uc3m

OPERATING SYSTEMS

Laboratory 3. Multi-thread Programming

100495956 Juan Sanchez Encinas
100495810 Alberto Alvarez Alcazar
100495746 Raul Armas Seriña

Table of contents

Description of the code

Factory Manager

The **belt_wrapper** function serves as a thread entry point that wraps the execution of the process_manager function. Its purpose is to provide controlled, concurrent execution of belt processing tasks. Each thread waits on a semaphore before proceeding, which ensures that the number of active process_manager instances never exceeds the configured limit. After acquiring the semaphore, the function calls process_manager with the belt's ID, size, and item count. If this function returns a non-zero value, indicating an error, a descriptive error message is printed to standard error. Once the function completes, the semaphore is released, and the dynamically allocated memory for the thread's arguments is freed to prevent memory leaks. The thread then exits gracefully.

The **main** function orchestrates the setup and execution of all belt processing tasks. It begins by validating that exactly one command-line argument is provided, which should be the path to a configuration file. If the argument is missing or incorrect, an error is reported and execution stops. It then attempts to open the specified file; if unsuccessful, the program exits with an error. The first line of the file is expected to contain an integer specifying the maximum number of concurrently running belts. If this value is invalid or non-positive, the program again terminates with an appropriate error message.

Subsequently, the program enters a loop that reads integers from the configuration file into an array. Each belt configuration is expected to be defined by a trio of integers: a unique ID, the size of the belt, and the number of items to produce. After reading, the program checks that the number of integers is divisible by three; if not, the file is deemed malformed and execution stops. It also verifies that the total number of belt configurations does not exceed the maximum number of concurrent tasks. Another loop then iterates over the triplets, creating an array of belt_info structures while validating that each field is a positive integer.

Once all belt configurations are validated and stored, a semaphore is initialized to control the number of concurrent threads, based on the earlier configuration. A fixed-size array of pthread_t is created to hold thread identifiers. The next loop iterates over all belt configurations, dynamically allocates a th_args_fm structure for each, sets its fields (semaphore and belt data), and creates a new thread using pthread_create, passing in the belt_wrapper as the start routine. Success messages are printed for each thread creation. After all threads have been created, the final loop waits for each to complete using pthread_join, printing confirmation once each belt process finishes. Finally, allocated memory is freed, the semaphore is destroyed, and the program exits cleanly with a completion message.

Process Manager

The **Produce** function is executed by the producer thread and is responsible for simulating the production of items onto a shared queue. It receives its arguments through a th_args_pm structure, which includes information such as the belt ID, number of items to produce, and a pointer to the queue. Inside a loop that runs exactly to_produce times, the function creates an element for each item, populating fields like the edition number and marking the last item accordingly. Each element is then inserted into the queue using the queue_put function. Once production is complete, the dynamically allocated argument structure is freed, and the thread exits cleanly.

The **Consume** function is the counterpart to Produce and runs on the consumer thread. Like the producer, it receives arguments via a th_args_pm structure and uses the to_produce field to determine how many items to consume. In its loop, it retrieves elements from the queue using queue_get, which simulates the consumption of items from the conveyor belt. Each consumed element is freed to avoid memory leaks. After consuming the expected number of items, the function also frees its argument structure and exits the thread.

The **process_manager** function acts as the core controller for a simulated production line, coordinating both production and consumption via threading and queue operations. It starts by validating its input parameters—belt ID, belt size, and number of items to produce—ensuring they are all positive integers. If validation fails, it prints an error message and exits early. The function then initializes a queue structure with the specified belt size, which acts as a bounded buffer between the producer and consumer. Failure to initialize the queue results in an error and premature exit.

If the queue is initialized successfully, the function prints confirmation messages and proceeds to allocate memory for two argument structures: one for the producer and one for the consumer. These structures are populated with relevant details including the belt ID, item count, and queue pointer. If memory allocation fails at any point, it reports the error and performs appropriate cleanup.

The function then spawns two threads: one running the Produce function and the other running Consume. It checks for thread creation errors and exits with an error code if either fails. If both threads are successfully created, the main function waits for them to complete using pthread_join. Once both threads have terminated, it prints a success message indicating how many items were processed and then destroys the queue to free resources. Finally, it returns 0 to indicate successful completion.

Queue

Queue init initializes a queue structure by allocating memory for the queue elements, setting up initial positions for head, tail, and count, and initializing synchronization primitives (a mutex and two condition variables). The mutex ensures mutual exclusion when multiple threads access the queue, while the condition variables allow threads to wait for or signal changes in the queue's state (e.g., not full or not empty). The function returns an error code if memory allocation fails, making it robust for concurrent applications.

Queue put allows a producer thread to insert an element into the queue. It begins by acquiring the mutex to ensure exclusive access. Inside a loop, it waits (using pthread_cond_wait) if the queue is currently full, ensuring the producer doesn't overflow the queue. When space becomes available, it copies the new element into the queue at the tail position, increments the count, and advances the tail pointer in a circular manner. It then signals any waiting consumer threads that the queue is no longer empty. A log message is printed for debugging or monitoring purposes before releasing the mutex and returning success.

Queue get allows a consumer thread to retrieve an element from the queue. Like the producer, it locks the mutex to ensure thread safety. If the queue is empty, the consumer waits on the not_empty condition variable until an item becomes available. Once available, it

allocates memory for the element to be returned and copies the element from the queue's head position. The head pointer and the count are updated accordingly, using circular indexing. It signals any waiting producer that the queue is not full anymore, logs the operation, and then returns the newly consumed element. If memory allocation fails, it logs an error and returns NULL.

Queue empty and **queue full** provide a quick way to check if the queue is empty or full based on the count variable. They are used in the producer and consumer functions to determine whether to wait or proceed.

Queue destroy deallocates the memory used by the queue and destroys the synchronization primitives. It's crucial for avoiding memory leaks and ensuring clean shutdowns of applications using the queue.

Set of Tests

Description	Input	Output	Expected result
	9 2 3 4 5 6 7 8 9 1 7 5 6	Behaves as intended.	
	4 5 5 2 1 2 3 3 5 2	Behaves as intended.	
	4 5 10 2 1 2 3 13 5 2	Behaves as intended.	
Base case	4 1 3 5 2 4 10 3 8 2	Behaves as intended.	
	5 10 20 50 20 15 10	Behaves as intended.	
	1 100 1 1000	Behaves as intended.	The program should support 1000 item queues
	6 1 1 1 2 1 1 3 1 1 4 1 1 5 1 1 6 1 1	Behaves as intended.	The program should manage inputs of greater length than usual

Conclusions

Overall this laboratory was less problematic than previous assignments. Nonetheless, the process manager had to be revised due to inconsistent outputs. This was however quickly fixed after redesigning the structure and how Produce and Consume interacted with it.

We believe this assignment provided an enjoyable challenge as it was easy to understand the applications of the code and better discern how functionalities should be implemented.