CS 452 – Kernel 4

February 17, 2017

Zhengkun Chen

Student ID: 20557054

Xiwen Liu

Student ID: 20468618

Operating Instructions

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

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 kernel 3 assignment documents.

Kernel Structure

<u>Context Switch</u>

- Software Interrupts
  - 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.
- Hardware Interrupts
  - During initialization, the label for the IRQ handler is loaded into address 0x38. The SWI handler is a subset of the IRQ handler.
  - Entering kernel: On entering the kernel, the IRQ handler will first save user state, including the sp. Then it transfers the spsr and lr of IRQ mode into SVC mode. After this, it runs the same instructions as SWI.
  - Exiting kernel: If the kernel is exiting from an interrupt, it will load back all of the user registers it initially saved, as well as the user lr. Then the rest of the instructions are the same as SWI.

<u>User Tasks</u>

Algorithms

- 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.

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.

Scheduling

Algorithms

- 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

- 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. There are 32 priorities available.
- 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.

Message Passing

Algorithms

- Send before Receive: send adds the current task to receiver task's sendQ; blocks itself. And then receiver task will extract the info from its sendQ. Copy the content of the message from sender's space to receiver's space (done by kernel). And then receiver should call reply accordingly to send back the reply to the sender, and unblocks the sender.
- Receive before Send: the receiver first blocks itself, we implemented this by setting the state of current task to blocked and calling a pass. And once a sender is detected, it is resumed.

Data Structures

- The sendQ structure is a circular queue, with pointers to the first and last elements in the queue.

Interrupt Handling

- AwaitEvent(): It sets the current task to EVENT_BLOCKED. During the IRQ handling, the corresponding task (which is usually a notifier) that is waiting on the interrupt will be unblocked and resumed to process the data received from that interrupt.

Nameserver

Algorithms

- The nameserver performs linear searching for WhoIs() and linear insertion for RegisterAs(). The worst case time is O(n), where n is the number of names that have been inserted. The maximum number of names that can be inserted is 50, so the worst case time is short for both operations.

- Our implementation of strcpy, which is identical to its implementation in string.h, was used to copy the given name to the nameserver structure.

Data Structures

- The nameserver begins at address 0x018000000, and grows up. It is an array of nameserver structures, which contain the fields name and tid. Name a C-style string and tid is an integer.
- The array is not circular, and names cannot be removed, so a maximum of 50 names can be registered.

<u>Clock server</u>

Algorithms

- The clock server functions Time, Delay, and DelayUntil use the message passing primitives Send, Receive, and Reply to block the calling task until it receives the requested data. The clock server calls receive first and is receive blocked until a client calls one of Time, Delay, or DelayUntil, which will then call Send and so on.
- Delay is a wrapper for DelayUntil. It adds the current time to the input time, and then calls DelayUntil with the new time. DelayTime then causes the server to add the task to the delay queue. Calling Delay or DelayUntil blocks the task until the calculated system time has been reached.
- New delayed tasks are added to the delay queue with insertion sort, so the task with the shortest delay time is always at the front of the queue. This runs in O(n). Removing a task from the queue also takes O(n), since it is not a circular array. We intend to replace these structures and algorithms with more efficient ones when the load on the program increases.

Data Structures

- For delaying tasks, the clock servers uses an array of structures. This structure contains the fields tid and delay time. The delay time is the system time of when the task should be unblocked.
- The clock server uses Timer3 to keep time.

<u>Idle Task</u>

- The idle task is set at the lowest priority. When it first runs, it initiates Timer1, and then enters a while loop. The idle task keeps track of how many ticks has passed while it is running. The idle usage is calculated by dividing the amount of ticks counted by the idle task over the amount of ticks counted by the program.

<u>Input/Output</u>

Data Structures

- Four circular character buffers:
  - Terminal send buffer: characters sent to the terminal (keystrokes)
  - Terminal receive buffer: characters from the terminal (output to monitor)
  - Train send buffer: characters sent to the train (train commands)
  - Train receive buffer: characters received from the train (sensor data)
- Two servers were implemented. Handling IO from the terminal and train controller was separated into two servers to improve performance, since the controllers operate at significantly different speeds. Creating four servers would have created more overhead and been more confusing to manage.
  - An IO server manages adding and removing characters from the four buffers, when a transmit or receive interrupt occurs. This server handles
  - A courier server sends command requests to the train controller. This centralises control so that commands sent to the train cannot be interrupted and sent out of order.

Algorithms

- Insertion and deletion for the four circular buffer takes O(1) time.
- Putc(uart, c) directly adds character c to the send buffer for the UART specified, since sending a request to the IO server has additional overhead, and leads to rescheduling.
- Getc sends a request to the IO server.
    - If there are no characters in the receive buffer, it puts the task into a wait queue. When a receive interrupt occurs, the first task in the wait queue is activated and can then receive the character.
    - If there are characters in the receive buffer, it removes and returns the first character from it.
- A character is added to the appropriate receive buffer whenever a receive interrupt raises.

Source Code

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

Sha1 of the commit: 95362e45126d99c7e820e51d22d2f4cd1d6f0433

All files in the repository are part of this submission.

Train Control / Assignment 0

Commands

tr <train-num> <speed>

Sets a train in motion at the specified speed.

rv <train-num>

Reverses the selected train.

Reversing the train takes 7 seconds in total. The train speed is first set to 0. Then the program waits 5 seconds for train to stop, and then sends the command to reverse the direction of the train. Then it waits for another 2 seconds before sending the command to set the train speed to its previous speed. These time intervals were chosen to ensure that the train has time to slow down and will receive the commands properly.

The program uses a circular wait queue to keep track of when a command should be issued to a train. The buffer stores the train number, speed, and the amount of time it needs to wait. During each timer interrupt, this queue is checked to see if any trains are ready for commands. Any train instructions entered during the 7 seconds it takes to reverse the train have no effect on the train.

sw <switch-num> <direction>

Sets the selected switch in the selected direction.

When the switches are set during initialization, the program waits 0.5 seconds after each switch has been set to ensure that the command is received properly.

q – Quits the program

Things to note:

- Backspace and the Enter key are recognized.
- There is unexpected behaviour if special characters such as the arrow keys are pressed. Please try to enter commands exactly as specified.
- While the train is reversing, do not send any other instructions to the train, or they may be unexpected behaviour.