DCS216 Operating Systems

Lecture 06
Processes (2)

Mar 13th, 2024

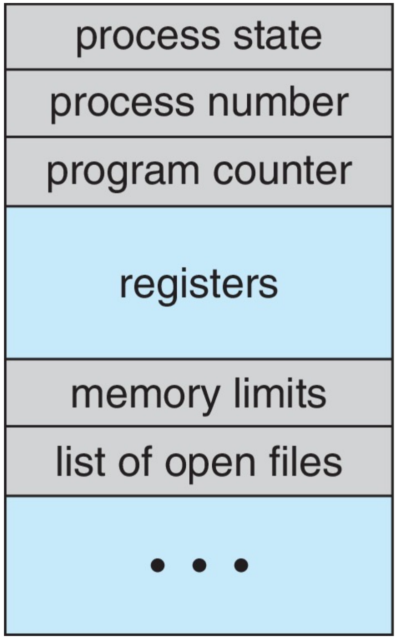
Instructor: Xiaoxi Zhang
Sun Yat-sen University

Content

Content

Multiplexing Processes

- **Process Control Block (PCB) (Information associated with each process):**
  - **Process State:** running, waiting, etc.
  - **Program Counter:** location of instruction to execute next
  - **CPU Registers:** contents of all process-centric registers
  - **CPU Scheduling Info:** priorities, scheduling queues
  - **Memory Management Info:** memory allocated to process
  - **Accounting Info:** CPU used, clock time elapsed, time limits
  - **I/O Info:** I/O devices allocated to process, list of open files

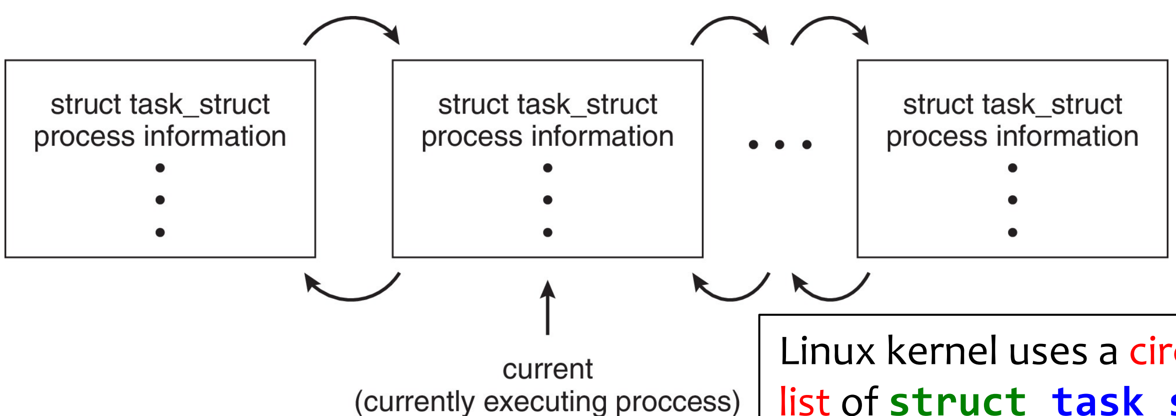


What does PCB actually look like in Linux source code?

struct task_struct (Simplified)

- Process in Linux is represented by the **task_struct** data structure

```
struct task_struct {
    pid_t pid;                       /* process identifier */
    long state;                      /* state of the process */
    unsigned int time_slice;         /* scheduling information */
    struct task_struct *parent;      /* this process's parent */
    struct list_head children;       /* this process's children */
    struct files_struct *files;      /* list of open files */
    struct mm_struct *mm;            /* address space of this process */
    ...
};
```



Linux kernel uses a circular doubly-linked list of **struct task_struct** to store process descriptors.

Process in Linux

struct task_struct (in the real world)

- Process in Linux is represented by the **task_struct** data structure

```
$ cd ~/work/linux-6.6.20/include/linux/
$ vim sched.h
```

```

struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	unsigned int			__state;

#ifdef CONFIG_PREEMPT_RT
	/* saved state for "spinlock sleepers" */
	unsigned int			saved_state;
#endif

	/*
	 * This begins the randomizable portion of task_struct. Only
	 * scheduling-critical items should be added above here.
	 */
	randomized_struct_fields_start

	void				*stack;
	refcount_t			usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;
	unsigned int			ptrace;
```

Process in xv6

- struct proc (much simpler in the good old days)
  - Process in **xv6** (a re-implementation of **UNIX** Version 6 by MIT PDOS)
    https://github.com/mit-pdos/xv6-public: proc.h

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // switch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

Process States

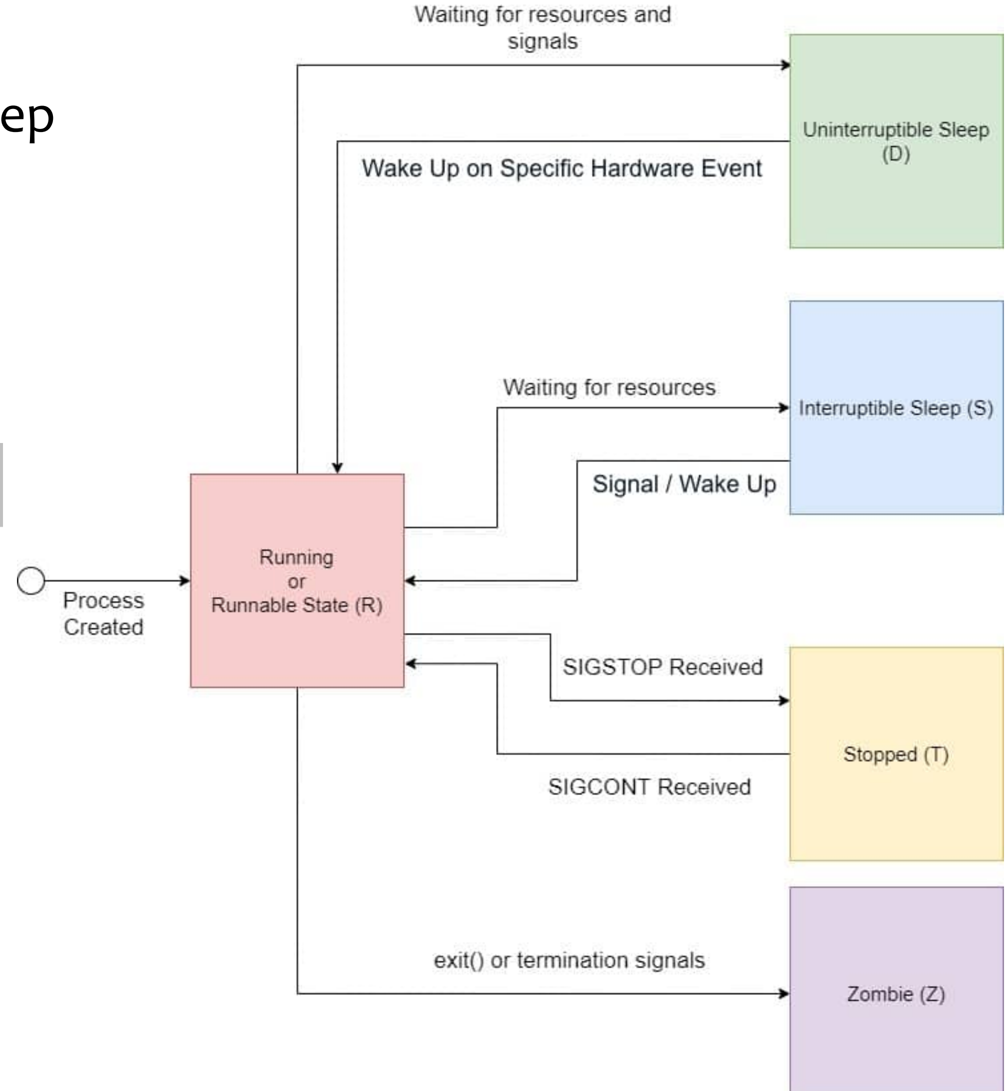
- R: Running or Runnable
- S: Interruptible Sleep
- D: Disk (Uninterruptible) Sleep
- T: Stopped
- Z: Zombie

/ fs / proc / array.c

```
static const char * const task_state_array[] = {

	/* states in TASK_REPORT: */
	"R (running)",		/* 0x00 */
	"S (sleeping)",		/* 0x01 */
	"D (disk sleep)",	/* 0x02 */
	"T (stopped)",		/* 0x04 */
	"t (tracing stop)",	/* 0x08 */
	"X (dead)",		/* 0x10 */
	"Z (zombie)",		/* 0x20 */
	"P (parked)",		/* 0x40 */

	/* states beyond TASK_REPORT: */
	"I (idle)",		/* 0x80 */
};
```



Process in Linux

Process States

- R: Running or runnable, it is just waiting for the CPU to process it
- S: Interruptible Sleep, waiting for an event to complete, such as input from the terminal
- D: Disk (Uninterruptible) sleep, processes that cannot be **killed** or interrupted with a signal, usually to make them go away you have to reboot or fix the issue
- T: Stopped, a process that has been suspended/stopped
- Z: Zombie, terminated processes that are waiting to have their

```
Tasks: 183 total,   1 running, 182 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.7 us,  1.1 sy,  0.0 ni, 97.1 id,  0.4 wa,  0.0 hi,  0.7 si,  0.0 st
MiB Mem :   3936.4 total,   1925.0 free,    850.6 used,   1160.8 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   2834.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   2237 bob       20   0  252252  81740  49204 S   2.3   2.0   0:09.37 Xorg
   2519 bob       20   0 3428664 375256 125080 S   2.0   9.3   0:19.57 gnome-shell
   2909 bob       20   0  966852  49944  37308 S   1.0   1.2   0:02.28 gnome-terminal-
      1 root      20   0  103500  13312   8620 S   0.7   0.3   0:04.44 systemd
   3588 bob       20   0   20600   3936   3380 R   0.3   0.1   0:00.01 top
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
```

Process Scheduling

- **Two Different Approaches in OS Design**
  - **Multiprogramming**
  - **Time sharing**

Two Different Approaches in OS Design

- **Multiprogramming**
  - **Objective:** Maximize CPU Utilization. The idea is to keep the CPU busy at all times by having multiple programs in memory. When one process is **waiting for I/O** or some other event, the CPU can switch to another process that is ready to execute.
  - **Focus:** Efficiency and throughput
- **Time sharing**
  - **Objective:** Minimize Response Time. The goal is to allow multiple users to interact with their programs as if each one had its own processor. The CPU switches rapidly between different tasks, giving each task a small time slice. This creates the illusion of simultaneous execution and immediate response. The result is better user experience.
  - **Focus:** Responsive user experience rather than maximize CPU utilization.

- **Two Different Approaches in OS Design**
  - **Multiprogramming**
    - **Objective:** Maximize CPU Utilization. The idea is to keep the CPU busy at all times by having multiple programs in memory. When one process is **waiting for I/O** or some other event, the CPU can switch to another process that is ready to execute.
    - **Focus:** Efficiency and throughput
  - **Time sharing**
    - **Objective:** Minimize Response Time. The goal is to allow multiple users to interact with their programs as if each one had its own processor. The CPU switches rapidly between different tasks, giving each task a small time slice. This creates the illusion of simultaneous execution and immediate response. The result is better user experience.
    - **Focus:** Responsive user experience rather than maximize CPU utilization.
  - How to achieve **multiprogramming** or **time sharing**?

Process Scheduling

- How to achieve **multiprogramming** or **time sharing**?
  - Different **policies**: which process to run at any given time?
    - Resource Utilization vs. Responsiveness
    - System-Oriented vs. User-Oriented
    - Batch Processing vs. Interactive Use
  - **Mechanism**: Process scheduler provides the **mechanism** that enables both multiprogramming and time sharing, or any other OS design objectives, such as **fairness**.

Process Scheduling

- The process scheduler selects an available process
  - ...(possibly from a set of several available processes)
- ...for program execution on a CPU core
- Each CPU core can only run one process at a time.
- If there are more processes than CPU cores, excess processes will have to wait until a core is free and can be rescheduled.
- Degree of multiprogramming(多道程序的程度): the number of processes currently in memory

Process Scheduling

- Scheduler needs to consider the behavior of processes:
  - **CPU-bound**(计算密集型): spends most of its time doing computation
    - Example: Video editing, scientific computing
  - **I/O-bound**(I/O密集型): spends most of its time requesting I/O
    - Example: web servers, databases, editors like VSCode

Process Scheduling

Multiprogramming

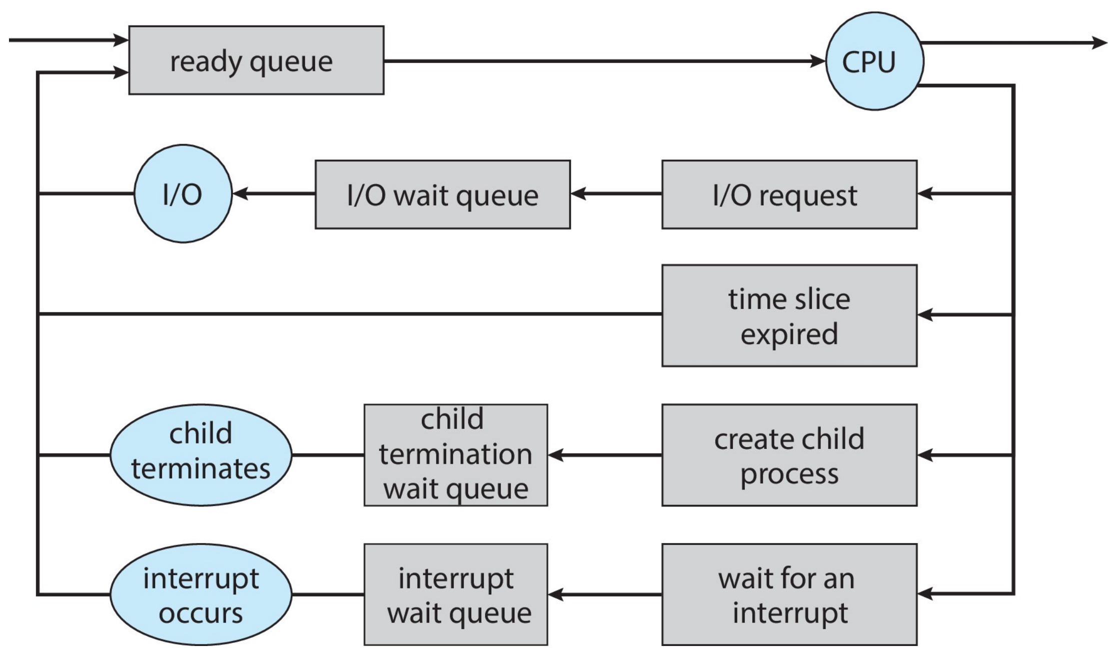
- CPU-bound processes benefit less from multiprogramming because they rarely enter a waiting state. However, multiprogramming still helps by switching to another process when a CPU-bound process completes its time quantum.
- I/O-bound processes spends most of their time waiting for I/O. Multiprogramming is particularly effective for I/O-bound processes. While one process waits for I/O, the CPU can be allocated to other processes, thereby improving overall efficiency and throughput.

Time Sharing

- CPU-bound processes are regularly interrupted to allow other processes to run in time sharing systems. This ensures that no single process monopolizes the CPU, which is crucial for improving responsiveness
- I/O-bound processes also benefit from time sharing, as it allows them to quickly respond to user inputs and complete their I/O operations. Time sharing ensures that these processes receive timely CPU attention for their short bursts of processing needs.

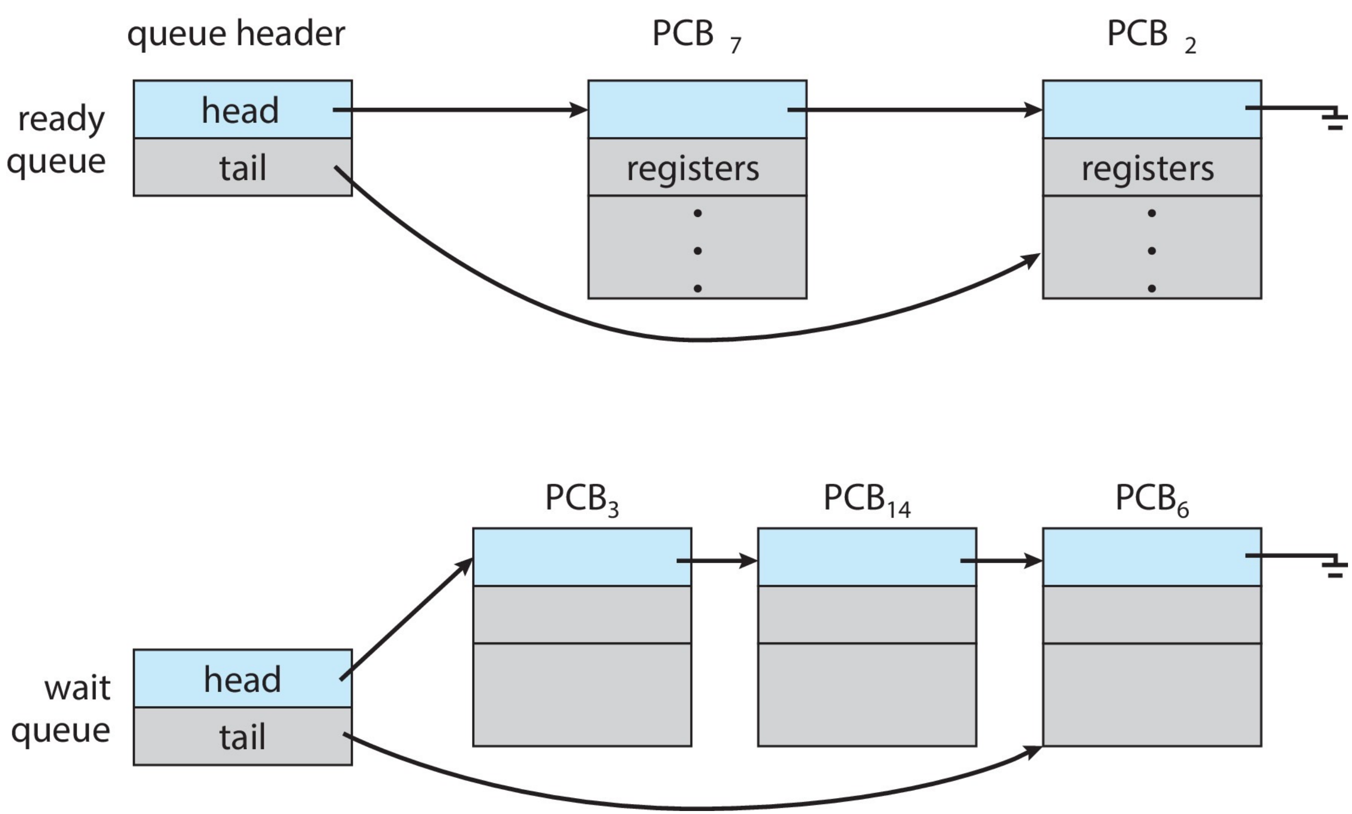
Process Scheduling

- Process scheduler selects process for next execution on CPU core
- Goal: Maximize CPU use, quickly switch processes onto CPU core
- Maintains scheduling queues of processes
  - Ready queue: set of all processes residing in main memory, ready and exiting to execute
  - Wait queues: set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues



Ready and Wait Queues

- Process not running → PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler **policy**



Process Scheduling

- **Types of Process Schedulers**
  - **Long-term** Scheduler
    - High-level scheduler, or **job scheduler**
    - controls degree of multiprogramming (number of processes in memory)
  - **Short-term** Scheduler
    - Low-level scheduler, aka, **CPU scheduler**
    - Process/thread scheduler in a narrow sense (狭义上的进程/线程调度)
    - Selects from among the processes that are ready to execute
  - **Medium-term** Scheduler
    - Medium-level scheduler, or **swapping** scheduler
    - reduce the degree of multiprogramming by swapping idle processes out of memory (into disk)

Long-term Scheduler

- Long-term scheduler selects which programs/processes should be brought into the ready queue
  - determines which programs are admitted to the system for processing
  - controls the degree of multiprogramming
  - strives for good process mix of CPU-bound and I/O-bound processes
  - Long-term scheduler is invoked infrequently:
    - minutes, or hours
    - might take a long time to make a scheduling decision
  - Example: job scheduler for batch processing systems, e.g., **SLURM**
    - **SLURM** usually runs on a cluster of computers, such as 天河二号
  - Modern commercial OSes like UNIX and Windows normally don't have a default long-term scheduler.
    - In a sense, the USER acts as the manual *long-term scheduler* via a shell or a GUI.

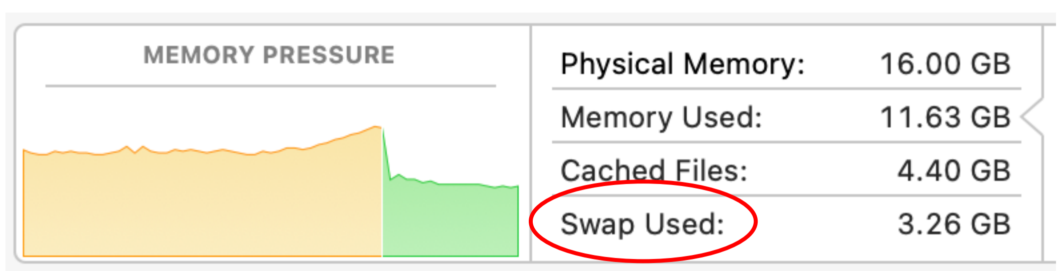
Process Scheduling

Short-term Scheduler

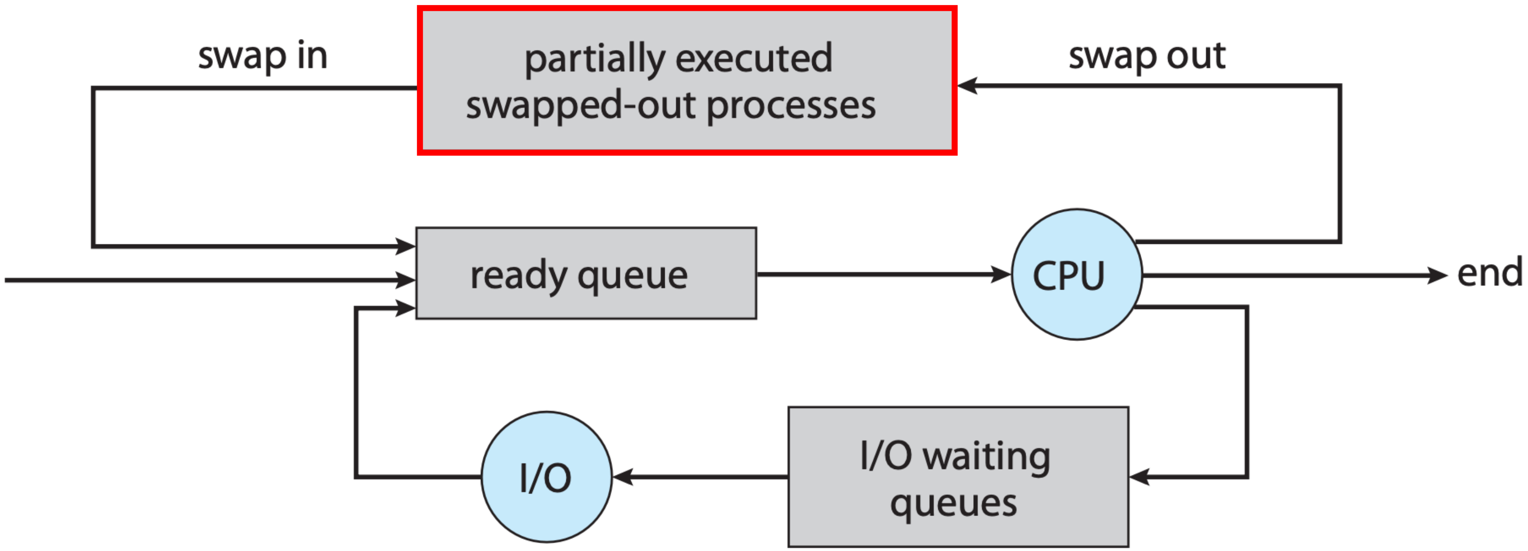
- Short-term scheduler selects which process should be executed next and allocates CPU – also called CPU scheduler
  - CPU scheduling determines which process is going to execute next according to a **scheduling algorithm**.
  - Short-term scheduler is also known as the **dispatcher(分派器)** that moves the processor from one process to another, and prevents a single process from monopolizing CPU time
  - Short-term scheduler is invoked on an event that may lead to choose another process for execution:
    - Timer interrupts
    - I/O interrupts
    - System calls and exceptions/traps
    - Signals
  - Short-term scheduler is invoked very frequently
    - milliseconds
    - must be fast and efficient

Medium-term Scheduler

- Medium-term scheduler selects which processes should be swapped out if the system is overloaded
  - So far, all processes have to be in main memory
  - Even with virtual memory, keeping too many processes in main memory will deteriorate the system's performance
  - OS may need to swap out some process to disk, and then swap them back in when system is less crowded
  - Swapping reduces the degree of multiprogramming
    - Degree of multiprogramming definition: number of processes in **memory**.



- **Medium-term Scheduler**



CPU Scheduler

- Scheduling: **Mechanism** for deciding which process to run on the CPU
  - One could say that this is all that the OS does; it never stops
- Lots of different scheduling **policies**:
  - Utilization optimization or …
  - Responsiveness optimization or …
  - Fairness …

Loop

```
if ( readyProcesses(PCBs) ) {
    nextPCB = selectProcess(PCBs);
    SaveStateOfCPU( currPCB );
    LoadStateOfCPU( nextPCB );
    run( nextPCB );
}
```

CPU Scheduler

- Scheduling: **Mechanism** for deciding which process to run on the CPU
  - One could say that this is all that the OS does; it never stops
- Lots of different scheduling **policies**:
  - Utilization optimization or ...
  - Responsiveness optimization or ...
  - Fairness ...

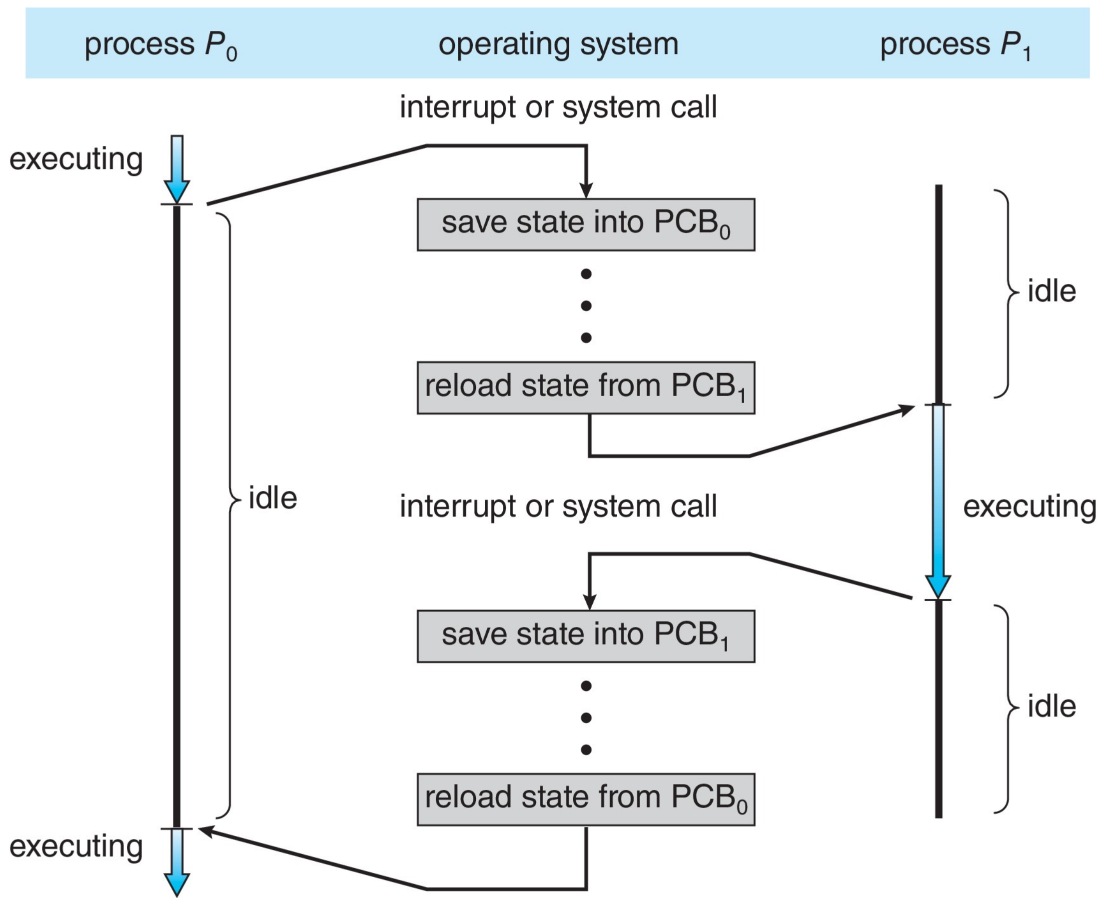
Loop

```
if ( readyProcesses(PCBs) ) {
    nextPCB = selectProcess(PCBs);
    SaveStateOfCPU( currPCB );
    LoadStateOfCPU( nextPCB );
    run( nextPCB );
}
```

Context Switch

Context Switch

- Context switch (上下文切换)occurs when CPU switches from one process to another

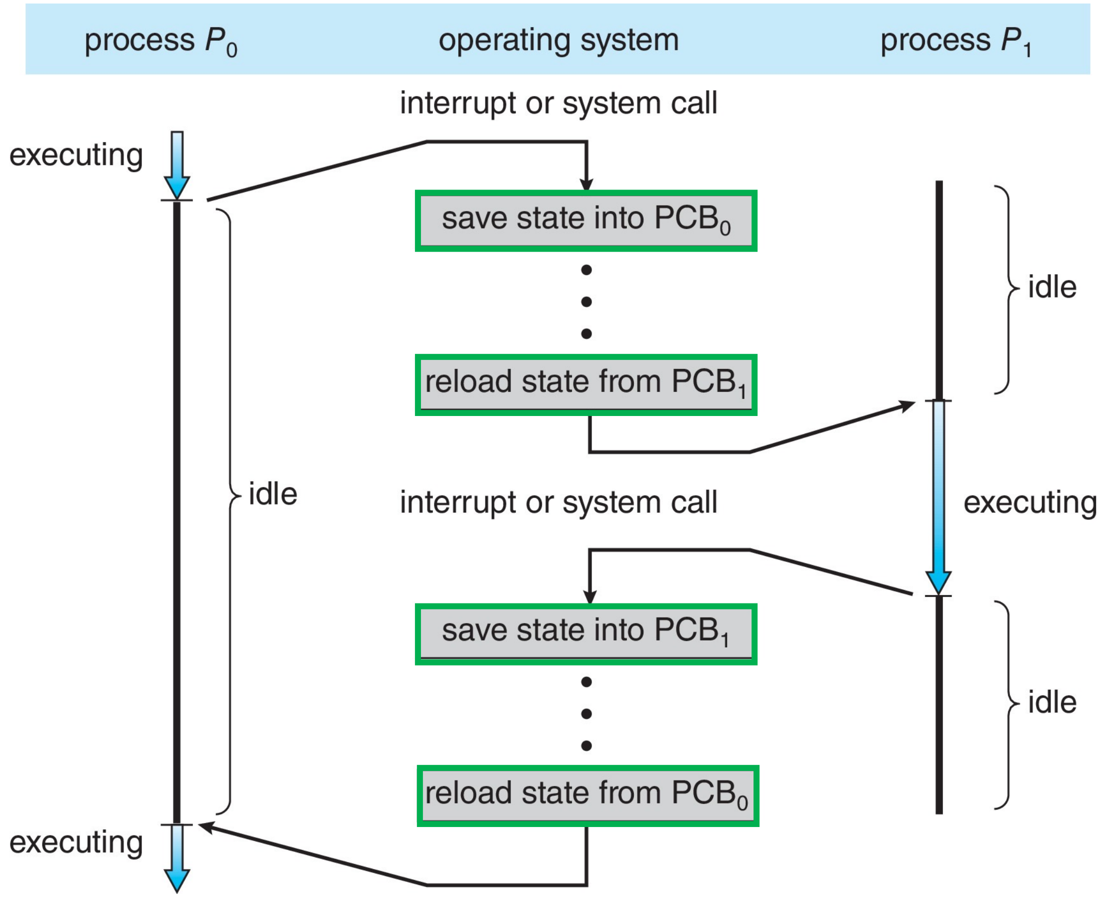


Context Switch

- **Context Switch**
  - When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
  - **Context** of a process represented in the **PCB**
  - Context switch time is pure overhead; the system does no useful work while switching
    - The more complex the OS and the PCB → the longer the context switch
  - Time dependent on hardware support
    - Some hardware provides multiple sets of register per CPU
      - → multiple contexts loaded at once

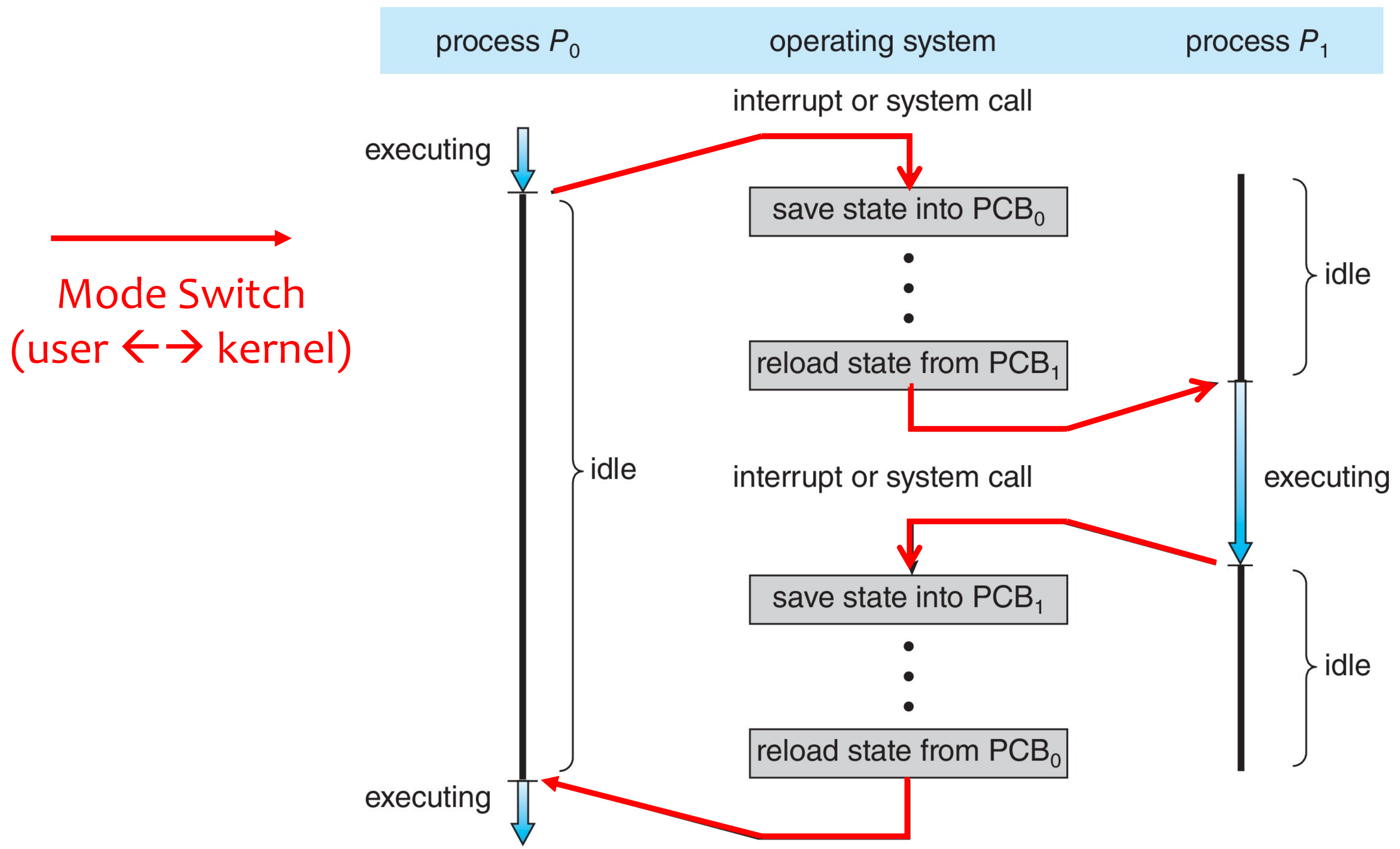
- **Context Switch**
  - Context switch occurs when CPU switches from one process to another

Context Switch



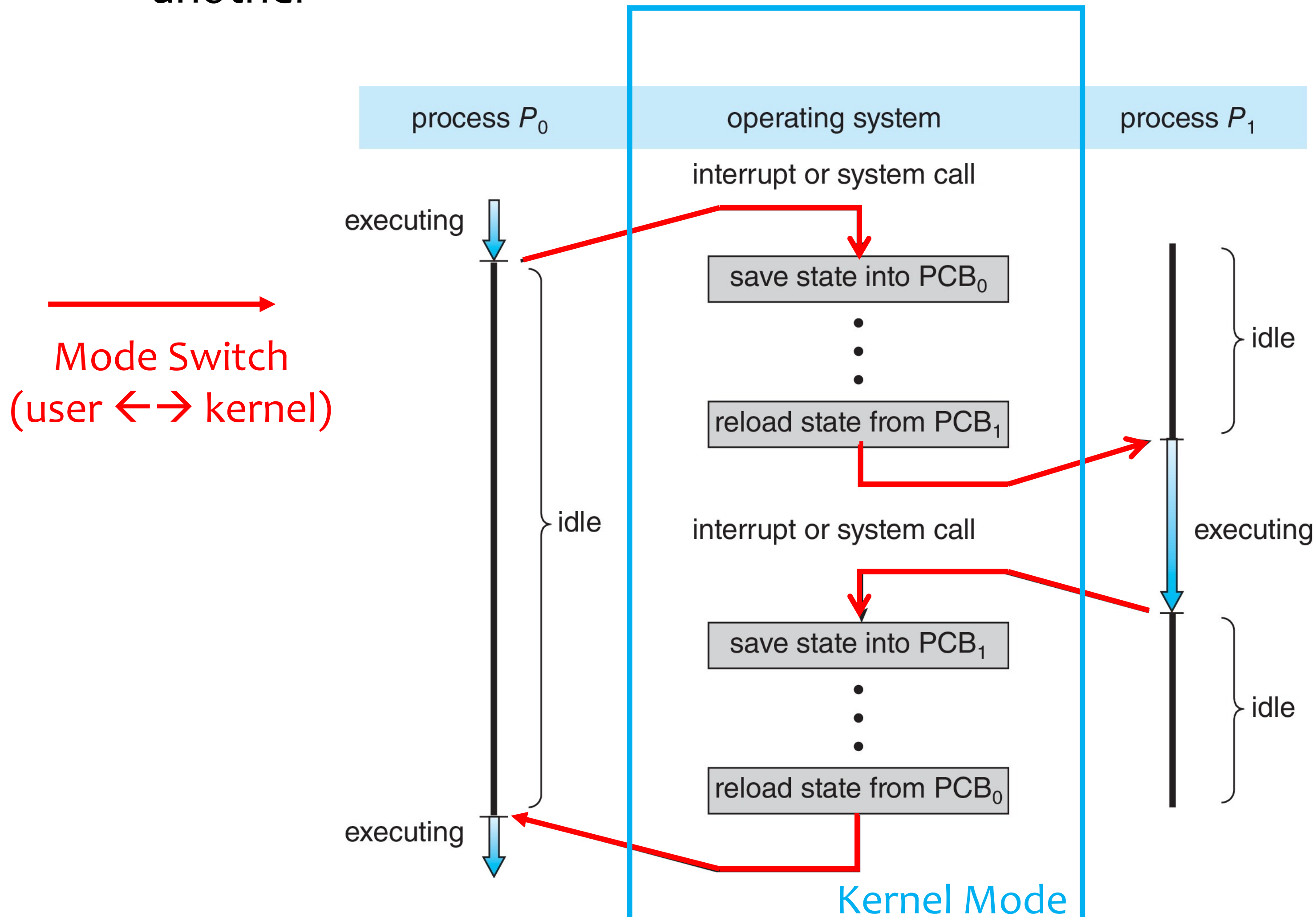
Context Switch

- Context switch occurs when CPU switches from one process to another



Context Switch

- Context switch occurs when CPU switches from one process to another



Context Switch

- **Context Switch vs Mode Switch**
  - Do not confuse **context switch** with **mode switch**, which typically happens in system calls, interrupts
  - A **mode switch** refers to the change in privilege level within the CPU, specifically switching between user mode and kernel mode.
  - A **mode switch** does not necessarily imply **context switch**
    - Example: The process performing mode switch (e.g., syscalls) may be running on the same CPU, without switching context.
  - A **context switch** does not necessarily imply **mode switch**
    - Example: The scheduler decides to switch CPU from executing user-mode Process A to user-mode Process B, without involving transition to kernel mode in between, i.e., **preemptive multitasking**.

Context Switch

- **Example: Multitasking in Mobile Systems**
  - Some mobile systems (e.g., **early version** of iOS) allow only one process to run, others suspended
  - Due to screen real estate, user interface limits iOS provides for a
    - Single foreground process – controlled via user interface
    - Multiple background processes – in memory, running, but not on the display and with limits
    - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
  - Android runs foreground and background, with fewer limits
    - Background process uses a service to perform tasks
    - Service can keep running even if background process is suspended
    - Service has no user interface, small memory use

Thank you!