1 Introducción a los sistemas operativos

Un programa en ejecución toma una instrucción en memoria, la decodifica, y la ejecuta. Una vez esto se realiza, el procesador se mueve a la próxima instrucción, y así sucesivamente. Este es el modelo de Von Neumann.

El **sistema operativo** (SO) es un cuerpo de softwares que facilitan la ejecución de programas, permitiéndoles compartir memoria, interactuar con dispositivos, y simular su ejecución simultánea. La técnica principal que permite conseguir esto se llama **virtualización**.

La virtualización es una técnica general que permita tomar un recurso físico y transformarlo en una representación virtual más'general, poderosa y fácil de utilizar.

(†) **Problema central.** ¿Cómo pueden virtualizarse los recursos? Es decir, ¿qué políticas y mecanismos debe implementar el SO para alcanzar la virtualización?

El sistema operativo provee interfaces (APIs) para que los usuarios se comuniquen con él y le den instrucciones. Entre otras cosas, provee **llamadas a sistema** que están disponibles para las aplicaciones.

Finalmente, el sistema operatitvo es un **administrador de recursos**, decidiendo de manera eficiente y justa cómo deben utilizarse los recursos principales (CPU, memoria, y disco).

1.1 Virtualización de la CPU

La virtualización de la CPU consiste en la simulación de múltiples CPUs con una sola (o pocas). En términos prácticos, esto significa simular que múltiples programas se ejecutan simultáneamente.

(†) Considere el siguiente código.

```
# code name: cpu_virt.c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
        return 1;
    }
```

```
    char *msg = argv[1];

    while (1) {
        printf("%s ", msg);
        fflush(stdout);   // flush so it prints immediately
        usleep(100000);   // sleep 0.1s to let the scheduler switch
    }

    return 0;
}
```

Si ejecutamos `./cpu_virt a  ./cpu_virt b  ./cpu_virt c` , esperaríamos que el primer programa nunca deje que los demás se ejecuten (pues nunca termina). Sin embargo, se imprime primero *a*, después *b*, después *c*, y luego se repite indefinidamente. ¿Qué impide que el primer programa monopolize la CPU? Su virtualización.

La virtualización de la CPU trae problemas de política. Si dos programas quieren ejecutarse al mismo tiempo, ¿cuál priorizar?

1.2 Virtualización de la memoria

El modelo de la memoria es simple. La memoria es una *array* de bytes. Para leerla, uno debe espeficiar una dirección en la array. Para escribir, se especifica una dirección y la información que desea escribirse.

Imaginemos un programa que ejecuta `malloc`, guarando el valor entero 0 en la dirección de memoria p, e imprimiendo p. Imaginemos que inmediatamente después, el programa entra en un loop y actualiza el valor de la dirección p incrementándolo por uno, e imprime el valor resultante. El resultado de ejecutar el programa será la impresión de 1, 2, 3, etc.

Ahora, imaginemos que ejecutamos este programa dos veces, ”al mismo tiempo”. Creeríamos que p, la dirección de memoria de cada instancia del programa, sería diferente. Pero es la misma. Más aún, los programas no sobreescriben la dirección de memoria a la que accede el otro. Es decir, veremos impreso: 1 1 2 2 3 3 etc. Si la dirección p es la misma, ¿por qué los programas no se pisan?

La razón es que el SO virtualiza la memoria. Cada proceso accede a su propio **espacio virtual de direcciones**, que el OS se encarga de mapear a direcciones de memoria física. Una referencia dentro de un programa no afecta el espacio de memoria de otro programa. Hasta donde el programa sabe, él tiene su propia memoria física.

1.3 Concurrencia

El término concurrencia refiere un conjunto de problemas que surgen cuando un único programa opera sobre múltiples recursos concurrentemente. El SO es el principal ejemplo: es un conjunto de programas que manipula al mismo tiempo múltiples cosas. Pero el problema de la concurrencia no se limita al SO: muchos programas *multi-threaded* muestran problemas similares.

2 El proceso (¿de Kafka?)

Una de las formas fundamentales de abstracción que provee el SO es **el proceso**. Un proceso es un programa en ejecución. El programa en sí es algo muerto: instrucciones en memoria, tal vez data estátitca. Es el sistema operativo quien insufla vida.

(†) **Problema central.** ¿Cómo virtualizar la CPU?

El SO virtualiza la CPU ejecutando un proceso, deteniéndolo para ejecutar otro, y así sucesivamente. Esta técnica se llama **time sharing** de la CPU. El costo potencial es eficiencia, pues los programas tardarán más en terminar si la CPU debe compartirse.

Para implementar sus funcionalidades, el SO necesita operar en el bajo bajo y en el alto nivel. Las operaciones de bajo nivel se llaman **mecanismos**. **Time sharing** es un mecanismos.

Por encima de los mecanismos, hay **políticas**. Las políticas son algoritmos que toman decisiones dentro del SO.

2.1 Estado de máquina (machine state)

Para entender un proceso, hay que entender qué es el estado de la máquina. El estado de la máquina es lo que un programa puede leer o actualizar mientras se ejecuta. En un momento dado, ¿qué partes de la máquina importan para la ejecución de un programa?

El espacio de memoria, con sus instrucciones y datos, es parte del estado de máquina. Contiene algunos registros especiales, como el program counter (PC), también llamado instruction pointer (IP), que nos dice qué instrucción del programa se ejecutará en el instante siguiente. Similarmente, está el stack pointer y el frame pointer asociado, que sirven para administrar el stack con las variables, parámetros de funciones, y return addresses.

2.2 API de procesos

Todo SO moderno provee las siguientes APIs:

- **Crear**. El SO debe permitir la creación de procesos nuevos.
- **Destruir**. El SO debe permitir la destrucción de procesos.
- **Espera**. El SO debe poder esperar que un proceso termine.

- **Control misceláneo.** Controles varios, como suspender un proceso para resumirlo después.
- **Status.** El SO debe dar una interfaz para obtener información acerca del estado de un proceso: hace cuánto se ejecuta, por ejemplo.

2.3 Creación de procesos

¿Cómo se transforma un programa en un proceso? El SO carga su código y cualquier información estática necesaria desde el disco en la memoria virtual. Esto suele hacerse *lazily*: se van cargando en la memoria las partes a medida que se las van necesitando en la ejecución. Luego el SO aloca memoria para el stack del programa, donde residen las variables locales, parámetros de funciones, y direcciones de retorno. El SO también aloca memoria para el heap, donde reside la información dinámica (lo alocado y liberado por `malloc` y `free`, por ejemplo). Aquí viven las estructuras de datos como arrays, árboles, etc.

El SO también se encarga de otras tareas de inicialización, en particular relativas al input/output (I/O). Por ejemplo, cada proceso tiene tres **file descriptors**: `stidin, stdout, stderr`. Éstos permiten al programa leer input e imprimirlo o dirigirlo a alguna fuente.

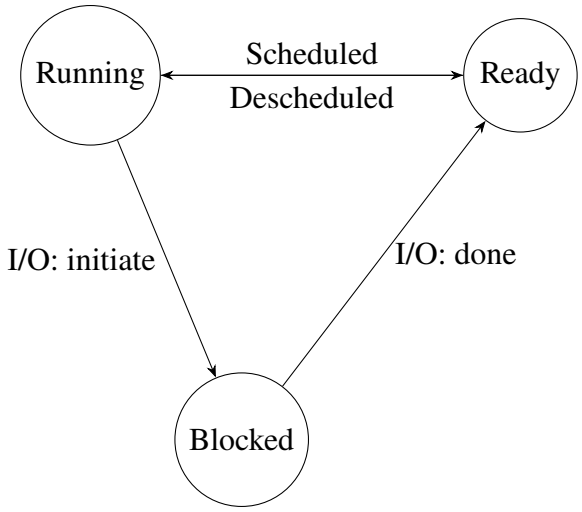
Sólo después de todo esto, el SO comienza la ejecución, iniciando el programa desde su entry point (i.e. `main()`). Una vez "salta" a la rutina `main()`, el SO transfiere control de la CPU al proceso recién creado, y el programa comienza.

2.4 Estados de procesos

Un proceso puede estar en uno de los siguientes estados:

- **Running:** El programa se está ejecutando.
- **Ready**: El programa está listo para ejecutarse.
- **Blocked:** El programa está bloqueado, es decir a realizado algún tipo de operación y no puede continuar hasta que cierto evento ocurra. Por ejemplo, solicitó I/O en el disco y se bloquea hasta que la operación de I/O termine.

Cuando un programa pasa de **Running** a **Ready**, decimos que fue *descheduled*. En la transición inversa, decimos que fue *scheduled*.



La parte del SO que se encarga de administrar estas transiciones de estado en los distintos programas se llama **OS scheduler**.

A veces un proceso está en estados que llamamos **inicial** y **final**, los estados resultanates justo cuando es creado y justo cuando termina, respectivamente. Cuando un proceso termina pero aún no fue destruido, decimos que está en **estado zombie**. El estado zombie permite que otros procesos (como el proceso padre) examinen el código de retorno del proceso que terminó y ver si se ejecutó con éxito.

2.5 Estructuras de datos del SO

El SO es un programa con sus propias estructuras de datos. Entre ellas, destacarmemos las siguientes.

El SO tiene una **lista de procesos**, con todos los procesos que están en estado *ready* y alguna información adicional para registrar cuál se está ejecutando. El SO también lleva algún registro de los procesos bloqueados.

El SO también tiene, para cada proceso bloqueado, un **register context**. Dicho registro guarda el contenido de los registros del proceso bloqueado. Así, al resumir el proceso, sus registros pueden restaurarse.

3 API de procesos

3.1 La llamada a sistema `fork()`

Todo proceso tiene un **PID** (process identifier) y un *process group ID* (PGID). El ID identifica al proceso, y el PGID se usa para identificar procesos que están relacionados (e.g. los procesos hijos heredan el PGID del padre).

. La llamda a sistema `fork()` se utiliza para crear un proceso nuevo, pero de una forma un tanto bizarra.

`fork()` crea una copia *casi* exacta del proceso dentro del cual ocurre la llamada. Dicha copia se llama *proceso hijo*, y el proceso que dentro del cual se llamó `fork()` se llama *proceso padre*. La ejecución del proceso hijo no comienza en el entry point `main()`, sino justo donde la llamada a `fork()` termina.

El hijo no es una copia exacta por dos razones. Por un lado, tiene su propio espacio de direcciones, sus propios registros, su propio PC, etc. Por otro lado, la llamada a `fork()` en el padre devuelve el PID del proceso hijo (un `int`). Por otro lado, en el hijo la llamada a `fork()` devuelve `0` si el proceso se creó con éxito, y `-1` de otro modo.

Ejemplo. El siguiente código explica con comentarios lo antedicho.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

int main() {
    // pid_t es el data type usado para representar PIDs. Suele ser un signed
    // int, aunque esto puede variar.
    pid_t pid;

    // Crea un proceso hijo. En esta línea empieza el hijo,
    // y el valor de pid diferirá en padre e hijo.
    pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    else if (pid == 0) {
        printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());
    }
    else {
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
```

```
    }

    // Padre e hijo, ambos, ejecutarán la siguiente línea.
    printf("Soy ejecutado por ambos procesos. (PID = %d).\n", getpid());

    return 0;
}
```

La ejecución de `fork()` es **no-determinística**: cuando se crea el hijo, dos procesos activos (padre e hijo) compiten por la CPU, y cualquiera de los dos puede ser ejecutado dependiendo de la decisión tomada. El **scheduler** determinará quién se ejecuta primero, pero esto no podemos determinarlo de antemano.

3.2 La llamada a sistema `wait()`

Generalmente, es útil pausar el proceso padre hasta que el proceso hijo termine. Esto puede hacerse con `wait()`. La llamada a `wait()` combinada con `fork()` hace que `fork()` sea determinístico.

En la práctica, `wait()` tiene dos comandos asociados, `waitpid()` y `waitid()`. Todos estos comandos se usan para esperar un **cambio de estado** en un proceso hijo (el hijo terminó, fue interrumpido por una señal, fue resumido por una señal). Cuando un hijo termina, es la llamada a `wait()` lo que permite liberar los recursos asociados al hijo; de otro modo, el hijo quedaría en un estado zombie.

La llamada `waitpid()` es un wrapper sobre `wait()`. En particular, `wait()` tiene la siguiente signature:

```
pid_t wait(int *_Nullable wstatus)
```

`wstatus` es un puntero a un `int`, y se corresponde con la dirección de memoria donde se escribirá qué sucedió con el proceso hijo. En general, en `wstatus` se escribe la PID del proceso hijo cuando el mismo terminó con éxito, y se escribe `-1` si hubo un error. El puntero `wstatus` puede ser nulo (de allí el `_Nullable`), lo cual significa: ”no me importa qué suceda con el proceso hijo”.

La llamada `waitpid()` tiene la siguiente signatura:

```
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options)
```

Esta llamada suspende la ejecución hasta que el proceso hijo con PID `pid` cambie de estado. Por

defecto, `waitpid()` espera solo terminaciones y no otros cambios de estado, como interrupciones. Este comportamiento se puede modificar vía el argumento `options`.

El argument `pid` puede tomar los siguientes valores:

- Valores menores a `-1`: Espera a cualquier proceso con PGID idéntico al valor absoluto de `pid`.
- `-1`: Espera por cualquier proceso hijo.
- `0`: Espera a cualquier proceso cuyo PGID es idéntico al del proceso llamador, *en el momento en que se llamó* `waitpid()`.
- Valores mayores a `0`: Espera por el proceso hijo cuyo PID es exactamente igual al valor de `pid`.

Como los hijos heredan el PGID del padre, pareciera que las opciones `<-1` y `=-1` son idénticas. Pero el PGID de un proceso puede cambiarse. Por ejemplo, un proceso puede tener dos hijos, uno de los cuales es asignado en algún momento un PGID diferente (ver `setpgid()`). Entonces, `waitpid(-1, ...)` esperará a cualquiera de los hijos, mientras que `waitpid(-2, ...)` esperará solo al que sigue teniendo una PGID idéntica a la del padre (en valor absoluto).

El valor de `options` es un `OR` de cero o más constantes, entre las cuales contamos: `WNOHANG`, que retorna inmediatamente si ningún hijo ha exited (?), `WUNTRACED`, que también retorna si un hijo se ha detenido (no terminado), y `WCONTINUED`, que también retorna si un hijo detenido ha sido resumido. Más info en `man wait`.

Notar que si `wstatus` es un `int`,

$$\texttt{wait(wstatus)} \equiv \texttt{waitpid(-1, wstatus, 0)}$$

Es decir, `wait(status)` expresa: Esperá por cualquier proceso hijo (`pid = - 1`), guardá la información de la espera en `wstatus`, sin ninguna flag en particular (`options = 0`).

3.3 La llamada a sistema `exec()`

La llamada `exec()` permite ejecutar, desde un proceso llamador, un programa diferente al mismo. En realidad, `exec` es parte de una familia de programas: `execl, execlp, execle,` etc. Veremos en particular `execv, execvp`.

La familia de programas `exec()` *reemplaza* la **imagen** del proceso actual por una nueva.

(♣) La **imagen** de un proceso es el estado total de un proceso en memoria: su código, sus variables, su heap, su stack, sus file descriptors, etc.

Cuando se llama `exec()` exitosamente, el proceso ya no corre su código viejo, sino que se convierte en el programa a ejecutar. El PID permanece idéntico, los file descriptors siguien abiertos, pero la memoria, el stack, y el código son reemplazados. Por esta razón, `exec()` suele usarse después de `fork()`, transformando el proceso hijo en un nuevo programa, y lograndoo así ejecutar programas nuevos desde un programa padre.

4 Ejecución directa limitada

La ejecución directa de los programas no es deseable. Si la CPU ejecuta un programa de manera directa y libre, ¿cómo garantizar que el programa no haga algo indeseado? ¿Cómo lo detenemos, o cómo implementamos **time sharing**, si el programa tiene control de la CPU?

La ejecución directta, por ende, debe limitarse. Llamamos al paradigma de ejecución resultante **ejecución directa limitada**.

4.1 Problema 1: Restricción de operaciones

La ejecución directa tiene como ventaja ser rápida: el programa corre nativamente en la CPU (hardware). El primer problema que debemos enfrentar, sin embargo, es permitir que el programa pueda realizar operaciones restringidas, como I/O, sin tener control completo de la CPU.

La solución es introducre dos **modos de procesador**: *user mode* y *kernel mode*. El primero es restringido, mientras el segundo es privilegiado.

Cuando un programa desea realizar una operación privilegiada, como leer del disco, se utiliza una **llamada a sistema**. Las llamadas a sistema permiten que el *kernel* exporte ciertas funcionalidades esenciales a los programas del usuario, como acceder a un archivo del sistema, crear y destruir procesos, alocar memoria, etc.

Para ejecutar una llamada a sistema, el programa jecuta una **trap instruction**. Esta instrucción simultáneamente salta al kernel y eleva el nivel de privilegio al *kernel mode*. Una vez en este estado, el sistema puede realizar la operación privilegiada deseada. Al terminar, el SO ejecuta una **return-from-trap instruction**, que vuelve al programa original y a la vez reestablece el *user mode*.

Al realizar la **trap instruction**, el procesador guarda en el **kernel stack** los registros, PC y flags del programa, a fin de reestablecer una vez que se vuelve del kernel.

¿Cómo sabe la **trap** qué codigo ejecutar dentro del SO? El proceso no puede decirle a qué instrucción ir, porque eso permitiría que los programas vayan a cualquier lugar del kernel (malísimo). Por el contrario, cuando se inicia la computadora (en kernel mode), el kernel establece una **trap table**, informande al hardware en qué lugar de memoria residen los **trap handlers**, i.e. las instrucciones que se encargan de ejecutar una llamada a sistema. Por ende, cuando una **trap instruction** ocurre, el hardware ya sabe dónde residen los **handlers**.

En particular, cada llamada sistema tiene un **system-call number**, un identificador. El SO, cuando maneja una llamada a sistema dentro de un trap handler, ve si el identificador es válido y, si lo es, ejecuta el código correspondiente. Este nivel añadido de indirección provee protección: el código de usuario no especifica a qué dirección saltar, sino que solicita un servicio a través de un número identificador.

De este modo, la ejecución directa limitada tiene dos fases. En boot time, el kernel inicializa la trap table y la CPU recuerda su ubicación. Luego, cuando un proceso se ejecuta, el kernel organiza algunas cosas (alocar un nodo en la lista de procesos, alocar memoria) y luego usa una return-from-trap instruction para empezar la ejecución del proceso. Esto hace que la CPU pase a user mode y la ejecución comienze. Si el proceso desea hacer una llamada a sistema, se vuelve al SO vía una trap instruction, y el SO maneja la llamada y luego retorna al programa vía una return-from-trap.

4.2 Problema 2: Cambio entre procesos

Si un proceso se ejecuta en la CPU, por definición se sigue que el SO no se está ejecutando. ¿Cómo puede el SO cambiar de un proceso a otro entonces?

4.2.1 Approach cooperativo

Una estrategia se denomina **cooperativa**. El SO confía en que el proceso será razonable y devolverá el control eventualmente para que el SO decida cómo continuar. Esto suele hacerse mediante una llamda especial, i.e. **yield**. También se asume que se devuelve el control al SO cuando se hace una operación ilegal (e.g. div. por cero o acceder a memoria privilegiada). En esta estrategia, entonces, el SO retoma control porque los programas lo ceden.

Problema obvio: programas maliciosos, loops infinitos, etc.

4.2.2 Approach no cooperativo

La pregunta central es cómo puede el SO retomar control sin cooperación de los programas. La respuesta es simple: se usa un **timer interrupt**, es decir un contador que llama una interrupción cada k milisegundos. Cuando una interrupción se llama, un **interrupt handler** pre-configurado en el SO toma control, y el SO decide qué hacer. El código a ejecutar cuando se produce el timer interrupt se define en boot time, donde también se inicia el timer.

4.3 Guardando y restableciendo contextos

Ya tenemos un sistema que permita el SO retomar control. Para decidir qué hacer cuando el control es retomado, el SO usa un **scheduler**. Lo que nos interesa ahora es qué sucede cuando el **scheduler** decide cambiar el programa que se está ejecutando.

Cuando esto sucede, el SO ejecuta un código de bajo nivel que llamaremos **context switch**. Conceptualmente, guarda los valores de registros del proceso actual dentro de su entrada correspondiente

en el kernel stack (una por proceso). Luego se encarga de que la return-from-trap no rediriga al proceso actual, sino al que ahora debe ejecutarse. El kernel stack pointer se mueve al stack de este proceso sucesor. Así, se entra al kernel en el contexto de un proceso (el interrumpido) y se sale en el contexto de otror (el sucesor).

Notar que hay dos tipos de guardados/restablecimientos ocurriendo. El primero es cuando ocurre el timer interrupt: los *user registers* del proceso interrumpido son guardados en el hardware, en el kernel stack de ese proceso. Luego, el SO switchea al sucesor: los *kernel registers* son guardados por el SO en memoria, lo cual significa que ahora el sistema está en un estado idéntico al que resultaría si hubiera realizado la trap instruction desde el proceso sucesor. Por esto es que la return-from-trap instruction nos lleva al sucesor, que ahora empieza a ejecutarse.

5 Scheduling

Scheduling se refiere a las políticas tomadas para asignar recursos (principalmente la CPU) a los procesos que los demandan. Pueden compararse usando métricas de performance o de justicia (fairness).

5.1 Supuestos sobre el workload

Al conjunto de procesos que corren en un sistema se lo denomina colectivamente *workload*. Dependiendo de los supuestos que uno haga respecto del *workload*, distinas políticas serán posibles.

Por ahora, asumimos:

1. Todo proceso se ejecuta por la misma cantidad de tiempo.
2. Todos los procesos llegan al mismo tiempo.
3. Una vez iniciado, un proceso se ejecuta hasta termina.
4. Todos los procesos sólo usan la CPU (nada de I/O).
5. El tiempo de ejecución de cada proceso es conocido *a priori*.

5.2 Métricas

Para comparar distintas políticas, necesitamos métricas. La primera que usaremos se denomina **turnaround time**:

$$T_{\text{turnaround}} = \text{T}_{\text{terminación}} - T_{\text{llegada}}$$

Como hemos asumido que todos los procesos llegan al mismo tiempo, por ahora el tiempo de llegada es cero y por lo tanto el turnaround time es simplemente el tiempo en que el proceso terminó.

5.3 Política FIFO

La política FIFO es razonable bajo nuestros supuestos. Si A, B, C llegan en ese orden y al mismo tiempo $T_{\text{llegada}} = 0$, y duran cada uno 10s, entonces el tiempo promedio de turnaround será

$$\frac{\text{Turnaround}(A) + \text{Turnaround}(B) + \text{Turnaround}(C)}{3} = \frac{10 + 20 + 30}{3} = 20$$

Pero si relajamos el supuesto (1) y admitimos que los procesos pueden durar una cantidad distinta de tiempo, FIFO puede ser muy malo. Por ejemplo, si A dura 100s mientras B y C siguen durando 10, el tiempo promedio de turnaround será 110, porque B y C (que son breves) deben esperar que A termine.

A esto se le llama **convy effect**: cierta cantidad de consumidores relativamente breves de un recurso quedan esperando detrás de un consumidor excesivo.

5.4 Shortest Job First (SJF)

Como asumimos que los procesos llegan al mismo tiempo y que conocemos su duración, otra política posible es ordenarlos y correr los más cortos primero (con algún criterio arbitrario para los empates). Entonces, si A dura 100 y B, C duran 10, el tiempo promedio de turnaround es de 50, porque

$$\text{Turnaround}(A) = 10, \qquad \text{Turnaround}(B) = 20,\ \text{Turnaround}(C) = 120$$

Puede demostrarse que bajo los supuestos dados, SJF es un algoritmo de scheduling **óptimo**. El problema es que si relajamos el supuesto (2) y pensamos que los procesos pueden llegar en cualquier momento, tenemos problemas. Por ejemplo, si A llega un poquito antes que B y C, otra vez sucederá que B y C deberán esperar que A termine.

5.5 Shortest time-to-completion first (STCF)

Para resolver el problema, debemos relajar el supuesto (3) y permitir que los programas puedan detenerse en vez de ejecutarse siempre hasta terminar.

Si B y C llegan después de A, la idea es que el scheduler pueda pausar A y decidir ejecutar alguno de los otros programas, tal vez continuando A después. Si añadimos esta facultad a SJF, tendremos STCF. Cada vez que un nuevo proceso entra al sistema, el scheduler determina cuál de todos los procesos (incluyendo el que está ejecutándose) tiene el menor tiempo restante, y prioriza ese. Así, STCF pausaría A priorizando B y C, y sólo cuando éstos terminan continuaría corriendo A.

Por ejemplo, asumamos que A llega en $t = 0$ y B y C llegan en tiempo $t = 10$, donde A dura 100s y los demás 10s. Los turnaround son:

1. B llega en $t = 10$ y termina en $t = 20$, así que su turnaround es 10.

2. C llega en $t = 10$ y termina en $t = 30$, su turnaround es 20.

3. A llega en $t = 0$ y se ejecuta por diez segundos, tras lo cual pausa 20 segundos mientras A y B corren, y luego hace sus 80 segundos restantes. O sea que A termina en $t = 120$ y su turnaround es 120.

El turnaround promedio entonces es $(20 + 30 + 120)/3 = 50$. Lo cual no está mal.

El problema con STCF es que no siempre tiene un buen tiempo de respuesta, donde

$$\mathrm{T}_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$$

j

El tiempo de respuesta (**response time**) es el tiempo que pasa entre el momento en que el proceso llega y el momento en que es scheduled. Si tres procesos llegan al mismo tiempo, el tercero debe esperar que los dos primeros terminen en su totalidad antes de arrancar. Esto destruye la interactividad.

5.6 Round Robin (RR)

La idea del algoritmo de Round-Robin es no ejecutar los procesos hasta que terminan, sino durante cierta ventana de tiempo denominada **quantum**, pasando luego al próximo programa en la cola. Hace esto repetidamente hasta que todos los trabajos terminan.

(†) Notar que el **quantum** debe ser un múltiplo del timer interrupt.

Ejemplo. Asuma que A, B, C llegan al mismo tiempo y desean correr por 5s. Asuma que el quantum es 1s. Entonces se ejecuta A por 1s, B por 1s, C por 1s, y se repite, así 5 veces. El response time de A es cero, el de B es 1 y el de C es 2, dando un promedio de 1. Observar que en SJF el response time promedio sería $(0 + 5 + 10)/3 = 5$.

¿Y el turnaround time? Todos los procesos llegan en $t = 0$, y cada uno se ejecuta una vez cada tres segundos, necesitando cinco segundos en total para terminar. Por lo tanto, A se ejecuta en los tiempos $t_A = \{0, 3, 6, 9, 12\}$, B en los tiempos $t_B = \{1, 4, 7, 10, 13\}$, y C en los tiempos $t_C = \{2, 5, 8, 11, 14\}$. Por ende, A termina en el tiempo 13, B en el 14, y C en el 15. El promedio de turnaround time es 14: ¡malardo!

El quantum es crítico: cuanto menor sea, mejor la performance bajo la métrica de response time. Sin embargo, si es muy breve, el context switch dominará la performance. (Imagine el ejemplo extremo: un quantum menor a la duración del context switch!)

Por eso se deseas **amortizar** el costo de hacer cambio de contexto, encontrar un trade-off que permita rapidez sin dejar que el cambio de contexto domine.

5.7 Incorporando I/O

Cuando un programa solicita una operación de I/O al kernel, pasa al estado **blocked** esperando que termine la solicitud. El proceso puede ser relativamente largo, y por ende el scheduler podría decidir encolar otro proceso mientras se espera que la operación de I/O termine.

6 Multi-level Feedback Queue

El problema a resolver es: optimizar el *tiempo de retorno* (tiempo que un proceso pasa en el sistema) mientras se minimiza el *tiempo de respuesta* (logrando así interactividad), y hacerlo bajo supuestos débiles.

La *cola multinivel con retroalimentación* (MLFQ) manejará varias colas distintas, cada una asignada a un **nivel de prioridad**. En todo momento, cada proceso pertenece a una y solo una cola.

Dentro de cada cola, usamos *round-robin*. Entre colas, usamos el nivel de prioridad de las colas. Así, para dos procesos A, B, las dos primeras reglas son:

1. Si $Pr(A) > Pr(B)$, A se ejecuta y B no.
2. Si $Pr(A) = Pr(B)$, entonces A y B se ejecutan en RR.

Obviamente, si las prioridades fueran constantes, el sistema no tendría sentido. Por ello, la prioridad de un trabajo varía según su comportamiento observado. Si un proceso cede repetidamente la CPU mientras espera entrada desde el teclado, su prioridad será alta. Si un proceso usa CPU intensivamente por mucho tiempo, MLFQ reducirá su prioridad. Así, MLFQ tratará de aprender acerca de los procesos mientras ellos corren, usando el pasado para predecir el futuro.

6.1 Cambios de prioridad

En el workload, tal vez hayan procesos breves e interactivos que ceden la CPU frecuentemente, y procesos codiciosos (CPU-bound) que necesitan un uso largo de la CPU pero no requieren interactividad.

Llamamos **allotment** a la cantidad de tiempo que un proceso puede durar en cierta prioridad k antes de que el scheduler cambie su prioridad. Por simplicidad, asumiremos por el momento que el allotment es igual al quantum. Un intento de algoritmo para ajustar prioridades es el siguiente:

- Cuando un proceso ingresa al sistema, se le asigna prioridad máxima.
- Si usa todo su allotment mientras se ejecuta, bajamos su prioridad.
- Si el proceso cede la CPU antes de su allotment, se queda en el mismo nivel de prioridad.

Analicemos algunos casos. Imaginemos un único proceso codicios que dura en un sistema con un quantum (y por ende un allotment) de 10ms. Imaginemos que hay tres queues de prioidad. El proceso empieza en la más alta, baja después de un quantum a la segunda, y baja después de otro quantum a la tercera, y permanece en ella hasta que termina.

Ahora imaginemos dos procesos: A, codicioso y largo (100ms), y B, que es breve e interactivo (20ms). Asumamos que A ha estado corriendo por un tiempo mayor a 20ms y ahora ingresa B al sistema. Para cuando B ingresa, A ya está en la cola de más baja prioridad, y B es insertado en la máxima prioridad. B hace sus primeros 10ms en la máxima prioridad, y sus últimos 10ms en la segunda prioridad, y nunca llega a la más baja. Cuando B termina, A vuelve a comenzar.

(†) Fijate lo que hace MLFQ en ese último ejemplo: no sabe si B, al ingresar, es breve o no, pero asume que lo es. Si no lo es, irá bajando su prioridad. En este sentido, MLFQ se aproxima a SJF.

Imaginemos ahora un proceso B con I/O. Una regla es que si un proceso cede la CPU antes de su allotment, queda en la misma prioridad. Imaginemos que B es breve e interactivo, y un proceso A codicioso y largo se estuvo ejecutando hace rato cuando B entra al sistema. A ya está en la última cola, y cuando B ingresa en máxima prioridad toma la CPU y la cede antes del allotment. B sigue haciendo esto, y queda siempre en la máxima prioridad, y A se ejecuta sólo en los espacios de tiempo en los que B cede la CPU. Así, MLFQ logra interactividad y eficiencia.

6.2 El boost de prioridades

Las reglas presentadas antes tienen problemas. Uno de ellos es **starvation**: si hay demasiados procesos breves e interactivos, los codiciosos y largos nunca recibirán la CPU.

Otro problema es que un usuario maligno podría escribir un programa que hackee el scheduler. Por ejemplo, un programa que justo antes del allotment haga un llamado I/O trivial (e.g. `open` un archivo), soltando momentáneamente la CPU. De este modo, se quedaría siempre en la máxima prioridad y tener un porcentaje de uso de CPU mayor. Si esto se hace bien (e.g. haciendo la operación I/O después de que pasó 99% del tiempo de allotment), el proceso podría monopolizar la CPU.

Por último, un programa puede cambiar su comportamiento. Un programa codicioso puede pasar a una fase interactiva, y necesitaría poder subir de prioridad.

Para lidiar con estos problemas, añadimos una nueva regla:

- Después de cierto tiempo S, todos los procesos se mueven a la más alta prioridad.

Notar que esta regla resuelve el problema de **starviation**, pues todo proceso después de una cantidad de tiempo S alcanzará máxima prioridad y recibirá CPU según el esquema de Round-Robin. También resuelve el problema de boostear y reorganizar prioridades. El único problema que falta resolver es el hackeo.

Este problema se hace *acumulando* el allotment time. Es decir, descendemos un proceso si ha usado su allotment, independientemente de si lo usó en una sola ejecución corrida o en una ráfaga de varias ejecuciones distintas.

6.3 Resumen de las reglas

La MLFQ final usa las siguientes reglas:

1. Si la prioridad de A es mayor a la de B, ejecutar A.
2. Si tienen la misma prioridad, usar Round-Robing en la cola.
3. Cuando un proceso entra, se le asigna máxima prioridad.
4. Cuando un proceso usa todo su allotment (no importa cuántas veces haya cedido la CPU), se reduce su prioridad.
5. Después de un periodo de tiempo *S*, todos los procesos son lifteados a la máxima prioridad.

7 Ejercicios: Virtualización de la CPU

(1) En un sistema operativo que implementa procesos se ejecutan instancias del proceso `pi` que computa los dígitos de π con precisión arbitraria.

```
$ time pi 1000000 > /dev/null & ... & time pi 1000000 > /dev/null &
```

Y se registran los siguientes resultados, donde en las mediciones se muestra *(real, user)*, es decir el tiempo del reloj de la pared (walltime) y el tiempo que insumió de CPU (cputime).

#Instancias	Medición	Descripción
1	(2.56, 2.44)	
2	(2.53, 2.42), (2.58, 2.40)	
1	(3.44, 2.41)	
4	(5.12, 2.44), (5.13, 2.44), (5.17, 2.46), (5.18, 2.46)	
3	(3.71, 2.42), (3.85, 2.42), (3.86, 2.44)	
2	(5.04, 2.36), (5.09, 2.43)	
4	(7.67, 2.41), (7.67, 2.44), (7.73, 2.44), (7.75, 2.46)	

(a) ¿Cuántos núcleos tiene el sistema?

(b) ¿Por qué a veces el cputime es menor que el walltime?

(c) Indique en la **Descripción** qué estaba pasando en cada medición.

El CPU time se mantiene prácticamente constante, como es esperable. La segunda fila indica que hay al menos ≥ 2 cores, porque con dos procesos el CPU time y el walltime siguen siendo aproximadamente iguales. El hecho de que un solo proceso en la tercera medición tiene $real > user$ indica que la medición se llevó a cabo mientras otros procesos eran ejecutados. Esto también explica que $real > user$ en la penúltima medición.

La cuarta medición es interesante, porque con cuatro procesos tenemos $real \approx 5s$, es decir aproximadamente el doble de lo que toma una sola instancia aislada. Esto soporta la idea de que la máquina tiene = 2 cores.

Bajo la misma línea, en la tecera medición, tres instancias toman $real \approx 3.80s$, es decir ≈ 1.5 veces lo que toma una sola instancia. Esto soporta la idea de que hay < 3 cores, y en particular de que hay exactamente 2, porque $3/2 = 1.5$.

(2) En un sistema operativo que implementa procesos e hilos se ejecutan el siguiente proceso.

```
$ time ./dgemm 2000 2000 2000
test!
m=2000,n=2000,k=2000,alpha=1.200000,beta=0.001000,sizeofc=4000000
real 0m1.027s
user 0m1.752s
```

Explique porque ahora $wall < cpu$.

Cuando usamos k hilos, el cpu time se mide como

$$\text{CPUTime} = \sum_{i=1}^{k} T(\text{thread}_i)$$

donde $T(\text{thread}_i)$ es el tiempo utilizado en el hilo i. Sin embargo,

$$\text{Walltime} \geq \max\{T(\text{thread}_i) : 1 \leq i \leq k\}$$

porque una vez que todos los hilos terminan se puede devolver el resultado y terminar el proceso. Más aún, si la paralelización es buena,

$$\text{Walltime} \approx \max\{T(\text{thread}_i) : 1 \leq i \leq k\}$$

Claramente, acá puede acontecer que $wall < cpu$. Por ejemplo, si cada hilo uno de tres hilos toma $0.5s$, el cpu time es 1.5 pero el wall time es 0.5.

(4) Un programa define la variable `int x=100` detro de `main()` y hace `fork()`. (a) ¿Cuánto vale `x` en el proceso hijo? (b) ¿Qué le pasa a la variable cuando el proceso padre y el proceso hijo le cambian de valor? (c) Contestar nuevamente las preguntas si el compilador genera código de máquina colocando esta variable en un registro del microprocesador.

(a) En el proceso hijo, `x` vale `100`, porque el hijo es una copia exacta del padre excepto su PID.

(b) La variable `x` cambia de valor independientemente en ambos procesos. Si el padre la cambia, el cambio no se ve reflejado en el proceso hijo, y viceversa.

(c) Recordemos que los registros son ubicaciones de memoria de muy rápido acceso que están directamente en la CPU. Cuando un proceso se ejecuta, estos registros contienen información específica de ese proceso. Cuando se produce un context switch, los valores de los registros se guardan en el PCB (process control block) para poder ser restaurados luego, y se reemplazan con los valores del proceso sucesor.

Supongamos que el compilador genera código de máquina guardando la variable en un registro especial R dentro del microprocesador, que *no forma parte del contexto de cada proceso* (es decir, no se guarda ni restaura durante un context switch).

Bajo este supuesto hipotético:

1. Después de `fork()`, tanto el proceso padre como el hijo tienen inicialmente `x = 100`, ya que el hijo copia el valor inicial.
2. Si el proceso padre cambia `x`, ese cambio se refleja inmediatamente en el proceso hijo, y viceversa, porque ambos están accediendo al mismo registro físico R.
3. Esto rompe el comportamiento normal de variables locales tras `fork()`, pero es consistente con la hipótesis de un registro "fuera del user space" que no se guarda en el PCB.

Nota: Este escenario es puramente teórico y no ocurre en sistemas operativos reales, donde todos los registros de usuario se guardan y restauran durante los context switches.

(5) Indique cuantas letras "a" imprime este programa, describiendo su funcionamiento.

```
printf("a\n");
fork();
printf("a\n");
fork();
printf("a\n");
fork();
printf("a\n");
```

Generalice a n forks. Analice para $n = 1$, luego para $n = 2$, etc., busque la serie y deduzca la expresión general en función de n.

Hay un impresión de *a* antes de la primer llamada a `fork()`.

Luego se llama `fork()`, con lo cual hay un hijo y un padre, y ambos ejecutan la segunda instrucción `print`, con lo cual se imprime *a* dos veces más.

Tanto el hijo como el padre que hasta ahora tenemos llegan al segundo `fork`, con lo cual los dos generan un hijo. Esto quiere decir que ahora el hijo que ya existía tiene un nuevo hermano y un hijo propio, resultando en un total de 4 procesos. Con lo cual se imprimen 4 *a*s.

En el último fork, el padre original tiene un nuevo hijo (ya lleva tres es en total). Su primer hijo genera un nuevo hijo (teniendo dos en total), y también lo hace su segundo hijo (teniendo uno en total). Por otra parte, el nieto del proceso original (el primer hijo del primer hijo) también genera un hijo nuevo. Esto nos da:

1 (Proceso padre) + 3 (Sus tres hijos) + 3 (sus tres nietos) + 1 (su bisnieto) = 8.

Por ende, se imprime *a* ocho veces, y la cantidad de impresiones totales son 1 (la inicial) + 2 + 4 + 8 = 15.

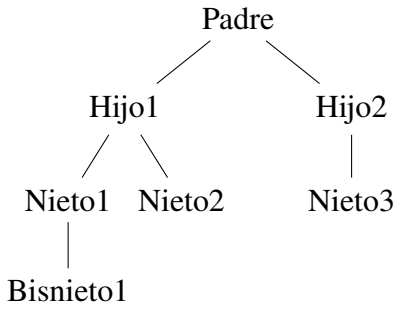


Figure 1: Diagrama de procesos de tres llamadas a `fork()`

En el diagrama se hace claro que cada llamada a `fork()` agrega un nodo hijo *a todos los nodos existentes*.

Esto significa que si hay n procesos (nodos), cada uno de los cuales ejecuta `fork()`, después de dicha ejecución habrán n procesos más, es decir $n + n = 2n$ procesos. La secuencia por ende es dada por un caso base de un proceso y un caso inductivo de $2k$ procesos. Fácilmente se ve que esto resulta en la regla general de 2^n procesos después de una llamada a fork.

Fácilmente vemos que esto coincide con la cantidad de *a*s escritas: $2^3 + 1$, donde sumamos uno para contar la vez anterior a todas las llamadas de `fork()`.

(6) Indique cuantas letras “a” imprime este programa:

```
char * const args[] = {"/bin/date", "-R", NULL};
execv(args[0], args);
printf("a\n");
```

El programa ejecuta `date` vía `execv`. Después de la llamada a `execv`, la imagen del proceso es reemplazada y el proceso se ”convierte” en `execv`. Por ende, la instrucción `printf("a")` nunca es alcanzada. ∴ El programa imprime tantas letras ”a” como haya en el output de `date`, asumiendo que `execv` no falla.

(7) Indique que hacen estos programas.

```c
int main(int argc, char ** argv) {
    if (0<--argc) {
        argv[argc] = NULL;
        execvp(argv[0], argv);
    }

    return 0;
}
```

```c
int main(int argc, char ** argv) {
    if (argc<=1)
        return 0;

    int rc = fork();
    if (rc<0)
        return -1;
    else if (0==rc)
        return 0;
    else {
        argv[argc-1] = NULL;
        execvp(argv[0], argv);
    }
}
```

(*a*) En el primer código, podríamos bien sustituir `0 < --argc` por `argc >= 2`, porque

$$0 < \#\text{args} - 1 \iff 1 < \#\text{args} \iff 2 \leq \#\text{args}$$

Pero el Wolopriest nos quiere confundir.

Recordemos que `argc` es la cantidad de command line arguments dados al programa más uno (pues también cuenta el nombre del programa), y que `argv` es una null-terminated array de `argc + 1` elementos.

Asumiendo que la condición se cumple (es decir, que al programa se le pasa al menos un argumento), debemos notar que la línea `argv[arc] = NULL` no hace nada, porque el último elemento de la array ya es `NULL`. Es decir que el programa equivale a llamar `execvp(argv[0], argv)`. Por ejemplo, si el programa se ejecuta con los arguments `ls -l`, se ejecuta el programa `ls` con argumentos `-l`. Etc.

(*b*) Este programa también requiere `argc >= 2` para hacer algo efectivo. Asumamos que ese es el caso. El programa `fork`ea. Si el `fork` falla, devuelve `-1`. Si el fork no falla, devuelve `0` en el hijo, pero en el padre ejecuta el programa indicado por los command line arguments con la última flag removida. (Notemos que `argv[argc - 1] = NULL` no hace más que quitar la última flag). Para usar el mismo ejemplo que antes, en este caso llamar el programa con `ls -l` como command line arguments resultaría en la ejecución de `ls`.

(8) Si estos programas hacen lo mismo. ¿Para que está la syscall dup()? ¿UNIX tiene un mal diseño de su API?

```
// Programa 1
close(STDOUT_FILENO);
open("salida.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
printf("¡Mira mama salgo por un archivo!");


// Programa 2
fd = open("salida.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
close(STDOUT_FILENO);
dup(fd);
printf("¡Mira mama salgo por un archivo!");
```

Estudiemos estos programas.

(1) Recordemos que `STDOUT_FILENO == fileno(stdout)`, es decir es el `int` file descriptor usado para implementar el stream `stdout` (que al ser un stream es un `FILE*` pointer).

La primera línea cierra el file descriptor que identifica a `stdout`, después de lo cual `STDOUT_FILENO` (que en realidad no es más que el entero 1) no identifica ninguna file y puede ser reusado.

La función open abre el archivo `salida.txt`. La flag `O_CREATE` lo crea si no existe, $O_W RONLY eswrite\ only$, S_

La función open devuelve (y asigna) al archivo abierto el menor file descriptor disponible, que al haber cerrado 1 es 1. Por ende, ahora 1 (es decir, STDOUT_FILENO) refiere al archivo salida.txt.

> **Nota.** Con "asigna", quise decir: open crea una entrada en la tabla de open files del sistema, cuyo índice es el file descriptor que devuelve.

(2) El segundo programa guarda en fd el file descriptor asignado al archivo de salida, cierra STDOUT_FILENO, y llama dup(fd). La función dup(fd) toma un file descriptor abierto fd, y crea un nuevo file descriptor que apunta al mismo archivo fd, pero es el menor número disponible. Por lo tanto, dup(fd) hace que 1 ==STDOUT_FILE apunte a salida.txt.

(Diferencias) En el segundo programa, fd y STDOUT_FILENO pueden usarse intercambiableme porque fd nunca se cierra. dup permite más control porque uno puede elegir qué file descriptors redirigir.