Assignment K4

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

git checkout k4

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`

To clean the build file:

```
cd ./build

make clean
```

Operating Instructions

To run the program, issue the following:

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

Program Description

(NEW) Trainserver

The trainserver employs a trainworker. The best way to illustrate this is with an example. Suppose the shell issues a `REVERSE` command to the trainserver. The trainserver then adds three train tasks to the scheduling min heap. The first train task created sets the train speed to 0, the second sets the train to go in reverse, the final sets the train speed back to its original speed. These three train tasks have incrementing times of planned execution. The trainserver uses a min heap based on time. The train worker task then reports for work to the trainserver via `Send`. If the train task at the top of the min heap is able to be performed (its scheduled time has passed), then the trainserver replies to the trainworker with scheduled task. The trainworker performs the task and reports back for more work.

(NEW) Uartserver

For Uart2, two separate servers were created. One for RX, one for TX (`uart2txserver`, `uart2rxserver`), each with a notifier. This design decision was chosen due to the terminal being full duplex and is computationally fast enough to handle RX and TX happening within close time distances of each other. Therefore there is no need for a state machine that is updated and read by RX and TX as opposed to UART1. Before talking more about the uart2txserver and uart2rxservers, lets talk about the notifiers.

The Uart2 tx notifier awaits for the `UART2_TX_HALF_EMPTY` event. In the syscall handler for `UART2_TX_HALF_EMPTY` await event, the interrupt for `UART2TXINTR` is turned on at the uart level. When the `UART2TXINTR` interrupt occurs, the interrupt is then turned off at the uart level as well. This is to simplify handling combined interrupts, as we can turn off `RTIEINTR` (receive timeout interrupt) specifically in the uart level.

The uart2 rx notifier awaits for the `UART2_RX_INCOMING` event. In the syscall handler for `UART2_RX_INCOMING` await event, the interrupt for `RXINTR`(rx buffer half full interrupt) and `RTIEINTR` are turned on at the uart level. When either `RXINTR` and `RTIEINTR` interrupts occur, the IRQ handler turns off both interrupts on the uart level. Not that there is no VIC interrupt disabling or enabling as turning off on the uart level avoids juggling with combined interrupts and VIC RX interrupts.

Now back to talking about uart2txserver and uart2rxservers. For uart2txserver, when a task calls `Putc`, it requests `SEND_CHAR` to the uart2txserver, the server first checks if its lock is being held. A lock is implemented to ensure that multiple tasks will not print to the same cursor. This is especially useful for printf. `Putc` implicitly acquires the lock if its not being held by any tasks. The lock is released by a `RELEASE_LOCK` request. If the task is holding the lock, the txserver will attempt to put the character in the `uartserver_request` struct into the uart2 tx buffer. If the buffer is not full, then the txserver will reply to the task, unblocking it. Otherwise the txserver will not reply to the task, and will place the character to be printed into a buffer. Likewise for tasks that do not own the lock, they will be placed into a buffer as well. When the notifier tasks notifies the uart2txserver that the transit FIFO is less than half full, the uart2txserver will place the buffered char that is requested by the task that owns the lock into the uart transit FIFO.

For uart2rxserver, it is a similar story. It is assumed that only one task would ever request from the uart2rxserver. In particular, the shell. When a task issues `Getc`, if the receive FIFO is not empty, then the task is replied with the character. Otherwise its taskid is stored, and it remains blocked. When the notifier task for uart2rxserver notifies the server, if a task is waiting, then it is replied with the character.

For Uart1, one server and two notifiers are used. This design was chosen as it is necessary to include a state machine that encompasses both rx and tx. The state machine counts specific interrupts caused by CTS and TXINTR. If the notifier receives a TXINTR and 2 CTS changes, then the uart1txserver is able to send to the track. Any tasks that try to send to the track via the uartserver when these conditions are not

met will be blocked. The receive FIFO is turned off for uart1, so when the notifier notifies the rxserver about a RXINTR, the rxserver is able to reply to tasks for `Getc`.

(NEW) Shell

The shell is a remake of A0.

The shell uses cursor addressing to display on its screen the following:

1. An elegant yet ergonomic banner
2. the current time
3. the idle percentage
4. a table of switch positions,
5. a list of the most recently triggered sensors (newest on the right, oldest on the left)
6. a prompt at which the user can type commands.

Clock notifier

The clock notifier is an infinite loop that awaits for a clock tick and then notifies the clock server that a tick has passed. A design decision was made to have the clock notifier be created by the clock server. This allows for the clock notifier to avoid looking up the clock server's tid via the nameserver. Instead, `MyParentTid()` was used.

Timer 1 was chosen to be the timer that fires interrupts. Since the load value is $508 * 10 = 5080$, this can fit in 16 bits. Since the clock rate is 508000hz, an interrupt is fired every 10ms.

Since the clock notifier never terminates, there will always be an event blocked task so the kernel will not terminate. Creating a proper shutdown was determined to be out of scope due to having to determine when all of the useful tasks have terminated and sending this signal to all the servers.

Clock server

Clockserver follows a similar pattern as the nameserver and rpsserver (See below). There are request and response structs that get serialized into char arrays when the kernel is copying messages. The char arrays are then converted back into structs to be used by other tasks. A linked list is used to implement `delay` and `delayuntil`. Firstly, all `delay` requests are converted into `delayuntil` requests by adding the current tick count by the `delay`'s offset. Each node contains the task tid, the `delayuntil` time, and a reference to the next node. A new `delayuntil` awaiting task is placed in sorted order in the linkedlist. This is to ensure that the node at the head of the list will have the closest `delayuntil` time to the current tick. When a tick happens, only check the nodes at the front of the linkedlist until we reach a node that has not yet exceeded `delayuntil`. That node and all nodes after it must not have reached their `delayuntil` ticks. The clockserver responds to the tasks whose `delayuntil` had reached the tick count, thereby unblocking them and allowing for them to continue execution.

Interrupts

The handler for IRQ is placed at `0x38`. When an interrupt happens, we enter the handler. We treat interrupts the same way we handle software interrupts. However the PC is temporarily set to be odd in order for a quick way to distinguish IRQ's from SWI's. The same context saving routine is called by IRQ as SWI. Upon return into the kernel (switch_user), check if the user task's stored PC is odd. If it is then that means that an

IRQ had happened. The PC is decremented again so that when returning, it goes to LR-4 as specified by the ARM documentation. The kernel then checks all interrupt sources and unblocks tasks that were waiting for those interrupts.

AwaitEvent

A design decision was made so that only at most one task can wait for a particular event. This allows for a quick array lookup in `event_mapping`, where the eventid is the index. When a task performs `AwaitEvent`, its pointer is placed in the corresponding entry in `event_mapping` and is no longer placed back in the ready queue. When a particular event happens, the task in the `event_mapping` that corresponds to the event is placed back into the ready queue.

Idle Task

The idle task awaits for the `BREAK_IDLE` event. This event type is handled differently than others. The kernel first gets the current time. Then the kernel halts the processor. When an interrupt occurs, the processor exits halt. The time is then taken again to obtain the elapsed halt time. This halt time is returned back to the idle task, which is used to calculate the halt time percentage. The interrupt that ended the halting period is not yet acknowledged, and will be acknowledged in the next kernel loop.

K3 Client Task Output Explained

```
TID: 5, Delay: 10, Completed: 1
Idle: 98% (98 ms)
TID: 5, Delay: 10, Completed: 2
Idle: 97% (195 ms)
TID: 6, Delay: 23, Completed: 1
TID: 5, Delay: 10, Completed: 3
Idle: 96% (290 ms)
TID: 7, Delay: 33, Completed: 1
TID: 5, Delay: 10, Completed: 4
Idle: 96% (385 ms)
TID: 6, Delay: 23, Completed: 2
TID: 5, Delay: 10, Completed: 5
Idle: 96% (480 ms)
TID: 5, Delay: 10, Completed: 6
Idle: 96% (577 ms)
TID: 7, Delay: 33, Completed: 2
TID: 6, Delay: 23, Completed: 3
TID: 5, Delay: 10, Completed: 7
Idle: 95% (671 ms)
TID: 8, Delay: 71, Completed: 1
TID: 5, Delay: 10, Completed: 8
Idle: 95% (766 ms)
TID: 5, Delay: 10, Completed: 9
Idle: 95% (863 ms)
TID: 6, Delay: 23, Completed: 4
TID: 7, Delay: 33, Completed: 3
TID: 5, Delay: 10, Completed: 10
Idle: 95% (956 ms)
TID: 5, Delay: 10, Completed: 11
```

```
Idle: 95% (1053 ms)
TID: 6, Delay: 23, Completed: 5
TID: 5, Delay: 10, Completed: 12
Idle: 95% (1148 ms)
TID: 5, Delay: 10, Completed: 13
Idle: 95% (1244 ms)
TID: 7, Delay: 33, Completed: 4
TID: 6, Delay: 23, Completed: 6
TID: 5, Delay: 10, Completed: 14
Idle: 95% (1338 ms)
TID: 8, Delay: 71, Completed: 2
TID: 5, Delay: 10, Completed: 15
Idle: 95% (1433 ms)
TID: 5, Delay: 10, Completed: 16
Idle: 95% (1529 ms)
TID: 6, Delay: 23, Completed: 7
TID: 7, Delay: 33, Completed: 5
TID: 5, Delay: 10, Completed: 17
Idle: 95% (1623 ms)
TID: 5, Delay: 10, Completed: 18
Idle: 95% (1719 ms)
TID: 6, Delay: 23, Completed: 8
TID: 5, Delay: 10, Completed: 19
Idle: 95% (1814 ms)
TID: 7, Delay: 33, Completed: 6
TID: 5, Delay: 10, Completed: 20
Idle: 95% (1909 ms)
TID: 6, Delay: 23, Completed: 9
Idle: 95% (2004 ms)
TID: 8, Delay: 71, Completed: 3
```

We have that the idle percentage is printed every 10 ticks. Removing this from the output gives:

```
TID: 5, Delay: 10, Completed: 1
TID: 5, Delay: 10, Completed: 2
TID: 6, Delay: 23, Completed: 1
TID: 5, Delay: 10, Completed: 3
TID: 7, Delay: 33, Completed: 1
TID: 5, Delay: 10, Completed: 4
TID: 6, Delay: 23, Completed: 2
TID: 5, Delay: 10, Completed: 5
TID: 5, Delay: 10, Completed: 6
TID: 7, Delay: 33, Completed: 2
TID: 6, Delay: 23, Completed: 3
TID: 5, Delay: 10, Completed: 7
TID: 8, Delay: 71, Completed: 1
TID: 5, Delay: 10, Completed: 8
TID: 5, Delay: 10, Completed: 9
TID: 6, Delay: 23, Completed: 4
TID: 7, Delay: 33, Completed: 3
TID: 5, Delay: 10, Completed: 10
TID: 5, Delay: 10, Completed: 11
```

```
TID: 6, Delay: 23, Completed: 5
TID: 5, Delay: 10, Completed: 12
TID: 5, Delay: 10, Completed: 13
TID: 7, Delay: 33, Completed: 4
TID: 6, Delay: 23, Completed: 6
TID: 5, Delay: 10, Completed: 14
TID: 8, Delay: 71, Completed: 2
TID: 5, Delay: 10, Completed: 15
TID: 5, Delay: 10, Completed: 16
TID: 6, Delay: 23, Completed: 7
TID: 7, Delay: 33, Completed: 5
TID: 5, Delay: 10, Completed: 17
TID: 5, Delay: 10, Completed: 18
TID: 6, Delay: 23, Completed: 8
TID: 5, Delay: 10, Completed: 19
TID: 7, Delay: 33, Completed: 6
TID: 5, Delay: 10, Completed: 20
TID: 6, Delay: 23, Completed: 9
TID: 8, Delay: 71, Completed: 3
```

While the tasks all start at the same time, they all have different delay intervals. Whenever a task finishes its delay, it will output a line before starting the next delay or exiting. In this case, we have that all tasks managed to start on time and finish in their tick so we did not have to worry about priority or multiple tasks being runnable at the same time. We have that the output from different tasks are interleaved as they become blocked when starting their delay and only become unblocked after the time passes. While all tasks are blocked, we have that the idle task will be able to run and halt the CPU.

Kernel Changes for Interrupts

Due to forward planning, most of the kernel was already ready for interrupts. Only some minor changes to the existing structure were needed.

Startup:

- The startup process now explicitly sets a 64 byte stack for interrupts. Only 8 bytes are currently used to temporary store 2 registers.
- The VIC is also now configured for interrupts.

User Tasks:

- Interrupts are now enabled in the CPSR.

Kernel Structures:

- The timer api was refactored to not require timer objects. Multiple timers are now used.
- There is an event mapping from eventids to tasks.
- We now keep track of the number of event blocked tasks and use it to decide when to exit the kernel. We also now directly check the number of runnable tasks to exclude the idle task.
- We handle clearing interrupts as part of the run loop.

- We now support AwaitEvent for updating the event mapping. There is also special support for halting the system until the next interrupt.

Send Receive 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`.

Scenario 2

B first calls `Receive`. It sees if any other TCBs wish to send to it. Since there are none yet, 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. When A calls `Send`, it will find B and see it is in the `RECEIVE` state. We will then call `handle_send`.

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.
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 another task executes `Reply` to A, `msg_copy` is executed to copy that task's message back to A and then both tasks get placed back onto the ready queue.

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 modulo 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 of the response, and so it will 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 initiated this request would also know to look in the body to obtain the returned task_id.

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 facilitate 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).

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.