操作系统内容整理_第一版

一、操作系统引论

1.1 操作系统的定义与目标

定义：操作系统（Operating System, OS）是一组能有效地组织和管理计算机硬件和软件资源，合理地对各类作业进行调度，以及方便用户使用的程序的集合。

核心角色：

- **资源管理者**：统一管理 CPU、内存、I/O 设备、文件等软硬件资源
- **用户与硬件的接口**：屏蔽底层硬件细节，提供便捷的使用方式

操作系统的设计目标：

目标	说明
方便性	提供友好接口（CLI/GUI），使用户无需了解硬件细节
有效性	提高系统资源利用率和吞吐量
可扩充性	便于添加新功能、新设备，适应技术发展
开放性	遵循国际标准，实现软硬件兼容

1.2 操作系统的四大核心功能

功能模块	别名	主要职责	关键技术
处理器管理	进程管理	负责 CPU 的分配、调度与协调；管理进程/线程状态；实现多任务并发	进程控制、进程同步、进程通信、调度算法
存储器管理	内存管理	内存分配与回收；地址映射；内存保护；内存扩充（虚拟存储）	连续/离散分配、分页/分段、页表机制、置换算法
I/O 设备管理	设备管理	管理各类输入/输出设备；提供设备驱动、缓冲机制、设备分配策略	缓冲技术、SPOOLing、设备独立性、磁盘调度
文件管理	信息管理	负责文件的存储、检索、命名、共享、保护与备份	文件逻辑/物理结构、目录管理、文件共享与保护

1.3 操作系统的五大特征

特征	说明	要点
并发（Concurrency）	宏观上多个事件"同时"发生，微观上单核 CPU 上交替执行	并发性是操作系统最基本的特征

特征	说明	要点
共享（Sharing）	互斥共享（如打印机，一次仅一进程使用）；同时共享（如磁盘，宏观同时微观交替）	资源复用
虚拟（Virtualization）	时分复用（虚拟处理器、虚拟设备）；空分复用（虚拟存储器）	逻辑资源大于物理资源
异步（Asynchronism）	进程以不可预知的速度推进，相同输入多次执行结果可能不同	需同步机制保证结果一致性
不确定性（Indeterminacy）	多道程序环境下，执行顺序不确定	需要调度与协调

1.4 操作系统的发展过程

```
无操作系统时代
        ↓
单道批处理系统（自动性、顺序性、单道性）
        ↓
多道批处理系统（多道性、无序性、调度性）← 引入"进程"概念
        ↓
分时系统（多路性、交互性、独占性、及时性）
        ↓
实时系统（硬实时：严格时限 / 软实时：相对时限）
        ↓
个人计算机操作系统 → 网络操作系统 → 分布式操作系统 → 嵌入式操作系统
```

1.5 操作系统的分类

类型	典型代表	核心特点	适用场景
单用户单任务	MS-DOS	一次仅一个用户运行一个程序	早期个人计算机
单用户多任务	Windows、macOS	单用户可同时运行多程序	现代个人计算机
多用户多任务	Linux、UNIX	多用户登录，各用户多任务并行	服务器、工作站
实时操作系统	VxWorks、μC/OS	严格响应时限，高可靠性	工业控制、航空航天
嵌入式操作系统	Android、HarmonyOS	资源受限，专用性强	移动设备、物联网
网络操作系统	Windows Server	网络通信、资源共享、安全管理	企业网络环境
分布式操作系统	研究阶段	多机协同、透明性、高可用性	云计算、大数据平台

1.6 操作系统的结构设计

结构类型	核心思想	优点	缺点	代表系统
无结构	过程集合，无明确模块划分	简单直接	逻辑混乱，难以维护	早期 OS
模块化结构	按功能划分独立模块，模块间调用	结构清晰，便于修改	接口复杂，模块依赖难管理	部分早期 UNIX
分层式结构	系统分层，低层为高层服务，单向调用	层次分明，易于调试	层次过多效率低，设计困难	THE 系统
微内核结构	仅核心功能（调度、通信）在内核，其他移至用户态	安全、稳定、可扩展、易移植	用户态/内核态切换频繁，性能开销大	Minix、QNX、macOS 部分组件
外核结构	内核仅负责资源保护与分配，应用直接管理硬件	极致性能，应用定制化	编程复杂，应用需处理底层细节	研究型系统

1.7 操作系统的运行环境

```
┌──────────────────────────────────┐
|          用户态（User Mode）        |
|  • 应用程序运行环境                  |
|  • 权限受限，不能直接访问硬件        |
|  • 通过系统调用请求内核服务          |
├──────────────────────────────────┤
|          内核态（Kernel Mode）      |
|  • 操作系统内核运行环境              |
|  • 拥有最高权限，可访问所有资源      |
|  • 处理中断、系统调用、异常          |
└──────────────────────────────────┘
```

系统调用（System Call）流程：

```
应用程序 → 陷入指令（trap）→ 内核态 → 执行服务 → 返回用户态
```

二、进程

2.1 进程的概念与特征

进程定义：进程是程序在一个数据集上的一次执行过程，是系统进行资源分配和调度的基本单位。

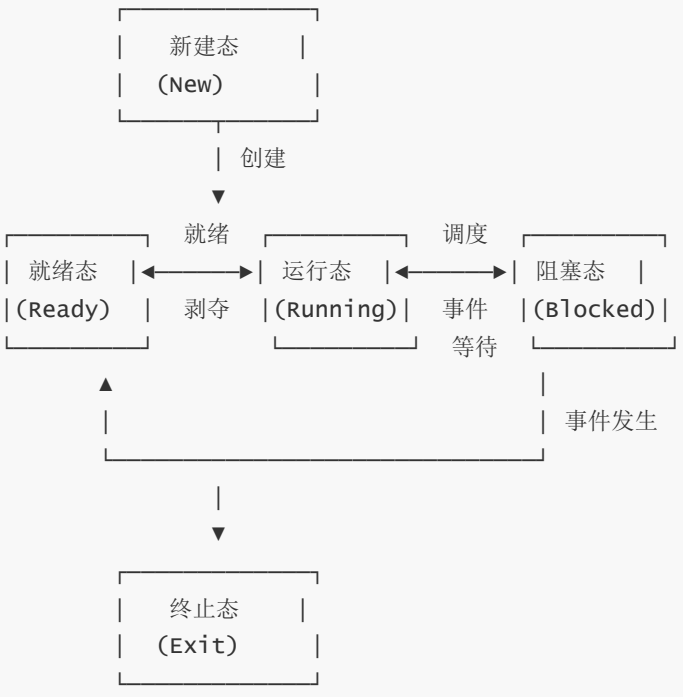
进程与程序的区别：

对比项	进程	程序
动态性	动态实体，有生命周期	静态指令集合
并发性	可与其他进程并发执行	不能独立运行
独立性	独立调度和资源分配单位	无独立性
异步性	以不可预知速度推进	无执行概念
结构性	PCB + 程序段 + 数据段	仅代码和数据

进程的组成:

```
进程控制块（PCB） ← 进程存在的唯一标志
    |
    ├── 进程标识信息（PID、父进程 ID）
    ├── 处理机状态信息（寄存器、PC、PSW）
    ├── 进程调度信息（状态、优先级、等待事件）
    ├── 进程控制信息（程序/数据地址、资源清单、同步通信机制）
    |
    ├── 程序段 ← 可重入的纯代码
    |
    └── 数据段 ← 全局变量、静态变量、动态数据
```

2.2 进程的状态与转换



状态转换详解:

转换	触发条件	系统操作
就绪→运行	调度程序选中	保存当前进程上下文，加载选中进程上下文

转换	触发条件	系统操作
运行→就绪	时间片用完/更高优先级进程就绪	保存运行进程上下文，插入就绪队列
运行→阻塞	请求 I/O、等待事件、申请资源失败	保存上下文，插入阻塞队列，调度其他进程
阻塞→就绪	等待事件完成、资源可用	从阻塞队列移至就绪队列
新建→就绪	进程创建完成，资源分配完毕	初始化 PCB，加入就绪队列
运行→终止	任务完成、异常终止、被杀死	回收资源，撤销 PCB

2.3 进程控制

进程控制原语（原子操作，不可中断）：

```
// 1. 进程创建（fork/CreateProcess）
OS_CreateProcess(PCB_new, parent_PCB) {
    申请空白 PCB;
    初始化进程标识、状态、优先级;
    分配内存空间、I/O 资源;
    初始化 PCB 其他信息;
    插入就绪队列;
}

// 2. 进程终止（exit/ExitProcess）
OS_TerminateProcess(PCB) {
    根据终止标识递归终止子进程;
    回收该进程所有资源;
    将 PCB 从所在队列移除;
}

// 3. 进程阻塞（block/sleep）
OS_BlockProcess(PCB, event) {
    停止执行，保存现场到 PCB;
    PCB 状态改为阻塞;
    插入对应事件的阻塞队列;
    调度其他进程;
}

// 4. 进程唤醒（wakeup）
OS_WakeupProcess(PCB) {
    PCB 状态改为就绪;
    从阻塞队列移至就绪队列;
    // 不立即执行，等待调度
}

// 5. 进程切换（context switch）
OS_SwitchProcess(old_PCB, new_PCB) {
```

```
    保存 old_PCB 的寄存器、PC、PSW;
    更新 old_PCB 状态;
    加载 new_PCB 的寄存器、PC、PSW;
    更新 new_PCB 状态为运行;
}
```

2.4 进程同步与互斥

基本概念:

- **临界资源**：一次仅允许一个进程使用的资源（如打印机、共享变量）
- **临界区**：进程中访问临界资源的代码段
- **同步**：协调进程执行顺序，保证合作进程正确协作
- **互斥**：保证同一时刻仅一个进程进入临界区

同步机制应遵循的原则:

原则	说明
空闲让进	无进程在临界区时，请求进入的进程应立即进入
忙则等待	有进程在临界区时，其他请求进程必须等待
有限等待	进程等待进入临界区的时间应有上限（避免饥饿）
让权等待	进程不能进入临界区时应释放处理器（避免忙等待）

2.5 信号量与 P、V 操作

信号量（Semaphore）定义:

```
typedef struct {
    int value;               // 资源数量或状态
    struct process *queue;   // 等待该信号量的进程队列
} semaphore;
```

P 操作（Wait/Down）：申请资源

```
void P(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        // 资源不足，当前进程阻塞
        block 当前进程();
        将当前进程加入 S->queue;
    }
}
```

V 操作（Signal/Up）：释放资源

```
void V(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        // 有进程在等待，唤醒一个
        wakeup(S->queue 中一个进程);
        将该进程从队列移除;
    }
}
```

信号量应用分类：

类型	初值	含义	应用场景
整型信号量	正整数	可用资源数量	资源计数、互斥
记录型信号量	1 或 0	资源数量 + 等待队列	通用同步互斥
AND 信号量	—	一次性分配多个资源	避免死锁
管程	—	高级同步机制，封装共享数据 + 操作	复杂同步场景

2.6 经典同步问题详解

（1）生产者 - 消费者问题（有界缓冲区）

问题描述：生产者生成产品放入缓冲区，消费者从缓冲区取出产品消费，缓冲区大小为 n。

信号量设置：

```
semaphore mutex = 1;     // 互斥访问缓冲区
semaphore empty = n;     // 空缓冲区数量
semaphore full = 0;      // 满缓冲区数量
```

完整代码：

```
// 生产者
void producer() {
    while (true) {
        生产一个产品;
        P(empty);         // 等待空缓冲区
        P(mutex);         // 进入临界区
        将产品放入缓冲区;
        V(mutex);         // 退出临界区
        V(full);          // 满缓冲区 +1
    }
}

// 消费者
void consumer() {
    while (true) {
        P(full);          // 等待满缓冲区
        P(mutex);         // 进入临界区
        从缓冲区取出产品;
```

```
        V(mutex);          // 退出临界区
        V(empty);          // 空缓冲区 +1
        消费产品;
    }
}
```

关键点：

- P 操作顺序：先资源信号量（empty/full），后互斥信号量（mutex），避免死锁
- 多个生产者/消费者时，mutex 保证缓冲区操作原子性

（2）哲学家进餐问题

问题描述：5 位哲学家围坐圆桌，思考或进餐，需同时拿起左右两支筷子才能进餐。

死锁原因：5 人同时拿起左筷子，均等待右筷子，形成循环等待。

解决方案对比：

```
// 方案1：限制同时拿筷子人数（最多4人）
semaphore mutex = 4;                // 限制人数
semaphore chopstick[5] = {1,1,1,1,1};

哲学家 i:
    while (true) {
        思考;
        P(mutex);                   // 申请拿筷子资格
        P(chopstick[i]);            // 拿左筷子
        P(chopstick[(i+1)%5]);      // 拿右筷子
        V(mutex);                   // 释放资格（可提前）
        进餐;
        V(chopstick[i]);
        V(chopstick[(i+1)%5]);
    }

// 方案2：奇偶哲学家拿筷子顺序不同
哲学家 i:
    while (true) {
        思考;
        if (i % 2 == 0) {           // 偶数号先左后右
            P(chopstick[i]);
            P(chopstick[(i+1)%5]);
        } else {                    // 奇数号先右后左
            P(chopstick[(i+1)%5]);
            P(chopstick[i]);
        }
        进餐;
        V(chopstick[i]);
        V(chopstick[(i+1)%5]);
    }

// 方案3：原子操作（同时拿两支筷子）
semaphore mutex = 1;                // 保护拿筷子动作
哲学家 i:
    while (true) {
```

```
        思考;
        P(mutex);                    // 原子地拿两支筷子
        P(chopstick[i]);
        P(chopstick[(i+1)%5]);
        V(mutex);
        进餐;
        V(chopstick[i]);
        V(chopstick[(i+1)%5]);
    }
```

（3）读者 - 写者问题

问题描述：多个读者可同时读，但写者需独占访问；读者与写者、写者与写者互斥。

读者优先方案（可能导致写者饥饿）：

```
semaphore mutex = 1;     // 保护 readcount
semaphore write = 1;     // 写锁
int readcount = 0;        // 当前读者数量

// 读者进程
void reader() {
    while (true) {
        P(mutex);
        if (readcount == 0) P(write);  // 第一个读者加写锁
        readcount++;
        V(mutex);

        执行读操作;

        P(mutex);
        readcount--;
        if (readcount == 0) V(write);  // 最后一个读者释放写锁
        V(mutex);
    }
}

// 写者进程
void writer() {
    while (true) {
        P(write);           // 申请写锁
        执行写操作;
        V(write);           // 释放写锁
    }
}
```

写者优先方案（增加写者优先信号量）：

```
semaphore mutex = 1, write = 1, wmutex = 1;
int readcount = 0, writecount = 0;

// 读者需先通过 wmutex 检查是否有写者等待
void reader() {
    P(wmutex);              // 检查写者
```

```
    P(mutex);
    if (readcount == 0) P(write);
    readcount++;
    V(mutex);
    V(wmutex);              // 释放写者检查

    读操作;

    P(mutex);
    readcount--;
    if (readcount == 0) V(write);
    V(mutex);
}

// 写者增加 writecount，优先获得写锁
void writer() {
    P(wmutex);
    writecount++;
    if (writecount == 1) P(write);  // 第一个写者阻塞读者
    V(wmutex);

    P(write);                       // 申请独占写
    写操作;
    V(write);

    P(wmutex);
    writecount--;
    if (writecount == 0) V(write);  // 最后一个写者释放
    V(wmutex);
}
```

2.7 进程通信（IPC）

通信方式	原理	优点	缺点	适用场景
共享存储器	进程映射同一物理内存区域	速度最快，零拷贝	需同步机制，编程复杂	高频数据交换
消息传递	通过 send/recv 原语交换消息	接口简单，天然同步	消息拷贝开销，缓冲区管理	分布式系统、微内核
管道通信	内核维护环形缓冲区，单向/双向	简单可靠，适合父子进程	容量有限，半双工	命令行管道、进程间流式数据
信号	异步通知机制，传递简单事件	轻量级，异步通知	信息量少，不可靠	进程控制（终止、暂停）
信号量	计数器 + 等待队列，同步互斥	灵活控制资源访问	需编程实现，易死锁	资源管理、同步协调
套接字	网络/本地进程间通信端点	跨主机、跨平台	协议开销，配置复杂	网络应用、C/S 架构

2.8 线程

线程定义：线程是进程中执行流的分支，是 **CPU 调度的最小单位**，同一进程的线程共享进程资源。

线程与进程对比：

对比项	进程	线程
资源拥有	系统资源分配的基本单位	仅拥有必要资源（栈、寄存器、TCB）
调度单位	传统系统的基本调度单位	现代系统的基本调度单位
并发性	进程间可并发	同一进程内线程可并发
开销	创建/切换/通信开销大	开销小，效率高
独立性	地址空间独立，故障隔离	共享地址空间，一线程崩溃影响全进程
通信	需 IPC 机制	直接读写共享变量（需同步）

线程实现方式：

实现方式	说明	优点	缺点
用户级线程（ULT）	线程管理在用户空间完成，内核无感知，一个进程对应一个内核线程	切换快，可定制调度策略	一线程阻塞，全进程阻塞
内核级线程（KLT）	线程管理由内核完成，每个线程对应一个内核调度实体	一线程阻塞不影响其他线程	切换需陷入内核，开销大
混合模型（多对多）	多个用户线程映射到多个内核线程，兼顾灵活性与并行性	兼顾灵活性与并行性	实现复杂

三、处理机调度

3.1 调度层次与类型

调度层次	别名	调度对象	触发频率	主要功能
高级调度	作业调度	作业（从外存）	低（分钟级）	从后备队列选作业调入内存，创建进程
中级调度	内存调度	挂起进程	中（秒级）	进程换入/换出内存，调节多道程序度
低级调度	进程调度	进程/线程	高（毫秒级）	从就绪队列选进程分配 CPU，最核心调度

三状态模型与调度关系：

```
外存（后备队列）
     | 高级调度（作业调度）
     ▼
内存（就绪队列）←→ 运行态 ←→ 阻塞态
     |                ▲
     | 低级调度       | 事件完成
     ▼                |
     运行态 ————————┘
     |
     | 中级调度（挂起/激活）
     ▼
外存（挂起队列）
```

3.2 调度算法详解

（1）先来先服务（FCFS）

原理：按进程到达就绪队列的先后顺序分配 CPU

特点：

- ✓ 优点：实现简单，公平
- ✗ 缺点：对短作业不利（护航效应），不利于 I/O 密集型进程

示例：

进程	到达时间	服务时间	完成时间	周转时间	等待时间
P1	0	24	24	24	0
P2	0	3	27	27	24
P3	0	3	30	30	27

> 平均等待时间 = (0+24+27)/3 = **17**

（2）短作业优先（SJF/SPF）

原理：优先调度估计运行时间最短的进程

变体：

- **非抢占式**：一旦开始执行，不被打断
- **抢占式（SRTF）**：新进程剩余时间 < 当前进程剩余时间则抢占

特点：

- ✓ 优点：平均等待时间最短（理论最优）
- ✗ 缺点：长作业可能饥饿，作业运行时间难以准确预估

示例（非抢占式）：

进程	到达时间	服务时间	完成时间	周转时间	等待时间
P1	0	6	16	16	10
P2	2	8	24	22	14
P3	0	7	7	7	0
P4	3	3	10	7	4

> 平均等待时间 = (10+14+0+4)/4 = **7**

（3）优先级调度（Priority）

原理：为每个进程分配优先级，优先调度高优先级进程

优先级类型：

- **静态优先级**：创建时确定，运行中不变
- **动态优先级**：随等待时间/执行时间动态调整

特点：

- ✓ 优点：紧急任务可优先处理
- ✗ 缺点：低优先级可能饥饿，优先级设置主观性强

改进：

- 优先级老化：等待时间越长，优先级越高
- 多级反馈队列：结合优先级与时间片

（4）时间片轮转（RR）

原理：为每个进程分配固定时间片，用完则剥夺 CPU，排到队尾

关键参数：时间片大小 q

- q 过大 → 退化为 FCFS，响应慢
- q 过小 → 频繁切换，系统开销大
- 经验值：10~100ms，切换开销占 1% 以内

特点：

- ✓ 优点：公平，响应时间有保障，适合分时系统、交互式应用
- ✗ 缺点：平均周转时间较长

示例（q=4）：

```
就绪队列：P1(6) → P2(8) → P3(7) → P4(3)
执行序列：P1(4)→P2(4)→P3(4)→P4(3)→P1(2)→P2(4)→P3(3)
```

（5）多级反馈队列（MLFQ）

设计思想：

1. 设置 n 个就绪队列，优先级 Q1 > Q2 > ... > Qn
2. 时间片：q1 < q2 < ... < qn（如 1,2,4,8...）
3. 新进程进入 Q1，未用完时间片则保持原队列
4. 用完时间片未完成，则降级到下一队列
5. 高优先级队列采用 RR，低优先级可用 FCFS

调度规则：

- 优先调度高优先级队列中的进程
- 同队列内按时间片轮转
- 进程降级后，仅当高优先级队列为空时才调度

特点：

- ✓ 优点：兼顾短作业优先与长作业公平，交互式进程获快速响应，计算密集型进程最终也能完成
- ✗ 缺点：参数配置复杂，需调优

3.3 调度算法对比与选型

算法	平均等待时间	响应时间	公平性	实现复杂度	适用场景
FCFS	长	差	一般	低	批处理系统
SJF	最短	一般	差	中	已知运行时间的批处理
优先级	取决于优先级	好（高优先级）	差	中	实时系统、紧急任务
RR	较长	好	好	中	分时系统、交互式应用
MLFQ	较短	好	较好	高	通用操作系统（Linux、Windows）

3.4 死锁

（1）死锁定义与产生原因

定义：多个进程因竞争资源而造成相互等待，若无外力干预，这些进程将无法继续推进。

产生原因：

- **资源竞争**：不可抢占资源（如打印机）数量不足
- **推进顺序不当**：进程请求/释放资源的顺序不合理

---

(2) 死锁的四个必要条件（缺一不可）

条件	说明	破坏方法
互斥条件	资源一次仅被一个进程占用	无法破坏（某些资源本质互斥）
请求与保持	进程已持有一资源，又申请新资源	一次性分配所有资源；或请求前释放已占资源
不可抢占	已分配资源不能被强制剥夺	允许抢占（仅适用于状态易保存的资源）
循环等待	存在进程 - 资源的循环等待链	资源编号，按序申请；或资源分配图简化

---

(3) 死锁处理策略

策略	核心思想	具体方法	优缺点
预防	破坏死锁必要条件之一	资源预分配、有序分配、可抢占	简单但资源利用率低
避免	动态判断分配是否安全	银行家算法	资源利用率较高，但计算开销大
检测与解除	允许死锁发生，定期检测并恢复	资源分配图、等待图；终止进程/资源剥夺	灵活但恢复成本高
忽略	假设死锁不发生	鸵鸟算法	简单，适用于死锁概率极低的系统

---

(4) 银行家算法详解

数据结构:

```
int Available[m];        // 系统可用资源向量
int Max[n][m];           // 进程最大需求矩阵
int Allocation[n][m];    // 进程已分配矩阵
int Need[n][m];          // 进程还需资源矩阵 = Max - Allocation
bool Finish[n];          // 进程是否可完成标记
```

安全性检查算法:

```
bool isSafe() {
    Work = Available;           // 工作向量
    Finish[i] = false for all i;

    while (存在 Finish[i]==false 且 Need[i] <= Work) {
        Work += Allocation[i];   // 模拟进程完成，释放资源
        Finish[i] = true;
    }

    return (所有 Finish[i] == true);  // 存在安全序列则安全
}
```

资源请求处理：

```
bool requestResources(int pid, int Request[]) {
    // 1. 合法性检查
    if (Request > Need[pid]) return ERROR;  // 请求超过声明最大值
    if (Request > Available) { 等待; return WAIT; }  // 资源不足

    // 2. 试探性分配
    Available -= Request;
    Allocation[pid] += Request;
    Need[pid] -= Request;

    // 3. 安全性检查
    if (isSafe()) {
        正式分配; return OK;
    } else {
        恢复原状态; 进程等待; return WAIT;
    }
}
```

示例计算：

系统资源：A(10), B(5), C(7)

进程	Max(A,B,C)	Allocation(A,B,C)	Need(A,B,C)
P0	7,5,3	0,1,0	7,4,3
P1	3,2,2	2,0,0	1,2,2
P2	9,0,2	3,0,2	6,0,0
P3	2,2,2	2,1,1	0,1,1
P4	4,3,3	0,0,2	4,3,1

```
Available = (10-7, 5-2, 7-5) = (3,3,2)

安全性检查：
1. P1 Need(1,2,2) <= Available(3,3,2) → 执行 P1
   Available = (3+2, 3+0, 2+0) = (5,3,2)
2. P3 Need(0,1,1) <= (5,3,2) → 执行 P3
   Available = (5+2, 3+1, 2+1) = (7,4,3)
3. P0 Need(7,4,3) <= (7,4,3) → 执行 P0
   Available = (7+0, 4+1, 3+0) = (7,5,3)
4. P2 Need(6,0,0) <= (7,5,3) → 执行 P2
   Available = (7+3, 5+0, 3+2) = (10,5,5)
5. P4 Need(4,3,1) <= (10,5,5) → 执行 P4

安全序列：P1→P3→P0→P2→P4，系统安全
```

（5）资源分配图法检测死锁

图元素：

- 进程节点（圆圈）、资源节点（方框）
- 分配边（资源→进程）、请求边（进程→资源）

检测规则：

- 若图中无环路 → 无死锁
- 若每类资源仅 1 个实例，有环路 → 必死锁
- 若资源有多个实例，有环路 → 可能死锁，需进一步分析

简化算法：

```
while (存在非阻塞进程) {
  移除该进程的所有边（释放资源）;
  唤醒因该资源而阻塞的进程;
}
若图中仍有进程 → 这些进程构成死锁
```

四、存储器管理

4.1 存储器管理功能

功能	说明	关键点
内存分配	为进程分配内存空间，提高内存利用率	减少外部碎片
地址映射	逻辑地址 → 物理地址转换	静态重定位、动态重定位（寄存器）
内存保护	防止进程越界访问	界地址寄存器、存储键
内存扩充	虚拟存储技术	逻辑内存 > 物理内存

4.2 程序的装入与链接

（1）从源代码到执行

```
源代码(.c)
    | 编译/汇编
    ▼
目标模块(.o) ← 相对地址（从0开始）
    | 链接
    ▼
装入模块(.exe) ← 逻辑地址（从0开始）
    | 装入
    ▼
内存中的进程 ← 物理地址（实际内存位置）
```

(2) 地址类型对比

地址类型	产生阶段	特点	示例
符号地址	源代码	变量名、函数名	`int count;`
相对地址	目标模块	相对于模块起始的偏移	`offset: 0x100`
逻辑地址	装入模块	进程虚拟地址空间	`0x00400000`
物理地址	内存	实际内存单元地址	`0x80000000`

(3) 链接方式

方式	时机	优点	缺点	适用
静态链接	装入前	执行效率高	模块更新需重新链接	嵌入式、固件
装入时动态链接	装入时	便于模块共享更新	装入时间略长	通用应用
运行时动态链接	执行时	按需加载，节省内存	首次调用有延迟	大型应用、插件系统

(4) 装入方式

```
// 绝对装入（编译时确定物理地址）
// 仅适用于单道程序，地址冲突风险高

// 可重定位装入（静态重定位）
void static_relocation(module, base_addr) {
    for (每条指令 in module) {
        if (指令含地址) {
            指令地址 += base_addr;  // 一次性修改
        }
    }
}

// 动态重定位（运行时转换）
// 使用重定位寄存器（基址寄存器）
物理地址 = 逻辑地址 + 重定位寄存器值
// 优点：进程可在内存中移动，支持虚拟存储
```

4.3 连续分配方式

(1) 单一连续分配

```
内存布局：
┌─────────────┐
|    系统区    | ← 操作系统内核
├─────────────┤
|    用户区    | ← 单道程序独占
```

```
└──────────────┘
```

- ✓ 特点：简单，无碎片
- ✗ 缺点：内存利用率低，仅支持单任务
- 适用：早期单用户系统（如 MS-DOS）

(2) 固定分区分配

```
内存布局（分区大小不等）：
┌──────────────┐
│    系统区    │
├──────┬───────┤
│128MB │ 256MB │ ← 分区1、2
├──────┴───────┤
│    512MB     │ ← 分区3
└──────────────┘
```

- ✓ 特点：实现简单，多道程序
- ✗ 缺点：内部碎片（分区内未用空间）；分区大小固定，灵活性差
- 内部碎片计算：`碎片 = 分区大小 − 进程实际大小`

(3) 动态分区分配

核心思想：按需分配，分区大小 = 进程需求

数据结构：

- 空闲分区表：记录空闲区起始地址、大小、状态
- 空闲分区链：双向链表，便于插入删除

分配算法对比：

算法	查找起点	排序方式	特点
首次适应（FF）	链首	地址递增	保留高址大分区，低址小碎片
循环首次适应	上次位置	循环链表	均匀使用内存，查找快
最佳适应（BF）	链首	大小递增	保留大分区，但产生大量小碎片
最坏适应（WF）	链首	大小递减	减少小碎片，但大分区易耗尽

(4) 分区回收与合并

回收分区 R，检查相邻分区状态：

```
情况①：仅与前空闲分区F1相邻
  ┌──────┬──────┬──────┐
  │ F1   │  R   │ 占用 │ → 合并：F1大小 += R大小

情况②：仅与后空闲分区F2相邻
  ┌──────┬──────┬──────┐
  │ 占用 │  R   │ F2   │ → 合并：F2起始地址 = R起始，大小 += R大小
```

情况③：与前后空闲分区均相邻

```
┌──────┬──────┬──────┐
| F1   |   R  | F2   | → 三者合并为新F1，大小 = F1+R+F2
```

情况④：前后均被占用

```
┌──────┬──────┬──────┐
| 占用 |   R  | 占用 | → R作为新空闲分区插入链表
```

（5）伙伴系统（Buddy System）

核心思想：所有分区大小为 2^k，便于快速合并

分配过程（请求大小为 s）：

1. 找最小 k 使 2^k ≥ s
2. 若存在 2^k 空闲块，直接分配
3. 否则找更大块 2^m (m>k)，递归拆分：2^m → 2^(m-1) + 2^(m-1) → … → 2^k + 2^k
4. 分配一个 2^k 块，其余加入对应空闲链表

回收过程：

1. 释放块大小为 2^k，起始地址 addr
2. 计算伙伴地址：`buddy = addr XOR 2^k`
3. 若伙伴空闲且大小相同，合并为 2^(k+1)
4. 递归检查更大伙伴，直至无法合并

优缺点：

- ✓ 优点：合并/拆分效率高（位运算），外部碎片少
- ✗ 缺点：内部碎片（请求大小非 2 的幂）
- 适用：内核内存分配（如 Linux 的 buddy 分配器）

（6）对换（Swapping）

目的：缓解内存紧张，提高多道程序度

类型：

- 整体对换：以进程为单位换入换出
- 部分对换：以页/段为单位（虚拟存储基础）

对换策略：

- 换出：优先换出阻塞、低优先级、长时间未用进程
- 换入：优先换入就绪、高优先级、刚换出进程

4.4 基本分页存储管理

（1）分页基本概念

逻辑空间划分：

- 页面（Page）：进程逻辑空间的等长块，大小 2^k（如 4KB）
- 页号 P：页面编号，从 0 开始
- 页内偏移 W：页内地址，0 ~ 页面大小 -1

物理空间划分：

- 页框（Frame）：内存的等长物理块，大小 = 页面大小
- 块号 F：页框编号

页表（Page Table）：

- 作用：建立页号→块号的映射
- 表项内容：块号 + 状态位（存在/修改/访问/保护等）

（2）地址结构与变换

逻辑地址结构（32 位系统，页面 4KB=2^12B）：

```
┌──────────────┬──────────────┐
| 页号P (20位) | 偏移W (12位)|
└──────────────┴──────────────┘
```

地址变换公式：

```
页号  P = 逻辑地址 >> 12       // 右移12位
偏移  W = 逻辑地址 & 0xFFF     // 低12位掩码
块号  F = 页表[P].frame_number
物理地址 = (F << 12) | W
```

示例：

```
逻辑地址 = 0x12345（32位）
P = 0x12345 >> 12 = 0x12
W = 0x12345 & 0xFFF = 0x345
若页表[0x12].frame = 0x89
物理地址 = (0x89 << 12) | 0x345 = 0x89345
```

（3）地址变换机构

基本地址变换（无快表）：

```
CPU → 逻辑地址
      |
      ▼
页表基址寄存器(PTBR) → 页表起始物理地址
      |
      ▼
访问内存：读取页表项 → 获取块号F
      |
      ▼
合成物理地址：(F << 偏移位数) | W
      |
      ▼
访问内存：读取/写入数据
```

- 缺点：每次内存访问需 2 次内存访问（页表 + 数据），速度减半

快表（TLB, Translation Lookaside Buffer）：

- 高速缓存，存储近期使用的页表项
- 并行查找：逻辑地址页号同时匹配 TLB 所有项
- 命中：直接获取块号，1 次内存访问
- 未命中：查内存页表，更新 TLB，2 次内存访问

性能提升：

```
有效访问时间 = 命中率×(TLB 时间 + 内存时间) + 未命中率×(TLB 时间 + 2×内存时间)

例：TLB 时间 1ns，内存时间 100ns，命中率 95%
EAT = 0.95×(1+100) + 0.05×(1+200) = 95.95 + 10.05 = 106ns
无 TLB 时：200ns，性能提升近1倍
```

（4）多级页表

问题：32 位地址 +4KB 页面 → 页表项数=2^20=1M 项；每项 4B → 页表大小=4MB，需连续内存，浪费严重

解决方案：多级页表（以两级为例）

两级页表结构：

```
┌──────────────────────────────────────┐
| 逻辑地址：页目录号(10位)+页表号(10位)+偏移(12位) |
└──────────────────────────────────────┘
```

变换过程：

1. 页目录基址寄存器 → 页目录表物理地址
2. 页目录号 → 查页目录表 → 获取页表物理地址
3. 页表号 → 查页表 → 获取物理块号
4. 块号 + 偏移 → 物理地址

优缺点：

- ✓ 优点：页表离散存储，无需连续大内存；按需分配：仅分配使用的页表，节省空间

- ✗ 缺点：地址变换需多次内存访问（3 次：页目录 + 页表 + 数据），可配合 TLB 优化

扩展：64 位系统常用 4 级或 5 级页表

4.5 基本分段存储管理

（1）分段思想

分段依据：程序的逻辑结构

- 代码段（.text）：只读，可共享
- 数据段（.data）：已初始化全局变量
- 未初始化段（.bss）：未初始化变量
- 堆（heap）：动态分配，向上增长
- 栈（stack）：函数调用，向下增长

分段优势：

- ✓ 符合程序员视角，便于理解
- ✓ 段可独立保护（读/写/执行权限）
- ✓ 段可共享（如共享库代码段）
- ✓ 段可动态增长（堆、栈）

（2）逻辑地址与段表

逻辑地址结构（二维）：

```
┌──────────┬──────────────┐
| 段号 S   | 段内偏移 d |
└──────────┴──────────────┘
```

段表项结构：

```
┌──────────────────────────────────┐
| 段基址（物理起始地址）           |
| 段长（界限）                      |
| 存取权限（R/W/X）                |
| 存在位（是否在内存）              |
| 修改位/访问位                     |
| 外存地址（若被换出）              |
└──────────────────────────────────┘
```

地址变换：

1. 段号 S ≥ 段表长度？ → 越界中断
2. 偏移 d ≥ 段长？ → 段内越界中断
3. 检查存取权限 → 保护异常
4. 存在位=0？ → 缺段中断，调入段
5. 物理地址 = 段基址 + 偏移 d

（3）分段与分页对比

对比项	分页	分段
划分依据	物理单位，大小固定	逻辑单位，大小可变
地址维度	一维（线性）	二维（段号 + 偏移）
碎片问题	仅内部碎片	仅外部碎片
共享粒度	页级共享（可能共享无关代码）	段级共享（语义完整）
保护粒度	页级权限	段级权限（更符合需求）
地址变换	查页表	查段表 + 越界检查

4.6 段页式存储管理

结合分段与分页优势：

- 用户视角：分段（逻辑清晰，便于共享保护）
- 系统实现：分页（离散分配，无外部碎片）

地址结构：

变换过程：

1. 段号 → 查段表 → 获取该段的页表基址
2. 页号 → 查页表 → 获取物理块号
3. 块号 + 偏移 → 物理地址

访问次数：

- 无 TLB：3 次内存访问（段表 + 页表 + 数据）
- 有 TLB：可缓存段表项 + 页表项，加速访问

应用：现代操作系统（如 x86 保护模式）实际多采用纯分页 + 软件模拟分段

4.7 内存保护机制

保护方式	说明	特点
界地址寄存器法	下界寄存器 ≤ 物理地址 ≤ 上界寄存器	硬件自动比较，越界则异常
基址 - 限长寄存器法	0 ≤ 逻辑地址 < 限长寄存器，物理地址 = 基址 + 逻辑地址	同时实现重定位与保护

保护方式	说明	特点
存储键保护法	每个物理块关联一个存储键，进程 PCB 中存访问键	支持细粒度共享与隔离
环保护机构	环 0（内核）→ 环 3（应用），低编号环可访问高编号环数据	高编号环访问低编号环需通过门机制

五、虚拟存储器

5.1 常规存储管理的局限

传统管理特征：

- 一次性：作业必须全部装入内存才能运行
- 驻留性：作业运行期间始终驻留内存

导致问题：

- ✗ 大作业无法运行（内存 < 作业大小）
- ✗ 内存利用率低（部分代码/数据很少使用）
- ✗ 多道程序度受限（内存装不下更多作业）
- ✗ 并发性能瓶颈

5.2 局部性原理（虚拟存储理论基础）

类型	说明	原因	应用
时间局部性	刚被访问的指令/数据，近期可能再次访问	循环、子程序调用、栈操作	Cache、预取、保留最近页面
空间局部性	访问某地址后，其邻近地址可能被访问	顺序执行、数组遍历、结构体	预取相邻页面、大块传输

工作集模型：

- 定义：进程在时间窗口 Δ 内实际访问的页面集合
- 性质：工作集大小相对稳定，随程序阶段变化
- 应用：确定进程最小驻留集，预防抖动

5.3 虚拟存储器定义与特征

定义：具有请求调入和置换功能，能从逻辑上扩充内存容量的存储系统。

特征	说明
多次性（基础）	作业分多次调入内存运行，无需一次性装入全部代码/数据
对换性（基础）	运行中页面可换进换出，内存页面 ≠ 外存页面一一对应

特征	说明
虚拟性（目标）	用户感知的内存容量 >> 物理内存（32 位系统：4GB，64 位：理论 16EB）
离散性（实现手段）	页面/段可离散存放于内存/外存，通过页表/段表映射

5.4 请求分页存储管理

（1）页表机制扩展

请求分页页表项结构：

```
┌──────────────────────────────────────┐
| 物理块号（帧号）                      |
| 状态位 P（Present）：1=在内存，0=不在 |
| 访问位 A（Accessed）：记录是否被访问  |
| 修改位 M（Modified/Dirty）：1=已修改  |
| 保护位：读/写/执行权限                |
| 外存地址：页面在交换区的位置          |
└──────────────────────────────────────┘
```

各字段作用：

- P=0 时，访问该页触发缺页中断
- A 位用于页面置换算法（如 LRU、Clock）
- M=1 时，换出前需写回外存；M=0 可直接丢弃
- 保护位实现段级权限控制

（2）缺页中断机构

缺页中断处理流程：

1. CPU 访问逻辑地址 → 查页表 → 发现 P=0
2. 硬件产生缺页异常（Page Fault）
3. 陷入内核，保存现场
4. 检查访问合法性（地址是否有效、权限是否允许）
5. 若非法 → 发送 SIGSEGV 信号终止进程
6. 若合法 → 执行缺页处理：
    - a. 选择牺牲页（置换算法）
    - b. 若牺牲页 M=1，写回外存
    - c. 从外存读入缺页到空闲块
    - d. 更新页表项（块号、P=1、A=0、M=0）
    - e. 若使用 TLB，更新或失效对应项
7. 恢复现场，重新执行触发缺页的指令

特点：

- 在指令执行期间产生（非指令边界）
- 一条指令可能产生多次缺页（如跨页访问）
- 处理开销大（磁盘 I/O），需优化缺页率

（3）地址变换过程（请求分页）

```
逻辑地址 → 页号P + 偏移W
         |
         ▼
查页表项[ P ]
         |
         ├─ P=1（在内存）—→ 合成物理地址 → 访问内存
         |
         └─ P=0（不在内存）—→ 缺页中断
                                  |
                                  ▼
                              选择置换页面
                                  |
                                  ├─ 牺牲页 M=1 → 写回外存
                                  |
                                  ▼
                              从外存读入缺页
                                  |
                                  ▼
                              更新页表 + TLB
                                  |
                                  ▼
                              重新执行原指令
```

5.5 页面置换算法

（1）算法对比总览

算法	策略	优点	缺点	实现开销	实际应用
OPT	淘汰最远将来使用的页	理论最优，缺页率最低	需预知未来，无法实现	—	评价基准
FIFO	淘汰最早进入的页	简单，易于实现	可能违反局部性，出现 Belady 异常	低	少用
LRU	淘汰最近最久未使用的页	符合局部性，性能接近 OPT	需硬件支持，开销大	高	高端系统
Clock	近似 LRU，循环扫描	性能较好，实现简单	不如 LRU 精确	中	Linux/Windows
改进 Clock	考虑访问位 + 修改位	减少写回开销，更实用	逻辑稍复杂	中	主流系统

（2）各算法详解

最佳置换算法（OPT）：

```
// 伪代码（理论算法）
PageOptimalReplacement(pages[], future_access[]) {
    for each page in pages {
        next_use = 找到 page 下一次被访问的位置（在 future_access 中）;
        if (next_use == ∞) return page;  // 永不使用，优先淘汰
    }
    return next_use 最大的 page;  // 最远将来使用
}
```

先进先出算法（FIFO）：

```
// 队列实现
queue<Page> fifo_queue;

PageFIFOReplacement() {
    victim = fifo_queue.front();  // 队首页最老
    fifo_queue.pop();
    return victim;
}

void addPage(Page p) {
    fifo_queue.push(p);  // 新页加入队尾
}
```

Belady 异常示例：

```
物理块数=3，序列：1,2,3,4,1,2,5,1,2,3,4,5 → 缺页9次
物理块数=4，同一序列 → 缺页10次（反而增加！）
原因：FIFO 未考虑局部性，可能淘汰常用页
```

最近最久未使用算法（LRU）：

```
// 方法1：栈法（双向链表）
list<Page> lru_stack;  // 最近使用在栈顶

void accessPage(Page p) {
    if (p in lru_stack) {
        将 p 移至栈顶;  // 更新为最近使用
    } else {
        if (栈满) {
            victim = 栈底元素;  // 最久未使用
            移除栈底;
        }
        p 压入栈顶;
    }
}

// 方法2：时间戳法
struct Page {
    int last_access_time;  // 最后访问时间
```

```
};

PageLRUReplacement(pages[]) {
    return min_element(pages, by last_access_time);
}
```

Clock 置换算法（简单版）：

```
struct ClockNode {
    Page page;
    bool accessed;  // A位
    ClockNode* next;
};

ClockNode* hand;  // 时钟指针

PageClockReplacement() {
    while (true) {
        if (hand->accessed == false) {
            victim = hand->page;
            hand = hand->next;
            return victim;
        } else {
            hand->accessed = false;  // 清除访问位
            hand = hand->next;
        }
    }
}
```

改进型 Clock 算法：

```
// 考虑 (A, M) 组合，优先淘汰未修改页
// 扫描策略（四轮）：
// 第1轮：找 (0,0) - 最近未访问且未修改（最佳）
// 第2轮：找 (0,1) - 最近未访问但已修改（次选）
// 第3轮：重复第1轮（此时所有(0,1)的A位已清零）
// 第4轮：重复第2轮

PageImprovedClockReplacement() {
    for round in [1, 2, 3, 4] {
        start = hand;
        do {
            A = hand->accessed;
            M = hand->modified;

            if (round == 1 && A==0 && M==0) return hand->page;
            if (round == 2 && A==0 && M==1) return hand->page;

            // 第一轮扫描时，将(0,1)的A位清零
            if (round == 1 && A==0 && M==1) {
                hand->accessed = false;
            }

            hand = hand->next;
```

```
        } while (hand != start);
    }
}
```

（3）页面缓冲算法（PBA）

思想：区分"干净页"和"脏页"，延迟写回

置换流程：

1. 选牺牲页（如用 Clock 算法）
2. 若页未修改（M=0）：加入空闲链表，内容保留（零拷贝）
3. 若页已修改（M=1）：加入已修改链表，暂不写回
4. 需要新页面时：优先从空闲链表取页

优势：

- ✓ 减少磁盘写回次数（批量写）
- ✓ 若页面被再次访问，可直接重用（避免换入）
- ✓ 适合写多读少的工作负载

5.6 物理块分配策略

策略	说明	优点	缺点
固定分配局部置换	每个进程分配固定数量物理块，缺页时仅置换自己的页面	简单，进程间隔离	块数难设定，可能浪费
可变分配全局置换	进程块数动态调整，缺页时可置换任意进程的页面	内存利用率高	可能影响其他进程性能
可变分配局部置换	块数可变，但仅置换自己的页	平衡灵活性与隔离性	实现复杂，需监控缺页率

5.7 抖动与工作集

抖动（Thrashing）：

- 现象：页面频繁换进换出，CPU 利用率骤降
- 原因：多道程序度过高，总工作集 > 物理内存
- 表现：磁盘灯常亮，系统响应极慢

检测与预防：

方法	说明
降低多道程序度	挂起部分进程
工作集模型	确保 ∑工作集 ≤ 物理内存
局部置换策略	避免一个进程缺页影响全局

方法	说明
L/S 原则	若缺页处理时间 >> 缺页间隔，应挂起进程
增加物理内存	根本解决

工作集模型应用：

- 定义：进程在时间窗口 Δ 内访问的页面集合 W(t,Δ)
- 大小 |W|：进程最小驻留集
- 调度：仅当 ∑|W_i| ≤ 物理内存时，才允许进程运行

5.8 有效访问时间计算

公式推导：

```
设：
  ta = 内存访问时间（如100ns）
  td = 缺页处理时间（含磁盘I/O，如10ms=10^7ns）
  P  = 缺页率（0~1）

有效访问时间（EAT）：
  EAT = (1-P) × ta + P × (td + ta)
      = ta + P × td

示例：
  ta=100ns, td=10ms, P=0.001（千分之一缺页）
  EAT = 100 + 0.001×10^7 = 100 + 10000 = 10100ns
  → 性能下降100倍！

结论：缺页率必须极低（<0.1%）才能保证性能
```

5.9 请求分段存储管理

在分段基础上增加虚拟存储特性：

段表项扩展：

- 存在位：段是否在内存
- 外存地址：段被换出时的位置
- 段长动态增长：堆/栈段可自动扩展

缺段中断处理：

- 类似缺页中断，但以段为单位调入
- 一条指令可能触发多次缺段（跨段访问）

分段置换挑战：

- 段大小可变 → 外部碎片问题
- 解决方案：分段 + 分页（段页式）或纯分页

应用：

- 早期 UNIX 使用请求分段
- 现代系统多用分页模拟分段（如 mmap 映射文件段）

5.10 分段保护机制

保护方式	说明
越界检查	段号 ≥ 段表长度 → 越界中断；段内偏移 ≥ 段长 → 段内越界
存取权限检查	段表项含保护位：只读/只写/可执行，违规访问 → 保护异常
环保护机构	4 个环：0(内核)→3(应用)，低环可访问高环数据，反之需门机制

现代实践：

- x86 保护模式支持 4 环，但 Linux/Windows 仅用环 0 和环 3
- 段保护多由编译器 + 分页权限位实现
- DEP/NX 位：标记页不可执行，防止代码注入攻击

六、I/O 设备管理

6.1 设备分类

分类方式	类型	特点	示例
按使用特性	人机交互设备	低速，面向用户，需缓冲	键盘、鼠标、显示器
	存储设备	高速，块设备，需调度	磁盘、SSD、磁带
	网络通信设备	中高速，流式传输	网卡、调制解调器
按传输速率	低速设备	<1KB/s	键盘、鼠标
	中速设备	1KB/s~1MB/s	打印机
	高速设备	>1MB/s	磁盘、网卡
按信息交换单位	字符设备	以字符为单位，不可寻址	键盘、串口
	块设备	以数据块为单位，可随机访问	磁盘、U 盘

6.2 缓冲管理

（1）引入缓冲的原因

1. **解决速度不匹配**
    - CPU 速度 >> I/O 设备速度
    - 缓冲作为"蓄水池"，平滑速度差异

2. **减少中断频率**
    - 无缓冲：每字符/字节产生中断
    - 有缓冲：满/空时才中断，降低 CPU 开销
3. **放宽中断响应时限**
    - 缓冲提供时间窗口，允许延迟响应
4. **提高并行性**
    - CPU 与 I/O 设备可并行工作

(2) 缓冲类型与实现

缓冲类型	说明	特点
单缓冲	设备与进程共享一个缓冲区	简单，但并行度有限
双缓冲	两个缓冲区交替使用	实现设备与进程并行
循环缓冲	多个缓冲区组成环形队列	适用于高速设备（如磁盘）
缓冲池	系统维护一组公共缓冲区，三类队列：empty/full/dirty	支持动态分配，提高利用率

(3) 缓冲池管理

```
// 缓冲池数据结构
typedef struct {
    char *buffer;           // 缓冲区起始地址
    int size;               // 缓冲区大小
    enum { EMPTY, FULL, DIRTY } state;
    struct buffer *next;    // 链指针
} Buffer;

BufferPool {
    Buffer *emptyQ;    // 空缓冲区队列
    Buffer *fullQ;     // 满缓冲区队列（设备→进程）
    Buffer *dirtyQ;    // 脏缓冲区队列（进程→设备）
    semaphore mutex;   // 互斥访问缓冲池
};

// 进程从设备读数据
Buffer* getBufferFromDevice() {
    P(mutex);
    while (emptyQ == NULL) {
        V(mutex);
        // 可触发设备启动填充缓冲
        P(mutex);
    }
    Buffer *b = emptyQ;
    emptyQ = emptyQ->next;
    V(mutex);
    return b;
```

```
}
```

6.3 设备独立性

目标：应用程序与物理设备解耦

实现机制：

```
┌──────────────────────────────────┐
| 1. 逻辑设备名 → 物理设备映射     |
|    • 应用程序使用逻辑名（如"/dev/printer"）|
|    • 系统维护逻辑 - 物理映射表       |
|    • 支持动态绑定（如USB热插拔）   |
├──────────────────────────────────┤
| 2. 设备驱动程序抽象              |
|    • 统一接口：open/read/write/close|
|    • 不同设备实现各自驱动          |
├──────────────────────────────────┤
| 3. 设备无关性软件层              |
|    • 缓冲管理、错误处理、权限检查 |
└──────────────────────────────────┘
```

优势：

- ✓ 应用可移植：更换设备无需修改代码
- ✓ 系统可扩展：新增设备只需添加驱动
- ✓ 资源可共享：多进程通过逻辑名竞争设备

6.4 设备分配与回收

分配方式	说明	适用场景
独占分配	进程独享设备直至释放	打印机、磁带机
共享分配	多进程交替使用设备	磁盘、网卡
虚拟分配	将独占设备虚拟为共享设备（SPOOLing）	打印假脱机

分配数据结构：

- 设备控制表（DCT）：设备状态、类型、队列指针
- 控制器控制表（COCT）：控制器信息
- 通道控制表（CHCT）：通道信息
- 系统设备表（SDT）：全系统设备清单

6.5 磁盘存储器管理

（1）磁盘结构与参数

物理结构：

- 盘片（Platter）：双面存储，每面一磁头
- 磁道（Track）：同心圆，编号从外向内
- 扇区（Sector）：磁道分段，基本存储单位（通常 512B/4KB）
- 柱面（Cylinder）：各盘片同半径磁道集合

性能参数：

- 寻道时间（Seek Time）：磁头移动到目标磁道的时间（平均 3~15ms）
- 旋转延迟（Rotational Latency）：等待目标扇区旋转到磁头下
  - 平均 = 1/2 旋转周期（7200RPM → 4.17ms）
- 传输时间（Transfer Time）：读写数据的时间

（2）磁盘调度算法

算法	说明	特点
FCFS	按请求到达顺序服务	公平但寻道时间长
SSTF	选择距当前磁头最近的请求	减少平均寻道时间，可能饥饿
SCAN（电梯算法）	磁头单向移动，到达端点后反向	避免饥饿，响应时间较均匀
C-SCAN	单向扫描，到达端点后快速返回起点	响应时间更均匀，适合负载稳定
LOOK/C-LOOK	SCAN/C-SCAN 改进：不走到物理端点	减少无效移动，提高效率

算法对比（请求序列：98,183,37,122,14,124,65,67；起始磁道 53）：

算法	寻道顺序	总寻道距离
FCFS	53→98→183→37→…→67	640
SSTF	53→65→67→37→14→98→122→124→183	236
SCAN(向高)	53→65→67→98→122→124→183→37→14	299
C-SCAN	53→65→67→98→122→124→183→14→37	391
LOOK	类似 SCAN，但 183 后直接到 37	273

（3）磁盘可靠性技术

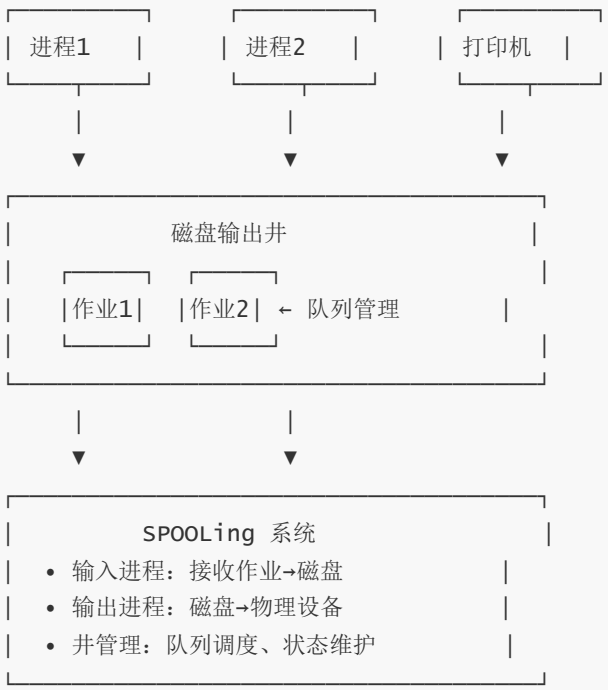
技术	说明	特点
磁盘镜像（RAID 1）	数据完全复制到另一磁盘	读性能提升，空间利用率 50%
磁盘条带化（RAID 0）	数据分块交替写入多磁盘	读写性能线性提升，无冗余

技术	说明	特点
分布式奇偶校验（RAID 5）	数据 + 奇偶校验块条带化分布	允许 1 盘故障，空间利用率 (n-1)/n
双重校验（RAID 6）	两个独立校验块，可容忍 2 盘故障	适合大容量磁盘阵列

6.6 SPOOLing 技术（假脱机）

目的：将独占设备虚拟为共享设备

工作原理（以打印为例）：



优势：

- ✓ 提高独占设备利用率
- ✓ 支持多进程"同时"使用设备
- ✓ 进程无需等待设备，提高并发度

七、文件管理

7.1 文件概念与属性

文件定义：具有文件名、逻辑上完整的信息集合，是外存管理的基本单位。

文件属性（元数据）：

属性	说明
文件名	用户标识

属性	说明
类型	普通文件/目录/设备文件等
位置	外存物理地址
大小	字节数/块数
保护	读/写/执行权限
时间	创建/修改/访问时间
所有者	用户/组标识

文件操作（系统调用）：创建/删除（create/delete）、打开/关闭（open/close）、读/写（read/write）、定位（lseek）、属性操作（chmod/chown/stat）

7.2 文件逻辑结构

类型	说明	适用场景
无结构文件	字节序列，无内部结构，操作系统不解释内容	二进制文件、可执行文件
有结构文件	由若干逻辑记录组成（定长/变长）	数据库、文本文件

记录组织方式：顺序文件（按顺序存储）、索引文件（建立索引加速查找）、散列文件（哈希函数直接定位）

7.3 文件物理结构（存储方式）

（1）连续分配

原理：文件占用磁盘上连续的物理块

- ✓ 优点：顺序访问快（无需寻址计算）；支持随机访问；实现简单
- ✗ 缺点：外部碎片；文件扩展困难；需预分配或动态移动
- 适用：只读文件、光盘、嵌入式系统

（2）链接分配（隐式链接）

原理：文件块离散存放，每块含指向下一块的指针

- ✓ 优点：无外部碎片；文件扩展灵活；无需预分配
- ✗ 缺点：仅支持顺序访问；指针占用数据空间；指针损坏导致文件断裂

改进：显式链接（FAT 表）

- 指针集中存于内存 FAT 表，磁盘块全用于数据
- 支持随机访问（查 FAT 链）
- 但 FAT 表可能很大

（3）索引分配

原理：为每个文件建立索引块，记录所有物理块号

多级索引（UNIX inode）：

- 直接地址：12 个指针 → 小文件快速访问
- 一级间接：1 个指针→索引块→1024 数据块
- 二级间接：1→1024→1024^2数据块
- 三级间接：1→1024^3数据块
- 理论最大：~4TB@4KB 块
- ✓ 优点：支持随机访问；无外部碎片；大文件支持好
- ✗ 缺点：索引块占用额外空间；多级索引访问需多次磁盘读取

（4）三种结构对比

特性	连续分配	链接分配	索引分配
顺序访问	快（连续）	中（需读指针）	快（索引缓存）
随机访问	快（直接计算）	慢（遍历链表）	快（查索引）
插入/删除	慢（需移动）	快（改指针）	快（改索引）
外部碎片	严重	无	无
空间利用率	低	高	中高
实现复杂度	低	中	高

7.4 目录管理

（1）目录结构演进

类型	说明	特点
单级目录	全系统一个目录表	简单，但不允许重名
两级目录	主目录（用户）+ 用户目录（文件）	不同用户文件可重名
树形目录	目录可含子目录，形成树	现代系统标准结构
无环图目录	允许文件/目录有多个父节点，通过硬链接/软链接实现共享	需引用计数避免循环

（2）文件共享与保护

共享机制：

机制	说明
硬链接	多个目录项指向同一 inode，仅同一文件系统内有效
软链接	特殊文件，内容为目标路径字符串，可跨文件系统

保护机制：

机制	说明
访问控制列表（ACL）	为文件指定用户/组的权限列表
权限位（UNIX rwx）	三类主体：所有者/组/其他，三种权限
密码保护	文件加密或访问密码
审计与日志	记录文件访问操作

7.5 文件系统实现

（1）文件系统层次

层次	说明
应用层	fopen/fread 等标准库调用
逻辑文件系统	解析路径，查找目录项，权限检查
文件组织模块	逻辑块号 → 物理块号映射
基本文件系统	读写物理块（扇区），缓冲区管理
设备驱动层	控制磁盘硬件，发送命令，处理中断

（2）磁盘空间管理

方法	说明
空闲表法	记录所有空闲区（起始 + 长度），适用连续分配
空闲链表法	空闲块链接成链表，适用离散分配
位图法	每块对应 1 位：0=空闲，1=已用，查找快
成组链接法	空闲块分组，每组记录下一组地址，高效批量分配/回收

八、磁盘存储器管理（专项补充）

8.1 磁盘性能优化技术

技术	说明
磁盘缓存	内存中缓存频繁访问的磁盘块，写策略：写直达/写回
磁盘预取	检测到顺序读模式时，提前读入后续块
磁盘调度优化	电梯算法（SCAN/C-SCAN），请求合并与排序
固态盘（SSD）优化	无机械部件，寻道时间≈0，重点优化写入放大、垃圾回收

8.2 磁盘可靠性与数据保护

技术	说明
校验与纠错（ECC）	每扇区附加校验码，检测/纠正位错误
坏块重映射	出厂预留备用扇区，运行时检测坏块自动替换
定期巡检	后台读取所有数据，校验完整性，发现静默错误
快照与克隆	写时复制（COW）实现时间点快照，用于备份、快速恢复

8.3 现代存储技术趋势

技术	说明
NVMe 协议	专为 SSD 设计，低延迟高并行，多队列、用户态驱动
存储级内存（SCM）	如 Intel Optane，介于内存与磁盘间，字节寻址
分布式存储	数据分片 + 多副本/纠删码，自动负载均衡，故障自愈
计算存储一体化	存储设备内置计算单元，数据本地处理

> **总结**：操作系统通过分层抽象与资源管理，将复杂硬件转化为易用接口。掌握进程调度、内存管理、文件系统等核心机制，是理解计算机系统工作原理的关键。