Multiprocessing: `wait` and `exec`

COSC 208: Intro to Computer Systems

Week 12, Wednesday (19 April 2023)

Warm-up

- Q1: *What does the following code output?*

```
int main() {
    int x = 10;
    int y = 20;
    int retval = fork();
    if (retval == 0) {
        y -= 5;
    } else {
        y+= 5;
    }
    printf("x=%d y=%d\n", x, y);
    return 0;
}
```

```
x=10 y=25
x=10 y=15
```

OR

```
x=10 y=15
x=10 y=25
```

Q2. Consider the following program. How many processes are created as a result of running this program?

```
#include <unistd.h>

int main() {
    fork();
    fork();
    sleep(5); // pause the process for 30 seconds
    return 0;
}
```

3.

Q3. Consider the following program. How many times does `here` get printed?

```
#include <stdio.h>
#include <unistd.h>

int main() {
    if (fork()) {
        fork();
    }
    fork();
    printf("here\n");
    sleep(5); // pause the process for 5 seconds
    return 0;
}
```

```
#include <stdio.h>
#include <unistd.h>
```

```c
int main() {
    if (fork()) {
        fork();
    }
    fork();
    printf("here\n");
    sleep(5); // pause the process for 30 seconds
    return 0;
}
```

```
here
here
here
here
here
here
```

Q4. What values of `i` are printed by this program, and how many times?

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int i = 0;
    fork();
    i++;
    fork();
    i++;
    printf("%d\n", i);
    sleep(5); // pause the process for 30 seconds
    return 0;
}
```

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int i = 0;
    fork();
    i++;
    fork();
    i++;
    printf("%d\n", i);
    sleep(5); // pause the process for 30 seconds
    return 0;
}
```

```
2
2
2
2
```

- Q5: *What is the output produced by running `./progA`, assuming no errors occur?*

progA:

```c
int main() {
    pid_t a = fork();
```

```
    if (a == 0) {
        char *cmd[] = { "./progB", NULL };
        execv(cmd[0], cmd);
        printf("A 2nd gen\n");
        return 0;
    } else {
        wait(NULL);
        printf("A 1st gen\n");
        return 0;
    }
}
```

progB:

```
int main() {
    pid_t b = fork();
    if (b == 0) {
        printf("B 2nd gen\n");
        return 0;
    } else {
        wait(NULL);
        printf("B 1st gen\n");
        return 0;
    }
}
```

```
B 2nd gen
B 1st gen
A 1st gen
```

Extra practice

- Q7: *What are all possible outputs of this program (assuming the new process has PID 13346)?*

```
int main() {
    int pid = fork();
    printf("A %d\n", pid);
    if (pid == 0) {
        printf("B\n");
    } else {
        wait(NULL);
        printf("C\n");
    }
}
```

```
A 0
A 13346
B
C
```

OR

```
A 13346
A 0
B
C
```

OR

```
A 0
```

B
A 13346
C

- Q8: *What does the following code output?*

```
int main(int argc, char **argv) {
    int value = 100;
    int pid = fork();
    if (pid == 0) {
        value -= 50;
    } else {
        value += 50;
    }
    printf("My value is %d\n", value);
    return 0;
}
```

```
My value is 50
My value is 150
```

OR

```
My value is 150
My value is 50
```

- Q10: *How would you modify the above program such that* `Child` *always prints before* `Parent`*?*

```
int main(int argc, char **argv) {
    printf("Begin\n");
    int pid = fork();
    if (pid == 0) {
        printf("Child\n");
        return 0;
    } else {
        wait(NULL);
        printf("Parent\n");
    }
    printf("End\n");
}
```

Q11. What is printed by each process created by running the following program? You can assume that process IDs start at 1000 and increment sequentially.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    for (int i = 0; i < 2; i++) {
        fork();
        printf("%d %d\n", getpid(), i);
    }
}
```

Note that fork lines below aren't in output; they just indicate where/when forks happen

1000 (parent)

```
(fork - first child)
1000 0
(fork - second child)
1000 1
```

1001 (first child)

```
1001 0
(fork – third child)
1001 1
```

1002 (second child)

```
1002 1
```

1003 (third child)

```
1003 1
```

Q12. A *forkbomb* is a program that creates an unlimited number of new processes in an effort to cripple a computer (this is a type of *denial of service* attack). Write a forkbomb.

Note: most modern operating systems have internal measures to limit the impact of forkbombs, but beware: if you compile and run your program you may need to power-cycle the system to regain control.

```
int main() {
    while (1) {
        fork();
    }
}
```