Project II: Designing a Thread Scheduler for your Prototype OS

1 Deadline

See course's homepage.

2 Objectives

In this project, you will design and implement a thread scheduling subsystem to allow our prototype OS to create, schedule, maintain, and terminate multiple threads on an Altera board that only supports single-thread execution. You also need to provide the basic execution statistics for each thread.

3 Tasks

You need to complete at least six specific tasks in this project.

1. You need to create a *thread control block* (*TCB*) structure to maintain the information of a thread in your system. Such information includes thread_id, scheduling status, execution information, and execution context. The space for storing a threads context **must be allocated by using *malloc* that is provided by the Altera standard C library.**

2. You need to create two functions, *mythread_create* and *mythread_join*, and use them to create 12 TCBs to support 12 running threads (with the same function body mythread) and make them ready to run respectively (see Table 1). **Note that *mythread_join* must also be implemented.** It is used to suspend the main thread while other created threads are running. Once all the other created threads have died, *mythread_join* would allow the main thread to continue its execution.

3. Based on your work in Project I, your timer interrupt handler can be modified into a thread scheduler (see Table 1). The scheduler should have a run queue that runnable threads can wait to be scheduled. The scheduler is periodically invoked by the timer interrupt. If there are still runnable threads in the running queue, the scheduler preserves the context of the current thread and restores that of the next one to be executed. Which thread will be executed next depends on your scheduling policy. We will extend the existing interrupt handling mechanism already provided by Altera to accomplish this task (in `alt_exception_entry.S`).

4. As shown in Table 1, since two instances of *mythread* can perform a different amount of work, they won't finish at the same time. However, after a thread finishes, it is required that it should be terminated with its TCB destroyed. This can be done right away (per thread) or deferred until all threads have died before being destroyed. Prior to destroying a TCB, print out its execution information (number of times it was scheduled and the time on the CPU for those who attempt the extra credit) as a way to show that the destroy code is executed.

5. You need to keep track of the number of times each thread has been scheduled. The information should be kept on the TCB (required). **Extra Credit:** In addition, you can earn up to 5 additional points if you can keep track of the amount of time each thread is spent running on the processor. To do this, you need to find a way to keep track of time a thread is on the processor each time it is scheduled. The total time is the accumulation of this time.

6. After all threads finish and their TCBs destroyed, your program has to resume the execution of *prototype_os()*.

4 Constraints

1. The body of *mythread* function is given in Table 1 and CANNOT be modified.

2. The *mythread* function should NEVER be explicitly called anywhere.

3. *prototype_os* can ONLY be called in main, and its context should NOT be kept in the running queue of the scheduler at any time.

4. File `alt_exception_entry.S` and other Altera default library files can't be modified.

5 Useful Hints

- The programming of this project will involve a certain amount of inline assembly to manipulate the frame pointer (fp), stack pointer (sp), exception return address (ea), regular return address (ra), estatus, r2 or r4, etc. Correctly using them will be critical to switching context. You will need to carefully decide what a thread's context should include by investigating the disassembly code of your C/C++ functions. You will also need to provide atomicity for *mythread_sheduler()*, meaning that another timer interrupt cannot occur while *mythread_scheduler()* is running. The following two macros can be used to enable and disable interrupts.

```
// disable an interrupt
 #define DISABLE_INTERRUPTS() {  \
     asm("wrctl status, zero");  \
 }
 // enable an interrupt
 #define ENABLE_INTERRUPTS() {   \
   asm("movi et, 1");          \
   asm("wrctl status, et");    \
 }
```

- How to initialize the TCB for each thread: the context space need to be dynamically allocated (*malloc*) during thread initialization and liberated (*free*) when the thread dies. You will need to analyze the disassembly of *mythread()* to get to know the layout of its context.

- How to create and initialize threads: as stated in the constraint requirements, the thread body *mythread()* cant be explicitly called anywhere. In order to start an initial execution of thread *mythread()*, you can set up the threads initial context as if it got interrupted and its context preserved. Then, the thread can get executed when its context is restored by the interrupt handler routines.

- The strategy of switching contexts: in Project I, we have learned that *myinterrupt_handler()* can be used to handle the timer interrupts. However, it is not the only routine involved in interrupt handling. The whole interrupt handling process will roughly take the 6 steps as illustrated in Figure 1. In particular, when a regular program gets interrupted, the routine that first preserves and finally restores the context is in `alt_exception_entry.S`.

  - Between step 1 and step 2, code section between Line 75 and Line 150 in file alt_execption_entry.S will be executed to preserve the context of your current program/thread; between step 5 and step 6, code section between Line 295 and Line 352 in the same file will take effect to restore the context of the program/thread that currently gets interrupted. So, you need to carefully analyze `alt_execption_entry.S` to understand the process to preserve and restore the context of a program/thread.
  - Your strategy of switching context can be something like this: by intercepting the context restoration of the current program/thread, you can restore the context of your next program/thread instead. In order to do so, you can work around Line 295 - Line 352 in file `alt_execption_entry.S`. You cannot modify `alt_execption_entry.S` but you can instruct the compiler to automatically inject certain code (written by you) into Line 295 of `alt_exeception_entry.S` by following the steps: (1) create an assembly file `myScheduler.S` in your project; (2) at the very beginning of file `myScheduler.S`, add the directive shown below to guide the compiler to inject assembly code into Line 295 of file `alt_execption_entry.S`. Please be aware that the injected code won't overwrite the original assembly between Line 295 and Line 352.
    Note that line number may change in the newer or older edition of Quartus. The one used here is version 11.0.

```
.section .exceptions.exit.user, "xa"
/* if global_flag is se                    */
/*     call mythread_scheduler             */
/*     reset global_flag                   */
/* else                                    */
/*     follow the original procedure       */
/*     in alt_execption_entry.S            */
```

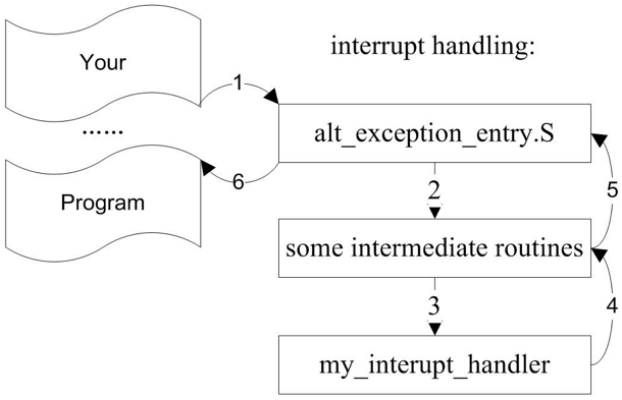

Figure 1. Steps in Interrupt Handling

6 Submission

You will submit a zipped file that contains the following components by the deadline:

1. A project report that includes:
   - the project title & names of the team members,
   - an introduction to your project goals, problem description and project management (including project timeline, teamwork, risk anticipation, resources, etc.),
   - your key ideas of solving the project problems, and
   - an elaboration of how you accomplish your project tasks (including data structures, algorithms, strategy, etc.) using diagrams, flowcharts, tables or descriptions.
2. A summary of:
   - what you have accomplished in this project,
   - how it can help you understand what you have learned in class,
   - your evaluation of the work that you have done (e.g., how creative your proposed idea is, how simple and sweet your design and implementation are, how well your project is organized, etc.),
   - how the course project and its specifications can be improved,
   - references: list the sources of citations appearing in your report,
3. You report should be formated as follows: 11pt Times-New-Roman fonts (the title and section headings can be larger); single-column; 1 inch margins all the way around a page; page numbers; no more than 8 pages in all; DO NOT paste source code with more than 10 lines; diagrams/charts/graphs with brief explanations are more preferable than thousands of words; submitted as a PDF file.

4. A folder containing your project source code with meaningful comments, as well as a README file that details the organization of your own source code (e.g., what each file is used for and how they function together) , as well as how to compile and run the program. Please DO NOT include any files or folders automatically generated by the NIOS-II IDE!

7 Grading Criteria

1. Project report: 30%
2. Correctness of the program: 65%
3. Detailed source code comments and README: 5%

Table 1. Description of Necessary Functions

mythread()	void mythread(int thread_id)
	{
	int i, j, n;
	n = (thread_id % 2 == 0)? 10: 15;
	for (i = 0; i ¡ n; i++)
	{
	printf("This is message %d of thread # %d.\n", i, thread_id);
	for (j = 0; j ¡ MAX; j++);
	}
	}

os_prototype()	void os_prototype()
	{
	// do all the necessary set up.
	for (i = 0; i ¡ num_threads; i++)
	{
	// ...
	// somewhere in here, you need to create threads.
	// ...
	}
	for (i = 0; i ¡ num_threads; i++)
	{
	// ...
	// somewhere in here, you need to join threads.
	// ...
	}
	while (1)
	{
	alt_printf ("This is the OS prototype for my exciting CSE351 course projects!\n");
	for (j = 0; j ¡ MAX; j++)
	}
	}

mythread_scheduler()	void * mythread_scheduler(void * context)
	{
	// do all the necessary setup.
	if (runQ_size > 0)
	{
	// suspend the current thread and schedule a new thread.
	}
	else
	{
	alt_printf("Interrupted by the DE2 timer!\n");
	}
	// do whatever you need to do.
	}

myinterrupt_handler()	alt_u32 myinterrupt_handler(void * context)
	{
	global_flag = 1;
	return ALARMTICKS(QUANTUM_LENGTH);
	}

mythread_create()	This function creates a TCB for a thread.
mythread_join()	This function blocks the main thread from continuing until joined threads have died.