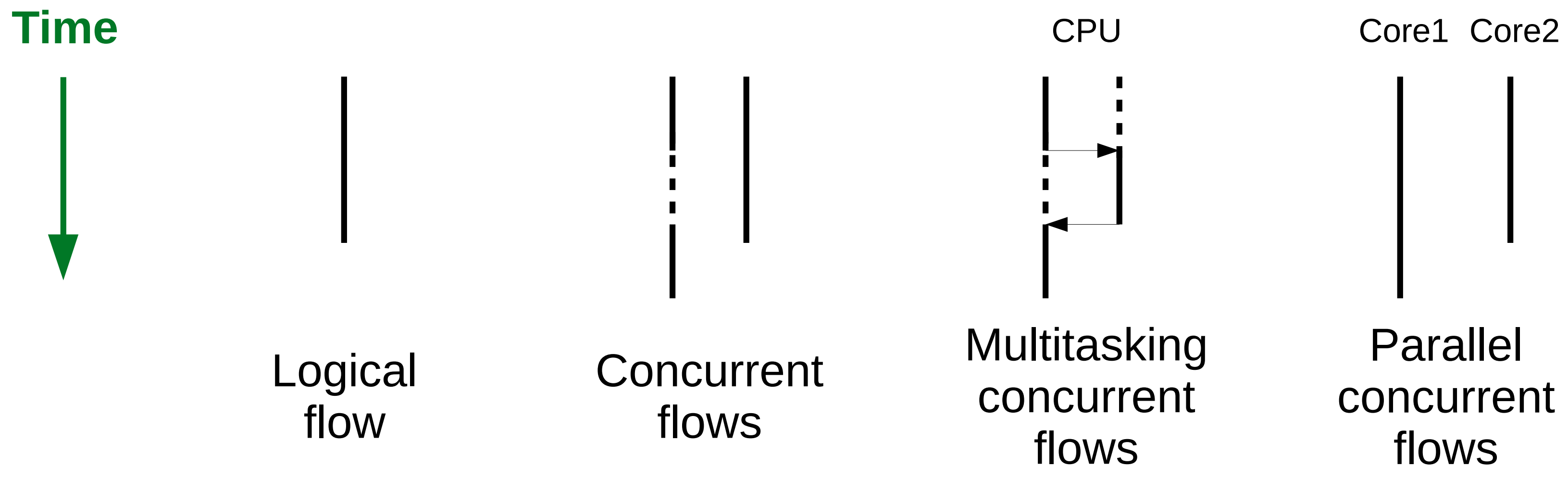
CS 261
Fall 2016

Mike Lam, Professor

Processes

Processes

- Process: instance of an executing program
  - Independent single logical flow and private virtual address space
- Logical flow: sequence of executed instructions
- Concurrency: overlapping logical flows
- Multitasking: processes take turns
- Parallelism: concurrent flows on separate CPUs/cores



Implementing processes

- Processes are abstractions
  - Implemented/provided by the operating system kernel
  - Kernel maintains data structure w/ process information
    - Including an ID for each process (pid)
  - Multitasking via exceptional control flow
    - Periodic interrupt to switch processes
    - Called round-robin switching
  - Context switch: swapping current process
    - Save context of old process
    - Restore context of new process
    - Pass control to the restored process

Linux process tools

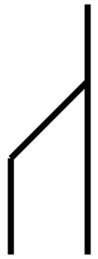
- `ps` – list processes
  - "`ps -fe`" to see all processes on the system
  - "`ps -fu <username>`" to see your processes
- `top` – list processes, ordered by current CPU
  - Auto-updates
- `/proc` – virtual filesystem exposing kernel data structures
- `pmap` – display memory map of a process
- `strace` – prints a list of system calls from a process
  - Compile with "`-static`" to get cleaner traces

Process creation

- The `fork()` syscall creates a new process
  - Initializes new entry in the kernel data structures
  - **To user code, the function call returns twice**
    - Once for original process (parent) and once for new process (child)
    - Returns 0 in child process
    - Returns child pid in parent process
    - Both processes will continue executing concurrently
  - Parent and child have separate address spaces
    - Child's space is a duplicate of parent's at the time of the fork
    - They will diverge after the fork!
  - Child inherits parent's environment and open files

Process creation example

- Fork returns twice!



```
int main ()
{
    printf("Before fork\n");

    int pid = fork();

    printf("After fork: pid=%d\n", pid);

    return 0;
}
```

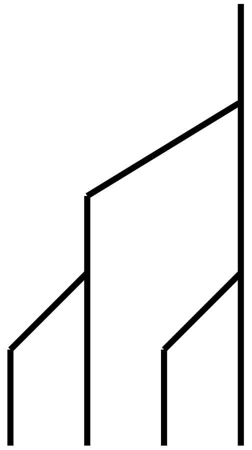
Process creation example

- What does this code do?

```
int main ()
{
    printf("Before fork\n");

    int pid1 = fork();

    printf("After fork: pid1=%d\n", pid1);

    int pid2 = fork();

    printf("After second fork: pid1=%d pid2=%d\n", pid1, pid2);

    return 0;
}
```

Process creation example

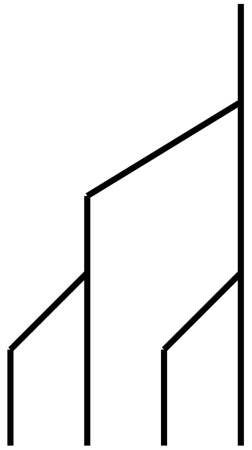
- Fork returns twice! (every time)
  - Beware of non-determinism and I/O interleaving



```
int main ()
{
    printf("Before fork\n");

    int pid1 = fork();

    printf("After fork: pid1=%d\n", pid1);

    int pid2 = fork();

    printf("After second fork: pid1=%d pid2=%d\n", pid1, pid2);

    return 0;
}
```

Process creation example

- Fork returns twice! (every time)
  - Beware of non-determinism and I/O interleaving

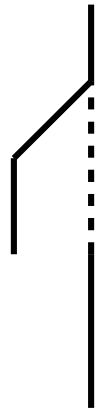


```
int main ()
{
    printf("Before fork\n");

    int pid1 = fork();

    printf("After fork: pid1=%d\n", pid1);

    int pid2 = fork();

    printf("After second fork: pid1=%d pid2=%d\n", pid1, pid2);

    return 0;
}
```

Exercise: Modify this program to fork a total of **three** processes

Parent/child process example

- Parents can wait for children to finish



```
int main ()
{
    printf("Before fork\n");

    int pid = fork();

    if (pid != 0) {     // parent
        wait(NULL);
        printf("Child has terminated.\n");

    } else {            // child
        printf("Child is running.\n");
    }

    printf("After fork: pid=%d\n", pid);

    return 0;
}
```

Process control syscalls

- **`#include <stdlib.h>`**
  - `getenv`: get environment variable value
  - `setenv`: change environment variable value
- **`#include <unistd.h>`**
  - `fork`: create a new process
  - `getpid`: return current process id (pid)
  - `exit`: terminate current process
  - `execve`: load and run another program in the current process
  - `sleep`: suspend process for specified time period
- **`#include <sys/wait.h>`**
  - `waitpid`: wait for a child process to terminate
  - `wait`: wait for all child processes to terminate

Fork/execve example

- Shells use `fork()` and `execve()` to run commands

```
int main ()
{
    printf("Before fork\n");
    int pid = fork();

    if (pid != 0) {      // parent
        wait(NULL);
        printf("Child has terminated.\n");

    } else {             // child
        printf("Child is running.\n");
        char *cmd    = "/bin/uname";
        char *args[] = { "uname", "-a", NULL };
        char *env[]  = { NULL };
        execve(cmd, args, env);
        printf("This won't print unless an error occurs.\n");
    }

    printf("After fork: pid=%d\n", pid);
    return 0;
}
```