CS 351 Fall 2017

Midterm Exam

October 18^{th}, 2017

Instructions:

- This exam is closed-book, closed-notes. Calculators are not permitted.
- For numbered, multiple-choice questions, fill your answer in the corresponding row on the "bubble" sheet.
- For problems that require a written solution (labeled with the prefix "WP"), write your answer in the space provided on the written solution sheet. Please write legibly and clearly indicate your final answer.
- Turn in the exam question packet, bubble sheet, and written solution sheet separately.
- Good luck!

Multiple Choice (24 points):

1. Which exemplifies C's *weak type checking*?

   (a) the ability to assign pointers of any type to/from a `(void *)` variable without warning

   (b) the requirement that all variables be declared before being used

   (c) the static nature of variable types — e.g., once declared an `int`, always an `int`

   (d) the fact that data type widths (e.g., for `int`s) can be dependent on the platform

2. Consider the following macro definition and variable declaration:

```
#define FOO(x) (2 * x - x)
int val = 10;
```

   What is the value of the expression `FOO(val + 5)`?

   (a) 15

   (b) 20

   (c) 25

   (d) 35

3. Consider the following variable and function definitions:

```
int g = 10;
int q3() {
  static int g = 5;
  return ++g;
}

int q4() {
  extern int g;
  return ++g;
}

int q5() {
  int g = 1;
  return ++g;
}
```

   What is the value of the expression `q3() + q3() + q4() + q4() + q5() + q5()`?

   (a) 32

   (b) 34

   (c) 38

   (d) 40

4. Given the following variable declarations:

```
char *args[] = {"hello", "world"};
void *v = args;
```

Which prints out `hello world`?

(a) `printf("%c %c\n", *v, *(v + 1));`

(b) `printf("%s %s\n", *(char **)v, *((char **)v + 1));`

(c) `printf("%s %s\n", *(char *)v, *((char *)v + 1));`

(d) `printf("%s %s\n", **v, **(v + 1));`

5. Given the following code, which performs dynamic memory allocation:

```
void *v = 0, *p, *q;
for (int i=0; i<10; i++) {
  p = malloc(sizeof(void *));
  *(void **)p = v;
  v = p;
}
```

Which correctly deallocates the memory allocated above?

(a)
```
for (int j=0; j<10; j++) {
  free p[j];
}
```

(b)
```
for (int j=0; j<10; j++) {
  p = v[j];
  q = *p;
  free(p);
  p = q;
}
```

(c)
```
p = *(void **)v;
while (p) {
  q = *p;
  free(p);
  p = *q;
}
```

(d)
```
p=v;
while (p) {
  q = *(void **)p;
  free(p);
  p = q;
}
```

6. Which correctly completes the blanks in the following code so that the `todo` function is effectively called a total of 10 times?

```
void repeat(_________________, int n) {
  for (int i=0; i<n; i++) {
    _______________;
  }
}

void todo() {
  /* do something */
}

main() {
  repeat(_________________, 10);
}
```

   (a) `void (*f)()` / `f()` / `todo`
   (b) `void *f()` / `*f` / `&todo`
   (c) `(void)(*f())` / `*f()` / `*todo`
   (d) `void *(f)()` / `(*f)()` / `&todo`

7. Which is an example of an asynchronous exception?

   (a) a call to `fork`
   (b) a segmentation fault
   (c) unexpected loss of power to the system
   (d) a kernel panic due to accessing defective memory

8. Which is an example of a `trap`?

   (a) a call to `fork`
   (b) a segmentation fault
   (c) unexpected loss of power to the system
   (d) a kernel panic due to accessing defective memory

9. Which is **not** inherited by a child process from its parent (i.e., by way of `fork`-ing)?

   (a) group ID
   (b) signal handlers
   (c) registered `atexit` handlers
   (d) the "pending" signals vector

10. Which is **not** retained across a successful call to `exec`?

    (a) group ID
    (b) process ID
    (c) signal handlers
    (d) the "pending" signals vector

11. What condition(s) will lead to a process being "adopted" (and reaped upon termination) by the kernel?

    (a) when all its children have been reaped

    (b) when it terminates due to an uncaught signal

    (c) when its parent terminates

    (d) when it invokes the `exec` system call

12. Given the following global variable declarations:

```
char a, b;
```

    Which of the following functions is *reentrant*?

    (a)
```
void f_a() {
  char *p = &a;
  *p += 1;
}
```

    (b)
```
char f_b(char c) {
  c = a;
  c += b;
  return c;
}
```

    (c)
```
void f_c(char c) {
  c = a;
  a = b;
  b = c;
}
```

    (d)
```
void f_d() {
  static char c = 0;
  c -= a;
}
```

WP1. Process Trees (8 points):

For each of the following programs, (1) sketch the corresponding process tree — being sure to indicate any outputs and synchronization points, if they exist — and (2) write down at least three separate outputs that could be produced when it is executed.

A)
```
main() {
  for (int i=0; i<2; i++) {
    if (fork()) {
      printf("%d", i);
    } else {
      printf("%d", 3-i);
    }
  }
}
```

B)
```
main() {
  if (fork()) {
    printf("0");
    wait(NULL);
    printf("1");
  } else {
    if (fork()) {
      printf("2");
      wait(NULL);
      printf("3");
    } else {
      printf("4");
    }
  }
}
```

WP2. Signal Handlers (6 points):

For the following program, (1) sketch the corresponding process tree — being sure to indicate any outputs and synchronization points, if they exist — and (2) write down the output it will produce when executed.

```
int done = 0;

void handler(int sig) {
  printf("0");
  done = 1;
}

main() {
  pid_t pid;
  int status;
  signal(SIGUSR1, handler);
  if ((pid = fork()) == 0) {
    while (!done) ;
    printf("1");
    exit(2);
  } else {
    printf("3");
    kill(pid, SIGUSR1);
    if (wait(&status) > 0)
      printf("%d", WEXITSTATUS(status));
  }
  if (done)
    printf("4");
  else
    printf("5");
}
```

WP3. Shell Implementation (8 points):

Provide a rough implementation of the central loop of a UNIX shell that correctly handles *only foreground jobs*, even when an invalid command is entered. Your implementation should use the `fork`, `execv`, `wait`, and `exit` system calls (whose prototypes are given below), and demonstrate that you understand their semantics. Note that you do not need to implement any signal handlers nor handle any built-in commands.

- `pid_t fork(void);`
- `int execv(const char *path, char *const argv[]);`
- `pid_t wait(int *stat_loc);`
- `void exit(int status);`

To read and parse a command line input into an array of arguments, assume that the following code will suffice:

```
char cmdline[MAXLINE];
char *argv[MAXARGS];
fgets(cmdline, MAXLINE, stdin);
parseline(cmdline, argv);
```

State any assumptions in a comment at the start of your code.