

Trabajo Práctico N°2

Construcción del Núcleo de un Sistema Operativo y estructuras de administración de recursos.

Alumnos:

De Simone, Franco
Dizenhaus, Manuel
Cornídez, Milagros

61100
61101
61432

1 Introducción

Para este trabajo práctico, se nos requirió retomar el trabajo práctico realizado en la materia "Arquitectura de Computadoras", y reinventarlo con implementación de procesos, semáforos, pipes, dos administradores de memoria por el lado del backend, y varias funciones para el front end (entre ellas el reconocido problema de los filósofos). Siguiendo el consejo de la cátedra, se procedió en el orden indicado en la consigna. El orden de la explicación del trabajo se corresponde con el de la consigna.

2 Backend

2.1 System Calls

Para esta porción del trabajo, no hubo que realizar mayores modificaciones. Si bien la estructura del trabajo de arquitectura de computadoras no fue suficiente para soportar la cantidad de funciones que necesitábamos invocar, la modificación fue hacer un *"if-else"* para mantenerla estructura de lo realizado previamente, y a su vez colocar un switch que nos permitiera invocar a las funciones nuevas a declarar. Creímos que era relevante mantener la estructura previa del trabajo, mas allá que no era relevante al trabajo en cuestión, dado que era el punto de partida de lo que íbamos a realizar en este caso.

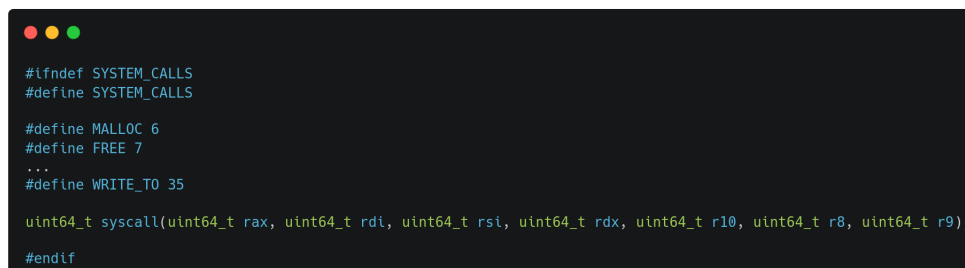
A screenshot of a code editor showing a C function named sysCallDispatcher. The function takes six uint64_t arguments (rdi, rsi, rdx, r10, r8, r9). It first checks if rdi is less than 6. If so, it looks up a syscall in a table, checks if it's non-zero, and then calls it with rsi, rdx, and r10. If rdi is 6 or greater, it enters an else block with a switch statement on rdi. A comment below the switch says '<Todas nuestras syscalls implementadas>'.

```
uint64_t sysCallDispatcher (uint64_t rdi, uint64_t rsi, uint64_t rdx, uint64_t r10, uint64_t r8, uint64_t r9){
    if(rdi < 6){
        PSysCall sysCall = sysCalls[rdi];
        if (sysCall != 0)
            return sysCall(rsi, rdx, r10);
        return 0;
    }
    else{
        switch(rdi){
            <Todas nuestras syscalls implementadas>
        }
    }
}
```

Imagen 1: Extracto del código del archivo *sysCallDispatcher.c*

En cuanto a la implementación de nuevas system calls, hubo que replantear la manera de realizar una desde el frontend, dado el volumen de posibilidades (considerando que pasamos de contar con 6 que manejaban nuestras funcionalidades básicas, a 36 posibles interrupciones.)

Se pasó de tener una función individual para cada system call (como era el caso en el trabajo base), a parametrizar todo bajo una única función llamada *syscall*, lógicamente.

A screenshot of a code editor showing the header file sysCalls.h. It includes conditional compilation directives for SYSTEM_CALLS and MALLOC. It defines several macros for system calls: MALLOC (6), FREE (7), and WRITE_TO (35). It also shows the prototype for the syscall function, which takes rax, rdi, rsi, rdx, r10, r8, and r9 as arguments.

```
#ifndef SYSTEM_CALLS
#define SYSTEM_CALLS

#define MALLOC 6
#define FREE 7
...
#define WRITE_TO 35

uint64_t syscall(uint64_t rax, uint64_t rdi, uint64_t rsi, uint64_t rdx, uint64_t r10, uint64_t r8, uint64_t r9);

#endif
```

Imagen 2: Extracto de código del archivo *sysCalls.h*, ubicado dentro de la carpeta *Userland/SampleCodeModule/include*

2.2 Administradores de memoria

En lo que concierne a los administradores de memoria, se requirió la implementación de dos variantes para administrar la misma. Por un lado, el administrador *buddy*, cuya lógica está basada en la división recursiva de la totalidad de la memoria en potencias de 2, dejando de esta manera todas los bloques en tamaños que sean potencias del mismo. Para ello, utilizamos un arbol como estructura de datos para administrar los nodos. Se puede observar una estructura básica del funcionamiento del administrador de memoria en la imagen [n].

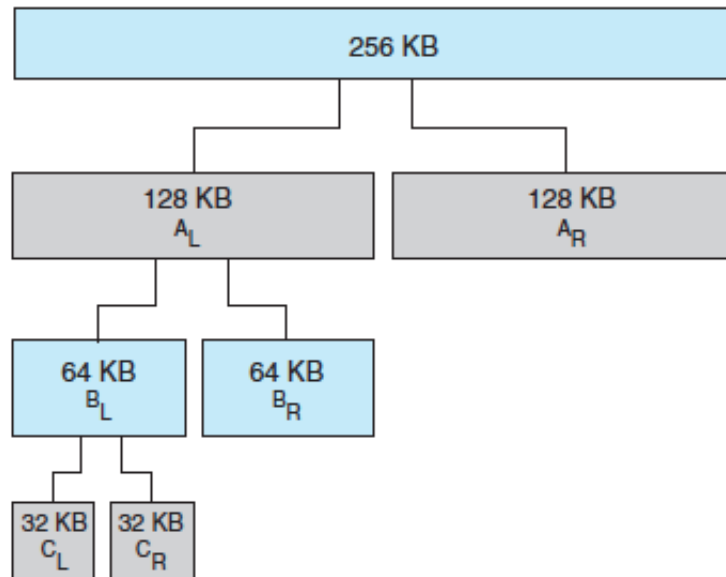


Imagen 3: Representación gráfica del funcionamiento de un administrador *buddy*¹

La implementación siguió la lógica clásica de asignación y borrado de nodos de un árbol. Se creó un *struct* nodo, con la información del nodo como punteros a sus dos hijos. Logicamente, el tamaño de un nodo hijo es la mitad del nodo padre.

¹Fuente original de imagen: <https://www.chegg.com/homework-help/questions-and-answers/1-024-kb-segment-memory-allocated-using-buddy-system-using-figure-926-attached-picture-gui-q11679881>Link

Por otro lado, se requirió la implementación de otro administrador de memoria a elección. Dada la practicidad del manejo del sistema por nodos, y luego de una profunda investigación sobre diversas implementaciones de manejo de memoria, decidimos estructurar nuestro segundo administrador de la misma manera que el buddy, con la salvedad que no se trata de un árbol si no de una lista encadenada de nodos. La idea es comenzar con un gran bloque, e ir partiéndolo a medida que se demanda memoria. Cuando se aloca, se busca el "hueco" dentro de la memoria disponible y se dispone de un nodo para esto. Para liberar, es un análisis similar. Se busca el nodo pedido con su dirección, y se une a derecha o izquierda, dejando un nodo "libre" de mayor tamaño.

En cuanto a la posibilidad de selección entre memory managers, lo que utilizamos para que el usuario pueda decidir si compilar con el administrador buddy o no, colocamos un parametro a la hora de compilación: si el usuario quiere compilar con buddy, simplemente debe colocar

```
make buddy
```

a la hora de compilar el trabajo. Caso contrario, utiliza el clásico

```
make all
```

y dispondrá del administrador alternativo. Esto fue implementado utilizando el poder del *Makefile*, que con una directiva *-D*. nos permitió usar *#ifndef*, e *#ifdef* para compilar con un administrador o el otro. (Si se desea ver la implementación, dirigirse al archivo ubicado en *Kernel/Makefile*.)

2.3 Procesos, Context Switching y Scheduling

El desafío mas importante que tuvimos que enfrentar fue el desarrollo del scheduler. Si bien la lógica en sí de un scheduler *Priority Based Round Robin* no es imposible de entender, a la hora de desarrollarlo en código presentó una demanda importante. Esto se debió principalmente a poder realizar efectivamente cambios de contexto y reestablecer contextos de ejecución previos. Para esto, primero definimos la utilización de una lista de nodos, donde cada nodo contenía el contexto de ejecución (*PCB*), el estado del mismo (Preparado, Bloqueado, o Finalizado), y un puntero al siguiente nodo. A su vez, creamos un *header* para la lista, con punteros al primero y al último, y con parametros para definir la cantidad de procesos totales y los listos. Todo esto se puede apreciar en la imagen n.



```
typedef struct PNode{
    struct PNode * next;

    uint64_t pid;
    uint64_t ppid;

    State state;

    void * rsp;
    void * rbp;

    int fd[2];
    int priority;
    int fg;

    char name[NAME_SIZE];
    int argc;
    char ** argv;
} PNode;

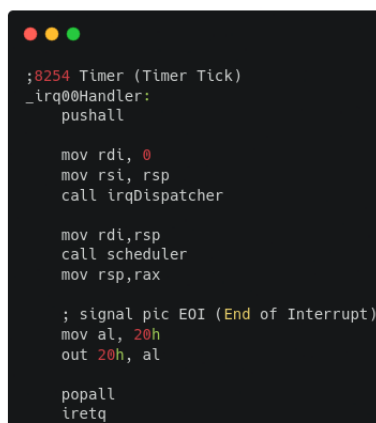
typedef struct PList{
    PNode * first;
    PNode * last;

    uint32_t size;
    uint32_t prepared;
} PList;
```

Imagen 4: Estructura de los nodos del sistema de scheduling (El archivo fuente se encuentra en *Kernel/sync/schedule.c*)

La prioridad de los procesos fue dada con la cantidad de *ticks* que tenían para ejecutarse. De esta manera, una "mayor" prioridad permitía tener mas ticks de ejecución.

El cambio de contexto, como fue mencionado, fue el palo en la rueda mas problemático durante el desarrollo del trabajo, tanto esta vez como el año pasado. Nuestra intención esta vez fue mantenerlo tan simple como era posible: tomar el *RSP* anterior, llamar al scheduler, y setear la respuesta en *RSP* nuevamente.



```
;8254 Timer (Timer Tick)
_irq00Handler:
    pushall

    mov rdi, 0
    mov rsi, rsp
    call irqDispatcher

    mov rdi, rsp
    call scheduler
    mov rsp, rax

    ; signal pic EOI (End of Interrupt)
    mov al, 20h
    out 20h, al

    popall
    iretq
```

Imagen 5: Función *handler* en *assembler* que administra la funcionalidad del sistema cuando se produce una interrupción de *TimerTick* (El archivo fuente se puede encontrar en *Kernel/asm/interrupts.asm*)

Para dejar al scheduler "suspendido", o en estado *idle*, creamos un proceso cuando configuramos el Kernel llamado "*haltP*", cuya funcionalidad es hacer nada. Lógicamente, este proceso tiene un tratamiento especial, dado que no podemos encolarlo como un proceso más. Su función a nivel conceptual consiste en mantener al scheduler "haciendo algo" mientras no hay procesos para correr. Es por eso que, si se analiza la función *scheduler*, se puede encontrar que la única manera de posibilitar su activación, es que no haya un proceso corriendo actualmente, y que no existan procesos en estado *listo*.

2.4 Sincronización

Como mecanismo de sincronización de procesos, se utilizaron semáforos, que permitían el control de acceso de procesos a sectores críticos, como, por ejemplo, en el caso de el problema de los filósofos, que dos procesos no evalúen simultáneamente si pueden tomar o no los tenedores. Para la implementación, se utilizó un struct con la información básica de un semáforo (un identificador, un valor, los procesos bloqueados). También implementado mediante una lista, lo que destaca de este struct es el campo mutex. Como estudiamos en la parte teórica de la materia, el mutex es quien bloquea a dos procesos de acceder simultáneamente a una sección crítica. Para esto, creímos que era interesante utilizar la instrucción `_xchg`, que utiliza el método de **Test and Set** para controlar esto. Es lo que denominamos como sincronización por hardware. Al tratarse de una instrucción atómica, permite que los procesos que se encuentren escuchando de este semáforo no puedan acceder al mismo a la vez. Las instrucciones `acquire` y `release` justamente se encargan de esto.

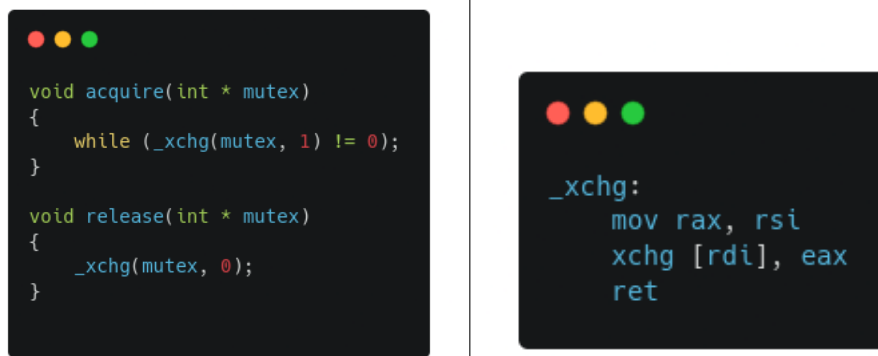


Imagen 6: Instrucciones *Acquire* y *Release* del lado de C (ubicados en el archivo *Kernel/misc/lib.c*), y a la derecha la implementación de la instrucción *Test and Set* (ubicado en *Kernel/asm/libasm.asm*)

2.5 Inter Process Communication

Como método de IPC, se buscó implementar un sistema de *pipes* que permitieran a dos procesos comunicarse entre sí. Se modelaron los pipes como buffers, de los cuales los procesos podían estar asociados como lectores o escritores. Si bien la función de escritura permite el pasaje de strings en vez de chars individualmente, la implementación tanto de escritura como de lectura es char a char. Lógicamente, los pipes se encuentran apropiadamente seteados con semáforos para que dos procesos no puedan ni leer ni escribir concurrentemente.

Un desafío importante que se planteó con los pipes fue como realizar la escritura cuando se realiza desde frontend. Es decir, un proceso quiere hacer un write, ¿cómo redirigirlo hacia pantalla o hacia un pipe? Para esto, se implementó, en el struct *PCB* el campo *FD*, y por otro lado se crearon las funciones *getCurrentInFD()* y *getCurrentOutFD()*, que indican pertinentemente los FD de escritura y lectura del proceso ejecutándose actualmente. Luego, realizando las syscalls pertinentes, podíamos indicarle al proceso que escriba a un pipe de ID conocido, o a *STDOUT*, y la misma lógica aplicó a la lectura.

3 Frontend

3.1 Aplicaciones de User Space

Para las aplicaciones de User Space, se debió realizar un trabajo profundo sobre la base con la que contábamos. Siendo un tanto tajantes, el frontend con el que contamos para el trabajo de "Arquitectura de Computadoras" resultó completamente inservible, por lo que hubo que comenzar prácticamente de 0. Esto se debió a diversos factores: primero, el uso de procesos para los comandos en vez de utilizar un simple llamado a función local dentro del front. Se debió crear un proceso cada vez que se quiere ejecutar un comando, lo que llevó a la implementación de un nuevo sistema de parseo y ejecución. A su vez, otro factor clave es que estos procesos deben poder pipearse entre si, y también poder correr en *background*, por lo que virtualmente no quedo nada de la base previa. Debido justamente a la implementación de procesos, todas estas funciones tuvieron que cambiar su declaración por el estándar para programas de C con *argc* y *argv*.

Si bien podríamos ahondar en explicaciones sobre la reversión de los puntos mencionados anteriormente, tales como el cambio de estructura para parseo, creación y ejecución de procesos, pipeo, entre otras cosas, creemos que lo mas relevante dentro del frontend fue la implementación del famoso "Problema de los filósofos". Al utilizar procesos que necesitan coordinarse correctamente, es un gran desafío para poner todo lo estudiado y construido en practica. Consideremos por un lado que el problema utiliza un proceso *philo* cada vez que se instancia un filósofo nuevo, por lo que se necesita que la creación y sincronización de procesos se encuentre operante. Por otro lado, estos procesos buscarán atentar contra la sincronización de los mismos, ya que deberán chequear el estado de dos procesos "vecinos en la mesa", sumando una nueva responsabilidad en cuanto al funcionamiento de los semáforos. A su vez, todos estos procesos (no exclusivamente los procesos de esta porción si no en general) utilizan frecuentemente *Malloc* y *Free* para pedir y liberar memoria, por lo que también será necesario que se encuentre completamente funcional. Y un detalle mas es que, de la manera en que lo implementamos, el proceso de impresión y todos los procesos filósofos corran en *background* (logicamente el proceso que debe tener el CPU debería ser el controlador de los filósofos para escuchar ante un agregado o borrado de alguno). Por lo que esto también debería funcionar de manera efectiva.

Con todo este preámbulo, podemos pasar a lo que fue la implementación, dado que su análisis es pertinente y englobante de lo que fue el trabajo en frontend. Para hacer un "camino" de partida, primero se crea el proceso *"runPhilos()"*, que se encarga de la administración de la impresión, creación y borrado de los filósofos, y de cerrar el juego oportunamente. Para esto, debe estar escuchando por una interrupción de teclado pertinente todo el tiempo, por lo que logicamente correrá en *foreground*. Mientras no lea una interrupción de teclado que le indique la finalización, seguirá leyendo.

```

void lifecycle(int argc, char *argv[]){
    int idx = atoi(argv[1]);
    while (working){
        // Thinking...
        // Time to eat
        attemptForForks(idx);
        // Eating...
        sleep(1);
        // Done!
        releaseForks(idx);
        // Sleeping...
        sleep(1);
    }
}

void attemptForForks(int i){
    semWait(tableMutex);
    philos[i]->State = HUNGRY;
    checkForForks(i);
    semPost(tableMutex);
    semWait(philos[i]->sem);
}

void releaseForks(int i){
    semWait(tableMutex);
    philos[i]->State = THINKING;
    checkForForks(LEFT(i));
    checkForForks(RIGHT(i));
    semPost(tableMutex);
}

void checkForForks(int i){
    if (philos[i]->State == HUNGRY
        && philos[LEFT(i)]->State != EATING
        && philos[RIGHT(i)]->State != EATING){
        philos[i]->State = EATING;
        semPost(philos[i]->sem);
    }
}

```

Imagen 7: Funcionalidad de proceso de filósofos. (Archivo fuente ubicable en *Userland/SampleCodeModule/misc/phylo.c*)

Como se observa en la imagen, los procesos individuales de cada filósofo hacen recurrentes intentos por adquirir los tenedores. Su ciclo de vida, representado unicamente por comer, pensar, y dormir (quién pudiera), consiste en preguntar si puede adquirir los tenedores. Debe realizar un wait para poder consultar el estado actual de la mesa, y luego pregunta si los filósofos a su izquierda y derecha se encuentran comiendo. Si la respuesta es negativa, simplemente cambia su estado y postea su semáforo (es consultable ahora). Y luego para liberarlos, hace un procedimiento similar. Toma el mutex de la mesa, pasa a pensar, y permite que sus procesos izquierdo y derecho consulten si pueden tomar los tenedores (esto es posible dado que contamos con el mutex de la mesa bajo nuestro "control", ningún otro proceso puede consultarlo). Finalmente hace un *post* al mutex de la mesa, y vuelve a comenzar el ciclo.

4 Instrucciones de compilación y ejecución

Para consultar las instrucciones de compilación y ejecución, referirse al README.md que se encuentra en el repositorio. Allí se encuentra desarrollado de manera detallada cada instrucción necesaria para la ejecución y compilación de cada paso.

5 Conclusión

Así como dijimos el cuatrimestre pasado, se trató de uno de los trabajos prácticos mas desafiantes que nos tocó enfrentar.

Si bien luego de prácticamente un año de trabajo, ciertas cosas empezaron a fluir, no se puede decir que fue un trabajo simple bajo ningún concepto. Sirvió para afianzar conceptos de todo lo estudiado en la teórica, desde scheduling hasta semáforos.

Conllevó mucho tiempo de elaboración, y mas que nada frustraciones (no poder debuggear este trabajo mas que con el GDB nos llevó a apelar a esto en reiteradas ocasiones.)