



Trabajo Práctico N° 2:  
Construcción del Núcleo de un  
Sistema Operativo

Grupo 2:

Della Sala, Rocío 56507  
Giorgi, Maria Florencia 56239  
Rodriguez, Ariel Andrés 56030  
Santoflaminio, Alejandro 57042

# **Índice**

Introducción .....	4
Instrucciones de compilación .....	4
Instrucciones de ejecución .....	4
Physical Memory Management .....	5
Implementación .....	5
Funcionamiento .....	5
System calls.....	5
Problemas encontrados y soluciones.....	6
Limitaciones.....	6
Test .....	6
Procesos, Context Switching y Scheduling.....	8
Implementación.....	8
Funcionamiento.....	8
System calls.....	9
Problemas encontrados y soluciones .....	9
Limitaciones.....	10
IPCs.....	11
Pipes.....	11
Funcionamiento.....	11
System calls.....	11
Problemas encontrados y soluciones.....	12
Limitaciones.....	12
Test.....	12
Mutexes.....	13
Funcionamiento.....	13
System calls.....	14

Test.....	14
Aplicaciones de UserSpace.....	15
Importante.....	17
Conclusión .....	17

## **Introducción**

El trabajo práctico consistió en extender el TP especial de la asignatura anterior, Arquitectura de Computadoras (72.08), implementando nuevos mecanismos que el mismo carecía, como Scheduling, Memory Management e IPC y además agregando nuevas aplicaciones de usuario.

## **Instrucciones de compilación**

Para compilar el sistema será necesario dirigirse a la carpeta principal del trabajo práctico y ejecutar el comando `make all` en la terminal. Esto será suficiente para compilar todos los módulos con los flags `Wall` y `pedantic` que fueron agregados a los respectivos *Makefiles*.

También es posible compilar el sistema mediante Docker. Para ello debemos dirigirnos a la carpeta Docker dentro de la carpeta del trabajo. Luego hacemos `start docker.service` si es que docker no estaba inicializado. Ejecutamos `Docker build . -tag 'ejemplo:1.0'` (siendo 'ejemplo' el nombre que le queremos dar a nuestro build y 1.0 la versión). A continuación, realizamos `docker run -v /home/miusuario/carpetaTP:/root/tp2 -ti ejemplo:1.0` (siendo el directorio antes de los ":" donde está ubicado el TP en nuestra PC). Una vez en Docker hacemos `cd tp2` y `make all`.

## **Instrucciones de ejecución**

Para ejecutarlo lo primero que tenemos que hacer es ejecutar el comando `./run.sh` en la terminal. Una vez hecho esto, se abrirá QEMU. Lo primero que vamos a ver es el prompt de la terminal. En este punto podemos realizar dos cosas; o ejecutar comandos (usar `help` para ver los comandos que hay en la terminal) o ejecutar programas de usuario (usar el comando `ls` para ver cuales se pueden correr).

Para correr un proceso será necesario anteponer `./` al nombre que aparece en la lista. Si queremos correrlo en foreground, bastará con poner `&` al final del mismo. Por ejemplo: `./linearGraph&`. Tener en cuenta que no todos los procesos pueden correrse en background. Más adelante se detalla cada una de las aplicaciones de usuario del sistema y su funcionalidad.

# **Physical Memory Management**

## *Implementación*

En la implementación del Memory Manager, decidimos realizar un Buddy allocator. Esto fue debido a que en comparación a los demás algoritmos que investigamos, éste es el que mejor puede administrar la memoria para la reserva y liberación de páginas de un sistema operativo a largo plazo. Su principal ventaja es la estructura de árbol binario que utiliza para la búsqueda y liberación/reserva de memoria.

## *Funcionamiento*

El funcionamiento del sistema es bastante sencillo de entender:

Para la reserva de páginas basta con llamar a la función `allocPage(int size)`, la cual primero, por un lado transformará `size` en una potencia de 2, si es que no lo era, y luego buscará en el árbol binario algún nodo que esté disponible y que tenga suficiente espacio para lo que se pide reservar. Cabe resaltar que, en este algoritmo, si ve que un nodo está disponible y tiene un espacio suficiente como para reservar a `size` 2 veces, dicho nodo tendrá 2 hijos con la memoria partida; el hijo izquierdo tendrá la primera mitad de posiciones de memoria y el hijo derecho la segunda mitad de posiciones de memoria. Este paso se repetirá recursivamente hasta que el nodo no pueda subdividirse más.

## *System calls*

Nombre	Número	Acción
Sys_malloc	10	Reserva la cantidad de memoria pedida

Por otro lado, cualquier proceso al finalizar, hace un `free` de todas las páginas que reservó (1 página para su `stack` y todas las páginas que pidió para su `heap`). Esta función es llamada por la syscall `sys_pkill` la cual finaliza un proceso. Se hace referencia a esta syscall más adelante. Al igual que un Buddy allocator, buscará el nodo a liberar, y si su “hermano” también tiene todas sus posiciones libres, entonces estas 2 particiones se fusionarán en una sola, lo que se traduce a que, el padre no tendrá más a sus hijos

## *Problemas encontrados y soluciones*

El mayor problema al que nos enfrentamos fue a la hora de juntar el Memory Manager con el TP de Arquitectura de Computadoras, puesto que no entendíamos en un inicio como era que los programas pedían memoria periódicamente. Luego de consultarlo con los docentes, entendimos cómo crear y manejar el heap de cada programa (incluido el kernel) y pudimos unificar ambos.

## *Limitaciones*

- La mayor limitación del Buddy allocator que implementamos, es que las operaciones de búsqueda y liberación pueden optimizarse aún más para generar un memory manager mucho más rápido en tiempo de ejecución.
- El sistema no deja que un usuario reserve una cantidad mayor a 1 página por puntero. Es decir, como máximo puede reservar 4096 bytes por puntero. Esta limitación si bien no es del memory manager en sí, ya que él puede reservar cualquier cantidad de memoria menor a 1GB, es una modificación que debe realizarse para que el Buddy allocator funcione en su totalidad.
- Otra limitación por fuera del Buddy allocator, pero que nos parece importante resaltar en este apartado es que actualmente, tenemos un heap de kernel estático. Esto se hizo más que nada para controlar la cantidad de estructuras que manejará el sistema operativo (processSlots, pipes, mutexes, etc) ya que, si genera infinitas, podría generar conflictos de memoria. Por otro lado, como aspecto positivo, es importante decir que todas estas estructuras se “reciclan”. Cuando un proceso finaliza todas sus estructuras son liberadas, para que otro proceso pueda reemplazar dichas estructuras por las que éste necesite

## *Tests*

Para el apartado de testeos, dentro de la carpeta de *Userland*, hay un módulo llamado “*testMemoryManager*” el cual es un programa que cuenta con 4 testeos los cuales serán comentados a continuación:

1. El primero, consta de pedir una cierta cantidad de memoria. Dicho puntero debería tener una posición de memoria retornada por el Buddy allocator. Si dicho puntero no es NULL, el test es satisfactorio.  
Lo llamaremos *puntero1*. *Página1* apunta a *puntero1*.
2. El segundo, consta de pedir una segunda cantidad de memoria y comparar que el puntero que se recibe apunta a *Página1*+ la cantidad de memoria

reservada por *puntero1*. Si dicha condición se cumple el test es satisfactorio. Llamaremos a este punteropuntero2.

Hasta este punto ambos punteros deben entrar en una página del heap.

3. El tercero, consta de pedir una cantidad de memoria que supere la cantidad que puede guardarse en la primera página del heap del proceso. Es decir, si el *puntero1* y *puntero2* pidieron en total 4000 bytes, el *puntero3* debe pedir entre 97 bytes y 4096 bytes, para que de esta manera el puntero busque ser alojado en una segunda página del heap. En este test, se compara que los 3 punteros sean diferentes y que el *puntero3* apunte a la segunda página que el Buddy allocator le da. Lo llamaremos *puntero3*.
4. El cuarto, consta de pedir una cantidad de memoria que esté entre la cantidad que pidieron el *puntero1* más el *puntero2* y 4096 bytes. Este test tiene el objetivo de mostrar que, si bien ya un puntero apunta a la segunda página reservada por el proceso, los bytes que quedaron en la primera página del heap no se desperdician. Se comprueba que los 4 punteros son distintos y que el *puntero4* apunta a la siguiente posición del *puntero2*.  
Ejemplo: Si es el *puntero1* reservó 5 bytes y el *puntero2* reservó 5 bytes, entonces *puntero4* apunta a 0x6700A.

Aclaración: El test puede correrse en cualquier momento en el Sistema Operativo y debería darse que todos los tests sean satisfactorios.

## Procesos, Context Switching y Scheduling

### *Implementación*

En el trabajo práctico, cada proceso se representa con una estructura llamada *Process* que contiene aquella información necesaria para que el sistema operativo pueda administrarlo: el PID, el estado, el nombre, un puntero al stack, su heap, entre otras.

Estos procesos se agrupan en una tabla dinámica de *ProcessSlot* como “*slots*”, que representaría la tabla **PCB** (Process Control Block) vista en la teoría. Cada slot contiene un proceso y un puntero al siguiente. Con la ayuda de esta tabla, el Sistema Operativo sabe qué procesos están actualmente corriendo en memoria, cuales están listos, cuales están bloqueados y cuales finalizaron.

A su vez, implementamos un **Round Robin**, con un quantum definido con un define. Creamos un puntero *currentProcess* a la tabla que indica cual es el proceso que se debe correr (si su estado lo permite) en ese momento. Este algoritmo nos pareció el más sencillo de implementar que cumpla con la condición de ser pre-emptivo y soportar multitasking.

### *Funcionamiento*

En primer lugar se crea un proceso mediante la función *createProcess*, que crea un “slot” y lo agrega a la tabla. Cabe aclarar que nuestro proceso principal es shell, que está siempre corriendo, y cualquier proceso que se cree se ubicará en la última posición de la tabla.

Una vez que el timertick interrumpe al procesador, este pasa a ejecutar una rutina de atención de interrupción que lo que hace es realizar el cambio de contexto para correr el siguiente proceso de la tabla. En primer lugar, guarda el estado del proceso actual pusheando los registros al stack del proceso correspondiente. Además, como la estructura del proceso guarda el valor de *esp* en una variable llamada *userStack*, la rutina llama a *switchUserToKernel* para guardar ese valor.

Esta función devuelve el valor de *esp* del stack del kernel, para que, en segundo lugar, se proceda a ejecutar la función *runScheduler*. La misma evalúa el número de ticks del proceso actual. Si este es igual al quantum, entonces es momento de cambiar de proceso y busca en la tabla hasta encontrar alguno que se encuentre en estado READY y lo procede a ejecutar. Para ello, retorna a la rutina de atención de interrupción y llama a *switchKernelToUser* que lo que hace es posicionar el registro *esp* al stack del proceso actual. Por último, restaura los registros de dicho proceso y sale de la interrupción.



## System calls

Nombre	Número	Acción
sys_createProcess	12	Crea un proceso
sys_ls	13	Lista los procesos
sys_pkill	14	Elimina un proceso
sys_killPID	25	Elimina un proceso por pid, por consola.

## Problemas encontrados y soluciones

- **Terminar un proceso:** Cuando se ejecutaba un proceso y este llegaba al return 0, el sistema quedaba colgado. Por ende, no podíamos continuar administrando el resto de los procesos que estaban listos para correr. Lo que se realizó fue una función *exitProcess* al final de cada uno, que lo que hace es una syscall. El sistema operativo pone dicho proceso en estado FINISHED y luego lo elimina de la tabla de procesos, liberando las páginas usadas por éste y procediendo a correr el siguiente. Para ello se realizó una función en assembler *restoreContext* que posiciona el registro *esp* al stack del siguiente proceso a ejecutar.
- **Buffer de teclado:** El trabajo práctico está implementado de tal forma que tenemos en kernel un buffer para stdin. El mismo se llena a medida que el usuario aprieta teclas y se consume a medida que algún programa de usuario hace getchar. Como este buffer es compartido por todos los programas, se nos presentó el problema de que si apretamos teclas mientras estaba corriendo un programa que no hacía getchar, las consumiría el primero que lo haga. Si escribimos mientras corre un programa en background, esto no parece raro a la vista del usuario. Pero si escribimos mientras corre un programa en foreground, que por ejemplo está graficando, termina y empieza a correr shell, todo lo que escribimos lo consumirá shell (y también lo mostrará en pantalla). Para solucionar esto, cuando termina un programa hacemos una limpieza del buffer.
- **Multitasking:** Para lograr el multitasking, era necesario que dos o más procesos estén corriendo “en simultáneo” (o lograr ese efecto). El problema era; ¿Cómo hacíamos para correr dos o más procesos? ¿Qué íbamos a ver en la pantalla si esto sucedía?. Para esto, se definió a la shell como un proceso FOREGROUND. Como mencionamos antes, este proceso está continuamente corriendo. Lo que sucede cuando corremos otro proceso en la

terminal se puede separar dependiendo si el proceso correrá en BACKGROUND o FOREGROUND.

- Si el proceso va a correr en background, entonces el único proceso que está en foreground es la shell. Podemos seguir ejecutando la misma, corriendo otros procesos, o ejecutando algún otro comando. Los procesos background no tienen acceso a stdin del teclado ni tampoco al video\_driver.
- Si el proceso va a correr en foreground, entonces la shell se bloquea hasta que este finalice, ya que no podrá utilizar los recursos del teclado ni del video. No nos pareció lógico que la shell continúe corriendo si no la podemos usar. Además, esto genera cambio de contexto y tiempo perdido en donde nuestro proceso foreground tardaría más en ejecutarse. Esto no quiere decir que no se puedan correr otros procesos cuando tenemos uno en foreground: antes de correr el mismo, podemos correr otros n procesos en background.

### *Limitaciones*

- No se pueden correr dos procesos en foreground simultáneamente. No encontramos la forma de entender cómo podríamos hacer que esto funcione y no aparezcan ambos solapados en pantalla. Quizá con algún algoritmo de scheduling más trabajado se podría lograr, pero por cuestiones de tiempo decidimos dejarlo así.
- Sabemos que el proceso de PID = 0 no debería ser la shell, debería ser init. Como nos dimos cuenta de esto una vez que el trabajo se encontraba en sus etapas finales, decidimos dejarlo así.
- Si tenemos corriendo n procesos en background y uno en foreground y escribimos en pantalla, por ejemplo, para ingresar algún parámetro o comando, esto se verá retrasado unos milisegundos (o más, dependiendo de cuantos procesos en background tengamos y de cuanto sea nuestro quantum). Esto sucede ya que al estar ejecutándose otros procesos un cierto tiempo, más el cambio de contexto, lo que ingresemos en el teclado recién se podrá imprimir en pantalla cuando volvamos al proceso foreground (qué estará haciendo getchar y luego putchar).
- No todos los procesos se pueden correr en background. Será necesario que no requieran de la entrada estándar. No tiene sentido que se esté corriendo la shell y el proceso en background consuma el buffer para su input, sin que el usuario pueda ver que input le está pidiendo.

## IPCs

### Pipes

#### *Funcionamiento*

Se implementó un sistema de pipes para lograr la comunicación entre 2 procesos de manera parecida a la que se estudió en clase. Cada proceso tiene una cantidad fija de pipes, para permitir que un proceso esté conectado con 1 o más procesos. La cantidad fijada se encuentra dentro de "structs.h". Cada vez que un proceso crea un pipe, se crea una estructura dejando "en blanco" el PID del proceso al que se conectará. Dado que los procesos no tienen conocimiento del PID del proceso al que se quieren conectar, crean el pipe informando el nombre del proceso al que se quieren conectar. Por seguridad, la estructura del pipe tiene guardado el nombre del proceso al que el proceso que creó el pipe se quiere conectar. Esto es para que cuando el segundo proceso cree el pipe, busque en el primer proceso (el que ya creó el pipe) para revisar si ya hay abierto un canal de comunicación y directamente comunicarlos en vez de crear otra estructura.

Por lo dicho anteriormente, podemos decir que nuestro sistema permite comunicaciones cuasi biunívocas (dado que un proceso inicia una comunicación con otro a través de su nombre y no de su PID) entre múltiples procesos (un proceso puede tener más de 1 pipe).

#### *System calls*

Nombre	Número	Acción
sys_pipeCreate;	15	Crea un pipe
sys_pipeWrite;	16	Escribe a través del pipe
sys_pipeRead;	17	Lee a través del pipe
sys_pipeClose	18	Cierra la posibilidad de lectura/escritura del proceso que la invoca
sys_pipeOpen;	19	Abre la posibilidad de lectura/escritura del proceso que la invoca

## *Problemas encontrados y soluciones*

- En un principio se nos dificultó lograr que un proceso pudiera tener más de una vía de comunicación, es decir que tuviera múltiples pipes. Esto se resolvió generando esquemas en papel, para luego codear las respectivas ideas.
- Otra de las dificultades fue lograr que read sea una función bloqueante. La misma se resolvió creando 2 funciones adicionales en scheduler, las cuales bloquean al proceso si en el pipe no hay caracteres para leer y desbloquean el proceso periódicamente, para que el proceso revise si, una vez desbloqueado, hay caracteres en el pipe, sino vuelve a bloquearse. Esta última parte debería modificarse de manera tal que el scheduler desbloquee el programa si y solo si ya hay caracteres para leer.

## *Limitaciones*

La mayor limitación a la que nos enfrentamos a la hora de implementar IPC fue que cuando un proceso crea un pipe, no tiene forma de saber cual es PID del proceso al que quiere conectarse, por ello fue que anteriormente nombramos que la comunicación entre 2 procesos es **cuasi biunívoca**.

## *Tests*

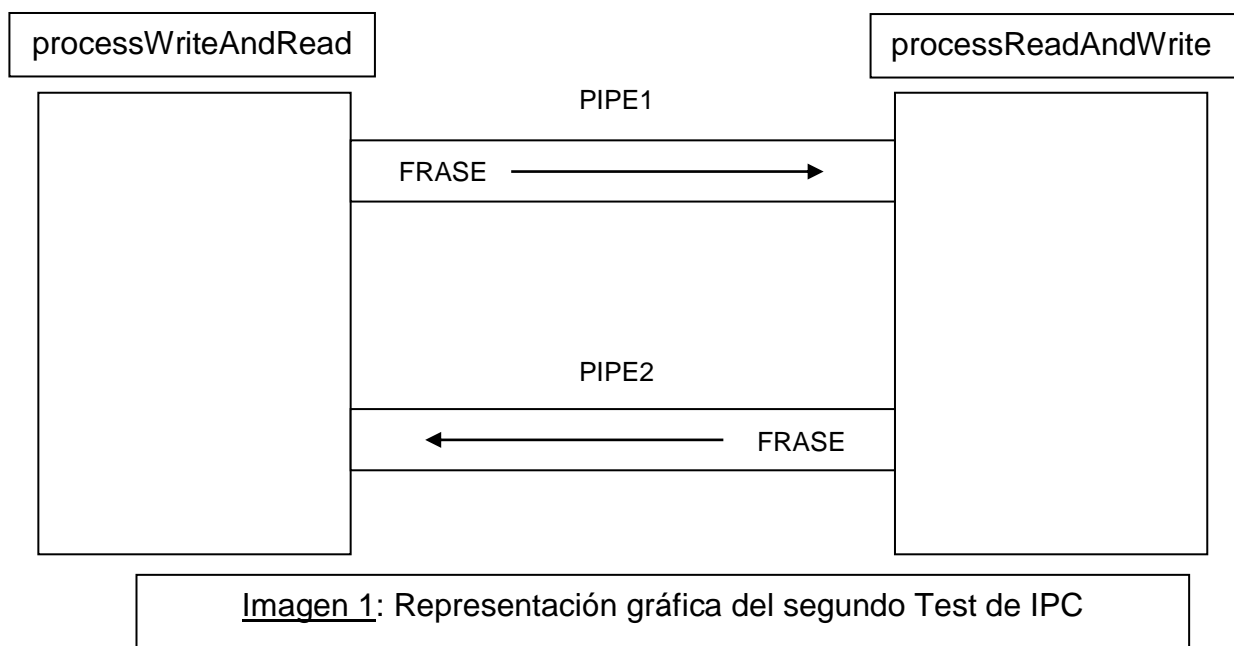
Para verificar el correcto funcionamiento de IPC, se crearon 4 módulos (2 por cada test).

En el primer test, que involucra a los módulos *“processWrite”* y *“processRead”*, se busca que processWrite inicie un canal de comunicación con processRead y le mande dos frases. Luego cuando se invoca al proceso processRead, éste debe conectarse al canal de comunicación, leer el string en dos tandas y verificar que las 2 frases se recibieron correctamente. Este test permite comprobar el correcto funcionamiento de múltiples read y writes.

En el segundo test, que involucra los módulos *“processReadAndWrite”* y *“processWriteAndRead”*, se busca crear 2 canales de comunicación distintos entre estos 2 procesos. La idea es que primero se invoque a processReadAndWrite y se quede bloqueado esperando a que processWriteAndRead escriba una frase. Una vez invocado processWriteAndRead, este escribirá una frase para desbloquear a processReadAndWrite y esperará a que, por otra vía de comunicación, processReadAndWrite le escriba una frase, es decir processWriteAndRead quedará bloqueado. Una vez que processReadAndWrite le retorna la frase por la 2da vía de comunicación, processWriteAndRead se desbloquea, imprime dicha frase y ambos

procesos finalizan. Este test comprueba el correcto funcionamiento de las conexiones múltiples entre procesos y que la función read es bloqueante si no hay caracteres para leer. A continuación, una muestra gráfica del test

1. Se invoca *processReadAndWrite*
2. *processReadAndWrite* intenta leer y se bloquea
3. Se invoca *processWriteAndRead*, se conecta con *processReadAndWrite* a través del pipe1 y le escribe "FRASE", y trata de leer a través de pipe2, quedando bloqueado.
4. *processReadAndWrite* se desbloquea, lee "FRASE", se conecta con *processWriteAndRead* a través de pipe2 y le escribe "FRASE"
5. *processWriteAndRead* se desbloquea, lee "FRASE", la imprime.
6. Ambos procesos finalizan



## **Mutexes**

### *Funcionamiento*

Cada mutex tiene una estructura, la cual posee un nombre, una cola para aquellos procesos que hagan signal y deban esperar, una cola para aquellos procesos que hagan wait y deban esperar, un valor (true o false) y un valor para saber si dicho mutex está disponible para usarse.

Cada vez que un proceso hace un signal, si el valor del mutex era true, el proceso se bloquea y se lo agrega a la cola de signal.

De la misma manera ocurre con aquellos procesos que quieran hacer un wait.

### *System calls*

<b>Nombre</b>	<b>Número</b>	<b>Acción</b>
sys_getMutex	20	Obtiene un mutex por nombre, o lo crea si no existe
sys_wait	21	Cambia el valor a false, si es posible, sino se bloquea
sys_signal	22	Cambia el valor a true, si es posible, sino se bloquea
sys_freeMutex	23	Libera el mutex

### *Tests*

El mayor programa de prueba que muestra el correcto funcionamiento de los mutex es “ProdCons” del cual hablaremos en la siguiente sección”

## Aplicaciones de Userspace

### *linearGraph y parabolicGraph*

Estos dos módulos grafican funciones, tomando por stdin los valores correspondientes. Se pueden ingresar valores negativos. Al mostrar el gráfico, cicla durante algunos segundos así el usuario puede ver el resultado final. Ambos se ejecutan únicamente en foreground.

### *background*

Este módulo consiste en unas pocas líneas de código. Fue creado para probar los procesos background, aunque también puede ejecutarse en foreground. Imprime un mensaje por consola, pero solo lo veremos si lo corremos en foreground. Si lo corremos en background, nunca veremos el mensaje. **Al ejecutarlo en background, será necesario teclear enter para continuar usando la shell.**

### *processWrite y processRead*

Son dos módulos que se utilizaron para mostrar el correcto funcionamiento las *sysCallpipeCreate*, *pipeRead* y *pipeWrite*. A través de ellos, podemos comprobar que podemos realizar una conexión simple entre 2 procesos permitiendo múltiples escrituras y lecturas.

### *processWriteAndRead y processReadAndWrite*

Son dos módulos que se utilizaron para testear que un proceso podía mantener varios canales de comunicación entre procesos y además se los utilizó para comprobar el correcto funcionamiento del modo bloqueante de la función read. Para una muestra gráfica ver Imagen 1.

### *testMemoryManager*

Es un módulo que permite mostrar el correcto funcionamiento del memory manager implementado en el Sistema Operativo. Muestra los punteros que debería retornar el memory manager y permite ver que además no se desperdicia memoria en el heap.

### *Producer Consumer*

Son 2 módulos diferentes que se crearon con el fin de mostrar el correcto funcionamiento de los mutex. Producer crea el mutex, y se encarga de hacer X cantidad de signals. Consumer busca el mutex creado, y se encarga de hacer X cantidad de waits. El proceso "Consumer" será un proceso de foreground para mostrar el correcto funcionamiento de los mutex. En este módulo se puede ver el

correcto funcionamiento de los mutex ya que cuando “Consumer” hace un wait, se queda bloqueado hasta que “Producer” hace un signal (lo mismo con producer, se queda bloqueado hasta que el otro haga un wait). Este proceso se repite hasta que lo hagan tantas veces como una constante que está definida en ambos programas. Al termina, “Consumer” libera el mutex.

Importante: Para el buen funcionamiento del Producer Consumer, la única limitación que tiene es que deben ejecutarse primero los consumidores y luego los productores. Esto es debido al valor inicial que tienen los mutex.

Otra cosa importante a resaltar es que, en ningún momento el sistema queda colgado, pero si, por ejemplo, hacemos kill de algún producer/consumer no podemos asegurar que sigan funcionando de la forma que es debida.



## **Importante**

- El TP de la asignatura anterior contenía dos comandos en la shell para producir excepciones, manejarlas, imprimir el contenido de los registros en el momento en el cual está se produjo. Decidimos dejar esto en el trabajo, aunque no estarían funcionando como queremos, ya que la rutina de atención de la excepción llama a main, y por ende, se genera un nuevo proceso shell, lo cual no es correcto.
- Se agregaron a los Makefiles -pedantic -ansi -Wall para compilar tanto el Kernel como los módulos de Userland. Algunos warnings como los del ascii del teclado, ya existían desde el TP anterior, y no fuimos capaces de solucionarlos. Algunos otros quedaron por conflictos de punteros, y como estos flags fueron agregados en último momento, preferimos dejarlo así a arriesgarnos a cambiarlos y que algo comience a funcionar mal. Estos últimos conflictos sobre los punteros, es más que nada sumas aritméticas que se hacen sobre ellos para retornar otra posiciones de memoria

## **Conclusión**

Luego de implementar las diferentes partes del trabajo práctico y ver cómo están interconectadas entre sí, pudimos entender que ante un trabajo semejante lo más conveniente es estar al tanto de la teoría y de los diferentes algoritmos que existen para cada sección. También conviene realizar esquemas en papel lo más detallados posibles, para que a la hora de pasarlo a código se tengan en cuenta todas las conexiones que deben hacerse entre las diferentes partes del sistema y para cubrir y resolver todos los casos límite.

Por otro lado, nos hubiera gustado tener más tiempo para mejorar algunos algoritmos del TP, generar nuevos tests y refinar aquellos puntos donde sentimos que somos capaces de implementar un sistema operativo más útil. Son aspectos que intentaremos revisar y, si es posible, mejorar para la tercera entrega.