Princeton University
COS 217: Introduction to Programming Systems
Fall 2009 Final Exam Answers

The exam was a 3 hours, open-book, open-notes exam.

Question 1 Part (a)

What is being passed is a pointer to the function f, which is cast as a function pointer that takes two parameters: an int * and a const void *, and returns a float.

It is important to note that the expression contains a cast.

Question 1 Part (b)

Void pointers (void *). The major problem is that using void pointers subverts the compile-time type-checking system. In other words, type-safety can be violated.

Another problem is that a primitive type (int, char, etc.) cannot be matched with a void pointer, and so a C generic data structure implemented via void pointers cannot contain data of primitive types. But that problem is less important than the aforementioned one. (That answer received partial credit.)

Question 1 Part (c)

The compiler determines which variables should be placed in registers and when, and which should not. Since using registers is much faster than using memory, the compiler attempts to place in registers those variables that will be used frequently in the near future.

Question 1 Part (d)

The first instruction copies 32 to the EBX register. The second instruction attempts to fetch four bytes from memory at addresses 45, 46, 47, and 48, and then copy those four bytes to the EAX register. That attempt fails on our system, generating a segmentation fault, because application programs are prohibited from accessing memory at such low addresses. The programmer probably meant to write "movl $45, %eax". The third instruction executes a trap, thus invoking an operating system service.

In summary… As given, the instruction sequence executes a trap, determining the kind of trap from the four bytes in memory at addresses 45, 46, 47, and 48. If the second instruction were changed as noted above, then the instruction sequence would execute the brk() system call (system call 45) with the argument 32.

Question 2 Part (a)

EA**BC**DF
EA**CB**DF

Explanation: While the parent process is printing C, concurrently the child process is printing B. So those two letters might appear in either order.

Question 2 Part (b)

EA**BCDF**
EA**CBDF**
EA**CDBF**
EA**CDFB**

Explanation: While the parent process is printing the CDF sequence, concurrently the child process is printing B. So the B might appear before, within, or after the CDF sequence.

Question 2 Part (c)

EA**BFC**DF
EA**BCF**DF
EA**CBF**DF

Explanation: While the parent process is printing C, concurrently the child process is printing the BF sequence. So the C might appear before, within, or after the BF sequence.

Question 2 Part (d)

Any sequence that is possible and shows EA or A printed twice is acceptable.

Explanation: Because EA (or A) might be in the stdout buffer at the time of the fork(), EA (or A) might be written twice: once by the parent process and again by the child process.

Question 3

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "dynarray.h"

enum {MAX_LINE_LENGTH = 1024};

static int compareStringLengths(const void *item1, const void *item2) {
   return strlen((char*)item2) - strlen((char*)item1);
}

static void printAndFreeString(void *item, void *extra) {
   fputs((char*)item, stdout);
   free(item);
}

int main(void) {
   DynArray_T array;
   char line[MAX_LINE_LENGTH];
   char *linedup;

   array = DynArray_new(0);

   while (fgets(line, MAX_LINE_LENGTH, stdin) != NULL) {
      linedup = (char*)malloc(strlen(line) + 1);
      strcpy(linedup, line);
      DynArray_add(array, linedup);
   }

   DynArray_sort(array, compareStringLengths);
   DynArray_map(array, printAndFreeString, NULL);
   DynArray_free(array);

   return 0;
}
```

Notes: This question is a follow-up to Question 2 of the midterm exam. The most important aspect of the answer is the code that makes a copy of each string in a distinct area of memory. Without that code, all elements of the DynArray object would point to the same area of memory.

Question 4

```
ans1)
enter cal, a = 4
enter cal, a = 3
enter cal, a = 2
enter cal, a = 1
exit cal, result = 1
exec add
exit cal, result = 3
exec add
exit cal, result = 6
exec mul
exit cal, result = 24
ans2) 24
```

Question 5 Part (a)

```
int fd = dup(1);
close(1);
dup(2);
close(2);
dup(fd);
close(fd);
```

Notes:

Students were awarded complete credit even if they assumed that file descriptors 3 onwards are free/closed. For example, this answer was acceptable, although not as good:

```
dup(1);
dup(2);
close(1);
dup(4);
close(4);
close(2);
dup(3);
close(3);
```

If the program was correct, but there were side effects (such as an unnecessary file was created), the penalty was 2 points. E.g. int fd = creat ("temp", permission); close(fd); dup(1); close(1); dup(2); close(2); dup(fd); close(fd); received 8 points.

The penalty was 2 points for not knowing that stderr corresponds to 2, stdout corresponds to 1.

Sometimes there was a mistake in logic, and both streams ended up pointing to one of stdout/stderr. In that case the penalty was 5 points. E.g. close(1); dup(2); close(2); dup(1); received 5 points.

Instead of int fd = dup(1), some students wrote int fd = open(stdout,..) or int fd = creat(stdout, ..), but the rest was as in the best solution. In that case the penalty was 6 points.

Question 5 Part (b)

(i) No.

(ii) No.

(iii) execvp() overlays the original process. Thus, all sections of memory are changed (text, rodata, data, bss, stack, and heap).

(iv) execvp() does not change the file descriptor array. This design is for many reasons: the process can redirect file descriptors for the execvp() process; doing so allows one to treat execvp() as a black box; doing so eliminates the overhead that would be needed to appropriately set the file descriptors.

Question 6

```
############################################
    # int baz(int op1, int op2, int oper)

    .globl baz
    .type baz,@function

    # Formal parameter offsets:
    .equ OPER, 4

    # Local variable offsets:
    .equ RESULT, -4

    # Callee-saved register offsets:
    .equ EP1, -8
    .equ EP2, -12

baz:

    # prologue
    pushl %ebp
    movl %esp, %ebp

    # reserve space on stack for RESULT, EP1, EP2
    subl $12, %esp

    # save callee-saved registers (bar is called by main)
    movl %ep1, EP1(%ebp)
    movl %ep2, EP2(%ebp)

    # if(oper != 1) goto else
    cmpl $1, OPER(%ebp)
    jne else

    # result = op1 + op2
    movl %ep1, RESULT(%ebp)
    addl %ep2, RESULT(%ebp)
    jmp endif

else:

    # result = op1 - op2
    movl %ep1, RESULT(%ebp)
    subl %ep2, RESULT(%ebp)

endif:

    # restore callee-saved registers (bar is called by main)
    movl EP1(%ebp), %ep1
    movl EP2(%ebp), %ep2

    # return result
    movl RESULT(%ebp), %erv
    movl %ebp, %esp
    popl %esp
    ret

########################################
    # int bar(int a, int b, int c, int d)

    .globl bar
    .type bar,@function

    # Formal parameter offsets:
    .equ C, 4
    .equ D, 8

    # Local variable offsets:
    .equ I, -4
    .equ J, -8

    # Callee-saved register offsets:
    .equ EP1, -12
    .equ EP2, -16

    # Caller-saved register offsets:
    .equ ERA, -20

bar:

    # prologue
    pushl %ebp
    movl %esp, %ebp

    # reserve space on stack for I, J, EP1, EP2, ERA
    subl $20, %esp

    # save callee-saved registers (bar is called by main)
    movl %ep1, EP1(%ebp)
    movl %ep2, EP2(%ebp)

    # save caller-saved registers (bar calls baz)
    movl %era, ERA(%ebp)

    # i = baz(a, b, 1)
    # a in %ep1 and b in %ep2 already
    pushl $1
    call baz
    movl %erv, I(%ebp)
    addl $4, %esp

    # j = baz(c, d, 1)
    pushl $1
    movl D(%ebp), %ep2
    movl C(%ebp), %ep1
    call baz
    movl %erv, J(%ebp)
    addl $4, %esp

    # i = baz(i, j, 0)
    pushl $0
    movl J(%ebp), %ep2
    movl I(%ebp), %ep1
    call baz
    movl %erv, I(%ebp)
    addl $4, %esp

    # restore callee-saved registers (bar is called by main)
    movl EP1(%ebp), %ep1
    movl EP2(%ebp), %ep2

    # restore caller-saved registers (bar calls baz)
    movl ERA(%ebp), %era

    # return i
    movl I(%ebp), $erv
    movl %ebp, %esp
    popl %esp
    ret

#########################################
    # int main(void)

    .globl main
    .type main,@function
    # Local variable offsets:
    .equ F, -4

    # Callee-saved register offsets:
    .equ EP1, -8
    .equ EP2, -12

    # Caller-saved register offsets:
    .equ ERA, -16

main:

    # prologue
    pushl   %ebp
    movl    %esp, %ebp

    # space for F, EP1, EP2, ERA
    subl $16, %esp

    # save callee-saved registers (main is called by start)
    movl %ep1, EP1(%ebp)
    movl %ep2, EP2(%ebp)

    # save caller-saved registers (main calls bar)
    movl %era, ERA(%ebp)

    # f = bar(1, 2, 3, 4)
    pushl $4
    pushl $3
    movl $2, %ep2
    movl $1, %ep1
    call bar
    movl $erv, %F(%ebp)
    addl $8, %esp

    # restore callee-saved registers (main is called by start)
    movl EP1(%ebp), %ep1
    movl EP2(%ebp), %ep2

    # restore caller-saved registers (main calls bar)
    movl ERA(%ebp), %era

    # return f
    movl F(%ebp), $erv
    movl %ebp, %esp
    popl %esp
    ret
```

###

Grading Notes:

(A) General distribution of the points (one point for each item):
[1] prologue, epilogue
[2] reserve space on stack
[3] save callee-saved registers
[4] save caller-saved registers
[5] pass parameters
[6] function call
[7] get return value
[8] reset ESP (after function call)
[9] restore callee-saved registers
[10] restore caller-saved registers

[11] set return value
[12] return

(B) The code from this answer sheet is a baseline implementation. Obviously, you can optimize the code. For example, you can use %erv (instead of another local variable) in main() and baz(). But of course we deducted some points if your code produced the wrong result.

(C) We were relatively loose on the syntax and format of the code, but we still deduct points for serious flaws (the order of operands, for example).

Question 7

(i) Yes. A DFA is composed of a fixed number of states that transition from one to another. To have nested comments, there need to be many more states corresponding to each level of nesting. If there is a fixed maximum nesting level of comments then it is entirely possible to be detected by a DFA. It is interesting to note that if there is not a fixed maximum nesting level then it would not be possible to accomplish this by a DFA.

(ii) No. Because in order to build a DFA, we need to know how many states are there in all. However, in this question, since the nesting level is arbitrary, it is impossible to know how many states we need.