Processes (ch. 4+5)

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

Moshe Sulamy

Tel-Aviv Academic College

The Process

A **running program**

- Lots of processes seemingly running at the same time
- The challenge:
    - Few physical CPUs, illusion of many CPUs

The Process

- **Virtualizing** the CPU
  - Running one process, stopping it, running another, and so forth
  - **Time sharing** of the CPU
  - Illusion that many virtual CPUs exist

The Process

- **Virtualizing** the CPU
  - Running one process, stopping it, running another, and so forth
  - **Time sharing** of the CPU
  - Illusion that many virtual CPUs exist
- **Context switch**
  - Low-level mechanism
  - Stop running one program and start running another
- **Scheduling policy**
  - Algorithm to decide which process should run next
  - By history, workload, performance

Context switch example (xv6-riscv64)

```
swtch:
        sd ra, 0(a0)
        sd sp, 8(a0)
        sd s0, 16(a0)
        sd s1, 24(a0)
        sd s2, 32(a0)
        sd s3, 40(a0)
        sd s4, 48(a0)
        sd s5, 56(a0)
        sd s6, 64(a0)
        sd s7, 72(a0)
        sd s8, 80(a0)
        sd s9, 88(a0)
        sd s10, 96(a0)
        sd s11, 104(a0)
```

```
        ld ra, 0(a1)
        ld sp, 8(a1)
        ld s0, 16(a1)
        ld s1, 24(a1)
        ld s2, 32(a1)
        ld s3, 40(a1)
        ld s4, 48(a1)
        ld s5, 56(a1)
        ld s6, 64(a1)
        ld s7, 72(a1)
        ld s8, 80(a1)
        ld s9, 88(a1)
        ld s10, 96(a1)
        ld s11, 104(a1)

        ret
```

Time and Space Sharing

- **Time sharing**
  - Resource used for a little while by one entity, then a little while by another, and so forth
  - e.g., CPU
- **Space sharing**
  - Resource is divided (in space) among those who wish to use it
  - e.g., memory, disk

Process vs. Program

- **Program**: static code and static data
- **Process**: dynamic instance of the program

- Multiple processes of the same program can exist

What constitutes a process?

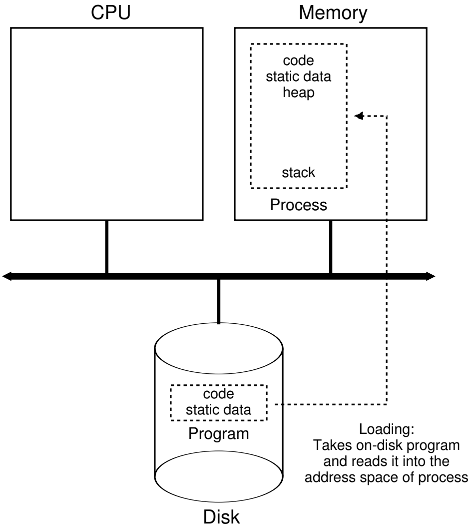
- Memory (**address space**)
  - Instructions (program code)
  - Data (static and dynamic)
  - `cat /proc/<PID>/maps`
- Registers
  - Program counter (PC)
  - Stack pointer
  - etc.
- I/O information
  - e.g., open files
  - `cat /proc/<PID>/fdinfo/*`

Process Creation

- Unix likes OSes: A process is a replica of a currently existing process.
    - There is a way to load an executable file into an existing process.
- Non-Unix like OSes: A process is created with information from an exe file.

Either way, the first process is created by the OS on initialization.

Executable Loading



Executable Loading

- **Load** code and static data into memory
  - Program initially on disk
  - Loading can be done **lazily** (via **paging** and **swapping**)

Executable Loading

- **Load** code and static data into memory
  - Program initially on disk
  - Loading can be done **lazily** (via **paging** and **swapping**)
- Allocate the **stack**
  - Used for local variables, function parameters, return addresses
  - Initialized with `main` arguments: `argc`, `argv`

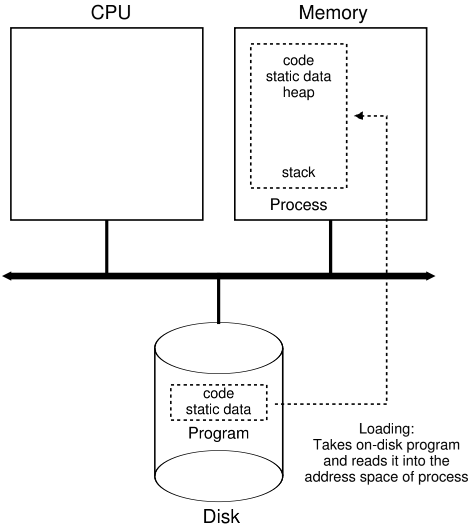
Executable Loading

- **Load** code and static data into memory
  - Program initially on disk
  - Loading can be done **lazily** (via **paging** and **swapping**)
- Allocate the **stack**
  - Used for local variables, function parameters, return addresses
  - Initialized with `main` arguments: `argc`, `argv`
- Allocate the **heap**
  - Used for dynamically-allocated data
  - Request space by calling `malloc`, free it by `free`

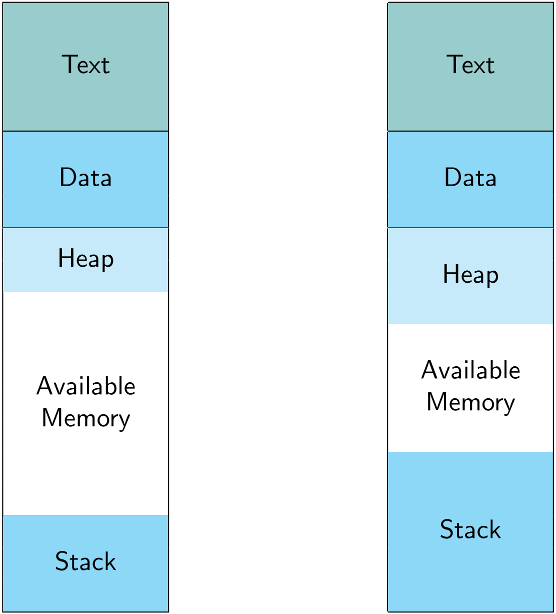
Executable Loading

- **Load** code and static data into memory
  - Program initially on disk
  - Loading can be done **lazily** (via **paging** and **swapping**)
- Allocate the **stack**
  - Used for local variables, function parameters, return addresses
  - Initialized with `main` arguments: `argc`, `argv`
- Allocate the **heap**
  - Used for dynamically-allocated data
  - Request space by calling `malloc`, free it by `free`
- Start program at entry point (NOT necessarily `main()`)
  - Transfer control of CPU to the newly-created process

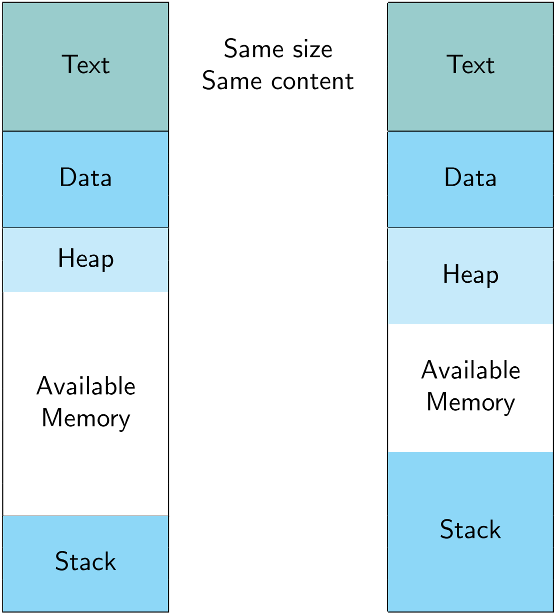
Executable loading



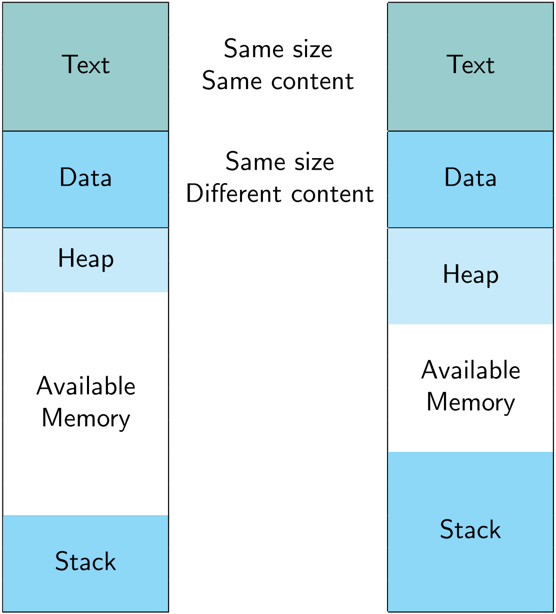
Two Processes for One Program (Virt Mem View)



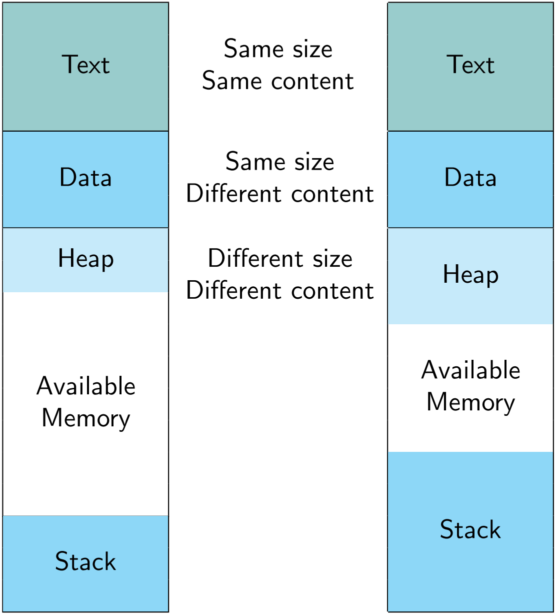
Two Processes for One Program (Virt Mem View)



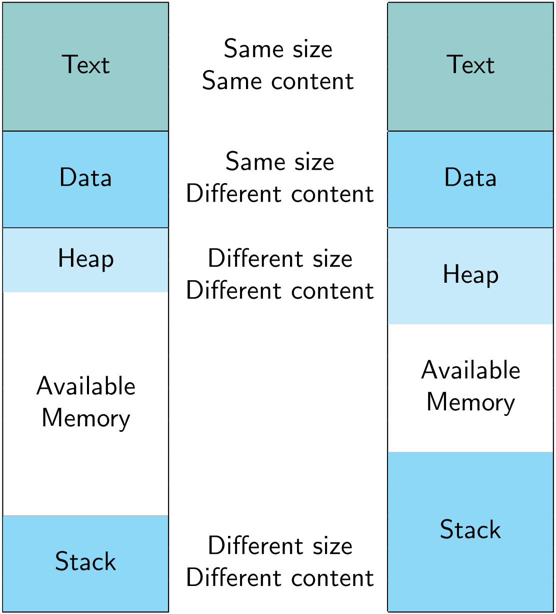
Two Processes for One Program (Virt Mem View)



Two Processes for One Program (Virt Mem View)



Two Processes for One Program (Virt Mem View)



Process Life Cycle

- A process can be in a finite number of **states**
- Events cause **transitions** between states

Process Life Cycle

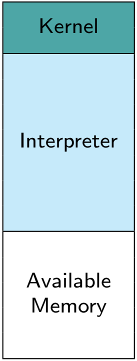
- A process can be in a finite number of **states**
- Events cause **transitions** between states

- **Single-tasking** OS (Ancient, Small phys. memory)
  - Only one process at a time
  - **Interpreter** loaded on boot, overwrites part of itself into process

Process Life Cycle

- A process can be in a finite number of **states**
- Events cause **transitions** between states

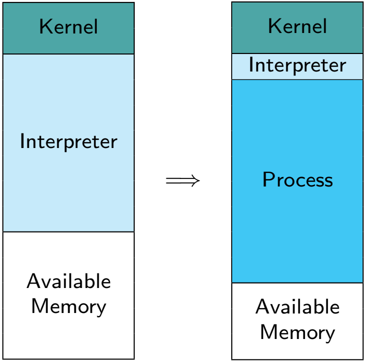
- **Single-tasking** OS (Ancient, Small phys. memory)
  - Only one process at a time
  - **Interpreter** loaded on boot, overwrites part of itself into process



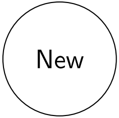
Process Life Cycle

- A process can be in a finite number of **states**
- Events cause **transitions** between states

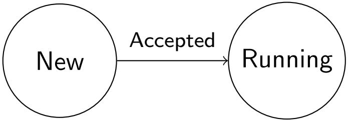
- **Single-tasking** OS (Ancient, Small phys. memory)
  - Only one process at a time
  - **Interpreter** loaded on boot, overwrites part of itself into process



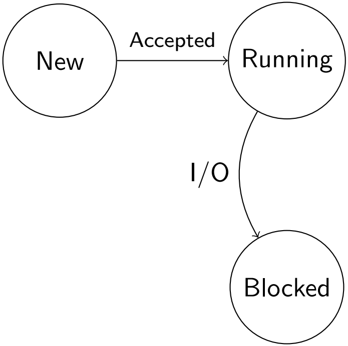
Process Life Cycle - Single Tasking



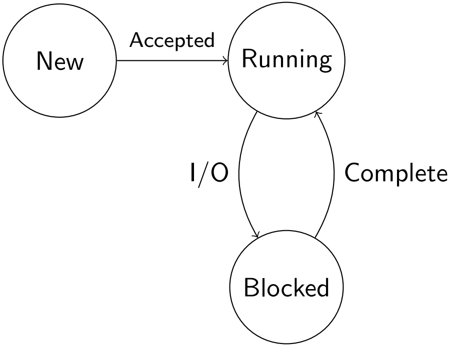
Process Life Cycle - Single Tasking



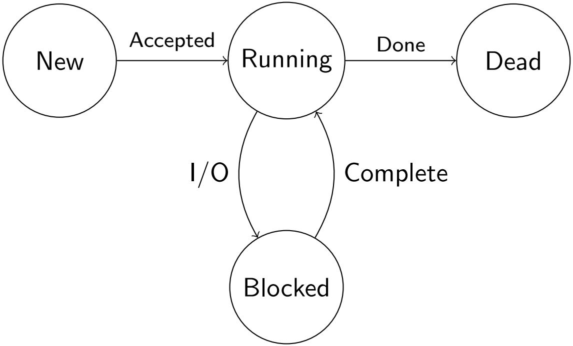
Process Life Cycle - Single Tasking



Process Life Cycle - Single Tasking



Process Life Cycle - Single Tasking



Process Life Cycle

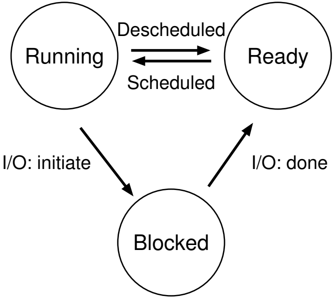
- Modern operating systems: **multi-tasking**
  - Multiple processes co-exist
  - **Cooperative** multi-tasking: `yield`
  - **Preemptive** multi-tasking: **interrupts**

Process Life Cycle

- Modern operating systems: **multi-tasking**
  - Multiple processes co-exist
  - **Cooperative** multi-tasking: `yield`
  - **Preemptive** multi-tasking: **interrupts**
- A process can be **ready** to run, but not running
  - OS schedules a process to run for a while, then deschedules it and picks another process, and so forth
  - A new state: **ready**

Process States

- **Running**: executing on CPU
- **Ready**: ready to run, waiting to be scheduled
- **Blocked**: suspended, waiting for some event



Process States - Example I

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process 0 done
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	Process 1 done

Process States - Example II

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	0 initiates I/O
4	Blocked	Running	0 is blocked
5	Blocked	Running	so 1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process 1 done
9	Running	-	
10	Running	-	Process 0 done

Data Structures

- OS maintains a data structure of active processes
  - The **process table**
  - Limited size - `cat /proc/sys/kernel/threads-max`

Data Structures

- OS maintains a data structure of active processes
  - The **process table**
  - Limited size - `cat /proc/sys/kernel/threads-max`
- Process Control Block (**PCB**):
  - Process identifier (**PID**)
  - State
  - Related processes (parent)
  - CPU context, e.g., registers (saved when suspended)
  - Memory locations
  - Open files

Summary (Process Abstraction)

- **Process**: OS abstraction of a running program
- Can be described by:
  - **Address space**
  - CPU registers (inc. **program counter** & **stack pointer**)
  - I/O information (e.g., open files)
- **Process state**: running, ready to run, blocked.
  - transition by different events
- **Process list**: information about all processes in the system
  - **Process control block**: a structure with information about a specific process

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 details

fork xv6-x86

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

fork Linux-x86

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

close xv6-x86

```
    pushl    fd
    subl     $4,%esp
    movl     $21,%eax
    int      $64
    addl     $4,%esp
```

close Linux-x86

```
    movl     fd,%ebx
    movl     $6,%eax
    int      $127
```

Posix Process API

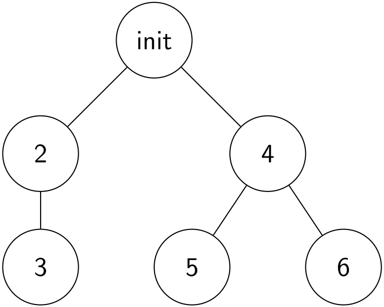
- `fork()`: create a new process
- `wait()`: block until a child process terminates
- `exec()`: make the process execute a given program

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

```c
pid = fork();
```

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

<table>
<tr><th>Parent</th><th>Child</th></tr>
<tr><td><pre>movl $1,%eax
int $64</pre></td><td></td></tr>
<tr><td><pre>movl   %eax,pid</pre></td><td><pre>movl   %eax,pid</pre></td></tr>
</table>

fork()

Typical usage example (fork.c):

```
printf("hello world (pid:%d)\n", getpid());
int rc = fork();
if (rc < 0) {
    fprintf(stderr, "fork failed\n");
    exit(1);
}
else if (rc == 0) {
    // child (new process)
    // sleep(5);   // Try with and without
    printf("I am child of %d (pid:%d)\n", getppid(), getpid());
}
else {
    // parent
    printf("I am parent of %d (pid:%d)\n", rc, getpid());
}
```

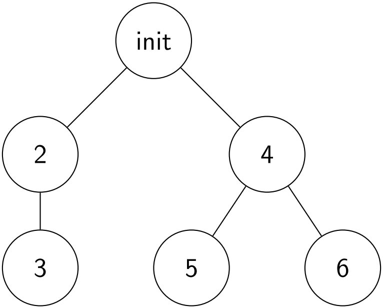
fork()

Output:

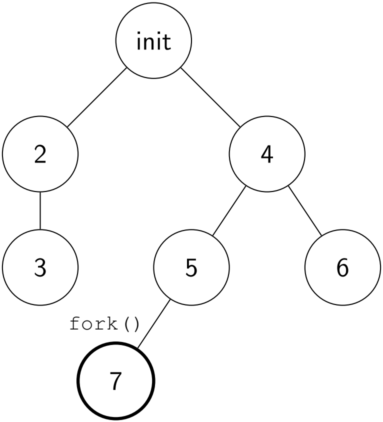
```
prompt> gcc -o fork fork.c -Wall
prompt> ./fork
hello world (pid:1300)
I am parent of 1301 (pid:1300)
I am child of 1 (pid:1301)
prompt>
```

- Child of **1**??

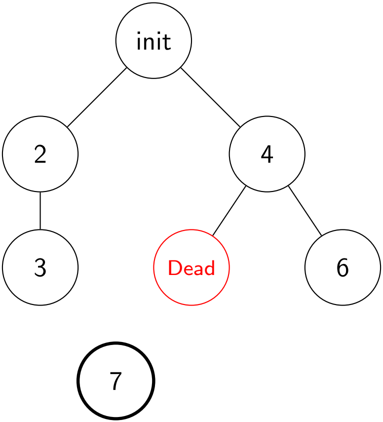
fork()



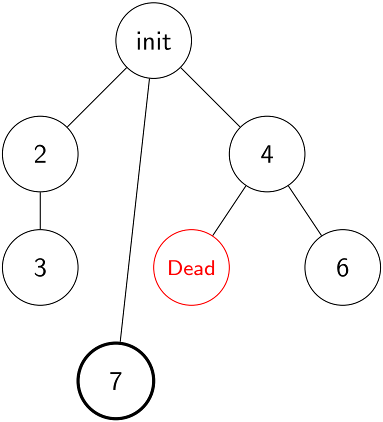
fork()



fork()



fork()



fork()

peculiar1.c:

```c
int main(int argc, char *argv[])
{
    fork();
    fork();
    printf("hello there\n");
}
```

What is the output?

fork()

peculiar1.c:

```c
int main(int argc, char *argv[])
{
    fork();
    fork();
    printf("hello there\n");
}
```

What is the output?

```
hello there
hello there
hello there
hello there
```

fork()

peculiar2.c:

```c
int main(int argc, char *argv[])
{
    int pid = fork();
    if (pid)
        fork();
    fork();
    printf("hello there\n");
}
```

What is the output?

fork()

peculiar2.c:

```c
int main(int argc, char *argv[])
{
    int pid = fork();
    if (pid)
        fork();
    fork();
    printf("hello there\n");
}
```

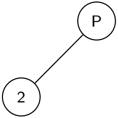
What is the output?

fork()

peculiar2.c:

```c
int main(int argc, char *argv[])
{
    int pid = fork();
    if (pid)
        fork();
    fork();
    printf("hello there\n");
}
```

What is the output?

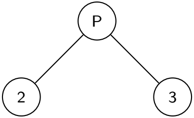


fork()

peculiar2.c:

```c
int main(int argc, char *argv[])
{
    int pid = fork();
    if (pid)
        fork();
    fork();
    printf("hello there\n");
}
```

What is the output?

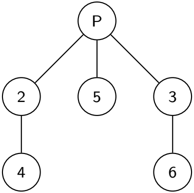


fork()

peculiar2.c:

```c
int main(int argc, char *argv[])
{
    int pid = fork();
    if (pid)
        fork();
    fork();
    printf("hello there\n");
}
```

What is the output?



fork()

peculiar3.c:

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

Can this print “hehellollo”?

fork()

peculiar3.c:

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

Can this print "hehellollo"?

- No! Due to how `printf` works
- But... very important to consider these cases
- More on this in the future (**concurrency**)

fork()

peculiar4.c:

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

What is the output?

fork()

peculiar4.c:

```c
int main(int argc, char *argv[])
{
    int x = 0;
    if (fork()) {
        sleep(5); // BLOCKED state for 5 seconds
        printf("%d\n", x);
    }
    else {
        x += 3;
    }
}
```

What is the output? **0**

- Why?

fork()

peculiar4.c:

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

What is the output? **0**

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

fork()

peculiar5.c:

```
fork();
if (fork()) {
    fork();
}
fork();
```

How many processes does this code create?

fork()

peculiar5.c:

```c
fork();
if (fork()) {
    fork();
}
fork();
```

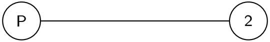
How many processes does this code create?

fork()

peculiar5.c:

```c
fork();
if (fork()) {
    fork();
}
fork();
```

How many processes does this code create?

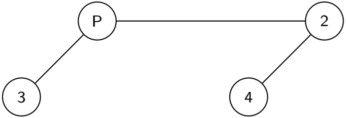


fork()

`peculiar5.c`:

```
fork();
if (fork()) {
    fork();
}
fork();
```

How many processes does this code create?

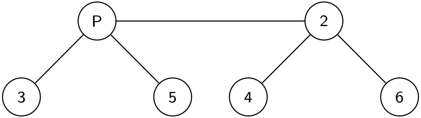


fork()

peculiar5.c:

```c
fork();
if (fork()) {
    fork();
}
fork();
```

How many processes does this code create?

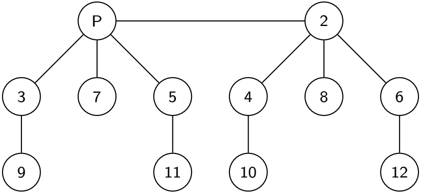


fork()

peculiar5.c:

```
fork();
if (fork()) {
    fork();
}
fork();
```

How many processes does this code create?



fork()

peculiar6.c:

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

Last one - what is the output?

fork()

peculiar6.c:

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

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

- Depends on scheduling

fork()

peculiar6.c:

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

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

- Depends on scheduling
- Can we make it deterministic?

`wait()`

- `wait()`: waits for a child process to finish
  - Any child process (if several exist)
  - Returns PID of terminated child process (-1 on error)
  - `waitpid()`: waits for a specific child process (by PID)
- To wait for all child processes to end:
  - `while (wait(NULL) != -1);`

wait()

wait.c:

```c
int main(int argc, char *argv[])
{
    int x = 0;
    int rc = fork();
    if (rc) {
        wait(NULL); // BLOCKED until child terminates
        // equivalent here: waitpid(rc, NULL, 0);
    }
    else {
        x += 3;
    }
    printf("%d\n", x);
}
```

Output is always **30**

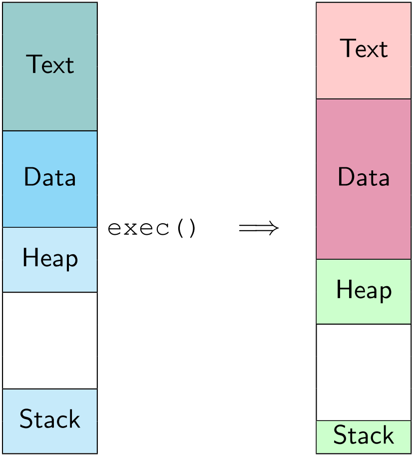
`exec()`

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

exec()



exec()

exec.c:

```
int main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0) {
        char* args[4] = { "wc", "-l", "exec.c", NULL };
        execvp(args[0], args);
        printf("this shouldn't print out\n");
    }
    else {
        int rc_wait = wait(NULL); // or waitpid(rc,NULL,0)
        printf("I am parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, getpid());
    }
}
```

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 (or its parent terminates)
  - 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 <u>specific</u> 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:
  - ˆC 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`:

```
int main(int argc, char *argv[])
{
    struct sigaction act;
    sigemptyset(&act.sa_mask);
    act.sa_handler = SIG_IGN;
    act.sa_flags = 0;

    if (sigaction(SIGINT, &act, NULL) == -1) {
        fprintf(stderr, "sigaction failed\n");
        exit(1);
    }
    while (1);
}
```

Signal Handlers

`signal2.c`:

```
void signal_handler(int signal) {
    if (signal == SIGCHLD) {
        int rc = wait(NULL);
        printf("child terminated %d (pid:%d)\n", rc, getpid());
    }
}
int main(int argc, char *argv[])
{
    struct sigaction act;
    sigemptyset (&act.sa_mask);
    act.sa_handler = signal_handler;
    act.sa_flags = 0;

    sigaction(SIGCHLD, &act, NULL);
    if (fork()) {
        while (1);
    }
}
```

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