CSL862 Minor1 Exam
Advanced Topics in Software Systems
Sem I, 2013-14
August 29, 2013

Answer all 5 questions — Max. Marks: 44

1. Answer **True/False**. Give reasons. No marks if no reason (or incorrect reason) given.
   a. Without using the happens-before graph, it is possible for the CHESS algorithm to never finish on a program (assume that the program is terminating and CHESS algorithm finishes when using the happens-before graph). Explain with examples if needed. [4]

      False. Happens-before graph is only used to prune the search space, as an optimization. CHESS algorithm is guaranteed to finish (albeit sometimes with a very large running time) if the program itself is guaranteed to terminate.

      No marks if explain what a happens before graph is, but give no reason why this can not happen. 2 marks if vague explanation, but in right direction. Also, no marks for "True" answer.

   b. Consider a program using Dthreads: Adding an extra non-synchronization instruction to the middle of this program can cause the interleaving behaviour of the program to change. Explain with example, if needed. [4]
      Accept both answers below:
      False. If the extra non-synchronization instruction does not alter the control flow of the program, the interleaving behaviour of the program will not change.

      True. If the extra non-synchronization instruction alters the control flow of the program, the interleaving behaviour can change.
      For example,

```
Thread 1
a++
lock()
unlock()

Thread 2
if (a == 1) {
  lock()
  unlock();
}
```

      (In case of the second answer, the example should be relevant. 2 marks if example is not relevant)

2. Consider Derandomized-PCT where a deterministic and systematic exploration of the schedules is performed in increasing order of bug depth. i.e., first all schedules to uncover depth-1 bugs are explored exhaustively, then all schedules to uncover depth-2 bugs are

explored exhaustively, then all schedules to uncover depth-3 bugs are explored exhaustively, and so on . . .

a. Assume a program with n threads and a total of k instructions. After how many schedules (in order notation) are we guaranteed to find a bug of depth d? [4]
   $O(nk^{d-1})$

b. Give an example of a program with a bug, such that the bug is of depth 2 and requires at least 2 pre-emptive context switches to be exposed (in CHESS). The program need not do anything useful. You can use the ASSERT statement in your program, where an assertion failure indicates a bug. [8]

```
Thread 1
Set(e);
a = NULL;                          … (2)


Thread 2
Wait(e);
if (a != NULL) {                   … (1)
   ASSERT(a);   //proxy for *a     … (3)
}
```

   This is a depth 2 bug with dependencies (1) → (2) → (3)

   But it also requires two preemptive context switches to manifest. The first before statement (2) and the second before statement (3).

   Notice that if there were no Wait/Set statements, this bug could have been uncovered with 1 preemptive context switch.

   6 marks if use “exit()” syscall to construct an example.

c. Give an example of a program with a bug, which will be found sooner with CHESS than with Derandomized-PCT? The program need not do anything useful. You can use the ASSERT statement in your program, where an assertion failure indicates a bug. [8]
   Consider a program with (n+1) threads, and assume that initially a = 0.

```
Thread 1:
if (a == 0) {
  a = 1;
}

Thread 2:
if (a == 1) {
  a = 2;
}

…

Thread n:
if (a == n - 1) {
  a = n;
}

Thread  (n+1):
ASSERT(a != n);
```

This is a bug with depth (n+1) exposed only with the following ordering constraints:
Thread 1 → Thread 2
Thread 2 → Thread 3
...
Thread n → Thread (n+1)

This bug will be found (or is guaranteed to be found) by Derandomized-PCT only when exploring schedules uncovering depth-n bugs. On CHESS however, the bug will definitely be found at c = 0 (after n! schedules).

3. PRES works by recording certain sources of non-determinism (e.g., result of synchronization operations) and ignoring others (e.g., data races). During replay, it explores the search space of the unrecorded non-determinism to try and reproduce the bug that caused a failure during production run. Answer the following questions

a. During replay, the "feedback generator" generates feedback for future replays. What kind of feedback is generated, and how is it used? [4]
PRES generates feedback on what non-deterministic choice should be tried next (for

data races only, as other non-deterministic choices have already been recorded).

b. While searching for the bug during replay, does it make sense to use context-bounding (from CHESS)? If so, how and why? If not, why not? [6]
   Yes, it makes sense to use context bounding. The problem at hand is to search for a schedule (among the possible interleavings of the unrecorded non-determinism, i.e., data races) that matches the recorded log and reproduces the bug. Because bugs are still likely to be of types that can be exposed at low ‘c’ values (assuming non-adversarial bugs), using context bounding (i.e., searching the schedule space in increasing order of ‘c’) should allow us to find a matching schedule faster.

   2 marks if wrong answer but show understanding of the PRES approach and of the question.

4. Consider the following program:

```
int x = 1, y = 2;
thread_fork(child_thread);
```

parent thread	child thread
x = y;	y = x;

```
thread_join(child_thread);
printf("x=%d, y=%d\n", x, y);
```

Assume that the statements “x = y” (or “y = x”) **atomically** move the contents from memory location ‘y’ to memory location ‘x’ (or from memory location ‘x’ to memory location ‘y’ respectively).

What are the possible final contents of x and y with

a. pthreads?
b. dthreads?

Explain. [4]

a. (x = 1, y = 1) if child thread executes before parent thread
or (x = 2, y = 2) if parent thread executes before child thread.

b. x = 2, y = 1. Because each thread starts with a private copy of the memory, the net effect would be of the values getting swapped.

5. Translation validation: What is a simulation relation? How is it verified? [2]

A simulation relation is the set of elements (PCs, PCt, E) defined on two programs ‘S’ and ‘T’ such that:

if S is at instruction PCs and T is at instruction PCt, E is true, then the programs are equivalent.

E is a boolean expression involving variables live at PCs and PCt.

A simulation relation can be verified by using symbolic execution and Satisfiability solvers.

Deduct 1.5 marks if the definition of a simulation relation is incorrect.
Deduct 0.5 marks if the verification method is incorrectly explained.