

Operating System Labs

Yuanbin Wu
cs@ecnu

Announcement

- Project 1 due
 - 21:00 Oct. 2
- Project 2 due
 - 21:00 Oct. 23
- TA email update:
 - os_ecnu_class1@163.com
 - os_ecnu_class2@163.com

Operating System Labs

- The abstraction of process
- CPU virtualization
 - Low level and high level mechanisms
- Process API
- Project 2

Process

- The abstraction: Process
 - Running programs
- What does a process consist of?
 - CPU
 - Program Counter (PC)
 - Stack Pointer / Frame Pointer
 - Memory
 - Address space
 - Disk
 - Set of file descriptors

Process

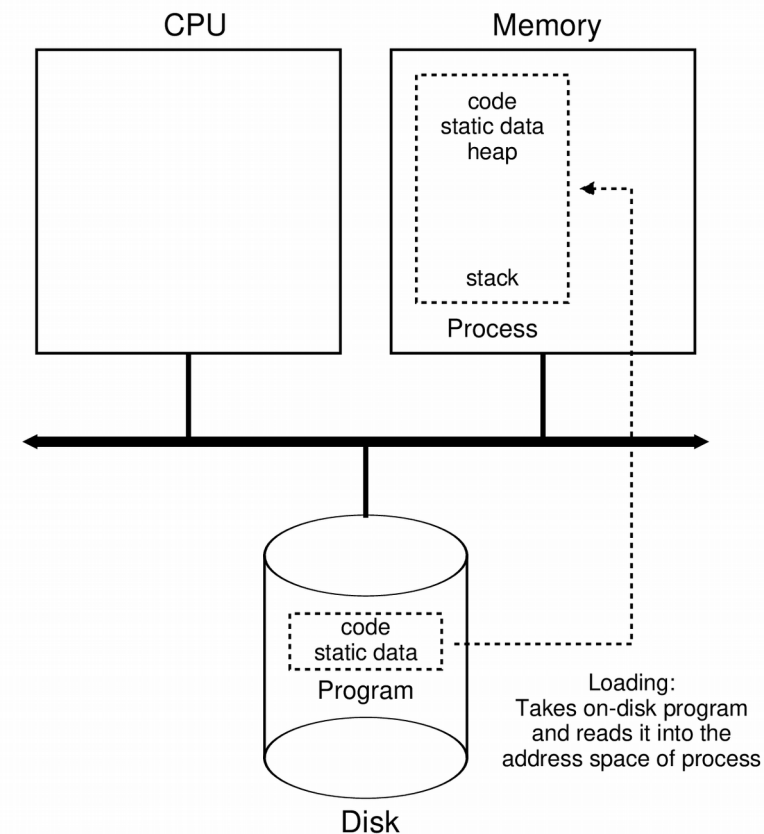
- What does a process consist of?
 - proc file system
 - Example
 - `cat /proc/<PID>/status`
 - `cat /proc/<PID>/maps`
 - `cat /proc/<PID>/fd`
 - `cat /proc/<PID>/io`
 - Provide a method of communication between kernel space and user space
 - `ps` command

Process

- Process API
 - Create
 - Destroy
 - Wait
 - Miscellaneous Control
 - Get status

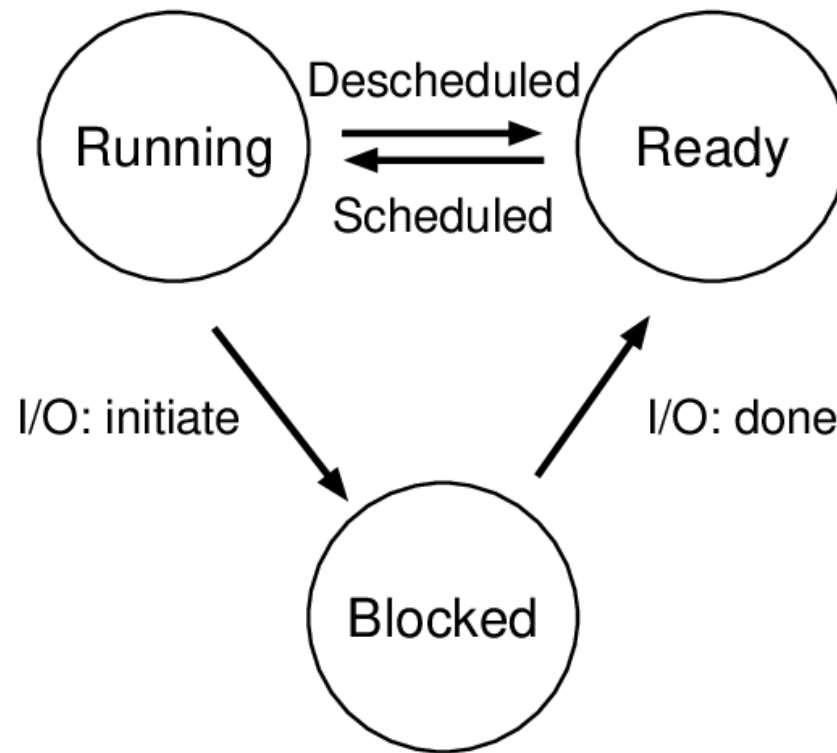
Process

- Details of process creation
 - Load code and static data
 - local variables, function calls
 - Establish stack
 - local variables, function calls
 - Init heap
 - malloc, free
 - Allocate file descriptors
 - STDIN_FILENO
 - STDOUT_FILENO
 - STDERR_FILENO



Process

- Process States



Process

- Process States

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process ₀ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process ₁ now done

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process ₀ initiates I/O
4	Blocked	Running	Process ₀ is blocked,
5	Blocked	Running	so Process ₁ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process ₁ now done
9	Running	–	
10	Running	–	Process ₀ now done

CPU Virtualization

- What
 - Provide the illusion of many CPUs
- Why
 - Multi-task
- How
 - Time sharing

CPU Virtualization

- Mechanisms
 - Low level mechanisms
 - Context switch
 - High level intelligence
 - Scheduling policy

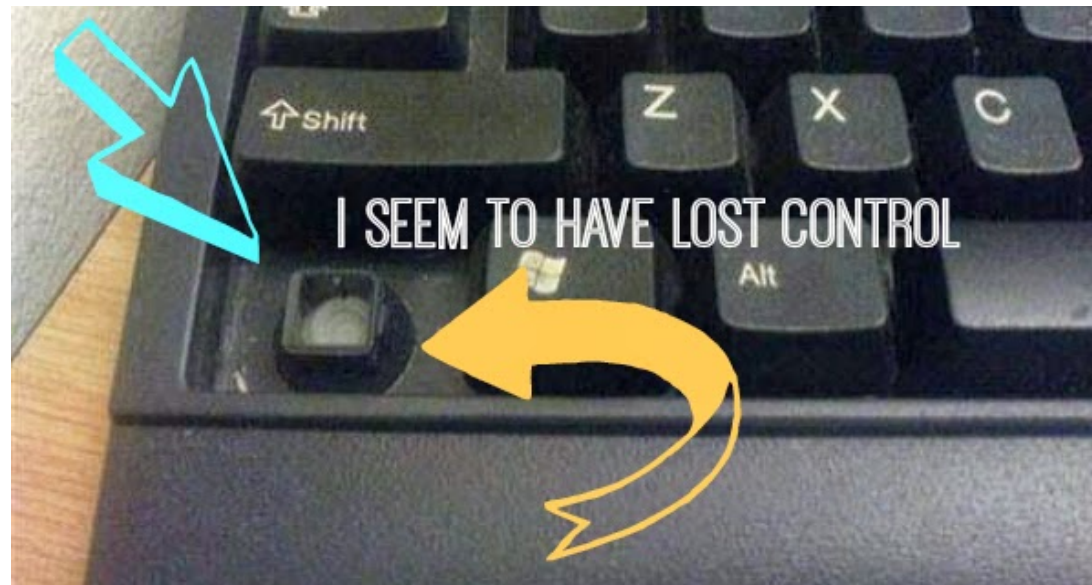
Limited Direct Execution

- Low level mechanisms
 - Direct Execution
 - Just run a program on CPU directly

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute call main()	Run main()
	Execute return from main
Free memory of process	
Remove from process list	

Limited Direct Execution

- Problems of direct execution
 - No control
 - Visit any memory address
 - Open any file
 - Directly play with hardwares (e.g. I/O)



Limited Direct Execution

- Limited Direct Execution
 - Kernel model and user model
 - “restricted operations”
 - By the kernel
 - When a thread need do restricted operations
 - System call

Limited Direct Execution

- User mode
 - Code is restricted in what it can do
- Kernel mode
 - Code can do what it likes
 - Issue I/O, executing all types of instructions
- How to switch?
 - System call

Limited Direct Execution

- Hardware supports on system call
 - A bit in CPU identifies kernel/user mode
 - “trap” instruction
 - “return-from-trap” instruction
 - Save the registers before do the restricted operation (kernel stack)

OS @ run
(kernel mode)

Hardware

Program
(user mode)

Run main()
...
Call system call
trap into OS

Limited Direct Execution

- Switching between processes
 - Cooperative approach
 - OS trusts the process to yield CPU properly
 - Non-cooperative approach
 - OS revokes the control of CPU periodically
 - Time interrupt
 - Scheduler

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember addresses of... syscall handler timer handler	
start interrupt timer	start timer interrupt CPU in X ms	
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call <code>switch()</code> routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B
		...

Limited Direct Execution

- Low-level mechanisms: summary
 - Direct execution
 - Limited direct execution
 - Switch between processes

Scheduling Policy

- High level intelligence
 - Scheduling policy
 - First In, First Out
 - Shortest job first
 - Shortest time to complete first
 - Round Robin

CPU virtualization

- Summary of CPU virtualization
 - Low level mechanisms
 - A little hardware support goes a long way
 - High level mechanisms

Process API

- Process API
 - `fork()`, `exec()`, `wait()`, `exit()`
 - Create, execute, wait and terminate a process
 - May be the strangest API you've ever met

Process API

- `fork()`
 - Create a new process
 - Exactly copy the calling process
- The return code of `fork()` is different
 - In parent: `fork()` return the pid of the child
 - In child: `fork()` return 0
- Who will run first is not determined

Process API

- `wait()`
 - Wait for child to finish his job
 - The parent will not proceed until `wait()` return.
- `waitpid()`

Process API

- `exec()`
 - Execute a different program in child process
- A group of system calls:
 - `execl`, `execv`, `execle`, `execve`, `execvp`, `execvp`, `fexecv`

Process API

- Some Coding
 - fork
 - fork, wait
 - fork, wait, execvp

Process API

- What's happening behind fork()?
 - The child get a “copy” of parent's data space, stack, heap
 - the system call: clone()
 - “Copy-on-write”
 - Not really copy the data, but share the data with “read only” flag
 - If parent or child writes on a shared address, the kernel make a copy of that piece of memory only (usually a page)

Process API

- What's happening behind `fork()`?
 - File sharing
 - fd
 - File offsets

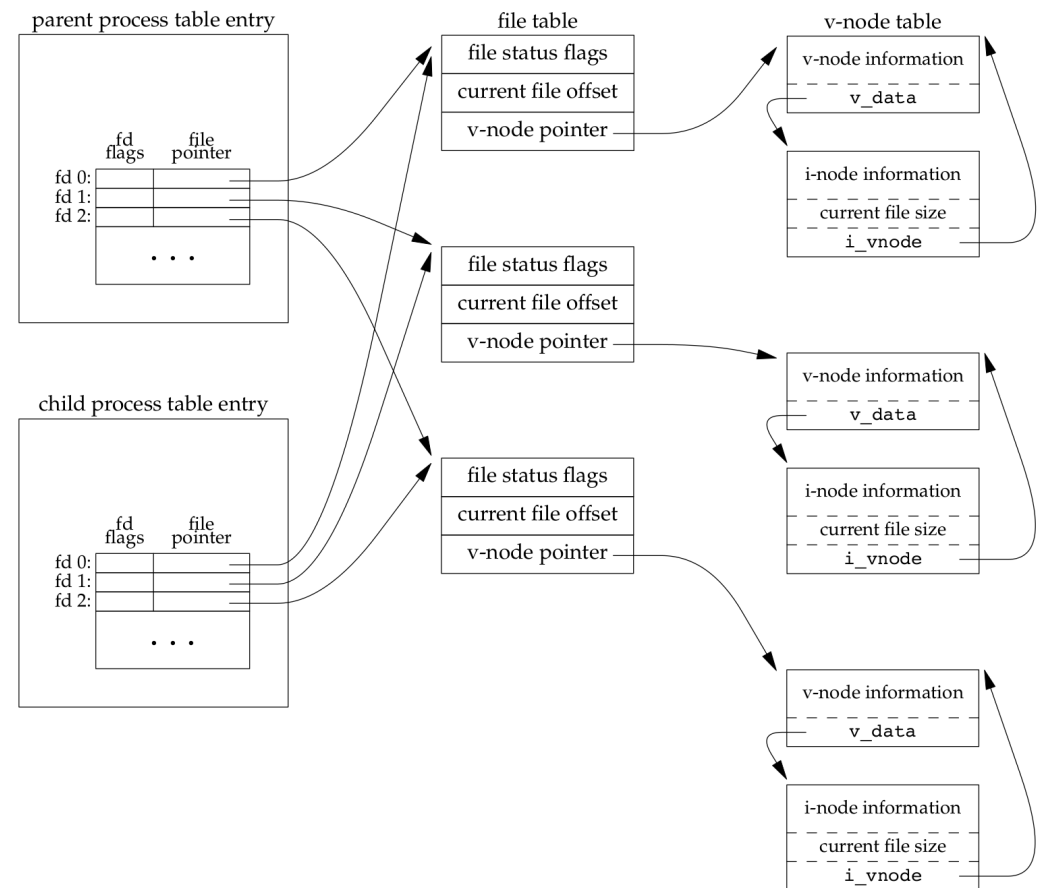


Figure 8.2 Sharing of open files between parent and child after `fork`

Process API

- What's happening behind `fork()`?
 - Other shared data:
 - User ID, group ID...
 - Current working directory
 - Environment
 - Memory mapping
 - Resources limits
 - ...

Process API

- What's happening behind `exit()`?
 - Close all fds, release all memory, ...
 - Inform the **exit status** to the parent process, which can be captured by `wait()`

Process API

- What's happening behind wait()?
 - The parent terminates first?
 - The init process (PID=0)
 - The child terminates first?
 - The kernel keeps a small amount of information for every terminating process
 - Available when the parent calls wait()
 - PID, termination status, the amount of CPU time
 - zombies

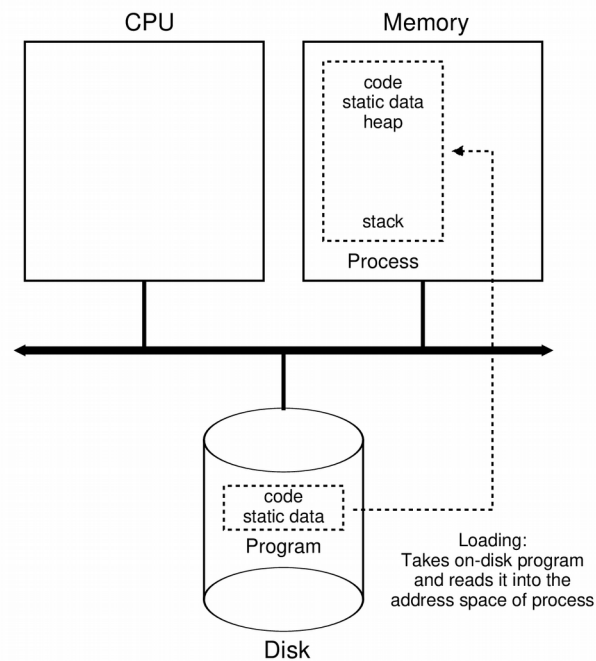


Process API

- What's happen behind wait()/waitpid()
 - wait(): block the caller until a child process terminates
 - waitpid(): wait which child, and some other options

Process API

- What's happening behind `exec()`?
 - Replace the current process with a new program from disk
 - Text, data, heap, stack
 - Start from the `main()` of that program



Process API

- Process API summary
 - `fork()`: create a new process
 - `wait()`: wait for a child
 - `exit()`: destroy a process
 - `exec()`: execute a program in child

Project2

- Implement your own shell
 - Use fork, wait, execvp
 - Also open, close, dup2

Project2 Details

- Basic shell
 - Run your shell by: ./mysh
 - It will print a prompt:

```
mysh>
```

- You can type some commands

```
mysh> ls
```

- Hit ENTER, the command will be executed

Project2 Details

- Build-in Commands
 - When “mysh” execute a command, it will check weather it is a **build-in** or not.
 - For build-in commands, you should involve your implementation.
 - They are:
 - exit
 - wait
 - cd
 - pwd

Project2 Details

- Redirection

- Your shell should support redirection:

```
mysh> ls -l > output
```

- The file “output” contain the result of “ls -l”

Project2 Details

- Background Jobs

- Your shell should be able to run jobs in the background

mysh> ls &

- Your shell will continue to work rather than wait.

Project2 Details

- Batch mode
 - Your shell should be able to run in batch mode

```
./mysh batch_file
```

- Your shell will run the commands in batch_file
- E.g, “batch_file” contains

```
ls -l
```

```
cat batch_file
```

Project2 Details

- Bonus: Pipe
 - The pipe connect the input/output of different commands

```
mysh> grep "hello" FILE | wc -l
```

- How many lines have "hello"