CS 452 – Kernel 1

January 27, 2017

Zhengkun Chen
Student ID: 20557054

Xiwen Liu
Student ID: 20468618

Operating Instructions

Full pathname to executable: /u/cs452/tftp/ARM/z283chen/kernel.elf

Redboot must be restarted before loading the program.

To load and run the program in the Redboot terminal, run the following command:

> load -b 0x00218000 -h 10.15.167.5 "ARM/z283chen/kernel.elf" ; go

The executable can also be created by going into the directory “kernel” and calling make:

> cd kernel

> make

And then copy the elf to wherever you want to run it.

Kernel Primitives

All kernel primitives were implemented as specified in the assignment documents provided:

- int Create(int priority, void (*code) ( ))
- int MyTid( )
- int MyParentTid( )
- void Pass( )
- void Exit( )

Each of these functions create a software interrupt with a designated code, and gives control to the kernel to handle the request. The active task is rescheduled whenever it calls Create, MyTid, MyParentTid, or Pass.

Kernel Structure

Algorithms

- Context switching
    - During initialization, the label for the software interrupt handler is loaded into address 0x28.
    - Context switching was implemented with inline assembly.
    - Entering Kernel: The kernel saves user’s registers onto user stack, and retrieve registers from its own stack.
    - Exiting Kernel: The kernel saves its registers onto the kernel stack, retrieve registers for user program and load link register into pc and then hands over the control to user program.
- Creating task descriptors
    - Task descriptors are located at a reserved memory address that is slightly under kernel stack. A task descriptor contains information for task to run, pause, and resume, including link register (when entering kernel), return value of a syscall, and much more.

- Deleting tasks
    - When a task calls Exit(), it is removed from the priority queue.
    - Task descriptors are not removed from memory. A maximum of 50 tasks can be created.
- Scheduling
    - The scheduler chooses the task with the highest priority. The scheduling functions require the priority of the task as a parameter. Adding a new task to the priority queues, removing a task, scheduling a new task, and rescheduling a task each take O(1) time.
    - Round robin scheduling is used for adding new tasks and rescheduling tasks, so the task is always added to the back of the queue.

Data Structures

- User task stack: This stack starts at 0x01200000, and each user task is allocated a stack size of 0x00015000 bytes. The stack grows downward as per GCC convention.
- Task Descriptors: An array of task descriptors begins at 0x01400000, which is a space reserved only for task descriptors. Using a global variable could be an alternative but it will be more work to access them via assembly code. A maximum of 50 tasks and task descriptors can be created.
    - A copy of the active task's task descriptor is maintained at address 0x01300000. This copy helps simplify retrieving information needed about the active task, such as its task id and priority.
- Priority Queues
    - The priority queue structure contains a circular queue of task id's, and the indices of the first and last values in the task id array. The size of the task id array is set to the maximum number of tasks.
    - An array of priority queue structures is allocated beginning at address 0x01600000, and grows upward. Each element in this array is a priority queue for a different priority. Priorities can be dynamically added or removed by changing the size of this array. Three priorities are currently implemented (high, medium, and low), but this can be increased simply by changing the array size. The highest priority is at index 0.
    - Since the scheduling functions take in a priority as a parameter, the array allows the correct queue to be selected in O(1) time. The first and last indices in the priority queue structure allow functions to add and remove tasks in O(1) time as well.

Source Code

Git repository: https://git.uwaterloo.ca/x272liu/cs452

kernel/

0c4fc8e853569cb03ceb76eee46d3c39 functions.c

4f76e12b039081aff09394da7367f37a kernel.c

d89514a0e29f25222329a2b4f3435416 Makefile

4aa618b9753c5292e5d9e5c95d297f10 orex.ld

811d4b793258c3dbd37a38b96fedfdd4 priorityqueue.c
72211e2f82546c6260deb79a3ba1bf09 run.sh
52146414fb75f94a6d6b9019f4ba2669 syscall.c
e7165db845560311cd14081188e7366c td.c
fc5375c81855cc1d031026f8ed8629b9 user_syscall.c

io/
d32dda3f6cd59b210c03d1ed8332c581 bwio.h
9af226f127c1fd759530cd45236c37b8 ts7200.h
ba868ea1845b6aa4af4cb1feee528228 libbwio.a

include/
cdf840278d6a47b16e8d028f0a752ba5 functions.h
0e018e36e07584d323e6df1c78782c25 kernel.h
19f116734e97d3c7d880086f904f9ac3 priorityqueue.h
77ceaab8b180bced834383ccc5599429 syscall.h
5424452ee669af151c103455cf2706b2 td.h
1ee81c8c1042e4e7b17f209f0b6bb9e0 user_syscall.h

Sha1 of the commit: 67733b194528002776037f829381044844a45efb

Program Output

Output:

1. Created task: tid # 2
2. Created task: tid # 3
3. My tid is: 4. My parent's tid is: 1.
4. My tid is: 4. My parent's tid is: 1.
5. Created task: tid # 4
6. My tid is: 5. My parent's tid is: 1.
7. My tid is: 5. My parent's tid is: 1.
8. Created task: tid # 5
9. First user task: exiting.
10. My tid is: 2. My parent's tid is: 1.
11. My tid is: 3. My parent's tid is: 1.
12. My tid is: 2. My parent's tid is: 1.
13. My tid is: 3. My parent's tid is: 1.

Explanation of output:

1. The first user task (tid 1) is created during initialization.
2. Task 1 first creates a lower priority task 2. Task 2 is a lower priority task, so task 1 is still the active task after rescheduling. Create() returns, so task 1 prints line 1.
3. Task 1 then creates task 3. Task 3 is also a lower priority task, so task 1 is still the active task after rescheduling. Create() returns, so task 1 prints line 2.
4. Task 1 creates task 4, which has higher priority than task 1. Since task 4 is of higher priority, it is scheduled as the active task.
5. Task 4 prints out line 3. It calls Pass(), and is selected as the active task again since it still has the highest priority. It prints out line 3, and then calls Exit().
6. Task 1 is scheduled as the active task since it now has the highest priority. Create() returns, so it prints line 5.
7. Task 1 creates task 5, which has higher priority than task 1. Since task 5 is of higher priority, it is scheduled as the active task.
8. Task 4 prints out line 6. It calls Pass(), and is selected as the active task again since it still has the highest priority. It prints out line 7, and then calls Exit().
9. Task 1 is scheduled as the active task since it now has the highest priority. Create() returns, so it prints line 8.
10. Task 1 prints out line 9 and then calls Exit().
11. Task 2 is scheduled as the active task, since it was created before task 3.
12. Task 2 calls tid(), it is pushed to the end of queue. then task 3 calls tid() it gets pushed to the end of queue.
13. Repeat step 12 for parent_tid() for task 2 and task 3.
14. Task 1 prints out line 10, and then calls Pass(). It is moved to the end of the priority queue.
15. Task 2 prints out line 11, and then calls Pass(). It is moved to the end of the priority queue.
16. Repeat Step 11 to 13.
17. Task 2 is scheduled as the active task. It prints line 12, and then calls Exit(). It is removed from the priority queue.
18. Task 3 is scheduled as the active task. It prints line 13, and then calls Exit(). It is removed from the priority queue.
19. No more tasks are in the priority queue, so the program exits.