Chapter 3
Process Concept

Processes

- Objective:
  - Process Concept & Definitions
- Process Classification:
  - Operating system processes executing system code
  - User processes executing system code
  - User processes executing user code

Processes

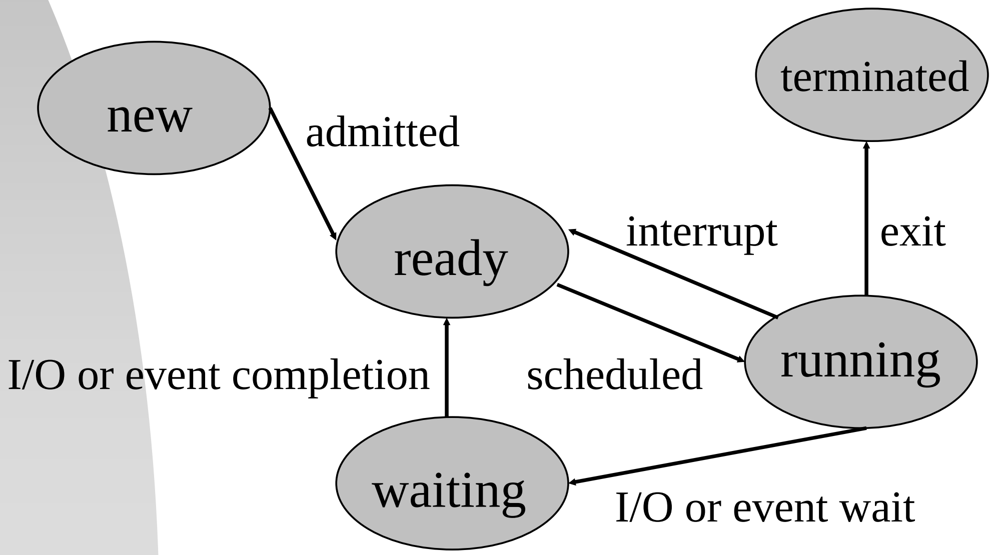
- Example: Special Processes in Unix
  - PID 0 – *Swapper* (i.e., the scheduler)
    - Kernel process
    - No program on disks correspond to this process
  - PID 1 – *init* responsible for bringing up a Unix system after the kernel has been bootstrapped. (/etc/rc* & init or /sbin/rc* & init)
    - User process with superuser privileges
  - PID 2 - pagedaemon responsible for paging
    - Kernel process

Processes

- Process
  - A Basic Unit of Work from the Viewpoint of OS
  - Types:
    - Sequential processes: an activity resulted from the execution of a program by a processor
    - Multi-thread processes
  - An Active Entity
    - Program Code – A Passive Entity
    - Stack and Data Segments
  - The Current Activity
    - PC, Registers , Contents in the Stack and Data Segments

Processes

- Process State

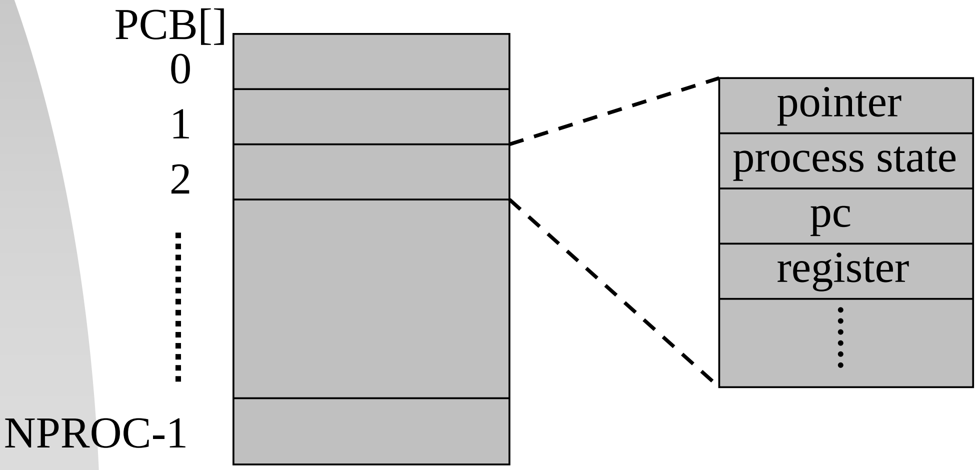


Processes

- Process Control Block (PCB)
  - Process State
  - Program Counter
  - CPU Registers
  - CPU Scheduling Information
  - Memory Management Information
  - Accounting Information
  - I/O Status Information

Processes

- PCB: The repository for any information that may vary from process to process

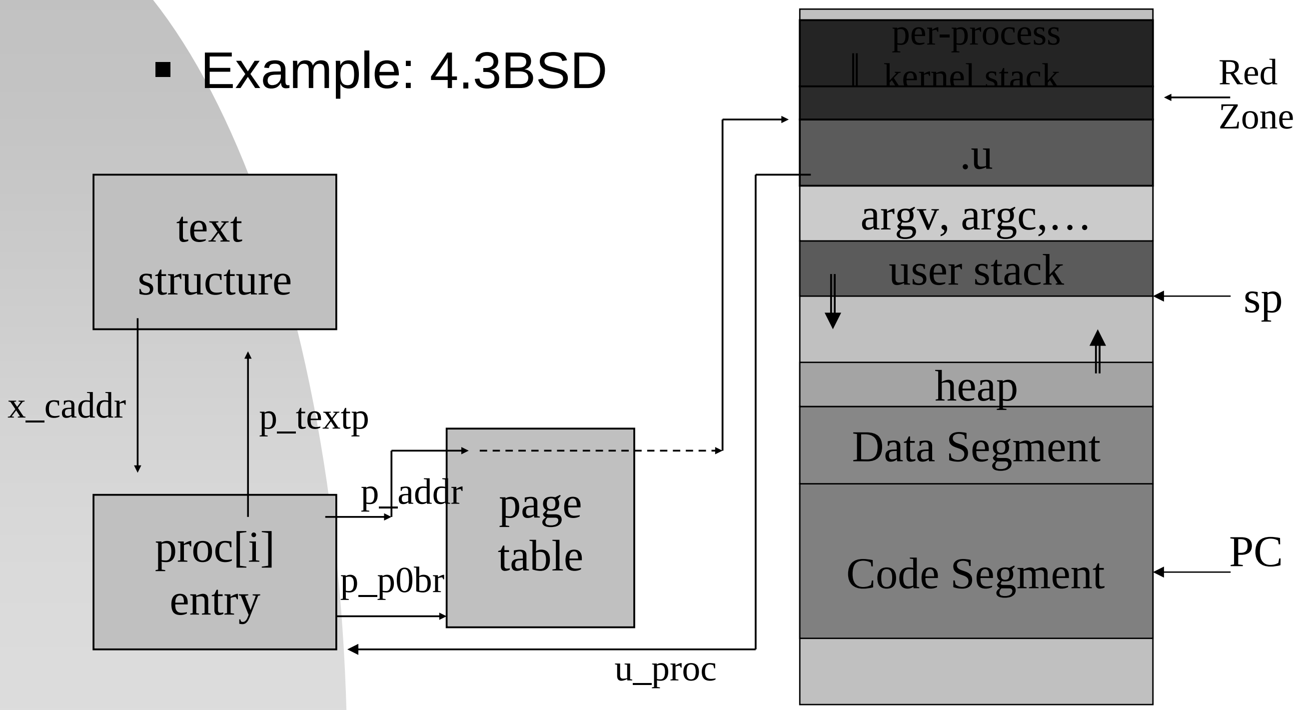


Processes

- Process Control Block (PCB) – An Unix Example
  - proc[i]
    - Everything the system must know when the process is swapped out.
      - pid, priority, state, timer counters, etc.
  - .u
    - Things the system should know when process is running
      - signal disposition, statistics accounting, files[], etc.

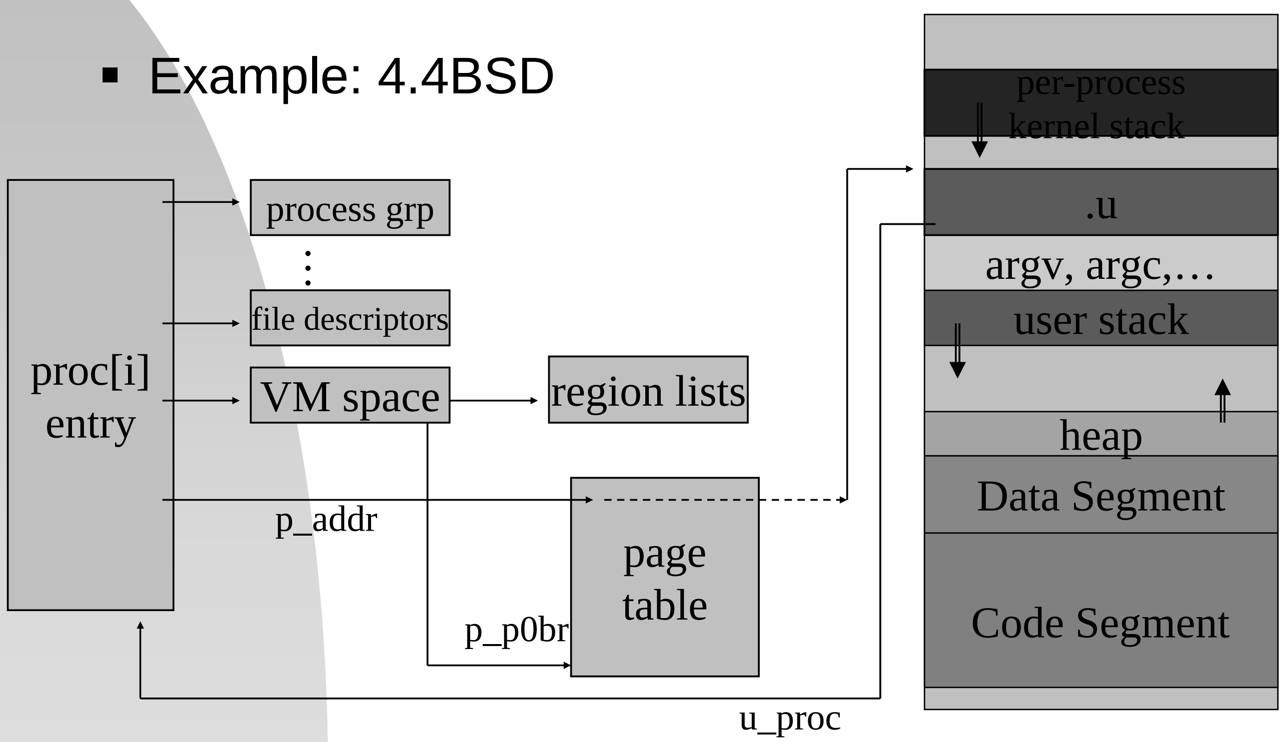
Processes

- Example: 4.3BSD



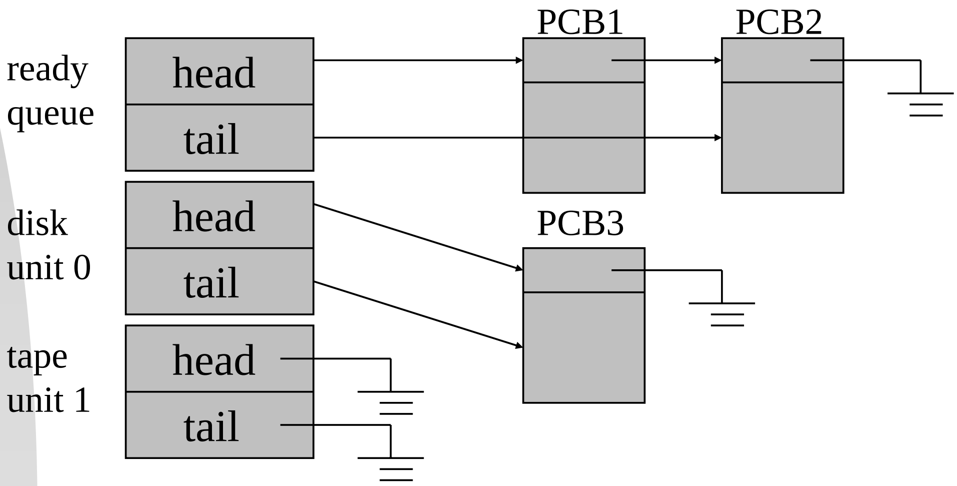
Processes

- Example: 4.4BSD

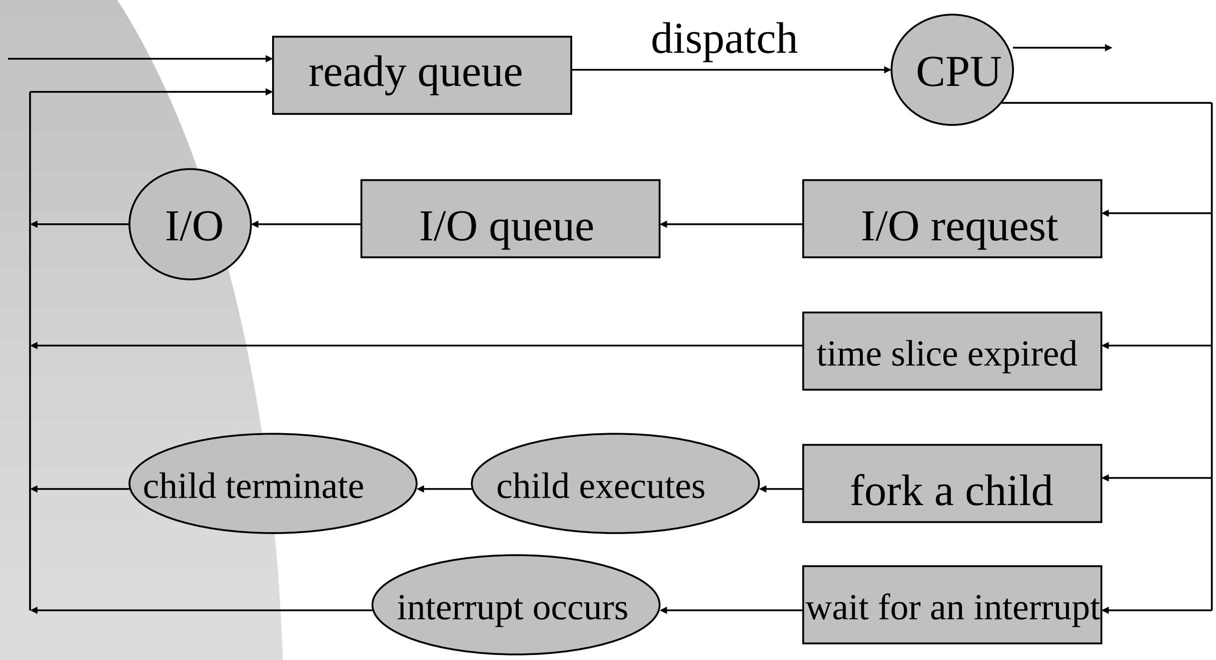


Process Scheduling

- The goal of multiprogramming
  - Maximize CPU/resource utilization!
- The goal of time sharing
  - Allow each user to interact with his/her program!

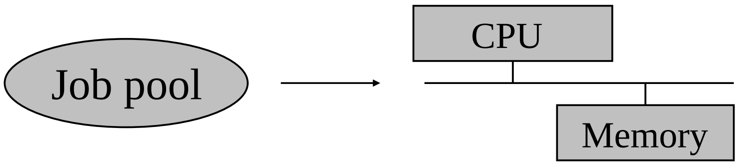


Process Scheduling – A Queueing Diagram



Process Scheduling – Schedulers

- Long-Term (/Job) Scheduler



- Goal: Select a good mix of I/O-bound and CPU-bound process
- Remarks：
  1. Control the degree of multiprogramming
  2. Can take more time in selecting processes because of a longer interval between executions
  3. May not exist physically

Process Scheduling – Schedulers

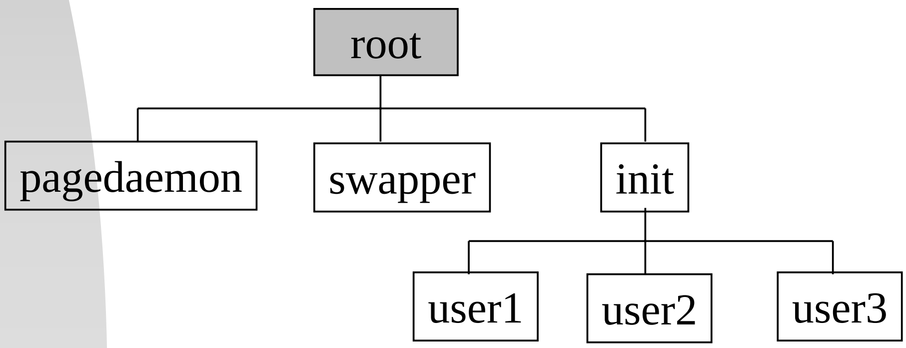
- Short-Term (/CPU) Scheduler
  - Goal：Efficiently allocate the CPU to one of the ready processes according to some criteria.
- Mid-Term Scheduler
  - Swap processes in and out memory to control the degree of multiprogramming

Process Scheduling – Context Switches

- Context Switch ~ Pure Overheads
  - Save the state of the old process and load the state of the newly scheduled process.
    - The context of a process is usually reflected in PCB and others, e.g., .u in Unix.
- Issues：
  - The cost depends on hardware support
    - e.g. processes with multiple register sets or computers with advanced memory management.
  - Threads, i.e., light-weight process (LWP), are introduced to break this bottleneck！

Operations on Processes

- Process Creation & Termination
  - Restrictions on resource usage
  - Passing of Information
  - Concurrent execution



Operations on Processes

- Process Duplication
  - A copy of parent address space + context is made for child, except the returned value from fork()：
    - Child returns with a value 0
    - Parent returns with process id of child
  - No shared data structures between parent and child – Communicate via shared files, pipes, etc.
  - Use execve() to load a new program
  - fork() vs vfork() (Unix)

Operations on Processes

- A Unix Example:

```
…
 if ( pid = fork() ) == 0) {
    /* child process */
     execlp(“/bin/ls”, “ls”, NULL);
 } else if (pid < 0) {
    fprintf(stderr, “Fork Failed”);
    exit(-1);
 } else {
    /* parent process */
    wait(NULL);
  }
```

Operations on Processes

- A Win32 API Example:

```
STARTUPINFO si;  // properties, e.g., window size, handles to infile
 PROCESS.INFORMATION pi; // a handle and ID's to the newly
   …                                  //       created process & its thread
  if (!CreateProcess(NULL, //use command line
      "c:\\WINDOWS\\system32\\mspaint.exe", // command line
      NULL, // don't inherit process handle
      NULL, // don't inherit thread handle
      FALSE, // disable handle inheritance
      0, // no creation flags
      NULL, // use parent's environment block
      NULL, // use parent's existing directory
     &si, &pi)) {
      fprintf(stderr, "Create Process Failed");
      return -1;
   }
  WaitForSingleObject(pi.hProcess, INFINITE);
   …
  }
```


Operations on Processes

- Termination of Child Processes
  - Reasons:
    - Resource usages, needs, etc.
  - Kill, exit, wait, abort, signal, etc.
- Cascading Termination

Interprocess Communication

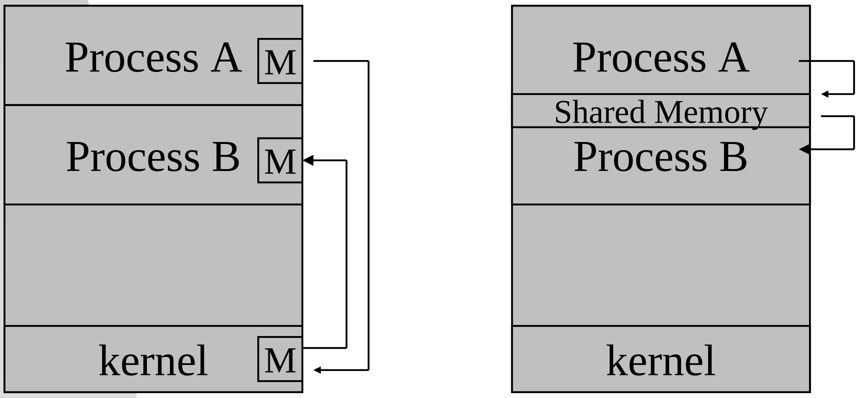
- Cooperating processes can affect or be affected by the other processes
  - Independent Processes
- Reasons:
  - Information Sharing, e.g., files
  - Computation Speedup, e.g., parallelism.
  - Modularity, e.g., functionality dividing
  - Convenience, e.g., multiple work

Interprocess Communication

- Why Inter-Process Communication (IPC)?
  - Exchanging of Data and Control Information!
- Why Process Synchronization?
  - Protect critical sections!
  - Ensure the order of executions!

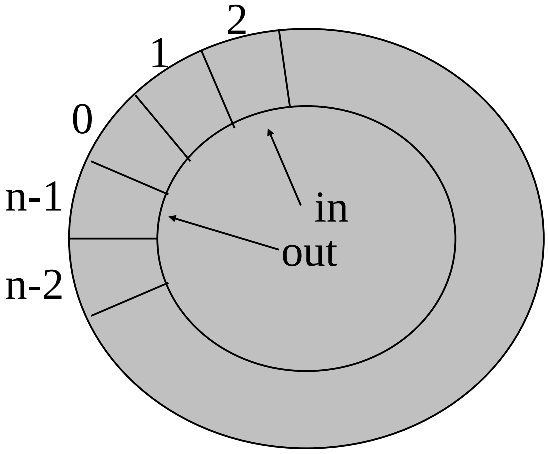
Interprocess Communication

- Shared Memory
  - Max Speed & Comm Convenience
- Message Passing
  - No Access Conflict & Easy Implementation



Interprocess Communication – Shared Memory

- A Consumer-Producer Example:
  - Bounded buffer or unbounded buffer
    - Supported by inter-process communication (IPC) or by hand coding



buffer[0…n-1]

Initially,

in=out=0 ;

Interprocess Communication – Shared Memory

Producer:

```
while (1) {
    /* produce an item nextp */
      while (((in+1) % BUFFER_SIZE) == out)
              ;  /* do nothing */
      buffer[ in ] = nextp;
      in = (in+1) % BUFFER_SIZE;
}
```


Interprocess Communication – Shared Memory

Consumer:

```
while (1) {
      while (in == out)
              ; /* do nothing */
      nextc = buffer[ out ];
      out = (out+1) % BUFFER_SIZE ;
      /* consume the item in nextc */
}
```

Interprocess Communication – Message Passing

- Logical Implementation of Message Passing
  - Fixed/variable msg size, symmetric/asymmetric communication, direct/indirect communication, automatic/explicit buffering, send by copy or reference, etc.

Interprocess Communication – Message Passing

- Classification of Communication by Naming
  - Processes must have a way to refer to each other!
  - Types
    - Direct Communication
    - Indirect Communication

Interprocess Communication – Direct Communication

- Process must explicitly name the recipient or sender of a communication
  - Send(P, msg), Receive(Q, msg)
- Properties of a Link:
  a. Communication links are established automatically.
  b. Two processes per a link
  c. One link per pair of processes
  d. Bidirectional or unidirectional

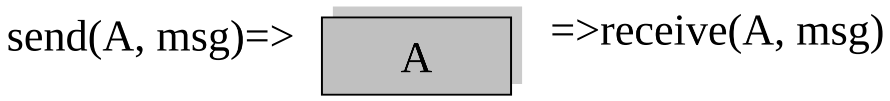
Interprocess Communication – Direct Communication

- Issue in Addressing:
  - Symmetric or asymmetric addressing

    Send(P, msg), Receive(id, msg)
- Difficulty:
  - Process naming vs modularity

Interprocess Communication – Indirect Communication

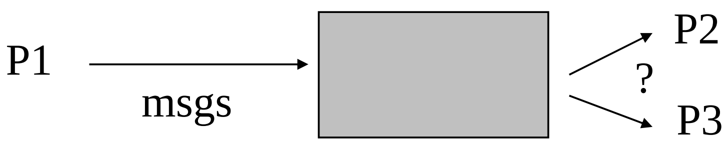
- Two processes can communicate only if the process share a mailbox (or ports)



- Properties:
  1. A link is established between a pair of processes only if they share a mailbox.
  2. *n* processes per link for *n* >= 1.
  3. *n* links can exist for a pair of processes for *n* >=1.
  4. Bidirectional or unidirectional

Interprocess Communication – Indirect Communication

- Issues:
  a. Who is the recipient of a message?



  b. Owners vs Users
  - Process → owner as the sole recipient?
  - OS → Let the creator be the owner?
    Privileges can be passed?
    Garbage collection is needed?

Interprocess Communication – Synchronization

- Blocking or Nonblocking (Synchronous versus Asynchronous)
  - Blocking send
  - Nonblocking send
  - Blocking receive
  - Nonblocking receive
- Rendezvous – blocking send & receive

Interprocess Communication – Buffering

- The Capacity of a Link = the # of messages could be held in the link.
  - Zero capacity(no buffering)
    - Msg transfer must be synchronized – rendezvous!
  - Bounded capacity
    - Sender can continue execution without waiting till the link is full
  - Unbounded capacity
    - Sender is never delayed!
- The last two items are for asynchronous communication and may need acknowledgement

Interprocess Communication – Buffering

- Special cases:
  a. Msgs may be lost if the receiver can not catch up with msg sending → synchronization
  b. Senders are blocked until the receivers have received msgs and replied by reply msgs

     → A Remote Procedure Call (RPC) framework

Interprocess Communication – Exception Conditions

- Process termination
  a. Sender Termination→ Notify or terminate the receiver!
  b. Receiver Termination
     a. No capacity → sender is blocked.
     b. Buffering→ messages are accumulated.

Interprocess Communication – Exception Conditions

- Ways to Recover Lost Messages (due to hardware or network failure):
  - OS detects & resends messages.
  - Sender detects & resends messages.
  - OS detects & notify the sender to handle it.
- Issues:
  a. Detecting methods, such as timeout!
  b. Distinguish multiple copies if retransmitting is possible
- Scrambled Messages:
  - Usually OS adds checksums, such as CRC, inside messages & resend them as necessary!

Example – POSIX

- Creation of Shared Memory Segment

  segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

  - IPC_PRIVATE → new, size in bytes, rights
- Attachment, Detachment, & Deletion

  sh_mem = (char *) shmat(segment_id, NULL, 0)

  shmdt(sh_mem)

  shmctl(segment_id, IPC_RMID, NULL);

  - Seg_ID, location to attach, mode (0:rw, 1:r)
- Access
  - Sprintf(sh_mem, “Writing to shared mem”);

Example – Mach

- Mach – A message-based OS from the Carnegie Mellon University
  - When a task is created, two special mailboxes, called ports, are also created.
    - The *Kernel* mailbox is used by the kernel to communication with the tasks
    - The *Notify* mailbox is used by the kernel sends notification of event occurrences.

Example - Mach

- Three system calls for message transfer:
  - msg_send:
    - Options when mailbox is full:

    a. Wait indefinitely

    b. Return immediately

    c. Wait at most for *n* ms

    d. Temporarily cache a message.

       a. A cached message per sending thread for a mailbox

* One task can either own or receive from a mailbox.

Example - Mach

- msg_receive
  - To receive from a mailbox or a set of mailboxes. Only one task can own & have a receiving privilege of it

    * options when mailbox is empty:

    a. Wait indefinitely
    b. Return immediately
    c. Wait at most for *n* ms
- msg_rpc
  - Remote Procedure Calls

Example - Mach

- port_allocate
  - create a mailbox (owner)
  - port_status ~ .e.g, # of msgs in a link
- All messages have the same priority and are served in a FIFO fashion.
- Message Size
  - A fixed-length head + a variable-length data + two mailbox names
- Message copying: message copying → remapping of addressing space
- System calls are carried out by messages.

Example – Windows XP

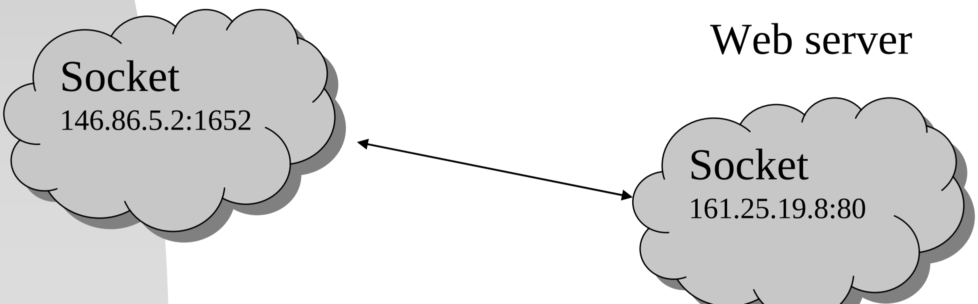
- Local Procedure Call (LPC) – Message Passing on the Same Processor
  1. The client opens a handle to a subsystem's *connection port* object.
  2. The client sends a connection request.
  3. The server creates two private *communication ports*, and returns the handle to one of them to the client.
  4. The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Example – Windows XP

- Two Types of Message Passing Techniques
  - Small messages (<= 256 bytes)
    - Message copying
  - Large messages – section object
    - To avoid memory copy
    - Sending and receiving of the pointer and size information of the object
- A callback mechanism
  - When a response could not be made immediately.

Communication in Client-Server Systems

- Socket
  - An endpoint for communication identified by an IP address concatenated with a port number
    - A client-server architecture



* /etc/services: Port # under 1024 ~ 23-telnet, 21-ftp, 80-web server, etc.

Communication in Client-Server Systems

- Three types of sockets in Java
  - Connection-oriented (TCP) – Socket class
  - Connectionless (UDP) – DatagramSocket class
  - MulticastSocket class – DatagramSocket subclass

Server

```
sock = new ServerSocket(5155);
…
client = sock.accept();
pout = new PrintWriter(client.getOutputStream(),
        true);
…
Pout.println(new java.util.Date().toString());
pout.close();
client.close();
```

Client

```
sock = new Socket("127.0.0.1",5155);
…
in = sock.getInputStream();
bin = new BufferReader(new
       InputStreamReader(in));
…
sock.close();
```

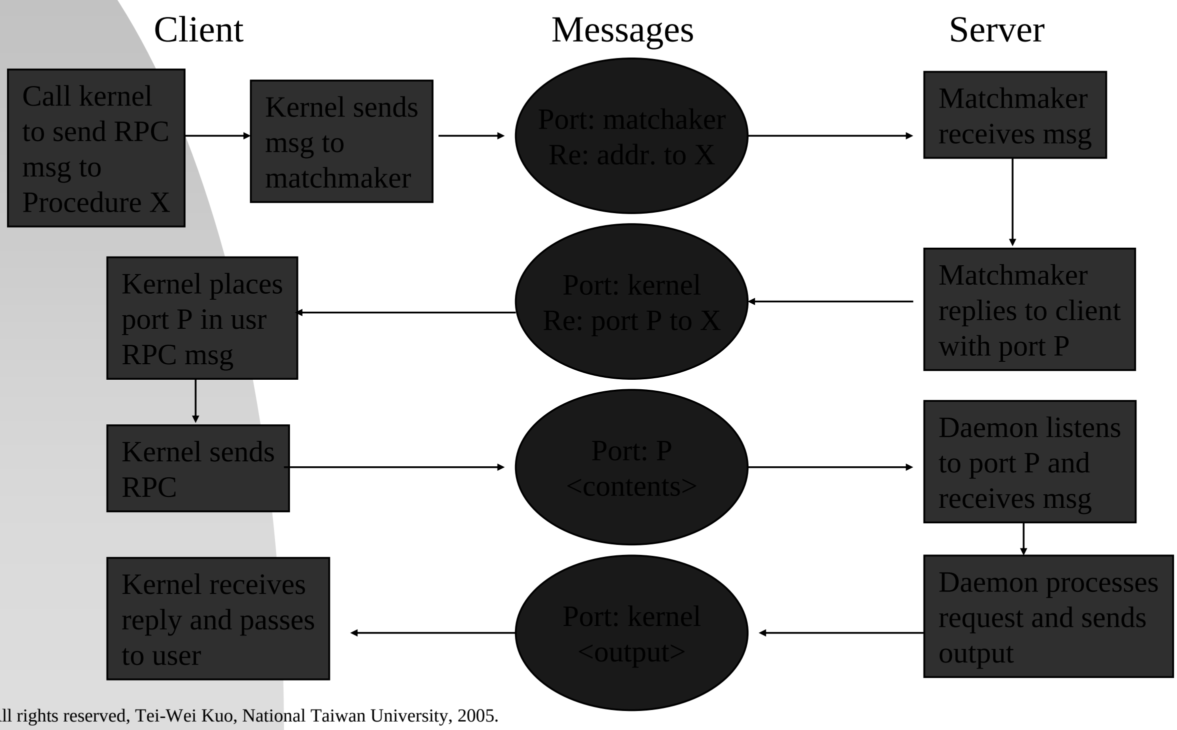
Communication in Client-Server Systems

- Remote Procedure Call (RPC)
  - A way to abstract the procedure-call mechanism for use between systems with network connection.
  - Needs:
    - Ports to listen from the RPC daemon site and to return results, identifiers of functions to call, parameters to pack, etc.
    - Stubs at the client site
      - One for each RPC
      - Locate the proper port and marshall parameters.

Communication in Client-Server Systems

- Needs (continued)
  - Stubs at the server site
    - Receive the message
    - Invoke the procedure and return the results.
- Issues for RPC
  - Data representation
    - External Data Representation (XDR)
      - Parameter marshalling
  - Semantics of a call
    - History of all messages processed
  - Binding of the client and server port
    - Matchmaker – a rendezvous mechanism

Communication in Client-Server Systems



Communication in Client-Server Systems

- An Example for RPC
  - A Distributed File System (DFS)
    - A set of RPC daemons and clients
    - DFS port on a server on which a file operation is to take place:
      - Disk operations: read, write, delete, status, etc – corresponding to usual system calls

Communication in Client-Server Systems

- Remote Method Invocation (RMI)
  - Allow a thread to invoke a method on a remote object.
    - boolean val = Server.someMethod(A,B)
- Implementation
  - Stub – a proxy for the remote object
    - Parcel – a method name and its marshalled parameters, etc.
  - Skeleton – for the unmarshalling of parameters and invocation of the method and the sending of a parcel back

Communication in Client-Server Systems

- Parameter Passing
  - Local (or Nonremote) Objects
    - Pass-by-copy – an object serialization
  - Remote Objects – Reside on a different Java virtual machine (JVM)
    - Pass-by-reference
  - Implementation of the interface – *java.io.Serializable*