5118020-03 Operating Systems

Process

OSTEP Chapters 4 & 6

Shin Hong

Motivation

Process

- a running instance of a program
  - program vs. process
  - a kernel object that contains all information and resources given to the running instance
    - identified by a unique number (process ID)

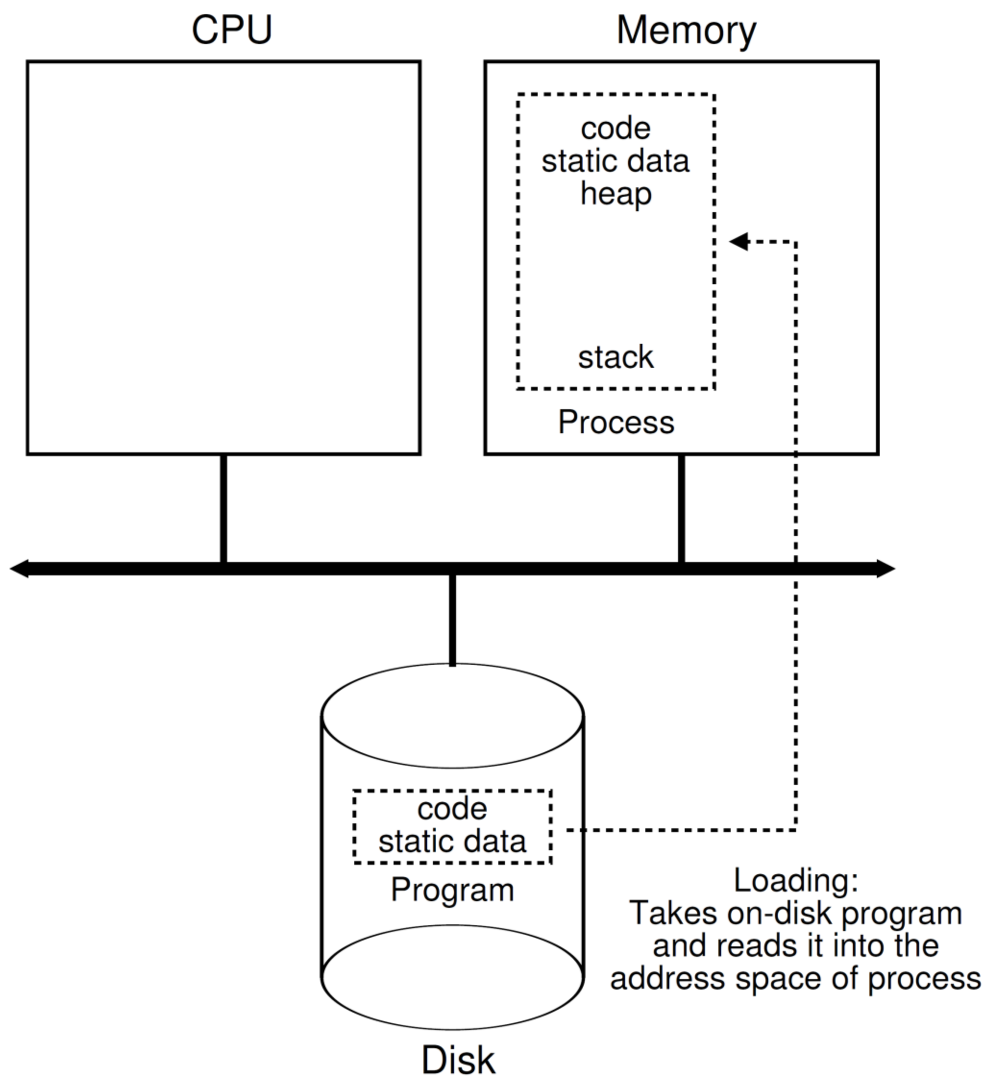
- time-sharing of a CPU provides the illusion of many CPUs
  - concurrency vs. parallelism
  - policies: scheduling
  - mechanism: context switching

Constitution of Program Execution Context

- memory states
  - address space
- CPU states
  - registers: general-purpose and special-purpose
- I/O information

Life Cycle of a Process

- process creation
  - resource allocation
  - loading
    - eager manner
    - lazy manner



Process Creation

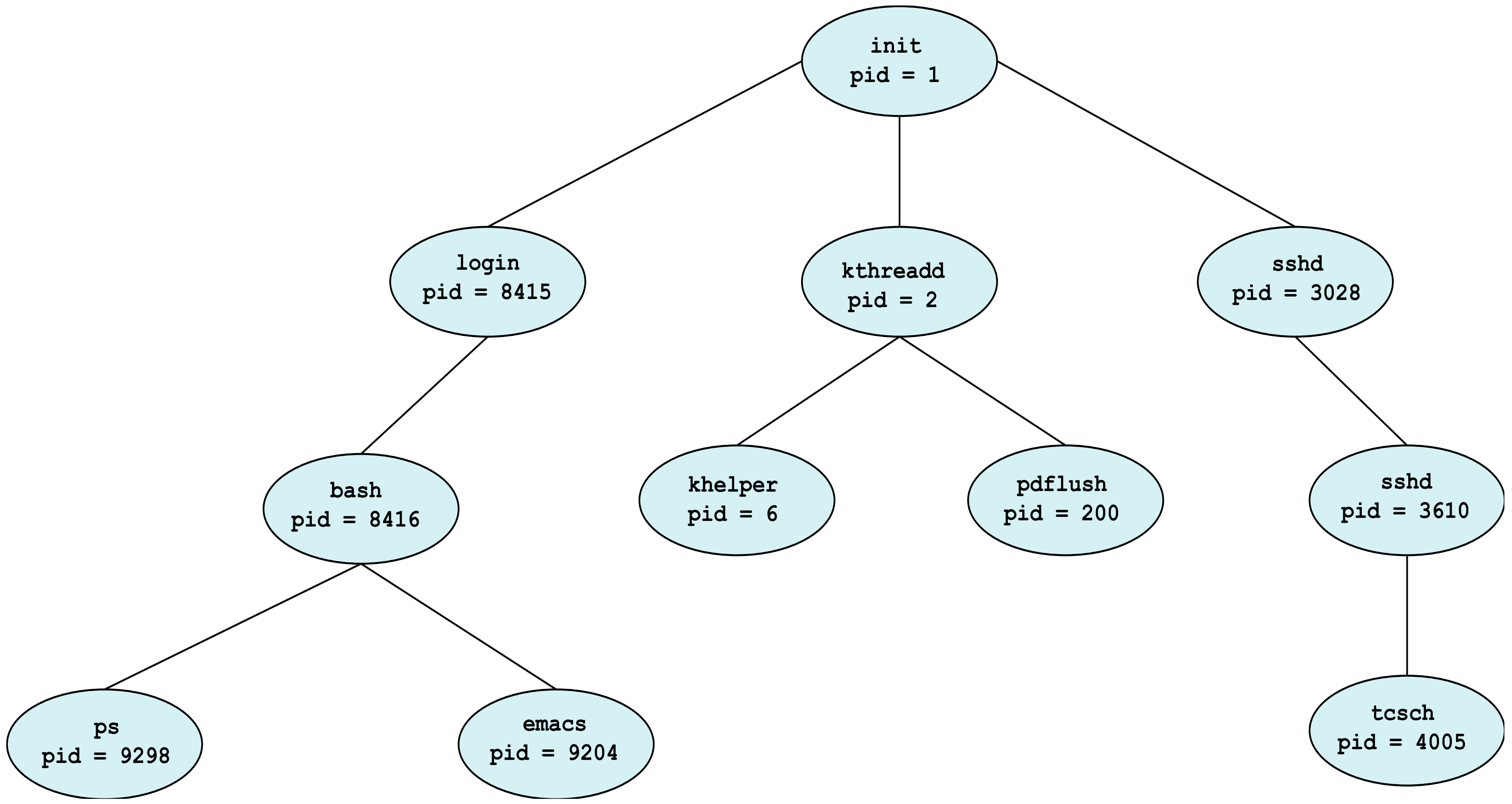
- A process is identified and managed via a process identifier (pid)

- A parent process can spawn a child process to delegate a subtask
  - A process can spawn multiple children processes
  - A parent process can run concurrently with its children processes
  - A child process, in turn create other processes, forming a tree of processes
  - A parent can wait until a child (or children) terminates

- A parent and its children can share resources
  - Children may share a subset of parent's resources

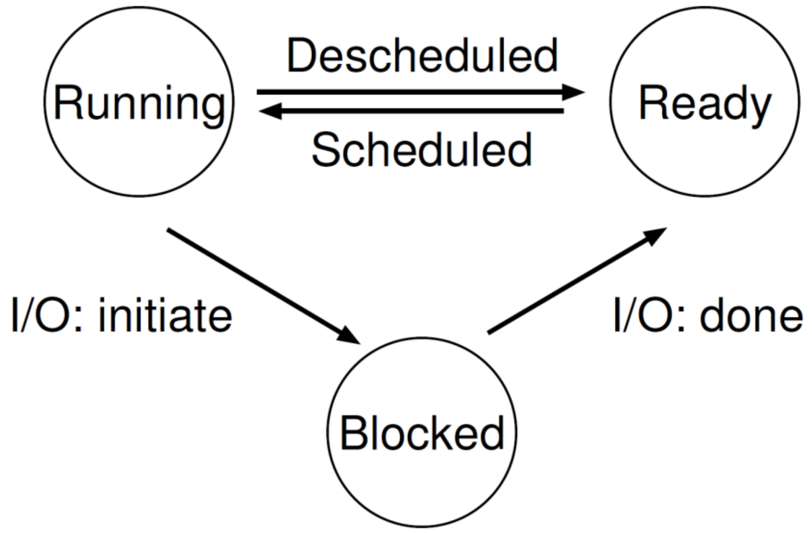
- Process in UNIX
  - a system call **`fork()`** system call creates a new process
  - a child process duplicates the memory of its parent

A Tree of Processes in Linux



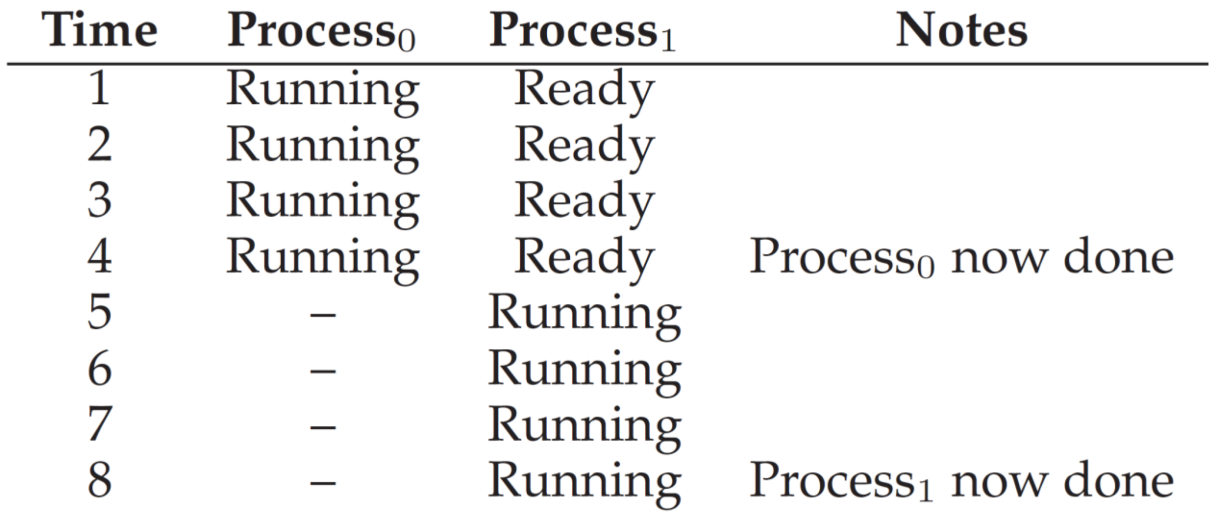
Life Cycle of a Process

- Running state
  - hold a CPU and execute instructions
- Ready state
  - can make a progress, but do not hold a CPU
- Blocked state
  - cannot make a progress since it wait for a certain event/condition (i.e., I/O)

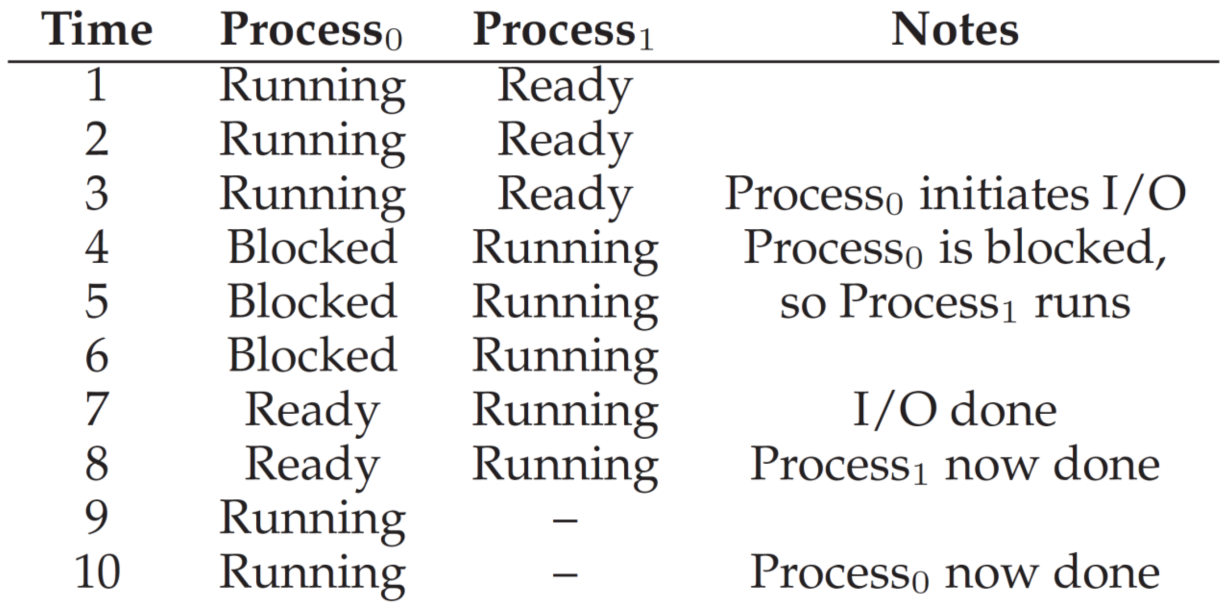


- CPU scheduler decides process state transitions

Example



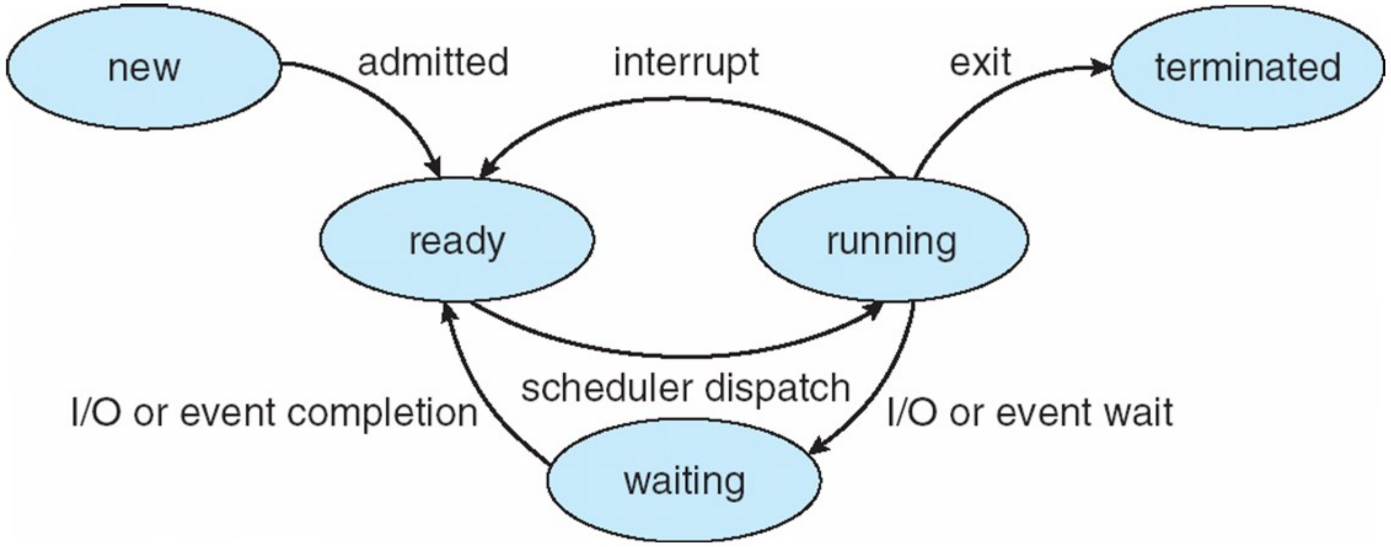
Time	Process$_0$	Process$_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process$_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process$_1$ now done



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

Another Representation of Process Life Cycle

- As a process executes, the process state changes
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution



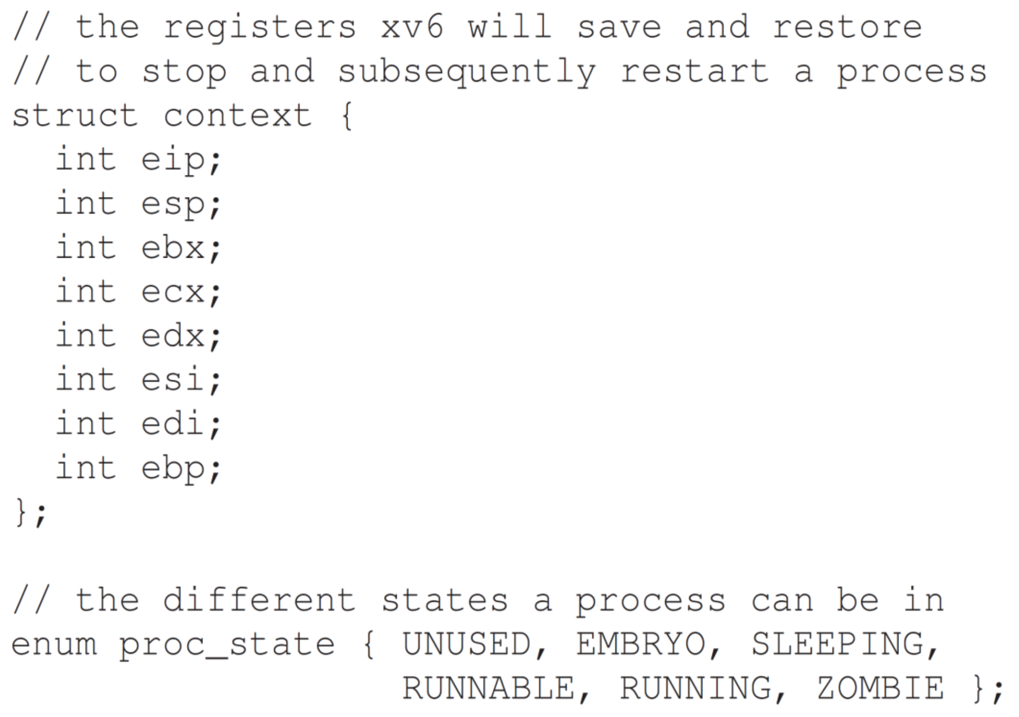
Process Termination

- When it finishes the given task, a process invokes the **exit** system call to delete itself
  - Given resources are reclaimed (deallocated)
  - Returns the exit status to the parent process (via `wait()`)

- A parent process can cut off a child process with **abort** system call
  - Examples
    - when a child exhausts all resources or time
    - when the task given to a child is not needed anymore
    - when the parent quits

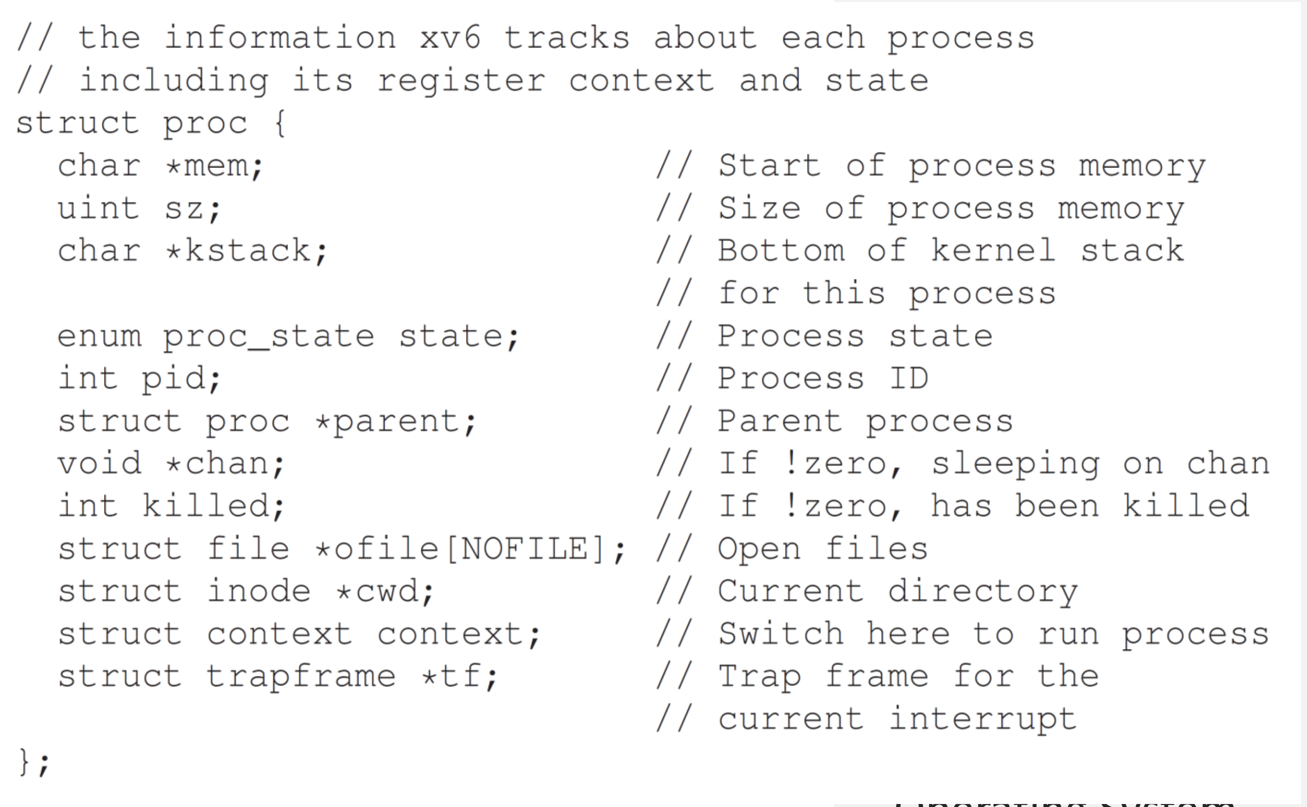
Process Data Structure in Kernel: xv6 Example

- Called as Process Control Block
- Example of the xv6 kernel



```
// 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 !zero, sleeping on chan
  int killed;                 // If !zero, has 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
};
```

Remainig Issues

- What’s the overhead for virtualizing CPU?
- How to implement context switching?
- How to manage multiple processes?
- How a scheduler determines the next process to run?

Mechanisms for CPU Virtualization

- time-sharing
  - run one process for a while, and then switch to another one
  - issue
    - managing control
    - obtaining performance

- limited direct execution
  - run an application program directly on the CPU with some restriction
  - restriction
    - restricted memory accesses (H/W manipulation, resource allocation, etc.)
    - restricted instruction

Dual Mode Operation: User Mode & Kernel Mode

- Most contemporary processors provide dual mode operation

- User mode
    - for application program execution
    - restriction is enforced
        - a trap occurs when a process executes a restricted instruction under user mode

- Kernel model
    - for kernel execution
    - all privileged operations can be executed

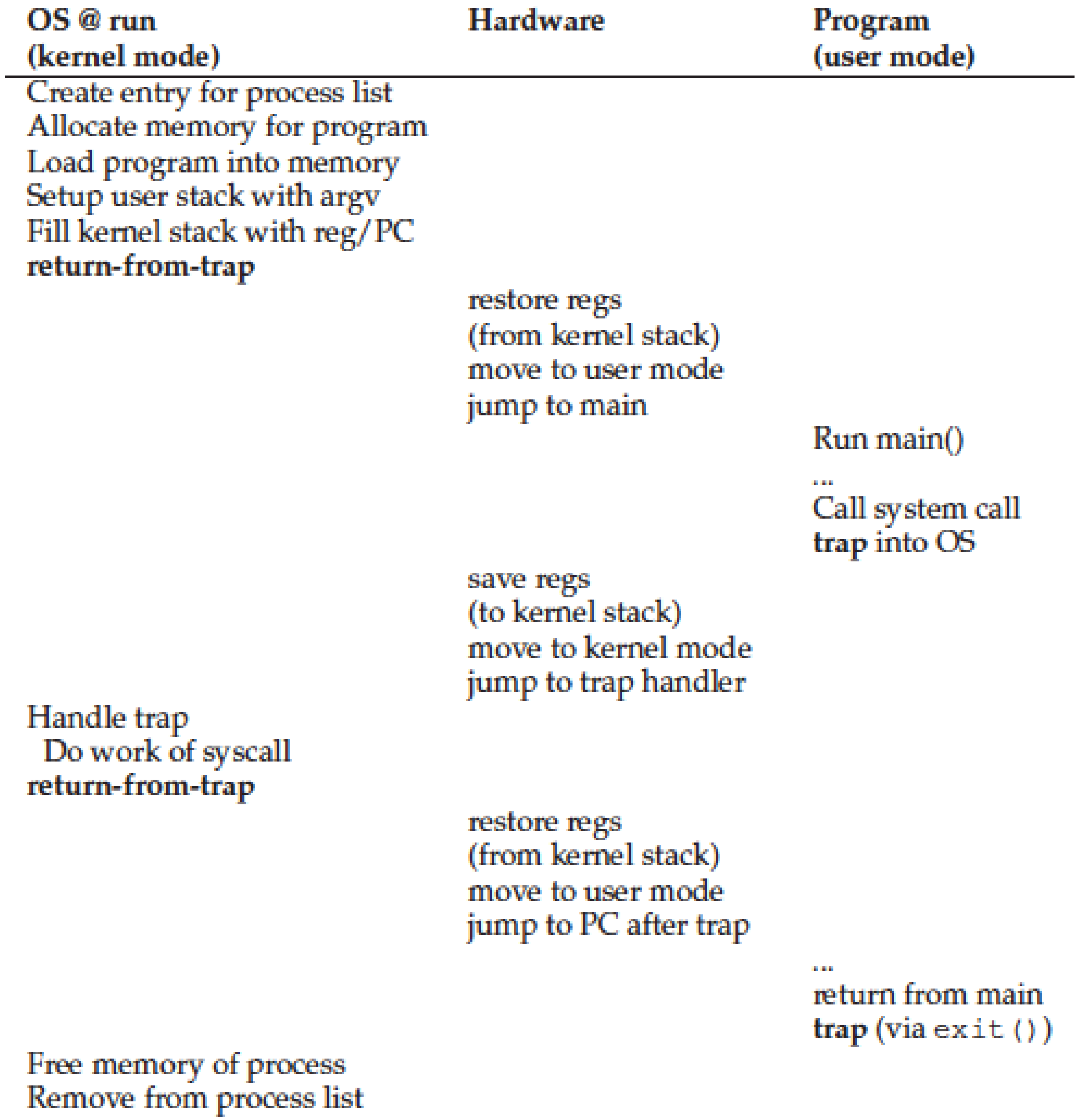
- What if an application program needs to execute privileged operations?

System Call

- A mechanism for an application program to call the kernel to run to provide expected services
    - not possible to do this with the function-call mechanism

- To execute a system call, a program must execute a trap instruction
  - the operation of a trap instruction
    - change the mode into the kernel mode
    - store the current PC at a kernel stack
    - jump to a predefined program location for handling the trap
      - trap table
      - trap handler
  - each system call is identified by its unique number (i.e. system-call number)

Example of System Call Workflow



OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to main	
		Run main() ... Call system call **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 (from kernel stack) move to user mode jump to PC after trap	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

Switching Between Processes

- How can OS regain control of the CPU when it is given to an application program?
- Natural chance: blocked operation
- Periodic scheduling
  - Cooperative approach
    - blocked operation
  - Preemptive scheduling approach
    - exploit a timer interrupt and its interrupt handler
    - requires HW support

Preemptive Scheduling

- scheduler: a kernel module that determines which process to dispatch at a chance (e.g., timer interrupt)

- context switch: a kernel module that is executed to switch processes running on a CPU
  - registered as a timer interrupt handler
  - steps
    - store the process status of a currently-running process to memory
      - CPU states
    - find the stored status of the next process from the memory
    - restore the stored status at the CPU
    - return the control back to the application program

Example

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) $\rightarrow$ k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) $\rightarrow$ proc_t(A)		
restore regs(B) $\leftarrow$ proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) $\leftarrow$ k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

The 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
```