BUAA_OS_FINAL

BUAA_OS_FINAL
- 引言与启动
- 内存管理
- 进程管理
- IO管理
- 磁盘管理
- 文件系统
- 补充知识（期中）

引言与启动

批处理系统一次只执行一个任务，CPU在等待I/O时处于空闲状态，因此发展**多道程序设计**：在一个时间段内，**多个程序（进程）同时驻留在内存中**，操作系统通过**调度机制**来在它们之间切换，让**CPU总在运行某个进程**，从而提高效率。多道程序设计用户提交作业后无法实时交互，等待时间较长。随着计算机性能提升和**交互需求增加**，发展为时间片轮转调度的**分时系统。**

上面总结：

批处理系统的主要优点是系统的吞吐量大、资源利用率高、系统的切换开销较小是正确的；批处理系统的**最大缺点**是**无交互能力。**

操作系统中的 Spooling 技术，实质是将独占设备转换为共享设备的技术。

在一个操作系统中编译好的程序在另一个 **ABI 兼容** 的操作系统中无需重新编译就能运行。

内存中无法被利用的存储空间称为**碎片。**

在C语言之类的程序编译完成之后，已初始化的**全局变量或静态局部变量**保存在 data 段中，未初始化的**全局变量或静态局部变量**保存在 bss 段中。

内存管理

反置页表：**每个物理页框在页表中占一个表项**，记录该页框被哪个进程的哪个虚拟页占用。所以：**页表大小只与物理内存大小有关，与虚拟地址空间大小无关。反置页表的优点是节省内存。**反置页表可以建立快表。

在一个纯分页系统中，采用二级页表，一条访存指令成功执行，最多会产生 **3 次**实际访存操作（假设此过程中操作系统相关**代码**都已在 Cache 中，不涉及访存）。（**访问一级页表**（页目录）→ 取出指向二级页表的地址；**访问二级页表** → 取出指向物理页的页框地址；**访问实际数据页** → 取出真正的数据内容）。

分页存储管理技术，是用于虚存管理的技术，**但也可以用于物理内存管理。**

最优置换（Optimal）：从主存中移出永远不再需要的页面，如无这样的页面存在，则应选择**最长时间不需要访问**的页面。它会将内存中的页 P 置换掉，页 P 满足：从现在开始到未来某刻再次需要页 P，这段时间最长。也就是 OPT 算法会置换掉未来最久不被使用的页 。

s：进程平均大小；p：页的大小（单位：字节）；e：一个页表项的大小（单位：字节），则有：

$$\text{开销} = \frac{se}{p} + \frac{p}{2}$$

给进程大小求最优页面大小： p = √2se.

“页”是信息的“物理”单位，大小固定。 “段”是信息的逻辑单位，即它是一组有意义的信息，其长度不定。

虚拟地址空间可以大于、等于或小于物理地址空间。

特性	内碎片（Internal Fragmentation）	外碎片（External Fragmentation）
碎片位置	分配给程序的**块内部未使用的空间**	**空闲内存之间的零散空隙**
产生原因	分配的内存**大于实际所需**	多次分配释放后，**内存被零碎地占用**
是否连续	是：块内部的空闲空间	否：多个不连续的小空闲区

管理方式	能否消除外碎片	说明
分页	✅ **能，没有外碎片**	**页是固定大小、可以非连续映射物理地址**
分段	❌ 不能	每段仍需物理内存中**连续分配**
段页式	✅ 能	分段 + 分页，结合逻辑与物理管理的优点

消除**外部碎片**的方法：紧凑技术。注意是消除**外碎片**！

多级页表**能够减少页表占用内存的大小**。但是使用二级页表的平均访存性能**不如一级页表。**

覆盖与交换技术的区别：

- 覆盖可减少一个程序运行所需的空间。交换可让整个程序暂存于外存中，让出内存空间。
- 覆盖是由程序员实现的，操作系统根据程序员提供的覆盖结构来完成程序段之间的覆盖。交换技术不要求程序员给出程序段之间的覆盖结构。
- **覆盖**技术主要对**同一个作业**或程序进行。**交换**主要在作业或程序间**之间**进行。

Belady 现象：在FIFO等页面置换算法中，分配的页框数增多，但缺页率反而提高。

大端存储意思是高位在低地址。也就是正常读取顺序。

做二级页表题的时候，从一级页表项中拿到二级页表基地址，要加上二级页表偏移得到二级页表项，此时是二级页号乘上**页表项**大小！不是乘页面大小！

页表自映射：

三个概念：页表基址（从哪里开始映射4MB的页表）、一级页表基址（起始虚拟地址）、一级页表中映射自己的表项的虚拟地址。**“二级页表起始逻辑地址” 就是页表基址！**

构建方法

1. 给定一个页表基址PT_{base}，该基址需4M对齐，即：
   $$PT_{base} = ((PT_{base}) >> 22) << 22;$$
   不难看出，PT_{base}的低22位全为0。

2. 页目录表基址PD_{base}在哪里？
   $$PD_{base} = PT_{base} | (PT_{base}) >> 10$$

3. 自映射目录表项$PDE_{self\text{-}mapping}$在哪里？
   $$PDE_{self\text{-}mapping} = PT_{base} | (PT_{base}) >> 10 | (PT_{base}) >> 20$$

在工作集模型中，当**所有工作集的页面数**高于可用的物理页框数就会导致系统抖动。

MMU 是**执行**时重定位所需的硬件支持（不是装载）。

进程管理

LRU 算法做题：假设分配 k 个页框，那么给定一个序列，问输入一个新的数应当淘汰的页号。从该序列的最后一个数数起，第 k 个不同的数字就是需要淘汰的页号。

进程和程序**不是**一一对应的。

线程不共享栈空间和栈指针。如果需要线程之间共享数据，应该使用 **共享堆**（如使用全局变量、共享内存、同步机制等）而不是栈。

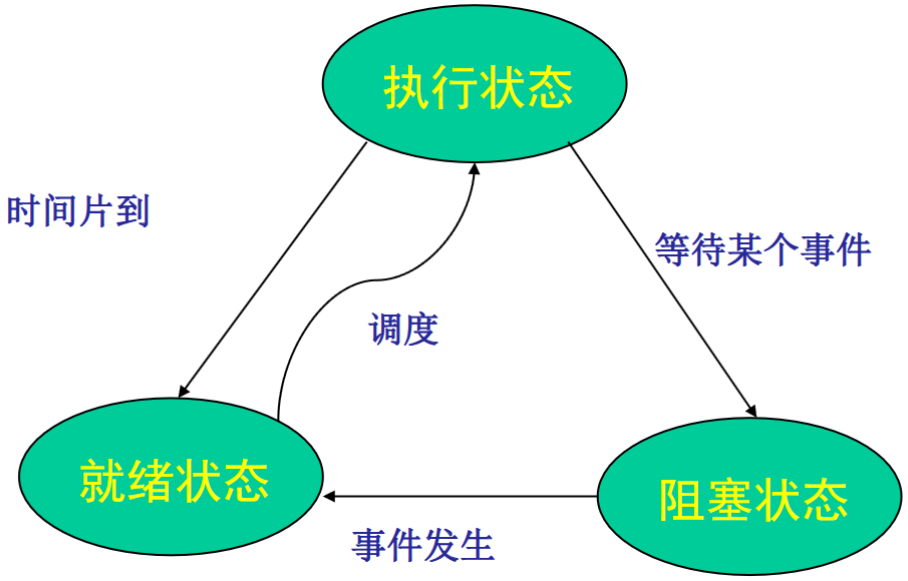
同一进程的所有线程**共享相同的页表**！因此线程上下文切换不会导致地址空间变化，也就**不需要刷新 TLB。**

抖动不是 CPU 忙闲问题，而是 **内存不足问题。**

从一个进程切换到另一个进程的上下文切换，**必须在内核模式下**由操作系统执行相应代码才能完成。

如果一个用户进程进入死循环，操作系统可以终止该用户进程执行。

进程三种状态以及相互转换发生的条件举例：



因此，一个进程被唤醒**并不立即重新占有 CPU。**而是进入就绪状态。

SJF（Shortest Job First，短作业优先） 是**理论上平均周转时间最短的调度算法，但它可能导致长作业饥饿**，因为短作业总是被优先调度。

模型类型	映射关系	是否支持并发系统调用	核心优劣对比
多对一（Many-to-One）	多用户线程 ↔ 1内核线程	❌ 否	实现简单，效率高；**同一进程的两个线程不能同时陷入内核系统调用**
一对一（One-to-One）	1用户线程 ↔ 1内核线程	✅ 是	支持并发，系统开销大
多对多（Many-to-Many）	多用户线程 ↔ 多内核线程	✅ 是	平衡性能与并发性，复杂度较高

操作系统在执行系统调用时**可以被中断。系统调用运行在内核态，但内核态是可以被中断的。**

`fork()` 后**不共享地址空间**（虚拟地址空间独立），**不共享全局变量（不可通过全局变量实现数据传递）**、初始时共享物理页（写入会被断开因为 COW）、**不可通过全局变量通信**（需使用管道、共享内存、socket 等机制）。

两个不同进程对应的页表中**可能包含内容相同的页表项。**

IPC（进程间通信）重要概念梳理：

IPC 方式	特点	优点	缺点/限制
信号（Signal）	简单事件通知，携带信息少	快速、低开销	信息量小，处理不灵活
管道（Pipe）	字节流通信，常用于父子进程，**管道机制中的数据只存在于内存中。**	简单易用	单向通信、仅限亲缘进程
消息队列	内核维护消息结构，可存入多条消息，**比信号的信息承载量要大**	支持结构化通信、非亲缘进程	内核开销大，效率低于共享内存
共享内存	多进程共享同一物理页框	**效率最高、最快**	需要显式同步，**容易数据冲突**
套接字（Socket）	**不仅可用于不同机器之间的进程通讯，也可用于本机的两进程通讯**	灵活强大，支持多平台	通信开销大于共享内存、编程复杂

共享内存在**安全性上不如**消息传递。**管道机制中的数据只存在于内存中。**

最少资源数公式：

- **确认有几个进程（哲学家/章鱼/线程）N**；
- **确认每个进程最大资源需求为 K**；
- **套公式：最小资源数 = N × (K − 1) + 1**

周转时间 = 完成时间 - 到达时间。

最高响应比优先：**响应比 = 1 + 已等待时间 / 要求运行时间。**

凡是进程调度的题（算平均周转时间），对每个调度算法列一个这样的表：

序号	到达时间	开始时间	完成时间	周转时间

读者写者问题（根据题目意思看要不要while(1)）：

```
Semaphore rw = 1;       // 读写互斥访问
Semaphore mutex = 1;    // 用于读进程中对 count 的判断和赋值一气呵成
Semaphore w = 1;        // 用于读写公平（看题目意思要不要）
int count = 0;          // 当前读进程访问个数

writer() {
    P(w);               // 读写公平
    P(rw);
    write();
    V(rw);
    V(w);               // 这里可以在写之后释放读写锁
}
```

```
13
14  reader() {
15      P(w);                   // 读写公平
16      P(mutex);               // 读进程中对 count 的判断和赋值一气呵成
17      if (count == 0) {
18          P(rw);
19      }
20      count++;
21      V(mutex);
22
23      V(w);                   // 一定注意！！在开始读之前释放读写锁！否则无法并发读取
24      read();
25
26      P(mutex);               // 读进程中对 count 的判断和赋值一气呵成
27      count--;
28      if (count == 0) {
29          V(rw);
30      }
31      V(mutex);
32  }
```

进程执行 P 操作**阻塞**时，不会占用 CPU 资源。**信号量操作是原子操作。**

自旋锁（忙等）和互斥锁的概念：

特性	自旋锁（Spin Lock）	互斥锁（Mutex）
等待方式	自旋等待（不停查）	睡眠等待（阻塞调度）
CPU消耗	高（持续占用）	低（让出CPU）
上下文切换	无（高效）	有（慢）
适用场景	临界区短，频繁访问	临界区长，资源紧张

上下文切换两类：

- **自愿**上下文切换：**进程主动进入等待状态**，例如因为：请求**阻塞** I/O（如读取磁盘、网络）；等待互斥锁；调用 `sleep()`；然后调度程序将该进程换出。
- **非自愿**上下文切换（**抢占式**切换）：进程正在运行时，**被调度器强制换出。**原因包括：时间片耗尽；被更高优先级进程抢占；中断处理等。进程**未主动放弃 CPU。**也就是原因完全是由操作系统造成的。

任一时刻，**管程中最多只能有 1 个活跃进程。**

IO管理

三种 IO 控制方式对比（这三种方式的名称要熟练背诵）：

特性 / 方式	中断方式	DMA（直接存储器访问）	通道控制方式（Channel I/O）
数据传输方式	CPU 逐字节或逐块处理中断响应	DMA 控制器自动传输数据	通道处理器成批处理数据

特性 / 方式	中断方式	DMA（直接存储器访问）	通道控制方式（Channel I/O）
CPU 参与程度	频繁中断，CPU 每次都要处理数据传输	CPU 只负责发起传输，**传输过程不干预**	CPU 仅初始化和处理完成中断
是否支持并发传输	否，只能串行处理	有限支持（通常 1 个 DMA 控制器）	支持多个设备并发传输（多通道）
传输效率	低（频繁切换上下文）	中等（批量传输效率高）	高（通道并行处理，可流水）
硬件复杂度	最低	中等	最高（需独立的通道控制器）
典型应用场景	小数据量设备，如键盘、鼠标等	大数据量外设，如磁盘、网络设备	大型主机系统，批处理、带库、打印阵列等

因此，说 "I/O通道控制方式不需要任何CPU干预" 是**错误**的！

以下说法是**正确**的：中断方式的数据传送是在中断处理时由 CPU 控制完成的；而在 DMA 方式下，**数据传送过程不经过 CPU，是在 DMA 控制器的控制下完成的。**

由专门的控制器完成数据在内存和设备间传输工作的I/O控制方式称为 **DMA。**

设备控制器是一块能控制一台或多台外围设备与 CPU 并行工作的硬件。

操作系统中用户使用 I/O 设备时，通常使用的是**逻辑设备名。**

内存映射的 I/O 设备**不能**被用户级线程访问。

I/O 软件四层结构及其典型功能对照表：

层次	主要职责 / 功能	典型功能举例
用户层	发出 I/O 请求，使用系统调用接口	read() / write() / open() / close()
设备无关软件层	提供设备独立的 I/O 功能，统一接口封装	✅ 假脱机（Spooling） ✅ 缓冲机制（Buffering） ✅ 错误报告 ✅ 设备命名 ✅ 统一接口封装（如字符设备 vs 块设备）
驱动层	控制具体硬件设备，翻译为设备控制命令	向磁盘发送读写命令、处理中断、执行命令排队
中断处理/硬件控制层	执行实际的 I/O 操作，与控制器交互，或由硬件完成地址转换等底层操作	处理中断信号、**计算磁道/扇区/磁头**、DMA 控制、LBA 转换

磁盘管理

FCFS（先来先服务） 是**唯一一个同时适用于进程调度和磁盘调度**的算法。

磁盘平均访问时间 = 寻道时间 + 旋转延迟时间 + 传输时间。

- 寻道时间有时候题目会给，具体计算方式为：启动磁盘的时间 s 与磁头移动 n 条磁道所花费的时间之和。m × n + s, 其中 m 是一个常数（每移动一条磁道需要花费的时间）。

- 旋转延迟时间：1/(2r)。r 为转速。
- 传输时间：每次所读/写的字节数b，旋转速度r以及磁道上的字节数N关系：b/(rN)

访问磁盘次数 = **路径解析访问的目录块数 + 文件数据块数 +（必要时）文件控制块/索引块访问。**

磁盘驱动器上只有一个 **MBR**（主引导记录），但可能有多个引导扇区。

在一个磁盘上设置多个分区**不能**改善磁盘设备 I/O 性能。想改善磁盘设备 I/O 性能的方法：重排 IO 请求次序、多块磁盘并行、提高转速、增大缓存、**使用 SSD 替代机械硬盘。**

RAID（独立磁盘冗余阵列，Redundant Array of Independent Disks）：

RAID等级	特点	冗余	容错	最少磁盘数	容量利用率	读性能	写性能
RAID 0	条带化（Striping）	❌ 无	❌ 无	2	100%	✅ 高	✅ 高
RAID 1	镜像（Mirroring）	✅ 有	✅ 1块	2	50%	✅ 高	❌ 低
RAID 5	条带化 + 奇偶校验（分布式）	✅ 有	✅ 1块	3	(n-1)/n	✅ 高	⚠️ 较低（写校验）
RAID 6	RAID5 + 双重奇偶校验	✅ 有	✅ 2块	4	(n-2)/n	✅ 高	⚠️ 更低
RAID 10（1+0）	镜像+条带化	✅ 有	✅ 多块	4	50%	✅ 高	✅ 高

文件系统

树型目录结构**能够**解决文件重名问题。

文件类型按数据组织方式分类：

文件类型	特点描述
流式文件（Stream File）	字节序列构成，无结构，操作系统不理解其内容
记录式文件（Record File）	数据以记录（record）为单位，结构化，例如数据库文件、某些系统日志等
索引文件 / 目录文件等	通常具有特殊结构供系统识别管理

文件系统中的源程序文件是无结构的流式文件。（如 `.c`, `.java`, `.py` 等）

read 系统调用的参数**不包含文件的名称**！！通常包含文件描述符 `Fd`。

填空题：按**数据组织方式**分类，设备可以分为两类：**字符设备** 和 **块设备。**

文件三种**物理结构**的比较：

- **连续**文件：优点是不需要额外的空间开销，只要在文件目录中指出文件的大小和首块的块号即可，对**顺序**访问效率很高。适应于**顺序存取**。缺点是**动态地增长和缩小系统开销很大**；文件创建时要求用户提供文件的大小；存储空间浪费较大。
- **串联**文件：克服了连续文件的不足之处，但**文件的随机访问系统开销较大**。**适应于顺序访问的文件。**
- **索引**文件：**既适应于顺序存访问，也适应于随机访问**，是一种比较好的文件物理结构，但要有用于索引表的空间开销和文件索引的时间开销。

补充知识（期中）

请分别解释调用一次P(s)和V(s)操作后，s.count和s.queue会产生什么样的变化。

```
P(s):
    s.count--;
    if (s.count < 0) {                  // 注意是小于！！
        将当前进程加入 s.queue;
        阻塞当前进程;
    }

V(s):
    s.count++;
    if (s.count <= 0) {                 // 注意是小于等于！！！
        从 s.queue 中唤醒一个进程;
    }
```

用test-and-set实现 PV操作：

```
P(lock) {
    while(test_and_set(lock) == 1);
}

critical section;       // 进入临界区

V(lock) {
    lock = 0;
}
```

用 swap 实现 PV 操作：

```
bool use = 0;

void P(use) {
    bool k = 1;
    while (k != 0) {
        swap(&use, &k);
    }
}

// 临界区

void V(use) {
    use = 0;
```

```
14 }
```

并行性的确定 - Bernstein条件：两个进程可并发，当且仅当两个线程不会一个写一个读、或2个同时写。

R(S1) ∩ W(S2) = Φ W(S1) ∩ R(S2) = Φ W(S1) ∩ W(S2) = Φ