Procesos e Hilos

JOAQUÍN ROIZ PAGADOR[1]

[1] *Universidad Pablo de Olavide, Ctra. Utrera 1, 41013, Dos Hermanas, Sevilla, España.*
[*] *Contacto: quiniroiz@gmail.com*

Compiled May 21, 2018

Resumen del temario de la asignatura de Sistemas Operativos del Grado en Ingeniería Informática en Sistemas de Información. © 2018 Joaquín Roiz Pagador

OCIS codes: (130.6750) Systems.

quiniroiz@gmail.com

1. PROCESOS E HILOS

A. Procesos

Consideremos primero un servidor Web, a donde convergen las peticiones de páginas Web provenientes de todos lados. Cuando llega una petición, el servidor verifica si la página que se necesita está en la caché. De ser así, devuelve la página; en caso contrario, inicia una petición al disco para obtenerla y desde la perspectiva de la CPU, estas peticiones tardan eternidades.

En cualquier sistema de multiprogramación, la CPU conmuta de un proceso a otro con rapidez, ejecutando cada uno durante décimas o centésimas de milisegundos: hablando en sentido estricto, en cualquier instante la CPU está ejecutando sólo un proceso.

A.1. El modelo del Proceso

En este modelo, todo el software ejecutable en la computadora se organiza en varios procesos secuenciales (procesos, para abreviar). Un proceso es una instancia de un programa en ejecución, incluyendo los valores actuales del contador de programa, los registros y las variables. Cada proceso tiene su propia CPU virtual; en la realidad, la CPU real conmuta de un proceso a otro, pero para entender el sistema es mucho más fácil pensar en una colección de procesos que se ejecutan en paralelo, en vez de tratar de llevar la cuenta de cómo la CPU conmuta de programa en programa.

Proceso

- Podemos decir que un proceso es un programa en ejecución.
- El programa debe tener asignado algún recurso.
- Procesos variados pueden pertenecer al mismo programa.
- Un proceso no puede acceder al espacio de memoria de otro proceso.

Partes de un Proceso

- Program code → text section.
- Current Activity including program counter, processor registers.
- Stack containing temporaty data.
- Function parameters, return addresses, local variables.
- Data section → global variables.
- Heap → memory dynamically allow during runtime.

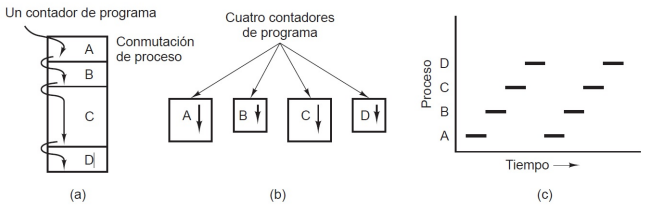


Fig. 1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptural de cuatro procesos secuenciales independientes. (c) Sólo hay un programa activo a la vez.

De ahora en adelante supondremos que sólo hay una CPU. Dado que la CPU conmuta rápidamente entre un proceso y otro, la velocidad a la que un proceso ejecuta sus cálculos no es uniforme y tal vez ni siquiera sea reproducible si se ejecutan los mismos procesos de nuevo. Por ende, al programar los procesos debe asumirse esta variación de velocidad.

La diferencia entre un proceso y un programa es sutil pero

crucial.

proceso es una actividad de cierto tipo: tiene un programa, una entrada, una salida y un estado Vale la pena recalcar que sin programa se está ejecutando por duplicado cuenta con dos procesos.

Programa: Entidad pasiva. Se almacenan en HDD.

A.2. Creación de un Proceso

En sistemas muy simples (sólo una aplicación) es posible tener presentes todos los procesos que se vayan a requerir cuando el sistema inicie. No obstante, en los sistemas de propósito general se necesita cierta forma de crear y terminar procesos según sea necesario durante la operación.

Hay cuatro eventos principales que provocan la creación de procesos:

1. El arranque del sistema.
2. La ejecución, desde un proceso, de la llamada al sistema para creación de procesos.
3. Una petición de usuario para crear un proceso.
4. El inicio de un trabajo por lotes.

A.3. Terminación de Procesos

Tarde o temprano el nuevo proceso terminará por alguna de las siguientes condiciones:

1. Salida normal (voluntaria).
2. Salida por error (voluntaria).
3. Error fatal (involuntaria).
4. Eliminado por otro proceso (involuntaria).

La mayoría de los procesos terminan debido a que han concluido su trabajo.

A.4. Jerarquía de Procesos

En algunos sistemas, cuando un proceso crea otro, el proceso padre y el proceso hijo continúan asociados en ciertas formas.

UNIX se inicializa a sí mismo cuando se enciende la computadora. Hay un proceso especial (init) en la imagen de inicio. Cuando empieza ejecutarse, lee un archivo que le indica cuántas terminales hay. Después utiliza *fork* para crear un proceso por cada terminal. Estos procesos esperan a que alguien inicie la sesión. Si un inicio de sesión tiene éxito, el proceso de inicio de sesión ejecuta un shell para aceptar comandos. Éstos pueden iniciar más procesos y así sucesivamente. Por ende, todos los procesos en el sistema completo pertenecen a un solo árbol, con *init* en la raíz.

En contraste, Windows no tiene un concepto de una jerarquía de procesos.

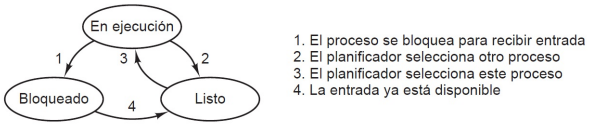


Fig. 2. Un proceso puede encontrarse en estado ”en ejecución”, ”bloqueado” o ”listo”. Las transiciones entre estos estados son como se muestran.

A.5. Estados de un Proceso

Un proceso puede generar cierta salida que otro proceso utiliza como entrada. Los tres estados en los que se puede encontrar un proceso son:

1. En ejecución(en realidad está usando la CPU en ese instante).
2. Listo (ejecutable; se detuvo temporalmente para dejar que se ejecute otro proceso).
3. Bloqueado (no puede ejecutarse sino hasta que ocurra cierto evento externo).

En sentido lógico, los primeros dos estados son similares. En ambos casos el proceso está deseoso de ejecutarse; sólo en el segundo hay temporalmente una CPU para élo. El tercer estado es distinto de los primeros dos en cuanto a que el proceso no se puede ejecutar, incluso aunque la CPU no tenga nada que hacer.

A.6. Implementación de los procesos

Para implementar el modelo de procesos, el sistema operativo mantiene una tabla llamada **tabla de procesos**, con sólo una entrada por cada proceso (algunos autores llaman a estas entradas bloques de control de procesos). Esta tabla entrada contiene información como el estado del proceso, contador de programa, apuntador de pila, asignación de memoria, estado de sus archivos abiertos, información de contabilidad y planificación y todo aquello que debe guardarse acerca del proceso cuando éste cambia de estado en ejecución a listo o bloqueado, de manera que se pueda reiniciar posteriormente como si nunca se hubiera detenido.

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		

Fig. 3. Algunos de los campos de una entrada típica en la tabla de procesos

Ahora que hemos analizado la tabla de procesos, es posible explicar un poco más acerca de cómo la ilusión de varios procesos secuenciales se mantiene en una (o en varias) CPU.

Con cada clase de E/S hay una ubicación asociada, a la cual se le llama vector de interrupción. Esta ubicación contiene la dirección del procedimiento del servicio de interrupciones.

1. El hardware mete el contador del programa a la pila, etc.
2. El hardware carga el nuevo contador de programa del vector de interrupciones.
3. Procedimiento en lenguaje ensamblador guarda los registros.
4. Procedimiento en lenguaje ensamblador establece la nueva apila.
5. El servicio de interrupciones de C se ejecuta (por lo general lee y guarda la entrada en el búfer).
6. El planificador decide qué proceso se va a ejecutar a continuación.
7. Procedimiento en C regresa al código de ensamblador.
8. Procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

Fig. 4. Esqueleto de lo que hace el nivel más bajo del sistema operativo cuando ocurre una interrupción.

Procesos en c.

1. **Fork**: Creación de proceso hijo.
   Necesidad de incluir librería **unistd.h**. Estructura: *int fork(void);*. Fork() crea un nuevo proceso, con exactamente el mismo código que invoca la función. El proceso continuará su ejecución después de la llamada a fork. El valor devuelto puede ser:
   - -1 si no se ha creado.
   - 0 si es el proceso hijo
   - distinto si es el padre.

```
switch (fork())
{
case -1:
/* Error code */
...
break;
case 0:
/* Child process code */
...
break;
}
```

2. **Wait()**: Esta función acepta un parámetro de tipo entero, que será usado para guardar el valor de salida del proceso hijo. La función *wait()* bloqueará al proceso padre **hasta que uno de los procesos hijos termine** su ejecución.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
        int childState;
        int pid = getpid();
        int pid_son;
        int pid_wait;
        pid_son = fork();
        if (pid_son == -1) {
                printf("Error, child has
                    not been created\n");
                exit(-1);
        }
        if (pid_son == 0){
                printf("Child process, pid
                    %d\n",getpid());
                exit(0);
        }
        else {
                pid_wait=wait(&childState)
                    ;
                printf("Father (PID: %d),
                    child with PID: %d
                    terminated with exit
                    code %d\n",pid,pid_son,
                    childState);
                printf("Pid returned by
                    wait is %d",pid_wait);
        }
}
```

3. **Macros**: Información extra que puede ser interpretada utilizando los siguientes comandos, definidos en <*sys/wait.h*> y evaluan las expresiones integrales.
   - **WIFEXITED(statVal)**: Evalúa un valor no cero. Si el estado devuelto por un proceso hijo indica que ha terminado normalmente.
   - **WEXITSTATUS(statVal)**: si el valor de *WIFEXITED(statVal)* es distinto a cero, este comando evalúa si el valor del proceso del hijo ha devuelto desde el main.
   - **WIFISIGNALED(statVal)**: Evalúa un valor distinto a cero. Si el estado devuelto por un hijo que terminó debido a la llegada de una se´nal que no fue capturada.
   - **WTERMSIG(statVal)**: Si el valor de *WIFSIGNALED(statVal)* es no cero, este comando evalúa la causa de finalización del proceso hijo.
   - **WIFSTOPPED(statVal)**: Evalúa un valor distinto a cero si el estado devuelto por un proceso hijo se acaba de parar.
   - **WSTOPSIG(statVal)**: Si el valor de *WIFSTOPPED(statVal)* es distinto a cero, esta orden evalúa el número de código que causó la parada del proceso hijo.
   - **WIFCONTINUED(statVal)**: Evalúa un valor cero si el estado devuelto por un hijo que ha continuado de un trabajo de control para.

```
if ( WIFEXITED(childState) != 0)
//child terminated normally with an exit
{
printf ("My child executed an exit(%d)\n",
WEXITSTATUS(childState)); }
```

4. **Wait pid**: Utilizar en lugar de wait() si queremos esperar hasta que un proceso hijo en particular termine.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
  int main(void)
  {
     int i, status;
     pid_t childID, endID;
     time_t when;
     if ((childID = fork()) == -1) {    /*
         Start a child process.      */
        exit(EXIT_FAILURE);
     }
     else if (childID == 0) {            /*
         This is the child.          */
        time(&when);
        sleep(10);
            /* Sleep for 10 seconds.     */
        exit(EXIT_SUCCESS);
     }
     else {
         /* This is the parent.       */
        /* Wait 15 seconds for child
            process to terminate.
            */
        for(i = 0; i < 15; i++) {
           endID = waitpid(childID, &
               status, WNOHANG|WUNTRACED);
           if (endID == -1) {
               /* error calling waitpid
                   */
              exit(EXIT_FAILURE);
           }
           else if (endID == 0) {
               /* child still running
                     */
              sleep(1);
           }
           else if (endID == childID) {
               /* child ended
                           */
              if (WIFEXITED(status))
                 printf("Child ended
                     normally \n");
              else if (WIFSIGNALED(status)
                  )
                 printf("Child ended
                     because of an uncaught
                     signal \n");
              else if (WIFSTOPPED(status))
                 printf("Child process has
                     stopped \n");
              exit(EXIT_SUCCESS);
           }
        }
     }
  }
```

Procesos en ejecución.

```
SYNOPSIS
#include<unistd.h>
int execl(const char *path, const char *arg
    ,....);
DESCRIPTION
execl()
replaces the current process image with a new
    process image
PARAMETERS
path: absolute route of the command to be run
arg: list of arguments to be passed to the
    program
RETURNED VALUES
-1 in case of errors
```

La función **execv** es utilizada para ejecutar procesos previamente compilados en una ruta específica.

B. Hilos

En los S.O. tradicionales, cada proceso tiene un espacio de direcciones y un solo hilo de control. De hecho, ésa es casi la definición de un proceso. Sin embargo, con frecuencia hay situaciones en las que es conveniente tener varios hilos de control en el mismo espacio de direcciones que se ejecuta en cuasi-paralelo, como di fueran procesos (casi) separados).

Hilo → Unidad básica de uso de CPU.

- Tiene ID, contador de registro, conjunto de registro y pila.
- Comparte con otros hilos del mismo proceso.
- Si el proceso tiene múltiple control de hilo, puede ejecutar *más de una petición* a la vez.

B.1. El modelo clásico de Hilo

El término **multihilamiento** también se utiliza para describir la situación de permitir varios hilos en el mismo proceso. Algunas CPUs tienen soporte directo en el hardware para el **multihilamiento** y permiten que las conmutaciones de hilos ocurran en una escala de tiempo en nanosegundos.

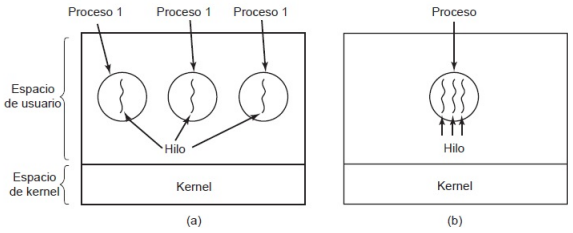


Fig. 5. (a) Tes procesos, cada uno con un hilo. (b) Un proceso con tres hilos.

Cuando se ejecuta un proceso con **multihilamiento** en un sistema con una CPU, los hilos toman turnos para ejecutarse.

B.2. Implementación de Hilos en el espacio de usuario.

Cada proceso necesita su propia **tabla de hilos** privada para llevar la cuenta de los hilos en ese proceso. Esta tabla es similar a la tabla d eprocesos del kernel, excepto porque sólo lleva la cuenta de las propiedades por cada hilo, con el contador de

programa, apuntador de pila, registros, estados, etc. Cuando un hilo pasa al estado listo o bloqueado, la información necesaria para reiniciarlo se almacena en la tabla de hilos.

Cuando un hilo hace algo que puede ponerlo en estado bloqueado en forma local llama a un procedimiento del sistema en tiempo de ejecución. Este procedimiento comprueba si el hilo debe ponerse en estado bloqueado.

- **Thread Creation**

```
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*start_routine)
                       (void *),
                   void * arg)
```

- **Thread Exit**

```
void pthread_exit(void *retval)
```

- **Wait for a Thread**

```
int pthread_join(pthread_t thread,
void **thread_return)
```

- Giving the CPU up

```
void pthread_yield()
```

C. Comunicación entre Procesos

Existe una necesidad de comunicación entre procesos o IPC. Las tres cuestiones son:

- La primera es cómo pasar información a otro.
- La segunda es que dos o más procesos no se interpongan entre sí.
- La tercera trata acerca de obtener la secuencia apropiada cuando hay dependencias presentes.

C.1. Condiciones de Carrera

En algunos sistemas operativos, los procesos que trabajan en conjunto pueden compartir cierto espacio de almacenamiento en el que pueden leer y escribir datos. El almacenamiento compartido puede estar en la memoria principal, o puede ser un archivo compartido; la ubicación de la memoria compartida no cambia la naturaleza de la comunicación o los problemas que surgen. Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un **directorio de spooler** especial. Otro proceso, el **demonio de impresión**, comprueba en forma periódica si hay archivos que deban imprimirse y si lo hay, los imprime y luego elimina sus nombres del directorio.

Las situaciones en donde dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y exactamente cuándo lo hace, se conocen como **condiciones de carrera**.

C.2. Regiones Críticas

¿Cómo evitamos las condiciones de carrera? La clave es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo. La dificultad ocurrió debido a que el proceso B empezó a utilizar una de las variables compartidas antes de que el proceso A terminara con ella.

El problema se puede formular de una manera abstracta. Parte del tiempo, un proceso está ocupado realizando cálculos internos y otras cosas que no producen condiciones de carrera. Algunas veces un proceso tiene que acceder a la memoria compartida o archivos compartidos, o hacer otras cosas críticas que pueden producir carreras. Esa parte del programa en la que se accede a la memoria compartida se conoce como **región crítica** o **sección crítica**. Requeriríamos algún tipo de orden.

El recurso compartirdo define la región crítica.
Aunque evita las condiciones de carrera, no es suficiente. Necesitamos cumplir con cuatro condiciones para una buena solución:

1. No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.
2. No pueden hacerse suposiciones acerca de las velocidades o el úmero de CPUs.
3. Ningún proceso que se ejecute fuera de su región cr ítica puede bloquear otros procesos.
4. Ninguún proceso tiene que esperar para siempre para entrar a su región crítica.

Aquí el proceso A entra a su región crítica en el tiempo T_1. Un poco después, en el tiempo T_2 el proceso B intenta entrar a su región crítiica, pero falla debido a que otro proceso a se encuentra en su regi ón crítica y sólo se permite uno a la vez. En consecuencia, B se suspende temporalemnte hasta el tiempo T_3 cuando A sale de su región crítica, con lo cual se permite a B entrar de inmediato. En algún momento dado B sale (en T_4) y regresamos a la situación original, sin procesos en sus regiones críticas.

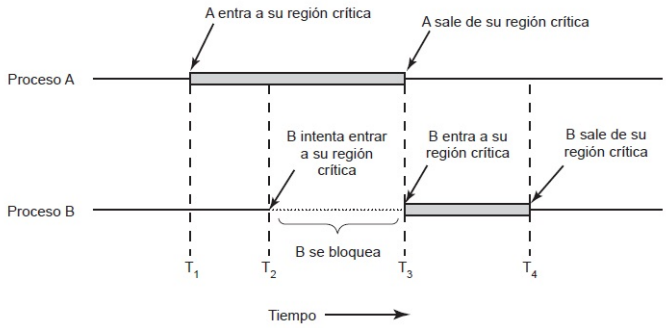


Fig. 6. Exclusión mutua mediante el uso de regiones críticas.

C.3. Exclusión mutua con espera ocupada

Deshabilitando interrupciones En un sistema con un solo procesador, la solución más simple es hacer que cada proceso deshabilite todas las interrupciones justo después de entrar a su región crítica y las rehabilite justo después de salir. COn las

interrupciones deshabilitadas, no pueden ocurrir interrupciones de reloj. Después de todo, la CPU sólo se conmuta de un proceso a otro como resultado de una interrupción de reloj o de otro tipo, y con las interrupciones desactivadas la CPU no se conmutará a otro proceso.

Es conveniente para el mismo kernel deshabilitar las interrupciones poor unas cuantas instrucciones mientras actualiza variables o listas.

La posibilidad de lograr la exclusión mutua al deshabilitar las interrupciones está disminuyendo día con día debido al creciente número de chips multinúcleo que se encuentran hasta en las PCs de bajo rendimiento. En consecuencia, se requieren esquemas más sofisticados.

Solución de Peterson Al combianr la idea de variables de candado y las variables de advertencia, un matemático holandés llamado T. Dekker fue el primero en idear una solución de software pare el problema de la exclusión mutua que no requiere de una alternancia estricta.

```
#define FALSE 0
#define TRUE 1
#define N   2                                   /* número de procesos */

int turno;                                      /* ¿de quién es el turno? */
int interesado[N];                              /* al principio todos los valores son 0 (FALSE) */

void entrar_region(int proceso);                /* el proceso es 0 o 1 */
{
    int otro;                                   /* número del otro proceso */

    otro = 1 – proceso;                         /* el opuesto del proceso */
    interesado[proceso] = TRUE;                 /* muestra que está interesado */
    turno = proceso;                            /* establece la bandera */
    while (turno == proceso && interesado[otro] == TRUE)  /* instrucción nula */;
}

void salir_region(int proceso)                  /* proceso: quién está saliendo */
{
    interesado[proceso] = FALSE;                /* indica que salió de la región crítica */
}
```

Fig. 7. Solución de Peterson para lograr la exclusión mutua.

Antes de utilizar las variables compartidas, cada proceso llama a *entrarRegion* con su propio número de proceso como parámetro. Esta llamada hará que se espere, si es necesario, hasta que sea seguro entrar. Una vez haya terminado con las variables compartidas, el proceso llama a *salirRegion* para indicar que ha terminado y permitir que los demás procesos entren, si así lo desea.

Problema: Requiere espera activa.

C.4. Dormir y Despertar

Tanto la solución de Peterson como las soluciones mediante TSL o XCHG son correctas, pero todas tienen el defeto de requerir la espera ocupada. En esencia, estas soluciones comprueban si se permite la entrada cuando un proceso desea entrar a su región crítica. Si no se permite, el proceso sólo espera en un ciclo estrecho hasta que se permita la entrada.

Este método no sólo desperdicia tiempo de la CPU, sino que también puede tener efectos inesperados.

Primitivas de comunicación entre procesos que bloquean en vez de desperdiciar tiempo de la CPU cuando no pueden entrar a sus regiones críticas:

1. Sleep (dormir) $\rightarrow$ Es una llamada al sistema que hace que el proceso que llama se desbloquee o desactive.

2. Wakeup (despertar) $\rightarrow$ Esta llamada tiene un parámmetro, el proceso que se va a despertar o activar.

El problema del productor-consumidor Dos procesos comparten un búfer común, de tamaño fijo. Uno de ellos coloca la información en el búfer y el otro lo saca.

El problema surge cuando el productor desea colocar un nuevo elemento en el búfer, pero éste ya se encuentra lleno. La solución es que el productor se vaya a dormir y que se despierte cuando el consumidor se haya quitado uno o más elementos. Si el consumidor desea quitar un elemento del búfer y ve que éste se encuentra vacío, se duerme hasta que el productor coloca algo en el búfer y lo despierta.

Exclusión mutua + sincronización con sleep y wakeup.
Problema: Ambos pueden quedar dormidos a la vez.

Semáforos

En 1965, E. W. Dijkstra sugirió el uso de una variable entera para contar el número de señales de desperatar, guardadas para un uso futuro. En su propuesta introdujo un nuevo tipo de variable, al cual é le llamó **semáforo**. Un semáforo podría tener el valor 0, indicando que no se guardaron señales de despertar o algún valor positivo si estuvieran pendientes una o más señales de despertar.

Dijkstra propueso que se tuvieran dos operaciones, *down* y *up*. La operación *down* en un semáforo comprueba si el valor es mayor que 0. De ser así, disminuye el valor y sólo continúa. Si el valor es 0, el proceso se pone a dormir sin completar la operación *down* por el momento. Las acciones de comprobar el valor, modificarlo y posiblemente pasar a dormir, se realizan en conjunto como una sola **acción atómica** indivisible.

La operación *up* incrementa el valor del semáforo direccionado. Si uno o más procesos estaban inactivos en ese semáforo, sin poder completar una operación *down* anterior, el sistema selecciona uno de ellos (al azar) y permite que complete su operación *down*. Después de una operación *up* en un semáforo que contenga procesos dormidos, el semáforo seguirá en 0, pero habrá un proceso menos dormido en él.

- Down(semáforo), **operación atómica**.
  - Si semáforo > 0, decrementa el valor y continúa.
  - Si semáforo $= 0$, el proceso se pone a dormir sin completar el *down* (por el momento)..
- Up(semáforo), incrementa el valor del semáforo especificado.
- *Up* es también una **operación atómica**.
- Teóricamente, un semáforo nunca tendrá valor negativo (en alguna implementación puede darse el caso de que sí).

Un semáforo consta de dos operaciones:

1. **Bloquear** - Sitúa al proceso que se invoca en una cola de espera.

```
typedef struct{
        int value;
        struct process *list;
        } semaphore;
```

Fig. 8. Estructura de un semáforo.

2. **Desperatar** - Elimina uno de los procesos de la cola de espera y lo mueve a la cola de procesos listos.

```
#define N 100
typedef int semaforo;   /* por defecto 1 si
    binario*/
semaforo mutex = 1;             /* semaforo
    binario => Acceso a region critica,
    exclusion mutua. */
semaforo vacias = N;
semaforo llenas = 0;

void productor(void)
{
        int elemento;
        while(TRUE)
        {
                elemento = producir_elemento()
                    ;
                down(&vacias);
                down(&mutex);
                insertar_elemento(elemento);
                up(&mutex);
                up(&llenas);
        }
}
/* No cambios de contexto en las operaciones
    atomicas como down y up */
void consumidor(void)
{
        int elemento;
        while(TRUE)
        {
                down(&llenas);
                down(&mutex); /* el consumidor
                     no llegara si el
                    productor no ha llegado */
                elemento = quitar_elemento();
                    /* region critica */
                up(&mutex);
                up(&vacias);
                consumir_elemento(elemento);
        }
}
```

Monitores

Con los semáforos y los mutexes la comunicación entre procesos se ve sencilla, ¿verdad? Olvídelo. Suponga que se invirtió el orden de las dos operaciones *down* en el código del productor, de manera que *mutex* se disminuyó antes de *vacías*, en vez de hacerlo después. Si el búfer estuviera completamente lleno el productor se bloquearía, con *mutex* estrablecida en 0. En consecuencia, la próxima vez que el consumidor tratara de acceder al búfer, realizaría una operación *down* en *mutex* y se bloquearía también. Ambos procesos permanecerían bloqueados de manera indefinida y no se relizaría más trabajo. Esta desafortunada situación se conoce como interbloqueo.

Para facilitar la escritura de programas correctos, Brinch Hansen (1973) y Hoare(1974) propusieron una primitiva de sincronización de mayor nivel, conocida como **monitor**. Sus proposiciones tenían ligeras variaciones, como se describe a continuación. Un monitor es una colección de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete.

No todos los lenguajes de programación incluyen estas estructuras.

```
monitor ejemplo
        integer i;
        condition c;

        procedure productor();
        signal(c); /* P */
        i++;
        wait(c); /* Q */
        end;

        procedure consumidor();
        ...
        end;
end monitor;
```

Fig. 9. Un Monitor.

¿Cómo debería bloquearse el productor si encuentra que el búfer está lleno?

La solución está en la introducción de las **variables de condición**, junto con dos operaciones de éstas: *wait* y *signal*. Cuando un procedimiento de monitor descubre que no puede continuar realiza una operación *wait* en alguna variable de condición. Esta acción hace que el proceso que hace la llamada se bloquee. También permite que otro proceso que no haya podido entrar al monitor entre ahora. En el contexto de Phreads antes descrito vimos las variables de condición y estas operaciones.

```
monitor ProductorConsumidor
        condition llenas, vacias;
        integer cuenta;
        procedure insertar(elemento: integer);
        begin
                if cuenta = N then wait(llenas
                    );
                insertar_elemento(elemento);
                cuenta := cuenta + 1;
                if cuenta = 1 then signal(
                    vacias)
        end;
        function eliminar: integer;
        begin
                if uenta = 0 then wait(vacias)
                    ;
                eliminar = eliminar_elemento;
                cuenta := cuenta - 1;
                if cuenta = N - 1 then signal(
                    llenas)
        end;
        cuenta := 0;
end monitor;
procedure poductor;
begin
        while true do
        begin
                elemento = producir_elemento;
                ProductorConsumidor.insergtar(
                    elemento)
        End
end;
procedure consumidor;
begin
        while true do
        begin
                elemento = ProductorConsumidor
                    .eliminar;
                consumir_elemento(elemento)
        end
end;
```

Fig. 10. Un esquema del problema productor-consumidor con monitores. Sólo hay un procedimiento de monitor activo a la vez. El búfer tiene N ranuras.

C.5. El problema de los filósofos comilones

En 1965, Dijkstra propuso y resolvió un problema de sincronización al que llamó el **problema de los filósofors comilones**.

La vida de un filósofo consiste en periodos alternos de comer y pensar. Cuando un filósofo tiene hambre, trata de adquirir sus tenedores izquierdo y derecho, uno a la vez, en cualquier orden. Si tiene ésito al adquirir dos tenedores, come por un momento, después deja los tenedores y continúa pensando. La pregunta clave es: ¿puede usted escribir un programa para cada filósofo, que haga lo que se supone que debe hacer y nunca se trabe?

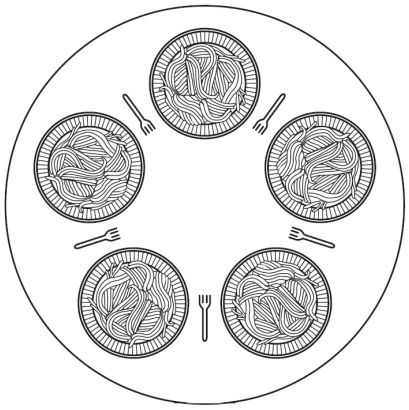


Fig. 11. Hora de comer en el Departamento de Filosofía.

```
#define N 5 /* Numero filosofos */
void filosofo(int i)
{
        while(TRUE)
        {
                pensar();
                tomar_tenedor(i);
                tomar_tenedor((i+1)%N);
                comer();
                poner_tenedor(i);
                poner_tenedor((i+1)%N);
        }
}
```

Fig. 13. Una solución incorrecta para el problema de los fil,ósofos comilones.

2. PLANIFICACIÓN DE LA CPU

LLa planificación de CPU es la base de los sistemas operativos multiprogramados. Al conmutar la CPU entre procesos, el sistema operativo puede hacer más productivo al computador.

A. Conceptos básicos

El objetivo es tener algún proceso en ejecución en todo momento, a fin de maximizar el aprovechamiento de la CPU. En un sistema mono-procesador, nunca habrá más de un proceso en ejecución. Si hay más procesos, los demás tendrán que esperar hasta que la CPU esté lista y pueda reasignarse.

La idea de la multiprogramación es relativamente simple. Un proceso se ejecuta hasta que tiene que que esperar. Todo este tiempo de espera se desperdicia; no se efectúa trabajo útil. Con la multiprogramación intentamos usar este tiempo de forma productiva. Se mantienen varios procesos en memoria a la vez. Cuando un proceso necesita esperar, el sistem aoperativo le quita la CPU y se la da a otro proceso.

La planificación es una función fundamental del sistema operativo. Casi todos los recursos del computador se planifican antes de usarse. La planificación de la CPU es parte central del diseño del sistema operativo.

```
#define N 5
#define IZQUIERDO        (i+N-1)%N
#define DERECHO          (i+1)%N
#define PENSANDO         0
#define HAMBRIENTO       1
#define COMIENTO         2
#typedef int semaforo;
int estado[N];
semaforo mutex = 1;
semaforo s[N];

void filosofo(int i)
{
        while(TRUE)
        {
                pensar();
                tomar_tenedores(i);
                comer();
                poner_tenedores(i);
        }
}
void tomar_tenedores(int i)
{
        down(&mutex);
        estado[i] = HAMBRIENTO;
        probar(i);
        up(&mutex);
        down(&s[i]);
}

void poner_tenedores(int i)
{
        down(&mutex);
        estado[i] = PENSANDO;
        probar(IZQUIERDO);
        probar(DERECHO);
        up(&mutex);
}

void probar(int i)
{
        if(estado[i] == HAMBRIENTO && estado[
            IZQUIERDO] != COMIENDO && estado[
            DERECHO] != COMIENDO)
        {
                estado[i] = COMIENDO;
                up(&S[i]);
        }
}
```

Fig. 12. Una solución al problema de los filósofos comilones.

A.1. Ciclo de ráfagas de CPU y E/S

El éxito de la planificaón depende de la siguiente propiedad: la ejecución de un proceso consiste en un *ciclo* de ejecución y esperar por E/S. Los procesos alteran entre estos dos estados. La ejecución de un proceso inicia con una *ráfaga* de CPU, seguida de una *ráfaga* de E/S, seguida de otra ráfaga de CPU, luego otra ráfaga de E/S...

Aunque en muchas mediciones realizadas se ha obtenido que las duraciones de las ráfagas de CPU varían considerablemente de un proceso a otro y de un computador a otro, tienden a tener una curva de frecuencia similar a la que se muestra en la figura **??**.

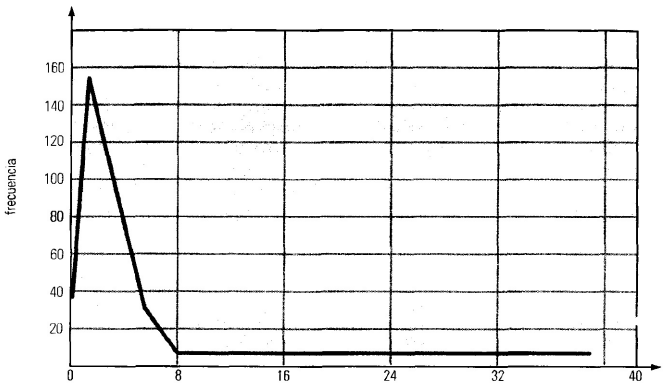


Fig. 14. Histograma de tiempos de ráfaga de CPU.

Un programa limitado por E/S suele tener muchas ráfagas de CPU muy cortas; uno limitado por CPU podría tener unas cuantas ráfagas de CPU muy largas. Esta distribución puede ser importante para la selección de un algoritmo de planificaón de la CPU apropiado.

A.2. Planificador de CPU

Siempre que la CPU está ociosa, el sistema operativo debe escoger uno de los procesos que están en la cola de procesos listos para ejecutarlo. El proceso de selección ejecuta el planificador a corto plazo, el cual escoge uno de los procesos en memoria listos para ejecutarse y le asigna la CPU.

La cola de procesos listos no es necesariamente una cola de primero que entra, primero que sale (FIFO). Una cola de procesos listos puede implementarse como cola FIFO, cola por prioridad, árbol o simplemente como lista enlazada no ordenada. Desde el punto de vista conceptual todos los procesos listos están haciendo fila en espera de una oportunidad de ejecutarse en la CPU.

A.3. Planificación expropiativa

Las decisiones de planificación de la CPU se toman en las cuatro situaciones siguientes:

1. Cuando un proceso pasa del estado en ejecución al estado en espera.
2. Cuando un proceso pasa del estado de ejecución al estado listo.
3. Cuando un proceso pasa del estado en espera al estado listo.
4. Cuando un proceso termina.

En las circunstancias 1 y 4, no hay opción en términos de planificación. Se deberá escoger un proceso nuevo para ejecutarse. En las situaciones 3 y 4 hay opciones.

Si la planificación tiene lugar en las circunstancias 1 y 4, decimos que el esquema de planificación es *no expropiativo*;

en los demás casos, es *expropiativo*. Con una planificación no expropiativa, una vez que la CPU se ha asignado a un proceso, éste la conserva hasta que la cede ya sea porque terminó o porque pasón al estado en espera.

Desafortunadamente, implica un costo: en el caso de dos procesos que comparten datos. Uno podría estar a la mitad de una operación de actualización de los datos en el momento en el que se le desaloja y se ejecuta el segundo proceso. Éste podría intentar leer los datos, que por el momento están en un estado inconsistente. Por tanto, se requieren nuevos mecanismos para coordinar el acceso a datos compartidos.

También afecta el diseño del núcleo del sistema operativo. En una llamada al sistema, el núceo podría estar ocupado con una actividad a nombre de un proceso. Podrían incluir la modificación de datos importantes del núcleo, es decir: un caos. Ciertas implementaciones resuelven este problema esperando hasta que se completa la llamada al sistema, o hasta que ocurre un bloqueo por E/S, antes de efectuar una conmutación de contexto. Asegura que el núcleo tenga una estructura sencilla, ya que el núcleo no desalojará un proceso mientras las estructuras de datos del núcleo estén en un estado inconsistente. La desventaja es que este modelo de ejecución del núcleo es deficiente en cuanto al apoyo para la computación en tiempo real y el multiprocesamiento.

En el caso de UNIX, todavía hay secciones del código que están en peligro. Dado que las interrupciones puede, por definición, ocurrir en cualquier momento, y dado que el núcleo no siempre puede hacer caso omiso de ellas, las secciones del código afectadas por las interrupciones deben protegerse contra un uso simultáneo.

A.4. Despachador

Otro componente de planificación de la CPU es el *despachador*. Cede el controlo de la CPU al proceso seleccionado por el planificador a corto plazo:

- Cambiar de contexto.
- Cambiar a modo de usuario.
- Saltar al punto apropiado del programa del usuario para reiniciar ese programa.

El despachador debe ser lo más rápido posible, se invoca en cada conmutación de proceso. El tiempo que el despachador tarda en detener un proceso y poner otro en ejecución se denomina *latencia del despachador*.

B. Criterios de planificación

Los diferentes algoritmos de planificación de la CPU tienen diferentes propiedades y podrían favorecer a una clase de procesos más que a otra. Al escoger debemos considerar las propiedades de diversos algoritmos.

- **Utilización de la CPU**: Queremos mantener la CPU tan ocupada como se pueda.
- **Rendimiento**: Si la CPU está ocupada ejecutando procesos, se está efectuando trabajo. Una medida del trabajo es el número de procesos que se completan por unidad de tiempo: el *rendimiento*.
- **Tiempo de retorno**: El creiterio importante es el tiempo que tarda la ejecución de ese proceso. *Tiempo de retorno* o *tiempo de servicio*, y es la suma de los periodos durante los cuales espera entrar en la memoria.
- **Tiempo de espera**: El algoritmo de planificación no afecta la cantidad de tiempo que un proceso pasa ejecutándose o realizando E/S. El tiempo de espera es la suma de los periodos que el proceso pasa esperando en dicha cola.
- **Tiempo de respuesta**: Un proceso puede producir algunas salidas en poco tiempo. Otra medida es el tiempo que transcurre entre la presentación de una solicitud y la producción de la primera respuesta. Llamada *tiempo de respuesta*, es el tiempo que el proceso tarda en comenzar a responder, pero no incluye el tiempo que toma exhibir la respuesta.

Es deseable maximizar el aprovechamiento y la productividad de la CPU, y minimizar los tiempos de retorno, espera y respuesta.

C. Algoritmos de planificación

C.1. Planificación de "servicio por orden de llegada"

El algoritmo de planificación más sencillo es el de *servicio por orden de llegada(FCFS, first-come, first served)*. La implementación de la política FCFS es fácil como una cola FIFO. El tiempo de espera promedio suele ser muy largo.

Proceso	Tiempo de ráfaga
P_1	24
P_2	3
P_3	3

Si los procesos llegan en el orden P_1, P_2,P_3, y se atienden en orden FCFS, obtenemos el resultado que se muestra en el siguiente *diagrama de Gantt*:

El tiempo de espera es de 0 milisegundos para el proceso P_1, 24 milisegundos para el proceso P_2, y 27 milisegundos para el proceso P_3. Entonces, el tiempo de espera promedio es $(0 + 24 + 27)/3 = 17$ milisegundos. En cambio, si los procesos llegan en orden P_2, P_3, P_1, los resultados son los que se muestran en este diagrama de Gantt:

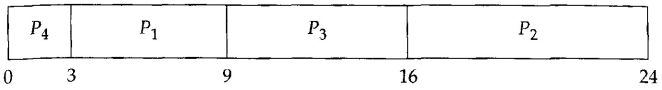
El tiempo de espera promedio ahora e $(6 + 0 + 3)/3 = 3$ milisegundos. Esta reducción es sustancial. Así, el tiemop de espera promedio cuando se instituye una política FCFS casi nunca es mínimo, y podría variar sustancialmente si los tiempos de ráfaga de CPU del proceso varían mucho.

C.2. Planificación de "primero el trabajo más corto"

El algoritmo de *primero el trabajo más corto (SJF, shortest job first)* asocia a cada proceso la longitud de la siguiente ráfaga de CPU de ese proceso. Cuando la CPU queda disponible, se asigna al proceso cuya siguiente ráfaga de CPU sea más corta. Si hay dos procesos cuyas siguientes ráfagas de CPU tienen la misma duración, se emplea planificación FCFS para romper el empate. Por ejemplo:

Proceso	Tiempo de ráfaga
P_1	6
P_2	8
P_3	7
P_4	3

Utilizando una política SJD, planificaríamos estos procesos según el diagrama de Gantt siguiente:



El tiempo de espera es de 3 milisegundos para el proceso P_1, 16 milisegundos para el prceso P_2, 9 milisegundos para el proceso P_3 y 0 milisegundos para el proceso P_4. Así, el tiempo de espera promedio es $(3 + 16 + 9 + 0)/4 = 7$ milisegundos.

El algoritmo de planificación SJF es *óptimo*, en cuanto a que da el tiempo de espera promedio mínimo para un conjunto dado de procesos. Si atendemos a un proceso corto antes que uno largo, el tiempo de espera del proceso corto disminuirá más de lo que aumenta el tiempo de espera del proceso largo. En consecuencia, el tiempo de espera *promedio* disminuye.

Lo difícil del algoritmo SJF es conocer la duración de la siguiente solicitud de CPU. La planificación SJF se usa a menudo en la planificación a largo plazo. No se puede implementar en nivel de pla aplicación a corto plazo. No hay forma de conoceer la duración de la siguiente ráfaga de CPU.

Sea t_n la duración de la n-ésima ráfaga de CPU, y sea τ_{n+1} el valor que predecimos para la siguiente ráfaga de CPU. Entonces, para α, $0 \leq \alpha \leq 1$, definimos:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

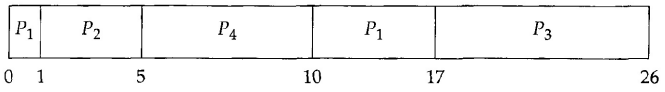
Esta fórmula define un promedio exponencial. Para entender el comportamiento del promedio exponencial podemos expandir la fórmula para t_{n+1} sustituyendo t_n; obtenemos

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0$$

Ejemplo:

Proceso	Tiempo de llegada	Tiempo de ráfaga
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Si los procesos llegan a la cola de procesos listos en los instantes que se indican y necesitan los tiempos de ráfaga dados, el plan SJF expropiativo resultante es como el que se muestra:



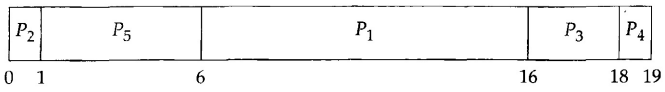
Una planificación expropiativa tendría un tiempo de espera promedio de 7.75 milisegundos.

C.3. Planificación por prioridad

El algoritmo SJF es un caso especial del algoritmo de planificación por *prioridad general*. La CPU se asigna al proceso que tiene la prioridad más alta. Un algoritmo por prioridad en el que la prioridad (p) es el recíproco de la duración de la siguiente ráfaga de CPU.

Proceso	Tiempo de ráfaga	Prioridad
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Con el siguiente diagrama de Gantt:



La planificación por prioridad puede ser expropiativa o no expropiativa. Un problema importante es el bloqueo indefinido o inanición. Una solución al problema del bloqueo indefinido de procesos de baja prioridad es el envejecimiento, que consiste en aumentar gradualmente la prioridad de los procesoss que esperan mucho tiempo en el sistema.

C.4. Planificación por turno circular

El algoritmo de planificación por *turno circular(RR, round-robin)* se diseñó especialmente para los sistemas de tiempo compartido y es similar a la planificación FCFS, pero con la adición de expropiación para conmutar entre procesos. Se define una unidad de tiempo pequeña que generalmente es de 10 a 100 milisegundos. La cola de procesos listos se trata como cola circular. el planificador recorre la cola de procesos listos, asignando la CPU a cada proceso durante un intervalo de tiempo.

Para implementar la planificación RR, mantenemos la cola de procesos listos como estructura FIFO. Los procesos

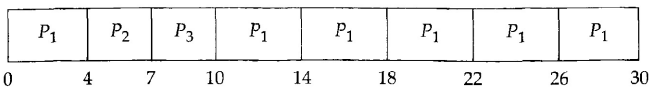
nuevos se añaden al final de la cola. El planificador escoge el primer proceso de la cola, ajusta un temporizador de modo que interrumpa después de un tiempo, y despacha el proceso.

A continuación sucederá una de dos cosas. El proceso podría tener una ráfaga de CPU más corta que el tiempo. En este caso, el proceso mismo liberará la CPU voluntariamente. En caso contrario, si la ráfaga de CPU del proceso que se está ejecutando dura más de un cuanto de tiempo, el temporizador terminará y generará una interrupción para el sistema opeartivo. Se efectuará una conmutación de contexto, y el proceso se colocará *al final* de la cola de procesos listos. El planificador escogerá entonces el siguiente proceso de la cola.

El tiempo de espera promedio cuando se instituye una política RR suele ser muy grande:

Proceso	Tiempo de ráfaga
P_1	24
P_2	3
P_3	3

Si el cuanto de tiempo es de 4 milisegundos, el proceso P_1 obtenderá los primeros 4 milisegundos. Dado que requiere 20 milisegundos más, será desalojado después del primero cuanto, y se concederá la CPU al siguiente proceso de la cola, P_2. Como este proceso no necesita 4 milisegundos, terminará antes de que expire su cuanto y la CPU se asignará al siguiente proceso, P_3. Una vez que todos los procesos han recibido un cuanto de tiempo, se devuelve la CPU al proceso P_j durante un cuanto adicional. El plan RR resultante es:



El tiempo de espera promedio es de $17/3 = 5.66$ milisegundos.

El desempeño del algoritmo RR depende mucho del tamaño del cuanto de tiempo.. En un extremo, si el cuanto de tiempo es muy grande (infinito), la política RR equivale a FCFS. Si el cuanto es muy pequeño la estrategia RR es la denominada compartir al procesador. Desde el punto de vista del usuario parece comosi cada uno de los n procesos tiviera su propio procesador que se ejecuta a $1/n$ de la velocidad del procesador real. Este enfoque se adoptó en el hardware de la COntrol Data Corporation (CDC) para implementar 10 procesadores periféricos con un solo juego de hardware y 10 juegos de registros.

También es necesario considerar el efecto de la conmutación de contexto sobre el desempeño de la planificación RR. Es importante que el cuanto de tiempo sea grande en comparación con el tiempo que tarda una conmutación de contexto. Si este último tiempo es aproximadamente el 10% del cuanto, aproximadamente el 10% del tiempo de CPU se gastará en connmutar contextos.

C.5. Planificación con colas de múltiples niveles

Se ha creado otra clase de algoritmos de planificación para situaciones en las que es fácil clasificar los procesos en diferentes grupos. Por ejemplo, suele hacerse una división entre los procesoss que se ejecutan en primer plano y los que lo hacen en segundo plano. Éstos tienen diferentes necesidades en cuanto al tiempo de respuesta, así que podrían tener diferentes necesidades de planificación. Los procesos de primer plano podrían tener mayor prioridad que los de segundo plano.

Un algoritmo de planificación con colas de múltiples niveles divide la cola de procesos listos en varias colas distintas. Los procesos se asignan permanentemente a una cola. Cada cola tiene su propio algoritmo de planificación. Por ejemplo, podrían utilizarse colas distintas para los procesos de primer y segundo plano. La cola de primer plano podría planificarse según un algoritmo RR, y la de segundo plano, con un algoritmo FCFS.

C.6. Planificación con colas de múltiples niveles y realimentación

En un algoritmo de planificación con colas de múltiples niveles, lo normal es que los procesos se asignen permanentemente a una cola al ingresar en el sistema. Los procesos no se mueven de una cola a otra. Este método tienen la ventaja de que el gasto extra por planificación es bajo, y la desventaja de que es inflexible.

La planificación con colas de múltiples niveles y realimentación, en cambio, permite a un proceso pasar de una cola a otra. La idea es separar procesos con diferentes carracterísticas en cuanto a sus ráfagas de CPU. Si un proceso gasta demasiado tiempo de CPU, se le pasará a una cola con menor prioridad. Si un proceso espera demasiado tiempo en una cola de baja prioridad, podría pasarse a una de mayor prioridad. Esta forma de envejecimiento evita la inanición.

En general, un planificador de colas multinivel conrealimentación está definido por los parámetros siguientes:

- El número de colas
- El algoritmo de planificación para cada cola
- El método empleado para determinar cuándo se debe promover un proceso a una cola de mayor prioridad
- El método empleado para determinar cuándo se debe degradar un proceso a una cola de menor prioridad
- El método empleado para determinar en cuál cola ingresará un proceso cuando necesite un servicio

La definición de un planificador de colas multinivel con realimentación lo convierte en el algoritmo de planificación de la CPU más general, ya que se le puede configurar para adaptarlo a cualquier sistema específico que se esté diseñando.

REFERENCES

1. Andrew S. Tanenbaum, Sistemas Operativos Modernos, Tercera Edición, Ed. Pearson (2009).
2. Silberschatz Galvin, Sistemas Operativos, Quinta Edición, Ed. Addison Wesley (1999).

SOBRE MI

Joaquín Roiz Pagador

Estudiante del Grado en Ingeniería Informática en Sistemas de Información en la universidad *Pablo de Olavide*, promoción de 2016.