

# Introduction to Operating Systems

## Virtual Memory<sup>†</sup>

Due: May 19, 2005, 23:59

Spring 2005

### 1 Ground Rules

The basic rules and project information are the same for this assignment as they were for the second assignment: you may work with a single project partner (and this must be the same partner as before).

As with other assignments, the following suggestions apply:

- Start early! You can always take a break after the program is done.
- Do your design document before writing a single line of code. You're welcome to discuss the design at the TA's or professor's office hours before you start implementation, and we're going to ask to see the design document before helping you debug code.

### 2 Goals

The goal of this assignment is to implement virtual memory using two-level page tables.

There are three parts to this assignment:

- Implement two-level page tables, replacing the primitive page tables currently provided in `process.c` and `memory.c`. Memory (*i.e.* pages) should be allocated when a process is created and deallocated when a process ends.
- Allow programs to request more memory via a trap. This trap should take a single argument – the amount of memory desired – and return a pointer to the additional memory. The amount of memory may be more than a single page, so the allocated memory should be contiguous in virtual memory (though not necessarily in physical memory). It's probably easiest to do this by simply adding more pages at the end of the current memory space for the process. The trap should be called `Allocate()` and be given the trap number `0x510`.
- Allow individual pages to reside on "disk" rather than in "memory." To do this, you'll need to use the file system calls provided in the simulator (`FsRead()`, `FsWrite()`, *et cetera*). Store your pages in a file called "vm," which should have enough space for 20MB of memory. Keep in mind that you'll have to keep track of which areas on the disk are in use and which are free in addition to remembering which areas belong to which in-memory pages.

---

<sup>†</sup>Derived from the virtual memory assignment developed by Prof. Scott Brandt.

## 3 Specifications

### 3.1 Changes to `ProcessFork()`

There is a single (external) change to `ProcessFork()`: the stack pointer for user processes should be set to point to just below the end of a process's address space. Since each process has 16MB of address space, the user stack pointer should be set to `0x00ffffe0` (`0x1000000 - 0x20`).

There may, of course, be other changes to `ProcessFork()` required for the implementation of two-level page tables. The above change is not strictly required for two-level page tables, but is a good idea (so you must implement it). This change allows the stack to grow down from the top of memory and the heap (user data allocated by `Allocate()`) to grow up from the top of the program code. The same approach is used by UNIX to allow for growth in stack and user space. Note that this means that the stack can grow by page faults. Initially, allocate 1–2 pages for the stack, and allow the stack to grow if a page fault occurs adjacent to stack pages that have already been allocated.

### 3.2 Two-level Page Tables

Your first goal will be to implement two-level page tables for DLX. Currently, DLX uses only single-level page tables, but the “hardware” is capable of two-level tables. In order to activate two-level page tables, you'll need to modify the constants `MEMORY_L1_PAGE_SIZE_BITS` and `MEMORY_L2_PAGE_SIZE_BITS`. They're currently set to the same value (16), but you'll need to set them appropriately for your page tables, which should have 512 byte pages (L2 bits = 9) and map 64KB per level 2 page table (L1 bits = 16). Your level 1 page tables should have 256 entries – enough to map 16MB of memory per process, which should be plenty for a CPU with only 2MB of memory by default.

Level 1 page tables only have pointers to level 2 page tables in them. If the L2 page table for a particular entry doesn't exist, the PTE in the L1 page table should be 0. If it does exist, the PTE should just contain the address of the L2 table, which may start at any address divisible by 4 (except 0, of course).

Level 2 page tables have PTEs in them that point to real pages of memory. In addition to the page address itself, there are three special bits in each PTE: valid, dirty, and reference. The valid bit must be set by the operating system to indicate that the page in memory is valid. The dirty and reference bits are set by the “hardware” when an address in the page is modified and accessed (read or written), respectively. The bits are never reset by the “hardware,” but they may be reset by the OS if desired. Definitions for the bits are in `memory.h`.

If a user process attempts to access an invalid page, a page fault trap is generated. This occurs if either the entry in the level 1 page table is 0, or if the entry in the level 2 page table is marked with an invalid bit. If the access is past the end of the level 1 page table, an illegal access trap is generated instead. You don't need to handle illegal accesses (other than killing off a process that makes one – crashing the OS is not acceptable), but you do need to handle page faults (see below). After you've loaded the missing page and fixed up the page tables, you return from the trap the same way you returned from other traps.

In addition to implementing 2-level page tables, you'll need to modify `MemoryTranslateUserToSystem()` to be able to translate user addresses into system addresses. This currently works for the simple single level page tables already implemented, but you'll need to modify it to handle your 2-level page tables.

Note that the page table and associated data needs to be freed up when a process exits, so you should keep track of which pages have been allocated to each process. Pay attention: you don't want to have a memory leak! The page table does this implicitly.

### 3.3 `Allocate()` Trap

In this version of the operating system, processes can request more memory via the `Allocate()` trap. This trap takes one argument: the number of bytes the process wants to allocate. The return value should be a pointer (in the user's address space) to the extra memory that was allocated.

Implementing `Allocate()` will involve modifying the page tables for the process to make more pages valid, and any bookkeeping that might entail.

### 3.4 On-disk Virtual Memory

Since processes in DLXOS might use more memory than the system has available, you'll need to implement on-disk virtual memory. This means you'll need to be able to keep pages of memory on disk if there's no space for them in memory. To do this, you should use the `FsOpen()`, `FsRead()`, and `FsWrite()` calls on a file called "vm" that you create. Sample usage for these calls (`FsOpen()` and `FsRead()`) is in `process.c`; `FsWrite()` uses similar arguments to `FsRead()`.

On a page fault, you'll need to choose a page to remove from memory and find the page on disk that you're going to replace it with. You may also need to write the page being replaced to disk if it's dirty. These tasks will involve keeping track of which parts of the disk are available and which parts are free. You'll also want to quickly be able to figure out where on the disk the page you're trying to bring in is located. The easiest way to do this is to somehow store the disk location in the PTE.

### 3.5 Hints

- Write your design document first!
- Implement the pieces in the following order:
  - Two-level page tables
  - `Allocate()` trap
  - Paging to disk
- You might want to build a bitmap that shows you which disk blocks are available. In such a structure, you have one bit per block, with a 0 indicating the space is available and a 1 indicating that it's in use. For a 20MB virtual memory, you'll need 40K bits, or just 5KB. A similar bitmap is used to keep track of which in-memory pages are available – check out the code.
- You'll need to implement a page replacement algorithm. The clock-based algorithms are typically easy to implement and perform quite well. FIFO is another easy-to-implement choice.
- Don't forget that you'll need to dynamically allocate space for page tables as well as for the memory space for a process. Allocate the minimum space for page tables that you need, and grow the tables as necessary.
- Use simple processes that allocate lots of memory to test your code. Make use of `Putchar()` and/or `Printf()` to show progress in various processes.
- System processes can't benefit from virtual memory; you'll have to use user processes to test it out.
- Use debugging statements or instruction and data traces (see the `dlxsim` page) to track where your code is going.

## 4 What to Hand In

As with other assignments, you'll need to hand in your design documentation and your code. Because your code may include both operating system code and user programs (such as the shell), please make sure your `Makefile` builds them separately.