Xv6: Sistema Operativo Educacional

Tomás Juárez
Cátedra de Sistemas Operativos, Universidad Nacional del Centro de la Provincia de Buenos Aires.

Introducción

El sistema operativo Xv6 fue llevado a cabo de una manera muy simplificada dado que es de propósito educacional, desarrollado en el el Instituto de Tecnología de Massachusetts (MIT) para su curso de sistemas operativos. La implementación del sistema está basada en la versión 6 de Unix, motivo por el cual se lo llamó de tal forma.

En el presente trabajo se exploran diferentes funcionalidades del sistema operativo a modo de extenderlas o cambiar su comportamiento. Para ello es menester comprender las distintas partes del sistema, teniendo en cuenta que esto implica conocer ciertos aspectos de la teoría de lenguajes de programación y de arquitectura de computadores.

En los siguientes apartados se discutirán cada uno de los ejercicios solicitados por la cátedra, dando primero una reducida introducción teórica para luego especificar su implementación en Xv6 y posteriormente las soluciones propuestas por el alumno.

Arquitectura x86

Los procesadores de la arquitectura x86 poseen un conjunto de *registros de propósitos generales* de 32 bits para ser utilizados en la programación en assembler. Esto se debe a que los procesos deben ser lo más rápidos posibles, y acceder a memoria cada vez que estos requieren guardar o referenciar cierta información no parece ser una buena opción. Incluso utilizando una jerarquía de memorias con cachés en varios niveles[1], en el peor de los casos se debe llegar a la memoria RAM, lo cual supone una serie de fallos cuya penalización aumenta el tiempo medio de acceso a memoria, impactando directamente en el CPI del procesador. Asimismo, existen los llamados *registros de segmentación*, cuyo fin es retener algunas direcciones de memoria útiles, y su variación podría alterar la ejecución normal de un programa. Esta arquitectura también posee registros llamados *índices* o *punteros*, los cuales comienzan con la letra 'E', y sólo pueden modificarse en modo protegido del procesador, esto es, cuando el sistema operativo se encuentra en *modo kernel*. Finalmente, existe un registro especial llamado EFLAGS el cual contiene una etiqueta a modo de identificar el estado del procesador. Esto es útil por ejemplo, cuando se quiere obtener información luego de ejecutar una instrucción aritmética; el resultado de una operación puede resultar en cero o negativo. Estos registros, entre otros, son los que se corresponden con los datos del *trapframe*, estructura que se discutirá más adelante.

Si bien muchos de los registros son de uso general, EAX y EDX suele utilizarse como registro acumulador para acceso de puertos de entrada/salida o mapeo de interrupciones. Por otro lado, EBX suele ser un puntero base de acceso a memoria y ECX es utilizado como registro contados para realizar operaciones de *shift* o como contador de loops, entre otros. Los registros de segmentación contienen datos específicos: CS contiene el segmento de código en el cual se ejecuta el programa actual, DS contiene el segmento de datos al cual accede el programa actual, ES, FS y GS son registros de segmentación extra para direccionar direcciones de memorias lejanas y el último registro llamado

[1] Los procesadores actuales suelen incluir hasta 3 niveles de caché (L1,L2,L3) en la jerarquía de memorias antes de la memoria RAM. De L1 a L3 son cada vez más lentas, pero con mayor capacidad.

SS sirve para contener el segmento de stack que usa el programa actual. Por último, los llamados punteros o índices: EDI (destination index register) y ESI (source index register) son usados para copia de arreglos, strings y direccionar posiciones de memoria lejanas en conjunto con el registro de segmentación ES. El índice EBP (stack base pointer register) apunta a la dirección de memoria base del stack, ESP (stack pointer register) apunta al tope del stack y EIP (index pointer) contiene el offset de la siguiente instrucción.

Lenguajes de Programación

A modo de explicar el funcionamiento del sistema operativo Xv6, es necesario comprender algunos conceptos base sobre los lenguajes de programación ya que estos tienen una relación intrínseca con varios elementos de este sistema operativo.

Pila de Ejecución

Retomando los registros de la arquitectura x86, al comienzo de la pila se tiene el ejecutable del programa, sobre el se encuentran las variables estáticas del mismo y por encima se encuentran los distintos registros de activación. El último registro de activación tendrá dos punteros: su base estará apuntada por el registro del procesador EBP y su tope por ESP. Los llamados registros de activación contienen la información necesaria para que una unidad del programa pueda ser ejecutada [figura 1].

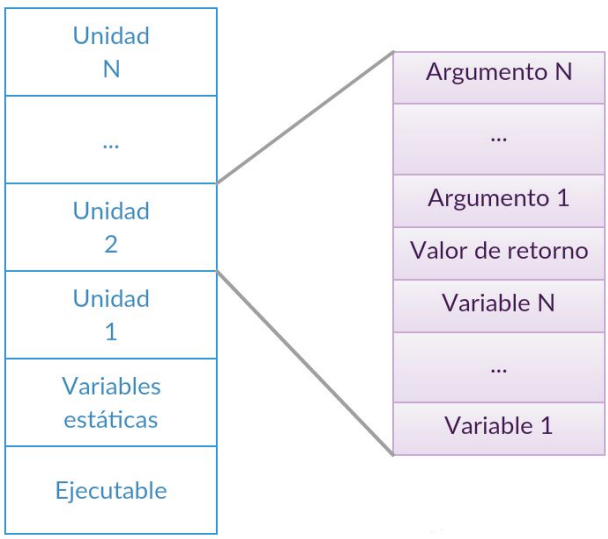

Figura 1: *registros de activación de un lenguaje genérico.*

Supongamos una secuencia de llamados en un programa determinado, por ejemplo, *main->foo->bar*. Estas son unidades del lenguaje, y pueden definir información que se encontrará en su registro de activación; pueden ser funciones, procedimientos, expresiones lambda, entre otros. El registro de activación de *foo* es apilado en la pila de ejecución, entonces EBP apunta a su base y ESP al tope. Luego, *foo* llama a *bar*, y el puntero EBP apunta a la base del nuevo registro de activación, el de *bar* (esto es, mover el valor de ESP al registro EBP) y luego, se cambia la dirección de memoria apuntada por ESP por el tope del registro de activación de *bar* [figura 2].

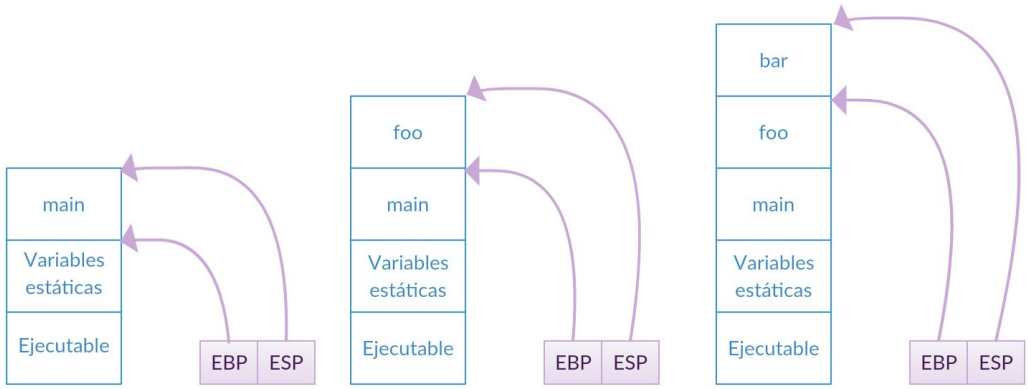

Figura 2**:** *proceso de apilado con registros del procesador.*

Cabe destacar que estas representaciones son simples esquemas, puesto que los procesadores de lenguajes de programación incluyen muchos otros datos, como punteros de cadenas estática y dinámica, entre otros, y varían en cada implementación.

Variables

Las variables pueden clasificarse según su tipo o su modo de almacenamiento. En este trabajo, importa su almacenamiento, puesto que se intentará explicar cómo Xv6 obtiene distintos parámetros de llamadas a funciones y define diferentes estructuras.

Las variables estáticas se encuentran en una dirección fija de la pila de ejecución y poseen un tamaño fijo. Las variables semi-estáticas y semi-dinámicas se encuentran en los registros de activación de una unidad en particular (la que define las variables). Sin embargo, las semi-estáticas tienen un tamaño fijo y las semi-dinámicas sólo varían su tamaño en los distintos registros de activación, pero una vez definida en uno de ellos no varía su tamaño. Por otro lado, las variables dinámicas pueden cambiar de tamaño con cualquier instrucción en cualquier momento, con lo cual es necesario almacenarlas en una estructura llamada *heap*. Cualquier elemento de la pila de ejecución tiene espacios de memoria asignados por el sistema operativo para cada proceso. Tanto los registros de activación como el heap se encuentran en la misma porción de memoria y la pila de ejecución se encuentra en la base de este espacio de memoria. El heap se encuentra en el tope de este espacio de memoria y crece en sentido contrario a los registros de activación, y de igual manera ocurre para los registros de activación, pero en dirección contraria. Estos últimos crecen de una manera controlada y cada vez que termina la ejecución de una unidad, se desapilan y se reacomodan los registros ESP y EBP del procesador. En el caso del heap se dan dos posibilidades: o bien la variable se declaró en forma anómina, con lo cual el programador debe gestionar esa variable en el heap manualmente y eventualmente puede considerarse como basura (punteros colgados, por ejemplo), o bien que la variable tenga un identificador asignado. El heap crece de manera descontrolada por su propia desorganización; consecuencia inherente de tal estructura. En el peor de los casos (un caso extremo), si los registros de activación crecen demasiado y/o el heap crece mucho (llamadas recursivas sin corte aparente con muchas definiciones de variables dinámicas), estos se “chocan”. [figura 3].

Figura 3: *ubicación esquemática de heap y stack de usuario en un espacio de memoria asignado a un proceso por el sistema operativo.*

Proceso de Compilación de C

En este trabajo en particular se utiliza el lenguaje de programación C, el cual es un lenguaje compilado, base de la mayoría de los sistemas. Dentro de los lenguajes de alto nivel, el de más bajo nivel es C, con lo cual es muy eficiente y es esa la razón por la cual los sistemas operativos se construyen con este lenguaje en particular.

La primera etapa es llamada *etapa de preprocesamiento* y es realizada para reemplazar todas directivas encontradas en el archivo en cuestión. Estas directivas comienzan con el caracter '#' y las hay de varios tipos; condicionales, inclusión de código externo y definición de macros. En el caso de la inclusión de bibliotecas, se utiliza #include para que el preprocesador pueda reemplazar esa línea por la cabecera del archivo a incluir, es decir, todas las definiciones de sus funciones, estructuras e incluso otras macro. En el caso de #define, se reemplazará en cada nombre de la macro declarada según su valor inmediato. La segunda etapa es llamada *etapa de compilación* en donde se verifica la sintaxis y semántica del código, utilizando un analizador sintáctico-léxico con estructuras y una tabla de símbolos; para la semántica se evalúan diferentes situaciones, tales como el alcance y coherencia entre tipos de variables, funciones, etcétera. Una vez pasada esta prueba, se procede a generar código intermedio y luego se da una salida en assembler. Una tercer etapa convierte la salida previa de assembler a código objeto, llamada *etapa assembly* en donde básicamente se tienen en cuenta ciertos parámetros de la arquitectura en la cual va a correr el programa y estará escrito en binario. Por último, la *etapa de linking* en donde se unen distintos archivos de código objeto (que se obtienen a partir del archivo principal, y luego por cada una de sus inclusiones, recursivamente) para formar un único archivo ejecutable, el que luego se convertirá en proceso.

Interrupciones

Introducción

Los sistemas operativos modernos necesitan mantener el sistema en un estado seguro. Ese es el motivo principal por el cual todos los recursos requeridos por un proceso de usuario son solicitados

mediante excepciones. Si el sistema operativo no funcionase de esta manera, entonces un programa cualquiera podría hacer lo que quisiera con nuestra computadora, incluso dañar el hardware o interferir en el funcionamiento del mismo sistema operativo, como por ejemplo quedarse con el CPU durante un tiempo indeterminado, lo cual no es deseado.

El soporte de excepciones del procesador no sólo tiene como fin cumplir con las necesidades del sistema operativo, sino que surge como una medida para atender eventos de entrada y salida en general, y como un método de handling de los errores que ocurren en al ejecutar un programa. El procesador trata excepciones; dentro de esas excepciones se encuentran las llamadas interrupciones. Las excepciones son síncronas, es decir que ocurren en una etapa específica del pipeline del procesador y es causado por una instrucción en particular; como por ejemplo una división por cero, la cual es arrojada en la etapa de ejecución en alguna unidad funcional. Por otro lado, las interrupciones son asíncronas y pueden deberse tanto a fallas en el hardware de la computadora, o por un llamado de atención de un componente de E/S como una alternativa al *polling*, por cuestiones de eficiencia. Cuando un componente de E/S requiere la atención del procesador, este genera una interrupción y almacena en un registro la causa de la misma, para que el CPU entienda que se requiere de su atención y pueda recuperar el motivo del llamado de atención en tal registro. De igual manera, ambas modifican el curso normal de un programa [1].

Dentro de las excepciones se encuentran los llamados *traps*. Como se mencionó previamente, muchos autores no dejan claro la diferencia entre excepciones e interrupciones. Los traps son en realidad excepciones, pero se hace referencia a ellos como *interrupciones definidas por el usuario*. De aquí en más se hará referencia a estas como *traps* o *interrupciones* (a pesar de que sean excepciones), indistintamente, con el fin de evitar confusiones.

En las arquitecturas x86, para lanzar un trap se debe utilizar la instrucción especial *INT N* donde *N* es un número que referencia a la interrupción vectorizada por el sistema operativo. En DOS, por ejemplo, para abrir un archivo hay que lanzar la interrupción *INT 21h* con el seteo de los parámetros correctos en los registros del procesador[2]. Surge entonces la necesidad de una estructura para almacenar información útil de los traps, cada vez que estos ocurren. Dicha estructura es llamada *trapframe* y se discutirá luego.

Interrupciones en Xv6

En este sistema, el archivo *traps.h* define una serie de macros para identificar las interrupciones. Por ejemplo, al modificar o agregar código (de forma descuidada) generalmente termina en una interrupción número 14 o más bien *page fault*, por un mal acceso a memoria, al acceder a una posición a la cual no tenemos permiso, o al hacer referencia a una posición de memoria inexistente.

Una vez que una interrupción es arrojada, el hardware necesita realizar un poco de trabajo: primero se debe incrementar el nivel de privilegio actual o CPL por sus siglas en inglés. Una vez realizado esto, se delega el control a un vector de traps, inicializado por el sistema operativo. Este vector es llamado IDT (*Interrupt Descriptor Table*) y cada elemento del vector es llamado *Trap Gate*. Este vector es inicializado por el sistema operativo al bootear según el archivo *vector.S*. Este código assembler es tedioso, con lo cual se utiliza un script perl para generarlo, el cual se halla en *vectors.pl*. Una vez creado el vector, es necesario setear los trap gates, tarea realizada por *tvinit(void)* del archivo *trap.c*. Este instancia una estructura llamada *gatedesc* (gate descriptor) y la asigna a cada posición del

[2] Interrupción int 21h: http://ict.udlap.mx/people/oleg/docencia/ASSEMBLER/asm_interrup_21.html

vector. Tal estructura se encuentra en *mmu.h,* y allí mismo se halla la función *SETGATE* para instanciar la estructura según algunos parámetros pasados.

En el archivo assembler *trapasm.S* se crea el trapframe para la interrupción y se setean los datos según los registros del procesador, nombrados en secciones anteriores. [figura 4].

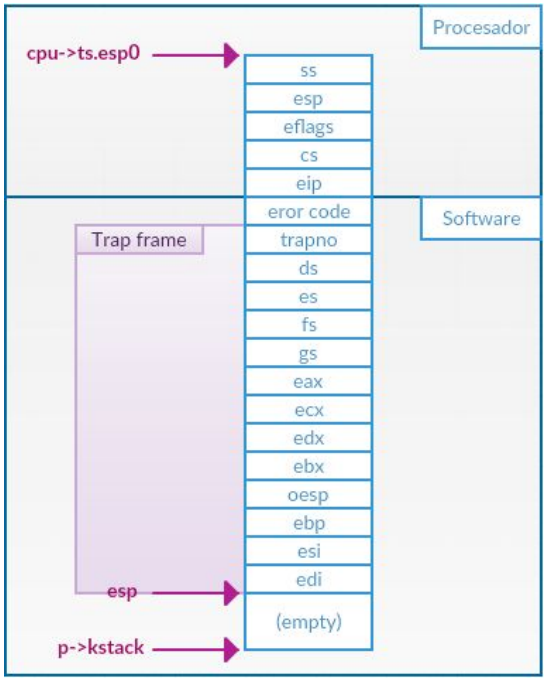

Figura 4: *esquema del trapframe dentro del kernel stack.*

Un caso particular de los traps es la interrupción número 64 y es utilizada pura y exclusivamente para llamadas al sistema, la cuales se discutirán luego.

Llamadas al sistema

Introducción

Los procesadores modernos tienen un juego de instrucciones bien definido para ejecutar en modo usuario y una extensión de ellas para ejecución en modo kernel, para ayudar a mantener la seguridad del sistema operativo. De hecho, el procesador es quien realiza el cambio de modo privilegiado a modo usuario y viceversa, el cual consiste en intercambiar un campo de dos bits que representan el *nivel de privilegio actual* o *Current Privilege Level* (CPL). Este campo se encuentra dentro del registro CS del procesador, y el nivel con mayor privilegio es representado con un 0, pero a medida que el número es más grande, este tendrá un menor privilegio. En Linux se utilizan sólo los niveles 0 y 3, los cuales representan al modo kernel y el modo usuario, respectivamente.

Todos los procesos deberían poder comunicarse con el núcleo del sistema operativo, puesto que es el encargado de administrar los recursos del sistema, por lo cual es necesaria una interfaz a los servicios y funcionalidades que el sistema operativo nos provee. Esta interfaz está dada por las llamadas al sistema, las cuales son simples funciones con la particularidad de ser implementadas en el núcleo del sistema operativo. Esto implica que el procesador tenga que cambiar constantemente de modo usuario a modo privilegiado en cada llamada al sistema, ya que el usuario invoca a tal llamada al sistema, pero es el sistema operativo el que tiene que ejecutar la lógica de la misma, la cual se encuentra en el kernel y no puede ser ejecutada directamente por el usuario [figura 5].

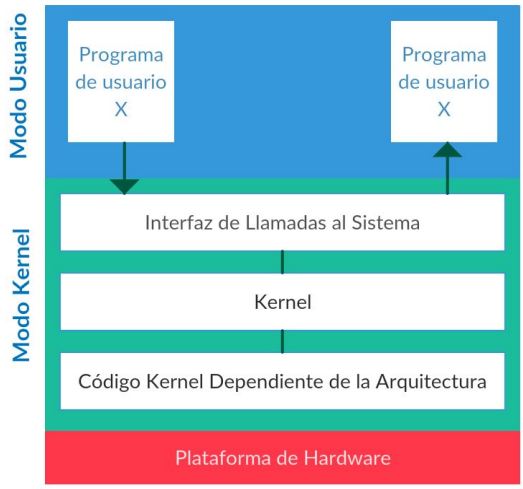

Figura 5: *Capas de un sistema operativo, a grandes rasgos.*

Las llamadas al sistema son iniciadas por una interrupción de software, específicamente por una instrucción assembler. El número de interrupción se ve correspondido con un índice del arreglo de llamadas al sistema, interno al kernel del sistema operativo el cual contiene punteros a las funciones que las implementan. Básicamente es una tabla de punteros a funciones. Ahora bien, algunas llamadas al sistema requieren parámetros para realizar sus operaciones. Existen tres formas de pasar parámetros a las llamadas al sistema:

1. Pasar parámetros por medio de registros que provee el procesador.
2. Cuando existen más parámetros que registros, los parámetros pueden ser almacenados en un bloque y su dirección de memoria puede ser pasada como un parámetro al registro (puntero).
3. Los parámetros pueden ser insertados y quitados en una pila por el sistema operativo.

Llamadas al sistema en Xv6

Las llamadas al sistema tienen un número asociado, el cual se define en *syscall.h*, estas son simples macros para luego matchear sus respectivos números en caso de un trap 64 (la comparación para saber si el trap es una llamada al sistema se realiza dentro del archivo *trap.c*, en la función *trap*). En *syscall.c* se define una tabla de punteros a funciones, cuyos índices son justamente esas macros, y apuntan a una función que contiene la lógica de la llamada al sistema, o *system calls handlers*. En esta instancia, las funciones referenciadas reciben void como parámetro. Esto ocurre, puesto que para obtener sus parámetros debe accederse al stack, dado que este sistema operativo utiliza pasaje de parámetros por medio de la pila de llamada a la función. Las funciones para obtener parámetros son posicionales y según sus tipos; *integer, pointer, char* y se encuentran en *syscall.c*. Las signaturas formales de las llamadas al sistema deben definirse en el archivo *defs.h*.

Por otro lado, el nombre de la llamada al sistema debe declararse en *usys.S* según *SYSCALL(nombre)*, la cual realiza las siguientes operaciones:

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
```

```
    ret
```

Es decir que si nosotros definimos la llamada al sistema número 22 (ps) en *syscall.h*, en la etapa de preprocesamiento se reemplaza tal valor de la macro resultando en un código assembler de la siguiente forma:

```
.globl ps;
read:
       movl $22, %eax;
       int $64;
       ret
```

de esta manera podemos referenciar a la llamada al sistema correcta en la tabla definida en *syscall.c*. Luego de la ejecución se retorna al punto desde donde se llamó a la llamada al sistema, *iret*, valor que se encuentra dentro del trapframe.

También es necesario describir la cabecera de las funciones con sus parámetros en *user.h*, en donde se define formalmente cada una de las llamadas al sistema a modo de que el usuario pueda referenciarlas desde el lado del usuario, una vez invocadas estas funciones, se lanza el trap para dar inicio al proceso previamente descrito.

Más allá de los detalles de implementación, xv6 sigue el mismo proceso que en cualquier otro sistema operativo moderno, posiblemente con alguna variación, puesto que la simplificación es el objetivo en este sistema. Los procesos tienen su propio espacio de direcciones de memoria, el cual es asignado a cada uno de ellos por el sistema operativo. Esto se logra gracias a la memoria virtual, en donde se asigna a todos los procesos direcciones lógicas que se deben mapear o "traducir" con direcciones de memoria física. En el espacio de direcciones de cada proceso, se comienza desde un cero lógico, aunque ese cero se corresponde con direcciones de memoria diferentes entre cada uno de ellos. Asimismo, cada uno de estos espacios tienen una dirección de memoria en común que representa el tope de la misma, 0xFFFFFFFF. De nuevo, cada proceso tiene asignado una dirección física diferente para cada tope de su espacio. Esto da cuenta de cómo un proceso cree que tiene su propio sistema de memoria, sólo para él. Esto es deseado, puesto que un proceso no debería poder leer o escribir datos en la memoria de otros procesos, ya que esto representa una vulnerabilidad en el sistema operativo y si el fin es justamente comunicar dos o más procesos, existen otros métodos para hacerlo[3]. Ese conjunto de direcciones de memoria asignado a cada proceso está particionado en dos secciones: la memoria de usuario y la del kernel. La memoria del usuario se compone de las instrucciones a ejecutar en el programa junto con las variables globales, arriba se encuentra el stack de usuario y encima de este se halla el heap. Este esquema coincide con el mismo que se describió en la sección de lenguajes de programación, en donde se dio una introducción a esta misma parte de la memoria. Por otro lado, en el mismo espacio de direcciones se halla la memoria del kernel. En esta parte se encuentra, al inicio 0x80000000 y hasta la posición 0x80100000 los datos de los componentes de E/S conectados a la computadora. Aquí la descripción no es tan lineal, puesto que hay dos caminos posibles. Si el proceso se está ejecutando en modo usuario, entonces no hay nada desde 0x80100000 hacia arriba, con lo cual sólo se encuentra la bios en la sección de memoria del kernel. Ahora bien, si el proceso se está ejecutando en modo kernel, esto es, luego de una interrupción 64, encima de

[3] Los procesos deben comunicarse entre ellos con algún recurso compartido, como un archivo, o bien mediante *requests* y *responses*, como la arquitectura cliente-servidor. Otra forma de realizar esto son los sockets o señales. Un método muy común es la comunicación entre procesos mediante *pipes*, los mismos que se utilizan en la práctica al programar en bash.

0x80100000 se insertan las instrucciones a ejecutar por la llamada al sistema y encima de esta, el stack de kernel. Asimismo, cuando se encuentra en modo kernel ejecutando un proceso, el stack de usuario no se elimina, pero no es usado activamente puesto que si el kernel leyera información introducida por el programador, el sistema quedaría vulnerable debido a un error o bien, adrede. Cuando se entra en modo monitor, se cambia el privilegio actual y el procesador realiza un switch al stack de kernel. Cuando termina la ejecución, decrece el nivel de privilegio actual y se retorna al stack de usuario, obviamente volviendo nulo el stack de kernel. Si un proceso de usuario quisiera acceder a una dirección de memoria de otro proceso o quisiera realizar una modificación sobre alguna sección de memoria del lado kernel, el sistema arrojará un trap 14 (page fault), por obvias razones. El proceso de switching entre stacks se discutirá en la sección de *scheduling*.

Cabe destacar que este proceso de mapeo entre memoria virtual y real son realizados vía hardware por una *unidad de gestión de memoria* o MMU por sus siglas en inglés.

Implementación del comando 'Process Status'

Para llevar a cabo el comando '*process status*' fue necesario implementar una llamada al sistema, puesto que el usuario no debería poder acceder directamente a las estructuras que contienen información acerca de los procesos. El lugar adecuado para implementar la lógica de la llamada al sistema fue en el archivo *proc.c*, en donde se tienen todos los datos de los procesos del sistema operativo. Además de la modificación de los archivos nombrados en la subsección anterior, se matchearon los debidos parámetros en el archivo *sysfile.c* y desde aquí sí se puede llamar efectivamente a la función que implementa la lógica de la llamada al sistema, en *proc.c*. Esta recibe un puntero a una estructura llamada *psOutput* definida en *psOutput.h* cuyo único fin es el de retornar la información de los procesos al usuario en dicho contenedor. La estructura tiene un arreglo de string, y tres de enteros, ambos de *#NPROC* elementos. En el arreglo de strings (que en realidad cada elemento del arreglo es un arreglo de chars, otra forma de llamar a un string, pero única forma de definirlo en C) contiene los nombres de los procesos. Un arreglo de enteros posee las prioridades de cada uno de los procesos (propiedad añadida que se explicará posteriormente) y el segundo posee la cantidad de memoria que ocupa cada uno de ellos. El tercer arreglo de enteros almacena el id de cada proceso. Estos están indexados posicionalmente, y dichos arreglos se recorren hasta un número definido en la estructura que indica cuántos procesos hay actualmente. Puesto que se deben definir arreglos de tamaño fijo y sabemos que como máximo hay NPROC procesos en el sistema, se debe incorporar una variable que contenga la cantidad de elementos válidos en dichos arreglos, puesto que puede ser posible (en la mayoría de los casos) que haya menos procesos que NPROC. Este número sirve para iterar e imprimir los números, la variable de control en un loop. La estructura en cuestión es:

```
typedef struct {
    char name[16];
} strs;

struct psOutput {
    strs     pname        [NPROC];
    int      pid          [NPROC];
    uint     psz          [NPROC];
    int      priority     [NPROC];
    int      processes;
};
```

Esta llamada al sistema sigue los mismos pasos mencionados en la sección previa, en donde se explicó cómo era el proceso para utilizar estas funciones. Un breve acercamiento, mediante un esquema, puede verse en la figura [figura 6].

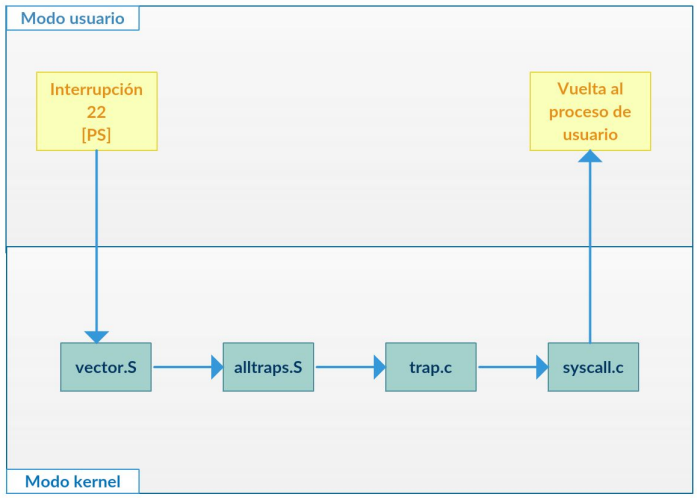

Figura 6: *acercamiento a la llamada al sistema 22 [Process Status Command]*

Scheduler

Introducción

La planificación o *scheduling* es una de las tareas centrales de un sistema operativo, y es realizada por el scheduler, valga la redundancia. Los sistemas operativos tienen distintas colas en donde se encuentran los procesos. Cuando estos son cargados y están listos para ejecutarse, se encuentra en la *cola de procesos listos*, o *ready queue*. Cuando un proceso están esperando para un dispositivo de entrada/salida y este no puede responder porque está atendiendo a otro proceso, estos se insertan en una *cola de espera*, o bien *device queue*. La tarea del planificador consiste en seleccionar los procesos de estas colas según cierto criterio.

Existen varios tipos de schedulers; los hay de largo plazo, los cuales eligen procesos de la cola de listos o *ready queue*, para cargarlos en memoria. Existen también de corto plazo o *CPU schedulers*, los cuales eligen entre los que están listos en memoria para darle la CPU. Y finalmente, los hay también de mediano plazo, en donde se selecciona un proceso de entre los que están en memoria para llevarlo temporariamente a memoria secundaria por necesidades de, esto es conocido como swapping[4]. El planificador de corto plazo está llevando procesos de memoria al CPU muy frecuentemente, con lo cual debe ser muy rápido (si el context switch tarda mucho, se produce un overhead, lo cual no es deseado). Por otro lado, un planificador de largo plazo puede tomar mucho más tiempo para decidir qué proceso seleccionar dado que hay un lapso de tiempo muy amplio entre ejecuciones de procesos. Los schedulers de mediano plazo toman procesos de la memoria principal y los asignan al CPU para su ejecución, y en otros casos se quita un proceso de la memoria principal para llevarlo a disco (swap), con lo cual se reduce el grado de *multiprogrmación*, es decir, la cantidad de procesos que se encuentran en memoria en un momento determinado.

[4] Los sistemas operativos modernos utilizan esto todo el tiempo. De hecho Linux posee una partición extra en el disco para realizar esta tarea llamada *linux-swap*.

Esto parece muy sencillo, pero la realidad está muy lejos de ello. Lo que hace el scheduler para tomar un proceso y asignarlo a la CPU o quitarlo de ella para volver a ponerlo en memoria principal es llamado *context switch* o cambio de contexto. Esto es, cada proceso posee una estructura con todos sus elementos de ejecución; los estados de los registros del procesador mientras se ejecutaba, sus estructuras, entre otras cosas. En las secciones anteriores se explicó cómo se ubicaba esquemáticamente un heap y un stack de usuario. Resulta que esto ocurre en cada proceso, es decir, cada uno de ellos tiene su propio stack de usuario, su propio heap y sus propias direcciones de memoria, protegidas de los demás procesos, lo cual encapsula toda la información necesaria por una cuestión de seguridad (una función elemental, pero no sencilla del sistema operativo), punteros a segmentos y páginas, datos de memoria, entre otros. Ahora bien, todos estos elementos nombrados deben ser almacenados en algún lugar antes de ser quitados del CPU (*preemption*) para que el proceso pueda volver al estado desde el cual fue desalojado previamente, dado que todos estos registros y componentes se van a utilizar para nuevos procesos. La tarea de guardar el estado de un proceso y luego correr otro proceso es muy lento y consume cientos de ciclos. Uno de los grandes retos de los arquitectos de procesadores es optimizar todo lo posible el funcionamiento del CPU y una métrica para evaluar el rendimiento es el CPI o (*Cycles Per Instruction*). Si bien los procesadores superescalares actuales suelen ser vistos como “egoístas” en referencia a que llevan al extremo la idea de funcionar más rápido tomando todos los recursos posibles, sería en vano agregar memorias más rápidas, más potencia de cálculo, entre otras cosas, si al fin y al cabo el context switch (tarea natural del sistema operativo) va a ocurrir muy frecuentemente y va a desperdiciar cientos y cientos de ciclos. Es por ello que esta tarea no la realiza solamente el sistema operativo -scheduler-, sino también el hardware. Los procesadores modernos tienen mecanismos para llevar a cabo este tipo de tareas, como por ejemplo la tecnología *hyperthreading* de intel, los cuales son procesadores SMT que consisten, a grandes rasgos, en insertar un campo llamado *process id* en muchos de estos elementos, como por ejemplo en memorias caché, TLB (Translation Lookaside Buffer), BTB (Branch Target Buffer), entre otros, implicando entonces que el cambio de contexto -el cual en realidad no existe, porque todos los contextos están activos simultáneamente- consuma pocos ciclos (2 o 3, por ejemplo). Si el hardware no posee tecnología de este tipo, puede ayudar a realizar ciertas tareas, pero requiere un cambio de contexto explícito de parte del sistema operativo.

Hay varias políticas para realizar la tarea de planificación. En general, los sistemas operativos buscan el equilibrio entre dos aspectos fundamentales: *fairness* y maximización del uso del CPU para que este no quede ocioso (todo tiempo que el CPU esté sin uso es tiempo desperdiciado). A su vez se requiere, además, que se minimicen los tiempos de context switching y demás. Existen varios tipos de schedulers; *First Come First Served*, *Shortest Job First*, *Priority Scheduling*, *Round Robin*, entre otros. Algunos de ellos no se utilizan con desalojo, es decir, hasta que el proceso no se completa, no se quita del CPU, sin embargo, estos no conservan la idea de justicia entre asignación de recursos a procesos, pero son muy simples. Otros, como el Round Robin, tienen desalojo para lograr la equidad entre la utilización del CPU entre los procesos, sin embargo, este tipo de scheduling tiene una desventaja y es que realiza cambios de contexto muy frecuentemente, puesto que el desalojo se da cada vez que un *timer* llega a cierto punto y ocurre una interrupción que desaloja al proceso actual (o a los procesos en caso de que el cpu tenga varios núcleos), pero como ventaja, tiene un tiempo de respuesta muy bueno. Además de guardar los registros de cada procesador, cada vez que se desaloja un proceso, se debe buscar a otro para ejecutar, y esa decisión debe tomarse rápido.

Implementación en Xv6

Tal como se explicó en la introducción, el scheduling ocupa un lugar central en el sistema operativo. El sistema Xv6 utiliza un algoritmo de scheduling Round Robin simple, sin soporte de prioridades. Este scheduler se encuentra en el archivo *proc.c*. El intervalo de tiempo para desalojo, o timer es de 10ms para cada procesador. En *trap.c* se genera el trap para desalojar un proceso, según el siguiente código:

```
if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
       yield();
```

Luego, la función *yield()* realiza un lock sobre la tabla de procesos, asigna al proceso desalojado el estado RUNNABLE y luego llama a la función *sched()*, la cual mueve los elementos del procesador a la estructura del proceso desalojado; guarda su estado. Finalmente se llama a la función *schedule()*, la cual selecciona el próximo proceso a correr de una lista de procesos, también bloqueando la tabla de otros hilos.

Cabe destacar, que en Xv6, como en cualquier sistema operativo, los procesos tiene diferentes estados, los cuales tienen nombre muy particulares [figura 7]. El estado EMBRYO se le asigna a un proceso cuando está siendo creado, RUNNABLE cuando está listo para empezar a correr, RUNNING cuando está ejecutándose en algún CPU, SLEEPING cuando está bloqueado y esperando por algún componente de entrada/salida y finalmente ZOMBIE, que se da cuando un proceso termina, y se discutirá más adelante, en la sección de threads. También existe el estado UNUSED, el cual sirve para ver si un lugar determinado de la tabla de procesos está ocupado o no en el proceso de creación y si no está usado pasa a el estado EMBRYO. Sin embargo, mi opinión personal es que esto no es un estado en sí, sino más bien un control para la creación que bien podría realizarse con un puntero a la próxima posición libre.

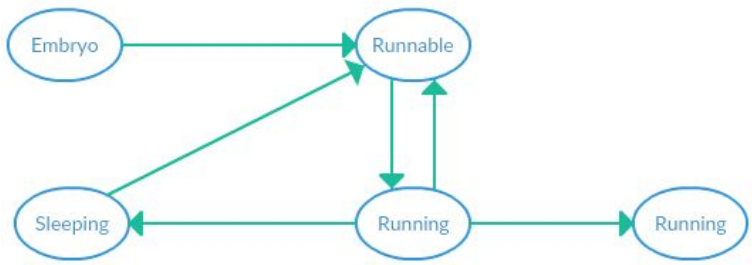

Figura 7: *diagrama de transición de estados de procesos.*

Modificación con Prioridades

El trabajo consistió en modificar este scheduler para establecer propiedades a los procesos. Primero se agregó un nuevo atributo a la estructura proc, situada en *proc.h*, llamado *priority*. Este atributo es un entero simple, que se puede establecer mediante una llamada al sistema, implementada para este propósito, llamada *setPriority(int)*, también situado en *proc.c*. Esta cambia la prioridad del proceso actual, el que está en estado RUNNING por el parámetro dado. Por otro lado, cuando se crea un proceso, su prioridad por defecto es 0. Asimismo, cuando se utiliza fork, se asigna por defecto una prioridad al proceso hijo, en este caso 15. Los procesos más prioritarios son aquellos con número mayor al de otros. Todos estos números asignados por defecto, fueron elegidos arbitrariamente para testear la modificación al scheduler, y no por otro motivo.

Luego se modificó la función *schedule()*, en donde se agregó la lógica para obtener cada dos ciclos el proceso de mayor prioridad, y en el tercer ciclo, el de menor prioridad. Esto implica una búsqueda lineal sobre la tabla de procesos al próximo proceso a ejecutar, lo cual consume más tiempo

que antes, como contra. Por otro lado, se podría optimizar realizando una búsqueda binaria sobre el arreglo, aunque previamente haya que insertar elementos de forma ordenada según su prioridad.

Para modificar la prioridad del proceso actual, se implementó la llamada al sistema `int sprior(int)` la cual fue implementada en *proc.c* y puede ser utilizada como un comando, dada su implementación en el archivo *sprior.c* en donde recibe un argumento tipo entero.

Ahora, esto trae un inconveniente; la idea central del algoritmo Round Robin es que se mantenga la imparcialidad al asignar procesos al CPU. En esta nueva implementación, si tuviésemos, por ejemplo, los procesos siguientes [tabla 1], el scheduler siempre tomará al proceso 1 (con prioridad 50) dos veces y al proceso 4 (con prioridad 0) una vez, y así sucesivamente, y por consiguiente se bloquearán muchos otros (los de prioridad media), ergo, se genera *starvation* y no es deseable.

Pid	1	2	3	4	5	...	NPROC
Prioridad	50	50	20	0	10	...	25

Tabla 1: *tabla de procesos en un determinado momento.*

Una forma de solucionar este inconveniente es utilizar una técnica llamada *aging*, la cual consiste en aumentar su prioridad si ha pasado más de un valor umbral de tiempo en la cola de ready. Como en la selección del próximo proceso se recorre la tabla de procesos entera (búsqueda lineal), se aprovecha y se verifica que el proceso se haya ejecutado recientemente, si esto no ocurrió entonces se aumenta su prioridad en una unidad. El valor umbral elegido es 20, pero bien podría ser 50 o 5. Un criterio para seleccionar un número es cuánto importa la imparcialidad en el sistema; si los procesos de alta prioridad son críticos entonces el valor umbral debe ser muy grande dado que se opta por ejecutar generalmente los procesos con mayor prioridad. El código para añadir esta particularidad al planificador es muy sencillo y consiste en agregar al PCB de los procesos (struct proc) un nuevo atributo llamado *lastExcution*, el cual se inicializa en 0 al momento de crearse y cada vez que se lo encuentra en la tabla de procesos pero no se selecciona para correr, se aumenta su valor de última ejecución en uno. Eventualmente, su última ejecución superará el valor umbral y comenzará a aumentar su prioridad, gradualmente. Esto asegura que el proceso no permanecerá totalmente bloqueado, más allá de que tarde en ejecutarse.

Threads

Introducción

En todas las secciones anteriores se habló de procesos. Resulta ser que los sistemas operativos, en general, proveen soporte para la creación y manipulación de threads. Una forma de verlo -muy simplista- es decir que cada proceso ‘se divide’ en pequeñas partes que comparten ciertos recursos. Estas pequeñas partes son llamados *hilos* o *threads* y comparten entre ellos la misma tabla de páginas, es decir, tienen asociado las mismas direcciones de memoria o al menos tienen permiso para acceder a ellas, aunque no tendrán exactamente las mismas páginas durante su ejecución (se discutirá más adelante). Esto quiere decir que los procesos ya no son una unidad indivisible de ejecución, sino que ahora los hilos lo son, y por consiguiente, el scheduler ya no opera sobre procesos, sino sobre threads. Si un proceso no tiene threads, entonces este es un único hilo. Esto no implica que los procesos no deban ser utilizados, por el contrario; si bien no se los trata como antes, son una

estructura en donde se guardan los datos antes del cambio de contexto y sirve además, para que nuevas ramificaciones o nuevos hilos utilicen datos de el.

En los sistemas operativos se distinguen dos tipos de threads: los threads de usuario y los threads de kernel. Estos tienen ligeras diferencias y tal vez la más importante es que el scheduling sobre estos últimos los realiza el sistema operativo, en los otros casos, la tarea la realiza el usuario y en general se utilizan ciertas librerías para ello. Otra diferencia es que los threads de kernel generalmente producen overhead por los cambios de contexto, con lo cual suelen ser más lentos que los hilos de usuario, dado que estos últimos son más "livianos". Además, el kernel no tiene idea de la existencia de hilos de usuario, pero sí de los hilos de kernel. Existen varios modelos de hilos de kernel-usuario; uno a uno, uno a muchos, muchos a muchos [figura 8].

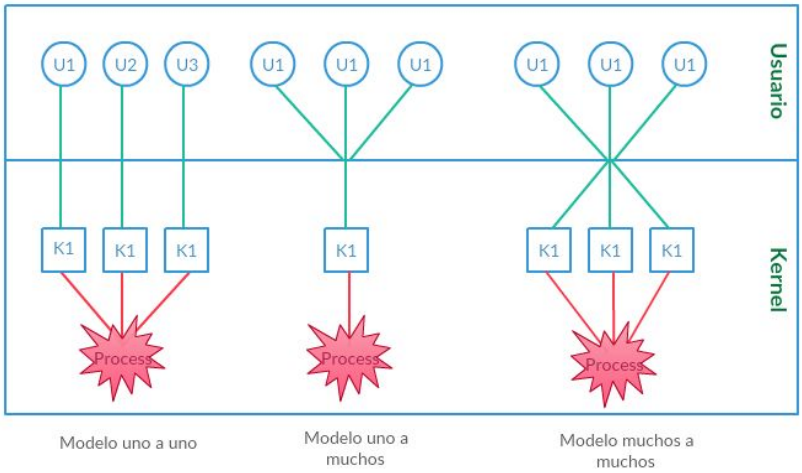

Figura 8: *modelos de hilos kernel-usuario.*

Implementación en Xv6

Xv6 no soporta hilos de kernel por defecto, sin embargo, tiene muchos elementos necesarios para realizar esta tarea y algo muy importante es que posee tabla de páginas, lo cual hace sencillo la posibilidad de compartir direcciones de memoria entre hilos del mismo proceso. Tal vez, lo más cercano que Xv6 posee a la creación de un thread es la creación de un proceso y asignarlo como hijo del proceso actual, con la llamada al sistema *fork()*. Es muy importante destacar que está se aplica para crear un nuevo proceso, no un hilo.

En las secciones anteriores se habló del espacio de direcciones de memoria de un proceso; estos tienen una serie de páginas asignadas, las cuales están protegidas de otros procesos, lo que se conoce como encapsulamiento, tarea realizada por la MMU en conjunto con el sistema operativo (no es trivial). Esto da varias ventajas, y es que crear un nuevo proceso hijo es sencillo dado que estos comparten las mismas direcciones de memorias, es decir, cuando se crea un proceso hijo, se le asigna las mismas páginas que posee el padre [figura 12], luego se inicializan algunos elementos del proceso y este ya puede correr, una vez insertado en la tabla de procesos. Esto debe tenerse en cuenta para implementar los procesos livianos (threads). Una cuestión importante es que Xv6 implementa *Copy On Write*, es decir, inicialmente los procesos hijos y su proceso padre comparten las mismas páginas hasta que un proceso modifica una de ellas; recién ahí se crea una nueva para el proceso que quiso modificarlo. Esto se da por una cuestión de performance, para retrasar tanto como sea posible la copia de páginas, evitando penalizaciones por swapping entre páginas de los distintos procesos. De todas formas, siempre estamos hablando de procesos, no de threads; esto debe quedar en claro, ya que sólo

se describen algunas de las funcionalidades importantes ya implementadas para evaluar su relación con el soporte de threads agregado.

Es ahora donde tiene sentido hablar del estado ZOMBIE de procesos: siempre que un proceso termina su ejecución se vuelve en estado *zombie*, y no son removidos de la tabla de procesos aún, pero el scheduler tampoco los selecciona. Esto sirve para que el padre pueda leer es estado de salida del proceso hijo, y cuando el padre lee esta señal, sus procesos hijos zombie son retirados del sistema operativo. Si el padre no lee esta señal y llama a wait(), entonces estos se mantienen indefinidamente. Esto es diferente de un proceso huérfano, el cual en Linux y Xv6, se asigna a la raíz del árbol de procesos; *init*.

Soporte de Kernel Threads

Este fue, sin dudas, el problema más difícil de solucionar de la guía de ejercicios. Primero se realizó la implementación de la llamada al sistema que permite crear threads: *kthread_create()*. Para ello hay que empezar por algunos supuestos; cada proceso tiene un máximo de dos threads para simplificar la manipulación de la memoria a la hora de programarlo. A cada uno de estos threads se les asigna dos páginas[5] (ni más ni menos, sólo dos, para simplificar), esto ocurre en *exec.c*, en el mismo lugar que se crea un proceso a partir de un programa, ni bien se lo crea se le asignan cuatro páginas (dos para cada thread). Esto representa dos problemas; el primero que se desperdicia la memoria, puesto que si se tienen NPROC procesos y ninguno crea threads, entonces desperdiciamos 1,048576 MB[6]. Por otro lado, uno de los procesos podría necesitar muchos más threads, no sólo dos y esta implementación estaría limitando estas necesidades. Por ello se recuerda nuevamente que la finalidad de este sistema operativo es orientada a la enseñanza y el ejercicio práctico es pura y exclusivamente para que el alumno pueda verter los conocimientos adquiridos de la materia, y no un sistema que se va a utilizar en sistemas reales, con lo cual ni se van a crear 64 procesos, ni se van a necesitar más de dos threads por proceso, y si así fuera debería reservarse espacio para más threads. En el archivo *exec.c* se encuentra la porción de código que realiza dicha tarea:

```
//Reservo 4 páginas para el stack del nuevo thread.
if((sz = allocuvm(pgdir, sz, sz + 4*PGSIZE)) == 0)
  goto bad;

proc->childs = 0;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
proc->secondChild = (uint) sz;
clearpteu(pgdir, (char*)(sz - 4*PGSIZE));
proc->firstChild = (uint)sz - 2*PGSIZE;
```

Como se puede ver, en el PCB (Process Control Block, estructura proc) se almacena información sobre estas páginas; punteros a los lugares en los que comienzan cada una de ellas, para asignarlas a threads que se creen posteriormente (si se crean). Otro detalle importante es que las restas en la porción de código de arriba se deben a que la memoria es decreciente, entonces se apilan "hacia abajo". Luego se define la llamada al sistema `int add_kthread(void (*fcn)())`, la cual recibe una función como parámetro; la función que va a ejecutar el thread. Esta implementación es demasiado simplista y en ella se asumen muchas cosas que en un sistema real sería intolerable; la función no

[5] Se asignan dos páginas por proceso para permitirles crecer más de la cuenta, pero imposibilitando acceder a los datos que rebalsaron en la primer página.

[6] NPROC es 64, y es una macro definida en *param.h*. PGSIZE se define en *mmu.h* y es igual a 4096 Bytes. La cantidad de memoria creada para threads si tuviesemos 64 procesos (peor caso) es 4*NPROC*PGSIZE=4*64*4096 Bytes = 1,048576 MB.

recibe parámetros, con lo cual tendrá que operar con alguna estructura global definida fuera de su ámbito local.Además no tiene valor de retorno, es void y en su cuerpo debe tener la función *exit()*. Esto se debe a que agregar los punteros a parámetros modificaban el espacio de memoria y se torna más complicado operar con tales direcciones, pero se podría agregar. El enunciado no pedía esto explícitamente, con lo cual se podría plantear como trabajo futuro. En *add_kthread* debe verificarse que si el proceso actual tiene creados dos threads, entonces no se debe permitir que se creen más, puesto que no tendrían más páginas para asignarse. En el cuerpo de tal llamada al sistema, entre otras cosas, ocurre lo siguiente:

```
//Retorno a la función pasada por parámetro.
np->tf->eip = (uint)fcn;
//Registros que apuntan al stack.
np->tf->esp = (proc->childs == 0) ? proc->secondChild : proc->firstChild;
np->tf->ebp = np->tf->esp;
```

Se ve como eip apunta a la próxima instrucción a ejecutar a nivel de usuario, en donde se le asigna un puntero a tal función. Luego, se setean la base y el tope del stack de usuario, dependiendo de cuántos threads se hayan creado, el tope será la dirección de la página del primer proceso o del segundo (está al revés porque la memoria es decreciente).

Al llamar a la llamada al sistema *add_kthread*, los threads creados permanecerán en estado zombie, porque el proceso padre termina antes que ellos. Para evitar esto hay que invocar a otra llamada al sistema llamada según su definición int `join_kthreads(void)` la cual llama a tantas funciones wait como threads existan. Estas llamadas están implementadas en *proc.c*, de forma contigua. Una vez que se llama a *add_kthread*, al final debe llamarse a la función que realiza el join, para que no queden en estado zombie.

Conclusión

La enseñanza de los sistemas operativos modernos suele ser muy complicada puesto que los conceptos explicados en clase no se alcanzan a comprender totalmente por su complejidad. Por otro lado, muchos de estos conceptos se explican de forma teórica, y fuerzan al alumno a tomarlos como algo abstracto. Por tal motivo se incluyó un trabajo práctico especial, el cual dio un acercamiento a la implementación de los sistemas operativos modernos pero simplificando la complejidad de los mismos.

En el presente trabajo se profundizó sobre varios componentes clave de un sistema operativo y en particular, se realizaron diversas modificaciones en el código fuente del sistema operativo Xv6, otorgando una visión un poco más realista al alumno.

Referencias

[1] D.A Patterson y J.L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th Ed. Morgan Kaufmann, 2013.
[2] A. Silberschatz y P. B. Galvin, “Operating Systems Concepts”, 7th Ed. Addison-Wesley, 2003