Operating Systems, a 240 view
barely scraping the surface

Key abstractions provided by kernel
- process
- virtual memory

Virtualization mechanisms and hardware support:
- context-switching
- exceptional control flow
- address translation, paging, TLBs

Processes

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

Key illusions:
Logic 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

Why? How?

Context Switching

Kernel (shared OS code) switches between processes
Control flow passes between processes via context switch.

Implementing logical control flow

Abstraction: every process has full control over the CPU

Implementation: time-sharing
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

fork is unique: called in one process, returns in two processes! (once in parent, once in child)

*almost. See man 3 fork for exceptions.

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);
}

Creating a new process with fork

Process n

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

Child Process m

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

3. 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-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");
}

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

hello from parent hello from child

Which prints first? hello from parent
hello from child
**Exec-ing a new program**

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

- **Stack**:
  - Child process
  - Child stack
  - Child code
  - Child heap

- **Heap**:
  - Child heap

- **Data**:
  - Child data

**exec**: load/start program

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

loads/starts program in current process:

- **Executable** `filename`
- **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`.

**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.

- **Return `pid`** when child terminates.
- **Reap child**.
- If `stat` != `NULL`, `waitpid` saves termination reason where it points.

See also: `man 3 waitpid`