Procesos e hilos

T2

Sistemas Operativos

Iván Valero Rodríguez – 2º Grado Ingeniería Informática

Procesos e hilos

Contenido

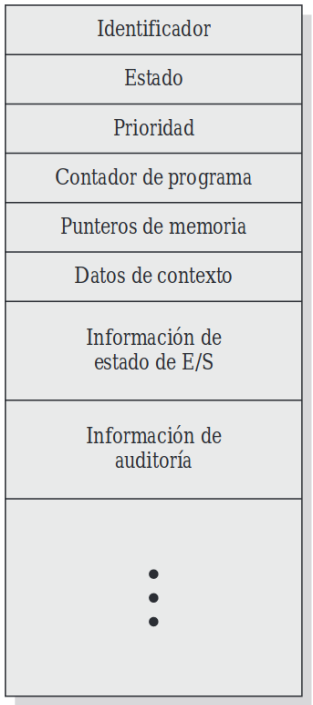
A. Generalidades sobre Procesos, Hilos y Planificación

1 Procesos

Un programa es una unidad inactiva almacenada en el disco, el proceso sería una instancia del mismo programa. Con lo cual establecemos la siguiente definición de proceso:
Una unidad de actividad cargada en memoria principal y caracterizada por un solo hilo secuencial de ejecución, un estado actual y un conjunto de recursos del sistema asociados.

2 Bloque de Control de Procesos / PCB

En cualquier instante de tiempo, mientras el proceso está en ejecución, este se caracteriza por un conjunto de elementos llamado contexto de ejecución.



El contexto de ejecución comprende los siguientes datos:
Identificación: Un identificador único asociado al proceso para distinguirlo del resto.
Estado: Ejecución, bloqueado, Listo, suspendido, terminado o zombi.
Información de planificación: Nivel de prioridad relativo al resto de procesos.
Descripción de los segmentos de memoria asignados al proceso: Espacio de memoria asignado en RAM.
Punteros a memoria: Incluye los punteros al código del programa y datos asociados a dicho proceso, además de otros datos como si el proceso utiliza memoria virtual.
Datos de contexto: Datos presentes en los registros del procesador necesarios para continuar la ejecución del proceso cuando el planificador lo decida.
Información de estado E/S y recursos asignados: Peticiones de E/S pendientes y los dispositivos E/S asignados a dicho proceso.
Comunicación entre procesos: Se guarda cualquier dato que tenga que ver con la comunicación de procesos independientes.
Información de auditoria: Incluye la cantidad de tiempo de procesador o ciclos utilizados, así como cuanto le queda por ejecutar.
El contexto de ejecución se almacena en el Bloque de control de Proceso o PCB.
El PCB permite que, tras interrumpir un proceso se pueda restaurar su estado con la información que contiene. Por tanto, es la herramienta clave que permite al sistema operativo dar soporte a múltiples procesos y a la multiprogramación.
Cada vez que el planificador acepta un proceso, se crea un PCB y conforme la ejecución del proceso avanza, el PCB se va actualizando.
En sistemas GNU, cada proceso tiene asociado un PCB en una lista simplemente enlazada a la que llamamos tabla de procesos.
Se llama imagen del proceso al conjunto de códigos, datos, pila, montículo y PCB de un proceso, dicha imagen debe estar cargada en memoria principal para posibilitar la ejecución.

3 Dispatcher

El programa que permite la multiprogramación, alternando entre distintos procesos según la política de planificación del sistema se llama Dispatcher.
Este se ejecuta en las alarmas de temporización (cuanto de tiempo) o interrupciones de otro tipo, así como operaciones de E/S.
El planificador le dice al Dispatcher que proceso introduce en la CPU. Gracias al BCP el Dispatcher introduce un proceso nuevo o hace que otro continúe por donde se quedó (salvado y restaurando el contexto).

4 Creación y terminación de procesos

La creación se produce por 4 eventos distintos.

- Arranque del sistema
- Ejecución de una llamada al sistema para creación de procesos, como fork() mediante llamadas POSIX.
- Petición de usuario para crear un proceso (el clásico doble click para abrir un programa)
- Inicio de un trabajo por lotes, que se utiliza en clusters grandes, donde los usuarios envían trabajos al sistema de forma remota.

Una vez que el sistema operativo decide, por cualquier motivo de los anteriores , crear un proceso procederá de la siguiente manera:
1. **Asignar un identificador de proceso único al proceso.** En este instante, se añade una nueva entrada a la tabla primaria de procesos, que contiene una entrada por proceso.
2. **Reservar espacio para proceso**. Esto incluye todos los elementos de la imagen del proceso. Para ello, el sistema operativo debe conocer cuánta memoria se requiere para el espacio de direcciones privado (programas y datos) y para la pila de usuario. Estos valores se pueden asignar por defecto basándonos en el tipo de proceso, o pueden fijarse en base a la solicitud de creación del trabajo remitido por el usuario. Si un proceso es creado por otro proceso, el proceso padre puede pasar los parámetros requeridos por el sistema operativo como parte de la solicitud de la creación de proceso. Si existe una parte del espacio de direcciones compartido por este nuevo proceso, se fijan los enlaces apropiados. Por último, se debe reservar el espacio para el bloque de control de proceso (BCP).
3. **Inicialización del bloque de control de proceso**. La parte de identificación de proceso del BCP contiene el identificador del proceso así como otros posibles identificadores, tal como el indicador del proceso padre. En la información de estado de proceso del BCP, habitualmente se inicializa con la mayoría de entradas a 0, excepto el contador de programa (fijado en el punto entrada del programa) y los punteros de pila de sistema (fijados para definir los límites de la pila del proceso). La parte de información de control de procesos se inicializa en base a los valores por omisión, considerando también los atributos que han sido solicitados para este proceso. Por ejemplo, el estado del proceso se puede inicializar normalmente a Listo o Listo/Suspendido. La prioridad se puede fijar, por defecto, a la prioridad más baja, a menos que una solicitud explicita la eleve a una prioridad mayor. Inicialmente, el proceso no debe poseer ningún recurso (dispositivos de E/S, ficheros) a menos que exista una indicación explícita de ello o que haya sido heredados del padre.

4. **Establecer los enlaces apropiados**. Por ejemplo, si el sistema operativo mantiene cada cola del planificador como una lista enlazada, el nuevo proceso debe situarse en la cola de Listos o en la cola de Listos/Suspendidos.
5. **Creación o expansión de otras estructuras de datos**. Por ejemplo, el sistema operativo puede mantener un registro de auditoría por cada proceso que se puede utilizar posteriormente a efectos de facturación y/o de análisis de rendimiento del sistema.

Los procesos terminan a partir de los siguientes eventos que pueden ser o no voluntarios.
a. **Salida normal (voluntaria)**: El trabajo ha concluido y ejecuta un exit() o el usuario a hecho click en el botón X, lo que también invoca un exit().
b. **Salida por error (voluntaria)**: El proceso descubre un error contemplado por el propio programador que quiso que el programa se cerrase al producirse dicho error.
c. **Error Fatal (involuntario):** Puede producirse por una ejecución de instrucción ilegal, referencia a una zona de memoria inexistente, división por cero, etc…
d. **Eliminado por otro proceso (involuntario):** Algún proceso ejecuta una llamada al sistema que elimina otro proceso, por ejemplo la llamada al sistema kill().

5 Suspensión de procesos, swapping y el modelo de los 5 + 2 estados.

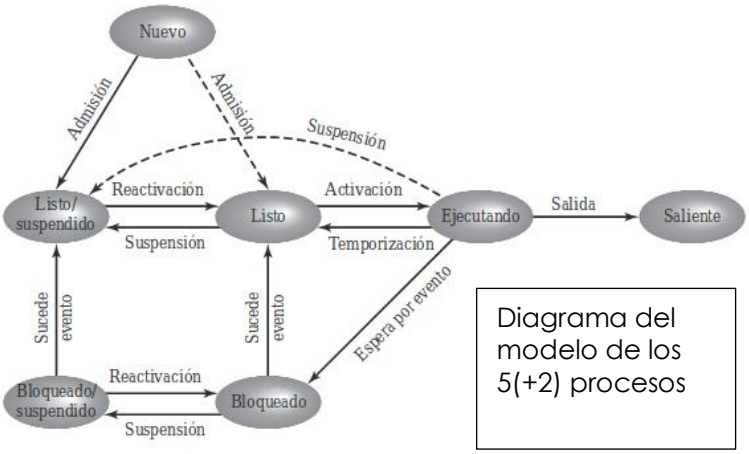


Diagrama del modelo de los 5(+2) procesos

La gestión básica de un proceso se basa en estos 5 estados.
Ejecutando: Proceso actualmente en ejecución.
Listo: El proceso se ejecutara cuando tenga la oportunidad y el planificador lo decida
Bloqueado: La ejecución de un proceso bloqueado no es posible hasta que no se cumpla un evento determinado.
Nuevo: Proceso recién creado, tiene BCP pero no esta cargado en memoria principal.
Saliente: Proceso que ha sido expulsado del grupo de procesos ejecutables.

Swapping: (del inglés "*swap*", que significa intercambiar) es mover un proceso o parte de él temporalmente desde la memoria principal a un dispositivo secundario de almacenamiento (memoria de apoyo) para luego devolverlo a la memoria principal. El propósito de esta técnica es que el sistema operativo sea capaz de asignar más memoria de la que tiene físicamente. De esta forma, en el caso que algún proceso requiera de memoria y ya esté toda asignada o no haya suficiente, el kernel se preocupa de intercambiar páginas de memoria desde y hacia la memoria de apoyo para hacerle espacio.

Con el concepto del *swapping* aparecen dos nuevos estados:
Bloqueado/Suspendido. El proceso está en almacenamiento secundario y esperando un evento.
Listo/Suspendido.El proceso está en almacenamiento secundario pero está disponible para su ejecución tan pronto como sea cargado en memoria principal.

5.1 Transición entre estados

Null->Nuevo: Se crea un nuevo proceso para ejecutar un programa.

Nuevo -> Listo: El proceso está preparado para su ejecución. Muchos sistemas operativos limitan los procesos en estado LISTO para evitar problemas de rendimiento.
Listo->Ejecutando: Se selecciona un proceso listo para ejecutar, esta tarea la realiza el planificador.
Ejecutando->Saliente: El proceso actual en ejecución se finaliza por la razón que sea.
Ejecutando->Listo: Una razón por la que esto pasa es que el proceso en ejecución haya alcanzado su tiempo de ejecución de forma ininterrumpida (interrupción por alarma de reloj). Esto puede ocurrir por otros motivos como la prioridad de ejecución.
Ejecutando->Bloqueado: El proceso se bloquea porque ha solicitado algo por lo que debe esperar.
Bloqueado->Listo: Cuando sucede el evento por el que esperaba, pasa al estado listo.
Listo->Saliente: Se termina el proceso en ejecución por alguna razón (ejemplo: ejecución de kill()).
Bloqueado->Saliente: Lo mismo que en el caso anterior.
Bloqueado/Suspendido -> Listo/Suspendido: Sucede un evento al que estaba esperando.
Listo/Suspendido ->Listo: Cuando no hay más procesos listos en memoria principal, o si el SO determina que es más prioritario un proceso suspendido, el sistema operativo necesitará traer uno para continuar la ejecución.
Listo ->Listo/Suspendido: A pesar de que no se suele bloquear un proceso Listo, puede ser necesario suspender un proceso Listo si con ello se consigue liberar un bloque suficientemente grande de memoria.

6 Hilos (hebras)

Un hilo (proceso ligero) es una unidad básica de utilización de la CPU. Si un proceso tiene múltiples hilos, puede realizar más de una tarea a la vez.
Cada hilo tiene:

- Un estado de ejecución por hilo (Ejecutando, listo, bloqueado, etc…)
- Un contexto o "BCP" de hilo igual que con los procesos.
- Una pila de ejecución
- Un espacio de almacenamiento para variables locales.
- Acceso a la memoria y recursos de su proceso, compartido con todos los hilos de su mismo proceso.

En un entorno multihilo, hay un único BCP y un espacio de direcciones asociado al proceso, pero ahora tenemos varias pilas separadas para cada hilo, y un bloque de control para cada hilo que contienen la información relativa de cada uno.
Todos los hilos de un proceso comparten el estado y los recursos de ese proceso, residen en el mismo espacio de direcciones y tienen acceso a los mismos datos.
Si un hilo cambia un dato en memoria, los otros hilos lo ven, si un hilo abre un archivo de solo lectura, el resto de hilos también lo pueden hacer, etc…

6.1 Estados de los hilos

No tiene sentido aplicar estado de suspensión a un hilo, ya que es un estado a nivel de proceso. Suspender un proceso significa expulsar su imagen de memoria principal para dejar hueco a la imagen de otro proceso, como todos los hilos de un proceso comparten la misma imagen, todos los hilos se suspenden. Lo mismo ocurre al terminar un proceso.

Dicho esto, tenemos los siguientes estados:

Creación: Al crear un proceso, se crea un hilo de dicho proceso, posteriormente ese hilo puede crear otro. El nuevo hilo tiene su propio registro de conexto y espacio de pila, y se coloca a la cola de listos.
Bloqueo: El hilo necesita esperar por un evento.
Desbloqueo: Cuando sucede el evento por el que el hilo espera este se desbloquea.
Finalización: El hilo se completa liberando su registro de contexto y pilas.

6.2 Motivación y ventajas

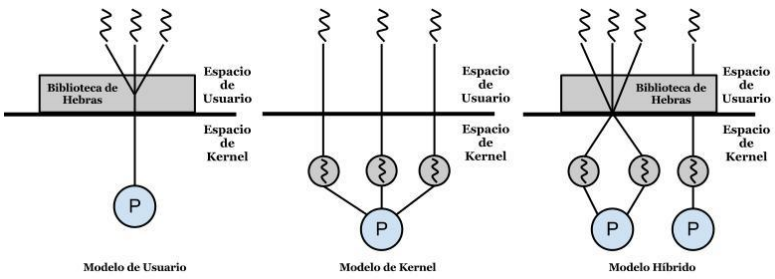
Las ventajas de la programación multihilo se puede dividir en cuatro categorías principales.

Capacidad de respuesta: Permite que un programa continúe ejecutándose incluso si parte del está bloqueado o falla.
Compartición de recursos: Las hebras comparten la memoria y los recursos del proceso al que pertenecen.
Economía y tiempo de ejecución: Consume más tiempo crear y gestionar procesos que hilos.
- *Crear o terminar una hebra*: Al terminar un proceso se elimina su BCP y sus estructuras de datos y espacio de memoria, al acabar un hilo se termina su contextoy pila, igual al crearlos.
- *Cambio de contexto entre hebras*: Al cambiar de contexto el SO genera un "overhead" es el tempo desperciado en el cambio, que en el caso de los hilos es mucho menor.
- *Comunicación entre hebras*: Los datos están inmediatamente habilitados ya que los hilos comparten memoria. Sin necesidad de la invocación al núcleo que necesitarían los procesos para comunicarse.
Utilización sobre arquitecturas multiprocesador: Las ventajas de usar multihilos se benefician de la arquitectura multiprocesador donde los hilos se ejecutan en paralelo en cada procesador.

6.3 Enfoques sobre las hebras

6.3.1 Hebras usuario

La hebras son manejadas en espacio de usuario, y el sistema operativo no es consciente de ellas, por ello asigna solo una hebra de kernel al proceso, analoga a la tabla de procesos. Esto permite gran velocidad, pero con el peligro de que el sistema operativo bloquee el proceso completo si una hebra se bloquea y no e ntrega el control a otra. Al no existir interrupción de reloj no se podrá ejecutar otra hebra hasta que alguna deje de procesar voluntariamente. La conmutación entre hebras es varios órdenes de magnitud más rápida que la de procesos.



6.3.2 Hebras núcleo/kernel

El sistema operativo controla la hebras y por ende asigna una hebra de kernel a cada hebra por proceso, esto permite que el S.O. puede cambiar el control entre hebras y evitar el bloqueo de procesos, esto requiere una tabla de hebras analoga a la tabla de procesos y no se necesita sistema de gestión o tablas de hebras dentro de cada proceso. La gestión de hebras se hace a través de llamadas al sistema, aunque implica un mayor coste de operación

6.3.3 Enfoque híbrido

Aquí existen variados modelos que buscan combinar las ventajas de cada modelo anterior reduciendo sus desventajas. Utiliza hebras de tipo usuario pero multiplexadas sobre hebras tipo núcleo, donde estas ejecutan un conjunto de hebras que turnan su utilizacion

7 Planificación de procesos.

7.1 El proceso nulo

Cuando el planificador a corto plazo no encuentra procesos al ser invocado se introduce lo que se conoce como proceso nulo. Este es un proceso que siempre está listo para ejecutarse y que tiene la prioridad más baja.

7.2 Tipos de planificadores

Largo plazo: procesos por lotes
Corto plazo o *scheduler*
Medio plazo – gestor de memoria. Tipos de planificación:

- Planificación apropiativo: al proceso actual se le puede retirar la CPU
- Planificación no apropiativa: al proceso actual no se le puede retirar la CPU

7.3 Políticas de planificación

FCFS o FIFO (First Came First Served/ First In, First Out): El primero en llegar es el primero en recibir la CPU. Es no apropiativo. Favorece a los procesos largos frente a los cortos.
SJF(Shortest Job First): cuando el procesador queda liberado, selecciona el proceso que requiere un tiempo de servicio menor. Es no apropiativo y en caso de que haya dos procesos en igualdad de condiciones se aplica FCFS
SRTF(Shortest Remaining Time First): cada vez que entra un proceso a la cola de preparados se comprueba si su tiempo de servicio es menor que el tiempo de servicio que le resta al que está en ejecución. Si es menor, se cambia el proceso, y si hay dos en igualdad de condiciones se aplica FIFO. Se trata de una política apropiativa.
Planificación por prioridades. Asociamos a cada proceso una prioridad (entero menor=mayor prioridad). Se le asigna la CPU al proceso más prioritario. Existe tanto de manera apropiativa como no apropiativa. En caso de igualdad se emplea FIFO.
Round-Robin (planificación por turnos): se basa en el quantum (trozo de tiempo) que da y retira la CPU. Es una planificación apropiativa basada en FCFS.
Colas multiples: a cada proceso se le asigna una cola. Cada cola se planifica de manera independiente. Se hace necesario una planificación entre colas.
Colas multiples con realimentación: a cada proceso se le asigna una cola inicial pero posteriormente se le asignará otra en función de sus necesidades. Cada cola tiene su

propia planificación y es necesaria una planificación entre colas.
CFS(Completely Fair Scheduler): no asigna rodajas de tiempo sino una proporción del procesador depediente de la carga del sistema. Cada procesa se ejecuta durante un tiempo proporcional a su peso dividido por la suma total de pesos.

$$T_{ni} = \left(\frac{Peso\ P_i}{\Sigma Pesos\ P_j} \right) \times P$$

*/*Con P=sched_latency si n>nr.latency o min_granuralidad x en otro caso.*/*

7.4 Planificación en multiprocesadores

Tres aspectos interrelacionados:
Asignación de procesos a procesadores
- Cola dedicada para cada procesador
- Cola global para todos los procesadores

Uso de multiprogramación en cada procesador individual
Activación del proceso
Planificación de procesos → igual que en monoprocesadores pero teniendo en cuenta:
- Número de CPUs
- Asignación/Liberación proceso-procesador

Planificación de hilos→ permiten explotar el paralelismo real dentro de una aplicación

7.5 Planificación de hilos en multiprocesadores

1) Compartición de carga
- Cola global de hilos preparados
- Cuando un procesador está ocioso, se selecciona un hilo de la cola (método muy usado)

2) Planificación en pandilla
–Se planifica un conjunto de hilos afines (de un mismo proceso) para ejecutarse sobre un conjunto de procesadores al mismo tiempo (relación 1 a 1)
–Útil para aplicaciones cuyo rendimiento se degrada mucho cuando alguna parte no puede ejecutarse (los hilos necesitan sincronizarse)

3) Asignación de procesador dedicado
–Cuando se planifica una aplicación, se asigna un procesador a cada uno de sus hilos hasta que termine la aplicación
–Algunos procesadores pueden estar ociosos → No hay multiprogramación de procesadores

4) Planificación dinámica
–La aplicación permite que varíe dinámicamente el número de hilos de un proceso
–El SO ajusta la carga para mejorar la utilización de los procesadores

7.6 Sistemas de tiempo real

●La exactitud del sistema no depende sólo del resultado lógico de un cálculo sino también del instante en que se produzca el resultado.
●Las tareas o procesos intentan controlar o reaccionar ante sucesos que se producen en "tiempo real" (eventos) y que tienen lugar en el mundo exterior.
Características de las tareas de tiempo real:

➜Tarea de tº real duro: debe cumplir su plazo límite

➜Tarea de tº real suave: tiene un tiempo límite pero no es obligatorio

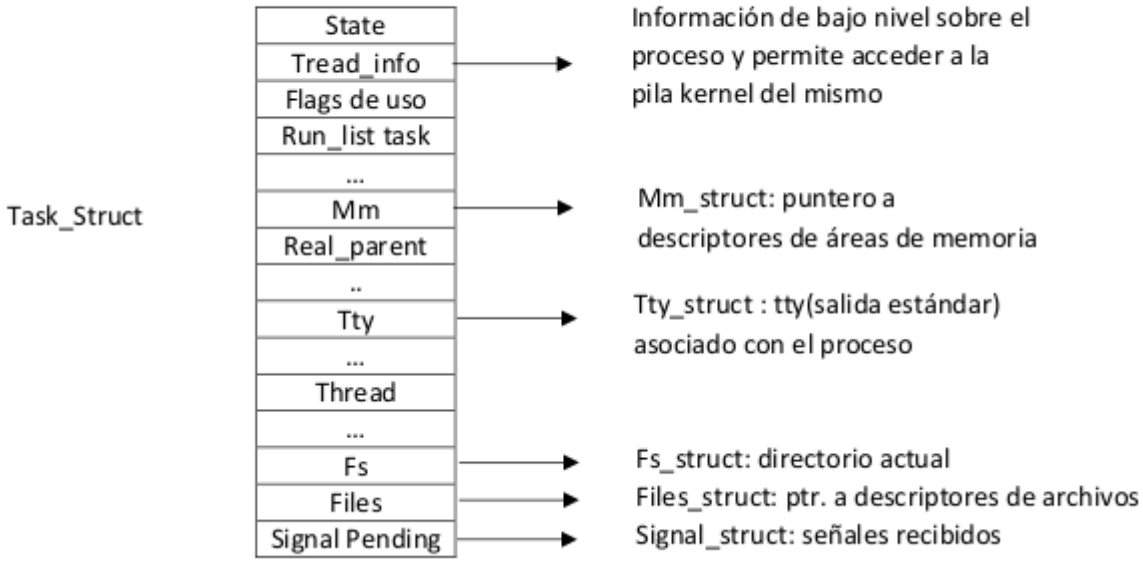
➜Periódicas: se sabe cada cuánto tiempo se tiene que ejecutar

➜Aperiódicas: tiene un plazo en el que debe comenzar o acabar o restricciones respecto a esos tiempos pero son impredecibles

B. Implementación de procesos / hilos en Linux

8 Bloque de Control de Proceso (PCB)

En Linux, al PCB se lo denomina descriptor de proceso y viene descrito por un task_struct.

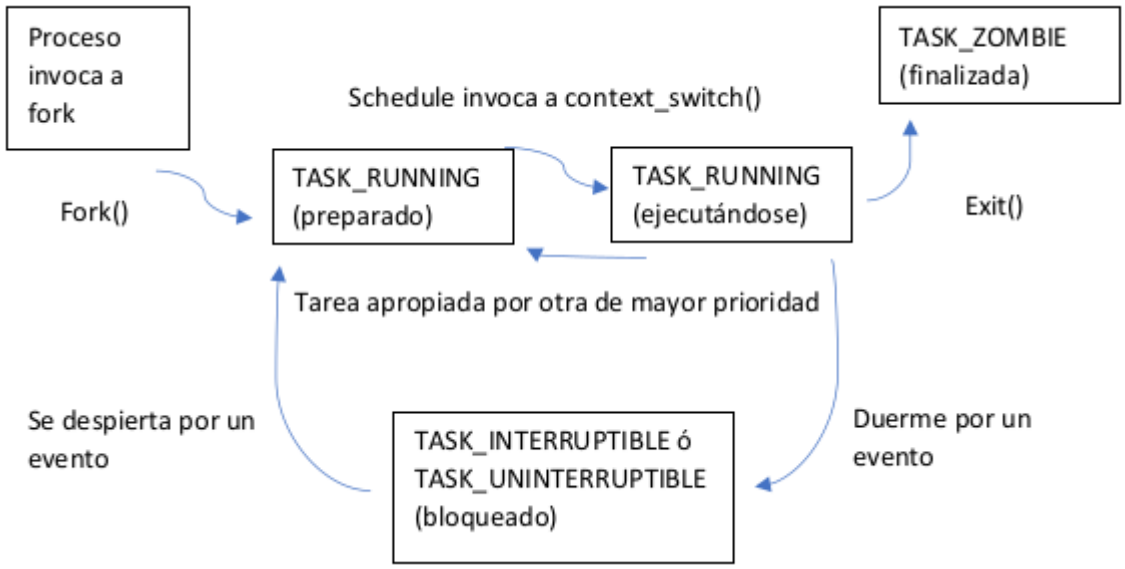


Las sub-estructuras mm-struct, files_struct, fs_struct, tty_struct y signal_struct se desgajan de la estructura principal por los siguientes motivos:
• No son asignados cuando no son necesarios
• Permiten su compartición cuando sea necesario

9 Estado de los procesos

El campo state almacena el estado:
•**Task_running:** El proceso es ejecutable o está en ejecución
•**Task_interruptible**: El proceso está bloqueado de forma que puede ser interrumpido con una señal
•**Task_uninterruptible**: Proceso bloqueado no despertable por una señal
•**Task_traced**: Proceso que está siendo "traceado" por otro
•**Task_stopped**: La ejecución se ha detenido a causa de alguna de las señales de control de trabajo
•**Exit_state**: almacena la condición en que ha finalizado pudiendo darse 2 casos:

- ***Exit_dead***: va a ser eliminado, su padre invoca wait().
- ***Exit_zombie***: aún conserva los recursos, el padre no ha realizado wait().



10 Transición entre estados

Clone(): Llamada al sistema para crear un proceso / hilo.
Exit(): Llamada para finalizar un proceso.
Sleep(): bloquea / duerme a un proceso.
Wakeup(): desbloquea / despierta a un proceso cuando se ha producido el evento por el que esperaba
Schedule(): planificador que decide que proceso tiene el control de la CPV.

11 Jerarquía de procesos

Todos los procesos forman parte de una jerarquía, con el proceso system / init (PID=1) a la cabeza. Todos los procesos tienen un único padre, pero un proceso padre puede tener varios hijos (denominados entre sí hermanos)
La relación entre procesos se almacena en el PCB:
Parent: puntero al padre,**Children:** Lista de hijos,**Sibling**: Lista de hermanos
Un proceso localiza a su padre con *my_parent = current -> parent.*
El padre recorre la lista de sus hijos con: *list_for_children(list, &current > children)*

12 Manipulación de procesos

Dos ejemplos de llamada para la manipulación de procesos serían las siguientes:
Clone(): Crea un proceso o hilo desde otro con las características que se especifican en los argumentos
Exec(): Ejecuta un programa dentro de un proceso existente a partir del ejecutable que se pasa como argumento

12.1 Clone

El modelo de hilos de Linux es 1:1, es decir, cada hilo de usuario está soportado por un hilo kernel.
Algunos indicadores de clone serían:
CLONE_FILES: padre e hijo comparten los mismos archivos abiertos.
CLONE_FS: padre e hijo comparten la información del sistema de archivos.
CLONE_VM: comparten el espacio de direcciones de usuario

CLONE_SIGHAND: comparten los manejadores de señales y señales bloqueadas
CLONE_THREAD: ambos procesos / hilos pertenecen al mismo GID.
*/*Nota: La llamada para crear procesos (no hilos) es fork()*/*
El trabajo que realiza clone es:
Dup_task_struct: copia el descriptor del proceso actual
Alloc_pid: le asigna un nuevo PID
Al iniciar la task_struct, diferenciamos los hilos padre e hijo y ponemos a este último en estado no interrumpible
Sched_fork: marcamos el hilo como TASK_RUNNING y se inicializa información sobre la planificación

- Se comparten componentes según los indicadores
- Se asigna ID, relaciones de parentesco…

Sin embargo clone() debe hacer algunas comprobaciones:

- Algunos indicadores no tienen sentido juntos, en cambio otros deben aparecer a la vez.
- Cuando aparecer el flag CLONE_XXX indica que XXX_struct debe compartirse en vez de copiarse.

12.2 Hilos Kernel

Son hilos que no tienen espacio de direcciones de usuario, por tanto su descriptor tiene task_struct -> mm = NULL.
Se encargan de realizar labores del sistema. Estos solo se pueden crear desde otro hilo kernel con la función *kthread_create()*

12.3 Creación de un proceso

1. Crea la estructura thread_info (pila Kernel) y la task_struct para el nuevo proceso con los valores de la tarea actual.
2. Para los elementos de task_struct del hijo que deban tener valores distintos a los del padre, se les dan los valores iniciales correctos.
3. Se establece el estado del hijo a TASK_UNINTERRUPTIBLE mientras se realizan las restantes acciones.
4. Se establecen valores adecuados para los flags de la task_struct del hijo: flag PF_SUPERPRIV = 0 (la tarea no usa privilegio de superusuario) y flag PF_FORKNOEXEC =1 (el proceso ha hecho fork pero no exec)
5. Se llama a alloc_pid()para asignar un PID a la nueva tarea
6. Según cuáles sean los flags pasados a clone()
, duplica o comparte recursos como archivos abiertos, información de sistemas de archivos, manejadores de señales, espacio de direccionamiento del proceso.....
7. Se establece el estado del hijo a TASK_RUNNING
8. Finalmente copy_process() termina devolviendo un puntero a la task_struct del hijo.

12.4 Terminar un proceso

Se produce cuando un proceso:
Invoca voluntariamente:
Exit() o return() (en el main) finalizando el proceso primero a nivel de biblioteca
Exit() llamado al SO, si se invoca directamente no da la oportunidad de finalizar el proceso a nivel de biblioteca
Involuntariamente recibe una señal y se aplica la acción de terminar.

12.4.1 Exit()

Activa el PF_EXITING. Decrementa los contadores de uso de mm_struct, fs_struct,

files_struct. Si estos contadores alcanzan el valor 0 se libera los recursos.
Ajusta el exit_code del descriptor, que será devuelto al padre, con el valor pasado al invocar a exit().
Envío al padre la señal de finalización: si tiene algún hijo le busca un padre en el grupo y pone el estado a TASK_ZOMBIE.
Para finalizar invoca a Schedule() para ejecutar otro proceso

12.4.2 Wait()

Se trata de una llamada que bloquea un proceso padre hasta que un hijo finaliza. Cuando esto ocurre, devuelve al llamador el PID del hijo finalizado y el estado de finalización. Esta función invoca a reléase_task() que:
Si es la última tarea de su grupo y se encuentre en estado zombie, se le notifica al padre.
Libera la memoria de la pila Kernel, thread_info y task_struct.

13 Planificación de procesos en Linux

13.1 Linux y la apropiatividad

Los kernel actuales son apropiativos y permiten ajustar el grado de apropiatividad que le vayamos a dar.
Se implementa a través de puntos de apropiación, los cuales son puntos de flujo de ejecución de kernel donde es posible apropiar al proceso actual.
Los RSIs en lugar de invocar a Schedule() directamente, indican que es necesario planificar cuando sea posible. Fuera de tratamiento de la interrupción y cuando las EDs kernel estén en estado seguro, se invocará al planificador.

13.2 Clases de planificación

Un SO debe soportar al menos 3 clases de planificación, 2 de tiempo real (sched_FIFO y sched_RR) y una de tiempo compartido (sched_OTHERS).
Algunos campos relacionados con la planificación:

- static_prio: prioridad usada por el planificador

[0, 99] Prioridades para procesos de tiempo real
[100, 139] Prioridades para los procesos normales o regulares.

- Rt_priority: prioridad de tiempo real
- Sched_class: clase de planificación a la que pertenece el proceso.
- Sched_entity: se planifican entidades (proceso o grupos de procesos) permitiendo asegurar un porcentaje de CPU.

*Nota: Cada procesador tiene su cola de ejecución (rq-run queue) y cada proceso está en una única cola

13.3 Planificador periódico: scheduler_tick()

Se invoca con una frecuencia de Hz para contabilizar el tick transcurrido al proceso actual. Tiene dos funciones principales:

- Maneja estadísticas kernel relativas a planificación
- Activar el planificador periódico de la clase de planificación responsable del proceso actual, delegando la labor en el planificador de clase

13.4 Planificador principal

La función Schedule() se invoca directamente en diversos puntos del kernel para cambiar de proceso. Además cuando retornamos de una llamada al sistema, comprobamos si hay que replanificar mediante TIF_NEED_RESCHED, y si es necesario se invoca a Schedule().

13.5 Planificador: algoritmo

1. Selecciona la cola y el proceso actual
2. Desactiva la tarea actual de la cola
3. Selecciona el siguiente proceso a ejecutar
4. Invoca el cambio de contexto
5. Comprueba si hay que replanificar

14 Cambio de contexto

El cambio de contexto lo realizan dos funciones: **Switch_mm()** cambia el contexto de memoria del usuario descrito por task_struct-> mm ; y **Switch_to (prev, next, prev)** cambia los contenidos de los registros del procesador y la pila kernel
Si la hebra entrante / saliente es kernel utiliza un mecanismo denominado TLB perezosa, en el que se indica al procesador que realmente no hay que conmutar la memoria.