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

CI-1310 Sistemas Operativos
Grupo 02
I Semestre

# Proyecto NachOS # 1

**Profesor:**
Francisco Arroyo

**Estudiantes:**
Luis Porras Ledezma | B65477
Jose Pablo Ramírez Méndez | B65728

**15 de Junio del 2018**

# Índice

# 1. Introducción

En este proyecto se van implementar 15 system calls para el ”sistema operativo”(emulador de un sistema operativo) NachOS y al mismo tiempo se va a obtener una idea general de como los sistemas operativos reales y más complejos implementan equivalentes a estos system calls. Además se va a implementar en NachOS un page table que le permita a este sistema operativo administra su memoria principal, la cual esta dividida en 32 paginas de 128 bytes cada una y además por defecto solo puede cargar un proceso en memoria a la vez. De igual forma se pretende con la mejora de page table por defecto de NachOS entender una manera en la que otros sistemas operativos administran su memoria.

# 2. Objetivos

- Crear y comprender de forma general que hace un sistema operativo ”por debajo” al ejecutar un system call.
- Analizar y entender una forma en la los sistemas operativos administran su memoria principal.

# 3. Descripción

Este proyecto se centra en los servicios del sistema operativo hacia un usuario programador por lo que prácticamente todo lo que debe implementar se debe hacer en la carpeta ”userprog” de NachOS.
Para la realización de este proyecto se pretende implementar en NachOS los siguientes system calls, estos se deben implementar pensando en un usuario (rojo con sombrero azul) que los va a ejecutar desde sus código por lo que los system call deben respetar la estructura descrita en ”syscall.h”:

1. Halt
   **Descripción:** Este llamado al sistema detiene por completo al sistema operativo Nachos y termina su ejecución.

```
/* Stop Nachos, and print out performance stats */
void Halt();
```

2. Exit
   **Descripción:** Este llamado al sistema finaliza la ejecución del hilo que este corriendo y que realice el llamado. Dicho hilo, antes de terminar, debe verificar si hay más hilos listos para correr a través del `scheduler` de nachos y también, verificar si hay algún otro hilo que hizo `join` al actual para despertarlo.

```
/* This user program is done (status = 0 means exited normally). */
void Exit(int status);
```

3. Exec
   **Descripción:** Realiza le ejecución de un archivo válido compilado para el procesador `MIDS` a través de nachos. El nombre del archivo es recibido por parámetro y se devuelve un entero con la identificación del hilo que corre dicho nuevo archivo.

```
/* Run the executable, stored in the Nachos file "name", and return the
 * address space identifier
 */
SpaceId Exec(char *name);
```

4. Join
   **Descripción:** Este llamado del sistema recibe un identificador válido de un archivo que este siendo ejecutado por un hilo, con el fin de esperar por la ejecución y finalización del mismo para continuar con su ejecución.

```
/* Only return once the the user program "id" has finished.
 * Return the exit status.
 */
int Join(SpaceId id);
```

5. Create
   **Descripción:** Este llamado al sistema, realiza la creación de un nuevo archivo a través del llamado al sistema de UNIX que recibe el mismo nombre.

```
/* Create a Nachos file, with "name" */
void Create(char *name);
```

6. Open
   **Descripción:** Este llamado al sistema realiza la apertura de un archivo existente a través del llamado al sistema de UNIX que recibe el mismo nombre. Para el usuario de nachos, se retorna un identificador válido que hace referencia 1:1 con el archivo abierto. Por parámetro se recibe el nombre del archivo para ser abierto.

```
/* Open the Nachos file "name", and return an "OpenFileId" that can
 * be used to read and write to the file.
 */
OpenFileId Open(char *name);
```

7. Write
   **Descripción:** Este llamado al sistema realiza la escritura de un buffer de caracteres de un tamaño definido, dentro de un archivo con identificador y válido de archivo abierto desde nachos.

```
/* Write "size" bytes from "buffer" to the open file. */
void Write(char *buffer, int size, OpenFileId id);
```

8. Read
   **Descripción:** Este llamado al sistema realiza la lectura de un número de caracteres dado desde un archivo abierto con identificador válido de nachos. Dicha lectura se almacena en el buffer de datos que ingresa por parámetro. Por último, se retorna la cantidad de caracteres leídos.

```
/* Read "size" bytes from the open file into "buffer".
 * Return the number of bytes actually read -- if the open file isn't
 * long enough, or if it is an I/O device, and there aren't enough
 * characters to read, return whatever is available (for I/O devices,
 * you should always wait until you can return at least one character).
 */
int Read(char *buffer, int size, OpenFileId id);
```

9. Close
   **Descripción:** Finalmente, este llamado al sistema relacionado con archivos realiza la clausura de un archivo válido a través del identificador proporcionado por nachos.

```
/* Close the file, we're done reading and writing to it. */
void Close(OpenFileId id);
```

10. Fork
    **Descripción:** Este llamado del sistema relacionado con la ejecución de hilos para el usuario, crea un nuevo hilo de kernel para la ejecución de una función de usuario. Este nuevo hilo comparte el segmento de código y de datos del hilo que lo crea, pero con su segmento de pila individual.

```
/* Fork a thread to run a procedure ("func") in the *same* address space
 * as the current thread.
 */
void Fork(void (*func)());
```

11. Yield
    **Descripción:** Este llamado al sistema realiza llamado al Yield() de nachos para agregar al hilo actual

al final de la cola de los hilos listos para correr. Lo anterior tiene sentido ya que la relación de hilos de usuario creados con Fork() e hilos de kernel es 1:1.

```
/* Yield the CPU to another runnable thread, whether in this address space
 * or not.
 */
void Yield();
```

12. SemCreate
    **Descripción:** Para estos llamados al sistema relacionados con la creación de semáforos para el usuario, el procedimiento recae en crear un semáforo de kernel para el usuario y devolverle un identificador válido al cual referirse. El parámetro indica el valor inicial para crear el semáforo.

```
/* SemCreate creates a semaphore initialized to initval value
 * return the semaphore id
 */
int SemCreate( int initval );
```

13. SemDestroy
    **Descripción:** Llamado al sistema que destruye un semáforo de kernel proporcionado al usuario, a través del identificador válido dado al ser creado.

```
/* SemDestroy destroys a semaphore identified by id */
int SemDestroy( int SemId );
```

14. SemSignal
    **Descripción:** Llamado al sistema que realiza la operación Signal() al semáforo indicado por el identificador válido de nachos.

```
/* SemSignal signals a semaphore, awakening some other thread if necessary */
int SemSignal( int SemId );
```

15. SemWait
    **Descripción:** Llamado al sistema que realiza la operación Wait al semáforo indicado por el identificador válido de nachos.

```
/* SemWait waits a semaphore, some other thread may awake if one blocked */
int SemWait( int SemId );
```

Además se debe modificar los archivos ”addrspace.h” y ”addrspace.cc” también de la carpeta userprog de NachOS, para permitir que NachOS tenga un mejor manejo de memoria y específicamente de su page table.

# 4. Desarrollo

El comienzo de la primera etapa de este proyecto, es decir de la implementación de los 15 system call para NachOS, se realizo siguiendo las instrucciones del laboratorio 8, en el cual se explicaba como implementar los system calls: Open y Write.
Lo primero que se hizo fue modificar el método ExceptionHandler del archivo ”exception.cc” de manera que este ahora pueda llamar un system call o dar un mensaje de error correctamente según corresponda (es decir segun los valores de las variables: which:ExceptionType y type:int), tal y como se muestra a continuación:

```
void ExceptionHandler(ExceptionType which)
{
  int type = machine->ReadRegister(2);

  switch ( which ) {

    case SyscallException:
    switch ( type )
    {
      case SC_Halt:                 //System call # 0
      Nachos_Halt();
      break;
      case SC_Exit:                 //System call # 1
      Nachos_Exit();
      break;
      case SC_Exec:                 //System call # 2
      Nachos_Exec();
      break;
      case SC_Join:                 //System call # 3
      Nachos_Join();
      break;
      case SC_Create:               //System call # 4
      Nachos_Create();
      break;
      case SC_Open:                 //System call # 5
      Nachos_Open();
      break;
      case SC_Read:                 //System call # 6
      Nachos_Read();
      break;
      case SC_Write:                //System call # 7
      Nachos_Write();
      break;
      case SC_Close:                //System call # 8
      Nachos_Close();
      break;
      case SC_Fork:                 //System call # 9
      Nachos_Fork();
      break;
      case SC_Yield:                //System call # 10
      currentThread->Yield();
      returnFromSystemCall();
      break;
      case SC_SemCreate:            //System call # 11
      Nachos_SemCreate();
```

```
46          returnFromSystemCall();
47          break;
48          case SC_SemDestroy:           //System call # 12
49          Nachos_SemDestroy();
50          returnFromSystemCall();
51          break;
52          case SC_SemSignal:            //System call # 13
53          Nachos_SemSignal();
54          returnFromSystemCall();
55          break;
56          case SC_SemWait:              //System call # 14
57          Nachos_SemWait();
58          returnFromSystemCall();
59          break;
60          default:
61          printf("Unexpected syscall exception %d\n", type );
62          ASSERT(false);
63          break;
64        }
65       break;
66       case PageFaultException:
67       printf("\nPageFaultException\n");
68       ASSERT(false);
69       break;
70       case ReadOnlyException:
71       printf("\nReadOnlyException\n");
72       ASSERT(false);
73       break;
74       case BusErrorException:
75       printf("\nBusErrorException\n");
76       ASSERT(false);
77       break;
78       case AddressErrorException:
79       printf("\nAddressErrorException\n");
80       ASSERT(false);
81       break;
82       case OverflowException:
83       printf("\nOverflowException\n");
84       ASSERT(false);
85       break;
86       case IllegalInstrException:
87       printf("\nIllegalInstrException\n");
88       ASSERT(false);
89       break;
90       case NumExceptionTypes:
91       printf("\nNumExceptionTypes\n");
92       ASSERT(false);
93       break;
94       default:
95       printf("\nUnexpected exception %d\n", which );
96       ASSERT(false);
97       break;
98     }
99   }
```

Además se agrego al archivo ”execption.cc” de la carpeta userprog de NachOS el método return-

FromSystemCall necesario para actualizar los instrucción pointers de MIPS y evitar que el ”procesador” se quedara en ciclado en cada system call.

```
/*RETURN FROM SYSTEM CALL*/

void returnFromSystemCall() {

  int pc, npc;

  pc = machine->ReadRegister( PCReg );
  npc = machine->ReadRegister( NextPCReg );
  machine->WriteRegister( PrevPCReg, pc );        // PrevPC <- PC
  machine->WriteRegister( PCReg, npc );           // PC <- NextPC
  machine->WriteRegister( NextPCReg, npc + 4 );   // NextPC <- NextPC + 4

}// returnFromSystemCall
```

Para la implementación de los system calls: Create, Close, Open, Read y Write fue necesaria la implementación de una tabla de archivos abiertos, implementada como se muestra a continuación: ”nachostabla.h”:

```
#ifndef NachosOpenFilesTable_H
#define NachosOpenFilesTable_H

#include "bitmap.h"

#define MAX_FILES 128


class NachosOpenFilesTable {
  public:
    NachosOpenFilesTable();       // Initialize
    ~NachosOpenFilesTable();      // De-allocate

    int Open( int UnixHandle ); // Register the file handle
    int Close( int NachosHandle );      // Unregister the file handle
    bool isOpened( int NachosHandle );
    int getUnixHandle( int NachosHandle );
    void addThread();           // If a user thread is using this table, add it
    void delThread();           // If a user thread is using this table, delete it

    void Print();               // Print contents

  private:
    int * openFiles;            // A vector with user opened files
    BitMap * openFilesMap;      // A bitmap to control our vector
    int usage;                  // How many threads are using this table
};
#endif
```

”nachostabla.cc”:

```
#include "nachostabla.h"

NachosOpenFilesTable::NachosOpenFilesTable()
```

```
{
        this->openFiles = new int[ MAX_FILES ];
        this->openFiles[0] = 0; //std::in
        this->openFiles[1] = 1; //std::out
        this->openFiles[2] = 2; //std::cerr

        for (int x = 3; x < MAX_FILES; ++x)
                this->openFiles[x] = 0;

        this->openFilesMap = new BitMap( MAX_FILES );
        this->openFilesMap->Mark(0);
        this->openFilesMap->Mark(1);
        this->openFilesMap->Mark(2);
}


NachosOpenFilesTable::~NachosOpenFilesTable()
{
        if(usage <= 0){
                        delete[] openFiles;
                        delete openFilesMap;
                        printf("Ultimo hilo borra tabla de archivos\n");
        }
}

bool NachosOpenFilesTable::isOpened( int NachosHandle ){
    if(NachosHandle >= 0 && NachosHandle < MAX_FILES){
        return openFilesMap->Test(NachosHandle);
    }
    return false;
}

int NachosOpenFilesTable:: Open( int UnixHandle )
{
        int freeFile = this->openFilesMap->Find();
        if (freeFile != -1)
        {
                this->openFiles[ freeFile ] = UnixHandle;
        }
        return freeFile;
}

int NachosOpenFilesTable::Close( int NachosHandle )
{
    if(isOpened(NachosHandle)){
            this->openFilesMap->Clear( NachosHandle );
            this->openFiles[ NachosHandle ] = 0;
        return   0;
    }
    return -1;
}

int NachosOpenFilesTable::getUnixHandle( int NachosHandle ){
    if(isOpened(NachosHandle)){
        return openFiles[NachosHandle];
    }
```

```
60      return -1;
61  }
62
63  void NachosOpenFilesTable::addThread(){
64      ++usage;
65  }
66
67  void NachosOpenFilesTable::delThread(){
68      --usage;
69  }
70
71  void NachosOpenFilesTable::Print(){
72      for(int i = 0; i < MAX_FILES; ++i){
73          printf("Nachos handle: %d, Unix handle: %d\n", i, openFiles[i]);
74      }
75      printf("\n");
76  }
```

De forma similar para la implementación de los system calls relacionados a semaforos (SemCreate, SemDestroy, SemSignal y SemWait) fue necesario la creación una tabla de semáforos como se muestra a continuación:

”NachosSem.h”

```
1   #ifndef NACHOSSEMS_H
2   #define NACHOSSEMS_H
3
4   //#include "synch.h"
5   #include "bitmap.h"
6
7   #define MAX_SEMS 6
8
9   class NachosSems
10  {
11  public:
12    NachosSems();
13    ~NachosSems();
14
15    int registerSem( long s ); // Register the Nachos sem pointer
16    long unRegisterSem( int id );      // Unregister the fNachos sem pointer
17    long getNachosPointer( int id );
18    void addSem();                // If a user thread is using this table, add it
19    void delSem();                // If a user thread is using this table, delete it
20    void print();
21
22    private:
23    BitMap * openSemsMap; // A bitmap to control our vector
24    long* semaphores;             // A vector with user created semaphores
25    int usage;                    // How many threads are using this table
26
27  };
28
29  #endif
```

”NachosSem.cc”

```
#include "NachosSems.h"

NachosSems::NachosSems()
{
  usage = 0;
  semaphores = new long[MAX_SEMS];
  openSemsMap = new BitMap(MAX_SEMS);
  // initialize pointer in NULL
  for ( int x = 0; x < MAX_SEMS; ++x )
  {
    semaphores[ x ] = -1;
  }
}

NachosSems::~NachosSems()
{
  if( usage == 0 ){
    delete openSemsMap;
    delete[] semaphores;
    printf("Borrando tabla de semaforos. Ultimo Hilo\n");
  }
}

void NachosSems::print()
{
  for (int x= 0; x < MAX_SEMS; ++x)
  printf("Valor en el vector de info: %ld\n", semaphores[x] );

}

int NachosSems::registerSem( long s )
{
  int freeSemSpace = this->openSemsMap->Find();

  if (-1 != freeSemSpace )
  {
  //  printf("Valor libre para registrar sem: %d\n", freeSemSpace );
    semaphores[ freeSemSpace ] = s;
  }
  //print();
  return freeSemSpace;
}

long NachosSems::unRegisterSem( int id )
{
    if ( semaphores[ id ] != -1 )
    {
      this->openSemsMap->Clear( id );
      long copy = semaphores[ id ];
      semaphores[ id ] = -1;
      return copy;
    }
    return -1;
}
long NachosSems::getNachosPointer( int id )
{
```

```
57      //print();
58      return semaphores[ id ];
59    }
60    void NachosSems::addSem()
61    {
62      ++usage;
63    }
64    void NachosSems::delSem()
65    {
66      --usage;
67    }
```

Como es nesesario que cada ”familia” de Threads de NachOS tenga su propia tabla de archivos abiertos y tabla de semaforos se tuvo que agregar instancias de ambas clase anteriores en el constructor de la clase Thread de NachOS (revisar ”Thread.h” y ”Thread.cc” en la carpeta threads).

Implementación de los diferentes system calls en NachOS (específicamente en ”exception.cc”) raptando las declaraciones descritas anteriormente:

- Halt
  Este era el único de los system calls de este proyecto que ya estaba implementado.

```
void Nachos_Halt() {                    // System call 0
  DEBUG('a', "Shutdown, initiated by user program.\n");
  interrupt->Halt();

}// Nachos_Halt
```

Figura 1: Implementación Halt

- Exit

```
void Nachos_Exit(){
  /* This user program is done (status = 0 means exited normally). */
  //void Exit(int status);
  Thread* nextThread;
  IntStatus oldLevel = interrupt->SetLevel(IntOff);

  DEBUG('t', "Finishing thread \"%s\"\n", currentThread->getName());

  //printf("currentThread: %ld\n", (long)currentThread);

  for(int ind = 0; ind < 128; ++ind){
    if(execFilesMap->Test(ind)){
      //printf("I'm a EXEC thread %ld\n", execFiles[ind]->threadId);
      if((long)currentThread == execFiles[ind]->threadId){
        if(execFiles[ind]->s != NULL){
          //printf("Someone is wainting for me: %ld\n", execFiles[ind]->threadId);
          execFiles[ind]->s->V();
        }
        else{
          //printf("No one is wainting for me: %ld\n", execFiles[ind]->threadId);
          delete execFiles[ind];
          execFilesMap->Clear(ind);
        }
      }
    }
  }
  machine->WriteRegister(2, machine->ReadRegister(4));
  nextThread = scheduler->FindNextToRun();
  if (nextThread != NULL) {
    scheduler->Run(nextThread);
  }else
  {
    currentThread->Finish();
  }
  interrupt->SetLevel(oldLevel);
  //returnFromSystemCall();
}//Nachos_Exit
```

Figura 2: Implementación Exit

- Exec

```
void Nachos_Exec(){
  /* Run the executable, stored in the Nachos file "name", and return the
  * address space identifier
  */
  //SpaceId Exec(char *name);

  DEBUG( 't', "Entering EXEC System call\n" );
  long r4 = machine->ReadRegister( 4 ); // read from register 4
  char name[256] = {0}; // need to store file name to unix create sc
  int c, i; // counter
  i = 0;

  do
  {
    machine->ReadMem( r4 , 1 , &c ); // read from nachos mem
    r4++;
    name[i++] = c;
  }while (c != 0 );
  std::string s = name;
  joinS* newE = new joinS();

  // We need to create a new kernel thread to execute the user thread
  Thread * newT = new Thread( "HILO EXEC" );
  long fileToExec = execFilesMap->Find();
  if(fileToExec == -1){
    machine->WriteRegister(2, fileToExec );
    return;
  }

  newE->threadId = (long) newT;
  newE->fileName = s;
  execFiles[fileToExec] = newE;

  newT->Fork( NachosExecThread, (void*) fileToExec ); // ojo se elimino warning
  machine->WriteRegister(2, fileToExec );
  returnFromSystemCall(); // This adjust the PrevPC, PC, and NextPC registers

  DEBUG( 't', "Exiting EXEC System call\n" );
}// Nachos_Exec
```

Figura 3: Implementación Exec

Implementación del método NachosExecThread necesario para el system call Exec.

```
void NachosExecThread( void* id)
{
  joinS* info = execFiles[(long)id];

  OpenFile *executable = fileSystem->Open(info->fileName.c_str());
  AddrSpace *space;

  if (executable == NULL) {
    printf("Unable to open file %s\n",info->fileName.c_str());
    return;
  }
  space = new AddrSpace( executable );
  delete currentThread->space; // i dont need may space anymore
  currentThread->space = space;

  delete executable;              // close file
  space->InitRegisters();   // set the initial register values
  space->RestoreState();    // load page table register
  machine->Run();     // jump to the user progam
  printf("\t\t\t\t\tError\n");
  ASSERT(false);      // machine->Run never returns;
}
```

Figura 4: Implementación NachosExecThread

- Join Declaración de las estructuras usadas en el Join, Exec y Exit.

```
void Nachos_Join()
{
  /* Only return once the the user program "id" has finished.
  * Return the exit status.
  */
  //int Join(SpaceId id);

  DEBUG( 't', "Entering JOIN System call\n" );

  //First I need to read the SpaceID of the thread I must wait for
  long id = machine->ReadRegister( 4 ); // read from register 4
  //I need to make sure it is a valid thread
  if(execFilesMap->Test(id)){
    //I need to create a Semaphore asosieted with this SpaceID
    Semaphore* joinSem = new Semaphore("JOIN Semaphore", 0);
    execFiles[id]->s = joinSem;
    joinSem->P();
    //printf("%ld: %s\n", execFiles[id]->threadId, "just give me a sigh");
    delete execFiles[id];
    execFilesMap->Clear(id);
    machine->WriteRegister(2, 0);
  }
  else{ //invalid SpaceID
    //printf("%s\n", "Error JOIN: ¡Invalid SpaceID!");
    machine->WriteRegister(2, -1 );
  }
  returnFromSystemCall(); // This adjust the PrevPC, PC, and NextPC registers
}// Nachos_Join
```

Figura 5: Implementación Join

```
struct joinS
{
  long threadId;
  std::string fileName;
  Semaphore* s;
  inline joinS():threadId(-1),fileName(),s(NULL){}
};

joinS** execFiles = new joinS*[128];
BitMap* execFilesMap = new BitMap(128);
```

Figura 6: Implementación estructuras para el Join

- Create

```
void Nachos_Create(){
  ///printf("Creating!\n");
  int r4 = machine->ReadRegister(4); // read from register 4
  char fileName[256] = {0}; // need to store file name to unix create sc
  int c, i; // counter
  i = 0;

  do
  {
    machine->ReadMem( r4 , 1 , &c ); // read from nachos mem
    r4++;
    fileName[i++] = c;
  }while (c != 0 );

  int createResult = creat (fileName, O_CREAT | S_IRWXU ); // create with read write destroy authorization
  close(createResult);
  if (-1 == createResult )
  {
    printf("Error: unable to create new file\n");
  }
  returnFromSystemCall();   // Update the PC registers
}// Nachos_Create
```

Figura 7: Implementación Create

- Open

```
void Nachos_Open() {                    // System call 5
  /* System call definition described to user

  */
  // Read the name from the user memory, see 5 below
  // Use NachosOpenFilesTable class to create a relationship
  // between user file and unix file
  // Verify for errors
  //printf("Opening!\n");

  int r4 = machine->ReadRegister( 4 );
  char fileName [128] = {0};
  int c = 0, i = 0;
  do{
    machine->ReadMem(r4, 1, &c);
    r4++;
    fileName[i++] = c;
  }while(c != 0);

  int unixOpenFileId = open( fileName, O_RDWR );
  if( unixOpenFileId != -1 ){
    int nachosOpenFileId = currentThread->mytable->Open(unixOpenFileId);
    if(nachosOpenFileId != -1){
      ///currentThread->mytable->Print();
      machine->WriteRegister(2, nachosOpenFileId);
      returnFromSystemCall();   // Update the PC registers
      return;
    }
  }
  printf("Error: unable to open file\n");
  returnFromSystemCall();   // Update the PC registers
}// Nachos_Open
```

Figura 8: Implementación Open

- Read

```
void Nachos_Read(){
  //printf("Reading!\n");
  int r4 =  machine->ReadRegister(4); // pointer to Nachos Mem
  int size = machine->ReadRegister(5); // byte to read
  OpenFileId fileId = machine->ReadRegister(6); // file to read
  char bufferReader[size + 1] = {0}; // store unix result
  int readBytes = 0; // amount of read bytes
  int count = 0;

  // verify if file is one of standar output/input
  switch ( fileId ) {
    case ConsoleOutput:
    printf("%s\n", "Error, can not read from standard output");
    break;
    case ConsoleError:
    printf("%s\n", "Error, can not read from standard error");
    break;
    case ConsoleInput:
    //fgets( bufferReader, size , stdin );
    /*readBytes =  read( ConsoleInput,
    (void *)bufferReader, size );*/

    while (count < size )
    {
      std::cin>> bufferReader[count];
      ++count;
    }

    bufferReader[ size + 1 ]= '\0';
    readBytes = strlen( bufferReader );
    // write into Nachos mem
    for (int index = 0; index < readBytes; ++  index )
    {
      machine->WriteMem(r4, 1, bufferReader[index] );
      ++r4;
    }
    machine->WriteRegister(2, readBytes );
    break;
```

Figura 9: Implementación Read (parte 1)

```
    default:
    if ( currentThread->mytable->isOpened( fileId ) ) // if file is still opened
    {
      //  read using Unix system call
      readBytes =  read( currentThread->mytable->getUnixHandle(fileId),
      (void *)bufferReader, size );
      // write into Nachos mem
      for (int index = 0; index < readBytes; ++  index )
      {
        machine->WriteMem(r4, 1, bufferReader[index] );
        ++r4;
      }
      // return amount of read readBytes
      machine->WriteRegister(2, readBytes );
    }else // otherwise no chars read
    {
      printf("Error: unable to read file\n");
      machine->WriteRegister(2,-1);
    }
    break;
  }
  returnFromSystemCall();
}// Nachos Read
```

Figura 10: Implementación Read (parte 2)

- Write

```
void Nachos_Write() {                   // System call 7

  /* System call definition described to user
  void Write(
  char *buffer, // Register 4
  int size, // Register 5
  OpenFileId id // Register 6
);
*/
int r4 = machine->ReadRegister( 4 );
int size = machine->ReadRegister( 5 );  // Read size to write
char buffer[size+1] = {0};

int c = 0, i = 0;
do{
  machine->ReadMem(r4, 1, &c);
  r4++;
  buffer[i++] = c;
}while(i < size);

//printf("%s\n", buffer);

// buffer = Read data from address given by user;
OpenFileId id = machine->ReadRegister( 6 ); // Read file descriptor

// Need a semaphore to synchronize access to console
Console->P();
switch (id) {
  case  ConsoleInput: // User could not write to standard input
  machine->WriteRegister( 2, -1 );
  size = 0;
  break;
  case  ConsoleOutput:
  printf( "%s", buffer );
  break;
  case ConsoleError:  // This trick permits to write integers to console
  printf( "%d\n", machine->ReadRegister( 4 ) );
  size = 1;
  break;
```

Figura 11: Implementación Write (parte 1)

```
  default:  // All other opened files
  // Verify if the file is opened, if not return -1 in r2
  if(!currentThread->mytable->isOpened(id)){
    machine->WriteRegister( 2, -1 );
    size = 0;
    return;
  }
  // Get the unix handle from our table for open files
  int unixOpenFileId = currentThread->mytable->getUnixHandle(id);
  // Do the write to the already opened Unix file
  int charCounter = write(unixOpenFileId, buffer, size);
  // Return the number of chars written to user, via r2
  machine->WriteRegister( 2, charCounter );
  break;

}
// Update simulation stats, see details in Statistics class in machine/stats.cc
stats->numConsoleCharsWritten += size;
Console->V();
returnFromSystemCall();   // Update the PC registers

}// Nachos_Write
```

Figura 12: Implementación Write (parte 2)

- Close

```
void Nachos_Close(){
  /* Close the file, we're done reading and writing to it.
  void Close(OpenFileId id);*/
  //printf("Closing!\n");
  OpenFileId id = machine->ReadRegister( 4 );
  int unixOpenFileId = currentThread->mytable->getUnixHandle( id );
  int nachosResult = currentThread->mytable->Close(id);
  int unixResult = close( unixOpenFileId );
  if(nachosResult == -1 || unixResult == -1 ){
    printf("Error: unable to close file\n");
  }
  //currentThread->mytable->Print();
  returnFromSystemCall();   // Update the PC registers
}// Nachos_Close
```

Figura 13: Implementación Close

- Fork

```
void Nachos_Fork()
{
  DEBUG( 'u', "Entering Fork System call\n" );
  // We need to create a new kernel thread to execute the user thread
  Thread * newT = new Thread( "child to execute Fork code" );

  delete  newT->mySems;
  newT->mySems = currentThread->mySems;
  newT->mySems->addSem();


  // We need to share the Open File Table structure with this new child
  delete  newT->mytable;
  newT->mytable = currentThread->mytable;
  newT->mytable->addThread();

  // Child and father will also share the same address space, except for the stack
  // Text, init data and uninit data are shared, a new stack area must be created
  // for the new child
  // We suggest the use of a new constructor in AddrSpace class,
  // This new constructor will copy the shared segments (space variable) from currentThread, passed
  // as a parameter, and create a new stack for the new child
  newT->space = new AddrSpace( currentThread->space );

  // We (kernel)-Fork to a new method to execute the child code
  // Pass the user routine address, now in register 4, as a parameter
  // Note: in 64 bits register 4 need to be casted to (void *)
  newT->Fork( NachosForkThread, (void*)(long)(machine->ReadRegister( 4 ))); // ojo se elimino warning
  currentThread->Yield();
  returnFromSystemCall(); // This adjust the PrevPC, PC, and NextPC registers

  DEBUG( 'u', "Exiting Fork System call\n" );
} // Nachos_Fork
```

Figura 14: Implementación Fork

Implementación del método NachosForkThread necesario para el Fork.

```
void NachosForkThread( void * p ) { // for 64 bits version
  AddrSpace *space;
  long dir = (long) p;

  space = currentThread->space;
  space->InitRegisters();             // set the initial register values
  space->RestoreState();              // load page table register

  // Set the return address for this thread to the same as the main thread
  // This will lead this thread to call the exit system call and finish
  machine->WriteRegister( RetAddrReg, 4 );

  machine->WriteRegister( PCReg, dir );
  machine->WriteRegister( NextPCReg, dir + 4 );


  machine->Run();                     // jump to the user progam

  ASSERT(false);
}
```

Figura 15: Implementación NachosForkThread

- Yield

```
case SC_Yield:                  //System call # 10
currentThread->Yield();
returnFromSystemCall();
break;
```

Figura 16: Implementación Yield

- SemCreate

```
void Nachos_SemCreate()
{
  long initValue = machine->ReadRegister( 4 );
  //printf("Valor inicial: %ld\n", initValue );
  Semaphore* sem = new Semaphore("Sem usuario", initValue);
  if ( sem != NULL )
  {
    //printf("Se crea sem\n");
    int nachosSemIdentfier = currentThread->mySems->registerSem( (long)sem );
    //printf("Su identificador es : %d\n", nachosSemIdentfier );
    machine->WriteRegister( 2, nachosSemIdentfier );
  }else
  {
    // return invalid id for sem
    machine->WriteRegister( 2, -1 );
  }
}// Nachos_SemCreate
```

Figura 17: Implementación SemCreate

- SemDestroy

```
void Nachos_SemDestroy()
{
  int semId = machine->ReadRegister( 4 );
  //printf("id: %d\n", semId );
  long sem = currentThread->mySems->unRegisterSem( semId );
  //printf("Valor direccion %ld\n",sem);
  if ( sem != -1 )
  {
    Semaphore* s = (Semaphore*) sem;
    s->Destroy();
    machine->WriteRegister( 2, 0 );
  }else
  {
    // sem isnt destroied.
    machine->WriteRegister( 2, -1 );
  }
}// Nachos_SemDestroy
```

Figura 18: Implementación SemDestroy

- SemSignal

```
void Nachos_SemSignal()
{
  //printf("%s\n", "Nachos signal");
  int semId = machine->ReadRegister( 4 );
  //printf("id: %d\n", semId );
  long pointerToCast = currentThread->mySems->getNachosPointer( semId );
  //printf("Valor direccion %ld\n",pointerToCast);
  if ( pointerToCast != -1 )
  {
    Semaphore* sem = (Semaphore*)pointerToCast;
    //printf("%s%d\n","Hago signal valor semaforo: ", sem->getValue() );
    sem->V(); // then wait
    //printf("%s%d\n","Hago signal valor semaforo: ", sem->getValue() );
    machine->WriteRegister( 2, 0 );
  }else
  {
    printf("%s\n","No Hago signal" );
    machine->WriteRegister( 2, -1 );
  }
}// Nachos_SemSignal
```

Figura 19: Implementación SemSignal

- SemWait

```
void Nachos_SemWait()
{
  //printf("%s\n", "Nachos wait");
  int semId = machine->ReadRegister( 4 );
  //printf("id: %d\n", semId );
  long pointerToCast = currentThread->mySems->getNachosPointer( semId );
  //printf("Valor direccion %ld\n",pointerToCast);
  if ( pointerToCast !=  -1 )
  {
    //printf("%s\n","Hago wait" );
    Semaphore* sem = (Semaphore*)pointerToCast;
    sem->P(); // then wait
    machine->WriteRegister( 2, 0 );
  }else
  {
    printf("%s\n","NO hago wait" );
    machine->WriteRegister( 2, -1 );
  }
}// Nachos_SemWait
```

Figura 20: Implementación SemWait

# 5. Manual de usuario

- **Sistema Operativo:** [Linux]
- **Arquitectura:** [64 bits]
- **Ambiente:** [Consola (Shell)]

## Compilación

Para compilar el programa, se utiliza los $Makefiles$ que trae NachOS por defecto.
En la capeta userprog de NachOS se debe correr primero el comando:

```
make depend
```

Seguido se debe correr el archivo $Makefile$ creado luego de correr el ”make depend”, de la siguiente manera:

```
make
```

Finalmente se pueden probar las implementaciones de los diferentes system calls y del manejo de memoria, corriendo programas de ususario para NachOS de la siguiente manera:

```
./nachos -x ../test/<nombre del archivo ejecutable a correr>
```

Ojo para ejecutar todos los comandos anteriores de sebe estar en la carpeta userprog de NachOS.

## Especificación de las funciones del programa

La única especificación que cabe destacar es que el system call Read cuando se pide que lea de consola (0), este va a leer un número de caracteres especificado por el usuario y no hasta una condición tipo: hasta un cambio de linea o un fin de archivo(EOF).

# 6. Casos de Prueba

A continuación se desprenden algunos casos de prueba de los llamados al sistemas implementados.

1. Ejecución del test `copy.c`

```c
#include "syscall.h"

int
main()
{
    OpenFileId input;
    OpenFileId output;
    char buffer[1024];
    int n = 0;


    Create( "nachos.2" );
    input  = Open( "nachos.1" );
    output = Open( "nachos.2" );

    while( (n = Read( buffer, 1024, input ) ) > 0 ) {
        Write( buffer, n, output);
    }
    Close( input );
    Close( output );

    Exit( 0 );
}
```

Para la prueba de este test, es necesario crear los archivos nachos.1 y nachos.2 en la misma carpeta de `userprog`. Se espera la siguiente salida en la consola:

```
ubuntolab@ubuntolab-VirtualBox:~/Desktop/REPO_FINAL_NACHOS/OSProject1/nachos64/c
ode/userprog$ ./nachos -x ../test/copy
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 500, idle 0, system 190, user 310
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 8748
Paging: faults 0
Network I/O: packets received 0, sent 0
```

Figura 21: Salida test copy

2. Ejecución del test addrspacetest

```c
#include "syscall.h"

/* Este proceso sirve para probar que el programa cargue correctamente las
        p g i n a s  en  el  addrspace.
        Requiere  que  se  encuentre  implementado  el  system  call  Write()  y  el  system
        call  Exit()  (aunque   nicamente   porque  el  programa  lo  llama  al  final)

```

```
8           Se recomienda que las p ginas  f sicas en memoria se guarden en desorden
                (p.e.
9           p gina virtual 1 en p gina  f sica 2, p gina virtual 2 en p gina
                f sica 4,
10          etc.)
11
12          El programa crear un buffer de 1024 bytes (4 p ginas) y lo llena con
13          27 car cteres. Si el programa addrspace est  correctamente implementado
14          deber a escribir:
15
16  abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
        abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqr
17  stuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
        abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghi
18  jklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
        abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
19  abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
        abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqr
20  stuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
        abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghi
21  jklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
        abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
22  abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
        abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqr
23  stuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{
        abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghi
24  jklmnopqrstuvwxy
25  */
26
27  void main () {
28          int i = 0, j = 0;
29          char buffer[1024];
30
31          for (j = 0; j<1024;j++) {
32                  buffer[j]=(char)((j%27)+'a');
33          }
34
35
36          while (i<1) {
37                  Write(buffer,1024,1);
38                  i++;
39          }
40  }
```

La salida resultante en consola se muestra a continuación:

```
ubuntolab@ubuntolab-VirtualBox:~/Desktop/REPO_FINAL_NACHOS/OSProject1/nachos64/code/userprog$ .
/nachos -x ../test/addrspacetest
abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmn
opqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{a
bcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmno
pqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{ab
cdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnop
qrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abc
defghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopq
rstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcd
efghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqr
stuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcde
fghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyz{abcdefghijklmnopqrstuvwxyNo threads ready or r
unnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 30802, idle 0, system 30, user 30772
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 1024
Paging: faults 0
```

Figura 22: Salida test addrspace

3. Ejecución del test pingPong.

```
#include "syscall.h"

void SimpleThread(int);

int
main( int argc, char * argv[] ) {

    Fork(SimpleThread);
    SimpleThread(1);

    Write("Main  \n", 7, 1);
    Write(argc, 4, 1);
    Write(argv, 4, 1);
}


void SimpleThread(int num)
{

    if (num == 1) {
        for (num = 0; num < 5; num++) {
                Write("Hola 1\n", 7, 1);
                Yield();
        }
    }

    else {
        for (num = 0; num < 5; num++) {
                Write("Hola 2\n", 7, 1);
                Yield();
        }
    }
    Write("Fin de\n", 7, 1);
}
```

La captura de la corrida se muesta a continuación:

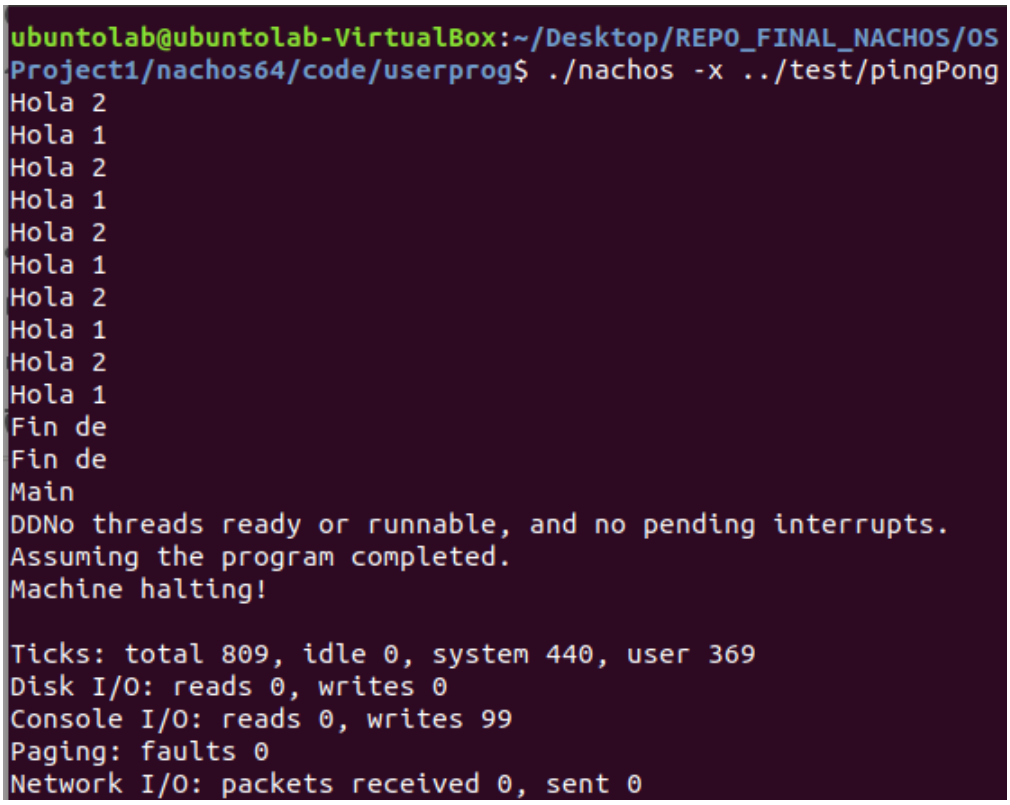


Figura 23: Salida test ping pong

4. Ejecución del test para el laboratorio 10

```
#include "syscall.h"

void hijo(int);
int id;
int main(){
        id = SemCreate(0);
        Fork(hijo);

        SemWait(id);
        Write("padre\n", 6, 1);
        SemDestroy(id);
        Exit(0);
}


void hijo(int dummy){
        Write( "hijo\n", 6, 1 );
        SemSignal(id);
}
```

La captura de la salida en consola se muestra a continuación

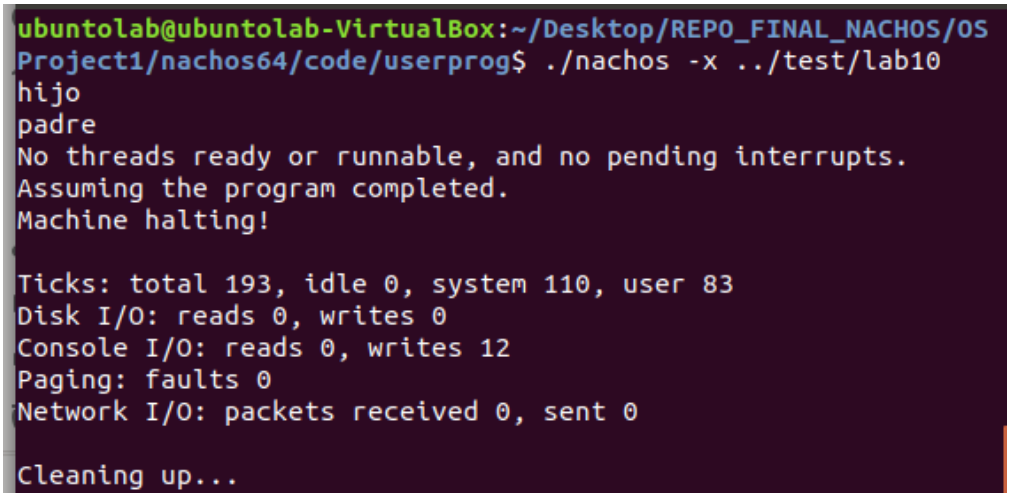


Figura 24: Salida test usado en el lab 10

5. Ejecución del test create

```
int main(){
        int fd;
char * buf;// = new int[6];
        Create("archivo.nuevo");
        fd = Open("archivo.nuevo");
        Write("prueba", 6, fd);
        Close(fd);
        Exec("../test/brillo");
        Exec("brillo1");

        //char* buf = new int[6];
        fd = Open("archivo.nuevo");
        Read(buf, 6, fd);
        Write(buf, 6, 1);

        Exit(0);
        fd = 1;
        fd++;
        return 0;
}
```

Contenidos del archivo brillo:

```
#include "syscall.h"

int main(){
    Write(" Brillo !", 9, 1);
        return 0;
}
```

Contenidos del archivo brillo1:

```
#include "syscall.h"

int main(){
    Write(" Brillo  1 / Oscuridad 0!", 25, 1);
        return 0;
}
```

A continuación se muestra la salida en consola:

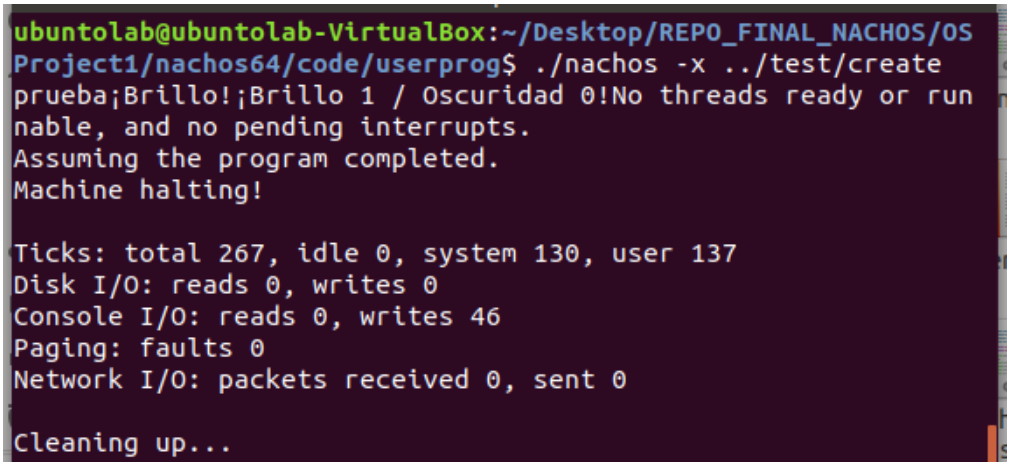


Figura 25: Salida test create

6. Ejecución de archivo todos

```
#include "syscall.h"

int resultado;
char Salida[10];

void Verifica();

int main() {

    SpaceId i=0;
    int a = 0;

    resultado = -1;
    Salida[0] = 'n';
    Salida[1] = 'a';
    Salida[2] = 'c';
    Salida[3] = 'h';
    Salida[4] = 'o';
    Salida[5] = 's';
    Salida[6] = '.';
    Salida[7] = '1';
```

```
22      Salida[8] = 0;
23      i = Exec( "../test/copy" );
24  //    Write( "Espera por el copy ...\n", 23, 1 );
25      a = Join( i );
26
27      Fork( Verifica );
28      Yield();
29      if ( resultado >= 0 )
30         Write("Bien!\n", 6, 1 );
31      else
32         Write( ":( \n", 4, 1 );
33  }
34
35  void Verifica() {
36
37      OpenFileId prueba;
38
39      Write( "Verificando ...\n", 16, 1 );
40      prueba = Open( Salida );
41      resultado = prueba;
42      Yield();
43      Close( prueba );
44  }
```

Finalmente, se desprende la captura de la corrida en consola:

```
ubuntolab@ubuntolab-VirtualBox:~/Desktop/REPO_FINAL_NACHOS/OS
Project1/nachos64/code/userprog$ ./nachos -x ../test/todos
Ultimo hilo borra tabla de archivos
Verificando ...
Bien!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 758, idle 0, system 310, user 448
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 8770
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

Figura 26: Salida del test que comprueba todos los llamados al sistema