Complete Kernel Report

CS 652

Real Time Programming

Alexandr Murashkin {amurashk} {20465003}

Priyaa Varshinee {pvsriniv} {20456492}

University of Waterloo

Table of Contents

Chapter 1. ZX_KERNEL_OVERVIEW 7

I. Introduction 7

Chapter 2. TASK MANAGEMENT 8

I. Kernel coarse-grain structure overview 8

II. Kernel initialization 8

Resetting devices and interrupts 8

Pre-configuration of devices 9

Installing interrupt handlers 9

Initializing the ready queues 9

Creating the first user task 9

Creating Idle Task 9

III. API and System Calls 10

Standard 10

Additional 10

IV. Constants 11

V. Internal structure 11

Kernel main loop 11

Cycling through the tasks in priority queue (zx_get_next_request) 12

Handling System Call requests (zx_handle) 12

VI. Context switching 13

Step 1. Restoring the context – from kernel to a user task – zx_restore_the_context 13

Step 2. User Tasks execution, System calls and Hardware interrupts 13

Step 3. Transition from user task to the kernel entry 14

VII. Algorithms and data structures 16
Mode switching algorithms 16
Task descriptor 17
Task States 17
Request Structure 17
Queue and scheduler 18
Kernel Termination 18
Chapter 3. RESOURCE AND MEMORY MANAGEMENT 19
I. API and System Calls 19
Standard 19
Additional 19
II. Memory allocation and stack 19
III. Heap Memory / Dynamic Memory Allocation 20
Implementation 20
Chapter 4. INTER-PROCESS COMMUNICATION 21
I. API and System Calls 21
Standard 21
Additional 22
II. Constants 22
III. Servers associated 22
IV. Internal structure and implementation 22
V. Data structures and algorithms 23
Receive Queue 23
Chapter 5. TIME KEEPING 24
I. API and System Calls 24

Standard 24
Additional 24
II. Constants 24
III. Associated servers 24
Chapter 6. INPUT-OUTPUT FUNCTIONS 26
I. API and System Calls 26
Standard 26
Additional 26
II. Constants 26
III. Associated servers 26
IV. Implementation 26
Chapter 7. SERVERS, NOTIFIERS AND IDLE TASK 27
I. Name server 27
Implementation Details 27
Data Structure and Algorithms: Self-Balancing Binary Search Tree (BST) for Name Server 28
Design Decisions: Hash Function for the name server 29
II. Clock server and notifier 29
Implementation details 29
Data structure and algorithms – MIN heap 30
III. UART Servers 30
Internal work principles of UART1_TX_SERVER and UART2_TX_SERVER 31
Internal work principles of UART1_RX_SERVER and UART2_RX_SERVER 31
IV. Server and notifier priorities 32
V. Idle task and profiling 33

Chapter 8. ENABLING INTERRUPTS AND PER INTERRUPT HANDLING 34
I. How are the interrupts enabled 34
II. Interrupt processing and priority 34
III. Timer interrupts 34
Interrupt Workflow Diagram - Timer 35
IV. UART Interrupts 36
Transmit Interrupts 36
Receive Interrupts 36
Interrupt Workflow Diagram - UART 36
Curious Case of UART1 Transmit Interrupt 37
Chapter 9. DEBUGGING AND ERROR HANDLING 38
Chapter 10. ZX_KERNEL FREEBIES 40
I. Pseudo-Random Numbers Generator 40
II. Send Function Timing Measurements and Analysis 40
Chapter 11. ZX_KERNEL CONSTANTS AND CONFIGURATION 42
I. Configurable Values 42
II. Non-Configurable Values 43
Chapter 12. ASSIGNMENT 0 IMPLEMENTATION 44
I. Overview 44
II. Tasks 44
1. Timer Task 44
2. Sensor Task 44
3. Command Task 44
III. Assignment Questions: 44
1. How do we manage clock updates and still not lose ticks? 44

2. What is the delay observed in sensor data received once the controller has been queried for data? 45

Chapter 13. README 46

Chapter 14. FILES AND HASH VALUES 47

Chapter 1. ZX_KERNEL_OVERVIEW

Introduction

ZX (pronounced as "*zox*") kernel is a real time microkernel which provides a basic framework for running a task on TS7200 board with ARM processor. The kernel runs on top of the red boot. Like any other real time kernel, the core functionalities include Task Management, Memory Management, Inter-process Communication Mechanism and Time Keeping. This section gives a brief overview of how ZX_KERNEL provides each of these functionalities to the user task by means of system calls. The complete list of system calls can be found at the end of this section. The internal design of the kernel is presented in the next section. ZX_KERNEL includes some additional system calls apart from the ones mentioned in kernel specification (link below).

(http://www.cgl.uwaterloo.ca/~wmcowan/teaching/cs452/pdf/kernel.pdf).

Throughout the text, we try not to include duplicates from the kernel specification. We don't describe the return values of system calls since the document above does it. We focus more on implementation and key special features.

The kernel is named after the two random ASCII values (which turned out to be *Z* and *X*) chosen by each member of the team who created the kernel.

Chapter 2. TASK MANAGEMENT

Kernel coarse-grain structure overview

The kernel structure follows the recommended by the course instructor structure.

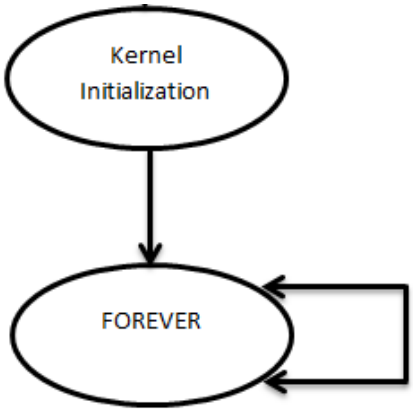

Fig 1.1. Kernel Overview

Kernel initialization (zx_kernel_init) includes:

1) Resetting devices and interrupts
2) Pre-configuration of devices
3) Installation of interrupt handlers
4) Initialization of task descriptors
5) Initialization of important memory locations
6) Initialization of tasks ready queues
7) Creation of idle task and first user task

FOREVER loop consists of (coarse view):

1) Getting next request
2) Handling next request
3) Condition of absence of requests
4) Returning to the RedBoot

Kernel initialization

Resetting devices and interrupts

The kernel disables all the interrupts on VIC in order to prevent the appearance of non-desired interrupts. The kernel disables all the timers as well. Then, it enables only the desired interrupts on VIC, however, does not configure them.

Pre-configuration of devices

The kernel initializes both UART1 and UART2 by turning FIFOs off, initialization of the baud rate and other parameters to make UARTs ready-for-use.

Installing interrupt handlers

As per our instructor's advice, which turned out to be very helpful, we use the same handler to catch both hardware and software interrupts.

- For SWI (Software Interrupts)

We rewrite location 0x08, where the code is executed when SWI interrupt is initiated. We put there a new instruction - Branch (B). Since the branching instruction supports 24-word offsets, it is enough to fit the starting address of the SWI handler. We take the absolute offset of SWI handler label, transform it (subtract vector table offset and prefetched delta of program counter, and then right shift to get the word offset), and add it to the instruction code 0xEA000000.

The Handler itself is embedded into the initialization function surrounded by starting and ending labels, but the compiler thinks that the code is not used, and does not compile it. To disable such type of optimization, we add a dummy condition (program argument count is less than -1) which always evaluates to false, but makes the compiler not to optimize any code.

- For HWI (Hardware Interrupts)

We put the same branching link instruction code (adjusted by the offset of the jump table entry, of course) to 0x18. Then, we prepended our Generic handler with the code specific for routing - whether we have come from a HWI or SWI. The generic handler in turn stores a flag onto the stack, so that in our kernel we can easy check what happened before the kernel entry.

Initializing the ready queues

Here all we need to do is to set the head pointers to NULL that means all queues are empty. Also we put a new next task id to the memory.

Creating the first user task

A new task of the one set up in the kernel configuration is created and appended to the ready queue.

Creating Idle Task

See Idle task section for reference.

API and System Calls

Standard

/* Create a task */ **int** Create (**int** priority, void *code()) ;
/* Pause Execution of the calling task */ **void** Pass();
/* Exit from executing task */ **void** Exit();
/* Get Task ID of calling Task */ **int** MyTid();
/* Get Task ID of the parent */ **int** MyParentTid();

Additional

void Shutdown(); Shutdown system call: 1) Shuts down all the server programs of the kernel. 2) Clears and turns off all the interrupts. 3) Kills the idle task because the kernel will be running only as long it has a task to execute. 3) Saves the profiling information to pre-defined memory locations. This includes percentage of time the idle task has run, time elapsed since kernel began running. 4) If all tasks are exited, returns to the RedBoot. The kernel can be terminated and exited to RedBoot by calling Shutdown()system call from one of the user tasks.
void Kill(**int** tid); Kill system call: 1) Frees the task ID held by the task. 2) Changes task state to ZOMBIE.

3) Removes the task from ready queue if it is in ready state.

Before or immediately after the Shutdown system call, all the user tasks created and running should be either terminated using Exit system call or killed using the Kill system call. tid is the TID of the task that has to be killed.

Constants

INVALID_TASK_ID – a placeholder for all invalid task ids
FIRST_USER_TASK_PRIORITY – a priority assigned to the first user task
ERR_CANNOT_CREATE_TASK – an error value indicating the system cannot create a task

Internal structure

Kernel main loop

Kernel main loop runs infinitely, getting next request from the user and serving them sequentially. In the get_next_request, the next task to be run is read from the priority queue. zx_resume_context() resumes the task that is supposed to be run currently and switches the system to user mode. The user mode task generates software interrupt by calling one of the system calls. Otherwise the task can be interrupted by one of the hardware interrupts. The interrupt handler saves the context of the task which generated SWI/interrupted by Hardware Interrupt (HWI) and restores the kernel context. The program gets back to kernel's main loop after this where SWI/HWI request is handled.

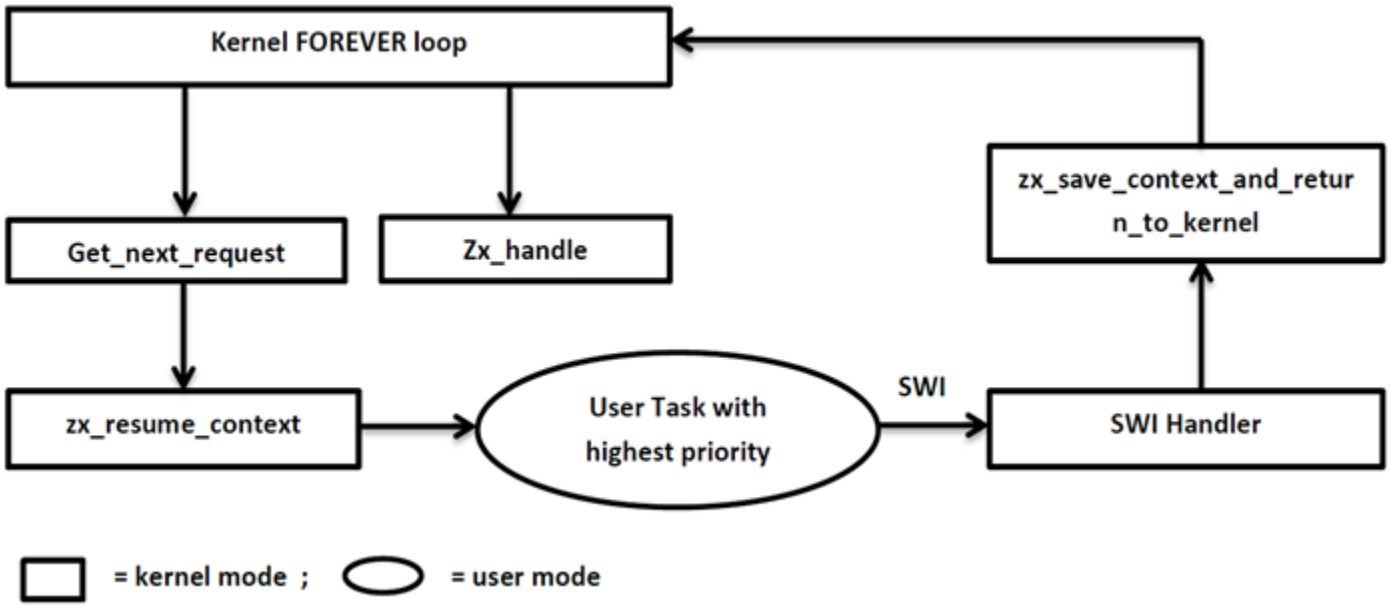

Fig 1.2: Kernel FOREVER loop

At the loop body, zx_get_next_request is called, which returns the pointer to the request structure. Next, the zx_handler is called that takes in this request pointer and processes it. This processes repeats until the request pointer is evaluating to NULL, which can mean that there are no ready tasks anymore to process, or, for the future, if some error occurs.

Cycling through the tasks in priority queue (zx_get_next_request)

The function zx_get_next_request calls the get_next_task_id that returns the task id that should be run right now according to the priority (or -1 if there are no more ready tasks and has to exit). Then, zx_restore_the_context is called with this task id that actually "pauses" the kernel and switches the context to the task with this task id.

Handling System Call requests (zx_handle)

Once the kernel receives the request structure from zx_get_next_request function, it calls zx_handle to handle the request. We treat both hardware and software interrupts in a similar manner, distinguishing all of them by the *request number*.

Below are the steps followed by the request handling routine:

- Check the *request number* to see if it is Hardware or Software Interrupt.
- If Software Interrupt, depending on the swi_number call the corresponding handler routine specific to the request.
- If system call should return a value to a caller task, update the return value of the Task descriptor (of the task which raised the request) accordingly.
- If the request number corresponds to hardware interrupt, then a wrapper function is called to service the request. The device number corresponding to the source of the interrupt is passed as parameter.
- Call the specific interrupt handler from the wrapper by examining the device number.

Context switching

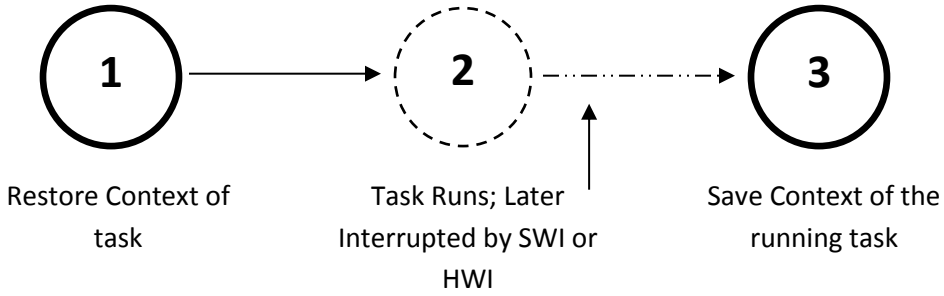

Fig 1.3: Coarse-grain view of context switching

Step 1. Restoring the context – from kernel to a user task – zx_restore_the_context

This function zx_restore_the_context sets the passed task's tid as the active task id. So now the task with tid becomes the active task, and we need to switch to this active task from the kernel.

But first, we need to store the state of the kernel. This is done by pushing all the general purpose registers onto the kernel stack.

Next, we get the *Task Descriptor* structure of the current task (the one we are switching to is now the current one). Next, we push the task's *return value*, *SPSR* into the kernel stack as well as backup the value of the task's stack pointer. Then we are switching to the system mode, restore the user's stack pointer, load all the general-purpose registers and *LR* of the task, as well as backup *LR*. Next, we are switching to the kernel (supervisor) mode. Now we pop the value of task's *CPSR* and store it to the *SPSR*, so that it can be restored quickly afterwards. We pop the task's *return value* and put it to *r0*, according to the argument and return value passing conventions. Finally, we restore the backed up *LR* and transfer the execution flow to the task, at the same time switching to the user mode (using MOVS command).

For software interrupts, we do not restore the scratch registers. However for hardware interrupts, we need to restore the scratch registers too. This makes things harder. Because once one we restore registers, we cannot use them anymore inside the kernel. So the decision was the following: switch to one mode using registers r4-r12, load r0-r3, switch back using r4-r12, store r0-r3 to the kernel's stack. After, switch back to the system mode, load r4-r12 and don't use them anymore. So, we still can use scratch registers, because at the end of the routine we can restore them from the kernel's stack.

Step 2. User Tasks execution, System calls and Hardware interrupts

User tasks are running until they are interrupted by either software interrupts (system calls) or hardware interrupts (generated by a device or peripheral).

Each system call function generates a software interrupt using SWI command. If the system call has a return value, then it is processed in the following pattern:

```
asm("mov %[v], r0" : [v]"=r"(result));/* getting the result from r0 */
return result; /* return the result */
```

This makes extra movements like: mov r3, r0 / mov r0, r3, but it makes the compiler happy, and there are no warnings at all.

Hardware interrupts can occur at any point of the task.

Step 3. Transition from user task to the kernel entry

Step 3a. Interrupt Handler

The handler is written in a pure assembly language. The purpose of this handler is to save user's registers to kernel stack and get ready for entering the kernel. It consists of the following sections:

Generic entry

First, to know at the beginning of the handler routine, whether HWI or SWI has just been generated, we just check the mode using CPSR register. If it has supervisor's mask (the last 4 bits are 0011), then go to the SWI specific code, else remain executing the following instructions. Eventually, both branches converge, and the generic code is executed. This is because the ARM processor of TS7200 board provides banked registers for each mode, which means that there is a separate set CPSR, SPSR, Stack pointer, Link Registers for each mode. The banked registers of one mode can't be accessed from another mode. If we reached the handler because of a hardware interrupt, we will be in IRQ mode. So before reaching the generic handler, we should copy the values in IRQ registers to Supervisor registers.

HWI-specific handler

As mentioned in the previous paragraph, in HWI-specific subroutine (handler) we have another stack pointer - the stack pointer of IRQ node. But, once we enter the handler, we store scratch registers and link register to work with. Therefore, in case of IRQ mode, we have to copy the registers from IRQ stack to the kernel's stack to make the kernel look the same when two branches converge. This is what we have done: just copy from IRQ's stack and put everything to kernel's stack. Also, SPSR of the kernel has to be set to the SPSR of the IRQ, we do this as well. And of course, we decrease LR by four, because in case of HWI, the LR will point one instruction forward than we need. Finally, just switch the mode to the kernel's one and that's it. However, the first thing we copy to the stack is the flag that says that we have come from IRQ mode.

SWI-specific handler

SWI Handler subroutine is very simple: just put at the particular place of the stack (the most bottom one, so that it will be easily accessible when pop back all the registers) the flag, which says that we have come from the SWI. That's it, then the branch converges, and the handler is executed as we never have hardware interrupts at all!

Generic handler

First, it saves all the arguments and *LR* (which is PC+4 of the previous task in case of SWI and PC+8 in case of HWI) to the kernel's stack and backs up the *LR*. Next, it switches to the system mode, where it stores task's general-purpose registers as well as backed up *LR*, and backs up the task's stack pointer *SP*. Then, it switches back to the kernel (supervisor) mode, stores the backed up task's stack pointer *SP*, *SPSR* that is actually the *CPSR* of the task. Finally, the handler ends with the jumping to the **zx_save_context_and_return_to_kernel** function using the **B** branching command.

In general, in zx_save_context_and_return_to_kernel routine, we can check whether we have come from HWI or SWI by looking up the stack and checking what the flag we stored there is.

Step 3b. Saving context and returning to the kernel (zx_save_context_and_return_to_kernel)

This function copies user's registers from the kernel stack and saves them to user task's stack and returns to kernel main loop.

First, it loads the stored on the stack values of task's *CPSR*, stack pointer *SP* to general-purpose registers as well as pops the arguments (*R0-R3*) and *LR* from the stack. This is done using *IP*, since *IP* is a saved version of the *SP*, which is modified automatically at the beginning of the function. Then, it saves everything to the corresponding local variables. The assembly code ends here, and now all the values are ready to work with.

Values of *CPSR* and *SP* are stored to the current task's descriptor.

Then, the request structure is filled in with the arguments and stored into the memory. For this, we follow the below steps.

- Find the type of interrupt (hardware or software interrupt) by reading the information pushed into the kernel stack by the handler.
- If hardware interrupt, set appropriate request_number in the request structure. Find the source of interrupt (ID of the device which generated interrupt) and add it to the request structure. Disable interrupts from the device by enabling an appropriate bit of the INTR_CLEAR_REGISTER. Clear the interrupt line of the device.

Further, assembly block begins. We need to restore the kernel's stack to make it look exactly as it was before switching to the task in **zx_restore_context** function. In case of the hardware interrupts, we need to make the kernel store the registers r0-r3 too. We did it in the following way: in context saving, it was quite easy: we just copied user's stack pointer to the register ip (of course, backing it up first!). Then, we just use both stacks: kernel's one pointed by sp, and user's one pointed by ip, so that we can store something on one stack and put it to another very easily, including r0-r3. After restoring general purpose registers, we need to restore the status registers.

Since the compiler automatically adds assembly instructions at the beginning and the end of the function code, we need to take it into account, and do exactly which is done by the compiler. That's why we put the following structures there:

```
sub     sp, fp, #12

ldmfd   sp, {fp, sp, pc}
```

So, first, we restore *sp* and *fp* by cancelling the actions that are done at the beginning of the current function. Next, we restore the stack by removing the arguments, *CPSR* and task's *SP* we put there before (we haven't popped them up yet since we got them using *IP*). Then, we pop all the general purpose registers and restore the kernel state. And finally, we cancel the effect of **zx_restore_context** function by executing the assembly instructions similar to the above ones, and get into the **zx_get_next_request** function as we have just returned from the **zx_restore_context** function**.** Then we can guarantee that the stack and frame pointers are fully restored, and the switching cycle to the kernel entry is done successfully.

Algorithms and data structures

Mode switching algorithms

1) Switch from kernel (supervisor's) mode to the system mode:

We indirectly add the mask **0x0000000f** to the *CPSR* using ORR. So that last four bits will be on, and the fifth is on in any mode.

2) Switch from system mode to the kernel (supervisor's) mode:

We indirectly AND the *CPSR* with **0xfffffff0** to clear the last four bits, and then ORR it with 3, since the supervisor's mode code is **10011**.

3) Switch from the supervisor's mode to the user mode:

We use the MOVS command and assign *SPSR* in advance to the saved user *CPSR*. After the execution of MOVS the mode will be switched to the user mode. The default user

CPSR is set to be 0x00000010, since it contains the code of the user mode, and has everything else off.

4) Switch from IRQ to Supervisor mode

We indirectly AND the *CPSR* with **0xffffff0** to clear the last four bits, and then ORR it with 3, since the supervisor's mode code is **10011**.

Task descriptor

The task descriptor structure follows the one presented in the kernel specification. We store task stack base (the beginning, initial value) and stack limit as well for extra information.

The implemented the task id system defines task id to contain two parts: index of task descriptor in the array of task descriptors, and the generation number, which is not zero if the task descriptor has been reused after task exited. Moreover, now the kernel has 64-bit variable that reflects the status of task descriptor id usage. We support 64 tasks. The bit corresponding to each task id is set to 0 if the task descriptor id is being used now, and is set to 1 if the task has exited, and the task descriptor id is available now to reassign. The kernel, when needs to create a new task, calculates the number of trailing zeros in the 64-bit variable, and reuses the task descriptor id with index that equals this number of trailing zeros. It means that the task descriptor ids that are used will be packed together as well, which is good. The algorithm is constant-time (log 64), because the number of trailing zeros is calculated using binary search and logical operations. To sum it up, we allow up to 64 tasks to be run at the same time.

Task States

At any point in time, a task in the system will be in one of the following states:

1. **TASK_STATE_READY** - Waiting in the ready queue to be scheduled by the CPU

2. **TASK_STATE_ACTIVE** - Currently running

3. **TASK_STATE_ZOMBIE** - Task has terminated

4. **TASK_STATE_RECEIVE_BLOCKED** – Task is waiting for other task that has to call receive

5. **TASK_STATE_SEND_BLOCKED** Task is waiting for other task that has to send data

6. **TASK_STATE_REPLY_BLOCKED** Task is waiting for other task that has to reply to the message

7. **TASK_STATE_EVENT_BLOCKED** Task is waiting for event to happen

Request Structure

The request that is returned to the kernel's FOREVER loop is represented by the **zx_request** data structure. It contains fields of argument 1 to argument 5 (since the maximum number of arguments according to the kernel specification is 5), and those arguments are treated differently depending on the call. It also contains SWI value that indicates which system call, if any, has been initiated.

Queue and scheduler

In our case, we have decided to integrate the scheduler and the queue together. The API s are defined in *xz_queue.c*. It has the following basic functions:

zx_q_get_next_task() - return the next task to be scheduled

zx_q_append_to_end(int tid, int priority) - add a task to the end of a priority ready queue.

zx_q_update_highest_priority(int temp_priority) - update the highest priority value, when a new task of a higher priority is created

zx_q_remove_node(int tid, int priority) - remove a task from a priority ready queue

Each of the functions may store the new values of the highest priority at the memory space. Queue has a certain memory location to store the head pointers and nodes.

Queue is implemented as a doubly-linked list, and there is a list of head pointers for ready queues of each priority.

We have implemented fast search for the highest available priority using a 32-bit variable. Each bit contains 1 if there is a task of the corresponding priority, and 0 if there is not (the order is left to right, so that 5-th bit corresponds to the priority 5). Therefore, in order to get the highest priority, the kernel calculates the number of trailing zeros in this 32-bit variable, and the result will be exactly what we need. The algorithm is constant-time, and the time is small (log 32), because the number of trailing zeros is calculated using binary search and logical operations. So, we support 32 task priorities.

Kernel Termination

We distinguish kernel termination and kernel shutdown. The kernel is terminated by returning from its main function. Kernel shutdown is more complex, with freeing the resources and described in Shutdown system call description.

Chapter 3. RESOURCE AND MEMORY MANAGEMENT

API and System Calls

Standard

In the kernel specification, no system calls defined for memory management

Additional

```
/* Allocate memory for a node */
void* Alloc();

/* Free the memory location */
void Free(void* mem);
```

Memory allocation and stack

Regarding stack allocation, we divide a particular memory into uniform blocks of **TASK_STACK_SPACE_SIZE**. Each task gets its own block of memory.

We use post-incremental load and pre-decremental store type of the stack, so we load with **LDMFD** instruction and store with **STMFD** instruction. We have chosen this way, because the default compiler settings use the same approach (so we avoiding the confusion), and it seems to be very convenient (unless empty, stack always points to some stored value).

Besides that, for this assignment, we often allocate the memory in the available address space for other things. We put data structures like queue and variables like next task id into a particular constant place (like **READY_QUEUE_HEADER_BASE** or **NEXT_NEW_TASK_ID**) in the available memory calculated at the compilation time. Those places are chosen in a way that it will not intersect with any stacks, including the kernel and tasks' stacks. This approach allows simple access and convenient debugging. Therefore, we do not have to pass lots of arguments, and we can view the data whenever we like (the last one is particularly important as it is harder to debug if you allocate the variables into a kernel stack space).

Heap Memory / Dynamic Memory Allocation

This is to allow tasks to use and free the memory dynamically as and when it is needed. This is particularly useful for Name Server which uses Binary Search Tree. Nodes of the tree are allocated dynamically since the number of nodes in the tree is not known in prior.

Implementation

Each task when it is created is provided a heap space in its address space form which memory could be allocated and freed as per the need of the program. By the current design, the heap space is $1/4^{th}$ of the task's address space which is 64KB. The heap is a singly linked list with head and tail pointer. The linked list is the list of free memory locations that could be allocated to requesting process. Memory is always allocated in chunks of 16 bytes. This is because heap memory has been mainly implemented for trees. It allows the tree nodes to store right and left node along with two data values.

Chapter 4. INTER-PROCESS COMMUNICATION

This part of the kernel implements the communication mechanism between tasks.

API and System Calls

Standard

The system calls that support communication are defined in the kernel specification.

```
int Send( int tid, char *message, int message_length, char *reply, int reply_len);
```

Sends a message of the length message_length to the task with specific ID and receives the reply (if any) of the available length reply_len.

```
int Receive ( int *tid, char *message, int message_len);
```

Receives the message of the available length message_len from any process (tid is modified upon the receiving), and returns the number of bytes read or error code.

```
int Reply( int tid, char *reply, int reply_len);
```

Replies to the process with task ID tid and passes the pointer to the reply buffer reply and the length of the reply reply_len.

Name Server API allows user tasks providing services to other tasks by registering their names and getting task IDs of already registered tasks. A task that needs a particular service, queries the name server with the name of the service, and gets the task ID providing that service. The name server API includes:

```
int RegisterAs( char* name );
```

Assigns a name for the current task.

```
int WhoIs( char* name );
```

Returns the task ID of the process that is assigned the particular name, or error if there is no such task.

The details implemented for this assignment will be provided in the "Definitions, Data Structures and Algorithms".

Additional

No additional services provided besides the kernel specification.

Constants

ERR_IMPOSSIBLE_TASKID = -1 – an error value indicating an invalid task id is specified in some of the functions above
ERR_NONEXISTENT_TASK = -2 – an error value indicating a task id is valid, but the task does not exist
ERR_SEND_RECEIVE_REPLY_TRANSACTION_INCOMPLETE = -3 – an error value indicating a communication transaction was incomplete, but the task was sent a message

Servers associated

Server	Purpose
Name Server	Centralized service to store the names of the servers and their associated Task ID s. Any task can query the name server to by the name of the server to know the TID of the server. The TID is used by the task in the Send () call. Nameserver provides this service by WhoIs and RegisterAs system calls.

Internal structure and implementation

The algorithm of implementing these routines mostly follows the specification. The only difference is that we call the queue we store all the incoming messages as Receive queue, not Send queue.

If both the send and reply tasks are of the same priority, the sender will be put to the front of the reply task as soon as the reply is sent.

Another point is that we use stack whenever we add a task to a blocking state, to store the necessary arguments (pointer to the reply buffer and reply length, for example). Then, when we unblock the task, we pop the arguments from the stack of the task.

In addition, we handle errors of send-receive-reply transaction is incomplete indirectly. Upon the completion of the program, we run a checker that finds all the tasks that are in blocked state. If such are tasks found, their IDs and states are printed, so it is easy to get whether we have an incomplete transactions.

Data structures and algorithms

Receive Queue

Purpose:
The purpose of the receive Queue is to queue up the messages that is being sent to a particular task before a Receive() call is made to by the task to read the messages.

Implementation:
Each task has a receive queue associated with it. Each node of the queue stores the following details- 1) Sender ID, 2) Pointer to Message Structure, 3) Message Length. Each queue has one head pointer and one tail pointer. Nodes are removed from the queue and added to the tail.

Time & Space Complexity:
Search: O(1)
Insert: O(1)
Space: O(n)

API:

```
/* Create a node to be added to the queue for the message received */
zx_receive_queue_node* zx_receive_q_create_node( int tid, void *msg, int msglen);

/* Add a message to the end of the queue */
void zx_receive_q_append_to_end( int tid, zx_receive_queue_node* new_node );

/* Get the next message from the queue */
zx_receive_queue_node* zx_receive_q_get_next_node(int tid);
```

Memory Allocation:
The memory for the queues is allocated **statically** when the kernel is initialized. Since each task can send only one message to any task before it will be back to READY state again, the system will have only 64 messages utmost queued up at any point in time. So we allocated 64 nodes when the kernel begins to run. Message node of a particular task can be retrieved by mapping the task ID to corresponding memory location. A message sent by a task is always allocated a node only at that memory location. We also 64 head pointers and 64 tail pointers, one per task. The head pointers and tail pointers are updated accordingly as messages are added and removed from the queues.

Chapter 5. TIME KEEPING

API and System Calls

Standard

/* The task which executes this call is woken up after 'ticks' mentioned */ **int** Delay(**int** ticks)
/* The task which executes this call is woken up when the global time matches the ticks */ **int** DelayUntil(**int** ticks)
/* Returns the current time in terms of number of ticks passed since the kernel began execution */ **int** Time()

Additional

/* Creates a clock server with its standard priority and returns its tid */ **int** CreateClockServer( )
/* Close the clock server created in the previous routine and returns the nymber of ticks passed before closing the server */ **int** CloseClockServer( )

Constants

ONE_TICK = 5080 – a number of clock ticks in one operating system tick (10 ms, 508 KHz clock).

ONE_MILLISECOND_TICK = 508 – a number of clock ticks in one millisecond (508 KHz clock)

Associated servers

Server	**Purpose**
Clock Server	Serves all the above system calls

INTERNAL WORKING

Interrupts are enabled from the timer and the clock granularity is set to 10 ms. Each time the timer underflows, it generates an interrupt which is notified to the clock server. The clock server keeps track of the global time since the kernel began, in terms of number of ticks.

Chapter 6. INPUT-OUTPUT FUNCTIONS

API and System Calls

Standard

/* To output a single character to COM2 or COM1 */ **int** PutC (channel, byte)
/* To read a single character from COM2 or COM1 */ **int** GetC (channel)

Additional

/* Outputs an array of characters to the given UART */ **int** PutMultiple (channel, **char*** array)
/* Creates all the four UART servers and associated notifiers */ **void** CreateUARTServers ()
/* Closes all the four UART servers and associated notifiers */ **void** CloseUARTServers ()

Constants

UART_PACKET_END_CHAR = 255 – a character indicating the end of the packet in PutMultiple routine.

Associated servers

Server	**Purpose**
UART1 Transmit Server	Transmit Byte to COM1
UART2 Transmit Server	Transmit Byte to COM2
UART1 Receive Server	Receive Byte from COM1
UART2 Transmit Server	Receive Byte from COM2

Implementation

See associated servers in the Chapter 7- “SERVERS, NOTIFIERS AND IDLE TASK”.

Chapter 7. SERVERS, NOTIFIERS AND IDLE TASK

The kernel provides some important services to the user tasks by means of servers and notifiers. Servers are tasks that provide the service. Notifier is a task that notifies the server when to perform its task, or better notifies an event to the server. Each server is coupled with a notifier, except for the name server, and vice versa which provides a clean design. Below table summarizes the Servers and the data structures used by them.

SERVER	**PURPOSE**	**ASSOSCIATED SYSTEM CALLS**	**DATA STRUCTURE**
Name Server	For registering tasks that provide services to other tasks. Stores name, TID association.	WhoIs() RegisterAs()	AVL tree
Clock Server	For time keeping and notifying tasks of current time	Delay() DelayUntil() Time()	Min Heap
UART Server	Input Output Functions to COM1 and COM2	PutC(),GetC(), PutMultiple()	Queues

Detailed explanation of each server is below.

Name server

Implementation Details

Name server is a separate task and is started by the first user task when the kernel boots. So we have decided to create a wrapper for the creation of it and called it CreateNameServer(int priority**).** The main part of Name Server is a loop, which starts with Receive command. Once the name server receive a message (which is of the zx_ns_request type we defined), it checks for its type. If its type is **API_NS_WHOIS**, then it means that WhoIs is called, and the server will query the data structure and return the result.

If the type is **API_NS_REGISTER**, it means that RegisterAs is called. The server will add the entry to the data structure.

We have also defined the request type **API_NS_EXIT** that will shut down the name server. It is done for nice exiting and we have implemented CloseNameServer() function for this

purpose. However, the program will still be executed correctly if this function is not called, but it will show the warning that the name server task is in SEND_BLOCKED state.

The server will reply with the erroneous message type if it receives the request type it cannot process. We are relying on the user (because we are the users) regarding the request and response sizes, so we decided not to implement runtime type-checking.

Data Structure and Algorithms: Self-Balancing Binary Search Tree (BST) for Name Server

Purpose:

Self-balancing binary search tree (AVL tree) is used by the name server to maintain the record to server tasks registered to it. We chose self-balancing binary search for better space complexity. Through there are other methods which perform better search (for example, in O(log log n) time), we find the space complexity of those structures is poor for small number of nodes. Hence as an essential trade off, we settled for self-balancing binary search tree.

Time & Space Complexity:
Search: O(log n)
Insert: O(log n)
Space: O(n)

Implementation:

Each node of the BST (for obvious reasons) store pointers to left and right nodes, task_id and hash_value. Currently, BST is used only by the name server. The hash_value is the hash generated for the name of the server task that's is interested in registering itself. Task_id is the ID of the task whose name is being registered to the name server. Hence, the BST effectively stores the key-value pair.

API:

```
/* Add task name to the tree */
/* Returns 1 on SUCCESS, -1 on FAILURE */
/* Node allocation fails when there is not enough memory */
int zx_tree_add_by_name( zx_bst_node **root,  char *name, int tid );

/* get the task ID for the given task name */
/* Returns the ID of the task if a match is found */
/* Returns -1 if match is not found */
int zx_tree_get_by_name( zx_bst_node  *root,  char *name);

/* free the memory allocated for the tree */
void zx_free_bst( zx_bst_node **root);
```

Memory Allocation:
Memory is allocated from the task's memory pool by calling Alloc() function as when a node needs to be added to the tree. In the end, the task is expected to free the memory it has used by freeing up the nodes.

Design Decisions: Hash Function for the name server

Purpose:
We generate a perfect hash value for every task name that is being registered to the name server. This is to enable fast and convenient searching. The hash value stores the key values into a self-balancing binary search tree. There are no collisions at all (perfect hashing), so this approach is very safe.

Implementation:

$$Hash\_value(string) = (\text{string}_{\text{length}} \ll 24) \mid \sum_{i=1}^{string_{length}} i * string[i]$$

API:

```
/* Find Hash Value of a string */
int zx_get_hash_value( char *string )
```

Clock server and notifier

For each tick, the clock server updates the global time counter (a pseudo global variable). This is used for profiling when the kernel exits. Time-Keeping function of kernel is supported by the clock server. All the calls to Delay, Time and DelayUntil are redirected to the clock server.

Implementation details

The Clock server receives a message. If it hears from the notifier, it almost immediately replies to make the notifier be awaiting for an event. If it hears from a client and the API request is Delay or DelayUntil, puts the task id to the heap as well as the wake up time: (time + delay interval) or just (end time) correspondingly. If it receives a Quit message, then next time it replies to the notifier, it replies with the Quit message. After the notifier is exited, the clock server exits as well so that the termination of the system is safe and nice.

Clock notifier design is very simple. First it handshakes and synchronizes with the clock server, so that they both become ready at the same point. The main loop of the notifier consists of AwaitEvent call, and then Send to the clock server. After then, it checks whether the reply says to quit, and if so, just exits.

Data structure and algorithms – MIN heap

What is a Min-Heap ? It is a complete binary tree with least value node always at the root of the tree.

Why we used Min-Heap: To store the delayed tasks in sorted order. The tasks are sorted in increasing order of wake_up_time.

Data Stored in Min-Heap: Each node of the min-heap stores the ID of the task which called Delay() or DelayUntil(), and the wake_up_time of the task. The heap is sorted by the wake_up_time.

Min-Heap Interfaces:

```
/* Insert a node in the heap */
void zx_clockserver_heap_insert( int tid, int wake_up_time )

/* To get the task with earliest wake up time */
void zx_clockserver_heap_remove()

/* To delete a node from the heap */
zx_clockserver_heap_node* zx_clockserver_heap_get_min_node()
```

UART Servers

UART servers enable tasks to send and receive data to the train controller (COM1) and to the output screen (COM2). Interrupts are enabled in these peripheral devices and these interrupts are handled by the four servers and four notifiers as listed below:

ZX_UART1_TX_SERVER and **ZX_UART1_TX_NOTIFIER** handle UART1 Transmit Interrupt.
ZX_UART1_RX_SERVER and **ZX_UART1_RX_NOTIFIER** handle UART1 Receive Interrupt.
ZX_UART2_TX_SERVER and **ZX_UART2_TX_NOTIFIER** handle UART2 Transmit Interrupt.
ZX_UART2_RX_SERVER and **ZX_UART2_RX_NOTIFIER** handle UART2 Receive Interrupt.

There is no server and notifier for the UART1 Modem Interrupt. This interrupt is handled internally by the kernel. The task that is awaiting for UART1 Transmit interrupt will not be awakened until all the required Modem interrupts are received. This implementation will not make user to worry about handling modem interrupts, and this makes sense.

Internal work principles of UART1_TX_SERVER and UART2_TX_SERVER

First, the creation of the corresponding notifier is done, and handshaking goes after that. Next. in the main loop, the server receives requests from the notifier and clients.

If heard from a notifier, it means we are ready to send a next character. If the character queue has at least one element, then take this element out and, by replying to the notifier, ask the notifier to send this character. If the queue is empty, then put the notifier into a ready state so that it can send a new character when it appears in the queue.

If heard from a client, then if the client has called PutC, then the following happens. If the notifier is in ready state, then ask the notifier to put the character and reply to the client. Else, put the character to the character queue. Reply to the client with the success message UART_TX_SUCCESS, if no errors. Reply to the client with UART_TX_ERROR if the queue was full when tried to put a character.

If heard from a client, and if the client has called PutMultiple, then the following happens. If the notifier is in ready state, then ask notifier to put the first character from the packet and reply to the client. Put all remaining characters to the character queue and reply to the client. If the notifier is not ready, then put all the characters into the queue and reply to the client. The packet ends with the character UART_PACKET_END_CHAR. There is no artificial limit on the length of the packet, so available memory is the only limit.

If heard from a client, and if the client has sent an exit message, then the following happens. The server is put into the exiting state. In the exiting state, the server does not accept any incoming requests from clients. It does not respond to them even with an error message, because there is an automated checking of all tasks that are REPLY_BLOCKED upon the completion of the program, and the only reason the server cannot respond is that it has been exited. In the exiting state, the server, as soon as it hears from the notifier, replies to the notifier with an exit message, and the notifier stops its execution as well. Then, the server exits, and the exit transaction is complete.

Internal work principles of UART1_RX_SERVER and UART2_RX_SERVER

First, the creation of the corresponding notifier is done, and handshaking goes after that. Next, in the main loop, the server receives requests from the notifier and clients.

If heard from a notifier, it means there is a character that has been input. The server

replies to the notifier after getting the value returned by the notifier. The server has a task queue that stores all the tasks that are waiting for the character. If no tasks are waiting for an input, then throw an error. If the task queue has at least one element, then take this client out and, by replying to the client, send the character to the client. The notifier is turned unto EVENT_BLOCKED state right after it received the reply from the server.

If heard from a client, then if the client has called GetC, then add the client to the task queue, and the task will remain there until there is a character from the notifier.

Server and notifier priorities

```
#define FIRST_USER_TASK_PRIORITY   15

// definitions of priorities of clock and uart servers and notifiers

#define ZX_CLOCK_NOTIFIER_PRIORITY 2
#define ZX_CLOCK_SERVER_PRIORITY 3

#define ZX_UART1_TX_NOTIFIER_PRIORITY 0
#define ZX_UART1_TX_SERVER_PRIORITY 1

#define ZX_UART1_RX_NOTIFIER_PRIORITY 0
#define ZX_UART1_RX_SERVER_PRIORITY 1

#define ZX_UART2_TX_NOTIFIER_PRIORITY 4
#define ZX_UART2_TX_SERVER_PRIORITY 5

#define ZX_UART2_RX_NOTIFIER_PRIORITY 6
#define ZX_UART2_RX_SERVER_PRIORITY 7

#define ZX_NAME_SERVER_PRIORITY 10
```

The servers and notifiers run in non –interruptible mode. We achieve this by setting SPSR of the notifier and server tasks appropriately when the tasks are created. The reason for making the notifier non-interruptible is that generation of an INTERRUT X when notifier for Y is running, corrupts the control flow of Notifier Y which sometimes results in the crash of the kernel.

UART1 RECEIVE AND TRANSMIT SEVERS and NOTIFIERS are given the highest priority. The reason for this is in explained below diagram

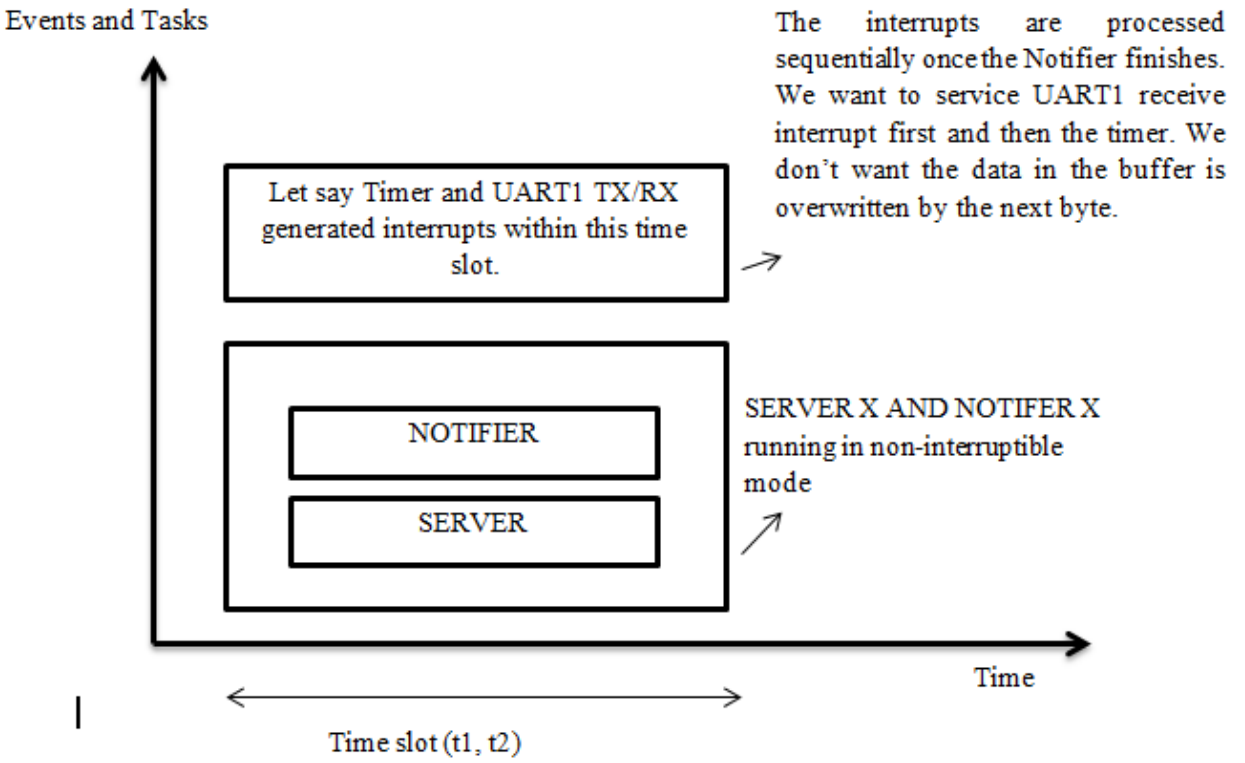

Fig 7.1. Notifier Priority Explained

Idle task and profiling

The idle task is a lazy task which ideally does nothing but running in an empty forever loop. To profile the idle task, each time the idle task starts running and TIMER_2 is also enabled. The timer is stopped when the idle task is interrupted. The number of milliseconds passed since in and out of idle task is added to the global counter (which is a pseudo global variable). When the idle task exits, it prints the time (in ms) the entire program has spent in idle task and time for which the program has run. From these two values, it calculates the percentage of time spent in idle task.

For our code, the average of idle task running time is 89% (ran the program for 10s).

Chapter 8. ENABLING INTERRUPTS AND PER INTERRUPT HANDLING

How are the interrupts enabled

ZX_KERNEL handles six different interrupts – one for time keeping and remaining five for Input Output functionality. To receive interrupt from a particular device, the kernel enables the interrupt in the VIC (Vectored Interrupt Controller) and also in the device. Both are done my setting appropriate bit in the VIC register as well as the Device Interrupt Status Register. Once interrupt is received from a device, we turn off any further interrupts from the device by setting the bit of the Device Interrupt Status Clear Register. The interrupts are turned on again once the current interrupt has been handled.

Interrupt processing and priority

Interrupts arranged in the order of processing, (required when more than one interrupt is generated simultaneously)

CLOCK

Clock Tick Interrupt [INTR_SOURCE_TIMER_1] - has the highest priority because we don't want to lose clock ticks.

UART1

Modem Status Interrupt [INTR_SOURCE_UART1_MODEM] - since it is the most frequent interrupt, it is given highest priority among UART1 interrupts

Receive Interrupt [INTR_SOURCE_UART1_RX] - more important than transmit because we need to react to sensor data as fast as possible

Transmit Interrupt [INTR_SOURCE_UART1_TX] - less important interrupt in this group

UART2

Transmit Interrupt [INTR_SOURCE_UART2_TX]- we print to the screen a lot

Receive Interrupt [INTR_SOURCE_UART2_RX] - we are typing in something less frequently than printing.

Timer interrupts

Interrupt : Clock Tick Interrupt [INTR_SOURCE_TIMER_1]

Clock used: 508 KHZ

Timer used: Timer 1

Pseudo-global variables:
(Always read from pre-defined memory locations, check zx_kernel_config.h):

AWAIT_TIMER_ID to store the ID of the task waiting for the timer event, which is effectively clock notifier task

What does the timer interrupt handler do?

The timer interrupt handler reads from global predefined location, the ID of the task which is waiting for timer event. Then moves the task from EVENT_BLOCKED state to READY_STATE and appends it to the ready queue as well as assigns the return value.

Interrupt Workflow Diagram - Timer

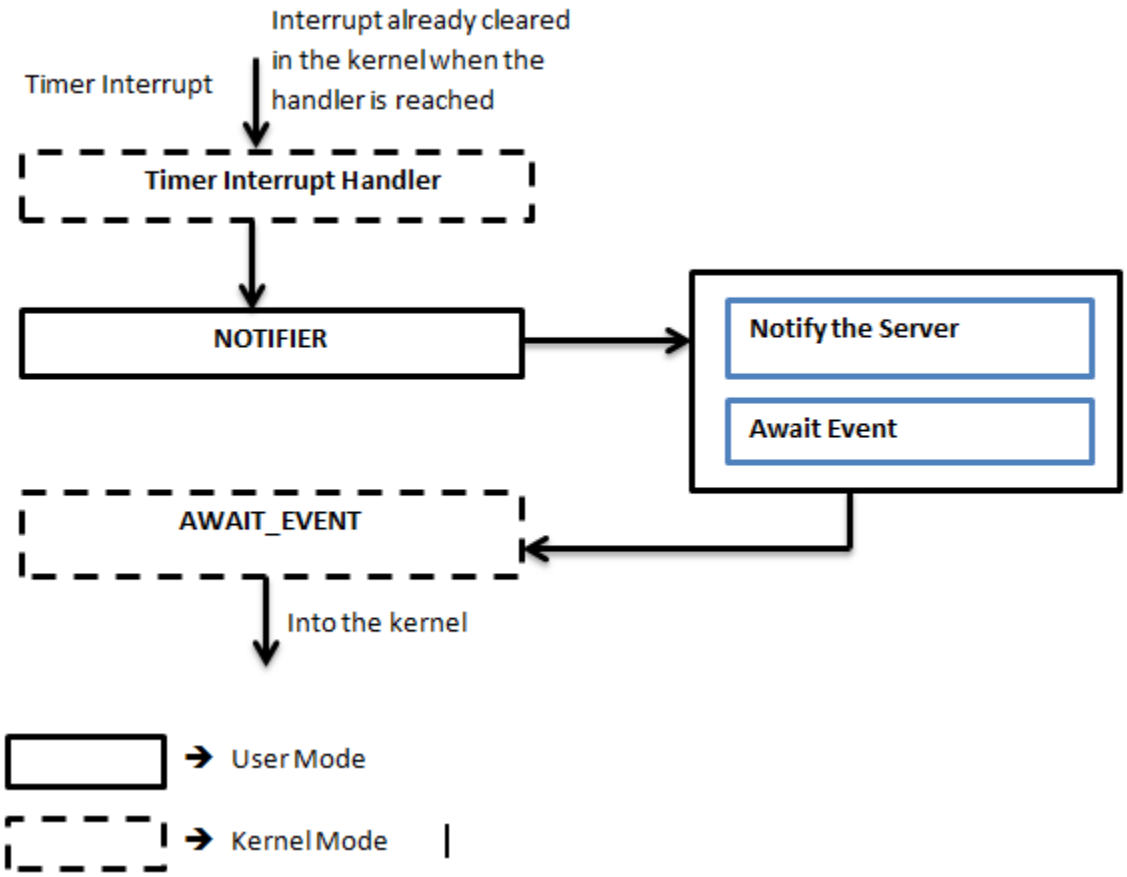

Fig 8.1. Timer Interrupt Flow diagram

UART Interrupts

Transmit Interrupts

UART1 Clear To Send Interrupt from Modem [INTR_SOURCE_UART1_MODEM]
UART2 Transmit FIFO Empty Interrupt [INTR_SOURCE_UART2_TX]
UART1 Transmit FIFO Empty Interrupt [INTR_SOURCE_UART1_TX]

Receive Interrupts

UART1 Character Received Interrupt [INTR_SOURCE_UART2_TX]
UART2 Character Received Interrupt [INTR_SOURCE_UART2_RX]

Interrupt Workflow Diagram - UART

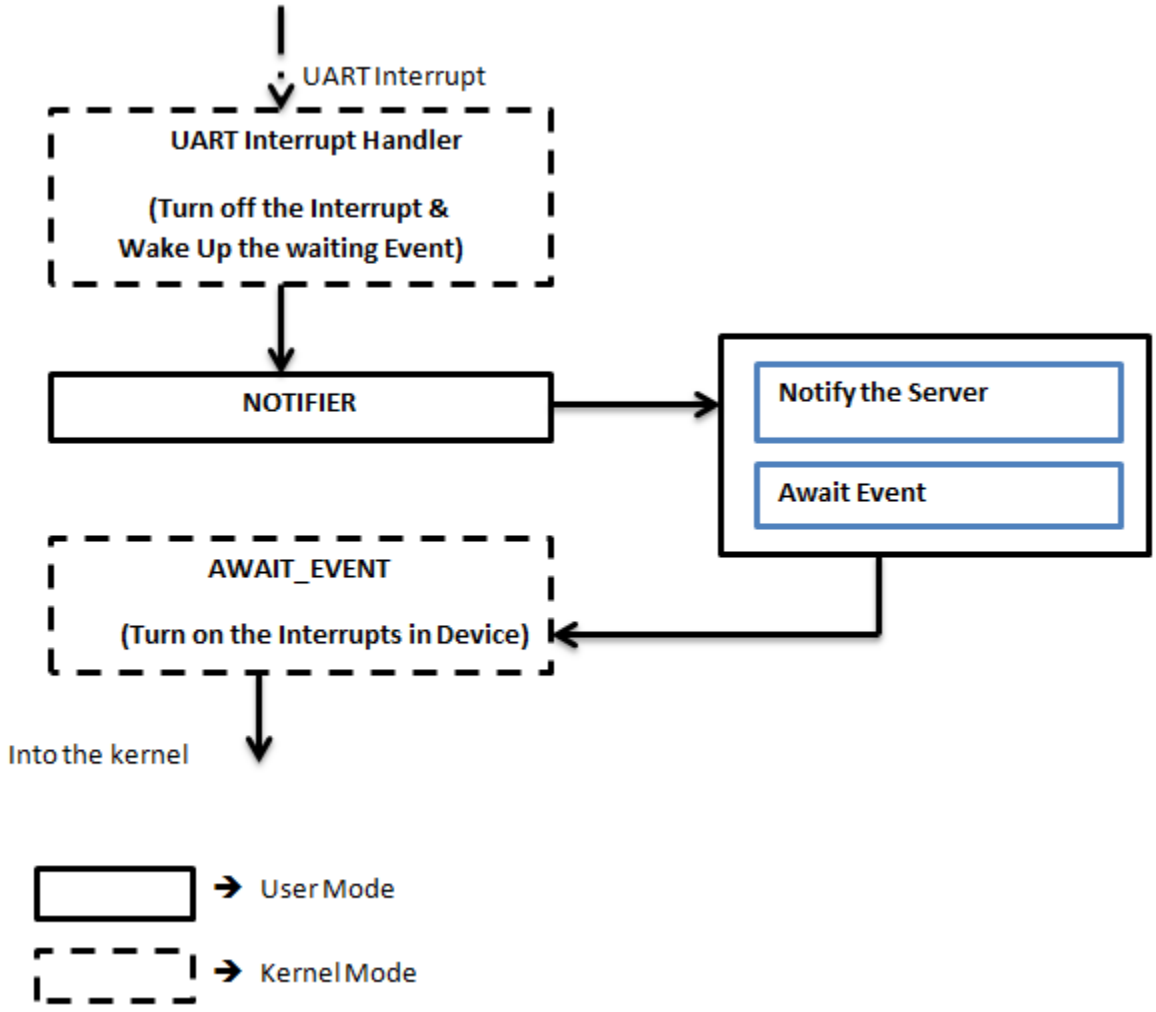

Fig 8.2. UART Interrupt Flow diagram

For UARTs, we turn the interrupts OFF and ON only inside the kernel. We turn off the interrupt as soon as we find the source of the interrupt and wake up the corresponding notifier. When the

notifier blocks on AwaitEvent, the interrupt is again enabled by the AwaitEvent system call handler.
The UART transmit notifiers after notifying server remains in blocked state. It is unblocked only when the server sends a character back to it to transmit.
Except the MODEM status interrupt, other interrupts of UART need not be cleared. They are cleared automatically when transmit or receive happens.

Curious Case of UART1 Transmit Interrupt

Unlike UART2, UART1 transmit is controlled by two interrupts because of its low baud rate:

1. UART1 TXFE interrupt (INTR_SOURCE_UART1_TX) - generated when the TRANSMIT FIFO is empty in UART1.
2. UART1 Modem Status Interrupt ([INTR_SOURCE_UART1_MODEM]) - asserted when the Train Controller toggles CTS (Clear to Send flag)

We maintain a state machine in the kernel and awake the UART1 transmit notifier only when the CTS is high and the Transmit FIFO interrupts is also asserted. This is essentially for flow control since the train controller accepts data in much lower baud rate (2400 bps). If we do not check the CTS line, we may end up sending the bytes consecutively just before when the CTS is low. This caused the byte sent to be dropped by the controller and not processing it.
The state machine (below) runs inside UART1 transmit handler.

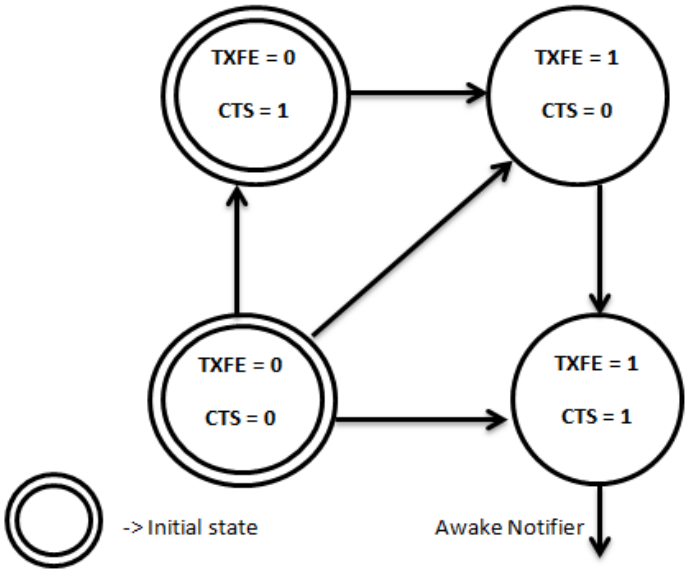

Fig 8.3. UART1 Transmit Interrupt State Machine

Chapter 9. DEBUGGING AND ERROR HANDLING

Exception handling is done in the following way.

In non-critical parts of the code, the system simply prints an error message if something went wrong (e.g. the server is unavailable, a task has an invalid task id).

In more critical parts (e.g. receiving interrupts, in wrapper to PutC), it is risky to print error messages, because of the two reasons:

1) They may introduce delays and affect the execution of the program.

2) The UART servers may be down as well, so it will not be possible to print anything

3) The printing is done with some delay until the UART transmit server is running, so it might not be possible to see error message at the same time it was printed.

So, the following solution has been proposed to solve the debugging and error handling problems.

A memory address range (DEBUG_VAR_BASE) has been defined. Whenever an exception occurs, a corresponding memory location within this range is written. For example, the event has come and the interrupt is high, but the notifier is not in the EVENT_BLOCKED state. This is bad, because we can lose timer ticks. So, of this happens, the memory location DEBUG_VAR_MISSED_INTR will contain non-zero value. The program can still run or crash, it does not matter. After the execution of the program, we make reset. Then, we can check memory location for non-zero values.

However, it is often hard to parse errors if there are. They may contain hexadecimal values that are hard to recognize. So, a special error recognizer (as a separate function or a new program loaded into the memory) can be created that executes. It looks at the same memory address and looks for non-zero values. If there are, the program shows the memory address value in different representations: hexadecimal, decimal (and can be extended to support ASCII). Moreover, the program can be extended to print the exact error reasons.

Advantages:

1) Saves time and effort to recognize error messages

2) Good solution for error handling that are recovered at the runtime, but should be pointed attention after the execution is complete.

3) Easy to implement

Disadvantages:

1) If there is a serious crash with memory damage, then this approach does not work. But we assume that this is an extremely rare case.

2) Hard to maintain error locations in order not to confuse them and have false-negative results.

However, the approach has been tested and it works well. Some errors have been captured using this approach: for example, losing ticks because the clock server has been interrupted and did not reply back to the clock notifier within the desired period of time.

Chapter 10. ZX_KERNEL FREEBIES

Pseudo-Random Numbers Generator

Purpose:

To allow random moves to be made by game clients. Useful for other purposes as well.

Implementation:

Linear congruent generator: $x_{i+1} = (a\, x_i + C) \pmod m$.
We took a = 214013, C = 2531011, m = (1 << 15), since they follow the rules of the algorithm, and suitable for our needs.
As a seed we use the timer value. Since we are waiting for the user input, the timer value will be different, so we can easily get different numbers.

Send Function Timing Measurements and Analysis

For the case of **4 byte** message, the time is the following:
Time (508 KHz). Ticks: '214'. ~Micro seconds: 428.

For the case of **64 byte** message, the time is the following:
Time (508 KHz). Ticks: '452'. ~Micro seconds: 904.

For the performance analysis, we used 508 KHz clock, because the one with 2KHz is not accurate at this point, and the value tells nothing. We put the measurement code above the SWI statement inside the Send wrapper, and below the SWI statement, right after we saved the return value (we also had to put the store multiple/load multiple above SWI because our kernel assumes the first four arguments are passed as registers, which are corrupted in case we put measurements there, this can be fixed in the future by looking up the stack rather than registers). The amount of microseconds is approximate, since we just multiply the value by 2 (1000000 / 508000 is almost 2).

We have looked at the time and see that despite the message is 16 times longer, the time is just two times bigger (even if we take into account both send and reply message). It means that some time (call it A) is spent just for switching and execution of memory copying-irrelevant code. And some time B is spent on memory copying itself. Probably, time B depends on how fast we can copy memory from one place to another, and we can improve it by using caching. Time B depends on how fast we can switch and execute mandatory code (getting arguments, working with stack, etc.), which is hard to eliminate given we do this part as optimal as possible.

We do not have something we can compare to, but we think that the result is quite good. First, we are not losing extra time since we switch immediately from the sender to the receiver after we

sent, and after we reply we switch immediately to the sender task. Next, the time remains unchanged when we run the program multiple times (variations are ~2 ticks only). And finally, in case of very small messages, 0.5 milliseconds delay is very small.

We could improve the result by turning caches on. However, we could not do this using RedBoot (cache command), but the other way seems to be very hard, and the assignment does not specify this requirement, so we have decided to limit ourselves to measuring the time without cache.

Task State Info

1. On exit, the program displays warning if any of the task is in blocked state (including EVENT_BLOCKED one) and has not yet completed.
2. On exiting, the program prints the time it has spent in idle task and the total time spent running all the tasks. Finally it prints the percentage of time spent in idle task.
3. On exit, the program unblocks EVENT_BLOCKED tasks. So that they return some value meaning that they were forced to exit

Chapter 11. ZX_KERNEL CONSTANTS AND CONFIGURATION

Configurable Values

These values can be reassigned and do not require additional code support

TITLE	DEFAULT VALUE	MEANING
SWI_VECTOR	0x08	The location of the instruction jumping to the SWI handler
IRQ_VECTOR	0x18	The location of the instruction jumping to the IRQ handler
MEMORY_BASE	0x00044f88	The starting address of the available memory (shown by RedBoot)
MEMORY_END	0x1fdd000	The ending address of the available memory (shown by RedBoot)
TASK_STACK_SPACE_SIZE	360 * KB	The size of the stack of each task
HEAP_SIZE	(TASK_STACK_SPACE_SIZE / 4)	The amount of heap memory available for each task
ZX_UART2_TX_BUFFER_SIZE	1200	The size of the transmit buffer for UART2
ZX_UART2_RX_BUFFER_SIZE	100	The size of the receive buffer for UART2
ZX_UART1_TX_BUFFER_SIZE	400	The size of the transmit buffer for UART1

ZX_UART1_RX_BUFFER_SIZE	100	The size of the receive buffer for UART1

Non-Configurable Values

These values cannot be reassigned: for information only.

PRIORITY_COUNT	32	The number of supported priorities
TASK_COUNT	64	The number of supported tasks

Chapter 12. ASSIGNMENT 0 IMPLEMENTATION

Overview

Assignment 0 previously was implemented by polling mechanism. With the microkernel, the assignment 0 now has been implemented by means of interrupts.

Tasks

Assignment 0 is accomplished by 3 tasks.

1. Timer Task

Calls Delay () and calculates the correct time each time it's woken up. It also updates the current time on screen

2. Sensor Task

This task continuously queries the sensors and update the sensor information on the screen.

3. Command Task

This task reads input characters from screen, accumulates the characters in a buffer and process the command once the complete command has been received.

Error Message Printing

The command task prints appropriate error message on the screen when the command is incorrect.

Error Messages & Scenarios:

ERROR MESSAGE DISPLAYED ON SCREEN	**SCENARIO**
Invalid Command	When an incorrect command is typed
Switch position is invalid	When the user forgets to mention the switch position in the command "sw <switch_number> <position>"

Assignment Questions:

1. How do we manage clock updates and still not lose ticks?

There is a clock task that delays each 100 ms and updates the timer display. The time is calculated according to the current time received from the clock server. The time is decomposed into components (minutes, seconds, tenths of seconds). We do not store number of seconds or minutes anywhere, this makes our timer more precise. So we don't lose ticks.

2. What is the delay observed in sensor data received once the controller has been queried for data?

6 ticks on average (~60 ms). We found this by taking the time difference between 2 consecutive sensor dumps. We request a sensor dump only when the data for the previous dump command had been received completely.

Chapter 13. README

/* READ ME : zx_kernel */

Project Directory : /u0/pvsrinivasan/cs452/k4/

Makefile : /u0/pvsrinivasan/cs452/k4/Makefile
Executable : /u0/pvsrinivasan/cs452/k4/bin/zx_kernel_a4.elf

This directory contains the source files, header files, executables and README of zx_kernel_code

The directory structure is as follows
/u0/pvsrinivasan/cs452/k4/src - c source files
/u0/pvsrinivasan/cs452/k4/include - header files
/u0/pvsrinivasan/cs452/k4/bin - compiler generated binaries (.elf file)
/u0/pvsrinivasan/cs452/k4/obj - compiler generated object files (.obj files)
/u0/pvsrinivasan/cs452/k4/asm - compiler generated assembly files (.asm files)

How to run?
---
load -b 0x00218000 -h 10.15.167.18 “ARM/priyaa/zx_kernel_a4.elf”

To quit the program, press `'q'` and Enter any time.

How to make?
---
the makefile generates executable for arm3 CPU. To generate
the executable, run the makefile in this directory. You could
do

1. make clean - to delete all compiler generated files from previous compilation
2. make - build source and produce executable
3. make upload - make + upload the executable to tftp server

Chapter 14. FILES AND HASH VALUES

8957c223616bb9213d57e439736f1821 zx_a0_graphics.c
4f67325e59a51a7fa832a6abf8a39c56 zx_a0_main.c
00f8ff3db6b7a5b59402e09ee0c1a955 zx_a0_utilities.c
565c871d7ec328427c3a236af3945162 zx_clock_server.c
eb9c4eb1f57c182e9018eb2316cd03d7 zx_clockserver_heap.c
594fe26968ac075958590669c1659ced zx_helper.c
9943dee2db94abfec80495ef4f3aa2fa zx_irq_device.c
23cdc70a761b391bbb18ba81117856fa zx_irq_vic.c
be68ec1080f3063bd86f23cae649541a zx_kernel_body.c
598fe5dcb7a84a499953f3938a98225c zx_kernel_head.c
e18ed8c0c9add335ef82297dd3d81300 zx_memory.c
0e48815f1725c7cbb798d2e5927893f4 zx_ns.c
b06c303610e2c3422b0dc541112a9cfd zx_ns_tree.c
a6b3474693586c7af7e5eebefc9d3d96 zx_queue.c
dcb75f4ce3b61b3f03802bbabc870a39 zx_ready_queue.c
f7292a8977b0f433dfe4cf2a8c8bcafc zx_receive_queue.c
642bfa5510c410c7d87b3be66fdb316d zx_system_call_handlers.c
18be562f899ef31099af37792897b670 zx_system_calls.c
10e5ee5a290bbe2c003e9d4f96f5e1f2 zx_testing.c
49c7e18346e2e5b225cdf499e4c144cb zx_timer.c
25b7fbbc7fc67af1f50dc5bb5f2c38cf zx_train_a0.c
8bb0b06ceb6a53e17151df846dcae1cd zx_uart1_receive_server.c
c0e85303442932744f12b2082c7463ff zx_uart1_transmit_server.c
144d52da8fb5efe80092a6135810aeb6 zx_uart2_receive_server.c
a0782fde3cf0e33def59ad7d243eacd1 zx_uart2_transmit_server.c
77de18c1c07aaef8393609140ee363b4 zx_uart_io.c
c337166a5595f3fb1800447ec7471023 zx_uart_server.c

include:

e07e1ea67c27e44762ccd7817401e88f zx_a0.h
c02c2b28702bc33dcf38f272563ca56a zx_a0_graphics.h
7adc8460cf736be9c4e223eaf0633afc zx_a0_timer.h
5286e65adbfb9017464aac47f227aa68 zx_a0_utilities.h
7ee00ed811872d725844a3abcfdd28aa zx_clock_server.h
caa42f2e4177ccb1d02f3fd5f80bdc3e zx_clockserver_heap.h
d1b353be4f3928b11925639fdaaca8e2 zx_err.h
9e32191fb8b40e6cbd735897125704fa zx_err_strings.h
2ff840df7ee79ab4f3c4eebf9f760cf3 zx_helper.h
2b436e2a3d2ff878391f8e2fc8d46bbb zx_irq_device.h
494163cbb2d81762735c910a137a1250 zx_irq_vic.h
ee7673fd64f0c2349136d9eb10df4323 zx_kernel_body.h
9ae1b540375b0f33adc6d63fe425f201 zx_kernel_config.h
eab9946a4ebfa26b9cdc67ac7bace8e5 zx_kernel_head.h
ad2993bf3363c7b2a7f20e87fefdc9a7 zx_memory.h
c2d330e9e755d5189f3bd8694725285e zx_ns.h
fa4fc47c10f1465afaec93b397a297da zx_ns_tree.h
690c8cef532b450948a9f0c42e576290 zx_queue.h
1c6dc58b4c98bc2a88e58429795f5ef1 zx_ready_queue.h
4722805f37ba8c426a785f0824145575 zx_system_call_handlers.h
81eb55cecaa2eabc9dc5d3b0667aa378 zx_system_calls.h

```
8afe2176854720202becb1de4645ee9c  zx_tasks.h
7125d55529c35244212f3f2b69ad858c  zx_testing.h
8506954fda8294b1fa21d4ebee7f7c19  zx_timer.h
26fa48d5406a3ea894e080c6976e62db  zx_uart.h
62d5ed643fb28a86286c2a4f7f3038c5  zx_uart_io.h
7ee00ed811872d725844a3abcfdd28aa  zx_uart_server.h
```