W4118 Operating Systems

Junfeng Yang

Outline

- What is a process?
- Process dispatching
- Common process operations
- Inter-process Communication

What is a process

- "Program in execution" "virtual CPU"
- Process: an execution stream in the context of a particular process state
- Execution stream: a stream of instructions
  - Running piece of code
  - *sequential sequence of instructions*
- Process state: determines the effect of running code
  - Stuff the running code can affect or be affected by

Process state

- Registers
  - General purpose, floating point, instruction pointer (program counter) …
- Memory: everything a process can address
  - Code, data, stack, heap, …
- I/O status:
  - File descriptor table, …
- …

Program v.s. process

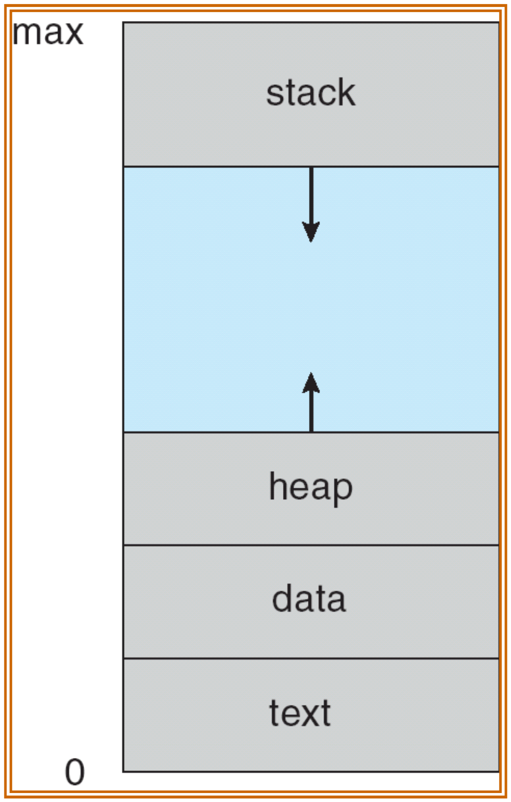
- Program != process
  - Program: static code + static data
  - Process: dynamic instantiation of code + data + more

- Program ⇔ process: no 1:1 mapping
  - Process > program: more than code and data
  - Program > process: one program runs many processes
  - Process > program: many processes of same program

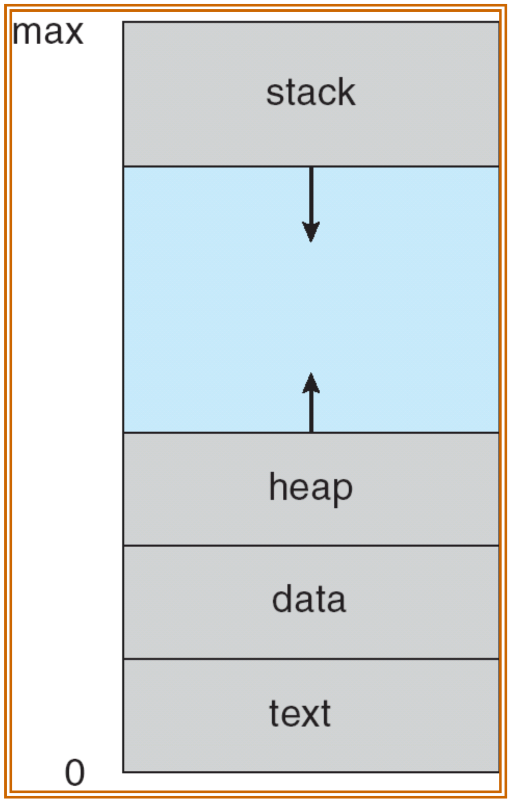
Address Space

- Address Space (AS): all memory a process can address
  - Really large memory to use
  - Linear array of bytes: [0, N), N roughly 2^32, 2^64
- Process ⬄ address space: 1 : 1 mapping
- Key: an AS is a protection domain
  - OS isolates address spaces
  - One process can't access another process's address space
  - Same pointer address in different processes point to different memory

Address space examples



Process A



Process B

Process v.s. Thread

- Thread: separate streams of execution that share the same address space
- Process != Thread
  - One process can have multiple threads
  - Threads communicate more efficiently
- More on thread later

Why use processes?

- General principle of divide and conquer
  - Decompose a large problem into smaller ones → easier to think well contained smaller problems
- Systems have many concurrent jobs going on
  - E.g. Multiple users running multiple shells, I/O, …
  - OS must manage
- Easier to reason about processes than threads
  - Sequential activities with well defined interactions

Outline

- What is a process?
- Process dispatching
- Common process operations
- Inter-process Communication

System categorization

- Uniprogramming: one process at a time
  - Eg., early main frame systems, MSDOS
  - Good: simple
  - Bad: poor resource utilization, inconvenient for users

- Multiprogramming: multiple processes, when one waits, switch to another
  - E.g, modern OS
  - Good: increase resource utilization and user convenience
  - Bad: complex

  - Note: multiprogramming != multiprocessing

Multiprogramming

- OS requirements for multiprogramming
  - Scheduling: what process to run? (later)
  - Dispatching: how to switch process? (today)
  - Memory protection: how to protect process from one another? (later)
- Separation of policy and mechanism
  - Recurring theme in OS
  - Policy: decision making with some performance metric and workload (scheduling)
  - Mechanism: low-level code to implement decisions (dispatching)

Process dispatching mechanism

OS dispatching loop:

```
while(1) {
    run process for a while;
    save process state;
    next process = schedule (ready processes);
    load next process state;
}
```

Q1: how to gain control?

Q3: where to find processes?

Q2: what state must be saved?

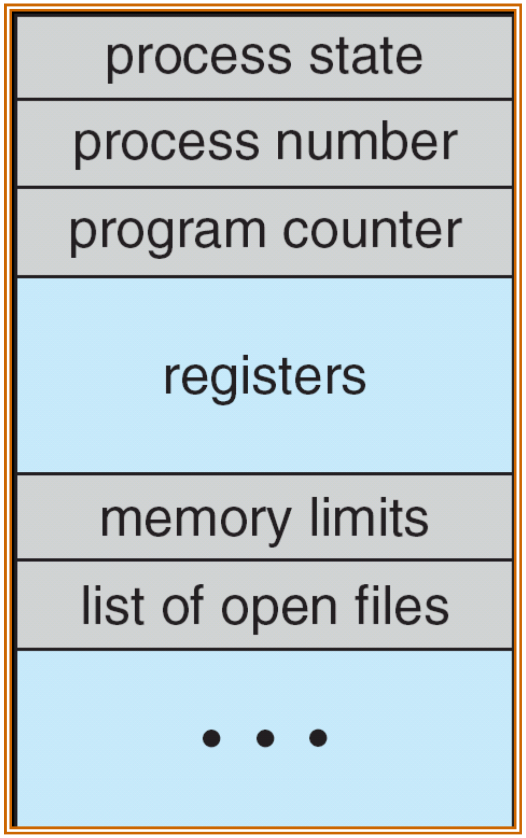
Q1: How does Dispatcher gain control?

- Must switch from user mode to kernel mode
- Cooperative multitasking: processes voluntarily yield control back to OS
  - When: system calls that relinquish CPU
  - Why bad: OS trusts user processes!
- True multitasking: OS preempts processes by periodic alarms
  - Processes are assigned time slices
  - Dispatcher counts timer interrupts before context switch
  - Why good: OS trusts no one!

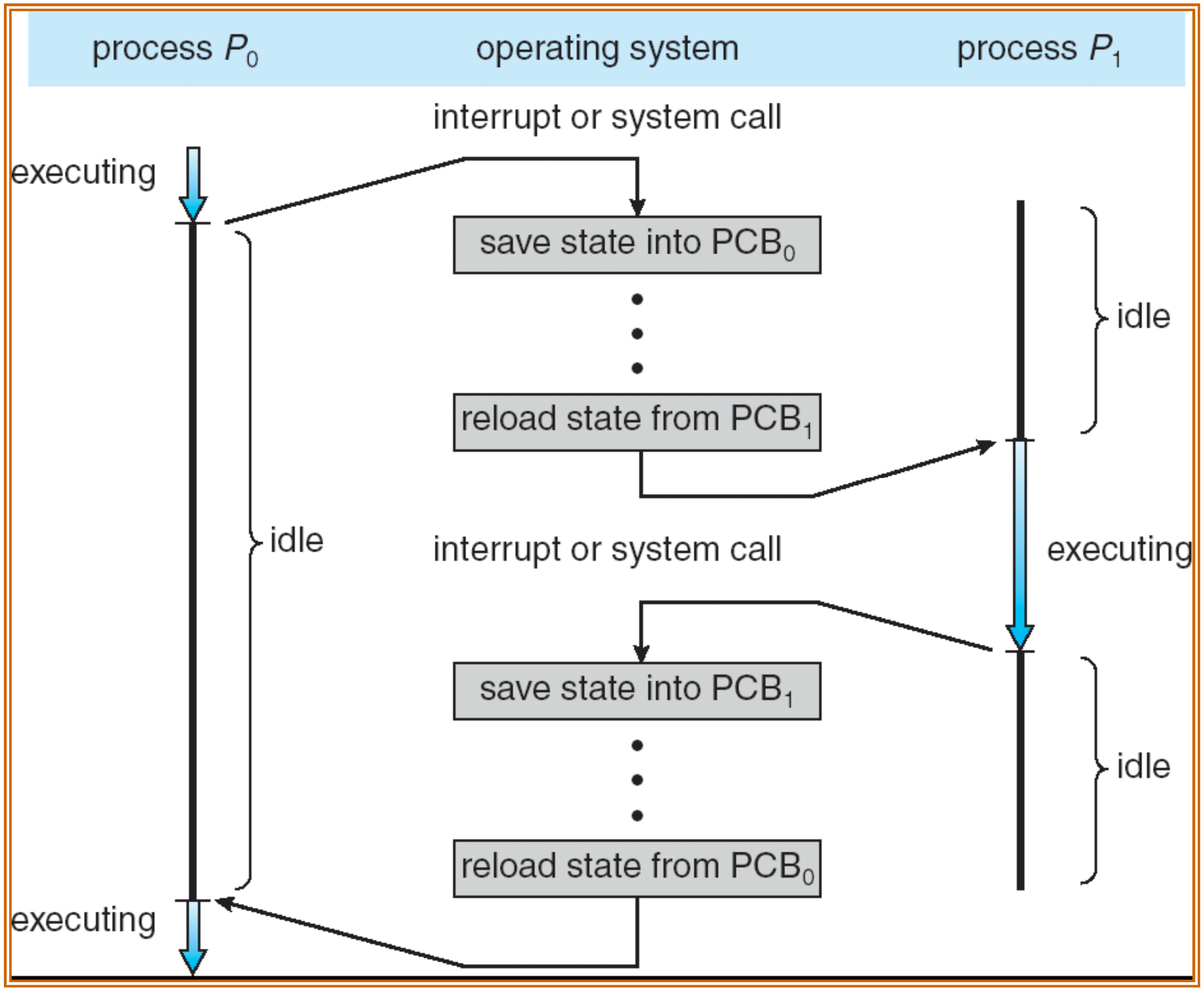
Q2: What state must be saved?

- Dispatcher stores process state in Process Control Block (PCB)

- What goes into PCB?
  - Process state (running, ready …)
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information



CPU Switch From Process to Process



Context switch

- Implementation: machine dependent
  - Tricky: OS must save state w/o changing state !
    - Need to save all registers to PCB in memory
    - Run code to save registers, but code changes registers
  - Solution: hardware support
- Performance?
  - Can take long. A lot of stuff to save and restore. The time needed is hardware dependent
  - Context switch time is pure overhead: the system does no useful work while switching
  - Must balance context switch frequency with scheduling requirement

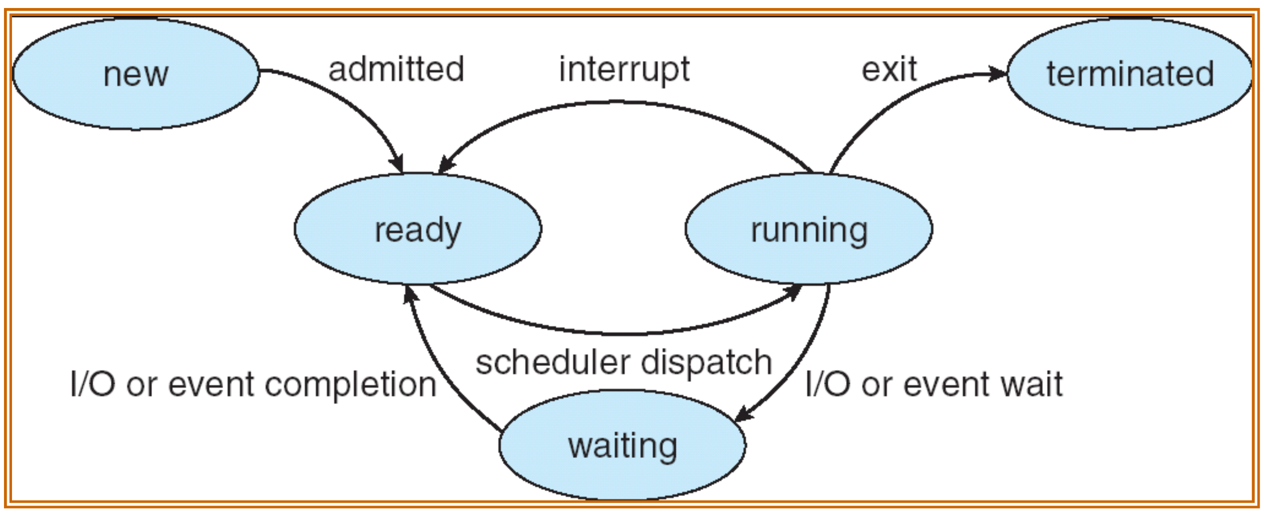
Q3: where to find processes?

- Data structure: process scheduling queues
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device

- Processes migrate among the various queues when their states change

Process state diagram

- Process state
  - New: being created
  - Ready: waiting to be assigned a CPU
  - Running: instructions are running on CPU
  - Waiting: waiting for some event (e.g. IO)
  - Terminated: finished



Outline

- What is a process?
- Process dispatching
- Common process operations
- Inter-process Communication

Process creation

- Option 1: from scratch (e.g, Win32 CreateProcess())
  - Load code and data into memory
  - Create and initialize PCB (make it like saved from context switch)
  - Add new PCB to ready queue
- Option 2: cloning (e.g., Unix fork(), exec())
  - Pause current process and save its state
  - Copy its PCB (can select what to copy)
  - Add new PCB to ready queue
  - Anything else?
    - Must distinguish parent and child

Process termination

- Normal: exit(int status)
  - OS passes exit status to parent via wait(int *status)
  - OS frees process resources

- Abnormal: kill(pid_t pid, int sig)
  - OS can kill process
  - Process can kill process

Zombie and orphan

- What if child exits before parent?
  - Child becomes zombie
    - Need to store exit status
    - OS can't fully free
  - Parent must call wait() to reap child

- What if parent exits before child?
  - Child becomes orphan
    - Need some process to query exit status
  - Re-parent to process 1, the init process

Outline

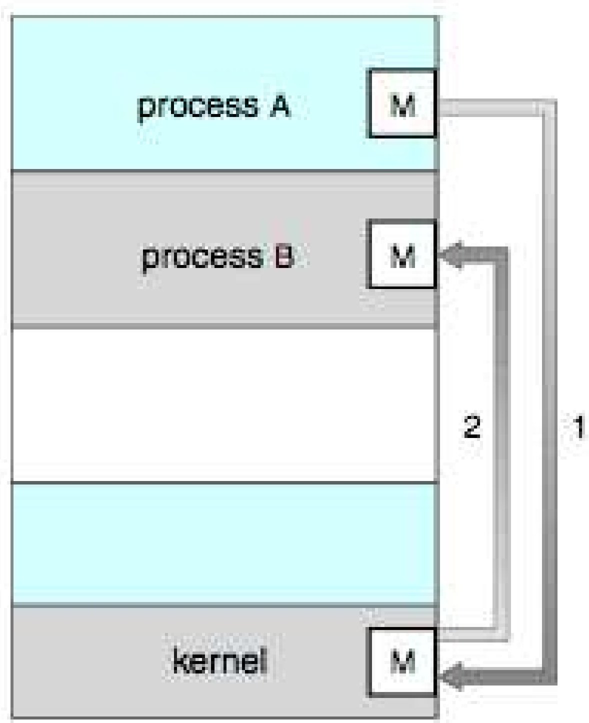
- What is a process?
- Process dispatching
- Common process operations
- Inter-process Communication

Cooperating Processes

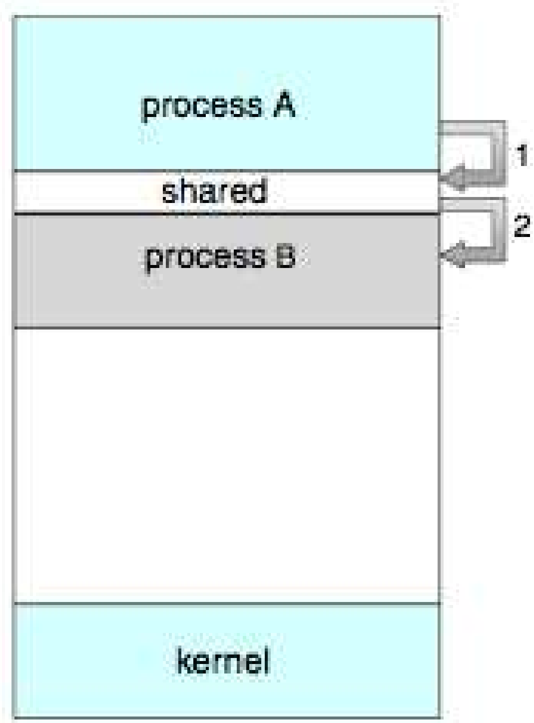
- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity/Convenience

Interprocess Communication Models

Message Passing



Shared Memory



Message Passing v.s. Shared Memory

- Message passing
  - Why good? All sharing is explicit ➔ less chance for error
  - Why bad? Overhead. Data copying, cross protection domains
- Shared Memory
  - Why good? Performance. Set up shared memory once, then access w/o crossing protection domains
  - Why bad? Things change behind your back ➔ error prone

IPC Example: Unix signals

- Signals
  - A very short message: just a small integer
  - A fixed set of available signals. Examples:
    - 9: kill
    - 11: segmentation fault
- Installing a handler for a signal
  - sighandler_t signal(int signum, sighandler_t handler);
- Send a signal to a process
  - kill(pid_t pid, int sig)

IPC Example: Unix pipe

- int pipe(int fds[2])
  - Creates a one way communication channel
  - fds[2] is used to return two file descriptors
  - Bytes written to fds[1] will be read from fds[0]

```
int pipefd[2];
pipe(pipefd);
     switch(pid=fork()) {
     case -1: perror("fork"); exit(1);
     case 0:  close(pipefd[0]);
              // write to fd 1
              break;
     default:  close(pipefd[1]);
              // read from fd 0
        break;
}
```

IPC Example: Unix Shared Memory

- int shmget(key_t key, size_t size, int shmflg);
  - Create a shared memory segment
  - key: unique identifier of a shared memory segment, or IPC_PRIVATE
- int shmat(int shmid, const void *addr, int flg)
  - Attach shared memory segment to address space of the calling process
  - shmid: id returned by shmget()
- int shmdt(const void *shmaddr);
  - Detach from shared memory
- Problem: synchronization! (later)

Next lecture

- Process in Linux