M1 MOSIG
Project 4 : NachOS Virtual Memory

Vincent Danjean, Guillaume Huard, Arnaud Legrand,
Vania Marangozova-Martin, Jean-François Méhaut

The purpose of this project is to implement in NachOS a multiple process model, like in Linux. Each process should be able to launch multiple threads. In fact, you should implement a mechanism for managing virtual memory using paging.

Partie I. Paging and Virtual Addresses

The MIPS machine uses vurtual addresses implemented using either a *page table* or a *TLB*. In this project we will focus on the page table mechanism.

Look how this pages table is used in `userprog/addrspace.h`. A virtual address is composed of a page number and an offset within the page. The table associates a physical frame number (also referred to as a page) with each virtual page number. The (physical) address of the frame and the offset within the page determine the address physically accessed. The mechanism is implemented by the function :

```
Translate(int virtAddr, int* physAddr, int size, bool writing)
```

in `machine/translate.cc`. All memory accesses within the interpreter are done via this function. Observe the behavior and the source code of `WriteMem` and `ReadMem`.

Action I.1. *Write a small test program `test/userpages0` that starts one or two user threads, and that writes a few interlaced characters to screen.*

Action I.2. *Examinethe operation of `executable->ReadAt` in `AddrSpace::AddrSpace` of `userprog/addrspace.cc`. Curieusly (?), this function writes directly to physical memory. How can you see that?*

Action I.3. *Define a new* local *function*

```
static void ReadAtVirtual(OpenFile *executable, int virtualaddr,
                          int numBytes, int position,
                          TranslationEntry *pageTable,unsigned numPages)
```

that does the same thing as `ReadAt`, but writes to the virtal space defined by `pageTable` and `numPages`. You can use a temporary buffer, that you will fill with `ReadAt`, and then you will then re-copy into memory with `WriteMem`, for example...

Use `ReadAtVirtual` instead of `ReadAt`, and check that all user programs still work.

Action I.4. *Edit the pages table so that virtual page i is mapped onto the physical frame $i + 1$.*
Restart your program. Everything should work, and the user threads should execute normally. Observe the address translations with the trace option `-d a`.

More generally, it is useful to encapsulate the allocation of physical pages to virtual pages in a special class `FrameProvider`. Note that it is then up to this class to zero-reset the provided pages, and no longer the job of the `AddrSpace` constructor.

Action I.5. *Create a class `FrameProvider` in the file `userprog/frameprovider.cc` that is based on tha class `Bitmap` for managing frames. It will make it possible to :*
- *retrieve a free frame initialised to 0 by the function bzero (`GetEmptyFrame\)` \ `item release a frame obtained via GetEmptyFrame` (\|`ReleaseFrame`) and*
- *ask how many frames are still available (`NumAvailFrame`).*

Note that the frame allocation policy is completely local to this class.

Action I.6. *Restructure the constructor method of `AddrSpace` to use these primitives, and run your program with various allocation strategies. For example, randomly allocate frames. Check carefully that all your user programs still work.*

Partie II. Executing several user programs at the same time

One user program uses only a part of the physical memory. Why not keep *several* user programs in memory at the same time then ?

Action II.1. *Define a system call `int ForkExec(char *s)` that takes an executable file name, creates a thread within the NachOS system and starts execution of the program driven by this thread, in parallel with the current program. Note that the current program and the started program can themselves contain threads (started by a function at MIPS level) ! Therefore, the following program should work :*

```
#include "syscall.h"

main()
{
  ForkExec("../test/userpages0");
  ForkExec("../test/userpages1");
}
```

avec pour `userpages0` et `userpages1` des programmes du genre

```
#include "syscall.h"
#define THIS "aaa"
#define THAT "bbb"

const int N = 10; // Choose it large enough!

void puts(char *s)
{
  char *p; for (p = s; *p != '\0'; p++) PutChar(*p);
}

void f(void *s)
{
  int i; for (i = 0; i < N; i++) puts((char *)s);
}

main()
{
  int i;
  UserThreadCreate(f, (void *) THIS);
  f((void*) THAT);
}
```

Action II.2. *efine your implementation so that the last process stops the machine via a call to method* `Halt()`*. Each process that ends and that is not the last should immediately release all the resources that it uses (its space structure, etc.).*

Action II.3. *Show that you can run a large number of processes (a dozen) with each one having a large number of threads.*

Partie III. Bonus : shell

Action III.1. *Implement a tiny shell, drawing inspiration from the program* `test/shell.c`*.*

Partie IV. Bonus+ : Dynamic Memory Allocation

The size of a process is static and calculated by the compiler with an arbotrary size for the stack. All memory pages are valid and it is possible to write anywhere in the process mamory (even in the code, try it !).

To implement a dynamic memory allocator, tehre is a need of an additional memory space betwen the code and the stack. However, it should be impossible to write in this zone without an explicit memory allocation. The memory pages sould exist but be invalid (even maye not allocated). In Linux/Unix, it is the `sbrk` system call that changes the limit between allocated and not allocated memory : `man sbrk`.

To make it simple, we will wotk at the page level. We want to define a system call `AllocEmptyPage` that will return a pointer to a new memory page. The system call `FreePage` will free the page.

Action IV.1. *Add in the* `AddrSpace` *class a private variable* `brk` *and a system call* `Sbrk(unsigned n)` *that allocates n new* `frame`*s* vides *for the process, validates the next n pages starting from* `brk` *assotiates them to these n frames respectively, increments* `brk` *($+n$) and return a pointer to the new memory zone.*
What should be done in the case of an error ?

Now we will implement the memory allocator you programmed during the semester.

The `mem_init` function becomes :

```
void *mem_init (size_t size) ;
```

The `mem` parameter has disappeared. The memory zone will be allocated directly in physical memory. Depending on *s*ize, a number of pages shoud be allocated with the *system call* `Sbrk`. `mem_init` will return the MIPS pointer to the firts byte of the first allocated page.

Action IV.2. *Implement* `mem_init` *in NachOS.*

The other two functions (`mem_alloc` and `mem_free`) keep their interface. The implementation of these functions would a priori not change.

Action IV.3. *Take you test programs for the memory allocator and verify that they have the same behavior as the one you tested in Linux.*