

Concurrency

Stefan D. Bruda

CS 464/564, Fall 2023



- 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: a single machine runs multiple programs in an interleaved fashion (**context switching**)
- Whether things run in a time-share fashion or on a multiprocessor computer is immaterial; the observable behaviour is the same



- 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: a single machine runs multiple programs in an interleaved fashion (**context switching**)
- 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 of an **address space** and one (or more) **threads of execution** (i.e., instruction pointers)
 - Each process receives a separate copy of all the variables (address space)
 - Each thread in a process have a copy of local variables (stack) but they all share the rest of the address space (global variables, heap)



PROCESS CREATION

- We now 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";
}
```

to To other side. get the
Why did the multithreaded chicken
cross the road?
other to side. To the get



- 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**
- Processes are identified in a Unix system by a unique **process identifier** (or PID, unsigned 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 can provide different code for the parent and the child, by surrounding them in appropriate conditional 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 << "+";    // child process
        else
            cout << "-";    // parent process
        cout.flush();
        sum = sum + i;    // both processes
    }
    if (pid == 0)
        cout << "\n[Child] I computed " << sum << "\n";
    else
        cout << "\n[Parent] I computed " << sum << "\n";
}
```



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



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




- 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 run of the external command is complete (**synchronous** as opposed to asynchronous **execution**)



- 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 run of the external command is complete (**synchronous** as opposed to asynchronous **execution**)

```
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?



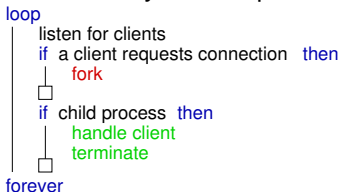
- We will build eventually servers (programs that serve requests from clients)
- Instead of serving requests from one client at a time, our servers will handle many clients quasi-simultaneously

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

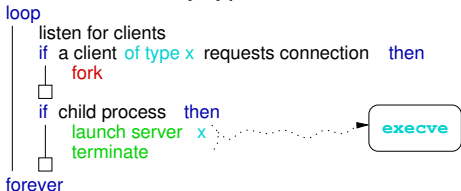


WHY CREATE PROCESSES?

- 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



- ... or even many types of clients





WHY CREATE PROCESSES? (CONT'D)

