CSL373 Minor 1

Answer all 7 questions. 04/02/2013 Max. Marks: 15

OS Abstractions

1. Consider the C function 'printf()' on UNIX. Is printf() implemented by the OS, or by an application-level library? What system call does printf() make internally? [0.5]

printf() is part of the C library (glibc on Linux). This library is not part of the kernel. Printf() calls write() internally. 0.25 marks for each answer.

2. Consider the following code:

```
for (i = 0; i < 4; i++) {  fork();  }
```

If we start with one process, what is total number of processes spawned by this loop (excluding the first process). Explain. [2]

Let T(n) be the number of processes spawned by a loop of size n.

T(0) = 0.

T(n) = n + T(n-1) + T(n-2) + .. T(1)

T(1) = 1

T(2) = 2 + T(1) = 3

T(3) = 3 + T(2) + T(1) = 7

T(4) = 4 + T(3) + T(2) + T(1) = 4 + 7 + 3 + 1 = 15

3. How does the shell implement “&”, backgrounding? e.g., $ “./compute &” [1.5]

By not calling wait immediately. Instead it installs a SIGCHLD handler, which gets invoked on the termination of the child program. On termination, the handler may print an appropriate message on console.

Only 0.5 marks if do not mention SIGCHLD handler, as that would create zombie processes

4. Because threads can access shared state concurrently, a bad thread interleaving could potentially result in incorrect program behavior (if the program is not written carefully). Such a situation is called a race condition. Assume that you are developing a new OS, let's call this YOS (your-own OS). Assume that YOS runs only on uniprocessor machines.
    a. Is it possible for a multi-threaded application to have a race-condition when running on YOS (on a uniprocessor system)? Why/why not? Clearly state the assumptions you are making regarding your OS design to justify your answer. [1]
    Yes, assuming that YOS supports pre-emptible threads. A preemption can occur at any instruction boundary resulting in a race condition. This preemption will be caused by a timer interrupt.

    0.25 marks if only say context switch, but do not mention "preemptibility" or "timer interrupt"

    0.5 marks if write OS scheduling, but do not indicate how it happens

b. One way of disallowing race conditions is to use “locks”. Your lab partner suggests that a simple way of implementing locks in YOS is to implement two system calls called “disable_interrupts()” and “enable_interrupts()”. He suggests that because YOS runs only on uniprocessor systems, an application programmer can use these system calls to ensure atomicity of its critical sections. If you agree with him, implement functions “lock(L)” and “unlock(L)” using the system calls “disable_interrupts()” and “enable_interrupts()”. [1.5]

Some possible solutions:

A:

```
lock(L) { disable_interrupts(); }

unlock(L) {enable_interrupts(); }
```

B:

```
lock(int L)  {
  disable_interrupts();
  while (L == 1) { enable_interrupts(); disable_interrupts(); };
  L = 1;
  enable_interrupts();
}
unlock(int L)  {
  disable_interrupts();
  L = 0;
   enable_interrupts();
}
```

Deduct marks if do not implement lock() and unlock() but only show how to use enable_interrupts() and disable_interrupts(). We wanted to also test your understanding of the lock abstraction.

c. Is it a good/bad idea to provide such system calls (enable/disable interrupts) to the application developer? Why/why not? [1.5]

For solution A, this is a security flaw as an untrusted program could disable interrupts forever. No marks for other answers like “deadlock”, etc. is wrong, as the disabling interrupts forever is a much graver issue.

For solution B, this is a performance problem as repeatedly disabling and enabling interrupts (while waiting for lock to be released) is expensive.

Function Calls

5. Consider the following function:

```
int foo(int a,int b) {
    int c;
    c = a * b;
    int d;
    d = a + b;
    return c + d;
}
```

Assume all optimizations are disabled and the assembly code for a C statement (including declarations) is generated in the same order as the original source code.

a. Write the assembly pseudo-code generated for the function `foo()' on 32-bit x86. We will not focus on the syntax of the assembly written by you, we are only looking to test your understanding of how a compiler generates code for a function. [3]

```
1.  push %ebp
2.  mov %esp, %ebp
3.  sub $4, %esp
4.  mov 8(%ebp), %eax
5.  imull 12(%ebp)          #  edx : eax ← 12(%ebp) * eax
6.  mov %eax, (%esp)
7.  sub $4, %esp
8.  mov 8(%ebp), %eax
9.  add 12(%ebp), %eax
10. mov %eax, (%esp)
11. add 4(%esp), %eax
12. mov %ebp, %esp
13. pop %ebp
14. ret
```

Deduct 0.5 marks if prologue (lines 1,2) not specified properly. Deduct 0.5 marks if epilogue (lines 12,13) not specified properly. Deduct 0.5-1.5 marks if stack not conceived properly. Deduct 0.5 marks if local variables allocated in registers (optimizations were disabled). Deduct 0.5 marks if offsets for accessing arguments or local variables incorrect. Deduct 0.5 marks if use %esp to access function arguments. Deduct 0.5 marks if caller/callee registers not used properly. Zero if no idea about stack, etc. and just write assembly to do multiplication and addition.

b. Assume that the compiler decides not to use a frame pointer register (ebp on x86). Is it still possible to generate correct code for function foo()? Which assembly instructions will you change in your previous answer, and to what? [2]

Remove lines 1,2,12,13.

Replace lines 4,5 with

1. mov 8(%esp), %eax
2. imull 12(%esp)

Replace lines 8,9 with

1. mov 12(%esp), %eax
2. add 16(%esp), %eax

0.5 marks if only say “manipulate esp” or “replace ebp with appropriate esp offsets”. Deduct marks if offsets to %esp incorrect. 0 marks if you say it is possible but no explanation.

Virtual Memory

6. List the primary advantages of segmentation over paging. [1]

1. Less hardware complexity (lower power consumption, less area on chip)
2. Faster VA -> PA translation

Deduct 0.5 marks if do not mention any one of these. Zero for ambiguous answers.

7. List the primary advantages of paging over segmentation.[1]

1. No (or less) fragmentation
2. Allows demand paging
3. Allows sharing (copy-on-write) optimizations

Deduct 0.5 marks if not specify any one of these. Zero for ambiguous answers.