UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Ingeniería en Sistemas de Información

[Ráfaga]

La herramienta más eficaz de depuración sigue siendo una cuidadosa reflexión, junto con las declaraciones de impresión juiciosamente colocadas. — Brian W. Kernighan (1979)

Cátedra de Sistemas Operativos

Trabajo Práctico Cuatrimestral

- 2C2014 -
Versión [1.0.a]

Tabla de Contenidos

Introducción

El trabajo práctico consiste en simular ciertos aspectos de un sistema multiprocesador con la capacidad de ejecutar un lenguaje creado para esta ocasión. Este sistema planificará y ejecutará estos códigos (en adelante, “Procesos”) controlando sus solicitudes de memoria, llamadas al sistema y administrando los accesos a recursos, como los semáforos compartidos.

Objetivos del Trabajo Práctico

Mediante la realización de este trabajo se espera que el alumno:

- Adquiera conceptos prácticos del uso distintas herramientas de programación (API) que brindan los sistemas operativos modernos.
- Entienda aspectos del diseño de un sistema operativo moderno.
- Afirme diversos conceptos teóricos de sistemas operativos mediante la implementación práctica de algunos de ellos.
- Se familiarice con técnicas de programación de sistemas, como el empleo de makefiles, archivos de configuración o archivos de log.
- Conozca con grado de detalle la operatoria de Linux mediante la utilización de un lenguaje de programación de relativamente bajo nivel como C.

Principales temas a desarrollar

La siguiente es una lista de los temas de la materia incluidos en el trabajo práctico que se espera que todos los alumnos dominen luego del desarrollo del mismo.

- ➢ Algoritmos de Planificación.
- ➢ Estados de un proceso en el sistema y sus transiciones.
- ➢ Estructura de un proceso/hilo.
- ➢ Mecanismos IPC.
- ➢ Programación concurrente: procesos e hilos.
- ➢ Sincronización y semáforos.
- ➢ Gestión de Memoria: Segmentación Paginada, Memoria Virtual, Paginación bajo demanda, swapping, algoritmos de reemplazo de páginas, asignación de memoria dinámica.

Características

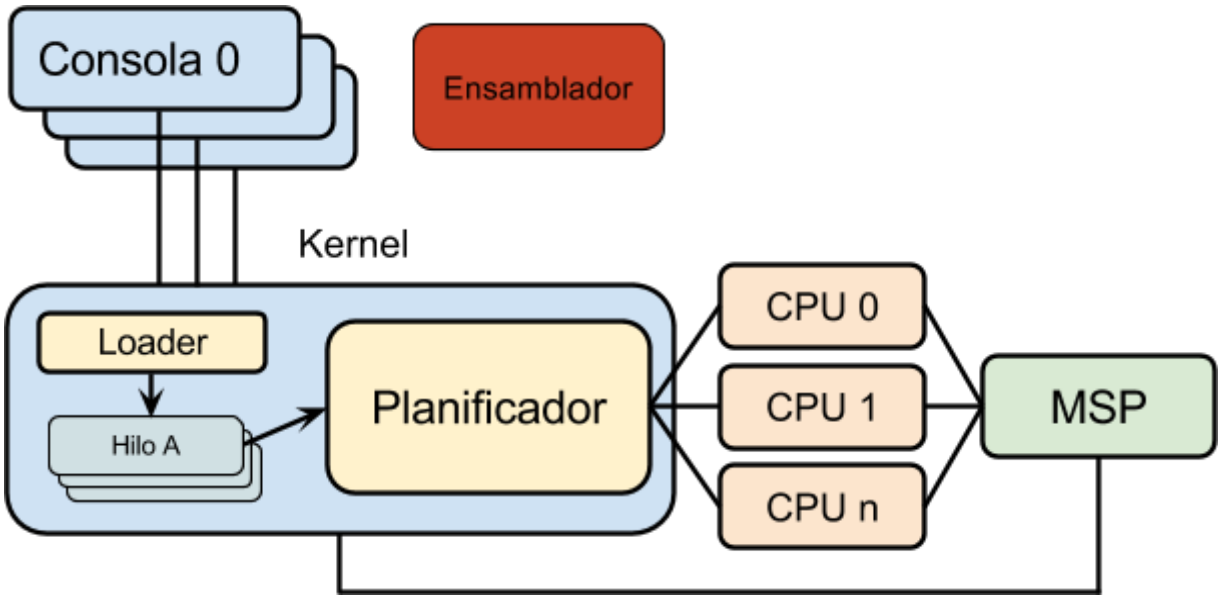
- Modalidad: grupal (5 integrantes) y obligatorio
- Fecha de entrega: Sábado 29 de Noviembre
- Recuperatorios: Sábados 6 y 20 de Diciembre
- Lugar de evaluación: Laboratorio de Medrano

Aclaraciones

Debido al fin académico del trabajo práctico, los conceptos reflejados son, en general, versiones simplificadas o alteradas de los componentes reales de hardware y de sistemas operativos modernos, a fin de resaltar algún aspecto de diseño. En algunos casos los aspectos no fueron tomados de manera literal, por lo que invitamos a los alumnos a leer las notas y comentarios al respecto que haya en el enunciado, como así también a reflexionar y discutir con sus compañeros, ayudantes y docentes al respecto.

Siempre que en el enunciado se lea la palabra socket, se refiere a los sockets STREAM tipo AF_INET.

Arquitectura del sistema



Ensamblador

Este ejecutable es brindado por la cátedra[1] como una forma de transformar código **ESO** (*Ensamblador de Sistemas Operativos*) a una abstracción de código de máquina (denominado *Código Ensamblado*), ejecutable por los procesos CPUs, explicados más adelante en este documento.

Los programas ESO son compilados por el Ensamblador al formato **BESO** *(Bytecodes[2] del Ensamblador de Sistemas Operativos)*, y deberán ser ubicados en algún directorio accesible por los procesos Consola (también descritos en este documento) para que éstos puedan enviarlos a ejecutar.

[1] Link al repositorio: https://github.com/sisoputnfrba/so-ensamblador

[2] A este código de máquina se lo conoce como “archivo con código objeto”

Proceso Kernel

El proceso Kernel es el proceso principal del sistema. Recibirá el código de los Programas a través de las Consolas, y planificará sus Hilos a través de los diversos estados del sistema, siendo responsable de enviarlos a la CPU adecuada cuando corresponda. Dado que los Procesos, durante su funcionamiento, pueden crear nuevos Hilos de ejecución, el Kernel planificará únicamente Hilos (entendiendo a un Programa que no crea ningún Hilo como un Programa mono-hilo).

Al iniciar el proceso Kernel, éste deberá intentar conectarse a la MSP (Memoria de Segmentación Paginada) para poder reservar y escribir los segmentos de los Programas.

Loader (Creación del TCB)

Al recibir la conexión de una nueva Consola para ejecutar un Proceso, el subsistema Loader creará la estructura TCB *(Thread Control Block)*, asignándole un identificador único del Proceso (PID) y un identificador único de Hilo (TID). Luego podrá reservar en la MSP los segmentos necesarios para iniciar el Programa[3]:

- Segmento de Código, que contendrá el código del BESO tal y como está almacenado en disco
- Segmento de Stack, con un tamaño fijado por archivo de configuración

Tras reservar cada Segmento, el Loader recibirá de la MSP la dirección base del mismo (es decir, la primera dirección utilizable del segmento) que servirá también como su identificador. Luego de reservados ambos, escribirá el código del Programa en el segmento correspondiente.

En caso de que al reservar los segmentos la MSP retornará un error por no tener memoria disponible, se abortará la creación del Programa, notificando a la Consola correspondiente y finalizando la conexión. Luego de esta situación, el Kernel debe continuar funcionando normalmente y seguir aceptando conexiones de nuevas Consolas.

El siguiente paso de una carga de Proceso exitosa será inicializar el Program Counter (Puntero de Instrucción) del Proceso, la base y el cursor del stack y sus registros de programación. Estos serán 5 registros identificados por una letra mayúscula (’A’, ’B’, ’C’, ’D’, y ’E’), cada uno de 4 bytes de tamaño, conteniendo números enteros con signo.

[3] En algunas implementaciones reales, cierta información (como el PID, y el puntero al Segmento de Código) que es compartida por todos los hilos se guarda en una estructura Process Control Block (PCB) en vez del TCB.

Con esta información el Loader podrá llenar la estructura del TCB del primer Hilo del Proceso:

Campo	ID	Tipo de datos	Información
PID		Numérico	Identificador del Proceso
TID		Numérico	Identificador del Hilo del Proceso
KM		Booleano	Indicador de Modo Kernel *(Kernel Mode)*. Desactivado para todos los Hilos de Programas de usuario.
Base del Segmento de Código	M	Numérico	Primer dirección utilizable del segmento de código del Programa
Tamaño del Segmento de Código		Numérico	Tamaño en bytes del segmento solicitado a la MSP para alojar el código ensamblado
Puntero de Instrucción	P	Dirección	Dirección de memoria que apunta al principio de la próxima sentencia a ejecutar. Se inicializa con la base del segmento de código.
Base del stack	X	Dirección	Primer dirección utilizable del segmento de stack del hilo.
Cursor de Stack	S	Dirección	Dirección de memoria utilizada por el programa que apunta a una sección dentro del segmento de stack. Se inicializa con el valor de la base del segmento de stack.
Registros de programación	A,B,C,D,E	Numérico	Almacenamiento numérico para uso de la CPU

Nota 1: No será necesario almacenar los TCBs en la MSP.

Nota 2: Es posible agregar los campos que se consideren útiles y apropiados a este TCB, **previa validación** con el ayudante asignado.

Nota 3: Todas las direcciones de memoria son números enteros sin signo de 32 bits. Ver sección MSP -> Interfaz.

Nota 4: Todos los Registros de Programación son números enteros con signo de 32 bits.

Nota 5: Algunos campos no tienen ID porque, al no poder ser accedidos desde ESO, el ID queda a criterio de cada grupo.

Tras generar el TCB, el Loader moverá el Hilo creado del Proceso al final de la cola NEW, dejándolo disponible para que el subsistema Planificador pueda planificarlo.

Planificador

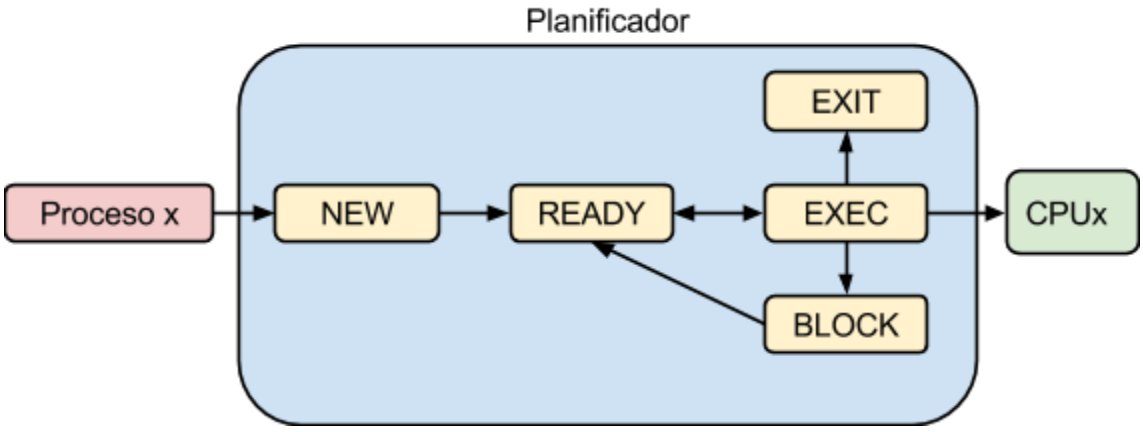
El subsistema Planificador es el encargado de distribuir los Hilos en los distintos CPUs.

Éste recibirá las conexiones de los distintos CPUs, y quedará a la espera de que se envíen a ejecutar nuevos Procesos para comenzar la planificación.

Ante la eventual desconexión de un CPU con un Proceso en ejecución, el Planificador deberá notificar a la Consola del Proceso dicha excepción y abortar la ejecución de ese Proceso. Se deberá considerar que el Proceso puede haber creado hilos; estos también tendrán que abortar.

Realizará las siguientes tareas:

- Recibirá los TCB de los Procesos y los encolará en la cola de READY, según el algoritmo de planificación de corto plazo Boolean Priority Round Robin (BPRR) por quantums.
- Moverá a la cola de READY todos los hilos que hayan terminado su Quantum de CPU.
- Recibirá de cada CPU la información actual de cada TCB que termine su ejecución, enviará a ejecutar los hilos que estén en READY a los CPUs disponibles, informando al CPU el quantum asignado. Moverá estos hilos a la cola EXEC.
- Moverá a la cola de EXIT todos los Hilos que:
    - Hayan concluido su ejecución
    - Deban abortar como resultado de una ejecución errónea (El proceso CPU debe informar esta situación al Kernel)
    - Deban abortar porque la CPU en que estaban ejecutando se desconectó del Kernel
    - Deban abortar porque la Consola asociada se desconectó del Kernel
- Moverá al estado BLOCK a todos los hilos que se bloqueen como resultado de la ejecución de alguna instrucción (El proceso CPU debe informar esta situación al Kernel).
- Creará nuevos Hilos cuando el proceso CPU lo indique y los moverá a la cola READY. Ver la sección de Servicios expuestos a la CPU.



Arranque (Boot)

Los Hilos de los Procesos pueden ejecutar llamadas que les permiten acceder a los recursos protegidos

del sistema (ver Llamadas al sistema (System Calls)). Para que esto sea posible, inmediatamente después de conectarse a la MSP, el Kernel deberá crear una estructura TCB especial con KM = 1 y encolarlo en el estado BLOCK hasta que la CPU notifique que el Proceso que está ejecutando solicita una llamada al sistema (a través de la ejecución de la instrucción de interrupción). Este será el denominado **Hilo Kernel**.

Una vez generado este TCB Kernel Mode (KM), el Kernel creará su segmento de código en la MSP y lo llenará con el contenido compilado del Archivo de System Calls provisto por la cátedra[4]. Este archivo contiene el código ESO que se ejecutará en la CPU tras una llamada al sistema. Después creará su segmento de stack y los registros de programación de la forma habitual. Notar que la estructura de este TCB es igual a los TCBs de los programas: la única diferencia es el valor del campo KM.

Por último, quedará a la escucha de nuevas conexiones de Consolas y CPUs para poder planificar y enviar a ejecutar.

Llamadas al sistema (System Calls)

Las llamadas al sistema deberán estar cargadas en memoria luego de bootear, por lo que el programa que invocó a alguna en particular puede interrumpirse y dar lugar al hilo Kernel, quien ejecutará esta rutina en la CPU.

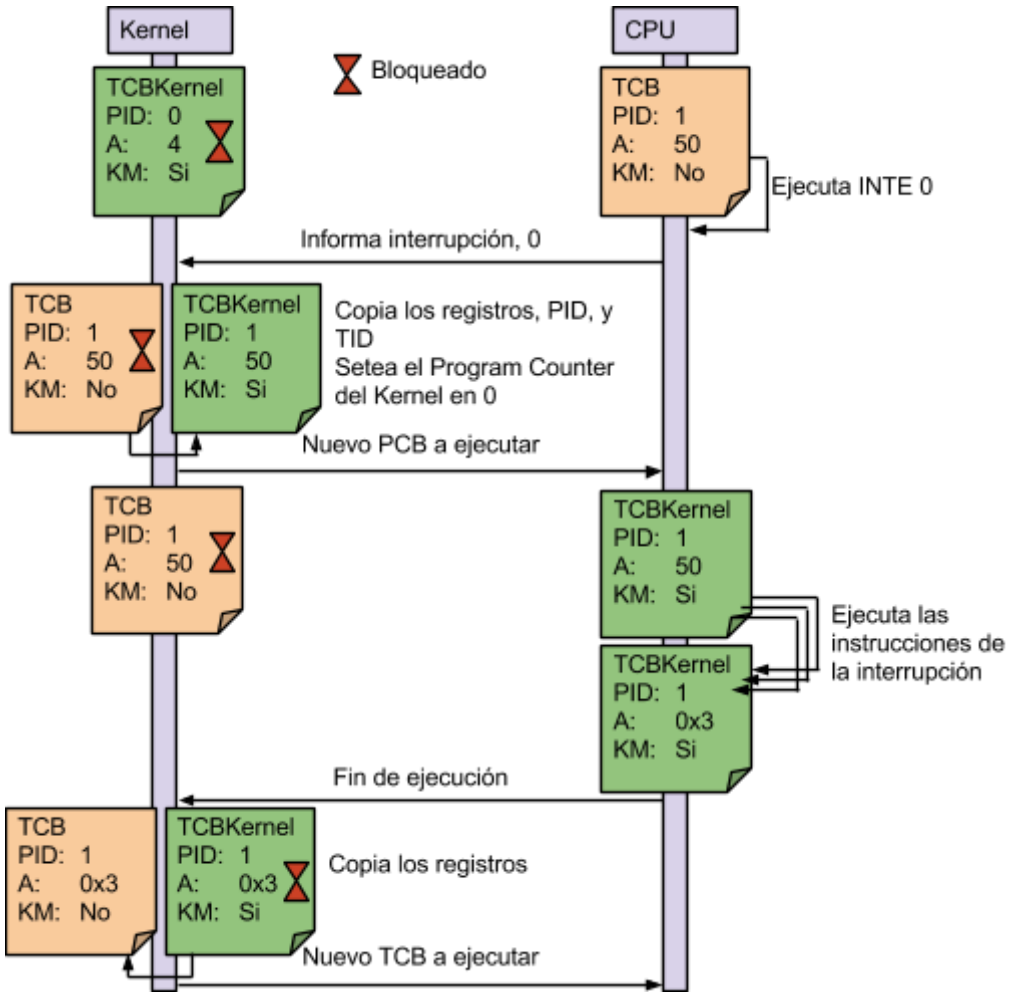
Para lograrlo, cuando el proceso CPU notifique al Kernel que un hilo desea ejecutar una llamada al sistema que requiere su atención (INTE), el Kernel recibirá el TCB de este hilo cargado y la dirección en memoria de la llamada a ejecutar y lo encolará en el estado BLOCK[5]. Luego agregará una entrada en la cola de llamadas al sistema con el TCB en cuestión.

Como el hilo Kernel tiene prioridad máxima en la planificación, siempre que haya una CPU disponible y necesidad de ejecutar una llamada al sistema:

1. Se tomará la primer entrada de la cola y se copiarán los registros de programación y el PID del TCB en el TCB KM.
2. Se cargará la dirección de la llamada al sistema en el Puntero de Instrucción del TCB KM.
3. Se enviará a ejecutar en la CPU el TCB KM antes mencionado.
4. Se ejecutará este TCB KM en algún CPU.
5. Una vez terminada la ejecución del TCB KM, el Kernel volverá a bloquear al TCB KM.
6. Se copiarán los registros del TCB KM nuevamente al TCB que pidió la llamada al sistema.
7. Se desbloqueará este TCB de usuario, para ser re-planificado.

Al bloquearse, los hilos pierden el Quantum que les restaba ejecutar. Una vez que se finaliza la llamada al sistema deben ser planificados al final de la cola READY y se les entrega una nuevo Quantum.

[4] Se encuentra en el repositorio: https://github.com/sisoputnfrba/so-ensamblador, archivo “systemCalls.txt”.
[5] En este trabajo práctico todas las llamadas al sistema son consideradas bloqueantes.



Notar que la forma de ejecución del hilo Kernel es idéntica a la de un programa normal, solo que éste puede ejecutar operaciones protegidas, y además, no puede interrumpirse por Quantum.

Dado que solo existe una única estructura TCB del hilo Kernel, es **imposible** que, en algún momento dado, se estén ejecutando dos llamadas al sistema en forma simultánea.

Liberación de Recursos

Además, el subsistema Planificador será el encargado de liberar los recursos de los hilos que lleguen a la cola EXIT. Deberá contemplar que la finalización de un hilo ESO puede no implicar la finalización del Proceso padre completo, ya que este puede estar compuesto por varios otros hilos.

Boolean Priority Round Robin (BPRR)

Algoritmo cíclico en donde cada hilo le corresponde una cantidad fija de operaciones, sin importar el tiempo que tarden en ejecutar. Esta cantidad (Quantum) se fija en el archivo de configuración. Adicionalmente se utilizan dos grados de prioridad; cada uno manejado como Round Robin, pero FIFO entre sí. Por lo que si existen hilos con prioridad 0 dentro de una cola con otros hilos de prioridad 1;

estos primeros (prioridad 0) deberán ser planificados en su totalidad, antes de empezar a planificar los de prioridad 1.

Se considera prioritario (prioridad 0) a todo TCB en modo Kernel. De lo contrario, se lo considera no-prioritario (prioridad 1)

Servicios expuestos a la CPU

El Kernel, como núcleo del sistema, expondrá a la CPU una serie de servicios centralizados que esta última no puede resolver por sí misma:

- Interrupción: Recibe una estructura TCB y una dirección de memoria. Ver sección Llamadas al Sistema más arriba.
- Entrada Estándar: Recibe un PID y un identificador de tipo. Pide a la consola del programa identificado por ese PID que se ingrese una cadena de texto/números (según el identificador de tipo). Devuelve a la CPU la cadena ingresada.
- Salida Estándar. Recibe un PID y una cadena de texto. Escribe en la consola del programa identificado por ese PID la cadena de texto recibida.
- Crear Hilo. Recibe una estructura TCB. Crea un nuevo hilo con el TCB recibido y lo envía a planificar normalmente. Se le debe crear un nuevo segmento de stack en la MSP pero no un nuevo segmento de código ya que este es igual para todos los hilos de un mismo programa.
- Join: Recibe un TID llamador y un TID a esperar. Envía al hilo identificado por el TID llamador recibido al estado BLOCK hasta que el otro hilo del mismo programa identificado por el TID a esperar finalice su ejecución.
- Bloquear: Recibe una estructura TCB e Identificador de Recurso. Envía al estado BLOCK al hilo identificado en el TCB recibido. Además, lo introduce al final de la cola de espera del recurso especificado en el Identificador de Recurso recibido.
- Despertar: Recibe un Identificador de Recurso. Remueve al primer hilo de la cola de espera del recurso especificado por Identificador de Recurso, además lo remueve del estado BLOCK y lo coloca al final de la cola READY para que vuelva a ser planificado.

Archivo de Configuración

Parámetro	Valor	Descripción
Puerto de Escucha PUERTO	[numérico]	Puerto TCP donde el Kernel escucha conexiones entrantes.
Dirección IP de la MSP IP_MSP	[texto]	Dirección IP donde se encuentra escuchando conexiones la MSP.
Puerto de la MSP PUERTO_MSP	[numérico]	Puerto TCP donde se encuentra escuchando conexiones la MSP.
Tamaño del Quantum QUANTUM	[numérico]	Cantidad de instrucciones que los hilos de usuario pueden ejecutar en la CPU antes de que se los interrumpa y se planifique otro hilo.
Ubicación Syscalls SYSCALLS	[texto]	Ruta completa hasta el archivo que contiene el código BESO de las llamadas al sistema.

Ejemplo:

```
PUERTO=112233
IP_MSP=192.168.1.104
PUERTO_MSP=54321
QUANTUM=4
SYSCALLS=/home/utnso/kernel/syscalls.bc
```

Proceso Consola

La Consola es una interfaz simple del Kernel que permite enviar a ejecutar un programa compilado al sistema y funcionar como consola del mismo para recibir los resultados de su ejecución o los mensajes que el Proceso necesite imprimir por pantalla. No es el encargado de compilar el código; esto es una etapa previa.

Arquitectura de la Consola

Recibirá por parámetro la ubicación del archivo a ejecutar, se conectará al proceso Kernel, y le enviará el contenido del archivo con el código BESO.

A partir de ese momento, el proceso quedará a la espera de mensajes correspondientes a las sentencias de entrada y salida con valores que deberá pedir o mostrar por consola, hasta que su ejecución haya concluido.

Podrán existir varias instancias de este proceso en el sistema ejecutando diferentes Procesos de manera independiente. Las Consolas y el Kernel deben poder ejecutar en máquinas diferentes.

Archivo de Configuración

Existirá un archivo de configuración con los parámetros necesarios para el funcionamiento del Proceso Programa, guardado en una ubicación en el disco que deberá ser referenciada por la variable de entorno `ESO_CONFIG`.

Parámetro	Valor	Descripción
IP Kernel `IP_KERNEL`	[texto]	Dirección IP donde se encuentra ejecutando el Kernel
Puerto Kernel `PUERTO_KERNEL`	[numérico]	Puerto TCP donde se encuentra ejecutando el Kernel

Ejemplo:

```
IP_KERNEL=192.168.0.20
PUERTO_KERNEL=5000
```

Proceso CPU

El proceso CPU es el encargado de interpretar y ejecutar las sentencias de ESO. Se encontrará en permanente contacto el Kernel (para tomar programas a ejecutar) y la MSP (para poder acceder al espacio de direcciones del hilo en ejecución).

Dado que el proceso Kernel planificará hilos de ESO, la CPU, a su vez, ejecutará esos hilos.

Al iniciar, intentará conectarse a estos dos procesos. En caso de no poder conectarse, abortará su ejecución informando por pantalla y log el motivo.

Una vez conectado, le solicitará al Kernel el TCB del próximo hilo a ejecutar y comenzará a ejecutar la operación de ESO especificada por el Puntero de Instrucción del mismo.

Además, junto con el TCB, también recibirá del Kernel el tamaño del Quantum que indicará la cantidad de instrucciones a ejecutar antes de solicitar el próximo TCB al Kernel.

Ciclo de ejecución

Cada vez que el CPU quiera ejecutar una instrucción, realizará los siguientes pasos:

1. Cargar los registros de la CPU con los datos del TCB a ejecutar.
2. Usando el registro Puntero de Instrucción, le solicitará a la MSP la próxima instrucción a ejecutar.
3. Interpretará la instrucción en BESO y realizará la operación que corresponda. Para conocer todas las instrucciones existentes y su propósito, ver el Anexo I: Especificación de ESO.
4. Actualizará los registros de propósito general del TCB correspondientes según la especificación de la instrucción.
5. Incrementa el Puntero de Instrucción.
6. En caso que sea el último ciclo de ejecución del Quantum, devolverá el TCB actualizado al proceso Kernel y esperará a recibir el TCB del próximo hilo a ejecutar. Si el TCB en cuestión tuviera el flag KM (Kernel Mode) activado, se debe ignorar el valor del Quantum.
7. Volver al punto 1).

Aclaraciones

- En cualquier caso, si cualquier llamada diera como resultado un error por **Violación de Segmento** o **Memoria Llena**, se cancelará la ejecución del programa, se le notificará al Kernel la situación, y se solicitará el TCB del próximo hilo a ejecutar.
- Solo cuando el TCB en ejecución tenga el flag KM esté activado se podrán ejecutar las instrucciones protegidas.
- En este modelo de llamadas al sistema, se logra ejecutarlas cargando valores en los registros, que servirán como parámetros de las funciones del kernel. Luego se ejecuta la instrucción de INTErrupción, con el número de rutina de Kernel, que casualmente es el desplazamiento dentro del archivo de system calls, cargado con anterioridad.

Archivo de Configuración

Parámetro	Valor	Descripción
Dirección IP del Kernel `IP_KERNEL`	[texto]	Dirección IP donde se encuentra escuchando conexiones el Kernel.
Puerto del Kernel `PUERTO_KERNEL`	[numérico]	Puerto TCP donde se encuentra escuchando conexiones el Kernel.
Dirección IP de la MSP `IP_MSP`	[texto]	Dirección IP donde se encuentra escuchando conexiones la MSP.
Puerto de la MSP `PUERTO_MSP`	[numérico]	Puerto TCP donde se encuentra escuchando conexiones la MSP.
Retardo Instrucción `RETARDO`	[numérico]	Tiempo en milisegundos que debe esperar la CPU antes de ejecutar cada operación ESO.

Ejemplo:

```
IP_KERNEL=192.168.1.103
PUERTO_KERNEL=12345
IP_MSP=192.168.1.104
PUERTO_MSP=54321
RETARDO=800
```

Proceso Memoria con Segmentación Paginada (MSP)

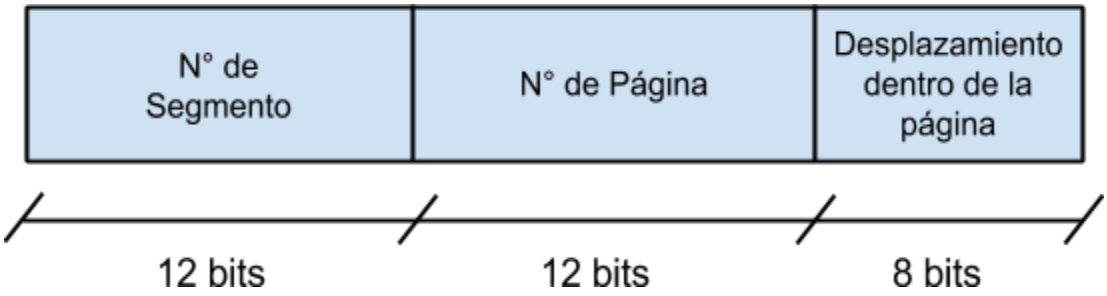
El sistema contará con un proceso encargado de proveer y administrar la memoria utilizada por los Procesos que se ejecutan en él. Permitirá a los mismos agrandar o achicar su espacio de direcciones creando y eliminando segmentos de manera de poder satisfacer sus cambiantes necesidades de memoria de una forma eficiente, evitando espacios inutilizados.

Para ello, este subsistema de gestión de memoria utilizará un esquema de segmentación paginada con memoria virtual[6], y permitirá que los procesos que la utilizan crean que disponen de la totalidad de la memoria del sistema de forma que puedan operar abstraídos de la misma. En pos de este objetivo, además, la MSP implementará un mecanismo de intercambio (swapping) de páginas bajo demanda que le facilitará ampliar el espacio de memoria disponible almacenando en disco aquellas páginas menos utilizadas.

Inicio

Una vez iniciada, la MSP reservará un único gran bloque de memoria que utilizará como memoria principal. El tamaño de este bloque estará definido por archivo de configuración.

Luego, se generarán todas las estructuras administrativas necesarias para administrar la memoria según el mecanismo de segmentación paginada[7]. La MSP soportará una cantidad máxima de segmentos por programa de 4096, tamaño de los mismos de hasta 4096 Bytes, páginas de tamaño fijo 256 Bytes. Esto da como resultado direcciones lógicas de tamaño 32 bits, según el siguiente esquema:



Una vez inicializado el subsistema, quedará a la escucha, de nuevas conexiones por parte del Kernel y las CPUs. Por cada conexión creará un hilo dedicado a atenderla, los cuales ejecutarán en forma independiente de los demás y responderán a los operaciones que la correspondiente CPU o Kernel soliciten. Además, quedará a la espera del ingreso de comandos a través de la consola (Ver sección Consola de la MSP más abajo).

[6] Se recomienda leer los capítulos del libro correspondientes a *Gestión de Memoria*, *Segmentación*, y *Paginación*.
[7] Estas estructuras administrativas no se guardarán en el gran bloque de memoria alocada al inicio (memoria principal), en él se guardará exclusivamente la información de los procesos ejecutando en el sistema, por ejemplo: código, stack, porciones de memoria dinámica, etc.

Interfaz

Las operaciones a las que podrá responder la MSP, y sus correspondientes parámetros, son:

- **Crear Segmento** (PID, Tamaño en Bytes): Crea un nuevo segmento para el programa PID del tamaño Tamaño en Bytes. Devuelve la dirección virtual base del segmento. Se debe verificar que la MSP tenga espacio disponible en memoria principal o en el espacio de swapping, en caso de no tener espacio deberá retornar Error de Memoria Llena.
- **Destruir Segmento** (PID, Base del Segmento): Destruye el segmento identificado por Base del Segmento del programa PID. Libera la memoria que ocupaba dicho segmento.
- **Solicitar Memoria** (PID, Dirección Lógica, Tamaño): Para el espacio de direcciones del proceso PID, devuelve la cantidad en bytes Tamaño comenzando desde Dirección Lógica. En caso que el pedido intente solicitar datos desde una posición de memoria inválida o que el mismo exceda los límites del segmento, retornará el correspondiente error de Violación de Segmento (Segmentation Fault).
- **Escribir Memoria** (PID, Dirección Lógica, Bytes a Escribir, Tamaño): Para el espacio de direcciones del proceso PID, escribe hasta Tamaño bytes del buffer Bytes a Escribir comenzando en la dirección Direccion Lógica. En caso que el pedido intente escribir datos en una posición de memoria inválida o que el mismo exceda los límites del segmento, retornará el correspondiente error de Violación de Segmento (Segmentation Fault).

Todas las direcciones de memoria retornadas por la MSP deben ser **números enteros sin signo de 32 bits**.

Notar que esta interfaz de operaciones no puede ser ampliada, y es la única forma que tienen los procesos CPU y Kernel de comunicarse con la MSP.

Intercambio a disco (Swapping)

Cuando la MSP requiera asignar un marco a un segmento determinado y no lograra encontrar uno disponible deberá intercambiar a disco alguno de los que estén en uso para poder liberarlo y luego asignarlo a dicho segmento. Esta técnica es conocida como Swapping y permite extender el espacio total de memoria disponible más allá de los límites de la memoria principal, abstrayendo a los programadores de aplicaciones del hecho de que la memoria es limitada.

Para ello, utilizará una serie de archivos de swapping. Es responsabilidad de la MSP utilizar sus estructuras administrativas para:

- ★ Marcar páginas pertenecientes a un segmento como “swappeadas”.
- ★ Proporcionar los mecanismos necesarios para que, en caso de que un programa intentara leer o escribir una página swappeada, se cargue dicha página en un marco de memoria principal antes de permitir el acceso.

La MSP creará un archivo de swapping por cada página que intercambie a disco. El nombre de cada

archivo estará compuesto por el PID del Proceso, el número de segmento y el número de página, de forma de poder identificar exactamente a que página de que Proceso corresponde.

Cuando la página es intercambiada de vuelta a memoria principal, el archivo debe borrarse, liberando el espacio en disco. Todos los archivos de swapping ocupan el mismo espacio en disco, que es el tamaño de la página.

El tamaño del espacio de intercambio será especificado por archivo de configuración.

Algoritmo de sustitución de páginas

A la hora de seleccionar un marco en uso como víctima para intercambiarlo a disco, el sistema deberá soportar al menos dos[8] de los siguientes algoritmos:

- FIFO
- Clock
- Clock Modificado o Algoritmo mejorado de segunda oportunidad.
- LRU

Consola de la MSP

Al iniciar, la MSP también deberá esperar la recepción de comandos por teclado que le permitirán al usuario ejecutar las operaciones que la misma soporta, además de solicitar información sobre el estado de la memoria.

Deberá soportar al menos los siguientes comandos:

- **Crear Segmento** [PID], [Tamaño]
  Crea un nuevo segmento en el espacio de direcciones del Proceso <**PID**> de tamaño <**Tamaño**>. Imprime la dirección base del segmento creado.
- **Destruir Segmento** [PID], [Dirección Base]
  Destruye el segmento identificado por <**Dirección Base**> en el espacio de direcciones del Proceso <**PID**>.
- **Escribir Memoria** [PID], [Dirección Virtual], [Tamaño], [Texto]
  Escribe hasta <**Tamaño**> bytes del <**Texto**> ingresado en la dirección <**Dirección Virtual**> del espacio de direcciones del Proceso <**PID**>. En caso de error por violación de segmento debe ser informado.
- **Leer Memoria** [PID], [Dirección Virtual], [Tamaño]
  Imprime hasta <**Tamaño**> del contenido de la memoria comenzando en <**Dirección Virtual**>, del espacio de direcciones del Proceso <**PID**>.

[8] La cátedra determinará en forma aleatoria cuales son los 2 algoritmos asignados a cada grupo.

- **Tabla de Segmentos**
  Imprime el contenido de la/las tablas de segmentos de todos los procesos ESO. Por cada segmento se debe imprimir al menos: PID, número de segmento, tamaño, dirección virtual base del segmento.
- **Tabla de Páginas** [PID]
  Imprime el contenido de la/las tablas de páginas del Proceso ESO <**PID**>. Por cada segmento se debe imprimir al menos: Número de segmento al que pertenece, si se encuentra en memoria principal o esta swappeada.
- **Listar Marcos**
  Imprime una lista de todos los marcos de memoria existentes en el sistema, indicando: Número de Marco, si está ocupado o no y por qué programa, y toda la información relacionada con los algoritmos de reemplazo de páginas implementados[9].

El resultado de todos los comandos especificados más arriba debe ser escrito tanto en el archivo de log de la MSP como en la pantalla.

Aclaraciones

Algunas aclaraciones sobre el funcionamiento de la MSP:

- Todos los Procesos tendrán como primer segmento al de número 0 (cero), por lo tanto la primer dirección lógica referenciable de memoria será la 0x00000000.
- Crear nuevos segmentos con sus nuevas páginas no implica asignarles marcos libres de memoria principal en forma inmediata. Los marcos se asignarán al segmento conforme el programa vaya utilizando (escribiendo o leyendo) las páginas referenciadas. Será responsabilidad de las estructuras de administración de la MSP detectar esta situación y conceder el marco correspondiente cuando llegue el momento[10]. Sin embargo, la MSP es responsable de detectar que tiene memoria suficiente para alojar el nuevo segmento (considerando la memoria principal + el espacio de intercambio) y responder en consecuencia.

[9] Esta información se utilizará para validar el correcto funcionamiento de los algoritmos de sustitución de páginas.
[10] Leer la sección del libro correspondiente a *Paginación bajo demanda*

Archivo de configuración

Parámetro	Valor	Descripción
Puerto de escucha de conexiones PUERTO	[numérico]	Puerto TCP utilizado para recibir las conexiones de los CPUs y el Kernel.
Tamaño de memoria principal CANTIDAD_MEMORIA	[numérico]	Tamaño en kilobytes de la memoria principal sin contar el espacio de intercambio (swapping).
Tamaño del archivo de swapping CANTIDAD_SWAP	[numérico]	Tamaño máximo en megabytes que pueden ocupar los archivos de swapping.
Algoritmo de Sustitución de Páginas SUST_PAGS	[texto]	Indica cual es el algoritmo de sustitución de páginas que utilizará la MSP.

Ejemplo:

```
PUERTO=12345
CANTIDAD_MEMORIA=100
CANTIDAD_SWAP=10
SUST_PAGS=CLOCK
```

Eventos de logueo obligatorio

El logueo de los eventos que ocurren en cada proceso desarrollado es muy importante para la verificación de su correcto funcionamiento. Es por eso que, más abajo, se definen una serie de eventos de logueo obligatorios.

Kernel, CPU

Para estos procesos se deberá utilizar la shared library ansisop-panel[11] y llamar a las funciones correspondientes (declaradas en panel.h, kernel.h, y cpu.h) cada vez que ocurran ciertos eventos. En principio, la biblioteca, al recibir los eventos, los guardará en un archivo de log.

Por ejemplo, cuando una CPU reciba un TCB para ejecutar con quantum de 3, se deberá invocar a:

```
ejecucion_hilo(tcb, 3);
```

MSP

Se deberán loguear los siguientes eventos:

- ❏ Inicio de la MSP, indicando: tamaño de la memoria principal, tamaño del archivo de paginación.
- ❏ Conexión de un CPU/Kernel.
- ❏ Recepción de una solicitud por parte del Kernel o alguna CPU, indicando cual fue, y que parámetros se recibieron.
- ❏ Creación/destrucción de segmentos indicando si se pudo satisfacer la operación o si hubo un error.
- ❏ Asignación/desasignación de marcos de memoria principal a un programa.
- ❏ Intercambio (swapping) de una pagina/paginas hacía/desde disco.
- ❏ Memoria principal llena/espacio de intercambio lleno.

En general, no se debería imprimir por pantalla ninguno de estos eventos, sino que para ello debe usarse el archivo de log que es persistente. En general, se reservará la impresión por pantalla para indicar terminaciones anormales, escribiendo los detalles en el archivo de log.

Para poder seguir las entradas que va logueando un programa, se recomienda usar el comando: "`tail -f <archivo de log>`", que va imprimiendo nuevas líneas conforme va creciendo dicho archivo. Por ejemplo:

```
> tail -f kernel.log
```

Cada vez que un proceso inicie su ejecución, deberá imprimir en el log una línea especial indicando que

[11] Link al repositorio: https://github.com/sisoputnfrba/ansisop-panel. La interfaz de la misma puede sufrir modificaciones. La implementación final de la misma será liberada próximamente.

comenzó la ejecución del mismo. Además, es posible ampliar la cantidad de eventos a loguear por encima de los mínimos especificados arriba, siempre que contribuya a entender el funcionamiento del programa y el mismo siga siendo legible y fácil de seguir. Se recomienda utilizar el TAD de Logging de la Commons Library (link) que facilita la generación y escritura de archivos de log.

Anexo I: Especificación de ESO

El lenguaje que deberán interpretar consta de un bytecode de 4 bytes seguido de 0, 1, 2 o 3 operadores de tipo registro (1 caracter, o sea 1 byte), número (1 entero, o sea 4 bytes) o direccion (1 entero, o sea 4 bytes). Los códigos de operación coinciden con su nombre en ASCII, por lo que el código para la instrucción “MOVR”, es 1297045074 en un número entero, que en hexadecimal es 0x4d4f5652, que en binario es: 01001101 (M) 01001111 (O)01010110 (V) 01010010 (R).

1. **LOAD** [*Registro*], [*Numero*]
   Carga en el registro, el número dado.
2. **GETM** [*Registro*], [*Registro*]
   Obtiene el valor de memoria apuntado por el segundo registro. El valor obtenido lo asigna en el primer registro.
3. **SETM** [*Numero*], [*Registro*], [*Registro]*
   Pone tantos bytes desde el segundo registro, hacia la memoria apuntada por el primer registro
4. **MOVR** [*Registro*], [*Registro*]
   Copia el valor del segundo registro hacia el primero
5. **ADDR** [*Registro*], [*Registro*]
   Suma el primer registro con el segundo registro. El resultado de la operación se almacena en el registro A.
6. **SUBR** [*Registro*], [*Registro*]
   Resta el primer registro con el segundo registro. El resultado de la operación se almacena en el registro A.
7. **MULR** [*Registro*], [*Registro*]
   Multiplica el primer registro con el segundo registro. El resultado de la operación se almacena en el registro A.
8. **MODR** [*Registro*], [*Registro*]
   Obtiene el resto de la división del primer registro con el segundo registro. El resultado de la operación se almacena en el registro A.
9. **DIVR** [*Registro*], [*Registro*]
   Divide el primer registro con el segundo registro. El resultado de la operación se almacena en el registro A; a menos que el segundo operando sea 0, en cuyo caso se asigna el flag de *ZERO_DIV* y no se hace la operación.
10. **INCR** [*Registro*]
    Incrementar una unidad al registro.
11. **DECR** [*Registro*]
    Decrementa una unidad al registro.
12. **COMP** [*Registro*], [*Registro*]
    Compara si el primer registro es igual al segundo. De ser verdadero, se almacena el valor 1. De lo contrario el valor 0. El resultado de la operación se almacena en el registro A.

13. **CGEQ** [*Registro*], [*Registro*]
    Compara si el primer registro es mayor o igual al segundo. De ser verdadero, se almacena el valor 1. De lo contrario el valor 0. El resultado de la operación se almacena en el registro A.
14. **CLEQ** [*Registro*], [*Registro*]
    Compara si el primer registro es menor o igual al segundo. De ser verdadero, se almacena el valor 1. De lo contrario el valor 0. El resultado de la operación se almacena en el registro A.
15. **GOTO** [*Registro*]
    Altera el flujo de ejecución para ejecutar la instrucción apuntada por el registro. El valor es el desplazamiento desde el inicio del programa.
16. **JMPZ** [*Numero*]
    Altera el flujo de ejecución, solo si el valor del registro A es cero, para ejecutar la instrucción apuntada por el registro. El valor es el desplazamiento desde el inicio del programa.
17. **JPNZ** [*Numero*]
    Altera el flujo de ejecución, solo si el valor del registro A es cero, para ejecutar la instrucción apuntada por el registro. El valor es el desplazamiento desde el inicio del programa.
18. **INTE** [*Direccion*]
    Interrumpe la ejecución del programa para ejecutar la rutina del kernel que se encuentra en la posición apuntada por la direccion. El ensamblador admite ingresar una cadena indicando el nombre, que luego transformará en el número correspondiente. Los posibles valores son “MALC”, “FREE”, “INNN”, “INNC”, “OUTN”, “OUTC”, “BLOK”, “WAKE”, “CREA” y “JOIN”. Invoca al servicio correspondiente en el proceso Kernel. Notar que el hilo en cuestión debe bloquearse tras una interrupción.
19. **FLCL**
    Limpia el registro de flags.
20. **SHIF** [*Número*], [*Registro*]
    Desplaza[12] los bits del registro, tantas veces como se indique en el Número. De ser desplazamiento positivo, se considera hacia la derecha. De lo contrario hacia la izquierda.
21. **NOPP**
    Consume un ciclo del CPU sin hacer nada
22. **PUSH** [*Número*], [*Registro*]
    Apila los primeros bytes, indicado por el número, del registro hacia el stack. Modifica el valor del registro cursor de stack de forma acorde.
23. **TAKE** [*Número*], [*Registro*]
    Desapila los primeros bytes, indicado por el número, del stack hacia el registro. Modifica el valor del registro de stack de forma acorde.
24. **XXXX**
    Finaliza la ejecución.

Instrucciones Protegidas

Además el CPU puede ejecutar otro conjunto de instrucciones solo si el valor de KM es 1. Ninguna de

[12] http://es.wikipedia.org/wiki/Operador_a_nivel_de_bits#Desplazamiento_l.C3.B3gico

estas operaciones tiene operadores:

25. **MALC**
    Reserva una cantidad de memoria especificada por el registro A. La direccion de esta se almacena en el registro A. Crea en la MSP un nuevo segmento del tamaño especificado asociado al programa en ejecución.
26. **FREE**
    Libera la memoria apuntada por el registro A. Solo se podrá liberar memoria alocada por la instrucción de MALC. Destruye en la MSP el segmento indicado en el registro A.
27. **INNN**
    Pide por consola del programa que se ingrese un número, con signo entre –2.147.483.648 y 2.147.483.647. El mismo será almacenado en el registro A. Invoca al servicio correspondiente en el proceso Kernel.
28. **INNC**
    Pide por consola del programa que se ingrese una cadena no más larga de lo indicado por el registro B. La misma será almacenada en la posición de memoria apuntada por el registro A. Invoca al servicio correspondiente en el proceso Kernel.
29. **OUTN**
    Imprime por consola del programa el número, con signo almacenado en el registro A. Invoca al servicio correspondiente en el proceso Kernel.
30. **OUTC**
    Imprime por consola del programa una cadena de tamaño indicado por el registro B que se encuentra en la direccion apuntada por el registro A. Invoca al servicio correspondiente en el proceso Kernel.
31. **CREA**
    Crea un hilo, hijo del TCB que ejecutó la llamada al sistema correspondiente. El nuevo hilo tendrá su Program Counter apuntado al número almacenado en el registro B. El identificador del nuevo hilo se almacena en el registro A.
    Para lograrlo debe generar un nuevo TCB como copia del TCB actual, asignarle un nuevo TID correlativo al actual, cargar en el Puntero de Instrucción la rutina donde comenzará a ejecutar el nuevo hilo (registro B), pasarlo de modo Kernel a modo Usuario, duplicar el segmento de stack desde la base del stack, hasta el cursor del stack. Asignar la base y cursor de forma acorde (tal que la diferencia entre cursor y base se mantenga igual)[13] y luego invocar al servicio correspondiente en el proceso Kernel con el TCB recién generado.
32. **JOIN**
    Bloquea el programa que ejecutó la llamada al sistema hasta que el hilo con el identificador almacenado en el registro A haya finalizado. Invoca al servicio correspondiente en el proceso Kernel.
33. **BLOK**

[13] De tener una base de stack en 100, y un cursor en 130 *(S-X=30)*. Al crear un nuevo stack, la dirección de este podría ser 500, por lo que el cursor tendrá que ser 530 *(S-X=30)*.

Bloquea el programa que ejecutó la llamada al sistema hasta que el recurso apuntado por B se libere.
La evaluación y decisión de si el recurso está libre o no es hecha por la llamada al sistema WAIT pre-compilada.

34. **WAKE**
    Desbloquea al primer programa bloqueado por el recurso apuntado por B.
    La evaluación y decisión de si el recurso está libre o no es hecha por la llamada al sistema SIGNAL pre-compilada.

Notar que las instrucciones son de tamaño variable tanto por la cantidad de parámetros que reciben como por el tamaño de cada uno de los mismos. Sin embargo, interpretando los primeros 4 bytes de cada una es posible conocer de qué instrucción se trata, y, por lo tanto, cual es el tamaño de la misma.

Anexo II: Especificación de las llamadas al sistema

Como se explicó antes, el sistema cuenta con algunas llamadas al sistema (system calls) que son ejecutadas luego de pedir por ellas con la operación de INTErrupción. Muchas de estas son simplemente una operación codificada en ESO junto con algunas operaciones para consumir algunos ciclos.

1. **MALLOC**
   Ejecuta la operación de **MALC**.
2. **FREE**
   Ejecuta la operación de **FREE**.
3. **INN**
   Ejecuta la operación de **INNN**.
4. **INC**
   Ejecuta la operación de **INNC**.
5. **OUTN**
   Ejecuta la operación de **OUTN**.
6. **OUTC**
   Ejecuta la operación de **OUTC**.
7. **CREATE_THREAD**
   Ejecuta la operación de **CREA**.
8. **JOIN_THREAD**
   Ejecuta la operación de **JOIN**.
9. **WAIT**
   Toma el valor apuntado por el registro B, resta una unidad, guarda en la memoria correspondiente el nuevo valor y evalúa si este contador es negativo o cero. De serlo, invoca a la operación de **BLOK**; de lo contrario no. Altera el registro A por uso interno.
10. **SIGNAL**
    Toma el valor apuntado por el registro B, suma una unidad, guarda en la memoria correspondiente el nuevo valor y evalúa si este contador es negativo o cero. De serlo, invoca a la operación de **WAKE**; de lo contrario no. Altera el registro A por uso interno.
11. **SETSEM**
    Fija el valor del recurso apuntado por el registro B al valor del registro D. Altera el registro A por uso interno.

Anexo III: Uso del ensamblador

El ensamblador es un programa que brinda la cátedra que permite compilar el código fuente ESO en código ensamblado BESO listo para ejecutar, entendible por la CPU. El código del ejecutable se encuentra en el repositorio: https://github.com/sisoputnfrba/so-ensamblador que podrá sufrir modificaciones a lo largo del cuatrimestre, en cuyo caso se avisará oportunamente a través del foro de Novedades del Campus Virtual.

Opciones

- **v**: Verboso, imprime en la consola el resultado.
- **o**: Output, indica el archivo a donde guardar el resultado. De evitarlo y no ser verboso, se guarda en "*out*".
- **d**: Decompile, intenta des-compilar el código de máquina.
- **c**: Compile, modo por defecto, compila.

El ensamblador soporta etiquetas, que traducirá a números al ensamblar, por lo que es válido el código de la izquierda, que se transformara en el de la derecha

LOAD A,#Salto GOTO A LOAD B,5 :Salto LOAD C,9 XXXX	LOAD A,23 GOTO A LOAD B,5 LOAD C,9 XXXX

También soporta comentarios, tanto en una línea vacía, como al final de una línea, por lo que efectúa esta transformación:

LOAD A,7^ Cargar 7 en A ^Cargar 90 en B LOAD B,90 MULR A,B^ 7 * 90	LOAD A,7 LOAD B,90 MULR A,B

Dado que al transformar a código de máquina, el archivo es ilegible, el ensamblador también nos brinda la posibilidad de des-ensamblar, pero los comentarios y las etiquetas se habrán perdido.

Descripción de las entregas

Para permitir una mejor distribución de las tareas y orientar al alumno en el proceso de desarrollo de su trabajo, se definieron una serie de puntos de control y fechas que el alumno podrá utilizar para comparar su grado de avance respecto del esperado.

Checkpoint 1

Fecha: 13 de Septiembre

Objetivos:

- ★ Familiarizarse con Linux y su consola, el entorno de desarrollo y el repositorio
- ★ Aplicar las Commons Libraries, principalmente las funciones para listas, archivos de conf y logs
- ★ Implementar en la CPU el interpretador de instrucciones y modelar algunas de las funcionalidades
- ★ Modelar las estructuras necesarias para gestionar las páginas y segmentos de la MSP
- ★ Desarrollar un modelo de consola para la MSP
- ★ Familiarizarse con el modelo de Segmentación Paginada con Paginación bajo demanda

Lectura recomendada:

http://faq.utn.so/arrancar
Beej Guide to Network Programming - link
Linux POSIX Threads - link
SisopUTNFRBA Commons Libraries - link
Capítulos de Gestión de memoria de los libros de la materia

Checkpoint 2

Fecha: 27 de Septiembre

Objetivos:

- ★ Construir dos programas que permitan recibir múltiples conexiones por sockets y gestionarlas utilizando por un lado un multiplexor de entrada/salida (`select`, `epoll`, etc) y por el otro delegar su atención a hilos independientes.
- ★ Realizar un cliente simple que permita enviar mensajes estructurados a los servidores
- ★ Utilizar semáforos para sincronizar hilos que acceden a una lista compartida
- ★ Diseñar el diagrama de estados de un TCB en el sistema
- ★ Diseñar el impacto de las operaciones sobre las estructuras del Proceso
- ★ Implementar la interfaz de mensajes de la MSP y su correspondiente validación

Lectura recomendada:

Sistemas Operativos, Silberschatz, Galvin - Capítulo 4: Hilos
Sistemas Operativos, Silberschatz, Galvin - Capítulo 5: Planificación del Procesador

Checkpoint 3

Fecha: 18 de Octubre

Objetivos:

- ★ Implementar operaciones de usuario de ESO en el CPU en las estructuras de la MSP
- ★ Implementar el cambio de contexto con el TCB del Kernel
- ★ Modelar los distintos estados de los TCBs en el Kernel
- ★ Modelar el sistema de paginación y segmentación de la MSP

Checkpoint 4 - Presencial

Fecha: 1 de Noviembre

Objetivos:

- ★ Implementar las llamadas al sistema
- ★ Desarrollar la planificación completa de un Proceso en el sistema
- ★ Implementar los algoritmos de selección determinados en la MSP
- ★ Implementar interfaz completa de la MSP y junto con el mecanismo de segmentación paginada

Checkpoint 5

Fecha: 20 de Noviembre

Objetivos:

- ★ Implementar swapping y los algoritmos de reemplazo de páginas
- ★ Validar los requisitos funcionales del trabajo práctico
- ★ Realizar pruebas de stress en el sistema
- ★ Programar los Makefiles correspondientes y hacer pruebas de funcionamiento en el laboratorio en la VM server

Normas del Trabajo Práctico

El trabajo práctico de este cuatrimestre se rige por las Normas del Trabajo Práctico (NTP), detalladas en el siguiente link: http://faq.utn.so/ntp