CS 351 Fall 2019

Midterm Exam

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 (30 points):

Choose the *single best answer* to each question.

1. Which best describes the type of `x` in the following C declaration?

   ```
   int (*x[10])(char *);
   ```

   (a) a pointer to a function that takes an array of 10 strings and returns an `int`
   (b) an array of 10 pointers to functions that take strings and return `int`s
   (c) a pointer to a function that takes a string and returns an array of 10 `int`s
   (d) a function that takes an array of 10 strings and returns an `int`

2. Consider the following C declarations:

   ```
   struct s {
     int x, y;
   };

   struct s sarr[20];
   ```

   Which of the following expressions is semantically equivalent to "`sarr[10].x`"?

   (a) `(sarr + 10)->x`
   (b) `(&sarr + 10).x`
   (c) `*(sarr + 10 * sizeof(struct s)).x`
   (d) `((char *)sarr + 10 * sizeof(struct s))->x`

3. At which stage of the (extended) compilation process are `#define`'d symbols replaced with their values?

   (a) Preprocessing
   (b) Compilation
   (c) Assembly
   (d) Linking

4. Which of the following constitutes a memory leak in a C program?

   (a) forgetting to `free` a pointer declared as `static`
   (b) calling `fork` before returning to the `main` function
   (c) returning a pointer to a locally declared array from a function
   (d) failing to `free` a previously dynamically allocated block of memory

5. What is a *synchronous* exception triggered by?
   (a) an I/O device
   (b) the foreground job
   (c) the currently executing instruction
   (d) the interrupt vector

6. Which of the following is *not* an example of an interrupt?
   (a) a system call
   (b) a keystroke (e.g., ctrl-C)
   (c) a disk controller event
   (d) the arrival of network data

7. After a fault is handled by the kernel, where does control typically return to in the user program?
   (a) the corresponding fault handler
   (b) the beginning of `main`
   (c) the return address of the current function (stored on the stack)
   (d) the instruction that generated the fault

8. Which of the following is *not* inherited by a child process from its parent when `fork`-ing?
   (a) `atexit` handlers
   (b) pending signals
   (c) signal handlers
   (d) blocked signals

9. Under what condition(s) does a process turn into a zombie after terminating?
   (a) when it has been orphaned
   (b) when the parent has previously invoked `wait` (or a variant)
   (c) when it is run as a background job
   (d) All of the above

10. What action does the kernel take when a signal arrives for a process that is currently executing the handler in response to a previous signal of the same type?
    (a) it preempts the handler and re-enters it from the beginning
    (b) it ignores the signal (i.e., it neither delivers it nor marks it as pending)
    (c) it marks the signal as pending, but doesn't deliver it
    (d) it blocks the signal to prevent additional signals of that type from being delivered

11. When it comes to implementing a *reentrant* function, which of these actions is most likely "safe" to perform (i.e., won't make the function non-reentrant)?
    (a) reading a global data structure
    (b) modifying a global variable
    (c) modifying a local variable
    (d) calling another (possibly non-reentrant) function

12. Which of the following statements concerning signal handling is *false*?
    (a) it is not possible to accurately determine how many signals of a given type were sent over a given period
    (b) when delivering a signal, the kernel informs the receiver of the pid of the sending process
    (c) signals are prioritized based on their position in the pending and blocked vectors
    (d) signal handlers are executed in user mode (i.e., not as the kernel)

13. Which of the following statements is *true* following a successful call to `exec`?
    (a) any child processes will be orphaned and adopted by the kernel
    (b) there is no return to the calling program
    (c) the process group id will be set equal to the process id
    (d) All of the above

14. What best describes the purpose of the `kill` system call?
    (a) it will immediately terminate the identified process
    (b) it is used to register a handler for the `SIGINT` signal
    (c) it is used to send a signal to a process
    (d) it is the counterpart to the `exec` system call

15. What is responsible for deciding whether to switch to a different process during the kernel's exception handling procedure?
    (a) the scheduler
    (b) the clock interrupt
    (c) the interrupt vector
    (d) All of the above

WP1. Memory Management (8 points):

Consider the following code, which contains a type definition and a function that uses it to dynamically allocate a structure in memory.

```
typedef struct pyr pyr_t;

struct pyr {
  int n;
  int **levels;
};

pyr_t *alloc_pyr(int n) { // assume n > 0
  pyr_t *p = malloc(sizeof(pyr_t));
  p->n = n;
  p->levels = malloc(n * sizeof(int *));
  for (int k=0; k<n; k++) {
    p->levels[k] = malloc((k+1) * sizeof(int));
  }
  return p;
}
```

Complete the implementation of `void free_pyr(pyr_t *p);`, which, when called with a pointer to a structure returned by a call to `alloc_pyr` (with an arbitrary argument $n > 0$), will correctly free *all* the memory allocated for the structure. E.g., `free_pyr(alloc_pyr(10))` should result in no memory leaks or errors.

WP2. Process Trees (8 points):

For each of the following programs, (1) sketch the corresponding process tree — being sure to indicate outputs and circle synchronization points, if they exist — and (2) list the outputs that could be produced when it is executed. If there are multiple possible outputs, you need list only three distinct ones.

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

B)
```
main() {
    if (fork() == 0) {
      printf("0");
      for (int i=1; i<3; i++) {
        if (fork() == 0) {
          printf("%d", i);
          exit(0);
        }
      }
      printf("3");
    } else {
      wait(NULL);
      printf("4");
    }
}
```

WP3. Signal Handlers (8 points):

Consider the following program:

```
int counter = 0;

void handler (int sig) {
  counter++;
}

int main() {
  signal(SIGUSR1, handler);
  signal(SIGUSR2, handler);
  if (fork() == 0) {
    /* insert snippet here */
    exit(0);
  }
  wait(NULL);
  printf("%d\n", counter);
  return 0;
}
```

Replacing the comment in the above code with each of the snippets below, indicate *all* possible outputs of the program (i.e., the printed value of `counter`) and briefly explain why they may occur. Assume that no external signals are sent to the process. Note that `SIGUSR1` and `SIGUSR2` correspond to signal numbers 30 and 31, respectively.

A)
```
kill(getppid(), SIGUSR1);
kill(getppid(), SIGUSR1);
```

B)
```
kill(getppid(), SIGUSR1);
kill(getppid(), SIGUSR1);
kill(getppid(), SIGUSR1);
```

C)
```
kill(getppid(), SIGUSR2);
kill(getppid(), SIGUSR1);
```

D)
```
kill(getppid(), SIGUSR2);
kill(getppid(), SIGUSR1);
kill(getppid(), SIGUSR1);
kill(getppid(), SIGUSR1);
```

WP4. Why `fork` and `exec`? (8 points):

As you've discovered, Unix provides separate `fork` and `exec` APIs, whereas some other operating systems provide a single API used for creating processes and running new programs in them. List three distinct reasons why separating `fork` and `exec` is a good API design decision. Support your reasons with concise examples.