COS 217 Final Exam Fall 2009

- **This examination is open-book and open-notes. ONLY BOOKS AND NOTES. You are not permitted to use ANY electronic devices (laptops, PDAs, tablets, cell-phones, calculators …)**
- **You have 3 hours to complete the exam. The total number of points is 150.**
- **Please answer all questions.**
- ***Please always read code very carefully. There may be subtleties there.***
- **Please read through all questions before starting to answer any. It is advisable that you determine which questions you are most comfortable with or that will be easiest for you to answer correctly, and do those first.**
- **Please make sure you PRINT your name and preceptor's name clearly and legibly below (use block lettering), and write and sign the honor code below.**

Name:

Login ID:

Preceptor:

Honor Code Pledge:

Honor Code Signature: _____

For instructor use only:

Question	Points
I	
II	
III	
IV	
V	
VI	
VII	
Total	

Question I. Short Answer Questions [20 points]

(a) The following expression is passed to a function as a parameter. Describe in words what is being passed:

```
(float (*)(int *, const void*)) f
```

(b) What is the best mechanism that C provides for implementing generic data structures? State, in one sentence, one key problem with this mechanism?

(c) Typically, which one among the hardware, the preprocessor, the compiler, the assembler or the linker determines which variables should be placed in registers and when, and which should not? What is one of the main criteria used to determine such "register allocation"?

(d) What does the following assembly code do? Describe as completely as you can.

```
movl   $32, %ebx
movl    45, %eax
int     $128
```

Question II. 20 points

(a) What does this program print? Show every possible output. Draw a box around each possible output.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

static void f(void) {
  putchar('A');
  fflush(NULL);
  if (fork() == 0) {
     putchar('B');
     exit(0);
  }
  putchar('C');
  wait(NULL);
  putchar('D');
}

int main(void) {
  putchar('E');
  f();
  putchar('F');
  return 0;
}
```

(b) What does this program print? Note that it's the same as the program from part (a) except that the "wait(NULL);" statement has been removed. Assume all #includes from part (a) are included. Show every possible output. Draw a box around each possible output.

```
static void f(void) {
  putchar('A');
  fflush(NULL);
  if (fork() == 0) {
     putchar('B');
     exit(0);
  }
  putchar('C');
  putchar('D');
}

int main(void) {
  putchar('E');
  f();
  putchar('F');
  return 0;
}
```

(c) What does this program print? Note that it's the same as the program from part (a) (*Note*: not part (b) but part (a)), except that the "exit(0);" statement has been replaced with "return;". Assume all #includes from part (a) are included. Show every possible output. Draw a box around each possible output.

```
static void f(void) {
  putchar('A');
  fflush(NULL);
  if (fork() == 0) {
     putchar('B');
     return;
  }
  putchar('C');
  wait(NULL);
  putchar('D');
}

int main(void) {
  putchar('E');
  f();
  putchar('F');
  return 0;
}
```

(d) What does this program print? Note that it's the same as the program from part (a) (*Note*: not part (c) but part (a)) except that the "fflush(NULL);" statement has been removed. Assume all #includes from part (a) are included. This time, you need not show every possible output. Instead show **one** possible output that illustrates the effect of removing the "fflush(NULL);" statement. Draw a box around the output.

```
static void f(void) {
  putchar('A');
  if (fork() == 0) {
     putchar('B');
     exit(0);
  }
  putchar('C');
  wait(NULL);
  putchar('D');
}

int main(void) {
  putchar('E');
  f();
  putchar('F');
  return 0;
}
```

Question III. 25 points

Write a C program that reads lines from stdin, sorts them in decreasing order by length (that is, longest lines first), and writes the results to stdout. Your program:

-- May assume that each line of stdin contains no more than 1024 characters.

-- Should make no assumptions about how many lines are in stdin.

-- May assume that sufficient memory exists.

-- Should be well-styled, but need not include comments.

-- May use the DynArray ADT from precepts.

For your convenience, the interface of the DynArray ADT is shown below.

```
#ifndef DYNARRAY_INCLUDED
#define DYNARRAY_INCLUDED

typedef struct DynArray *DynArray_T;
/* A DynArray_T is an array whose length can expand dynamically. */

DynArray_T DynArray_new(int iLength);
/* Return a new DynArray_T object whose length is iLength, or NULL if insufficient memory is available.
*/

void DynArray_free(DynArray_T oDynArray);
/* Free oDynArray. */

int DynArray_getLength(DynArray_T oDynArray);
/* Return the length of oDynArray. */

void *DynArray_get(DynArray_T oDynArray, int iIndex);
/* Return the iIndex'th element of oDynArray. */

void *DynArray_set(DynArray_T oDynArray, int iIndex, const void
*pvElement);
/* Assign pvElement to the iIndex'th element of oDynArray.  Return the old element. */

int DynArray_add(DynArray_T oDynArray, const void *pvElement);
/* Add pvElement to the end of oDynArray, thus incrementing its length. Return 1 (TRUE) if successful, or
0 (FALSE) if insufficient memory is available. */

int DynArray_addAt(DynArray_T oDynArray, int iIndex, const void
*pvElement);
/* Add pvElement to oDynArray such that it is the iIndex'th element. Return 1 (TRUE) if successful, or 0
(FALSE) if insufficient memory is available. */

void *DynArray_removeAt(DynArray_T oDynArray, int iIndex);
/* Remove and return the iIndex'th element of oDynArray. */

void DynArray_toArray(DynArray_T oDynArray, void **ppvArray);
/* Fill ppvArray with the elements of oDynArray.  ppvArray should point to an area of memory that is
large enough to hold all elements of oDynArray. */

void DynArray_map(DynArray_T oDynArray, void (*pfApply)(void
*pvElement, void *pvExtra),
  const void *pvExtra);
/* Apply function *pfApply to each element of oDynArray, passing pvExtra as an extra argument.  That is,
for each element pvElement of oDynArray, call (*pfApply)(pvElement, pvExtra). */
```

```
void DynArray_sort(DynArray_T oDynArray,
  int (*pfCompare)(const void *pvElement1, const void *pvElement2));
```
/* Sort oDynArray in the order determined by *pfCompare. *pfCompare should return <0, 0, or >0 depending upon whether *pvElement1 is less than, equal to, or greater than *pvElement2, respectively. */

```
int DynArray_search(DynArray_T oDynArray, void *pvSoughtElement,
  int (*pfCompare)(const void *pvElement1, const void *pvElement2));
```
/* Linear search oDynArray for *pvSoughtElement using *pfCompare to determine equality. Return the index at which *pvSoughtElement is found, or -1 if there is no such index. *pfCompare should return 0 if *pvElement1 is equal to pvElement2, and non-0 otherwise. */

```
int DynArray_bsearch(DynArray_T oDynArray, void *pvSoughtElement,
  int (*pfCompare)(const void *pvElement1, const void *pvElement2));
```
/* Binary search oDynArray for *pvSoughtElement using *pfCompare to determine equality. Return the index at which *pvSoughtElement is found, or -1 if there is no such index. *pfCompare should return <0, 0, or >0 depending upon whether *pvElement1 is less than, equal to, or greater than *pvElement2, respectively. oDynArray should be sorted as determined by *pfCompare. */

```
#endif
```

Question IV (20 points)

What will be printed by the following program?

```
#include <stdio.h>

int add(int a, int b) {
 printf("exec add\n");
 return a+b;
}

int mul(int a, int b) {
 printf("exec mul\n");
 return a*b;
}

int calculate(int a, int (*op)( int, int), int (*op2)( int, int)) {
 int result = 1;
 int (*func) (const int, const int);
 printf("enter cal, a = %d\n", a);
 if(a >  1) {
   result = calculate(a-1, op, op2);
   if(result & 1) {
     func = op;
   } else {
     func = op2;
   }
   result = (*func)(a, result);
 }
 printf("exit cal, result = %d\n", result);
 return result;
}

int main(int argc, char **argv) {
 int result;

 printf("ans1)\n");
 result = calculate(4, add, mul);
 printf("ans2) %d\n", result);
 return 0;
}
```

Question V (20 points)

(a) Write less than ten lines of C code to redirect `stdout` and `stderr` to each other. i.e. to redirect `stdout` to `stderr` and `stderr` to `stdout`. You do not need to handle errors explicitly.

(b) In order to execute a binary command in the shell assignment, we changed the file descriptors in the child process, and then `execvp`'ed the binary command.

(i) Does changing the file descriptors in the child process change the file descriptors in the parent process?

(ii) Does `execvp` create a new process?

(iii) What sections of the memory does `execvp` change?

(iv) Does `execvp` change the file descriptor array? Why might the designers of Unix have made this decision?

Question VI (35 points)

Sarah believes exploiting the storage hierarchy can help improve program performance. In particular, she likes the idea that the access to registers is faster than that to memory. To exploit registers better to speed up programs, she designs a new computer architecture, JA-32, based on IA-32. Sarah makes only the following changes to the IA-32 architecture:

[1] She adds two 32-bit registers for parameter passing, EP1 and EP2. Upon a function call, the caller will place the first two integral-types variables (32-bits long each) in EP1 and EP2, respectively, but not on the stack. EP1 and EP2 are both callee-saved registers.

[2] She adds one 32-bit register, ERV, for storing the return value of a function. Upon a function return, the callee will place the return value (if any) in the register ERV, but not on the stack. ERV is a caller-saved register.

[3] She adds one 32-bit register for storing the return address, ERA. The call/ret instruction formats remain the same, and their effects are the same, but the return address ("old EIP") is placed in ERA, not on the stack. ERA is a caller-saved register.

The stack memory is available for you to use, as long as you don't break the rules described above (e.g., the caller should not place the first integral parameter on the stack). The four additional registers can be used in any instructions, just like EAX, ESP, etc., but they can not allowed to use them to store any other data than as described above.

Construct the corresponding JA-32 assembly code for the following C code (note that the code is continued on the next page). Be careful about JA-32's stack frame structure, as it may be different from IA-32's.

```
int baz(int op1, int op2, int oper) {
  int result;
  if(oper == 1) {
    result = op1 + op2;
  }
  else {
    result = op1 - op2;
  }
  return result;
}
```

```
int bar(int a, int b, int c, int d) {
  int i;
  int j;
  i = baz(a, b, 1);
  j = baz(c, d, 1);
  i = baz(i, j, 0);
  return i;
}

int main(void) {
  int b;
  b = bar(1, 2, 3, 4);
  return b;
}
```

Question VII (10 points)

In an assignment, you were asked to use a DFA to detect comments of the form

```
/* COMMENT HERE */
```

These comments were not nested; that is, if there were two /* markers one inside the other, only one */ symbol was necessary to end the comment. This is an example:

```
/* THIS IS A COMMENT /* THIS IS STILL A COMMENT */ THIS IS NOT A COMMENT
```

The following question is not based on material we have directly covered in lecture. Rather, it is intended to test your thinking a little beyond lecture material though based on an understanding of DFAs that we've seen in lecture. We're looking for answers stated in a couple of sentences.

Suppose we change the specification of a comment such that /* */ comments are nested. Thus, if there are two /* markers one inside the other in the text, there need to be two */ markers to end the comment. For example:

```
/* THIS IS A COMMENT /* THIS IS STILL A COMMENT */ THIS IS STILL A COMMENT */ THIS IS NOT A COMMENT.
```

(i) Can these comments still be detected by a DFA if there is a fixed maximum nesting level of comments allowed? Why or why not? We do not need a formal answer in terms of languages and grammars, but please state the key intuitive reason as to why or why not.

(ii) How about with an arbitrarily large (but not infinite) nesting level? Why or why not? We do not need a formal answer in terms of languages and grammars, but please state the key intuitive reason as to why or why not.