

**Universidad de Costa Rica**

Facultad de Ingeniería

Escuela de Ciencias de la Computación e Informática

**Proyecto integrador de sistemas operativos y redes de comunicación**

**CI - 0121**

Segundo avance Proyecto Integrador

**Profesores**:

Francisco Arroyo

José Antonio Brenes

**Estudiantes**:

Gustavo Jiménez Hernández **B84060**

Javier Sandoval Zamora **B56762**

Isaac Zúñiga Serrano **B68038**

19 de junio, 2020

**1. Objetivos**

Realizar las siguientes tareas asignadas dentro del sistema operativo NachOS:

**Correr:**

* Fork
* Exec
* Join
* Yield
* Exit

**Socket:**

* Constructor
* Close
* Connect
* Bind
* Listen
* Accept

**Loader:**

* Paginación de memoria
* Carga de programas a memoria

**Threads:**

* Semáforo
* Lock
* Barrier
* Mutex
* Variable de condición

**Cliente NachOS:**

* Socket cliente

**2. Requisitos**

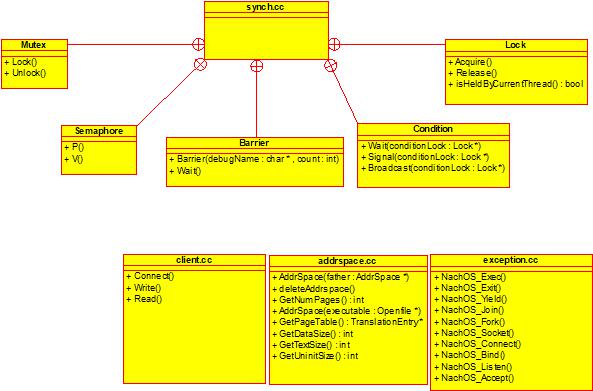
* Sistema operativo Linux
* El código brindado actualizado

**3. Restricciones**

Solo funcionan los programas de prueba que incluyen a nuestras tareas del avance, otros programas de prueba que involucren otros “System Call” u otras características del sistema operativo que involucren tareas diferentes a las realizadas en este avance, pueden no funcionar correctamente.

**4. Esquema**

El siguiente esquema muestra en cuáles archivos y en cuáles funciones específicas se realizaron las tareas mencionadas:



**5. Descripción de la solución:**

Fork: Se crea un nuevo hilo y un AddrSpace con el constructor para hilos, que toma de argumento el addrSpace del padre, para poder compartir los datos y los datos globales. Al nuevo hilo se le asigna este nuevo addrSpace y luego se le hace Fork, para que ejecute la función NachosForkThread y se le manda de argumento el registro 4 que tiene un puntero a la dirección de memoria de la función que se quiere que ejecute el hilo. En la funcion NachosForkThread, el nuevo hilo inicializa sus registros y además configura su registro PCReg para que sea igual a la dirección de memoria que le llega del registro 4, así el hilo sabe donde tiene que empezar a correr.

Exec: Se lee del registro 4 la dirección de memoria de NachOS de donde esta el puntero al nombre del programa que se quiere ejecutar con el nuevo proceso. Después de copiar en el espacio de memoria del sistema el nombre del archivo a abrir, se crea un OpenFile con el nombre del archivo, luego se crea un AddrSpace con el constructor para procesos, que aloca memoria para los datos, el texto y la pila, se crea un nuevo Thread al que se le asigna este nuevo espacio de memoria y se llama a la función Fork de Thread para correr la función NachosExecThread que inicializa los registros del thread y manda a correr al hilo con el programa que solicito el usuario.

Join: Se lee del registro 4, el id que le corresponde al thread que se desea esperar y de igual manera se obtiene el id del thread en ejecución del currentThread, se crea una instancia de la struct “father\_son\_t”, la cual posee dos integers correspondientes al id de los threads que van a participar en el Join, es decir, padre espera hijo, aunque en el fondo pueden ser procesos aparte y no necesariamente padre e hijo, es decir, para esperar un thread no necesita que le haya hecho fork, pero por simplicidad se llamó la struct de esa manera, puesto que de todas formas el exec es ejecutado por el thread que espera, aunque el nuevo thread no herede el pageTable del que ejecuta el exec, como ocurre en el fork. Posteriormente se ingresa la instancia de “father\_son\_t” a una lista denominada “waiting\_list” y se agrega un semáforo a una lista de semáforos, llamada “process\_threads”, ya que es tanto para procesos, como para hilos, el semáforo agregado posee de nombre el mismo id del thread que ejecuta el Join, puesto que va a ser su semáforo, la idea de agregar la struct “father\_son\_t” y el “semaforo” es que el thread decrementa el mismo su semáforo, para realizar un wait y quedarse dormido, hasta que el thread que lo espera, al finalizar, es decir, cuando ejecuta el exit, le haga signal.

Yield: llama a la subrutina Yield, que ya se encuentra programada en la clase Thread, la cual se encarga de encolar el currentThread y mandarlo a dormir, sacar el thread siguiente que espera por procesador y ponerlo a correr.

Exit: Se saca el primer nodo correspondiente a la lista “waiting\_list”, y se llama a una subrutina con el nombre de “is\_someone\_waiting\_for\_me”, la cual busca a partir del primer nodo de la lista si encuentra su id en alguna de las instancias de la struct “father\_son\_t”, de encontrar su id como hijo de alguien, significa que existe un thread padre que espera por él y se retorna dicho id, para posteriormente buscar dicho id en la lista de semáforos, y poder realizarle signal al thread que espera en el join. Posteriormente se elimina el addrspace que le correspondía a dicho thread, para esto se toma en cuenta si ese addrspace es propio o heredado, de ser heredado solo se limpia la memoria correspondiente al stack, y de ser propio se limpia completamente toda la memoria física que le indica su pageTable, por último se llama la subrutina Finish, del currentThread.

Socket Constructor: Se leen los registros 4 y 5 para extraer la información de dominio y tipo del socket, respectivamente, luego se llama al System call “socket” de linux con esos parámetros, devuelve -1 si hubo algún error, si no, devuelve el identificador del socket creado.

Socket Close: Llama al “close” de linux para cerrar el socket con el identificador leído del registro 4.

Socket Connect: Se lee del registro 4 el identificador del socket, del registro 5, la dirección ip del servidor al cual se conectará, y del registro 6, se lee el puerto por el cual se conectará al servidor. Con esta información se llama al “connect” de linux, el cual devuelve un 0 si se conectó correctamente o -1 si hubo un error.

Socket Bind: Se lee el identificador del socket del registro 4, y del registro 5, el puerto, con esta información se hace el llamado a “bind” de linux, que devuelve -1 si hubo un problema o un 0 si fue exitoso.

Socket Listen: Se lee del registro 4 el identificador del socket, del registro 5, cuántas conexiones va a aceptar el servidor. Con esto, se llama a “listen” de linux que devuelve -1 si hubo un error o 0 si fue exitoso.

Socket Accept: Se lee el identificador del socket que solicita conexion, con esto se llama al “accept” de linux que devuelve -1 si hubo un error, o un 0 si fue exitoso.

Paginación de memoria: La paginación de memoria se maneja en el los constructores de AddrSpace, se puede hacer de dos maneras, para crear un proceso o un hilo. Estos difieren de cuanto campo requieren y que datos comparten. Para un proceso, el constructor recibe un OpenFile que es el ejecutable a cargar a memoria. De este OpenFile se extrae el tamaño de los datos sin inicializar, el tamaño del código y el tamaño de los datos inicializados. Luego se procede a calcular el número de páginas requeridas para el programa, que sería, la suma de los datos inicializados y no inicializados, el código y la pila. Luego se crea un pageTable, que es un arreglo de TranslationEntry del tamaño del número de páginas requeridas. Luego se usa el BitMap del sistema para buscar páginas disponibles y reservar el espacio. El constructor para hilos que recibe otro AddrSpace en vez de un OpenFile, hace lo mismo pero solo reserva espacio para su pila y comparte las demas paginas con su proceso padre.

Carga de programas a memoria: En el constructor de AddrSpace para crear procesos, se cargan los programas a memoria, además del código se carga los datos inicializados. Usando el método ReatAt del OpenFile, se va cargando página a página los datos del Open File a memoria.

Semáforo: El funcionamiento de el semáforo está dictado por un int, que va a ser el que controla toda la estructura. Si este valor está en 0, los hilos que traten de entrar a la variable compartida, van a esperar hasta que ese recurso se libere, cuando el hilo que estaba ocupando el recurso termina de hacer sus tareas, suma 1 a la variable y desencola un hilo de la cola de espera para que acceda a ese recurso, y así sucesivamente con los demas hilos.

Lock: El lock puede tener dos estados Ocupado o Libre, y tiene dos operaciones Adquirir y Liberar. La característica particular del semáforo es que solo puede ser liberado por el hilo que lo adquirirlo, para esto es necesario guardar el identificador del hilo que lo adquirió, para a la hora que un hilo le haga Liberar se pueda hacer la revisión. A nivel interno usa un semáforo para implementar sus dos operadores. Siendo Adquirir similar a P del semáforo y Liberar a V.

Barrier: El constructor recibe un nombre para fines de corrección de errores, y el número total de hilos que se van a tener, además de inicializar “sem” en 0, que será un semáforo privado de esta clase. Cada vez que un hilo llama al “Barrier.Wait()”, suma 1 a la variable initialValue, que va a ser un contador de los hilos que van llegando a ese punto. Si llega un hilo y no es el último en llegar, se llama a “sem->P()”, que pone a ese hilo en estado “dormido” hasta que ese mismo semáforo lo libere. En el caso de que llegue el último hilo a ese punto (el que faltaba para que todos avancen), entonces va a hacer n-1(n siendo el total de hilos) “sem->V()” que hará que los procesos que estaban en la cola de “dormidos”, se despierten de nuevo mediante este ciclo.

Mutex: Posee un semáforo, el cual solo se le permite poseer valores de 0 ó 1, ya que esa es la funcionalidad de un mutex.

Variable de condición: Posee un semáforo interno que se inicializa en “0”, además de que le entra de parámetro un Lock, la idea es que cada vez que se ejecuta wait, se libere dicho Lock, pero el thread se vaya a dormir, y para esto utiliza el semáforo, el cual no podrá seguir ejecutándose hasta que alguien le haga signal a dicha variable de condición, una vez salga del semáforo vuelve a adquirir el Lock.Posee las funciones, wait, signal, broadcast y down.

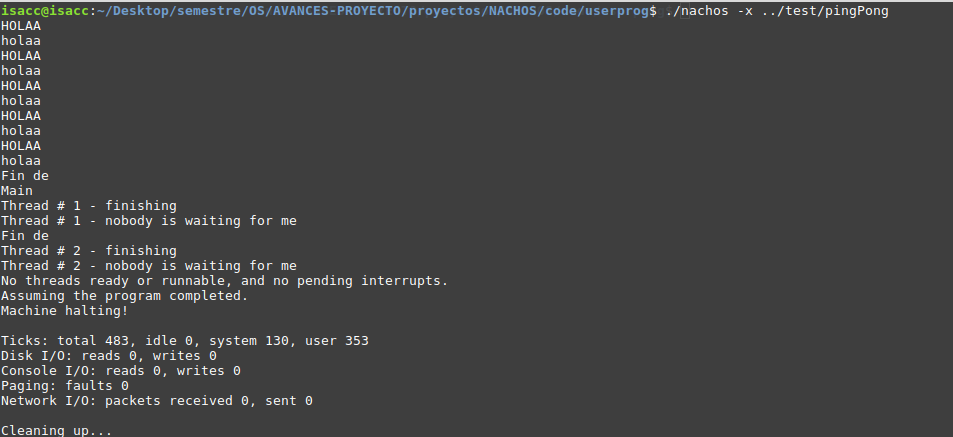
Socket cliente: El cliente de NachOS usa los llamados al sistemas descritos y desarrollados anteriormente. Primero se crea un socket, luego se conecta con un servidor y le hace una solicitud GET con el Write de NachOS. Finalmente hace un Read, para leer la respuesta del servidor e imprimirla en la consola.

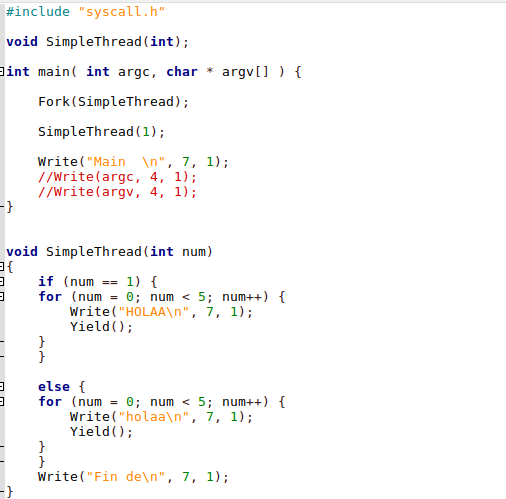
**6. Casos de prueba**

En los casos de prueba que se enseñarán a continuación, en muchos de ellos se prueban varios system call al mismo tiempo, más de igual manera se mostrarán varios ejemplos para ser más específicos.

**Fork, Yield y Exit**

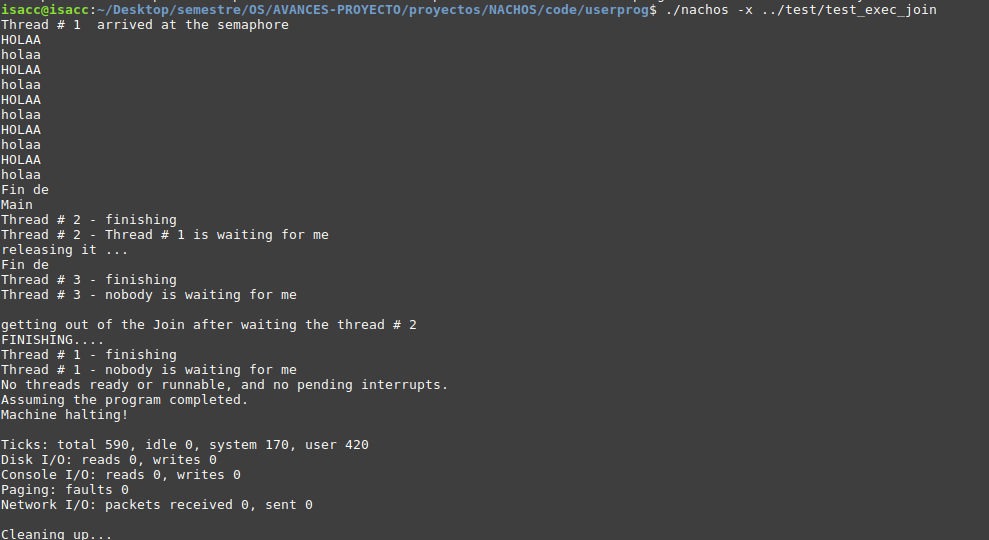
Para probar los system call Fork, Yield y Exit, se puede correr el programa pingPong, ubicado en la carpeta “test” de nachOS, tal y como se observa en la siguiente imagen:





En esta prueba se puede observar que se utilizan directamente los llamados a los system call “Yield” y “Fork”, no obstante, cada vez que se finaliza un programa se llama al system call “Exit” y por eso se dice que también se está probando dicho system call, ya que lo ejecuta sin ningún problema.

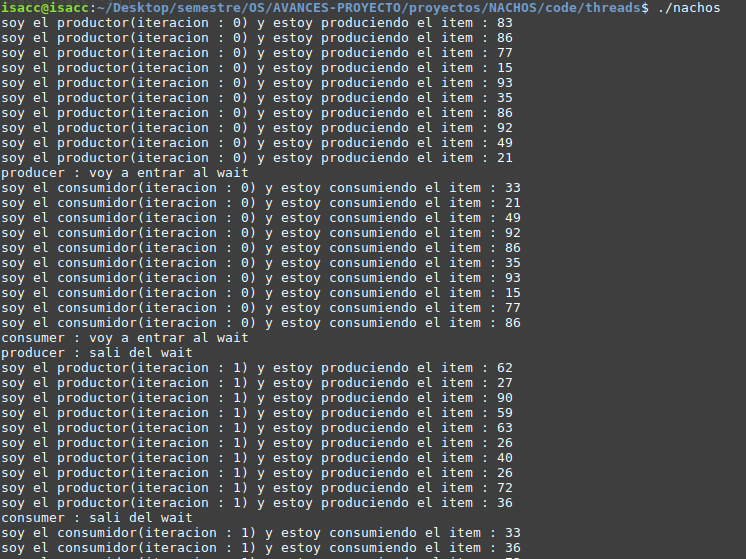
Para probar más específicamente los system call de “Exec” y “Join” se diseñó un programa de prueba llamado “test\_exec\_join”



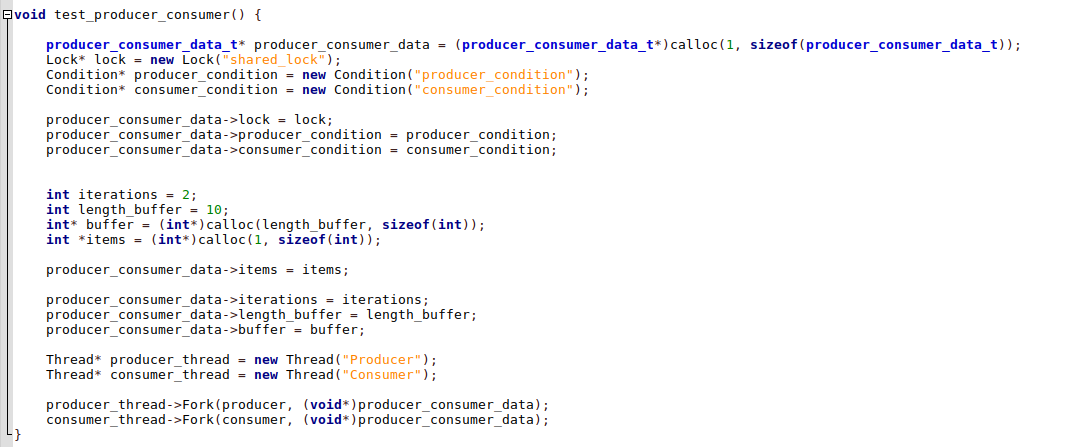
En este test, se ejecuta Exec llamando a correr al pingPong del que ya se mencionó anteriormente, la única diferencia es que al inicio se puede ver como el thread principal que ejecuta el “Exec” se queda en un semáforo producto del Join, y posteriormente en otro thread, se corre pingPong, que como se había mencionado anteriormente el mismo se encarga de hacer fork, y por eso al final quedan 3 threads que participan en la prueba, lo importante es observar, cómo el thread 1, es el primero y al final es el último en imprimir diciendo un mensaje de “getting out of the Join after waiting the thread #2” , esto quiere decir que el “Exec” funciono correctamente, pero que también el “Join” funcionó correctamente y que al mismo tiempo el “Exit” del thread 2, funcionó correctamente, ya que antes de finalizar liberar el semáforo del therad 1, ya que espero a que el thread creado finalizara para poder continuar.

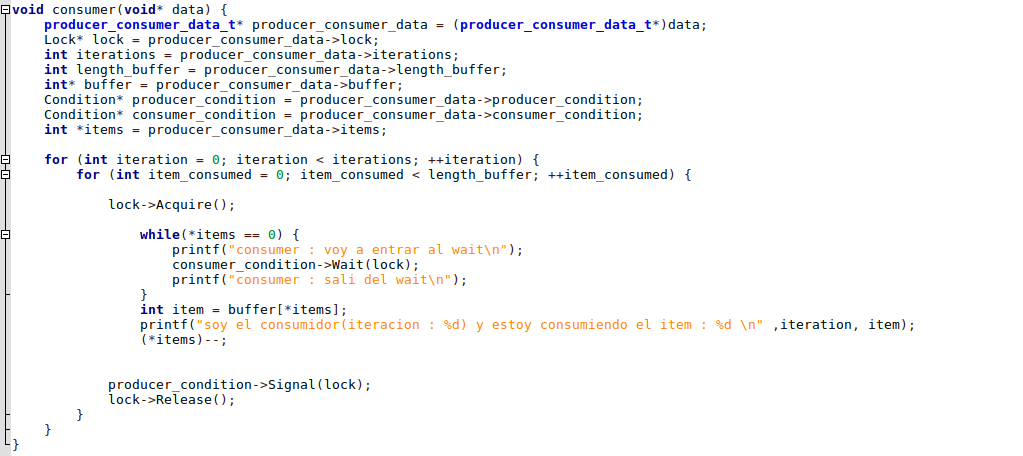


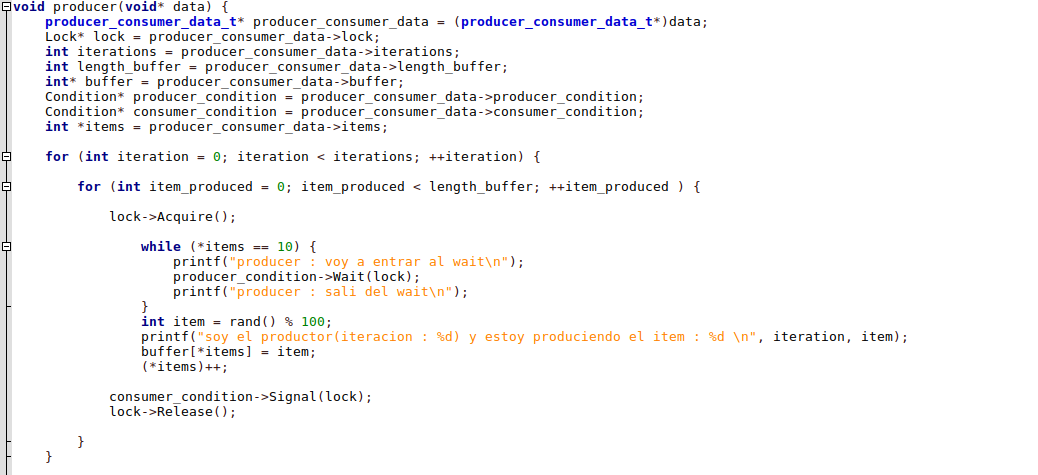
Para probar el system call correspondiente a la variable de condición en el archivo de “threadtest” ubicado en la carpeta de “thread”, se corre una prueba del productor-consumidor, en ese mismo archivo también existen otras pruebas adicionales, solo que hay que llamarlas desde el main, ya que las subrutinas se encuentran ahí mismo.



Para realizar esta prueba, se realizan varias iteraciones de productor-consumidor, y se tiene en cuenta que existe un buffer máximo para los items producidos, dado que nachOS es un sistema operativo que solo cuenta con un núcleo, no puede ejecutar el paralelismo como tal, más se puede ver como la concurrencia funciona correctamente, una vez el productor llena el buffer y ya no puede producir, el consumidor comienza a consumir y así sucesivamente, esto se puede realizar las “n” iteraciones deseadas.







Esta prueba de productor, consumidor se realizó directamente desde la memoria del kernel.

**Mecanismos de Sincronización**

Los mecanismos de sincronización se prueban usando el ejecutable nachos ubicado en la carpeta threads. Para las pruebas se crearon programas fuera de threadtest que es el programa que ejecuta el nachos en la carpeta threads. De la función ThreadTest se llaman a los programas externos que se realizaron. En esta función se puede comentar o descomentar el llamado a las funciones para realizar la prueba que se desea.

**Lock**

Nombre de la función para prueba: lockTest

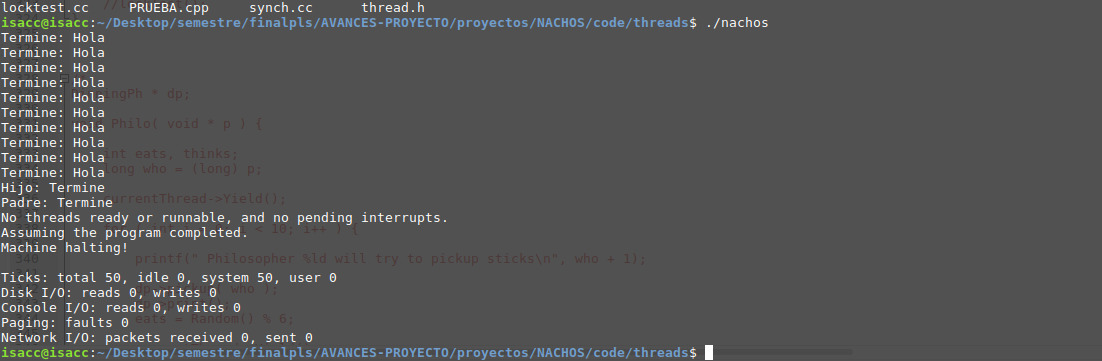
El programa usado para probar el Lock se encuentra en locktest.cc de threads. El programa crea dos hilos y usa un Lock para controlar el acceso a la consola y poder imprimir unos mensajes. Como se puede ver en la siguiente imagen gracias al Lock un hilo imprime todo lo que tiene que imprimir, libera el candado y el otro hilo puede imprimir lo que le toca.



**Barrier**

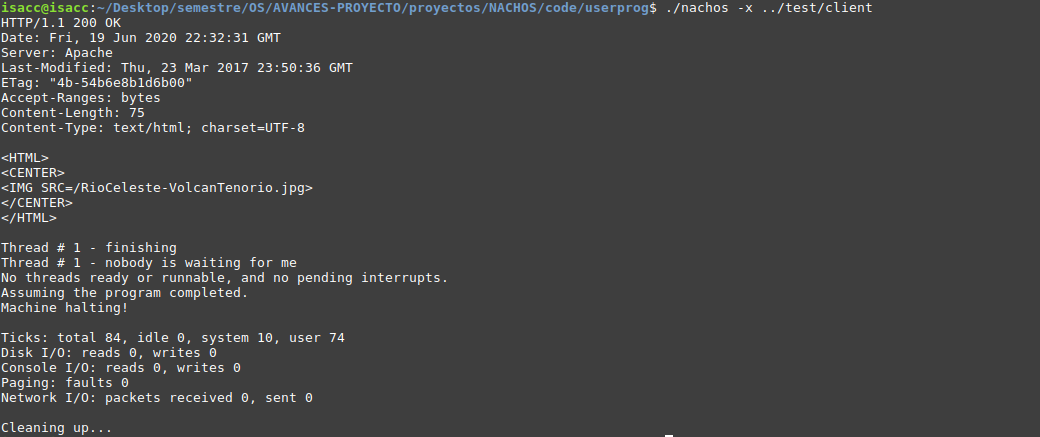
Nombre de la función para prueba: barrierTest

El programa para probar la Barrier se llama barriertest.cc, ubicado en la carpeta “threads”. Se crean dos hilos, uno realiza una función corta y otro un ciclo más largo, el primer hilo entra al barrier.Wait() primero, por lo que espera al hilo dos hasta que realice su ciclo, cuando ese hilo dos termina, los dos imprimen que terminaron, probando así el funcionamiento de la barrera.



**Cliente NachOS**

En la carpeta userprog se encuentra client, que se puede ejecutar como se ve en la figura de abajo.



Esta programa se comunica con el servidor con la direccion IP 163.178.104.187 y le hace una solicitud para obtener el index.html del servidor. Luego de obtenerlo usando los llamados de sistemas implementados para NachOS, imprime en consola el index.html.

**Referencias**

Repositorio: <https://github.com/gustavojh106/AVANCES-PROYECTO>