Project 1 Architectural Document

Yuteng Zhou (ID:827819800)
Department of Electrical and Computer Engineering, Worcester Polytechnic Institute

I. A List of What is Included in My Design

In this project, all the tests from test0 to test11 has been successfully tested. In my design, besides files provided by professor, I have added three header files and three C source files into the design. They are queue_function.h, queue_function.c, message_function.h, message_function.c, yuteng.h, yuteng.c. Below is a list of what's included in these files, more details can be seen in my code.

First of all, to give a brief impression of my file system. The file association is plotted in figure 1.

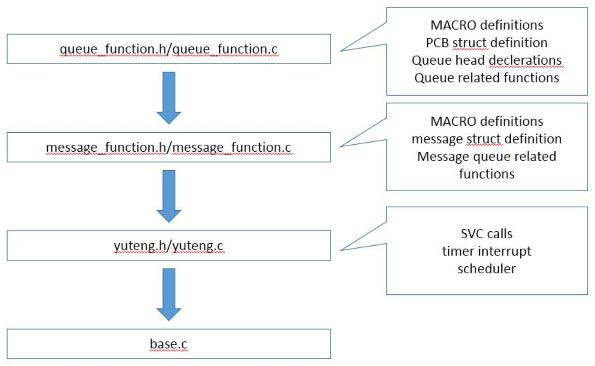

Figure 1. Files Overview

A. queue_function.h

Basically, it defines some MACROs, the PCB struct, three queues (timer queue, ready queue, suspend queue), and function declarations related to queue operation. PCB is the crux of my design, it contains pointer to next PCB, so in my system, no matter what kind of queues I am working on, the queue item's type is PCB, there is no need to make another struct to do queue operations. The struct PCB also contains pointer to context which is a necessity to keep record of the context which is running on hardware. Besides, the PCB has items such as process ID, priority, sleep time, absolute wake up time of the current PCB, and its process name. PCB also keeps records of current state, it can be in timer queue, in ready queue, or in suspend queue, and there is another special state of the PCB, it can be in suspend queue with the state waiting for message in suspend queue.

B. queue_function.c

In this file, all the queue functions are to be realized. These queue methods are add one PCB to the queue, remove one PCB from the queue, remove one PCB from the queue by its process ID, check whether target PCB exists in the queue, print out all the queue items for debugging purpose and so on.

C. message_function.h

In this header file, it defines the structure of message struct as well as message related functions. The struct message contains message ID, target process ID, source process ID, message sending length, message receive length, one message buffer which is defined as a char array with 512 bytes maximum capacity. Of course, the struct message has the pointer to struct message.

D. message_function.c

This file realizes all the message related methods, such as put one message into message queue, take out one message from message queue either by its target process ID or by both source and target process IDs.

E. yuteng.h

This header file contains operating system related methods declarations, it includes all the SVC calls like sleep, get time of the day, terminate process, get process ID, suspend process, resume process, change priority. Also, it contains restart timer function which manipulates the hardware timer.

F. yuteng.c

This file realizes all the functions defined in yuteng.h. All the SVC calls defined in yuteng.h, as well as the scheduler, which will take out the PCB with highest priority in ready queue and switch context to newly updated PCB's context. Besides, timer interrupt is also coded here.

II. High Level Design

Since my operating system is expected to respond to various kinds of system service calls, the core of my design is to writes codes in different SVC calls to interact with user methods. The high level design part mainly shows how these SVC routines are implemented. Besides, the routine of the scheduler and the routine of the timer interrupt is presented.

A. Scheduler

Since scheduler will be used by the sleep routine and the terminate routine, so I am to explain it as the first routine. The flow of scheduling is shown in figure 2.

First of all, the scheduler checks whether there are no more processes existing in the system, if no, halt the machine. Otherwise, check whether ready queue is empty, if so, indicating there are still some processes in the system, but since no one is ready, the machine goes to idle state to wait for the interrupt.

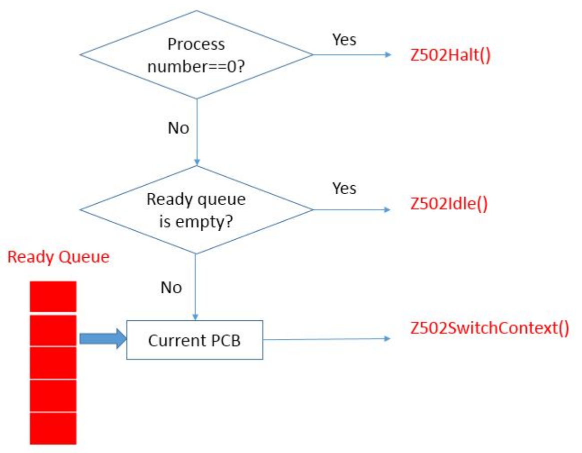

Figure 2. Scheduler

If there is at least one valid PCB in the ready queue, simply take one PCB with the highest priority, and switch context to it.

B. Restart Timer

Each time the interrupt handler detects an interrupt from the hardware timer, or sleep is called, the restart timer routine is needed. The flow of restart timer is shown in figure 3.

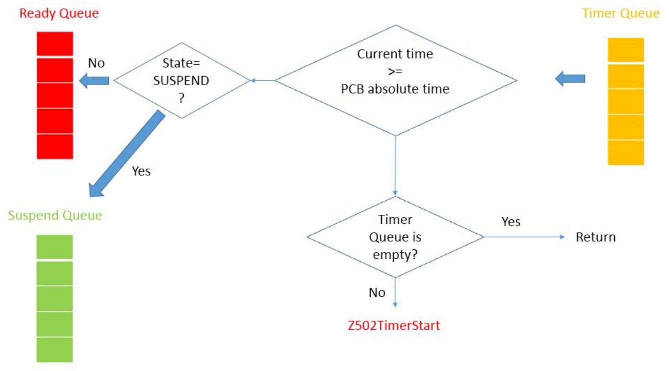

Figure 3. Restart Time

First of all, restart timer checks whether there are redundant PCBs in timer queue.There is such a possibility, for some reasons one PCB in timer queue has not been successfully waked up by the timer, or even the timer at that time didn't successfully trigger the interrupt. So some PCBs in the timer queue may have absolute wake up time smaller than current time. So restart timer routine simply pops out all the PCBs that should be waked up from timer queue, and according to its process state, to decide which queue it goes to. Then it checks whether the timer queue is empty, if so, there is no need to reset the hardware timer. If there is something in the timer queue, simply pop out the first PCB from the timer queue, based on its absolute wake up time, calculate how long the timer needs to give out an alarm, and restart the timer.

C. Timer Interrupt

The process of timer interrupt is quite easy with restart timer function ready. It simply cleans up the timer queue by comparing current time with PCB's absolute wake up time. After all the zombie PCBs as well as the PCB which causes the timer interrupt in timer queue has been popped out, it simply calls restart timer routine to reset the timer.

D. Create Process

The process of create process routine is shown in figure 4. First of all, it checks whether a valid PCB can be created, illegal priority, process with the same name with another process which already exists in the system, too many processes created exceeding the maximum number of processes my system supports can cause a failure to create a new process. After checking the validity of the new process, the routine mallocs corresponding space in memory and put the newly created process to ready queue.

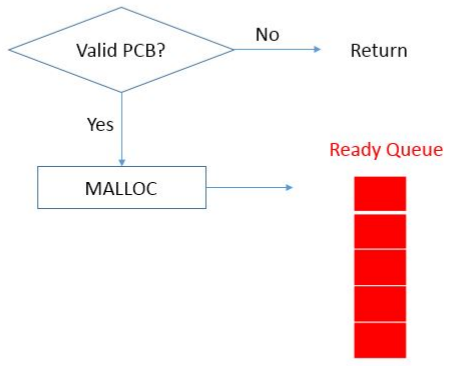

Figure 4. Create Process

E. Sleep

The process of sleep routine is shown in figure 5. In my system, current running context exists in current PCB as well as in the ready queue. So when sleep routine is called, current PCB will be taken out of the ready queue, and be added to the timer queue. After then, the sleep routine calls restart timer followed by the scheduler.

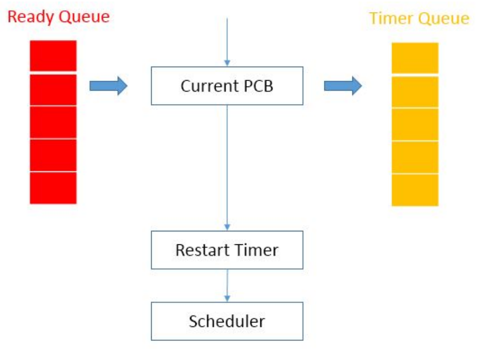

Figure 5. Sleep Routine

F. Get Time of Day

It simply calls MEM_READ(Z502ClockState, &current_time) function to get current time of the machine, then returns it to user function.

G. Terminate Process

Terminate process is shown in figure 6. If input tag is -2, then the routine halts the machine. If input tag is -1, which means terminate current PCB, it will remove current PCB from the ready queue since in my design, current PCB coexists in ready queue. Otherwise, terminate a specific PCB by its process ID, and removes it from one of the three queues (timer queue, ready queue, suspend queue). No matter current PCB or another PCB has been removed, the terminate process routine needs to check again whether the deleted PCB is the last PCB in my system, if so, halt the machine, if not, call scheduler.

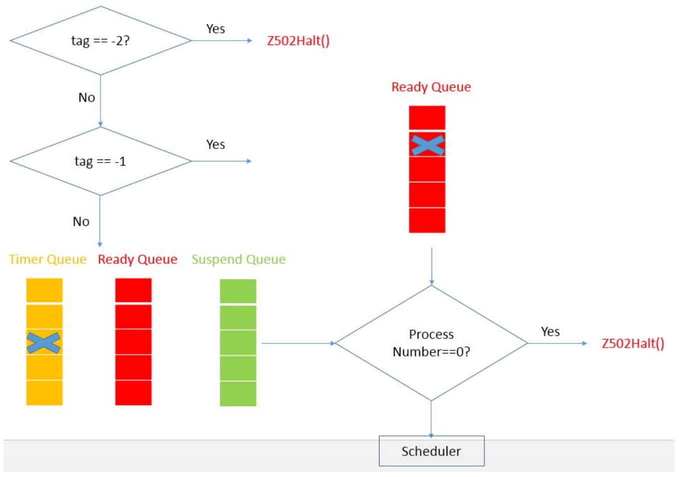

Figure 6. Terminate Process

H. Get Process ID

The get process ID routine simply checks whether input name is empty, if so, return current PCB's process ID. Then it will check the input process name, whether there is such a process in one of the three queues, timer queue, ready queue, suspend queue. If such a process exists, return its process ID, otherwise return NULL. The process is plotted in figure 7.

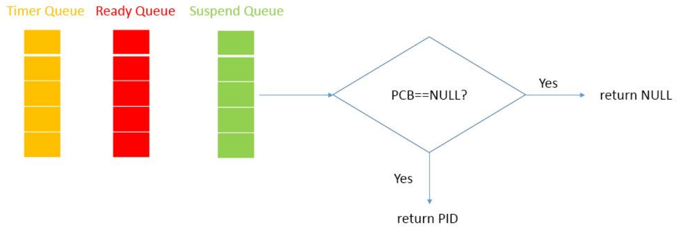

Figure 7. Get Process ID

I. Suspend Process

The suspend process routine is shown in figure 8. First of all, the routine checks whether it's legal to suspend target PCB. Situations like PCB with the input process ID doesn't exist, the target PCB has already been suspended, trying to suspend current running PCB are illegal operations to suspend a process. After checking the validity of suspend target PCB, it checks whether target PCB exists in ready queue, if so, move it to suspend queue. Otherwise, which means target PCB exists in timer queue, it simply sets process state to SUSPEND mode, then in other routines like sleep routine, when PCBs are popped out from timer queue, it will push the PCB to either suspend queue or ready queue according to its process state.

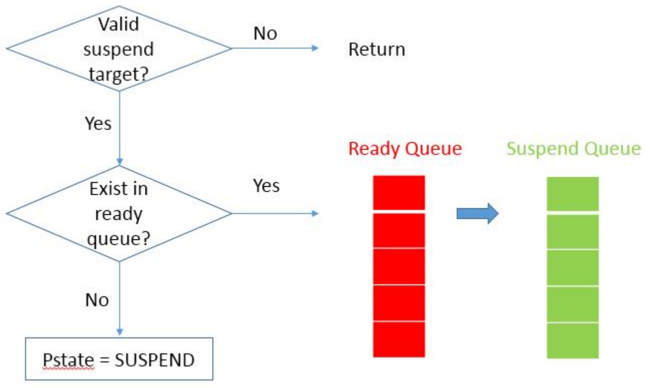

Figure 8. Suspend Process

J. Resume Process

The resume process is simply the inverse of suspend process routine, the procedure is shown in figure 9.

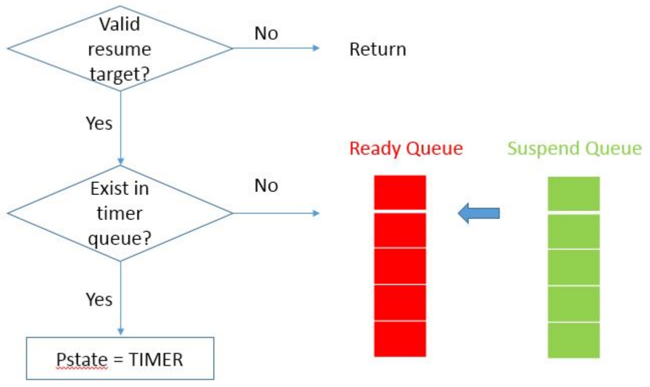

Figure 9. Resume Process

First of all, the routine checks whether target PCB is a valid PCB to be resumed. PCB with target process ID doesn't exist or target PCB has not been suspended will cause resume process to be failed. After then, it checks whether target PCB exists in timer queue, if so, simply sets its process state to TIMER mode. Otherwise take target PCB out of the suspend queue, and push it into ready queue. In this case, even if a PCB is in WAITING_IN_SUSPEND state, it can still be resumed by the resume process routine.

K. *Change Priority*

The change priority routine is simple, first check whether target PCB exists in the system and target priority is legal which should fall in the range 0 to 99. Then if changed PCB exists in ready queue, the ready queue needs to be reordered since ready queue in my system is arranged by PCB's priority. Otherwise, directly return success. The procedure is shown in figure 10.

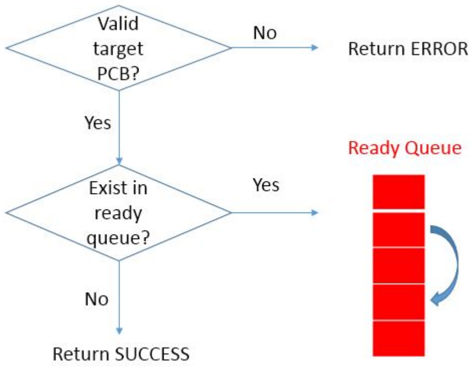

Figure 10. Change Priority

L. *Send Message*

Send message routine is a little complex, the process is shown in figure 11. First of all, it checks whether it's safe to create a new message. The attempt to create the 11th message, the attempt to create a message with message contents larger than 512 characters, and the attempt to send to a non-existing PCB causes failure to send a message. After checking the validity of message, the routine mallocs corresponding space for the new message, then add the newly created message to message queue. After then, the routine checks whether user wants to broadcast the message, this may lead to different PCBs to be taken from the suspend queue. At the same time, the PCB taken from suspend queue must be in the state of WAITING_IN_SUSPEND, which differentiates itself from those PCBs which has been pushed into suspend queue by suspend process call. It's quite clear, although I have only one suspend queue in the system, PCBs in suspend queue are either in SUSPEND state or WAITING_IN_SUSPEND state, WAITING_IN_SUSPEND state means the PCB is waiting for message in suspend queue not waiting for resume call, although it can still be resumed by resume process routine. At last, the send message routine calls the scheduler.

M. *Receive Message*

The receive message routine is like the inverse of the send message routine. First of all, it checks whether the message can be received. If the message receive length is larger than 512 chars, or the source process ID of the message doesn't exist, the routine returns an error. If there is no message sending to current PCB, the current PCB needs to be put into suspend queue, and the process state is set to WAITING_IN_SUSPEND indicating the PCB will wait for message not for resume process. Now the scheduler needs to be called to switch context to a new PCB from ready queue. If there is a message sending to current PCB, then the routine searches for a valid message depending on whether it's receiving from broadcast or not. If no such message exists, then suspend current PCB, otherwise, copy over the message back to user function and call the scheduler. The process is shown in figure 12.

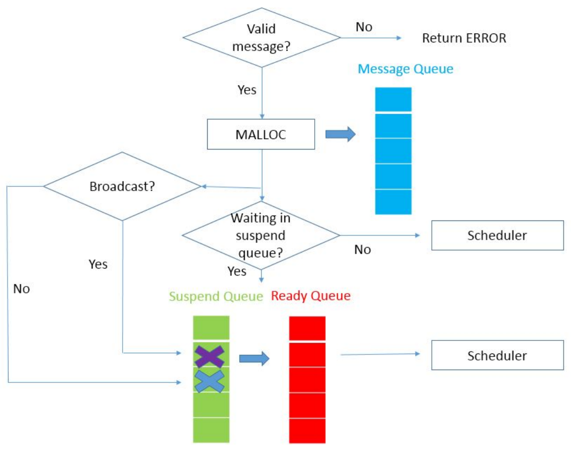

Figure 11. Send Message

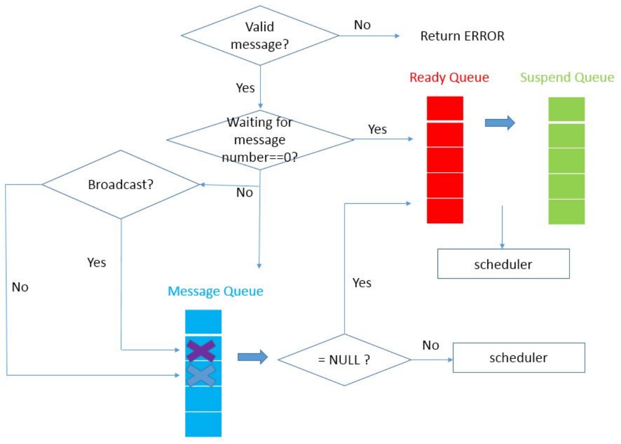

Figure 12. Receive Message

III. Justification of High Level Design

Here, all the tests from test0 to test11 have been successfully tested. These tests test my operating system from many aspects. Here all the tests are to be explained one by one to justify my operating system works properly. Since test0 is the default test, and very easy, I will not explain it, the test result is in test0.txt.

A. test1a

Test1a tests get time of day, the sleep call puts current PCB to timer queue and set machine to idle state, then machine enters interrupt handler, and terminate call ends the machine.

B. test1b

Test1b creates as many processes as it can, in my system, only up to 20 processes are allowed to be created, after the creation, the user tests get process ID, then terminate the machine.

C. test1c

It is a little complex, since 6 processes will sleep in turn, so my operating system will switch context back and forth from ready queue and timer queue. In figure 13, test1c starts from creating test1c_a, test1c_b, test1c_c, test1c_d, test1c_e besides test1c itself. Then it goes to a long sleep for 1000 time units. Now the current running PCB will be taken out of ready queue and be pushed into timer queue, hardware timer will be set in the meantime. Then scheduler is called to run the process with the first item in ready queue which is test1c_a. Test1c_a runs function test1x, in test1x, except SVC calls like get process ID, get time of day, it triggers another sleep routine, which will put test1c_a into timer queue. So in the current running thread, test1c_b o test1c_e are to be pushed into timer queue one by one. On the other hand, interrupt handler runs simultaneously and can be triggered any time by the hardware timer. Each time the timer interrupt is triggered, it first cleans up PCBs should be waked up in timer queue, then goes to the restart timer to set the next alarm.

So in test1c, PCBs are jumping between ready queue and timer queue. Since when one PCB has been executed the second time, the context knows where it was stopped from, avoiding running the test1x all over again, which will cause the program run forever. So after all the processes from test1c_a to test1c_e have been executed, the test goes back to execute test1c and terminate the machine successfully. More details are shown in test1c.txt file.

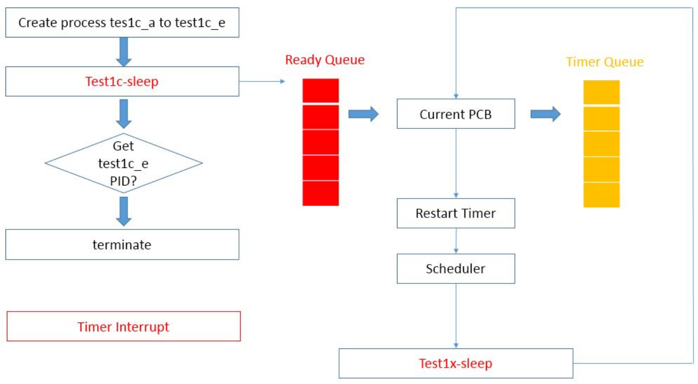

Figure 13. Test1c

D. test1d

Test1d resembles test1c, and the only difference is test1d's newly created processes come with different priorities.

E. test1e

Test1e mainly tests the routines of suspend and resume process. It first creates a new process test1e_a, then suspends it. Then it suspends it again, which causes an error. After suspending, test1e resumes the suspended process. Then it tries to resume it again which causes an error. And it tests whether it is illegal to resume itself in the current machine, due to architectural decision, it is illegal to resume itself in my operating system. At last, test1e terminates the test. My operating system runs successfully for test1e.

F. test1f

The brief process of test1f is shown in figure 14. Firstly, test1f creates test1f_a to test1f_e these 5 processes. Then it calls sleep which puts test1f itself to timer queue, in this way scheduler switches context to run test1f_a, here the procedure is quite like what test1c does. However, when test1f is waked up from the timer queue and then taken from ready queue to be the current running process, it goes into a while loop. In the loop, test1f_a, tes1f_c, test1f_e are suspended, then test1f calls sleep to put test1f process into timer queue. Now the scheduler can only choose test1f_b or test1f_d to run, it will wait until test1f has been waked up, and becomes current running process, test1f_a, test1f_c, test1f_e can get the chance to be resumed from suspend queue into ready queue. Then the process will go over the same process again. After the loop has been executed, test1f goes on a long sleep, which gives other processes enough time to execute.After all the processes from test1f_a to test1f_e have been executed, test1f waits to be waked up by the timer, after test1f has been waked up, at last, the test1f terminates the test.

In this process, I can see test1f_a, test1f_c, test1f_e, with process ID 1, 3, 5 in my system, have fewer chances to execute than test1f_b, test1f_d.

So here I copy over part of the print information of test1f:

```
Test1x, PID 2, Ends at Time 1851
Test1x, PID 4, Ends at Time 2786
Test1x, PID 1, Ends at Time 3450
Test1x, PID 3, Ends at Time 4169
Test1x, PID 5, Ends at Time 4805
```

Since test1f_b, test1f_d have more chances to execute, so they finish earlier than the other three processes.

From here, test1f can justify that my operating system works properly with sleep, suspend, resume. More details regarding the three queues' status are shown in test1f.txt.

G. test1g

Test1g tests the change priority routine, it creates a new process test1g_a, then tries to change the priority of an illegal process ID, gets an error, then changes its priority to an illegal one, which causes an error. At last, test1g changes priority of test1_a to 10 which is in the range [0,99] indicating legal in my system. Test1g runs successfully.

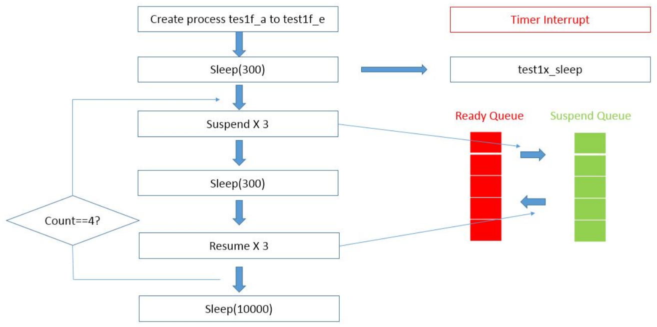

Figure 14. Test1f

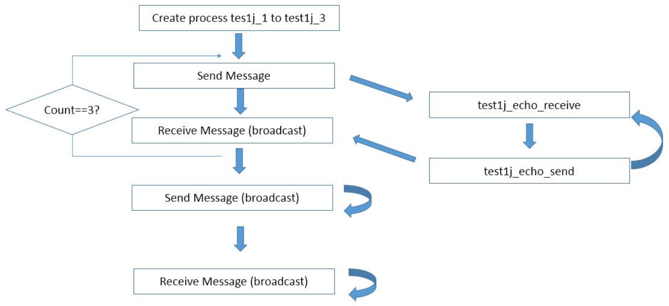
Create process tes1j_1 to test1j_3
Send Message
Count==3?
test1j_echo_receive
Receive Message (broadcast)
test1j_echo_send
Send Message (broadcast)
Receive Message (broadcast)

Figure 15. Test1j

H. *test1h*

Test1h tests change priority routine working together with sleep routine. It first creates three new processes test1h_a, test1h_b, test1h_c, then calls sleep to put test1h into timer queue. Now test1h_a becomes current running process, and it starts executing test1x. This process is like what test1c does, while since test1f only sleeps for 200 time units, it soon wakes up from timer queue and becomes current running process. Then it changes priority of test1h_a and test1h_c, then sleeps again for 200 time units. Then in the ready queue, test1h_a always stands ahead of test1h_b and test1h_c. Then test1f changes the priority again, now test1h_c becomes the most favorable PCB in ready queue. At last, test1f sleeps for 600 time units and terminates. Test1h runs successfully.

I. *test1i*

Test1i mainly tests send message and receive message routines. It tests whether my send message routine and receive message routine can handle some error cases. And at last, it sends as many legal messages as it can to test1i_a, in my system, up to 10 message are allowed to be created.

J. *test1j*

Since all the other system service calls used in test1i has already been tested with previous tests, so I only focus on send and receive routines in test1j. First of all, test1j creates three processes tes1j_1, test1j_2, tes1j_3. Test1j sends a message to test1j_1, then test1j_1 receives the message and is scheduled to run as current running process. It executes test1j_echo then, here it first receives message from broadcast, then sends exact the same message back to test1j. The loop will execute three times in this way. After then, test1j broadcasts as many messages as it can, then it receives these 10 messages successfully.

K. *test1k*

Test1k is very easy, simply print out a warning information in fault handler when the user attempts to access hardware related instructions at user mode. And the halt the machine after detecting fault handler detects the error.

IV. Additional features

In this project, test1l tests additional features of my operating system, I have passed test1l successfully.

A. *test1l*

Test1l further tests three cases. Here I am going to explain them one by one. Test1l first makes test1l process to be the highest priority process. Then it creates a new process test1l_1, which will execute test1j_echo.

1) case 1: Sleep puts test1j into timer queue, now test1j_1 is running. The test1j_1 has been suspended, scheduler takes test1j to be current running process. Then test1j_1 is resumed. Now test1j_1 is to be current running process and executes test1j_echo. It receives one message with length 30 from test1j, and sends it back. Test1j receives the message successfully.

2) case 2: In case 2, it first creates a new process test1l_2. Then test1j is pushed into timer queue by sleep call, test1j_1 is suspended. Test1j_2 executes test1j_echo, while there is no message sent so test1j_2 simply suspends itself, and test1j comes back to running again. Then it broadcasts a message to let test1j_2 no longer waiting in suspend queue, and then resumes test1j_1 from suspend queue. Then it wants to receive a message from test1j_2, but test1j_2 didn't send it yes. So test1j suspends itself and wait for test1j_2 to execute test1j_echo, which sends out a message to test1j. After then test1j can go on receive message successfully.

3) case 3: It sends a message specifically to test1j_1, then resumes it. The rest of case 3 resembles that of case2, and case 3 has been executed successfully on my system.

V. Anomalies and Bugs

Many times Z502Idle() in my case cannot truly put the machine in idle state, so I have to put it into a while loop in my scheduler function like this:

```
        while(ready_head == NULL)
        {
        READ_MODIFY(READYQ_LOCK,    0,    1,
    &lock_result);
        Z502Idle();
        READ_MODIFY(READYQ_LOCK,    1,    1,
    &lock_result);
        }
```