



Trabajo Práctico N° 3:
Estructuras de administración de
recursos

Grupo 2:

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

Introducción

El trabajo práctico consistió en realizar mejoras all TP anterior, donde habíamos implementado un kernel con procesos, IPCs, Memory Managment y Scheduler. Optamos por realizar kernel-threads, Buddy Allocator y pipes. Además, agregamos nuevas aplicaciones en Userspace.

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. Con esto será suficiente para compilar todos los módulos del mismo con los flags de Wall, ansi y pedantic.

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. Los módulos que se repiten del TP anterior no se detallarán, salvo que presenten alguna modificación considerable en su implementación.

Kernel-threads

Implementación:

Para la implementación de kernel-threads, en primer lugar se modificó la estructura de los procesos del sistema.

Se crearon dos nuevas estructuras `ThreadSlot` y `Thread`. La primera contiene la estructura del thread y un puntero al siguiente slot de hilo y la segunda contiene la estructura del hilo formada por el TID, el estado, el quantum, etc.

Además se cambió la estructura de los procesos que teníamos en el TP anterior, que contenía el stack y el starting point, moviendo estos punteros a la nueva estructura de threads. Por otro lado, en la estructura de los procesos se agregaron dos punteros a `ThreadSlot`, uno apuntando al primer slot y otro al actual slot y una variable entero que indica la cantidad actual de threads que tiene el proceso (ver Imagen 1 y 2).

Podemos ver también que se respetó la estructura que tienen los threads, ya que los mismos comparten su heap pero el stack que poseen es único para cada uno.

```
typedef struct Process{
    int PID;
    int fatherPID;
    int childsPID[MAX_CHILDs];
    int status;
    int foreground;
    char processName[MAX_PROCESS_NAME];
    void * pages[MAX_PAGES];
    int threadSize;
    ThreadSlot * threads;
    ThreadSlot * currentThread;
    p_heapPage heap;
    p_pipe pipes[MAX_PIPES];
    int pipeIndex;
    p_pipe stdin;
    p_pipe stdout;
}Process;

typedef struct ProcessSlot{
    struct ProcessSlot * next;
    Process process;
}ProcessSlot;
```

Imagen 1: Estructuras para el slot del proceso y el proceso.

```
typedef struct Thread{
    int TID;
    void * startingPoint;
    void * userStack;
    void * baseStack;
    int status;
    int threadQuantum;
}Thread;

typedef struct ThreadSlot{
    struct ThreadSlot * next;
    Thread thread;
}ThreadSlot;
```

Imagen 2: Estructuras para el slot del hilo y el hilo.

Al cambiar estas estructuras, ahora es necesario que cada vez que accedemos al `startingPoint` o al `userStack` para restablecer el contexto, acceder al thread actual del proceso.

Otra modificación que se realizó en la implementación fue en la función principal del scheduler que es `runScheduler()`. La misma era llamada luego de que se ejecutara la interrupción del timer tick y chequeaba si se había completado el quantum para cambiar de proceso. Lo que se realizó ahora fue que si el quantum del proceso todavía no se agotó, entonces que se chequee si el quantum del thread actual si se agotó mediante la función `checkIfThreadChange()`. Si el thread actual debe cambiar, entonces se llamará a `nextThread()`.

Para la creación de un nuevo proceso, lo que se agregó fue la creación de un slot para el hilo inicial mediante las funciones `createThreadSlot()` y `createThread()`.

Por último, tenemos dos funciones nuevas, las cuales son invocadas cuando se ejecuta una syscall correspondiente a threads, como `addThreadToProcess()` y `deleteThreadFromProcess()`. También tenemos otras dos funciones que son útiles para poder bloquear o desbloquear un thread.

La primera es llamada por la syscall `sys_threadCreate` y lo que hace es crear un nuevo slot para el hilo, agregandolo al array de threads en la última posición e incrementar la variable `threadSize` que pertenece al proceso.

La segunda es llamada por la syscall `sys_threadRemove` y lo que hace es eliminar el slot del hilo, haciendo un release de las páginas del mismo y acomodando los punteros para que el hilo anterior apunte al siguiente hilo del que voy a eliminar.

Funcionamiento:

Una vez que el timer tick interrumpe al procesador y se pasa a correr un proceso, accedemos al `startingPoint` que ahora está guardado en la estructura del hilo. Además, al cambiar de thread (o de proceso) el `esp` se guarda en `userStack` que ahora se encuentra también en dicha estructura.

Antes de pasar a correr el siguiente thread o proceso, accedemos a la función `runScheduler()`, que si todavía no se agotó el quantum del proceso, chequea si hay que cambiar el thread por el siguiente. Para esto comparamos el quantum del thread con el definido en el kernel mediante un `define`. Si ya es momento de cambiar de thread, guardamos el contexto en el stack, cambiamos el estado del thread y pasamos al siguiente thread.

Podemos crear un thread desde un programa en Userland a través de la función `thread()` enviándole como parámetro el puntero a función. Esto agrega el thread a correr al final del arreglo de hilos del proceso.

Podemos eliminar un thread desde un programa en Userland a través de la función `exitThread()`, que acomodará los punteros del arreglo de tal forma que ya no se tenga en cuenta el thread a eliminar a la hora de cambiar de hilo. Además se hace un `release`, así se libera la memoria que fue reservada para el stack del mismo.

Por último, podemos bloquear un thread si hacemos un `waitThread()`. El scheduler no tendrá en cuenta este thread a la hora de cambiar de hilo hasta que el último thread sea eliminado y en ese caso se desbloqueará. Este funcionamiento está restringido al caso de que el hilo bloqueado sea el main.

System calls:

Nombre	Número	Acción
<code>sys_threadCreate</code>	28	Crea un nuevo thread en el proceso actual
<code>sys_threadRemove</code>	29	Elimina el thread actual
<code>sys_threadWait</code>	30	Bloquea al proceso actual
<code>sys_threadCount</code>	36	Retorna la cantidad de threads del proceso actual

Limitaciones:

- **ThreadWait:** Aunque existe una syscall para que el thread actual espere a los threads que lanzó, la misma no funciona como se esperaría. Lo que hacemos en

esta syscall es bloquear al thread llamador, por lo tanto el scheduler solo correrá el resto de los threads, hasta que ya no queden más, y el thread llamador se vuelva a desbloquear. El problema existente es que una vez que el thread llamador hace la llamada al sistema, se hace un return, volviendo al código del hilo. Por lo tanto hasta que no se produzca una interrupción, se seguirá ejecutando este thread. Entonces es necesario hacer un wait de unos segundos tras esta interrupción para que el timer tick vuelva a interrumpir y se pase a correr otro hilo. También hubiera sido deseable poder especificar por parámetro que hilo quiero esperar, pero por falta de tiempo no lo realizamos, ya que deberíamos guardar en el thread llamador el hilo o los hilos por los cuales quedará bloqueado.

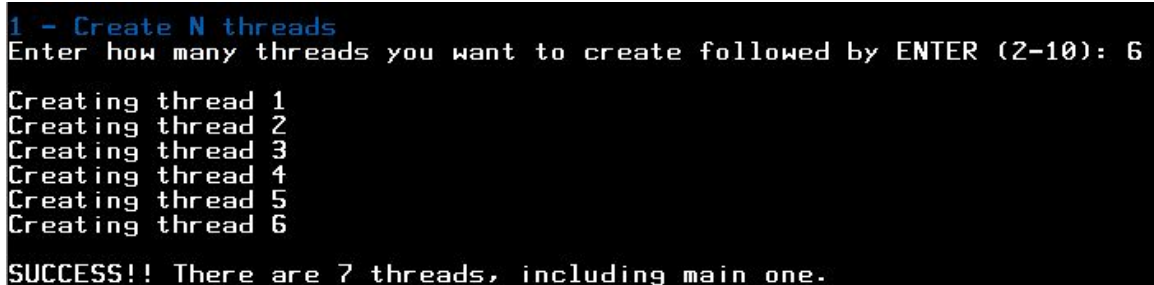
- **Parámetros:** Actualmente, una limitación que presentan los threads es que al crearlos solo podemos hacerlo sin pasaje de parámetros a los mismos. Esto puede limitar la implementación del alcance de algunas funcionalidades de userspace. No pudimos realizarlo por falta de tiempo. Preferimos utilizar el tiempo para realizar casos de prueba.

Casos de prueba:

```
$>: ./threadTest&
```

Realizamos un test que consiste en cuatro partes:

1. La primera parte consiste en crear tantos threads como el usuario ingrese. Una vez creados, mediante la syscall threadCount, se obtiene cuantos threads contiene el thread llamador. Si el valor que retorna la llamada al sistema coincide con el valor ingresado por el usuario, entonces la creación de threads está funcionando de la forma correcta (ver Imagen 4).



```
1 - Create N threads
Enter how many threads you want to create followed by ENTER (2-10): 6
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
SUCCESS!! There are 7 threads, including main one.
```

Imagen 4: Primera parte del test donde creamos hilos.

2. La segunda parte consiste en eliminar los threads creados anteriormente. Si luego de eliminarlos, solo queda un thread (que sería el thread llamador) entonces la eliminación de threads está funcionando de la forma correcta.
3. La tercera parte consiste en verificar que el heap es compartido por los threads. Para ello lo primero que hacemos es imprimir el valor donde empieza el heap de cada uno

usando una syscall, guardando el valor en la variable heapSP. Podemos ver que si es la misma dirección de memoria, es porque los dos utilizan el mismo heap.

Para hacer esto, previamente hicimos un `malloc(sizeof(int))` ya que a los procesos se le asigna un heap una vez que piden memoria por primera vez. Entonces luego de esto, nuestro próximo bloque de memoria a otorgar empezará en la dirección `heapSP + 4`, algo que también vamos a verificar.

Además, realizamos un `malloc(sizeof(int))` dentro del thread creado e imprimimos su valor. Luego, hacemos lo mismo en el thread llamador (es decir el main). Podemos ver que luego del primer malloc, el puntero apuntará a `heapSP + 4` y en el segundo malloc a `heapSP + 8`. Es decir que una vez que uno de los threads pidió memoria, cuando cualquiera del resto de los threads vuelva a pedir, se le otorgará desde donde pidió el último thread ya que los mismos comparten su heap.

```
3 - Two threads (or more) share their heap
To continue press 1 followed by ENTER: 1

I make the first malloc of the program in order to get given a heap. It size is of sizeof(int) = 4...
Now I'll print heap starting point in two different threads...
- Main thread: Heap Starting Point: heapSP = 29000
Creating thread 1...
- Thread: Heap Starting Point: heapSP = 29000

Heap starting point is the same in both threads. SAME PROCESS THREADS ---> SAME HEAP

- Thread: Now I make a malloc inside the thread... Remember that I made a malloc of size sizeof(int) before so the pointer will value heapSP + 4
The result is: threadPointer = 29004

- Main thread: Now I make a malloc on main thread of size sizeof(int).. I recently made malloc inside the thread so if the heap is shared, my pointer will value threadPointer + 4
The result is: mainPointer = 29008
```

Imagen 5: Tercera parte del test donde verificamos que el heap se comparte.

4. La cuarta parte consiste en verificar el orden en el cual se ejecutan los threads, es decir, una vez que creamos un thread, este se coloca a final de la cola de threads. Se predice el orden en el cual se deberían ejecutar, y después se debe verificar (a ojo del usuario) si se cumplió la predicción o no.

```
$>: ./multithread&
```

Este programa tiene tres threads (más el thread llamador) que imprimen el valor de una variable hasta un cierto tope determinado en el bucle for. La idea de esto es mostrar principalmente dos cosas:

- En primer lugar que cuando un thread retoma su ejecución, lo hace restaurando su contexto de forma correcta. Es decir, si el thread 1 se interrumpió luego de imprimir el valor 25, cuando retome su ejecución deberá imprimir el valor 26 (ver Imagen 6).
- En segundo lugar, vemos que el orden de los threads es el correcto ya que si se creó primero el thread 1, luego el 2 y por último el 3, deberán ejecutarse siempre en ese orden. Además, cada bucle for tiene un tope distinto, por lo cual es probable que el thread 2, por ejemplo, termine antes que el thread 3. De esta forma demostramos

que si por ejemplo, el thread 2 es eliminado porque terminó, entonces una vez que el thread 1 sea interrumpido, se pasará a correr el thread 3.

El main thread (thread llamador) no corre porque queda bloqueado esperando a que los threads 1, 2 y 3 terminen de ejecutarse.

```
To start please press 1 followed by ENTER: 1
Creating thread 1...
Creating thread 2...
Creating thread 3...
--- MAIN THREAD STATUS: Locked. Main thread will wait until all threads finished
THREAD 1 - a: 0
THREAD 1 - a: 1
THREAD 1 - a: 2
THREAD 1 - a: 3
THREAD 1 - a: 4
THREAD 1 - a: 5
THREAD 1 - a: 6
THREAD 1 - a: 7
THREAD 1 - a: 8
THREAD 1 - a: 9
THREAD 1 - a: 10
THREAD 1 - a: 11
THREAD 1 - a: 12
THREAD 1 - a: 13
THREAD 1 - a: 14
THREAD 1 - a: 15
THREAD 1 - a: 16
THREAD 1 - a: 17
THREAD 1 - a: 18
THREAD 1 - a: 19
THREAD 1 - a: 20
THREAD 1 - a: 21
THREAD 1 - a: 22
THREAD 1 - a: 23
THREAD 1 - a: 24
THREAD 1 - a: 25
THREAD 2 - b: 0
THREAD 2 - b: 1
THREAD 2 - b: 2
THREAD 2 - b: 2
THREAD 2 - b: 3
THREAD 2 - b: 4
THREAD 3 - c: 0
THREAD 3 - c: 1
THREAD 3 - c: 2
THREAD 3 - c: 3
THREAD 1 - a: 26
THREAD 1 - a: 27
THREAD 1 - a: 28
THREAD 1 - a: 29
THREAD 1 - a: 30
THREAD 2 - b: 5
```

Imagen 6: Ejecución del programa multithread.

```
$>: ./bubbleSort&
```

La idea de este programa es mostrar algún caso donde la utilización de threads tenga una importancia relevante. Como nuestro sistema carece de algunas estructuras que podrían servirnos para armar un programa potente y además nuestros threads no reciben parámetros, se nos complicó pensar una idea donde el uso de los hilos se pudiera lucir.

Se nos ocurrió crear un programa donde tengamos un array desordenado y tengamos que ordenarlo. En este caso lo que hacemos es crear dos threads y cada uno ordena una mitad del arreglo mediante Bubble Sort. Claramente, como el ordenamiento no es inmediato, podemos ver por pantalla que la ejecución de ambos se va intercalando (ver Imagen 7).

Quizá en este caso, la eficiencia no es notoria a si realizamos Bubble Sort con un único thread en la totalidad del array. Pero podríamos extender esta implementación a más

threads ordenando el array de a partes más pequeñas, asemejando esto a un Merge Sort donde en este caso la eficiencia sí sería notoria.

```

--- BUBBLE SORT WITH THREADS ---
We have an array of 20 numbers ... Let's sort it through two threads
We have this array...
Array={79,41,122,19,142,159,162,174,189,198,60,90,61,77,100,1,106,14,16,170}
To start please press 1 followed by ENTER:
Creating thread 1...
Creating thread 2...
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array
Ordering half 0 of the array

```

Imagen 7: Ejecución del programa bubbleSort.

Buddy Allocator

Implementación y Funcionamiento:

El funcionamiento e implementación del Buddy Allocator, no sufrió grandes modificaciones con respecto al TP anterior. De todos modos, sí cabe destacar que eliminamos limitaciones que tenía el buddy en cuanto a ineficacia espacial y temporal ya que ahora podemos decir que funciona lo más rápido y con el menor espacio posible tal y como lo hace un Buddy Allocator real.

La clave para solucionar dichos problemas de ineficacia fue agregar un estado a los posibles estados que tiene un nodo. En el TP 2 un nodo del árbol del Buddy allocator solo podía tener 2 estados posibles (libre u ocupado) lo cual dificultaba la tarea de hacer el release de los nodos de manera recursiva.

Al agregar un nuevo estado (parcialmente ocupado) solucionó el problema de tener que revisar nodos de más.

Syscalls:

Nombre	Número	Acción
sys_malloc	10	Reserva la cantidad de memoria pedida
sys_printMemoryTree	26	Imprime todo el árbol del Buddy Allocator para muestra y debuggeo
sys_printMemoryVertical	27	Imprime una visión simplificada del estado de la

		memoria mostrando si está ocupada o no y que proceso la ocupa
--	--	---

Por otro lado, como ya explicamos en el TP anterior, al finalizar un proceso (ya sea por propia voluntad o por el uso de kill) se liberan todas las estructuras utilizadas por dicho proceso (slot, heap, thread, pipe, mutex, etc).

Limitaciones:

La única limitación del Buddy Allocator que vale la pena mencionar, es que como los módulos de Userland se compilan en posiciones altas de memoria, no pudimos reservar dichas posiciones de memoria.

Casos de prueba:

`$>: ./TestMemoryManager&`

Es uno de los módulos heredados del TP anterior. La explicación del funcionamiento de dicho test se encuentra en el informe anterior, ya que no sufrió alteraciones en este TP 3.

Por otro lado como comentamos en la sección de syscalls, creamos 2 comandos para poder visualizar el funcionamiento interno y detallado del buddy allocator y el estado de la memoria.

`$>: buddyInfo`

Comando que sirve para imprimir el árbol completo de nodos de el Buddy Allocator, mostrando todas sus particiones, su estado, sus hijos y a que zona de la memoria apuntan. Un comando realmente útil para demostrar el funcionamiento interno del buddy y a la vez debuggear problemas de administración de la memoria (ver Imagen 8).

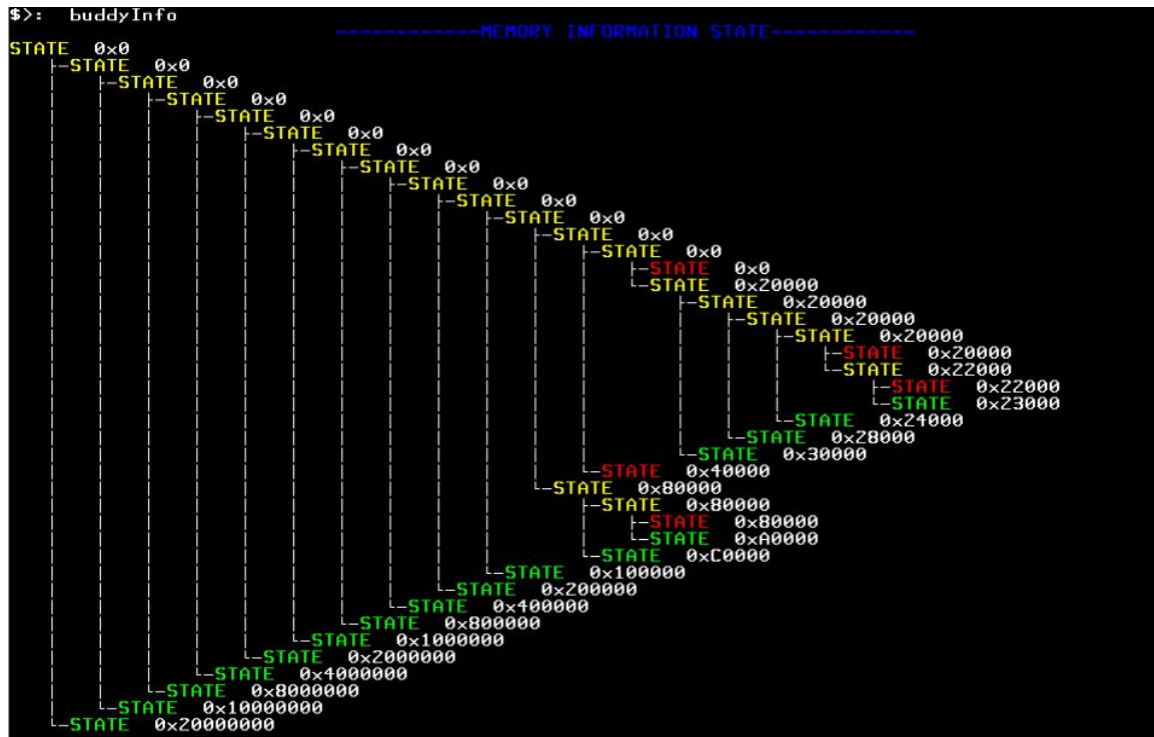


Imagen 8: Árbol de nodos del Buddy Allocator.

`$>: memInfo`

Comando que sirve para imprimir el estado actual de la memoria de una manera mucho más amigable para el usuario. Un comando realmente útil para ver de manera simplificada que zonas de la memoria están ocupadas, por qué proceso y más aún, por qué sección del proceso (heap/stack/thread).



Imagen 9: Estado actual de la memoria cuando no hay ningún proceso corriendo salvo shell.

Pipes

Implementación y Funcionamiento:

Al igual que el Buddy Allocator, pipes no sufrió grandes modificaciones en su implementación y funcionamiento para el TP 3. Pero sí cabe aclarar que en el anterior TP, pipes tenía la limitación de que no tenía una conexión biunívoca entre 2 procesos. Esto se vio solucionado en este TP al implementar la creación de procesos como una estructura de árbol donde cada proceso puede tener hijos de manera tal que cada hijo puede conocer el PID de su padre y de la misma manera un padre puede conocer el PID de sus hijos. Por otro lado, en el anterior TP no podían “pipearse” 2 procesos por la terminal. Este fue uno de los grandes avances que tuvo pipes en el desarrollo de este TP al permitirse dicha acción.

Syscalls:

Nombre	Número	Acción
sys_pipeCreate	15	Crea un pipe
sys_pipeRead	17	Lee a través del pipe, si no hay para leer bloquea el proceso
sys_pipeWrite	16	Escribe a través del pipe, si el pipe está lleno bloquea el proceso
sys_pipeClose	18	Cierra la posibilidad de escritura/lectura del proceso que la invoca
sys_pipeOpen	19	Abre la posibilidad de escritura/lectura del proceso que la invoca
sys_pipeStdoutStdin	37	Pipea el stdout de un proceso al stdin de otro proceso
sys_getPipeBuffer	38	Devuelve el buffer del pipe especificado (para muestreo)

Casos de prueba:

Antes de detallar los casos de prueba, cabe aclarar que cada vez que se utiliza la syscall sys_pipeCreate, el kernel envía un mensaje mostrando la información relevante del pipe que se está creando, su PID, los procesos involucrados y si ambos están conectados o no.

Estos mensajes aparecerán en consola, independientemente si el proceso ejecutado es de foreground o background.

Dado que ambos procesos hacen un pipe para crear la conexión, el kernel enviará 2 mensajes del pipe, en el primero el pipe se lo verá con el estado de "Free" ya que falta que uno de los procesos se conecte al pipe y el segundo mensaje se lo verá con el estado de "Full" mostrando que ambos procesos están conectados al pipe (ver Imagen 10 y 11).

```
*****Kernel message*****
Pipe PID: 1
Process connected:      PID 1: 2      PID 2: 3
STATUS: FREE
*****End kernel message*****
```

Imagen 10: Estado del pipe cuando falta un proceso para terminar la conexión 1:1.

```
*****Kernel message*****
Pipe PID: 1
Process connected:      PID 1: 2      PID 2: 3
STATUS: FULL
*****End kernel message*****
```

Imagen 11: Estado del pipe cuando ya se conectaron dos procesos.

\$>: ./processWriteAndRead&

Es uno de los módulos heredados del TP anterior, que demuestra que un proceso se puede conectar a múltiples procesos a través de la utilización de pipes. La única diferencia con el TP anterior es que ahora processWriteAndRead además ejecuta al proceso processReadAndWrite como su hijo para crear esa conexión biunívoca que mencionamos anteriormente.

\$>: ./processRead&

Es uno de los módulos heredados del TP anterior, que demuestra que un pipe puede realizar múltiples escrituras y lecturas de un mismo pipe. Al igual que el módulo anterior, este ejecuta a processWrite como su hijo para crear la conexión biunívoca.

\$>: ./proA&

Este es un nuevo módulo que pretende demostrar una combinación de lecturas y escrituras múltiples junto con conexión entre múltiples procesos (proA, proB y proC).

Para hacerlo más interactivo le permitimos al usuario elegir la manera en que se establece dicha conexión de procesos.

La idea de este caso de prueba es tener 2 palabras en proA (hola y chau) y concatenarlas con cada palabra guardada en los demás procesos.

El programa nos pedirá elegir si queremos generar una conexión:

1. proA -> proB -> proA -> proC -> proA.
2. proA -> proC -> proA -> proB -> proA.

Dependiendo la opción elegida el mensaje final quedará concatenado como:

“holaholasadioschau” o “holaadiosholaschau”

La idea es enviar la palabra “hola” del proA a proB o proC a través de pipes; una vez ahí la palabra se concatenará a la palabra almacenada en proB o proC y será retornada a proA a través del mismo pipe.

Luego esa palabra será enviada a proB o proC dependiendo cual se haya elegido antes a través de pipes; una vez ahí la palabra se concatenará a la palabra almacenada en proB o proC y será retornada a proA a través del mismo pipe.

Finalmente a la palabra retornada se la concatenará con la palabra “chau” y se la imprimirá por salida estándar para mostrar su correcto funcionamiento.

```
$>: ./producer|||./consumer
```

Este es un módulo heredado del TP anterior, pero que fue fuertemente modificado para mostrar la solución del productor-consumidor para 1 productor y 1 consumidor.

Como vemos en el comando, la salida estándar del proceso “producer” está conectada a la entrada estándar del proceso “consumer”, de manera tal que cada vez que “producer” hace un putchar/printf en realidad está escribiendo a pipe que será leído por “consumer” a través de getchar.

Este comando muestra no sólo la solución correcta al problema al productor-consumidor sino que también, muestra el correcto funcionamiento del bloqueo de la función read y write de pipes al bloquear el proceso cuando el buffer está lleno (si se hace un write) o al leer del buffer vacío (si se hace un read) (ver Imagen 12).

```
Produced! Buffer: [ffff]. Lenght: 3
Produced! Buffer: [ffffff]. Lenght: 6
Produced! Buffer: [ffffffff]. Lenght: 9
Produced! Buffer: [ffffffffffff]. Lenght: 12
Produced! Buffer: [fffffffffffffff]. Lenght: 15
Produced! Buffer: [fffffffffffffffffff]. Lenght: 18
Produced! Buffer: [fffffffffffffffffffff]. Lenght: 21
Produced! Buffer: [ffffffffffffffffffffff]. Lenght: 24
Produced! Buffer: [fffffffffffffffffffffff]. Lenght: 27
Produced! Buffer: [fffffffffffffffffffffff]. Lenght: 30
Produced! Buffer: [fffffffffffffffffffffff]. Lenght: 33
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 32
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 31
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 30
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 29
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 28
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 27
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 26
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 25
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 24
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 23
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 22
Produced! Buffer: [fffffffffffffffffffffff]. Lenght: 25
Produced! Buffer: [fffffffffffffffffffffff]. Lenght: 28
Produced! Buffer: [fffffffffffffffffffffff]. Lenght: 31
Produced! Buffer: [fffffffffffffffffffffff]. Lenght: 34
Produced! Buffer: [fffffffffffffffffffffff]. Lenght: 37
Produced! Buffer: [fffffffffffffffffffffff]. Lenght: 40
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 39
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 38
Consumed! Buffer: [fffffffffffffffffffffff]. Lenght: 37
```

Imagen 12: Ejecución del producer-consumer.

Conclusión

Al ser este un TP que venimos heredando de la asignatura anterior, Arquitectura de Computadoras (72.08), nos hubiera gustado finalizar con el mismo sin ninguna limitación. Esto no fue posible por falta de tiempo, pero aun así estamos conformes con el trabajo realizado a lo largo del TP 2 y TP 3, ya que pensamos que salvo kernel-threads, el trabajo tiene muy pocas cosas que se deberían pulir.

Más allá de eso, ambos trabajos estuvieron buenos para reforzar los conceptos vistos en la teoría, ya que al intentar crear tus propias estructuras, uno se interioriza mucho más con la teoría.

También nos gustó el hecho que para este TP, tengamos la libertad de elegir qué estructuras crear, ya que muchas veces ciertos temas se nos hacen más fáciles e intuitivos que otros.