

Processes and Threads

Week 05 - Lecture 1

Interprocess Communication (IPC)

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
 - Tanenbaum & Bo, *Modern Operating Systems*: 4th edition, 2013
Prentice-Hall, Inc.and customised for the needs of this course.
- Additional input for the slides are detailed later

Interprocess Communication (1)

- Processes frequently need to communicate with other processes
- For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line
- Such a communication between processes (**InterProcess Communication**, or **IPC**), should be carried out in a well-structured way not using interrupts

Interprocess Communication (2)

- Very briefly, there are three issues here:
 - how one process can pass information to another
 - making sure two or more processes do not get in each other's way accessing the same resource
 - proper sequencing when dependencies are present: if process A produces data and process B prints them, B has to wait until A has produced some data before starting to print
- The latter two of these issues apply equally well to threads, however, the same solutions also apply

Race Conditions (1)

- To see how IPC works, let's consider a common example: a print spooler:
 - When a process wants to print a file, it enters the file name in a special spooler directory
 - Another process, the printer daemon, periodically checks to see if there are any files to be printed, prints them and then removes their names from the directory
 - The spooler directory has a number of slots, numbered 0 to n , each one capable of holding a file name

Race Conditions (2)

- There are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory
- These two variables might well be kept in a two-word file available to all processes
- At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes A and B decide they want to queue a file for printing (Fig. 2-21)

Race Conditions (3)

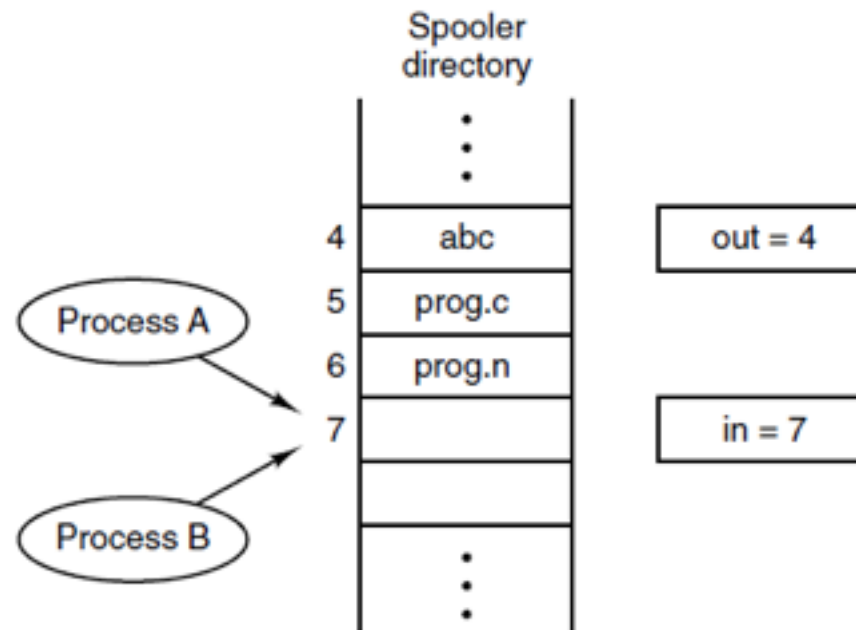
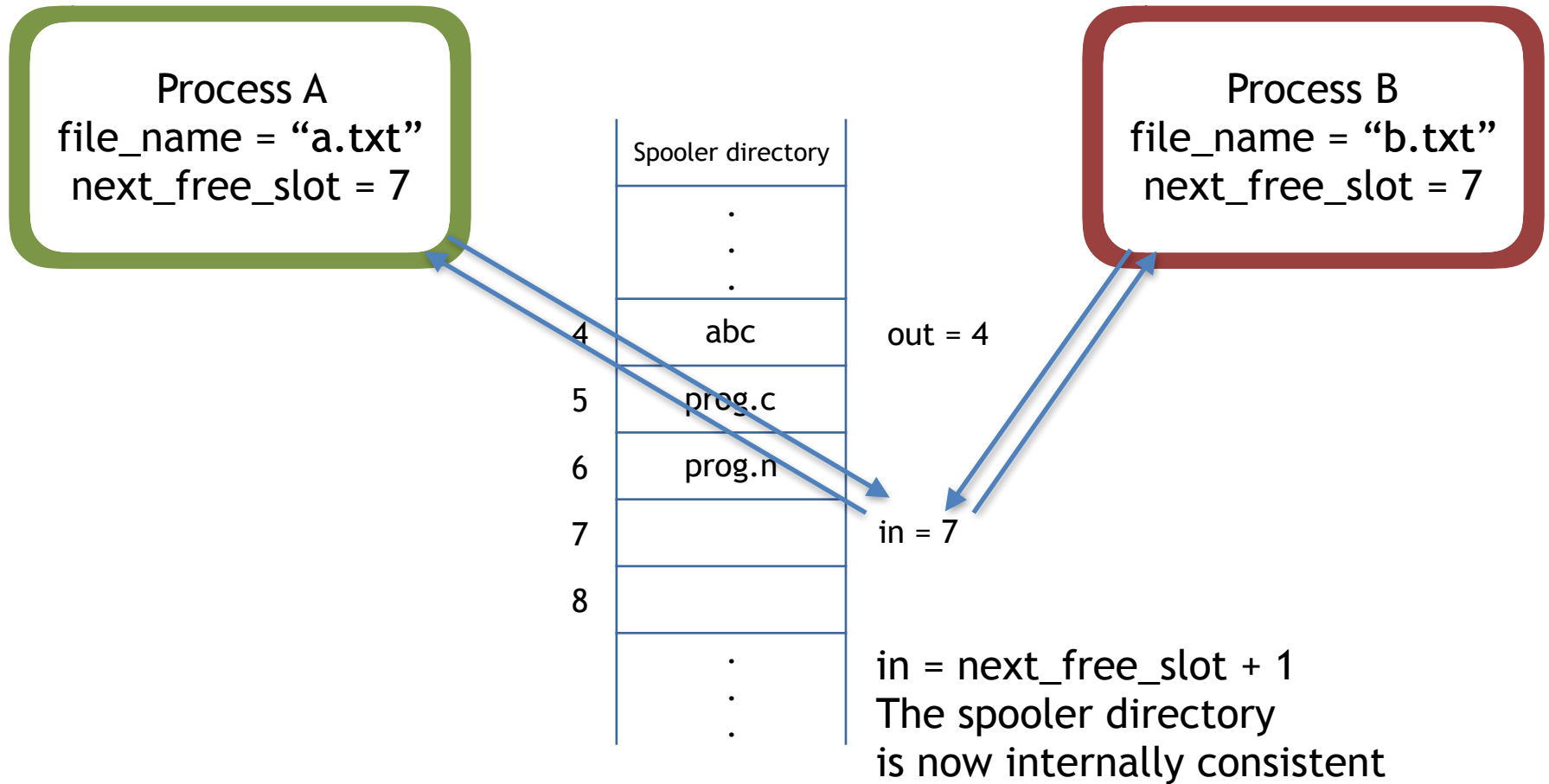


Figure 2-21. Two processes want to access shared memory at the same time.

Race Conditions (4)



Avoiding Race Conditions

- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Semaphores
- Mutexes
- Monitors
- Message Passing
- Barriers
- Avoiding Locks: Read-Copy-Update

Critical Regions (1)

- A way to prohibit more than one process from reading and writing the shared data at the same time is called **mutual exclusion**:
 - if one process is using a shared variable or file, the other processes will be excluded from doing the same thing
- The part of the program where the shared memory is accessed is called the **critical region** or **critical section** (figure 2-22)

Critical Regions (2)

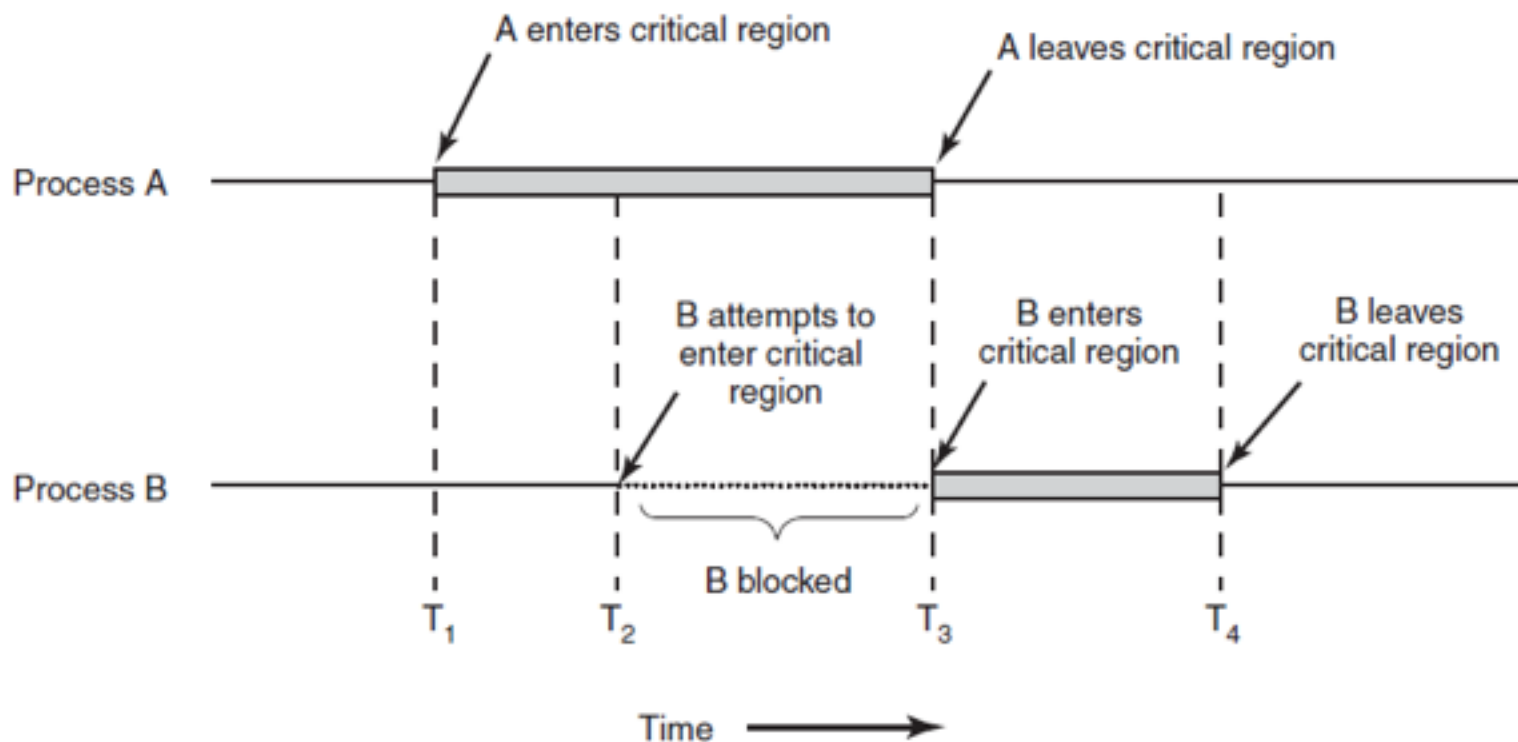


Figure 2-22. Mutual exclusion using critical regions.

Requirements to Implement Critical Regions

- No two processes may be simultaneously inside their critical regions
- No assumptions may be made about speeds or the number of CPUs
- No process running outside its critical region may block other processes
- No process should have to wait forever to enter its critical region

Proposals to Implement Critical Regions

- Disabling interrupts
- Lock variables
- Strict Alternation
- Peterson's Solution
- The TSL Instruction

Mutual Exclusion with Busy Waiting: Disabling Interrupts

- The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it:
 - Too much power for a process: if, because of a bug, a process will not re-enable interrupts, this will stop the system
 - Will not work in the multiprocessor systems since only interrupts for the CPU that executed the ***disable*** instruction will be stopped
- Disabling interrupts is often a useful technique within the OS itself, since it may update some variables, for example, a list of ready processes

Mutual Exclusion with Busy Waiting: Lock Variables

- A possible software solution:
 - A lock variable is a single, shared variable that is initially 0
 - A process entering its critical region tests the lock first
 - If the lock is 0, the process sets it to 1 and enters the critical region
 - If the lock is already 1, the process just waits until it becomes 0
- This idea contains exactly the same fatal flaw that was described earlier:
 - One process might read the lock and see that it is 0
 - Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1
 - When the first process runs again, it will also set the lock to 1
 - Two processes will be in their critical regions at the same time

Mutual Exclusion with Busy Waiting: Strict Alternation (1)

- The integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region
- Initially, process 0 inspects turn, finds it to be 0, and enters its critical region
- Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1
- Continuously testing a variable until some value appears is called **busy waiting**
- A lock that uses busy waiting is called a **spin lock**

Mutual Exclusion with Busy Waiting: Strict Alternation (2)

```
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.

Mutual Exclusion with Busy Waiting: Strict Alternation (3)

- This solution does avoid all races, but violates condition 3:

No process running outside its critical region may block other processes

- If one process is significantly slower, it will enter a critical region, exit it then will pass the turn to the second process and will continue executing in its noncritical region
- The second process will quickly finish its job, will pass the turn to the first process which might still not be done
- Thus, the second process will have to wait for a process in noncritical region (condition 3)

Mutual Exclusion with Busy Waiting: Peterson's Solution (1)

- Before entering its critical region, each process calls *enter_region()* with its own process number, 0 or 1, as parameter
- This call will cause it to wait until it is safe to enter
- After it has finished with the shared variables, the process calls *leave_region()* to indicate that it is done

Mutual Exclusion with Busy Waiting: Peterson's Solution (2)

```
#define FALSE 0
#define TRUE 1
#define N      2                /* number of processes */

int turn;                       /* whose turn is it? */
int interested[N];              /* all values initially 0 (FALSE) */

void enter_region(int process);  /* process is 0 or 1 */
{
    int other;                   /* number of the other process */

    other = 1 - process;         /* the opposite of process */
    interested[process] = TRUE;  /* show that you are interested */
    turn = process;              /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)   /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Mutual Exclusion with Busy Waiting: Peterson's Solution (3)

- Let's consider the case that both processes call enter region almost simultaneously:
 - Both will store their process number in *turn*
 - Whichever store is done last is the one that counts; the first one is overwritten and lost
 - Suppose that process 1 stores last, so turn is 1
 - When both processes come to the while statement, process 0 executes it zero times and enters its critical region
 - Process 1 loops and does not enter its critical region until process 0 exits its critical region

Mutual Exclusion with Busy Waiting: The TSL Instruction (1)

- Hardware-assisted approach; works with multiple CPUs
- An instruction like ***TSL RX,LOCK*** (Test and Set Lock) reads the contents of the memory word *lock* into register *RX* and then stores a nonzero value at the memory address *lock*
- The operations of reading the word and storing into it are guaranteed to be indivisible - no other processor can access the memory since The CPU executing the TSL instruction **locks the memory bus**
- Different from disabling interrupts - doing it on processor 1 has no effect at all on processor 2

Mutual Exclusion with Busy Waiting: The TSL Instruction (2)

enter_region:

```
TSL REGISTER,LOCK  
CMP REGISTER,#0  
JNE enter_region  
RET
```

```
| copy lock to register and set lock to 1  
| was lock zero?  
| if it was nonzero, lock was set, so loop  
| return to caller; critical region entered
```

leave_region:

```
MOVE LOCK,#0  
RET
```

```
| store a 0 in lock  
| return to caller
```

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

Mutual Exclusion with Busy Waiting: The TSL Instruction (3)

- The first instruction copies the old value of lock to the register and then sets lock to 1
- The old value is compared with 0:
 - If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again
 - Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set
- To clear the lock the program just stores a 0 in *lock*

Mutual Exclusion with Busy Waiting: The TSL Instruction (4)

```
enter_region:
    MOVE REGISTER,#1      | put a 1 in the register
    XCHG REGISTER,LOCK     | swap the contents of the register and lock variable
    CMP REGISTER,#0       | was lock zero?
    JNE enter_region      | if it was non zero, lock was set, so loop
    RET                   | return to caller; critical region entered

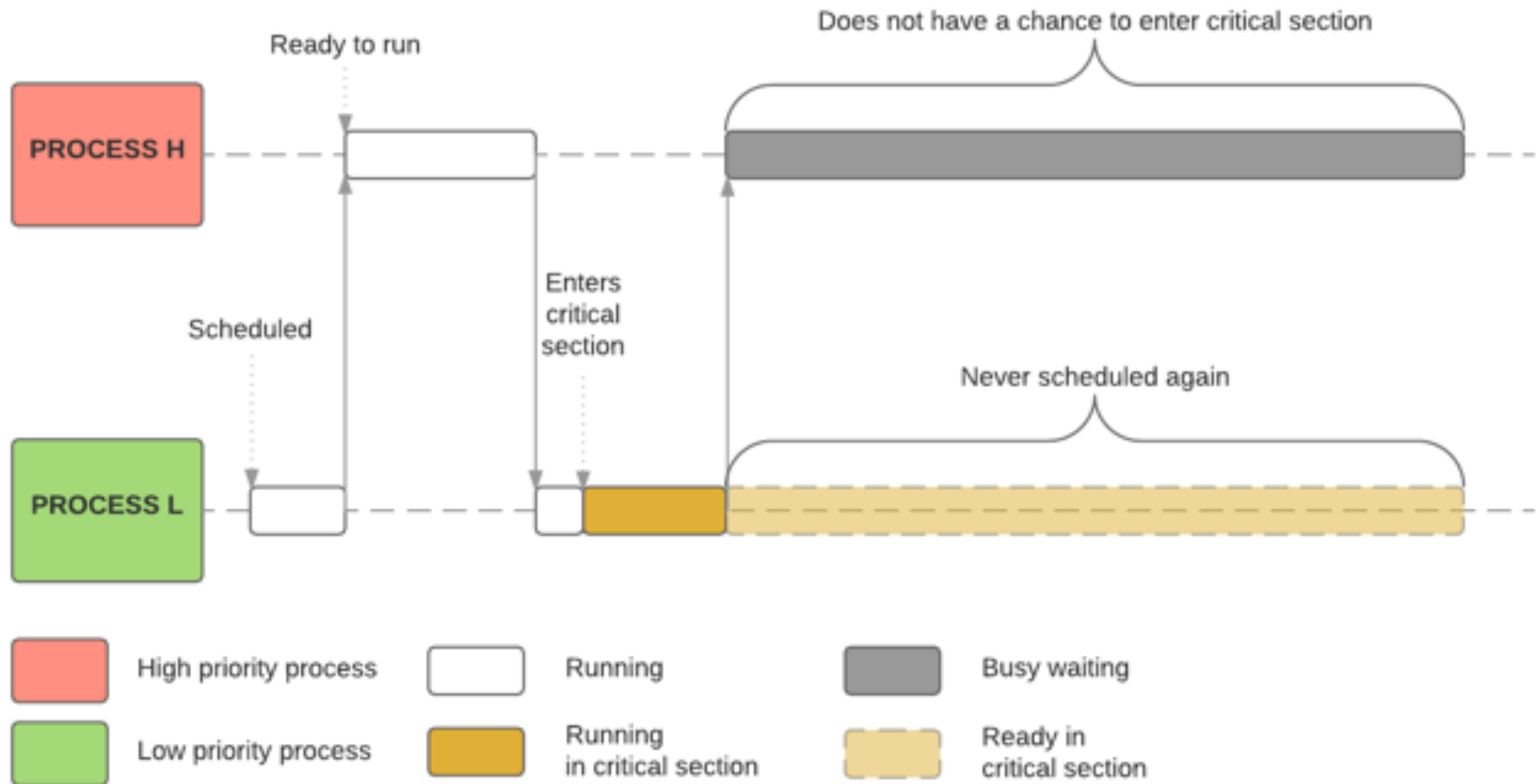
leave_region:
    MOVE LOCK,#0          | store a 0 in lock
    RET                   | return to caller
```

Figure 2-26. Entering and leaving a critical region
using the alternative XCHG instruction

Sleep and Wakeup (1)

- Both Peterson's solution and the solutions using TSL or XCHG are correct, but both have the **defect of requiring busy waiting**
- It can also have unexpected effects:
 - Consider two processes, H , with high priority, and L , with low priority
 - The scheduling rules are such that H is run whenever it is in ready state
 - At a certain moment, with L in its critical region, H becomes ready to run
 - H now begins busy waiting, but L is never scheduled while H is running
 - L never gets the chance to leave its critical region and H loops forever

Sleep and Wakeup (2)



Priority inversion problem

Sleep and Wakeup (3)

- This situation is sometimes referred to as the **priority inversion problem**
- One of the simplest interprocess communication primitives that block instead of wasting CPU time is the pair of system calls **sleep** and **wakeup**:
 - **Sleep** causes the caller to be suspended (blocked) until another process wakes it up
 - **Wakeup** accepts the process to be awakened as a parameter and wakes this process up (supposedly, changes its state to *Ready*)

Sleep and Wakeup

The Producer-Consumer Problem (1)

- The producer-consumer problem (bounded-buffer problem):
 - Two processes share a common, fixed-size buffer
 - One of them, the producer, puts information into the buffer
 - The other one, the consumer, takes it out
 - When the producer wants to put a new item in the buffer, but it is already full, it goes to sleep
 - Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep too

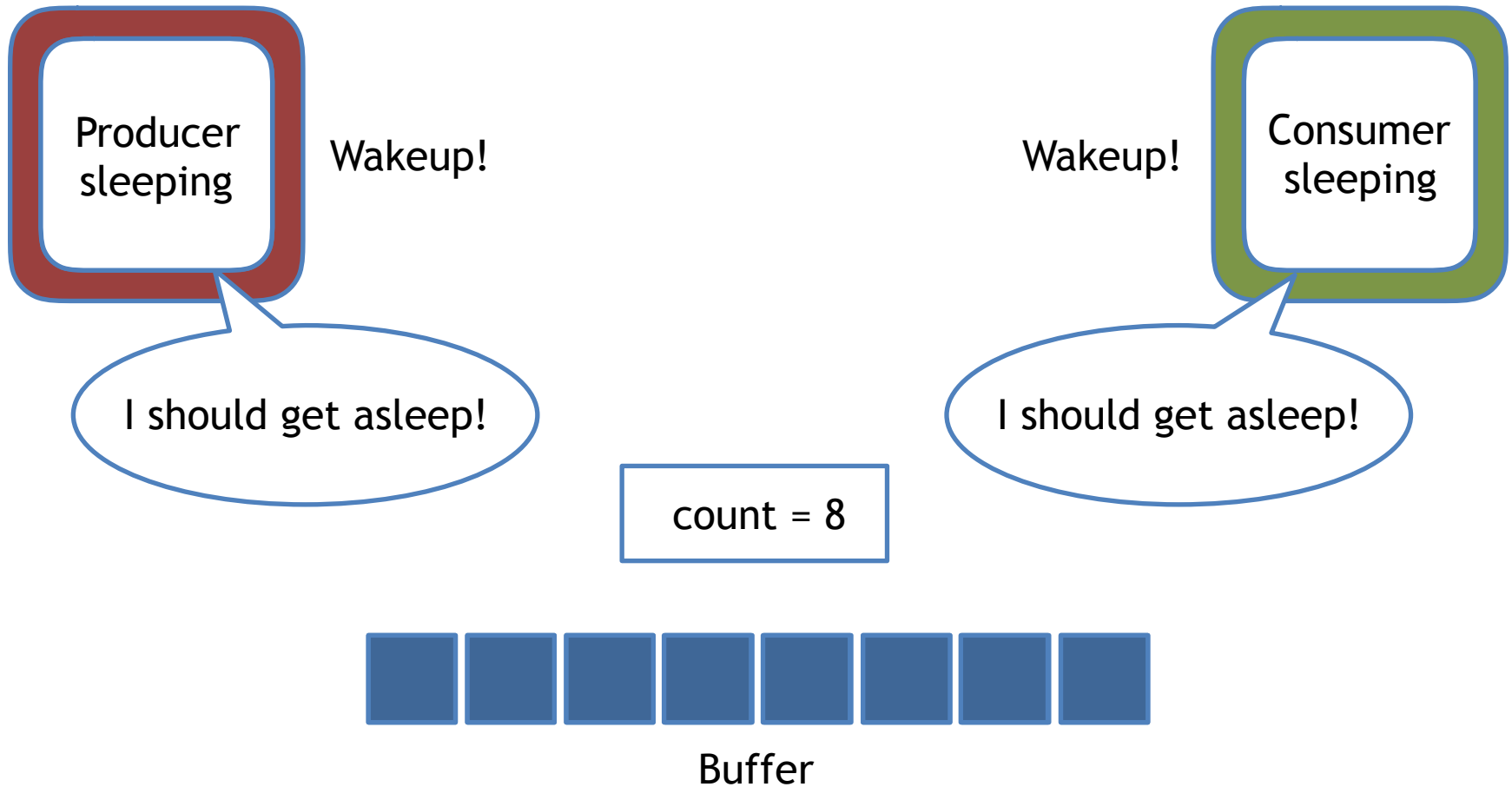
Sleep and Wakeup

The Producer-Consumer Problem (2)

- To keep track of the number of items in the buffer, we will need a variable, *count*
- The producer's code will first test to see if *count* is N (the maximum number of items in the buffer)
- If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*
- Consumer first tests *count* to see if it is 0
- If it is, go to sleep; if it is nonzero, remove an item and decrement the counter
- Each of the processes also tests to see if the other should be awakened, and if so, wakes it up

Sleep and Wakeup

The Producer-Consumer Problem (3)



Sleep and Wakeup

The Producer-Consumer Problem (3)

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

void producer(void) {
    int item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep(); /*if buffer is full, go
                                   to sleep*/
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
                                   /*was buffer empty?*/
    }
}
```

Sleep and Wakeup

The Producer-Consumer Problem (4)

```
void consumer(void) {
    int item;

    while (TRUE) {
        if (count == 0) sleep( );
        /* buffer empty => to sleep */
        item = remove_item( );
        /*take item out of buffer*/
        count = count - 1;
        if (count == N - 1) wakeup(producer);
                                /*was buffer full?*/
        consume_item(item);      /*print item*/
    }
}
```

Sleep and Wakeup

The Producer-Consumer Problem (5)

- Race condition can occur because access to *count* is unconstrained
- The following situation could possibly occur:
 - The buffer is empty and the consumer has just read **count** to see if it is 0 (it hasn't fallen asleep yet)
 - The scheduler decides to stop running the consumer temporarily and start running the producer
 - The producer inserts an item in the buffer, increments count, and notices that it is now 1
 - The producer calls wakeup to wake the consumer up

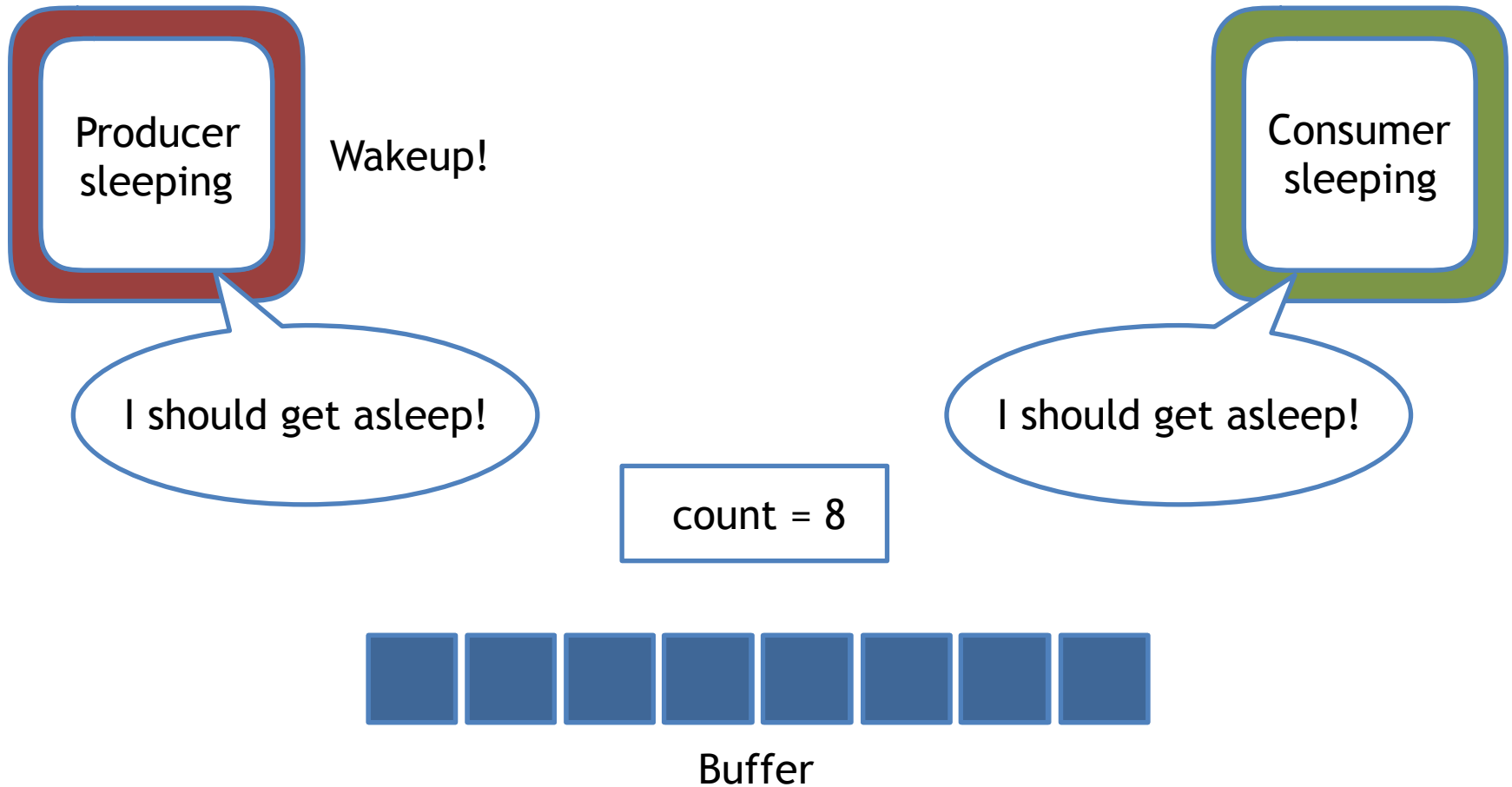
Sleep and Wakeup

The Producer-Consumer Problem (6)

- Since the consumer is not yet logically asleep, the wakeup signal will be lost
- When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep
- Sooner or later the producer will fill up the buffer and also go to sleep
- Both will sleep forever

Sleep and Wakeup

The Producer-Consumer Problem (7)



Sleep and Wakeup

The Producer-Consumer Problem (8)

- To fix this situation, it is possible to add a **wakeup waiting bit**:
 - When a wakeup is sent to a process that is still awake, this bit is set
 - 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 consumer clears the wakeup waiting bit in every iteration of the loop
- However, it is easy to construct examples with three or more processes in which one wakeup waiting bit is insufficient. Even when using more bits, the essence of the problem is still here

Semaphores (1)

- In 1965, Dijkstra suggested using an integer variable to count the number of wakeups saved for future use - a **semaphore**
- A semaphore could have some positive value if one or more wakeups were pending, or could be 0 otherwise
- Dijkstra proposed having two operations on semaphores, now usually called **down** and **up** (generalizations of **sleep** and **wakeup**, respectively)
- The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues

Semaphores (2)

- If the value is 0, the process is put to sleep without completing the down for the moment
- Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible atomic action
- It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked
- This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions

Semaphores (3)

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

Semaphores (4)

- Semaphores solve the lost-wakeup problem
- To make them work correctly, it is essential that they be implemented in an indivisible way
- The normal way is to implement *up()* and *down()* as system calls, with the OS briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary
- All of these actions take only a few instructions, so no harm is done in disabling interrupts
- If multiple CPUs are being used, each semaphore should be protected by a *lock variable* with the TSL or XCHG instructions

Semaphores (5)

```
void producer(void) {
    int item;

    while (TRUE) {
        item = produce_item();    /*generate an item*/

        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*/
    }
}
```

Semaphores (6)

```
void consumer(void) {
    int item;
    while (TRUE) {
        down(&full);
        down(&mutex);

        item = remove_item();

        up(&mutex);
        up(&empty);

        consume_item(item);
    }
}
```

*/*infinite loop*/*
*/*decrement full count*/*
*/*enter critical region*/*

*/*take item from buffer*/*
*/*leave critical region*/*
*/*increment count of empty slots*/*
*/*do something with the item*/*

Semaphores (7)

- The solution uses three semaphores
 - one called *full* for counting the number of slots that are full
 - one called *empty* for counting the number of slots that are empty
 - one called *mutex* to make sure the producer and consumer do not access the buffer at the same time
- *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer, and *mutex* is initially 1
- Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**
- If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed

Semaphores (8)

- We used semaphores in two different ways:
 - The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables
 - The other use of semaphores is for **synchronization**. The *full* and *empty* semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and that the consumer stops running when it is empty

Mutexes (1)

- A **mutex** is a shared variable that can be in one of two states: unlocked or locked
- Only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked
- Mutexes are good only for managing mutual exclusion to some shared resource or piece of code
- They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space

Mutexes (2)

- Two procedures are used with mutexes:
 - When a thread (or process) needs access to a critical region, it calls ***mutex_lock***. If the mutex is currently unlocked, the call succeeds and the calling thread is free to enter the critical region
 - If the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls ***mutex_unlock***. If multiple threads are blocked on the mutex, one of them is chosen at random
- Mutexes can easily be implemented in user space provided that a TSL or XCHG instruction is available (Fig. 2-29)

Mutexes (3)

```
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 (4)

- *mutex_lock* is similar to the code of *enter_region* that was provided as an example of TSL instruction (Fig. 2-25), but there is a crucial difference:
 - When *enter_region* fails to enter the critical region, it keeps testing the lock repeatedly (busy waiting)
 - With (user) threads, the situation is different because there is no clock that stops threads that have run too long
 - A thread that tries to acquire a lock by busy waiting will loop forever and never acquire the lock because it never allows any other thread to run and release the lock

Mutexes (5)

- When the *mutex_lock* fails to acquire a lock, it calls *thread_yield* to give up the CPU to another thread - there is no busy waiting
- Since *thread_yield* is just a call to the thread scheduler in user space, it is very fast.
- Neither mutex lock nor mutex unlock requires any kernel calls
- Using them, user-level threads can synchronize entirely in user space using procedures that require only a handful of instructions

Mutexes (6)

- The mutex system just described is a bare-bones set of calls
- With all software, there is always a demand for more features, and synchronization primitives are no exception
- For example, sometimes a thread package offers a call *mutex_trylock* that either acquires the lock or returns a code for failure, but does not block

Mutexes (7)

- When we work with user-space threads, there is no problem with multiple threads having access to the same mutex
- If processes have disjoint address spaces, how can they share the *turn* variable, semaphores or a common buffer?
 - Some of the shared data structures, such as the semaphores, can be stored in the kernel and accessed only by means of system calls
 - Most modern OSs offer a way for processes to share some portion of their address space with other processes. In this way, buffers and other data structures can be shared

Monitors (1)

- A **monitor** is a higher-level synchronization primitive - it is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package (Fig. 2-33)
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor

Monitors (2)

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    end;

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

Figure 2-33. A monitor.

Monitors (3)

```
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;
```




Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

Monitors (4)

```
~~~~~  
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;  
end;
```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

Monitors (5)

- Only one process can be active in a monitor at any moment
- When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor
- If no other process is using the monitor, the calling process may enter
- It is up to the compiler to implement mutual exclusion on monitor entries, but a common way is to use a mutex or a binary semaphore

Monitors (6)

- We also need a way for processes to block when they cannot proceed
- The solution lies in the introduction of condition variables, along with two operations on them, *wait* and *signal*
- When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a *wait* on some condition variable. This action causes the calling process to block
- The other process, for example, the consumer, can wake up its sleeping partner by doing a *signal* on the condition variable that its partner is waiting on

Monitors (7)

- To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a signal:
 - One solution is letting the newly awakened process run, suspending the other one
 - The second one is requiring that a process doing a signal must exit the monitor immediately (a signal statement may appear only as the final statement in a monitor procedure)
 - The third one is to let the signaler continue to run and allow the waiting process to start running only after the signaler has exited the monitor

Monitors (5)

```
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
    }

    static class consumer extends Thread {
```




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

Monitors (6)

```
private int produce_item() { ... }    // actually produce
}

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) {
        // insert item into buffer if the buffer is full then
    }
}
```

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

Monitors (7)

```
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
}

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 fewer 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.

Monitors (8)

- The producer and consumer threads are functionally identical to their counterparts in all previous examples
- The interesting part of this program is the class *our_monitor*, which holds the buffer, the administration variables, and two synchronized methods
- When the producer is active inside *insert*, it knows for sure that the consumer cannot be active inside *remove*, making it safe to update the variables and the buffer without fear of race conditions
- The variable *count* (any value from 0 to $N - 1$) keeps track of how many items are in the buffer
- The variable *lo* is the index of the buffer slot where the next item is to be fetched
- *hi* is the index of the buffer slot where the next item is to be placed
- It is permitted that $lo = hi$, which means that either 0 items or N items are in the buffer. The value of *count* tells which case holds

Monitors (9)

- By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error prone than using semaphores
- Nevertheless, they too have some drawbacks:
 - Monitors are a **programming-language concept**. C, Pascal, and most other languages do not have monitors, so it is unreasonable to expect their compilers to enforce any mutual exclusion rules
 - Another problem with monitors, and also with semaphores, is that they were designed for solving the mutual exclusion problem on one or more CPUs that all **have access to a common memory**. In a distributed system consisting of multiple CPUs, each with its own private memory, these primitives become inapplicable

Message Passing

- **message passing** is the method of IPC that uses two primitives, *send* and *receive*, which, like semaphores and unlike monitors, are system calls rather than language constructs (Fig. 2-36)
- They can easily be put into library procedures, such as
 - `send(destination, &message);`
 - `receive(source, &message);`
- The former call sends a message to a given destination and the latter one receives a message from a given source (or from ANY, if the receiver does not care)
- If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code
- The main issue: **messages can be lost by the network**. The receiver might not be able to distinguish a new message from the retransmission of an old one

End

Week 05 - Lecture 1

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
Prentice-Hall, Inc.