CS452 Real-time Programming
Final Examination
Spring 2015

Jason Sun
20387090
j53sun

Aug 5, 2015

Question 1. User State in the Kernel

1.a In the task descriptor

(i) How big would the task descriptor be (in 32-bit machine words)?

To determine how big the task descriptor has to be, I first look at what is set of information that must be kept track of. I describe what that set should consists of, assuming the kernel itself doesn't use its own data structures to keep track of its tasks[1].

1. The **program counter**. Assuming 32-bit architecture, this would be a 32-bit pointer to the task's current instruction. The content of the PC register would be stored here. Size: **exactly 1 word**.
2. The **stack pointer**. Another 32-bit pointer to the address of where the stack was. Size: **exactly 1 word**.
3. The **task state**. The minimum number of states a task would be ready, blocked (be it send blocked, received blocked, etc). There may be a few more states required in the future, but definitely would fit inside an enum (which is an int). Size: **less than 1 word**, because there are a only handful of states a tasks can be in.
4. The **task id**. This is useful to uniquely identify the task for handling its syscalls. Size: **less than 1 word**, because there is a limit on the number of active tasks allowed, due to resource scarcity, and recycling of task id.
5. The **parent id**. Size: also **less than 1 word**.
6. The **priority**. Size: **less than 1 word**, because the number of active tasks is less than 1 word (and having too many priorities is not useful).

For those that have size of less than 1 word, it is possible to use bit masks to combine them together. Below, I construct some new fields that stores several fields together.

- I can **combine the task *state*, *id*, *parent*, and *priority* to one word**[2] **– I will call this word SIPP**. Limiting the task id to 8 bits (2^8 max active tasks) means we have other bits to store the task state and priority.

[1] Example of kernel using its own data structure to keep track of its tasks:
The kernel can use its own data structure to map the relation of task ids to their program counters.

[2] For example, combining them:

```
(state << TASK_STATE_OFFSET) |
(priority << TASK_PRIORITY_OFFSET) |
(id << TASK_ID_OFFSET).
```

And to access, for example, the task state:

```
(TASK_STATE_MASK & task->state) >> TASK_STATE_OFFSET.
```

Suppose I limit the task state to generous 4 bits (2^4 states) and priority to 5 bits (2^5 priorities), I still have some left over bits $(32 - 8 - 4 - 5) = 15$. With this, I can store the parent id using 8 bits.

The total size of the task descriptor would be 3 words, as a struct has no size in C[3].

(ii) Drawing of what's in the data cache when the FirstUserTask has just been created, and just after it has been activated for the first time.

The *data cache*, for the ARM920T processor

- has 512 lines of 32 bytes (8-words), and
- a linefill always loads a complete 8-word line, and
- has either random or round-robin replacement[4].

When the data is not aligned with the size of a line of cache, the compiler for ARM architectures aligns data to closest word boundary[5]. Meaning the compiler pads things up to size, so anything smaller is still a word.

Since it's an 8 word cache line, the bottom 3 bits of any address are ignored, so 8 words are fetched in the neighbourhood of whatever was accessed.

For clarity in the drawing, I

- assume the round-robin replacement method is used (consequently only overwriting the oldest line),
- draw from the most recent on the top to the newest fetch at the bottom,
- draw only the relevant lines (of the 512 total lines) that were brought in with respect to the question.

[3]Source: `http://stackoverflow.com/questions/3849334/sizeof-empty-structure-is-0-in-c-and-1-in-c-why`

[4]The ARM920T processor includes a 16KB DCache and a write buffer to reduce the effect of main memory bandwidth and latency on data access performance. The DCache has 512 lines of 32 bytes (8-words), arranged as a 64-way set-associative cache and uses MVAs translated by CP15 register 13 (see Address translation) from the ARM9TDMI CPU core. The write buffer can hold up to 16 words of data and four separate addresses.

The DCache implements allocate-on-read-miss. Random or round-robin replacement can be selected under software control by the RR bit (CP15 register 1, bit 14). Random replacement is selected at reset. A linefill always loads a complete 8-word line.

Source: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0151c/I1004722.html`

Section: *Home > Caches, Write Buffer, and Physical Address TAG (PA TAG) RAM > DCache and write buffer.*

[5]ARM9 doesn't allow unaligned loads, so the compiler makes it word aligned.

The compiler writers chose to allow the declaration of types shorter than a word (chars and shorts) but aligned all structures to a word boundary to increase performance when accessing these items.

Source: `http://www.aleph1.co.uk/chapter-10-arm-structured-alignment-faq`

Generally, for this and subsequent questions, I trace the path of execution of what happens from the moment the kernel creates the FirstUserTask, to its activation. Then I figure out for each of the execution step, what would've been an access to the data cache.

Execution steps.

1. Access task descriptor

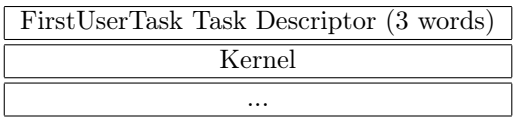

Figure 1: (1.a.ii) Data cache when FirstUserTask has just been created and just activated for the first time.
Each box represent a line of a cache, and each line contains 8 words.

(iii) Repeat of (ii) but assuming as many arguments and variables local to the kernel as you think is reasonable.

I think reasonable number of arguments/variables local to the kernel is probably around 5, as that is how many our kernel main loop had.

Execution steps.

1. Creating local args/vars.
2. Access task descriptor.

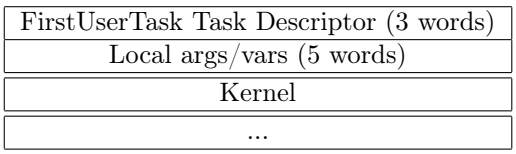

Figure 2: (1.a.iii) Data cache, FirstUserTask created activated for the first time, with local args/vars.
Each box represent a line of a cache, and each line contains 8 words.

(iv) Assuming tasks have no arguments and no local variables, how many task descriptors can be squeezed into the data cache under best conditions.

The task descriptor is 3 words, and a line of cache is 8 words. At most two task descriptors can fit in one line, using 6 words. Using

1.b On the stack of the kernel

For this question, some assumptions on the implementation has to be made.

- The kernel always pushes the most recent task descriptor on the top of its stack. So this means:
  - Context switching in on a user task A, task descriptor A always gets pushed on the stack.
  - Context switching to another user task B, there is a scan down the kernel stack, and the most recent task descriptor B is used. That task descriptor used for B is not popped off.
  - Note: consequently these assumptions means the kernel's stack overflows eventually if more than 1 user task runs.
- It makes no attempt to replace the already-existing task descriptor, if any[6].
- The user task registers' are saved on the user task's stack. It is not stored in the task descriptor.
- The kernel schedules the top most task descriptor to be run next.
- Destroying a task is achieved by scanning down the kernel stack and marking any matching task descriptor as invalid.

(i) Draw the kernel stack immediately after FirstUserTask is created during kernel initialization and just after it has been activated for the first time.

Execution:

1. Kernel pushes FirstUserTask task descriptor on its stack.

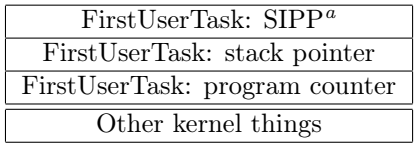

Figure 3: (1.b.i) FirstUserTask just created and activated.

[a] SIPP (reminder to earlier definition): refers to a word that contains the task's state, id, parent, and priority

[6] This setups for a great answer to (v) What happens to the long run, and (vi) Overcoming the problem (of exploding the stack). Whereby I introduce a method to scan down the stack and replace the existing kernel.

So because of how the question is structured, I will choose this dumb and naive method.

(ii) Draw the stack of the kernel just after FirstUserTask re-enters the kernel to Create the first task, First, in the program to be run, and again immediately after the task has been created.

Execution:

1. Kernel pushes FirstUserTask task descriptor on its stack.
2. Kernel re-enters to create the First task.

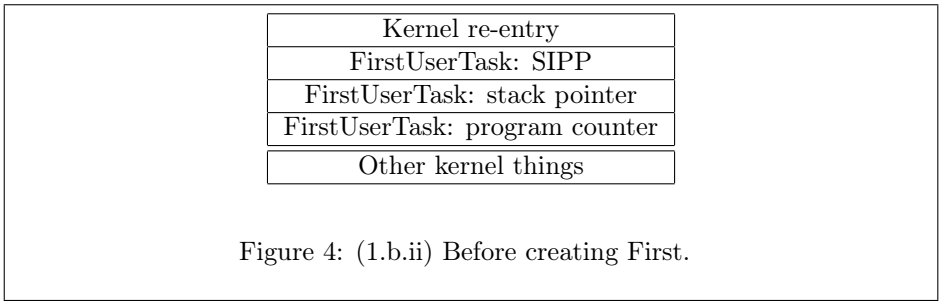

Figure 4: (1.b.ii) Before creating First.

Execution:

1. Kernel pushes FirstUserTask task descriptor on its stack.
2. Kernel re-enters to create the First task.
   (a) The kernel was already setup, so the only reason to go back to the kernel is either to handle a syscall, or to context switch. I'm going to assume the implementation doesn't leave things on the kernel stack.
3. Immediately after, the First task has been created.

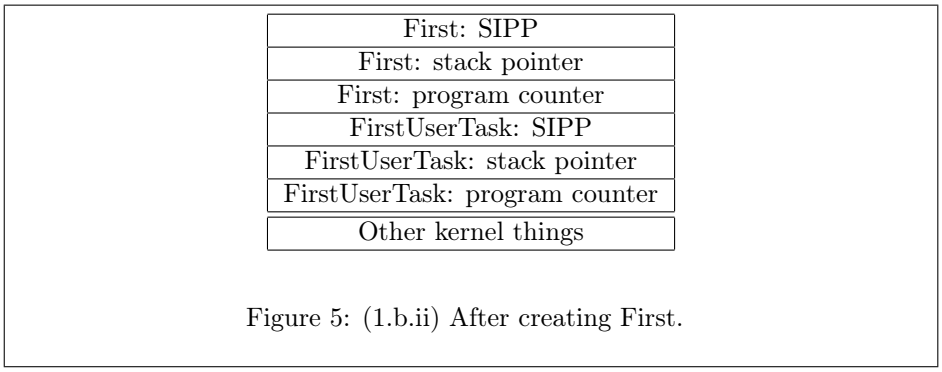

Figure 5: (1.b.ii) After creating First.

(iii) Assuming that First is the same priority as FirstUserTask, show the stack immediately after the next task activation.

Execution:

1. Kernel pushes FirstUserTask task descriptor on its stack.
2. Kernel re-enters to create the First task.
3. Immediately after the task has been created
4. Activate First task. The kernel's stack still contains the First task descriptor because the stack is never popped (see assumptions).

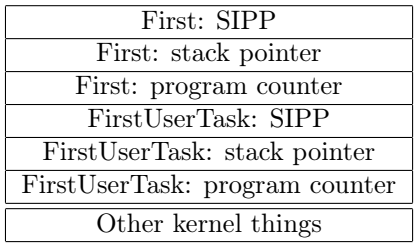

Figure 6: (1.b.iii) First just activated.
First task is not on the kernel stack because it is running (kernel isn't running).

(iv) Suppose that whatever task was activated creates Second, equal in priority to First. Draw the kernel stack, before and after creation.

Execution:

1. Kernel pushes FirstUserTask task descriptor on its stack.
2. Kernel re-enters to create the First task.
3. Immediately after the task has been created
4. Activate First task, removing it from tip of kernel stack.

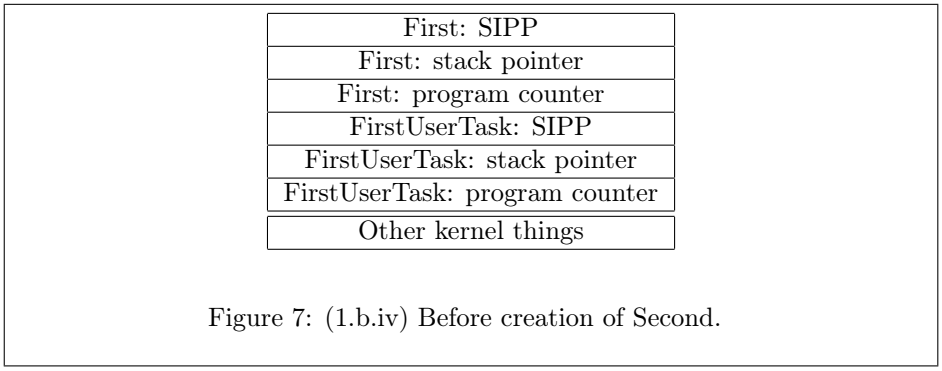

Figure 7: (1.b.iv) Before creation of Second.

Execution:

1. Kernel pushes FirstUserTask task descriptor on its stack.
2. Kernel re-enters to create the First task.
3. Immediately after the task has been created
4. Activate First task, removing it from tip of kernel stack.

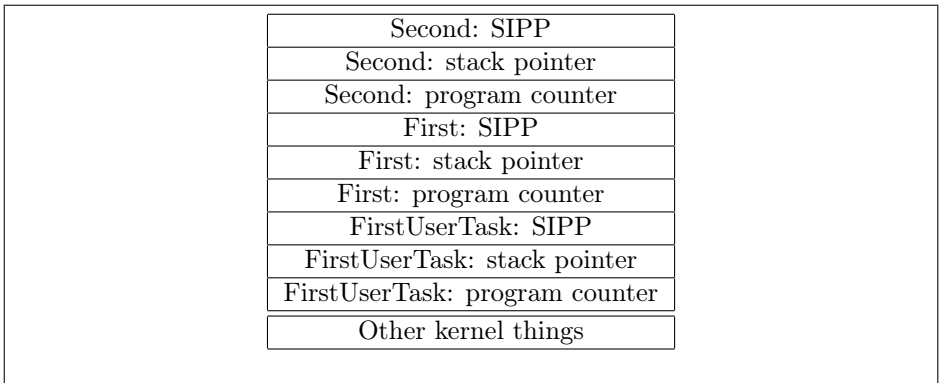

Figure 8: (1.b.iv) After creation of Second.

(v) What happens in the long run if no more task creation occurs? If much task creation occurs? If much task creation and destruction occurs?

- When **no more task creation occurs, the kernel stack keeps growing in size** but in the **replace-if-same-task-descriptor strategy**[7]**, the**

[7]The replace-if-same-task-descriptor method: If upon coming in from a context switch, the tip of the kernel stack has a matching task descriptor id, then you replace the tip with a

stack stops growing in size when there is only one user task. Because stack is not popped.

  - **Context switching in:** always puts their task descriptor on the top of the stack
  - **Context switching out:** scans down the kernel stack for the most recent task descriptor for that task, and use that.

- When **much task creation occurs, the stack keeps growing**. Just faster.
- When **much task creation and destruction occurs, the stack keeps growing**, a lot.
  - Destroying a task is made by scanning downward and marking it as invalid, but not popped.

(vi) Can these problems be overcome?

Yes, with these methods.

1. Instead of a naive strategy of always pushing onto the stack, use a **find-and-replace strategy**. That means, scan down the kernel stack for the matching task descriptor, and update its content with the user task that is being context switched out.
   (a) This means on the kernel stack, all of the task descriptors are unique.
2. A table mapping task and their respective storage (which otherwise would've been on the kernel's stack) can be set aside.
   (a) It can be on kernel's stack, then the kernel could use a preallocated chunk of space (from its stack). Benefit would be the kernel's stack is not moving up and down.
   (b) It can be in static space, pre-allocated by the compiler. This would mean it's not going to be on the kernel's stack. However, this solution is exactly 1.a, so it would not be acceptable.

1.c In my kernel

(i) Some task state must be kept in the kernel. How much was kept in your task descriptor?

This is my task descriptor in my kernel.

updated task descriptor. This method will stop the kernel stack from growing larger, when there is only one user task.

It's not much better, but I guess it's worth mentioning, given the type of exam this is.

```
typedef struct TaskDescriptor {
    int id;
    int parent_id;
    unsigned int *sp;
    enum { ready, send_block, receive_block, reply_block, zombie, } status;
    int *send_id;
    void *send_buf, *recv_buf;
    unsigned int send_len, recv_len;
    struct TaskDescriptor *next;
    char name[TASK_MAX_NAME_SIZE];
    unsigned int cpu_time_used;
    int originalReceiverId;
} TaskDescriptor;
```

State of the tasks were implicitly tracked in the kernel by moving around the pointer to the current task descriptor onto different queues.

- Scheduling system had 32 ready queues (thus 32 priorities). Using a machine word (32 bit), the state of whether a queue contained any waiting tasks were recorded, and the highest priority task scheduled in off the highest bit set.
- Delayed queue for any tasks that called the Delay syscall, which was a linked list sorted by the closest up-coming delay first.
- Send queue for tasks that had to do Send() message passing.
- Task descriptor contained a task descriptor pointer called *next* so a daisy-chain of waiting tasks can be queued up. This was for checking whether there are senders in the receiver's send queue, and receivers waiting for the send queue. In other words, if a task wanted to talk to another task, by way of Send() or Receive() or Reply(), then they would be queued up on the *next* field of their desired task.

In my kernel, the task descriptor had only these following fields set during a context switch.

- id,
- parent id,
- stack pointer,
- status.

The kernel did not explicitly track where the task descriptor is, in terms of what queue for any task descriptor is currently on.

Additionally, if the task was doing a syscall for interprocess communication, the memory locations were stored in the task descriptor for

- send id pointer,
- send buffer pointer,
- receive buffer pointer,
- send length,
- receive length.

(ii) For each item of state in your task descriptor, why did you include it?

Information that is changed often and task related are generally included in the task descriptor for easy access and modification. Generally, each item in the task descriptor has something to track about that task, and putting it in the task descriptor makes for better logical data organization (instead of the alternative of not putting it here and tracking these data in another data structure in the kernel, such as a table). For each item, I will comment on why it is included.

id, parent_id; associating a unique id with a task descriptor allows kernel to easily identify which task descriptor this is. Parent information stored here for easy tracking.

***sp, status;** Context switching stores the stack pointer here, and may modify the task status, putting it on other task queues. If it weren't stored here, kernel would've needed additional tracking of task data.

***send_id, *send_buf, *recv_buf, send_len, recv_len;** Interprocess message communication uses these. Set when needing to communicated between some tasks. Used in conjunction with putting the task descriptor in the right queue.

***next;** A pointer to other task descriptors allows daisy-chain of task descriptors, for constructing a singly linked list. Message passing

***name;** This was the only field that is not used. It was supposed to be for identifying the task to have human readable names.

cpu_time_used; Each time a task is scheduled in to be run, to its interruption, the time is added here so an idle time can be calculated by summing all the cpu time all tasks used, and taking a ratio of how much time the idle task ran.

originalReceiverId; This is used for message passing, its purpose is to make sure a sending task can only pass a message to the intended recipient.

(iii) In retrospect would you change what's in your task descriptor? If so, how? If not, why not?

These are few things I would change.

- The message passing items cluttered the task descriptor, and should be
  - Either be combined into a new structure to have the task descriptor look better organized.
    For example, the new structure could enclose all of these:

    ```
    struct {

        int *send_id, send_len, recv_len, originalReceiverId;
        void *send_buf, *recv_buf;
    } ipc_info;
    ```

  - Or have message passing be tracked elsewhere and not in the task descriptor.
    One method of doing this is to simply push all this information on the task's stack during the message passing syscall, and modify the status field to send/receive/reply blocked. To use it, the kernel can access from that task's stack. And to remove it, simply move the stack pointer back.
- Name isn't used and can be removed. Originally intended for debugging use, and logging. But our debug messages and asserts contained which file and line the problem originated from, so that was good enough. And I didn't do logging.

Everything else I would leave as-is, because storing those information in the task descriptor makes logical sense – example: I would expect to find the task id in the task descriptor, due to it being relevant data only to that task.

Question 2. Debugging

2.a. Symptoms of bugs.

Consider the sequence of steps in debugging a bug. At each step of debugging, a difficult bug hides itself and makes it difficult for you to proceed to the next step of debugging. The hardest steps of fixing bugs are generally [8] to:

1. **Error identification.** The expected behaviour is wrong, and it is reproducible.
2. **Error location.** Once an error is known to exist, figure out the minimum set of code that is responsible.

Each of the eight qualities given in question associates with somewhere in the bug fixing step. Characteristics have an association with one (or more) of the debugging steps. I explain, with examples, why each of the eight characteristics are associated with ease or difficulty.

1. A **frequently** occurring bug is easy because it contributes to *identifying* the existence of an error.
   For example, incorrectly drawing the time on screen.
   Because the clock is updated so frequently, any input mistakes or formatting mistakes would be noticed immediately.
2. An **infrequently** bug is difficult to identify the existence of the bug.
   For example, for our nameserver, I had a bug in the WhoIs call handling. The nameserver would look through its registration for the requested name, and send a reply when it found it, then break. However, this break only breaks out of the local linear-scan, and not the case statement.
   This bug was hard to find because the WhoIs is called so infrequently (only during initialization code). The extra reply went unnoticed for a long time, until we noticed someone was sending an extra reply to our couriers.
3. A **predictable** bug is easy because it would be reproducible, allowing for easy *identification* and location of the bug.
   For example, a missing break statement in the parser. The deterministic finite state automaton simply goes to the next state, causing that command to fail to parse. So the same command keeps failing to parse, which is predictably buggy.
4. A **irregularly** occurring bug is difficult because the code does not get run often. This is related to an infrequent bug, because the irregular bug shows up when the infrequently running code (that contains the bug) runs.
   For example, races on input output to the train control module. Because

[8] Briefly summarized from: `http://www.makinggoodsoftware.com/2009/06/14/7-steps-to-fix-an-error/`.

the train control module is so slow (relative to the ARM processor), there may result in *multi-track drift*[9]. This is irregular because most of the time the train IO is timed correctly, but sometimes the location of the train is estimated incorrectly and a command was sent too quick.

5. An **immediate crash** bug is easy to *identify* because clearly something went wrong (crashed).
   For example, incorrectly pushing arguments onto a task's stack during a context switch can cause the task to crash upon switching back to that task's context. The result is usually immediate, as context switches happen very often.

6. An **non-immediate** crash bug is difficult to *identify* because it's not obvious *when* the bug was triggered, consequently, not clear *who* triggered it.
   For example, a buffer overflow. When the rate of consuming elements from the buffer is slower than the rate which elements are added to the buffer, the buffer overflows eventually. It takes time to fill up the buffer, so the crash is non-immediate. It's hard to identify this crash was due to an overflow, because it crashes later, so one has to back track many sequence of actions to find the source of the problem.

7. A bug producing **same symptoms** is easy to *identify* because a deterministic set of actions can be identified. So upon inspection of the sequence of actions, say using printouts, I can see where in the sequence the problems started to occur.
   This is related to predictable bugs, because having the same symptoms implies predictability. So the parser example can apply here.

8. A bug producing **different symptoms** is difficult to *identify* because the same actions repeated over and over may or may not reproduce the bug.
   For example, a context switch bug. We had this bug, unknowingly. It only caused a problem once in a while, but each problem is always different. All the code we had seems to break, arbitrarily. It was extremely difficult identifying the context switch bug, because of non-deterministic behaviour. We even got our kernel to run for two minutes without seeing a problem.

2.b IRQ bugs.

Description of typical behaviour between instruction pair and kernel bug occurrence and a subsequent crash.

This is a bug that occurs only when:

- it is between a pair of user instruction (during context switch), and

[9]Multi-track drift occurs when a train travels over a switch. The front wheels of the train passed the switch, but before the back wheels pass the switch, the switch changes direction. This causes the front and back wheels to be on both sides of a branch.

- a hardware interrupt occurs between the pair.

I will describe a bug that occurs when one user process interferes with the control flow structure of another only when a hardware interrupt occurring during their context switch.

This is the setup[10][11].

- Suppose the bug is in the context switch, where the CPSR is not saved properly.
- There are two programs A and B.
  - They have the same code.
  - They run forever to do arithmetic and checks it along.
  - Hence modifying their state register (CPSR)[12].
  - The programs themselves run normally without preemption.
- The hardware interrupt is by the watchdog timer, causing preemption to the other program.

Here is sequence of execution that would crash the user programs.

1. Program A did some computation and stored its ALU result in the CPSR.
2. Hardware interrupt from watchdog timer causes preemption context switch into program B.
   (a) CPSR of A was stored correctly onto its stack.
3. Program B did some computation and store its ALU result in the CPSR.
4. Hardware interrupt from watchdog timer causes preemption context switch into program A.
   (a) CPSR of A is supposed to be restored correctly. But it wasn't, due to bug.
5. Program A is now actually continuing on, using program B's CPSR result.
   (a) Program A check fails here.

[10] This is a boiled down version of what our actual context switching bug was in our kernel. We were restoring the user's CPSR to the supervisor instead of the user. And this was fine, until hardware interrupts were added (watchdog timer preemptions).

[11] "Two such possible bugs were described in class." – I actually don't remember this. I apologize if my example is the same as the one in class.

[12] The CPSR contains condition code flags: N, Z, C, V from the 31:28 bits. These store the result of ALU operations.

2.c Link register bugs.

There's actually 6 link registers, one for each of the 6 modes[13]: User/System, FIQ, IRQ, Supervisor, Abort, Undefined.

I am going to assume the **3 registers** refers to User, Supervisor, and IRQ modes (we do not use FIQ).

The Supervisor, and IRQ modes are both privileged, while User mode is not.

The function of the link register is to store the return address when branch and link operations are performed, calculated from the program counter. It is also known as register 14 in the ARMv4 architecture.

For each of the 3 link registers, I describe its function, where and when I get it, and where and when I put it back:

1. User link register
   (a) function is to track the current running user program.
   (b) get it from user link register and put it on user's stack when going into context switch.
   (c) put it back to user link register (from user's stack) when coming out of context switch.
2. Supervisor link register
   (a) function is to store the kernel's link register.
   (b) get it from the kernel stack when kernel becomes active.
   (c) put it back to the kernel stack when going to user mode.
3. IRQ link register [14]
   (a) function is to store the user pc.
   (b) get it from the IRQ register in IRQ mode, during IRQ interrupt handling.
   (c) put it back to the user's stack, so that the kernel can use it to restore the user task.

There are three ways to pair up a set of three things. For each possible pair, I explain the future execution of the program when that pair of registers is interchanged.

1. User/Supervisor pair
   (a) **Supervisor resumes execution of User's link register.** This would work, because being in privileged mode, user mode code can definitely be executed. Of course, assuming the user's link register didn't contain garbage.

[13] Source: `http://www.eecs.umich.edu/courses/eecs370/eecs370.w15/resources/materials/arm_inst.pdf`

[14] Goes into kernel mode after a hardware interrupt, in my kernel.

   (b) **User resumes execution of Supervisor's link register.** which causes access violation, breaking the abstraction barrier. Abort data [15] is triggered.

2. Supervisor/IRQ pair

   (a) **Supervisor resumes execution of IRQ's link register.** This would work, because both are privileged modes. In my kernel, it always goes from IRQ mode into Supervisor mode, and we abuse this resumption of execution. Of course, assuming the IRQ link register didn't contain garbage.

   (b) **IRQ resumes execution of Supervisor's link register.** This would work as well, both are privileged modes. Of course, assuming the Supervisor's link register didn't contain garbage.

3. IRQ/User pair

   (a) I**RQ resumes execution of User's link register.** This would work, for the same reason why Supervisor can execute User's link register.

   (b) **User resumes execution of IRQ's link register**. This would *not* work, for the same reason why User cannot execute Supervisor's link register (access violation).

[15] Abort data exception is a response by a memory system to an invalid data access. Source: `http://www.embedded.com/design/prototyping-and-development/4006695/How-to-use-ARM-s-data-abort-exception`

Question 5. Destroy

5.a. Destroy()

(i) List all the internal adjustments that the kernel must do to ensure consistency with the task never running again.

The kernel must ensure the task is not run again, and recycle any resources the task used.

Here is a list of the resources a task uses.

- Stack. Whatever memory that was reserved for the task's stack can be marked as not-used. Destroying this simply marks the space as free.

(ii) How is Exit() is related to Destroy() and Suicide()?

Exit is similar to Destroy and Suicide because it is also a syscall that tells the kernel that the resources given to this task can be freed up and recycled.

5.b. Suicide()

(i) Describe when Suicide() could work and when it can't.

Suicide(), implemented as the idle task doing a Send() message to the desired task to be removed, would only work if the task

- does Receive() and
- listens for a suicide message and
- it is possible for the said task to be receive blocked (and not, say, in an infinite loop).

Suicide() does not work unless all the above requirements are met[16].

(ii) Assuming Destroy() to be available how would Suicide() be implemented?

Assuming Destroy() is available, suicide can be implemented as a goto the cleanup section of the code, doing whatever cleanup necessary, and calling Destroy() on any children it may have created and wish to destroy, and Exit() itself. Pseudocode below.

```
FOREVER {
    Receive(sendertid, message, sizeof(message));
    switch(message.type) {
    case SUICIDE: goto cleanup;
    }
```

[16] Example: does not work if task is a courier, doing only Send() calls.

```
}
cleanup: // a label
Destroy(childtid); // destroy any children it may have
Exit();
```

5.c. Children

(i) When is destroying the children is a good idea?

If the children contains any references to the task that was destroyed, then the children no longer operates in a valid context and should be destroyed as well.

For example, a child task that replies to the parent with some computation result the parent had asked. This child should be destroyed, because it would be trying to reply to a nonexistent task after its parent was destroyed.

Suppose we use a static variable to keep track of the task id the child task should reply to. If the static variable was reassigned after the destroy, it would not a good idea to that static variable replying, because this introduces race conditions. Even if successfully executed, the content of the reply message might no longer be valid. Furthermore, the task might not be re-created, in which case the child (who was not destroyed) is blocked forever.

(ii) Give an example of a task configuration when destroying the children is not desirable.

If the child task is expected to operate independently of whether the parent exists or not[17], then it is not desirable to destroy the child.

For example, a task may be created and used as an in-memory-database, to sequentialize read and writes (say, to the track data structure), then it makes no sense to destroy this task simply because its parent was destroyed.

(iii) Because your system has an upper limit on the number of live tasks, you can bound above the amount of time required to permanently inactivate a task and its children. Calculate the worst case running time for inactivating a task and its children.

The worst possible run time is the result of calling destroy on the root task. Suppose all the tasks are created in a tree (graph) manner. Then calling destroy on its root means destroying all the tasks in the entire tree. The size of the tree is bounded to be the maximum number of live tasks m.

I argue the worst setup occurs when each task creates a child, whereby destroying that task incurs a destroy to its child, causing a daisy-chain of destroy syscalls. I argue this is worse than the other limit-case, of destroying m tasks individually, and each task is not related to others.

My intuition is there is an reduction in the task search space – unless the task id lookup to its task descriptor entry is via a hashmap (or some other constant

[17] Consequently the children must not refer to the parent task.

lookup method[18]). Because calling destroy on a child requires the kernel to lookup the task descriptor, it is more costly to do m lookups than none at all, as in the case of each task not related to others.

In the worst case, having to do lookups, for each of the m tasks there could be a potential message passing overhead to pass a message to its children.

$MaxTimeRequired = m(MaxDestroyTime + MaxMessagePassingOverhead)$

- $MaxDestroyTime$ is the maximum time to destroy a task. The time it takes to destroy a task is the cost of marking the stack that used as free-to-use.
- $MaxMessagePassingOverhead$ is a bound on the lookup time to find the task descriptor corresponding to the child's task id.

5.d. Resource reclamation

(i) Some resource reclamation can be done by the kernel. What are the resources and how are they made available for future use?

Resources that tasks themselves cannot keep track of should be reclaimed by the kernel. These resources could be the

- stack memory space for a task that was destroyed, or
- tasks that might be blocked on a message passing to the task that was destroyed – in which these tasks can now be considered dead, since they'll never be unblocked (unless the task destroyed was destroyed was replaced with another of the same task id, or the kernel could wake up those blocked tasks and fail their syscall).

The freed up resources can go into a queue, so to keep track of what resources are available to assign to newly created tasks.

(ii) If memory is to be zeroed, you probably don't want the kernel doing that. How might it be done?

Indeed having kernel zero (a possibly large chunk of) memory is undesirable due to real-time constraints, so to be out of the kernel mode as soon as possible.

- A zeroing task can be created initially by the kernel and thus be invisible to user tasks (or the first-user task can create it).
- Optionally, the idling task can be given the duty of zeroing, but this could be undesirable because of its low priority, because we might want the zeroing to happen as soon as possible so that task can created faster.

This zeroing task would be blocked and waiting for a message of what address to zero (and how long) passed to it by a kernel.

[18] In my kernel, task descriptors are stored in a table, and the task-id stored in the task descriptor is bit masked together with the index into that table, thus no lookup to the task descriptor index is required.

(iii) There are resources about which the kernel knows nothing, track reservations for example. How might these resources be freed?

These resources could be kept in the stack of a separate task, for the sole purpose of acting as a simple in-memory database. Destroying the task is the equivalent of removing the resource.

For a large structure such as the track data, it would be useful to extend a new form of the Create() syscall to also take a desired a stack size to be allocated[19].

Keeping resources in a task has the added benefit of atomicity guarantee, since reads/writes can to be serialized using the Send/Receive syscalls.

[19] And better memory management. In my kernel, memory is chunked into 128 equally sized pieces.