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? Mark Greenstreet
   (b) **(1 point)** What is your student number? 00000000

1. **(16 points)** Give a 2-4 sentence answer to each question below:
   (a) **(4 points)** What is the zero-one principle?

   *The **zero-one principle** states that a sorting network will sort any array correctly iff it sorts all arrays consisting only of zeros and ones correctly. A sorting network is a sorting algorithm that only performs compare-and-swap operations.*

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

   In the CTA model, λ is the measure of the cost for global communication. In particular, it is the time to access a remote data value divided by the time to access a local data value. For typical parallel computers, λ can be from 100 to one million or larger.

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

   *A **barrier** is a synchronization mechanism where all participating threads must arrive at the barrier, and then they all can proceed. Early arriving threads wait at the barrier until the last one arrives. Barriers are provided in Peril-L, pthreads and by the Java threads API.*

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

   *With **task parallelism**, a computation is divided into separate tasks. For example, a video game could use task parallelism by creating parallel tasks for sound generation, high-level game strategy, physical simulation, and graphics rendering. With **data parallelism**, independent computations are performed for each data word or block of data words. For example, a matrix multiplication algorithm could perform parallel operations on different blocks of rows and columns. With data parallelism, the available parallelism grows with the problem size, but with task parallelism, the amount of available parallelism remains fixed, even for large problems.*

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?

   *The report uses the term **multi-core** to refer to typical chip multiprocessors that have 2, 4, or 8 super-scalar cores on a die. They consider that the number of such cores could continue to double with each fabrication generation, but they expect the number of such cores to stay lower than 100 for the medium-term future. They coined the term **many-core** to refer to a chip that has a much larger number of simpler processor cores, probably not cache coherent. Here, they expect chips to be built with several hundred to a thousand or more cores.*
   The authors for Landscape advocate the many-core approach. They argue that simple cores offer the greatest total performance per watt, per joule, per square millimeter, and for the same design effort. They also argue that many-core designs are more tolerant of memory latency and of hard faults and soft errors. They note that programming a many-core chip is an unsolved

problem, but they suggest that by having a large number of cores, programmers may be able to write code that is independent of the exact number of processors available, and that this could potentially simplify the programming problem.

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

*The **13 dwarfs** are patterns or templates of computation. In particular, the dwarfs classify a computation according to its communication pattern. For example, some computations (such as many dense-linear algebra problems) only require nearest-neighbour communication, while others (e.g. FFT) have an all-to-all pattern, and some (e.g. fluid dynamics problems) have a banded structure where each processor communicates with a few others according to a regular pattern.*
Landscape *proposes that the dwarfs can be a useful way to understand parallel programs and parallel architectures. In particular, they propose that the strengths and weaknesses of an architecture can be identified by the dwarf patterns that it supports efficiently and those that it does not. They note that there is at least one pattern, the finite-state machine pattern that does not seem to be benefit from parallel computation.*

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

Landscape *says that results from **cognitive psychology** should be applied to the design of programming languages and parallel programming models in a way similar to their successful application to HCI. In particular, studies should be done of the kinds of errors that humans make when writing parallel programs. Likewise, the impact of various language design choices on programmer productivity should be investigated.*
They give an example of this with transactional memory and synchronization. They note that human programmers are prone to make errors when writing code for synchronization. Models like transactional memory relieve the programmer of some of this burden by providing atomic behaviour (potentially with some performance degradation) when a programmer does not provide sufficient synchronization.

(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.

This is connected to question 2a about multi-core vs. many-core architectures. The authors of Landscape *are emphatic in recommending designs with a large number of simple processors, quite possibly to the point of abandoning super-scalar architectures altogether and returning to simple, RISC style pipelines (See section 4.1.1). Four reasons that they give for this position are:*

1 *They argue that simple cores offer the best performance per joule or watt.*

2 *They argue that simple cores are simpler to design.*

3 *They argue that by using a large number of simple cores, the design is more tolerant to hardware failures. The processor can simply use the subset of cores that are working.*

4 *Small cores allow more fine-grain for choices involving voltage, frequency, and power consumption.*

Any three of these are fine for an answer, and there are other arguments lurking in Landscape *as well.*

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

Many parallel programs include a large number of parameters such as the number of threads or processes to run, parameters specifying how data should be partitioned amongst these threads or processes, the size of data transfers to use, etc. Often, when the code is ported to a new machine,

a human runs the program numerous times for different data inputs to determine good values for these parameters.
*An **auto-tuner** is a program that performs these optimizations automatically. The authors of Landscape noted that because autotuners have in some cases outperformed manually optimized software. This is because the autotuner can try many more configurations than are practical for a human to evaluate. Furthermore, the auto-tuner may try configurations that a human would exclude because they don't match the human's preconceptions of the underlying trade-offs. The authors of Landscape propose that autotuners may provide a functionality for large parallel programs similar to what optimizing compilers do for programs running on a single machine.*

3. **(24 points)**

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

   *An **empty/full** variable provides both communication and synchronization in a parallel program. If an empty/full variable is "empty" its value can be set, and this will also make the variable full. Reading the value of a full variable returns the variable to the empty state. This means that each value written to an empty/full variable can only be read once. Furthermore, after an empty/full variable has been written, it must be read before it can be written again. If a thread attempts to write to an empty/full variable that is already full, the writing thread will block until the variable becomes empty. Likewise, if a thread attempts to read from an empty variable, it will block until the variable becomes full.*

   (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.

   Figure 7 shows my solution. It's the first one I thought of; I compiled it; and it works.
   After writing it, I realized that a more efficient implementation is possible for the case that there are multiple writers and/or multiple readers. This better implementation would have a one `pthread_cond_t` variable for waiting writers and a separate one for waiting writers. There would also be a count of how many writers are currently waiting and a separate count for how many readers are currently waiting. If in a call to `ef_set`, there is a reader waiting, then the reader's condition variable is signaled. Likewise for calls to `ef_get`. But, I didn't write that one.
   Full credit will be given for any reasonable solution.

   (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.

   When the three threads attempt to read the variable, one of them will succeed and the other two will block. When the two threads attempt to read, one will get the first value written, and the other will block. The choice of which thread reads this value and which blocks is arbitrary. After the first read of the variable, one of the two blocked writers will set variable and continue. Then, the blocked reader will get this value, and then the final writer will be allowed to set the variable. At the end, the variable is full with the value set by the last writer. The choice of which writer threads perform which writes is arbitrary, and likewise for the reader threads.

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:

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

(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.

I've updated Figures 5 and —reffig:proof-I.2 with my solution.

(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 $!\theta$ 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
  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 $\texttt{pc}[\theta] = 4$, 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
  neither $\texttt{pc}[!\theta]$ nor $\texttt{flag}[!\theta]$ are modified by this action.
- If $\texttt{pc}[\theta] \in \{5, 6, 7\}$, then
  the value of the `flag` variables are unchanged. Because I_1 held before executing the statement, it will continue to hold after executing the statement.
- If $\texttt{pc}[\theta] = 8$, then
  executing the statement sets $\texttt{pc}[\theta]$ to 9 and $\texttt{flag}[\theta]$ to false which establishes the clause.
- 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
  for both values of θ, $\texttt{pc}[\theta] = 1$.
- 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,
  executing the statement sets $\texttt{pc}[\theta]$ *to 6 which establishes* $Q(\theta)$ *and* `turn` *to* $!\theta$ *which estabilishes* $Q(!\theta)$.
- 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
  $I \wedge \neg(\texttt{flag}[(]!\theta)\&\&(\texttt{turn} \neq \theta)) \wedge (\texttt{pc}[\theta] = 6)$ *held before executing the statement. We show by contradiction that* $\texttt{pc}[!\theta] \neq 7$ *before executing this statement. From* I_1, $(\texttt{pc}[!\theta] = 7) \Rightarrow \texttt{flag}[!\theta]$. *As noted above, when the while loop at line 6 exits,* $\neg(\texttt{flag}[(]!\theta)\&\&(\texttt{turn} \neq \theta))$. *Thus, if* $\texttt{flag}[!\theta]$ *holds, we must have* $\texttt{turn} = \theta$. *Because* I_2 *holds before executing the statement, we know that* $Q(!\theta)$ *holds, which means (if* $\texttt{pc}[!\theta] = 7$*) that* $\neg\texttt{flag}[\theta] \vee (\texttt{turn} = !\theta) \vee (\texttt{pc}[\theta] = 5)$ *But, we have refuted each of these three disjuncts. Therefore, we conclude that* $\texttt{pc}[!\theta] \neq 7$ *before executing this statement.*
  Executing this statement does not change $\texttt{pc}[!\theta]$. *Therefore,* $\texttt{pc}[!\theta] \neq 7$ *after executing the statement. This shows that* $Q(!\theta)$ *holds after executing the statement. It was shown above that* $Q(\theta)$ *holds after executing the statement. Therefore,* I_2 *holds as required.*
- 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)

```
struct EmptyFull {
     void *value;
     int full;
     pthread_mutex_t *lock;
     pthread_cond_t *cond;
};

EmptyFull ef_create(void) {
     EmptyFull ef = (EmptyFull)malloc(sizeof(struct EmptyFull));
     ef->value = NULL;
     ef->full = FALSE;
     ef->lock = (pthread_mutex_t *)malloc(sizeof(pthread_mutex_t));
     pthread_mutex_init(ef->lock, NULL);
     ef->cond = (pthread_cond_t *)malloc(sizeof(pthread_cond_t));
     pthread_cond_init(ef->cond, NULL);
     return(ef);
}

void ef_set(EmptyFull ef, void *value) {
     pthread_mutex_lock(ef->lock);
     while(ef->full)
        pthread_cond_wait(ef->cond, ef->lock);
     ef->value = value;
     ef->full = TRUE;
     pthread_cond_broadcast(ef->cond);
     pthread_mutex_unlock(ef->lock);
}

void *ef_get(EmptyFull ef) {
     pthread_mutex_lock(ef->lock);
     while(!ef->full)
        pthread_cond_wait(ef->cond, ef->lock);
     void *value = ef->value;
     ef->full = FALSE;
     pthread_cond_broadcast(ef->cond);
     pthread_mutex_unlock(ef->lock);
}
```

Figure 7: An implementation of an Empty/Full variable