Trabajo Práctico – Hilos de Kernel

Adrián Fungueiriño, Sebastián Deuteris, Hernan Jalil, Emiliano Faraone, Jesus Ruiz

Abstract. El presente trabajo práctico tiene como objetivo desarrollar una investigación acerca de la administración e implementación de hilos a nivel Kernel. La finalidad del documento es establecer un diseño base para poder implementar este tipo de hilos en el sistema operativo S.O.D.I.U.M.[1]

Keywords: Hilo, Thread, TCB, ULT, KLT.

1 Introducción

Cuando nos enfrentamos a la presente investigación nos fue indispensable tomar distintas visiones en lo que respecta a la ejecución. Una ejecución tiene diferentes significados según de que lado la veamos. Para el procesador la ejecución es en base a una Tarea (“Task”), esa es su unidad de trabajo. Por otra parte, cuando nos paramos en el sistema operativo vemos que la ejecución es sobre procesos y según si está implementado, sobre hilos también. Tanto los procesos como los hilos le son transparentes al procesador ya que él maneja tareas que respetan una estructura definida, esto no sucede a la inversa, por lo tanto se implementan distintas metodologías en su administración a fin de sacar el mejor provecho en la ejecución.

Al enfocarnos en la administración de hilos, para poder diseñar una implementación, debemos por un lado comprender que tipos podemos tener, que significa cada uno de ellos, es decir, trabajar con hilos a nivel Kernel (KTL), a nivel Usuario (ULT) y combinados (KLT + ULT) y que metodología de trabajo comprenden.

[1] SODIUM.: Sistema Operativo Del Departamento de Ingeniería de la Universidad de La Matanza (UNLaM).

2 Marco Teórico

Para el procesador, la unidad de trabajo es una tarea, esta se encuentra formada de dos partes, un espacio de ejecución y un segmento de estado de tarea (también llamado contexto). Este último no solo define el estado sino que además define al espacio de ejecución.

El estado de una tarea en ejecución está definido por lo siguiente:

- El espacio de ejecución de la tarea actual, definido por los selectores de segmentos CS, DS, SS, ES, FS y GS.
- El estado de los registros de uso general.
- El estado del registro EFLAGS
- El estado del registro EIP
- El estado del registro de control CR3 (cuando en la tarea se implementa paginado, se almacena la dirección base de la página).
- El estado del registro de tareas (Task Register - TR)
- El estado del registro LDTR
- La dirección base del mapa de direcciones de Entrada / Salida
- Punteros a las distintas pilas (pila de nivel de privilegio 0, 1 y 2)
- Link a la tarea previamente ejecutada (para el caso de tareas anidadas).

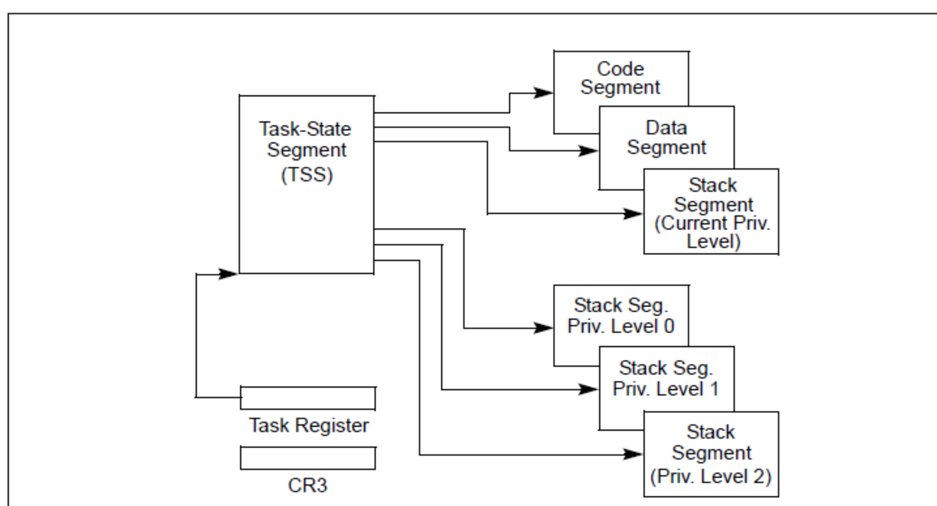


Figure 7-1 Manual INTEL Vol. 3A “Structure of a Task”

Estructuras de una tarea:

El procesador define cinco estructuras para el manejo de tareas:

- Task-State Segment (TSS)
- Task-Gate Descriptor
- TSS Descriptor
- Task Register
- NT Flag (ubicado dentro del registro EFLAGS).

Task-State Segment (TSS)

El TSS contiene un conjunto de campos que definen el estado de una tarea. Se encuentra dividido entre campos dinámicos y estáticos. La diferencia radica en que los campos dinámicos son modificados cuando una tarea es suspendida durante un cambio de contexto (Context-Switch), almacenando en el TSS su último estado. Y los estáticos normalmente no son modificados, son cargados al momento de la creación de la tarea y no varían durante la ejecución.

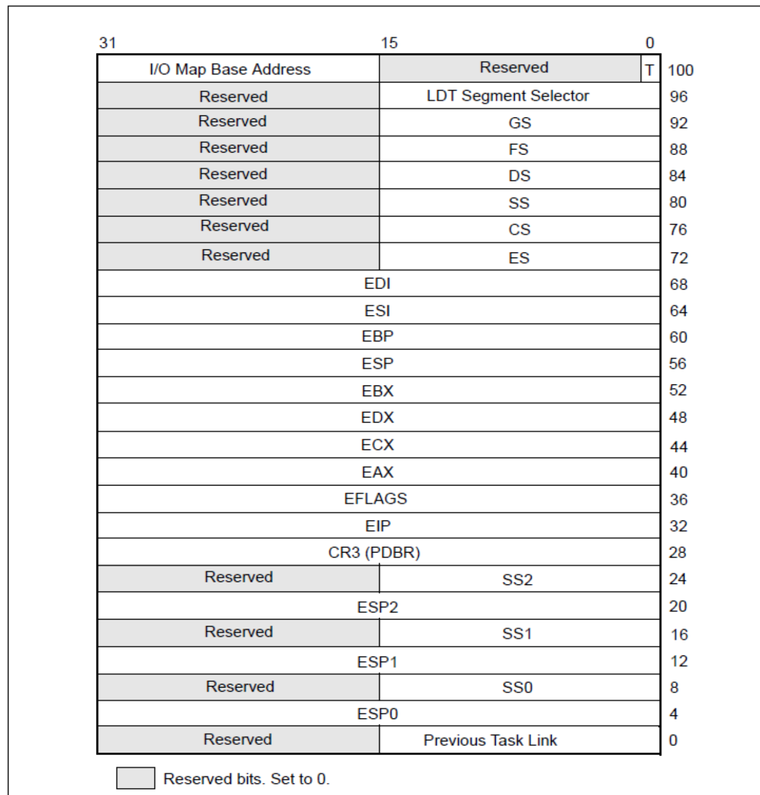


Figure 7-2 Manual INTEL Vol. 3A “32-Bit Task-State Segment (TSS)”

Los campos dinámicos son los siguientes:

- Campos para registros de uso general (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI);
- Campos para selectores de segmentos (ES, CS, SS, DS, FS, GS);
- Campo para el registro EFLAGS;
- Campo para el registro EIP;
- Campo para la relación con la tarea anterior (En caso de que la tarea sea anidada).

Y los campos estáticos son los siguientes:

- Campo para el selector de segmento LDT;
- Campo para el registro de control CR3 (Contiene la dirección física de la base del directorio de páginas a ser usados por la tarea);
- Campos para los punteros a las pilas de los distintos niveles de privilegio. Estos punteros a pila están formados para cada nivel como un selector de segmento para la pila (SS0, SS1, SS2) y un offset dentro de la misma (ESP0, ESP1, ESP2);
- T (debug trap) Flag - Cuando se encuentra activo, el T Flag causa que el procesador lance una excepción de debug ante un Context-Switch hacia esta tarea;
- Campo para la dirección base del mapa de Entrada/Salida.

TSS Descriptor:

El Descriptor del TSS define al segmento, y se encuentra almacenado dentro de la GDT. Cabe destacar que cada TSS tiene uno y solo un descriptor.

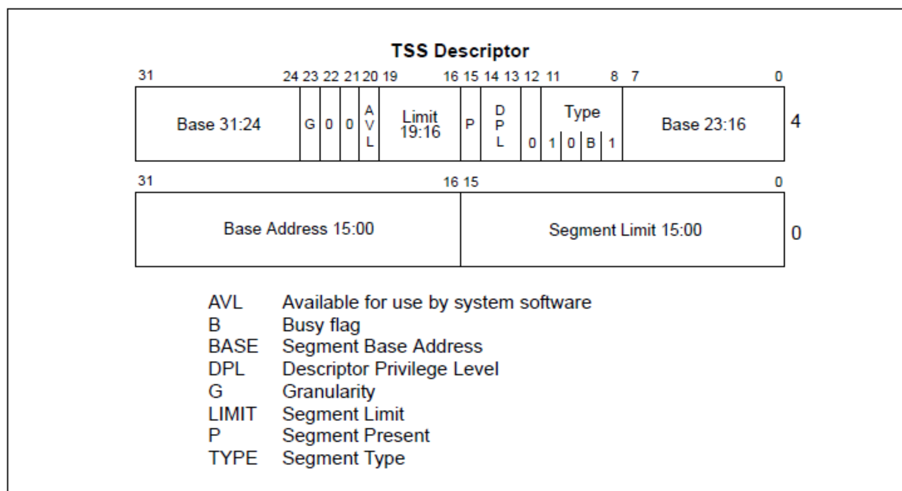


Figure 7-3 Manual INTEL Vol. 3A “TSS Descriptor”

Entre otros campos, el TSS Descriptor contiene un identificador llamado Busy Flag (B). El B Flag indica si la tarea se encuentra ocupada. Este se activa cuando una tarea se encuentra en ejecución o suspendida. Esto sirve para controlar que no ocurra recursividad y también para no permitir que se llame a una tarea que esté en estado de ejecución o suspendida. Esto último toma mucha importancia en configuraciones multiprocesador ya que este flag se utiliza para prevenir que dos procesadores invoquen a la misma tarea al mismo tiempo.

Cualquier programa o procedimiento con acceso al TSS Descriptor puede enviar a ejecución una tarea mediante una instrucción CALL o JUMP siempre y cuando el valor del nivel de privilegio actual (CPL) sea menor o igual que el nivel de privilegio del descriptor (DPL).

Comúnmente el valor del DPL perteneciente al TSS Descriptor es configurado menor a 3, para que solo los programas con privilegio puedan ejecutar cambios de tareas. Sin embargo, en aplicaciones multitarea (multitasking), el valor del DPL para algunos descriptores de TSS son configurados a 3 para permitir intercambios de tareas a nivel de privilegio de aplicación o usuario.

Task Register:

El registro de tareas contiene el selector de segmento y todo el descriptor del TSS de la tarea actual. Este consta de dos partes, una visible que puede ser modificada por un programa, y una parte invisible que es inaccesible y solo modificada por el procesador. La parte visible contiene el selector de segmento del TSS, este apunta al TSS Descriptor ubicado en la GDT. En la parte invisible el procesador almacena dicho descriptor.

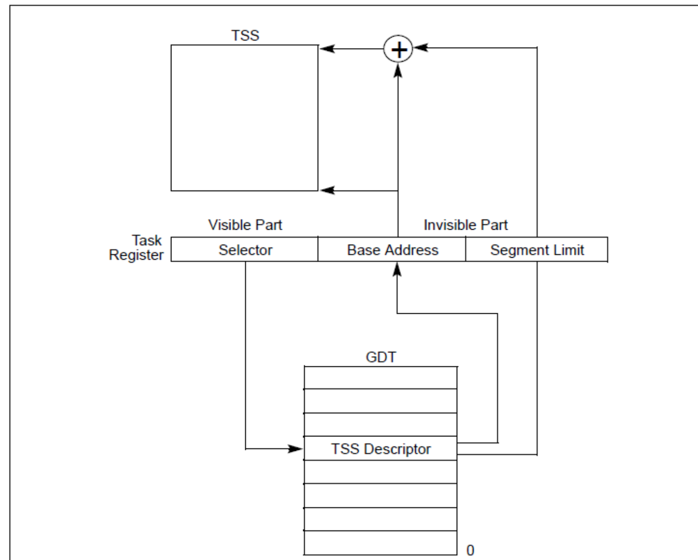


Figure 7-5 Manual INTEL Vol. 3A “Task Register”

Task-Gate Descriptor:

El Task-Gate Descriptor provee una manera indirecta y protegida de acceder a una tarea. Este descriptor puede estar ubicado en la GDT, la IDT y la LDT, y no necesariamente es único por tarea, puede haber más de uno. La indirección esta en que el Task-Gate Descriptor apunta al TSS Descriptor (ubicado en la GDT).

Al utilizar el Task-Gate Descriptor, el control de privilegios de la tarea recae sobre éste. Se utiliza su campo DPL en vez del ubicado en el TSS Descriptor. Lo que permite acceso a distinto nivel de privilegio según desde que Task-Gate se acceda.

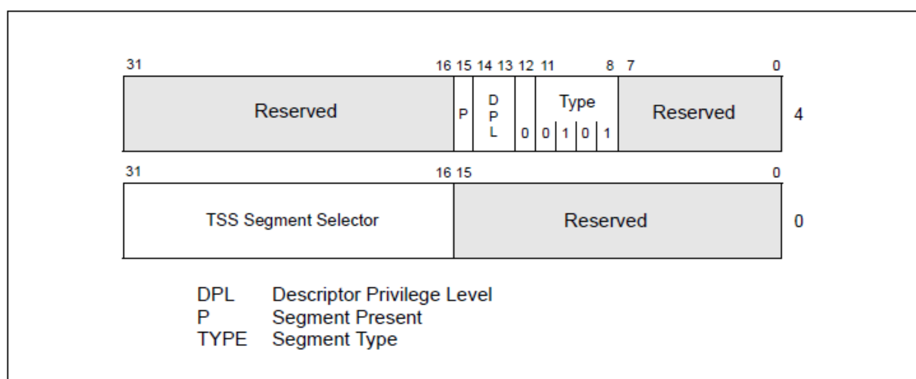


Figure 7-6 Manual INTEL Vol. 3A “Task Gate Descriptor”

Task-Linking:

Cuando se ejecuta una tarea que se encuentra anidada dentro de otra, hay un flag dentro del registro EFLAGS que es activado. Éste es el llamado NT Flag (Nesting Flag). Este anidamiento conduce a que luego de finalizada la ejecución de la tarea se tiene que volver a la llamadora. Esta relación se resuelve mediante un campo que se encuentra dentro del TSS como fue mencionado anteriormente. Lo que se almacena en ese campo es el selector de segmento del TSS llamador.

Cuando se ejecuta una instrucción CALL, una interrupción, o una excepción, se genera un Task-Switch y se activa el NT Flag. Cuando el programa ejecuta la instrucción IRET, el procesador antes de utilizar el campo del TSS anterior, verifica que el NT Flag se encuentre previamente activo, para que sea válida esa vuelta a la tarea anidada.

Cuando una instrucción JMP genera un Task-Switch, la nueva tarea no es anidada. No se usa el campo para la tarea anterior y se desactiva el NT Flag.

Intercambio de tareas (Task-Switch):

El intercambio puede ocurrir en las siguientes situaciones:

- La tarea ejecuta una instrucción CALL o JMP hacia el descriptor del TSS;
- La tarea ejecuta una instrucción CALL o JMP hacia el descriptor Task-Gate en la GDT o en su LDT;
- El vector de interrupciones o excepciones apunta a un Task-Gate Descriptor que se encuentra en la IDT;
- La tarea en ejecución ejecuta un IRET cuando el NT Flag en el registro EFLAGS esta activo.

Análisis de Almacenamiento de TSS

A continuación se realizará una comparativa entre la metodología de administración de KLT en la cual cada proceso consta de un TSS que es compartido por sus hilos, en contra partida a otro enfoque donde cada hilo consta de un TSS propio. Es importante destacar antes de iniciar el análisis, que a nivel procesador la ejecución es sobre tareas, el procesador desconoce los conceptos de proceso e hilo.

- **TSS única por proceso**

La administración de hilos en la cual comparten el TSS del proceso al que pertenecen consta de los siguientes pasos:

Nuevo proceso:

1.- Creación de la estructura del PCB y un primer TCB.
2.- Creación de un TSS con los datos del proceso y su primer hilo.
3.- Nueva entrada en la GDT para el descriptor del TSS.
4.- Opcional: Creación de un Task Gate dentro de la GDT o LDT que apunta al descriptor del TSS en la GDT.

Nuevo hilo:

5.- Creación de un nuevo TCB dentro de la estructura del PCB del proceso al que pertenece.

Intercambio de ejecución entre hilos:

Supongamos que disponemos de un proceso “PA” el cual contiene dos hilos “H1” y “H2”. Se ejecuta el planificador, se selecciona en primer nivel el proceso a ejecutar

y luego, en segundo nivel, el hilo. Cuando comienza por primera vez la ejecución de PA, este ya tiene asignado en su TSS los datos correspondientes a H1. Luego de finalizada la ejecución, ya sea por quantum o por su finalización propiamente dicha, se ejecuta nuevamente el planificador. Este punto genera una bifurcación:

- Si en la planificación se resuelve que vuelve a ejecutar el mismo hilo o se ejecuta un hilo de otro proceso, no se realiza ninguna modificación en el TSS.
- Si en la planificación se resuelve que pasa a ejecución otro hilo del mismo proceso, debemos, por un lado, salvar en el TCB el estado del último hilo ejecutado y, por otro lado, ubicar en el TSS los datos del próximo hilo a ejecutar.

Por último, se carga en el Task Register el selector de segmento del descriptor del TSS a ejecutar. En este caso el propio del proceso.

Ventajas:

- El Segmento del TSS es compartido. Por lo tanto hay uso eficiente de la GDT ya que se encuentra un solo descriptor de TSS por proceso.

Desventajas:

- Ante un cambio de contexto entre hilos del mismo proceso, se debe resguardar el estado del último hilo y cargar el estado del nuevo hilo a ejecutar.

- **TSS por hilo**

La administración de hilos en la cual cada uno dispone un TSS propio consta de los siguientes pasos:

Nuevo proceso:

1.- Creación de la estructura del PCB y un primer TCB.
2.- Creación de un TSS para el hilo.
3.- Creación de una entrada en la GDT para el descriptor del TSS.
4.- Opcional: Creación de un Task Gate dentro de la GDT o LDT que apunta al descriptor del TSS en la GDT.

Nuevo hilo:

5.- Creación de un nuevo TCB dentro de la estructura del PCB del proceso al que pertenezca.
6.- Se repiten los pasos 2, 3 y 4.

Intercambio de ejecución entre hilos:

- Se ejecuta el planificador. Se selecciona en primer nivel el proceso a ejecutar y luego, en segundo nivel, el hilo.
- Se carga en el Task Register el selector de segmento del descriptor del TSS a ejecutar. En este caso, el propio del hilo a ejecutar.

Ventajas:

- Ante un cambio de contexto, no es necesario guardar ningún estado del hilo ya que al disponer de un TSS propio, el estado se encuentra ahí mismo.
- Nos brinda la posibilidad de realizar procesamiento en paralelo en el caso de poseer más de un procesador sobre hilos de un mismo proceso.

Desventajas:

- Se sobrecargará la GDT con los descriptores de los distintos TSS de los hilos. Esto es un inconveniente ya que dicha tabla tiene un límite que es 8192 entradas en general.

Ambas administraciones tienen sus pros y sus contras. Hay dos puntos clave que nos dan la pauta de las situaciones en que nos conviene la implementación de una metodología por sobre la otra. Estas son al momento de la creación de un hilo y al momento de un cambio de contexto.

Si nos enfrentamos a un programa donde se utilizan pocos hilos y con alta concurrencia, resultaría más eficiente una administración de TSS por hilo debido a que por un lado al ser pocos hilos, el costo de la creación es bajo (proporcional a la cantidad) en relación a la ganancia frente a los cambios de contexto entre ellos que no requieren salvar y cargar sus estados.

Si nos encontramos ante un programa donde se ejecuta una cantidad considerable de hilos, corremos el riesgo de ocupar completamente la GDT si estamos en un nivel elevado de multiprogramación. Ya que no solo se almacenan descriptores de TSS, sino también Task Gate Descriptors en caso de utilizarlos, que apuntan al primero, entre otros descriptores que no hacen a nuestro trabajo de hilos.

Si analizamos el concepto de multiprocesamiento, hay que tener en cuenta que la ejecución de hilos de un mismo proceso en paralelo utilizando la primera metodología no es posible, es posible la concurrencia de hilos en paralelo pero de distintos procesos. Esto es así porque el descriptor del TSS contiene un flag de ocupado (Busy Flag) que indica cuando una tarea se encuentra en ejecución, por ende, cuando el segundo procesador quiera acceder al mismo TSS (con los datos del nuevo hilo) no será posible y se lanzará una excepción de protección general. En cambio, siguiendo la metodología de un TSS por hilo esto no es problema porque cada hilo dispondría de un Busy Flag propio en su descriptor de TSS.

Tipos de hilos:

Cuando hablamos de hilos nos enfrentamos a distintos tipos de implementaciones. Tenemos hilos a nivel de usuario (ULT - User Level Threads), hilos a nivel del Kernel (KLT - Kernel Level Threads) y una combinación de ambos (ULT + KLT) también llamada híbrida. Cada una de estas implementaciones tiene características diferentes. A continuación analizaremos cada una de ellas.

ULT (User Level Threads):

Bajo una implementación de hilos a nivel ULT, la administración y la planificación de éstos es realizada por la biblioteca de hilos de usuario. El Kernel no es conciente de la existencia de ellos, le es transparente. Por lo tanto, el intercambio en la ejecución de los hilos no requiere pasar a modo Kernel.

Ventajas:

- Se evita la sobrecarga por cambio de modo.
- Se puede ejecutar sobre cualquier sistema operativo.
- La planificación de los hilos la realiza la biblioteca de hilos de usuario.

Desventajas:

- Como los hilos son transparentes al Kernel, si un hilo se bloquea, se bloquea todo el proceso con todos los hilos que este contenga.
- No existe multitasking. Los hilos de usuario para el Kernel no existen, todo es un proceso, entonces solo lo puede asignar a un procesador.

KLT (Kernel Level Threads):

En una implementación de tipo KLT pura, todo el manejo de hilos es realizado por el Kernel. Tanto la administración como la planificación. Del lado del usuario solo se provee una interfaz para crearlos.

Ventajas:

- Esto permite el multitasking, es decir, múltiples hilos del mismo proceso en distintos procesadores ejecutando en paralelo.
- Si se bloquea un hilo se puede seguir con la ejecución de otro perteneciente al mismo proceso ya que el bloqueo no es a nivel proceso, sino a nivel de hilo.

Desventaja:

- Se produce una sobrecarga en lo que respecta a cantidad de operaciones cuando se trabaja con este tipo de hilos, ya que constantemente se requiere cambiar de modo, por ejemplo, cuando se quiere crear un hilo, se pasa del modo usuario a modo Kernel y luego a modo usuario otra vez.

Enfoque Combinado (ULT+KLT)

Este esquema permite combinar las ventajas de ULT y KLT disminuyendo algunas de las desventajas. En este enfoque la creación de los hilos se realiza en el espacio de usuario al igual que la planificación dentro de la aplicación. Los hilos que se crean del lado del usuario son mapeados en menor o igual número de KLTs.

Ventajas:

- El usuario crea el thread de nivel de usuario y casi toda la administración se hace a ese mismo nivel.
- Por otro lado el SO se maneja como con KLT puro, por lo que sigue permitiendo un alto nivel de concurrencia y también contamos con la posibilidad de ejecución en paralelo en caso de contar con múltiples procesadores.
- Cuando un ULT llama por ejemplo a una entrada salida, los threads que están dentro del mismo proceso no son bloqueados debido a que el SO hace distinción entre los mismos y los trata individualmente por medio de los hilos de nivel Kernel asociados.
- El desarrollador tiene la posibilidad de ajustar el número de KLT para cada aplicación y máquina para optimizar el resultado de su programa.
  Se obtienen mejores tiempos con respecto a KLT puro porque evitamos el cambio de contexto del nivel de usuario.

Posix Threads:

Históricamente los fabricantes de hardware han implementado sus propias versiones de threads. Estas implementaciones diferían sustancialmente entre ellas, haciendo de esta manera difícil para el desarrollador de software crear aplicaciones portables utilizando threads.

Para lograr obtener ventajas sobre las capacidades brindadas por los threads, se necesitaba una interfaz de programación estandarizada. Para sistemas UNIX, esta interfaz fue especificada por el estándar IEEE POSIX 1003.1c (1995). Las implementaciones que adoptan este estándar son llamadas POSIX threads o Pthreads. El estándar ha seguido avanzando con el tiempo, y su última versión es la IEEE std 1003.1 (2004)

Las funciones definidas en Pthreads pueden ser informalmente agrupadas en cuatro grandes grupos:

1.- Manejo de threads .
2.- Funciones de exclusión mutua.
3.- Variables de Condición.
4.- Sincronización.

En este informe describiremos funciones de los primeros dos grupos.

Manejo de Threads

pthread_create (thread,attr,start_routine,arg)

Inicialmente el programa main comprende un solo hilo por defecto. Todos los otros hilos deben ser explícitamente creados por el programador.

"pthread_create" crea un nuevo hilo y lo hace ejecutable. Esta rutina puede ser llamada tantas veces se desee, y desde cualquier parte del código.

Argumentos:

- Thread: Un identificador único y opaco para el nuevo thread retornado por la subrutina.
- Attr: Un objeto atributo opaco que puede ser usado para establecer los atributos del thread. Se puede especificar un objeto de atributos, o NULL para los valores por defecto.
- Start_routine: La rutina C que el thread va a ejecutar una vez que este sea creado.
- Arg: Un argumento simple el cual puede ser pasado a "start_routine". Se debe pasar como un puntero de tipo void. Se puede utilizar NULL si no hay argumento para pasar.

La cantidad máxima de threads que pueden ser creados por un proceso depende exclusivamente de la implementación.

Una vez creados, los threads son pares entre ellos, y estos pueden crear otros threads. No existe una jerarquía implícita o dependencia entre los mismos.

pthread_exit (status):

Hay varias formas en las que un Pthread puede ser eliminado:

- El thread retorna a su "start_routine"(la rutina principal para el thread inicial).
- El thread hace una llamada a la rutina "pthread_exit".

- El thread es cancelado por otro thread por medio de la rutina “pthread_cancel”.
- El proceso completo es eliminado debido a una llamada a ejecución o salida de subrutina.

“pthread_exit” es usado para explícitamente salir de un thread. Típicamente esta rutina es llamada después de que el thread ha completado su trabajo y no necesita seguir existiendo.

Si el main () termina antes del thread que este ha creado, y termina con “pthread_exit”, los otros threads van a continuar ejecutándose. De otra manera, estos terminarían automáticamente cuando el main termina.

El programador puede opcionalmente especificar un “estado de terminación”, el cual es guardado como un puntero a void para cualquier thread que pueda unirse al thread llamador.

Limpieza: La rutina pthread_exit no cierra archivos, cualquier archivo abierto dentro del thread permanecerá abierto luego de que el thread haya finalizado.

pthread_join (threadid,status) y **pthread_detach (threadid)**

Unirse es una de las maneras de lograr la sincronización entre hilos.

La rutina “pthread_join” bloquea la llamada a thread hasta que el thread especificado por threadid termina.

El programador tiene la posibilidad de obtener el target de estado de terminación del thread si este fue especificado en el target de la llamada del thread a pthread_exit.

Un hilo de unión puede coincidir con una llamada a pthread_join(). Es un error lógico intentar varias uniones en el mismo thread.

Cuando un hilo es creado, uno de sus atributos define si este es “unible” o “separable” (joinable o detached). Solo los hilos que son creados como joinables pueden ser unidos. Si un hilo es creado como detached, este nunca podrá ser unido.

El último draft de POSIX especifica que los threads deberían ser creados como joinables.

pthread_attr_init (attr) y pthread_attr_destroy (attr)

Por defecto, los threads son creados con ciertos atributos. Algunos de esos atributos pueden ser modificados por el programador por medio de las funciones de atributos de thread.

Estas dos funciones son utilizadas para inicializar o destruir los objetos atributos de los threads. Otras rutinas son usadas para consultar o establecer atributos específicos en los objetos atributos de los threads.

Para crear un thread como joinable o detached explícitamente, se usa el argumento “attr” en el “pthread_create()”. Los 4 pasos típicos del proceso son:

1. Declarar la variable atributo del tipo de data “pthread_attr_t”

2. Inicializar la variable atributo con “pthread_attr_init”
3. Especificar el estado del atributo detached con “pthread_attr_setdetachstate()”
4. Una vez listo, liberar los recursos de biblioteca usados por el atributo con “pthread_attr_destroy()”.

La rutina “pthread_detach()” puede ser usada para explícitamente separar un hilo aunque este haya sido creado como joinable.

Se recomienda que si el hilo requiere unión, se considere crearlo explícitamente como joinable. Esto permite portabilidad, ya que no todas las implementaciones pueden crear hilos como joinable por defecto.

Si se sabe con anticipación que el hilo nunca necesitará unirse con otro hilo, se considere crearlo con estado “detached”. De esta manera algunos recursos de sistema pueden ser liberados.

Funciones de exclusión mutua

pthread_mutex_init (mutex,attr)

Las variables mutex deben ser declaradas con el tipo “pthread_mutex_t”, y deben ser inicializadas antes de ser usadas. Hay dos maneras de inicializar una variable mutex:

1. Estáticamente, cuando es declarada. Por ejemplo
   pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
2. Dinámicamente, con la rutina “pthread_mutex_init()”. Este método permite establecer atributos de objetos mutex, attr.

El mutex está inicialmente desbloqueado.

pthread_mutex_destroy (mutex)

Esta rutina es usada para liberar un objeto mutex el cuál ya no es necesario.

pthread_mutexattr_init (attr) y **pthread_mutexattr_destroy (attr)**

Ambas rutinas son usadas para crear y destruir los atributos de los objetos mutex respectivamente.

pthread_mutex_lock (mutex)

Esta rutina es usada por un thread para lograr un bloqueo en la variable mutex especificada. Si este mutex ya estaba bloqueado por otro thread, esta llamada bloqueará al thread llamador hasta que el mutex se desbloquee.

pthread_mutex_trylock (mutex)

La función de "trylock" intentará bloquear un mutex. Sin embargo, si el mutex ya está bloqueado, la rutina retornará inmediatamente con el código de error "busy". Esta rutina puede ser útil para prevenir condiciones de "deadlocks".

pthread_mutex_unlock (mutex)

Esta función desbloqueará un mutex si es llamada por el thread que la posee. Se llama a esta rutina luego de que un thread ha completado su uso de datos protegidos, si otros threads van a adquirir el mutex para su propio trabajo con el área protegida de datos. Se producirá un error si:

- Si el mutex ya estaba desbloqueado
- Si el mutex le pertenece a otro thread.

3 Implementación:

Inicialmente se diseñó una implementación basada en el concepto de **TSS única por proceso**. Luego de realizada la investigación acerca de las ventajas y desventajas entre esta forma y la de **un TSS por hilo,** se decidió cambiar el diseño de implementación.

El diseño inicial de implementación requería de una estructura de datos para almacenar la threadTSS (una TSS pero solo con los datos que varían por cada hilo). A nivel de planificación se requería que al momento de realizar un cambio de contexto entre dos hilos, se realizara un "thread context switch", el cual implicaba salvar los datos de la TSS del hilo saliente en su estructura threadTSS, y luego cargar en la TSS del proceso los datos de la threadTSS del hilo entrante.

Todo esto fue descartado y se pasó a tener una TSS por cada hilo, a continuación se detallará como es el diseño propuesto.

3.1 TSS por Hilo:

Nivel Estructural:

La estructura de un proceso varía cuando trabajamos con hilos. A la estructura del PCB (Process Control Block) se le agregará una lista dinámica de los distintos TCB (Thread Control Block) correspondiente a cada hilo del proceso. Cabe aclarar que todo proceso dispondrá mínimamente de un hilo que corresponderá al proceso en sí.

Al PCB se le agregan:

- iUltimoHiloEjecutado.
- iAnteriorHiloEjecutado.

La variable "iAnteriorHiloEjecutado" se utiliza para tratar el manejo de la excepción de CPU7, la cual se detallara mas adelante.

La variable iUltimoHiloEjecutado es necesaria para la planificación de los hilos.

Al PCB se le quita :

- ulLugarTSS.
- uiIndiceGDT_TSS.

El TCB estará conformado por:

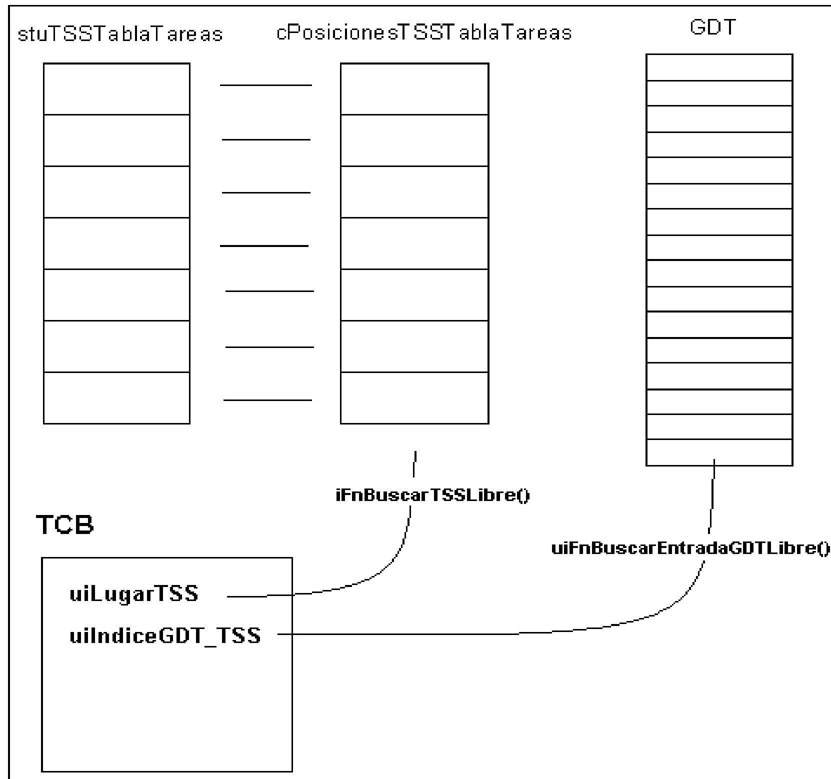
- ID de hilo.
- PID del proceso al que pertenece.
- Estado del hilo (Ejecutando / Listo / Bloqueado).

- ulLugarTSS (posición dentro de stuTSSTablaTareas en la que se encuentra la TSS del hilo)
- uiIndiceGDT_TSS (posición en la GDT donde se encuentra el descriptor de la TSS del hilo)
- stNombre[25]

El hecho de haber agregado una TSS por cada Hilo de kernel genera un gran inconveniente. Antes de esta implementación los vectores pstuPCB (vector de los PCB del sistema) y stuTSSTablaTareas (vector con las TSS de los proceso del sistema) compartían el índice. Si teníamos la posición XXX en pstuPCB, sabíamos que la TSS de ese proceso estaba ubicada en la posición XXX de stuTSSTablaTareas, ahora al haber un TSS por hilo, hay mayor cantidad de elementos en stuTSSTablaTareas que en pstuPCB, por lo tanto se modifica completamente la forma de referenciar a la posición de la TSS de un proceso en particular.

Se realiza una recorrida de todo el código de SODIUM identificando los lugares donde se tomo el supuesto de que una posición de pstuPCB correspondía a la misma posición de stuTSSTablaTareas y se modifico en cada caso para que apunten a la TSS deseada, para realizar dicha tarea se pasaron las variables ulLugarTSS y uiIndiceGDT_TSS a la estructura del TCB ya que ahora corresponden al hilo.

La organización implementada para esta metodología la podemos visualizar en el siguiente grafico:



Disponemos de un nuevo vector “cPosicionesTSSTablaTareas” de relación uno a uno con stuTSSTablaTareas, utilizado para saber que TSS se encuentra disponible. Anteriormente, la relación uno a uno era con el vector de procesos, no era necesario chequear disponibilidad sobre las TSS.

Ahora, los campos índices para la TSS, tanto en el vector de TSS como de la GDT, son propios del TCB y no del proceso. Para obtenerlos se utilizan dos funciones. Para la GDT se continúa utilizando la misma función, pero ahora para el lugar de la TSS en stuTSSTablaTareas se utiliza iFnBuscarTSSLibre.

Modificaciones realizadas al momento para esta implementación:

MAIN.C:

Modificado el tamaño asignado a stuTSSTablaTareas.
Creado el vector “cPosicionesTSSTablaTareas”.

GDT:

iFnCrearPCB
iFnDuplicarProceso
iFnReemplazarProceso
iFnEliminarProceso
iFnCrearProceso
iFnInstanciarTrabajadorKernel

Nuevas Funciones:
iFnCrearTCB
iFnDuplicarHilos
iFnBuscarTSSLibre
vFnLiberarTSS
iFnBuscaPosicionHilo

SYSCALL.C:

lFnSysTime
iFnSysSumar
lFnSysIdle
lFnSysPtrace

Nuevas Funciones:
lFnSysPthreadCreate

SCRAP.C

vFnMenuStack
vFnMenuTSS
vFnMenuPs
vFnMenuDesc

SYSTEM.C

vFnExcepcionCPU7

RR.C

uiFnAlgunHiloListo

3.2 Planificación:

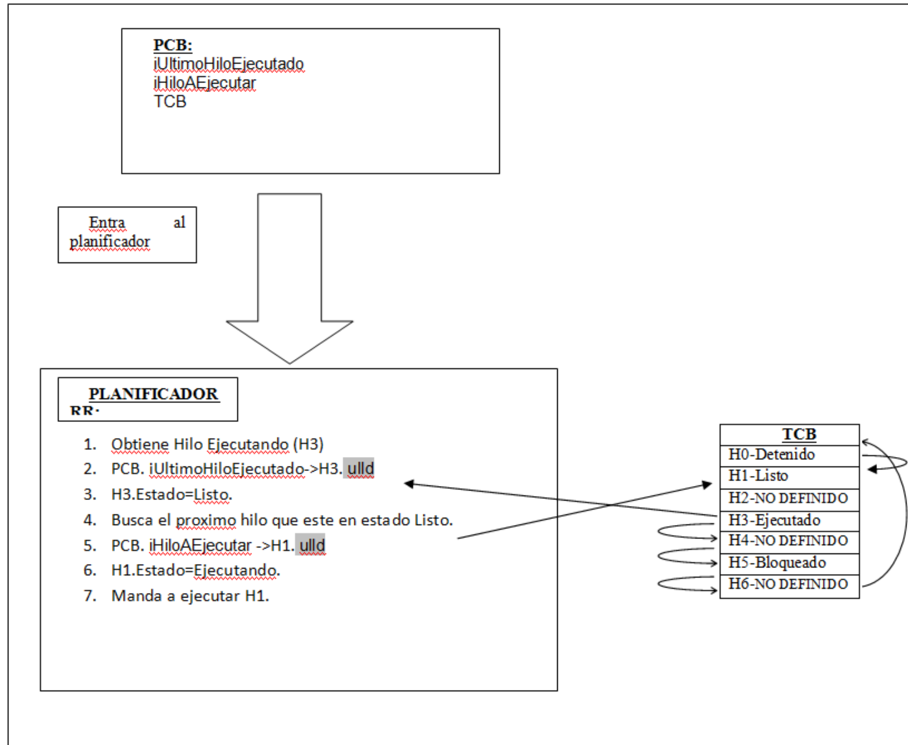
Actualmente el planificador de SODIUM funciona con un nivel de Round Robin (RR). Para la planificación de hilos se anexará a este un nuevo nivel de RR anidado al anterior.

Los pasos en la planificación serán los siguientes:

El planificador de SODIUM designa cual es el hilo a ejecutar. Esto lo realiza buscando en primer lugar dentro de la lista de PCBs al primer proceso que se encuentre en estado Listo. Una vez encontrado, se recorre la lista de TCB perteneciente a él hasta encontrar el primer hilo en estado Listo. Si no encuentra ninguno en estado listo llama a la tarea nula.

Al implementarse el concepto de Thread Kernel (KLT), era necesario llevarlo más allá de los procesos. Por este motivo se incorporo un segundo nivel de planificación(a nivel de hilos). Este nivel tiene una lógica muy parecido que la de proceso, es decir, el hilo que se está ejecutando lo pasa a listo y guarda su ID en un registro del PCB (iUltimoHiloEjecutado), para que posteriormente cuando se reanude el proceso se continúe desde dicho hilo. Luego busca en el TCB el próximo hilo que se encuentra en listo para ejecutarlo, cambiando su estado a Ejecutando y guardando en un registro del PCB (iHiloAEjecutar) el ID del nuevo hilo que se esta ejecutando.

Una vez definido el nuevo hilo que entra en ejecución, se ejecuta el cambio de contexto como se venía haciendo antes a la TSS del hilo que le toca ejecutar.


PCB:
iUltimoHiloEjecutado
iHiloAEjecutar
TCB
Entra al planificador
PLANIFICADOR RR:
1. Obtiene Hilo Ejecutando (H3)
2. PCB. iUltimoHiloEjecutado->H3. uIId
3. H3.Estado=Listo.
4. Busca el proximo hilo que este en estado Listo.
5. PCB. iHiloAEjecutar ->H1. uIId
6. H1.Estado=Ejecutando.
7. Manda a ejecutar H1.
TCB
H0-Detenido
H1-Listo
H2-NO DEFINIDO
H3-Ejecutado
H4-NO DEFINIDO
H5-Bloqueado
H6-NO DEFINIDO

3.3 Funciones implementadas de Posix para hilos:

Funciones para el manejo de hilos implementadas:

pthread_create
pthread_exit
pthread_join
pthread_sched_yield

Todas las funciones fueron agregadas en pthread.c (usr/lib), a continuación se detalla el procedimiento para la ejecución de cada una.

Para explicar dicho procedimiento se tomará como ejemplo la implementación de pthread_exit.

En el archivo pthread.c (usr/lib) se desarrolló la función pthread_exit, la cual llama a la función lFnPthreadExit en el archivo libsodium.c (usr/lib). Esta segunda función es quién llama a un syscall con el parámetro "__NR_pthread_exit". Este número de syscall tuvo que ser definido en syscall_def.h (include/common).

Luego en system.c (kernel) existe una funcion llamada "lFnHandlerSyscall" donde para cada número de syscall definido se llama a la función correspondiente de manejo de esa syscall. En nuestro ejemplo es "lFnSysPthreadExit" desarrollada syscall.c (kernel).

Quedando el camino definido de la siguiente manera:
pthread.c -> libsodium.c -> system.c (case) -> syscall.c (donde finalmente se encuentra la función que realiza los cambios necesarios)

Syscall.h/.c:

```
long lFnSysPthreadCreate(unsigned long ulFunction);
long lFnSysPthreadJoin(int ulIdHiloAEsperar);
long lFnSysPthreadExit(void);
long lFnSysPthreadYield(void);
```

System.c:

```
case( __NR_pthread_create): return lFnSysPthreadCreate((unsigned long) ebx);
case( __NR_pthread_join): return lFnSysPthreadJoin((int) ebx);
case( __NR_pthread_exit): return lFnSysPthreadExit();
case( __NR_pthread_yield): return lFnSysPthreadYield();
```

Pthread.c:
Biblioteca de funciones de Posix Threads para la utilización por parte del usuario.

pthread_create
pthread_join
pthread_exit
pthread_sched_yield

Funciones anexas necesarias para esta implementación:

GDT.c:
iFnCrearHilo

pthread_create

Inicialmente todo proceso consta de un hilo por defecto. Todos los otros hilos deben ser explícitamente creados por el programador.
"pthread_create" crea un nuevo hilo y lo hace ejecutable. Esta rutina puede ser llamada tantas veces se desee, y desde cualquier parte del código. Una vez creados, los threads son pares entre ellos, y estos pueden crear a su vez otros threads.

pthread_exit

Es usado para explícitamente salir de un thread. Típicamente esta rutina es llamada después de que el thread ha completado su trabajo y no necesita seguir existiendo.

Identifica cuál es el hilo que se está ejecutando y cambia su estado a "HILO_NO_DEFINIDO".
Luego para cada hilo que estaba esperando su finalización para continuar (caso de un pthread_join hacia este), cambia sus estados a "HILO_LISTO".

pthread_join

Join es una de las maneras de lograr la sincronización entre hilos. La rutina "pthread_join" bloquea la ejecución del hilo hasta que el thread especificado por parametro termina.

Identifica el hilo que se está ejecutando y setea su estado como "HILO_DETENIDO".
Luego guarda en el campo "ulIdHiloQueEspera" el id del hilo pasado por parámetro para que vuelva a pasar a listo una vez que este finalice.

pthread_sched_yield

Esta función se utiliza básicamente para voluntariamente ceder la ejecución del procesador desde el hilo actual, posicionándose en la lista de hilos listos para ser llamado nuevamente cuando el planificador vuelva a brindarle la ejecución.

Simplemente llama al planificador. Al igual que en la función manejada a nivel proceso. Para explicar cómo aseguramos que esto es suficiente tenemos que sumergirnos un poco en planificación y cómo se realiza un Context-switch en SODIUM.

Para nuestra implementación de hilos a nivel kernel, estamos usando una TSS por hilo en vez de una TSS por proceso. El procesador no trabaja ni con hilos ni con procesos, sino con tareas, por lo que luego de nuestra implementación, la manera en que el procesador manejará los hilos en un task-switch, será muy similar a la anteriormente utilizada para los procesos.

Cuando se ejecuta el planificador este deshabilita las interrupciones, luego busca qué hilo de qué proceso es el próximo listo para entregarle la ejecución del procesador. Una vez detectado, se ejecuta un jmp hacia la posición del descriptor de TSS.

Cabe destacar que el procesador guarda automáticamente solo los registros de propósito general, sin tener en cuenta por ejemplo los registros de punto flotante (FPU), pero en SODIUM esta situación está salvada de la siguiente manera:
En system.c existe la función vFnExcepcionCPU7 (invocada por vFnExcepcionCPU7_Asm desde system_asm.asm), la cual se encarga de salvar el contexto de la FPU en la TSS de la última tarea que la utilizó y cargar la FPU de la nueva tarea.

<u>Análisis de Syscalls relacionadas con cambios de estado:</u>

LfnSysExit

La llamada al sistema LfnSysExites muy sencilla. Sirve para finalizar la ejecución del programa actual. Eso sí, hemos de tener cuidado de liberar todos los recursos que el proceso estaba ocupando, especialmente la memoria y el objeto Thread que está a cargo del proceso.
La llamada LfnSysExit recibe un parámetro que es el código de salida del programa. Este código de salida puede ser consultado por otros procesos para averiguar si el proceso actual ha finalizado correctamente o bien ha tenido algún problema en su ejecución. Por ejemplo, en UNIX los programas que finalizan correctamente invocan a la llamada al sistema exit(0), y los que finalizan incorrectamente llaman a LfnSysExit con un valor no nulo.
De acuerdo a esto podemos ver que cuando invocamos a exit estamos decidiendo finalizar la ejecución actual y liberando recursos, entre los cuales se encuentran los hilos. Al pasar el proceso actual a 'no definido' lo que se hace es decir

que se esta liberando sus recursos, por otro lado, su espacio será considerado como libre para futuros usos por parte del planificador. Con lo cual no es necesario que se baje al nivel de los hilos para que ir dándolos de baja a cada uno dado que se tomara el pcb en su totalidad a la hora de reasignarse el recurso.

lFnSysKill

El lFnSysKill función solicita que la señal sig se entregará al subproceso especificado. La señal que se envía es especificada por el SIG y es cero o una de las señales de la lista de las señales se definen en el archivo de cabecera <include/common/signal.h>. Si sig es menor a cero, la comprobación de errores se lleva a cabo, pero no hay señal es enviada al subproceso de destino, caso que tambien se da si el proceso se encuentra en 'estado zombie'.

Un hilo puede utilizar lFnSysKill para enviar una señal a sí mismo. Si la señal no es bloqueada o ignorada, por lo menos una señal pendiente desbloqueado se envía al remitente antes del regreso de lFnSysKill. Si no hay otras señales de desbloquear la espera, la señal enviada es sig. El lFnSysKill () API no altera en absoluto el efecto o el alcance de una señal.

Por ejemplo, el envío de una señal SIGKILL a un tema utilizando lFnSysKill termina el proceso, simplemente no es el subproceso de destino. SIGKILL se define para poner fin a todo el proceso, independientemente del hilo que se entrega a, o la forma en que se envía.

LfnSysRead

Read() intenta leer hasta 'count' bytes desde el archivo descriptor 'fd'en el búfer que comienza en buf.

Si count es cero, read () devuelve cero y no tiene otros resultados. Si count es mayor que SSIZE_MAX, el resultado es indefinido.

Esta función es 'administrada' por un proceso que esta activo mientras se intenta ingresar datos por medio de teclado. En caso de que se nos llene el buffer de teclado este también se bloquea y en el momento de un nuevo ingreso o vaciado de buffer se vuelve a poner a activo. En nuestro caso se usa la idea de que al ser tan repetitivo el ingreso desde teclado, se usa un proceso liviano para que al momento de restaurar contexto sea una tarea mas rápido y eficiente. En cuanto a las modificaciones en la función se pasa a espera tanto al proceso como al hilo en ejecución en el momento de una de las condiciones de bloque. Por otro lado se mantiene constantemente en la pcb del proceso cual es el hilo que se bolquee en ese momento. Estos dos, hilo y proceso, pasan a estar activos desde la función vFnManejadorTecladoShell en kernel/drivers/teclado.c en el momento en que se capta que el proceso que maneja a read esta en espera o se esta a la espera de teclado.

LfnSysIdle

Esta función llama a la tarea nula, por lo que tanto el proceso como el hilo que se están ejecutando deben pasar a “LISTO” para volver a ser llamados cuando el planificador les ceda su turno.

lFnSysWaitPid

WaitPid es una función que espera la finalización de un hijo en particular, esta implementación es a nivel proceso y no debe ser bajado a nivel hilos. En caso de necesitar que un hilo espere a otro en particular, se puede utilizar la función pthread_join de la biblioteca de POSIX.

En conclusión solo encontramos necesidad de modificar LfnSysRead y LfnSysIdle.

4 Conclusión:

A lo largo de la investigación se analizaron muchos aspectos en lo que es la ejecución de tareas a nivel procesador, proceso y sobre todo hilos que de hecho fue el punto principal. Se analizaron los distintos tipos, sus ventajas y desventajas. Esto nos dio la posibilidad de abrirnos camino hacia un esquema, un diseño de implementación de KTL a nivel funcional y procedimental, definición de sus estructuras y modo de planificación de ejecución para ser llevado a cabo sobre SODIUM.

4.1 Tareas Pendientes:

4.1.1 Hilos de Kernel:

Estado actual:

En la implementación del presente trabajo quedó pendiente lograr el correcto funcionamiento de la función pthread_create, que se encarga de crear un nuevo hilo.

Al crear un nuevo hilo se le asignan todos los datos a su TSS igual que como se realizaba anteriormente con procesos. Por un lado se cambia el EIP (que se usa la posición de la dirección donde esta la función que va a ejecutar el hilo), y por otro se le da un offset al EBP y al ESP (ambos se ponen en la misma posición) para que la pila del nuevo hilo no pise los datos de pila de los demás. Finalmente el resto de la TSS de ambos hilos queda igual.

En ejecución surgen los siguientes inconvenientes:

Al iniciar la ejecución del segundo hilo lo ejecuta, pero al terminar, es decir al ejecutar todas las instrucciones y querer volver al hilo que lo llamo, nos da un error de protección general (Excepción 13).

El otro inconveniente es que al intentar ejecutar ambos hilos, el planificador los envía a ejecución pero siempre se ejecuta el 2do hilo, por más que a ejecución se envió la TSS del otro.

Creemos que en algún punto del circuito se están perdiendo los valores del EBP o ESP, o se están pisando.

El probable lugar donde se este perdiendo este valor es durante la rutina de atención de interrupción de reloj (vFnHandlerTimer_Asm, en sodium_asm.asm), al pasar por las Macro ENTRADA_STACK_KRNL y SALIDA_STACK_KRNL.

Investigando este error descubrimos que el contenido de stuTSSTablaTareas, que en teoría debería ser el valor de la TSS del proceso en el instante en el que se lo saco de ejecución, en realidad es el estado de la TSS del planificador al momento de hacer el context-switch, lo cual a nuestro entender es un error sobre lo que debería tener guardada dicha estructura de datos, esto dificulta mucho realizar un seguimiento de los valores del nuevo hilo.

El planificador de dos niveles según nuestras pruebas esta funcionando correctamente, al estar los dos hilos del mismo proceso en ejecución pudimos observar que enviaba a los dos hilos a ejecución en forma alternada, pero debido al inconveniente mencionado anteriormente, no se ejecuta el contenido del hilo 0, pero si se envía el descriptor de su TSS a ejecución.

En la corrección de las syscall que eran bloqueantes a nivel proceso, se las alteró para su implementación con hilos. Se realizaron pruebas sobre dichas modificaciones, pero no hay forma de estar seguros de su total funcionamiento hasta que se puedan tener dos hilos en ejecución de un mismo proceso (esto es, cuando funcione pthread_create).

Lo mismo ocurre con las funciones de threads pthread_join, pthread_exit, pthread_sched_yield. Si bien se pudieron realizar algunas pruebas y parecen estar funcionando correctamente no hay forma de asegurar el funcionamiento hasta realizar una prueba real sobre dos hilos en ejecución.

La modificación general que hubo que realizar sobre todo SODIUM para implementar el concepto de un TSS por hilo esta funcionando en su totalidad al utilizar memoria en modo particionado fijo y variable. En modo paginado, nuestra implementación presenta problemas al intentar crear SODSHELL. Quedaría pendiente hacer un chequeo sobre el funcionamiento de SODIUM con paginación e hilos.

Necesidad actual:

- Definir un segmento de memoria fijo para cada stack de cada hilo.
- Hacer que SODIUM maneje LDT.
- Crear una LDT por proceso donde se almacenen los descriptores de los stack segment de cada hilo.

En conclusión, la idea para el manejo de los stacks es que cada uno tenga uno propio en segmentos diferentes de la memoria, y no sobre un mismo segmento a distintos offsets como se encuentra ahora. Esto nos da la ventaja de que siempre se tendrá el tamaño reservado justo para la cantidad de hilos que tenga el proceso. De esta manera no desperdiciamos memoria.

Estos segmentos serán accedidos a través de sus descriptores que estarán almacenados en la LDT del proceso. Como la LDT se direcciona a través de la TSS, todos los hilos de un mismo proceso deberán apuntar a la misma LDT.

4.1.2 Funcionamiento de TSS:

Estado actual:

La situación actual de SODIUM en la ejecución de Tareas la mostramos a continuación siguiendo una ejecución ficticia:

1. T1 (Tarea uno) en ejecución.
2. Int 0x20 (Interrupción de reloj).
3. Ejecución de vFnHandlerTimer_Asm.
4. Se guarda en el stack de T1 el estado del procesador en su última instrucción previa a la interrupción.
5. Entrada al planificador.
6. Envío a ejecución de T2. (jmp TSS2)

Lo que sucede es que la TSS1 (TSS de T1) tiene almacenado el estado del procesador que este tenía en 6 y en el stack se encuentra el estado del procesador en la ultima instrucción verdadera de T1.
Error: La TSS debe almacenar los valores de la T1 y no el stack.

7. T2 en ejecución.
8. Int 0x20 (Interrupción de reloj).
9. Ejecución de vFnHandlerTimer_Asm.
10. Se guarda en el stack de T2 el estado del procesador en su última instrucción previa a la interrupción.
11. Entrada al planificador.
12. Envío a ejecución de T1. (jmp TSS1)

Lo mismo sucede con TSS2, tiene almacenado el estado del procesador que este tenía en 12 y en el stack se encuentra el estado del procesador en la ultima instrucción verdadera de T2.

Ahora surge el siguiente interrogante. ¿Cómo es que al realizar el paso 12 continúa T1 desde donde estaba? La respuesta es simple, no lo hace. Al ejecutar T1 se vuelve al planificador luego del paso 6, en la instrucción siguiente. Se continúa la ejecución volviendo por cada llamada realizada por vFnHandlerTimer_Asm. Éste, al final, por medio de una macro, quita del stack el estado del procesador (real de T1) y los carga en los registros, para de esta manera continuar con la ejecución de T1.

(Para ver con mayor detalle los stacks donde se almacenan los registros del procesador y como funcionan las macros de vFnHandlerTimer_Asm que se encargan de ello, en el Anexo 1 tenemos un seguimiento realizado.)

Necesidad actual:

- TSS almacenando verdaderamente el estado de la tarea.
- Eliminación de Stacks temporales.

Soluciones propuestas:

<u>Solución fallida:</u>

<u>Opción 1:</u>

1. T1 (Tarea uno) en ejecución.
2. Int 0x20 (Interrupción de reloj).
3. Resguardar en TSS1 el estado de T1 previo a la interrupción.
4. Entrada al planificador.
5. Se leen los registros de TSS1 y se los carga en el procesador por medio de inline assembler.
6. jmp TSS2 (Envío a ejecución de T2)

La propuesta surgió como una solución provisoria para no generar un cambio global. El punto 5 era necesario ya que cuando pasamos por el punto 6 se guarda en TSS1 el estado del procesador, y este sin pasar por 5 contiene el estado en planificación.

El punto 3 lo obtenemos del stack definido para la tarea

<u>Problema:</u> El registro EIP al no ser accesible por el programador directamente no puede ser modificado manualmente. Por ende, se tienen todas las TSS bien pero con EIP no válido.

<u>Solución Alternativa:</u>

<u>Opción 2:</u>

Esta opción se basa en la premisa de que el planificador debe correr dentro de una Tarea, con TSS propia y Segmento de Stack propio.

1. T1 (Tarea uno) en ejecución.
2. Int 0x20 (Interrupción de reloj).
3. Resguardar en un auxiliar de tipo stuTSS el estado de T1 previo a la interrupción.
4. jmp TSSPlanificador (Envío a ejecución TPlanificación).
5. Guardar en TSS1 el contenido del auxiliar.
6. Ejecución de HandlerTimer y Planificador

El paso 3 y 4 son necesarios ya que frente al jmp al planificador se genera el cambio de contexto y se pisa la TSS1, por eso usando un auxiliar y cargando los datos en mi TSS1 a mano luego del cambio del contexto nos salvamos de este problema.

Atención: Es necesario analizar en que estado queda la habilitación de las interrupciones frente a los cambios de contexto

Inconveniente: Frente a multiprocesamiento, esto no sirve, ya que puede pisarse el auxiliar antes de ser utilizado. Debería analizarse la posibilidad de tener más de un auxiliar y controlarlo mediante algún flag dentro de su estructura.

Solución Final:

La siguiente solución, a nuestro criterio es la correcta, la más transparente que permite un flujo natural de la ejecución del planificador frente a interrupciones de reloj.

Opción 3:

Como ya sabemos, dentro de nuestra IDT podemos contener:

- Task-Gate descriptor
- Interrupt-gate descriptor
- Trap-gate descriptor

Por lo tanto, si para la interrupción 0x20 que es la del reloj nosotros en vez de definir un Iterrupt-gate o un Trap-gate como esta definido en la función *vFnIniciarIDT*, creamos un Task-gate descriptor con la TSS del planificador, nos estaríamos asegurando de esta manera que frente a la interrupción se genere el cambio de contexto automáticamente sin nosotros realizar ningún cambio, dejando que de esta manera se almacene en la TSS de la tarea saliente el verdadero valor de ejecución.

Tarea de planificador:

Tanto para la opción 2 como para la opción 3 se propuso crear una Tarea que se encargue de ejecutar *vFnHandlerTimer* que se encarga de la administración del *Timer*, para luego realizar la llamada al planificador. Este handler del timer debería tener toda su funcionalidad inmersa en una iteración indefinida, para que de esta manera, cuando se vuelve a ejecutar esta tarea, y vuelva por el *iret* del a llamada, llegue al final y vuelva a iniciar.
Usamos este mecanismo de iteración indefinida y no una nueva llamada, porque cada llamada guarda en el stack dos registros para realizar el "*stack frame*", lo que puede terminar sobrecargando el stack innecesariamente.

References

- Operating Systems 6th Edition – William Stallings
- Fuente: "POSIX Threads Programming", Guía del programador de hilos POSIX, web: https://computing.llnl.gov/tutorials/pthreads/
- Fundamentos de Sistemas Operativos 7ma Edición – Silberschatz Abraham
- Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1