OS 是中断/事件驱动. 中断: 硬件向 CPU 发信号, 软件**错误**/**系统调用**. 软件中断-trap. 中断向量包括所有中断处理程序的地址（函数指针）. 一个中断正在被处理时, 中断禁用. signal: 硬中断, 被动, 异步中断. I/O 设备 trap: 软中断, 主动, 同步（一定是一个原子指令执行结束后产生的）原子操作 (屏蔽中断). 存储结构-速度. 先查缓存, 若有则使用, 没有则调入缓存. 数据一致性. **API**(用户应用程序和硬件的唯一接口-通用目的) 用户大多根据 API 编程, 语言库实现. 一个 api 调用可能涉及 0/多个系统调用. 如 malloc, free, printf. **系统调用**(进程控制, 文件管理, 设备管理, 信息维护（获取时间/日期）, 通信)运行程序的 OS 接口, 通过软件中断向内核发出请求, 通常作为汇编语言指令提供.

CPU 保护-硬件支持, 用户模式 1, 内核模式 0. 特权指令: 内核模式执行, 被用户系统调用请求. 跳转到子程序是(非)特权指令. 内存保护: 基址寄存器+限长寄存器. CPU 保护: 定时器-用于分时, 加载定时器是特权指令. OS-管理、分配资源, 控制程序、IO 操作. 多道程序设计: 最大限度利用 CPU, 分时: 交互, 减小响应时间. **操作系统调度**:OS 能有机会使用 CPU—用户进程不使用 CPU 时：自愿退出(不需要抢占): 1. 执行完毕; 2. 要做 0, 然后把自己 block; 被动退出(抢占): 1. 时间片时间到了, 时钟硬件中断; 2. 执行出错, 硬件处理; 3. OS 响应中断/0 操作。

文件管理:

操作系统抽象(进程线程-CPU，地址空间-内存，文件-磁盘)

文件（长期储存，大量信息，掉电不丢失，多进程并发访问）信息存在磁盘上，“文件”一逻辑存储单元，由 OS 管理

对用户：文件显示；命名保护；允许的操作　对 OS 设计人员：决定是否执行有链接的文件；文件块映射到磁盘区；跟踪空闲存储空间

命名：除了可执行文件（.com.exe）文件类型只能被应用程序识别（不是 OS）；提供 OS hint 用合适的方法处理文件

文件有名称、数据、**属性**(是 OS 维护的关于文件的数据，而不是存储在文件中的用户数据->文件的**元数据**）管理文件所需要的文件元数据，都记录在 **FCB**

属性存储：磁盘的目录文件**目录**（将*文件名*映射为*文件属性*的表）：OS 能识别的有结构文件，由**记录**（目录项）组成，每个记录对应一个放在该目录下的文件/子目录，每条记录是一个文件控制块 **FCB**（未引入 inode 时可以这么说）

目录结构：目录文件+inodes **记录**:定长组块, 变长非跨越式组块(记录变长, 组块大小不变), 变长跨越式组块.

文件逻辑结构（纯字节序列-无结构|OS 不关心内容，记录序列-有内部结构的固定/变长序列，树-键可访问记录|大型商业数据处理）

域：基本数据单元；记录：一个数据元素；文件：相关记录组成

顺序文件（定长-键排序，可变长度-时间顺序排列|穷举访问）**索引文件**（可使用变长记录|信息及时性高 ie 航空库存控制|空间换时间）

文件访问：**顺序访问**（文件的磁带模型顺序处理|操作：按块读写下一步、重置 ie 将文件指针定位起点、跳过、倒带 n 记录）**直接（相对）访问** pc, pc=pc+1（磁盘|访问文件的任意位置的元素|包括 block#作为文件参数|随机访问）

索引访问（index 包含指向记录（文件块）的指针）

FS(文件系统)通过系统调用执行文件操作（创建/删除; 创建文件，初始文件属性；打开/关闭：在内存中加载文件属性和磁盘地址；读/写, 追加：从当前位置开始将数据从/传输到缓冲区；seek：在随机访问文件中：重定位文件指针）

打开文件：在磁盘搜索文件 F 的目录结构；将磁盘的目录条目复制到内存的打开文件表中；返回一个文件句柄（文件描述符->文件指针，指向 openfile 表中的条目）用于后续 r/w()系统调用

打开文件表：**per-process**~（跟踪该进程打开的所有文件、文件指针、访问权限和计数信息）**system-wide**（每个 per 表中的项指向这个表，包括进程无关的信息，打开数-判断何时 close）

原始**分区**（没有 FS）卷:带 FS 的格式化分区. 分区可以是磁盘的一部分/多个磁盘；软盘没有分区. 文件系统使用**目录**存储分区中元数据的信息. mount 表:记录已挂载设备信息的表格.

目录结构：单级目录结构（包含所有文件）两级目录结构（每个用户私有目录）树状目录结构（依赖路径名，工作目录）无环图目录（使用链接共享文件/目录，符号链接，一个文件可以有多个绝对路径|*删除*文件留下链接->悬挂指针|当参考计数器为 0 时删除文件）通用图目录（加链接不检测是否有环则可能转化，遍历搜索无限循环，删除文件用 counter 方法失效|第一遍便利整个图形标记可访问的文件和目录，第二次收集释放所有没被标记的内容|创建链接使用循环检测）

Inode（磁盘 inode 依赖 FS, 定位保存文件数据的磁盘块-不是文件，内核 inode 独立 FS，暂态管理信息）目录文件值包含一个列表of(**文件名, inode 号**)对，inode 号定位一个**包含文件属性**的数据结构(FCB/inode)**根节点的 inode 号(0)必须已知 open** 生成的 IO 量与路径名长度、目录大小成正比, **read** 查询 inode、**更新** time etc（对内存操作）

目录操作**系统调用** link/ulink 文件保护(访问类型 RWX，人|物理损坏-RAID|访问不当-密码）->(**访问控制列表 ACL**~与每个文件和目录关联)每个用户(userID, groupID 每个文件（owner, group, others 由 owner/root 设置描述对应人的权限）精简访问控制列表：每个用户-1ACL，每个文件-3ACL（RWX）ie owner:7=RWX=111

用户-磁盘一组文件：OS-磁盘连续逻辑块. 内存-磁盘块（1KB~4KB 一个~512B/多个扇区）传输. 一个 OS 支持多于 1FS 类型. FS-用户程序+物理存储接口.

磁盘：引导控制块/分区（~分区第一个块，空表示没有 OS）分区控制块/分区（分区详细信息）FCB/文件（）目录结构/FS（组织文件）

每个分区有一个独立的 FS. 磁盘块/扇区 o 包含 MBR（主引导记录）-BIOS 读入引导计算机-MBR 定位表中活动分区, 在块 o 中加载执行引导块

硬盘恒定角速度|外轨浪费空间, 光盘恒定线速度. **数据访问/读写时间=寻道时间+旋转延迟+数据传输时间. 存取时间=寻道时间+旋转延迟. 磁盘调度**:FIFO 先进先出(公平, 局部性|平均寻道时间较长)SSTF 最短服务时间优先(饥饿)SCAN 扫描（电梯）(兼顾了较小的寻道时间和**减缓**饥饿|两端)C-SCAN 循环扫描（兼顾了较小的寻道时间和**减缓**饥饿|磁臂黏着）

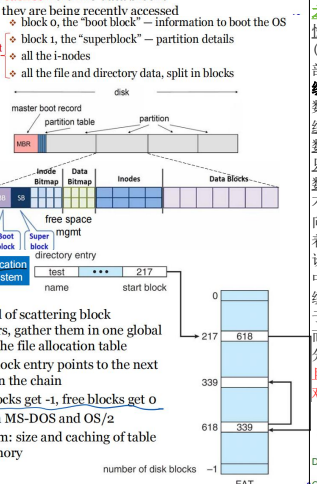
内存：FS 管理提升性能（缓存）.

文件块分配：连续分配(连续磁盘块|~动态内存分区|简单访问速度快局部性高|外部碎片声明大小|CDROMs|压缩算法释放空间), 链接分配（显式链式分配：每个文件分散在块中彼此链接|~分页|文件大小可变|访问时间顺序ok 随机无效, 指针-磁盘空间浪费；隐式链式：**FAT**|表需要放在内存中-大小和缓存），索引分配（块指针放索引块中|**文件分配表**:文件名+索引块序号|起始块+长度|索引链接多级索引/组合）

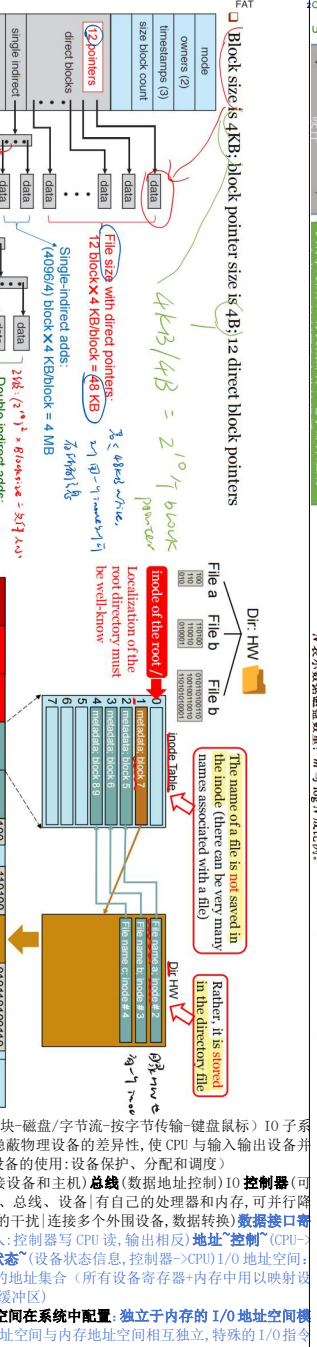
磁盘分配表:位图（bitmap, 每个块一位. 1-空闲|简单高效需缓存才性能好），链表(~链接分配|空闲块链接, 第一个空闲块指针保存在磁盘特殊位置，缓存在内存中->不遍历**FAT**)，分组（链接索引分配），计数（连续分配）

In-Memory Structure

- Used for FS management and performance improvement (via caching)
- An in-memory mount table
  - contain info about each mounted volume
- An in-memory directory-structure cache
  - hold information of recently accessed directories
  - contain (file/subdirectory name, inode number) pairs
- The system-wide open-file table
  - A copy of FCB of each open file
- The per-process open-file table
  - pointer (file handler /descriptor) to the corresponding entry in the above table
- Buffers/caches hold file data blocks
  - When they are being recently accessed



- instead of scattering block pointers, gather them in one global table: the file allocation table
- each block entry points to the next block in the chain
- end blocks get -1, free blocks get 0
- used in MS-DOS and OS/2
- problem: size and caching of table in memory



IO 设备（块-磁盘/字节流-按字节传输-键盘鼠标）IO 子系统功能(隐藏物理设备的差异性, 使 CPU 与输入输出设备并行, 共享设备的使用: 设备保护、分配和调度)

端口(连接设备和主机) **总线**(数据地址控制) IO **控制器**(可操作端口、总线、设备|有自己的处理器和内存, 可并行降低对 CPU 的干扰|连接多个外围设备, 数据转换)**数据接口寄存器**(输入: 控制器写 CPU 读, 输出相反)**地址~控制~**(CPU->控制器)**状态~**(设备状态信息, 控制器->CPU)I/O 地址空间: I/O 功能的地址集合（所有设备寄存器+内存中用以映射设备的数据缓冲区）

I/O 地址空间在系统中配置: **独立于内存的 I/O 地址空间模式**(I/O 地址空间与内存地址空间相互独立, 特殊的 I/O 指令对 I/O 地址空间访问 | 每个控制寄存器被分配一个 I/O 端口号, I/O 端口->I/O 端口空间 | CPU 使用特殊的 I/O 指令 ie IN REG, PORT 将 I/O 端口 PORT 的内容读入 CPU 寄存器 REG; OUT PORT, REG 将 CPU 寄存器 REG 的内容写入 PORT | CPU 内存指令: I/O 指令-IN R0, 4 将 I/O 端口 4 的内容读入 CPU 寄存器 R0; 内存指令-MOV R0, 4 将内存地址 4 的内容读入 CPU 寄存器 R0 | I/O 端口数少, 少量地址线, 译码简单, 寻址快) **映射到内存空间~(内存映射 I/O**| 将所有控制寄存器映射到内存空间 | 每个控制寄存器-唯一的内存地址, 不会有内存被分配到这一地址 | CPU 的内存指令与 I/O 指令同 | CPU 访问 I/O 的操作更加灵活, 端口有较大的编址空间, 访问的保护机制可由虚拟存储管理系统来实现 | 内存的可用容量变小) **混合模式**(数据缓冲区-内存映射 I/O, 控制寄存器-单独的 I/O 端口)

CPU 和 IO 通信: **可编程 I/O**(CPU 轮询设备检查 IO 是否完成) **中断 I/O**(**设备->controler 寄存器->CPU->内存**(缓冲区) | **硬件**, IO 控制器发中断信号, 系统找到相应中断处理程序、执行 | 以**字节为**单位进行 I/O, 控制器请求一次中断, 对于块设备的 I/O 低效) **直接存储器访问 DMA**(**IO->DMA->内存** CPU 只在传输的起/末干预 | 数据块传输, 块的传送在控制器的控制下完成 | 减少 CPU 对 IO 的干预, 提高 CPU 和 IO 的并行程度)

缓冲:缓和 CPU 与 I/O 设备之间速度不匹配; 减少对 CPU 的中断频率, 放宽对 CPU 对中断响应时间的限制; 提高 **CPU 和 I/O 设备之间的并行性**实现方法:硬件缓冲; 软件缓冲/内存缓冲区. **有效性前提**:阵发性(包括突发性与间歇性)的快, 而不是持续性的快(各种 I/O 缓冲区) | 与缓存(Cache)所基于的访问局部性原理(局部的数据被连续多次访问)不同.

缓冲策略:单缓冲(max(T, C)+M | 块设备:CPU 处理+设备将下一块数据输入到缓冲区-并行进行, 提高 I/O 的速度 | 字符设备输入: 缓冲区暂存一行数据, 在输入期间用户进程被挂起以等待输入数据; 输出时用户进程将一行数据放入缓冲区后继续执行程序, 只有在缓冲区中的数据没有被设备取完而用户又想输出第二行数据时, 才需要把它挂起) 双缓冲(C<T 块设备连续输入, C>T CPU 不停 | 块设备输入: 先将一块数据送入第一缓冲区, 装满后便转向第二缓冲区, OS 可以从第一缓冲区中移出数据送用户空间, 接着由 CPU 对数据进行处理, CPU 和设备的并行工作程度更高 | 字符设备: 能消除用户的等待时间, 如 I/O 设备在输出第一个缓冲区中的数据时用户要求输出第二行, 可以将输出数据送入第二个缓冲区而不会发生阻塞) 循环缓冲（若 CPU 处理数据的速度快于数据 I/O 的速度, 即使双缓冲, 用户进程也必须经常等待 I/O 而阻塞, 用多个组织成环状的缓冲区来改善）缓冲池(多个进程分享一个缓冲池-包含多个缓冲区) 直接涉及到**硬件具体细节、且与中断无关**的操作-在**设备驱动程序层**完成; **没有涉及硬件**的, 对**各种设备都需要进行**的管理工作-在**设备独立性软件层**完成

级别	说明	磁盘请求	数据可用性	大 I/O 数据量传送能力	小 I/O 请求率
0	非冗余	N	低于单个磁盘	很高	读和写都很高
1	镜像	2N	高于 RAID2、3、4 或 5；低于 RAID6	读时高于单个磁盘；写时与单个磁盘相近	读时最快为单个磁盘的两倍；写时与单个磁盘相近
2	通过汉明码实现冗余	N + m	明显高于单个磁盘；与 RAID3、4 或 5 可比	所有列出方案中最高的	约为单个磁盘的两倍
3	交错位奇偶校验	N + 1	明显高于单个磁盘；与 RAID2、4 或 5 可比	所有列出方案中最高的	约为单个磁盘的两倍
4	交错块奇偶校验	N + 1	明显高于单个磁盘；与 RAID2、3 或 5 可比	读时与 RAID0 相近；写时明显慢于单个磁盘	读时与 RAID0 相近；写时明显慢于单个磁盘
5	交错块分布奇偶校验	N + 1	明显高于单个磁盘；与 RAID2、3 或 4 可比较	读时与 RAID0 相近；写时慢于单个磁盘	读时与 RAID0 相近；写时通常慢于单个磁盘
6	交错块双重分布奇偶校验	N + 2	所有列出方案中最高的	读时与 RAID0 相近；写时明显慢于 RAID5	读时与 RAID0 相近；写时明显慢于 RAID5

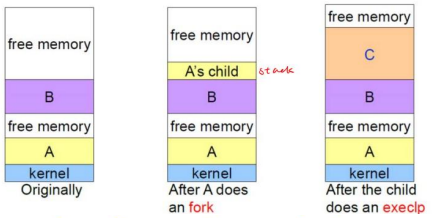
N 表示数据磁盘数量, m 与 log N 成比例。

进程一个程序可对应多个进程, 进程-孤立实体. PCB-进程唯一标志, 存在 OS 内存空间（内核地址空间-高址）中. **上下文切换**:完全开销, 硬件支持: 寄存器组.

父进程和子进程共享所有/部分/0 资源. 并发/父等待子终止. 地址空间; 子复制父: 共享变量通信；子有一个程序装入: 消息传递通信. Terminate when the last statement is executed or exit() is called: All resources of the process, including physical & virtual memory, open files, I/O buffers, are deallocated by the OS. Parent may terminate execution of children processes by specifying its PID (abort) Child has exceeded allocated resources Task assigned to child is no longer required. 多线程：一个地址空间；可在不同处理器并行处理；寄存器切换必须，但是不需要内存管理. 调度的基本单位. pc，寄存器组（TCB），栈；共享数据、代码、OS 资源. 用户线程-OS 无关.

UNIX/Linux process creation

- fork system call
  - Create a new (child) process
  - The new process **duplicates** the address space of its parent
  - Child & Parent execute **concurrently** after fork
  - Child: return value of fork is 0
  - Parent: return value of fork is PID of the child process
- execlp system call
  - Load a new binary file into memory – destroying the old code
- wait system call
  - The parent waits for one of its child processes to complete
- Memory space of fork():
  - Old implementation: A's child is an exact copy of parent
  - Current implementation: use copy-on-write technique to store differences in A's child address space

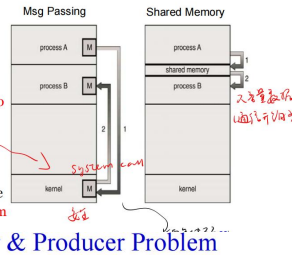


Inter-Process Communication

- IPC: a set of methods for the exchange of data among multiple threads in one or more processes

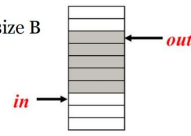
Communication Methods

- Shared memory:
  - Require more careful user synchronization
  - Implemented by memory access: faster speed
  - Use memory address to access data
- Message passing:
  - No conflict: more efficient for small data
  - Use send/recv message
  - Implemented by system call: slower speed



Consumer & Producer Problem

- Paradigm for cooperating processes
  - Producer process produces information that is consumed by a consumer process
- Buffer as a circular array with size B
  - next free: *in*
  - first available: *out*
  - empty: *in = out*
  - full: *(in+1) % B = out*
- The solution allows at most (B-1) item in the buffer



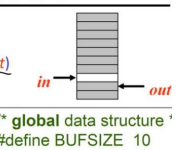
Shared-Memory Solution

```
/*producer*/
while (1) {
    while (((in + 1) % BUFFER_SIZE) == out)
        ; //wait if buffer is full
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
/*consumer*/
while (1) {
    while (in == out); //wait if buffer is empty
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

"in" only modified by producer

"out" only modified by consumer

```
/* global data structure */
#define BUFFSIZE 10
item buffer[BUFSIZE];
int in = out = 0;
```



IPC：直接/间接(共同邮箱-os/进程所有)/阻塞/非阻塞(缓冲区 0 容量-阻断发送接收，无界-发送方永远不阻塞)

Direct Communication

- Processes must name each other explicitly:
  - Send (P, message) – send a message to proc P
  - Receive (Q, message) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - One-to-One relationship between links and processes
  - The link may be unidirectional, but is usually bi-directional

1B=8b 按字节编址-每个存储单元大小为 1B8GB 内存=内存中可以存放 8GB=2^33 个地址=33 个二进制位表示地址

操作系统**并发**:分时实现.

OS 好处:隔离(时间:时间片;内存:空间划分 CPU 隔离：分时+上下文切换)+分配（策略和机制的分离）

OS 功能:提供服务（软件接口）+管理（硬件）

❑ Solution for producer-consumer problem:

```
/*producer*/
while (1) {
        send (consumer, nextProduced);
}
/*consumer*/
while (1) {
        receive (producer, nextConsumed);
}
```

❑ limited modularity: if the name of a process is changed, all old names should be found

Socket-IP 和端口标识，字节流.

进程调度：长期调度，中期(内存)，短期(->CPU)

调度延迟：调度时间，中断重新启动时间，上下文切换时间.

❑ CPU utilization: keep the CPU as busy as possible

❑ Throughput: # of processes that complete their execution per time unit

❑ Turnaround time: amount of time to execute a particular process

❑ Waiting time: amount of time a process has been waiting in the ready queue

❑ Response time: amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

	Non-Preemptive	Preemptive
Basics	FCFS	RR
Prediction	SPN	SRTF
Priority	Priority	Preemptive Priority
Real time	RM	EDF

相同长度 RR 不好. Shortest Process Next/First (SPN)

Shortest Remaining Time First (SRTF)

SPN和SRTF

SPN/SRTF是您可以最大限度减少平均周转时间的最佳选择

❖ 可证明最优（非抢占式中的SPN，抢占式中的SRTF）

❖ 由于SRTF始终至少与SPN一样好，因此重点关注SRTF

SRTF与FCFS和RR的比较

❖ 如果所有工作都是相同的长度?

·SRTF与FCFS相同（即，如果所有作业的长度相同则FCFS是最好的）

❖ 如果作业的长度不同怎么办?

·SRTF（和RR）：短期工作不会落后于长期工作

多级队列调度

❑ 过程通常可以根据其目的进行分类和行为，例如。

*系统进程

*互动过程

*交互式编辑流程

*批处理

❑ 另外，将进程分为两个主要类别：前台进程和后台进程

前台进程需要响应，CPU突发量小；后台进程CPU突发量大，不具有交互性。多级队列调度为每一类进程维护一个队列

+队列的优先级递减-例如，(系统进程 优先级最高，批处理优先级最低)

*进程在启动时被永久分配给特定队列，并且不会在不同队列之间移动

❑ 每个队列都有自己的固定优先级

高优先级 → 低优先级

系统进程 / 交互过程 / 交互式编辑过程 / 批处理

❑ 每个队列也可以有自己的调度算法和参数（例如时间片大小）

批处理进程可以采用先到先服务的调度方式运行，也可以采用具有非常大时间片的循环调度方式运行（对于失控进程）。其他进程通常采用循环调度方式运行。在高优先级队列中也可能有实时进程，对该队列使用实时调度算法

❑ 多级反馈队列调度：允许进程在不同优先级队列之间移动

❖ Frequently, all queues are scheduled using round-robin scheduling, with shorter time-slices for higher-priority queues

❑ New processes are added to end of highest priority queue

❑ If a process is preempted by the system, it is sent to the next lower queue

❑ Lower-priority processes can also be promoted for good behavior ☺

Real-time scheduling

❑ Rate-Monotonic (RM) algorithm:

❖ Tasks assigned a priority inversely based on their period

❖ The shorter the period, the higher the priority!

❖ Fixed-priority scheduling algorithm

❑ Earliest-Deadline-First (EDF) algorithm:

❖ The earlier a process' deadline, the higher its priority!

❖ Processes must state their deadlines to the scheduler

❖ Dynamic priority algorithm

❑ Race condition

❖ A situation where several processes access and manipulate the same data (critical section)

❖ The outcome depends on the order in which the access take place

❖ Prevent race conditions by synchronization (mutual exclusion)

– Ensure only one process at a time manipulates the critical data

同步：互斥(数据一致性)，条件同步(特定操作顺序)

原子指令、全局变量、忙等待.用户进程不能关中断.

信号量:wait-阻塞态.3 原子操作.强信号量 FIFO.管程：把临界区集中起来管理，共享临界资源只需调用管程。进程只能通过调用管程的过程间接访问其数据结构.联合进程图、资源分配图.无环无死锁，单点成环则死锁.死锁:)2 进程(1 进程只有饥饿)死锁-block,活锁-ready

```
boolean locked = false;  //must be in shared memory
P1:                                  P2:
while (true) {                       while (true) {
  while (locked) do {nothing};         while (locked) do {nothing};
  locked = true;                       locked = true;
  C.S.1;                               C.S.2;
  locked = false;                      locked = false;
  remainder section;                   remainder section;
}                                    }
```

```
while (true) {
  P1WantIn = true;
  while (P2WantIn) do {nothing};
  C.S.1;
  P1WantIn = false;
  remainder section;
}
```

```
while (true) {
  P2WantIn = true;
  while (P1WantIn) do {nothing};
  C.S.2;
  P2WantIn = false;
  remainder section;
}
```

```
boolean P1WantIn = false; //must be in shared memory
boolean P2WantIn = false; //must be in shared memory
P1:                                  P2:
while (true) {                       while (true) {
  P1WantIn = true;                     P2WantIn = true;
  If (P2WantIn) P1WantIn = false;      If (P1WantIn) P2WantIn = false;
    else { C.S.1;                        else { C.S.2;
           P1WantIn = false;                    P2WantIn = false;
           remainder section;}                  remainder section;}
}                                    }
```

Dekker's algorithm: two with alternatir

```
boolean P1WantIn = false;  //must be in sh
boolean P2WantIn = false;  //must be in sh
int turn= 1;  //must be in shared memory
P1:
while (true) {
  P1WantIn = true;
  while (P2WantIn) {
    if (turn = 2){
      P1WantIn = false;
      while (turn = 2) do {nothing};
      P1WantIn = true;};
  }
  C.S.1;
  turn= 2;
  P1WantIn = false;
  remainder section;
}
```

Peterson's algorithm

```
boolean P1WantIn = false;  //must be in shared memo
boolean P2WantIn = false;  //must be in shared memo
int turn = 1;  //must be in shared memory
P1:
while (true) {
    P1WantIn = true;
    turn = 2;
    while (P2WantIn && turn == 2) do {nothing};
    C.S.1;
    P1WantIn = false;
    remainder section;
}
P2:
while (true) {
    P1WantIn = true;
    turn = 2;
    while (P2WantIn && turn == 2) do {nothing};
    C.S.1;
    P1WantIn = false;
    remainder section;
}
```

Enter C.S. when e
1. A process gets
2. The other proce

❑ More elegant, understand tha Dekker's algor

❑ If the order is reversed and a process is preempted after setting the turn variable but before setting the WantIn variable, then both processes can get into the critical section!

```
while (locked) do {nothing};
locked = true;  // lock
```

← Make it atomic

```
producer put(item) //放数据
  { item  nextp;
    if (count==n)        //所有缓冲区为满缓冲区
       cwait(notfull); //等待空缓冲区
    buffer(in)=nextp;
    in=(in+1)mod n;
    count=count+1;
    if( full队列不为空) csignal(notempty);
                    //唤醒等待满缓冲 区队列中的进程
  }
```

The Conditions for Deadlock

Design conditions for deadlock (create the swamps)

❖ 资源互斥 mutual exclusion—the design contains protected critical regions; only one process at a time may use these

❖ 持有等待 hold & wait—the design is such that, while inside a critical region, a process may have to wait for another critical region 进程在请求新资源时，其已获得的资源并不释放

❖ 不可抢占 no resource preemption —there must not be any hardware or OS mechanism forcibly removing a process from it 凡是可抢占资源均不构成死锁的原因！(见下页的解释)

Scheduling condition for deadlock (go to the swamps)

❖ 循环等待circular wait — two or more hold-&-wait's are happening in a circle: each process holds a resource needed by the next

Design + Scheduling Condition = Deadlock

处理死锁的方法有四种，可分类如下：

允许死锁发生

❖ 忽视这个问题，假设系统中不可能发生死锁：鸵鸟策略

❖ 死锁检测和死锁恢复

处理依据

死锁的四条件
(1) 资源互斥
(2) 持有等待
(3) 不可抢占
(4) 循环等待

不允许死锁发生

❖ 死锁预防：事先消除四条件中的任一个来预防死锁的出现(静态预防)

❖ 死锁避免：基于资源分配的当前状态做动态选择来避免死锁(动态避免)

资源分配图化简

❖ 找一个非孤立点进程结点且只有分配边，去掉分配边，将其变为孤立结点。

❖ 再把相应的资源分配给一个等待该资源的进程，即将某进程的申请边变为分配边。

❖ 如果进程-资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件。

❖ 如果能在进程-资源分配图中消去此进程的所有请求边和分配边，成为孤立结点。经一系列简化，使所有进程成为孤立结点，则该图是可完全简化的；否则则称该图是不可完全简化的。

系统为死锁状态的充分条件：

❖ 当且仅当该状态的进程-资源分配图是不可完全简化的。该充分条件称为死锁定理。

一开始的可用资源不需要与总的最大需求相同-死锁避免与死锁预防的最大区别

内存

虚拟内存管理：**驻留集管理**:决定给每个进程分配多少页框;当页框不够选取被置换出去的页面.**置换策略**:当页框不够,新的页面又一定要调入内存时,决定被置换出去的页面.抖动:当需要将一个新页面调入内存时,因内存空间紧张,不得不将一个老页面置换出去,而刚刚置换出去的老页面可能又要被使用,需要重新将它调入.

驻留集管理

❑ 可采取两种内存分配策略，即固定和可变分配策略。在进行置换时，也可采取两种策略，即全局置换和局部置换。于是可组合出以下三种适用的策略

❖ 固定分配局部置换：为每个进程分配固定页框数的内存空间，在整个运行期间不再改变。分配页框数少，会频繁缺页中断；分配页框数多，浪费资源，内存中运行作业少。

❖ 可变分配局部置换：为每个进程分配一定量的内存空间。若运行中频繁缺页中断，则再增加若干页框，直至缺页率减少到适当值为止。反之，若缺页率很低，则减少分配的页框数

❖ 可变分配全局置换：先为每个进程分配一定量的页框，OS有一个空闲页框队列，某进程缺页时从空闲队列取页框装页，当空闲队列为空后，OS再调出某一进程的页置换

❑ 第二种可变分配的主导权在置换算法，而第一种可变分配的主导权在驻留集管理。可变分配局部置换的关键是确定驻留集大小和变化的算法，即工作集策略

置换策略:**最优置换策略**(OPT)未来最长时间内不再使用的页面.无法实现的最优策略.**先进先出策略**(FIFO)-最直观也是效果最差的策略.**最近最久未使用策略**(LRU)可行的最优策略,但硬件代价高.**时钟策略**(CLOCK)LRU 的简化版本.

CPU 控制部件只能从主存中取指令、数据. 程序装载内存->执行. 每个进程有自己的地址空间. 地址重定位：逻辑->物理. 时间局部性：循环操作；空间局部性：顺序执行. 磁盘缓存：在普通物理内存中分配的内存区域, 减少访问磁盘的次数.

源程序要运行：符号名地址(编译：多目标模块)独立的逻辑地址(链接：+库函数链接->装入模块)统一的逻辑地址(装入：模块装入主存)物理地址. **程序和数据装入内存**:**绝对装入**-不需对地址修改;**可重定位**-多道程序, 装入时的修改：重定位. **动态**-程序执行才地址转换, 装入内存的是相对地址(硬件支持)内存中可移动. **动态分区放置算法**：最佳适配×，首次适配 √，下次适配—，最坏~. 内存紧凑, 覆盖系统调用, 交换

❑ 从搜索空闲区速度及主存利用率来看, 最先适应分配算法>下次适应分配算法和最佳适应算法比最坏适应算法性能好。

❑ 如果空闲区按从小到大排列, 则最先适应分配算法等于最佳分配算法。反之，如果空闲区按从大到小排列，则最先适应算法等于最坏适应分配算法。

❑ 空闲区按从小到大排列时，最先适应分配算法能尽可能使大空闲区址区，从而，在高地址空间有较多较大的空闲区来容纳大作业。下次适应分配算法会使存储器空间得到均衡使用。

❑ 最优适应分配算法的主存利用率最好，因为，它把刚好或申请要求的空闲区分给作业；但是它可能会导致空闲区分配后的部分很小。在处理某种作业序列时，最坏适应分配算法能最佳，因为，它选择最大空闲区，使得分配后剩余下来的空闲区不会太小，仍能用于再分配。

❑ 连续模式：给一个进程分配一个连续的内存空间

❖ 单一连续分配：只适用于单道程序系统

❖ 固定分区：分区大小固定

❖ 动态分区：分区大小依程序大小变化

❖ 伙伴系统：固定分区与动态分区的综合

❑ 非连续模式：给一个进程分配多个分散的内存空间

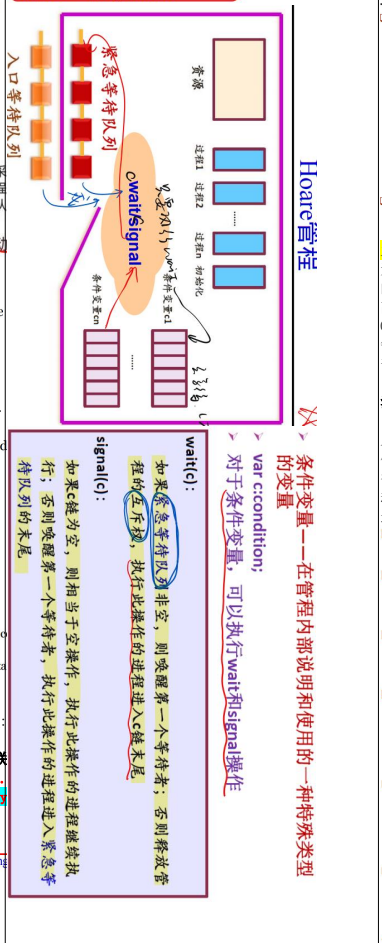
❖ 分页：存储单元固定

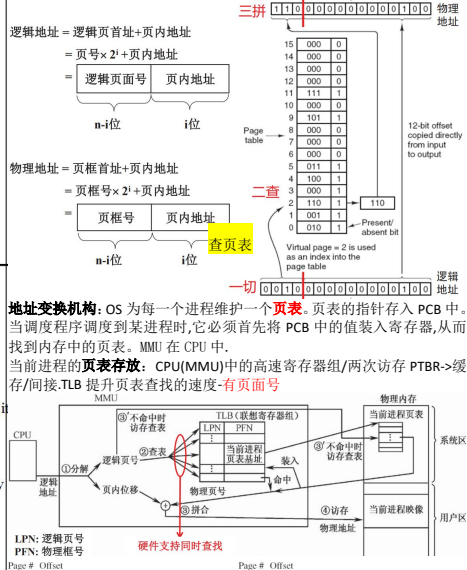
❖ 分段：存储单元依模块大小变化

❖ 段页式：分段与分页的综合

❖ 偏移量d也称页内地址。图中的地址长度为32位，每页大小为4KB，地址空间最多2^{20}（1M）个页。对于特定的机器，其地址长度一定的。

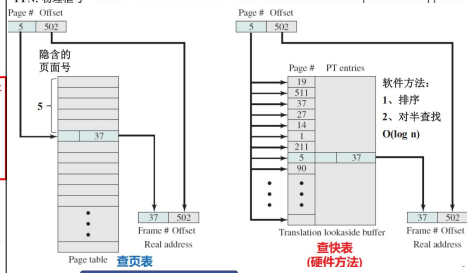
❖ 若逻辑地址为A，页面大小为L，则页号P和页内地址d可按下得：P=int(A/L)；d=A mod L

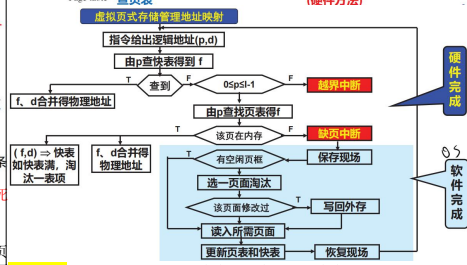




地址变换机构：OS 为每一个进程维护一个**页表**。页表的指针存入 PCB 中。当调度程序调度到某进程时,它必须首先将 PCB 中的值装入寄存器,从而找到内存中的页表。MMU 在 CPU 中.

当前进程的**页表存放**：CPU(MMU)中的高速寄存器组/两次访存 PTBR->缓存/间接.TLB 提升页表查找的速度-有页面号

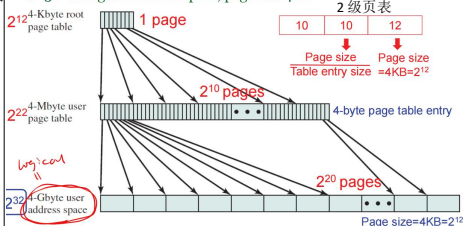




多级页表二级页表的问题:1 次内存访问变成 3 次内存访问,若次级页表不在内存,还有 1 次磁盘访问,系统速度大为下降

❑ A two-level hierarchical page table as an example

❑ 32-bit logical address space, page size 4KB



段是信息的逻辑单位.动态连接:在作业运行之前,并不把几个目标程序链接起来。要运行时,先将主程序所对应的目标程序装入内存并启动运行,当运行过程中需要调用某段时,才将该段(目标程序)调入内存并进行链接.