“Only a brain-damaged operating system would support task switching and not make the simple next step of supporting multitasking.”

– Calvin Keegan

Processes

- Abstraction of a running program
- Unit of work in the system
- Pseudoparallelism
- A process is traced by listing the sequence of instructions that execute for that process
- The process model
  - Sequential Process / Task
    - A program in execution
    - Program code
    - Current activity
    - Process stack
      - subroutine parameters
      - return addresses
      - temporary variables
    - Data section
      - Global variables
- Concurrent Processes
  - Multiprogramming
  - Interleaving of traces of different processes characterizes the behavior of the CPU
  - Physical resource sharing
    - Required due to limited hardware resources
  - Logical resource sharing
    - Concurrent access to the same resource like files
  - Computation speedup
    - Break each task into subtasks
    - Execute each subtask on separate processing element
  - Modularity
    - Division of system functions into separate modules
  - Convenience
    - Perform a number of tasks in parallel
  - Real-time requirements for I/O
- Process Hierarchies
  - Parent-child relationship
    - \texttt{fork(2)} call in Unix
    - In MS-DOS, parent suspends itself and lets the child execute
- Process states
  - Running
- Ready (Not running, waiting for the CPU)
- Blocked / Wait on an event (other than CPU) (Not running)
- Two other states complete the five-state model – New and Exit
  - A process being created can be said to be in state New; it will be in state Ready after it has been created
  - A process being terminated can be said to be in state Exit

Above model suffices for most of the discussion on process management in operating systems; however, it is limited in the sense that the system screeches to a halt (even in the model) if all the processes are resident in memory and they all are waiting for some event to happen

- Create a new state Suspend to keep track of blocked processes that have been temporarily kicked out of memory to make room for new processes to come in
- The state transition diagram in the revised model is

- Which process to grant the CPU when the current process is swapped out?
  - Preference for a previously suspended process over a new process to avoid increasing the total load on the system
  - Suspended processes are actually blocked at the time of suspension and making them ready will just change their state back to blocked
  - Decide whether the process is blocked on an event (suspended or not) or whether the process has been swapped out (suspended or not)
- The new state transition diagram is

• Implementation of processes
Interprocess Communication

- Process table
  - One entry for each process
  - program counter
  - stack pointer
  - memory allocation
  - open files
  - accounting and scheduling information
- Interrupt vector
  - Contains address of interrupt service procedure
    - saves all registers in the process table entry
    - services the interrupt

• Process creation
  - Build the data structures that are needed to manage the process
  - When is a process created? – job submission, login, application such as printing
  - Static or dynamic process creation
  - Allocation of resources (CPU time, memory, files)
    - Subprocess obtains resources directly from the OS
    - Subprocess constrained to share resources from a subset of the parent process
  - Initialization data (input)
  - Process execution
    - Parent continues to execute concurrently with its children
    - Parent waits until all its children have terminated

• Processes in Unix
  - Identified by a unique integer – process identifier
  - Created by the fork(2) system call
    - Copy the three segments (instructions, user-data, and system-data) without initialization from a program
    - New process is the copy of the address space of the original process to allow easy communication of the parent process with its child
    - Both processes continue execution at the instruction after the fork
    - Return code for the fork is
      - zero for the child process
      - process id of the child for the parent process
  - Use exec(2) system call after fork to replace the child process’s memory space with a new program (binary file)
    - Overlay the image of a program onto the running process
    - Reinitialize a process from a designated program
    - Program changes while the process remains
  - exit(2) system call
    - Finish executing a process
  - wait(2) system call
    - Wait for child process to stop or terminate
    - Synchronize process execution with the exit of a previously forked process
  - brk(2) system call
Interprocess Communication

- Change the amount of space allocated for the calling process’s data segment
- Control the size of memory allocated to a process
  - `signal(3)` library function
    - Control process response to extraordinary events
    - The complete family of `signal` functions (see man page) provides for simplified signal management for application processes

- MS-DOS Processes
  - Created by a system call to load a specified binary file into memory and execute it
  - Parent is suspended and waits for child to finish execution

- Process Termination
  - Normal termination
    - Process terminates when it executes its last statement
    - Upon termination, the OS deletes the process
    - Process may return data (output) to its parent
  - Termination by another process
    - Usually terminated only by the parent of the process because
      - child may exceed the usage of its allocated resources
      - task assigned to the child is no longer required
  - Cascading termination
    - Upon termination of parent process
    - Initiated by the OS

- `cobegin/coend`
  - Also known as `parbegin/parend`
  - Explicitly specify a set of program segments to be executed concurrently
    ```
    cobegin
      p_1;
      p_2;
      ...
      p_n;
    coend;
    
    (a + b) \times (c + d) - (e/f)
    ```

    ```
    cobegin
      t_1 = a + b;
      t_2 = c + d;
      t_3 = e / f;
    coend
    t_4 = t_1 \times t_2;
    t_5 = t_4 - t_3;
    ```

- `fork`, `join`, and `quit` Primitives
  - More general than `cobegin/coend`
  - `fork x`
* Creates a new process q when executed by process p
* Starts execution of process q at instruction labeled x
* Process p executes at the instruction following the fork

– quit
  * Terminates the process that executes this command
– join t, y
  * Provides an indivisible instruction
  * Provides the equivalent of test-and-set instruction in a concurrent language
  \[\text{if } (!--t) \text{ goto } y;\]

– Program segment with new primitives
  \[
m = 3; \\
fork p2; \\
fork p3;
\]
  \[
p1 : t1 = a + b; \text{ join } m, p4; \text{ quit}; \\
p2 : t2 = c + d; \text{ join } m, p4; \text{ quit}; \\
p3 : t3 = e / f; \text{ join } m, p4; \text{ quit}; \\
p4 : t4 = t1 \times t2; \\
t5 = t4 - t3;
\]

Process Control Subsystem in Unix

– Significant part of the Unix kernel (along with the file subsystem)
– Contains three modules
  – Interprocess communication
  – Scheduler
  – Memory management

Interprocess Communication

– Race conditions
  – A race condition occurs when two processes (or threads) access the same variable/resource without doing any synchronization
  – One process is doing a coordinated update of several variables
  – The second process observing one or more of those variables will see inconsistent results
  – Final outcome dependent on the precise timing of two processes
– Example
  * One process is changing the balance in a bank account while another is simultaneously observing the account balance and the last activity date
  * Now, consider the scenario where the process changing the balance gets interrupted after updating the last activity date but before updating the balance
  * If the other process reads the data at this point, it does not get accurate information (either in the current or past time)

Critical Section Problem

– Section of code that modifies some memory/file/table while assuming its exclusive control
• Mutually exclusive execution in time

• Template for each process that involves critical section

\[
\text{do} \\
\{
\ldots /* \text{Entry section}; */
\text{critical_section(); /* Assumed to be present */}
\ldots /* \text{Exit section */}
\text{remainder_section(); /* Assumed to be present */}
\}\text{while ( 1 );}
\]

You are to fill in the gaps specified by \ldots for entry and exit sections in this template and test the resulting program for compliance with the protocol specified next

• Design of a protocol to be used by the processes to cooperate with following constraints
  
  – Mutual Exclusion – If process \( p_i \) is executing in its critical section, then no other processes can be executing in their critical sections.
  
  – Progress – If no process is executing in its critical section, the selection of a process that will be allowed to enter its critical section cannot be postponed indefinitely.
  
  – Bounded Waiting – There must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

• Assumptions
  
  – No assumption about the hardware instructions
  
  – No assumption about the number of processors supported
  
  – Basic machine language instructions executed atomically

• Disabling interrupts
  
  – Brute-force approach
  
  – Not proper to give users the power to disable interrupts
    
    * User may not enable interrupts after being done
    
    * Multiple CPU configuration

• Lock variables
  
  – Share a variable that is set when a process is in its critical section

• Strict alternation

\[
\text{extern int turn; /* Shared variable between both processes */}
\]

\[
\text{do} \\
\{
\text{while ( turn != i ) /* do nothing */ ;}
\text{critical_section();}
\text{turn = j;}
\text{remainder_section();}
\}\text{while ( 1 );}
\]
- Does not satisfy progress requirement
- Does not keep sufficient information about the state of each process

• Use of a flag

extern int flag[2]; /* Shared variable; one for each process */

do
  flag[i] = 1; /* true */
  while ( flag[j] );
  critical_section();
  flag[i] = 0; /* false */
  remainder_section();
while ( 1 );

- Satisfies the mutual exclusion requirement
- Does not satisfy the progress requirement

Time $T_0$ $p_0$ sets flag[0] to true
Time $T_1$ $p_1$ sets flag[1] to true

Processes $p_0$ and $p_1$ loop forever in their respective while statements
- Critically dependent on the exact timing of two processes
- Switch the order of instructions in entry section
  * No mutual exclusion

• Peterson’s solution

- Combines the key ideas from the two earlier solutions

/* Code for process 0; similar code exists for process 1 */

extern int flag[2]; /* Shared variables */
extern int turn; /* Shared variable */

process_0( void )
{
  do
    /* Entry section */
    flag[0] = true; /* Raise my flag */
    turn = 1; /* Cede turn to other process */
    while ( flag[1] && turn == 1 ) ;

    critical_section();

    /* Exit section */
    flag[0] = false;

    remainder_section();
  while ( 1 );
}

• Multiple Process Solution – Solution 4

- The array flag can take one of the three values (idle, want-in, in-cs)
enum state { idle, want_in, in_cs };  
extern int turn;  
extern state flag[n];  // Flag corresponding to each process (in shared memory)  

// Code for process i  

int  j;  // Local to each process  

  do  
  do  
    flag[i] = want_in;  // Raise my flag  
    j = turn;  // Set local variable  
    while ( j != i )   
      j = ( flag[j] != idle ) ? turn : ( j + 1 ) % n;  
  // Declare intention to enter critical section  
  flag[i] = in_cs;  
  // Check that no one else is in critical section  
  for ( j = 0; j < n; j++ )   
    if ( ( j != i ) && ( flag[j] == in_cs ) )       
      break;  

  while ( j < n ) || ( turn != i && flag[turn] != idle );  

  // Assign turn to self and enter critical section  
  turn = i;  
  critical_section();  

  // Exit section  
  j = (turn + 1) % n;  
  while (flag[j] == idle) do  
    j = (j + 1) % n;  

  // Assign turn to the next waiting process and change own flag to idle  
  turn = j;  
  flag[i] = idle;  

  remainder_section();  
  while ( 1 );  

  – \( p_i \) enters the critical section only if \( \text{flag}[j] \neq \text{in-cs} \) for all \( j \neq i \).  
  – \( \text{turn} \) can be modified only upon entry to and exit from the critical section. The first contending process enters its critical section.  
  – Upon exit, the successor process is designated to be the one following the current process.  
  – Mutual Exclusion  
    * \( p_i \) enters the critical section only if \( \text{flag}[j] \neq \text{in-cs} \) for all \( j \neq i \).  
    * Only \( p_i \) can set \( \text{flag}[i] = \text{in-cs} \).
Interprocess Communication

- \( p_i \) inspects flag\[j\] only while flag\[i\] = in_cs.
  - Progress
    - turn can be modified only upon entry to and exit from the critical section.
    - No process is executing or leaving its critical section \( \Rightarrow \) turn remains constant.
    - First contending process in the cyclic ordering \( \text{turn}, \text{turn+1}, \ldots, n-1, 0, \ldots, \text{turn-1} \) enters its critical section.
  - Bounded Wait
    - Upon exit from the critical section, a process must designate its unique successor the first contending process in the cyclic ordering \( \text{turn+1}, \ldots, n-1, 0, \ldots, \text{turn-1}, \text{turn} \).
    - Any process waiting to enter its critical section will do so in at most \( n-1 \) turns.

- Bakery Algorithm
  - Each process has a unique id
  - Process id is assigned in a completely ordered manner

```c
extern bool choosing[n]; /* Shared Boolean array */
extern int number[n];   /* Shared integer array to hold turn number */

void process_i ( int i ) /* ith Process */
{
    do
        choosing[i] = true;
        number[i] = 1 + max(number[0], ..., number[n-1]);
        choosing[i] = false;
        for ( int j = 0; j < n; j++ )
            { }
    critical_section();
    number[i] = 0;
    remainder_section();
    while ( 1 );
}
```

- If \( p_i \) is in its critical section and \( p_k \) \( (k \neq i) \) has already chosen its number\[k\] \( \neq 0 \), then \( \text{(number[i],i)} < \text{(number[k],k)} \).

Synchronization Hardware

- test_and_set instruction

```c
int test_and_set (int& target )
{
    int tmp;
    tmp = target;
    target = 1; /* True */
    return ( tmp );
}
```
• Implementing Mutual Exclusion with test_and_set

    do
    while test_and_set ( lock );
    critical_section();
    lock = false;
    remainder_section();
    while ( 1 );

Semaphores

• Producer-consumer Problem
  – Shared buffer between producer and consumer
  – Number of items kept in the variable count
  – Printer spooler
  – The | operator
  – Race conditions

• An integer variable that can only be accessed through two standard atomic operations – wait (P) and signal (V)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Semaphore</th>
<th>Dutch</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait</td>
<td>P</td>
<td>proberen</td>
<td>test</td>
</tr>
<tr>
<td>Signal</td>
<td>V</td>
<td>verhogen</td>
<td>increment</td>
</tr>
</tbody>
</table>

• The classical definitions for wait and signal are

    wait ( S ): while ( S <= 0 );
    S--;

    signal ( S ): S++;

• Mutual exclusion implementation with semaphores

    do
    wait (mutex);
    critical_section();
    signal (mutex);
    remainder_section();
    while ( 1 );

• Synchronization of processes with semaphores

    \[
    \begin{array}{c|c}
    p_1 & S_1: \\
    \hline
    signal (synch); \\
    \hline
    p_2 & wait (synch); \\
    \hline
    S_2: & \\
    \end{array}
    \]

• Implementing Semaphore Operations
  – Binary semaphores using test_and_set
    * Check out the instruction definition as previously given
  – Implementation with a busy-wait
class bin_semaphore
{
    private:
        bool    s;  /* Binary semaphore */
    public:
        bin_semaphore ( void ) // Default constructor
        {
            s = false;
        }
        void P ( void ) // Wait on semaphore
        {
            while ( test_and_set ( s ) );
        }
        void V ( void ) // Signal the semaphore
        {
            s = false;
        }
}

– General semaphore

class semaphore
{
    private:
        bin_semaphore   mutex;
        bin_semaphore   delay;
        int              count;
    public:
        void semaphore ( void ) // Default constructor
        {
            count = 1;
            delay.P();
        }

        void semaphore ( int num ) // Parameterized constructor
        {
            count = num;
            delay.P();
        }

        void P ( void )
        {
            mutex.P();
            if ( --count < 0 )
            {
                mutex.V();
                delay.P();
            }
            mutex.V();
        }

        void V ( void )


```
{
    mutex.P();
    if ( ++count <= 0 )
        delay.V();
    else
        mutex.V();
}
```

– Busy-wait Problem – Processes waste CPU cycles while waiting to enter their critical sections
  * Modify wait operation into the block operation. The process can block itself rather than busy-waiting.
  * Place the process into a wait queue associated with the critical section
  * Modify signal operation into the wakeup operation.
  * Change the state of the process from wait to ready.

– Block-Wakeup Protocol

```c
// 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 );
        }
    }

```
Interprocess Communication

Producer-Consumer problem with semaphores

```c
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 if order of wait is reversed in producer

Event Counters

- Solve the producer-consumer problem without requiring mutual exclusion
- Special kind of variable with three operations
  1. E.read(): Return the current value of E
  2. E.advance(): Atomically increment E by 1
  3. E.await(v): Wait until E has a value of v or more
- Event counters always start at 0 and always increase

```c
class event_counter
{
    int ec;       // Event counter

public:
    event_counter ( void ) // Default constructor
    {
        ec = 0;
    }
    int read ( void ) const { return ( ec ); }
    void advance ( void ) { ec++; }
    void await ( const int v ) { while ( ec < v ); }
};
```
extern event_counter in, out; // Shared event counters

void producer ( void )
{
    int sequence ( 0 );
    do
        produce ( item );
        sequence++;
        out.await ( sequence - num_buffers );
        put ( item );
    in.advance();
    while ( 1 );
}

void consumer ( void )
{
    int sequence ( 0 );
    do
        sequence++;
        in.await ( sequence );
        remove ( item );
        out.advance();
        consume ( item );
    while ( 1 );
}

Higher-Level Synchronization Methods

• P and V operations do not permit a segment of code to be designated explicitly as a critical section.

• Two parts of a semaphore operation
  - Block-wakeup of processes
  - Counting of semaphore

• Possibility of a deadlock – Omission or unintentional execution of a V operation.

• Monitors
  - Implemented as a class with private and public functions
  - Collection of data [resources] and private functions to manipulate this data
  - A monitor must guarantee the following:
    * Access to the resource is possible only via one of the monitor procedures.
    * Procedures are mutually exclusive in time. Only one process at a time can be active within the monitor.
  - 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
Choice 1 advocated by Hoare

- The Dining Philosophers Problem – Solution by Monitors

```cpp
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();
        }
    }

class 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 );
    }
}

- Philosopher i must invoke the operations pickup and putdown on an instance dp of the dining_philosophers monitor

dining_philosophers dp;

dp.pickup(i); // Philosopher i picks up the chopsticks
...

dp.eat(i); // Philosopher i eats (for random amount of time)
...

dp.putdown(i); // Philosopher i puts down the chopsticks
```
Interprocess Communication

- No two neighbors eating simultaneously – no deadlocks
- Possible for a philosopher to starve to death

**Implementation of a Monitor**

- Execution of procedures must be mutually exclusive
- A wait must block the current process on the corresponding condition
- If no process in running in the monitor and some process is waiting, it must be selected. If more than one waiting process, some criterion for selecting one must be deployed.

**Implementation using semaphores**

- Semaphore mutex corresponding to the monitor initialized to 1
  - Before entry, execute wait(mutex)
  - Upon exit, execute signal(mutex)
- Semaphore next to suspend the processes unable to enter the monitor initialized to 0
- Integer variable next_count to count the number of processes waiting to enter the monitor
  
  ```
  mutex.wait();
  ...
  ```

  ```
  void P ( void ) { ... } // Body of P()
  ...
  if ( next_count > 0 )
      next.signal();
  else
      mutex.signal();
  ```

- Semaphore x_sem for condition x, initialized to 0
- Integer variable x_count

```
class condition
{
    int num_waiting_procs;
    semaphore sem;
    static int next_count;
    static semaphore next;
    static semaphore mutex;

    public:
        condition ( void ) // Default constructor
            : sem ( 0 )
        {
            num_waiting_procs = 0;
        }

        void wait ( void )
        {
            num_waiting_procs++;
            if ( next_count > 0 )
                next.signal();
            else
                mutex.signal();
            sem.wait();
            num_waiting_procs--;
        }
```
void signal ( void )
{
    if ( num_waiting_procs <= 0 )
        return;
    num_waiting_procs++;
    sem.signal();
    next.wait();
    next_count--;
}

• Conditional Critical Regions (CCRs)
  – Designed by Hoare and Brinch-Hansen to overcome the deficiencies of semaphores
  – Explicitly designate a portion of code to be critical section
  – Specify the variables (resource) to be protected by the critical section
    resource r :: v_1, v_2, ..., v_n
  – Specify the conditions under which the critical section may be entered to access the elements that form the
    resource
    region r when B do S
    * B is a condition to guard entry into critical section S
    * At any time, only one process is permitted to enter the code segment associated with resource r
  – The statement region r when B do S is implemented by
    semaphore mutex ( 1 ), delay ( 0 );
    int delay_cnt ( 0 );
    mutex.P();
    del_cnt++;
    while ( !B )
    {
        mutex.V();
        delay.P();
        mutex.P();
    }
    del_cnt--;
    S;       // Critical section code
    for ( int i ( 0 ); i < del_cnt; i++ )
        delay.V();
    mutex.V();

Message-Based Synchronization Schemes

• Communication between processes is achieved by:
  – Shared memory (semaphores, CCRs, monitors)
  – Message systems
    * Desirable to prevent sharing, possibly for security reasons or no shared memory availability due to different
      physical hardware
• Communication by Passing Messages
  – Processes communicate without any need for shared variables
Interprocess Communication

- Two basic communication primitives
  - *send message*
  - *receive message*

  \[
  \text{send}(P, \text{message}) \quad \text{Send a message to process } P \\
  \text{receive}(Q, \text{message}) \quad \text{Receive a message from process } Q
  \]

- Messages passed through a communication link

- Producer/Consumer Problem

  ```c
  void producer ( void )
  {
    while ( 1 )
    {
      produce ( data );
      send ( consumer, data );
    }
  }

  void consumer ( void )
  {
    while ( 1 )
    {
      receive ( producer, data );
      consume ( data );
    }
  }
  ```

- Issues to be resolved in message communication
  - *Synchronous v/s Asynchronous Communication*
    - Upon send, does the sending process continue (asynchronous or nonblocking communication), or does it wait for the message to be accepted by the receiving process (synchronous or blocking communication)?
    - What happens when a receive is issued and there is no message waiting (blocking or nonblocking)?
  - *Implicit v/s Explicit Naming*
    - Does the sender specify exactly one receiver (explicit naming) or does it transmit the message to all the other processes (implicit naming)?
      \[
      \text{send}(p, \text{message}) \quad \text{Send a message to process } p \\
      \text{send}(A, \text{message}) \quad \text{Send a message to mailbox } A
      \]
    - Does the receiver accept from a certain sender (explicit naming) or can it accept from any sender (implicit naming)?
      \[
      \text{receive}(p, \text{message}) \quad \text{Receive a message from process } p \\
      \text{receive}(id, \text{message}) \quad \text{Receive a message from any process; } id \text{ is the process id} \\
      \text{receive}(A, \text{message}) \quad \text{Receive a message from mailbox } A
      \]

- Ports and Mailboxes
  - Achieve synchronization of asynchronous process by embedding a busy-wait loop, with a non-blocking receive to simulate the effect of implicit naming
    - Inefficient solution
  - Indirect communication avoids the inefficiency of busy-wait
    - Make the queues holding messages between senders and receivers visible to the processes, in the form of mailboxes
    - Messages are sent to and received from mailboxes
    - Most general communication facility between \( n \) senders and \( m \) receivers
Interprocess Communication

- Unique identification for each mailbox
- A process may communicate with another process by a number of different mailboxes
- Two processes may communicate only if they have a shared mailbox

• Properties of a communication link
  - A link is established between a pair of processes only if they have a shared mailbox
  - A link may be associated with more than two processes
  - Between each pair of communicating processes, there may be a number of different links, each corresponding to one mailbox
  - A link may be either unidirectional or bidirectional

• Ports
  - In a distributed environment, the receive referring to same mailbox may reside on different machines
  - Port is a limited form of mailbox associated with only one receiver
  - All messages originating with different processes but addressed to the same port are sent to one central place associated with the receiver

Remote Procedure Calls

• High-level concept for process communication
• Transfers control to another process, possibly on a different computer, while suspending the calling process
• Called procedure resides in separate address space and no global variables are shared
• Communication strictly by parameters

  send (RP_guard, parameters);
  receive (RP_guard, results);

• The remote procedure guard is implemented by

  void RP_guard ( void )
  {
    do
      receive (caller, parameters);
      ...
      send (caller, results);
    while ( 1 );
  }

• Static versus dynamic creation of remote procedures
• rendezvous mechanism in Ada