15-410, Spring 2008, Homework Assignment 1.
Due Wednesday, February 27, 20:59:59 p.m.

You need complete only Question 3 and one other question.

Please observe the non-standard submission time... As we intend to make solutions available on the web site immediately thereafter for exam-study purposes, please turn your solutions in on time.

Homework must be submitted in either PostScript or PDF format (not: Microsoft Word, Word Perfect, Apple Works, LaTeX, XyWriter, WordStar, etc.). Submit your answers by placing them in the appropriate hand-in directory, e.g., `/afs/cs.cmu.edu/academic/class/15410-s08-users/$USER/hw1/$USER.ps` or `/afs/cs.cmu.edu/academic/class/15410-s08-users/$USER/hw1/$USER.pdf`. A plain text file (.text or .txt) is also acceptable, though it must conform to Unix expectations, meaning lines of no more than 120 characters separated by newline characters (note that this is *not* the Windows convention or the MacOS convention). Please avoid creative filenames such as `hw1/my_15-410_homework.PdF`.

1 PUSHA (10 pts.)

There is a problem with the definition of the `PUSHA` instruction as found on page 3-584 (PDF page 624) of `http://www.cs.cmu.edu/~410/doc/intel-isr.pdf`. The text claims:

> The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, EBP, ESP (original value), EBP, ESI, and EDI [...]

Somehow you suspect `EBP` isn't really pushed twice. One way to figure out what `PUSHA` really does would be to write some assembly code and run it in Simics.

1.1 5pts

Add code to the template below so that running it and inspecting memory appropriately would make it possible to see what `PUSHA` really does. You may assume that `pushy()` will be called by `kernel_main()` of a Project 1 game kernel.

```
.globl pushy

pushy:
...some stuff...
    PUSHA
...some stuff...
    POPA
...some stuff...
    RET
```

1.2 5pts

Now specify a small number of Simics commands you would use to make your decision. What would you type at the first Simics prompt? Then what? You should strive to provide us with a small number of deterministic commands and a brief description of how to interpret the output.

2 Safety in Numbers (10 pts.)

Consider a system with three processes and three tape drives. The maximal needs of each process are declared below:

Resource Declarations

Process A	Process B	Process C
2 tape drives	2 tape drives	2 tape drives

Assume the system is employing a naïve resource allocator as described in class.

2.1 4pts

Use the table below and the execution-trace format presented in the lecture slides to show a sequence of requests which would take the system into an unsafe state and back to a safe state without encountering a deadlock. Mark the unsafe state with an asterisk.

You may not need to fill in all lines of the table, and you may use more lines than we provide if you wish.

Execution Trace

time	Process A	Process B	Process C
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

2.2 6pts

Explain why the state you marked is unsafe.

3 Latches and Locks (20 pts.)

Assume you are writing code based on a Project 2-compliant thread library. Assume further that the implementor has taken a "hard core" approach to mutexes—they are designed under the assumption that they will be held only for *very brief* periods of time. Therefore the implementation techniques are chosen so that a mutex will be acquired very quickly if it is unlocked, and will still be acquired quickly in the assumed-rare case when it is found to be locked. You may wish to assume that this thread library is designed to run on multiprocessor systems.

Note that such a mutex is *not* appropriate for guarding access to objects which will remain locked for long periods of time. For example, it would not be reasonable for threads doing I/O to a file to use such a light-weight mutex, as disk I/O can block a thread for many milliseconds.

To solve this problem, define a new construct, `lock_t`. Start by filling in a structure definition:

```
typedef struct lock {
    ...
    ...
} lock_t;
```

Then show code for the following primitives:

```
int lock_init(lock_t *lp)    // initialize a lock pointed to by lp
int lock_acquire(lock_t *lp) // ensure mutual exclusion until a call to release()
int lock_release(lock_t *lp) // signals the end of mutual exclusion region
```

You need not show code for `lock_destroy()`.

You may use the Project 2 thread-library primitives (unaltered). Your implementation should be much more efficient than using mutexes (assuming locks are held for a long time), but you do not need to make the implementation as efficient as possible–a brief, reasonable implementation is sufficient.