CSC 112: Computer Operating Systems
Lecture 2

Processes and Threads Exercises

Department of Computer Science,

Hofstra University

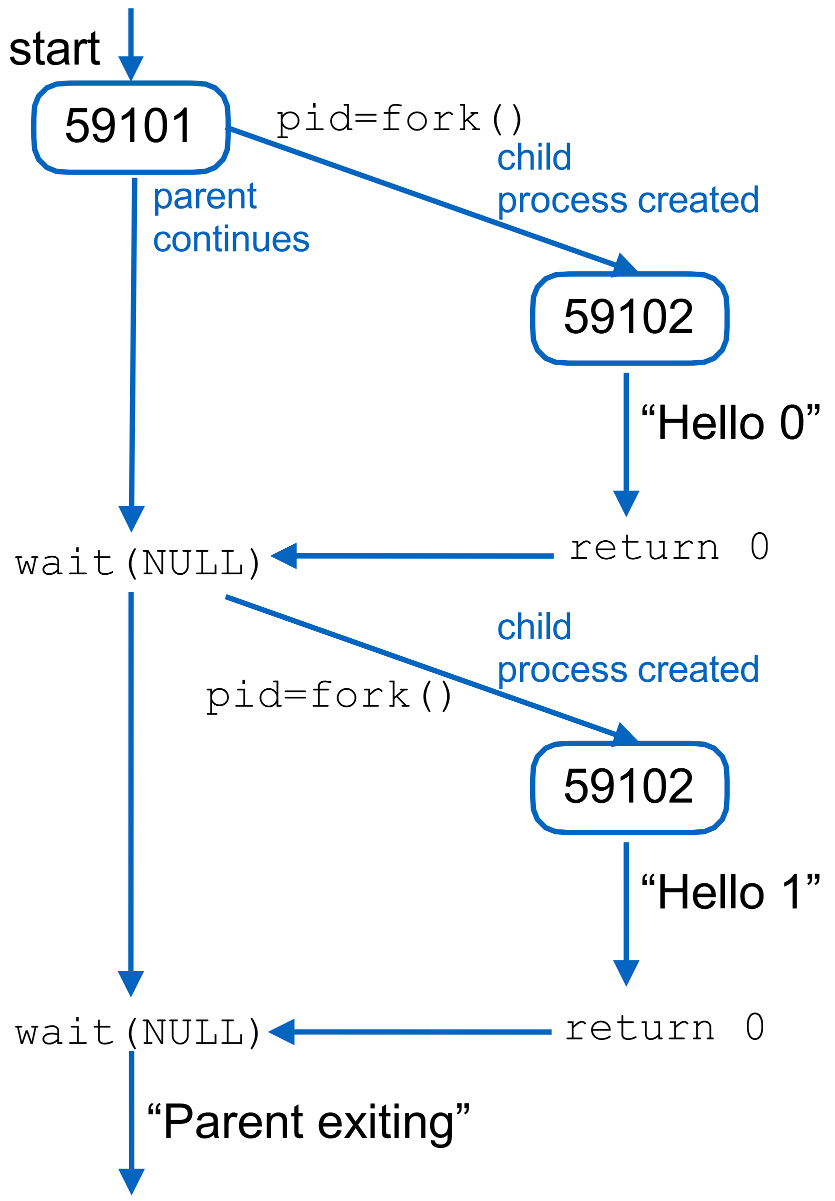
Wait() I

```
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;
}
```



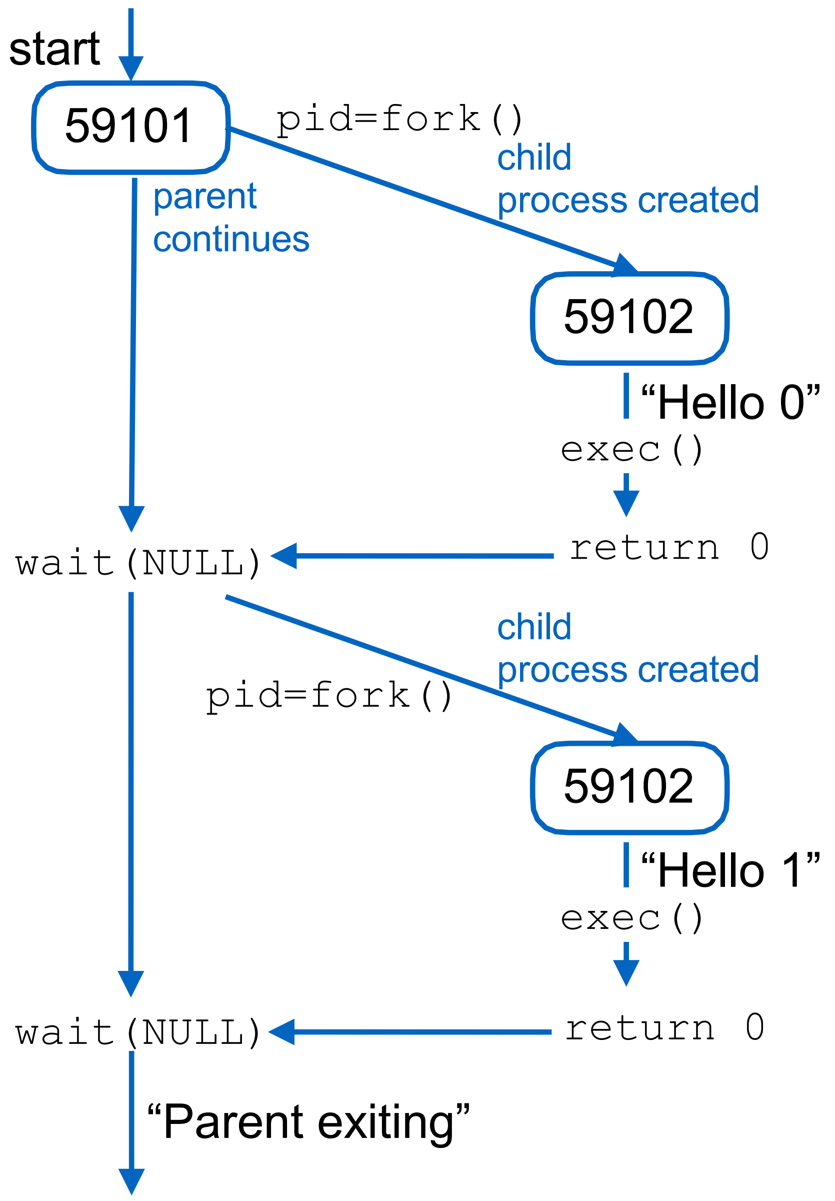
Wait() I with exec()

```
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;
}
```



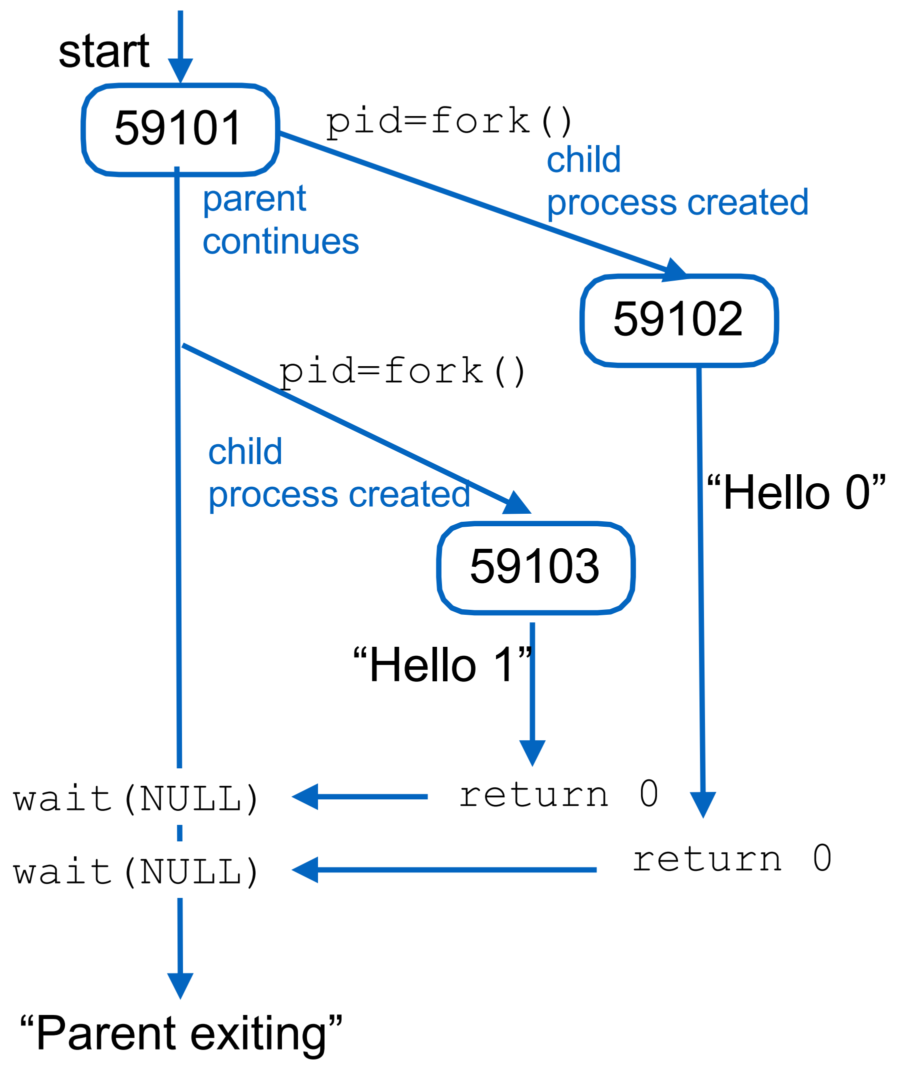
Wait() II

```
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.

Quiz: Fork

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
  pid_t pid = fork();
  if(pid<0){
      perror("fork fail");
      exit(1);
  }
  printf("Hello world!, process_id(pid) =
%d\n", getpid());
  return 0;
}
```

```
Output: parent before child
Hello world!, process_id(pid) = 32
Hello world!, process_id(pid) = 33
or child before parent
Hello world!, process_id(pid) = 33
Hello world!, process_id(pid) = 32
```

- Since we do not check for return value of fork(), both child process and parent process run the same code after fork, and print out its own pid. (The pids 32, 33 shown are just examples.)
- Since parent process and child process run concurrently without wait(), two output interleavings are possible.
- In the following examples, we omit the check for p<0 and assume fork() calls are always successful.

https://www.geeksforgeeks.org/fork-system-call/

Quiz: Fork

```
a = 5;
if (pid=fork()==0) {
    a = a + 5;
    printf("In child, a=%d, a memory
address=%d\n", a, &a);
}
else {
    a = a - 5;
    printf("In parent, a=%d, a memory
address=%d\n", a, &a);
}
```

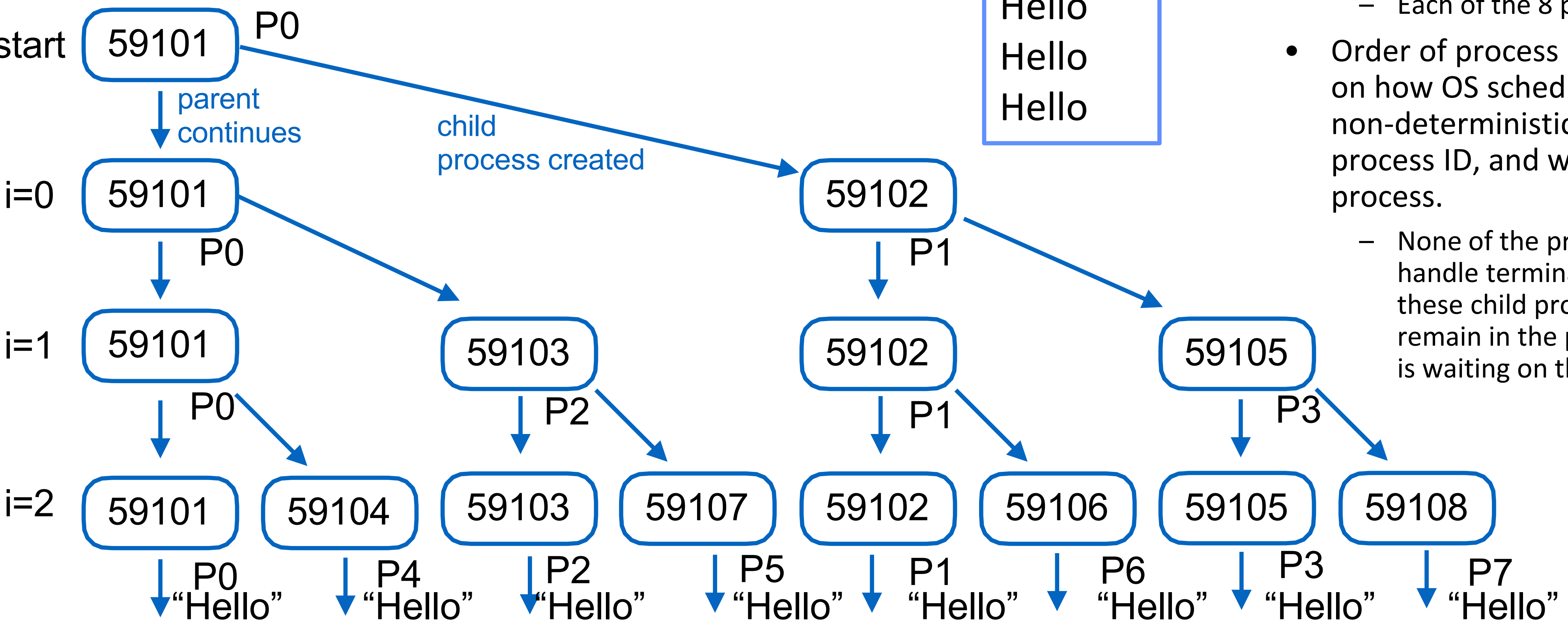
```
Output:
In parent, a = 0, a memory address=0x1234
In child, a=10, a memory address=0x1234
Or,
In child, a=10, a memory address=0x1234
In parent, a = 0, a memory address=0x1234
```

- In Child (x), a = a + 5 = 10; In Parent (u), a = a – 5 = 0.
- The physical addresses of ‘a’ in parent and child must be different. But our program accesses virtual addresses (assuming we are running on an OS that uses virtual memory). The child process gets an exact copy of parent process and virtual address of ‘a’ doesn’t change in child process. Therefore, we get same addresses in both parent and child. (0x1234 is just an example address.)

Quiz: Fork

```
int main() {
  int i;

  for (i = 0; i < 3; i++)
  {fork();}
  printf("Hello\n"); //outside for loop
  return 0;
}
```

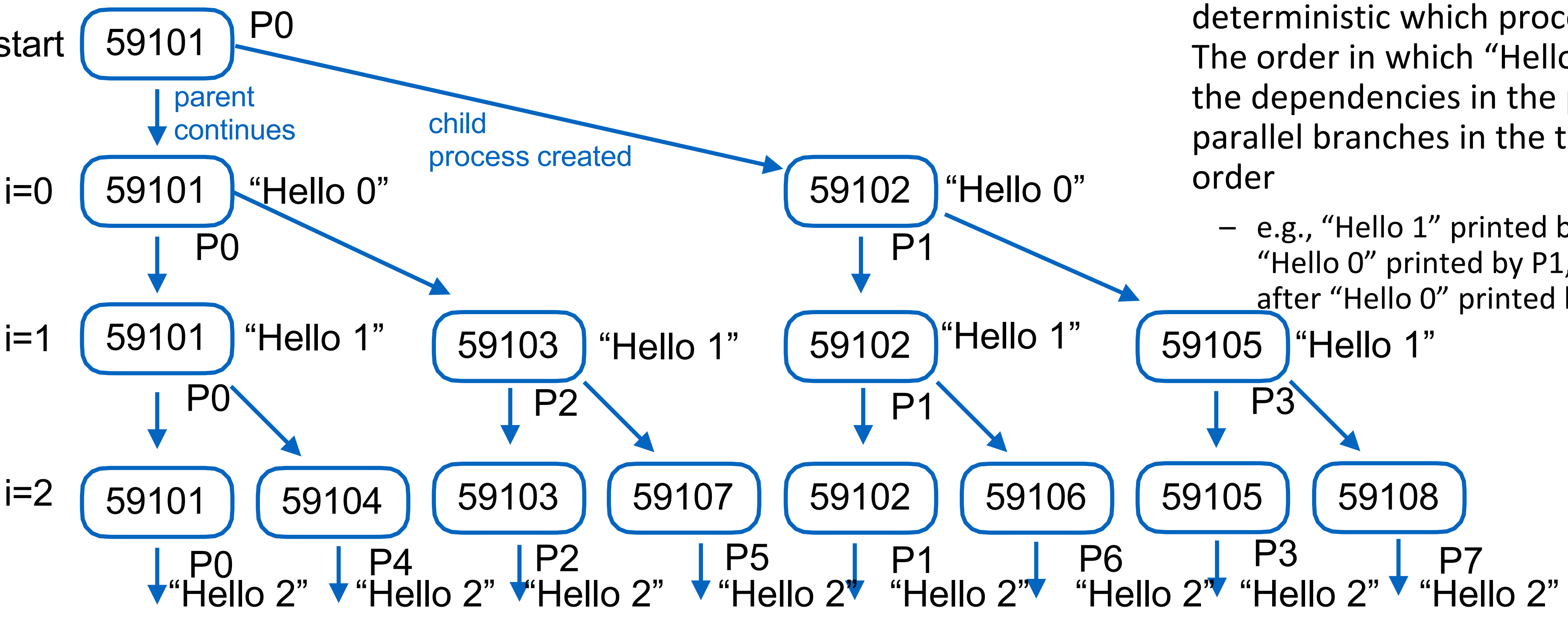
```
Output:
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```



- In general, "for (i = 0; i < n; i++) fork();" creates 1+2+…+2^(n-1)=(2^n)-1 child processes. Plus the main process P0, we have a total of 2^n processes, hence "Hello" is printed 2^n times. Here n = 3, 2^3 = 8.
  - Main process: P0
  - P0 creates 1 child process by the 1st fork: P1
  - P0, P1 create 2 child processes by the 2nd fork: P2, P3
  - P0, P1, P2, P3 create 4 child processes by the 3rd fork: P4, P5, P6, P7
  - Each of the 8 processes P0 to P7 prints a "Hello".
- Order of process execution may vary depending on how OS schedules these processes, so it is non-deterministic which process gets which process ID, and which Hello is printed by which process.
  - None of the processes include a wait() call to handle terminated child processes. When any of these child processes terminate, their PCBs remain in the process table as no parent process is waiting on them, resulting in zombie processes.

Quiz: Fork

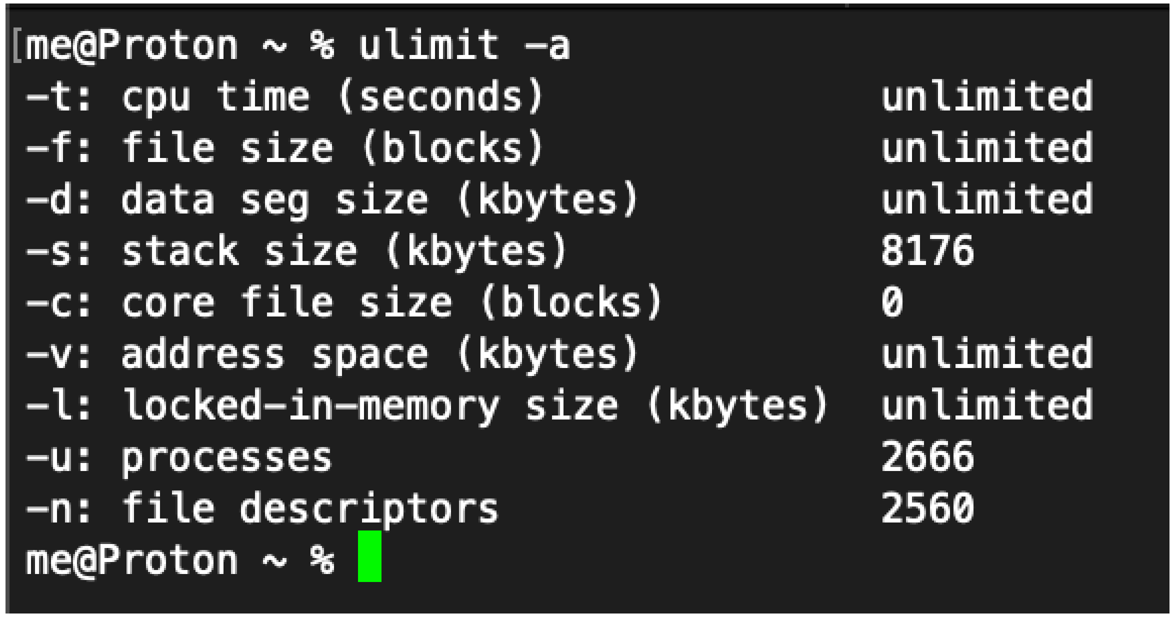
```
int main() {
  int i;

  for (i = 0; i < 3; i++)
  {fork();
  printf("Hello i\n"); } //inside for loop
  return 0;
}
```



- This program will print 14 lines.
  - Main process: P0
  - P0 creates 1 child process by the 1st fork: P1. Then P0 and P1 each prints "Hello 0"
  - P0, P1 create 2 child processes by the 2nd fork: P2, P3. Then P0, P1, P2, P3 each prints "Hello 1"
  - P0, P1, P2, P3 create 4 child processes by the 3rd fork: P4, P5, P6, P7. Then P0 to P7 each prints "Hello 2"
- Order of process execution may vary depending on how OS schedules these processes, so it is non-deterministic which process gets which process ID. The order in which "Hello i" is printed will respect the dependencies in the process creation tree, but parallel branches in the tree may execute in any order
  - e.g., "Hello 1" printed by P1 or P3 must appear after "Hello 0" printed by P1, but it may appear before or after "Hello 0" printed by P0

Quiz: Fork

```
int main() {
  While(true) fork();
  return 0;
}
```

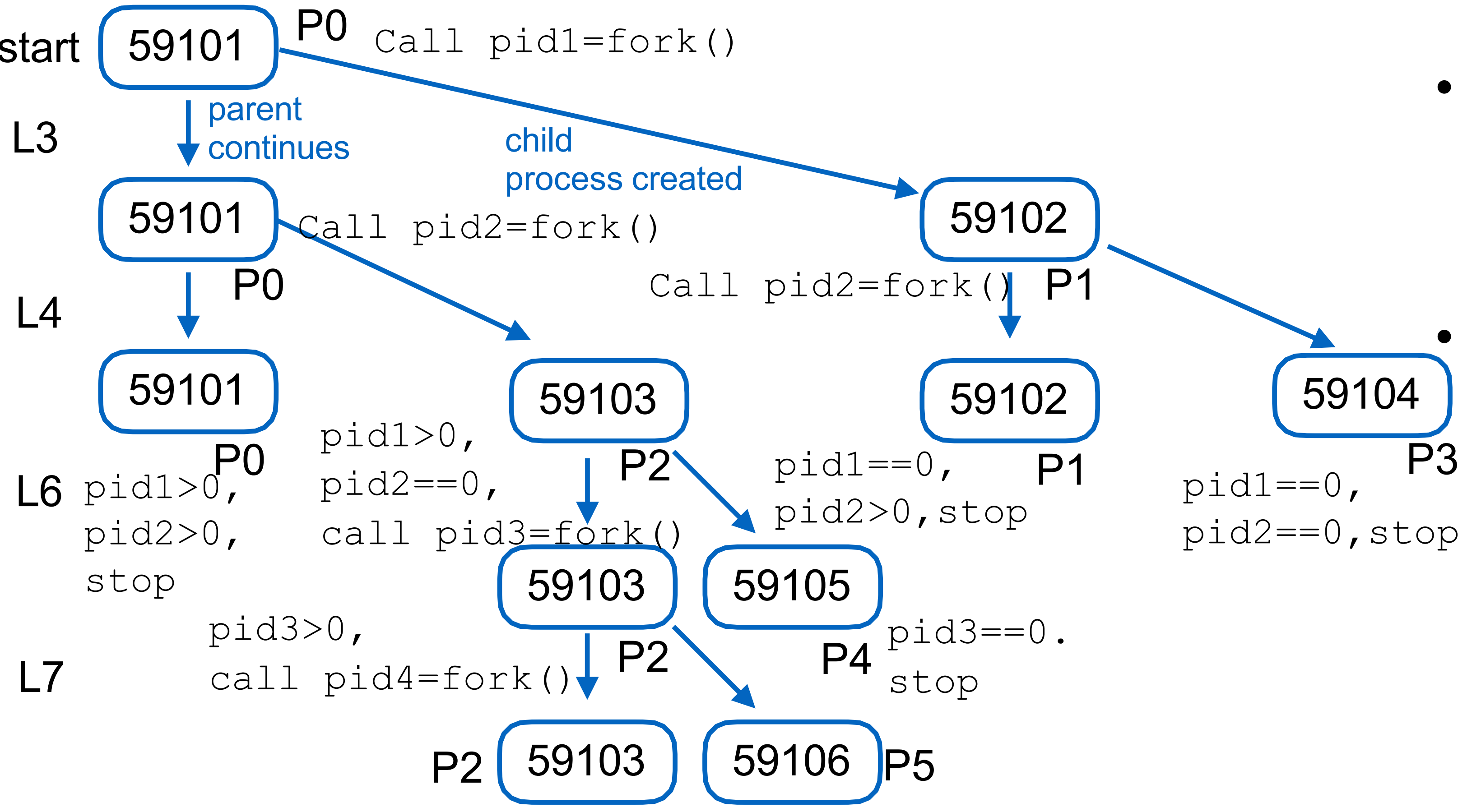


```
[me@Proton ~ % ulimit -a
 -t: cpu time (seconds)              unlimited
 -f: file size (blocks)              unlimited
 -d: data seg size (kbytes)          unlimited
 -s: stack size (kbytes)             8176
 -c: core file size (blocks)         0
 -v: address space (kbytes)          unlimited
 -l: locked-in-memory size (kbytes)  unlimited
 -u: processes                       2666
 -n: file descriptors                2560
 me@Proton ~ %
```

- A fork bomb is a type of denial-of-service (DoS) attack designed to exhaust system resources by creating an exponential number of processes. This is achieved through self-replicating code that repeatedly calls the fork() system call. The result is resource starvation, which can slow down or crash the system.
- Prevention countermeasures:
  - Limit User Processes: Use ulimit in Linux to restrict the number of processes a user can create:
    - » ulimit -u 30 # Limits user to 30 processes
  - Configure /etc/security/limits.conf for persistent limits:
    - » username hard nproc 30

OPTIONAL

```
1 #include <unistd.h>
2 int main (void)  {
3   pid_t pid1 = fork();
4   pid_t pid2 = fork() ;
5   if (pid1>0 && pid2==0){
6    if (pid3=fork()>0){
7      pid4=fork();}
8   } // end if
9   return 0;
10} // end main
```

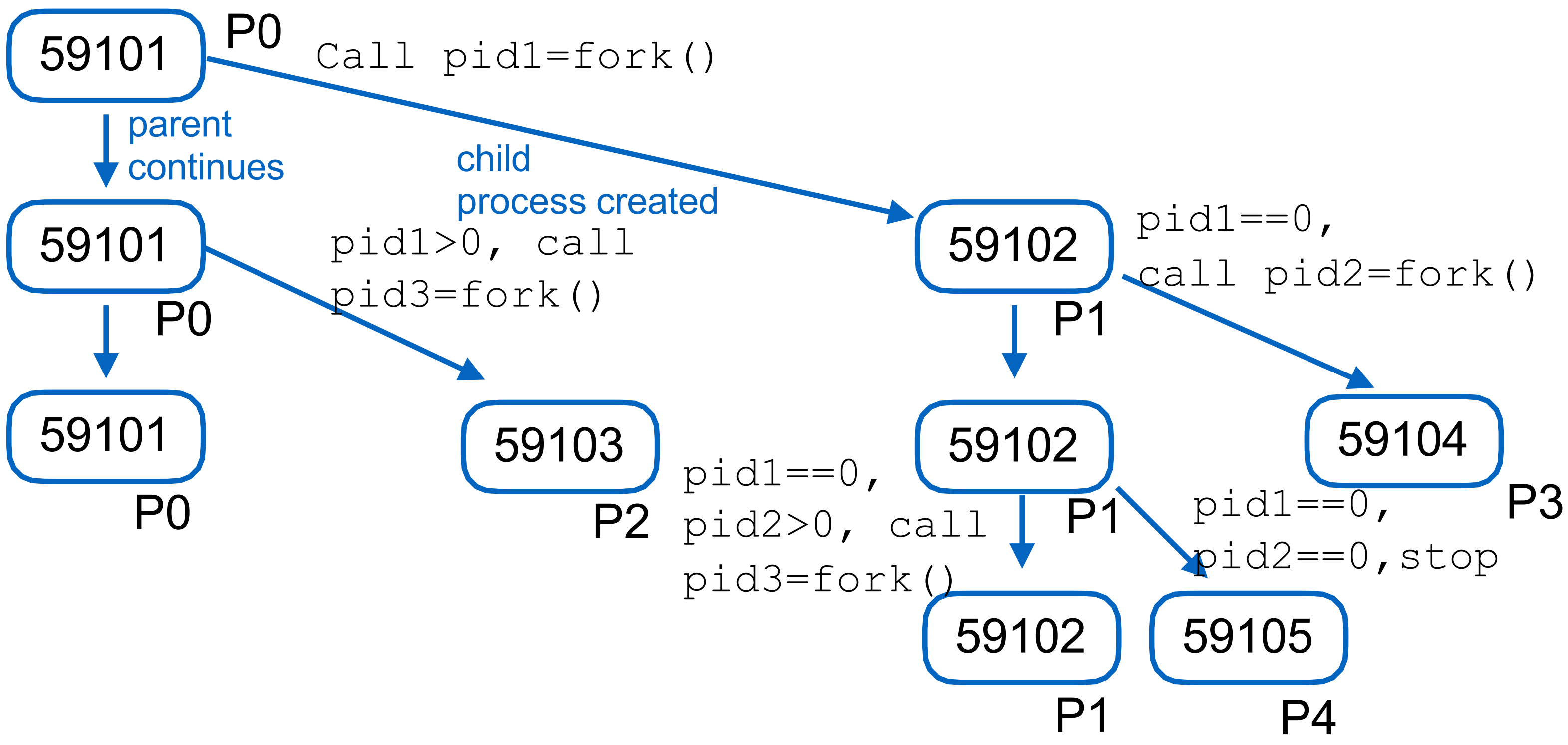


- Q: How many processes are generated in total?
- A: There are 6 processes in total.
- The initial process P0 calls pid1=fork() to generate one child process P1. P0 and P1 each calls pid2=fork() to generate child processes P2 and P3.
- The if condition (pid1 > 0 && pid2 == 0) is checked in all four processes P0 to P3, and it is true only in P2 created by the pid2=fork() in P0, so P2 calls pid3=fork() to generate child process P4.
- The if condition (pid3> 0) is checked in both P2 and P4. It is true in P2, so P2 calls pid4=fork() to generate child process P5. It is false in P4, so P4 stops here and does not call any more fork().

OPTIONAL

Quiz: Fork

```c
#include <unistd.h>
int main (void) {
if(pid1=fork()>0||pid2=fork()>0)
 {pid3=fork();}
 return 0:
}
```



- Q: How many processes are generated in total? In C, the logical OR operator (||) employs short-circuit evaluation, meaning it evaluates expressions from left to right and stops as soon as the result of the entire expression is determined. Specifically for (cond1||cond2): If cond1 evaluates to true (non-zero), the overall result of the || operation is already known to be true, so cond2 is not evaluated. If cond1 evaluates to false (zero), the evaluation proceeds to the next operand cond2.
- A: There are 5 processes in total.
- The initial process P0 calls pid1=fork() to create child process P1. In P0, the if condition (pid1>0||?) = (true&&?)=true, so P0 skips the call pid2=fork() and calls pid3=fork() to create child process P2.
- P1 has pid1==0, so it calls pid2=fork() and creates child process P3. The if condition (pid1>0||pid2>0) is checked in both P1 and P3. In P1, it is (false||true)=true, so P1 calls pid3=fork() to create child process P4. In P3, it is (false||false)=false, so it stops here and does not call any more fork().