

Sistemas operativos Resumen

4. La abstracción: el proceso

Un proceso es un programa corriendo, el sistema operativo es el encargado de transformar los bytes que contiene al programa en algo útil.

Para ejecutar muchos programas y que parezca que se hace al tiempo, el sistema operativo crea una ilusión virtualizando la CPU, de tal manera que corre un proceso, lo detiene y luego corre otro, y así. Esto se conoce como **Time sharing**. Para poder hacer esto se necesitan **mecanismos** que son métodos de bajo nivel que implementan una pieza de funcionalidad. Por encima de estos mecanismos están las **políticas**, que son algoritmos que toman algunas decisiones dentro del sistema operativo, como a cuál programa darle tiempo de CPU.

4.1 Lo abstraccion: un proceso

Un proceso se puede resumir al tomar un inventario de los diferentes piezas del sistema a los que accede o afecta durante el transcurso de su ejecución. Para entender qué constituye a un proceso hay que entender su estado de máquina, es decir, qué puede leer o actualizar el programa cuando está corriendo, por ejemplo

- Memoria
- Registros
- Dispositivos de almacenamiento

4.2 API de procesos

Estos deben ser añadidos en alguna interfaz de un sistema operativo

- Crear
- Destruir
- Esperar
- Misc Control
- Estado

4.3 Creación de un proceso (un poco más de detalles)

El sistema operativo debe realizar varias tareas para poder crear un proceso.

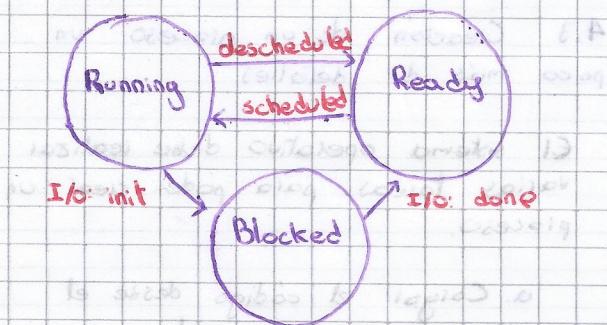
- a. Carga el código desde el almacenamiento o la memoria, y el espacio de direcciones del proceso.
- b. Se dispone de alguna memoria en el stack de ejecución, como variables locales, parámetros y direcciones de retorno.
- c. Se crea el heap de memoria, que se usa para datos dinámicamente ubicados solicitados explícitamente. Sirve para estructuras de datos como listas ligadas o árboles.
- d. Se inicializan cosas relacionadas con la entrada y salida. En Unix cada proceso tiene tres descriptor de archivo: entrada estándar, salida estándar y error, lo que permite tener salidas de la terminal y mostriallas en pantalla.
- e. Ejecuta la función main()

4.4 Estados de los procesos

Los procesos pueden estar uno de tres estados

- Corriendo
- Listo
- Bloqueados

Se pueden dar transiciones en los estados



4.5 Estructuras de datos

Un sistema operativo, al igual que otros programas hace uso de estructuras de datos para su funcionamiento.

Una de las estructuras más simples es la lista de procesos. Es necesario para poder realizarle seguimiento a los programas que corren en el sistema.

Un proceso puede ser descrito por su estado: el contenido de memoria en su espacio de direcciones, el contenido de sus registros de CPU incluyendo el program counter y el stack pointer, y la información acerca de la entrada y salida.

5. API de procesos

Existen varios llamados al sistema que permiten crear y controlar procesos, en UNIX están los siguientes:

5.1 fork()

Crea una copia casi idéntica del proceso, copiando su espacio de direcciones por ejemplo, pero tiene sus propios apuntadores, registros, etcétera.

Cuando se ejecuta un fork(), este retorna 0 al proceso hijo y el PID del hijo al proceso padre. La ejecución es no determinista, es decir, puede que se ejecute primero el hijo o puede que se ejecute el padre, esto es determinado por el Scheduler.

5.2 wait()

Obliga al proceso padre a esperar la ejecución del proceso hijo.

5.3 exec()

Los llamados anteriores permiten mantener copias de los procesos, pero exec() funciona diferente, pues permite ejecutar un programa diferente, sobreescribiendo los espacios de memoria y re-inicializando algunas cosas.

Un ejemplo de programa que usa estos es la consola, por ejemplo bash, que se vale de fork() y exec() para ejecutar programas y eso.

6. Ejecución directa limitada

Para virtualizar la CPU, el sistema operativo ejecuta un proceso por un tiempo y luego lo cambia a otro, esto se conoce como **Time sharing**.

Este tipo de virtualización implica algunos retos, como el rendimiento y el control del sistema.

6.1 Técnica básica: EDL

La idea de esta técnica es correr el programa directamente en la CPU, luego cuando el OS deseé empezar un programa, crea una entrada del proceso en una lista de procesos, carga el programa y lo ejecuta. Sin embargo, con este enfoque surgen un par de problemas: cómo evitar que se ejecutan cosas que no queremos que se ejecuten? y cómo puede hacer el OS para detener el proceso y cambiar la ejecución a otro?

6.2 Problema 3: Operaciones restringidas

La ejecución directa tiene el beneficio de ser rápido, pero existe el peligro de que pueda realizar tareas o consumir recursos que no debe, por ejemplo, un solo proceso podría acceder a todo el sistema de archivos, lo cual no es deseable.

Se introducen entonces dos modos de procesador:

- **Modo usuario:** es un modo restringido en el que al realizar una petición I/O resulta en una excepción.

- **Modo Kernel:** No hay restricciones. Es el modo en el que corre el OS.

Cuando un proceso usualmente quiere realizar una tarea que requiere privilegios, hace una llamada al sistema. Estas llamadas son piezas de funcionalidad que brindan los sistemas operativos.

Para ejecutar una llamada al sistema, se debe realizar una instrucción trap, que se va al Kernel y cambia la ejecución a modo Kernel, cuando termina, se ejecuta un return-from-trap que devuelve el proceso a modo usual.

El hardware debe guardar los registros suficientes del proceso para poder hacer correctamente el return from trap. Para ello guarda cosas como registros, el PC, banderas y eso a algo conocido como **Kernel stack**. Hay un stack por proceso. Estos valores se "rescatan" cuando se hace el return-from-trap.

Para saber qué código ejecutar en un trap, se define una **trap table** cuando se inicia el OS, por ejemplo qué código se debe ejecutar cuando ocurre una interrupción de teclado. El OS le informa al HW dónde están ubicados estos códigos, que los recuerda hasta el próximo reinicio. Cada llamada de sistema se identifica con un **número de sistema call**.

Los pasos de ejecución serían:

1. Crear una entrada en la lista de procesos, asignar memoria, crear stack
2. recuperarregs desde el stack
3. ejecutar el main

6.3 Problema 2: Cambiar entre procesos

Cuando se ejecuta un proceso en la CPU significa que le OS no se está ejecutando, por lo que él solo no puede recuperar el control de la ejecución.

Existen dos enfoques para recuperar el control.

• **Enfoque cooperativo:** El sistema operativo confía en que el proceso le va a retornar el control con algún llamado al sistema, como cuando se lee un archivo, o cuando se genera un error, cuando por ejemplo se divide por cero.

• **Enfoque no cooperativo:** En el anterior, si por ejemplo había un ciclo infinito, tocaba reiniciar. Para poder retomar el control incluso en esos casos se introdujo un **timer interrupt** que se encarga de interrumpir los procesos y devolverle la ejecución al OS cada tantas milisegundos. Iniciar el timer y decidir al HW qué hacer en las interrupciones se realiza en el hardware.

Guardando y restaurando el contexto

Después de recuperar el control, el OS debe decidir si continua ejecutando el proceso o cambiar, esta decisión la toma el **scheduler**.

Si se decide cambiar de proceso, se ejecuta un código llamado **context switch**, que consiste en guardar los registros del proceso, por ejemplo en su Kernel stack, y recuperar los registros del proceso que se va a ejecutar.

El cambio de contexto se da así:

1. se guardan los registros de A, pasamos a modo Kernel (HW)

2. Se llama a `switch()`, se guardan los registros de A en una estructura, se recuperan los de B y nos pasamos al K-stack de B (OS)

3. Se recuperan los registros de B, pasamos a modo usuario y se salta al PC de B (HW)

6.4 Sobre la concurrencia

Cuando el OS está manejando una interrupción y le llega otra, debe saber qué hacer. Algo que hace es deshabilitar las interrupciones.

7. Scheduling: introducción

Se ha tomado el conocimiento de otras disciplinas para crear políticas de schedule en los sistemas operativos.

7.1 Supuestos

Se asumían las siguientes cosas sobre la carga de trabajo:

1. Cada proceso corre el mismo tiempo
2. los procesos llegan al mismo tiempo
3. Cada proceso se ejecuta hasta completarse
4. Solo se usa la CPU
5. Se conoce el tiempo de ejecución

Estos supuestos se irán removiendo de a poquito, ya que no son muy realistas.

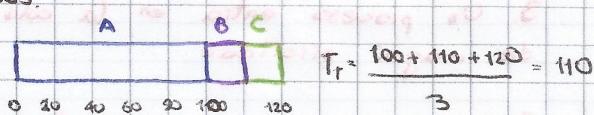
7.2 Métricas de scheduling

Una métrica es algo que usamos para medir otra cosa. Por ahora usaremos la medida de rendimiento **turnaround time** que se define como

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$$

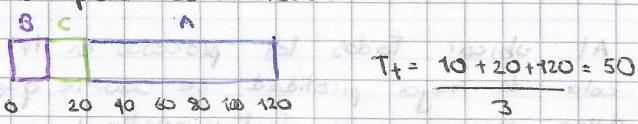
7.3 First In, First Out (FIFO)

Cada proceso se ejecuta en orden de llegada, pero si rompemos el supuesto 1, esto lleva a que el desempeño del scheduler baje, por ejemplo, si el primer proceso es el más largo de todos.

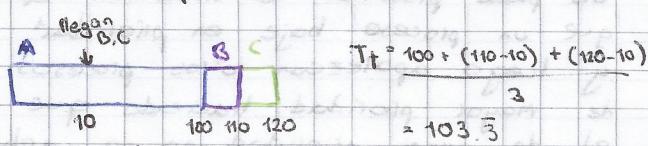


7.4 Shortest Job first (SJF)

Si ejecutamos los procesos más cortos primero, podemos obtener una mejora de desempeño considerable



Sin embargo, si rompemos con el supuesto #2, el scheduler SJF vuelve a tener un pobre desempeño

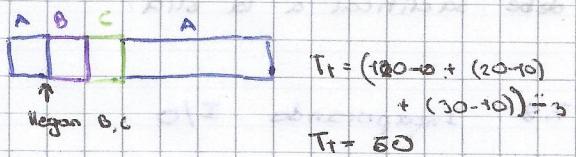


¿Cómo mejorar esto?

7.5 Shortest time to completion first (STCF)

Cada que llegue un nuevo proceso, se evalúa cuál es el que menos se demora en completarse, y si es necesario

se pospone la ejecución actual, esto implica romper con el supuesto 3



7.6 Nueva métrica: tiempo de respuesta

Con la llegada de máquinas de tiempo compartido, se hizo necesaria la introducción de una nueva métrica, el tiempo de respuesta, que se define como

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

Las políticas que vimos se desempeñan muy mal con esta métrica, pues tendríamos que esperar a que se complete un proceso para poder tener una respuesta

7.7 Round Robin

Es una política que consiste en cortar los procesos en pedacitos que tienen como duración un múltiplo de la duración del times interrupt.

Mientras más pequeña sea la duración del slice, mejor se va a compartir el tiempo de respuesta, sin embargo, estos cambios de procesos constantemente implican un costo computacional que, si es muy elevado, puede tirarse el sistema y que deje de responder



Esta política tiene muy mal desempeño en el **turnaround**, pues alarga lo más que puede la ejecución de un proceso

La verdad es que si se desea tener un buen desempeño en una métrica, se debe sacrificar a la otra.

7.3 Incorporando I/O

Aquí romperemos el supuesto 4. El scheduler debe tomar una decisión, pues cuando un proceso realiza una instrucción de I/O, sería un tiempo en el que no está usando la CPU, este tiempo puede ser aprovechado asignándole el CPU a otro proceso.

Si un proceso realiza una I/O, cada pedacito del proceso que se ejecute en CPU se considerará un proceso independiente, para luego realizar un schedule STCF

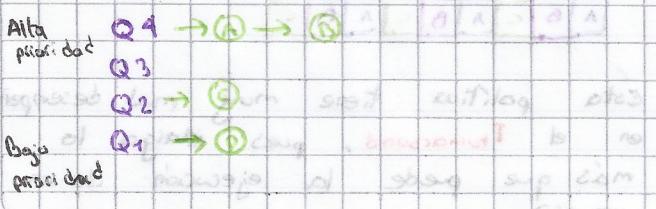


De esta manera producimos un overlap, pero aún nos falta deshacernos del 5º supuesto.

8. Multi-level feedback Queue

Es una técnica de scheduling que busca optimizar el turnaround time y minimizar el response time, pero cómo lograrlo sin conocer información o priori del proceso?

Para construir el scheduler usaremos la estructura **queue**, o cola, cada una con una prioridad asignada



Los procesos que se ejecutan son los que tienen mayor prioridad, si hay dos procesos con la misma prioridad, se ejecutan como RR

1. Si Prioridad(A) > Prioridad(B), se ejecuta A

2. Si prioridad(A) = prioridad(B) Se ejecutan A y B en RR

La prioridad de un proceso puede cambiar a lo largo de su ejecución, además puede ser muy interactivo o demandar mucha CPU. La primera aproximación es:

3. Un proceso entra en la cola de mayor prioridad

4.a Si un proceso se ejecuta en todo su pedacito de tiempo, se reduce su prioridad.

4.b Si un proceso suelta la CPU, se mantiene en el mismo nivel de prioridad.

Al ubicar todos los procesos en la cola de mayor prioridad se asume que duran poco, y si efectivamente lo hacen, terminan rápido, si no, descienden en la cola de prioridad.

Se puede presentar la situación de que un proceso baje en prioridad y luego aparezcan otros procesos de mayor prioridad, haciendo que el de menor prioridad no se vea a ejecutar. Esto se solventa con la regla 5.

5 Despues de un tiempo S, se suben todos los procesos a la cola de mayor prioridad.

Así, se puede garantizar que todos los procesos tengan un tiempo de

ejecución. Si es un palindromo conocido como uno-dos constante, que es una de esas cosas que uno suele no saber cuál es el valor adecuado, así que uno traejui con los valores por defecto.

La regla #4 tiene un problema, y es que puede que un programa se adueñe de la CPU al realizar una operación de I/O en cada una de sus piezas de tiempo de ejecución, haciendo que nunca se reduzca su prioridad. Para evitar eso, se modifica la regla 4 a:

4. Apenas un proceso haya usado su tiempo asignado en un nivel dado, se reduce su prioridad, sin importar cuántas veces haya cedido la CPU

Siguen existiendo problemas para parametrizar a los schedulers. Hay muchas implementaciones diferentes y cosas así que pueden servir bien en determinados contextos.

9. Proportional Share

Intenta garantizar que cada proceso obtenga un porcentaje del tiempo de CPU

Tickets representan la proporción

Los tickets representan los recursos que se deben compartir y que el proceso debe recibir. El porcentaje de tickets representa el porcentaje de recursos que debe recibir.

Diganos que A tiene 75% tickets y B tiene 25, significa que A tiene que recibir el 75% de tiempo de CPU

El scheduler por lotería funciona de manera probabilística, generando un número al

azar. A pesar de que no siempre se brinde el mismo tiempo de probabilidad que de tiempo de ejecución, cuando los procesos son más largos, el scheduler tiende a compartir el mismo tiempo que porcentaje de ejecución.

Mecanismos de ticket

Existen algunos métodos para manejar los tickets, como:

- **Ticket currency:** Un usuario puede distribuir sus tickets en sus procesos
- **Ticket transfer:** Un proceso puede darle sus tickets a otro que los pide necesitar más, como en una arquitectura cliente-servidor
- **Inflar los tickets:** Es una técnica que permite obtener más recursos. Es útil en ambientes donde confían entre ellos.

La implementación es fácil, se pone cada proceso con su respectiva cantidad de tickets en una lista ordenada según la cantidad de tickets, se recorren con un apuntador y un contador, la idea es que sea lo más justo posible.

El problema difícil es cómo asignar los tickets

Stride scheduling

Se crea una cantidad llamada stride que es el inverso del número de tickets.

Se divide un número grande arbitrario entre el stride, se define un valor conocido como pass, que empieza en cero y funciona como un contador. El procesamiento se le asigna al de menor

pass y se le suma el stride, pero tiene el inconveniente de no tener un statement global, lo que dificulta "ingresar" nuevos procesos.

13. La abstracción: el espacio de direcciones

Los primeros sistemas no brindaban muchas abstracciones, sólo tenían un proceso que disponía de toda la memoria.

Luego llegó la era de la multiprogramación que buscaba optimizar el tiempo de procesamiento al poner un proceso en la CPU mientras otro realiza una tarea de I/O.

Darle toda la memoria a un proceso es muy lento, y tener varios procesos en memoria puede ser difícil de manejar concurrentemente, hay que proteger a los procesos.

13.3 El espacio de direcciones

Para manejar la memoria más fácil se crea la abstracción del espacio de direcciones. Este contiene todo el estado de memoria de un proceso, como el código fuente (instrucciones), stack que sirve para saber en qué parte de la cadena de ejecución está y un heap para ubicar la memoria pedida manualmente. El heap crece hacia abajo, mientras que el stack lo hace hacia arriba.

Este espacio se logra virtualizar haciendo que cada proceso empieza en la dirección cero de memoria, cuando en realidad tiene otra dirección física, creyendo que toda la memoria del sistema está disponible para ese proceso.

13.4 Objetivos

Los objetivos de virtualizar la memoria son

- Transparencia: Que el programa no sepa que su memoria está siendo virtualizada.
- Eficiencia: Virtualizar de la manera más eficiente posible, en tiempo y espacio.
- Protección: Que un proceso no pueda acceder al espacio de direcciones de otro proceso.

14. API de memoria

En un programa en C existen dos tipos de memoria

- Stack: Es manejado automáticamente por el compilador, se ejecuta cuando se hacen llamados o métodos.
- Heap: Es manipulado por el usuario cada que se hace un llamado como malloc.

malloc se encarga de crear un espacio de memoria del tamaño solicitado y devolver un apuntador a dicho espacio. La memoria pedida se puede liberar llamando al método free.

14.4 Errores comunes

Suelen deberse al manejo de la memoria con malloc y free

- No ubicar memoria
- No ubicar suficiente memoria
- No inicializar la memoria ubicada

- No libera la memoria
- Libera la memoria antes de terminar de usarla
- Libera memoria repetidamente
- Llama mal a free

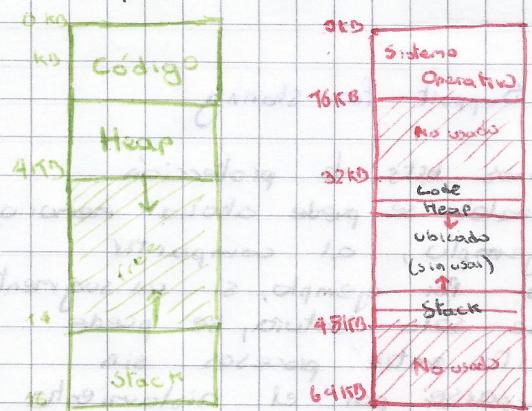
Estos métodos no son llamados al sistema, sino que son llamados a librería.

15. Mecanismo: traducción de dirección

Una técnica genérica es la traducción de direcciones, donde el hardware se encarga de traducir la dirección de memoria virtual proporcionada por el proceso a su dirección física real, donde el OS maneja los espacios libres.

Asumiremos que la memoria es contigua y menor una dirección de memoria que lo memoria.

Debido a la ilusión de la virtualización, el proceso cree que su memoria emplea en la dirección 0, pero físicamente puede que no sea así.



15.3 Relocación dinámica

La primera técnica que revisaremos la conocemos como relocación dinámica, conocida también como base y límite.

La base y el límite son registros de cada CPU, con esta pareja de

registros se puede ubicar el proceso en cualquier parte de la memoria física.

Cuando se lanza un programa, el sistema operativo decide dónde ubicar el programa en memoria y define los registros base y límite. Todos las respectivas direcciones del proceso deben estar entre estos dos registros. De este manera, podemos calcular las direcciones como

$$\text{dirección} = \text{dirección virtual} + \text{base}$$

El registro límite nos permite validar que las direcciones sean válidas. Los registros son HW.

Para mantener control sobre los espacios libres, el sistema operativo

15.4 Resumen del soporte de HW

Hay un bit que indica si estamos en modo usuario o modo kernel.

Los registros base y límite hacen parte de la unidad de manejo de memoria (MMU), modificables separadamente.

Si se intenta acceder a una posición indebida, se debe lanzar una excepción.

15.5 Issues de los OS

Manejar la memoria libio.

Manejar cuando termina un proceso.

Manejar los cambios de contexto.

Manejar las excepciones.

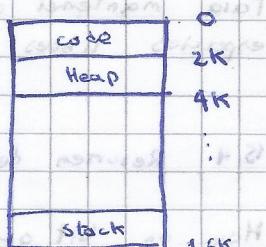
16. Segmentación

La aproximación de los registros base y límite tiene el inconveniente de que deja mucha memoria desperdiciada, para ello se generaliza con el concepto de segmentación.

16.1 Generalized base/bounds

En vez de tener un par de registros base y límite por espacio de dirección se pasan a tener muchos un par de registros por **segmento**, lo que permite ubicar a cada segmento (código, stack y Heap) en una posición diferente de memoria física, sin necesidad de ser contiguas.

Para realizar una traducción del código y del Heap hay que restar sus offsets. Así, para obtener la dirección física de la dirección virtual, debemos tomarla y restarle el offset y luego sumar el resultado al registro base.



16.2 A qué segmento nos referimos?

Una aproximación consiste en tomar dos bits de toda la dirección virtual y usarlos para identificar a qué segmento pertenece una dirección.

1010010010

segment offset

se puede saber si una dirección es válida comparándola con el offset. Aunque esto reduce el tamaño máximo direccionable.

Otras aproximaciones existen, como la implícita, donde el hardware determina de qué segmento pertenece, por ejemplo, si la instrucción dirección fue generada por el PC, seguramente esté en el segmento del código.

16.3 Acerca del stack

Para manejar el stack necesitamos de otro bit que nos diga si el segmento crece hacia arriba o hacia abajo. Para hacer la traducción tomamos la dirección virtual y le restamos el tamaño máximo que pueda tener un segmento, para luego sumarse al registro base.

$$DV: 3 KIB$$

$$MS: 4 KIB$$

$$\text{Base: } 28 \text{ KIB}$$

$$(3 - 4) KIB = -1 KIB$$
$$\Rightarrow (28 - 1) KIB = 27 KIB \cdot PA$$

16.4 Support for sharing

Con unos bits de protección adicionales se puede aislar memoria, por ejemplo, al compartir código. Por ejemplo, si un segmento es de sólo lectura, se puede compartir entre procesos sin preocuparse por el aislamiento.

Los registros de segmentos se verían así:

Segment	Base	Size (max 4k)	Protect?	Protection
Code ₀₀	32K	2K	1	RX
Heap ₀₁	34K	3K	1	RW
Stack ₁₁	28K	2K	0	RW

18. Paginación

Tener segmentos de diferente tamaño conduce a varios problemas, como la segmentación, por lo que mejor pasamos a una aproximación que separe la memoria en espacios físicos. El espacio de direcciones virtual lo dividimos en páginas, que se almacenan en la memoria física como marcos de página. Este enfoque tiene como ventaja principal la flexibilidad, pues no asumimos cosas sobre cómo un proceso usa el espacio de direcciones; otra es la simplicidad, pues el espacio libre se maneja bien con una lista que "contenga" las páginas que estén libres.

Para saber dónde está cada página, el sistema operativo almacena las direcciones físicas en una Tabla de página por proceso, que se encarga de guardar las traducciones para cada página virtual.

Ejemplo Dado un espacio de direcciones de 64 bytes y un tamaño de página, el formato de la dirección es el siguiente

$$64/16 = 4 \text{ páginas}$$

$$\log_2(64) = 6 \text{ bits}$$

$$\text{VPN} = 2 \text{ bits} \text{ (para dirigir las páginas)}$$

VPN	offset
-----	--------

El VPN representa el índice en la tabla de página, y contiene el valor de la dirección física, que se junta con el offset para traducir la dirección completa.

18.2 Dónde se guardan las tablas de página?

Los tablos de página pueden llegar a ser muy, muy grandes, por lo que no es posible almacenarlos en la MMU, por ahora asumimos que está en la

memoria del sistema operativo.

18.3 ¿Qué hay en la entrada de tabla de página?

La tabla de página es sólo una estructura de datos que mapea una dirección virtual a una física. Su forma más simple es una tabla de página lineal. El sistema operativo indexa el arreglo por el número de la página virtual (VPN) y obtiene el valor de la entrada de la tabla de página para así obtener el PFN.

Además de los bits del PFN hay muchos otros bits que nos sirven como mecanismos de control, como los siguientes:

- Bit de validez
- bits de protección
- bit sucio, cuando se ha modificado una página en memoria
- bit de presencia, si está en memoria física o en swap
- bit de referencia, para saber si se ha accedido a una página, útil para las políticas de reemplazo

18.4 Paginación es muy lento

Tal y como lo conocemos, la paginación puede ser muy lento, p.ej por ejemplo, para cargar un registro se debe

- Hacer la traducción de VPN a PFN
- Obtener el PTE de la tabla del proceso
- Traducirlo
- Cargarlo de la memoria física

Basicamente estamos haciendo dos accesos a memoria (PTE y load), lo que es muy costoso, reduciendo el desempeño a incluso menos de la mitad.

19. Página: Traducciones más rápidas (TLB)

La paginación entra en el problema de la fragmentación externa, pero tiene problemas de rendimiento y de espacio de memoria.

Para mejorar el rendimiento, introducimos un elemento de hardware llamado **Translation lookaside buffer (TLB)** ubicado en la MMU (memory management unit) que funciona como una cache de traducción de direcciones. Cuando se hace una referencia de memoria virtual, el hardware verifica si la traducción existe en la TLB, y si efectivamente se encuentra allí, realiza la traducción sin tener que hacer un acceso a memoria física.

19.1 Algoritmo básico de TLB

- Tomamos la VPN de la dirección virtual.
- Verificamos si la TLB contiene la traducción de la VPN.

↳ Si la traducción existe ocurre un bit y se crea la dirección concatenando el PFN de la TLB con el offset.

↳ Si la traducción no existe ocurre un miss, se carga la dirección a la TLB y se intenta la instrucción.

La TLB se aprovecha de los principios de **localidad espacial** y **localidad temporal** para mejorar el rendimiento de los procesos.

19.3 ¿Quién maneja los miss de la TLB?

Algunas aproximaciones las manejan directamente en el hardware, otras más modernas lo que hacen es que en un miss se lanza una excepción que es manejada por

el trap handler en modo Kernel. Esto lo hace el sistema operativo. En este caso, el return from trap debe retornar a la instrucción que causó la excepción.

El sistema operativo también debe tener cuidado de no crear una cadena infinita de misses, por ejemplo, dejando al handler en la memoria física.

19.4 ¿Qué contiene la TLB?

Una TLB típica es completamente **asociativa**, lo que significa que una traducción puede estar en cualquier parte de la caché, por eso es que debe ser pequeña.

El formato puede ser como el siguiente:

VPN | PFN | Otros bits

En los otros bits puede haber cosas como

- bit de validez
- bit de protección
- identificador de espacio de direcciones
- bit sucio
-

Otras otras

19.5 Cambios de contexto

Al realizar cambios de contexto se pueden presentar algunos retos, por ejemplo, puede que dos procesos diferentes tengan generada la misma memoria virtual, por lo que la TLB no puede distinguir cuál entrada tiene que traducir.

V	P	N	P	F	N	valid	prot	ASID
10		100	1			1	1w x	1
-		-	0			0	-	-
10		170	1			1	1w x	2
-		-	0			0	-	-

El problema es que se incurre en fragmentación interna, al tener espacios muy grandes asignados para muy pocas variables.

Para solventar esto se puede tratar de borrar la caché cada que se haga un cambio de contexto, pero eso también implicaría incursión en misses. Otra opción es agregar más hardware, de tal manera que el campo **ASID** nos permita identificar el proceso.

Otra situación que sí se puede presentar es que dos direcciones virtuales de procesos diferentes apunten a la misma página física, cuando comparten código, por ejemplo.

La TLB puede fallar muy frecuentemente si el número de páginas a las que se accede en un pequeño periodo de tiempo es mayor que el tamaño de la caché.

20. Página: Tablas más pequeñas

Las tablas de página lineales usualmente son muy grandes, y cuando tenemos muchos procesos puede que terminemos destinando mucha memoria sólo a las tablas de página.

20.1 Páginas más grandes

Una solución simple consiste en hacer las páginas más grandes, así reducimos el número de páginas que hay que direccionar.

Ejemplo: Si tenemos direcciones de 32 bits y pasamos de páginas de 4KB a 16KB entonces el tamaño de la tabla pasa a ser 1MB en vez de 4MB

20.2 Página y segmentos

Con la paginación tal y como la conocemos, es muy probable que despedazemos muchos espacios de la tabla de página porque simplemente no se están usando.

Una opción es manejar la segmentación nuevamente, usando el bit base para apuntar en qué dirección física está almacenado cada segmento lógico.

Para dirigirnos a los segmentos lógicos, tomamos los bits que necesitamos del campo de la VPN ASI, si tenemos 3 segmentos (Heap, Stack y código) necesitamos dos bits para dirigirlos.

Seg	VPN	Offset
-----	-----	--------

Para cada uno de los segmentos, tenemos parejas de registros para la base y el límite. El registro base contiene entonces la dirección en memoria física de una tabla de página lineal para ese segmento, por lo que cada proceso contaría con tres tablas de página.

Cuando ocurre un miss, el HW usa los bits del segmento para determinar que par de registros usar.

Con la segmentación volvemos a tener problemas de flexibilidad y de fragmentación, por lo que hay otras aproximaciones para tener páginas más pequeñas.

20.3 Tablas de páginas multnivel

La idea es simple, partit el la tabla de página en espacios de página del mismo tamaño, si la página de la entrada de la tabla de páginas es inválida, no se ubica esa página.

Para saber si una página es válida o no creamos una nueva estructura llamada **directorio de página**, que se encarga de decir donde se encuentra una página de la tabla de páginas o si toda la página de la tabla de páginas no contiene páginas válidas. Así el directorio lo que nos permite es liberarnos de espacio que en la tabla de página linear no aprovecharíamos.

Una entrada del directorio de páginas contiene la dirección de una página de la tabla de páginas y está conformada por un bit de validez y la dirección del marco de página (PFN). Cuando una entrada de la tabla de páginas es válida significa que por lo menos uno de los páginas de la tabla de página.

Una ventaja de las tablas de página multnivel es que usamos la memoria de manera proporcional, otra es que se puede manejar de manera muy fácil la memoria.

Se puede presentar el caso de que necesitemos más niveles cuando el directorio de tabla de página está muy grande, simplemente dividimos el directorio según los niveles que necesitemos.

Todos estos accesos sólo se hacen si hay un miss, sino, la dirección se reforma directamente desde la TLB.

20.4 Páginas invertidas

Consiste en tener muchas tablas de página (una por proceso del sistema) donde cada tabla de páginas tiene una entrada para cada página física del sistema. Así, encontrar la entrada correcta es buscar en toda la estructura de datos, lo que sería costoso, por lo que se suele usar una tabla de hash.

20.5 Cambiando páginas al disco

Las tablas de página pueden seguir haciendo muy grandes, por lo que con algunas máquinas, se pueden sacar páginas de la **Memoria virtual del Kernel** al disco, usando una operación conocida como swap.

21. Mecanismos

Hasta ahora hemos asumido que el espacio virtual cabe en el espacio físico, pero eso no es realista. Dejaremos esa asunción de lado agregando un nivel adicional en la jerarquía de memoria. Así, mantendremos la ilusión de un gran espacio de memoria.

21.1 Espacio de Swap.