W4118: Process and Address Space

Junfeng Yang

References: Modern Operating Systems (3rd edition), Operating Systems Concepts (8th edition), previous W4118, and OS at MIT, Stanford, and UWisc

Outline

- Process
- Address space
- Process dispatch
- Common process operations

What is a process

- Process: an execution stream in the context of a particular process state
  - "Program in execution" "virtual CPU"

- Execution stream: a stream of instructions

- Process state: determines effect of running code
  - 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

Why use processes?

- Express concurrency
  - Systems have many concurrent jobs going on
    - E.g. Multiple users running multiple shells, I/O, …
  - OS must manage
- General principle of divide and conquer
  - Decompose a large problem into smaller ones → easier to think well contained smaller problems
- Isolated from each other
  - Sequential with well defined interactions

Process management

- Process control block (PCB)
  - Process state (new, ready, running, waiting, finish …)
  - CPU registers (e.g., %eip, %eax)
  - Scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

- OS often puts PCBs on various queues
  - Queue of all processes
  - Ready queue
  - Wait queue

Outline

- Process
- Address space
- Process dispatch
- Common process operations

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

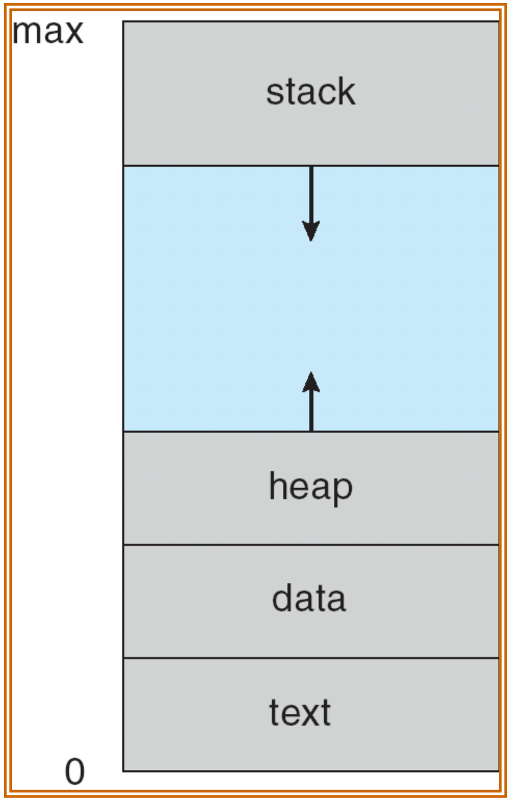
- OS requirements for multiprogramming
  - Scheduling: what process to run? (later)
  - Dispatching: how to switch? (today + later)
  - Memory protection: how to protect from one another? (today + later)

- 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, protection)

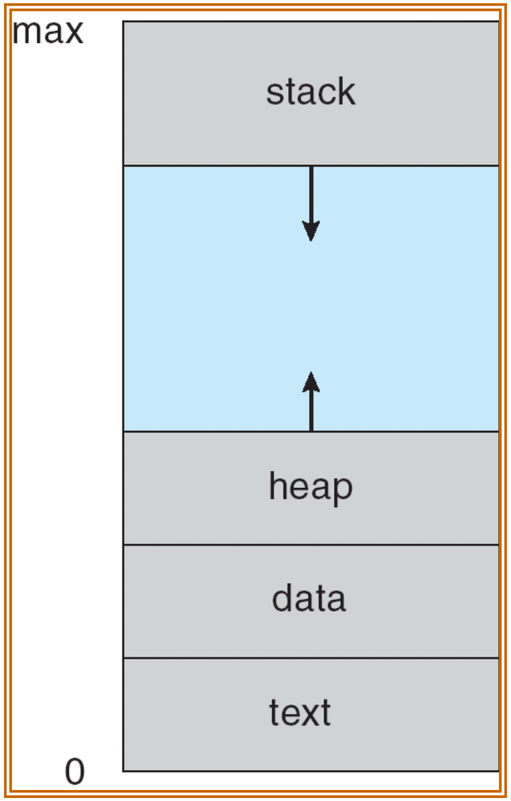
Address Space

- 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
- Address space = protection domain
  - OS isolates address spaces
  - One process can't access another's address space
  - Same pointer address in different processes point to different memory

Address space examples



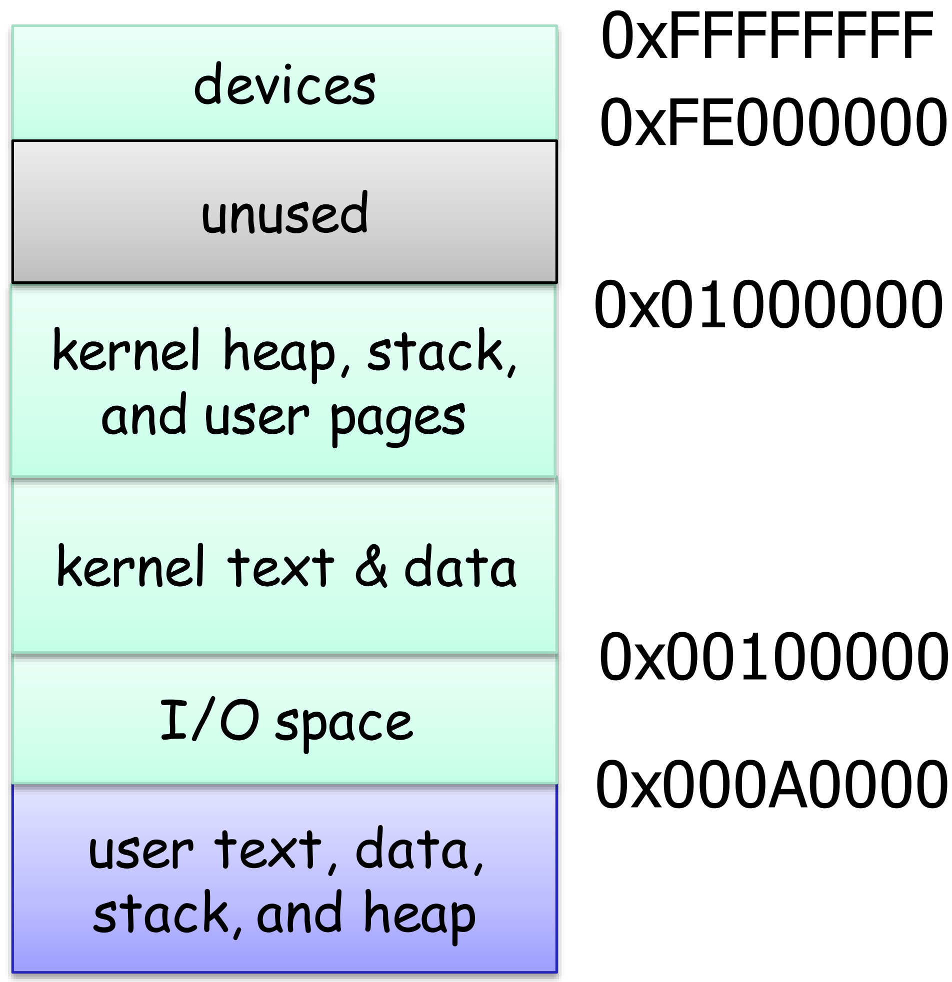
Process A



Process B

xv6 address space

- Split into kernel space and user space
- User: 0-640KB
  - User text, data, stack, and heap
- Kernel: 640KB – 4GB
  - Direct (virtual = physical)
- Real world
  - Also split
  - User space much bigger
    - Linux: 3GB, 1GB



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?

Q2: how to switch context?

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!

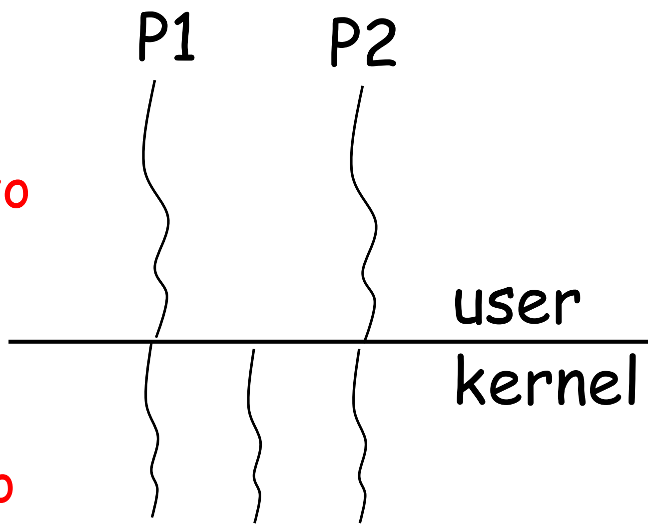
Q2: how to switch context?

- 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? Code changes registers
  - Solution: software + hardware

- 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

xv6 context switch

- Save P1's user-mode CPU context and switch from user to kernel mode (hw)
- Handle system call or interrupt (os)
- Save P1's kernel CPU context and switch to scheduler CPU context (os + hw)
- Select another process P2 (os)
- Switch to P2's address space (os + hw)
- Save scheduler CPU context and switch to P2's kernel CPU context (os + hw)
- Switch from kernel to user mode and load P2's user-mode CPU context (hw)



- swtch.S

Outline

- What is a process?
- Address space
- Process dispatch
- Common process operations

Process creation

- Option 1: 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
  - Must distinguish parent and child
- Option 2: from scratch (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

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

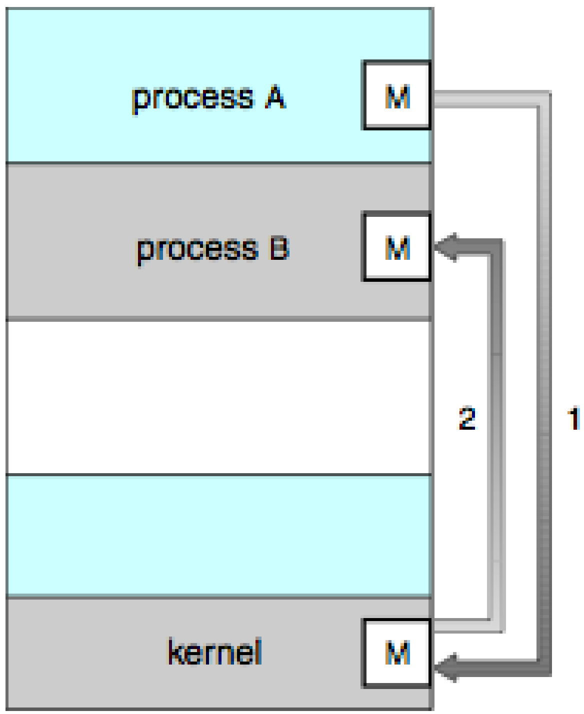
- What if parent exits before child?
  - Child becomes orphan
    - Need some process to query exit status and maintain process tree
  - Re-parent to the first process, the init process

Cooperating Processes

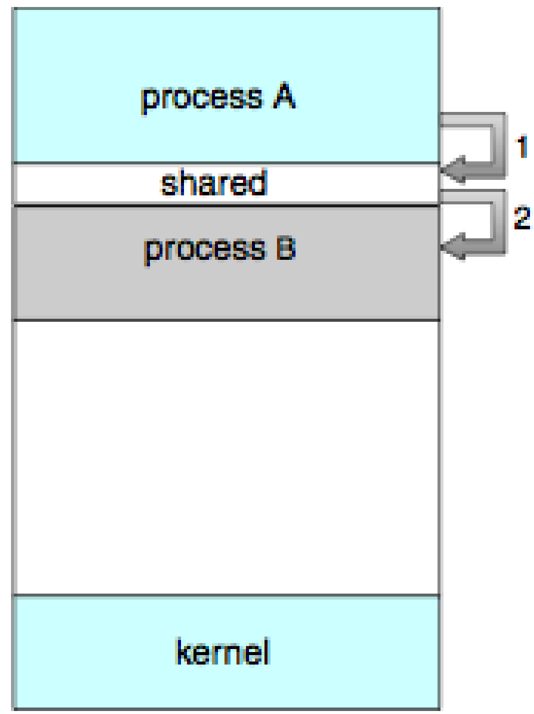
- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity/Convenience

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

- Memory management