SFWRENG 3SH3 Operating Systems

by Dr. Anwar M. Mirza

Lab Assignment 02

Date: February 23rd, 2022
Report Due before
March 6th, 2022, midnight

Process and Thread Synchronization

1. Background

1.1. Threads and Pthreads

A thread can be defined as an independent stream of instructions that can be scheduled to run as such by the operating system. They exist within a process and use the process resources. They maintain their own stack pointer, registers, scheduling priorities, set of signals and thread specific data. They can share the process's resources with other threads. They are "lightweight", as most of the overhead has been accomplished by their parent process on its creation.

Pthreads (POSIX threads) give a standardized programming interface for using threads. Pthreads are defined as a set of C language, programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library. This library may be a part of another C library such as `libc`, in some implementations. The pthreads API (Appilication Programming Interface), as originally defined by ANSI/IEEE POSIX 1003.1 – 1995 standard, has over 100 functions.

1.2. Pthreads Management

1.2.1. Creating Pthreads

All threads must be created explicitly by the programmer. The function `pthread_create` creates a new thread and can be called any number of times from anywhere within your program. It has the following arguments:

- `thread_ID` – A unique identifier for the thread as returned by the function.
- `attr` – An attribute object that may be used to set thread attributes. You can specify a thread attribute object or set it to `NULL` for the default values. By default, a thread is created with certain attributes. Some of these can be changed by the programmer. The routine `pthread_attr_init` and `pthread_attr_destroy` are used to initialize and destroy the thread attribute object, respectively. Other routines are used to set or get specific attributes in the thread attribute object. The attributes include detached or joinable state, scheduling policy, scheduling parameters, stack size, stack address etc.
- `start_routine` – The C function that the thread will execute once it is created.
- `arg` – A single argument that may be passed to `start_routine`. It mus be passed by reference as a pointer cast to type void. `NULL` may be used if no argument is to be passed.

The maximum number of threads that can be created by a process is implementation dependent. Various routines are available in the pthread API to schedule threads (using FIFO, RR or other scheduling schemes). The pthread API does not provide routine for binding threads to specific cores in a multi-core processor architecture. The local operating system may provide a way of doing this. In Linux, the `sched_setaffinity` routine provides this functionality.

1.2.2. Terminating Pthreads

A thread can be terminated in several ways:

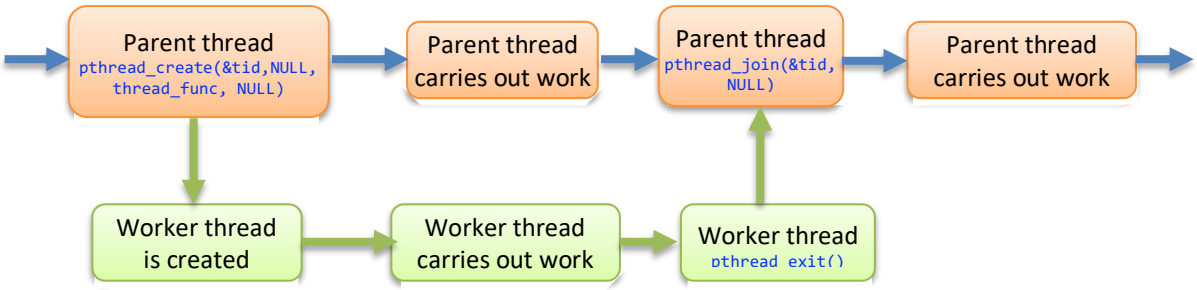
- On the completion of its work, a thread can return normally from its `start_routine`.
- The thread can make a call to `pthread_exit` function from within its `start_routine`.
- The thread can be canceled by another thread by the `thread_cancel` routine.
- The entire process is terminated due to making a call to either the `exec()` or `exit()`.
- The `main()` finishes first, without calling `pthread_exit` explicitly itself.

The pthread_exit() returns an optional termination status to the thread/process joining the terminated thread. It must be noted that the pthread_exit() routine does not close files already opened inside the thread.

If the `main()` finishes before the threads it has spawned without explicitly calling `pthread_exit()`, all threads it has created will terminate with the `main()` termination. The explicit call to `pthread_exit()` is made as the last thing in the `main()` - the `main()` will block and be kept alive to support the threads it has created until they are done.

1.2.3. Joining and Detaching Threads

Joining is one of attaining synchronization between threads.



In the figure above the parent thread is creating a worker thread by call `pthread_create()` routine. The parent thread then carries on doing work concurrently with the worker thread. The parent thread calls the `pthread_join()` routine by explicitly specifying the ID of the worker thread. The `pthread_join()` routine blocks the calling thread until the specified thread terminates. A thread can be made "joinable" or "detached" by setting the pthread attributes using an attribute variable of type `pthread_attr_t` and using the routines `pthread_attr_init()` and `pthread_attr_setdetachstate()`. A thread initially created as joinable, can be explicitly detached using the routine `pthread_detach()` at any time.

1.2.4. Passing Arguments to Threads

Sometimes we need to pass one or more arguments to a thread. We can pass one argument to the thread while creating the thread using `pthread_create()` function. This capability can be extended to more than one argument by grouping them in the form of a structure. A pointer to this structure can then be passed to thread via `pthread_create()`. All arguments must be passed by reference and cast to the generic pointer type `(void *)`. A "generic" ponter type. A `(void *)` can be converted to any other pointer type without an explicit cast.

1.3. POSIX Semaphores

The Portable Operating System Interface (`POSIX`) is a family of standards given by IEEE Computer Society and is used for maintaining compatibility between operating systems. `POSIX` semaphores allow process and thread synchronization. All `POSIX` semaphore functions and types are defined in `semaphore.h`. A semaphore object is defined as:

```
sem_t semaphoreName;
```

1.3.1. Initialization

A semaphore can be initialized using:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- `sem` points to a semaphore object to be initialized.
- `pshared` is a flag used to share the semaphore pointed by sem with forked processes.
- `value` is an initial value assigned to the semaphore.

For example

```
sem_init(&semaphoreName, 0, 21);
```

1.3.2. Waiting on a semaphore

To wait on a semaphore (and decrementing its value by one) we use:

```
int sem_wait(sem_t *sem);
```

For example

```
sem_wait(&semaphoreName);
```

In case the semaphore value is negative, the calling process/thread blocks; the blocked process will wake up when another process/thread calls `sem_post()`.

1.3.3. Incrementing the value of a semaphore

To increment the value of a semaphore we use:

```
int sem_post(sem_t *sem);
```

It increments the vaue of the semaphore and wakes up any blocked process/thread that may be waiting on the semaphore.

An example is,

```
sem_post(&semaphoreName);
```

1.3.4. Finding the value of a semaphore

To find the value of a semaphore and place it in a location pointed to by `val`, we use:

```
int sem_getvalue(sem_t *sem, int *val);
```

An example is,

```
int value;
sem_getvalue(&semaphoreName, &value);
printf("Semaphore value : %d\n", value);
```

1.3.5. Destroying a semaphore

To destroy a semaphore, we use:

```
int sem_destroy(sem_t *sem);
```

No thread should be waiting on the semaphore for its destruction to be successful.

An example is,

```
sem_destroy(&semaphoreName);
```

2. Assignment

Q.No. 1. Write a C program to create 5 threads. Define three types of variables: global, static, and local. The global variable is defined global to the overall program, and static, and local variables are defined local to the threads. Your threads will update the values of these variables. The program should demonstrate how the updates on each of these variables are reflected in the multiple threads. Comment on why your program is showing the various updated values by running the program.

Q.No. 2. In this exercise, you are going to implement a semaphore-based solution to the bounded buffer producer/consumber problem.

The buffer is manipulated by two functions: `insert_item()` and `remove_item()` called by the producer and the consumber threads, respectively. Using three semaphores (you may call them `empty`, `full` and `mutex`), synchronize the working of the producer and the consumer threads. The semaphores `empty` and `full` count the number of empty and full slots in the bounded buffer. The semaphore `mutex` is a binary semaphore and protects the critical sections of the producer and consumer threads.

The `main()` function initializes the buffer and creates separate threads for the producer and consumer. After creating the producer and consumer threads, the `main()` function sleeps for a period of time. Upon awakening, the `main()` thread terminates the application. The main function is passed three parameters on the command line: (1) How long the `main()` function sleeps before terminating, (2) the number of producer threads, and (3) the number of consumer threads.

The producer thread alternates between sleeping for a random period of time and, upon awakening attempting to insert a random integer into the buffer. The consumer thread also alternates between sleep for a random period of time and, upon awakening, attempting to remove an item from the buffer.

3. Guideline

- Work in your already allocated teams.
- You are recommended to use the Lab hours to carry out this work. TAs will be available during their respective Lab sessions for your in-person / online help.
- For this lab, you need to submit only a group report in MS Word or PDF to the TA by

  March 6th, 2022, midnight.
- Copying/reproduction of (all or parts of) each others' reports will lead to zero marks for the whole team.

Please use MS Teams Lab Q&A and Lab Section channels to coordinate your work with your team members and with the TAs.