
Introduction

中断和陷阱

硬件中断 | Interrupt

- 外部 IO 设备有事情要通知 CPU 进行处理，通过中断完成
- 硬件部分：
  - CPU 内部有中断控制器
  - IO 通过硬件连线向中断控制器发送信号
  - CPU 发现有中断在 pending
  - CPU 将 pc 设置为中断向量开始的位置
- 软件部分：
  - 跳转的位置存放了中断处理程序（Interrupt Service Routine, ISR）
  - 关掉中断（不希望在处理中断的时候再次出现中断时被 CPU 跳转 pc）
  - 检查目前是什么中断，并且 dispatch 到指定的处理程序（interrupt handler）
  - 上下文处理：
    - 在中断处理前要保存上下文（pc 等系列寄存器）
    - 处理后恢复上下文，继续执行原来程序

陷阱 | Trap

- trap，又叫软中断（soft interrupt）
- 硬件中断，由硬件产生，有外设硬件事件要处理
  - 是一个异步事件（asynchronous），即无法预测出现的时间
- trap 由软件产生，可能是运行出现了错误或者用户显式地请求中断
  - 是一个同步时间（synchronous）
  - 例如：系统调用，divided-by-zero 异常……

操作系统是 interrupt-driven 的

- 特殊的中断：计时器中断
- 定时产生计时器中断，让操作系统内核得到机会运行，进行评估、调度
- 否则如果没有其它中断产生，则用户态程序会一直运行占用资源，操作系统也没有机会介入管理

> 用户态的程序不允许直接操控磁盘控制器，但又需要读取其中的文件，会通过运行在特权态的操作系统内核操作（系统调用）

中断处理

- 先进行 CPU 执行状态的保存
  - 将 pc 等寄存器保存在内存中
- 两种处理机制
  - vectored interrupt system:
    - 硬件中断产生不会到统一的地址
    - 记录一个中断向量基地址，后面依次存储各个中断号对应的处理程序地址
    - 直接根据中断号来计算偏移，跳转到对应的处理程序中
  - polling:
    - 所有中断都跳转到统一的地址
    - 由软件来区分中断种类，决定如何处理等
- OS handles the interrupt by calling the **device’s driver (设备驱动程序)**

  > 宏内核的重要组件。而对于微内核，往往设备驱动程序放在操作系统内核范围外，为了安全性，常将设备驱动程序的大部分代码移到用户态。

- 处理后恢复执行状态
  - 恢复 pc 等寄存器，继续回去执行
  - 或者恢复其它进程的寄存器，实现调度

I/O: from System Call to Devices, and Back **

- A program uses a **system call** to access system resources
  - e.g., files, network
- Operating system converts it to device access and issues I/O requests
  - I/O requests are sent to the device driver, then to the controller
  - e.g., read disk blocks, send/receive packets...
- OS puts the program to wait (synchronous I/O) or returns to it without waiting (asynchronous I/O)

    - OS may switches to another program when the requester is waiting
    - 同步I/O：进行I/O的时候block掉当前请求I/O的程序，等待I/O结束（可能会调度运行另一个程序）
        - 常用，易用，但不高效
    - 异步I/O：即使I/O没有完成，仍然继续运行原程序，当I/O完成时，执行用户态定义的callback回调函数
        - 高效，常用于处理大量网络请求时

    > 即使是同步I/O，CPU也不是盲目等待状态。某一进程阻塞时，CPU可以调度另一进程

- I/O completes and the controller interrupts the OS
- OS processes the I/O, and then wakes up the program (synchronous I/O) or send its a signal (asynchronous I/O)

> 一种自上而下，一种自下而上

- 用户态程序想要操作外设
    - 通过系统调用（system call）来告诉 CPU 要操作外设
- 操作系统通过 MMIO（Memory Mapping I/O）来操纵外设

Direct Memory Access | DMA

- DMA is used for high-speed I/O devices able to transmit information at close to memory speeds
    - e.g., Ethernet, hard disk, cd rom...
- Device driver sends an I/O descriptor the controller
    - **I/O descriptor**: operation type (e.g., send/receive), memory address...
- The controller transfers blocks of data between its *local buffer* and *main memory* **without CPU intervention**(提高效率)

> CPU会告诉显卡需要的数据起始地址在哪，有多长，随后由DMA搬运数据。搬运结束会通过硬中断的方式告诉CPU

    - only one interrupt is generated when whole I/O request completes

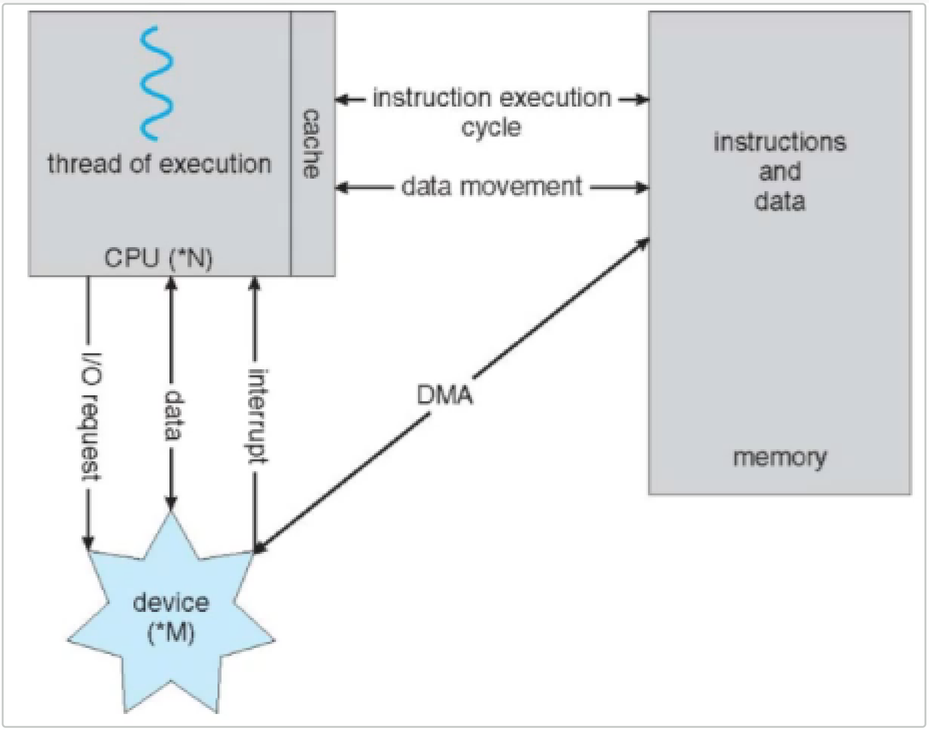
> 磁盘控制器分为常见的HDD和SSD

- 比如 CPU 想让显卡在显示器上显示数据
    - 正常情况
        - CPU 将大量数据依次搬运到显卡对应的内存空间上
        - 期间 CPU 只做这一件事，无法运行其它程序
    - 通过 DMA

        - CPU 告诉显卡的 DMA 控制器要搬运的数据在哪、有多长
        - DMA 控制器会进行大量数据的搬运
        - 此时 CPU 可以进行其它运算
        - 完成后通过中断来通知 CPU
- 有 DMA 可以在没有 CPU 参与的情况下进行大量数据的搬运
- 问题
    - 各个设备都有自己的 DMA 控制器，在 DMA 控制器进行数据搬运的时候全部的物理内存都暴露给了外设
        - 通过内存和 DMA 直接额外的硬件 IOMMU (I/O 内存管理单元) 来控制可以访问哪些物理内存

Put it together

> Computer System的示意图



Storage Structure

- Main memory: the only large storage that CPU can directly access
    - **random access**, and typically **volatile**
- Secondary storage: large **nonvolatile** storage capacity (文件和大量数据存在此处，包括HDD和SSD)

    > 具体内容系统Ⅲ讲

    - Magnetic disks are most common second-storage devices (HDD) (系统Ⅲ)
        - rigid metal or glass platters covered with magnetic recording material

    - disk surface is logically divided into **tracks** and **sectors**
    - disk controller determines the interaction between OS and the device

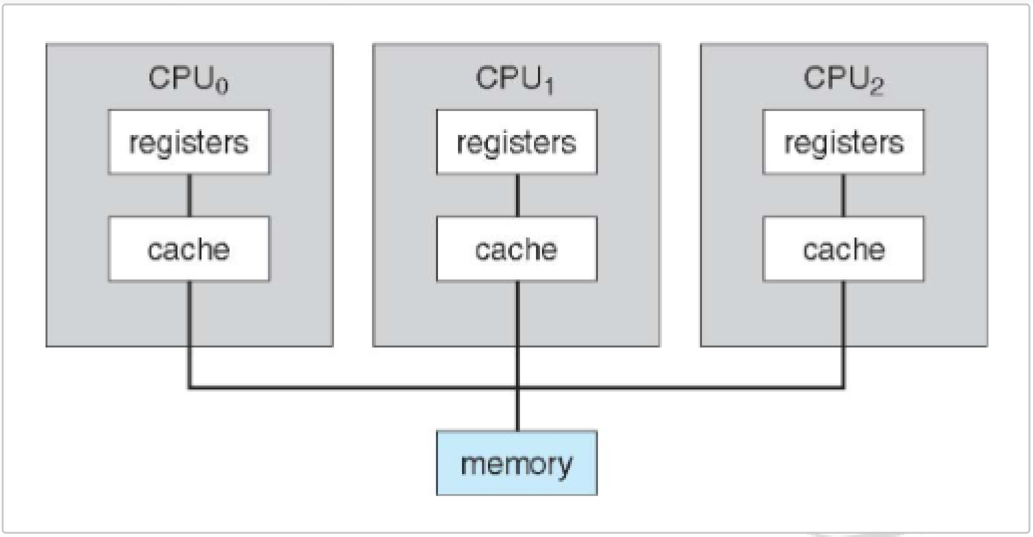
MUltiprocessor Systems

Multiprocessor systems have grown in use and importance

- also known as parallel systems, tightly-coupled systems
- advantages: increased throughput, economy of scale, increased reliability -- graceful degradation or fault tolerance
- two types: asymmetric multiprocessing and symmetric multiprocessing (SMP)

Symmetric Multiprocessing Architecture

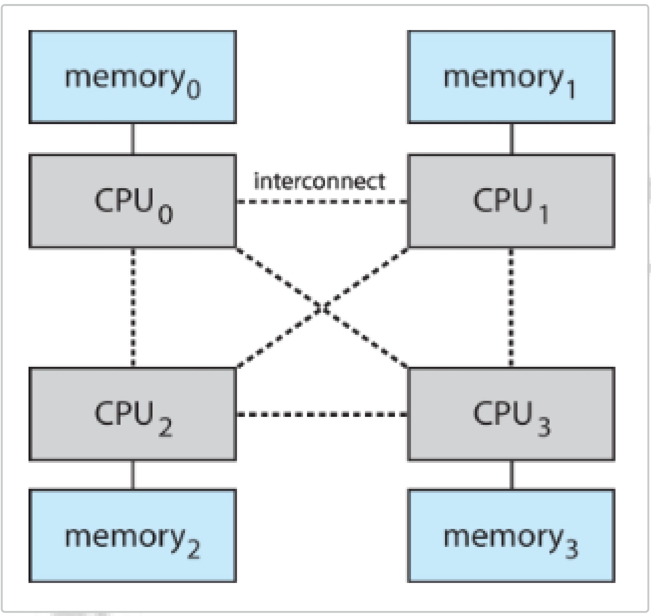
> 即指CPU有同等地位，共享一级或二级的cache，最后一级不共享



NUMA

- Non-Uniform Memory Access System
    - Access local memory is fast, scale well

> 每个CPU有自己的fixed memory，但所有的memory是统一编制的，即CPU可以访问所有memory



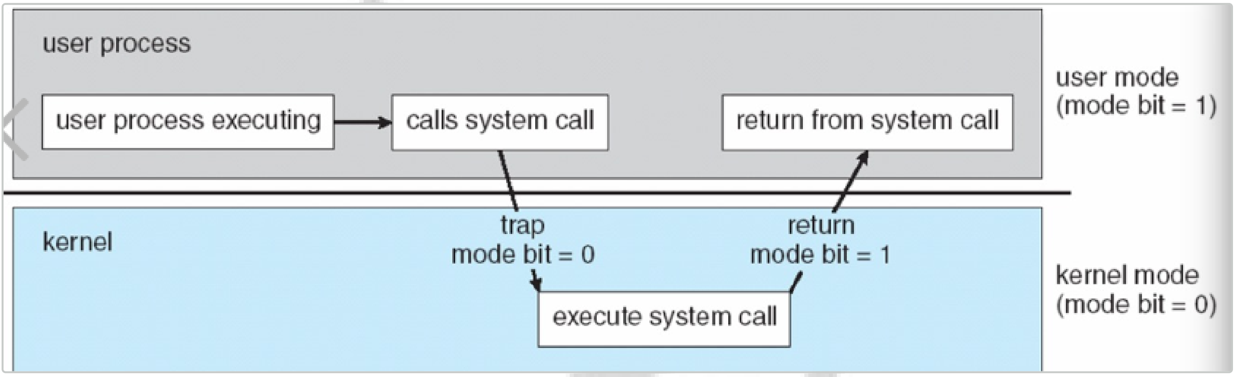
Dual-mode operation

- Operating system is usually interrupt-driven
    - Efficiency, regain control (timer interrupt)
- **Dual-mode operation** allows OS to protect itself and other system components
- **user mode** and **kernel mode** (or **other names**)
- a **mode** bit distinguishes when CPU is running user code or kernel code
- some instructions designated as **privileged**, only executable in kernel
- **system call** changes mode to kernel(通过trap), return from call resets it to user

> 1. 分两个模式是为了隔离，防止用户态程序去干扰操作系统内核的数据或数据结构。
> 2. 常用MMU部件（Memory Management Unit | 内存管理单元）来指定当CPU运行在特权态时可以访问所有物理地址空间，在用户态下只能访问有限空间达到二者的隔离
> 3. 部分指令必须在特权态下运行，比如关闭中断，打开中断、配置MMU的指令

Transition between Modes

- **System calls**, **exception**, **interrupts** cause transitions between kernel/user modes



Timer

- Timer used to prevent infinite loop or process hogging resources
  - to enable a timer, set the hardware to interrupt after some period
  - OS sets up a timer before scheduling process to regain control
    - the timer for scheduling is usually periodical(e.g., 250HZ)
    - **tickless kernel**: on-demand timer interrupts(Linux)

Resource Management: Process Management

- A *process* is a program in execution
  - program is a passive entity, *process* is an active entity
  - a system has many processes running concurrently

> 不同进程之间是隔离的，即使是同一个program加载的进程

- Process needs resources to accomplish its task
  - OS reclaims all reusable resources upon process termination
  - e.g., CPU, memory, I/O, files, initialization data
- 进程是资源分配的最小单元，线程是调度的最小单元

> 操作系统内核会管理分配给每一个内核的CPU资源、memory资源、I/O资源……是通过分时复用的方式，时间片轮转，每一个process运行一段时间就开始根据优先级调度

from Process to Thread

- Single-threaded process has one program counter
  - **program counter** specifies location of next instruction to execute
  - processor executes instructions sequentially, one at a time, until completion

> 线程之间是不隔离的

- Multi-threaded process has **one program counter per thread**
- Quiz: What are the benefits of using thread instead of process?

> 不同的进程间要共享数据，如果是线程间共享，那数据会很庞大；
> 引入线程，线程可以被调度，只需要load/store再实现同步就可以实现共享。

- 一个进程中的多个线程共享了memory、global data、heap；不共享stack
  - 每个线程有自己的栈和PC
  - 线程和线程的栈不是隔离的，是可以直接访问到

Separate Policy and Mechanism

- Mechanism(机制): **how** question about a system | 怎么实现
    - How does an operating system performs a context switch

> 不同的调度器实现不同的Policy

- Policy(策略): **which** question
    - Which process should the process to be switched
- Any other examples about mechanism & policy?
- Advantages & Disadvantages
    - Advantages of separation:
        - 增加整个的灵活性.

另一个视角

- 程序运行时OS 在干什么
    - 使程序正常运行
- 允许程序使用、共享存储
    - 允许程序和外设进行沟通
- 在做“虚拟化” (Virtualization) 的事情
    - 给用户态程序提供虚拟的运行环境
    - 使用户态程序认为自己可以拥有全部 CPU 资源、IO 资源等
- 程序可以通过 syscall 来与 OS 打交道

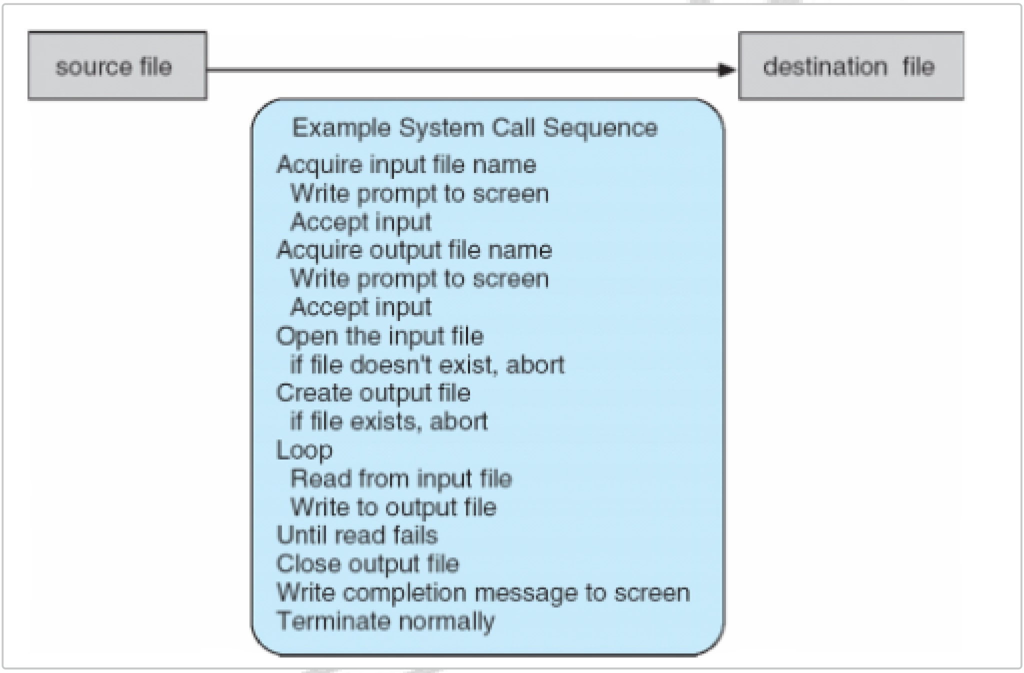
Structure

System Calls

- System call is a programming **interface**(接口) to access the OS services
    - Typically written in a high-level language (C or C++)
    - Certain low level tasks are in assembly languages

Example of System Calls

- `cp in.txt out.txt`

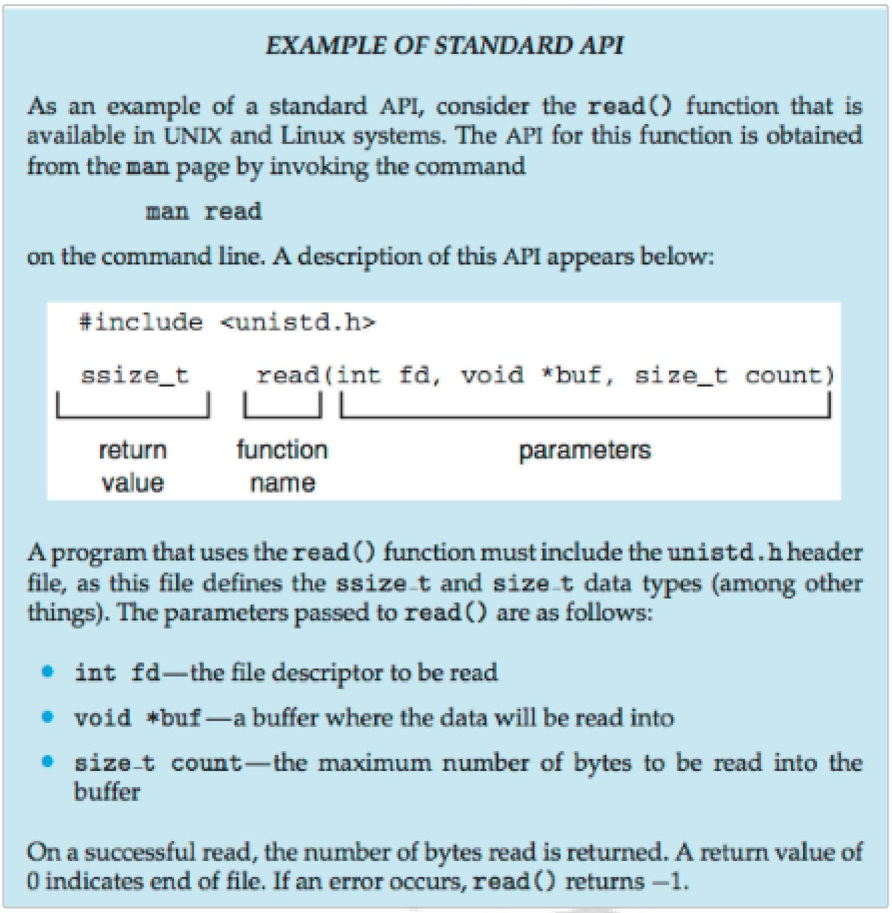


Application Programming Interface | API

- Mostly accessed by programs via a high-level Application (再由API去调用具体的系统调用) Programming Interface (API) rather than direct system call use
  - three most common APIs:
    - Win32 API for Windows
    - POSIX API for POSIX-based systems (UNIX/Linux, Mac OS X)
    - Java API for the Java virtual machine (JVM)
    - why use APIs rather than system calls?
      - portability

Example of Standard API

> 经常使用read这个API



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

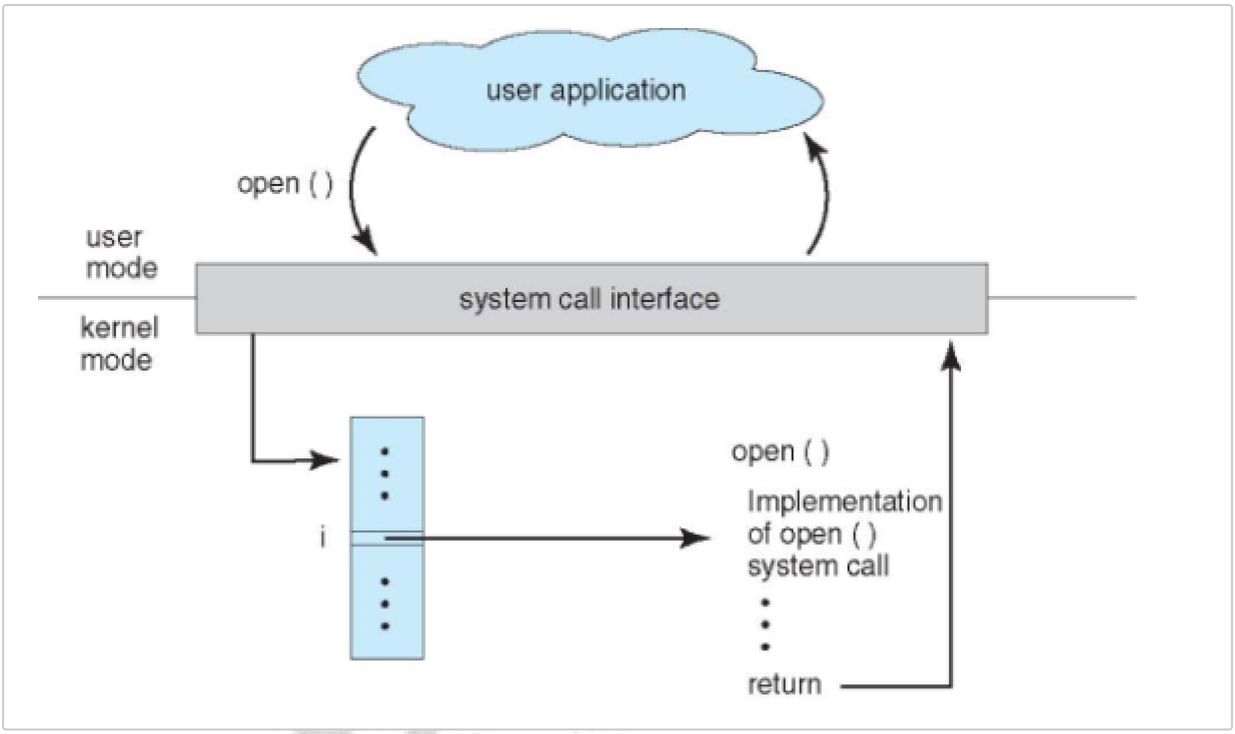
- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

System Calls Implementation

- Typically, a number is associated with each system call
  - system-call interface maintains a table indexed by these numbers
  - e.g., Linux has around 340 system call (x86: 349, arm: 345)
- Kernel invokes intended system call and returns results
- User program needs to know nothing about syscall details
  - it just needs to use API (e.g., in libc) and understand what the API will do
  - most details of OS interface hidden from programmers by the API

API – System Call – OS Relationship



System Call Parameter Passing

- Parameters are required besides the **system call number**(系统调用号)
  - exact type and amount of information vary according to OS and call
- Three general methods to pass parameters to the OS(传递参数)
  - Register:

    > 寄存器传参方式—通过一些通用的寄存器传参（速度快，数据数量和长度有限）

    - pass the parameters in registers

    > simple, but there may be more parameters than registers

  - Block:

    > 将参数放到memory中

    - parameters stored in a memory block (or table)
    - address of the block passed as a parameter in a register

    > taken by Linux and Solaris

  - Stack:

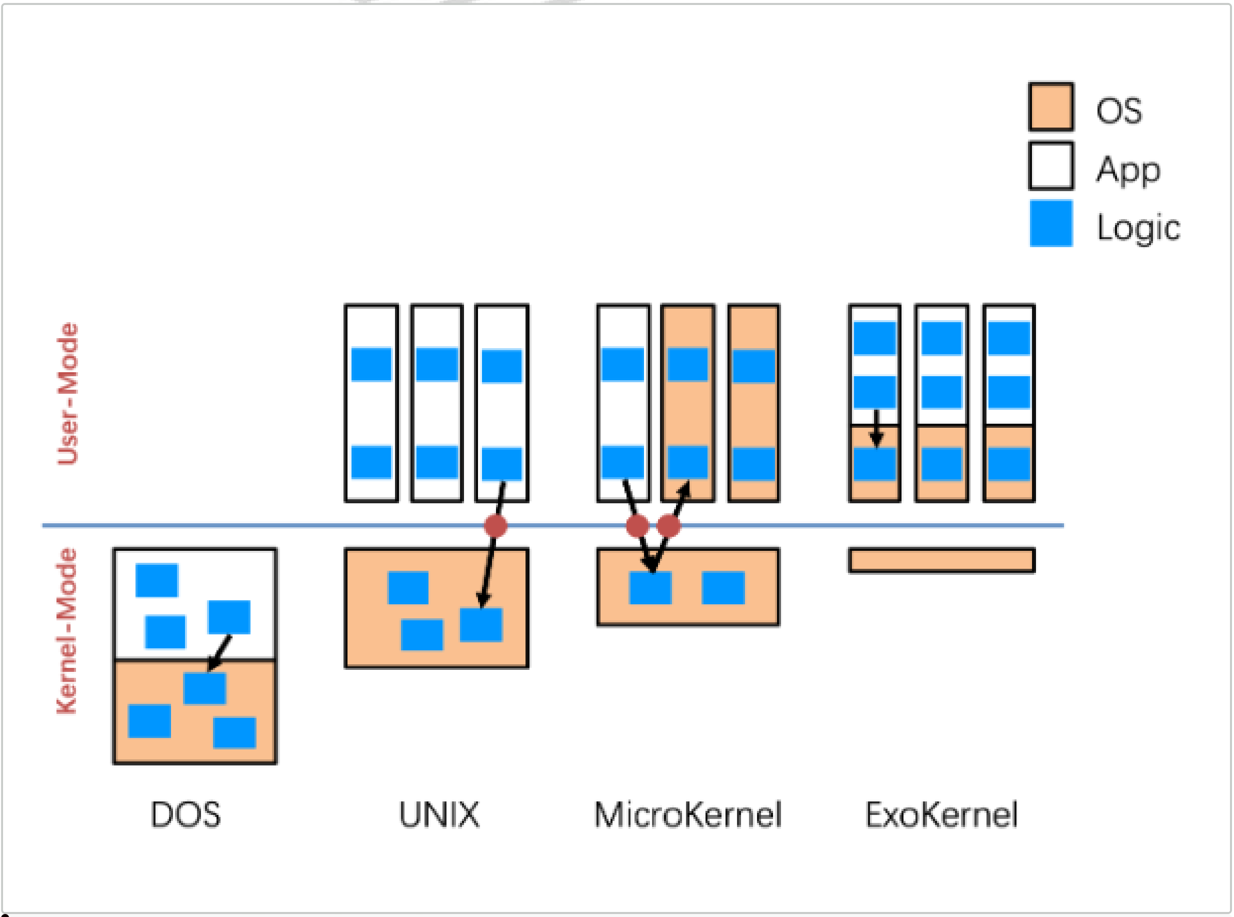
    > 将参数放在特殊的memory，可以通过pop直接获得

    - parameters placed, or pushed, onto the stack by the program
    - popped off the stack by the operating system
  - Block and stack methods don’t limit number of parameters being passed

Operating System Structure

- Many structures:
  - simple structure - MS-DOS(无结构，不存在隔离)
  - Monolithic structure(more complex， 整体内核) -- UNIX(分用户态内核态)
  - layered structure - an abstraction
  - microkernel system structure(微内核) - L4
  - hybrid: Mach, Minix
  - research system: exokernel

Comparison



> 微内核和整体内核的优缺点

Monolithic (整体) Structure - Original UNIX | 整体内核 **

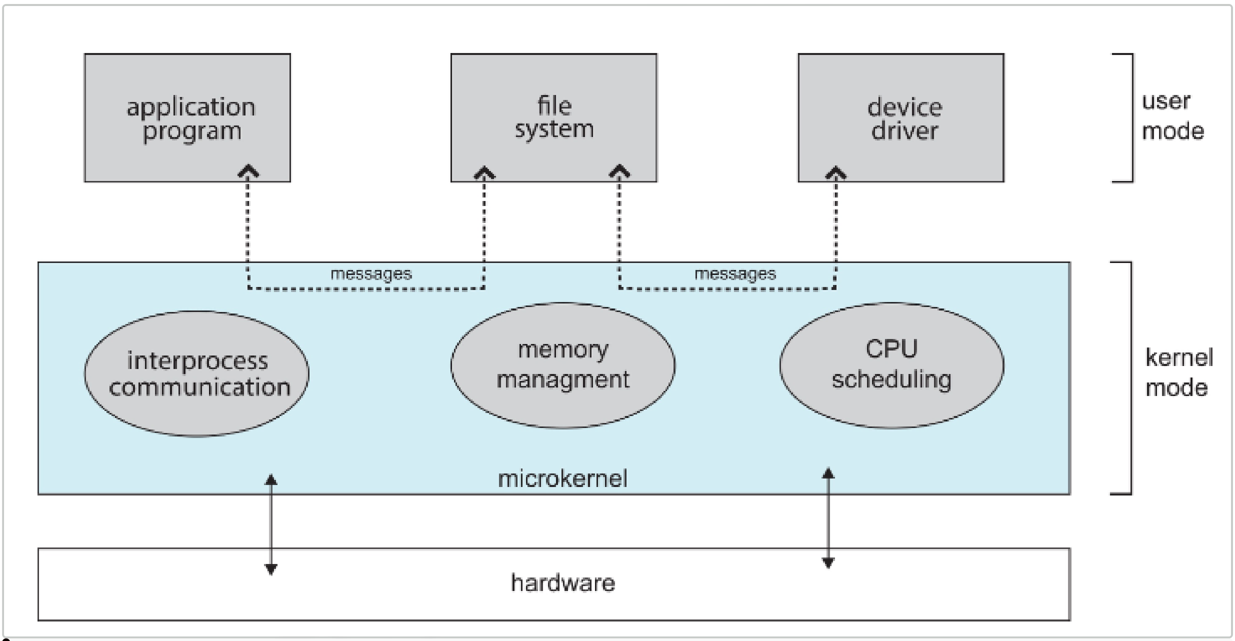
> 区分用户态和内核态，内核态在特权模式下通过系统调用给用户态提供接口

- Limited by hardware functionality, the original UNIX had limited structure
- UNIX OS consists of two separable layers
  - systems programs
  - the kernel: everything below the system-call interface and above physical hardware
    - a large number of functions for one level: file systems, CPU scheduling, memory management ...

Microkernel System Structure | 微内核 **

> 将很多内核态的component移到用户态，通过消息传递的方式进行交互

- Microkernel moves as much from the kernel (e.g., file systems) into “user” space
- Communication between user modules uses message passing
- Benefits :
  - easier to extend a microkernel | 容易扩展，微内核放在用户态，扩展内核像扩展一个用户态程序一样
  - easier to port the operating system to new architectures | 只需要将内核态的进行移植
  - more reliable (less code is running in kernel mode)
  - more secure | TCB (Trust Computer Base) 变小
- Detriments :
  - performance overhead of user space to kernel space communication | 性能开销大，原先的一些同态的 `function call` 现在变成跨用户态特权态的调用
- Examples: Minix, Mach, QNX, L4...

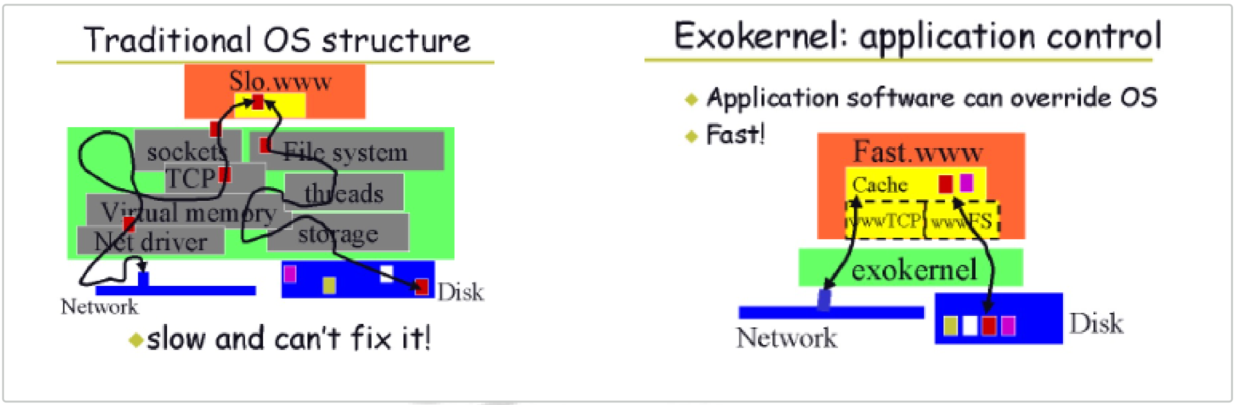


> 如今，整体内核可以通过增加内核模块（kernel module）的方式进行扩展。
> 但弊端是，因为都要在内核运行会导致内核的安全性降低，并且会对模块有严格的标准，比如API要一致

Exokernel: Motivation | 外内核 **

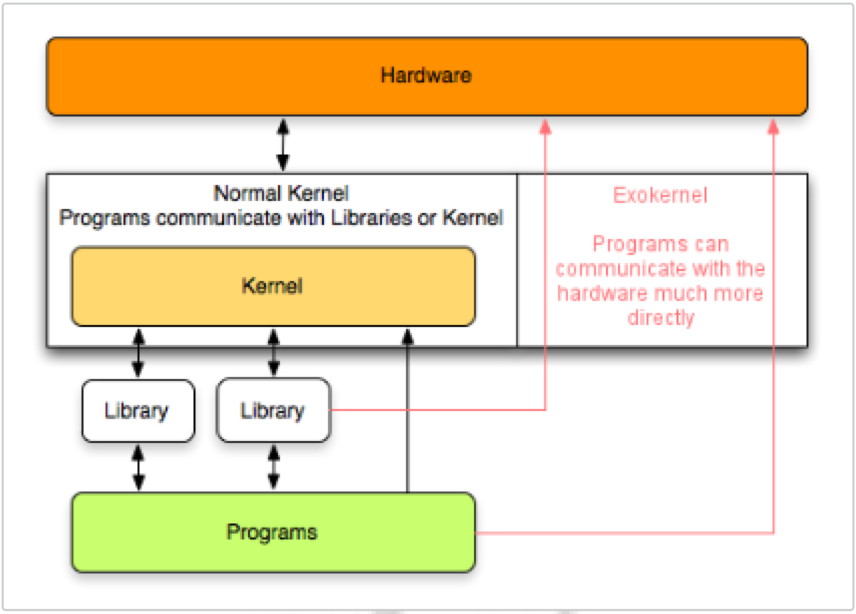
> 外内核只实现很薄的操作系统部分，怎么使用硬件资源的policy都由用户态决定

- In traditional operating systems, only privileged servers and the kernel can manage system resources
- Un-trusted applications are required to interact with the hardware via some abstraction model
  - File systems for disk storage, virtual address spaces for memory, etc.
- But application demands vary widely!!
  - An interface designed to accommodate every application must anticipate all possible needs



- Give un-trusted applications as much control over physical resources as possible
- To force as few abstraction as possible on developers, enabling them to make as many decisions as possible about hardware abstractions.

    - Let the kernel allocate the basic physical resources of the machine
    - Let each program decide what to do with these resources
- Exokernel separate protection from management
    - They protect resources but delegate management to application
- Exokernel give more direct access to the hardware, thus removing most abstractions


Hardware
Normal Kernel
Programs communicate with Libraries or Kernel
Kernel
Exokernel
Programs can communicate with the hardware much more directly
Library
Library
Programs

System Call **

Examples

fork

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int)getpid());
    int rc = fork();
    if(rc < 0)
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if(!rc) //child (new process)
    {
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    }
    else
    {
        printf("hello, I am parent of %d(pid:%d)\n", (int)getpid());
    }
    return 0;
}
```

- A way to create a new process
- Odd part
  - the newly created process is an exact copy of the calling process | 子进程会精确地拷贝一份父进程的代码
  - return twice
  - new process has its own memory address space and etc

> fork是调用一次，返回两次 | 一次父进程返回一次子进程返回.对于父进程返回值是子进程的pid；对子进程返回值是0

fork+wait

> wait指父进程等待子进程的完成

- Parent process can use the wait system call to wait the child process finishes executing

fork+wait+exec

- Exec is useful when you want to run a program that is different from the calling program
- Exec never returns
- Why separating fork and exec?
  - Essential building UNIX shell
- Shell: a user program
  - Wait for inputs
  - execute commands: fork, exec, and wait
  - Separating fork and exec can make the shell something interesting and useful — make something happen after fork but before exec

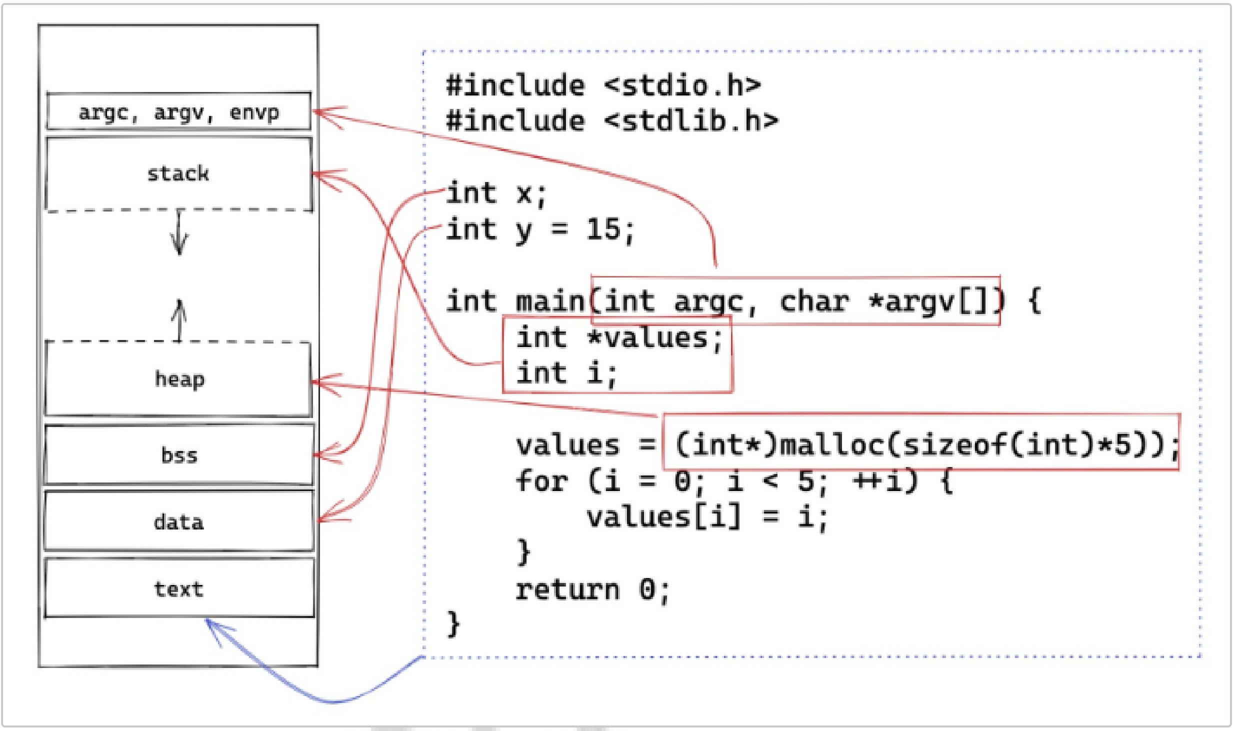
---

Process & Thread

Process

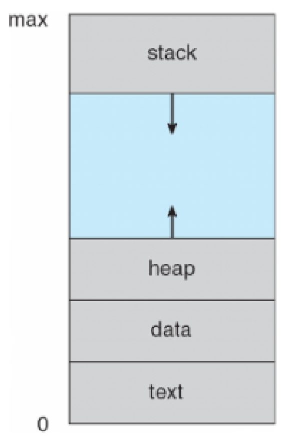
Process Concept **

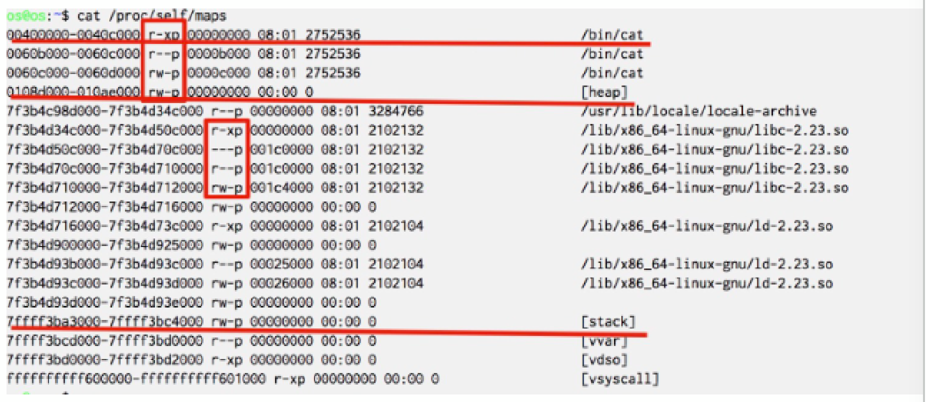
- An operating system executes a variety of programs:
  - batch system - jobs
  - time-shared systems - user programs or tasks
- **Process is a program in execution**, its execution must progress in sequential fashion
  - a program is static and passive, process is dynamic and active
  - one program can be several processes(e.g., multiple instances of browser, or even on instance of the program)
  - process can be started via GUI or command line entry of its name
    - through system calls
- A process has multiple parts:
  - the program code, also called **text section**
  - runtime CPU states, including program counter, registers, etc
  - various types of memory:
    - stack: temporary data | 临时的数据：局部变量、函数调用参数、返回地址等
      - e.g., function parameters, local variables, and return addresses
    - data section: global variables | 全局变量
    - heap: memory dynamically allocated during runtime | 运行时动态分配的内存（比如 malloc)
      - security: heap feng shui → how to provide randomness
      - Further reading: FreeGuard: A Faster Secure Heap Allocator (CCS 17), Guarder: A Tunable Secure Allocator (USENIX Sec 18)
- **通过 cat /proc/$pid/maps 可以查看进程的虚拟内存布局。**



```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[]) {
    int *values;
    int i;

    values = (int*)malloc(sizeof(int)*5);
    for (i = 0; i < 5; ++i) {
        values[i] = i;
    }
    return 0;
}
```

Process in Memory



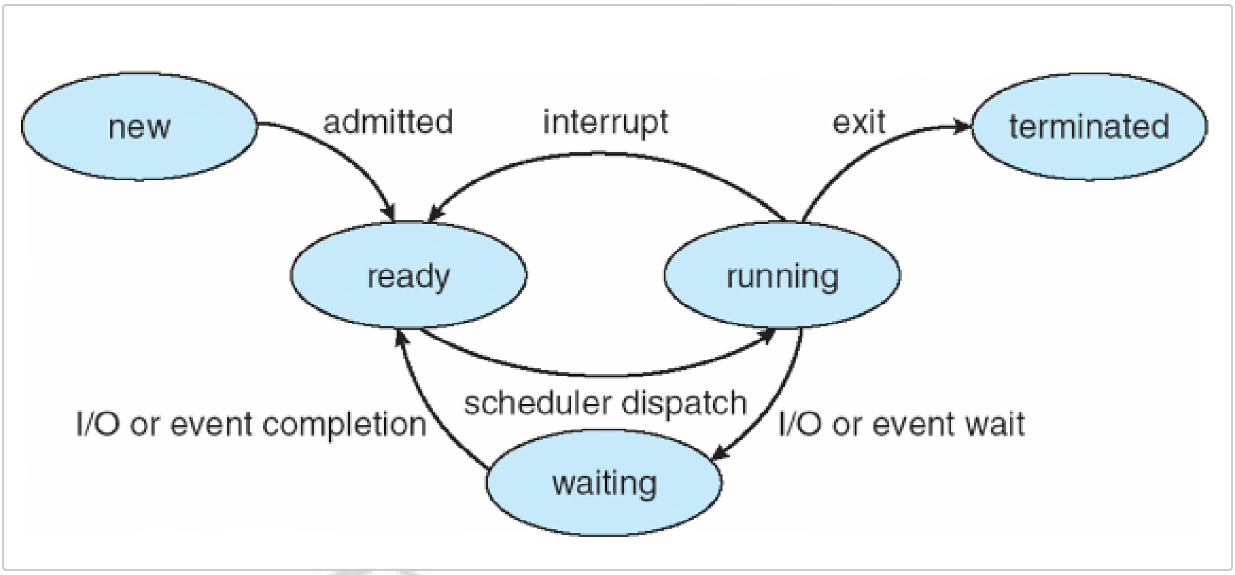


```
os@os:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 2752536                            /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 2752536                            /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 2752536                            /bin/cat
0108d000-010ae000 rw-p 00000000 00:00 0                                  [heap]
7f3b4c98d000-7f3b4d34c000 r--p 00000000 08:01 3284766                    /usr/lib/locale/locale-archive
7f3b4d34c000-7f3b4d50c000 r-xp 00000000 08:01 2102132                    /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d50c000-7f3b4d70c000 ---p 001c0000 08:01 2102132                    /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d70c000-7f3b4d710000 r--p 001c0000 08:01 2102132                    /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d710000-7f3b4d712000 rw-p 001c4000 08:01 2102132                    /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d712000-7f3b4d716000 rw-p 00000000 00:00 0
7f3b4d716000-7f3b4d73c000 r-xp 00000000 08:01 2102104                    /lib/x86_64-linux-gnu/ld-2.23.so
7f3b4d900000-7f3b4d925000 rw-p 00000000 00:00 0
7f3b4d93b000-7f3b4d93c000 r--p 00025000 08:01 2102104                    /lib/x86_64-linux-gnu/ld-2.23.so
7f3b4d93c000-7f3b4d93d000 rw-p 00026000 08:01 2102104                    /lib/x86_64-linux-gnu/ld-2.23.so
7f3b4d93d000-7f3b4d93e000 rw-p 00000000 00:00 0
7ffff3ba3000-7ffff3bc4000 rw-p 00000000 00:00 0                          [stack]
7ffff3bcd000-7ffff3bd0000 r--p 00000000 00:00 0                          [vvar]
7ffff3bd0000-7ffff3bd2000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Process State

> 掌握状态时是怎么迁移的以及当什么情况发生的时候会发生迁移

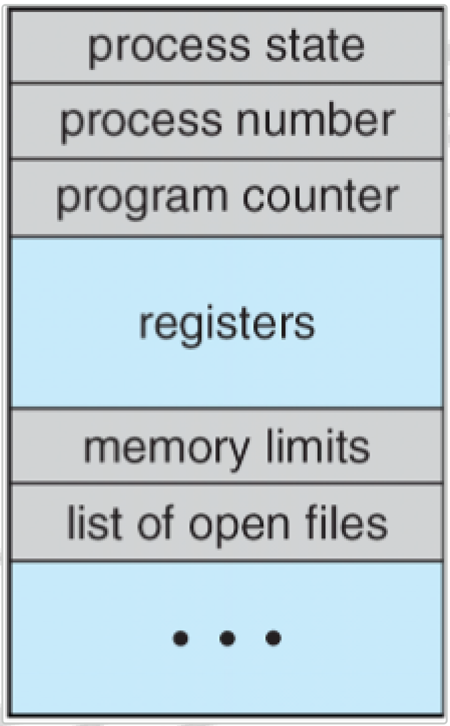
- As a process executes, it changes state
  - `new` : the process is being created
  - `running` : instructions are being executed
  - `waiting/blocking` : the process is waiting for some event to occur
  - `ready` : the process is waiting to be assigned to a processor
  - `terminated` : the process has finished execution



Process Control Block | PCB

> 保存整个进程状态的数据结构—PCB(在操作系统内核里维护每一个进程状态信息)

- In the kernel, each process is associated with a process control block
  - process number (pid)
  - process state
  - program counter (PC)
  - CPU registers
  - CPU scheduling information
  - memory-management data
  - accounting data
  - I/O status
- Linux's PCB is defined in struct task_struct: http://lxr.linux.no/linux+v3.2.35/include/linux/sched.h#L1221



Process Scheduling

- CPU scheduler(CPU调度器) selects which process should be executed next and allocates CPU
  - invoked very frequently, usually in milliseconds: it must be fast
- To maximize CPU utilization, kernel quickly switches processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Kernel maintains scheduling queues of processes(为了更好地调度，在内核中设置queue，调度器只需要在queue中选(queue已经拍好序，直接调出链表头就可以了)):
  - 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

Scheduler

- Scheduler needs to balance the needs of(进程分为下面两大类):
  - I/O-bound process | 大部分时间在做I/O请求
    - spends more time doing I/O than computations
    - many short CPU bursts
  - CPU-bound process | 大部分时间在做CPU计算
    - spends more time doing computations

            - few very long CPU bursts

Context Switch

- Context switch | 上下文切换: the kernel switches to another process for execution
- save the state of the old process
- load the saved state for the new process

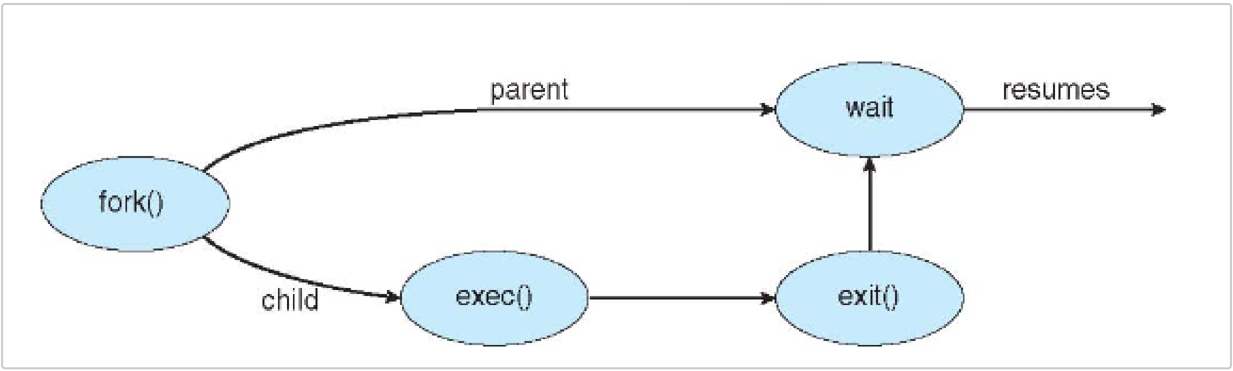
> 保存旧进程的状态，加载新进程的状态

- Context-switch is overhead; CPU does no useful work while switching
    - the more complex the OS and the PCB, longer the context switch
- Context-switch time depends on hardware support
    - some hardware provides multiple sets of registers per CPU: multiple contexts loaded at once

Process Creation

> 通过fork系统调用创建

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
    - process identified and managed via a process identifier (pid)
- Design choices:
    - three possible levels of resource sharing: all, subset, none
    - parent and children's address spaces
        - child duplicates parent address space (e.g., Linux)
        - child has a new program loaded into it (e.g., Windows)
    - execution of parent and children
        - parent and children execute concurrently
        - parent waits until children terminate
- UNIX/Linux system calls for process creation
    - fork creates a new process
    - exec overwrites the process' address space with a new program
    - wait waits for the child(ren) to terminate



Zombie vs Orphan

- When child process terminates, it is still in the process table until the parent process calls wait() | 取决于wait
  - zombie | 僵尸: child has terminated execution, but parent did not invoke wait() (子进程执行结束，父进程还存在且未调用wait)
    - 这时子进程的相关资源不会被释放
    - 只有手动kill掉父进程才会释放
      - 即让僵尸进程变为孤儿进程，从而被 init 进程接管、释放
      - kill 掉僵尸进程无效，因为进程已经死了
  - orphan | 孤儿: parent terminated without invoking wait. Systemd will take over. Systemd will call wait() periodically
    - 子进程会被 init 进程 (pid 1) 接管
      - 例如 systemd 会通过 wait 来等待孤儿进程结束

Process Termination

- Process executes last statement and asks the kernel to delete it (exit)
  - OS delivers the return value from child to parent (via wait)
  - process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort), for example:
  - child has exceeded allocated resources
  - task assigned to child is no longer required
  - if parent is exiting, some OS does not allow child to continue
    - all children (the sub-tree) will be terminated - cascading termination

Interprocess Communication **

- Processes within a system may be independent or cooperating
    - independent process: process that cannot affect or be affected by the execution of another process
    - cooperating process : processes that can affect or be affected by other processes, including sharing data
        - reasons for cooperating processes: information sharing, computation speedup, modularity, convenience, Security
- Cooperating processes need interprocess communication (IPC) | 协作的进程通过IPC交流
- Two models of IPC
    - Shared memory
    - Message passing

Remote Procedure Call | RPC *

- Remote procedure call (RPC) abstracts function calls between processes across networks (or even local processes)
- Stub:a proxy for the actual procedure on the remote machine
    - client-side stub locates the server and marshalls the parameters
    - server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
    - return values are marshalled and sent to the client

Thread

Motivation

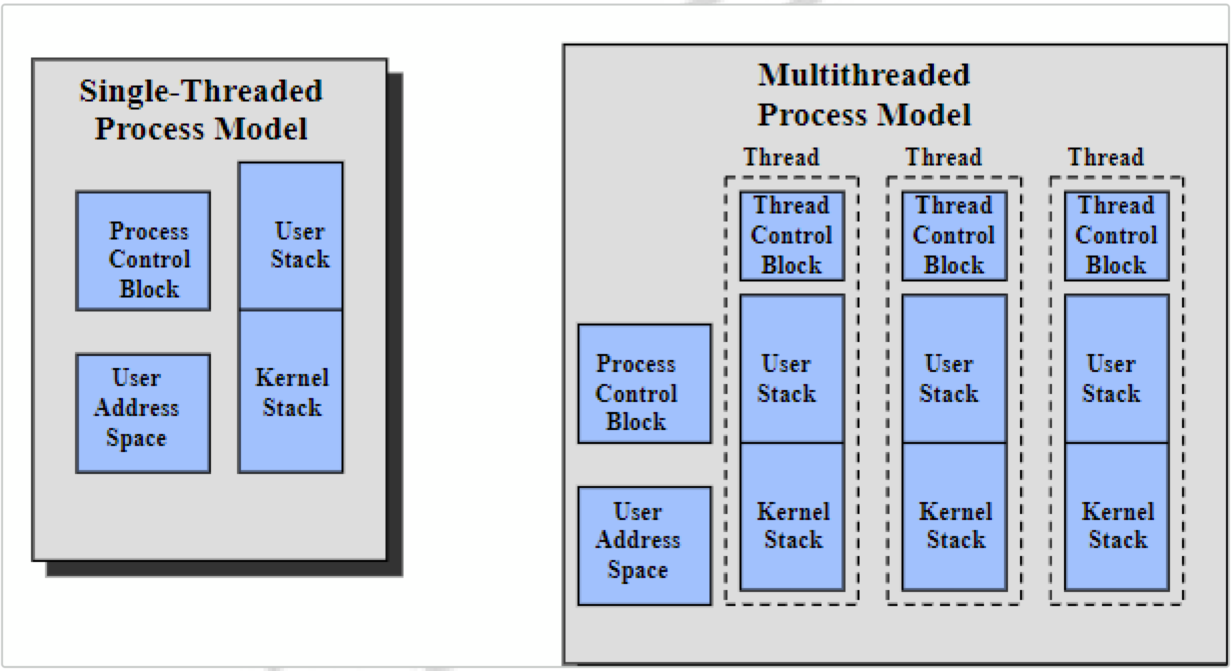
- Why threads?
    - multiple tasks of an application can be implemented by threads | 把不同任务变成进程从而做到并发提高吞吐率，但**进程开销较大且进程间不共享数据。**因此可以将不同任务变成线程，可共享数据又可并发
        - e.g., update display, fetch data, spell checking, answer a network request
    - process creation is heavy-weight while thread creation is light-weight - why?
    - threads can simplify code, increase efficiency
- Kernels are generally multithreaded

What is Thread

- A thread is an independent stream of instructions that can be scheduled to run as such by the kernel
- Process contains many states and resources
    - code, heap, data, file handlers (including socket), IPC
    - process ID, process group ID, user ID
    - stack, registers, and program counter (PC)
- Threads exist within the process, and shares its resources
    - each thread has its own essential resources (per-thread resources): stack, registers, program counter, thread-specific data……
    - access to shared resources need to be synchronized
- Threads are individually scheduled by the kernel
    - each thread has its own independent flow of control
    - each thread can be in any of the scheduling states

> 每个线程虽然有不同的pre-thread resources,但数据不是隔离的

Thread and Process



Implementing Threads

- Thread may be provided either at the user level, or by the kernel
    - user threads are supported above the kernel and managed **without kernel support**
        - three thread libraries: POSIX Pthreads, Win32 threads, and Java threads

    - kernel threads are supported and managed directly by the kernel
        - all contemporary OS supports kernel threads

Thread Benefits

- Responsiveness
    - multithreading an interactive application allows a program to continue running even part of it is blocked or performing a lengthy operation
- Resource sharing
    - sharing resources may result in efficient communication and high degree of cooperation. Threads share the resources and memory of the process by default.
- Economy
    - thread is more lightweight than processes: create and context switch
- Scalability
    - better utilization of multiprocessor architectures: running in parallel

Multithreading Models

- A relationship must exist between user threads and kernel threads
    - Kernel threads are the real threads in the system, so for a user thread to make progress the user program has to have its scheduler take a user thread and then run it on a kernel thread.
- 包括:
    - Many-to-One Model
    - One-to-One Model | 最常见的
    - Many-to-Many Model
    - Two-Level Model

Threading Issues

- Semantics of fork and exec system calls | fork和exec的语义(fork之后duplicate哪些资源, exec执行哪一部分)
- Signal handling
- Thread cancellation of target thread
- Thread-specific data
- Scheduler activations

CPU Schedule

> 实际上调度的最小基本单元是线程调度，但算法可以混用进程和线程调度

Some Terms

- Kernel threads - not processes - are being scheduled by the OS
- However, “thread scheduling” and “process scheduling” are used interchangeably.
- We use “process scheduling” when discussing general ideas and “thread scheduling” to refer thread-specific concepts
- Also “run on a CPU” → run on a CPU’s core

Basic Concepts

- Process execution consists of a cycle of CPU execution and I/O wait
  - CPU burst: 大部分时间在做 CPU 运算
    - 不同进程、不同计算机的 CPU burst 长度差别很大
  - I/O burst : 大部分时间在做 I/O
    - 进程大部分都是 I/O burst
  - CPU burst distribution varies greatly from process to process, and from computer to computer, but follows similar curves
- Maximum CPU utilization obtained with multiprogramming
  - 当前进程在 I/O burst 时会调度到另一个进程进行执行 | CPU scheduler selects another process when current one is in I/O burst

CPU Scheduler

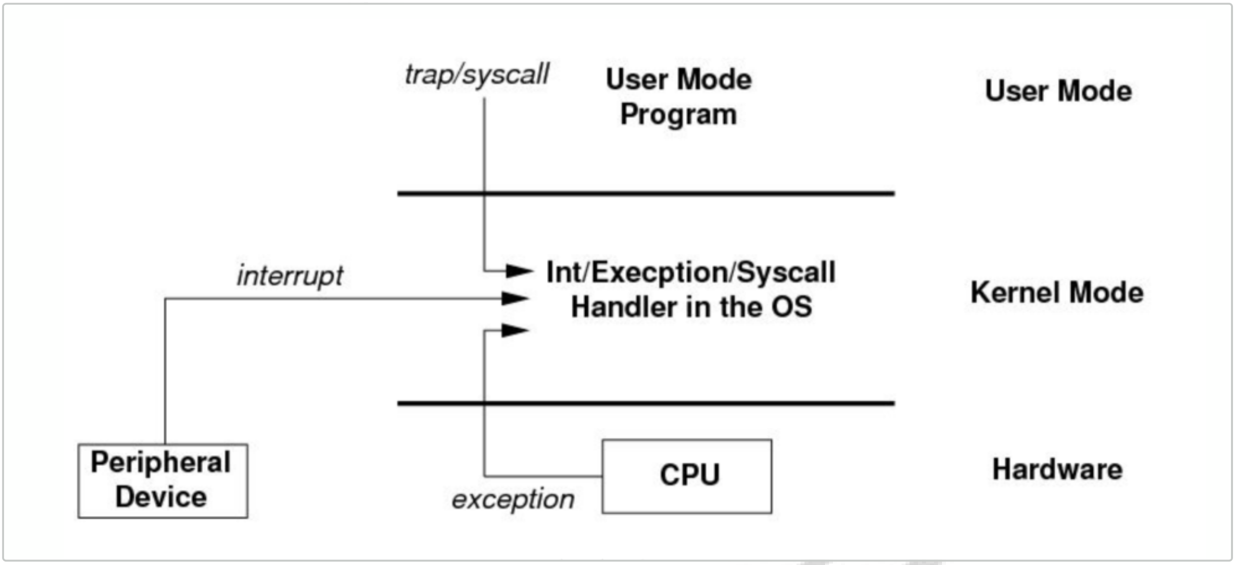
- CPU Scheduler 是操作系统内核中负责调度的部件，工作方式是从 ready queue 中选择一个进程来将 CPU 资源分配给它 | CPU scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. switches from running to waiting state (e.g., wait for I/O)
  2. switches from running to ready state (e.g., when an interrupt occurs)
  3. switches from waiting to ready (e.g., at completion of I/O)
  4. terminates
- Scheduling under condition 1 and 4 only is nonpreemptive (非抢占式的)
  - 使用非抢占式调度的情况下，一旦一个程序被分配了 CPU 资源，则会一直执行，直到结束或者等待 I/O | once the CPU has been allocated to a process, the process keeps it until terminates or waiting for I/O

- also called cooperative scheduling(协作式调度)

- Preemptive scheduling schedules process also in condition 2 and 3
  - 需要硬件支持，例如计时器 | preemptive scheduling needs hardware support such as a timer
  - 需要一些同步的元语（synchronization primitives）| synchronization primitives are necessary

When

- The scheduling happens when the CPU is in kernel model, either because of hardware interrupt of software interrupt (e.g.,system call)



Preemption | 抢占

- Preemption: involuntarily suspending a running process is called preemption
- cooperative multitasking os: a process runs until it voluntarily stops or waits for IO

> 协作式也叫非抢占

- preemptive multitasking os: scheduler running in the kernel space to switch between processes

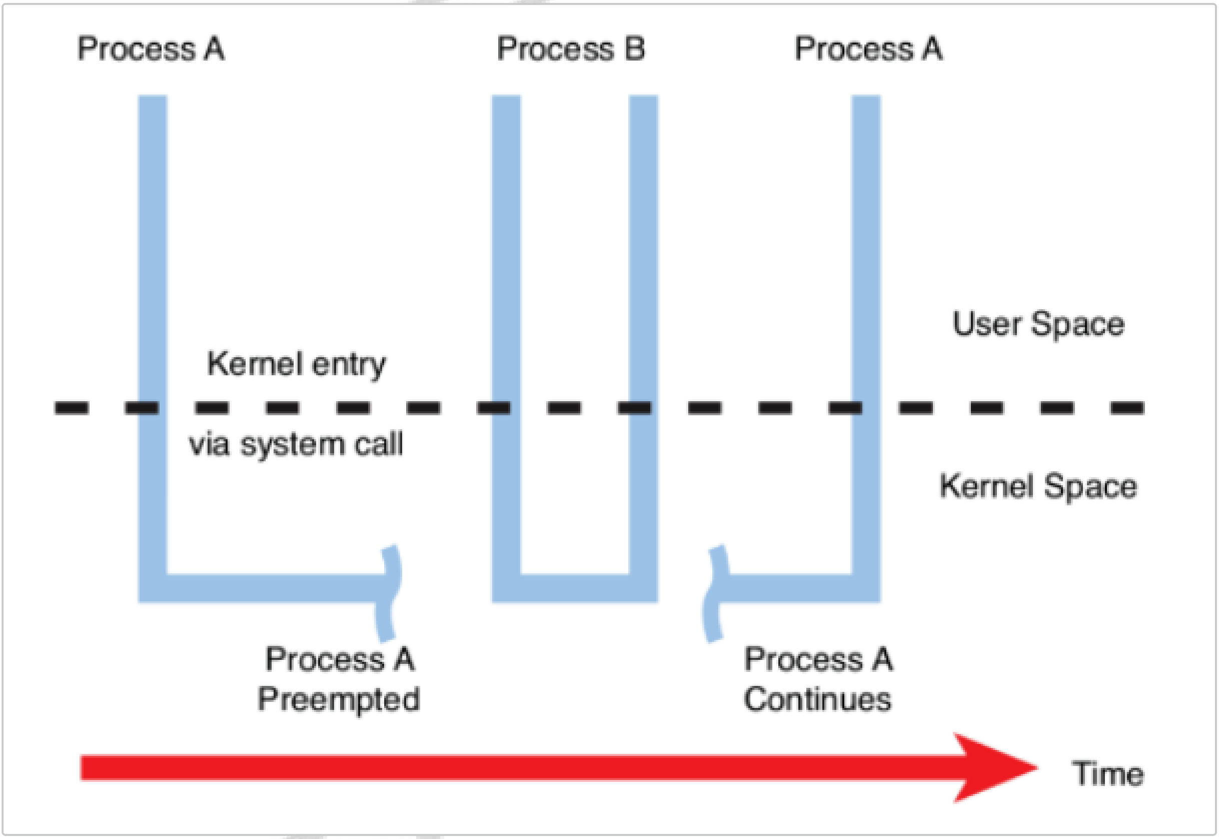
User Preemption | 用户态抢占

- User preemption
  - When returning to user-space from a system call
  - When returning to user-space from an interrupt handler

> 非内核抢占的情况下，用户态是不是可以抢占?
>
> √，因为内核抢占和用户态抢占的不同只是体现在调度的时间点不同，即使在非内核抢占的情况下仍然可以在从内核态返回到用户态的时候做调度的。

Kernel Preemption

- 如果用户态在执行时进行了系统调用，此时在内核态执行代码，如果此时发生了中断，也有两种情况：
- 处理结束后仍然返回到被打断前的程序，继续完成系统调用
    - 此时就称为内核非抢占 (kernel nonpreemption)
    - 但是仍然会发生用户态抢占
        - 可能在返回用户态时发生抢占，切换到另一个进程
- 处理结束后返回到调度器，选择了一个更高优先级的进程进行执行
    - 此时就称为内核抢占 (kernel preemption)
- When an interrupt handler exits, before returning to kernel-space
- When kernel code becomes preemptible again
- If a task in the kernel explicitly calls schedule()
- If a task in the kernel blocks (which results in a call to schedule())



- Preemption also affects the OS kernel design
    - kernel states will be inconsistent if preempted when updating shared data
    - i.e., kernel is serving a system call when an interrupt happens
- Two solutions:
    - waiting either the system call to complete or I/O block
        - kernel is nonpreemptive (still a preemptive scheduling for processes!)
    - disable kernel preemption when updating shared data
        - recent Linux kernel takes this approach:

        - Linux supports SMP
        - shared data are protected by kernel synchronization
        - disable kernel preemption when in kernel synchronization
        - turned a non-preemptive SMP kernel into a preemptive kernel

Dispatcher

- Dispatcher 是操作系统内核中负责进程切换的部件：
    - 切换上下文
    - 切换到用户态
    - 跳转到正确的位置恢复执行
- Dispatch latency：dispatcher 进行切换时的花费的时间（从暂停一个进程到恢复另一个进程）

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler
    - switching context
    - switching to user mode
    - jumping to the proper location in the user program to restart that program
- Dispatch latency：the time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria | 调度准则

- CPU utilization：percentage of CPU being busy
- Throughput：# of processes that complete execution per time unit
- Turnaround time：the time to execute a particular process
    - from the time of submission to the time of completion
- Waiting time：the total time spent waiting in the ready queue
- Response time：the time it takes from when a request was submitted until the first response is produced
    - the time it takes to start responding

在调度的时候进行选择时需要考虑的因素

- CPU 利用率（utilization）：CPU 资源的利用率，越大越好
- 吞吐量（throughput）：单位时间内完成的工作量，越多越好
- 周转时间（turnaround time）：从提交一个作业到完成该作业所需要的时间
- 等待时间（waiting time）：在 ready queue 中等待的总时间
- 响应时间（response time）：从用户提交请求到得到第一次响应所需要的时间

调度算法优化时希望更高的 CPU 利用率、吞吐量，更短的周转时间、等待时间、响应时间。

- 大部分情况是优化平均值
- 一些情况下想要优化最小 / 最大值
  - 例如实时系统，必须要保证在一定时间内完成
- 对于交互系统，希望响应时间的变化尽可能小

Scheduling Algorithm Optimization Criteria

> 抢占式和非抢占式区别就是当一个新的进程来的时候，是否需要rescheduling(重新分配CPU资源)：
> 抢占式：不管进程是否结束或有I/O，当一个新的进程来的时候要做重新调度
> 非抢占式：只有一个进程结束主动出让CPU资源或发I/O请求的时候才会让出CPU资源

- Generally, maximize CPU utilization and throughput, and minimize turnaround time, waiting time, and response time
- Different systems optimize different values
  - in most cases, optimize average value
  - under some circumstances, optimizes minimum or maximum value
    - e.g., real-time systems
  - for interactive systems, minimize variance in the response time

First-come,first-served scheduling (FCFS)

- FCFS (First Come First Served) : 先来先服务
  - 是一种非抢占式调度
  - waiting time 即将所有等待时间加起来除以总任务数
  - burst time指进程的运行时间

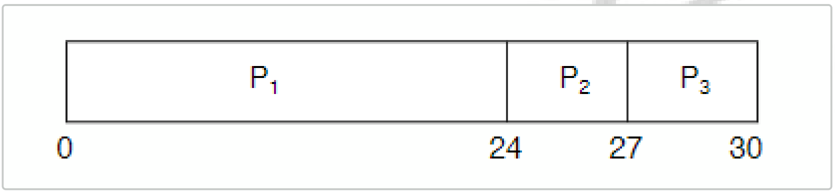
> 会画图，会算 `waiting time` 和 `average waiting time`

- Example processes:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3

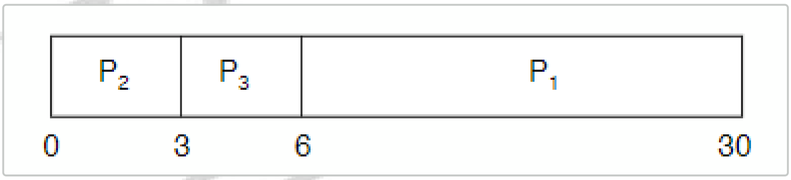
- the Gantt Chart for the FCFS schedule is:



- Waiting time for P$_1$ = 0; P$_2$ = 24; P$_3$ = 27, average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling | 非抢占式调度

- Suppose that the processes arrive in the order: P$_2$, P$_3$, P$_1$
  - the Gantt chart for the FCFS schedule is:



- Waiting time for P$_1$ = 6; P$_2$ = 0; P$_3$ = 3, average waiting time: (6 + 0 + 3)/3 = 3
- Convoy effect: all other processes waiting until the running CPU-bound process is done
  - considering one CPU-bound process and many I/O-bound processes

Shortest-Job-First Scheduling

> 最优的一种调度算法

SJF (Shortest Job First) : 最短作业优先

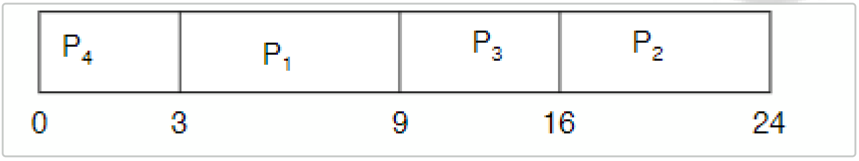
- 下一个 CPU burst 最短的进程优先执行
- 是可证明的最优（平均等待时间最短的）调度算法
- SJF 可以是抢占式的，也可以是非抢占式的
  - 抢占式的 SJF 又称为 shortest-remaining-time-first 调度算法
- 难点是如果预测 CPU burst 的长度
  - 假设和历史 CPU burst 相关
  - 通过 exponential averaging 来预测
  - $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$，其中：
    - τ_{n+1}：预测的下一个 CPU burst 长度
    - τ_n：第 n 次预测的 CPU burst
    - t_n：第 n 次实际的 CPU burst
    - α：平滑因子，表示历史预测的权重
  - 越老的历史占的权重越小，新的预测占的权重越大

- Associate with each process: the length of its next CPU burst
  - the process with the smallest next CPU burst is scheduled to run next
- SJF is provably optimal : it gives minimum average waiting time for a given set of processes
  - moving a short process before a long one decreases the overall waiting time
  - the difficulty is to know the length of the next CPU request
    - long-term scheduler can use the user-provided processing time estimate
    - short-term scheduler needs to approximate SFJ scheduling
- SJF can be preemptive or nonpreemptive
  - preemptive version is called shortest-remaining-time-first

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

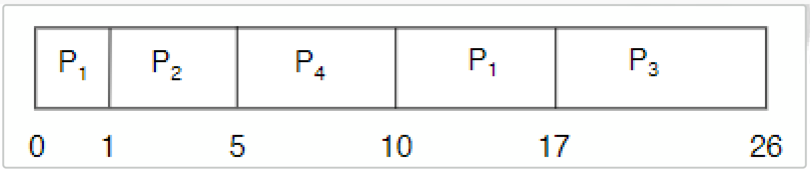
Shortest-Remaining-Time-First

- SJF can be preemptive: reschedule when a process arrives

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart

> 要求会画



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Priority Scheduling

优先级调度算法 (priority scheduling) : 根据优先级来调度

- 为每一个进程评估一个优先级
- 最高优先级的进程优先进行
- SJF

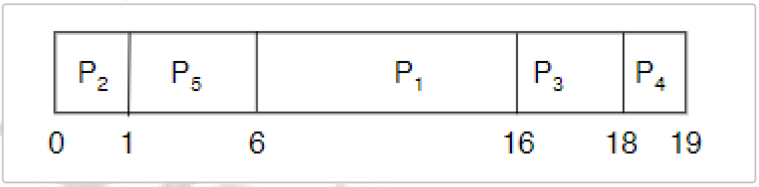
  是优先级调度的一种特例
  - 即令预测下一个 CPU burst 最短的进程优先级更高
- 同样可以是抢占式的也可以是非抢占式的
- 优先级调度算法的弊端是可能会导致"饥饿" (starvation)
  - 即低优先级的进程可能永远不会被执行
  - 解决方案是引入 aging，等待时间越长，优先级会提高

> - Priority scheduling selects the ready process with highest priority
>   - a priority number is associated with each process, smaller integer, higher priority
>   - the CPU is allocated to the process with the highest priority
>   - SJF is special case of priority scheduling
>   - priority is the inverse of predicted next CPU burst time
> - Priority scheduling can be preemptive or nonpreemptive, similar to SJF
> - Starvation is a problem: low priority processes may never execute
>   - Solution: aging — gradually increase priority of processes that wait for a long time

Example of Priority Scheduling

ProcessA	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

> We use small number to denote high priority.

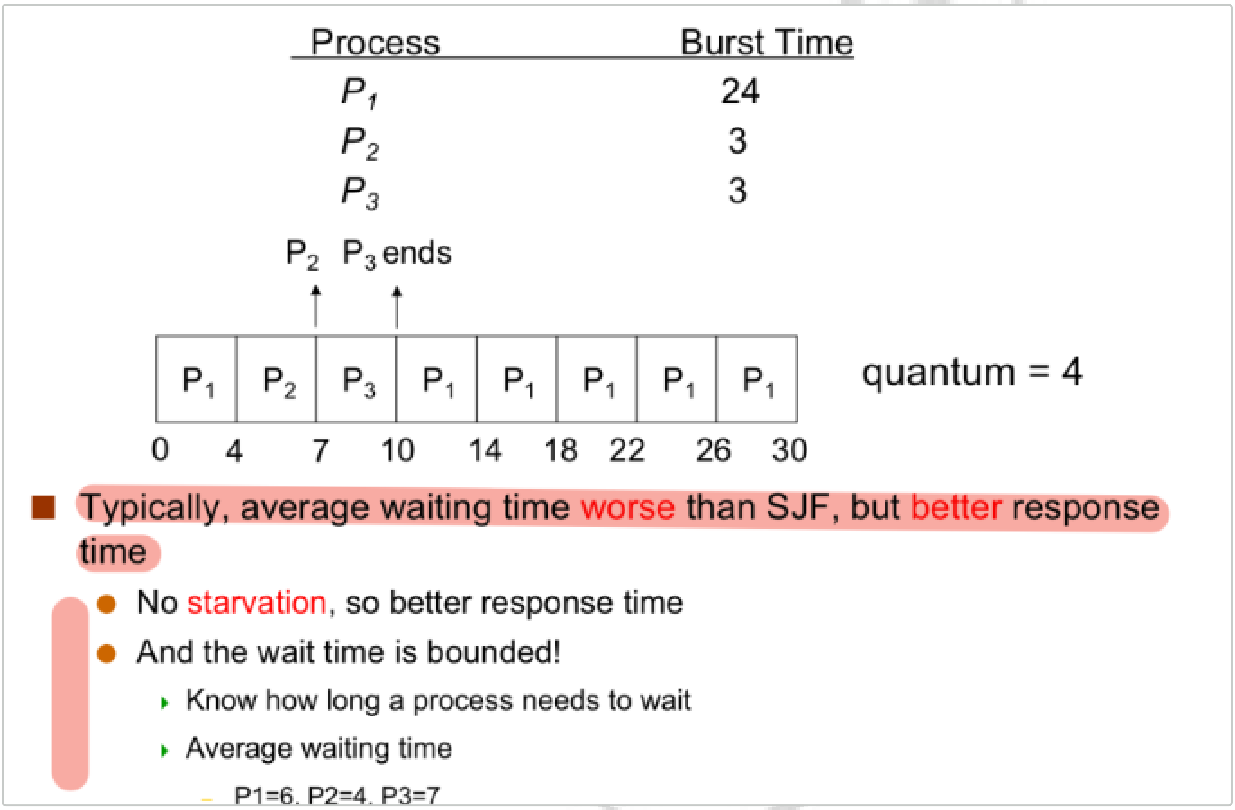
Round Robin (RR) | 时间片轮转

Round Robin (RR) : 时间片轮转

- 每一个进程都有一个时间片 (time quantum) q
- 在时间片用完之后会被切换到下一个进程
    - 也就是说，每个进程都会被执行 q 时间
- 时间片 q 的大小会影响到性能
    - q 太大，相当于退化到 FCFS
    - q 太小，切换过于频繁，上下文切换的开销过大
    - 通常是 10-100 ms
- 一定是抢占式的调度

- Round-robin scheduling selects process in a round-robin fashion
    - each process gets a small unit of CPU time (time quantum, q)
        - q is too large → FIFO, q is too small → context switch overhead is high
        - a time quantum is generally 10 to 100 milliseconds
    - process used its quantum is preempted and put to tail of the ready queue
        - a timer interrupts every quantum to schedule next process
- Each process gets 1/n of the CPU time if there are n processes
    - no process waits more than (n-1)q time units
    - Example: 5 processes with 20 ms time unites, then every process cannot et more than 20ms per 100ms

Example of Round-Robin



Process	Burst Time
P_1	24
P_2	3
P_3	3

- Typically, average waiting time worse than SJF, but better response time
  - No starvation, so better response time
  - And the wait time is bounded!
    - Know how long a process needs to wait
    - Average waiting time
      - P1=6, P2=4, P3=7

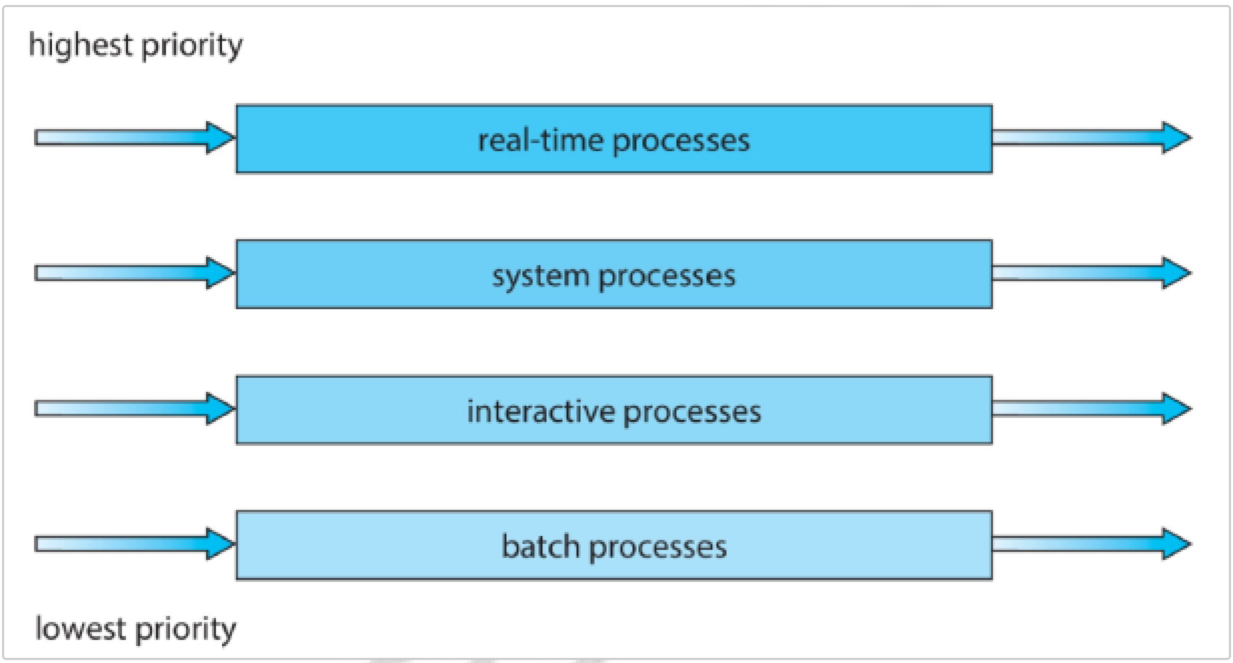
Multilevel Queue

Multilevel Queue Scheduling **

Multi-level Queue Scheduling: 多级队列调度算法

- ready queue 被分成多个队列
  - 比如分成交互性队列和批处理队列
- 一个进程会被永久地分到一个队列中
- 每个队列可以有自己的调度算法
  - 例如，高交互性的进程队列可以使用 RR 算法，低交互的批处理队列可以使用 FCFS 算法
- 队列之间也需要进行调度
  - 可以是固定优先级的调度，但是会导致饥饿
  - 可以通过时间片轮转来调度

在实际应用中，进程通常被分为不同的组，每个组有一个自己的 ready queue，且每个队列内部有自己独立的调度算法。例如，前台队列使用 RR 调度以保证 response，后台队列可以使用 FCFS。同时，队列之间也应当有调度。通常使用 preemptive priority scheduling，即当且仅当高优先级的队列（如前台队列）为空时，低优先级的队列（如后台队列）中的进程才能获准运行。也可以使用队列间的 time-slicing，例如一个队列使用 80% 的时间片而另一个使用 20%。例如：

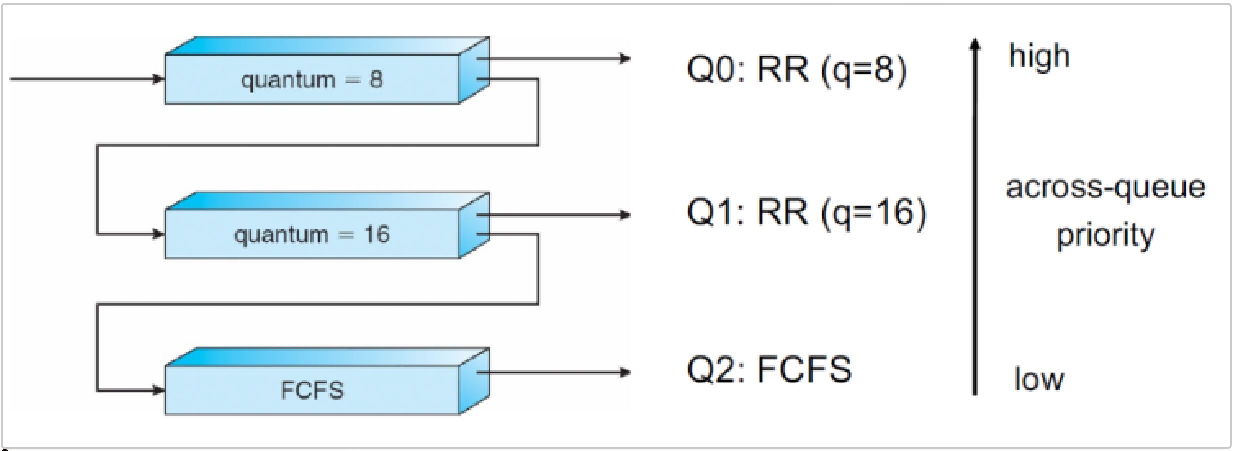


Multilevel Feedback Queue

Multi-level Feedback Queue Scheduling：多级反馈队列调度算法

- 和多级队列调度的区别是一个进程可以在不同的队列之间进行迁移
- 尝试推测进程的性质
    - 即是交互性的还是批处理的
- aging 也可以通过这种方式来实现
- 目的是来给交互性、I/O intensive 的进程更高的调度优先级
- 是最通用的调度算法

Multilevel Feedback Queue Scheduling 允许进程在队列之间迁移。这种算法可以有很多种实现，因为队列的数量、每个队列中的调度策略、队列之间的调度算法以及将进程升级到更高优先级/降级到更低优先级的队列的条件都是可变的。一个系统中的最优配置在另一个系统中不一定很好。这种算法也是最为复杂的。
看这样一个例子：有三个队列 0, 1, 2，优先级逐次降低。当进程 ready 时被添加到 Q0 中，Q0 内部采用 RR Scheduling，的每个进程都有 8ms 的时间完成其运行，如果没有完成则被打断并进入 Q1；只有当 Q0 为空时 Q1 才可能被运行。Q1 内部也使用 RR Scheduling，每个进程有 16ms 时间完成其运行，如果没有完成则被打断并进入 Q2；只有当 Q1 也为空时 Q2 才可能被运行。Q2 内部采用 FCFS 算法。



> 几个调度算法，会算average waiting time

Thread Scheduling **

- OS kernel schedules kernel threads
    - system-contention scope (SCS): competition among all threads in system
    - kernel does not aware user threads
- Thread library schedules user threads onto LWPs
    - used in many-to-one and many-to-many threading model
    - process-contention scope (PCS): scheduling competition within the process
    - PCS usually is based on priority set by the user
    - user thread scheduled to a LWP do not necessarily running on a CPU
        - OS kernel needs to schedule the kernel thread for LWP to a CPU

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
    - pthread_attr_set/getscope is the API
        - PTHREAD_SCOPE_PROCESS: schedules threads using PCS scheduling : number of LWP is maintained by thread library
        - PTHREAD_SCOPE_SYSTEM: schedules threads using SCS scheduling
- Which scope is available can be limited by OS
    - e.g., Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Multiple-Processor Scheduling

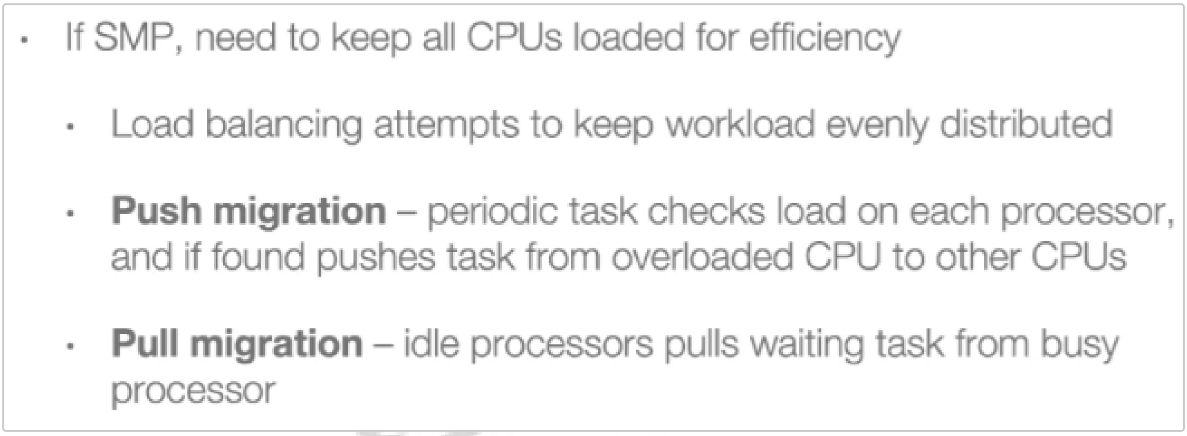
- 两种多处理器调度方式
    - asymmetric multiprocessing
        - 只有一个处理器用来决策调度，处理 I/O，以及其他的活动
        - 其它处理器作为执行单元

    - symmetric multiprocessing (SMP)
        - 每个处理器都在做自己的调度
        - 所有线程可能都在同一个 ready queue 中（所有 core 共享），也有可能不同 core 有自己的 ready queue
        - 常见的操作系统用的都是这种方式
- 硬件线程 chip multitreading (CMT) 技术
    - 在一个 CPU core 中运行两个硬件线程
    - 两个硬件线程共享 CPU 的执行单元，但是有自己的上下文
    - 依靠 memory stall 来实现
        - 访问 memory 是较慢的
        - 在一个硬件线程访问 memory 的时候可以调度另一个硬件线程执行
    - 每个处理器的逻辑 CPU 数 = 处理器中核心数 * 每个核心中硬件线程个数
    - 两级调度
        - 操作系统决策哪个软件线程在逻辑 CPU 上运行
        - CPU 决定如何在物理 core 上运行硬件线程
            - 例如避免将两个计算密集型的线程放在同一个 core 上（这样 memory stall 的概率会减小）
- Load Balancing 负载均衡
    - SMP 情境下让所有 CPU 的负载更加平均
    - 两种方法
        - push migration：周期性检查所有处理器上的负载情况，发现空闲的则将任务 push 过去
        - pull migration：空闲的处理器主动从繁忙的处理器上拉来任务
- Processor Affinity 处理器亲核性
    - 当一个线程在一个 CPU 上运行一段时间之后，cache 内容会被当前线程填充
    - 如果为了负载均衡而将这个线程 migrate 到其它处理器上，会因为 cache miss 导致效率下降
    - 两种机制
        - soft affinity：操作系统尽可能保持线程在同一个处理器上运行，但不保证（例如负载实在不均衡）
        - hard affinity：强制规定一个线程只能在哪些处理器上运行
- NUMA 架构系统
    - NUMA 下每个 CPU 有自己对应的 memory，可以快速访问，也可以访问其它 CPU 的 memory 不过速度较慢
    - NUMA-aware 的操作系统会在调度的时候尽可能的避免跨 CPU 的 memory 访问
- 实时操作系统调度
    - soft real-time systems 软实时：将关键的实时任务提高优先级，但并不保证如何调度
        - 例如 Linux 就是软实时操作系统
        - Linux 无法达到硬实时：因为硬实时一定要能精确控制非预期事件的产生（例如中断）
    - hard real-time systems 硬实时：规定任务必须在它的 ddl 之前完成执行，如果超时了，系统 watchdog 部件会察觉到系统有异常，会将系统 reboot 来恢复保证实时性
-

> 里面有几个问题

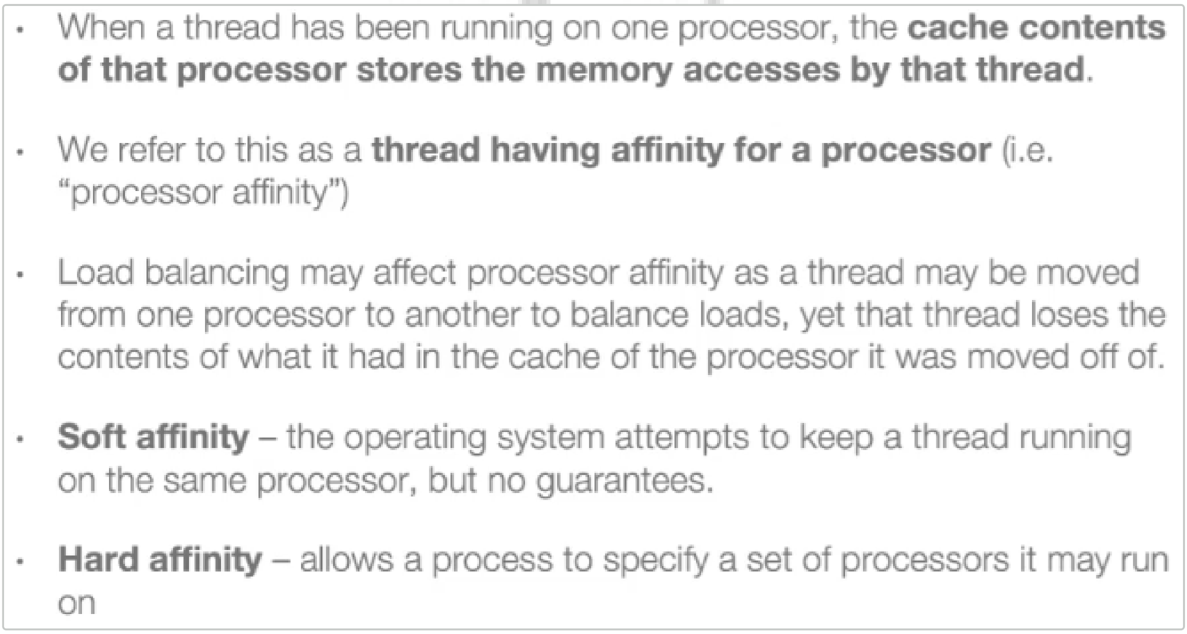
1. Load Balancing

   > 为了使所有CPU负载均衡，防止一个busy一个empty



- If SMP, need to keep all CPUs loaded for efficiency
  - Load balancing attempts to keep workload evenly distributed
  - **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  - **Pull migration** – idle processors pulls waiting task from busy processor

2. Processor Affinity



- When a thread has been running on one processor, the **cache contents of that processor stores the memory accesses by that thread**.
- We refer to this as a **thread having affinity for a processor** (i.e. "processor affinity")
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on

NUMA and CPU Scheduling

- If the operating system is NUMA-aware, it will assign
- memory closes to the CPU the thread is running on.

Real-Time CPU Scheduling

- Can present obvious challenges
  - **Soft real-time systems** - Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
  - **Hard real-time systems** - task must be serviced by its deadline

---

Synchronization

- 进程在执行的时候可能会被随时打断
  - 并发访问共享的资源可能会导致数据不一致性 (data inconsistency) 4
- 比如，进程中创建了两个线程，分别对共享的全局变量 counter 进行循环 + 1:

```c
static volatile int counter = 0;
void *mythread(void *arg) {
    printf("%s: begin\n", (char*)arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char*)arg);
    return NULL;
}
```

最后在两个线程结束的时候，counter 值会小于 2e7。原因:

- **counter = counter + 1** 在 C 语言里是一条语句
- 但是在汇编层面并不是

  原子操作

  ，有三条语句来完成这一操作:

  ```
  mov eax, <addr of counter>
  add eax, 1
  mov <addr of counter> eax
  ```

  - 分别进行 读、加一、写
- 如果一个线程在写之前被打断然后执行另一个线程了，会产生如下问题:

  ```
      Thread 1      OS      Thread 2
  ---------------- ---- ----------------
    read (eax=50)
      +1 (eax=51)
                    →
                              read (eax=50)
                                +1 (eax=51)
                             write (51)
                    ←
   write (51)
  ```

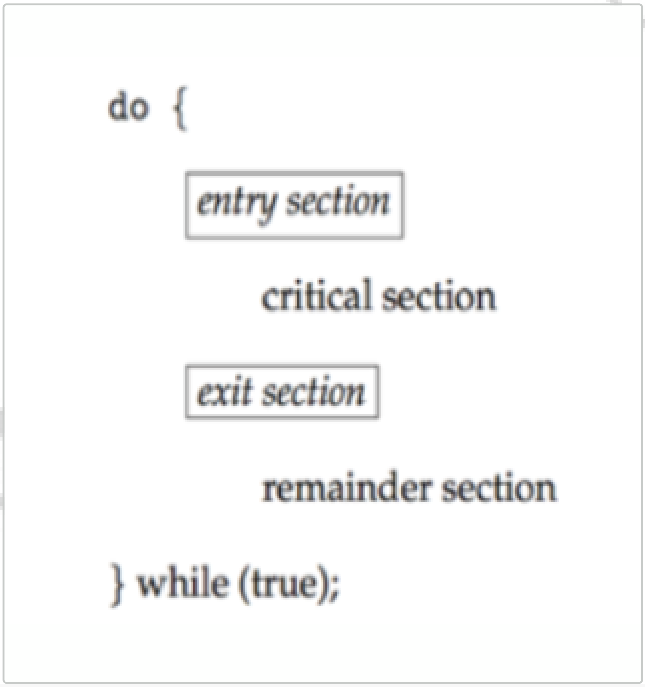
  - 两个线程都读到了 50，然后都加了一，最后都写回了 51
  - 虽然两个线程都被执行了一次，但最后 counter 的值只加了 1

Race Condition

- 几个进程并发地访问、修改同一个共享变量，其结果取决于访问的顺序，这种情况称为**条件竞争**（race condition）
- 如上就是一个条件竞争的例子
- 内核中也会发生条件竞争
  - 例如两个进程都在 fork 子进程，请求新的进程号
  - 进程号由内核全局变量 next_available_pid 维护
  - 内核并发处理两个 fork 请求时，可能会导致两个进程得到相同的进程号

Critical Section

- Consider system of n processes {p_0, p_1, ⋯, p_{n-1}}
- Each process has a critical section segment of code
  - e.g., to change common variables, update table, write file, etc.
- 同一时刻，只有一个进程可以处在 critical section
  - 在进入 critical section(以下简称为“CS”) 之前需要请求进入 CS 的权限，这部分代码称为 entry section
  - 退出 CS 后释放权限，这部分代码称为 exit section
  - 剩下的部分称为 remainder section
- General structure of process p_i is



```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

> Critical Section不是data，而是访问资源的一段代码不是一片区域

Solution to Critical-Section: Three Requirements

- **Mutual Exclusion** | 互斥，ME
  - 任意时刻只有一个进程在 CS 中
- **Progress**
  - 如果没有进程在 CS 且有进程需进入 CS，那么只有不在 remainder section 内执行的进程可参加选择，以确定谁能下一个进入 CS，且这种选择不能无限推迟 | if no process is executing in its critical section and some processes wish to enter their critical section, then only those processes that are not executing in their retainer sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely
- **Bounded waiting**
  - 从一个进程做出进入 CS 的请求，直到该请求允许为止，其他进程允许进入其 CS 的次数有上限 | There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - 防止了饥饿 | it prevents starvation

> 各自具体是什么

Peterson’s Solution

- Peterson 算法解决了两个进程的同步问题 | Peterson’s solution solves two-processes synchronization
- 其假设 load 和 store 是原子性 (atomic) 的
  - 原子性即不可分割，中途不能被打断
- 两个进程共享两个变量：
  - `int turn` ：标记当前正在 CS 中的进程
  - `bool flag[2]` ：标记进程是否想进入 CS
- Process 0:

```
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    // critical section
    flag[0] = false;
    // remainder section
} while (true);
```

- Process 1:

```
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    // critical section
    flag[1] = false;
    // remainder section
} while (true);
```

- 在现代体系结构乱序执行下不能保证正常工作

> 不行的原因是由于会重排

Hardware Instructions

- Special hardware instructions that allow us to either test-and-modify the content of a word, or two swap the contents of two words atomically (uninterruptibly.)
    - **Test-and-Set** instruction

    ```
    bool test_set(bool *target) {
        bool rv = *target;
        *target = true;
        return rv;
    }
    ```

        - 整体是一个原子操作
        - 用于实现 Peterson 算法

    ```
    do {
        while (test_set(&lock)); // busy wait
        /* critical section */
        lock = false;
        /* remainder section */
    } while (true);
    ```

        - 这种解法不满足 Bounded Waiting，无法保证 waiting 的一定被执行（取决于调度）
        - 满足 Bounded Waiting 的解法：

```
do {
    waiting[i] = true;
    while (waiting[i] && test_set(&lock));
    waiting[i] = false;
    /* critical section */
    j = (j + 1) & n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i) lock = false;
    else waiting[j] = false;
    /* remainder section */
} while (true);
```

- **Compare-and-Swap** instruction
  - 语义为:

    ```
    bool compare_and_swap(int *ptr, int expected, int new_value) {
        int rv = *ptr;
        if (rv == expected) {
            *ptr = new_value;
        }
        return rv;
    }
    ```

    - 整体是一个原子操作
    - returns the original calue of passed parameter value
    - setthe variable `value` of the passed parameter `new_value` but only if `*value == expected` is true. That is, the swap takes place only under this condition
  - 使用 compare-and-swap 指令的解法

    ```
    do {
        while (compare_and_swap(&lock, 0, 1) != 0);
        /* critical section */
        lock = 0;
        /* remainder section */
    } while (true);
    ```

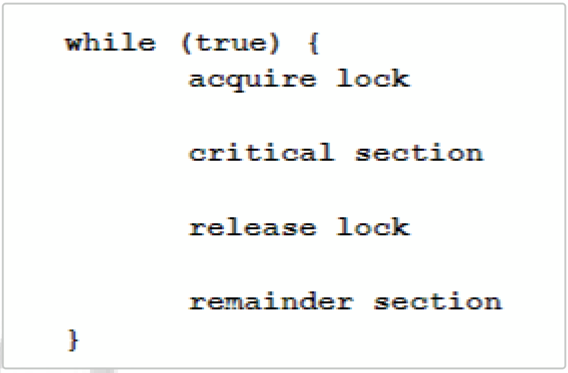
> 两个都有隐含要求，指令是atomical(一串要么都执行要么都不执行，整体执行不能被打断)

> - 原子变量
>   - 提供了对于整型、浮点型这种基础数据类型的原子性修改
>   - 例如 `increment(&sequence)` 来原子性增加 `sequence` 的值
>   - 使用 compare-and-swap 的实现:

```c
void increment(int *sequence) {
    int temp;
    do {
        temp = *sequence;
    } while (compare_and_swap(sequence, temp, temp + 1) ≠ temp);
}
```

Mutex Locks | 互斥锁

- Previous solutions are complicated and generally inaccessible to application programmers
- OS 设计者通过软件工具来解决 CS 问题，最简单的就是互斥锁（mutex lock）| OS designers build software tools to solve critical section problem. Simplest is mutex lock
- 通过先 acquire() 锁，然后执行 CS，再 release() 锁来保护 CS | Protect a critical section by first acquire() a lock then release() the lock
    - Boolean variable indicating if lock is available or not
- acquire() 和 release() 必须是原子的 | Calls to acquire() and release() must be atomic
    - 常通过硬件原子指令来实现 | Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires busy waiting
- This lock therefore called a spin lock(自旋锁)



```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

```c
void acquire() {
    while (!available); // busy wait
    available = false;
}
void release() {
    available = true;
}
```

- 通过忙等待（busy waiting）来实现
- 但是忙等待会影响效率（CPU 会一直执行等待的 while 循环）即很多自旋

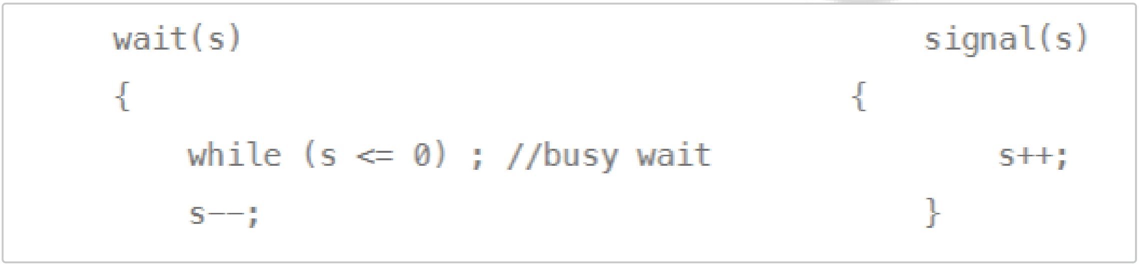
- 通过 yield 主动放弃 CPU 资源来减少无用自旋

```c
void lock() {
    while (test_set(&flag, 1) == 1)
        yield();
}
void unlock() {
    flag = 0;
}
```

  - 但是在多处理器上效率依旧不高

Semaphore | 信号量

- Semaphore S is an integer variable
  - e.g., to represent how many units of a particular resource is available
- 只可以通过 wait 和 signal 来原子性修改信号量的值 | It can only be updated with two atomic operations: wait and signal
  - spin lock can be used to guarantee atomicity of wait and signal
  - originally called P and V (Dutch)
  - wait 操作 (P) 会将信号量的值减一
  - signal 操作 (V) 会将信号量的值加一
  - a simple implementation with busy wait can be:



```
wait(s)                                   signal(s)
{                                        {
    while (s <= 0) ; //busy wait             s++;
    s--;                                  }
```

- **Binary semaphore** | 相当于 **锁** : 信号量的值只能为 0 或 1 | integer value can be only 0 or 1
  - also known as mutex lock to provide mutual exclusion(用于实现互斥锁)

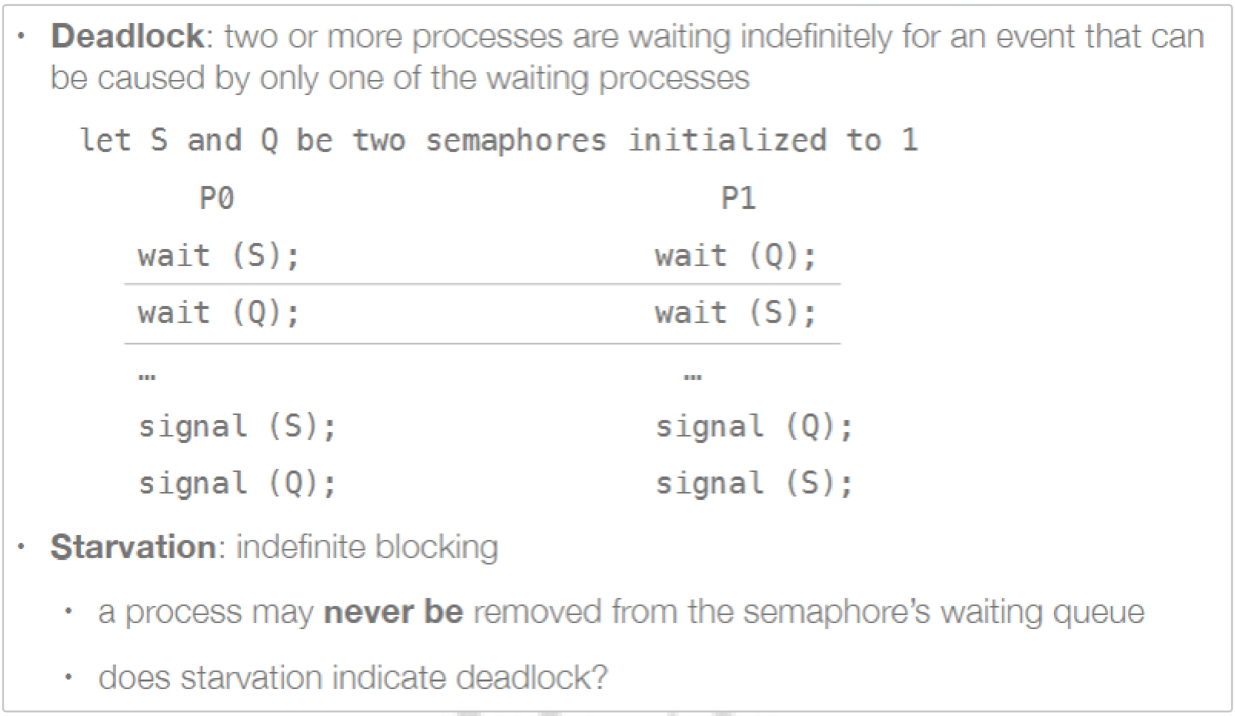
```c
Semaphore mutex = 1;
do {
    wait(mutex);
    // critical section
    signal(mutex);
    // remainder section
} while (true);

```

- **Counting semaphore:** 信号量的值可以为任意非负整数，来统计资源 | allowing arbitrary resource count
- Waiting Queue
    - 每个信号量关联一个 waiting queue
    - wait 没有立即返回的话就加入 waiting queue
    - signal 的时候唤醒一个在 waiting 的进程
    - 不需要忙等待

```c
void wait(Semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add process to S->list;
        block();
    }
}
void signal(Semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Deadlock and Starvation

- 死锁（Deadlock）是指两个或多个进程互相等待对方释放资源，导致所有进程都无法继续执行的情况
- 饥饿（Starvation）是指一个进程由于长时间无法获得资源而无法继续执行的情况



- **Deadlock**: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

```
let S and Q be two semaphores initialized to 1
       P0                           P1
    wait (S);                    wait (Q);
    wait (Q);                    wait (S);
    …                             …
    signal (S);                  signal (Q);
    signal (Q);                  signal (S);
```

- **Starvation**: indefinite blocking
  - a process may **never be** removed from the semaphore's waiting queue
  - does starvation indicate deadlock?

Priority Inversion

- 优先级反转（Priority Inversion）是指一个高优先级的进程由于等待一个低优先级的进程而导致自己的优先级降低，从而导致其他进程无法获得资源而无法继续执行的情况
- 例如：
  - 三个进程 A B C，优先级 A < B < C
  - 进程 A 持有锁，C 在等待这把锁
  - B 进程 ready 而且打断了 A
  - 效果上反转了 B 和 C 的优先级
- 解决方法：优先级继承（Priority Inheritance）
  - 临时的将持有锁的进程 A 的优先级赋值为正在等待的 C 的优先级
- Priority Inversion: a higher priority process is indirectly preempted by a lower priority task
  - e.g., three processes, P_L, P_M, and P_H with priority $P_L < P_M < P_H$
  - P_L holds a lock that was requested by P_H⇨P_H is blocked
  - P_M becomes ready and preempted the P_L
  - It effectively "inverts" the relative priorities of P_M and P_H
- Solution: priority inheritance
  - temporary assign the highest priority of waiting process (P_H) to the process holding the lock (P_L)