Centro de Enseñanza Técnica Industrial

Plantel Colomos

Ingeniería en Desarrollo de Software

Nombre Alumno: José Rafael Ruiz Gudiño

Registro: 20110374

Arquitectura de Sistemas Operativos

Act.1 Administración de Procesos. Resumen y crucigrama.

4°P

T/M

27/09/2021

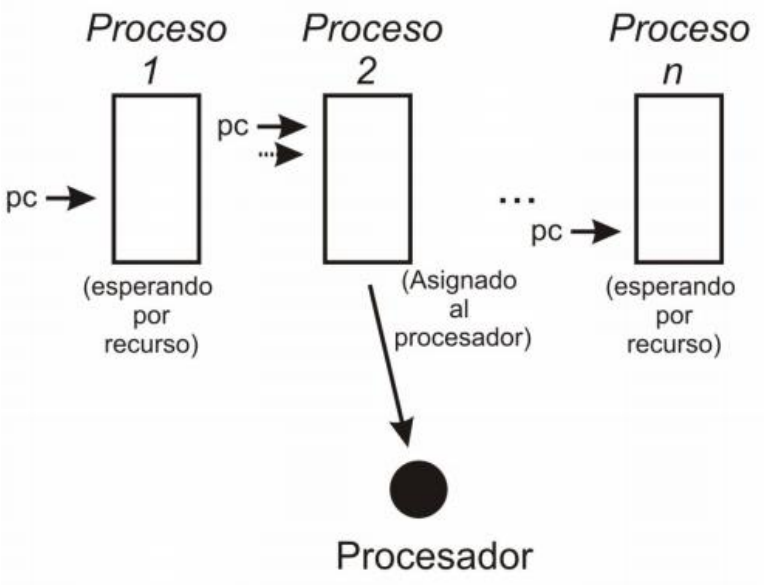
Procesos. Concepto y Estados de un Proceso

Definición

- El principal concepto en cualquier sistema operativo es el de proceso.
- Un proceso es un programa en ejecución, incluyendo el valor del program counter, los registros y las variables.
- Conceptualmente, cada proceso tiene un hilo (thread) de ejecución que es visto como un CPU virtual.
- El recurso procesador es alternado entre los diferentes procesos que existan en el sistema, dando la idea de que ejecutan en paralelo (multiprogramación).

Contador de programa

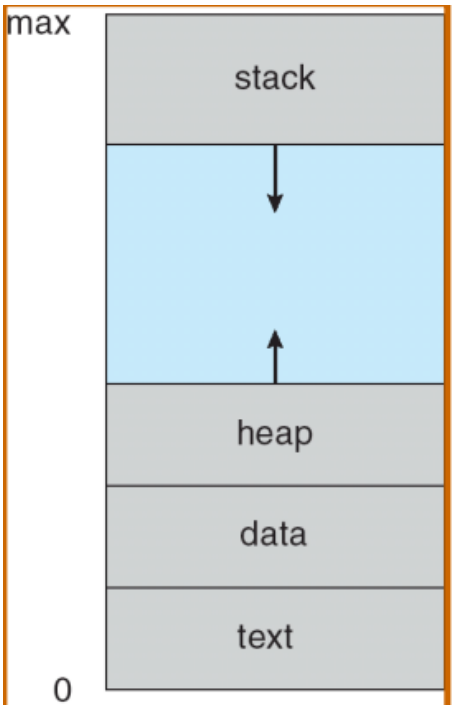
Cada proceso tiene su program counter, y avanza cuando el proceso tiene asignado el recurso procesador. A su vez, a cada proceso se le asigna un número que lo identifica entre los demás: identificador de proceso (process id).



Memoria de los procesos

Un proceso en memoria se constituye de varias secciones:

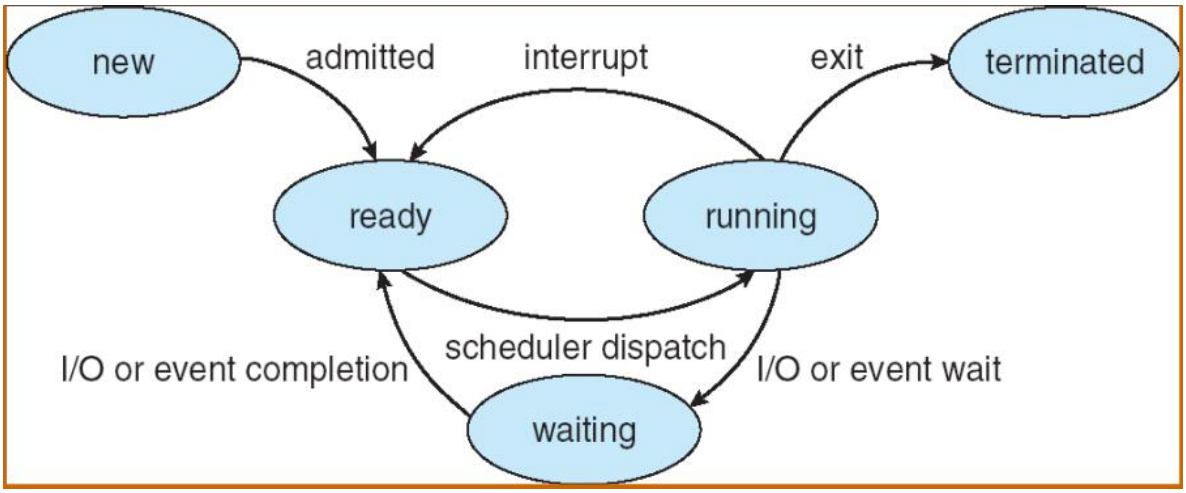
- Código (text): Instrucciones del proceso.
- Datos (data): Variables globales del proceso.
- Memoria dinánica (heap): Memoria dinámica que genera el proceso.
- Pila (stack): Utilizado para preservar el estado en la invocación anidada de procedimientos y funciones.



Estados de los procesos

El estado de un proceso es definido por la actividad corriente en que se encuentra. Los estados de un proceso son:

- Nuevo (new): Cuando el proceso es creado.
- Ejecutando (running): El proceso tiene asignado un procesador y está ejecutando sus instrucciones.
- Bloqueado (waiting): El proceso está esperando por un evento (que se complete un pedido de E/S o una señal).
- Listo (ready): El proceso está listo para ejecutar, solo necesita del recurso procesador.
- Finalizado (terminated): El proceso finalizó su ejecución

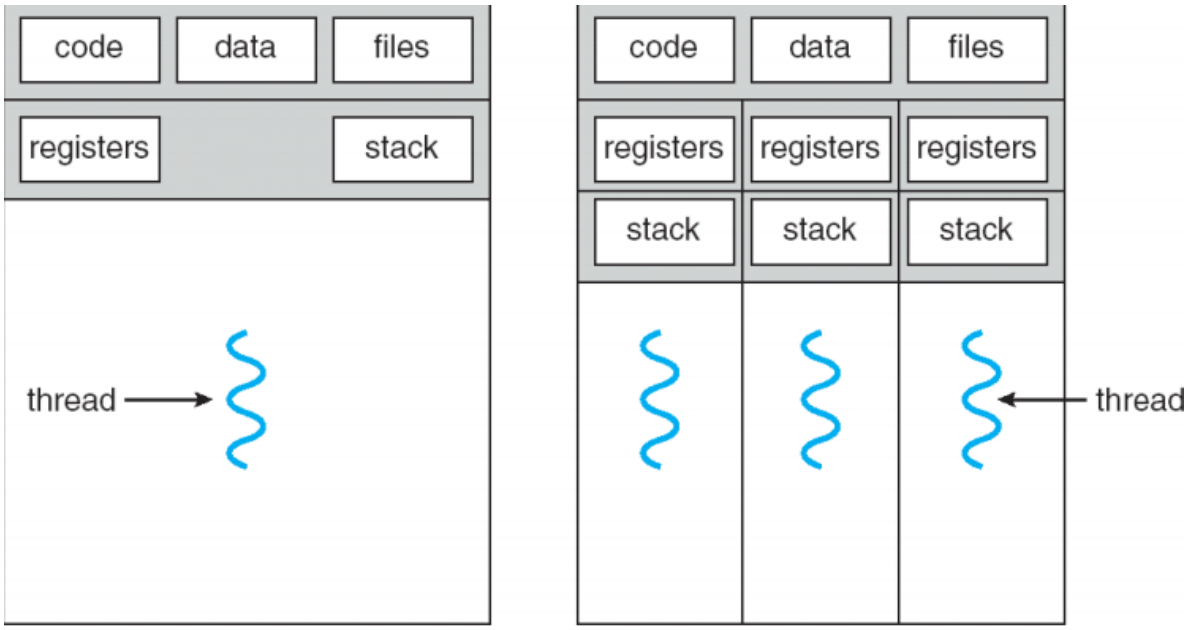


. Transiciones entre estados

- Nuevo ⇒ Listo
  - Al crearse un proceso pasa inmediatamente al estado listo.
- Listo ⇒ Ejecutando
  - En el estado de listo, el proceso solo espera para que se le asigne un procesador para ejecutar (tener en cuenta que puede existir más de un procesador en el sistema). Al liberarse un procesador el planificador (scheduler) selecciona el próximo proceso, según algún criterio definido, a ejecutar.
- Ejecutando ⇒ Listo
  - Ante una interrupción que se genere, el proceso puede perder el recurso procesador y pasar al estado de listo. El planificador será el encargado de seleccionar el próximo proceso a ejecutar.
- Ejecutando ⇒ Bloqueado
  - A medida que el proceso ejecuta instrucciones realiza pedidos en distintos componentes (ej.: genera un pedido de E/S). Teniendo en cuenta que el pedido puede demorar y, además, si está en un sistema multiprogramado, el proceso es puesto en una cola de espera hasta que se complete su pedido. De esta forma, se logra utilizar en forma más eficiente el procesador.
- Bloqueado ⇒ Listo
  - Una vez que ocurre el evento que el proceso estaba esperando en la cola de espera, el proceso es puesto nuevamente en la cola de procesos listos.
- Ejecutando ⇒ Terminado
  - Cuando el proceso ejecuta su última instrucción pasa al estado terminado. El sistema libera las estructuras que representan al proceso.

Hilos (subprocesos) y Concurrencia

- Un Thread (Hilo) es una unidad básica de utilización de la CPU consistente en un juego de registros y un espacio de pila. Es también conocido como proceso ligero.
- Cada thread contendrá su propio program counter, un conjunto de registros, un espacio para el stack y su prioridad.
- Comparten el código, los datos y los recursos con sus hebras (thread) pares.
- Una tarea (o proceso pesado) está formado ahora por uno o varios threads.
- Un thread puede pertenecer a una sola tarea.
- Todos los recursos, sección de código y datos son compartidos por los distintos threads de un mismo proceso.



single-threaded process multithreaded process

Concurrencia

Concurrencia: es una propiedad de los sistemas en la cual los procesos de un cómputo se hacen simultáneamente, y pueden interactuar entre ellos, es decir son procesados al mismo tiempo, de manera que, para ejecutar uno de ellos, no hace falta que se haya ejecutado otro.

En otras palabras la concurrencia es simplemente la ejecución de varias tareas al mismo tiempo. Ahora esto de ejecutarse "al mismo tiempo" es relativo de acuerdo a la arquitectura del ordenador, - ¿espera que? de seguro preguntaras- esto es debido a que, de acuerdo a la arquitectura básica, el ordenador solo puede ejecutar un programa o tarea a la vez, y no más, si nosotros percibimos la ilusión de que varios programas se ejecutan al mismo tiempo (Multiprogramación) es debido a que el sistema operativo intercala las tareas que se ejecutan en el procesador a una velocidad sumamente rápida.

Jerarquía de Procesos

En algunos sistemas, cuando un proceso crea otro, el proceso padre y el proceso hijo continúan asociados en ciertas formas. El proceso hijo puede crear por sí mismo más procesos, formando una jerarquía de procesos. Observe que, a diferencia de las plantas y los animales que utilizan la reproducción sexual, un proceso sólo tiene un padre (pero cero, uno, dos o más hijos).

En UNIX, un proceso y todos sus hijos, junto con sus posteriores descendientes, forman un grupo de procesos. Cuando un usuario envía una señal del teclado, ésta se envía a todos los miembros del grupo de procesos actualmente asociado

con el teclado (por lo general, todos los procesos activos que se crearon en la ventana actual). De manera individual, cada proceso puede atrapar la señal, ignorarla o tomar la acción predeterminada que es ser eliminado por la señal.

En contraste, Windows no tiene un concepto de una jerarquía de procesos. Todos los procesos son iguales. La única sugerencia de una jerarquía de procesos es que, cuando se crea un proceso, el padre recibe un indicador especial un token (llamado manejador) que puede utilizar para controlar al hijo. Sin embargo, tiene la libertad de pasar este indicador a otros procesos, con lo cual invalida la jerarquía. Los procesos en UNIX no pueden desheredar a sus hijos.

Llamadas al sistema para administración de procesos

Este tipo de llamadas al sistema permiten realizar "actividades" relacionadas con los programas que están en espera para ejecución o que se están ejecutando, es decir, cuando son un proceso; cada sistema operativo tiene sus formas de invocar dichas llamadas, en el caso de los sistemas operativos bajo el núcleo Linux estas se pueden invocar desde el Shell o interprete de comandos y desde el lenguaje C. Las llamadas más comunes de este tipo son

1. ***fork( ):***

   Esta es una llamada a sistema para el control de procesos que al ser invocada genera un proceso hijo, como una copia del proceso en donde fue invocada la llamada, este hijo es casi exactamente igual a su padre (lo único que los diferencia es su pid), retorna el pid del hijo al padre, al hijo le retorna en la copia de la llamada fork() el valor cero sin generar otra copia y ejecuta el proceso hijo desde este punto del código, es decir, lo que hay tras la invocación fork() no se ejecuta.

2. ***getpid( )***
   Es una llamada para el control de procesos que retorna el pid del proceso que la invoco.
   Esta función no recibe ningún argumento y retorna un entero del tipo pid_t.
   Para hacer uso de esta llamada se debe incluir en el código los archivos de cabecera:
   sys/types.h
   unistd.h
3. ***getppid( )***
   Es una llamada para el control de procesos que retorna el pid del padre del proceso que la invoca.

En forma análoga a getpid no recibe ningún argumento, retorna un entero del tipo pid_t y requiere de los archivos de cabecera:

sys/types.h
unistd.h

4. ***La familia de llamadas exec…(…)***

   La familia exec... () es un conjunto de funciones que en esencia realizan la misma actividad ya que solo difieren en la forma de pasar sus argumentos, son utilizadas para poner en ejecución un proceso determinado, la característica es que las instrucciones del proceso que las invoca son sustituidas por las instrucciones del proceso indicado. Dichas funciones son:

   ```
   execl(const char *path, const char *arg, ...);
   execlp(const char *file, const char *arg, ...);
   execle(const char *path, const char *arg , ..., char * const envp[]);
   execv(const char *path, char *const argv[]);
   execvp(const char *file, char *const argv[]);
   ```

   Todas las funciones reciben como primer argumento la ruta absoluta del archivo a ejecutar (ej.: "/bin/echo") la diferencia está en los demás argumentos:

   -En unas cada uno de los siguientes argumentos (a partir del segundo) es una opción o parámetro que recibe el archivo ejecutable para su funcionamiento, si lo requiere, y el último argumento es NULL.
   -En las demás el segundo argumento es un arreglo de caracteres, donde cada elemento del arreglo es una opción o parámetro que el archivo ejecutable requiere para funcionar, si lo requiere, y el último elemento es NULL.

5. ***wait( )***

   Al ser invocada, suspende el proceso que la invocó para que su proceso hijo termine de ejecutar su código (es decir, espera a su hijo) o hasta que el proceso invocador reciba una señal de terminación de proceso.

   Esta función recibe como argumento un puntero a una variable entera en la que se colocara el estado actual del proceso hijo y retorna el pid del proceso hijo que termino.

   El argumento también puede colocarse NULL

   Para poder ser invocada deben incluirse los archivos de cabecera:

   sys/types.h

   sys/wait.h

6. ***waitpid(…)***

   Al ser invocada, suspende el proceso que la invocó para que su proceso hijo termine de ejecutar su código (es decir, espera a su hijo) o hasta que el proceso invocador reciba una señal de terminación de proceso.

   Esta función recibe como argumento un puntero a una variable entera en la que se colocara el estado actual del proceso hijo y retorna el pid del proceso hijo que termino.

   El argumento también puede colocarse NULL

   Para poder ser invocada deben incluirse los archivos de cabecera:

   sys/types.h
   sys/wait.h

7. ***exit(…)***

   Su funcionamiento es análogo a la instrucción return x;, es decir, finaliza el proceso que la invoca y retorna al proceso padre el valor colocado como argumento (si es cero indica que se terminó normalmente, en caso contrario indica que se produjo un error).

   No retorna ningún valor al proceso que la invoca y debe incluirse el archivo de cabecera: stdlib.h para poderse usar.

   Para esta llamada no se darán ejemplos pues su uso es obvio y además los códigos de todos los ejemplos expuestos la usan para terminar el proceso.

8. ***system(…)***

   La acción que realiza es similar a la de la familia exec, con la diferencia de que solo permite invocar comandos del shell y cuando termina la ejecución del comando retorna el control al proceso que lo invoco, es decir, el proceso continúa ejecutando las instrucciones que hay después de la llamada.

   Recibe como argumento el nombre del comando y retorna al proceso que la invoco -1 si se produce un error, 0 si no hay shell disponible o diferente de cero en cualquier otro caso. Para poder usarse se debe incluir el archivo de cabecera: stdlib.h

   La función system(...) busca el comando en el directorio /bin/.

Planificación de procesos y sus tipos

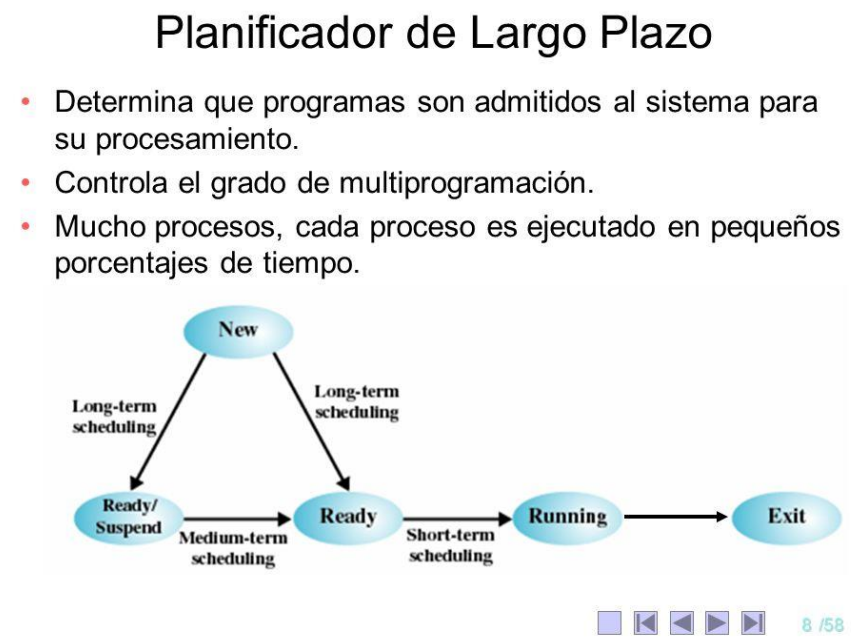
La planificación de procesos se refiere a cómo determina el sistema operativo al órden en que irá cediendo el uso del procesador a los procesos que lo vayan solicitando, y a las políticas que empleará para que el uso que den a dicho tiempo no sea excesivo respecto al uso esperado del sistema.

Podemos hablar de tres tipos principales de planificación:

A largo plazo

Decide qué procesos serán los siguientes en ser iniciados. Este tipo de planificación era el más frecuente en los sistemas de lotes (principalmente aquellos con spool) y multiprogramados en lotes; las decisiones eran tomadas principalmente considerando los requisitos pre-declarados de los procesos y los que el sistema tenía libres al terminar algún otro proceso. La planificación a largo plazo puede llevarse a cabo con periodicidad de una vez cada varios segundos, minutos e inclusive horas.

En los sistemas de uso interactivo, casi la totalidad de los que se usan hoy en día, este tipo de planificación no se efectúa, dado que es típicamente el usuario quien indica expresamente qué procesos iniciar.



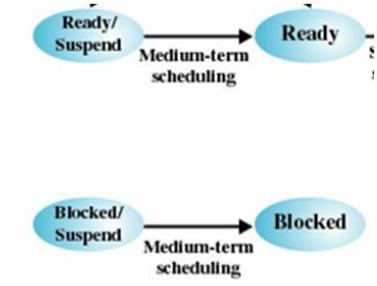
A mediano plazo

Decide cuáles procesos es conveniente *bloquear* en determinado momento, sea por escacez/saturación de algún recurso (como la memoria primaria) o porque están realizando alguna solicitud que no puede satisfacerse momentaneamente; se encarga de tomar decisiones respecto a los procesos conforme entran y salen del estado de *bloqueado* (esto es, típicamente, están a la espera de algún evento externo o de la finalización de transferencia de datos con algún dispositivo).

En algunos textos, al *planificador a mediano plazo* se le llama *agendador* (*scheduler*).

Planificador de Mediano Plazo

- Parte de la función de intercambio (swaping).
- Basado en la necesidad de administrar el grado de multi-programación.



A corto plazo

Decide cómo compartir *momento a momento* al equipo entre todos los procesos que requieren de sus recursos, especialmente el procesador. La planificación a corto plazo se lleva a cabo decenas de veces por segundo (razón por la cual debe ser código muy simple, eficiente y rápido); es el encargado de planificar *los procesos que están listos para ejecución*.

En algunos textos, al *planificador a corto plazo* se le llama *despachador* (*dispatcher*).

PLANIFICADOR A CORTO PLAZO