OS Practice Assignment 3

---

1 IPC

1.1 Toggling Work with Signals

Write a C program that forks a child process. The child process should start an infinite loop, printing "Working" once per second. The child must install a signal handler for SIGUSR1 that toggles a global volatile sig_atomic_t "paused" flag. When "paused" is true, the child's loop should not print "Working" but should continue to sleep, effectively pausing its work. The parent process should:

1. Sleep for 3 seconds.
2. Send SIGUSR1 to the child (to pause it).
3. Print "Parent: Paused child."
4. Sleep for 3 more seconds (during which the child should be silent).
5. Send SIGUSR1 to the child again (to unpause it).
6. Print "Parent: Unpaused child."
7. Sleep for 2 seconds (to see the child working again).
8. Send SIGTERM to the child to terminate it.
9. Reap the child with wait().

1.2 Process Ping-Pong (Signals)

Write a C program that forks a child process. The parent and child must communicate using two user-defined signals (SIGUSR1 and SIGUSR2).

1. The parent sends SIGUSR1 to the child and then waits.
2. When the child receives SIGUSR1, it prints "Ping!" and sends SIGUSR2 to the parent.
3. When the parent receives SIGUSR2, it prints "Pong!", waits for 1 second (sleep(1)), and then sends SIGUSR1 to the child again. This "Ping-Pong" exchange should happen exactly 5 times, after which both processes should exit.

1.3 Parent-to-Child Data Reversal (Anonymous Pipe)

Write a C program where a parent process forks a child and creates an anonymous pipe (pipe()). The parent process must read a string from the user (e.g., "hello"). The parent will then send this string to the child process using the pipe. The child process must wait to read the string from the pipe, and once it receives it, it should print the string to the console in reverse (e.g., "olleh"). The parent must then wait for the child to exit.

Input: The parent process will expect a single line of text (ending with a newline) from standard input.

2 Concurrency

2.1

Write a C program with a global character array (e.g., char message[100]). Create two threads.

- The first thread's function should write the string 'Hello World' into the message array, character by character.
- The second thread's function should write the string 'Goodbye' into the same message array, character by character. After joining both threads, the main thread should print the final string stored in the message array.

2.2 Producer-Consumer Problem (Bounded Buffer)

Implement a single-slot "bounded buffer" problem using one producer thread and one consumer thread. The threads must use a mutex and two condition variables to synchronize.
Requirements:

1. Define global variables for the shared state: int buffer (The single slot); int item_count = 0; (0 = empty, 1 = full); pthread_mutex_t mutex; pthread_cond_t cond_full; (Signaled when the buffer is full); pthread_cond_t cond_empty; (Signalled when the buffer is empty)
2. Create one producer thread. This thread must: Loop 10 times (to produce items 1 through 10); Inside the loop, lock the mutex; Wait on cond_full while item_count is 1 (i.e., while full); Put the item (e.g., 1, 2, 3...) into the buffer; Set item_count = 1; Signal cond_empty (to wake up the consumer); Unlock the mutex
3. Create one consumer thread. This thread must: Loop 10 times (to consume 10 items); Inside the loop, lock the mutex; Wait on cond_empty while item_count is 0 (i.e., while empty); Read the item from the buffer and print it; Set item_count = 0; Signal cond_full (to wake up the producer); Unlock the mutex
4. The main thread must wait for both the producer and consumer threads to join.

2.3

Write a C program that uses a condition variable to make a "worker" thread wait for a "main" thread to signal that work is ready.

1. Define global variables: pthread_mutex_t mutex; pthread_cond_t cond; int work_ready = 0; (This is the shared condition)
2. Create one worker thread. This thread's function must: Lock the mutex; Wait on the condition variable (pthread_cond_wait) as long as work_ready is 0. (Use a while loop for this check); Once signaled, print "Worker: Work started!"; Unlock the mutex.
3. The main thread must: Print "Main: Sleeping for 2 seconds"; Call sleep(2); Lock the mutex; Set work_ready = 1; Signal the worker thread using pthread_cond_signal; Unlock the mutex.
4. The main thread must pthread_join the worker thread before exiting.

2.4

Write a C program to simulate a resource pool with a limited number of "slots". The pool has 3 available slots, which must be managed by a POSIX semaphore.

1. Declare a global POSIX semaphore: sem_t pool_sem;.
2. In the main function, initialize the semaphore to have a value of 3.
3. Create 5 threads. You should pass a unique ID (e.g., 1, 2, 3, 4, 5) to each thread as an argument.
4. Each thread's function must: Call sem_wait() to acquire a slot; Print "Thread [ID] got a slot."; Simulate work by calling sleep(1); Print "Thread [ID] released a slot."; Call sem_post() to release the slot.
5. The main thread must wait for all 5 threads to join before exiting.

2.5

Write a C program that uses a semaphore to force one thread to wait for another thread to complete a specific task.

1. Define a global sem_t task_A_done; and initialize it to 0.
2. Create two threads: "Thread A" and "Thread B".
3. Thread A's function must: Print "Thread A: Performing task"; sleep(2) to simulate work.; Print "Thread A: Task finished."; Call sem_post(&task_A_done) to signal that it is done.

4. Thread B's function must: Print "Thread B: Waiting for Task A"; Call sem_wait(&task_A_done) to wait for the signal; After sem_wait returns, print "Thread B: Starting its own task."

5. The main thread must join both threads.