# CS421 Introduction to Operating Systems
# Fall 1998 : Project 2
# Introduction to Multi-Threaded Programming
# by Implementing a Memory Management System

Davin Milun

## 1 Objective

To familiarize you with:

- Designing and implementing a multi-threaded applications.
- Implementing memory management algorithms.
- Simulating virtual memory subsystems.

## 2 Problem Statement

### 2.1 Introductory Parts: Due October 13th

1. (10 points) Write a program than does the same as part 2 of project 1, except that you are required use POSIX threads rather than fork'ed processes.

   In addition to the main thread, create two additional threads:

   - one to calculate (but not print) the number of iterations that the *sin* formula takes. And then make that result available to the other thread,
   - the other to calculate and print the number of iterations that the *cos* formula takes, and receive the result from the first thread, print it, and compare the two values.

   (So the main thread ends up doing very little, other than settings things up, spawning the two threads, and waiting for them to die.)

   Use shared variables as the means of communication between the threads. (Hint: You don't need `pipe`.) Also, since the two threads are started at the same time, and the second needs to know when the data from the first is available, some form of synchronization is required. You may do this in one of two ways (your choice):

   - Use a variable as a flag, and have the second thread wait in a tight loop, until the flag indicates that the data is ready. (Note: this is a *very* bad way to solve a synchronization problem. It is being permitted only in this one case of this part of the project.)
   - Use the `pthread_mutex_t` mutual exclusion calls. (For example: have the main thread lock the mutex before it calls the two threads, have the first thread unlock it when the data is ready, and have the second thread lock it as it starts.)

   Note: You **must** start both threads before you wait for either of them. You will lose points if you call `pthread_join` on the first thread before you start the second thread running.

2. (15 points) Implement two solutions for the bounded-buffer producer/consumer problem.

   Both your implementations should consist of a control program that (i) initializes the buffer and the synchronization variables and (ii) creates and terminates the threads for the producer and the consumer. You only need to have a single producer and a single consumer, but you should code with locks in place so that multiple of each would not cause problems.

   The producer generates a stream of printable characters (either fixed or random) and places them into the buffer; and the consumer pulls the characters out of the buffer one at a time and prints them out. Let the buffer be of size 50. For the general test case, let the producer generate 500 characters and then quit, and devise some way of having the consumer quit once it has consumed the last character. While testing, it's probably useful for to also have the producer print them out, so that you can verify what's happenning.

   Implement this using two different methods:

   (a) Using POSIX threads and the synchronization primitives `pthread_mutex_t` and `sem_t`. Your code should simply realize the pseudocode from figure 5.17 (page 218) of Stallings.

   Use `pthread_mutex_t` for mutual exclusion (from the `pthread` library) and `sem_t` for minding the buffer size (from the `posix4` library).

   (b) Use POSIX threads and Peterson's software solution for realizing mutual exclusion. Your code should implement the pseudocode on page 214 of Stallings, and use the Peterson's locking code used in figure 5.6 (page 204).

## 2.2 Main Parts: Due November 3rd

3. (12 points) Implement a simulation of a memory management system. For this part, your system should have the following features:

   - The system must use the Simple Paging scheme (as described in Stallings in Section 7.3, page 304). This scheme means that you need things such as page tables, etc.
   - Pagesize = 128 bytes. Number of page frames (memorysize) = 80.
   - Simulate processes (the entities for which memory is allocated) by having the main program generate (on the fly, as needed) a list of process creation requests. Each of these requests should contain two random values: the size (from 1 byte, to 25 times the pagesize) of the process, and the lifetime of the process (from 2 to 30 units).
   - The main program should loop (70 times):
     - Display a current snapshot of memory. (For each page, show the PID of the process that owns it. A table would be good.)
     - Select the size and lifetime for the process, as specified above. Display these values.
     - Allocate memory for the process (or print failure message if insufficient memory available for that process).
     - Display number of free pages remaining.
     - Decrement lifetimes of all existing processes.
     - For each process now with lifetime 0, display the process's Page Table, free up the memory allocated to it, and display number of free pages.
   - After the loop, the main program should continue decrementing the lifetimes of any remaining processes, until they all eventually reach 0, and are freed. And it should then print out summary numbers, indicating: the number and percentage of requests that failed.
   - All this should occur in a single process, with no additional threads.

4. (15 points) Implement a simulation of a memory management system. For this part, your system should have the following features:

   - The system must use the Simple Paging scheme (as described in Stallings in Section 7.3, page 304). This scheme means that you need things such as page tables, etc.
   - Pagesize = 128 bytes. Number of page frames (memorysize) = 80.
   - Simulate processes (the entities for which memory is allocated) by having the main program generate a random size for the process (from 1 byte, to 25 times the pagesize, same as previous part), allocate memory for it (same as previous part) but then create a new thread for each process.
   - Each of these threads should choose a random value to act as its lifetime. However, this should be a number of milliseconds, between 0 and 200.
   - The main program should loop (70 times):
     - Display a current snapshot of memory. (For each page, show the PID of the process that owns it. A table would be good.)
     - `nanosleep` for 10ms.
     - Select the size for the process as described above. Display this value.
     - Allocate memory for the process (or print failure message if insufficient memory available for that process).
     - Display number of free pages remaining.
     - Create a new pthread for this process.
   - After the loop, the main program should wait for any remaining threads to complete. And it should then print out summary numbers, indicating: the number and percentage of requests that failed.
   - Each thread should:
     - `nanosleep` for the randomly chosen number of milliseconds (between 0ms and 200ms, which is between 0 and 200,000,000 nanoseconds).
     - Display a message indicating how many milliseconds it slept.
     - Display a message that it is about to quit.
     - Display the process's Page Table.
     - Free up the memory allocated to it, and display number of free pages.
     - Quit.

5. (23 points) Implement a simulation of a memory management system. For this part, your system should have the following features:

   - The system must use the Simple Paging scheme (as described in Stallings in Section 7.3, page 304). This scheme means that you need things such as page tables, etc.
   - Pagesize = 128 bytes. Number of page frames (memorysize) = 120.
   - Simulate processes (the entities for which memory is allocated) by having the main program generate a random size for the process (from 1 byte, to **50** times the pagesize), allocate memory for it (same as previous part) but then create a new thread for each process (same as previous part, except that what the threads do is different).
   - Each of these threads will actually run a simulated program, as described below. So their lifetimes will be determined by how long their "program" takes to run.
   - The main program should loop (70 times):
     - Display a current snapshot of memory. (For each page, show the PID of the process that owns it. A table would be good.)
     - Select the size for the process, as described above. Display this value.
     - Allocate memory for the process (or print failure message if insufficient memory available for that process).
     - Display number of free pages remaining.
     - Create a new pthread for this process.
   - After the loop, the main program should wait for any remaining threads to complete. And it should then print out summary numbers, indicating: the number and percentage of requests that failed.
   - Each thread should:
     - Perform between 8 and 100 simulated "blocks" of code. (See below.)
     - Display a message that it is about to quit.
     - Display the process's Page Table.
     - Free up the memory allocated to it, and display number of free pages.
     - Quit.
   - Remember that you need to do locking on shared data structures, or else they *will be* corrupted.
   - Each "block" is one of the following. Each one operates starting at the current position. Before the first block runs, current position is address 0 in the simulated process's address space. *size* is number of bytes in this process, as chosen by the main program. (See "Hints and Implementation Details" section for explanation of what the percentages in the following list mean, in case they're unclear.)
     - 60% of the time, linearly access $n$ memory locations ($0 \leq n \leq \frac{size}{2}$).
     - 20% of the time, loop $m$ times linearly accessing $n$ memory locations ($0 \leq n \leq \frac{size}{8}$) ($2 \leq m \leq 6$).
     - 20% of the time, jump to $n$ locations away from current location ($-(\frac{size}{16}) \leq n \leq (\frac{size}{16})$).

     Before performing the action for each block (linear access, loop or jump), it should print out a one-line message, containing PID, action-type, and action parameters.

     Treat all math on memory addresses as being modulo the size of the process space. For example, within each block's run, if a linear or loop would overflow the end of memory, have it continue back at the front, as if memory was a circular list. (For example, if current position is 80, and *size* is 100, and the value of $n$ randomly chosen for a linear access is 35; then the addresses accessed should be 80,81,...,98,99,0,1,...,13,14.)

6. (25 points) Implement a simulation of a memory management system. For this part, your system should have the following features:

   - The system must use the Virtual Memory Paging scheme (as described in Stallings starting on page 324). This scheme means that you need things such as page tables, a free lists, etc.
   - Pagesize = 128 bytes. Number of page frames (memorysize) = 120.
     Number of pages of swap space = 400.
   - Simulate processes (the entities for which memory is allocated) by having the main program generate a random size for the process (from 1 byte, to 50 times the pagesize, same as the previous part), allocate memory for it (same as previous part) but then create a new thread for each process (same as previous part).
   - Each of these threads will actually run a simulated program, as described below. So their lifetimes will be determined by how long their "program" takes to run.
   - The main program should loop (70 times):
     - Display a current snapshot of memory. (For each page, show the PID of the process that owns it. A table would be good.)
     - Display a current snapshot of swap space. (For each page, show the PID of the process that owns it. A table would be good.)
     - Select size for the process. Display this value.
     - Allocate swap space for the process (or print failure message if insufficient memory available for that process).
     - Display number of free pages remaining, in each of memory and swap.
     - Create a new pthread for this process.
   - After the loop, the main program should wait for any remaining threads to complete. And it should then print out summary numbers, indicating: the number and percentage of requests that failed, the number of Pagein's required, the number of Pageout's required, the number of Dirty/Modified pages written back to disk.
   - Each thread should:
     - Perform between 8 and 100 simulated "blocks" of code. (See below.)
     - Display a message that it is about to quit.
     - Display the process's Page Table.
     - Free up the memory allocated to it, and display number of free pages in each of memory and swap. (Note that you need not call Pageout on the pages of a process that is about to quit!)
     - Quit.
   - Each "block" should operate exactly as in the previous part, except that 20% of the time, the block should additionally be considered a "write block", and calls to the access function should also set the Dirty/Modified bit.
   - For each "access" to memory, call a function Access (with the processes virtual address) and have it do the following:
     - Use page table to map to real address.
     - If that real page is not currently in memory, then call Pagein for that page.
     - Set the Used bit for that page.
     - If the block is a "write block", then set the Dirty/Modified bit too.

- The Pagein function implements the page replacement policy, as follows:
  - If there are no free pages in main memory, then use the Clock algorithm (Stallings page 345, 346) to find a page to replace, and call Pageout on that page.
  - If there were free pages, claim one.
  - Load the page (on which Pagein was called) into memory.
  - Update tables accordingly.
  - Print out a line of output indicating what was/had-to-be done.
  - Display a current snapshot of memory.
- The Pageout function does all the work required to free up a page in memory:
  - Finds the process to which this page belongs.
  - Finds the swap page associated with it.
  - If its Dirty/Modified bit is set, writes it out to that swap page.
  - Updates various tables, as needed.
  - Print out a line of output indicating what was/had-to-be done.

# 3 Hints and Implementation Details

1. One way to think of parts 3–6 is as one big part, with the goal being part 6. Parts 3–5 are simply there to develop the project in logical steps. (However, you *are required* to submit independent working versions of all parts.)

2. If you design/write your parts 3–6 well, then you'll likely only need to change certain sections of each one, as you "upgrade" it to become the next part. Pay careful attention to the numbers/sizes/counters, as some do change between parts.

3. For this project efficiency is not a real issue, so there is no need to implement time-saving features such as TLB's, hash tables, inverted tables etc. Doing linear searches through tables is perfectly acceptable.

4. When the project states that an action needs to get taken *N*% of the time, it means that a random number has to be chosen, and that action should occur if the random number is within *N*% of its possible range. For example, if it were stated that action *Q* needs to be taken 50% of the time, and action *R* the other 50%, then you could choose a random number 0 or 1, and do *Q* if the random number if 0, and *R* otherwise. If it is some number other than 50%, then choose a random number from a larger range, and take the appropriate action. So, for ranges 60%, 20%, 20%, you could choose a number from 0 to 99, and then check if it's 0–59, 60–79, and 80–99 for the respective actions.

5. See the man pages for more details about specific system or library calls and commands, such as: `pthread_create(3T)` `nanosleep(3R)`, `pthread_mutex_lock(3T)`, `lrand48(3C)`, `srand48(3C)` etc.

6. Be sure that you set `PTHREAD_SCOPE_SYSTEM` on all your threads.

7. Be sure to always have your program wait (using `pthread_join` repeatedly as required) for all threads to complete. Remember that when the process exits (either by explicitly calling `exit` or by falling off the end of `main`) *all* threads die immediately.

8. In threaded programs, remember that you must protect accesses to shared data structures, or else they are liable to get corrupted. In designing your locking mechanisms, try to use as few locks as possible (as then it is then easier avoid deadlock) but with sufficient locks to allow for concurrency.

9. A good first check that you're doing the previous two points correctly, is that the last status message from the program should indicate that all memory is free. If that is not the case, then either all your threads are not completing, or you have some memory corruption due to bad locking, or else you have some other basic algorithm(s) incorrect.

10. In the threaded parts, the output from the various threads is likely to be interleaved on the screen. That is perfectly acceptable and expected—do not implement any output locking to prevent that from happening.

11. In the threaded programs, I recommend that you use `cc` or `CC` (rather than `gcc` or `g++`) as your compiler. And, be sure to pass the compiler the `-mt` flag, or else various bad stuff might happen!

12. Your program *must* be robust. If any of the calls fail, it should print error message and exit with appropriate error code. *Always* check for failure when invoking any system or library call **especially** calls to locking functions (such as `pthread_mutex_lock`). By convention, most UNIX calls return a value of NULL or negative one ($-1$) in case of an error (but always check the `RETURN VALUES` section of each man page for details), and you can use this information for a graceful exit. Use `cerr, perror(3)`, or `strerror(3)` library routines to generate/print appropriate error messages.

13. There are some sample implementations of some of the Main Parts in `~milun/cs421/DEMOS/` on armstrong/gagarin/yeager/fork. Note that these are there to demonstrate the basics of what your projects should do. But they do *not* necessarily meet the project specifications exactly: they might be logging extra or insufficient information; they might have some parameters/counters set to different values than the project description, etc. In all cases that these demos differ from the project specification, the project specifications takes precedence.

# 4 Material to be Submitted

1. Submit the **source code** for the programs. Use meaningful names for the files so that the contents of the file is obvious from the name.

2. Submit a `README` file describing which program file corresponds to which part of the project. Also include any special notes, instructions, etc..

3. You are required to submit a Makefile for your project. It should be set up so that just typing `make` in your submission directory should correctly compile all parts.

4. You need to submit a printed Technical Report for the Memory Subsystem implementation (parts 3–6). It should contain two parts:

   (a) A users manual.

   (b) A technical manual describing your design of the database, the main structures/classes in your program, the main algorithms, the details of your database API, etc..

   This Technical Report should be a professional looking document. It should preferably be typeset (using LaTeX, FrameMaker, IslandWrite, MS Word etc.), and should definitely not be handwritten (except possibly for diagrams).

# 5 Due Dates

- Submit on-line the Introductory Parts by 11:59pm of Tuesday October 13.
- Submit on-line the Main Parts by 11:59pm of Tuesday November 3.
- The printed Technical Report is due in class the following day, Wednesday November 4. (Or by 5pm on Nov 4, if you sign in in class that morning.)