Final Exam SECTION 002 Dec 19, 2017

COMS W3157 Advanced Programming Prof. Jae Woo Lee
Fall 2017 Columbia University

About this exam:

- There are 3 problems totaling 100 points:

  Problem 1: 48 points
  Problem 2: 32 points
  Problem 3: 20 points

- Assume the following programming environment:

  All programs are built and run on Ubuntu Linux 16.04, 64-bit version, where sizeof(int) is 4 and sizeof(int *) is 8.

  All library function calls and system calls are successful. For example, you can assume malloc() does not return NULL.

  For all program code in this exam, assume that all the necessary #include statements are there even if they are not shown.

  If this exam refers to lab code, assume the versions provided by Jae, i.e., skeleton code and solutions.

  When writing code, avoid using hardcoded numbers as much as possible. Hardcoded numbers make your program error prone, less extensible, and less portable. For example, using "sizeof(int *)" instead of "8" will make it correct for both 32-bit and 64-bit environments.

What to hand in and what to keep:

- At the end of the exam, you will hand in only the answer sheet, which is the last two pages (one sheet printed double-sided) of this exam booklet. You keep the rest of the exam booklet.

- Make sure you write your name & UNI on the answer sheet.

- Please write only your final answers on the answer sheet. Verbosity will only hurt your grade because, if we find multiple answers to a question, we will cherry-pick the part that will result in the LOWEST grade. This policy ensures that a shotgun approach to solving a problem is never rewarded. Please make sure you cross out clearly anything that you don't consider your final answer.

- Before you hand in your answer sheet, please copy down your answers back onto the exam booklet so that you can verify your grade when the solution is published in the mailing list.

```
+-------------------------------------------------------------------+
|          This exam is for SECTION 002, 4:10pm class.              |
|     You MUST be registered for this section to take this exam.    |
+-------------------------------------------------------------------+
|  PLEASE DO NOT OPEN THIS EXAM BOOKLET UNTIL YOU ARE TOLD TO DO SO!  |
+-------------------------------------------------------------------+
```

```
References: SmartPtr, classify_pointer(), print(), PRINT_TYPE()
------------------------------------------------------------
template <class T>
class SmartPtr {
    private:
        T *ptr;
        int *count;
    public:
        explicit SmartPtr(T *p = 0) { ptr = p; count = new int(1); }
        SmartPtr(const SmartPtr<T>& sp)
            : ptr(sp.ptr), count(sp.count) { ++*count; }
        ~SmartPtr() { if (--*count == 0) { delete count; delete ptr; } }
        SmartPtr<T>& operator=(const SmartPtr<T>& sp) {
            if (this != &sp) {
                if (--*count == 0) { delete count; delete ptr; }
                ptr = sp.ptr; count = sp.count; ++*count;
            }
            return *this;
        }
        T& operator*() const { return *ptr; }
        T* operator->() const { return ptr; }
        T* getPtr() const { return ptr; }
        operator void*() const { return ptr; }
};

template <typename T> struct SmartPtrStruct { T *ptr; int *count; };

// Returns the current reference count of the given SmartPtr
template <typename T>
int ref_count(const SmartPtr<T>& p) {
    // A hack to access count, which is a private member
    return *(((SmartPtrStruct<T> *)&p)->count);
}

/* Assume the classify_pointer function exists, is correct, doesn't
 * leak memory, and works as follows:
 *
 *   It prints "STACK" if p contains an address on the stack;
 *   it prints "HEAP" if p contains an address on the heap;
 *   it prints "NEITHER" if p contains an address neither on the stack
 *                                                nor on the heap.
 */
void classify_pointer(const void *p);

template <typename T> void print(T t) { cout << t << endl; }

/* Assume that PRINT_TYPE( <expression> ) will print the type name of
 * the given expression. For example,
 *
 *   int x;
 *   PRINT_TYPE( &x );
 *
 * will print the following:
 *
 *   int*
 *
 * PRINT_TYPE() is implemented as a macro and the code is not shown here.
 */
```

Problem [1]: 48 points total, 16 parts, 3 points each

The following program builds without error, runs without crashing, and successfully produces 16 lines of output for (1.1) ... (1.16). Fill in the blanks on the answer sheet to match the program's output.

```
int main() {
    SmartPtr<MyString>            sp;
    vector< SmartPtr<MyString> >   v;
    vector< SmartPtr<MyString> >   w;

    char a[4] = { 'A', 'B', 'C', 0 };
    char *p = a;
    assert('A' == 65);

    for (int i = 0; i < 3; i++) { // i goes from 0 to 2
        SmartPtr<MyString> sp(new MyString(p++));
        v.push_back(sp);
    }

    cout << "\n (1.1) "; print(*v[0] + *v[1] + *v[2]);

    cout << "\n (1.2) "; print((int)( a + 3 == NULL ));

    cout << "\n (1.3) "; print((int)( a == *v[0] ));

    cout << "\n (1.4) "; print((int)( *a == (*v[0])[0] ));

    cout << "\n (1.5) "; classify_pointer( a );

    cout << "\n (1.6) "; classify_pointer( &p );

    cout << "\n (1.7) "; classify_pointer( &p[0] );

    cout << "\n (1.8) "; classify_pointer( v[0] );

    cout << "\n (1.9) "; classify_pointer( &v[0] );

    cout << "\n(1.10) "; classify_pointer( &*v[0] );

    cout << "\n(1.11) "; PRINT_TYPE( &v[0] );

    cout << "\n(1.12) "; PRINT_TYPE( &*v[0] );

    cout << "\n(1.13) "; PRINT_TYPE( &p );

    sp = v[0];
    w.push_back(sp);
    w.push_back(sp);
    w.push_back(sp);

    cout << "\n(1.14) "; print( ref_count(sp) );

    *sp = *v[1] = *v[2];

    cout << "\n(1.15) "; print( ref_count(sp) );

    cout << "\n(1.16) "; print(*w[0] + *w[1] + *w[2]);                }
```

Problem [2]: 32 points total

Consider the following C++ program, fork-sleep.cpp:

```
        void PRINT(int i, const vector<pid_t>& v)
        {
            // Implementation for (2.1)-(2.3) only

            fprintf(stderr, "%d,", i);
        }

        int main()
        {
            pid_t         p;
            vector<pid_t> v;
            const int four_seconds = 4;

            for (int i = 0; i < 3; i++) { // i goes from 0 to 2
                v.push_back(p = fork());
                if (p == 0) { // child process
                    sleep(four_seconds); // pause execution for 4 seconds
                    PRINT(i, v);
                    return 0;
                }
                v.push_back(p = fork());
                if (p == 0) {
                    sleep(four_seconds); // pause execution for 4 seconds
                    PRINT(i, v);
                    return 0;
                }
                sleep(four_seconds); // pause execution for 4 seconds
            }
            for (int i = 0; i < v.size(); i++) {
                waitpid(v[i], NULL, 0); // no status & no options
            }
            fprintf(stderr, "\n");
        }
```

We ran the program on a lightly loaded system so that, if we had removed all calls to the sleep() function, the program would have finished running almost instantly (well under 0.05 sec). Answer the following questions on the answer sheet.

(2.1)

Is the output always the same? Write FIXED if the output is the same every time the program runs; write UNPREDICTABLE if the output can vary from one run to another.

(2.2)

Write the output of the program. If you wrote UNPREDICTABLE for (2.1), write any one of the possible outputs.

(2.3)

How many processes of the program are paused at the sleep() function after 2, 10, 18, 26, 34, 42 seconds from the start of the program execution?

Problem [2] continued

For (2.4) - (2.8), we changed the PRINT() function as follows:

```
        void PRINT(int i, const vector<pid_t>& v)
        {
            // Implementation for (2.4)-(2.8)

            fprintf(stderr, "%d,", (int)v.size());
        }
```

(2.4)

Is the output always the same? Write FIXED if the output is the same every time the program runs; write UNPREDICTABLE if the output can vary from one run to another.

(2.5)

How many numbers are printed? For example, if you think the output alternates between "31,31,57,57," and "57,57,31,31,", you would write 4.

(2.6)

How many DISTINCT numbers are printed? For example, if you think the output alternates between "31,31,57,57," and "57,57,31,31,", you would write 2.

(2.7)

How many possible outputs are there? If you wrote FIXED for (2.4), your answer here would be 1.

(2.8)

Write all possible outputs. Write each possible output on a separate line. Write your lines in ascending order. For example, if you think the possible outputs are "100,400,", "200,300,", "200,400,", write them like this:

```
        100,400,
        200,300,
        200,400,
```

Problem [3]: 20 points total, 5 parts, 4 points each

(3.1)

Which of the following Socket API functions are called by tcp-sender.c, which was one of the TCP sample programs we studied in class?

Select ALL that apply. If you think all of them apply, write "A,B,C,D,E,F,G". If you think none of them apply, write "NONE" (you must write NONE in this case; a blank answer will not be accepted).

(A) socket
(B) bind
(C) listen
(D) accept
(E) connect
(F) send
(G) recv

(3.2)

Which of the following Socket API functions return a socket descriptor?

Select ALL that apply, using the same answer format from (3.1).

(A) socket
(B) bind
(C) listen
(D) accept
(E) connect
(F) send
(G) recv

(3.3)

In HTTP/1.0, the connection is closed after a single request/response pair. In HTTP/1.1 a keep-alive-mechanism was introduced, where a connection could be reused for more than one request. (True / False)

(3.4)

C/C++ programs can use stdin, stdout, and stderr objects to access standard input, standard output, and standard error, respectively. Which of the following correctly describe the objects?

Select ALL that apply, using the same answer format from (3.1).

(A) They are of type "FILE&".
(B) They are of type "int".
(C) They are of the same type as socket descriptors.
(D) stdin can be passed to fdopen() to wrap it in a file descriptor.
(E) By default, stderr is block-buffered.

(3.5)

In C++, the vector template class guarantees that all its elements are stored in a contiguous block of memory. The deque template class makes the same guarantee, but the list template class does not. (True / False)

UNI: Name:

[2]

(2.1) _____

(2.2) _____

(2.3)

_____ processes are paused at sleep() after 2 secs

_____ processes are paused at sleep() after 10 secs

_____ processes are paused at sleep() after 18 secs

_____ processes are paused at sleep() after 26 secs

_____ processes are paused at sleep() after 34 secs

_____ processes are paused at sleep() after 42 secs

(2.4) _____

(2.5) _____

(2.6) _____

(2.7) _____

(2.8)

[3]

(3.1)

(3.2)

(3.3)
True/
False

(3.4)

(3.5)
True/
False

COMS 3157, Fall 2017
Final Exam, SECTION 002
Answer Sheet, page 2 of 2

Your UNI: +---+---+---+---+---+---+---+---+
| | | | | | | | |
+---+---+---+---+---+---+---+---+

Your Name: _____

front->UNI: _____

left->UNI: _____ [You] right->UNI: _____

back->UNI: _____

---

[1]

(1.1) _____

(1.2) _____

(1.3) _____

(1.4) _____

(1.5) _____

(1.6) _____

(1.7) _____

(1.8) _____

(1.9) _____

(1.10) _____

(1.11) _____

(1.12) _____

(1.13) _____

(1.14) _____

(1.15) _____

(1.16) _____