Concurrency: Mutual Exclusion and Synchronization

Chapter 5

Contents

- Principles of concurrency
- Mutual exclusion : software approach
- Mutual exclusion : hardware approach
- Semaphore
- Monitors
- Message passing
- Readers/Writers problem

Central theme of OS design

- 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

Concurrency arises in ....

- Multiple applications at the same time
  - Multiprogramming
- Structured application
  - Application can be a set of concurrent processes
- OS structure
  - Operating systems are often implemented as a set of processes or threads

Difficulties with Concurrency

- Sharing of global resources
  - if two processes make use of the shared variable, then the order of access is critical
- Management of allocation of resources
  - may lead to a deadlock
- Programming errors difficult to locate
  - results are typically not deterministic and reproducible

An Example(uniprocessor)

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

An Example(multiprocessor)

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

Lesson to be learned

- It is necessary to protect the shared global variables
  - the only way to do that is to control the code that accesses the variable
  - only one process at a time may access the shared variable

Operating System Concerns

- Keep track of active processes
- Allocate and deallocate resources
  - processor time
  - memory
  - files
  - I/O devices
- Protect data and resources of each process
- Results of a process must be independent of the speed at which the execution is carried out relative to the speed of other concurrent processes

Process Interaction

- Processes unaware of each other
  - independent processes that are not intended to work together
- Processes indirectly aware of each other
  - processes are not necessarily aware of each other but they share access to some object
- Process directly aware of each other
  - processes are able to communicate directly with each other

Competition Among Processes for Resources

- Execution of one process may affect the behavior of competing processes
  - If two processes wish access to a single resource, one process will be allocated the resource and the other will have to wait
- It is possible that the blocked process will never get access to the resource and never terminate

Potential Problems

- Mutual Exclusion
  - critical sections
    - portion of a program that accesses the shared resource
      - only one process at a time is allowed in it
    - example
      - only one process at a time is allowed to send command to the printer
- Deadlock
- Starvation

Cooperation Among Processes by Sharing

- Processes use and update shared data such as shared variables, files, and data bases
  - Writing must be mutually exclusive
- Critical sections are used to provide data integrity

P1:

```
a = a +1;
b = b + 1;
```

P2:

```
b = 2 * b;
a = 2 * a;
```

```
a = a +1;
b = 2 * b;
b = b + 1;
a = 2 * a;
```

```
/* program mutualexclusion */
const int n = /* number of processes */ ;

void P(int i)
{
    while (true)
    {
        entercritical (i);
        /* critical section */;
        exitcritical (i);
        /* remainder */;
    }
}
void main( )
{
    parbegin (P(R1 ), P(R2 ), . . ., P(Rn));
}
```

Figure 5.1 Mutual Exclusion

Cooperation Among Processes by Communication

- Communication provides a way to synchronize, or coordinate the various activities
- Possible to have deadlock
  - each process waiting for a message from the other process
- Possible to have starvation
  - two processes sending message to each other repeatedly while another process is waiting

Requirements for Mutual Exclusion

- Only one process at a time is allowed into its critical section
- A process that halts in its non-critical section must do so without interfering with other processes
- A process requiring access to a critical section must not be delayed indefinitely: no deadlock or starvation

Requirements for Mutual Exclusion

- A process must not be delayed access to a critical section when there is no other process in it
- No assumptions are made about relative process speeds or number of processors
- A process remains inside its critical section for a finite time only

Mutual Exclusion: Software Approaches

- Dekker's algorithm
  - an algorithm for mutual exclusion for two processes
- Let's develop the solution in stages

Busy-Waiting First Attempt

/* PROCESS 0 /* • • **while** (turn != 0) /* do nothing */ **;** /* critical section*/; turn = 1; •	**/* PROCESS 1 */** • • **while** (turn != 1) /* do nothing ***/;** /* critical section*/; turn = 0; •

Busy-Waiting First Attempt

- Guarantees mutual exclusion
- Processes must strictly alternate in use of their critical sections
- If one process fails, the other process is permanently blocked
  - Each process should have its own key to the critical section so if one process is eliminated, the other can still access its critical section

Busy-Waiting Second Attempt

/* PROCESS 0 */ • • **while** (flag[1]) /* do nothing */; flag[0] = true; /*critical section*/; flag[0] = false; •	**/* PROCESS 1 */** • • **while** (flag[0]) /* do nothing */; flag[1] = true; /* critical section*/; flag[1] = false; •

Busy-Waiting Second Attempt

- Each process can examine the other's status but cannot alter it
- When a process wants to enter the critical section, it checks the other process first
- If no process is in the critical section, it sets its status for the critical section
- This method does not guarantee mutual exclusion
  - Each process can check the flags and then proceed to enter the critical section at the same time

Busy-Waiting Third Attempt

/* PROCESS 0 */	/* PROCESS 1 */
•	•
•	•
flag[0] = true;	flag[1] = true;
while (flag[1])	**while** (flag[0])
/* do nothing */;	/* do nothing */;
/* critical section*/;	/* critical section*/;
flag[0] = false;	flag[1] = false;
•	•

Busy-Waiting Third Attempt

- Set flag to enter critical section before checking other processes
- If another process is in the critical section when the flag is set, the process is blocked until the other process releases the critical section
- Deadlock is possible when two process set their flags to enter the critical section
  - Now each process must wait for the other process to release the critical section

Busy-Waiting Fourth Attempt

```
/* PROCESS 0 */
•
•
flag[0] = true;
while (flag[1])
{
    flag[0] = false;
    /*delay */;
    flag[0] = true;
}
/*critical section*/;
flag[0] = false;
•
```

```
/* PROCESS 1 */
•
•
flag[1] = true;
while (flag[0])
{
    flag[1] = false;
    /*delay */;
    flag[1] = true;
}
/* critical section*/;
flag[1] = false;
•
```

Busy-Waiting Fourth Attempt

- A process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag
- Other processes are checked. If they are in the critical region, the flag is reset and later set to indicate desire to enter the critical region. This is repeated until the process can enter the critical region.

Busy-Waiting Fourth Attempt

- It is possible for each process to set their flag, check other processes, and reset their flags
  - this sequence could be extended indefinitely, and neither process could enter its critical section
  - this condition is referred to as livelock

Busy-Waiting Correct Solution

- Each process gets a turn at the critical section
- If a process wants the critical section, it sets its flag and may have to wait for its turn

Busy-Waiting Correct Solution(Dekker's)

```
boolean flag [2];
int turn;
void P0( )
{
    while (true)
    {
        flag [0] = true;
        while (flag [1])
          if (turn == 1)
          {
            flag [0] = false;
            while (turn == 1)
            /* do nothing */;
            flag [0] = true;
          }
        /* critical section */;
        turn = 1;
        flag [0] = false;
        /* remainder */;
    }
}
```

```
void P1( )
{
    while (true)
    {
        flag [1] = true;
        while (flag [0])
          if (turn == 0)
          {
            flag [1] = false;
            while (turn == 0)
             /* do nothing */;
            flag [1] = true;
          }
        /* critical section */;
        turn = 0;
        flag [1] = false;
        /* remainder */;
    }
}
```

```
void main ( )
{
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    parbegin (P0, P1);
}
```

Busy-Waiting Correct Solution(Peterson's)

```
boolean flag [2];
int turn;
void P0( )
{
    while (true)
    {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1)
          /* do nothing */;
        /* critical section */;
        flag [0] = false;
        /* remainder */;
    }
}
```

```
void P1( )
{
    while (true)
    {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0)
          /* do nothing */;
        /* critical section */;
        flag [1] = false;
        /* remainder */
    }
}
```

```
void main( )
{
    flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}
```

Mutual Exclusion - Interrupt Disabling

- A process runs until it invokes an operating system service or until it is interrupted
  - So disabling interrupts can guarantee mutual exclusion
  - But processor is limited in its ability to interleave programs
    - Efficiency of execution could be noticeably degraded
- Not working for multiprocessing
  - disabling interrupts on one processor will not guarantee mutual exclusion

Mutual Exclusion - Interrupt Disabling

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

Mutual Exclusion - Using Machine Instructions

- Special Machine Instructions
  - Carry out two actions atomically
    - performed in a single instruction cycle
    - not interrupted in the middle of execution
  - Test and Set instruction
  - Exchange instruction

Mutual Exclusion - Using Machine Instructions

- Test and Set Instruction

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

Mutual Exclusion - Using Machine Instructions

- Exchange Instruction

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

```
/* 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));
}
```

(a) Test and set instruction

```
/* 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));
}
```

(b) Exchange instruction

Figure 5.5 Hardware Support for Mutual Exclusion

Mutual Exclusion - Using 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

Mutual Exclusion - Using 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 preempts, the higher priority process will obtain the processor to wait for the critical region

Semaphores(by Dijkstra)

- Special variable called a semaphore is used for signaling
  - to transmit a signal, execute signal(s)
  - to receive a signal, execute wait(s)
    - If a process is waiting for a signal, it is suspended until that signal is sent
- Wait and Signal operations cannot be interrupted
- A queue is used to hold processes waiting on the semaphore

Semaphores(by Dijkstra)

- Operations defined on a semaphore
  - semaphore may be initialized to a nonnegative value
  - wait operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked
  - signal operation increments the semaphore value. If the value is not positive, then a process blocked by a wait operation is unblocked

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

Figure 5.6 A Definition of Semaphore Primitives

```
struct binary_semaphore {
      enum (zero, one) value;
      queueType queue;
};
void waitB(binary_semaphore s)
{
      if (s.value == 1)
          s.value = 0;
      else
      {
          place this process in s.queue;
          block this process;
      }
}
void signalB(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 5.7 A Definition of Binary Semaphore Primitives

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

Figure 5.9 Mutual Exclusion Using Semaphores

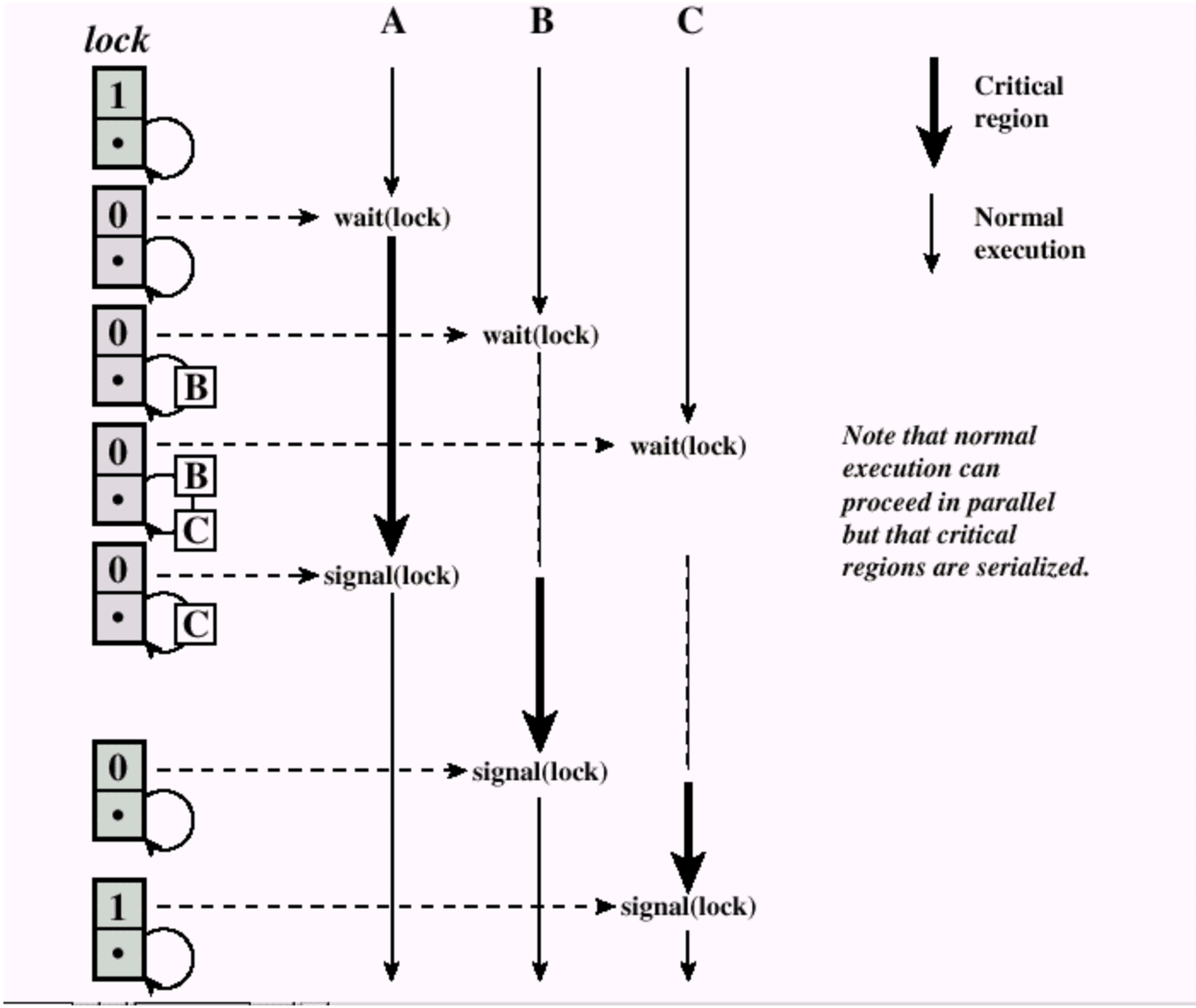


Figure 5.10 Processes accessing shared data protected by a semaphore

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
- Two semaphores are used
  - one to represent the amount of items in the buffer
  - one to signal that it is all right to use the buffer

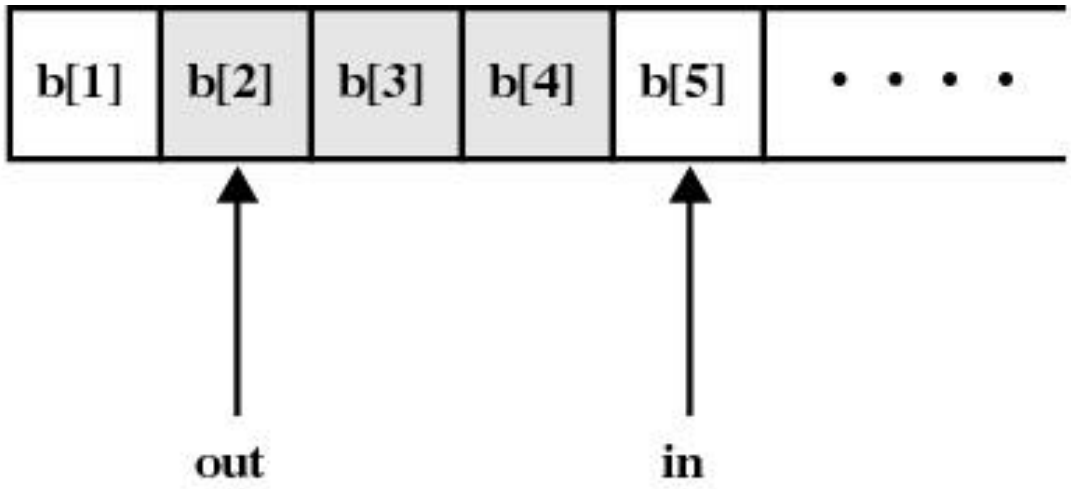
Producer

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

Consumer

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

Infinite Buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
     while (true)
     {
          produce();
          waitB(s);
          append();
          n++;
          if (n==1)
               signalB(delay);
          signalB(s);
     }
}
```

```
void consumer()
{
     waitB(delay);
     while (true)
     {
          waitB(s);
          take();
          n--;
          signalB(s);
          consume();
          if (n==0)
               waitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Figure 5.12 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

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

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
void producer()
{
     while (true)
     {
          produce();
          waitB(s);
          append();
          n++;
          if (n==1) signalB(delay);
          signalB(s);
     }
}
```

```
void consumer()
{
     int m; /* a local variable */
     waitB(delay);
     while (true)
     {
          waitB(s);
          take();
          n--;
          m = n;
          signalB(s);
          consume();
          if (m==0) waitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Figure 5.13 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/* program producerconsumer */
semaphore n = 0;
semaphore s = 1;
void producer()
{
     while (true)
     {
          produce();
          wait(s);
          append();
          signal(s);
          signal(n);
     }
}

void consumer()
{
     while (true)
     {
          wait(n);
          wait(s);
          take();
          signal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

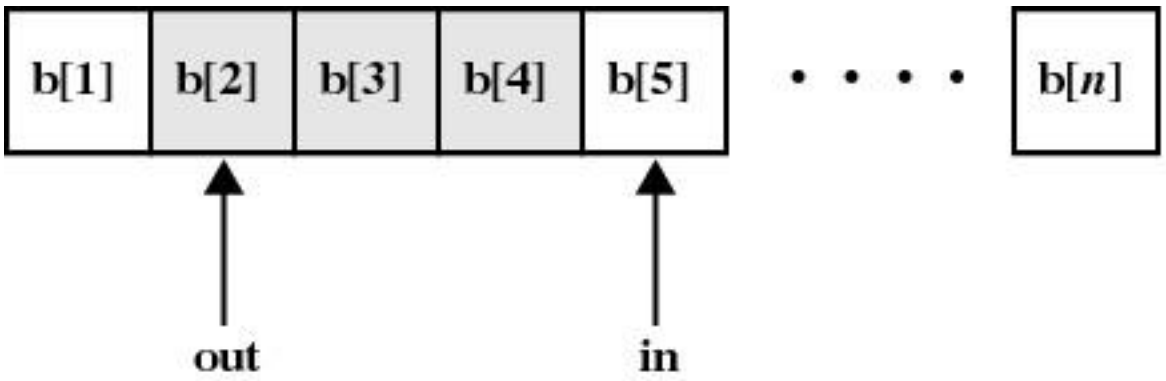
Figure 5.14 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

Producer with Circular Buffer

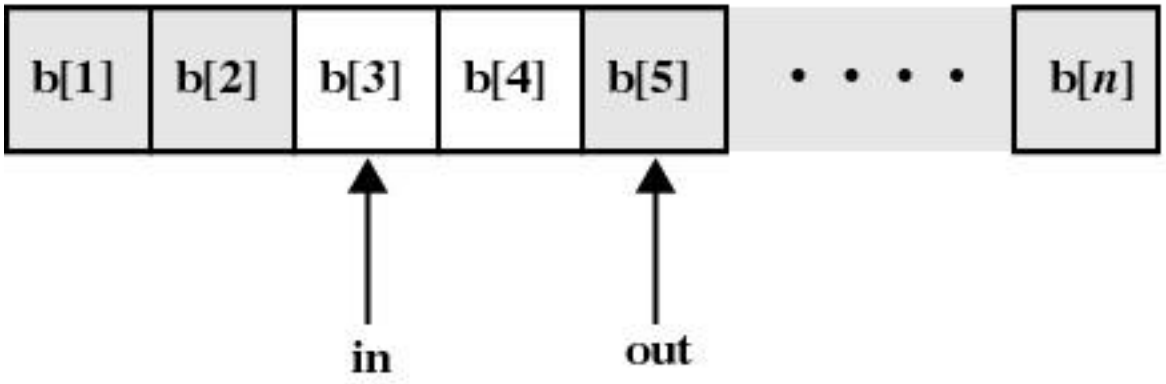
```
producer:
while (true) {
  /* produce item v */
  while ((in + 1) % n == out)  /*
  do nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```

Consumer with Circular Buffer

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



(a)



(b)

Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1;
semaphore n= 0;
semaphore e= sizeofbuffer;
void producer()
{
     while (true)
     {
          produce();
          wait(e);
          wait(s);
          append();
          signal(s);
          signal(n)
     }
}
```

```
void consumer()
{
     while (true)
     {
          wait(n);
          wait(s);
          take();
          signal(s);
          signal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

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

Implementation of Semaphores

- Wait and Signal should be implemented as atomic primitives
  - can be implemented as hardware instructions
  - software schemes can be used
    - this would entail a substantial processing overhead
  - hardware-supported schemes can be used

```
wait(s)
{
    while (!testset(s.flag))
        /* do nothing */;
    s.count--;
    if (s.count < 0)
    {
        place this process in s.queue;
        block this process (must also set s.flag to 0)
    }
    else
        s.flag = 0;
}
signal(s)
{
    while (!testset(s.flag))
        /* do nothing */;
    s.count++;
    if (s.count <= 0)
    {
        remove a process P from s.queue;
        place process P on ready list
    }
    s.flag = 0;
}
```

(a) Testset Instruction

```
wait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0)
    {
        place this process in s.queue;
        block this process and allow interrupts
    }
    else
        allow interrupts;
}
signal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0)
    {
        remove a process P from s.queue;
        place process P on ready list
    }
    allow interrupts;
}
```

(b) Interrupts

Figure 5.17 Two Possible Implementations of Semaphores

Barbershop Problem

- Problem
  - 3 chairs, 3 barbers, and a waiting area
  - fire code limits the total number of customers in the shop to 20
  - barbershop will eventually process 50 customers
  - a customer will not enter the shop if it is filled to capacity

- once inside, the customer takes a seat on the sofa or stands if the sofa is filled
- when a barber is free, the customer that has been on the sofa the longest is served and if there are any standing customers, the one that has been in the shop the longest takes a seat on the sofa
- when a customer's haircut is finished, any barber can accept payment, but because there is only one cash register, payment is accepted for one customer at a time

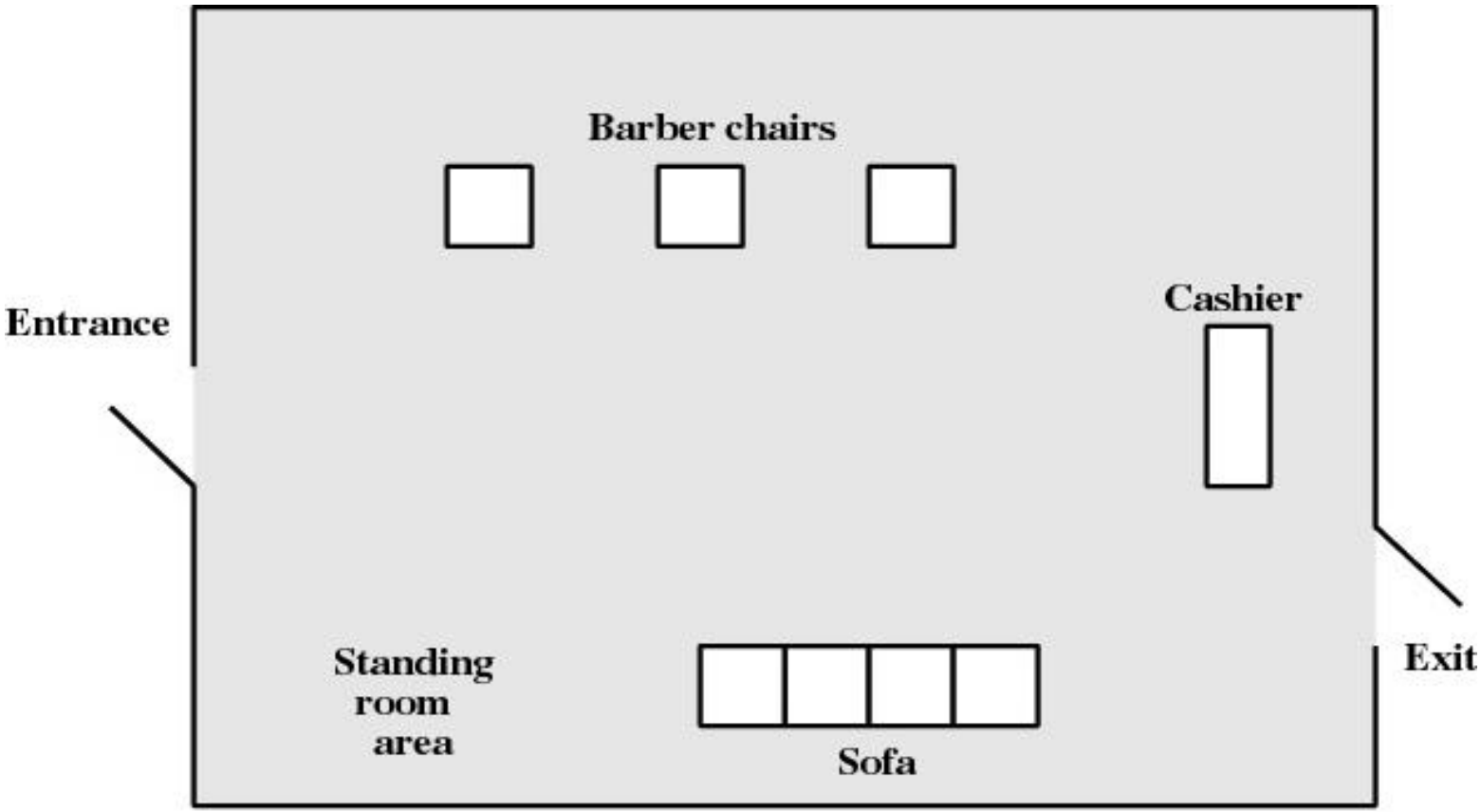


Figure 5.18 The Barbershop

```
/* program barbershop1 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3;
semaphore coord = 3;
semaphore cust_ready = 0, finished = 0, leave_b_chair = 0, payment= 0, receipt = 0;
void customer ()
{
    wait(max_capacity);
    enter_shop();
    wait(sofa);
    sit_on_sofa();
    wait(barber_chair);
    get_up_from_sofa();
    signal(sofa);
    sit_in_barber_chair;
    signal(cust_ready);
    wait(finished);
    leave_barber_chair();
    signal(leave_b_chair);
    pay();
    signal(payment);
    wait(receipt);
    exit_shop();
    signal(max_capacity)
}
void barber()
{
    while (true)
    {
        wait(cust_ready);
        wait(coord);
        cut_hair();
        signal(coord);
        signal(finished);
        wait(leave_b_chair);
        signal(barber_chair);
    }
}
void cashier()
{
    while (true)
    {
        wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}
void main()
{
parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber, cashier);
}
```

Figure 5.19 An Unfair Barbershop

```
/* program barbershop2 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3, coord = 3;
semaphore mutex1 = 1, mutex2 = 1;
semaphore cust_ready = 0, leave_b_chair = 0, payment = 0, receipt = 0;
semaphore finished [50] = {0};
int count;
void customer()
{
    int custnr;
    wait(max_capacity);
    enter_shop();
    wait(mutex1);
    count++;
    custnr = count;
    signal(mutex1);
    wait(sofa);
    sit_on_sofa();
    wait(barber_chair);
    get_up_from_sofa();
    signal(sofa);
    sit_in_barber_chair();
    wait(mutex2);
    enqueue1(custnr);
    signal(cust_ready);
    signal(mutex2);
    wait(finished[custnr]);
    leave_barber_chair();
    signal(leave_b_chair);
    pay();
    signal(payment);
    wait(receipt);
    exit_shop();
    signal(max_capacity)
}
```

```
void barber()
{
    int b_cust;
    while (true)
    {
        wait(cust_ready);
        wait(mutex2);
        dequeue1(b_cust);
        signal(mutex2);
        wait(coord);
        cut_hair();
        signal(coord);
        signal(finished[b_cust]);
        wait(leave_b_chair);
        signal(barber_chair);
    }
}
```

```
void cashier()
{
    while (true)
    {
        wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}
```

```
void main()
{
    count := 0;
    parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber,
              cashier);
}
```

Figure 5.20 A Fair Barbershop

Monitor(with signal)

- Problems using semaphores
  - may be difficult to produce a correct program
  - operations are scattered throughout a program
- Monitor is a programming language construct
  - 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

Monitor(with signal)

- Operations for synchronization
  - cwait(c)
    - suspend execution of the calling process on condition c
  - csignal(c)
    - resume execution of some process suspended after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing

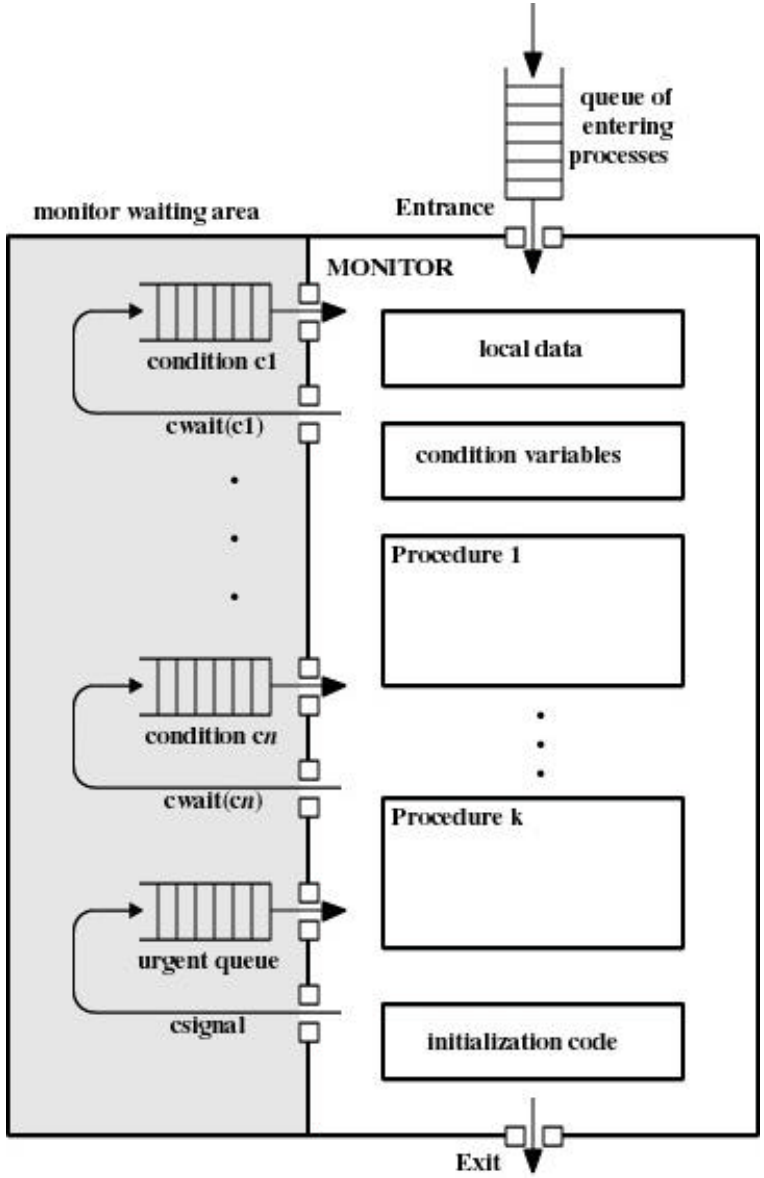


Figure 5.21 Structure of a Monitor

Monitor(with signal)

- What if the csignal does not occur at the end of the procedure
  - process issuing the signal is suspended to make the monitor available and placed in a queue until the monitor is free
    - urgent queue
  - Concurrent Pascal requires that csignal only appears as the last operation executed by a monitor procedure

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

Figure 5.22 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a 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);
}
```

Figure 5.22 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

Monitor(with notify and broadcast)

- Drawbacks of Hoare's monitors
  - if the process issuing csignal has not finished with the monitor, then two additional process switches are required
  - process scheduling associated with a signal must be perfectly reliable
    - when a csignal is issued, a process from the corresponding condition queue must be activated immediately and the scheduler must ensure that no other process enters the monitor before activation

Monitor(with notify and broadcast)

- Lampson/Redell Monitor
  - cnotify instead of csignal
    - cnotify(x) causes the x condition queue to be notified, signaling process continues to execute
    - the process at the head of the condition queue will be resumed at some convenient future time when the monitor is available
      - because there is no guarantee that some other process will not enter the monitor before the waiting process, the waiting process must recheck the condition

Monitor(with notify and broadcast)

- pros and cons
  - at least one extra evaluation of the condition variable
  - no extra process switches
  - no constraints on when the waiting process must run after a cnotify
- cbroadcast
  - causes all processes waiting on a condition to be placed in Ready state
  - convenient when a process does not know how many other processes should be reactivated

```
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 5.23 Bounded Buffer Monitor Code

Message Passing

- Requirements for process interaction
  - synchronization
  - communication
- Message passing can provide both of the above functions

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

Table 5.4 Design Characteristics of Message Systems for Interprocessor Communication and Synchronization

- **Synchronization**
  - Send
    - blocking
    - nonblocking
  - Receive
    - blocking
    - nonblocking
    - test for arrival
- **Addressing**
  - Direct
    - send
    - receive
      - explicit
      - implicit
  - Indirect
    - static
    - dynamic
    - ownership
- **Format**
  - Content
  - Length
    - fixed
    - variable
- **Queuing Discipline**
  - FIFO
  - Priority

Message Passing - 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

Message Passing - Synchronization

- Nonblocking send, blocking receive
  - sender continues processing such as sending messages as quickly as possible
  - receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - neither party is required to wait

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

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

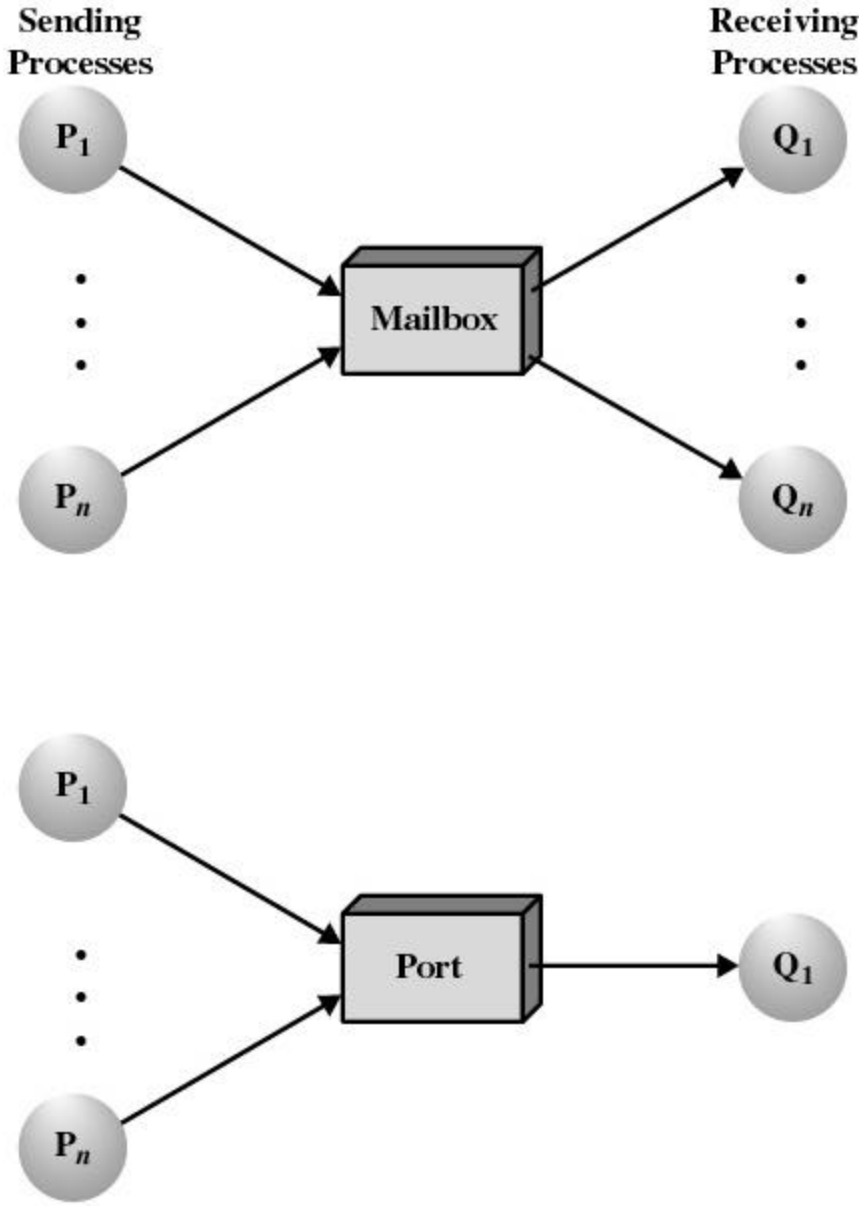


Figure 5.24 Indirect Process Communication

Message Format

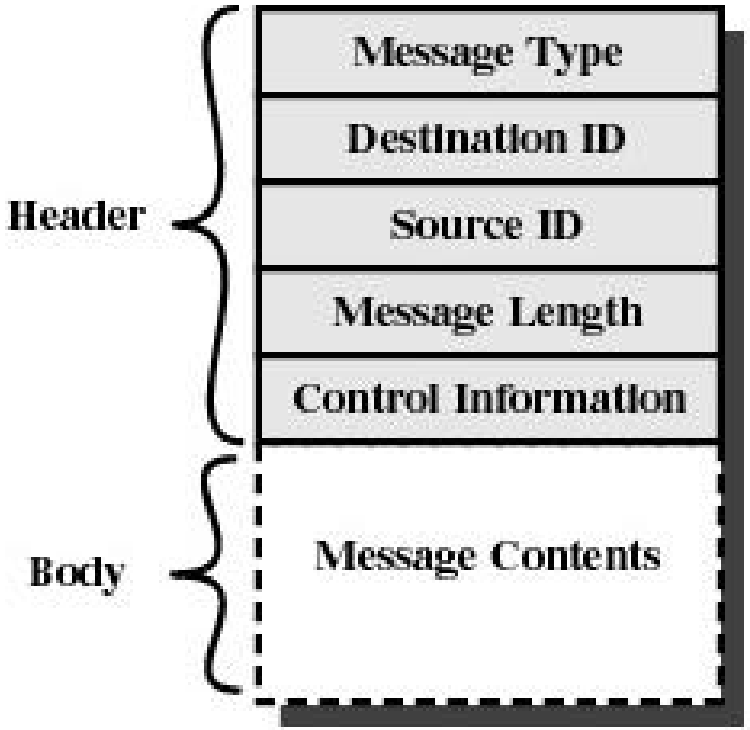


Figure 5.25 General Message Format

```
/* 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 5.26 Mutual Exclusion Using 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);
}
```

Figure 5.27 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

Readers/Writers Problem

- Readers have priority
  - Any number of readers may simultaneously read the file
    - when there is already at least one reader reading, subsequent readers need not wait before entering
  - Only one writer at a time may write to the file
  - If a writer is writing to the file, no reader may read it

Readers/Writers Problem

- Semaphores and variables
  - wsem : enforce mutual exclusion
  - readcount : keep track of the number of readers

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

```
void writer()
{
    while (true)
    {
        wait (wsem);
        WRITEUNIT();
        signal (wsem);
    }
}
void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Figure 5.28 A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority

Readers/Writers Problem

- Writers have priority
  - no new readers are allowed access to the data area once at least one writer has declared a desire to write
  - additional semaphores and variables
    - rsem : inhibits all readers while there is at least one writer desiring access
    - writecount : control the setting of rsem
    - y : control the updating of writecount

```
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
      while (true)
      {
          wait (z);
          wait (rsem);
          wait (x);
          readcount++;
          if (readcount == 1)
          {
              wait (wsem);
          }
          signal (x);
          signal (rsem);
          signal (z);
          READUNIT();
          wait (x);
          readcount--;
          if (readcount == 0)
              signal (wsem);
          signal (x);
      }
}
```

```
void writer ()
{
      while (true)
      {
          wait (y);
          writecount++;
          if (writecount == 1)
              wait (rsem);
          signal (y);
          wait (wsem);
          WRITEUNIT();
          signal (wsem);
          wait (y);
          writecount--;
          if (writecount == 0)
              signal (rsem);
          signal (y);
      }
}
void main()
{
      readcount = writecount = 0;
      parbegin (reader, writer);
}
```

Figure 5. 29 A Solution to the Readers/Writers Problem Using Semaphores: Writers Have Priority