Process Concurrency

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Readers/Writers Problem
- Monitors
- Message Passing

Principles of Concurrency

- Central themes of operating system design are all concerned with the management of processes and threads
  - Multiprogramming - management of multiple processes within a uniprocessor system
  - Multiprocessing - management of multiple processes within a multiprocessor system
  - Distributed Processing - management of multiple processes executing on multiple, distributed computer systems.
- Fundamental to all of these areas, and fundamental to OS design, is concurrency.
- Big Issue is Concurrency
  - Managing the interaction of all of these processes
- Concurrency is interleaving of processes in time to give the appearance of simultaneous execution

Interleaving and Overlapping Processes

- Processes may be interleaved on uniprocessors

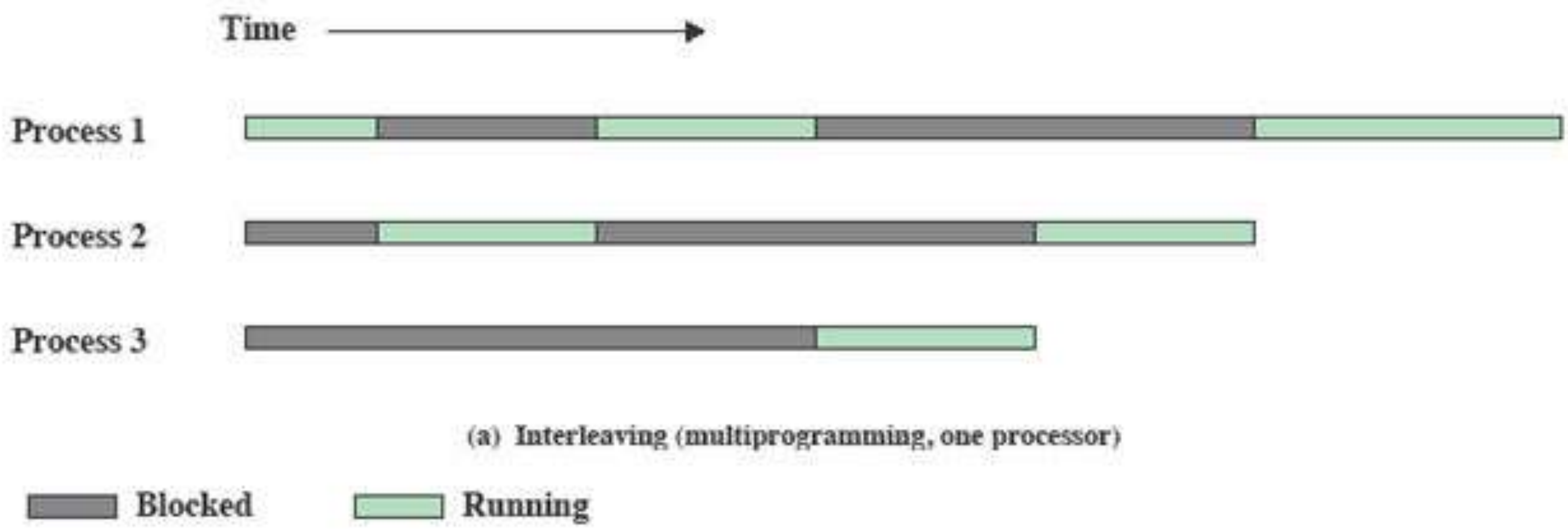
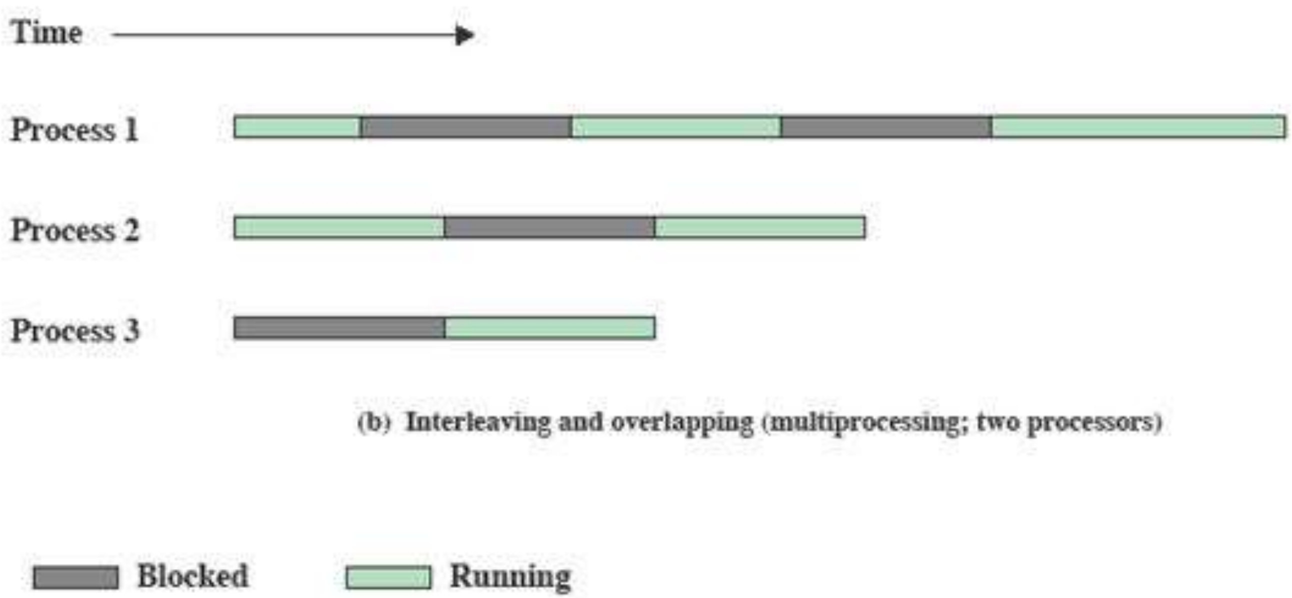


Figure 2.12 Multiprogramming and Multiprocessing

Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors



(b) Interleaving and overlapping (multiprocessing; two processors)

Figure 2.12 Multiprogramming and Multiprocessing

Difficulties of Concurrency

- Sharing of global resources
- Optimally managing the allocation of resources
- Difficult to locate programming errors.

A Simple Example

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

A Simple Example: On a Multiprocessor

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

Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then:
- P1 & P2 run on separate processors
- P1 enters echo first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes echo

Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items
  - They do so in a way where the final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.

Example of Race Condition

- b=1 ; c=2
  - – P3 – b=b+c
  - – P4 – c=b+c
  - – P3 -> P4 : b=3 c=5
  - – P4 -> P3 : b=4 c=3
- a=2
  - – P3 = a=a+1 =3
  - – P4 = a=a+2 =5

Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes
  - Allocate and de-allocate resources
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed

Critical Section Problem

- Each process has segment of code called as **Critical Section**
- CS avoids race condition
- In CS, process may change variables, update a table, write a file, etc.
- At any moment, only one process can execute in CS
- When one process is executing in CS, no other process is allowed to execute in its CS.
- Any solution to CS problem must satisfy the following requirements :
- **Mutual exclusion** : When one process is executing in CS, no other process is allowed to execute in its CS.
- **Progress** : If no process is executing in its CS, and if there are some processes that wish to enter CS, then one of these processes will get into CS.
- **Above mentioned both the conditions are mandatory conditions**
- **Bounded Waiting :** A process cannot be denied access to CS indefinitely. Each process must have a limited waiting time. It should not wait endlessly to access the critical section.

Do

{

Non CS code

Enter CS

Critical Section

Exit CS

Remainder section

} while(1)

- Mutual Exclusion
- Progress
- Bounded Wait

Competition among Processes for Resources

Three main control problems:

- Need for Mutual Exclusion
  - Critical sections
- Deadlock
- Starvation

Competition among Processes for Resources

```
/* PROCESS 1 */

void P1
{
 while (true) {
   /* preceding code */;
   entercritical (Ra);
   /* critical section */;
   exitcritical (Ra);
   /* following code */;
 }
}
```

```
/* PROCESS 2 */

void P2
{
 while (true) {
   /* preceding code */;
   entercritical (Ra);
   /* critical section */;
   exitcritical (Ra);
   /* following code */;
 }
}
```

• • •

```
/* PROCESS n */

void Pn
{
 while (true) {
   /* preceding code */;
   entercritical (Ra);
   /* critical section */;
   exitcritical (Ra);
   /* following code */;
 }
}
```

Figure 5.1 Illustration of Mutual Exclusion

Requirements for Mutual Exclusion

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

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support.
- Semaphores
- Readers/Writers Problem
- Monitors
- Message Passing

Disabling Interrupts

- Uniprocessors only allow interleaving
- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Will not work in multiprocessor architecture

Pseudo-Code

```
while (true) {
   /* disable interrupts */;
   /* critical section */;
   /* enable interrupts */;
   /* remainder */;
}
```

Special Machine Instructions

- Test & Set Instruction
- Compare & Swap Instruction
  - also called a “compare and exchange instruction”
- Exchange Instruction

Test & Set Instruction

```
boolean lock=false;
...
void critical()
{
while(lock==true)
{
//while another process is in CS
//do nothing (Busy Wait)
}
lock=true; //Entering CS
}

void leaveCritical()
{
lock=false; //set lock to 0 and allow other process in
}
```

Compare & Swap Instruction

```
int compare_and_swap (int *word,
  int testval, int newval)
{
   int oldval;
   oldval = *word;
   if (oldval == testval)
   *word = newval;
   return oldval;
}
```

Mutual Exclusion

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

}
```

(a) Compare and swap instruction

Exchange Instruction

void Swap (boolean &a , boolean &b)
{
boolean temp = a ;
a= b ;
b = temp ;
}

Implementation :

boolean lock = false ;

Process Pi :

key = true ;
while (key == true)
Swap (lock , key) ;
< critical section >
lock = false ;
< remainder section > }

Hardware Mutual Exclusion: 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

Hardware Mutual Exclusion: Disadvantages

- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Some process could indefinitely be denied access.
- Deadlock is possible

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Readers/Writers Problem
- Monitors
- Message Passing

Semaphore

- Semaphore:
  - An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - initialize,
  - Decrement (`Wait`)
  - Increment (S`ignal`)

Counting 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

Binary Semaphore Primitives

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
               /* place this process in s.queue */;
               /* block this process */;
     }
}
void semSignalB(semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
               /* remove a process P from s.queue */;
               /* place process P on ready list */;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Mutex

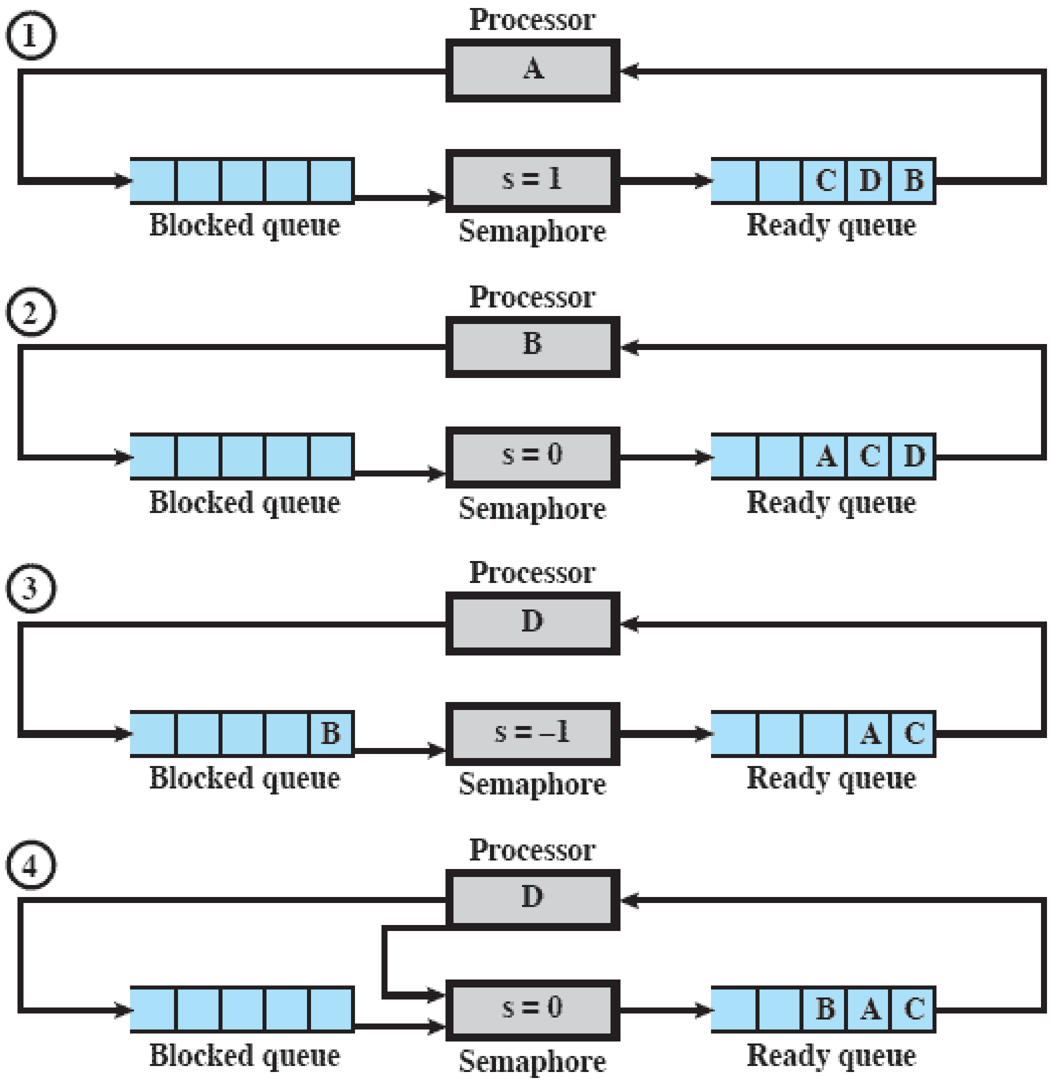
- Some Binary semaphores are called as mutex or mutex locks
- A process that locks the mutex must be the one to unlock it.

```
do
{
wait(mutex);
//Critical Section
signal(mutex);
//exit critical section
remainder section
}while(true);
```

Strong/Weak Semaphore

- A queue is used to hold processes waiting on the semaphore
  - In what order are processes removed from the queue?
- ***Strong Semaphores*** use FIFO
- ***Weak Semaphores*** don’t specify the order of removal from the queue

Example of Strong Semaphore Mechanism



Example of Semaphore Mechanism

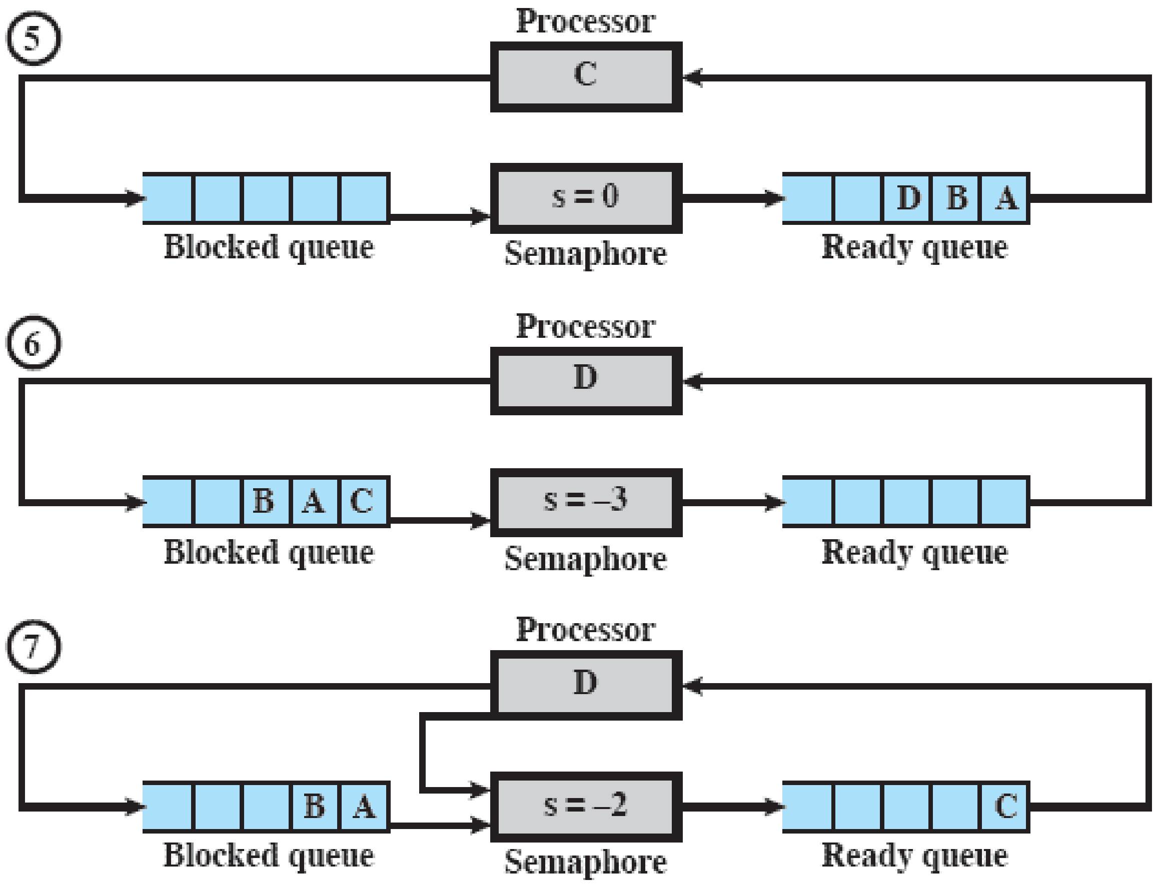


Figure 5.5 Example of Semaphore Mechanism

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 Using Semaphore

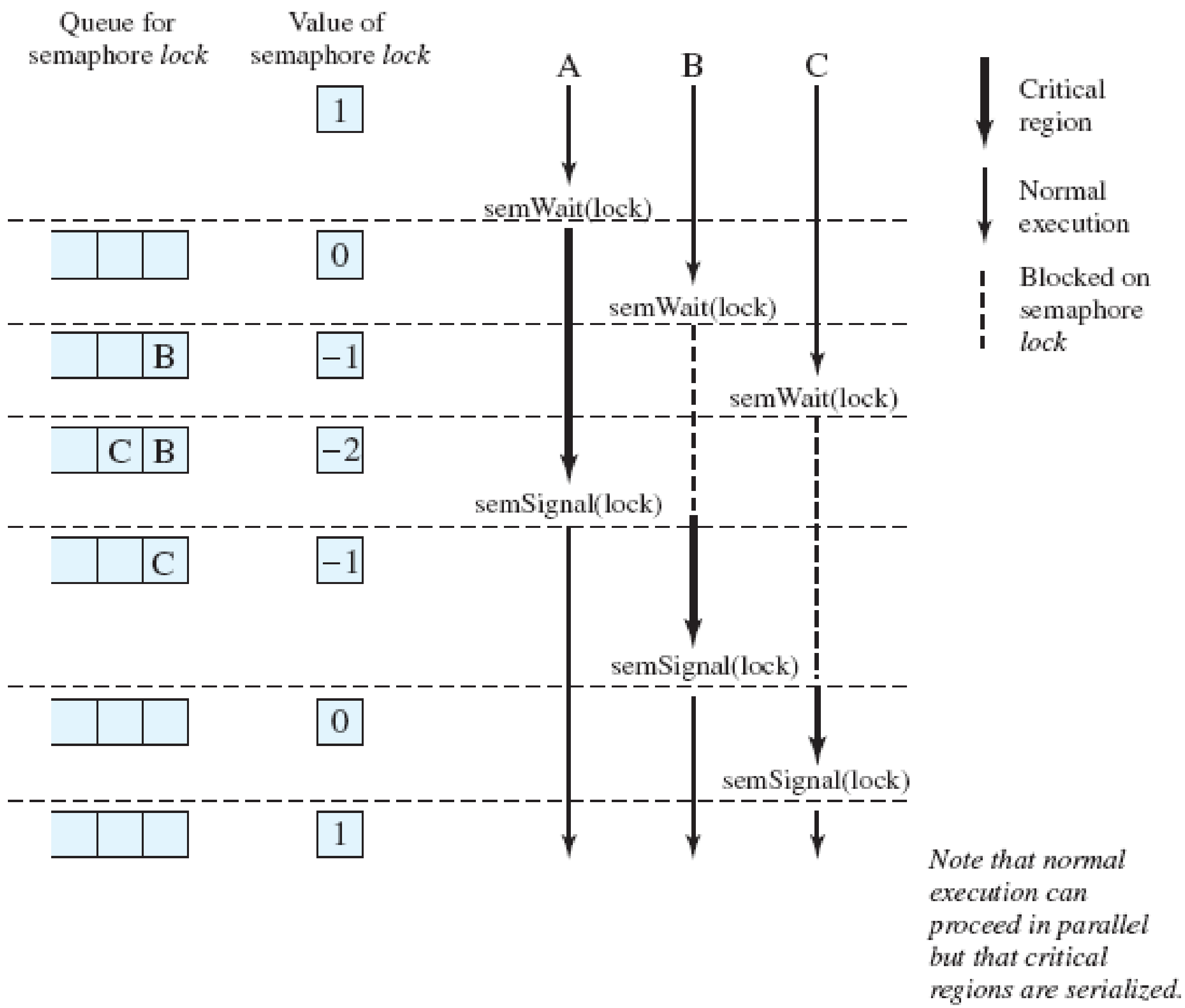


Figure 5.7 **Processes Accessing Shared Data Protected by a Semaphore**

Producer/Consumer Problem

- General Situation:
  - 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 Problem:
  - Ensure that the Producer can’t add data into full buffer and consumer can’t remove data from empty buffer

Producer/Consumer Problem

```
semaphore mutex=1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer()
{
    while (true)
    {
        item = produceItem();
        wait(emptyCount);
        wait(mutex);
        putItemIntoBuffer(item);
        signal(mutex);
        signal(fillCount);
    }
}
```

Producer/Consumer Problem

```
procedure consumer()
{
    while (true)
    {
        wait(fillCount);
        wait(mutex);
        item = removeItemFromBuffer();
        signal(mutex);
        signal(emptyCount);
        consumeItem(item);
    }
}
```

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Readers/Writers Problem
- Monitors
- Message Passing

Readers/Writers Problem

- A data area is shared among many processes
  - Some processes only read the data area, some only write to the area
- Conditions to satisfy:
  1. Multiple readers may read the file at once.
  2. Only one writer at a time may write
  3. If a writer is writing to the file, no reader may read it.

```
int readcnt=0;
Semaphore mutex=1;
Semaphore wrt=1;
```

// Reader process :

```
do {
      // Reader wants to enter the critical section
   wait(mutex);

   // The number of readers has now increased by 1
   readcnt++;

   // there is atleast one reader in the critical section
   // this ensure no writer can enter if there is even
one reader
   // thus we give preference to readers here
   if (readcnt==1)
      wait(wrt);
 signal(mutex);

   // other readers can enter while this current reader
is inside
   // the critical section
// current reader performs reading here
   wait(mutex);   // a reader wants to leave
   readcnt--;
// that is, no reader is left in the critical section,
   if (readcnt == 0)
      signal(wrt);        // writers can enter

   signal(mutex); // reader leaves

} while(true);
```

// Writer process :

```
do {
    // writer requests for critical section
    wait(wrt);
   // critical section
    // performs the write
    // leaves the critical section
    signal(wrt);
} while(true);
```

Sleeping Barber Problem

Sleeping Barber Problem

- In computer science, the sleeping barber problem is a classic inter-process communication and synchronization problem between multiple operating system processes.
- The problem is analogous to that of keeping a barber working when there are customers, resting when there are none, and doing so in an orderly manner.
- Problem : The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.
  - If there is no customer, then the barber sleeps in his own chair.
  - When a customer arrives, he has to wake up the barber.
  - If there are many customers and the barber is cutting a customer’s hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.

Sleeping Barber Problem

```
Semaphore barberReady = 0
Semaphore accessWRSeats = 1     # if 1, the number of seats in the waiting room can be incremented or decremented
Semaphore custReady = 0         # the number of customers currently in the waiting room, ready to be served
int numberOfFreeWRSeats = N     # total number of seats in the waiting room

def Barber():
  while true:                   # Run in an infinite loop.
    wait(custReady)             # Try to acquire a customer - if none is available, go to sleep.
    wait(accessWRSeats)         # Awake - try to get access to modify # of available seats, otherwise sleep.
    numberOfFreeWRSeats += 1    # One waiting room chair becomes free.
    signal(barberReady)         # I am ready to cut.
    signal(accessWRSeats)       # Don't need the lock on the chairs anymore.
    # (Cut hair here.)
```

Sleeping Barber Problem

```
def Customer():
  while true:                  # Run in an infinite loop to simulate multiple customers.
    wait(accessWRSeats)        # Try to get access to the waiting room chairs.
    if numberOfFreeWRSeats > 0: # If there are any free seats:
      numberOfFreeWRSeats -= 1  #   sit down in a chair
      signal(custReady)         #   notify the barber, who's waiting until there is a customer
      signal(accessWRSeats)     #   don't need to lock the chairs anymore
      wait(barberReady)         #   wait until the barber is ready
      # (Have hair cut here.)
    else:                      # otherwise, there are no free seats; tough luck --
      signal(accessWRSeats)     #   but don't forget to release the lock on the seats!
      # (Leave without a haircut.)
```

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Readers/Writers Problem
- Monitors
- Message Passing

Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus,
  - Modula-2, Modula-3, and Java.

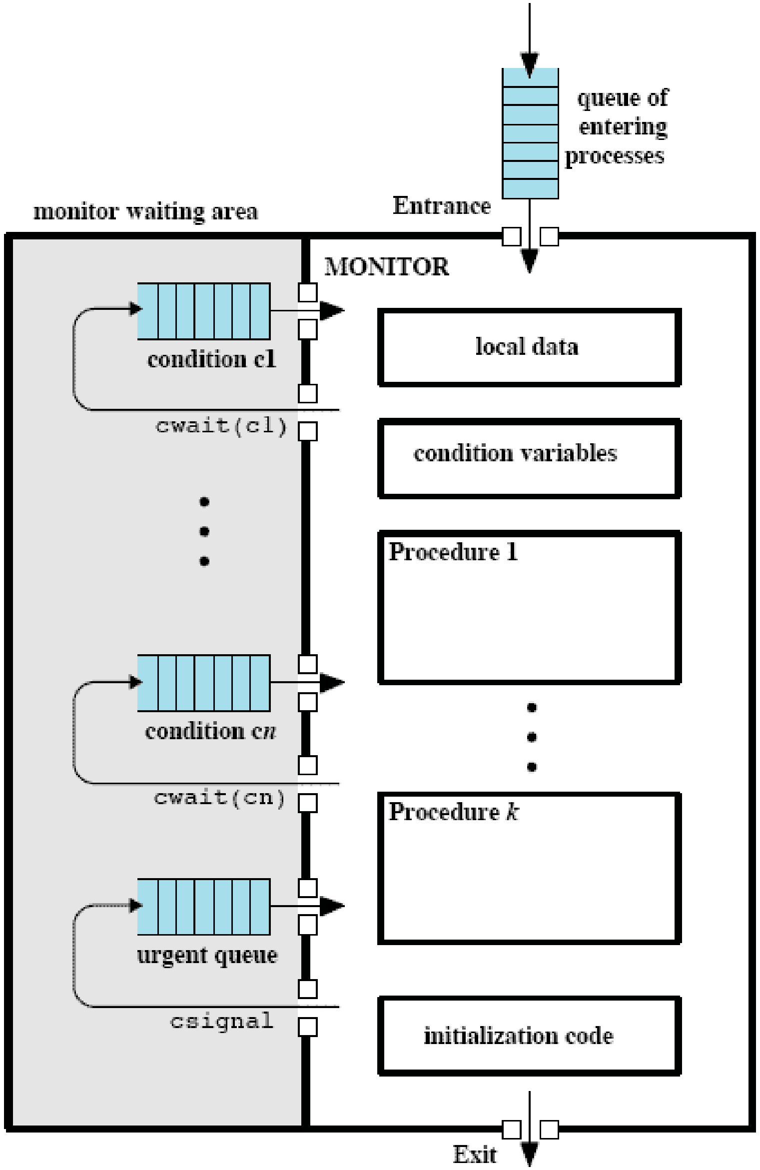
Chief characteristics

- Local data variables are accessible only by the monitor
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time

Synchronization

- Synchronisation achieved by **condition variables** within a monitor
  - only accessible by the monitor.
- Monitor Functions:
  - Cwait(c): Suspend execution of the calling process on condition *c*
  - Csignal(c) Resume execution of some process blocked after a cwait on the same condition

Structure of a Monitor



Bounded Buffer Solution Using Monitor

```
/* 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 */
}
```

Solution Using Monitor

```
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);
}
```

Roadmap

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Readers/Writers Problem
- Monitors

→ Message Passing

Process Interaction

- When processes interact with one another, two fundamental requirements must be satisfied:
  - synchronization and
  - communication.
- Message Passing is one solution to the second requirement
  - Added bonus: It works with shared memory *and* with distributed systems

Message Passing

- The actual function of message passing is normally provided in the form of a pair of primitives:
- send (destination, message)
- receive (source, message)

Synchronization

- Communication requires synchronization
  - Sender must send before receiver can receive
- What happens to a process after it issues a send or receive primitive?
  - 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
- Known as a *rendezvous*
- Allows for tight synchronization between processes.

Non-blocking Send

- More natural for many concurrent programming tasks.
- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither party is required to wait

Addressing

- Sendin process need to be able to specify which process should receive the message
  - Direct addressing
  - Indirect Addressing

Direct Addressing

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

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

Indirect Process Communication

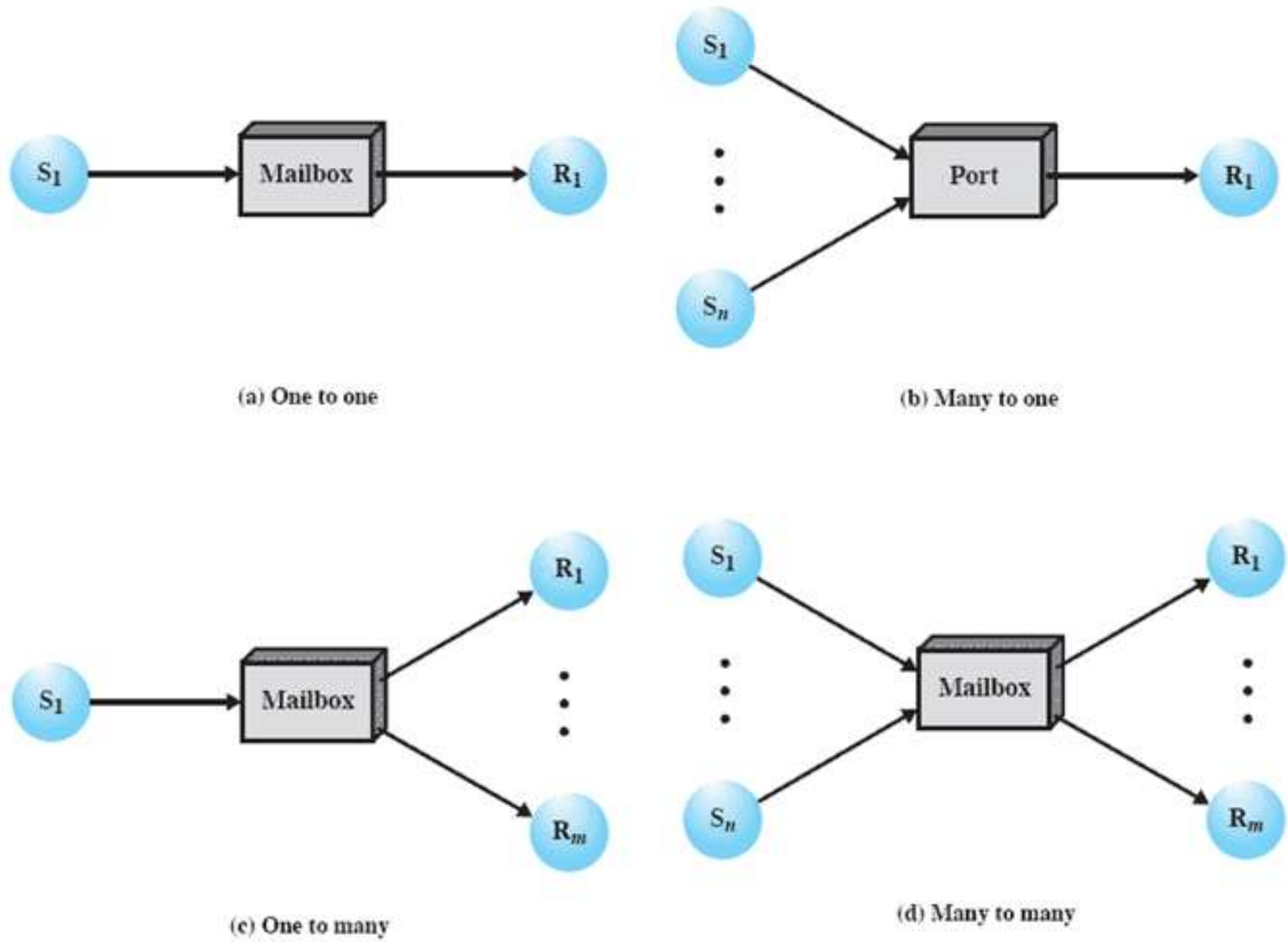


Figure 5.18 Indirect Process Communication

General Message Format

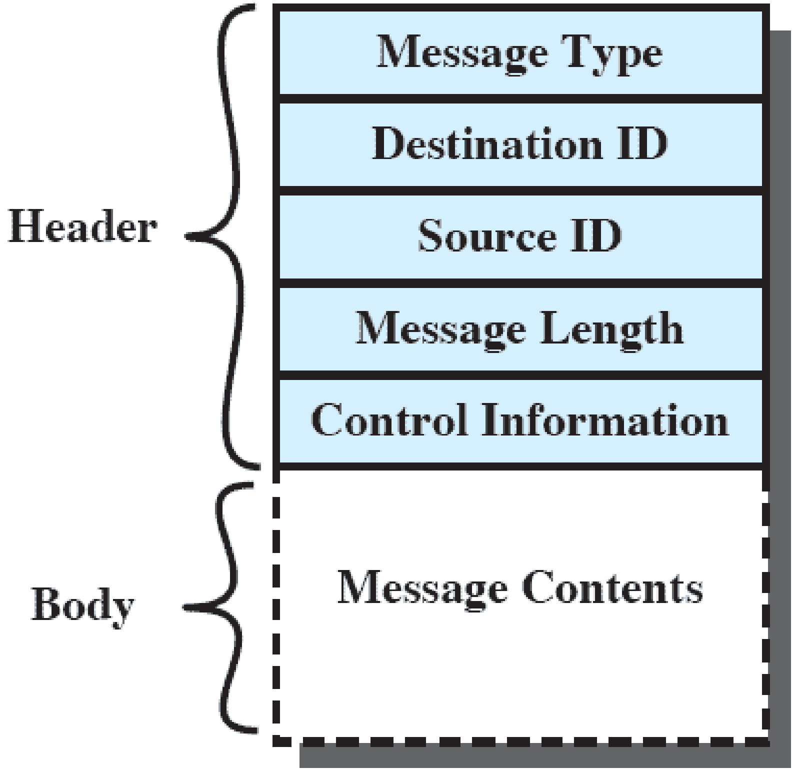


Figure 5.19 General Message Format

Mutual Exclusion Using Messages

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

Figure 5.20 Mutual Exclusion Using Messages

Producer/Consumer Messages

```
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);
}
```

Some Key Terms Related to Concurrency

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must 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 states 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.