Instituto de Matemática e Estatística

USP

MAC422
Sistemas Operacionais

Prof. Marcel P. Jackowski

Aula #3

Processos, contextos e espaços de endereçamento

Referências: Sistemas Operacionais Modernos (3a. Ed.), Fundamentos de Sistemas Operacionais; aulas antigas de MAC-422 (IME-USP), W4118 (Columbia University), 6.828 (MIT)

O que é um processo ?

- 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

Programa vs. processo

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

- Program ⇔ process: no 1:1 mapping

Por que processos ?

- 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

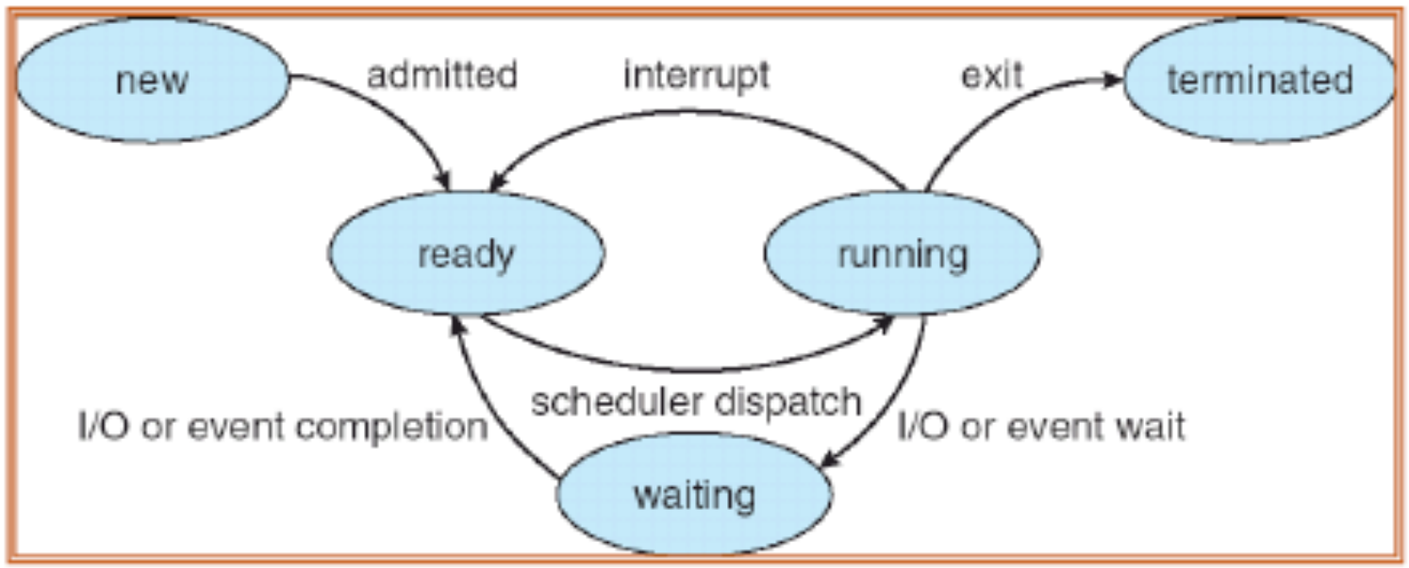
- Isolated from each other
  - Sequential with well defined interactions

Gerenciamento de processos

- 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

Estados de um processo

- 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



Categorização

- 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

Multiprogramação

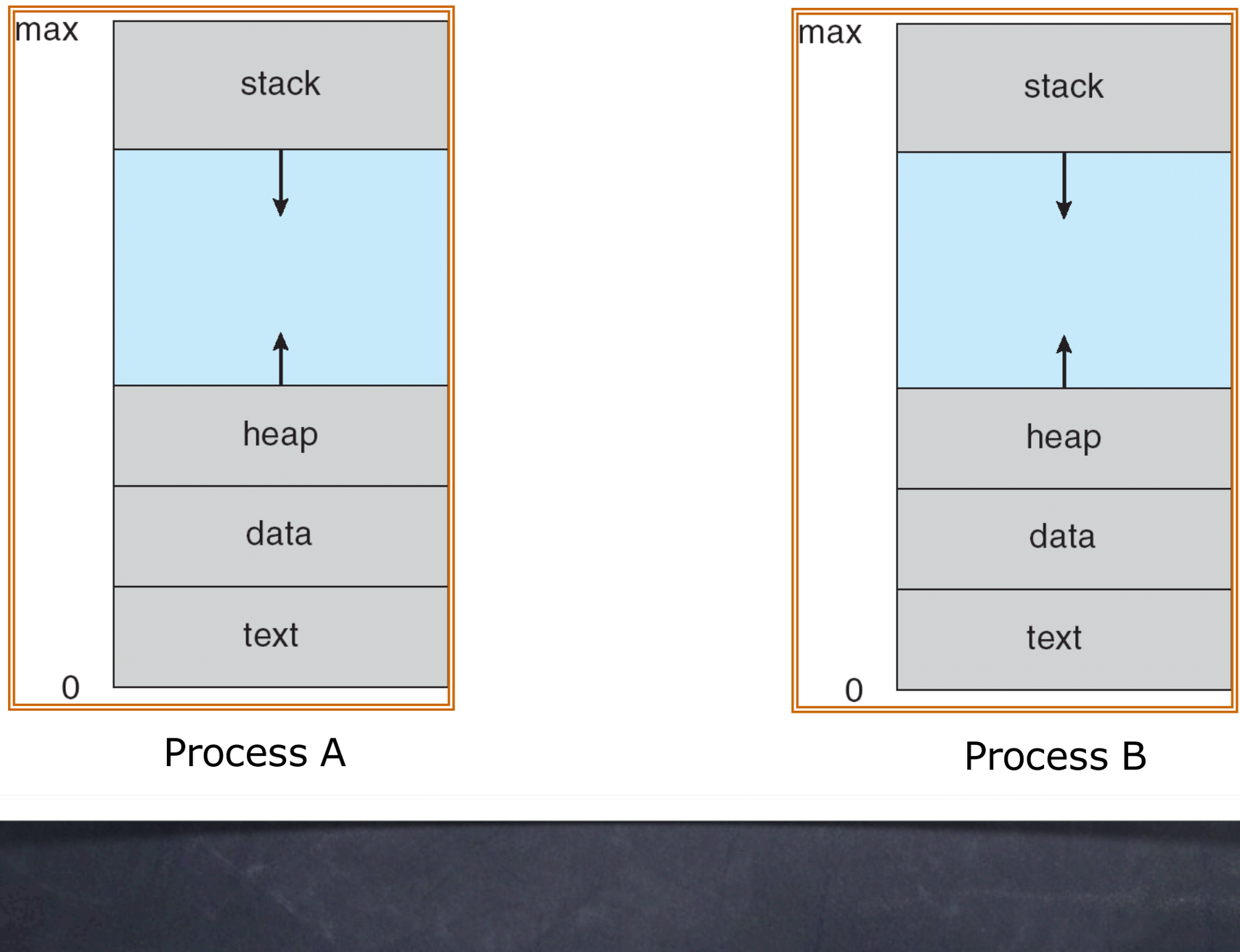
- OS requirements for multiprogramming
  - Scheduling: what process to run?
  - Dispatching: how to switch?
  - Memory protection: how to protect from one another?

- 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)

Espaço de endereçamento

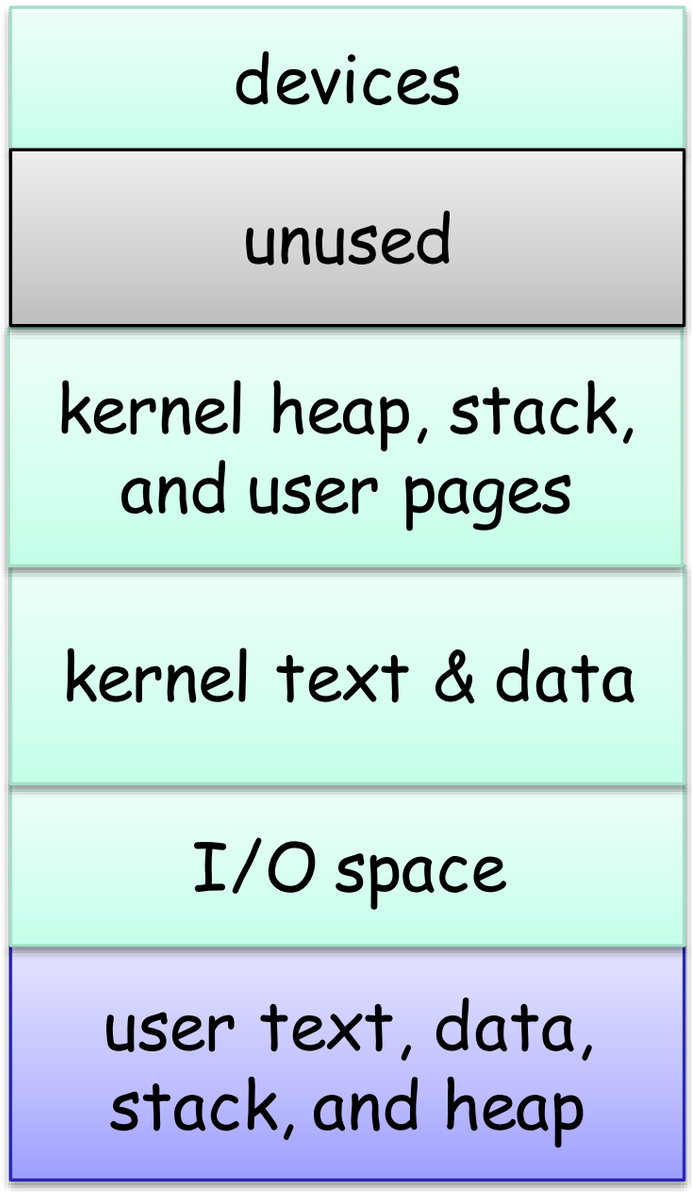
- Address Space (AS): all memory a process can address
  - Really large memory to use
  - Linear array of bytes: [0, N), N roughly 2^32, 2^64
- Process ⇔ address space: 1 : 1 mapping
- Address space = protection domain
  - OS isolates address spaces
  - One process can't access another's address space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

Exemplos de “Address Space”



No xv6

- 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



Despacho de processos

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?

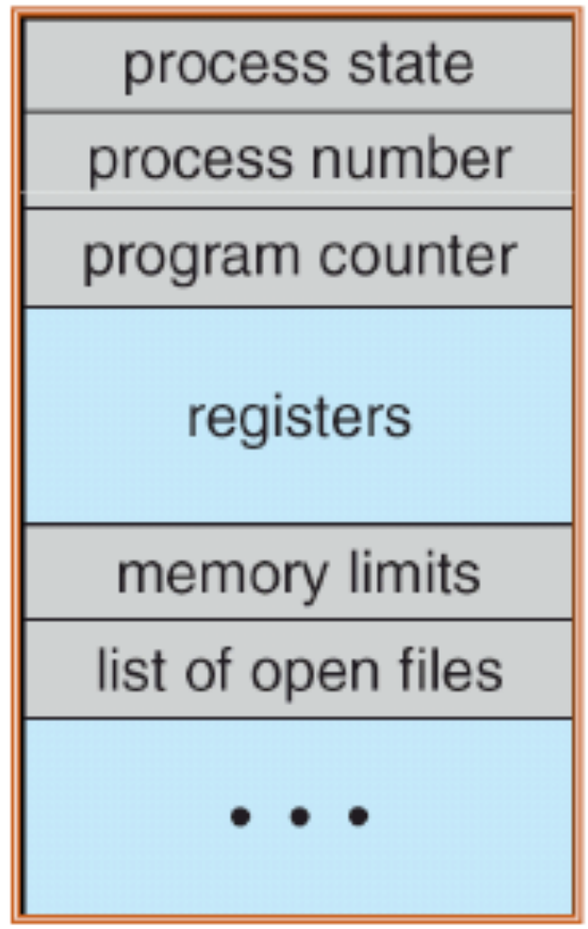
Questão #1: Como retomar o controle?

- 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!

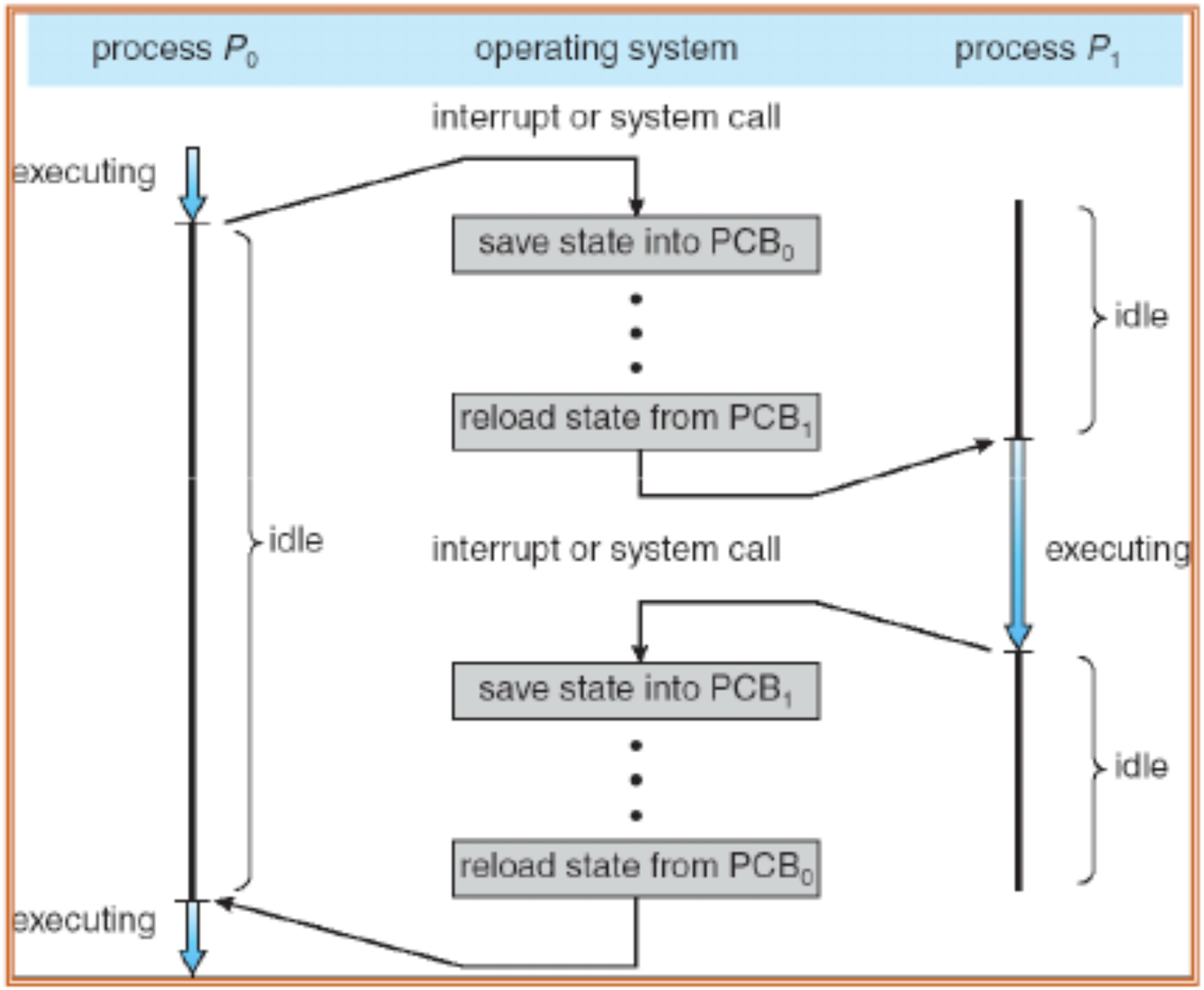
Que estado deve ser salvo ?

- 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



Troca de CPU entre processos



Questão #2: Como trocar o contexto ?

- 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

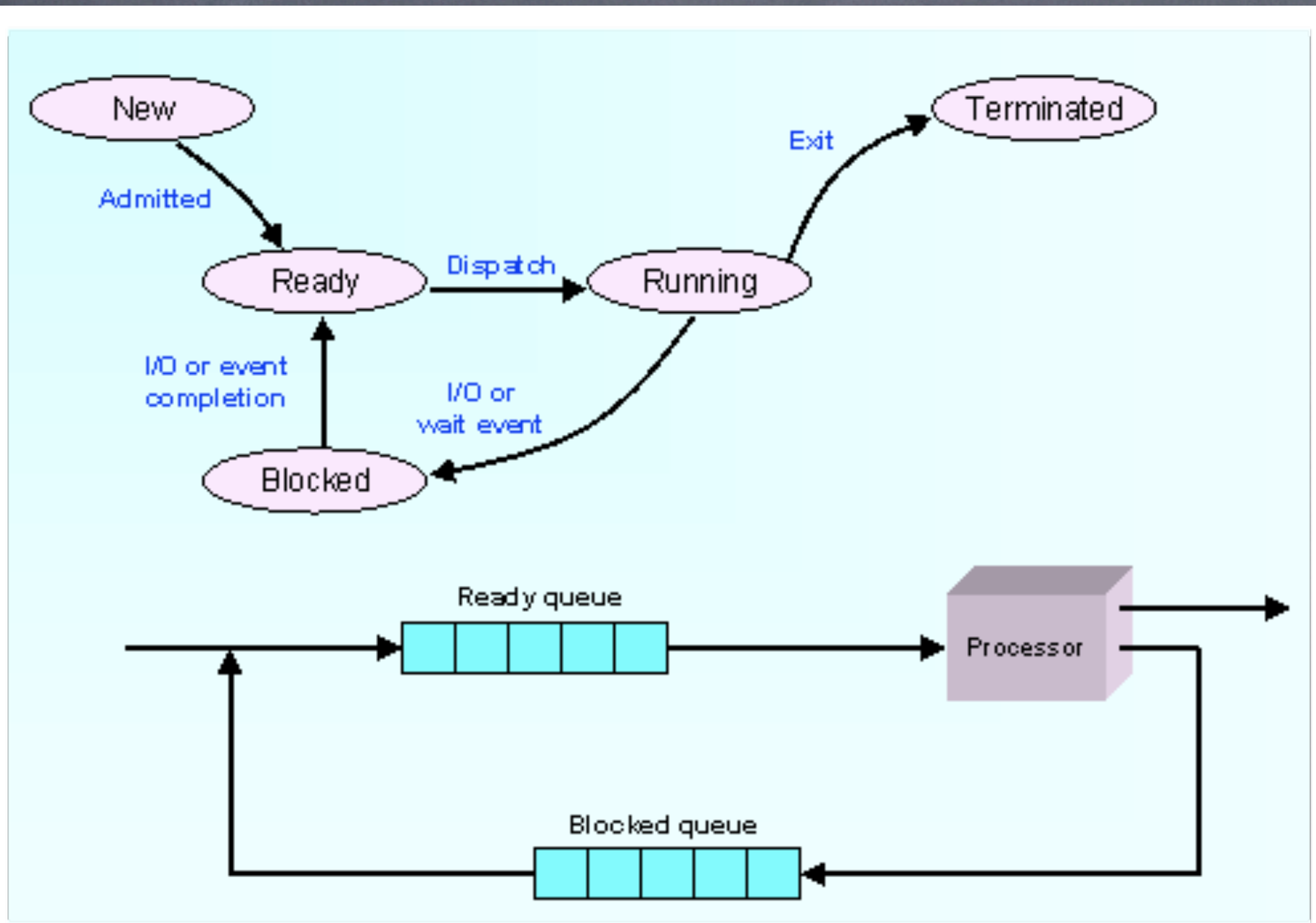
- Performance?
  - Can take long. A lot of stuff to save and restore. The time needed is hardware dependent
  - Context switch time is pure overhead: the system does no useful work while switching
  - Must balance context switch frequency with scheduling requirement

Onde achar os processos (PCBs) ?

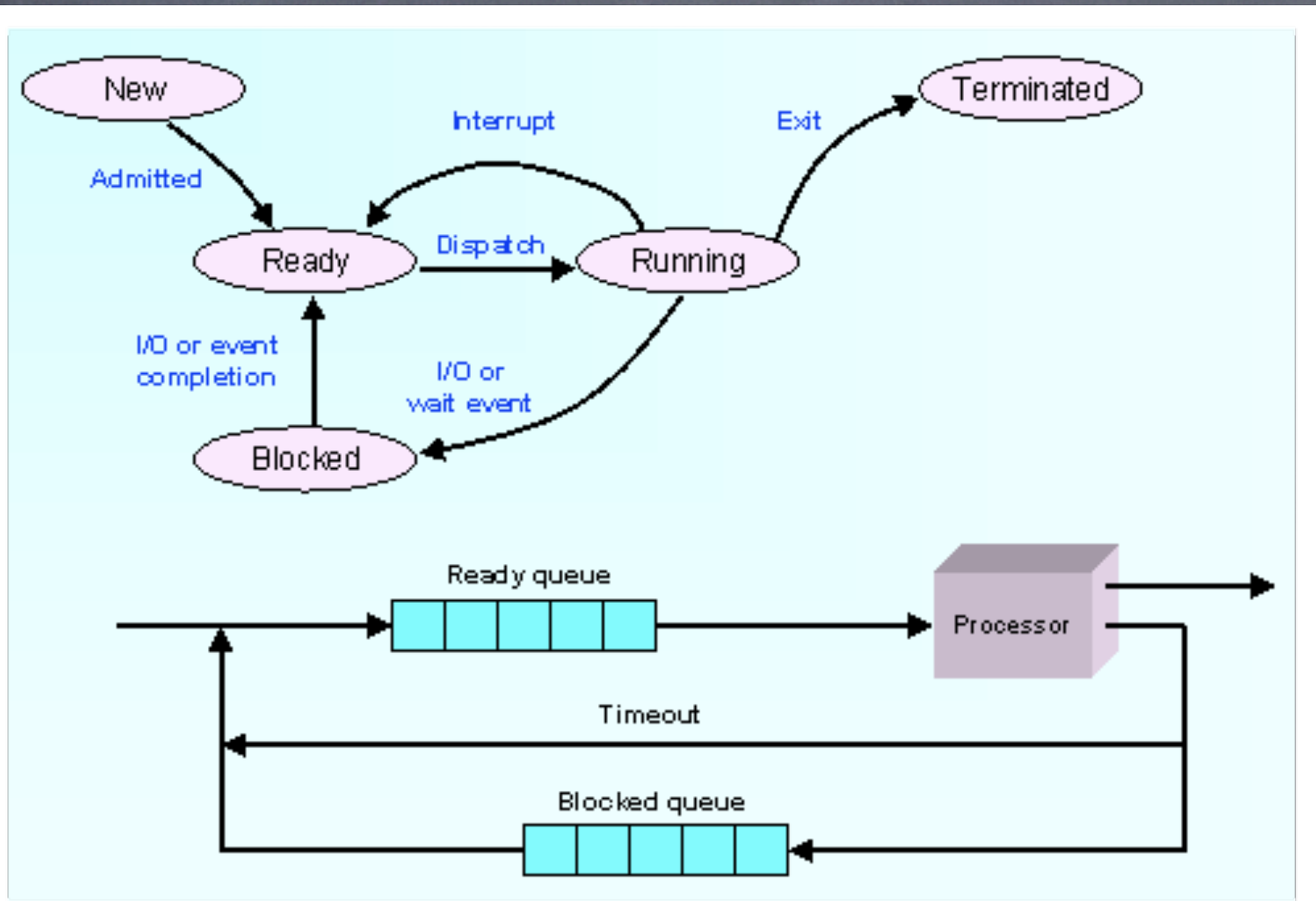
- Data structure: process scheduling queues
  - Job queue – set of all processes in the system
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute
  - Device queues – set of processes waiting for an I/O device
- Processes migrate among the various queues when their states change

Como funciona no xv6 ?

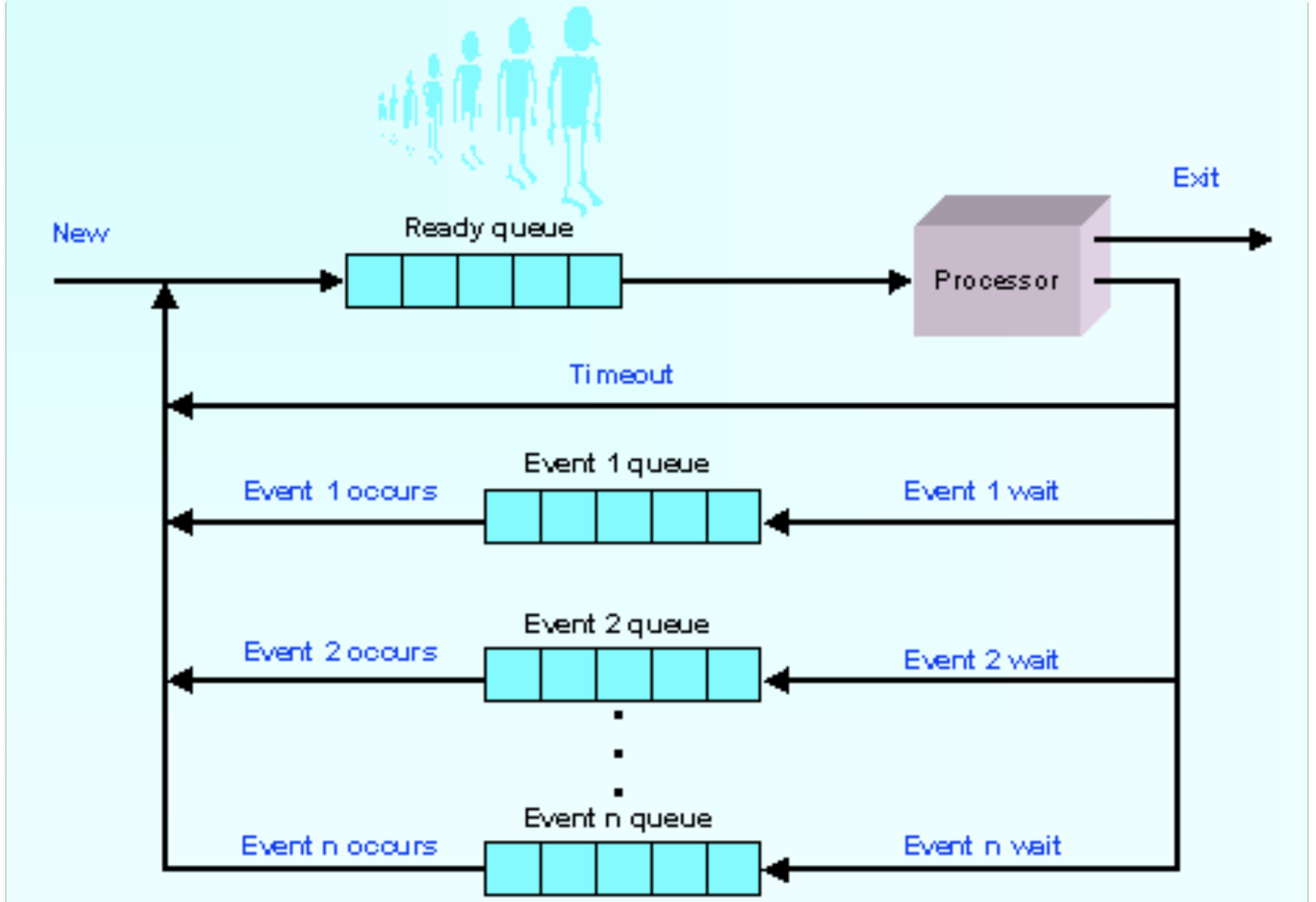
Transição entre estados



Transição entre estados

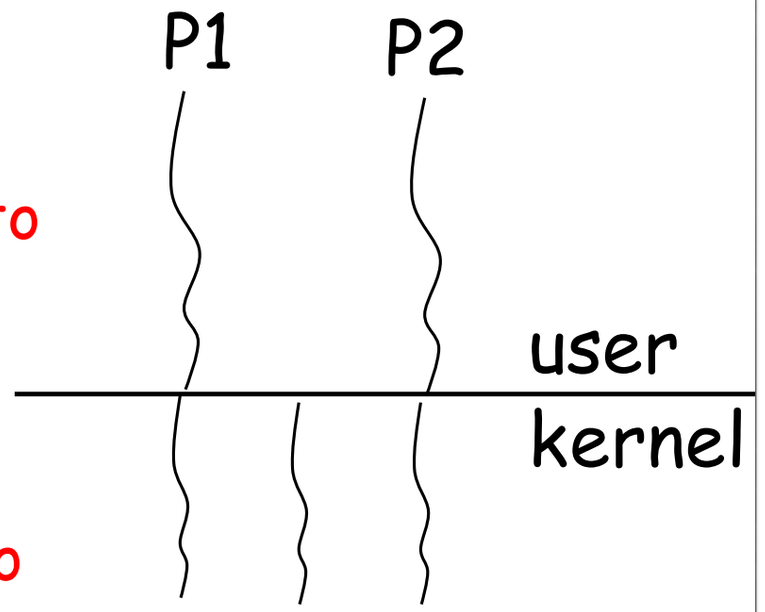


Múltiplas filas de espera



Troca de contexto no xv6

- 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

Criação de processos

- 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

Término de processos

- 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

Entre zumbis e órfãos

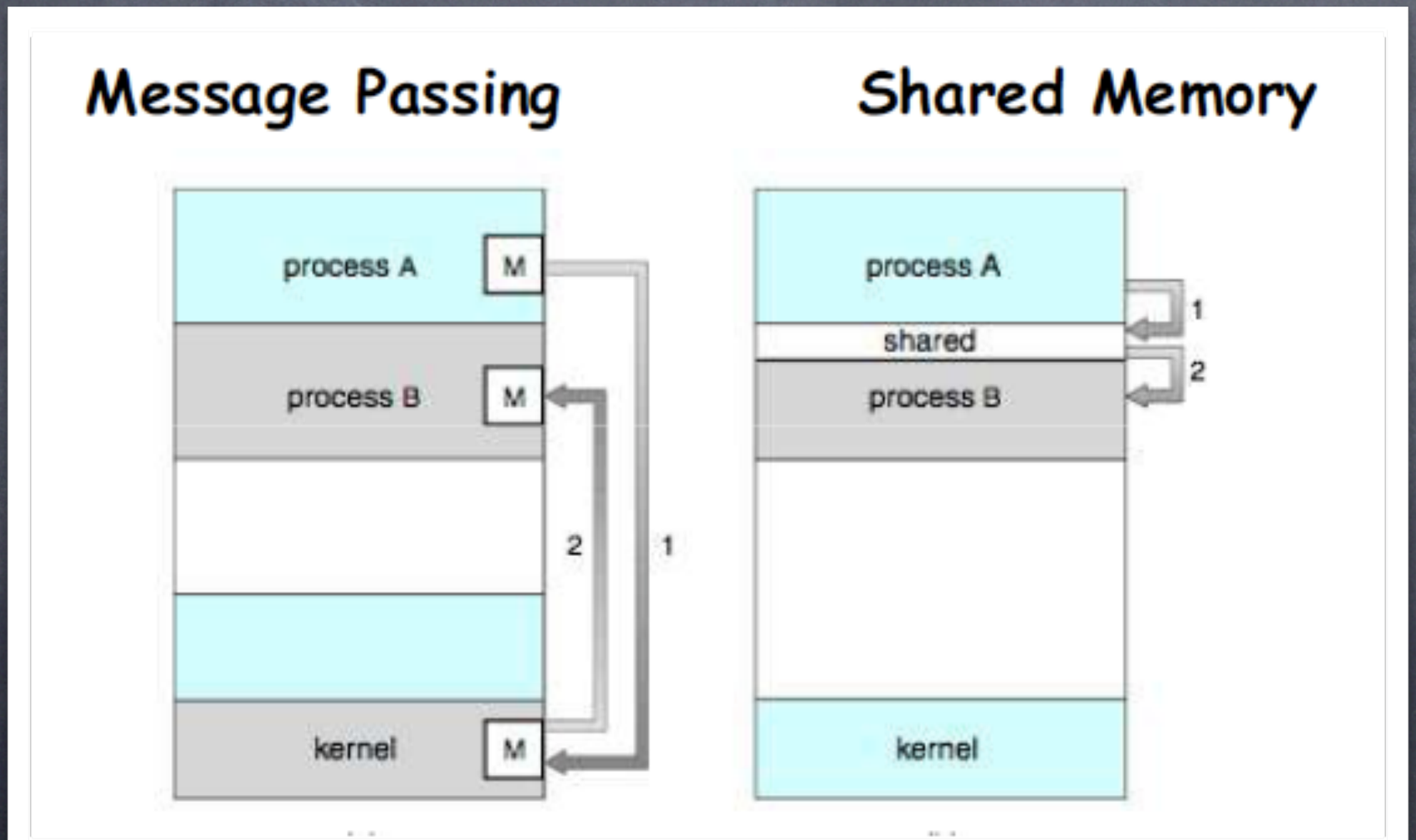
- What if child exits before parent?
  - Child becomes zombie
    - Need to store exit status
    - OS can't fully free
  - Parent must call wait() to reap child

- 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

Processos cooperativos

- 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

Modelos de comunicação interprocessos (IPC)



Troca de mensagns vs. Memória compartilhada

- 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

Exemplo de IPC: Sinais em Unix/Linux

- 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)

Exemplo de IPC: Pipes

- 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;
}
```

Exemplo de IPC: Memória compartilhada

- 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()

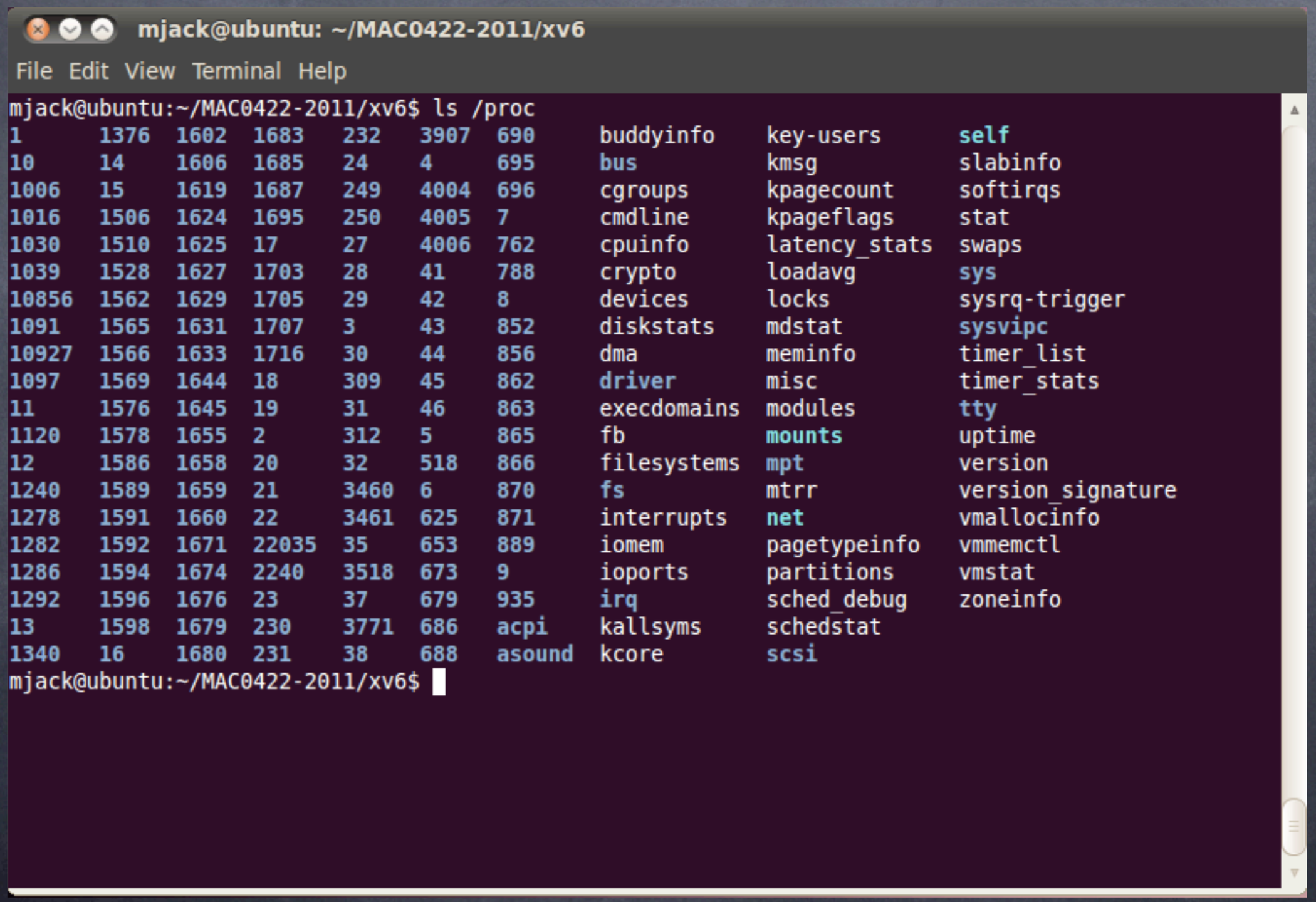
- int shmdt(const void *shmaddr);
  - Detach from shared memory

- Problem: synchronization!

Informações sobre processos

- ps
- top
- For each process, there is a corresponding directory /proc/<pid> to store this process information in the /proc pseudo file system

ls /proc


mjack@ubuntu: ~/MAC0422-2011/xv6
File Edit View Terminal Help
mjack@ubuntu:~/MAC0422-2011/xv6$ ls /proc
1 1376 1602 1683 232 3907 690 buddyinfo key-users self
10 14 1606 1685 24 4 695 bus kmsg slabinfo
1006 15 1619 1687 249 4004 696 cgroups kpagecount softirqs
1016 1506 1624 1695 250 4005 7 cmdline kpageflags stat
1030 1510 1625 17 27 4006 762 cpuinfo latency_stats swaps
1039 1528 1627 1703 28 41 788 crypto loadavg sys
10856 1562 1629 1705 29 42 8 devices locks sysrq-trigger
1091 1565 1631 1707 3 43 852 diskstats mdstat sysvipc
10927 1566 1633 1716 30 44 856 dma meminfo timer_list
1097 1569 1644 18 309 45 862 driver misc timer_stats
11 1576 1645 19 31 46 863 execdomains modules tty
1120 1578 1655 2 312 5 865 fb mounts uptime
12 1586 1658 20 32 518 866 filesystems mpt version
1240 1589 1659 21 3460 6 870 fs mtrr version_signature
1278 1591 1660 22 3461 625 871 interrupts net vmallocinfo
1282 1592 1671 22035 35 653 889 iomem pagetypeinfo vmmemctl
1286 1594 1674 2240 3518 673 9 ioports partitions vmstat
1292 1596 1676 23 37 679 935 irq sched_debug zoneinfo
13 1598 1679 230 3771 686 acpi kallsyms schedstat
1340 16 1680 231 38 688 asound kcore scsi
mjack@ubuntu:~/MAC0422-2011/xv6$

Código-fonte do Linux relacionado à processos

- Header files
  - include/linux/sched.h - declarations for most process data structures
  - include/linux/wait.h - declarations for wait queues
  - include/asm-i386/system.h - architecture-dependent declarations

- Source files
  - kernel/sched.c - process scheduling routines
  - kernel/signal.c - signal handling routines
  - kernel/fork.c - process/thread creation routines
  - kernel/exit.c - process exit routines
  - fs/exec.c - executing program
  - arch/i386/kernel/entry.S - kernel entry points
  - arch/i386/kernel/process.c - architecture-dependent process routines

Processo vs. Thread

- Thread: separate streams of execution that share the same address space

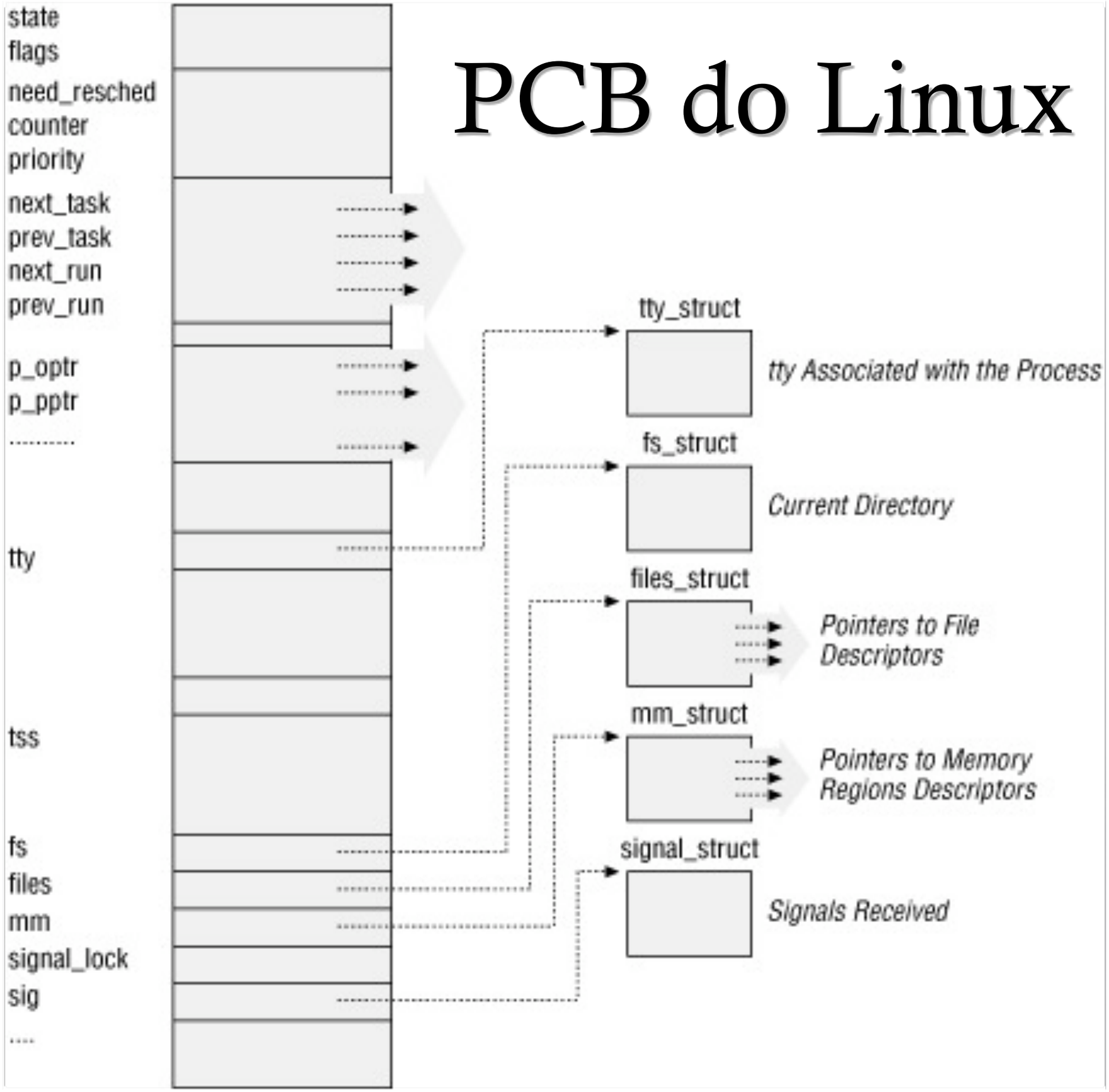
- Process != Thread
  - One process can have multiple threads
  - Threads communicate more efficiently

Linux: Processos ou Threads ?

- Linux uses a neutral term: tasks
  - Tasks represent both processes and threads
  - When processes trap into the kernel, they share the Linux kernel's address space → kernel threads

- task_struct
  - include/linux/sched.h
  - Each task has a unique task_struct

PCB do Linux


state
flags
need_resched
counter
priority
next_task
prev_task
next_run
prev_run
p_optr
p_pptr
.....
tty
tss
fs
files
mm
signal_lock
sig
....
tty_struct
tty Associated with the Process
fs_struct
Current Directory
files_struct
Pointers to File Descriptors
mm_struct
Pointers to Memory Regions Descriptors
signal_struct
Signals Received

Estados de um processo em Linux

- state: what state a process is in
  - TASK_RUNNING – the thread is running on the CPU or is waiting to run
  - TASK_INTERRUPTIBLE – the thread is sleeping and can be awoken by a signal (EINTR)
  - TASK_UNINTERRUPTIBLE – the thread is sleeping and cannot be awakened by a signal
  - TASK_STOPPED – the process has been stopped by a signal or by a debugger
  - TASK_TRACED – the process is being traced via the ptrace system call
- exit_state: how a process exited
  - EXIT_ZOMBIE – the process is exiting but has not yet been waited for by its parent
  - EXIT_DEAD – the process has exited and has been waited for

Identificadores de processos

- process ID: pid
- thread group ID: tgid
  - pid of first thread in process
  - getpid() returns this ID, so all threads in a process share the same process ID

- many system calls identify a process by its PID
  - Linux kernel uses pidhash to efficiently find processes by pids
  - see include/linux/pid.h, kernel/pid.c

Tarefa de casa

- Identifique a estrutura de dados que armazena a PCB no xv6
  - Que informações ela contém, onde está localizada e quais arquivos a manipulam ?
- Agora que você já sabe (!) os estados de um processo em xv6, responda:
  - Processos em diferentes estados aguardam em diferentes filas (e.g. esperando por E/S, tempo de execução expirado) no xv6 ?
- Responda essas perguntas no fórum da disciplina.