2 Assignment 1

2.1 Introduction to Advanced IPC and Multithreading

This assignment will deepen your understanding of how processes and threads communicate and synchronize their activities, which are critical aspects of modern operating systems and concurrent programming. You will implement various IPC mechanisms and explore the power and challenges of multithreaded applications.

2.2 Interprocess Communication (IPC) - Pipes

Pipes provide a simple and efficient way for related processes to communicate.

2.2.1 Unnamed Pipes

Unnamed pipes facilitate one-way communication between processes that share a common ancestor (e.g., parent-child processes).

- **Concept:** A pipe is a unidirectional data channel that can be used for interprocess communication. Data written to one end of the pipe can be read from the other end.
- **System Call:** `pipe()`
  - `int pipe(int pipefd[2]);`
  - Creates a pipe and returns two file descriptors: `pipefd[0]` for the read end and `pipefd[1]` for the write end.
  - Typically used with `fork()` where one process closes the read end and the other closes the write end to establish communication.
- **Reading and Writing:** Use `read()` and `write()` system calls on the respective file descriptors.

2.2.2 Named Pipes (FIFOs)

Named pipes (FIFOs - First-In, First-Out) allow communication between unrelated processes on the same system, behaving like special files in the file system.

- **Concept:** A FIFO is a named pipe that exists as a file in the file system. Any process can open it for reading or writing, just like a regular file.
- **System Call:** `mkfifo()`
  - `int mkfifo(const char *pathname, mode_t mode);`
  - Creates a new FIFO special file at `pathname` with permissions specified by `mode`.
- **Opening and Closing:** Use `open()` and `close()` to interact with a FIFO, just like regular files.
- **Reading and Writing:** Use `read()` and `write()` on the opened FIFO file descriptor.

2.3 Interprocess Communication (IPC) - Shared Memory

Shared memory is the fastest form of IPC as it allows processes to directly access a common region of memory.

2.3.1 Introduction to Shared Memory

- **Concept:** Shared memory allows multiple processes to have direct read and write access to a common segment of memory. This eliminates the need for data copying between kernel and user space, making it very efficient.
- **Synchronization:** While fast, shared memory *requires* explicit synchronization mechanisms (like semaphores or mutexes) to prevent race conditions and ensure data consistency.
- `ipcs -m` **(Inspect Shared Memory):** Command-line tool to view existing shared memory segments.

2.3.2 System V Shared Memory

- `shmget()` **(Create/Access):**
  - `int shmget(key_t key, size_t size, int shmflg);`
  - Creates a new shared memory segment or gets the ID of an existing one.
  - `key`: A unique identifier for the shared memory segment.
  - `size`: The size of the shared memory segment in bytes.
  - `shmflg`: Flags like `IPC_CREAT`, `IPC_EXCL`, and permissions.
- `shmat()` **(Attach):**
  - `void *shmat(int shmid, const void *shmaddr, int shmflg);`
  - Attaches the shared memory segment identified by `shmid` to the calling process's address space.
  - Returns a pointer to the attached segment.
- `shmdt()` **(Detach):**
  - `int shmdt(const void *shmaddr);`
  - Detaches the shared memory segment located at `shmaddr` from the calling process's address space.
- `shmctl()` **(Control/Delete):**
  - `int shmctl(int shmid, int cmd, struct shmid_ds *buf);`
  - Performs various control operations on a shared memory segment, including deleting it (`IPC_RMID`).

2.4 Multithreading with POSIX Threads (pthreads)

Threads are lightweight units of execution within a single process, sharing the same memory space.

2.4.1 Introduction to Threads

- **Concept:** A thread is a single sequence of instructions that can be executed independently by a CPU. Multiple threads can exist within a single process, sharing the process's memory space, open files, and other resources.
- **Processes vs. Threads:**
  - **Processes:** Independent, have separate address spaces, communicate via IPC. Context switching is heavier.

  - **Threads:** Share the same address space, lighter weight, communicate directly through shared memory. Context switching is lighter.
- **Advantages:** Improved responsiveness, faster context switching, efficient resource sharing, utilization of multi-core processors.
- **Disadvantages:** Increased complexity (synchronization issues), potential for deadlocks and race conditions.

2.4.2 Basic pthreads Operations

The POSIX Threads (pthreads) library provides a standard API for thread management.

- **Header:** `<pthread.h>`
- **Compilation:** Link with `-lpthread` (e.g., `gcc myprogram.c -o myprogram -lpthread`).
- `pthread_create()` **(Create Thread):**
  - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
  - Creates a new thread.
  - `thread`: Pointer to a `pthread_t` variable to store the new thread's ID.
  - `attr`: Thread attributes (can be `NULL` for default).
  - `start_routine`: The function that the new thread will execute.
  - `arg`: Argument to pass to the `start_routine` (can be `NULL`).
- `pthread_join()` **(Wait for Thread):**
  - `int pthread_join(pthread_t thread, void **retval);`
  - Waits for the specified `thread` to terminate.
  - `retval`: Pointer to store the return value of the joined thread's `start_routine`.
- `pthread_exit()` **(Terminate Thread):**
  - `void pthread_exit(void *retval);`
  - Terminates the calling thread and makes `retval` available to any joining thread.

2.4.3 Thread Synchronization - Mutexes

Mutexes (mutual exclusion locks) are used to protect shared resources from concurrent access by multiple threads, preventing race conditions.

- **Concept:** A mutex ensures that only one thread can access a critical subsection of code (where shared data is manipulated) at any given time.
- **Initialization:**
  - Static: `pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;`
  - Dynamic: `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexatt *attr);`
- **Locking and Unlocking:**
  - `pthread_mutex_lock()`: Acquires the mutex. If the mutex is already locked, the calling thread blocks until it becomes available.
  - `pthread_mutex_unlock()`: Releases the mutex.
- **Destruction:** `pthread_mutex_destroy()` (for dynamically initialized mutexes).

2.5 Assignment Tasks

Students should write C programs for each of the following scenarios. Ensure proper error handling and resource cleanup (e.g., deleting IPC objects when no longer needed).

1. **Unnamed Pipe Communication:**
   - Write a C program where a parent process creates a child process using `fork()`.
   - Establish an unnamed pipe between them.
   - The parent writes a message (e.g., "Hello from parent!") to the pipe.
   - The child reads the message from the pipe and prints it to the console.
   - Ensure proper closing of unused pipe ends in both parent and child.
2. **Named Pipe (FIFO) Communication:**
   - Create two separate C programs: a `writer.c` and a `reader.c`.
   - The `writer.c` program should create a named pipe (e.g., `/tmp/my_fifo`) using `mkfifo()`. It then opens this FIFO for writing and continuously reads lines from standard input, writing each line to the FIFO.
   - The `reader.c` program should open the same FIFO for reading and continuously read lines from it, printing each line to standard output.
   - Test by running `writer.c` in one terminal and `reader.c` in another. Ensure the FIFO is cleaned up (deleted) after use.
3. **Shared Memory with Semaphore Synchronization (Producer-Consumer Problem):**
   - Implement a simplified Producer-Consumer problem using shared memory for the buffer and semaphores for synchronization.
   - **Shared Resources:**
     - A shared memory segment to act as a circular buffer (e.g., an array of integers).
     - Two semaphores: `empty` (counts empty slots, initialized to buffer size) and `full` (counts full slots, initialized to 0).
     - A mutex (binary semaphore) to protect access to the buffer itself (ensuring only one process modifies the buffer at a time).
   - **Producer Process:**
     - Continuously produces items (e.g., random integers).
     - Waits on `empty` semaphore, then locks the mutex.
     - Adds an item to the shared buffer.
     - Unlocks the mutex, then signals `full` semaphore.
   - **Consumer Process:**
     - Continuously consumes items.
     - Waits on `full` semaphore, then locks the mutex.
     - Removes an item from the shared buffer.
     - Unlocks the mutex, then signals `empty` semaphore.
   - Create a parent process that forks one producer and one consumer process. The parent should initialize the shared memory and semaphores, then wait for the children to complete (or run for a fixed duration). Ensure all IPC resources are cleaned up before the parent exits.
4. **Multithreaded Summation:**

- Write a C program that calculates the sum of a large array of integers using multiple threads.
- Divide the array into equal parts, with each thread responsible for summing its assigned part.
- Each thread should store its partial sum in a shared variable.
- Use a mutex to protect the shared total sum variable to prevent race conditions during updates.
- The main thread should create the worker threads, wait for them to complete, and then print the final total sum.

5. **Thread-Safe Counter:**
   - Implement a simple global counter variable.
   - Create multiple threads (e.g., 5-10 threads).
   - Each thread should increment the global counter a large number of times (e.g., 100,000 times).
   - First, demonstrate the race condition by running without any synchronization. Observe the incorrect final count.
   - Then, modify the program to use a mutex to protect the counter increment operation, ensuring a correct final count.

Advanced Topics to Explore (Optional)

For students who complete the core tasks and wish to delve deeper into concurrency and synchronization.

2.5.1 Condition Variables

- **Concept:** Used with mutexes to allow threads to wait for a specific condition to become true.
- **System Calls:** `pthread_cond_init()`, `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`.
- **Application:** Re-implement the Producer-Consumer problem using condition variables in addition to mutexes, which is a more idiomatic and efficient way to handle waiting conditions compared to busy-waiting or just semaphores for this pattern.

2.5.2 Reader-Writer Locks

- **Concept:** A synchronization primitive that allows multiple readers to access a shared resource concurrently, but only one writer at a time.
- **System Calls:** `pthread_rwlock_init()`, `pthread_rwlock_rdlock()`, `pthread_rwlock_wrlock()` `pthread_rwlock_unlock()`.
- **Application:** Implement a simple database access simulation where multiple threads can read data concurrently, but only one thread can write data at a time.

2.5.3 Deadlock Detection and Prevention

- **Concept:** Understanding the four necessary conditions for deadlock (Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait).
- **Prevention Strategies:** Explore techniques like resource ordering, breaking hold and wait, etc.

- **Simulation:** Create a small simulation (e.g., using threads and mutexes) that demonstrates a deadlock scenario and then modify it to prevent the deadlock.

2.5.4 Thread-Specific Data

- **Concept:** How threads can have their own private data, even though they share the same address space.
- **System Calls:** `pthread_key_create()`, `pthread_setspecific()`, `pthread_getspecific()`.
- **Application:** Use thread-specific data to manage per-thread error codes or logging information.