Assignment 3

CIS3110/Operating Systems/W25
updated: February 23, 2025

CODING ASSIGNMENT

1 Threads and Processes

This exercise will give you some experience working with threads, and comparing their properties to those of full processes.

2 Objective

In assignment 2 we implemented a set of multiple Unix *processes* communicating via a shared memory region to simulate a Token Ring network.

In this assignment we will take that code and adapt it so that instead of each node being modelled using a separate process, instead we will model each node using a separate POSIX *thread*.

Keep in mind that the significant difference between a new child process being created by `fork(2)` and an additional *thread* being created within a process is that the pages of memory shared between the two process contexts are different.

- In the child *process* case, the only memory shared between the parent and a child is memory expressly created as shared memory.
- In the *thread* case, all pages of memory are shared with the exception of those making up the *stack*.

Obviously, as memory remains shared, we still have issues to manage regarding critical sections. It now requires a little more thought to identify where the critical sections are, however, as we need to consider which parts of a much larger shared region of memory (*i.e.*; the entire text and data segments).

3 Specific areas of focus

To adapt your program, you will need to focus your attention on these three concepts:

- how to create, manage and clean up new *thread*s instead of new *child processes*
- how to manage the new strategy for sharing memory
- how to manage the critical sections within your shared memory

3.1 Thread versus process management

In our previous code, we used `fork(2)`, `exit(3)` and `wait(2)` to create, destroy and clean up behind our child processes.

Your task is to adjust your code from A2 so that it uses thread based tools instead of the process based tools just mentioned.

3.1.1 Creation

Instead of `fork(2)` we will create new threads using `pthread_create(3)`. This will create a new thread (with its own stack) running within the same process, sharing the pages of the data and text segments.

Note that unlike `fork(2)` which identifies which new process is the child through the exit status of the `fork()` call, `pthread_create(3)` takes as an argument a pointer to a function that gets run by the new thread.

3.1.2 Destruction

Instead of `exit(3)` which terminates the whole process, we will use `pthread_exit(3)` to cause only the single calling thread to terminate.

If your design uses an explicit call to `pthread_exit(3)` then the thread is stopped, and the value supplied to the `pthread_exit(3)` call is available through the call to "join" explained below.

If you have not made any call to `pthread_exit(3)` within the function you supplied to `pthread_create(3)` and that function returns, then `pthread_exit(3)` is effectively called for you at that time, making the return value of the function you supplied available as though you passed it to `pthread_exit(3)`.

3.1.3 Cleanup

Just like in the case of processes, the parent must clean up after child threads. To do this, the parent must wait for a given child to complete, and collect the information supplied at completion, just like when a parent process "waits" on its child.

We don't have a full process to clean up from, so instead of using a `wait(2)` to clean up our "zombie" child process, we use `pthread_join(3)` to clean up from and synchronize with that child thread.

The thread waiting to "join" with the indicated completing thread blocks until that thread completes, so this also provides synchronization information – we know that the thread calling "join" will do nothing until the thread it is looking to join with has completed.

3.2 Memory Sharing

Since we are using threads, a huge fraction of the program memory image will be shared. We therefore do not need to set up any formally specified shared memory, and the calls to `shmget(2)`, `shmat(2)`, `shmdt(2)` and `shmctl(2)` should be removed, and replaced with a `malloc(3)` based strategy.

Consider how you can use `malloc(3)` to obtain a block of memory that can be used in place of the shared memory from A2.

3.3 Synchronization and Critical Sections

Little will change here – although the *type* of shared memory you are managing has changed, the fact that it is shared has not. You should find that if you have a successful semaphore based strategy to coordinate and manage your cooperating child processes from A2 that you can use this with little adaptation to manage your cooperating thread in this version.

4 Overall function

The objective here is to model the same system we had in A2, so therefore it should function in the same way – it is simply built using different tools.

5 Submission Requirements

The assignment should be handed in as a single `tar(1)` file, through CourseLink.

Be sure that:

- *your name* is in a comment at the beginning of the program
- you have included a functional `makefile`
- your submission contains a brief discussion describing your program design in a file entitled `README.md` or `README.txt`

Be absolutely sure that your code compiles and runs on `linux.socs.uoguelph.ca`.

As before, there are several system dependencies that will differ having to do with header files and the usage of some of the low level semaphore primitives that will mean that if you develop your code elsewhere and hand that in without checking it is unlikely to compile and run properly on our grading platform. Whatever you hand in is what we will grade, so please make sure that you are handing in working code.