CS 452 – Kernel 2

January 30, 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( )
- int Send(int tid, void *msg, int msglen, void *reply, int replylen)
- int Receive(int *tid, void *msg, int msglen), and
- int Reply(int tid, void *reply, int replylen)

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

<u>Algorithms</u>

- 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.

- Message passing
    - Send before Receive: send adds the current task to receiver task's sendQ and then blocks itself. Then the receiver task will extract the info from its sendQ. The content of the message is copied 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.
- Nameserver
    - 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.
- RPS server
    - The RPS server uses similar algorithms as the nameserver, so the worst case time is also O(n) for updating client data.
    - We created our own srand() and rand() to generate the client moves (i.e. rock, paper, or scissors).
    - In our implementation, a client must sign up before playing or quitting. Once the client plays, it must sign up again in order to play.

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.
- Nameserver:
    - 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.

Source Code

Git repository: https://git.uwaterloo.ca/x272liu/cs452
Sha1 of the commit: 2758a896e761cc83de94c724a558555420758517
All files in the repository are part of this submission.

Priority chosen for game tasks

All game tasks were set to the lowest priority, with the nameserver and rps server at the highest priority. This ensures that the servers are run often and process all incoming requests.

Time

Message	Caches	Send before Reply	Optimization	Time (µs)
4 bytes	off	yes	off	830
64 bytes	off	yes	off	1470
4 bytes	on	yes	off	50
64 bytes	on	yes	off	100
4 bytes	off	no	off	920
64 bytes	off	no	off	1570
4 bytes	on	no	off	60
64 bytes	on	no	off	100
4 bytes	off	yes	on	broken
64 bytes	off	yes	on	broken
4 bytes	on	yes	on	broken
64 bytes	on	yes	on	broken
4 bytes	off	no	on	broken
64 bytes	off	no	on	broken
4 bytes	on	no	on	broken
64 bytes	on	no	on	broken

We believe that the majority the time spent is by the memcpy(). Since it reads and writes sequential data in RAM whose performance is improved dramatically by turning on cache. Because turning on the cache only affects the speed of memcpy(), we know that the context switches must take at most 50 microseconds, which is the fastest time we achieved with cache on. The time taken up by context switching remains constant in each of the tests conducted. This means that any additional time must be cause by memcpy(). Since the extra time is mostly larger than 50 microseconds with cache on, this extra time must be spent in memcpy().

Program Output

```
TID 4 plays TID 5
TID 4 has quit.

Press any key to continue.

TID 6 plays Rock and TID 7 plays Scissors
Result: TID 6 wins!

Press any key to continue.

TID 8 plays Rock and TID 9 plays Paper
Result: TID 9 wins!

Press any key to continue.

TID 10 plays Scissors and TID 11 plays Paper
Result: TID 10 wins!

Press any key to continue.

TID 12 plays Scissors and TID 5 plays Scissors
Result: Tied

Press any key to continue.

TID 6 plays Scissors and TID 9 plays Paper
Result: TID 6 wins!

Press any key to continue.

TID 10 plays Paper and TID 12 plays Rock
Result: TID 10 wins!

Press any key to continue.

TID 5 plays Rock and TID 6 plays Rock
Result: Tied

Press any key to continue.

TID 10 plays Rock and TID 12 plays Paper
Result: TID 12 wins!

Press any key to continue.

TID 5 plays Rock and TID 6 plays Scissors
Result: TID 5 wins!

Press any key to continue.
```

Explanation of output:

The first client created is a client that signs up and then quits.

Then we generate 8 tasks that will sign up and play until it loses or until there is only one task left. We do this by calling Create() on the same function 8 times. This task never quits.

In the first round, the rps server gets the client choices from tid 4 and tid 5. Since tid 4 quits and exits, it does not play again.

The first two lines:

```
TID 4 plays TID 5
TID 4 has quit.
```

is the result of TID 4 quitting.

TID 4 exits after it quits, so it never plays again. From here, the remaining 8 tasks play repeatedly with each other, until all but one have lost. The output from this point is random, since the client choices are randomly generated.

Since all clients are at the same priority, they are scheduled sequentially in the order of created, which is why the clients appear in order.