CSC 112: Computer Operating Systems
Lecture 2

Processes and Threads Exercises

Department of Computer Science,
Hofstra University

Wait() I

```
int main() {
    int i;

    for (i = 0; i < 2; i++) {
        pid_t pid = fork();

        if (pid == 0) {
            // Child process
            printf("Hello %d\n", i);
            return 0;  // Exit child process
        } else if (pid > 0) {
            // Parent process
           wait(NULL);  // Wait for immediate child to
terminate
        }
    }

    printf("Parent exiting\n");
    return 0;
}
```

- Due to the use of wait(NULL), the parent waits for each child to complete before creating another child. This enforces sequential execution, meaning there is no interleaving between outputs from different iterations.
  - Hello 0
  - Hello 1
  - Parent exiting
- “return 0” here is the same as “exit()”

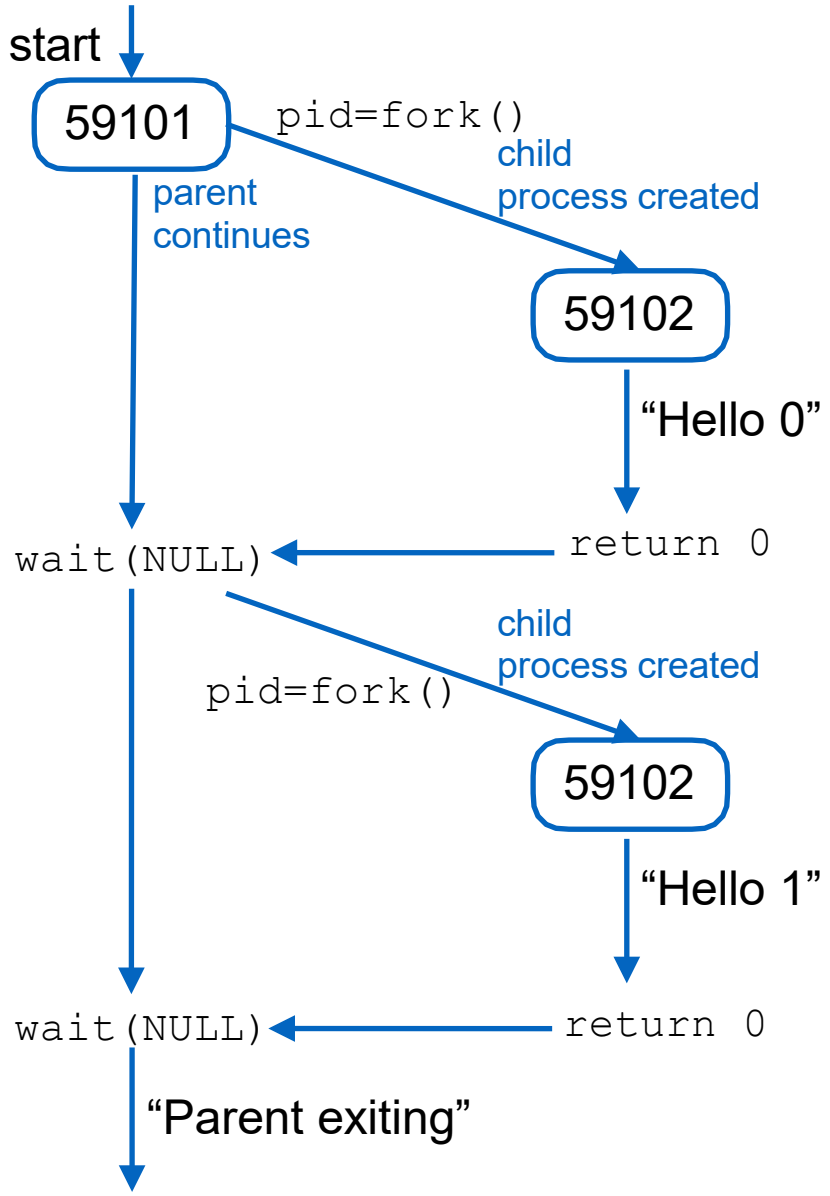
Wait() I

```
int main() {
    int i;

    for (i = 0; i < 2; i++) {
        pid_t pid = fork();

        if (pid == 0) {
            // Child process
            printf("Hello %d\n", i);
            return 0;  // Exit child process
        } else if (pid > 0) {
            // Parent process
           wait(NULL);  // Wait for immediate child to
terminate
        }
    }

    printf("Parent exiting\n");
    return 0;
}
```



Wait() I with exec()

```
int main() {
    int i;

    for (i = 0; i < 2; i++) {
        pid_t pid = fork();

        if (pid == 0) {
            // Child process
            printf("Hello %d\n", i);
            exec(SOME_COMMAND); //SOME_COMMAND is a
Linux command that does not print anything
            printf("Hello again %d\n", i);
          return 0;  // Exit child process
        } else if (pid > 0) {
            // Parent process
            wait(NULL);  // Wait for immediate child to
terminate
        }
    }

    printf("Parent exiting\n");
    return 0;
}
```

- In Child process: exec() replaces the current process image with a new program called SOME_COMMAND. The child process will execute the command and terminate. The code following it (e.g., printf("Child\n")) will not be executed because it is now running SOME_COMMAND, not the code shown in the text box.
- Output:
  - Hello 0
  - Hello 1
  - Parent exiting

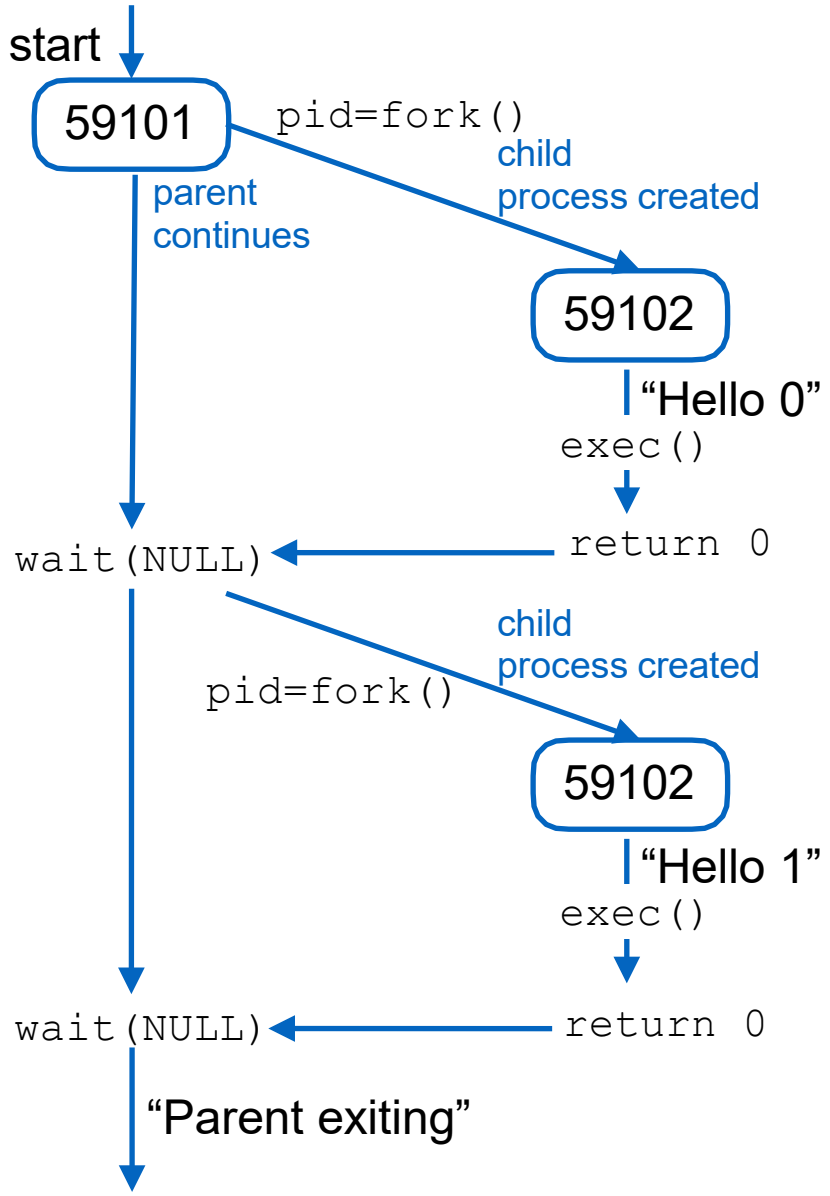
Wait() I with exec()

```
int main() {
    int i;

    for (i = 0; i < 2; i++) {
        pid_t pid = fork();

        if (pid == 0) {
            // Child process
            printf("Hello %d\n", i);
            exec(SOME_COMMAND); //SOME_COMMAND is a
Linux command that does not print anything
            printf("Hello again %d\n", i);
            return 0;  // Exit child process
        } else if (pid > 0) {
            // Parent process
            wait(NULL);  // Wait for immediate child to
terminate
        }
    }

    printf("Parent exiting\n");
    return 0;
}
```



Wait() II

```
int main() {
    int i;

    for (i = 0; i < 2; i++) {
        pid_t pid = fork(); // Create a child process

        if (pid == 0) {
            // Child process
            printf("Hello %d\n", i);
            return 0; // Exit child process
        } else if (pid > 0) {
            // Parent process continues to next
iteration
            continue;
        }
    }

    // Parent process waits for all child processes to
terminate
    if (pid > 0) {
      for (i = 0; i < 2; i++) {
        wait(NULL); // Wait for a child process to
terminate
      }
    }
    printf("Parent exiting\n");
    return 0;
}
```

- Since the parent does not wait immediately after creating each child, the outputs of "Hello" messages from children can interleave. However, due to the final waiting loop (wait(NULL)), "Parent exiting" is always printed last.
- Two possible outputs:
  - Hello 0
  - Hello 1
  - Parent exiting
- Or
  - Hello 1
  - Hello 0
  - Parent exiting

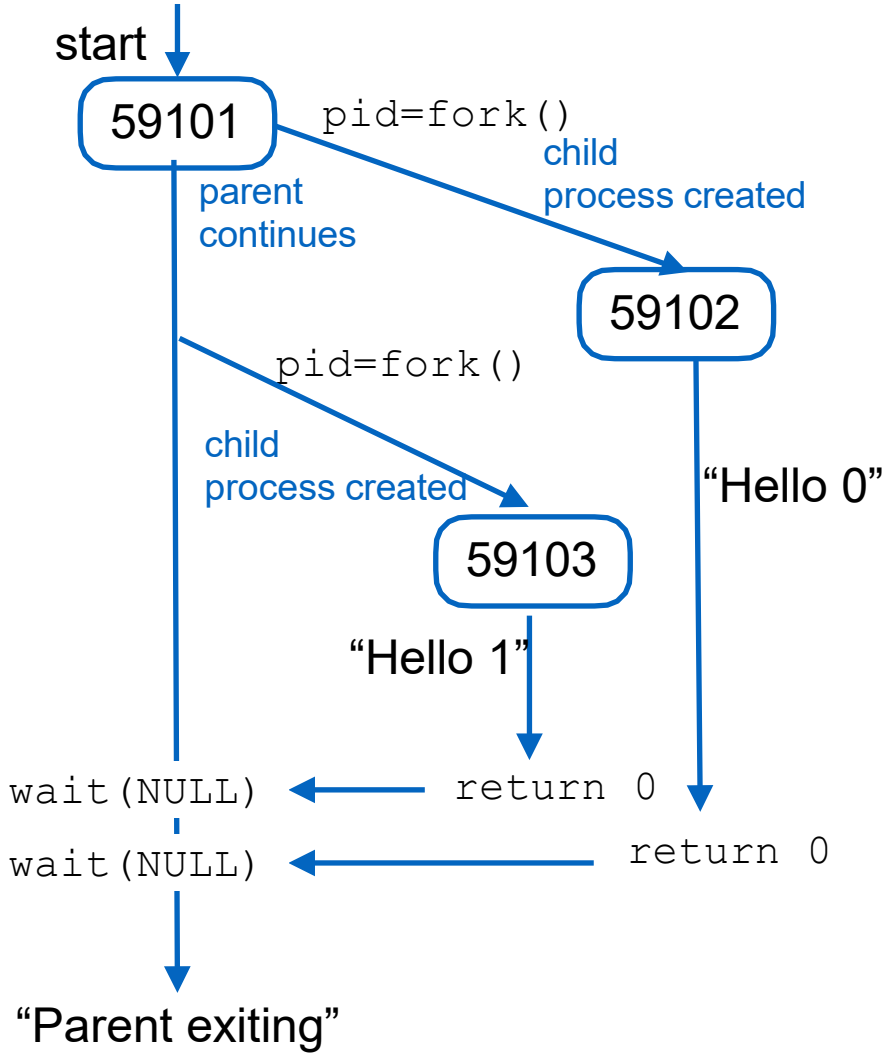
Wait() II

```
int main() {
    int i;

    for (i = 0; i < 2; i++) {
        pid_t pid = fork(); // Create a child process

        if (pid == 0) {
            // Child process
            printf("Hello %d\n", i);
            return 0; // Exit child process
        } else if (pid > 0) {
            // Parent process continues to next
iteration
            continue;
        }
    }

    // Parent process waits for all child processes to
terminate
    if (pid > 0) {
      for (i = 0; i < 2; i++) {
        wait(NULL); // Wait for a child process to
terminate
      }
    }
    printf("Parent exiting\n");
    return 0;
}
```



Either child process may finish first, and Parent uses wait(NULL) to wait for ANY child process to finish.