Lab 2 Design Document

Group 2
Aaron Meurer
Oran Wallace
Sheng Lundquist

Introduction

This lab is used to design and implement a process manager. This process manager will be used further on in our operating system design. A process manager is important to schedule processes to allow for multiple processes on limited hardware.

Our process manager will be abstracted into three different components:

Test Implementation
Process Manager
Queue Manager

There will also be a test generator file to help us generate files that have commands for the test implementation to read.

For security reasons, the process control block, the queue_t structure, and the queue_enum structure will be placed in a header file (the old test implementation included a .c file instead of a .h file). The queue_t structure will define the queue being specified in the queue manager's calls. This structure will contain the head and tail pointers of a certain queue, the pointer to the memory stack used by enqueue, and the size of the array. This structure was used to easily implement the current queue manager code to support multiple queues.

Test implementation

The test implementation will call functions in the process manager and interpret error codes provided by the process manager. The main function that the test implementation will handle that is not implemented in the process manager is the list command.

Process Manager

The QUEUES enumerated type will define naming conventions used in the process manager. This type will help in naming parameters for a move function that will dequeue from a certain queue and enqueue into another queue. This function will be called by the go, eoquantem, eolife, and wait commands. These commands will do the following transitions:

- go - from ready to running
- eoquantem - from running to ready
- eolife - from running to terminated
- wait - from running to waiting

The unwait command will not use the move function, since it needs to search through the waiting queue. This will be done through the delete function in the queue manager. This process will be enqueued into the ready queue.

The create command in the process manager will use the enqueue function on the new queue. The move function will be called to move the created process from the new queue into the ready queue (this step is redundant and was done to satisfy the test implementation requirements). The create function will be responsible for making sure the number of processes does not exceed the maximum amount of processes allowed in the process manager (we will use 20, but this can easily be modified by changing MAX_PROCESSES). The function will return error codes for two cases: (1)The maximum number of processes has been reached, or (2) the pid already exists. To find if the pid exists, the queue manager's find_process function will be called on all queues.

Queue Manager

A few changes were made to the old queue manager. enqueue, dequeue, delete, find_process, and find_empty now take the queue_t structure to specify which queue is being modified or searched. A get_process function is provided to easily change the enum type to the required queue_t structure used in differentiating between queues by the queue manager. init goes through each queue that exists and clears each element. dequeue and delete both return a process control block instead of a pid.

State Diagram Description

Waiting

Our state diagram begins with the Process Manager starting and then immediately performing an epsilon transition into the "Waiting" state. In this state the Process Manager can transition to 5 different states depending on the command input from the Test Interface.

Creating

If the create command is executed the Manager will transition into the "Creating" state. In this state, the Manager will make sure the pid is unique and the created process does not exceed the maximum allowed number of processes. If this does happen, the Manager will instantly transition to the "Error" state and return the correct error output. From the "Creating" state the Manager will directly epsilon transition into the the "Moving" state.

Moving

The "Moving" state will be used for the following commands: eoquantem, eolife, wait, or go. This state will transition into the "Error" state depending on the command. All four commands will transition into the "Error" state if the running queue is empty. Additionally, go will also move into the "Error" state if the ready queue is empty. If there are no errors, the manager will epsilon transition back into the "Waiting" state. Since the "Create" state will take care of the maximum number of processes, full queues will not have to be taken care of in the "Move" state.

Searching

If the unwait command is inputted the Manager will transition into the "Searching" state. If the pid specified is not in the Waiting queue the Manager will transition into the "Error" state. If no error is encountered the Manager will epsilon transition to the "Waiting" state.

Listing

If the list command is entered the Manager will transition into the “Listing” state. If a queue is specified that does not exist the Manager will transition into the “Error” state. If no error is encountered the Manager will epsilon transition back to the “Waiting” state.

Possible Process States (Queues):

- **New** - In this states the process is in the process of being created along with the PCB and placed in the New Queue, after it has been created it is placed in the Ready Queue. The size of this queue is one for now, since it doesn’t need to be bigger than that for this implementation.
- **Ready** - In this state the process has already been created and is in need of the CPU to execute. From this state the process will be placed in the Running Queue after a GO command has been issued. The size of this queue is 20, since potentially all processes could be in this state.
- **Running** - In this state a process will be handled by the scheduler and be placed into the Running Queue. From this state the process can either be placed into the Waiting Queue or the Terminated Queue. The size of this queue is one, since we are assuming only a single core CPU, so only one process can be running at once.
- **Waiting** - In this state a process is placed into the Waiting Queue. From this state the process can only be moved back to the Ready Queue. The size of this queue is 20, since potentially all processes could be in this state.
- **Terminated** - In this state a process has been removed from the Running Queue and placed into the Terminated Queue. The size of this queue is 20, since potentially all processes could be in this state.

Process Control Block (PCB)

- pid, psw, page_table, regs[NUM_REGS] - all are required as per Lab 1.
- int empty - flag used to tell if the current index in the struct array is empty.
- struct process_control_block *next - pointer pointing to the next process in the queue.
- struct process_control_block *prev - pointer pointing to the previous process in the queue.

Test Cases

There are a few test cases we did for each function we wrote in the process manager.

CREATE

- Check if putting in a non-unique throws the correct error
- Check if process manager limits number of process allowed

EOQUANTEM, WAIT, EOLIFE

- Check if running queue is empty, throws the correct error

UNWAIT

- Check if waiting queue is empty, throws the correct error
- Check if the pid specified doesn’t exist in the waiting queue, throws the correct error

GO

- Check if running queue is full (something already running), throws the correct error
- Check if ready queue is empty, throws the correct error

LIST

- Check if list sched of an empty ready queue throws an error