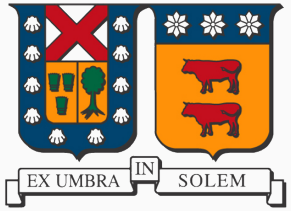


Sistemas Operativos

Procesos

Viktor Andrés Tapia Vásquez

Segundo Semestre 2021

Departamento de Informática, Campus SSJJ.

Índice de Contenidos

Definición

Definición

- El SO provee un entorno de ejecución concurrente de programas.
- Un programa es una entidad pasiva. Está almacenado en el disco.
- Un **proceso** es una abstracción del SO para ejecutar un programa.
- La ejecución comienza a través de una GUI, CLI, etc.
- El ejecutable es cargado en memoria principal.
- Existen de usuario y del mismo SO.

Definición

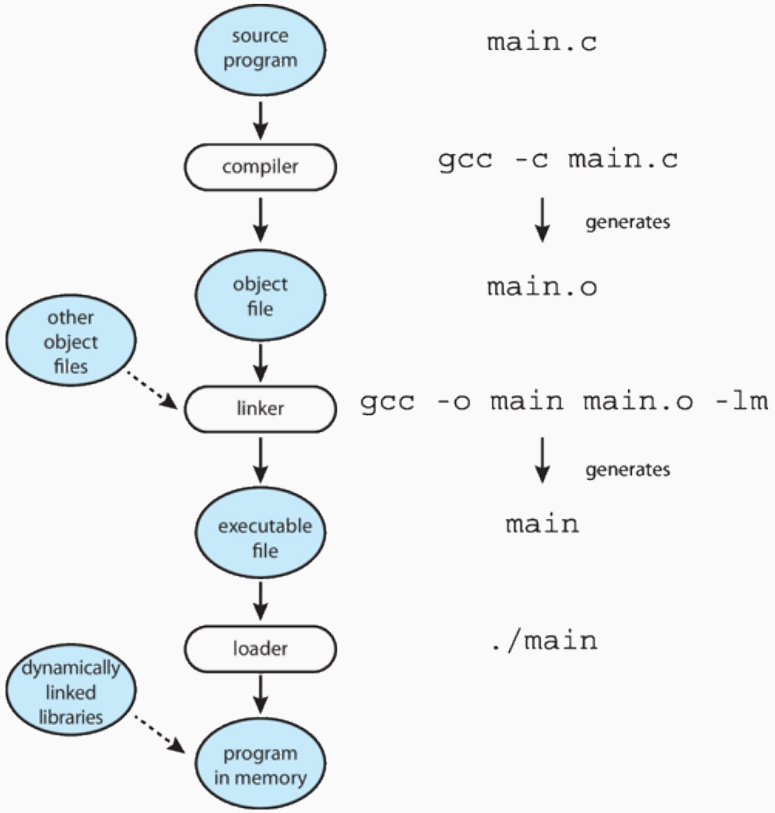


Figure 1: Compilación y Carga

Definición

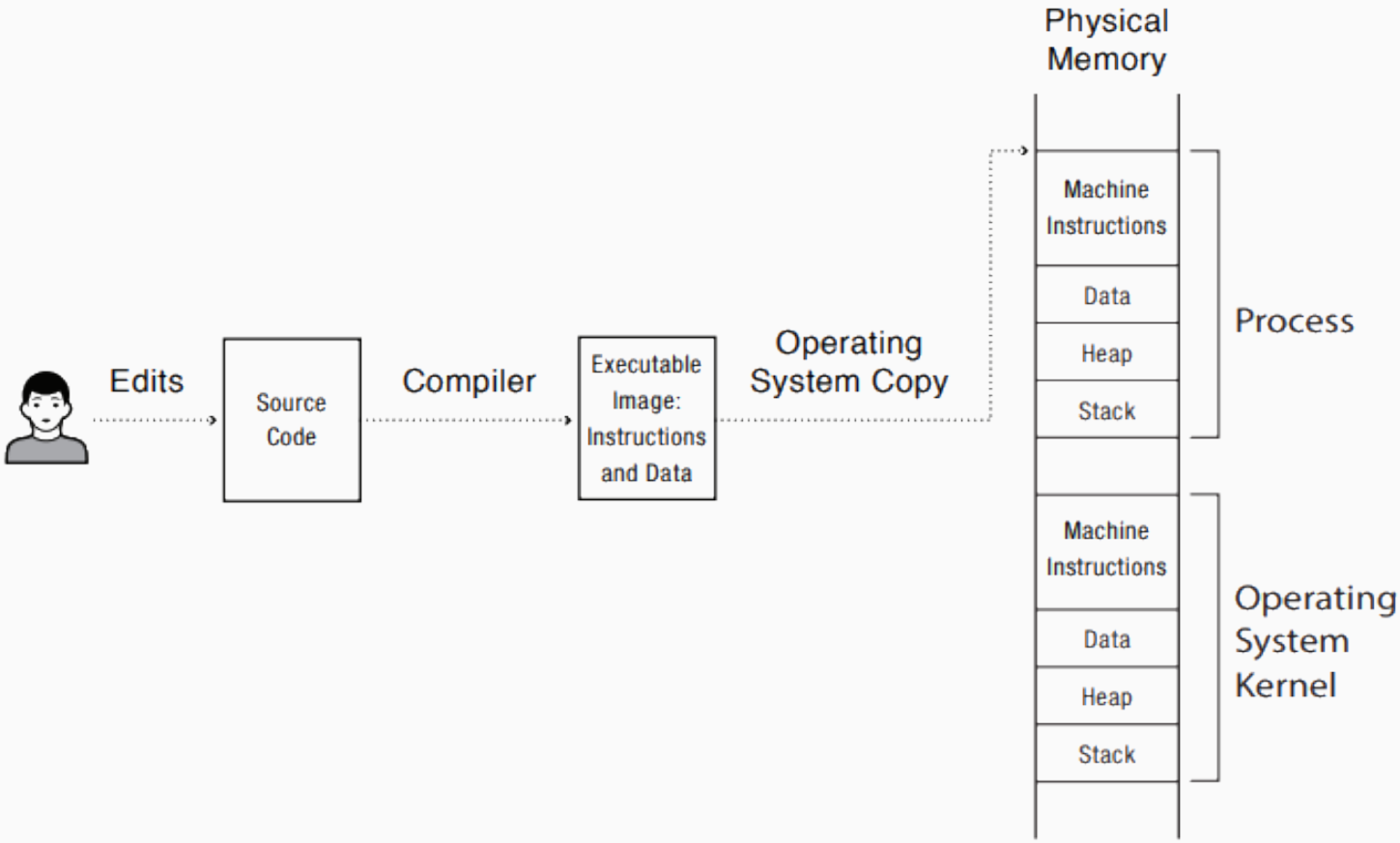


Figure 2: Abstracción de Proceso

Definición

Todo proceso está compuesto por:

- **Código:** Instrucciones a ejecutar.
- **Datos:** Variables globales, constantes, etc.
- **Stack:** Datos temporales, i.e: variables locales.
- **Heap:** Para asignar datos en tiempo de ejecución.

Definición

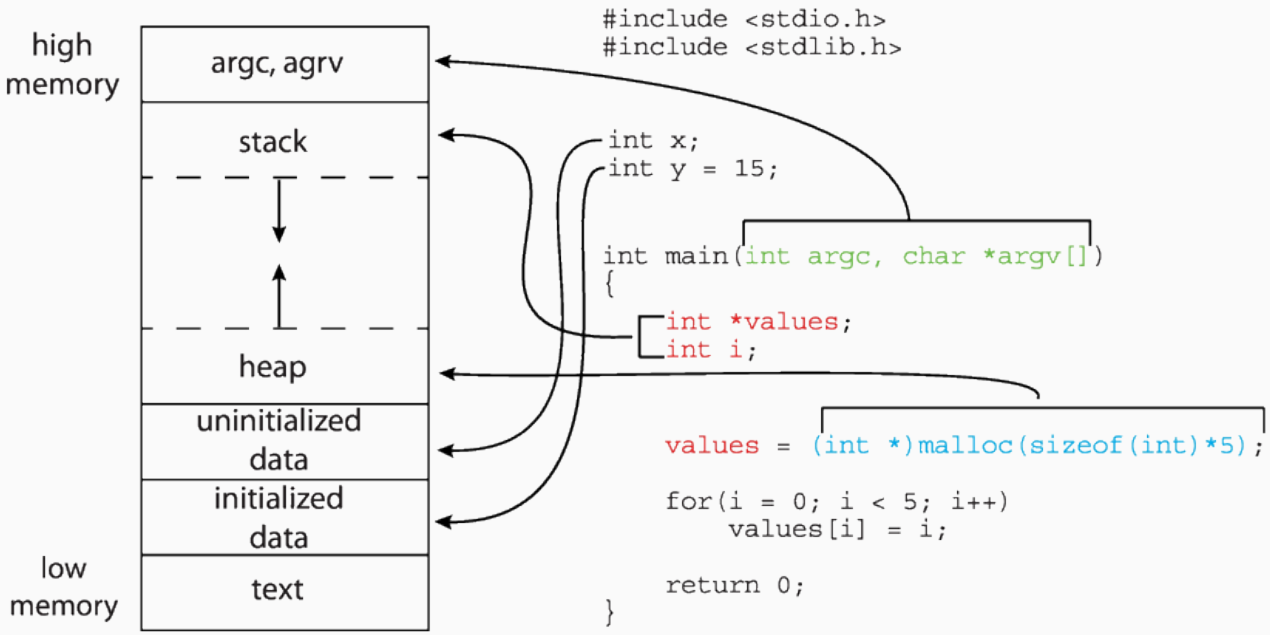


Figure 3: Programa en C mapeado a memoria

Definición

Aspectos adicionales del **stack**:

- Cuando se llama una función se debe guardar una estructura de datos llamada *stack frame*.
- Ésta contiene dirección de retorno, argumentos y variables locales.
- Se accede a través del stack pointer (registro).
- Entre otras cosas, permite llamadas recursivas a funciones.

Definición

Aspectos adicionales del **Heap**:

- Se asigna durante la ejecución. Es más lento que el stack.
- El programa debe liberar el espacio explícitamente.
- Su tamaño se ve limitado solo por la memoria virtual.

Definición

```
void liberar();
int *x;

void main(){
    if(true){
        x = malloc(sizeof(int));
    }
    *x = 1;
    liberar();
}

void liberar(){
    free(x);
}
```

Figure 4: Código de ejemplo

La región de memoria para x se asigna al momento de ejecutar el main. Permanece en ella hasta ejecutar el free.

Definición

Todo proceso cambia de estado:

- Nuevo.
- En ejecución.
- En espera.
- Listo.
- Finalizado.

Definición

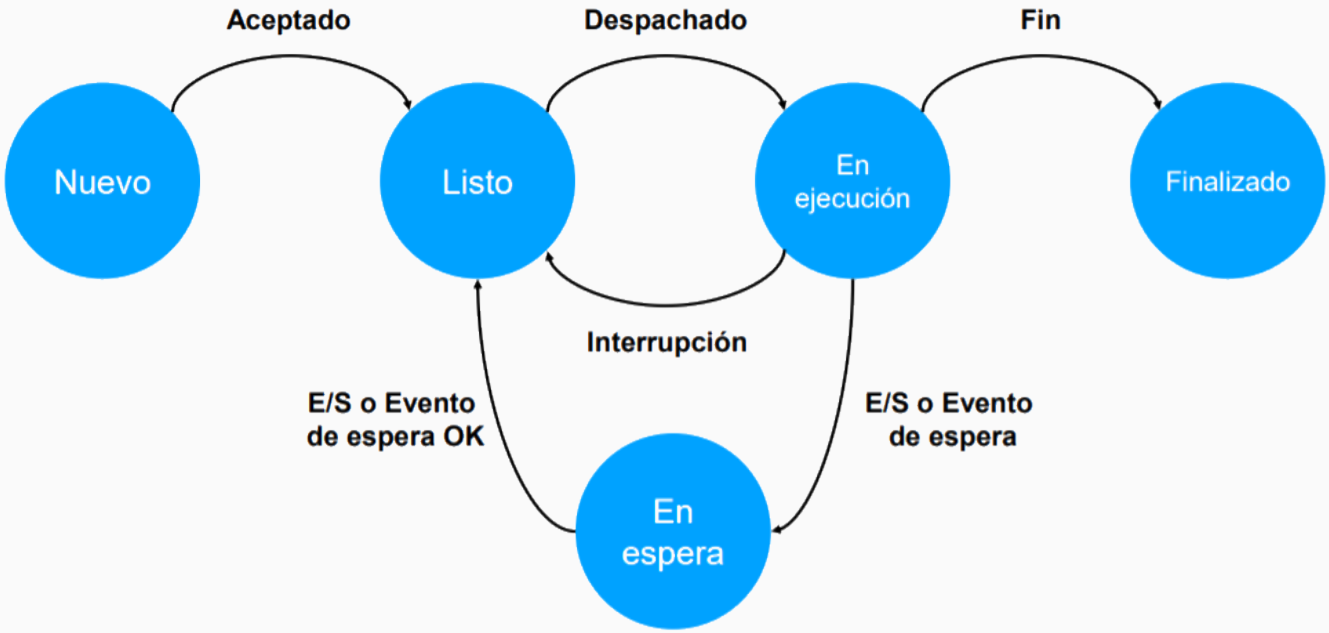


Figure 5: Diagrama de estados

Definición

La información de cada proceso se almacena en el *Process control block*.

- **Estado:** Los diferentes estados.
- **PC:** Dirección de la siguiente instrucción a ejecutar.
- **Registros CPU:** Contenido de los registros de apoyo.
- **Itineración:** Prioridades, punteros a colas, etc.
- **Contabilidad:** CPU utilizada, ciclos utilizados, etc.
- **E/S:** Recursos asignados, estado de uso, etc.

La PCB de cada proceso la almacena el Kernel del SO.

Itineración

Itineración

- **Itinerador:** Asigna la CPU a un proceso entre todos los disponibles.
- El cambio debe ser rápido, se busca maximizar el uso de la CPU.
- Será necesario disponer de colas de itineración para los procesos:
  - ✓ **Ready Queue:** En memoria principal listos para ejecutarse.
  - ✓ **Wait Queue:** Esperando algún evento.
- Los procesos a lo largo de su vida se van moviendo entre colas.

Itineración

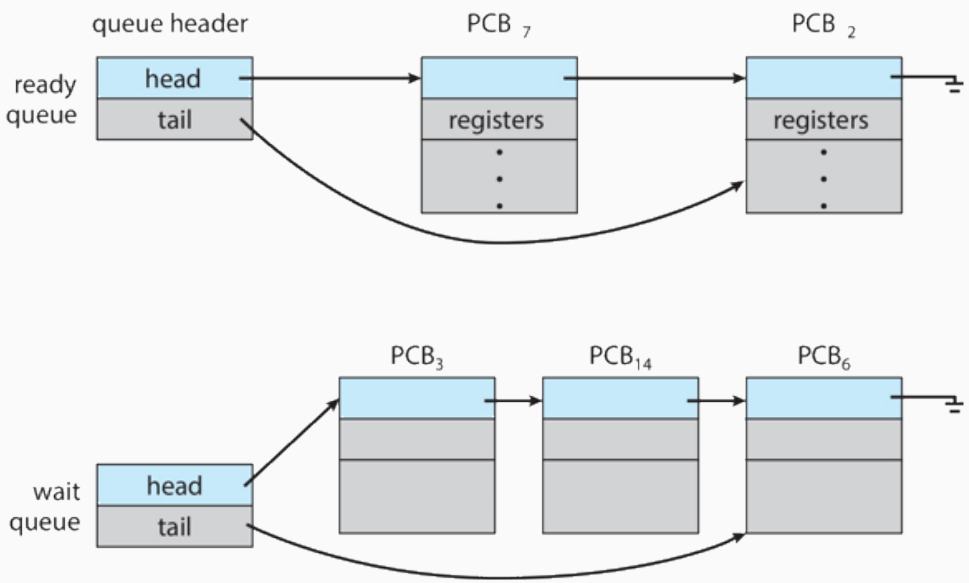


Figure 6: Implementación colas de itineración

Itineración

- Si un proceso que está en la cola ready es seleccionado para utilizar la CPU se dice que es *despachado*.
- Mientras se está ejecutando puede:
  - ✓ Solicitar E/S y ser asignado a las colas de espera.
  - ✓ Crear un nuevo proceso y esperar.
  - ✓ Ser desalojado como resultado de una interrupción.

Itineración

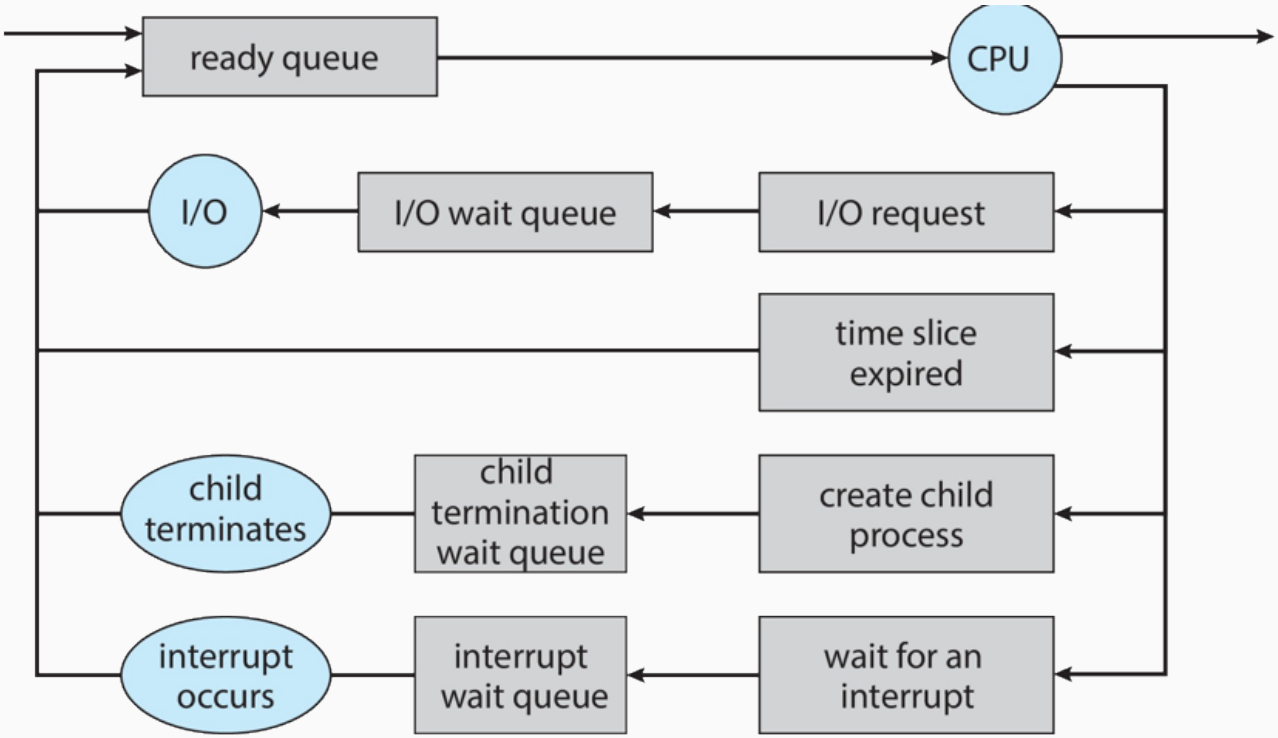


Figure 7: Interrupciones durante la ejecución de un proceso

Itineración

Existen dos tipos de itineradores:

- **Largo Plazo:** Carga los procesos en MP por primera vez.
- **Corto Plazo:** Asigna la CPU a los procesos de la cola ready.

¿Cuál de los dos controla el grado de multiprogramación?

Itineración

- El itinerador debe ser cuidadoso con los procesos que selecciona.
- Los procesos se pueden clasificar en:
  - ✓ **E/S:** Ocupa la mayor parte del tiempo en operaciones E/S.
  - ✓ **CPU:** Ocupa la mayor parte del tiempo en operaciones CPU.
- Se debe elegir un mix de procesos para mantener el equilibrio.

Itineración

- Caso Borde 1: Todos los procesos E/S.
  - ✓ Cola ready siempre vacía.
  - ✓ Itinerador de corto plazo con poca pega.
  - ✓ CPU Ociosa.
- Caso Borde 2: Todos los procesos CPU.
  - ✓ Colas de dispositivos vacías.
  - ✓ E/S poco utilizados.
  - ✓ CPU con sobrecarga.

Itineración

- **Context Switch:** Ocurre cuando la CPU cambia entre procesos.
- Se guarda el estado del proceso saliente y se carga el del entrante.
- El contexto de un proceso es representado por la PCB.
- El tiempo utilizado en realizar el cambio es desperdiciado.
- El tiempo depende del soporte de hardware de la máquina.

Itineración

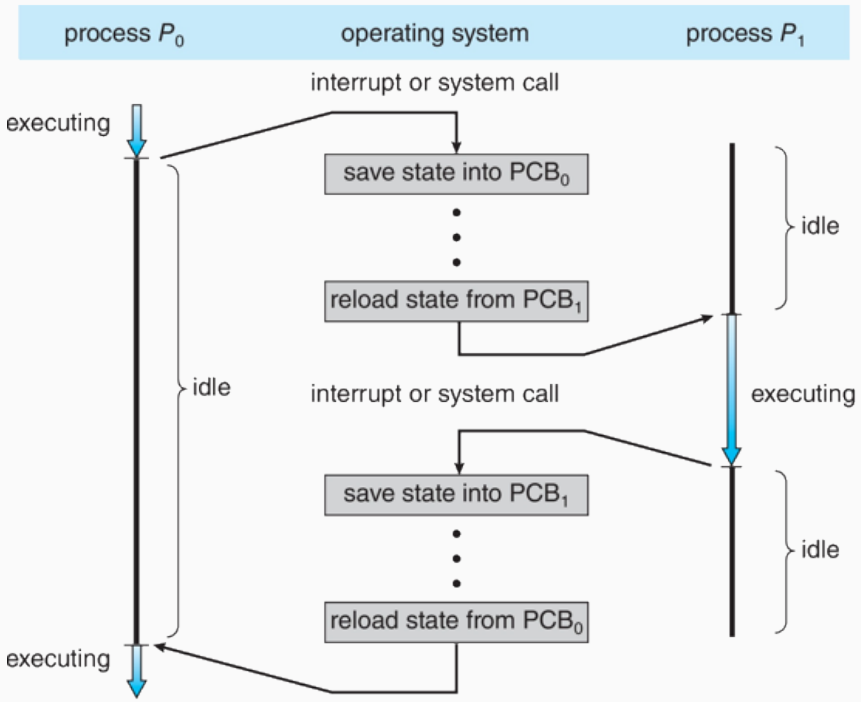


Figure 8: Context switch

Operaciones

Operaciones

Los SO's deben proveer acciones para trabajar con procesos.

- Primero lo primero, creación.
- Existen 3 tipos de eventos que la gatillan:
  - ✓ Arranque del sistema.
  - ✓ Proceso en ejecución crea uno.
  - ✓ Petición del usuario.

Operaciones

- Mientras se ejecuta, un proceso puede crear varios procesos.
- Para hacerlo debe utilizar llamadas al sistema.
- El proceso que crea será el **padre**, los procesos creados serán **hijos**.
- Cada hijo a su vez puede crear otros procesos.
- Tendremos una jerarquía de procesos.
- Unix y Windows usan un identificador unívoco (pid) para cada uno.

Operaciones

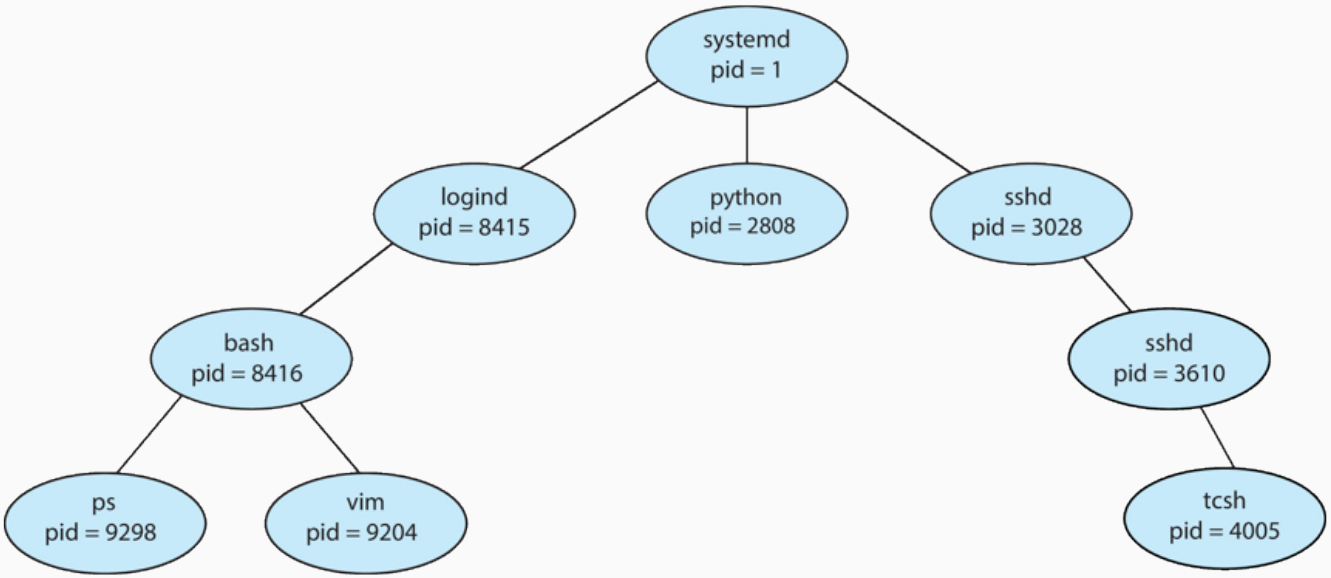


Figure 9: Árbol de procesos en Unix

Operaciones

- Recursos compartidos:
  - ✓ Padre e hijo comparten todos los recursos.
  - ✓ Los hijos pueden utilizar solo algunos
  - ✓ No comparten.
- Ejecución:
  - ✓ Padre e hijo se ejecutan concurrentemente.
  - ✓ El padre espera hasta que el hijo termine.
- Espacio direcciones:
  - ✓ El hijo es un duplicado del padre.
  - ✓ El hijo carga un nuevo programa.

Operaciones

En **Windows**: Se ejecuta una llamada al sistema para crear y ejecutar.

- Crear e inicializar la PCB en el kernel.
- Crear e inicializar el nuevo espacio de direcciones.
- Cargar el programa en el espacio.
- Copiar los argumentos en el espacio.
- Inicializar contexto de hw para comenzar ejecución.
- Informar al itinerador que el proceso está OK.

Operaciones

```
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
)
```

Figure 10: Creación de procesos en Windows

Operaciones

En **Unix**: Se utilizan las siguientes llamadas al sistema:

- **fork:** crea una copia del proceso que lo invoca y comienza su ejecución. No tiene argumentos.
- **exec:** cambia el programa que se está ejecutando por el proceso que se quiere activar.
- **wait:** espera a que termine un proceso.
- **signal:** notifica a otro proceso de agún evento.

Operaciones

ejemplo: El shell

- Lee una línea de comando desde la entrada y hace un fork para crear el proceso.
- Al hacer un fork(), Unix automáticamente duplica todos los archivos abiertos en el padre.
- El padre espera que el hijo termine antes de leer el siguiente comando.

Operaciones

La ejecución de un **fork** retorna:

- Un número negativo cuando existe un error.
- Pid $==$ 0 para el proceso hijo.
- Pid $>$ 0 (el pid del hijo) para el proceso padre.

Operaciones

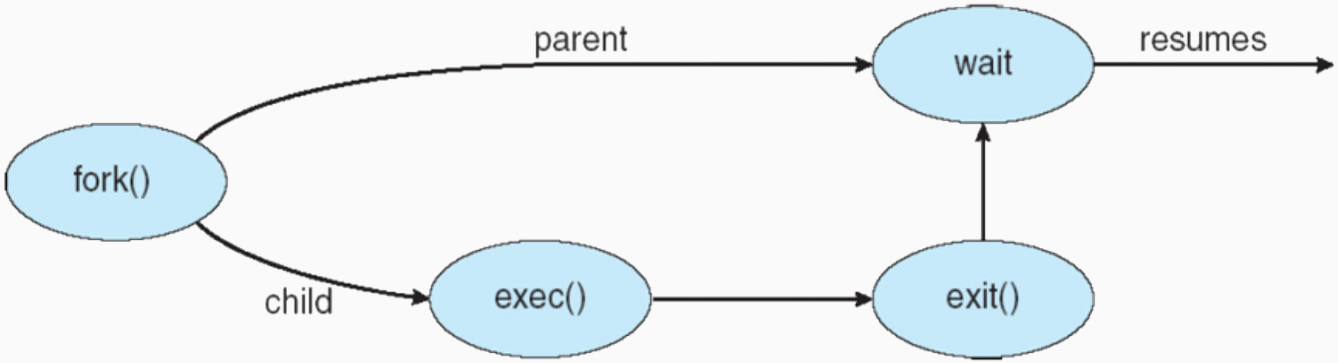


Figure 11: Fork en Unix

Operaciones

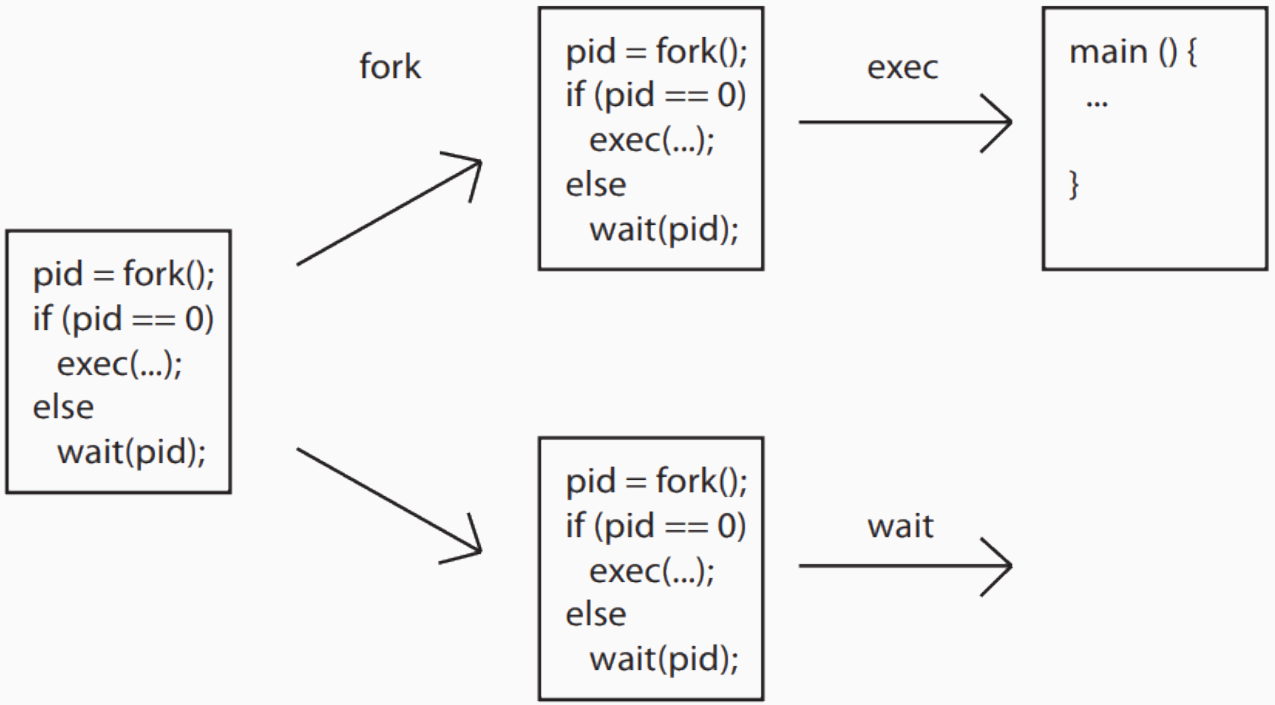


Figure 12: Fork en Unix

Operaciones

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    int child_pid = fork();
    if (child_pid == 0) {    // I'm the child process
        printf("I am process #%d\n", getpid());
        return 0;
    }
    else {                   // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
}
```

Figure 13: ¿Qué se imprime?

Operaciones

En **Unix**: Pasos del kernel para el fork

- Crear e inicializar la PCB en el kernel.
- Crear un nuevo espacio de direcciones.
- Inicializar el espacio de direcciones con una copia completa del espacio de direcciones del proceso padre.
- Heredar el contexto de ejecución del proceso padre.
- Informar al scheduler que un proceso está listo para ejecución.

Operaciones

En **Unix**: Pasos del kernel para el exec

- Cargar el programa en el espacio de direcciones actual.
- Copiar argumentos en memoria dentro del espacio de direcciones.
- Inicializar el contexto del hardware para comenzar ejecución en el punto de inicio.

Operaciones - Creación / Ejemplos

Para los siguientes ejemplos se solicita:

- Dibujar la jerarquía de procesos
- ¿Aparecen mensajes repetidos? ¿Por Qué?
- ¿Qué ocurre con el orden de finalización?

Operaciones

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

int main (int argc, char *argv[]) {

  int num;
  pid_t pid;

  for (num= 0; num< 3; num++) {
    pid= fork();
    printf ("Soy el proceso de PID %d y mi padre tiene %d de PID.\n",
            getpid(), getppid());
    if (pid!= 0)
      break;
    srandom(getpid());
    sleep (random() %3);
  }
  if (pid!= 0)
    printf ("Fin del proceso de PID %d.\n", wait (NULL));

  return 0;
}
```

Figure 14: Ejemplo I

Operaciones

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

int main (int argc, char *argv[]) {

  int num;
  pid_t pid;

  srandom(getpid());
  for (num= 0; num< 3; num++) {
    pid= fork();
    printf ("Soy el proceso de PID %d y mi padre tiene %d de PID.\n",
             getpid(), getppid());
    if (pid== 0)
      break;
  }
  if (pid== 0)
    sleep(random() %5);
  else
    for (num= 0; num< 3; num++)
      printf ("Fin del proceso de PID %d.\n", wait (NULL));

  return 0;
}
```

Figure 15: Ejemplo II

Operaciones

```
for (num= 0; num< 2; num++) {
  nuevo= fork();   /* 1 */
  if (nuevo== 0)
    break;
}
nuevo= fork();     /* 2 */
nuevo= fork();     /* 3 */
printf("Soy el proceso %d y mi padre es %d\n", getpid(), getppid
```

Figure 16: Ejemplo III

Operaciones

En **Unix**: Otras operaciones

- **Wait** suspende al padre hasta que el hijo termina, se cae (crash) o es terminado externamente. Como un padre puede crear muchos hijos, su parámetro es el pid del proceso hijo.
- **Signal** (kill) se utiliza para enviar a otro proceso una notificación (upcall). Se utiliza para terminar una aplicación, suspenderla temporalmente para debugging y reasumir después de una suspensión.

Operaciones

- Un proceso finaliza cuando ejecuta su última instrucción.
- Al hacerlo, solicta al SO que lo elimine utilizando exit().
- Los recursos que tenía asignados son liberados.
- Un padre puede terminar la ejecución de sus hijos:
  - ✓ El hijo excedió los recursos asignados.
  - ✓ La tarea del hijo ya no es necesaria.
  - ✓ Algunos SO no permiten que los hijos sigan si el padre termina.

Comunicación

Comunicación

- Los procesos pueden ser dependientes o independientes.
- Independientes no afectan ni se ven afectados por otros procesos.
- Cualquier proceso que comparte datos es cooperativo.
- Algunos beneficios:
  - ✓ Compartir información.
  - ✓ Cálculos más rápidos.
  - ✓ Modularidad y diseño.
  - ✓ Conveniencia.
- La cooperación requiere mencanismos de comunicación.

Comunicación

Existen dos mecanismos fundamentales:

- **Memoria Compartida:** Define una región común para compartir.
  - ✓ Máxima velocidad y conveniencia.
  - ✓ Se utilizan llamadas al sistema solo para crear la región.
  - ✓ Dicha región es parte del espacio de direcciones del proceso.
- **Paso de Mensajes:** Útil para pequeñas cantidades de datos.
  - ✓ Fácil de implementar.
  - ✓ Es lento dado que involucra la participación del Kernel.

Comunicación

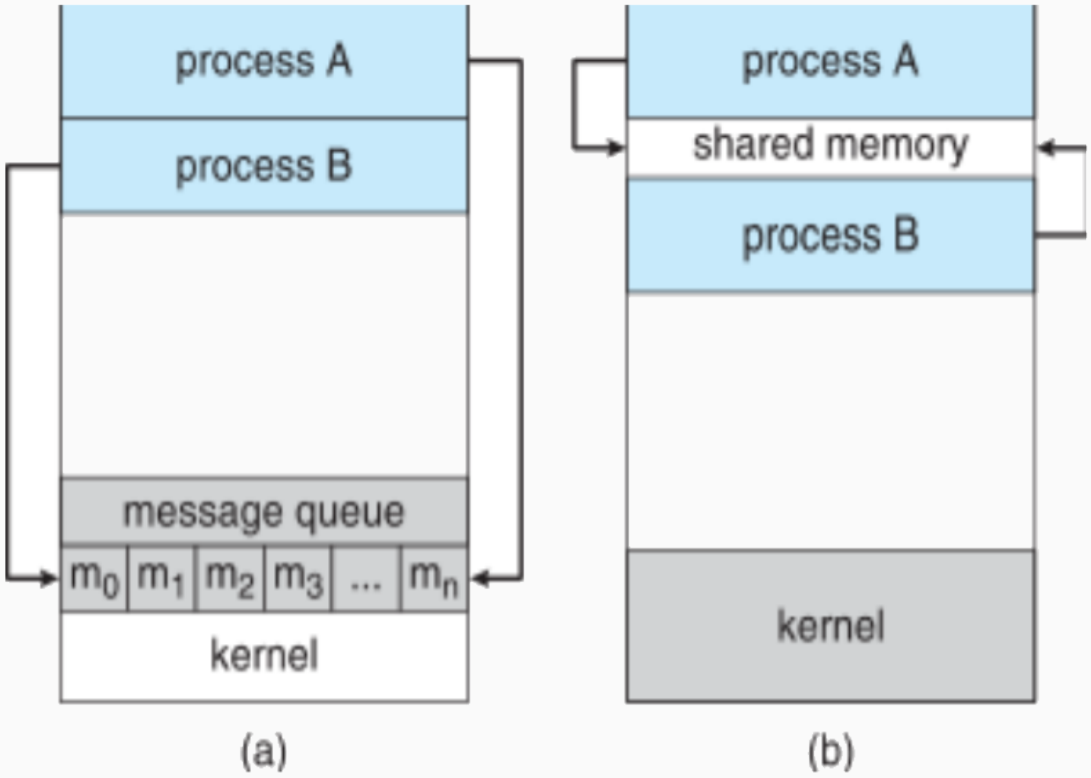


Figure 17: Mecanismos de comunicación

Comunicación

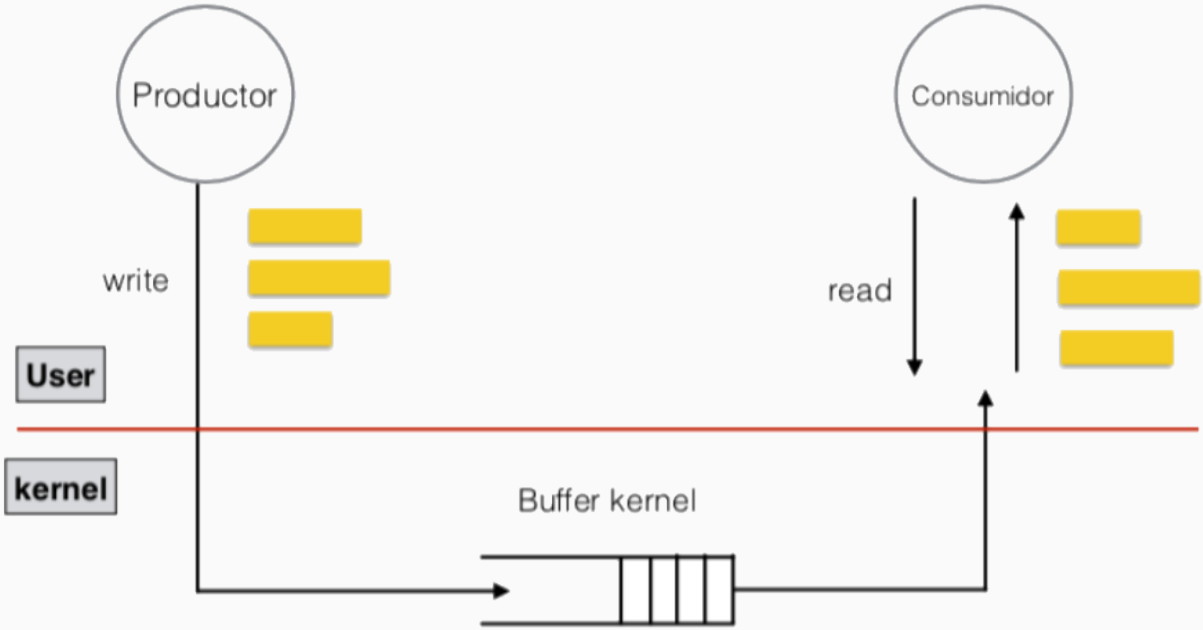


Figure 18: Ejemplo: Productor - Consumidor

Comunicación - Paso de Mensajes

- Procesos se comunican y sincronizan sin compartir memoria.
- No es necesario utilizar variables compartidas.
- Se usan dos mecanismos: send() & receive()
- Si P y Q quieren comunicarse necesitan:
  - ✓ Crear un enlace entre ellos.
  - ✓ Intercambiar mensajes con los métodos.

Comunicación - Paso de Mensajes

Problemas en la implementación:

- ¿Cómo se establecen los enlaces?
- ¿Un enlace puede asociarse a más de dos procesos?
- ¿Cuantos enlaces entre los mismos procesos?
- ¿Cuál es la capacidad del enlace?
- ¿El enlace es uni o bi direccional?

Comunicación - Paso de Mensajes

Directo: El proceso debe nombrar explícitamente al receptor.

- Se establece automáticamente entre cada par de procesos.
- Cada enlace se establece entre solo dos procesos.
- send(P, mssg) & receive(Q, mssg).
- El enlace puede ser bi direccional.

Comunicación - Paso de Mensajes

Indirecto: Los mensajes son enviados a buzones.

- Cada buzón tiene un id único.
- Los procesos se comunican solo sin comparten un buzón.
- El enlace se establece solo si se cumple lo anterior.
- Un enlace puede asociarse con muchos procesos.
- Entre cada par de procesos puede haber más de un enlace.

Comunicación - Paso de Mensajes

Operaciones:

- Crear un buzón.
- send() & receive() a través del buzón.
- Destruir el buzón.

Métodos:

- send(A, mssg)
- receive(A, mssg)

Comunicación - Paso de Mensajes

Este mecanismo puede ser bloqueante o no bloqueante.

- **Bloqueante:** Sincrónico.
  - ✓ send() bloquea hasta que se recibe.
  - ✓ receive() bloquea hasta que que esté disponible.
- **No bloqueante:** Asíncrono.
  - ✓ send() envía y continua.
  - ✓ receive() recibe null o el mensaje.

Comunicación - Paso de Mensajes

Sea directo o indirecto, los mensajes residen en una cola temporal:

- **Zero:** Proceso que envía debe esperar al receptor.
- **Acotada:** N mensajes.
- **Sin limites:** Nunca se espera.

Comunicación - Pipes

- Primer mecanismo de comunicación de Unix.
- **Simples:** Relación padre e hijo. No se puede acceder desde afuera.
- **Nombrados:** No es necesaria la relación anterior.
  - ✓ Comunicación bi direccional.
  - ✓ Muchos procesos usando el pipe.
  - ✓ Windows y Unix los proveen.

Comunicación - Pipes

- En Unix se crea un pie usando: pipe(int fd[]).
- Se utiliza el descriptor:
  - ✓ fd[0] para leer.
  - ✓ fd[1] para escribir.
  - ✓ Se utilizan llamadas al sistema read() y write().

Comunicación - Pipes

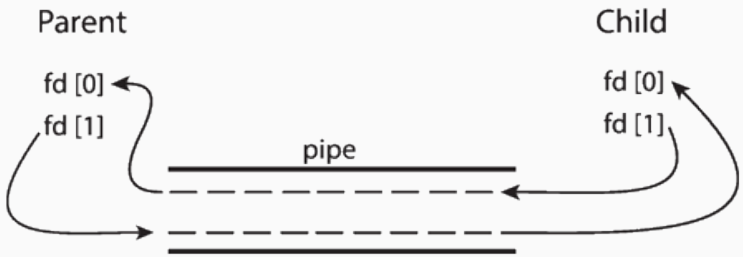


Figure 19: Simple Pipe

Comunicación - Pipes

Ejemplo

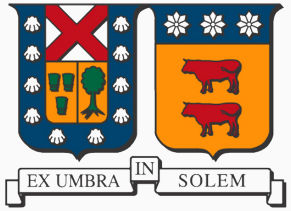
```c
/* IPC Ordinary pipes in UNIX*/
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFF_SIZE 25
#define READ 0
#define WRITE 1

int main(void)
{
    char write_msg[BUFF_SIZE] = "Viva Chile\n";
    char read_msg[BUFF_SIZE];
    int fd[2];
    pid_t pid;
    /*crear el pipe*/
    pipe(fd);

    /* se crea un hijo */
    pid = fork();
    if (pid > 0) {
        close(fd[READ]); /*no se usa*/
        write(fd[WRITE], write_msg, strlen(write_msg)+1);
        close(fd[WRITE]); /* Ya no se usa*/
    }
    else { /*proceso hijo*/
        close(fd[WRITE]); /*no se usa*/
        read(fd[READ], read_msg, BUFF_SIZE);
        printf("El mensaje dice: %s", read_msg);
        close(fd[READ]);
    }
    return 0;
}
```

Figure 20: Ejemplo



Sistemas Operativos

Procesos

Viktor Andrés Tapia Vásquez

Segundo Semestre 2021

Departamento de Informática, Campus SSJJ.