

MC Hilda Castillo Zacatelco
hildacz@gmail.com

PROCESOS

Definición y atributos

Un **proceso** es la instancia de un programa en ejecución. Desde el punto de vista del SO, un proceso es la entidad más pequeña a quien le proporciona un servicio (despacho)

•Varios procesos pueden correr el mismo programa, pero cada uno de ellos es un proceso distinto con su propio **estado**.

•Un proceso consiste de **código**, **datos** y demás atributos.

•El código se compone de **instrucciones de máquina** y **llamados a servicios del sistema.**

Definición y atributos

- El **estado** de un proceso consiste de al menos:
  - El código para el programa ejecutándose
  - Los datos estáticos para el programa ejecutándose
  - Espacio para datos dinámicos
  - El contador del programa, indicando la próxima instrucción
  - Un stack de ejecución con el stack pointer
  - Valores de registros de CPU
  - Un conjunto de recursos en uso del SO (archivos abiertos, conexiones a otros programas, etc.)
  - El estado del proceso

Estados de un proceso

Cada proceso tiene un estado de ejecución el cual indica lo que esta actualmente haciendo. El SO ve la ejecución de un proceso típico como una sucesión de estados.

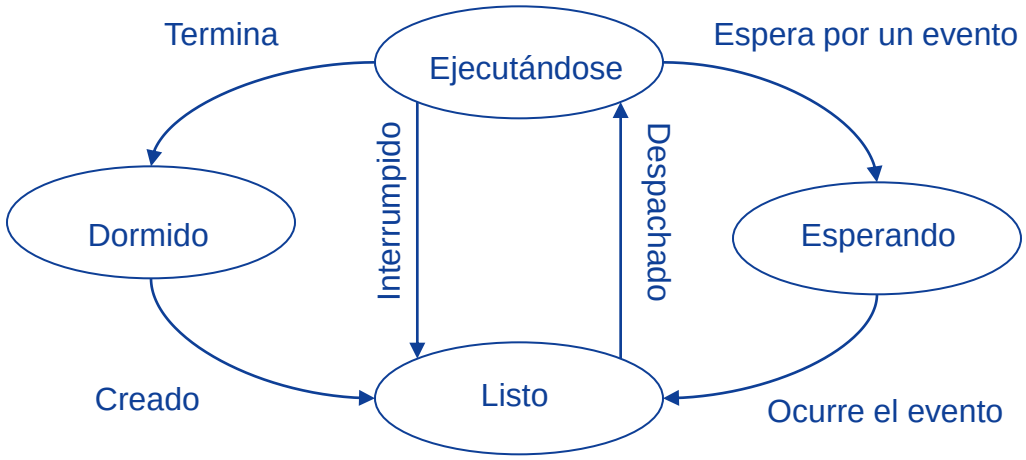


Diagrama de Transición de Estados

Estados de un proceso

✓*Dormido*. Proceso inexistente. Sólo se indica para uso de una tabla de procesos

✓*Listo*. Cuando un proceso es creado se pasa a una cola de procesos listos para ejecutarse. Los procesos listos no tiene recursos asignados

✓*Ejecutándose*. El proceso tiene todos los recursos necesarios para su ejecución. Si se dispone de un solo procesador, solamente un proceso puede estar ejecutándose. Este puede ser interrumpido y pasar a la cola de listos

✓*Esperando*. Espera a que ocurra un evento externo, no tiene recursos asignados. Cuando el evento ocurre el proceso suspendudi normalmente pasa a la cola de listos a esperar que se le otorgue nuevamente un turno para el uso del procesador

Tabla de control de procesos

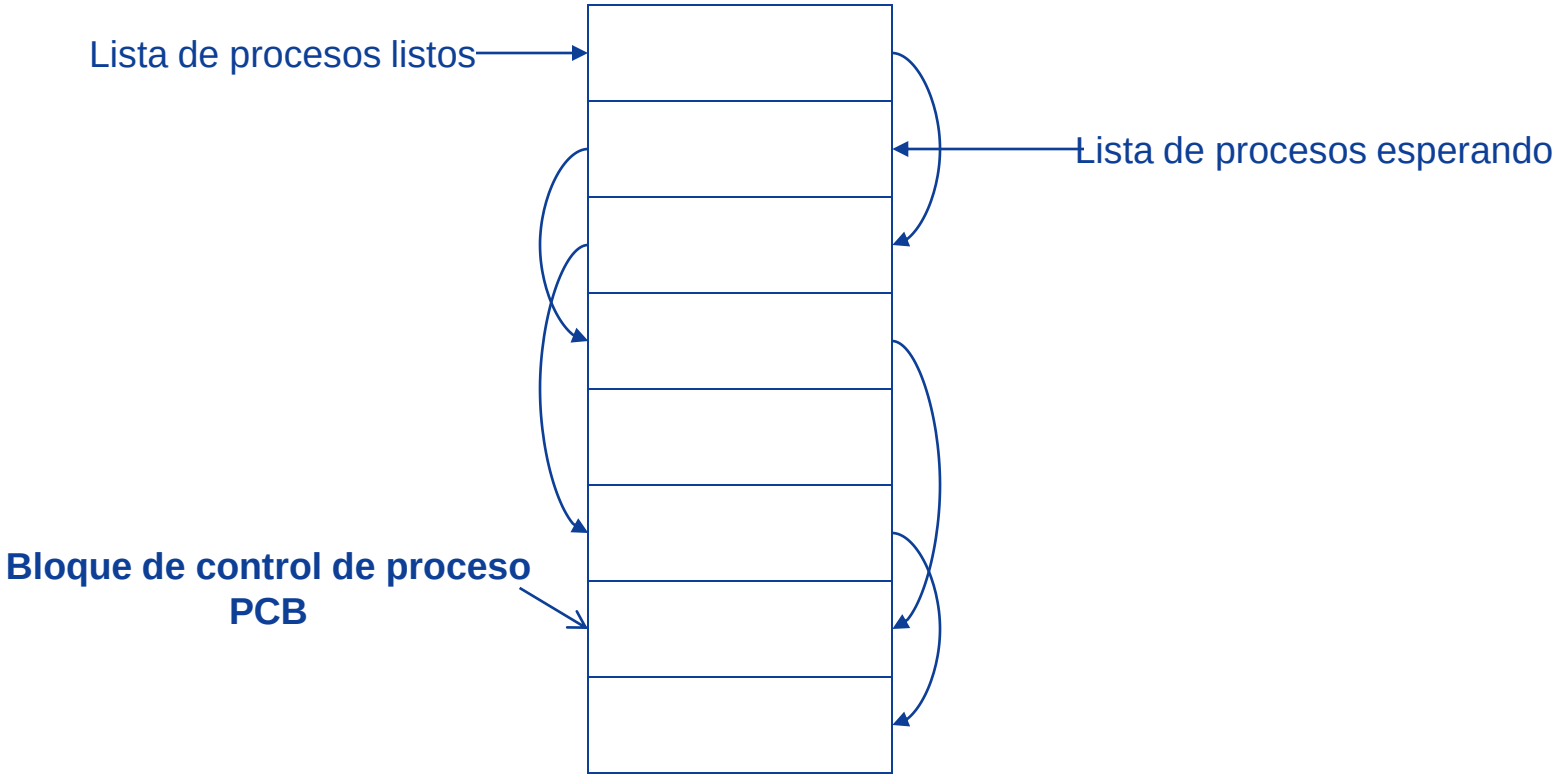
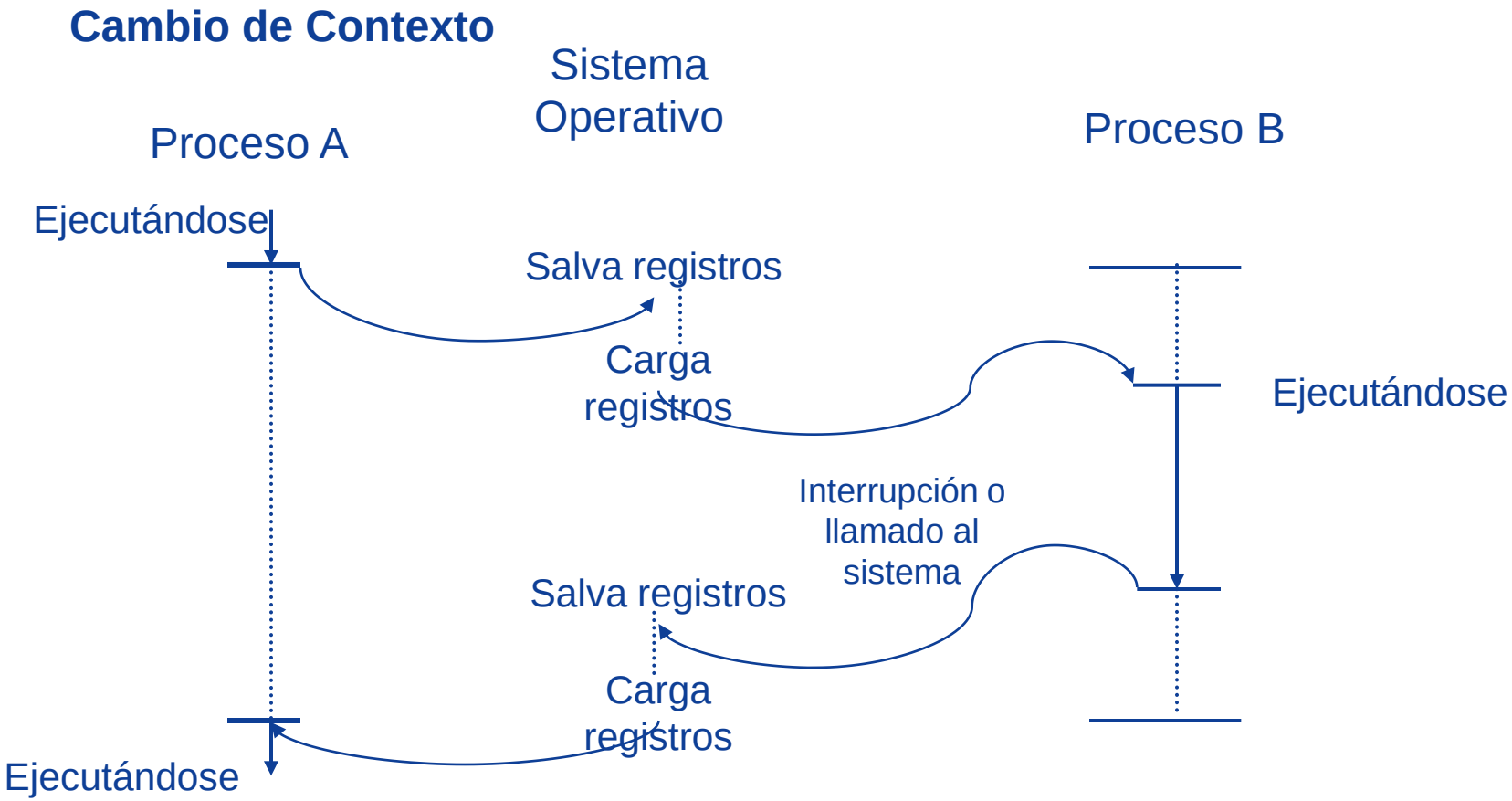


Tabla de Control de procesos

Cambio de contexto



Servicios del sistema para el manejo de procesos

Los sistemas operativos que manejan procesos deben tener llamadas al sistema como:

- Crear proceso
- Borrar proceso
- Abortar un proceso
- Bloquear proceso
- Asignar prioridad a un proceso
- Obtener atributos del proceso

En el SO Linux existen llamadas al sistema para el manejo de procesos tales como: fork, getpid, getppid, wait, etc.

Llamadas al sistema para manejo de procesos en el sistema operativo LINUX (de acuerdo al estándar POSIX).

a) **fork**. Función que crea un proceso, el cual será una réplica de su padre.

Devuelve 0 al proceso en caso de que la llamada haya sido exitosa, y un número mayor que cero al proceso padre (que será el identificador del proceso hijo).

Retorna -1 en caso de error.

El proceso padre no comparte direcciones de memoria con su hijo, cada uno de ellos se ejecuta en direcciones de memoria diferentes.

Sintaxis:

```
pid_t  fork( );
```

Llamadas al sistema para manejo de procesos en el sistema operativo LINUX (de acuerdo al estándar POSIX).

Enseguida se muestra un programa que crea un proceso hijo.

```
main( )  {
            fork( );
}
```

El tiempo de ejecución del proceso hijo se iguala a cero.
Todas las alarmas pendientes se desactivan en el proceso hijo.
El conjunto de señales pendientes se pone a vacío.

Servicios para la identificación de procesos.

b) **Getpid** Obtiene el identificador de proceso del proceso que invoca la función getpid() (obtiene su propio pid).

Sintaxis:

```
pid_t getpid();
```

c) **Getppid**. Obtiene el pid de su proceso padre.

Sintaxis:

```
pid_t getppid();
```

Ejemplos

Ejemplo: Programa que crea un proceso hijo.

```
#include <unistd.h>

main(){
  if (fork()>=0)
            printf("Creación de proceso exitosa "");   (1)
  else printf("Error! No se  ha podido  crear el proceso");   (2)
  exit(0);
}
```

En el caso de que se haya creado un proceso hijo entonces tanto el padre como el proceso hijo escriben el mensaje (1), si el fork falla entonces el padre escribe el mensaje (2).

Ejemplos

```
1.  #include <unistd.h>
2.  main(){
3.  int pid;.

4.        pid=fork();
5.         if (pid>0)  printf("Proceso padre ... %d ", getpid());   (1)
6.         else if (pid==0) printf("Proceso hijo...%d  padre=%d " , getpid(),
getppid());  (2)
7.         else printf("Error! No se  ha podido  crear el proceso");   (3)
8.         exit(0);
9.    }
```

Ejemplos

```
main(){
   int pid;

switch (pid=fork()) {
      case 0: while (1) printf("\nProceso hijo %d  padre=%d", getpid(),
getppid());
      case –1: printf ("Error al crear el proceso"); exit(1);
      default: while(1) printf("\nProceso padre %d", getpid());
   }
}
```

Ejemplos

Ejemplo: Programa que crea una cadena de procesos

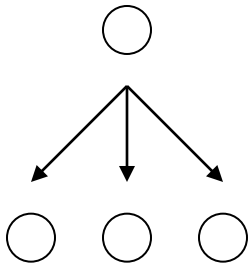
```
#include <unistd.h>
#include <sys/types.h>
main(){
  pid_t pid;
  int i, n=3;

  for (i=0; i<n; i++) {
     pid=fork();
     if (pid != 0) break;
  }
  printf("\nEl padre del proceso %d es %d", getpid(),getppid());
}
```

Ejemplos

```
#include <unistd.h>
#include <sys/types.h>
main(){
  pid_t pid;
  int i, n=3;

  for (i=0; i<n; i++) {
     pid=fork();
     if (pid == 0) break;
  }
  printf("\nEl padre del proceso %d es %d", getpid(),getppid());
}
```



EXEC

En POSIX existe una familia de funciones exec, cuyos prototipos se muestran a continuación:

int execl(char *path, char *arg, ...);

int execv(char *path, char *argv[]);

int execle(char *path, char *arg, ...);

int execve(char *path, char *argv[], char *envp[]);

int execlp(char *file, const char *arg, ...);

int execvp(char *file, char *argv[]);

argv es un vector de cadenas que representan los argumentos pasados al programa, el primer elemento del vector, argv[0], es el nombre del programa, argv debe terminar con NULL.

path apunta al nombre del archivo ejecutable

file se utiliza para construir el nombre del archivo ejecutable

envp apunta al entorno que se pasará al nuevo proceso y se obtiene de la variable externa environ.

Ejemplo con EXEC

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

main() {
   pid_t pid;
   char *argumentos[3];

   argumentos[0]="ls";
   argumentos[1]="-l";
   argumentos[2]=NULL;
   pid=fork();
   switch(pid) {
      case –1: exit(-1);
      case 0:
          execvp(argumentos[0], argumentos);
          perror("exec");
          break;
      default: printf("Proceso padre\n");
   }
}
```

Esperar por la finalización de un proceso hijo

Esperar por la finalización de un proceso hijo.

Permite a un proceso padre esperar hasta que termine la ejecución de un proceso hijo.

Existen dos formas de invocar este servicio:

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);

La llamada wait suspende la ejecución del proceso hasta que finaliza la ejecución de uno de sus procesos hijos. Esta función devuelve el identificador del proceso hijo cuya ejecución ha finalizado. Si status es distinto de NULL, entonces se almacena en esta variable información relativa al proceso que ha terminado.

Si el hijo retornó un valor desde la función main, utilizando la sentencia return, o pasó un valor como argumento a exit, éste valor puede ser obtenido por medio de la variable utilizando las siguientes macros definidas en el archivo de cabecera **sys/wait.h**

Esperar por la finalización de un proceso hijo

- **WIFEXITED(status),** devuelve un valor verdadero (distinto de cero) si el hijo terminó normalmente.
- **WEXITSTATUS(status)**, permite obtener el valor devuelto por el proceso hijo en la llamada exit o el valor devuelto en la función main, utilizando la sentencia return. Esta macro solo puede ser utilizada cuando WIFEXITED devuelve un valor verdadero (distinto de cero).
- **WIFSIGNALED(status)**, devuelve un valor verdadero si el proceso hijo finalizó su ejecución como consecuencia de la recepción de una señal para la cual no se había programado manejador.
- **WTERMSIG(status)**, devuelve el número de la señal que provocó la finalización del proceso hijo. Esta macro sólo puede utilizarse si WIFSIGNALED devuelve un valor verdadero.

Ejemplo wait

Programa que imprime información sobre el estado de terminación de un proceso hijo.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
main(int argc, char **argv) {
   pid_t pid;
   int valor;

   pid=fork();
   switch(pid) {
      case –1: exit(1);
      case 0: if (execvp(argv[1], &argv[1]) < 0)
                 perror(“exec”);
      default: while(wait(&valor) != pid);
                  if (valor == 0)
                    printf(“El mandato se ejecuto de forma normal\n”);
                 else {
                    if (WIFEXITED(valor))
                       printf(“El hijo termino normalmente y su valor devuelto fue %d\n”,
     WEXITEDSTATUS(valor));
                    if (WIFSIGNALED(valor))
                       printf(“El hijo termino al recibir la señal %d\n”, WTERMSIG(valor));
                 } //else
         } //switch
}
```

Comunicación entre procesos

Es deseable construir software que consista de procesos diferentes cooperativos, en vez de un solo programa, por lo tanto es necesario tener mecanismo de comunicación entre procesos.

Las tres técnicas más ampliamente usadas son:

- señales
- pipes
- fifos.

en linux, utilizando lenguaje de programación se puede utilizar **memoria compartida (shmget, shmat)**.

Cuando los procesos comparten memoria, para procesos en diferentes computadores se utilizan **sockets**.

Comunicación entre procesos

- ***Condiciones de competencia***

Son las situaciones en las que dos o más procesos leen o escriben en ciertos datos compartidos y el resultado final depende de quién ejecuta qué y en qué momento

- ***Exclusión mutua***

Forma de garantizar que si un proceso utiliza una variable o archivo compartidos, los demás procesos no puedan utilizarlos. Forma de prohibir que más de un proceso lea o escriba en los datos compartidos a la vez

- ***Secciones o regiones críticas***

Parte del programa en el cual se tiene acceso a la memoria compartida

Comunicación entre procesos

Para implantar exclusión mutua en los programas, se pueden utilizar técnicas como:

- Semáforos
- Monitores

Llamadas al sistema para manejo de señales

signal. Asocia una acción particular con una señal.

Sintaxis:

signal(señal, acción);

kill . Envía una señal a un proceso especificado en los argumentos.

Sintaxis:

Retorna=kill(pid, señal);

Ejemplo

Programa que atrapa la señal SIGINT durante los primeros 2 segundos, después la señal SIGINT restablece su acción por default.

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

void tratamiento ( ) {
  printf (“\n Atrapada “);
}

main( ) {
  signal (SIGINT, tratamiento);
  sleep(2);
  signal (SIGINT, SIG_DFL);
}
```

Ejemplo

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

void tratar_alarma(){
     printf("\nActivada");
}

main() {
   struct sigaction act;
   act.sa_handler=(void *)tratar_alarma;
   act.sa_flags=0;

   sigemptyset(&act.sa_mask);
   sigaddset(&act.sa_mask, SIGINT);
   sigaction(SIGALRM, &act, NULL);

   for (;;) {
      alarm(3);
      pause();
   }
}
```

DESPACHO

Despacho

Conjunto de políticas y mecanismos construidos en el sistema operativo que gobiernan el orden en el cual se realiza la carga de trabajo del sistema

Despachador

Módulo del SO que selecciona el siguiente trabajo a ser admitido en el sistema y el siguiente proceso a ejecutarse

Tipos de despachadores

En un sistema operativo pueden actuar tres tipos diferentes de despachadores:

1. De largo plazo
2. De mediano plazo
3. De corto plazo

El despachador de largo plazo

Trabaja con la cola de los trabajos en lotes y selecciona el siguiente trabajo de lotes a ejecutarse

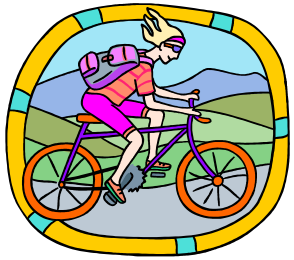
Su objetivo es proporcionar una mezcla balanceada de trabajos al despachador de corto plazo

Es invocado cada vez que un proceso termina y abandona el sistema

Se encarga de la transición de un proceso del estado de dormido al estado de listo

Tipos de despachadores

El despachador de mediano plazo



Se encarga de manejar los procesos que temporalmente se han enviado a memoria secundaria. No afecta al proceso mientras permanezca suspendido

Selecciona al siguiente proceso enviado a memoria secundaria y lo coloca en la cola de listos

Se encarga de la transición suspendido a listo

El despachador de corto plazo

Asigna el CPU entre los procesos listos en memoria principal

Se encarga de las transiciones de listo a ejecutándose

Se invoca cada vez que ocurre un cambio de proceso en el estado de ejecutándose

Criterios de eficiencia del despachador

- Porcentaje de utilización del CPU
- Throughput (cantidad de trabajo terminado por una unidad de tiempo)
- Tiempo de terminación
- Tiempo de espera
- Tiempo de respuesta

Algoritmos de despacho

- **FIFO**
- Prioridades
- **Round Robin (rebanadas de tiempo)**
- **Colas múltiples**
- **Lotería**
- **Dos niveles**
- **El trabajo mas corto primero**
- **Calendarizacíón garantizada**