Operating System Labs

Yuanbin Wu
cs@ecnu

Announcement

- Project 1 due
  - 21:00 Oct. 6

Operating System Labs

- The abstraction of process
- CPU virtualization
  - Low level and high level mechanisms
- Process API
- Project 1a

The Abstraction of Process

- 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

The Abstraction of Process

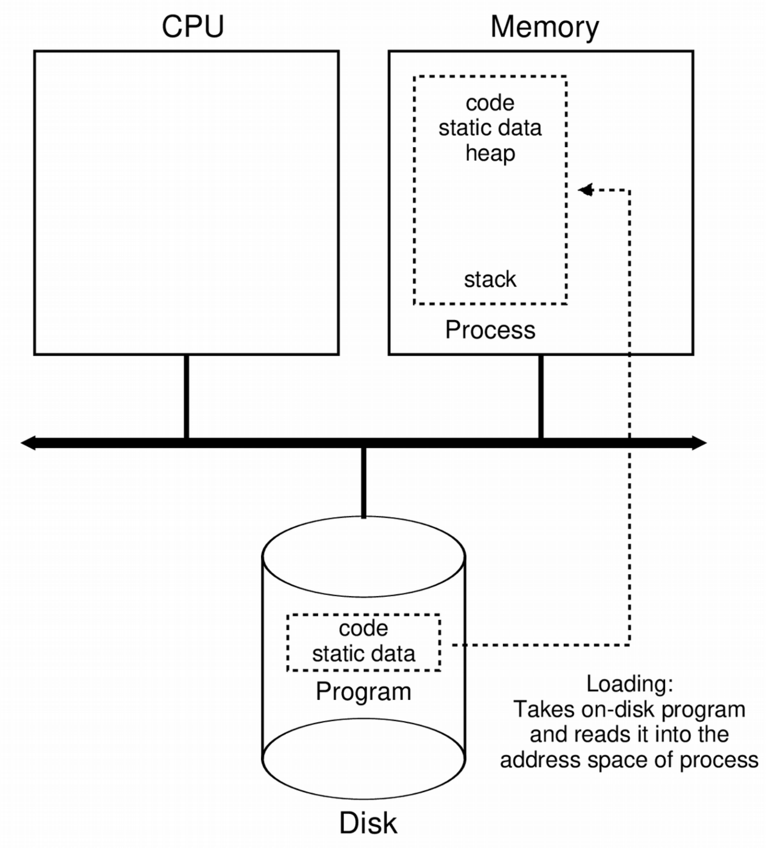
- proc file system
  - “Everything is file”
  - Example
    - cat /proc/<PID>/status
    - cat /proc/<PID>/maps
    - cat /proc/<PID>/fd
    - cat /proc/<PID>/io
  - Provide a method of communication between kernel space and user space
    - ps command

The Abstraction of Process

- Operations on a Process
  - Create
  - Destroy
  - Wait
  - Miscellaneous Control
  - Get status

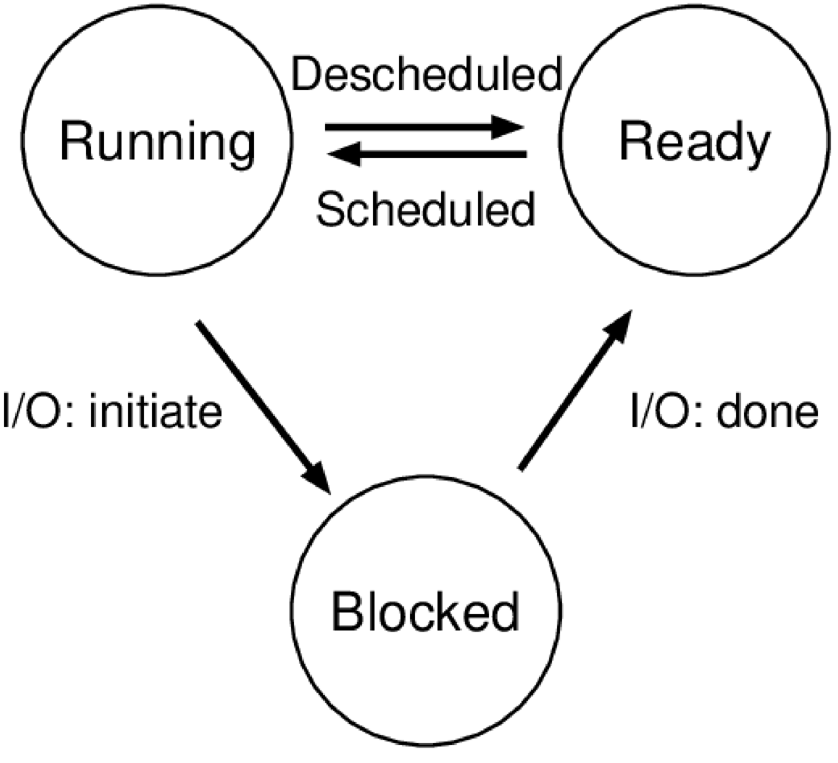
The Abstraction of Process

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



The Abstraction of Process

- Process States



The Abstraction of 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

The Abstraction of Process

- Data structures

```c
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                   // Start of process memory
  uint sz;                     // Size of process memory
  char *kstack;                // Bottom of kernel stack
                               // for this process
  enum proc_state state;       // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;           // Current directory
  struct context context;      // Switch here to run process
  struct trapframe *tf;        // Trap frame for the
                               // current interrupt
};
```

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

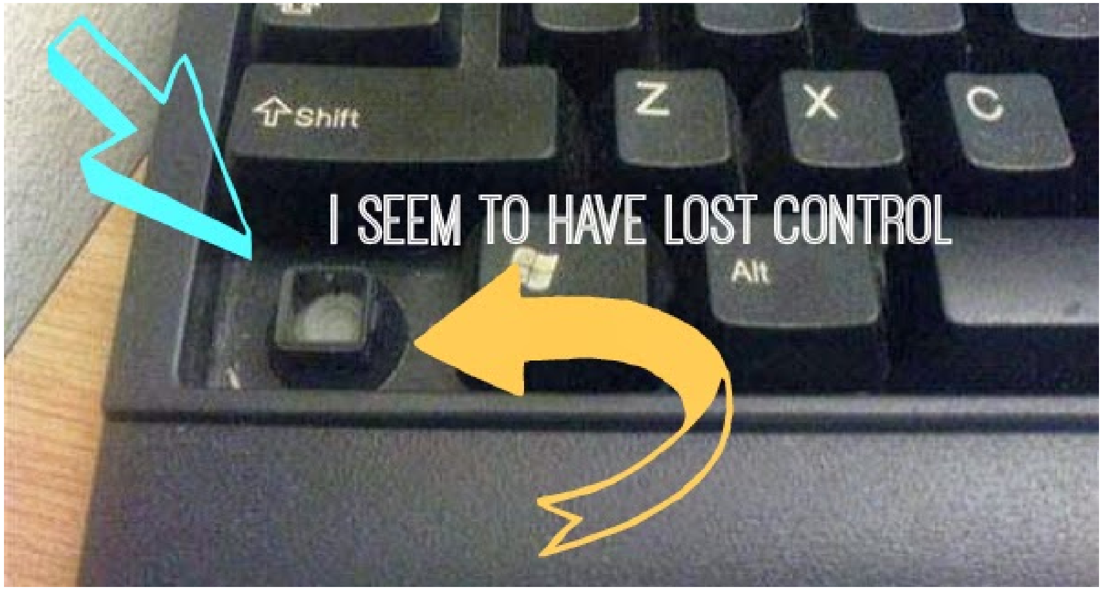
CPU Virtualization

- 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	

Direct Execution

- Problems of direct execution
  - Visit any memory address
  - Open any file
  - Directly play with hardwares (e.g. I/O)



Lost control

Limited Direct Execution

- Limited Direct Execution
  - Kernel model and user model
  - “restricted operations”
    - By OS
  - When a thread needs “restricted operations”
    - System call

Limited Direct Execution

- User mode
  - The behavior of the code is restricted
- Kernel mode
  - The code can do what it likes to do
    - Issue I/O, executing all types of instructions,…
- How to switch?
  - System call

System Call

- 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 `switch()` 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 Roubin

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, execlp, 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

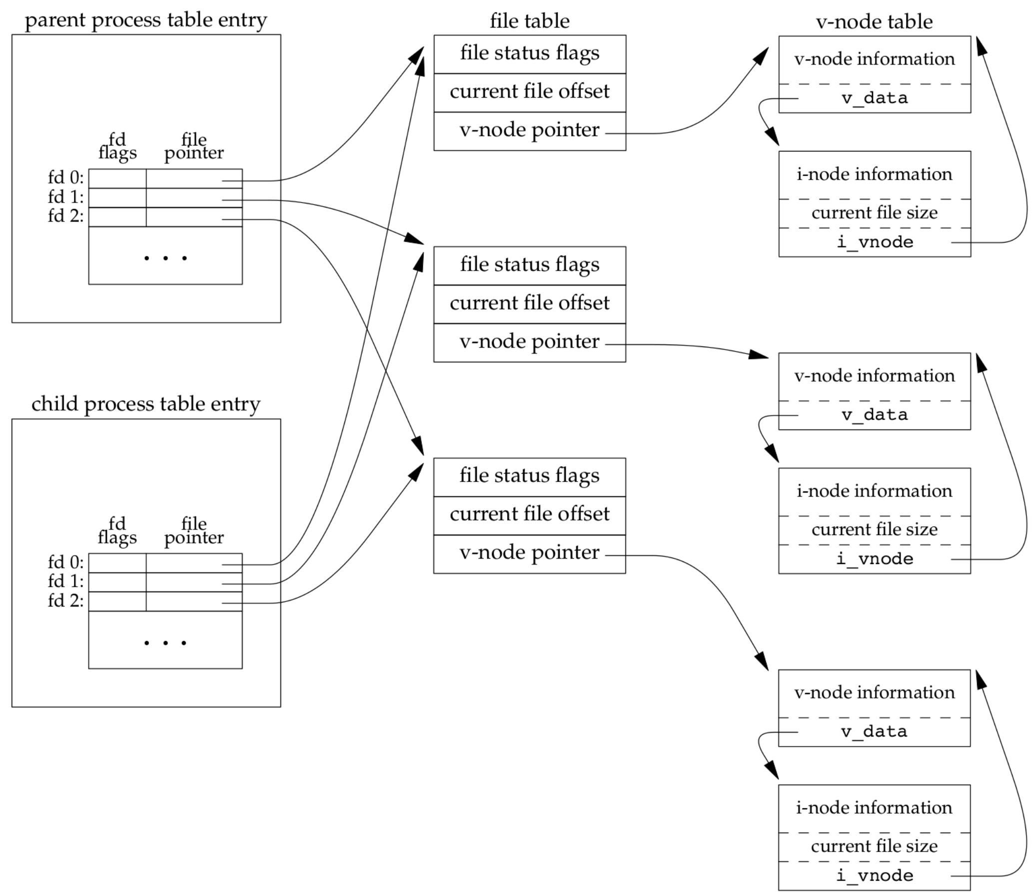


Figure 8.2 Sharing of open files between parent and child after `fork`

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

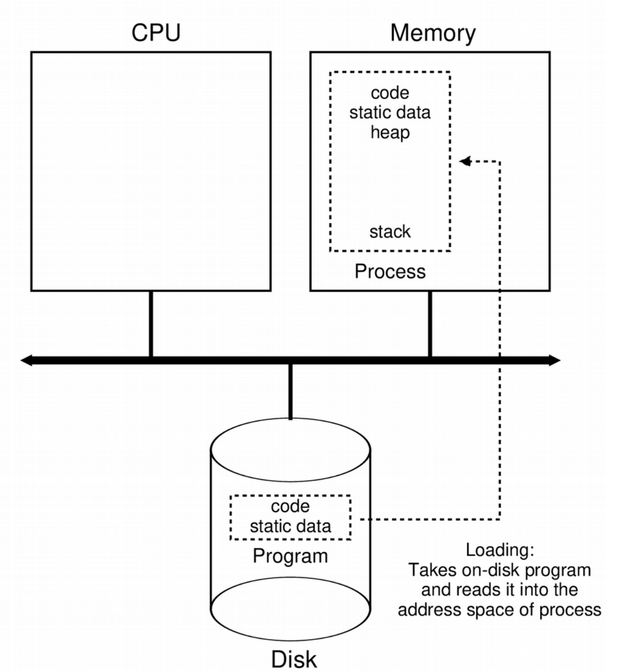
- What's happening behind wait()?
  - The parent terminates first?
    - The init process (PID=0)
  - The child terminates first?
    - The kernel keeps a small amount of information for every terminating process
    - Available when the parent calls wait()
      - PID, termination status, the amount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

Project1a

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

Project1a 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

Project1a 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

Project1a Details

- Redirection
  - Your shell should support redirection:

    mysh> ls -l > output

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

Project1a 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.

Project1a 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

Project1a Details

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

    mysh> grep “hello” FILE | wc -l

  - How many lines have “hello”

Project1b Details

- Adding a system call for xv6
  - Understanding the low-level mechanism
  - Kernel mode, user mode
  - Trap
  - Interrupt handler

Project1b Details

- The system call
  - int getreadcount()
  - Return how many times the read() system call has been called

Project1b Details

- Get familiar with xv6
  - QEMU emulator
    - Installed with make
  - Compile and run xv6
    - Compile: make
    - Run: make qemu-nox
    - Debug: make qemu-nox-gdb