CS 351 Spring 2020

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?

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

   (a) a pointer to an array of 10 pointers to functions returning strings
   (b) a pointer to a function that takes an array of 10 void pointers and returns
   (c) an array of 10 pointers to functions returning strings
   (d) an array of functions that take arrays of 10 character pointers a string

2. Consider the following C declaration:

```
int iarr[100];
void *p = iarr;
```

   Which of the following expressions is semantically equivalent to "`iarr[50]`"?

   (a) `*(int *)((char *)p + 50 * sizeof(int))`
   (b) `*(int *)(p + 50 * sizeof(int *))`
   (c) `((int *)((char *)p + 50))[0]`
   (d) `*(char *)((int *)p + 50)`

3. At which stage of the compilation process are `extern`'d symbols resolved between files?

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

4. In which of the following situations is it arguably unnecessary to `free` dynamically allocated structures that are no longer needed by a process?

   (a) when the function that allocated the structures is about to return
   (b) when the structures are no longer reachable from static memory
   (c) when the process is about to reap a child
   (d) when the process is about to terminate

5. What class of exceptions does a system call belong to?

   (a) aborts
   (b) traps
   (c) faults
   (d) signals

6. What in-memory structure is consulted to determine what handler to execute for a hardware-level interrupt?

   (a) the process control block
   (b) the interrupt vector
   (c) the pending vector
   (d) the blocked vector

7. Which of the following happens *first*, following a hardware interrupt that occurs during the execution of a user-level process?

   (a) context switch to a new process
   (b) execution of a registered signal handler
   (c) termination of the interrupted process
   (d) transition to kernel mode

8. Which of the following is inherited by a child process from its parent when `fork`-ing?

   (a) process ID
   (b) pending signals
   (c) blocked signals
   (d) parent process ID

9. Which category of process does the kernel take over responsibility for reaping?

   (a) those whose parents have already terminated
   (b) those that terminate abnormally (i.e., without calling `exit`)
   (c) those that terminate due to a signal
   (d) those that have unterminated children

10. Which of the following statements about signal delivery is *true* by default?

    (a) multiple signals of the same kind are queued
    (b) when a process receives the `SIGINT` signal, it is also delivered to all of its children
    (c) if a signal is blocked, newly arriving signals of that type will not be made pending
    (d) while a given signal is being handled, it will not be delivered again

11. Which of the following actions is most likely to cause the containing function to be non-reentrant?

    (a) modifying a local variable
    (b) calling itself recursively
    (c) carrying out a lengthy loop
    (d) modifying a static data structure

12. In your implementation of a shell, you invoked the `waitpid` system call in a loop within the `SIGCHLD` handler. Why was this important?

    (a) multiple child processes terminating could result in only one `SIGCHLD` being delivered
    (b) failing to do this would make all child processes background jobs
    (c) doing this ensures that we reap all descendants (in addition to immediate children)
    (d) this prevents the kernel from adopting processes that we fail to reap in the handler

13. In which situation is a handler function registered via `atexit` *not* executed on process termination?

    (a) when the process returns from `main` instead of calling `exit` to terminate
    (b) when the process inherits its handler via a `fork`
    (c) when the process terminates due to a segmentation fault
    (d) when the process is due to become a zombie after terminating

14. Consider a scenario where a process running program `A` performs an `exec` in order to run program `B`. What happens when program `B` terminates the process by calling `exit`?

    (a) program `A`, which was preempted, will resume execution
    (b) the process will turn into a zombie
    (c) the kernel will send a `SIGCHLD` to program `A`
    (d) program `A` is now responsible for reaping `B`

15. Consider a scenario where a single handler function has been registered for two distinct signals. Given that the higher priority signal has just been delivered and the handler is currently executing, what happens if the lower priority signal arrives?

    (a) the higher priority handler is preempted and the lower priority one is run
    (b) the signal is marked as pending but is not delivered
    (c) the lower priority handler is started, then preempted to return to the higher priority one
    (d) the signal is not marked as pending, as all signals are blocked while the higher priority handler is being run

WP1. Process Trees (10 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 all distinct outputs that could be produced when it is executed.

A)
```
main() {
  int x = 0;
  printf("1");
  if (fork() == 0) {
    printf("2");
    x += 1;
  } else if (fork() == 0) {
    printf("3");
    x += 1;
  }
  if (x < 1) {
    while (wait(NULL) > 0) ;
    printf("4");
  } else {
    printf("5");
  }
}
```

B)
```
main() {
  printf("1");
  if (fork() == 0) {
    for (int i=2; i<4; i++) {
      printf("%d", i);
      if (fork() == 0) {
        printf("%d", i+10);
        exit(0);
      }
      wait(NULL);
    }
  } else {
    wait(NULL);
    printf("4");
  }
}
```

WP2. Signal Handling (10 points):

Consider the following program:

```
int arr[100] = { 0 };
int idx = 0;
int val = 10;

void sighandler(int sig) {
  arr[idx] = val;
  idx++;
  val--;
}

main() {
  signal(SIGUSR1, sighandler);
  signal(SIGUSR2, sighandler);

  if (fork() == 0) {
    for (int i=0; i<10; i++) {
      kill(getppid(), SIGUSR2);
      kill(getppid(), SIGUSR1);
    }
    exit(0);
  } else {
    wait(NULL);
    for (int i=0; i<10; i++) {
      printf("%d ", arr[i]);
    }
  }
}
```

A) During testing, we discover that the output of the program is unpredictable, with separate test runs producing the following outputs:

```
10 9 8 7 6 5 4 3 2 1

10 9 8 7 0 0 0 0 0 0
```

Explain how these different outputs might be produced.

B) Occasionally, the program behaves even more oddly, and produces output like this:

```
10 10 9 8 6 6 5 3 2 1
```

Explain how this output might be produced.

WP3. Modified Shell (10 points):

For this problem you will complete the implementation of a shell with the following behavior:

- All commands are effectively run as background jobs (i.e., the shell will resume reading commands after they are started).
- When background jobs terminate, they should be reaped as soon as possible.
- If the total number of running background jobs is equal to a threshold (`MAX_BG`), the shell should stop accepting new commands until *all* its children have been reaped. For this purpose, the shell should maintain a count (`n_children`) of running background jobs.

On the written worksheet we provide a template for your solution. Note:

- All your code should go in the `main` and `sigchld_handler` functions. You should not modify other or add new functions.
- You can call `toggle_sigs` with `1` to block the `SIGCHLD` signal or `0` to unblock it.
- Assume that `parseline` behaves the way it did in your shell lab implementation. Also assume that all system calls are successful (i.e., you do not need to handle errors).
- You do not need to call any of the job management functions from the shell lab.