Informe sobre NachOS (Not Another Completely Heuristic Operating System)

Galuppo, Raúl I.
Graf, Andrea

4 de agosto de 2017

Contenidos

Capítulo 1

Resolución de la Práctica N°1

1. *¿Cuánta memoria tiene la máquina simulada para **NachOS**?*
   La máquina simulada tiene 32 páginas de 128 bytes.

2. *¿Cómo cambiaría ese valor?*
   Para cambiar ese valor hay que modificar la cantidad de páginas y/o modificar el tamaño de cada una de ellas. Las respectivas variables son `NumPhysPages` y `PageSize`.

3. *¿De qué tamaño es un disco?*
   El disco simulado por ***NachOS*** tiene 131 MB. Su estructura interna consta de 32 pistas de 32 sectores de 128 bytes cada una.

4. *¿Cuántas instrucciones del **MIPS** simula **NachOS**?*
   NachOS puede simular hasta 63 instrucciones del ***MIPS*** (`MaxOpcode=63`) aunque sólo hay definidas 59 de ellas. El listado se encuentra en `mipssim.h`.

5. *Explicar el código que procesa la instrucción `add`*
   `OP_ADD` realiza la suma del contenido de dos registros cuyo valor son enteros con signo, `rs` y `rt`, y siempre que no ocurra ***overflow*** almacena el resultado en `rd`; en caso contrario lanza la excepción `OverflowException`.

```
case OP_ADD:
  sum = registers[(int)instr->rs] + registers[(int)instr->rt];
  if (!((registers[(int)instr->rs] ^ registers[(int)instr->rt])
             &  SIGN_BIT)
      && ( (registers[(int)instr->rs] ^ sum)
             & SIGN_BIT)) {

                  RaiseException(OverflowException, 0);
                  return;
  }

  registers[(int)instr->rd] = sum;
```

   Además, se utiliza la constante `SIGN_BIT`, definida en `mipssim.h`. Es una máscara cuyo valor es `0x80000000`. Nos sirve para saber que signo tiene un número.

Por otro lado, ***Overflow*** en una suma solamente ocurre cuando los operandos tienen el mismo signo y el resultado tiene distinto signo. Esto sucede porque el acarreo cambia el bit de signo. Voliendo a la condición del if:

```
~( sign1 XOR sign2) && (sign1 XOR signResult)
```

donde:

```
sign1 = num1 & SIGN_BIT      // Obtenemos el signo del operando1
sign2 = num2 & SIGN_BIT      // Obtenemos el signo del operando2
signResult = sum & SIGN_BIT  // Obtenemos el signo del resultado de la suma
```

6. *Nombrar los archivos fuente en los que figuran las funciones y métodos llamados por el* `main` *de* ***NachOS*** *al ejecutarlo en el directorio* `threads` *(hasta dos niveles de profundidad).*

Funciones llamadas por `main.cc`	Están definidas en
DEBUG	threads/utility.cc
ASSERT	threads/utility.h
Initialize	threads/system.cc
ThreadTest	threads/threadtest.cc
StartProcess	userprog/progtest.cc
ConsoleTest	userprog/progtest.cc
Halt	machine/interrupt.cc
Copy	filesys/fstest.cc
Print	filesys/fstest.cc
Remove	filesys/directory.cc
List	filesys/directory.cc
PerfonmanceTest	filesys/fstest.cc
Delay	machine/sysdep.cc
MailTest	network/nettest.cc
Finish	threads/thread.cc

7. *¿Porqué se prefiere emular una CPU en vez de utilizar directamente la CPU existente?*
   En primer lugar, si no se simulara ***NachOS***, se necesitaría una máquina con ***MIPS*** para correrlo; o que se implemente para otras arquitecturas(Intel 64, por ejemplo).
   En segundo lugar, si se utilizara una máquina real, si por error de programación un test genera un proceso que cae en deadlock, habría que reiniciar la máquina real. Al simular la CPU, podemos matar ese proceso y volver a testear. Lo mismo sucedería si un test escribe mal en el disco o en la memoria (Segmentation Fault, por ejemplo).
   Finalmente, se desea que el aprendizaje del alumno esté enfocado al SO y no al hardware, los dispositivos emulados resultan sencillos de entender y manejar. Y dado que ***NachOS*** se ejecuta como un proceso, se puede desarrollar software y depurarlo con absoluto control. Incluso podrían correr varios ***NachOS*** a la vez en la misma máquina.

8. *Comente el efecto de las distintas banderas de* `debug`.

Bandera	Efecto
`+`	muestra todos los mensajes de `debug`. Es decir, activa todas las banderas.
`t`	mensajes relacionados al manejo de threads.
`s`	mensajes relacionadas a la sincronización de threads.
`i`	mensajes relacionados con interrupciones.
`m`	mensajes relacionados a la máquina ***MIPS*** simulada por ***NachOS***.
`d`	mensajes relacionados al disco simulado por ***NachOS***.
`f`	mensajes relacionados al sistema de archivos.
`a`	mensajes relacionados con el manejo de direcciones.
`n`	mensajes relacionados a la red simulada por ***NachOS***.

Cuadro 1.1: Debugging flags.

9. *¿Qué efecto hacen las macros* `ASSERT` *y* `DEBUG` *definidas en* `utility.h`*?*

   ASSERT Recibe una condición. Si se reduce a `false`, imprime un mensaje y hace un `core dump`, indicando el archivo y la línea donde ocurrió el mismo. Sirve para documentar condiciones ciertas.

   DEBUG Imprime un mensaje de depuración sólo si las banderas están habilitadas(`./nachos -d`). Además, permite clasificar los mensajes, asignado a cada clase una bandera (ver Table 1.1).

10. *¿Dónde están definidas las constantes* `USER_PROGRAM`*,* `FILESYS_NEEDED`*,* `FILESYS_STUB` *y* `NETWORK`*?*

    Las constantes de pre-procesador mencionadas están definidas en los archivos `Makefile` de ciertos directorios, los cuales se mencionan en el Table 1.2.

Constante	Directorios
USER_PROGRAM	userprog, vm, network, filesys
FILESYS_NEEDED	userprog, vm, network, filesys
FILESYS_STUB	userprog, vm
NETWORK	network

Cuadro 1.2: Ubicación de las constantes de pre-procesamiento.

11. *¿Cuál es la diferencia entre las clases* `List` *y* `SynchList`*?*
    La principal diferencia es que la segunda clase ofrece estructuras de datos que permiten un acceso sincronizado a las listas (sus métodos son *thread safety*). Mantiene los siguientes invariantes:

    a) Si un `thread` intenta remover un elemento de la lista, va a esperar a que dicha lista tenga al menos un elemento.

    b) Un `thread` a la vez puede acceder a la lista.

12. *¿En qué archivos está definida la función `main`? ¿En qué archivo está definida la función `main` del ejecutable **NachOS** del directorio `userprog`?*
    La función `main` está definida en: test/matmult.c, test/halt.c, test/sort.c, test/shell.c, bin/-coff2noff.c, bin/coff2flat.c, bin/main.c, bin/out.c, bin/disasm.c y en threads/main.cc.
    La función `main` del ejecutable ***NachOS*** del directorio `userprog` está definida en el archivo `threads/main.cc`.

13. *¿Qué línea de comandos soporta **NachOS**? ¿Qué efecto hace la opción `-rs`?* Signatura de ***NachOS***, definida en el archivo `main.cc`.

```
nachos -d <debugflags> -rs <random seed #>
       -s -x <nachos file> -c <consoleIn> <consoleOut>
       -f -cp <unix file> <nachos file>
       -p <nachos file> -r <nachos file> -l -D -t
       -n <network reliability> -m <machine id>
       -o <other machine id>
       -z
```

    El comando `-rs` produce que se llame al método `Yield` aleatoriamente dentro de la ejecución. La función de dicho método es detener la ejecución del `thread` actual y selecciona uno nuevo de la lista de `thread` listos para ejecutar, si no hay ninguno otro listo, sigue ejecutando el `thread` actual. En otras palabras, el `Scheduller` pasa a ser preemtivo.

14. *Modificar el ejemplo del directorio threads para que se generen 5 threads en lugar de 2.*
    Se modificó el archivo `threadtest.cc` para que contenga varios tests. Cuando se llama a `ThreadTest`, se ejecutan una serie de test, cada uno de ellos separados en subrutinas. La modificación que se pide se encuentra en el método `SimpleTest`.

15. *Modificar el ejemplo para que estos cinco hilos utilicen un semáforo inicializado en 3. Esto sólo debe ocurrir si se define la macro de compilación SEMAPHORE_TEST.*
    Vale la misma aclaración que en la pregunta anterior. En esta ocasión, la modificación se encuentra en el método `SemaphoreTest`.

16. *Agregar al ejemplo anterior una linea de debug donde diga cuando cada hilo hace un P() y cuando un V(). La salida debe verse por pantalla solamente si se activa ESA bandera de debug.*
    Se modificó el archivo `synch.cc` para agregar las lineas de debug. Se utilizo la bandera `'S'`.

Capítulo 2

Resolución de la Práctica N°2: Sincronización de hilos en NachOS

2.1. Implementación de cerrojo y variables de condición

Cerrojo

Marco teórico

Un cerrojo provee una exclusión mutua para utilizar ciertos recursos. Puede tener dos estados: libre y ocupado. Cabe aclarar que nadie excepto el hilo que tiene adquirido el cerrojo puede liberarlo. Además, sólo se permiten dos operaciones:

- **Acquire :** Si el cerrojo está libre, lo obtiene y lo marca como ocupado.
- **Realese :** Libera el cerrojo.

```
class Lock {
  public:
        // Constructor: inicia el cerrojo como libre
        Lock(const char* debugName);
        ~Lock();          // destructor
        const char* getName() { return name; }  // para depuracion

        // Operaciones sobre el cerrojo. Ambas deben ser *atomicas*
        void Acquire();
        void Release();

        // devuelve 'true' si el hilo actual es quien posee
        // el cerrojo. Util para comprobaciones en el Release()
        // y en las variables condicion.
        bool isHeldByCurrentThread();
  private:
    Thread *blocker;    // thread que adquirio el cerrojo.
    const char* name;   // nombre del cerrojo.
    Semaphore *semLock; // Semaforo para aislar la zona compartida
};
```

Implementación

Para implementar los métodos, lo primero que se hizo es el método `Lock::isHeldByCurrentThread`

```
bool Lock::isHeldByCurrentThread(){
    return (blocker == currentThread);
}
```

En el constructor, se crea un semáforo `semLock` cuyo valor inicial es `1`.
Para implementar `Lock::Acquire` se realizaron los siguientes pasos:

1. Preguntar si el cerrojo esta libre, utilizando `isHeldByCurrentThread`.
2. Decremento el semáforo `semLock`.
3. Asignar `currentThread` a la variable `blocker`, para indicar que dicho thread obtuvo el cerrojo.

En cambio, para implementar `Lock::Release` se realizaron los siguientes pasos:

1. Hay que asegurar, mediante un ASSERT, que el thread actual es el que había obtenido el cerrojo. Para ello, se tiene que cumplir `isHeldByCurrentThread`.
2. Incrementar el semáforo `semLock`.
3. Liberar la variable `blocker`, para indicar que se liberó el cerrojo.

Variables de condición

Marco teórico

Las variables de condición usadas en conjunto con cerrojo permiten a un hilo esperar por la ocurrencia de una condición arbitraria. Se utilizan para encolar hilos que esperan (`Wait`) a que otro hilo les avise (`Signal`) cuando se cumpla dicha condición.
Se definen tres operaciones sobre una variable condición:

- **Wait:** Libera el cerrojo y expulsa al hilo de la CPU. El hilo se espera hasta que alguien le hace un `Signal`.
- **Signal:** Si hay alguien esperando en la variable, despierta a uno de los hilos. Si no hay nadie esperando, no ocurre nada.
- **Broadcast:** Despierta a todos los hilos que están esperando.

Todas las operaciones sobre una variable condición deben ser realizadas adquiriendo previamente el cerrojo. Esto significa que las operaciones sobre variables condición han de ejecutarse en exclusión mutua.

```
class Condition {
  public:
    // Constructor: se le indica cual es el cerrojo al que pertenece
    // la variable condicion
        Condition(const char* debugName, Lock* conditionLock);

    // libera el objeto
    ~Condition();
    const char* getName() { return (name); };

    // Las tres operaciones sobre variables condicion.
    // El hilo que invoque a cualquiera de estas operaciones debe
    // tener adquirido el cerrojo correspondiente; de lo contrario
    // se debe producir un error.
    void Wait();
    void Signal();
    void Broadcast();

  private:
    const char* name; // Nombre de la v.c.
    Lock* lock; // Cerrojo vinculado a la v.c.
    List<Semaphore*> * semList; //Lista de semaforos correspondiente
                     // a los hilos que se fueron bloqueando.
};
```

Detalles de la implementación

En el constructor, se inicializa a `semList` con una lista vacía; se guardan el nombre del semáforo y el cerrojo vinculado, en las variables correspondientes.
Para implementar `Condition::Wait` se realizaron los siguientes pasos:

1. Hay que asegurar, mediante un ASSERT, que el thread actual es el que había obtenido el cerrojo. Para ello, se tiene que cumplir `isHeldByCurrentThread`.
2. Se crea un semáforo llamado `sem`, cuyo valor inicial es `0`.
3. Se lo agrega al final de la lista `semList`.
4. Se libera el `lock`.
5. Se decrementa el semáforo `sem`, logrando que se bloquee el thread que esta ejecutándose (`currentThread`).
6. Se obtiene el `lock`.

Para implementar `Condition::Signal` se realizaron los siguientes pasos:

1. Hay que asegurar, mediante un `ASSERT`, que el thread actual es el que había obtenido el cerrojo. Para ello, se tiene que cumplir `isHeldByCurrentThread`.
2. Si hay algún semáforo en la lista `semList`, se quita el primero y se lo incrementa. De esta forma, logramos que se libere el thread bloqueado, para que se agregue a la lista de threads listos para ejecutarse.

Finalmente, para implementar `Condition::Broadcast` se realizaron los siguientes pasos:

1. Hay que asegurar, mediante un ASSERT, que el thread actual es el que había obtenido el cerrojo. Para ello, se tiene que cumplir `isHeldByCurrentThread`.

2. Mientras haya algún semáforo en la lista `semList`, se invoca a `Signal`. De esta forma, logramos que se liberen todos los threads que se encontraban bloqueados.

2.2. Implementación de puertos

Introducción

La siguiente clase implementa un sistema de mensajes entre hilos, el cual permite que los emisores se sincronicen con los receptores mediante un buffer.
Se definen dos operaciones: `Send` y `Receive`. La primera espera a que esté libre el buffer e inserta un mensaje en él. La segunda, se encarga de la recepción de un mensaje, bloqueándose mientras no llegue ninguno.

```
class Puerto {
    public:
        Puerto(const char* name);
        ~Puerto();
        const char* getName() { return portname; } //Para depuracion

        void Send(int message);
        void Receive(int* postbox);
    private:
        const char* portname; // nombre del puerto.
        Lock* lockPort;         // cerrojo asociado a las v.c.
        Condition* sending; // v.c. que contralan el acceso
        Condition* receiving; // sincronizado al buffer.
        int  buffer;    // buffer de comunicacion
        bool access;    // indica si el buffer esta libre.
}
```

Detalles de la implementación

En el constructor, se crean el cerrojo y las variables de condición. Además, se asigna el nombre del puerto y se indica que el buffer está libre.
Para implementar `Puerto::Send`, se realizaron los siguientes pasos:

1. Se obtiene el cerrojo `lockPort`.

2. Mientras el buffer esté ocupado, se invoca al método `Condition::Wait`, correspondiente a la v.c *sending*.

3. Se guarda el mensaje recibido en `buffer`.

4. Se indica que el buffer está ocupado, seteando `access` en `false`.

5. Se libera el cerrojo `lockPort`.

Finalmente, para implementar `Puerto::Receive`, se realizaron los siguientes pasos:

1. Se obtiene el cerrojo `lockPort`.
2. Mientras el buffer esté libre, se invoca al método `Condition::Wait`, correspondiente a la v.c `receiving`.
3. Se copia `buffer` en el argumento.
4. Se indica que el buffer está libre, seteando `access` en `true`.
5. Se libera el cerrojo `lockPort`.

2.3. Implementación del método *Thread* :: *Join*

El método `Thread::Join` permite al thread que la invoca esperar la finalización de otro proceso antes de continuar con su ejecución.
Para implementarlo, se agregó lo siguiente a la clase `Thread`:

```
class Thread {
public:
        ...
        void Join(); //  Bloquea al llamante hasta que el hilo en
           cuestion termine.
private:
        int joinFlag; // Indica si se tiene que realizar el join.
        Puerto* joinPort; // Sirve para sincronizar los dos threads.
}
```

En el constructor, se inicializaron las variables agregadas; en el destructor, se libera el miembro `joinPort`.
Luego, se implementó el método `Thread::Join`:

```
void
Thread::Join()
{
    if(joinFlag){
        int * msg = (int*) malloc (sizeof(int));
        joinPort->Receive(msg);
    }
}
```

Además, se modificaron los siguientes métodos:

- `Thread::Fork` Se le agregó un argumento para setear la bandera `joinFlag` al momento de crear un hijo. Si el parámetro viene con un número mayor a cero, significa que el padre espera a que este nuevo hijo termine.
- `Thread::Finish` Si está activada la bandera `joinFlag`, se envía un mensaje a través del puerto `joinPort`, indicando de esta forma que finalizo. Por lo tanto, se puede desbloquear al padre que espera.

Finalmente, mostraremos la secuencia de ejecución para utilizar esta característica:

1. El thread `Padre` ejecuta unas tareas.
2. `Padre` crea un thread `Hijo` indicando que quiere hacer `join`:

```
Thread *hijo = new Thread("Hijo");
hijo->Fork(childFunction, (*void) 1, 1); //Al ser el
    3er argumento mayor que 0, indica que quiero
   hacer join.
```

3. El padre se queda esperando a que el hijo termine mediante:

```
hijo->Join();
```

4. Cuando el thread `Hijo` termina, su padre continua su ejecución.

2.4. Implementación de multicolas con prioridad

Implementación de la propiedad prioridad en los Threads

A cada proceso se le determinará, mediante un criterio, su prioridad. Para ello, se agregó una propiedad a los Threads llamada `priority`, que indica cuál es la prioridad de dicho Thread.
Todos los threads tienen una prioridad positiva, siendo 0 la prioridad de menor importancia.
Se realizaron los siguientes cambios en el archivo `thread.h`:

```
public:
  ...
  int getPriority(){ return priority; } //Retorna la prioridad.
  void setPriority(int p){ priority = p; } // Asigna la prioridad.
private:
  ...
  int priority; // Indica la prioridad.
```

Al constructor de `Threads` se le agregó un parámetro para indicar la prioridad de un thread.

Modificaciones al Scheduler para que utilice Multicolas

Esquema de planificación Múltiples colas de prioridad
Se definen la cantidad de colas que se encuentran en el sistema. Su número determina cuántas prioridades distintas se tienen para clasificar los procesos.
Cuando un proceso tenga su estado en listo, será colocado en la cola correspondiente a su prioridad.
Una aclaración importante es que cada cola puede manejar un algoritmo de planificación diferente a las demás.
Se modificó el archivo `scheduler.h` de la siguiente manera:

```
class Scheduler {
    public:
    ...

    private:
        List<Thread*> **readyList;  // Priority queue of threads
                                    // that are ready to run,
                                    // but not running
        int max_priority;    // Cantidad de colas.
}
```

Implementación

Para desarrollar las colas de prioridad múltiple, se modificó `scheduler.cc`:

- **Constructor de la clase** Se indica la cantidad de colas. Además, se inicializan las colas.
- **Destructor de la clase** Se eliminan las colas.
- `Scheduler::ReadyToRun` Agrega al thread al final de la cola correspondiente a su prioridad.
- `Scheduler::FindNextToRun` Empezando por las colas de mayor prioridad, obtiene un thread para ejecutarse. Si la cola superior está vacía, sigue con la cola de prioridad inmediatamente menor. Y así sucesivamente hasta encontrar un thread.

Inversión de prioridades

Descripción del problema

Sean dos procesos A,B que comparten un recurso. Supongamos que se da la siguiente situación:

- El proceso A, de prioridad baja, pide el recurso compartido y es interrumpido luego de obtenerlo.
- El proceso B, de prioridad alta, pide el recurso compartido.

Esta situación hace que queden invertidas las prioridades relativas entre ambos, ya que el proceso B tiene que esperar a que el proceso A libere el recurso compartido. Como consecuencia, el proceso de mayor prioridad se ejecutará luego del proceso de menor prioridad.

Detalles de la implementación de la solución

Para solucionar el problema de inversión de prioridades, aplicamos una técnica llamada *Herencia de Prioridades*: dados dos procesos que comparten un recurso, tales que tengan distintas prioridades. Supongamos que el proceso de menor prioridad esté bloqueando el recurso compartido. Entonces procedemos a cambiar la prioridad de dicho proceso, asignándole la prioridad del otro.
En ***NachOS***, este problema puede ocurrir cuando dos threads comparten un cerrojo o una variable de condición. Para solucionar este problema, se modificaron las siguientes clases:

- `synch.h` y `synch.cc`: Se agregó un cerrojo llamado `invPrController`, cuya función es controlar el cambio de prioridades.

- `scheduler.h` y `scheduler.cc`: Se agregó un método llamado `Scheduller::ChangePriorityQueue` cuya función es cambiar de cola a un thread. Para ello, recibe dos argumentos: un thread y una prioridad.

- `Lock::Acquire`: Se modificó este método de modo tal que si un thread `T2` quiere adquirir el cerrojo pero éste está en manos de otro thread `T1` cuya prioridad es menor que la de `T2`, se proceda a asignarle a `T1` la misma prioridad que `T2`. Para ello, se invoca al método `Scheduller::ChangePriorityQueue`.

¿Por qué no se puede aplicar herencia de prioridades en semáforos?
Notemos que la clase `Semaphore` guarda una cola con los threads bloqueados pero se olvida del thread que lo bloquea. Por lo tanto, no sabemos a cuál thread cambiar la prioridad.

Capítulo 3

Resolución de la Práctica N°3: Programas de usuario

3.1. Desarrollo del API para copiar datos desde el núcleo al espacio de memoria del usuario y viceversa

La interfaz `usertranslate.h` contiene las funciones que integran la API para traducir `Strings` y `Arrays`. La API contiene las siguientes funciones:

```
// Lee una string desde la memoria de usuario.
void readStrFromUsr(int usrAddr, char *outStr);
// Lee un array desde la memoria del usuario.
void readBuffFromUsr(int usrAddr, char *outBuff, int byteCount);
// Traduce una string hacia la memoria de usuario.
void writeStrToUsr(char *str, int usrAddr);
// Traduce un array hacia la memoria de usuario.
void writeBuffToUsr(char *str, int usrAddr, int byteCount);
```

Para implementarlas, se utilizaron los métodos `Machine::ReadMemory` y `Machine::WriteMemory`.

```
void readStrFromUsr(int usrAddr, char *outStr) {
    int value, count = 0;
    bool done = machine->ReadMem(usrAddr, 1, &value);

    if(!done)
        ASSERT(machine->ReadMem(usrAddr, 1, &value));

    while((char) value != '\0') {
        outStr[count] = (char) value;
        count++;
        if( !machine->ReadMem(usrAddr + count, 1, &value))
            ASSERT(machine->ReadMem(usrAddr + count, 1, &value));
    }
    outStr[count] = '\0';
}
```

```
void
readBuffFromUsr(int usrAddr, char *outBuff, int byteCount) {
    int value;
    bool done;

    for(int i=0; i < byteCount; i++) {
        done = machine->ReadMem(usrAddr + i, 1, &value);
        if(!done)
            ASSERT(machine->ReadMem(usrAddr + i, 1, &value));
        outBuff[i] = (char) value;
    }
}
```

```
void
writeStrToUsr(char *str, int usrAddr) {
    bool done;

    while(*str != '\0') {
        done = machine->WriteMem(usrAddr, 1, *(str));
        if(!done)
            ASSERT(machine->WriteMem(usrAddr, 1, *(str)));
        usrAddr++;
        str++;
    }
}
```

```
void
writeBuffToUsr(char *str, int usrAddr, int byteCount) {
    bool done;

    for(int i=0; i < byteCount; i++) {
        done = machine->WriteMem(usrAddr + i, 1, (int) str[i]);
        if(!done)
            ASSERT(machine->WriteMem(usrAddr + i, 1, (int) str[i]));
    }
}
```

3.2. Implementación de las llamadas de sistema y la administración de interrupciones

Los programas de usuario invocan llamadas al sistema ejecutando la instrucción `syscall` de ***MIPS***, la cuál genera una trampa de hardware en el kernel de ***NachOS***. El simulador ***NachOS/MIPS*** implementa trampas invocando al método `RaiseException`, pasándole argumentos que indican la causa exacta de la trampa. `RaiseException`, a su vez, llama a `ExceptionHandler` para que se ocupe del problema específico.

Por convención, los programas de usuario colocan el código que indica la llamada de sistema deseada en el registro `R2` antes de ejecutar la instrucción `syscall`. Mientras que los argumentos adicionales se encuentran en los registros `R4 a R7`. Se espera que los valores de retorno de la función (y de la llamada del sistema) estén en el registro `R2` al regresar.

Para poder implementarla las llamadas de sistema, se crearon las siguientes funciones y estructuras auxiliares:

- `type`: en esta variable se guarda el tipo de llamada.
- `arguments`: aquí se guardan las direcciones de los argumentos.
- `result`: en esta variable se guarda el resultado de procesar la llamada. Luego, se escribirá dicho valor en el registro `R2`.
- `movingPC`: este método actualiza el PC para mantener el correcto funcionamiento del stack de registros. Se invoca al finalizar el procesamiento de una llamada a sistema.

```
void
movingPC()
{
      int pc = machine->ReadRegister(PCReg);
      machine->WriteRegister(PrevPCReg, pc);
      pc = machine->ReadRegister(NextPCReg);
      machine->WriteRegister(PCReg, pc);
      pc += 4;
      machine->WriteRegister(NextPCReg, pc);
}
```

El procesamiento de las llamadas a sistema quedó de la siguiente manera:

```
void
ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2),
        arguments[4],
        result;
    OpenFile* file;
    char name386[128];

    arguments[0] = machine->ReadRegister(4);
    arguments[1] = machine->ReadRegister(5);
    arguments[2] = machine->ReadRegister(6);
    arguments[3] = machine->ReadRegister(7);

    if (which == SyscallException) {
        switch(type) {
            case SC_Halt:
                ...
                break;
            case SC_Create:
                break;
            case SC_Exit:
                break;
            case SC_Exec:
                break;
```

```
26              case SC_Join:
27                  break;
28              case SC_Open:
29                  break;
30              case SC_Read:
31                  break;
32              case SC_Write:
33                  break;
34              case SC_Close:
35                  break;
36              default:
37                  printf("Unexpected syscall exception %d %d\n",
38                          which, type);
39                  ASSERT(false);
40          }
41          machine->WriteRegister(2, result);
42          movingPC();
43      } else {
44          DEBUG('e', "Is not a SyscallException\n");
45          printf("Unexpected exception:\t which=%s  type=%d\n",
46                  exception, type);
47          ASSERT(false);
48      }
49    }
```

En las próximas subsecciones se muestra cómo se desarrollaron las llamadas de sistema. Las mismas fueron implementadas en el orden propuesto por la cátedra.

Consola sincrónica

Se crea la clase `SynchConsole`, la cual provee una abstracción de acceso sincronizado a la consola. Un requisito que cumple es que un hilo queriendo escribir no bloquea a un hilo queriendo leer.

```
1  class SynchConsole {
2  public:
3          SynchConsole(const char *readFile, const char *writeFile);
4          ~SynchConsole();
5      void WriteConsole(char c); // Write a char into console.
6      char ReadConsole(); // Read a char from console.
7      void RequestWrite();    // Provides sync access for writing
8      void RequestRead();     // Provides sync access for reading
9
10 private:
11     Console *console;   // NacOS console.
12     Semaphore *readAvail,   // For reader console handler.
13               *writeDone;   // For writer console handler.
14     Lock *writer,   // For sync writing access.
15          *reader;   // For sync reading access.
16 };
```

Para desarrollarla, se tuvieron en cuenta las clases `progtest.cc` y `synchdisc.cc`.
Los semáforos `cread` y `cwrite`, permiten el acceso sincronizado a la lectura y escritura de la consola.
Cuando se levanta el hilo principal de ***NachOS***, se crea una consola de acceso sincronizado. Ademas, se reservan dos descriptores para representar la entrada y la salida estándar. Dichos descriptores son el `0` y el `1`. En `syscall.h` se definen dos constantes que modelan lo mencionado anteriormente:

```
#define CONSOLE_INPUT   0
#define CONSOLE_OUTPUT  1
```

Create

Dado un nombre de archivo, su función es crearlo.

```
void Create(char *name);
```

Recibe un parámetro, el cual es la dirección de memoria donde se aloja un string para indicar el nombre del archivo. Lo traduce mediante `readStrFromUsr` e invoca a `FileSystem::Create`.
A través de `R2`, devuelve `0` en caso de crear el archivo; sino devuelve `-1`.

Read

Dado un descriptor de un archivo `id`, lee `size` bytes y los guarda en el array `buffer`. Devuelve la cantidad de bytes leídos.

```
int Read(char *buffer, int size, OpenFileId id);
```

Si el descriptor es `CONSOLE_INPUT`, lee de la consola `size` caracteres o hasta que aparezca el caracter de salto de linea('\n'). Para leerlos, se invoca a `synchConsole->ReadConsole`.
En caso de que el descriptor corresponde a un archivo previamente abierto por el usuario, se leen de dicho archivo `size` caracteres. Para leer desde un archivo, se invoca a `OpenFile::Read`.
En cambio, si el descriptor `id` no corresponde a ningún archivo abierto por el usuario, se imprime un mensaje de error y devuelve `-1`.
Una vez que es leído exitosamente, se invoca a la función `writeBuffToUsr` para traducir el resultado de la lectura a una dirección en el espacio de direcciones indicado por el usuario en el primer argumento(`R4`).

Write

Dado un array `buffer`, escribe `size` bytes en el archivo cuyo descriptor es `id`. Devuelve la cantidad de bytes leídos.

```
void Write(char *buffer, int size, OpenFileId id);
```

Es similar a la llamada SC_Read. Se puede escribir en la consola sincrónica o en un archivo.
Al principio, se lee una dirección del espacio de memoria del usuario, la cual contiene la información a escribir. Para ello, se traduce el primer argumento invocando a readBuffFromUsr y guardándolo en la variable buffer.
Si el descriptor es CONSOLE_OUTPUT, escribe en la consola size caracteres. Para ello, invoca a synchConsole->WriteConsole.
En caso de que el descriptor corresponde a un archivo previamente abierto por el usuario, se escribe en dicho archivo size caracteres, almacenados en buffer. Para escribir en un archivo, se invoca a OpenFile::Write.
En cambio, si el descriptor id no corresponde a ningún archivo abierto por el usuario, se imprime un mensaje de error y devuelve -1.

Open

Dado un nombre de un archivo, lo abre y retorna su descriptor.

```
OpenFileId Open(char *name);
```

Al principio, se traduce el primer argumento invocando a readStrFromUsr. A continuación, se invoca a OpenFile::Open para abrir un archivo. Si la apertura es exitosa, se le indica al currentThread que abrió dicho archivo, mediante la invocación a Thread::AddFile. Luego, se retorna el descriptor recibido. Si falla la apertura, se retorna -1.
Por otro lado, cabe aclarar que una restricción que tiene un programa al manejar archivos, es la cantidad máxima de archivos que puede tener abiertos al mismo tiempo. Para modelar ello, se definió la siguiente constante en threads.h:

```
#define MAX_FILES_OPENED 5
```

Otra restricción que tiene un programa al manejar archivos es que sólo puede leer o escribir los archivos abiertos por él. Para modelar esto, se creo en threads.h una estructura de control para llevar un registro de los archivos abiertos por cada programa usuario.

```
class Thread {
private:
...
 // Structure for maintain all files opened for this user program.
    OpenFile* filesDescriptors[MAX_FILES_OPENED];

public:
...
// Given a descriptor, returns the corresponding file.
    OpenFile* GetFile(OpenFileId descriptor);
     // Loads a file, assing an descriptor and return its.
    OpenFileId AddFile(OpenFile* file);
     // Given a descriptor, unload its corresponding file.
     void RemoveFile(OpenFileId descriptor);
```

Dicha estructura consta de un arreglo de archivos llamado `filesDescriptors`, de tamaño `MAX_FILES_OPENDED` y tres métodos para interactuar con dicha estructura:

Thread::AddFile Dado un archivo, lo guarda en el arreglo y devuelve un `OpenFileId`. Guardamos cada archivo en el indice del arreglo, correspondiente a su descriptor. Para buscar un descriptor nuevo, basta con localizar un lugar vacío en el arreglo y devolver su posición. Se reservan el `0` y el `1` para la entrada y la salida estándar. Si no hay espacio libre, retorna `-1`.

Thread::GetFile Dado un descriptor de un archivo, retorna el archivo correspondiente. Para ello, busca en el arreglo. Si no lo encuentra, devuelve `NULL`.

Thread::RemoveFile Dado un descriptor de un archivo, limpia la posición dada.

Close

Dado un descriptor de un archivo, lo cierra.

```
1 void Close(OpenFileId id);
```

Recibe como argumento el descriptor del archivo a cerrar. Con él, obtiene el archivo invocando `Thread::GetFile`. En caso de obtener un archivo, significa que fue abierto por el usuario. Entonces lo elimina y lo quita. Para ello, invoca a `OpenFile:: OpenFile` y `Thread::RemoveFile`, respectivamente. Luego retorna `0`.
En caso contrario, imprime el error en pantalla y retorna `-1`.

Test para manejo de archivos

`testConsole.c`: Lee de la consola caracteres hasta llegar a 255 o hasta presionar `ENTER`. Luego, imprime los caracteres leídos en la consola.

`writetest.c`: Crea un archivo llamado `writetest.txt`. Lo abre, escribe un mensaje dentro y lo cierra.

`readtest.c`: Primero abre el archivo `readtest.txt`. Lee los primeros 100 caracteres y los imprime en la consola. Luego cierra el archivo.

Exit

Finaliza el programa según el estado dado.

```
1 void Exit(int status);
```

Si el argumento es `0`, termina correctamente. Sino imprime un cartel advirtiendo que el programa tuvo problemas.

Join

Dado un id de un proceso, espera a que termine y luego continua su ejecución.

```
1 int Join(SpaceId id);
```

Para poder esperar a un proceso, se tiene que llevar cuenta de los procesos que se están ejecutando. Por ello, creamos la clase `ProcessTable` que abstrae la relación mencionada anteriormente.

```
class ProcessTable{
  public:
    ProcessTable();
    ~ProcessTable();    // De-allocate an process table

    void addProcess(SpaceId pid, Thread* executor); // Add process
       at the table.
    int getFreshSlot(); // Returns an empty slot of the table.
    Thread* getProcess(SpaceId pid); // Given a pid, returns the
       appropiate process.
    void freeSlot(SpaceId pid);         // Delete process from the
       table.

  private:
    Thread** table;
};
```

Se agregó una variable llamada `processTable` en `system.h`. La tabla se crear cuando se inicializa el sistema. Ademas, se agregó una cota máxima de procesos que se ejecutan al mismo tiempo. Esto define la cantidad de entradas que tiene la tabla y se encuentra en `processtable.h`.

```
#define MAX_EXEC_THREADS 20
```

Todos los procesos que se ejecutan en modo usuario, tienen que estar registrados en dicha tabla. En particular, el proceso 'Main' también tiene que registrarse. Por ello, se agrega a la tabla momentos después de crearse, en el método `Inicialize`.
Volviendo a la implementación de la llamada a sistema `Join`, utilizando el argumento se invoca a `ProcessTable::getProcess` para obtener el thread que ejecutó dicho id.
Si es distinto de `NULL`, le hace Join invocando a `Thread::Join`. Caso contrario, imprime un mensaje de error y retorna `-1`.

Exec sin argumentos

Dado un archivo ejecutable, lo ejecuta. Para ejecutar un programa sin argumentos, se lo invoca con `0` y `'\0'` en el segundo y el tercer parámetro, respectivamente.

```
SpaceId Exec(char *name, int argc, char** argv);
```

Inicialmente, traducimos el nombre del archivo mediante `readStrFromUsr` e intentamos abrirlo invocando `OpenFile::Open`. Si falla, imprimimos un mensaje de error y retornamos el valor `-1`.
En caso de poder abrir el archivo, obtenemos un id para el nuevo proceso. Para ello, invocamos a `ProcessTable::GetFreshSlot`. Luego, se crea un thread para ejecutar el archivo y se asigna un espacio de direcciones y se carga el archivo ejecutable en memoria. Todo ello esta modularizado en la función `makeProcess`.

```
void
makeProcess(int pid, OpenFile* executable, char* filename, int argc,
    char** argv)
{
    DEBUG ('e',"C: currenThread:%s\t pid %d\t filename=%s\n",
       currentThread->getName(),
           pid, filename);

    Thread *execThread = new Thread(filename, 0);    //Creation of
       thread executor.
    execThread->setThreadId(pid);
    processTable->addProcess(pid, execThread);

    // Creation of space address for process.
    AddrSpace *execSpace = new AddrSpace(executable, argc, argv);
    execThread->space = execSpace;
    delete executable;

    amountThread++;
    execThread->Fork(doExecution, (void*) filename, 1);   //Create
       process.

    DEBUG('e', "After finish makeProcess, pid=%d\t name=$s\n",
       execThread->getThreadId(),
          execThread->getName());
}
```

Cuando el **Scheduler** ejecute el thread `execThread`, va inicializar su espacio de memoria y va a llamar a la maquina MIPS de `NachOS`, como vemos en el cuerpo de la función `doExecution`.

```
doExecution(void* arg)
{
    DEBUG ('e',"doExecution: currentThread name=%s\t id=%d\n",
        currentThread->getName(), currentThread->getThreadId());

  //Inicialization for MIPS registers.
        currentThread->space->InitRegisters();

  //Load page table register.
        currentThread->space->RestoreState();

        machine->Run();

    ASSERT(false);  //Machine->Run never returns.
}
```

3.3. Multiprogramación con rebanadas de tiempo (time-slicing)

Múltiples programas de usuario

Para soportar la multiprogramación, proponemos utilizar una estructura llamada **Bitmap** (definida en `bitmap.h`) para administrar los marcos de la memoria física; de tal forma que podamos saber cuales están asignados y cuales están libres. Dicho mapa de bits va a ser una variable global llamada `memoryMap` y se define en `system.cc`, pasándole como argumento el numero de paginas físicas (`NumPhysPages`).
A continuación, modificamos la clase `AddrSpace` para que soporte la multiprogramación:

- **Chequeo de espacio**: Se modifico el calculo de cuanto espacio necesita un archivo ejecutable en memoria para que soporte el paginado.

```
size = noffH.code.size + noffH.initData.size
       + noffH.uninitData.size + UserStackSize;
numPages = divRoundUp(size, PageSize);
size = numPages * PageSize;
```

- **Asegurarse de que hay espacio disponible**: Luego de calcular la cantidad de paginas que necesitamos, tenemos que chequear que tengamos libre la cantidad suficiente.

- **Guardar el marco en cada pagina**: Cuando se arma cada pagina, guardamos en `physicalAddr` el marco correspondiente a la memoria física. Dicho marco, es uno libre que se obtiene mediante el mapa de bits.

```
int firstFreePhySpace = -1;
for (i = 0; i < numPages; i++) {
    firstFreePhySpace = memoryMap->Find();
    ASSERT(firstFreePhySpace != -1);    //Always found space in
       physical memory.
    pageTable[i].physicalPage = firstFreePhySpace;
    ...
}
```

- **Inicializa la pagina**: Se inicializa cada pagina a traves del metodo `bzero`.

```
for (i = 0; i < numPages; i++) {
    ...
    bzero(&(machine->mainMemory[firstFreePhySpace*PageSize]),
       PageSize);
}
```

- **Carga de segmentos**: Finalmente, se cargan los segmentos del archivo ejecutable en memoria. Dicho proceso, se modularizo en el método `LoadSegment`, cuya función es cargar un segmento en memoria.

Cambio de contexto con rebanadas de tiempo

Para forzar los cambios de contexto, se modificó la creación del `timer`, al momento de iniciar ***NachOS***.

```
#ifdef USER_PROGRAM
    randomYield = false;
#endif
    timer = new Timer(TimerInterruptHandler, 0, randomYield);
```

Al invocar el constructor con el tercer parámetro en `false`, se crea un temporizador cuyo timeout ocurrirá cada `TimerTicks` unidades. La definición de `TimerTicks` se encuentra en `stats.h`.

3.4. Exec con argumentos

A lo desarrollado en Exec sin argumentos, se agrego el procesamiento del segundo y tercer argumento. La cantidad de argumentos nos viene en el registro `R5` y el string de argumentos sin traducir, vienen por el registro `R6`. Para traducir el vector de argumentos, invocamos a método `readSpecialStringFromUser`, declarado en `usertranlate.h`, para traducir cada argumento y guardarlos en el array `argv`. Por convención, el string con los argumentos estará separado por espacios. Al finalizar la traducción, se invoca a `makeProcess` y se continua como mencionamos en Exec sin argumentos. Ademas, se modifico la función `doExecution` para que inicialice los argumentos antes de ejecutar el archivo.
Por otro lado, se modifico la clase `AddSpace`, ya que necesitamos copiar los argumentos en el stack de usuario. En la llamada al constructor, se guardan `argc` y `argv`. Tambien se creo el método `AddrSpace::InitArguments`, el cual se encarga de almacenar correctamente los argumentos en el stack. Empieza guardando en la parte superior de la pila el valor de los argumentos, seguidos los punteros a esas direcciones. Finalmente, mueve el stackpointer cuatro posiciones mas abajo del ultimo argumento ingresado en la pila.

3.5. Interpretes de comandos y programas de usuario

Se implemento un interprete de comandos, el cual lee un comando de la consola y lo ejecuta. Si el comando comienza con el carácter `&`, debe ser ejecutado en background. En `test/shell.c` se encuentra la implementación del interprete.
Ademas, se implementaron los siguientes programas utilitarios:

cat Su función es leer un archivo e imprimirlo en la consola.

cp Su función es copiar el contenido de un archivo a otro. Para ello, necesita el nombre del archivo a copiar y el nombre del archivo destino.

clear Su función es limpiar la consola. Para ello, imprime 250 saltos de linea.

Las implementaciones de todos ellos se encuentran la carpeta `'test/'`.

Capítulo 4

Resolución de la Práctica N°4: Memoria virtual y TLB

4.1. Implementar TLB

Capturamos las excepciones `PageFaultException` y `ReadOnlyException`.
Cuando ocurre la primera, se obtiene la dirección de memoria que la originó leyendo el valor del registro `R39`. Luego se invoca a `AddrSpace::UpdateTLB` para que cree una nueva entrada en la tabla para dicho valor. Notemos que al retornar no se realiza un aumento en el `PC`. Por lo tanto, se ejecutara nuevamente la sentencia que origino la excepción.

```
1  if (which == SyscallException) {
2          ...
3  } else {
4    DEBUG('e', "Is not a SyscallException\n");
5
6    const char *exception = "";
7    switch(which) {
8      ...
9      case PageFaultException:
10     {
11       int failVirtAddr = machine -> ReadRegister(BadVAddrReg);
12
13       DEBUG('v',"Antes de actualizar la TLB: failVAddr=%d\n",
             failVirtAddr);
14       currentThread->space->UpdateTLB(failVirtAddr / PageSize);
15       return;
16     }
17     case ReadOnlyException:
18       exception = "ReadOnlyException";
19    // Solamente notificamos que ocurrio.
20       printf("An exception was triggered: which=%s  type=%d\n",
21              exception, type);
22       return;
23       ...
24
```

```
25          default:
26            printf("Unexpected user mode exception.\n");
27            ASSERT(false);
28      }
29      printf("Unexpected exception:\t which=%s  type=%d\n",
30             exception, type);
31      ASSERT(false);
32  }
```

Como mencionamos anteriormente, se modificó la clase `AddrSpace`

```
1  void UpdateTLB(int position);   // update TLB table;
```

Se utiliza para cuando la página a utilizar no se encuentra en el **TLB**. Se encarga de buscarla en la memoria y guardarla en la **TLB**.

4.2. Análisis de TLB cambiando la cantidad de entradas

Modificaciones

Para ver como influye el tamaño de la **TLB** respecto a la cantidad de paginas que fallan, se agregaron dos indicadores a las estadísticas de **NachOS**. Ellos se encuentran en `stats.h`.

```
1  int numPageFaults;  // number of virtual memory page faults
2  int numPagesFound;  // number of virtual memory page founds
```

El primero se incrementa cuando ocurre un fallo de paginación. Esto ocurre cuando se captura la excepción `PageFaultException`. En cambio, el segundo se incrementa cuando se encuentra la pagina. Ocurre al final del constructor `Translate::Translate`.
Para modificar el tamaño de la TLB, se modifica la variable `TLBSize`, declarada en `system.h`.

Resultados

Se realizaron distintas pruebas variando la cantidad de entradas de la tabla. Los valores de prueba fueron 4, 16, 32, 64 y 128. Además, la página que sale es elegida al azar.
En 4.1 se muestra los porcentajes de aciertos y fallas de la **TLB**, para los programas `matmult.c` y `sort.c`.
Para ésta forma de seleccionar la página que sale de la **TLB**, encontramos una cota máxima de rendimiento en una tabla con 64 entradas. Notamos que una tabla con mas entradas, se sigue teniendo el mismo rendimiento.

TLB size	Programa			
	`mathmult`		`sort`	
	faults	hits	faults	hits
4	7,60 %	92,39 %	4,36 %	95,64 %
16	0,569 %	99,43 %	0,042 %	99,958 %
32	0,013 %	99,986 %	0,035 %	99,996 %
64	0,006 %	99,993 %	0,0001 %	99,999 %
128	0,006 %	99,993 %	0,0001 %	99,999 %

Cuadro 4.1: Estadísticas para los programas `mathmult` y `sort`, variando el tamaño de la TLB.

4.3. Implementar carga por demanda

Cuando se inicia un programa, no va a tener ninguna de sus páginas en memoria. Ellas, se cargarán cuando el sistema solicite acceder a alguna de ellas.
Para implementar esto, cuando se crea el espacio de direcciones de un thread se le asigna `-1` a la dirección física de cada una de sus páginas y a su vez se setea el bit de `valid` en `false`. Con esto último se indica que la página no está en memoria. Por lo tanto, las páginas se irán cargando en memoria conforme se reciban las `PageFaultException`.
Se agregaron los siguientes métodos a la clase `AddrSpace`:

```
public:
    ...
    void LoadPage(TranslationEntry *page);//Load a page into memory.
private:
    OpenFile *executable_file; // Save executeble for load later.
    NoffHeader noff_hdr; // Save header for load later

    // Class constructor with DEMAND_LOADING flag
    void costructorForDemandLoading(OpenFile *executable,
                int prg_argc, char** prg_argv, int pid);
```

Se modificó el método `AddrSpace::UpdateTLB` para que se verifique el estado de la página. Si ya tiene asignada una dirección física, se continúa con la actualización del **TLB** como se explicó en Implementar TLB. En caso contrario, se llama al método `AddrSpace::LoadPage`, el cual le asigna a la página una posición en el mapa de bits y la carga en memoria. Finalmente, continua la ejecución de `AddrSpace::UpdateTLB` para actualizar el **TLB**.

4.4. Implementar la política de paginación FIFO e implementar SWAP

Desarrollo de SWAP

Con el objetivo de proveer la ilusión de una memoria mucho más grande que la física, cuando se crea el espacio de direcciones para un programa, se creará un archivo que servirá de **SWAP**.
Además, necesitamos saber qué páginas se encuentran *swappeadas*. Para manejar esto, se crea un arreglo `swapMemory` de booleanos, donde el `i-ésimo` elemento indica si la página `i` está o no *swappeada*.

Se agregaron los siguiente miembros a la clase `AddrSpace`:

```
private:
  ...
  OpenFile *swapFile;     // File for swapping.
  char swapFileName[8];   // Name of swapping file.
  int* swapMemory; //Boolean list that indicates if a page is on
     SWAP.

// Class constructor with VM_SWAP flag
  void costructorForSwap(OpenFile *executable, int prg_argc,
                         char** prg_argv, int pid);

  void MemToSwap(int virtualAddr, int physicalAddr);
  void SwapToMem(TranslationEntry *page);
```

Finalmente, se debe llevar un control de las direcciones físicas ocupadas y con qué direcciones virtuales y procesos están vinculadas. Para ello, se modificaron dos clases: `BitMap` y `ProcessTable`. En la primera se guarda la vinculación entre `physical address` y `virtual address`.

```
class BitMap {
  ...
  public:
// effect, set a bit. If no bits are clear, find a candite to leave
// the memory and put it into swap. Then, return his position.
    int FindFrameForVirtualAddress(int virtualPage);
  private:
// Map from physical address to virtual address.
    int* physicalToVirtual;
```

Mientras que en la segunda, se guarda la vinculación entre `physical address` y `threadId`.

```
class ProcessTable {
  public:
    ...
 //Given a physical address, returns the appropiate process.
    Thread* getProcessByPhysAddr(int physAddr);
    void SetPhysAddress(SpaceId pid, int physAddr);
    void ClearProcessPhysAddress(int physAddr);
  private:
    ...
    SpaceId* pids;  // Map of physical address to pid.
```

Cuando se necesita espacio para cargar una página, en vez de invocar a `memoryMap->Find()`, se invocará a `memoryMap->FindFrameForVirtualAddress()`. Su función es la siguiente:

1. Invoca a `BitMap::Find()`. Si obtiene un espacio, lo devuelve.
2. Si el resultado fue `-1`, entonces la memoria está llena. Por lo tanto, tiene que aplicar una política para obtener un espacio. Cada algoritmo se encarga de liberar un espacio en memoria, enviando a un archivo de **SWAP**, la página que se eligió como víctima.

3. Marca esa dirección como ocupada, invocando a `BitMap::Mark()`.
4. Asocia el `pid` del proceso actual con la dirección obtenida, invocando a `ProcessTable::SetPhysAddress`.
5. Asocia la dirección virtual dada con la dirección física obtenida, guardándola en el mapa `physicalToVirtual`.
6. Retorna la dirección obtenida.

Política de paginación FIFO

Para implementar esta política se define en `system.cc` una cola `fifo`, la cual se encargará de guardar el orden en que las direcciones físicas van siendo asignadas. Al iniciarse el sistema, se creará dicha cola.
La lógica del algoritmo queda encapsulada en el método privado `BitMap::fifoAlgorithm`, el cual será invocado por `BitMap::FindFrameForVirtualAddress`, tal como se explicó en Desarrollo de SWAP

```
  private:
    ...
// Return the virtual address of the page candidated to leave
// the memory using FIFO algorithm.
    int fifoAlgorithm();
```

4.5. Mejorar la política de paginación

Se desarrollo la política llamada *Segunda oportunidad mejorada*. Para ello, necesitamos información del estado de los bits de las páginas. También necesitamos poder modificar el `bit 'modificación'`. Se modificó la clase `AddrSpace` agregando los siguientes métodos:

```
public:
  ...
  bool IsValid (int pos); //  Getter for bit valid of a page.
  bool IsUsed (int pos);  //  Getter for bit used of a page.
  void SetUse (int pos, bool b);  //  Setter for bit used a page.
  bool IsDirty (int pos); //  Getter for bit dirty of a page.
```

Finalmente, se encapsula el algoritmo en el método privado `BitMap::secondChanceAlgorithm`. Como se mencionó en la sección anterior, este método será invocado por `BitMap::FindFrameForVirtualAddress`, tal como se explicó en Desarrollo de SWAP.

```
  private:
    ...
// Return the virtual address of the page candidated to leave
// the memory using 'second chance' algorithm.
    int secondChanceAlgorithm();
```