion
  - Make sure that only one process is within the critical section
- Good solution should satisfy 4 conditions:
  1. Only one process may be in its critical section at a time
  2. No assumptions may be made about the speed and number of CPU's
  3. Process may block other processes only inside the critical section
  4. No processes should have to wait forever to enter its critical region

Mutual Exclusion – Solutions (I)

- Disable interrupts
  - Generally unattractive for user processes (Why?)
  - Often useful within the OS itself
- Lock variables
  - A single shared (lock) variable – if the lock is 0, set it to 1 and enter Otherwise, just wait until it becomes 0.
  - Contains the same race-condition problem (How?)
- Spin lock
  - Each process is busy waiting till its own turn
  - Busy waiting – a waste of CPU time
  - Taking turns is not a good idea when one process is much slower than the other. (Why?).

Spin Lock (I)

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

(a)

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

(b)

Figure 2-23. A proposed solution to the critical region problem
(a) Process 0 (b) Process 1
In both cases, be sure to note the semicolons terminating the while statements

Spin lock (II)

- Spin lock is used when:
  a. Critical section is very short – low chance of a resource conflict
  b. The wait time is smaller than the context switch time
  c. No OS

Mutual Exclusion – Solutions (II)

- TSL instruction
  - Reads the contents of the lock and writes to it in one single machine instruction. The memory bus is locked until TSL command is finished
  - Busy waiting defect

TSL – Test & Set Instruction

```
enter_region:
        TSL REGISTER,LOCK        | copy lock to register and set lock to 1
        CMP REGISTER,#0          | was lock zero?
        JNE enter_region         | if it was nonzero, lock was set, so loop
        RET                      | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0             | store a 0 in lock
        RET                      | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction

Semaphore & Mutex

Producer-Consumer Problem

- Two processes share a common, fixed-sized buffer:
  - The producer puts information into the buffer
  - The consumer takes it out

- Can a race-condition occur?
  (Clue: How can a wakeup be lost?)

Producer-Consumer (II)

```
#define N 100                                    /* number of slots in the buffer */
int count = 0;                                   /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                               /* repeat forever */
        item = produce_item( );                  /* generate next item */
        if (count == N) sleep( );                /* if buffer is full, go to sleep */
        insert_item(item);                       /* put item in buffer */
        count = count + 1;                       /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);        /* was buffer empty? */
    }
}
void consumer(void)
{
    int item;

    while (TRUE) {                               /* repeat forever */
        if (count == 0) sleep( );                /* if buffer is empty, got to sleep */
        item = remove_item( );                   /* take item out of buffer */
        count = count – 1;                       /* decrement count of items in buffer */
        if (count == N – 1) wakeup(producer);    /* was buffer full? */
        consume_item(item);                      /* print item */
    }
}
```

Semaphore – Dijkstra ‘65

- Semaphore – counts the number of wake-ups (resources)
- Semaphore implementation
  - Checking the value, changing it, and going sleep – are all an atomic action
  - No race condition. No busy waiting.
  - Stored in the kernel and accessed via system calls
- Binary Semaphore:
  - Initialized to 1.
  - Used by two or more processes for mutual exclusion (down & up).
- General Semaphore:
  - Initialized to value different than 1
  - Used for synchronization.

Semaphore – Basic Functionality

```
/* Each operation is atomic */
void down(Semaphore s)
{
   if (s>0)
   {
      --s;
   }
   else
   {
      Place the process on the semaphore Q and move it to
      the block state;
   }
}

void up(Semaphore s)
{
   if (There is at least one process sleeping in the Q)
   {
      wake it up.
   }
   else
   {
       ++s;
   }
}
```

Producer-Consumer Using Semaphore

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}
```

```
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Exercise #1: Producer-Consumer

- Solve the producer-consumer problem using
  - Two Counting Semaphores
  - One Mutual Exclusion Lock

Mutex

- Can be in one of two states: Locked and Unlocked
- Used only for mutual exclusion; no busy waiting
- Only owner of a lock can (should) perform unlock
- Locked mutex cannot be locked again – no recursion

- Mutex is a non-recursive binary semaphore with ownership

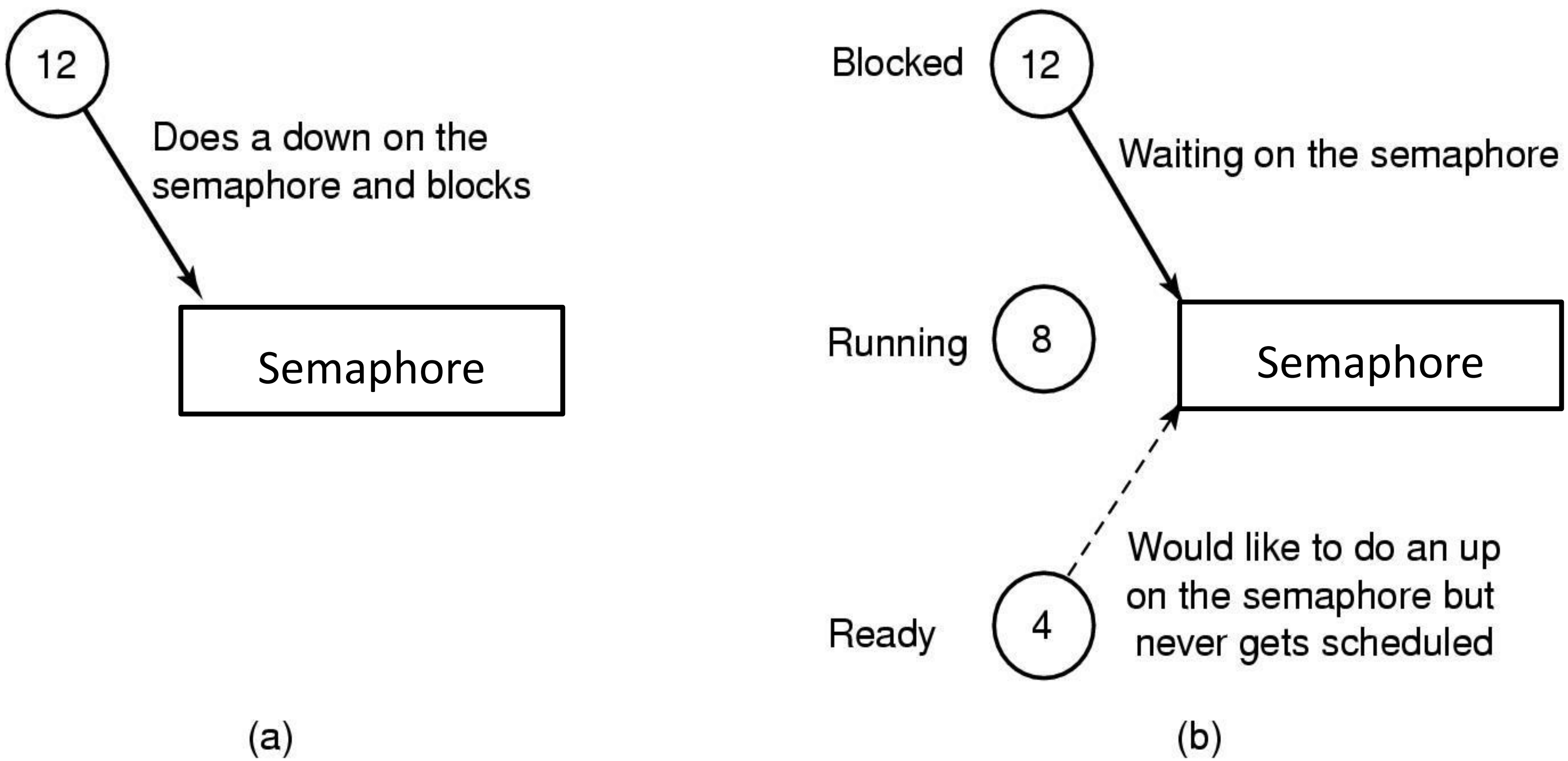
Mutex Implementation

```
mutex_lock:
        TSL REGISTER,MUTEX         | copy mutex to register and set mutex to 1
        CMP REGISTER,#0            | was mutex zero?
        JZE ok                     | if it was zero, mutex was unlocked, so return
        CALL thread_yield          | mutex is busy; schedule another thread
        JMP mutex_lock             | try again
ok:     RET                        | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0              | store a 0 in mutex
        RET                        | return to caller
```

Figure 2-29. Implementation of mutex lock and mutex unlock.

Priority Inversion (I)



Priority Inversion (II)

- Problem:
  - Low priority task owns a resource
  - High priority task is blocked waiting for the resource
  - Intermediate priority tasks keep preempting the low priority task
  - Hence, no progress towards releasing the resource
- Solution:
  - Priority inheritance
  - The lower-priority task inherits the priority of any higher-priority task pending on a resource they share

Homework + Interview Questions

- What is *critical section*?
- What is *mutual exclusion*?
- What is *spin lock*? When should/may be used?
- Describe how *semaphore* works
- Describe how *mutex* works?
- What is the difference between a *mutex* and a binary *semaphore*
- What is *Priority Inversion*? Propose a solution