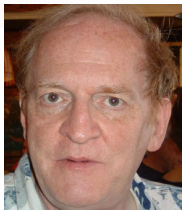
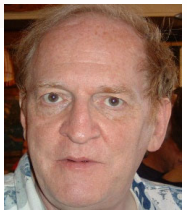
COSC 3360/6310
Monday, February 22

Announcements



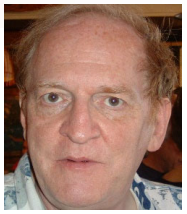
- First quiz now completely graded
  - Very slow process
  - Resolved to make next quiz self-grading
  - Answers to questions now posted on Prulu.com
- One week extension for the first assignment
- Second quiz will stay on Monday, March 1
- Do not be afraid to ask ***me*** assignment questions

Three honest reasons why you should ask instructor's help



- The programs wll be graded by the TAs
  - Will never know how much help you get from the instructor
- You show that you have been working on the assignment
  - Make extensions more likely
- You will not labelled for life as a bad programmer
  - We only remember final outcomes

Unfinished business



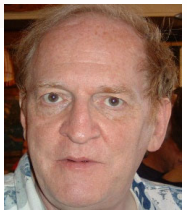
- Some people have not yet accepted the invitations I sent to their UH Mail accounts in January

- Need to reschedule a makeup exam.
  - Still time to catch up if you were unable to take the first quiz along with the other students.

Chapter II
Processes

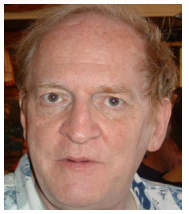
Jehan-François Pâris
jfparis@uh.edu

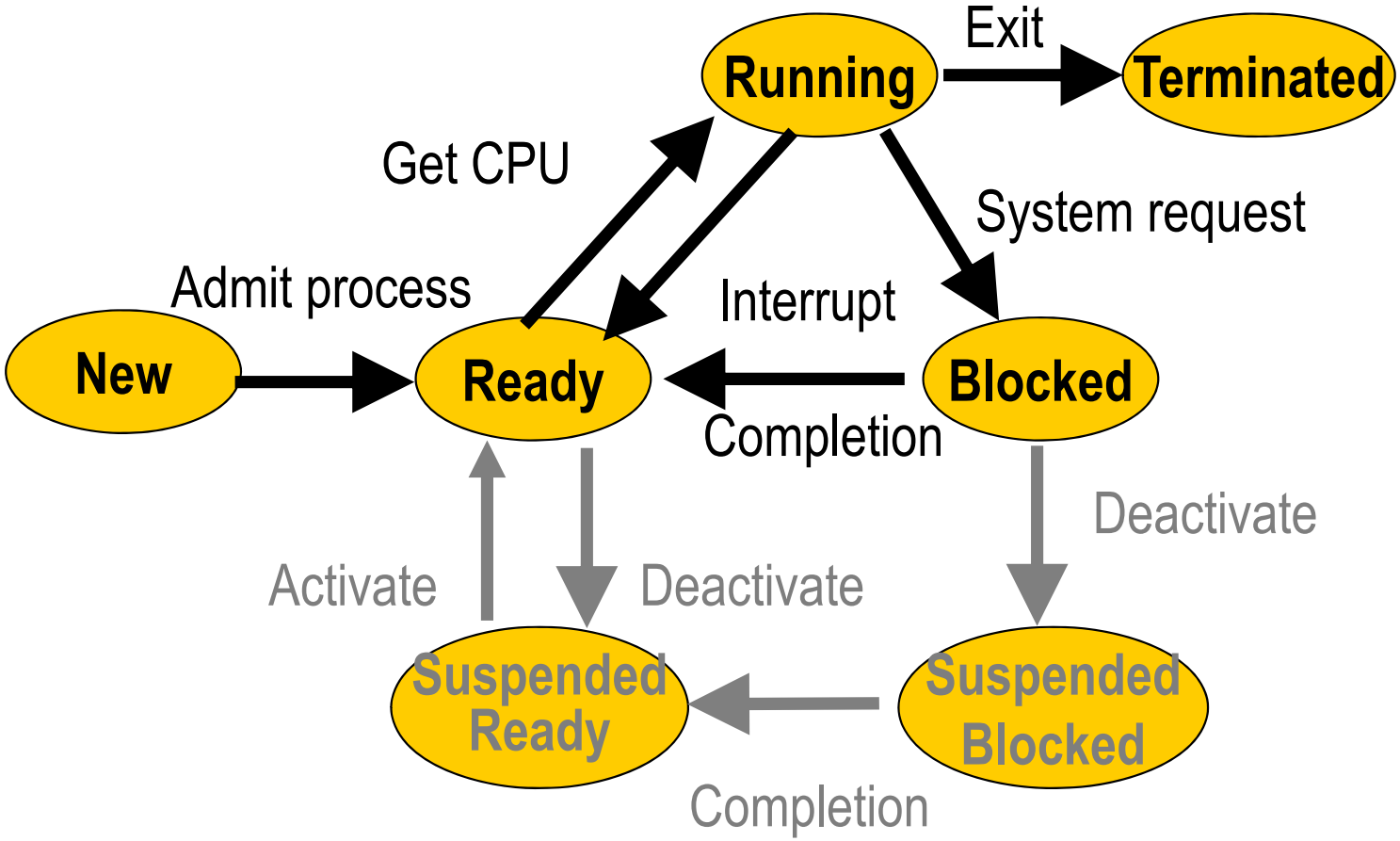
Chapter Overview

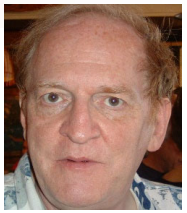


- Processes
- States of a process
- Operations on processes
  - **`fork()`**, **`exec()`**, **`kill()`**, **`signal()`**
- Threads and lightweight processes

How it works



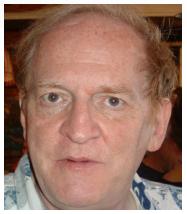

Running
Exit
Terminated
Get CPU
System request
Admit process
Interrupt
New
Ready
Blocked
Completion
Deactivate
Activate
Deactivate
Suspended Ready
Suspended Blocked
Completion



Operations on processes

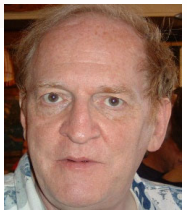
Process creation, deletion, …

Operations on processes



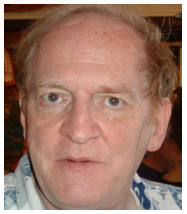
- Process creation
  - **`fork()`**
  - **`exec()`**
  - The argument vector
- Process deletion
  - **`kill()`**
  - **`signal()`**

Process creation



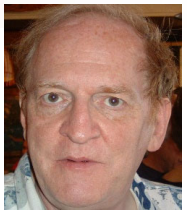
- Two basic system calls
  - **`fork()`** creates a carbon-copy of calling process sharing its opened files
  - **`execv()`** overwrites the contents of the process address space with the contents of an executable file

fork() (I)



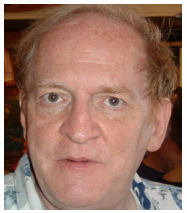
- First process of a system is created when the system is booted
- All other processes are forked by another process
  - Their ***parent process***
  - Said to be ***children*** of that process

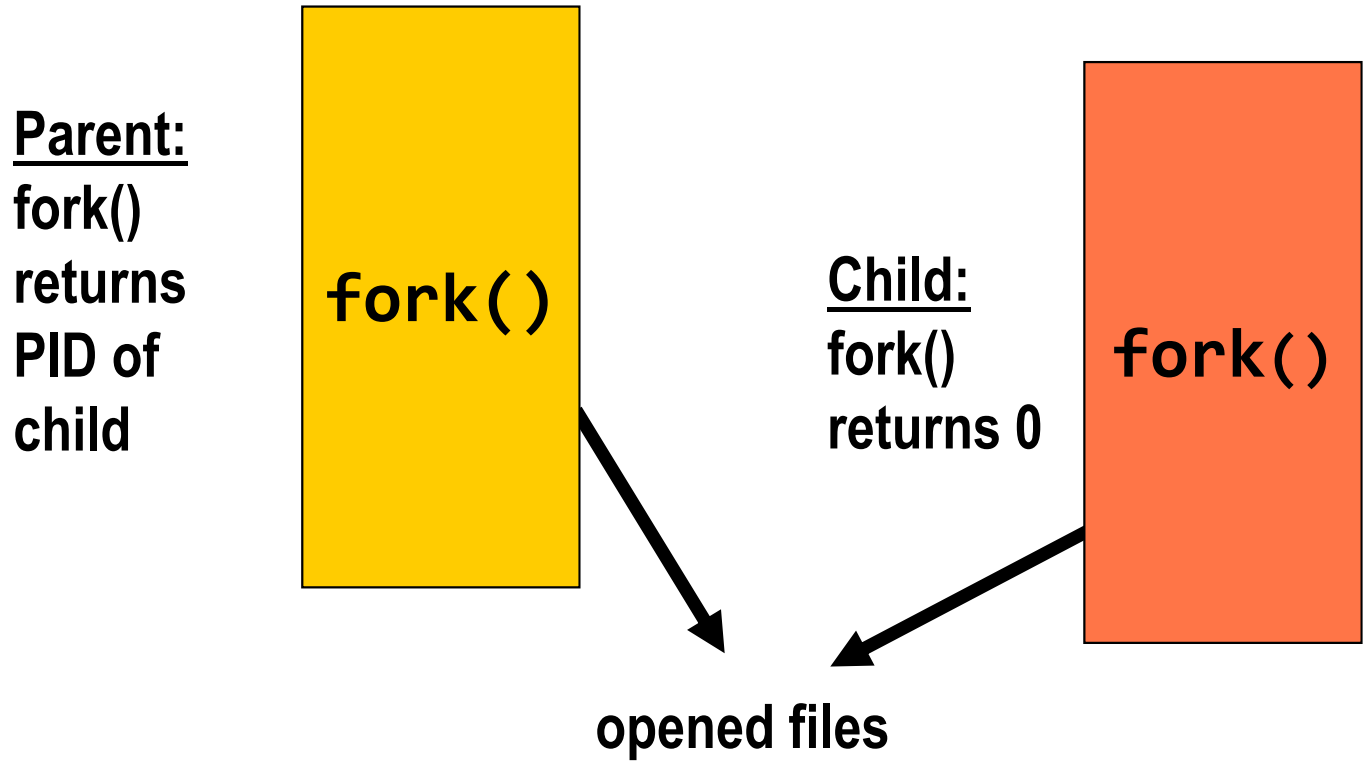
fork() (II)



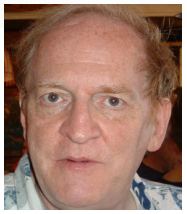
- When a process forks, OS creates an ***identical copy*** of forking process with
  - A ***new address space***
  - A ***new PCB***
- The ***only*** resources shared by the parent and the child process are the ***opened files***

fork() (III)




Parent:
fork()
returns
PID of
child
fork()
Child:
fork()
returns 0
fork()
opened files

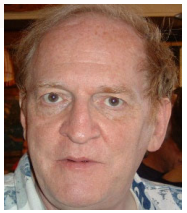
First example

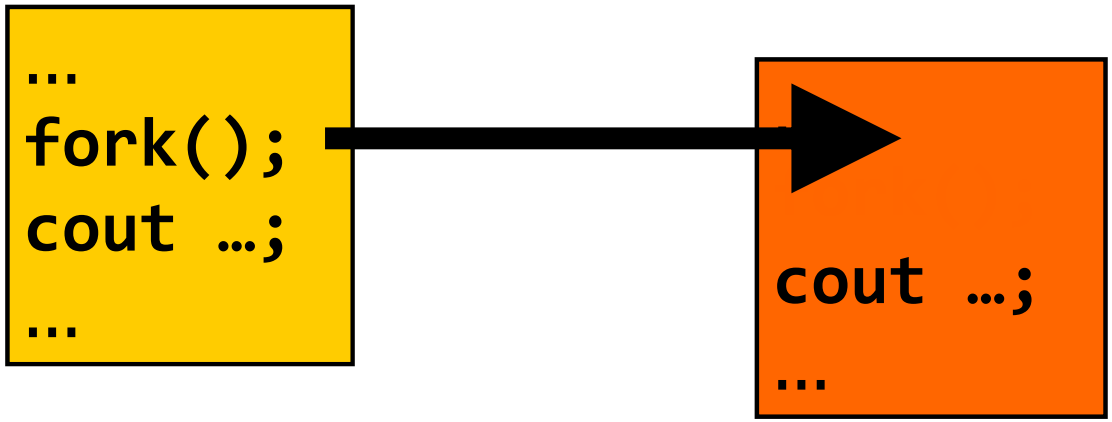


```
#include <iostream>
using namespace std;
main() {
    fork();
    cout << "Hello" << endl;
} // main
```

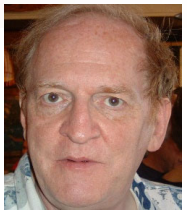
will print two lines as **`cout`** will be executed by ***both*** the parent and the child

How it works





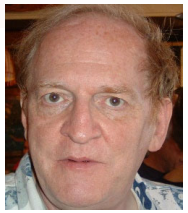
Second example

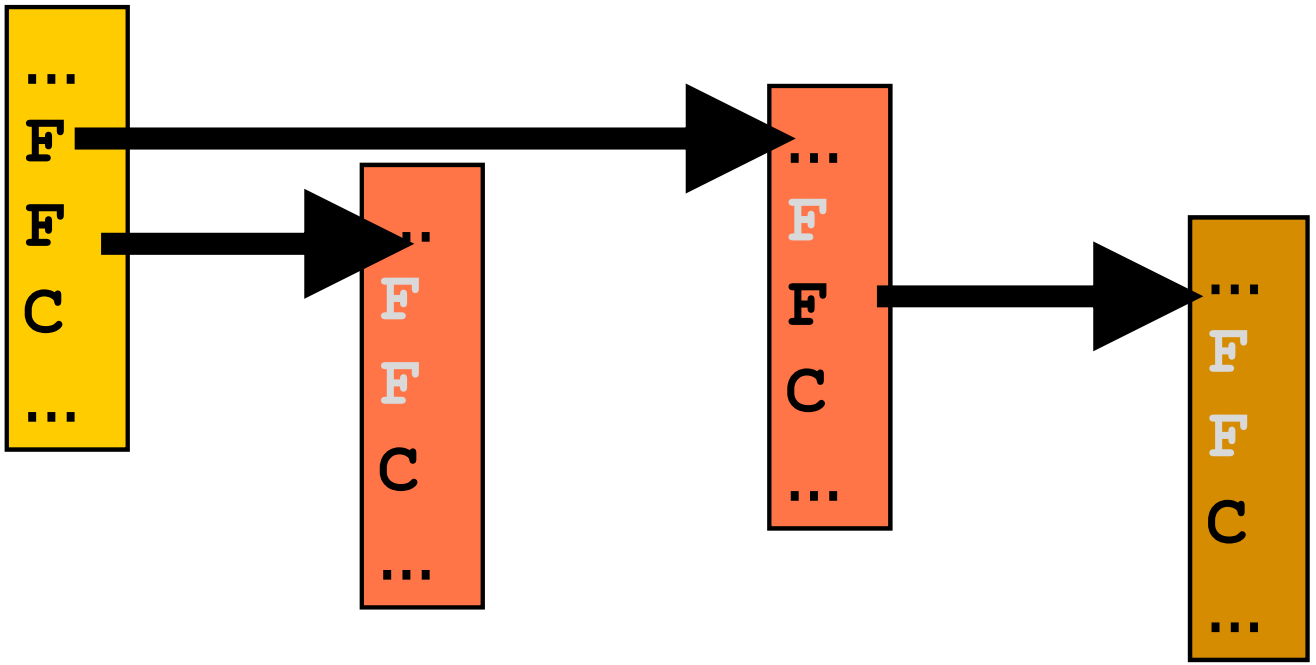


```
main() {
   fork();
   fork();
     cout << "Hello" << endl;
} // main
```

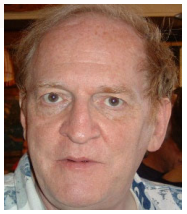
will print four lines as **`cout`** will be executed by the parent, its two children and its grandchild

How it works



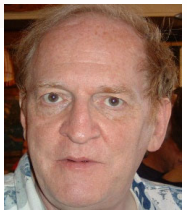


Something smarter



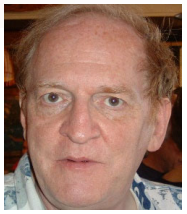
```
int pid;
pid = fork();
if (pid == 0) {
      // child process
      ...
} else {
      // parent process
      ...
}
```

First simplification



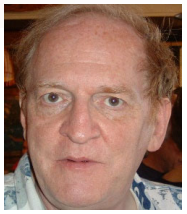
```
int pid;
pid = fork();
if (pid == 0) {
      // child process
      ...
      _exit(0); // normal exit
} // if
// parent process continues
      ...
```

Second simplification



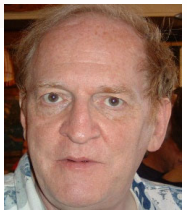
```
int pid;
if ((pid = fork())== 0) {
      // child process
      ...
      _exit(0); // normal exit
} // if
// parent process continues
      ...
```

Waiting for child completion



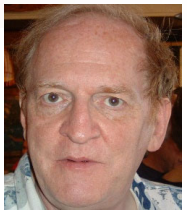
- **`wait(0)`**
  - Waits for the completion of any child
  - No wait if any child has already completed
- **`while (wait(0) != kidpid)`**
  - Waits for the completion of a specific child identified by its ***pid***

An example (I)



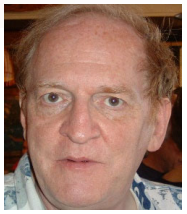
```
#include <iostream>
#include <sys/types.h>
#include <sys/wait.h>
using namespace std;
```

An example (II)



```
main() {
     int pid;
     if((pid = fork()) == 0)  {
          cout << "Hello !" << endl;
           _exit(0);
     } // child
     wait(0);
     cout << "Goodbye!" << endl;
} // main
```

Notes

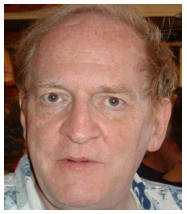


- Unix keeps in its process table all processes that have terminated but their parents have not yet waited for their termination
  - They are called ***zombie processes***
- The statement

```
while (kidpid != wait(0));
```

  is a loop with an ***empty body***

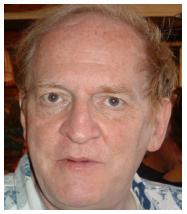
Putting everything together



```
int kidpid;
if ((kidpid = fork())== 0) {
        // child process
        ...
        _exit(0); // normal exit
} // if
// parent waits for child
while (wait(0) != kidpid);
        ...
```

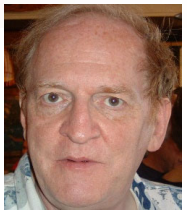
Must use the while loop if the process has already forked other children

exec



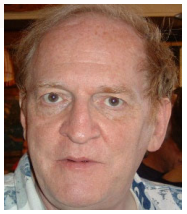
- Whole set of exec() system calls
- Most interesting are
  - **`execv(pathname, argv)`**
  - **`execve(pathname, argv, envp)`**
  - **`execvp(filename, argv)`**
- All **`exec()`** calls perform the same two tasks
  - Erase current address space of process
  - Load specified executable

execv



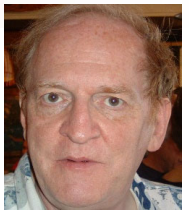
- **`execv(pathname, argv)`**
  - **`char pathname[]`**
    - ***full pathname*** of file to be loaded:
      `/bin/ls` instead of **`ls`**
  - **`char argv[][]`**
    - the ***argument vector***:
      passed to the program to be loaded

Argument vector (I)

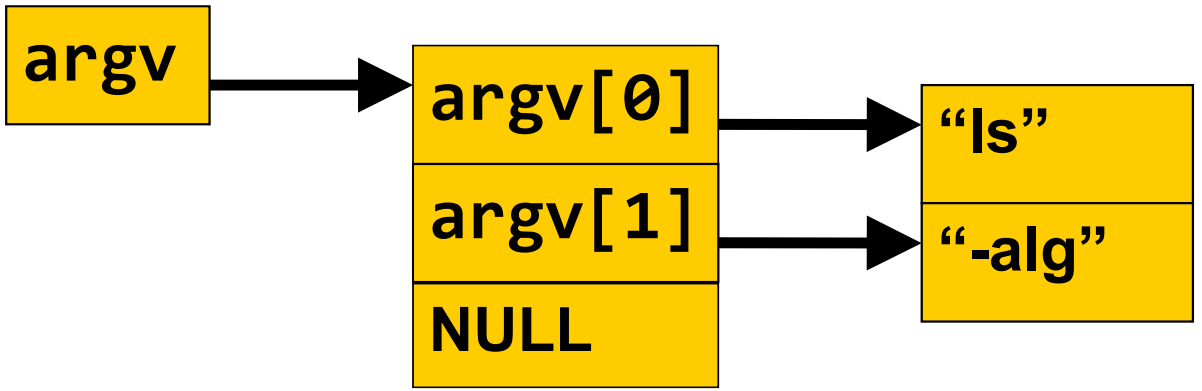


- An array of pointers to the individual argument strings
  - **`arg_vector[0]`** contains the name of the program *as it appears in the command line*
  - Other entries are parameters
  - End of the array is indicated by a **NULL** pointer

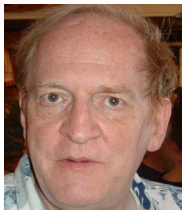
Argument vector (II)



- `char argv[][];`
- `char **argv;`

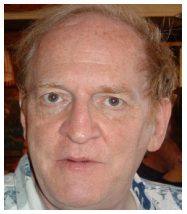


execve() and execvp()



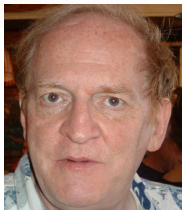
- **`execve(pathname, argv, envp)`**
  - Third argument points to a list of environment variables
- **`execvp(argv[0], argv)`**
  - Lets user specify a command name instead of a full pathname
  - Looks for **`argv[0]`** in list of directories specified in environment variable **`PATH`**

Putting everything together



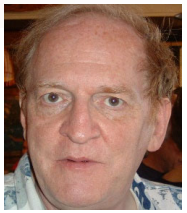
```
int pid;
if ((pid = fork()) == 0) {
   // child process
   ...
   execvp(filename, argv);
   _exit(1); // exec failed
 }  // if
 while (pid != wait(0));
 // parent waits
  ...
```

Observations (I)

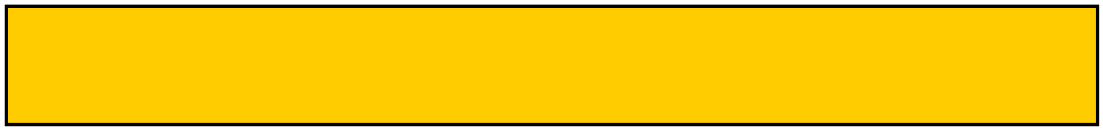


- Not cheap
  - `fork()` makes a ***complete copy*** of parent address space
    - ***Very costly*** in a virtual memory system
  - `exec()` thrashes that address space
- Best solution is copy-on-write (COW)

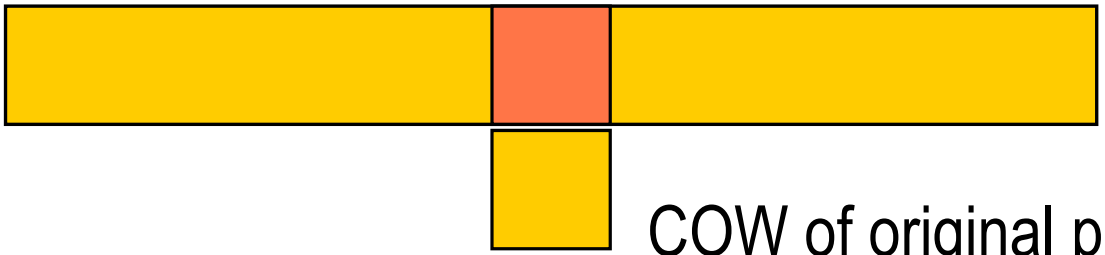
Copy-on-write



Parent and child share same address space

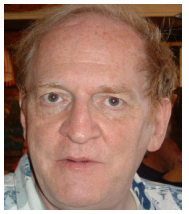


When either of them modifies a page,
other gets its ***own copy*** of original page



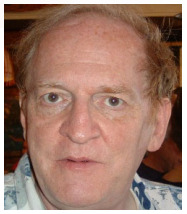
COW of original page

Copy-on-write as a lazy approach



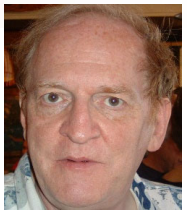
- Copy-on-write postpones address space copying until it is actually needed
  - Do the strict minimum
- ***Lazy approach***
  - Betting that very little copying will be actually needed
    - An **`execv()`** will quickly follow
- Opposite is ***eager approach***

Observations (II)



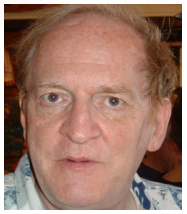
- Neither **`fork()`** nor **`exec()`** affect opened file descriptors
  - They remain unchanged
- Important for Unix I/O redirection mechanism

How this happened



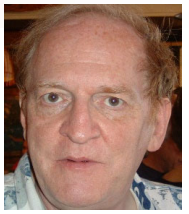
- Fork was not that expensive on a minicomputer with a 16-bit address space
  - Never had to copy more than **64KB**
- Using a fork/exec allowed a very easy implementation of I/O redirection
  - After the **`fork()`** thus in the child
  - Before the **`exec()`** while parent is still in control

A very basic shell (I)



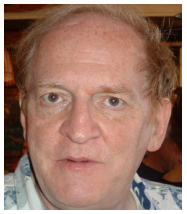
```
for (;;) {
    parse_input_line(argv);
    if built_in(argv[0]) {
        do_it(arg_vector);
        continue;
     } //built_in command
     path = find_path(argv[0]);
```

A very basic shell (II)



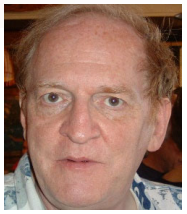
```
        if ((pid = fork()) == 0) {
            // put here I/O
            // redirection code
           execv(path, argv);
           _exit(1); // execv failed
         } //child process
         if (interactive())
             while (wait(0) != pid);
} // main for loop
```

Comments



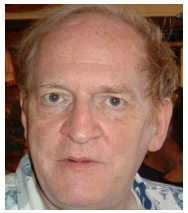
- Shell built-in commands include
  - **`exit`**
    terminates the shell
  - **`cd`**
    changes current directory
- Commands are assumed to be interactive
  - ***Non-interactive*** commands end with an “&”

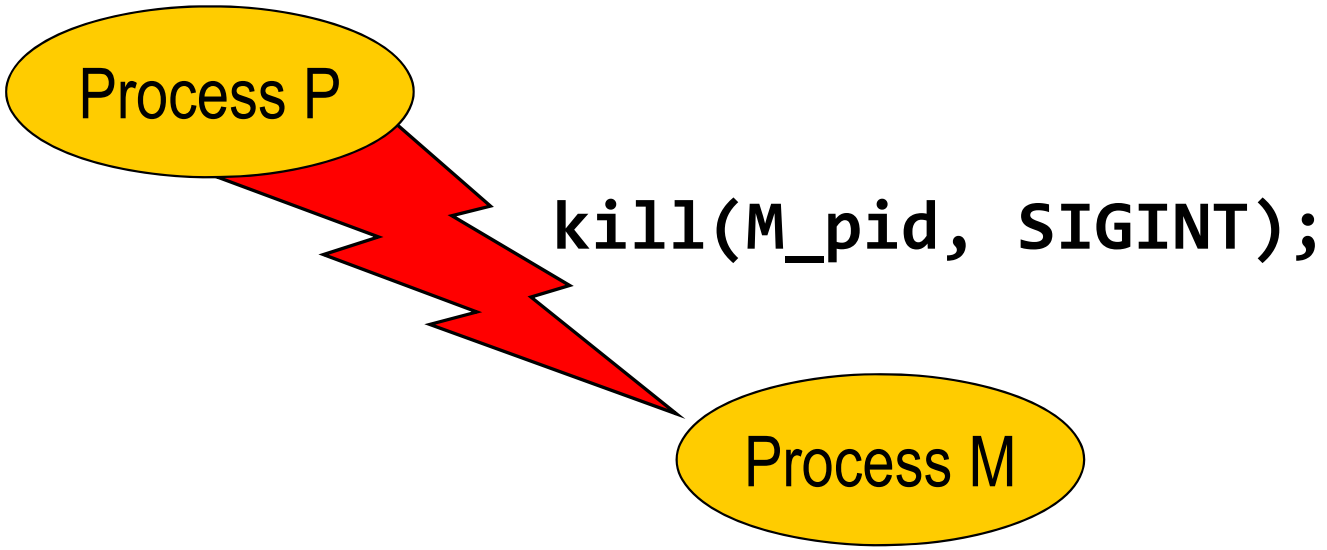
Terminating a process (I)



- Sending a signal:
  - **`kill()`** has two arguments
    - The ***process id*** of the receiving process
    - A ***signal name*** or a ***signal number***
- **`#include <signal.h>`**
  `kill(this_pid, this_signal);`
- Process receiving the signal will ***terminate***

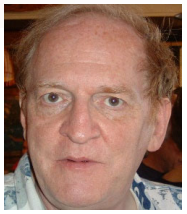
Terminating a process (II)





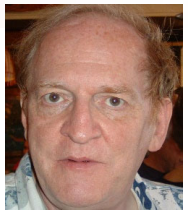
What should I do? **AARGH!**

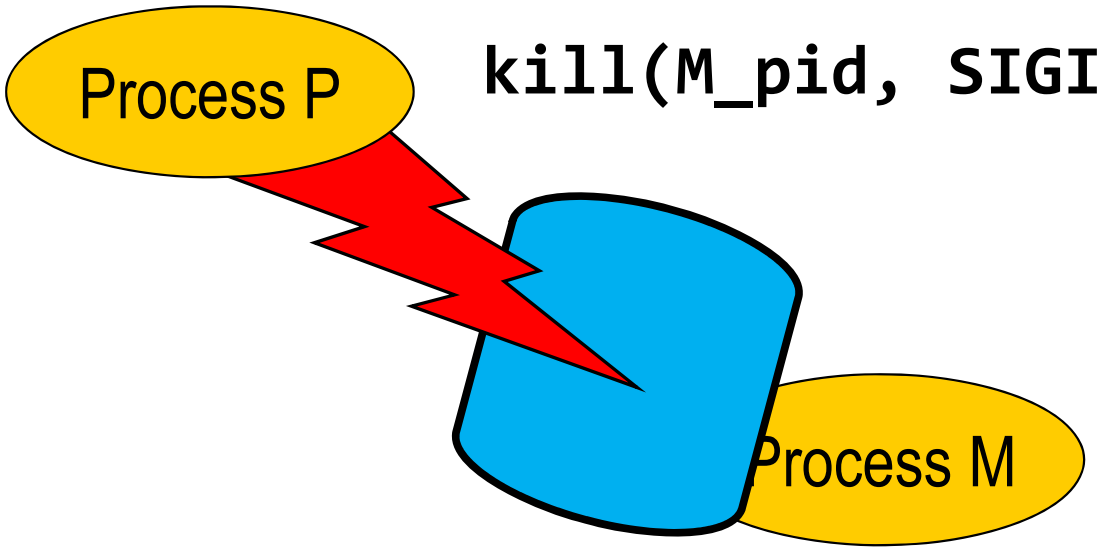
Catching a signal (I)



- The process receiving signal can ***catch*** it by using **`signal()`**
  - Will not terminate
- **`signal(a_signal, catch_it);`**
  - **`catch_it`** function is executed when **`a_signal`** signal is received.
- The ninth signal, **`SIGKIL`**, cannot be caught.

Catching a signal (II)





Process is now **shielded** by **`signal()`** call