University of Toronto, Faculty of Arts and Science
Department of Computer Science

CSC 372 – Microprocessor Software

Laboratory 3: Context Switching

Objective

The objective of this lab is to implement kernel code for context switching. Through this exercise you should come to understand what a context switch really is.

Specification

The lab consists of two parts: (i) the "user-level" part and (ii) the "kernel-level" part. The two parts should be fully separated in that no memory data structures are shared between the two. TRAP and ERET are used to switch between the two modes. Since you are not allowed to use shared memory, can you think of another way these two parts can communicate? Note that there are usually only a few parameters that have to be transferred between these two parts.

In the kernel, you will need a number of lists to organize thread descriptors that are defined as follows:

```
typedef uval32 char  uval8;
typedef uval32 unsigned int uval32;
typedef uval32 short uval16;

typedef uval32 ThreadId;

struct TD
  {
    struct TD *link ;
    ThreadId tid ;
    struct Registers regs ;
    uval32 priority;
    uval32 returnCode ;
    LL *inlist ;
  };


struct Registers {
  uval32 sp;
  uval32 pc;
  uval32 sr;
};
```

In the TD structure, the various fields have the following meanings:

- ***link*** is used to point to the next TD in whatever queue the TD is in;
- ***tid*** is a unique number that can be used to identify a thread once it has been created;
- ***regs*** is a structure used for saving CPU registers and other CPU state when the state of the thread needs to be saved;
- ***priority*** holds the current priority of the thread by convention in systems software (particularly for UNIX), the higher the number in Priority the lower the priority (importance) of the thread
- ***returnCode*** is used to temporarily hold the return value of a system call; and
- ***inlist*** identifies the queue that the thread is currently in.

You will notice that this structure is similar to the structure you used in lab 1. Minor modifications to the code from lab 1 are needed, because the structures are not identical, but majority of the lab 1 code should be the same.

You should allocate a fixed size array of TDs, instead of using malloc to allocate space for the TDs at run time.

When a new thread is created, it should be assigned a unique identifier, stored in the ***tid*** field of the TD. Make sure that you do not re-use tids for as long as possible. Not reusing the old tids (as much as possible) is a normal design practice to avoid confusion in case tids are saved locally somewhere. If a thread doesn't exist anymore, its tid will be simply "invalid", and hence any activity on it (e.g., sending a message to it) will simply return an error, as opposed to applying the operation on a wrong thread (the newly created one).

In the kernel you will need to include:

- ***Active***: contains the actively running thread. This is just a pointer to a TD.
- ***Kernel***: contains the kernel's stack pointer, default status register, and program counter of system call handler. Used to enter/exit to/from system calls.
- ***ReadyQ***: contains the TD's of all threads that are ready to run, **ordered by priority**. When a thread is entered into the list, then it is to be positioned **after all TD's with higher or equal priority.**
- ***BlockedQ***: contains the TD's of all threads currently blocked. See explanation of `Suspend ()` below for explanation of what it means for a thread to be blocked.
- ***FreeQ***: contains all TD's that are currently unallocated. You have an array of **x** thread descriptors, and not all of them will be always be used. Any descriptor that is not used should be placed into this queue, so that they are easily accessible when a new descriptor is needed.

At the user level, you will need to provide the following functions an application may want to call:

- ***CreateThread(uval32 pc, uval32 stackSize, int priority)***

  Creates a new thread that should start executing the procedure pointed to by ***pc***. Creating a new thread should be done by first allocating a stack at the user level (***using malloc()*** with a minimum size of 8K). At the kernel level, this should cause a new thread descriptor to be allocated, its fields to be initialized and the descriptor to be enqueued in the ReadyQ. If the new thread has higher priority than the invoking thread then the invoking thread should yield the processor to the new thread.

  CreateThread() should return the thread Id of the new thread,
  RESOURCE_ERROR of there are no thread descriptors available,
  STACK_ERRROR if ***stackSize*** is not at least 8K large, and
  PRIORITY_ERROR if priority is not in the range of valid priorities.

- ***DestroyThread(ThreadId tid)***

  Destroys the thread identified by ***tid***. If ***tid*** is 0 or the same as that of the invoking thread, then the invoking thread should be destroyed. (Note that this means that no thread should have a tid of 0, otherwise no other thread could destroy it). In the kernel, this can be achieved by removing the thread descriptor from whatever queue it is in and adding it to the list of free descriptors. If the Active thread is to be killed, then a new thread should be dispatched.

  DestroyThread() should return TID_ERROR if there is no thread corresponding to ***tid***, or OK otherwise.

- ***Yield()***

  The invoking thread should yield the processor to the highest priority ready-to-run thread with equal or higher priority. In the kernel, this is achieved by enqueueing the Active thread onto the ReadyQ behind all threads of the same, or higher, priority (remember that the higher a thread's Priority number, the lower its priority), and dispatching the ready-to-run thread with the highest priority (which happens to be at the head of the ReadyQ).

  Yield() should return OK.

- ***Suspend()***

  The invoking thread should block until it is woken up again. This can be achieved by enqueueing the Active thread onto BlockedQ, and dispatching the ready-to-run thread with the highest priority.

Suspend() should return OK.

- ***ResumeThread(ThreadId tid)***

  Wakes up the thread identified by the ***tid*** and makes it ready to run. If that thread has higher priority than the invoking thread then the invoking thread should yield the processor.

  ResumeThread() should return TID_ERROR if there is no thread with Id ***tid***, NOT_BLOCKED if the target thread is not blocked, and OK otherwise.

- ***ChangeThreadPriority(ThreadId tid, int newPriority)***

  Changes the priority of the target thread identified by ***tid*** to ***newPriority***. This can be achieved by setting the priority field of the thread descriptor to ***newPriority***. If the corresponding TD is in the ReadyQ, then remove and re-insert it, so that the ReadyQ remains sorted according to Priority. If the target thread now has higher priority than the invoking thread then the invoking thread should yield the processor to the target thread.

  ChangeThreadPriority() should return TID_ERROR if there is no thread with Id ***tid***, PRIORITY_ERROR if ***newPriority*** is not valid, and OK otherwise.

These functions will be used by user programs to run multithreaded programs, similar to the way you run a multithreaded program in a modern operating system such as Linux. As you can see, you are developing an operating system kernel for an embedded system.

When you run the program used in the lab, by default, the program runs in supervisor mode. We will use this fact to initialize the kernel before running user applications. `main()` function provided on the website is kernel code that prepares the system for user applications. User programs start from a function called `mymain()` and is called by the kernel once the kernel is fully initialized, and the processor is switched into user mode.

Download the files from the website for a demo of one functioning system call. Note that this code will only allow for system calls between one user thread and the kernel.

For the lab, in addition to any worker threads you create, add a thread called "idle" that always exists, is always ready to run, and is assigned the lowest priority.

```
void
idle()
  {
      {
         …
         …
         Yield() ;
      }
  }
```

The code for one system call is given in the files downloadable from the website. Extend it to implement other system calls in this lab and to provide their desired functionality.

A few comments that describe this code are in order. First, at the beginning of ***K_SysCall()***, a label points to the beginning of the function, skipping the usual stack allocation done to store the return address. Since your code will not perform a proper return from this function (which would pop the return address off the stack), but rather exit via a TRAP, that code needs to be skipped. Otherwise for every invocation of the system call, kernel stack space would be wasted and eventually run out.

However, by skipping the stack initialization code, stack space for local variables is also not allocated. Since the kernel stack pointer goes to the top of the byte array, the system call handler would trash memory not allocated for the stack. If you have local variables in the ***K_SysCall()*** function, as the code on the website does, then you need to adjust the default stack pointer for the kernel before jumping to this label. The **SYS_HANDLER_OFFSET** variable is used to decrement the kernel stack pointer before jumping to the system call handler. If you add more stack variables to the system call handler, disassemble your code and adjust this variable to reflect your scenario.

Secondly, the functionality for creating a thread is split up into two functions, indirectly through ***K_SysCall()*** and directly through ***CreateThread().*** You should design your code such that threads with supervisor privileges access ***CreateThread()*** directly, whereas user threads perform the aforementioned system call.

Third, note that ***K_SysCall()*** may change the thread that is currently active (if the newly created or dequeued thread has higher priority than the calling thread). In this case the kernel will need to enqueue the old active thread and replace it with the higher priority thread. This is what effectively implements the context switch.

NOTE:

In order to be marked you will need to display to the TA that your kernel is capable of managing several threads. Keep in mind that user-level threads are not allowed to read or write to control registers, yet they will execute the bulk of your code. Therefore in future labs you will need to add system calls to properly configure hardware devices to generate interrupts like you did in lab 2. Consequently design your code such that adding new system calls is trivial.