The Process API (ch. 5)

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Process API

- API: Application Programming Interface
- The API of the OS: **system calls**
  - Function call into OS code
  - Higher privilege level, for sensitive operations (e.g., hardware)

Process API

- API: Application Programming Interface
- The API of the OS: **system calls**
  - Function call into OS code
  - Higher privilege level, for sensitive operations (e.g., hardware)

- Rewrite code for each OS?
  - **POSIX API**: standard set for each POSIX-compliant OS (`write`)

POSIX hides OS specific details

fork xv6-x86

```
1 movl    $1, %eax
2 int     $64
```

fork Linux-x86

```
1 movl    $2, %eax
2 int     $128
```

close xv6-x86

```
1     pushl   fd
2     subl    $4,%esp
3     movl    $21,%eax
4     int     $64
5     addl    $4,%esp
```

close Linux-x86

```
1     movl    fd,%ebx
2     movl    $6,%eax
3     int     $128
```

Posix Process API

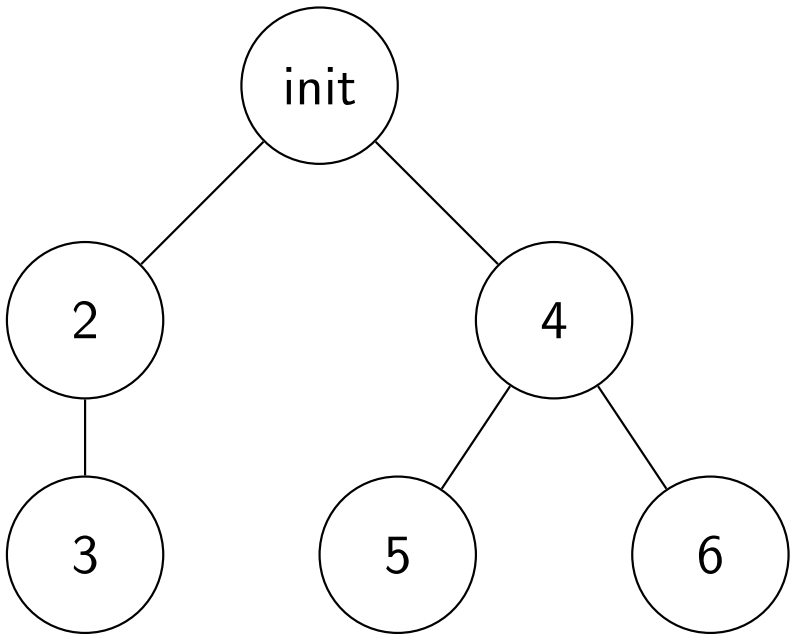
- `fork()`: create a new process
- `wait()`: block until a child process terminates
- `exec()`: make the process execute a given program

Process Tree

- Start with one process: `init` (PID 1)
- A process can create processes
  - Process A creates B: A is the **parent** of B, B is the **child** of A
  - Can create many children, only one parent
  - Parent can **wait** for child process to finish
- Process ID (**PID**): increasing identifier
  - Get PID: `getpid()`
  - Get parent PID: `getppid()`

Process Tree

- Processes form a tree:



- `ps --forest -eaf`
- `pstree`

`fork()`

- `fork()`: creates a new process
  - Wrapper for `clone` (in Linux)
- New process: almost exact copy of parent
  - Same: memory, execution point, open files
  - Different: PID, return value
  - **Copy-on-write** (Optimization)

`fork()`

- `fork()`: creates a new process
  - Wrapper for `clone` (in Linux)
- New process: almost exact copy of parent
  - Same: memory, execution point, open files
  - Different: PID, return value
  - **Copy-on-write** (Optimization)

- Parent: `fork()` returns an integer:
  - If successful returns the **PID** of created child process
  - If fails negative number for erro code

`fork()`

- `fork()`: creates a new process
  - Wrapper for `clone` (in Linux)
- New process: almost exact copy of parent
  - Same: memory, execution point, open files
  - Different: PID, return value
  - **Copy-on-write** (Optimization)

- Parent: `fork()` returns an integer:
  - If successful returns the **PID** of created child process
  - If fails negative number for erro code

- Child process:
  - Begins to run at the point after the fork.
  - 'return value' is zero.

`fork` in details

```
1  pid = fork();
```

```
1      movl    $1,%eax
2      int     $64
3      movl    %eax,pid
```

Parent	Child
1 `movl $1,%eax` 2 `int $64` 3 `movl %eax,pid`	3 `movl %eax,pid`

fork()

Typical usage example (`fork.c`):

```
1  printf("hello world (pid:%d)\n", getpid());
2  int rc = fork();
3  if (rc < 0) {
4      fprintf(stderr, "fork failed\n");
5      exit(1);
6  }
7  else if (rc == 0) {
8      // child (new process)
9      // sleep(5);   // Try with and without
10     printf("I am child of %d (pid:%d)\n", getppid(), getpid());
11 }
12 else {
13     // parent
14     printf("I am parent of %d (pid:%d)\n", rc, getpid());
15 }
```

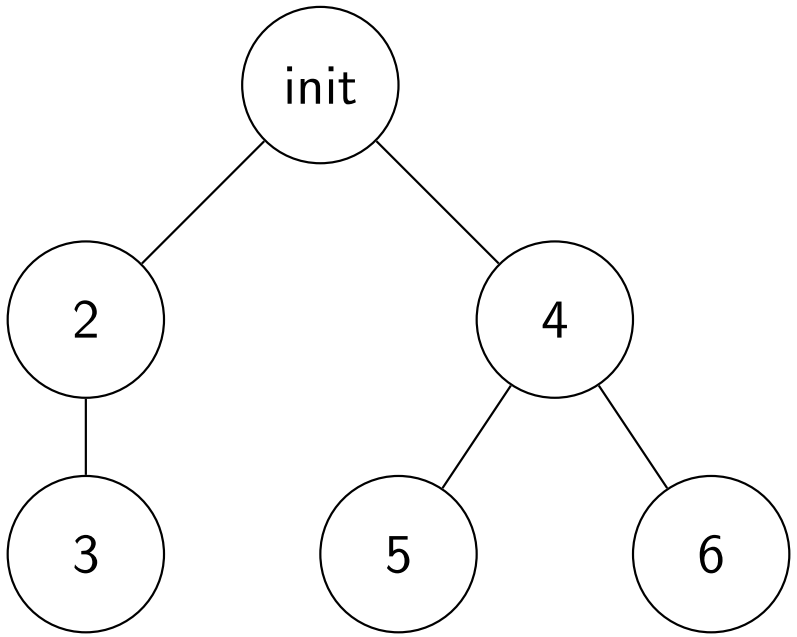
fork()

Output:

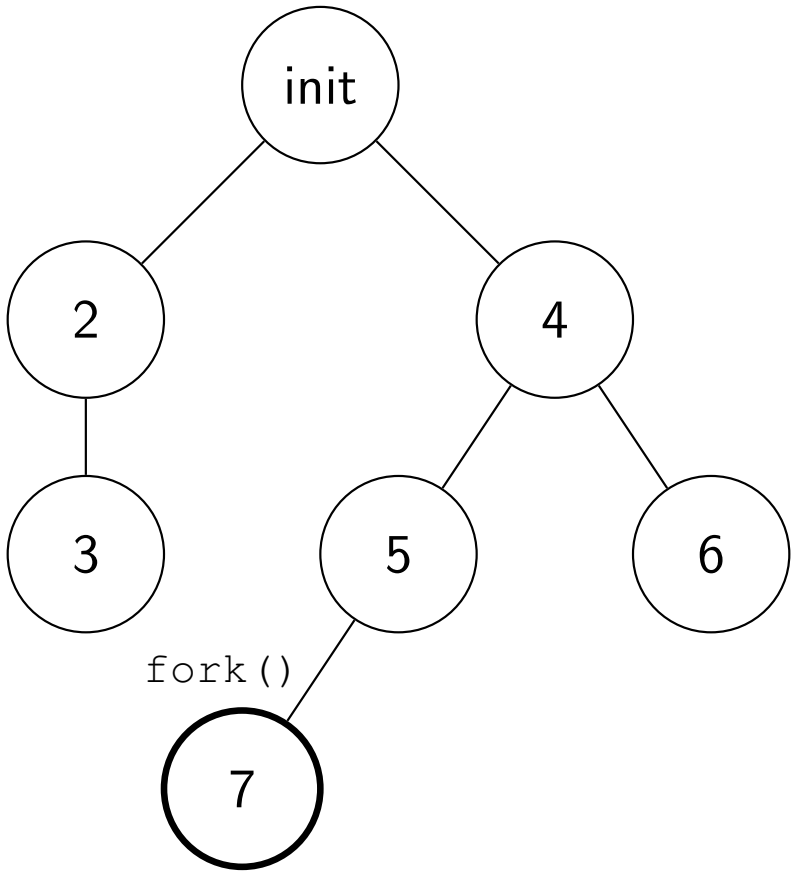
```
prompt> gcc -o fork fork.c -Wall
prompt> ./fork
hello world (pid:1300)
I am parent of 1301 (pid:1300)
I am child of 1 (pid:1301)
prompt>
```

- Child of **1**??

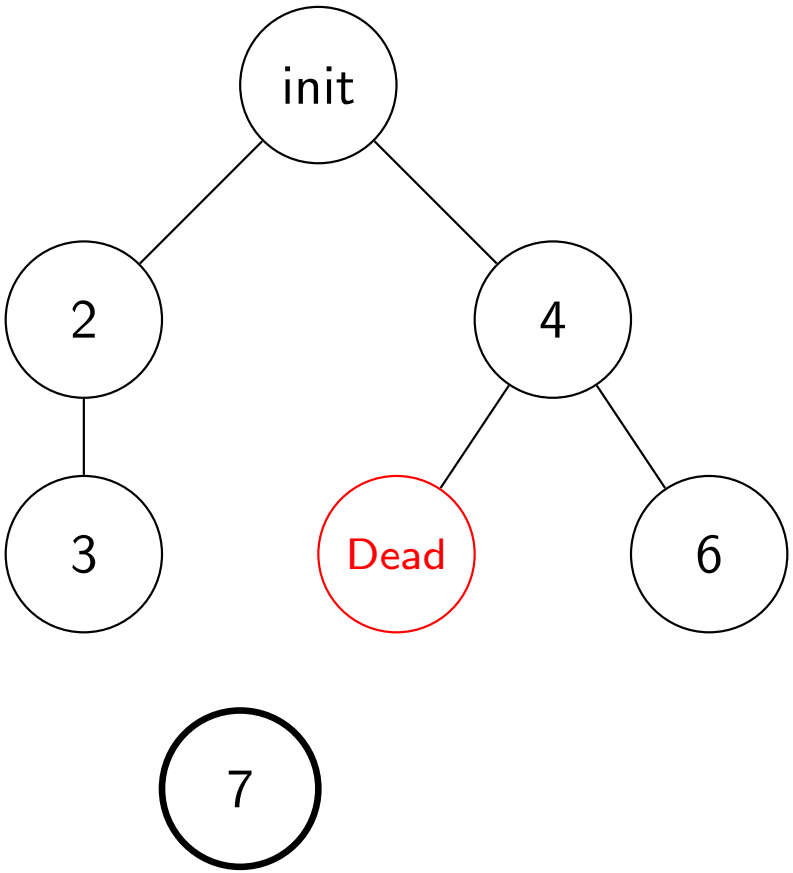
fork()



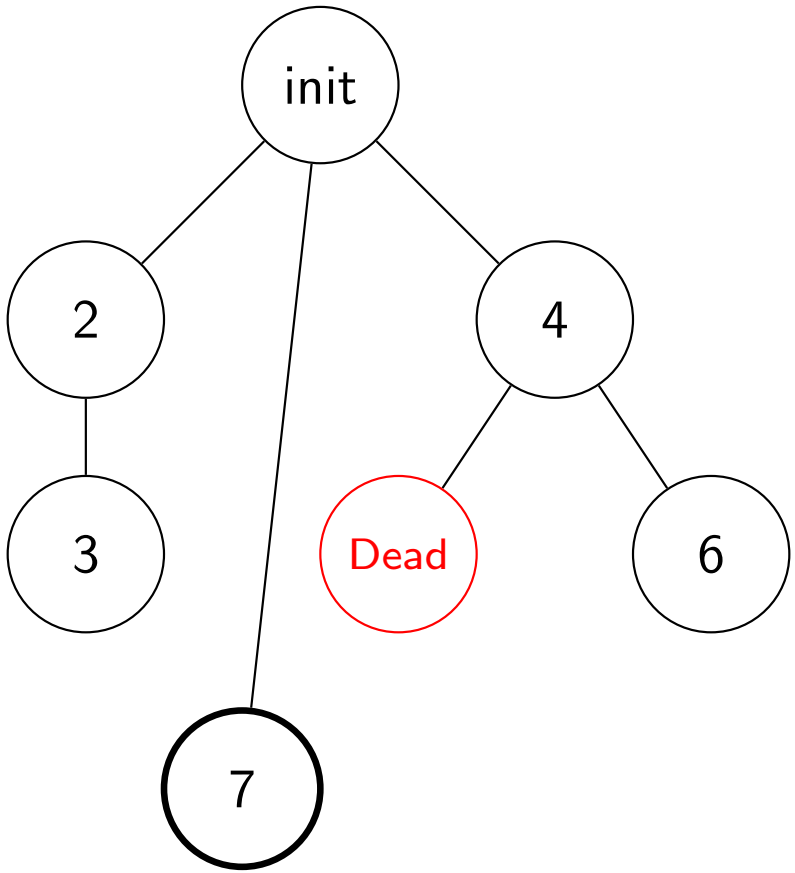
fork()



fork()



fork()



fork()

peculiar1.c:

```
1 int main(int argc, char *argv[])
2 {
3     fork();
4     fork();
5     printf("hello there\n");
6 }
```

What is the output?

fork()

peculiar1.c:

```
1 int main(int argc, char *argv[])
2 {
3     fork();
4     fork();
5     printf("hello there\n");
6 }
```

What is the output?

```
1 hello there
2 hello there
3 hello there
4 hello there
```

peculiar1

- P0 runs

peculiar1

- P0 runs
  - create P1 which begins on line 4

peculiar1

- P0 runs
  - create P1 which begins on line 4
  - create P2 which begins on line 5

peculiar1

- P0 runs
  - create P1 which begins on line 4
  - create P2 which begins on line 5
  - prints.

peculiar1

- P0 runs
  - create P1 which begins on line 4
  - create P2 which begins on line 5
  - prints.
- P1 runs

peculiar1

- P0 runs
  - create P1 which begins on line 4
  - create P2 which begins on line 5
  - prints.
- P1 runs
  - create P3 which begins on line 5

peculiar1

- P0 runs
  - create P1 which begins on line 4
  - create P2 which begins on line 5
  - prints.
- P1 runs
  - create P3 which begins on line 5
  - prints

peculiar1

- P0 runs
  - create P1 which begins on line 4
  - create P2 which begins on line 5
  - prints.
- P1 runs
  - create P3 which begins on line 5
  - prints
- P2 runs

peculiar1

- P0 runs
  - create P1 which begins on line 4
  - create P2 which begins on line 5
  - prints.
- P1 runs
  - create P3 which begins on line 5
  - prints
- P2 runs
  - prints

peculiar1

- P0 runs
  - create P1 which begins on line 4
  - create P2 which begins on line 5
  - prints.
- P1 runs
  - create P3 which begins on line 5
  - prints
- P2 runs
  - prints
- P3 runs

peculiar1

- P0 runs
  - create P1 which begins on line 4
  - create P2 which begins on line 5
  - prints.
- P1 runs
  - create P3 which begins on line 5
  - prints
- P2 runs
  - prints
- P3 runs
  - prints

fork()

peculiar2.c:

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

What is the output?

fork()

peculiar2.c:

```
1  int main(int argc, char *argv[])
2  {
3      int pid = fork();
4      if (pid)
5          fork();
6      fork();
7      printf("hello there\n");
8  }
```

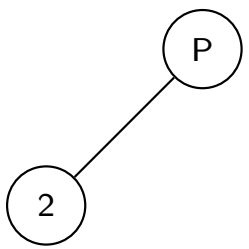
What is the output?

fork()

peculiar2.c:

```
1  int main(int argc, char *argv[])
2  {
3      int pid = fork();
4      if (pid)
5          fork();
6      fork();
7      printf("hello there\n");
8  }
```

What is the output?

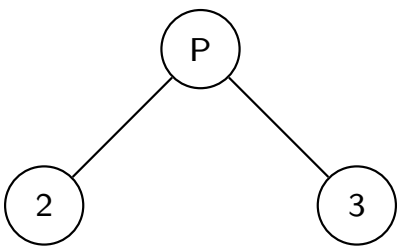


fork()

peculiar2.c:

```
1  int main(int argc, char *argv[])
2  {
3      int pid = fork();
4      if (pid)
5          fork();
6      fork();
7      printf("hello there\n");
8  }
```

What is the output?

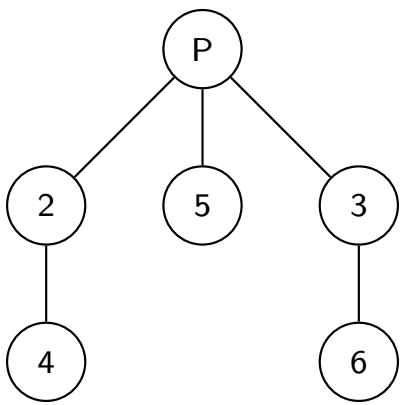


fork()

peculiar2.c:

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

What is the output?



peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P0 runs.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P0 runs.

- P1 L4.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P0 runs.

- P1 L4.
- P2 L6.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P0 runs.

- P1 L4.
- P2 L6.
- P3 L7

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P0 runs.

- P1 L4.
- P2 L6.
- P3 L7
- Prints.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P1 L4 runs.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P1 L4 runs.

- P4 L7.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P1 L4 runs.

- P4 L7.
- prints.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P2 L6 runs.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P2 L6 runs.

- P5 L7.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P2 L6 runs.

- P5 L7.
- prints.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P3 L7 runs.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P3 L7 runs.

- prints.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P4 L7 runs

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P4 L7 runs

- prints.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P5 L7 runs.

peculiar2

```
1 int main(int argc, char *argv[])
2 {
3     int pid = fork();
4     if (pid)
5         fork();
6     fork();
7     printf("hello there\n");
8 }
```

P5 L7 runs.

- prints.

fork()

peculiar3.c:

```
1 int main(int argc, char *argv[])
2 {
3     fork();
4     printf("hello\n");
5 }
```

Can this print "hehellollo"?

fork()

peculiar3.c:

```
1 int main(int argc, char *argv[])
2 {
3     fork();
4     printf("hello\n");
5 }
```

Can this print "hehellollo"?

- This is kernel implementaion dependent!
- Very important to consider these cases
- More on this in the future (**concurrency**)

fork()

peculiar4.c:

```
1  int main(int argc, char *argv[])
2  {
3      int x = 0;
4      if (fork()) {
5          sleep(5); // BLOCKED state for 5 seconds
6          printf("%d\n", x);
7      }
8      else {
9          x += 3;
10     }
11 }
```

What is the output?

fork()

peculiar4.c:

```
1  int main(int argc, char *argv[])
2  {
3      int x = 0;
4      if (fork()) {
5          sleep(5); // BLOCKED state for 5 seconds
6          printf("%d\n", x);
7      }
8      else {
9          x += 3;
10     }
11 }
```

What is the output? **0**

- Why?

fork()

peculiar4.c:

```
1  int main(int argc, char *argv[])
2  {
3      int x = 0;
4      if (fork()) {
5          sleep(5); // BLOCKED state for 5 seconds
6          printf("%d\n", x);
7      }
8      else {
9          x += 3;
10     }
11 }
```

What is the output? **0**

- Why? Child's memory is a **copy**

fork()

peculiar5.c:

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

How many processes does this code create?

fork()

peculiar5.c:

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

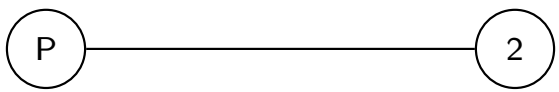
How many processes does this code create?

fork()

peculiar5.c:

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

How many processes does this code create?

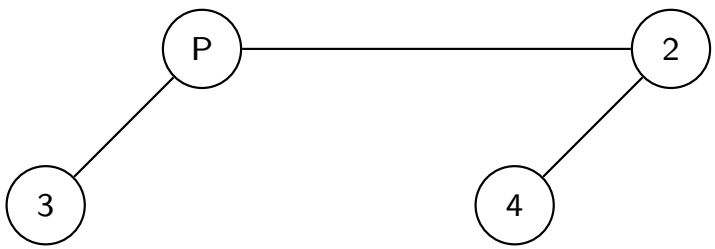


fork()

peculiar5.c:

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

How many processes does this code create?

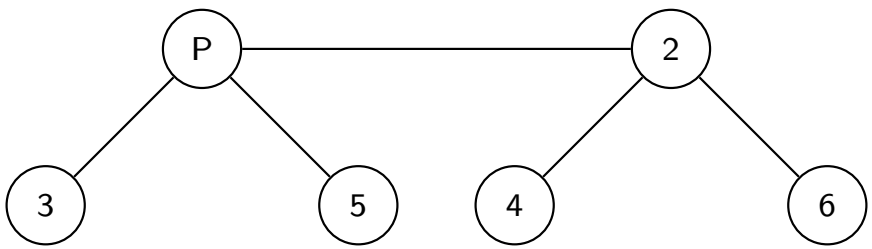


fork()

peculiar5.c:

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

How many processes does this code create?

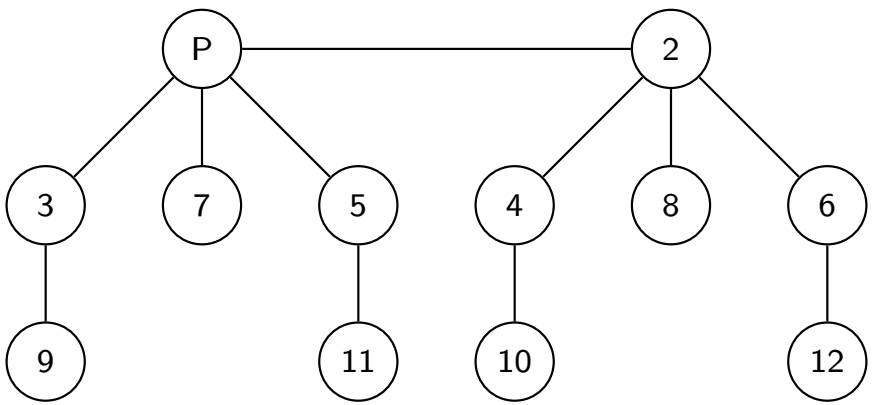


fork()

peculiar5.c:

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

How many processes does this code create?



peculiar5

P0

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P0

- P1 L2

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P0

- P1 L2
- P2 L2.5

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P0

- P1 L2
- P2 L2.5
- P3 L4

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P0

- P1 L2
- P2 L2.5
- P3 L4
- P4 L6

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P1 L2 runs

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P1 L2 runs

- P5 L2.5

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P1 L2 runs

- P5 L2.5
- P6 L4

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P1 L2 runs

- P5 L2.5
- P6 L4
- P7 L6

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P2 L2.5 runs

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P2 L2.5 runs

- P8 L6

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P3 L4 runs

- P9 L6

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P4 L6

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P5 L2.5 runs

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P5 L2.5 runs

- P10 L6

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P6 L4 runs

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P6 L4 runs

- P11 L6

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P7 L6 runs

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P8 L6 runs

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P9 L6 runs

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P10 L6 runs

peculiar5

```
1 fork();
2 if (fork()) {
3     fork();
4 }
5 fork();
```

P11 L6 runs

fork()

peculiar6.c:

```
1  int main(int argc, char *argv[])
2  {
3      int x = 0;
4      if (fork()) {
5          sleep(5); // Play with sleep
6      }
7      else {
8          x += 3;
9      }
10     printf(''%d'', x);
11 }
```

Last one - what is the output?

fork()

peculiar6.c:

```
1  int main(int argc, char *argv[])
2  {
3      int x = 0;
4      if (fork()) {
5          sleep(5); // Play with sleep
6      }
7      else {
8          x += 3;
9      }
10     printf(''%d'', x);
11 }
```

Last one - what is the output? **30** or **03**

fork()

peculiar6.c:

```
1  int main(int argc, char *argv[])
2  {
3      int x = 0;
4      if (fork()) {
5          sleep(5); // Play with sleep
6      }
7      else {
8          x += 3;
9      }
10     printf(''%d'', x);
11 }
```

Last one - what is the output? **30** or **03**

- Most chances 30

fork()

peculiar6.c:

```
1  int main(int argc, char *argv[])
2  {
3      int x = 0;
4      if (fork()) {
5          sleep(5); // Play with sleep
6      }
7      else {
8          x += 3;
9      }
10     printf(''%d'', x);
11 }
```

Last one - what is the output? **30** or **03**

- Most chances 30
- Depends on scheduling

fork()

peculiar6.c:

```
1  int main(int argc, char *argv[])
2  {
3      int x = 0;
4      if (fork()) {
5          sleep(5); // Play with sleep
6      }
7      else {
8          x += 3;
9      }
10     printf(''%d'', x);
11 }
```

Last one - what is the output? **30** or **03**

- Most chances 30
- Depends on scheduling
- Can we make it deterministic?

`wait()`

- `wait(*status)`: waits for a child process to finish
  - Any child process (if several exist)
  - Returns PID of terminated child process (negative if no child)
  - `waitpid(pid, ...)`: waits for a specific child process (by PID)
- To wait for all child processes to end:
  - `while (wait(NULL) != -1);`
- (It really is waiting for child state change)

wait()

wait.c:

```
1  int main(int argc, char *argv[])
2  {
3      int x = 0;
4      int rc = fork();
5      if (rc) {
6          wait(NULL); // BLOCKED until child terminates
7          // equivalent here: waitpid(rc, NULL, 0);
8      }
9      else {
10         x += 3;
11     }
12     printf("%d", x);
13 }
```

Output is always **30**

exec()

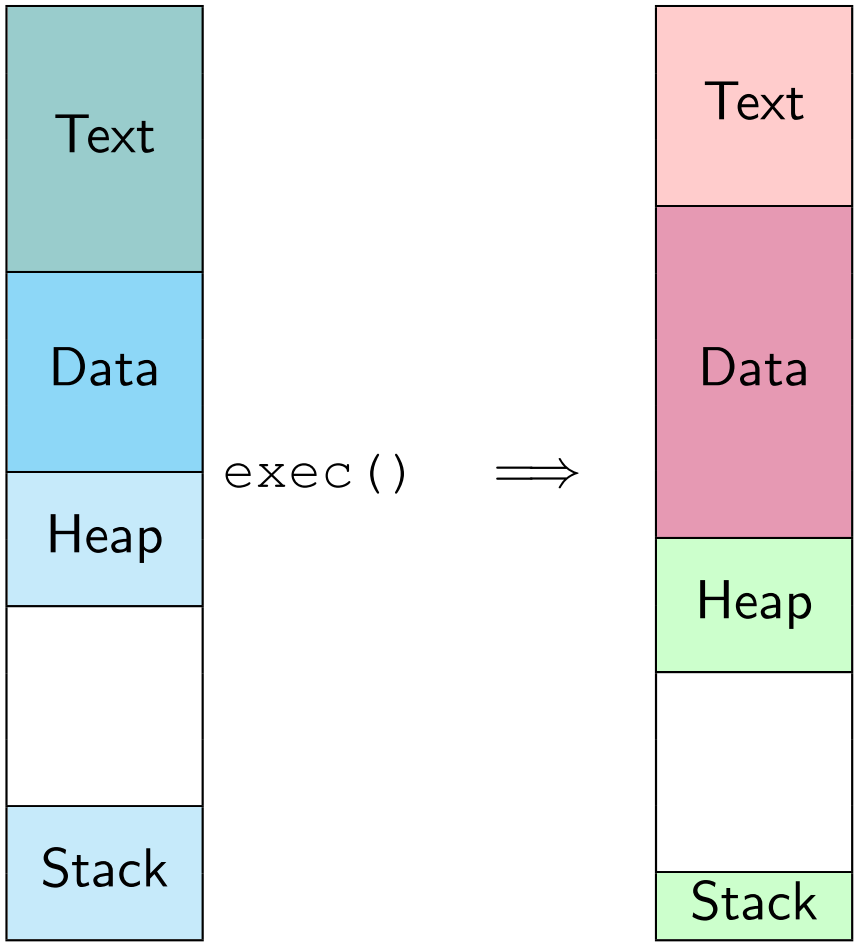
- After `fork()`, parent and child execute same code
  - What if we want to run a different program?
  - `exec()` does just that
- Six variants of `exec()`: `execl`, `execlp`, `execle`, `execv`, `execvp`, `execvpe`. Read `man` for details

exec()

- After `fork()`, parent and child execute same code
  - What if we want to run a different program?
  - `exec()` does just that
- Six variants of `exec()`: `execl`, `execlp`, `execle`, `execv`, `execvp`, `execvpe`. Read `man` for details

- `exec()`: Replaces current program with a different program
  - Receives program name and arguments (`argv`)
  - Overwrites and re-initializes process memory
  - A successful `exec()` never returns!

exec()



exec()

exec.c:

```
1  int main(int argc, char *argv[])
2  {
3      int rc = fork();
4      if (rc < 0) {
5          fprintf(stderr, "fork failed\n");
6          exit(1);
7      }
8      else if (rc == 0) {
9          char* args[4] = { "wc", "-l", "exec.c", NULL };
10         execvp(args[0], args);
11         printf("this shouldn't print out\n");
12     }
13     else {
14         int rc_wait = wait(NULL); // or waitpid(rc,NULL,0)
15         printf("I am parent of %d (rc_wait:%d) (pid:%d)\n",
16             rc, rc_wait, getpid());
17     }
18 }
```

The Living Dead

- When a process terminates, it remains in the process list as a **zombie**
  - Parent process may want to know its status
- Zombie remains until it is reaped (i.e., waited upon)
  - Process 1 adpots orphans (zombied or live)

- A program should not leave zombies!

The Living Dead

- How to avoid zombies?
  - `wait()`: blocks until a child completes & reaps it
  - `waitpid()`: blocks until a specific child completes & reaps it
- Not enough
  - The terminal (shell) executes processes in the background, wants to continue accepting user input
  - It is possible to `wait()` without blocking, but very inconvenient
- What can we do?

Signals

- **Software interrupts**
  - Asynchronous notification of an event
  - Inter-process communication (**IPC**) or messages from OS

Signals

- **Software interrupts**
  - Asynchronous notification of an event
  - Inter-process communication (**IPC**) or messages from OS

- Various signals exist:
  - ^C in the terminal sends `SIGINT` ("interrupt from keyboard")
  - Invalid memory reference causes `SIGSEGV`
  - A process can send `SIGKILL` to another process
  - Child process terminated - **SIGCHLD**

Signal Handlers

- Some signals are handled automatically by the OS
  - `SIGKILL`, `SIGSTOP`
- Others are handled by a **signal handler**
  - Each signal has a default behavior, e.g., `SIGINT` causes the process to terminate
  - Can override default with `sigaction()`
- Let's write our own **signal handler**!

Signal Handlers

signal1.c:

```
1  int main(int argc, char *argv[])
2  {
3      struct sigaction act;
4      sigemptyset(&act.sa_mask);
5      act.sa_handler = SIG_IGN;
6      act.sa_flags = 0;
7
8      if (sigaction(SIGINT, &act, NULL) == -1) {
9          fprintf(stderr, "sigaction failed\n");
10         exit(1);
11     }
12     while (1);
13 }
```

Signal Handlers

signal2.c:

```
1  void signal_handler(int signal) {
2      if (signal == SIGCHLD) {
3          int rc = wait(NULL);
4          printf("child terminated %d (pid:%d)\n", rc, getpid());
5      }
6  }
7  int main(int argc, char *argv[])
8  {
9      struct sigaction act;
10     sigemptyset (&act.sa_mask);
11     act.sa_handler = signal_handler;
12     act.sa_flags = 0;
13
14     sigaction(SIGCHLD, &act, NULL);
15     if (fork()) {
16         while (1);
17     }
18 }
```

No zombies!

`kill()`

- `kill()`: send a signal to another process
  - `kill(pid_t` *pid*`,` `int` *sig*`)`
  - *pid*: process id to send signal to
  - *sig*: signal to send
- Name is misleading
  - Can send any signal

Case Study

- How does a shell work?
  - Reads user command
  - Forks a child
  - Sets up process (e.g., redirection)
  - Execs the relevant program
  - Waits for it to finish (if not background)
  - Reads next command

Summary (Process API)

- `fork()`: create a new process (clone current)
- `wait()`: waits for a child process to finish
  - Also `waitpid()`
- `exec()`: transform program into a different program
  - Successful `exec()` never returns
- Terminated process remains as a **zombie**, to avoid:
  - Parent terminates
  - `wait()` or `waitpid()` by parent
- **Signals** are software interrupts
  - Can write our own **signal handlers**
  - Also helps with zombies
- `kill()`: send a signal to another process