TEMA 2: PROCESOS E HILOS

- El concepto más importante de los SOs es el de **proceso**, una **abstracción** que permite **operar concurrentemente** incluso cuando sólo hay una CPU en el sistema.

PROCESOS

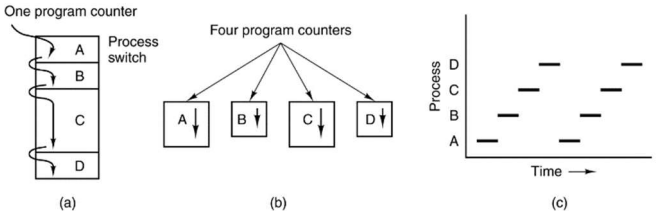
- Un PROCESO es una **instancia** de un programa en ejecución, incluyendo los valores actuales del **contador de programa**, **registros** y **variables**.
- Cada proceso se identifica gracias a su PID (identificador de proceso).
- La ejecución de varios procesos a la vez se conoce como **multiprogramación** y se basa en que:
  - En concepto, cada proceso tiene su propia CPU virtual.
  - En realidad, los procesos compiten por una o varias CPUs reales, que conmutan de un proceso a otro con rapidez.
    - Así, aunque en todos los instantes la CPU está ejecutando un único proceso, da una apariencia de **pseudoparalelismo**, pues en el transcurso de un breve intervalo de tiempo trabaja en varios procesos.
    - El algoritmo de **planificación de procesos** determina cuándo se debe detener el trabajo en un proceso para dar servicio a otro y qué proceso será el nuevo.

PROCESO vs PROGRAMA

- PROGRAMA → secuencia de **instrucciones** y **estructuras de datos** necesarias para la ejecución. Se almacena en un fichero ejecutable en código máquina.
- PROCESO → **programa en ejecución**. Incluye código, datos, pilas, espacio direcciones, señales, archivos, etc.

MODELO DE PROCESO

- En este modelo, todo el software en ejecución se organiza como un conjunto de **procesos secuenciales**.
- Cada proceso secuencial tiene su propio **PC lógico** que le permite ejecutarse de manera **aparentemente independiente** de los demás.
  - En realidad, como sólo hay un PC físico, cuando se ejecuta un proceso se carga su PC lógico en el real.
    Cuando para de ejecutarse, el PC actual se guarda en el PC lógico del proceso almacenado en memoria.



GESTIÓN DE PROCESOS

CREACIÓN DE PROCESOS

Hay 3 **maneras de crear** un proceso:

– Durante el arranque del sistema.
  - Se crean procesos { en **primer plano** → interactúan con los usuarios. / en **segundo plano** (DAEMONS) → realizan una función específica.

– La ejecución, desde otro proceso, de una llamada al sistema para la creación de procesos.
  - En UNIX, la única syscall que hace esto es $fork$.

– Por petición de un usuario.
  - Los usuarios inician un programa (y por lo tanto un proceso) escribiendo un comando o haciendo doble clic sobre un icono.

Hay 3 **tipos** de proceso:

- Procesos de kernel.
- Procesos de usuario.
- Daemons o demonios.

SYSCALL FORK

- La syscall $fork()$ se usa para crear un proceso que es una copia exacta del proceso padre.
  – Su salida es { para el padre → pid del hijo. / para el hijo → 0. / en caso de error → -1.
- Los procesos padre e hijo tienen la misma **memoria**, las mismas **variables** y los mismos **archivos abiertos**.
  Sin embargo, tienen **espacios de direcciones** distintos → si uno de ellos modifica una palabra en su espacio de direcciones, la modificación no es visible para el otro.
  - CARRERA CRÍTICA → como el padre y el hijo comparten el mismo puntero en el archivo, si ambos actual sobre él al mismo tiempo pueden aparecer errores de lectura o escritura, pues es imposible predecir cuál ejecutará primero el SO.

```
main() {
    int par, x=0;
    if ((par=fork())==-1){
        printf("error en la ejecución del fork");
        exit(0); }
    else if par==0 {  /* proceso hijo
        x=x+2;
        printf("\nproceso hijo, x= %d\n",x); }
     else   /* proceso padre
        printf("\nproceso proceso padre, x= %d\n",x);
     printf("Finalizar\n");}
```

```
int fichero1, fichero2;
char caracter;

main(int argc, char* argv[])
{
  if (argc !=3)
      exit(1)
  fichero1=open(argv[1], 0444);
  fichero2=open(argv[2], 0666);
  if (fichero1==-1)
      printf("\nError al abrir el fichero 1\n");
  if (fichero2==-1)
      printf("\nError al abrir el fichero 2\n");
  fork();
  leer_escribir();
  exit(0);
}

int leer_escribir() {
  for (;;) {
      if(read(fichero1,&caracter,1)!=1)
          return(0);
      write(fichero2,&caracter,1);}}
```

TERMINACIÓN DE PROCESOS

Hay 4 **maneras de finalizar** un proceso:

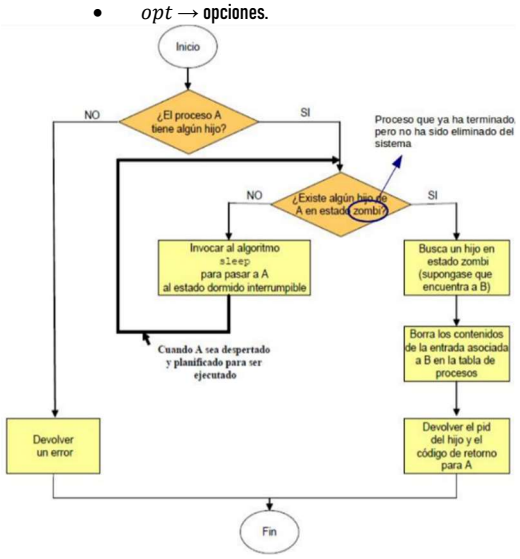
- Salida **normal** (voluntaria) → se produce cuando el proceso termina su trabajo. Se realiza mediante la función $exit(codigo)$.
- Salida por **error** (voluntaria) → se produce cuando el proceso encuentra un error que fue contemplado en su programación. Normalmente imprimirá un mensaje de error y realizará un $exit(estado)$.
- **Error fatal** (involuntaria) → se produce cuando el proceso encuentra un error fatal no contemplado en su programación (división entre 0, acceso no permitido a la memoria, etc.).
- **Eliminación por otro proceso** (involuntaria) → se produce cuando otro proceso le manda una señal que le indica al SO que lo elimine.
  - La llamada $kill(pid, senal)$ envía señales a otros procesos. Estas pueden indicar que se debe finalizar un proceso, pero tiene otros usos.

SYSCALL EXIT

- La syscall $exit(codigo)$ se usa para finalizar un proceso y proporcionarle un código entero de salida al padre.
  - No tiene ninguna salida.
  - $codigo$ → entero que el padre del proceso puede leer para actuar en función de su valor.
- Cuando se ejecuta, el proceso se desconecta del árbol de procesos, pero se mantiene en la lista de procesos como **ZOMBIE** hasta que finalice su padre.
  - La única información que se guarda de los procesos zombie en la lista de procesos es el hecho de que lo son y el valor de su $codigo$.
    - Esto sólo se almacena con objeto de que el padre use el valor de $codigo$, por lo que cuando el padre finaliza el zombie se elimina de la lista.
- $exits$ **implícitos** → si un proceso finaliza sin realizar un $exit$ el compilador lo incluirá de manera automática.

ESPERAR A LA TERMINACIÓN DE UN PROCESO

- La syscall $wait(\&ret)$ se usa para bloquear el proceso padre hasta que alguno de sus hijos (cualquiera) finaliza.
  - Su salida es $\begin{cases} \text{en caso de éxito} \rightarrow \text{pid del hijo.} \\ \text{en caso de fracaso} \rightarrow -1. \end{cases}$
  - ret → variable de retorno que se establecerá al estado de salida del hijo.
    - Para poder interpretarla tendremos que compararla con ciertas macros.
- La syscall $waitpid(pid, \&ret, opt)$ se usa para bloquear el proceso padre hasta que su hijo de pid pid termina.
  - Su salida es $\begin{cases} \text{en caso de éxito} \rightarrow \text{pid del hijo.} \\ \text{en caso de fracaso} \rightarrow -1. \end{cases}$
  - pid → pid del hijo por el que se va a esperar.
    - Si se pone -1 esperará por cualquier hijo, como si fuera un $wait$.
  - ret → lo mismo que en $wait$.
  - opt → opciones.




```
main()
{
     int pid, estado;

      if (fork()==0){
           printf("\nMensaje 1\n");
           sleep(4);
           exit(3); }
      else {
           pid=wait(&estado);
           printf("\nFinalizar); }
}
```

CAMBIO DE IMAGEN DE PROCESOS

- La syscall $execv(nombre, argv, entornp)$ se usa para cambiar la imagen (sección de código, pila y datos) del proceso actual.
  - Su salida es $\begin{cases} \text{en caso de éxito} \rightarrow \text{nada.} \\ \text{en caso de fracaso} \rightarrow -1. \end{cases}$
  - $nombre$ → nombre del archivo a ejecutar.
  - $argv$ → puntero al array de argumentos.
  - $entornonp$ → puntero al array de cadenas con la información de las variables de entorno.
- Sus argumentos están relacionados con los de $main$.

```
#define TRUE 1

while (TRUE) {                          /* se repite en forma indefinida */
        type_prompt();                  /* muestra el indicador de comando en la pantalla */
        read_command(command, parameters);      /* lee la entrada de la terminal */

        if (fork() !=0) {               /* usa fork para el proceso hijo */
                /* Codigo del padre. */
                waitpid(-1, &status, 0);        /* espera a que el hijo termine */
        } else {
                /* Codigo del hijo. */
                execve(command, parameters, 0);         /* ejecuta el comando */
```

SEÑALES

- Las SEÑALES son avisos que notifican a los procesos de los eventos que ocurren en el sistema.
  - Cada señal se identifica con un **entero** y/o con una **constante simbólica** (un mnemotécnico).
- Las señales se usan como **mecanismo de comunicación y sincronización** entre procesos.
  - En este aspecto son similares a las interrupciones.
    Sin embargo, las señales son lanzadas por el software y se puede definir qué rutina se ejecuta cuando se reciben.

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

SIGINT: Interrupción. Se envía cuando se pulsa CRTL+C en el teclado. Por defecto se interrumpe el programa

SIGCHLD: Terminación de algún proceso hijo. Se envía al proceso padre. Ignorada por defecto

GENERACIÓN DE SEÑALES

- **Excepciones** → se genera una señal como un aviso del SO (por ejemplo, para terminar un proceso que pretende dividir entre 0).
- **Otros procesos** → un proceso puede enviar una señal a otro con la syscall $kill$.
- **Interrupciones de la terminal** → el shell manda una señal a uno de sus hijos para finalizarlo ($CTRL + C$)
  - Pertenece al grupo anterior, ya que la terminal es un proceso.
- **Notificaciones de E/S** → siempre que no sean interrupciones generadas por las controladoras (hardware).
- **Alarmas** → llamadas que, cuando pase un número determinado de segundos, mandan una señal al proceso que los invocó.
- **Control de tareas, gestión de cuotas,** etc.

SYSCALL KILL

- La syscall $kill(pid, senal)$ se usa para enviar una señal de un proceso a otro.
  - Su salida es { en caso de éxito → 0. / en caso de fracaso→ -1.
  - pid → PID del proceso al que se le enviará la señal.
    - Puede ser cualquier proceso, no tiene por qué ser un hijo o padre del emisor.
  - $senal$ → entero/constante simbólica que identifica la señal a enviar.
- En función de la señal enviada, el proceso receptor tendrá que ejecutar una subrutina u otra.
- ▷ Desde la consola se puede invocar como $kill \ -senal \ \ pid$.

```
#include <signal.h>
main()
{
  int a;
  if ((a=fork())==0) {
    while(1) {
        printf("pid del proceso hijo = \%d\n", getpid());
        sleep(1); } }
  sleep(10);
  printf("Terminación del proceso con pid= \%d\n",a);
  kill(a,SIGTERM);
}
```

RECEPCIÓN DE SEÑALES

- La recepción de una señal lleva a una **acción predeterminada**. En función de qué señal sea, la acción podría finalizar el proceso, suspenderlo, reanudarlo, etc.
  - Esta acción ejecutará siempre el proceso receptor, por lo que tiene que estar en su **espacio de direcciones**.
  - Un proceso necesita ser **planificado** (es decir, estar en ejecución en ese momento) para realizar la acción correspondiente a la señal recibida.
- Se puede **modificar la captura de la señal**, es decir, qué subrutina se ejecuta cuando se recibe la señal, usando las syscalls $signal$ y $sigaction$.

SYSCALL SIGNAL

- La syscall $signal(senal, accion)$ se usa para definir qué manejador se ejecutará cuando el proceso reciba la señal $senal$, es decir, define la captura de la señal.
  - Su salida es { en caso de éxito → el manejador anterior. / en caso de fracaso→ SIG_ERR.
  - $senal$ → entero/constante simbólica que identifica la señal cuyo manejador se quiere especificar.
  - $accion$ → dirección del manejador que se asociará a la señal.
    - Si se pone SIG_DFL realizará la acción por defecto.
    - Si se pone SIG_IGN ignorará la señal.
- El manejador especificado sólo se utilizará para la señal **después de la invocación** a $signal$.
  Si se recibe la señal antes de la invocación, ejecutará la rutina por defecto.
  - Por esto, normalmente $signal$ se invocará al principio del programa.
- La **definición del manejador** debe tener un entero que represente a la señal recibida como argumento.
  - De esta manera se pueden especificar en una única función manejadores para distintas señales (aun así se debe invocar a $signal$ una vez por cada señal).
- ▷ Existen 2 señales que existen para que el usuario redefina sus manejadores, $SIGUSR1$ y $SIGUSR2$.
  - Sus manejadores por defecto provocan la finalización del proceso.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
void manejador(int sig);

main()
{
    if (signal(SIGUSR1,manejador)==SIG_ERR)
        exit(1);
    for(;;);
}

void manejador(int sig);
{
    printf("\n\n%s recibida. \n", strsignal(sig));
    exit(2);
```

SYSCALL SIGACTION

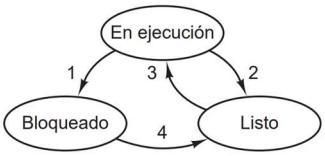
- La syscall $signaction(senal, accion, opt)$ funciona como $signal$, pero permite especificar otros parámetros y banderas de gestión de la señal.
  - Su salida es { en caso de éxito → 0. / en caso de fracaso→ SIG_ERR.
  - $senal$ → entero o constante simbólica que identifica la señal cuyo manejador se quiere especificar.
  - $accion$ → dirección del manejador que se asociará a la señal.
  - opt → puntero a una estructura $sigaction$ con las opciones de la gestión de la señal.
- Algunas de las **opciones** de la captura de la señal son:
  - Configuración de **máscaras de señales bloqueadas**.
  - **Desactivación del manejador** una vez recibida la señal → sólo se ejecuta el manejador especificado la primera vez que se recibe la señal. El resto de veces se ejecuta el predeterminado.
  - **Proporcionar información** sobre una señal y el proceso que la ha enviado.
  - Etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
main(void)
{
      int codigo_error=0;
      struct sigaction gestion;
      gestion.sa_handler = gestor;
      gestion.sa_mask = 0;
      gestion.sa_flags = SA_ONESHOT;
      codigo_error = sigaction  ( SIGINT, gestion, 0);
      if( codigo_error == SIG_ERR )
      {
            perror("Error al definir el gestor de SIGINT");
            exit(-1);
      }
      /** Código del programa ***/
      while(1);
}

void gestor( int señal )
{
     printf("Señal SIGINT recibida");
}
```

JERARQUÍAS DE PROCESOS

- En UNIX, un proceso y todos sus descendientes forman un **grupo**.
  - ↳ Los grupos se usan para poder **enviar señales** de manera simultánea **a todos** sus procesos, permitiendo que **cada uno** de ellos **trate** la señal de forma individual
- En UNIX, la **formación de la jerarquía** de procesos comienza con un proceso especial denominado $init$.
  - ↳ $init$ es el único proceso que nace sin padre y tiene el PID 0 o 1.
  1. $init$ se ejecuta nada más arranca el sistema, y crea un nuevo proceso para cada terminal.
  2. Los procesos de cada terminal esperan a que algún usuario inicie sesión.
  3. Cuando algún inicio de sesión tenga éxito, el proceso de inicio de sesión ejecuta un shell para aceptar comandos.
  4. A partir de estos comandos se crean cada vez más procesos en la sesión.
  - Por tanto, todos los procesos del sistema pertenecen al árbol de procesos cuya raíz es $init$.
- En Windows **no existe** la jerarquía de procesos.
  Lo único similar son los *tokens*, identificadores que reciben los padres al generar un hijo para poder gestionarlo.
  - ↳ Sin embargo, el padre puede pasar este *token* a otros procesos, lo cual invalida la jerarquía.

ESTADOS DE UN PROCESO



1. El proceso se bloquea para recibir entrada
2. El planificador selecciona otro proceso
3. El planificador selecciona este proceso
4. La entrada ya está disponible

- **Estados** en los que se puede encontrar un proceso:
  - **En ejecución**→ está avanzando, usando la CPU en este instante.
    - ↳ En un sistema con una CPU sólo 1 proceso puede estar en este estado. En un sistema con varias CPUs puede haber **tantos procesos en este estado como CPUs**.
  - **Listo** → es ejecutable, pero se detuvo temporalmente pues se planificó otro proceso para ocupar la CPU en este instante.
    - ↳ El planificador sólo tomará como candidatos a ejecutarse los procesos que se encuentren en este estado.
  - **Bloqueado** → no es ejecutable hasta que ocurra un cierto evento externo.
- **Transiciones** entre estados de un proceso:
  - De en ejecución a bloqueado (1) → el SO descubre que el proceso en ejecución no puede continuar en este instante (se inició una operación de E/S, se llamó a $pause$, etc.).
  - De en ejecución a listo (2) → el planificador decide cambiar el proceso en ejecución por otro, así que el primero pierde el acceso a la CPU.
  - De listo a en ejecución (3) → el planificador escoge el proceso para ocupar la CPU.
  - De bloqueado a listo (4) → se produce el evento externo por el que estaba esperando el proceso (la llegada de una entrada, la finalización de un $wait$, etc.).
  - × De bloqueado a en ejecución → **no existe** pues, si un proceso no está listo, el planificador no lo considerará como candidato, así que nunca decidirá ejecutarlo.
  - × De listo a bloqueado → **no existe** pues, si un proceso está detenido (pese a estar preparado para seguir ejecutándose), no podrá realizar ninguna acción que lo bloquee.

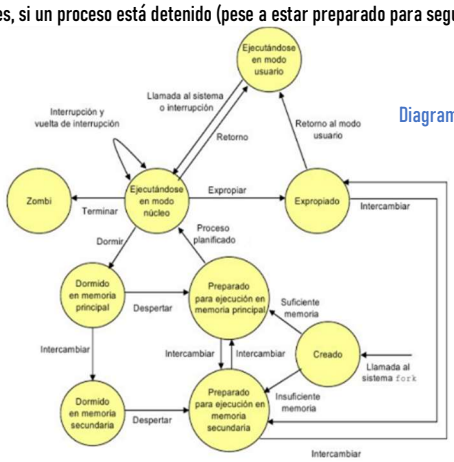


Diagrama más realista de estados de UNIX

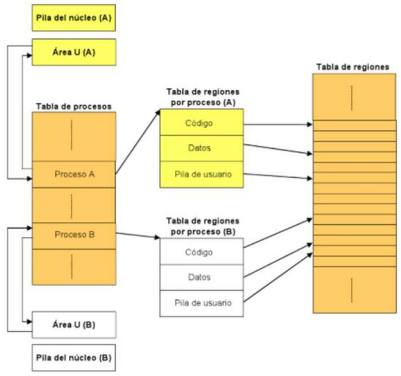
IMPLEMENTACIÓN DE PROCESOS

TABLA DE PROCESOS

- La TABLA DE PROCESOS (TPP) almacena información importante acerca del estado de los procesos activos o en espera del sistema.
  Es un **array de estructuras** con una entrada para cada proceso.
  - Guarda todos los datos necesarios para poder **reiniciar la ejecución** del proceso después de que se planifique otro.

  Algunos campos de las estructuras son (derecha):

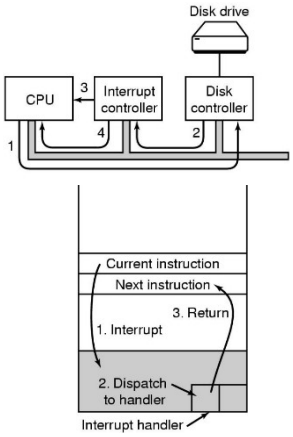
Administración de procesos	Administración de memoria	Administración de archivos
Registros	Apuntador a la información del segmento de texto	Directorio raíz
Contador del programa	Apuntador a la información del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Apuntador a la información del segmento de pila	Descripciones de archivos
Apuntador de la pila		ID de usuario
Estado del proceso		ID de grupo
Prioridad		
Parámetros de planificación		
ID del proceso		
Proceso padre		
Grupo de procesos		
Señales		
Tiempo de inicio del proceso		
Tiempo utilizado de la CPU		
Tiempo de la CPU utilizado por el hijo		
Hora de la siguiente alarma		



- **Pila del núcleo** → almacena funciones o rutinas invocadas durante la ejecución de un proceso en modo núcleo.
  - ↳ Si el proceso se estaba ejecutando en modo usuario estará vacía.
- **Área U** → almacena información de control necesaria para que el kernel gestione el proceso mientras se está ejecutando.
  - ↳ Contiene un puntero a la entrada del proceso en la tpp.
- **Tabla de regiones** → almacena una entrada para cada región (código, datos y pila de usuario) asignada a algún proceso.

INTRODUCCIÓN A LAS INTERRUPCIONES

- Cada clase (tipo) de E/S tiene una entrada en el VECTOR DE INTERRUPCIÓN del sistema, que contiene la dirección del procedimiento de interrupción para esa clase.
  - ↳ El procedimiento de interrupción de una clase de E/S sirve para manejar las interrupciones de dicha clase.



1. Se recibe una interrupción desde una controladora E/S (o desde un reloj).
2. El hardware guarda en la **pila del proceso en ejecución** actual ciertos registros como el PC, el PSW, etc.
3. El hardware carga como nuevo PC la dirección especificada en la entrada adecuada del vector de interrupción para ejecutar el procedimiento de interrupción.
   - 3.1. Rutina 1 (en ensamblador) ⟶ igual para todas las interrupciones de todas las clases.
     - 3.1.1. Guarda los registros en la entrada de la tpp del proceso en ejecución actual.
     - 3.1.2. Establece una nueva pila temporal, que usará el procedimiento de interrupción.
   - 3.2. Rutina 2 (en C) ⟶ específica para cada interrupción.
4. El planificador decide qué proceso va a ejecutar a continuación.
5. El control se devuelve al código ensamblador para cargar los registros y el mapa de memoria del proceso seleccionado.
6. Se ejecuta el proceso seleccionado.

HILOS

- Un HILO es una línea de ejecución dentro de un proceso, es decir, un "miniproceso".
  - En los SOs tradicionales cada proceso tiene su propio espacio de direcciones y un solo hilo de control.
- Sin embargo, puede ser conveniente tener **varios hilos** en un proceso de manera que **tengan el mismo espacio de direcciones** y **se ejecuten pseudoparalelamente**.
  - ↳ Por tanto, cada hilo sería como un proceso independiente, pero con el mismo espacio de direcciones.

UTILIDAD DE LOS HILOS

- Permiten desarrollar aplicaciones que requieren varias **actividades simultáneas** que colaboran entre sí.
  - ↳ El uso de **procesos** se justifica por el mismo motivo. Sin embargo, a diferencia de los procesos, los hilos comparten un **mismo espacio de direcciones**, lo cual es esencial para ciertas aplicaciones.
- Son mucho más **ligeros** que los procesos, así que se **crean y destruyen muy rápido.**
- La **conmutación** entre hilos de un mismo proceso es mucho más rápida que la conmutación entre procesos, pues no implica copiar toda su información.
- Son más útiles cuando **no están asociados** todos **a la misma CPU**:
  - Cuando se **reparten las tareas de cálculo y de E/S** entre varios hilos permiten que se **solapen**, lo cual agiliza la aplicación.
  - En los sistemas de **varias CPU** existe verdadero **paralelismo** entre hilos.

MODELO DE HILO

DIFERENCIAS ENTRE PROCESOS E HILOS

- El **modelo de procesos** se basa en:
  - La **agrupación de recursos relacionados** ⟶ los procesos son una manera de agrupar recursos relacionados (archivos abiertos, hijos, alarmas pendientes, etc.) para administrarlos con más facilidad.
  - La ejecución de **un solo hilo** ⟶ cada proceso tiene un único hilo de ejecución con su propio PC, registros y pila.
- El **modelo de hilo** agrega al modelo de proceso la posibilidad de llevar a cabo **varias ejecuciones en el mismo entorno** del proceso, de manera que son en gran parte **independientes** unas de otras.
- En conclusión, **los procesos compiten** por los recursos que comparten, mientras que **los hijos colaboran** usando los recursos que comparten.
  - ↳ Si un hijo trae una palabra de memoria principal a caché, esta palabra también será accesible desde este nivel más alto de la memoria para el resto de hijos. Sin embargo, si un proceso hace lo mismo, le habrá "quitado" un espacio de memoria de alto nivel a otro proceso, pues ninguno más puede acceder a esa palabra

EJECUCIÓN DE HILOS

- El MULTIHILAMIENTO consiste en permitir varios hilos para un mismo proceso.
- Cuando se ejecuta un proceso con multihilamiento en un sistema con una CPU, los hilos del proceso toman turnos para ejecutarse.
  Al **conmutar** de un hilo a otro, el sistema da una apariencia de **ejecución paralela de hilos** (igual que en la multiprogramación de procesos).

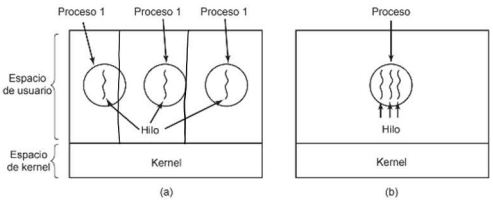


Figura 2-11. (a) Tres procesos, cada uno con un hilo. (b) Un proceso con tres hilos.

INDEPENDENCIA EN HILOS

- Los distintos hilos de un proceso **no son tan independientes como los procesos**.
  - Por tanto, no es posible que exista ningún tipo de **protección entre hilos**, pero tampoco es necesaria.
    - ↳ Los procesos sí necesitan esta protección, pues pueden pertenecer a usuarios diferentes, hostiles entre sí.
      Sin embargo, como todos los hilos de un proceso tienen el mismo propietario (el propietario del proceso) no tienen ese problema.

ELEMENTOS QUE COMPARTEN LOS HILOS DE UN PROCESO
- Espacio de direcciones.
- Variables globales.
- Archivos abiertos.
- Procesos hijos.
- PID.
- Alarmas pendientes.
- Señales y manejadores de señales.
- Información contable.

ELEMENTOS EXCLUSIVOS PARA CADA HILO DE UN PROCESO
- Registros.
- Contador de programa.
- PILA.
- Estado ⟶ igual que los procesos, un hilo puede estar bloqueado, listo o en ejecución.

MANEJO DE HILOS EN POSIX

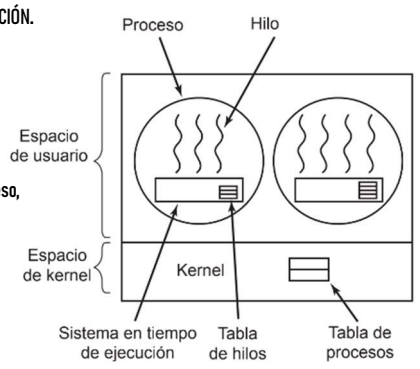
- Cuando existe multihilamiento en un proceso, este comienza con un único hilo.
  Este **hilo principal** puede de crear nuevos hilos mediante la función $pthread\_create(\&idHilo, opt, accion, \&arg)$:
  - Su salida es { en caso de éxito ⟶ 0. / en caso de fracaso ⟶ número de error (≠0).
  - $idHilo$ ⟶ puntero a variable de tipo $pthread\_t$ donde se almacenará el identificador del hilo creado.
  - opt ⟶ puntero a la estructura de los atributos del hilo ($NULL$ para usar los predeterminados).
  - $accion$ ⟶ función que ejecutará el hilo creado.
  - arg ⟶ es el único argumento que puede recibir la función que ejecutará el hilo.
    - ↳ Debe pasarse siempre por referencia y castearse a $void*$.
    - ↳ Si se crean hilos en un bucle y comparten variable de argumento, el comportamiento puede ser inesperado.

- La función $pthread\_exit(ret)$ finaliza el hilo y elimina su pila
  - $ret \rightarrow$ valor de retorno del hilo.
    - ↳ Puede ser leído por otro hilo en el mismo proceso que llame a $pthread\_join$.
  - Si alguno de los hilos ejecuta un $exit$ el proceso finaliza (incluyendo todos su s hilos activos).
- La función $pthread\_join(hilo, \&ret)$ se usa para bloquear el hilo actual hasta que finalice otro hilo del mismo proceso.
  - Su salida es $\begin{cases} \text{en caso de éxito} \rightarrow 0. \\ \text{en caso de fracaso} \rightarrow \text{número de error } (\neq 0). \end{cases}$
  - $hilo \rightarrow$ identificador del hilo por el que se espera
  - $retval \rightarrow$ variable de retorno del hilo.
- La función $pthread\_yield()$ se usa para que el hilo actual le ceda la CPU a otro hilo del proceso.
  - Su salida es $\begin{cases} \text{en caso de éxito} \rightarrow 0. \\ \text{en caso de fracaso} \rightarrow \text{número de error } (\neq 0). \end{cases}$
  - El hilo que cede la CPU pasa de estar en estado en ejecución a estado listo, es decir, no se bloquea y será planificable nada más se llame a la función.
  - No hay una llamada así para procesos porque se supone que ellos son competitivos, así que no desearán ceder la CPU.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS    10

void *print_hello_world(void *tid)
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread %d0, tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d0, i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d0, status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

IMPLEMENTACIÓN DE HILOS EN EL ESPACIO DE USUARIO

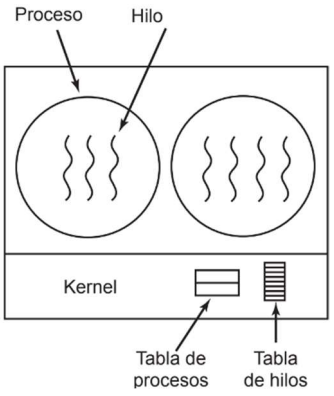
- El **kernel** no sabe nada acerca de los hilos. Por lo que le concierne, está administrando procesos ordinarios de un solo hilo.
- Las funciones de hilos se implementan mediante una **biblioteca de procedimientos**, conocida como SISTEMA EN TIEMPO DE EJECUCIÓN.
  - ↳ Esto permite que se puedan usar hilos hasta en sistemas que no los aceptan.
- Cada proceso necesita su propia TABLA DE HILOS, que almacenará la información exclusiva de cada uno de sus hilos.
  - Esta tabla es **privada**, así que ni el kernel ni ningún otro proceso pueden acceder a ella.
  - Administrada por el **sistema en tiempo de ejecución**.
- Por tanto, cuando un hilo de un proceso **se bloquea**, se bloquearán también todos los demás hilos de ese proceso.
  - ↳ Esto se debe a que el bloqueo es una syscall y cuando el kernel la recibe, no conoce nada sobre los hilos del proceso, así que bloquea el proceso en sí.
  - Para **evitar** que un hilo bloquee a todo un proceso se usa la ENVOLTURA, que es un procedimiento que se llama en tiempo de ejecución cuando un hilo pretende hacer algo que lo puede bloquear.
    1. El procedimiento comprueba si el hilo se va a bloquear.
    2. De ser así, cede la CPU a otro hilo.
    3. Se almacenan los registros del hilo actual en la tabla de hilos del proceso.
    4. Se busca en la tabla un hilo listo para ejecutarse (será un hilo **del mismo proceso**).
    5. Se cargan los registros con los valores del nuevo hilo.
    6. Se ejecuta el nuevo hilo.
    - ↳ Se ejecutarán hilos del mismo proceso hasta que el sistema le retira la CPU al proceso o ya no tiene más hilos ejecutables.
  - Realizar una **conmutación de hilos** de esta manera es **mucho más veloz** que la conmutación de procesos que se tendría que hacer si se bloquease el proceso entero.



- ✓ Las **funciones de manejo** de hilos y el **planificador** de hilos son procedimientos del espacio de usuario, por lo que invocarlos es mucho más rápido que invocar el trap al kernel que se usa en la implementación en el núcleo.
  - ↳ Un trap al kernel es más costoso pues implica realizar un cambio de contexto, vaciar la cache, etc.
- ✓ Permite definir **algoritmos de planificación de hilos personalizados** para cada proceso.
- ✓ Tiene una mejor **escalabilidad** que la del kernel, ya que la tabla de hilos central puede llegar a ocupar mucho espacio cuando hay muchos procesos con muchos hilos.
- × No se puede impedir que un **fallo de página** (que es un bloqueo impredecible) en un hilo bloquee a todo su proceso, pues no se le puede programar una envoltura.
- × Los hilos **no cederán la CPU automáticamente** nunca pues dentro de un mismo proceso no hay interrupciones de reloj.
  - ↳ Por tanto, cuando se comienza a ejecutar un hilo, no se podrá ejecutar ningún otro hasta que este ceda la CPU voluntariamente .

IMPLEMENTACIÓN DE HILOS EN EL NÚCLEO

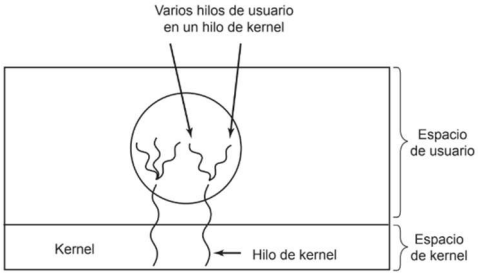
- El **kernel** sí conoce los hilos y es quien los administra.
- Las funciones de los hilos se implementan mediante **syscalls, no** se necesita un **sistema en tiempo de ejecución**.
- Hay una **única** TABLA DE HILOS ubicada en el **kernel**, que almacenará la información de todos los hilos de todos los procesos.
  - Administrada por el **kernel**.
- Todas las instrucciones que podrían **bloquear** a un hilo son syscalls.
  Así, cuando un hilo de un proceso se bloquea, el núcleo puede ejecutar otro hilo **del mismo proceso o de otro** sin problema.



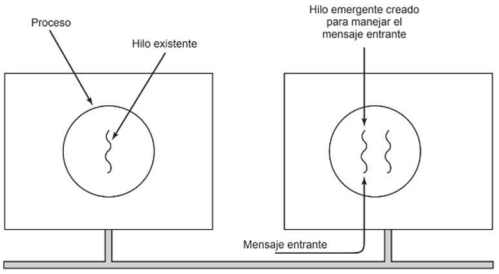
- ✓ **No** se necesita crear **envolturas**.
- ✓ Si un hilo se encuentra con un **fallo de página**, se puede ejecutar sin problema cualquier otro hilo del mismo proceso mientras se soluciona.
- ✓ Los hilos pueden **ceder la CPU automáticamente**.
- × Las **funciones de manejo** de hilos y el **planificador** son syscalls, así que su invocación será más lenta que en la implementación en espacio de usuario.
  - ↳ Un trap al kernel es más costoso pues implica realizar un cambio de contexto, vaciar la cache, etc.
- × Todos los hilos usarán el **mismo planificador**.
- × Tiene una mala **escalabilidad**, ya que la tabla de hilos central puede llegar a ocupar mucho espacio cuando hay muchos procesos con muchos hilos.

IMPLEMENTACIONES HÍBRIDAS

- Pretende aprovechar las ventajas de **ambas implementaciones**, pero sufre también de sus desventajas.
- Consiste en implementar un sistema de **hilos en el núcleo que multiplexan varios hilos de usuario.**
- ▷ El **SO** sólo es consciente de los **hilos de kernel**, que será los que planifique.
  Cuando se selecciona uno de estos hilos, se debe decidir cuál de los hilos de usuario asociados a él se ejecutará.
- ▷ El **programador** puede determinar cuántos hilos de kernel va a utilizar y cuántos hilos de nivel usuario va a multiplexar en cada uno de ellos.
  - ↳ Esta decisión afectará mucho al **rendimiento** de la aplicación.



HILOS EMERGENTES



▷ El método **tradicional** para manejar los mensajes entrantes de un **sistema distribuido** es hacer que un proceso o hilo, que está bloqueado en una syscall para recibir mensajes, espere al mensaje entrante.

○ Una alternativa es que la **llegada de un mensaje** haga que el sistema **cree un nuevo hilo**, un HILO EMERGENTE, para manejar el mensaje.

- Como este hilo es completamente nuevo, no tiene ninguna información que restaurar, así que **se crea muy rápido**.

HILOS EN LINUX

- Linux es un SO con **implementación de hilos en el núcleo**.

○ La syscall $clone(accion,\ pila,\ flags,\ arg)$ permite disolver la distinción entre hilos y procesos, creando un nivel de abstracción superior a ellos.
En función de las banderas pasadas, creará un hilo o un proceso, pero en principio no hay manera de saberlo.

– Su salida es { en caso de éxito → PID del hijo/TPID del hilo; en caso de fracaso → -1

- $accion$ → función que ejecutará el hilo/proceso creado.
- $pila$ → tamaño de la pila para el nuevo hilo/proceso.
- $flags$ → especifican qué se comparte y qué se mantiene en privado.
  ↳ También permiten especificar si el hilo se creará en el proceso actual o en uno nuevo.
- arg → único argumento de la función $accion$.

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PID	New thread gets old PID	New thread gets own PID
CLONE_PARENT	New thread has same parent as caller	New thread's parent is caller

PLANIFICACIÓN DE PROCESOS

○ El PLANIFICADOR DE PROCESOS es la parte del SO que, usando el ALGORITMO DE PLANIFICACIÓN, decide cuál de los procesos activos va a adquirir la CPU.

- El planificador sólo considera como candidatos los procesos que están en estado **listo**.
  ↳ Siempre habrá como mínimo un proceso en estado listo, el **proceso inactivo** (o idle).

- La **planificación es diferente en diferentes sistemas**. Por ejemplo:
  – En las PC hay pocos procesos activos, por lo que la planificación es poco importante.
  – En los servidores, hay muchos procesos activos, por lo que la planificación es crítica.
  - En cualquier caso, el principal objetivo del planificador es **buscar un uso eficiente de la CPU**.

CAMBIOS DE CONTEXTO

- El planificador debe escoger muy bien cuándo realizará un cambio de contexto, pues es una operación **muy cara computacionalmente**.
  1. Se pasa de modo usuario a modo núcleo.
  2. Se guarda el estado del proceso actual (incluyendo los registros) en la tabla de procesos y el mapa de memoria.
  3. Se selecciona un nuevo proceso a ejecutar mediante el algoritmo de planificación.
  4. Se cargan los datos del nuevo proceso.
  5. Se inicia el nuevo proceso.
  6. Probablemente sucedan muchos fallos de caché pues, desde que este nuevo proceso perdió en su momento la CPU hasta ahora, es posible que toda la información que usaba en la caché ya no esté en ella (pues los procesos que se ejecutaron entre tanto la fueron eliminando).
- Por tanto, si las conmutaciones de procesos son muy **frecuentes**, pueden llegar a consumir mucho tiempo de CPU, que podría haber sido invertido en avanzar procesos.

¿CUÁNDO SE PLANIFICAN PROCESOS?

El planificador se usa cuando:

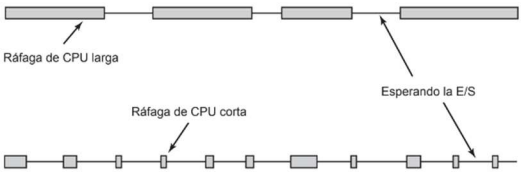
- Se **crea** un nuevo proceso → ¿se debe ejecutar el padre o el hijo?
- **Termina** un proceso → ¿qué proceso listo se ejecuta?
- Se **bloquea** un proceso (por una operación de E/S o un fallo de página) → ¿se tiene en cuenta la razón del bloqueo?
- Se recibe una **interrupción de E/S** → ¿se ejecuta el destinatario de la interrupción, el que se estaba ejecutando cuando se recibió u otro proceso?
- Se recibe una **interrupción de reloj** → ¿se cambia de proceso o se sigue ejecutando el mismo?

Hay dos **tipos** de algoritmos de planificación según como manejen las interrupciones de reloj:

○ Algoritmos **no apropiativos** → al recibir la interrupción no planifican.
Los procesos se ejecutan hasta que se bloquean o liberan la CPU voluntariamente.

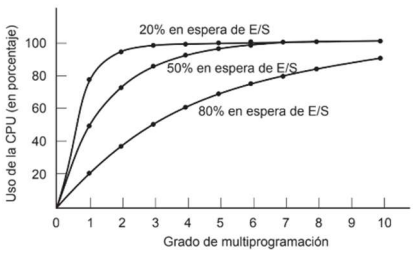
○ Algoritmos **apropiativos** → al recibir la interrupción de reloj se planifica otro proceso.
Los procesos sólo se pueden ejecutar durante un tiempo máximo (**quantum**).

PLANIFICACIÓN DE PROCESOS LIMITADOS A CÁLCULO Y A E/S



- Los procesos alternan **ráfagas de CPU** con **esperas de E/S** o de disco.
- Según la **longitud de sus ráfagas de cálculo** podemos distinguir dos tipos de procesos:
  – Procesos **limitados a cálculo** → invierten la mayor parte de su tiempo realizando cálculos, por lo que tienen ráfagas de CPU largas y esperas de E/S infrecuentes.
  – Procesos **limitados a E/S** → invierten la mayor parte de su tiempo esperando a la E/S, por lo que tienen ráfagas de CPU cortas y esperas de E/S muy frecuentes.
    ↳ A medida que las CPUs se hacen más rápidas, más procesos son de este tipo, pues, aunque las ráfagas de CPU sean cada vez más cortas, el número de esperas por E/S se mantiene.
    ▪ El factor clave es la longitud de la ráfaga de CPU y no la espera de E/S, pues los procesos limitados a E/S lo son debido a que no realizan muchos cálculos entre una petición y otra, no debido a que tengan esperas de E/S especialmente largas.

- La estrategia que adopta el planificador es darles **más prioridad a los procesos limitados a E/S**.
  ↳ Así, nada más un proceso limitado a E/S desee ejecutarse, lo hará y liberará la CPU poco tiempo después, permitiendo que los procesos limitados a cálculo (que normalmente estarán ya listos) puedan seguir avanzando.
  ↳ Si se le diera prioridad a los procesos limitados a cálculo, como casi siempre estarán listos y realizan muy pocas esperas, estancarían los procesos limitados a E/S.
- Con esta estrategia se pretende **desperdiciar poco tiempo de CPU**.
  ↳ Si tratamos con procesos limitados a E/S que usan un 80% de su tiempo en esperas, necesitaríamos tener 10 procesos activos para que el desperdicio de la CPU sea menor que el 10%.
  ↳ Sin embargo, si tratamos con procesos limitados a cálculo que sólo usan un 20% de su tiempo en esperas, solamente necesitaríamos 2 procesos activos para que el desperdicio de la CPU sea menor que el 10%.



METAS DE LOS ALGORITMOS DE PLANIFICACIÓN

El planificador debe cumplir las siguientes **condiciones** dependiendo del tipo de sistema:

- En **todos** los sistemas:
  - **Equidad** → los procesos comparables deben recibir un servicio comparable.
  - **Aplicación de políticas del sistema** → por ejemplo, darle más prioridad a un tipo de procesos que a otro.
  - **Balance** → se debe intentar mantener ocupadas todas las partes del sistema cuando sea posible (tanto la CPU como los dispositivos de E/S).
- En los sistemas de **procesamiento por lotes**:
  - **Maximizar el rendimiento** → el rendimiento es el número de trabajos por hora que completa el sistema.
  - **Minimizar el tiempo de retorno** → el tiempo de retorno es la media aritmética de los tiempos de respuesta de todos los procesos.
  - **Utilización de la CPU** → se debe intentar mantener ocupada a la CPU todo el tiempo.
- En los sistemas con **usuarios interactivos**:
  - **Minimizar el tiempo de respuesta** → el tiempo de respuesta es el tiempo que transcurre entre emitir un comando y obtener su respuesta.
  - **Proporcionalidad** → se debe intentar cumplir las expectativas de los usuarios.
- En los sistemas de **tiempo real**:
  - **Cumplir con los plazos** → los plazos son el límite de tiempo que tiene una tarea para finalizar.
  - **Predictibilidad** → se debe evitar la degradación repentina de calidad en los sistemas multimedia.

CATEGORÍAS DE LOS ALGORITMOS DE PLANIFICACIÓN

No hay un algoritmo de planificación óptimo para todos los sistemas posibles. Sistemas con **distintos objetivos** necesitan **distintos algoritmos** de planificación:

- Sistemas de **procesamiento por lotes** → algoritmos de planificación **no apropiativos** o **apropiativos con largo periodo**.
  - ↳ Se aprovecha que no hay usuarios que esperen para obtener una respuesta rápida para usar estos algoritmos porque permiten reducir la conmutación entre procesos, aumentando el rendimiento.
- Sistemas con **usuarios interactivos** → algoritmos de planificación **apropiativos**.
  - ↳ Se desea evitar que un proceso acapare la CPU y le niegue el servicio a los demás.
- Sistemas de **tiempo real** → depende de las exigencias del sistema.
  - ↳ La apropiación a veces no es necesaria porque los procesos ya saben que no se pueden ejecutar durante mucho tiempo.

PLANIFICACIÓN EN SISTEMAS DE PROCESAMIENTO POR LOTE

Hay distintas variantes de algoritmos de planificación para estos sistemas: **FCFS (no apropiativo)**, **SJF (no apropiativo)** y **SRTN (apropiativo)**.

PRIMERO EN ENTRAR PRIMERO EN SALIR (no apropiativo)

- Su funcionamiento se basa en una **cola de procesos listos**.
  1. Mientras se ejecuta un proceso, todos los que van llegando se introducen al final de la cola.
  2. Cuando se bloquea el proceso en ejecución, **se escogerá siempre como proceso a ejecutar el primero de la cola**.
  3. Cuando el proceso que se había bloqueado pase a estar listo otra vez se colocará al final de la cola como todos los demás.

✓ Es muy **sencillo** y **fácil de implementar** mediante una **lista enlazada** de procesos listos.

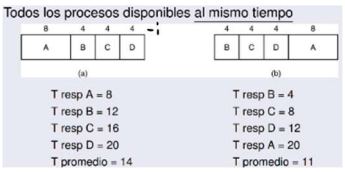
× Si se ejecutan **muchos procesos limitados a E/S**, cada vez que un proceso limitado a CPU se bloquee (por ejemplo, por un fallo de página), tendrá que esperar a que terminen de ejecutarse todos los de E/S antes de poder continuar.

TRABAJO MÁS CORTO PRIMERO (no apropiativo)

- Su funcionamiento se basa en **ejecutar primero el proceso con menor tiempo de ejecución**.
  - ↳ El tiempo de ejecución del primer trabajo ejecutado repercutirá en el tiempo de respuesta de todos los demás, por lo que debe ser el más veloz.
- Naturalmente, se deben **conocer de antemano** los **tiempos de ejecución** de todos los procesos.
  - ↳ Normalmente se consigue pidiéndole al usuario que lanza el proceso una estimación de cuánto tardará.

✓ **Aumenta el rendimiento** del sistema pues realiza muchos procesos cortos.

× **Sólo es óptimo** en términos del tiempo de retorno si todos los trabajos están **disponibles al mismo tiempo**.



Procesos disponibles a destiempo:

Cinco trabajos (A, B, C, D, E) con tiempos de ejecución (2, 4, 1, 1, 1) y tiempos de llegada (0, 0, 3, 3, 3)
Orden de ejecución: A, B, C, D, E. El tiempo de respuesta promedio 4.6
Si el orden fuese B, C, D, E, A, el tiempo de respuesta promedio sería 4.4

TRABAJO DE MENOR TIEMPO RESTANTE A CONTINUACIÓN (apropiativo)

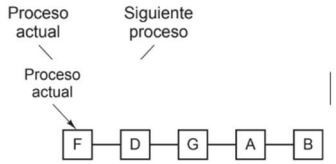
- Su funcionamiento se basa en **seleccionar el proceso con menor tiempo de ejecución restante**.
- Cuando llega un nuevo proceso:
  1. El tiempo total del nuevo proceso se compara con el tiempo restante del que se estaba ejecutando.
  2. Si necesita menos tiempo, se suspende el proceso actual y se guarda su tiempo restante.
  3. Se inicia el nuevo proceso.
- Naturalmente, se deben **conocer de antemano** los **tiempos de ejecución** de todos los procesos.
  - ↳ Normalmente se consigue pidiéndole al usuario que lanza el proceso una estimación de cuánto tardará.

✓ Da muy **buen servicio** a los procesos cortos.

PLANIFICACIÓN EN SISTEMAS CON USUARIOS INTERACTIVOS

Hay distintas variantes de algoritmos de planificación para estos sistemas: **por turno circular (apropiativo), por prioridad (apropiativo)** y otros (también **apropiativos**).

PLANIFICACIÓN POR TURNO CIRCULAR – round robin (apropiativo)



- Su funcionamiento se basa en **asignar un quantum a cada proceso y ejecutarlos secuencialmente**.
- Cuando el proceso actual pierde la CPU se escogerá como nuevo proceso al **primero** de la lista, y el actual se pondrá al **final**.

✓ Es muy **fácil de implementar** mediante una **lista enlazada** de procesos listos.
✗ Supone que todos los procesos tienen igual **importancia**.

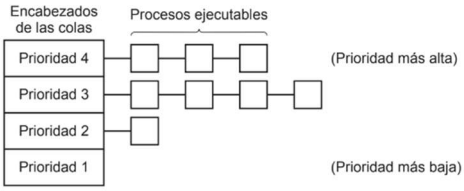
DETERMINAR LA LONGITUD DEL QUANTUM

- Como vimos antes, la **conmutación de procesos** lleva un tiempo que colabora a la **sobrecarga administrativa** (overhead), que es el tiempo de CPU gastado en ejecutar código que no es de procesos.
- Por tanto, hay que escoger muy bien la longitud del quantum:
  - Si se escoge un quantum demasiado **corto** $\begin{cases}\text{se realizan demasiados cambios de contexto} \rightarrow \text{el overhead es una proporción grande del tiempo de CPU}\\ \text{los procesos tendrán que esperar menos para poder ejecutarse}\end{cases}$.
  - Si se escoge un quantum demasiado **largo** $\begin{cases}\text{se realizan menos cambios de contexto "innecesarios" pues muchos procesos no agotan sus quantums}\\ \text{si hay muchos procesos los tiempos de espera serán muy largos}\end{cases}$.

▸ Los quantums entre 20ms y 50ms son apropiados.

PLANIFICACIÓN POR PRIORIDAD (apropiativo)

- Su funcionamiento se basa en **asignar una prioridad a cada proceso** y **ejecutar aquel proceso listo con prioridad más alta**.
- Se debe **evitar** que los procesos con alta prioridad **acaparen la CPU**. Hay 2 métodos para conseguir esto:
  - Envejecimiento → cada vez que el proceso en ejecución agote un quantum, se reduce su prioridad y si otro proceso tiene alguna más alta, se conmuta.
  - Cuotas de CPU → cada vez que el proceso en ejecución agote un quantum, se cambia al siguiente proceso con más prioridad.
- La **asignación de prioridad** a los procesos se puede hacer de dos formas:
  - Estática → se asigna la prioridad del proceso cuando este comienza y se mantiene toda su ejecución.
  - Dinámica → la prioridad del proceso puede cambiar durante su ejecución.
    - Por ejemplo, se le asigna a cada proceso una prioridad de $\mathbf{1/f}$ donde f es la fracción del último quantum usada por el proceso. Así, nos aseguramos de darle **más prioridad a los procesos limitados a E/S**, como habíamos explicado antes.
- Se **agrupan** los procesos en **colas de prioridad** y se usa $\begin{cases}\text{entre clases} \rightarrow \text{planificación por prioridad.}\\ \text{dentro de cada clase} \rightarrow \text{planificación round robin.}\end{cases}$



1. Mientras haya procesos en la clase de mayor prioridad, se ejecutarán estos en round robin.
2. Si la clase de mayor prioridad está vacía, se ejecutarán los de la siguiente, y así sucesivamente.

- Para asegurarse de que incluso los procesos de prioridad más baja consiguen ejecutarse, se deben **ajustar las clases dinámicamente**.

OTROS ALGORITMOS DE PLANIFICACIÓN

PROCESO MÁS CORTO A CONTINUACIÓN

▸ Es una adaptación del algoritmo de trabajo más corto primero de los sistemas de procesamiento por lotes.

- Por tanto, su funcionamiento se basa en **seleccionar el proceso con menor tiempo de ejecución**.
- Se deben **conocer de antemano estimaciones del tiempo de ejecución** de todos los procesos.
  - ↳ La primera estimación se consigue normalmente pidiéndole al usuario que lanza el proceso una estimación de cuánto tardará.

$$T = a * T_{estimación\ anterior} + (1 - a) * T_{tiempo\ última\ ejecución}$$

  - El parámetro de envejecimiento, a $(0 \leq a \leq 1)$, nos permite decidir si le damos más importancia a $\begin{cases}\text{las ejecuciones anteriores} \rightarrow a \text{ cercano a 1}\\ \text{la última ejecución} \rightarrow a \text{ cercano a 0}\end{cases}$.

PLANIFICACIÓN GARANTIZADA

- Su funcionamiento se basa en que, con n **usuarios** (o **procesos**), cada uno recibe un $\mathbf{1/n}$ de tiempo de la CPU.

PLANIFICACIÓN POR SORTEO

- Su funcionamiento se basa en **darle boletos a cada proceso** y **planificar seleccionando un boleto al azar**, para que el proceso que lo tenga obtenga la CPU.
- La **prioridad** se implementa dándole más boletos a los procesos más importantes.
- Un proceso le puede **dar sus boletos a otro** mientras espera a que termine para que tenga más prioridad y así acabe más rápido.

PLANIFICACIÓN POR PARTES EQUITATIVAS

- Su funcionamiento se basa en **asignar una fracción de la CPU a cada usuario en función del número de procesos que tenga.**
  - ↳ Así se evita que un usuario con muchos procesos consiga una cantidad injustamente grande de tiempo de CPU.

PLANIFICACIÓN DE HILOS

La planificación en sistemas que permiten multihilamiento será muy distinta en función de si los hilos están implementados a nivel de usuario o a nivel de núcleo.

– En los hilos a nivel de usuario:
  - El **kernel planifica procesos**, pues no es consciente de la existencia de hilos.
  - Cuando se planifica un proceso, el **planificador de hilos del sistema en ejecución** de ese proceso **selecciona un hilo** en concreto. Mientras dure el quantum de ese proceso, seguirá seleccionando sus hilos.
  - Permite crear **planificadores específicos** adaptados a las características de una aplicación concreta.
– En los hilos a nivel de núcleo:
  - El **kernel planifica hilos**, en principio sin tener en cuenta de qué proceso son (aunque puede hacerlo).
    - ↳ Puede darle más importancia a planificar hilos del proceso que se está ejecutando actualmente pues conmutar a un hilo de otro proceso sería más caro.
  - Cuando un hilo se **bloquea**, no se suspende todo el proceso.