Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 8: Exceptional Control Flow
May 10, 2024

## Practice Problems

**Exercise 8.1.** Consider three processes with the following starting and ending times:

| Process | Start time | End time |
|---|---|---|
| A | 0 | 2 |
| B | 1 | 4 |
| C | 3 | 5 |

For each pair of processes, indicate whether they run concurrently (Y) or not (N).

**Solution:** A pair of processes is concurrent if one starts after the other begins and before the other one ends. A and B are concurrent because B starts at $t = 1$, which is after the start time of A ($t = 0$) and the end time of A ($t = 2$). A and C are not concurrent because A starts at 0 and C has not yet begun (it starts at $t = 3$). Lastly, B and C are concurrent because C begins at $t = 3$ after B has started ($t = 1$) but before B ends ($t = 4$).

| Process pair | Concurrent? |
|---|---|
| AB | Yes |
| AC | No |
| BC | Yes |

**Exercise 8.2.** Consider the following program:

```c
int main()
{
    int x = 1;

    if (Fork() == 0)
        printf("p1: x=%d\n", ++x);
    printf("p2: x=%d\n", --x);
    exit(0);
}
```

(a) What is the output of the child process?

(b) What is the output of the parent process?

**Solution:**

(a) `fork()` system call in the child process returns 0. Therefore, both print statements execute. The output is:

```
p1: x=2
p2: x=1
```

(b) The `fork()` system call in the parent returns the process ID of the child, which is guaranteed to be a positive integer. Therefore, only the last print statement executes:

```
p2: x=0
```

**Exercise 8.3.** List all of the possible output sequences for the following program:

```c
int main()
{
    if (Fork() == 0) {
        printf("a"); fflush(stdout);
    }
    else {
        printf("b"); fflush(stdout);
        waitpid(-1, NULL, 0);
    }
    printf("c"); fflush(stdout);
    exit(0);
}
```

**Solution:**

1. One possibility is that the child process completes all of its instructions before the parent gets a time slice after the work. Then the parent displays `b`, calls `waitpid` to reap the child but returns immediately because the child has already exited, and finally, the parent displays `c` before it too exits:

   ```
   acbc
   ```

2. Another possibility is that the child gets control after the work before the parent, displays `a`, but is then preempted. The parent begins executing and displays `b`. Then either it executes `waitpid` or is preempted before it does, allowing the child to display `c`. Finally, the parent displays `c`:

   ```
   abcc
   ```

3. Another possibility is that immediately after the work, the parent has the opportunity to displays `b`. Then the child gets control because the parent's time slice ends or because the parent calls `waitpid`. At this point, the child displays `a`. Since it has not terminated, the parent remains suspended due to `waitpid`, so the child gets to display `c`. Finally the parent reaps the child and displays `c`:

```
bacc
```

**Exercise 8.4.** Consider the following program:

```c
int main()
{
    int status;
    pid_t pid;

    printf("Hello\n");
    pid = Fork();
    printf("%d\n", !pid);
    if (pid != 0) {
        if (waitpid(-1, &status, 0) > 0) {
            if (WIFEXITED(status) != 0)
                printf("%d\n", WEXITSTATUS(status));
        }
    }
    printf("Bye\n");
    exit(2);
}
```

(a) How many output lines does this program generate?

(b) What is one possible ordering of these output lines?

**Solution:**

(a) Assuming no errors, the child displays two lines: it executes `printf` statement immediately following the `Fork()`, skips the `if` block because its `pid` value is 0, and displays the `Bye` message. The parent, on the other hand, displays three lines: the one immediately after the `Fork()`, the exit status of the child inside the nested `if`, and then `Bye`. In total there are 5 output lines.

(b) Since the `stdout` stream from `stdio.h` is line-buffered by default, all lines will be displayed in the order they execute because they all have a
`n` as the last character in their strings. The initial `Hello` is always printed first by the parent. One possibility is that the parent gets control immediately after the work, displaying the result of the second `printf` call. Since the result of the `fork` call in the parent is the process ID of the child, the value of `pid` is a positive integer, so the expression `!pid` evaluates to 0, which in turn is displayed.

Control is then gained by the child because either the parent is preempted or because it is willfully suspended as a result of the `waitpid` call. The child now calls its first `printf` call; since `pid` is 0 in the child, the result is that `!pid` is 1, so that its printed to standard out. Since the parent is waiting for the parent to terminate, the child is able to print its next statement, displaying `Bye`. Assuming no errors occur and the `exit(2)` call runs successfully, the `waitpid` calls returns the (positive) process ID of the child, entering the branch of the if-statement. The `WIFEXITED` macro detects that the child exited normally by calling `exit`, so the next if-statement also evaluates to true. Then the `WEXITSTATUS` macro returns the exit status, which was 2, and that value is displayed by `printf`. Finally,t he `Bye` message is displayed by the parent:

```
Hello
0
1
Bye
2
Bye
```

**Exercise 8.5.** Write a wrapper function for `sleep`, called `snooze`, with the following interface:

```
unsigned int snooze(unsigned int secs);
```

The `snooze` function behaves exactly as the `sleep` function, except that it prints a message describing how long the process actually slept:

```
Slept for 4 of 5 secs.
```

**Solution:** Below is my implementation, which is found at `./05-snooze/snooze.c`:

```
#include <unistd.h> /* sleep() */
#include <stdio.h> /* printf() */
#include "snooze.h"

unsigned int
snooze(unsigned int secs)
{
    unsigned int remaining = sleep(secs);
    printf("Slept for %d of %d secs\n", secs - remaining, secs);
    return remaining;
}
```

**Exercise 8.6.** Write a program called `myecho` that prints its command-line arguments and environment variables. For example:

```
linux> ./myecho arg1 arg2
Command-line arguments:
    argv[ 0]: myecho
    argv[ 1]: arg1
    argv[ 2]: arg2
```

```
Environment variables:
    envp[ 0]: PWD=/usr0/doh/ics/code/ecf
    envp[ 1]: TERM=emacs
    .
    .
    .
    envp[25]: USER=droh
    envp[26]: SHELL=/usr/local/bin/tcsh
    envp[27]: HOME/usr0/droh
```

**Solution:** Below is my implementation, which is found at `./06-myecho/myecho.c`:

```c
#include <stdio.h> /* printf() */

int
main(int argc, char *argv[], char *envp[])
{
    /* Compute width of argv indices */
    char buf[BUFSIZ];
    int argvIndexWidth = snprintf(buf, BUFSIZ, "%d", argc);

    /* Display command line arguments */
    printf("Command-line arguments\n");
    for (int i = 0; i < argc; i++)
        printf("\targv[%*d]: %s\n", argvIndexWidth, i, argv[i]);

    /* Count number of env variables and compute envp index width */
    int envpCount = 0;
    for (char **p = envp; *p != NULL; p++)
        envpCount++;
    int envpIndexWidth = snprintf(buf, BUFSIZ, "%d", envpCount);

    /* Display environment variables */
    printf("Environment variables:\n");
    for (int i = 0; i < envpCount; i++)
        printf("\tenvp[%*d]: %s\n", envpIndexWidth, i, envp[i]);
}
```