Procesos

Semestre II-2013

Procesos

- Concepto de proceso
- Planificación de procesos
- Operaciones sobre procesos
- Comunicación entre procesos
- Ejemplos de un sistema con soporte de IPC
- Comunicación en sistemas Cliente-Servidor

Objetivos

- Introducir la noción de proceso – Un programa en ejecución, el cual básicamente es la base de la computación
- Describir las diferentes características de un proceso, incluyendo planificación, creación y terminación, y comunicación
- Explorar la comunicación entre procesos usando memoria compartida y pase de mensajes
- Describir la comunicación en un sistemas Cliente-Servidor

Concepto de Proceso

- Un SO ejecuta una gran variedad de programas:
  - Sistemas Batch – *jobs*
  - Sistemas de Tiempo-Compartido – Programas de usuario o *tasks*
  - En la bibliografía se utilizan los términos *job* y proceso se utilizan de forma indiferente
  - Proceso – Un programa en ejecución; la ejecución de un proceso debe ser progresiva y de forma secuencial

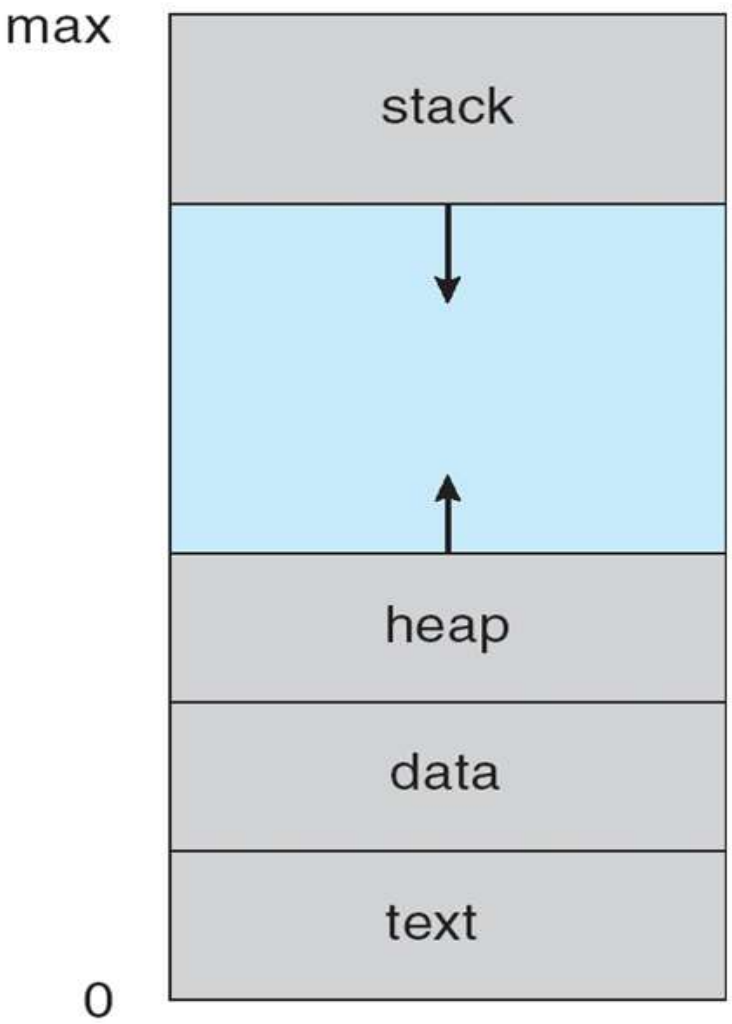
Concepto de Proceso

- Múltiples partes
  - El código del programa, también llamado sección de texto
  - El contador de programa, que registra la actividad actual junto a los registros del procesador
  - La pila, la cual contiene datos temporales
    - Parámetros pasados a una función, dirección de retorno, variables locales
  - La sección de datos, la cual contiene las variables globales
  - El *heap* que representan un porción de memoria a asignar de forma dinámica durante la ejecución

Concepto de Proceso

- Un programa en una entidad pasiva almacenada en disco (archivo ejecutable), mientras que el proceso es una entidad activa
  - Un programa se convierte en un proceso una vez que el archivo ejecutable se carga en memoria
- La ejecución de un programa inicia vía un click en una GUI, o mediante la ejecución de un comandos en un interprete de comandos, etc.
- Un programa puede desencadenar varios procesos
  - Considere diferentes usuarios ejecutando el mismo programa

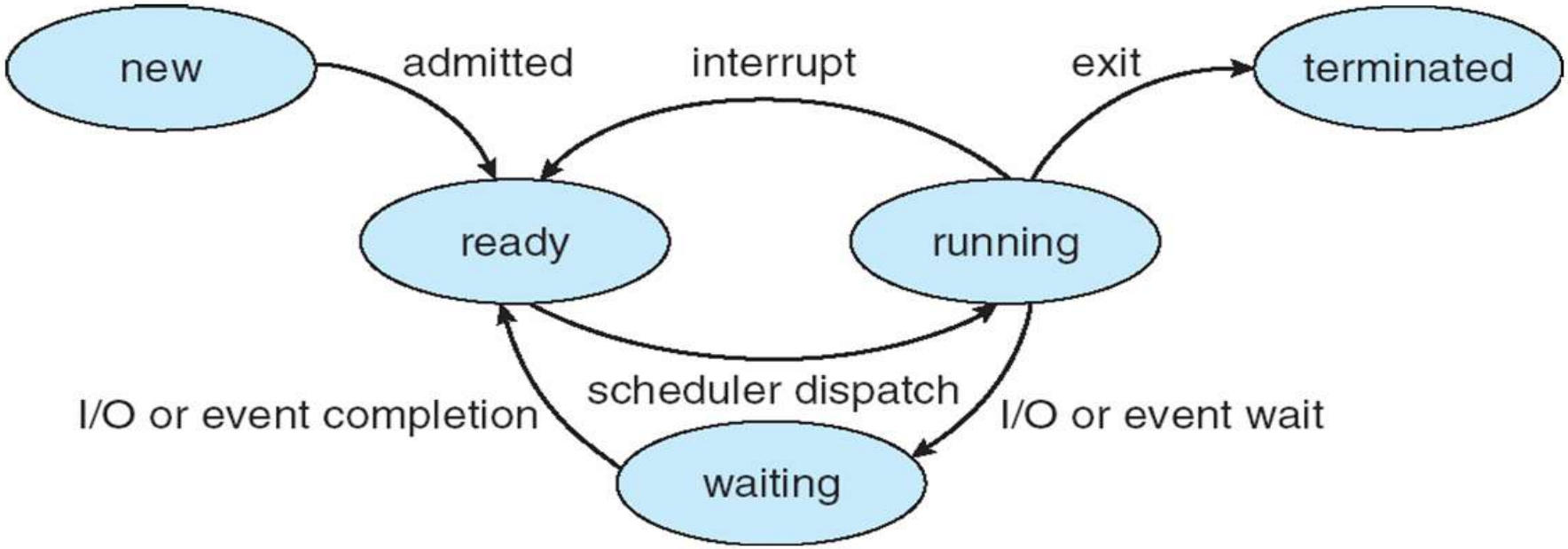
Un Proceso en Memoria



Estados de un Proceso

- Durante la ejecución de un proceso, este puede estar en los siguientes estados
  - New: El proceso ha sido creado
  - Running: Instrucciones del proceso están siendo ejecutadas
  - Waiting: El proceso se encuentra esperando a que algún evento ocurra
  - Ready: El proceso se encuentra esperando a que se le asigne el procesador
  - Terminated: El proceso ha finalizado su ejecución

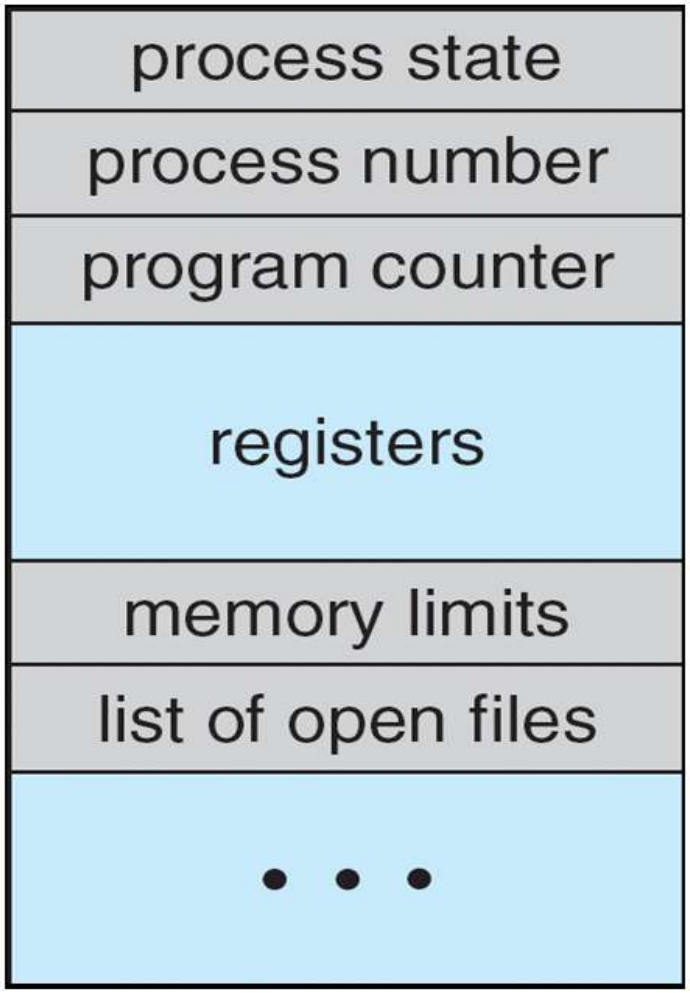
Diagrama de Estado de un Proceso



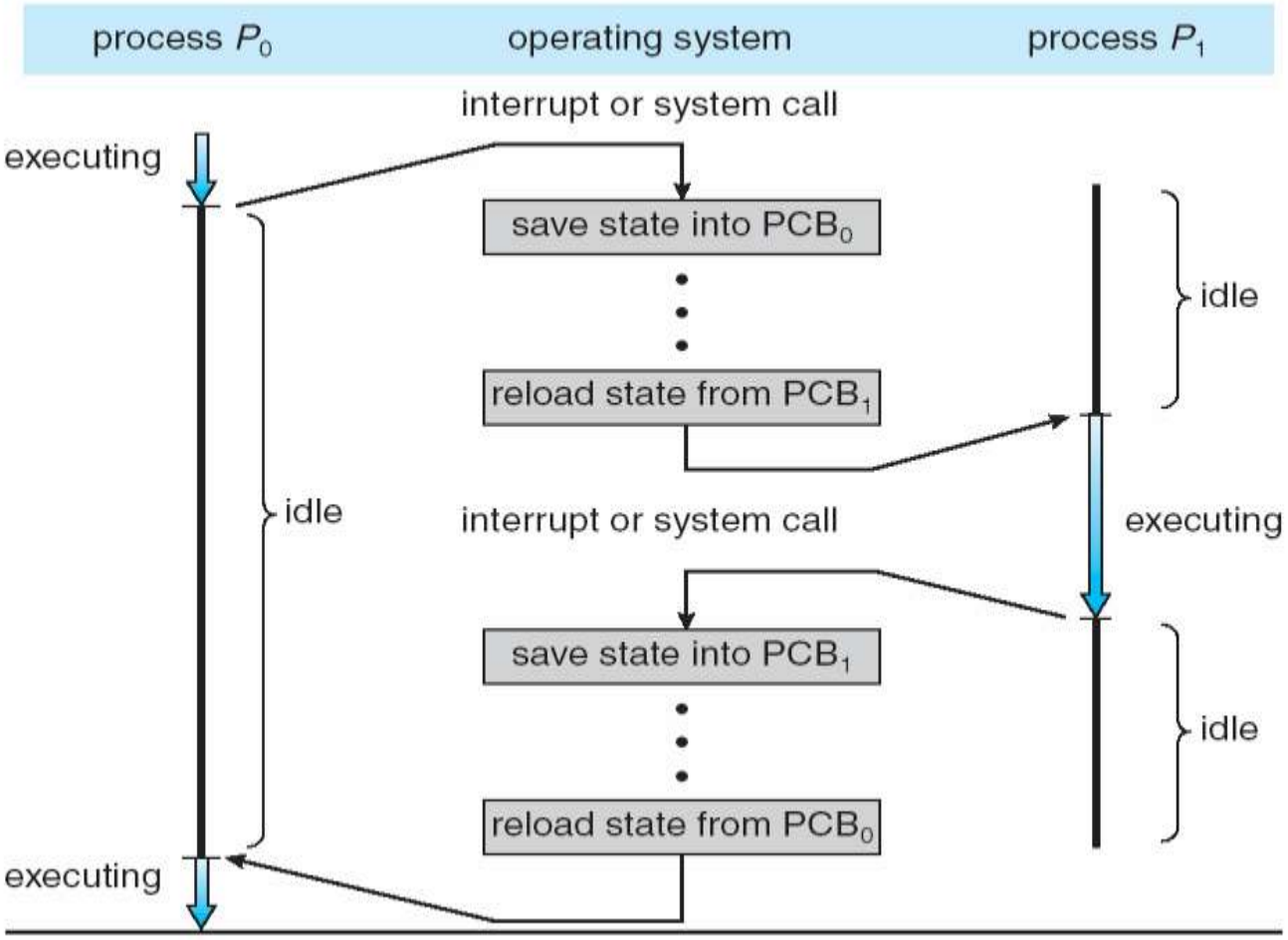
PCB (Process Control Block)

- Información asociada con cada proceso (en ocasiones llamada task control block)
  - Estado del proceso – Running, waiting, etc.
  - Contador de programa – Dirección de la próxima instrucción a ejecutar
  - Registros del CPU – Contenido de todos los registros del procesador
  - Información de planificación del CPU – Prioridades, apuntadores a las colas de planificación
  - Información de la administración de memoria – Memoria asignada al proceso
  - Información de contabilización – CPU usado, tiempos limites, hora de inicio y finalización
  - Información del estado de I/O – Dispositivos de I/O asignados al proceso, lista de archivos abiertos

PCB (Process Control Block)



Intercambio de Procesos en el CPU



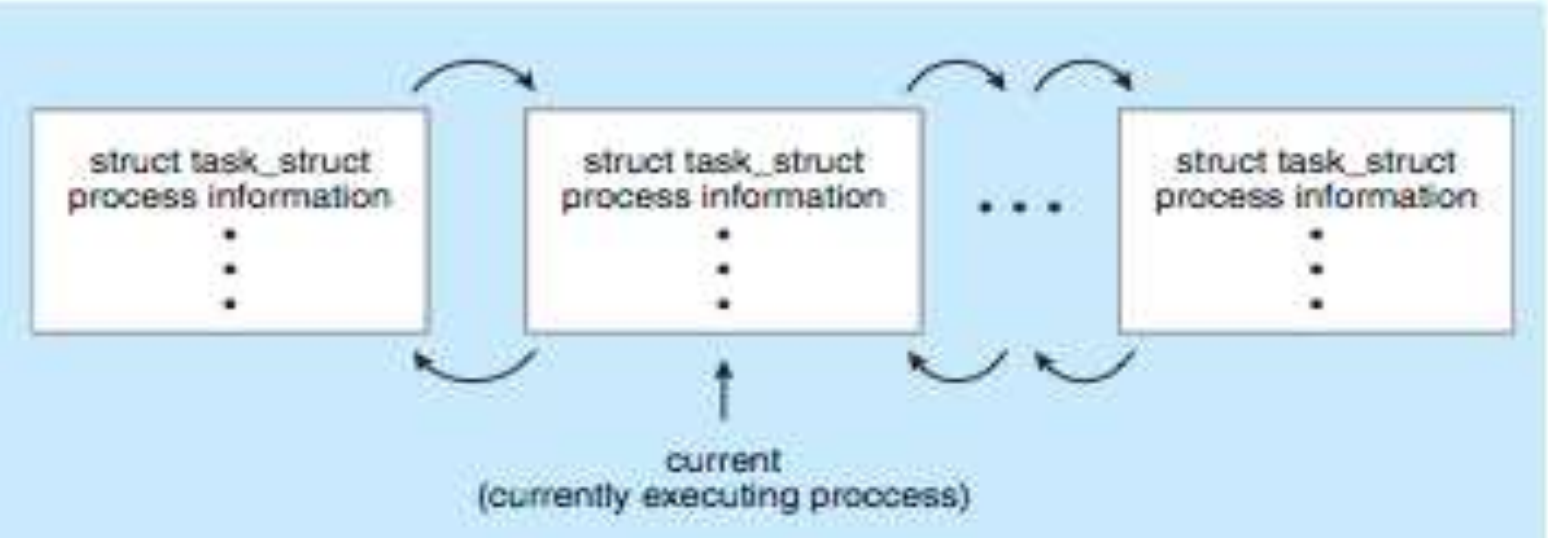
Threads

- Hace bastante tiempo, un proceso solo tenia un thread en ejecución
- Considere un programa que contenga múltiples contadores de programa por proceso
  - Diferentes instrucciones pueden ejecutarse a la vez
    - Múltiples threads de control → threads
- ¿Cómo es esto posible?
  - Es necesario almacenar todos los detalles concernientes a los threads, por ejemplo, múltiples PCBs
  - Siguiente capítulo

Representación de un Proceso en Linux

- Representado por la estructura `task_struct` en C

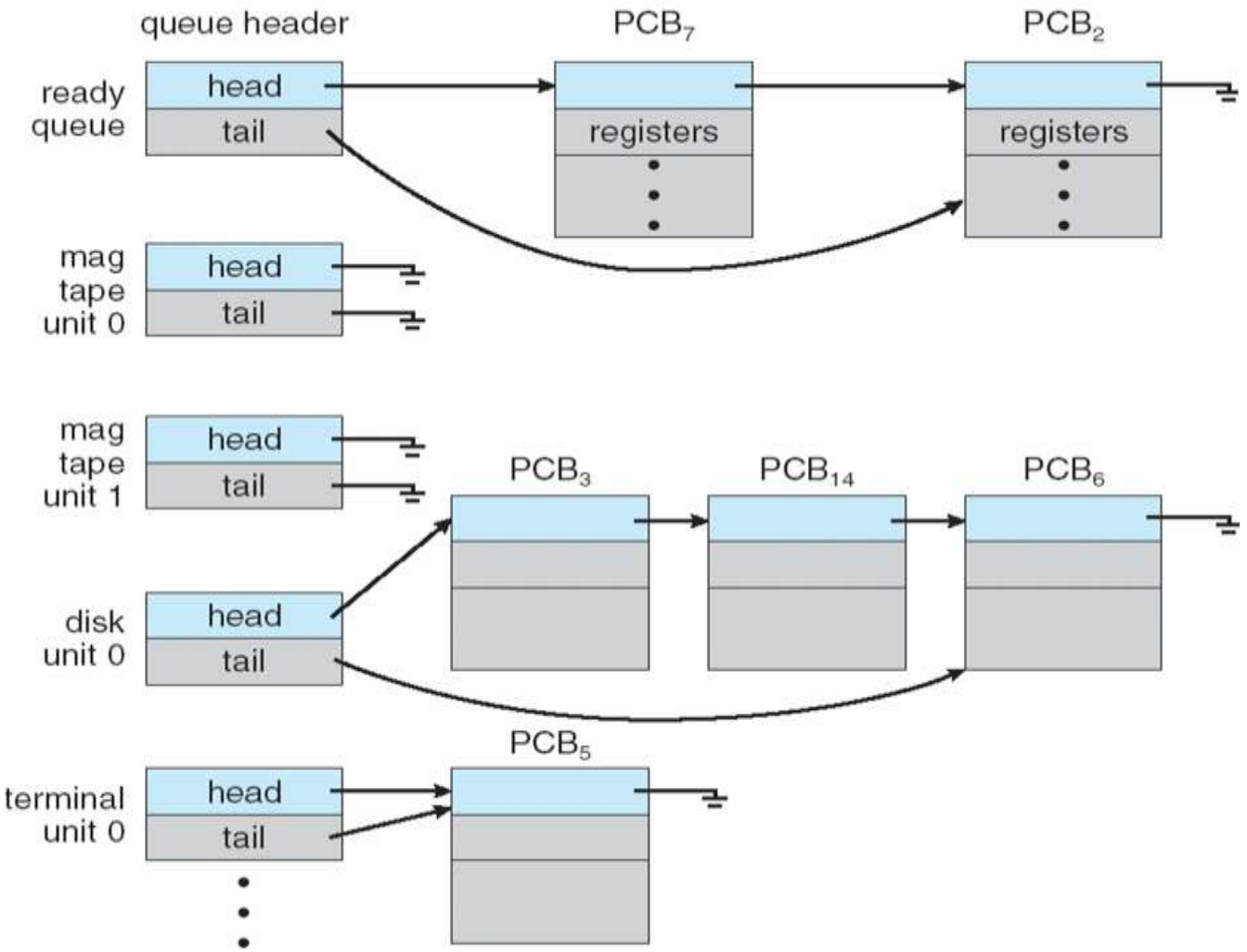
```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children; /* this process's children */
struct files struct *files; /* list of open files */
struct mm struct *mm; /* address space of this process */
```



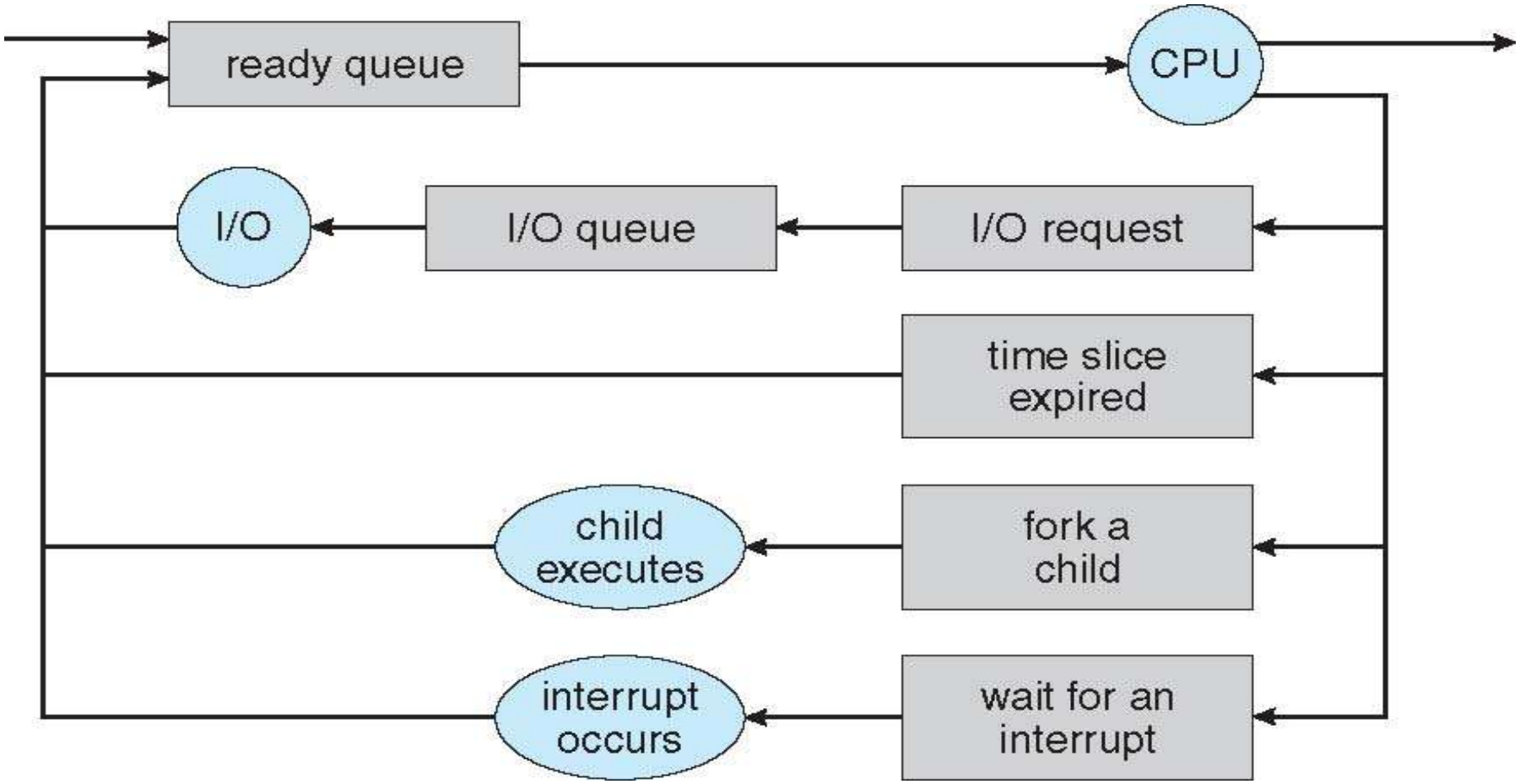
Planificación de Procesos

- Maximizar el uso del CPU, es necesario intercambiar rápidamente los procesos a nivel de CPU para garantizar un adecuado rendimiento a nivel de tiempo compartido
- La planificación de procesos se encarga de seleccionar el próximo proceso disponible para su ejecución en CPU
- Es necesario mantener colas de planificación de procesos
  - *Job queue* – Contiene a todos los procesos del sistema
  - *Ready queue* – Contiene a todos los procesos del sistema que residen en memoria
  - *Device queues* – Contiene a todos los procesos que se encuentran en espera de un dispositivo de I/O
  - Un proceso durante su vida puede migrar entre estas colas

Diferentes Colas en el SO



Representación de la Planificación de Procesos



Planificadores

- Planificador a Largo-Plazo *(job scheduler)* – Selecciona que proceso debe ser traído a la ready queue
- Planificador a Corto-Plazo *(CPU scheduler)* – Selecciona que procesos debe ser ejecutado y le asigna el CPU
  - En algunas ocasiones solo existe un planificador en el sistema
- El planificador a Corto-Plazo es invocado con mucha frecuencia (milisegundos) → Debe ser rápido
- El planificador a Largo-Plazo es invocado con poca frecuencia (segundos, minutos) → Podría ser lento

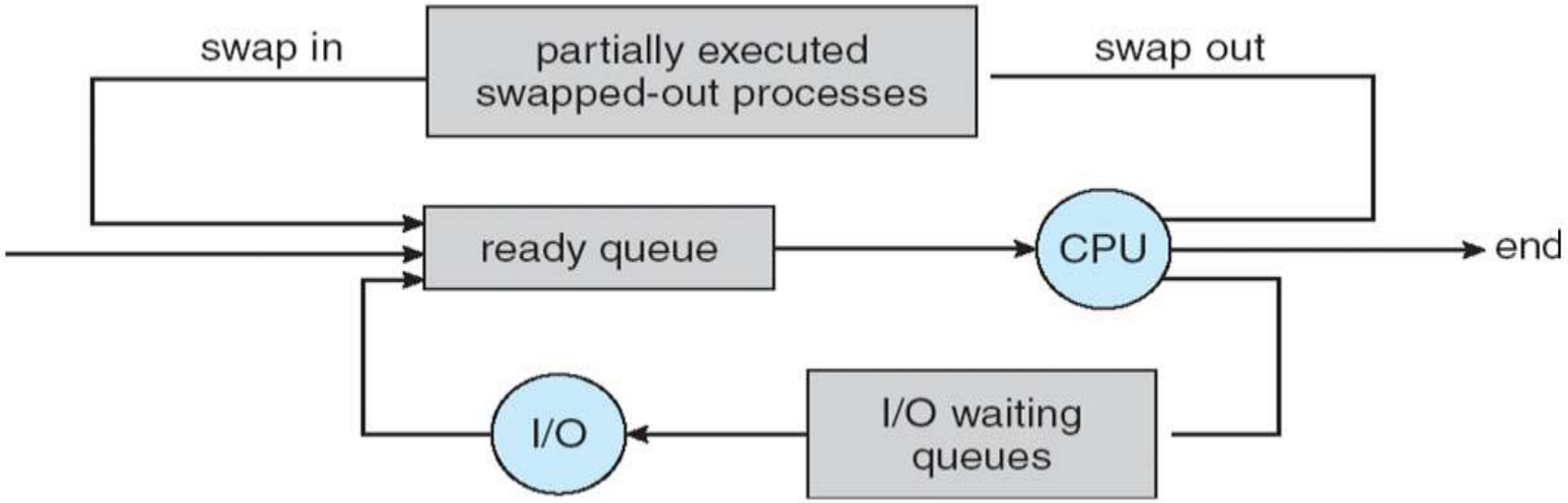
Planificadores

- El planificador a Largo-Plazo controla el grado de multiprogramación en el sistema
- Los procesos pueden describirse como:
  - Procesos I/O-bound – Invierten más tiempo realizando I/O que ejecutando instrucciones, es decir, cortas ráfagas de CPU
  - Procesos CPU-bound – Invierten más tiempo ejecutando instrucciones, lo cual implica unas cuantas ráfagas largas de CPU
- Un buen planificador a Largo-Plazo debería intercalar de manera adecuada procesos de ambas naturaleza

Planificador a Mediano-Plazo

- El planificador a Mediano-Plazo en caso de ser necesario la reducción el nivel de multiprogramación
  - ¿Cómo? – Ideas
  - Remover procesos de memoria principal, almacenarlos en disco, y traerlos de vueltas desde el disco para continuar con su ejecución
    - Swapping

Planificador a Mediano-Plazo



Multitasking en Sistemas Móviles

- Algunos sistemas / sistemas primitivos solo permitían un proceso en ejecución, mientras que los demás se encontraban suspendidos
- Debido al espacio disponible en pantalla, la interfaz de iOS presenta ciertas limitaciones
  - Solo un proceso en foreground – Controlado vía interfaz de usuario
  - Múltiples procesos en background – Cargados en memoria, ejecutándose, pero no visibles en la pantalla, y con un límite de procesos establecido
  - Las limitaciones incluyen tareas cortas las cuales reciben notificaciones o eventos, y solo algunas tareas de ejecución prolongada, como un reproductor de audio

Multitasking en Sistemas Móviles

- En Android los procesos pueden ejecutarse en foreground y background, con algunas limitaciones
  - Los procesos en background utilizan servicios para ejecutar tareas
  - Los servicios pueden continuar en ejecución incluso si el proceso en background se suspende
  - Los servicios no poseen una interfaz de usuario, utilizan un pequeña cantidad de memoria

Cambio de Contexto

- Cuando el CPU cambia de un proceso a otro, el sistema debe salvar el estado del proceso que va de salida y cargar el estado del proceso que va a ejecutarse. Esta acción se conoce como cambio de contexto
- El contexto de un procesos esta representado por su PCB
- Un cambio de contexto implica una sobrecarga en tiempo, dado que el sistema no se encuentra ejecutando un trabajo útil durante el intercambio
  - Mientras más complejo sea el SO y el PCB → Más largo será el cambio de contexto

Cambio de Contexto

- El tiempo que toma el cambio de contexto dependen del soporte de hardware
  - Algunos hardware proveen múltiples conjuntos de registros por CPU
    - Múltiples contextos pueden cargarse a la vez

Operaciones sobre Procesos

- El sistema debe proveer mecanismos
  - Creación de procesos
  - Terminación de proceso
  - Y más
    - Ideas

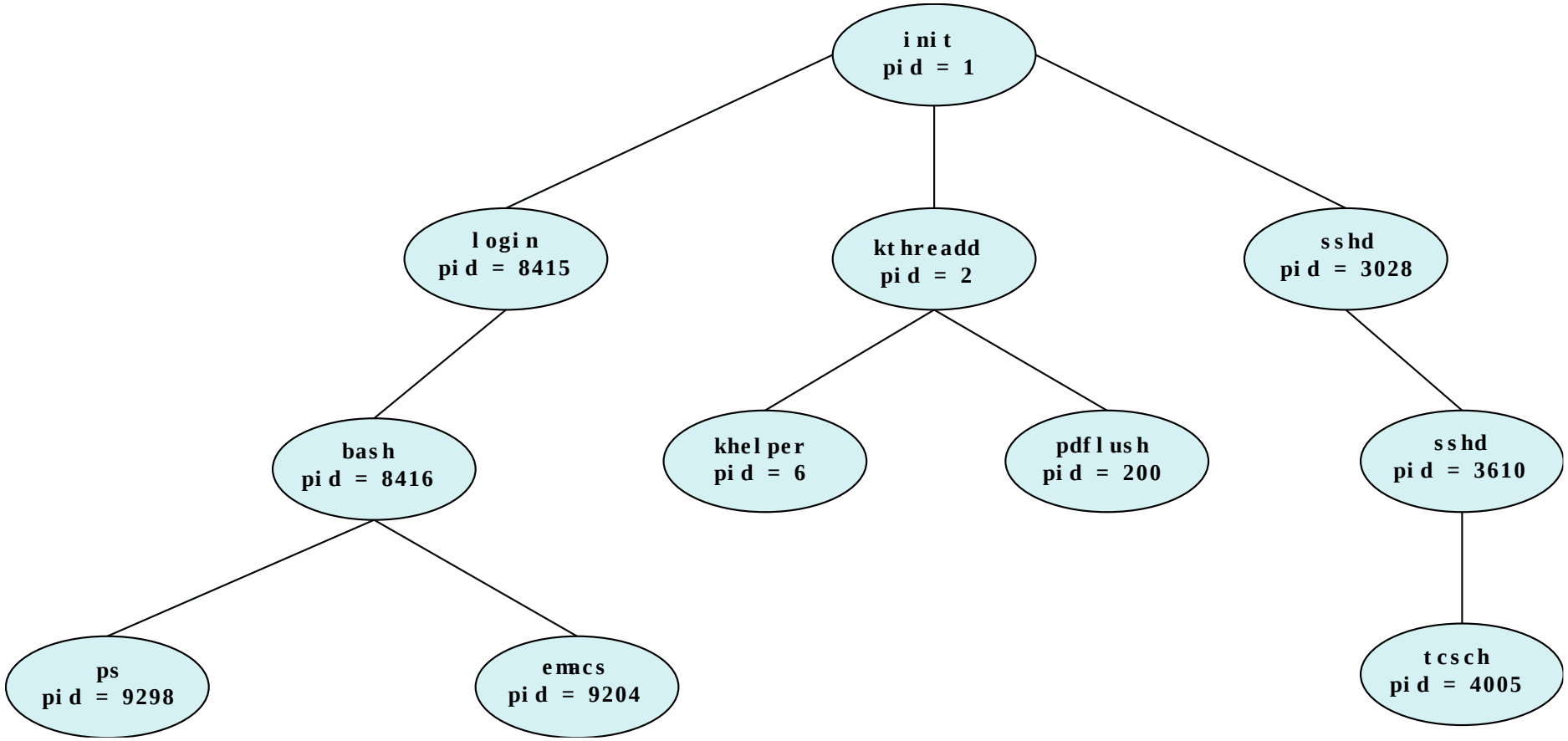
Creación de Procesos

- Un proceso padre puede crear procesos hijos, los cuales a su vez pueden crear otros procesos, formando un árbol de procesos
- Generalmente, los procesos se identifican y manejan vía su identificador de proceso (PID – Process Identifier)

Creación de Procesos

- Opciones en la compartición de recursos
  - Padre e hijo comparten todos los recursos
  - El hijo puede compartir un subconjunto de los recursos del padre
  - Padre e hijo no comparan recursos
- Opciones de ejecución
  - Padre e hijo se ejecutan de manera concurrente
  - El padre espera hasta que su hijo termine

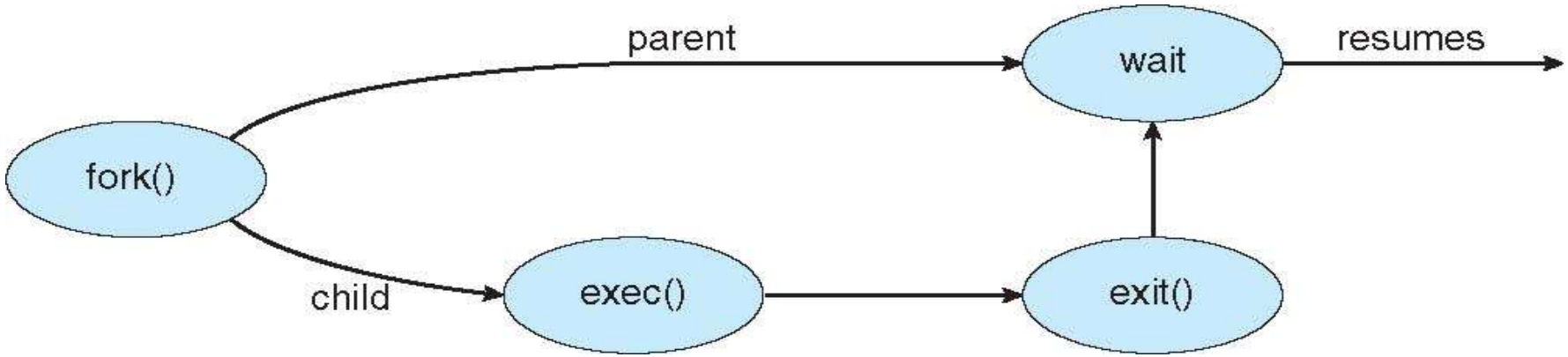
Árbol de Procesos en Linux



Creación de Procesos

- Espacio de direcciones
  - El hijo duplica el espacio de direcciones del padre
  - El hijo carga un nuevo programa en el espacio de direcciones
- Ejemplos en UNIX
  - fork() – Llamada al sistema que crea un nuevo proceso
  - exec () – Llamada al sistema usada después del fork() para remplazar el espacio de direcciones del invocador

Creación de Procesos



Ejemplo de Creación de Procesos

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Ejemplo de Creación de Procesos

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     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;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Terminación de Procesos

- Procesos que ejecutan su última instrucción e indican al SO que pueden ser borrados (exit())
  - Si se desea esperar por los datos de cualquier hijo es necesario esperar por su terminación (wait())
  - Los recursos del procesos son liberados por el SO
- El padre puede terminar la ejecución de sus procesos hijos (abort())
  - El hijo ha excedido los recursos asignados
  - La tarea asignada al hijo ya no es necesaria
  - Si el proceso padre aborta
    - Algunos SO no permiten la ejecución de los hijos si el padre a terminado
      - Terminación en cascada

Terminación de Procesos

- Para esperar la terminación, es vital retornar el pid:
  - pid t pid; int status;
  - pid = wait(&status);
- Si no hay un padre esperando, cuando los proceso hijos terminan son zombies
- Si el padre termina, los procesos hijos son huérfanos

Arquitectura Multiprocesos – Chrome Browser

- Muchos browsers se ejecutan como un solo proceso
  - Si uno de los websites ocasiona un problema, el browser completo sufre los inconvenientes
- Google Chrome Browser es multiprocesos en tres diferentes categorías
  - El proceso navegador gestiona la interfaz de usuario, el disco y la I/O de red
  - El proceso render renderisa las páginas web, lidia con HTML, Javascripts, y se crea un nuevo proceso por cada website abierto
    - Se ejecuta en un sandbox que se encuentra restringida en acceso a disco y I/O de red, minimizando el efecto de cualquier fallo
  - El proceso plugin lidia con los diferentes tipos de plugins

Arquitectura Multiprocesos – Chrome Browser

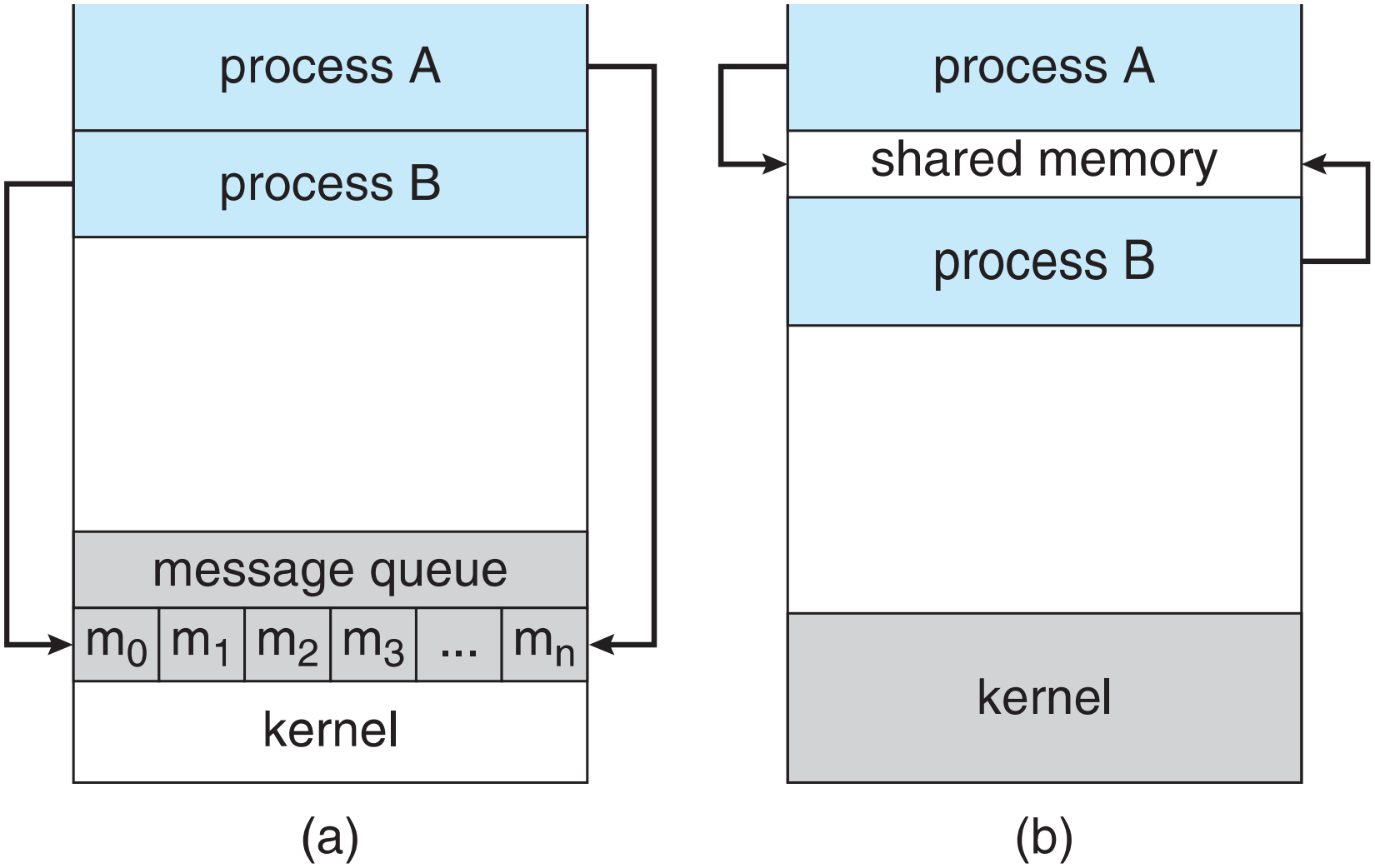
Comunicación entre Procesos

- Los procesos dentro del sistema pueden ser independientes o cooperativos
- Los procesos cooperantes pueden afectar o verse afectados por otros procesos, incluyendo la compartición de datos
- Razones por la que es necesario lidiar con procesos cooperativos
  - Compartición de información
  - Incremento de la eficiencia en labores computacionales
  - Modularidad
  - Conveniencia

Comunicación entre Procesos

- Los procesos cooperantes necesitan de un protocolo de comunicación
  - Comunicación entre procesos
    - Interprocess Communications – IPC
- Dos modelos de IPC
  - Memoria compartida
  - Pase de mensajes

Modelos de Comunicación



Cooperación entre Procesos

- Independientes – Los procesos no afectan o se ven afectados por la ejecución de otros procesos
- Cooperantes – Los procesos pueden afectar o verse afectados por la ejecución de otros procesos

Solución Memoria-Compartida (Bounded-Buffer)

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

(Bounded-Buffer) - Productor

```
item next produced;
while (true) {
     /* produce an item in next produced */
     while (((in + 1) % BUFFER SIZE) == out)
          ; /* do nothing */
     buffer[in] = next produced;
     in = (in + 1) % BUFFER SIZE;
}
```

(Bounded-Buffer) - Consumidor

```
item next consumed;
while (true) {
     while (in == out)
          ; /* do nothing */
     next consumed = buffer[out];
     out = (out + 1) % BUFFER SIZE;

     /* consume the item in next consumed */
}
```

IPC – Pase de Mensajes

- Mecanismo para que los procesos puedan comunicarse y sincronizar sus acciones
- Los procesos se comunican entre si sin la necesidad de variables compartidas
- Este tipo de IPC provee dos operaciones:
  - send(message) – Envía un mensaje fijo o variable
  - receive(message)
- Si P y Q desean comunicarse, ellos necesitan:
  - Establecer un enlace de comunicación entre ellos
  - Intercambiar mensajes vía send/receive

IPC – Pase de Mensajes

- La implementación del enlace de comunicación
  - Físico (p.e: memoria compartida, bus del hardware)
  - Lógico (p.e: directo o indirecto, síncrono o asíncrono, usando buffers automáticos o explícitos)

Preguntas de Implementación

- ¿Cómo se establece el enlace?
- ¿Puede un enlace asociarse con más de dos procesos?
- ¿Cuántos enlaces puede haber entre cada par de procesos que se comunican?
- ¿Cuál es la capacidad de un enlace?
- ¿El tamaño del mensaje es fijo o variable?
- ¿Un enlace es unidireccional o bidireccional?

Comunicación Directa

- Los procesos deben nombrarse explícitamente
  - send (P, message) – Envía un mensaje al proceso P
  - receive (Q, message) – Recibe un mensaje del proceso Q
- Propiedades del enlace de comunicación
  - El enlace se establece automáticamente
  - Un enlace es asociado con exactamente un par de procesos en comunicación
  - Entre cada par de procesos existe exactamente un enlace
  - El enlace puede ser unidireccional, pero usualmente el bidireccional

Comunicación Indirecta

- Los mensajes son dirigidos y recibidos por mailboxes (conocidos también como puertos)
  - Cada mailbox tiene un único id
  - Los procesos pueden comunicarse solo si poseen un mailbox compartido
- Propiedades del enlace de comunicación
  - El enlace se establece solo si el proceso comparte un mailbox común
  - Un enlace puede asociarse con varios procesos
  - Cada par de enlaces puede compartir varios enlaces de comunicación
  - El enlace puede ser unidireccional o bidireccional

Comunicación Indirecta

- Operaciones
  - Crear un nuevo mailbox
  - Enviar y recibir mensajes a través del mailbox
  - Destruir el mailbox
- Las siguientes primitivas están definidas:
  - send (A, message) – Enviar un mensaje al mailbox A
  - receive (A, message) – Recibir un mensaje desde el mailbox A

Comunicación Indirecta

- Compartición de mailbox
  - P1, P2, y P3 comparten el mailbox A
  - P1, envía, P2 y P3 reciben
  - ¿Quién obtiene el mensaje?
- Soluciones
  - Permitir que un enlace este asociado máximo con dos procesos
  - Permitir que solo un proceso ejecuta la operación receive a la vez
  - Permitir que el sistema seleccione de manera arbitraria al receptor. El emisor debe ser notificado sobre la identidad del proceso que recibo el proceso

Sincronización

- El intercambio de mensaje puede ser bloqueante o no bloqueante
- Bloqueante se considera síncrono
  - Un send bloqueante implica que el emisor se bloquea hasta que el mensaje es recibido
  - Un receive bloqueante implica que el receptor se bloquea hasta que el mensaje esta disponible
- No bloqueante se considera asíncrono
  - Un send no bloqueante implica que el emisor se envía el mensaje y continua
  - Un receive no bloqueante implica que el receptor pude recibir un mensaje válido o nulo

Sincronización

- Diferente combinaciones son posibles
  - Si tanto el send como el receive son bloqueando, se dice que se esta en presencia de un rendezvous
  - El Productor-Consumir se vuelve trivial

```
message next produced;
while (true) {
    /* produce an item in next produced */
   send(next produced);
}

message next consumed;
while (true) {
   receive(next consumed);

   /* consume the item in next consumed */
}
```

Buffering

- La cola de mensajes adjunta a un enlace puede ser implementada de la siguiente manera:
  - Capacidad cero → 0 mensajes
    - El emisor debe esperar por el receptor (rendezvous)
  - Capacidad limitada → Longitud finita de n mensajes
    - El emisor debe esperar si el enlace esta lleno
  - Capacidad ilimitada → Longitud infinita
    - El emisor nunca debe esperar

Ejemplo IPC - POSIX

- POSIX memoria compartida
  - Primero el proceso crea un segmento de memoria compartida
    - shm_fd = shm_open(name, O CREAT | O RDRW, 0666);
  - Esta codificación también puede usarse para abrir un segmento existente y compartirlo
  - Se establece el tamaño del objeto
    - ftruncate(shm fd, 4096);
  - Ahora el proceso puede escribir en la memoria compartida
    - sprintf(shared memory, "Writing to shared memory");

IPC POSIX - Productor

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

IPC POSIX – Consumidor

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Ejemplo IPC - Mach

- La comunicación en Mach se basa en mensajes
  - Incluso las llamadas al sistema son mensajes
  - Cada tarea tiene dos mailboxes desde su creación – Kernel y Notificación
  - Solo se necesitan tres llamadas al sistema para la transferencia de mensajes
    - msg_send(); msg_receive(); msg_rpc()
  - Los mailboxes necesarios para la comunicación se crean vía
    - port_allocate()

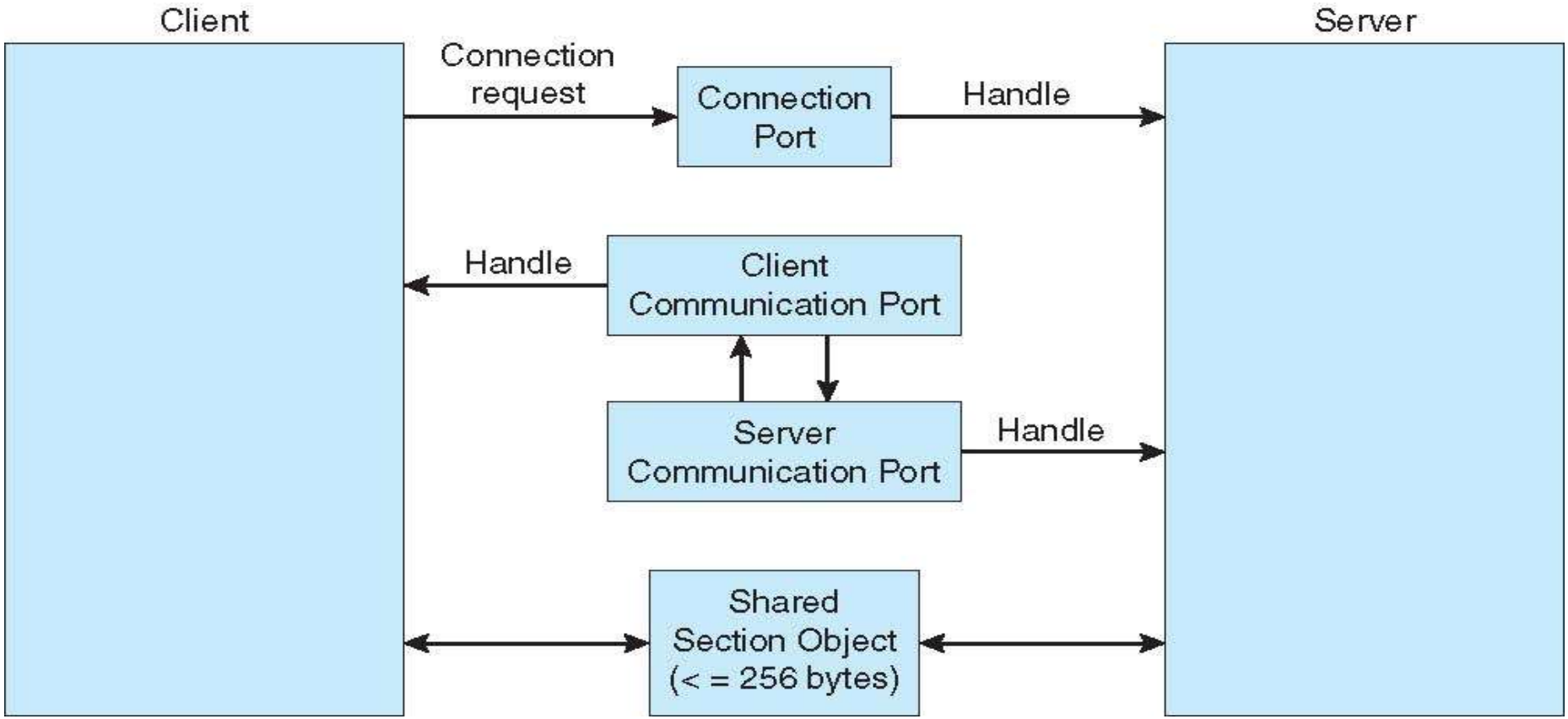
Ejemplo IPC - Mach

- Enviar y recibir son flexibles. Por ejemplo, existen cuatro opciones para un mailbox lleno
  - Esperar de forma indefinida
  - Esperar a lo sumo n milisegundos
  - Retornar de forma inmediata
  - Almacenar un mensaje temporalmente en cache

Ejemplo IPC - Windows

- El pase de mensajes se centra en el soporte de LPC (Advanced Local Procedure Call)
  - Solo funciona entre procesos del mismo sistema
  - Usa puertos (mailboxes) para establecer y mantener canales de comunicación
  - La comunicación funciona de la siguiente manera:
    - El cliente abre un manejador al puerto de conexión del subsistema
    - El cliente envía una solicitud de conexión
    - El servidor crea dos puertos de comunicación privados y retorna el manejador de uno de ellos al cliente
    - El cliente y el servidor usan los manejadores correspondientes para enviar mensajes y recibir respuestas

LPC – Windows XP



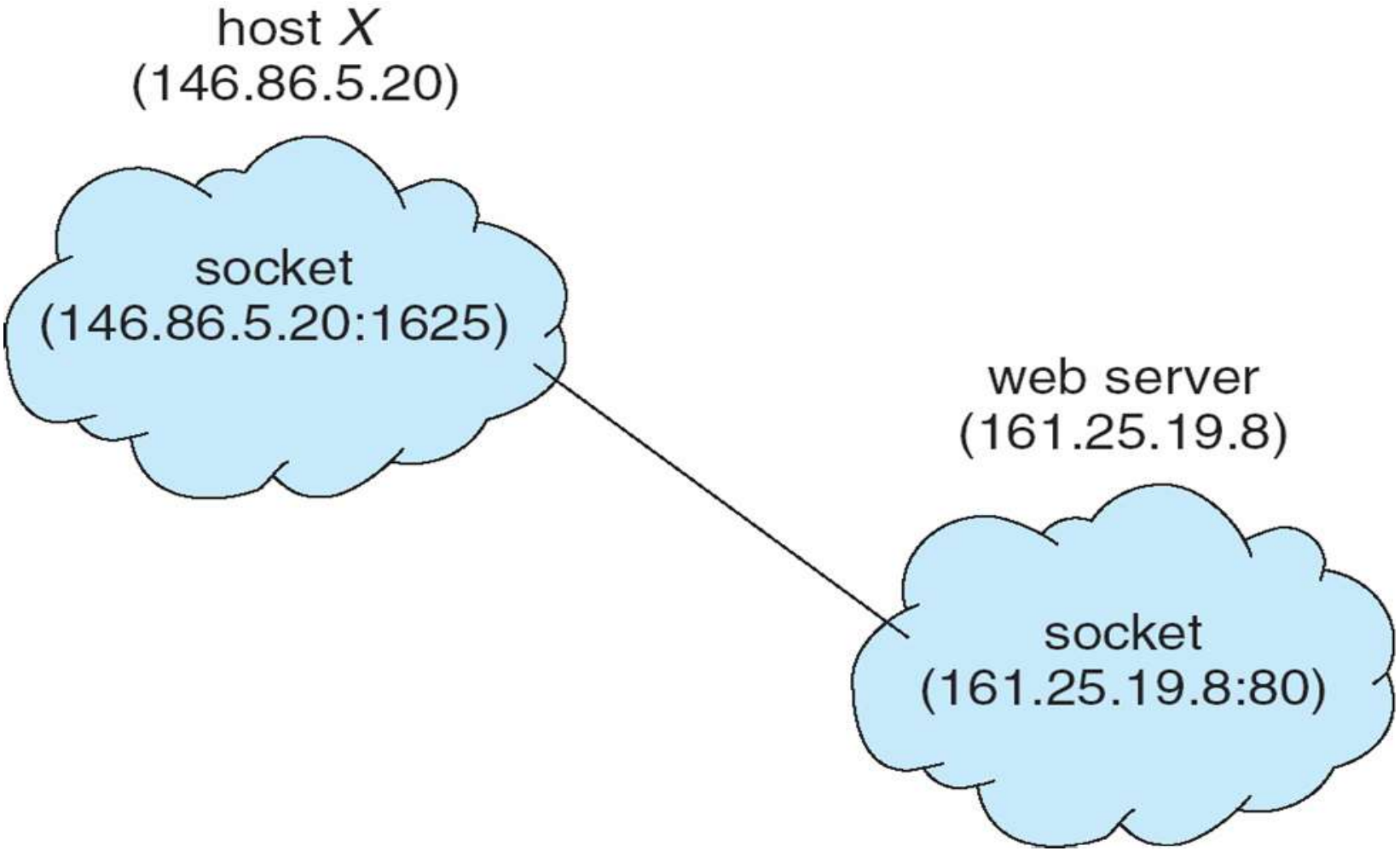
Comunicación en Sistemas Cliente-Servidor

- Sockets
- Pipes

Sockets

- Un sockets es un extremo para la comunicación
- Concatenación de dirección IP y puerto – Un número incluido al comienzo del mensaje para diferenciar los servicios de red en un host
- Formato <dirección>:<puerto>
- La comunicación se da entre un par de sockets
- Todos los puertos por debajo del 1024 se llaman bien conocidos, y se usan en servicios básicos
- La dirección IP 127.0.0.1 (loopback) se refiere al sistema local

Comunicación con Sockets



Socket en Java

- Tres tipos de sockets
  - Orientados a conexión
  - No orientado a conexión
  - Multicast

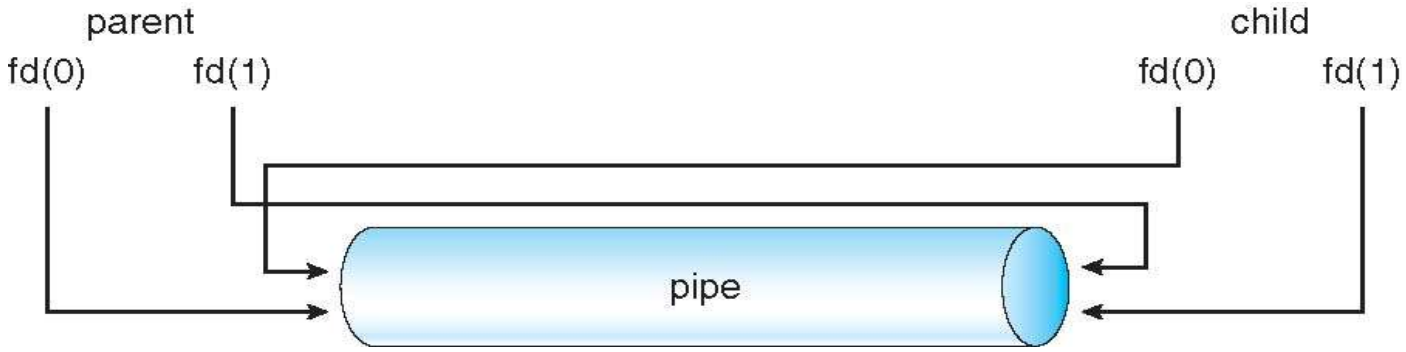
```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Pipes

- Actúa como un conducto permitiendo la comunicación entre procesos
- Consideraciones
  - ¿La comunicación es unidireccional o bidireccional?
  - ¿En caso de comunicación bidireccional, es halfduplex o fullduplex?
  - ¿Debe existir una relación entre los procesos?
  - ¿Los pipes se pueden usar sobre una red?

Pipes Ordinarios

- Permiten la comunicación estilo Productor-Consumidor
- El productor escribe en un extremo
- El consumidor lee del otro extremo
- Son unidireccionales
- Requieren una relación Padre-Hijo entre los procesos
- Windows los llama pipes anónimos



Pipes Nombrados

- Más poderosos que los ordinarios
- La comunicación es bidireccional
- No se necesita relación Padre-Hijo entre los procesos
- Varios procesos pueden usar el pipe nombrado para la comunicación
- Provisto en sistemas UNIX y Windows