OPERATING SYSTEM
NachOS TD 3: Paging

Global instructions

The goal of this TD is to go further towards the realization in NachOS of a process model in the Unix way, each process possibly containing an arbitrary number of threads. More precisely, it involves setting up a simple paged memory management within the system.
You have to provide your source code files and a report of 3 to 5 pages according to the procedure described at the following address `http://adrien.krahenbuhl.fr/en/teaching`*. Respect the following rules:*

- ✓ *You do* **not** *have the obligation give answers in the question order.*
- ✓ *Some requirements are deliberately imprecise on several points.*
- ✓ *Some non-trivial implementation choices are therefore left to your discretion.*
- ✓ *Before beginning to code,* **read first of all the whole TD parts***: they contain both descriptive passages to explain some concepts of the lectures or about NachOS. You have also some* **actions** *indicating precisely how to design your implementation step-by-step.*
- ✓ *Warning: this TD requires the add of methods from your part. Think before coding, otherwise it will be harder!*
- ✓ *To be reversible, all your changes must be framed with:*
  ```
  #ifdef CHANGED
  ...
  #endif CHANGED
  ```
  By default, you already compile with `-DCHANGED`*.*
- ✓ *Changes not reported are strictly prohibited.*

1 Virtual Addressing by a Table of Pages

The entire MIPS machine works in virtual addressing, according to two mechanisms chosen by the user: the page table or the TLB (Translation Lookaside Buffer). We will be interested only in the mechanism of the page table. See how it is initialized in `userprog/addrspace.h` and `userprog/addrspace.cc`. A virtual address is composed by a page number and an page offset. For each virtual page number, the table associates a physical page number. The (physical) address of the page and the page offset determine the physically accessed address. The mechanism is implemented in the following function:

```
Translate(int intAddr, int* physAddr, int size, bool writing)}
```

in the `machine/translate.cc` file. All accesses to the memory in the interpreter are processed through this function. Look at how work `WriteMem` and `ReadMem`.

(1.1) Write a small test program `test/userpages0` that writes some characters to the screen. It will serve as a test program.

Now, you will load the program into memory by shifting everything from one page.

(1.2) Examine carefully the use of `executable->ReadAt` at the end of the `AddrSpace::AddrSpace` function of `userprog/addrspace.cc`. No need to go and see `ReadAt` code, it is simply the combination of an `lseek` and a `read`). Oddly (?), it is made to be write directly in physical memory MIPS.

How do we see this?

(1.3) Define a new local function:

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

which does the same as `ReadAt` (read `numBytes` bytes from the `position` position in the executable file) but by writing in the virtual address space defined in `pageTable` of size `numPages`. You can use a temporary buffer that you will fill with `ReadAt`, then that you will copy byte-by-byte in memory with `WriteMem` for example (even if it is not very efficient, do not try to optimize, it is very difficult). Think to temporarily change the page table in the machine in order that `WriteMem` uses the page table constructed in the `AddrSpace::AddrSpace` constructor, and restores it correctly (get inspired by `space->RestoreState`).

Use `ReadAtVirtual` in place of `ReadAt` where it is required, and check that the program execution is still working, especially by using `PutString`.

(1.4) Modify the creation of the page table in order that the virtual page `i` be a projection of the physical page `i+1`. Restart your program. Everything must work, and user threads must run normally. Observe address translations with the `-d a` trace option.

Overall, it is useful to encapsulate the allocation of physical pages in a special class `PageProvider`, global to NachOS. Since the physical pages will be reused, this class will have to reset the contents of the allocated pages.

(1.5) Create a `PageProvider` class in the `userprog/pageprovider.cc` file that relies on the `BitMap` class to manage physical pages. It allows:

- ✓ to retrieve the number of a free page and initialized to 0, thanks to the `memset` function (`GetEmptyPage` method)
- ✓ to free a page obtained by `GetEmptyPage` (`ReleasePage` method)
- ✓ to ask how much pages remain available (`NumAvailPage` method).

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

Why do I need only one object of this `PageProvider` class?

It can therefore be created at the same time as the machine in `Initialize`.

(1.6) Fix the `AddrSpace` constructor and destructor to use these primitives, and run your program with various allocation strategies. For example, just to test and "stress" the implementation, randomly allocate the pages.

What type of error can occur?

Think of treat this error (Put at least one `ASSERT`). If you have any problem, consider using the `-d a` option to observe the address translations.

Make sure that all your programs are still working.

2 Run Multiple Programs at the Same Time

Since for now only a part of the physical memory is used to project the virtual pages (`size` is not necessarily equal to `MemorySize`), why do not keep in memory several programs at the same time?

(2.1) Define a system call `int ForkExec(const char * s)` that takes an executable file name, creates an `AddrSpace` object from this executable file, and creates a kernel thread. This kernel thread runs the new process in parallel with the parent process (get inspired by the `StartProcess` function). Therefore, the next program must work:

```
#include "syscall.h"
```

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

You may need to increase `NumPhysPages`. Initially, put `while(1);` at the end of all your `main` functions to let sufficiently time at the different processes to make their display. You will solve the finishing problems later.

(2.2) Refine your implementation to automatically execute a `Halt()` when the last process stops. In particular, the system call `Exit()` mus not systematically calls `interrupt->Halt()`: if there are other processes, it is necessary to finish uniquely the current process to let run the others. Note that the threads are not yet processed here, they will be the object of the next action. Think however to immediately release all the resources that it uses (its space structure, ...).

(2.3) Now that the current program and the launched program may contain threads (launched by a function at the MIPS level). The below program should work:

```
#include "syscall.h"

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

with for userpages0 and userpages1 a programs of this type:

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

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

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

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

int main()
{
    ThreadCreate(f, THIS);
    f(THAT);
    ThreadExit();
}
```

To make it simpler at first, you can assume that MIPS programs do not mix threads and `Exit`: either they do not use threads and they finish with `Exit`, or they use threads and all threads call `ThreadExit` (including the main thread). In this last case, `Exit` is never called. The mixing of both will be treated in the bonus question.

(2.4) Show that you can launch a lot of processes (a dozen), each with a large number of threads (a dozen as well).

(2.5) (Bonus) Now mix threads and `Exit`. When a thread in a process calls `Exit`, all threads in this process must finish (and finish the process). You must keep somewhere the thread list of a process.

(2.6) (Bonus) If one of the threads brutally destroyed was performing a `PutString`, it still held the lock that you added to prevent the mix of `PutString` from different threads.

How to correct this?

(2.7) (Bonus) Show by monitoring the resource consumption by your Unix process that the MIPS/Nachos processes release their resources once completed. For this you can use `valgrind`:

```
valgrind --leak-check=full
```

which will indicate the unreleased areas.

It may signal a problem for the stack of the last kernel thread. This is normal. Why?

3 Bonus: a Shell

Implement a tiny shell using the `test/shell.c` program as an inspiration.