Report OS Lab 2: Designing a multithreaded runtime environment for parallel computing

Student: PHAN MANH TUNG

Stage 1:

Concurrency Issues:

1. Task Queue Handling

Description: The task queue is a shared resource among multiple threads, leading to potential race conditions when threads enqueue or dequeue tasks simultaneously.

Solution: Implemented mutual exclusion using mutex locks around critical sections in the enqueue_task and dequeue_task functions. Mutexes ensure exclusive access to the queue to prevent multiple threads from modifying it concurrently.

2. Active Threads Count

Description: Multiple threads decrement the active thread count simultaneously, leading to incorrect signaling for program termination.

Solution: Used a mutex lock to safely decrement the active_threads_count variable in the thread_start_routine. Added signaling with conditions to ensure proper termination signaling in task_waitall.

3. Full/Empty Queue Management with Condition Variables

Description: When the queue is full or empty, it immediately signifies error and returns NULL.

Solution: Implemented conditional waits and signals (pthread_cond_wait and pthread_cond_signal) around enqueue and dequeue operations to ensure correct signaling when the queue is not full or empty, respectively.

Successfully passed tests:

1/simple_test.c
2/pi.c (~0.17s execution time with 10 threads)

Answer to questions:

1. When should the threads be created?

Threads should be created during the initialization phase of the program or system. Specifically, in this context, threads are created after initializing the system state and creating task queues. The function create_thread_pool() creates a pool of threads that will execute tasks from the queue.

2. What are the different steps executed by a worker thread?

Worker threads execute a loop (thread_start_routine) where they continuously check for tasks in the queue. Key steps include:
Checking for available tasks in the queue (is_tasks_queue_empty_wrapper).
Acquiring and releasing mutex locks to manage shared resources.
Retrieving tasks from the queue (get_task_to_execute).
Executing tasks (exec_task).
Handling task completion (terminate_task) or marking as waiting if dependencies exist.

3. To which famous synchronization problem corresponds the problem to be solved for the shared queue?

The shared queue's problem corresponds to the Producer-Consumer synchronization problem. Threads acting as consumers (thread_start_routine) consume tasks from the queue, and other parts of the system (producer threads) generate tasks and add them to the queue (enqueue_task). The challenge is to ensure that the consumer threads wait when the queue is empty and are signaled when new tasks are available (Producer-Consumer synchronization).

4. What should happen if the queue gets full?

If the queue gets full, the enqueue_task function waits on Condition Variable queue_not_full_cv and release the queue mutex. When the dequeue_task is executed, it signals the thread to continue producing tasks.

5. At what moment can the task_waitall() function return?

The task_waitall() function returns when two conditions are met:
All worker threads (sys_state.active_threads_count) have completed their tasks.
The task queue is empty (is_tasks_queue_empty_wrapper() returns true). It implies that all tasks have been executed.

Stage 2:

I did not complete the work for this stage. Here are several steps that I tried to modify as follows:

Step 1: Implement Task Queue Resizing (implement a different enqueue_task)

Modify the tasks queue implementation to support dynamic resizing.
Detect when the queue is full and needs resizing.
Use realloc() to allocate memory for a larger queue and memcpy() to transfer existing tasks to the new queue.
Update queue-related variables and indices accordingly.

Step 2: Enhance Task Status Handling (modify task_check_runnable)

Ensure that the "WAITING" task status is appropriately handled during dependency resolution. Implement logic to change task status to "WAITING" when it has dependencies.

Step 3: Handle Task Routine Return Cases (modify exec_task)

Implement logic to handle different return values from the task routine:
Mark task as terminated when returning TASK_COMPLETED with no dependencies.
Check and dispatch waiting parent tasks when dependencies are fulfilled.

Questions to answer:

1. Explain why the task queue resizing operation is necessary. Also briefly describe how this operation must be performed.

Task queue resizing is necessary to accommodate additional tasks when the current queue reaches its maximum capacity, preventing loss of tasks due to queue overflow. Resizing the queue involves dynamically allocating a larger memory space and copying existing tasks to the new space. This is done using realloc() to allocate a new block of memory and memcpy() to copy existing tasks into the new memory block.

2. What does the WAITING task status correspond to?

The WAITING task status indicates that the task is waiting for its dependencies to be fulfilled before it can be executed. When a task has dependencies that haven't been completed, it is in the WAITING state.

3. What happens after the task routine for a task returns?

After a task routine returns, the following four cases are possible, depending on the returned value and the task's dependencies:
Task returns TASK_COMPLETED and has no dependencies: Marks the task as terminated.
Task returns TASK_COMPLETED and has dependencies: Checks if the waiting parent task can become runnable and dispatches it accordingly.
Task returns TASK_TO_BE_RESUMED and has no dependencies: Marks the task as ready and dispatches it for execution later.
Task returns TASK_TO_BE_RESUMED and has dependencies: Initiates the necessary actions to handle the waiting parent task similar to the TASK_COMPLETED case.

Passed tests: None