



Instituto Tecnológico de Buenos Aires

**CONSTRUCCIÓN DEL NÚCLEO DE UN SISTEMA
OPERATIVO Y ESTRUCTURAS DE
ADMINISTRACIÓN DE RECURSOS**

Sistemas Operativos - Curso 2023 - TP2

Autores:

Tomás Marengo - 61587

Ian James Arnott - 61267

Juan Adolfo Rosauer Herrmann - 61240

1. Decisiones tomadas durante el desarrollo

A continuación resumimos la implementación de las funcionalidades más importantes según la teoría de sistemas operativos.

1.1. Sobre el Algoritmo de Scheduling

En principio se definen **PRIORITY_MIN = 10**, **PRIORITY_MAX = -10**, **PRIORITY_DEFAULT = 0** y **PRIORITY_IMPORTANT = -5**. Los primeros dos, arbitrarios; el default deduciblemente en el centro del intervalo; y el último, cercano a MAX. El algoritmo proporciona a cada proceso un quantum según su prioridad tal que:

$$\begin{cases} t = \text{tiempo entre interrupción del timer,} \\ \text{Quantum}(pid) = (PRIORITY_MIN - Priority(pid)) * t \end{cases}$$

De esta manera, la única diferencia entre procesos con más o menos prioridad es la cantidad de tiempo que pueden correr "sin interrupciones". Además, siempre toma el proceso siguiente (en pids) que esté ready, al terminar el quantum del proceso actual (ver *getNextReadyPid()*). Esto es una alternativa a que procesos con mayor prioridad ejecuten primero, pero más simple de implementar, ya que no existe forma de que siempre corra el mismo proceso habiendo varios y que produzca inanición u otros problemas, por lo que no hay que tener en cuenta esos casos límites en la programación del algoritmo.

Luego, se definieron **PSEUDOPID_KERNEL = -1** y **PSEUDOPID_NONE = -2** para cuando se inicia el sistema o cuando la shell está bloqueada (la shell no puede recibir un kill). Esto se junta con la variable **forceRunNextPID** produciendo:

1. Al iniciar el sistema,

forceRunNextPID = *PSEUDOPID_NONE*

currentRunningPID = *PSEUDOPID_KERNEL*

2. Si se desbloquea un proceso con *pid* = *x* con *Priority* ≤ *PRIORITY_IMPORTANT*

⇒ *forceRunNextPID* = *x*

3. Si no hay procesos ready, *currentRunningPID* = *PSEUDOPID_KERNEL*

Algo más para desarrollar quizás sea el *yield()*, que deja el quantum en 0 y provoca una interrupción que llama a *switchProcess()*, para que se elija a otro proceso.

1.2. Sobre File Descriptors: Pipes, Keyboard, Screen

Otra de las funcionalidades importantes a mencionar es el manejo de recursos y sus respectivos file descriptors. En los archivos donde se utilicen file descriptors, por ejemplo `graphics.c`, `keyboard.c` o `pipes.c` se encontrarán handlers de `add`, `close`, `read`, `write`, y/o `dup` para manejar a estos.

1.2.1. Añadir un File Descriptor

Los handlers de `add` son wrappers de `addFd()`, en `process.c`. `addFd()` guarda en una tabla para cada proceso, el filedescriptor asociado a un recurso, junto con los handlers de `close`, `read`, `write` y/o `dup` de dicho recurso, para saber como utilizarlos en su posteridad.

1.2.2. Cerrar/Eliminar un File Descriptor

La `syscall close` simplemente es un wrapper de `deleteFd()`, que elimina dicho recurso de la tabla de file descriptors de un proceso en particular.

1.2.3. Leer un File Descriptor

Como se mencionó, los recursos que pueden leerse, le darán a `process.c` un handler de `read` indicando cómo se deben leer. A continuación, se mencionan los handlers de lectura que se encontrarán en el código.

1. **Keyboard:** Si hay caracteres para leer, listo. Si no hay, se agrega el proceso a una lista de espera de lectura, se bloquea y se hace `yield`, para que otro proceso tome el control de la CPU.
2. **Pipe:** De la misma manera que el teclado, los pipes tienen una lista de espera de lectura y escritura, y de la misma manera, si no hay nada para leer, se agrega el proceso a la lista de espera de lectura de ese pipe, se bloquea y se hace `yield`, para que otro proceso tome el control de la CPU. Cuando si hay para leer, luego se desbloquean los procesos que están en lista de esperar para escribir para simular la unidireccionalidad.

1.2.4. Escribir un File Descriptor

Como se mencionó, los recursos que pueden escribirse, le darán a `process.c` un handler de `write` indicando cómo se deben escribir. A continuación, se mencionan los handlers de escritura que se encontrarán en el código (reutilizando el código texto de arriba xD).

1. **Screen:** No cambia mucho del TP de Arqui, sólo que ahora el recurso es un color y se mapea, por ejemplo, el blanco a `STDOUT` y el rojo a `STDERR`. Lo malo de esto es que al ya estar mapeado así desde el kernel, el usuario no puede pedir otro color para imprimir.

2. **Pipe:** Se escribe en el buffer circular del pipe, luego de desbloquean los procesos en lista de espera de escritura.

1.2.5. Dupear un File Descriptor

Se mapea un recurso de un file descriptor de un proceso a un file descriptor de otro proceso. Son todos wrapper de sus funciones de añadir un file descriptor, sólo que en `process.c` se llama al handler de `dup` del proceso (p2) que llama, y no del proceso que se está creando (p1).

```
"p2->fdTable[fd2].dupHandler(pid2, pid1, fd2, fd1, p2->fdTable[fd2].resource)"
```

1.3. Sobre Pipes y Semáforos

1. Se creó un **namer**, una estructura que guarda recursos con nombre en `sem.c` y `pipes.c` para mantener una base de datos de los semáforos y pipes con nombre que existen en el sistema, de esta manera se facilita el CRUD de estos recursos.
2. Se creó un **WaitingQueue**, anteriormente nombrado, para manejar el bloqueo y desbloqueo de procesos que esperan el acceso a semáforos y pipes.

2. Instrucciones de compilación y ejecución

2.1. Compilación

Ejecutar `compile.sh`, en la carpeta root del proyecto (tOS). Si se le agrega el parámetro "buddy", se usará este memory manager, de lo contrario se utilizará el "Free" List.

2.2. Ejecución

Ejecutar `run.sh` para correrlo de manera gráfica con QEMU. Para ello se deberá tener un ambiente apto para ejecutar aplicaciones gráficas, como en Ubuntu o en Windows con WSL2 actualizado a la última versión.

3. Funcionamiento de cada uno de los requerimientos

1. **Dos memory manager:** se puede compilar con uno o con otro. `./compile.sh buddy` utilizará Buddy, de lo contrario se usará Free List. Ambos implementan `malloc` y `free`, además de `realloc`, y un método que retorna información pertinente.

Se puede chequear con el comando *mem* el memory manager que se está usando, además de otra información. Y también utilizar el comando *testmem* para ejecutar el testeo de memoria.

```
>mem
Memory Manager Type: BUDDY
Total memory: 8388608.
Used: 5632 (0%).
Available: 8382976.
```

Figura 1.1: Comando mem utilizando Buddy.

```
>mem
Memory Manager Type: LIST
Total memory: 17825792.
Used: 4736 (0%).
Available: 17821056.
```

Figura 1.2: Comando mem utilizando List.

2. **Crear y finalizar un proceso, con soporte de pasaje de parámetros:** se puede ejecutar el comando *"loop &"* para crear un proceso que se supone que imprimiría cada 3 segundos su pid en pantalla, lo que pasa hicimos que no pueda, ya que está en background (esto se podría cambiar comentando dos líneas de código). Igualmente se puede ahora ejecutar *ps* para ver que sigue vivo y conocer su PID, para luego *"kill <PID>"* (pasaje de parámetros) y matarlo.
3. **Obtener el ID del proceso que llama:** cualquiera puede llamar a *sys_getpid()* para saber su PID, como lo hace loop.
4. **Listar todos los procesos, matar a un proceso:** ya mencionado.
5. **Bloquear y desbloquear un proceso:** se puede seguir probando con *"loop &"*. Luego se ejecuta *ps* para saber su PID. Por último se puede bloquear con *"block <PID>"*, y desbloquear con *unlock <PID>"*.

```
>loop &
>ps
PID=0 Name=shell Status=RUNNING Priority=-10 Foreground=1 stackEnd=F80FF8 stackStart=F81FF8 RSP=F81CB0
PID=1 Name=loop Status=READY Priority=0 Foreground=0 stackEnd=F81FF8 stackStart=F82FF8 RSP=F82E50
>block 1
Blocked process with PID 1.
>ps
PID=0 Name=shell Status=RUNNING Priority=-10 Foreground=1 stackEnd=F80FF8 stackStart=F81FF8 RSP=F81CB0
PID=1 Name=loop Status=BLOCKED Priority=0 Foreground=0 stackEnd=F81FF8 stackStart=F82FF8 RSP=F82E50
>unlock 1
Unblocked process with PID 1.
>ps
PID=0 Name=shell Status=RUNNING Priority=-10 Foreground=1 stackEnd=F80FF8 stackStart=F81FF8 RSP=F81CB0
PID=1 Name=loop Status=READY Priority=0 Foreground=0 stackEnd=F81FF8 stackStart=F82FF8 RSP=F82E50
```

Figura 1.3: Bloqueando y desbloqueando un proceso.

6. **Renunciar al CPU y esperar a que los hijos terminen:** userland posee las syscalls `sys_yield()` y `sys_waitpid()`. Se utilizan en `testSync`.
7. **Implementar semáforos:** se puede testear con el comando `"testsync [n] [use_sem]"`. El `testsync` usando semáforos (`use_sem=1`) da el resultado correcto. Sin utilizar semáforos nos da el mismo n, pero no llegamos a testearlo bien como para saber por qué daba eso. También se puede imprimir información importante con el comando `"sem"`

```
>testsync 100000 1
Final value: 0

>testsync 100000 0
Final value: 100000
```

Figura 1.4: Output testsync.

```
>testsync 1000000 1 &

>ps aux
PID=0  Name=shell      Status=RUNNING  Priority=-10    Foreground=1   stackEnd=F000A0  stackStart=F010A0  RSP=F00D50
PID=1  Name=testSync    Status=BLOCKED  Priority=0      Foreground=0   stackEnd=F012A8  stackStart=F022A8  RSP=F02050
PID=2  Name=processDec  Status=READY   Priority=0      Foreground=1   stackEnd=F02650  stackStart=F03650  RSP=F03490
PID=3  Name=processInc  Status=BLOCKED  Priority=0      Foreground=1   stackEnd=F03878  stackStart=F04878  RSP=F046B0
PID=4  Name=processDec  Status=BLOCKED  Priority=0      Foreground=1   stackEnd=F04B18  stackStart=F05B18  RSP=F05950
PID=5  Name=processInc  Status=READY   Priority=0      Foreground=1   stackEnd=F05DB8  stackStart=F06DB8  RSP=F06BF0

>sem
Listing 1 semaphore:
name=sem, value=0, linkedProcesses=4, processesWQ={4, 5}
>sem
Listing 1 semaphore:
name=sem, value=0, linkedProcesses=4, processesWQ={5, 4}
>sem
Listing 1 semaphore:
name=sem, value=0, linkedProcesses=4, processesWQ={3, 4}
```

Figura 1.5: Output sem.

8. **Implementar pipes:** Se puede probar directamente en la consola. Por ejemplo ejecutando `"mem@filter"` y sólo `"mem"` se obtienen las siguientes salidas. También se puede imprimir información importante con el comando `"pipe"`

```
>mem@filter
Mmry Mngr Typ: LST
Ttl mmry: 17825792.
sd: 5080 (0%).
ulbl: 17820712.
>mem
Memory Manager Type: LIST
Total memory: 17825792.
Used: 4736 (0%).
Available: 17821056.
```

Figura 1.6: Comando mem y filter "pipeados".

9. **Símbolos de foreground y pipe:** usamos & y @ respectivamente, pero se pueden cambiar rápidamente en defs.h.
10. **CTRL+D, CTRL+C:** se puede probar CTRL+C ejecutando *loop* en foreground y ver que deja de imprimir. Se puede probar CTRL+D con *wc*, ya que este termina al encontrar el EOF.
11. **Aplicaciones User space:**

help: le agregamos los correspondientes programas.

```
>help
The available commands are:
'help' - Displays a list of all available commands.
'clear' - Clears the window.
'echo' - Prints the parameters passed to it to standard output.
'time' - Displays the system's date and time.
'mem' - Displays information about the system's memory.
'ps' - Displays a list of all the currently running processes with their properties.
'loop' - Creates a process that prints it's PID once every 3 seconds.
'kill' - Kills the process with the parameter-specified PID.
'nice' - Changes the priority of the process with the parameter-specified PID.
'block' - Blocks the process with the given PID.
'unblock' - Unblocks the process with the given PID.
'sem' - Displays a list of all the currently active semaphores with their properties.
'cat' - Creates a process that prints the standard input onto the standard output.
'wc' - Creates a process that counts the newlines coming in from standard input and prints that number to standard output.
'filter' - Creates a process that filters the vowels received from standard input and prints them to standard output.
'pipe' - Displays a list of all the currently active pipes with their properties.
'testmm' - Runs a test for memory manager.
'testsync' - Runs a synchronization test with multiple processes with semaphores.
'testprocesses' - Runs a test for processes.
'testprio' - Runs a test on process priorities.
'phylo' - Runs the philosopher, add one philosopher with "a", remove one philosopher with "r".
```

Figura 1.7: I need somebody.

ps, loop, kill, nice, block: ya mencionados.

cat: funcionando, lo único es que no se comporta **específicamente** como en linux ya que este vuelve a imprimir lo que usuario escribió al presionar enter.

wc, filter: ya mencionados.

phylo: con como están los define actualmente (8 procesos máximos, arbitrario), solo se podrían crear hasta 6 filósofos, ya que uno es la shell y el otro proceso es el manejador de los filósofos. Se pueden agregar filósofos con la letra “a”, y removerlos con la letra “r”. Previamente se debería cambiar la constante que define un límite a la cantidad de procesos.

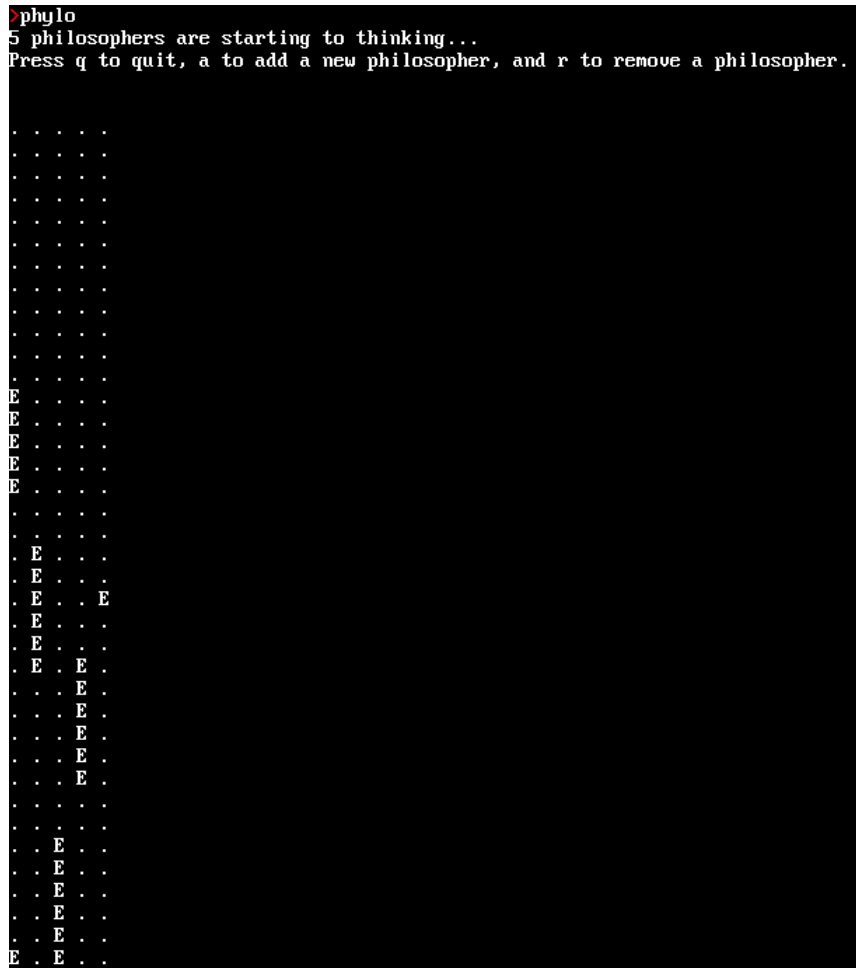


Figura 1.8: Output phylo.

4. PVS Studio Output

1. *"(warning) %d in format string (no. 2) requires 'int' but the argument type is 'unsigned int'."* Warnings en la carpeta bmfs.c que no nos corresponde ya que es algo del Bootloader. Esto también aplica a otros warnings o styles que son de carpetas que no tocamos (Toolchain).
2. *"When using void pointers in calculations, the behaviour is undefined."* Se puede usar void* en un ambiente que vos controlás y sabés que recibe cada cosa, por lo que no es problema.
3. *"The scope of the variable 'total'/rq can be reduced."* Ponerlas más abajo? Innecesario, mejor visualización de las variables.

4. *"The function XXX is never used."* Los últimos son todos de este estilo, la mayoría son funciones que usa pero no las está contemplando bien, y otras son funciones que están de librería porque son importantes, como itoa(). No molestan y son útiles si se quisiera seguir con el proyecto.

5. Limitaciones

1. No sabemos si se cuenta como limitación, pero el block terminó como block y unblock, por si querés probar a bloquear un proceso y este quedara siempre bloqueado.
2. Como se mencionó, el usuario no tiene forma de cambiar los colores en los cuales imprime.
3. Cantidad de procesos al mismo tiempo limitado por una constante definida.
4. Desconocemos el porqué de los resultados de testsync sin semáforos.

6. Problemas encontrados en el desarrollo

La primera etapa de nuestro desarrollo tuvo complicaciones graves. En un principio decidimos utilizar el sistema operativo base de uno de nuestros integrantes, pero al avanzar y comenzar a implementar las funcionalidades en el Kernel, rápidamente nos encontramos con dificultades de compilación y manejo de recursos/interrupciones. Al agregar alguna variable o archivo que ni siquiera interfería en el sistema, obteníamos resultados distintos, sin sentido. Luego de vernos inhabilitados a continuar por problemas surgidos de las implementaciones previas, como grupo decidimos iniciar nuevamente, con un TP de Arqui de otros compañeros.