Procesos
3

Sistemas operativos y distribuidos

Gustavo Distel
gd@cs.uns.edu.ar

DCIC - UNS

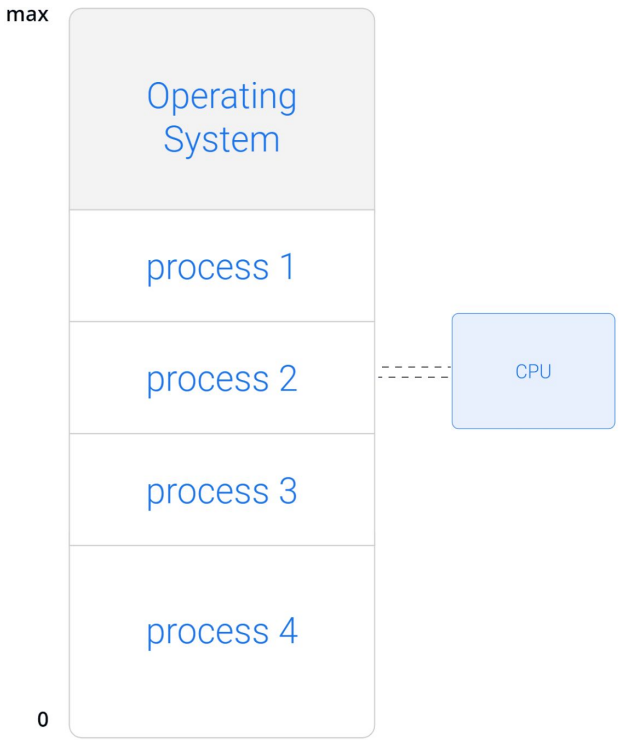
Procesos

Contenido

- Concepto de proceso
- Planificación de Procesos
- Operaciones en procesos
- Comunicación entre procesos
- *IPC* en sistemas de memoria compartida
- *IPC* en sistemas de paso de mensajes
- Ejemplos de sistemas *IPC*
- Comunicación en sistemas cliente-servidor

Procesos

- Informalmente, un proceso es un programa en ejecución.
- Unidad de trabajo en un sistema informático moderno.
- Un sistema consiste en una colección de procesos, algunos ejecutan código de usuario, otros ejecutan código de SO.
- Potencialmente, todos estos procesos pueden ejecutarse simultáneamente, con la *CPU* (o *CPUs*) multiplexando entre ellos.



Procesos

Almacenamiento temporal al invocar funciones (parámetros de funciones, direcciones de retorno y variables locales), tamaño variable en tiempo de ejecución.

Memoria asignada dinámicamente en tiempo de ejecución del programa, tamaño variable en tiempo de ejecución.

Variables globales, tamaño fijo, no cambia en tiempo de ejecución.

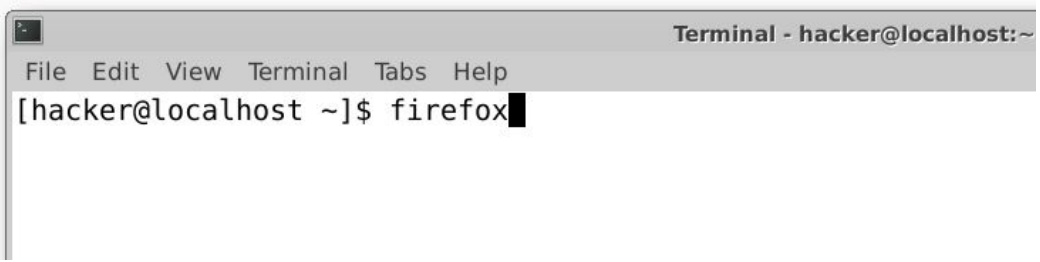
Código ejecutable, tamaño fijo, no cambia en tiempo de ejecución.

El estado actual de un proceso está representado por el valor del *program counter* y el contenido de los registros del procesador.

Un programa es una **entidad pasiva**, un archivo. En contraste, un proceso es una **entidad activa**, con un *program counter* que especifica la siguiente instrucción a ejecutar y un conjunto de recursos asociados.

Procesos

- Un programa se convierte en un proceso cuando se carga un archivo ejecutable en memoria.

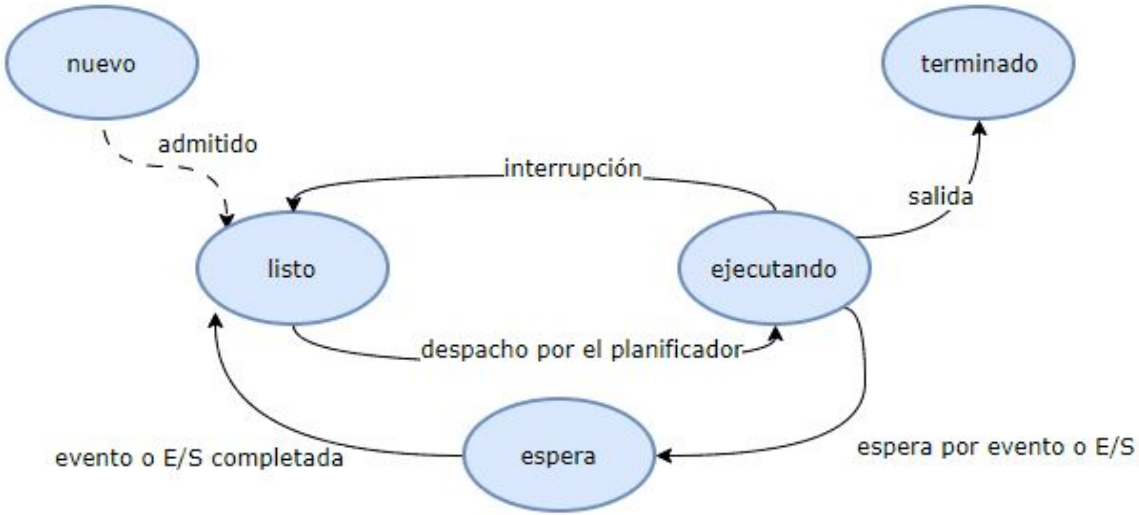


- Aunque dos procesos pueden estar asociados con el mismo programa, se consideran dos secuencias de ejecución separadas y, si bien las secciones de texto son equivalentes, las secciones de datos, *heap* y *stack* varían.
- Un proceso en sí mismo puede ser un entorno de ejecución para otro código → *Java*

Procesos

Estado

- A medida que se ejecuta un proceso, cambia de estado.
- El estado de un proceso se define en parte por la actividad actual de ese proceso.
- Un proceso puede estar en uno de los siguientes estados:
- **Nuevo**: proc. siendo creado.
- **Ejecutando**: proc. ejecutando instrucciones.
- **Espera**: proc. esperando que ocurra algún evento (como la finalización de E/S o la recepción de una señal).
- **Listo**: proc. esperando ser asignado a un procesador.
- **Terminado**: proc. finalizado.



Procesos

PCB

- Cada proceso está representado en el SO por un bloque de control de proceso (*process control block - PCB*). Datos necesarios para iniciar o reiniciar un proceso y datos de conteo.

process state
process number
program counter
registers
memory limits
list of open files
...

- **Estado del proceso:** nuevo, listo, en ejecución, en espera, detenido, etc.
- ***Program counter***: dirección de la siguiente instrucción que se ejecutará en este proc.
- **Registros de *CPU*:** los registros varían en número y tipo, dependiendo de la arquitectura de la computadora.
- **Información de planificación de la CPU:** prioridad de proc., punteros a las colas de planificación y cualquier otro parámetro de planificación.
- **Información de gestión de memoria:** esta información puede incluir elementos tales como, el valor de los registros base y de límite y las tablas de páginas, o las tablas de segmentos, dependiendo del sistema de memoria utilizado por el SO.
- **Información de conteo:** esta información incluye la cantidad de *CPU* y tiempo real utilizado, límites de tiempo, números de cuenta, números de trabajo o proc., etc.
- **Información de estado de E/S:** esta información incluye la lista de dispositivos de E/S asignados al proc., lista de archivos abiertos, etc.

Procesos

Hilos (*Threads*)

- La mayoría de los SO modernos han ampliado el concepto de proceso para permitir que un proceso tenga múltiples hilos de ejecución y, por lo tanto, realizar más de una tarea a la vez.
- Esta característica es beneficiosa en sistemas *multicore*, donde varios hilos pueden ejecutarse en paralelo.
- Por ejemplo, en un navegador *web* implementado con hilos, se podría crear un hilo para cada uno de lo tabs abiertos que a su vez contendrá una página *web* en particular.
- En los sistemas que soportan hilos, el *PCB* se expande para incluir información de cada hilo.

```
File  Edit  View  Terminal  Tabs  Help
top - 00:36:55 up 29 min,  1 user,  load average: 0.65, 1.30, 0.72
Threads: 570 total,   1 running, 569 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.4 us,  0.7 sy,  0.0 ni, 97.7 id,  0.0 wa,  0.1 hi,  0.1 si,  0.0 s
MiB Mem :   3914.8 total,   1168.6 free,   1330.3 used,   1415.9 buff/cache
MiB Swap:   4064.0 total,   4064.0 free,      0.0 used.   2216.5 avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  3732 hacker    20   0 2719376 224000 133644 S   6.6   5.6   0:18.43 Web Cont
  3858 hacker    20   0  219380   4392   3308 R   1.3   0.1   0:00.37 top
  1075 root      20   0  689196 152468  80092 S   0.7   3.8   0:57.25 Xorg
  1580 hacker    20   0  519924  38332  31156 S   0.3   1.0   0:03.26 vmtoolsd
  2557 hacker    20   0  667592  55156  37632 S   0.3   1.4   0:05.12 xfce4-te
  3590 root      20   0       0      0      0 I   0.3   0.0   0:00.13 kworker/
  3862 hacker    20   0  647304  49916  34432 S   0.3   1.2   0:00.56 gedit
```

Ejemplo Linux

```
[gdistel@localhost ~]$ top -H
```

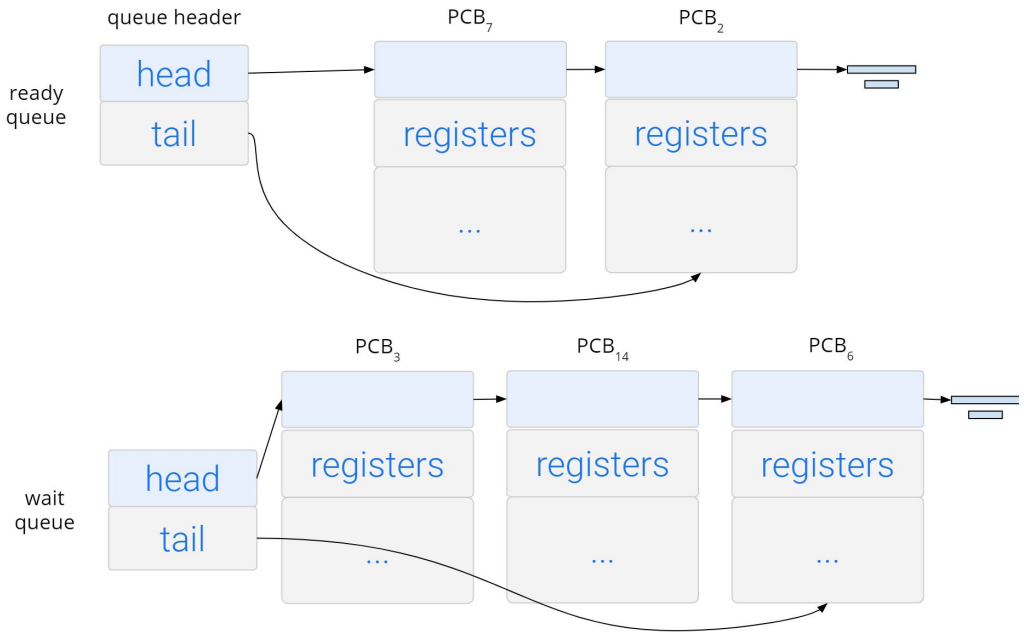
Planificación de procesos

- El objetivo de la multiprogramación (*multiprogramming*) es tener algún proceso en ejecución en todo momento para maximizar la utilización de la *CPU*.
- El objetivo del tiempo compartido (*time sharing*) es intercambiar muchos procesos en un *core* de la *CPU* con tanta frecuencia, que los usuarios puedan interactuar con cada programa mientras se ejecutan.
- Para esto, el planificador de procesos (*process scheduler*) selecciona un proceso para ejecutar en un *core* de la *CPU*.
- Si hay más procesos que *cores*, el exceso de procesos tendrá que esperar hasta que un *core* esté libre y puedan replanificarse.
- La cantidad de procesos en memoria se conoce como el grado de multiprogramación (*degree of multiprogramming*).
- Equilibrar los objetivos mencionados requiere tener en cuenta el comportamiento de un proceso.
    - Proceso limitado por E/S: pasa más tiempo haciendo E/S que haciendo cálculos.
    - Proceso limitado por *CPU*: pasa más tiempo haciendo cálculos que haciendo E/S.

Planificación de procesos

Colas de planificación

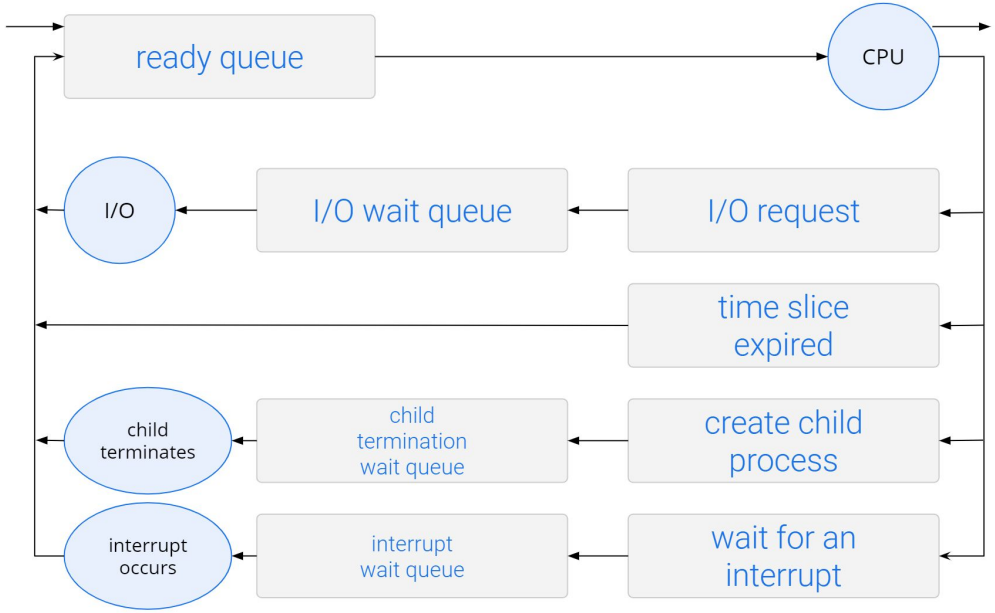
- A medida que los procesos ingresan al sistema, se colocan en la cola de listos (*ready queue*), dónde están listos y esperando para ejecutarse en un *core* de la *CPU*.
- El sistema también incluye otras colas; los procesos que esperan que ocurra un determinado evento, como la finalización de E/S, se colocan en la cola de espera (*wait queue*).



Planificación de procesos

Colas de planificación

- Una representación común de la planificación de procesos es un diagrama de colas, con dos tipos de colas: la cola de listos y un conjunto de colas de espera.
- Los círculos representan los recursos que abastecen a las colas, y las flechas indican el flujo de procesos en el sistema.



Planificación de procesos

Colas de planificación

- Un nuevo proceso se coloca inicialmente en la cola listos. Espera allí hasta que se seleccione para su ejecución o sea despachado (***dispatched***). Una vez que el proceso tiene asignado un *core* y se está ejecutando, podría ocurrir uno de varios eventos:
    - El proceso podría solicitar E/S y luego colocarse en la cola de espera de E/S.
    - El proceso podría crear un nuevo proceso hijo y luego colocarse en una cola de espera mientras espera la terminación del hijo.
    - El proceso podría eliminarse a la fuerza del *core*, como resultado de una interrupción o al expirar su intervalo de tiempo, y volver a colocarse en la cola de listos.
- En los primeros dos casos, el proceso eventualmente cambia del estado de espera al estado listo y luego se vuelve a colocar en la cola de listos.
- Un proceso continúa este ciclo hasta que finaliza, en ese momento se elimina de todas las colas y su *PCB* y recursos son desasignados.

Planificación de procesos

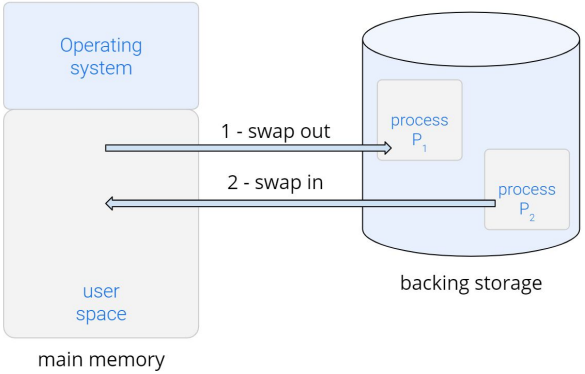
Planificación de *CPU*

- La función del planificador de la *CPU* (*CPU scheduler*) es seleccionar entre los procesos que están en la cola de listos y asignar un *core* de la *CPU* a uno de ellos.
- El planificador de la *CPU* debe seleccionar con frecuencia un nuevo proceso para la *CPU*.
- Un proceso limitado por E/S puede ejecutarse solo unos pocos milisegundos antes de esperar una solicitud de E/S.
- Un proceso limitado por *CPU* requerirá un *core* de *CPU* por mucho tiempo, aunque es poco probable que el planificador otorgue el *core* a un proceso durante un período prolongado.
- Es probable, en cambio, que esté diseñado para eliminar por la fuerza la *CPU* de un proceso y planificar otro proceso para que se ejecute.
- Por lo tanto, el planificador de la *CPU* se ejecuta al menos una vez cada 100 milisegundos, aunque generalmente es mucho más frecuente.

Planificación de procesos

Planificación de *CPU*

- Algunos SO tienen una forma intermedia de planificación, conocida como *swapping*, cuya idea es que en determinadas circunstancias puede ser ventajoso eliminar un proceso de la memoria (y de la disputa activa por la *CPU*) y así reducir el grado de multiprogramación.
- Más tarde, el proceso puede re-introducirse en la memoria y su ejecución puede continuar.
- Este esquema se conoce como *swapping* porque un proceso puede "*swapped out*" de la memoria al disco, donde se guarda su estado actual, y luego "*swapped in*" del disco a la memoria, donde se restaura su estado.
- Normalmente, el *swapping* solo es necesario cuando la memoria se ha comprometido en exceso y debe liberarse.



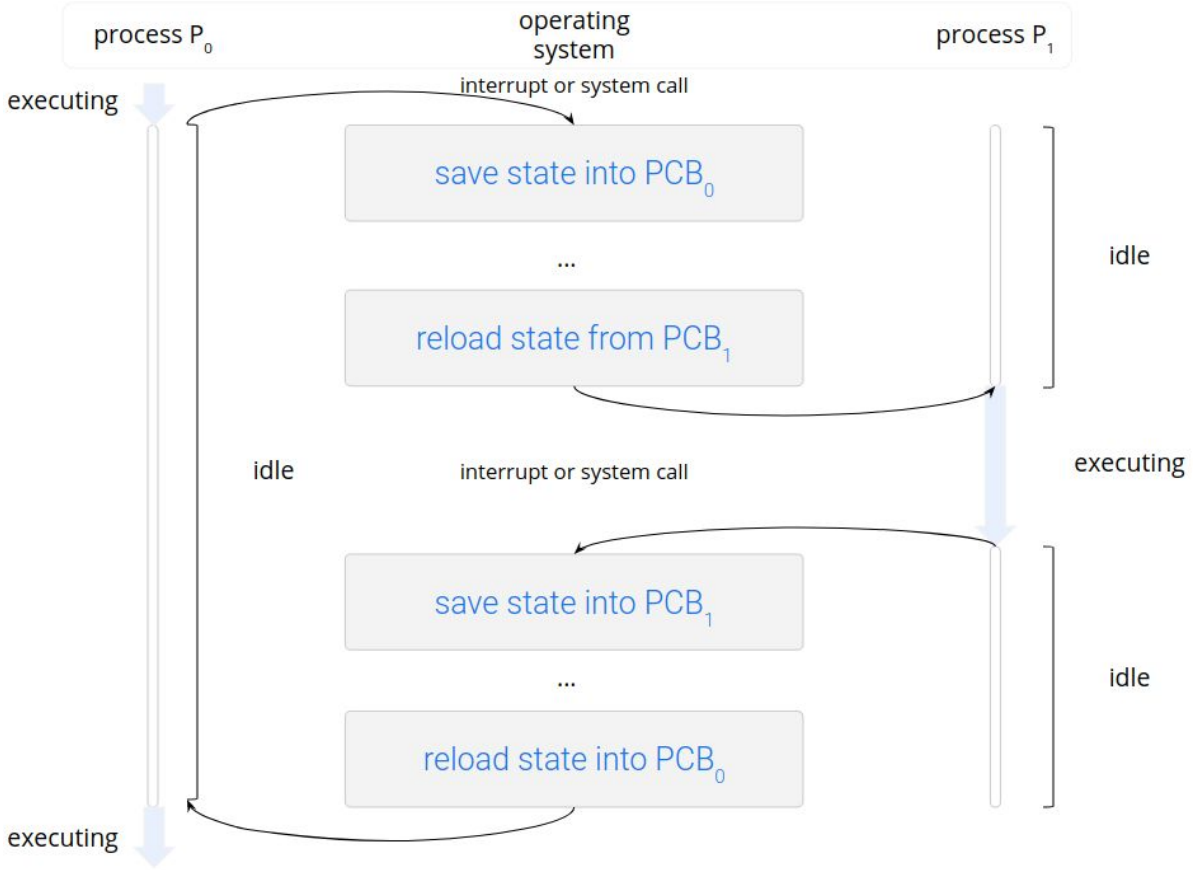
Planificación de procesos

Cambio de contexto

- Cuando se produce una interrupción, el sistema necesita guardar el contexto actual del proceso que se está ejecutando en el core de la *CPU* para que pueda ser restaurado más tarde.
- El contexto está representado en el *PCB* del proceso.
- Cambiar el *core* de la *CPU* a otro proceso requiere guardar el estado del proceso actual y restaurar el estado de un proceso diferente → cambio de contexto (*context switch*).
- Cuando ocurre un cambio de contexto, el *kernel* guarda el contexto del proceso anterior en su *PCB* y carga el contexto guardado del nuevo proceso que ha sido planificado para ejecutarse.
- El tiempo de cambio de contexto es pura sobrecarga (*overhead*), porque el sistema no realiza un trabajo útil durante este momento.
- La velocidad de cambio de contexto varía de una máquina a otra, dependiendo de la velocidad de la memoria, el número de registros que se deben copiar y la existencia de instrucciones especiales (como una sola instrucción para cargar o almacenar todos los registros). Una velocidad típica es de varios microsegundos.

Planificación de procesos

Cambio de contexto



Operaciones en procesos

Creación

- El proceso padre (*parent*) crea procesos hijos (*children*), que a su vez crean otros procesos, formando un árbol de procesos.
- La mayoría de los sistemas operativos (incluidos UNIX, Linux y Windows) identifican procesos de acuerdo con un identificador de proceso único (*PID: process identifier*), que generalmente es un número entero.
- El *PID* proporciona un valor único para cada proceso en el sistema y puede usarse como índice para acceder a varios atributos de un proceso dentro del *kernel*.

Operaciones en procesos

Creación

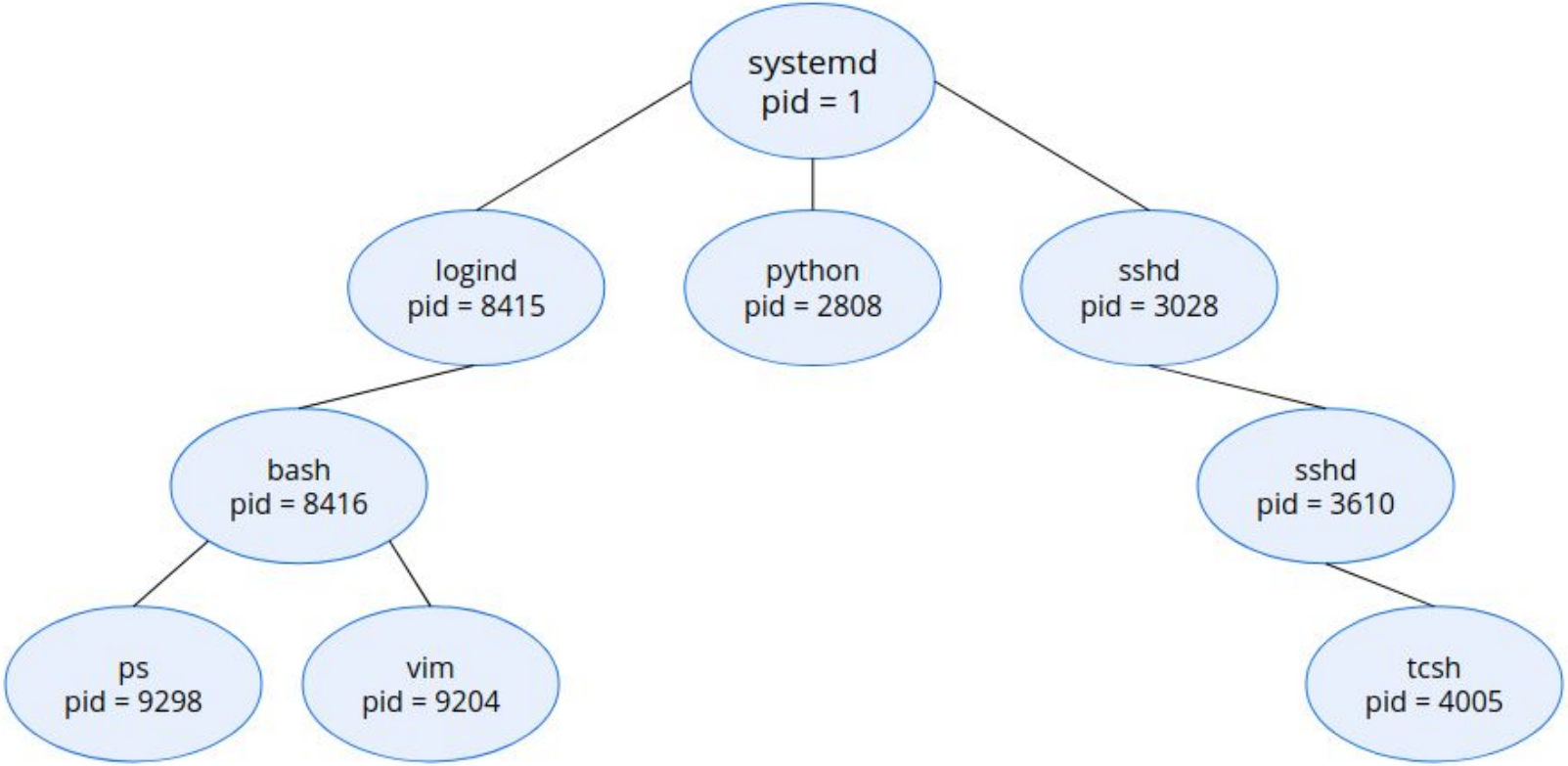
- `pstree`: muestra los procesos en ejecución como un árbol.

```
[gdistel@battlecruiser ~]$ pstree -p
systemd(1)─┬─ ...
           ...
           ├─login(7876)───bash(7884)───htop(7916)
           ...
           ├─firewalld(866)───{firewalld}(1078)
           ...
           ├─sshd(982)───sshd(8504)───sshd(8512)───bash(8513)───top(8550)

           ├─gdm(989)─┬─gdm-session-wor(1462)─┬─gdm-x-session(1501)─┬─Xorg(1503)─┬─{Xorg}(1507)
                      │                       │                     │            ├─{Xorg}(1508)
                      │                       │                     │            ├─{Xorg}(1509)
                     ...                     ...                   ...          ...
                      │                       │                     │            ├─startdde(1577)─┬─abrt-applet(1983)─┬
                      │                       │                     │            │                │
                      │                       │                     │            │                ├─chrome(2200)─
                      │                       │                     │            │                ├─doublecmd(6955)
                      │                       │                     │            │                ├─firefox(3851)─
                      │                       │                     │            │                ├─firefox(3851)─
```

Operaciones en procesos

Creación



Operaciones en procesos

Creación

- Cuando un proceso crea un proceso hijo, ese proceso hijo necesitará ciertos recursos (tiempo de *CPU*, memoria, archivos, dispositivos de E/S) para realizar su tarea.
  - Puede obtener los recursos directamente desde el SO o,
  - Puede estar limitado a un subconjunto de los recursos del proceso padre.
    - El padre puede tener que dividir sus recursos entre sus hijos, o puede compartir algunos recursos (como memoria o archivos) entre varios de sus hijos.
- Opciones de ejecución:
  - El padre continúa ejecutándose concurrentemente con sus hijos.
  - El padre espera hasta que algunos o todos sus hijos hayan terminado.
- Espacio de direcciones para el nuevo proceso:
  - El proceso hijo es un duplicado del proceso padre (tiene el mismo programa y datos que el padre).
  - El proceso hijo tiene un nuevo programa cargado.

Operaciones en procesos

Creación

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>

int main(void){
    pid_t child_pid;

    child_pid = fork();

    if(child_pid == -1){
        printf("Error al crear proceso, valor de errno: %d\n ", errno);
        printf("Mensaje de error : %s\n", strerror(errno));
        exit(EXIT_FAILURE);

    }else if(child_pid > 0){
        printf("Proceso padre con PID [%i]\n", getpid());

    }else{
        printf("Proceso hijo con PID [%i]\n", getpid());
    }
    exit(EXIT_SUCCESS);
}
```

Ejemplo Linux

```
[gdistel@localhost ~]$ man 2 fork
DESCRIPTION: creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.
```

Operaciones en procesos

Creación

Ejemplo Linux

```
[gdistel@localhost ~]$ man execl
DESCRIPTION: The exec() family of functions replaces the current process image with a new process image.
```

```c
#include <unistd.h>
int main(void){
    // Funciones: execl(...) - execlp(...) - execle(...)
    // La "l" es de list, lista de argumentos.
    // La "p" es de PATH, se puede usar esta variable de entorno para determinar el proceso ejecutable,
    // es decir, no es necesario especificar el path completo.
    // La "e" es para especificar el entorno.

    execl("/bin/ls", "/bin/ls", "-l", "-a",(char *)NULL);
    /*
    execlp("ls", "ls", "-l", "-a",(char *)NULL);
    */

    /*
    //para la sig declarar:
    char *envp[]={"PATH=/tmp","USER=games",NULL};
    execle("/bin/env","env", NULL, envp);
    */
}
```

Operaciones en procesos

Creación

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

void _tmain( int argc, TCHAR *argv[] )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( argc != 2 )
    {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
        argv[1],        // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

Ejemplo Windows

https://docs.microsoft.com/es-es/windows/win32/procthread/creating-processes

Compilado y ejecutado con *visual studio community 2019*

Operaciones en procesos

Terminación

- Un proceso finaliza cuando ejecuta su última sentencia y le pide al SO que lo elimine mediante la llamada al sistema *exit()*.
  - Devuelve un valor de estado (generalmente un entero) a su proceso padre.
  - El SO desasigna y reclama todos los recursos del proceso finalizado.
- Un proceso padre puede terminar la ejecución de su hijo a través de una *system call*. Debe conocer el *PID* de su hijo, que generalmente se transmite cuando se crea. Las posibles razones son:
  - El hijo ha excedido el uso de algunos de los recursos asignados.
  - La tarea asignada al hijo ya no es necesaria.
  - El padre terminó y el SO no permite que un hijo continúe si su padre ha terminado.

Operaciones en procesos

Terminación

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <string.h>

int main(void){
    pid_t child_pid, child_pid2;

    child_pid = fork();

    if(child_pid == -1){
        printf("Error a crear el proceso, valor de errno: %d\n", errno);
        printf("Mensaje de error: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }else if(child_pid > 0){ // o != 0
        printf("Proceso padre con PID [%i], espero a mi hijo\n", getpid());
        child_pid2 = wait(NULL);
        printf("Proceso padre con PID [%i], mi hijo termino PID[%i]\n", getpid(), child_pid2);
    }else{
        printf("Proceso hijo con PID [%i], lo hago esperar a mi padre 10 segundos\n", getpid());
        sleep(10);
    }
    exit(EXIT_SUCCESS);
}
```

Ejemplo Linux

```
[gdistel@localhost ~]$ man 3 wait
DESCRIPTION: All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.
```

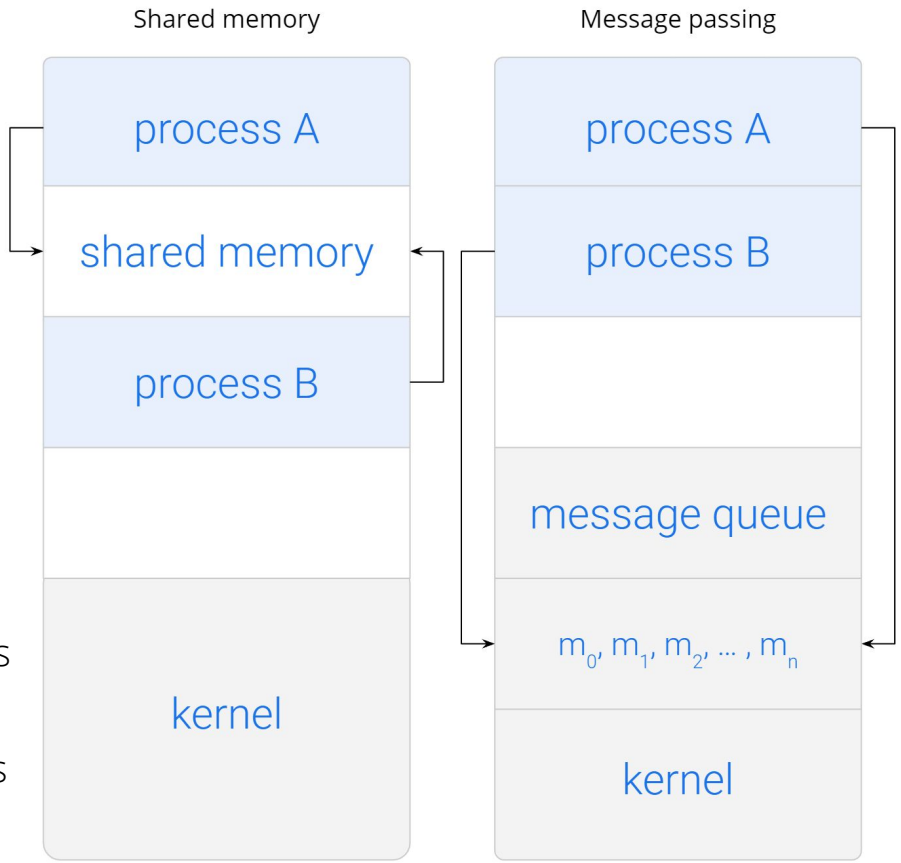
- Un proceso que ha finalizado, pero cuyo padre aún no ha llamado a `wait()`, se conoce como proceso zombie.
- Si un padre no invoca a `wait()` y termina, deja a sus procesos hijos como huérfanos.

Comunicación entre procesos

- Los procesos que se ejecutan concurrentemente en un SO pueden ser:
  - **Independientes:** un proceso es independiente si no comparte datos con ningún otro proceso que se ejecute en el sistema.
  - **Cooperativos:** un proceso es cooperativo si puede afectar o verse afectado por otros procesos que se ejecutan en el sistema.
- Razones para la existencia de procesos cooperativos:
  - Compartir información.
  - Acelerar los cálculos (*Computation speedup*).
  - Modularidad.
- Los procesos cooperativos requieren un mecanismo de comunicación entre procesos (*interprocess communication (IPC)*) que les permita intercambiar datos.

Comunicación entre procesos

- Hay dos modelos fundamentales de comunicación entre procesos:
  - **memoria compartida** y
  - **paso de mensajes**.
- Memoria compartida:
  - Más rápida que paso de mensajes.
  - *System calls* sólo para establecerla.
- Paso de mensajes:
  - Útil para intercambiar datos pequeños.
  - Más fácil de implementar en sistemas distribuidos.
  - Implementada con *system calls* → más intervención del *kernel*.



IPC en sistemas de memoria compartida

- Requiere que los procesos establezcan una región de memoria compartida.
- Generalmente, dicha región reside en el espacio de direcciones del proceso que crea el segmento de memoria compartida. Otros procesos que deseen utilizarla deben vincularla a su espacio de direcciones.
- La forma de los datos y su ubicación están determinados por estos procesos y **no están bajo el control del SO**.
- Los procesos también son responsables de garantizar que no se escriba en la **misma ubicación simultáneamente**.

IPC en sistemas de memoria compartida

El problema productor-consumidor

- Un proceso productor produce información que es consumida por un proceso consumidor.
- Ej.: servidor *web,* servidor de impresión.
- Una solución al problema utiliza memoria compartida, en donde hay un búfer utilizado concurrentemente por el productor y el consumidor.
- El productor y el consumidor deben estar **sincronizados**, de modo que el consumidor no intente consumir un artículo que aún no se ha producido.
- Existen dos tipos de búfer:
  - **Búfer no limitado (*unbounded buffer*)**: no establece un límite práctico en el tamaño del búfer. El consumidor podría tener que esperar a que se produzcan nuevos artículos, pero el productor siempre puede producir.
  - **Búfer limitado (*bounded buffer*)**: asume un tamaño de búfer fijo. En este caso, el consumidor debe esperar si el búfer está vacío, y el productor debe esperar si el búfer está lleno.

IPC en sistemas de memoria compartida

- Variables en memoria compartida de productor y consumidor:
- El búfer compartido se implementa como una matriz circular con dos punteros: `in` y `out`.
  - `in` apunta a la siguiente posición libre en el búfer.
  - `out` apunta a la primera posición llena en el búfer.
- buffer vacío cuando `in == out`.
- buffer lleno cuando `((in + 1) % BUFFER SIZE) == out`.

```
#define BUFFER SIZE 10
typedef struct {
...
} item;
item buffer[BUFFER SIZE];
int in = 0;
int out = 0;
```

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;
}
```

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 en sistemas de paso de mensajes

- Otra forma por la cual los procesos cooperativos pueden comunicarse entre si, es que el SO, proporcione alguna facilidad de paso de mensajes.
- El paso de mensajes permite que los procesos se comuniquen y sincronicen sus acciones sin compartir el mismo espacio de direcciones.
- Es útil en un entorno distribuido, donde los procesos que se comunican pueden residir en diferentes computadoras conectadas en red.
- *IPC* proporciona dos operaciones:
  - `send(message)`
  - `receive(message)`
- El tamaño del mensaje puede ser fijo o variable.
  - Fijo: implementación de SO sencilla, programación compleja.
  - Variable: implementación de SO compleja, programación sencilla.

IPC en sistemas de paso de mensajes

- Si los procesos P y Q desean comunicarse utilizando paso de mensajes, debe existir un *link* de comunicación entre ellos.
- Este *link* se puede implementar de varias maneras.
  - No se verá la implementación física (como memoria compartida, *bus* de *hardware* o la red), sino su implementación lógica.
- Hay varios métodos para implementar lógicamente un *link* y las operaciones `send()/receive()`:
  - Comunicación directa o indirecta.
  - Comunicación sincrónica o asincrónica.
  - Almacenamiento en búfer automático o explícito.

IPC en sistemas de paso de mensajes

Nombrado (*Naming*)

- Los procesos que se comunican deben tener una manera de referirse entre sí. Pueden usar comunicación directa o indirecta.
- **Comunicación directa:**
  - Los procesos se nombran entre sí explícitamente:
    - `send(P, message)` : envía un mensaje al proceso P.
    - `receive(Q, message)` : recibe un mensaje del proceso Q.
  - Propiedades del *link* de comunicación:
    - Los *links* se establecen automáticamente. Los proc. sólo conocen la identidad del otro.
    - Un *link* está asociado exactamente con dos procesos.
    - Entre cada par de procesos existe exactamente un *link*.
  - El esquema anterior es simétrico: hay que nombrar el proceso en la comunicación.
  - Esquema asimétrico: Solo el que envía nombra al que recibe.
    - `send(P, message), receive(id, message).` id se instancia con el proc.
  - Desventaja: *hard-coding*.

IPC en sistemas de paso de mensajes

Nombrado (*Naming*)

- Comunicación indirecta:
  - Los mensajes se envían y se reciben de *mailboxes* (buzones), también conocidos como puertos (*ports*).
    - Cada *mailbox* tiene una identificación única.
      - *POSIX message queues* → valor entero.
    - Los procesos solo pueden comunicarse si comparten un *mailbox*.
  - Primitivas
    - `send(A, message)` : envía un mensaje al *mailbox* A.
    - `receive(A, message)` : recibe un mensaje del *mailbox* A.
  - Propiedades del link de comunicación.
    - Se establece un *link* sólo si los procesos comparten un buzón común.
    - Un *link* puede estar asociado con más de dos procesos.
    - Cada par de procesos puede compartir varios *links* de comunicación, uno por cada buzón.

IPC en sistemas de paso de mensajes

Nombrado (*Naming*)

- Supongamos que los procesos P1, P2 y P3 comparten el *mailbox* A. Luego, P1 envía mientras que P2 y P3 reciben. ¿Quién recibe el mensaje?.
- Soluciones:
  - Permitir que un *link* se asocie con, a lo sumo, dos procesos.
  - Permitir que solo un proceso a la vez ejecute una operación de recepción.
  - Permitir que el sistema seleccione arbitrariamente el receptor, el sistema puede identificar al receptor ante el transmisor.
- El *mailbox* puede pertenecer al proceso o al SO.
  - *mailbox* en proceso:
    - Propietario y usuarios.
  - *mailbox* en SO, mecanismos para:
    - Crear un nuevo *mailbox*.
    - Operaciones para enviar y recibir.
    - Borrar un *mailbox*.

IPC en sistemas de pasaje de mensajes

Sincronización

- Hay diferentes opciones para la implementación de las primitivas `send()` y `receive()`.
- El paso de mensajes puede ser bloqueante o no bloqueante (sincrónica o asincrónica):
  - **Bloqueo de envío:** el proc. que envía se bloquea hasta que se recibe el mensaje.
  - **Envío sin bloqueo:** el proc. envía el mensaje y continúa.
  - **Bloqueo de recepción:** el receptor está bloqueado hasta que haya un mensaje disponible.
  - **Recepción sin bloqueo:** el receptor recibe:
    - Un mensaje válido, o
    - Un mensaje nulo.
- Diferentes combinaciones de `send()` y `receive()` son posibles:
  - Si tanto el envío como la recepción están bloqueados, tenemos un *rendezvous*.
    - Solución trivial al problema de productor consumidor.

IPC en sistemas de pasaje de mensajes

Buffering

- Los mensajes intercambiados por los procesos residen en una cola temporal, implementada en una de las siguientes tres formas:
  - **Capacidad cero:** la cola tiene una longitud máxima de cero.
    - No puede haber ningún mensaje esperando.
    - El remitente debe bloquearse hasta que el destinatario reciba el mensaje.
  - **Capacidad limitada:** la cola tiene una longitud finita n; por lo tanto puede haber al menos n mensajes.
    - Si la cola no está llena cuando se envía un mensaje nuevo, el mensaje se coloca en la cola (se copia el mensaje o se mantiene un puntero al mensaje), y el remitente puede continuar la ejecución sin esperar.
    - Si está lleno, el remitente debe bloquearse hasta que haya espacio disponible.
  - **Capacidad ilimitada:** la longitud de la cola es potencialmente infinita; por lo tanto, podría existir cualquier número de mensajes. El remitente nunca se bloquea.

Ejemplos de sistemas *IPC*

- *POSIX Shared Memory* (se ve en la práctica)
- *Mach Message Passing*
- *Windows*
- *Pipes*

Comunicación en sistemas cliente-servidor

Sockets

- Un *socket* se define como un punto final para la comunicación.
- Un par de procesos que se comunican a través de una red emplea un par de *sockets*, uno para cada proceso.
- Un *socket* se identifica mediante una dirección *IP (Internet Protocol address - IP address)* concatenada con un número de puerto (*port*).
- En general, los *sockets* usan una arquitectura cliente-servidor.
- El servidor espera las solicitudes de los clientes escuchando en un puerto específico. Una vez que se recibe una solicitud, el servidor acepta una conexión del *socket* del cliente para completar la conexión.
- Los servidores que implementan servicios específicos (como *SSH*, *FTP* y *HTTP*) escuchan en puertos conocidos (*well-known*), un servidor *SSH* escucha en el puerto 22; un servidor *FTP* escucha en el puerto 21; y un servidor web o *HTTP* escucha en el puerto 80.
- Todos los puertos por debajo de 1024 se consideran *well-known* y se utilizan para implementar servicios estándar.

Comunicación en sistemas cliente-servidor

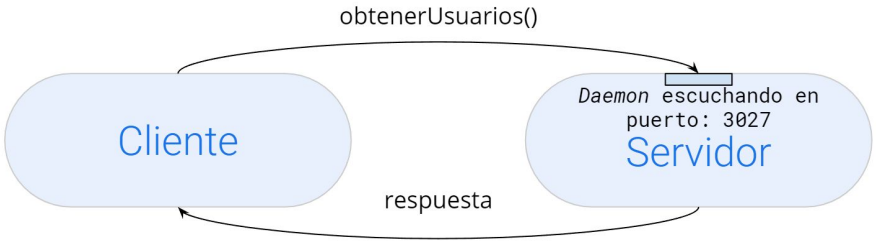
Sockets

- Cuando un proceso cliente inicia una solicitud de conexión, su computadora *host* le asigna un puerto. Este puerto tiene un número arbitrario mayor que 1024.
- La conexión consistirá en un par de *sockets*, ejemplo:
  - 146.86.5.20:1625 en el host X y
  - 161.25.19.8:80 en el servidor *web*.
- Los paquetes que viajan entre los *hosts* se entregan al proceso apropiado en función del número de puerto de destino.
- Todas las conexiones deben ser únicas.
  - Si otro proceso también en el host X desea establecer otra conexión con el mismo servidor *web*, se le asignará un número de puerto mayor que 1024 y no igual a 1625. Esto asegura que todas las conexiones consistan en un par único de *sockets*.

Comunicación en sistemas cliente-servidor

Remote Procedure Calls (RPC)

- Abstracción del mecanismo de llamada a procedimiento para ser utilizado entre sistemas conectados en red.
- A diferencia de los mensajes *IPC,* los mensajes en la comunicación *RPC* están bien estructurados y, por lo tanto, no son solo paquetes de datos.
- Cada mensaje se dirige a un demonio *RPC* que escucha en un puerto del sistema remoto, y cada uno contiene un identificador que especifica la función a ejecutar y los parámetros a pasar a esa función.
- Un puerto es un número incluido al comienzo de un paquete de mensajes. Mientras que un sistema normalmente tiene una dirección de red, puede tener muchos puertos en esa dirección para diferenciar los diversos servicios de red que soporta.
- Si un proceso remoto necesita un servicio, dirige un mensaje al puerto adecuado.



Comunicación en sistemas cliente-servidor

Remote Procedure Calls

- La semántica de *RPCs* permite que un cliente invoque un procedimiento en un *host* remoto como si invocara un procedimiento local.
- El sistema *RPC* oculta los detalles que permiten la comunicación al proporcionar un *stub* en el lado del cliente. Por lo general, existe un *stub* diferente para cada procedimiento remoto.
- Cuando el cliente invoca un procedimiento remoto, el sistema *RPC* llama al *stub* apropiado y le pasa los parámetros proporcionados al procedimiento remoto.
- Este *stub* (con el SO de intermediario) ubica el puerto en el servidor y *marshals* (envuelve/empaqueta) los parámetros para transmitirlos en red.
- Luego, el *stub* transmite un mensaje al servidor mediante el paso de mensajes.
- Un *stub* similar en el lado del servidor recibe este mensaje e invoca el procedimiento en el servidor. Si es necesario, los valores de retorno se devuelven al cliente utilizando la misma técnica.

Comunicación en sistemas cliente-servidor

Remote Procedure Calls

- **Semántica de llamada:** las llamadas a procedimientos locales fallan solo bajo circunstancias extremas. Las RPCs pueden fallar, ejecutarse más de una vez, como resultado de errores comunes de la red.
- Una forma de abordar este problema es que el SO garantice que se actúe en respuesta a los mensajes exactamente una vez (*exactly once*), en lugar de como máximo una vez (*at most once*).
- Como máximo una vez:
  - Esta semántica puede implementarse adjuntando una marca de tiempo a cada mensaje.
  - El servidor debe mantener un historial de todas las marcas de tiempo de los mensajes que ya ha procesado o un historial lo suficientemente grande como para garantizar que se detecten los mensajes repetidos.
  - Los mensajes entrantes que ya tienen una marca de tiempo en el historial se ignoran. El cliente puede enviar un mensaje una o más veces y estar seguro de que solo se ejecuta una vez.

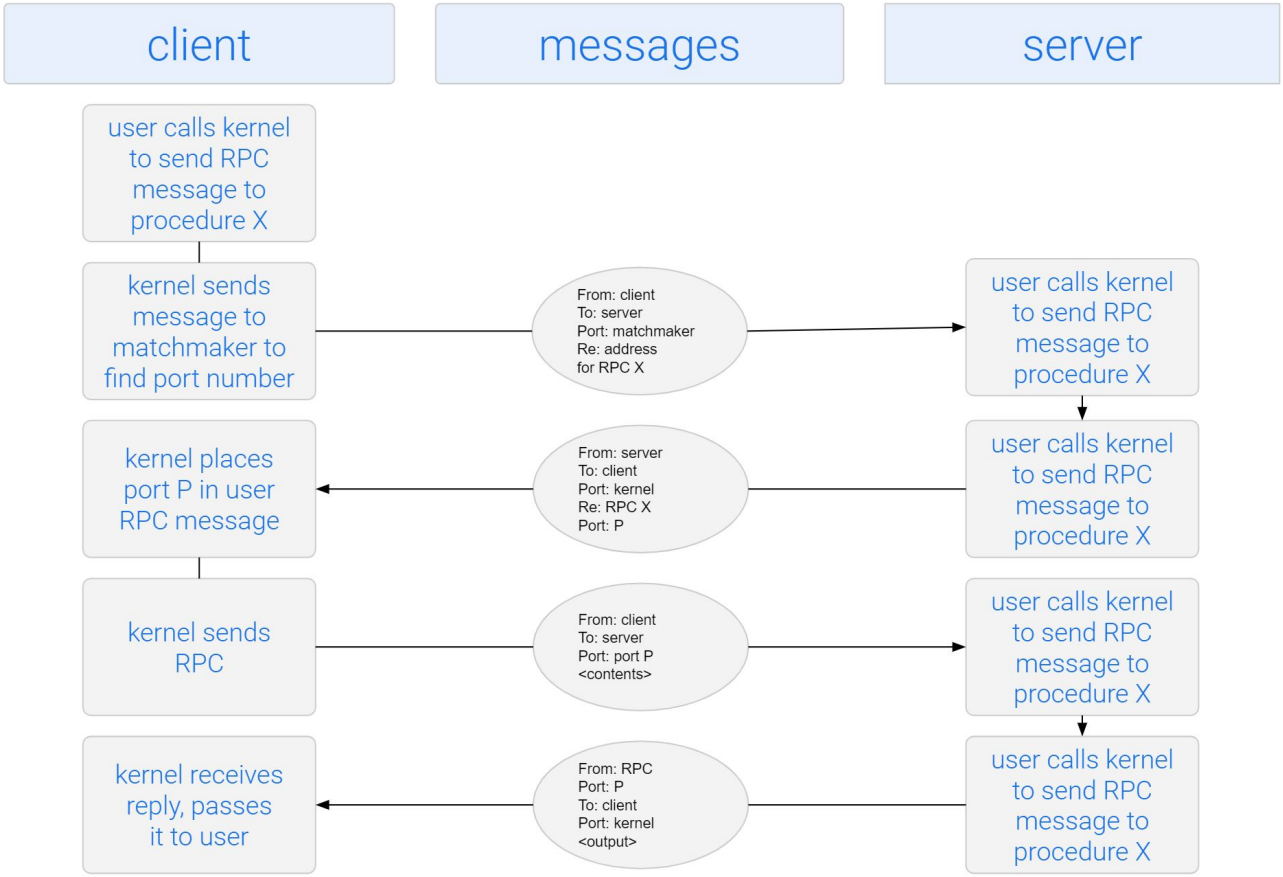
Comunicación en sistemas cliente-servidor

Remote Procedure Calls

- Exactamente una vez:
  - Se debe eliminar el riesgo de que el servidor nunca reciba la solicitud.
  - Para esto, el servidor debe implementar el protocolo "como máximo una vez" descrito y también tiene que confirmar al cliente que la llamada *RPC* fue recibida y ejecutada.
  - Los mensajes de confirmación (*ACK*, acknowledge) son comunes en todas las redes.
  - El cliente debe reenviar cada llamada *RPC* periódicamente hasta recibir el *ACK*.
- Comunicación entre un servidor y un cliente:
  - Fija: en compilación → puerto fijo.
  - Dinámica: negociación. El SO proporciona un *daemon* de *rendezvous* (o *matchmaker*) en un puerto *RPC* fijo. Primero el cliente envía un mensaje al *daemon* solicitando el puerto. Luego el cliente hace las llamadas *RPC* a ese puerto hasta que finalice el proceso.
- El esquema *RPC* es útil para implementar un sistema de archivos distribuidos, puede implementarse como un conjunto de demonios y clientes *RPC*.

Comunicación en sistemas cliente-servidor

Remote Procedure Calls



Sistema Distribuidos

Consultar transparencias del Profesor Prof. Dr. Javier Echaiz

https://cs.uns.edu.ar/~gd/soyd/clasesjavi/je.tar.gz