

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 permite 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 operativo 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 especificar 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 pisán?

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 Concurrency

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 estática. 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 nivel y en el alto nivel. Las operaciones de bajo nivel se llaman **mecanismos**. **Time sharing** es un mecanismo.

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**: `stdin`, `stdout`, `stderr`. Éstos permiten al programa leer input e imprimirlo o dirigirlo a alguna fuente.

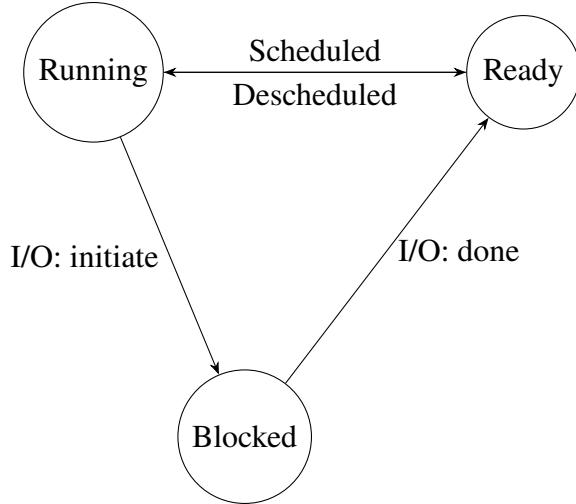
Sólo después de todo esto, el SO comienza la ejecución, iniciando el programa desde su entry point (i.e. `main()`). Una vez "salta" a la rutina `main()`, el SO transfiere control de la CPU al proceso recién creado, y el programa comienza.

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 resultantes 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, destacaremos 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 llamada 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 argumento `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`,

```
wait(wstatus) ≡ 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 siguen 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 logrando 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 directa, 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 introducir 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, alojar memoria, etc.

Para ejecutar una llamada a sistema, el programa ejecuta 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 código 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**, informando 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, alojar 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 comience. 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 llamada 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-configureado 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 permite 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 otro (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*, distintas 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**:

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

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

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

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

$$T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$$

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: ¡malarlo!

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 desea **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 codicioso que dura en un sistema con un quantum (y por ende un allotment) de 10ms. Imaginemos que hay tres queues de prioridad. 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 haciee 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 **starvation**, 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: Mecanismos

(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` dentro 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 as .

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:

$$1 \text{ (Proceso padre)} + 3 \text{ (Sus tres hijos)} + 3 \text{ (sus tres nietos)} + 1 \text{ (su bisnieto)} = 8.$$

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

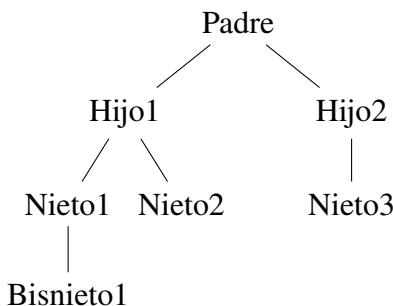


Figure 1: Diagrama de procesos de tres llamadas a `fork()`

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 *as* 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.

```
int main(int argc, char ** argv) {      int main(int argc, char ** argv) {  
    if (0<--argc) {                      if (argc<=1)  
        argv[argc] = NULL;                  return 0;  
        execvp(argv[0], argv);  
    }  
    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 $0 < \text{argc} - 1$ por $\text{argc} \geq 2$, porque

$$0 < \#args - 1 \iff 1 < \#args \iff 2 \leq \#args$$

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 -1`, se ejecuta el programa `ls` con argumentos `-1`. Etc.

(b) Este programa también requiere `argc >= 2` para hacer algo efectivo. Asumamos que ese es el caso. El programa forkea. 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 -1` 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_WRONLY` es *write only*, `S_IRWXU` no me queda claro.

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`.

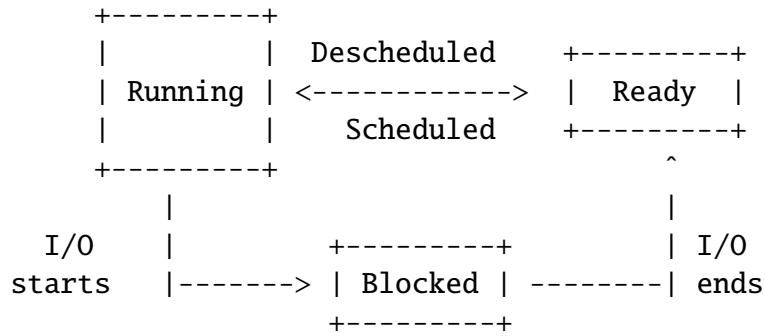
Nota. Con "asigna", quise decir: `open` crea una entrada en la tabla de open files del sistema, cuyo índice es el file descriptor que devuelve.

(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_FILENO` apunte a `salida.txt`.

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

(10) Para el diagrama de transición de estados de un proceso, describa cada uno de los cuatro escenarios posibles acerca de cómo funciona (o no) el SO si se quita solo una de las cuatro flechas.

El diagrama es el siguiente.



Imaginemos que quitamos la transición de running a ready. ...

(12) Verdadero o falso.

(a) Es posible que `user + sys < real`.

Recordemos que `user time` es el tiempo que la CPU ejecuta el programa en el `user space`, es decir fuera del kernel. `sys time` es lo opuesto: es el tiempo que el tiempo que la CPU ejecuta instrucciones del programa *kernel mode* (privilegiado), por ejemplo tras hacer una llamada a sistema.

`real time` es el tiempo de reloj desde el inicio de la ejecución hasta su terminación. Incluye todo: tiempo que el programa está pausado esperando que otros procesos se ejecuten, los tiempos que toma cada context switch, etc.

Es posible que `user + sys < real`. Por ejemplo, asuma un programa que debe esperar 10 segundos para que termine una I/O operation, pero que la ejecución de sus instrucciones (en kernel o user mode) dura solo 1 segundo. Entonces $sys \geq 11s$, pero $user + sys = 1$.

(b) Dos procesos no pueden usar la misma dirección de memoria virtual.

Falso. Siempre y cuando dicha dirección virtual sea traducida a distintas direcciones de memoria física, todo bien. Es más, si se hace una `read` operation con esa dirección, hasta puede pasar que la dirección virtual se traduzca a la misma dirección física (i.e. dos programas lean el mismo archivo en momentos distintos, y justo sucede que la dirección virtualizada coincide en ambos), y en ese caso también está todo bien.

(c) Para guardar el estado de un proceso, es necesario salvar el valor de todos los registros del microprocesador.

Verdadero. Hay que poder restaurarlos después a todos.

(d) Un proceso puede ejecutar cualquier instrucción de la ISA.

Falso. Muchas instrucciones no pueden ejecutarse directamente por un proceso. Debe solicitar un servicio al SO para que éste haga su magia en *kernel mode*.

(e) Puede haber traps por timer sin que esto implique cambiar de contexto.

Parece razonable que sí. Imaginemos el caso fantástico de un sistema en el que un único proceso se ejecuta. Las traps por timer seguirán ejecutándose, pero el contexto será el mismo. Lo mismo si un solo proceso está ready y running, y todos los demás están bloqueados.

(f) `fork()` devuelve cero para el hijo porque ningún proceso tiene PID 0.

Falso. Sí devuelve cero para el hijo, pero no tiene nada que ver con el hecho de que ningún proceso tiene PID cero. Simplemente 0 codifica el éxito en C, mientras -1 o valores distintos a 0 codifican error.

(g) Las syscall `fork()` y `execv()` están separadas para poder redireccionar los descriptores de archivo.

(?) Redireccionar los descriptores de archivo es sin duda parte importante de la ejecución con `fork` y `execv`. Pero no es la causa de su separación. `fork()` y `execv` existen separadamente porque `execv()` por sí solo no es suficiente para crear nuevos procesos (siempre destruye/reemplaza al proceso llamador). De ahí la utilidad de poder duplicar procesos con `fork()`. Ni hablar de que la separación también permite que el *calling process* supervise el estado del proceso generado y actúe en consecuencia (e.g. esperar que termine).

(h) Si un proceso padre llama a `exit()` el proceso hijo termina su ejecución inmediatamente.

Falso. El proceso padre y el proceso hijo se independizan casi completamente. Su único vínculo es que el padre puede tener la PID del hijo. No más que eso.

(i) Es posible pasar información de padre a hijo a través de `argv`, pero el hijo no puede comunicar información al padre ya que son espacios de memoria independientes.

Falso. El padre no pasa información al hijo a través de `argv`. El padre crea al hijo con `fork()` sin pasarle nada. Es el hijo quien luego puede auto-transformarse en un proceso nuevo con `execv`, donde sí puede pasar información con `argv`. Pero es el hijo pasando información a la forma que tomará (a su mutación, digamos), no el padre pasando información al hijo.

(j) Nunca se ejecuta código que está después de `execv()`.

Verdadero. Una vez se llama `execv()` en un proceso, toda su imagen es reemplazada, y esto incluye su código.

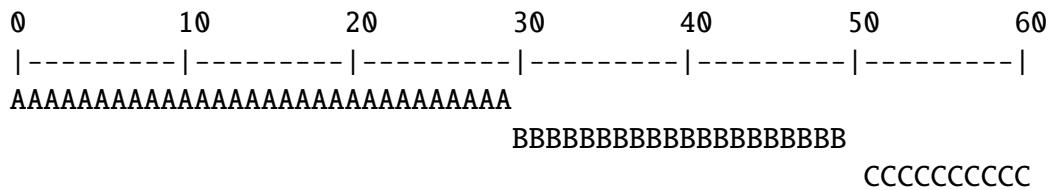
(k) Un proceso hijo que termina no se puede liberar de la tabla de procesos hasta que el padre no haya leído el exit status via `wait()`.

Verdadero. Cuando el hijo termina pasa al estado zombie conservando su exit status. Sólo cuando el padre lee ese estado, el sistema puede eliminar completamente al hijo de la tabla de procesos. Si fuera de otro modo, el padre no podría saber el exit status del hijo, porque apenas éste terminara desaparecería completamente.

8 Ejercicios: Políticas

(13) Dados tres procesos CPU-bound puros A, B, C con tiempo de llegada cero y tiempo de CPU 30, 20 y 10 respectivamente. Dibujar la línea de tiempo para las políticas de planificación FCFS y SJF. Calcular el promedio de turnaround y response times.

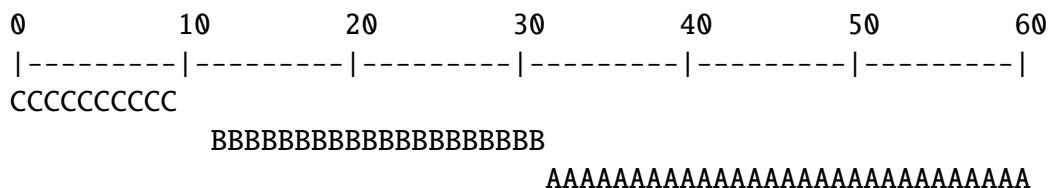
(a) En FCFS, podemos asumir que el orden de ejecución es A, B, C, pues llegan al mismo tiempo y un orden arbitrario debe tomarse (alfabético, en este caso). La línea de tiempo es:



El turnaround time es el tiempo total desde que el proceso llega hasta que termina su ejecución. En este caso, los response times son 30 (A), 50 (B) y 60 (C), con un promedio de 46.67.

El response time es el tiempo desde que el proceso llega hasta que empieza a ejecutarse. En este caso, 0 (A), 30 (B) y 50 (C). El promedio es 26.67.

(b) En SJF, se ejecutarán los procesos en el orden C, B, A. La línea de tiempo es:

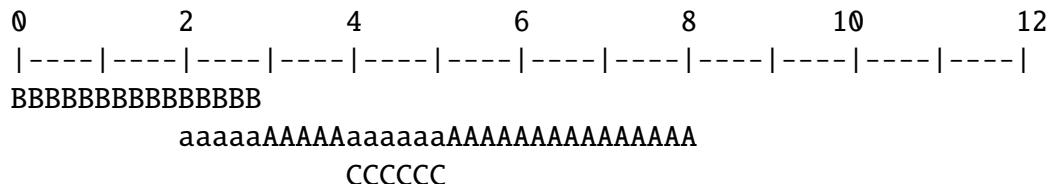


El turnaround promedio será $10 + 30 + 60 = 33.33$, y el response time promedio será $0 + 10 + 30 = 13.33$.

(14) Completar la siguiente tabla para las políticas STCF y RR (quantum = 2). Asuma que los procesos son CPU-bound puros.

Proceso	Tarrival	TCPU	Tfirstrun	Tcompletion	Tturnaround	Tresponse
A	2	4				
B	0	3				
C	4	1				

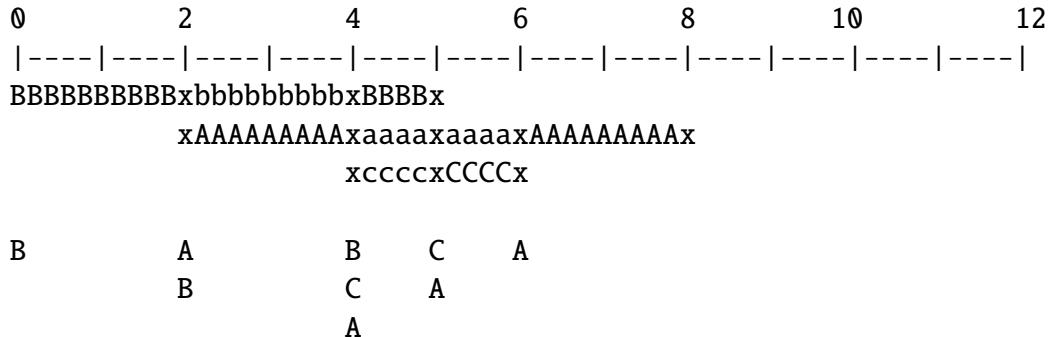
(a) Veamos el caso STCF primero. Claramente, cuando llega *B* (en el tiempo cero), se ejecuta inmediatamente. En el tiempo 2 llega *A*, con TCPU de 4. Como el TCPU restante de *B* es 1, *B* se sigue ejecutando hasta el tiempo 3, y entonces *A* se ejecuta. En el tiempo 4 llega *C*, con TCPU de 1. Como en ese momento *A* tiene un TCPU restante de 3, se ejecuta *C* hasta el tiempo 5, y termina. Luego *A* ejecuta sus tres segundos restantes, terminando en el tiempo 8. La línea de tiempo es la siguiente, con mayúsculas indicando tiempo de ejecución, y minúsculas tiempo en el sistema pero en espera.



Entonces:

Proceso	Tarrival	TCPU	Tfirstrun	Tcompletion	Tturnaround	Tresponse
A	2	4	3	8	6	1
B	0	3	0	3	3	0
C	4	1	4	5	1	0

(b) Ahora veamos RR con quantum 2. Para no escribir el razonamiento textualmente, hagamos la línea de tiempo directamente. Marcamos con x los instantes en los que ocurre un quantum, y debajo de estas x el estado de la cola. Recordemos que en cada quantum, si llega un proceso se lo encola inmediatamente, y solo después se encola el proceso que estaba en ejecución.



Resulta entonces:

Proceso	Tarrival	TCPU	Tfirstrun	Tcompletion	Tturnaround	Tresponse
A	2	4	2	8	6	0
B	0	3	0	5	5	0
C	4	1	5	6	2	1

9 Abstracción del espacio de direcciones

El sistema deseado es que los procesos permanezcan en memoria incluso cuando switcheamos entre ellos. De este modo, time-sharing puede implementarse eficientemente.

Example. Imaginemos un sistema con 512Kb de memoria en bloques de 64Kb. Cada programa tendría una pequeña parte de los 512Kb de memoria física reservada únicamente para ellos.

Dirección	Contenido
0KB	Sistema operativo, código, data estática
64KB	Espacio libre ...
128KB	Proceso C (código, data, etc.)
192KB	Proceso B (código, data, etc.)
256KB	Espacio libre ...
320KB	Proceso A (código, data, etc.)
448KB	Espacio libre ...
512KB	Espacio libre ...

El SO debe crear una abstracción fácil de usar de la memoria física. Esta abstracción se llama **address space**, o espacio de memoria. Es la visión que tiene el programa de la memoria del sistema.

El **address space** de un programa contiene toda la memoria relativa al estado del programa: su código (las instrucciones), el **stack**, el **heap**, entre otras cosas.

Example. Ejemplo de un address space de 16Kb.

Dirección	Contenido
0KB	Program Code (the code segment: where instructions live)
1KB	Heap (the heap segment: contains malloc'd data, dynamic data structures; it grows positively)
2KB - 15KB	Free Memory (space between heap and stack)
15KB	Stack (it grows negatively; contains local variables, arguments to routines, return values, etc.)
16KB	End of address space

Cuando describimos el **address space**, describimos la abstracción que el SO le provee al programa en ejecución. La memoria física es totalmente distinta. Las direcciones de memoria del address space se corresponden con direcciones diferentes de la memoria física. El problema entonces es: ¿cómo puede el SO hacer una abstracción privada del espacio de memoria para cada uno de múltiples procesos, los cuales comparten una única memoria física? Cuando el SO hace esto, decimos que **virtualiza** la memoria.

Los objetivos de esta abstracción son:

- **Transparencia.** Los programas no deberían saber que la memoria está virtualizada: deberían comportarse como si cada uno tuviera su propia memoria física.
- **Eficiencia.** La virtualización debería ser eficiente en términos de tiempo y espacio.
- **Protección.** El SO debería impedir que los procesos alteren la memoria usada por otros, es decir cada proceso debería vivir en **aislamiento**.

10 API de memoria

10.1 Tipos de memoria

Un programa en C tiene dos tipos de memoria allocadas. El **stack**, también llamado memoria automática, porque el compilador lo maneja implícitamente. Cuando declaramos una variable como `int x;`, el compilador se encarga de hacer espacio en el **stack** para la misma. El stack es una estructura anidada.

El otro tipo es el **heap**, que el programador maneja de manera explícita. Por ejemplo, al hacer `int * x = (int *) malloc(sizeof(int))`, estamos allocando en el stack. Más interesante aún: `x` se guarda en el stack como un puntero, pero el espacio al que apunta está en el heap.

10.2 La llamada `malloc()`

La llamada a `malloc` es simple: recibe un tamaño que especifica cuánta memoria asignar, y devuelve un puntero al espacio asignado (o falla y devuelve `NULL`).

Usar `void *malloc(size_t size)` de manera "cruda" se considera mala práctica. Generalmente, se castea el resultado (de tipo `void *`) a un puntero de tipo específico. Por ejemplo, `double *d = (double *) malloc(sizeof(double))`. El casteo no hace nada más que decirle al compilador y a otros programadores qué clase de valor está guardado en el espacio asignado.

11 Address translation

La técnica general que estudiaremos se llama **address translation**, o traducción de direcciones. En esta técnica, el hardware transforma los accesos a memoria (fetch, load, store) cambiando la dirección virtual que trae la instrucción a la dirección física donde la información deseada reside. En cada referencia de memoria, sucede una traducción.

Pero el hardware solo no puede virtualizar. El SO debe encargarse de que la traducción se haga correctamente, llevando registro de qué direcciones están libres y cuáles en uso.

11.1 Supuestos

Para un modelo simple, supondremos que el espacio de memoria del usuario se ubica *contiguamente* en memoria, y que no es muy grande. En particular, asumiremos que es más pequeño que el tamaño de la memoria física. También asumiremos que cada address space es del mismo tamaño.

11.2 Dynamic (hardware-based) relocation

Dynamic relocation es la primera forma de address translation. También es llamado **base and bounds**. Consiste en tener dos registros de CPU, el **base** y el **bounds**. Cada programa es escrito y compilado como si empezara en la dirección cero. Pero cuando un programa arranca, el SO decide dónde en la memoria física registrarlo, y guarda en **base** dicho valor. De este modo,

$$\text{dir física} = \text{dir virtual (0)} + \text{base}$$

Cada dir. de memoria referenciada por el programa estará en términos de la memoria virtual. Es decir, la dirección 128 será base + 128 en la memoria física.

En el registro **bounds** se guarda el tamaño del address space, y sirve para chequear que el programa no esté haciendo referencia a una dirección fuera de su address space. (Notar el supuesto de contigüidad.) Puede que el bound contenga el tamaño del address space, o la dirección física del final del address space. Depende del diseño, pero ambos sirven. Siempre asumiremos el primero.

La parte de la CPU que se encarga de hacer address translation se llama **Memory Management Unit** (MMU).

11.3 Hardware support

Entonces, vemos que el SO necesita apoyo del hardware para lograr la virtualización de la memoria. Necesita:

- Un **kernel mode** donde poder acceder a todos los lugares de memoria.
- Registros **base** y **bound**.
- Instrucciones especiales para modificar los registros antedichos. Solo en kernel mode!
- **Excepciones** que levantar cuando hay referencias ilegales (out of bounds) y un **exception handler** con instrucciones que correr cuando hay excepciones.

11.4 Temillas del SO

El SO, por otro lado, se encarga de:

- Cuando inicia un proceso, encontrar espacio disponible para su address space. Generalmente lo hace buscando una estructura de datos llamada **free list**, encontrando espacio para el nuevo address space y marcándolo como **usado** luego.

- El SO debe liberar la memoria de un proceso que ha terminado, poniendo esa memoria otra vez en la **free list** y limpiando lo que haga falta.
- El SO debe ajustar los registros base y bound cuando ocurre un context switch, asegurándose de que la traducción de direcciones se haga correctamente. En particular, esto significa que el SO debe tener guardados los valores de estos registros para cada proceso en memoria, para poder restaurarlos cuando vuelvan a activarse en un context switch. Estos se guardan en *per-process structures* como el PCB (process control block).
- El SO debe proveer **exception handlers**.

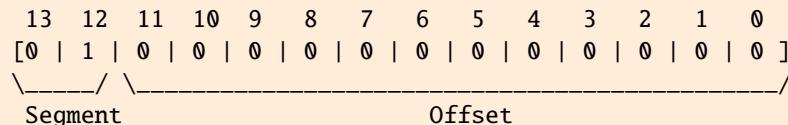
12 Segmentación

La idea de la segmentación es dividir el address space en segmentos y tener un par de registros **base** y **bound** por cada uno. Un segmento es una porción contigua del address space con una longitud particular. Asumiremos que un address space canónico tiene tres segmentos: código, heap y stack. El punto clave es que cada uno de estos segmentos puede ubicarse en diferentes partes de la memoria física. De este modo, sólo memoria utilizada es alocada.

La MMU de la CPU tiene tres pares de registros **base** y **bound**, uno por cada segmento. La pregunta es cómo saber a qué segmento pertenece una dirección virtual dada y cuál es el offset de dicho segmento. Hay dos formas de hacer esto.

La forma explícita es partir el address space en segmentos basados en los bits más altos de la virtual address. Por ejemplo, si el address space es de 16Kb, y cada segmento es de 4Kb, los dos bits más altos de la dirección virtual indican el segmento (00: código, 01: heap, 10: espacio libre, 11: stack), y los 12 bits restantes indican el offset dentro del segmento.

(†) Si nuestro address space es de 16Kb, necesitamos 14 bits para representar todas las direcciones. Si cada segmento es de 4Kb, necesitamos 12 bits para representar todas las direcciones dentro de un segmento. Por lo tanto, nos quedan 2 bits para identificar el segmento.



El código de la CPU podría verse así:

```
// Get top 2 bits of 14-bit VA.  

Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
```

```

// Get offset within segment.
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment]))
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    Register = AccessMemory(PhysAddr)

```

Un problema de este sistema es que cada segmento se limita a un tamaño máximo (4Kb en nuestro ejemplo). Si un programa necesita más espacio para el **stack** o el **heap**, no puede.

El approach implícito es más flexible. En este caso, el hardware determina el segmento viendo cómo está formada la dirección virtual. Por ejemplo, si el address fue generado con el program counter (fue un instrucción fetch), es del segmento de código. Si fue generado con el stack pointer, es del segmento de stack. Si fue generado con un puntero a heap, es del segmento de heap.

12.1 El stack

Para el stack, el hardware no solo necesita base and bounds sino también saber hacia dónde crece. El stack generalmente crece hacia direcciones más bajas. Los registros de segmento, por ende, se ven algo así:

Segment	Base	Bound	Grows positive?
Code	32K	2K	1
Heap	34K	3K	1
Stack	28K	2K	0

La traducción de direcciones dependerá de hacia dónde crece el segmento. Por ejemplo, si el stack crece hacia direcciones más bajas, la dirección física se calcula así:

```

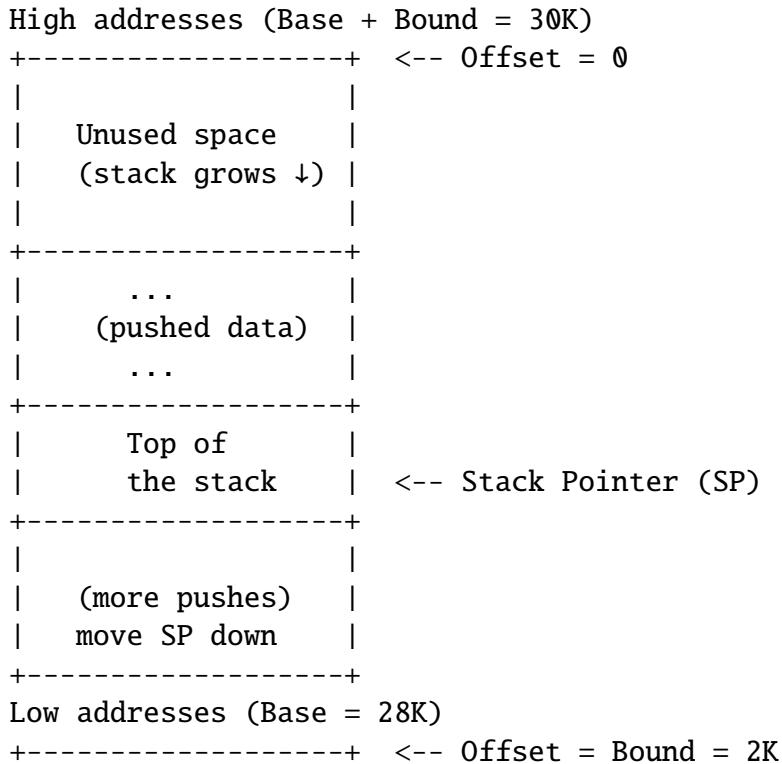
if (GrowsPositive[Segment])
    PhysAddr = Base[Segment] + Offset
else
    // Restamos al offset el tamaño del segmento
    NegOffset = Offset - Bound[Segment]
    PhysAddr = Base[Segment] + NegOffset

```

Notar que **NegOffset** será cero cuando el offset sea igual al bound, es decir igual al tamaño del segmento. En ese caso, la dirección física será igual a la base, es decir el stack pointer apunta a la

base del segmento de stack. Inversamente, si el offset es cero, `NegOffset` será igual a el tamaño del segmento, y la dirección física será igual a `Base + Bound`, es decir el final del segmento de stack.

Stack Segment (Base = 28K, Bound = 2K)



13 Administración de la memoria libre

Si la memoria se parte en segmentos de tamaño fijo, es fácil manejar el espacio libre: pasás el primer segmento libre a quien quiera que lo pida y adiós. Lo complicado es si el tamaño de los segmentos utilizados varía (e.g. en el address space con malloc, que extiende el heap, o en la memoria física con segmentación implícita).

14 Práctico 2

(1) Para cada una de las variables de este código indicar si están en el segmento de código, de pila o de montón (heap). Si hay punteros indicar a qué segmento apuntan.

Extra: ¿Dónde se ubica el arreglo global si lo declaramos inicializado a cero? `int a[N] = {0};`

```
#include <stdlib.h>
#define N 1024

int a[N];

int main(int argc, char **argv)
{
    int i;
    register int s = 0;
    int *b = calloc(N, sizeof(int));
    for (i=0; i<N; ++i)
        s += a[i] + b[i];

    free(b);
    return s;
}
```

Notemos que `N` no es una variable sino una directiva del procesador (macro). Recordemos además que las variables globales y estáticas se guardan en `.data` o `.bss`, y no en el stack, el heap ni el code. Por lo tanto, el array estático `a` no está en ninguno de los segmentos del modelo de memoria que utilizamos.

La variable `s` se define con la keyword `register`, y por lo tanto *si es posible* el compilador la guarda en un registro de CPU. No sabemos si esto se logra o no, y por lo tanto puede que esté en un registro de CPU o en el stack.

Todas las demás variables están en el stack, incluido el puntero `b`. Sin embargo, dicho puntero apunta al heap.

Extra. Si `a[N]` no se inicializa, se guarda en `.bss`; así como está inicializada, se guarda en `.data`. Una explicación de por qué existe esta diferencia en esta pregunta de StackOverflow.

(3) Dé V o F en los siguientes enunciados.

(a) `malloc` es una syscall.

La función `malloc` **no** es una syscall, así como tampoco lo es `free`. El diseño UNIX es usar las syscalls `brk`, `sbrk` o `mmap` para expandir el segmento de datos (i.e. la parte de la memoria que incluye el heap más datos estáticos como las regiones `.bss` y `.data`). La función `malloc` no expande ni reduce el tamaño del heap, solo administra la memoria dentro del segmento, y esto no requiere interacción (directa) con el kernel. Si `malloc` necesita más memoria de la disponible, entonces llama `brk`, `sbrk` o `mmap`, pero si la memoria que hay disponible alcanza no se solicita servicio privilegiado alguno.

(b) `malloc` siempre llama a una syscall.

Falso. `malloc` llama `brk`, `sbrk` o `mmap` sólo si la memoria disponible en el heap no alcanza para el tamaño de la allocation que se quiere hacer.

(c) `malloc` a veces llama a una syscall

True, por lo dicho anteriormente.

(e) Same para `free`.

`free` no es una syscall, y nunca llama a una syscall. Sin embargo, si se hace `free` sobre un puntero a un bloque de memoria grande y originalmente allocated con `mmap`, entonces se dispara una señal en el allocator (e.g. en `ptmalloc` de GLib) que verifica si debe llamarse `munmap` para remover la memoria del address space.

(f) La complejidad de `malloc(x)` es proporcional a `x`.

Falso. Si ya existe memoria disponible en el heap para guardar los `x` bytes solicitados, el tiempo de `malloc` es casi constante.

Si se pide un bloque tan grande que debe llamarse `mmap`, `malloc` puede tardar un poco más, pero más o menos independientemente de `x`.

En general, $O(\text{malloc}) = O(1) \neq O(x)$.

Ejercicio 4. Mostrar la secuencia de accesos a la memoria física que se produce al ejecutar este programa assembler x86_32, donde el registro base=4096 y bounds=256.

```
0: movl $128,%ebx
5: movl (%ebx),%eax
8: shll $1,%ebx
10: movl (%ebx),%eax
13: retq
```

La instrucción cero sólo guarda 128 en el registro de CPU %ebx. No hay acceso a memoria. La instrucción 5 guarda en %eax el contenido de la dirección de memoria %ebx, es decir lo que está en la dirección virtual 128. Para leer lo que está en dicha dirección, se traduce:

$$\text{PhysAddr} = 128 + \text{base} = 4224$$

La instrucción `movl` mueve una palabra (4 bytes), y por ende se accederá a las direcciones de memoria 4224, 4225, 4226, 4227. Todas están dentro de [base, base + bounds]. Todo OK.

La instrucción `shll $1, %ebx` significa hacer $\%ebx = \%ebx \ll 1$, con \ll el long shift. Es decir que simplemente guarda en %ebx el mismo contenido de %ebx, shifteado un lugar a izquierda, que asumiendo que no hay overflow equivale a multiplicar el contenido del registro por 2. Pero sabemos que %ebx contiene 128, es decir contiene

$$128 = 0x00000080 = (0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000\ 0000)_2$$

Por lo tanto, no habrá overflow, y el valor en %ebx ahora será

$$256 = 0x00000100 = (0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0000\ 0000)_2$$

La instrucción 10 entonces lo que hace es leer la dirección física

$$256 + \text{base}$$

Pero esto excede el bound. Hay error.

(6) Distinguir relocalización dinámica de relocalización estática.

La relocalización dinámica consiste en la utilización de hardware support, en particular de registros *base* y *bounds*, para implementar la virtualización de la memoria y efectuar address translations. Se le llama dinámica porque las traducciones ocurren en *run time*: cuando se ejecuta una instrucción de *load* o *store* con referencia a la memoria (e.g. `movl (%ebx), %eax`), la MMU realiza la traducción de la virtual address a la physical address, junto al chequeo de bounds.

La relocalización estática consiste en la utilización de un programa (*loader*) que transforma las direcciones virtuales de memoria de un ejecutable en sus direcciones físicas, agregándoles un offset. No se hace en runtime, sino sobre todo el ejecutable a la vez, y no utiliza hardware support. Al no utilizar hardware support, no implementa ningún tipo de protección: si una dirección virtual está out of bounds, no se alzará excepción alguna, y es posible que el programa termine leyendo/escribiendo direcciones ajena a su propio espacio.

(8) Una computadora proporciona a cada proceso 65536 bytes de espacio de direcciones dividido en páginas de 4 KiB. Un programa específico tiene el segmento código de 32768 bytes de longitud, el segmento montículo de 16386 bytes de longitud, y un segmento pila de 15870 bytes. ¿Cabría el programa en el espacio de direcciones? ¿Y si el tamaño de página fuera de 512 bytes? Recuerde que una página no puede contener segmentos de distintos tipos así se pueden proteger cada uno de manera adecuada.

(a : Page size = 4Kb) El segmento código ocupa 8 páginas exactamente. El montículo ocupa casi exactamente 4 páginas, pero marginalmente más que eso. La pila ocupa 3.87 páginas. Es decir que deben ponerse el código en 8 páginas, la pila en 4 páginas (con un poco de espacio extra), lo cual nos deja 4 páginas libres para poner el heap. Pero ya dijimos que el heap ocupa un poco más de 4 páginas. Por poco, pero el programa *no* entra en el address space.

(b : Page size = 512b). Hay $65536/512 = 128$ páginas en el address space. Los segmentos ocupan:

$$\begin{aligned} \text{Code} &\mapsto \frac{32768}{512} = 64 \text{ páginas} \\ \text{Heap} &\mapsto \frac{16386}{512} = 32.00390625 \text{ páginas} \\ \text{Stack} &\mapsto \frac{15870}{512} = 30.99609375 \text{ páginas} \end{aligned}$$

Es decir, necesitamos 33 páginas para el heap, 31 para el stack, en total = 64, más 64 páginas para el código nos da 128 páginas necesarias en total. Y es justo lo que tenemos.

(9) Suponga un sistema de memoria contiguo con la siguiente secuencia de tamaños de huecos: 10 KiB, 4 KiB, 20 KiB, 18 KiB, 7 KiB, 9 KiB, 12 KiB, 15 KiB.

Para la siguiente **secuencia de solicitudes** de segmentos de memoria: 12 KiB, 10 KiB, 9 KiB.

¿Cuáles huecos se toman para las distintas políticas?

- (a) Primer ajuste (*first fit*).
- (b) Mejor ajuste (*best fit*).
- (c) Peor ajuste (*worst fit*).
- (d) Siguiente ajuste (*next fit*).

(a) Se pide 12Kib y el primer ajuste es el de 20 Kib. La free list ahora es

$$10 \rightarrow 4 \rightarrow 18 \rightarrow 7 \rightarrow 9 \rightarrow 12 \rightarrow 15$$

Se piden 10 y el primer ajuste es la cabeza de la lista.. Después se piden 9 y el primer ajuste es el nodo de 18.

(b) Se piden 12Kib. Los candidatos (nodos posibles) son 20, 18, 12, 15. El menor es el de 12. Se lo entrega. Se piden 10. Misma lógica: se entrega el de 10. Se piden 9, se entrega el de 9.

(c) Se da el de 20. Luego se da el de 18. Luego se da el de 15.

(d) Se da el de 20. Luego se da el de 18. Luego se da el de 9.

(10) La TLB de una computadora con una pagetable de un nivel tiene una eficiencia del 95%. Obtener un valor de la TLB toma 10ns. La memoria principal tarda 120ns. ¿Cuál es el tiempo promedio para completar una operación de memoria teniendo en cuenta que se usa tabla de páginas lineal?

Tomamos el promedio ponderado:

$$\frac{1}{2}(0.95 \times 10\text{ns} + 0.05 \times 120\text{ns}) = 7.75\text{ns}$$

(11) Considere el siguiente programa que ejecuta en un microprocesador con soporte de paginación, páginas de 4KiB y una TLB de 64 entradas.

```
1 int x[N];
2 int step = M;
3 for (int i=0; i<N; i+=step)
4     x[i] = x[i]+1;
```

(a) ¿Qué valores de N, M hacen que la TLB falle en cada iteración del ciclo?

(b) ¿Cambia en algo si el ciclo se repite muchas veces? Explique

(a) Hay un caso trivial: si $M \geq N$, ocurre una sola iteración y en dicha iteración hay un TLB miss. Asumamos entonces que $M < N$.

Incluso en este supuesto, hay un caso trivial; a saber, si la array x no ocupa más de una página. Es obvio que entonces habrá una sola TLB miss, y todas las otras referencias a memoria son un hit. Obviemos también este caso.

Para garantizar que x ocupe al menos dos páginas, x debe ocupar más de 4KiB de memoria. Asumamos que cada entero ocupa 4 bytes = 32 bits, porque eso se hace en el libro.

Hay 4KiB por página y por ende $4\text{KiB}/4\text{byte} = (4096/4)\text{bytes} = 1024 \text{ bytes} = 1\text{KiB}$. Por lo tanto, una página se llena con exactamente 1024 enteros. Se necesita entonces $N > 1024$ para que x ocupe más de una página. Definimos entonces $N > 1024$, $M < N$ como el caso interesante.

Para que cada iteración produzca un TLB miss, $a[i]$ y $a[i+M]$ deben estar en páginas distintas, para todo i . Esto se garantiza si la distancia entre sus direcciones de memoria es de 4KiB (una página entera). Pero ya dijimos que una página consiste en 1024 enteros. Es decir, $M = 1024$ garantiza que $a[i]$ y $a[i+1]$ estén a una página de distancia en memoria. Notemos que por supuesto $N > 1024$ y por lo tanto estamos respetando $M < N$.

Conclusión. Si $N > 1024$ y $M = 1024$ se garantiza que la TLB falla en cada iteración.

Nota. Si generalizamos y decimos que un entero ocupa I bits, que el address space es de A bits, y las páginas son de P bits, entonces P / I es el número de enteros por página. Se requiere entonces

1. $N > P/I$
2. $M \equiv 0 \pmod{P/I}$

es el caso general que garantiza solo miss, donde debe cumplirse que P es divisible por I . Esto es casi dado por supuesto, dado que tanto I como P son alguna potencia de dos, y no es concebible en términos prácticos que $I \geq P$.

(b) Si el ciclo se repite muchas veces, seguirán siendo todos misses la primera vez que se corra el ciclo. Las siguientes veces, depende del tamaño de la TLB y de la array. Si la array tiene 64 elementos o menos, todo seguirá en caché y cada referencia será un hit. Si no, habrán 64 hits (lo que quedó en caché) y todo lo demás será un miss.

(12) Dado un tamaño de página de $4\text{KiB} = 2^{12}$ bytes y la page table dada abajo:

(a) ¿Cuántos bits de direccionamiento hay para cada espacio?

(b) Determine las direcciones físicas a partir de las virtuales: 39424, 12416, 26112, 63008, 21760, 32512, 43008, 36096, 7424, 4032.

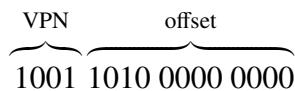
(c) Determine el mapeo inverso, i.e. las direcciones virtuales a partir de las físicas, para 16385, 4321.

V	F	¿Válida?
0	000	1
1	111	1
2	000	0
3	101	1
4	100	1
5	001	1
6	000	0
7	000	0
8	011	1
9	110	1
10	100	1
11	000	0
12	000	0
13	000	0
14	000	0
15	010	1

(a) Se trata de una page table con 16 páginas, lo cual significa que se utilizan cuatro bits para el virtual page number (VPN). Los page frame numbers (número de página física) tienen tres bits, indicando $2^3 = 8$ páginas físicas en total. El tamaño de página es de 4KiB , dando 12 bits en total por cada virtual address.

(b)

(† : 39424) La forma larga es notar que, en binario, 39424 es



lo cual da $\text{VPN} = 9$ y offset de $2^{11} + 2^9 = 2560$. Una forma equivalente es hacer la división entera del número decimal (dividendo) por el tamaño de página (divisor), de lo cual resulta

$$39424 = 9 \times 4096 + 2560$$

Por lo tanto, 9 es el número de página y 2560 el offset. En cualquier caso, sabiendo el VPN, en la tabla vemos que el VPN de 9 se corresponde con el PFN 110 =6. Por lo tanto, la dirección final es 110 1010 0000 0000, o bien

$$2^{14} + 2^{13} + 2^{11} + 2^9 = 27136$$

También podríamos notar que, siendo el offset el mismo, podemos multiplicar el tamaño de página por el PFN, y luego sumar el offset:

$$6 \times 4096 + 2560 = 27136$$

(† : 43008) Como $43008 = 4096 \times 10 + 2048$, el VPN es 10 y el offset 2048. El VPN 10 se traduce al PFN 100 = 4. Entonces la dirección física es

$$4 \times 4096 + 2048 = 18432$$

Alternativamente, $2048 = 2^{11}$ y por lo tanto la dirección virtual es

$$100\ 1000\ 0000\ 0000$$

lo cual da $2^{11} + 2^{14} = 18432$.

El procedimiento es el mismo para las demás direcciones.

(c) Ahora damos mapeos inversos.

(† : 16385) Notar que $16385 = 4096 \times 4 + 1$, lo cual significa que se trata de la PFN 4 con offset 1. La PFN 4 = 100 se corresponde con los VPNs 10 y 4. Por lo tanto,

$$10 \times 4096 + 1 = 40961, \quad 4 \times 4096 + 1 = 16385$$

son las direcciones virtuales que mapean a la dirección física 16385.

(14) Dado el sistema de paginado de dos niveles del i386 direcciones virtuales de 32 bits, direcciones físicas de 32 bits, 10 bits de índice de *page directory*, 10 bits de índice de *table directory*, y 12 bits de *offset* dentro de la página, o sea un (10, 10, 12), indicar:

1. Tamaño de total ocupado por el directorio y las tablas de página para mapear 32 MiB al principio de la memoria virtual.
2. Tamaño total del directorio y tablas de páginas si están mapeados los 4 GiB de memoria.
3. Dado el ejercicio anterior ¿Ocuparía menos o más memoria si fuese una tabla de un solo nivel? Explicar.
4. Mostrar el directorio y las tablas de página para el siguiente mapeo de virtual a física:

Virtual	Física
[0MiB, 4MiB)	[0MiB, 4MiB)
[8MiB, 8MiB + 32KiB)	[128MiB, 128MiB + 32KiB)

(1) Notar que:

- Tamaño de page directory: $2^{10} \times \text{sizeof(PDE)} = 2^{10} \times 4$
- Tamaño de page table: $2^{10} \times \text{sizeof(PTE)} = 2^{10} \times 4$
- Tamaño de página: $2^{12} = 4\text{KiB}$.

donde `sizeof(PDE) = sizeof(PTE) = 4bytes` porque las direcciones físicas y virtuales son de 32 bits. La memoria virtual está partida en páginas de tamaño 4KB, y 32MiB es lo mismo que

$$2^{25} = 2^{15} \times 2^{10} = 2^{15} \times 1\text{KiB} = 32768\text{KiB}$$

Por lo tanto, 32MiB ocupan $32768/4 = 8192$ páginas.

En cada page table hay $2^{10} = 1024$ entradas. Por lo tanto, para 8192 páginas, necesitamos $8192/1024 = 8$ page tables. Hay un único page directory por proceso.

Conclusión. Los recursos necesarios para mapear 32MiB son 8 page tables + 1 page directory. Memoria necesaria:

$$\begin{aligned}
8 \times \text{size_of(PT)} + 1 \times \text{size_of(PD)} &= 8 \times (2^{10} \times 4\text{bytes}) + 1 \times (2^{10} \times 4\text{bytes}) \times \\
&= 8 \times 2\text{KiB} + 4\text{KiB} \\
&= 32\text{KiB} + 4\text{KiB} \\
&= 36\text{KiB}
\end{aligned}$$

(b) 4GiB de memoria son 2^{32} bytes. Cada página es de 2^{12} bytes. Por lo tanto, se necesitan $2^{32}/2^{12} = 2^{20}$ páginas.

Cada page table referencia 2^{10} páginas. Por lo tanto, se necesitan $2^{20}/2^{10} = 2^{10}$ page tables.

Hay un único PD por proceso.

∴ Se necesitan

$$\begin{aligned}
2^{10} \times \text{sizeof(PT)} + \text{sizeof(PD)} &= 2^{10} \times 2^{10} \times 4 + 2^{10} \times 4 \\
&= 2^{22} + 2^{12} \\
&= 4\text{MiB} + 4\text{KiB}
\end{aligned}$$

de memoria.

(c) Ocuparía un poco menos: la page table gigante pesaría lo mismo que todas las page tables pequeñas del esquema multi-nivel, pero nos ahorraremos la memoria necesaria para la PD.

Para probarlo, veamos cómo queda un esquema de paginación de un solo nivel (lineal). Se sigue teniendo un offset de 12 bits, y ahora los 20 bits restantes son el VPN. Entonces:

- Tamaño de página: 2^{12}
- Tamaño de la page table: $2^{20} \times \text{sizeof(PDE)} = 2^{20} \times 4\text{bytes}$.

Por ende, usando todas las páginas, se necesitan $2^{20} \times 4\text{bytes} = 4\text{MiB}$.

(d) Sea Block A el bloque de 4MiB al principio del address space, Block B el bloque de 32KiB. Es fácil ver que:

- Block A necesita 2^{10} páginas, una sola page table.
- Block B necesita $2^3 = 8$ páginas, una sola page table.

¿Cómo se ve el diagrama? A la izquierda de todo, la PD con 1024 entradas.

La entrada PDE 0 apunta a PT A, la page table del proceso A. La entrada PDE 1 apunta a PT B, la page table del proceso B. Todas las demás entradas son inválidas.

PT A tiene 1024 entradas, y todas son válidas, y todas apunta a alguna dirección de la memoria física entre PFN1 y PFN 1024.

PT B tiene 1024 entradas, sólo 8 de ellas son válidas, y esas 8 apuntan a direcciones de memoria entre PFN X y PFN Y, donde PFN X se corresponde con la dirección de memoria 8MiB, y PFN Y con la dirección 8MiB + 32KiB.

Explique porque un i386 no puede mapear los 4 GiB completos de memoria virtual. ¿Cuál es el máximo?

Ya vimos que para mapear 4GiB se necesitan 4MiB + 4KiB de espacio para las PTs y el PD. Por ende, sería imposible mapear los 4GiB, pues no quedaría ese espacio adicional para dichas estructuras.

Obviamente, si alocamos la máxima cantidad de espacio posible, usamos todas las páginas. Y seguimos usando un sola PD. Por ende, la cantidad de espacio necesaria para las estructuras, en el caso en que alocamos la máxima memoria posible, es 4MiB + 4KiB. Por ende, el máximo es 4GiB - (4MiB + 4KiB).

(★ : Ej. de parcial) Tenemos un esquema de paginación multinivel de dos niveles, que mapea (10, 10, 12) a (20, 10), con tablas de página de 4KiB. El registro de paginación CR3 apunta al marco físico **0xFA110**. Se tiene que en las direcciones **0xFA110**, **0xB0CA**, **0xCA5A**, existe el siguiente contenido (indexado):

0xCA5A	0xB0CA	0xFA110
0x3FF: 0xFA110, SWP	0x3FF: 0xFA110, SWP	0x3FF: 0xFA110, SWP
0x3FE: 0x00000, SR-	0x3FE: 0x00000, SR-	0x3FE: 0x00000, SR-
:	:	:
0x004: 0x00000, SR-	0x004: 0x00000, SR-	0x004: 0x00000, SR-
0x003: 0x00000, SR-	0x003: 0x00000, SR-	0x003: 0x00000, SR-
0x002: 0x00CAB, SR-	0x002: 0xB0CA, SR-	0x002: 0xB0CA, SR-
0x001: 0x7A11A, UWP	0x001: 0x7A11A, UWP	0x001: 0x7A11A, UWP
0x000: 0x00CAB, UWP	0x000: 0xCA5A, UWP	0x000: 0xB0CA, UWP

Traducir las direcciones virtuales:

1. **0x00000E5A**
2. **0x00800F00**
3. **0xFFFFFFFF**

Solución. Notemos que cada marco de página va desde la dirección (interna) **0x000** hasta la dirección **0x3FF**, que es el número 1023. Esto es correcto porque cada page table tiene $2^{10} = 1024$ entradas. (Sanity check.)

Notar que la tabla que nos dieron de la dirección **0xFA110** es el PD.

(1) Tomemos la dirección virtual **0x00000E5A**. Su offset son los últimos 12 bits, i.e. **E5A**. Claramente, su PD index es cero, y su PT index es 0. Es decir, esta dirección se traduce yendo: (a) a la primer entrada del PD, que nos lleva a una page table; (b) a la primera entrada de la page table, que nos lleva a una dirección de memoria física.

La primer entrada del PD es **0xB0CA**. La page table en esta dirección es dada en la tabla. Su primera entrada es **0xCA5A**. La página con dicha dirección nos es dada, y agarramos el elemento de dicha página con el offset.

Dirección final: **0xCA5AE5A**.

(2) Notar que **0x00800F00** se traduce en bits como

0000 0000 1000 0000 0000 1111 0000 0000

que tiene como PD index **0000 0000 10** = 2, como PT index **0x0** y como offset **0xF00**. Es decir, hay que buscar la tercera entrada del PD (index 2), lo cual nos lleva a una page table; luego la cero-éxima entrada de esa page table, lo cual nos da la dirección. Pero la tercera entrada del PD es **0x0B0CA**. La primera entrada de la PT correspondiente es **0x0CA5A**, igual que antes.

Dirección final: **0x0CA5AF00**.

(3) Obviamente, esto es "la última entrada del PD", luego "la última entrada de la PT correspondiente", y luego un offset de FFF. Pero la última entrada del PD es **0xFA110** (sí mismo, i.e. es un PD recursivo). Luego la última entrada de la "PT" correspondiente (que es el PD) es otra vez **0xFA110**. Agregándole el offset, obtenemos:

Dirección final: **0xFA110FFF**.

(★) Ahora, para el mismo esquema, hacer el mapeo inverso de la dirección **0xFA110505** a todas sus direcciones virtuales.

Ignoremos el offset **0x505** que no se traduce. Podemos pensar en los "caminos" de una columna a otra de la tabla que nos llevan a la dirección física **0xFA110**.

(1) 0x3FF de PD index nos deja en la columna derecha. 0x3FF de PT index, 0x505 de offset. Dir: **0xFFFFF505**.

(2) 0 de PD index nos lleva a la columna del medio. 0x3FF de PT index nos lleva a **0xFA110**. O sea,

0000 0000 00 | 11 1111 1111 | offset = **0x003FF505**

(3) 2 de PD index también nos lleva a la columna del medio, así que

0000 0000 10 | 11 1111 1111 | offset = **0x00BFF505**.

(4) 2 o 0 de PD index nos lleva a la columna del medio, luego un PT index de 0 nos lleva a la columna izquierda.

15 Ejercicios de parciales

(★) El siguiente código de máquina y su desensamblado RISC-V **computa la suma prefijo en el mismo arreglo** (*in-place prefix sum*). El arreglo a está en el segmento ELF .bss y empieza en **0x2FC0** y termina en **0x3080** exclusive. Como sus elementos son `unsigned long`, cada uno ocupa 8 bytes y por lo tanto tiene 9 elementos.

```
0000000000000634 <main>:  
634: 0613      li    a2 ,0x3080      # _BSS-END_ -> &a [ 9 ]  
636: b206      li    a5 ,0x2FC8      # <a+0x8> &a [ 1 ]  
638: 6398      ld    a4 ,0( a5 )      # a4 = a[ i ]  
63a: ff87b683  ld    a3 ,-8( a5 )    # a3 = a[ i-1 ]  
63e: 9736      add   a4 ,a4 ,a3  
640: e398      sd    a4 ,0( a5 )      # a[ i ] = a4  
642: 07a1      addi  a5 ,a5 ,8       # i++  
644: fec79ae3  bne   a5 ,a2 ,0x638     # <main+0x10>, "i<9"  
648: 8082      ret
```

Escribir la **traza de memoria** completa que genera la ejecución del proceso **incluyendo los instruction fetch**.

El arreglo de unsigned longs (8 bytes c/u) empieza en **0x2FC0** y por ende en dicha dirección está el primer elemento, en **0x2FC8** el segundo, etc. Las únicas instrucciones que hacen acceso a memoria para algo distinto a un fetch de instrucción son: 638, 63a, 640, que acceden a la dirección (a5).

Inicialización

La inicialización son las instrucciones 634, 636.

La instrucción 634 hace un load `li` en el registro `a2` con el valor **0x3080**. Sólo se hace acceso a memoria para el fetch de `li`.

La instrucción 636 guarda en `a5` el valor **0x2FC8** (la dirección del segundo elemento de la array). También solo se hace acceso a memoria para el fetch de `li`.

Vuelta 1

Instructions fetch para `ld`, `add`, `sd`, `addi`, `bne`. En la primera vuelta, el registro `a5` tiene el valor **0x2FC8**, y por ende en la instrucción 638 se hace un load desde esa dirección en `a4`. 63a hace un load en `a3` de la dirección `a5` shifteada -8, i.e. de **0x2FC0**, el primer elemento de la array. Same para 640 pero hace un store.

Se dejan sin completar las vueltas siguientes porque si uno entiende cómo hacer la vuelta 1, entiende cómo hacer las demás.

(★) Supongamos que en `trampoline.S`, la rutina en ensamblador RISC-V que guarda los registros de espacio de usuario se comete un pequeño error por culpa del gato. La parte que los restituye está perfecta.

<code>sd a1 , 120(a0)</code>	<code>ld a1 , 120(a0)</code>
<code>sd a2 , 128(a0)</code>	<code>ld a2 , 128(a0)</code>
<code>sd a3 , 136(a0)</code>	<code>ld a3 , 136(a0)</code>
<code>sd a4 , 144(a0)</code>	<code>ld a4 , 144(a0)</code>
<code>sd a2 , 152(a0) # ERROR!</code>	<code>ld a5 , 152(a0)</code>
<code>sd a6 , 160(a0)</code>	<code>ld a6 , 160(a0)</code>
<code>sd a7 , 168(a0)</code>	<code>ld a7 , 168(a0)</code>

Indicar en el código de máquina del ejercicio anterior a este, **ENTRE** qué líneas se puede producir un **TRAP** sin cambiar el funcionamiento del programa (poner TRAP) y entre qué líneas ese **TRAP resulta fatal** para la ejecución de nuestro código (poner F).

Recalcamos: poner TRAP o F entre cada una de las líneas de código indicando si se puede producir un **TRAP** de manera inocua o no.

Primero entendamos el error inducido por el minino. La quinta línea debería haber sido `sd a5 , 152(a0)`, pero fue `sd a2 , 152(a0)`. Es decir, hay un error en el store: la dirección (152a0) debería tener el valor del registro a5, pero tiene el del registro a2.

El registro a5 es el que lleva i, la variable de iteración. El registro a2 es el que tiene la dirección donde termina el array. El loop es tal que si a5 no es igual a a2 (si todavía en la iteración no llegamos al final del array), se vuelve a iterar. (Ver instrucción bne).

La situación por ende es la siguiente: si hay un trap, la variable de iteración i tendrá cuando se regrese al programa el valor "límite" que hace que el ciclo termine.

Obviamente, si hay un trap justo entre la primera y la segunda línea, no pasa nada, porque en la segunda línea se carga en a5 el valor correcto. El programa, sin darse cuenta, "corregiría" el error del gato.

Si hay un trap entre la segunda y la tercera línea, ya se rompe el funcionamiento del programa, porque la instrucción `ld a5 , 0(a5)` se correspondería semánticamente con `ld a5 , 0(a2)`. Lo mismo para todas las demás líneas del bucle, con **una excepción**.

Entre la línea 642 y 644, si un trap se produce en cualquier iteración anterior a la última, la instrucción bne va a salir del loop antes de tiempo. Pero si justo se produce un trap en la última iteración, y no en ningnua anterior, el valor de a5 ya coincide con el de a2, y el error del gato no cambiaría nada. Por ende, entre estas dos líneas una trap podría ser inocua o fatal, dependiendo de cuántas iteraciones van.

Entre la instrucción 644 y 648, asumiendo que se salió del loop, un trap resulta inocuo.

En conclusión:

```
634: li      a2 ,0x3080
      TRAP
636: li      a5 ,0x2FC8
      F
638: ld      a4 ,0( a5 )
      F
63a: ld      a3 ,-8( a5 )
      F
63e: add    a4 ,a4 ,a3
      F
640: sd      a4 ,0( a5 )
      F
642: addi   a5 ,a5 ,8
      F o TRAP, dependiendo de la iteracion
644: bne    a5 ,a2 ,0x638
      TRAP
648: ret
```

(★) Planificar con Round Robin ($Q = 2$) para los siguientes procesos que tienen mezcla entre cómputo CPU y espera IO. Ante situaciones de simultaneidad, ordenar alfabéticamente, por ejemplo: ¿Cuál de los tres procesos inicia en tiempo 0?: el “A”. Hacerlo para las unidades de tiempo $t = 0, 1, \dots, 16$.

Proceso	Inicio	CPU	IO	CPU
A	0	1	4	3
B	0	1	1	1
C	0	8		

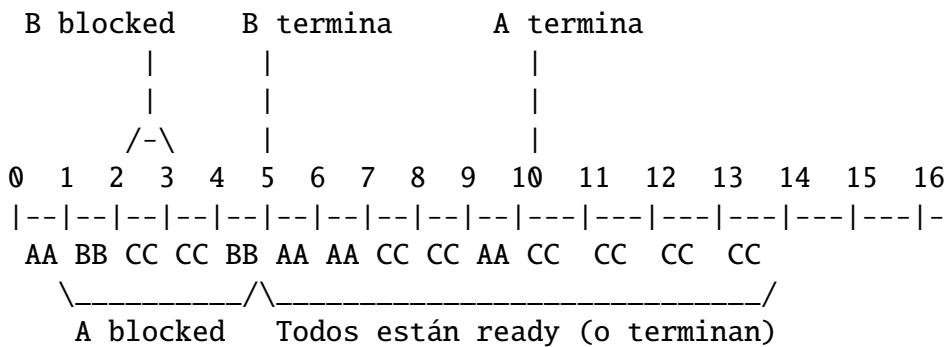
El proceso A usa la CPU el segundo 0, pasando a bloqueado luego por 4 segundos (hasta el tiempo 5). Cuando pasa a bloqueado, se asigna la CPU a B, que usa la CPU por un segundo y se bloquea por un segundo, dando a C usar la CPU por al menos un segundo.

Es decir, en los primeros tres segundos, usan un segundo cada uno en orden alfabético.

En el tiempo 3, C corrió el primer segundo su quantum y B se acaba de desbloquear. C correrá el segundo segundo de su quantum, y cederá el control a B, que ya está ready.

En $t = 5$, A se acaba de desbloquear, y B acaba de correr el primer segundo de su quantum, y con eso termina. Como A y B están ready, se le da la CPU a A, que corre dos segundos (todo su quantum). Ahora se le da la CPU a C, que corre dos segundos (todo su quantum). Ahora se le da a A, a la cual le queda un solo segundo. Lo corre, termina, y la CPU queda solo para C, que corre por 4 unidades de tiempo (2 quantums), porque es lo que le falta.

El esquema resultante es:



(★) Se tiene un esquema de paginación RISC-V con páginas de 4KiB de tres niveles con formato (9, 9, 9, 12) -> (44, 12). Supongamos que el registro de paginación apunta al marco físico **satp=0x00000000FE0**, y se tiene el esquema de memoria dado en la tabla de abajo.

(a) Traducir de virtual a física las direcciones **0x0000**, **0x1000**, **0x2000**, **0x3000**.

(b) Traducir de física a virtual **0xDECADA980**, incluyendo **todas** las direcciones virtuales.

0x000000000FE0	0x000000000FEA	0x000000AD0BE
0x1FF: 0x0000000000000000 , ----	0x1FF: 0x0000000000000000 , ----	0x1FF: 0x0000000000000000 , ----
:	:	:
0x004: 0x0000000000000000 , ----	0x004: 0x0000000000000000 , ----	0x004: 0x0000000000000000 , ----
0x003: 0x0000000000000000 , ----	0x003: 0x0000000000000000 , ----	0x003: 0x000001D1AB10 , XWR-
0x002: 0x0000000000FEA , XWRV	0x002: 0x000000AD0BE , XWRV	0x002: 0x00000DECADA , -WRV
0x001: 0x0000000000FEA , XWRV	0x001: 0x000000AD0BE , XWRV	0x001: 0x000CAFECAFE , ----
0x000: 0x0000000000FEA , XWRV	0x000: 0x000000AD0BE , XWRV	0x000: 0x0000000ABAD , X--V

(a) (Dir. **0x0000**). Esto significa.

- Usar los primeros nueve bits (todos cero) para obtener una dirección en el nivel 2, que nos da la ubicación del nivel 1.
- Usar los siguientes nueve bits (todos cero) para obtener una dirección en el nivel 1, obteniendo la ubicación del nivel 0.
- Usar los siguientes nueve bits (todos cero) para obtener una dirección en el nivel 0, que nos da la ubicación de la página física.
- Usar el offset (todos los bits son cero) para hallar la traducción en la página física.

En el nivel más alto (**0x000000000FE0**), como los primeros nueve bits son cero, vamos a la primera entrada, que tiene el valor **0x0000000FEA**. Esa es la dirección del nivel 1.

En el nivel 1 (**0x000000000FEA**), vamos a la primera entrada, que es **0x000000AD0BE**. Esa es la dirección del nivel cero.

En el nivel cero, vamos a la primera entrada, cuyo valor es **0x0000000ABAD**. Este es el PFN al que corresponde el VPN de la virtual address. Se nos dice que las direcciones físicas PFN tienen 44 bits = 11 bytes, así que la dirección final es, y los 12 bits (3 bytes) finales son el offset, que en este caso es cero. La traducción final, partida en PFN y offset, es

- **0000000ABAD / 000**

(b) (Dir **0x2000**). El offset (últimos tres bytes) siguen siendo cero. Escribamos la dirección virtual de 39 bits completa:

$$0x2000 = 0\ 0000\ 0000 / 0\ 0000\ 0000 / 0\ 0000\ 0010 / 0000\ 0000\ 0000$$

Se ve entonces que esto es: primer index 0, segundo index 0, tercer index 2, offset cero. Dirección física, con offset añadido:

0x00000DECADA000

(b) (Dir **0x3000**)

Igual que antes:

$$0x3000 = 0\ 0000\ 0000 / 0\ 0000\ 0000 / 0\ 0000\ 0011 / 0000\ 0000\ 0000$$

Los index de cada nivel son 0, 0 y 3 y el offset es cero. Dir física:

0x000001D1AB10

etc.

(b) Notemos que **0xDECADA980** se obtiene traduciendo el VPN (desconocido) al PFN **0xDECADA** más un offset de **0x980**. El PFN **0xDECADA** está contenido en la tercer entrada del tercer nivel. Al tercer nivel se llega de 3×3 maneras (es fácil ver por qué), y una vez en dicho nivel, a la tercer entrada se llega de una sola forma (con un tercer index de 2).

Observación. Notar que en bits, el offset de **0x980** es **1001 1000 0000**.

Por ende, las direcciones virtuales que mapean a dicha dirección física son:

- **000000000 / 000000000 / 000000010 / 1001 1000 0000**
- **000000000 / 000000001 / 000000010 / 1001 1000 0000**
- **000000000 / 000000010 / 000000010 / 1001 1000 0000**
- **000000001 / 000000000 / 000000010 / 1001 1000 0000**

- 000000001 / 000000001 / 000000010 / 1001 1000 0000
- 000000001 / 000000010 / 000000010 / 1001 1000 0000
- 000000010 / 000000000 / 000000010 / 1001 1000 0000
- 000000010 / 000000001 / 000000010 / 1001 1000 0000
- 000000010 / 000000010 / 000000010 / 1001 1000 0000

16 Concurrency

16.1 Threads

Un *thread* (hilo) es una abstracción que describe uno de posiblemente varios puntos de ejecución de un programa. Un *multi-threaded program* tiene varios puntos de ejecución. El estado de una thread es caracterizado por:

- Un PC propio.
- El valor de sus registros
- Su stack privado

Como distintas threads no comparten registros, cuando se pasa de ejecutar una thread a otra debe ocurrir un context switch, y un *thread control block* (TCB) debe guardar el valor de los registros de cada thread para poder efectuar el switch. Aunque distintas threads comparten el address space, tienen un stack propio cada una. Esto significa que el address space se segmenta en *code*, *heap*, y *stack 1*, ..., *stack n* para un programa con n threads.

Una vez que una thread se crea, puede iniciarse inmediatamente o pasar a un estado de *ready* (no *running*). Esto depende de lo que decida el scheduler.

Naturalmente, existe información compartida entre threads. Distintas threads pueden modificar simultáneamente una variable, por ejemplo. Esto lleva a la posibilidad de que se produzcan race conditions.

(†) Veamos un ejemplo. Imaginemos un código que inicia dos threads, t_1 y t_2 . Cada thread simplemente itera $n = 10,000,000$ veces, y en cada iteración suma 1 a una variable S que inicialmente es cero. Esperaríamos que al terminar la ejecución de las dos threads, el valor final de S fuera $2n$. Pero esto no es necesariamente es así. Y para entender por qué, hay que considerar el código máquina.

Si asumimos que la variable S está en la ubicación de memoria `0x8049a1c`, el código que le suma 1 se ve algo así:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Imaginemos que t_1 entra en esta región de código en un momento en que la variable S tiene el valor 50, y que hace el primer `mov`. Es decir, en `%eax` ahora está el valor 50. Luego efectúa el `add`, dejando en `%eax` el valor 51. Entonces un timer interrupt ocurre, y el OS guarda el estado de los registros en el TCB.

Imaginemos que entonces el scheduler le da el control a t_2 , y que t_2 logra efectuar las tres instrucciones sin ser interrumpido. Al iniciar, el registro `eax` para t_2 tiene el valor 50, y después del `add` tiene el valor 51, que finalmente se guarda en la variable S . Es decir, ahora la variable compartida S tiene el valor 51.

Ahora hay un timer interrupt, se devuelve control a t_1 , y t_1 termina de operar. Sólo le faltaba la última instrucción, el `mov %eax, 0x8049a1c`. Pero esto setea la variable S en 51, porque ese es el valor del registro `%eax` para t_1 .

El problema es claro. Ambas threads terminaron de ejecutar, pero la variable S sólo aumentó su valor en uno. La raíz de este problema es que no se está regulando de manera adecuada el acceso al recurso compartido S , y se está dando una *race condition*.

(★) Glosario:

- **Sección crítica:** Una región de código que accede a un recurso compartido, generalmente una variable o estructura de datos.
- **Race condition:** Situación que ocurre cuando múltiples threads entran a una sección crítica más o menos al mismo tiempo, intentando modificar el recurso compartido y produciendo resultados inesperados.
- **Programa indeterminado:** Un programa con una o más race conditions. El output del programa varía dependiendo de qué threads se corran en cada momento. Por ende, no es determinístico.
- **Exclusión mutua:** Restricción que se impone a las secciones críticas, garantizando que sólo una thread pueda ingresar a las mismas, evitando así condiciones de carreras.

16.2 API de threads

Creación. La creación de una thread se hace con

```
#include <pthread.h>
```

```

int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void*),
    void *arg)
)

```

Revisemos uno por uno los argumentos de la función. `thread` es un puntero a una estructura de tipo `pthread_t` que se usará para interactuar con la thread. `attr` es constante y son los atributos de la thread (el tamaño de su stack, por ejemplo). Si se deja en `NULL`, se usan los atributos por defecto. El tercer argumento es más compliquetí y merece atención.

En cierto modo, el argumento pregunta: ¿en qué función debería iniciar la ejecución de esta thread? El nombre del argumento, `start_routine`, denota justamente la función a ejecutar al iniciar. Dicha función debe tomar un argumento de tipo `void*` (paréntesis derecho) y devolver un tipo `void *`.

El último argumento `arg` es el argumento que debe pasarse a la `start_routine`.

Terminación. Para terminar una thread se usa

```
int pthread_join(pthread_t thread, void **value_ptr);
```

El primer argumento es la thread por la que se debe esperar; es la variable inicializada en `pthread_create`. El segundo argumento es un puntero al valor que se espera recibir de la thread. Como la rutina de la thread puede devolver cualquier cosa, se devuelve un puntero a `void`, que luego puede ser casteado al tipo que se desee.

Es importante que la rutina de una thread no devuelva punteros que apunten a algo guardado en el stack de la thread. El stack se destruye al terminar la thread, y el puntero quedaría apuntando a memoria inválida.

16.3 Locks

Un *lock* es una primitiva de sincronización que permite implementar exclusión mutua. Tienen dos rutinas básicas:

- `int pthread_mutex_lock(pthread_mutex_t *mutex)`: Intenta tomar el lock `mutex`. Si el lock ya está tomado por otra thread, la thread que llama a esta rutina queda bloqueada hasta que el lock se libere.

- `int pthread_mutex_unlock(pthread_mutex_t *mutex)`: Libera el lock. Si hay threads esperando por el lock, una de ellas (cualquiera) lo toma y continúa su ejecución.

Los locks sirven para regular acceso a secciones críticas. Las funciones anteriores asumen que existe una variable de tipo `pthread_mutex_t` que representa el lock. Dicha variable debe inicializarse antes de usarla. Las threads de POSIX pueden inicializarse como

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

lo cual setea los valores del thread a sus valores por defecto. Alternativamente, se puede inicializar dinámicamente con

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // Check success
```

donde `lock` es la dirección del lock y el segundo argumento son los atributos (NULL da los atributos default).

16.4 Variables de condición

Una variable de condición es una primitiva de sincronización que permite a threads esperar hasta que ocurra una condición particular. Las variables de condición se usan junto con locks para implementar sincronización entre threads. Tienen tres rutinas básicas:

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`: Libera el lock `mutex` y bloquea la thread hasta que otra thread llame a `pthread_cond_signal` en la variable de condición `cond`. Cuando la thread se despierta, vuelve a tomar el lock `mutex` antes de continuar su ejecución.
- `int pthread_cond_signal(pthread_cond_t *cond)`: Despierta una thread que esté esperando en la variable de condición `cond`. Si no hay threads esperando, no hace nada.

(†) Por ejemplo, imaginemos este código corriendo en una thread:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```

Pthread_mutex_lock(&lock);
while (ready == 0) {
    pthread_cond_wait(&cond, &lock);
}
Pthread_mutex_unlock(&lock);

```

En otra thread podríamos tener:

```

Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);

```

La primera thread toma el lock y chequea la variable `ready`. Si `ready` es cero, la thread llama a `pthread_cond_wait`, que libera el lock y bloquea la thread hasta que otra thread llame a `pthread_cond_signal` en la variable de condición `cond`. Cuando la thread se despierta, vuelve a tomar el lock antes de continuar su ejecución.

La segunda thread toma el lock, setea `ready` en 1, llama a `pthread_cond_signal` en la variable de condición `cond`, y libera el lock. Esto despierta a la primera thread, que vuelve a tomar el lock y continúa su ejecución.

Observaciones:

- Cuando una thread señala o modifica la variable de condición, debemos asegurarnos de que dicha thread tiene el lock. De otro modo podríamos introducir una race condition.
- La llamada a `wait` toma el lock como su segundo parámetro. Esto es así porque la llamada no solo pone la thread a dormir, sino que libera el lock.
- La waiting thread verifica la condición en un ciclo `while`. Esto es importante porque cuando la thread se despierta, no hay garantía de que la condición que estaba esperando se haya cumplido. Podría haberse despertado por una señalización de otra thread, o por un *spurious wakeup*. Por ende, siempre debe verificar la condición en un ciclo.

16.5 Implementando locks

Una lock es una variable y por ende debe tener un tipo. Más generalmente, un lock es una data structure que contiene la información necesaria para implementar exclusión mutua. Pero esto puede implementarse de muchas maneras.

Para comparar distintas implementaciones, usamos tres criterios. El más obvio es: ¿permite garantizar exclusión mutua? El segundo es **fairness**: ¿todas las threads que intentan tomar el lock tienen una chance justa de conseguirlo? La tercera es **performance**: ¿cuál es el overhead introducido por la implementación del lock?

Interrupt-based locks. Una forma primitiva de implementar locks es controlando los interrupts. Si justo antes de entrar a una sección crítica apagamos los interrupts, por ejemplo con una instrucción especial del hardware, garantizamos que ninguna otra thread pueda interrumpirnos mientras estamos en la sección crítica. Lo bueno de esto es que es simple, pero tiene muchos contras. Requiere una operación privilegiada (apagar los interrupts) e implica confiar en que dicha operación no sea abusada. Además, no funciona si hay más de un procesador: si dos threads, en distintos procesadores, intentan entrar a la sección crítica al mismo tiempo, ambas lo lograrán.

Busy-waiting locks. Otro intento es usar *busy waiting*. La idea es tener una variable compartida flag que indique si el lock está tomado o no. La primera thread en entrar a la sección crítica llama lock(), que

- Se fija si `flag == 0`. Si es así, setea `flag = 1` y entra a la sección crítica.
- Si `flag == 1`, queda en un ciclo infinito hasta que `flag == 0`, y luego setea `flag = 1` y entra a la sección crítica.

Este sistema tiene problemas de corrección de y de performance. La incorrección sucede cuando consideramos programas concurrentes. Asuma que `flag = 0` cuando empieza a correrse lo siguiente:

THREAD 1

```
-----  
call lock();  
while (flag == 1)  
INTERRUPT: SWITCH TO THREAD 2  
flag = 1
```

THREAD 2

```
call lock();  
while (flag == 1)  
flag = 1  
INTERRUPT: SWITCH TO THREAD 1
```

Claramente, ambas threads entraron a la sección crítica, violando la exclusión mutua. El problema de performance se sigue del spin-waiting.

Test-and-set locks. Una mejora sobre el esquema anterior es usar una hardware instruction atómica denominada **test-and-set**. Si expresáramos la instrucción en C, sería algo así:

```
int TestAndSet(int *old_ptr, int new){  
    int old = *old_ptr;  
    *old_ptr = new;  
    return old;  
}
```

La instrucción toma un puntero a una variable y un nuevo valor. La instrucción guarda el valor actual de la variable en una variable temporal `old`, setea la variable apuntada por `old_ptr` en `new`, y devuelve el valor original de la variable. La clave es que todo sucede atómicamente: ninguna otra thread puede interrumpir la ejecución de la instrucción.

Resulta que esta instrucción simple nos permite construir un spin lock que funciona. La idea es tener una variable compartida `lock` que inicialmente es cero (indica que el lock está libre). La rutina `lock()` hace lo siguiente:

```
typedef struct __lock_t {  
    int flag;  
} lock_t;  
  
void init(lock_t *lock) {  
    // 0: lock is available, 1: lock is held  
    lock->flag = 0;  
}  
  
void lock(lock_t *lock) {  
    while (TestAndSet(&lock->flag, 1) == 1)  
        ; // spin-wait (do nothing)  
}  
  
void unlock(lock_t *lock) {  
    lock->flag = 0;  
}
```

Si razonamos, vemos que esto sí garantiza exclusión mutua. Sin embargo, no garantiza fairness: una thread podría quedar spinning para siempre. El costo de performance sigue siendo el mismo que antes para una sola CPU, pero en múltiples CPUS funciona decentemente.

17 File system

17.1 Files, directories y API básica

Un *file* es la abstracción básica del sistema de archivos. Es una array de bytes que pueden leerse o escribirse. Es identificada con un **inode number** único (low level) y un nombre (high level).

Un directorio es un archivo que contiene una lista de 2-uplas (name, inode number), cada una de las cuales identifica a otro archivo (o directorio). La ubicación de directorios dentro de otros directorios conforma el **directory tree**.

La interfaz del sistema de archivo consiste de llamadas a sistema, algunas de las cuales ya hemos visto antes (`open()`, `write()`, `read()`).

Dentro de cada proceso, los archivos son representados vía **file descriptors** (fd). En este sentido un fd es una handle opaca que "apunta" a un archivo y permite al proceso acceder a él.

Como los fd son de cada proceso, la estructura `proc` debe tener un registro de los archivos disponibles al proceso. En particular,

```
struct proc {  
    ...  
    struct file *ofile[NOFILE]; // array of open files  
    ...  
}
```

Acá NOFILE es el número máximo de archivos que pueden tenerse abiertas. La array `ofile` es indexada con los file descriptors. Cada entrada en la array es un puntor a un `struct file`, que veremos más adelante.

(†) Recordar que la estructura `proc` es una estructura de datos dentro del kernel que contiene toda la información necesaria para gestionar un proceso, como su estado, identificador, descriptores de archivos, memoria y relaciones con otros procesos.

Una llamada a sistema que no hemos mencionado aún es `lseek`. Por cada archivo que un proceso abre, el SO registra el **file offset** (inicializado en cero). El file offset determina en qué byte iniciará la próxima lectura/escritura del archivo (recordar que archivo = array de bytes). Cuando llamamos `read` o `write` con N bytes, al offset se le suma N implícitamente para que la próxima lectura/escritura empiece donde la anterior terminó.

La llamada `lseek` modifica explícitamente el offset de un archivo. Toma un `fd`, un `offset`, y un entero `whence` que dictamina si el `offset` debe fijarse directamente, si debe sumarse al offset actual, o si debe sumarse al tamaño del archivo.

El offset, junto con otras cosas, se guarda en el `struct file`.

```
struct file {
    int ref; // ref count, will be discussed later
    char readable; // read permission
    char writable; // write permission
    struct inode *ip; // to which underlying file it refers
    uint off; //the offset
}
```

Estos `struct file` representan todos los archivos abiertos en el sistema, y tomados en su conjunto conforman la `open file table`. El kernel los guarda en una array con un lock:

```
struct {
    struct spinlock lock;
    struct file file[NFILE];
} ftable;
```

Como ya sabemos, el mapeo `file descriptor ---> file table entry` no siempre es uno a uno. Por ejemplo, la file table es compartida por padres e hijos tras un `fork()`. Si un padre tiene el fd *n*, el hijo también lo tendrá. Y si el hijo actualiza el offset del archivo apuntado por el fd *n* usando (por ejemplo) `lseek`, el padre compartirá ese. Es decir que no sólo el fd es el mismo en padre e hijo, sino que dicho fd apunta a la misma entrada en la file table.

Precisamente porque varios procesos pueden referenciar al mismo archivo, el `struct file` lleva un `ref count`, que cuenta cuántos procesos tienen al archivo abierto. Notar que cuando hacemos `close` sobre un fd, el `ref count` del archivo señalado por él se decrementará en uno, y sólo se eliminará de la file table si el `ref count` ahora es cero.

Otro caso interesante lo generan las llamadas `dup`, `dup2`, `dup3`. Si recordamos, `dup` crea un nuevo fd que se refiere al mismo archivo abierto que un fd pre-existente. Es decir, `int new_fd = dup(old_fd)` produce un estado en que `new_fd` y `old_fd` apuntan al mismo archivo.

Otra llamada a sistema interesante es `fsync`. Para entenderla, repasemos un poco un detalle de la syscall `write`. En particular, recordemos que cuando un proceso llama `write(fd, data, data_size)`, el SO puede demorar la escritura por cierta cantidad de tiempo (digamos 5 o 30 segundos), guardando `data` en memoria (RAM). Más específicamente, el SO copiará los bytes de `data` en una página que está en RAM. Dicha página será marcada con un **dirty bit**, y diremos que `data` está **dirty**, i.e. esperando a ser escrita en el disco persistentemente. La llamada a `write` termina rápidamente, pero hablando rigurosamente el servicio que solicitó no termina hasta que `data` sea pasada de la página en RAM al disco.

Esta característica de `write` es razonable, pero a veces necesitamos forzar la escritura en disco de nuestra `data`. La llamada `fsync`) toma un fd y el sistema de archivos responde forzando que toda

dirty data asociada al `fd` sea escrita en disco. Interesantemente, para forzar la escritura de un archivo en disco, no sólo hay que llamar `fsync` sobre su `fd`, sino sobre el `fd` del directorio que lo contiene. Así garantizamos que el archivo modificado también haya quedado persistentemente escrito "desde la perspectiva" del directorio.

17.2 Links

17.2.1 Hard links

La función `rm` que borra archivos usa una llamada a sistema llamada `unlink`. Para saber qué es `unlink` hay que saber qué es un `link` (brillancy prize observation).

La llamada a sistema `link()` toma un `old_path` y un `new_path`, y es lógicamente análoga a `dup`. Después de la llamada, el filename `new_path` hará referencia al mismo `file` que el filename `old_path`.

En el bajo nivel, `link()` funciona creando otro nombre en el directorio donde estamos creando el `link` y haciendo que refiera al mismo **inode number** que el del archivo original. El archivo *no es copiado*, solamente son dos nombres (alto nivel) apuntando al mismo archivo (bajo nivel).

Cuando creamos un archivo (e.g. usando `open()` con la flag `O_CREATE`), se hacen dos cosas:

- Se crea una nueva estructura en disco, el **inode**, que llevará registro de la información pertinente a dicho archivo (tamaño, dónde reside en disco, etc).
- Se crea un *link* que vincula un file name (humanamente legible) con dicho archivo.

¿Y por qué hacer `unlink` para eliminar un archivo? Cuando hacemos `unlink()` sobre un archivo, se chequea el **reference count** dentro del inode. El ref count del inode (a veces llamado **link count**) nos dice cuántos file names están linkeados a dicho inode. Cuando se llama `unlink()`, dicho ref count se decrementa en uno, y si es cero el file system elimina el archivo.

(†) No confundir estas dos cosas:

- El file struct, que se guarda *en memoria*, y contiene entre otras cosas el **ref count** de cuántos procesos tienen el archivo abierto. Si dicho ref count se vuelve cero, el proceso es eliminado de la file table, que también es una estructura que está *en memoria*.
- El inode, que se guarda *en disco*, y contiene otras cosas el ref count de cuántos links vinculan filenames con dicho inode (i.e. el link count). Si el link count se vuelve cero, el inode y sus data blocks en disco son liberados.

17.2.2 Symbolic links

Los links descritos arriba son llamados *hard links*. Existe otro tipo de link llamado *symbolic link*. Se diferencian en varios aspectos.

- (1) Un symbolic link *es un archivo* de un tipo especial (*symbolic link*).
- (2) Un symbolic link lleva consigo el *pathname* del archivo original. Esto significa que si hacemos un symbolic link a un archivo llamado abcd, el tamaño del symbolic link será de 4 bytes (porque abcd son cuatro chars).
- (3) Abren la posibilidad de que quede una **referencia colgante** (dangling reference). Para entenderlo, considere los comandos:

```
prompt> echo hello > file
prompt> ln -s file file2 // ln -s crea el symbolic link
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

¿Cuál es el problema? Como el symbolic link file2 estaba vinculado a file a través de su pathname, y no por referencia al file de bajo nivel subyacente, al hacer rm file el archivo subyacente desaparece y también lo hace el pathname file. Al hacer cat file2, el vía el symbolic link se busca el archivo file, pero ya no existe.

17.3 Permission bits

Los archivos y directorios, en contraste con otras abstracciones que permitían a los procesos vivir como si tuvieran su propia memoria o su propia CPU, son *comunes*. Por ende, se necesitan mecanismos que regulen la forma en que se comparten.

El primero son los **permission bits**. Los permission bits establecen qué puede hacer el dueño de un archivo, lo que alguien en un **grupo** puede hacer, y lo que otros (ni dueño ni miembros del grupo) pueden hacer. Con "pueden hacer" siempre hablamos de **read**, **write**, **read/write**.

El dueño del archivo puede cambiar estos permisos usando **chmod**.

17.4 Creando y montando un file system

Para crear un file system se usa típicamente la herramienta `mkfs` (make fs), provista por el file system que será creado. La herramienta toma como input un dispositivo (e.g. una partición del disco), el tipo del file system (e.g. ext3), y simplemente escribe un file system vacío, con solo un root directory, en dicha partición.

Ahora bien, una vez que es creado, el file system (FS) debe estructurarse en forma de árbol. Esto es lo que se llama **montar** el FS. La syscall `mount()` toma un directorio y lo convierte en un "portal" que conecta con el nuevo file system.

(†) **Ejemplo.** Digamos que tenemos un FS en `/dev/sda1` con carpetas `dirA`, `dirB`, con archivos `fileA` y `fileB` respectivamente. Digamos que dicho FS no ha sido montado, pero ha sido creado en el directorio `/home/newfsroot`.

Nuestro FS principal, con root en `/`, y el FS en `/dev/sda1`, están *totalmente separados*. Si hacemos `ls /home/newfsroot`, no veremos nada. El FS principal no tiene forma de acceder al sistema que fue creado dentro de él.

Digamos que ahora hacemos `mount -t ext3 /dev/sda1 /home/newfsroot`. Esto le dice al SO: agarrá el FFS que está en `/dev/sda1` y hace que su root directory *sea visible en* `/home/newfsroot`. Es decir que ahora `/home/newfsroot` actúa como una especie de portal entre el FS principal y el FS que está en `/dev/sda1`. Ahora, hacer `ls /home/fsroot` veremos las carpetas `dirA`, `dirB`.

En resumen, `mount` hace que un FS sea *accesible* "pegándolo" en un directorio existente, haciendo que todo parezca un sólo árbol unificado en vez de dos árboles separados que no se conectan entre sí.

18 File system implementation

Acá se presenta el **Very Simple File System** o **vsfs**, un FS concreto. Es puramente software, i.e. no necesita hardware adicional que lo asisita.

(★) Cuando pensás en un FS, pensá en dos aspectos separados pero interrelacionados:

- ¿Qué data structures usa el FS? En particular, ¿qué estructuras están siendo guardadas en el disco para organizar data y metadata?
 - ¿Qué métodos de acceso provee? ¿Cómo mapea las llamadas de un proceso, como `write`, `read`, `open`, a dichas estructuras? ¿Qué estructuras son leídas en una syscall particular? ¿Cuáles son escritas?

Vamos a empezar con la organización de las data structures en disco. Primero, dividimos el disco en **bloques**, y usaremos un BLOCK_SIZE de 4KB. Asumamos que nuestro disco es muy pequeño y tiene solo 64 bloques, para un tamaño total de $64 \times 4 \text{ KB} = 256\text{KB}$.

Vamos a partir esto en una **data region**, donde estarán los datos del usuario. Para llevar adelante qué bloques de la data region están siendo usados, usaremos un **bitmap**.

(★) En su forma más simple, un bitmap es una array de bits, como `bitmap = [0, 0, 1, 0, 0, 1, 1, ...]`, tales que `bitmap[i]` representa alguna información relevante respecto al i-ésimo elemento de alguna otra secuencia de datos.

Por ejemplo, en un *labeled graph* de n vértices podemos tener un bitmap de n bits tal que $\text{bitmap}[i] = 1$ si y solo si el i -éximo vértice es de grado mayor a 3.

En nuestro caso, `bitmap[i]` = 1 si y solo si el i-ésimo bloque está siendo utilizado.

Además de la **data region**, necesitamos guardar los **inodes**. Recordemos que un **inode** es una estructura en disco que lleva información respectiva a un archivo, como su tamaño, sus permisos, etc. Por ende, definimos una **inode region**, i.e. un conjunto de bloques donde guardaremos los inodes. Los llamaremos **inode blocks**, en contraste con los **data blocks**.

Para saber qué **inode blocks** están en uso, también deberemos guardar en disco un **inode bitmap**. Asumamos que usamos 5 de los 64 bloques para la inode region y un solo bloque para el inode bitmap. Asumamos además que cada inode es de 256 bytes, lo cual significa que un bloque de 4KB puede contener 16 inodes. Como existe un inode por archivo, y tenemos 5 inode blocks, esto significa que nuestro sistema podrá guardar $16 \times 5 = 80$ archivos.

En resumen, la estructura básica de nuestro FS es así.

[S] [i] [d] [I] [I] [I] [I]	[D] [D] [D] [D] [D] [D] [D]				
0	7 8	15 16	23 24	31	

[D] [D] [D] [D] [D] [D] [D]	[D] [D] [D] [D] [D] [D]				
32	39 40	47 48	55 56	63	

donde [i] es el inode bitmap, [d] es el data bitmap, [I] es un inode block, y [D] es un data block. Se ve claramente que los bloques 3, 4, ..., 8 son la inode region, 8, 9, ..., 63 la data region. El bloque inicial [S] es un bloque especial llamado **superblock**, que contiene información sobre el file system en particular (e.g. cuántos inode blocks y data blocks hay, dónde empieza cada región, etc.).

18.1 ¿Cómo leer un inode?

Cada **inode** está implícitamente referenciado por un número identificador, llamado **i-number**. Es lo que antes llamamos el nombre de bajo nivel de un archivo. En **vsfs**, dado un i-number, podemos calcular dónde en disco está el inode correspondiente.

Digamos que queremos leer el inode 32. El FS debe hacer lo siguiente:

1. Calcular la dirección absoluta en bytes, i.e. en qué byte address está el inode.
2. Como el disco se lee no con byte addresses, sino con sectores, hay que determinar a qué sector del disco corresponde la byte address calculada en (1). convertir la byte address calculada.
3. Determinar, dentro del sector del disco, en qué offset reside el inode.

El paso (1) es relativamente simple. La dirección absoluta del inode será

```
// Calculamos el offset del inode *dentro de la inode region*.
INODE_OFFSET_RELATIVE = INUMBER * INODE_SIZE // 32 * 256bytes = 8192
// Vemos dónde empieza la inode region, esto está en el bloque S.
INODE_REGION_OFFSET = 12KB
// La byte address del inode será la suma de las dos anteriores.
INODE_OFFSET_ABSOLUTE = INODE_REGION_OFFSET + INODE_OFFSET_RELATIVE
// Nos queda el offset absoluto = 12288 + 8192 = 20480 bytes = 20KB
```

Ahora viene el paso (2). Típicamente, el tamaño de cada sector en disco es de 512 bytes.

```
SECTOR_NUMBER = INODE_OFFSET_ABSOLUTE / SECTOR_SIZE  
// Nos queda SECTOR_NUMBER = 20480 bytes / 512 bytes = 40
```

Ahora que sabemos el sector en disco, tenemos que determinar dónde en dicho sector reside el inode.

```
OFFSET_IN_SECTOR = ABSOLUTE_BYTE_ADDRESS % SECTOR_SIZE  
// Es el resto de la división entre la dirección absoluta en bytes y el  
// tamaño de sector. Nos queda 20480 % 512 = 0
```

(★) ¿Qué demonios es un **inode** exactamente? Ya dijimos que es una estructura de datos en disco que contiene la metadata de un archivo. ¿Pero qué metadata? La respuesta es: *una banda de metadata*. Entre otras cosas,

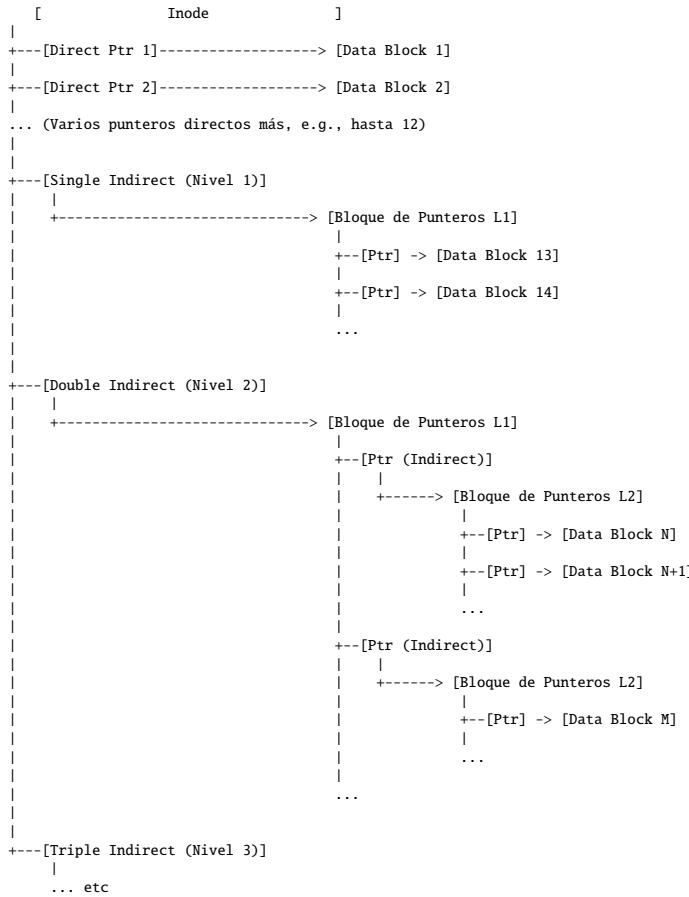
- Permisos de escritura/lectura/ejecución.
- Owner.
- Tamaño.
- Información temporal (cuándo fue creada, cuándo fue modificada por última vez, cuándo fue leída por última vez, etc)
- A qué grupo pertenece.
- Cuántos hard links hay al archivo.
- Cuántos bloques fueron asignados para el archivo.
- (★) Un conjunto de 15 punteros que apuntan a los bloques donde está el archivo.

y muchas cosas más. La última es muy importante, porque es lo que nos permite usar el inode para determinar dónde en memoria está el archivo propiamente dicho (i.e. la array de bytes).

En el cuadro anterior notamos que los inodes tienen *K direct pointers*, punteros a bloques donde está el archivo, con $K = 15$ típicamente. En particular, si el archivo es tan grande que necesita más de 15 bloques para ser escrito, los 15 punteros no van a alcanzar. ¿Cómo resolverlo?

18.2 Multi-level indexing

Una solución al problema anterior es otorgar al inode no sólo **direct pointers**, i.e. punters a direcciones en el disco, sino también **indirect pointers**. Un **indirect pointer** no apunta a un bloque de datos, sino a un bloque que contiene más punteros. Los punteros en el bloque apuntado por un **indirect pointer** pueden a su vez ser directos (apuntar a datos) o indirectos (apuntar a otros bloques con más punteros), con lo cual formamos una estructura multi-nivel que corresponde a un árbol desbalanceado. Este árbol desbalanceado está representado en el diagrama abajo:



Si un archivo es demasiado grande para guardarse en tantos bloques como punteros directos hay, se alloca en memoria un **indirect block** (tomado de la región de datos) con punteros (directos o indirectos) y se hace que un **indirect pointer** del inode apunte a él. Cuántos indirect pointers se le da a un inode es una cuestión de diseño.

(†) **Ejemplo.** Asumamos que un inode tiene 12 punteros directos. Asumamos bloques de 4KB con direcciones en disco de 4 bytes. Un solo **indirect block** puede entonces contener 1024 direcciones, i.e. 1024 punteros. Esto significa que el archivo ahora puede ser de tamaño $(12 + 1024) \cdot 4K = 4144KB$, lo cual supera ampliamente el tamaño $15 \times 4KB = 60KB$ permitido por los 15 punteros

directos. Notar cuánto crece el tamaño posible agregando *un solo* puntero indirecto de *un solo nivel*.

Imaginemos que ahora nuestro inode tiene 12 punteros directos, un puntero indirecto de un nivel, y un puntero indirecto de dos niveles. Entonces el tamaño máximo de archivo es $(12 + 1024 + 1024^2) \cdot 4KB = 4GB$. Si agregáramos además un puntero indirecto más pero de tres niveles, podríamos tener archivos de tamaño $(12 + 1024 + 1024^2 + 1024^3) \cdot 4KB = 4TB + 4GB + 4MB + 48KB$. Una locura.

18.3 Reading y writing

Cada directorio es un tipo especial de archivo y por lo tanto tiene su propio inode. Cada archivo tiene un **access path** que indica dónde en el árbol se encuentra, iniciando desde el root directory (raíz del árbol). Acá mostramos la ridícula cantidad de reads y de writes que se hacen en operaciones sencillas.

Tomemos `open(foo/bar)` y asumamos que `bar` es de 12KB (3 bloques). Para leer el archivo, el FS debe recorrer el **pathname** para ubicar el inode, empezando por el root directory. El root directory suele tener el inumber 2, así que lo primero que se hace es leer este inode.

Una vez que el inode 2 es leído, el FS puede ver lo que hay dentro para encontrar los punteros a los bloques de datos donde residen los contenidos de root. El FS va a usar estos punteros para buscar una entrada que corresponda a `foo`, leyendo bloque por bloque. Una vez que lo encuentre, tendrá el inode number de `foo` (digamos 44).

Ahora hace lo mismo: lee el inode 44 y busca `bar` usando los punteros. Finalmente, lee `bar` para pasarlo del disco a la memoria, alocando un FD en la open-file table del proceso, y lo devuelve.

Completar.

18.4 Caching and buffering

Completar.

19 Fast file system

Los FS primitivos tenían un problema serio de performance, debido principalmente a la fragmentación. Usaban una **free list** (en vez de bitmaps) para manejar el espacio libre, y la free list

apuntaba a bloques muy distantes entre sí. La consecuencia es que datos lógicamente contiguos terminaban guardados en bloques muy distantes, y acceder a ellos requería recorrer el disco constantemente. La pregunta central entonces es: ¿cómo organizar la información en el disco para mejorar la performance? ¿Cómo mejorar las estructuras del FS para el mismo fin? El objetivo central es que el FS sea **disk aware**, i.e. que opere con "conocimiento" de cómo funciona el disco.

19.1 El grupo de cilindros

Primero, cambiamos las **on-disk structures**. El **fast file system** (FFS) divide el disco en **cylinder groups**. Cada **cilindro** es un conjunto de vías en distintas superficies del disco, a determinada distancia del centro. Se agrupan los cilindros en conjuntos de N cilindros contiguos, **llaves cylinder groups**.

(†) El orden, de abajo hacia arriba, entonces es:

1. Sectores, la unidad más pequeña del disco.
2. Vías (tracks), círculos concéntricos sobre la superficie del disco, compuestos de muchos sectores.
3. Cilindros, que no son parte de una organización física sino estrictamente lógica. Un cilindro es un conjunto de vías, cada una en un nivel (superficie) distinto del disco, con la misma distancia al centro.
4. Grupos de cilindros, también organización lógica. Es un conjunto de cilindros que el FS considera como agrupados.

Dato importante: cuando la read/write head del disco está en una posición dada, puede leer/escribir el sector correspondiente de cualquiera de las superficies del disco. Es decir que, si ubicamos la read/write head en un lugar, podemos leer el sector correspondiente a cualquiera de las vías, es decir cualquier valor del cilindro sobre el cual está la read/write head.

Esto ya nos da una clave. Si un archivo está guardado en muchos bloques, pero cada bloque está "uno arriba del otro" en el disco, i.e. podemos leer cualquiera de los bloques sin mover la read/write head.

Un detalle importante es que los discos no exportan ninguna información de su geometría o estructura física. Sólo exportan un address space lógico conformado por bloques. Por ende, los FS modernos se organizan en función de **block groups**, cada uno de los cuales es una porción contigua de dicho address space.

Da igual si los llamamos cylinder groups o block groups, son la pieza central que permite mejorar

la performance. Si dos archivos están en el mismo grupo, garantizamos que acceder a uno después del otro no requerirá mover demasiado el cabezal.

Para mantener todo esto