Assignment K2

Edwin Zhang and Bill Cui

Path to executable

`cs452-a1/build/kernel`

Access, make, operate

```
git clone ist-git@git.uwaterloo.ca:b22cui/cs452-a1.git

cd cs452-a1

chmod a+x compile_target.sh

./compile_target.sh

cd ./build
```

Then, move the `kernel` onto the track computer

```
cp kernel /u/cs452/tftp/ARM/e42zhang

chmod o+r /u/cs452/tftp/ARM/e42zhang/kernel
```

To run the program, issue the following:

1. `load -h 10.15.167.5 ARM/e42zhang/kernel`
2. `go`

Operating Instructions

To run the program, issue the following:

1. `load -h 10.15.167.5 ARM/e42zhang/kernel`
2. `go`

Program Description

(NEW) Send Recieve Reply

Send receive reply is implemented through the following syscalls:

```
int Send(int tid, const char *msg, int msglen, char *reply, int rplen);,

int Receive(int *tid, char *msg, int msglen);,

int Reply(int tid, const char *reply, int rplen);
```

Lets say we have two tasks. A and B. We hope to illustrate the algorithm with the following scenarios:

Scenario 1

A performs `Send` to B. Since B has not yet executed `Receive`, A is not added back into the ready queue as it is now blocked. Each TCB contains a queue called `want_send` implemented as a linked list. When A wants to `Send` to B before B is able to execute `Receive`, A is added to B's `want_send` queue. Now later on when B executes `Receive`, it first sees if any other TCBs wish to send to it. If there is then we call `handle_send`. Otherwise B is taken off the ready queue with state == `RECEIVE`. B can then only be placed back into the ready queue when a TCB wants to send to it.

Lets continue to assume that A is sending to B, `handle_send` performs the following:

1. A is set to be REPLY. It is still blocked, but is now awaiting a reply from B.
2. B is placed back onto the ready queue$ with status READY
3. The contents in A's `send_args` is then copied to B's `receive_args` through `msg_copy`.

Now A is still blocked but B is free to run. Later on when B executes `Reply`, `msg_copy` is executed to copy B's message back to A, both tasks then get placed back onto the ready queue.

(NEW) Nameserver

The nameserver provides functionality such as `int RegisterAs(const char *name);` and `int WhoIs(const char *name);`

These are really just wrappers around `Send`, `Receive`, and `Reply`.

Nameserver's overarching datastructure is a hashtable. Since there are no syscalls for "deregistering" a TCB from a name, the hashtable only has get and insert operations. The hashtable uses a singular modulos hash function that is unit tested in `/tests/unit/hashtable.c`.

When a task wishes to register itself to a name, it calls `RegisterAs`. The nameserver, upon `Receive` is able to get the TCB_tid of the task through the `closure_mechanism`. In other words the task's id is obtained as a part of `send_params` instead of manually requiring the TCB to pass its id for `RegisterAs`. Now that the nameserver knows the taskid of the TCB as well as the name it wishes to register as (passed as a `nameserver_request` struct). It can perform the hashtable insert. When a different task wishes to register as the same name, the previous task's entry would be overwritten.

When a task performs `WhoIs`, the nameserver performs a hash table lookup and will then `Reply` the tid as part of the `nameserver_response` struct. Before passing to the kernel the struct is serialized into a char array and is deserialized on the other end back into a struct.

The `nameserver_response` and `nameserver_request` structs contains a `type` and `body`. The `type` allows for an easy and readable way to differentiate between request and response types. Kind of like the `GET`, `POST`,... pattern in web servers. The body is an unstructured array that is given structure through context given by the `type`. The receive and response end can parse the `body` according to the `type`. For instance if the request type is `REQUEST_REGISTER_AS`, then the nameserver knows that there is no information in the body, and to return a response of type `RESPONSE_GOOD` with no body. If the request type was `REQUEST_WHO_IS` instead, then the nameserver knows to look in the body to obtain the name. The task who initated this request would also know to look in the body to obtain the returned task_id.

(NEW) RPS

The `RPSserver` follows a similar pattern as the nameserver. There are many different response and request types to organize the code.

There is a pointer `free_game` that always points to the next game that is free to be used.

Suppose there are two tasks, A and B. A calls `SignUp`. Since there is no player 1 in `free_game`, the RPSserver does not reply to A, therefore blocking it. A is then set to be player 1. Later on when B calls `SignUp` as well, since there is a player 1 the server replies to both tasks, allowing for them to go back into the ready queue.

Similar story for playing (`REQUEST_PLAY`). The server will block a task before the other task makes a move as well. Once both tasks/players have made their moves, the server evaluates the moves and determines a winner. The corresponding response types are then used as an easy way for tasks to determine if they had won, lost, or tied. There is a fourth response type, `RESPONSE_GAME_ENDED`, which is used when a player quits a game.

There are different scenarios for quitting.

1. A performs `Quit` before B has made a move. In this situation we can just remove A from the game. Later on when B makes a move it notices that A is no longer in the game and thus it quits as well. The game is then recycled.
2. A performs `Quit` after B has made a move. B is waiting for the server to reply in this situation, so we just reply to B with the response that A has quit.

The user task `task_k2rpsinit` creates the nameserver, the RPSserver, as well as two players. Be sure to add it to the schedular in `main.c` if you wish to test out RPS. One player is user controlled, the other is a bot. The bot is set to quit after 10 moves. This allows for testing of various quit scenarios.

(NEW) RPS Program Output Explained

Upon executing the user task, both the player and the bot calls `SignUp`. The first game, which is game 0, is replied to both players. This is why `You are in game 0` appears on the screen.

```
======================================================
|                      RPS GAME                      |
======================================================
You are in game 0

Choose move? (R=rock|P=paper|S=scissor|Q=quit)


Wins: 0, Losses: 0, Ties: 0
```

Now the user chooses `Rock` as its next move. We are met with this output:

```
You are in game 0
You tied
```

```
Choose move? (R=rock|P=paper|S=scissor|Q=quit)

[Bot] Chose rock
Wins: 0, Losses: 0, Ties: 1
```

The bot chose rock as well, so both players tied. The number of ties increments.

Now the user chooses `Rock` again.

```
You lost
Choose move? (R=rock|P=paper|S=scissor|Q=quit)

[Bot] Chose scissor
Wins: 0, Losses: 1, Ties: 1
```

Asserts

Defined in `my_assert.h`, when an assertion fails, a sad train ascii art is printed onto the screen. Additionally, developers can add context about the failed assertion as part of the second parameter of `KASSERT`

Tasks

A TCB struct serves two purposes: 1. It is a free slab of memory that points to the next free slab of memory. 2. It contains information of a task.

There is a pointer to the next free slab of memory that can used to hold a new task. All the task memory slabs reside in a TCB array. This implements intrusive linkage and avoids the use of `free` and `malloc`, as all the memory that tasks will ever need is allocated on the stack as the `TCB[]` array.

Tasks contain the stack (an array of 32 bit integers), and its register struct. When the task is running, its register struct can also be accessed globally (and in assembly) as a global variable points to it.

Since the word size of ARM is 4 bytes, the stack is an array of `uint32_t`.

The stack size was chosen to be $2048 \cdot wordsize$. This allows for, theoretically, $\frac{32 \cdot 10^{6}}{2048 \cdot 4} \approx 3906$ tasks. In reality that is definitely not the case because available memory would have been used to store other data structures as well. Thus, each task has a stack size of $2048 \cdot wordsize = 8192$ bytes.

For K1, the maximum number of tasks allowed (`MAX_NUM_TASKS`) was chosen to be 10. This is because only a small fixed number of tasks are executed in this version of the kernel. This number will be increased for K2. [K2 UPDATE] the `MAX_NUM_TASKS` is now set to be 1000 to faciliate for multiple RPS games.

Scheduling

The scheduler uses a fixed size array based heap that stores the pointers of the TCB's. The actual contents of the TCB's are stored in the TCB array declared on the stack of the kernel's main function.

The heap is a max heap and thus allows for tasks with the highest priorities to be popped in $O(\log(n))$ time. For tasks that have equivalent priorities, the timestamp of when they were added to the queue is used as a tie breaker. This also ensures that the heap behaves like a FIFO queue when all the tasks have the same priorities.

Context Switching

The kernel register is a global struct, similarly the pointer to the register struct of the current task is also a global variable. This allowed for us to reference the register structs directly in assembly. Registers can be accessed and modified without needing to directly modify the task stack.

`switch_user` is used to store kernel registers and load the registers of the next task

`return_swi` is used to store the registers of the currently running task and load the kernel registers. It is the swi handler so it's memory address is stored at `0x28`, where the hardware uses as the swi vector.

The context switching implementation can be better explained by following through with a walkthrough of a sample program flow:

Upon creating and adding the first task, the kernel goes into a continuous loop. It first takes the task at the head of the ready queue and selects for it to be executed. The task register pointer `user_reg` is updated to point to the register struct of the chosen task. `switch_user` is then called which saves the kernel registers. In particular, the return address of `switch_user` is stored as the PC register in the register struct, so that when kernel state is restored the execution will resume past `switch_user`. The user task's registers are then loaded, and thus the user task continues execution.

When the user task performs a syscall, for example `Create(...)`, the arguments are placed on to `r0`. Then a software interrupt is triggered. PC is then set to `return_swi` by the hardware. User task registers are stored and the kernel state is reloaded. As mentioned earlier, since the PC of the kernel was saved to be the return register of `switch_user`, the kernel continues execution at the instruction right after `switch_user`. `switch_user` and `return_swi` combine to create the appearance that running the user task is as simple as calling a function. From the kernel's point of view, it has simply called a function and the function has returned what to do next. This luxury of simplicity enjoyed by the kernel is the materialization of the blood, sweat, and tears of assembly developers (me).

The arguments are then retrieved from the user task's `r0`. The type of the syscall is retrieved by retrieving the parameter of the `swi [...]` instruction that was executed by the user task. Since that instruction was the last instruction to be executed by the user task before the software interrupt, we were able to retrieve it by just decrementing the PC of the user task by 4. Once the system call has been serviced, the return result is stored in user task's `r0`. The user task is then placed back into the ready queue to be executed in the future. When the user task is selected in the future to continue execution, and once its registers and PC are restored, it would continue execution from the point after the `swi` instruction. The return result is stored in `r0`, which is retrieved and returned as a C function return. From the user task's point of view, it has simply called a function and the function has returned a value. This luxury of simplicity enjoyed by the user task is the materialization of the blood, sweat, and tears of kernel developers (also me).

Context Switching Program Output Explained

```
Created: 1
Created: 2
Me: 3 Parent: 0
Me: 3 Parent: 0
Created: 3
Me: 4 Parent: 0
Me: 4 Parent: 0
Created: 4
FirstUserTask: exiting
Me: 1 Parent: 0
Me: 2 Parent: 0
Me: 1 Parent: 0
Me: 2 Parent: 0
```

The first task created is `task_k1init`. Tasks 1 and 2 created by `task_k1init` have lower priority than `task_k1init`. Therefore `task_k1init` is able to print `Created: 1` and `Created: 2` without interruptions. However once `task_k1init` creates task 3, which has higher priority than `task_k1init`, task 3 takes over the CPU. Even after making syscalls or yielding, task 3 is still brought back to the front of the ready queue because it has the highest priority. Therefore it is able to complete execution and print `Me: 3 Parent: 0`, `Me: 3 Parent: 0`. Once task 3 finishes execution `task_k1init` once again has the highest priority. It is able to continue execution and print `Created: 3`. Once `task_k1init` creates task 4 which has a higher priority, the same thing that happened with task 3 happens again. Task 4 is able to complete execution and print `Me: 4 Parent: 0`, `Me: 4 Parent: 0`. Once task 4 is completed, it is no longer in the ready queue and `task_k1init` can continue and print `Created: 4` and `FirstUserTask: exiting`. At this point `task_k1init` has finished execution and therefore only tasks 1 and 2 are in the ready queue. Tasks 1 and 2 alternate execution as they have the same priority and are moved to the back of the ready queue during syscalls and yields. The ready queue behaves like a FIFO queue in this case since both tasks have the same priorities. This is why we see:

```
Me: 1 Parent: 0
Me: 2 Parent: 0
Me: 1 Parent: 0
Me: 2 Parent: 0
```

Finally, both tasks complete and the kernel has no more tasks in the ready queue. The while loop terminates and the kernel exits.

Creating your own user tasks

Since user task's have their lr's initialized to call Exit(), there is no need for users to explicitly include `Exit()` in their user tasks.