Processes (Intro)

Yannis Smaragdakis, U. Athens

Process: CPU Virtualization

- Process = Program, instantiated
  - has memory, code, current state
- What kind of memory do we have?
  - registers + address space
- Let's do a hardware review

Process API

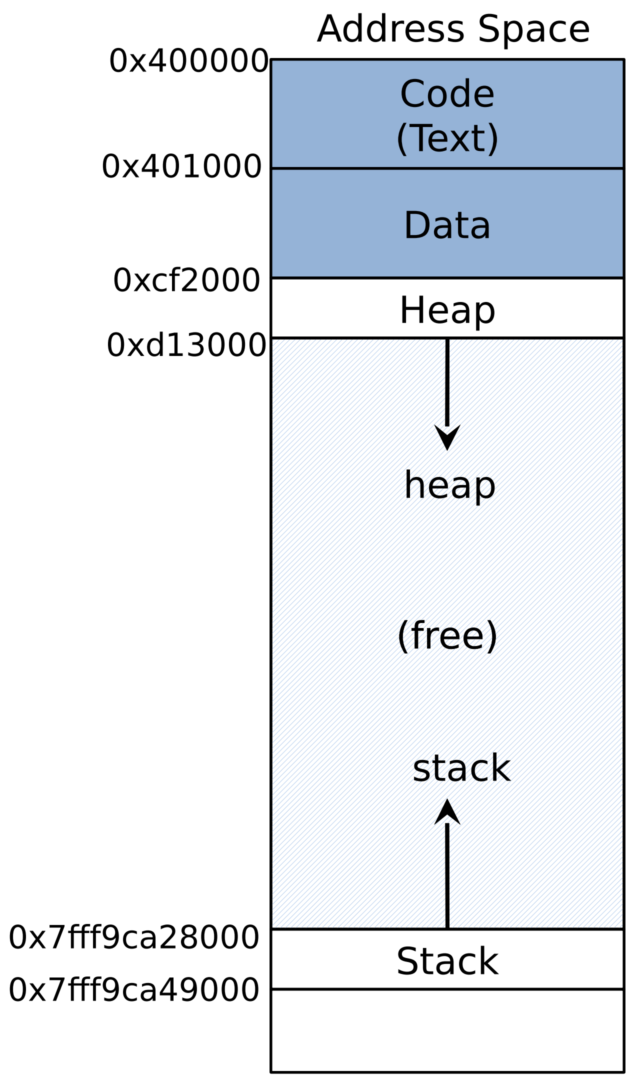
- OSes will typically let you do the following with processes
  - create
  - destroy
  - wait
  - control (e.g., suspend) and notify
  - get status, info
- Demo process queries

Process Creation

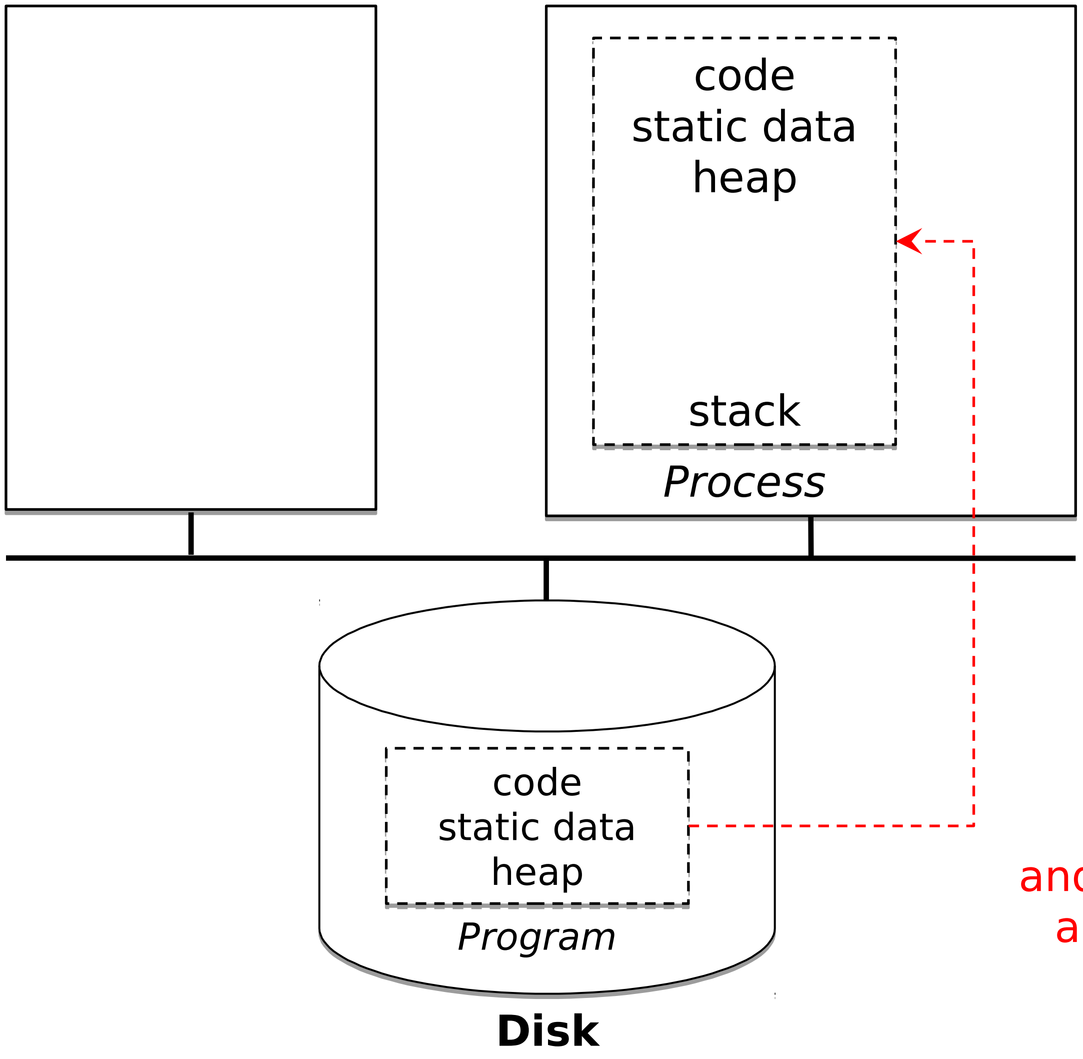
- Load code in memory
- Allocate *stack*, set initial args for main
- Set up *heap*
- Set up communication channels (open files)
- Call main

Example Modern Address Space
(64-bit Linux)

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```

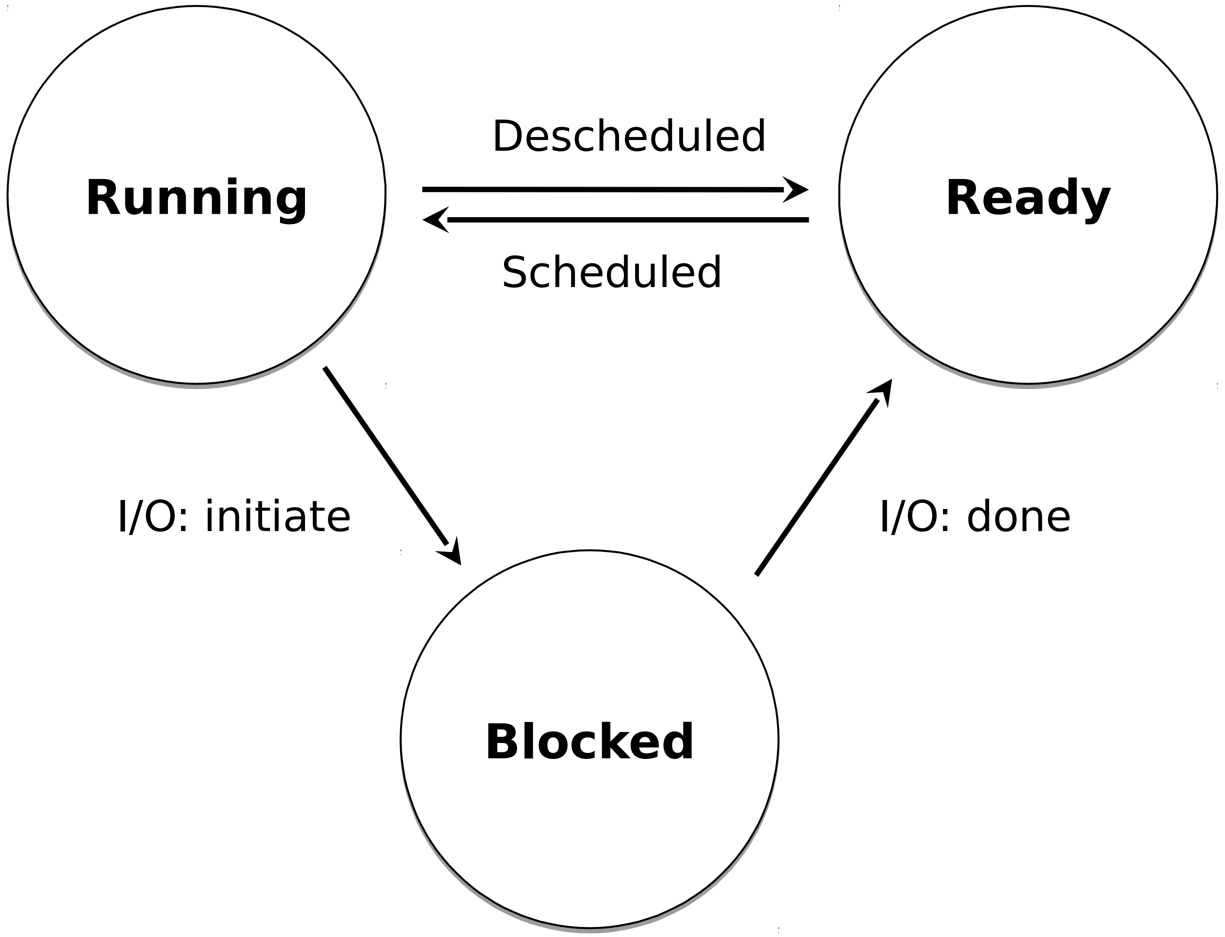


Loading



Loading:
Takes on-disk program
and reads it into the address space of process

Simplified Process States



OS Structures

- Structure holding all processes, per state
  - how do you think this looks?
- PCB (Process Control Block) per process

xv6 Kernel Structures

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;   // Index pointer register
    int esp;   // Stack pointer register
    int ebx;   // Called the base register
    int ecx;   // Called the counter register
    int edx;   // Called the data register
    int esi;   // Source index register
    int edi;   // Destination index register
    int ebp;   // Stack base pointer register
};

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

xv6 Kernel Structures

```
// 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
};
```

How Is Kernel and User Code Execution Interleaved?

- Principle of OSes: *the same physical core runs both kernel and user (process) code*
- User code runs at full CPU speed
- But kernel maintains control

How???

Example Execution

OS	**Program**
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` / `argv` 5. Clear registers 6. Execute call `main()`	
	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

- But the OS is the boss of all resources, not just a library, so how does it take back control?

System Calls, Interrupts

- User vs. Kernel CPU mode
  - not all programs can do everything
- *System calls* for all resource access
  - trap, return-from-trap instructions
  - “trap” = synchronous, user-initiated interrupt

OS @ run (kernel mode)	Hardware	Program (user mode)
initialize trap table		
	remember address of syscall handler ...	
Create process structs Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from -trap**		
	restore regs move to user mode jump to main	
		Run main() … Call system **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs move to user mode jump to PC after trap	

Is This Enough?

- What if a process is stuck in infinite loop?
  - historical note: cooperative multiprocessing
- Hardware again to the rescue!
  - OS has set a timer interrupt
  - a process only runs for a *time slice*
  - scheduling decision afterwards
  - *context switch* if needed

xv6 Context Switch Code

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
   # Save old registers
   movl 4(%esp), %eax       # put old ptr into eax
   popl 0(%eax)        # save the old IP
   movl %esp, 4(%eax)       # and stack
   movl %ebx, 8(%eax)       # and other registers
   movl %ecx, 12(%eax)
   movl %edx, 16(%eax)
   movl %esi, 20(%eax)
   movl %edi, 24(%eax)
   movl %ebp, 28(%eax)

   # Load new registers
   movl 4(%esp), %eax       # put new ptr into eax
   movl 28(%eax), %ebp      # restore other registers
   movl 24(%eax), %edi
   movl 20(%eax), %esi
   movl 16(%eax), %edx
   movl 12(%eax), %ecx
   movl 8(%eax), %ebx
   movl 4(%eax), %esp       # stack is switched here
   pushl 0(%eax)       # return addr put in place
   ret            # finally return into new ctxt
```

What If Interrupted During Interrupt Handling?

- OS can briefly disable interrupts
- ... or ensures safe access to data structures through concurrency control mechanisms (e.g., locking)