CS350 **Operating Systems** **Fall 2006**

Assignment 2: Virtual Memory & Paging

Due date: 12:00 (noon), Monday, November 6, 2006

In this assignment, you will add support for virtual memory and paging to Nachos. For all parts of the assignment, you are not allowed to make any changes to the files in the `machine/` directory. Changing those files would correspond to changing the hardware in the real world – something that is rarely possible. All changes you make to the system must be in the kernel part of Nachos (i.e., in `threads/`, `userprog/`, etc.).

For all parts of the assignment, your implementation must be stable and robust. Repeatedly calling `Exec`, for instance, may produce error codes after a certain point, but should not crash your kernel. Likewise, calling `Free` with a pointer that was not obtained from `Alloc` should not crash the kernel.

You are expected to implement test cases for all parts of the assignment (with the exception of question 7), through either user space programs or output produced by your kernel, and to document the testing you have done.

Before you start working on this assignment, change the value of `TLBSize` in `machine.h` to 8. The original value 4 is too small to obtain any sensible results.

1. Implement support for hardware TLB. The stock version of Nachos uses the page tables of the individual processes directly to translate virtual addresses to physical addresses. In order to enable the TLB, you will have to change the Nachos `Makefile` (add `-DUSE_TLB` to the `DEFINES` line in `build.solaris/Makefile`) rebuild the whole package (`make clean ; make`). After you have enabled the TLB, hardware will only consult the TLB for address translations; it will no longer look up address translations in the processes' page tables. If you run Nachos with a test program now, it will crash. You will need to change certain parts of Nachos so that it does not crash any more.

   The TLB in Nachos is a small array of `TranslationEntry` instances (copies of page table entries) that can be accessed through `kernel->machine->tlb`. The number of elements in the array is `TLBSize` (defined in `machine.h`).

   When the hardware tries to translate a virtual address into a physical address and it does not find an appropriate entry in the TLB, it will raise a `PageFaultException`. This exception will have to be handled by your kernel (add a handler to the `ExceptionHandler` function). It is important to understand that the exception, even though it is of type `PageFaultException`, is not necessarily a *page fault*, but only a *TLB miss*. A TLB miss occurs when the MMU cannot find an appropriate entry for the given virtual address in the TLB. It might very well be that the respective page currently is in main memory; the hardware just does not know this, because all it sees is the TLB.

   Your kernel is responsible for placing page table entries in the TLB so that the hardware can use them for address translation. It is also responsible for keeping the information in the page tables consistent with the data in the TLB (short periods of inconsistency are acceptable and in fact inevitable).

   Since the TLB is a cache for an address space's page table, you will need to define a replacement policy for the TLB. Implement two policies, FIFO and CLOCK. The default policy is CLOCK. When Nachos is started with command-line parameter `-tlb_fifo`, it should use your FIFO implementation instead.

2. Implement support for *on-demand loading* of pages. When a program is started, instead of loading all its pages into physical memory, they should be copied into the swap file (`kernel->swapDisk`). A page should be loaded from the swap file into physical memory only when it is accessed by the program. Thus, parts of the program that are never touched should never be loaded into main memory.

   In order to realize this behaviour, you will need to change the code in `AddrSpace::AddrSpace`. You will also need to be aware of the different data sections in a Nachos executable file (code, read-only data, initialized data, uninitialized data, stack).

Extend your handler for `PageFaultExceptions`. Whenever your handler notices that the exception is indeed a page fault (as opposed to a mere TLB miss) and the page referred to by the given virtual address is currently not in primary memory, your kernel will need to load the page from the swap file into primary memory. In order for this to work, your implementation needs to maintain a free-frame map and also a map that can be used to determine the sector (i.e., file position) of each page in the swap file.

For this question, you may think of the swap file as the memory and treat the frames in physical memory as a big cache for the swap file. In particular, every page in the system will have a copy in the swap file. When a page is loaded into main memory, a copy of it will still be in the swap file, so there will be two versions of the page. The maximum possible number of pages in the system is given by `NumSectors` (defined in `disk.h`). You will have to change this in question 5.

Note: Since you have not implemented a replacement policy for pages at this point, make sure the programs you use for testing this feature are small enough to fit entirely into main memory.

3. Implement a global FIFO page replacement policy. Whenever a page has to be loaded into primary memory, and there is no free frame available, your implementation will identify the page that has been in main memory for the longest time and evict it from RAM.

   If a dirty page is chosen for eviction, it will need to be written to the swap file (`kernel->swapDisk`) first before it can be replaced by the new page. Make sure that your kernel only writes dirty pages to the swap file. If a page has not been modified after it was loaded into memory, it must not be written to the swap file, since the swap file already contains the most up-to-date version of the page.

4. As a second page replacement policy (global as well), implement the Enhanced Second Chance algorithm (CLOCK + cleanliness). A descriptions of this policy can be found in the textbook and the course notes.

   By default, Nachos should use your new replacement policy. When the parameter `-paging_fifo` is passed on the command line, your kernel should use the original FIFO replacement policy instead of your new policy.

5. Refine your paging mechanism and implement support for uninitialized pages. Implementing these changes will require some time. It is therefore recommended that you only start with this part of the assignment after everything else is working.

   In a first step, allow a page to only have a single copy in the system. That is, when a page is loaded into main memory, its copy in the swap file may be replaced by some other page. Note that from the page replacement algorithm's point of view this makes the page dirty, even if it has not been changed, since there is no copy of the page in the swap file. Therefore, it should be avoided whenever possible:

   - If the total number of pages in the system is smaller than `NumPhysPages`, all pages should be kept in primary memory and not in the swap file at all. That is, a page can only be evicted from primary memory if all frames are in fact occupied.
   - If the total number of pages in the system is larger than `NumPhysPages`, but smaller than `NumSectors`, a page that is loaded into primary memory should keep its copy in the swap file so that, unless it is actually dirty, it can be evicted from main memory without being written back to disk first.
   - If the total number of pages in the system is larger than `NumSectors`, a page that is currently in memory can lose its copy in the swap file to make room for more pages. Conceptually, this will make the page dirty.

   After you have implemented this change to your kernel, the total number of pages in the system is no longer limited by `NumSectors`, but by `NumSectors + NumPhysPages`.

   In a second step, implement support for uninitialized pages. An uninitialized page is a page that has never been accessed and whose contents are undefined. Examples of uninitialized pages are pages from a program's uninitialized data segment and unused stack pages. An uninitialized page should not consume any space in primary memory or in the swap file, except for some meta-information in

the kernel that tells you which pages are initialized and which are not. When an uninitialized page is accessed for the first time, your kernel has to initialize it by setting its contents to zero (like in the original implementation of `AddrSpace::AddrSpace`) and loading it into primary memory. If accessing an uninitialized page leads to a total number of initialized pages in the system that exceeds `NumSectors` + `NumPhysPages`, your kernel should terminate the process that is trying to access that page.

After you implement this change, your kernel will be able to run a program even if it has more pages than `NumSectors` + `NumPhysPages`. Your kernel will only start terminating processes if the number of initialized pages in the system exceeds `NumSectors` + `NumPhysPages`.

Note: Even if you do not implement this part, you can still receive full marks for questions 6 and 7. If you don't any see light at the end of the tunnel, just skip the question and finish the remaining ones.

6. Record statistics about virtual memory usage. Keep counters for the number of TLB misses, the number of page faults, the number of pages evicted, and the number of dirty pages evicted from main memory.

   The Nachos hardware itself already keeps track of some statistics. You may not use or modify these data. Instead, maintain your own counters. Statistics should be printed to `stdout` (or `cout`) when the machine halts and when Ctrl-Z is pressed (handled in the function `ctrlZhandler` in `main.cc`). All counters should be reset when Nachos' built-in counters are reset.

7. Run the three test programs `bubblesort.c`, `heapsort.c`, and `mergesort.c` (available from the assignment Web page) with your implementation of the FIFO page replacement policy and with your implementation of Enhanced Second Chance. Change the value of `ARRAY_SIZE` and observe how the number of page faults changes for the different sorting algorithms. Report all numbers (TLB misses, page faults, pages evicted, dirty pages evicted) recorded by your kernel for different values of `ARRAY_SIZE`. Explain the different behaviour of the three sorting algorithms. Which algorithm is the best when sorting very large arrays?

   This question does not count towards the page limit of your design document. Answer the question on a separate page, attached to your design and testing documents.

8. **Bonus Question** – optional

   Add two new system calls `Alloc` and `Free` to `syscall.h` and `start.S`. Implement a kernel handler for the new system calls. `Alloc` takes a single parameter n, the number of bytes to allocate. Your system call handler for `Alloc` will extend the virtual address space of the calling process by a number of pages ($\lceil$ n / `PageSize` $\rceil$) in order to create some room in the process' address space and return a virtual address to the calling thread that represents the start address of the freshly allocated memory region. `Free` takes a single parameter p, a pointer into the process' virtual address space that represents the start of a memory region previously allocated via `Alloc` and deallocates the memory region. A page created by calling `Alloc` is an uninitialized page and should be treated the same way other uninitialized pages are treated (cf. question 5).

   In order to get full marks, you do not need to implement sophisticated memory management strategies that deal with fragmentation. However, your implementation has to meet the following two criteria:

   - Accessing a memory address that has been allocated via `Alloc` and then deallocated by calling `Free` must result in an `AddressErrorException` and must be treated accordingly.
   - After all memory regions allocated by calling `Alloc` have been deallocated through `Free`, the address space of the program must look exactly the way it looked before performing the first allocation. In particular, an infinite sequence of the form

     ```
     while (1) { Free(Alloc(1024)); }
     ```

     must not result in the termination of the program.

   Because the pages allocated by `Alloc` are uninitialized, `Alloc` will always return a valid pointer (unless the 32-bit virtual address space of the calling process is exhausted). Thus, it is similar to the optimistic allocation strategy used in Linux.