ITP 30002 Operating System

Process

OSTEP Chapter 4

Shin Hong

Motivation

Process

- a running instance of a program
  - program vs. process

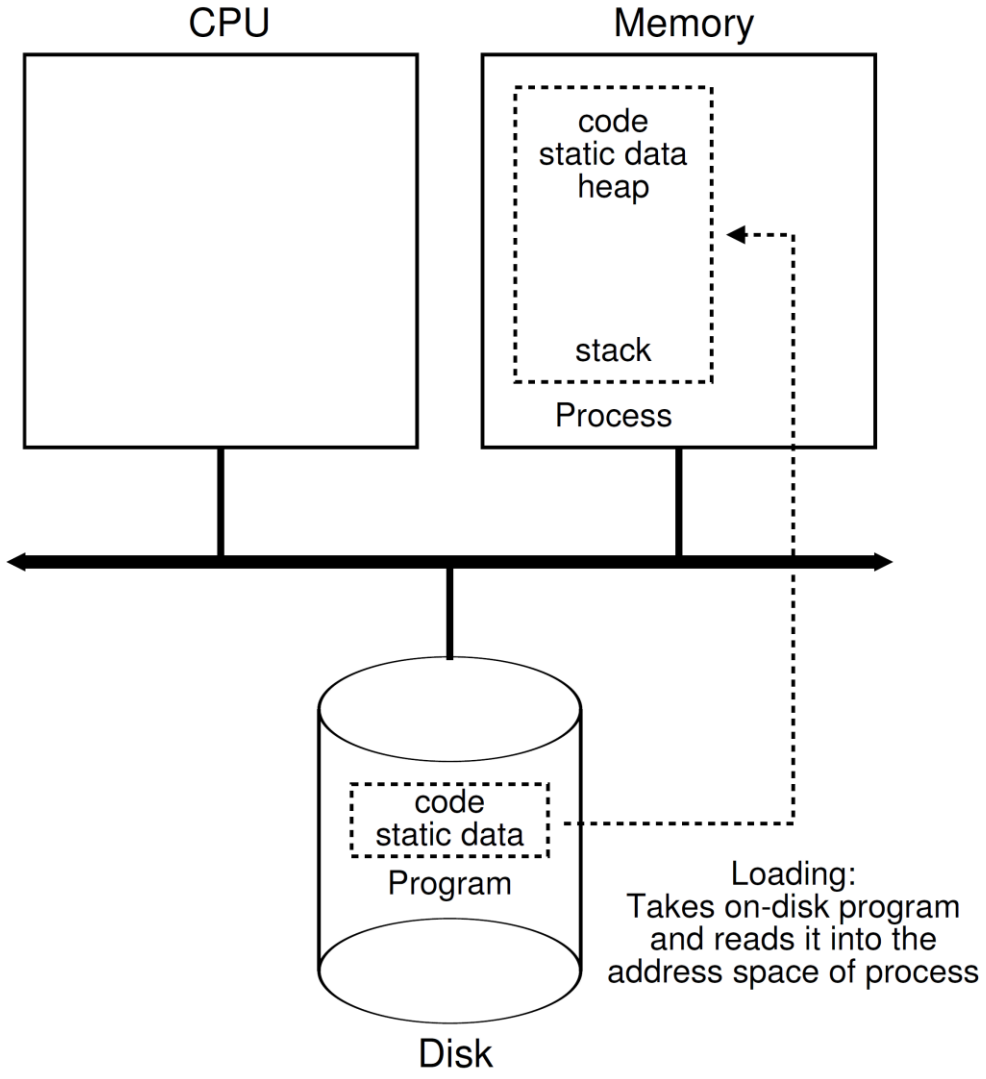
- time-sharing of a CPU provides the illusion of many CPUs
  - concurrency vs. parallelism
  - mechanism: context switching
  - policies: scheduling

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
    - eagle manner
    - lazy manner



Process Creation

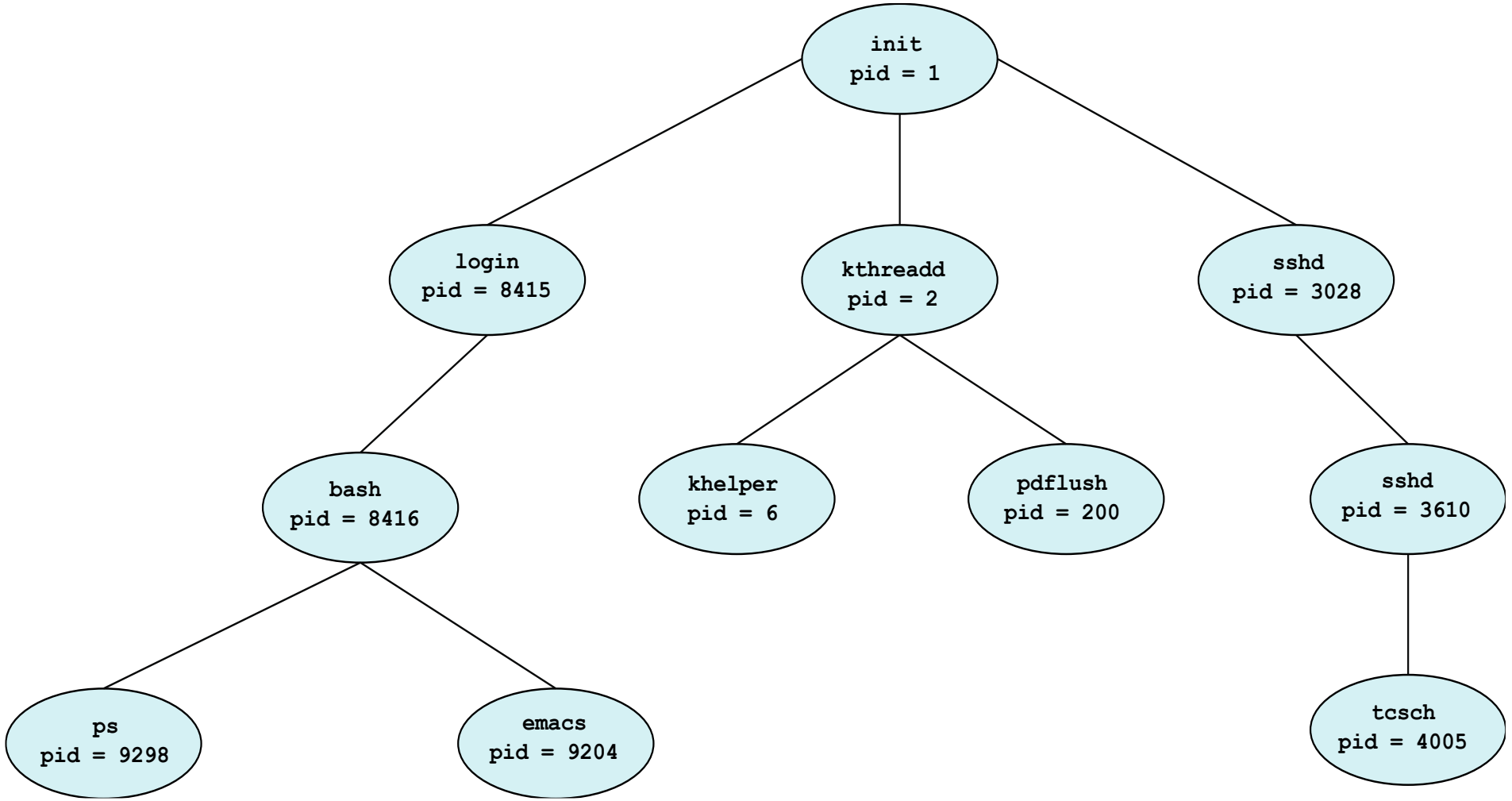
- A process is identified and managed via a process identifier (pid)

- 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

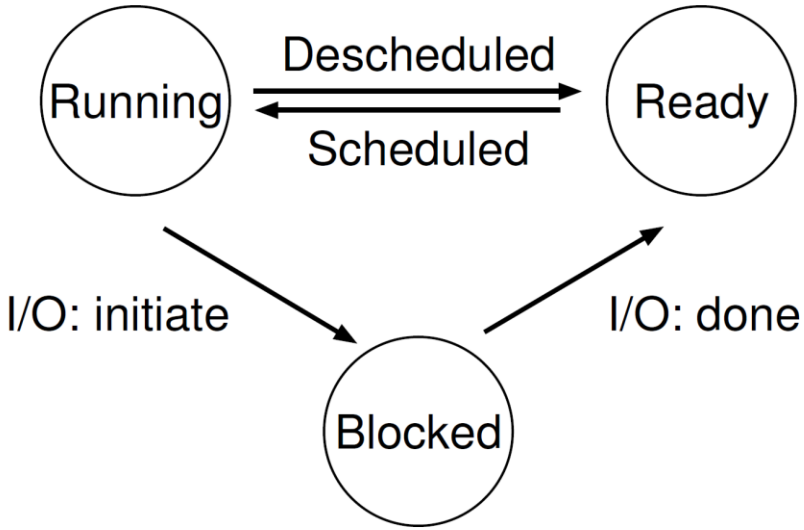
- Process in UNIX
  - a system call **`fork()`** system call creates a new process
  - a child process duplicates the memory of its parent

A Tree of Processes in Linux



Life Cycle of a Process

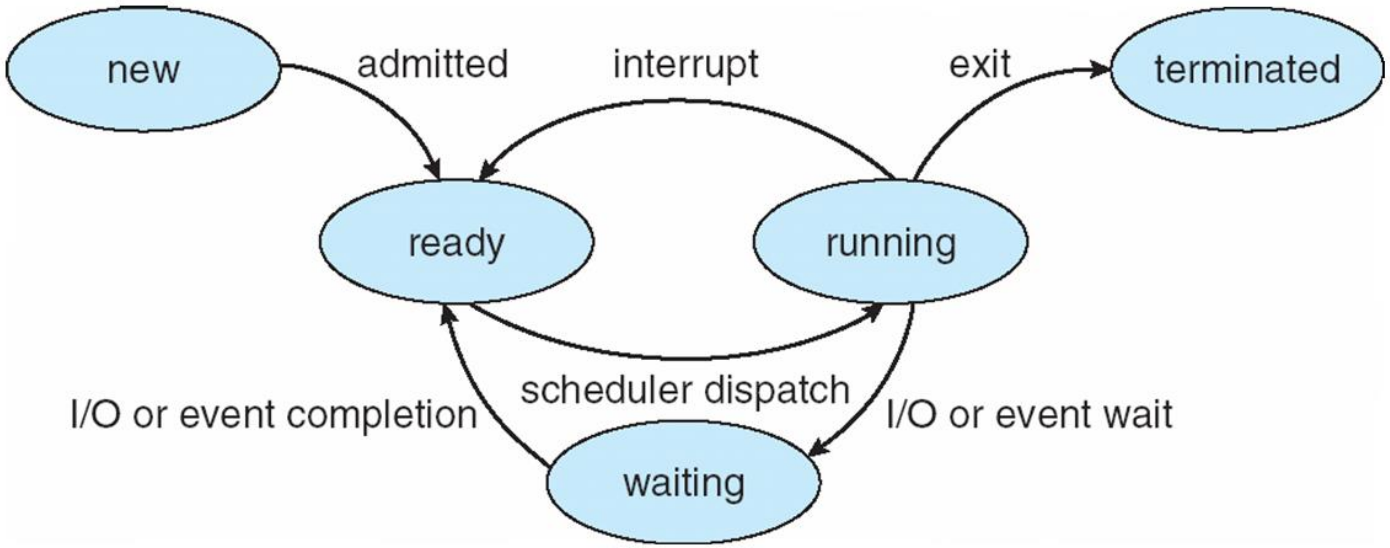
- Running state
  - hold a CPU and execute instructions
- Ready state
  - can make a progress, but cannot hold a CPU
- Blocked state
  - cannot make a progress since it needs to wait for a certain condition (i.e., I/O)



- CPU scheduler makes a decision for process state transitions

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

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system

- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

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

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 a next process to execute?