CpSc 448B Final Exam Dec. 14, 2010
12:00 noon – 2:30 pm

Graded on a scale of 100 points (yes, I know that there is 1 extra point built-in; it's extra credit!).
Bug-bounties are in effect.

0. **(2 points)** Write your solutions to this question on the spaces provided and on your exam book:

   (a) **(1 point)** What is your name? ________________________________

   (b) **(1 point)** What is your student number? ________________________________

1. **(16 points)** Give a 2-4 sentence answer to each question below:

   (a) **(4 points)** What is the zero-one principle?

   (b) **(4 points)** What is the role of λ in the CTA model?

   (c) **(4 points)** What is a barrier?

   (d) **(4 points)** What is the difference between task and data parallelism?

2. **(35 points)** The following questions pertain to the Berkeley EECE technical report: *The Landscape of Parallel Computing Research: A View from Berkeley*. Answer each question with one or two paragraphs.

   (a) **(7 points)** What is the difference between a multi-core and a many-core processor? Which approach does *Landscape* promote and why?

   (b) **(7 points)** What are the 13 Dwarfs? What is the main property of a computation that classifies it as one dwarf or another?

   (c) **(7 points)** What role does *Landscape* propose that psychology should have in parallel programming and why?

   (d) **(7 points)** Does *Landscape* believe that CPU cores should exploit instruction-level parallelism (ILP) by continuing earlier trends of increasing the issue-width of superscalar architectures? State three reasons that they give for their recommendation.

   (e) **(7 points)** What is an autotuner? What role do the authors of *Landscape* envision for autotuners for the development of parallel software?

3. **(24 points)**

   (a) **(4 points)** What is an "empty/full" variable (e.g. in Peril-L)?

   (b) **(16 points)** Figures 1, 2, and 3 sketch an implementation of empty/full variables using pthreads. The methods `ef_create` and `ef_destroy` create and free empty/free variables. The method `ef_set` sets the value of an empty/free variable, and the method `ef_get` returns the value of an empty/free variable.

      Complete the definition of `struct EmptyFull` (Figure 1), and the functions `ef_create` (Figure 1), `ef_set` (Figure 2), and `ef_get` (Figure 2). Note that I have provided more blank lines in the figures than I used in my implementation – you don't have to use up all of the blank lines.

   (c) **(4 points)** Consider a situation where an empty/full variable is initially empty, three different threads attempt to set the variable, and then two other threads attempt to read the value of the variable. Briefly describe what happens.

4. **(24 points)** Figure 4 shows Peterson's mutual exclusion algorithm. To prove that Peterson's algorithm guarantees mutual exclusion, we can use the invariant:

$$\begin{array}{rcl} I & \equiv & \forall \theta \in \{0,1\}. \\ & & \quad \texttt{flag}[\theta] = (\texttt{pc}[\theta] \in \{5,6,7,8\}) \\ & & \wedge \quad (\texttt{pc}[\theta] = 7) \Rightarrow (\neg\texttt{flag}[!\theta] \vee (\texttt{turn} = \theta) \vee (\texttt{pc}[!\theta] = 5)) \end{array}$$

   (a) **(16 points)** Figures 5 and 6 sketch a proof that I is an invariant of Peterson's algorithm. Fill in the missing steps. You may write write on the figures and include them with your solutions to the other problems.

   (b) **(4 points)** Write a predicate, M, that specifies mutual exclusion for the code from Figure 4. Prove that Peterson's algorithm guarantees mutual exclusion by showing $I \Rightarrow M$.

   (c) **(4 points)** Consider a variation of Peterson's algorithm that exchanges the statements at lines 4 and 5. Show that this version does not guarantee mutual exclusion.

References

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "empty_full.h"    /* See Figure 3 */
#define TRUE 1
#define FALSE 0

struct EmptyFull {
     void *value;
     int full;

     ______________________________________________________________

     ______________________________________________________________

     ______________________________________________________________

     ______________________________________________________________
};

EmptyFull ef_create(void) {
     EmptyFull ef = (EmptyFull)malloc(sizeof(struct EmptyFull));
     ef->value = NULL;
     ef->full = FALSE;

     ______________________________________________________________

     ______________________________________________________________

     ______________________________________________________________

     ______________________________________________________________

     ______________________________________________________________

     ______________________________________________________________
     return(ef);
}

void ef_destroy(EmptyFull ef) {
     You don't need to write anything for this function.
}
```

Figure 1: Write your implementation of an Empty/Full variable here and on Figure 2

```
void ef_set(EmptyFull ef, void *value) {

}

void *ef_get(EmptyFull ef) {

}
```

Figure 2: Write your implementation of an Empty/Full variable here and on Figure 1

```
typedef struct EmptyFull *EmptyFull;

extern EmptyFull ef_create(void);
extern void ef_destroy(EmptyFull ef);
extern void ef_set(EmptyFull ef, void *value);
extern void *ef_get(EmptyFull ef);
```

Figure 3: `empty_full.h`

INITIALLY: (flag[0] = false) && (flag[1] = false) && (turn = 0);
ASSUME: The “user” code at lines **3** and **7** does not change the values of `flag` or `turn`.

```
 1:  forall(θ in (0..1)) {          /* create two threads */
 2:      while(true) {
 3:          this process does some work;
 4:          flag[θ] = true;          /* indicate intention to enter critical region */
 5:          turn = !θ;               /* yield if contested */
 6:          while(flag[!θ] && turn != θ);    /* spinning wait */
 7:          critical section;
 8:          flag[θ] = false;
 9:      }
10:  }
```

Note: the “!” in !θ is the logical negation operator from C. In particular, !0 is 1 and !1 is 0.

Figure 4: Peterson’s Mutual Exclusion Algorithm [Pet81].

Let

$$\begin{aligned} I_1 &\equiv \forall\theta \in \{0,1\}.\ \texttt{flag}[\theta] = (\texttt{pc}[\theta] \in \{5,6,7,8\} \\ Q(\theta) &\equiv (\texttt{pc}[\theta] = 7) \Rightarrow (\neg\texttt{flag}[!\theta] \vee (\texttt{turn} = \theta) \vee (\texttt{pc}[!\theta] = 5)) \\ I_2 &\equiv \forall\theta \in \{0,1\}.\ Q(\theta) \end{aligned}$$

Note that $I = I_1 \wedge I_2$. We'll first show that I_1 is an invariant by itself. Then, we'll show that $I_1 \wedge I_2$ is an invariant.

Proof that I_1 is an invariant of Peterson's algorithm:

- I_1 holds initially because $\texttt{flag}[0]$ and $\texttt{flag}[1]$ are both false and $\texttt{pc}[0]$ and $\texttt{pc}[1]$ are both initially set to 1.
- If $\texttt{pc}[\theta] = 1$, then executing the statement leaves the `flag` variables unchanged.
  Because I_1 held before executing the statement, it will continue to hold after executing the statement.
- If $\texttt{pc}[\theta] \in \{2,3\}$, then

  ____

  ____

  ____

- If $\texttt{pc}[\theta] = 4$, then executing the statement sets $\texttt{flag}[\theta] = \texttt{true}$ and $\texttt{pc}[\theta] = 5$ which establishes the clause
  $\texttt{flag}[\theta] = (\texttt{pc}[\theta] \in \{5,6,7,8\})$
  The clause
  $\texttt{flag}[!\theta] = (\texttt{pc}[!\theta] \in \{5,6,7,8\})$
  holds after executing the statement because

  ____

  ____

  ____

- If $\texttt{pc}[\theta] \in \{5,6,7\}$, then

  ____

  ____

  ____

- If $\texttt{pc}[\theta] = 8$, then

  ____

  ____

  ____

- If $\texttt{pc}[\theta] = 9$, then executing the statement just sets $\texttt{pc}[\theta] = 3$ which preserves I_1.

Figure 5: Proof of Invariant I (part 1, to be completed by you for question 4a)

Now, we'll show that $I_1 \wedge I_2$ is an invariant. Because we just showed that I_1 is an invariant, we can assume that it holds before *and after* each statement. We just need to show that if $I_1 \wedge I_2$ hold before executing a statement, then I_2 will hold after executing the statement.

I_2 holds initially because

If $\texttt{pc}[\theta] \in \{1, 2, 3, 7, 9\}$, then executing the statement doesn't change `flag` or `turn`, and it doesn't set either `pc` value to 7. By the assumption that I_2 held before executing the statement, it will continue to hold afterwards.

If $\texttt{pc}[\theta] = 4$, then the clause $Q(\theta)$ holds after executing the statement because $\texttt{pc}[\theta] = 5$. The clause $Q(!\theta)$ also holds because $\texttt{pc}[\theta] = 5$. Thus, I_2 holds after executing the statement.

If $\texttt{pc}[\theta] = 5$, then,

If $\texttt{pc}[\theta] = 6$, then upon exiting the while loop, $\texttt{pc}[\theta] = 7$, and either $\texttt{flag}[!\theta] = \mathsf{false}$ or $\texttt{turn} = \theta$. Thus, $Q(\theta)$ holds. Furthermore, we know that $\texttt{pc}[!\theta] \neq 7$ because

If $\texttt{pc}[\theta] = w$, then after executing the statement, $\texttt{pc}[\theta] = 9$ which establishes $Q(\theta)$ and $\texttt{flag}[\theta] = \mathsf{false}$ which establishes $Q(!\theta)$.

Figure 6: Proof of Invariant I (part-2, to be completed by you for question 4a)