4

La Abstracción: El Proceso

En este capítulo, discutimos una de las abstracciones más fundamentales que el SO proporciona a los usuarios: el **proceso**. La definición de un proceso, informalmente, es bastante simple: es un **programa en ejecución** [V+65,BH70]. El programa en sí mismo es una cosa sin vida: sólo está allí en el disco, un montón de instrucciones (y quizás algunos datos estáticos), esperando entrar en acción. Es el sistema operativo el que toma estos bytes y los pone en marcha, transformando el programa en algo útil.

Resulta que a menudo uno quiere ejecutar más de un programa a la vez. Por ejemplo, en tu computadora o notebook, donde podrías querer ejecutar un navegador web, un programa de correo, un juego, un reproductor de música, etc. De hecho, un sistema típico puede parecer que ejecuta decenas o incluso cientos de procesos al mismo tiempo. Esto hace que el sistema sea fácil de usar, ya que uno nunca tiene que preocuparse de si una CPU está disponible; uno simplemente ejecuta programas. De ahí nuestro reto:

EL PROBLEMA EN CUESTIÓN:
¿CÓMO PROPORCIONAR LA ILUSIÓN DE MUCHAS CPUS?
A pesar de que sólo hay unas pocas CPUs físicas disponibles, ¿Cómo puede el sistema operativo proporcionar la ilusión de un suministro casi infinito de dichas CPUs?

El SO crea esta ilusión **virtualizando** la CPU. Esto lo logra ejecutando un proceso, luego deteniéndolo y ejecutando otro, y así sucesivamente, el SO puede crear la ilusión de que existen muchas CPUs virtuales cuando en realidad hay sólo una CPU física (o unas pocas). Esta técnica básica, conocida como **tiempo compartido** de la CPU, permite a los usuarios ejecutar tantos procesos concurrentes como deseen; el costo potencial es el rendimiento, ya que cada proceso se ejecutará más y más lentamente si la(s) CPU(s) debe(n) ser compartida(s).

TIP: USAR TIEMPO COMPARTIDO (Y ESPACIO COMPARTIDO)
El **tiempo compartido** es una técnica básica usada por un SO para compartir un recurso. Al permitir que el recurso sea usado un ratito por una entidad y luego otro ratito por otra, y así sucesivamente, el recurso en cuestión (por ejemplo, la CPU, o un enlace de una red) puede ser compartido por muchos. La contrapartida del tiempo compartido es el **espacio compartido**, donde un recurso se divide (en el espacio) entre aquellos que deseen utilizarlo. Por ejemplo, el espacio en disco es naturalmente un recurso de espacio compartido; una vez que se asigna un bloque a un archivo, normalmente no se asigna a otro archivo hasta que el usuario elimina el archivo original.

Para implementar la virtualización de la CPU, y para implementarla bien, el SO necesitará tanto maquinaria de bajo nivel como inteligencia de alto nivel. A la maquinaria de bajo nivel la llamamos **mecanismos**; los mecanismos son métodos o protocolos de bajo nivel que implementan una parte de la funcionalidad necesaria. Por ejemplo, más adelante aprenderemos cómo implementar un **cambio de contexto**, que le da al SO la capacidad de dejar de ejecutar un programa y empezar a ejecutar otro en una CPU determinada; este mecanismo de tiempo compartido es empleado por todos los SO modernos.

Encima de estos mecanismos reside parte de la inteligencia del SO, en forma de **políticas**. Las políticas son algoritmos para tomar algún tipo de decisión dentro del SO. Por ejemplo, dado un número de programas posibles para ejecutar en una CPU, ¿qué programa debería ejecutar el SO? Una **política de planificación** en el SO tomará esta decisión, probablemente utilizando información histórica (por ejemplo, ¿qué programa se ha ejecutado más en el último minuto?), conocimiento de la carga de trabajo (por ejemplo, qué tipos de programas se ejecutan) y métricas de rendimiento (por ejemplo, ¿el sistema está optimizando el rendimiento interactivo o el rendimiento de procesamiento?) para tomar su decisión.

4.1 La Abstracción: Un Proceso

La abstracción que proporciona el SO de un programa en ejecución es algo que llamaremos **proceso**. Como mencionamos antes, un proceso es simplemente un programa en ejecución; en cualquier momento, podemos resumir un proceso haciendo un inventario de las diferentes piezas del sistema a las que accede o afecta durante el curso de su ejecución.

Para entender lo que constituye un proceso, tenemos que entender su **estado**: lo que un programa puede leer o actualizar cuando

se está ejecutando. En un momento dado, ¿qué partes de la máquina son importantes para la ejecución de este programa?

Un componente obvio del estado que comprende un proceso es su *memoria*. Las instrucciones se encuentran en la memoria; los datos que el programa en ejecución lee y escribe también se encuentran en la memoria. Por lo tanto, la memoria a la que el proceso puede direccionar (llamada su **espacio de direcciones**) es parte del proceso.

También forman parte del estado del proceso los *registros*; muchas instrucciones leen o actualizan explícitamente los registros y, por lo tanto, es evidente que son importantes para la ejecución del proceso.

TIP: SEPARAR POLÍTICAS Y MECANISMOS

En muchos sistemas operativos, un paradigma de diseño común es separar las políticas de alto nivel de sus mecanismos de bajo nivel [L+75]. Se puede pensar en el mecanismo como una forma de proporcionar la respuesta del *cómo* sobre un sistema; por ejemplo, ¿*cómo* realiza un sistema operativo un cambio de contexto? La política proporciona la respuesta del *cuál*; por ejemplo, ¿*cuál* proceso debería ejecutar el sistema operativo en este momento? Separarlos permite cambiar fácilmente las políticas sin tener que repensar el mecanismo y, por lo tanto, es una forma de **modularidad**, un principio general de diseño de software.

Notar que hay algunos registros particularmente especiales que forman parte de este estado. Por ejemplo, el **contador de programa** (PC) (a veces llamado **puntero de instrucción** o **IP**) nos dice qué instrucción del programa se ejecutará a continuación; de forma similar, un **puntero de stack** y el **puntero de frame** asociado se utilizan para gestionar el stack para parámetros de función, variables locales y direcciones de retorno.

Por último, los programas también suelen acceder a dispositivos de almacenamiento persistente. Esta *información de E/S* (entrada, salida) puede incluir una lista de los archivos que el proceso tiene abiertos en ese momento.

4.2 La API de los procesos

Aunque discutiremos de manera más profunda la API de procesos en un capítulo posterior, aquí daremos una primera idea de lo que debe incluirse en cualquier interfaz de un sistema operativo. Estas APIs, de alguna forma, están disponibles en cualquier sistema operativo moderno.

- **Crear**: Un sistema operativo debe incluir algún método para crear nuevos procesos. Cuando se escribe un comando en la terminal, o se hace doble clic en el ícono de una aplicación, se

invoca al sistema operativo para que cree un nuevo proceso que ejecute el programa indicado.

- **Destruir**: Al igual que existe una interfaz para la creación de procesos, los sistemas también proporcionan una interfaz para destruir procesos de forma forzosa. Por supuesto, muchos procesos se ejecutarán y saldrán por sí mismos cuando se completen; sin embargo, cuando no lo hacen, el usuario puede desear matarlos, y por lo tanto una interfaz para detener un proceso fuera de control es bastante útil.
- **Esperar**: A veces es útil esperar a que un proceso deje de ejecutarse, por lo que a menudo se proporciona algún tipo de interfaz de espera.
- **Controles varios:** Aparte de matar o esperar a un proceso, a veces hay otros controles posibles. Por ejemplo, la mayoría de los sistemas operativos proporcionan algún tipo de método para suspender un proceso (hacer que deje de ejecutarse durante un tiempo) y luego reanudarlo (que siga ejecutándose).
- **Estado**: Normalmente también hay interfaces para obtener información sobre el estado de un proceso, como el tiempo que lleva en ejecución o el estado en el que se encuentra.

4.3 Creación de Procesos: Un poco mas detallado

Un misterio que deberíamos desenmascarar un poco es cómo los programas se transforman en procesos. En concreto, ¿cómo consigue el SO que un programa se ejecute? ¿Cómo funciona realmente la creación de procesos?

Lo primero que debe hacer el SO para ejecutar un programa es **cargar** su código y cualquier dato estático (por ejemplo, variables inicializadas) en la memoria, en el espacio de direcciones del proceso. Los programas residen inicialmente en el **disco** (o, en algunos sistemas modernos, en **unidades SSD**) en algún tipo de **formato ejecutable**; por lo tanto, el proceso de cargar un programa y los datos estáticos en la memoria requiere que el SO lea esos bytes del disco y los coloque en algún lugar de la memoria (como se muestra en la Figura 4.1).

En los primeros sistemas operativos (o los más sencillos), el proceso de carga se realiza de forma **anticipada**, es decir, todo de una vez antes de ejecutar el programa; los SO modernos realizan el proceso de forma **perezosa**, es decir, cargando partes del código o de los datos sólo cuando se necesitan durante la ejecución del programa. Para entender realmente cómo funciona la carga perezosa de trozos de código y datos, hay que entender más sobre la maquinaria de la **paginación** y el **swapping**, temas que cubriremos en el futuro

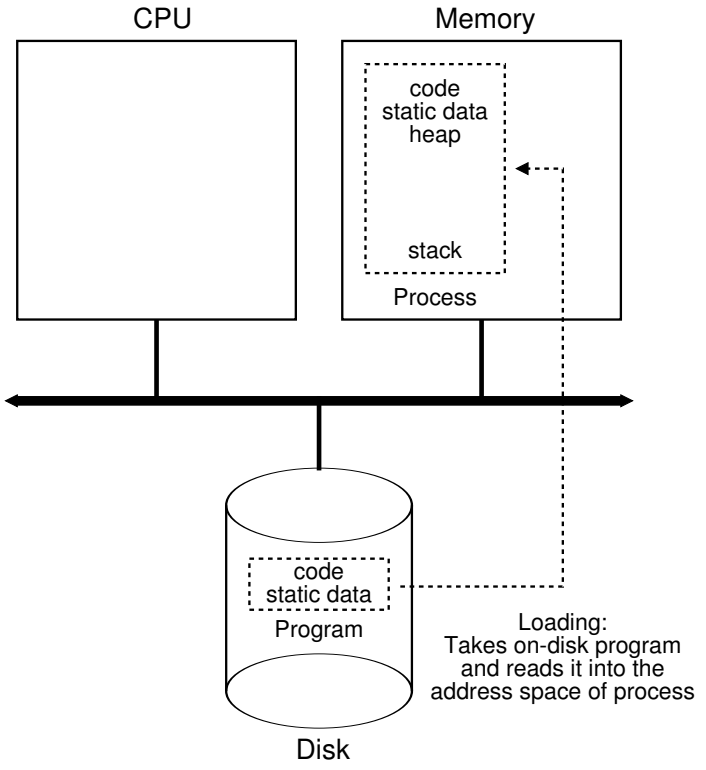

Figure 4.1: **Cargando: De Programa a Proceso**

cuando hablemos de la virtualización de la memoria. Por ahora, sólo recordemos que antes de ejecutar cualquier cosa, el SO claramente debe hacer algún trabajo para llevar los bits importantes del programa desde el disco a la memoria.

Una vez que el código y los datos estáticos se cargan en la memoria, hay algunas otras cosas que el SO necesita hacer antes de ejecutar el proceso. Se debe asignar algo de memoria para la **stack de ejecución del programa** (o simplemente stack). Como probablemente ya sepas, los programas C utilizan el stack para las variables locales, los parámetros de las funciones y las direcciones de retorno; el SO asigna esta memoria y se la da al proceso. El SO también inicializará el stack con argumentos; específicamente, llenará los parámetros de la función `main()`, es decir, `argc` y la matriz `argv`.

El SO también puede asignar algo de memoria para el **heap** del programa. En los programas de C, el heap se utiliza para los datos que se pidieron explícitamente y fueron asignados dinámicamente; los programas solicitan dicho espacio llamando a `malloc()` y lo liberan explícitamente llamando a `free()`. El heap es necesario para estructuras de datos como listas enlazadas, tablas hash, árboles y otras estructuras de datos interesantes. El heap será pequeño al principio; a medida que el programa se ejecuta, y solicita más memoria a través de la API de la biblioteca de `malloc()`, el SO puede involucrarse y asignar más memoria al proceso para ayudar a satisfacer dichas llamadas.

El sistema operativo también realizará algunas otras tareas de inicialización, especialmente las relacionadas con la entrada/salida (E/S o I/O). Por ejemplo, en los sistemas UNIX, cada proceso tiene por defecto tres **descriptores de archivo** abiertos, para la entrada, la salida y el error estándar; estos descriptores permiten a los programas leer fácilmente la entrada de la terminal e imprimir la salida en la pantalla. Aprenderemos más sobre I/O, descriptores de archivo y similares en la tercera parte del libro sobre **persistencia**.

Al cargar el código y los datos estáticos en la memoria, al crear e inicializar una pila y al realizar otros trabajos relacionados con la configuración de E/S, el SO (finalmente) ha preparado todo para la ejecución del programa. Por lo tanto, tiene una última tarea: iniciar la ejecución del programa en el punto de entrada, es decir, `main()`. Al saltar a la rutina `main()` (a través de un mecanismo especializado que discutiremos en el próximo capítulo), el SO transfiere el control de la CPU al proceso recién creado y, por lo tanto, el programa comienza su ejecución.

4.4 Estados de los procesos

Ahora que tenemos una idea de qué es un proceso (aunque continuaremos refinando este concepto) y (más o menos) cómo se crea, hablemos de los diferentes estados en los que puede estar un proceso en un momento dado. La noción de que un proceso puede estar en uno de estos estados surgió en los primeros sistemas informáticos [DV66,V+65]. A simple vista, un proceso puede estar en uno de tres estados:

- **Corriendo**: Si un proceso está en corriendo, está ejecutándose en un procesador. Esto significa que está ejecutando instrucciones.
- **Listo**: Si un proceso está listo, ya está preparado para ejecutarse pero, por algún motivo, el sistema operativo ha optado por no ejecutarlo en este momento.

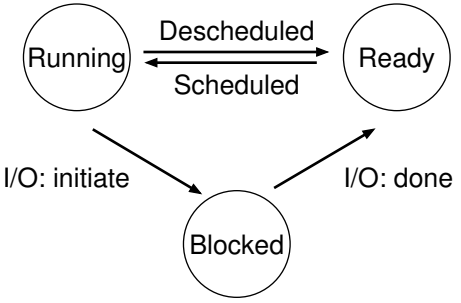

Figure 4.2: **Process: State Transitions**

- **Bloqueado**: Si un proceso está bloqueado, ha realizado algún tipo de operación que hace que no esté listo para ejecutarse hasta que ocurra algún otro evento. Un ejemplo común: un proceso inicia una solicitud de E/S en un disco, se bloquea y, por lo tanto, algún otro proceso puede usar el procesador.

Si mapeáramos estos estados en un gráfico, llegaríamos al diagrama de la figura 4.2. Como puede ver en el diagrama, un proceso se puede mover entre los estados listo y en ejecución a discreción del SO. Pasar de listo a en ejecución significa que el proceso ha sido **planificado**; ser movido de en ejecución a listo significa que el proceso ha sido **desplanificado**. Una vez que un proceso se ha bloqueado (por ejemplo, al iniciar una operación de E/S), el sistema operativo lo mantendrá así hasta que ocurra algún evento (por ejemplo, la finalización de E/S); en ese punto, el proceso vuelve al estado listo (y puede inmediatamente ejecutarse nuevamente, si el SO así lo decide).

Veamos un ejemplo de cómo dos procesos pueden pasar por algunos de estos estados. Primero, imagine dos procesos ejecutándose, cada uno de los cuales solo usa la CPU (no realizan E/S). En este caso, un rastro del estado de cada proceso podría verse así (Figura 4.3).

En el ejemplo, el primer proceso realiza una E/S después de ejecutarse durante algún tiempo. En ese momento, el proceso se bloquea, dando al otro proceso la oportunidad de ejecutarse. La Figura 4.4 muestra la traza de este escenario.

Más específicamente, Proceso_0 inicia una E/S y se bloquea esperando que se complete; los procesos se bloquean, por ejemplo, al leer de un disco o esperar un paquete de una red. El sistema op-

Tiempo	Proceso_0	Proceso_1	Notas
1	Corriendo	Listo	
2	Corriendo	Listo	
3	Corriendo	Listo	
4	Corriendo	Listo	Proceso_0 terminó
5	-	Corriendo	
6	-	Corriendo	
7	-	Corriendo	
8	-	Corriendo	Proceso_1 terminó

Figure 4.3: **Traza del estado del proceso: Solo CPU**

Tiempo	Proceso_0	Proceso_1	Notas
1	Corriendo	Listo	
2	Corriendo	Listo	
3	Corriendo	Listo	Proceso_0 inicia E/S
4	Bloqueado	Corriendo	Proceso_0 está bloqueado
5	Bloqueado	Corriendo	entonces corre Proceso_1
6	Bloqueado	Corriendo	
7	Listo	Corriendo	termina la E/S
8	Listo	Corriendo	Proceso_1 terminó
9	Corriendo	-	
10	Corriendo	-	Proceso_0 terminó

Figure 4.4: **Traza del estado del proceso: CPU y E/S**

erativo reconoce que Proceso_0 no está usando la CPU y comienza a ejecutar Proceso_1. Mientras se ejecuta el Proceso_1 , la E/S se completa y el Proceso_0 vuelve a estar listo.

Notar que hay muchas decisiones que debe tomar el SO, incluso en este ejemplo simple. Primero, el sistema tuvo que decidir ejecutar Proceso_1 mientras Proceso_0 emitía una E/S; hacerlo mejora la utilización de los recursos al mantener la CPU ocupada. En segundo lugar, el sistema decidió no volver a cambiar a Proceso_0 cuando se completó su E/S; no está claro si esta es una buena decisión o no. ¿Qué opinás? Este tipo de decisiones las toma el **planificador** del sistema operativo, un tema del que hablaremos en algunos capítulos más adelante.

4.5 Estructuras de Datos

El SO es un programa, y como cualquier programa, tiene algunas estructuras de datos clave que guardan varias piezas de información relevantes. Para guardar el estado de cada proceso, por ejemplo, el SO probablemente tendrá algún tipo de **lista de procesos** para todos los procesos que están listos y alguna información adicional para

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};
// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;     // Start of process memory
  uint sz;       // Size of process memory
  char *kstack; // Bottom of kernel stack
                // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;       // If !zero, sleeping on chan
  int killed;       // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;//Switch here to run
  struct trapframe *tf;   // Trap frame for the
                          // current interrupt
};
```

Figure 4.5: **La estructura `proc` de xv6**

saber qué proceso se está ejecutando actualmente. El SO también debe guardar, de alguna manera, los procesos bloqueados; cuando se completa un evento de E/S, el SO debe asegurarse de activar el proceso correcto y prepararlo para que se ejecute nuevamente.

La figura 4.5 muestra qué tipo de información necesita guardar un sistema operativo sobre cada proceso en el kernel de xv6 [CK+08]. Existen estructuras de procesos similares en SOs reales como Linux, Mac OS X o Windows; búsquenlos y vean cuán más complejos son.

En la figura, se pueden ver un par de datos importantes que el SO monitorea sobre un proceso. El **resgistro de contexto** tendrá, para un

proceso detenido, el contenido de sus registros. Cuando se detiene un proceso, sus registros se guardarán en esta ubicación de memoria; al restaurar estos registros (es decir, colocar sus valores nuevamente en los registros físicos reales), el sistema operativo puede reanudar la ejecución del proceso. Aprenderemos más sobre esta técnica conocida como **cambio de contexto** en capítulos futuros.

También se puede ver en la figura que hay otros estados en los que puede estar un proceso, además de corriendo, listo y bloqueado. A veces, un sistema tendrá un estado **inicial** en el que se encuentra el proceso cuando se crea. Además, un proceso podría colocarse en un estado **final** en el que ha salido pero aún no se ha limpiado (en los sistemas basados en UNIX, esto se denomina estado **zombie**[1]). Este estado final puede ser útil ya que permite que otros procesos (generalmente el padre, que creó el proceso) examinen el código de retorno del proceso y vean si el proceso que acaba de finalizar se ejecutó correctamente. (Por lo general, los programas devuelven cero en los sistemas basados en UNIX cuando han realizado una tarea con éxito y no-cero en caso contrario). Cuando termine, el padre hará una última llamada (por ejemplo, `wait()`) para esperar la finalización del proceso hijo y también para indicar al SO que puede limpiar cualquier estructura de datos relevante que se refiera al proceso extinto.

4.6 Resumen

Hemos introducido la abstracción más básica del sistema operativo: el proceso. Es es bastante simple si se ve como un programa en ejecución. Con esta visión conceptual en mente, ahora pasaremos a la parte esencial del asunto: los mecanismos de bajo nivel necesarios para implementar procesos y las políticas de alto nivel necesarias para programarlos de manera inteligente. Al combinar mecanismos y políticas, desarrollaremos nuestra comprensión de cómo un sistema operativo virtualiza la CPU.

[1] Si, estado zombie. Como los verdaderos zombies, estos zombies son relativamente fáciles de matar. Sin embargo, usualmente se recomiendan diferentes técnicas.

APARTE: TÉRMINOS CLAVE SOBRE PROCESOS

- El **proceso** es la principal abstracción del SO de un programa en ejecución. En cualquier momento, el proceso puede ser descripto por su estado: el contenido de la memoria en su espacio de direcciones, el contenido de los registros de la CPU (incluido el contador de programa y el puntero de pila, entre otros) y la información sobre E/S (como archivos abiertos que se pueden leer o escribir).
- La **API de procesos** consta de llamadas que los programas pueden hacer relacionadas con los procesos. Por lo general, esto incluye creación, destrucción y otras llamadas útiles.
- Los procesos existen en uno de los muchos **estados de proceso** diferentes, incluidos *corriendo*, *listos para ejecutar* y *bloqueados*. Diferentes eventos (por ejemplo, planificar o des-planificar, o esperar a que se complete una E/S) hacen que un proceso pase de uno de estos estados al otro.
- Una **lista de procesos** contiene información sobre todos los procesos del sistema. Cada entrada se encuentra en lo que a veces se denomina **bloque de control de procesos (PCB)**, que en realidad es solo una estructura que contiene información sobre un proceso específico.

4.7 Referencias

[BH70] "The Nucleus of a Multiprogramming System" por Per Brinch Hansen. Communications of the ACM, volumen 13:4, abril de 1970. *Este paper introduce uno de los primeros microkernels en la historia de los sistemas operativos, llamado Nucleus. La idea de sistemas más pequeños y mínimos es un tema que aparece repetidamente en la historia de los sistemas operativos; todo comenzó con el trabajo de Brinch Hansen descrito aquí.*

[CK+08] "The xv6 Operating System" por Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. De:
https://github.com/mit-pdos/xv6-public. *El sistema operativo real y pequeño más genial del mundo. Descargalo y jugá con él para obtener más información sobre los detalles de cómo funcionan realmente los sistemas operativos. Hemos estado usando una versión anterior (2012-01-30-1-g1c41342) y, por lo tanto, es posible que algunos ejemplos del libro no coincidan con la última versión de la fuente.*

[DV66] "Programming Semantics for Multiprogrammed Computations" por Jack B. Dennis, Conde C. Van Horn. Communications of the ACM, volumen 9, número 3, marzo de 1966. *Este documento definió muchos de los primeros términos y conceptos relacionados con la construcción de sistemas multiprogramados.*

[L+75] "Policy/mechanism separation in Hydra" por R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf. SOSP '75, Austin, Texas, noviembre de 1975. *Un artículo inicial sobre cómo estructurar los sistemas operativos en un SO de investigación conocido como Hydra. Si bien Hydra nunca se convirtió en un sistema operativo convencional, algunas de sus ideas influyeron en los diseñadores de sistemas operativos.*

[V+65] "Structure of the Multics Supervisor" de V.A. Vyssotsky, F. J. Corbato, R. M. Graham. Fall Joint Computer Conference, 1965. *Un artículo inicial sobre Multics, que describía muchas de las ideas y términos básicos que encontramos en los sistemas modernos. Parte de la visión detrás la computación como una utilidad finalmente se están realizando en los sistemas de nube modernos.*

4.8 Tarea (Simulación)

Este programa, `process-run.py`, permite ver cómo cambian los estados del proceso a medida que se ejecutan los programas y usan la CPU (por ejemplo, realizar una instrucción de suma) o realizar operaciones de E/S. (Enviar una solicitud a un disco y esperar a que se complete). Consultar el README para obtener más detalles.

Preguntas

1. Correr `process-run.py` con las siguientes flags: `-l 5:100 5:100` ¿Cuál debería ser el uso de la CPU (el porcentaje de tiempo que la CPU está en uso)? ¿Por qué se sabe esto? Usar las flags `-c` y `-p` para ver si tenías razón.
2. Ahora ejecutar con las siguientes flags: `./process-run.py -l 4:100, 1:0`. Estas flags especifican un proceso con 4 instrucciones (todas para usar la CPU), y uno que simplemente emite una E/S y espera a que se realice. ¿Cuánto tiempo lleva completar ambos procesos? Usar `-c` y `-p` para saber si tenías razón.
3. Modificar el orden de los procesos: `-l 1:0,4:100`. ¿Qué sucede ahora? ¿Importa cambiar el orden? ¿Por qué? (Como siempre, usar `-c` y `-p` para ver si tenías razón).
4. Ahora exploraremos algunas de las otras flags. Una importante es `-S`, que determina cómo reacciona el sistema cuando un proceso emite una E/S. Con la flag en SWITCH_ON_END, el sistema NO cambiará a otro proceso mientras uno está realizando E/S, sino que esperará hasta que el proceso finalice por completo. ¿Qué sucede cuando ejecuta los siguientes dos procesos (`-l 1:0,4:100 -c -S SWITCH_ON_END`), uno haciendo E/S y el otro haciendo trabajo de CPU?
5. Ahora, ejecutar los mismos procesos, pero cambiar a otro proceso siempre que uno esté **esperando** E/S (`-l 1:0,4:100 -c -S SWITCH_ON_IO`). ¿Que pasa ahora? Use `-c` y `-p` para confirmar que tenías razón.
6. Otro comportamiento importante es qué hacer cuando se completa una E/S. Con `-I IO_RUN_LATER`, cuando se completa una E/S, el proceso que la emitió no necesariamente se ejecuta de inmediato; más bien, lo que sea que estaba funcionando en ese momento sigue funcionando. ¿Qué sucede cuando ejecuta esta combinación de procesos? (Ejecute `./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p`) ¿Se están utilizando los recursos del sistema de manera efectiva?

7. Ahora ejecutar los mismos procesos, pero seteando `-I IO_RUN_IMMEDIATE`, que ejecuta inmediatamente el proceso que emitió la E/S. ¿Cómo difiere este comportamiento? ¿Por qué podría ser una buena idea volver a ejecutar un proceso que acaba de completar una E/S?

8. Ahora ejecutar algunos procesos generados aleatoriamente: `-s 1 -l 3:50,3:50` o `-s 2 -l 3:50, 3:50` o `-s 3 -l 3:50,3:50`. Vea si puede predecir cómo resultará la traza. ¿Qué pasa cuando usas la flag `-I IO_RUN_INMEDIATE` vs. `-I IO_RUN_LATER`? ¿Qué sucede cuando usa `-S SWITCH_ON_IO` versus `-S SWITCH_ON_END`?