Practical Number: 9

CO/PO: CO1, CO2, CO3

Problem Definition

To implement multi-threaded programming using POSIX threads (pthreads) and solve classic synchronization problems like mutual exclusion, reader-writer locks, and producer-consumer using mutexes, semaphores, and condition variables in C language.

1 hello pthreads!

In this section, we will use the pthread (POSIX threads) library to understand the working of threads, share data across threads and synchronization.

(a) Read the sample program threads.c provided. In this program, 100 threads are created which execute a function that updates a shared counter. pthreads are declared using pthread_t. To start a thread, use pthread_create(). This function takes four arguments: address of the pthread variable representing this thread (pthread ID), attributes of the new thread, start routine (function) and parameters for the start routine. On pthread_create() a new thread is created with the above arguments and commences execution at the start routine. The main thread waits for the thread to exit and then reaps the thread using pthread_join(). Compile the program using:

gcc threads.c -lpthread

Read the following man pages to understand pthreads in more detail:

pthreads, pthread_create, pthread_join, pthread_detach, pthread_exit, pthread_kill, pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_trylock, pthread_spin_lock, pthread_spin_unlock, pthread_cond_signal, pthread_cond_wait, pthread_cond_broadcast, pthread_cond_init, ...

(b) Next, use pthread locks to synchronize access to shared data across threads.

Write a program threads-with-mutex.c which synchronizes access to the shared counter in threads.c using a mutex lock. You can declare the mutex as a global variable.

The functions of interest are: pthread_mutex_init(), pthread_mutex_lock(), pthread_mutex_unlock(), and pthread_mutex_destroy().

---

2 argumentative pthreads

The pthread_create() function allows a single argument to the start function. The argument has to be passed by reference as a void pointer. Think about how to pass multiple arguments to the start function.

Write a program nlocks.c that does the following:

- Creates and initializes 10 shared counters and 10 locks.
- The parent process creates 10 threads.
- Each thread adds 1 to its corresponding data value 1000 times.
- The parent also updates the data items — adds 1 to data[0], then to data[1], etc. for all 10 data items and then repeats for a total of 1000 iterations.
- With correct synchronization, each of the data values should be 2000.
- Note: The index to process data items by each thread must be carefully passed to the start function.

3

With a reader-writer lock, multiple readers can concurrently access the data. However, a writer must not access the data concurrently when either readers or writers access the data. A writer has to wait until all the readers or writers complete access to the shared data, while a reader does not need to wait if readers are currently active.

It would be unfair for a reader to jump in immediately ahead of a waiting writer. If that happened often enough, writers would starve.

Implement a reader-writer lock with writer preference, where new readers do not access the shared data in case of writers waiting to write, even if readers are active.

Semantics:

- With readers already present, a new reader can read only if no writer is already waiting to write.
- If a writer is waiting to write, writers get preference; readers have to wait until all writers clear out.
- A writer cannot write if readers are active or another writer is active.

Use the sample program reader-writer.c as skeleton code to get started.
Note: Implementation should use pthread conditional variables and mutex.

---

4 extra sync (For Fast Learner)

- Implement the producer-consumer using pthread condition variables and mutex locks. Assume a bounded buffer for production.

- Assume two threads, with functions hydrogen and oxygen. Each thread periodically creates a hydrogen or oxygen atom and when two hydrogen atoms and one oxygen atom is available, water is formed. Incomplete formations block further partial molecule completions. Implement this using semaphores.

.

Key Questions to be evaluated during Implementation

- Why is mutual exclusion required when multiple threads access shared data?

Ans: Mutual exclusion is required to prevent race conditions and ensure that only one thread accesses or modifies shared data at a time, maintaining data consistency and correctness.

- How does a reader-writer lock work with writer preference?

Ans: A reader-writer lock allows multiple readers to access shared data simultaneously but gives priority to writers. When a writer is waiting, new readers are blocked until the writer finishes, preventing writer starvation.

- How do you pass multiple arguments to a thread's start routine?

Ans: You can pass multiple arguments by creating a structure (struct) containing all the required data and passing its pointer to the thread's start routine.

Post Laboratory Work Description

In this practical, you implement multi-threaded programs using POSIX threads (pthreads) in C to handle concurrent execution. The focus is on solving classic synchronization problems such as mutual exclusion, reader-writer scenarios, and the producer-consumer problem. You use mutexes to ensure exclusive access to shared resources, semaphores to control resource availability, and condition variables to coordinate thread execution. Through these exercises, you learn how to manage shared data safely, prevent race conditions, and synchronize thread operations effectively in a concurrent environment.

1)

Open pr9_1.cpp

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
pthread_t tid[2];
int counter;
void* trythis(void* arg)
{
unsigned long i = 0;
counter +=1;
printf("\n Job %d has started\n", counter);
for (i = 0; i < (0xFFFFFFFF); i++) ;
printf("\n Job %d has finished\n", counter);
return NULL;
}
int main(void)
{
int i = 0;
int error;
while (i < 2) {
error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
if (error != 0)
printf("\nThread can't be created : [%s]", strerror(error));
i++;
}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
return 0;
}
```

```
student@studen:~$ gedit pr9_1.cpp
student@studen:~$ g++ pr9_1.cpp -o pr91
student@studen:~$ ./pr91

 Job 1 has started

 Job 2 has started

 Job 2 has finished

 Job 2 has finished
```

2)

Open pr9_2.cpp ~/

```
#include <string.h>
#include <unistd.h>
pthread_t tid[2];
int counter;
pthread_mutex_t lock;
void* trythis(void* arg)
{
pthread_mutex_lock(&lock);
unsigned long i = 0;
counter += 1;
printf("\n Job %d has started\n", counter);
for (i = 0; i < (0xFFFFFFFF); i++);
printf("\n Job %d has finished\n", counter);
pthread_mutex_unlock(&lock);
return NULL;
}
int main(void)
{
int i = 0;
int error;
if (pthread_mutex_init(&lock, NULL) != 0) {
printf("\n mutex init has failed\n");
return 1;
}
while (i < 2) {
error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
if (error != 0)
printf("\nThread can't be created :[%s]",
strerror(error));
i++;
}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
pthread_mutex_destroy(&lock);
return 0;
}

```

```
student@studen:~$ gedit pr9_2.cpp
student@studen:~$ g++ pr9_2.cpp -o pr92
student@studen:~$ ./pr92

 Job 1 has started

 Job 1 has finished

 Job 2 has started

 Job 2 has finished
```

1. Flow and Working of the Program

The program demonstrates **thread creation** and **synchronization** using **mutexes** in C/C++ with the POSIX threads (pthreads) library.

Program Flow (Without Mutex)

- **Global Variables**:
  - `pthread_t tid[2]`: Stores thread identifiers.
  - `int counter`: Shared variable incremented by threads.
- **Function** `trythis()`:
  - Each thread runs this function.
  - It increments `counter` and prints a start message.
  - Executes a long dummy loop (`for (i = 0; i < 0xFFFFFFFF; i++);`) to simulate work.
  - Prints a finish message.
- **Main Function**:
  - Initializes variables.
  - Creates two threads using `pthread_create`.
  - Waits for both threads to finish using `pthread_join`.

Problem Without Mutex

- **Race Condition**: Both threads access and modify `counter` simultaneously, leading to inconsistent output.
- **No Synchronization**: Threads may interleave unpredictably.

2)

Flow with Mutex (Thread Synchronization)

The second version of the program introduces a **mutex** to protect shared data.

Key Additions

- `pthread_mutex_t lock;`: Declares a mutex.
- `pthread_mutex_init(&lock, NULL);`: Initializes the mutex before thread creation.
- `pthread_mutex_lock(&lock);`: Acquires the mutex before accessing `counter`.
- `pthread_mutex_unlock(&lock);`: Releases the mutex after access.
- `pthread_mutex_destroy(&lock);`: Cleans up the mutex after threads finish.

Updated Flow

1. **Mutex Initialization**:
   - Ensures the mutex is ready for use.
2. **Thread Execution**:
   - Each thread locks the mutex before modifying `counter`.
   - This guarantees that only one thread accesses `counter` at a time.
   - After finishing, the thread unlocks the mutex.
3. **Thread Joining**:
   - Main thread waits for both threads to complete.
4. **Mutex Destruction**:
   - Frees resources used by the mutex.

Key Skills to be Addressed

- Multi-threaded Programming
- Synchronization using Mutex, Semaphore, Condition Variables
- Deadlock Avoidance and Process Coordination

- Classic OS concurrency problem solutions

Applications

- Server-client concurrent systems
- Real-time embedded control systems
- Operating System kernel process management

Learning Outcome

- Understand thread creation and management using pthreads
- Implement synchronization techniques for concurrent threads
- Demonstrate concurrency control through classical problems
- Apply synchronization mechanisms to real-world application systems

Tools/Technology to Be Used

- Ubuntu/Linux Terminal
- GCC Compiler (for compiling C programs)
- Text Editors (nano/vim)

- Total Hours of Problem Definition Implementation
  3 Hours
- Total Hours of Engagement
  4 Hours
  (*Includes implementation + modification + faculty testing*)

Evaluation Strategy Including Viva

- Correct implementation and synchronization technique usage
- Code correctness, readability, and output validation
- Viva on pthreads, mutex, semaphores, reader-writer locks