OS Chapter 1 Introduction

What is an Operating System

- An Operating System (OS) is a **program** that manages the computer **hardware**
  - 操作系统(OS)是管理计算机**硬件**的**程序**
- An Operating System (OS) provides a **basis** for Application Programs
  - 操作系统(OS)为应用程序提供了**基础**
- A Operating System (OS) acts as an **intermediary** between computer **user** and computer **hardware**
  - 操作系统(OS)作为**用户**和计算机**硬件**之间的**中介**

What Operating Systems Do — User view

- One PC user want **convenience**, **ease of use** and **good performance**, and don't care about **resource utilization**
  - 需要**方便**，**易用**以及**高性能**，但不关心**资源利用率**
- Shared computer such as mainframe or minicomputer must **keep all users happy**
  - 大型机或小型机等共享计算机必须**让所有**用戶**都满意**
- Handheld computers are **resource poor**, **optimized for usability** and **battery life**
  - 掌上型电脑资源(算力、存储容量)匮乏，需要针对可用性和电池寿命进行优化
- Some computers have little or no user interface, such as embedded computers in devices and automobiles
  - 有些计算机几乎没有用戶界面，例如设备和汽车中的嵌入式计算机

What Operating Systems Do — System view

- OS is a **resource allocator资源分配器**
  - Manages all resources(CPU time, memory space, file-storage space, I/O devices, and so on)
    - 管理所有的资源(CPU时间，内存空间，文件存储空间，I/O设备等等)
  - Decides between conflicting requests for efficient and fair resource use
    - 在冲突请求之间做出决定，保证整个计算机系统的高效和公平的资源利用
- OS is a **control program控制程序**
  - Controls execution of programs to prevent errors and improper use of the computer, especially for I/O
    - 执行控制程序以防止错误和不当使用计算机，尤其是I/O的不正当使用

Where is the OS stored on the computer & What operation or program starts the OS

- **bootstrap program引导程序** is loaded at power-up or reboot

  - Loads operating system kernel and starts execution
    - Locate the operating-system kernel and load it into memory
  - Some services are provided outside of the kernel, by system software
  - Typically stored in ROM, generally known as **firmware**
    - 一般来说引导程序存储在ROM中，而整个操作系统存储在硬盘中，引导程序引导操作系统从硬盘加载到主存储器(内存)中

Storage Structure

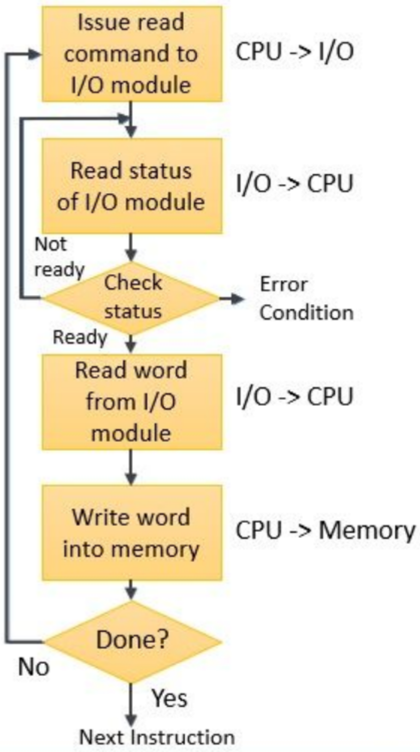
- Main memory – **only** large storage media that the CPU **can access directly**
  - Also called Random access memory(RAM)
  - Rewritable
- Computers use other forms of memory as well
  - Read-only memory(ROM)
- Interaction is achieved through a sequence of **load** or **store** instructions to specific memory addresses
  - 系统交互是通过对特定内存地址的一系列**加载**或**存储**指令来实现的
- Ideally, we want the programs and data to reside in main memory permanently. Usually, it is **impossible**
  - Main memory is too **small**
  - Main memory is **volatile易失性的**
- Secondary storage – extension of main memory that provides large **nonvolatile** storage capacity
  - Hard disks(HDD) – the most common secondary-storage device
  - Solid-state disks(SSD) – faster than hard disks, nonvolatile
    - Stores data in a large DRAM, a hidden hard disk, a battery
    - Becoming more popular: flash memory
  - Caching(Cache) – copying information into faster storage system; main memory can be viewed as a cache for secondary storage

I/O

Input & Output

Programmed I/O(程序控制下的I/O)

A technique that we use to transfer data between the processor and the I/O module.



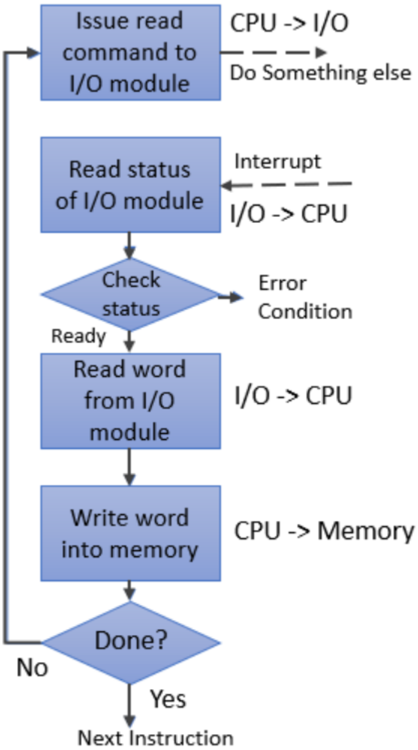
Programmed I/O to transfer data from I/O module to memory

特点就是利用编程，实现`while()`式的等待，处理器不断检查I/O模块是否准备好接收和发送数据，或者I/O模块是否完成了期望的任务。 处理器的这种长时间等待会降低系统的性能

编程式I/O的主要特点是简单、**易于实现**，但是CPU需要不断轮询I/O设备的状态，会**浪费CPU资源**，并且响应速度较慢

Interrupted I/O(中断型I/O)

An approach to transfer data between ‘memory’ and ‘I/O devices’ through the ‘processor’.



Interrupted I/O to Transfer Data from I/O Module to Memory

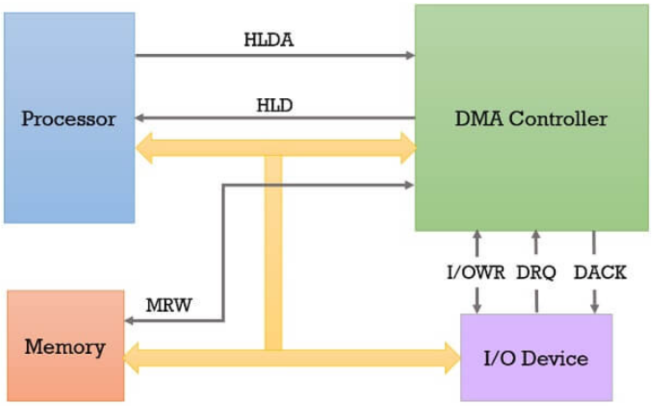
在中断式I/O中，处理器向相应的I/O模块发出`READ`I/O命令，并继续执行其他一些有用的任务。它**不会等待I/O**模块准备好所需的数据

当I/O设备完成操作时，它会**向CPU发送中断信号**，当当处理器发现I/O模块的中断时，它会暂停当前的执行并保存上下文（例如，程序计数器、处理器寄存器），并跳转到中断处理程序来处理I/O设备的数据

中断式I/O的主要特点是能够**减少CPU的轮询操作**，提高了CPU的利用率和响应速度，但是中断处理程序会占用CPU时间，并且需要进行中断处理程序的设计和实现

Direct Memory Access(DMA/直接内存访问)

Transfers the block of data between the memory and I/O devices of the system, **without the participation of the processor**.



DMA Controller Data Transfer

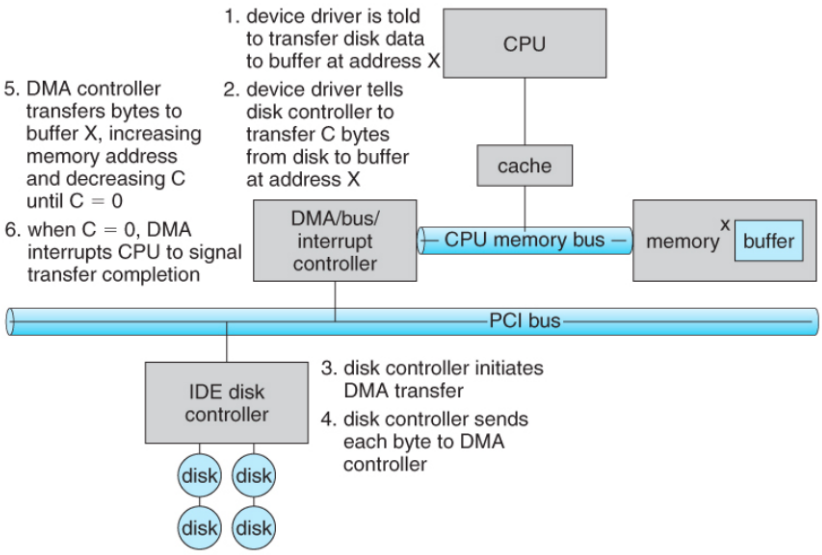


Figure 13.5 - Steps in a DMA transfer.

设备控制器(Device Controller)将数据块从缓冲存储器**直接传输到主存储器**，无需CPU干预。对于数据，**每个块只产生一个中断**，而不是每个字节产生一个中断，因此DMA是用于能够以接近内存速度传输信息的高速I/O设备

DMA controller modes

1. **Burst Mode**(突发模式)
   - The DMA controller gains the charge of the system bus, then it releases the system bus only after completion of data transfer. Till then the CPU has to wait for the system buses.
   - DMA控制器在**突发模式**下直接**占据总线**，它优先访问总线，而CPU则会被暂停访问总线，直到DMA传输操作完成或者达到传输长度的限制，才会释放总线给CPU进行访问。这种方式可以避免CPU和DMA控制器之间的竞争，提高系统的性能和效率(单次全部数据传输)
2. **Cycle Stealing Mode**(周期传输/周期窃取模式)
   - The DMA controller forces the CPU to stop its operation and relinquish the control over the bus for a short term to DMA controller. After the transfer of every byte, the DMA controller releases the bus and then again requests for the system bus. In this way, the DMA controller steals the clock cycle for transferring every byte.
   - DMA控制器在**周期传输模式**下强制CPU停止运行，并在**短期内将对总线的控制权交给DMA控制器**。传输完每个字节后，DMA 控制器释放总线，然后再次请求系统总线(多次传输数据)
3. **Transparent Mode**(透明模式)
   - The DMA controller takes the charge of system bus only if the processor does not require the system bus.

- DMA控制器通过总线接口，直接读写内存和外设寄存器，进行数据传输和处理，在完成数据传输和处理后，通过DMA中断或者其他方式向CPU发出通知。需要注意的是，在**透明模式**下，DMA控制器**具有和CPU相同的访问权限**，因此需要谨慎处理访问冲突和竞争。同时，由于Transparent Mode需要直接访问系统内存和外设寄存器，因此需要进行适当的安全性和稳定性考虑，以避免对系统造成损害和影响。

Comparison

- **Programmed I/O**: It transfers data at a high rate, but it can't get involved in any other activity during data transfer.
  - 程序控制下的I/O可以高速地传输数据，易于编程实现，但在数据传输过程中CPU不能参与其他任何活动
- **Interrupted I/O**: the processor doesn't keep scanning for I/O devices ready for data transfer. But, it is fully involved in the data transfer process.
  - 中断型I/O也可以高速地传输数据，它利用操作系统提供的中断机制，通过中断通知CPU当前的传输状态，虽然CPU不会持续扫描I/O设备，但是CPU还是参与了整个数据传输过程(CPU干预了数据从I/O与内存之间的传输)
- **DMA**: completes this task at a faster rate and is also effective for transfer of large data block.
  - DMA虽然也利用了中断机制，但它只需要在传输的开始与结束进行中断通知，同时CPU不需要介入数据传输的干预，数据可以通过I/O设备直达内存

Operating System Structure

- **Multiprogramming** is needed for efficiency(保证多程序运行)
  - Single user cannot keep CPU and I/O devices busy at all times(保证多用户运行时的公平性)
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute(保证CPU总是有活可干)
- **Basic idea** of multiprogramming
  - A subset of total jobs in system is kept in memory(一部分工作将会被保存在内存中)
  - One job selected and run via job scheduling(存在工作调度)
  - When it has to wait (for I/O for example), OS switches to another job(需要等待时，操作系统将会到切换其他的工作)
- **Timesharing**(**multitasking**) is logical extension in which CPU switches jobs **so frequently** that users can interact with each job while it is running, creating interactive computing(分时/多任务的概念——CPU非常频繁地在任务间切换导致用户可以与任一工作交互)
  - Response time should be < 1 second(程序的回复/反应时间总是低于1秒)
  - Each user has at least one program executing in memory(process进程的概念)
- Timesharing and multiprogramming require that **several jobs be kept simultaneously** in memory(多项工作同时保存在内存中)
  - If several jobs ready to be brought into memory $\rightarrow$ **Job scheduling(工作调度)**
  - If several jobs ready to run at the same time $\rightarrow$ **CPU scheduling(处理器调度)**
- In timesharing, the OS must **ensure reasonable response time**
  - If processes don't fit in memory, **swapping** moves them in and out to run(内存与硬盘之间的swap策略)

- **Virtual memory** allows execution of processes not completely in memory(虚拟内存允许进程的执行有部分不在内存中——地址空间可以扩展到硬盘)

> **Prompt: 能向我解释一下分时操作系统和实时操作系统的特点和区别吗**
>
> **Reply be ChatGPT:**
>
> 当涉及到操作系统时，分时操作系统和实时操作系统是两个非常不同的概念。下面是它们的特点和区别：
>
> **分时操作系统（Time-sharing Operating System）特点**：
>
> - 分时操作系统是一种多任务操作系统，允许多个用户同时访问计算机资源，并在同一时间内运行多个程序。
> - 它为每个用户分配一定的时间片，以轮流执行用户的请求，从而在感觉上实现同时操作的效果。
> - 分时操作系统能够有效地利用计算机的资源，提高计算机的利用率和效率。
>
> **实时操作系统（Real-time Operating System）特点**：
>
> - 实时操作系统是一种专门为实时应用设计的操作系统，它需要在特定的时间范围内完成任务，并保证任务能够按照规定的时间完成。
>   实时操作系统通常分为硬实时系统和软实时系统。
> - 硬实时系统要求任务必须在规定的时间内完成，否则会导致系统故障或者损失，这种系统通常用于控制和自动化领域。
> - 软实时系统要求任务在规定的时间内完成，但可以容忍一定的延迟，这种系统通常用于多媒体和通信领域。
>
> **区别**：
>
> - 分时操作系统是为了处理多个用户请求而设计的，它的主要目标是为了提供良好的用户体验。
> - 分时操作系统的响应时间通常不需要非常短，因为它更注重于用户的交互性和易用性，而不是实时性能。
> - 实时操作系统的主要目标是保证任务能够按照规定的时间完成，因此它的响应时间非常短，通常只有几毫秒的时间。
> - 实时操作系统需要特定的调度算法和内存管理技术，以确保任务能够按时完成，并且必须保证系统的可靠性和稳定性。
> - 分时操作系统和实时操作系统在设计和实现上有很大的差异，因此它们通常用于不同的应用场景。

Interrupt

Interrupt driven(中断的驱动形式)

- **Hardware** interrupt by one of the devices
  - 硬件中断——发送电信号给CPU(例如：I/O传输完毕)
- **Software** interrupt (**exception** or **trap**):
  - Software error (e.g., division by zero)
    - 软件运行错误
  - Invalid memory access
    - 非法内存访问

- A specific request from user program that an OS service be performed
    - 程序的特定请求，请求执行操作系统服务

Exceptions(异常)

- Any event that can alter normal CPU instruction execution flow
  - **异常**的定义是：任何可以改变CPU正常指令执行流的事件

Interrupt Service Routine(中断服务程序)

- Interrupt transfers control to the **interrupt service routine** generally, through the **interrupt vector**, which contains the addresses of all the service routines
  - 中断将控制转移到**中断服务程序**，通过**中断向量**，中断向量中包含所有中断服务程序的地址
- Every interrupt type is assigned a number:
  - **Interrupt vector中断向量**
- A table of pointers to interrupt routine
- When an interrupt occurs, the vector determines what code is invoked to handle the interrupt, the **interrupt vectors table** is as follows
- The interrupt must transfer control to the appropriate interrupt service routine.Thus, the interrupt routine is called indirectly through the table
  - **中断处理的流程：**
    外部设备/内部软件发出中断 $\rightarrow$ 查找中断向量表对应的中断类型 $\rightarrow$ CPU挂起当前进程 $\rightarrow$ 中断服务程序(ISR)处理中断 $\rightarrow$ 控制流归还给进程

System Call

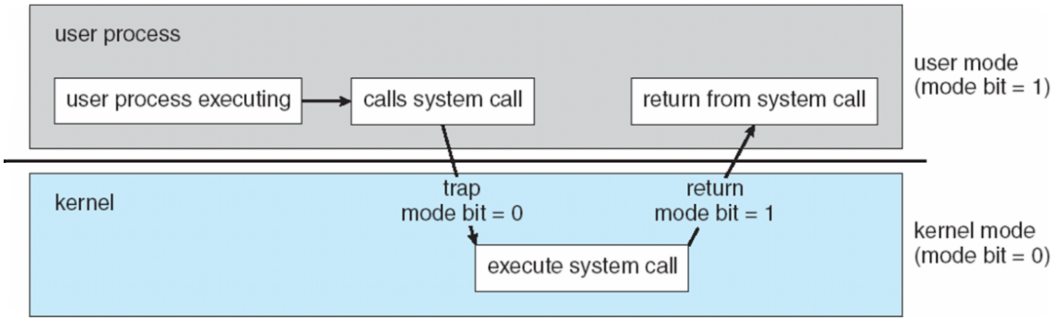
- System call provides the services of the operating system to the user programs via **Application Program Interface (API)**.
  - 系统调用通过**应用程序接口**向用户程序提供操作系统的服务
- It provides an **interface** between a process and operating system to allow user-level processes to request services of the operating system.
  - 它提供进程和操作系统之间的**接口**，允许用户级进程请求操作系统的服务
- System calls are the **only** entry points into the kernel system.
  - 系统调用是进入内核系统的**唯一**入口点

Operating-System Operations

Operating System Mode and Effect

- **Proper execution**: **distinguish** between the execution of operating-system code and user-defined code区分操作系统代码和用户定义的代码的执行
  - **Dual-mode** operation allows OS to protect itself and other system components双模式的操作系统可以保护自身的代码
    - User mode and kernel mode它区分了用户态和系统内核态
    - Mode bit provided by hardware

        - Provides ability to distinguish when system is running user code or kernel code
    - System call changes mode to kernel, return from call resets it to user使用系统调用将会将系统从用户态切换至内核态，操作系统接管并执行操作后返回到用户态
- **Protection** operating system保护操作系统
    - Some instructions designated as privileged, only executable in kernel mode高权限的指令只能在内核态下被执行



User Mode

- Because an application's virtual address space is **private**, one application cannot alter data that belongs to another application.
    - 用户应用程序的虚拟地址空间是**私有**的，一个应用程序不能更改属于另一个应用程序的数据
- Each application runs in **isolation**, and if an application crashes, the crash is limited to that one application.
    - 每个应用程序都**独立运行**，如果一个应用程序崩溃，则崩溃仅限于该应用程序
- Other applications and the operating system are **not affected by the crash**.
    - 其他应用程序和操作系统**不受崩溃的影响**
- In addition to being private, the virtual address space of a user-mode application is **limited**.
    - 用户应用程序除了虚拟地址空间私有之外，用户模式应用程序的**虚拟地址空间也是有限的**
- A processor running in user mode **cannot access** virtual addresses that are reserved for the operating system.
    - 以用户模式运行的处理器**无法访问为操作系统保留的虚拟地址**
- Limiting the virtual address space of a user-mode application **prevents** the application from altering, and possibly damaging, critical operating system data.
    - 限制用户模式应用程序的虚拟地址空间可**防止**应用程序更改并可能损坏关键操作系统数据

Kernel Mode

- All code that runs in kernel mode **shares** a single virtual address space.
    - 在内核模式下运行的所有代码**共享一个虚拟地址空间**
- This means that a kernel-mode driver is **not isolated** from other drivers and the operating system itself.
    - 这意味着内核模式驱动程序并**不**与其他驱动程序和操作系统本身**隔离**
- If a kernel-mode driver accidentally writes to the **wrong virtual address**, data that belongs to the operating system or another driver could be compromised.

    - 如果内核模式驱动程序不小心写入了**错误的虚拟地址**，则属于操作系统或其他驱动程序的数据可能会**受到损害**
- If a kernel-mode driver crashes, the **entire operating system crashes**.
    - 如果内核模式下的驱动程序崩溃，则**整个操作系统都会崩溃**

Management

Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- **Scheduling** processes and threads on the CPUs
    - 在CPU上**调度**进程和线程
- **Creating** and **deleting** both user and system processes
    - **创建**和**删除**用户和系统进程
- **Suspending** and **resuming** processes
    - **挂起**和**恢复**进程
- Providing mechanisms for **process synchronization**
    - 提供**进程同步**机制
- Providing mechanisms for **process communication**
    - 提供**进程通信**机制
- Providing mechanisms for **deadlock handling**
    - 提供**死锁处理**机制

Memory Management Activities

- To execute a program all (or part) of the **instructions must be in memory**.
    - 要执行程序，所有（或部分）**指令必须在内存中**
- All (or part) of the **data** that is needed by the program **must be in memory**.
    - 程序所需的全部（或部分）**数据必须在内存中**
- Memory management determines **what is in memory** and when optimizing **CPU utilization** and computer **response** to users
    - 内存管理确定**内存中的内容**，**何时优化CPU利用率**，以及计算机**对用户的响应时间**
- Keeping **track** of which parts of memory are currently being used and by whom
    - **追踪**内存当前正在使用哪些部分以及由谁使用
- **Deciding** which processes (or parts thereof) and data to move into and out of memory
    - **决定**将哪些进程（或其部分）和数据移入和移出内存
- **Allocating** and deallocating memory space as needed
    - 根据需要**分配**和释放内存空间

Storage Management

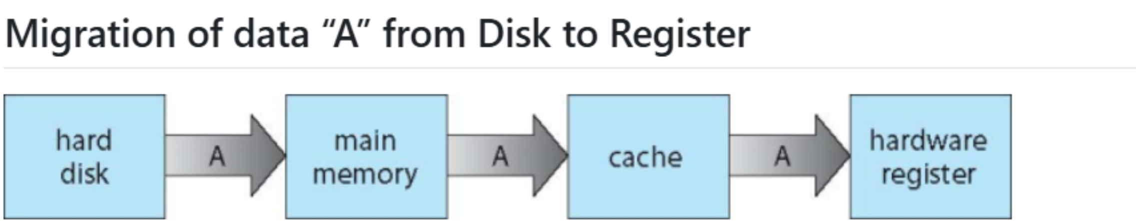
OS provides uniform, logical view of information storage(统一的、逻辑的信息存储视图)

- Abstracts physical properties to logical storage unit - **file**
  - 将物理属性抽象到逻辑存储单元——**文件**
- **File**
  - a collection of related information defined by its creator
    - 由其创建者定义的相关信息的集合
  - files represent programs and data
    - 文件代表程序和数据

Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time, and entire speed of computer operation hinges on disk subsystem and its algorithms.Therefore, proper management is of central importance
  - 磁盘通常用于存储无法放入主内存的数据或必须“长时间”保存的数据，而计算机运行的整体速度取决于磁盘子系统及其算法。因此，适当的管理至关重要
- OS activities
  - Free-space management自由空间管理
  - Storage allocation存储分配
  - Disk scheduling磁盘调度

Migration of data “A” from Disk to Register



- Multitasking environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy
  - 多任务环境必须小心使用最近的值，无论它存储在存储层次结构中的哪个位置
- Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache
  - 多处理器环境必须在硬件中提供高速缓存一致性，以便所有CPU在其高速缓存中具有最新值

1. For example, suppose that an integer A that is to be incremented by 1 is located in file B, B resides on disk;
   假设一个要加1的整数A位于文件B中，B存储在磁盘上
2. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory;
   在整个增量操作中，首先发出I/O操作，将A所在的磁盘块复制到主存；

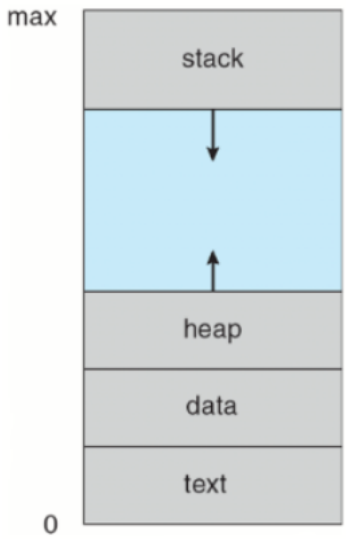
3. This operation is followed by copying A to the cache and to an internal register
   此操作之后是将A复制到缓存和CPU的内部寄存器
4. Thus, the copy of A appears in several places: on the disk, in main memory, in the cache, and in a integer register
   因此，A的副本出现在几个地方：磁盘上、主存中、高速缓存中和整数寄存器中
5. On the increment takes place in the internal register, the value of A differs in the various storage systems; The values of A becomes the same only after the new value of A is written from the internal register back to the disk
   由于发生在内部寄存器的增量，A的值在不同的存储系统中不一定是相同的；只有将新的A值从内部寄存器写回磁盘后，A的值才变得相同
6. This situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache.
   这种情况在多处理器环境中变得更加复杂，因为在多处理器环境中，除了维护内部寄存器外，每个CPU还包含一个本地缓存
7. In such an environment, a copy of A exist simultaneously in several caches.
   在这样的环境中，A的副本同时存在于多个缓存中
8. Since the various CPUs can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**
   由于各种 CPU 都可以并行执行，我们必须确保对一个缓存中 A 的值的更新立即反映到 A 所在的所有其他缓存中。 这种情况称为**缓存一致性**
9. In a distributed environment, several copies of the same file can be kept on different computers
   在分布式环境中，同一个文件的多个副本可以保存在不同的计算机上

OS Chapter 2 Process

Process Concept

- A process is the instance of computer program that is being executed.
  - 进程是计算机程序运行的实例
- Process is basically a program in execution.
  - 进程就是一个在运行中的程序

Multiple parts



- The program **code**, also called **text section文本段**
- **Data section数据段** containing **global variables**, **static variables**
- **Heap堆区** containing memory **dynamically allocated** during run time
- **Stack栈** containing temporary data like **function parameters**, **return addresses**, **local variables**

Supplement

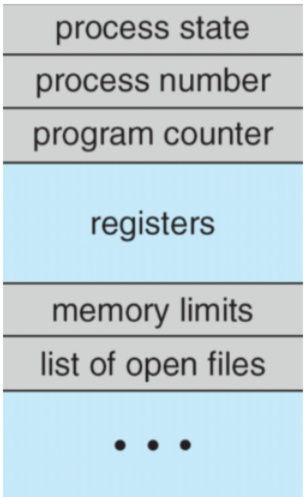
- **Program** is **passive** entity stored on disk (executable file), **process** is **active**, and program becomes process when executable file loaded into memory.
    - 程序是存储在磁盘上的被动的实体(以可执行文件的形式)，而进程是主动的。当可执行文件被加载进内存时，程序就成为了进程
- **One program** can be **serveral processes**, when multiple users executing the same program, these processes associated with the same program are **separate execution sequences**, although the **text sections are equivalent**, the data, heap, and stack sections vary
    - 一个程序可以成为多个进程，当多个用户执行相同的程序时，这些与同一个程序相关联的进程都是单独的执行序列，尽管它们共享了相同的文本段(代码)，但是它们的数据段、堆区、栈区的内容是各不相同的

Process Control Block(进程控制块/PCB)

- Each process is represented in the operating system by a **process control block**
    - 每个进程在操作系统中由**进程控制块**表示

Information associated with each process(与每个进程相关的信息)

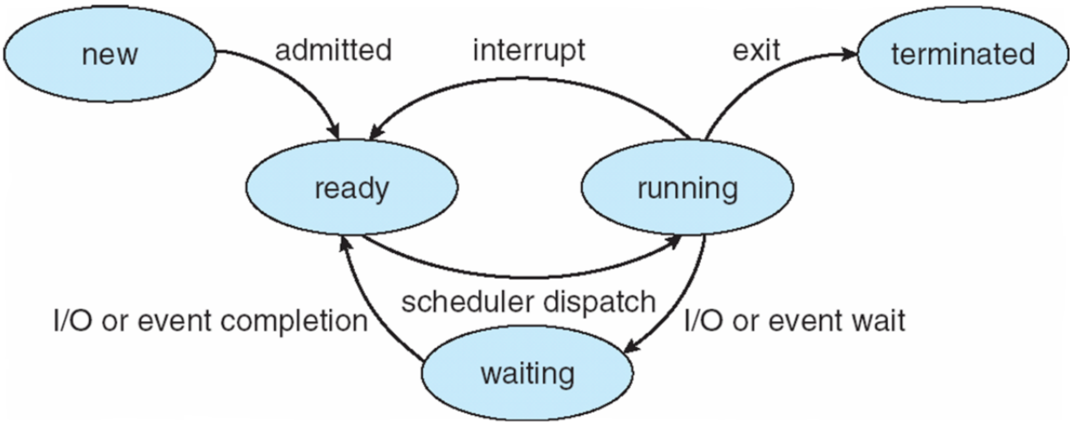
*also called **task control block***



- **Program counter程序计数器** – address of next instruction to be executed
- **Process state进程状态** – running, waiting, etc
- **CPU registers CPU寄存器** – contents of all process-centric registers所有以进程为中心的寄存器的内容
  - **Process Register进程寄存器**
    - SP(stack pointer栈指针)
    - BP(base pointer基指针)
    - IP(instruction pointer指令指针)
    - When a processes is running and it's time slice expires, the current value of process specific registers would be stored in the PCB and the process would be swapped out(PCB会在进程运行时间片用完，被切换出去时，将进程特有的寄存器值保存起来)
    - When the process is scheduled to be run, the register values is read from the PCB and written to the CPU registers. This is the main purpose of the registers in the PCB(当进程被调度执行时，PCB中保存的该进程的寄存器值将会被加载进CPU寄存器中，这是PCB寄存器的主要用途)
- **CPU scheduling information CPU调度信息** - priorities, scheduling queue pointers
  - **Process Priority进程优先级**, eg: OS process priority > User process priority
- **Memory-management information内存管理信息** – memory allocated to the process
- **Accounting information统计信息** – CPU used, clock time elapsed since start, time limits
- **I/O status information I/O状态信息** – I/O devices allocated to process
  - **I/O info I/O信息**
  - **List of Open Files打开文件的列表**

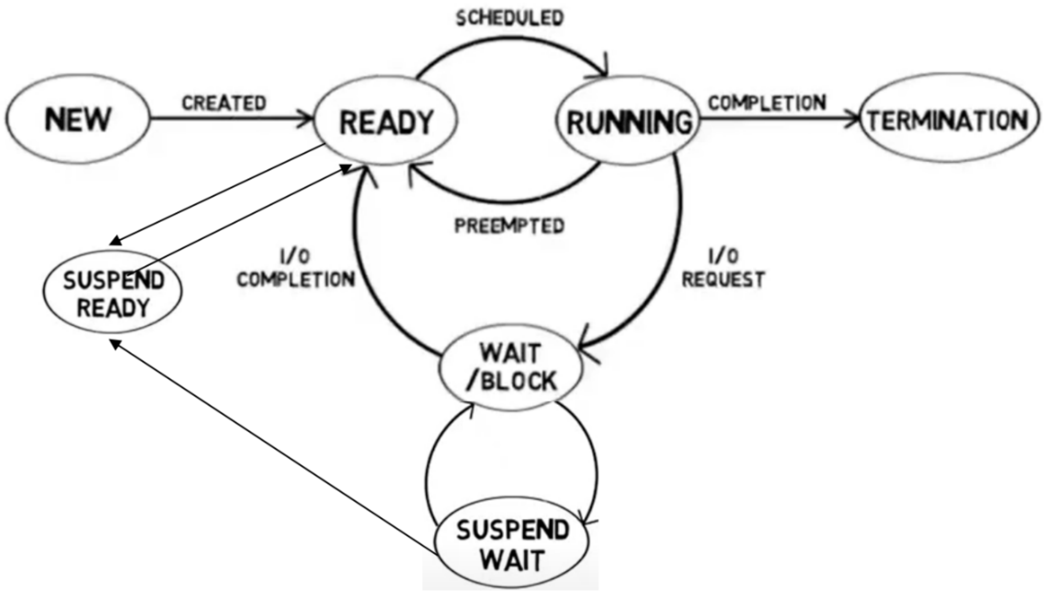
Process state

- The state of a process is defined in part by the current activity of that process.
  - 进程的状态部分由该进程的当前活动定义



States

- **new**: The process is being created
- **ready**: The process is waiting to be assigned to a processor
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
    - For waiting, there is another graph to show the status:



- **terminated**: The process has finished execution

Process Scheduling

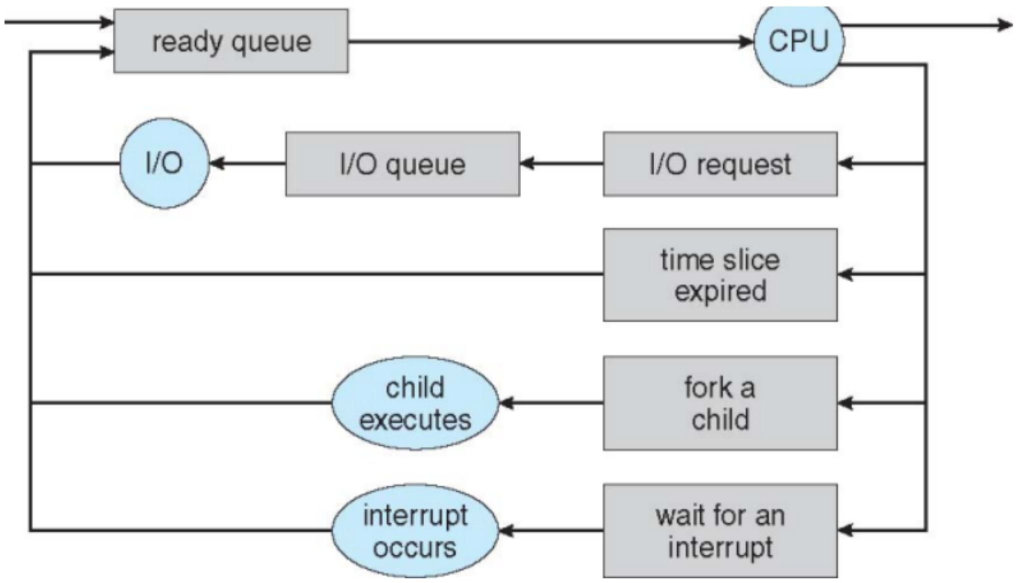
- The objective of multiprogramming is to have some processes running at all times, to **maximize CPU utilization**.
    - 多道程序设计的目标是让某些进程一直运行，以**最大限度地提高CPU利用率**
- The objective of **time sharing** is to switch the CPU among processes **so frequently** that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process for program execution on the CPU
    - **分时**的目的是**频繁地在进程之间切换**CPU，以便用户可以在每个程序运行时与其进行交互。 为了满足这些目标，进程调度程序选择一个可用进程在CPU上执行程序

Time sharing system

- Time sharing, or multitasking, is a **logical extension逻辑延展** of multiprogramming.
- Multiple jobs are executed by **switching the CPU** between them.
- In this, the CPU time is **shared by different processes**, so it is called as "Time sharing Systems"
- Time slice is **defined by the OS**, for sharing CPU time between processes.
- Examples: **Multics**, **Unix**, **Linux**, **Windows**, etc.

Implement

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
    - **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 or I/O queue** – set of processes waiting for an I/O device
    - Processes migrate among the various queues**进程将会在各个队列中不断切换**



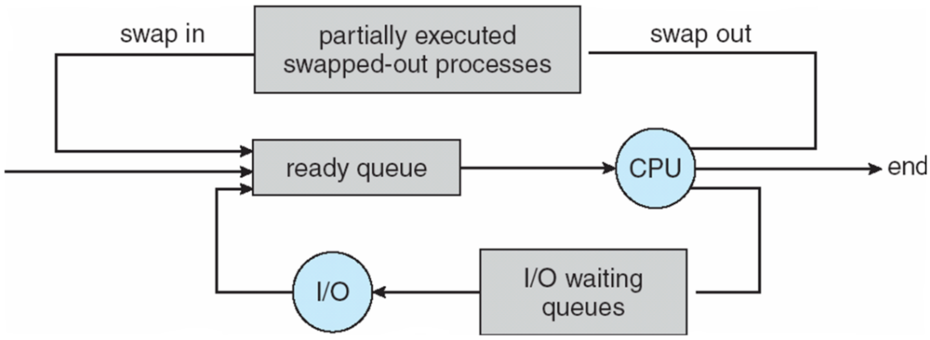
Schedulers(调度器)

Long-term scheduler

- **Long-term scheduler**(or **job scheduler**) – selects which processes should be brought into the ready queue
    - Long-term scheduler is invoked **infrequently**(seconds, minutes)
    - The long-term scheduler controls the degree of multiprogramming(the number of processes in memory)
    - May be slow

Mid-term scheduler

- **Medium-term scheduler** can be added **if degree of multiple programming needs to decrease**
  - Remove process from memory, store on disk, bring back from disk to continue execution: **swapping**



Short-term scheduler

- **Short-term scheduler**(or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Short-term scheduler is invoked **frequently**(milliseconds)
  - It must be fast, because of the short time between executions

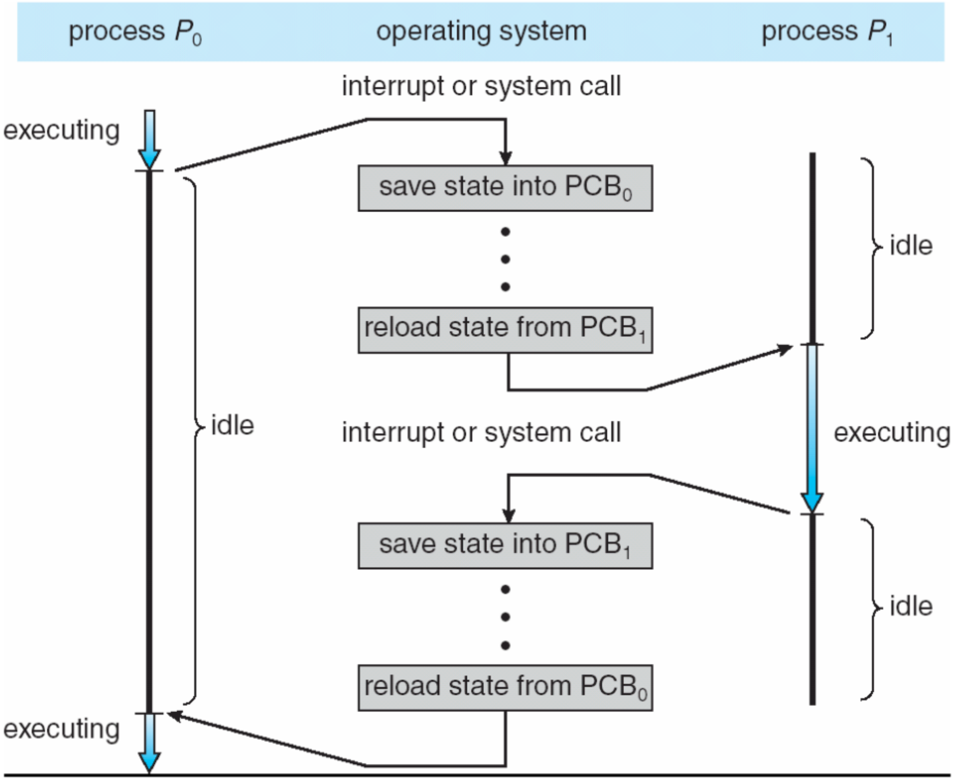
Process type

Processes can be described as either:

- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - If **all processes are I/O bound**, the ready queue will almost always be empty, and the **short-term scheduler** will have little to do
- **CPU-bound process** – spends more time doing computations; very long CPU bursts
  - If **all processes are CPU bound**, the I/O waiting queue will almost always be empty, **devices will go unused**, and again the **system will be unbalanced**

Context switch

- **Interrupts** cause the OS to change a CPU from its current task and to run a kernel routine. Such operation happen frequently on general-purpose systems
  - **中断**导致操作系统将CPU从其当前任务更改为运行内核程序。 这种操作在通用系统上经常发生
- 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 scheduled to run via a **context switch**
  - 当发生上下文切换时，CPU切换到另一个进程，需要保存正在运行的进程的状态，
- Context of a process represented in the PCB
- Context-switch time is **overhead**; the system does no useful work while switching
  - The **more complex** the OS and the PCB $\rightarrow$ the **longer** the context switch



Operations on processes

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination
  - 大多数系统中的进程可以并发执行，并且可以动态地创建和删除。 因此，这些系统必须提供进程创建和终止的机制

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
  - 父进程创建子进程，子进程又创建其他进程，形成进程树
- Generally, process identified and managed via a **process identifier**(**pid**)
  - A unique value for each process每个进程独一无二的标识符
  - An index to access various attributes of a process作为访问一个进程属性的唯一下标
- `fork()` function
  - Returns a value of type `pid_t` (essentially, an integer)
  - Does not take any input parameters, what is indicated by the formal parameter void.
  - The return code for fork is `0` **for the child process** and the **process identifier** of child is returned to the parent process.
  - On success, both processes continue execution at the instruction after the fork call, parent and children execute concurrently
    - 得到的返回值为`0`，则当前进程为子进程，若为`pid_t`，则当前进程为父进程，若为`-1`，则代表`fork()`失败

        - We don't know whether the OS will first give control to the parent process or the child process(在老版本的Linux系统中，子进程是优先的，但后来发现如果不断地fork，父进程就永远不执行，因此移除了这个设定)
    - On failure, `-1` is returned to the parent process.
- **Resource Sharing Options父子进程资源共享选项**
    - Child processes can obtain resources directly from the operating system
        - 子进程可以直接从操作系统获取资源
    - Child processes may be limited to a subset of parent process resources
        - 子进程可用的资源被限制为父进程资源的子集
        - Prevent any process that creates too many child processes from overloading the system
            - 这是为了防止进程创建太多的子进程而导致系统过载
- There are also **two address space possibilities(地址空间)** for new processes:
    - The child process is a copy of the parent process (it has the same programs and data as the parent process)
        - 子进程是父进程的副本
    - The child process loaded a program(Use `execve()`)
        - 子进程加载了新的程序

Process Termination

exit() system call

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
    - 使用系统调用`exit()`删除/退出进程
    - Returns status data from child to parent (via `wait()`)
        - 在子进程中，执行`exit()`系统调用后会返回其状态信息到父进程，若父进程正在`wait()`，则它将会得到这条信息并被唤醒
    - Process'resources are deallocated by operating system
        - 结束的进程资源将会被操作系统释放
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
    - 父进程可用使用`abort()`系统调用终止子进程的执行，其场景有：
    - Child has exceeded allocated resources
        - 子进程使用的资源超过了已分配的资源
    - Task assigned to child is no longer required
        - 不再需要分配任务给子进程
    - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
        - 父进程正在退出，此时操作系统不允许子进程继续执行

wait() system call

- When a process creates a child process, sometimes it becomes necessary that the parent process should execute only after the child has finished.
    - 当进程创建子进程时，有时父进程只在子进程完成后才执行是必要的
- `wait()` system call does exactly this. It makes the parent process wait for child process to finish and then the parent continues its working from the statement after the `wait()`.
    - `wait()`系统调用使父进程等待子进程完成，然后父进程从`wait()`之后的语句继续其工作
- On success, `wait()` returns the PID of the terminated child process while on failure it returns `-1`.
    - 成功时，`wait()`返回已终止子进程的 PID，失败时返回`-1`
- If a process terminates, then all its children must also be terminated.
    - 如果一个进程终止了，那么它的子进程也必须终止
    - **cascading termination级联终止**. All children, grandchildren, etc. are terminated.
    - The termination is initiated by the operating system.
        - 操作系统进行对这些子进程的终止
- The parent process may wait for termination of child process by using the wait() system call. The call returns status information and the PID of the terminated process `pid = wait(&status);`
    - 父进程可以使用`wait()`系统调用来等待子进程的终止，该系统调用返回状态信息以及终止的子进程的PID
- If no parent waiting (did not invoke wait () yet), terminated process is a **zombie**
    - 当子进程`exit()`退出之后，他的父进程没有通过`wait()`系统调用回收他的进程描述符的信息，该进程会继续停留在系统的进程表中，占用内核资源，这样的进程就是**僵尸进程**
- If parent terminated without invoking wait(), process is an **orphan**
    - 当一个进程正在运行时，他的父进程忽然退出，此时该进程就是一个**孤儿进程**。作为一个进程，需要找到一个父进程，否则这种进程在退出之后没人回收他的进程描述符，空耗内存。此时该进程会找到一个父进程，如果自己所在的进程组没人收养，那就作为init进程的子进程

Inter-process Communication

Processes within a system may be **independent** or **cooperating**. Cooperating process can affect or be affected by other processes, including sharing data. And cooperating processes need **inter-process communication (IPC)进程间通讯**

系统中的进程可以是独立或协作的，协作进程可以影响其他进程或被其他进程影响，包括共享数据。而协作进程需要**进程间通信(IPC)**

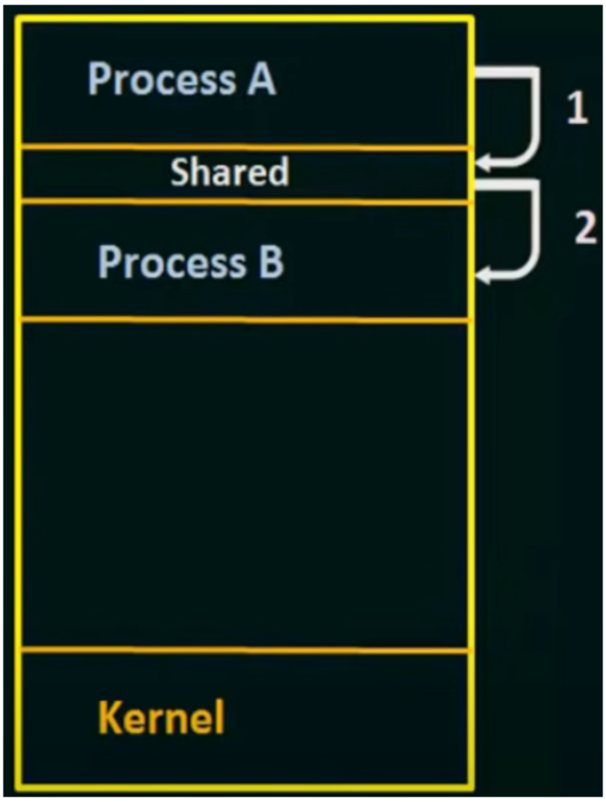
- Reasons for cooperating processes:
    - **Information sharing信息共享**
    - **Computation speedup计算加速**
    - **Modularity模块化**
    - **Convenience方便**
- Two fundamental models of IPC

- **Shared memory共享内存**
- **Message passing消息传递**

Shared memory

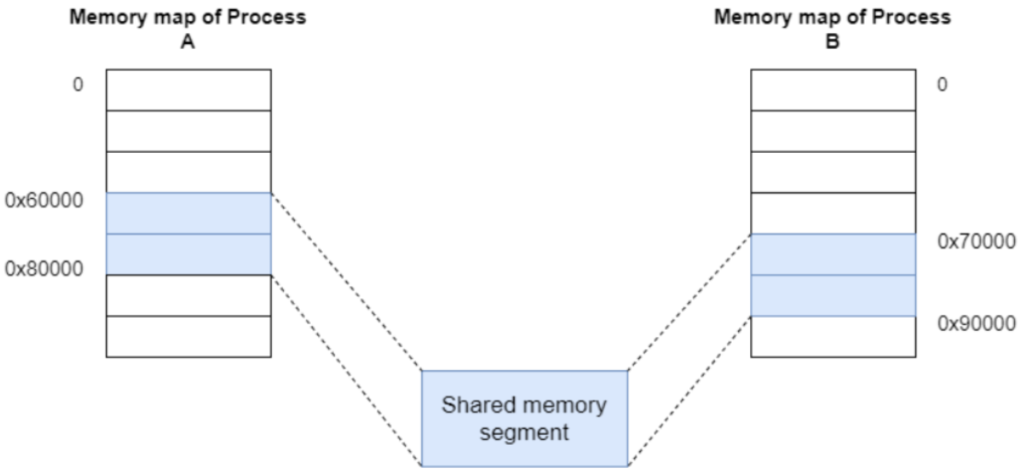
A region of **memory shared by cooperating processes** is established. Processes can then exchange information by reading and writing data to the shared region.

建立由协作进程**共享的内存区域。**进程可以通过向共享区域读取和写入数据来交换信息



The processes initially set up a region of their virtual memory to use for the IPC. Once the region is established within the process, the process issues a system call to request that the kernel make the region shared. After the initial system call to set up the shared memory, the processes can read from and write to the region just as it would access non-shared data on its own heap. This data then appears within the context of the other process automatically. There is no explicit system call required to read the new data.

一个进程设置了它的虚拟内存区域以用于IPC。一旦在进程中建立了区域，进程就会**发出系统调用请求内核共享该区域。**在设置共享内存的初始化系统调用之后，进程可以读取和写入该区域，就像访问其自己堆上的非共享数据一样。然后，**此数据会自动出现在其他进程的上下文中**，读取新数据不需要明确的系统调用。



在两个进程各自的地址空间中，共享内存的地址可以是不一致的，但是都指向同一块区域

The communication is under the control of the users processes not the operating system.

通信在**用户进程**而非操作系统的控制下(共享内存是用户态通讯)

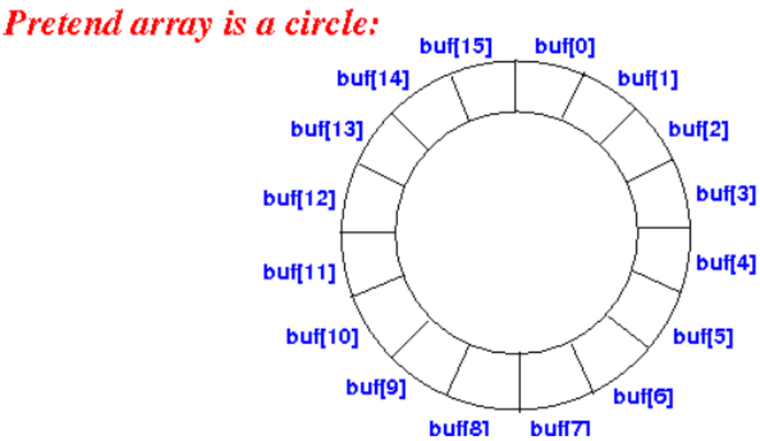
Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

在使用共享内存时，存在的主要问题是需要提供一种机制，允许用户进程在访问共享内存时同步它们的操作

Bounded-buffer

Producer-Comsumer model:

- Producer: Producer process produces information
- Comsumer: Comsumer process comsumes information



- The buffer is **empty** when `in==out`
- The buffer is **full** when `((in + 1) % BUFFER_SIZE) == out`

```c
#define BUFFER_SIZE 15

typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Producer:

```c
while (true) {
  /* produce an item in next produced */
  while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

- Comsumer:

```c
while (true) {
    /* consume the item in next consumed */
  while (in == out)
    ; /* do nothing */
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
}
```

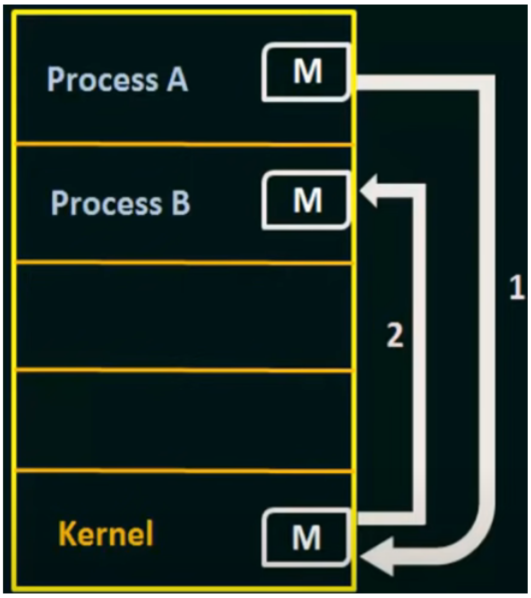
Message passing

Message passing is particularly useful in a **distributed environment**, where the communicating processes may reside on **different computers** connected by a network.

消息传递在**分布式环境**中特别有用，在这种环境中，通信进程可能驻留在通过网络连接的**不同计算机**上(此时不容易建立共享内存区块)

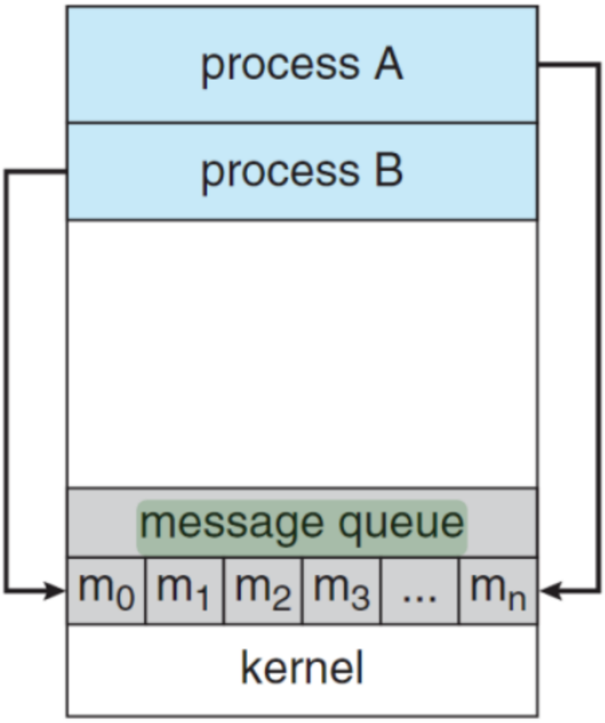
The user-mode process(sender) will copy the data into a buffer, then **issue a system call to request the data be transferred**. Once the kernel is invoked, it will copy the transferred data first into its own memory. The target process(receiver) will also **issue a system call to retrieve the data**.

用户态进程(发送者)将数据复制到缓冲区中，然后发出**系统调用请求传输数据。**一旦内核被调用，它将首先将传输的数据复制到自己的内存中。目标进程(接受者)还将**发出系统调用以检索数据**



In message passing, every piece of data exchanged requires two system calls: one to read and one to write. In addition, the transferred data must be copied twice: once into the kernel memory and once into the receiving process.

在消息传递中，每交换一条数据都需要两次系统调用：一次读取，一次写入。此外，传输的数据必须复制两次：一次复制到操作系统内核，一次复制到接收进程内



Shared memory vs Message passing

If processes P and Q wish to communicate, they need to:

- Establish a **communication link** between them
- Exchange messages via send/receive

Implementation of communication link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
    - Direct Communication直接通讯——端到端
    - Indirect Communication间接通讯——端到服务器再到端(在操作系统中，可以表示为：进程→内核→进程)
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Comparison

The shared memory techniques only require a **one-time performance penalty** during the set-up phase. Once the memory has been shared, there is no additional penalty, regardless of the amount of data transferred.

共享内存技术在设置阶段只需要**一次性的性能损失。**共享内存后，无论传输的数据量如何，都不会产生额外的损失

If the two processes will be **exchanging a lot of data** back and forth repeatedly, shared memory performs very well. While the work to **set up the shared memory** is **expensive**.

如果两个进程将反复来回**交换大量数据**，则共享内存性能非常好。而**设置共享内存**的工作是**昂贵的**

If processes only need to **exchange a single message of a few bytes**, shared memory will perform very poorly. **Message passing** techniques impose **significantly smaller overhead** to set up a one-time data exchange.

如果进程只需要**交换几个字节的单个消息**，共享内存的性能会很差。**消息传递**技术为建立一次性数据交换施加的**开销要小得多**

- **Advantage of message passing**
  - Useful for exchanging smaller amount of data
    - 在少量数据传输时很好用
  - Easier to implement in distributed system
    - 在分布式系统中容易实现
- **Advantages of shared memory**
  - Can be faster than message passing, since once the shared memory is established, all accesses are treated as routine memory accesses
    - 当共享内存区间设置完成，一切信息传输都像正常的内存读写一样高速
  - In contrast, message passing are typically implemented using system calls. Time consuming!
    - 信息传递需要多次系统调用，系统用户态和内核态切换也是耗时的
  - Disadvantage: synchronization
    - 需要同步

  - Shared memory suffers from cache coherency issues
    - 会受到缓存一致性的影响(也是涉及同步的内容)

OS Chapter 3 OS Synchronization

Background

Processes can execute concurrently, but may be interrupted at any time, partially completing execution. **Concurrent access to shared data** may result in **data inconsistency**. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

进程可以并发执行，但其随时可能被中断，只有部分完成执行。**并发访问共享数据**可能会导致**数据不一致。** 维护数据一致性需要机制来确保协作进程的有序执行

Race condition

A **race condition** is a situation where several processes access and manipulate the same data concurrently and the outcome of the execution **depends on the particular order** in which the access takes place.

竞争条件是指多个进程同时访问和操作相同数据并且执行结果**取决于访问顺序**的特定的情况

Race condition problems often occur when:

- one process does a "**check-then-act**" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X)
  - 一个进程执行"**先检查后操作**"（例如，"检查"值是否为X，然后"执行"以X为条件的操作）
- and **another process** does something to the value in between the "check" and the "act".
  - **另一个进程**对"检查"和"操作"之间的值进行了操作

Which of the following can cause a race condition?

- [ ] Read-read
- [x] Write-write
- [x] Read-write/Write-read
  - One reader and one writer
  - One reader and multiple writers
  - Multiple readers and one writer

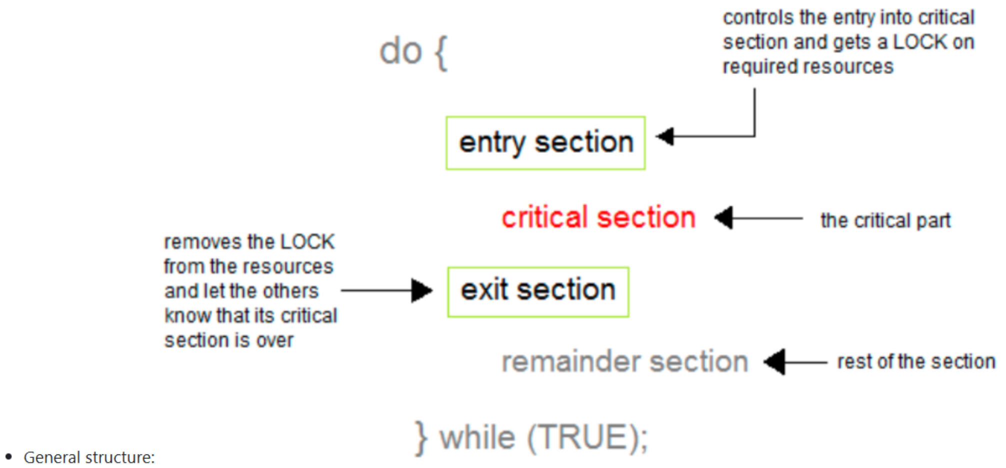
If there is **only one producer and one consumer**, the solution for the producer-consumer problem does not have race condition.

If there are **mulitiple producers or multiple consumers**, the solution for the producer-consumer problem have race condition.

Critical Section Problem

Critical Section Definition

- When more than one process access a same code segment that segment is known as **critical section**
  - 当多个进程访问同一代码段时，该段称为**临界区**
- Critical section contains **shared variables** or resources which are needed to be **synchronized** to maintain consistency of data variable
  - 临界区包含需要**同步**的**共享变量**或资源，以保持数据变量的一致性
- a critical section is group of instructions/statements or region of code that need to be **executed atomically**
  - 临界区是一组需要**原子执行**的指令/语句或代码区域
- General structure:



Solution to Critical-Section Problem

- A solution to the critical-section problem must satisfy the following **three** requirements:
  - 临界区问题的解决方案必须满足以下三个要求：
    - Mutual Exclusion
    - Progress
    - Bounded Waiting

Mutual Exclusion

- Mutual Exclusion - When one process is executing in its critical section, no other process is allowed to execute in its critical section
  - 互斥 —— 当一个进程在其临界区执行时，不允许其他进程在其临界区执行

Progress

- Progress – No process running outside the critical section should block the other interesting process from entering into a critical section when in fact the critical section is free
  - 进展 —— 在临界区之外运行的任何进程都不应阻止其他感兴趣的进程进入临界区，而此时实际上临界区是空闲的

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
    - 如果没有进程在其临界区执行，并且存在一些希望进入其临界区的进程，则不能无限期地推迟选择下一个将进入临界区的进程

Bounded Waiting

- Bounded Waiting - A bound must exist 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
    - 有界等待 —— 在进程发出进入其临界区的请求之后和该请求被授予之前，必须存在允许其他进程进入其临界区的次数的界限
- No process should have to wait forever to enter into the critical section. there should be a boundary on getting chances to enter into the critical section
    - 任何进程都不应该永远等待才能进入临界区。 获得进入关键部分的机会应该有一个界限

Lock And Unlock