CENTRO UNIVERSITARIO
DE TECNOLOGÍA
Y ARTE DIGITAL

Sistemas Operativos

3. Gestión de procesos

Javier García Algarra
javier.algarra@u-tad.com
Miguel Ángel Mesas
miguel.mesas@u-tad.com
2021-2022

Introducción

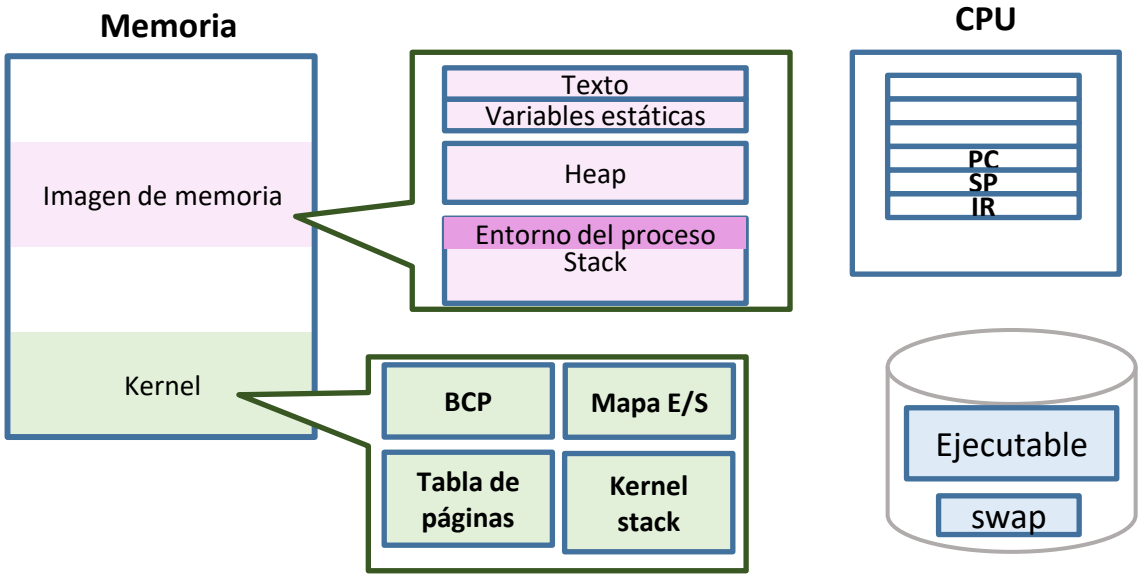
En la lección anterior hicimos una analogía con la programación orientada a objetos. El programa ejecutable es la clase y el proceso es una instancia de dicha clase.

Al igual que los objetos, los procesos se tienen que instanciar, inicializar, ejecutar y destruir de forma ordenada. La gestión de procesos es la función clave del Sistema Operativo. Tiene qué decidir en todo momento qué proceso ocupa la CPU, debe ser capaz de que los cambios de contexto sean transparentes al programador, debe asegurar que todos reciben atención adecuada en función de sus características… y todo en el tiempo más reducido posible.

En esta lección aprenderemos todas estas funciones usando ejemplos de Linux, xv6 y BSD.

Proceso

Un proceso es un programa en ejecución. El proceso consta de una **imagen de memoria**, compuesta por el código ejecutable, sus variables, el *heap*, el *stack* y las librerías que usa si son dinámicas (si son estáticas son parte del texto). Esta imagen de memoria se instancia en la memoria física del ordenador de forma completa si trabaja con memorial real o una combinación de memoria y disco si usa memoria virtual.



Mapa del Proceso

Además de los datos en memoria, el proceso necesita otras informaciones para funcionar correctamente.

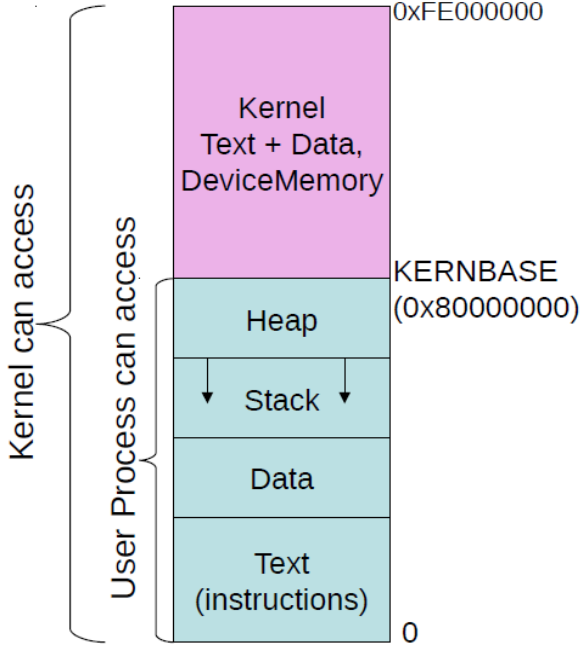
- **Registros de la CPU**. Mientras dura la ejecución de un proceso los registros de propósito general contienen datos de su propiedad. También es información crítica para el proceso la del contador de programa (PC), puntero de pila (SP) o registro de interrupción (IR), entre otros. Estos registros se guardarán en el BCP proceso si es expulsado de la CPU para poder volver al punto correcto de ejecución. Se conocen como **estado del procesador.**
- **Entorno.** Datos globales que hereda el proceso al nacer como $PATH o $HOME. Son pares clave/valor que se pasan por el stack del proceso al crearlo.
- **Metadatos**. Datos sobre el proceso que produce el Sistema Operativo durante la ejecución y que residen en la zona de memoria del kernel. El **Bloque de Control de Proceso (BCP)** guarda la identidad del proceso, su prioridad, su estado, etc. La **tabla de páginas** guarda la correspondencia entre las páginas de memoria virtual y los marcos en los que está cargado el proceso. El **mapa de E/S** almacena información sobre los descriptores de dispositivos de entrada/salida y ficheros que usa el proceso.
- En el **disco** se guarda el código ejecutable del programa. También las páginas de la zona de swap cuando hay memoria virtual.

Mapa del proceso en xv6

En xv6 la imagen del proceso se instancia entre la dirección 00000000H y 80000000H. Por encima de esa dirección y hasta FE000000H se ubica el kernel. Este es el mapa de memoria que ve la CPU.

Los procesos tienen dos stacks, uno en la zona de usuario (stack del proceso) y otro en la del kernel (stack del kernel). Esto es posible porque la arquitectura de Intel ofrece dos registros distintos SP y SP' con este fin.

La existencia de dos pilas permite no tener que hacer cambio de contexto solo por el paso de modo usuario a supervisor. Además, aunque la pila del proceso se corrompa, la del kernel no sufre y puede tratar la excepción



Mapa del proceso en xv6

Cada proceso tiene su propio BCP, con la siguiente estructura (fichero `proc.h`):

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Tamaño
Puntero a la tabla de páginas
Pila del kernel
Estado
pid
Puntero al BCP del padre
Estado del procesador
Contexto de la CPU en la pila del kernel

Vector de ficheros abiertos
Directorio de ejecución
Nombre del ejecutable

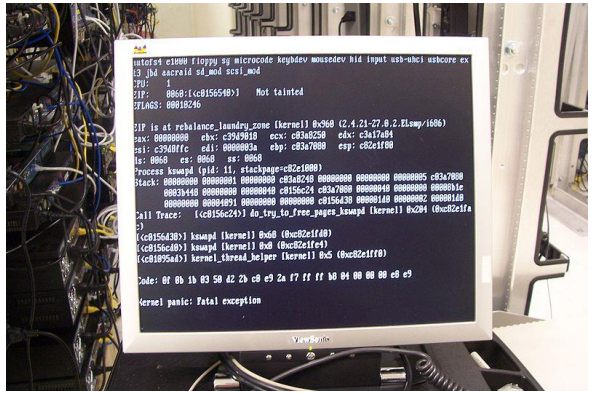
La estructura del BCP en Linux se llama `task_struct` y puede consultarse aquí:

https://www.tldp.org/LDP/tlk/ds/ds.html

Mapa del proceso en Linux

La estructura del BCP en Linux, es mucho más compleja que en xv6. Se llama `task_struct` y puede consultarse en https://www.tldp.org/LDP/tlk/ds/ds.html

La `task_struct` se instancia en la pila del espacio del kernel del proceso. Para que el sistema pueda recorrer todos los BCP se organizan como una lista circular doblemente ecandenada, esto es, cada `task_struct` contiene un puntero hacia el stack del proceso anterior y del siguiente. Cuando se crea o destruye un proceso, el SO tiene que ser capaz de insertar o eliminar el BCP de la lista. Recuerda que un fallo en el funcionamiento de esta parte crítica provocaría un *kernel panic (*una "muerte azul" en Windows, por ejemplo).



Un kernel panic en Unix. Mal asunto

Ciclo de vida del proceso

La vida del proceso abarca desde su primera invocación hasta la liberación de todos recursos que ocupaba, tanto en el espacio usuario como en las tablas del sistema. La ejecución se verá interrumpida en múltiples ocasiones durante su existencia.

Creación

El proceso solo se crea por petición expresa de otro proceso (primitivas `fork()` y `exec()`). Así, cuando en la shell invocamos cualquier comando (`ls`, `cat`, `rm`,…) la shell es el proceso padre del ejecutable de cada uno de esos comandos. Las operaciones necesarias para crear un proceso son:

- Asignar la imagen de memoria. Si el sistema usa memorial virtual (lección 4) esta parte incluirá crear y rellenar la tabla de páginas.
- Seleccionar un BCP libre en la tabla correspondiente del kernel y rellenarlo con los datos del nuevo proceso.
- Cargar el ejecutable y los datos estáticos inicializados en la memoria del proceso.
- Crear la pila y el heap.

Mientras duren estas tareas en el estado del proceso en **Nuevo**. Al completarse se marca como **Listo** y entra en la cola del planificador.

Ciclo de vida del proceso

Activación

La activación del proceso consiste en ponerlo en ejecución, de ello se ocupa un componente del *kernel* llamado *dispatcher*. Para activarlo hay que recuperar los valores de los registros del micro (estado del procesador) que el proceso tenía antes de la última interrupción. Esos datos están almacenados en el BCP.

El kernel entra en ejecución en modo privilegiado, salva el contexto del proceso que estaba ocupando la CPU, rescata el nuevo del BCP del proceso seleccionado y antes de retornar de la interrupción al modo usuario coloca en el PC el valor de retorno de la dirección siguiente a aquella en que se interrumpió el proceso activado.

Todo este proceso es lo que se conoce como cambio de contexto y es muy costoso en tiempo. En sistemas con memoria virtual, la tabla de páginas cambia y se recarga la TLB (circuito de traducción que se explica con detalle en la lección 4).

Ciclo de vida del proceso

Interrupción

La interrupción es una fase crítica porque activa al kernel. Puede producirse por un evento hardware o por una interrupción software (también conocido como *trap*). La interrupción hace que el procesador ejecute en modo privilegiado.

Lo primero que hace el kernel es atender la interrupción en sí misma, enmascarando el resto. A continuación salva los registros del procesador visibles por el usuario en el stack de usuario del proceso que se está ejecutando.

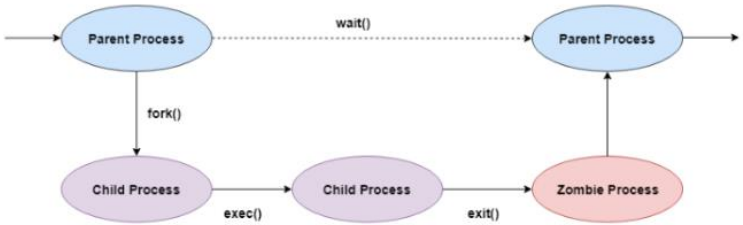
La rutina de atención de la interrupción puede ser muy compleja pero en algún momento finalizará. El kernel puede haber bloqueado al proceso como consecuencia de la interrupción o haberlo expulsado a la cola de procesos en espera de ejecución (estado **Listo**). Si se produce una de esas dos circunstancias, el scheduler deberá decidir cual es el nuevo proceso que ocupa la CPU y proceder al cambio de contexto. Si, por el contrario, el proceso actual puede seguir ejecutándose, el kernel recupera el contexto desde la pila de usuario y al invocar la instrucción de retorno de interrupción, se regresa al entorno de usuario en el punto en que se interrumpió.

Ciclo de vida del proceso

Finalización

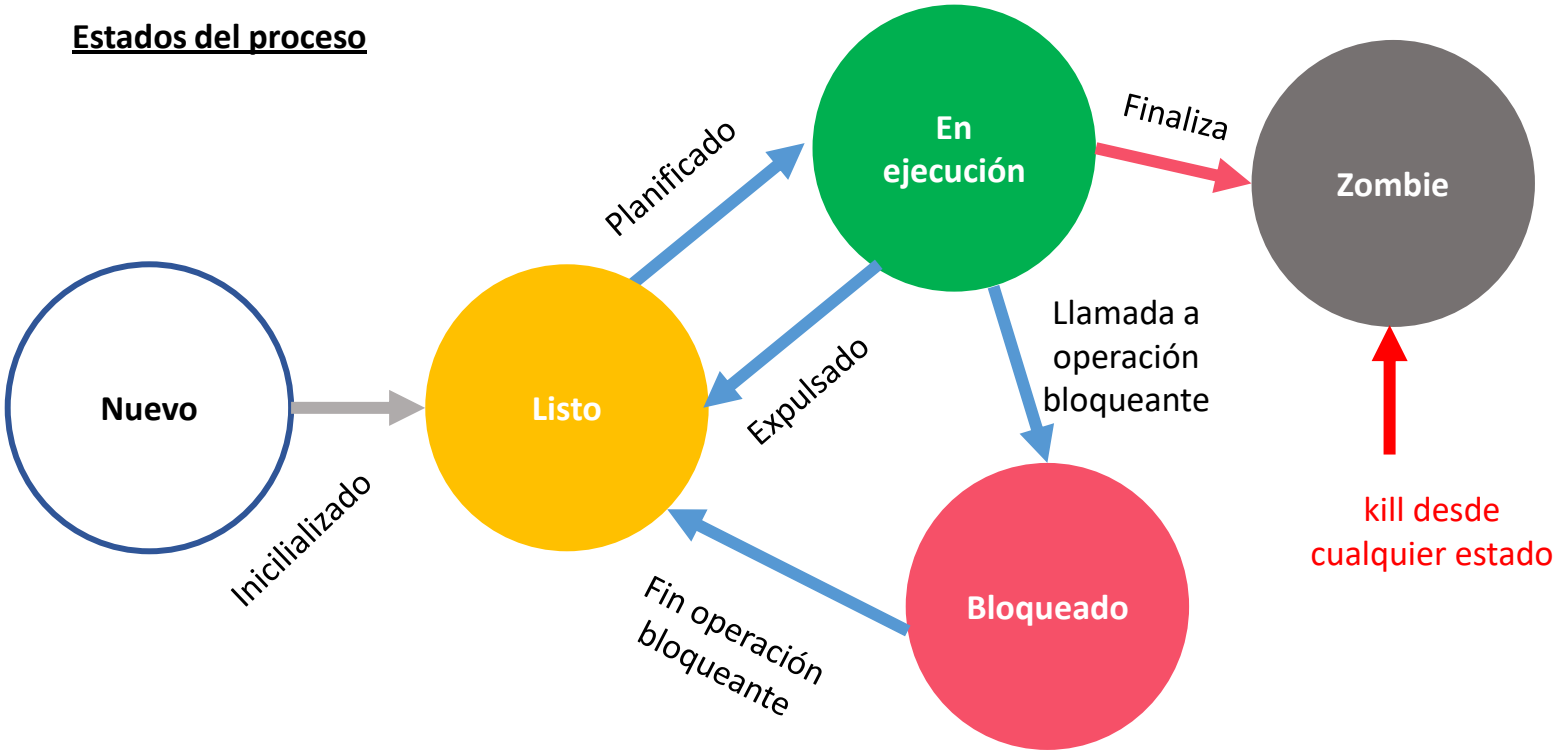
Un proceso termina cuando el programador así lo decide, por ejemplo usando `exit()` o `return()` en la función `main()` de un programa en C. También puede hacerlo de forma prematura, al recibir una señal como consecuencia de una situación anormal, como división por cero, violación de segmento, etc…

Sea cual sea el motivo, la finalización del código del usuario no significa que el proceso deje de existir de inmediato. Hay recursos del Sistema Operativo que deben liberarse, como el BCP o la tabla de memoria y funciones que dependen del proceso muerto, como `wait()`. Así, un proceso padre puede estar bloqueado esperando la muerte de un proceso hijo. Por estas razones, el proceso no muere de inmediato sino que pasa por un estado terminal llamado zombie.



Zombie Process in Linux

Ciclo de vida del proceso



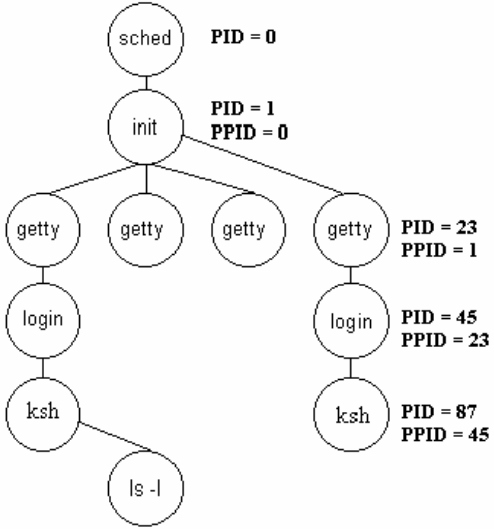
Creación

En UNIX todos los procesos descienden de un único padre, el proceso `init` que se crea durante el arranque. Hay dos mecanismos para crear procesos, `fork()` que produce un clon del proceso que lo invoca y `exec()` que crea un proceso con un ejecutable distinto.

Cuando un proceso invoca un fork(), tanto el padre como el hijo continúan la ejecución en el retorno de la función. La forma de distinguirlos es porque devuelve la identidad del hijo, si ese valor es 0 entonces se trata del hijo, de lo contrario es el padre.

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

int main(int argc, char *argv[])
{
    int child;
    if ((child = fork()) > 0)
        fprintf(stdout, "Parent process");
    else if (child == 0)
        fprintf(stdout, "Children process");
}
```



En xv6 el proceso inicial está en el fichero `initcode.S`

En Unix en `/sbin/init` y arranca como hijos los scripts especificados en `/etc/init.d`

Creación

Código de fork() en xv6
Fichero proc.c

Datos del proceso padre

Busca un BCP no usado, fijar pid como un número secuencial y crear el stack del nuevo proceso.

Copia la tabla de páginas de memoria del padre al hijo, si no puede libera los recursos y devuelve -1 que es el valor de retorno de error de la función

Copia de valores del BCP padre (tamaño, ppid y trapframe, que es el contexto del procesador)

Devuelve pid == 0 en el registro eax si es el hijo

Copia de otros valores del padre

Se marca el proceso como RUNNABLE (Listo)

Devuelve el pid del hijo

```
int fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

Cambio de contexto (Activación)

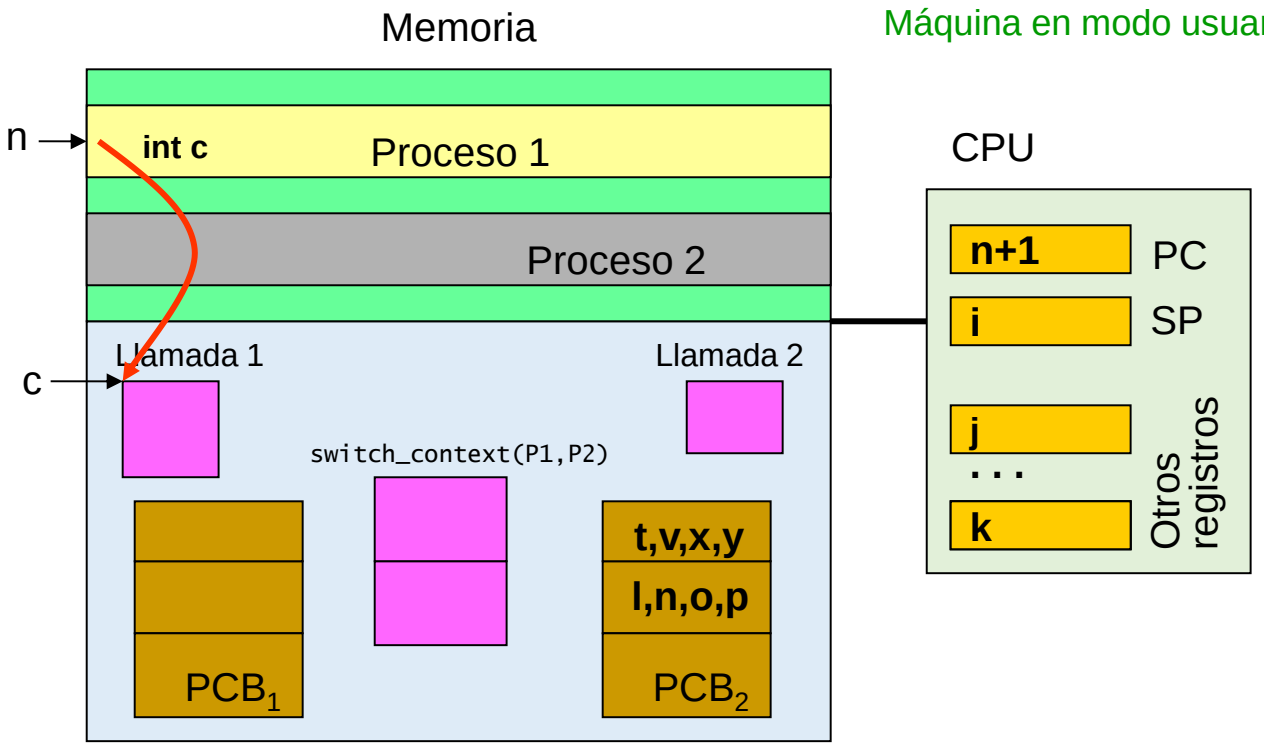
Se **denomina cambio de contexto** al procedimiento de desalojar a un proceso de la CPU y sustituirlo por otro. Es consecuencia de una decisión del *scheduler*. La función del kernel que se encarga de esta tarea se llama *dispatcher*.

No hay que confundir el cambio de contexto con el paso de ejecución de modo usuario a modo supervisor que ocurre cuando llega una interrupción. Esta situación es mucho más liviana en cuanto a consumo de CPU, el estado del procesador se guarda en el *stack* del usuario, la interrupción se atiende, se recupera el estado desde el propio *stack* y el proceso sigue su ejecución. Puede ocurrir que como consecuencia de la interrupción el *scheduler* decidiera realizar un cambio de contexto, pero no siempre es así. En esta operación no hay cambio de la tabla de páginas de memoria, ni del código ejecutable, no se pierde la ventaja de la proximidad referencial.

En el cambio de contexto hay que salvar los registros del procesador cuando se produjo la interrupción en su BCP. También hay que salvar los registros en el momento en que el kernel llama a la función de cambio de contexto. A continuación se recuperan los valores de los registros del BCP del proceso que se va a instanciar y finalmente se le concede control.

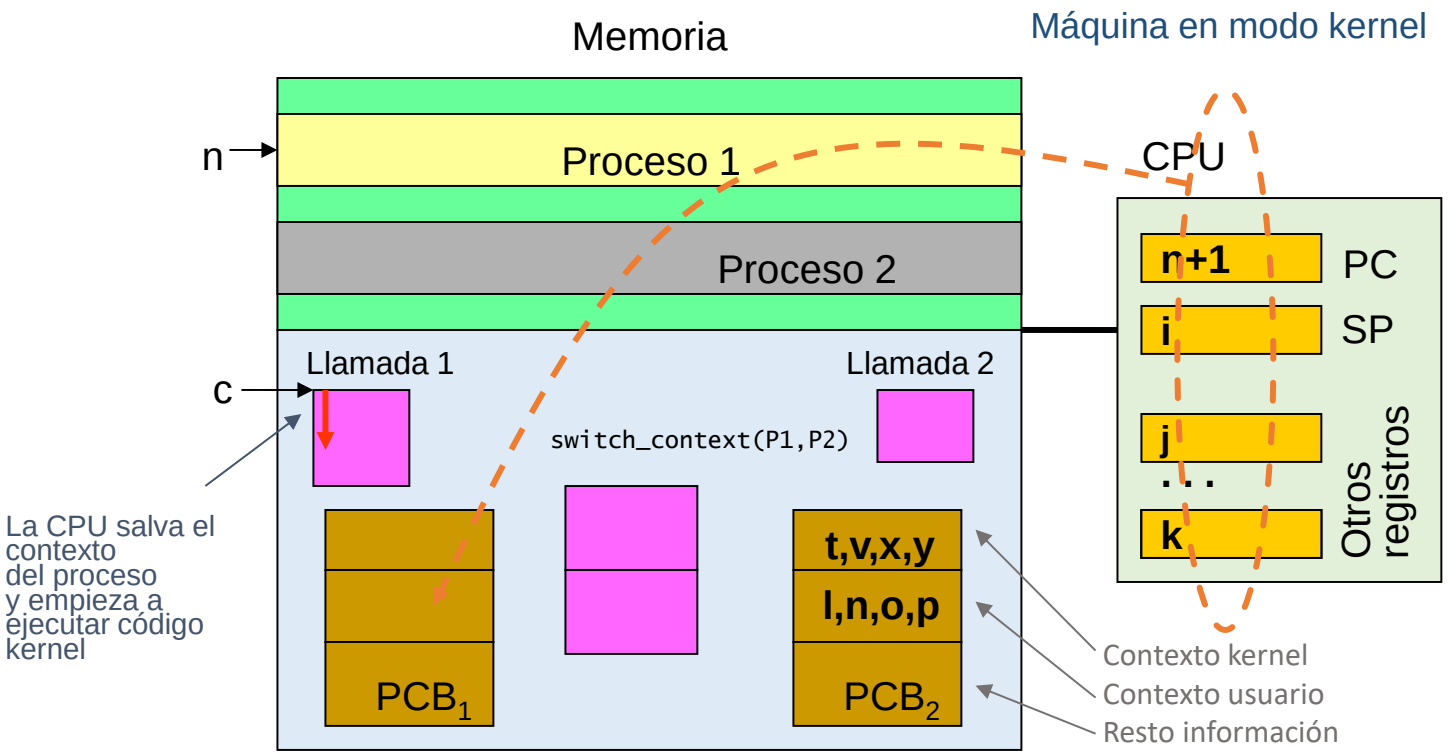
Cambio de contexto (Activación)

Inicialmente se está ejecutando el proceso 1 y se produce una interrupción por invocación de una operación bloqueante cuando el contador de programa ejecutaba la instrucción n.



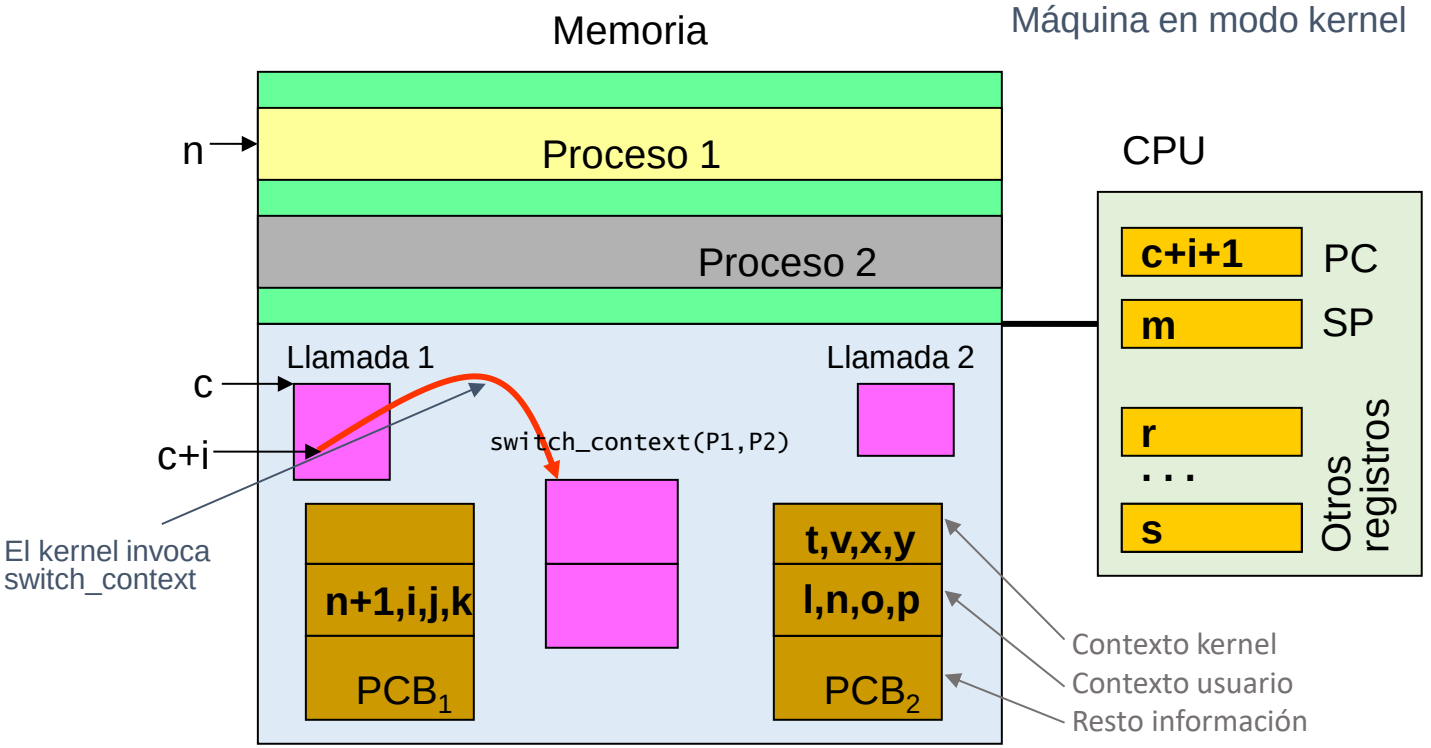
Autor: José Antonio Gómez, UGR, reproducido con permiso

Cambio de contexto



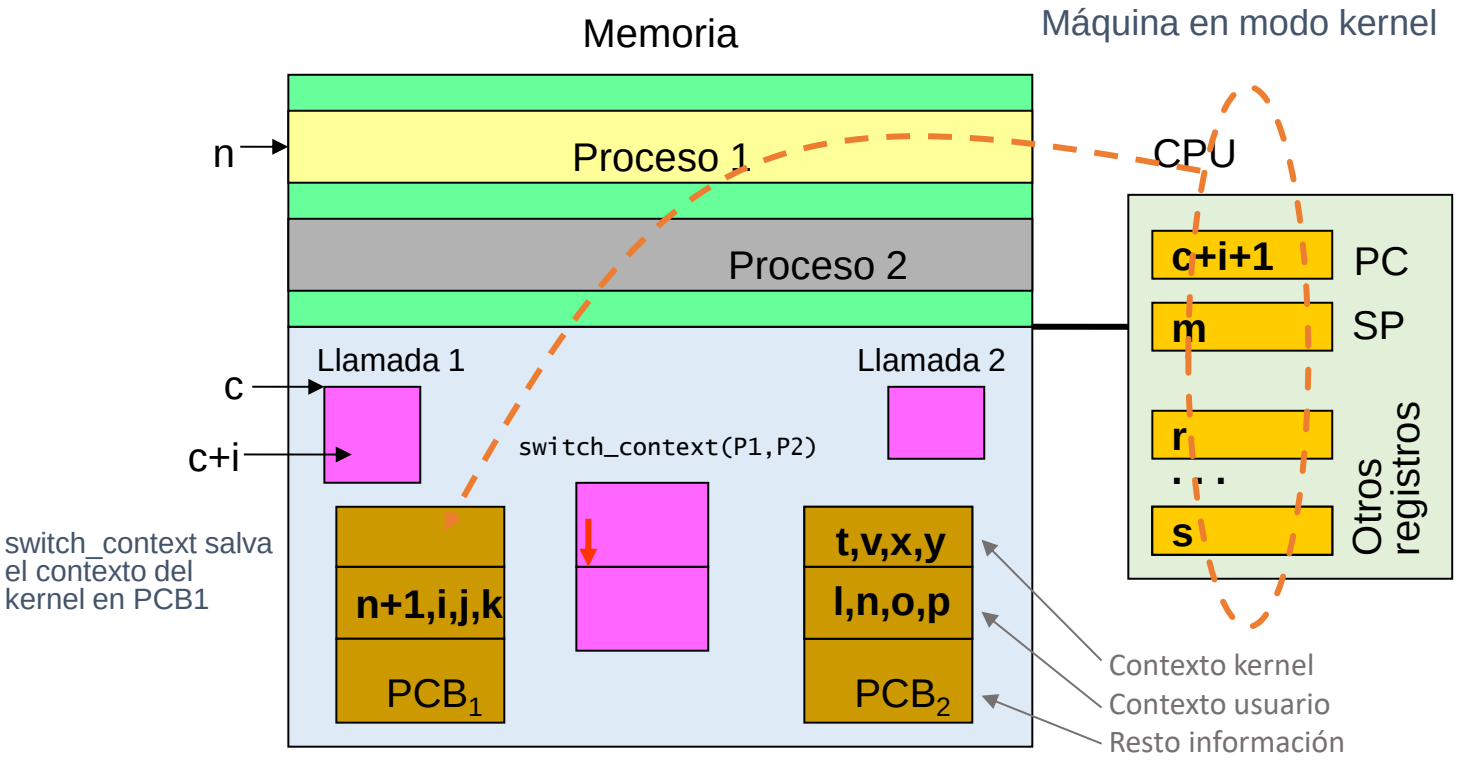
Autor: José Antonio Gómez, UGR, reproducido con permiso

Cambio de contexto



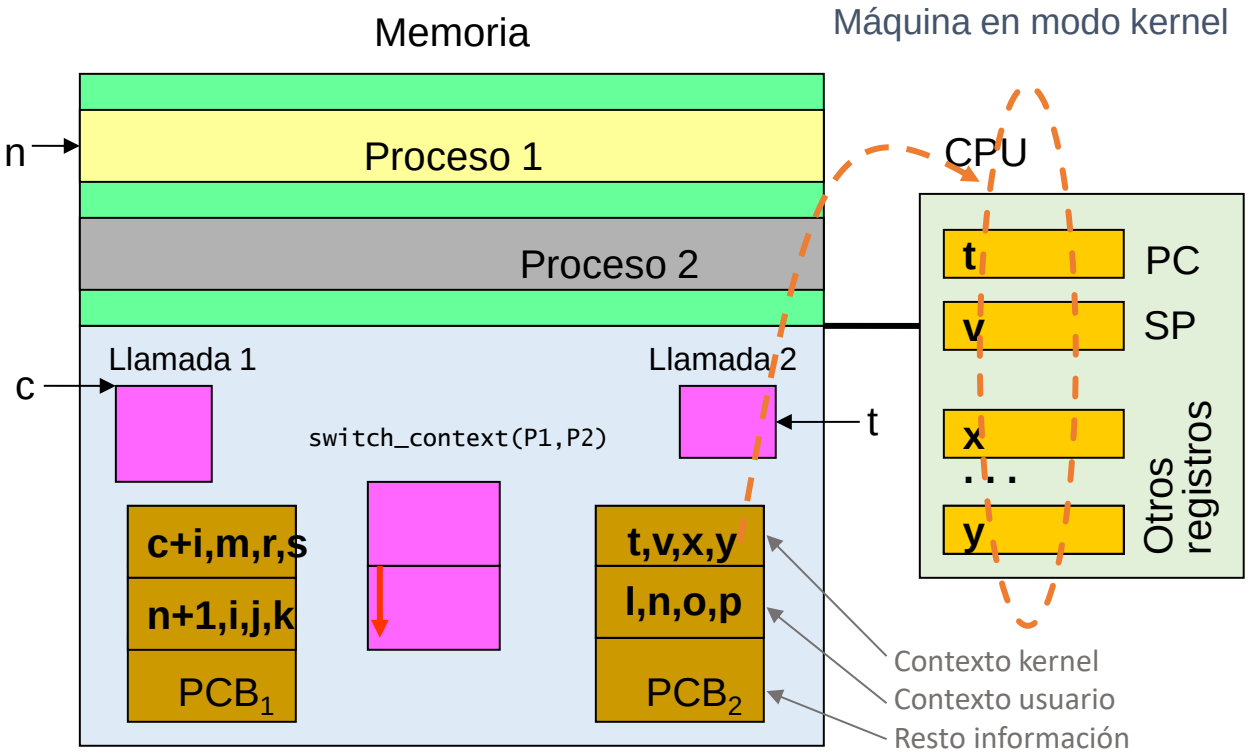
Autor: José Antonio Gómez, UGR, reproducido con permiso

Cambio de contexto



Autor: José Antonio Gómez, UGR, reproducido con permiso

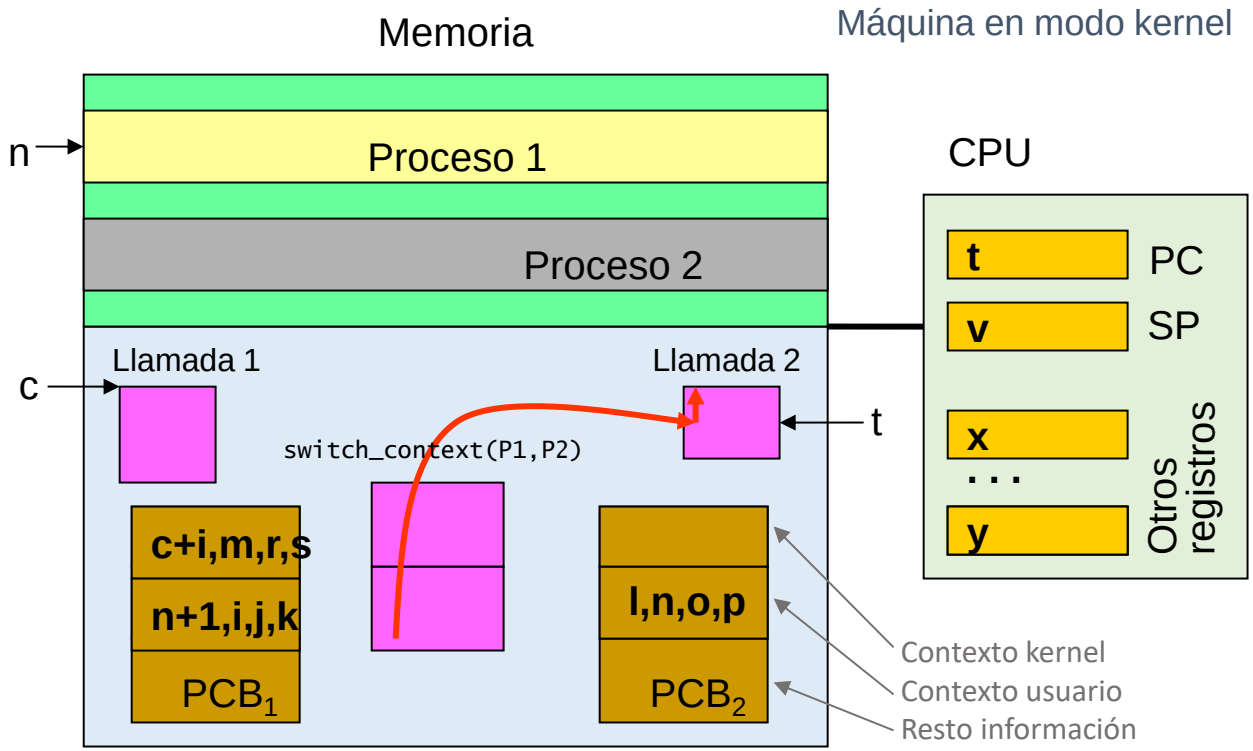
Cambio de contexto



El kernel restaura el contexto del procesador desde PCB2 con los datos del stack de kernel. Está en el punto en que bloqueó a P2.

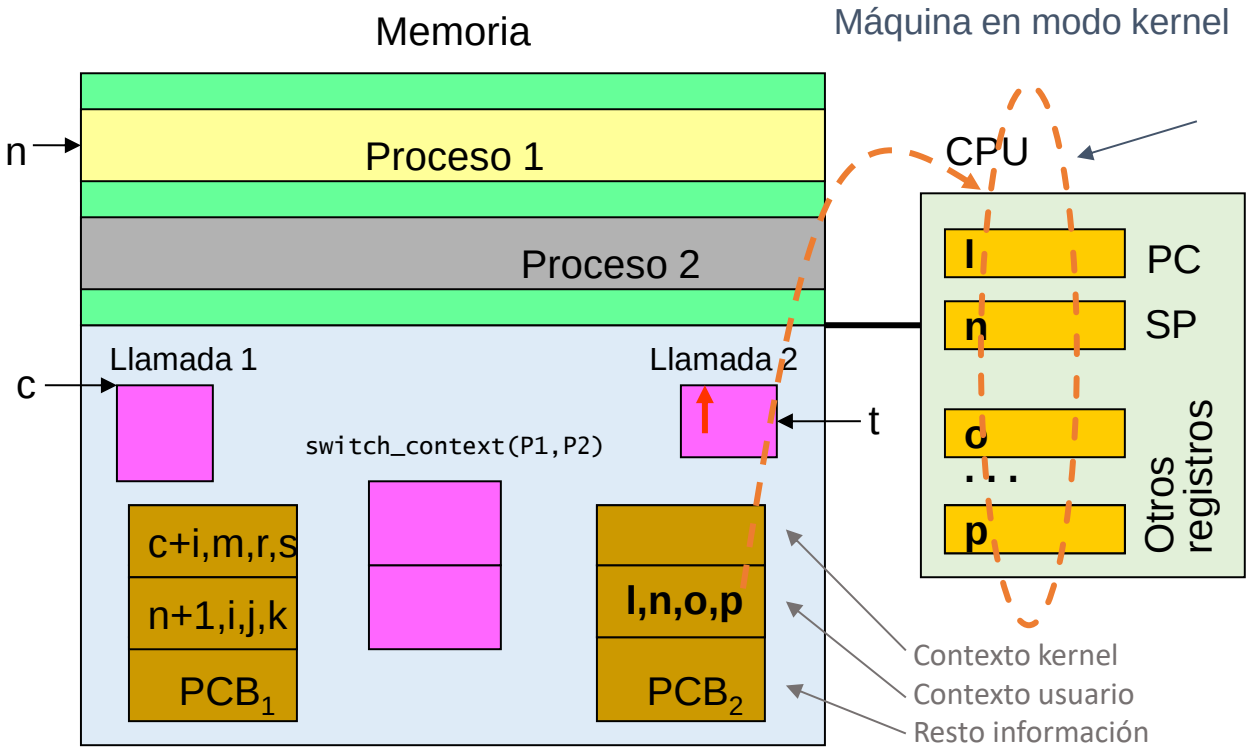
Autor: José Antonio Gómez, UGR, reproducido con permiso

Cambio de contexto



El kernel termina en el entorno de P2 la primitiva que bloqueó al proceso. Ahora tiene que restaurar el estado del procesador con los datos de usuario de P2

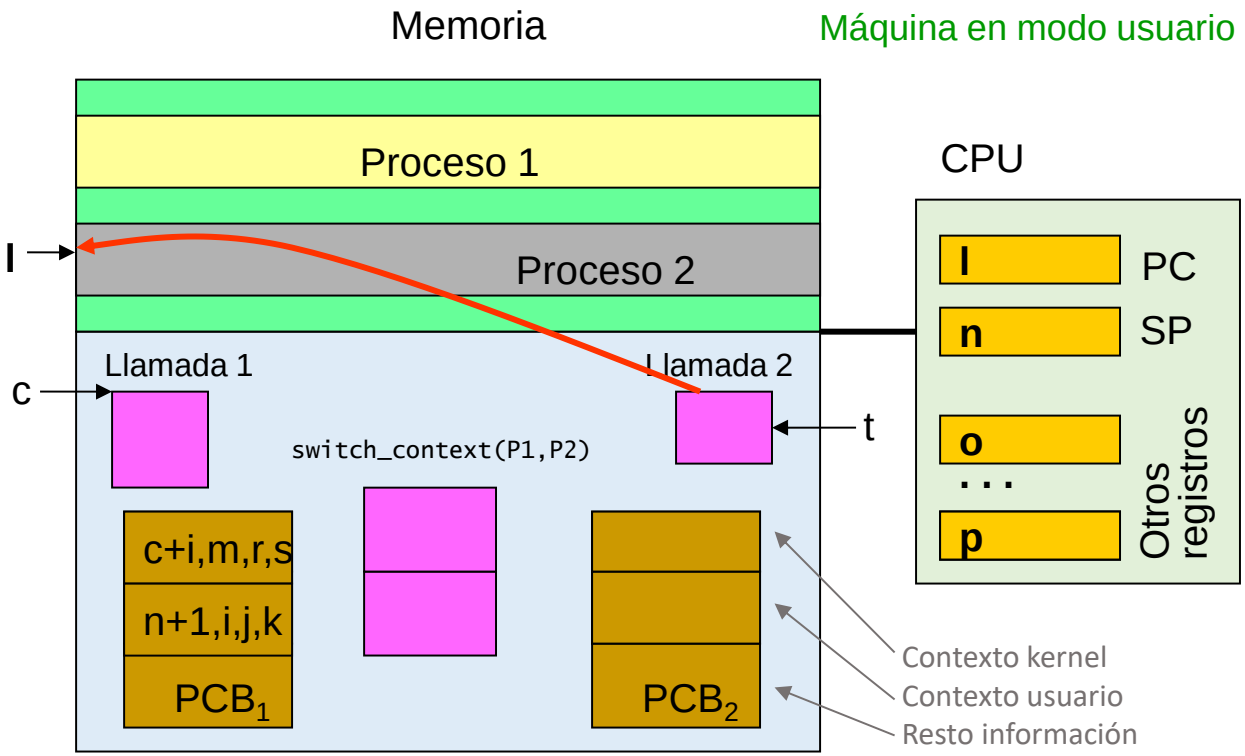
Cambio de contexto



El kernel repone el estado del procesador para P2

Autor: José Antonio Gómez, UGR, reproducido con permiso

Cambio de contexto



Finalmente, P2 recupera la ejecución en la instrucción I, la siguiente a aquella que lo bloqueó

Autor: José Antonio Gómez, UGR, reproducido con permiso

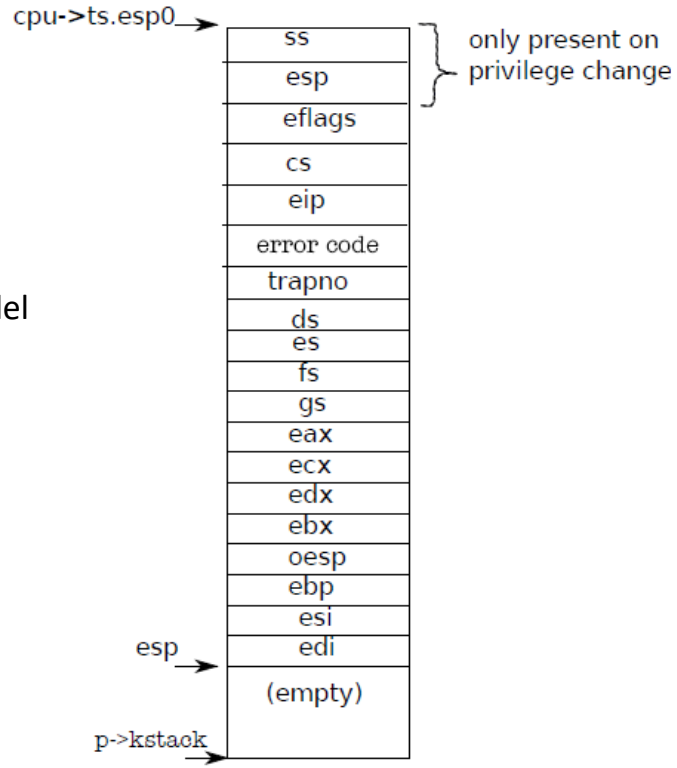
Cambio de contexto

Trapframe en xv6

En xv6 se define una estructura *trapframe* que sirve para almacenar el estado del procesador, tanto en un cambio de contexto como en una interrupción. Es uno de los campos de la estructura de tipo `proc`, el BCP de esta implementación.

Puede encontrarse una descripción muy detallada del cambio de contexto en xv 6 en la siguiente URL:

https://medium.com/@ppan.brian/context-switch-from-xv6-aedcb1246cd



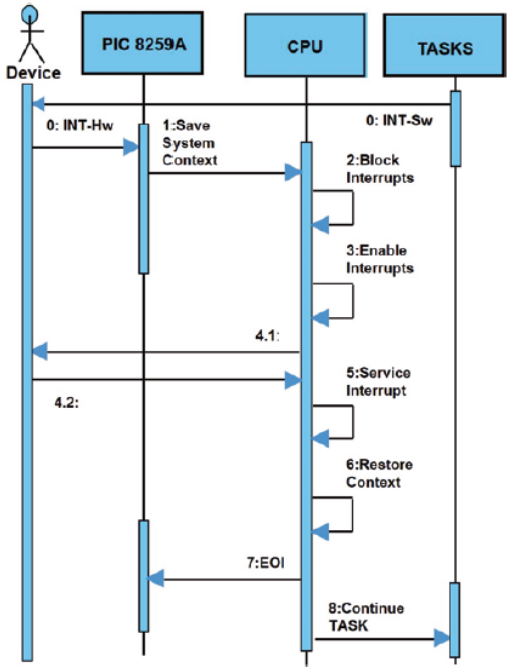
Interrupción

Las interrupciones pueden tener origen en el hardware o ser consecuencia de la ejecución de una instrucción de código máquina especial (INT en x86). Toda interrupción hace que el microprocesador salte a una dirección de programa especial donde está la rutina única de atención, que forma parte del kernel. Esta rutina tiene que salvar el contexto del proceso actual en el stack de usuario, reconocer cual es el origen de la interrupción por un número que esta envía por hardware y con este seleccionar en una tabla de memoria donde está el código para tratar esa interrupción específica. Al terminar debe restaurar el contexto del proceso y devolver el control.

Esta es una descripción muy simple. Las interrupciones se pueden anidar y enmascarar. Lo habitual es que la atención se divida en dos partes, como en Linux. En la primera (top half), el kernel enmascara todas las interrupciones y reconoce la actual. A continuación programa un tasklet, una forma especial de código que se encola en el propio kernel para su ejecución posterior (bottom half). La mayor parte del código de la rutina de atención está en el tasklet que sí puede interrumpirse.

Si no se hiciese de esta forma y todo fuera en una secuencia única, dejarían de atenderse otras interrupciones, y podría ser catastrófico para el sistema.

Interrupción



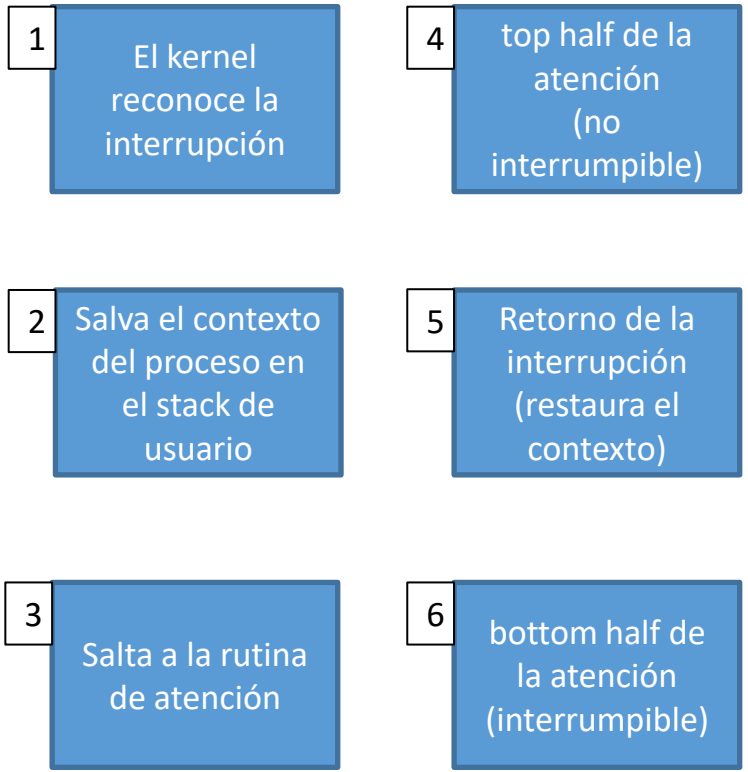
Actores que intervienen en una interrupción en la arquitectura Intel x86:

- Dispositivo HW que genera la interrupción.
- Circuito PIC8259A. Este circuito (o varios de ellos conectados en cascada) recibe las peticiones de interrupción y arbitra (selecciona) la más prioritaria si hay varias simultáneas. Activa la petición hacia la CPU.
- La CPU salva contexto, bloquea todas las interrupciones, reconoce la actual (4.1, 4.2), obtiene el vector de interrupción y desbloquea las interrupciones.
- A continuación ejecuta la rutina de interrupción y al terminar envía la señal EOI al 8259ª para indicarlo.
- Restaura contexto

Mejía-Álvarez et al: “Interrupt handling schemes in Operating Systems”, 2018, Springer, p.9

Interrupción

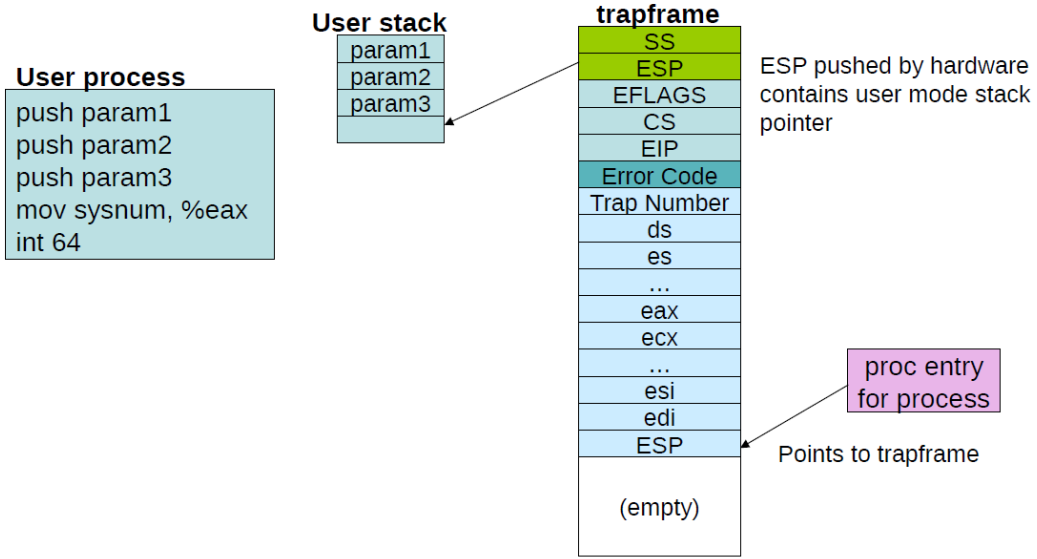
Esquema de atención de las interrupciones en Linux. La separación en dos partes es una técnica avanzada para evitar largos segmentos de código no interrumpible. Los detalles exceden del contenido de este curso pero puede ampliarse la información en el siguiente paper:



I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues,

Interrupción

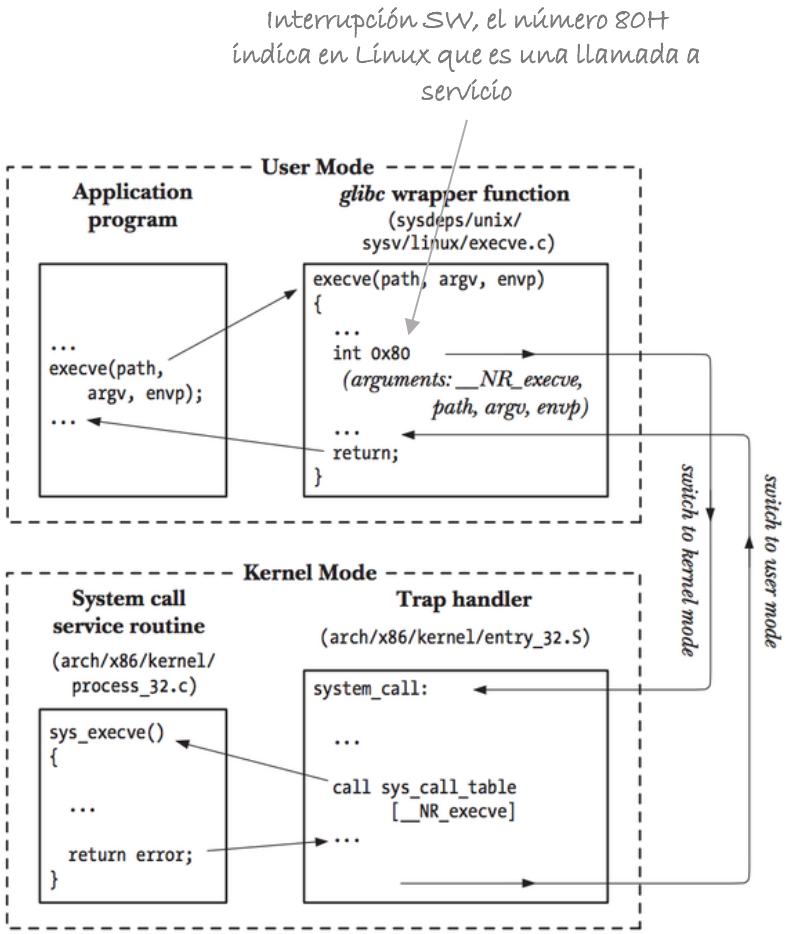
La interrupción software sirve para invocar los servicios del Sistema Operativo. Los parámetros y el resultado se intercambian por el stack de usuario del proceso. En Linux, si la primitiva tiene menos de 6 parámetros, se pasan por valor en los registros %eax, %ebx, %ecx,, %esi, %edi, %ebp. En %eax se escribe el código de operación para que el sistema sepa qué primitiva se demanda. Si hay 6 o más parámetros, se pasa un puntero como referencia a la memoria en la que residen los valores. El resultado se devuelve también en %eax.



Interrupción

Ejemplo de invocación de llamada en Unix. El proceso de usuario utiliza la función `excve` de la librería `glibc` para crear un proceso. El programador ni siquiera tiene que ser consciente de que está haciendo una llamada al SO, ve una función convencional.

La función `execve`, prepara los datos en el stack e invoca la interrupción software. La rutina de atención del kernel identifica que el origen es SW, con el contenido de `%eax` sabe que es una llamada a `execve` y lanza el código correspondiente. Devuelve el resultado en `%eax` y esto es lo que recibe el proceso de usuario como resultado de la función que invocó.



Finalización

El proceso init es el único que no puede invocar exit, si muere es un kernel panic [crash del Sistema]

Cerrar todos los procesos que estén abiertos aunque el programa no lo haya hecho

Decrementa los contadores de inodo [lección 5]

Despierta al proceso padre si está en un wait() a la espera de que el hijo termine

Se marca el proceso como RUNNABLE (Listo)

Se marca como ZOMBIE e invoca al scheduler. No debería volver si la invocación falla es un kernel panic

Código de exit() en xv6
Fichero proc.c

```
void exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

Finalización

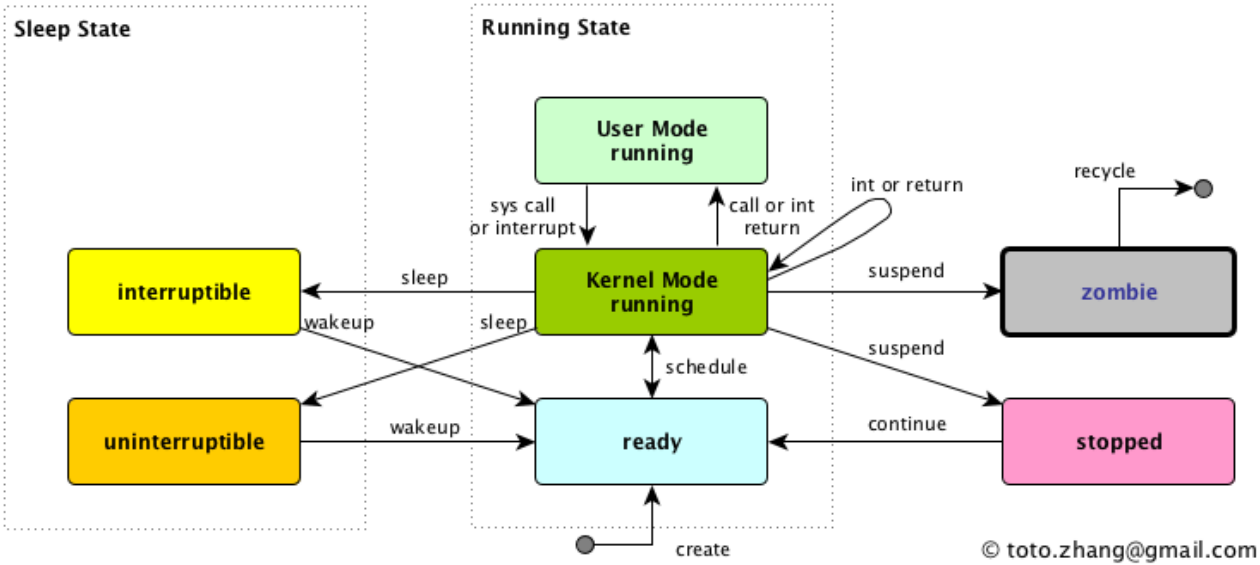


Diagrama de estados y transiciones en Linux.

https://totozhang.github.io/2016-01-16-linux-zombieprocess/

Procesos especiales

Un **proceso servidor** es aquel que está en un bucle infinito pendiente de recibir peticiones de los usuarios y atenderlas de forma atómica. Ejemplos: servidor web, servidor de base de datos.

Un **demonio** es un proceso que se arranca en el inicio del sistema y es inmortal, o con más precisión reencarnante porque el sistema lo arranca de nuevo si termina de forma anormal. Pueden ser a su vez servidores y se ejecutan en background, por lo que no tienen una terminal de usuario asociada.

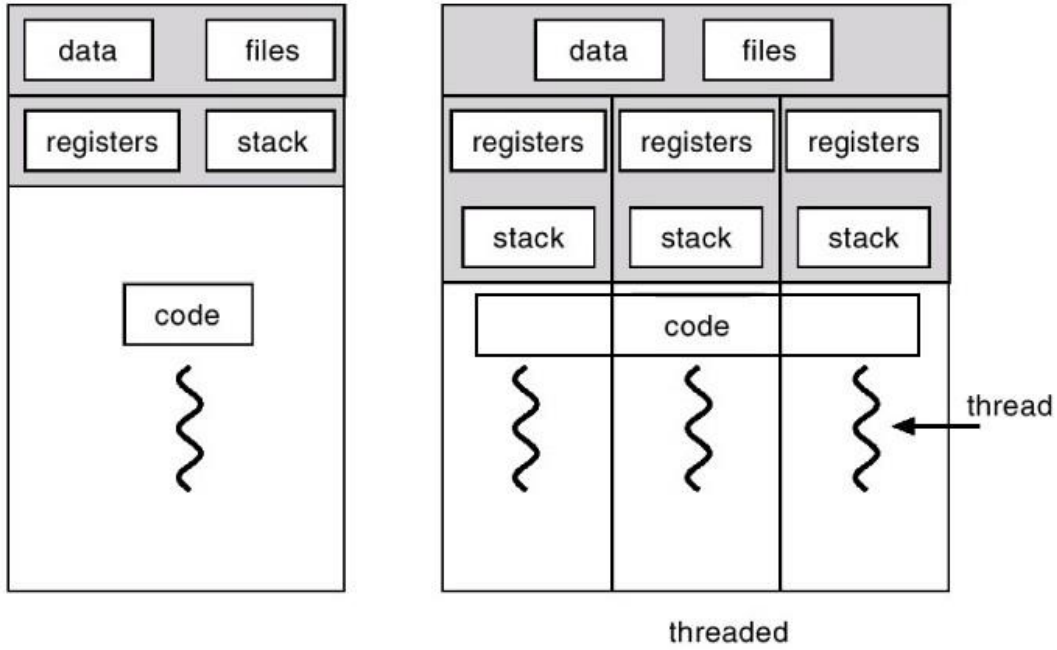
Existen **procesos de sistema** que se ejecutan en modo privilegiado, como el proceso nulo, el demonio de paginación, el de caché de ficheros, etc().

- Estos procesos se crean con un servicio especial durante el arranque.
- Tienen acceso a todo el mapa de memoria.
- Son flujos de ejecución independiente dentro del Sistema Operativo.

No hay que confundir un proceso de sistema con los que lanza el superusuario. A pesar de su nombre, este solo puede ejecutar procesos en modo usuario, no en modo kernel.

Threads

Los *threads* (hilos de ejecución), también llamados de forma impropia procesos ligeros, nacieron para aprovechar las arquitecturas multiprocesador. En un proceso convencional o *monothread* hay un único hilo de ejecución, determinado por el valor del PC y del resto de registros.



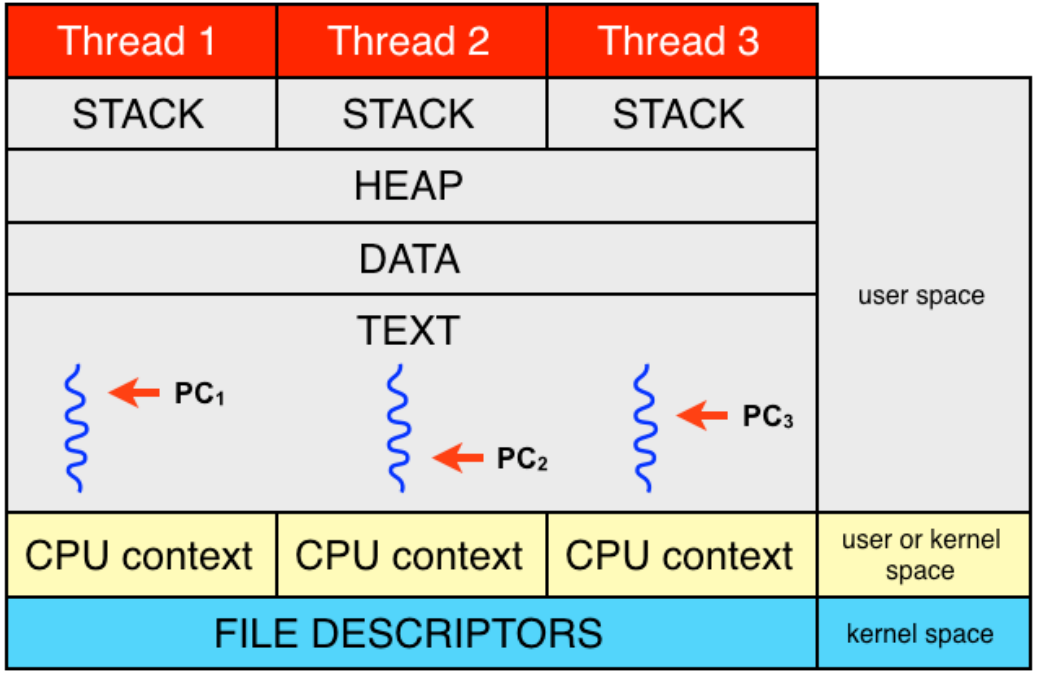
Threads

Los **threads** de un mismo proceso comparten código, datos (*heap* incluido) y recursos del Sistema Operativo (por ejemplo, ficheros abiertos) y tienen su propio contador de programa, registros y *stack*. Al acceder a los mismos datos puede haber conflictos, pero también aparecen nuevas posibilidades de programación concurrente. Por ejemplo, en un procesador de textos un *thread* puede estar calculando la repaginación de todo el documento cuando el usuario incluye una nueva imagen en otro hilo que se ocupa de su presentación en pantalla.

Una ventaja de la programación *multithread* frente a la multiproceso es que un cambio de contexto de *thread* es muy rápido comparado con el de un proceso. Dado que comparten datos y código no hay que salvar más que el contexto del *thread*, en una tabla similar a la de BCPs de la que también es responsable el kernel. No hay que recargar la tabla de páginas de memoria dinámica puesto que es la misma.

Además, si el procesador es multicore, como todos los actuales de gama media o alta, cada uno de ellos puede estar ejecutando un *thread* del mismo proceso. Esta es la razón por la que la programación concurrente con *threads* ha ganado tanta popularidad.

Threads

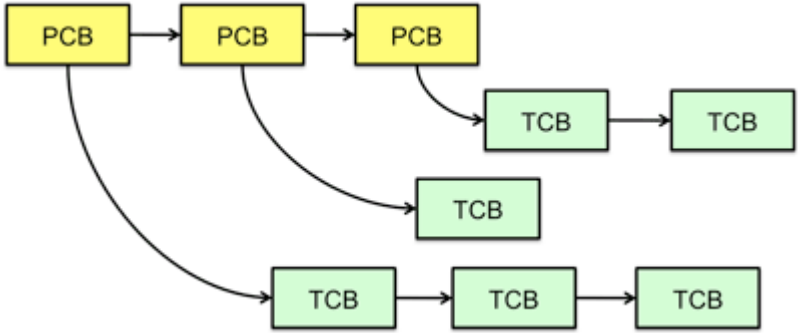


En este esquema se pueden apreciar muy bien los recursos que comparten los hilos de un mismo proceso y cuáles son privativos.

Threads

La planificación de threads es la base de los *schedulers* actuales como se describe en esta misma lección. Cada thread tiene su *Thread Control Block*, que es la versión reducida del PCB.

El planificador utiliza la lista de threads y sus estados para realizar su función. Este es el esquema de Windows, BSD otros sistemas Unix descendientes de la rama Solaris.



Linux los *threads* como procesos convencionales, asignando a cada hilo un `task_struct`, si son del mismo proceso comparten dato.

[Threads en Solaris, BSD y Linux]

Planificación

La **planificación del procesador** es una de las misiones más importantes del Sistema Operativo y consiste en decidir a qué proceso se le concede la CPU y cuándo debe expulsarse el que la está ocupando.

Tradicionalmente se distinguen tres tipos de planificación a largo, medio y corto plazo. La **planificación a largo** plazo sirve para decidir en qué orden se ejecutan las tareas en un gran sistema, con mucha componente batch, en función de las características estadísticas de sus tiempos de ejecución conocidas de antemano. Pensemos en un centro de proceso de datos de una empresa de servicios (electricidad, gas, telefonía) que por la noche lanza procesos muy costosos en tiempo de emisión de recibos o liquidación de impuestos.

La **planificación a medio plazo** sirve para decidir qué procesos van a bloquearse**. La planificación a corto**, que es la que vamos a ver con detalle es la que decide a qué proceso preparado para ejecución se le concede la CPU. Aunque la palabra en inglés *scheduling* estrictamente se refiera a la planificación a medio plazo, su uso se ha generalizado para todo el proceso de planificación. El procedimiento para cargar un nuevo proceso en la CPU se llama *dispatching*.

En esta lección nos limitaremos a la **planificación con un solo núcleo.**

Planificación

Estados del proceso

El diagrama de estados puede variar de un Sistema Operativo a otro, aunque todos son parecidos. Este sigue la estructura de Unix.

- **Nuevo**. Estado especial con el que el Sistema Operativo marca un proceso desde que se inicia su carga hasta que está terminado para ejecutarse por primera vez. Al completarse la inicialización transita al estado **Listo.**
- **Listo**. Proceso que está preparado para ejecutarse. En esta cola se encuentran todos los procesos a la espera de que el Sistema Operativo les conceda CPU.
- **En Ejecución**. Estado del único proceso que ocupa la CPU, al que transita desde Listo cuando el Sistema Operativo lo planifica. Puede salir de él de tres formas. Por finalización voluntaria del proceso [`exit()`], porque invoca un servicio bloqueante (p. ej. `scanf()`) o si el sistema es expulsivo porque expropiación de la CPU.
- **Bloqueado**. El proceso está en una cola con otros procesos durmientes a la espera de que se complete la operación bloqueante que han invocado. Cuando suceda vuelven a **Listo**.
- **Zombie**. Cuando un proceso termina no se borra de inmediato su BCP, para que el proceso padre pueda completar una llamada a `wait()`. Es un estado sin retorno y breve. Desde cualquier estado se puede llegar a zombie si se recibe una señal.

Planificación

KPIs

La planificación es un problema de optimización y como en todos ellos hay una serie de indicadores de rendimiento (KPI) que pueden entrar en conflicto. Nos referiremos a procesos pero todo lo dicho es válido también para threads.

- **Tiempo de ejecución (t)**. Tiempo que necesita un proceso para completarse si ocupase la CPU al 100%**.**
- **Tiempo de espera (E)**. Tiempo que pasa el proceso en el estado Listo, preparado para ejecutarse pero sin que le concedan la CPU.
- **Tiempo de respuesta (Trespuesta).** Tiempo desde que el proceso está Listo hasta que responde a una petición de usuario.
- **Penalización (P)**. Cociente entre el tiempo de respuesta y el tiempo de ejecución

No todos los procesos tienen los mismos requisitos. Un proceso intensivo en uso de CPU (CPU bounded) como un reproductor de vídeo necesita un tiempo de ejecución previsible. Un proceso a ráfagas de usuario (user bounded) como un navegador web o un videojuego necesita un tiempo de respuesta muy pequeño (latencia).

Planificación

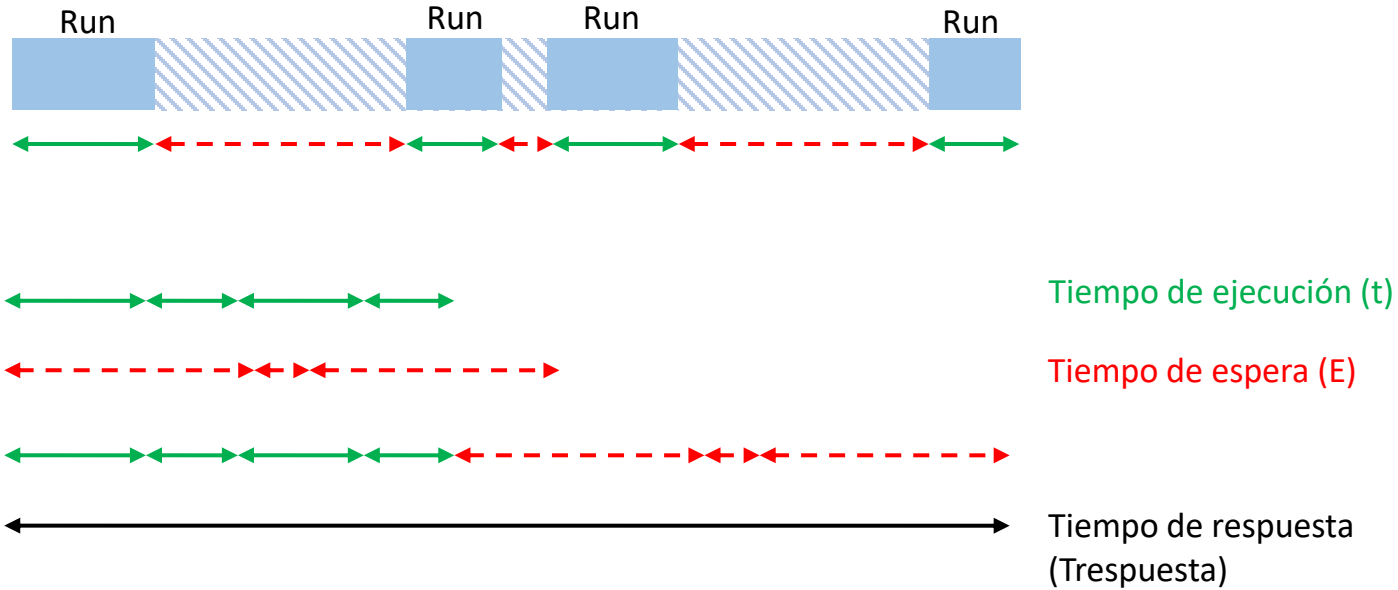


Ilustración de la relación entre los distintos tiempos usados en planificación. La penalización es el cociente entre el tiempo de respuesta y el tiempo de ejecución

Planificación

El planificador se despierta cuando un proceso nace, muere o pasa a bloqueo y siempre que se activa una interrupción periódica llamada **tick**, una vez cada 10ms (puede modificarse por configuración).

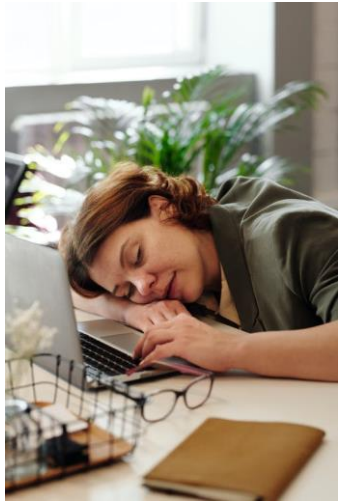
Esta interrupción es la basa para medir otro periodo llamado **quantum** (en Linux *timeslice*), que es él tiempo máximo que un proceso puede ejecutarse de forma continua habiendo otros en la cola de espera del estado Listo. El quantum es múltiplo del tick [entre 10 y 200 en Linux] y garantiza que un único proceso no monopoliza la CPU. Si se produce un bucle infinito, como consecuencia de un error, el proceso es expulsado a **Listo** cada vez que agota el quantum. En MS-DOS la única solución era reiniciar la máquina en esas circunstancias.

En los sistemas no expulsivos, el proceso completa su quantum con independencia de los que estén esperando en cola. En los **expulsivos**, el proceso puede ser expulsado aunque no hay terminado su quantum si hay a la espera otro de mayor prioridad.

Un proceso puede ejecutarse de forma continua múltiples quantum si no hay otros a la espera. Todos los sistemas tienen un **proceso nulo**, que se ejecuta si no hay ningún proceso en la cola **Listo**.

Planificación

Algoritmo FCFS (First Come, First Served)



Es un algoritmo muy simple, el scheduler concede la CPU al proceso que lleva más tiempo en la cola Listo. El proceso se ejecuta de forma continuada hasta terminar. No hay quantum.

Tiene la ventaja de resultar muy sencillo de implementar, pero es muy poco equitativo. Si en un mismo servidor hay un proceso de cálculo numérico con un largo tiempo de ejecución y un servidor web que atiende peticiones de usuarios remotos, éstas pueden demorar minutos si tienen la mala de suerte de encolarse tras el primer proceso.

El tiempo de respuesta percibido por el usuario que hace la petición web es inaceptable, mientras que el que lanzó el proceso numérico tendría una buena experiencia. Sin embargo, podría haberse interrumpido el cálculo durante dos segundos para atender al usuario web, que mejoraría mucho su percepción, mientras que para el segundo usuario se prolongaría de forma imperceptible una operación que necesita minutos.

FCFS

En un Sistema Operativo que usa planificador FCFS se presenta la siguiente combinación de procesos:

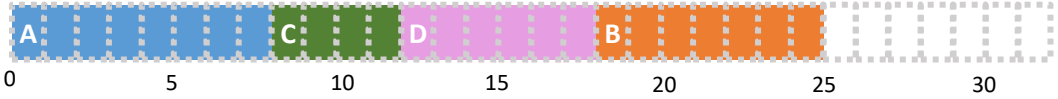
Proceso	Instante de llegada	Tiempo de ejecución
A	0	8
B	7	7
C	3	4
D	6	6

El cambio de contexto es instantáneo y sin coste. Utilizando las tablas siguientes escribir la secuencia de ejecución y rellenar los datos solicitados.

	T	CPU	Llega	Cola
	0	A<8>	A	
	3	A<5>	C	C<4>
	6	A<2>	D	C<4>,D<6>
	7	A<1>	B	C<4>,D<6>,B<7>
Fin A	8	C<4>		D<6>,B<7>
Fin C	12	D<6>		B<7>
Fin D	18	B<7>		
Fin B	25			

Como el proceso A tiene un tiempo de ejecución largo, los procesos A, B y C se ve bastante penalizados

Proceso	Llegada	Trun	Ini	Fin	Trespuesta	Tespera	Penalización
A	0	8	0	8	8	0	1
B	7	7	18	25	18	11	18/7
C	3	4	8	12	9	5	9/4
D	6	6	12	18	12	6	2



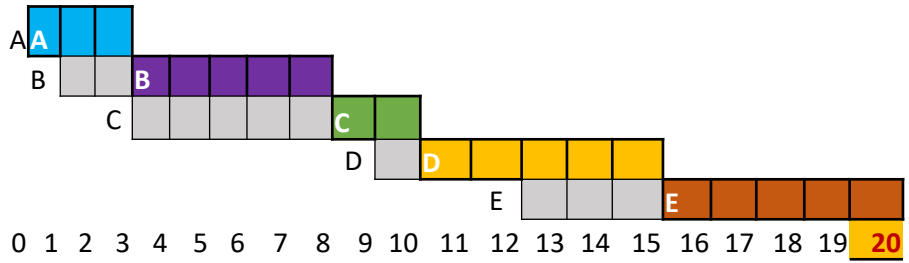
Ocupación de la CPU

FCFS

En un Sistema Operativo que usa planificador FCFS se presenta la siguiente combinación de procesos:

Proceso	Instante de llegada	tiempo de ejecución	Inicio	Fin	T. respuesta	T. espera	Penalización
A	0	3					
B	1	5					
C	3	2					
D	9	5					
E	12	5					

T	CPU	Llega	Cola	Finaliza
0	A<3>	A		



Ocupación de la CPU

Planificación

Round Robin

El algoritmo Round Robin es una mejora sobre el FCFS puro. Un proceso solo puede estar como máximo ejecutándose durante un múltiplo de ticks llamado **quantum**. Si llega a ese límite, el Sistema Operativo lo expulsa de la CPU, lo pone al final de la cola de procesos en estado **Listo** y concede a la CPU al proceso que lleve más tiempo en espera. Esto puede implementarse de forma sencilla con una cola FIFO de procesos en estado **Listo**.

La elección del quantum es importante. Si es muy grande, RR tiende a parecerse a FCFS. Si es muy pequeño favorece la equidad en el reparto de la CPU pero cada cambio de contexto supone una penalización por lo que el rendimiento global se vería perjudicado.

Round Robin es como la cola de espera de una montaña rusa. Al terminar el viaje (quantum) los pasajeros son expulsados del tren y si quieren volver a viajar tienen que situarse al final de la cola y esperar su turno.

Round Robin

Resolver el mismo problema planteado para FCFS usando Round Robin

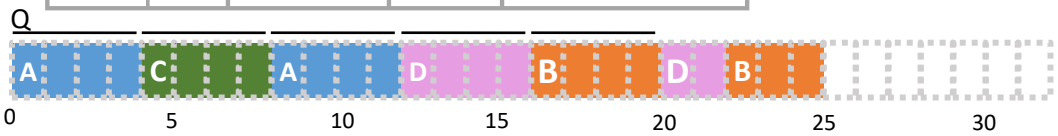
Proceso	Instante de llegada	Tiempo de ejecución
A	0	8
B	7	7
C	3	4
D	6	6

El cambio de contexto es instantáneo y sin coste, el valor del quantum es 4 y no hay prioridades.

Con Round Robin la penalización se reparte de forma más equitativa

	T	CPU	Llega	Cola
	0	A<8>	A	
	3	A<5>	C	C<4>
A exp	4	C<4>		A<4>
	6	C<2>	D	A<4>,D<6>
	7	C<1>	B	A<4>,D<6>,B<7>
Fin C	8	A<4>		D<6>,B<7>
Fin A	12	D<6>		B<7>
D exp	16	B<7>		D<2>
B exp	20	D<2>		B<3>
Fin D	22	B<3>		
Fin B	25			

Proceso	Llegada	Trun	Ini	Fin	Trespuesta	Tespera	Penalización
A	0	8	0	12	12	4	12/8
B	7	7	16	25	18	11	18/7
C	3	4	4	8	5	1	5/4
D	6	6	12	22	16	10	10/6



Ocupación de la CPU

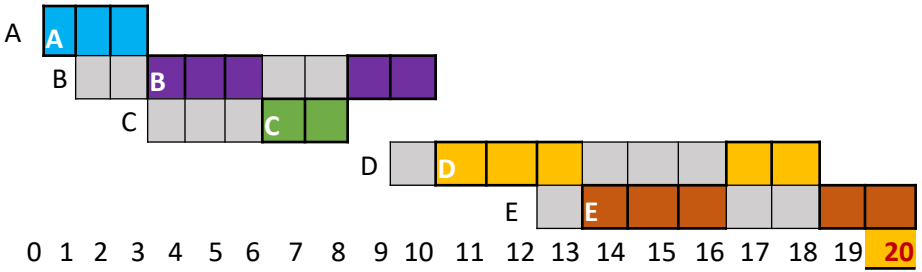
Round Robin

Resolved el problema planteado para FCFS con planificador Round Robin con Quantum(3) :

Proceso	Instante de llegada	tiempo de ejecución
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

Inicio	Fin	T. respuesta	T. espera	Penalización

T	CPU	Llega	Cola	Finaliza
0	A<3>	A		



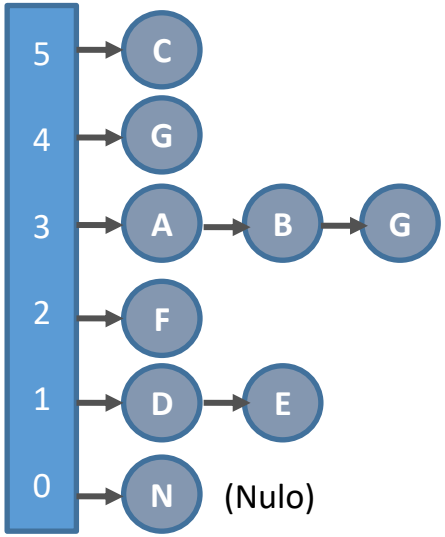
Ocupación de la CPU

Planificación

Planificación con prioridad

Los dos sistemas descritos suponen que todos los procesos descritos tienen la misma prioridad pero la realidad no es así. La mayor parte de los Sistemas Operativos son capaces de manejar prioridades para atender antes a un proceso crítico que a otro que puede esperar sin provocar una degradación. Esta característica es más acentuada en los SSOO de tiempo real pero tanto Windows (abre el Administrador de tareas y lo descubrirás) como la familia Unix (comando `nice`) manejan colas de prioridades.

La idea es que en lugar de una cola de espera hay varias en función de la prioridad y no se atiende ningún proceso de prioridad reducida mientras haya otro de prioridad más alta pendiente de ejecutar. Volviendo al ejemplo de la montaña rusa, hay una cola de baja prioridad donde los usuarios tienen largo tiempo de espera mientras que los de la priority queue se atienden de inmediato.



Planificación

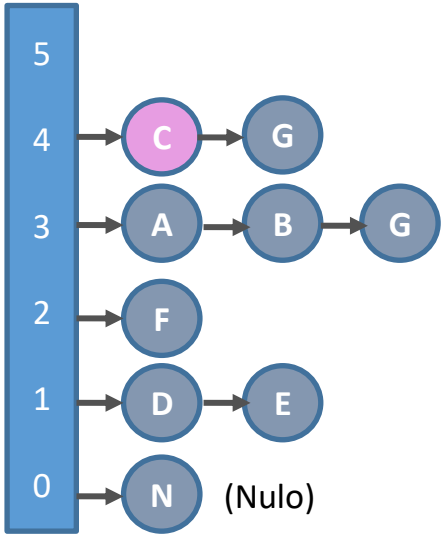
Planificación con prioridad

La gestión por prioridad conlleva el riesgo de ***starvation***, que toda la CPU sea consumida por procesos de prioridad máxima y el resto no pueda entrar casi nunca.

Para ello se usan las prioridades dinámicas. Si un proceso ha consumido n quantum, el SO rebaja de forma automática su prioridad. También puede incrementarse la prioridad de los que lleven determinado tiempo en cola. En los sistemas de tiempo real el programador puede modificar la prioridad en zonas críticas del código. Las colas pueden gestionarse por FCFS o Round Robin, siendo este último lo más habitual y puede haber distintos valores de quantum según sea la prioridad. La ejecución de los procesos de mayor prioridad se facilita con quantum más prolongados.

Si la prioridad es expulsiva se quita la CPU al proceso en curso en cuanto haya uno más prioritario aunque no haya consumido su quantum.

El SO ha rebajado la prioridad del proceso C dando a G la oportunidad de ejecutarse



Round Robin no expulsivo con prioridad

En un Sistema Operativo que usa planificador Round Robin no expulsivo con dos prioridades estáticas se presenta la siguiente combinación de procesos:

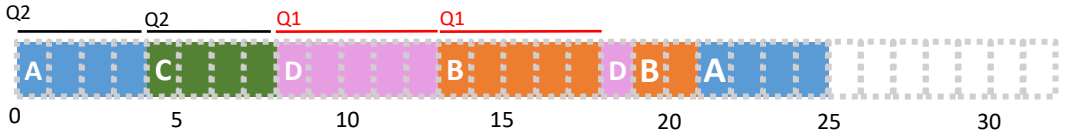
Proceso	Instante de llegada	Tiempo de ejecución	Prioridad
A	0	8	2
B	7	7	1
C	3	4	2
D	6	6	1

El quantum para la prioridad 1 es 5 y el quantum para la prioridad 2 es 4. El cambio de contexto es instantáneo y sin coste. Utilizando las tablas siguientes escribir la secuencia de ejecución y rellenar los datos solicitados.

	T	CPU	Llega	Colas
	0	A<8>	A	
	3	A<5>	C	2 C<4>
Exp A	4	C<4>		2 A <4>
	6	C<2>	D	1 D <6> 2 A <4>
	7	C<1>	B	1 D<6>,B<7> 2 A<4>
Fin C	8	D <6>		1 B<7> 2 A<4>
Exp D	13	B<7>		1 D<1> 2 A<4>
Exp B	18	D<1>		1 B<2> 2 A<4>
Fin D	19	B<2>		2 A<4>
Fin B	21	A<4>		
Fin A	25			

El proceso A se ve muy perjudicado por la ejecución de los de nivel 1

Proceso	Llegada	Trun	Ini	Fin	Trespuesta	Tespera	Penalización
A	0	8	0	25	25	17	25/8
B	7	7	13	21	14	7	2
C	3	4	4	8	5	1	5/4
D	6	6	8	19	13	7	13/6

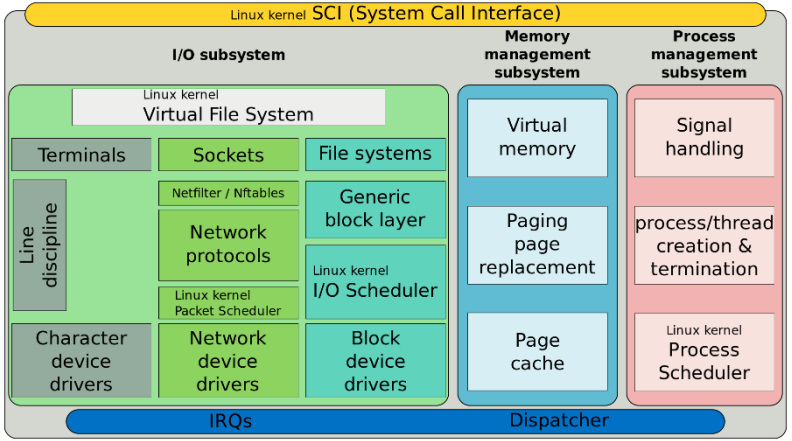


Ocupación de la CPU

Planificación

Linux

Todos los algoritmos descritos son teóricos, la realidad es bastante más complicada y los planificadores de los sistemas son un elemento crítico para su rendimiento. Tomando como ejemplo Linux, el mismo planificador tiene que servir para garantizar el rendimiento de un gran servidor web donde la mayoría del tráfico es interacción de usuario, el de un supercomputador, con procesos intensivos en el uso de CPU, o el de un ordenador personal en el que la mayor parte de los recursos los consume la interfaz gráfica y hay una variedad enorme de procesos.



Reconciliar todas estas necesidades ha hecho que el planificador de Linux se haya rediseñado dos veces desde cero.

Planificación

Linux O(1)

El planificador O(1) utilizaba distintas colas de prioridad, separadas en dos clases:

- Tiempo real: prioridades de 0 a 99
- Procesos normales: prioridades de 100 a 139

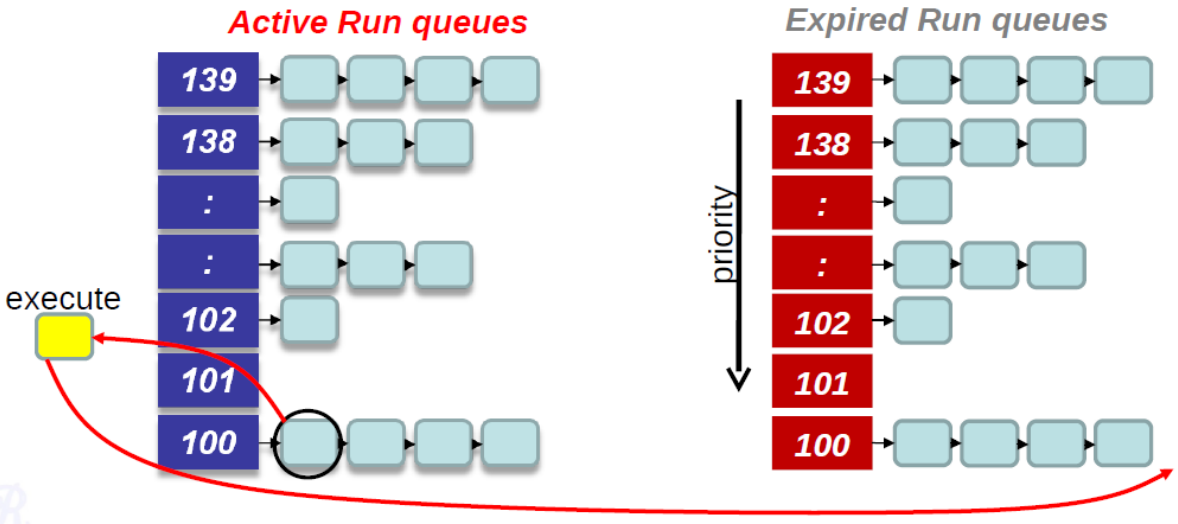
La prioridad base de un proceso es 120, estos valores pueden verse y modificarse con el comando `nice` [niceness es sinónimo de prioridad en Linux].

Las prioridades van en orden descendente 0 es la mayor y 139 la menor. O(1) mane dos conjuntos de colas, la de procesos activos y procesos ejecutados (expired). Cuando un proceso consume su parte de CPU se pasa a la cola de procesos (expired). Los procesos nuevos se añaden a la tabla de activos. Cuando no hay ningún proceso pendiente en activos se invierten los papeles, la tabla de expired pasa a ser de activos y viceversa. Este es el mecanismo para evitar la starvation.

O(1) gestionaba la prioridad con un método sencillo. En función del tiempo en el que el proceso permanece bloqueado es fácil predecir si es I/O bounded (mucho tiempo) o CPU bounded (poco tiempo). Con esta cuenta, O(1) disimuía la niceness de un proceso con un bonus de valor entre 0 y 10 subiendo así la prioridad de los procesos I/O bounded.

Planificación

Linux O(1)



El quantum también estaba ligado a la prioridad, y variaba entre 5 ms para los procesos de prioridad mínima a 800 para los de prioridad máxima.

Planificación

Linux CFS

Desde la versión 2.6.23 usa el **Completely Fair Scheduler** (**CFS**), un algoritmo con prioridades que en lugar de diferentes colas por prioridad utiliza un árbol binario. Esta estructura se explica en Matemática Discreta y cuya implementación en C se verá en la asignatura de Algoritmos.

El concepto de Fair Scheduling es sencillo, si en un sistema tenemos N procesos en un tiempo t cada uno de ellos debería ejecutarse t/N. Esta es la teoría, pero la realización práctica no en sencilla. ¿Cuál es el tiempo mínimo en el que debemos aplicar la política? Si t es grande, penalizamos el tiempo de respuesta de los procesos interactivos, si es muy pequeño penalizamos el rendimiento general al disparar el número de cambios de contexto.

Además, ¿cómo manejaríamos las prioridades? La idea del *fair scheduling* presupone que todos los procesos tienen la misma prioridad. No es difícil extenderla suponiendo que si un proceso consume una unidad de t/N, otro de prioridad triple debería consumir 3t/N, pero tendríamos que estar recalculando continuamente tiempos y prioridades en cada cambio de contexto.

Planificación

Linux CFS

El CFS se basa en dos ideas brillantes, usar un árbol con todos procesos en estado Listo y un **tiempo de ejecución virtual** que transcurre a velocidades diferentes en función de las prioridades. De esta manera, se simplifica la estructura de control y se eliminan los problemas de **starvation**. Las prioridades aprovechan el valor de *niceness* que ya se introdujo en O(1).

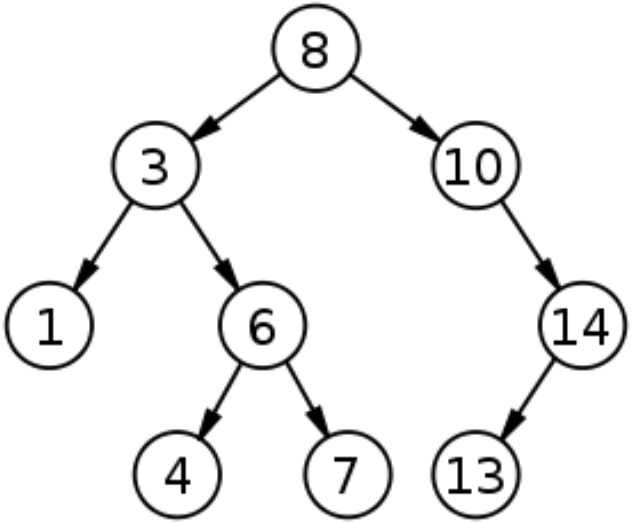
Cada vez que transcurre un *tick*, el *scheduler* actualiza el árbol. Para ello calcula el tiempo virtual transcurrido por proceso. Por simplificar, un tick puede suponer 8 unidades de tiempo virtual para un proceso de mínima prioridad y solo 1 para otro de máxima. Entre dos *ticks* solo puede variar el valor del proceso en ejecución porque el resto están fuera de la CPU a la espera de recibir turno. CFS es capaz de manejar prioridades dinámicas con lo que ese factor de multiplicación puede variar en el transcurso de la ejecución.

A la hora de decidir a qué proceso se le concede la CPU el criterio es muy sencillo, siempre al nodo de valor mínimo. Para evitar recorrer el árbol, CFS mantiene siempre un puntero a ese nodo. CFS no maneja *quantum,* el proceso correspondiente al nodo mínimo tendrá la CPU hasta superar el valor de otro nodo.

Planificación

Linux CFS

Un árbol binario es una estructura acíclica ordenada de grado 2 como máximo. La primera característica indica que no hay ningún recorrido en el que pueda pasarse más de una vez por un mismo nodo. El grado es el número de hijos que cada nodo puede tener y la ordenación se consigue porque el nodo de la izquierda tiene siempre valor menor que el del padre y el de la derecha mayor.

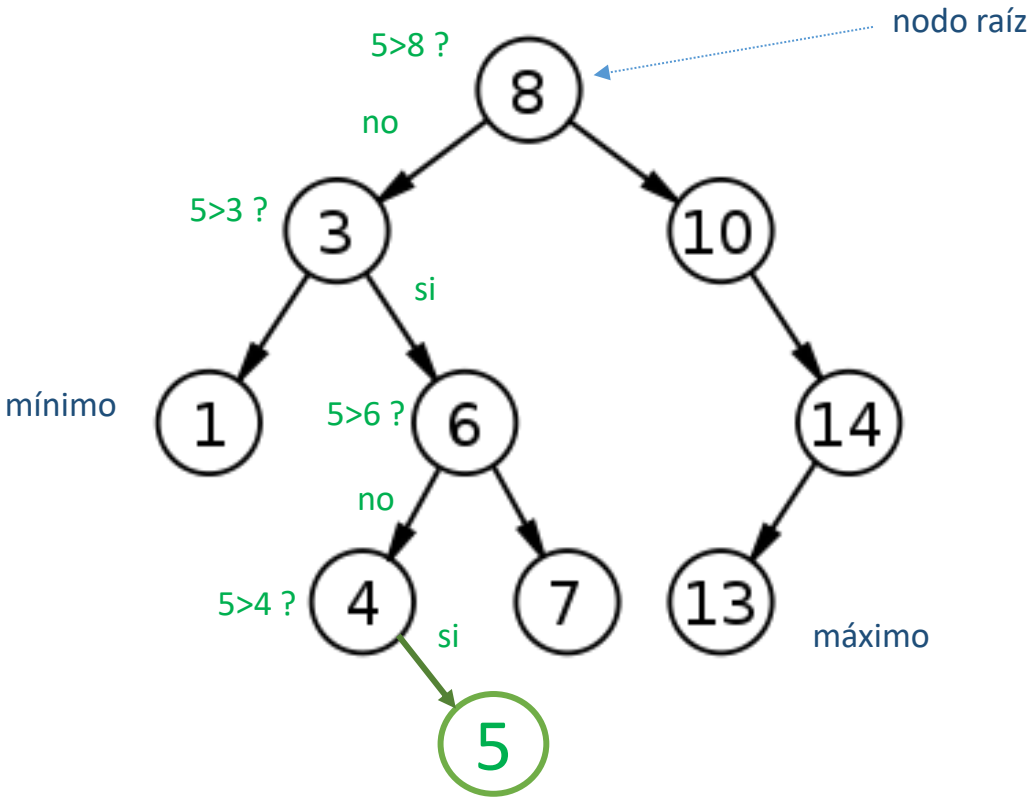


Un árbol binario tiene también la propiedad de que la rama más larga tendrá como mucho el doble de longitud que la más corta, lo que junto a la ordenación, hace que sea muy rápido recorrerlo.

En https://es.wikipedia.org/wiki/Árbol_binario se describe su implementación en C

Planificación

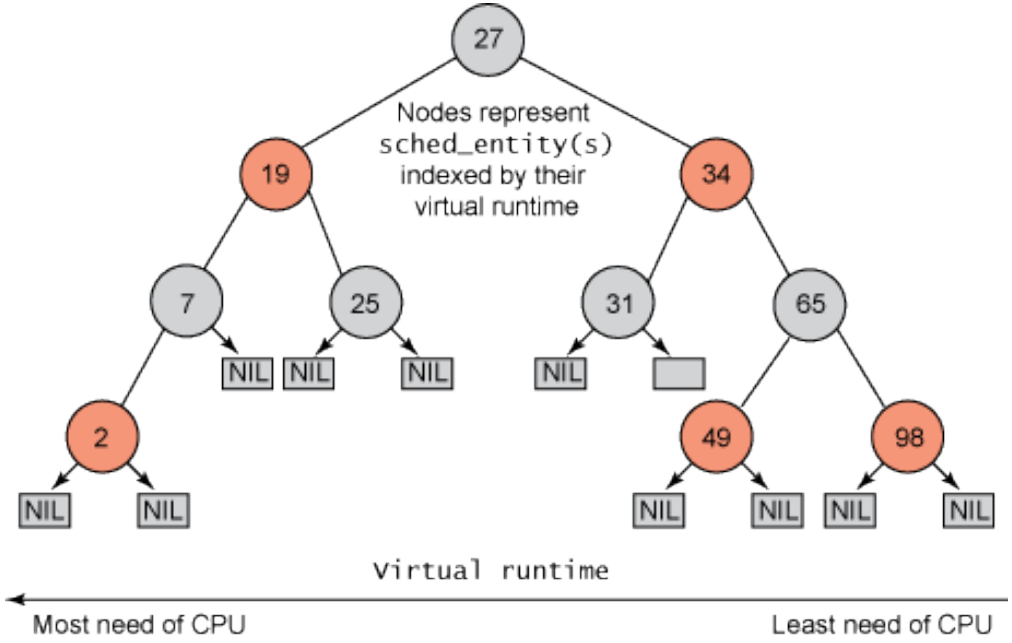
Linux CFS



Planificación

Linux CFS

Cada nodo contiene el valor del *virtual runtime* del proceso



Fuente: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

Planificación

Linux CFS

Con CFS se eliminan los heurísticos [forma elegante de llamar a las excepciones o trucos] para favorecer a los procesos I/O bounded. Si un proceso es de este tipo se ejecuta por ráafgas y tendrá un valor pequeño de *vruntime* por lo que tenderá a estar en la izquierda del árbol, con altas probabilidades de ejecución.

Por otra parte, cada vez que se crea un proceso, se añade al árbol con el valor mínimo de *vruntime*. Esto facilita que procesos dinámicos de vida muy corta, como la atención en un servidor web de una petición GET, tengan alta prioridad de ejecución, mejorando la responsividad del sistema.

Algunos usuarios se quejan de que CFS beneficia las aplicaciones tipo servidor porque los vendedores de HW pagan por optimizar esta vertiente mientras que no se invierte lo mismo para las aplicaciones de usuario que requieren altas prestaciones como los videojuegos. En cualquier caso, la historia demuestra que la evolución ha sido constante en Linux y CFS será eventualmente sustituido por otro scheduler.

Hisoria de los schedulers de Linux

Planificación

BSD (Berkeley Software Distribution)

Como comparación veamos cómo es el *scheduler* de la otra gran familia, BSD, descendiente del Unix original.

El algoritmo actual se llama ULE (no es un acrónimo, es un juego de palabras con el nombre del fuente `sched_ule.c`). Maneja dos colas separadas para procesos interactivos y procesos batch, la primera tiene prioridad absoluta sobre la segunda. A su vez, en cada clase hay distintas colas por prioridad. Se selecciona el proceso (o thread) de la cola de mayor prioridad y se gestiona por Round Robin.

ULE separa el *scheduling* en dos módulos, el de bajo nivel se ejecuta con cada tick y determina si el proceso actual debe ser expulsado y cuál ocupará la CPU. El de alto nivel entra a ejecutarse con una frecuencia mucho menor y se encarga de la gestión de las prioridades dinámicas.

ULE es muy eficaz para sistemas con mucha interactividad, como los equipos de usuario (mac OS es un derivado de FreeBSD).

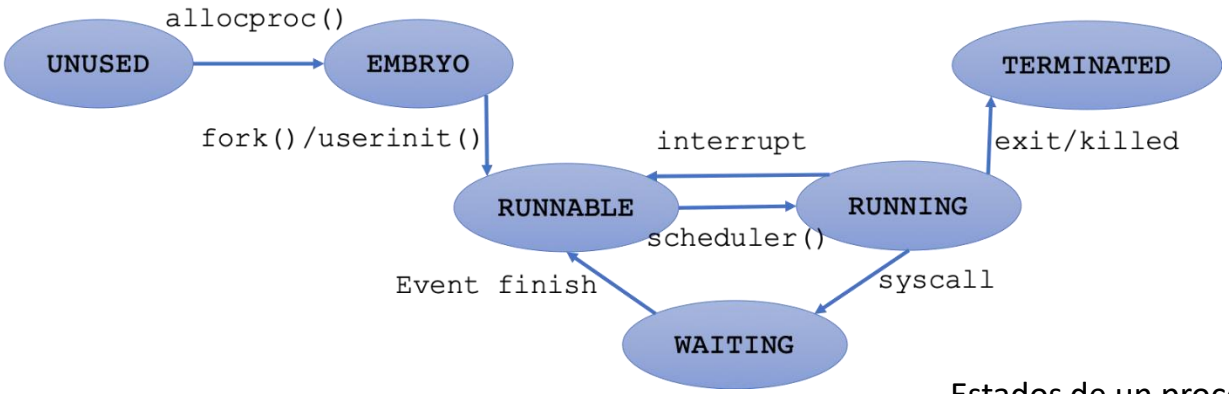
Información detallada en https://web.cs.ucdavis.edu/~roper/ecs150/ULE.pdf

Planificación

xv6

Como implementación muy simple de un sistema tipo Unix, el scheduler de xv6 no se acerca a los niveles de sofisticación de CFS o ULE.

La planificación se hace mediante una cola única de procesos, sin prioridades, con Round Robin y un quantum único de 100 ms.



Estados de un proceso en xv6

Planificación

Threads

Cuando se programa con threads hay dos niveles superpuestos de paralelismo el del proceso y el del hilo (thread). La gestión difiere si los threads se ejecutan en el espacio de usuario o en el de sistema. En el primero (p. ej. threads en Java), el scheduler es parte de la librería de threads y transparente al Sistema Operativo, se ejecutará cuando el proceso en sí tenga la CPU. No entraremos en su descripción, aunque una característica notable es que por su propia naturaleza la planificación en este nivel es no expulsiva y el thread se ejecutará de principio a fin.

Si los threads son de sistema, los threads se planifican como si fueran procesos, tal y como hemos descrito hasta ahora por simplificar. De hecho, **tanto Unix como Windows usan el thread como unidad de planificación**. La diferencia es que el cambio de contexto entre threads es mucho más rápido que entre procesos. Si el proceso es monothread tendrá solo una entrada en el planificador, si tiene varios, una por cada uno. El estado de los threads de un mismo proceso no tiene por qué coincidir, puede haber hilos en bloqueo y otros listos para ejecución.