CS240 Midterm 2 Answers 07/21/2016

P1(a) 12 pts

Both times the value 1 is printed.
3 pts

Since a is a local variable of myfunc() its memory is part of myfunc()'s stack area (i.e., frame)
which gets destroyed between the two calls to myfunc(). Hence, a is newly allocated and initialized
each time myfunc() is called.
3 pts

The values printed are 1 and 2.
3 pts

Since a is declared as a global variable, its memory resides in the global data area above the
text (i.e., code) of the running program. The global data area is not destroyed across calls to
myfunc(), hence its history of changes is preserved.
3 pts

P1(b) 12 pts

The declaration "char *s" only allocates memory for a pointer. Following the pointer (i.e., address
contained in s) is likely to lead to memory that does not belong to the running program of which the
code snippet is part..
6 pts

```
s = (char *) malloc(3 * sizeof(char));
```
6 pts

P1(c) 12 pts

```
unsigned int x, m, y;
m = ~(~0 << 1); // set up mask that 0 everywhere but at 0'th position
y = m & (x >> 7);
printf("%u", y); // printf("%d", y) is fine too
```
12 pts

P2(a) 20 pts

The main problem is that if the input is too long (over 1000 characters) then there may be a stack smashing
problem. A second issue is that r[] was not terminated with the end of string symbol '\0' which could cause

a problem.
10 pts

For the stack smashing problem: one way is to terminate the while-loop if i >= 1000 (i >= 999 is fine too).
An error message is then printed indicating that the input is too long. For the second issue, set
r[i] = '\0' after the while-loop.
10 pts

P2(b) 20 pts

An iterative server performs the requested task itself. Hence, it would call execlp() (or another variant
of exec()) itself which destroys the server's code and replaces it with the command passed to execlp().
This means that the iterative server can perform only one command and no more. A shell needs to stay alive
and accept further commands entered at its prompt.
4 pts

As seen in class, execlp() may fail such as when a bad command is passed. If this happens, then the child
process stays alive and executes the same code as the parent. That is, we now have two processes, parent
and child, running the concurrent shell program which is a bug. Hence if execlp() returns with -1, the child
process needs to be terminated.
4 pts

fork() returns 0 in the child process; it returns the PID of the child in the parent process. By checking the
return value, a process can determine whether it is the child (i.e., clone) or parent (i.e., original).
4 pts

Although rare, fork() can fail if there are too many processes in the system. Hence, a check needs to be done
in case fork() return -1 which means that the parent cannot handle the requested command.
4 pts

waitpid() makes the parent wait until the child, in our case, terminates. This agrees with the behavior of
shells such as bash, tcsh, csh, etc. that only show the prompt after the current command has completed.
2 pts

If waitpid() is not called, the parent immediately returns to showing the prompt and waiting for the next

command to be entered. In the meantime, the child process executes the requested command whose output will
appear after the prompt (in Linux the parent runs before the child) which is not what we expect from a typical
shell.
2 pts

P3 24 pts

```
// additional variables
int i, k, num;

// allocate num rows of string pointers
fscanf(fp,"%d",&num);
s = (char **) malloc(num * sizeof(char *));

// for each string pointer, allocate space for string
for(i=0; i<num; i++) {
  fscanf(fp,"%d",&k);
  *(s+i) = (char *) = malloc((k+1) * sizeof(char));
  fscanf(fp,"%s",*(s+i));
}
```

24 pts
[
deduct up to 8 pts for row malloc related errors,
deduct up to 8 pts for string malloc related errors,
deduct 2 pts in case of k instead of k+1,
deduct 2 pts per additional miscellaneous mistakes
]

Bonus 10 pts

printf(), as part of the stdio library, may not immediately send the requested print
output to the terminal display but keep it in RAM for efficiency purposes. In some
instances, when a process terminates abnormally (as in segmentation fault), the print
output remains is RAM without having been flushed to the display.
5 pts

fflush(stdout) forces the temporarily RAM buffered print output to be flushed to the
display. [A second solution seen in class was using the write() system call directly,
bypassing printf(). Both solutions are fine.]
5 pts