ARCHITECTURAL DOCUMENT Z502 - PHASE I

Some abbreviation:

PCB	Process Control Block
OS	Operating System
Prio	Priority
pid	Process ID

A. A list of what include in my design:

PCB_Current_Running_Process keeps track PCB of current running process.

TimerQueue contains PCB of processes in WAITING states because of SLEEP system call.

ReadyQueue With Priority and ReadyQueue Without Priority.

SuspendList is a Linked List to stores processes' PCBs because of SUSPEND_PROCESS and RESUME_PROCESS system call.

PCB_Table: global array to store all the processes' PCBs from start to finish the simulator.

MessageSuspendList to store processes' PCBs if the processes is waiting for a message because of SEND_MESSAGE and RECEIVE_MESSAGE system call

Message_Table is global message list to store all the messages among processes in the system. Its role likes a middle man.

In my OS design, I use **locks** whenever OS takes out or pushes a PCB into TimerQueue, ReadyQueue or SuspendList.

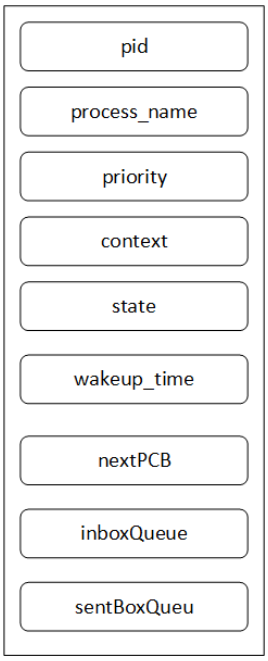
Tests I completed:

test0
test1a
test1b
test1c
test1d
test1e
test1f
test1g
test1h
test1i
test1j
test1k
test1l - extra features. OS runs this test from start to finish successfully.

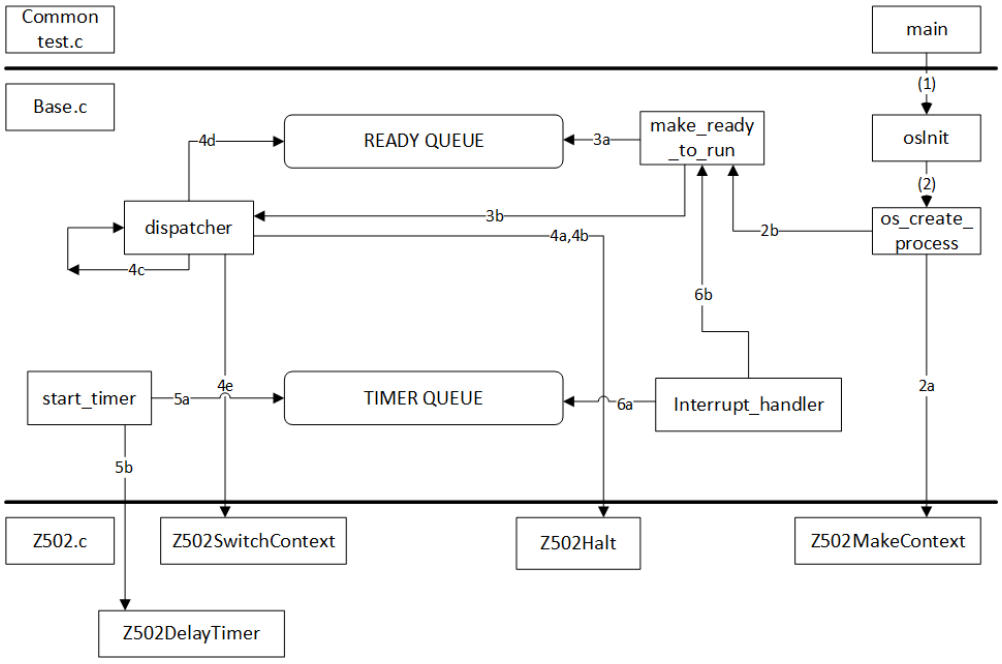
Please notice that in test1i, test1j and test1l, OS uses MessageSuspendList which is not be printed out just because I am not sure state = SWAPPED in state_printer can be used for phase 1.

B. High Level Design and C. Justification of High Level Design

This is the structure of PCB



Before going detail each design detail, I want to point out the ***common procedures*** (from step (1) to step(7) for every test:



1.The simulator starts with main. Main call to OSInit.

2. OsInit makes a call to os_create_process. In os_create_process, I create PCB for a new request create process after validating that it is legal request of creating process. Then I call MakeContext (2a) for the new process. Notice that: I do not SwitchContext immediately there. Finally, I call make_ready_to_run (2b).

3. In make_ready_to_run, I put the PCB into ReadyQueue based on the ReadyQueue is ordered by Priority or not. And then I call to dispatcher()

4. In dispatcher(), first of all I check if ReadyQueue is empty then I check two following cases:

(4a). If TimerQueue is empty but SuspendList is not empty then OS knows that some processes will never be resumed so OS stops the simulator by calling Z502Halt. This applies to test1e.

(4b) If TimerQueue is empty and SuspendList is empty then nothing actually runs yet. It happens with test0 and test1a then I just call Z502Halt.

(4c) If (4a) and (4b) is not satisfied then OS calls CALL() to advance Clock and stay on the loop.

After exit the loop then I take out the PCB in front of ReadyQueue and SwitchContext there (4d) and (4e) to start running new process.

5. When a process go to sleep by calling SLEEP system call, OS calls to start_timer. start_timer will push process's PCB into TimerQueue first (5a) and then set the Z502TimerStart letting interrupt happens (5b). Some notice is that: the PCB inserted into TimerQueue is ordered by its wake up time. And whenever change happens in front for TimerQueue then OS will update the Z502TimerStart to the relative time of new PCB at front.

6. When interrupt happens, OS calls to Interrupt_handler. This routine will pull out PCBs at front of TimerQueue (6a) if wakeup time less or equal than current time. Then OS pushes those PCBs into ReadyQueue by invoking make_ready_to_run (6b). There is a change in front of the TimerQueue so OS also updates Z502TimerStart to the relative time of new PCB at front.

7. TERMINATE system invokes terminate_proccess_id routine. This routine first check the legal parameter of system call. If all parameters are valid, then the routine checks:

First it checks there is a legal process passed into the function. Then it checks each TimerQueue or ReadyQueue. If the process needed to be terminated is in

(a) ReadyQueue: then just take out from the ReadyQueue (13b)

(b) TimerQueue: take out from the TimerQueue with comparision between time at front of TimerQueue and current time. (13c)

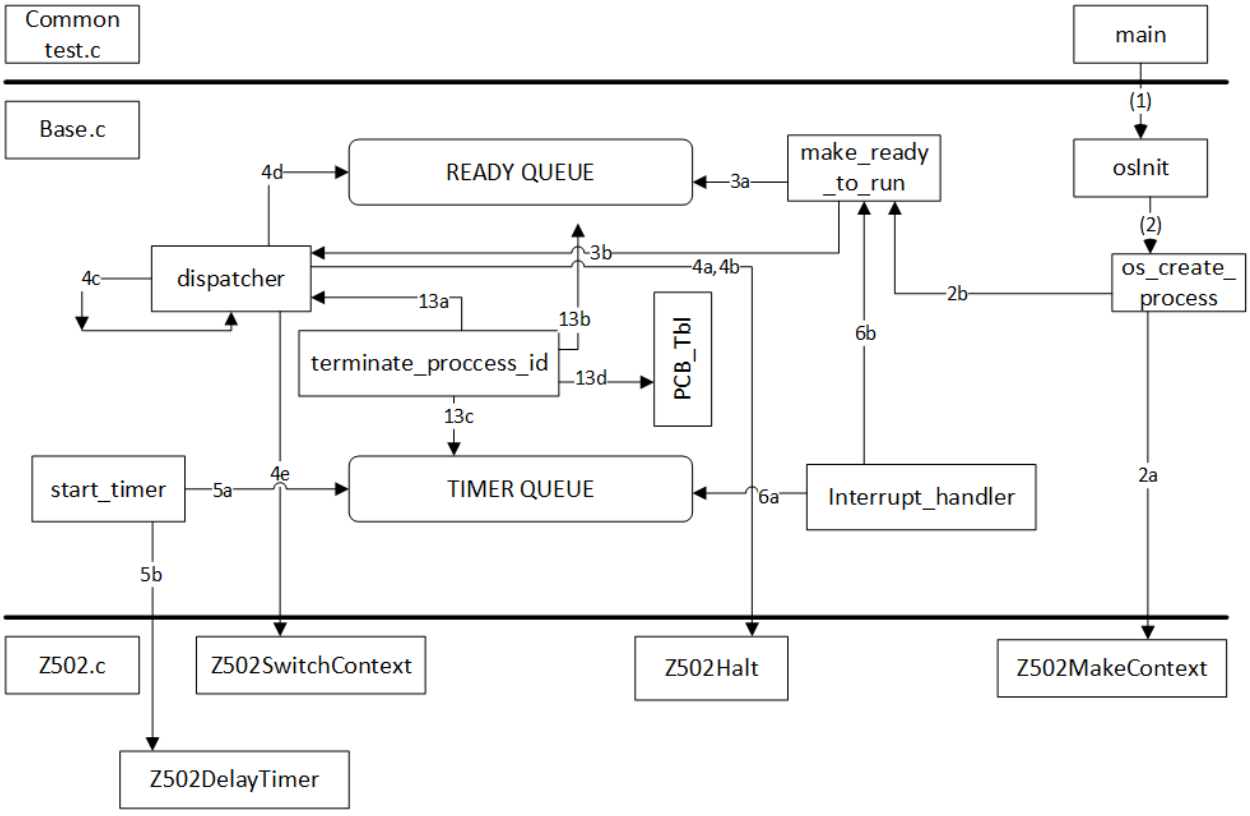
There are two cases to be considered:

(b1) If current time does not pass wake up time of PCB at front of TimerQueue then update SetTimer by different between those time. This update happens even head PCB of TimerQueue is changed or not. If change order happens at header TimerQueue then update SetTimer is needed. But even header does not change, I still update time in SetTimer because interrupt time is still correct but I can reduce the code.

(b2) If current time already passed, then generate interrupt immediately.

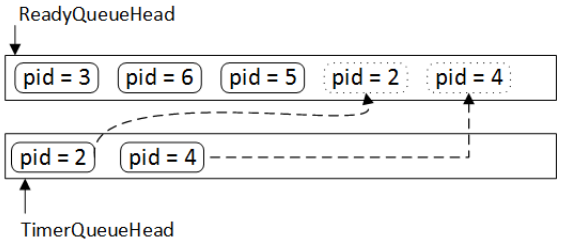
Finally, set Terminate state in PCB_Table so that this process never be scheduled again (13d).

After finishing this routine, program returns to SVC which invokes dispatcher() (13a) to let other processes run.

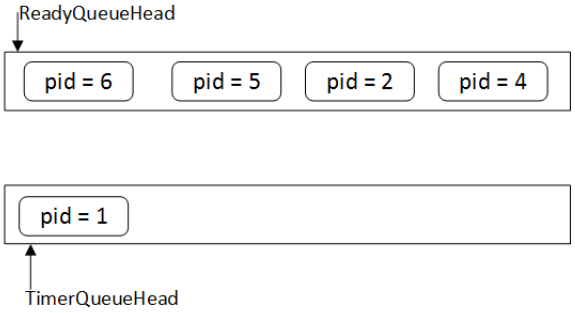


8. test1b and test1c:

For test1b, os_create_process will check is this a legal process or not. If all parameters of system call CREATE_PROCESS is valid then test1c runs all the steps in common part above. In here, I just want to demonstrate how PCB go out and in ReadyQueue because structure of the OS is the same as common part above.

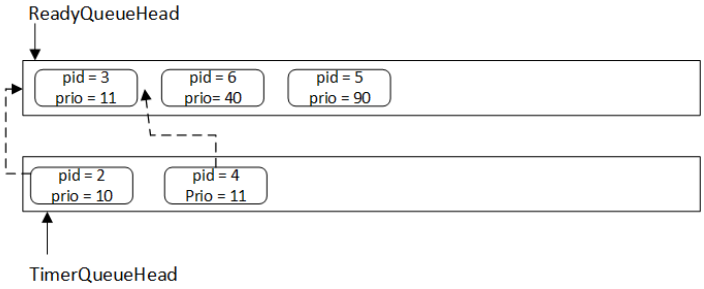


Pid = 2 and Pid=4 inserted at the end of ReadyQueue.
Pid=1 goes to sleep and Pid=3 is running.

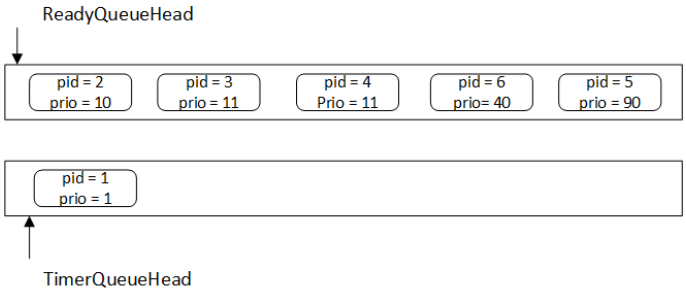


8. test1d

test1d, PCB will be inserted into ReadyQueue ordered by its priority. Structure of OS is the same as test1c and common parts list above.

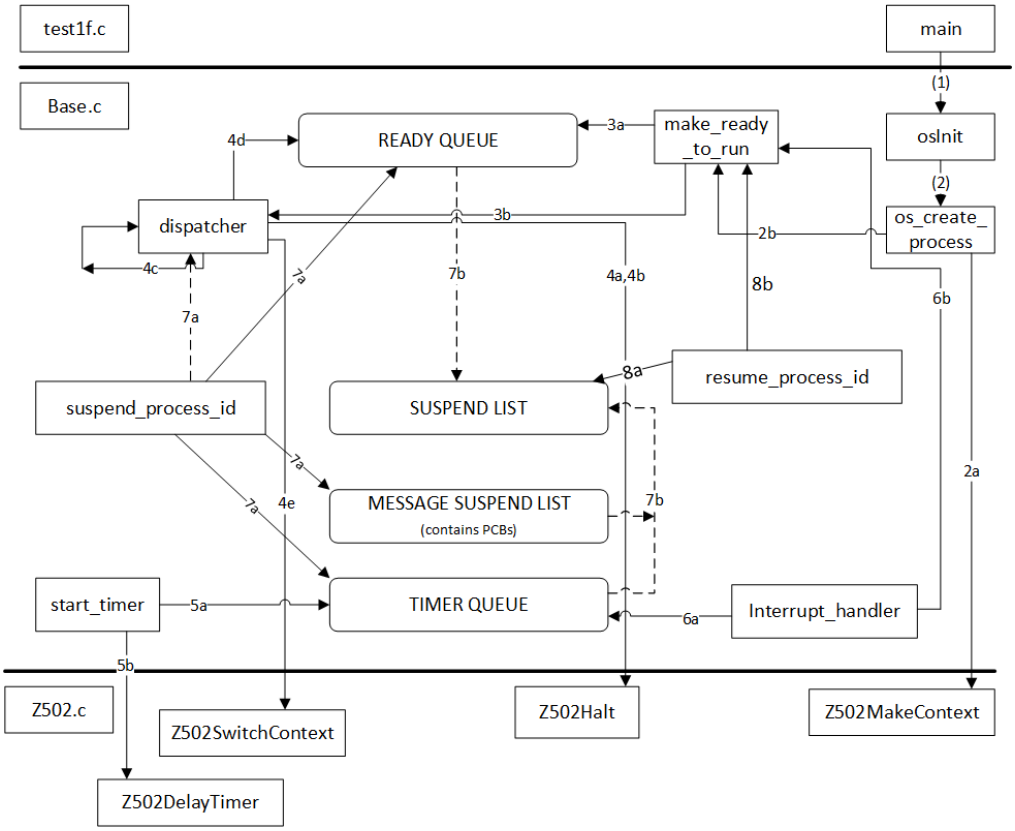


Pid = 2 with highest Priority inserted in front ReadyQueue
Pid=4 inserted into after Pid=3



9. test1e and test1f

In those test, OS uses SuspendList to manage suspend and resume process. SuspendList is a LinkedList structure. Because in test1i, test1j and test1l, OS uses another suspend list is SuspendListMessage so I mention in advance SuspendListMessage.



After invoking a system call SUSPEND_PROCESS, OS invokes `suspend_process_id`. In this routine, OS uses `legal_suspend_process` to check whether a process requested to suspend is legal or not. If the process to be suspended is legal, there are some cases:

(1). If the process suspend is the current process running (or Suspend Self) then OS just push its PCB into SuspendList and then calls dispatcher() to let other to run. The dash line from suspend_process_id to dispatcher() expresses idea of Suspend Self by invoking dispatcher to let another process to run.

(2). If the process is in TimerQueue, ReadyQueue or SuspendListMessage, OS just takes it out of those queue/list at Step 7a. And then OS pushes process's PCB into SuspendList at step 7b. Step 7b I mark as dash line because PCB is taken from one of the queue/list. If OS takes out in front of TimerQueue then updates Timer by using the wakeup time of a new front.

RESUME_PROCESS system calls will force OS to resume a process from SuspendList at step 8a. And then OS pushes back into ReadyQueue by calling my_ready_to_run at step 8b in the figure above.

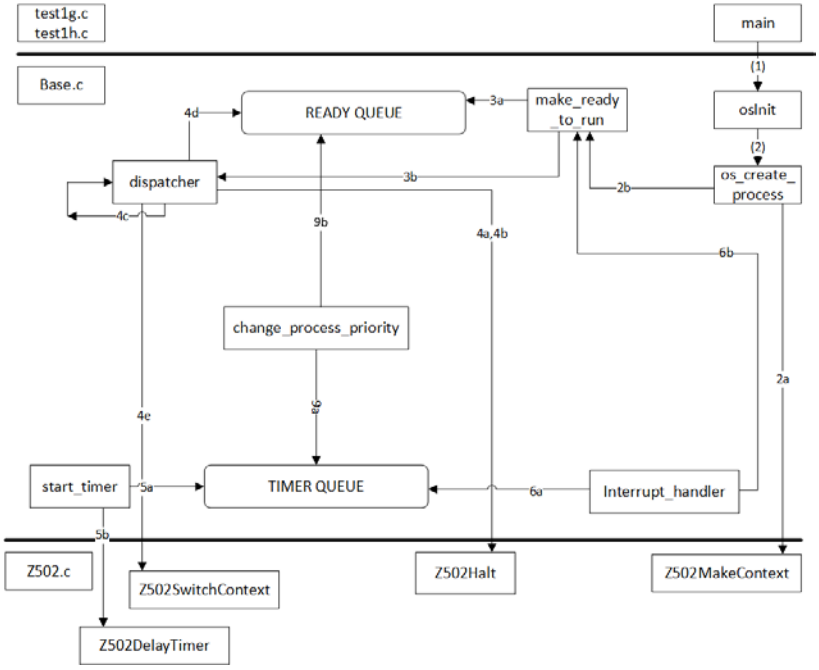
10. test1g and test1h

In those tests, OS responds to CHANGE_PRIORITY system call by invoking change_process_priority. First of all, this routine checks the legal of system call parameters by making a call to check_legal_change_process_priority. If all parameters are legal then change_process_priority continues to process with some cases:

(1). If the current running process wants to change its Priority then OS just changes its Priority to new value.

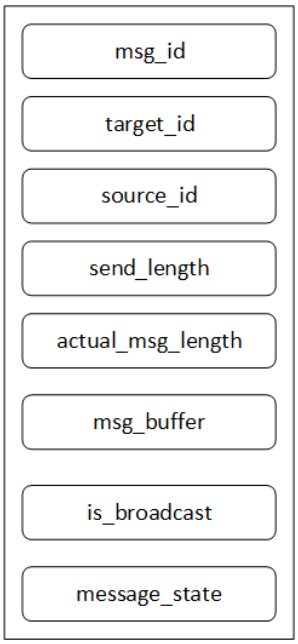
(2). If the change Priority process is in TimerQueue then OS just finds this process in TimerQueue. And then OS assigns a new value to its Priority (9a in figure below)

(3) . If the change Priority process is in ReadyQueue then OS first pulls out process's PCB. And then updates its Priority. And finally, OS pushes it back into ReadyQueue. (9b in figure below).



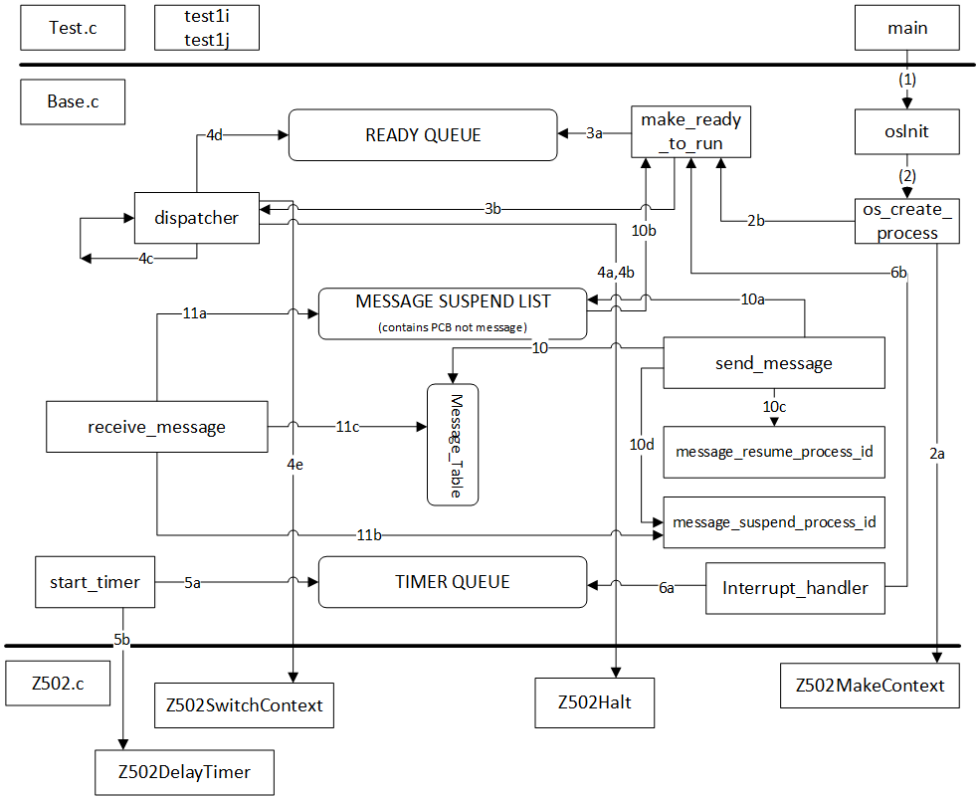
11. test1i and test1j

Message structure:



In those tests, OS responds to SEND_MESSAGE system call by calling send_message routine and RECEIVE_MESSAGE system call by invoking receive_message routine.

Moreover, OS uses another suspend list called MessageSuspendList to store PCBs of processes that are temporary suspended because of waiting for a response message. OS also uses a global array of sending and receiving messages called Message_Table.



In send_message routine, OS first check all parameters of system call SEND_MESSAGE is legal or not by invoking message_send_legal routine. If all parameters are valid then OS pushes new message into process's sentBoxQueue and global Message_Table (Step 10). And there are some cases the routine needs to consider:

(1). If sending message is a broadcast message then anyone can receive the message. OS pulls all the processes are in SuspendListMessage (step 10a) to ReadyQueue by calling make_ready_to_run (step 10b) and ordered by their Priority.

(2). If sending message is going to a specific target process, then the current running process will resume target process in MessageSuspendMessage by invoking message_resume_process_id (step 10c). This routine is similar to resume_process_id, which invoked by RESUME_PROCESS, but it works with SuspendMessageList instead of SuspendMessage. OS suspends current running process by inserting it into MessageSuspendMessage to wait for the respond (step 10d).

In receive_message routine, OS first check all parameters of system call RECEIVE_MESSAGE is legal or not by invoking message_receive_legal routine. If all parameters are valid then the routine continues to run with considering some cases:

(1). If the message is broadcast, then the current running process first checks its inbox to verify that if its inbox has this message or not.

(a) If it has, then it let another process to receive message by resuming waiting message process from MessageSuspendList to ReadyQueue by calling make_ready_to_run (Step 11a). Then current running process suspends itself to MessageSuspendList by invoking message_suspend_process_id (Step 11b).

(b) If the current running process has not then it inserts the message from global Message_Tbl by making a copy to its inboxQueue and marks the message in Message_Tbl as RECEIVED (Step 11c). The status informs processes that this message already received by another process so do NOT receive it.

(2) If a receiving message is sent from specified process (A) then current running process (B) suspends itself by invoking message_suspend_process_id (Step 11b) to let process (A) returns and continues its receiving.

C. More Justification of High Level Design

PCB_Table: at first I implement PCB_Table as LinkedList structure. Then when the test1c and 1d come in, and the actions of pull out and push process in Queue, List make LinkedList is so hard to maintain. So I convert PCB_Table into array.

State in Message: at first glance, I implement the OS that allows a broadcast message has to receive by everyone or every processes that active in the system. This requires OS have to store a broadcast message in a global message queue. OS never deletes messages in global message queue. That allows every process can scan the global message queue and **copy** of the message into its inbox. Then in class, Professor said a broadcast message should be received by only one process so I insert a state in a message so that if a message received by one then never received by others.

And then because my OS never deletes messages in global message queue, my OS can RESTORE messages a process. I think I can do RESTORE messages for a process, that will be fun but the deadline does not allow me to implement this functionality.

D. Additional Features

test1l

With test1l I do not insert any new structure from test1i and 1j. I just let the test runs and did some debug then test1l can run from start to finish without showing any errors.