Worksheet 7

Background

Paged memory access uses a page table to map pages in the virtual memory address space to pages in physical memory. Virtual memory and physical memory are each divided into pages of equal size, and the pages are numbered. The mapping from virtual to physical memory is specified by a page table, which contains information about each page in the virtual address space, including the physical page number that corresponds to each virtual page number.

For this lab, we imagine several virtual address spaces are in use, sharing the same physical memory. In a real computer, virtual address spaces correspond to processes. We won't impose that assumption on the simulation, but we do assume that there can be several threads using different virtual address spaces to access physical memory, so that race conditions and synchronization can be an issue.

A page in virtual memory space does not have to be mapped to a physical page until it is used. For this simulation, a physical page should be mapped to a virtual page only when some data is written to the virtual page. The first time that happens, the system will allocate a physical page and modify the page table to record the mapping from the virtual page to the physical page. If no more physical pages exist, then the write will fail.

For example:

The page size will be 4 kilobytes (2^{12} or 4096 bytes). Physical memory consists of 2^{12} (4096) pages. A virtual memory space is 2^{22} bytes. This means that physical memory has enough space for four complete virtual address spaces, but since a virtual memory pages only get physical pages when they need them, it is possible to have many more than four active virtual address spaces.

There are 2^{10} pages in a virtual address space, so a page table will have 1024 entries. A 22-bit virtual address consists of a 10-bit page number and a 12-bit offset. Virtual addresses are given as 32-bit ints, but only the last 22 bits are significant; the first 10 should be ignored. A physical address is 24 bits, consisting of a 12-bit page number and a 12-bit offset. The starting code for the project already defines:

- PAGE(n) — extract the 10-bit virtual page number from an int.
- OFFSET(n) — extract the 12-bit offset from an int.
- PHYS_PAGE(phys_page, offset) — convert a 12-bit physical page number and a 12-bit offset into a physical memory address.

You will also need some data structure to hold information about physical memory. For this lab, all that you really need to know about a physical page is whether or not it is currently allocated to a virtual memory space.

The C Version

The folder contains a C file *pgm.c* and a corresponding header file *pgm.h* that define the paged memory simulation. For this lab, the header file should not be modified. All the work is in *pgm.c*. There are also three test programs which you should run to test your simulation.

The interface to the memory system is defined by a data type *struct page_table* and by functions *pgm_init*, *pgm_create*, *pgm_discard*, *pgm_put*, *pgm_get*, *pgm_put_int*, and *pgm_get_int*. The last two of the functions are simple functions that are used in the test programs; they are already defined. You will need to complete the definitions of *page_table* and of the other five functions.

The meaning of the functions is specified in *pgm.h*, but note that *pgm_put* needs to allocate a new page whenever it writes to a virtual address that does not yet have an associated physical page. And if *pgm_get* tries to read from an unallocated page, it should simply return zero (without allocating the page). Both functions can simply do byte-by-byte copies, since we are not trying for maximum efficiency here.

You will need to add data structures and functions to implement physical memory. You need to think about synchronization. Note that the physical memory data are shared data structures, so access to them has to be synchronized. However, page tables and individual pages in memory are not shared, so access to them does not need to be synchronized. (Note: We assume that two threads will not share a page table or, at least, if they do they will be responsible for their own synchronization.)

For synchronization, you can use *pthread*, which is also used to create threads in *test2.c*. The pthread API is defined in the header file *pthread.h*, and programs that use pthreads must be compiled with the option *-pthread*. (See the compilation commands in the comments at the start of each test program.) A pthread mutual exclusion lock is of type *pthread_mutex_t*. A pthread lock can be initialized by calling

```
pthread_mutex_init(&lock, NULL);
```

and the acquire and release operations on a lock are given by

```
pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);
```

My completed *pgm.c* for lab 6, without new comments and counting blank lines, is 166 lines.