CS 452 Kernel Documentation

Xinxin Lin, 20480200

Yang Li, 20461548

Operating Instruction

The executable file can be found under /u/cs452/tftp/ARM/xx2lin which is named kernel.elf. To run the program, issue the following commands in RedBoot:

1. Load –b 0x00218000 –h 10.15.167.5 “ARM/xx2lin/kernel.elf”
2. Go

The kernel supports the following kernel primitives operating:

- int Send(int tid, char *msg, int msglen, char *reply, int replylen)
- int Receive(int *tid, char *msg, int msglen)
- int Reply(int tid, char *reply, int replylen)
- int RegisterAs(char *name)
- int WhoIs(char *name)
- int Create(int priority, void (*code) ())
- int MyTid(void)
- int MyParentTid(void)
- void Pass(void)

- void Exit(void)
- int AwaitEvent(int eventid)
- int Delay(int ticks)
- int Time(void)
- int DelayUntil(int ticks)
- int Putc(void)
- int Getc(void)
- int IdleTime(void) – return the amount of time in 0.01 ms that idle task has been executed
- void kExit(void) – exit the kernel

Important Data Structure

1. Kernel

- One array to store the head and tail of each priority queue. There are 32 priorities (0 – 31) so the size of this array is 32.
- One array to store task descriptors structure. Since we can not use heap, this array will be constructed in main function and elements will be distributed to initialize or create function as pointer to store or modify information about task descriptor. This array has size of 88. In later assignment, we will build a mechanism to reuse resources when task exit so this size should be enough.

- One linked list for each priority. Linked list maintains a FIFO order, that means in the same priority level, head task will be scheduled earlier than others.
- One fixed size array to store state of switch set.
- No special and fancy algorithm is used when writing kernel 1 since the sizes of all data structures are fixed, even the size of all linked lists are bounded by 88, so linear search will be accomplished in constant time. We think this is fairly enough for kernel 1.

2. Task Communication

- Message queue for every possible task. A one dimensional array is used to represent a two dimensional array. Its size is (88 * 87), since the highest possible number of tasks for the kernel is 88.
- Receive buffer and reply buffer. They are both one dimensional array of size 88. Receive buffer is used for when receive occurs and no messages are queued. The reply buffer is initialized when a task sends a message and filled by the receiving task.
- Linked list for registered server information in name server task. The reason we choose linked list is that we want name server has a better performance in linear searching task id to respond clients' query.

3. Hardware Interrupt

- Linked list for data structure that stores blocked task's information which are task id and type of event. Loop through entire linked list to unblock all tasks that are blocked on the corresponding event. There is another choice to create one linked list for each type of event, so that on triggered interrupt, kernel only need to loop through the linked list of that type

of event to unblocks tasks. Since right now there is only one type of event, we chose the first solution.

- A waiting queue is used. All the delayed tasks are stored in this queue in ascending order. The insertion algorithm is just finding the correct spot in terms of order, and then store the tid, delayed ticks pair in the correct position. The efficiency is linear time; however, there are only at most 88 tasks, so the performance should be fine. Moreover, this structure allows constant time checking for unblocking tasks. Overall, the performance should benefit from having a sorted queue.

4. Serial I/O

- One circular buffer for each I/O server. It is used to store charaters from input or output.
- One circular buffer for each get servers to store the task id of blocked user task.

Easy to reuse resources. Access is in constant time since always accessing the head of buffer.

Kernel Structure

1. Forever Loop

Each iteration of the forever loop:

1. schedule() will return the pointer of next ready task with highest priority. If the pointer is NULL, break the loop.
2. activate() will push return value to user stack, do context switch from kernel to user to execute user function and get user request.

3. handle() will handle user request and put the result in the return value field of task descriptor.

2. Context Switch

Initialization:

Initialize() is stored in include/taskDescriptor.h. It initializes first task descriptor, and add it onto the ready queue. It also initializes first task's stack with user entry point and a stack pointer. After this, it sets up swi jump table. Namely, stored the address of the label __SWI_HANDLER to 0x28. This way, when swi is called, our handler can be triggered correctly. Now that hardware interrupt is implemented, hwi jump table is also set up. Namely, store the address of the label__SWI_HANDLER to 0x38. In addition, necessary bits for enable timer3 and hardware interrupt for times is set.

Context Switch for hardware and software interrupt:

The main context switch function is in include/kernelFunction.h. The first half of Activate() is responsible for doing the context switch from kernel to user space. It saves kernel's r4- r9 and frame pointer, since there is no function call in activate(). Also, every time user function returns to kernel, this is the only kernel to go to. Next, change to system mode to change the SP register to the one stored in the active task's task descriptor. After the SP register is set, pop the saved user SP and LR. Use ip as a temporary variable for the LR on the user's stack. Immediately return to supervisor mode and assign IP to SP. This basically does the following: lr_svc = lr_usr. Since hardware interrupt needs to be handled, r0 - r12 and lr needs to be popped as well.

r0 is popped by assigning return value to r0. The remaining steps are to store the return value in the active task descriptor into r0 and movs pc,lr. Notice that movs will not only assign the link register to the program counter, it also installs saved program status register to the current program status register so that mode is switched from supervisor to user.

System calls are in include/syscall.h. When user does a system call, parameters are stored in r1 and r2. Most importantly, r4-r9 and frame pointer are pushed onto the stack. This is essential because there are multiple user tasks, so that work registers are likely to be used. The next step is to cal “swi #x” where x indicates the which system called was triggered.

When a hardware interrupt occurs, context switch will also happen. Same handler will be used. In Initialize(), the address of __SWI_HANDLER label is stored into 0x28 and 0x38. Therefore, when swi is called or hardware interrupt is triggered, program counter is set to the second half of Activate(). SWI will change the mode from user to supervisor mode, but hardware interrupt will lead to interrupt mode. The first thing that the interrupt handler will do is to push r0 to the stack, and assign cpsr to r0. Next compare r0 with 0xd2 to determine if the current mode is interrupt mode or supervisor mode. Before doing anything, pop r0 back, and push all the user’s registers(r1 - r12, lr) after switching to system mode. R0 is stored into task descriptor return value. If it is interrupt mode, then lr_irq needs to be popped from IRQ mode and overwrite the one in supervisor mode. Afterwards, pop frame pointer from the kernel’s stack; otherwise, all the variables are unusable. The next thing is to store arguments into request’s corresponding fields. The immediate value in swi instruction is also retrieved by using “ldr x, [lr, #-4]”. The next step is to acquire link register in supervisor mode, and then use it to overwrite the link register in system mode. Namely, this does lr_usr = lr_svc. SWI instruction changed lr_svc to the following instruction of swi in the user space. After overwriting the link register in user

mode, push user's registers onto user's stack and acquired the new stack pointer. After that, return to supervisor mode and store SPSR and the new stack pointer into the task descriptor. Now we need to restore kernel previous registers r4-r9. Since frame pointer is correctly stored and restored, activate() is able to return to the kernel's main loop and execute the next instruction in the kernel's main loop.

3. Task Communication

Send:

If the target task is not receive blocked, then the message is queued to its corresponding buffer. The task is also send blocked. Otherwise, write to the receiver's buffer and make the task reply blocked. The receiving task's state is also changed from receive blocked to ready. Also the reply buffer of its own should be set up under both circumstances.

Receive:

If there are messages queued in this task's message queue, then directly fetch from the task's queue, and change the sender task from send blocked to reply blocked. Otherwise set up its receive buffer, and change its own state to be receive blocked.

Reply:

First check if the target task is reply blocked. If no, then return the error code. Otherwise, the reply buffer should be ready to use. The message is going to be copied from the receiver's

message buffer to the sender's reply buffer. At the end, change the sender's state from reply blocked to ready.

Name server:

Name server has a constant task id of 1 and is default to every user task. Name server services two requests: RegisterAs() and WhoIs(), which are wrapper functions for Send(), Receive() and Reply(). For RegisterAs(), it currently can store information for up to 20 servers. For WhoIs(), it will do linear search in linked list to find task id of correspond query if exists. This will not be an overhead because usually this will be called only once at the beginning each client task.

4. Hardware Interrupt

Time:

Immediately reply with the current time elapsed.

Delay:

Add the task to the wait queue, and set the time_counter to timeElapsed + ticks.

DelayUntil:

Add the task to the wait queue, and set the time_counter to ticks.

clock notifying: Increment timeElapsed, and check if any task's time counter is up. If a task has reached its waiting time, un-block the task by replying to the task and remove the task from

the queue. Since the queue is sorted, unblock + remove will only require one traverse of the queue.

Clock Server:

Try to receive a request, which is a structure including request type and an optional parameter. The request type are: Time(), Delay(), DelayUntil() and clock notifying.

Idle Task:

Idle task only contains. To calculate the percentage of the execution of idle task, we uses two counters, the number of hardware interrupt occurs and the sum of time executing idle task. For each loop in main kernel forever loop, if the request type indicates that a hardware interrupt occurs, we increment the counter for hardware interrupt. If the scheduled active task has a priority of 31, which has to the the idle task, we will set the load of timer 2 before it is activated. After activation, we read value register of timer and compute the time that it executes in millisecond. When all tasks other than servers and idle tasks have exited, we compute the fraction of two counter and print it out.

5. Serial I/O

Output server to COM1 and COM2:

One server for each communication channel. Output server will block on Receive(). If sender is notifier and there is character to be printed, notifier will be replyed the character, otherwise notifier's id is store. If sender is user task and there is notifier waiting, notifier will be replyed

the provided character from user task and user task will be replayed success, otherwise only user task will be replyed since Putc() is non-blocking.

Input server to COM1 and COM2:

One server for each communication channel. Input server will block on Receive().If sender is notifier and there is user task waiting, user task will be replyed the provided character from notifier and notifier will be replayed success, otherwise only notifier will be replyed since Gutc() is blocking. If sender is user task and there is charaters to be picked up, user task will be replyed the first unpicked character otherwise user task's id will be buffered and user task will be blocked.

Output notifier to COM1:

One notifier for COM1 input. Notifier blocks on xmt1 event. Interrupt for TIS register is enabled when DCTS and CTS register is asserted (in interrupt handler) when any other interrupts occurs. Event occurs when TIS register of COM1 is asserted. In interrupt handler, interrupt of COM1 transmit is turned off. Then notifier will send a request to COM1 output server. After returning from Send(), notifier write character in reply to COM1 data register.

Output notifier to COM2:

One notifier for COM2 input. Notifier blocks on xmt2 event. Interrupt for TIS register is enabled when AwaitEvent() is called on xmt2. Event occurs when TIS register of COM2 is asserted. In interrupt handler, interrupt of COM2 transmit is turned off. Then notifier will send

a request to COM2 output server. After returning from Send(), notifier write character in reply to COM2 data register.

Input notifier to COM1:

One notifier for COM1 input. Interrupt is enabled in kernel initialization. Notifier blocks on rcv1 event. Event occurs when RIS register of COM1 is asserted. AwaitEvent() will return the character from COM1. Then notifier will send the charater to COM1 input server.

Input notifier to COM2:

One notifier for COM2 input. Interrupt is enabled in kernel initialization. Notifier blocks on rcv2 event. Event occurs when RIS register of COM2 is asserted. AwaitEvent() will return the character from COM2. Then notifier will send the character to COM2 input server.

Courier:

One courier for each communication channel. Courier simply receives string from user task that calls Printf(), then it will call Putc() to print character one by one. The purpose of courier is to avoid messing up output by interleaving call to Putc().

Choice of priorities:

Task name	Priorities
COM1GetServer()	4
COM1PutServer()	8
COM2GetServer()	28

COM2PutServer()	26
COM1GetNotifier()	3
COM1PutNotifier()	7
COM2GetNotifier()	27
COM2PutNotifier()	25
COM1Courier()	11
COM2Courier()	27

First of all, COM2 are not that crititcal, so they are put after everything.

COM1 sensor data receiving and train command sending are very important, so their priorities are very high.

At the same time, Get server should be in front of Put server.

Train Application

Train application consists of four user tasks.

1. init(void) – Initialize user interface and switches. After initialization completed, create the following two tasks.
2. timer(void) – Prints the amount of time that train application has been executed.
3. sensorData(void) – Request sensor data from train set and print most recent triggered sensor.

4. trainCommunication(void) – Parse user input command from keyboard and send command to train set.

Supports the following commands:

- tr <train_number> <train_speed>
    - set any train in motion at the desired speed (0 for stop)
- rv <train_number>
    - reverse the direction of train
- sw <switch_number> <switch_direction>
    - set the given switch to straight(S) or curved(C)
- q
    - halt the kernel and return to RedBoot

Choice of priorities:

Task name	Priorities
init()	29
timer()	17
sensorData()	16
trainCommunication()	17

Init() is 29 because it needs to be run after all the servers and notifiers are initialized.

Timer() is 17 because it is a user level task, and it is more important than COM2 IO servers.

sensorData() is 16 because it is fetching sensor data quickly, so high priority is chosen.

trainCommunication() is 17 because it is a user level task, and it is more important than COM2 IO servers.

Known Bugs

1. After kernel exits, device must be reseted in order to run kernel again.
2. By some chances, the output of sensor data will stop printing. As far as we know, the task that prints sensor data is stucked in sending request to train set.

Source Code

Url: https://git.uwaterloo.ca/xx2lin/cs452-kernel

Tag: k4

Commit: 8b36b5a946179d13dd14c9874c425a65d012478b

User name: xx2lin

Student id: 20480200