Processes

Focus:
Process model
Process management case study: Unix/Linux/Mac OS X
(Windows is a little different.)
Operating Systems

Problem: unwieldy hardware resources

Solution: operating system
Operating Systems, a 240 view

Focus: key abstractions provided by kernel

barely scraping the surface

Abstractions:

process
virtual memory

Virtualization mechanisms and hardware support:

c context-switching
exceptional control flow
address translation, paging, TLBs
Processes

*Program* = code *(static)*

*Process* = a running program instance *(dynamic)*
  
  code + state

Key illusions:

Why are these abstractions important?
How are these abstractions implemented?
Implementing logical control flow

**Abstraction:** every process has full control over the CPU

**Implementation:** time-sharing
Context Switching

*Kernel* (shared OS code) switches between processes

Control flow passes between processes via *context switch*. 

Context = 

![Diagram showing context switching between processes](image-url)
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)
Creating a new process with \texttt{fork}

\textit{Process n}

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

\textit{Child Process m}

\begin{verbatim}
if (pid == 0) {
  printf("hello from child\n");
} else {
  printf("hello from parent\n");
}
\end{verbatim}

1 \rightarrow \textit{m}

2 \rightarrow 0

3 \rightarrow 0

\textbf{Which prints first?}

hello from parent  \hspace{1cm}  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-exec model:

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

See `man 3 execv`, `man 2 execve`

```c
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");
}
```
Exec-ing a new program

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

```
parent

fork():

child

exec():

child
```

Code/state of shell process.

Copy of code/state of shell process.

Replaced by code/state of `ls`.
execv: load/start program

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

loads/starts 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.
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
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    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);
    }

    fork_wait();
    return 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.>"