操作系统期末复习 2023.2

1. 同步（Synchronization)

Process (threads) synchronization: 进程(线程)同步指的是多个进程(线程)在某个点上连接或握手，以达成协议或承诺某个动作序列。

Data synchronization: 数据同步指的是让数据集的多个副本彼此保持一致性，或者保持数据完整性的思想。

Critical Section (Region) 临界区：访问共享资源的代码段。

Requirements to avoid race conditions 避免竞态条件的要求:

1. 不能有两个进程同时在它们的关键区域内。(安全)
2. 对于速度或 cpu 的数量，不能做任何假设。
3. 任何运行在其临界区域之外的进程都不能阻塞其他进程。(进展)
4. 任何进程都不应该永远等待进入临界区域。(活性)

Mutual Exclusion with Busy Waiting（忙等待互斥）：

- **Disabling Interrupts 禁用中断：** 最简单的解决方案是拥有每个进程在进入关键区域后禁用所有中断，并在离开前重新启用它们。（进程可能永远不允许中断）
- **Lock Variables 锁变量：** 当进程处于临界区时，它被设置为(比如说)1，当进程退出临界区时，它被重置为 0。（锁变量仍然是共享变量）
- **Strict Alternation 严格交替：** 可能违反了第三条规则。

Peterson's Solution：

```
int No_Of_Processes; // Number of processes
int turn; // Whose turn is it?
int interested[No_Of_Processes]; // All values initially FALSE
void enter_region(int process) {
    int other; // number of the other process
    other = 1 - process; // the opposite process
    interested[process] = TRUE; // this process is interested
    turn = process; // set flag
    while(turn == process && interested[other] == TRUE); // wait
}
void leave_region(int process) {
    interested[process] = FALSE; // process leaves critical region
}
```

Condition Variables 条件变量：一个同步对象，让线程有效地等待一个由锁保护的共享状态的变化。

- Wait(Condition Variable *cv, Lock *lock): atomically release lock (unlock) and suspends execution of the calling thread(sleep); Reacquire the lock when wakened.

- Signal(Condition Variable *cv): wake up a waiter (which is waiting for this condition variable). If no threads are on the waiting list, signal has no effect.
- Broadcast(Condition Variable *cv): wake up all waiters, if any.
- 等待:原子释放锁(unlock)并暂停执行调用线程(sleep),当唤醒时重新获取锁
- 单个:唤醒一个等待(等待这个条件变量)。如果等待列表中没有线程，则 signal 没有作用。
- 广播(条件变量*cv):唤醒所有等待者，如果有的话。

```
// Condition variable -> wait
int cnd_wait(cnd_t *cv, mtx_t *mtx)
{
    int val = atomic_load(&cv->value);
    mtx_unlock(mtx);
    futex_wait(&cv->value, val);
    mtx_lock(mtx);
    return thrd_success;
}

// Condition variable -> signal
int cnd_signal(cnd_t *cv)
{
    atomic_fetch_add(&cv->value, 1);
    futex_wake(&cv->value);
    return thrd_success;
}
```

条件变量规则:

- 当调用 wait, signal, broadcast 时 ALWAYS 保持锁;
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- 条件变量是无记忆的;
  - If signal when no one is waiting, no operation
  - If wait before signal, waiter wakes up
- 等待自动释放锁;

Semaphores 信号量： 一个整数变量来记录保存了多少次唤醒。Dijkstra 算法称这个变量为信号量。如果它等于零，就表示没有唤醒信号被保存。正值表示有一个或多个唤醒等待。

- P(), (or down() or wait()); P for Proberen (荷兰语，检验)
- V() (or up() or signal()); V for Verhogen (荷兰语，自增)

---

2. 死锁（Deadlock)

Deadlock: 一组进程死锁，如果集合中的每个进程都在等待一个事件，该事件只能由另一个进程引起。

死锁的四个必要条件：

- **Mutual exclusion 互斥**：同一时间只能有一个进程使用同一资源。
- **Hold and wait 持有等待**：一个持有至少一个资源的进程正在等待获得其他进程持有的额外资源。
- **No preemption 不可抢占**：资源只能由持有它的进程自愿释放，在该进程完成它的任务之后。
- **Circular wait condition 环路等待**：存在一个集合{P0，P1，…，pn}，其中 n > 1，P0 正在等待一个(或多个)由 P1 持有的资源，P1 正在等待一个(或更多)由 P2 拥有的资源，…，pn–1 正在等待一个(或多个)由 Pn 持有的资源，和 Pn 正在等待一个(或更多)由 P0 拥有的资源。

死锁检测（Deadlock Detection）

- 每种类型一个资源 → 可以利用**环路检测算法(O(V+E))** 进行死锁检测
- 每种类型多个资源 → 环路检测算法失效，需要利用基于资源分配矩阵、请求矩阵、可用向量的算法进行检测
    - **现有资源向量 E**：表示每种类型现有资源总数；
    - **可用资源向量 A**：表示每种类型可有资源数量；
    - **资源分配矩阵 C**：当前分配给第i个进程的第j种资源数量；
    - **请求矩阵 R**：表示第i个进程当前请求第j种资源的数量。
    - **∑i(Cij) + Aj = Ej** 某种资源总量等于可用资源加已分配资源。

死锁检测算法:

1. 判断某进程Pi的请求资源向量小于等于可用资源向量（即每种资源都小于等于现有资源）
2. 将该进程的已分配资源加到可用资源向量中，并标记该进程。之后重复步骤一。
3. 没有符合条件的进程时终止。

死锁避免（Deadlock Avoidance）

- **安全状态（safe）、非安全状态（Unsafe）**
    - **最大请求矩阵 M**：表示第i个进程最多请求第j种资源的数量。
    - 如果存在某种调度顺序，即使所有进程突然立即请求其最大数量的资源，每个进程都可以运行到完成，则称该状态是安全的。
    - 不安全不等于死锁！死锁是请求资源R大于可用资源A，不安全是已分配资源C加可以用资源A小于最大资源M。
- **银行家算法（Banker's algorithm）**
    - **剩余需要矩阵 N**：等于最大资源数量 M 减去已分配资源数量 C 。
    - 先对当前请求矩阵 R 做死锁判断，没有死锁再对剩余需要矩阵 N 做死锁判断，若此时没有死锁则是安全的。

RCU(read-copy-update)： 是 Linux 中比较重要的一种同步机制。顾名思义就是 **"读，拷贝，更新"** ，再直白点是"随意读，但更新数据的时候，需要先复制一份副本，在副本上完成修改，再一次性地替换旧数据"。这是 Linux 内核实现的一种针对"读多写少"的共享数据的同步机制。

---

3. 内存管理

内存管理的功能

- 内存管理：当有多个进程在使用内存和资源时，处理主存中所有与内存相关的操作和资源。
- 跟踪哪些部分的内存在被使用；
- 给进程分配并回收内存；
- 保护正常运行需要的内存；
- 扩展物理内存的限制；
- 总结：最大化内存利用率和系统吞吐量

虚拟地址空间和物理地址空间

- 地址空间提供了一个易于使用的物理内存抽象(虚拟化内存)
- 虚拟地址空间:被进程看到的部分
- 物理地址空间:由内存单元看到部分
- 虚拟地址与物理地址的映射：⟨**pid, virtual address**⟩

连续内存分配

- 内部碎片和外部碎片:
    - **外部碎片(External Fragmentation, holes):** 分区之间的内存浪费，由分散的不连续的空闲空间造成
    - **内部碎片化(Internal Fragmentation):** 分区内的内存浪费，由分区大小与加载进程之间的差异引起
- 空闲空间管理
    - **位图（bitmaps）:** 内存划分为分配单位(几个字~千字节),每个分配单元对应位图中的一个位，如果该单元空闲则为 0，如果被占用则为 1。
    - **链表（Linked list）:** 维护一个已分配和空闲内存分区的链表，其中一个分区要么包含一个进程，要么是两个进程之间的空洞。
- 动态空间分配算法:
    - **First-Fit:** 分配第一个足够大的洞
    - **Next-Fit:** 跟踪上一个合适的洞，并在上次停止的地方开始 first-fit 搜索
    - **Best-Fit:** 分配足够大的最小孔
    - **Worst-Fit:** 分配最大的洞
    - **Quick-Fit:** 为一些比较常见的尺码单独列出清单

分页

- 地址转换
    - 逻辑地址包括两部分：⟨**segment number, offset**⟩
    - 逻辑地址由**段号加偏移量**构成。在段表中每个表项包括**段基址(base)和段大小(size)**.
    - 由于段的长度不同，内存分配是一个动态内存分配问题。
    - 内存中的同一个物理段可以映射到多个虚拟地址空间。
- 页表和页表项
    - 以固定大小的单位管理内存:页。对虚拟内存称为**页**，对物理内存称为**帧**。
    - 逻辑地址分为两部分：⟨**page number, offset**⟩

    - 逻辑地址由 **页面号和偏移量** 构成，一个 **2^m** 的虚拟空间地址: **2^(m-n)** 页, 页大小 **2^n**.
    - 页表:每个 **页表条目(PTE)** 包含一个帧号(PFN)，表示物理内存中每个页的基址，还有呈现位P，保护位（R/W），脏位（D）等。

- 页面共享：两个页表中的条目指向相同的页帧(具有一定的保护位)。

- **写时复制 (Copy on Write)**

    - 尽可能分享，只做"按需"的副本；
    - fork()通过创建父进程的完整副本来创建一个新进程
    - zero-fill-on-demand technique：分配前已归零

- **转换检测缓冲区/快表 (TLB)** 和 **TLB Miss**

    - Translation Look-aside Buffers (TLB)：如果 TLB 命中，转换地址。
    - 否则，TLB miss，然后在页表中查找映射(页表遍历)，并更新TLB。
    - 硬件管理的 TLB(CISC，像 x86)
    - 软件管理 TLB(RISC，如 MIPS)

- **有效访问时间**

    - Effective Access Time **(EAT) = (e + t) a + (e + 2t)(1 – a)**
    - Hit Ratio (a): percentage of times that a page number is found in the TLB.
    - Memory access time (t).
    - TLB search time (e).

- 多级页表

    - 两级地址转换：虚拟地址 = 页目录索引 p1 + 页表索引 p2 + 偏移地址 d .
    - **1B = 1 Byte = 8 Bit, 2^10 B = 1KB, 2^20 B = 1 MB**.
    - 假设一个 1GB(2^30) 的虚拟地址空间，512 字节(2^9) 的页面大小，每个 PTE 4 字节，则 PTE 总数为 **2^21** 个。若每页包含 2^7 = 128 个 PTE ，对于两级分页，页面目录有 **2^14 = 16K PTE(64KB 大小，跨度 128 页)**.

- 页面大小

    - 页表大小——大页
    - 内部碎片化——小页(避免浪费)
    - TLB 空间和页面故障——大页
    - 参考位置——小页面(更好的分辨率，更少的 I/O)
    - I/O 传输时间——大页

虚拟内存

- **缺页错误 (Page Fault)** 及其处理流程

    - 硬件捕获到内核，节省堆栈上的程序计数器
    - 汇编代码例程开始保存一般寄存器和其他易失性信息
    - 系统发现发生了页面错误，试图发现需要哪个虚拟页面
    - 一旦知道虚拟地址引起的故障，系统会检查地址是否有效，保护是否与访问一致

    - 找一个空闲帧：若没有空闲帧，运行页面替换选择一个替换者；若帧是脏帧，页面被计划转移到磁盘，发生上下文切换，暂停故障进程
    - 帧清理后，系统立即查找需要页面的磁盘地址，并安排磁盘操作将其引入(故障进程仍然挂起)
    - 当磁盘中断表示页面已经到达时，更新页表，并将帧标记为处于正常状态
    - 故障指令备份到它所拥有的状态，程序计数器复位
    - 故障进程被安排，操作系统返回到调用它的例程
    - 例程重新加载寄存器和其他状态信息，返回到用户空间继续执行

- **页面故障率**

    - 进程在内存中发现页面故障的概率
    - 有效访问时间 **EAT = （1-p) * 内存访问时间 + p * 页面故障开销**

- **页面置换算法:**

    - **FIFO 先进先出：** 替换最旧的页面，使用 FIFO 队列来跟踪页面的年龄
    - **Optimal 最优：** 替换那些最长时间不会被使用的页面
    - **Least Recently Used (LRU) 最近最少使用：** 替换掉在最长时间内没有被使用的页面
    - **Approximating LRU 近似LRU:**
        - reference bit **（Second Chane）**：设置一个引用位（R），寻找在最近的时钟间隔内没有被引用的旧页面
        - modified bit **(Not Recently Used, NRU)**: 设置修改位，为那些修改过的页面分配更高的优先级，以减少 I/O (增强第二次机会)
        - **Aging algorithm:** 保留一个软件计数器来跟踪每个页面被引用的频率，并用最小的计数替换页面

工作集模型

- **抖动 (Thrashing)**
    - 如果一个进程没有"足够"的页面，那么页面错误率就会非常高
    - 抖动:一个进程花费所有的时间来交换页面的进出(大多数引用都会导致页面错误)
- **访问局部性 (Principle of Locality)**
    - 时间局部性(Temporal locality):在不久的将来会再次访问相同的内存位置
    - 空间局部性(Spatial locality):附近的内存位置将在未来被访问
- 工作集：进程在最近的 k 次内存引用中使用的一组页面
- 工作集算法：当页面故障发生时，找到不在工作集中的页面并将其驱逐

页帧的全局和局部分配策略

- 本地页面替换:每个进程从自己分配的帧集中选择受害者
- 全局页面替换:从分配给任何进程的帧中选择受害者
    - OS 必须不断决定分配给每个进程的页帧数
    - 平等分配:为每个进程分配平等的份额
    - 比例分配:根据进程的大小进行分配(需要给每个流程一定的最小数量)
- 页面故障频率(Page-Fault Frequency, PFF)算法来管理帧分配

---

4. 文件系统

文件系统的功能： 为用户提供存储界面，将逻辑块映射到物理；通过方便地存储、定位和检索数据，高效地访问磁盘

- **Persistent Named Data**:持久的保存数据
- **Access and Protection**：提供不同用户的访问和保护
- **Disk Management**：公平高效的使用磁盘空间
- **Reliability**：不会丢失文件数据

文件： OS 创建的用于存储信息的逻辑存储单元

- 文件命名：文件系统抽象的一个重要特征
  - 文件扩展名：提示应用程序或 OS 以合理的方式操作文件。
- 文件结构：
  - 字节数组(无结构)
  - 固定长度的记录序列(具有一些内部结构)
  - 记录树(按关键字段排序)
- 文件属性(元数据)：文件位置，文件大小，文件时间，文件拥有者，文件保护，文件控制等
- **元数据：** 文件系统应该将文件元数据保存在一个结构中: **索引 index inode**
- 顺序和随机访问模式
  - 顺序访问:按顺序读取或写入数据
  - 随机(直接)访问:随机地址任何块
- 文件相关的操作：创建，删除，打开，关闭，Read，Write，Append，Seek（指定从哪里获取数据），读取和更改文件属性，重命名等。
- **文件描述符 (File Descriptor, FD)：** OS 分配给文件的唯一数字(每个进程私有)，使之具有执行某些操作的能力
- **打开文件表 (Open-File Table)：** 每个进程维护一个打开文件表，一个由文件描述符索引的数组，每个条目跟踪文件描述符引用的底层文件，当前偏移量和其他相关细节如文件权限等。
  - 单进程打开文件表
  - 系统范围打开文件表
  - 每个进程的打开文件表条目，指向现有的系统范围的打开文件表
- **偏移 (Offset)：** 对于进程打开的每个文件，OS 跟踪一个文件偏移量，该偏移 量决定了下一次读取或写入将从哪里开始
  - 隐式更新
  - 显式更新：系统调用 lseek( )

目录(Directory)

- 目录的层级结构
  - 文件系统通常有目录来跟踪文件
  - 通过在其他目录中放置目录，用户能够构建任意目录树(或目录层次结构)
  - 绝对路径和相对路径
- 从文件路径名到文件 inode 的转换
  - 目录是一张由 ⟨**file name, file index**⟩ **（文件名，文件索引）** 对构成的表。
  - 要查找文件，请找到包含到文件索引的映射的目录。
  - 为根目录指定一个固定的 **inode** 号。
- 硬链接 (Hard Link) 和符号链接 (Symbolic Link)
  - **Hard Link:** 在目录中创建另一个名称，并引用与原文件相同的 inode 号（相当于复制）

    - **Symbolic (Soft) Link:** 创建一个不同类型的文件(链接类型)，符号链接是路径名的别名(索引节点号的硬链接)，相当于快捷方式。

文件系统的布局

- **数据区域(Data Region):** 大多数块是数据块，并为这些块保留磁盘的固定部分
- **索引表(inode Table):** 每个文件的信息(元数据)
- **分配结构:** 关于已分配空间和空闲空间的信息(空闲空间跟踪)
- **超级块(Superblock):** 关于文件系统的信息

文件系统的实现

- **Contiguous 连续分配：** 每个文件占用一组连续的块
    - 只需要第一个块的磁盘地址和块的数量确定文件
    - 优点：可以高效地实现顺序访问也可以快速计算随机地址的数据位置
    - 缺点：创建时需要指导文件大小；产生外部碎片
- **Linked List：** 每个文件是一个区块链表，每个区块包含指向下一个区块的指针
    - 优点：目录条目仅存储第一个块的磁盘地址
    - 缺点：无法随即定位文件；额外指针空间
- **File Allocation Table 文件分配表 FAT：** 链接分配的一种变体(用于 MS-DOS)
    - 将所有指针放在一个表中
    - 优点：目录只要保存起始块号；可以随机访问
    - 缺点：FAT表一直放在内存中存储消耗大
- **Indexed (inode 的多级索引设计)**
    - 每个文件都有一个指向块的指针数组(索引块)
    - 间接指针: 它不是指向包含用户数据的块，而是指向包含更多指针的块，每个指针都指向用户数据
    - 保证了对小文件（2K）的高效索引

目录的实现

- 目录项中包含的信息
    - ⟨**file name, file index**⟩ 文件名-文件索引映射
    - **file attributes** or **inode** 文件属性或者只有inode
- 不同长度文件名的处理
    - 设置文件名长度限制，固定条目长度
    - 设置不同长度条目，其中属性长度固定，文件名以特殊字符表示结束
    - 文件名单独放在一个堆中，每个条目指向对应名称

空闲空间管理

- **位图 BitMap：** 每个块使用一个位，跟踪其分配状态
    - 容易找到连续空闲空间序列
    - 需要额外空间
- **空闲列表 Free List：** 在链表中保留空闲块
    - 不浪费空间，只使用空闲块中的内存作为指针
    - 分配空间时可能需要遍历链表

文件系统的性能

- **缓存 (Cache and Buffering)**

    - 读缓冲：
        - 将经常使用的块保留在内存中:如果需要的块在内存中，不需要 I/O 就可以读取它
        - 预读预取:尝试在需要块之前将它们放入缓存，以提高命中率
    - 写缓冲：
        - **回写策略：** 当修改一个块时，将其标记为脏，稍后再写入磁盘
        - **直写策略：** 每当修改缓存时，将缓存块写入磁盘
- **快速文件系统 (Fast File System, FFS)**
    - 磁盘感知的文件系统
    - 实现数据局部性: 把相关的东西放在一起(把不相关的东西分开很远)

文件系统的一致性

- **文件系统一致性检查 (File System Consistency Check, FSCK):** 在文件系统挂载之前运行(假设在运行时没有文件系统活动正在进行)，并确保文件系统元数据在内部是一致的，包括：
    - 检查**超级块**（比较文件系统大小是否等于分配的总大小）
    - 检查**空闲块和位图有效性**（inode 和 bitmap）
    - 检查**索引节点**是否损坏
    - 检查 **inode 链路**
    - 检查**重复的指针**和**坏的块**
    - 检查**目录**
- **日志文件系统 (Journaling)**
    - 基本思想:预写日志(来自数据库系统)
    - 将整个磁盘构造为一个大日志(就像磁带)
    - 当写入磁盘时，**日志文件系统 LFS(Log-structed File System)** 首先将所有更新缓冲到一个内存段中(延迟写入操作)
    - 当段已满时，它将被写入磁盘，在一个长时间和顺序的传输到磁盘

虚拟文件系统

- **虚拟文件系统(VFS)** 提供了一种面向对象的实现文件系统的方法
- 抽象出文件系统中对所有文件系统通用的部分，并将该代码放在单独的层中
- 为文件系统定义了一个通用接口，所有文件系统都需要实现它们

---

5. 输入输出

I/O 设备的交互

- **Polling 轮询**
    - 操作系统反复读取状态寄存器，等待设备准备好接受命令；
    - 操作系统将数据发送到数据寄存器中；
    - 操作系统向命令寄存器中写入命令；
    - 操作系统等待设备完成，再次进行轮询。
- **Interrupts 中断**
    - 操作系统发出请求，将调用进程进入睡眠状态，上下文切换到另一个任务上；
    - 当设备完成操作后，将引发一个硬件中断；
    - 中断使CPU进入操作系统中的中断处理程序中；
    - 处理程序完成请求并唤醒等待IO的进程。

- **DMA 直接内存访问**
  - DMA使系统中一个特定设备，可以协调设备和主存之间的传输，不需要CPU的干预。
  - 操作系统对DMA编程，提供数据位置，大小及发送的设备；
  - DMA完成后，DMA控制器会引发一个中断。

磁盘

- 寻道时间和旋转延迟
  - **寻道时间(Seek Time)：** 将盘臂移动到正确的轨道上所用时间；
  - **旋转延迟(Rotational delay)：** 等待所需扇区在磁盘磁头下旋转的时间。
  - 传输时间(Transfer)：读写数据的时间
- 磁盘调度算法:
  - **Shortest Seek First (SSF) 最短寻道时间优先：** 按照轨道对IO请求序列进行排序，选取离当前轨道最近的请求
  - **First Come First Service (FCFS) 先来先服务：** 按照顺序完成请求
  - **Elevator (SCAN) 电梯：** 在磁道上来回扫描
- RAID：使用多个磁盘拼成一个更快更大更可靠的磁盘系统

E.N.D.