Procesos

Módulo 3

Departamento de Informática
Facultad de Ingeniería
Universidad Nacional de la Patagonia “San Juan Bosco”

Procesos

- Concepto de Proceso
- Planificación de Proceso
- Operaciones sobre Procesos
- Comunicaciones Interprocesos (IPC)
- Ejemplos de Sistemas de IPC
- Comunicación en un Sistema Cliente-Servidor

Objetivos

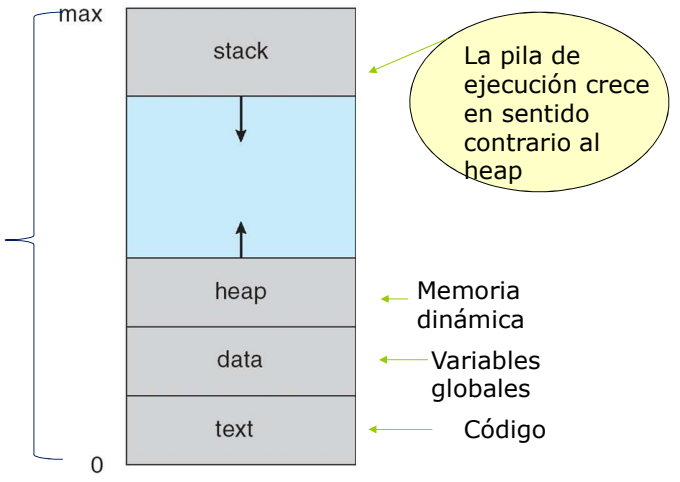
- Introducir la noción de **proceso** – un programa en ejecución, el cual conforma la base de toda computación
- Describir varias características de procesos, incluyendo planificación, creación, comunicación y terminación
- Describir la comunicación en sistemas cliente-servidor

Concepto de Proceso

- Sistema → colección de procesos: - del *SO* y del *usuario*. TODOS se ejecutan concurrentemente.
- Los términos *job, tarea* y *proceso* se usan con sentido similar.
- **Proceso** es un programa en ejecución; la ejecución debe avanzar en forma secuencial.
- Un proceso incluye:
  - contador de programa
  - stack
  - sección de datos

Organización de la memoria interna)

- Espacio de direcciones del proceso



ESTADOS DE UN PROCESO

- Cuando un proceso se ejecuta, cambia de *estado*
  - nuevo: es creado.
  - corriendo: está ejecutando las instrucciones.
  - espera: está esperando que ocurra algún evento.
  - listo: está esperando ser asignado a la CPU.
  - terminado: ha finalizado su ejecución.

Diagrama de Estados de un Proceso

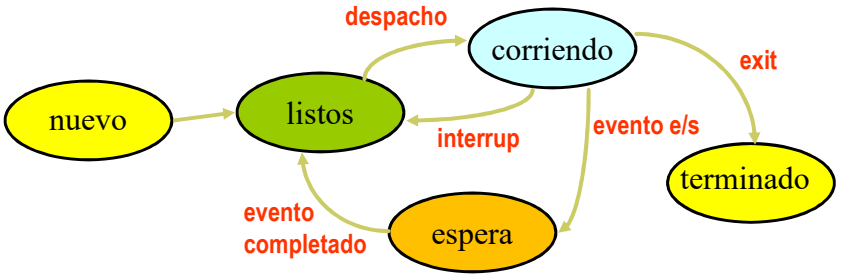
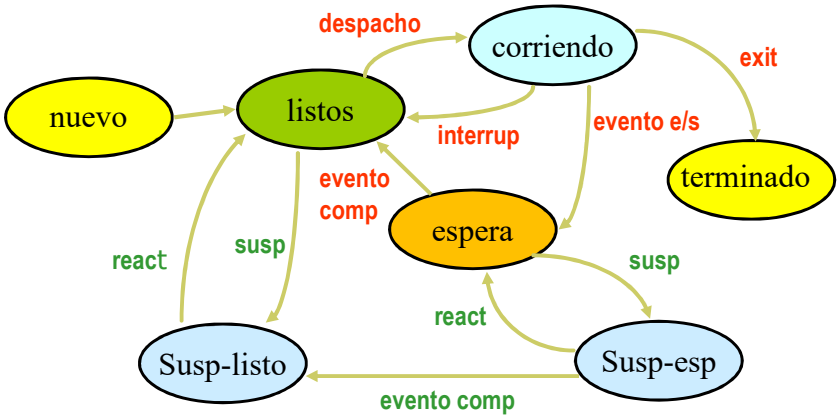


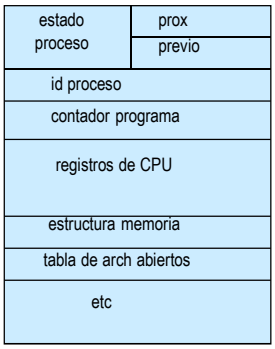
DIAGRAMA DE ESTADOS
5 Estados



Bloque de Control de Procesos (PCB)

Estructura de dato que contiene información asociada con cada proceso.

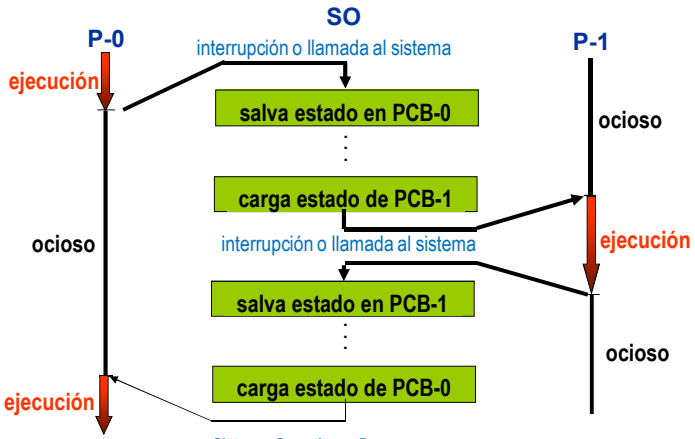
- Nombre del proceso (ID)
- Prioridad
- Estado de Proceso
- Contador de Programa
- Registros de CPU
- Información de planificación de CPU
- Información de administración de memoria
- Información contable
- Información de estado E/S



PCB: Process Control Block

CAMBIO DE CONTEXTO

- El contexto de un proceso está representado en el PCB
- El tiempo que lleva el cambio de contexto es sobrecarga; el sistema no hace trabajo útil mientras está conmutando.



COLAS DE PROCESOS

- Cola de Job (o tareas) – conjunto de todos los procesos en el sistema.
- Cola de listos – conjunto de todos los procesos que residen en memoria principal y están listos esperando ser ejecutados.
- Colas de dispositivos – conjunto de procesos esperando por una E/S en un dispositivo de E/S.

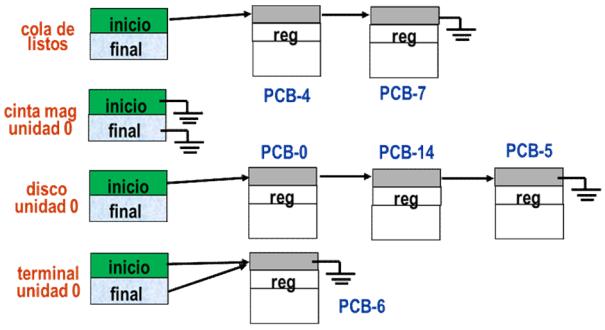
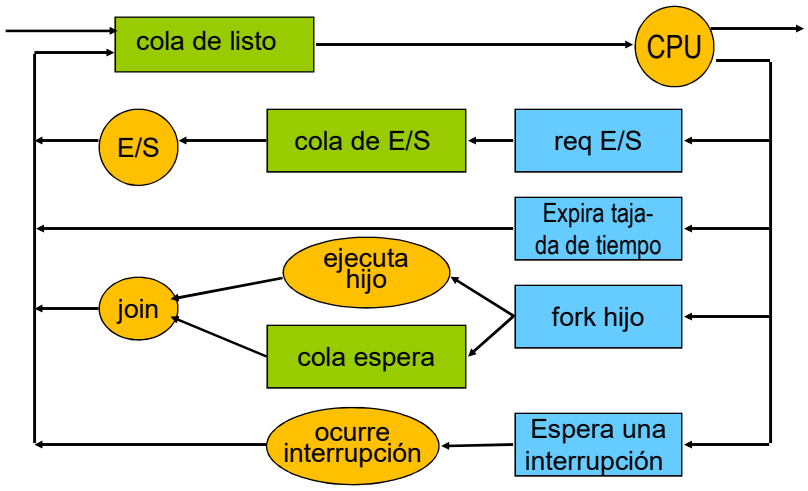
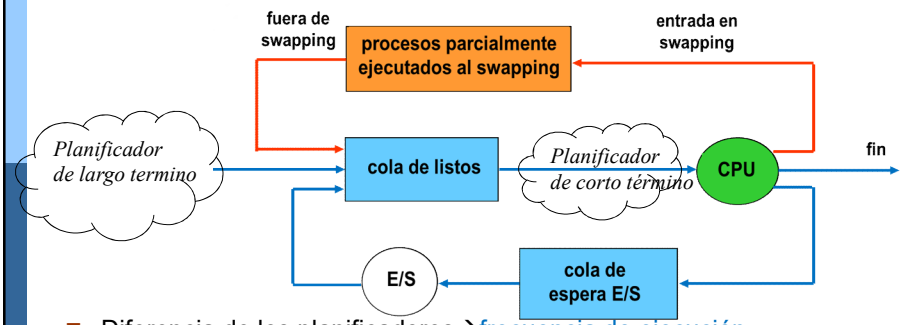


DIAGRAMA DE COLAS



PLANIFICADORES DE PROCESOS

- **Planificador de largo término** (o planificador de jobs)
- **Planificador de corto término** (o planificador de CPU).
- **Planificador de mediano término** realiza (swapping), para liberar a la memoria principal y reducir el grado de multiprogramación y también mejorar la mezcla de procesos.



- Diferencia de los planificadores→frecuencia de ejecución

Planificadores de Procesos (Cont.)

- El planificador de corto término es invocado muy frecuentemente (milisegundos) ⇒ (debe ser rápido).
- El planificador de largo término es invocado menos frecuentemente (segundos, minutos) ⇒ (puede ser muy lento).
- El planificador de largo término controla el *grado de multiprogramación*.
- Los procesos pueden ser descriptos como:
  - ***Procesos limitados por E/S*** – pasa mas tiempo haciendo E/S que computaciones, ráfagas (burst) de CPU muy cortas.
  - ***Procesos limitados por CPU*** – pasa mas tiempo haciendo computaciones que E/S, ráfagas (burst) de CPU muy largas.

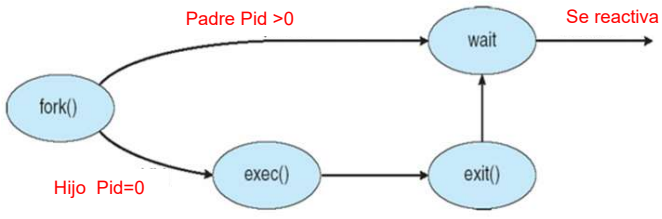
CREACIÓN DE PROCESOS

- Un proceso es creado **solo** por un padre.
- El hijo además crea otros procesos formando un *árbol de procesos*.
- **Recursos compartidos**
  - Padre divide todos sus recursos entre sus hijos.
  - Hijo comparte un subconjunto de los recursos del padre.
  - Padre e hijo no comparten ningún recurso.
- **Ejecución**
  - Padre e hijos ejecutan concurrentemente.
  - Padre espera hasta que los hijos terminen.
- **Espacio de direcciones**
  - El hijo es un duplicado del padre.(usa el mismo programa y datos)
  - El hijo carga un nuevo programa.

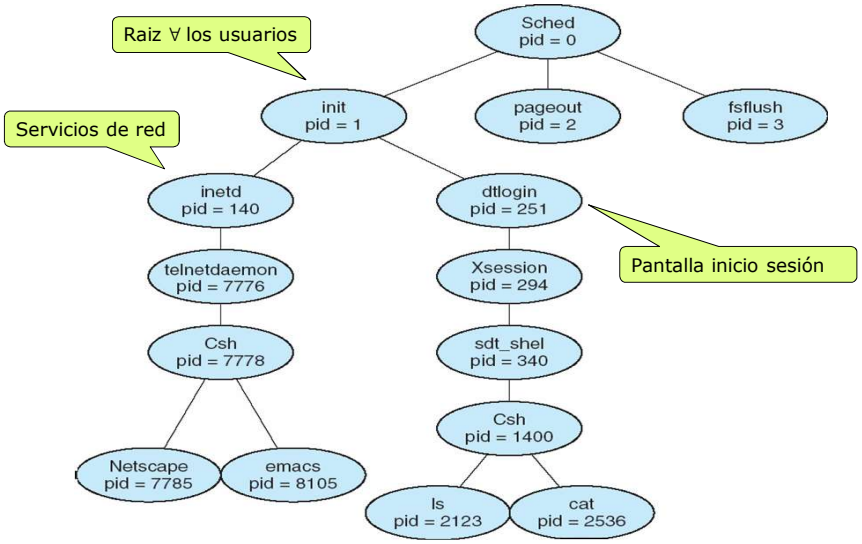
CREACIÓN DE PROCESOS

En UNIX

- La llamada a sistema **fork** crea un nuevo proceso.
- La llamada a sistema **exec** es usada después del **fork** para reemplazar el espacio de memoria del proceso con un nuevo programa.

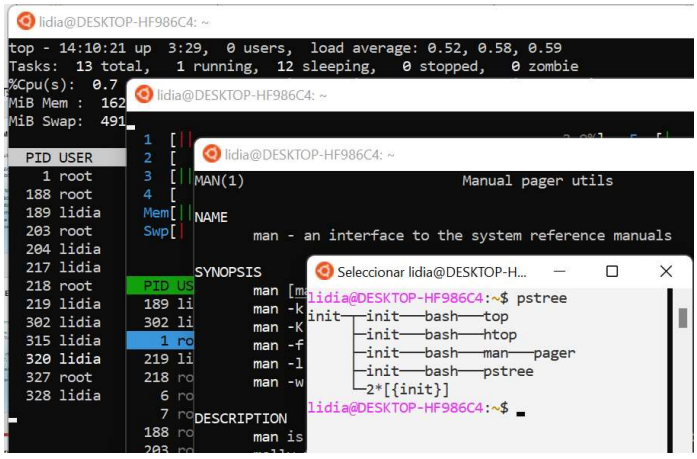


Ejemplo: Arbol de Procesos en Solaris



ARBOL DE PROCESOS

- **pstree** → obtener el árbol de creación de procesos del sistema.



TERMINACIÓN DE PROCESOS

Normal

- El proceso ejecuta la última sentencia (**exit**).
  - El padre espera la terminación del hijo con (**wait**).
  - El SO libera todos los recursos asignados del proceso.

Anormal

- El padre puede terminar la ejecución del proceso hijo (**abort**).
  - El hijo ha excedido el uso de algunos recursos.
  - La tarea asignada al hijo no es necesaria.
  - El padre abandona el Sistema.
    - El SO no permite a los hijos continuar. Terminación en cascada.
- Existe una llamada al sistema para abortar **otro proceso**. (**kill**)

Ejercicio 1 - FORK()

- Usamos un buble while(1) para ver el estado Run del proceso padre e hijo con igual código.

```
lidia@DESKTOP-HF986C4: ~/curso1
#include <unistd.h>
#include <stdio.h>

int main() {

int pid;
pid = fork(); //aqui recibimos lo que devuelve la función fork()

// ahora queremos saber si estamos en el proceso padre o el hijo

if ( pid > 0) {
            printf("Soy el proceso padre \n"); while(1);
          }
else {
       if (pid == 0) {
              printf("Soy el hijo\n"); while(1);
       }else {
              printf("Error en la función fork()\n");
              }
    }
}
lidia@DESKTOP-HF986C4:~/curso1$
```

```
lidia@DESKTOP-HF986C4: ~/curso1
lidia@DESKTOP-HF986C4:~/curso1$ ./programa1
Soy el proceso padre
Soy el hijo
```

```
lidia@DESKTOP-HF986C4: ~
lidia@DESKTOP-HF986C4:~$ ps -ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ssl    0:00 /init
   12 tty1     Ss     0:00 /init
   13 tty1     S      0:00 -bash
   38 tty1     R     19:47 ./programa1
   39 tty1     R     19:47 ./programa1
   40 tty2     Ss     0:00 /init
   41 tty2     S      0:00 -bash
   94 tty2     R      0:00 ps -ax
lidia@DESKTOP-HF986C4:~$
```

Compilar
cc programa1.c -o programa1

Ejecutar ./programa1

Ejercicio 2 –EXEC()

Fork de Procesos en "C"

```
int main()
{
pid_t  pid;
pid = fork();       /* bifurca un proceso hijo*/
     if (pid < 0) { /* ocurre un error*/
                    fprintf(stderr, "Fork Failed");
                    exit(-1);
                     }
     else if (pid == 0) { /* proceso hijo */
                          execlp("/bin/ls", "ls", NULL);
                              }
          else { /* proceso padre*/
                 wait (NULL);       /* padre espera a que se complete el hijo*/
                 printf ("Proceso padre   Child Complete. \n");
                exit(0);
     }
}
```

```
}lidia@DESKTOP-HF986C4:~/curso1$ ./programa3
a.out     programa.c  programa1.c  programa2.c  programa3.c
programa  programa1   programa2    programa3
Proceso padre  Child Complete
lidia@DESKTOP-HF986C4:~/curso1$
```


Ejercicio 3 - getpid(), getppid()

```
 8
 9  #include <sys/wait.h>
10  #include <unistd.h>
11  #include <stdio.h>
12  #include <stdlib.h>
13
14  int main () {
15  int num;
16  pid_t pid;  // define tipo pid
17
18
19  for (num= 0; num< 3; num++) {
20  pid= fork();       // crea un proceso
21
22  printf ("Soy el proceso de PID %d, mi padre tiene %d de PID.\n",
23  getpid(), getppid());
24  if (pid== 0)
25  break;
26  }
27  if (pid== 0)
28  sleep(5);
29  else
30  for (num= 0; num< 3; num++)
31  printf ("Fin del proceso de PID %d.\n", wait (NULL));
32  return 0;
33  }
```

```
Soy el proceso de PID 3055, mi padre tiene 3054 de PID.
Soy el proceso de PID 3055, mi padre tiene 3054 de PID.
Soy el proceso de PID 3060, mi padre tiene 3055 de PID.
Soy el proceso de PID 3055, mi padre tiene 3054 de PID.
Soy el proceso de PID 3061, mi padre tiene 3055 de PID.
Soy el proceso de PID 3059, mi padre tiene 3055 de PID.
Fin del proceso de PID 3060.
Fin del proceso de PID 3061.
Fin del proceso de PID 3059.


...Program finished with exit code 0
Press ENTER to exit console.
```

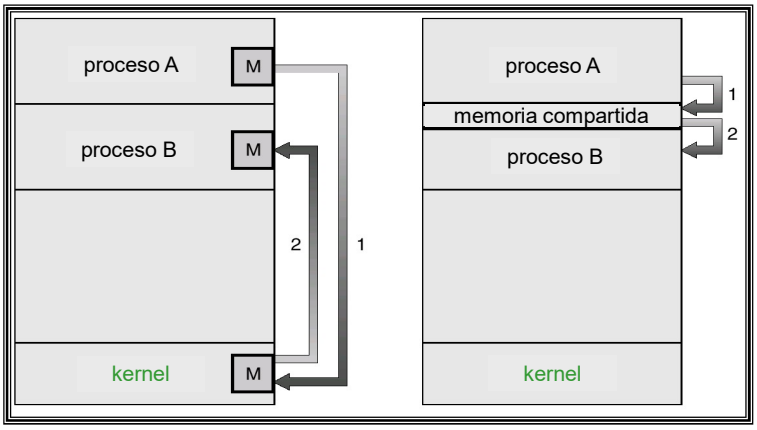
Comunicación Interprocesos (IPC)

- Los procesos en un sistema pueden ser **independientes** o **cooperativo**
- Procesos cooperativos: pueden afectar o ser afectados por otros procesos, cuando incluyen datos compartidos
- Razones para procesos cooperativos
  - Compartir Información
  - *Speedup* o acelerar los cálculos
  - Modularidad
  - Conveniencia
- Requieren mecanismos de *comunicación interprocesos (IPC)*
- Dos modelos de IPC
  - Memoria compartida
  - Pasaje de Mensajes

Modelos de Comunicación

Pasaje de mensajes

Memoria compartida



Problema del Productor-Consumidor

- Ejemplo del Paradigma procesos cooperativos: el proceso ***productor*** produce información que es consumida por un proceso ***consumidor*** .
  - *buffer ilimitado* - no tiene límites prácticos en el tamaño del buffer.
  - *buffer limitado* supone que hay un tamaño fijo de buffer.

Buffer limitado – Solución con memoria compartida

- Datos compartidos

```
constant n =10 ;  {tamaño del buffer}
type item = … ;
var buffer: array [0..n–1] of item;
    in, out: 0..n–1;
```

- Proceso Productor

```
repeat
    …
    produce un item en nextp
    …
    while in+1 mod n = out do no-op;
    buffer [in] :=nextp;
    in :=in+1 mod n;
until false;
```

Buffer limitado (cont.)

- Proceso Consumidor

```
repeat
    while in = out do no-op;
    nextc := buffer [out];
    out := out+1 mod n;
      …
    consume un item en nextc
      …
    until false;
```

- La solución es correcta, pero solo puede llenar el buffer hasta n–1.

Comunicación entre Procesos (IPC)

- Sistema paso de mensajes – los procesos se comunican uno con otro sin necesidad de compartir el mismo espacio de direcciones.
- La IPC provee dos operaciones:
  - **send**(*mensaje*) – mensaje de tamaño fijo o variable
  - **receive**(*mensaje*)
- Sea *P* y *Q*, dos procesos que desean comunicarse, necesitan:
  - Establecer un ***vínculo de comunicación*** entre ellos
  - Intercambiar mensajes via **send/receive**
- Implementación de un vínculo de comunicación
  - lógico (p.e., cx directa/indirecta, cx. síncrona/ asíncrona, almacenamiento en bufer explícito/ automático)
  - físico (p.e., memoria compartida, canal hardware o red)

Preguntas sobre la Implementación

- ¿Cómo se establecen los vínculos?
- ¿Puede un vínculo ser asociado con mas de dos procesos?
- ¿Cuántos vínculos puede haber entre cada par de procesos que se comunican?
- ¿Cuál es la capacidad de un vínculo?
- ¿El vinculo puede aceptar tamaño de mensajes fijo o variable?
- ¿El vínculo es unidireccional o bi-direccional?

Comunicación Directa

- Los procesos deben nombrar al otro explícitamente:
  - **send** (*P, mensaje*) – envía un mensaje al proceso P
  - **receive**(*Q, mensaje*) – recibe un mensaje del proceso Q
- Propiedades del *vínculo de comunicación*
  - Son establecidos automáticamente.
  - Un vínculo está asociado con exactamente un par de procesos que se comunican.
  - Entre cada par existe exactamente un vínculo.
  - Puede ser unidireccional, pero es usualmente bi-direccional.

Comunicación Indirecta

- Los mensajes son dirigidos y recibidos desde *mailboxes* (o *ports*).
  - Cada mailbox tiene una única identificación.
  - Los procesos pueden comunicarse solo si comparten un mailbox.
- Propiedades del *vínculo de comunicación*
  - Se establece solo si los procesos comparten un mailbox común.
  - Un vínculo puede ser asociado con muchos procesos.
  - Cada par de procesos puede compartir varios vínculos de comunicación.
  - Pueden ser unidireccionales o bi-direccionales.

Comunicación Indirecta (cont.)

- Operaciones
  - crear un nuevo *mailbox*
  - enviar y recibir mensajes por medio del *mailbox*
  - destruir un *mailbox*
- Las Primitivas están definidas como:

  send(*A, message*) – envía un mensaje al mailbox A

  receive(*A, message*) – recibe un mensaje del mailbox A

Comunicación Indirecta (cont.)

- Mailbox compartida
  - P_1, P_2, y P_3 comparten el mailbox A.
  - P_1, envía; P_2 y P_3 reciben.
  - ¿Quién toma el mensaje?
- Solución
  - Permitir que un vínculo está asociado a lo sumo con dos procesos.
  - Permitir que un solo proceso a la vez ejecute la operación *receive*.
  - Permitir que el sistema seleccione arbitrariamente el receptor. El enviador es notificado quien fue el receptor del mensaje.

Sincronización

- El pasaje de mensajes puede ser bloqueante o no bloqueante.
- **Bloqueante** es considerado **sincrónico**
  - ***Send bloqueante*** mantiene al enviador bloqueado hasta que el mensaje es recibido
  - ***Receive bloqueante*** mantiene al receptor bloqueado hasta que el mensaje esté disponible
- **No bloqueante** es considerado **asincrónico**
  - ***Send no bloqueante***: el emisor envía el mensaje y continúa
  - ***Receive no bloqueante***: el receptor extrae un mensaje válido o nulo

Buffering

- La cola de mensajes asociada al vínculo se puede implementar de tres maneras.
  1. Capacidad – 0 mensajes
     El enviador debe esperar por el receptor (*rendez-vous*). Es decir ambas operaciones se realizan con bloqueo.
  2. Capacidad limitada – longitud finita de *n* mensajes
     El enviador debe esperar si el vínculo está lleno.
  3. Capacidad ilimitada – longitud infinita
     El enviador nunca espera.

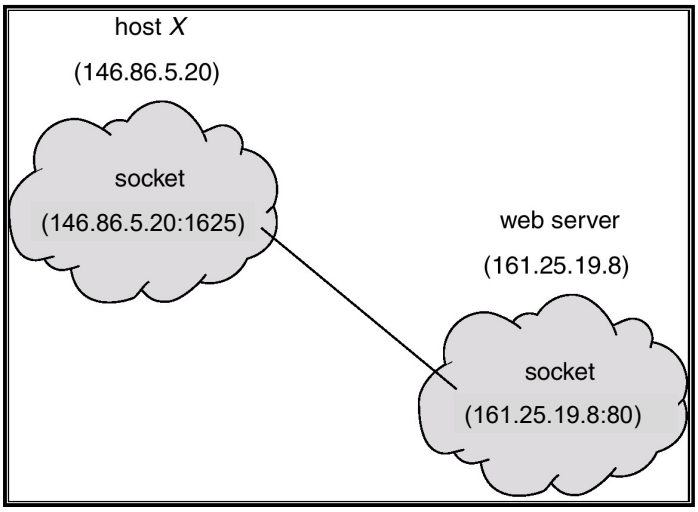
Comunicación Cliente-Servidor

- Sockets
- Llamadas a Procedimientos Remotors (**RPC**:Remote Procedure Calls)
- Invocación a Métodos Remotos (**RMI**:Remote Method Invocation (Java))

Sockets

- Un ***socket*** es definido como un *punto terminal de una comunicación*.
- Concatenación de dirección IP y pórtico
- El socket **161.25.19.8:1625** se refiere al pórtico **1625** en el host **161.25.19.8**
- La comunicación se lleva a cabo entre un par de sockets, uno para cada proceso.
- Puertos por debajo de 1024 son bien conocidos
  - 23 para telnet
  - 21 para ftp
  - 80 para servidor web o http

Comunicación por Sockets



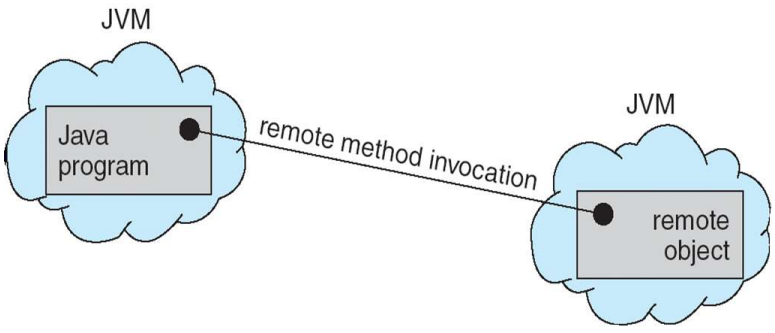
Llamada a Procedimiento Remoto (RPC)

- Las RPCs produce una llamada a procedimiento entre procesos sobre la red como si fuera local.
- El sistema RPC oculta detalles de la cx incorporando un *stub. (uno x cada RPC)*

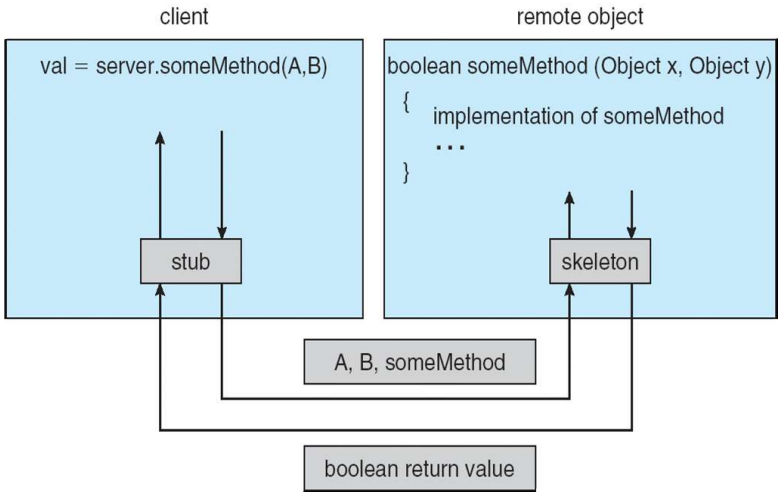
  Stubs – proxy del lado del cliente y del servidor.
- Cuando el cliente invoca un RPC, el sistema llama al stub apropiado junto a los parámetros.
- El stub del lado del cliente localiza el servidor y empaqueta (*marshall*) los parámetros.
- El stub del lado del servidor recibe este mensaje, desempaca los parámetros y ejecuta el procedimiento en el servidor.

Invocación de Métodos Remotos (RMI)

- RMI es un mecanismo de Java similar a los RPCs.
- RMI permite a un programa Java, sobre una máquina, invocar un método sobre un objeto remoto.



Marshalling de Parámetros



Fin

Módulo 3

Departamento de Informática
Facultad de Ingeniería
Universidad Nacional de la Patagonia “San Juan Bosco”