

Sistemas Operativos

# Construcción del núcleo de un sistema operativo

---

Fermín Gómez

Miguel Di Luca

Pedro Vedoya

Maite Herrán



## Índice

### **1      Introducción**

### **2      Implementación**

#### **2.1    Procesos**

2.1.1    Cambios de contexto

2.1.2    Instanciación de un proceso

2.1.3    Scheduler

#### **2.2    Manejo de memoria**

#### **2.3    IPC**

2.3.1    Mensajes

2.3.2    Mutexes

#### **2.4    Aplicaciones de User Space**



## 1. Introducción

El presente trabajo práctico busca la construcción del núcleo de un Sistema Operativo implementando mecanismos de IPC, Memory Management y scheduling. El kernel fue creado usando como base el utilizado en la materia Arquitectura de Computadoras en el primer cuatrimestre del 2018.

## 2. Implementación

### 2.1 Procesos

En el presente sistema operativo, un proceso es un programa con un solo hilo de ejecución. Cada proceso tiene asociado:

- un número identificador (pid) que no puede ser el mismo para dos procesos diferentes.
- un puntero al stack propio del proceso (el stack es un bloque de memoria reservado únicamente para el proceso en cuestión).
- un estado que puede ser listo (ready), corriendo (running), esperando (waiting) o muerto (dead).
- una lista “heap”. Cuando un proceso quiere reservar un pedazo de memoria utilizando la función malloc, el alocador de memoria se la dará y se guardará su dirección inicial en la lista “heap”. Dado el caso en el que el proceso termine sin haber liberado a todos los pedazos de memoria que había pedido alocar previamente, el sistema podrá recorrer la lista y liberarlos.

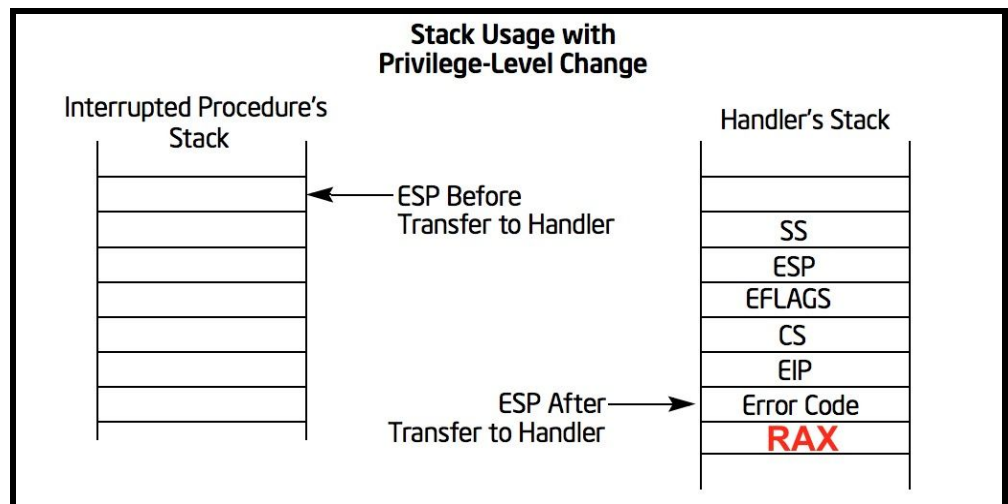
### 2.1.1 Cambios de contexto

Este sistema operativo ofrece multitarea apropiativa (*"preemptive multitasking"*), por lo que provee la posibilidad de ejecutar varios procesos al mismo tiempo. Esto lo hace mediante la rutina de atención a la interrupción del timer tick en la que se realiza un cambio de contexto (se para la ejecución de un proceso para dar paso a la ejecución de otro) por cada interrupción.

En la rutina de atención del timer tick, se guarda el contexto del proceso que se va a desalojar en su propio stack. Luego al scheduler (el programa del sistema que decide a qué proceso asignar el próximo intervalo de ejecución) se le pasa el puntero del stack del proceso a desalojar y el scheduler devuelve el puntero al stack del próximo proceso a ejecutarse. El scheduler se encargará de guardar el puntero del stack del proceso a desalojar en la estructura del propio proceso, mientras que en el context switch se cargará en el registro RSP el puntero recibido por el scheduler. Se desapilarán los registros y se restaurará el contexto del próximo proceso para que este pueda ser ejecutado. Cuando la interrupción finaliza, el RIP apuntará a la próxima instrucción a ejecutarse del próximo proceso.

### 2.1.2 Instanciación de un proceso


Al instanciar un proceso, se debe de crear una estructura para poder guardar los atributos previamente mencionados. En particular, se debe de reservar espacio para el stack del proceso y se le debe de pushear el instruction pointer, selectores de segmento y flags tal como lo hace el procesador cuando entra en una interrupción:



Luego se pushean valores que serán los que tomarán los registros cuando corra el proceso. Como el proceso todavía no corrió al menos una vez, el valor de estos registros puede ser cualquiera.

### 2.1.3 Scheduler

Cada vez que suena el timer tick, se llama a una función llamada dispatcher que lo que hace es correr al scheduler para que este seleccione al próximo proceso a ejecutarse y que así el dispatcher pueda




devolver a la rutina de atención del timer tick el puntero del stack del próximo proceso a ejecutarse, pudiéndose realizar el context switch.

El algoritmo utilizado para seleccionar al próximo proceso es el llamado Round Robin, un algoritmo sencillo y equitativo de reparto de la CPU entre procesos para evitar la monopolización del uso de esta. Gracias al timer tick que genera periódicamente interrupciones, se garantiza que nuestro sistema operativo (particularmente la rutina de servicio de interrupción del timer tick) tome el mando de la CPU periódicamente. El quantum de un proceso equivale, en este caso, a 1. Es decir, cada vez que suena el timer tick se hará un cambio de proceso. Cabe destacar que si el proceso se bloquea o muere, no se le otorgará la cpu.

## **2.2 Manejo de memoria**

El buddy allocator funciona de tal forma que va dividiendo reiteradamente bloques de memoria a la mitad para crear dos buddies más pequeños hasta que se obtenga un bloque del tamaño deseado. Cuando alocamos, verificamos si tenemos algún bloque libre del tamaño deseado. Si no, dividimos un bloque más grande tantas veces como sea necesario para obtener un bloque del tamaño adecuado. Entonces, si queremos 32 K, dividimos el bloque de 128 K en 64 K y luego dividimos uno en 32 K. Como se puede ver, este método de división significa que los tamaños de bloque siempre serán potencias de dos. Si se intenta asignar algo más pequeño, por ejemplo 13 K, la asignación se redondeará a la potencia más cercana de dos (16 K) y luego se le asignará un bloque de 16 K.

Se puede ver que hay una gran cantidad de fragmentación ocurriendo. Es importante notar que este tipo de fragmentación se llama fragmentación interna, ya que se desperdicia memoria dentro de un bloque pero no desperdicia espacio entre los bloques. Cada vez que se libera un bloque, verificamos si el buddy también es libre. De ser así, se fusionan los dos buddies en el bloque único que alguna vez fueron antes de ser separados. Esta acción se realiza recursivamente hasta que el bloque no tenga un buddy libre. La gran ventaja de este mecanismo es que se evita la fragmentación externa.



En este mecanismo de manejo de memoria, se utilizó un array con listas en donde cada lista contiene los bloques de un nivel (cada nivel tiene bloques de memoria del mismo tamaño). Hay 22 niveles en donde el nivel más grande tiene 512MB y el más chico tiene 128 bytes.

## **2.3 IPC**

### **2.3.1 Mensajes**

Implementamos un sistema bastante similar a como funcionan los unnamed pipes, que funciona como un buffer circular al que pueden acceder los dos procesos en las puntas de pipe (tanto para leer como para escribir). Para poder acceder a este espacio de memoria, se utiliza un sistema de 3 mutex, que le brindan al proceso que quiere acceder al buffer derecho de uso, escritura y lectura. Luego, se fija si el buffer tiene contenido, si quiere leer y está vacío o si quiere escribir y esta lleno el proceso correspondiente a la acción se bloquea hasta que se le vuelva a activar.

### **2.3.2 Mutexes**

Para los mutexes, utilizamos una estructura que guarda el valor del semáforo, el nombre, quien lo posee y una lista de procesos en espera para acceder al semáforo. En el kernel, todos los mutexes creados se guardan en una queue, donde se utiliza un comparados por nombre para acceder a ellos (no puede haber dos mutexes con el mismo nombre). Para poder acceder a el lock, se utiliza la función `acquire`, que primero intenta conseguir el lock, para lo que se utiliza una función en assembler que accede al registro `rdi`, donde se guarda la variable del semáforo. Se guarda 1 en el registro `eax`, y se intercambia por el valor de `rdi`, si este era 0 (desocupado), `eax` devuelve 0 y el mutex se toma, caso contrario, devuelve 1 y entra a un `if` que guarda en la cola de espera el proceso, para que al liberarse el mutex se llama dicho proceso. Al desbloquearse, se utiliza una metodología similar, solo que se le pone 0 a `eax`, y que antes de terminar la función `release` la lista de espera, si hay algún proceso en ella le avisa que es su turno despertándolo.

## 2.4 Aplicaciones de User Space

- **PS:** Para imprimir la información de los procesos disponibles, invocar este comando crea una syscall que imprime en pantalla el pid, el estado, el nombre y la memoria en uso. Hace esto usando un loop que imprime el proceso que está corriendo, y luego evalúa todos los que están listos para correr, accediendo e imprimiendo su pcb.
- **Sushi(ProdCons):** El programa sushi es un ejemplo de cómo implementamos el problema del productor-consumidor. El proceso comienza simulando un “SushiMan” (productor) y un “monstruo” (consumidor), ambos funcionando como procesos independientes, en el cual el sushiman es encargado de crear una cantidad aleatoria de “sushis”, que pone en un buffer con una capacidad máxima, y tiene una capacidad limitada, mientras que el monstruo es encargado de consumir una cantidad aleatoria de sushis del buffer. Estos procesos se activan usando el exec, por lo tanto se alternan en producir y consumir del buffer, que se va actualizando en pantalla. El usuario puede aumentar la cantidad de productores o consumidores (lo que cambiaría la dinámica de oferta y demanda del sushi) mediante el uso de teclas, o salir cuando se terminen de ejecutar todos los procesos de sushimans o monstruos.
- **Memory:** muestra los distintos bloques que hay armados en memoria, mediante una syscall, encargada de imprimir en la pantalla el bloque, donde comienza y dónde termina, iterando por la lista de los bloques creados.
- **Messages:** Para los pipes, se utilizó una función que crea un pipe, y luego pasa a leer lo que escribe, y repetir el procedimiento, todo desde distintos procesos..
- **Malloc:** este programa tiene como objetivo mostrar la alocaión de memoria en un proceso paso a paso.

## 3.0 Instrucciones de Compilación y Ejecución





Desde el directorio del TP, en BASH:

**\$>make clean**

**\$>docker run -v \${PWD}:/root -ti agodio/itba-so:1.0**

**\$>make all (en el directorio del TP)**

**\$>./run.sh**


## **4.0 Limitaciones**

El programa encuentra limitaciones a la hora de intentar imprimir muchas cosas en masa simultáneo, dado que el scheduler trabaja más rápido de lo que se puede imprimir una línea (ya que al hacer una nueva línea se copia y pega la pantalla una línea más arriba), por lo tanto al hacer un loop que imprima muchas cosas en líneas distintas puede haber problemas de impresión.

## **5.0 Problemas encontrados durante el desarrollo**

Uno de nuestros primeros problemas fue encontrar la manera más óptima para hacer funcionar el scheduler, y específicamente como hace cuando se termina de ejecutar un proceso, ya que al final de todo nos generaba problemas cuando se terminaba de ejecutar el último proceso.

Para el scheduler, decidimos utilizar un algoritmo round robin que itera sobre una lista de procesos disponibles, y tiene una segunda lista de procesos bloqueados, lo cual nos pareció la manera más eficiente de poder manejar los procesos. Para solucionar cómo se trabaja el código de un proceso, se utiliza un puntero en la estructura que hace referencia a este código que van a correr, y le seteamos el instruction pointer a otra función armada por nosotros, que va a ser ejecutada por todos los procesos cuando empiezan a correr. El trabajo de ésta es ejecutar el código almacenado en la estructura, y luego hacer un endProcess una vez que se termina de ejecutar, para hacer un halt que deja al scheduler esperando al siguiente proceso a ejecutarse cuando se corra el timer tick. Esto permite



también poder hacer un return al final de cada código y no un endProcess, ya que eso no es práctica común y no quedaba muy bien.

finalmente, también hubo problemas sobre cómo organizar las syscalls, ya que en el trabajo que habíamos exportado de Arquitectura de Computadoras la manera que habíamos implementado (un switch dependiendo de los parámetros), no era muy ordenada. Decidimos cambiar esta implementación a una donde se guarda en un array las distintas syscalls que se acceden mediante parámetro, y se ejecutan directamente utilizando los parámetros que se le pasaron al syscall handler, lo cual lo hace mas rapido y entendible.