Processes

*Program* = code (static)

*Process* = a running program instance (dynamic)
  code + state (contents of registers, memory, other resources)

Key illusions:

**Logical control flow**
  Each process seems to have exclusive use of the CPU

**Private address space**
  Each process seems to have exclusive use of full memory
Starting new processes: fork-exec model

fork() create a copy of the current process
execve() replace the current process’ code and context (registers, memory) with that of a different program.

fork() and execve() are system calls

Other system calls for process management:
  getpid()
  exit()
  wait() / waitpid()
  ...

Process management in Windows differs from this.
fork

pid_t fork()

1. **Clone** current *parent* process to create identical* child process, including all state (memory, registers, **program counter**, ...).

2. Continue executing both copies with **one difference:**
   - returns 0 to the **child** process
   - returns child’s process ID (**pid**) to the **parent** process

```
    pid_t pid = fork();
    if (pid == 0) {
        printf("hello from child\n");
    } else {
        printf("hello from parent\n");
    }
```

**fork** is unique: called *in one process*, returns *in two processes!*

(once in parent, once in child)

*almost. See **man 3 fork** for exceptions.*
Creating a new process with `fork`

### Process n

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

### Child Process m

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

### Which prints first?

- `hello from parent`
- `hello from child`
fork again

- Parent and child continue from *private copies* of same state.
  - Memory contents (*code*, *globals*, *heap*, *stack*, etc.),
    Register contents, *program counter*, file descriptors...

- Only difference: return value from `fork()`

- Relative execution order of parent/child after `fork()` undefined

```c
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```
fork-exec

fork()  clone current process
execv() replace process code and context (registers, memory) with a fresh program.

See man 3 execv, man 2 execve

// Example arguments: path="/usr/bin/ls",

void fork_exec(char* path, char* argv[]) {
    pid_t pid = fork();
    if (pid != 0) {
        printf("Parent: created a child %d\n", pid);
    } else {
        printf("Child: exec-ing new program now\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
**Executing a new program**

When you run the command `ls` in a shell:

1. Code/state of shell process.
2. Copy of code/state of shell process.
3. Replaced by code/state of `ls`.

---

**fork()**: parent

```
fork()
```

**child**

```
exec()
```

**child**

```
Stack
Heap
Data
Code: /usr/bin/bash
```

---

**child**

```
Stack
Heap
Data
Code: /usr/bin/ls
```
**execv: load/start program**

```c
int execv(char* filename,
        char* argv[])
```

loads/states program in current process:
- Executable `filename`
- With argument list `argv`

overwrites code, data, and stack
- Keeps pid, open files, a few other items

**does not return**
- unless error

Also sets up `environment`. See also: `execve`. 

```
Stack bottom
```

```
Stack top
```
wait for child processes to terminate

pid_t waitpid(pid_t pid, int* stat, int ops)

Suspend current process (i.e. parent) until child with pid ends.

On success:

  Return pid when child terminates.
  Reap child.

  If stat != NULL, waitpid saves termination reason where it points.

See also: man 3 waitpid
Zombies!

Terminated process still consumes system resources

Reaping with `wait/waitpid`

What if parent doesn’t reap?
- If any parent terminates without reaping a child, then child will be reaped by `init` process (pid == 1)
- What if parent runs a long time? *e.g.*, shells and servers
waitpid example

```c
void fork_wait() {
    int child_status;
    pid_t child_pid = fork();

    if (child_pid == 0) {
        printf("HC: hello from child\n");
    } else {
        if (-1 == waitpid(child_pid, &child_status, 0)) {
            perror("waitpid");
            exit(1);
        }
        printf("CT: child %d has terminated\n", child_pid);
    }
    printf("Bye\n");
    exit(0);
}
```
Error-checking

Check return results of system calls for errors! (No exceptions.)
Read documentation for return values.
Use perror to report error, then exit.

void perror(char* message)
    Print "<message>: <reason that last system call failed.>"
Examining Processes on Linux (demo)

- ps
- pstree
- top
- /proc
Process management summary

fork gets us two copies of the same process (but fork() returns different values to the two processes)

execve has a new process substitute itself for the one that called it

Two-process program:
- First fork()
  - if (pid == 0) { /* child code */ } else { /* parent code */ }

Two different programs:
- First fork()
  - if (pid == 0) { execve() } else { /* parent code */ }
  - Now running two completely different programs

wait / waitpid used to synchronize parent/child execution and to reap child process
Detailed examples
Fork Example #2

- Both parent and child can continue forking

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```
Fork Example #3

- Both parent and child can continue forking

```c
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```
Both parent and child can continue forking

```c
void fork4() {
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```
Fork Example #5

- Both parent and child can continue forking

```c
void fork5() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```
Zombie Example

```c
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

- `ps` shows child process as “defunct”

- Killing parent allows child to be reaped by `init`
Non-terminating Child Example

```c
void fork8() {
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", getpid());
        exit(0);
    }
}
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely
**wait() Example**

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
void fork10() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d
", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```
**waitpid()**: Waiting for a Specific Process

- `waitpid(pid, &status, options)`
  - suspends current process until specific process terminates
  - various options (that we won’t talk about)

```c
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}```