CS350 **Operating Systems** **Spring 2006**

Assignment Two

Due: July 5, 2006 (noon)
Returned: July 18, 2006
Appeal deadline: July 25, 2006

This assignment requires you to add support for virtual memory management to the NachOS operating system.

1 Design Requirements

The specific requirements for this assignment are as follows:

1. Implement support for the hardware TLB. To enable the TLB in the simulated hardware, you must edit the NachOS `Makefile` and then rebuild `nachos` from scratch. (In the `Makefile` add `-DUSE_TLB` to the `DEFINES` line.) After you have enabled the TLB, the hardware will look *only* in the TLB to find address translations. The hardware will no longer look up address translations in your page tables (but the OS still needs to maintain and will use them).

   The TLB itself is an array of `TLBSize` page table entries. Your OS can manipulate (read and modify) these entries through the `tlb` member of the `Machine` class, e.g.,

   ```
   kernel->machine->tlb[i]
   ```

   refers to the `i`th entry in the TLB.

   It is important to distinguish a TLB fault from a page fault and to understand the differences between the two. When a PageFaultException occurs in NachOS, it indicates that the TLB does not contain a valid translation for the virtual page being accessed. Although this is called a PageFaultException it is actually indicating a TLB fault. When a TLB fault occurs the ExceptionHandler is called with the exception type set to PageFaultException. The kernel needs to load a valid translation into the TLB (which may or may not require loading a page frame) and to re-execute the instruction that generated the TLB fault.

   When a TLB fault occurs and the page that needs to be accessed does not reside in memory, it is called a page fault. Note that a TLB fault can occur even when the page being referenced is already in memory.

   The TLB is used for all address translations, so any entry in the TLB that is marked as "valid" (i.e., the valid bit is set) will be used for translations. Therefore, your kernel must ensure that the correct page is in memory and that the running application should be permitted to access that page (i.e., it's part of the applications address space) if the TLB entry is valid.

   You are to implement a simple FIFO algorithm for TLB replacement.

2. Provide support for *demand paging/loading*. In particular, rather than loading all pages of the program into memory initially, load them only as required. If a page is never referenced, then it should never be loaded. This means that initially no pages of a process should be loaded into memory. The program actually begins execution without any pages in memory and pages each page in (including the first page referenced) as the result of handling page fault exceptions.

   Implementation hint: start simply by running only one small program that you know will fit completely in memory, since this part of the assignment does not require one to remove pages from memory. Once you are convinced this works move on to the next part of the assignment.

3. Provide support for larger virtual address spaces (and more concurrent processes) by adding the ability to choose a victim page to be replaced using the FIFO algorithm and replacing that page. Use a global page replacement scheme.

   If a page has been modified, before replacing it, you will need to save that page (page it out) to a NachOS simulated disk that is used as the the swap device. You can reference this in the kernel through `kernel->swapDisk`. So, the level of multiprogramming of your OS will be limited by the size of the disk, not by the size of RAM. Your kernel is responsible for managing the space on the swap device.

   Be sure to only write a page to the swap device if it has been modified; that is, load a page repeatedly from the executable file until the page content changes. You may assume that the executable file does not change while the program is running.

   Implementation hint: you might wish to start by first ensuring that you can run a program with a large memory footprint that does not modify any pages. Once you are convinced that your code works for this case, add the ability to page out to, and back in from, the swap device.

4. Implement the Enhanced Second-Chance (Clock) Algorithm as described in the course notes and the text book. Note that the text book has more details about the algorithm than the course notes.

   For all replacement algorithms in this assignment use a global page replacement scheme.

   By default NachOS should use your new algorithm. By using a -F option on the command line, your version of NachOS should use the FIFO page replacement algorithm.

   Implementation hint: it is probably a good idea to implement, test, and debug your system using FIFO first before implementing the additional page replacement algorithm.

5. Keep and print statistics for the number of TLB faults, pages faulted in, total pages replaced, dirty pages replaced (i.e., written to the swap device) and clean pages replaced (i.e., not written to the swap device). (Yes, clean pages replaced plus dirty pages replaced should equal the total pages replaced.) Note that although NachOS already tracks and prints some statistics, this is done by the hardware (in `code/machine/stats.h` and `code/machine/stats.cc`). You may NOT modify these files (or any files in `code/machine` because this would be equivalent to trying to modify the hardware. Instead add code elsewhere in your kernel to track these statistics.

   In the NachOS code provided to you, the statistics tracked by the hardware are printed out when the simulated machine halts. They can also be printed after "ctrl-z" has been used to suspend machine execution (see the `ctrlZhandler` function in code/threads/main.cc). Modify NachOS so that your kernel-tracked statistics will also be printed under these same circumstances.

   You should also reset your kernel-tracked statistics in the `ctrlZhandler` when the other hardware tracked statistics are reset.

```
case 'r': kernel->stats->Reset();
          break;
```

6. Provide support for automatic sharing of read-only code pages between address spaces. That is, if the same program is loaded into more than one process' address space, any pages that hold program code should not appear more than once in physical memory. To simplify the problem, assume that programs with the same name will use the same executable, and that the executable will not change.

7. In addition to the set of test programs you'll write to demonstrate that your OS works, write test programs (as outlined below) that initialize every element of a two-dimensional **square** matrix of characters. This must be done by touching (by writing the value 'a') once, and only once, to each element of the matrix. The size of the matrix must be as close as possible to, but no less than, N pages, where N is the total number of frames of memory in the workstation. Your programs should only initialize the matrix and therefore shouldn't be generating extra page faults by touching or using unnecessary code or data. Write five programs that differ in the order in which they initialize the elements of the matrix (some programs may use the same order if necessary). The five required programs are as follows:

(a) A program that generates as *few* page faults as possible when run with your FIFO page replacement algorithm.

(b) A program that generates as *many* page faults as possible when run with your FIFO page replacement algorithm.

(c) A program that generates as *few* page faults as possible when run with your additional page replacement algorithm.

(d) A program that generates as *many* page faults as possible when run with your additional page replacement algorithm.

(e) A program that generates as *many* TLB faults as possible when run with your FIFO page replacement algorithm.

Hand in a description of these test programs and their expected behaviour along with paging-related statistics (as described in point 5 above) that are output as a result of running these programs.