

DOING MORE THINGS SIMULTANEOUSLY

- Concurrency can be achieved by **multiprocessing** and **time-sharing**.
 - Best definition for concurrency: “apparently simultaneous execution.”
- Concurrency is fundamental to distributed computing.
 - multiprocessing: many machines run simultaneously many programs
 - time-sharing: servers may be capable of serving multiple clients concurrently.
- Whether things run in a time-share fashion or on a multiprocessor computer is immaterial; the observable behaviour is the same.
- The fundamental unit of computation: a **process**.
 - a process consists in an **address space** and one (or more) **threads of execution** (i.e., instruction pointers).
 - * Each process receives a separate copy of all the variables. Each thread in a process have a copy of local variables but they all share the same global variables.

PROCESS CREATION

We thus create **singly threaded processes**.

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char** argv) {
    int i;
    int sum = 0;

    fork();

    for (int i=1; i<10000; i++) {
        sum = sum + i;
    }
    cout << "\nI computed " << sum << "\n";
}
```

PARENTS AND CHILDREN

- The call to `fork()` **duplicates** the current process; both processes continue execution from the instruction following the call to `fork()`.
- The initial process is the **parent**, and the newly created copy is called a **child**.
- What if we want the two processes to do different things?
 - Processes are identified in a Unix system by a unique **process identifier** (or PID for short), which is an integer.
 - `fork()` returns two **different** integers in the child and parent processes.
 - * In the child process `fork()` returns **zero**.
 - * In the parent process `fork()` returns **the PID of the newly created child**.
 - * So we provide different code for the parent and the child, and we surround them by appropriate `if` statements.

DIVERGING PROCESSES

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char** argv) {
    int i;
    int sum = 0;
    int pid;

    pid = fork();

    for (int i=1; i<10000; i++) {
        if (pid == 0)
            cout << "+"; // printed by child process
        else
            cout << "-"; // printed by parent process
        cout.flush();
        sum = sum + i; // done by both processes
    }
    if (pid == 0)
        cout << "\n[Child] I computed " << sum << "\n";
    else
        cout << "\n[Parent] I computed " << sum << "\n";
}
```

CHILDREN DOING SOMETHING COMPLETELY DIFFERENT

- The call `execve` **replaces completely** the current process with another executable. The arguments are the name of the command to execute, then two arrays of strings containing the command line arguments and the environment, just as the ones received by the function `main`.
- Suppose now that we want to run an external command (so we use `execve`)...
- ...but we want to continue the execution of the main program too.

CHILDREN DOING SOMETHING COMPLETELY DIFFERENT

- The call `execve` **replaces completely** the current process with another executable. The arguments are the name of the command to execute, then two arrays of strings containing the command line arguments and the environment, just as the ones received by the function `main`.
- Suppose now that we want to run an external command (so we use `execve`)...
- ...but we want to continue the execution of the main program too.
- We use a combination of `fork` and `execve`:

```
int childp = fork();
if (childp == 0) { // child
    execve(command, argv, envp);
}
else { // parent
    // code that continues our program
}
```

KNOW WHAT YOUR CHILDREN DO

- Again, suppose that we want to execute an external command (`execve` again),
- we still want to continue the execution of the main program,
- **but only after the external command has been completed.**

KNOW WHAT YOUR CHILDREN DO

- Again, suppose that we want to execute an external command (`execve` again),
- we still want to continue the execution of the main program,
- **but only after the external command has been completed.**

```
int run_it (char* command, char* argv [], char* envp[]) {
    int childp = fork();
    int status;

    if (childp == 0) { // child
        execve(command, argv, envp);
    }
    else { // parent
        waitpid(childp, &status, 0);
    }
    return status;
}
```

WHY CREATE PROCESSES?

- We will build eventually servers (programs that serve requests from clients).
- Instead of serving requests from one client at a time, our servers will handle many clients quasi-simultaneously.

```
loop
| listen for clients
| if a client requests connection then
|   fork
|   □
|   if child process then
|     handle client
|     terminate
|   □
forever
```

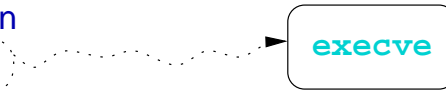
WHY CREATE PROCESSES?

- We will build eventually servers (programs that serve requests from clients).
- Instead of serving requests from one client at a time, our servers will handle many clients quasi-simultaneously.

```
loop
| listen for clients
| if a client requests connection then
|   fork
|   □
|   if child process then
|     handle client
|     terminate
|   □
| forever
```

- ... or even many types of clients

```
loop
| listen for clients
| if a client of type x requests connection then
|   fork
|   □
|   if child process then
|     launch server x
|     terminate
|   □
| forever
```



WHY CREATE PROCESSES? (CONT'D)

