# Practica 1 Sistemas Operativos II

## Ejercicio 1:

En machine/mmu.hh

```
MEMORY_SIZE = NUM_PHYS_PAGES * PAGE_SIZE;
PAGE_SIZE = SECTOR_SIZE;
NUM_PHYS_PAGES = 32;
```

Y en machine/disk.hh

```
 SECTOR_SIZE = 128;
```

Luego

```
32*128 = 4096
```

## Ejercicio 2:

Aumentando la cantidad de paginas fisicas o el tamaño de las paginas.

## Ejercicio 3:

En *machine/disk.cc*

```
MAGIC_SIZE = sizeof (int);
DISK_SIZE = MAGIC_SIZE + NUM_SECTORS * SECTOR_SIZE;
```

Y en *machine/disk.hh*

```
NUM_SECTORS = SECTORS_PER_TRACK * NUM_TRACKS;
NUM_TRACKS = 32;
SECTOR_SIZE = 128;
SECTORS_PER_TRACK = 32;
```

Luego

```
4 + (32*32) * 128 = 131072 + 4
```

## Ejecticio 4:

```
enum {
    OP_ADD      =  1,
    OP_ADDI     =  2,
    OP_ADDIU    =  3,
    OP_ADDU     =  4,
    OP_AND      =  5,
    OP_ANDI     =  6,
    OP_BEQ      =  7,
    OP_BGEZ     =  8,
    OP_BGEZAL   =  9,
    OP_BGTZ     = 10,
    OP_BLEZ     = 11,
    OP_BLTZ     = 12,
    OP_BLTZAL   = 13,
    OP_BNE      = 14,

    OP_DIV      = 16,
    OP_DIVU     = 17,
    OP_J        = 18,
    OP_JAL      = 19,
    OP_JALR     = 20,
    OP_JR       = 21,
    OP_LB       = 22,
    OP_LBU      = 23,
    OP_LH       = 24,
    OP_LHU      = 25,
    OP_LUI      = 26,
    OP_LW       = 27,
    OP_LWL      = 28,
    OP_LWR      = 29,

    OP_MFHI     = 31,
    OP_MFLO     = 32,

    OP_MTHI     = 34,
    OP_MTLO     = 35,
    OP_MULT     = 36,
    OP_MULTU    = 37,
    OP_NOR      = 38,
    OP_OR       = 39,
    OP_ORI      = 40,
    OP_RFE      = 41,
    OP_SB       = 42,
    OP_SH       = 43,
    OP_SLL      = 44,
    OP_SLLV     = 45,
    OP_SLT      = 46,
    OP_SLTI     = 47,
    OP_SLTIU    = 48,
```

```
    OP_SLTU     = 49,
    OP_SRA      = 50,
    OP_SRAV     = 51,
    OP_SRL      = 52,
    OP_SRLV     = 53,
    OP_SUB      = 54,
    OP_SUBU     = 55,
    OP_SW       = 56,
    OP_SWL      = 57,
    OP_SWR      = 58,
    OP_XOR      = 59,
    OP_XORI     = 60,
    OP_SYSCALL  = 61,

    OP_UNIMP    = 62,
    OP_RES      = 63,

    MAX_OPCODE  = 63
};
```

NachOS simula 60 instrucciones MIPS.

## Ejecticio 5:

```
case OP_ADD:
            sum = registers[instr->rs] + registers[instr->rt];
            if (!((registers[instr->rs] ^ registers[instr->rt]) & SIGN_BIT)
                  && (registers[instr->rs] ^ sum) & SIGN_BIT) {
                RaiseException(OVERFLOW_EXCEPTION, 0);
                return;
            }
            registers[instr->rd] = sum;
            break;
```

Primero, realiza la suma de los dos registros y la guarda en una variable auxiliar. Luego:

```
!((registers[instr->rs] ^ registers[instr->rt]) & SIGN_BIT)
```

chequea si los argumentos tienen el mismo signo. Y

```
(registers[instr->rs] ^ sum) & SIGN_BIT
```

chequea si el resultado tiene un signo diferente. Si ambas condiciones son verdaderas, esto indica que se produjo overflow, por lo que lanza una excepcion indicando esto. En caso de que no haya overflow, guarda el resultado en el registro correspondiente.

# Ejecticio 6:

## Primer Nivel:

**En main:**

1. Initialize: implementado en /code/threads/system.cc
2. DEBUG: definido en /code/lib/utility.hh
3. strcmp: implementado en string.h
4. PrintVersion: implementado en /code/threads/main.cc
5. ThreadTest: implementado en /code/threads/thread_test.cc
6. Thread::Finish: implementado en /code/threads/thread.cc

## Segundo Nivel:

**En Initialize:**

1. ASSERT: definido en /code/lib/utility.hh
2. strcmp: implementado en string.h
3. RandomInit: implementado en /code/machine/system_dep.cc
4. atoi: implementado en stdlib.h
5. Debug::SetFlags: implementado en /code/lib/debug.cc
6. Timer::Timer: implementado en /code/machine/timer.cc
7. Thread::Thread: implementado en /code/threads/thread.cc
8. Thread::SetStatus: implementado en /code/threads/thread.cc
9. Interrupt::Enable: implementado en /code/machine/interrupt.cc
10. CallOnUserAbort: implementado en /code/machine/system_dep.cc
11. PreemptiveScheduler::PreemptiveScheduler: definido en /code/threads/preemptive.hh
12. PreemptiveScheduler::SetUp: implementado en /code/threads/preemptive.cc

**En DEBUG:**

1. Debug::Print: implementado en /code/lib/debug.cc

**En PrintVersion:**

1. pirntf: implementado en stdio.h

**En ThreadTest:**

1. DEBUG: definido en /code/lib/utility.hh
2. strncpy: implementado en string.h
3. Thread::Thread: implementado en /code/threads/thread.cc
4. Thread::Fork: implementado en /code/threads/thread.cc
5. SimpleThread: implementado en /code/threads/thread_test.cc

**En Thread::Finish:**

1. Interrupt::SetLevel: implementado en /code/machine/interrupt.cc

2. ASSERT: definido en /code/lib/utility.hh
3. DEBUG: definido en /code/lib/utility.hh
4. Thread::GetName: implementado en /code/threads/thread.cc
5. Thread::Sleep: implementado en /code/threads/thread.cc

# Ejecticio 7:

Son varias las razones por las que se prefiere emular una CPU en lugar de usar la existente.

La primera es que el núcleo que estamos construyendo requiere un control completo del manejo de la memoria y de las interrupciones y excepciones (incluyendo las llamadas al sistema). Si usáramos la CPU existente, no tendríamos este acceso ya que tendríamos como intermediario el sistema operativo de nuestra computadora.

Por otro lado, si usáramos la PC real, nuestro núcleo debería poder compilarse en cualquier arquitectura. De lo contrario, no podríamos correrlo en cualquier computadora. Usando la máquina virtual, hacemos el núcleo para la arquitectura que esta posee y podemos correrlo en cualquier PC.

Por último, usar la CPU emulada facilita el testeo del código con GDB.

# Ejecticio 8:

## ASSERT:

ASSERT toma una condición y la checkea. Si esta es verdadera, no hace nada y sigue el flujo normal del programa. Si falla, avisa por pantalla que falló un assert, muestra la expresión y en qué archivo y línea se encuentra, y finalmente interrumpe la ejecución. Sirve para asegurar que se cumplan las condiciones que deberían darse para realizar cierta acción.

## DEBUG:

DEBUG llama al método Debug::Print. Este toma una bandera, un puntero al formato de lo que se desea imprimir y los argumentos que se van a imprimir. Si la bandera está habilitada (se habilitan cuando se llama al programa), entonces imprime el mensaje que se pasó por argumento. Sirve para que nosotros elijamos qué información queremos que nos muestre durante la ejecución utilizando las diferentes banderas.

# Ejecticio 9:

Las banderas de depuración predefinidas son:

- `+` : Habilita todos los mensajes de depuración.
- `t` : Mensajes del sistema de threads.
- `s` : Mensajes de semáforos, locks y condiciones.
- `i` : Mensajes de la simulación de interrupciones.
- `m` : Mensajes de la simulación de la máquina (requiere *USER_PROGRAM*).
- `d` : Mensajes de la simulación del disco (requiere *FILESYS*).

- `f` : Mensajes sobre el sistema de archivos (requiere *FILESYS*).
- `a` : Mensajes sobre los espacios de direcciones (requiere *USER_PROGRAM*).
- `n` : Mensajes de la simulación de red (requiere *NETWORK*).

## Ejecticio 10:

Las constantes estan definidas en los distintos Makefile para incluir distintos módulos. Marcamos con una *X* en los que se incluyen:

| | **USER_PROGRAM** | **FILESYS_NEEDED** | **FILESYS_STUB** | **NETWORK** |
|---|---|---|---|---|
| filesys | *X* | *X* | | |
| network | *X* | *X* | | *X* |
| userprog | *X* | *X* | *X* | |
| vmem | *X* | *X* | *X* | |

## Ejecticio 11:

List es una implementación de lista enlazada con prioridad, los elementos de List son del tipo ListElement que se encarga de un solo item de la lista.

SynchList es una lista sincronizada, es decir una List que tiene las siguientes restricciones:

```
1- Si un thread que intenta remover un item de List, va a esperar hasta que
la lista tenga un elemento en ella.
2- Sólo un thread a la vez puede acceder a la estructura de lista.
```

## Ejecticio 12:

Podemos encontrar definida la función *main* en:

- code/bin/coff2flat.c:
- code/bin/coff2noff.c:
- code/bin/disasm.c:
- code/bin/main.c:
- code/bin/out.c:
- code/bin/readnoff.c:
- code/bin/fuse/nachosfuse.c:
- code/userland/filetest.c:

- code/userland/halt.c:
- code/userland/matmult.c:
- code/userland/shell.c:
- code/userland/sort.c:
- code/userland/tiny_shell.c:

Podemos ver que el *main* del ejecutable *nachos* de *userprog* esta definida en: *code/threads/main.cc*.

Al inspeccionar *code/userprog/Makefile.depends* vemos:

```
main.o: ../threads/main.cc
```

REVISAR ESTA CONCLUSION !!!!!!!!!!1

## Ejecticio 13:

Nachos soporta las siguentes lineas de comandos:

- `-d`: Imprime algunos mensajes de depuración.
- `-p`: Habilita la multitarea preventiva para los threads del kernel.
- `-rs`: Hace que ocurran `Yield` en lugares aleatorios.
- `-z`: Imprime información sobre versión y copyrights.
- `-s`: Hace que los programas de usuarios se ejecuten en modo paso-simple.
- `-x`: Ejecuta programa de usuario.
- `-tc`: Testea la consola.
- `-f`: Formatea el disco físico.
- `-cp`: Copia archivo de UNIX a Nachos.
- `-pr`: Imprime un archivo de Nachos a la salida estandar.
- `-rm`: Elimina un archivo de Nachos del sistema.
- `-ls`: Lista el contenido de el directorio de Nachos.
- `-D`: Imprime el contenido de todo el sistema de archivos.
- `-tf`: Testea la performance del sistema de archivos de Nachos.
- `-n`: Establece la fiabilidad de la red.
- `-id`: Establece el host id de la maquina (necesaria para la red).

- `-tn`: Ejecuta un test simple sobre el software de red de Nachos.

La opcion `-rs` produce que ocurran `Yield` en lugares aleatorios.

## Ejecticio 14:

## Ejecticio 15:

## Ejecticio 16: