

**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 #2: Caching: TLB y memoria virtual**

**Profesor:**  
Francisco Arroyo

**Estudiantes:**  
Luis Porras Ledezma | B65477  
José Pablo Ramírez | B65728

**10 de Julio del 2018**

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Objetivos generales</b>	<b>4</b>
<b>3. Objetivos específicos</b>	<b>4</b>
<b>4. Descripción</b>	<b>5</b>
<b>5. Direcciones</b>	<b>7</b>
<b>6. Diseño</b>	<b>9</b>
<b>7. Desarrollo</b>	<b>10</b>
<b>8. Manual de usuario</b>	<b>25</b>
Compilación . . . . .	25
Especificación de las funciones del programa . . . . .	25
<b>9. Casos de Prueba</b>	<b>26</b>

# 1. Introducción

El proyecto consiste en mostrar, a groso modo, las estructuras, métodos, estrategias y soluciones desarrolladas para resolver el problema de memoria virtual en el sistema operativo emulado NachoOS. Dicho problema consiste en que, al cargar programas a memoria principal y el estado de la misma no da abasto (está llena) es necesario utilizar un espacio reservado en disco (SWAP) para mover contenido y dar espacio en memoria. En el proyecto se utilizó un esquema de paginación por demanda haciendo uso del `Translation Lookaside Buffer` (TLB) proporcionado por NachOS. Así también, se utiliza el algoritmo de reemplazo para TLB y SWAP second chance, como se mostrará más adelante. Finalmente, se mostraran los resultados al realizar pruebas que requieran de memoria virtual, como por ejemplo, sort.

## 2. Objetivos generales

- Entender el concepto de "page fault"(falta de página)
- Comprender los elementos involucrados cuando ocurre una excepción tipo "Page Fault" la manera de resolverla
- Características de la página faltante
- Actualización de estructuras

## 3. Objetivos específicos

- Atrapar la excepción de "PageFault" en "ExceptionHandler"
- Modificar el constructor de la clase "AddrSpace" para invalidar todas las páginas lógicas
- Modificar el método "AddrSpace::RestoreState" para conseguir que el sistema emplee TLB únicamente
- Agregar la lógica necesaria para cargar las páginas de memoria que necesita un proceso
- Establecer el procedimiento para el manejo de la excepción de falta de página ("page fault exception")
- Correr programas de usuario que llenen la memoria física
- Crear el archivo de "SWAP" en Linux para que funcione como intercambio para NachOS
- Determinar los distintos flujos de las páginas físicas
- Determinar las características de las páginas faltantes y su estrategia para colocarlas en memoria
- Establecer el procedimiento para el reemplazo de páginas en la memoria principal
- Establecer el procedimiento para el reemplazo de entradas en la TLB

## 4. Descripción

- En la tercera fase de Nachos se investigará el uso de caching que se utilizará en esta tarea con dos propósitos:
  - Primero, se va a emplear un traductor de direcciones basado en software (Translation Lookaside Buffer [TLB]) como una memoria cache para la tabla de páginas, con el fin de dar la ilusión de un acceso rápido al traductor de direcciones virtuales que funcionaría en un espacio de direcciones muy grande.
  - Además, se va a utilizar el disco como parte de la memoria principal, con el fin de proveer un espacio de direccionamiento (casi) ilimitado, con el desempeño cercano al de la memoria física.
  - Para esta asignación no se provee código nuevo, el único cambio necesario de compilar el código ya existente (threads y userprog) con las banderas DVM y DUSE\_TLB), incluidas dentro del Makefile del proyecto "vm" (-DVM y -DUSE\_TLB); su tarea será escribir el código para manejar el TLB e implantar la memoria virtual.
- En el proyecto #2 se utilizaron tablas de páginas para simplificar la asignación de memoria y aislar los fallos a un espacio de direcciones, de manera que no afecten a otros programas.
  - Para esta fase, el hardware no sabe nada de las tablas de páginas.
  - En su lugar, solo trabaja con un cache, cargado por software (el sistema operativo, alias usted) de las entradas de las tablas de páginas, denominado TLB.
  - En casi todas las arquitecturas modernas se incorpora un TLB a fin de aligerar las traducciones del espacio de direcciones.
    - Dada una dirección de memoria (una instrucción que se desea buscar o un dato para cargar o guardar), el procesador primero busca en el TLB para determinar si el mapeo de la dirección virtual a la dirección física ya se conoce.
    - Si es así (acierto TLB) entonces la traducción puede efectuarse rápidamente.
    - Pero si el mapeo no se encuentra en el TLB (fallo TLB) entonces se deben acceder las tablas de páginas y/o de segmentos.
    - En muchas arquitecturas, entre ellas Nachos, DEC MIPS y las HP Snakes, un fallo TLB causa una trampa al kernel del sistema operativo, que se encarga de efectuar la traducción, carga la traducción en el TLB y regresa al programa que efectuó el fallo.
    - Esto permite al kernel del sistema operativo elegir entre cualquier combinación de tabla de páginas, tabla de segmentos, tabla invertida, etc., necesaria para llevar a cabo la traducción de la dirección.
    - En sistemas que no utilicen TLB basado en software, entonces el hardware realiza la misma labor, pero en este caso el hardware debe especificar el formato exacto para las tablas de páginas y segmentos.
    - Por esta razón las TLB manipuladas por software son más flexibles, al costo de ser un poco más lentas para manejar los fallos TLB. Si estos fallos son muy infrecuentes, el impacto en el desempeño de un sistema TLB manejado por software es mínimo.
- La ilusión de tener memoria ilimitada es provista por el sistema operativo, utilizando la memoria principal como un cache para el disco (swap).
- Para esta asignación, la traducción de las direcciones de las páginas, permite la flexibilidad de traer páginas del disco en el momento en que se necesiten.
- Cada entrada en el TLB posee un bit de validez:
  - Si el bit está encendido entonces la página virtual está en memoria, solo se debe actualizar el TLB.

- Si está apagado o la página virtual no está en el TLB, entonces se necesita una tabla de páginas administrada por software (SO, alias usted) para saber si la página está en memoria (cargando el TLB cuando se efectúe la traducción) o si la página debe ser traída del disco (swap).
- Adicionalmente, el hardware coloca el bit de uso cada vez que la página se utiliza y el bit de suciedad si el contenido fue modificado.
- Cuando el programa hace referencia a una página que no se encuentra en el TLB, el hardware genera una excepción de falta de página (PageFault exception), llevando el control al kernel (ExceptionHandler).
- El kernel, verifica en la tabla de páginas del hilo/proceso, si la página no está en memoria, entonces lee la página del disco, corrige la entrada en la tabla para apuntar a la nueva página y luego pasa el control al programa que produjo la excepción.
- Por supuesto, el kernel debe encontrar primero espacio en la memoria principal para esta nueva página que debe ser traída, probablemente desocupando una casilla y escribiendo esa página removida en el disco (swap), si ésta fue modificada antes (dirty).
- Como en cualquier sistema de caching, el desempeño depende de la estrategia para determinar cuáles de las páginas permanecen en memoria y cuáles se mantienen en el disco.
- En un fallo de página, el kernel debe decidir cuál de las páginas va a reemplazar; idealmente debe reemplazar aquella que no se va a utilizar por un largo tiempo, manteniendo en memoria solamente las páginas que van a ser referenciadas pronto.
- Otra consideración importante, es que si la página que se reemplaza había sido modificada, ésta debe ser guardada en el disco antes de traer la nueva.
- En muchos sistemas de memoria virtual (como UNIX) se evita ese gasto de tiempo extra, escribiendo las páginas al disco por adelantado, de manera que cualquier fallo de página posterior podrá ser completado más rápidamente.

## 5. Direcciones

### 1. Implantar el TLB administrado por software.

- a) Para ello es necesario crear un sistema de traducción manipulado por software que maneje los fallos TLB.
  - 1) Revise la traducción de direcciones lógicas a físicas que ocurren en el método "Translate" en el archivo "translate.cc" del directorio "machine".
  - 2) Investigue cómo se realizan las traducciones y los tipos de excepciones que se pueden generar, ponga especial atención a "PageFaultException".
  - 3) Determine qué cosas debe cambiar para que esa excepción no ocurra de nuevo cuando se re-ejecuta la instrucción.
- b) Note que utilizando la bandera "DUSE\_TLB" en tiempo de compilación, el hardware no utiliza las tablas de páginas; por lo que es necesario asegurarse de que el estado de la TLB es colocado correctamente en los cambios de contexto.
  - 1) Revise el método "RestoreState" de la clase "AddrSpace".
  - 2) Revise los "ASSERT" que se encuentran al principio del método "Translate".
  - 3) Intente correr el programa de usuario "halt" y determine que ocurre.
  - 4) Corrija el método "RestoreState" para que no inicialice la variable "machine->pageTable" ni "machine->pageTableSize", utilice compilación condicional para que el proyecto de "user-prog" no se estropee.

```
1 void AddrSpace::RestoreState()  
2 {  
3     #ifndef VM  
4         machine->pageTable = pageTable;  
5         machine->pageTableSize = numPages;  
6     #endif  
7 }
```

- c) Muchos de los sistemas lo que hacen es invalidar al TLB completamente cada vez que ocurre un cambio de contexto; las entradas se van recargando conforme las páginas son referenciadas nuevamente.
- d) Para el ítem 2, su esquema de traducción de páginas debe llevar cuentas de las páginas sucias y utilizar las banderas que el hardware coloca en las entradas del TLB.

### 2. Implantar memoria virtual.

- a) Para ello se necesitan rutinas para mover páginas del disco a la memoria y viceversa.
- b) Se recomienda que utilice el sistema de archivos de Nachos como área de respaldo (backing-store), de esta manera, cuando se implante el sistema de archivos en la asignación 4, es posible utilizar el sistema de memoria virtual como un caso de prueba.
- c) A fin de encontrar las páginas que no se referencian y sacarlas cuando ocurre un fallo de página, es necesario llevar pista de todas las páginas que están siendo utilizadas en el sistema.
- d) Una manera simple de hacer esto, es empleando un "core map", que es básicamente una tabla de páginas invertida, en lugar de traducir números de página virtual en números de páginas físicas (marcos), un "core map" traduce de número de página física al número de página virtual del hilo/proceso que está utilizando esa página física.

3. Evalúe el desempeño de su sistema. Los fallos de cache (en este caso fallos TLB y fallos de página) pueden ser divididos en tres categorías:

- a) "Compulsory misses", aquellos que ocurren debido a la primera referencia de un ítem; siempre la página debe ser traída del disco y colocada en la memoria y en el TLB.
  - b) "Capacity misses", aquellos debidos al tamaño del cache; si el "working set" del programa de mayor que la memoria principal o que el número de entradas en el TLB, el programa incurre en fallos. Estos ocurren en un cache que no es infinito.
  - c) "Conflict misses", aquellos debidos a la política de reemplazo del cache. No ocurrirían si el cache utilizara una política óptima para efectuar el reemplazo de sus páginas, para el mismo programa en un cache del mismo tamaño.
4. Escriba un conjunto de programas de usuario "útiles" para demostrar fallos de TLB y páginas (pocos y muchos fallos).
- a) En otras palabras, escriba un programa de usuario de prueba que muestre cuando ocurren pocos fallos en el TLB; otro que muestre cuando ocurren pocos fallos de página; otro que muestre cuando ocurren muchos fallos de TLB; etc.
  - b) Como ejemplo, se presentan los programas "sort.c" y "matmult.c" en el directorio "test" que presentan un gran número de "conflict misses" para la mayoría de políticas de administración.
  - c) Para cada caso de prueba, explique el desempeño de su sistema e indique como se podría mejorar.
5. Probablemente le resulte útil reducir el tamaño de la memoria (en machine.h) para provocar más rápidamente el comportamiento de la paginación.



## 6. Diseño

A continuación se desprende una breve descripción del diseño utilizado y principalmente de las estructuras creadas para la solución del problema planteado de memoria virtual en NachOS.

### 1. Tabla de páginas invertidas.

Esta estructura cumple la funcionalidad de mantener una relación 1:1 de las páginas físicas de la memoria principal con el objeto de tipo `TranslationEntry` del hilo que tenga dicha página física en uso. De la misma manera es importante para poder utilizar el algoritmo de second chance para elegir una "víctima" que es enviada al SWAP.

### 2. Archivo SWAP en disco.

Este archivo creado en disco se utiliza para enviar páginas físicas elegidas como víctimas desde la memoria principal en NachOS.

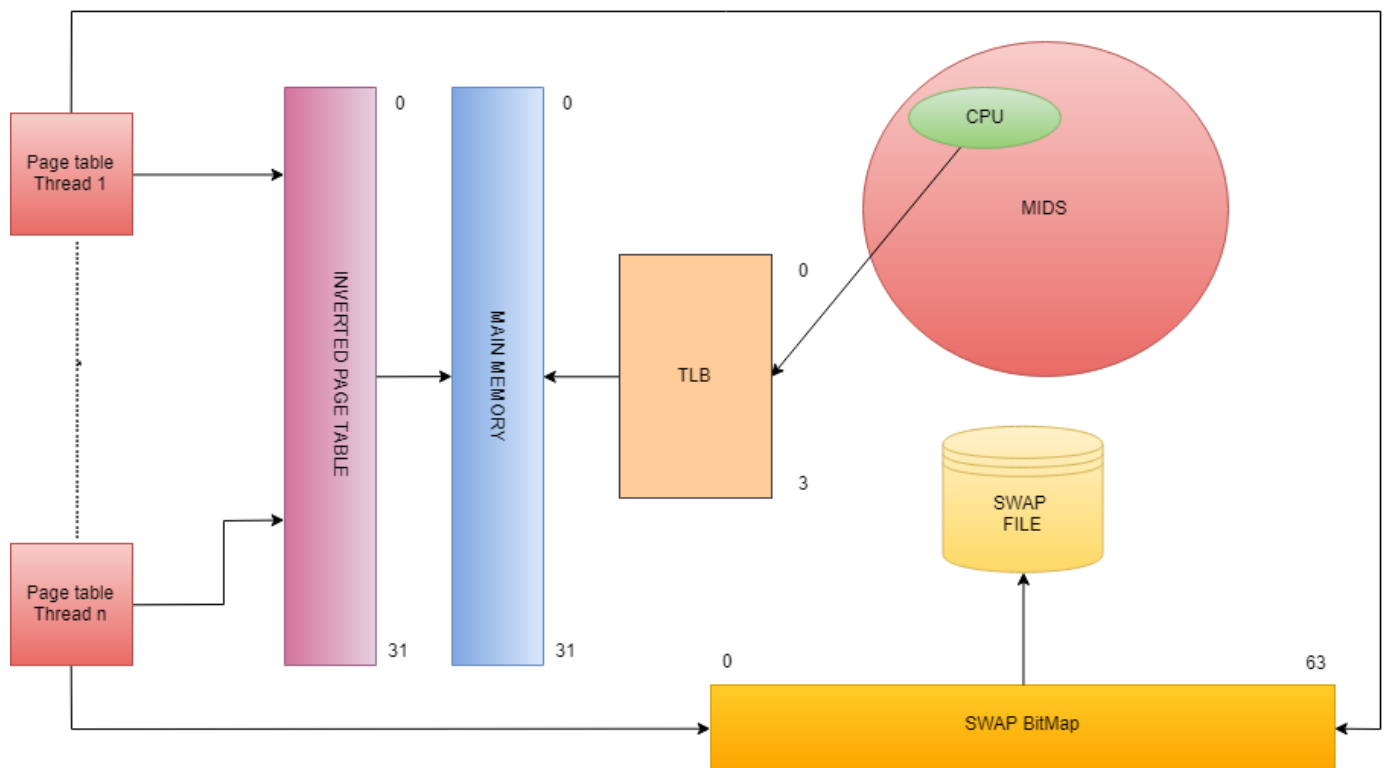
### 3. Estructura BitMap para operar sobre el archivo de SWAP

La estructura bitmap determinada para la relación 1:1 con las posiciones que representan "páginas físicas", pero en este caso para el archivo SWAP.

### 4. Estructura TLB que se habilita para Virtual Memory simulada por el "Hardware" de NachOS

Estructura creada en memoria dinámica, que representa un arreglo de cuatro entradas de tipo `TranslationEntry` que se utiliza para mantener páginas usadas de manera reciente de la memoria principal.

A continuación, se desprende un bosquejo del diseño implementado, a nivel de estructuras, para la solución.



## 7. Desarrollo

Para la realización de este proyecto fue necesario modificar los archivos "exception.cc" y especialmente "addrspace.h" y "addrspace.cc" todos de la carpeta "userprog" de NachOS. A continuación se mencionan los cambios realizados a los archivos mencionados y se explican los métodos implementados para solucionar el problema de virtual memory.

Lo primero que se hizo para solucionar el problema fue modificar el constructor del AddrSpace de la siguiente forma:

```
1 AddrSpace::AddrSpace(OpenFile *executable, std::string fn ){
2     ...
3     DEBUG('a', "Initializing address space, num pages %d, size %d\n",
4           numPages, size);
5     // first, set up the translation
6     pageTable = new TranslationEntry[numPages];
7     for (i = 0; i < numPages; i++) {
8         pageTable[i].virtualPage = i;    // for now, virtual page # = phys page
9         #
10        #ifdef VM
11        pageTable[i].physicalPage = -1;
12        pageTable[i].valid = false;
13        #else
14        pageTable[i].physicalPage = MemBitMap->Find();
15        pageTable[i].valid = true;
16        #endif
17        pageTable[i].use = false;
18        pageTable[i].dirty = false;
19        pageTable[i].readOnly = false; // if the code segment was entirely on
20        // a separate page, we could set its
21        // pages to be read-only
22    }
23
24    initData = divRoundUp(noffH.code.size, PageSize);
25    noInitData = initData + divRoundUp(noffH.initData.size, PageSize);
26    stack = numPages - divRoundUp(UserStackSize, PageSize);
27    ...
28 }
```

Entre los principales cambios realizados están el uso de del #ifdef VM para realizar las acciones pertinentes en el AddrSpace para que funcione virtual memory, entre estas: inicializar todas las paginas como invalidas y con la dirección física -1, pues ahora se quiere implementar paginación por demanda. Y no cargar ninguna pagina a memoria como se hacia normalmente. Otra utilidad de utilizar esa condiciones (#ifdef o #ifndef) es que permite que NachOS funciones normalmente aunque este se compile en la carpeta "userprog" aunque en esta no se defina VM. Esta condiciones tambien se utilizaron en los metodos "SaveState" y "RestoreState" como se muestra a continuación:

```
1 void AddrSpace::SaveState()
2 {
3     #ifdef VM
4     DEBUG ( 't', "\nSe salva el estado del hilo: %s\n", currentThread->getName() );
5     for(int i = 0; i < TLBSize; ++i){
6         pageTable[machine->tlb[i].virtualPage].use = machine->tlb[i].use;
7         pageTable[machine->tlb[i].virtualPage].dirty = machine->tlb[i].dirty;
```

```

8      }
9      /*
10     if (machine->tlb != NULL)
11     {
12         delete [] machine->tlb;
13         machine->tlb = NULL;
14     }
15     */
16     machine->tlb = new TranslationEntry[ TLBSize ];
17     for (int i = 0; i < TLBSize; ++i)
18     {
19         machine->tlb[i].valid = false;
20     }
21     #endif
22 }
23
24 void AddrSpace::RestoreState()
25 {
26     DEBUG ( 't', "\nSe restaura el estado del hilo: %s\n", currentThread->getName()
27         );
28     #ifndef VM
29     machine->pageTable = pageTable;
30     machine->pageTableSize = numPages;
31     #else
32     indexTLBFIFO = 0;
33     indexTLBSndChc = 0;
34     threadFirstTime = true;
35     /*
36     if (machine->tlb != NULL)
37     {
38         delete [] machine->tlb;
39         machine->tlb = NULL;
40     }
41     */
42     machine->tlb = new TranslationEntry[ TLBSize ];
43     for (int i = 0; i < TLBSize; ++i)
44     {
45         machine->tlb[i].valid = false;
46     }
47     #endif
48 }

```

Otros cambios importantes a destacar es la agregación de campos a la clase AddrSpace. Se agregó uno llamado filename de tipo std::string, utilizado para guardar el nombre del archivo ejecutable que esta corriendo el hilo con el AddrSpace actual, esto para poder abrirlo de nuevo en caso de ser necesario (tiende a ser bastante necesario) y también se agregaron enteros que indican el numero de pagina en el que empiezan los diferentes segmentos del programa (código, datos inicializados, datos no inicializados y pila). Como se mencionó anteriormente otro archivo que tuvo que ser modificado fue "exception.cc" específicamente su método "ExceptionHandler" para que ahora se pueda detectar un PageFaultException y tomar las medidas necesarias, de esta forma:

```

1 void ExceptionHandler(ExceptionType which)
2 {
3     int type = machine->ReadRegister(2);
4     unsigned int vpn;

```

```

5
6  switch ( which ) {
7      ...
8  break;
9      case PageFaultException:
10         DEBUG('v', "\nPageFaultException\n");
11         vpn = (machine->ReadRegister ( 39 ));
12         DEBUG('v', "Direccion logica: %d\n", vpn);
13         vpn /= PageSize;
14         DEBUG('v', "Pagina que falla: %d\n", vpn);
15         currentThread->space->load(vpn);
16     break;
17     case ReadOnlyException:
18         printf("\nReadOnlyException\n");
19         ASSERT(false);
20     break;
21     case BusErrorException:
22         printf("\nBusErrorException\n");
23         ASSERT(false);
24     break;
25     case AddressErrorException:
26         printf("\nAddressErrorException\n");
27         ASSERT(false);
28     break;
29     case OverflowException:
30         printf("\nOverflowException\n");
31         ASSERT(false);
32     break;
33     case IllegalInstrException:
34         printf("\nIllegalInstrException\n");
35         ASSERT(false);
36     break;
37     case NumExceptionTypes:
38         printf("\nNumExceptionTypes\n");
39         ASSERT(false);
40     break;
41     default:
42         printf("\nUnexpected exception %d\n", which );
43         ASSERT(false);
44     break;
45 }
46 }

```

Una vez realizadas las preparaciones descritas anteriormente se procedió a resolver el problema. Para resolver el problema se siguió el consejo del profesor y se hizo un método "load" en la clase AddrSpace el cual en grandes términos es el que hace que virtual memory funcione.

El método "load" recibe un unsigned int vpn que es el la pagina lógica que esta solicitando el proceso que esta en ejecución, como existen múltiples razones por las que se pudo general la excepción page fault, este método basándose en las banderas (dirty y valid) de la pagina pasada por parámetro determina las acciones a tomar, para permitir la ejecución del programa. Estas acciones que realiza ente método se describen en la siguiente tabla:

Valid	Dirty	Acción a tomar
false	false	Se debe revisar a cual segmento pertenece la pagina: <ul style="list-style-type: none"> <li>Codigo o datos inicializados: Se debe abrir el archivo ejecutable y cargar la pagina desde esta memoria.</li> <li>Datos no inicializados o pila: Se debe crear una pagina nueva vacia para el proceso y "cargarla" en memoria.</li> </ul>
false	true	Se debe recuperar la pagina del SWAP y cargarla a memoria.
true	false	La pagina ya esta en memoria, solo se debe actualizar el TLB.
true	true	La pagina ya esta en memoria, solo se debe actualizar el TLB.

En todos los casos se debe considerar el caso de que no haya memoria disponible y entonces esta se deba liberar, ya sea mandando una de las paginas que ya esta en memoria al SWAP o simplemente quitarla de ahí (va a depender de las banderas de esta memoria).

A continuación se adjunta el código del método "load":

```

1 void AddrSpace::load( unsigned int vpn )
2 {
3     int freeFrame;
4     DEBUG('v', "Numero de paginas: %d, hilo actual: %s\n", numPages, currentThread
        ->getName());
5     DEBUG('v', "\tCodigo va de [%d, %d[ \n", 0, initData);
6     DEBUG('v', "\tDatos inicializados va de [%d, %d[ \n", initData, noInitData);
7     DEBUG('v', "\tDatos no inicializados va de [%d, %d[ \n", noInitData, stack);
8     DEBUG('v', "\tPila va de [%d, %d[ \n", stack, numPages );
9
10    //Si la pagina no es valida ni esta sucia.
11    if ( !pageTable[vpn].valid && !pageTable[vpn].dirty ){
12        //Entonces dependiendo del segmento de la pagina, debo tomar la
            decisi n de donde cargar esta pagina?
13        DEBUG('v', "\t1-La pagina es invalida y limpia\n");
14        DEBUG('v', "\tArchivo fuente: %s\n", filename.c_str());
15        ++stats->numPageFaults;
16        OpenFile* executable = fileSystem->Open( filename.c_str() );
17        if (executable == NULL) {
18            DEBUG('v', "Unable to open source file %s\n", filename.c_str() )
                ;
19            ASSERT(false);
20        }
21        NoffHeader noffH;
22        executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
23
24        //Nesecito verificar a cual segmento pertenece la pagina.
25        if(vpn >= 0 && vpn < initData){ //segmento de Codigo
26            DEBUG('v', "\t1.1 Pagina de codigo\n");
27            //Se debe cargar la pagina del archivo ejecutable.
28            freeFrame = MemBitMap->Find();
29
30            if ( freeFrame != -1 )
31            {
32                DEBUG('v', "\tFrame libre en memoria: %d\n", freeFrame )

```

```

33         ;
34         pageTable[ vpn ].physicalPage = freeFrame;
35         executable->ReadAt( &(machine->mainMemory[ ( freeFrame *
36             PageSize ) ] ),
37         PageSize, noffH.code.inFileAddr + PageSize*vpn );
38         pageTable[ vpn ].valid = true;
39         //pageTable[ vpn ].readOnly = true;
40
41         //Se actualiza la TLB invertida
42         IPT[freeFrame] = &(pageTable[ vpn ]);
43         //Se debe actualizar el TLB
44         int tlbSPace = getNextSCTLB();
45         useThisTLBIndex( tlbSPace, vpn );
46
47     }else
48     {
49         //victima swap
50         indexSWAPFIFO = getNextSCSWAP();
51         updateSwapVictimInfo( indexSWAPFIFO );
52         //////////////fin de secondChance para el SWAP
53         //////////////
54         bool victimDirty = IPT[indexSWAPFIFO]->dirty;
55
56         if ( victimDirty )
57         {
58             DEBUG('v',"\\t\\t\\tVictima f=%d,l=%d y sucia\\n",
59                 IPT[indexSWAPFIFO]->physicalPage, IPT[
60                 indexSWAPFIFO]->virtualPage );
61             writeIntoSwap( IPT[indexSWAPFIFO]->physicalPage
62                 );
63
64             // pedir el nuevo freeFrame
65             freeFrame = MemBitMap->Find();
66             // verificar que sea distinto de -1
67             if ( -1 == freeFrame )
68             {
69                 printf("Invalid free frame %d\\n",
70                     freeFrame );
71                 ASSERT( false );
72             }
73             // actualizar la pagina fisica para la nueva
74             // virtual vpn
75             pageTable[ vpn ].physicalPage = freeFrame;
76             // cargar el código a la memoria
77             executable->ReadAt( &(machine->mainMemory[ (
78                 freeFrame * PageSize ) ] ),
79             PageSize, noffH.code.inFileAddr + PageSize*vpn
80             );
81             // actualizar la validez
82             pageTable[ vpn ].valid = true;
83             // actualizar la tabla de paginas invertidas
84             IPT[ freeFrame ] = &( pageTable[ vpn ] );
85             // finalmente, actualizar tlb
86             int tlbSPace = getNextSCTLB();
87             useThisTLBIndex( tlbSPace, vpn );
88             //ASSERT(false);
89         }else

```

```

79         {
80             DEBUG('v', "\t\t\tVictima f=%d,l=%d y limpia\n",
81                 IPT[indexSWAPFIFO]->physicalPage, IPT[
82                 indexSWAPFIFO]->virtualPage );
83             int oldPhysicalPage = IPT[indexSWAPFIFO]->
84                 physicalPage;
85             IPT[indexSWAPFIFO]->valid = false;
86             IPT[indexSWAPFIFO]->physicalPage = -1;
87             MemBitMap->Clear( oldPhysicalPage );
88             //clearPhysicalPage( oldPhysicalPage );
89
90             // cargamos pagina nueva en memoria
91             freeFrame = MemBitMap->Find();
92             if ( freeFrame == -1 )
93             {
94                 printf("Invalid free frame %d\n",
95                     freeFrame );
96                 ASSERT( false );
97             }
98             //++stats->numPageFaults;
99             pageTable[ vpn ].physicalPage = freeFrame;
100             executable->ReadAt (&(machine->mainMemory[ (
101                 freeFrame * PageSize ) ] ),
102                 PageSize, noffH.code.inFileAddr + PageSize*vpn
103                 );
104             pageTable[ vpn ].valid = true;
105             IPT[ freeFrame ] = &(pageTable [ vpn ]);
106             int tlbSPace = getNextSCTLB();
107             useThisTLBIndex( tlbSPace, vpn );
108         }
109         //ASSERT(false);
110     }
111 }
112
113 else if(vpn >= initData && vpn < noInitData){ //segmento de Datos
114     Inicializados.
115     //Se debe cargar la pagina del archivo ejecutable.
116     DEBUG('v', "\t1.2 Pagina de datos Inicializados\n");
117     freeFrame = MemBitMap->Find();
118
119     if ( freeFrame != -1 )
120     {
121         DEBUG('v',"Frame libre en memoria: %d\n", freeFrame );
122         //++stats->numPageFaults;
123         pageTable[ vpn ].physicalPage = freeFrame;
124         executable->ReadAt (&(machine->mainMemory[ ( freeFrame *
125             PageSize ) ] ),
126             PageSize, noffH.code.inFileAddr + PageSize*vpn );
127         pageTable[ vpn ].valid = true;
128
129         //Se actualiza la TLB invertida
130         IPT[freeFrame] = &(pageTable[ vpn ]);
131         //Se debe actualizar el TLB
132         int tlbSPace = getNextSCTLB();
133         useThisTLBIndex( tlbSPace, vpn );
134     }
135 else
136 {

```

```

127 //Se debe seleccionar una victima para enviar al SWAP
128 indexSWAPFIFO = getNextSCSWAP();
129 updateSwapVictimInfo( indexSWAPFIFO );
130
131 bool victimDirty = IPT[indexSWAPFIFO]->dirty;
132 // revisar si la victima est sucia
133 if ( victimDirty )
134 {
135     //si s
136     //env arla al swap
137     DEBUG('v',"\\t\\t\\tVictima f=%d,l=%d y
        sucia\\n",IPT[indexSWAPFIFO]->
        physicalPage, IPT[indexSWAPFIFO]->
        virtualPage );
138     writeIntoSwap( IPT[indexSWAPFIFO]->
        physicalPage );
139     //pedir el nuevo freeFrame
140     freeFrame = MemBitMap->Find();
141     //validar ese nuevo freeFrame
142     if ( -1 == freeFrame )
143     {
144         printf("Invalid free frame %d\\n
            ", freeFrame );
145         ASSERT( false );
146     }
147     // asignar al pageTable[vpn] es
        freeFrame
148     pageTable[ vpn ].physicalPage =
        freeFrame;
149     // leer del archivo ejecutable
150     executable->ReadAt (&(machine->
        mainMemory[ ( freeFrame * PageSize
        ) ] ),
151     PageSize, noffH.code.inFileAddr +
        PageSize*vpn );
152     //poner valida la paginas
153     pageTable[ vpn ].valid = true;
154     //actualiza la tabla de paginas
        invertidas
155     IPT[ freeFrame ] = &( pageTable[ vpn ]
        );
156     //hacer la actualizacin en la tlb
157     int tlbSPace = getNextSCTLB();
158     useThisTLBIndex( tlbSPace, vpn );
159 }else
160 {
161     //si no esta sucia
162     DEBUG('v',"\\t\\t\\tVictima f=%d,l=%d y limpia\\n",
        IPT[indexSWAPFIFO]->physicalPage, IPT[
        indexSWAPFIFO]->virtualPage );
163     // rescatar la antigua fisica de la
        victima
164     int oldPhysicalPage = IPT[indexSWAPFIFO
        ]->physicalPage;
165     MemBitMap->Clear( oldPhysicalPage );
166     // poner a la victima en valid = false

```



```

167 IPT[indexSWAPFIFO]->valid = false;
168 //ponerle la pagina fisica en -1
169 IPT[indexSWAPFIFO]->physicalPage = -1;
170 // pedir el nuevo freeframe
171 freeFrame = MemBitMap->Find();
172 //validar ese nuevo freeFrame
173 if ( freeFrame == -1 )
174 {
175     printf("Invalid free frame %d\n", freeFrame );
176     ASSERT( false );
177 }
178 //asignar ese freeFrame al pageTable[
    vpn]
179 pageTable[ vpn ].physicalPage =
    freeFrame;
180 //leer del archivo ejecutable
181 executable->ReadAt (&(machine->
    mainMemory[ ( freeFrame * PageSize
    ) ] ),
182 PageSize, noffH.code.inFileAddr +
    PageSize*vpn );
183 //valida dicha pageTable[vpn]
184 pageTable[ vpn ].valid = true;
185 //actualizar tabla de paginas
    invertidas
186 IPT[ freeFrame ] = &(pageTable [ vpn ])
    ;
187 // actualizar la tlp
188 int tlbSPace = getNextSCTLB();
189 useThisTLBIndex( tlbSPace, vpn );
190 }
191 //ASSERT(false);
192 }
193 }
194 else if(vpn >= noInitData && vpn < numPages){ //segmento de Datos No
    Inicializados o segmento de Pila.
195     DEBUG('v',"t1.3 P gina de datos no Inicializado o p gina de
        pila\n");
196     freeFrame = MemBitMap->Find();
197     DEBUG('v',"t\t\tSe busca una nueva p gina para otorgar\n" );
198     if ( freeFrame != -1 )
199     {
200         //DEBUG('v', "Se le otorga una nueva p gina en memoria
            \n" );
201         pageTable[ vpn ].physicalPage = freeFrame;
202         pageTable[ vpn ].valid = true;
203
204         //Se actualiza la TLB invertida
205         //clearPhysicalPage(freeFrame);
206         IPT[freeFrame] = &(pageTable[ vpn ]);
207
208         //Se debe actualizar el TLB
209         int tlbSPace = getNextSCTLB();
210         useThisTLBIndex( tlbSPace, vpn );
211     }else{

```

```

212 // usar swap
213 indexSWAPFIFO = getNextSCSWAP();
214 updateSwapVictimInfo( indexSWAPFIFO );
215
216 bool victimDirty = IPT[indexSWAPFIFO]->dirty;
217 // revisamos con la informaci n actualizada a la
    victima
218 if ( victimDirty )
219 {
220     DEBUG('v',"\\t\\t\\tVictima f=%d,l=%d sucia\\n",IPT
        [indexSWAPFIFO]->physicalPage, IPT[
        indexSWAPFIFO]->virtualPage );
221     writeIntoSwap( IPT[indexSWAPFIFO]->physicalPage
        );
222
223     // pedir el nuevo freeFrame
224     freeFrame = MemBitMap->Find();
225
226     // verificar que sea distinto de -1
227     if ( -1 == freeFrame )
228     {
229         printf("Invalid free frame %d\\n",
            freeFrame );
230         ASSERT( false );
231     }
232
233     // actualizar la pagina f sica para la nueva
        virtual vpn
234     pageTable [ vpn ].physicalPage = freeFrame;
235     pageTable [ vpn ].valid = true;
236
237     //actualizo invertida
238     IPT[ freeFrame ] = &(pageTable [ vpn ]);
239
240     //actualizo el tlb
241     int tlbSpace = getNextSCTLB();
242     useThisTLBIndex( tlbSpace, vpn );
243     //ASSERT(false);
244 } else
245 {
246     DEBUG('v',"\\t\\t\\tVictima f=%d,l=%d limpia\\n",
        IPT[indexSWAPFIFO]->physicalPage, IPT[
        indexSWAPFIFO]->virtualPage );
247     int oldPhysicalPage = IPT[indexSWAPFIFO]->
        physicalPage;
248     IPT[indexSWAPFIFO]->valid = false;
249     IPT[indexSWAPFIFO]->physicalPage = -1;
250     MemBitMap->Clear( oldPhysicalPage );
251     clearPhysicalPage( oldPhysicalPage );
252
253     // cargamos pagina nueva en memoria
254     freeFrame = MemBitMap->Find();
255
256     if ( freeFrame == -1 )
257     {
258         DEBUG('v',"Invalid free frame %d\\n",

```

```

259         freeFrame );
260         ASSERT( false );
261     }
262     pageTable [ vpn ].physicalPage = freeFrame;
263     pageTable [ vpn ].valid = true;
264
265     IPT[ freeFrame ] = &(pageTable [ vpn ]);
266     // finalmente se actualiza la tlb
267     int tlbSPace = getNextSCTLB();
268     useThisTLBIndex( tlbSPace, vpn );
269 }
270 //ASSERT(false);
271 }
272 else{
273     printf("%s %d\n", "Algo muy malo paso, el numero de pagina
274         invalido!", vpn);
275     ASSERT(false);
276 }
277 // Se cierra el Archivo
278 delete executable;
279 }
280 //Si la pagina no es valida y esta sucia.
281 else if(!pageTable[vpn].valid && pageTable[vpn].dirty){
282     //Debo traer la pagina del area de SWAP.
283     DEBUG('v', "\t2- Pagina invalida y sucia\n");
284     DEBUG('v', "\t\tPagina f sica: %d, pagina virtual= %d\n", pageTable[
285         vpn].physicalPage, pageTable[vpn].virtualPage );
286     freeFrame = MemBitMap->Find();
287     if(freeFrame != -1)
288     { //Si hay espacio en memoria
289         DEBUG('v', "%s\n", "Si hay espacio en memoria, solo leemos de
290             SWAP\n" );
291
292         //actualizar su pagina f sica de la que leo del swap
293         int oldSwapPageAddr = pageTable [ vpn ].physicalPage;
294         pageTable [ vpn ].physicalPage = freeFrame;
295         // cargarla
296         readFromSwap( freeFrame, oldSwapPageAddr );
297         // actualizar tambien su validez
298         pageTable [ vpn ].valid = true;
299         // actualiza tabla de paginas invertidas
300         IPT[ freeFrame ] = &(pageTable [ vpn ]);
301         //actualizar su posici n en la tlb
302         int tlbSPace = getNextSCTLB();
303         useThisTLBIndex( tlbSPace, vpn );
304         //ASSERT(false);
305     }
306     else
307     {
308         //Se debe seleccionar una victima para enviar al SWAP
309         DEBUG('v', "\n%s\n", "NO hay memoria, es m s complejo.\n" );
310         indexSWAPFIFO = getNextSCSWAP();
311         updateSwapVictimInfo( indexSWAPFIFO );
312
313         bool victimDirty = IPT[indexSWAPFIFO]->dirty;

```

```

311         if ( victimDirty )
312         {
313             DEBUG('v',"\\t\\t\\tVictima f=%d,l=%d sucia\\n",IPT[
                indexSWAPFIFO]->physicalPage, IPT[indexSWAPFIFO]->
                virtualPage );
314             writeIntoSwap( IPT[indexSWAPFIFO]->physicalPage );
315             // pido el freeFrame
316             freeFrame = MemBitMap->Find();
317             if ( freeFrame == -1 )
318             {
319                 printf("Invalid free frame %d\\n", freeFrame );
320                 ASSERT( false );
321             }
322
323             int oldSwapPageAddr = pageTable [ vpn ].physicalPage;
324             pageTable [ vpn ].physicalPage = freeFrame;
325             // cargamos la pagina que desea desde el SWAP
326             readFromSwap( freeFrame, oldSwapPageAddr );
327             pageTable [ vpn ].valid = true;
328             IPT[ freeFrame ] = &(pageTable [ vpn ]);
329
330             // finalmente actualizacom tlb
331             int tlbSPace = getNextSCTLB();
332             useThisTLBIndex( tlbSPace, vpn );
333             //ASSERT(false);
334         }else
335         {
336             DEBUG('v',"\\t\\t\\tVictima f=%d,l=%d limpia\\n",IPT[
                indexSWAPFIFO]->physicalPage, IPT[indexSWAPFIFO]->
                virtualPage );
337             int oldPhysicalPage = IPT[indexSWAPFIFO]->physicalPage;
338             IPT[indexSWAPFIFO]->valid = false;
339             IPT[indexSWAPFIFO]->physicalPage = -1;
340             MemBitMap->Clear( oldPhysicalPage );
341             //clearPhysicalPage( oldPhysicalPage );
342             freeFrame = MemBitMap->Find();
343             if ( freeFrame == -1 )
344             {
345                 printf("Invalid free frame %d\\n", freeFrame );
346                 ASSERT( false );
347             }
348             int oldSwapPageAddr = pageTable [ vpn ].physicalPage;
349             pageTable [ vpn ].physicalPage = freeFrame;
350             // cargamos la pagina que desea desde el SWAP
351             readFromSwap( freeFrame, oldSwapPageAddr );
352             pageTable [ vpn ].valid = true;
353             IPT[ freeFrame ] = &(pageTable [ vpn ]);
354             // finalmente actualizacom tlb
355             int tlbSPace = getNextSCTLB();
356             useThisTLBIndex( tlbSPace, vpn );
357         }
358     }
359     //ASSERT(false);
360 }
361 //Si la pagina es valida y no esta sucia.
362 else if( pageTable[vpn].valid && !pageTable[vpn].dirty ){

```

```

363         DEBUG('v', "\t3- Pagina valida y limpia\n");
364         //La pagina ya esta en memoria por lo que solamente debo actualizar el
            TLB.
365         int tlbSPace = getNextSCTLB();
366         useThisTLBIndex( tlbSPace, vpn );
367     }
368     //Si la pagina es valida y esta sucia.
369     else{
370         DEBUG('v', "\t4- Pagina valida y sucia\n");
371         //La pagina ya esta en memoria por lo que solamente debo actualizar el
            TLB.
372         int tlbSPace = getNextSCTLB();
373         useThisTLBIndex( tlbSPace, vpn );
374     }
375 }

```

Como se puede ver "load" es un método bastante grande, a continuación se muestra y explican cuatro de los métodos más importantes a los que llama "load".

writeIntoSwap y readFromSwap:

```

1  void AddrSpace::writeIntoSwap( int physicalPageVictim ){
2      int swapPage = SWAPBitMap->Find();
3      if ( physicalPageVictim < 0 || physicalPageVictim >= NumPhysPages )
4      {
5          DEBUG( 'v', "Error(writeIntoSwap): Direccion fisica de memoria
            inv lida: %d\n", physicalPageVictim );
6          ASSERT( false );
7      }
8      DEBUG('h', "\t\t\t\tSe escribe en el swap en la posici n: %d\n",swapPage );
9      if ( swapPage == -1 )
10     {
11         DEBUG( 'v', "Error(writeIntoSwap): Espacio en SWAP NO disponible\n");
12         ASSERT( false );
13     }
14     OpenFile *swapFile = fileSystem->Open( SWAPFILENAME );
15     if( swapFile == NULL ){
16         DEBUG( 'v', "Error(writeIntoSwap): No se pudo habir el archivo de SWAP\
            n");
17         ASSERT(false);
18     }
19     IPT[physicalPageVictim]->valid = false;
20     IPT[physicalPageVictim]->physicalPage = swapPage;
21     swapFile->WriteAt ( (&machine->mainMemory[physicalPageVictim*PageSize]),PageSize,
        swapPage*PageSize);
22     MemBitMap->Clear( indexSWAPFIFO );
23     //clearPhysicalPage( indexSWAPFIFO );
24     //++stats->numDiskWrites;
25     delete swapFile;
26 }
27
28 void AddrSpace::readFromSwap( int physicalPage , int swapPage ){
29     DEBUG('h', "\t\t\t\tSe lee en el swap en la posici n: %d\n",swapPage );
30     //SWAPBitMap->Print();
31     SWAPBitMap->Clear(swapPage);
32     //SWAPBitMap->Print();
33     OpenFile *swapFile = fileSystem->Open(SWAPFILENAME);

```

```

34     if ( (swapPage >=0 && swapPage < SWAPSize) == false )
35     {
36         DEBUG( 'v',"readFromSwap: invalid swap position = %d\n",
37             swapPage );
38         ASSERT( false );
39     }
40     if( swapFile == NULL ){
41         DEBUG( 'v', "Error(writeIntoSwap): No se pudo habir el archivo de SWAP\
42             n");
43         ASSERT(false);
44     }
45     if ( physicalPage < 0 || physicalPage >= NumPhysPages )
46     {
47         DEBUG( 'v', "Error(readFromSwap): Direccion fisica de memoria
48             inv lida: %d\n", physicalPage );
49         ASSERT( false );
50     }
51     swapFile->ReadAt( (&machine->mainMemory[physicalPage*PageSize]), PageSize,
52         swapPage*PageSize);
53     ++stats->numPageFaults;
54     //++stats->numDiskReads;
55     delete swapFile;
56 }

```

Como sus nombres lo indican son los métodos utilizados para escribir de memoria al SWAP y de SWAP a memoria(respectivamente). El área de SWAP corresponde a un archivo (SWAP.txt) en la carpeta "VM" de NachOS, por lo que para leer y escribir en el SWAP se llaman a los métodos ReadAt y WriteAt de la clase OpenFile que es el tipo de objeto con el que se representa el archivo de SWAP en el programa.

Para la selección de la pagina del TLB a la cual remplazar cuando este se llena se implemento el algoritmo second chance visto en clase, como se muestra adelante:

```

1  int AddrSpace::getNextSCTLB()
2  {
3      int freeSpace = -1;
4      // para una primera pasada
5      for ( int x = 0; x < TLBSize; ++x )
6      {
7          if ( machine->tlb[ x ].valid == false )
8          {
9              return x;
10         }
11     }
12     // si llega aquí ya no es la primera pasada
13     bool find = false;
14     while ( find == false )
15     {
16         if ( machine->tlb[ indexTLBSndChc ].use == true )
17         {
18             machine->tlb[ indexTLBSndChc ].use = false;
19             saveVictimTLBInfo( indexTLBSndChc, true );
20         }else
21         {
22             find = true;
23             freeSpace = indexTLBSndChc;

```

```

24         saveVictimTLBInfo( freeSpace, false );
25     }
26     indexTLBSndChc = (indexTLBSndChc+1) % TLBSize;
27 }
28 if ( freeSpace < 0 || freeSpace >= TLBSize )
29 {
30     DEBUG('v', "\ngetNextSCTLB: Invalid tlb information\n");
31     showTLBState();
32     ASSERT( false );
33 }
34 return freeSpace;
35 }

```

De la misma manera para el algoritmo de selección de víctima de memoria principal para determinar la pagina a desechar o enviar al SWAP cuando esta esta llena, se utilizo el mismo algoritmo second chance visto en clase, de la siguiente manera:

```

1 int AddrSpace::getNextSCSWAP()
2 {
3     if ( indexSWAPSndChc < 0 || indexSWAPSndChc >= NumPhysPages )
4     {
5         DEBUG('v', "getNextSCSWAP:: Invalid indexSWAPSndChc value = %d \n",
6             indexSWAPSndChc );
7         ASSERT( false );
8     }
9     int freeSpace = -1;
10    bool find = false;
11
12    while ( find == false )
13    {
14        if ( IPT[ indexSWAPSndChc ] == NULL )
15        {
16            DEBUG('v', "\ngetNextSCSWAP:: Invalid IPT state\n");
17            showIPTState();
18            ASSERT( false );
19        }
20
21        if ( IPT[ indexSWAPSndChc ]->valid == false )
22        {
23            DEBUG('v', "\ngetNextSCSWAP:: Invalid IPT[%d].valid values, is
24                false\n");
25            showIPTState();
26            ASSERT( false );
27        }
28
29        if ( IPT[ indexSWAPSndChc ]->use == true )
30        {
31            IPT[ indexSWAPSndChc ]->use = false;
32        } else
33        {
34            freeSpace = indexSWAPSndChc;
35            find = true;
36        }
37        indexSWAPSndChc = (indexSWAPSndChc+1) % NumPhysPages;
38    }
39 }

```

```

38     if ( freeSpace < 0 || freeSpace >= NumPhysPages )
39     {
40         DEBUG('v', "\ngetNextSCSWAP: Invalid IPT information\n");
41         showIPTState();
42         ASSERT( false );
43     }
44     return freeSpace;
45 }

```

El algoritmo second chance utilizado tanto para seleccionar la pagina víctima en la TLB como en memoria principal funciona de la siguiente manera: Se busca la pagina víctima como la primera pagina con la bandera "used" = false, empezando por la posición actual del cursor, si la pagina actual tiene used = false, esta es la víctima y me detengo. Si no le coloco used = false y me muevo a la siguiente pagina.

En este proyecto para el manejo de la memoria principal se siguió el consejo del profesor de utilizar un page table invertido, que asocia una pagina fisica de memoria principal con su respectiva pagina y page table. La page table invertida que se utilizó se llama "IPT" y esta definida en el archivo "system.h" de la carpeta "threads" de NachOS y es de tipo TranslationEntry\*.



## 8. 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
```

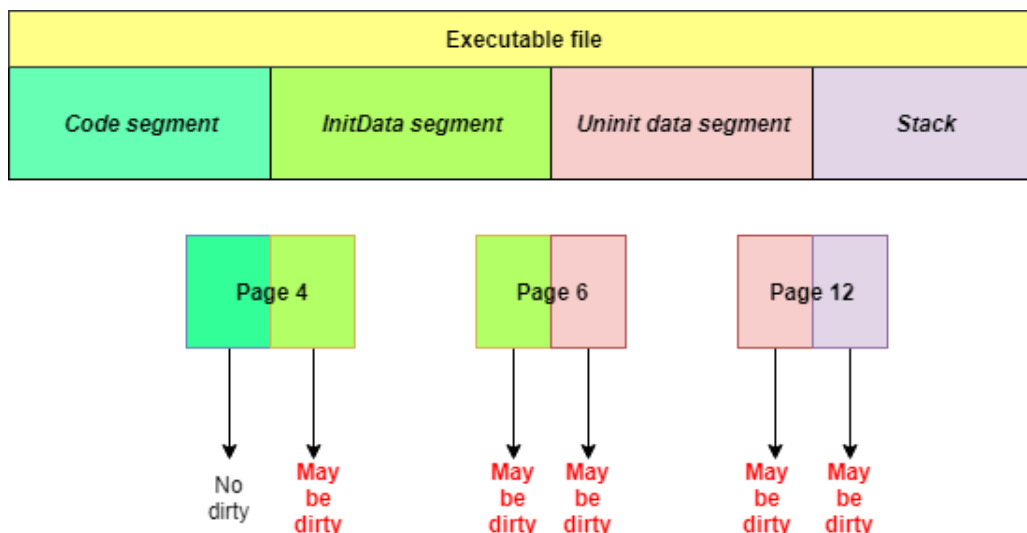
Finalmente se pueden probar las implementaciones de los diferentes system calls y del manejo de memoria, corriendo programas de usuario para NachOS de la siguiente manera:

```
./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

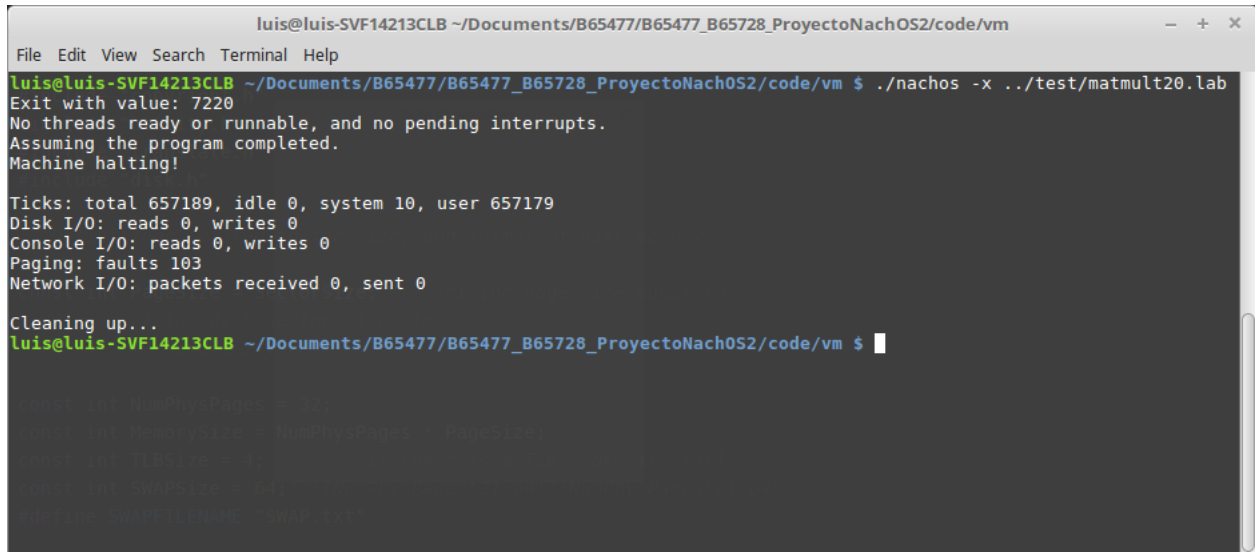
La siguiente imagen explica un problema que se encontró durante la realización del proyecto que no pudo ser resuelto. Como se puede ver sucede que como los segmento de un programa no son exactamente de 128 bytes o múltiplos de 128 bytes, cuando uno los calcula y divide en paginas, en una misma pagina pueden quedar pedazos de dos segmentos (código y datos inicializados por poner un ejemplo). Lo cual causa problemas como que se pueda .<sup>en</sup>suciar una pagina de código o que aumenten el numero de escrituras al disco.



## 9. Casos de Prueba

### Prueba 1: Ejecución de matmult20.lab con 32 paginas de memoria física

```
./nachos -x ../test/matmult20.lab
```



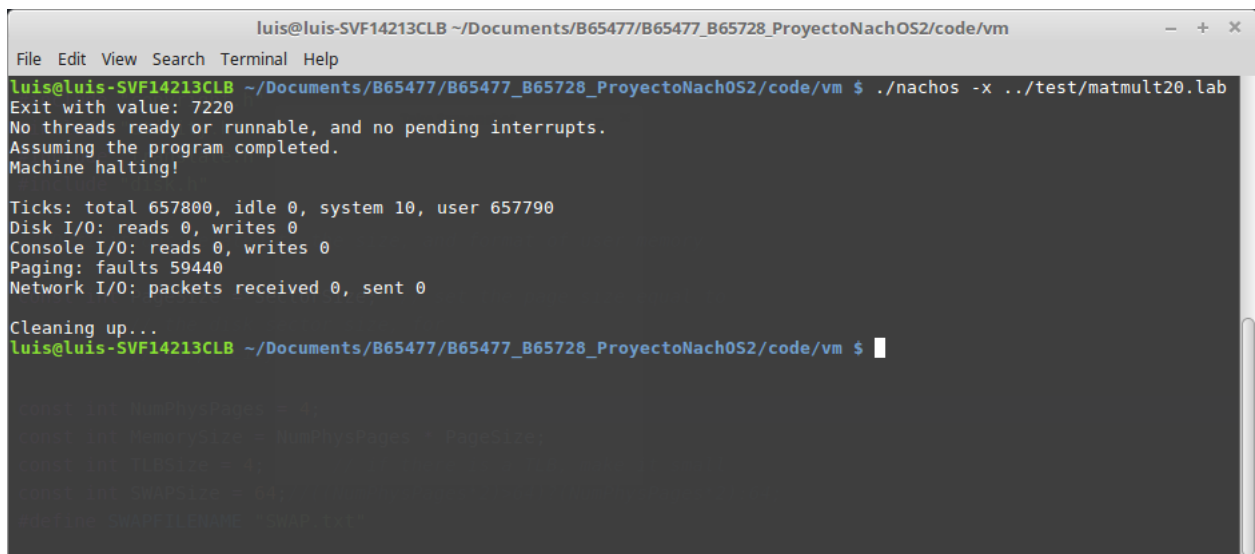
```
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm
File Edit View Search Terminal Help
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm $ ./nachos -x ../test/matmult20.lab
Exit with value: 7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 657189, idle 0, system 10, user 657179
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 103
Network I/O: packets received 0, sent 0

Cleaning up...
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm $
```

### Prueba 2: Ejecución de matmult20.lab con 4 paginas de memoria física

```
./nachos -x ../test/matmult20.lab
```



```
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm
File Edit View Search Terminal Help
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm $ ./nachos -x ../test/matmult20.lab
Exit with value: 7220
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 657800, idle 0, system 10, user 657790
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 59440
Network I/O: packets received 0, sent 0

Cleaning up...
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm $
```

### Prueba 3: Ejecución de sort-ok con 32 paginas de memoria física

```
./nachos -x ../test/sort-ok
```

```
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm
File Edit View Search Terminal Help
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm $ ./nachos -x ../test/sort-ok
Exit with value: 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 36307259, idle 0, system 10, user 36307249
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 4295
Network I/O: packets received 0, sent 0

Cleaning up...
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm $
```

#### Prueba 4: Ejecución de sort-ok con 4 paginas de memoria física

```
./nachos -x ../test/sort-ok
```

```
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm
File Edit View Search Terminal Help
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm $ ./nachos -x ../test/sort-ok
Exit with value: 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
Ticks: total 36708783, idle 0, system 10, user 36708773
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 2619293
Network I/O: packets received 0, sent 0
Cleaning up...
luis@luis-SVF14213CLB ~/Documents/B65477/B65477_B65728_ProyectoNachOS2/code/vm $
```

Diagram illustrating the execution of the sort-ok test with 4 physical memory pages. The diagram shows the execution file structure and the mapping of pages to physical memory.

**Execution File Structure:**

- Code segment
- Initial data segment
- Initial data segment
- Initial data segment

**Page Mapping:**

- Page 4: No dirty
- Page 8: No dirty
- Page 12: No dirty

**Physical Memory:**

- Page 4: No dirty
- Page 8: No dirty
- Page 12: No dirty

**Test Execution:**

- Prueba 2: Ejecución de sort-ok
- Execution file
- Code segment
- Initial data segment
- Initial data segment
- Initial data segment