操作系统面试题

> 基于简历：操作系统（进程/线程/内存管理）基础，项目中使用多线程编程

一、进程与线程

Q1: 进程和线程的区别?

定义：

- **进程**：资源分配的基本单位，拥有独立的地址空间
- **线程**：CPU调度的基本单位，共享进程的地址空间

详细对比：

特性	进程	线程
地址空间	独立	共享
资源	独立拥有	共享进程资源
开销	创建、切换开销大	开销小
通信	IPC（管道、消息队列等）	直接访问共享内存
安全性	互不影响	一个线程崩溃可能影响整个进程

形象比喻：

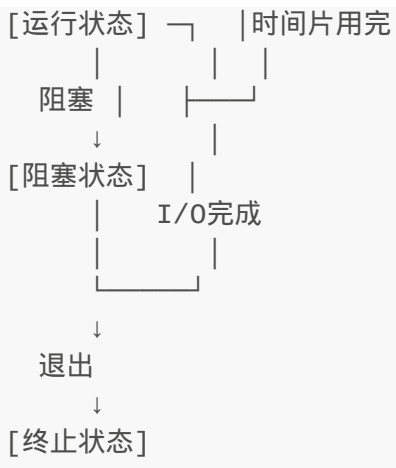
- 进程像一个公司，有独立的办公室、资源
- 线程像公司里的员工，共享公司资源

与你项目的联系： "文件传输系统使用多线程而不是多进程，因为线程间共享文件元数据哈希表，通信开销小。"

Q2: 进程的状态转换?

五态模型：

```
        创建
         ↓
    [新建状态]
         ↓
      调度
         ↓
    [就绪状态] ←──┐
         ↓        │
      获得CPU     │
         ↓        │
```



状态说明：

1. **新建**：进程正在创建
2. **就绪**：等待CPU调度
3. **运行**：正在CPU上执行
4. **阻塞**：等待I/O或事件
5. **终止**：执行结束

状态转换条件：

- 就绪→运行：调度器选中
- 运行→就绪：时间片用完、被抢占
- 运行→阻塞：等待I/O、等待锁
- 阻塞→就绪：I/O完成、事件发生

Q3: 线程的实现方式?

1. 用户级线程（User-Level Thread）

- 在用户空间实现，内核不感知
- 优点：切换快，无需系统调用
- 缺点：一个线程阻塞，整个进程阻塞

2. 内核级线程（Kernel-Level Thread）

- 由内核管理
- 优点：一个线程阻塞，其他线程继续执行
- 缺点：切换开销大，需要系统调用

3. 混合型

- 用户级线程映射到内核级线程
- Linux采用：NPTL（Native POSIX Thread Library）
- 一对一模型：一个用户线程对应一个内核线程

查看线程：

```
# 查看进程的所有线程
ps -T -p <pid>

# 查看线程详细信息
top -H -p <pid>
```

Q4: 进程间通信（IPC）的方式?

1. 管道（Pipe）

```
int pipefd[2];
pipe(pipefd);  // pipefd[0]读端，pipefd[1]写端

if (fork() == 0) {
    // 子进程：写数据
    close(pipefd[0]);
    write(pipefd[1], "Hello", 5);
    close(pipefd[1]);
} else {
    // 父进程：读数据
    close(pipefd[1]);
    char buf[100];
    read(pipefd[0], buf, sizeof(buf));
    close(pipefd[0]);
}
```

- 半双工，单向通信
- 只能用于父子进程或兄弟进程

2. 命名管道（FIFO）

```
mkfifo("/tmp/myfifo", 0666);
int fd = open("/tmp/myfifo", O_WRONLY);
write(fd, "Hello", 5);
```

- 有文件路径，无血缘关系的进程也能通信

3. 消息队列（Message Queue）

```
int msgid = msgget(key, IPC_CREAT | 0666);
msgsnd(msgid, &msg, sizeof(msg), 0);  // 发送
msgrcv(msgid, &msg, sizeof(msg), 0, 0);  // 接收
```

- 可以按消息类型接收

- 消息存在内核队列中

4. 共享内存（Shared Memory）

```
int shmid = shmget(key, size, IPC_CREAT | 0666);
void* addr = shmat(shmid, NULL, 0);  // 映射到进程地址空间
// 直接读写addr
shmdt(addr);  // 解除映射
```

- 最快的IPC方式
- 需要同步机制（信号量）保护

5. 信号量（Semaphore）

```
sem_t sem;
sem_init(&sem, 0, 1);  // 初始值1
sem_wait(&sem);  // P操作，减1
// 临界区
sem_post(&sem);  // V操作，加1
```

- 用于进程/线程同步
- 保护共享资源

6. 信号（Signal）

```
signal(SIGINT, handler);  // 注册信号处理函数
kill(pid, SIGINT);  // 发送信号
```

- 异步通知
- 常用于进程间通知

7. Socket

- 可用于本机进程通信（Unix域套接字）
- 也可用于网络通信

对比：

方式	速度	用途
管道	中等	父子进程简单通信
消息队列	中等	结构化消息传递
共享内存	最快	大量数据交换
信号	快	简单通知

方式	速度	用途
Socket	较慢	网络通信/复杂通信

Q5: 线程同步的方式?

1. 互斥锁（Mutex）

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&mutex);
// 临界区
pthread_mutex_unlock(&mutex);
```

- 保证同一时刻只有一个线程访问

2. 条件变量（Condition Variable）

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex;

// 等待
pthread_mutex_lock(&mutex);
while (!condition) {
    pthread_cond_wait(&cond, &mutex);
}
pthread_mutex_unlock(&mutex);

// 通知
pthread_cond_signal(&cond);  // 唤醒一个
pthread_cond_broadcast(&cond);  // 唤醒全部
```

- 配合互斥锁使用
- 用于线程间的等待/通知

3. 读写锁（RWLock）

```
pthread_rwlock_t rwlock;
pthread_rwlock_rdlock(&rwlock);  // 读锁（共享）
pthread_rwlock_wrlock(&rwlock);  // 写锁（独占）
pthread_rwlock_unlock(&rwlock);
```

- 读读不互斥，读写、写写互斥
- 适合读多写少场景

4. 信号量（Semaphore）

```
sem_t sem;
sem_init(&sem, 0, 1);
sem_wait(&sem);
// 临界区
sem_post(&sem);
```

- 可以控制多个线程访问

与你项目的联系： "文件传输系统：

- 用互斥锁保护对象池
- 用条件变量实现双缓冲日志的通知
- 用读写锁保护文件元数据哈希表"

二、死锁

Q6: 死锁的四个必要条件?

必要条件：

1. **互斥条件**：资源不能被共享，只能由一个线程使用
2. **请求和保持**：线程已经持有资源，又请求新资源
3. **不可剥夺**：资源不能被强制剥夺，只能主动释放
4. **循环等待**：存在一个线程链，每个线程持有下一个线程请求的资源

示例：

```
线程A持有锁1，等待锁2
线程B持有锁2，等待锁1
→ 形成循环等待 → 死锁
```

代码示例：

```
// 线程A
pthread_mutex_lock(&mutex1);
sleep(1);
pthread_mutex_lock(&mutex2);  // 死锁！
...
pthread_mutex_unlock(&mutex2);
pthread_mutex_unlock(&mutex1);

// 线程B
pthread_mutex_lock(&mutex2);
pthread_mutex_lock(&mutex1);  // 死锁！
...
pthread_mutex_unlock(&mutex1);
pthread_mutex_unlock(&mutex2);
```

Q7: 如何预防和避免死锁?

预防（破坏四个必要条件之一）：

1. 破坏互斥条件

- 将资源变为可共享（不总是可行）

2. 破坏请求和保持

- 一次性申请所有资源

```
lock_all({&mutex1, &mutex2});
// 使用资源
unlock_all({&mutex1, &mutex2});
```

3. 破坏不可剥夺

- 申请不到资源时，释放已持有的资源

```
while (true) {
    lock(mutex1);
    if (try_lock(mutex2)) {
        // 成功获取两个锁
        break;
    }
    unlock(mutex1);  // 释放已持有的锁
    sleep(rand());   // 随机等待，避免活锁
}
```

4. 破坏循环等待（最常用）

- 按固定顺序加锁

```
// 规定：总是先锁mutex1，再锁mutex2
void thread_func() {
    pthread_mutex_lock(&mutex1);  // 总是先锁1
    pthread_mutex_lock(&mutex2);  // 再锁2
    // 使用资源
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
}
```

避免（银行家算法）：

- 动态分配资源时，检查是否会导致死锁
- 如果会，则不分配，让进程等待

检测和恢复：

- 允许死锁发生，定期检测
- 检测到死锁后，终止进程或剥夺资源

与你项目的联系： "在实现对象池时，遇到过死锁问题。通过统一加锁顺序（总是先锁pool的mutex，再操作对象）解决。"

三、内存管理

Q8: 虚拟内存的作用?

虚拟内存：

- 每个进程有独立的虚拟地址空间
- 通过页表映射到物理内存

作用：

1. 隔离进程

```
进程A: 0x1000 → 物理地址 0x4000
进程B: 0x1000 → 物理地址 0x8000
```

- 不同进程访问相同虚拟地址，映射到不同物理地址
- 保护进程间不互相干扰

2. 扩大地址空间

- 32位系统：4GB虚拟地址空间
- 物理内存可能只有2GB
- 通过换页机制（Swap），虚拟内存 > 物理内存

3. 方便内存管理

- 连续的虚拟地址可以映射到不连续的物理内存
- 简化内存分配

4. 共享内存

- 不同进程的虚拟地址可以映射到同一物理页面
- 实现共享库、共享内存IPC

5. 保护机制

- 页表项包含权限位（读、写、执行）
- 非法访问触发异常

Q9: 分页和分段的区别?

分页（Paging）：

- 将地址空间分成固定大小的页（Page）
- 页大小：4KB、2MB、1GB
- 页表记录虚拟页到物理页的映射

```
虚拟地址 = 页号 + 页内偏移
    ↓
  页表
    ↓
物理地址 = 页框号 + 页内偏移
```

分段（Segmentation）：

- 将地址空间分成逻辑段（代码段、数据段、栈段）
- 段大小不固定

```
虚拟地址 = 段号 + 段内偏移
    ↓
  段表
    ↓
物理地址 = 段基址 + 段内偏移
```

对比：

特性	分页	分段
大小	固定	可变
逻辑意义	无	有（代码、数据）
碎片	内部碎片	外部碎片
用户感知	透明	用户可见

现代系统：

- Linux使用分页
- x86支持段页式（分段+分页）

Q10: 页面置换算法有哪些?

场景： 物理内存满了，需要换出一个页面，选择哪个?

1. FIFO（先进先出）

```
内存: [1][2][3]
访问: 4
换出: 1（最先进入）
内存: [4][2][3]
```

- 简单，但可能换出常用页面

2. LRU（最近最少使用）

```
内存: [1][2][3]  (1最久未使用)
访问: 4
换出: 1
内存: [4][2][3]
```

- 性能好，但实现复杂
- 需要记录每页的访问时间

实现：

```
class LRUCache {
    list<pair<int, int>> cache;  // key-value
    unordered_map<int, list<pair<int, int>>::iterator> map;  // key ->
iterator
    int capacity;

public:
    int get(int key) {
        if (map.find(key) == map.end()) return -1;
        // 移到链表头部
        cache.splice(cache.begin(), cache, map[key]);
        return map[key]->second;
    }

    void put(int key, int value) {
        if (map.find(key) != map.end()) {
            cache.erase(map[key]);
        } else if (cache.size() >= capacity) {
            // 删除链表尾部（最久未使用）
            map.erase(cache.back().first);
            cache.pop_back();
        }
        cache.push_front({key, value});
        map[key] = cache.begin();
    }
};
```

3. LFU（最不经常使用）

- 换出访问频率最低的页面

4. Clock（时钟）

- LRU的近似实现
- 使用访问位（Access Bit）
- 扫描页面，找到访问位为0的换出

与你项目的联系： "Raft项目中，如果实现缓存优化，可以用LRU算法缓存热点KV数据。"

Q11: 内存分配算法?

问题： 有多个空闲内存块，选择哪个分配?

1. 首次适应（First Fit）

```
空闲块: [10K] [5K] [20K]
请求: 8K
分配: 第一个 [10K]
```

- 从头开始找，找到第一个足够大的块
- 速度快，但产生小碎片

2. 最佳适应（Best Fit）

```
空闲块: [10K] [5K] [20K]
请求: 8K
分配: [10K]（最接近8K）
```

- 选择最小的足够大的块
- 减少浪费，但留下很小的碎片

3. 最坏适应（Worst Fit）

```
空闲块: [10K] [5K] [20K]
请求: 8K
分配: [20K]（最大的块）
```

- 选择最大的块
- 留下的碎片较大，可以继续使用

4. 伙伴系统（Buddy System）

```
初始: [1024K]
请求70K → 分裂 → [512K][512K]
```

```
          → 分裂 → [256K][256K][512K]
          → 分裂 → [128K][128K][256K][512K]
          → 分配128K
```

- Linux使用
- 大小为2的幂次
- 合并相邻的空闲块（伙伴）

四、调度算法

Q12: 进程调度算法有哪些?

1. 先来先服务（FCFS）

```
进程: P1(24ms) P2(3ms) P3(3ms)
顺序: P1 → P2 → P3
等待时间: 0, 24, 27
平均等待时间: 17ms
```

- 简单公平，但平均等待时间长

2. 短作业优先（SJF）

```
进程: P1(24ms) P2(3ms) P3(3ms)
顺序: P2 → P3 → P1
等待时间: 3, 0, 6
平均等待时间: 3ms
```

- 平均等待时间最短
- 但长作业可能饿死

3. 最短剩余时间优先（SRTF）

- SJF的抢占版本
- 来了更短的作业，抢占当前作业

4. 优先级调度

```
进程: P1(优先级3) P2(优先级1) P3(优先级2)
顺序: P2 → P3 → P1
```

- 按优先级调度
- 低优先级可能饿死
- 解决：优先级随时间提升（老化）

5. 时间片轮转（RR）

```
时间片: 10ms
进程: P1(24ms) P2(3ms) P3(3ms)
顺序: P1(10) → P2(3) → P3(3) → P1(10) → P1(4)
```

- 公平，响应时间好
- 时间片太小：切换开销大
- 时间片太大：退化为FCFS

6. 多级反馈队列

```
Q1: 时间片8ms, 最高优先级
Q2: 时间片16ms
Q3: FCFS, 最低优先级

新进程 → Q1
Q1用完时间片 → 降到Q2
Q2用完时间片 → 降到Q3
```

- Linux CFS（完全公平调度器）类似思想
- 兼顾响应时间和周转时间

五、文件系统

Q13: 硬链接和软链接的区别?

硬链接（Hard Link）：

```
ln file1 file2
```

- 多个文件名指向同一个inode
- 删除一个文件名，其他仍然有效
- 不能跨文件系统
- 不能链接目录

软链接（Symbolic Link）：

```
ln -s file1 file2
```

- file2存储file1的路径
- 类似Windows快捷方式
- 可以跨文件系统

- 可以链接目录
- 删除源文件，软链接失效

inode：

```
文件名 → inode号 → inode结构（元数据+数据块位置）
```

- inode存储文件的元数据（大小、权限、时间戳等）
- 硬链接共享inode
- 软链接有独立的inode

查看：

```
ls -li  # 显示inode号
stat file  # 查看inode信息
```

Q14: 如何减少磁盘I/O次数?

1. 缓存（Page Cache）

```
读文件 → 先查Page Cache → 没有再从磁盘读
写文件 → 先写Page Cache → 定期刷盘（sync）
```

- Linux自动缓存

2. 预读（Read-ahead）

- 顺序读时，预先读取后面的数据

3. 延迟写（Lazy Write）

- 写操作先放在缓存
- 积累一定量再批量写入磁盘

4. 合并I/O请求

- 多个小的I/O合并成一个大的I/O

5. 直接I/O（Direct I/O）

```
int fd = open(path, O_DIRECT);
```

- 绕过Page Cache
- 适合数据库等自己管理缓存的应用

与你项目的联系： "文件传输系统的异步日志，通过双缓冲批量刷盘，减少磁盘I/O次数。"

六、综合应用

Q15: fork()的工作原理?

fork()：

```
pid_t pid = fork();
if (pid == 0) {
    // 子进程
} else if (pid > 0) {
    // 父进程
} else {
    // fork失败
}
```

返回值：

- 子进程返回0
- 父进程返回子进程PID
- 失败返回-1

写时复制（Copy-on-Write）：

```
fork后：
父进程: [虚拟地址] → [物理页面]（只读）
子进程: [虚拟地址] → [同一物理页面]（只读）

写入时：
触发Page Fault → 复制页面 → 修改页表
```

- fork后不立即复制内存
- 只有写入时才复制（COW）
- 节省内存，提高效率

示例：

```
#include <unistd.h>
#include <stdio.h>

int main() {
    int x = 10;
    pid_t pid = fork();

    if (pid == 0) {
        x = 20;  // 子进程修改，触发COW
```

```
        printf("Child: x = %d\n", x);  // 20
    } else {
        sleep(1);
        printf("Parent: x = %d\n", x);  // 10
    }
    return 0;
}
```

Q16: 僵尸进程和孤儿进程?

僵尸进程（Zombie）：

- 子进程已结束，但父进程未调用wait()
- 子进程的PCB保留，占用资源
- 状态显示为<defunct>

危害：

- 占用进程号资源
- 大量僵尸进程可能耗尽进程号

解决：

```
// 父进程调用wait()
pid_t pid = wait(NULL);

// 或使用信号处理
signal(SIGCHLD, SIG_IGN);  // 忽略子进程结束信号
```

孤儿进程（Orphan）：

- 父进程先结束，子进程还在运行
- 子进程被init进程（PID=1）收养
- init会自动wait()，不会产生僵尸进程

示例：

```
// 创建守护进程（daemon）
if (fork() > 0) exit(0);  // 父进程退出
setsid();  // 创建新会话
// 子进程变为守护进程
```

七、快速复习清单

进程与线程

- [ ] 进程vs线程的区别
- [ ] 进程状态转换
- [ ] 进程间通信（7种方式）
- [ ] 线程同步（4种方式）

死锁

- [ ] 死锁四个必要条件
- [ ] 如何预防死锁
- [ ] 银行家算法

内存管理

- [ ] 虚拟内存的作用
- [ ] 分页vs分段
- [ ] 页面置换算法（FIFO/LRU/Clock）
- [ ] 内存分配算法

调度算法

- [ ] FCFS/SJF/RR
- [ ] 多级反馈队列

文件系统

- [ ] 硬链接vs软链接
- [ ] 如何减少磁盘I/O

综合

- [ ] fork()的COW机制
- [ ] 僵尸进程vs孤儿进程

面试技巧

结合项目回答

示例： "您问的XX问题，在我的文件传输项目中有实际应用。比如..."

画图说明

- 进程状态转换 → 画状态图
- 死锁 → 画资源分配图
- 内存管理 → 画地址转换图

对比回答

"X和Y的区别有几个方面：

1. 定义上...

2. 使用场景...
3. 性能上... 在我的项目中，我选择了X，因为..."

记住：理论 + 实践 + 项目 = 高分