OPERATING SYSTEMS

TANENBAUM and STALLINGS

Modified by S. Pearce

Processes and Threads 5
Process Cooperation

Summary of “Processes and Threads 4”

- Summary of requisites associated with solving the critical section problem:

Summary of “Processes and Threads 4”

- Summary of requisites associated with solving the critical section problem:
- Review of basic two-thread solutions.

Summary of “Processes and Threads 4”

- Summary of requisites associated with solving the critical section problem:
- Review of basic two-thread solutions.
- Provide such solutions taxonomically (H/W, S/W, with and without Busy-Wait, *etc.*)

Overview of “Processes and Threads 5”

- System Call methodology.

Overview of “Processes and Threads 5”

- System Call methodology.
- In this unit, we are interested in enforcing mutual exclusion through the ***operating system*** itself:
  - Semaphores, and
  - Message passing.

Overview of “Processes and Threads 5”

- System Call methodology.
- In this unit, we are interested in enforcing mutual exclusion through the operating system itself:
  - Semaphores, and
  - Message passing.
- Textbook topics include:
  - Producer-Consumer problem.
  - **Semaphores**
  - Mutex (a specific semaphore)
  - Pthreads (POSIX standards)
  - Monitors
  - **Message Passing**
  - Barriers

Overview

- Up to now, we have been looking at solving the mutual exclusion problem for two threads.
  - H/W Solutions: Disabling Interrupts
  - S/W Solutions: Lock Variable, Strict Alternation, Peterson's Solution

Overview

- Up to now, we have been looking at solving the mutual exclusion problem for two threads.
  - H/W Solutions: Disabling Interrupts
  - S/W Solutions: Lock Variable, Strict Alternation, Peterson's Solution
- One of the simplest solutions to the mutual exclusion problem for *N* threads is know as the Bakery Algorithm (not in text)
  - For a system that provides read-write atomicity only.
  - The basic idea is that each non-thinking process has a variable that indicates the position of that process in a hypothetical queue of all the non-thinking processes.
  - Each process in this queue scans the variables of the other processes, and enters the critical section only upon determining that it is at the head of the queue (DIFFICULT TO UNDERSTAND).

Sleep-Wakeup

- Using System Calls:

Sleep-Wakeup

- Using System Calls:
  - Much of what we have looked at involves **busy waiting** which wastes CPU time.

Sleep-Wakeup

- Using System Calls:
  - Much of what we have looked at involves **busy waiting** which wastes CPU time.
  - Interprocess communication **system call** primitives **block**, avoiding this situation.

Sleep-Wakeup

- Using System Calls:
  - Much of what we have looked at involves **busy waiting** which wastes CPU time.
  - Interprocess communication **system call** primitives **block**, avoiding this situation.
  - **Sleep** is a system call that causes the caller to be suspended until some other process wakes it up.

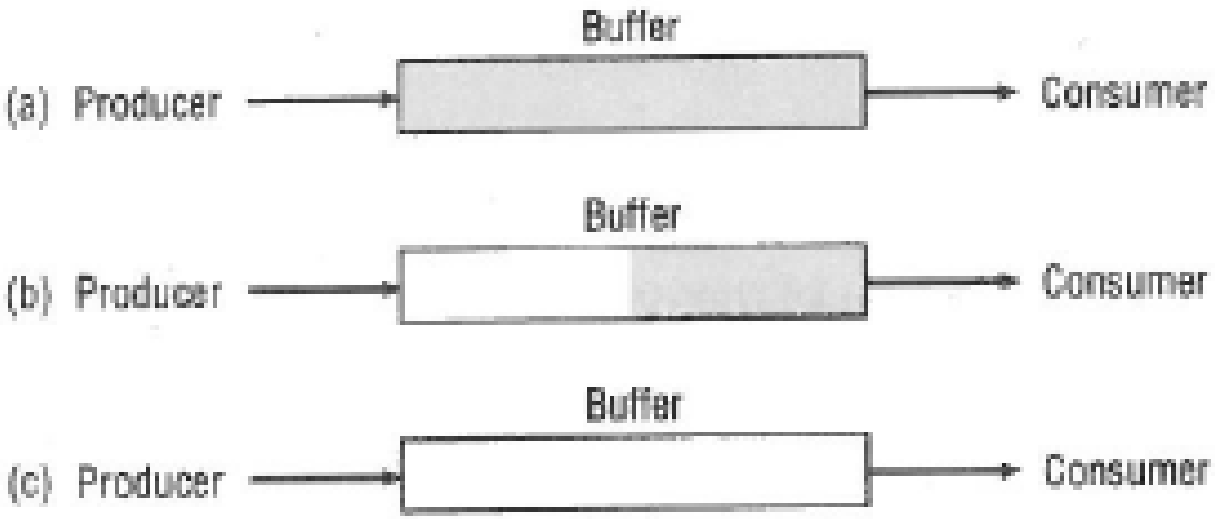
Sleep-Wakeup

- Using System Calls:
  - Much of what we have looked at involves **busy waiting** which wastes CPU time.
  - Interprocess communication **system call** primitives **block**, avoiding this situation.
  - **Sleep** is a system call that causes the caller to be suspended until some other process wakes it up.
  - **Wakeup** is a system call that wakes up the process.

Sleep-Wakeup

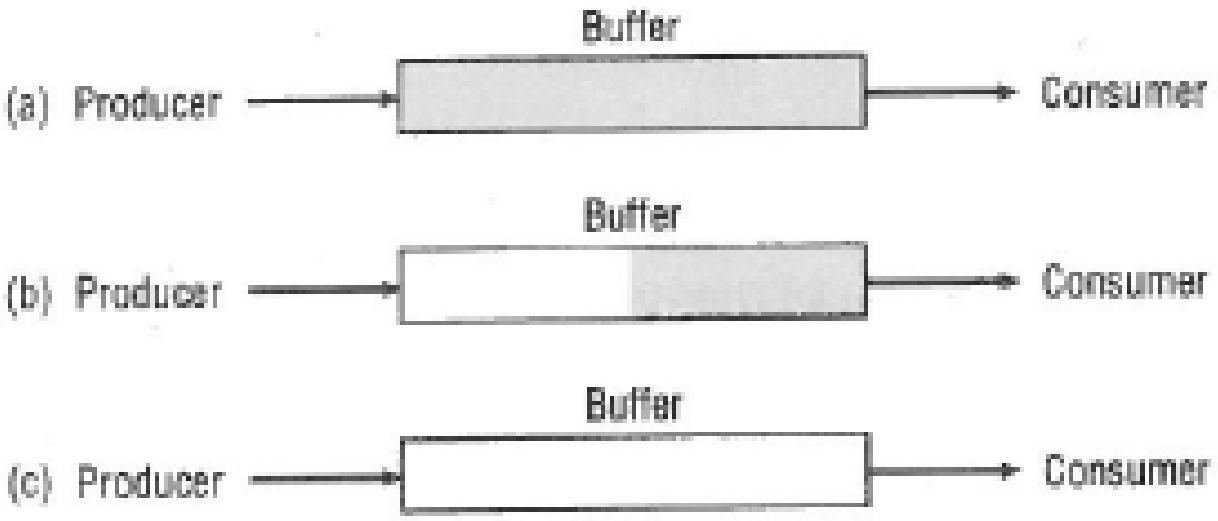
- Using System Calls:
  - Much of what we have looked at involves **busy waiting** which wastes CPU time.
  - Interprocess communication **system call** primitives **block**, avoiding this situation.
  - **Sleep** is a system call that causes the caller to be suspended until some other process wakes it up.
  - **Wakeup** is a system call that wakes up the process.
  - They both have one parameter that represents a memory address used to match up *sleeps* and *wakeups* .

The Producer-Consumer Problem



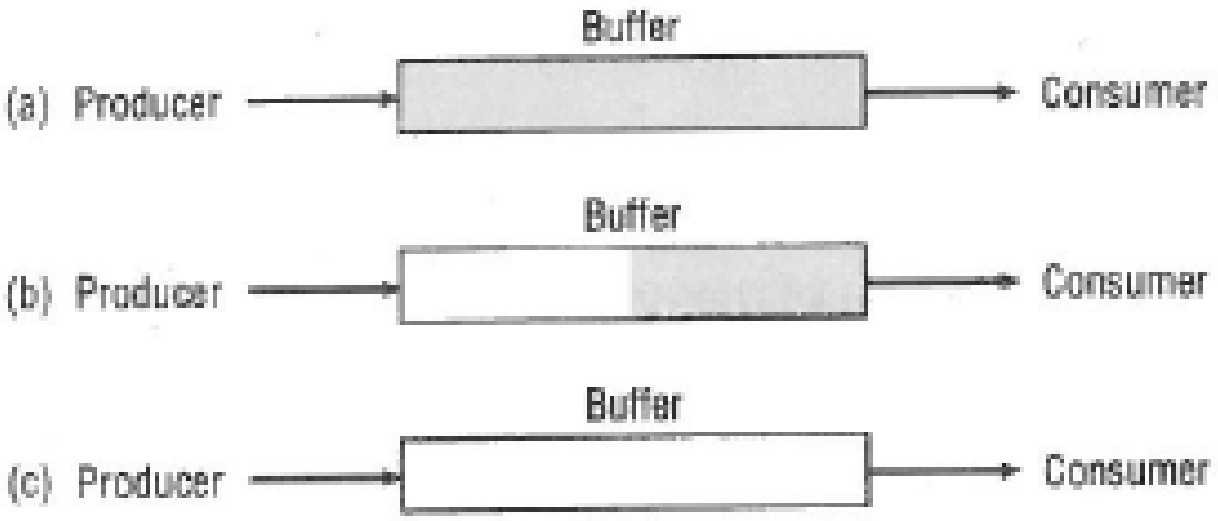
The Producer-Consumer Problem

- The classical example of multi-process synchronization:



The Producer-Consumer Problem

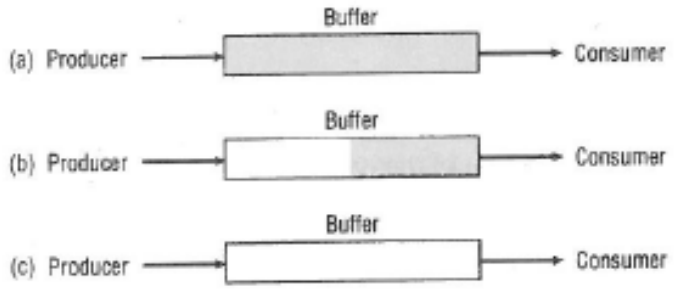
- The classical example of multi-process synchronization:



- Two extreme states can occur; (a) when the producer tries to add to a full buffer, and (c) when the consumer tries to draw from an empty buffer. The case, (b), represents the average situation.

The Producer-Consumer Problem

- Also known as the “bounded-buffer problem”, meaning that both producers and consumers have a fixed number of slots (buffer).



The Producer-Consumer Problem

- Example:
  - Two processes share a common, fixed-size (bounded) buffer.
  - The producer puts information into the buffer and the consumer takes information out.
- **Problem and Solution**:
- ***Problem***: The producer wants to put a new data in the buffer, but buffer is already full.
- ***Solution***: Producer goes to sleep and to be awakened when the consumer has removed data.
- ***Problem***: The consumer wants to remove data the buffer but buffer is already empty,
- ***Solution***: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

The Producer-Consumer Problem

- Conclusion:
  - These approaches also lead to same kind of race conditions that have been encountered in earlier approaches.
  - Race conditions can occur because access to *count* is unconstrained.
  - The problem is that a wakeup call sent to a process that is not sleeping can be lost.

The Producer-Consumer Problem

```
#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 */
     }
}
```

READ

Figure 2-27. The producer-consumer problem with a **fatal race condition**.

The Producer-Consumer Problem

```
#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 */
     }
}
```

Count is unconstrained

READ

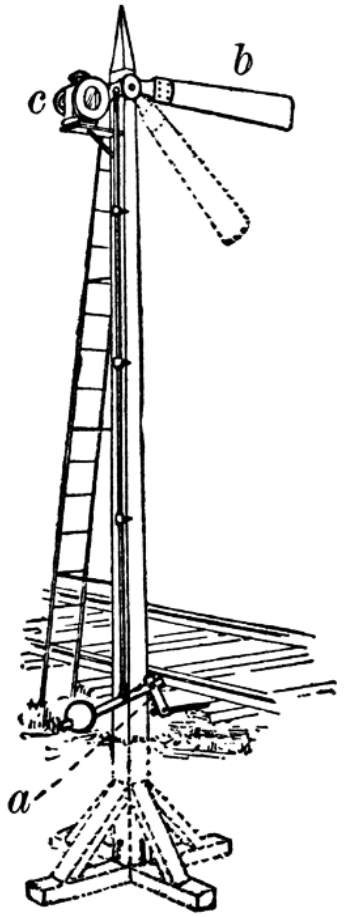
Figure 2-27. The producer-consumer problem with a **fatal race condition**.

The Producer-Consumer Problem

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work. A quick fix is to modify the rules to add a **wakeup waiting bit** to the picture. When a wakeup is sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake. The wakeup waiting bit is a piggy bank for storing wakeup signals. The consumer clears the wakeup waiting bit in every iteration of the loop.

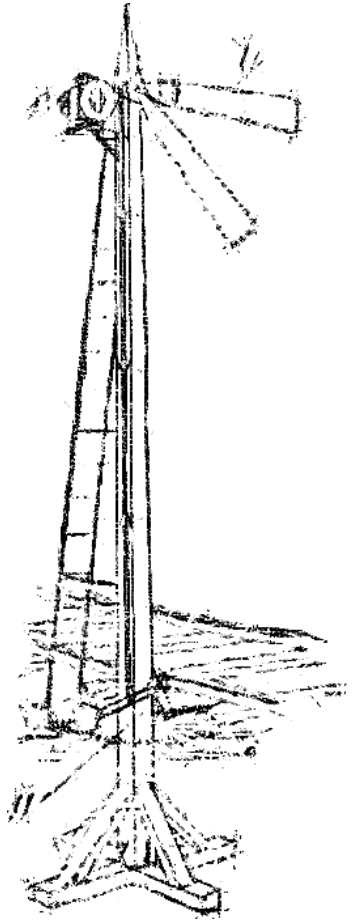
While the wakeup waiting bit saves the day in this simple example, it is easy to construct examples with three or more processes in which one wakeup waiting bit is insufficient. We could make another patch and add a second wakeup waiting bit, or maybe 8 or 32 of them, but in principle the problem is still there.

Semaphores



Semaphores

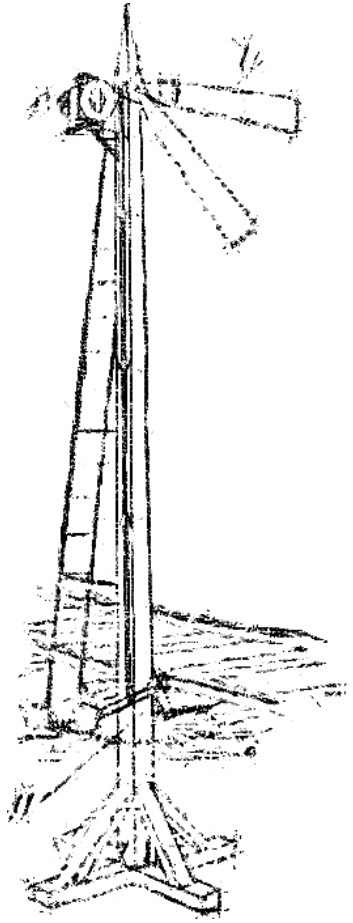
- … a semaphore is a variable or abstract data type that provides a simple but useful abstraction for controlling access by multiple processes to a common resource in a parallel programming environment. (Wikipedia)



Semaphores

- … a semaphore is a variable or abstract data type that provides a simple but useful abstraction for controlling access by multiple processes to a common resource in a parallel programming environment. (Wikipedia)

- "…Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, which he called a **semaphore**, was introduced."



Semaphores

- This was a major advance incorporated into many modern operating systems (Unix, OS/2)
- A semaphore has the following two properties:
  - It is a non-negative integer,
  - It has two *indivisible*, valid operations.

Busy Waiting Semaphores

- The simplest way to implement semaphores.
- Useful when critical sections last for a short time, or we have lots of CPUs.
- S initialized to positive value (to allow someone in at the beginning).

```
wait(S):
while S<=0 do ;
    S--;
```

```
signal(S):
    S++;
```

Semaphore Operations

- Wait(s)

  If s > 0, then s:= s - 1

  else *block this process*

- Signal(s)

  If *there is a blocked process on this semaphore*, then *wake it up*

  else s:= s + 1

Semaphores

- "A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore)."
  - Symbian Developer Library

Semaphores

- "A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore)."
  - Symbian Developer Library
- Like having a number of free identical room keys.
  - Example, say we have four motel rooms with identical locks and keys.
  - The semaphore count - the count of keys - is set to 4 at beginning (all four rooms are free), then the count value is decremented as people are coming in.
  - If all rooms are full the semaphore count is 0.
  - Now, when one person leaves a room, semaphore is increased to 1 (one free key), and given to the next person in the queue.

Semaphores

- "A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore)."
  - Symbian Developer Library
- Like having a number of free identical room keys.
  - Example, say we have four motel rooms with identical locks and keys.
  - The semaphore count - the count of keys - is set to 4 at beginning (all four rooms are free), then the count value is decremented as people are coming in.
  - If all rooms are full the semaphore count is 0.
  - Now, when one person leaves a room, semaphore is increased to 1 (one free key), and given to the next person in the queue.

Semaphores

```
#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 */
     }
}


void consumer(void)
{
     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 */
     }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Semaphores

- Advantages:

Semaphores

- Advantages:
  - Machine independent

Semaphores

- Advantages:
  - Machine independent
  - Simple.

Semaphores

- Advantages:
  - Machine independent
  - Simple.
  - Powerful (embodying both exclusion and waiting).

Semaphores

- Advantages:
  - Machine independent
  - Simple.
  - Powerful (embodying both exclusion and waiting).
  - Correctness is easy to determine.

Semaphores

- Advantages:
  - Machine independent
  - Simple.
  - Powerful (embodying both exclusion and waiting).
  - Correctness is easy to determine.
  - It works with multiple processes.

Semaphores

- Advantages:
  - Machine independent
  - Simple.
  - Powerful (embodying both exclusion and waiting).
  - Correctness is easy to determine.
  - It works with multiple processes.
  - It can be implemented with many different critical sections with different semaphores.

Semaphores

- Advantages:
  - Machine independent
  - Simple.
  - Powerful (embodying both exclusion and waiting).
  - Correctness is easy to determine.
  - It works with multiple processes.
  - It can be implemented with many different critical sections with different semaphores.
  - It can acquire many resources simultaneously.

Semaphores

- Advantages:
  - Machine independent
  - Simple.
  - Powerful (embodying both exclusion and waiting).
  - Correctness is easy to determine.
  - It works with multiple processes.
  - It can be implemented with many different critical sections with different semaphores.
  - It can acquire many resources simultaneously.
  - It can also allow multiple processes into the critical section at once, if that is desirable.

Semaphores

- Advantages:
  - Machine independent
  - Simple.
  - Powerful (embodying both exclusion and waiting).
  - Correctness is easy to determine.
  - It works with multiple processes.
  - It can be implemented with many different critical sections with different semaphores.
  - It can acquire many resources simultaneously.
  - It can also allow multiple processes into the critical section at once, if that is desirable.

- NOTE: Semaphores are like UNIX pipes.

Blocking Semaphores

In practice, *wait* and *signal* are system calls to the OS

- The OS implements the semaphore.
- To avoid busy waiting:
  - when a process has to wait on a semaphore, it will be put in a *blocked queue* of processes waiting for this to happen.

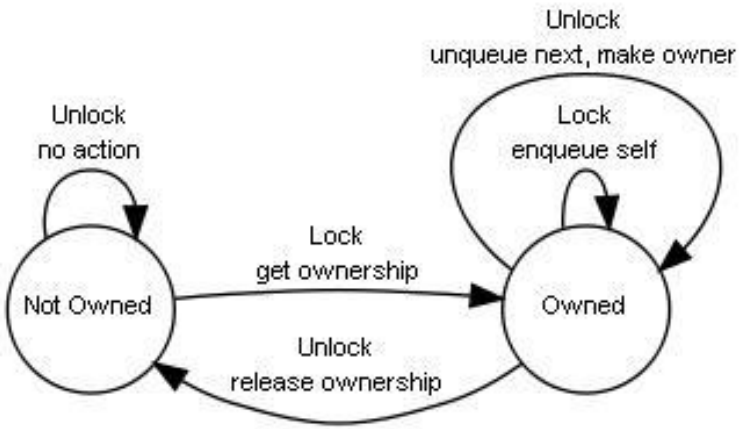
- Queues are normally FIFO. This gives the OS control on the order processes enter CS (Critical Section).
  - There is one queue per semaphore

MUTEX

- “When the semaphore’s ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used.”

MUTEX

- "When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used



Mutex is a synchronization object that can only have two states (owned/not owned), and must include the two operations, lock and unlock.

Using semaphores for solving critical section problems

For n processes

- Initialize semaphore "mutex" to 1
- Then only one process is allowed into CS (mutual exclusion)
- To allow k processes into CS at a time, simply initialize mutex to k

```
Process Pi:
    repeat
        wait(mutex);
            CS
        signal(mutex);
            RS
    forever
```

MUTEX

- Mutual Exclusion Object

MUTEX

- Mutual Exclusion Object
- Provides serial access to a controlled section.

MUTEX

- Mutual Exclusion Object
- Provides serial access to a controlled section.
- Like individual access (room key) to a motel that has only one room and only one person can occupy that room at any time.

MUTEX

- Mutual Exclusion Object
- Provides serial access to a controlled section.
- Like individual access (room key) to a motel that has only one room and only one person can occupy that room at any time.

MUTEX

```
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.

Mutexes in Pthreads (1)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads (2)

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads (3)

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{     int i;

      for (i= 1; i <= MAX; i++) {
            pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
            while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
            buffer = i;                         /* put item in buffer */
            pthread_cond_signal(&condc);        /* wake up consumer */
            pthread_mutex_unlock(&the_mutex);/* release access to buffer */
      }
      pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{     int i;

      for (i = 1; i <= MAX; i++) {
            pthread_mutex_lock(&the_mutex);     /* get exclusive access to buffer */
            while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
            buffer = 0;                         /* take item out of buffer */
            pthread_cond_signal(&condp);        /* wake up producer */
            pthread_mutex_unlock(&the_mutex);/* release access to buffer */
      }
      pthread_exit(0);
}

int main(int argc, char **argv)
{
      pthread_t pro, con;
      pthread_mutex_init(&the_mutex, 0);
      pthread_cond_init(&condc, 0);
      pthread_cond_init(&condp, 0);
      pthread_create(&con, 0, consumer, 0);
      pthread_create(&pro, 0, producer, 0);
      pthread_join(pro, 0);
      pthread_join(con, 0);
      pthread_cond_destroy(&condc);
      pthread_cond_destroy(&condp);
      pthread_mutex_destroy(&the_mutex);
}
```

Figure 2-32. Using threads to solve the producer-consumer problem.

Semaphore Summary

- Semaphores provide a primitive tool that is both flexible and powerful for enforcing mutual exclusion.

Semaphore Summary

- Semaphores provide a primitive tool that is both flexible and powerful for enforcing mutual exclusion.
- **PROBLEM**: It may be necessary to have the *wait* and *signal* operations scattered throughout the code, thereby making it difficult to code accurately.

Semaphore Summary

- Semaphores provide a primitive tool that is both flexible and powerful for enforcing mutual exclusion.
- **PROBLEM**: It may be necessary to have the *wait* and *signal* operations scattered throughout the code, thereby making it difficult to code accurately.
- **SOLUTION**: The “Monitor” is a programming language construct that provides identical functionality but is easier to control.

Semaphore Summary

- Semaphores provide a primitive tool that is both flexible and powerful for enforcing mutual exclusion.
- **PROBLEM**: It may be necessary to have the *wait* and *signal* operations scattered throughout the code, thereby making it difficult to code accurately.
- **SOLUTION**: The “Monitor” is a programming language construct that provides identical functionality but is easier to control.
- **HOMEWORK**: Study the classic problem involving semaphores known as “The Barbershop Problem”, or “The Sleeping Barber Problem”.

MONITORS

- An object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion.

MONITORS

- An object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion.
- Monitors use conditions to solve the synchronization problem by defining a new variable type called *condition*:
  - wait(condition): blocks the current process until another process signals the condition
  - signal(condition): unblocks exactly one waiting process (does nothing if no processes are waiting)

Monitors (1)

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .       .       .
        end;
end monitor;
```

Figure 2-33. A monitor.

Monitors (2)

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure 2-34. An outline of the producer-consumer problem with monitors.

Message Passing

- Processes must **communicate** (cooperating processes may need to share information) and be **synchronized** (to enforce mutual exclusion) for proper interaction.
- **Message Passing** accomplishes both these functions.

Message Passing

- Message Operations:
  - send (destination, message)
  - receive (source, message)

Message Passing

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

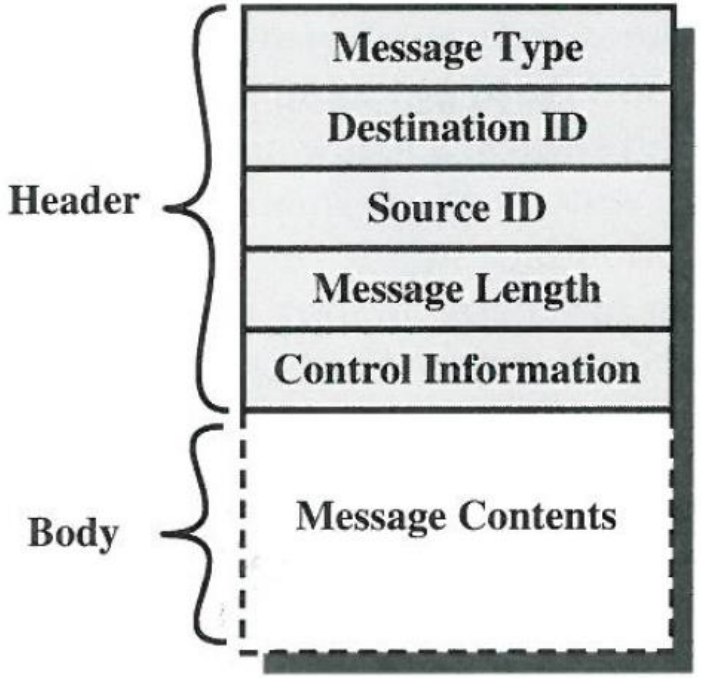
Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queuing Discipline
- FIFO
- Priority

Design characteristics of Message Passing systems for IPC and synchronization (Stallings)

Message Passing



General message format for Message Passing. (Stallings)

Message Passing (1)

```
public class ProducerConsumer {
        static final int N = 100;       // constant giving the buffer size
        static producer p = new producer( );    // instantiate a new producer thread
        static consumer c = new consumer( ); // instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );      // instantiate a new monitor

        public static void main(String args[ ]) {
           p.start( );      // start the producer thread
           c.start( );      // start the consumer thread
        }

        static class producer extends Thread {
           public void run( ) {// run method contains the thread code
              int item;
              while (true) {      // producer loop
                 item = produce_item( );
                 mon.insert(item);
              }
           }
           private int produce_item( ) { ... }      // actually produce
        }
```

. . .

Figure 2-35. A solution to the producer-consumer problem in Java.

Message Passing (2)

. . .

```
static class consumer extends Thread {
   public void run( ) {run method contains the thread code
      int item;
      while (true) {      // consumer loop
         item = mon.remove( );
         consume_item (item);
      }
   }
   private void consume_item(int item) { ... }// actually consume
}

static class our_monitor {  // this is a monitor
   private int buffer[ ] = new int[N];
   private int count = 0, lo = 0, hi = 0;  // counters and indices

   public synchronized void insert(int val) {
      if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
      buffer [hi] = val; // insert an item into the buffer
      hi = (hi + 1) % N;        // slot to place next item in
      count = count + 1;       // one more item in the buffer now
      if (count == 1) notify( );        // if consumer was sleeping, wake it up
   }
```

. . .

Figure 2-35. A solution to the producer-consumer problem in Java.

Message Passing (3)

. . .

```
        public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
            val = buffer [lo]; // fetch an item from the buffer
            lo = (lo + 1) % N;          // slot to fetch next item from
            count = count – 1;          // one few items in the buffer
            if (count == N – 1) notify( ); // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};
    }
}
```

Figure 2-35. A solution to the producer-consumer problem in Java.

Producer-Consumer Problem with Message Passing (1)

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}
```

. . .

Figure 2-36. The producer-consumer problem with N messages.

Producer-Consumer Problem with Message Passing (2)

. . .

```
void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);          /* get message containing item */
          item = extract_item(&m);        /* extract item from message */
          send(producer, &m);             /* send back empty reply */
          consume_item(item);             /* do something with the item */
     }
}
```

Figure 2-36. The producer-consumer problem with N messages.

Barriers

- Another type of synchronization method.
- A barrier for *a group of threads or processes* in the source code means that any thread/process must stop at this point and cannot proceed until *all* other threads/processes reach this barrier.

Barriers

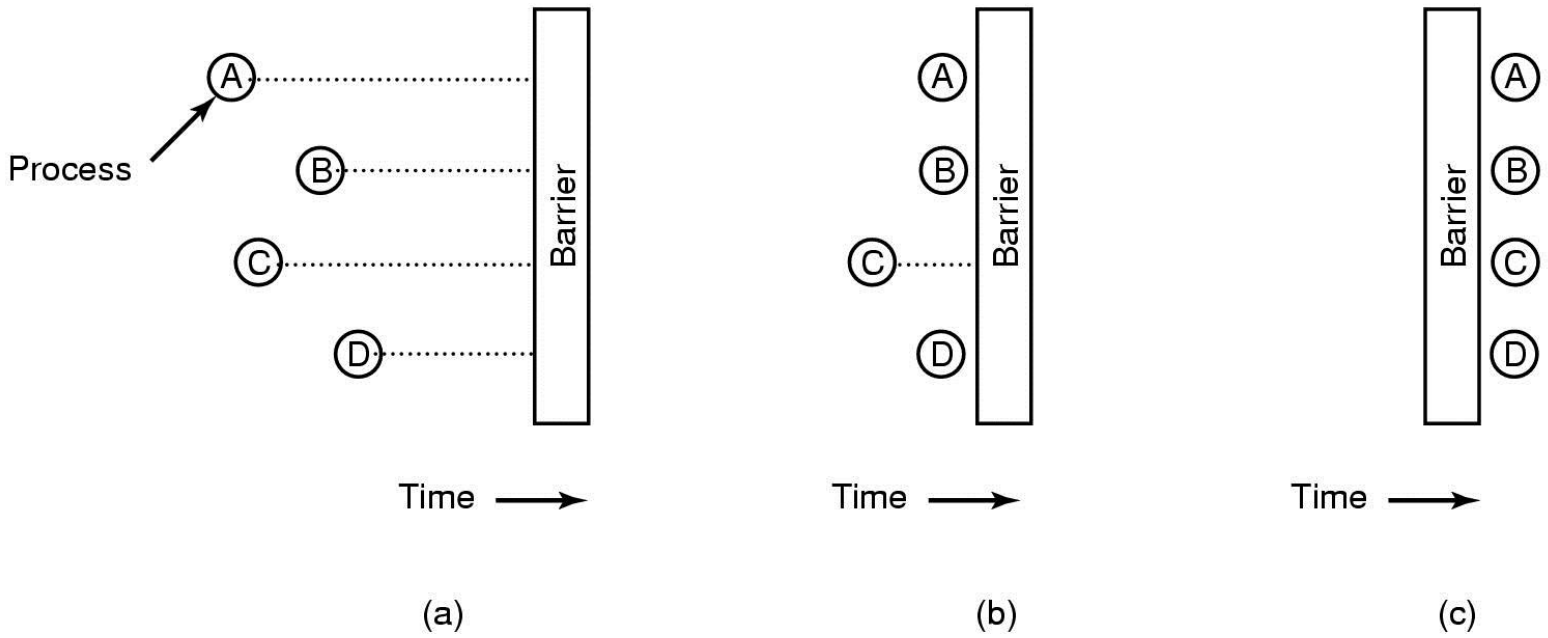


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

```
void send(PCB* from, PCB* to, char* buffer)
{
  // return error if process does not exist
  if(to == 0 || to->state == STOPPED)
  {
    from->syscall_ret = -1;
    return;
  }

  // if to is blocked waiting for this message, deliver it directly!
  if(to->state == BLOCKEDRECV && (to->waiting_for == from->pid || to->waiting_for == -1)
  {
    // if to is blocked & can recv msg, msg should be copied into recv buffer
    // and both procs placed on the ready queue
    copy_buffer(buffer, to->buffer, 8);
    to->syscall_ret = 0;
    ready(to);

    from->syscall_ret = 0;
    ready(from);
  }
  else
  {
    // this condition should satisfy
    //    - receiving process is blocked waiting for something else
    //    - receiving process is not blocked

    ipc_msg *msg = kmalloc(sizeof(ipc_msg));
    msg->from = from->pid;
    msg->to = to->pid;
    msg->next_msg = 0;
```

END PRESENTATION