CMSC 421 – Operating Systems
Krishna Sivalingam and Dennis Frey
Spring 2006
Project 2: Memory Management System
Due Date: **April 25, 9PM**; On-Line Submission via *submit* command
NO LATE SUBMISSIONS
This file is dated: March 30, 2006; and replaces all earlier versions.

1 Project description

The objective of this project is to implement a memory management system that handles paging and virtual memory. The project will be implemented in C/C++ that will run in the GL Linux environment using the /usr/local/bin/g++ or /usr/local/bin/gcc compilers with the -Wall and -ansi flags set.

Input values: Main memory size (M in KB), virtual memory size (V in KB), a set of executable files, and page size (P in bytes). For example, $M = 8$, $V = 16$ and $P = 512$ indicates an 8 KB main memory system with 16 KB virtual memory and page size of 512 bytes. Thus, there are 16 pages in main pages and 32 pages in virtual memory.

Input Executable data files: There will be a set of input executable files, the contents of which are:

- ▷ Size (in KB) of executable. This is not the real value of the executable size. It is only an assigned value, specified by the user. It has no relation to the list of commands in the executable. This size will be used for purposes of memory management. For example, if the size of the executable is 8 KB and page size is 512 bytes, then the corresponding process will require 16 logical pages of address space. See Section 2 for executable file examples.
- ▷ A set of commands in the format:
  - ⋆ add x, y, z

    Semantics: Add the contents (unsigned 8-bit integer values) of memory address x and memory address y and store it in memory address z; Add overflow need not be considered. Note that x, y and z are logical memory addresses.

    Output: Echo the command to the screen and the result of the addition, as in:

    Command: add x, y, z; Result: Value in addr x = 100, addr y = 50, addr z = 150
  - ⋆ sub x, y, z

    Semantics: Subtract the contents (unsigned 8-bit integer values) of memory address y from that of memory address x and store it in memory address z; Assume that contents of x is greater than or equal to the contents of y. Note that x, y and z are logical memory addresses.

    Output: Echo the command to the screen and the result of the addition, as in:

    Command: sub x, y, z; Result: Value in addr x = 100, addr y = 66, addr z = 34

- ⋆ print x
  Print to screen value of contents in memory address x, treating the value as an unsigned integer.
  Output:

  Command: print x; Result: Value in addr x = 45

- ⋆ load a, y
  Semantics: $y \leftarrow$ a
  Output:
  Command: load 10, y; Result: Value of 10 is now stored in addr y

Important: For all above commands, if an invalid logical memory address is specified, the program should output an error message. For example, if the executable size for process with *pid* of 2 (pid is explained later) is 4 KB (4096 bytes) and the user specifies an address of 12234 in the command, an output of the form below should be printed to the screen:

```
Invalid Memory Address 12234 specified for process id 2
```

The program interpreter should then stop running this executable and await the next user command.

System commands: On startup, the system initializes memory and waits for commands from user. That is, it is an interactive program that executes the user's commands, which will be from the following list:

▷ `load <filename1> <filename2> .. <filenameN>`

Attempt to load the specified executables into memory, IN ORDER. If there is room in main memory, a process is placed there; else, it is loaded in virtual memory if adequate space is available. Print an error message and stop loading when memory and virtual memory are full. This command DOES NOT RUN THE SPECIFIED EXECUTABLES.

Assign each executable a process id (*pid*) from a global *pid* counter in your program that starts at 1.

Output: For each file specified, output the assigned *pid* as in the sample below:

```
filename1 is loaded in main memory and is assigned process id 1
filename2 is loaded in virtual memory and is assigned process id 2
filename3 could not be loaded - file does not exist
filename4 could not be loaded - memory is full
```

▷ `run <pid>`

Execute the sequence of instructions in the program identified by its *pid*; If a process is in virtual memory, bring the process into physical memory and execute it. This may result in some other process being swapped out.

For each statement, print the corresponding output statements to screen, as explained elsewhere in this document.

If an invalid *pid* is specified, output an error message and continue to accept the next command.

▷ `kill <pid>`

Kill the program identified by process id; remove all memory allocated to the process. Print a corresponding confirmation message to the screen.

If an invalid *pid* is specified, output an error message and continue to accept the next command.

▷ `listpr`

Print to screen the identifier values of all processes in main memory and in virtual memory: print the *pid*s of processes in main memory first (sorted by *pid*), followed by *pid*s of processes in virtual memory (sorted by *pid*).

▷ `pte <pid> <file>`

Print the page table entry information of a process to the specified output file. This will contain the logical page number and the physical page number for each logical page.

If an invalid *pid* is specified, output an error message and continue too the next command.

If the file already exists, append the output to the file. Include the date/time at the start of output data each time this command is invoked.

▷ `pteall  <file>`

Print all the page table entries to the specified output file in ascending *pid* order (starting at *pid* 1).

If the file already exists, append the output to the file. Include the date/time at the start of output data each time this command is invoked.

▷ `swapout <pid>`

Swap out the process specified by *pid* into virtual memory. No other action needed. Print to screen corresponding messages.

If an invalid *pid* is specified, output an error message and continue to accept the next command.

▷ `swapin <pid>`

Swap in the process specified by *pid* into main memory. Note that other process(es) may have to be swapped out to make room. Print corresponding messages to screen. When a process is swapped in (or out), ALL the pages of that process will be swapped out (or in) at the same time.

If there is not enough capacity in main memory, the page / process replacement algorithm should swap out the last run process(es) in main memory.

If an invalid *pid* is specified, output an error message and continue to accept the next command.

▷ `print <memloc> <length>`

Print the values stored in the *physical memory* locations from *memloc* through *memloc* + *length - 1*. For example, *print 1001 3* will result in an output of:

```
Value of 1001: 23
Value of 1002: 45
Value of 1003: 113
```

▷ `exit`

Exit the system and clean up all allocated memory.

Assume that there is no I/O in any program and that all the processes in memory are in the ready queue.

Process identifier: Every process has a unique 32-bit process identifier (*pid*), that is assigned by YOUR program when the program is first loaded into memory. The first program loaded will have a *pid* of 1, and subsequent processes will be assigned values of 2, 3, etc. Assume that the process identifier values will not wrap-around and hence *pid* reuse is not necessary.

2 Sample Session

Assume that you have created the necessary files and the corresponding executables in your PROJECT2 directory.

Assume that $M = 32; V = 32; P = 512$. Also, there are four executable files as described below. Note that the first

▷ **file1** has size 4 KB and following contents:

```
4
load 11, 1001
load 21, 2001
add 1001, 2001, 3001
print 3000
print 3001
```

▷ **file2** has size 8 KB and following contents:

```
8
load 12, 1002
load 22, 2002
sub 2002, 1002, 3002
print 3002
```

▷ **file3** has size 8 KB and following contents:

```
8
load 13, 1003
load 33, 3003
add 1003, 3003, 3005
print 3005
print 10000
```

- **file4** has size 16 KB and following contents:

```
16
load 14, 1004
load 24, 2004
sub 2004, 1004, 3004
print 3004
```

The sample session is as follows:

```
% cd PROJECT2

% make proj2

% ./proj2 -M 32 -V 32 -P 512
<Command Please> load file1 file2 file3
<Command Please> run 1
<Command Please> pteall outfile1
<Command Please> swapout 2
<Command Please> pte 2 outfile1
<Command Please> load file4
<Command Please> swapin 2
<Command Please> listpr
<Command Please> pteall outfile1
<Command Please> run 2
<Command Please> kill 1
<Command Please> pteall outfile1
<Command Please> run 2
<Command Please> run 3
<Command Please> run 4
<Command Please> run 1
<Command Please> exit
```

3 What to Submit

Log into one of the campus GL machines. Change to your project directory. Make sure that ONLY files related to this project are in this directory - we do not want any temporary files, music files, etc.

Submit your project using the *submit* command. For help with submitting, click on the *Miscellaneous* link on the course homepage.

```
submit cs421 proj2 <list of files>
```

You can test the program execution after submission using *submitmake* and *submitrun* routines that are provided in Mr. Frey's public directory (`/afs/umbc.edu/users/d/e/dennis/pub/CMSC421`).

Submit the following files:

- Source Files

- ▷ Makefile
  Typing command 'make' at the Linux command prompt **MUST** generate the required executable. The executable **MUST** be named **proj2** as in the samples above. Make must use either the /usr/local/bin/g++ or the /usr/local/bin/gcc compiler as well as the -Wall and -ansi flags.

- ▷ A Script file obtained by running UNIX command *script* which will record the way you have finally tested your program. The script file will show the execution of the system demonstrating at least the sample session above. You can include additional runs of the system.

- ▷ A README file for the TA. The README should document known error cases and weaknesses with the program. You should also document if any code used in your submission has been obtained/modified from any other source, including those found on the web. If you helped any other 421 student and/or took help from/discussed with any other student, please describe it here.

- ▷ A COMMENTS file which describes your experience with the project, suggestions for change, and anything else you may wish to say regarding this project. This is your opportunity for feedback, and will be very helpful.

4 Help

1. WARNING ABOUT ACADEMIC DISHONESTY: Do not share or discuss your work with anyone else. The work YOU submit SHOULD be the result of YOUR efforts. The academic conduct code violation policy and penalties, as discussed in the class website, will be applied.

2. Ask questions EARLY. Do not wait until the week before. This project is quite time-consuming.

3. Implement the solutions, step by step. Trying to write the entire program in one shot, and compiling the program will lead to frustration, more than anything else. For example, you can start with interpretation of the commands in a file, then implement paging support, followed by virtual memory, etc.

5 Grading

- ▷ Basic Executable Command Interpretation Module: 15 points

- ▷ Memory System with Paging: 50 points

- ▷ Virtual Memory: 35 points

No README/COMMENTS: -5 points; No Script File: -10 points; Incomplete Compilation: -10 points