Lab 1 DRAFT: wait and exec (23.1.25)

This lab will introduce you to the `wait()` and `exec()` system calls. The *exec* family of functions are used to replace the current process image with a new process image. Here is a link to example exec calls (and other details about the exec functions).

Review Problems

Assume there are no errors (e.g. `fork` doesn't fail), but make no assumptions about how the processes are scheduled. Assume each call to `printf` flushes its output out just before it returns.

1. Examine the following program and circle *all* corresponding combinations of output that may be produced when it is run.

```
int main() {
   if (fork() != 0) {
      wait(NULL);
      printf("0");
   } else if (fork() == 0) {
      printf("1");
   } else {
      fork();
      printf("2");
      exit(0);
   }
   printf("3");
   return 0;
}
```

a. 213032 b. 123023 c. 130322

d. 132203 e. 220133 f. 022133

Possible - a, b, d, e
Not possible - c, f

2. Examine the following program and circle *all* corresponding combinations of output that may be produced when it is run.

```
int main() {
   fork();
   printf("0");
   if (fork() == 0) {
      /* this next exec prints "1" */
      execl("/bin/echo", "echo", "1", NULL);
      fork();
      printf("2");
  }
  printf("3");
  return 0;
}
```

a. 00112233223333 b. 0132310323 c. 013013

d. 001133 e. 013310 f. 030311

Possible - c, d, f
Not possible – a, b, e

3. Given the following program:

```
int main(int argc, char *argv[]) {
     int i;
     for (i = 1; i < argc; i++)
          execvp(argv[i], &(argv[i]));
     return 0;
}
```

Suppose that the above program is compiled in the current directory as `myprog` and that one gives the command:

```
$ myprog myprog ls myprog myprog myprog
```

Assume all system calls execute without error.

(a) How many times will `execvp()` be called during the run of this program? Explain.

2 times

(b) What will be the output of the run? Explain.

`myprog  myprog  myprog`

Programming task (to be checked off)

Write a program `tryit` that takes a command-line argument of the path to the program (absolute or relative), forks a child that tries to `exec()` the given program, and reports on its success or failure. A child that exits with a status of 0 is assumed to be success, non-zero is a failure. If the `exec()` fails, the child should print why (via `perror()`), and exit with a non-zero status.

This program does not have to support command line arguments to the other program. The working of the program is shown below.

Parent	**Child**
---- *before the fork* ----	
check the command line args	----
fork()	
---- *after the fork* ----	
wait() for child	exec the given program
---- *after the exit* ----	
report on child's success exit with child's status	---- ----

Sample Runs

```
$ ./tryit
usage: tryit command
$ ./tryit command with args
usage: tryit command
$ ./tryit non-existant
non-existant: No such file or directory
Process 2359 exited with an error value.
$ ./tryit ls
ls: no such file or directory
Process 2361 exited with an error value.
$ ./tryit /bin/ls
Makefile RCS forkit forkit.c tryit tryit.c
Process 2369 succeeded.
$ ./tryit /bin/false
Process 2371 exited with an error value.
$ ./tryit /bin/true
Process 2373 succeeded.
```

As seen in the example above, `/bin/true` always exits with a successful exited code, and `/bin/false` always exits with an unsuccessful one.