CS240 Midterm 2 Answers 07/27/2015

P1(a) 12 pts

In struct, memory required for all the member/field types is allocated.
3 pts

In union, memory required to store largest member/field type is allocated.
3 pts

Therefore, in union, assigning values to a member is destructive (i.e., changes) to the value of other members.
2 pts

struct ex1 {char a; int b;};
union ex2 {char a; int b;};

struct ex1 x;
struct ex2 y;

x.a = 'c';     
x.b = 10000;
// x can hold both members a and b hence printing x.a and x.b will
// show the correct values assigned

y.a = 'c';
y.b = 10000;
// the last assignments overwrites the y.a = 'c' assignment, hence
// printing y.b will be correct but printing y.a will not

P1(b) 12 pts

x is a pointer to a pointer of int, i.e., x holds an address at which
the stored value is another address at which a value of type int can
be stored.
4 pts

y[5][7] is organized in memory such that y is a pointer to memory
that contains 5 consecutive pointers *(y), *(y+1), ..., *(y+4). Each of
these pointers points to memory containing space to hold 7 values of
type int. In the case of *(y+1), they can be accessed by
*((*(y+1))), *((*(y+1)+1)), ..., *((*(y+1)+6)). Similarly for the others.
6 pts

Hence y is a pointer to a pointer of type int.
2 pts
P1(c) 12 pts

The number of arguments (following the first argument which is the format string) can be inferred/extracted by counting the number of '%' symbols in the format string.

4 pts

int printf(char *format, ...);

4 pts

When printf() counts the number of '%' symbols in the format string, it finds 2. Hence it will call va_arg() only twice to retrieve the remaining arguments, which ignores z. printf() will print x and y correctly but ignores z.

4 pts

P2(a) 12 pts

// allocate space for Alice
s = (contacts_t *) malloc(sizeof(contacts_t));
s->name = (char *) malloc(6));
s->telnum = (char *) malloc(13));
s->next = (contacts_t *) malloc(sizeof(contacts_t));
// In the above, malloc(6) for "Alice" can be 5 if \0 is to be ignored
// similarly for malloc(13) which can be 12.
6 pts

strcpy(s->nsme, "Alice");
strcpy(s->telnum, "765-000-0001");
2 pts

// allocate space for Bob
(s->next)->name = (char *) malloc(4));
(s->next)->telnum = (char *) malloc(13));
4 pts

strcpy(s->nsme, "Bob");
strcpy(s->telnum, "765-000-0002");
0 pts

P2(b) 12 pts

When a called function (callee) returns to the function that called it (caller), the process from which the caller called the callee remains.

2 pts

When the function exit() is called from a caller, it terminates the process that runs it (and the caller), hence exit() does not return to the caller as it does not exist anymore.
exit(1) conveys the value 1 to the parent process of the process that executed exit().

The process's image in memory is replaced by that specified in the argument of execl(). The process ID remains unchanged.

When execl() succeeds, it does not return to the caller since a new executable replaces the old one. When it fails (e.g., the specified binary does not exist) then execl() returns to the caller with an error.

A process that executes a system call interacts directly with the operating system.

P2(c) 12 pts

Store arguments passed from caller to callee and local variables of the callee.

z is allocated in the stack frame of myfunc() and takes up 4 bytes since it's a pointer.

Heap space which resides above the data area in a C program memory layout.

The space of z is deallocated as a result of popping the stack frame of myfunc().

Since the space allocated by malloc() resides in the heap area and not the stack frame of myfunc(), it remains allocated when myfunc() returns. However, there may be no way to know where in memory the allocated space is since it was stored in z.

P3(a) 14 pts

int timer;
main() {
    FILE *fp;

    // open file
    fp = fopen("stuff-to-run.txt","r");

    // register signal handler
    signal(SIGALRM, my_alrm_handler);

    // loop structure:
    // different ways to do, including calling sleep() instead of
    // alarm() followed by pause()
    while(fscanf(fp,"%d %s", &timer, app) != EOF) {
        alarm(60*timer);
        pause();
    }
}

void my_alrm_handler(int a) {
    pid_t x;

    if(a == SIGALRM) {
        x = fork();
        // child runs binary
        if (x == 0)
            execl(app, app, NULL);
    }
}

When grading, the important parts are the overall code structure and the components therein. Typos and syntactic mistakes are not as important as getting the gist of the logic using while (infinite) loop to process commands in the parent, registering a signal handler in the parent, forking a child in the signal handler that calls execl(), etc. Whether they use fscanf() or some other means to read is not that important as long as it's a valid approach.

P3(b) 14 pts

The memory pointed to by r has not been allocated.
2 pts

We can register a signal handler for SIGSEGV which the OS will execute when a segmentation fault arises. Inside the signal handler, we can print the value of r.
4 pts

float **z;
main() {
float *r;

    signal(SIGSEGV, mysegvhandler);
    z = &r;
    r = 5.5;
}

void mysegvhandler(int a) {
    if(a == SIGSEGV)
        printf("%p", *z);
}

8 pts

When grading, focus on, first, that they clearly understand what a signal does and how to use it, second, the key components. Since r is a local variable of main(), they should realize that to print its value (i.e., address), they will have to remember it separately so that it can be accessed from their signal handler.

Bonus

In C apps, main() is called (hidden) by a start-up function that the compiler inserts but from nowhere else. Hence, there is no burden to treat the variable number of arguments of main() as a general issue such as in the case of printf() which is called by user functions/code.

At compile time, the compiler know if the code reads main(), main(int argc, char **argv), or main(int argc, char **argv, char **environ) and therefore can generate object accordingly.

10 pts