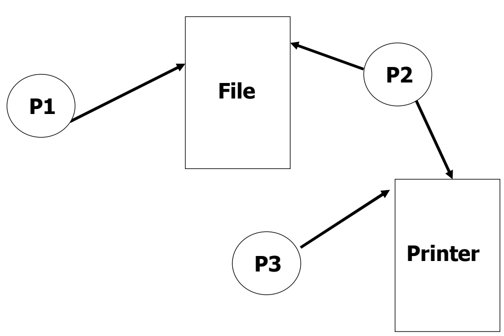
Concurrency:
Mutual Exclusion and Synchronization

Sistemas de Operação, 1º Semestre 2004-2005

Concurrency

- Communication among processes.
- Sharing resources.
- Synchronization of multiple processes.

Concurrent Access to Resources



Critical Section

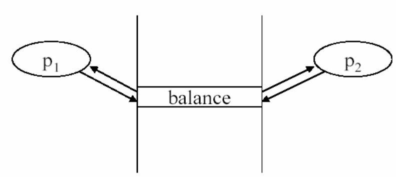
```
shared float balance;
```

Code for p_1

```
 . . .
balance = balance + amount;
 . . .
```

Code for p_2

```
 . . .
balance = balance - amount;
 . . .
```



Critical Section

```
shared double balance;

Code for p1                         Code for p2
 . . .                               . . .
balance = balance + amount;         balance = balance - amount;
 . . .                               . . .

Code for p1                         Code for p2
load     R1, balance                load     R1, balance
load     R2, amount                 load     R2, amount
add      R1, R2                     sub      R1, R2
store    R1, balance                store    R1, balance
```

Race Conditions

- There is a *race* to execute critical sections
- The sections may be different code in different processes
  - Cannot detect with static analysis
- Results of multiple execution are not *determinate*
- Need an OS mechanism to resolve races

Competition Among Processes for Resources

- **Mutual Exclusion**
  - Critical sections
    - Only one program at a time is allowed in its critical section
- **Deadlock**
- **Livelock**
- **Starvation**

Mutual Exclusion

```
/* 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

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 non-critical section must do so without interfering with other processes.
- There should be 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 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.

Busy Waiting: First Attempt

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

First Attempt

- **Busy Waiting**
  - Process is always checking to see if it can enter the critical section. CPU=100%!
  - Process can do nothing productive until it gets permission to enter its critical section.

- Guarantees mutual exclusion
- Processes must strictly alternate in use of their critical sections
- If one process fails, the other process is permanently blocked

Busy Waiting: Second Attempt

Flag[0] = false;	Flag[1] = false;
/* PROCESS 0 */	**/* PROCESS 1 */**

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

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

Second Attempt

- Each process can examine the other's status but cannot alter it.
- When a process wants to enter the critical section is checks the other processes first.
- If no other 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 */
•
•
flag[0] = true;
while (flag[1])
    /* do nothing */;
/* critical section*/;
flag[0] = false;
•
```

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

Third Attempt

- Set flag to enter critical section before check 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 */              /* PROCESS 1 */
•                            •
•                            •
flag[0] = true;              flag[1] = true;
while (flag[1])              while (flag[0])
{                            {
    flag[0] = false;             flag[1] = false;
    /*delay */;                  /*delay */;
    flag[0] = true;              flag[1] = true;
}                            }
/*critical section*/;        /* critical section*/;
flag[0] = false;             flag[1] = false;
•                            •
```

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.

Fourth Attempt

- P0 sets flag[0] to true
- P1 sets flag[1] to true
- P0 checks flag[1]
- P1 checks flag[0]
- P0 sets flag[0] to false
- P1 sets flag[1] to false
- P0 sets flag[0] to true
- P1 sets flag[0] to true

- 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

Dekker's and Peterson's Algorithms

- 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.

Dekker’s Algorithm

```
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);
}
```

Peterson’s Algorithm

```
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);
}
```

Hardware Support: Interrupt Disabling

```
shared double balance;
```

Code for p_1

```
disableInterrupts();
balance = balance + amount;
enableInterrupts();
```

Code for p_2

```
disableInterrupts();
balance = balance - amount;
enableInterrupts();
```

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.
  - Processor is limited in its ability to interleave programs.
  - Multiprocessing
    - disabling interrupts on one processor will not guarantee mutual exclusion

Mutual Exclusion: Hardware Support

- **Special Machine Instructions**
  - Performed in a single instruction cycle
  - Not subject to interference from other. instructions.
- **Test and Set Instruction**

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

Test-and-Set: 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));
}
```

(a) Test and set instruction

Disadvantages Machine Instructions

- 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.

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.
- **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

Semaphore Operations

Operation	Semaphore	Dutch	Meaning
Wait	P	*proberen*	test
Signal	V	*verhogen*	increment

Semaphores

- A <u>*semaphore*</u>, s, is a nonnegative integer variable that can only be changed or tested by these two indivisible functions:

```
V(s): [s = s + 1]
P(s): [while(s == 0) {wait}; s = s - 1]
```

V(s) = sem_signal(s) **P(s) = sem_wait(s)**

Pseudo-Code: Semaphore

```
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

Binary Semaphore

```
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

Semaphores and Mutual Exclusion

```
Proc_0() {                          proc_1() {
  while(TRUE) {                       while(TRUE {
    <compute section>;                  <compute section>;
  wait(mutex)                         wait(mutex)
      <critical section>;                 <critical section>;
  signal(mutex)                       signal(mutex)
  }                                   }
}                                   }

semaphore mutex = 1;
fork(proc_0, 0);
fork(proc_1, 0);
```

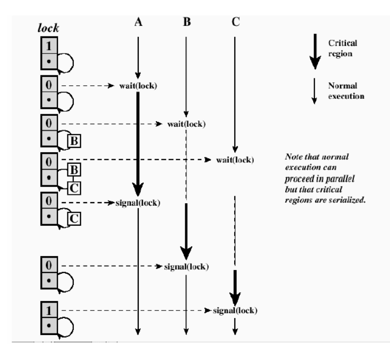
Shared Account Problem

```
Proc_0() {                        proc_1() {
  . . .                             . . .
/* Enter the CS */                /* Enter the CS */
wait(mutex)                       wait(mutex)
    balance += amount;                balance -= amount;
signal(mutex)                     signal(mutex)
  . . .                             . . .
}                                 }

semaphore mutex = 1;

fork(proc_0, 0);
fork(proc_1, 0);
```

Protecting Shared Data with a Semaphore



Weak and Strong Semaphores

- **Strong Semaphore**: fifo order.
- **Weak Semaphore**: does not specify the order in which processes are removed from the queue.

Deadlock

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let *S* and *Q* be two semaphores initialized to 1

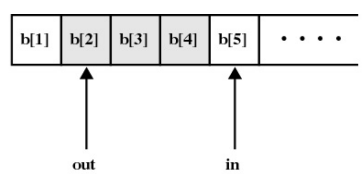
P0	*P1*
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
M	M
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Producer/Consumer Problem (Infinite Bufffer)

- 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.

Infinite Buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

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 */
}
```

Solution (w/ binary semaphores)

```
/* 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);
}
```

Solution with General 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);
}
```

Circular Bounded Buffer

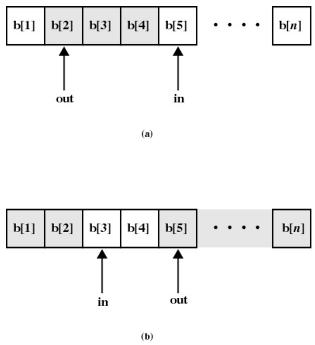


Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

Producer with Circular Buffer

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

Consumer with Circular Buffer

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

Solution for the Bounded Buffer

```
/* 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);
}
```

Producer-Consumer problem with semaphores

```
void producer ( void )
{
    do
        produce ( item );
        wait ( empty );        // empty is semaphore
        wait ( mutex );        // mutex is semaphore
        put ( item );
        signal ( mutex );
        signal ( full );
    while ( 1 );
}

void consumer ( void )
{
    do
        wait ( full );
        wait ( mutex );
        remove ( item );
        signal ( mutex );
        signal ( empty );
        consume ( item );
    while ( 1 );
}
```

Problem: what happens if the order of wait() is reversed in the Producer?

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

```
// Semaphore with block wakeup protocol

class sem_int
{
    private:
        int       value;     // Number of resources
        queue     l;         // List of processes

    public:
        void sem_int ( void )     // Default constructor
        {
            value = 1;
            l = create_queue();
        }

        void sem_int ( int n )    // Constructor function
        {
            value = n;
            l = create_queue();
        }

        void P ( void )
        {
            if ( --value < 0 )
            {
                enqueue ( l, p );   // Enqueue the invoking process
                block();
            }
        }

        void V ( void )
        {
            if ( ++value <= 0 )
            {
                process p = dequeue ( l );
                wakeup ( p );
            }
        }
```

Monitors

- 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

Operations for Synchronization

- 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

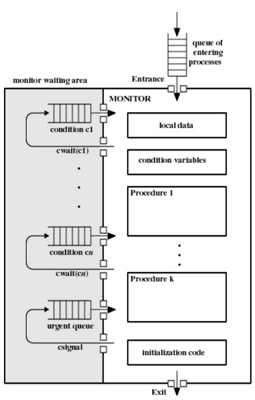


Figure 5.21 Structure of a Monitor

Monitors: Condition Variables

- Additional mechanism for synchronization or communication – the `condition` construct

  `condition x;`

  - `condition` variables are accessed by only two operations – `wait` and `signal`
  - `x.wait()` suspends the process that invokes this operation until another process invokes `x.signal()`
  - `x.signal()` resumes exactly one suspended process; it has no effect if no process is suspended
- Selection of a process to execute within monitor after `signal`
  - `x.signal()` executed by process `P` allowing the suspended process `Q` to resume execution
    1. `P` waits until `Q` leaves the monitor, or waits for another condition
    2. `Q` waits until `P` leaves the monitor, or waits for another condition

```
/* 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 */
}
```

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);
}
```

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

Message Passing

- Enforce mutual exclusion
- Exchange information

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

Message Passing

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

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

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 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

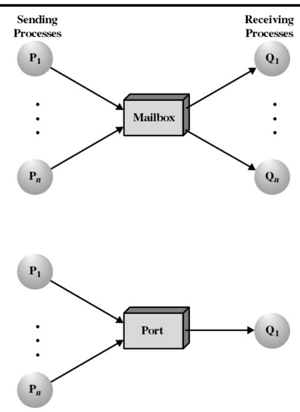


Figure 5.24 Indirect Process Communication

Message Format

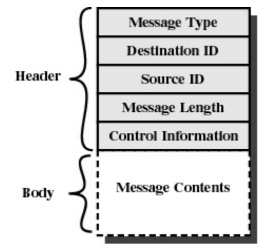


Figure 5.25 General Message Format

Mutual Exclusion using Messages

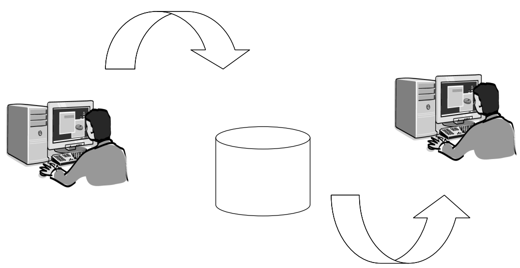
```
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true)
    {
        receive (mutex, msg);
        /* critical section */;
        send (mutex, msg);
        /* remainder */;
    }
}
void main()
{
    create_mailbox (mutex);
    send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

Bounded-Buffer using Messages

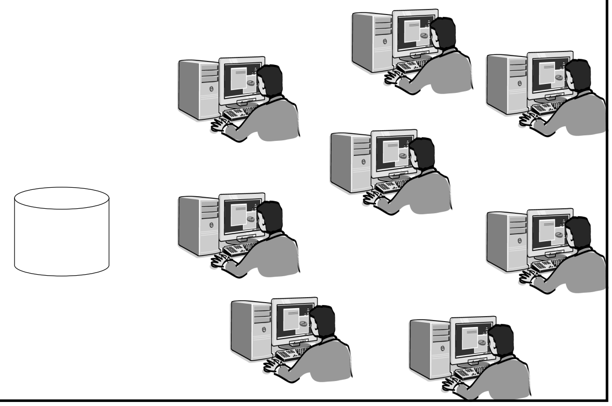
```
const int
    capacity = /* buffering capacity */ ;
    null = /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true)
    {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true)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

Produtor/Consumidor



Leitores/Escritores



O Problema dos Leitores/Escritores

- 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);
}
```

Readers have priority.

Readers/Writers Problem (2)

- 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);
}
```

Writers have priority.

Dining Philosophers

```
enum state_type { thinking, hungry, eating };

class dining_philosophers
{
    private:
        state_type state[5];      // State of five philosophers
        condition  self[5];       // Condition object for synchronization

        void test ( int i )
        {
            if ( ( state[ ( i + 4 ) % 5 ] != eating )  &&
                 ( state[ i ] == hungry )              &&
                 ( state[ ( i + 1 ) % 5 ] != eating ) )
            {
                state[ i ] = eating;
                self[i].signal();
            }
        }

    public:
        void dining_philosophers ( void )   // Constructor
        {
            for ( int i = 0; i < 5; state[i++] = thinking );
        }

        void pickup ( int i )     // i corresponds to the philosopher
        {
            state[i] = hungry;
            test ( i );
            if ( state[i] != eating )
                self[i].wait();
        }

        void putdown ( int i )    // i corresponds to the philosopher
        {
            state[i] = thinking;
            test ( ( i + 4 ) % 5 );
            test ( ( i + 1 ) % 5 );
        }
}
```