

Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ciencias de la Computación e
Informática

CI-1310 Sistemas Operativos

Grupo 02|1

II Semestre

Tarea programada NachOS#1

Profesor:

Francisco Arroyo

Estudiantes:

Dennis Abarca Quesada | A70014 | grupo 01

Steven Rojas Lizano 1 | A75623 | grupo 02

October 31, 2016

Contents

1	introducción	1
2	Objetivos	1
3	Descripción	2
4	Desarrollo	3
4.1	Estrategia de implementación	3
4.2	Archivos modificados	3
5	Pruebas	5
5.1	Archivo prueba create1.c	5
5.2	Archivo prueba: fork4.c	6
5.3	Prueba pingPong.c	6
6	Manual de usuario	9
6.1	Requerimientos	9
6.2	Restricciones	9
7	Bibliografía	10

1 Introducción

En esta tarea programada se realizó la implementación de los llamados al sistema de nachOS, en los cuales se realizaron modificaciones a los archivos de addressSpace, system, exception y machine.

2 Objetivos

- Leer y entender la parte del sistema operativo NachOs que se encuentra en la carpeta descargable.
- Implantar el manejo de excepciones y llamados al sistema.
- Implantar multiprogramación.
- Implantar programas de usuario multi-hilos.

3 Descripción

1. Implantar el manejo de excepciones y llamados al sistema. Se deben soportar todos los llamados al sistema definidos en "syscall.h". Presentamos una rutina en ensamblador "syscall" que provee la manera de invocar un llamado al sistema desde una rutina C (Unix tiene un método similar, intente "man syscall"). Usted necesita completar la parte 2 de esta asignación con el fin de probar los llamados al sistema "exec" y "wait"
2. Implantar multiprogramación. El código que presentamos le restringe a solo poder correr un programa de usuario a la vez. Usted necesita:
 - (a) Encontrar la manera de asignar los marcos de memoria física de tal manera que varios programas de usuario puedan ser colocados en la memoria principal a la vez. (ver "bitmap.h")
 - (b) Proveer una manera de copiar datos desde/hacia el kernel desde/hacia el espacio de direcciones virtual del usuario (ahora las direcciones que el programa del usuario 've' no son las mismas que el kernel 've')
3. Agregar sincronización a las rutinas que crean e inician el espacio de direcciones, de tal manera que puedan ser accedidas concurrentemente por múltiples programas. Note que "scheduler.cc" ahora guarda y recupera el estado de la máquina en los cambios de contexto. Es deseable tener algunas rutinas como "cp" y "cat" de Unix, y utilizar el shell que proveemos en el directorio "test" para verificar el manejo de llamados al sistema y la multiprogramación
4. Implantar programas de usuario multi-hilos. Implante los llamados al sistema "fork" y "yield", que le permita al usuario llamar a una rutina en el mismo espacio de direccionamiento, y hacer ping pong entre los hilos (Ayuda: necesita cambiar la manera actual del kernel para asignar memoria en el espacio de direcciones del usuario para cada una de las pilas de los threads)
5. (Extra, por 5%)La versión actual del llamado al sistema "exec" no ofrece ninguna manera para que el usuario pueda pasar parámetros o argumentos al nuevo espacio de direcciones creado. Unix permite esto, por ejemplo, se pueden pasar argumentos en la línea de comandos al nuevo espacio de direcciones. Implantar esta funcionalidad del "exec"

4 Desarrollo

4.1 Estrategia de implementación

La estrategia a utilizar para la solución del problema de la implementación de los llamados del sistema fue mediante la modificación de los archivos del sistema operativo y la adición de las respectivas funciones en el archivo de exception, esto para poder atrapar los llamados al sistema cuando se ejecute una exception del tipo system call.

Gran parte de la estrategia a seguir fue dada mediante los laboratorios de Nachos 6 , 7 y 8 donde se realiza la separación de los casos de excepciones mediante un switch, la implementación de los llamados al sistema de open, read y write, fork.

Finalmente la creación del archivo de nachosFileTabla para poder tener un control de los archivos que se encontraban abiertos por los procesos activos.

Los demás llamados al sistema fueron implementados mediante la utilización de llamados al sistema de unix y métodos propios del sistema de NachOS como lo son el yield , exit y los llamados al sistema para la utilización de semáforos.

4.2 Archivos modificados

Para la implementación de los llamados del sistema fueron necesarios realizar algunas modificaciones a los archivos principales del sistema operativo nachOS, los archivos de exception.cc, addressSpace tuvieron grandes cambios por lo que no se incluyen en el informe.

Sin embargo para el manejo de los hilos para el llamado de Fork fue necesario la utilización de un bitmap y un arreglo de hilos actuales, además para el uso de las páginas de memoria del simulador MIPS se utilizó también un bitmap para las páginas utilizadas. esto se puede ver en el siguiente extracto del archivo threads.h y threads.cc

```
1 class Thread{
2 ...
3     void Print() { printf("%s, ", name); }
4
5     int id;
6     BitMap * waitingProcess;
7     Semaphore * sem;
8     int archivosUsados[MaxArchivos];
9     int numArchivosUsados;
10
11 private:
12 ...
```

Luego se realiza la inicialización de estas variables en el constructor de la clase Thread.

```
1 Thread::Thread(const char* threadName)
2 {
3     name = threadName;
4     stackTop = NULL;
5     stack = NULL;
6     status = JUST_CREATED;
7     openFilesTable = new NachosOpenFilesTable();
8     waitingProcess = new BitMap(20);
```

```

10     sem = new Semaphore("semaforo",0);
11     numArchivosUsados = 0;
12     for(int i = 0; i < 100; i++){
13         archivosUsados[i] = -1;
14     }
15 #ifdef USER_PROGRAM
16     space = NULL;
17 #endif
18 }

```

Además de la clase Thread, también se realizaron pequeñas modificaciones en la clase system para poder implementar los llamados al sistema

```

1 Class System{
2     ...
3     //variables nuevas
4     extern BitMap * pageTableMap;
5     extern BitMap * bitmapSemaforos;
6     extern Semaphore* semaforosActuales[20];
7     extern BitMap * hilosMap;
8     extern Thread *hilosActuales[MaxHilos];
9     extern Semaphore* semExec;
10    extern char nombreEx[100];
11    extern int *states;
12    ...
13 }

```

Estas variables son nuevamente declaradas en el archivo system.cc y luego inicializadas mediante el método initialize, cabe destacar que estas variables son declaradas como globales para todo el sistema operativo por lo que pueden ser accedidas desde cualquier parte del código.

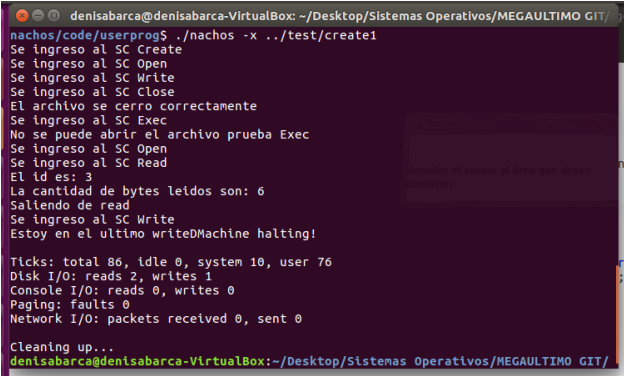
5 Pruebas

Para validar la implementación llevado a cabo, se probará el sistema operativo con los archivos de prueba create1.c, fork4.c y pingpong.c los cuales son archivos de prueba disponibles en la página web del curso de laboratorio del curso.

5.1 Archivo prueba create1.c

```
1 int main(){
2     int fd;
3     int e;
4     char * buf;
5     Create("archivo.nuevo");
6     fd = Open("archivo.nuevo");
7     Write("prueba", 6, fd);
8     Close(fd);
9     // Exec("../test/rillo");
10    Exec("prueba Exec");
11
12    //char* buf = new int[6];
13    fd = Open("archivo.nuevo");
14    Read(buf, 6, fd);
15    Write(buf, 6, 1);
16
17    Halt();
18    return 0;
19 }
```

En este caso se prueba los system calls Create, Open, Write, Close, Read y Exec. Como se puede ver en la Ilustración, se ingresa con éxito a los system calls Create, Open y Close. Una vez abierto y cerrado el archivo, envía un mensaje indicando que el archivo se cerró con éxito. Al ingresar a Exec, muestra que el archivo no se pudo ejecutar dado que no existe. Posteriormente vuelve a abrir el archivo con el SC Open y lee la cantidad de bytes solicitados. En la imagen se muestra que efectivamente esto ocurre.



```
denisabarca@denisabarca-VirtualBox: ~/Desktop/Sistemas Operativos/MEGAULTIMO GIT/
nachos/code/userprog$ ./nachos -x ../test/create1
Se ingreso al SC Create
Se ingreso al SC Open
Se ingreso al SC Write
Se ingreso al SC Close
El archivo se cerro correctamente
Se ingreso al SC Exec
No se puede abrir el archivo prueba Exec
Se ingreso al SC Open
Se ingreso al SC Read
El id es: 3
La cantidad de bytes leidos son: 6
Saliendo de read
Se ingreso al SC Write
Estoy en el ultimo writeMachine halting!

Ticks: total 86, idle 0, system 10, user 76
Disk I/O: reads 2, writes 1
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
denisabarca@denisabarca-VirtualBox:~/Desktop/Sistemas Operativos/MEGAULTIMO GIT/
```

Figure 1: Prueba de create.C

5.2 Archivo prueba: fork4.c

```
1  #include "syscall.h"
2
3  void nada(void);
4  void todo(void);
5
6  int semaforoID;
7
8  int main(){
9      semaforoID = SemCreate(0);
10     Fork(nada);
11 }
12
13
14 void nada(){
15     Fork(todo);
16     Write("Nada!", 5, 1);
17     SemSignal(semaforoID);
18 }
19
20 void todo(){
21     Write("Vamos", 5, 1);
22     SemWait(semaforoID);
23     Write("Final", 5, 1);
24 }
```

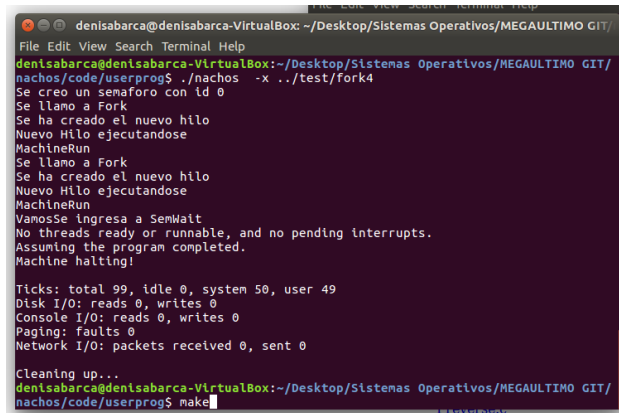
En este caso, se prueba el system Call Fork y los system calls asociados a la funcionalidad de los semáforos, es decir; semCreate, semDestroy, semSignal, semCreate.

Al inicio, se ingresa con éxito al SC semCreate e inicializa el semáforo en cero. Posteriormente se invoca el System Call Fork. Al crear el Fork la dirección de referencia ejecuta el método void Nada(), en donde al mismo tiempo vuelve a ejecutar el llamado al sistema Fork ejecutando el método void todo().

En ambos métodos, Vamos y Todo debería ejecutarse los métodos Write mostrando en pantalla los strings 'Vamos' y 'Nada'. Sin embargo, esta prueba presentó el error de que no mostró en pantalla la palabra 'Nada' pero sí 'Vamos' de los llamados al sistema Write. Lo anterior puede deberse a una falla en la sincronización entre el proceso padre y el proceso hijo.

5.3 Prueba pingPong.c

```
1  #include "syscall.h"
2
3  void SimpleThread(int);
4
5  int
6  main( int argc , char * argv[] ) {
7
8      Fork(SimpleThread);
9      SimpleThread(1);
10
11     Write("Main \n", 7, 1);
12 }
```

```
denisabarca@denisabarca-VirtualBox: ~/Desktop/Sistemas Operativos/MEGAULTIMO GIT/
denisabarca@denisabarca-VirtualBox:~/Desktop/Sistemas Operativos/MEGAULTIMO GIT/
nuchos/code/userprog$ ./nuchos -x ../test/fork4
Se creo un semaforo con id 0
Se llamo a Fork
Se ha creado el nuevo hilo
Nuevo Hilo ejecutandose
MachineRun
Se llamo a Fork
Se ha creado el nuevo hilo
Nuevo Hilo ejecutandose
MachineRun
VamosSe ingresa a SemWait
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 99, idle 0, system 50, user 49
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
denisabarca@denisabarca-VirtualBox:~/Desktop/Sistemas Operativos/MEGAULTIMO GIT/
nuchos/code/userprog$ make
```

Figure 2: Prueba de Fork.C

```
13
14 void SimpleThread(int num)
15 {
16
17     if (num == 1) {
18         for (num = 0; num < 5; num++) {
19             Write("Hola 1\n", 7, 1);
20             Yield();
21         }
22     }
23
24     else {
25         for (num = 0; num < 5; num++) {
26             Write("Hola 2\n", 7, 1);
27             Yield();
28         }
29     }
30     Write("Fin de\n", 7, 1);
31 }
```

En este caso se analiza la funcionalidad de los métodos Fork y Yield. Consiste en crear un nuevo proceso y posteriormente alternar el hilo que está corriendo empleando el llamado al sistema Yield.

Lo primero que ocurre es la creación de un nuevo hilo mediante Fork y posteriormente se llamara al método SimpleThread.

Como ahora hay dos procesos e ingresarán a SimpleThread() uno con un valor de argumento igual a 1 y el otro distinto de 1, el llamado al sistema Yield() intercambiará los procesos de manera alternada hasta que se cumple el condicional en el ciclo for. En la Imagen adjunta se muestra que efectivamente esto se logra, ya que Hola 2 y Hola 1 cambian de manera alternada.

```
denisabarca@denisabarca-VirtualBox: ~/Desktop/Sistemas Operativos/MEGAULTIMO GIT/  
denisabarca@denisabarca-VirtualBox:~/Desktop/Sistemas Operativos/MEGAULTIMO GIT/  
nuchos/code/userprog$ ./nuchos -x ../test/pingPong  
Se llamo a Fork  
Se ha creado el nuevo hilo  
Se ingreso al SC Write  
Hola 1  
Se ingreso al SC YieldNuevo Hilo ejecutandose  
MachineRun  
Se ingreso al SC Write  
Hola 2  
Se ingreso al SC YieldSe ingreso al SC Write  
Hola 1  
Se ingreso al SC YieldSe ingreso al SC Write  
Hola 2  
Se ingreso al SC YieldSe ingreso al SC Write  
Hola 1  
Se ingreso al SC YieldSe ingreso al SC Write  
Hola 2  
Se ingreso al SC YieldSe ingreso al SC Write  
Hola 1  
Se ingreso al SC YieldSe ingreso al SC Write  
Hola 2  
Se ingreso al SC YieldSe ingreso al SC Write  
Hola 1
```

Figure 3: Prueba de Ping Pong.C

6 Manual de usuario

6.1 Requerimientos

El sistema posee los siguientes requerimientos para su correcto funcionamiento

- Sistema operativo Fedora o Ubuntu
- Arquitectura 64 bits
- Ambiente NachOS
- Compilador: GNU GCC compiler

6.2 Restricciones del programa

- El sistema solamente puede ser ejecutado desde la terminal.
- El sistema solamente trabaja con archivos codificados en UTF-8 y codigos fuente del lenguaje C.
- El simulador MIPS se basa en un único procesador.
- el máximo número de páginas es de 32 y cada una es de 128 bytes por lo que los programas de usuario no deben ser mayores a 4kB
- EL número máximo de hilos a guardar en la cola es de 20 hilos

7 Bibliografía

[1] Silberchatz, Abraham, Galvin, Peter & Gagne, Greg. *Operating Systems Concepts*. Novena edición, Addison Wesley Publishing Co., Mass., 2013