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** provides a controlled and 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 a controlling semaphore is set, the belt's ID, size, and item count are passed to a process_manager call. Error handling is implemented for the case the call fails. Lastly the semaphore is released.

The **main** function orchestrates the setup and execution of all belt processing tasks. Firstly, error handling is implemented to check that a single, valid file path is provided. The first line of the file is expected to contain a valid positive integer specifying the maximum number of concurrently running belts.

Secondly, 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.

Next, a semaphore is initialized to control the number of concurrent threads. A fixed-size array of pthread_t is created to hold thread identifiers. The next loop assigns a th_args_fm structure, sets its fields (semaphore and belt data), and creates a new thread for each thread, 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. Finally, allocated memory is freed and the semaphore is destroyed.

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 (belt ID, number of items to produce, and pointer to the queue). A "to_produce" number of loops are run, creating 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, structure is freed and thread is exited.

The **Consume** function is the counterpart to Produce and runs on the consumer thread. Like the producer, it receives arguments from th_args_pm structure and consumes a "to_produce" numbr of items, using queue_get. 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 checking for valid positive parameters (belt ID, belt size, and number of items to produce). 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 the belt ID, item count, and queue pointer. Error handling is implemented for correct memory allocation and cleans up in case of an error raising.

The function then spawns two threads: one running the Produce function and the other running Consume, checking a correct initialization of both. If both threads are successfully created, the main function waits using pthread_join to then print a how many items were processed and then destroys the queue to free resources.

Queue

Queue init initializes a queue structure (designed and implemented in the queue.c file) by allocating memory, setting up initial positions for head, tail, and count, sets up a mutex to handle multiple threads accessing the queue, as well as condition variables to signal changes in the queue's state (not_empty and not_full). Error handling is inplemented.

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, **queue full** are used in the producer and consumer functions to determine whether to wait or proceed by returning count values reflecting that state. **Queue destroy** deallocates the memory and destroys mutex and condition variables before ending execution.

Set of Tests

Description	Input	Output	Expected result
	9 2 3 4 5 6 7 8 9 1 7 5 6	Behaves as intended.	
Non divisibles by three	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.	
Large queue	1 100 1 1000	Behaves as intended.	The program should support 1000 item queues
Long queue	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.