Pay super attention here since this is usually the most difficult. Also try implementing this on your own.

Concurrency: Mutual Exclusion and Synchronization

Chapter 5

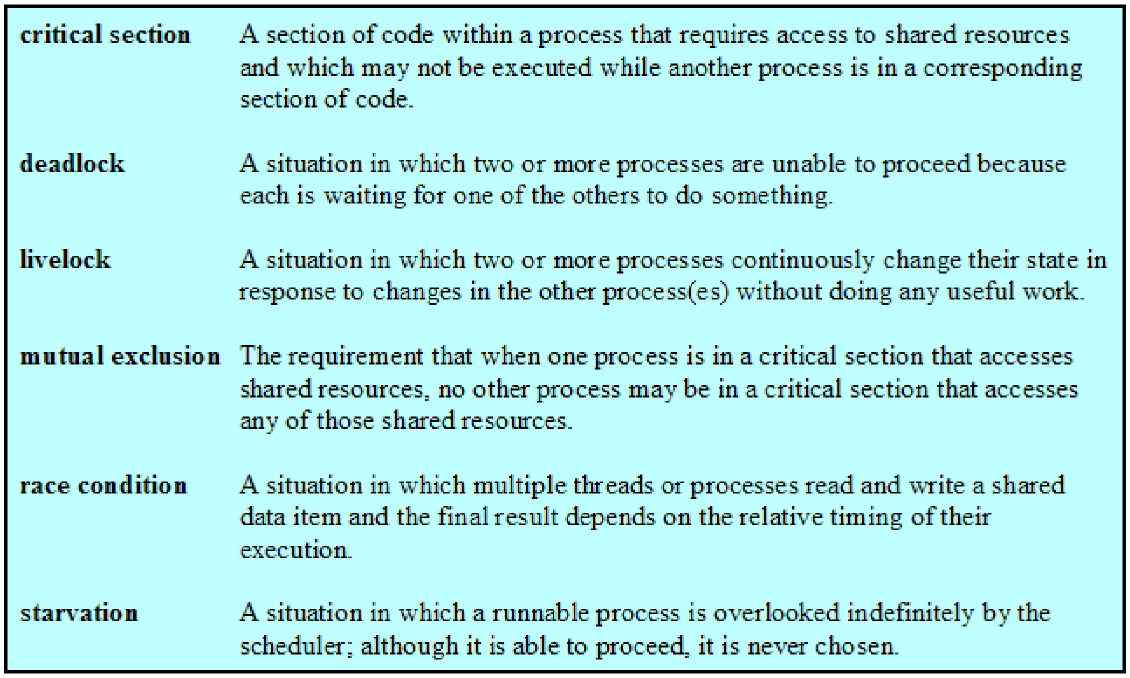
Concurrency

- Multiple applications
  - Multiprogramming
- Structured application
  - Application can be a set of concurrent processes
- Operating-system structure
  - Operating system is a set of processes or threads

Concurrency is just mulitple things running on your system. It leads to multiprogramming (able to execute multiple programs on a single processor) think threads, this is also just a part of the OS structure.

Concurrency

Table 5.1 Some Key Terms Related to Concurrency



critical section	A section of code within a process that requires access to shared resources and which may not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

KNOW THE FUCK OUT OF THESE!!!!!

- critical section: part that requires resources and cannot be executed by another processes, ex the operating system is the only thing that can execute some instructions
- deadlocks: processes get stuck waiting on eachother
- liveblock: processes that keep flipping state around eachother with keeps them from getting any work done
- mutual exclusion: enforces the rules around critical section
- race condition: multiple threads or processes that share a data item so timing of execution matters for the end result, you lose transparency
- starvation: a process is ready but the scheduler keeps skipping over it

Difficulties of Concurrency

- Sharing of global resources
- Operating system managing the allocation of resources optimally
- Difficult to locate programming errors

We need to share global resources between the multiple processes so that the processes leave in a consistent state and these resources need to be managed optimally. We cannot predict concurrency related problems (race conditions are the worst). It is possible to check for this using assertions, but we change the timing of things when these are removed outside of debugging mode which can show new race conditions that were hidden previously due to a change in timing.

A Simple Example

Setup: two processes/two threads

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

Notice that chin and chout are global variables so they are shared. Lets say we have T1 and T2 that are two threads in this address space. They will be sharing chin and chout. Say T1 gets picked up first if we have only one thread its a given that the input of getchar will be outputted in putchar. This may not be the case here.

A Simple Example

Setup: two processes/two threads, multiprocessor

Hint: chin, chout are global variables, getchar echoes characters.

```
  Process P1      Process P2
.                   .
chin = getchar();   .
.              chin = getchar();
chout = chin;     chout = chin;
putchar(chout);     .
     .            putchar(chout);
     .                .
```

The scheduler will try to do both at the same time which may result in getchar being called twice. This means that chin will be overwritten in the call of T2. Now say that both T1 and T2 are picked up so they both save chin to chout at the same time (pretend that this wont cause an issue) both times with the second input value since chin was overwritten then both threads will have the same value for chout and will output the same thing.

Operating System Concerns

- Keep track of various processes
- Allocate and deallocate resources
  - Processor time
  - Memory
  - Files
  - I/O devices
- Protect data and resources
- Functions of process must be independent of the speed of execution of other concurrent processes

The OS has to protect all resources managed by the OS this includes allocating and freeing them. The OS needs to ensure that when you access resources the OSs actions will be independent to speed (NO RACE CONDITIONS).

Interaction Among Processes

- Processes unaware of each other
- Processes indirectly aware of each other
- Process directly aware of each other

The most detached method is having the processes not know about each other.

Table 5.2 Process Interaction

Degree of Awareness	**Relationship**	**Influence that one Process has on the Other**	**Potential Control Problems**
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Competition Among Processes for Resources

- Mutual Exclusion
  - Critical sections
    - Only one program at a time is allowed in its critical section
    - Example only one process at a time is allowed to send command to the printer
- Deadlock
- Starvation

Mutual exclusion means we need to define critical sections which only one process at a time can access. Deadlock and starvation are going to be covered later.

Requirements for Mutual Exclusion

1. Only one process at a time is allowed in the critical section for a resource
2. A process that halts in its noncritical section must do so without interfering with other processes
3. No deadlock or starvation
4. A process must not be delayed access to a critical section when there is no other process using it
5. No assumptions are made about relative process speeds or number of processes
6. A process remains inside its critical section for a finite time only

Only one process at a time is allowed in a critical section. If a process halts outside of the critical section it cannot fuck with shit. No deadlock or starvation. If no process is executing the critical section other processes should not be forced to wait on it. Mutual exclusion should be independant of the order of processes. Set timeouts for the cricitcal section. The easiest way to enforce mutual exclusion is to use disable_irq for critical section (dont actually do this its really ineeficient).

Mutual Exclusion: Hardware Support

- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion in a uniprocessor system
  - Processor is limited in its ability to interleave programs
  - Multiprocessing
    - disabling interrupts on one processor will not guarantee mutual exclusion
  - NOC: open question

If we were a uniprocessing system disabeling interrupts to protect the critical section is viable but we dont care.

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - Performed in a single instruction cycle
  - Access to the memory location is blocked for any other instructions

Having a specific instruction to access that memory and only allow that one it. Here the processor doesnt know about the critical sections.

Mutual Exclusion: Hardware Support

- Test and Set Instruction

```
boolean testset (int i) {
  if (i == 0) {
    i = 1;
    return true;
  }
  else {
    return false;
  }
}
```

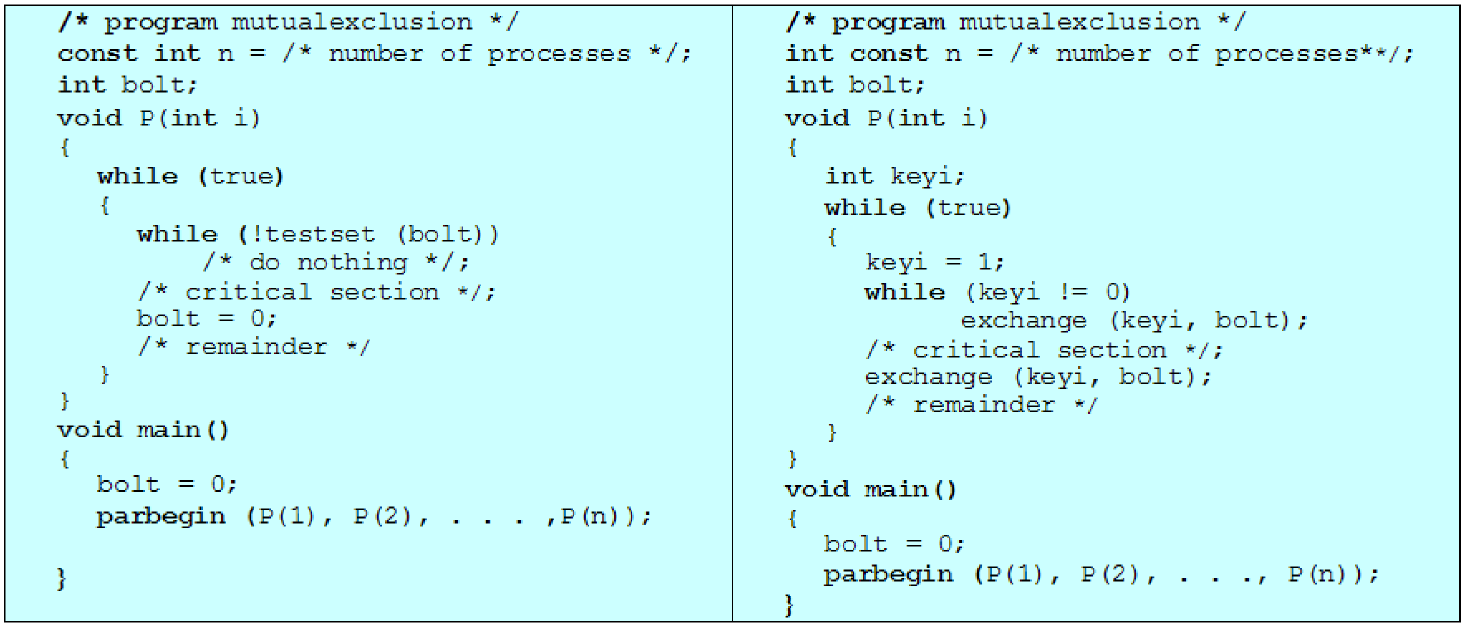
Mutual Exclusion: Hardware Support

- Exchange Instruction

```
void exchange(int register,
   int memory) {
 int temp;
 temp = memory;
 memory = register;
 register = temp;
}
```

The above two examples do pretty much the same thing. Need to look into this, don't really get it.

Mutual Exclusion



```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
            exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

(a) Test and set instruction

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Yeeeeeaaaah really dont get this. It looks like you need some test to check if something is being touched and whether it can be mucked with, but I'm not sure.

Mutual Exclusion Machine Instructions

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

We like doing this with machine instructions:

- can be used in tons of different places
- its pretty simple
- can support multiple critical sections
- if this fucks up we dont leave interrupts disabled

Some disadvantages:

- waiting for critical section to be freed is not handled by the processor so it keeps the processor busy which uses up CPU time
- can cause deadlocks if a low process has a critical section that the high process needs the higher process keeps getting the processor and waits for the critical section to be available but it doesnt happen because the lower process doesnt get called
- starvation if multiple processes are waiting on the same critical section
- pipeline stalls (remember pipelines) happen when be branch multiple times we get stalls as we wait on the critical section which delays the next pipeline

Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region
  - Pipeline stalls

Semaphores

- Special variable called a semaphore is used for signaling
- If a process is waiting for a signal, it is suspended until that signal is sent

Semaphores

- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - Wait operation decrements the semaphore value
  - Signal operation increments semaphore value

Every time you wait on a semaphore we decrease a variable and increase it when you signal for it.

Semaphore Operations

- Initialize with a non-negative value
- semWait() decrements the semaphore value
- semSignal() increments the value

- No other way to access the semaphore (!!)

- In general, anyone can call semWait() and semSignal()
- For a mutex, only the process with the lock can release the lock.

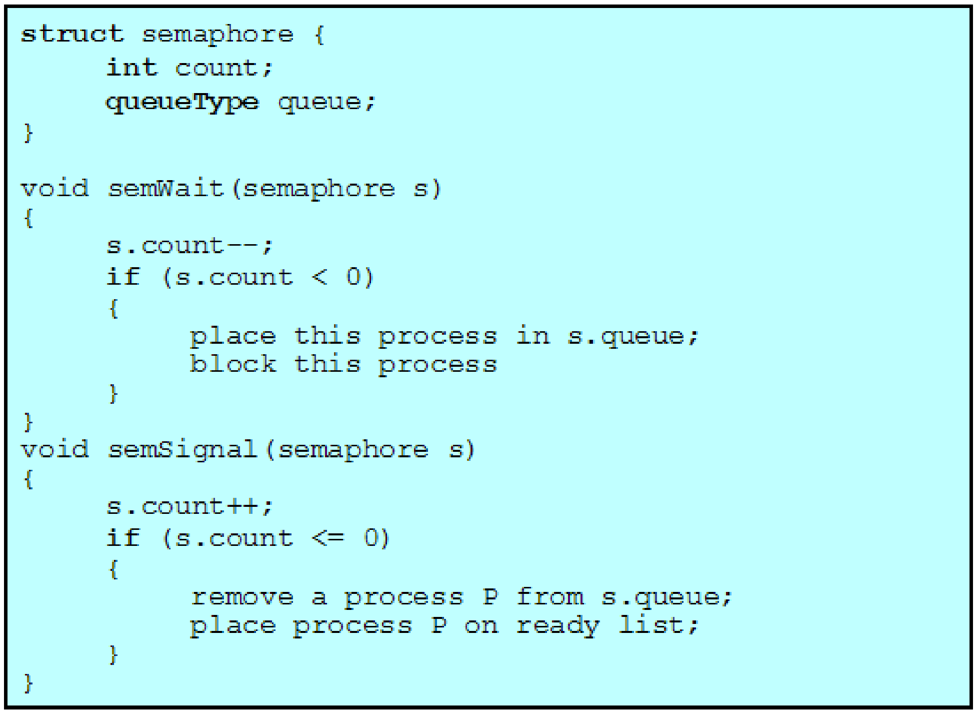
Think about the semaphore as a handler. Anyone who has access to it can send a signal.

Properties of Semaphore Use

- No way to know whether a semWait() will block or not.
- semSignal() may wake up a process. The signaling and unblocked process continue concurrently → you don’t know which is next on a uniprocessor system.
- On semSignal(), you don’t know whether it wakes up a process.

Everytime we enfore mutual exclusion we want to do it independant of the process order.

Semaphore Primitives



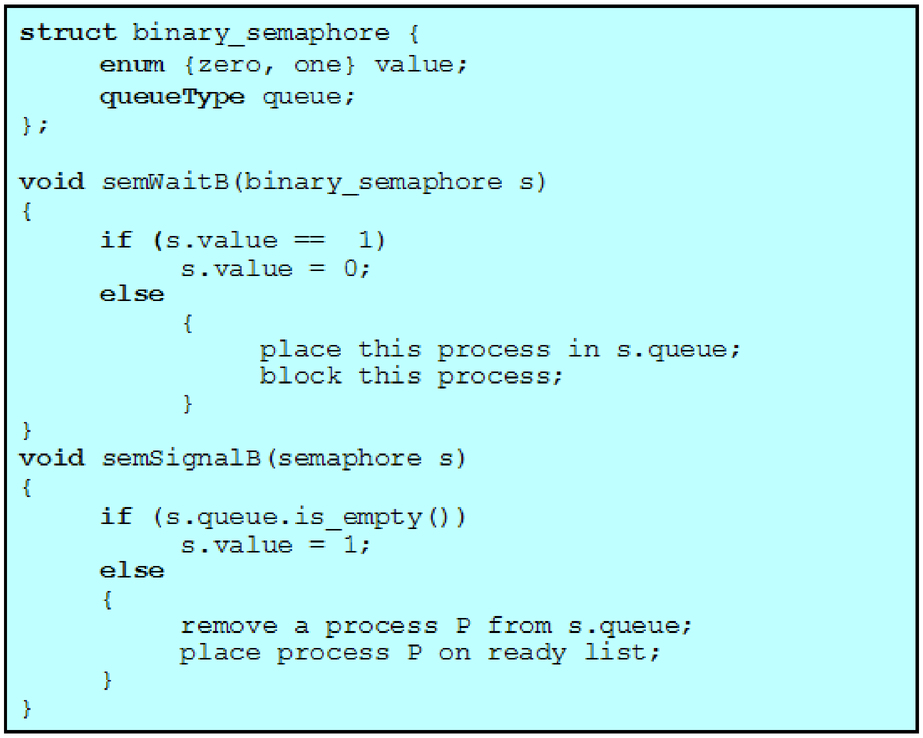
```
struct semaphore {
     int count;
     queueType queue;
}

void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0)
     {
          place this process in s.queue;
          block this process
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0)
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.3 A Definition of Semaphore Primitives

These queuing structures need to be manged by the os to be effective and we need to invoke the dispatcher to use it. Thus these are kernel level processes.

Binary Semaphore Primitives



```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};

void semWaitB(binary_semaphore s)
{
     if (s.value ==  1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
void semSignalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true)
     {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

- Processes A,B,C read data from Process D

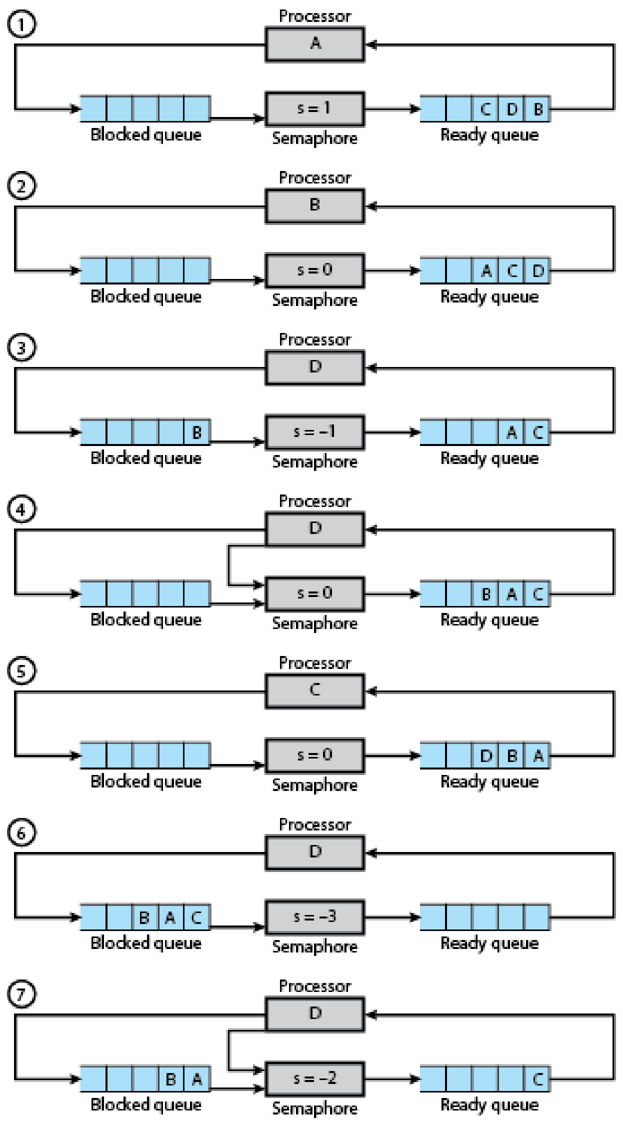


Figure 5.5 Example of Semaphore Mechanism

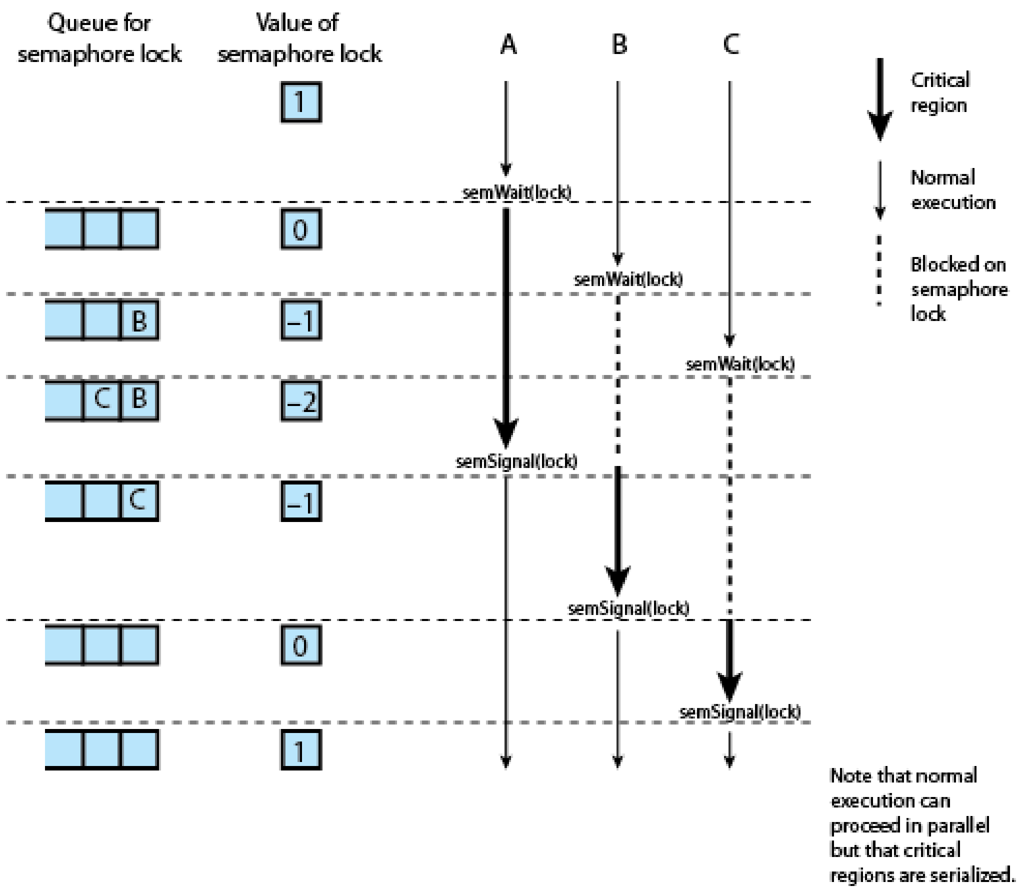


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

Review

- What do we need to implement Semaphores?
- What are the components of Semaphores?
- When do Semaphores block?
- Can Semaphores become negative?
- If Semaphores are used to protect code sections: what does the initial number control?
- … what does the current value indicate?

Producer/Consumer Problem

- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time

The simplest problem is this one. We have multiple producers that put data in a buffer and multiple consumers that can read off the buffer, but only one can go at a time.

Producer

```
producer:
while (true) {
  /* produce item v */
  b[in] = v;
  in++;
}
```

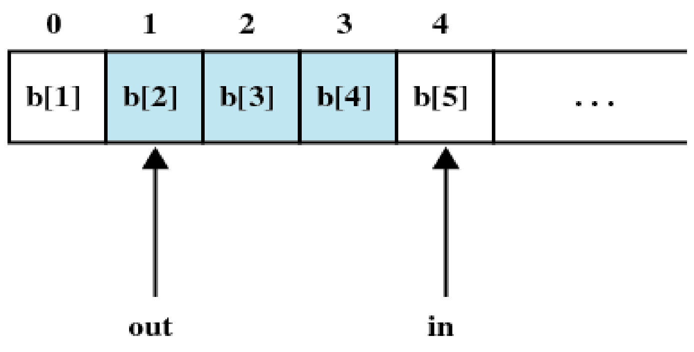
Think of it as an array that can infinitly grow.

Consumer

```
consumer:
while (true) {
  while (in <= out)
   /*do  nothing */;
  w = b[out];
  out++;
  /* consume item w */
}
```

As long as there is data available the consumer pops it off.

Producer/Consumer Problem



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Producer with Circular Buffer

```
producer:
while (true) {
  /* produce item v */
  while ((in + 1) % n == out)  /*
  do nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```

To do this we need to watch when pointers lap each other. We still need to ensure that only one process at a time.

Consumer with Circular Buffer

```
consumer:
while (true) {
  while (in == out)
   /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  /* consume item w */
}
```

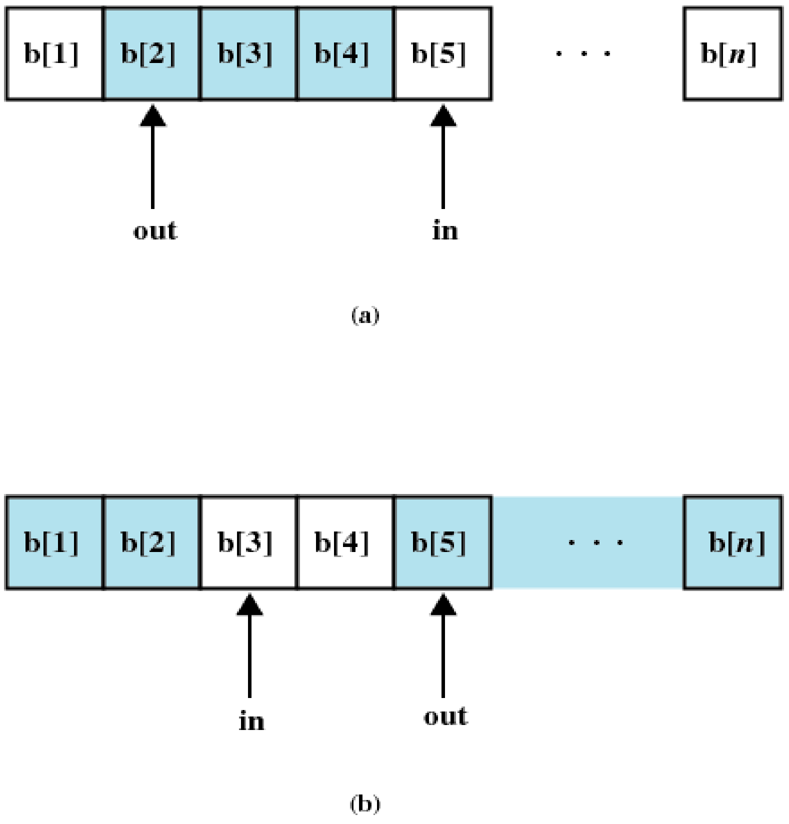


Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
     while (true)
     {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1)
               semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true)
     {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0)
               semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

We protect the append with a semWait(). Similar actions for the consumer. The consumer also needs to wait for data to be available (cant consume what is not there). In this consumer function we have a unprotected n when it is checked (if n == 0) which is bad, we should protect it as well.

Semaphores are used to protect a shared resources and for capacity control.

Table 5.4 Possible Scenario for the Program of Figure 5.9

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semiSignlaB(s)	1	–1	0

NOTE: White areas represent the critical section controlled by semaphore s.

Here the book tries to play through all combinations to watch for unprotected values.

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
     while (true)
     {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)
     {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Figure 5.10 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

We can fix this by protecting n or make a copy of it as a local variable that we test against. The producer here can't fuck with m since its local.

```
/* program  producerconsumer */
semaphore n = 0;
semaphore s = 1;
void producer()
{
        while (true)
        {
                produce();
                semWait(s);
                append();
                semSignal(s);
                semSignal(n);
        }
}
void consumer()
{
        while (true)
        {
                semWait(n);
                semWait(s);
                take();
                semSignal(s);
                consume();
        }
}
void main()
{
        parbegin (producer, consumer);
}
```

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

dWe can use a more general semaphore by taking advantage of the count variable.

Stepping into it:

1.

consumer()

semWait(n);

1) Can a process be inside a critical section if the semaphore value is negative? Yes. All that matters is the initial value.

2) Can a semSignal admit a blocked process inside a critical section if the semaphore value was -2 before semSignal was called? The -2 just indicates that there are currecntly 2 blocked processes. We can still wake up a processes from this queue.0

```
producer()
    produce()
    semWait(s)
    append()
    semSignal()
    semSignal()
    //loop back to start

consumer()
    semWait()
    semWait()
    take()
    semSignal(s)
    consume()
    //loop back to start
```

Say our ready process queue

ction s n Queue
start 1 0 RPQ[c1, c2, c3]
p1 executes three times 1 3 RPQ[c1, c2, c3, p1]
c1 exectutes but gets preempted 0 2 RPQ[c2, c3, p1, c1]
c2 exectutes -1 1 RPQ[c3, p1, c1] SEM[c2] is blocked
c3 executes -2 0 RPQ[p1, c1] SEM[c2, c3]
p1 executes -3 0 PRQ[c1] SEM[c2, c3, p1]
c1 executes -2 1 PRQ[c2] SEM[c3, p1]

With semaphores you can implement any concurrency pattern. Woooooo.

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1;
semaphore n= 0;
semaphore e= sizeofbuffer;
void producer()
{
     while (true)
     {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n)
     }
}
void consumer()
{
     while (true)
     {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.13 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

KNOW THE FUCK OUT OF SEMAPHORES, WILL BE ON QUIZ

Monitors

- Monitor is a software module
- Chief characteristics
  - Local data variables are accessible only by the monitor (=> shared data in monitor is “safe”)
  - Process enters monitor by invoking one of its procedures (=> controlled entry)
  - Only one process may be executing in the monitor at a time (=> mutex)
- Uses condition variables for signaling
- Unused signals are lost (!= semaphores)

Think of a container where you store all of your local variables that need to be safe data. In order to enter this procedure you must invoke a procedure and the monitor enforces that that procedure can only have one process running at a time. We use conditional variables to monitor this. We dont use a counter so unused signals are lost.

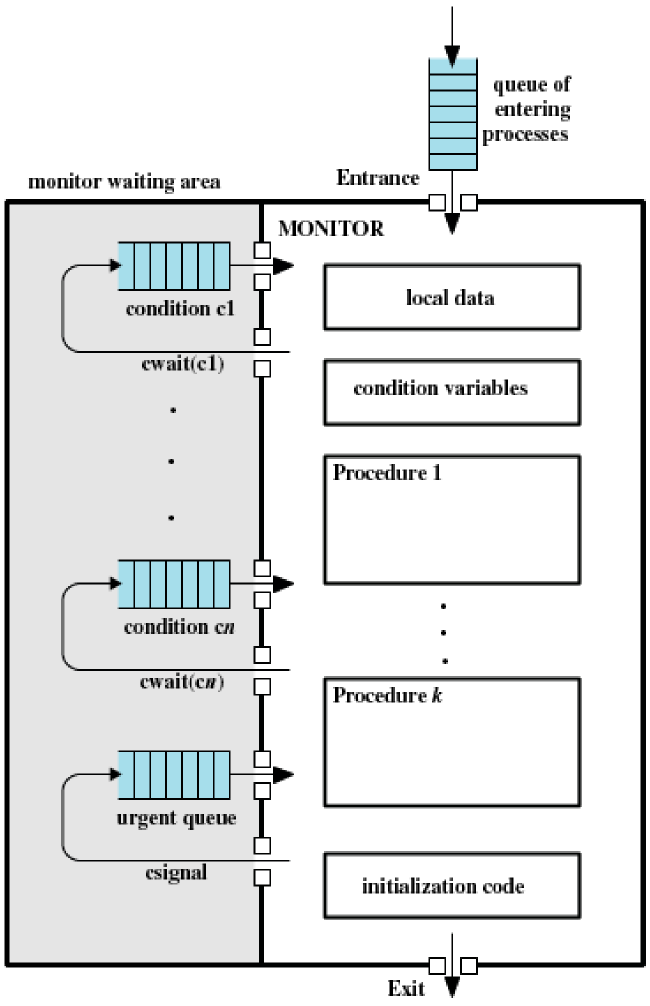


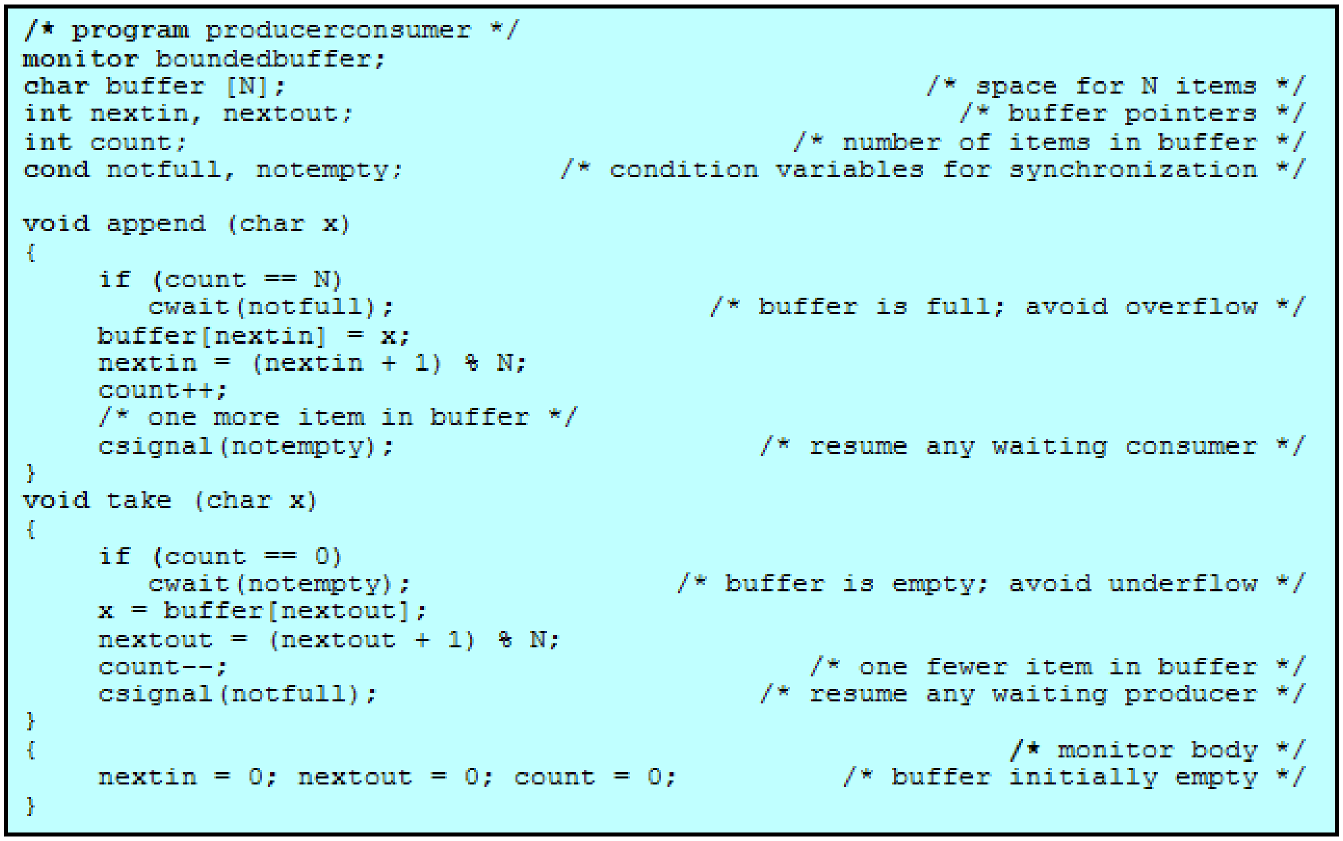
Figure 5.15 Structure of a Monitor

Think of the monitor as a object and that manages processes inside it.

```
void producer()
char x;
{
     while (true)
     {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true)
     {
       take(x);
       consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

We can hide all of the logic behind the monitors procedures to make our code cleaner.



```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                       /* space for N items */
int nextin, nextout;                                     /* buffer pointers */
int count;                                     /* number of items in buffer */
cond notfull, notempty;          /* condition variables for synchronization */

void append (char x)
{
    if (count == N)
       cwait(notfull);                    /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                       /* resume any waiting consumer */
}
void take (char x)
{
    if (count == 0)
       cwait(notempty);                 /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                                    /* one fewer item in buffer */
    csignal(notfull);                        /* resume any waiting producer */
}
{                                                           /* monitor body */
    nextin = 0; nextout = 0; count = 0;           /* buffer initially empty */
}
```

Here is the backend logic that is powering the monitor in the above interface.

```
append(x)
    if (count = n)
        cwait(notfull);
    bugger[nextin] = x;
    nexin = (nextin + 1) % n
    count ++
    signal(notempty)
take(x)
    if (count == 0)
        cwait(notempty)
    x = buffer[nextout]
    nextout = (nextout + 1) % n
    cout --
    cisignal(notfull)
```

First iteration:

- produce an item
- invoke append
- count is 0 so we dont wait
- we increase the count
- signal notempty

Second iteration

- produce an item
- invoke append
- count is 1 so we dont wait
- we increase the count
- signal notempty

Third iteration

- produce an item
- invoke append
- count is 3 so we dont wait
- we increase the count
- signal notempty

Fourth Iteration

- produce an item
- invoke append

count == n so we cwait

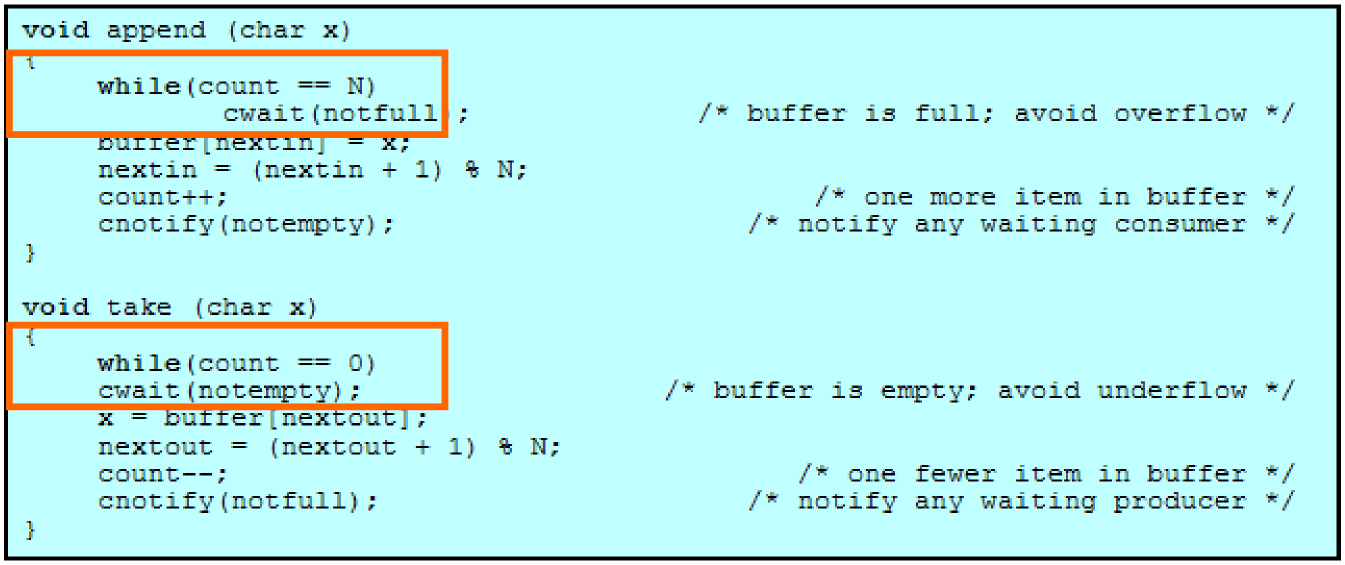
put it on the queue

Consumer first iteration

- consume an item
- invoke take
- count is 2 so we dont wait
- we decrease the count
- signal notfull

Keep iterating until we get count == 0

- cwait notempty



```
void append (char x)
{
     while(count == N)
               cwait(notfull);                    /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;                                          /* one more item in buffer */
     cnotify(notempty);                            /* notify any waiting consumer */
}

void take (char x)
{
     while(count == 0)
     cwait(notempty);                           /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                         /* one fewer item in buffer */
     cnotify(notfull);                             /* notify any waiting producer */
}
```

Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor

We replace the if with a while to avoid the count becoming inconsistent over multiple instances of the producer waking up. If you use monitor support this is the implementation that is used to date.

Message Passing

- Enforce mutual exclusion
- Exchange information

```
send (destination, message)
receive (source, message)
```

This is another method for communication and synchronization. We will have to do this in the project. We have messages and queues to store them in. The often comtain data about he program that sent them (pid mostly).

Synchronization

- Sender and receiver may or may not be blocking (waiting for message)
- Blocking send, blocking receive
  - Both sender and receiver are blocked until message is delivered
  - Called a rendezvous

Sender or receiver may be blocking depending on the implementation. Most common is a blocking send and blocking revice called the redezvous patternn.

Synchronization

- <u>Nonblocking send, blocking receive</u>
  - – Sender continues on
  - – Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - – Neither party is required to wait

A nonblocking send but blocking recieve. The sender just dumps the message onto the receiver who remains blocked untill it arrives. This is the one we want to do for our project.

We can also just not block at all.

Addressing

- Direct addressing
  - **Send** primitive includes a specific identifier of the destination process
  - **Receive** primitive could know ahead of time from which process a message is expected
  - **Receive** primitive could use source parameter to return a value when the receive operation has been performed

We refer to specific process that could be sending or revcieving (processes must be directly aware of each other).

Addressing

- Indirect addressing
  - Messages are sent to a shared data structure consisting of queues
  - Queues are called mailboxes
  - One process sends a message to the mailbox and the other process picks up the message from the mailbox

Implement a shared data storage where we send the data and another process retrieves it from there. We can make non blocking or send blocking based on this.

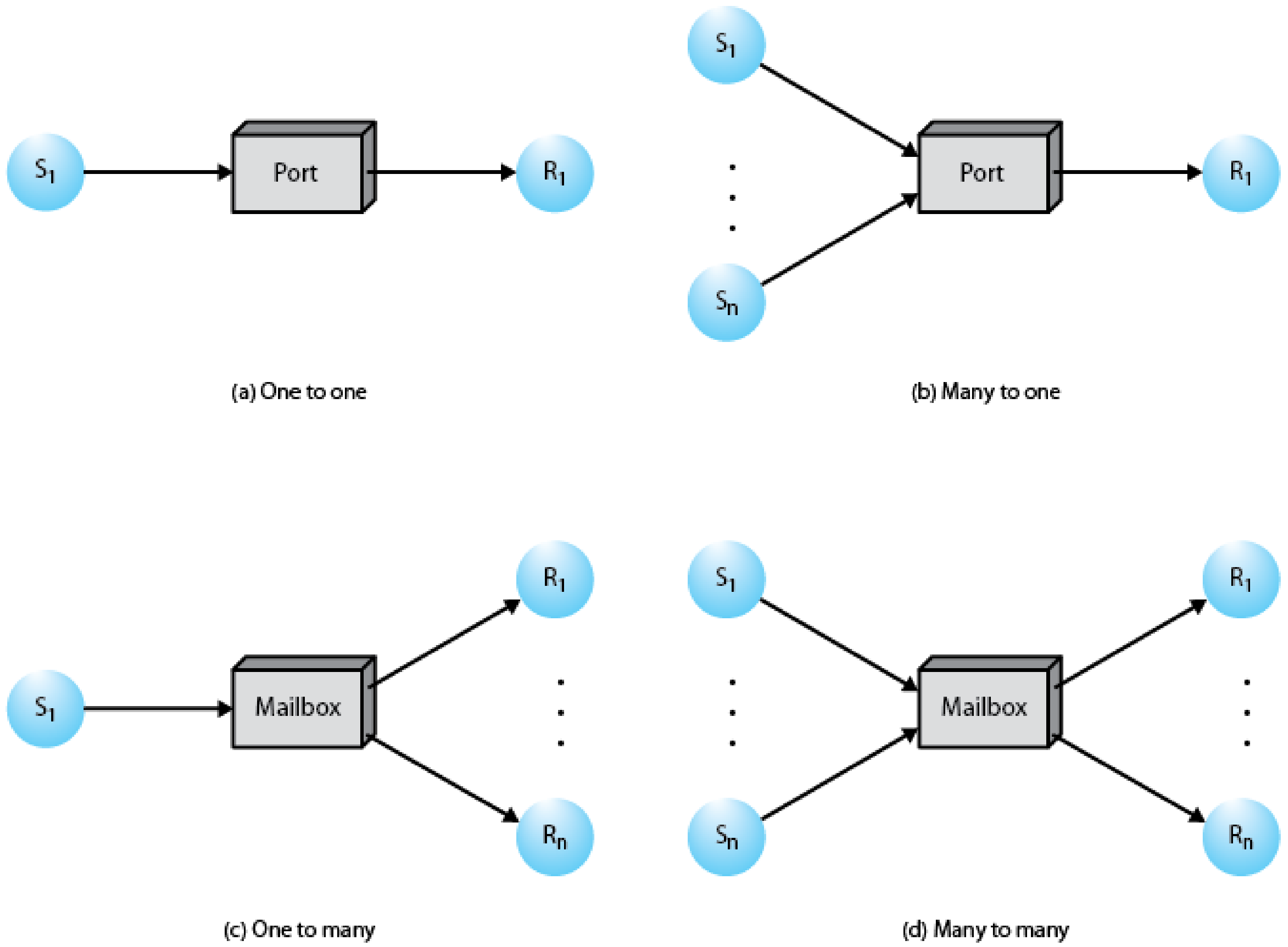
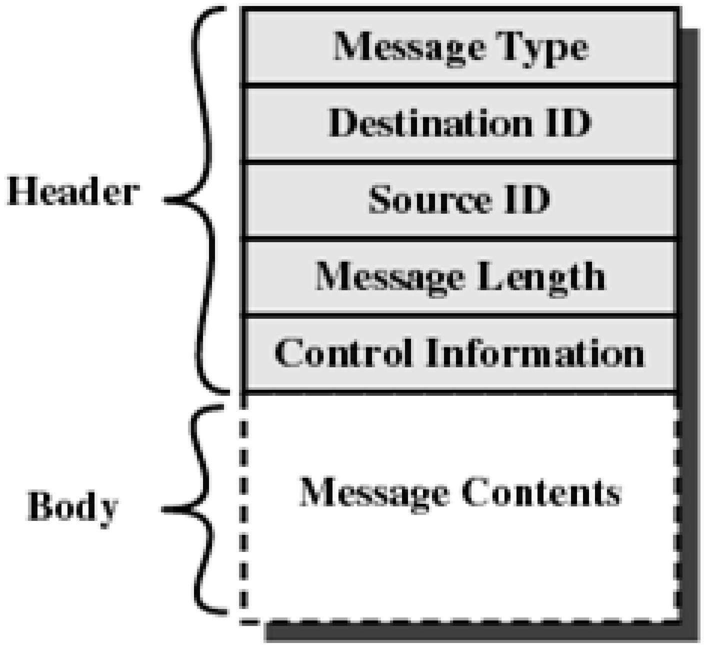


Figure 5.18 Indirect Process Communication

These are the options for message passing relationships. We currently use this just for synchronization but it is also used for netwroking and socketing. We assume all ports and mailboxes are in one system.

Message Format

… depends on the requirements of the OS (e.g., fixed-length vs. variable length messages)



We need some housekeeping data on the message. Destination id (if direct addressing), source id (who sent it), message length (how many bytes are valid), control information (data on the state of the message). Our project implementation should look very similar to this.

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true)
    {
     receive (mutex, msg);
     /* critical section   */;
     send (mutex, msg);
     /* remainder   */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages

Here is how we use messages to implement mutual exclusion. Uses indirect addressing. The mailbox is a critical resource. Create mailbox, send messae, begin a bunch of processes. Each process receives the message, does it stuff to the critical section, and sends the message along to the next one. This means that all but one process are blocked and waiting for the message always, so only one process can access the mailbox at a time. This is forces synchronization.

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true)
    {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true)
    {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
        send (mayproduce, null);
    parbegin (producer, consumer);
}
```

Figure 5.21   A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

We can do the same thing to implement the producer consumer pattern. Introduce a mailbox to produce items, another for consuming. To limit the buffer we put the capacity of the buffer number of messages in the queue.

```
producer()
    recieve(mayproduce)
    psmg = produce()
    send(mayconsume, pmsg)
    loop bacl

consumer()
    reviece(mayproduce, cmsg)
    consume(cmsg)
    send(mayproduce, null)
```

We have a N item count for our mayproduce queue (say N = 3).

```
1. mayproduce[m1, m2, m3]
producer executes
2. mayproduce[m2, m3], produce an item and send it to consumer mayconsume[pmsg1]
3. mayproduce[m3], mayconsume[pmsg1, psm2]
4. mayproduce[], mayconsume[pmsg1, psm2, pmsg3]
5. hit recieve on producer but there is nothing in the queue so producer is blocked
recieve on consumer, pick up item from queue and consume it and send message back
6. mayproduce[‘null’]  mayconsume [pmsg2, pmsg3]
7. mayproduce['null', ‘null’] mayconsume[pmsg3]
7. mayproduce['null', ‘null’, ‘null’] mayconsume[]
8. consumer gets blocked
```

There are some assumptions baked into this:

- there is not mutual exclusion baked into the mailbox, we solve this by making the mailbox part of the operating system. Since it is part of the kernel space we can protect it.

Readers/Writers Problem

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it

This looks a bit similar to producer consumer, but we have a one to many relationship. One writer but many readers are allowed at a time and the writer cannot allow a reader to access it.

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true)
    {
     semWait (x);
     readcount++;
     if (readcount == 1)
          semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0)
          semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true)
    {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
    }
```

Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority

The reader and writer code executes concurrently so we need to enforce this pattern on them. Make sure writer gets mutual exclusion on the buffer but the readers dont. Writer semaphore is initialized to one so this only allows one writer at a time. The reader needs to watch for the presence of the writer and not enter if so. If we are the first reader we need to make sure there is no writer waiting on the buffer. So first reader waits the writer semaphore and the last reader signals it. We also have a read count that needs to be protected which semaphore x watches.

Problem: While readers have the buffer they dominate the writer which may cause problems when you have many readers and only one writer. This can kill efficiency.

```
writer()
    semwait(wsem)
    write()
    semsignal(wsem)

reader()
    semwait(x)
        n++
    if(n == 1)
        semwait(wsem)
    semsignal(x)
    read()
    semwait(x)
    n--
    if(n==0)
        semsignal(wsem)
    semsignal(x)
```

Stepping through for 2

```
writer executes start: wsem = 1, n = 1, Qwsem = [], Qx = []
                 wsem = 0
R1 executes start:     wsem = 1, x = 0, Qwsem = [R1]
R2 executes start:     wsem = 1, x = -1, Qwsem = [R1] Qx = [R2]
writer executes semsignal: wsem = 0,         Qwsem = [], Qx = [R2]  Qrdy = [R1] //R1 got put on the ready
R1 executes, read:              x = 0,   Qsem=[],    Qx = [],   Qrdy = [R2]
```

```
/* program readersandwriters */
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0)
               semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true)
    {
     semWait (y);
          writecount++;
          if (writecount == 1)
               semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0)
               semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Figure 5. 23 A Solution to the Readers/Writers Problem Using Semaphores: Writers Have Priority

FAQ

- Example, why the order of operations matters. Specifically, why should semaphores for capacity management be outside the atomic buffer access. Example:

```
produce()
semWait(s)
semWait(e)
append()
semSignal(n)
semWait(s)
```

Hint: this is the wrong way!

```
semWait(n)
semWait(s)
append()
semWait(s)
semSignal(e)
consume()
```

- Now assume, we can only put one item into the buffer, and initially the buffer is full.

	producer	**consumer**	**s**	**n**	**e**	
			1	1	0	
1	produce()					
2	semWait(s)		0	1	0	
3	semWait(e)		0	0	0	<- producer blocked
4		semWait(n)	0	0	0	
5		semWait(s)	0	0	0	<- consumer blocked!

error was moving capacity mgt. inside critical section

- Tip: force buffer exhaustion to trigger cyclic dependency