Sistemas Operativos 2024
Parte I - Virtualización - Notas Teóricas

Estudiante Villar Pedro
Profesor Nicolas Wolovick

Índice

Índice de figuras

1. Procesos

Un programa es un objeto estático, mientras que un proceso es un objeto dinámico. Siempre que hablemos de procesos o programas vamos a estar hablando de código maquina. El SO toma este objeto estático, luego acomoda toda la memoria, luego carga este objeto estático en memoria y lo ejecuta, convirtiéndolo en un objeto dinámico, el proceso.

Definición

Un proceso es una instancia de un programa en ejecución, en cambio un programa es un archivo que contiene una variedad de información que describe como construir un proceso en tiempo de ejecución. Un programa puede usarse para construir muchos procesos o, dicho de otro modo, muchos procesos pueden estar ejecutando el mismo programa.
Si hablamos en términos del kernel, un proceso es una entidad abstracta, definida por el kernel, a la cual se le asignan recursos del sistema para ejecutar un programa. Desde el punto de vista del kernel, un proceso consiste en memoria en el espacio de usuario que contiene el código del programa y las variables utilizadas por dicho código, así como una serie de estructuras de datos del kernel que mantienen información sobre el estado del proceso. La información registrada en estas estructuras de datos del kernel incluye varios números identificadores (IDs) asociados con el proceso, tablas de memoria virtual, la tabla de descriptores de archivos abiertos, información relacionada con la entrega y manejo de señales, uso y límites de recursos del proceso, el directorio de trabajo actual, y una multitud de otra información. [a]

[a] The Linux programming interface

Con esto surge un **problema**, si se tienen dos cores y cuatro procesos, los cuales se quieren ejecutar en los cores. Se quiere saber si es posible ejecutar los procesos en los cores de manera tal que se maximice la utilización de los cores.

¿Cómo se resuelve este problema? Se realiza a través de **mecanismos de bajo nivel**, código de máquina con ciertas instrucciones especiales de la CPU, la CPU real se distribuye mediante lo llamado **políticas**. Es decir que las política son el cómo se van a distribuir esos recursos escasos entre los procesos. Las políticas de Planificación y los planificadores son los encargados de esto.
[1]

1.1. Definición y rol del SO

Podemos definir al proceso o tarea como la abstracción de un programa en ejecución. Un proceso es un conjunto que reune lo siguiente:

- Las **instrucciones y datos** del programa, también conocido como *espacio de direcciones*.
- Información sobre el **estado** de la ejecución del programa.
- Información de **control** sobre el programa para el SO.

El sistema operativo se va a encargar de lo siguiente:

- Intercalar la ejecución de los procesos en un orden tal que maximiza la utilización del procesador.
- Gestionar y asignar los otros recursos del sistema (memoria, E/S, etc.) para que los procesos puedan hacer uso de ellos.
- Gestionar la comunicación de datos entre diferentes procesos presentes en el sistema.
- Ayudar en la creación de nuevos procesos por parte de los usuarios.

[2]

[1] OSTEP Cap.4: La Abstracción de los Procesos
[2] Clase 3 - Hugo Carrer - FCEFyN - UNC

2. API de Procesos

Las operaciones típicas que se pueden realizar sobre un proceso son **crear**, **liberar recursos**, **esperar**, **controlar** y **saber su estado**:

- **Crear:** Un sistema operativo debe incluir algún método para crear nuevos procesos. Cuando escribes un comando en la shell o haces doble clic en el ícono de una aplicación, el sistema operativo es invocado para crear un nuevo proceso que ejecute el programa que has indicado.
- **Liberar recursos:** Así como hay una interfaz para la creación de procesos, los sistemas también proporcionan una interfaz para destruir procesos de manera forzada. Por supuesto, muchos procesos se ejecutarán y saldrán por sí mismos cuando terminen; sin embargo, cuando no lo hacen, el usuario puede desear matarlos, y por lo tanto, una interfaz para detener un proceso descontrolado es bastante útil.
- **Esperar:** A veces, un proceso necesita esperar a que otro proceso termine antes de continuar. Por ejemplo, un proceso puede necesitar esperar a que un proceso hijo termine antes de continuar. En este caso, el proceso padre puede esperar a que el proceso hijo termine.
- **Controlar:** Además de matar o esperar un proceso, a veces existen otros controles posibles. Por ejemplo, la mayoría de los sistemas operativos proporcionan algún tipo de método para suspender un proceso (detener su ejecución por un tiempo) y luego reanudarlo (continuar su ejecución).
- **Saber su estado:** Por lo general, hay interfaces para obtener información sobre el estado de un proceso, como cuánto tiempo ha estado ejecutándose o en qué estado se encuentra.

2.1. Creación de procesos

Tenemos que transformar un programa en un proceso, lo primero que debe hacer el sistema operativo para ejecutar un programa es cargar su código y cualquier dato estático en la memoria, dentro del espacio de direcciones del proceso. Inicialmente, los programas residen en el disco en algún tipo de formato ejecutable.

Una vez que el código y los datos estáticos están cargados en la memoria, hay algunas otras cosas que el sistema operativo necesita hacer antes de ejecutar el proceso. Se debe asignar algo de memoria para la pila (**stack**) de ejecución del programa, se asigna esta memoria y se la da al proceso. El sistema operativo probablemente también inicializará la pila con argumentos; específicamente, llenará los parámetros de la función `main()`, es decir, `argc` y la matriz `argv`.

El sistema operativo también puede asignar algo de memoria para el montículo del programa (**heap**). En los programas en C, se usa para datos dinámicamente asignados que se solicitan explícitamente; los programas solicitan dicho espacio llamando a `malloc()` y lo liberan explícitamente llamando a `free()`. El **heap** es necesario para estructuras de datos como listas enlazadas, tablas hash, árboles y otras estructuras de datos interesantes. Cabe destacar que será pequeño al principio y a medida que el programa se ejecuta y solicita más memoria a través de la API de la biblioteca `malloc()`, el sistema operativo puede intervenir y asignar más memoria al proceso para ayudar a satisfacer dichas llamadas. [3] Los procesos pueden ser creados por diferentes motivos:

- Un nuevo usuario que intenta acceder al sistema localmente o de manera remota.
- El sistema operativo genera un nuevo proceso para ofrecer un servicio, por ejemplo un servicio de impresión.
- A modo general los procesos pueden crear nuevos procesos llamados hijos.

[4]

2.2. Estados de un proceso

Los procesos pueden estar en uno de los siguientes estados:

- **En ejecución**: En el estado de ejecución, un proceso se está ejecutando en un procesador. Esto significa que está ejecutando instrucciones.
- **Listo**: En el estado listo, un proceso está listo para ejecutarse, pero por alguna razón el sistema operativo ha decidido no ejecutarlo en este momento.
- **Bloqueado**: En el estado bloqueado, un proceso ha realizado algún tipo de operación que lo hace no estar listo para ejecutarse hasta que ocurra algún otro evento. Un ejemplo común: cuando un proceso inicia una solicitud de entrada/salida (I/O) a un disco, se bloquea y, por lo tanto, otro proceso puede usar el procesador.

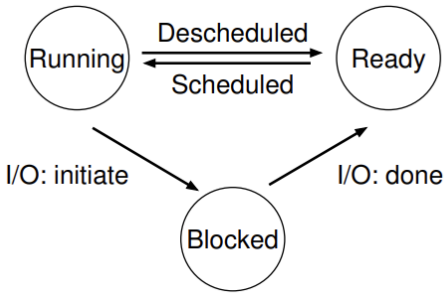

Figura 1: Estados de un proceso

Como se puede ver en el diagrama, un proceso puede moverse entre los estados de **listo** y **en ejecución** a discreción del sistema operativo. Ser movido de listo a en ejecución significa que el proceso ha sido *planificado*; ser movido de en ejecución a listo significa que el proceso ha sido *desplanificado*. Una vez que un proceso se ha bloqueado (por ejemplo, al iniciar una operación de entrada/salida), el sistema operativo lo mantendrá en ese estado hasta que ocurra algún evento (por ejemplo, la finalización de la operación de I/O); en ese momento, el proceso pasa nuevamente al estado listo (y potencialmente inmediatamente al estado de ejecución, si el sistema operativo así lo decide).

2.3. Estructuras de control de procesos

En resumen, el sistema operativo (SO) necesita conocer en todo momento el estado de los recursos administrados y el de todos los procesos del sistema. Con respecto a los recursos, el SO debe conocer el estado de: *Memoria, Dispositivos de E/S y Sistema de archivos*

Los sistemas operativos están llenos de diversas estructuras de datos importantes, la lista de procesos es la primera de estas estructuras. Es una de las más simples, pero ciertamente cualquier sistema operativo que tenga la capacidad de ejecutar múltiples programas a la vez tendrá algo similar a esta estructura para hacer un seguimiento de todos los programas en ejecución en el sistema.

2.4. Creación de procesos en sistemas Unix

2.4.1. Llamada al sistema `fork()`

La llamada al sistema `fork()` es una de las operaciones fundamentales en sistemas operativos UNIX y similares. Su función principal es crear un nuevo proceso duplicando el proceso que la llamó. El proceso original se conoce como el "proceso padre", mientras que el nuevo proceso creado se llama "proceso hijo". Ambos procesos continúan ejecutándose desde la línea donde se llamó a `fork()`, pero con una diferencia clave en su comportamiento debido al valor de retorno de `fork()`. Por ejemplo, veamos el siguiente código:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    printf("hello (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (main)
        printf("parent of %d (pid:%d)\n",
        rc, (int) getpid());
```

[3]OSTEP Cap.4: La Abstracción de los Procesos
[4]Clase 3 - Hugo Carrer - FCEFyN - UNC

```
    }
    return 0;
}
```

[5]

Se puede ver que antes de llamar a `fork()`, el proceso padre imprime su PID. Luego, después de llamar a `fork()`, el proceso hijo imprime su PID y el proceso padre imprime el PID del proceso hijo. La salida de este programa se verá algo así:

```
hello (pid:1234)
parent of 1235 (pid:1234)
child (pid:1235)
```

Pero ¿cómo funciona *fork()*? si fork falla retorna un valor negativo, en este caso el programa imprimirá un mensaje de error y terminará. Si *fork()* tiene éxito, se devolverá dos veces, una vez en el proceso padre y otra en el proceso hijo. En el proceso hijo, el valor devuelto será 0, mientras que en el proceso padre, el valor devuelto será el PID del proceso hijo. Esto es lo que permite que el programa diferencie entre el proceso padre y el proceso hijo.

Resumen

Con esto concluimos que, luego de llamar a `fork()` se crea un nuevo proceso hijo, el cual es una copia exacta del proceso padre. Lo que se clona es lo siguiente:

- Program Text: El código máquina del programa.
- Stack: Las variables locales y los argumentos de la función.
- PCB: El PCB del proceso.
- Data: Las variables globales.

Este comportamiento es esencial para operaciones de multitarea y creación de nuevos procesos en sistemas UNIX, y es la base sobre la que se construyen muchas otras operaciones del sistema operativo.

[6] Veamos ahora otro código:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;
    if (argc != 2){ /* check for valid number of command-line arguments */
        fprintf(stderr, "Usage: %s processes\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    for (i = 1; i < n; i++){
        if (childpid = fork())
        break;
    }
    fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
    i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```

[7] Este programa toma un argumento de línea de comandos que indica cuántos procesos se deben crear. Luego, crea `n` procesos, cada uno de los cuales imprime su PID, el PID de su padre y el PID de su hijo. La salida de este programa se verá algo así:

[5] OSTEP, Cap.5, La API de los Procesos
[6] Pablo Martínez - SO1 Prácticos 2021- FCEFyN - UNC
[7] Unix Systems Programming: Communication, Concurrency, and Threads - Sec 3.3

```
i:1 process ID:1234 parent ID:1233 child ID:1235
i:2 process ID:1235 parent ID:1234 child ID:1236
i:3 process ID:1236 parent ID:1235 child ID:1237
i:4 process ID:1237 parent ID:1236 child ID:1238
```

Graficamente se puede ver como funciona la llamada a `fork()` en el siguiente diagrama:

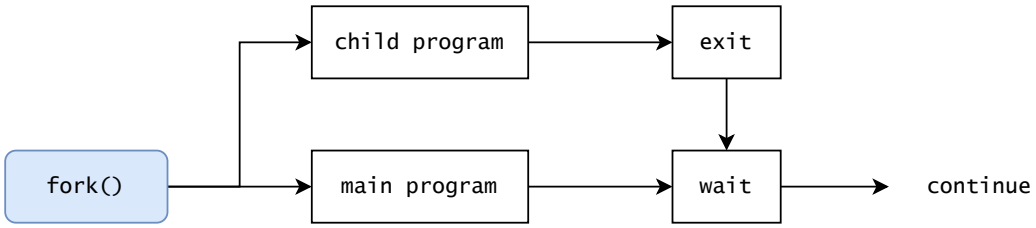

Figura 2: Diagrama de la llamada a `fork()`

Cabe destacar que el hijo no es una copia exacta. Específicamente, aunque ahora tiene su propia copia del espacio de direcciones (es decir, su propia memoria privada), sus propios registros, su propio contador de programa (PC), y demás, el valor que devuelve al que llamó a fork() es diferente. En particular, mientras que el padre recibe el PID del hijo recién creado, el hijo recibe un código de retorno de cero. Esta diferenciación es útil porque facilita escribir el código que maneja los dos casos diferentes.

Se puede notar en el diagrama una nueva llamada de espera `wait()` que se encarga de esperar a que el proceso hijo termine. Si no se llama a `wait()` el proceso hijo se convierte en un proceso huérfano, es decir, un proceso que no tiene padre. La vemos a continuación.

2.4.2. Llamada al sistema `wait()`

Hasta ahora, solo creamos un proceso hijo que imprime un mensaje y termina. A veces, resulta bastante útil que un proceso padre espere a que un proceso hijo termine lo que está haciendo. Esta tarea se realiza con la llamada al sistema `wait()`.

En el ejemplo anterior se puede ver que el proceso padre imprime su PID, luego crea un proceso hijo que imprime su PID y el PID del proceso padre. Pero, ¿qué pasa si el proceso hijo no termina inmediatamente? ¿Qué pasa si el proceso hijo realiza un trabajo que lleva tiempo? En este caso, el proceso padre puede esperar a que el proceso hijo termine. Esto se logra con la llamada al sistema `wait()`. Volvamos ahora para ver esto a otra versión del primer ejemplo:

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

int main(int argc, char *argv[]) {
    printf("hello (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("child (pid:%d)\n", (int) getpid());
    } else { // parent goes down this path
        int rc_wait = wait(NULL);
        printf("parent of %d (rc_wait:%d) (pid:%d)\n",
        rc, rc_wait, (int) getpid());
    }
    return 0;
}
```

[8]

[8]OSTEP, Cap.5, La API de los Procesos

En este caso, el proceso padre crea un proceso hijo y luego llama a `wait()`. La llamada a `wait()` hace que el proceso padre se bloquee hasta que el proceso hijo termine. Una vez que el proceso hijo termina, el proceso padre se desbloquea y continúa su ejecución. La salida de este programa se verá algo así:

```
hello (pid:29266)
child (pid:29267)
parent of 29267 (rc_wait:29267) (pid:29266)
```

9

2.4.3. Llamada al sistema `exec()`

La familia de funciones `exec()` reemplaza la imagen del proceso actual con una nueva imagen de proceso. Cuando un programa llama a una función exec, ese proceso deja de ejecutar el programa actual inmediatamente y comienza a ejecutar un nuevo programa desde el principio.

> Debido a que exec reemplaza el programa que hace la llamada con otro, nunca regresa, a menos que ocurra un error.

Es decir, que si un programa llama a una función exec, el programa actual se detiene y se carga un nuevo programa en su lugar. Por lo tanto, si la llamada a exec tiene éxito, el programa que la llamó nunca regresará. Si la llamada a exec falla, el programa que la llamó continuará ejecutándose como si nada hubiera pasado. Lo que hace es: dado el nombre de un ejecutable (por ejemplo, wc), y algunos argumentos (por ejemplo, test.c), carga el código de ese ejecutable y sobrescribe su segmento de código actual con él; el *stack*, el *heap* y otras partes del espacio de memoria del programa se reinicializan. Luego, el sistema operativo simplemente ejecuta ese programa, pasando los argumentos como el argv de ese proceso. Así, no crea un nuevo proceso; más bien, transforma el programa que se está ejecutando actualmente en un programa diferente (wc). [10]

Ahora bien, ¿por qué es útil esto? Bueno, la llamada a `exec()` es útil porque permite que un proceso ejecute cualquier programa que desee, no solo el programa que se cargó inicialmente. Por ejemplo, un shell puede ejecutar cualquier programa que desee, simplemente llamando a `exec()` con el nombre del programa y cualquier argumento que desee pasar a ese programa. Por lo tanto, `exec()` es una forma de que un proceso cambie de programa a mitad de camino, lo que es útil para muchas aplicaciones.

Existen distintos tipos de llamadas a `exec()` [11]:

- Llamadas que contienen la letra `p` en su nombre, como `execlp()` o `execvp()`, buscan el archivo ejecutable en el *path* del sistema.
- Llamadas que contienen la letra `l` en su nombre, como `execl()` o `execlp()`, requieren que los argumentos del programa se pasen como una lista de argumentos separados por comas.
- Llamadas que contienen la letra `v` en su nombre, como `execv()` o `execvp()`, requieren que los argumentos del programa se pasen como un vector de punteros a cadenas terminadas en `NULL`.
- Llamadas que contienen la letra `e` en su nombre, como `execve()` o `execvpe()`, aceptan un argumento adicional que especifica el entorno del nuevo programa. El argumento debe ser un array de punteros a cadenas terminadas en `NULL` donde cada cadena tiene la forma `nombre=valor`.

2.5. API de Procesos en la shell

El shell es simplemente un programa de usuario. Muestra un prompt y luego espera que escribas algo en él. Luego, escribes un comando (es decir, el nombre de un programa ejecutable, más cualquier argumento) en él; en la mayoría de los casos, el shell averigua dónde en el sistema de archivos reside el ejecutable, llama a `fork()` para crear un nuevo proceso hijo para ejecutar el comando, llama a alguna variante de `exec()` para ejecutar el comando, y luego espera a que el comando termine llamando a `wait()`. Cuando el proceso hijo termina, el shell retorna de `wait()` e imprime nuevamente un prompt, listo para tu próximo comando.

La separación de `fork()` y `exec()` permite que el shell haga un montón de cosas útiles con bastante facilidad. Por ejemplo:

```
$ wc p3.c > newfile.txt
```

[9] Pablo Martínez - SO1 Prácticos 2021- FCEFyN - UNC
[10] OSTEP, Cap.5, La API de los Procesos
[11] Pablo Martínez - SO1 Prácticos 2021- FCEFyN - UNC

En el ejemplo anterior, la salida del programa `wc` se redirige al archivo de salida `newfile.txt`. La forma en que el shell logra esta tarea es bastante simple: cuando se crea el proceso hijo, antes de llamar a `exec()`, el shell cierra la salida estándar y abre el archivo newfile.txt. Al hacer esto, cualquier salida del programa que está a punto de ejecutarse, wc, se envía al archivo en lugar de a la pantalla.

Aunque el shell es un programa bastante simple, es un buen ejemplo de cómo se pueden combinar las llamadas al sistema `fork()`, `exec()` y `wait()` para hacer cosas bastante útiles. Hay muchas formas de combinar las llamadas al sistema para lograr diferentes comportamientos, y el shell es un buen ejemplo de cómo se pueden combinar las llamadas al sistema para lograr un manejo de procesos bastante sofisticado.

2.5.1. Redirecciones de E/S

La mayoría de las operaciones de E/S de archivos en un sistema UNIX se pueden realizar utilizando solo cinco funciones: open, read, write, lseek y close. Luego se puede analizar el efecto de varios tamaños de buffer en las funciones de lectura y escritura. Estas funciones menudo se denominan E/S sin buffer, en contraste con las rutinas de E/S estándar. El término sin buffer significa que cada operación de lectura o escritura invoca una llamada al sistema en el kernel. Estas funciones de E/S sin buffer no forman parte de ISO C, pero sí son parte de POSIX.1 y la Especificación Única de UNIX. [12]

2.5.2. File Descriptors

Para el kernel, todos los archivos abiertos se identifican mediante descriptores de archivos. Un descriptor de archivo es un número entero no negativo. Cuando abrimos un archivo existente o creamos un archivo nuevo, el kernel devuelve un descriptor de archivo al proceso. Cuando queremos leer o escribir en un archivo, identificamos el archivo con el descriptor de archivo que fue devuelto por open o creat como argumento para las funciones read o write.

Por convención, las shells del sistema UNIX asocian el descriptor de archivo 0 con la entrada estándar de un proceso, el descriptor de archivo 1 con la salida estándar y el descriptor de archivo 2 con el error estándar.

Convenciones de los descriptores de archivo

Aunque sus valores están estandarizados por POSIX.1, los números 0, 1 y 2 deben ser reemplazados en aplicaciones compatibles con POSIX por las constantes simbólicas `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO` para mejorar la legibilidad. Estas constantes están definidas en el archivo de cabecera <unistd.h>.

2.5.3. Función `open()`

Un archivo se abre o se crea llamando a la función open:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Ambas devuelven: descriptor de archivo si todo está bien, -1 en caso de error.

1. El parámetro path es el nombre del archivo que se va a abrir o crear.
2. Esta función tiene una multitud de opciones, que se especifican mediante el argumento segundo argumento flag. Este argumento se forma combinando con OR:
   - **O_RDONLY**: Abrir solo para lectura.
   - **O_WRONLY**: Abrir solo para escritura.
   - **O_RDWR**: Abrir para lectura y escritura.

   La mayoría de las implementaciones definen `O_RDONLY` como 0, `O_WRONLY` como 1 y `O_RDWR` como 2.
   - **O_EXEC**: Abrir solo para ejecutar.
   - **O_SEARCH**: Abrir solo para buscar (aplica a directorios).

   De estos cinco, solo uno debe especificarse. Los siguientes flags son opcionales:
   - **O_APPEND**: Añadir al final del archivo en cada escritura.

[12] Advanced Programming in the UNIX Environment - Cap.3

- **O_CLOEXEC**: Configurar la bandera FD_CLOEXEC del descriptor de archivo.
- **O_CREAT**: Crear el archivo si no existe. Esta opción requiere un tercer argumento en la función open (un cuarto argumento en la función openat), el mode, que especifica los bits de permisos de acceso del nuevo archivo.
- **O_DIRECTORY**: Generar un error si el path no refiere a un directorio.
- **O_EXCL**: Generar un error si también se especifica O_CREAT y el archivo ya existe. Esta verificación de si el archivo ya existe y la creación del archivo si no existe es una operación atómica.
- **O_NOCTTY**: Si el path refiere a un dispositivo terminal, no asignar el dispositivo como terminal de control para este proceso.
- **O_NOFOLLOW**: Generar un error si el path refiere a un enlace simbólico.
- **O_NONBLOCK**: Si el path refiere a un FIFO, archivo especial de bloque o archivo especial de carácter, esta opción establece el modo no bloqueante para la apertura del archivo y la E/S posterior.

[13]

2.5.4. Función `close()`

La función close cierra un descriptor de archivo:

```
int close(int fd);
```

Devuelve 0 si todo está bien, -1 en caso de error. Se encarga de cerrar un archivo también libera cualquier bloqueo de registro que el proceso pueda tener sobre el archivo.

2.5.5. Función `read()`

La función read lee datos de un archivo abierto:

```
ssize_t read(int fd, void *buf, size_t count);
```

Devuelve el número de bytes leídos, 0 si se alcanza el final del archivo, -1 en caso de error.
Los argumentos que toman hacen referencia a:

- **fd**: Descriptor de archivo.
- **buf**: Buffer donde se almacenarán los datos leídos.
- **count**: Número de bytes a leer.

La salida de error se encargará del mal manejo de los argumentos, es decir si se le pasa un descriptor cerrado, un buf inválido o un count negativo, en esos casos simplemente devolverá -1.
Si la función recibe mas bytes que los disponibles, la función devolverá el número de bytes disponibles. Si se llega al final del archivo, la función en la siguiente llamada devolverá 0.

2.5.6. Función `write()`

La función write escribe datos en un archivo abierto:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Devuelve el número de bytes escritos, -1 en caso de error. Es decir, generalmente el valor de retorno es igual al argumento count, pero si se produce un error, el valor de retorno es -1. Una causa normal de error es que el disco esté lleno.

2.6. File Sharing

Un sistema UNIX soporta la compartición de archivos entre procesos. Para poder entender mas este suceso, es necesario examinar algunas estructuras del kernel asociadas a los archivos.

1. Cada proceso tiene una **tabla de descriptores de archivos** (file descriptor table) que contiene lo siguiente:
   - Los flags de archivo,
   - Un puntero a la entrada de la tabla de archivos del archivo en el kernel.

[13] Advanced Programming in the UNIX Environment - Cap.3

2. El kernel mantiene una **tabla de archivos** (file table) que contiene lo siguiente:
   - Flags de estado del archivo,
   - Offset de archivo (la posición actual del archivo),
   - Puntero a la estructura de v-node del archivo.
3. Una estructura de **v-node** (v-node structure) : es parte de la abstracción que permite a los sistemas operativos manejar múltiples tipos de sistemas de archivos, contiene lo siguiente:
   - Información vnode,
   - Información inode, almacena metadatos del archivo (por ejemplo, tamaño, propietario, permisos, etc.).

Visualmente podemos ver a un proceso con dos archivos abiertos en la siguiente figura:

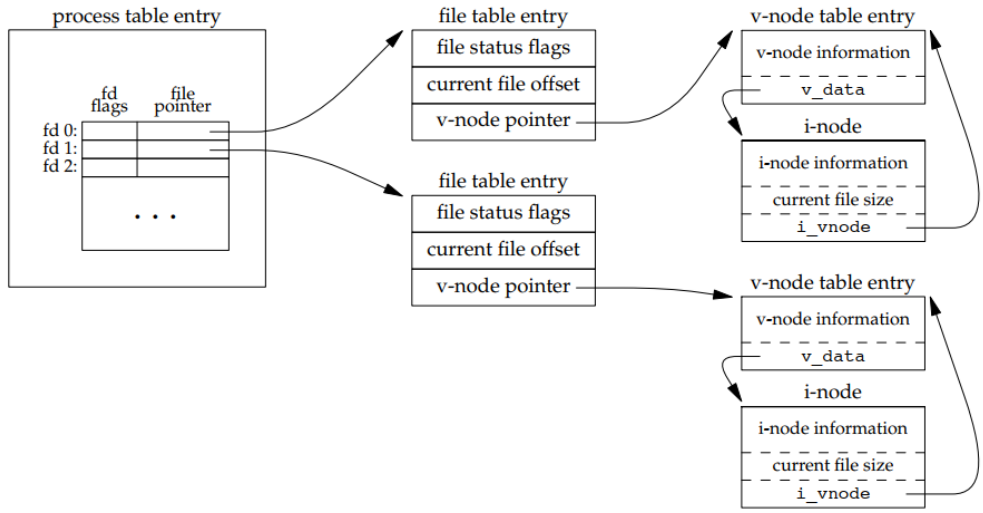

Figura 3: Files in processes

Ahora bien, si dos procesos comparten el mismo archivo no quiere decir que compartan la tabla de archivos. Una razón por la que cada proceso tiene su propia entrada en la tabla de archivos es para que cada uno tenga su propio desplazamiento actual en el archivo.

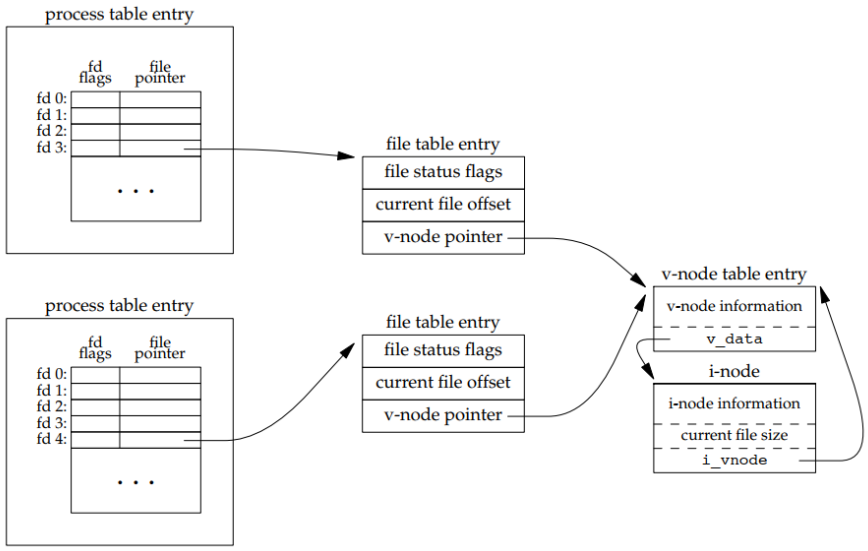

Figura 4: File Sharing

Pero es necesario también tener en cuenta como se maneja el offset con las funciones de E/S:

- Después de que cada escritura se completa mediante la función `write()`, el offset actual del archivo en la entrada de la tabla de archivos se incrementa en el número de bytes escritos. Si esto hace que el desplazamiento actual del archivo exceda el tamaño actual del archivo, el tamaño del archivo en la entrada de la tabla de i-nodes se ajusta al desplazamiento actual del archivo.
- Si un archivo se abre con la flag `O_APPEND`, se establece una flag correspondiente en los flags de estado del archivo en la entrada de la tabla de archivos. Cada vez que se realiza una escritura en un archivo con esta flag de append activada, el desplazamiento actual del archivo en la tabla de archivos se establece primero en el tamaño actual del archivo, obtenido de la tabla de i-nodes. Esto fuerza que cada escritura se agregue al final actual del archivo.
- Si un archivo se posiciona al final actual del archivo usando lseek, todo lo que sucede es que el desplazamiento actual del archivo en la tabla de archivos se ajusta al tamaño actual del archivo, obtenido de la entrada de la tabla de i-nodes.

Comportamiento con fork, dup

Es posible que mas de un file descriptor apunte a la misma entrada de la tabla de archivos:

- Por ejemplo, si un proceso llama a `fork()` después de abrir un archivo, el proceso hijo hereda la tabla de descriptores de archivos del proceso padre, incluido el descriptor de archivo abierto. Si el proceso hijo o el proceso padre cierran el descriptor de archivo, el archivo se cierra para ambos procesos. Si el proceso hijo o el proceso padre cambian el desplazamiento actual del archivo, el cambio se refleja en ambos procesos.
- Cuando se llama a dup o dup2, se crea un nuevo descriptor de archivo que apunta a la misma entrada de la tabla de archivos que el descriptor de archivo original. Si se cierra uno de los descriptores de archivo, el archivo permanece abierto hasta que se cierre el otro descriptor de archivo.

2.6.1. Funciones `dup` y `dup2`

Un file descriptor abierto puede ser duplicado con la función `dup` o `dup2`:

```
int dup(int oldfd);                    int dup2(int oldfd, int newfd);
```

El `file descriptor` devuelto por la función `dup` es el descriptor de archivo más bajo que no está en uso. En cambio con `dup2()` especificamos el valor del nuevo descriptor con el argumento `newfd`. Acá hay algunas decisiones que se deben tener en cuenta:

1. Si `newfd` es un descriptor de archivo abierto, primero se cierra.
2. Si `newfd` es igual a `oldfd`, `dup2()` devuelve `newfd` sin cerrar nada.
3. En caso contrario, el flag del descriptor de archivo `FD_CLOEXEC` se borra en el nuevo descriptor de archivo.

Por ejemplo, supongamos que se ejecuta la línea `newfd = dup(1)`, como el sistema tiene abiertos los descriptores 0, 1 y 2, el nuevo descriptor devuelto será 3. Y ambos apuntarán a la misma entrada de la tabla de archivos, compartiendo sus flags y offset.

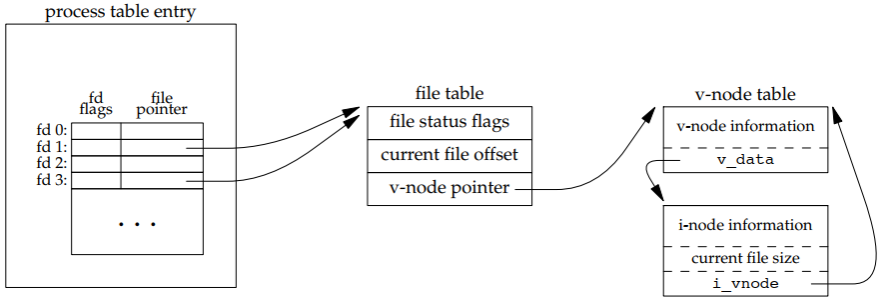

Figura 5: Función `dup`

Observaciones

El sistema operativo cuando libera un file descriptor, lo deja libre para su uso, es decir si por ejemplo cerramos la entrada estándar (descriptor 0), el próximo file descriptor que se abra será el 0. Por otro lado, si se llama a `dup()` o `dup2()` se crea un nuevo file descriptor que apunta a la misma entrada de la tabla de archivos que el descriptor de archivo original. Si se cierra uno de los descriptores de archivo, el archivo permanece abierto hasta que se cierre el otro descriptor de archivo.

3. Ejecución directa limitada

Como ya habia mencionado al principio, un sistema operativo debe ser capaz de ejecutar múltiples programas a la vez. Pero, ¿cómo se logra esto? una de las técnicas más simples para lograr esto es la **ejecución directa limitada**. En este modelo, el sistema operativo simplemente alterna la ejecución de los programas, permitiendo que cada uno se ejecute durante un tiempo limitado antes de pasar al siguiente.

¿A que llamamos ejecución directa? La parte de "ejecución directa" de la idea es simple: simplemente ejecuta el programa directamente en la CPU. Así, cuando el sistema operativo desea iniciar la ejecución de un programa, crea una entrada para él en la lista de procesos, asigna algo de memoria, carga el código del programa en la memoria (desde el disco), localiza su punto de entrada, salta a él y comienza a ejecutar el código del usuario.

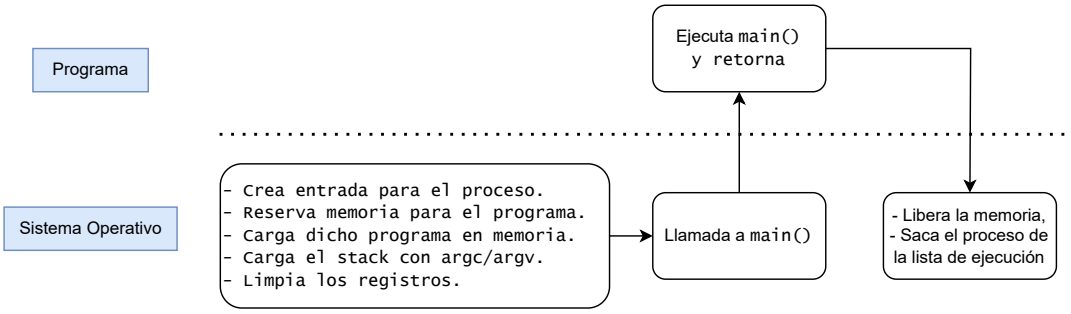

Figura 6: Ejecución directa limitada

[14]

Este enfoque da lugar a algunos problemas en nuestra búsqueda de *virtualizar la CPU*. El primero es sencillo: si simplemente ejecutamos un programa, **¿cómo puede el sistema operativo asegurarse de que el programa no haga nada que no queramos que haga, y al mismo tiempo lo ejecute de manera eficiente?** El segundo: cuando estamos ejecutando un proceso, **¿cómo detiene el sistema operativo su ejecución y cambia a otro proceso, implementando así el reparto de tiempo que necesitamos para virtualizar la CPU?**

3.1. Problema 1 - Operaciones privilegiadas

El primer problema es que el sistema operativo no quiere que los programas de usuario hagan cualquier cosa que quieran. Por ejemplo, no queremos que un programa de usuario pueda leer o escribir en cualquier parte de la memoria, o que pueda leer o escribir en cualquier archivo en el disco. En cambio, queremos que los programas de usuario se ejecuten en un modo restringido, donde solo pueden hacer cosas que el sistema operativo les permite hacer. A este modo restringido se le llama **modo usuario**. Por otro lado, el sistema operativo se ejecuta en un modo más privilegiado, llamado **modo kernel**. En este modo, el sistema operativo puede hacer cualquier cosa que desee, como leer y escribir en cualquier parte de la memoria, leer y escribir en cualquier archivo en el disco, y más.

> En resumen, el enfoque que tomamos es introducir un nuevo modo del procesador, conocido como *modo usuario*; el código que se ejecuta en modo usuario tiene restricciones sobre lo que puede hacer. Por otro lado, está el *modo kernel*, en el cual se ejecuta el sistema operativo. En este modo, el código que se ejecuta puede hacer lo que quiera, incluyendo operaciones privilegiadas como emitir solicitudes de E/S y ejecutar todo tipo de instrucciones restringidas.

Sin embargo, aún nos queda una incógnita ¿qué debe hacer un proceso de usuario cuando desea realizar algún tipo de operación privilegiada, como leer desde el disco? Para habilitar esto, prácticamente todo el hardware moderno proporciona la capacidad para que los programas de usuario realicen una llamada al sistema.

3.1.1. Ejecución de una `syscall`

Para ejecutar una llamada al sistema, un programa debe ejecutar una instrucción especial de trampa (trap). Esta instrucción simultáneamente salta al kernel y eleva el nivel de privilegio al modo kernel; una vez en el kernel, el sistema puede realizar las operaciones privilegiadas necesarias (si están permitidas) y, por lo tanto, hacer el trabajo requerido para el proceso que llamó. Cuando termina, el sistema operativo ejecuta una instrucción especial de retorno de trampa, que regresa al programa de usuario que llamó mientras simultáneamente reduce el nivel de privilegio de vuelta al modo usuario.

[14] OSTEP, Cap.6, Ejecución Directa Limitada

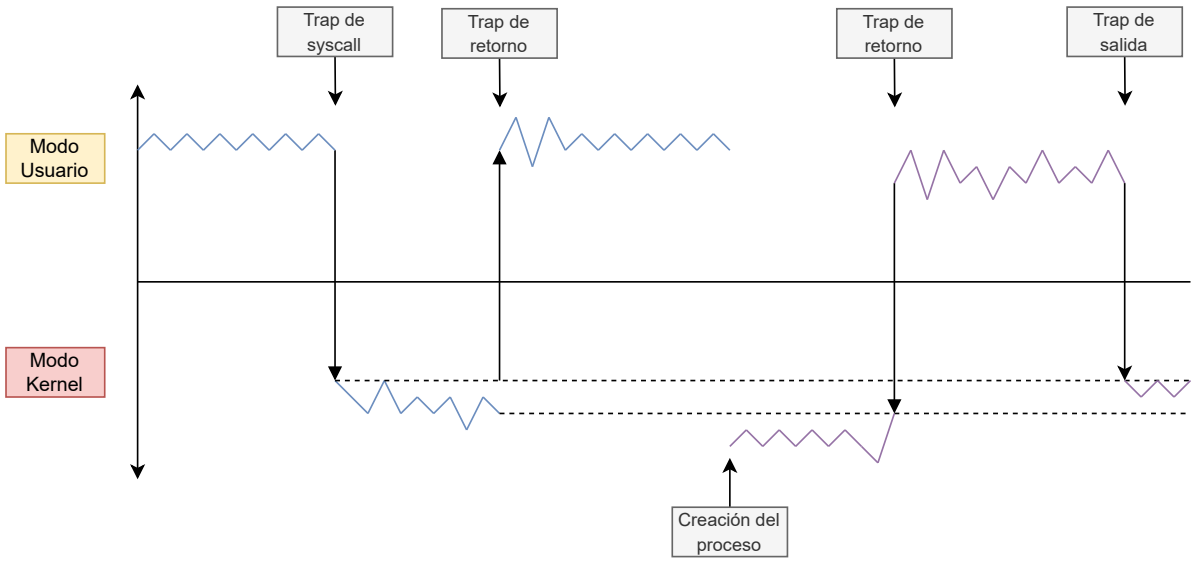

Figura 7: Ejecución de una `syscall`

[15]

Hay un detalle importante que se dejó fuera de esta discusión: ¿cómo sabe la trampa qué código ejecutar dentro del sistema operativo? Claramente, el proceso que llama no puede especificar una dirección a la cual saltar (como lo harías al hacer una llamada a un procedimiento); hacerlo permitiría que los programas saltaran a cualquier parte del kernel, lo cual claramente es una "Muy Mala Idea". Por lo tanto, el kernel debe controlar cuidadosamente qué código se ejecuta al ocurrir una trampa.

Una buena idea es agregar un nivel mas de indirección, las llamadas **trap tables**, cuando la máquina se inicia, lo hace en modo kernel y, por lo tanto, es libre de configurar el hardware de la máquina según sea necesario. Una de las primeras cosas que hace el sistema operativo es decirle al hardware qué código ejecutar cuando ocurren ciertos eventos excepcionales. Por ejemplo cuando ocurre una trampa, el sistema operativo configura la trampa para saltar a una dirección específica en la memoria, donde reside el código del kernel que maneja esa trampa en particular. De esta manera, cuando ocurre una trampa, el hardware salta a la dirección de la trampa, y el kernel se ejecuta en modo kernel, listo para manejar la trampa.

La trap table, la define el sistema operativo, pero la usa el hardware. El sistema operativo informa al hardware de las ubicaciones de estos **trap handlers**, generalmente con algún tipo de instrucción especial. Una vez que el hardware está informado, recuerda la ubicación de estos manejadores hasta que la máquina se reinicia nuevamente, y así el hardware sabe qué hacer (es decir, a qué código saltar) cuando se producen llamadas al sistema y otros eventos excepcionales.

3.1.2. Fases de ejecución directa limitada

En resumen, la ejecución directa limitada se puede dividir en las siguientes fases:

- **Booteo:** Se inicializa la **trap table** y se configura el hardware para que se ejecute en modo kernel.
- **Ejecución:** Aca se crea la entrada a la lista de procesos, junto con la asignación de memoria y la carga del código del programa en la memoria. Luego se ejecuta el código del usuario en modo usuario.

Esto es muy por encima de lo que realmente sucede, pero es un buen punto de partida para comprender cómo se ejecutan los programas en un sistema operativo moderno. En el siguiente gráfico se puede ver con mas profundidad el efecto:

[15] OSTEP, Cap.6, Ejecución Directa Limitada

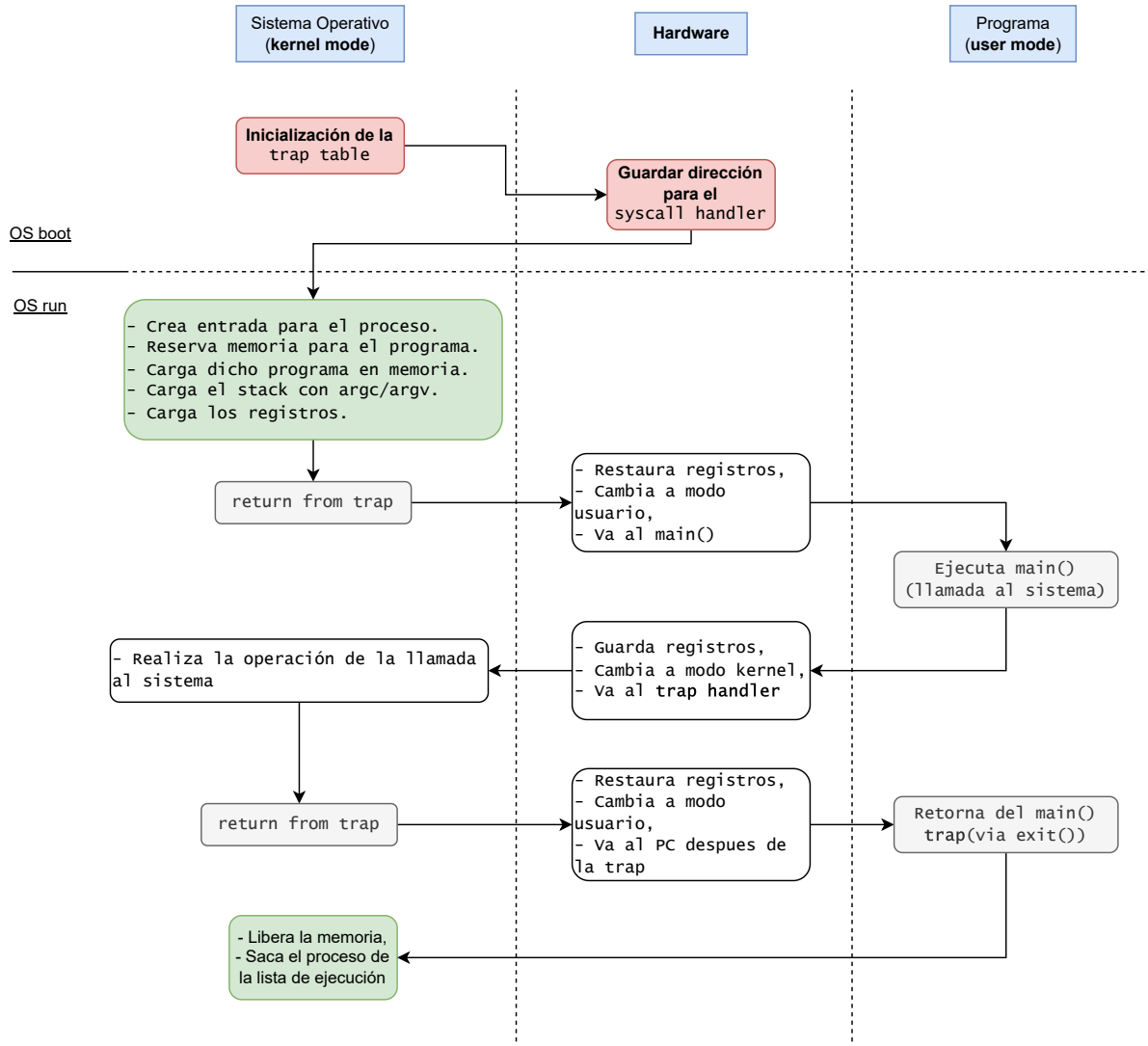

Figura 8: Ejecución directa limitada

[16]

Los puntos mas importantes de la ejecución directa limitada son:

- **Inicialización trap table**: Durante el arranque del sistema operativo, se inicializa la "trap table". Esta tabla contiene direcciones de manejadores de interrupciones, incluyendo el manejador de llamadas al sistema (syscall handler). Estos manejadores son funciones en el kernel que son llamadas cuando se produce una interrupción o una llamada al sistema.
- **Crear entrada para el proceso**: Se crea una entrada para el proceso en la lista de procesos del sistema operativo.
- **Carga y asignación de memoria**: Se asigna memoria para el proceso y se carga el código del programa en la memoria.
- **Carga el stack con argc y argv**: Se carga el stack con los argumentos del programa.
- **Limpia los registros:** Se llena la pila del kernel con los valores iniciales de los registros junto con la dirección de retorno (PC).
- **Return from trap**: Se ejecuta la instrucción de retorno de trampa, que cambia el modo de ejecución de kernel a usuario.
- **Ejecución del programa**: Se ejecuta el código del programa en modo usuario.

[16] OSTEP, Cap.6, Ejecución Directa Limitada

- **Handle trap:** cuando el kernel recibe la **trap** sucede lo siguiente: se guardan los registros del proceso, se cambia el modo de ejecución a kernel, se ejecuta el manejador de la trampa, se restauran los registros del proceso y se ejecuta la instrucción de retorno de trampa.
- **Terminar la ejecución:** El retorno del `main()` señala el fin de la ejecución del programa, luego se realiza una llamada `exit()` que genera una trampa y se ejecuta el manejador de la trampa; se libera la memoria asignada al proceso y se elimina la entrada del proceso de la lista de procesos.

3.2. Problema 2 - Cambio de procesos

El siguiente problema con la ejecucion directa es lograr cambiar de proceso. Un enfoque que algunos sistemas han adoptado en el pasado se conoce como el **enfoque cooperativo**. En este estilo, el sistema operativo confía en que los procesos del sistema se comporten razonablemente. Se asume que los procesos que se ejecutan durante mucho tiempo cederán periódicamente la CPU para que el sistema operativo pueda decidir ejecutar otra tarea. [17]

> En resumen, el enfoque cooperativo es simple: los procesos se ejecutan hasta que deciden ceder la CPU. En ese momento, el sistema operativo puede decidir ejecutar otro proceso. Este enfoque es simple y fácil de implementar, pero tiene un problema fundamental: si un proceso no cede la CPU, el sistema operativo no puede hacer nada al respecto.

Resulta que la mayoría de los procesos transfieren el control de la CPU al sistema operativo con bastante frecuencia al hacer llamadas al sistema. Los sistemas como este a menudo incluyen una llamada al sistema explícita de `yield`, que no hace nada excepto transferir el control al sistema operativo para que pueda ejecutar otros procesos. Las aplicaciones también transfieren el control al sistema operativo cuando hacen algo ilegal. Por ejemplo, si una aplicación divide por cero, o intenta acceder a una memoria a la que no debería poder acceder, generará una trampa al sistema operativo. El sistema operativo entonces recuperará el control de la CPU (y probablemente terminará el proceso infractor). [18]

Esta aproximación demanda de ayuda del hardware, timer interrupts, que son interrupciones que se generan a intervalos regulares. Cuando se produce un timer interrupt, el sistema operativo se despierta y puede decidir si desea cambiar de proceso. Si el sistema operativo decide que es hora de cambiar de proceso, puede hacerlo, y el proceso que estaba en ejecución se detiene y se guarda su estado. Luego, el sistema operativo puede elegir otro proceso para ejecutar, restaurar su estado y comenzar a ejecutarlo.

Una vez se gestiona la interrupción, el **scheduler** del sistema operativo decide qué proceso se ejecutará a continuación. El scheduler es un componente clave del sistema operativo, y su trabajo es decidir qué proceso se ejecutará a continuación. Hay muchos algoritmos de planificación diferentes que se pueden usar para decidir qué proceso se ejecutará a continuación, y cada uno tiene sus propias ventajas y desventajas.

3.2.1. Protocolo con timer interrupts

Solución

Podríamos plantearnos la pregunta de ¿qué sucede cuando un proceso se vuelve malicioso y niega el control del recurso? (es lo que se plantea en los párrafos anteriores), podriamos llamar a los `timer interrupts` como una función de seguridad integrada en el hardware, que permite al sistema operativo recuperar el control del mismo a intervalos regulares de tiempo, independientemente de si los procesos que utilizan el recurso completaron sus tareas o no. [a]

[a] CPU Virtualization via Limited Direct Execution - Shivam Mohan

Principalmente, el protocolo de cambio de procesos con timer interrupts se puede describir de la siguiente manera:

1. Inicializa la **trap table** y configura el hardware para que se ejecute en modo kernel.
2. Recordamos las direcciones de los manejadores de interrupciones, incluyendo el manejador de llamadas al sistema.
3. Inicializamos el timer interrupt y configuramos el hardware para que genere interrupciones a intervalos regulares.

[17] OSTEP, Cap.6, Ejecución Directa Limitada
[18] OSTEP 06 - Clase John Ordoñes

4. Luego de que transcurra un intervalo de tiempo, se genera un timer interrupt de CPU.

Luego saltamos a un escenario en donde ya hay un proceso en ejecución, el **proceso A**.

5. Se genera una interrupción de temporizador.
6. Guarda los registros del proceso A en la pila del kernel asociado al proceso A, luego nos movemos al modo kernel y nos movemos al manejador de trampas.
7. Ahora bien, guardamos los registros del proceso A en una tabla de procesos y tomamos de la tabla de procesos los registros del proceso B y los cargamos en los registros del procesador.
8. Nos vamos a la pila del kernel asociada al proceso B y salimos de la trampa.
9. Ahora se restauran los registros del proceso B, nos movemos al *program counter* del proceso B y se ejecuta el proceso B desde el modo usuario.

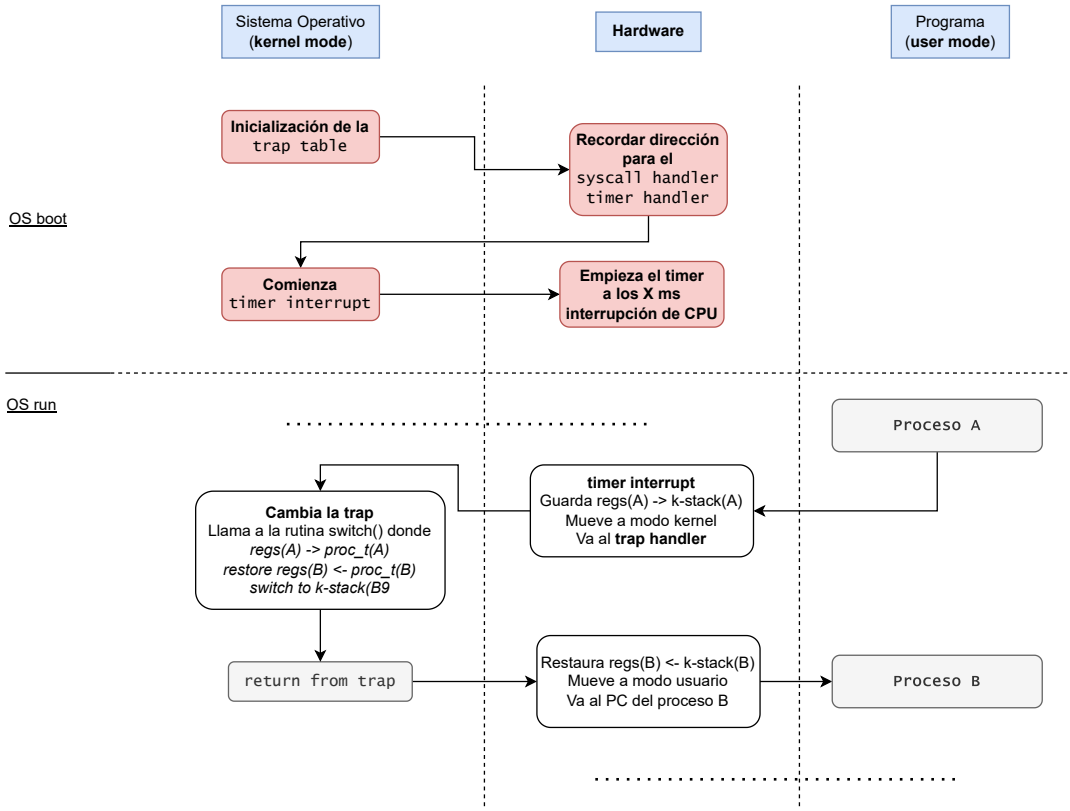

Figura 9: Switch de procesos

[19]

Ahora bien, ¿qué sucede si el proceso estaba en medio de una tarea y se produjo una interrupción y el proceso se vió obligado a liberar recursos? Acá entra el switch de procesos, el **cambio de contexto**, el cual tiene como idea principal es que cada vez que se le pide a un proceso que renuncie a un recurso que tiene y la tarea del proceso no se completa, el estado actual del proceso (es decir, su contexto) se almacena en la pila del kernel y cada vez que ese proceso recupera el acceso al recurso, el estado se restaura desde la pila del kernel y los procesos pueden continuar la ejecución desde el mismo estado exacto donde lo dejaron antes, de esta manera podemos asegurar que con el tiempo, los procesos puedan lograr un progreso real aunque no tengan acceso al recurso durante todo el tiempo.

[19] OSTEP, Cap.6, Ejecución Directa Limitada

4. Planificación

Las preguntas que surgieron en la sección anterior son: ¿cómo decide el sistema operativo qué proceso se ejecutará a continuación? y con esto llegamos a la **planificación de procesos**. La planificación de procesos es una de las tareas más importantes de un sistema operativo. El trabajo del planificador es decidir qué proceso se ejecutará a continuación. Hay muchos algoritmos de planificación diferentes que se pueden usar para decidir qué proceso se ejecutará a continuación, y cada uno tiene sus propias ventajas y desventajas.

Definición

Un **algoritmo de planificación** es una estrategia que se utiliza para determinar el orden en que los procesos se ejecutan en un sistema operativo. Debe cumplir con varios objetivos conflictivos: tiempo de respuesta rápido para los procesos, buen rendimiento para trabajos en segundo plano, evitar la inanición de procesos, conciliar las necesidades de procesos de baja y alta prioridad, entre otros. El conjunto de reglas que se utilizan para determinar cuándo y cómo seleccionar un nuevo proceso para ejecutarse se llama **política de planificación**. [a]

[a] Understanding the Linux Kernel - Cap.7

Definición

El tiempo de entrega de un trabajo se define como el momento en que se completa el trabajo menos el momento en que el trabajo llegó al sistema.

$$T_{\text{entrega}} = T_{\text{finalización}} - T_{\text{llegada}} \tag{1}$$

Como asumimos que todos los trabajos llegan al mismo tiempo, entonces por el momento tenemos que $T_{\text{llegada}} = 0$ y por lo tanto $T_{\text{entrega}} = T_{\text{finalización}}$. Esto irá cambiando a medida que se profundice en el tema.

4.1. Primero en llegar, primer en salir (FIFO)

El algoritmo de planificación más simple es el **primero en llegar, primero en salir (FIFO)**. En este algoritmo, los procesos se ejecutan en el orden en que llegan al sistema. En otras palabras, el sistema operativo mantiene una cola de procesos, y cuando un proceso llega al sistema, se coloca al final de la cola. Cuando el sistema operativo necesita decidir qué proceso se ejecutará a continuación, simplemente toma el primer proceso de la cola y lo ejecuta.

Supongamos que llegan 3 procesos al sistema, A llegó justo un momento antes que B, que a su vez llegó un momento antes que C. Supongamos también que cada trabajo se ejecuta durante 10 segundos. [20]

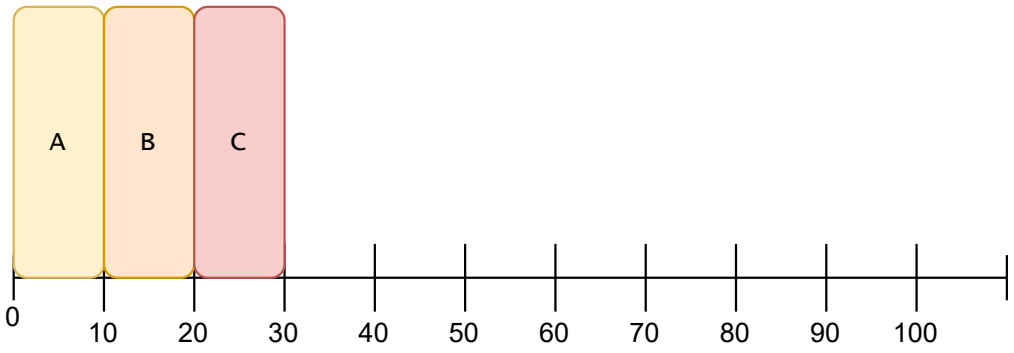

Figura 10: Primero en llegar, primero en salir (FIFO)

[20] OSTEP, Cap.7, Planificación de Procesos

¿Cuál será el tiempo medio de entrega de estos trabajos? se puede ver que A terminó en 10, B en 20 y C en 30. Por lo tanto, el tiempo medio de entrega para los tres trabajos es simplemente $\frac{10+20+30}{3} = 20$. Pero esto es porque estamos suponiendo que los trabajos se ejecutan la misma cantidad de tiempo.

Ahora supongamos que el trabajo A corre primero durante 100 segundos completos antes de que B o C tengan la oportunidad de ejecutarse. En este caso el tiempo medio de entrega para los tres trabajos es $\frac{100+110+120}{3} = 110$ que es relativamente alto.

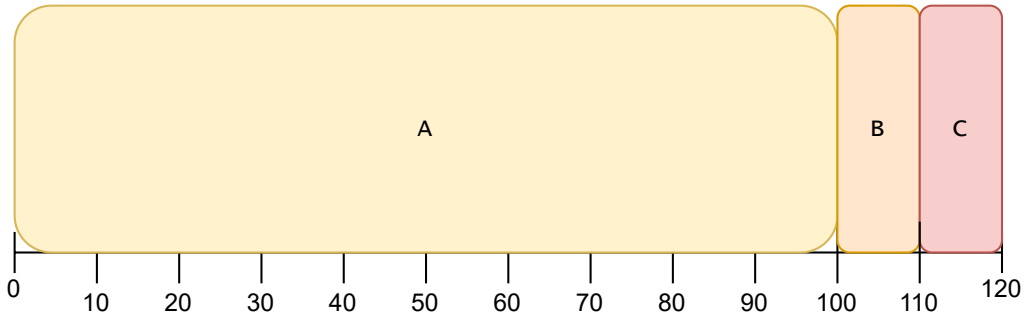

Figura 11: Primero en llegar, primero en salir (FIFO)

Este problema se conoce generalmente como el **efecto de convoy**, en el que una serie de consumidores potenciales de algún recurso, relativamente cortos, se ponen en cola detrás de un consumidor de gran peso. Puede que este escenario de planificación te suene a cuando hay una única fila en un supermercado y lo que sentís cuando ves a la persona en frente tuyo con tres carritos llenos de provisiones, y una chequera en mano; va a demorar un rato.

4.2. Trabajo mas corto primero (SJF)

El problema mencionado recien se puede solucionar con el algoritmo de planificación de **trabajo mas corto primero (SJF)**. En este algoritmo, el sistema operativo elige el proceso que se ejecutará a continuación en función de cuánto tiempo se espera que dure el proceso. En otras palabras, el sistema operativo mantiene una cola de procesos, y cuando un proceso llega al sistema, se coloca en la cola. Cuando el sistema operativo necesita decidir qué proceso se ejecutará a continuación, simplemente toma el proceso más corto de la cola y lo ejecuta. [21]

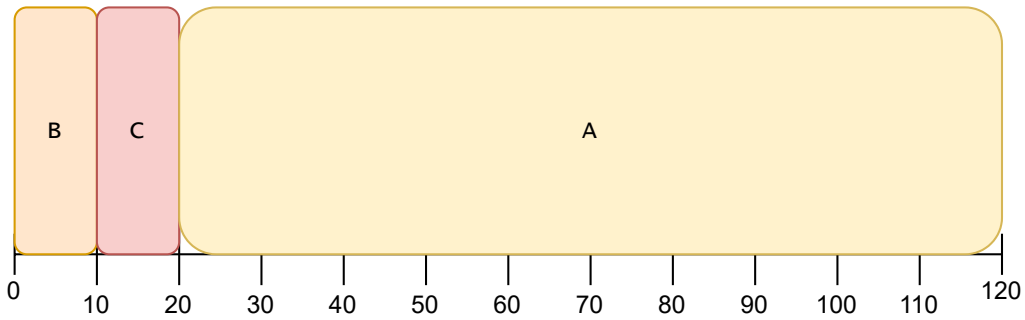

Figura 12: Trabajo mas corto primero (SJF)

Tomando el mismo ejemplo pero aplicando el algoritmo SJF, el diagrama debería aclarar por qué SJF tiene un rendimiento mucho mejor con respecto al tiempo medio de entrega. Simplemente ejecutando B y C antes que A, SJF reduce el tiempo medio de entrega de 110 segundos a 50 ($\frac{10+20+120}{3} = 50$).

[21] OSTEP, Cap.7, Planificación de Procesos

Ahora supongamos que los procesos no llegan al mismo tiempo: A llega en $t = 0$ y necesita ejecutarse durante 100 segundos, mientras que B y C llegan en $t = 10$ y cada uno necesita ejecutarse durante 10 segundos. Con el algoritmo SJF, el tiempo medio de entrega para los tres trabajos es $\frac{100+(110-10)+(120-10)}{3} = 103{,}33$.

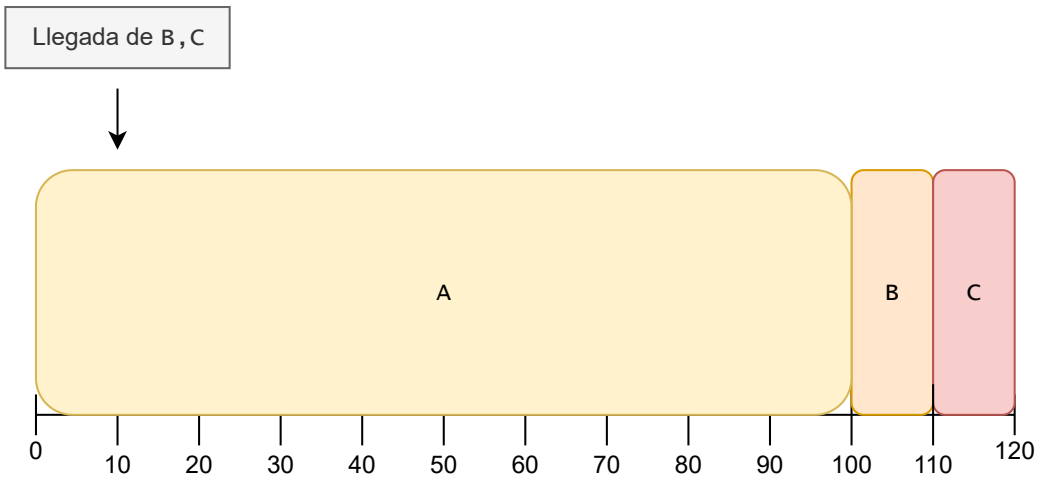

Figura 13: Trabajo mas corto primero (SJF)

4.3. Trabajo de Menor Tiempo Restante Primero (STCF)

Para solucionar el problema que surgió antes, llega un proceso largo y luego de un rato llegan muchos mas cortos, los cortos tienen un tiempo de espera muy largo cuando podrían ejecutarse antes, en este planificador esto no pasa, ya que cada vez que un nuevo trabajo ingresa al sistema, el planificador STCF determina a cuál de los trabajos restates (incluyendo el nuevo trabajo) le queda el menor tiempo hasta finalizar, y lo elige para ser ejecutado. Por lo tanto, en nuestro ejemplo, STCF habría detenido a A y ejecutado a B y a C hasta su finalización; y recién cuando hayan terminado, habría elegido ejecutar lo que quede de A.

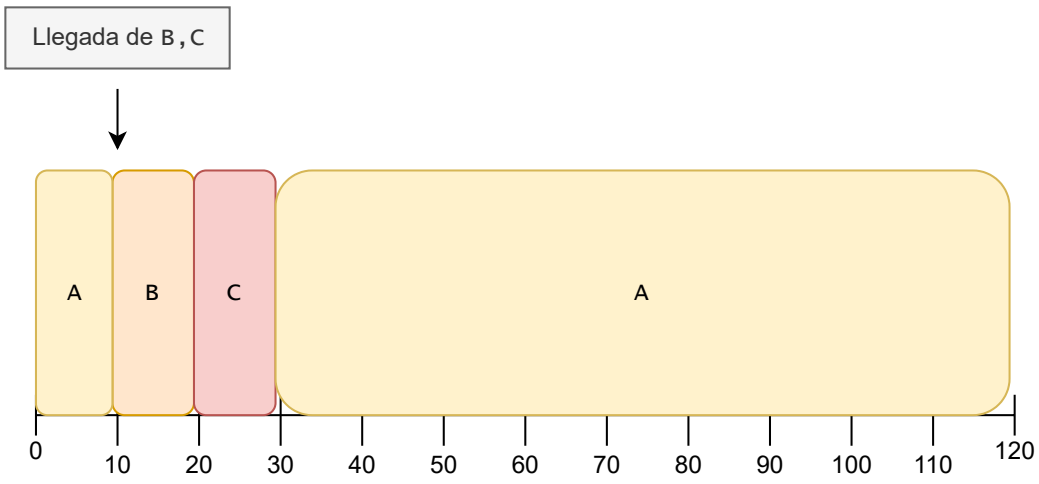

Figura 14: Trabajo de Menor Tiempo Restante Primero (STCF)

El resultado es un tiempo de entrega mucho mejor (50segs) que el obtenido con SJF (103.33segs).

4.4. Round Robin

Si hablamos de procesos interactivos, es decir aquellos que interactúan constantemente con sus usuarios y, por lo tanto, pasan mucho tiempo esperando pulsaciones de teclas y operaciones de ratón. Cuando se recibe

una entrada, el proceso debe despertarse rápidamente, o el usuario percibirá el sistema como poco receptivo. Se introduce un nuevo concepto de tiempo de respuesta:

Definición

El tiempo de respuesta se define como el tiempo desde que el trabajo llega a un sistema hasta la primera vez que es elegido para ser ejecutado:

$$T_{\text{respuesta}} = T_{\text{primera ejecución}} - T_{\text{llegada}}$$

Entonces, si tres trabajos llegan al mismo tiempo, por ejemplo, el tercer trabajo tiene que esperar a que los dos trabajos anteriores se ejecuten en su totalidad antes de ser elegido por primera vez. Si bien es excelente para el tiempo de entrega, este acercamiento es bastante malo para el tiempo de respuesta y para la interactividad. Es más, imaginate sentado en una terminal, escribiendo, y teniendo que esperar 10 segundos para ver una respuesta del sistema simplemente porque se eligió ejecutar otro trabajo antes que el tuyo. [22]

Acá entra el planificador **Round Robin**, que es sensible al tiempo de respuesta. La idea básica es simple: en lugar de ejecutar trabajos hasta su finalización, ejecuta cada trabajo durante un segmento de tiempo (a veces llamado quantum de planificación) y luego cambia al siguiente trabajo en la cola de ejecución. Esto lo hace repetidamente hasta que se terminan todos los trabajos. Por esta razón, RR se denomina a veces como división de tiempo. Notemos que la duración de un segmento de tiempo debe ser un múltiplo del período de interrupción del temporizador; por lo tanto, si el temporizador se interrumpe cada 10 milisegundos, el segmento de tiempo podría ser de 10, 20 o cualquier otro múltiplo de 10 ms.

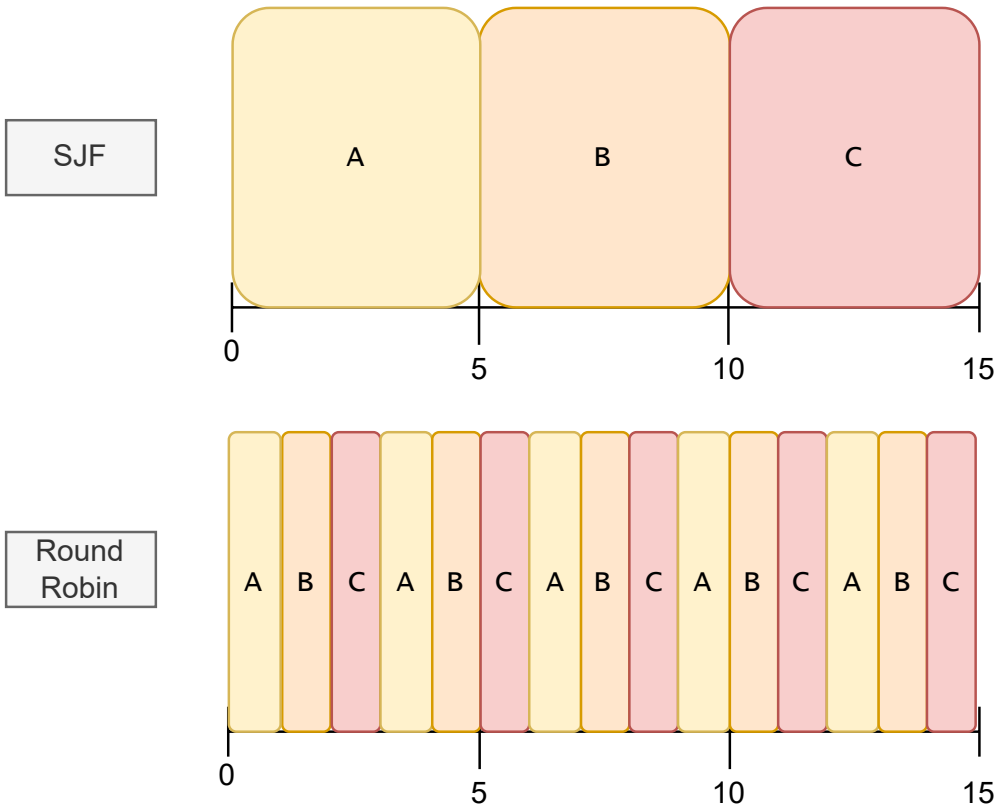

Figura 15: Round Robin

[22] OSTEP, Cap.6, Ejecución Directa Limitada

Observaciones

- Cuanto más corto sea el segmento de duración, mejor será el rendimiento de RR según la métrica del tiempo de respuesta. Sin embargo, si el segmento de tiempo es demasiado corto puede resultar problemático: de repente, el costo del cambio de contexto dominaría el rendimiento general. Por lo tanto, decidir la duración del segmento de tiempo presenta un intercambio que el diseñador del sistema debe estar dispuesto a hacer; tiene que ser suficientemente largo como para amortizar el costo del cambio, pero no tan largo como para que el sistema ya no responda.
- Generalmente, cualquier política (como RR) que sea justa, es decir, que divida uniformemente la CPU entre los procesos activos en una escala de tiempo pequeña, tendrá un desempeño deficiente en métricas como el tiempo de entrega. De hecho, este trato es inherente: si estás dispuesto a ser injusto, podés ejecutar los trabajos más cortos hasta su finalización, pero a costa del tiempo de respuesta; si, en cambio, valorás la justicia, el tiempo derespuesta se reduce, pero a costa del tiempo de entrega. Este tipo de intercambio es común en los sistemas; no se puede estar en la misa y en la procesión a la vez.

4.5. La Cola Multinivel con retroalimentación

Las ideas básicas detrás de la planificación, y desarrollado dos tipos de acercamientos distintos. El primero ejecuta el trabajo más corto restante y, por lo tanto, optimiza el tiempo de entrega; el segundo alterna entre todos los trabajos y optimiza así el tiempo de respuesta. Por desgracia, ambos son malos donde el otro es bueno, un intercambio inherente que es común en los sistemas.Pero no se resolvió el problema de la incapacidad fundamental del SO para ver el futuro. Ahora se va a produndizar en un planificador que *utiliza el pasado reciente* para predecir el futuro.

El problema fundamental que la MLFQ intenta abordar tiene dos partes. Primero, busca **optimizar el tiempo de entrega**, que se realiza ejecutando los trabajos más cortos primero; pero el sistema operativo generalmente *no sabe por cuánto tiempo se ejecutará un trabajo*, que es exactamente el conocimiento que requieren los algoritmos como SJF (o STCF). En segundo lugar, MLFQ quiere **hacer que el sistema se sienta receptivo a los usuarios interactivos**, y así minimizar el tiempo de respuesta; pero también hay una contra, algoritmos como Round Robin reducen el tiempo de respuesta, pero son terribles para el tiempo de entrega. Entonces, nuestro problema: dado que en general no sabemos nada sobre un proceso, ¿cómo podemos construir un planificador para lograr estos objetivos? ¿Cómo puede el planificador aprender, mientras el sistema se está ejecutando, las características de los trabajos que está corriendo, y así tomar mejores decisiones de planificación? [23]

La MLFQ tiene varias **colas** distintas, a cada una de las cuales se le asigna un nivel de prioridad diferente. En un momento dado, cada trabajo que está listo para ejecutarse se encuentra en una sola cola. MLFQ usa las prioridades para decidir qué trabajo debe ejecutarse en un momento determinado: se elige para ser ejecutado un trabajo con una mayor prioridad (es decir, un trabajo en una cola más alta). Por supuesto, puede haber más de un trabajo en una cola determinada y, por lo tanto, tener la misma prioridad. En este caso, simplemente usaremos planificación round-robin entre estos trabajos.

Con esto surge la pregunta de *¿cómo establece las prioridades?* En lugar de dar una prioridad fija a cada trabajo, MLFQ varía la prioridad de un trabajo en función de su comportamiento observado. Si, por ejemplo, un trabajo renuncia repetidamente a la CPU mientras espera la entrada del teclado, MLFQ mantendrá su prioridad alta, ya que así es como podría comportarse un proceso interactivo. Si, en cambio, un trabajo usa la CPU de manera intensiva durante largos períodos de tiempo, MLFQ reducirá su prioridad. De esta manera, MLFQ intentará aprender sobre los procesos a medida que se ejecutan y, por lo tanto, utilizará el historial del trabajo para predecir su comportamiento futuro.

4.5.1. Reglas de la MLFQ

El conjunto de reglas refinadas de la MLFQ son las siguientes:

1. Si Prioridad(A) > Prioridad(B), se ejecuta A.
2. Si Prioridad(A) = Prioridad(B), se ejecutan en round-robin.
3. Cuando un trabajo ingresa al sistema, se coloca en la prioridad más alta (la cola de más arriba).

[23] OSTEP, Cap.8, MLFQ

4. Una vez que un trabajo utilice su tiempo asignado en un nivel dado (independientemente de cuántas veces haya renunciado a la CPU), su prioridad se reduce (es decir, se mueve una cola hacia abajo).

5. Después de un período de tiempo determinado S, mover todos los trabajos del sistema a la cola más alta.

4.5.2. Ejemplos

En este ejemplo se tienen dos procesos, A en negro, y B en gris, A comienza ejecutandose, al pasar un segmento de tiempo baja y asi hasta llegar a la cola de menor prioridad donde se ejecuta por un tiempo prolongado. Pero en el tiempo $T = 100$ llega el proceso B, que comienza también en la cola de mayor prioridad pero con la diferencia de que como esté tiene un tiempo mas corto, al consumir dos segmentos de tiempo ya se terminó de ejecutar, por lo que no llega a estar en menor prioridad. Luego de terminar el proceso A sigue con normalidad.

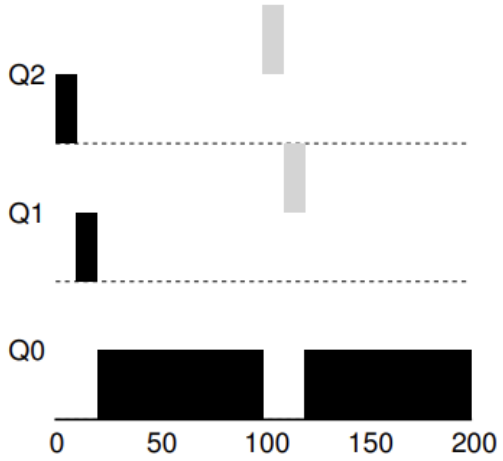

Figura 16: Cola Multinivel con retroalimentación - Ejemplo 1

Ahora supongamos que tenemos un proceso B interactivo, es decir con I/O incluido, sabemos que un proceso abandona la CPU cuando tiene muchas peticiones I/O porque las operaciones de entrada/salida tienden a ser más lentas que las operaciones de procesamiento que se realizan en la CPU. Cuando un proceso realiza una solicitud de I/O, como leer o escribir en un disco, comunicarse con una red o esperar una entrada del usuario, la CPU no puede continuar ejecutando ese proceso hasta que la operación de I/O se complete. Una consecuencia de la **regla 4**, es que si un proceso abandona el procesador antes de usar un segmento de tiempo, este se mantien en la misma prioridad. Acá podemos ver el objeto de esta regla: *si un trabajo interactivo, por ejemplo, está haciendo una gran cantidad de I/O, abandonará la CPU antes de que se complete su segmento de tiempo; en tal caso, no queremos penalizar el trabajo y, por lo tanto, simplemente lo mantenemos en el mismo nivel.*

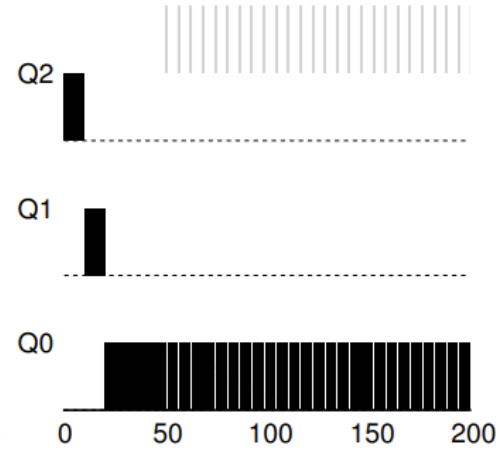

Figura 17: Cola Multinivel con retroalimentación - Ejemplo 2

Un ejemplo para comprender el objetivo de la **regla 5** es el siguiente, en la izquierda, no hay un impulso de prioridad y, por lo tanto el trabajo de larga duración se muere de hambre una vez que llegan los dos trabajos cortos; en la derecha, hay un impulso de prioridad cada 50 ms (que probablemente sea un valor demasiado pequeño, pero se utiliza aquí para el ejemplo) y, por lo tanto, al menos garantizamos que el trabajo de larga duración progresará, obteniendo un impulso a la máxima prioridad cada 50 ms y así llegar a ejecutarse periódicamente.

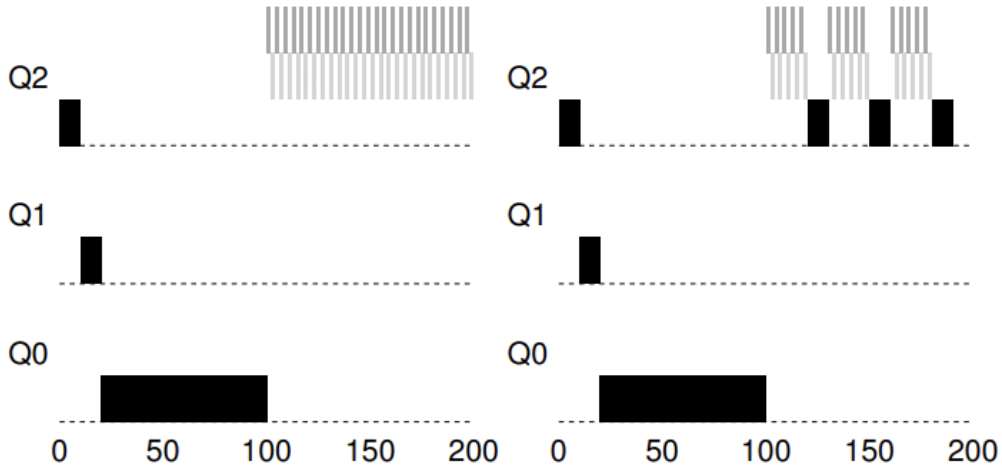

Figura 18: Cola Multinivel con retroalimentación - Ejemplo 3

Ahora bien, cuando un proceso intenta .[en]gañar.[a]l planificador, con peticiones I/O antes del segmento de tiempo para así mantenerse con mayor prioridad entra la **regla 4**, una vez que un proceso haya utilizado todo su tiempo asignado, es degradado a la siguiente cola de prioridad. No importa si utiliza el segmento de tiempo en una ráfaga larga o en muchas pequeñas. Así podemos ver en la figura del lado izquierdo no se implementaría esta regla, y el lado derecho si.

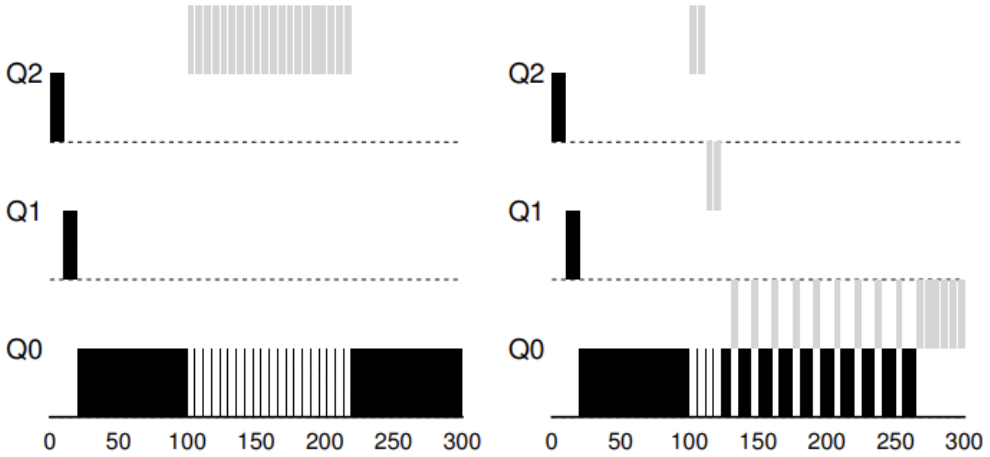

Figura 19: Cola Multinivel con retroalimentación - Ejemplo 4

5. Espacio de Direcciones

Llamamos espacio de direcciones a la vista de la memoria del programa en ejecución en el sistema. Este contiene todo el estado de la memoria del programa en ejecución:

- El código del programa.
- El programa mientras se ejecuta, utiliza una pila (**stack**) para llevar un registro de donde se encuentra en la cadena de llamadas a funciones, asi como para asignar variables locales y pasar parámetros y valores de retorno entre las rutinas.
- También tiene un montículo (**heap**) que se utiliza para almacenar datos dinámicos, como estructuras de datos que se asignan y liberan en tiempo de ejecución. Lo que se obtiene mediante llamadas a `malloc()`.

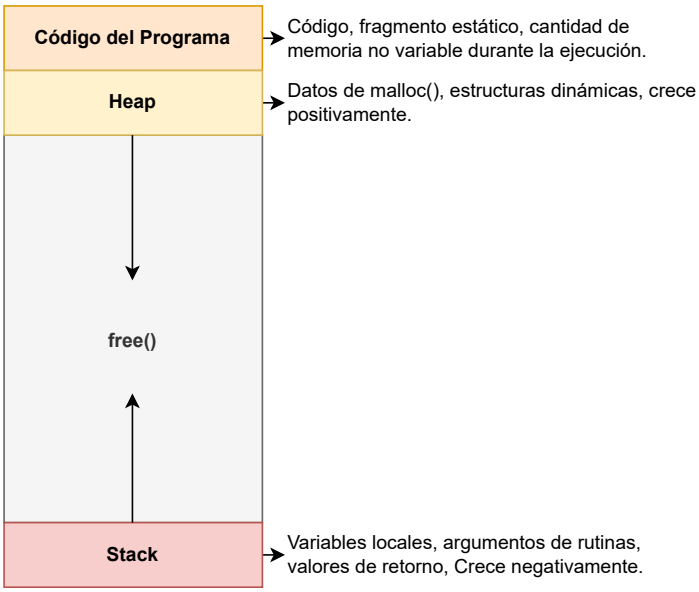

Figura 20: Stack y Heap

El concepto de **virtualización de memoria** está directamente relacionado con la estructura del espacio de direcciones que describes, ya que esta estructura es una abstracción que el sistema operativo proporciona al programa en ejecución, permitiéndole tener una vista unificada y continua de la memoria, a pesar de que los datos puedan estar físicamente dispersos o no corresponder a los mismos valores de direcciones.

- **Espacio de direcciones virtual**: El código, la pila (stack) y el montículo (heap) existen en el espacio de direcciones virtual, que es lo que el programa percibe como su propio entorno de memoria. Sin embargo, este espacio de direcciones no corresponde directamente a ubicaciones en la memoria física. El sistema operativo y el hardware (a través de la unidad de gestión de memoria o MMU, Memory Management Unit) se encargan de mapear estas direcciones virtuales a direcciones físicas reales.
- **Aislamiento y seguridad**: La virtualización de la memoria permite que múltiples procesos tengan su propio espacio de direcciones virtual, lo que asegura que cada proceso tenga una vista aislada de la memoria, sin interferir con los demás. Por ejemplo, aunque dos procesos distintos tengan una pila o un montículo ubicados en la misma dirección virtual (digamos, ambos comienzan su pila en la dirección 0x1000), sus datos se almacenan en diferentes lugares de la memoria física.
- **Optimización del uso de memoria**: La pila y el montículo en un espacio de direcciones pueden expandirse y contraerse según sea necesario, y el sistema operativo asigna más memoria física o la libera de acuerdo a estas solicitudes. El uso de virtualización de memoria permite que este crecimiento dinámico no esté limitado por la fragmentación o disponibilidad contigua de memoria física, ya que las direcciones virtuales pueden mapearse a bloques de memoria física discontiguos.
- **Manejo de la memoria**: El uso de llamadas como malloc() para asignar memoria en el montículo (heap) o el uso de la pila (stack) para la gestión de llamadas a funciones es completamente manejado en el espacio de direcciones virtual. El programa "piensa"que está operando en direcciones de memoria específicas, pero el sistema operativo se encarga de traducir esas direcciones virtuales en direcciones físicas reales.

5.1. Virtualización de memoria

Para asegurarse de que el sistema operativo pueda asignar y liberar memoria de manera eficiente, se utiliza la virtualización de memoria. La virtualización de memoria es un mecanismo que permite que un programa en ejecución tenga una vista unificada y continua de la memoria, a pesar de que los datos pueden estar físicamente dispersos o no corresponder a los mismos valores de direcciones. [24] Hay que tener en cuenta ciertos objetivos:

1. Un objetivo principal de un sistema de memoria virtual (VM) es la **transparencia**. El sistema operativo debe implementar la memoria virtual de una manera que sea invisible para el programa en ejecución. Es decir, el programa no debe ser consciente del hecho de que la memoria está siendo virtualizada; más bien, el programa debe comportarse como si tuviera su propia memoria física privada. Detrás de escena, el sistema operativo (y el hardware) realiza todo el trabajo para multiplexar la memoria entre muchos trabajos diferentes, creando así la ilusión de que cada programa tiene su propia memoria.

2. Otro objetivo de la memoria virtual es la **eficiencia**. El sistema operativo debe esforzarse por hacer que la virtualización sea lo más eficiente posible, tanto en términos de tiempo (es decir, sin hacer que los programas se ejecuten mucho más lentamente) como de espacio (es decir, sin usar demasiada memoria para las estructuras necesarias para soportar la virtualización). Para implementar una virtualización eficiente en tiempo, el sistema operativo tendrá que depender del soporte de hardware, incluyendo características de hardware como las TLB (de las que aprenderemos a su debido tiempo).

3. Un tercer objetivo de la memoria virtual es la **protección**. El sistema operativo debe asegurarse de proteger los procesos entre sí, así como de protegerse a sí mismo de los procesos. Cuando un proceso realiza una operación de carga, almacenamiento o ejecución de una instrucción, no debe poder acceder ni afectar de ninguna manera el contenido de memoria de ningún otro proceso ni del propio sistema operativo (es decir, nada fuera de su espacio de direcciones). La protección, por lo tanto, nos permite ofrecer la propiedad de aislamiento entre procesos; cada proceso debe ejecutarse en su propio entorno aislado, protegido de los fallos o incluso de los procesos maliciosos.

[24] OSTEP, Cap.9, Virtualización de Memoria

6. API de Memoria

6.1. Tipos de memoria

1. Memoria de la pila **(Stack memory)**: todas sus asignaciones y liberaciones se gestionan automáticamente. Para declarar algo en la pila en C, simplemente se declara una variable.

```
void foo() {
    int x = 42;
}
```

   En este caso, la variable `x` se asigna en la pila y se libera automáticamente cuando la función `foo()` termina de ejecutarse. Podría decirse que entonces si queremos un valor que permanezca mas allá de nuestra función, aca entra en juego el **heap**.

2. Memoria del montículo **(Heap memory)**: es un espacio de memoria más grande y más flexible que se puede asignar y liberar manualmente. En C, se utiliza la función `malloc()` para asignar memoria en el montículo y `free()` para liberarla.

```
void foo() {
    int *x = (int *)malloc(sizeof(int));
    ..
}
```

   En este caso, la variable `x` se asigna en el montículo y no se libera automáticamente cuando la función `foo()` termina de ejecutarse. Por lo tanto, es responsabilidad del programador liberar la memoria asignada en el montículo cuando ya no se necesite.

6.2. Llamadas `malloc()` y `free()`

1. La llamada `malloc()` es simplemente decirle que tamaño del espacio que estas solicitando del **heap** y te devuelve un puntero a la dirección de memoria donde se encuentra ese espacio o `NULL` si no hay suficiente espacio.

```
void *malloc(size_t size);
```

   El único parámetro que toma `malloc()` es de tipo `size_t`, que simplemente describe cuántos bytes necesitas. Por ejemplo si se quiere reservar memoria para un punto flotante de 8 bytes, se puede hacer de la siguiente manera:

```
double *x = (double *)malloc(sizeof(double));
```

   ¿Cómo está implementada la llamada a `malloc()`? Voy a introducir una posible implementación sin la certeza de que realmente este hecho de esa forma.

   - `malloc.c` contiene la estructura `meta_block` que almacena información sobre el bloque de memoria.
   - Cada `meta_block` es un nodo en la lista doblemente enlazada. El encabezado de la lista doblemente enlazada se mantiene globalmente en la variable base. Y almacena información como: el **tamaño del bloque de memoria** correspondiente, **si el bloque de memoria correspondiente está libre** o no, la **dirección del nodo siguiente y anterior** en la lista doblemente enlazada, **puntero al inicio del bloque de memoria** utilizado para almacenar información, una variable de matriz de caracteres que no almacena ningún valor, pero se utiliza para obtener la dirección desde donde se pueden almacenar los datos.
   - Cuando se invoca, la función `find_suitable_block()` busca si existe un bloque de memoria libre con al menos la cantidad de espacio requerida que esté libre al recorrer la lista vinculada.
   - Cuando se invoca, la función `split_space()` divide el bloque de memoria en dos si contiene suficiente espacio que pueda usarse para almacenar datos.

- Cuando se invoca, la función `extend_heap()` extiende el montón y agrega un nuevo bloque de memoria a la lista vinculada.
- Cuando se llama a la función `malloc()`:
  a) Crea un nuevo bloque invocando `extend_heap()` si basees NULL.
  b) De lo contrario, itera a través de la lista enlazada para encontrar si hay algún bloque adecuado (bloque con el tamaño mínimo solicitado).
  c) Si se encuentra un bloque adecuado, se devuelve.
  d) De lo contrario, extiende el montón y devuelve el bloque recién creado.
  e) También verifica si el bloque encontrado en la lista enlazada tiene suficiente espacio extra para ser dividido. En este caso, `split_space()` se invoca .

  El código completo se puede encontrar en este enlace.
- La llamada `free()` simplemente libera el espacio de memoria asignado previamente por `malloc()`. Simplemente toma un argumento, el puntero a la memoria retornado por `malloc()`.

Errore comunes

- Siempre que se declara un puntero para tomar memoria en el montículo, si o sí debe reservarse la memoria antes de usarla.
- Si se va a guardar dinámicamente una cadena de caracteres, siempre hay que reservar un byte adicional para el carácter nulo que indica el final de la cadena. Es decir:

```
char *src = "hello";
char *dst = (char *)malloc(strlen(src) + 1);
strcpy(dst, src);
```

- Si no se reserva la cantidad de memoria correspondiente y se intenta escribir en un espacio de memoria no asignado, se producirá un error de segmentación.
- Siempre que se alloca memoria, se debe liberar la memoria una vez que ya no se necesite. Si no se libera la memoria, se produce una fuga de memoria.
- Otro error común es el **double free**, que ocurre cuando se intenta liberar la misma memoria dos veces.

7. El mecanismo de traducción de direcciones

Cuando virtualizabamos la CPU, se buscaba correr el código de un programa directamente en el hardware con la ejecución directa limitada, pero también se necesita virtualizar la memoria. Se va a usar un enfoque similar intentanto lograr tanto **eficiencia** como **control** mientras se produce la virtualización. Además de esto, ahora necesitaremos flexibilidad, ya que vamos a necesitar acomodar todos los procesos que se puedan en RAM y que sean fáciles de programar. [25]

Definición

La **traducción de direcciones**: cada acceso a memoria por `instruction fetch`, `load` o `store` se traduce de memoria virtual, donde el proceso piensa que está accediendo a la memoria, a memoria física, donde realmente se almacenan los datos. La traducción de direcciones es un componente clave de la virtualización de memoria, ya que permite que un programa en ejecución tenga una vista unificada y continua de la memoria, a pesar de que los datos pueden estar físicamente dispersos o no corresponder a los mismos valores de direcciones.

[25] OSTEP, Cap.15, Mecanismo de Traducción de Direcciones

Todas y cada una de las referencias a la memoria virtual del proceso que esta corriendo en LDE, tienen que ser traducidas a direcciones reales. De esto se encarga el hardware, te brinda los mecanismos clave y el sistema operativo los tiene que configurar y usar. Para tomar acciones si un proceso se sale de su espacio y administrar la memoria libre de la computadora. [26].

7.1. Ejemplo simple de entendiiento

Se tiene un programa que simplemente agarra una variable del stack y la incrementa en 3. El programa se compila en código assembler de la siguiente manera:

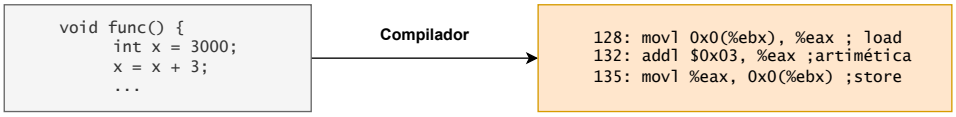

Figura 21: Código simple en C y su compilación

Ahora supongamos que la variable x se almacena en la dirección $15kb$. La traza de memoria sería la siguiente:

1. `Fetch instruction` en 128.
2. `Load` en 15kb.
3. `Fetch instruction` en 132.
4. `Fetch instruction` en 135.
5. `Store` en 15kb.

Desde perspectiva del programa, tiene su espacio de direcciones de una forma y la memoria física se ve diferente:

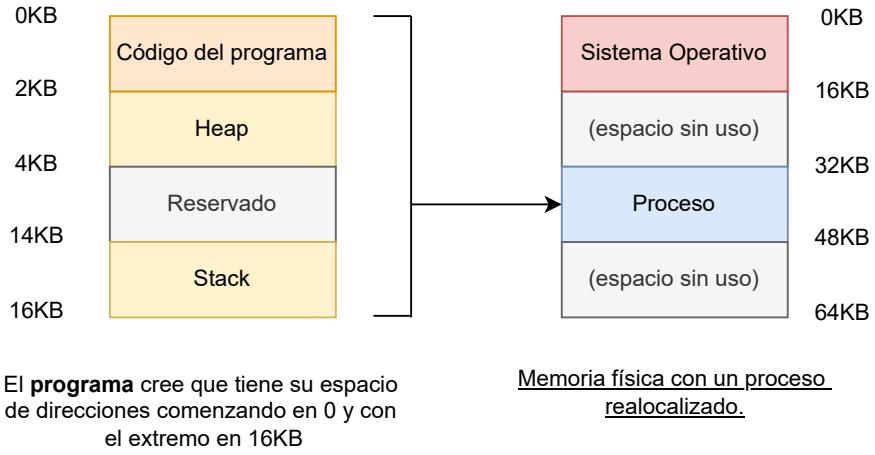

Figura 22: Traducción de direcciones

Este proceso de realocalización es lo que se conoce como **traducción de direcciones**. El hardware se encarga de realizar esta traducción, y el sistema operativo se encarga de configurar y mantener la tabla de traducción de direcciones.

7.2. Realocalización dinámica

Para comprender esta técnica, primero se necesitan dos registros, **registro base** y **registro límite**. El registro base contiene la dirección física donde comienza el espacio de direcciones del proceso, y el registro límite contiene el tamaño del espacio de direcciones del proceso. Cuando se realiza una referencia a la memoria, el hardware verifica que la dirección virtual esté dentro del rango permitido por el registro base y el registro

[26] Podcast - Nicolas Wolovick cap15

límite. Si la dirección virtual está fuera de este rango, se produce un error de segmentación. [27]. Con esta configuración, cada programa se escribe y compila como si estuviese cargado en la dirección cero. Sin embargo cuando se ejecuta, el sistema operativo decide donde cargarse en la memoria física y establece el registro base. Por ejemplo en el diagrama anterior el registro base se posiciona en 32KB. Entonces cuando el proceso genere cualquier referencia de memoria se traduce como: **dirección física = dirección virtual + registro base**. Y el registro límite se establece por el sistema operativo tomando en cuenta el espacio de direcciones, es decir **registro límite = registro base + tamaño del espacio de direcciones**. El registro límite se revisa en cada acceso a memoria virtual, si se accede por fuera, es decir a posiciones ilegales de la memoria, el hardware genera un trap.

Apoyo del hardware

- Modo protegido: un bit en la **processor status word**, para que los procesos de usuario no ejecuten operaciones privilegiadas.
- Nos brinda los registros **base and bounds**, base y límite, que permiten la traducción de direcciones de memoria y revisar los límites.
- Debe tener ademas **capacidad de traducción** de virtual a física y revisor de límites. Circuitos que comparan con el límite, y si está en dicho límite, se hace el desplazamiento necesario.
- También deben tener **instrucciones protegidas** que nos permitan cambiar la base y el límite.
- **Instrucciones protegidas** para registrar los administradores de excepciones, el sistema operativo le debe decir al hardware que ejecutar si ocurre algo raro, ya sea una dirección fuera de rango o una instrucción privilegiada.
- Finalmente, el hardware debe tener **capacidad de generar excepciones**, cuando ocurre algo excepcional, el procesador tiene que levantar una excepción.

Combinación de hardware y el sistema operativo

El hardware y el sistema operativo cooperan produciendo una implementación de memoria virtual muy sencilla:

- **Buscar espacio para un proceso nuevo**: teniendo un tamaño fijo, se define una lista enlazada donde se almacenen bloques libres, denominada **lista de bloques libres**. Cuando un proceso nuevo llega, el sistema operativo busca un bloque libre en la lista y lo asigna al proceso.
- **Proceso se retira de ejecución**: cuando un proceso termina, el sistema operativo libera el bloque de memoria asignado a ese proceso y lo devuelve a la lista de bloques libres.
- **Cambio de contexto**: en el cambio de contexto, el sistema operativo debe actuar, solamente hay un juego de registros base y límite, luego debemos multiplexar en el tiempo, por lo tanto, almacenarlos en el process control block, y al hacer el cambio de contexto cambiar a los nuevos registros base y límite.
- **Definir los handlers de excepción**: definir que se va a hacer cuando ocurra una excepción de fuera de rango, el sistema operativo debe tomar control con una rutina especial y usualmente se retirará al proceso.

[27] OSTEP, Cap.15, Mecanismo de Traducción de Direcciones

7.3. Ejercicio de realocalización dinámica

Asumiendo las siguientes características:

- Un espacio de direcciones virtuales de 1KB.
- Un registro base en 10000.
- Un registro límite en 100.

1. ¿A cúal de las siguientes **ubicaciones de memoria física** puede acceder legalmente el programa en ejecución?
   a) 0
   b) 1000
   c) 10000
   d) 10050
   e) 10100
2. ¿A cúal de las siguientes **direcciones virtuales** puede acceder legalmente el programa en ejecución?
   a) 0
   b) 1000
   c) 10000
   d) 10050
   e) 10100

Primero planteo los datos que tengo:

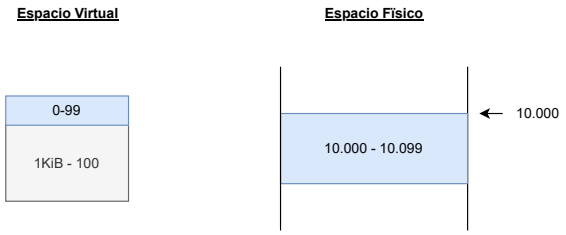

Figura 23: Datos del ejercicio

¿Cómo funcionaría el acceso a memoria dentro de esos límites? El sistema operativo cuando recibe una dirección la compara con el registro límite, y si esa comparación no es exitosa, se lanza la excepción, por lo tanto depende completamente del registro bound.

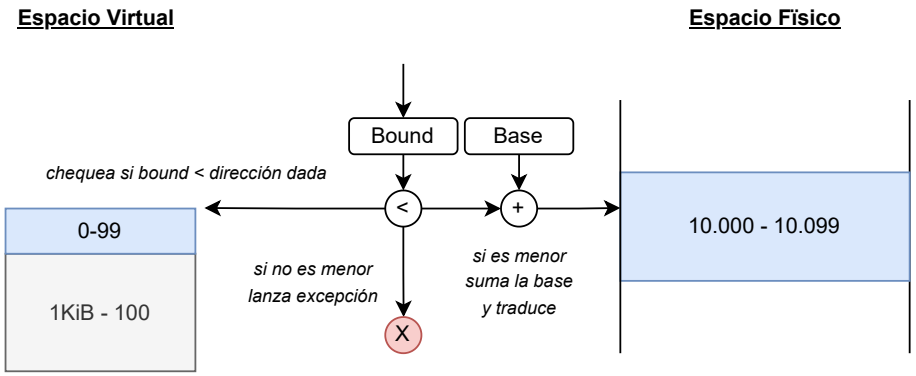

Figura 24: Acceso a memoria

Para el punto 1, sabemos que las direcciones de memoria física a las que se pueden acceder, deben estar dentro del registro bound, entonces, simplemente se debe chequear que queden dentro del rango de direcciones del espacio físico:

1. 0 **No**, ya que está por debajo del registro base.
2. 1000 **No**, ya que está por debajo del registro base.
3. 10000 **Sí**, ya que está dentro del rango.
4. 10050 **Sí**, ya que está dentro del rango.
5. 10100 **No**, ya que está en el límite. Los accesos son de 10000 a 10099.

Para el punto 2, es lo mismo, solo que ahora sabemos que el rango de direcciones virtuales es de 0 a 100, entonces:

1. 0 **Sí**, ya que está dentro del rango.
2. 1000 **No**, ya que está por encima del límite.
3. 10000 **No**, ya que está por encima del límite.
4. 10050 **No**, ya que está por encima del límite.
5. 10100 **No**, ya que está por encima del límite.

8. Administración del espacio libre

Definición

Un **administrador de memoria** es una abstracción que dado un pedazo de mmeoria brinca dos operaciones, `malloc()` y `free()`, que permiten pedir y devolver memoria contigua de cualquier tamaño, esta abstracción se usa tanto en espacio de usuario como en espacio del kernel.

El problema principal de administrar un espacio de memoria contigua donde se piden y devulven pedazos de memoria de longitud variable es la **fragmentación externa**, es decir, hay espacio libre pero no contiguo. [28].

8.1. Ejemplo de fragmentación externa

Supongamos que se produce la siguiente secuencia de eventos:

1. Llega el proceso A y pide una cantidad de memoria,
2. Llega el proceso B y pide el doble de memoria,
3. LLega el proceso C y pide otra cantidad arbitraria de memoria,
4. Se bloquea el proceso B,
5. Llega otro proceso D que se coloca donde estaba el proceso B, pero este ocupa menos por lo que queda otro espacio libre,
6. Ahora el proceso B se debloquea y pide la misma cantidad de memoria que antes, pero ahora no hay espacio contiguo.

El proceso B no se puede cargar a menos que se saque alguno de los procesos que está cargado en memoria. Es decir estamos aprovechando la memoria de mala manera, se llama fragmentación externa ya que es un problema por fuera de la partición. [29]

[28] OSTEP, Cap.17, Administración del Espacio Libre
[29] Clase 5 - Hugo Carrer - FCEFyN - UNC

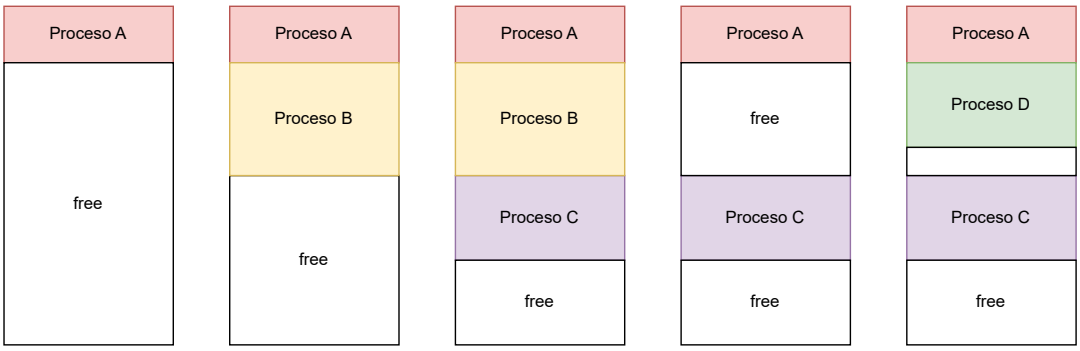

Figura 25: Fragmentación externa

8.2. Mecanismo de bajo nivel - División y fusión

La estructura de datos principal es la `free list`, que es una lista enlazada de bloques de memoria libres, dados por su dirección en el heap y su longitud. Cuando pedimos memoria de un tamaño t, lo que se debe hacer es recorrer los nodos cuyo tamaño sea mayor o igual a t y tomar el primero que se encuentre. Si el tamaño del bloque es mayor que t, se divide el bloque en dos, uno de tamaño t y otro de tamaño $s - t$, donde s es el tamaño del bloque original. Si se libera un bloque, se debe recorrer la lista de bloques libres y fusionar los bloques contiguos.

Definición

La fusión, o en inglés **coalescing**, es el proceso de combinar dos bloques de memoria contiguos en un solo bloque más grande. La fusión es una operación importante en la administración de la memoria, ya que ayuda a reducir la fragmentación externa y a mantener la lista de bloques libres lo más compacta posible.

Visualmente, supongamos que tenemos el siguiente escenario:

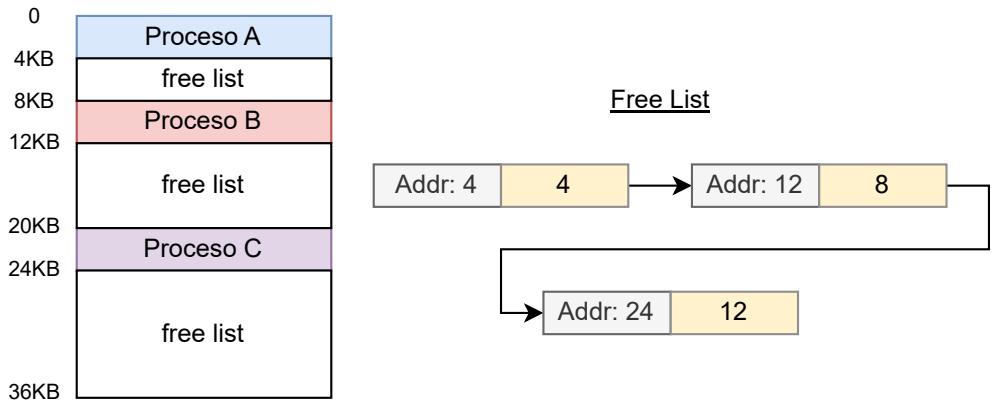

Figura 26: Fusión de bloques de memoria

Ahora bien, si se libera el Proceso B, que pasa con la free list, podría ya sea alargarse el valor del tamaño del primer nodo hacia adelante, o reducirse la dirección del segundo nodo y alargarse el tamaño también. Acá es donde entra el enfoque de la **fusión**, donde en vez de modificar uno u otro nodo representante de un segmento libre, los unimos en uno solo.

En el diagrama se mostró una representación visual mas amigable, formalmente podriamos decir que la estructura de la free list en realidad se almacena dentro de dichos espacios libres, de la siguiente manera:

Esto es ya que, como implementariamos una estructura donde se debe usar malloc y free cuando en realidad

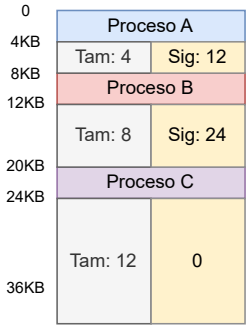

Figura 27: Fusión de bloques de memoria - Formal

estamos intentando hacer un manejo de memoria para estos mismos, el famoso problema recursivo de tantos problemas de la computación.

Observación

Notar que, la llamada `free(void ptr)` no toma un parámetro de tamaño; por lo tanto se asume que, dado un puntero, `malloc` puede determinar rápidamente el tamaño de la región de memoria que se está liberando y, por ende, reincorporar el espacio a la lista de memoria libre. Para lograr esta tarea, la mayoría de los asignadores almacenan un poco de información adicional en un bloque de encabezado que se mantiene en la memoria, generalmente justo antes del bloque de memoria asignado.

Supongamos que se llama a `ptr = malloc(20)`, el encabezado contiene, como mínimo, el tamaño de la región asignada (en este caso, 20); también puede contener punteros adicionales para acelerar la desasignación, un número mágico para proporcionar verificación de integridad adicional y otra información.

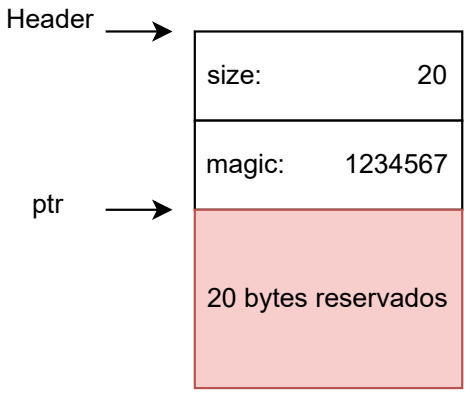

Figura 28: Encabezado de un bloque de memoria

Entonces se tendría una estructura como esta:

```
typedef struct {
    int size;
    int magic;
```

```
} header_t;
```

Cuando el usuario llame a `free(ptr)`, la biblioteca utiliza aritmética de punteros simple para calcular dónde comienza el encabezado:

```
void free(void *ptr) {
    header_t *hptr = (header_t *) ptr - 1;
    ...
}
```

Después de obtener dicho puntero al encabezado, la biblioteca puede determinar fácilmente si el número mágico coincide con el valor esperado como una verificación de coherencia (por ejemplo, assert(hptr-¿magic == 1234567)) y calcular el tamaño total de la región recién liberada mediante matemáticas simples (es decir, sumando el tamaño del encabezado al tamaño de la región).

Así se forma una estructura conjunta, en donde se almacena la información necesaria para la administración de la memoria, tanto de los espacios vacíos como de los ocupados. Solucionando el problema de interdependencia.

8.3. Políticas de asignación

Llamamos mecanismo a la forma de lidiar con la estructura de datos, una vez resuelto eso, hay que decidir en cual de todos los lugares posibles vamos a asignar memoria cuando un programa nos pide una cantidad determinada.

- **Best fit**: Elige de entre los espacios libres aquel que tiene la capacidad más parecida al espacio requerido por el nuevo proceso
  - Por lo general es el de peor performance.
  - Lo que garantiza es que el espacio sobrante es el menor posible.
  - Como consecuencia rápidamente la memoria queda fragmentada en muchos espacios chicos que no pueden albergar un proceso
- **Worst fit**: se asigna el bloque más grande disponible. Esto puede llevar a una fragmentación interna significativa, pero minimiza la fragmentación externa.
- **First fit**: Elige el primer espacio de memoria libre donde entre el proceso nuevo a cargar:
  - Es el más simple y rápido.
  - En muchas condiciones es el mejor.
  - Genera fragmentación al comienzo de la memoria lo que hace un poco mas lentas las búsquedas subsecuentes.
- **Next fit**: Elige el primer espacio de memoria libre donde entre el proceso nuevo a cargar pero buscando desde la posición donde se cargó el último proceso.
  - En este caso se genera fragmentación hacia el final de la memoria.
  - En general requiere mas frecuencia de compactación que el first-fit.

[30]

9. Paginación

Se dice que el sistema operativo toma uno de dos enfoques al resolver casi cualquier problema de gestión de espacio. El primer enfoque es dividir el espacio en piezas de tamaño variable, llamada **segmentación en memoria virtual**. Desafortunadamente, esta solución tiene dificultades inherentes. En particular, al dividir un espacio en fragmentos de diferentes tamaños, el espacio en sí puede fragmentarse, lo que hace que la asignación sea más complicada con el tiempo. Por lo tanto, puede valer la pena considerar el segundo enfoque: dividir el espacio en piezas de tamaño fijo. En memoria virtual, llamamos a esta idea **paginación**. La paginación, como veremos, tiene varias ventajas sobre nuestros enfoques anteriores. Probablemente, la mejora más importante será la flexibilidad: con un enfoque de paginación completamente desarrollado, el sistema podrá soportar la abstracción de un espacio de direcciones de manera efectiva, independientemente de cómo un proceso use el espacio de direcciones.

[30] Clase 5 - Hugo Carrer - FCEFyN - UNC

Definición

La **paginación** se desarrolla para reducir la fragmentación tanto interna como externa. Consiste en dividir la memoria principal en pedazos de la misma capacidad, relativamente chica, estos pedazos se llaman **marcos**. Se hace lo mismo con los procesos dividiéndolos en pedazos todos de la misma capacidad de los marcos, estos se conocen como **páginas**.

Al momento de cargar el proceso en memoria se asignan las páginas del mismo a un conjunto de marcos libres, no necesariamente contiguos.

Para ilustrar esto, supongamos que se tiene un pequeño espacio de direcciones, con solo 64 bytes en total, con cuatro páginas de 16 bytes (páginas virtuales 0, 1, 2 y 3). La memoria física, también consiste en un número de ranuras de tamaño fijo, en este caso ocho marcos de página. Se puede notar que las páginas del espacio de direcciones virtuales se han colocado en diferentes ubicaciones a lo largo de la memoria física; el diagrama también muestra que el sistema operativo usa parte de la memoria física para sí mismo.

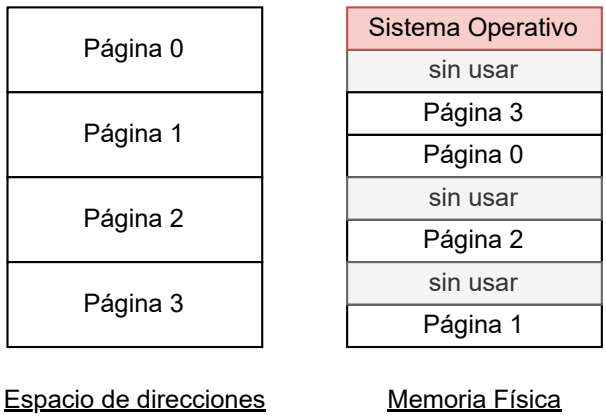

Figura 29: Ejemplo - Paginación

Para registrar dónde se coloca cada página virtual del espacio de direcciones en la memoria física, el sistema operativo generalmente mantiene una estructura de datos por proceso conocida como **tabla de páginas**. El papel principal de la tabla de páginas es almacenar las **traducciones de direcciones** para cada una de las páginas virtuales del espacio de direcciones, permitiéndonos saber dónde reside cada página en la memoria física.

En el programa cada dirección se representa como un número de página sumado a un desplazamiento dentro de la página, el número de página se representa como **VPN** (virtual page number) y el desplazamiento como **offset**.

9.1. Ejemplo de traducción de dirección

Supongamos que el proceso con el espacio de direcciones virtual de la figura anterior esta haciendo un acceso a memoria:

```
movl <virtual address>, %eax
```

Para traducir esta direccion virtual que genero el proceso, primero debemos dividirla en los dos componentes, VPN y offset. Como en este ejemplo el espacio de direcciones es de 64 bytes, necesitariamos 6 bits para la dirección virtual.

$Va5$	$Va4$	$Va3$	$Va2$	$Va1$	$Va0$

Donde Va5 sería el bit mas significativo, y Va0 el de menor orden, como sabemos que la página es de 16 bytes, deberiamos poder asignar 4 páginas, los dos primeros bits se encargan de eso. Ahora bien se quiere cargar la dirección virtual 21, es decir

```
movl 21, %eax
```

Se puede expresar como 010101 en binario, es decir está en el quinto byte de la página 1:

$Va5$	$Va4$	$Va3$	$Va2$	$Va1$	$Va0$
0	1	0	1	0	1

Ahora con el VPN y el offset, podemos, mediante PFN (physical frame number) y el offset, traducir la dirección virtual a la dirección física. En este caso, la VPN es 0101, es decir la página 1, y el offset es 0101, es decir el quinto byte de la página. La tabla de páginas nos dice que la página 1 se encuentra en el marco físico 7.

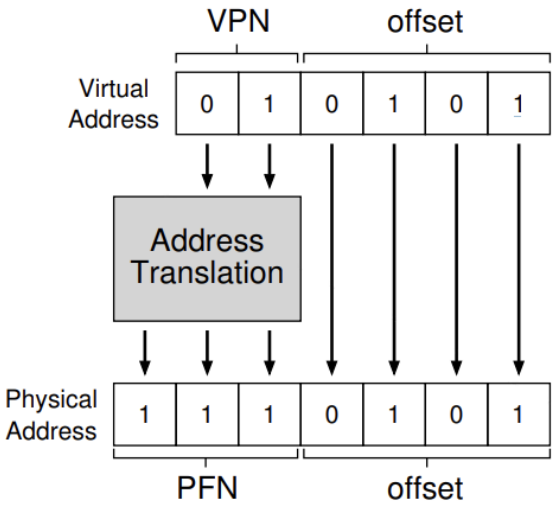

Figura 30: Traducción de dirección

Notar que el desplazamiento sigue igual, esto es ya que el desplazamiento solamente nos indica que byte dentro de la página queremos. La tabla de páginas se guardará en el sistema operativo de la siguiente manera:

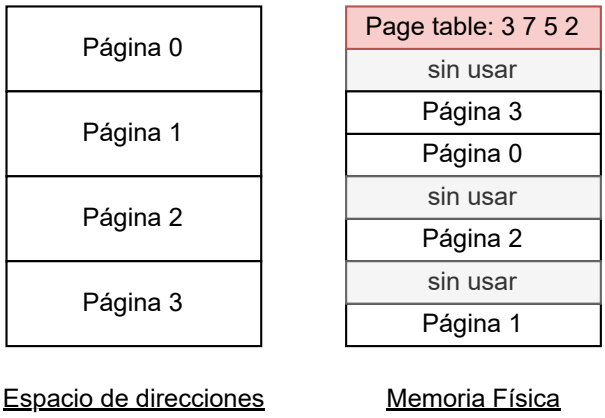

Figura 31: Tabla de páginas

9.2. Tabla de páginas

Definición

La tabla de páginas es simplemente una estructura de datos que se utiliza para mapear direcciones virtuales (o realmente, números de páginas virtuales) a direcciones físicas (números de marcos físicos).

La forma más simple se llama tabla de páginas lineal, que es solo un arreglo. El sistema operativo indexa el arreglo usando el número de página virtual (VPN) y busca la entrada de la tabla de páginas (PTE) en ese índice para encontrar el número de marco físico (PFN) deseado. Consta de lo siguiente:

- Un **bit de validez** es común para indicar si una traducción en particular es válida; por ejemplo, cuando un programa comienza a ejecutarse, tendrá código y un montón en un extremo de su espacio de direcciones, y la pila en el otro. Todo el espacio no utilizado en el medio será marcado como inválido, y si el proceso intenta acceder a esa memoria, generará una excepción al sistema operativo, que probablemente terminará el proceso. Por lo tanto, el bit de validez es crucial para soportar un espacio de direcciones disperso; simplemente marcando todas las páginas no utilizadas como inválidas, eliminamos la necesidad de asignar marcos físicos para esas páginas y, por lo tanto, ahorramos mucha memoria.

PFN El número de marco físico (PFN) es el número de marco físico en el que se encuentra la página virtual correspondiente.

R/W **Bits de protección**, que indican si se puede leer, escribir o ejecutar desde la página. == 1 si se puede leer y escribir, == 0 solo lectura.

P Un **bit de presencia** indica si esta página está en la memoria física o en el disco. == 1 la página está en memoria física, == 0 la página no está cargada, la CPU genera una excepción de fallo de página.

D Un **bit de modificación** es también común, indicando si la página ha sido modificada desde que se trajo a la memoria.

- D = 1: se ha escrito en la página, y cuando se necesite eliminarla de la memoria principal será preciso previamente actualizarla en la memoria virtual.
- D = 0: se puede sobrescribir cuando se necesita sustituirla por otra ya que no ha sido modificada durante su permanencia en la memoria principal.

A Un **bit de referencia** (también conocido como bit de acceso) se usa a veces para rastrear si una página ha sido accedida, y es útil para determinar qué páginas son populares y, por lo tanto, deberían mantenerse en la memoria. Se pone a 1 cada vez que se accede a dicha p´agina. En un lapso de tiempo, el SO lee este bit, si vale "1" lo pasa a "0" e incrementa el contador que tiene asociado a la página, al que le aplica el algoritmo LRU para eliminar de la memoria la página menos usada recientemente.

PCD (Aceptación de la Caché): indica si la página se puede o no meter en la memoria Caché.

PWT (Escritura obligada): Indica que la página, además de ser cacheable funcion en modo de escritura obligada.

[31]

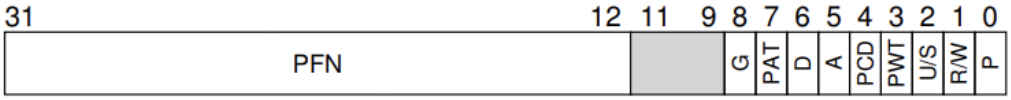

Figura 32: Entrada de la tabla de páginas

9.3. Ejemplo completo

Volviendo al ejemplo anterior, se tiene la siguiente instrucción que se quiere ejecutar:

```
movl 21, %eax
```

El sistema debe traducir la dirección 21 a 117, entonces, *antes de obtener los datos de la dirección 117, el sistema primero debe obtener la entrada de la tabla de páginas correcta del proceso, realizar la traducción y luego cargar los datos desde la memoria física.* Pero para hacer esto, el hardware debe saber dónde está la tabla de páginas del proceso que se está ejecutando. Supongamos por ahora que un solo registro base de la tabla de páginas contiene la dirección física de la ubicación inicial de la tabla de páginas. Para encontrar la ubicación de la PTE deseada, el hardware realizará las siguientes funciones:

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

[31] Mag. Ing. Miguel Solinas - Sistemas de Computación - FCEFyN - UNC

En este ejemplo, `VPN_MASK` es un número que se usa para enmascarar los bits de la dirección virtual que no son el número de página virtual, y `SHIFT` es el número de bits que se deben desplazar a la derecha para obtener el número de página virtual. En este caso, VPN_MASK sería 0x30 y SHIFT sería 4.

Por ejemplo, con la dirección virtual 21 (010101), al aplicar la mascara obtenemos 010000, que es el número de página virtual 1. Luego, se usa este número como índice en el arreglo de PTE's apuntado por el registro base de la tabla de páginas.

Una vez que se conoce esta dirección física, el hardware puede obtener la PTE de la memoria, extraer el PFN, y concatenarlo con el desplazamiento de la dirección virtual para formar la dirección física deseada. Podemos decir que el PFN se desplaza hacia la izquierda por SHIFT, y luego se combina con el desplazamiento mediante una operación OR bit a bit para formar la dirección final de la siguiente manera:

```
offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

Con esto el hardware ya puede obtener los datos de la dirección 117 y cargarlos en el registro `%eax`. Entonces, el enfoque completo del manejo de cada referencia de memoria sería algo como:

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
// Access OK: form physical address and fetch it
offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
Register = AccessMemory(PhysAddr)
```

Pero con esto surge un problema, sin un diseño cuidadoso tanto del hardware como del software, las tablas de páginas **harán que el sistema funcione demasiado lento** y ocupen demasiada memoria.

9.4. Traza de memoria

Ahora analizo la cantidad de accesos a memoria que se realizan en el método de paginación. Nos interesa el siguiente fragmento de código:

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

El código ensamblador que escupe el compilador es:

```
1024 movl $0x0,(%edi,%eax,4) ; mueve el 0 a la dirección de memoria a array[i]
1028 incl %eax               ; incrementa el contador
1032 cmpl $0x03e8,%eax       ; compara el contador con 1000
1036 jne 1024                ; si no es igual, salta a 1024
```

Asumamos que el espacio de direcciones virtual es de 64KB y un tamaño de página de 1KB. Lo único que necesitamos saber ahora es el contenido de la tabla de páginas y su ubicación en la memoria física. Supongamos que tenemos una tabla de páginas lineal (basada en un array) y que se encuentra en la dirección física 1KB (1024).

- Primero, está la página virtual donde reside el código. Como el tamaño de página es de 1KB, la dirección virtual 1024 reside en la segunda página del espacio de direcciones virtuales (VPN=1, ya que VPN=0 es la primera página). Supongamos que esta página virtual se mapea al marco físico 4 (VPN 1 → PFN 4).

- Luego, está el arreglo en sí. Su tamaño es de 4000 bytes (1000 enteros), y asumimos que reside en las direcciones virtuales 40000 a 44000 (sin incluir el último byte). Las páginas virtuales para este rango decimal son VPN=39 ... VPN=42. Por lo tanto, necesitamos mapeos para estas páginas. Supongamos los siguientes mapeos virtual-físicos para el ejemplo: (VPN 39 → PFN 7), (VPN 40 → PFN 8), (VPN 41 → PFN 9), (VPN 42 → PFN 10).

Cuando se ejecuta, cada búsqueda de instrucción generará dos referencias a la memoria: una a la tabla de páginas para encontrar el marco físico en el que reside la instrucción y otra a la instrucción misma para traerla a la CPU para su procesamiento. Además, hay una referencia explícita a la memoria en forma de la instrucción mov; esto añade otro acceso a la tabla de páginas primero (para traducir la dirección virtual del arreglo a la física correcta) y luego el acceso al arreglo en sí.

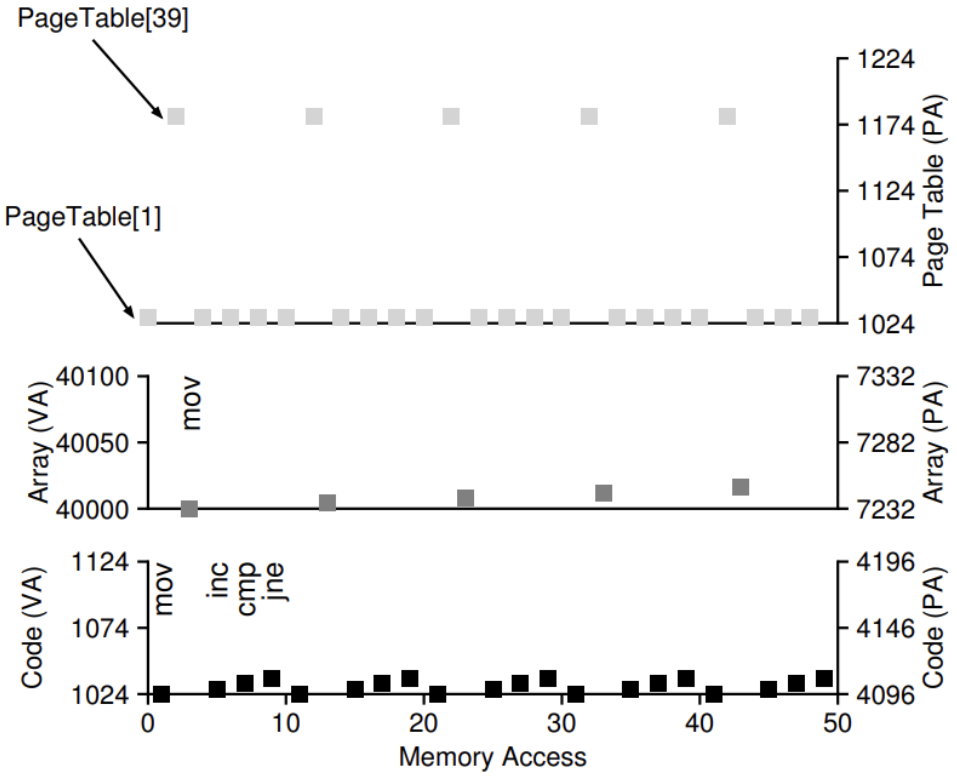

Figura 33: Traza de memoria

El gráfico inferior muestra las referencias de memoria de las instrucciones en el eje y en negro (con direcciones virtuales a la izquierda y las direcciones físicas reales a la derecha); el gráfico medio muestra los accesos al arreglo en gris oscuro (nuevamente con virtual a la izquierda y físico a la derecha); finalmente, el gráfico superior muestra los accesos a la tabla de páginas en gris claro (solo físico, ya que la tabla de páginas en este ejemplo reside en memoria física). El eje x, para todo el rastreo, muestra las referencias de memoria a lo largo de las primeras cinco iteraciones del bucle; hay 10 accesos a la memoria por iteración del bucle, que incluyen cuatro búsquedas de instrucciones, una actualización explícita de la memoria, y cinco accesos a la tabla de páginas para traducir esas cuatro búsquedas y una actualización explícita.

10. TLBs

La paginación como vimos cada instruction fetch, cada load y cada store, antes deben pasar por la tabla de traducción de marco virtual a marco físico. Para acelerar el proceso de traudcción, el hardware nos brinca una ayuda, introdujo dentro del propio chip una pequeña memoria caché especial ultra rápida, denominada **Translation Lookaside Buffer** (TLB). Es decir es solamente una caché de traducción.

10.1. Algoritmo básico de la TLB

Supongamos que tenemos una tabla de páginas lineal, tendríamos esta secuencia de trabajo:

1. Obtener el **VPN**,

2. Buscar en la TLB, si es un **hit**, obtenemos el **PFN** y el **offset**, si hay un **miss**, se busca en la tabla de páginas, la PTE que corresponda a esa VPN, se extraen los 20bits de la PFN, se actualiza la TLB con la nueva entrada y se reejecuta la instrucción para que funcione.

Lo que buscamos es aumentar la probabilidad de que la traducción de una dirección virtual a una dirección física se encuentre en la TLB.

10.2. Ejemplo

Supongamos que tenemos un arreglo de 10 enteros de 4-bytes en memoria. Se tendría un esquema de memoria virtual de 4 VPN, esto quiere decir que vamos a tener los 4 bits mas significativos de la dirección virtual para la VPN, y los otros 4 para el offset. Cada página tiene 4 enteros. Y vamos a tener A[0] esta en la segunda casilla del marco virtual número 6, entonces

- 3 en el marco virtual 6,
- 4 en el marco virtual 7,
- los 3 finales en el marco virtual 8.

Vamos a pensar en este fragmento de código que recorre cada uno de esos elementos en el arreglo y los acumula en un registro del procesador que va a ser la suma de ellos:

```
int i, sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

La secuencia va a ser la siguiente:

1. Se accede a a[0], el número de página virtual es el 6, que no esta en la TLB, se produce un miss, se busca en la tabla de páginas, se actualiza la TLB y se ejecuta la instrucción.
2. Vuelve a acceder a a[0] pero esta vez ya está en la TLB, por lo que se produce un hit, se obtiene el PFN y el offset, y se ejecuta la instrucción.
3. Se accede a a[1], el número de página virtual es el 6, que ya está en la TLB, por lo que se produce un hit, se obtiene el PFN y el offset, y se ejecuta la instrucción.
4. Se accede a a[2], sigo estando en el marco virtual 6, por lo que se produce un hit, se obtiene el PFN y el offset, y se ejecuta la instrucción.
5. Se accede a a[3], el número de marco virtual es el 7, que no está en la TLB, por lo que se produce un miss, se busca en la tabla de páginas, se actualiza la TLB y se ejecuta la instrucción.
6. Se accede a a[4], el número de marco virtual es el 7, que ya está en la TLB, por lo que se produce un hit, se obtiene el PFN y el offset, y se ejecuta la instrucción.
7. Se accede a a[5], el número de marco virtual es el 7, que ya está en la TLB, por lo que se produce un hit, se obtiene el PFN y el offset, y se ejecuta la instrucción.
8. Se accede a a[6], el número de marco virtual es el 7, que ya está en la TLB, por lo que se produce un hit, se obtiene el PFN y el offset, y se ejecuta la instrucción.
9. Se accede a a[7], no está en la TLB, se produce un miss, se busca en la tabla de páginas, se actualiza la TLB y se ejecuta la instrucción.
10. Se accede a a[8], el número de marco virtual es el 8, que no está en la TLB, se produce un miss, se busca en la tabla de páginas, se actualiza la TLB y se ejecuta la instrucción.
11. Se accede a a[9], el número de marco virtual es el 8, que ya está en la TLB, por lo que se produce un hit, se obtiene el PFN y el offset, y se ejecuta la instrucción.

De los 10 accesos que hice a la memoria **7** son TLB hits, y **3** son TLB misses.

Obervación

Notar que el tamaño de página juega un rol muy importante, si el tamaño de página fuese mas grande, producirá menos TLB misses, pero si el tamaño de página es muy grande, se producirá más fragmentación interna. Normalmente el tamaño de página es de 4KB.

10.3. TLB miss

¿Quién administra los TLB miss? En intel x86, el hardware se encarga de esto, esto implica que sabe donde está la page table, esto se logra a través de un registro llamado **CR3**, tiene que caminar dicha page table hasta encontrar la PTE que corresponde, y luego actualizar la TLB, para luego reintentar la ejecución.

Definición

Un TLB **miss** es un trap, una excepción, se pasa a kernel mode, se pasa de trap handler, se recorre la page table por software y se obtiene el PFN, se actualiza la TLB mediante una instrucción privilegiada y se hace return from trap.

10.4. Estructura y funcionamiento de la TLB

Una TLB es chica, tiene pocas entradas que asocian un marco virtual a un marco físico, es decir, una VPN a un PFN, este cache se llama completamente asociativo, ya que dado un VPN se consultan las entradas en forma paralela.

Una entrada del TLB debe contener **VPN**, **PFN**, y mas información como por ejemplo si la page table está presente o si es válida, además puede contener los bits R/W. Y también contienen un bit que indica si esa entrada de la TLB es válida.

Notar que al producirse un cambio de contexto, debo poner todos los bits de la TLB en 0, ya que no se puede garantizar que la información que está en la TLB sea válida.

10.4.1. Cambio de contexto

Ahora bien, ¿qué pasa cuando se produce un cambio de contexto? Cuando cambiamos de contexto, apuntamos a otra page table, esto implica que todas las entradas de la TLB quedan inválidas, no hay que usar traducciones de virtuales a físicas de otros procesos, es un problema típico de consistencia de caché.

Supongamos que tenemos el proceso P1 corriendo, y mapea de página virtual a física 10 a 100, cambia de contexto al proceso P2 y el mismo mapea la página virtual 10 en página física 170. ¿Cuál va a ser el contenido de la TLB para la página virtual número 10? La solución simple es tirar la cadena de la TLB en cada cambio de contexto.

Una solución a esto es agregar un campo llamado **ASID** (Address Space Identifier) a la TLB, que es un número que identifica a cada proceso, entonces cuando se produce un cambio de contexto, se cambia el ASID, y se puede saber si la entrada de la TLB es válida o no. Así la TLB del proceso se vería como:

VPN	PFN	$Valid$	R/W	$ASID$
10	100	1	rwx	1
–	–	0	–	–
10	170	1	rwx	2
–	–	0	–	–

10.5. Políticas de reemplazo de la TLB

Todo caché tiene un problema, se llena rapidamente, si se llena la TLB y viene una nueva página para traducir de virtual a física, ¿qué entrada tengo que usar para poner la nueva?, esto se denomina **política de reemplazo**. Si la política de reemplazo es **LRU** (Least Recently Used), se reemplaza la entrada que no se usó hace más tiempo. Tambíen son válidas las políticas aleatorias, es decir se elige una entrada al azar.

11. Paginación Multinivel

Las page table lineales son grandes, por ejemplo en un esquema 20 12, con páginas de 4KB, y 32 bits de espacio de direcciones, la page table lineal ocupa 4Bytes por 2^{20} entradas, es decir 4MB. Esto es un problema, ya que si tengo un proceso que solo usa 1MB de memoria, tengo que tener 4MB de page table. Y si tenemos 100 procesos, necesitamos 400MB de page table, lo cual es un problema de sobrecarga. El enfoque de paginación es flexible pero caro.

La solución es simple, páginas mas grandes, al agrandar el tamaño de página, pero si las páginas son grandes, se desperdicia lugar, el famoso problema de fragmentación interna. La mayoria de los sistemas operativos usan páginas chicas.

11.1. Solución híbrida

Se basa en combinar la paginación y la segmentación para reducir la sobrecarga de memoria de las tablas de páginas. Vamos a ver un ejemplo para analizar porqué esto podría funcionar:

Supongamos que tenemos un espacio de direcciones en el que las partes usadas del heap y la pila son pequeñas. Para el ejemplo, usamos un espacio de direcciones de 16KB con páginas de 1KB. La tabla de páginas

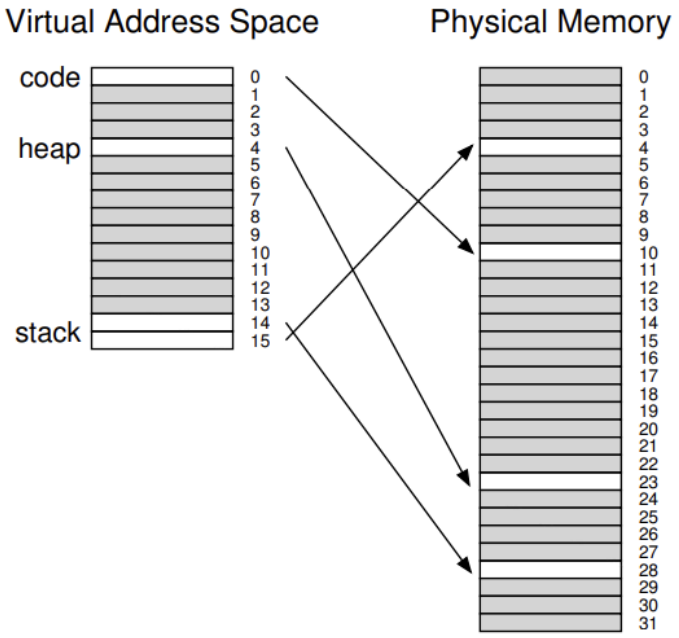

Figura 34: Espacio de direcciones - Ejemplo

para este espacio de direcciones se muestra en la figura siguiente: Este ejemplo asume que la única página de código (VPN 0) está asignada a la página física 10, la única página del heap (VPN 4) a la página física 23, y las dos páginas de la pila al otro extremo del espacio de direcciones (VPNs 14 y 15) están asignadas a las páginas físicas 28 y 4, respectivamente. Notar que la mayor parte de la tabla de páginas está sin usar, llena de entradas inválidas.

En nuestro enfoque híbrido, en lugar de tener una sola tabla de páginas para todo el espacio de direcciones del proceso, ¿por qué no tener una por cada segmento lógico? En este ejemplo, podríamos tener tres tablas de páginas, una para el código, el heap y la pila.

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Figura 35: Tabla de páginas - Ejemplo

Para determinar a qué segmento se refiere una dirección, usaremos los dos bits superiores del espacio de direcciones. Supongamos que 00 es el segmento no usado, 01 para el código, 10 para el heap y 11 para la pila. Así, una dirección virtual se ve como esto: En el hardware, supongamos que hay tres pares de registros base/límite,

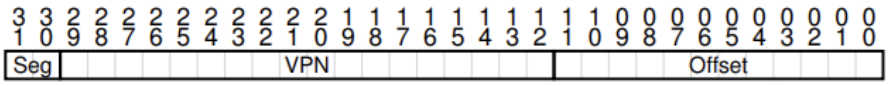

Figura 36: Dirección virtual - Ejemplo

uno para cada código, heap y pila. Cuando un proceso se está ejecutando, el registro base para cada uno de estos segmentos contiene la dirección física de una tabla de páginas lineal para ese segmento; por lo tanto, cada proceso en el sistema ahora tiene tres tablas de páginas asociadas. En un cambio de contexto, estos registros deben cambiarse para reflejar la ubicación de las tablas de páginas del nuevo proceso en ejecución.

En una falta de TLB (asumiendo una TLB gestionada por hardware), el hardware usa los bits del segmento (SN) para determinar qué par de registros base y límite usar. Luego, el hardware toma la dirección física allí y la combina con el VPN para formar la dirección de la entrada de la tabla de páginas (PTE):

```
SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

Este enfoque tiene sus problemas, primero, aún requiere que usemos segmentación; como discutimos antes, la segmentación no es tan flexible como nos gustaría, ya que asume un cierto patrón de uso del espacio de direcciones; si tenemos un heap grande pero poco usado, por ejemplo, todavía podemos terminar con mucho desperdicio en la tabla de páginas. Segundo, este híbrido vuelve a generar fragmentación externa. Aunque la mayor parte de la memoria se gestiona en unidades de tamaño de página, las tablas de páginas ahora pueden tener tamaños arbitrarios (en múltiplos de entradas de tabla de páginas). Así, encontrar espacio libre para ellas en la memoria es más complicado.

11.2. Paginación multinivel

Un enfoque diferente no depende de la segmentación, pero aborda el mismo problema: ¿cómo deshacerse de todas esas regiones inválidas en la tabla de páginas en lugar de mantenerlas todas en la memoria? A este enfoque lo llamamos una tabla de páginas multinivel, ya que convierte la tabla de páginas lineal en algo parecido a un árbol. Este enfoque es tan efectivo que muchos sistemas modernos lo emplean (por ejemplo, x86).

La idea básica detrás de una tabla de páginas multinivel es simple. Primero, divide la tabla de páginas en unidades del tamaño de una página; luego, si una página entera de entradas de tabla de páginas (PTEs) es inválida, no se asigna esa página de la tabla de páginas en absoluto. Para rastrear si una página de la tabla de páginas es válida (y si es válida, dónde está en la memoria), se usa una nueva estructura, llamada directorio de páginas. El directorio de páginas puede decirte dónde está una página de la tabla de páginas o si toda la página de la tabla de páginas no contiene páginas válidas.

Al partir la page table, las que al menos tengan una de sus entradas válidas, se van a "pegar", por medio de una meta page table, también conocida como page directory. Que también entra en una página y apunta a las otras page tables desperdigadas. Esta es una forma eficiente de codificar espacios de memoria raros.

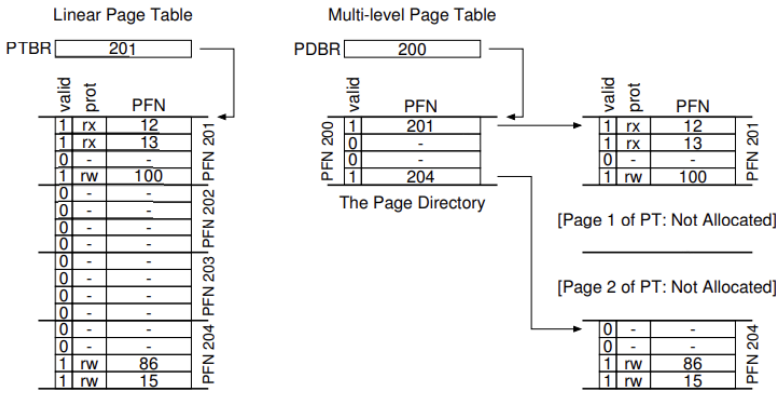

Figura 37: Paginación multinivel

A la izquierda está la clásica tabla de páginas lineal; aunque la mayoría de las regiones intermedias del espacio de direcciones no son válidas, aún se requiere espacio en la tabla de páginas asignado para esas regiones (es decir, las dos páginas intermedias de la tabla de páginas). A la derecha está una tabla de páginas multinivel. El directorio de páginas **marca solo dos páginas** de la tabla de páginas como **válidas** (la primera y la última); por lo tanto, solo esas dos páginas de la tabla de páginas residen en la memoria. Así se ve una forma de visualizar lo que está haciendo una tabla multinivel: simplemente hace que partes de la tabla de páginas lineal desaparezcan (liberando esos marcos para otros usos) y rastrea qué páginas de la tabla de páginas están asignadas mediante el directorio de páginas.

El directorio de páginas, en una tabla de dos niveles, contiene una entrada por cada página de la tabla de páginas. Consiste en un número de entradas del directorio de páginas (PDE). Un PDE (mínimamente) tiene un bit de validez y un número de marco de página (PFN), similar a un PTE. Sin embargo, como se mencionó antes, el significado de este bit de validez es ligeramente diferente: si el PDE es válido, significa que la page table que le sigue está presente o no, y si no es válido, significa que la page table no está presente.

Existe un registro llamado Page Directory Base Register (PDBR) que apunta a la dirección física del directorio de páginas.

11.3. Ejemplo completo

Supongamos que se tiene un pequeño espacio de direcciones de 16 KB, con páginas de 64 bytes. Así, tenemos un espacio de direcciones virtuales de 14 bits, con 8 bits para el VPN (número de página virtual) y 6 bits para el desplazamiento. Una tabla de páginas lineal tendría 2^8 (256) entradas, incluso si solo se utiliza una pequeña porción del espacio de direcciones. Como se ve acontinuación:

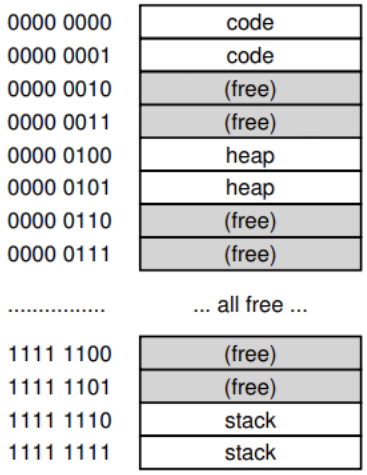

Figura 38: Tabla de páginas lineal

En este ejemplo, las páginas virtuales 0 y 1 son para código, las páginas virtuales 4 y 5 para el montón (heap), y las páginas virtuales 254 y 255 para la pila; el resto de las páginas del espacio de direcciones no se utilizan.

Para construir una tabla de páginas de dos niveles para este espacio de direcciones, comenzamos con nuestra tabla de páginas lineal completa y la dividimos en unidades del tamaño de una página. Recuerda que nuestra tabla completa (en este ejemplo) tiene 256 entradas; supongamos que cada entrada de la tabla de páginas (PTE) tiene 4 bytes de tamaño. Por lo tanto, nuestra tabla de páginas ocupa 1 KB (256 × 4 bytes). Dado que tenemos páginas de 64 bytes, la tabla de páginas de 1 KB puede dividirse en 16 páginas de 64 bytes; cada página puede contener 16 entradas de la tabla de páginas.

Ahora necesitamos entender cómo tomar un VPN y usarlo primero para indexar en el directorio de páginas y luego en la página de la tabla de páginas. Recuerda que ambos son arrays de entradas; por lo tanto, solo necesitamos averiguar cómo construir el índice de cada uno a partir de partes del VPN. Primero indexemos en el directorio de páginas. Nuestra tabla de páginas en este ejemplo es pequeña: 256 entradas, distribuidas en 16 páginas.

El directorio de páginas necesita una entrada por página de la tabla de páginas; por lo tanto, tiene 16 entradas. Como resultado, necesitamos cuatro bits del VPN para indexar en el directorio; usamos los cuatro bits superiores del VPN, de la siguiente manera:

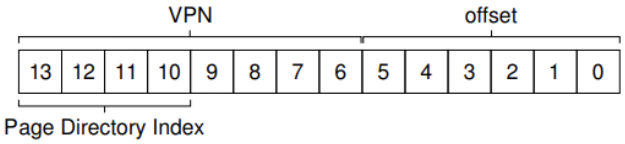

Figura 39: Direccionamiento

Una vez que extraemos el índice del directorio de páginas (PDIndex para abreviar) del VPN, podemos usarlo para encontrar la dirección de la entrada del directorio de páginas (PDE) con un cálculo simple:

$$\text{PDEAddr} = \text{PDBR} + (\text{PDIndex} \times \text{sizeof(PDE)})$$

Donde PDBR es el registro base del directorio de páginas, sizeof(PDE) es el tamaño de una entrada del directorio de páginas (en este caso, 4 bytes), y PDIndex es el índice del directorio de páginas que acabamos de calcular.

Si la entrada del directorio de páginas está marcada como inválida, sabemos que el acceso es inválido y, por lo tanto, se lanza una excepción. Sin embargo, si el PDE es válido, tenemos más trabajo por hacer. Específicamente, ahora tenemos que obtener la entrada de la tabla de páginas (PTE) de la página de la tabla de páginas a la que apunta esta entrada del directorio de páginas. Para encontrar este PTE, debemos indexar en la parte de la tabla de páginas usando los bits restantes del VPN:

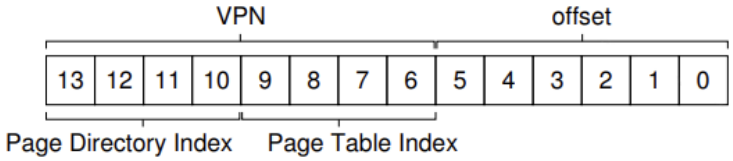

Figura 40: Entrada de la tabla de páginas

Este índice de la tabla de páginas (PTIndex para abreviar) puede usarse para indexar en la tabla de páginas en sí, dándonos la dirección de nuestro PTE:

$$\text{PTEAddr} = (\text{PDE.PFN} \ll \text{SHIFT}) + (\text{PTIndex} \times \text{sizeof(PTE)})$$

Observa que el número de marco de página obtenido de la entrada del directorio de páginas debe desplazarse a la izquierda antes de combinarlo con el índice de la tabla de páginas para formar la dirección del PTE.

Ahora, llenaremos una tabla de páginas multinivel con algunos valores reales y traduciremos una dirección virtual.

En este ejemplo, tenemos dos regiones válidas en el page directory (al principio y al final), y un número de mapeos inválidos en el medio. En la página física 100, tenemos la primera página de 16 entradas de la tabla de páginas para los primeros 16 VPN en el espacio de direcciones.

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Figura 41: Entrada de la tabla de páginas

Esta página de la tabla de páginas contiene los mapeos para los primeros 16 VPN; en nuestro ejemplo, los VPN 0 y 1 son válidos (el segmento de código), al igual que los 4 y 5 (el montón). Por lo tanto, la tabla tiene información de mapeo para cada una de esas páginas. El resto de las entradas están marcadas como inválidas.

La otra página válida de la tabla de páginas se encuentra en el marco de página 101. Esta página contiene los mapeos para los últimos 16 VPN del espacio de direcciones.

En el ejemplo, los VPN 254 y 255 (la pila) tienen mapeos válidos. Se ve que el espacio se puede ahorrar con una estructura indexada multinivel. En este ejemplo, en lugar de asignar las dieciséis páginas completas para una tabla de páginas lineal, asignamos solo tres: una para el directorio de páginas, y dos para los fragmentos de la tabla de páginas que tienen mapeos válidos. Los ahorros para espacios de direcciones grandes (de 32 o 64 bits) podrían ser obviamente mucho mayores.

11.4. Ventajas y desventajas

Ventajas:

- Se almacena lo que se mapea, no se desperdicia espacio.
- El tamaño total del page directory mas las page tables es lineal respecto a la memoria ocupada.

Desventajas:

- Cada vez que se necesita traducir de virtual a físico, se requiere acceder a dos tablas.

11.5. Método de resolución de ejercicios de paginación

11.5.1. Paginación Lineal

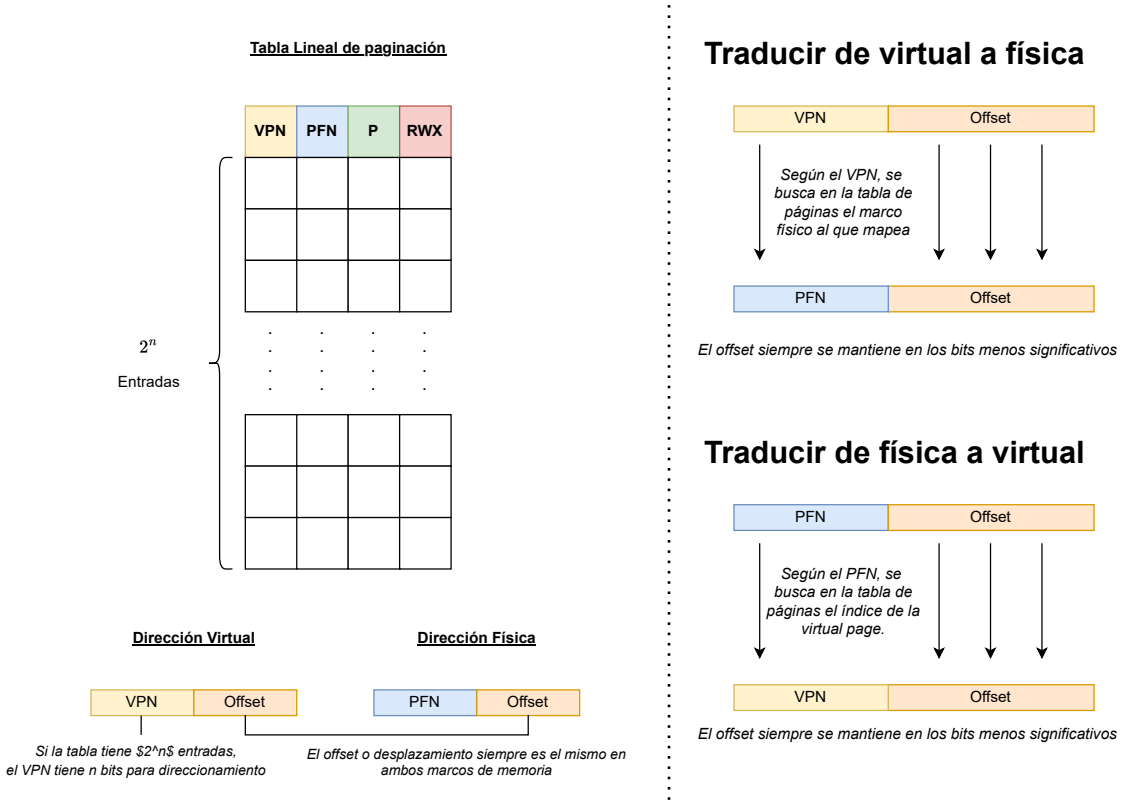

Figura 42: Paginación lineal

En el paso de traducción hay que tener en cuenta posibles casos no tan comunes:

- El mapeo de virtual a físico no es necesariamente inyectivo, es decir, una dirección física puede ser mapeada a varias direcciones virtuales.
- Una consecuencia de ese mapeo no inyectivo es que si hay que traducir una dirección física a una virtual, no se puede hacer de forma única.
- Se produce un **page fault** cuando se intenta acceder a una dirección virtual que no está mapeada a una dirección física.
- Si se habla de legalidad de manejo de una dirección, hay que tener en cuenta los bits de protección de la PTE.
- Cuando el bit de presencia es 0, se produce un page fault.

11.5.2. Paginación Multinivel

Hay una convención que se sigue para representar la distribución de bits en una dirección virtual. En el caso de la paginación multinivel, se divide la dirección virtual en tres partes:

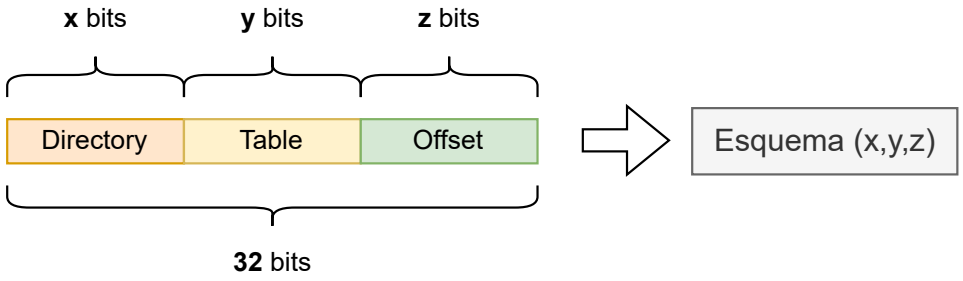

Figura 43: Paginación multinivel - Convención

A grandes rasgos, la dirección virtual se divide en tres partes cuyas funciones son:

1. **PDIndex**: Se usa para indexar en el directorio de páginas, que contiene las direcciones de las tablas de páginas. El registro base de esta estructura es el CR3.
2. **PTIndex**: Se usa para indexar en la tabla de páginas, que contiene las direcciones físicas de las páginas.
3. **Offset**: Se usa para indexar en la página física.

Cuando los ejercicios mencionan i386, se refieren a la arquitectura de 32 bits de Intel. Aquí se tiene un esquema (10,10,12):

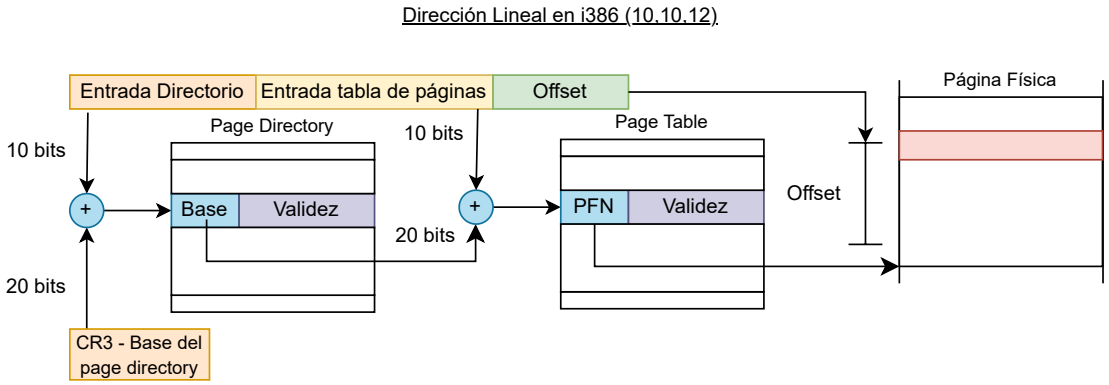

Figura 44: Paginación multinivel - i386

Método de traducción:

1. Se extraen los bits de PDIndex y se indexa en el directorio de páginas.
2. Una vez se tenga la dirección de la tabla de páginas, se extraen los bits de PTIndex y se indexa en la tabla de páginas.
3. Se extraen los bits de offset y se indexa en la página física.

12. Ejercicios de Examen

12.1. Ejercicio 1 - Parcial 1 2022

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 0x3008 **exclusive**. Como sus elementos son `unsigned long`, cada uno ocupa 8bytes y por lo tanto tiene 9 elementos.

```
0000000000000634 <main>:
634: 0613                               li      a2,0x3008       # <__BSS_END__> &a[9]
636: b206                               li      a5,0x2FC8       # <a+0x8> &a[1]
638: 6398                               ld      a4,0(a5)        # a4 = a[i]
63a: fff87b683                          ld      a3,-8(a5)       # a3 = a[i-1]
63e: 9736                               add     a4,a4,a3        # a4 = a[i] + a[i-1]
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**.

- Inicialización: Instruction fetch de la dirección 0x634.
- Inicialización: Instruction fetch de la dirección 0x636.
- Vuelta 1: 0x638, 0x2FC8, 0x63A, 0x2FC0, 0x63e, 0x640, 0x2FC8, 0x642, 0x644
- Vuelta 2: 0x638, 0x2FD0, 0x63A, 0x2FC8, 0x63e, 0x640, 0x2FD0, 0x642, 0x644
- Vuelta 3: 0x638, 0x2FD8, 0x63A, 0x2FD0, 0x63e, 0x640, 0x2FD8, 0x642, 0x644
- Vuelta 4: 0x638, 0x2FE0, 0x63A, 0x2FD8, 0x63e, 0x640, 0x2FE0, 0x642, 0x644
- Vuelta 5: 0x638, 0x2FE8, 0x63A, 0x2FE0, 0x63e, 0x640, 0x2FE8, 0x642, 0x644
- Vuelta 6: 0x638, 0x2FF0, 0x63A, 0x2FE8, 0x63e, 0x640, 0x2FF0, 0x642, 0x644
- Vuelta 7: 0x638, 0x3000, 0x63A, 0x2FF8, 0x63e, 0x640, 0x3000, 0x642, 0x644
- Fin: Instruction fetch de la dirección 0x648.

12.2. Ejercicio 2 - Parcial 1 2022

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

12.3. Ejercicio 4 - Parcial 1 2022

Tenemos un esquema de paginación de RISC-V con páginas de 4KiB de 3 niveles con formato 9,9,9,12 − > 44,12 como muestra la siguiente figura:

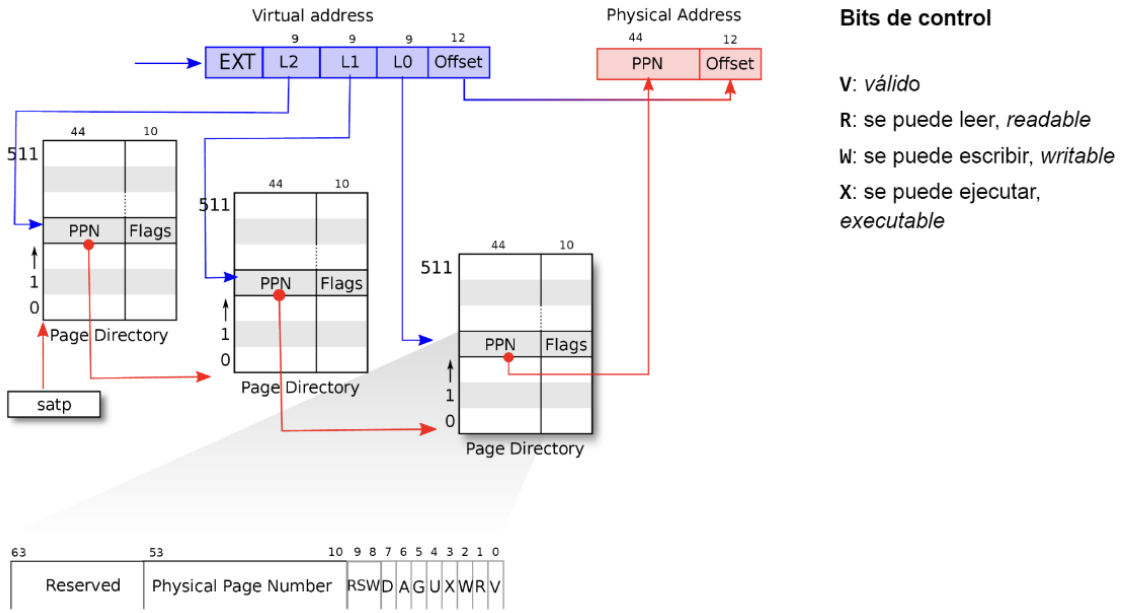

Figura 45: Esquema de paginación- Ejercicio de Parcial

Supongamos que tenemos el registro de paginación apuntando al marco físico satp=0x00000000FE0

0x00000000FE0	0x00000000FEA	0x000000AD0BE
0x1FF: 0x00000000000, ----	0x1FF: 0x00000000000, ----	0x1FF: 0x00000000000, ----
⋮	⋮	⋮
0x004: 0x00000000000, ----	0x004: 0x00000000000, ----	0x004: 0x00000000000, ----
0x003: 0x00000000000, ----	0x003: 0x00000000000, ----	0x003: 0x00000D1AB10, XWR-
0x002: 0x00000000FEA, XWRV	0x002: 0x000000AD0BE, XWRV	0x002: 0x00000DECADA, -WRV
0x001: 0x00000000FEA, XWRV	0x001: 0x000000AD0BE, XWRV	0x001: 0x000CAFECAFE, ----
0x000: 0x00000000FEA, XWRV	0x000: 0x000000AD0BE, XWRV	0x000: 0x0000000ABAD, X--V

Figura 46: Registro de paginación - Ejercicio de Parcial

1. Traducir de **virtual a física** las direcciones:
   a) 0x0000,
   b) 0x1000.
2. Traducir de la dirección física 0xDECADA980 a **todas** las virtuales que la apuntan.

Por lo pronto tenemos una paginación de 3 niveles con esquema 9,9,9,12. Esto quiere decir que los primeros 9 bits me indexan el primer directorio de páginas, los siguientes 9 bits me indexan el segundo directorio de páginas, los siguientes 9 bits me indexan la tabla de páginas y los últimos 12 bits me indexan la página física.
Punto 1:
0x0000: 000 0000 00 | 00 0000 000 | 0 0000 0000 | 0000 0000 0000

- Índice 0 en el primer directorio de páginas.
- Índice 0 en el segundo directorio de páginas.
- Índice 0 en la tabla de páginas.
- Mapea a la dirección 0x0000000ABAD000. Con un offset de 0x000.

0x1000: 000 0000 00 | 00 0000 000 | 0 0000 0001 | 0000 0000 0000

- Índice 0 en el primer directorio de páginas.
- Índice 0 en el segundo directorio de páginas.
- Índice 1 en la tabla de páginas.
- No mapea ya que la entrada no está presente.

Punto 2: Hay varias direcciones de memoria virtual que apuntan a la dirección física 0xDECADA980. Para encontrarlas, se debe recorrer la tabla de páginas y los directorios de páginas. Van a haber varios caminos que vayan a parar al mismo lugar, lo único que se mantiene fijo es el offset 980.

12.4. Ejercicio 5 - Parcial 1 2022

A la luz del esquema de paginación del ejercicio 4, indicar de que manera esquemática que es lo que pasaría con la traza de memoria respecto a la ejecución de un proceso corriendo el código máquina del ejercicio 1.

A modo de ilustración, si se tiene por ejemplo la dirección de instructión fech en 0x634, se tiene:
000 0000 00 | 00 0000 000 | 0 0000 0000 | 0110 0011 0100

- Índice 0 en el primer directorio de páginas.
- Índice 0 en el segundo directorio de páginas.
- Índice 0 en la tabla de páginas.
- Mapea a la dirección 0x0000000ABAD634. Con un offset de 0x634.

Esto quiere decir que el acceso mediante la instrucción fetch a la dirección 0x634 se traduce a la dirección física **0x0000000ABAD634**.

12.5. Ejercicio Parcial 2021

Tenemos un esquema de paginación i386 o sea (10,10,12).
10 bits de índice de directorio, 10 bits de índice de tabla de página y 12 bits de offset.

Dar la dirección física de la dirección virtual 0x00C03EEE.

Si hay *page fault* poner PF.

```
CR3=0x01011

0x01010                |  0x01011
---                    |
   0: 0x01010, P, RWX      0: 0x01011, P, RWX
   1: 0x01011, P, RWX      1: 0x01010, P, RWX
   2: 0x01010, P, RWX      2: 0x01011, P, RWX
   3: 0x01011, P, RWX      3: 0x01010, P, RWX
...
1023: 0x01011, P, RWX  1023: 0x01010, P, RWX
```

Respuesta: 0x01011EEE ✔

Figura 47: Ejercicio Parcial 2021

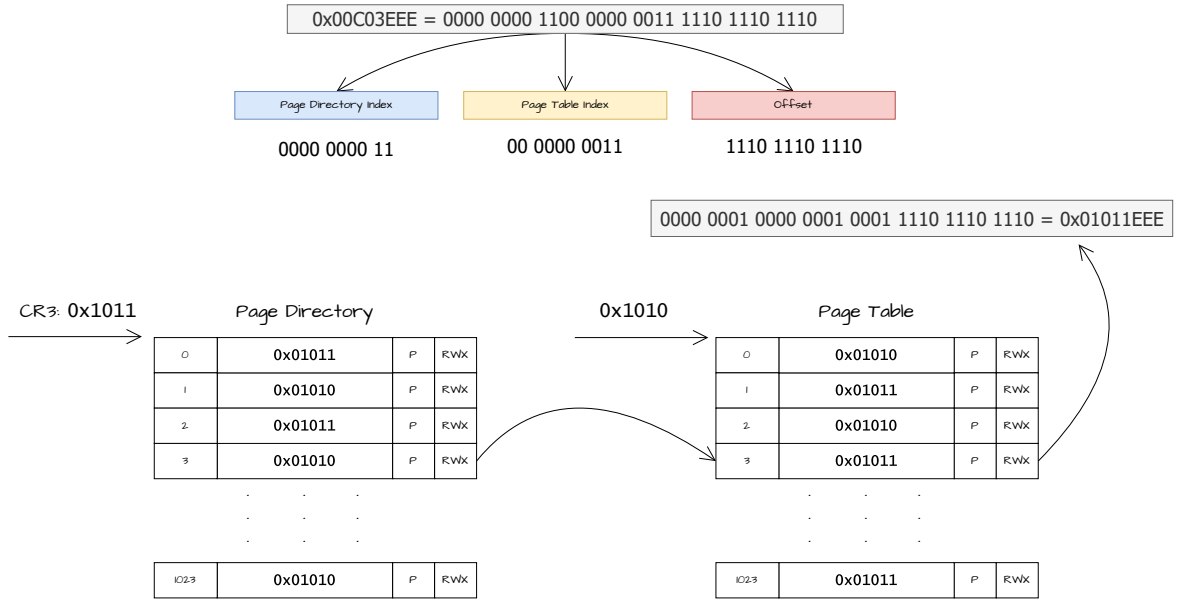

Figura 48: Ejercicio Parcial 2021 - Solución