Syracuse University **CIS 657: Principles of Operating Systems** **Spring 2020**

PA-4: System Calls and Virtual Memory System

Total points: **100 pts** Due on **Wednesday, Apr 1 2020 at 11:59 PM**

Instructor: Endadul Hoque

Logistics

1. This assignment can be done either **individually or in groups (no more than two students in a group)**.
2. For this programming assignment (PA-4), you have to download an archive called `student-pa4.tar.gz` from Blackboard. Upon extraction, you will see a directory named `Nachos` under the directory of `student`, which contains several directories.
3. In this assignment, some tasks are depended on other tasks. For instance, for testing **Task-2** with several large user programs, you may have to complete **Task-3**.
4. Unless stated otherwise, you **must not** modify/edit/rename/move other existing files under the Nachos directory.
5. For all the following tasks, you have to use the Linux server (`lcs-vc-cis486.syr.edu`) dedicated for this course. You **must** know how to remotely login to that machine, how to use `terminal` and how to copy files back and forth between your computer and the Linux server.
6. Note that for solving each task you can refer some helpful links listed in Appendix A.

Obtain the source code for this PA

Obtain `student-pa4.tar.gz` from `blackboard`. Unpack the package to find the skeleton code required for this PA. The extracted directory is named `student/`.

```
$ tar xzf student-pa4.tar.gz
```

Submission Instructions

Attention: You have to submit your code and a report (named `report-pa4.pdf`) in **PDF** format (no other format is allowed). This file must be stored under `student/` directory like this `student/report-pa4.pdf`. What you should include in your report is listed in "**What to report?**" on Page 7.

Once you have the report and are ready to submit, you need to create your submission package as follows.

1. Go to the `Nachos/code/build.linux` directory and clean the directory. For example,

```
$ cd student/Nachos/code/build.linux
$ make clean
```

2. Change directory (`cd`) to your `student/` directory. Now `student/` should be your current working directory.
3. **Use your netid** to create a compressed archive (`<netid>-pa4.tar.gz`) of your solution directory. For instance, if your are working individually and your netid is `johndoe`, create a compressed archive as follows:

```
$ cd ..                       # moving to the parent directory
$ mv student/ johndoe-pa4/   #johndoe is a netid of the student
$ tar czf johndoe-pa4.tar.gz johndoe-pa4/
```

If you are working in a group and the netids are `johndoe` and `jtyler`, you need to follow the following format to submit your assignment.

```
$ cd ..                             # moving to the parent directory
$ mv student/ johndoe-jtyler-pa4/  #johndoe, jtyler are netids of students in a group
$ tar czf johndoe-jtyler-pa4.tar.gz johndoe-jtyler-pa4/
```

4. Use **Blackboard** to submit your compressed archive file (*e.g.*, `johndoe-pa4.tar.gz` or `johndoe-jtyler-pa4.tar.gz`).

Grading

We will use a semi-automated grading tool with pre-determined inputs to check your solution for correctness. ***Please make sure you follow the instructions provided throughout this handout; otherwise your assignment will not be graded at all by the auto grader.***

- You must not modify/edit/rename/move other existing files/directories in `student/` or in its sub-directories.
- You must follow the naming convention for each task as directed, while creating files, directories, archives, and so on.
- Your solutions must execute on the Linux machine (`lcs-vc-cis486.syr.edu`), otherwise your solutions will not be considered for grading.
- Your must follow the submission instructions, or it will not be graded properly by the auto grader.

Overview

In a real operating system, the kernel not only uses its procedures internally, but also allows user programs to access some of its routines via "system calls". In this assignment, you use Nachos to implement **system calls** and **virtual memory system** (for more detail about Nachos, you can refer Appendix A about Nachos Tutorial). Since Nachos executes MIPS instructions, these user programs must be compiled into MIPS format using the cross compiler (for more detail about cross compiler, you can refer Appendix A about cross compiler).

How to build Nachos?

```
$ cd Nachos/code/         # top directory for Nachos source tree
$ cd build.linux/

$ make clean
$ make depend
$ make
```

The user-space programs (also, called *test* programs) have been implemented in C using the system calls and are located under the `test/` directory. Several example programs are provided, including `halt`, `shell`, etc. The command shown below will load the user program `prog` to Nachos memory (physical frames) and run it on Nachos (assuming the current working directory is `Nachos/code/build.linux/`).

```
$ ./nachos -x ../test/prog
```

In this project, you will implement the virtual memory system and swapping space (using the basic file system of Nachos). You need to design your Virtual Memory System and get it working. Physical memory is used as a cache for virtual pages to provide the abstraction of an (almost) unlimited virtual memory size. During address translation, you may have to swap pages into physical memory from backing store (aka swap space) or swap pages out from physical memory to backing store, as necessary.

You do **NOT** need to consider/simulate **the TLB mechanism**. In other words, you will implement paging with swapping but without TLB.

You can find how **page fault exception** is thrown when performing translation inside the `translate` function (location: `machine/translate.cc`). On a page fault, the kernel must decide which page to fetch in as well as replace if necessary.

Important: If not stated otherwise, all the paths to the file locations in this handout are relative to `Nachos/code/` directory.

Task 1 (30 pts)

Implement system calls and exception handling for user programs: `syscall.h` defines the system call prototypes of Nachos (location: `userprog/syscall.h`). These are kernel functions that user programs can invoke. You need to implement the system calls (*to be precise*, the **handlers**): `Halt` (already done), `Fork`, `Exit`, `Exec`, `Read`, and `Write`.

How to implement system call handlers?: Technically you will not directly implement a function named `Read()`, `Fork()`, and so on. In fact, these are system call APIs that test-programs can invoke. You have to implement the underlying system call handlers for `Read()`, `Write()`, and so on.

For this task, `exception.cc` must be modified. This file (location: `userprog/exception.cc`) implements the **handlers** for system calls and other user-level exceptions. The `Halt()` system call is already implemented. You can find another good example from system call `Add()`.

These files may be a good starting point to understand the execution flow of system calls: `userprog/syscall.h`, `userprog/exception.cc`, `test/start.S` (assembly routine for system calls under `test/` directory), and `threads/scheduler.cc`.

1. **Implement Read and Write for Console**: You need to implement `Read` and `Write` system calls, which will be used by test programs (*e.g.*, `test/Read.c` and `test/Write.c`) to read from and write to console (*i.e.*, `stdin` and `stdout`). You can refer to `#define CONSOLEINPUT 0` and `#define CONSOLEOUTPUT 1` in `userprog/syscall.h`.

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

     (a) Reads `size` bytes from the open file into `buffer`,

     (b) Check if `id` is `CONSOLEINPUT` (OpenFileId id 0 is “**stdin**”)

(c) Perform console I/O

(d) Returns the number of bytes actually read, which does not always have to be equal to the number of bytes requested.

- `int Write(char *buffer, int size, OpenFileId id)`:

  (a) Writes `size` bytes from `buffer` to the open file,

  (b) Check if `id` is `CONSOLEOUTPUT` (OpenFileId id 1 is "**stdout**")

  (c) Perform console I/O,

  (d) Returns the number of bytes successfully written.

Hint: You may leverage the existing functionalities in `class SynchConsoleInput` or `class SynchConsoleOutput` (see `userprog/synchconsole.[h|cc]`). See `class Machine` (location: `machine/machine.[h|cc]`) to directly read from or write to memory location.

2. **Implement Fork system call**:

- `ThreadId Fork()`:
  `Fork()` creates a child process (*i.e.*, a Nachos Thread) and returns `ThreadId` (see: `userprog/syscall.h`) of the new thread, which is a unique identifier assigned to the new thread (similar to UNIX `pid`).

  Hint: You need to complete the *copy constructor* of the address space (see `AddrSpace::AddrSpace(const AddrSpace& copiedItem)`) at `userprog/addrspace.cc`. You need to put the new Nachos thread into `readylist` which is inside `Scheduler`. You need to know how to save and restore the thread's state.

  The child thread (aka the child process) must continue its execution from the instruction immediately following the call to `Fork()` procedure, just the `fork()` syscall of UNIX.

  You need to refer to the comments in `exception.cc` for hints on how to return a value from system call.

3. **Implement Exec system call**:

- `SpaceId Exec(char* exec_name)`:
  `Exec()` takes one string argument that is the path name to a user program. When a user program calls `Exec()`, Nachos kernel creates a new thread for the executable file name given as the argument to `Exec()`. Nachos creates the address space, initializes states and registers, and jumps to user level, as is done in `RunUserProg()` (location: `threads/main.cc`). `Exec()` also needs to return a `SpaceId` (see: `userprog/syscall.h`) of the new thread.

  Example: `SpaceId id = Exec("../test/prog")`.
  Hint: You need to copy the filename argument from user memory to kernel memory safely. You need to refer to the comments in `exception.cc` for hints on how to return a value from system call.

4. **Implement Exit system call**

- `void Exit(int status)`:
  `Exit()` can be implemented with a simple call to `Thread::Finish()`. But you have to be careful. All resources including `AddrSpace` must be freed. The kernel handles the `Exit()` system call by

destroying the process (*i.e.*, Nachos Thread) data structures and thread(s), reclaiming any memory assigned to the process. There is no return value.

Attention: You need to protect the Nachos kernel from user program errors. User programs can do nothing to crash the operating system except `Halt` system call.

Task 2 (20 pts)

Implement multiprogramming: the original Nachos code is restricted to running one user program at a time. You need to come up with a way for allocating physical memory frames, so multiple programs can be loaded into the machine's memory, and provide a way for copying data from/to the user's virtual address space. **This task is highly related to Task 3.**

1. when the `-x` flag is used, the `main` function of Nachos calls the `RunUserProg()` function (location: `threads/main.cc`) with the string that follows `-x` as parameter. Remember that every user program should have its own thread structure. Before running any user program, there is only one thread, aka the **main** thread, for Nachos itself.

   Example: `./nachos -x ../test/prog1 -x ../test/prog2 -x ../test/prog3`
   What this means is that when you type this command, Nachos will load multiple user programs into its main memory and start executing them as threads with round-robin scheduling.

2. Add `-Q` option to dynamically set different **time slice** quantum for round-robin scheduling.

   Example: `./nachos -x ../test/prog1 -Q 100`

Attention: Do not use linear (contiguous) memory allocation for multi-programming

Task 3 (50 pts)

Implement Memory Manager for Virtual Memory: In simple words, you have to implement *paging with swapping but without TLB* for virtualizing (limited) physical memory. For your swap space, you can reserve/allocate a dedicated space (aka backing store) on the disk using the Nachos basic file system. You can name it `SwapSpace` by using `FILESYS_STUB` (location: `filesys/filesys.h`) in Nachos. Your implementation will let Nachos run the program with part of the address space loaded into memory, and load the other part(s) when they are accessed.

1. **Modify/Edit Functions related to Address Space (`AddrSpace`)**
   - You have to edit the appropriate function(s) while allocating the number of pages
     Hint: See `Load()` inside `userprog/addrspace.cc`
   - You have to edit the appropriate function(s) while creating a replica of a process (say, using `Fork()`)
     Hint: See `AddrSpace(const AddrSpace& copiedItem)` inside `useprog/addrspace.cc`
   - You have to edit the appropriate function(s) while terminating a process
     Hint: See `~AddrSpace()` inside `useprog/addrspace.cc`

2. **Implement backing store for demand paging**
   Implement functionalities to move a page from the backing store (swapping space: `SwapSpace`, the file) to memory and from memory to the backing store:
   (a) Using Nachos file system, you need to create a file (say, `SwapSpace`) to implement backing store
   (b) You may need a data structure to keep track of virtual pages in the backing store.
3. **Implement Page fault handler** Each process (*i.e.*, Nachos Thread) has a page table, where each PTE contains the virtual page number, the physical page frame number, other control bits (valid, dirty, referenced, and so on). (See `machine/translate.h`).
   (a) Find a free physical page frame (you need to maintain a Frame Table). If necessary, run **page replacement algorithm**. You need to implement **LRU** page replacement algorithm.
   (b) Find the desired virtual page on the backing store and swap the page into the free page frame.

   Hint: Related functions are `Load()` and `ExceptionHandler(ExceptionType which)`. See `useprog/addrspace.cc` and `useprog/exception.cc`.

4. **Required Print Format for Outputs**
   For the following outputs,
   `[pid]` is the id of the process (Nachos Thread); the operation is performed behalf of this `pid`.
   `[virtualPage]` is the involved virtual page number (*i.e.*, the page index into the process virtual address space).
   `[physicalPage]` is the involved physical page frame (*i.e.*, the page index into the physical memory of the Nachos virtual machine).
   - Whenever a page is loaded into physical memory, print
     `L [pid: %d]: [virtualPage: %d] → [physicalPage: %d]`
   - Whenever a page is evicted from physical memory and written back to the swap area, print
     `S [pid: %d]: [physicalPage: %d] → [virtualPage: %d]`

   Example:
```
...
S[pid: 1]: [physicalPage: 14] → [virtualPage: 112]
L[pid: 1]: [virtualPage: 46] → [physicalPage: 14]
S[pid: 2]: [physicalPage: 16] → [virtualPage: 113]
L[pid: 2]: [virtualPage: 101] → [physicalPage: 16]
S[pid: 3]: [physicalPage: 30] → [virtualPage: 34]
L[pid: 3]: [virtualPage: 156] → [physicalPage: 30]
...
```

   Hint: You may need to refer `stats.h` and `stats.cc` (location: `machine/stats.h` and `machine/stats.cc`) to calculate the number of page faults and print the number of page fault out.

Testing

Some test programs are provided in `test/` directory, including **Read**, **Write**, **Exit**, **Exec**, **Fork**, **matmul**. You can use them to verify that your system call implementations and multiprogramming are working properly with your Virtual Memory System.

If your program runs correctly, the output will include "`PASS`".

You can test each task as follows:

1. **For Task 1, run a single test program**
   - You need to test **Read**, **Write** and **Exit** programs.
   - You need to test **Exec** and **Exit** programs.
   - You need to test **Fork** program.
2. **For Task 2, stress things more and run multiple programs**
   - Run at least 4 (given) test programs located in `test/` directory. For example, you can run **Read**, **Write** and **matmul** under the `test/` directory.
3. **For Task 3, run user programs that require larger memory**
   - You need to run three **matmul** programs under `test/` directory.

Debugging

There are three ways to help you debug your code (for more detail about debugging in Nachos, you can refer Appendix A about debugging in Nachos):

1. You can insert some Standard Output Stream (`std::cout`) to the code.
2. You can use the `gdb` debugger or another debugger of your choice.
3. You can insert calls to the `DEBUG` function that Nachos provides. (Details about `DEBUG` can be found at the top of `threads/main.cc` and `lib/debug.h`)

What to Report?

You have to write a report named `report-pa4.pdf` in PDF format and store it under `student/report-pa4.pdf`.

You can use any typesetting programs/applications to write your report. But you **must** convert it to **PDF format** and submit the PDF file. **No other file format is allowed**.

You need to include the following information in your report:

- Mention your name(s), SU netid(s)
- List of your files with directory name that you modified or created for this PA
- Design of your solution for each task (briefly explain in your own words)
- Include parts of your implementations (*i.e.*, Code snippet with good comments)
  - Do not include Nachos original code
  - We want to see your own code/modification

    - Should be in (preformatted/code-block/regular) text, but no image/photo of code
- Report on Testing results
    - The outcome of your implementation for each test case/program mentioned earlier
    - you must test multiple test programs
    - Include results of your virtual memory management system per process (Nachos Thread) or entire system
- Report on any error or incompleteness in your submission that the TA should be aware of before grading your submission.

Appendix A: Some Useful Links

- **Nachos Tutorial**:
  `https://www.ida.liu.se/~TDDI04/material/begguide/`
  `https://www.student.cs.uwaterloo.ca/~cs350/common/NachosTutorialF06.pdf`
- **Cross Compiling User Applications**:
  `http://cseweb.ucsd.edu/classes/fa19/cse120-a/projects/crosscompile.html`
  `http://www.cas.mcmaster.ca/~rzheng/course/Nachos_Tutorial/nachossu15.html`
- **Debugging in Nachos**:
  `https://users.cs.duke.edu/~chase/nachos-guide/guide/nachos.htm#_Toc535602511`