Procesos e hilos

Pedro O. Pérez M., Phd.

Multiprocesadores
Tecnológico de Monterrey

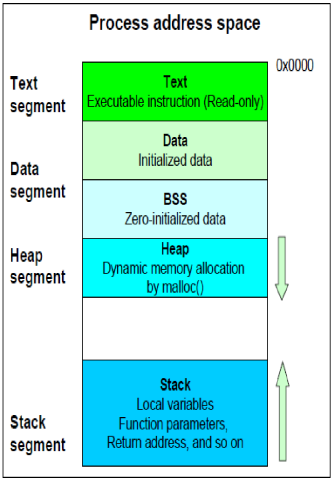
pperezm@tec.mx

08-2023

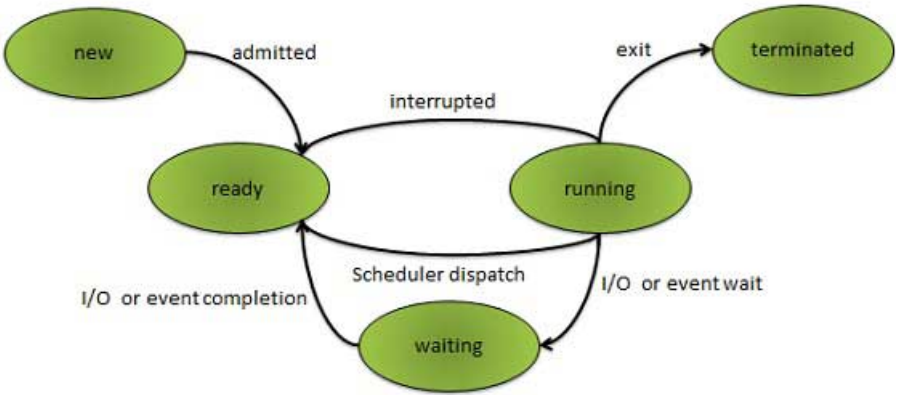
Contenido

- Las primeras computadoras de escritorio solo contaban con un CPU, y solo eran capaces de ejecutar un programa la vez. Posteriormente vino la multitarea y fue posible que las computadoras pudieran ejecutar múltiples programas (tareas o procesos) al mismo tiempo.
- Aunque, para ser sinceros, no se ejecutan al mismo tiempo. El CPU es compartido entre los programas y el sistema operativo el encargado de determinar cuál programa y por cuánto tiempo.
- Un proceso es la unidad de trabajo de la mayoría de los sistemas operativos. Un proceso puede ser visto como un programa en ejecución.

- Un proceso es más que el código del programa, que a veces es conocido como el segmento del código.
- También incluye la actividad actual, representada por el valor del contador del programa y los valores de los registros del procesador.
- Un proceso, generalmente, también incluye una pila que contiene datos temporales (como los parámetros de las funciones, la dirección de retorno y las variables locales) y un segmento de datos que contiene variables globales.
- Un proceso también puede incluir un "heap", que es la memoria que se asigna dinámicamente durante el tiempo de ejecución del proceso.



Estados de un proceso



- Los hilos a veces se llaman procesos livianos. Ambos, proceso e hilo, proporcionan un entorno de ejecución, pero la creación de un nuevo hilo requiere menos recursos que la creación de un nuevo proceso.
- Los hilos existen dentro de un proceso; cada proceso tiene al menos un hilo. Los hilos comparten los recursos del proceso, incluida la memoria y los archivos abiertos. Este lo convierte en una comunicación eficiente, pero potencialmente problemática.

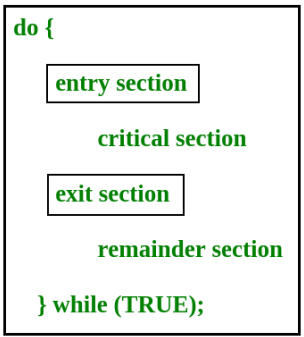
¿Por qué deberíamos tener múltiples hilos de ejecución dentro de un único contexto de proceso?

- Considera, por ejemplo, una aplicación GUI donde el usuario puede emitir un comando que requiere mucho tiempo para terminar (por ejemplo, bajar un archivo muy grande). A menos que se diseñe este comando para que se ejecute en un hilo separado, no se podrá interactuar con la GUI principal de la aplicación (por ejemplo, para actualizar una barra de progreso) porque no responderá mientras se lleva a cabo el cálculo.

Por supuesto, el diseño de aplicaciones multihilos requiere que el desarrollador maneje situaciones que simplemente no concurrente cuando se desarrollan aplicaciones secuenciales de proceso único. Por ejemplo, cuando dos o más hilos intentan acceder y modificar un recurso compartido (condición del carrera), el desarrollador debe asegurarse de que esto no dejará al sistema en un estado incoherente o de interbloqueo.

¿Qué es una sección crítica?

- Considera un sistema que consta de n procesos $(P_0, P_1, ..., P_{n-1}, P_n)$. Cada proceso tiene un segmento de código llamado sección crítica.
- Cuando un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su correspondiente sección crítica.
- Cualquier solución al problema de la sección crítica deberá satisfacer tres requisitos:
  - Exclusión mutua.
  - Progreso.
  - Espera limitada.



- El problema de la sección crítica puede resolverse de forma fácil en un entorno de un solo procesador si pudiéramos impedir que se produjeran interrupciones mientras se está modificando una variable compartida.
- Lamentablemente, esta solución no resultada adecuada en un entorno multiprocesador. Desactivar las interrupciones en un sistema multiprocesador puede consumir mucho tiempo, ya que hay que pasa el mensaje a todos los procesadores.

- Un semáforo S es una variable entera a la que, sin contar a la inicialización, sólo se accede mediante dos operaciones atómicas estándar: wait (P - proberen)y signal (V - verhogen).
- Los sistemas operativos diferencian entre semáforos contadores (más de una instancia a controlar) y semáforos binarios (una sola instancia a controlar).

Semaphore Structure:

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

Wait Operation:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}
```

Signal Operation:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```