j# 课堂小测

1. 不同 ISA 的状态寄存器

\* x86:EFLAGS(32位)或 RFLAGS(64位)寄存器.包含进位标志(CF)、奇偶标志(PF)、零标志(ZF)、符号标志(SF)、溢出标志(OF)、方向标志(DF)、中断允许标志(IF)等.

\* ARM:CPSR(Current Program Status Register)或 APSR(Application Program Status Register)以及 SPSR(Saved Program Status Register).包含条件码标志(N,Z,C,V),中断屏蔽位(I,F),Thumb状态位(T),模式位(M)等.

\* RISC-V:没有像 x86 或 ARM 那样的单一专用状态寄存器来存放所有条件标志.条件判断通常由比较指令直接产生结果存入通用寄存器或直接用于分支.但是,RISC-V 有一系列控制和状态寄存器(CSRs),例如 mstatus,sstatus,ustatus 用于管理特权级、中断使能状态等；fcsr 用于浮点状态.传统意义上的"状态"(如进位、零标志)的处理方式与其他 ISA 不同.

2. 不同 ISA 对应的访管指令是什么?

\* x86:INT n(例如 Linux 上的 INT 0x80),SYSCALL(64位模式下的快速系统调用),SYSENTER(32位模式下的快速系统调用).

\* ARM:SVC(Supervisor Call,曾用名 SWI - Software Interrupt).

\* MIPS:SYSCALL.

\* RISC-V:ECALL(Environment Call).

3. RISC-V 架构中用于发起系统调用的标准指令是什么?该指令的执行过程有哪些重要步骤?

\* 指令:ECALL(Environment Call).

\* 执行过程重要步骤: 1. 触发陷入(Trap):ECALL 指令触发一个环境调用异常,异常来源是当前执行的特权模式(用户态 U-mode,监管态 S-mode 或机器态 M-mode).

2. 切换特权级:处理器通常切换到更高的特权级(例如,从 U-mode 到 S-mode 或 M-mode,具体取决于系统配置).

3. 保存 PC:当前程序计数器(PC)的值被保存到相应的异常程序计数器寄存器中(M-mode 保存到 mepc,S-mode 保存到 sepc,U-mode 保存到 uepc - 如果配置了用户态陷入).

4. 保存状态:当前的一些状态信息(如先前的特权级、中断使能状态)被保存到相应的状态寄存器(mstatus,sstatus,ustatus)中.

5. 更新状态:状态寄存器被更新(例如,中断可能被禁用,特权级被改变).

6. 设置 PC:程序计数器(PC)被设置为异常处理程序的入口地址,该地址由相应的陷入向量基地址寄存器(mtvec,stvec,utvec)指定.

7. 执行处理程序:异常处理程序代码(通常是操作系统内核的一部分)开始执行.它首先检查陷入原因寄存器(mcause,scause,ucause)来确定陷入的原因(在此情况下是来自特定模式的 ECALL).

8. 系统调用分发:处理程序根据传递的系统调用号(见问题4)找到并执行相应的内核函数.

9. 返回:系统调用完成后,处理程序执行特权返回指令(MRET,SRET,URET)返回到用户程序.该指令会恢复之前保存的 PC(从 mepc/sepc/uepc)和状态(从 mstatus/sstatus/ustatus),使用户程序在 ECALL 指令之后继续执行.

4. 在 RISC-V 上执行系统调用时,系统调用号和参数是如何传递的?

\* 约定:遵循标准的 RISC-V ABI(Application Binary Interface).

\* 系统调用号:通过寄存器 a7 传递.

\* 参数:前六个参数通过寄存器 a0 到 a5 传递.

\* 返回值:通过寄存器 a0 返回(对于 64 位或更宽的返回值,可能还使用 a1).

5. Linux 中,syscall 和 int 0x80 的作用是什么?它们在执行时有什么不同?

\* 作用:两者都是用户空间程序向 Linux 内核请求服务(即发起系统调用)的机制.

\* int 0x80(传统方式): \* 机制:使用 x86 架构的软件中断机制.

\* 处理器:在 32 位(i386)和 64 位(x86-64)处理器上都可用,但主要是 32 位 Linux 的传统方法.

\* 性能:相对较慢,因为它涉及完整的中断处理流程,开销较大(保存/恢复更多状态,查询中断向量表等).

\* 参数传递(32位):系统调用号在 eax,参数依次在 ebx,ecx,edx,esi,edi,ebp.

\* syscall(现代/快速方式): \* 机制:x86-64 架构引入的专门用于系统调用的指令(32位对应有 SYSENTER).

\* 处理器:主要用于 x86-64 架构的 Linux 系统,需要 CPU 支持.

\* 性能:比 int 0x80 更快,因为它使用了更直接、优化的路径进入内核系统调用处理程序,通常涉及更少的状态保存/恢复和优化的模式切换(例如通过 MSR).它不经过通用的中断描述符表(IDT).

\* 参数传递(64位):系统调用号在 rax,参数依次在 rdi,rsi,rdx,r10,r8,r9.注意寄存器约定与 int 0x80 不同.syscall 会修改 rcx 和 r11 寄存器.

\* 主要不同点总结: \* 机制:中断 vs. 专用指令.

\* 速度:syscall 通常更快.

\* 架构:int 0x80 较老,兼容 32/64 位；syscall 是 64 位标准.

\* 寄存器约定:传递系统调用号和参数使用的寄存器不同.

\* 状态保存:syscall 保存和修改的寄存器较少(rcx 存 RIP,r11 存 RFLAGS 并被修改)；int 0x80 作为中断处理的一部分需要保存更多上下文.

6. 不同 ISA 上怎样找到系统调用入口程序?

\* 通用机制:操作系统通常会设置一个中断/异常向量表(Interrupt/Exception Vector Table)或一个特定的控制寄存器,指向一个统一的异常入口点或分发器.当系统调用指令(如 ECALL,SVC,SYSCALL,INT)执行时,硬件会触发一个特定的异常或中断,并将控制权转移到这个预设的入口地址.

\* x86: \* 对于传统的 INT 0x80,CPU 通过查询中断描述符表(IDT - Interrupt Descriptor Table)找到索引 0x80 对应的门描述符,从中获取内核系统调用处理程序的入口地址和段选择子.

\* 对于 SYSCALL(64位)/ SYSENTER(32位),CPU 通过特定的模型专用寄存器(MSRs)直接获取内核入口点的地址(例如,IA32\_LSTAR MSR 存储 SYSCALL 的目标 RIP).这种方式绕过了 IDT 查询,速度更快.

\* ARM: \* 当执行 SVC 指令时,会触发 SVC 异常.CPU 跳转到异常向量表中 SVC 异常对应的处理程序地址.该向量表的基地址通常存储在 VBAR(Vector Base Address Register)中.

\* RISC-V: \* 执行 ECALL 指令会根据当前特权级触发相应的环境调用异常(Environment Call Exception from U/S/M-mode).CPU 将 PC 设置为对应特权级的陷阱处理基地址寄存器(mtvec,stvec,utvec)中指定的地址.该寄存器可以指向一个统一的处理入口(Direct mode)或一个向量表(Vectored mode),操作系统在此入口处进一步判断异常原因(通过 mcause/scause/ucause)并分发到系统调用处理程序.

7. 不同的 ISA ,当发生中断/异常/系统调用时,上下文是怎么保存的?

上下文保存通常分两部分:硬件自动保存和软件(操作系统处理程序)保存.

\* 通用过程: 1. 硬件:自动保存最关键的几个寄存器,足以让系统能够恢复到中断/异常发生之前的状态并知道发生了什么.通常包括: \* 程序计数器(PC / EIP / RIP):保存中断/异常发生时的指令地址.

\* 状态寄存器(FLAGS / CPSR / xstatus):保存当时的处理器状态(如条件码、中断使能位、特权级等).

\* 有时硬件还会将栈指针切换到内核栈.

2. 软件(OS Handler):在异常处理程序的入口处,操作系统负责保存其余需要保留的用户态上下文,主要是通用寄存器(General-Purpose Registers,GPRs),可能还有浮点寄存器等.这些信息通常保存在当前进程的内核栈或特定的进程控制块(PCB)区域.

\* 具体 ISA: \* x86: \* 硬件:发生中断/异常时,硬件自动将 EFLAGS/RFLAGS,CS(代码段寄存器),EIP/RIP(指令指针)压入当前栈(如果是从用户态陷入内核,则切换到内核栈).对于某些异常,还会压入一个错误码.

\* 软件:内核处理程序入口代码(汇编)负责保存所有通用寄存器(rax,rbx,rcx,rdx,rsi,rdi,rbp,r8-r15 等)以及段寄存器(ds,es,fs,gs)等到内核栈.

\* ARM: \* 硬件:发生异常时,硬件将当前的程序状态寄存器(CPSR)保存到对应异常模式的保存程序状态寄存器(SPSR\_irq,SPSR\_svc 等),并将返回地址(下一条指令的地址或当前指令地址+偏移)保存到对应模式的链接寄存器(LR\_irq,LR\_svc 等).处理器模式切换到相应的异常模式.

\* 软件:内核异常处理程序负责保存通用寄存器(r0-r12,有时包括 sp,lr)到内核栈.

\* RISC-V: \* 硬件:发生陷阱(trap)时,硬件将当前的 PC 保存到相应的 xepc 寄存器(mepc,sepc,uepc).先前的特权级保存在 xstatus 寄存器(mstatus,sstatus,ustatus)的 xPP 字段中.先前 xstatus 中的中断使能位 xIE 被保存到 xPIE 字段,然后硬件通常会禁用中断.

\* 软件:内核陷阱处理程序负责保存所有通用寄存器(x1-x31,因为 x0 恒为 0)到内核栈或进程控制块中.

8. 如何给出当前系统中有多少个进程在运行?

这通常指的是处于"运行"(Running)或"就绪"(Ready/Runnable)状态的进程数量.在类 Unix 系统(如 Linux)中,可以通过以下方法查看: \* 命令行工具: \* ps aux:列出所有进程的详细信息.可以查看 STAT(状态)列,R 表示正在运行或在运行队列中(就绪).可以通过 grep 或 awk 进一步筛选和计数.例如:ps aux | awk '$8 == "R" { count++ } END { print count }'(注意:STAT 列的位置可能因 ps 版本或选项而异).

\* top 或 htop:交互式进程查看器.通常会在顶部摘要区域显示总进程数、正在运行(running)的进程数、睡眠(sleeping)、停止(stopped)、僵尸(zombie)的进程数.running 通常指 R 状态的进程.

\* vmstat:报告虚拟内存统计信息,其输出的第一列 r 表示在运行队列中等待运行的进程数(就绪状态)加上当前正在运行的进程数.

\* 程序化方式(Linux): \* 读取 /proc 文件系统.每个数字命名的子目录 /proc/[pid]/ 代表一个进程.可以遍历这些目录,读取 /proc/[pid]/stat 或 /proc/[pid]/status 文件来获取进程状态信息.状态为 R(running)的进程就是目标.

需要注意,"正在运行"可以有两种理解: 1. 当前真正在 CPU 上执行的进程(数量 <= CPU核心数).

2. 处于 TASK\_RUNNING 状态的进程,包括正在 CPU 上运行的和在运行队列中等待调度的(就绪状态).命令行工具通常显示的是后者.

9. 总结一下 3/5/7 状态的进程模型的异同点

\* 3-状态模型:最基础的模型.

\* 状态:运行(Running),就绪(Ready),阻塞(Blocked/Waiting).

\* 描述:进程要么在 CPU 上执行(Running),要么准备好可以执行只等 CPU(Ready),要么因等待某事件(如 I/O 完成)而不能执行(Blocked).

\* 5-状态模型:在 3 状态基础上增加了创建和结束状态.

\* 状态:新建(New),就绪(Ready),运行(Running),阻塞(Blocked/Waiting),终止(Terminated/Exit).

\* 描述:增加了进程刚被创建(资源尚未完全分配)的 New 状态和进程执行完毕等待系统回收资源的 Terminated 状态.更完整地描述了进程的生命周期.

\* 7-状态模型:在 5 状态基础上增加了挂起状态,用于处理内存不足或需要换出到外存的情况.

\* 状态:新建(New),就绪(Ready),运行(Running),阻塞(Blocked),挂起就绪(Suspended Ready),挂起阻塞(Suspended Blocked),终止(Terminated).

\* 描述:引入了"挂起"概念.Suspended Ready 指进程已就绪但被换出到外存,需要换入内存才能运行.Suspended Blocked 指进程在等待事件且被换出到外存.这允许操作系统在内存紧张时将非活动进程移出内存,提高内存利用率.

\* 相同点: \* 都包含核心的 Running,Ready,Blocked 三种状态,描述了进程执行、等待 CPU、等待事件的基本情况.

\* 状态间的核心转换逻辑(如 Ready -> Running 的调度,Running -> Blocked 的等待事件,Blocked -> Ready 的事件完成,Running -> Ready 的时间片用完或被抢占)在所有模型中都存在.

\* 不同点: \* 复杂度:状态数量和模型复杂度递增(3 < 5 < 7).

\* 完备性:5 状态比 3 状态增加了进程生命周期的开始和结束；7 状态比 5 状态增加了对内存管理(换入换出/挂起)的考虑.

\* 引入状态:5 状态引入 New 和 Terminated；7 状态在 5 状态基础上引入 Suspended Ready 和 Suspended Blocked.

10. XV6 的进程状态模型是怎样的?

XV6采用六状态进程模型,在proc.h中定义如下: enum procstate { UNUSED,USED,SLEEPING,RUNNABLE,RUNNING,ZOMBIE };

各状态含义及转换: - UNUSED:进程槽未被使用,表示初始状态或已释放的进程.

- USED:已分配资源但尚未完全初始化的临时状态.

- SLEEPING:进程正在等待某事件(如I/O完成),处于休眠状态.

- RUNNABLE:进程已准备好运行,在就绪队列中等待CPU调度.

- RUNNING:进程当前正在CPU上执行.

- ZOMBIE:进程已终止,但父进程尚未通过wait()回收其资源.

主要状态转换路径: - UNUSED → USED:allocproc()分配进程时

- USED → RUNNABLE:userinit()和fork()完成初始化后

- RUNNABLE → RUNNING:scheduler()调度进程

- RUNNING → RUNNABLE:yield()让出CPU或时间片用完

- RUNNING → SLEEPING:sleep()等待事件

- SLEEPING → RUNNABLE:wakeup()唤醒进程

- 任何状态 → ZOMBIE:exit()终止进程

- ZOMBIE → UNUSED:wait()回收子进程资源

11. XV6 的 PCB(Process Control Block)中包含哪些重要信息?

XV6的PCB定义在proc.h中的struct proc结构体: struct proc {

struct spinlock loc // 进程锁，保护进程结构的访问

// 需要持有p->lock访问的字段:

enum procstate state; // 进程状态 [进程共享]

void \*chan; // 睡眠通道 [进程共享]

int killed; // 是否被标记为终止 [进程共享]

int xstate; // 退出状态码 [进程共享]

int pid; // 进程ID [进程共享]

// 需要持有wait\_lock访问

struct proc \*parent; // 父进程 [进程共享]

// 以下为进程私有，无需持有p->lock:

uint64 kstack; // 内核栈虚拟地址 [每个线程不同]

uint64 sz; // 进程内存大小(字节) [进程共享]

pagetable\_t pagetable; // 用户页表 [进程共享]

struct trapframe \*trapframe;// 陷入帧 [线程不同]

struct context context; // 上下文(用于进程切换) [每个线程不同]

struct file \*ofile[NOFILE]; // 打开的文件 [进程共享]

struct inode \*cwd; // 当前工作目录 [进程共享]

char name[16]; // 进程名(用于调试) [进程共享]

}; - 进程状态管理: - state(enum procstate):记录进程当前状态(UNUSED/USED/SLEEPING/RUNNABLE/RUNNING/ZOMBIE),是调度决策的基础

- chan(void\*):睡眠通道指针,进程等待特定事件时使用,唤醒时通过此标识找到对应进程

- killed(int):标志进程是否被终止,值为1表示进程已被标记为需要终止

- xstate(int):进程退出状态码,供父进程通过wait()系统调用获取

- 进程标识: - pid(int):进程唯一标识符,用于区分系统中的不同进程

- name(char[16]):进程名称,主要用于调试和进程监控

- 进程关系: - parent(struct proc\*):指向父进程的指针,用于实现进程树结构和wait()机制

- 内存管理: - kstack(uint64):内核栈的虚拟地址,进程在内核态执行时使用

- 每个进程都有自己的内核栈(kstack),这是因为: 1. 并发安全:多个进程可能同时在内核态执行(如一个在系统调用中,另一个在中断处理中),各自独立的内核栈确保它们不会相互干扰

2. 上下文隔离:每个进程在内核态的执行上下文(如局部变量、返回地址)需要独立保存,防止进程间状态混淆

3. 嵌套中断处理:进程在处理一个系统调用时可能发生中断,需要单独的栈空间来处理这种嵌套情况

4. 安全性:独立内核栈防止一个进程的内核态操作破坏其他进程的内核栈内容,增强系统稳定性

- sz(uint64):进程用户空间大小(以字节为单位),用于内存管理和边界检查

- pagetable(pagetable\_t):指向进程页表的指针,定义了虚拟地址到物理地址的映射关系

- 上下文切换: - trapframe(struct trapframe\*):保存用户态到内核态切换时的寄存器状态,包含用户程序计数器(pc)、栈指针(sp)和其他通用寄存器,当系统调用或中断发生时用于保存用户态上下文

- context(struct context):保存内核态上下文切换信息,包含内核态寄存器状态,用于进程调度切换

- 文件系统相关: - ofile(struct file\*[NOFILE]):进程打开文件表,每个元素指向一个打开的文件结构

- cwd(struct inode\*):当前工作目录的inode指针,用于相对路径解析

进程切换机制: XV6中的进程切换主要通过以下步骤完成: 1. 保存当前进程上下文: - 当进程需要让出CPU时(如时间片用尽、等待I/O等),调用swtch()函数

- swtch()将当前CPU寄存器状态保存到当前进程的context结构中

- 保存的寄存器包括:ra,sp,s0-s11(RISC-V架构中的调用者保存寄存器)

2. 加载新进程上下文: - 从新进程的context结构中恢复寄存器状态到CPU

- 这使得CPU继续执行新进程之前的指令流

3. 页表切换: - 通过修改RISC-V的satp寄存器,将当前页表切换为新进程的页表(pagetable)

- 这改变了虚拟地址到物理地址的映射关系

4. 内核栈切换: - 每个进程有自己的内核栈(kstack),切换进程时也会切换内核栈

- 这确保了每个进程在内核态有独立的执行环境

进程切换通常发生在以下情况: - 时钟中断触发时间片用尽

- 进程主动调用sleep()等待某事件

- 进程执行系统调用需要等待资源

- 进程终止执行

12. 进程的有效代码在地址空间中是从 0x0 开始的吗?为什么?

答:不一定.虽然逻辑上代码段通常被链接器放置在虚拟地址空间的低地址区域,但并不总是从 0x0 开始.

\* 原因 1:空指针解引用检测(Null Pointer Dereference Detection):操作系统通常会将虚拟地址空间的最低部分(例如第一个页,地址 0 到 4095 或更大范围)标记为不可访问.这样,如果程序试图通过空指针(null pointer,通常值为 0)进行读写,会立即触发一个硬件异常(如段错误 Segmentation Fault),便于调试.

\* 原因 2:ELF/PE 头信息:可执行文件格式(如 ELF,PE)本身在文件开头包含元数据(头部信息),实际的代码段(.text section)会在这些头之后开始.加载器会将这些部分映射到内存,代码的起始虚拟地址由链接器决定,并记录在文件头中,不一定是 0x0.

\* 原因 3:地址空间布局随机化(ASLR - Address Space Layout Randomization):为了安全,现代操作系统常常启用 ASLR.这会随机化进程地址空间中各个段(代码、堆、栈、库)的基地址,使得攻击者难以预测内存地址.启用 ASLR 后,代码段的起始地址几乎肯定不是 0x0.

13. 进程运行时 PC 指向的地址是(物理内存地址/虚拟地址)?采用这一地址方式的好处是什么?

答:进程运行时 PC(Program Counter)指向的是 虚拟地址(Virtual Address).

\* 好处: 1. 进程隔离(Process Isolation):每个进程拥有自己独立的虚拟地址空间,防止相互干扰,提高安全性.

2. 内存管理简化(Simplified Memory Management):程序员和编译器面对的是连续的虚拟地址空间,操作系统负责复杂的物理内存映射.

3. 高效的内存使用(Efficient Memory Usage):通过按需分页和交换技术,可以使用大于物理内存的地址空间,提高物理内存利用率.

4. 共享内存(Shared Memory):不同进程可以方便地将同一物理内存映射到各自空间,实现共享库和进程间通信.

5. 灵活性(Flexibility):程序可以在物理内存的任何位置加载和运行,无需修改代码.

14. 课堂上讲解的 myval 程序的执行:重复运行两次相同程序 myval 5,位置一样吗?为什么地址不一样?

答:重复运行两次相同程序 myval 5,其加载到内存的位置(地址)通常不一样.

\* 原因:现代操作系统普遍采用 地址空间布局随机化(ASLR - Address Space Layout Randomization)技术.

\* ASLR 是一种安全机制,用于防止内存攻击.

\* 每次程序启动时,操作系统会随机化进程地址空间中各个主要部分(代码段、堆、栈、库等)的基地址.

\* 因此,同一程序每次运行时其虚拟内存地址都会不同,增加了攻击者预测地址的难度.

程序代码如下: int myval; // 全局变量

int main(int argc,char \*argv[])

{

myval = atoi(argv[1]); // 将命令行参数转换为整数

while(1)

printf("myval is %d,loc 0x%lx\n",myval,(long)&myval); // 打印值和地址

}

这个程序声明了一个全局变量 myval,从命令行参数读取值,然后无限循环打印该变量的值和内存地址.由于ASLR的作用,每次运行时 &myval 的地址都会不同.

15. 为什么线程要有自己的栈?栈在哪里?栈里保存什么信息?

答: \* 为什么需要自己的栈: 1. 独立的执行流:管理各自的函数调用、返回地址和执行状态.

2. 局部变量存储:每个线程需要独立空间存放自己的函数局部变量副本.

3. 函数调用参数和返回值:独立处理函数调用时的参数传递和返回值.

4. 上下文切换:独立的栈指针是线程上下文切换的关键部分.

\* 栈在哪里: \* 位于进程的虚拟地址空间内.

\* 主线程栈通常由 OS 在进程创建时分配在高地址区.

\* 其他线程栈由线程库或程序员在虚拟地址空间中(如堆区或映射区)动态分配.

\* 栈里保存什么信息: 1. 函数返回地址:调用函数后下一条指令的地址.

2. 函数参数:传递给函数的参数.

3. 局部变量:函数内部定义的非静态局部变量.

4. 保存的寄存器值:如基址指针、通用寄存器等,用于函数调用和返回.

5. 函数调用的上下文信息(栈帧 - Stack Frame):包含上述信息的结构,每次函数调用创建一个.

\* 线程切换过程: 1. 保存当前线程上下文:将当前线程的寄存器值(包括程序计数器PC、栈指针SP、通用寄存器等)保存到其线程控制块(TCB)或栈中.

2. 选择下一个线程:调度器根据调度算法选择下一个要运行的线程.

3. 恢复目标线程上下文:从目标线程的TCB或栈中加载其保存的寄存器值,包括恢复其栈指针SP指向该线程的栈.

4. 切换执行流:通过恢复程序计数器PC的值,CPU开始执行目标线程的指令.

5. 注意事项:与进程切换不同,线程切换不需要切换页表(因为同一进程的线程共享地址空间),因此开销更小、速度更快.线程切换主要涉及寄存器状态和栈指针的切换,而不涉及地址空间的切换.

16. 同一个进程的两个线程:例如线程1是否可以访问线程2的栈?举一个例子进行说明.

答: \* 理论上可以访问:同一个进程的所有线程共享相同的虚拟地址空间.这意味着一个线程的栈在内存中的地址对于该进程内的其他线程是可见的.因此,如果线程1获得了线程2栈上某个变量的地址,它理论上可以尝试去读写那个地址.

\* 实践中极其危险且不推荐: 1. 破坏封装和独立性:每个线程的栈是其私有执行上下文的核心,存储着函数调用帧、局部变量、返回地址等.直接访问其他线程的栈会破坏这种独立性,使得程序逻辑混乱且难以调试.

2. 数据竞争和状态不确定:线程的栈内容是动态变化的.当线程1试图访问线程2的栈时,线程2可能正在执行函数调用、返回,或者其栈上的数据可能已经被修改或失效.这种访问极易导致读取到脏数据、覆盖关键信息(如返回地址),引发程序崩溃或不可预测的行为.

3. 生命周期问题:栈上的局部变量生命周期与函数调用绑定.如果线程2从一个函数返回,其栈帧被回收,线程1持有的指向该栈帧内变量的指针就会变成悬空指针,解引用它会导致未定义行为.

\* 举例说明: 假设线程2执行一个函数,该函数有一个局部变量 int local\_var = 10;.线程2通过某种(通常是错误设计的)方式将 &local\_var 这个地址传递给了线程1(例如,通过一个共享的全局变量).线程1随后尝试读取或写入 \*(&local\_var)指向的内存.

\* 危险情况1:在线程1访问时,线程2可能已经从该函数返回,local\_var 所在的栈空间可能已被释放或被后续的函数调用覆盖.线程1的访问会操作无效内存,导致崩溃或读取垃圾数据.

\* 危险情况2:即使线程2仍在函数内,线程1的写入操作也可能干扰线程2的正常执行,因为它可能覆盖了线程2期望保持不变的局部变量值或其他栈上的重要数据.

结论:尽管地址空间共享使得理论上可以访问,但从编程实践和安全角度看,一个线程绝对不应该直接访问另一个线程的栈.线程间通信应该通过共享内存(如堆、全局变量)并配合适当的同步机制(如互斥锁、信号量)来进行.

17. 场景1

进程A:100ms；进程B:1ms；进程C:2ms 注:假设时间片大小1ms

请分别计算采用 FCFS,RR,SRTN 三种调度算法时,平均周转时间

假设进程按 A,B,C 顺序在时间 0 到达.

\* FCFS: \* 执行顺序:A -> B -> C

\* 完成时间:A(100),B(101),C(103)

\* 周转时间(TT):TT\_A = 100,TT\_B = 101,TT\_C = 103

\* 平均周转时间 =(100 + 101 + 103)/ 3 = 304 / 3 ≈ 101.33 ms

\* RR(时间片 Q=1ms): \* 执行序列:A(1),B(1),C(1),A(1),C(1),A(98)...(详细:A(0-1),B(1-2),C(2-3),A(3-4),C(4-5),A(5-103))

\* 完成时间:B(2),C(5),A(103)

\* 周转时间(TT):TT\_A = 103,TT\_B = 2,TT\_C = 5

\* 平均周转时间 =(103 + 2 + 5)/ 3 = 110 / 3 ≈ 36.67 ms

\* SRTN: \* t=0:就绪 {A(100),B(1),C(2)}.选择 B(剩余 1).

\* t=1:B 完成(TT\_B=1).就绪 {A(100),C(2)}.选择 C(剩余 2).

\* t=3:C 完成(TT\_C=3).就绪 {A(100)}.选择 A.

\* t=103:A 完成(TT\_A=103).

\* 平均周转时间 =(103 + 1 + 3)/ 3 = 107 / 3 ≈ 35.67 ms

18. 场景2

进程A:10ms；进程B:10ms；进程C:10ms

注:假设时间片大小1ms

请分别计算采用 FCFS,RR,SRTN 三种调度算法时,平均周转时间

假设进程按 A,B,C 顺序在时间 0 到达.

\* FCFS: \* 执行顺序:A -> B -> C

\* 完成时间:A(10),B(20),C(30)

\* 周转时间(TT):TT\_A = 10,TT\_B = 20,TT\_C = 30

\* 平均周转时间 =(10 + 20 + 30)/ 3 = 60 / 3 = 20 ms

\* RR(时间片 Q=1ms): \* 执行序列:A(1),B(1),C(1),A(1),B(1),C(1),...,A(1),B(1),C(1)(共30步)

\* 完成时间:A(28),B(29),C(30)(A 在 0,3,...,27 运行; B 在 1,4,...,28 运行; C 在 2,5,...,29 运行)

\* 周转时间(TT):TT\_A = 28,TT\_B = 29,TT\_C = 30

\* 平均周转时间 =(28 + 29 + 30)/ 3 = 87 / 3 = 29 ms

\* SRTN: \* t=0:就绪 {A(10),B(10),C(10)}.剩余时间相同,按 FCFS 规则选择 A.

\* 由于无新进程到达且运行时间相同,无抢占发生,行为同 FCFS.

\* 执行顺序:A -> B -> C

\* 完成时间:A(10),B(20),C(30)

\* 周转时间(TT):TT\_A = 10,TT\_B = 20,TT\_C = 30

\* 平均周转时间 =(10 + 20 + 30)/ 3 = 60 / 3 = 20 ms

19. 根据对场景1和场景2采用不同调度算法计算的结果,请给出一些点评(或结论).

\* 场景1点评(进程时长差异大): \* FCFS 表现最差(平均周转时间 101.33ms),受到"护航效应"影响,长进程 A 严重阻塞了短进程 B 和 C.

\* RR(36.67ms)和 SRTN(35.67ms)表现远好于 FCFS.它们都有效地让短进程优先执行,显著降低了平均周转时间.SRTN 理论最优,因为它精确地优先执行了最短的剩余任务.

\* 结论:对于进程时长差异大的负载,优先考虑短任务的算法(如 RR,SRTN)能显著改善平均周转时间.

\* 场景2点评(进程时长均匀): \* FCFS 和 SRTN 表现相同且最优(平均周转时间 20ms).因为所有进程长度相同且同时到达,SRTN 实际行为等同于 FCFS.顺序执行使得总完成时间最短.

\* RR 表现最差(29ms).频繁的上下文切换和任务交错执行,反而延长了每个进程的完成时间,导致平均周转时间增加.

\* 结论:对于进程时长均匀且同时到达的负载,简单的 FCFS(或 SRTN)可能是最优的.RR 在这种情况下会因切换开销和任务交错执行而导致性能下降.

\* 综合结论: \* 调度算法的性能高度依赖于工作负载的特性.没有一种算法在所有情况下都是最优的.

\* RR 通常能提供较好的响应时间(虽然本例未计算),适合交互式系统,但可能牺牲平均周转时间.

\* SRTN 平均周转时间最优,但需要预测运行时间且可能导致长任务饥饿.

\* FCFS 简单,但在混合负载下效率低下.

20. 对某系统进行检测后发现,在阻塞 I/O 之前,平均每个进程运行时间为 T .一次进程切换需要的时间为 S ,这里 S 实际上就是开销.对于采用时间片长度为Q的轮转调度,请给出以下各种情况的CPU利用率的计算公式: CPU利用率 = 有效运行时间 /(有效运行时间 + 切换开销时间)

(a)Q = ∞

\* 进程运行 T 后阻塞,然后切换 S.在一个(T + S)周期内,有效运行时间为 T.

\* CPU利用率 = T /(T + S)

(b)Q > T

\* 进程运行 T 后阻塞,时间片 Q 未用尽.情况同(a).

\* CPU利用率 = T /(T + S)

()S < Q < T

\* 进程每次运行 Q 后,需要进行一次切换 S.在一个(Q + S)的调度周期内,CPU有效运行时间为 Q.

\* CPU利用率 = Q /(Q + S)

(d)Q = S

\* 代入()的公式.

\* CPU利用率 = Q /(Q + Q)= Q /(2Q)= 1/2

(e)Q趋近于0

\* 时间片极小,绝大部分时间用于切换.

\* CPU利用率 = Q /(Q + S)→ 0 /(0 + S)= 0(假设 S > 0)

21. 什么是 C10K 问题?

- C10K问题是指服务器同时处理10,000个客户端连接的挑战,由Dan Kegel在1999年提出.

- 在传统的服务器架构下,使用"一个连接一个进程/线程"的模型,当并发连接数达到数千级别时,系统会因为以下原因而性能急剧下降: 1. 进程/线程创建和上下文切换开销巨大

2. 内存占用过高(每个连接需要独立的栈空间)

3. 阻塞I/O导致CPU利用率低

- 解决C10K问题的主要技术方向包括: 1. I/O多路复用:使用select、poll、epoll等系统调用在单线程中处理多个连接

2. 事件驱动架构:基于回调的非阻塞模型

3. 异步I/O:如Linux的AIO、Windows的IOCP

4. 协程:轻量级线程,可以高效处理大量并发连接

22. 请列举支持协程的语言.

- Go(goroutine)

- Python(asyncio,yield)

- Kotlin(coroutines)

- Lua(协同程序)

- JavaScript(async/await,Generator)

- C#(async/await)

- Rust(async/await)

- Ruby(Fiber)

- PHP(Generator,Fiber)

- Dart(async/await)

- Swift(async/await)

- Julia(Task)

23. FCFS、SJF、SRTN 和 RR 中,哪一个(些)是抢占式的?

- 抢占式调度算法: 1. SRTN(Shortest Remaining Time Next):当新进程到达时,如果该进程的预期运行时间比当前运行进程的剩余时间短,则抢占当前进程.

2. RR(Round Robin):基于时间片轮转,进程的CPU时间被限制在一个时间片内,时间片用完后自动被抢占.

- 非抢占式调度算法: 1. FCFS(First-Come,First-Served):按到达顺序执行,进程一旦获得CPU就运行到完成或主动放弃.

2. SJF(Shortest Job First):选择预期执行时间最短的进程运行,但一旦开始执行就不会被抢占.

24. 课上阅读 XV6 源代码,列举出2个引发调度的原因?给出代码的具体文件和行数.

1. 时钟中断触发时间片用尽: - 文件:kernel/trap. 第75-77行

// give up the CPU if this is a timer interrupt.

if(which\_dev == 2)

yield();

此处在usertrap函数中,当检测到类型为2的设备中断(即时钟中断)时,调用yield()函数让出CPU.

时钟中断处理的完整流程: 1. 时钟中断触发:在start.c的timerinit()函数中,通过w\_stimecmp(r\_time()+ 1000000)设置下一次时钟中断,约0.1秒后触发

2. 中断处理:中断发生时,CPU跳转到kernelvec.S中预先设置的中断处理入口点

3. 保存寄存器:kernelvec.S保存所有调用者保存寄存器(caller-saved registers)到内核栈

4. 调用C语言处理函数:跳转到trap.c中的kerneltrap()或usertrap()函数处理中断

5. 判断中断类型:通过devintr()函数确定是否为时钟中断(返回值为2)

6. 时钟中断处理:在trap.c中的devintr()函数内部,如果检测到时钟中断(scause寄存器值为0x8000000000000005L),会调用clockintr()函数

7. 计时器增加:clockintr()函数增加系统ticks计数,唤醒等待计时器的进程,并设置下一次时钟中断

8. 主动让出CPU:回到中断处理主流程后,如果确认是时钟中断(which\_dev == 2),调用yield()函数

2. 进程主动调用sleep等待事件: - 文件:kernel/proc. 第557-559行

// Go to sleep.

p->chan = chan;

p->state = SLEEPING;

sched();

在sleep函数实现中,将进程状态设置为SLEEPING并调用sched()函数进行调度切换.

25. CTSS:Compatible Time-Sharing System 兼容分时系统的历史意义是什么?

- CTSS是由MIT在1961-1963年开发的早期分时操作系统,其历史意义包括: 1. 分时技术的早期实现:是首批成功实现的分时系统之一,证明了多用户并发共享计算机资源的可行性

2. 交互式计算的开创者:打破了批处理系统的局限,允许用户与计算机实时交互

3. 现代操作系统特性的先驱:引入了文件系统、命令行解释器、在线帮助等现代操作系统特性

4. MULTICS的前身:为后续的MULTICS系统奠定了基础,而MULTICS又影响了UNIX的设计

5. 学术贡献:作为MIT的研究项目,培养了一代系统软件专家,产生了重要的学术成果

6. 商业影响:证明了分时系统的商业价值,促进了计算机从专用设备向普通用户开放的转变

26. 公平共享调度(Fair-share scheduling)、保证调度(Guaranteed scheduling)和彩票调度(Lottery scheduling):学习并总结这几种调度算法(原理,优缺点,策略,适用场景)

- 公平共享调度(Fair-share Scheduling): - 原理:根据用户或用户组分配CPU时间,确保每个用户获得预定比例的系统资源,而非只关注单个进程.即使一个用户有多个进程,也只能获得其应得的资源份额.

- 策略: 1. 每个用户(或用户组)被分配一个权重,表示其应获得的系统资源比例

2. 系统跟踪每个用户实际使用的资源量,并调整调度决策以平衡实际使用与目标分配之间的差距

3. 用户实际获得的CPU时间与其权重和活跃进程数相关

- 优点: 1. 防止单一用户通过运行大量进程垄断系统资源

2. 对多用户环境提供更好的隔离性和公平性

3. 适合于共享计算资源的环境(如教育机构、云计算平台)

- 缺点: 1. 实现复杂,需要跟踪用户资源使用历史

2. 可能导致某些关键任务得不到及时响应

- 适用场景:多用户计算环境、云计算平台、共享计算机集群

- 保证调度(Guaranteed Scheduling): - 原理:系统向用户承诺(保证)一定比例的CPU处理能力,并实际履行这一承诺.

- 策略: 1. 系统记录每个进程自创建以来获得的CPU时间

2. 计算每个进程应得的理论CPU时间(基于进程数量和运行时间)

3. 计算实际获得与应得之比(CPU时间比)

4. 优先调度比值最低的进程,确保所有进程获得公平份额

- 优点: 1. 提供可预测的性能保证

2. 确保资源分配的公平性

3. 防止进程被"饿死"

- 缺点: 1. 需要维护详细的进程历史记录(CPU使用时间)

2. 不考虑进程优先级或实时性需求

3. 实现复杂度较高

- 适用场景:需要资源使用保证的环境,如商业服务器、资源预留系统

- 彩票调度(Lottery Scheduling): - 原理:一种概率性、随机化的调度机制,通过给进程分配"彩票"来反映其优先级,定期进行"抽奖"决定下一个运行的进程.

- 策略: 1. 每个进程分配一定数量的彩票(票数反映优先级/权重)

2. 调度器随机抽取一张彩票

3. 持有该彩票的进程获得下一个CPU时间片

- 特性: 1. 彩票转让(Ticket Transfer):进程可临时将彩票转让给其他进程

2. 彩票通胀(Ticket Inflation):在适当情况下,进程可临时增加自己的彩票数量

3. 货币兑换(Currency Exchange):不同用户组可以有自己的"货币体系"

- 优点: 1. 实现简单,决策快速(只需随机数生成和简单计算)

2. 自然支持不同优先级(通过不同彩票数量)

3. 具有良好的统计公平性(长期运行中,进程获得的CPU时间与彩票数成正比)

4. 可以模拟多种调度算法

- 缺点: 1. 短期公平性无法保证(随机性可能导致短期内资源分配不均)

2. 不适合实时系统(无法提供确定性保证)

3. 难以精确控制响应时间

- 适用场景:通用操作系统、教学/模拟环境、希望避免传统优先级调度常见问题的系统

27. 请指出 Windows 的调度算法与多级反馈队列算法的2个相似点和2个不同点.

- 相似点: 1. 多级优先级机制:Windows调度器和多级反馈队列(MLFQ)都基于多级优先级队列实现,将进程/线程分配到不同优先级的队列中,高优先级队列中的任务优先获得CPU执行权.

2. 动态优先级调整:两者都会根据进程/线程的行为动态调整优先级,特别是对I/O密集型任务进行优先级提升,以提高系统响应性.例如,刚完成I/O操作的任务优先级会临时提高,而长时间占用CPU的任务优先级会降低.

- 不同点: 1. 时间片长度处理:Windows中,时间片长度与优先级相关(高优先级任务获得更长的时间片),而传统MLFQ算法中,时间片长度通常随着优先级降低而增加(较低优先级队列拥有更长的时间片).

2. 优先级提升机制:Windows使用复杂的优先级提升策略,如提升等待键盘/鼠标输入的前台应用、基于I/O完成插槽、处理器亲和性等特性；而MLFQ主要依赖单一的老化(aging)机制来防止饿死,通常通过周期性地将所有进程提升到最高优先级来实现.

28. 请给实时操作系统的调度算法选择3个关键词,能够把实时操作系统最主要考虑的问题覆盖到.请简单解释一下每个关键词.

- 截止时间(Deadline):实时系统的核心约束,表示任务必须在特定时间点前完成执行的硬性要求.实时调度算法必须确保任务能在其截止时间前完成,这是衡量实时系统成功与否的首要标准.EDF(最早截止期限优先)等算法直接基于截止时间做出调度决策.

- 可预测性(Predictability):实时系统需要具有确定性行为,即系统响应时间和任务完成时间必须是可预测和稳定的.这要求调度算法本身的执行时间可预测,调度开销恒定,且能提供对最坏情况执行时间的保证,使系统行为在各种条件下都可分析和验证.

- 优先级抢占(Priority Preemption):确保高优先级(通常是更紧急或更关键)的任务能立即获得系统资源的机制.在实时系统中,当高优先级任务就绪时,必须能立即抢占低优先级任务的执行,这是保证时间关键任务(尤其是硬实时任务)能及时响应的基础机制.

29. 请给出以下名词的对应英文:地址重定位(地址转换,地址映射,地址变换,地址翻译)；存储体系；交换技术；地址空间

- 地址重定位(地址转换,地址映射,地址变换,地址翻译):Address Relocation(Address Translation,Address Mapping,Address Transformation)

- 存储体系:Memory Hierarchy

- 交换技术:Swapping

- 地址空间:Address Space

30. 何时将指令、数据绑定到物理内存地址?

将指令和数据绑定到物理内存地址的时机有三种: - 编译时绑定(Compile-time Binding):如果编译时已知程序将驻留在内存中的具体位置,编译器可以生成包含绝对地址的代码.若加载位置发生变化,程序必须重新编译.这种方式主要用于嵌入式系统或单一程序运行的简单系统.

- 加载时绑定(Load-time Binding):如果编译时不知道程序将驻留在内存哪个位置,编译器生成可重定位代码(包含相对地址).加载程序在将程序装入内存时,根据实际的起始地址,将所有相对地址一次性转换为绝对地址.这是静态地址重定位的一种形式.

- 执行时绑定(Execution-time Binding):地址绑定推迟到程序执行时进行.每次CPU访问内存时,逻辑地址都被硬件(通常是MMU)动态转换为物理地址.这种动态地址重定位需要专门的硬件支持,但提供了最大的灵活性,允许程序在执行过程中在内存中移动,是现代操作系统常用的方式.

31. 进程的什么部分需要交换到磁盘?

在虚拟内存管理中,进程的以下部分可能需要交换到磁盘: - 整个进程:在传统的交换(Swapping)技术中,操作系统可能将整个进程从内存交换到磁盘,以便腾出空间给其他进程.

- 进程的部分页面:在分页式虚拟内存系统中,通常只交换进程的部分页面,包括: 1. 代码段(Text Segment):包含程序的执行指令.由于代码段通常是只读的,如果它来自可执行文件,被修改的页面可以直接丢弃而不是写回磁盘.

2. 数据段(Data Segment):包含静态分配的变量.

3. 堆(Heap):动态分配的内存区域.

4. 栈(Stack):包含函数调用信息、局部变量等.

- 不需要交换的部分: 1. 已映射到文件的只读代码页:这些可以在需要时从原始可执行文件重新加载,无需保存到交换空间.

2. 内核空间的关键部分:如中断处理例程等,通常会被锁定在物理内存中.

3. 处于活跃I/O操作中的缓冲区:这些区域通常被锁定,防止在I/O期间被交换出去.

32. 在磁盘的什么位置保存被换出的进程?换出后再换入的进程是否回到原处?

- 存储位置: - 交换分区(Swap Partition):许多操作系统(如Linux)使用专门的磁盘分区作为交换空间,这是一个独立于文件系统的连续磁盘区域,专门用于页面交换.

- 交换文件(Swap File):一些操作系统(如Windows)在常规文件系统中使用一个或多个特殊文件(如pagefile.sys)作为交换空间.

- 内存映射文件:对于代码段和只读数据,通常可以直接从原始可执行文件或共享库文件重新加载,而不需要专门的交换空间.

- 是否回到原处: - 整个进程交换:在传统的进程级交换中,进程通常不一定回到原来的物理内存位置.操作系统会根据当前可用的内存情况,将进程加载到任何足够大的可用内存区域.

- 页面交换:在现代分页式虚拟内存系统中: 1. 虚拟地址不变:从进程的角度看,它的虚拟地址空间保持不变.

2. 物理页框可变:同一个虚拟页面在被换出后再换入时,可能会被分配到不同的物理页框中.

3. 页表更新:操作系统会更新页表,将虚拟页面映射到新的物理页框.

这种灵活性是虚拟内存系统的关键优势之一,允许操作系统更有效地管理物理内存资源,而对应用程序透明.

33. 为什么不应换出处于等待I/O状态的进程?

不应换出处于等待I/O状态的进程,主要原因有: 1. I/O完成时间不可预测:I/O操作(特别是磁盘和网络I/O)可能随时完成.如果进程已被换出,I/O完成时系统需要处理中断,但找不到对应的内存缓冲区,会导致严重问题.

2. 增加响应延迟:当I/O完成时,如果相关进程已被换出,必须先将其换回内存才能处理I/O完成事件,这会导致显著的响应延迟.

3. 双重磁盘操作:将等待I/O的进程换出到磁盘,然后在I/O完成后又立即需要从磁盘换回内存,形成低效的磁盘"乒乓"操作(disk thrashing).

4. DMA缓冲区问题:许多I/O操作使用DMA(直接内存访问)技术,需要物理内存缓冲区在整个操作过程中保持固定位置.如果包含这些缓冲区的内存页被换出,会破坏正在进行的DMA传输.

5. 资源效率考量:等待I/O的进程通常不消耗CPU资源,只占用少量内存.与其执行代价高昂的换出操作,不如让它们留在内存中等待I/O完成.

因此,现代操作系统通常会锁定(pin)包含活跃I/O缓冲区的内存页,防止它们被换出.一些系统甚至会为等待I/O的进程提供优先级提升,确保它们保留在内存中.

34. David Wheeler在参加 EDSAC 研制过程中有哪些贡献?

David Wheeler(1927-2004)作为剑桥大学的学生和研究人员,在EDSAC(Electronic Delay Storage Automatic Calculator,电子延迟存储自动计算机)的研制过程中做出了几项重要贡献: 1. 发明了子程序概念和"Wheeler跳转":Wheeler最著名的贡献是发明了封闭子程序(closed subroutine)的概念,这是现代编程中函数/方法的前身.他设计了一种巧妙的跳转机制(后来被称为"Wheeler Jump"),允许代码从程序的任何部分跳转到子程序,执行完后再返回到原来的位置.这被认为是软件历史上的重大突破.

2. 开发了第一个汇编程序:Wheeler设计了EDSAC的"initial orders"(初始命令集),这实际上是第一个汇编程序或加载器的早期形式,极大简化了编程工作.程序员可以用符号形式而非机器码编写程序.

3. 编写了EDSAC的早期库例程:Wheeler编写了许多实用的库例程,包括数学函数、输入输出处理等,为EDSAC用户提供了基础功能支持.

4. 参与了EDSAC的硬件设计与调试:作为Maurice Wilkes领导的团队成员,Wheeler参与了EDSAC的硬件设计和调试工作.

5. 取得第一个计算机科学博士学位:Wheeler在1951年因其在EDSAC上的工作获得了剑桥大学博士学位,被认为是世界上第一个计算机科学博士.

35. RISCV 可以支持设计多大的虚拟地址空间?

RISC-V架构设计有可扩展的地址空间大小,通过不同的地址位宽支持多种虚拟地址空间: - RV32:32位地址空间,支持最大4GB(2³²字节)的虚拟地址空间.

- RV64:64位地址空间,理论上支持2⁶⁴字节(18.4 EB,即18.4×10¹⁸字节)的虚拟地址空间.

但在实际实现中,RISC-V规范目前限制只使用低48位作为有效地址位,提供256TB(2⁴⁸字节)的虚拟地址空间.这是因为: 1. 当前应用几乎不需要如此大的地址空间

2. 完整64位地址转换会需要过多级的页表,降低效率

3. 留出高位便于未来扩展(将来可以扩展到更多位)

- RV128:RISC-V规范也为未来保留了128位地址空间的可能性,虽然目前没有实际实现.

RISC-V的这种可扩展性是该架构的一个重要特点,允许从简单嵌入式系统到高性能计算机使用同一指令集架构.

36. RISCV 的页表项占多少字节?主要包括什么内容?

RISC-V的页表项(PTE)大小取决于所使用的虚拟内存方案: - Sv32(32位系统):PTE占用4字节(32位)

- Sv39/Sv48/Sv57(64位系统):PTE占用8字节(64位)

在最常用的64位系统中,一个页表项主要包含以下字段: | 位域 | 63-54 | 53-10 | 9-8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

|-|-|-|-|-|-|-|-|-|-|-|-|

| 含义 | Reserved | PPN | RSW | D | A | G | U | X | W | R | V |

| 描述 | 保留位 | 物理页号 | 软件保留 | 脏位 | 访问位 | 全局位 | 用户位 | 执行权限 | 写权限 | 读权限 | 有效位 |

- V(Valid,位0):有效位,表示PTE是否可用于地址转换

- R(Read,位1):读权限位,允许读取该页

- W(Write,位2):写权限位,允许写入该页

- X(Execute,位3):执行权限位,允许从该页执行指令

- U(User,位4):用户模式访问位,控制用户模式是否可访问该页

- G(Global,位5):全局映射位,表示在所有地址空间中都有效

- A(Accessed,位6):访问位,页面被访问时由硬件设置,用于页面置换算法

- D(Dirty,位7):脏位,页面被写入时由硬件设置,用于确定是否需要写回存储设备

- RSW(位8-9):保留给软件使用的位,硬件不使用

- PPN(Physical Page Number,位10-53):物理页号,存储物理内存页框的基地址

- Reserved(位54-63):保留位,供未来扩展使用

在多级页表结构中,PTE有两种形态: - 叶子PTE:指向最终物理页框,R/W/X至少有一位为1

- 指针PTE:指向下一级页表,R=W=X=0且V=1,PPN指向下一级页表的物理基地址

37. RISCV 可以支持几级页表?

RISC-V架构支持多种虚拟内存方案,不同的方案对应不同级数的页表: - Sv32(用于32位系统):支持2级页表.虚拟地址被划分为两级页表索引和页内偏移.

- Sv39(用于64位系统):支持3级页表.使用 39 位虚拟地址,划分为三级页表索引和页内偏移.

- Sv48(用于64位系统):支持4级页表.使用 48 位虚拟地址,划分为四级页表索引和页内偏移.

- Sv57(用于64位系统):支持5级页表.使用 57 位虚拟地址,划分为五级页表索引和页内偏移.

因此,RISC-V可以根据具体实现选择支持2级、3级、4级或5级页表.

38. RISCV 中页表的起始地址放在哪里?X86 呢?

RISC-V中的页表起始地址: 在RISC-V架构中,页表的起始地址存放在satp寄存器中(Supervisor Address Translation and Protection Register,监管者地址转换和保护寄存器).这是一个特权级控制状态寄存器(CSR),只能由操作系统内核访问.

satp寄存器的格式: - RV32: | 位域 | 31-22 | 21-0 |

| - | - | - |

| 含义 | MODE | PPN |

- MODE:地址转换模式(0=禁用分页,1=Sv32)

- PPN:根页表物理页号

- RV64: | 位域 | 63-60 | 59-44 | 43-0 |

| - | - | - | - |

| 含义 | MODE | ASID | PPN |

- MODE:地址转换模式(0=禁用分页,8=Sv39,9=Sv48,10=Sv57)

- ASID:地址空间标识符(用于TLB管理)

- PPN:根页表物理页号

X86中的页表起始地址: 在x86架构中,页表的起始地址存放在CR3寄存器中(Control Register 3,控制寄存器3),也被称为PDBR(Page Directory Base Register,页目录基址寄存器).

CR3寄存器格式(64位模式): | 位域 | 63-52 | 51-32 | 31-12 | 11-5 | 4-3 | 2-0 |

| - | - | - | - | - | - | - |

| 含义 | 保留/扩展 | PCD | 页表基址 | 保留 | PWT | 保留 |

- 页表基址:指向最高级页表(通常是PML4或页目录指针表)的物理地址

- PCD:Page-level Cache Disable位

- PWT:Page-level Write-Through位

比较: 两种架构类似之处在于都使用专用寄存器存储页表起始地址.主要区别在于: 1. RISC-V的satp寄存器还包含了地址转换模式选择(MODE)和ASID字段

2. x86的CR3包含了缓存控制位(PCD,PWT)

3. RISC-V的设计更加简洁和模块化,符合其精简指令集的设计理念

39. 对应页表项,给出产生 Page Fault 的具体原因?

根据页表项(PTE)状态的不同,可能由以下具体原因产生: 1. 无效页(Invalid Page): - PTE的有效位(V/Present bit)为0:表示该虚拟页面未映射到任何物理页框.这可能是因为: \* 该页面从未被分配

\* 该页面已被交换出到磁盘(被换出页)

\* 该虚拟地址超出了进程的地址空间范围

2. 权限违规(Permission Violation): - 访问权限不足:当PTE有效,但操作不被允许时,例如: \* 写入只读页面(PTE的W位为0,但进程尝试写入)

\* 执行不可执行页面(PTE的X位为0,但进程尝试执行代码)

\* 用户模式访问内核页面(PTE的U/User位为0,但用户模式进程尝试访问)

3. 页表层级不完整(Page Table Level Incomplete): - 多级页表中某级页表不存在(PTE指向的下一级页表无效)

4. 写时复制(Copy-on-Write): - 进程尝试写入标记为写时复制的页面(通常在fork()后)

- 这是一种特殊类型的权限违规,操作系统需要复制页面而不是简单地拒绝访问

5. 访问位和脏位设置(A/D Bit Handling): - 某些系统会通过故意使页表项无效,在访问时触发页错误,然后由操作系统处理A位和D位的设置

6. 硬件保护机制: - 某些架构实现了额外的内存保护机制,如: \* 边界检查违规

\* 对齐要求违反

\* 执行保护(NX/XD位):防止执行数据页中的代码

7. TLB不一致(TLB Inconsistency): - 页表更新后TLB未刷新,导致TLB和页表信息不一致

不同类型的页错误处理方式也不同: - 对于被换出的页面,操作系统会将其从磁盘加载回内存

- 对于权限违规,通常会向进程发送信号(如SIGSEGV)

- 对于写时复制,操作系统会创建页面的私有副本

40. PCID 和 ASID 的工作原理是什么?

PCID(Process Context ID)和ASID(Address Space ID)是两种用于优化TLB(Translation Lookaside Buffer)管理的技术,它们解决的问题相似,但实现于不同的处理器架构.

核心问题: 在传统的虚拟内存系统中,每当进程切换导致地址空间变化时,必须刷新TLB缓存,因为不同进程的相同虚拟地址可能映射到不同的物理地址.这种TLB刷新操作非常昂贵,会显著降低系统性能,特别是在多任务环境下.

工作原理: 1. PCID(Process Context Identifier):在x86-64架构中使用

- 为每个进程分配一个唯一的标识符(PCID),最多12位

- TLB条目被标记上当前进程的PCID

- 地址转换时,只有PCID匹配的TLB条目才会被使用

- 进程切换时,如果启用PCID,操作系统可以: \* 更改CR3寄存器加载新进程的页表,同时设置新的PCID

\* 不必刷新整个TLB,因为新旧进程的TLB条目可以通过PCID区分

- 使用控制寄存器位(CR4.PCIDE)启用PCID功能

2. ASID(Address Space Identifier):在ARM和RISC-V等架构中使用

- 基本概念与PCID类似,为不同地址空间分配唯一标识符

- 在RISC-V中,ASID存储在satp寄存器中

- ARM中ASID位宽从8位到16位不等,根据实现而定

- RISC-V中通常为16位

- 每个TLB条目都带有ASID标记

- 某些TLB条目可以标记为全局(global),这些条目在所有地址空间中共享,不受ASID影响

主要优势: 3. 避免不必要的TLB刷新:进程切换时不再需要完全刷新TLB

4. 提高上下文切换性能:特别是在多任务负载下,可显著减少TLB未命中率

5. 支持每进程TLB管理:允许操作系统针对特定进程有选择地刷新TLB条目

实现差异: - PCID(x86-64)通常需要显式标记要保留的TLB条目

- ASID(ARM/RISC-V)对软件更加透明,设计更加一致

- 两者在位宽、控制方式和具体细节上各有不同,但核心思想一致

这些技术对于现代操作系统性能至关重要,尤其是在频繁进程切换的工作负载下,可以减少地址转换开销,提高系统吞吐量.

41. 解释一下 global/non-global TLB

Global TLB(全局TLB): - 全局TLB条目在所有进程的地址空间中共享

- 不受进程上下文切换影响,切换进程时无需刷新

- 通常用于映射操作系统内核空间、共享库等全局资源

- 在页表项中通常有特殊的标志位(如G位)来标记全局页面

- 全局TLB条目不与ASID/PCID关联,对所有进程都有效

Non-global TLB(非全局TLB): - 非全局TLB条目特定于某个进程或地址空间

- 通常与ASID或PCID关联,用于区分不同进程的相同虚拟地址

- 映射用户空间中的私有内存区域

- 在进程切换时,如果新进程与旧进程的ASID/PCID不同,这些条目要么被刷新,要么被视为无效

优势与用途: - 结合使用全局和非全局TLB可以提高TLB利用效率

- 减少进程切换时的TLB刷新开销

- 特别适合频繁访问共享代码和数据的系统

42. 系统给某进程分配3个页框(固定分配策略),初始为空. 进程执行时,页面访问顺序为:2 3 2 1 5 2 4 5 3 2 5 2 请分别给出应用 FIFO、LRU 、OPT 页面置换算法时的缺页次数.

\* FIFO 算法过程: | 访问页面 | 2 | 3 | 2 | 1 | 5 | 2 | 4 | 5 | 3 | 2 | 5 | 2 |

| - | - | - | - | - | - | - | - | - | - | - | - | - |

| 页框1 | 2 | 2 | 2 | 2 | 5 | 5 | 5 | 5 | 3 | 3 | 3 | 3 |

| 页框2 | - | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 5 | 5 |

| 页框3 | - | - | - | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 4 | 2 |

| 缺页 | √ | √ | × | √ | √ | √ | √ | × | √ | × | √ | √ |

总计9次缺页

\* LRU 算法过程: | 访问页面 | 2 | 3 | 2 | 1 | 5 | 2 | 4 | 5 | 3 | 2 | 5 | 2 |

| - | - | - | - | - | - | - | - | - | - | - | - | - |

| 页框1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |

| 页框2 | - | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |

| 页框3 | - | - | - | 1 | 1 | 1 | 4 | 4 | 4 | 2 | 2 | 2 |

| 缺页 | √ | √ | × | √ | √ | × | √ | × | √ | √ | × | × |

总计7次缺页

\* OPT 算法过程: | 访问页面 | 2 | 3 | 2 | 1 | 5 | 2 | 4 | 5 | 3 | 2 | 5 | 2 |

| - | - | - | - | - | - | - | - | - | - | - | - | - |

| 页框1 | 2 | 2 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 2 | 2 | 2 |

| 页框2 | - | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

| 页框3 | - | - | - | 1 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |

| 缺页 | √ | √ | × | √ | √ | × | √ | × | × | √ | × | × |

总计6次缺页,可以发现重点是第五次访问要把 1 换出去,因为我们知道未来信息 1 不再被用到了

43. 系统给某进程分配 m 个页框,初始为空.页面访问顺序为 1 2 3 4 1 2 5 1 2 3 4 5 采用 FIFO 算法,请分别计算当 m=3 和 m=4 时的缺页异常次数.

\* m=3 时,缺页过程: | 访问页面 | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |

| - | - | - | - | - | - | - | - | - | - | - | - | - |

| 页框1 | 1 | 1 | 1 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |

| 页框2 | - | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 3 | 3 |

| 页框3 | - | - | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 4 | 4 |

| 缺页 | √ | √ | √ | √ | √ | √ | √ | × | × | √ | √ | × |

总计9次缺页

\* m=4 时,缺页过程: | 访问页面 | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |

| - | - | - | - | - | - | - | - | - | - | - | - | - |

| 页框1 | 1 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 4 | 4 |

| 页框2 | - | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 5 |

| 页框3 | - | - | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 |

| 页框4 | - | - | - | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 |

| 缺页 | √ | √ | √ | √ | × | × | √ | √ | √ | √ | √ | √ |

总计10次缺页

44. 系统给某进程分配了一个页框,页面大小4K；有矩阵 A[1024][1024] 按行存放.以下一段代码执行时的缺页异常次数是多少?

程序编制方法1: for(j = 0; j < 1024; j++)

for(i = 0; i < 1024; i++)

A[i][j] = 0;

空间局部性好,每次访问都在同一页或下一页,缺页少( 1024 次,每行开始时缺页).

45. 系统给某进程分配了一个页框,页面大小4K；有矩阵 A[1024][1024] 按行存放.以下一段代码执行时的缺页异常次数是多少?

程序编制方法2: for(i=0; i<1024; i++)

for(j=0; j<1024; j++)

A[i][j] = 0;

空间局部性差,每次访问 A[i][j] 和 A[i+1][j] 会跨越多个页面(1024\*4 bytes ≈ 1 page),导致大量缺页(1024 \* 1024 次).

46. 阅读 OSTEP 23章 13页,回答问题:Page Cache 的工作原理,它起到什么作用?

\* 工作原理:Page Cache(页面缓存)是操作系统内核维护的一块物理内存区域,用于缓存最近从磁盘读取的文件数据以及文件系统的元数据.当进程需要读取文件时,操作系统首先检查请求的数据块是否已在页面缓存中.如果在,直接从内存返回,避免了昂贵的磁盘I/O.如果不在,则从磁盘读取数据块到页面缓存,然后再拷贝给进程.写操作通常先写入页面缓存(写回缓存策略),然后由操作系统择机异步写回磁盘.

\* 作用:主要作用是加速文件访问.通过将常用的文件数据保存在内存中,显著减少磁盘I/O操作,提高文件读写性能和系统整体响应速度.它统一了文件数据和内存管理,使得文件数据可以像普通内存页面一样被管理(例如参与页面置换).

47. 简要概括 2Q 置换算法的思想,并与 LRU 进行比较.

\* 思想:2Q 算法旨在结合 LRU 的优点(利用局部性原理)并克服其缺点(容易被一次性大范围扫描污染).它维护两个队列:A1(FIFO 队列)和 Am(LRU 队列).

1. 新页面首次访问时放入 A1 队列尾部.

2. 如果 A1 中的页面被再次访问,则将其移动到 Am 队列.

3. Am 队列按照 LRU 规则管理.

4. 需要置换页面时,优先从 A1 队列头部选择(淘汰最先进入且未被再次访问的).如果 A1 为空,则从 Am 队列的尾部选择(淘汰最近最少使用的).

\* 与 LRU 比较: \* 相似点:两者都试图利用访问的时间局部性原理.2Q 中的 Am 队列本质上是 LRU 管理的.

\* 不同点: \* 抗扫描能力:LRU 对顺序扫描(访问大量不同页面一次)非常敏感,会导致 Page Cache 缓存中所有有用的页面被换出.2Q 通过 A1 队列过滤掉了这种一次性访问,只有被访问至少两次的页面才有机会进入 Am 队列长期驻留,因此抗扫描能力更强.

\* 复杂度:2Q 实现比纯粹的 LRU 更复杂,需要维护两个队列和额外的逻辑.

48. VMS 的内核是如何在进程之间共享的?

VMS(Virtual Memory System)操作系统通过将内核代码和大部分内核数据结构映射到每个进程虚拟地址空间的高地址部分来实现共享.这个共享区域对于所有进程都是相同的物理内存映射.这意味着: 1. 单一内核副本:内存中只需要存放一份内核代码和共享数据.

2. 高效切换:当进程从用户态切换到内核态(例如系统调用)时,不需要切换地址空间或页表,因为内核已经在当前进程的地址空间内,可以直接访问内核代码和数据.这降低了上下文切换的开销.

3. 保护:通过硬件内存保护机制,用户模式下的进程无法直接访问或修改映射在地址空间中的内核区域.

49. VMS 有哪两种减轻页表增长的压力的方式?

VMS 采用以下两种主要方式来管理大型地址空间并减轻线性页表带来的巨大内存开销: 1. 分段(Segmentation)与分页(Paging)结合:VMS 将虚拟地址空间划分为几个大的段(Segment),例如 P0 用于用户程序和堆,P1 用于用户栈,S0 用于内核代码和数据等.每个段内部再进行分页管理.不是为整个巨大的虚拟地址空间维护一个单一的、连续的页表,而是为每个活动的段维护独立的页表.对于未使用或稀疏使用的段,可以不分配页表,从而节省内存.

2. 两级页表结构:在段内部分页时,VMS 采用了类似现代操作系统的多级页表结构(虽然具体实现可能与x86/RISC-V不同),例如系统空间(S0,S1)使用一级线性页表,而进程空间(P0,P1)使用两级页表结构.多级页表允许按需分配页表页,只有在虚拟地址范围被实际使用时才创建相应的二级页表页,避免了为整个虚拟地址空间预先分配所有页表项.

50. 阅读 OSTEP 23.2节11～12页内容,简要概括大页模式解决了什么问题,又有什么代价?

\* 解决的问题: 1. 减少 TLB 未命中(Reduce TLB Misses):标准页面(如 4KB)较小,对于需要大量内存的应用程序,即使工作集不大,也可能跨越很多页面,导致 TLB 很快被填满并频繁发生未命中.大页(如 2MB 或 1GB)可以用一个 TLB 条目覆盖更大的内存范围,显著增加了 TLB 的覆盖率,减少了地址转换开销,提高了性能,尤其对于内存密集型应用.

2. 减少页表开销(Reduce Page Table Overhead):使用大页可以减少页表本身的层级或大小.例如,一个 1GB 的大页只需要一个页表项来映射,而用 4KB 页面则需要 262,144 个页表项和可能的多级页表结构,大页显著降低了页表占用的内存.

\* 代价: 1. 内部碎片(Internal Fragmentation):大页的主要缺点是可能导致更严重的内部碎片.如果应用程序只需要大页中的一小部分,剩余的大部分内存就被浪费了,因为页是内存分配的最小单位.

2. 分配灵活性降低(Reduced Allocation Flexibility):操作系统需要找到连续的、对齐的大块物理内存来支持大页,这比分配小的 4KB 页框更困难,尤其是在内存碎片化时.

3. 页面换出的复杂性(Complexity in Swapping):将一个非常大的页面换出到磁盘或从磁盘换回需要更长的时间和更复杂的管理.因此,大页通常被配置为不可换出(locked in memory).

51. Linux中 fork()执行时与 mmap()有关系吗?

\* 有关系.fork()创建子进程时会复制父进程的地址空间,这包括由 mmap()创建的所有内存映射区域(文件映射、匿名映射等).

\* 默认情况下,这些映射在父子进程间是共享的,并且通常采用写时复制(Copy-on-Write,CoW)机制.

\* 这意味着 fork()之后,父子进程共享相同的物理页面.只有当其中一个进程尝试写入这些共享页面时,内核才会为写入方创建该页面的私有副本.

\* 如果 mmap()创建的是共享文件映射(MAP\_SHARED),则父子进程对映射区域的修改会相互可见,并且会写回底层文件,此时不使用CoW.如果是私有文件映射(MAP\_PRIVATE)或匿名映射,则应用 CoW.

\* 因此,fork()的行为直接受到父进程中存在的 mmap()映射的影响.

52. Linux中执行 fork()时是怎么设置 CoW 的?

1. 复制页表:fork()时,内核为子进程创建父进程页表的副本.

2. 共享物理页:关键在于,内核并不立即复制物理内存页.相反,父子进程的页表项(PTE)最初都指向相同的物理页框.

3. 标记为只读:为了触发 CoW,内核将父子进程中指向这些共享物理页的 PTE 都标记为只读(清除 Write 权限位).

4. 记录 CoW 状态:内核还会更新相关的内存区域描述符(vm\_area\_struct)来指示这些区域是 CoW 区域.

5. 触发页错误:当父进程或子进程尝试写入某个共享的、只读的页面时,会触发一个页错误(Page Fault)异常.

6. 处理页错误:内核的页错误处理程序会检查到这是一个对 CoW 页面的写操作.

7. 分配与复制:内核为进行写入操作的进程分配一个新的物理页面,并将原始共享页面的内容复制到这个新页面.

8. 更新页表:内核更新写入进程的 PTE,使其指向这个新的物理页面,并将该 PTE 标记为可写.

9. 更新引用计数:同时,内核会减少原始物理页面的引用计数.如果引用计数降为 1,那么另一个进程的对应 PTE 也可以被标记为可写(因为它现在是唯一的拥有者).

53. XV6 中 uvmcopy()的作用?uvmcopy 函数中如何设置 CoW 标志的?

\* uvmcopy()的作用: 该函数用于在 fork()系统调用时,复制父进程的用户地址空间到子进程.它遍历父进程(old 页表)指定大小(sz)的用户空间虚拟地址.对于每个有效的父进程页面: 1. 查找对应的页表项(PTE).

2. 获取该页面的物理地址(pa)和权限标志(flags).

3. 分配一个新的物理页框(mem = kalloc()).

4. 将父进程物理页面(pa)的内容复制到新分配的物理页框(mem)中.

5. 使用 mappages()将这个新的、包含副本数据的物理页框(mem)映射到子进程的页表(new)中相应的虚拟地址,并继承父进程页面的权限标志(flags).

简而言之,uvmcopy 为子进程创建了父进程用户内存的一个完整物理副本.

\* 如何设置 CoW 标志: 在 xv6-riscv/kernel/vm. 这个版本的 XV6 实现中,uvmcopy 函数并没有实现写时复制(Copy-on-Write,CoW)机制.

\* 它没有让父子进程共享物理页面.

\* 它没有清除父进程或子进程页表项中的 PTE\_W(写权限)标志位.

\* 它没有使用或设置任何特定的标志来表示页面是 CoW 状态.

因此,在这个版本的 XV6 中,uvmcopy 不设置 CoW 标志,因为它执行的是完全的内存复制,而非 CoW 策略.实现 CoW 需要修改 fork()的逻辑,让父子进程 PTE 指向相同的物理页并清除写权限位,而不是调用这个 uvmcopy.

54. 若 fork()之后调用 exec()替换地址空间内容时,会发生什么事件?处理该事件时要检查哪些标志?

\* 事件:exec()系统调用会加载一个新的程序到当前进程的内存空间,完全替换掉 fork()之后继承自父进程的内存映像(代码、数据、堆栈等).

\* 过程: 1. 释放旧地址空间:内核首先会解除当前进程用户空间的大部分内存映射.这包括:遍历 VMA(虚拟内存区域),取消页面映射,减少 CoW 页面的引用计数(如果计数为零则释放物理页面),释放页表等.

2. 加载新程序:内核读取 exec()指定的可执行文件(例如 ELF 格式).

3. 解析文件头:分析文件头,确定程序的代码段、数据段、BSS 段的大小、位置以及程序的入口点.

4. 建立新地址空间:为新程序创建一套全新的页表和 VMA 结构.

5. 映射新内容:根据可执行文件的信息,将新程序的代码段、数据段等映射到新的地址空间.代码段和只读数据段通常是按需从文件惰性加载(Demand Paging).可写数据段可能需要复制或清零.

6. 创建新堆栈:分配内存并设置新的用户堆栈.

7. 设置启动参数:将命令行参数和环境变量复制到新堆栈上.

8. 重置状态:重置信号处理程序,关闭标记为 close-on-exec 的文件描述符.

9. 设置入口点:修改进程的程序计数器(PC)指向新程序的入口点,并设置好堆栈指针(SP).

\* 检查的标志/信息: 1. close-on-exec 标志:对于进程打开的每个文件描述符,检查此标志.如果设置,则在 exec()成功后关闭该文件描述符.

2. 可执行文件格式(如 ELF 头):内核需要解析它来理解如何加载和映射新程序.

3. 旧进程的 VMA 和 PTE:在释放旧地址空间时,内核检查这些结构,特别是 PTE 中的写权限位、可能的 CoW 状态以及物理页引用计数,以正确地回收资源.

4. 新程序的段权限:从可执行文件获取代码段、数据段等的读/写/执行权限,并在创建新的 PTE 时设置相应的权限位(PTE\_R,PTE\_W,PTE\_X,PTE\_U).

55. 课件134的图 页框数据库实现与应用 ,每一个箭头表示的场景/事件是什么?

1. 从 空闲页面链表 指向 进程的工作集: \* 事件:分配新物理页框.当进程需要一个新的物理内存页(例如,堆栈增长、堆分配、加载数据)并且无法通过其他方式(如软缺页)满足时,操作系统会从 空闲页面链表 中取出一个页框分配给该进程,并将其加入进程的 工作集.

2. 从 零初始化页面链表 指向 进程的工作集(由 需求零缺页异常 触发): \* 事件:分配零初始化页面.当进程发生 需求零缺页异常(Demand Zero Page Fault),即需要一个内容保证为全零的新页面时(例如,分配 BSS 段内存或某些匿名映射),操作系统优先从 零初始化页面链表 中取出一个预先清零的页框分配给进程.

3. 从 后备页面链表 指向 进程的工作集(由 "软"缺页异常 触发): \* 事件:软缺页异常处理(页面回收).当进程访问一个最近被从其 工作集 中移除但尚未被重新分配的页面(该页面位于 后备页面链表)时,会发生 "软"缺页异常(Soft Page Fault).操作系统可以快速地将该页面重新移回进程的 工作集,无需从磁盘读取,成本很低.

4. 从 进程的工作集 指向 修改页面链表(标有 工作集置换): \* 事件:置换脏页面.当发生 工作集置换(Working Set Replacement,通常由内存压力或工作集管理策略触发)时,如果被选中的牺牲页面是已被修改过的脏页面,则它会被移动到 修改页面链表.这些页面必须先被写回磁盘才能被重用.

5. 从 进程的工作集 指向 后备页面链表(标有 工作集置换): \* 事件:置换干净页面.当发生 工作集置换 时,如果被选中的牺牲页面是未被修改过的干净页面,则它可以被直接移动到 后备页面链表.因为其内容与磁盘副本一致,无需写回,可以快速被回收或重用.

6. 从 修改页面链表 经过 已修改页面写出线程 指向 后备页面链表: \* 事件:脏页面写回完成.后台的 已修改页面写出线程(Modified Page Writer Thread)会扫描 修改页面链表,将脏页的内容异步写回磁盘.一旦写操作完成,该页面就变成了干净页面,并被移动到 后备页面链表,准备被重用.

7. 从 后备页面链表 指向 空闲页面链表: \* 事件:重用后备页面.如果 后备页面链表 中的页面在一段时间内没有被任何进程通过软缺页回收,操作系统会认为它不再活跃,将其移动到 空闲页面链表,使其可用于满足新的页面分配请求.此时页面与原进程的关联断开.

8. 从 空闲页面链表 经过 零初始化页面线程 指向 零初始化页面链表: \* 事件:后台页面清零.一个低优先级的内核线程(零初始化页面线程)会在系统空闲时从 空闲页面链表 中取出页面,将其内容填充为零,然后放入 零初始化页面链表.这样可以加快后续 需求零缺页异常 的处理速度.

9. 从 修改页面链表 指向 进程的工作集: \* 事件:脏页面重新激活.当进程再次访问一个已被移至 修改页面链表 但尚未被写回磁盘的页面时,操作系统可以直接将该页面移回 进程的工作集,避免了不必要的磁盘写入操作.这是一种优化,可以在页面仍然被频繁访问的情况下减少 I/O 开销.

10. 从 进程的工作集 指向 后备页面链表: \* 事件:页面老化或主动释放.除了工作集置换外,还有其他情况会导致页面从工作集移动到后备页面链表: 1. 页面老化:某些操作系统会定期扫描进程的工作集,将一段时间内未被访问的页面(通过检查访问位)移至后备页面链表,即使没有内存压力.

2. 进程主动释放:当进程通过系统调用(如 madvise(MADV\_DONTNEED)或 munmap())显式释放某些内存区域时,相应的干净页面会被移动到后备页面链表.

3. 进程挂起:当进程被挂起(swapped out)时,其工作集中的干净页面可能被移动到后备页面链表,以便在进程恢复运行时能够快速恢复.

# 作业 1

## 中断响应流程

问题:复习课件 2-中断异常机制,回答中断响应流程部分的练习题:理解 Linux 的中断处理流程,解释为什么引入上半部和下半部处理.

回答:在 Linux 系统中,中断处理程序应该尽量短且快,以减少对正常进程调度的影响.然而,中断处理程序可能会暂时关闭中断,如果执行时间过长,可能会丢失其他设备的中断请求.为了解决这个问题,Linux 将中断过程分为上半部和下半部.

上半部用于快速处理中断,通常会暂时关闭中断请求,主要负责处理与硬件紧密相关或时间敏感的任务.下半部用于延迟处理上半部未完成的工作,一般以内核线程的方式运行.

上半部(Top Half):- 上半部是中断处理程序的第一部分,直接由硬件中断触发.

- 其主要任务是快速响应中断,处理与硬件紧密相关或时间敏感的操作.

- 上半部运行在中断上下文中,通常会暂时关闭中断,不能被阻塞,也不能进行复杂的操作.

- 典型的上半部操作包括:读取硬件寄存器、清除中断源、调度下半部等.

下半部(Bottom Half):- 下半部是中断处理程序的第二部分,通常由上半部调度执行.

- 其主要任务是延迟处理上半部未完成的工作,完成较为复杂和耗时的处理.

- 下半部运行在进程上下文中,可以被阻塞,也可以进行复杂的操作.

- 典型的下半部操作包括:数据处理、更新数据结构、唤醒等待的进程等.

例如,当网卡收到网络包后,通过 DMA 将数据写入内存,并通过硬件中断通知内核有新数据到达.内核调用中断处理程序,分为上半部和下半部.上半部会先禁止网卡中断,避免频繁硬中断降低内核效率,然后触发软中断,将耗时且复杂的任务交给软中断处理程序(下半部)处理,如解析网络数据并将其传递给应用程序.

为什么引入上半部和下半部处理?

1. 提高响应速度:上半部只执行最紧急的操作,尽量缩短中断处理时间,使系统能够快速响应其他中断.

2. 减少中断禁用时间:上半部运行在中断上下文中,系统在处理上半部时会禁用中断.通过将复杂操作移到下半部,可以减少中断禁用时间,提高系统的并发性.

3. 分离紧急和非紧急任务:将紧急任务放在上半部,非紧急任务放在下半部,有助于合理分配系统资源,提高系统的整体性能和稳定性.

所以,中断处理程序的上半部和下半部可以理解为:- 上半部直接处理硬件请求,也就是硬中断,主要是负责耗时短的工作,特点是快速执行；

- 下半部是由内核触发,也就是软中断,主要是负责上半部未完成的工作,通常都是耗时比较长的事情,特点是延迟执行.

还有一个区别,硬中断(上半部)是会打断 CPU 正在执行的任务,然后立即执行中断处理程序,而软中断(下半部)是以内核线程的方式执行,并且每一个 CPU 都对应一个软中断内核线程,名字通常为「ksoftirqd/CPU 编号」,比如 0 号 CPU 对应的软中断内核线程的名字是 ksoftirqd/0.

## xv6 源代码阅读相关问题 —— 启动与中断异常机制部分.

### a)特权模式及启动转换过程(boot)

问题:xv6-riscv 有⼏个特权模式,分别是什么?结合 entry.S,start. 以及 main. 部分代码,给出 xv6-riscv 启动阶段特权模式的转换过程.

回答:xv6-riscv 使用了 RISC-V 架构的三个特权模式:1. 机器模式(Machine Mode,M-mode):最高特权级别,可以访问和控制所有硬件资源

2. 监管者模式(Supervisor Mode,S-mode):操作系统内核运行的模式

3. 用户模式(User Mode,U-mode):用户程序运行的模式

启动阶段特权模式转换过程:1. 启动时(M-mode): \_entry: # set up a stack for C.

# stack0 is declared in start.,

# with a 4096-byte stack per CPU.

# sp = stack0 +(hartid \* 4096)

la sp,stack0

li a0,1024\*4

csrr a1,mhartid

addi a1,a1,1

mul a0,a0,a1

add sp,sp,a0

# jump to start()in start.

call start

QEMU启动时,将内核加载到物理地址0x80000000,处理器以M-mode开始执行\_entry,设置好栈后调用start()函数.

2. M-mode到S-mode转换(在start函数内): void start(){

// ...

// delegate all interrupts and exceptions to supervisor mode.

w\_medeleg(0xffff);

w\_mideleg(0xffff);

w\_sie(r\_sie()| SIE\_SEIE | SIE\_STIE | SIE\_SSIE);

// ...

// set M Previous Privilege mode to Supervisor,for mret.

unsigned long x = r\_mstatus();

x &= ~MSTATUS\_MPP\_MASK;

x |= MSTATUS\_MPP\_S;

w\_mstatus(x);

// set M Exception Program Counter to main,for mret.

// requires gcc -mcmodel=medany

w\_mepc((uint64)main);

// ...

// switch to supervisor mode and jump to main().

volatile("mret");

}

start()函数执行以下操作: - 将中断和异常委托给S-mode处理

- 设置mstatus的MPP字段为S-mode,这是为了指定mret指令执行后要切换到的特权模式.MPP(Machine Previous Privilege)字段保存了trap发生前的特权模式,但在这里是用来设置mret返回的目标模式.通过将MPP设为S-mode,确保执行mret指令后CPU会切换到监管者模式而不是用户模式或机器模式

- 设置mepc为main函数地址,这样当执行mret指令时,处理器会跳转到mepc指向的地址(即main函数).mepc(Machine Exception Program Counter)寄存器通常用于存储异常发生时的程序计数器值,但在这里被用来设置mret返回后的执行地址

- 执行mret指令,从M-mode切换到S-mode并跳转到main().这里使用mret而不是普通跳转指令是因为: 1. mret会自动将特权级从M-mode切换到MPP字段指定的模式(此处为S-mode)

2. mret会将PC设置为mepc寄存器的值(此处指向main函数)

3. mret会恢复中断使能状态,确保在新特权模式下正确设置中断处理

3. S-mode内核初始化(main函数内): void main(){

if(cpuid()== 0){

consoleinit();

printfinit();

printf("\n");

printf("xv6 kernel is booting\n");

printf("\n");

kinit(); // physical page allocator

kvminit(); // create kernel page table

kvminithart(); // turn on paging

procinit(); // process table

trapinit(); // trap vectors

trapinithart(); // install kernel trap vector

plicinit(); // set up interrupt controller

plicinithart(); // ask PLIC for device interrupts

// ...

userinit(); // first user process

// ...

}

// ...

scheduler(); }

main函数在S-mode中运行,初始化各种内核功能,最后调用userinit()创建第一个用户进程,并通过scheduler()开始进程调度.

4. S-mode到U-mode转换(在userinit和usertrapret内): 当内核通过调用usertrapret()准备执行用户进程时,会从S-mode切换到U-mode.

### b)系统调用寄存器使用(syscall)

问题:xv6-riscv 进⾏系统调⽤时,系统调⽤号、参数以及返回值分别通过哪些寄存器传递?

回答:在xv6-riscv中,系统调用使用以下寄存器:1. 系统调用号:通过寄存器a7传递

void

syscall(void)

{

int num;

struct proc \*p = myproc();

num = p->trapframe->a7; // 系统调用号存储在a7寄存器中

if(num > 0 && num < NELEM(syscalls)&& syscalls[num]){

// 调用对应的系统调用函数

p->trapframe->a0 = syscalls[num]();

} else {

// ...

}

}

2. 系统调用参数:通过寄存器a0-a5传递

void

argint(int n,int \*ip)

{

\*ip = argraw(n);

}

uint64

argraw(int n)

{

struct proc \*p = myproc();

switch(n){

case 0: return p->trapframe->a0;

case 5: return p->trapframe->a5;

}

panic("argraw");

return -1;

}

1. 系统调用返回值:通过寄存器a0返回

p->trapframe->a0 = syscalls[num](); // 返回值存储在a0寄存器中

总结:- 系统调用号:a7寄存器

- 系统调用参数:a0-a5寄存器(最多6个参数)

- 系统调用返回值:a0寄存器

### c)进程上下文切换寄存器(trap)

问题:xv6-riscv 在进⾏进程上下⽂切换时,需要保存和恢复的寄存器有哪些?

回答:在xv6-riscv中,进程上下文切换时需要保存和恢复的寄存器定义在struct context中:// Saved registers for kernel context switches.

struct context {

uint64 ra; // 返回地址寄存器

uint64 sp; // 栈指针寄存器

// callee-saved

uint64 s0; // 帧指针/保存寄存器

uint64 s1; // 保存寄存器

uint64 s11; // 保存寄存器

};在swtch.S中实现了上下文切换,保存和恢复这些寄存器:# a0 = 当前进程的context地址

# a1 = 目标进程的context地址

# 保存当前寄存器到当前进程的context

# 从目标进程的context加载寄存器

# 返回到新的进程上下文切换过程中:- 保存当前进程的 ra,sp,s0-s11 到当前进程的 context 结构

- 从目标进程的 context 结构加载 ra,sp,s0-s11

- ra 的值将决定返回地址,这使得执行流可以切换到新进程

当进程从用户态到内核态发生trap时,需要在trapframe中保存更多的寄存器.

另:xv6-riscv中用户态-内核态Trap的寄存器保存机制

在xv6-riscv中,进程上下文切换只需要保存/恢复少量寄存器,而用户模式到内核模式的trap(以及返回)则需要更全面的寄存器管理.下面详细解释这一机制:#### Trapframe结构

当从用户模式到内核模式发生trap时,xv6需要在trapframe结构中保存完整的处理器状态:struct trapframe {

/\* 0 \*/ uint64 kernel\_satp; // 内核页表

/\* 8 \*/ uint64 kernel\_sp; // 进程内核栈顶位置

/\* 16 \*/ uint64 kernel\_trap; // usertrap()函数地址

/\* 24 \*/ uint64 epc; // 保存的用户程序计数器

/\* 32 \*/ uint64 kernel\_hartid; // 保存的内核tp寄存器值

/\* 40 \*/ uint64 ra;

};这个结构包含:1. 内核配置数据(前5个字段):trap处理程序需要的关于内核的信息

2. 全部32个RISC-V整数寄存器:完整的用户模式CPU状态

#### 用户态到内核态的Trap过程

trap过程涉及几个关键组件:1. Trap入口(trampoline.S中的uservec)

当用户模式发生trap时:.globl uservec

uservec:# 交换a0和sscratch

csrrw a0,sscratch,a0

# 保存用户寄存器到TRAPFRAME

sd ra,40(a0)

sd sp,48(a0)

sd gp,56(a0)

sd tp,64(a0)

sd t0,72(a0)

# ...(保存所有寄存器)

sd t6,280(a0)

# 从trapframe恢复内核栈指针

ld sp,8(a0)

# 加载内核页表地址

ld t1,0(a0)

csrw satp,t1

sfence.vma zero,zero

# 跳转到usertrap()

ld t1,16(a0)

jr t1这段代码:1. 交换a0与sscratch(包含trapframe地址)

- 因为需要访问trapframe,但又不能丢失a0寄存器的值.sscratch在进入用户态前被设置为指向trapframe的指针,通过交换可以同时保存a0的值并获取trapframe地址.

2. 将所有用户寄存器保存到trapframe

- 保存完整的用户态CPU状态,以便在trap处理完成后能够正确恢复用户程序的执行.

3. 加载内核栈指针

- 需要切换到内核栈执行trap处理程序,每个进程都有自己的内核栈,其地址存储在trapframe中.

4. 切换到内核页表

- 用户程序和内核使用不同的页表,需要切换到内核页表才能访问内核空间的代码和数据.

5. 跳转到usertrap()函数

- usertrap()是C语言编写的trap处理主函数,负责根据trap原因执行相应的处理逻辑.

2. Trap处理程序(trap.c中的usertrap)

void

usertrap(void)

{

// ...

// 保存用户程序计数器

p->trapframe->epc = r\_sepc();

// 处理trap(系统调用、中断或异常)

if(r\_scause()== 8){

// 系统调用

syscall();

} else if((r\_scause()& 0x8000000000000000L)!= 0){

// 中断

// ...

} else {

// 异常

// ...

}

usertrapret();

}3. 返回用户空间(usertrapret和userret)

void

usertrapret(void)

{

// ...

// 为下一次用户空间trap配置trapframe

p->trapframe->kernel\_satp = r\_satp();

p->trapframe->kernel\_sp = p->kstack + PGSIZE;

p->trapframe->kernel\_trap =(uint64)usertrap;

p->trapframe->kernel\_hartid = r\_tp();

// 设置返回用户模式

w\_sepc(p->trapframe->epc);

// ...

// 跳转到trampoline.S中的userret

uint64 trampoline\_userret = TRAMPOLINE +(userret - trampoline);

((void(\*)(uint64))trampoline\_userret)(TRAPFRAME);

}

.globl userret

userret: # 切换到用户页表

# ...

# 恢复所有用户寄存器

ld ra,40(a0)

ld sp,48(a0)

# ...(从trapframe恢复所有寄存器)

ld t6,280(a0)

# 返回用户模式

csrw sscratch,a0

csrr a0,sscratch

sret返回过程:1. 准备下一次trap的trapframe

- 保存内核页表、内核栈指针等信息,为下次从用户空间进入内核做准备

- 确保trap处理程序能正确访问内核数据结构

2. 在sepc设置用户程序计数器

- 将用户程序的返回地址写入sepc寄存器

- 使sret指令能正确返回到用户程序的下一条指令

3. 跳转到trampoline.S中的userret

- trampoline页面在用户和内核页表中映射到相同位置

- 使切换页表时代码能继续执行

4. 切换到用户页表

- 将satp寄存器设置为用户页表

- 使进程能访问其用户空间内存

5. 从trapframe恢复所有用户寄存器

- 恢复用户程序的执行上下文

- 包括通用寄存器、栈指针等

6. 通过sret返回用户模式

- sret指令切换到用户特权级

- 从sepc恢复PC,继续执行用户程序

#### 上下文切换与用户-内核trap的比较

用户-内核trap处理与上下文切换的主要区别:| 方面 | 用户-内核Trap | 上下文切换 |

| - | - | - |

| 目的 | 在特权模式间转换 | 在内核模式下切换进程 |

| 保存的寄存器 | 全部32个整数寄存器 | 仅被调用者保存的寄存器(ra,sp,s0-s11)|

| 结构 | trapframe | context |

| 页表 | 从用户到内核页表转换 | 不改变 |

| 特权模式 | 改变(U→S或S→U)| 保持在S模式 |

| 触发 | 异常、中断、系统调用 | swtch()调用 |

总结来说,用户和内核模式之间的trap是比内核线程间上下文切换更复杂的操作,需要保存和恢复完整的CPU状态,同时还需要更改特权模式和页表.

### d)Trap 处理相关寄存器(trap)

问题:xv6-riscv 在处理 trap 时,使⽤了⼀系列控制寄存器,请简要说明以下寄存器的作⽤:stvec,sepc,scause,sstatus.

回答:1. stvec(Supervisor Trap Vector)

- 作用:存储处理trap的处理程序入口地址

- 代码参考: void

trapinithart(void)

{

w\_stvec((uint64)kernelvec);

}

- 当S-mode下发生异常或中断时,处理器会跳转到stvec指向的地址

- 在内核态设置为kernelvec,在用户态跳转前设置为trampoline.S中的uservec

2. sepc(Supervisor Exception Program Counter)

- 作用:保存trap发生时的程序计数器值

- 在trap发生时,处理器会自动将当前PC值保存到sepc中

- 在trap处理完成后,通过sret指令返回时,处理器会将sepc的值加载到PC中

- 代码参考: void

usertrapret(void)

{

// ...

// set up trapframe values that uservec will need when

// the process next traps into the kernel.

p->trapframe->kernel\_satp = r\_satp(); // kernel page table

p->trapframe->kernel\_sp = p->kstack + PGSIZE; // process's kernel stack

p->trapframe->kernel\_trap =(uint64)usertrap;

p->trapframe->kernel\_hartid = r\_tp(); // hartid for cpuid()

// set up the registers that trampoline.S's sret will use

// to get to user space.

// set S Previous Privilege mode to User.

unsigned long x = r\_sstatus();

x &= ~SSTATUS\_SPP; // clear SPP to 0 for user mode

x |= SSTATUS\_SPIE; // enable interrupts in user mode

w\_sstatus(x);

// set S Exception Program Counter to the saved user pc.

w\_sepc(p->trapframe->epc);

// ...

}

3. scause(Supervisor Cause Register)

- 作用:指示trap的原因(异常或中断类型)

- 代码参考: void

usertrap(void)

{

// ...

// ok,it's safe to enable interrupts.

intr\_on();

// check which type of trap occurred

uint64 cause = r\_scause();

if(cause == 8){

// system call

// ...

} else if((cause & 0x8000000000000000L)&&

(cause & 0xff)== 9){

// supervisor external interrupt,via PLIC.

// ...

} else if(cause == 0xf){

// ...

}

// ...

}

4. sstatus(Supervisor Status Register)

- 作用:控制和反映处理器的当前状态

- 包含各种状态位,如: - SIE(Supervisor Interrupt Enable):控制是否允许S-mode中断

- SPP(Supervisor Previous Privilege):记录trap前的特权模式

- SPIE(Supervisor Previous Interrupt Enable):记录trap前的中断使能状态

- 代码参考: // enable device interrupts

void

intr\_on()

{

w\_sstatus(r\_sstatus()| SSTATUS\_SIE);

}

// disable device interrupts

void

intr\_off()

{

w\_sstatus(r\_sstatus()& ~SSTATUS\_SIE);

}

这些寄存器共同协作,使处理器能够完成异常和中断的捕获、处理与返回流程.

### e)中断处理函数流程(interrupt)

问题:阅读 uart. 和 trap. 中有关中断的代码,简要总结 devintr 和 uartintr 两个函数内部流程,及 devintr 内调⽤ uartintr 的判断条件.

回答:1. devintr 函数流程: // 检查并处理设备中断

// 返回中断处理状态:2表示时钟中断,1表示其他中断,0表示不是中断

int

devintr()

{

uint64 scause = r\_scause();

// 检查是否为外部中断(最高位为1,且中断码为9)

if((scause & 0x8000000000000000L)&&(scause & 0xff)== 9){

// 是PLIC引起的中断

int irq = plic\_claim(); // 获取中断源ID

if(irq == UART0\_IRQ){

// UART中断

uartintr(); // 调用UART中断处理函数

} else if(irq == VIRTIO0\_IRQ){

// 磁盘中断

virtio\_disk\_intr();

} else if(irq){

// 其他设备中断

printf("unexpected interrupt irq=%d\n",irq);

}

// 完成中断处理并返回

if(irq)

plic\_complete(irq);

return 1; // 非时钟中断

}

// 检查是否为软件中断(最高位为1,且中断码为1)

else if(scause == 0x8000000000000001L){

// 如果是CPU计时器引起的软件中断...

if(cpuid()== 0){

clockintr(); // 处理时钟中断

}

// 清除软件中断待处理位

w\_sip(r\_sip()& ~2);

return 2; // 时钟中断

}

return 0; // 不是中断

}

devintr 函数

- 作用:检查和处理设备中断,即外部中断和时钟中断

- 流程: 1. 读取 scause 寄存器判断中断类型

2. 如果是外部中断(PLIC中断): - 通过 plic\_claim()获取中断源ID

- 根据中断源ID调用对应的处理函数(如 uartintr)

- 处理完成后调用 plic\_complete()通知PLIC

- 返回1表示处理了非时钟中断

3. 如果是软件中断(时钟中断): - 在CPU0上调用 clockintr()处理时钟中断

- 清除待处理位

- 返回2表示处理了时钟中断

4. 其他情况返回0表示不是中断

2. uartintr 函数流程: // UART中断处理函数

void

uartintr(void)

{

// 在存在输入字符时处理

while(1){

int = uartgetc(); // 尝试从UART读取一个字符

if( == -1)// 没有更多字符可读

break;

consoleintr(); // 将字符传递给控制台处理函数

}

// 发送缓冲区中的字符

uartstart();

}

- 作用:处理UART(通用异步收发器)设备产生的中断,主要用于处理串口通信时的数据接收中断

- 流程: 1. 检查UART控制寄存器判断是否有数据可读

2. 如果有数据,从UART数据寄存器读取数据

3. 将读取的数据加入到console buffer中

4. 如果读到回车符 \n ,唤醒等待输入的进程

5. 循环处理直到没有更多数据

3. devintr 内调用 uartintr 的判断条件:devintr 调用 uartintr 的条件是:- scause 寄存器显示这是一个外部中断(最高位为1)

- 中断码为9(PLIC中断),表示这是由PLIC(Platform Level Interrupt Controller)转发的外部设备中断

- PLIC返回的中断源ID为UART0\_IRQ(UART设备的中断请求号),表示这是一个来自UART设备的中断

这个流程反映了设备中断的层级化处理模式:PLIC硬件识别中断源→通过trap机制进入内核→devintr函数确定中断类型→根据中断源调用专门的处理函数(如uartintr).

# 作业 2

## 调度算法

复习课件 4-进程线程调度,结合课上讲述的批处理系统调度算法和实时系统调度算法,阅读 Three Easy Pieces 的第七章 Scheduling:Introduction,简要总结 FIFO、SJF、STCF(课件上称为 SRTN)、Round Robin 四种调度算法的特点及优劣.

Answer:\* FIFO(First-In,First-Out)/ FCFS(First-Come,First-Served)先来先服务: \* 特点:非抢占式.按照进程到达就绪队列的顺序进行调度.实现简单.

\* 优点:简单、公平(每个进程最终都会被服务).

\* 缺点:平均周转时间通常较差,特别是当短进程排在长进程之后时(护航效应 Convoy Effect),导致短进程等待时间过长.不适合交互式系统,响应时间无保证.

\* SJF(Shortest Job First)最短作业优先: \* 特点:可以是非抢占式或抢占式(STCF/SRTN).选择预计下一个 CPU 执行时间最短的进程进行调度.

\* 优点:在非抢占式情况下,如果所有进程同时到达,平均等待时间和平均周转时间最短.

\* 缺点: \* 难以准确预测下一个 CPU 执行时间.

\* 可能导致长作业饥饿(Starvation),即长作业一直得不到运行机会.

\* 非抢占式版本中,如果一个长作业已经开始执行,即使后面来了更短的作业,也必须等长作业完成或阻塞.

\* STCF(Shortest Time-to-Completion First)/ SRTN(Shortest Remaining Time Next)最短剩余时间优先: \* 特点:抢占式的 SJF.调度器选择剩余执行时间最短的进程.如果一个新到达的进程比当前正在执行进程的剩余时间还要短,则抢占当前进程.

\* 优点:提供了最优的平均周转时间.相比非抢占式 SJF,对后来到达的短作业响应更快.

\* 缺点: \* 同样存在预测执行时间的困难.

\* 长作业饥饿问题依然存在.

\* 抢占和上下文切换会带来额外的开销.

\* RR(Round Robin)时间片轮转: \* 特点:抢占式.将所有就绪进程按 FCFS 组成队列.调度器选择队首进程,分配一个固定的时间片(Quantum).时间片用完后,若进程未完成或阻塞,则被抢占并移到就绪队列末尾.

\* 优点:公平,每个进程都能获得运行机会.响应时间较快,适合分时系统和交互式系统.实现相对简单.

\* 缺点: \* 性能对时间片长度敏感:时间片太短,上下文切换开销大；时间片太长,则退化为 FCFS.

\* 平均周转时间通常比 SJF/STCF 差.对于运行时间相近的进程,平均周转时间可能比 FCFS 还差.

总结:| 算法名称 | 是否抢占式 | 核心特点 | 优点 | 缺点 |

| - | - | - | - | - |

| FIFO / FCFS | 否 | 按到达顺序执行 | 简单,公平 | 护航效应,平均周转时间差,响应时间无保证 |

| SJF | 可选 | 选择预计执行时间最短的作业 |(非抢占式)最优平均周转时间(若同时到达)| 难以预测执行时间,长作业可能饥饿 |

| STCF / SRTN | 是 | 选择剩余执行时间最短的作业,可抢占 | 最优平均周转时间,对短作业响应快 | 难以预测执行时间,长作业可能饥饿,上下文切换开销 |

| Round Robin(RR)| 是 | 按时间片轮流执行 | 公平,响应时间较好,适合交互式系统 | 性能对时间片敏感,平均周转时间通常不如SJF/STCF,上下文切换开销 |

## xv6 源代码阅读相关问题——进程模型与调度部分.

请阅读 xv6-riscv 源代码及 xv6-book 的相关章节进⾏学习后,回答下列问题.

[OSTEP 书籍在线阅读地址](https://pages.cs.wisc.edu/~remzi/OSTEP/)

参考 xv6-book 章节:2.5,7.1~7.4.

参考 xv6-riscv 相关源代码文件:kernel 目录下:proc.h,proc.,swtch.S；user 目录下:sh..

### 进程 API

阅读 Three Easy Pieces 的第五章 Process API 和 xv6 相应部分函数代码,了解 shell、fork()、exec()和 wait()之间的关联,体会这样设计的好处,简要给出你的概括(api).

Answer:简要概括:在 xv6 中,Shell(user/sh.)、fork()(kernel/proc.)、exec()(系统调用,最终在内核处理)和 wait()(kernel/proc.)协同工作,构成了基本的进程管理和命令执行流程.

详细分析:1. Shell 读取与解析命令:Shell 进程(sh)首先通过 getcmd()读取用户输入的命令行,然后调用 parsecmd()解析命令,将其构造成内部表示(如 execcmd,pipecmd 等).

2. 创建子进程(fork()):对于大多数命令(除了像 cd 这样的内建命令),Shell 调用 fork1()(它内部调用 fork()系统调用,对应 kernel/proc. 中的 fork 函数).

\* fork()会创建一个新的子进程,这个子进程几乎是父进程(Shell)的一个副本.它复制父进程的内存空间(uvmcopy)、文件描述符(filedup)、当前工作目录(idup)等.

\* 关键区别在于 fork()对父进程返回子进程的 PID,对子进程返回 0.这使得代码可以区分父子进程.

\* 子进程的初始状态是 RUNNABLE,并被设置为从 forkret 开始执行(通过设置 context.ra),最终返回用户空间时,其 a0 寄存器(返回值)被设置为 0(np->trapframe->a0 = 0).

// Create a new process,copying the parent.

// Sets up child kernel stack to return as if from fork()system call.

int

fork(void)

{

// ... 省略分配和检查代码 ...

// Copy user memory from parent to child.

if(uvmcopy(p->pagetable,np->pagetable,p->sz)< 0){

freeproc(np);

release(&np->lock);

return -1;

}

np->sz = p->sz;

// copy saved user registers.

\*(np->trapframe)= \*(p->trapframe);

// Cause fork to return 0 in the child.

np->trapframe->a0 = 0;

// ... 省略文件描述符和 cwd 复制 ...

pid = np->pid;

release(&np->lock);

// ... 省略设置 parent 和 state ...

return pid;

}

:Resource/xv6-riscv-阅读考察/user/sh.(lines 158-163)

int

main(void)

{

// ... 省略初始化代码 ...

while(getcmd(buf,sizeof(buf))>= 0){

// ... 省略 cd 处理 ...

if(fork1()== 0)// 子进程执行 runcmd

runcmd(parsecmd(buf));

wait(0); // 父进程等待子进程结束

}

exit(0);

}

3. 执行新程序(exec()):在 fork()创建的子进程中,通常需要执行用户指定的命令(例如 ls,grep 等).子进程会调用 exec()系统调用(在 sh. 的 runcmd 函数的 EXEC 分支中调用).

\* exec()会加载指定路径的可执行文件,用新的程序代码、数据和堆栈替换当前进程(子进程)的内存映像.

\* 如果 exec()成功,它不会返回到原来的调用点(即 runcmd 中的 exec 调用之后)；子进程从此开始执行新程序的 main 函数.如果 exec()失败(例如找不到文件或格式错误),它会返回 -1.

void

runcmd(struct cmd \*cmd)

{

// ... 省略其他 case ...

case EXEC: ecmd =(struct execcmd\*)cmd;

if(ecmd->argv[0] == 0)

exit(1);

exec(ecmd->argv[0],ecmd->argv); // 子进程调用 exec

fprintf(2,"exec %s failed\n",ecmd->argv[0]); // 只有 exec 失败才会执行这里

break;

// ... 省略其他 case ...

}

4. 等待子进程结束(wait()):父进程(Shell)在 fork()之后,通常需要等待子进程执行完毕,以便回收子进程资源并获取其退出状态.Shell 调用 wait()系统调用(对应 kernel/proc. 中的 wait 函数).

\* wait()会检查调用进程是否有处于 ZOMBIE 状态的子进程.ZOMBIE 状态表示子进程已经 exit()但其父进程尚未 wait()它.

\* 如果找到僵尸子进程,wait()会收集其退出状态,释放子进程占用的资源(如 proc 结构体、内核栈等,通过 freeproc),并返回子进程的 PID.

\* 如果没有僵尸子进程,且该父进程有子进程存在,wait()会使父进程进入 SLEEPING 状态(通过 sleep(p,&wait\_lock)),直到某个子进程调用 exit().

\* 子进程调用 exit()时,会将其状态设置为 ZOMBIE,保存退出状态 xstate,并唤醒可能正在 wait()的父进程(wakeup(p->parent)).

// Wait for a child process to exit and return its pid.

// Return -1 if this process has no children.

int

wait(uint64 addr)

{

// ... 省略变量定义 ...

acquire(&wait\_lock);

for(;;){ // 无限循环,直到找到僵尸子进程或没有子进程

// Scan through table looking for exited children.

havekids = 0;

for(pp = proc; pp < &proc[NPROC]; pp++){

if(pp->parent == p){ // 检查是否是自己的子进程

acquire(&pp->lock);

havekids = 1;

if(pp->state == ZOMBIE){ // 找到僵尸子进程

// Found one.

pid = pp->pid;

// ... 省略 copyout 退出状态 ...

freeproc(pp); // 释放子进程资源

release(&pp->lock);

release(&wait\_lock);

return pid; // 返回子进程 PID

}

release(&pp->lock);

}

}

// No point waiting if we don't have any children.

if(!havekids || killed(p)){

release(&wait\_lock);

return -1;

}

// Wait for a child to exit.

sleep(p,&wait\_lock); // 父进程睡眠,等待子进程退出

}

}

// Exit the current process. Does not return.

// An exited process remains in the zombie state

// until its parent calls wait().

void

exit(int status)

{

// ... 省略关闭文件、清理 cwd 等 ...

acquire(&wait\_lock);

// Give any children to init.

reparent(p);

// Parent might be sleeping in wait().

wakeup(p->parent); // 唤醒父进程

acquire(&p->lock);

p->xstate = status; // 保存退出状态

p->state = ZOMBIE; // 设置为僵尸状态

release(&wait\_lock);

// Jump into the scheduler,never to return.

sched(); // 切换到调度器,不再返回

panic("zombie exit");

}

设计好处:\* 隔离性(Isolation):通过 fork()创建独立的子进程来执行命令,使得命令的执行环境与 Shell 的执行环境隔离开.即使子进程崩溃或行为异常,也不会影响父进程 Shell 的稳定性.

\* 代码复用与简洁性(Simplicity & Reusability):Shell 不需要知道每个外部命令的具体实现.它只需要通过统一的 fork()和 exec()接口来创建和加载程序.exec()使得任何编译好的程序都可以被 Shell 执行.

\* 资源管理(Resource Management):fork()复制文件描述符等资源,使得 I/O 重定向(如 >、<、|)可以在子进程 exec()之前方便地设置(见 sh. 中 runcmd 的 REDIR 和 PIPE 处理).wait()机制确保父进程可以回收子进程结束时遗留的资源,防止资源泄漏.

\* 并发性(Concurrency):fork()后父子进程并发执行.Shell 可以选择 wait()等待子进程(前台命令),也可以不 wait()而继续接收新命令(后台命令,通过 sh. 中 runcmd 的 BACK 类型实现),提高了交互效率.

### 阅读 xv6 进程调度算法部分

简要回答:xv6 的调度基于线程还是基于进程?是否有进程优先级?按照什么顺序挑选要调度的进程?允许查阅资料和使用大模型辅助,简要描述可能的 xv6-riscv 调度算法的优化思路(提示:xv6 是一个多核系统,以及可结合本题前⾯⼏个问题的答案)(schedule).

Answer:简要概括:1. 调度单位:xv6 的调度是基于 进程(Process)的.内核中管理和调度的基本单位是 struct proc 结构体.xv6 没有实现内核级线程.

2. 进程优先级:xv6 的调度算法 没有实现进程优先级.

3. 挑选顺序:xv6 采用简单的 轮询(Round-Robin)方式,按固定顺序遍历全局进程表 proc 数组,选择第一个状态为 RUNNABLE 的进程执行.

4. 可能的优化思路: \* 引入优先级:为进程赋予不同优先级,优先调度高优先级进程.

\* 多级反馈队列(MLFQ):结合优先级和时间片,动态调整进程优先级,平衡响应时间和吞吐量.

\* 每个 CPU 独立的运行队列:减少多核环境下对全局进程表锁的争用,提高并行度.

\* CPU 亲和性(Affinity):尽量将进程固定在某个 CPU 上运行,提高缓存命中率.

\* 负载均衡:在各 CPU 的运行队列之间迁移进程,避免某些 CPU 过载而其他 CPU 空闲.

\* 近似 SJF/STCF:尝试预测进程运行时间(例如基于历史运行时间),优先调度短作业(实现复杂且预测不一定准确).

详细分析:1. 调度单位:xv6 的调度是基于 进程(Process)的.内核中管理和调度的基本单位是 struct proc 结构体(定义在 kernel/proc.h).xv6 没有实现内核级线程的概念,每个进程拥有自己独立的地址空间、内核栈、上下文等.上下文切换(swtch in kernel/swtch.S)保存和恢复的是整个进程的上下文(struct context).

2. 进程优先级:xv6 的调度算法 没有实现进程优先级.所有进程都被同等对待.

3. 挑选顺序:xv6 的调度器(scheduler 函数在 kernel/proc.)采用一种简单的 轮询(Round-Robin)方式.它按固定的顺序遍历全局的进程表 proc 数组(for(p = proc; p < &proc[NPROC]; p++)).当它找到第一个状态为 RUNNABLE 的进程时,就选择该进程运行.

\* 它将选中的进程状态置为 RUNNING,然后通过 swtch(&->context,&p->context)将 CPU 的控制权交给该进程.

\* 当该进程放弃 CPU 时(通过 yield(),sleep(),exit()中的 sched()调用),控制权会通过 swtch(&p->context,&mycpu()->context)返回到调度器.调度器随后会继续从上次停止的地方(或从头开始,取决于实现细节,但 xv6 是从头开始)扫描进程表,寻找下一个 RUNNABLE 的进程.

// Per-CPU process scheduler.

// Each CPU calls scheduler()after setting itself up.

// Scheduler never returns. It loops,doing: // - choose a process to run.

// - swtch to start running that process.

// - eventually that process transfers control

// via swtch back to the scheduler.

void

scheduler(void)

{

struct proc \*p;

struct cpu \* = mycpu(); // 获取当前 CPU

->proc = 0;

for(;;){ // 无限循环

// ... 省略中断开启 ...

int found = 0;

for(p = proc; p < &proc[NPROC]; p++){ // 遍历整个进程表

acquire(&p->lock);

if(p->state == RUNNABLE){ // 找到第一个可运行进程

// Switch to chosen process.

p->state = RUNNING;

->proc = p;

swtch(&->context,&p->context); // 切换到进程 p

// Process is done running for now.

->proc = 0;

found = 1;

}

release(&p->lock);

}

// ... 省略 wfi 处理 ...

}

}

4. 可能的优化思路:xv6 的简单轮询调度算法虽然易于理解和实现,但在性能、公平性和响应性方面有很大的提升空间,尤其是在多核环境下.结合 xv6 book 中提到的调度指标(周转时间、公平性)和算法(FIFO、SJF、STCF),以及 xv6 的多核特性,可以考虑以下优化思路: \* 引入优先级(Priority Scheduling): \* 在 struct proc 中增加一个 priority 字段.

\* 修改 scheduler,使其不再选择第一个 RUNNABLE 进程,而是选择具有最高优先级的 RUNNABLE 进程.

\* 优先级可以是静态的(创建时指定),也可以是动态调整的(例如,根据进程的等待时间或 CPU 使用情况调整,倾向于提高等待时间长的或 I/O 密集型进程的优先级).这可以部分模拟 SJF/STCF 的效果,改善平均周转时间.

\* 多级反馈队列(Multi-level Feedback Queue,MLFQ): \* 维护多个运行队列,每个队列对应一个优先级.

\* 新进程放入最高优先级队列.

\* 如果进程在一个时间片内用完了 CPU 时间,则降低其优先级,移入下一级队列(惩罚 CPU 密集型进程).

\* 如果进程在时间片结束前主动放弃 CPU(如进行 I/O 操作),则保持或提升其优先级(奖励 I/O 密集型进程).

\* 调度器优先从高优先级队列选择进程.

\* 这种方法能较好地平衡公平性和响应时间,无需预知进程运行时间.

\* 改进运行队列结构(Improved Run Queue): \* 当前的 scheduler 每次都需要 O(NPROC)的时间扫描整个 proc 数组.可以为每个 CPU 维护一个独立的、只包含 RUNNABLE 进程的链表或其它更高效的数据结构(如红黑树,如果需要按优先级排序).

\* 这样查找下一个可运行进程的时间复杂度可以降低到 O(1)(简单链表)或 O(log N)(优先级队列).

\* CPU 亲和性(CPU Affinity): \* xv6 是多核的(NCPU > 1).进程在不同 CPU 间切换会导致缓存失效,降低性能.

\* 可以记录每个进程最后运行在哪个 CPU 上,调度器在选择进程时,尽量让该进程在同一个 CPU 上运行(软亲和性),或者强制其只能在某个或某些 CPU 上运行(硬亲和性).

\* 多核负载均衡(Load Balancing): \* 如果使用每 CPU 的运行队列,可能会出现某些 CPU 很忙而其它 CPU 空闲的情况.

\* 需要实现负载均衡机制,定期检查各 CPU 运行队列的长度,将进程从繁忙的 CPU 迁移到空闲的 CPU,以提高整体吞吐量.

# Note 1

## 1. 操作系统是什么?

### 1.1 操作系统的定义

操作系统是计算机系统中的一个系统软件,是一些程序模块的集合:- 资源的管理者:能以尽量有效、合理的方式组织和管理计算机的软硬件资源

- 服务的提供者:合理地组织计算机的工作流程,控制程序的执行并向用户提供各种服务功能

- 机器能力的扩展:使得用户能够灵活、方便地使用计算机,使整个计算机系统能高效地运行

### 1.2 操作系统的作用

- 虚拟机:将物理资源(处理器、内存、磁盘)转换成更通用、更强大、更易用的虚拟形式

- APIs:提供可供用户调用的接口,提供应用程序的标准库

- 资源管理器:允许多程序运行(共享CPU),允许多程序并发访问内存,允许多程序访问设备

### 1.3 设计与实现目标

- 抽象

- 模块化

- 使用高级语言(C)而非汇编

- 性能(最小化开销)

- 最小化额外的时间(指令)

- 最小化额外的空间(内存/磁盘)

- 保护:应用之间、操作系统与应用之间的隔离

- 可靠性

- 节能

- 安全性:防止恶意应用的入侵

- 移动性:能够运行在越来越小的设备上

## 2. 从不同角度认知操作系统

### 2.1 资源管理的观点

- 自底向上 操作系统是资源的管理者

- 硬件资源:CPU,内存,设备(I/O设备、磁盘、时钟、网络接口等)

- 软件资源:磁盘上的文件、信息

- 资源管理目的:实现资源共享、提高资源利用率

- 复用方式:时间及空间

怎样管理资源?

- 数据结构与算法

- 跟踪记录资源使用状况

- 分配和回收资源(资源分配策略与算法)

- 静态分配策略

- 动态分配策略 ✓

- 资源管理目标

- 提高资源利用率

- 资源使用时的保护

- 协调多个进程对资源请求的冲突

### 2.2 进程的观点

从操作系统运行的角度动态观察操作系统:- 操作系统是由一些可同时、独立运行的进程和一个对这些进程进行协调的核心组成

- 进程:完成某一特定功能的程序,是程序的一次执行过程,动态的、有生命的,有诞生/消亡

### 2.3 虚机器观点

从操作系统内部结构来看:- 把操作系统分成若干层

- 每一层完成其特定功能,构成一个虚机器,并对上一层提供支持

- 通过逐层功能扩充,最终完成整个操作系统虚机器

- 操作系统虚机器向用户提供各种功能,完成用户请求

## 3. 操作系统的特征

### 3.1 并发(concurrency)

- 处理多个同时性活动的能力

- 计算机系统中同时运行多个程序

- 宏观上:这些程序同时在执行

- 微观上:单CPU情况下,任何时刻只有一个程序在执行,即这些程序在CPU上轮流执行

- 由并发引起的问题:活动切换、保护、相互依赖的活动间的同步

- 并行(parallel)与"并发"的区别

### 3.2 共享(sharing)

操作系统与多个用户程序共同使用计算机系统中的资源

- 共享有限的系统资源

- 操作系统要对系统资源进行合理分配和使用

- 资源在一个时间段内交替被多个进程所用

- 互斥共享(如打印机)

- 同时共享(访问)(如可重入代码,磁盘文件)

- 由于共享引发的问题:资源分配难以达到最优化,资源使用时需要保护

### 3.3 虚拟(Virtual)

- 一个物理实体映射为若干个对应的逻辑实体(分时或分空间)

- 虚拟技术是操作系统管理系统资源的重要手段,可提高资源利用率

- CPU:每个用户(进程)的"虚处理器"

- 存储器:每个进程都占有的地址空间(代码＋数据＋堆、栈)

- 显示设备:多窗口或虚拟终端

### 3.4 随机性(不确定性)

- 操作系统必须随时对以不可预测的次序发生的事件进行响应

- 进程的运行速度不可预知:多个进程并发执行,"走走停停",无法预知每个进程的运行推进快慢

- 难以重现系统在某个时刻的状态(包括重现运行中的错误)

## 4. 操作系统发展历史

### 技术变化与概念重用

技术变化导致某些思想过时并迅速消失,但技术的另一种变化还可能使它们复活.例如:- 磁盘上文件分配—连续文件(CD-ROM文件系统)

- 硬件保护

- 动态链接(MULTICS首先提出)

- 计算服务(MULTICS,以大量的、附有相对简单用户机器的、集中式Internet服务器形式回归)→ 云计算

### 4.1 操作系统发展阶段

#### 第1阶段(1948-1970):硬件昂贵,人工便宜

- 控制台:一次一个用户(独占资源)

- 批处理:装入程序 → 运行 → 打印输出结果(无保护)

- 多道程序设计:多个程序同时运行,多个用户共享系统(需要存储保护)

- SPOOLing技术:批处理作业处理流程

#### 第2阶段(1970-1985):硬件便宜,人工昂贵

- 交互、分时:多个用户同时与系统交互

- 用户可以在线工作:开发、调试、编辑等

- 问题:增加用户时 → 系统性能降低(响应时间、抖动)

第一个分时操作系统CTSS:- 1959年在MIT提出分时系统思想

- 每个用户有一个联机终端

- 计算机能够为许多用户提供交互式、快速服务,同时在CPU空闲时还能在后台运行大作业

重要历史事件:- OS/360:IBM发布时带着已知的1000个错误

- Multics:1963年开始,1969年才发布

- UNIX:一群计算机迷在贝尔实验室开发,初衷是在DEC PDP-7小型计算机上玩星际探险游戏

#### 第3阶段(1981-):硬件非常便宜,人工非常昂贵

- 个人计算时代

- 开始PC硬件资源有限,一次运行一个程序,OS是一个例程库,回归简单

- 逐渐PC资源丰富,OS又成为一个庞然大物(大型OS),存储保护、多道程序设计再次出现

#### 第4阶段(1981-):分布式

- 网络:允许不同机器很容易地相互共享资源(打印机、文件服务器、Web服务器)

- 解决问题:共享,安全

#### 第5阶段(1995-):移动计算时代

- 各种移动终端的出现(笔记本、平板、手机、机顶盒、可穿戴设备等)

- 特点:小型、移动、便宜,但能力有限

#### 第6阶段(2006-):云计算时代

- 提供可无限扩展的、可随时获取的、按需使用、按使用付费的资源

- 云计算操作系统:云计算后台数据中心的整体管理运营系统

- 作用: - 管理和驱动海量服务器、存储等基础硬件

- 为云应用软件提供统一、标准的接口

- 管理海量的计算任务以及资源调配

#### 第7阶段(200?-):泛在计算/普适计算/物联网

- 许多联网设备为许多人提供个性化的服务

## 5. 操作系统分类

### 5.1 传统分类

#### 1. 批处理操作系统

- 批:供一次加载的磁带或磁盘,通常由若干个作业组装成

- 工作方式: 1. 用户将作业交给系统操作员

2. 系统操作员将许多用户的作业组成一批作业,输入到计算机系统中

3. 系统操作员启动操作系统

4. 系统自动、依次执行每个作业

5. 由系统操作员将作业结果交给用户

- 典型的作业结构:由一张张卡片组成,卡片上是命令和程序

- 分类: - 单道批处理系统(simple batch processing,uni-programming)

- 多道批处理系统(multiprogramming system)

SPOOLing系统(技术)

- 1961年,英国曼彻斯特大学,Atalas机

- Simultaneous Peripheral Operation On-Line(同时的外围设备联机操作)-假脱机技术

- 思想:利用磁盘作缓冲,将输入、计算、输出分别组织成独立的任务流,使I/O和计算真正并行

- 工作原理

- 作业进入到磁盘上的输入井

- 按某种调度策略选择几种搭配得当的作业,并调入内存

- 作业运行的结果输出到磁盘上的输出井

- 结果从磁盘上的输出井送到打印机

- 主要特点: - 提高I/O速度

- 将独占设备改造为共享设备

- 实现虚拟设备功能

#### 2. 分时操作系统(time-sharing system)

- 时间片(time slice):操作系统将CPU的时间划分成若干个片段

- 操作系统以时间片为单位,轮流为每个终端用户服务,每次服务一个时间片(其特点是利用人的错觉,使人感觉不到)

- 追求目标:及时响应(依据是响应时间)

- 响应时间:从终端发出命令到系统给予回答所经历的时间

#### 3. 实时操作系统

- 使计算机能及时响应外部事件的请求,在规定的严格时间内完成对该事件的处理

- 分类: - 实时过程控制:工业控制,军事控制

- 实时通信(信息)处理:电讯(自动交换),银行,飞机订票,股市行情

- 追求目标:对外部请求在严格时间范围内作出反应,高可靠性

- 特征:关键参数是时间

- 类型: - 硬实时系统:某个动作绝对必须在规定的时刻或时间范围完成

- 软实时系统:接受偶尔违反最终时限

#### 4. 个人计算机操作系统

- 计算机在某一时间内为单用户服务

- 追求目标:界面友好,使用方便,丰富的应用软件

#### 5. 网络操作系统

- 基于计算机网络,在各种计算机操作系统上按网络体系结构协议标准开发的软件

- 功能:网络管理,通信,安全,资源共享和各种网络应用

- 追求目标:相互通信,资源共享

#### 6. 分布式操作系统

- 分布式系统:处理和控制的分散(相对于集中式系统)

- 以计算机网络为基础,基本特征是处理的分布(功能和任务的分布)

- 所有系统任务可在系统中任何处理机上运行,自动实现全系统范围内的任务分配

- 特征: 1. 是一个统一的操作系统

2. 资源进一步共享

3. 透明性:资源共享,分布对用户来讲是不知道的

4. 自治性:处于分布式系统的多个主机处于平等地位,无主从关系

5. 处理能力增强、速度更快、可靠性增强

#### 7. 嵌入式操作系统

- 嵌入式系统:在各种设备、装置或系统中,完成特定功能的软硬件系统

- 嵌入式操作系统:运行在嵌入式系统环境中,对整个嵌入式系统及其所操作、控制的各种部件装置等资源进行统一协调、调度、指挥和控制的系统软件

### 5.2 Tanenbaum分类

- 大型机操作系统

- 服务器操作系统

- 多处理机操作系统

- 个人计算机操作系统

- 掌上计算机操作系统(移动计算机操作系统)

- 嵌入式操作系统

- 传感器节点操作系统

- 实时操作系统

- 智能卡操作系统

#### 智能卡操作系统

- 智能卡:一种包含有一块CPU芯片的信用卡

- 特点: - 非常严格的运行能耗和存储空间的限制

- 有些智能卡只有单项功能,如电子支付

- 专用的操作系统: - 有些智能卡是面向Java的,ROM中有Java虚拟机解释器

- Java小程序被下载到卡中并由JVM解释器解释

- 有些卡可以同时处理多个Java小程序,需要多道程序调度

- 资源管理和保护由卡上的操作系统处理

- 通信方式: - 在读写器与智能卡之间通过"命令-响应对"方式进行通信和控制

- 读写器发出操作命令,智能卡接收命令

- 操作系统对命令解释,完成命令的解密与校验

- 操作系统调用相应程序进行数据处理,产生应答信息,加密后送给读写器

## 7. 常见问题解答

### 7.1 Buffer Cache的作用和工作原理

在UNIX操作系统中,文件子系统与块设备之间的Buffer Cache:- 作用: - 减少对物理设备的访问次数,提高I/O性能

- 协调CPU与I/O设备之间的速度差异

- 支持数据的读写缓冲,实现数据共享

- 工作原理: - 当进程请求读取数据时,系统首先检查Buffer Cache中是否有所需数据

- 如果有(命中),直接从Buffer Cache返回数据

- 如果没有(未命中),从物理设备读取数据到Buffer Cache,再返回给进程

- 写操作时,数据先写入Buffer Cache,再由系统决定何时写回物理设备

### 7.2 UNIX的名字来历(猜测)

UNIX名字可能来源于MULTICS(Multiplexed Information and Computing Service)的简化和双关语:- MULTICS是一个复杂的分时系统

- UNIX(Uniplexed Information and Computing Service)表示简化版的MULTICS

- 也有说法是"UNIX"是"eUNuchs"(阉割版的MULTICS)的变体

### 7.3 批处理操作系统的区别

- 单道批处理:一次只能执行一个作业,作业按顺序依次执行,前一个作业完成后才能执行下一个

- 多道批处理:内存中同时存放多个作业,CPU在这些作业之间切换执行,提高了CPU利用率和系统吞吐量

### 7.4 SPOOLing技术的现状

SPOOLing技术并未过时,在现代计算机系统中仍有广泛应用:- 打印系统:现代操作系统的打印队列管理

- 邮件系统:电子邮件的发送和接收过程

- 后台作业处理:批处理任务的调度和执行

- 数据库系统:事务处理和日志管理

### 7.5 传统分时系统在今天的意义

传统分时系统在今天仍有重要意义:- 多用户支持:现代操作系统继承了分时系统的多用户并发访问能力

- 资源共享:分时思想是云计算、虚拟化等现代技术的基础

- 交互式计算:分时系统建立的交互式计算模式仍是现代系统的核心特性

- 公平调度:分时系统的时间片轮转调度思想仍被广泛应用

### 7.6 操作系统的主要作用和典型特征

操作系统的主要作用:- 管理计算机硬件和软件资源

- 为应用程序提供统一的服务接口

- 实现人机交互,提供用户界面

- 提高系统资源利用率

- 保护系统安全和数据完整性

操作系统的典型特征(与其他软件相比):- 并发性:能够同时运行多个程序

- 共享性:多个用户/程序共享系统资源

- 虚拟性:将物理资源抽象为逻辑资源

- 异步性:程序执行的结果与时间有关

- 持久性:操作系统常驻内存

- 特权性:拥有对硬件的直接控制权

- 复杂性:结构复杂,功能丰富

# Note 2

## 1. 核心问题解答

1. 应用程序是如何与操作系统交互的?

- 应用程序通过系统调用与操作系统交互.系统调用提供了应用程序访问操作系统服务的接口,例如文件操作、进程管理和内存管理等.

2. 怎样理解“操作系统是由中断/异常/事件驱动的“这句话?

- 这句话的意思是操作系统的运行依赖于中断、异常和事件的触发.中断和异常是硬件或软件产生的信号,通知操作系统需要处理的事件.操作系统通过响应这些信号来管理系统资源和执行任务.

3. 中断/异常的来源有什么不同?处理方式是一样的吗?

- 中断通常由外部设备(如键盘、鼠标、网络接口等)产生,而异常通常由CPU在执行指令时检测到的错误(如除零错误、非法指令等)产生.处理方式有所不同,中断处理程序通常较为简单,主要负责响应外部设备的请求,而异常处理程序则需要更复杂的错误处理机制.

4. 回顾一下:ICS对异常的描述及分类

- ICS(计算机系统结构)将异常分为四类:陷入(Trap)、故障(Fault)、终止(Abort)和中断(Interrupt).陷入是由用户程序主动发起的系统调用,故障是可恢复的错误,终止是不可恢复的错误,中断是由外部设备发起的请求.

5. 中断/异常处理流程中,哪些工作是硬件(体系结构)负责的?哪些工作是软件(操作系统)负责的?

- 硬件负责检测中断/异常、保存当前的处理器状态、查找中断向量表并跳转到相应的中断/异常处理程序.软件负责具体的中断/异常处理逻辑,包括错误处理、资源管理和恢复系统状态等.

6. 从中断响应(硬件)到中断处理程序(软件)执行结束,计算机系统经过了哪些流程?

- 计算机系统首先由硬件检测到中断信号,保存当前处理器状态,查找中断向量表并跳转到中断处理程序.中断处理程序执行相应的处理逻辑,处理完成后恢复处理器状态,返回到中断前的执行点继续执行.

7. 操作系统初始化与中断/异常有哪些关联?

- 操作系统初始化时会设置中断向量表、初始化中断控制器、注册中断/异常处理程序等.中断/异常机制是操作系统正常运行的重要保障.

8. 什么是软件异常?它是如何工作的?

- 软件异常是由软件引发的异常情况,例如非法内存访问、除零错误等.软件异常通过硬件检测并触发相应的异常处理程序,操作系统负责处理这些异常并采取相应的措施,如终止进程、生成错误报告等.

9. X86有哪些控制和状态寄存器?所起的作用是什么?

- X86处理器有多个控制和状态寄存器,包括CR0-CR4(控制寄存器)、EFLAGS(状态寄存器)、GDTR/IDTR(全局/中断描述符表寄存器)等.控制寄存器用于控制处理器的操作模式,状态寄存器保存处理器的状态标志,描述符表寄存器用于存储全局和中断描述符表的地址.

10. X86在PentiumII 300之后提供了sysenter/sysexit指令,为什么?与int 0x80/iret有什么不同?X86-64提供的系统调用指令是什么?

- sysenter/sysexit指令提供了更高效的系统调用机制,减少了系统调用的开销.与int 0x80/iret相比,sysenter/sysexit指令不需要保存和恢复中断标志,减少了上下文切换的开销.X86-64提供的系统调用指令是syscall/sysret.

11. 关于基于x86体系结构的Linux的系统调用实现: 1. 系统调用入口程序system\_call()与中断描述符表是什么关系?与系统调用表是什么关系?

- system\_call()是系统调用的入口程序,通过中断描述符表(IDT)中的中断向量指向.系统调用表(sys\_call\_table)存储了所有系统调用的地址,system\_call()根据系统调用号查找并调用相应的系统调用处理程序.

2. 系统调用处理结束后,处理器转去执行哪个模块?

- 系统调用处理结束后,处理器会返回到用户态,继续执行被中断的用户程序.

12. 系统调用与函数/过程调用的区别是什么?系统调用与C函数调用的区别?系统调用与API的关系?

- 系统调用是操作系统提供的接口,用于应用程序请求操作系统服务.函数/过程调用是程序内部的调用机制.系统调用与C函数调用的区别在于系统调用需要从用户态切换到内核态,而C函数调用在用户态内执行.API(应用程序编程接口)是应用程序与操作系统或库函数之间的接口,系统调用是API的一部分,提供底层操作系统服务.

关键核心:ECF——异常控制流

理解ECF(异常控制流)是深入理解计算机系统和操作系统交互的关键.

1. 理解应用程序是如何与操作系统交互的: - ECF描述了应用程序在运行过程中如何通过系统调用、中断和异常与操作系统进行交互.通过理解ECF,可以更好地理解操作系统如何管理硬件资源和提供服务.

2. 编写有趣的新应用程序: - 通过掌握ECF的原理,开发者可以编写更高效、更可靠的应用程序.理解系统调用和异常处理机制,可以帮助开发者优化程序性能,并处理各种异常情况.

3. 理解并发: - ECF在并发编程中起着重要作用.通过理解中断和异常的处理流程,可以更好地设计和实现多线程、多进程的并发程序,确保程序的正确性和高效性.

4. 理解软件异常如何工作: - 软件异常是ECF的重要组成部分.通过理解软件异常的触发和处理机制,可以更好地调试和维护程序,提升程序的稳定性和安全性.

## 2. 中央处理器(CPU)

### 2.1 关于寄存器

处理器由运算器、控制器、寄存器及高速缓存构成:用户可见寄存器:- 机器语言可以直接访问

- 数据寄存器(通用寄存器)

- 地址寄存器

- 条件码寄存器:保存CPU操作结果的标记位

控制和状态寄存器:- 用于控制处理器操作,在特权级别下可访问

- 程序计数器(PC,Program Counter)

- 指令寄存器(IR,Instruction Register)

- 程序状态字(PSW,Program Status Word):存储处理器当前运行状态的关键寄存器,包含多种重要信息: - 条件码标志位:如零标志(Zero)、进位标志(Carry)、溢出标志(Overflow)等,反映算术和逻辑运算的结果

- 中断控制位:控制处理器对中断的响应方式,如中断使能/禁止标志

- 处理器模式位:指示当前CPU运行在什么特权级别(如用户态/内核态)

- 内存管理相关标志:如分页模式、虚拟内存使能等

PSW在进程上下文切换和中断处理过程中会被保存和恢复,是操作系统实现特权保护和进程隔离的核心机制.

### 2.2 操作系统的需求之一 —— 保护

#### 从操作系统的特征考虑

操作系统需要处理并发和共享资源的问题,这就提出了对系统进行保护与控制的要求.为了实现这一点,操作系统依赖于硬件机制来隔离操作系统和用户程序.

#### 硬件机制的支持

为了实现保护,硬件需要提供基本的运行机制:1. 处理器的不同运行模式: - 处理器具有不同的运行模式,每种模式下运行的指令集合不同,这些模式被称为特权级别.

- 通过特权级别,处理器可以区分内核态和用户态,从而控制哪些指令可以在特定模式下执行.

2. 特权级别: - 特权级别决定了处理器可以执行哪些指令以及访问哪些资源.

- 在高特权级别(如内核态),处理器可以执行所有指令并访问所有资源.

- 在低特权级别(如用户态),处理器只能执行非特权指令,访问受限的资源.

通过这些硬件机制,操作系统能够有效地保护自身不被用户程序破坏,同时也能控制用户程序的行为,确保系统的稳定和安全.

### 2.3 处理器的状态(模式)

现代处理器通常将CPU状态划分为两种、三种或四种,在程序状态字寄存器PSW中设置位,根据运行程序对资源和指令的使用权限设置不同的CPU状态.

例如X86架构中的EFLAGS寄存器,RISC-V的三种特权模式:- 机器模式(M模式)

- 用户模式(U模式)

- 监管模式(S模式)

### 2.4 特权指令和非特权指令

操作系统需要两种CPU状态:- 内核态(Kernel Mode):运行操作系统程序

- 用户态(User Mode):运行用户程序

特权指令:只能由操作系统使用、用户程序不能使用的指令

非特权指令:用户程序可以使用的指令

X86支持4个处理器特权级别(特权环 Ring):R0、R1、R2和R3

- R0相当于内核态,特权能力最高

- R3相当于用户态,特权能力最低

- 目前大多数基于x86处理器的操作系统只用了R0和R3两个特权级别

### 2.5 CPU状态之间的转换

- 用户态 → 内核态:唯一途径是通过中断/异常/陷入机制

- 内核态 → 用户态:通过设置程序状态字PSW

陷入指令(访管指令,supervisor call):提供给用户程序的接口,用于调用操作系统功能

例如:int,trap,syscall,sysenter/sysexit,ecall

## 3. 中断机制

中断对于操作系统的重要性就如同汽车发动机、飞机引擎的作用,操作系统是由"中断驱动"或"事件驱动"的.

主要作用:- 及时处理设备发来的中断请求

- 捕获用户程序提出的服务请求

- 防止用户程序执行过程中的破坏性活动

### 3.1 中断/异常的概念

CPU对系统发生的某个事件作出的一种反应:- CPU暂停正在执行的程序

- 保留现场后自动转去执行相应事件的处理程序

- 处理完成后返回断点,继续执行被打断的程序

特点:- 是随机发生的

- 是自动处理的

- 是可恢复的

#### 中断/异常的来源及处理方式

中断的来源:- 外部设备:如键盘、鼠标、网络接口卡等外设发出的中断信号

- 定时器:系统定时器发出的中断信号

- 其他硬件部件:如硬盘、打印机等

异常的来源:- 程序错误:如除零错误、非法指令、页面错误等

- 系统调用:用户程序通过系统调用引发的陷入

- 其他内部事件:如调试事件、断点等

处理方式:- 中断处理:中断处理程序通常是操作系统的一部分,负责响应外部设备的请求.处理过程包括保存当前CPU状态、执行中断处理程序、恢复CPU状态并返回被中断的程序.

- 异常处理:异常处理程序也由操作系统提供,负责处理程序运行过程中出现的错误或特殊事件.处理过程包括识别异常类型、执行相应的处理程序、根据异常类型决定是否返回被中断的程序或终止程序.

#### 术语演化的历史背景

- 中断的引入:为了支持CPU和设备之间的并行操作

- 当CPU启动设备进行输入/输出后,设备便可以独立工作,CPU转去处理与此次输入/输出不相关的事情；当设备完成输入/输出后,通过向CPU发中断报告此次输入/输出的结果,让CPU决定如何处理以后的事情

- 异常的引入:表示CPU执行指令时本身出现的问题

- 如算术溢出、除零、取数时的奇偶错,访存地址时越界或执行了“陷入指令”等,这时硬件改变了CPU当前的执行流程,转到相应的错误处理程序或异常处理程序或执行系统调用

### 3.2 事件

事件可分为中断(外中断)和异常(内中断,即下面三个表项):| 类别 | 原因 | 异步/同步 | 返回行为 |

| - | - | - | - |

| 中断(Interrupt)| 来自I/O设备、其他硬件部件 | 异步 | 总是返回到下一条指令 |

| 陷入(Trap)| 有意识安排的 | 同步 | 返回到下一条指令 |

| 故障(Fault)| 可恢复的错误 | 同步 | 返回到当前指令 |

| 终止(Abort)| 不可恢复的错误 | 同步 | 不会返回 |

### 3.3 中断/异常机制工作原理

中断/异常机制是现代计算机系统的核心机制之一,通过硬件和软件相互配合,使计算机系统得以充分发挥能力:硬件工作:中断/异常响应

- 捕获中断源发出的中断/异常请求

- 以一定方式响应

- 将处理器控制权交给特定的处理程序

软件工作:中断/异常处理程序

- 识别中断/异常类型

- 完成相应的处理

在每条指令执行周期的最后时刻扫描中断寄存器,查看是否有中断信号.

若无中断信号,继续执行下一条指令.

若有中断,中断硬件将该中断触发器内容按规定编码送入PSW的相应位,称为中断码,通过交换中断向量引出中断处理程序.

硬件——中断响应过程示意:1. 在每条指令执行周期的最后时刻,扫描中断寄存器.

2. 检查是否有中断信号.

- 若无中断信号,继续执行下一条指令.

- 若有中断信号,中断硬件将中断触发器内容按规定编码送入PSW的相应位,称为中断码.

3. 通过交换中断向量,引出中断处理程序.

4. 硬件将处理器控制权交给特定的中断处理程序.

5. 中断处理程序执行相应的中断处理任务.

6. 中断处理完成后,恢复CPU状态并返回被中断的程序.

### 3.4 软硬协同——中断向量表

中断向量:一个内存单元,存放中断处理程序入口地址和程序运行所需的处理机状态字

硬件执行流程按中断号/异常类型的不同,通过中断向量表转移控制权给中断处理程序

Linux中的中断向量(X86):- 0～19:不可屏蔽中断和异常

- 0:除法错误 #DE

- 1:调试异常 #DB

- 2:NMI中断

- 3:断点异常 #BP

- 4:溢出异常 #OF

- 5:边界检查异常 #BR

- 6:无效操作码异常 #UD

- 7:设备不可用异常 #NM

- 8:双重故障异常 #DF

- 9:协处理器段越界异常

- 10:无效TSS异常 #TS

- 11:段不存在异常 #NP

- 12:栈段错误 #SS

- 13:通用保护异常 #GP

- 14:页错误异常 #PF(包含COW写时复制机制的页错误)

- 15:保留

- 16:浮点异常 #MF

- 17:对齐检查异常 #AC

- 18:机器检查异常 #MC

- 19:SIMD浮点异常 #XF

- 20～31:Intel保留

- 32～127:外部中断(IRQ)

- 128(0x80):用于系统调用的可编程异常

- 129～238:外部中断

- 239:本地APIC时钟中断

- 240:本地APIC高温中断

- 241～250:Linux保留

- 251～253:处理器间中断

- 254:本地APIC错误中断

- 255:本地APIC伪中断

### 3.5 中断响应流程

1. 设备发中断信号

2. 硬件保存现场

3. 根据中断码查表

4. 把中断处理程序入口地址等推送到相应的寄存器

5. 执行中断处理程序

### 3.6 上半部和下半部处理

在 Linux 系统中,中断处理程序应该尽量短且快,以减少对正常进程调度的影响.然而,中断处理程序可能会暂时关闭中断,如果执行时间过长,可能会丢失其他设备的中断请求.为了解决这个问题,Linux 将中断过程分为上半部和下半部.

上半部用于快速处理中断,通常会暂时关闭中断请求,主要负责处理与硬件紧密相关或时间敏感的任务.下半部用于延迟处理上半部未完成的工作,一般以内核线程的方式运行.

上半部(Top Half):- 上半部是中断处理程序的第一部分,直接由硬件中断触发.

- 其主要任务是快速响应中断,处理与硬件紧密相关或时间敏感的操作.

- 上半部运行在中断上下文中,通常会暂时关闭中断,不能被阻塞,也不能进行复杂的操作.

- 典型的上半部操作包括:读取硬件寄存器、清除中断源、调度下半部等.

下半部(Bottom Half):- 下半部是中断处理程序的第二部分,通常由上半部调度执行.

- 其主要任务是延迟处理上半部未完成的工作,完成较为复杂和耗时的处理.

- 下半部运行在进程上下文中,可以被阻塞,也可以进行复杂的操作.

- 典型的下半部操作包括:数据处理、更新数据结构、唤醒等待的进程等.

例如,当网卡收到网络包后,通过 DMA 将数据写入内存,并通过硬件中断通知内核有新数据到达.内核调用中断处理程序,分为上半部和下半部.上半部会先禁止网卡中断,避免频繁硬中断降低内核效率,然后触发软中断,将耗时且复杂的任务交给软中断处理程序(下半部)处理,如解析网络数据并将其传递给应用程序.

为什么引入上半部和下半部处理?

1. 提高响应速度:上半部只执行最紧急的操作,尽量缩短中断处理时间,使系统能够快速响应其他中断.

2. 减少中断禁用时间:上半部运行在中断上下文中,系统在处理上半部时会禁用中断.通过将复杂操作移到下半部,可以减少中断禁用时间,提高系统的并发性.

3. 分离紧急和非紧急任务:将紧急任务放在上半部,非紧急任务放在下半部,有助于合理分配系统资源,提高系统的整体性能和稳定性.

所以,中断处理程序的上半部和下半部可以理解为:- 上半部直接处理硬件请求,也就是硬中断,主要是负责耗时短的工作,特点是快速执行；

- 下半部是由内核触发,也就是软中断,主要是负责上半部未完成的工作,通常都是耗时比较长的事情,特点是延迟执行.

还有一个区别,硬中断(上半部)是会打断 CPU 正在执行的任务,然后立即执行中断处理程序,而软中断(下半部)是以内核线程的方式执行,并且每一个 CPU 都对应一个软中断内核线程,名字通常为「ksoftirqd/CPU 编号」,比如 0 号 CPU 对应的软中断内核线程的名字是 ksoftirqd/0.

Ref:[软中断](https://xiaolincoding.com/os/1\_hardware/soft\_interrupt.html#什么是软中断)

### 3.7 软件——中断处理程序

设计操作系统时,为每一类中断/异常事件编好相应的处理程序,并设置好中断向量表.系统运行时若响应中断,中断硬件部件将CPU控制权转给中断处理程序:1. 保存相关寄存器信息

2. 分析中断/异常的具体原因

3. 执行对应的处理功能

4. 恢复现场,返回被事件打断的程序

## 4. IA32 体系结构对中断的支持

### 4.1 基本概念——X86处理器

中断:由硬件信号引发的,分为可屏蔽和不可屏蔽中断

异常:由指令执行引发的,比如除零异常

- 80x86处理器发布了大约20种不同的异常

- 对于某些异常,CPU会在执行异常处理程序之前产生硬件出错码,并压入内核态堆栈

系统调用:异常的一种,用户态到系统态的唯一入口

### 4.2 IA32体系结构对中断的支持

中断控制器(PIC或APIC):- 负责将硬件的中断信号转换为中断向量,引发CPU中断

实模式:中断向量表(Interrupt Vector)

- 存放中断服务程序的入口地址

- 不支持CPU运行状态切换

- 中断处理与一般的过程调用相似

保护模式:中断描述符表(Interrupt Descriptor table)

- 采用门(gate)描述符数据结构描述中断向量

- 表项包含四种类型门描述符: - 任务门(Task Gate)

- 中断门(Interrupt Gate)

- 陷阱门(Trap Gate)

- 调用门(Call Gate)

中断向量表/中断描述符表

- 表项包含四种类型门描述符: - 任务门(Task Gate)

- 中断发生时,必须取代当前进程的那个进程的TSS选择符存放在任务门中(Linux没有使用任务门)

- 中断门(Interrupt Gate)

- 给出段选择符(Segment Selector)、中断/异常程序的段内偏移量(Offset)

- 通过中断门后系统会自动禁止中断

- 陷阱门(Trap Gate)

- 与中断门类似,但通过陷阱门后系统不会自动关中断

- 调用门(Call Gate)

中断/异常的硬件处理过程:1. 确定与中断或异常关联的向量i

2. 通过IDTR寄存器找到IDT表,获得中断描述符(表中的第i个表项)

3. 从GDTR寄存器获得GDT的地址,结合中断描述符中的段选择符,在GDT表获取对应的段描述符

4. 特权级检查

5. 检查是否发生了特权级的变化,如需要则进行堆栈切换

6. 硬件压栈,保存上下文环境

7. 如果是中断,清IF位

8. 通过中断描述符中的段内偏移量和段描述符中的基地址,找到中断/异常处理程序的入口地址,执行其第一条指令

## 5. 系统调用(System call)

系统调用是用户在编程时可以调用的操作系统功能:- 系统调用是操作系统提供给编程人员的唯一接口

- 使CPU状态从用户态陷入内核态

每个操作系统都提供几百种系统调用,包括进程控制、进程通信、文件使用、目录操作、设备管理、信息维护等.

#### 经典问题:系统调用与C函数调用的区别?

1. 系统调用: - 定义:系统调用是操作系统提供给用户程序的接口,用于执行特权操作,如文件操作、进程控制、内存管理等.

- 执行环境:系统调用会导致CPU从用户态切换到内核态,执行内核中的代码.

- 实现方式:通过特定的陷入指令(如int 0x80)触发中断或异常,进入内核态执行相应的系统调用服务例程.

- 开销:由于涉及用户态到内核态的切换,系统调用的开销较大.

2. C函数调用: - 定义:C函数调用是程序内部的函数调用,用于实现特定的功能或算法.

- 执行环境:C函数调用在用户态执行,不涉及特权操作.

- 实现方式:通过函数调用指令(如call)在程序内部跳转到函数的入口地址执行.

- 开销:C函数调用的开销较小,因为不涉及用户态和内核态的切换.

例如,printf函数是一个C库函数,它最终会调用系统调用write来将数据输出到终端.printf函数本身在用户态执行,而write系统调用会切换到内核态执行实际的输出操作.

### 5.1 静态:系统调用机制的设计

机制与策略分离原则指导下的系统调用设计:1. 中断/异常机制:支持系统调用服务的实现

2. 陷入指令:引发异常,完成用户态到内核态的切换

3. 系统调用号和参数:每个系统调用都事先给定一个编号(功能号)

4. 系统调用表:存放系统调用服务例程的入口地址

### 5.2 静态:参数传递过程问题

怎样实现用户程序的参数传递给内核?常用的3种实现方法:- 由陷入指令自带参数:陷入指令的长度有限,只能自带有限的参数

- 通过通用寄存器传递参数:寄存器的个数会限制传递参数的数量

- 在内存中开辟专用堆栈区来传递参数

数据段部分:section .data

output: ascii "Hello!\n"

output\_end:equ len,output\_end - output1. section .data:定义数据段,用于存放程序中的数据

2. output::定义一个标签,表示数据的起始位置

3. ascii "Hello!\n":定义一个ASCII字符串"Hello!",后跟换行符

4. output\_end::定义另一个标签,表示数据的结束位置

5. equ len,output\_end - output:定义一个常量len,其值为output\_end和output之间的字节数,即字符串的长度

代码段部分:section .text

globl \_start

\_start: movl $4,%eax #eax存放系统调用号

movl $1,%ebx

movl $output,%ecx

movl $len,%edx

int $0x80 #引发一次系统调用

end: movl $1,%eax #1这个系统调用的作用?

movl $0,%ebx

int $0x801.

1. section .text:定义代码段,用于存放程序的指令

2. globl \_start:声明\_start标签为全局的,使链接器能够找到程序的入口点

3. \_start::程序的入口点

4. movl $4,%eax:将4放入eax寄存器,4是Linux系统调用表中write函数的调用号

5. movl $1,%ebx:将1放入ebx寄存器,1代表标准输出文件描述符

6. movl $output,%ecx:将output字符串的地址放入ecx寄存器,作为要输出的数据

7. movl $len,%edx:将len(字符串长度)放入edx寄存器

8. int $0x80:触发中断0x80,执行系统调用,这里执行的是write(1,"Hello!\n",7)

9. movl $1,%eax:将1放入eax寄存器,1是Linux系统调用表中exit函数的调用号

10. movl $0,%ebx:将0放入ebx寄存器,作为exit()的参数,表示程序正常退出(返回值为0)

11. int $0x80:再次触发中断0x80,执行系统调用exit(0),终止程序

### 5.3 动态:系统调用的执行过程

当CPU执行到特殊的陷入指令时:1. 中断/异常机制:硬件保护现场；通过查中断向量表把控制权转给系统调用总入口程序

2. 系统调用总入口程序:保存现场；将参数保存在内核堆栈里；通过查系统调用表把控制权转给相应的系统调用处理例程或内核函数

3. 执行系统调用例程

4. 恢复现场,返回用户程序

## 6. Linux系统调用实现

基于x86体系结构的Linux系统调用实现:- 陷入指令选择:int $0x80

- 门描述符:系统初始化时对IDT表中的第128号门初始化

- 门类型:15,陷阱门:陷阱门不会自动屏蔽中断,允许在处理系统调用时继续响应其他中断,提高系统的并发性和响应速度.

- DPL:3,与用户级别相同,允许用户进程使用该门描述符

### 6.2 系统执行 int $0x80 指令

1. 特权级的改变:由于从用户态切换到内核态,CPU需要切换栈.

- 用户栈切换到内核栈:CPU从任务状态段(TSS)中装入新的栈指针(SS:ESP),指向内核栈.

2. 保存用户态信息:用户栈的信息(SS:ESP)、EFLAGS、用户态CS、EIP寄存器的内容会被压栈,以便返回时使用.

- 将EFLAGS压栈后,复位TF(陷阱标志),IF(中断标志)位保持不变.

3. 查找IDT:使用128在中断描述符表(IDT)中找到对应的门描述符,从中找出段选择符装入代码段寄存器CS.

- 代码段描述符中的基地址加上陷阱门描述符中的偏移量,定位到system\_call的入口地址.

4. 特权级检查:代码只能访问相同或较低特权级的数据.

- 确保系统调用在内核态执行,防止用户态代码直接访问内核数据.

5. 系统调用号和参数传递: - 系统调用号:通过EAX寄存器传递.

- 系统调用参数:通过EBX、ECX、EDX、ESI、EDI寄存器传递.

6. 执行系统调用:根据系统调用号,查找系统调用表,找到对应的系统调用处理例程并执行.

- 处理完成后,将结果放入EAX寄存器,并通过ret\_from\_sys\_call例程返回用户态程序.

### 6.3 Linux系统调用执行流程

应用程序 → 封装例程 → 陷入处理 → 内核函数1. 用户态下调用C库的库函数,比如write()

1. 封装后的write()先做好参数传递工作,然后使用int 0x80指令产生一次异常

2. CPU通过0x80号在IDT中找到对应的服务例程system\_call(),并调用之

3. system\_call()将参数保存在内核栈；根据系统调用号索引系统调用表,找到系统调用程序入口,比如sys\_write()

4. sys\_write()执行完后,经过ret\_from\_sys\_call()例程返回用户程序

### 6.4 示例:系统调用的参数传递

系统调用使用寄存器传递参数,要传递的参数包括:- 系统调用号

- 系统调用所需的参数

用于传递参数的寄存器有:- eax用于保存系统调用号和系统调用返回值

- 系统调用参数保存在ebx,ecx,edx,esi和edi中,参数个数不超过6个

进入内核态后,system\_call再将这些参数保存在内核堆栈中.

假如C库中封装的系统调用号3的函数原型如下:movl 0x8(%esp),%ecx # 将用户态堆栈中的para2放入ecx

movl 0x4(%esp),%ebx # 将用户态堆栈中的para1放入ebx

movl $0x3,%eax # 系统调用号保存在eax中

int $0x80 # 引发系统调用

movl %eax,errno # 将结果存入全局变量errno中

movl $-1,%eax # eax置为-1,表示出错则调用时,参数传递如下:- eax = 3

- ebx = para1

- ecx = para2

# Note 3

## 1. 核心问题解答

1. 怎样理解“进程是对CPU的抽象”这句话?

\* 解答:物理CPU只有一个(或有限个),但通过多道程序设计和操作系统的进程调度,可以让多个程序在宏观上“同时”运行.操作系统为每个运行的程序创建一个进程,并管理它们对CPU的使用(分时复用).这使得每个进程都感觉自己仿佛独占了一个CPU(或一个虚拟CPU)来执行自己的指令序列.因此,进程机制将一个或多个物理CPU虚拟化成了多个虚拟CPU,供多个程序并发执行,这是对CPU计算能力的抽象.

2. 何谓进程映像?进程有实体吗?在哪里?

\* 解答:进程映像(Process Image)是指进程在执行时的完整状态描述,是进程实体的静态体现.它包括: \* 程序代码(Code Segment):进程要执行的指令.

\* 程序数据(Data Segment):程序使用的全局变量、静态变量等.

\* 程序堆栈(Stack):用于函数调用、局部变量存储.

\* 堆(Heap):动态分配内存的区域.

\* 进程控制块(PCB):包含进程的所有管理信息(状态、ID、寄存器值、资源列表等).

\* 进程是有实体的.它的实体就是进程映像所包含的内存区域(代码、数据、堆栈、堆)以及在操作系统内核中的数据结构(PCB).这些实体主要存在于 内存 中(代码、数据、堆栈、堆)和 操作系统内核空间(PCB及其相关数据结构).当进程被挂起时,部分映像可能被交换到 磁盘(交换空间)上.

3. 怎样描述进程?一个进程都有什么(组成要素)?

\* 解答:描述一个进程主要通过其 进程控制块(PCB).PCB是操作系统感知进程存在的唯一标志,包含了描述和控制进程运行所需的所有信息.

\* 一个进程的组成要素(即进程映像)包括: \* 程序代码

\* 数据集合(全局变量、静态变量、动态分配的内存)

\* 执行上下文(CPU寄存器值、程序计数器PC、程序状态字PSW、栈指针等)

\* 进程控制块(PCB)(包含进程标识符、状态、优先级、资源列表等)

4. 创建进程主要完成哪些工作?

\* 解答:创建一个进程(例如通过 fork()或 CreateProcess)主要包括: \* 分配进程标识符(PID):给新进程一个唯一的ID.

\* 创建和初始化进程控制块(PCB):分配PCB结构,并填入初始信息(如PID、父进程ID、初始状态设为New或Ready、优先级等).

\* 分配地址空间:为进程分配独立的虚拟内存空间(可能通过复制父进程空间或加载新程序).

\* 加载程序和数据:将可执行文件的代码和数据加载到进程的地址空间中(exec 的工作).

\* 初始化执行上下文:设置PC指向程序入口,初始化栈指针和寄存器.

\* 分配资源:分配进程所需的其他资源(如文件描述符,继承自父进程或新创建).

\* 状态设置与调度:将进程状态设置为就绪态(Ready),并将其链入就绪队列,等待调度器分配CPU.

5. 进程的生命周期内都会经历哪些变化?怎样表示这些变化?

\* 解答:进程在其生命周期中会经历状态的转换.基本的状态包括: \* 创建态(New):进程正在被创建.

\* 就绪态(Ready):具备运行条件,等待CPU.

\* 运行态(Running):正在CPU上执行.

\* 等待态/阻塞态(Waiting/Blocked):等待某个事件(如I/O完成)而暂停执行.

\* 终止态(Terminated):进程执行完毕或被终止,等待系统回收资源.

\* 这些变化通常用 进程状态转换图 来表示,图中的节点代表状态,有向边代表状态之间的转换及其触发条件(如调度、等待事件、事件完成等).

6. 进程有哪些状态?进程状态之间的转换(条件?操作?)

\* 解答: \* 基本状态:运行态(Running)、就绪态(Ready)、等待态(Waiting/Blocked).

\* 其他状态:创建态(New)、终止态(Terminated).还可能引入挂起态(Suspended Ready,Suspended Blocked).

\* 常见转换及条件/操作: \* New -> Ready:OS完成进程创建的必要工作,资源基本到位,允许参与调度.

\* Ready -> Running:进程被调度器(Scheduler)选中,获得CPU使用权.操作:上下文切换,恢复进程现场.

\* Running -> Ready:时间片用完；或被更高优先级的进程抢占.操作:上下文切换,保存进程现场.

\* Running -> Waiting:进程请求I/O操作或等待某一资源/事件.操作:进程主动调用阻塞原语(e.g.,wait()),保存现场,移入等待队列.

\* Waiting -> Ready:进程等待的事件发生或资源可用(如I/O完成).操作:中断处理程序或相关内核线程执行唤醒原语(e.g.,wakeup()),将进程移入就绪队列.

\* Running -> Terminated:进程正常执行完毕或出错退出.操作:进程调用退出原语(e.g.,exit()),进入终止态.

\* Terminated -> Gone:OS回收进程所占资源(PCB、内存等).

7. 进程状态转换的发生,是否一定导致另一个转换发生?

\* 解答:是的,通常是这样. 操作系统是一个动态系统,进程状态转换往往是相互关联的.

\* 例如,一个进程从 Running -> Waiting,会释放CPU,这使得调度器可以选择另一个处于 Ready 状态的进程,使其发生 Ready -> Running 的转换.

\* 一个进程从 Waiting -> Ready(如I/O完成),它进入就绪队列,可能在未来某个时刻引发 Ready -> Running 的转换(当它被调度时).

\* 一个进程 Running -> Terminated,会释放它占有的资源,这可能使得另一个 Waiting(等待该资源)的进程变为 Ready.

8. 操作系统给进程提供内存空间,该空间的地址是虚拟地址还是物理地址?为什么?

\* 解答:操作系统提供给进程的地址空间是 虚拟地址空间(Virtual Address Space).

\* 原因: \* 隔离与保护:每个进程拥有独立的虚拟地址空间,一个进程无法直接访问另一个进程的内存,提供了安全保护.

\* 地址空间扩展:虚拟地址空间可以大于物理内存,借助内存管理单元(MMU)和磁盘交换空间,给进程提供更大的可用地址范围.

\* 内存管理简化:操作系统可以更灵活地管理物理内存,例如将非连续的物理内存页映射到连续的虚拟地址空间,简化了内存分配和程序加载.

\* 程序加载和链接简化:程序可以在编译链接时确定其在虚拟地址空间的布局,而无需关心实际加载到物理内存的哪个位置.

9. 操作系统如何描述进程的地址空间?

\* 解答:操作系统内核通常使用特定的数据结构来描述进程的地址空间.例如: \* 在 Linux 中,使用 mm\_struct 结构来表示一个进程的整个地址空间.mm\_struct 内部包含一个 vm\_area\_struct(VMA)的链表或树,每个 VMA 描述了虚拟地址空间中的一个连续区域(段),包括其起止地址、访问权限(读/写/执行)、映射的文件(如果有)等信息.

\* 通过 页表(Page Tables)或 段表(Segment Tables)将虚拟地址映射到物理地址.这些表由硬件(MMU)使用,操作系统负责维护.

\* 可以通过 cat /proc/<PID>/maps 命令查看一个进程的虚拟地址空间布局和 VMA 信息.需要将 <PID> 替换为实际的进程 ID(可通过 ps 命令查找),否则会提示文件或目录不存在.

10. 为什么有了进程后又引入线程?

\* 解答:引入线程主要是为了解决进程的以下不足: \* 并发应用需求:许多应用内部包含多个并发执行的任务(如Web服务器处理多个请求,GUI程序响应用户输入同时后台处理).用多进程实现这些任务,开销较大且通信复杂.

\* 开销问题:创建进程、撤销进程、以及在进程间切换(上下文切换)都需要较大的时间和系统资源开销.线程作为“轻量级进程”,创建、销毁和切换的开销小得多.

\* 通信效率:同一进程内的线程共享地址空间和大部分资源,它们之间的通信(通过共享内存)非常高效,无需内核介入.进程间通信(IPC)通常需要内核的协调,更复杂且效率较低.

\* 性能提升:在多核处理器上,同一进程的多个线程可以真正并行执行在不同的核心上,提高应用程序的吞吐量.

11. 怎样实现线程机制?为什么有各种支持线程的方式?

\* 解答:线程机制主要有三种实现方式: \* 用户级线程(User-Level Threads,ULT):线程的管理(创建、调度、同步)完全在用户空间由一个线程库来完成.内核对线程无感知,只管理进程.

\* 优点:切换快(不需内核模式),可自定义调度算法,可移植性好.

\* 缺点:一个线程阻塞(如系统调用),整个进程会阻塞；无法利用多核并行.

\* 核心级线程(Kernel-Level Threads,KLT):线程的管理由操作系统内核完成.内核知道每个线程的存在,并进行调度.

\* 优点:一个线程阻塞不影响其他线程；可以利用多核并行.

\* 缺点:线程创建、销毁、切换需要进入内核,开销比ULT大.

\* 混合实现(Hybrid Implementation):结合了ULT和KLT.内核管理KLT,用户空间线程库将多个ULT映射到一个或多个KLT上.

\* 目标:兼具两者的优点,但实现复杂.

\* 存在多种方式的原因:是在性能、并发能力、实现复杂度、系统资源消耗之间进行权衡的结果.

\* ULT优先考虑低开销和灵活性.

\* KLT优先考虑真正的并发和对阻塞系统调用的处理.

\* 混合模型试图找到一个平衡点.

12. 线程包Pthreads中相关的函数的功能?

\* 解答:Pthreads(POSIX Threads)是一个线程API标准,提供了一系列函数来管理线程: \* pthread\_create():创建一个新的线程.

\* pthread\_exit():终止调用该函数的线程.

\* pthread\_join():等待指定的线程终止.

\* pthread\_yield():主动让出CPU,让其他线程运行.

\* pthread\_self():获取调用线程自身的线程ID.

\* pthread\_mutex\_init(),pthread\_mutex\_lock(),pthread\_mutex\_unlock(),pthread\_mutex\_destroy():互斥锁相关操作,用于保护临界区,实现线程互斥.

\* pthread\_cond\_init(),pthread\_cond\_wait(),pthread\_cond\_signal(),pthread\_cond\_broadcast(),pthread\_cond\_destroy():条件变量相关操作,用于线程间的同步(等待某个条件满足).

13. 中断/异常机制与进程线程模型的关联?

\* 解答:中断和异常是操作系统得以实现进程/线程调度和管理的关键机制.

\* 上下文切换触发:时钟中断(Timer Interrupt)使得操作系统可以剥夺当前运行进程/线程的CPU使用权(时间片用完),进行调度,切换到其他就绪的进程/线程.这是实现分时复用的基础.

\* 状态转换:I/O完成中断会通知操作系统,操作系统可以将等待该I/O的进程/线程从等待态转换为就绪态.

\* 系统调用:进程通过执行特定的指令(如 syscall 或 int 0x80)产生异常(陷阱 Trap),主动陷入内核态,请求操作系统服务(如创建进程、读写文件、阻塞等待).内核处理完请求后,可能会进行调度.

\* 错误处理:异常(如除零、缺页故障 Page Fault)也需要内核介入处理.缺页故障处理是虚拟内存管理的核心部分,可能导致进程阻塞(等待页面从磁盘调入).

\* 保存与恢复现场:发生中断/异常时,硬件和操作系统内核协作,必须保存当前进程/线程的执行上下文(寄存器、PC、状态等),处理事件后,再恢复某个进程/线程(可能是同一个,也可能是不同的)的上下文继续执行.

14. 机制和策略分离的原则在进程线程模型中的体现?

\* 有点像操作系统提供的 API 和调用这些 API 的策略之间的关系

\* 解答:机制与策略分离(Separation of Mechanism and Policy)是操作系统设计的重要原则,意指提供实现某种功能的基础能力(机制),与决定何时、如何使用这些能力的决策逻辑(策略)分开.

\* 进程/线程状态管理(机制)vs. 调度算法(策略):操作系统提供了进程/线程状态(就绪、运行、等待等)以及在它们之间转换的机制(如阻塞/唤醒原语、上下文切换).但是,选择 哪个就绪进程/线程投入运行,则是调度算法(策略)决定的(如FIFO、轮转、优先级调度等).

\* 挂起/激活(机制)vs. 负载调节(策略):操作系统提供将进程换出到磁盘(挂起)和换回内存(激活)的机制.但是,决定 何时挂起哪个进程(例如,为了降低内存压力或提高系统吞吐量),则是系统负载调节策略的一部分.

\* 线程实现(机制)vs. 应用并发模型(策略):用户级线程库提供创建和管理线程的机制.应用程序如何 利用这些线程来构建并发逻辑(例如,线程池大小、任务分配方式)则是应用层面的策略.

15. 协程是什么?为什么引入协程?协程怎么用?

\* 解答: \* 是什么:协程(Coroutine)是一种比线程更轻量级的用户态并发(或协作式多任务)实现方式.它们是可以在特定点暂停执行,并在稍后从同一点恢复执行的计算过程.协程之间的切换由程序员(或协程库/语言运行时)显式控制,通常发生在用户态,不需要内核介入.

\* 为什么引入: \* 极低的切换开销:协程切换完全在用户态进行,避免了内核态和用户态之间的切换以及内核调度,开销远小于线程切换.

\* 高并发能力:单个线程可以管理成千上万个协程,特别适合处理大量并发连接(如网络服务器)或I/O密集型任务,能有效减少线程数量和内存消耗.

\* 简化异步编程:允许使用看似同步的代码风格来编写异步逻辑(例如,使用 async/await 关键字),从而避免 回调地狱(Callback Hell).回调地狱是指在传统异步编程中,当一个操作依赖于另一个异步操作的结果时,需要将后续操作放在前一个操作的回调函数中,如果存在多层依赖,就会形成层层嵌套的回调函数结构,导致代码难以阅读、理解和维护.

\* 怎么用:通常通过编程语言或第三方库提供的支持来使用.

\* 语言原生支持:如 Python(async/await),Go(goroutine),C++20(co\_await,co\_yield,co\_return),Rust(async/await).

\* 库支持:通过特定的协程库在不支持原生协程的语言中使用.

\* 用法:开发者定义协程函数,在需要等待的操作(通常是I/O)前使用特定关键字(如 await 或 yield)暂停当前协程,让出执行权给其他协程或事件循环,当操作完成后,协程从暂停点恢复执行.

## 2. 进程模型(Process Model)

### 2.1 基本概念

1. 顺序程序与顺序环境(Sequential Program & Environment)

\* 程序(Program):指令或语句的序列,体现某种算法,是静态的.

\* 顺序环境:系统中只有一个程序在运行,独占所有资源,执行不受外界干扰.

\* 特征: \* 顺序性:指令严格按程序规定顺序执行.

\* 封闭性:程序运行时独占资源,不受外界干扰.

\* 可再现性:只要输入相同,程序执行结果总是相同,与速度无关.

2. 多道程序设计(Multiprogramming)

\* 允许多个程序 同时 进入内存并 交替 运行.

\* 目的:提高CPU利用率和系统整体效率.当一个程序等待I/O时,CPU可以切换去执行另一个程序.

3. 并发环境与并发程序(Concurrent Environment & Program)

\* 并发环境:一段时间间隔内,单处理器上有两个或以上程序同时处于开始运行但尚未结束的状态,并且执行次序不确定.宏观上并行,微观上串行(在单核CPU上).

\* 并发程序:在并发环境中执行的程序.

\* 特征: \* 间断性:程序执行走走停停(执行 -> 停 -> 执行).

\* 资源共享:多个程序可能共享系统资源(CPU、内存、I/O设备).

\* 不可再现性:由于执行走停的时机和顺序不确定,以及共享资源可能被修改,程序执行结果可能与执行速度有关,变得不可再现.

\* 独立性与制约性:程序各自独立运行,但也可能因共享资源或需要协作而相互制约.

\* 程序与计算不再一一对应:一个程序可能对应多次执行(多个进程),一次执行也可能断续完成.

4. 进程(Process)

\* 定义: \* 程序的一次执行过程.

\* 正在运行程序的抽象.

\* 操作系统进行 资源分配 和 调度 的 独立单位.

\* 具有独立功能的程序在某个数据集合上的一次运行活动.

\* 进程是对CPU的抽象:如前所述,它将物理CPU虚拟化为多个逻辑CPU.

\* 资源分配单位:系统资源(如内存、文件句柄)以进程为单位进行分配.每个进程通常拥有独立的 地址空间.

\* 调度单位:操作系统将CPU时间片调度给进程(或进程中的线程).

\* 进程与程序的区别: \* 动态 vs. 静态:进程是动态的(有生命周期),程序是静态的(文件).

\* 并发描述:进程是描述并发的基本单位,程序不能.

\* 生命周期:进程是暂时的(创建、运行、消亡),程序是相对长久的.

\* 对应关系:一个程序可以对应多个进程实例.

### 2.2 进程模型详解

1. 进程状态(Process States)

\* 三种基本状态: \* 运行态(Running):进程占有CPU,并在CPU上运行.

\* 就绪态(Ready):进程已具备运行条件(资源到位),但因无空闲CPU而等待.

\* 等待态(Waiting/Blocked):进程因等待某一事件(如I/O完成、信号量)而暂时不能运行.

\* 其他状态: \* 创建态(New):进程正在被创建,OS已分配PCB,但尚未完成所有初始化或未被批准执行.

\* 终止态(Terminated):进程已停止执行,等待OS回收资源.

\* 挂起态(Suspended):进程映像被从内存移到外存(磁盘),用于调节系统负载或用户请求.可以有 挂起就绪(Suspended Ready)和 挂起阻塞(Suspended Blocked)两种状态.

2. 进程状态转换模型(State Transition Models)

\* 三状态模型:mermaid

graph LR

Ready(就绪)-> |调度|Running(运行)

Running -> |时间片到/高优先级进程抢占|Ready

Running -> |等待事件|Waiting(阻塞)

Waiting -> |事件发生|Ready

\* 五状态模型:mermaid

graph LR

New(创建)-> |提交|Ready(就绪)

Ready -> |调度|Running(运行)

Running -> |时间片到/高优先级进程抢占|Ready

Running -> |等待事件|Waiting(阻塞)

Waiting -> |事件发生|Ready

Running -> |完成|Terminated(终止)

\* 七状态模型:mermaid

graph LR

New(创建)-> |提交|Ready(就绪)

Ready -> |调度|Running(运行)

Running -> |时间片到/高优先级进程抢占|Ready

Running -> |等待事件|Waiting(阻塞)

Waiting -> |事件发生|Ready

Running -> |完成|Terminated(终止)

Ready -> |挂起Suspend|SReady(就绪挂起)

SReady -> |激活Activate|Ready

Waiting -> |挂起Suspend|SWaiting(阻塞挂起)

SWaiting -> |事件发生|SReady

SWaiting -> |激活Activate|Waiting

New -> |提交|SReady

Running -> |挂起Suspend|SReady

\* Linux 进程状态:包括 R(TASK\_RUNNING)(运行或就绪),S(TASK\_INTERRUPTIBLE)(可中断睡眠),D(TASK\_UNINTERRUPTIBLE)(不可中断睡眠),T(TASK\_STOPPED)(停止),Z(TASK\_DEAD - ZOMBIE)(僵尸)等.其状态模型与理论模型有所差异,更贴近实现.

\* XV6 进程状态:UNUSED,USED,SLEEPING,RUNNABLE,RUNNING,ZOMBIE.这是一个简化的教学模型.

\* 不同模型的意义:体现了 机制和策略分离,基础状态转换是机制,增加挂起等状态是为了实现更复杂的内存管理和负载均衡策略.

1. 进程控制块(Process Control Block,PCB)

\* 定义:操作系统用于管理进程的核心数据结构,是进程存在的唯一标志.也称进程描述符.所有进程的PCB集合构成进程表.

\* 作用:保存进程状态、资源、上下文等信息,供OS进行调度和管理.

\* 主要内容: \* 进程描述信息:PID(唯一标识),进程名,用户ID(UID),进程组关系.

\* 进程控制信息: \* 当前状态(State).

\* 优先级(Priority).

\* CPU现场信息(Context):程序计数器(PC),各种CPU寄存器,程序状态字(PSW),栈指针(SP).这是进程切换时需要保存和恢复的关键信息.

\* 调度相关信息(如等待事件、时间片等).

\* 所拥有的资源和使用情况: \* 虚拟地址空间描述(指向页表/段表的指针).

\* 打开文件列表.

\* I/O设备信息.

\* 进程间通信与同步信息:消息队列指针,信号量等.

\* 记账信息:CPU使用时间,内存使用量等.

\* 具体实现:不同OS有不同结构,如 Linux 的 task\_struct,Windows 的 EPROCESS/KPROCESS/PEB,Solaris 的 proc\_t.真实系统中的PCB结构非常庞大复杂.

2. 进程地址空间(Process Address Space)

\* 概念:操作系统为每个进程分配的、独立的 虚拟内存 范围.是对内存的抽象.

\* 典型布局(从低地址到高地址): \* 代码段(.text):存放程序指令,通常只读.

\* 数据段(.data,.bss):存放已初始化的全局/静态变量(.data)和未初始化的全局/静态变量(.bss).

\* 堆(Heap):动态内存分配区域(malloc,new),向上增长.

\*(文件映射区/共享库):加载动态链接库、内存映射文件等.

\* 栈(Stack):存放函数参数、局部变量、返回地址等,向下增长.

\* 内核空间:每个进程地址空间的高地址部分映射到操作系统的内核空间,供系统调用和中断处理使用(用户态不可直接访问).

\* 独立性来源:每个进程有自己的页表/段表,将相同的虚拟地址映射到不同的物理内存页(或相同的只读页,如共享库代码).

\* 写时复制(Copy-on-Write,COW):fork()创建子进程时,并不立即复制整个地址空间,而是让父子进程共享物理页面,并将页面标记为只读.当任何一方尝试写入时,触发异常,内核才真正复制该页面,使其私有化.这极大地优化了 fork()的效率,特别是 fork()后立即 exec()的情况,因为exec()会替换整个地址空间,使得大部分共享页面在被写入前就已被丢弃,从而避免了不必要的复制开销.

\* COW异常的详细处理流程: 1. 写操作触发页面故障:当父进程或子进程尝试写入共享的只读页面时,CPU检测到违反内存保护,触发页面故障异常(page fault).

2. 进入内核态:CPU立即切换到内核态,保存当前上下文,并跳转到页面故障处理程序.

3. 异常处理:内核的页面故障处理程序检查故障原因,发现是COW页面的写操作.

4. 页面复制:内核为写操作进程分配一个新的物理页框,将原共享页面的内容完整复制到新页框中.

5. 页表更新:修改发起写操作的进程的页表,将相关虚拟地址映射到新分配的物理页框,并设置为可写权限.

6. 恢复执行:内核返回用户态,恢复被中断的进程执行,此时写操作可以正常进行,且不会影响另一进程的内存视图.

\* 查看:cat /proc/<PID>/maps(Linux)可以查看进程的虚拟内存区域布局.

3. 进程队列(Process Queues)

\* 操作系统通常根据进程状态将PCB组织在不同的队列中.

\* 就绪队列(Ready Queue):存放所有处于就绪态的进程PCB.调度器从中选择下一个要运行的进程.可能按优先级组织成多个队列.

\* 等待队列(Waiting Queues):可能有多个,每个队列对应一个特定的等待事件(如等待磁盘I/O、等待键盘输入、等待某个信号量).当进程等待某事件时,其PCB被移入相应的等待队列.

\* 进程状态的改变伴随着其PCB在不同队列间的移动.

### 2.3 进程控制

#### 进程控制原语

原语(Primitive)是完成某种特定功能的一段程序,具有不可分割性或不可中断性,即原语的执行必须是连续的,在执行过程中不允许被中断,也称为原子操作(Atomic).

进程控制操作完成进程各状态之间的转换,由具有特定功能的原语完成:- 进程创建原语

- 进程撤销原语

- 阻塞原语

- 唤醒原语

- 挂起原语

- 激活(解挂)原语

- 改变进程优先级原语

#### 进程的生命周期

进程创建的时机:- 系统初始化时

- 操作系统提供的服务

- 交互用户登录系统

- 由现有的进程派生出一个新进程

- 提交一个程序执行(例如,命令行)

进程终止的时机:- 正常退出(自愿的)

- 出错退出(自愿的)

- 严重错误(非自愿)

- 被其他进程杀死(非自愿)

进程终止的各种事件:- 正常结束

- 给定时限到

- 缺少内存

- 存储器出界

- 保护性出错(写只读文件)

- 算术错误

- 超出时间(进程等待超过对某事件的最大值)

- I/O 失败

- 无效指令(如试图执行数据)

- 特权指令

- 操作系统干预(如当死锁发生时)

- 父进程请求中止某一子进程

- 父进程中止(子进程也中止)

#### 进程控制操作

##### 进程的创建

进程创建的主要步骤:- 给新进程分配一个唯一标识(pid)以及进程控制块(PCB)

- 为进程分配地址空间

- 初始化进程控制块

- 设置默认值(如:状态为 New,...)

- 设置相应的队列指针(如:把新进程加到就绪队列的链表中)

- 创建或扩充其他数据结构

不同操作系统的实现:- UNIX:fork/exec

- WINDOWS:CreateProcess

##### 进程的撤销

进程撤销的主要步骤:- 结束子进程或线程

- 收回进程所占有的资源

- 关闭打开的文件

- 断开网络连接

- 回收分配的内存等

- 撤销该进程的PCB

不同操作系统的实现:- UNIX:exit

- WINDOWS:ExitProcess

##### 进程阻塞和进程唤醒

处于运行状态的进程,在其运行过程中期待某一事件发生(如等待键盘输入、等待磁盘数据传输完成、等待其它进程发送消息),当被等待的事件未发生时,由进程自己执行阻塞原语,使自己由运行态变为阻塞态.

不同操作系统的实现:- UNIX:wait

- WINDOWS:WaitForSingleObject

##### UNIX系统设计的进程控制操作

UNIX系统提供了一系列系统调用来实现进程控制:- fork():通过复制调用进程来建立新的进程,是最基本的进程建立过程

- exec():包括一系列系统调用,它们都是通过用一段新的代码覆盖原来的内存空间,实现进程执行代码的转换

- wait():提供初级的进程同步措施,能使一个进程等待,直到另外一个进程结束为止

- exit():用来终止一个进程的运行

这些系统调用之间的关联(shell、fork()、exec()、wait())体现了UNIX进程管理的设计哲学,通过简单而正交的原语组合实现复杂功能.

#### UNIX的fork()实现及优化

UNIX的fork()实现步骤:- 为子进程分配一个空闲的进程描述符(proc结构)

- 分配给子进程唯一标识pid

- 以一次一页的方式复制父进程地址空间

- 从父进程处继承共享资源,如打开的文件和当前工作目录等

- 将子进程的状态设为就绪,插入到就绪队列

- 对子进程返回标识符0

- 对父进程返回子进程的pid

优化方案:Linux的解决方案是利用存储管理模块中的"写时复制技术"COW(Copy-On-Write)对fork()进行了优化.

#### 写时复制(Copy-on-Write,COW)技术

重新审视fork函数:- 虚拟内存和内存映射解释了fork如何为每个进程提供私有地址空间

- fork()后跟exec()的常见情况的完美方法

- 为新进程创建虚拟地址空间的步骤: - 创建新的进程mm\_struct、vm\_area\_struct、页表的精确副本

- 将两个进程中的每个页面标记为只读

- 将两个进程中的每个vm\_area\_struct标记为私有COW

- 返回时,每个进程都有虚拟内存的精确副本

- 后续写入使用COW机制创建新页面

## 3. 线程模型(Thread Model)

### 3.1 线程的引入

1. 为什么引入线程?

\* 应用的需要:一个应用程序内部往往有多个并发执行流的需求.例如: \* 字处理软件:用户输入(前台线程)、后台自动保存(后台线程)、拼写检查(后台线程).

\* Web服务器:主线程监听连接,每个连接分配一个工作线程处理请求(读文件、网络发送).

\* 开销的考虑: \* 进程创建、销毁、切换的开销(时间、空间)较大.

\* 线程是轻量级的,其创建、销毁、切换开销小得多.

\* 性能的考虑: \* 通信效率:同一进程的线程共享地址空间和资源,通信(共享内存)非常高效,无需内核干预.

\* 并行计算:在多核CPU上,同一进程的多个线程可以真正并行执行.

### 3.2 线程的基本概念

1. 线程(Thread)

\* 进程内的一个 执行实体(或执行流).

\* 是 CPU调度 的基本单位.

\* 有时称为 轻量级进程(Lightweight Process,LWP).

\* 进程现在被视为 资源分配 的基本单位.

2. 线程的属性: \* 拥有独立的状态:(Running,Ready,Blocked等),需要进行状态转换管理.

\* 拥有独立的执行上下文:程序计数器(PC),寄存器集合,栈(Stack)和栈指针(SP).线程切换时保存/恢复的是这部分私有上下文.

\* 共享所在进程的资源: \* 地址空间(代码段、数据段、堆).

\* 打开的文件.

\* 全局变量.

\* 信号处理器等.

\* 可以创建、撤销、同步其他线程.

3. 多线程进程模型:一个进程包含一个PCB和多个线程控制块(Thread Control Block,TCB).所有TCB共享进程的地址空间和资源,但每个TCB有自己独立的PC、寄存器和栈.

### 3.3 线程的实现

1. 用户级线程(User-Level Threads,ULT)

\* 实现:在用户空间通过线程库实现,内核对线程无感知.线程调度由库函数完成.

\* 优点: \* 创建、销毁、切换非常快(不涉及内核模式切换).

\* 调度算法可以由应用程序定制.

\* 可以运行在不支持线程的操作系统上(只需有线程库).

\* 缺点: \* 阻塞问题:如果一个用户级线程执行了阻塞式系统调用,整个进程都会被内核阻塞,即使其他线程是就绪的.

\* 多核利用问题:内核只把CPU分配给进程,所以一个进程中的多个ULT不能在多核上并行执行.

\* 阻塞处理: \* 使用非阻塞系统调用.

\* 使用 "Jacketing" / "Wrapper" 技术:库函数在调用可能阻塞的系统调用前检查,如果会阻塞,则不调用,而是切换到另一个用户线程.

2. 核心级线程(Kernel-Level Threads,KLT)

\* 实现:线程的管理(创建、调度、同步)由操作系统内核完成.内核维护每个线程的TCB.

\* 优点: \* 一个线程阻塞不影响进程内其他线程的执行.

\* 内核可以直接调度线程,可以在多核CPU上实现真正的并行.

\* 缺点: \* 线程的创建、销毁、切换都需要进入内核态,开销比ULT大(但仍远小于进程切换).

\* 例子:Windows 线程,Linux 的 NPTL(Native POSIX Thread Library,实际上是KLT).

3. 混合模型(Hybrid Implementation)

\* 实现:内核支持KLT,用户空间线程库将多个ULT映射到少量KLT上(M:N模型).线程创建在用户态快,调度利用内核.

\* 例子:早期的 Solaris.

\* 目标:试图结合ULT的低开销和KLT的并发优势,但实现复杂,现在较少见,Linux和Windows都主要采用KLT模型.

## 4. 协程(Coroutine)

1. 为什么引入协程?

\* 为了在单线程内实现更高效率的并发,尤其是针对 I/O 密集型任务和需要管理大量连接的场景.

\* 解决线程在高并发场景下的资源消耗(内存、内核调度开销)问题.

\* 用同步的方式编写异步代码,提高可读性.

2. 协程是什么?

\* 一种 用户态的、协作式 的多任务实现.

\* 可以看作是比线程更轻量级的执行单元,由 程序员/运行时 在 用户态 控制切换.

\* 协程可以在执行过程中的特定点 暂停(yield),然后在未来从同一点 恢复(resume).

3. 协程怎么用?

\* 依赖于 编程语言或库 的支持.

\* 常见模式: \* 定义协程函数(如 Python 的 async def).

\* 在协程函数内部,遇到需要等待的操作(如异步I/O)时,使用特定关键字(如 await,yield)主动让出控制权.

\* 一个事件循环(Event Loop)或调度器负责管理协程的暂停和恢复.

\* 例子:Python asyncio,Go goroutine,C++20 coroutine,Rust async/await.

4. 纤程(Fiber)

\* Windows 操作系统提供的一种类似协程的机制,也是用户态调度的轻量级执行单元,一个线程内可以包含多个纤程.

# Note 4

## 1. 进程/线程调度问题分析

### 1.1 When and How(调度时机与原因)

#### 调度这件事儿什么时候做?做的理由有哪些?

调度时机(When):操作系统需要在特定事件发生时决定哪个进程接下来应该占用CPU.这些时机主要包括:1. 进程创建(Process Creation):当一个新进程被创建时(例如,通过fork()系统调用),需要决定是运行父进程还是子进程,或者其他进程.

2. 进程终止(Process Termination):当一个进程执行完毕或被终止时(例如,调用exit()),它占用的CPU必须分配给其他就绪的进程.

3. 进程阻塞(Process Blocking):当一个进程因等待某个事件(如I/O操作完成、等待信号量、等待用户输入等)而无法继续执行时,它会进入阻塞状态,此时调度器需要选择另一个进程来运行.

4. I/O中断发生(I/O Interrupt):当一个I/O操作完成,产生中断时,原先等待该I/O的进程可能会从阻塞态变为就绪态.这时,调度器可能需要重新评估,决定是继续运行当前进程,还是切换到刚刚变为就绪的、可能优先级更高的进程.

5. 时钟中断发生(Clock Interrupt):在分时系统中,为了防止某个进程长时间独占CPU,操作系统会设置一个定时器.当定时器中断发生时,当前运行进程的时间片(Time Slice/Quantum)可能已用完,调度器会介入,决定是继续运行该进程(如果时间片未用完或没有其他就绪进程),还是切换到另一个就绪进程(抢占式调度).

调度的理由(Why):调度的根本目的是有效、公平地管理和分配有限的CPU资源给多个并发执行的进程.具体理由包括:1. 提高CPU利用率:尽量让CPU保持忙碌状态,减少空闲时间.当一个进程等待I/O时,可以让其他就绪进程使用CPU.

2. 提高系统吞吐量(Throughput):单位时间内完成的进程数量.好的调度算法可以在满足其他目标的同时,尽可能多地完成任务.

3. 减少周转时间(Turnaround Time):指一个进程从提交到完成所花费的总时间(等待进入内存、在就绪队列等待、CPU执行、I/O执行的总和).

4. 减少等待时间(Waiting Time):指进程在就绪队列中等待CPU所花费的总时间.

5. 减少响应时间(Response Time):对于交互式系统尤其重要,指从用户发出请求到系统首次产生响应(而非完成任务)所花费的时间.

6. 确保公平性(Fairness):保证每个进程都能获得合理的CPU时间份额,防止某些进程被饿死(Starvation).

7. 满足实时性要求(Real-time Constraints):对于实时系统,调度必须保证关键任务在它们的截止时间(Deadline)之前完成.

#### 如果没有可被调度的进程,系统做什么呢?

如果当前没有用户进程或系统核心任务进程处于就绪状态(Ready State),CPU不能完全停止.操作系统通常会执行一个特殊的空闲进程(Idle Process)或称为系统空闲任务(System Idle Task).

\* 作用:这个进程拥有最低的优先级.当没有其他任何事情可做时,调度器就会选择它来运行.

\* 行为: \* 它通常执行一个无限循环.

\* 在这个循环中,它可以执行一些低优先级的系统维护任务.

\* 更重要的是,在许多架构上(如x86),它可以执行一个特殊的指令(如HLT - Halt),使CPU进入低功耗状态,直到下一个中断(如时钟中断、I/O中断)唤醒CPU.这有助于节能和降低温度.

\* 目的:确保CPU总是有事可做(即使是“等待”),并提供一个合法的状态供调度器切换,同时优化能源使用.

#### 上下文切换的过程?有哪些开销?

上下文切换(Context Switch)是指操作系统保存当前正在运行进程的状态(上下文),并加载另一个进程的状态,以便让后者开始或继续运行的过程.这是实现多任务处理的基础.

过程:1. 中断/系统调用触发:调度发生(如时间片用完、进程阻塞等).

2. 保存当前进程上下文: \* 保存程序计数器(Program Counter,PC)和其他CPU寄存器(通用寄存器、状态寄存器等)的值.这些值通常保存在该进程的进程控制块(Process Control Block,PCB)中.

\* 保存当前进程的栈指针.

\* 更新进程状态(例如,从 "Running" 变为 "Ready" 或 "Blocked").

\* 可能需要保存内存管理相关信息(如页表基址寄存器).

3. 执行调度算法:操作系统调度器代码运行,根据调度策略选择下一个要运行的进程.

4. 加载新进程上下文: \* 从选定进程的PCB中恢复其状态.

\* 加载新进程的程序计数器和CPU寄存器.

\* 恢复新进程的栈指针.

\* 更新新进程的状态(通常是从 "Ready" 变为 "Running").

\* 恢复内存管理信息(可能需要刷新TLB - Translation Lookaside Buffer).

5. 跳转执行:CPU跳转到新进程被中断时的下一条指令地址(或其入口点,如果是首次运行)开始执行.

开销:上下文切换本身并不执行任何有用的用户工作,它是一种纯粹的开销(Overhead).开销主要包括:1. 直接开销(Direct Costs): \* 保存和加载寄存器:CPU需要时间来读写寄存器和PCB.

\* 执行调度器代码:选择下一个进程也需要CPU时间.

\* 更新PCB和其他数据结构:维护进程队列等操作需要时间.

\* MMU操作:可能需要加载新的页表基址,这可能导致TLB被刷新(TLB flush),增加后续内存访问的延迟.

2. 间接开销(Indirect Costs): \* 缓存污染(Cache Pollution):当新进程开始运行时,CPU缓存(L1,L2,L3 Cache)中很可能包含的是前一个进程的数据和指令.新进程运行时会发生大量的缓存未命中(Cache Miss),需要从内存中重新加载数据,这会显著降低执行速度,直到新进程的“工作集”(Working Set)被加载到缓存中.这是上下文切换最主要的性能影响之一.

\* CPU流水线冲刷:切换可能导致CPU的指令流水线被清空和重建.

最佳实践:频繁的上下文切换会显著降低系统整体性能.因此,调度算法和系统设计(如时间片大小的选择)需要在响应时间和系统吞吐量/效率之间找到平衡.

#### 关于调度算法,我们都关心什么?

我们在评估和选择调度算法时,主要关心以下几个性能指标(Performance Metrics)或目标(Goals):\* CPU 利用率(CPU Utilization):CPU处于忙碌状态的时间百分比.越高越好,但100%可能意味着没有冗余,响应性可能变差.

\* 系统吞吐量(Throughput):单位时间内完成的进程(或作业)数量.越高越好.

\* 周转时间(Turnaround Time):从进程提交到完成的总时间.越短越好(平均周转时间、最差周转时间).

\* 等待时间(Waiting Time):进程在就绪队列中等待CPU的总时间.越短越好(平均等待时间、最差等待时间).

\* 响应时间(Response Time):从提交请求到产生第一个响应的时间(交互式系统关键).越短且越稳定越好.

\* 公平性(Fairness):每个进程获得合理的CPU份额,防止饿死.

\* 可预测性(Predictability):对于实时系统,执行时间的可预测性比平均性能更重要.

\* 满足截止时间(Meeting Deadlines):对于实时系统,这是硬性或软性要求.

\* 优先级处理(Priority Handling):系统能否有效处理不同优先级的进程.

\* 资源平衡(Resource Balance):尽量保持所有资源(CPU,I/O设备)都处于忙碌状态.

#### 不同类型的操作系统都适用同一种调度算法吗?

不适用. 不同类型的操作系统有着不同的设计目标和用户需求,因此需要采用不同的调度策略.

\* 批处理系统(Batch Systems): \* 目标:最大化吞吐量和CPU利用率,减少平均周转时间.用户通常不直接与系统交互.

\* 适用算法:先来先服务(FCFS)、最短作业优先(SJF)、最高响应比优先(HRRN).公平性和响应时间相对不重要.

\* 交互式系统(Interactive Systems / Time-Sharing Systems): \* 目标:最小化响应时间,提供良好的用户体验,兼顾公平性.

\* 适用算法:轮转法(Round Robin,RR)、优先级调度、多级队列调度(Multi-level Queue)、多级反馈队列调度(Multi-level Feedback Queue,MLFQ).

\* 实时系统(Real-Time Systems,RTOS): \* 目标:满足任务的截止时间要求,可预测性至关重要.分为硬实时(必须满足)和软实时(尽量满足).

\* 适用算法:速率单调调度(Rate Monotonic Scheduling,RMS - 用于静态优先级)、最早截止时间优先(Earliest Deadline First,EDF - 用于动态优先级)、优先级调度(配合精确的优先级分配).

#### 对于一个调度算法,应该追求什么样的目标?

一个调度算法应该追求的目标组合取决于具体的系统类型和应用场景.通常需要在多个(有时是相互冲突的)目标之间进行权衡(Trade-off).

\* 通用目标(All Systems): \* 公平性:防止饿死.

\* 策略强制:确保系统设定的策略(如优先级)得到执行.

\* 平衡:保持系统的各个部分都处于活动状态(例如,CPU密集型和I/O密集型进程交替运行).

\* 批处理系统目标: \* 高吞吐量

\* 低周转时间

\* 高CPU利用率

\* 交互式系统目标: \* 快速响应时间

\* 低响应时间方差(稳定性)

\* 满足用户期望(感觉流畅)

\* 实时系统目标: \* 满足截止时间

\* 高可预测性

最佳实践:没有“万能”的调度算法.选择或设计算法时,必须明确系统的主要目标,并接受在其他方面可能存在的不足.例如,追求极低响应时间可能会牺牲一些吞吐量.

#### 选进程时都考虑了哪些点?单一因素还是多因素?

选择下一个要运行的进程时,调度算法可能考虑单一因素或多个因素.

\* 单一因素: \* FCFS:只考虑进程到达就绪队列的时间.

\* SJF(非抢占式):只考虑预估的下一个CPU脉冲(burst)长度.

\* 简单优先级调度:只考虑静态分配的优先级.

\* 多因素:现代操作系统和更复杂的调度算法通常是多因素的.

\* 优先级调度(带老化):考虑静态优先级,但也考虑进程等待时间(老化机制,aging),提高等待过久进程的优先级以防饿死.

\* RR:考虑到达时间和时间片轮转.

\* MLFQ:考虑优先级、进程行为(CPU密集型 vs I/O密集型)、等待时间等,动态调整进程在不同队列间的移动.

\* HRRN:考虑等待时间(W)和服务时间/脉冲长度(S),计算响应比(W+S)/S.

\* Linux CFS:考虑进程的虚拟运行时间(virtual runtime),旨在给每个进程公平的CPU时间比例.它间接考虑了进程的等待时间和已运行时间.

\* Windows调度:考虑基础优先级、动态优先级提升(如完成I/O、处于前台窗口)、时间片消耗情况等.

结论:简单算法可能只关注单一因素,但为了在复杂环境中平衡多个目标(如响应时间、公平性、吞吐量),现代通用操作系统广泛使用多因素调度算法.

### 1.2 How(调度算法详解)

#### 适用批处理系统的调度算法有哪些?

主要目标是效率(吞吐量、CPU利用率)和整体完成时间(周转时间).

1. 先来先服务(First-Come,First-Served,FCFS): \* 实现:按进程到达就绪队列的顺序进行调度.使用FIFO队列.

\* 优点:简单,易于实现,公平(按到达顺序).

\* 缺点:平均等待时间可能很长,尤其当短进程排在长进程之后时(护航效应 Convoy Effect).不适合交互式系统.是非抢占式的.

2. 最短作业优先(Shortest Job First,SJF): \* 实现:选择预计CPU执行时间(下一个CPU burst)最短的进程.可以是抢占式(SRTF - Shortest Remaining Time First)或非抢占式.

\* 优点:理论上具有最优的平均等待时间和平均周转时间.

\* 缺点: \* 需要预测下一个CPU burst长度,这很难精确做到(通常基于历史数据估计).

\* 可能导致长作业饿死(Starvation),即长时间得不到CPU.

\* 非抢占式SJF不适合交互系统.抢占式SRTF开销较大.

3. 最高响应比优先(Highest Response Ratio Next,HRRN): \* 实现:非抢占式.计算每个进程的响应比 R =(等待时间 W + 服务时间 S)/ 服务时间 S,选择R最高的进程.

\* 优点:结合了FCFS和SJF的优点.短作业容易被选中(S小,R大).同时,等待时间长的进程其响应比也会增加,避免了饿死.

\* 缺点:仍需要预测服务时间S.计算响应比有额外开销.

#### 适用交互式系统的调度算法有哪些?

主要目标是提供快速响应和用户满意度.

1. 轮转法(Round Robin,RR): \* 实现:类似于FCFS,但增加了时间片和抢占.每个进程被分配一个固定的时间片(Quantum),运行时间超出时间片后会被强制切换(抢占),放回就绪队列尾部.

\* 优点:公平,响应时间相对较短(特别是对于短请求),简单.

\* 缺点: \* 性能对时间片大小非常敏感.太小则上下文切换频繁,开销大；太大则退化为FCFS,响应时间变长.

\* 平均周转时间通常比SJF长.

\* 没有考虑优先级.

\* 最佳实践:时间片大小通常选择比平均交互响应所需时间稍长,但足够短以保证多个交互进程能快速轮换.

2. 优先级调度(Priority Scheduling): \* 实现:为每个进程分配一个优先级,调度器总是选择就绪队列中优先级最高的进程.可以是抢占式或非抢占式.

\* 优点:可以明确区分重要任务和次要任务.

\* 缺点: \* 可能导致低优先级进程饿死.

\* 优先级的确定可能是个问题(静态 vs 动态).

\* 改进: \* 老化(Aging):随时间增加等待进程的优先级.

\* 动态优先级:根据进程行为(如I/O等待)调整优先级.

3. 多级队列调度(Multi-level Queue Scheduling): \* 实现:将就绪队列划分为多个独立的队列,每个队列有自己的优先级和调度算法(如:前台交互队列用RR,后台批处理队列用FCFS).进程被永久分配到一个队列.

\* 优点:可以为不同类型的进程应用不同的调度策略.开销较低.

\* 缺点:缺乏灵活性,进程无法在队列间移动.低优先级队列可能饿死.

4. 多级反馈队列调度(Multi-level Feedback Queue,MLFQ): \* 实现:允许多个队列,并且进程可以在队列之间移动.这是目前通用操作系统中最常用的调度方法之一.

\* 规则示例: \* 新进程进入最高优先级队列.

\* 如果在时间片内完成,离开系统；如果用完时间片,则降级到下一个较低优先级队列.

\* 在较低优先级队列中等待时间过长的进程可以被提升到较高优先级队列(防止饿死,即老化).

\* I/O密集型进程(经常阻塞放弃CPU)通常会停留在较高优先级队列,保证响应性.CPU密集型进程会逐渐下降到较低优先级队列.

\* 优点:非常灵活,自适应.能同时照顾到交互式和批处理式需求,兼顾响应时间、周转时间和公平性.

\* 缺点:设计和调优复杂(队列数量、各队列调度算法、时间片大小、升级降级策略).

#### 适用实时系统的调度算法有哪些?

主要目标是满足时间约束(截止时间).

1. 速率单调调度(Rate Monotonic Scheduling,RMS): \* 类型:静态优先级,抢占式.

\* 原理:周期性任务的优先级与其执行频率(速率)成正比.周期越短(频率越高),优先级越高.

\* 优点:简单,易于实现,是最佳的静态优先级调度算法(如果任务集可调度,RMS就能找到调度方案).可进行理论上的可调度性分析(例如,利用率测试).

\* 缺点:只适用于周期性任务；对任务特性有较强假设；CPU利用率上限不如动态优先级算法高.

2. 最早截止时间优先(Earliest Deadline First,EDF): \* 类型:动态优先级,抢占式.

\* 原理:当前就绪任务中,绝对截止时间最早的任务拥有最高优先级.

\* 优点:理论上是最优的动态优先级调度算法.只要任务集的总CPU利用率不超过100%,EDF就能找到调度方案(对于可抢占、独立任务等理想情况).可以处理周期性和非周期性任务.

\* 缺点:实现比RMS复杂；可能出现多米诺骨牌效应(一个任务错过截止时间可能导致后续任务也错过)；动态优先级变化导致上下文切换可能更频繁.

3. 基于优先级的抢占式调度: \* 通用方法:给实时任务分配高优先级,使用标准优先级调度器.可以通过精心设置优先级来模拟RMS或EDF的行为.常用于软实时系统或硬实时系统中与其他任务共存的情况.

\* 需要确保高优先级任务能抢占低优先级任务,并且优先级反转(Priority Inversion)问题得到处理(如使用优先级继承 Priority Inheritance 或优先级天花板 Priority Ceiling Protocol).

#### 怎样理解抢占式和非抢占式?

这是调度器决定何时进行调度的两种基本模式.

\* 非抢占式调度(Non-preemptive / Cooperative Scheduling): \* 定义:一旦CPU分配给某个进程,该进程将一直运行,直到它主动放弃CPU(完成任务、阻塞等待I/O、或显式调用yield).调度器不能在进程运行中途强制剥夺其CPU使用权.

\* 优点:实现简单；上下文切换只在进程自愿放弃CPU时发生,开销相对较小；不会有并发访问内核数据结构的竞争问题(在单处理器上).

\* 缺点:一个长时间运行或行为不当的进程可以独占CPU,导致其他进程(特别是需要快速响应的交互式进程)长时间等待,响应性差；不适合分时和实时系统.

\* 例子:早期的Windows(如Windows 3.1),早期的Mac OS,某些简单的嵌入式系统.

\* 抢占式调度(Preemptive Scheduling): \* 定义:操作系统可以强制暂停当前正在运行的进程(即使它并未主动放弃CPU),并将CPU分配给另一个进程.这种抢占通常发生在时钟中断(时间片用完)或更高优先级进程变为就绪时.

\* 优点:能够保证CPU在进程间公平分配；提供更好的响应时间；可以有效处理优先级,防止低优先级任务阻塞高优先级任务(前提是优先级设置合理)；是现代多任务操作系统的标准做法.

\* 缺点:实现更复杂；上下文切换更频繁,开销更大；需要处理内核数据结构的并发访问问题(需要锁或其他同步机制).

\* 例子:Unix/Linux,Windows NT及后续版本,macOS,大多数现代RTOS.

#### 从哪几方面对调度算法进行比较?

主要从前面提到的性能指标/目标来进行比较和评估:1. CPU利用率:哪个算法更能让CPU保持忙碌?

2. 吞吐量:哪个算法单位时间内能完成更多任务?

3. 周转时间(平均/最坏):哪个算法下进程从提交到完成更快?

4. 等待时间(平均/最坏):哪个算法下进程在就绪队列中等待的时间更短?

5. 响应时间(平均/方差):哪个算法对交互式请求的响应更快、更稳定?

6. 公平性:哪个算法更能保证所有进程获得合理的CPU时间,避免饿死?

7. 可预测性/满足截止时间:对于实时系统,哪个算法更能保证任务按时完成?

8. 算法开销:算法本身的计算复杂度以及它导致的上下文切换频率和开销如何?

9. 实现复杂度:算法是否容易实现和调试?

10. 对参数的敏感性:算法性能是否严重依赖于某些参数(如时间片大小、优先级设置)?

比较方法:\* 确定性建模(Deterministic Modeling):给定一组特定的进程及其属性(到达时间、CPU burst),模拟运行不同算法,计算性能指标.简单但只反映特定场景.

\* 排队论建模(Queueing Models):使用数学方法(基于概率分布描述进程到达和CPU burst)来分析平均性能.能提供理论洞察但模型可能简化现实.

\* 模拟(Simulation):编写程序模拟操作系统调度行为,使用随机生成的进程或真实系统负载的轨迹(trace)作为输入.灵活且能反映动态行为,是常用的评估方法.

\* 实际系统测量(Implementation & Measurement):在真实操作系统中实现算法,运行基准测试(Benchmark)或实际负载进行测量.最准确但成本最高.

#### 机制和策略分离的原则在调度算法中的应用

机制与策略分离(Separation of Mechanism and Policy)是一个重要的操作系统设计原则,也适用于调度.

\* 机制(Mechanism):提供如何做(How)的基础能力或工具.在调度中,机制包括: \* 上下文切换的代码(保存/加载寄存器、切换页表).

\* 维护进程状态(就绪、运行、阻塞)和PCB的数据结构.

\* 管理就绪队列(如链表、优先级队列、红黑树).

\* 时钟中断处理程序.

\* 提供设置和读取进程优先级的接口.

\* 进程挂起和唤醒的原子操作.

\* 策略(Policy):决定做什么(What)或何时做(When).在调度中,策略是指具体的调度算法逻辑: \* 如何选择下一个运行的进程(FCFS规则?SJF规则?RR规则?优先级规则?).

\* 时间片长度是多少?

\* 优先级如何确定?是静态还是动态调整?如何调整?

\* 何时进行抢占?

\* 如何处理不同类型的进程(前台/后台,实时/普通)?

应用与好处:1. 灵活性与可扩展性:将调度算法(策略)与底层的上下文切换等(机制)分开,使得修改或更换调度策略更加容易,而无需改变底层的核心机制代码.例如,Linux内核允许通过sched\_setscheduler()系统调用为进程选择不同的调度策略(如SCHED\_FIFO,SCHED\_RR,SCHED\_NORMAL/CFS),但它们都使用相同的底层上下文切换机制.

2. 模块化:代码结构更清晰,职责分明.调度策略模块可以独立开发、测试和更新.

3. 可定制性:用户或管理员可以更容易地根据特定需求调整调度策略参数,甚至在某些系统中插入自定义的调度模块.

例子:调度器的主循环可能是一个通用的框架(机制),它调用一个函数(策略)来选择下一个进程.不同的调度算法可以通过实现这个选择函数来插入.上下文切换函数是另一个独立的机制.优先级队列的实现(如使用堆或链表)是机制,而如何利用这个队列(是按优先级取还是按FIFO取)是策略.

#### 实例操作系统的调度算法都是什么?

现代通用操作系统通常采用复杂且混合的调度策略,以平衡各种需求.

1. Linux: \* 主要调度器(针对普通进程):完全公平调度器(Completely Fair Scheduler,CFS),自内核2.6.23起引入.

\* 目标:为所有运行中的任务提供尽可能公平的CPU时间份额.

\* 机制:不再基于固定时间片,而是维护每个任务的虚拟运行时间(vruntime).总是选择vruntime最小的任务来运行.任务运行会增加其vruntime.I/O等待的任务vruntime增长慢,因此返回时更容易被选中.

\* 实现:使用红黑树来高效地找到vruntime最小的任务.

\* 实时调度策略: \* SCHED\_FIFO:静态优先级的先来先服务(非时间片轮转).相同优先级的任务按到达顺序执行,直到阻塞、退出或被更高优先级抢占.

\* SCHED\_RR:静态优先级的轮转法.同SCHED\_FIFO,但增加了时间片,同一优先级任务轮流运行.

\* 优先级:实时任务优先级高于普通任务.普通任务也有优先级(nice值),但CFS主要通过vruntime实现公平性,nice值影响vruntime增长的速度.

2. Windows(NT内核及以后): \* 采用基于优先级的抢占式调度算法,结合了多级反馈队列的思想.

\* 优先级:分为32个优先级.0为系统空闲线程,1-15为可变优先级类(Variable Priority Classes),16-31为实时优先级类(Real-time Priority Classes).内核线程可能使用更高的内部优先级.

\* 动态调整: \* 对于可变优先级类,系统会动态提升线程的优先级(Priority Boost),例如: \* 当线程完成I/O操作时.

\* 当等待事件/信号量被满足时.

\* 前台窗口的线程优先级通常会被提升,以改善交互响应.

\* 短时间片用完后,优先级可能会暂时降低.

\* 长时间消耗CPU的线程其动态优先级会逐渐衰减回基础优先级.

\* 时间片:不同优先级的线程可能有不同的时间片长度.前台进程的时间片通常比后台进程长且可变.

\* Quantum:Windows中称为Quantum,不是固定的,可以根据系统设置和前后台状态调整.

3. macOS: \* 基于XNU内核(混合了Mach微内核和BSD Unix).

\* 调度也采用基于优先级的抢占式模型,具有多级反馈特性.

\* 线程优先级分为几个主要波段(bands):内核模式、系统高优先级、用户交互(UI响应关键)、用户启动、后台任务等.

\* 动态调整:系统会根据线程的行为(如CPU使用情况、是否阻塞等待I/O、是否与用户界面交互)动态调整其优先级,类似于Windows.

\* 也使用了类似Mach的线程调度原语和概念,例如时间片捐赠(Time-sharing donation)等机制来优化性能.

\* 近年来引入了服务质量(Quality of Service,QoS)类的概念,让开发者可以指定任务的意图(如用户交互、后台数据处理、维护任务),系统据此进行更智能的资源(包括CPU调度)管理.

总结:现代主流操作系统都使用抢占式、基于优先级的调度框架,并结合动态优先级调整、多级队列/反馈机制,以及针对公平性(如Linux CFS)或服务质量(如macOS QoS)的特定优化,以适应通用计算环境中复杂多变的需求.

## 2. 处理器调度的基本概念

### 2.1 调度的三个层次

操作系统中的调度可以发生在不同层面,对应不同的资源管理和时间尺度:1. 长程调度(Long-term Scheduling / 作业调度): \* 时机:创建新进程时.

\* 决策:决定是否将新创建的进程纳入当前活跃进程集合(即是否允许进入内存和就绪队列).

\* 目标:控制系统的并发度(道数).

2. 中程调度(Medium-term Scheduling / 内存调度): \* 时机:内存资源紧张或需要优化内存使用时.

\* 决策:决定哪些进程的部分或全部从内存换出到外存(挂起),以及何时将挂起的进程换回内存.

\* 目标:提高内存利用率和系统吞吐量,通过交换(Swapping)技术实现.

3. 短程调度(Short-term Scheduling / CPU调度 / 微观调度): \* 时机:发生特定事件(如中断、系统调用、进程阻塞/唤醒、时间片用完等)后,需要选择下一个占用CPU的进程时.

\* 决策:从就绪队列中选择一个进程/线程,将CPU的使用权分配给它.

\* 频率:非常频繁,通常在毫秒级.

\* 要求:实现必须高效.

联系:这三个层次的调度相互关联,共同管理进程从创建到完成的整个生命周期及其资源使用.

### 2.2 处理器调度的定义与场景

\* 定义:控制和协调多个进程对CPU资源的竞争.

\* 场景:系统中有 N 个进程处于就绪状态,等待在 M 个CPU(M ≥ 1)上运行.

\* 任务:调度程序(内核函数)根据特定的调度算法,从就绪队列中选择一个进程,并将CPU使用权交给它.

\* Idle进程:如果就绪队列为空(没有可运行的用户或系统进程),系统会调度一个特殊的空闲进程(idle process)来运行,它通常执行一些低优先级任务(如系统监控、节能)或简单地循环等待中断.

### 2.3 调度需要解决的核心问题

1. 调度时机(When):何时进行处理器分配决策.

2. 调度算法(What):依据何种原则挑选进程/线程.

3. 调度过程(How):如何完成CPU的分配,即上下文切换.

### 2.4 CPU调度的时机

调度通常在以下事件发生后,内核处理完相应事件并准备返回用户态之前的最后时刻进行:\* 进程生命周期变化: \* 进程执行完毕并退出(exit()).

\* 进程因错误或异常而终止(abort).

\* 创建新进程(fork()).

\* 进程状态转换: \* 运行进程因等待I/O或资源而进入阻塞态(wait()).

\* 阻塞进程被唤醒,回到就绪态(I/O完成中断).

\* 运行进程用完分配的时间片,回到就绪态(时钟中断).

\* 运行进程主动放弃 CPU(yield()).

\* 中断: \* I/O 中断.

\* 时钟中断(用于时间片、计时器).

\* 系统调用返回前.

\* 异常处理后.

流程:事件发生 → 暂停当前进程 → 硬件响应 → 进入内核处理事件 → 事件处理结束(可能导致进程状态变化、就绪队列调整)→ 执行进程调度 → 选择新进程运行.

### 2.5 调度过程:上下文切换(Context Switching)

\* 定义:将CPU的控制权从一个进程(或线程)转移给另一个进程(或线程)的过程.这涉及到保存当前进程的状态并加载新进程的状态.

\* 上下文(Context):进程运行时,其执行状态(硬件上下文)保存在CPU的寄存器中(如程序计数器PC,程序状态字PSW,栈指针SP,通用寄存器等).进程不运行时,这些信息保存在其进程控制块(PCB)中.

\* 主要工作: 1. 切换地址空间:修改页目录寄存器(如CR3 on x86)以指向新进程的页表,加载新的虚拟地址空间.

2. 切换内核栈和硬件上下文: \* 保存当前进程的寄存器值到其PCB或内核栈.

\* 从新进程的PCB或内核栈中恢复其寄存器值到CPU.

\* 内核栈(Kernel Stack): \* 每个进程都有自己的内核栈,用于在进程执行内核代码时存储函数调用、局部变量和上下文信息.

\* 当进程从用户态切换到内核态(如系统调用、中断)时,CPU会自动切换到该进程的内核栈.

\* 内核栈位于内核地址空间,对用户程序不可见,大小通常是固定的(如 Linux 中为 8KB 或 16KB).

\* 内核栈的地址通常保存在进程的 PCB 中,在上下文切换时需要更新相关寄存器(如栈指针)指向新进程的内核栈.

\* 具体步骤(进程A切换到进程B): 1. 保存进程A的硬件上下文(寄存器值).

2. 更新进程A的PCB(如状态改为就绪或阻塞,记录PC等).

3. 将进程A移入相应的队列(就绪队列、等待队列).

4. 选择进程B作为下一个运行进程.

5. 更新进程B的PCB(状态改为运行).

6. 加载进程B的上下文(恢复寄存器值,切换地址空间).

7. 开始执行进程B.

\* XV6 Context Switch Example(swtch.S):(此处展示汇编代码逻辑,具体代码略)

\* swtch 函数接受两个参数:旧进程上下文指针(old)和新进程上下文指针(new).

\* 它负责保存 old 进程的callee-saved寄存器到其上下文结构中.

\* 然后,从 new 进程的上下文结构中恢复callee-saved寄存器.

\* 最后,通过 ret 指令返回,此时CPU将跳转到 new 进程之前保存的PC地址,effectively switching execution flow. The key is switching the stack pointer.

\* 上下文切换开销(Cost): \* 直接开销:内核执行切换操作所花费的CPU时间.

\* 保存和恢复寄存器.

\* 切换地址空间(TLB Flush相关指令通常较昂贵).

\* 执行调度算法本身的代码.

\* 间接开销:切换导致缓存性能下降.

\* CPU Cache 失效:新进程的代码和数据不在缓存中,需要从内存加载.

\* TLB(Translation Lookaside Buffer)失效:地址翻译缓存失效,需要重新查询页表.

\* 缓冲区缓存(Buffer Cache)可能失效:文件系统相关的缓存可能对新进程无效.

## 3. 处理器调度算法的设计

### 3.1 不同操作系统类型的调度目标

调度算法的选择与操作系统的主要应用场景和目标密切相关:\* 批处理系统(Batch Systems): \* 特点:通常运行长任务,无需用户交互.

\* 目标: \* 高吞吐量(Throughput):单位时间内完成的作业数量最大化.

\* 短周转时间(Turnaround Time):作业从提交到完成的总时间最小化.

\* 高CPU利用率(CPU Utilization):让CPU尽可能处于忙碌状态.

\* 交互式系统(Interactive Systems): \* 特点:需要频繁与用户交互,用户等待输入.

\* 目标: \* 快速响应时间(Response Time):从用户输入命令到系统首次给出反馈的时间要短(通常要求低于50-150ms).

\* 均衡性(Proportionality):用户感觉系统性能稳定,符合预期.

\* 实时系统(Real-time Systems): \* 特点:任务有严格的时间限制(截止时间,Deadline).

\* 目标: \* 满足最后期限(Meeting Deadlines):关键任务必须在规定时间内完成(硬实时)或尽可能满足(软实时).

\* 可预测性(Predictability):系统行为在时间上是确定的.

### 3.2 调度算法的设计考量

设计调度算法时,需在多个目标之间进行权衡(Trade-off):\* 用户角度(User-oriented): \* 周转时间(Turnaround Time):T(completion)- T(arrival).进程从进入系统到完成的总时间.目标:最小化平均周转时间.

\* 响应时间(Response Time):从请求发出到第一次产生响应的时间.目标:最小化响应时间(对交互式系统尤为重要).

\* 最后期限(Deadline):实时任务必须在规定时间前完成.目标:确保满足所有(硬实时)或重要(软实时)的截止时间.

\* 可预测性(Predictability):任务运行时间稳定,尤其对实时系统.

\* 系统角度(System-oriented): \* 吞吐量(Throughput):单位时间内完成的进程数量.目标:最大化吞吐量.

\* CPU 利用率(CPU Utilization):CPU忙于执行有效工作的时间百分比.目标:最大化CPU利用率.

\* 公平性(Fairness):各进程获得合理的CPU时间份额,防止饥饿.

\* 均衡性(Balance):系统资源(CPU,I/O设备等)应保持忙碌,充分利用.

\* 强制优先级(Enforcing Priorities):确保高优先级进程优先获得服务.

### 3.3 调度算法的关键决策点

设计或选择调度算法时,需要考虑以下几个方面:1. 进程优先级(Priority): \* 优先数:用于表示优先级的数值(数值越大优先级越高或越低,取决于系统定义).

\* 静态优先级(Static Priority):进程创建时指定,运行期间不变.简单,但可能不适应进程行为变化.

\* 动态优先级(Dynamic Priority):进程优先级在运行过程中可以调整.例如,可以提升长时间等待的进程的优先级(老化,Aging),或降低长时间占用CPU进程的优先级.更能适应系统变化.

\* PCB记录:PCB中需要包含优先级信息.

2. 就绪队列组织(Ready Queue Organization): \* 单一队列:所有就绪进程放在一个队列中,按某种顺序(如FCFS、优先级)排列.

\* 多级队列(Multiple Queues):按进程属性(如优先级、类型)划分多个队列.不同队列可采用不同调度策略.

\* 按优先级排队:每个优先级一个队列.调度器先服务高优先级队列.

\* 按类型排队:如前台(交互)进程队列、后台(批处理)进程队列.

3. 抢占 vs. 非抢占(Preemptive vs. Non-preemptive): \* 非抢占式(Non-preemptive / 不可剥夺):一旦进程获得CPU,它将一直运行,直到它自愿放弃(完成、阻塞、yield).适用于批处理,简单,但响应性差.

\* 抢占式(Preemptive / 可剥夺):当前运行的进程可以被更高优先级的就绪进程或时钟中断强制中断,CPU被分配给新进程.适用于交互式和实时系统,响应性好,但有上下文切换开销.

4. I/O密集型 vs. CPU密集型进程(I/O-bound vs. CPU-bound): \* I/O密集型:进程大部分时间在等待I/O操作完成,CPU计算时间短.

\* CPU密集型(计算密集型):进程大部分时间在进行CPU计算,很少I/O操作.

\* 调度倾向:现代系统通常倾向于优先调度I/O密集型进程,以保持I/O设备忙碌,提高系统整体吞吐量和响应性.让I/O进程尽快发出下一个I/O请求,然后在其等待时运行CPU密集型进程.

5. 时间片(Time Slice / Quantum): \* 定义:在抢占式调度(特别是轮转RR)中,分配给进程一次连续运行的最大CPU时间.

\* 选择:时间片大小的选择是一个重要的权衡: \* 太长:接近非抢占,长任务会阻塞短任务,交互式响应变慢.退化为FCFS.

\* 太短:频繁发生上下文切换,系统开销增大,有效工作比例下降.

\* 合适的大小:通常需要在几十到几百毫秒之间,取决于系统负载、CPU速度、上下文切换开销和对响应时间的要求.应略大于典型的一次交互所需CPU时间.

\* 固定 vs. 可变:时间片可以是固定的,也可以根据进程优先级或行为动态调整.

## 4. 典型的处理器调度算法

### 4.1 批处理系统调度算法

主要目标:高吞吐量、低周转时间、高CPU利用率.

\* 先来先服务(FCFS - First Come First Serve): \* 策略:按进程到达就绪队列的顺序进行调度.非抢占式.

\* 优点:公平(按到达顺序)、简单易实现.

\* 缺点: \* 平均周转时间和平均等待时间可能很长,特别是当短进程排在长进程之后时(护航效应,Convoy Effect).

\* 不利于I/O密集型进程(长CPU进程运行时,I/O进程等待；I/O进程运行时,CPU空闲).

\* 对交互式用户不友好.

\* 例子:进程P1(24s),P2(3s),P3(3s)按 P1,P2,P3 顺序到达.

\* 执行顺序:P1 -> P2 -> P3

\* 完成时间:P1(24),P2(27),P3(30)

\* 周转时间:P1(24),P2(27),P3(30)-> 平均 27s

\* 若按 P2,P3,P1 顺序调度: \* 执行顺序:P2 -> P3 -> P1

\* 完成时间:P2(3),P3(6),P1(30)

\* 周转时间:P2(3),P3(6),P1(30)-> 平均 13s(显著改善)

\* 最短作业优先(SJF - Shortest Job First): \* 策略:选择预计运行时间最短的进程投入运行.

\* 版本: \* 非抢占式 SJF:当前进程一直运行直到结束或阻塞.

\* 抢占式 SJF(最短剩余时间优先,SRTN - Shortest Remaining Time Next):当一个新进程到达,其预计总运行时间比当前进程的 \*剩余\* 运行时间还短时,抢占当前进程.

\* 优点:理论上可证明,在所有进程同时到达时,SJF(非抢占)具有最低的平均周转时间.SRTN通常比非抢占SJF的平均周转时间更短.

\* 缺点: \* 需要预测未来:如何准确知道进程的运行时间?通常基于历史数据进行估计,可能不准.

\* 饥饿(Starvation):长进程可能永远得不到CPU,如果总有短进程到来.

\* 不公平:明显偏袒短进程.

\* 例子(SRTN): | 进程 | 到达时刻 | 运行时间 |

| :- | :- | :- |

| P1 | 0 | 7 |

| P2 | 2 | 4 |

| P3 | 4 | 1 |

| P4 | 5 | 4 |

\* 0:P1 运行(剩余 7)

\* 2:P2 到达(剩余 4)< P1(剩余 5),P2 抢占 P1.P2 运行(剩余 4)

\* 4:P3 到达(剩余 1)< P2(剩余 2),P3 抢占 P2.P3 运行(剩余 1)

\* 5:P3 完成.P4 到达(剩余 4).比较 P1(剩余 5),P2(剩余 2),P4(剩余 4).P2 剩余时间最短,P2 运行(剩余 2)

\* 7:P2 完成.比较 P1(剩余 5),P4(剩余 4).P4 剩余时间最短,P4 运行(剩余 4)

\* 11:P4 完成.只剩 P1,P1 运行(剩余 5)

\* 16:P1 完成.

\* 执行序列:P1(0-2)-> P2(2-4)-> P3(4-5)-> P2(5-7)-> P4(7-11)-> P1(11-16)

\* 最高响应比优先(HRRN - Highest Response Ratio Next): \* 策略:综合考虑等待时间和运行时间,选择响应比最高的进程.非抢占式.

\* 响应比 R =(等待时间 + 预计运行时间)/ 预计运行时间 = 1 +(等待时间 / 预计运行时间)

\* 优点: \* 试图在SJF和FCFS之间取得平衡.

\* 短进程:预计运行时间小,响应比增长快,容易被选中(类似SJF).

\* 长进程:等待时间足够长后,响应比会提高,最终能获得CPU,避免了饥饿.

\* 缺点:仍需预测运行时间.计算响应比有额外开销.

\* 抢占式HRRN? 理论上可以,但每次事件(如新进程到达)都需要重新计算所有就绪进程的响应比并排序,开销较大.

### 4.2 交互式系统调度算法

主要目标:快速响应时间、均衡性、公平性.

\* 时间片轮转(RR - Round Robin): \* 策略:将所有就绪进程按FCFS排成队列.调度器选择队首进程,分配一个时间片(quantum).进程用完时间片后,若未完成或阻塞,则移到队尾.抢占式.

\* 优点: \* 公平:每个进程都能获得运行机会.

\* 响应时间快:短进程能较快完成或得到响应.非常适合分时系统.

\* 缺点: \* 上下文切换开销:时间片过短会导致开销过大.

\* 性能与时间片长度密切相关.

\* 对周转时间不一定最优.对于运行时间相近的进程,RR的平均周转时间可能比FCFS差.

\* 例子(时间片 q=20):P1(53),P2(8),P3(68),P4(24)

\* 执行序列:P1(0-20)-> P2(20-28)-> P3(28-48)-> P4(48-68)-> P1(68-88)-> P3(88-108)-> P4(108-112)-> P1(112-125)-> P3(125-145)-> P3(145-153)

\* 平均等待时间 =((68-20)+(112-88)+(20-0)+(28-0)+(88-48)+(125-108)+(48-0)+(108-68))/ 4 =(72 + 20 + 85 + 88)/ 4 = 66.25 ms(假设到达时间为0)

虚拟轮转(Virtual RR - VRR): \* 动机:RR对I/O密集型进程可能不公平.I/O进程经常在时间片未用完时就阻塞,返回就绪队列时排在队尾,下次获得CPU可能要等很久.

\* 策略:维护一个辅助就绪队列(如AUX队列).当一个进程因I/O阻塞完成而返回时,不放入主RR队列尾部,而是放入AUX队列头部.调度器优先检查AUX队列,若非空则调度AUX队首进程,给其一个较短的时间片(通常是其上次阻塞时剩余的时间片)；若AUX队列为空,则按标准RR调度主队列.从AUX队列运行完时间片的进程回到主RR队列尾部.

\* 目标:给I/O密集型进程更多机会运行,提高I/O设备利用率.

\* 优先级调度(Priority Scheduling): \* 策略:选择就绪队列中优先级最高的进程运行.可以是抢占式或非抢占式.

\* 优点:实现简单,能满足不同进程的紧急程度需求(如系统进程>用户进程,前台>后台,I/O型>CPU型).

\* 缺点: \* 饥饿:低优先级进程可能永远无法运行.

\* 优先级反转(Priority Inversion):一个低优先级进程持有高优先级进程所需的资源(如锁),导致高优先级进程被迫等待低优先级进程.更糟的是,如果此时有一个中等优先级的CPU密集型进程就绪,它会抢占低优先级进程,使得高优先级进程的等待时间变得更长甚至不可预测.

\* 优先级反转解决方案: \* 优先级继承(Priority Inheritance):当高优先级进程等待低优先级进程持有的资源时,暂时将低优先级进程的优先级提升到与高优先级进程相同,使其能尽快运行并释放资源.

\* 优先级天花板协议(Priority Ceiling Protocol):给每个资源预设一个优先级上限(等于可能使用该资源的所有进程中的最高优先级).当一个进程获得资源时,将其优先级提升到该资源的优先级上限.这能预防死锁并限制阻塞时间.

\* 中断禁止:在临界区执行期间禁止中断(简单粗暴,在通用操作系统中通常不可取,但用于某些嵌入式或实时内核).

\* 多级队列调度(Multilevel Queue Scheduling): \* 策略:将就绪队列划分为多个独立的队列,每个队列有自己的调度算法和优先级.例如: \* 系统进程队列(最高优先级,RR 或 FCFS)

\* 交互式进程队列(中优先级,RR)

\* 批处理进程队列(最低优先级,FCFS)

\* 调度器首先处理高优先级队列中的所有进程,然后才处理次高优先级队列,以此类推.队列之间通常是抢占式的(高优先级队列进程可抢占低优先级队列进程).

\* 优点:灵活性高,可以为不同类型的进程定制调度策略.

\* 缺点:进程通常被固定分配到一个队列,缺乏灵活性；低优先级队列可能饥饿.

\* 多级反馈队列调度(Multilevel Feedback Queue Scheduling - MFQ): \* 策略:结合了多级队列和动态优先级调整.进程可以在不同队列之间移动.

\* 典型实现: 1. 设置多个优先级队列(Q0,Q1,...,Qn),优先级 Q0 > Q1 > ... > Qn.

2. 不同队列分配不同的时间片长度,优先级越高的队列时间片越短(如 Q0=q,Q1=2q,Q2=4q...).

3. 新进程进入最高优先级队列 Q0.

4. 调度器总是先运行最高非空队列中的进程,同队列内通常用RR.

5. 如果进程在一个队列中用完了其时间片但未完成,它会被 降级 到下一个较低优先级队列.

6. 如果进程在时间片未用完前因 阻塞(如等待I/O)而放弃CPU,当它再次就绪时,通常会回到 原来的 队列(或有时提升一级),以优待I/O密集型进程.(讨论点:回到原队列还是队首/队尾?提升吗?具体策略不同系统可能不同.)

7. 最低优先级队列通常采用FCFS或很长的时间片RR.

8.(可选)可以加入 老化(Aging)机制:在低优先级队列等待过久的进程可以被提升到较高优先级队列,防止饥饿.

\* 优点:非常灵活,能同时满足交互式(响应快)和批处理(吞吐量)的需求,能自动适应进程行为,是最常用的调度算法之一.

\* 缺点:设计和调优(队列数量、时间片大小、升级降级策略)比较复杂.

\* 其他交互式算法(简述): \* 公平共享调度(Fair-share Scheduling):不仅考虑单个进程,还考虑进程所属的用户或用户组,确保CPU时间在用户/组之间公平分配.

\* 保证调度(Guaranteed Scheduling):向用户承诺每个进程将获得 CPU 时间的 1/n(如果有n个进程),并跟踪进程实际获得的CPU时间,优先运行获得时间最少的进程.

\* 彩票调度(Lottery Scheduling):给每个进程分配一定数量的“彩票”,调度器随机抽取一张彩票,持有该彩票的进程获得CPU.进程持有的彩票越多,获得CPU的机会越大.优先级可以通过分配不同数量的彩票来体现.简单,易实现概率公平.

### 4.3 实时系统调度算法

主要目标:满足任务截止时间、可预测性.

\* 可调度性分析:对于周期性实时任务,需要判断系统是否能在所有任务的截止时间内完成它们.

\* 若有 m 个周期任务,任务 i 的周期为 Pi,每次执行需 Ci 的CPU时间,则一个简单的(充分非必要)可调度条件是: Σ(Ci / Pi)≤ 1(CPU利用率不超过100%)

\* 更精确的条件取决于具体算法(如RM,EDF).

\* 速率单调调度(Rate-Monotonic Scheduling - RM): \* 类型:静态优先级,抢占式.

\* 策略:任务的优先级根据其 周期(Rate)设定:周期越短(频率越高),优先级越高.

\* 适用:周期性实时任务.

\* 优点:简单,理论成熟,可进行精确的可调度性分析(Liu & Layland 条件:Σ(Ci/Pi)≤ n(2^(1/n)-1)).

\* 缺点:仅适用于周期任务,对任务集利用率上限有要求(不是100%).

\* 最早截止时间优先(Earliest Deadline First - EDF): \* 类型:动态优先级,抢占式.

\* 策略:调度器在每次调度时,选择就绪队列中 绝对截止时间(Deadline)最早的任务运行.

\* 适用:周期性和非周期性实时任务.

\* 优点:理论上是最优的动态优先级算法,只要系统总利用率 ≤ 1,EDF就能找到一个可行的调度(如果存在的话).CPU利用率上限可达100%.

\* 缺点:实现比RM复杂(需要跟踪每个任务的截止时间),可能出现瞬时过载导致多米诺骨牌效应(一个任务错过deadline可能导致后续任务都错过).

### 4.4 各种调度算法比较总结

| 调度算法 | 选择依据 | 决策模式 | 吞吐量 | 响应时间 | 开销 | 对进程影响 | 饥饿问题 |

| :- | :- | :- | :- | :- | :- | :- | :- |

| FCFS | max[w] | 非抢占 | 不强调 | 可能很差(长作业阻塞短作业)| 最小 | 对短进程/IO密集型不利 | 无 |

| RR | 固定时间片 | 抢占(时间片)| q过小则低 | 短进程好 | 较小 | 公平 | 无 |

| SJF | min[s] | 非抢占 | 高 | 短进程好 | 可能较高 | 对长进程不利 | 可能 |

| SRTN | min[s-e] | 抢占(到达时)| 高 | 好 | 可能较高 | 对长进程不利 | 可能 |

| HRRN | max[(w+s)/s] | 非抢占 | 高 | 较好 | 可能较高 | 平衡 | 无 |

| Feedback | 见算法思想 | 抢占(时间片)| 不强调 | 较好(可调优)| 可能较高 | 可优待IO密集型,可能对某些进程不利 | 可能(需老化)|

\*(表中 w:等待时间,s:总服务时间,e:已执行时间)\*

## 5. 调度中的重要原则与实践

### 5.1 机制与策略分离(Mechanism vs. Policy Separation)

\* 原则:将调度的具体实现(机制,如何进行上下文切换、如何管理队列等)与调度的决策逻辑(策略,选择哪个进程运行、优先级如何确定等)分离开.

\* 为什么?

\* 灵活性:更容易修改或替换调度策略,而无需改变底层机制.

\* 可扩展性:方便添加新的调度策略.

\* 模块化:代码结构更清晰,易于理解和维护.

\* 怎么做? 操作系统内核提供通用的调度框架(机制,如优先级队列、上下文切换函数),而具体的调度算法(策略)作为可配置或可插拔的模块实现.例如,Linux 的 sched\_class 结构就体现了这种思想.

### 5.2 线程调度(Thread Scheduling)

\* 背景:现代操作系统多数支持内核级线程.调度单元从进程变为线程.

\* 用户级线程 vs. 内核级线程: \* 用户级线程:调度由用户空间的线程库管理,内核只看到一个进程.切换快,但一个线程阻塞会导致整个进程阻塞.无法利用多核.

\* 内核级线程:调度由内核管理,每个线程有自己的上下文.切换开销比用户级线程大,但比进程切换小.一个线程阻塞不影响其他线程.可以并发运行在多核上.

\* 调度对象:内核调度器直接调度内核级线程.对于用户级线程,内核调度的是其所属的进程(或承载用户线程的内核线程LWP).

## 6. 实例:操作系统调度算法

### 6.1 典型系统采用的算法概览

\* UNIX(早期):动态优先级,基于nice值和CPU使用情况调整.

\* 5.3BSD:多级反馈队列算法.

\* Windows:基于优先级的抢占式多任务调度(细节见下).

\* Linux:抢占式调度,主要使用CFS(普通进程)和实时调度策略(实时进程)(细节见下).

\* Solaris:综合调度算法,支持多种调度类(实时、分时、交互、系统等).

### 6.2 Windows 线程调度

\* 调度单位:线程(内核级线程).

\* 核心算法:基于 动态优先级 的 抢占式 调度,结合 时间配额(Quantum)调整.

\* 就绪队列:维护多个优先级队列(0-31).系统总是选择当前最高非空优先级队列中的线程运行.

\* 同优先级调度:同一优先级队列内部,线程按 时间片轮转(RR)方式调度.

\* 多处理器:允许多个线程在不同处理器上并行运行.

\* 调度触发条件: \* 线程创建、终止.

\* 线程状态改变(运行->阻塞,阻塞->就绪,运行->就绪).

\* 线程优先级改变.

\* 线程改变其处理器亲和性(Affinity).

\* 时间片用完.

\* 主动放弃(yield).

\* 线程优先级: \* 共32个优先级级别(0-31).

\* 实时优先级(16-31):优先级固定不变.用于需要紧急响应的任务.最高.

\* 可变优先级(1-15):线程有一个基本优先级(Base Priority),其当前优先级(Current Priority)可以在此基础上动态调整(提升或降低).用于普通用户和系统线程.

\* 系统线程(1-15中的一部分):用于操作系统内部任务.

\* 零页线程(0):特殊线程,优先级最低,用于在系统空闲时将物理内存页清零.

\* 时间配额(Quantum): \* 不是绝对时间值,而是以 配额单位(quantum unit)的整数表示.系统时钟中断时递减.

\* Quantum 和 QuantumReset 记录在 KTHREAD 结构中.

\* 当线程用完时间配额: \* 如果 没有 其他同优先级或更高优先级的线程就绪,Windows会 重新分配 一个新的时间配额给该线程,让它继续运行(避免不必要的切换).

\* 如果 有 其他同优先级线程就绪,该线程移到其优先级队列的末尾,调度器选择下一个线程.

\* 如果用完时间配额 且 优先级被降低,则会被抢占.

\* 作用:调整时间配额(而非仅优先级)可以影响进程获得CPU时间的比例,而不会完全饿死其他进程.例如,给前台游戏进程更大配额,使其运行更流畅,同时后台计算任务也能获得一些CPU时间.

\* 调度数据结构: \* 每个进程有默认优先级、亲和性、时间配额.

\* 每个线程有基本优先级、当前优先级、亲和性、时间配额.

\* Dispatcher Ready List:包含32个就绪线程队列的数组.

\* KiDispatcherReadyListHead:指向就绪队列的指针数组.

\* Ready Summary(就绪位图):一个32位掩码,每一位对应一个优先级队列,指示该队列是否为空.调度器通过查找第一个置位的位(Find First Set bit,FFS)快速找到最高优先级的非空队列.

\* Idle Summary(空闲位图):(多处理器)位图,指示哪些处理器当前处于空闲状态.

\* 调度策略细节: \* 主动切换:进程自愿放弃CPU(阻塞、Yield等).

\* 抢占: \* 更高优先级的线程变为就绪.

\* 当前线程优先级降低,低于另一个就绪线程.

\* 被抢占线程放回其 原 优先级就绪队列的 队首.

\* 实时优先级线程被抢占,下次运行时获得完整时间配额.

\* 可变优先级线程被抢占,下次运行时继续执行剩余时间配额.

\* 时间配额用完: \* 优先级不降低:若队列无其他线程则重置配额继续,否则移到队尾.

\* 优先级降低:移到新优先级的队列,可能被抢占.

\* 线程优先级提升(Priority Boost): \* 目的:改善响应性、解决饥饿、提高吞吐量.仅针对可变优先级线程(1-15).

\* 触发情况: 1. I/O操作完成:临时提升等待该I/O的线程优先级,幅度由设备驱动程序建议(与设备响应要求相关),使其能快速处理数据.提升后时间配额会减1(避免不公平利用I/O提升).

2. 等待事件或信号量结束:线程优先级提升1级(不超过15),以补偿其等待时间.完成提升后的运行后,优先级会逐渐衰减回基本优先级.时间配额减1.

3. 前台进程中的线程 完成等待操作.

4. 因 窗口消息(GUI活动)而唤醒的线程.

5. 反饥饿:系统线程"平衡集管理器"(Balance Set Manager)定期扫描,将等待过久(如 > 300时钟中断)的线程优先级提升到15,并给予4倍时间配额.用完后优先级立即恢复.

\* 空闲线程(Idle Thread): \* 每个处理器核都有一个对应的空闲线程.优先级为0.

\* 当没有其他可运行线程时,调度器调度空闲线程.

\* 功能:循环检测是否有工作要做:处理挂起的中断(DPCs)、检查是否有新就绪线程、调用HAL执行电源管理(如让CPU进入低功耗状态).

### 6.3 多处理器调度(Multiprocessor Scheduling)

\* 特点:系统包含多个CPU(核),可共享负载.

\* 对称多处理(SMP - Symmetric Multiprocessing): \* 所有CPU地位平等,都可以运行内核代码和用户进程.

\* 每个CPU通常有自己的调度器实例.

\* 调度器访问共享数据结构(如就绪队列)需要同步(锁、原子操作).

\* 设计挑战: \* 进程/线程分配:决定哪个任务在哪个CPU上运行.

\* 负载均衡(Load Balancing):使各CPU负载大致均匀,避免某些CPU过载而其他CPU空闲.

\* 处理器亲和性(Processor Affinity):尽量让一个进程/线程连续在同一个CPU上运行,以利用CPU缓存(L1/L2 cache)中已加载的数据和TLB条目,减少缓存失效带来的开销.

\* 缓存一致性(Cache Coherence):硬件机制(如MESI协议)确保多个CPU缓存中共享数据的副本是一致的.

\* 进程分配策略: \* 静态进程分配(Static Assignment):进程从创建到结束都绑定在某个特定CPU上.每个CPU有自己的私有就绪队列.调度开销小,易于维护亲和性,但可能导致负载不均.

\* 动态进程分配(Dynamic Assignment):进程可以在不同CPU之间迁移.通常有一个全局共享就绪队列,或各CPU有私有队列但允许任务迁移.负载均衡好,但调度开销大(需要同步、迁移成本).

\* 多核处理器问题: \* 缓存一致性:如上所述,硬件解决.

\* 缓存亲和性:调度器需要考虑.让任务倾向于留在上次运行的核上.

\* 核间数据共享:需要高效的同步机制.

\* 负载均衡:需要策略在CPU间迁移任务.

\* 缓存亲和性 vs. 负载均衡:这是个权衡.过于强调亲和性可能导致负载失衡；过于频繁地迁移以追求负载均衡则会破坏缓存亲和性,增加开销.

\* 例子: \* 并行计算/渲染:任务与数据绑定到核心可利用缓存,但计算量不均时需迁移任务以平衡负载,导致缓存失效.

\* CDN:内容按地理位置缓存(亲和性),但负载高时请求可能路由到其他节点(破坏亲和性)以均衡负载.

\* 工作窃取(Work Stealing):一种常见的负载均衡策略.每个CPU维护一个本地任务队列(通常是双端队列).CPU优先执行自己队列的任务.当一个CPU空闲时,它会随机选择另一个CPU,并从其任务队列的 尾部 “窃取” 一个任务来执行.(被窃取的CPU从头部获取任务).

\* 优点:实现了负载均衡,同时本地任务优先执行保证了一定的缓存亲和性,分布式决策减少了中心瓶颈.

\* 缺点:仍有通信和同步开销.

\* 实例:Go语言的GMP调度器,Java的ForkJoinPool,Hadoop YARN.

### 6.4 实时调度(Real-time Scheduling)(回顾)

\* 目标:满足时间约束(截止时间),高可靠性,确定性.

\* 类型:硬实时(必须满足)vs. 软实时(尽量满足).周期性 vs. 偶发性 vs. 非周期性任务.

\* 关键参数:时间(周期、执行时间、截止时间).

\* 算法:RM(静态优先级,周期短优先),EDF(动态优先级,截止时间早优先).

### 6.5 OpenEuler 多核调度技术(简述)

\* 基础:CPU调度是为保证并发性,通过调度程序(Scheduler)按调度策略(Policy)选择进程占用CPU.

\* 算法:结合使用 FIFO,RR,优先级调度(用于实时进程).

\* 普通进程:主要采用 CFS(Completely Fair Scheduler)算法,追求公平性,基于虚拟运行时间(vruntime)按优先级比例分配CPU时间.

\* 多核调度: \* 早期单队列问题:所有CPU共享一个队列.

\* 策略一(简单RR):进程在CPU间频繁迁移,破坏缓存亲和性.

\* 策略二(带亲和性):尽量让进程留在一个CPU,但可能牺牲某些进程(如E)的公平性或导致负载失衡.

\* 多队列调度:每个CPU维护自己的就绪队列(如Q0 for CPU0,Q1 for CPU1).

\* 优点:提高缓存亲和性,减少锁竞争.

\* 问题:可能导致负载失衡(如一个队列空了,另一个还很忙).

\* 迁移线程(Migration Thread):OpenEuler 使用迁移线程解决负载不均衡.每个CPU有一个 migration/CPUID 内核线程.当检测到负载不均时(如CPU0空闲,CPU1忙),CPU0可以向CPU1的停机工作队列(stop machine workqueue)添加一个任务,唤醒CPU1的迁移线程.该线程优先级很高,会立即执行迁移任务(如将进程D从CPU1迁移到CPU0),从而实现负载均衡.

### 6.6 Linux 进程调度

\* 调度单位:线程(内核级线程,Linux中称为'进程'或'任务').

\* 进程分类与调度策略: \* 实时进程(Real-time Processes): \* 要求:调度延迟最低,立即响应.

\* 策略:SCHED\_FIFO(静态优先级,非抢占式,除非更高优先级到达或阻塞)、SCHED\_RR(静态优先级,抢占式,带时间片轮转).优先级范围 1-99.

\* 普通进程(Normal Processes): \* 包括交互式进程(需要快速响应)和批处理进程(后台运行,容忍延迟).

\* 策略:SCHED\_NORMAL(也叫 SCHED\_OTHER),SCHED\_BATCH,SCHED\_IDLE.主要由 CFS(Completely Fair Scheduler)算法管理.优先级范围 100-139(对应nice值 -20 到 +19).

\* Linux调度算法演化: \* Linux 2.4:简单 O(n)调度器.基于优先级和时间片.遍历整个运行队列找最高优先级进程.所有进程时间片用完后统一重新计算.对交互式进程通过剩余时间片补偿来提升优先级.缺点:扩展性差(高负载时慢),交互性优化不完善,非抢占内核.

\* Linux 2.6(早期):O(1)调度器(by Ingo Molnar): \* 引入 active/expired 两个优先级数组队列.调度只需 O(1)时间找到最高优先级非空队列.

\* 动态优先级基于静态优先级(nice值)和平均睡眠时间 bonus 计算,试图区分交互式/批处理.

\* 进程时间片用完后移入 expired 队列(除非是特殊情况).active 队列空后,交换 active 和 expired 指针.

\* 缺点:区分交互式的启发式规则复杂难懂且易失效,代码难维护.

\* Linux 2.6(中期):SD(Staircase Scheduler by Con Kolivas)/ RSDL(Rotating Staircase Deadline Scheduler): \* 追求公平,抛弃复杂动态优先级.

\* SD:进程用完时间片后优先级降低一级(下楼梯),到底后回到较高层并获更多时间片.交互进程睡眠时停留在高层,唤醒后响应快.

\* RSDL:引入 group quota(Tg)和 expired 数组.高优先级组用完 Tg 后整体降级(minor rotation),保证低优先级任务的可预测等待时间.时间片用完进 expired 队列.active 队列空或到底后触发 major rotation(交换 active/expired).

\* 影响:启发了CFS的公平思想.

\* Linux 2.6.23 至今:CFS(Completely Fair Scheduler by Ingo Molnar): \* 核心思想:完全公平.理想情况下,每个进程获得 1/n 的CPU时间.不再区分交互式/批处理,不再使用固定时间片.

\* 虚拟运行时间(vruntime):vruntime 记录进程的加权运行时间.vruntime 增长速度与实际运行时间成正比,与进程权重(优先级)成反比.

vruntime ≈ 实际运行时间 \*(NICE\_0\_LOAD / 进程权重)

(NICE\_0\_LOAD 是 nice=0 进程的权重).

\* 调度决策:总是选择就绪队列中 vruntime 最小 的进程运行.

\* 数据结构:使用 红黑树(Red-Black Tree)存储就绪进程,按 vruntime 排序.插入、删除、查找最小节点都是 O(log n)时间.调度器取最左节点运行.

\* 公平性实现:优先级高的进程权重高,vruntime 增长慢,更容易被选中；优先级低的进程权重低,vruntime 增长快.最终达到按权重比例分配CPU时间的效果.

\* Linux 6.6+(实验性/可选):EEVDF(Earliest Eligible Virtual Deadline First): \* 对CFS的改进,旨在解决CFS在极短任务和延迟敏感任务上的一些问题,进一步改善延迟和公平性.它结合了虚拟时间和截止时间的概念.

\* CFS 调度器详解: \* task\_struct:Linux 进程/任务描述符.

\* sched\_entity:调度实体,嵌入 task\_struct 中,包含CFS调度所需信息(如 load\_weight 权重,rb\_node 红黑树节点,vruntime 等).一个 sched\_entity 可以代表一个任务或一个任务组(用于组调度).

\* sched\_class:调度类结构体,定义了一套调度器操作函数接口(如 enqueue\_task,dequeue\_task,pick\_next\_task).CFS,RT(FIFO/RR),Idle 都有自己的 sched\_class 实现.内核按优先级顺序查询 sched\_class 来决定使用哪个调度器.

\* cfs\_rq:CFS 运行队列,每个CPU有一个.包含红黑树 tasks\_timeline 和 min\_vruntime 等信息.min\_vruntime 记录该队列中所有进程的最小 vruntime,作为新进程/唤醒进程 vruntime 计算的基准.

\* 红黑树(rb\_node,rb\_root tasks\_timeline):按 vruntime 组织就绪的 sched\_entity.

\* CFS 关键情景: \* 新进程创建(fork()): \* vruntime 初始值通常设为当前 cfs\_rq->min\_vruntime(或略大),确保新进程不会立即获得过多优势.

\* 父子 vruntime 交换? 如果设置了sysctl\_sched\_child\_runs\_first,且父子在同CPU,父vruntime < 子vruntime,则交换,让子进程优先运行.

\* 插入红黑树.

\* 检查是否需要抢占当前进程.

\* 进程唤醒(wake\_up\_process()): \* 调整 vruntime:通常设为 max(waker->vruntime,cfs\_rq->min\_vruntime - delta),其中 delta 是一个小的补偿值.确保进程不会因睡眠获得不公平优势,但也给予一定补偿使其尽快运行.

\* 插入红黑树.

\* 检查是否需要抢占当前进程(如果唤醒进程 vruntime 足够小).

\* 时钟中断(scheduler\_tick()): \* 更新当前运行进程的 vruntime(actual\_runtime \* NICE\_0\_LOAD / weight).

\* 更新 cfs\_rq->min\_vruntime.

\* 检查当前进程是否已运行超过其“理想运行时间”(基于调度周期和权重计算得出).如果是,则设置抢占标记(TIF\_NEED\_RESCHED),在中断返回前会调用 schedule().

\* 主动调度(schedule()): \* 当前进程阻塞、yield 或被标记抢占时调用.

\* 更新当前进程 vruntime.

\* 如果当前进程仍是就绪态,将其重新插入红黑树.

\* 调用 pick\_next\_task()选择下一个运行进程: \* 按优先级查询 sched\_class(RT -> CFS -> Idle).

\* CFS 中,通常选择红黑树最左节点(vruntime 最小者).

\* 特殊情况:考虑 cfs\_rq->next(上次被抢占者)和 cfs\_rq->last(刚运行完者)的缓存亲和性,可能优先选择它们.

\* 从红黑树中移除被选中进程的 sched\_entity.

\* 执行上下文切换(context\_switch()).

\* CFS 与进程状态转换图示: mermaid

graph TD

New - fork()-> Ready(Ready State:In Red-Black Tree);

Ready - schedule()selects -> Running(Running State);

Running - Block(I/O,wait)-> Blocked(Blocked State);

Blocked - Wakeup -> Ready;

Running - Timeslice Check in Tick / Preempted -> Ready;

Running - exit()-> Terminated(Terminated State);

subgraph CFS Logic

direction LR

Ready - select min vruntime -> Running;

Running - update vruntime & re-insert -> Ready;

end

# Note 5

## 1. 重要概念

### 1.1 存储体系(Memory Hierarchy)

计算机存储器按照速度、容量和成本排列成层次结构.

\* 寄存器(Registers):最快,容量最小,成本最高,直接由 CPU 访问.

\* 高速缓存(Cache - L1,L2,L3):速度快,容量较小,成本高,用于缓解 CPU 和主存之间的速度差异.

\* 内存(Main Memory / RAM):速度、容量和成本居中,是程序运行时代码和数据的主要存储区域.

\* 本地磁盘(Local Disk / Secondary Storage):速度慢,容量大,成本低,用于持久化存储,如硬盘(HDD)、固态硬盘(SSD).

\* 远程磁盘(Remote Storage):通过网络访问,速度最慢,容量可以很大.

内存管理主要关注的是主存(内存)的管理.

### 1.2 地址空间(Address Space)

\* 定义:操作系统为每个进程分配的一个独立的逻辑地址范围.进程所能“看到”和访问的地址集合.

\* 独立性:每个进程拥有自己独立的地址空间,一个进程默认不能访问另一个进程的地址空间,这是实现存储保护的基础.

\* 管理:操作系统需要管理地址空间的分配(放置 placement)、回收、分割与合并.

### 1.3 逻辑地址 vs 物理地址

\* 逻辑地址(Logical Address): \* 也称为相对地址(Relative Address)或 虚拟地址(Virtual Address).

\* 是用户程序(或 CPU 发出)使用的地址.

\* 通常从 0 开始编址,地址是相对于进程自身的地址空间.

\* 编译、汇编后生成的目标代码通常使用逻辑地址.

\* 不能直接用于在物理内存中寻址.

\* 物理地址(Physical Address): \* 也称为绝对地址(Absolute Address)或 实地址(Real Address).

\* 是内存存储单元的实际地址,硬件内存总线可以直接访问.

### 1.4 地址重定位(Address Relocation)

\* 定义:将用户程序中的逻辑地址转换为运行时可由机器直接寻址的物理地址的过程.

\* 别名:也常被称为地址转换(Address Translation)、地址变换(Address Transformation)或 地址映射(Address Mapping).这些术语基本同义,都指代从逻辑地址到物理地址的转换过程.

\* 目的:保证 CPU 执行指令时能够正确访问到物理内存单元.

\* 为什么需要?

\* 多道程序环境下,内存中有多个进程.

\* 程序加载到内存的位置通常在运行时才能确定,无法在编译时预知其物理地址.

\* 地址绑定时机(Binding Time):指令和数据绑定到内存地址的时间点: \* 编译时(Compile Time):如果编译时就知道程序将驻留在内存的哪个位置,编译器可以直接生成绝对代码.但如果加载位置改变,程序就必须重新编译.很少用.

\* 加载时(Load Time):如果编译时不知道加载位置,编译器生成可重定位代码(Relocatable Code).加载器(Loader)在将程序加载到内存时,根据实际加载的起始地址,一次性地将所有逻辑地址转换为物理地址.这称为静态地址重定位.

\* 运行时(Run Time):地址转换延迟到程序运行时才进行.CPU 每次访问内存(取指或访存)时,都会将逻辑地址转换为物理地址.这需要硬件支持(如 MMU),称为动态地址重定位.现代操作系统普遍采用此方式,因为它提供了最大的灵活性(如进程可以在内存中移动).

#### 静态地址重定位(Static Relocation)

\* 过程:在程序被加载进内存时,由加载器一次性完成逻辑地址到物理地址的转换.

\* 实现:通常由软件(加载器)完成.

\* 缺点:程序加载后不能在内存中移动；不灵活.

#### 动态地址重定位(Dynamic Relocation)

\* 过程:在进程执行过程中,每次访问内存地址时进行转换.

\* 实现:需要硬件支持,通常是 内存管理单元(MMU - Memory Management Unit).

\* MMU:一个硬件设备,负责将 CPU 发出的逻辑地址实时转换为物理地址.

\* 实现方式(简单示例):使用基址寄存器(Base Register)和重定位寄存器(Relocation Register).逻辑地址加上重定位寄存器的值得到物理地址.

\* 优点:进程可以在内存中移动(例如,为了内存紧凑)；支持更高级的内存管理技术(如虚拟内存).

### 1.5 存储保护(Memory Protection)

\* 目的: \* 确保每个进程有自己独立的地址空间,防止一个进程访问或修改另一个进程的数据.

\* 防止进程访问其不应访问的内存区域(如操作系统的核心代码).

\* 防止进程执行不适当的操作(如写入只读代码段).

\* 实现(基于基址/界限寄存器): \* 基地址寄存器(Base Register):存放进程在物理内存中的起始地址.

\* 界限寄存器(Limit Register):存放进程的逻辑地址空间的大小(或最大合法逻辑地址).

\* 检查过程:CPU 产生的每个逻辑地址 addr 必须满足 0 <= addr < Limit Register.转换后的物理地址 Base Register + addr 也必须在分配给该进程的物理内存范围内(有时 limit check 可以直接在物理地址上做,Base <= Physical Address < Base + Limit).

\* 加载:基址和界限寄存器的值由操作系统通过特权指令(Privileged Instructions)加载,用户程序无法修改.

\* 其他机制:页表/段表中的保护位(读/写/执行权限).

### 1.6 存储共享(Memory Sharing)

允许多个进程安全地共享同一段物理内存区域.例如,共享库(如 C 库)的代码段通常可以在多个进程间共享,只需在物理内存中保留一份副本,节省内存.实现通常依赖于分页或分段机制.

### 1.7 局部性原理(Principle of Locality)

程序在执行过程中的一个普遍倾向:\* 时间局部性(Temporal Locality):如果一个内存位置被访问,那么它在不久的将来很可能再次被访问(如循环中的指令、变量).

\* 空间局部性(Spatial Locality):如果一个内存位置被访问,那么它附近的内存位置也很可能在不久的将来被访问(如顺序执行的代码、数组元素).

意义:存储体系(特别是 Cache)和虚拟内存管理(如页面置换算法)都依赖局部性原理来提高性能.

## 2. 内存管理的目标与功能

\* 目标:有效、安全地管理计算机主存资源,提升系统性能和易用性.主要追求: \* 透明性(Transparency):内存管理对用户程序应该是透明的,程序员不需要关心物理内存的细节.

\* 效率(Efficiency):最小化内存访问时间,高效利用内存资源(减少浪费),降低管理开销.

\* 保护(Protection):确保进程间地址空间隔离,保护操作系统自身.

\* 基本功能: \* 内存分配与回收:为进程按需分配内存空间,并在进程结束或不再需要时回收.管理空闲内存.

\* 地址映射/转换:实现逻辑地址到物理地址的转换.

\* 内存保护:提供机制防止非法内存访问.

\* 内存共享:支持多个进程共享部分内存区域.

\* 内存扩充:通过覆盖、交换、虚拟内存等技术,在有限的物理内存上运行更大的程序或更多的进程.

多道程序设计对内存管理提出的挑战:\* 多个进程同时在内存中,需要管理它们各自的空间.

\* 需要支持地址重定位(程序加载位置不确定).

\* 需要支持地址保护(进程间隔离).

## 3. 进程地址空间

### 3.1 地址空间布局(典型 Linux 布局)

\* 加载来源:代码段、数据段、BSS 段通常从可执行文件中加载.栈和堆是在运行时动态创建和增长的.共享库在运行时动态链接和映射.

\* PC(Program Counter):指向当前执行的指令(在代码段内).

\* SP(Stack Pointer):指向栈顶.

### 3.2 XV6 示例(地址空间)

XV6 是一个教学用的简单操作系统,其地址空间布局相对简单:\* 地址从低到高依次为: \* 代码(text):程序指令.

\* 数据(data)& BSS:初始化和未初始化的全局/静态变量.

\* 栈(stack):函数调用栈.

\* 堆(heap):动态分配区域.

\* 高地址处有 Trampoline 和 Trapframe 区域,用于用户态和内核态之间的切换,并且在两种模式下都映射.(kernel/memlayout.h 中定义).

### 3.3 相关概念解释(常用于虚拟内存)

\* 活跃页面(Active Page):指当前正在被进程频繁访问的页面.

\* 工作集(Working Set):一个进程在最近一段时间 Δ 内所访问到的页面集合.这是衡量进程当前运行所需内存大小的一个动态指标,反映了程序的局部性.如果一个进程的工作集能完全驻留在内存中,那么它就能高效运行,很少发生缺页中断.

\* 常驻集(Resident Set):指一个进程当前时刻实际驻留在物理内存中的页面集合.常驻集的大小受操作系统分配策略的影响.理想情况下,常驻集应该包含进程的工作集.

## 4. 物理内存管理(空闲空间管理)

操作系统需要跟踪哪些物理内存是空闲的,哪些已被分配.

### 4.1 管理数据结构

\* 位图(Bitmap / Bit Vector): \* 将物理内存划分为固定大小的分配单元(通常大小等于页框或几倍页框大小).

\* 位图中的每一位对应一个分配单元.

\* 位的值表示单元状态:0 表示空闲,1 表示已分配(或相反).

\* 优点:简单,易于快速找到连续的空闲块.

\* 缺点:位图本身需要占用内存空间(空间开销与内存总量成正比).查找指定大小的空闲块可能需要扫描整个位图.

\* 空闲区表/链表(Free List): \* 维护一个包含所有空闲内存块(区)信息的数据结构.

\* 每个节点/表项记录一个空闲区的起始地址(Start Address)和 长度(Length).

\* 可以组织成表(Array)或链表(Linked List).

\* 链表类型: \* 隐式空闲链表(Implicit Free List):空闲块和已分配块都存储在内存中,通过块头部信息(大小、是否空闲)来遍历.分配和回收时需要查找.

\* 显式空闲链表(Explicit Free List):只将空闲块链接起来.头部包含指向下一个(有时还有上一个)空闲块的指针.查找空闲块更快,但维护链表指针有开销.

\* 分离空闲链表(Segregated Free List):维护多个空闲链表,每个链表负责特定大小范围的空闲块.分配时,根据请求大小直接去对应链表查找.可以加快分配速度,减少碎片.伙伴系统和 SLAB 分配器属于此类.

\* 优点:空间开销只与空闲块数量有关,不一定与内存总量成正比(对于大块空闲区更有效).

\* 缺点:分配和回收时可能需要遍历链表/表,查找合适空闲块.可能会产生外部碎片.

### 4.2 衡量指标

\* 内存资源利用率:目标是减少浪费.

\* 内碎片(Internal Fragmentation):分配给进程的内存块大于进程实际请求的大小,块内部未被使用的部分.常见于固定分区、页式管理和固定大小分配策略(如伙伴系统分配的块可能大于请求).

\* 外碎片(External Fragmentation):内存中存在足够多的空闲空间总和来满足一个请求,但这些空间不连续,而是散布在已分配块之间的小块.导致无法分配较大的连续内存块.常见于可变分区、段式管理.

\* 性能:分配和回收内存操作的速度,以及这些操作占用的 CPU 时间.

### 4.3 内存分配算法(针对空闲区表/链表)

当需要为进程分配长度为 s 的内存时,如何在空闲区列表中选择一个合适的空闲块?

\* 首次适配(First Fit):从链表/表的开头开始查找,选择第一个找到的大小 >= s 的空闲块.

\* 优点:算法简单,速度较快.倾向于在低地址区域留下碎片.

\* 缺点:可能产生较多的小碎片,且每次查找都从头开始.

\* 下次适配(Next Fit):从上次分配操作结束的位置开始查找,选择第一个找到的大小 >= s 的空闲块.

\* 优点:避免了每次都从头查找,空闲块的使用更均匀分布.

\* 缺点:可能导致大的空闲块很快被分割完.

\* 最佳适配(Best Fit):遍历整个空闲链表/表,找到大小 >= s 且最小的那个空闲块(即差值 size - s 最小).

\* 优点:试图留下最大的可用空闲块,减少大碎片的产生.

\* 缺点:速度最慢(需要全表扫描),容易产生大量非常小而难以利用的碎片.

\* 最差适配(Worst Fit):遍历整个空闲链表/表,找到大小 >= s 且最大的那个空闲块.

\* 优点:试图将剩余部分保持为较大的可用块,避免产生过多小碎片.

\* 缺点:速度慢(需要全表扫描),可能导致大的空闲块很快被消耗掉,无法满足后续对大内存的需求.

分配过程:选定一个空闲块后,如果其大小 Size 远大于请求大小 s,通常会将其分割为两部分:一部分(大小为 s)分配给进程,另一部分(大小为 Size - s)变回一个新的、更小的空闲块.

示例:假设空闲区为 [15K,23K],[48K,20K],[80K,30K].进程 P5 请求 5K,进程 P6 请求 13K.

\* First Fit:P5 分配到 [15K,5K],剩余 [20K,18K].P6 分配到 [20K,13K],剩余 [33K,5K].空闲区变为 [33K,5K],[48K,20K],[80K,30K].

\* Best Fit:P5 分配到 [48K,5K],剩余 [53K,15K].P6 分配到 [53K,13K],剩余 [66K,2K].空闲区变为 [15K,23K],[66K,2K],[80K,30K].

\* Worst Fit:P5 分配到 [80K,5K],剩余 [85K,25K].P6 分配到 [85K,13K],剩余 [98K,12K].空闲区变为 [15K,23K],[48K,20K],[98K,12K].

### 4.4 内存回收(针对空闲区表/链表)

当一个进程释放内存块时,需要将其归还给空闲列表.

\* 合并(Coalescing):为了减少碎片,回收时需要检查该块是否与物理上相邻的空闲块接壤.

\* 上相邻:与前面的空闲块合并.

\* 下相邻:与后面的空闲块合并.

\* 上下都相邻:与前后两个空闲块合并成一个大空闲块.

\* 上下都不相邻:直接将回收块作为一个新的独立空闲块添加到链表/表中.

\* 更新数据结构:相应地修改空闲区表/链表.

### 4.5 特定分配策略

#### 伙伴系统(Buddy System)

\* 思想:一种特殊的“分离适配”算法,用于管理大小为 2 的幂次的内存块.Linux 内核底层内存管理曾采用(现在也部分采用).

\* 结构:将整个可用内存(假设大小为 2<sup>U</sup>)看作一个块.维护 U+1 个空闲链表,分别管理大小为 2<sup>0</sup>,2<sup>1</sup>,...,2<sup>U</sup> 的空闲块.

\* 分配过程(请求大小为 s): 1. 计算满足需求的最小 2 的幂次 k,使得 2<sup>k-1</sup> < s ≤ 2<sup>k</sup>.

2. 查找大小为 2<sup>k</sup> 的空闲链表.

3. 如果找到,分配该块.

4. 如果没找到,则查找更大的块(2<sup>k+1</sup>).找到后,将其分裂(Split)成两个大小相等的伙伴(Buddies)(均为 2<sup>k</sup>).一个用于分配,另一个放入 2<sup>k</sup> 的空闲链表.

5. 如果 2<sup>k+1</sup> 也没有,继续向上查找并递归分裂,直到找到可分配的块.

\* 回收过程(释放大小为 2<sup>k</sup> 的块 B): 1. 查找块 B 的伙伴 B'(地址可以通过异或运算计算得到).

2. 检查伙伴 B' 是否也空闲且大小相同(2<sup>k</sup>).

3. 如果是,则将 B 和 B' 合并(Merge)成一个更大的块(2<sup>k+1</sup>),并递归尝试与新块的伙伴合并.

4. 如果否,则将 B 加入 2<sup>k</sup> 的空闲链表.

\* 优点:分裂和合并相对高效(伙伴地址计算快),能较好地控制外部碎片(合并机制).

\* 缺点:存在内碎片(分配的块大小必须是 2 的幂,可能大于实际需求).

伙伴系统示例(1MB 内存):1. 初始:Free list for 1M:[Block 1M]

2. A 申请 100K:需要 128K(2<sup>7</sup>).分裂 1M -> 512K + 512K; 分裂 512K -> 256K + 256K; 分裂 256K -> 128K + 128K.分配一个 128K 给 A.

\* Memory:[A=128K][Free 128K][Free 256K][Free 512K]

\* Free lists:128K:[1 block],256K:[1 block],512K:[1 block]

3. B 申请 240K:需要 256K(2<sup>8</sup>).分配链表中的 256K 给 B.

\* Memory:[A=128K][Free 128K][B=256K][Free 512K]

\* Free lists:128K:[1 block],512K:[1 block]

4. C 申请 64K:需要 64K(2<sup>6</sup>).分裂 128K -> 64K + 64K.分配一个 64K 给 C.

\* Memory:[A=128K][C=64K][Free 64K][B=256K][Free 512K]

\* Free lists:64K:[1 block],512K:[1 block]

5. D 申请 256K:需要 256K(2<sup>8</sup>).分裂 512K -> 256K + 256K.分配一个 256K 给 D.

\* Memory:[A=128K][C=64K][Free 64K][B=256K][D=256K][Free 256K]

\* Free lists:64K:[1 block],256K:[1 block]

6. 释放 B(256K):它的伙伴 D=256K 未释放.将 B 加入 256K 链表.

\* Memory:[A=128K][C=64K][Free 64K][Free 256K][D=256K][Free 256K]

\* Free lists:64K:[1 block],256K:[2 blocks]

7. 释放 A(128K):它的伙伴 C=64K+Free 64K 中的 Free 64K 是否是伙伴?需要看具体地址.假设 A 的伙伴是 [C=64K][Free 64K] 中的 Free 64K 之前的那个 128K 块,则 A 不能立即合并.将 A 加入 128K 链表.(\*原始图示似乎假设 A 的伙伴是 Free 128K,那释放 A 时应该合并成 256K\*.我们按原始图示逻辑继续)

\* 按图示:释放 A(128K)-> 与其伙伴(Free 128K)合并 -> 256K.

\* Memory:[Free 256K][C=64K][Free 64K][B=256K][D=256K][Free 256K](B 已经释放)-> [Free 256K][C=64K][Free 64K][Free 256K][D=256K][Free 256K]

\* Free lists:64K:[1 block],256K:[3 blocks]

8. E 申请 75K:需要 128K.从 Free 256K 中分裂 -> 128K+128K.分配一个 128K 给 E.

\* Memory:[E=128K][Free 128K][C=64K][Free 64K][Free 256K][D=256K][Free 256K]

\* Free lists:64K:[1 block],128K:[1 block],256K:[2 blocks]

9. 释放 C(64K):伙伴是 Free 64K.合并 -> 128K.这个 128K 的伙伴是 E=128K(未释放).将这个新 128K 加入链表.

\* Memory:[E=128K][Free 128K][Free 128K][Free 256K][D=256K][Free 256K]

\* Free lists:128K:[2 blocks],256K:[2 blocks]

10. 释放 E(128K):伙伴是 Free 128K(来自 C 的合并).合并 -> 256K.这个 256K 的伙伴是另一个 Free 128K(原始 A 分裂剩下的)?地址决定.按图示,它与另一个 128K 合并成 256K,再与相邻的 Free 256K 合并成 512K.

\* Memory:[Free 512K][Free 256K][D=256K][Free 256K]

\* Free lists:256K:[2 blocks],512K:[1 block]

11. 释放 D(256K):伙伴是 Free 256K.合并 -> 512K.这个 512K 的伙伴是 Free 512K.合并 -> 1M.

\* Memory:[Free 1M]

\* Free lists:1M:[1 block]

#### SLAB/SLUB/SLOB 分配器

\* 目的:高效管理内核中频繁分配和释放的小内存对象(如 inode、task\_struct 等).伙伴系统分配的最小块可能仍太大,导致内碎片.

\* 基本思想: \* 将伙伴系统分配的大块内存(称为 "slab")进一步细分成多个固定大小的小对象(object).

\* 为每种类型的对象维护一个或多个 slab 缓存(cache).

\* 分配对象时,从对应的 cache 中快速获取一个空闲对象.

\* 释放对象时,将其放回原 cache,通常无需立即归还给伙伴系统.

\* 优点: \* 减少内碎片(对象大小精确匹配).

\* 分配和释放速度快(对象通常已初始化,且无需查找).

\* 利用缓存局部性(对象在 L1/L2 cache 中可能仍然有效).

\* SLAB 分配器(原始,Jeff Bonwick,Solaris -> Linux): \* 为每种对象类型维护一个 kmem\_cache.

\* 每个 cache 包含多个 slab(通常是 1 或多个物理页).

\* Slab 内包含对象和元数据.

\* Slab 分为:全满(full)、部分空闲(partial)、全空(empty)三种链表.分配优先从 partial slab 获取.

\* 问题:实现相对复杂,元数据管理开销较大,多核环境下锁竞争可能成为瓶颈.

\* SLUB 分配器(改进,Pekka Enberg,Linux 2.6.22+): \* 目标:简化设计,提高性能和可伸缩性.是当前 Linux 默认的分配器.

\* 简化:去除了复杂的 slab 链表管理,主要将 page(物理页)作为 slab 进行管理.元数据存储开销更小.

\* 性能:减少锁竞争,对 NUMA(Non-Uniform Memory Access)架构和多核系统优化更好(利用 per-CPU 缓存).

\* SLOB 分配器(简单): \* 目标:极简、紧凑,适用于代码大小和内存开销受限的嵌入式系统.

\* 实现:使用简单的首次适配算法在小的内存块(slab)内分配.不适合高性能、大内存系统.

\* 查看 Slab 信息(Linux): cat /proc/slabinfo

# 或者使用 slabtop 工具

slabtop

(slabinfo 提供详细的 cache 列表,包括对象大小、活动对象数、总对象数、slab 数等信息.)

## 5. 基本内存管理方案

不同的策略将进程的逻辑地址空间映射到物理内存.

| 方案 | 加载单位 | 内存划分 | 碎片类型 | 地址转换 | 特点 |

| - | - | - | - | - | - |

| 单一连续区 | 进程 | 不划分(除 OS 外)| 无(低利用率)| 简单基址(或固定地址)| 最简单,同一时间只有一个用户进程,内存利用率低 |

| 固定分区 | 进程 | 预先固定大小的分区 | 内碎片 | 基址+界限寄存器(每个分区)| 简单,允许多道程序,但分区大小固定不灵活 |

| 可变分区 | 进程 | 动态按需划分 | 外碎片 | 基址+界限寄存器(每个进程)| 按需分配,灵活,但产生外碎片,需要压缩技术 |

| 页式(Paging)| 页 | 固定大小的页框 | 内碎片(最后一页)| 页表(MMU硬件查表)| 消除外碎片,分配管理简单,不要求连续,常用 |

| 段式(Segmentation)| 段 | 动态按需划分的段 | 外碎片 | 段表(MMU硬件查表)| 符合程序逻辑,易于共享和保护,但产生外碎片 |

| 段页式 | 页 | 固定大小的页框 | 内碎片(段内最后一页)| 段表 + 页表(MMU硬件查表)| 结合二者优点,管理复杂,开销大 |

### 5.1 单一连续区(Single Contiguous Allocation)

\* 特点:内存除操作系统区域外,全部由当前运行的一个用户程序独占.

\* 实现:程序总是加载到同一个内存地址(或通过一个简单的基址寄存器).

\* 优点:非常简单.

\* 缺点:内存利用率极低,无法支持多道程序设计.适用于非常早期的或简单的嵌入式系统.

### 5.2 固定分区(Fixed Partitioning)

\* 特点:内存被预先划分成若干个大小固定的分区.分区大小可以相同也可以不同.

\* 分配:每个分区装入一个进程.当进程需要内存时,操作系统寻找一个足够大且空闲的分区分配给它.

\* 优点:实现简单,支持了多道程序.

\* 缺点: \* 内碎片:分配给进程的分区可能大于进程实际需要的大小.

\* 不灵活:分区大小固定,大进程可能无处容身,小进程占用大分区造成浪费.

### 5.3 可变分区(Variable Partitioning / Dynamic Partitioning)

\* 特点:内存不预先划分.根据进程的实际需求,从空闲内存(洞 Hole)中动态地分割出一个分区分配给它.

\* 分配:使用 First Fit,Best Fit,Worst Fit 等算法在空闲区列表中查找并分配.

\* 优点:按需分配,没有内碎片,比固定分区灵活.

\* 缺点: \* 外碎片:随着进程的分配和回收,内存中会产生许多不连续的小空闲区,即使总空闲量足够,也可能无法满足新的较大内存请求.

\* 管理复杂:需要维护空闲区列表,分配和回收时涉及查找、分割和合并.

\* 外碎片解决方案: \* 紧缩技术(Compaction / Memory Compaction): \* 思想:通过移动内存中的进程,将所有小的空闲区合并成一个或几个大的连续空闲区.

\* 实现:需要动态重定位支持(因为进程物理地址改变了).

\* 问题: \* 开销大:移动内存内容非常耗时,期间系统性能会下降.

\* 移动时机:何时进行紧缩?(例如,当分配失败且有足够总空闲空间时,或定时进行).

### 5.4 页式管理(Paging)

\* 核心思想: \* 逻辑地址空间:划分为固定大小的块,称为 页(Page).

\* 物理内存空间:划分为与页大小相同的块,称为 页框(Page Frame)或物理页面、内存块.

\* 分配:以页为单位进行.进程需要的页可以加载到任意空闲的页框中.逻辑上相邻的页在物理上不必相邻.

\* 逻辑地址结构:逻辑地址 = 页号(Page Number)+ 页内偏移(Offset)

\* 例如,32位地址,页面大小 4KB(2<sup>12</sup> B): \* 高 20位(31-12)是页号.

\* 低 12位(11-0)是页内偏移.

\* 数据结构: \* 页表(Page Table):每个进程都有一个页表.

\* 功能:记录逻辑页号到物理页框号的映射关系.

\* 页表项(Page Table Entry - PTE):至少包含 页框号(Frame Number).通常还包含其他控制位: \* 有效位/驻留位(Valid/Present Bit):标记该页是否在物理内存中.

\* 保护位(Protection Bits):控制读/写/执行权限.

\* 修改位(Modified/Dirty Bit):标记该页加载到内存后是否被修改过.

\* 访问位(Accessed/Referenced Bit):标记该页是否被访问过.

\* 存储:页表本身也存储在内存中.操作系统通过页表基址寄存器(Page Table Base Register - PTBR)(如 x86 的 CR3 寄存器)指向当前进程的页表起始地址.

\* 空闲页框列表:操作系统需要维护一个数据结构(如位图或链表)来跟踪哪些物理页框是空闲的.

\* 地址转换过程(硬件 MMU): 1. CPU 发出逻辑地址.

2. MMU 将逻辑地址分解为页号 p 和页内偏移 d.

3. 使用页号 p 作为索引,访问当前进程的页表(基址在 PTBR).

4. 找到对应的页表项(PTE).

5. 检查 PTE 中的有效位和保护位.如果无效或权限不足,则产生缺页异常(Page Fault)或保护异常,陷入操作系统处理.

6. 如果有效且权限允许,从 PTE 中取出页框号 f.

7. 将页框号 f 与页内偏移 d 拼接(或 f \* PageSize + d)得到最终的物理地址.

8. 访问物理内存.

\* 优点: \* 无外碎片:以固定大小的页框为单位分配,总能利用空闲页框.

\* 内存利用率高:物理内存不必连续.

\* 易于实现共享:让多个进程的页表项指向同一个物理页框即可共享页面(如共享库代码).

\* 支持虚拟内存的基础.

\* 缺点: \* 内碎片:进程的最后一页通常不会完全占满,导致该页框内产生少量内碎片.

\* 页表开销:页表本身需要占用内存空间.对于大地址空间和小编页面,页表可能非常大(需要多级页表等技术解决).

\* 地址转换开销:每次访存理论上需要两次内存访问(一次查页表,一次访问数据).实际使用 TLB(Translation Lookaside Buffer),一种页表项的高速缓存,来加速转换.

### 5.5 段式管理(Segmentation)

\* 核心思想: \* 逻辑地址空间:按照程序的逻辑结构划分为多个段(Segment),如代码段、数据段、栈段等.每个段有自己的名字(通常用段号代替)和长度.段的长度可以不同.

\* 物理内存空间:仍然是线性地址空间,但分配时按整个段分配.

\* 分配:以段为单位.每个段需要分配一块连续的物理内存空间,但不同段之间可以不相邻.

\* 逻辑地址结构:逻辑地址 = 段号(Segment Number)+ 段内偏移(Offset within Segment)

\* 数据结构: \* 段表(Segment Table):每个进程一个段表.

\* 功能:记录逻辑段号到物理内存信息的映射.

\* 段表项(Segment Table Entry - STE):通常包含: \* 段基址(Segment Base):该段在物理内存中的起始地址.

\* 段限长(Segment Limit):该段的长度.

\* 保护位(Protection Bits):如读/写/执行权限.

\* 存储:段表本身也存储在内存中.操作系统通过段表基址寄存器(Segment Table Base Register - STBR)指向当前进程的段表.

\* 物理内存管理:类似于可变分区管理,需要维护空闲区列表,使用 First Fit 等算法分配连续空间.

\* 地址转换过程(硬件 MMU): 1. CPU 发出逻辑地址.

2. MMU 将逻辑地址分解为段号 s 和段内偏移 d.

3. 使用段号 s 作为索引,访问当前进程的段表(基址在 STBR).

4. 找到对应的段表项(STE).

5. 检查: \* 段号 s 是否合法(在段表范围内).

\* 段内偏移 d 是否小于段限长 Limit(0 <= d < Limit).如果超出,则产生地址越界异常.

\* 访问权限是否允许.如果不允许,则产生保护异常.

1. 如果检查通过,取出段基址 Base.

2. 计算物理地址:物理地址 = 段基址 Base + 段内偏移 d.

3. 访问物理内存.

\* 优点: \* 符合程序逻辑:分段是用户可见的,便于程序员组织代码和数据.

\* 易于共享和保护:可以方便地对整个逻辑段(如代码段)进行共享或设置保护属性.

\* 缺点: \* 外碎片:段的长度可变,分配和回收类似于可变分区,会产生外碎片.需要紧缩技术.

\* 内存分配复杂:需要找到足够大的连续空闲块.

### 5.6 段页式管理(Segmented Paging)

\* 核心思想:结合段式和页式的优点.

\* 用户视角 / 逻辑地址空间:仍然按段划分(用户可见).

\* 内存管理 / 物理内存:按页框划分和分配(系统底层).

\* 实现:每个逻辑段内部再进一步划分为固定大小的页.

\* 逻辑地址结构:逻辑地址 = 段号 s + 段内页号 p + 页内偏移 d

(或者看作 段号 s + 段内偏移 offset,其中 offset 再被解释为 页号 p + 页内偏移 d)

\* 数据结构: \* 段表(Segment Table):每个进程一个.

\* 段表项:不再直接指向物理基址,而是指向该段对应的页表的基址,并包含页表的长度(或段的页数).

\* 页表(Page Table):每个段拥有一个页表.

\* 页表项:记录段内的逻辑页号到物理页框号的映射.

\* 地址转换过程(硬件 MMU): 1. CPU 发出逻辑地址(s,offset).

2. 用段号 s 查段表,找到对应段的页表基址和段限长.

3. 检查段内偏移 offset 是否小于段限长.如果超出,则地址越界.

4. 将段内偏移 offset 分解为段内页号 p 和页内偏移 d.

5. 使用段内页号 p 作为索引,访问该段的页表(基址来自段表项),找到对应的页表项(PTE).

6. 检查 PTE 的有效位和保护位.

7. 从 PTE 中取出页框号 f.

8. 计算物理地址:物理地址 = 页框号 f \* PageSize + 页内偏移 d.

9. 访问物理内存.

\* 优点: \* 结合了段式的逻辑清晰、易于共享保护和页式的内存利用率高、无外碎片的优点.

\* 缺点: \* 系统开销大:需要维护段表和多个页表,增加了内存占用.

\* 地址转换更复杂:需要多次内存访问(查段表 -> 查页表 -> 访问数据).同样需要 TLB 来加速.

## 6. 内存 "扩充" 技术

在物理内存不足时,让系统能运行更大程序或更多进程的技术.

### 6.1 覆盖技术(Overlaying)

\* 目的:在物理内存小于程序总大小的情况下运行程序.

\* 思想:程序的不同模块(覆盖段)按照它们的调用关系在同一块内存区域中相互替换.只有当前需要的模块和常驻模块保留在内存中.

\* 实现: \* 程序员负责:需要程序员手动划分程序模块,并指定它们之间的覆盖结构(Overlay Structure).

\* 操作系统提供加载覆盖模块的机制.

\* 示例:程序 A 调用 B 或 C；B 调用 D 或 E；C 调用 F.

\* 常驻区:A(8K)

\* 覆盖区 0:B(8K)或 C(10K)-> 需要 10K

\* 覆盖区 1:D(12K)或 E(4K)(当B在内存时)或 F(10K)(当C在内存时)-> 需要 12K

\* 总需内存 = 常驻区 + Max(覆盖区0)+ Max(覆盖区1)= 8K + 10K + 12K = 30K(远小于原始总和 54K).

\* 优点:能够在小内存上运行大程序.

\* 缺点: \* 对用户不透明:增加了程序员的负担,编程复杂.

\* 执行时间增加:需要从外存动态加载覆盖模块,属于“时间换空间”.

\* 应用:主要用于早期内存极其有限的操作系统.

### 6.2 交换技术(Swapping)

\* 目的:提高内存利用率和系统吞吐量,允许运行的进程总大小超过物理内存.

\* 思想:将暂时不运行的进程完整地从内存移动到外存(交换区 Swap Space),称为换出(Swap Out / Roll Out).当需要再次运行时,再从外存将其换回(Swap In / Roll In)到内存中.

\* 实现: \* Swapper(交换程序):操作系统中负责执行交换操作的模块.

\* 交换区(Swap Space):通常是磁盘上一块连续或特殊管理的区域,用于快速读写整个进程映像.

\* 关键问题与讨论: \* 交换内容:进程的哪些部分需要交换?通常是进程的整个用户地址空间(代码、数据、堆、栈等运行时状态).

\* 交换位置:被换出的进程保存在磁盘的交换区.

\* 交换时机: \* 内存空间不足时触发换出.

\* 进程长时间阻塞或优先级低时可能被换出.

\* 与调度器结合,选择合适的进程换入换出.

\* 换出进程选择:考虑进程状态(不应换出等待 I/O 的进程)、优先级、在内存驻留时间等因素.

\* 换入位置:换回内存时不一定回到原来的物理地址.需要动态地址重定位支持.

\* 进程空间增长:如果进程在内存中时其地址空间增长(如堆或栈扩展),分配可能需要更多内存.如果此时内存不足,可能需要换出其他进程.如果进程在交换区时需要增长,则处理更复杂,通常不允许或有预留机制.

\* 优点:提高了内存利用率,支持运行比物理内存更大的进程集合.

\* 缺点: \* 开销大:整个进程映像的磁盘 I/O 非常耗时.

\* 可能导致抖动(Thrashing):如果内存严重不足,系统可能花费大量时间在换入换出进程上,而实际执行用户代码的时间很少.

\* 应用:曾用于分时系统,现代系统中的虚拟内存可以看作是更精细化的交换技术(以页为单位).

# Note 6

## 1. 虚拟内存基础概念

### 1.1 虚拟地址空间(Virtual Address Space)

\* 定义:操作系统为每个进程提供的、看起来连续的、私有的内存空间.它是对物理内存和磁盘空间的抽象.

\* 作用: \* 隔离进程,提供保护.

\* 简化内存管理,允许程序使用比物理内存更大的地址空间.

\* 实现内存共享.

\* 提问:CPU取到的地址是什么地址?物理地址还是虚拟地址?

\* 解答:CPU发出的地址通常是 虚拟地址(Virtual Address).这个虚拟地址随后会被 内存管理单元(MMU)转换为物理地址(Physical Address).

mermaid

graph LR

CPU - Virtual Address -> MMU;

MMU - Physical Address -> PhysicalMemory[物理内存];

MMU - Page Fault -> OS[操作系统];

OS - Data from Disk -> PhysicalMemory;

PhysicalMemory <-> Disk[磁盘];

### 1.2 虚拟内存管理的目标

1. 透明性(Transparency):运行的程序不应感知到虚拟内存机制的存在.程序员可以像操作一个巨大的连续内存一样编程.

2. 效率(Efficiency):地址转换和页面调度应尽可能快,减少性能开销.需要 硬件支持(如MMU,TLB).

3. 保护(Protection):确保进程之间、进程与操作系统之间相互隔离,互不干扰.

### 1.3 存储体系(Memory Hierarchy)

\* 结构:寄存器 -> Cache -> 内存(RAM)-> 磁盘(Disk)

\* 操作系统角色:协调各级存储器的使用.

\* 目标:结合速度快但容量小的存储(如Cache,RAM)和速度慢但容量大的存储(如磁盘),为用户提供一个既“快”又“大”的逻辑内存(虚存).

### 1.4 相关术语辨识

\* 虚拟内存(Virtual Memory): \* 解释:将物理内存与磁盘结合使用,为程序提供一个容量远大于物理内存的逻辑存储空间.

\* 关键:程序引用的地址(虚拟地址)与物理内存地址不同,由系统自动转换.虚存大小受限于计算机寻址能力和可用磁盘空间.

\* 虚拟地址空间(Virtual Address Space): \* 解释:分配给一个进程的逻辑地址范围.

\* 虚拟地址(Virtual Address): \* 解释:虚拟地址空间中的某个地址.进程通过虚拟地址访问数据,仿佛它就在内存中.

\* 虚拟存储技术(Virtual Memory Technology): \* 解释:一种内存管理技术.程序运行时,只将其一部分装入内存,其余部分留在磁盘.当需要访问不在内存中的部分时,操作系统自动将其从磁盘调入内存.

## 2. 虚拟页式存储管理(Paged Virtual Memory)

### 2.1 基本思想

\* 按需加载:装载程序时,只装入部分(甚至零个)页面到物理内存.

\* 动态调页:当进程执行需要访问不在内存中的页面时,产生 页错误(Page Fault),操作系统负责将所需页面从磁盘动态调入内存.

\* 页面换出:当内存不足时,将内存中暂时不用的页面交换(写回)到磁盘,以腾出空间.

\* 实现方式: \* 请求调页(Demand Paging):只有当页面被访问时才调入.(最常用)

\* 预先调页(Prepaging):预测进程可能需要的页面并提前调入.

\* 本质:资源转换技术,用CPU时间和磁盘空间换取(看似无限的)物理内存空间.

### 2.2 核心策略(Coffman & Denning)

1. 取页策略(Fetch Policy):决定何时将页面从磁盘调入内存.

\* 请求调页:发生缺页时才调入.

\* 预调页:预测并提前调入.

2. 放置策略(Placement Policy):决定将调入的页面放置在物理内存的哪个 页框(Page Frame)中.

\* 解释:在分页系统中,任何空闲页框都可以存放任何页面,所以此策略相对简单.

3. 置换策略(Replacement Policy):当内存没有空闲页框时,决定选择哪个页框中的页面换出到磁盘.

### 2.3 设计与实现问题

\* 页表表项(PTE)的设计.

\* 如何处理页表过大的问题(如多级页表).

\* 地址重定位与快表(TLB).

\* 缺页异常(Page Fault)的处理机制.

\* 驻留集(Resident Set)管理.

\* 置换策略(Replacement Algorithms).

\* 清除策略(Cleaning Policy).

\* 加载控制(Load Control).

## 3. 硬件支持与核心机制

### 3.1 页表表项(Page Table Entry - PTE)设计

\* 关键字段: \* 页框号(Page Frame Number - PFN):该虚拟页对应的物理内存块号.

\* 有效位/驻留位(Valid/Present Bit - P):标记该页是否在物理内存中(1=在内存,0=不在内存/在磁盘).

\* 访问位/引用位(Accessed/Referenced Bit - A/R):标记该页近期是否被访问过(硬件在访问时设置,OS定期清零).用于置换算法.

\* 修改位/脏位(Dirty/Modified Bit - D/M):标记该页在内存中是否被修改过(硬件在写入时设置).如果为1,换出时必须写回磁盘.

\* 保护位(Protection Bits - R/W/X):控制对该页的访问权限(读/写/执行).

\* i386 PDE/PTE 示例:(展示了具体位域)

\* P(Present),A(Accessed),D(Dirty),R/W(Read/Write),U/S(User/Supervisor),PWT(Page Write Through),PCD(Page Cache Disable),PS(Page Size - for large pages).

### 3.2 处理页表过大的问题

\* 问题: \* 32位地址空间(4KB页面,4B PTE):页表本身占用内存 2^20 个 PTE \* 4B = 4MB = 2^22 = 1024 个 4KB 页面 的空间.

\* 若用户拥有 2G = 2^31 = 2^19 个 4KB 页面 的物理空间,索引这块内存的有效(有效位 P=1)的页表就占 512 页(2^19 \* 4B / 4KB = 512).

\* 64位地址空间:页表大小会变得极其巨大(理论上 2^52 \* 8B,不可行).

\* 解决方案: 1. 多级页表(Multi-Level Page Tables): \* 思想:将巨大的线性页表变成树形结构.外层页表(页目录)的条目指向内层页表.只有被用到的内层页表才需要分配内存.

\* 二级页表示例:虚拟地址分为 页目录偏移 | 页表偏移 | 页内偏移.CR3寄存器指向页目录基址 -> 查页目录得页表基址 -> 查页表得页框号 -> 拼接页内偏移得物理地址.

\* Core i7 示例(四级页表):48位虚拟地址,分为 9 | 9 | 9 | 9 | 12 位,对应四级页表的索引和页内偏移.

\* 优点:节省空间,只有实际使用的页表部分才需载入内存.虽然理论上总页表项数量不变,但实际上大多数进程只使用地址空间的一小部分.

\* 空间节省示例:\* 在32位系统中(4GB地址空间),使用4KB页面,线性页表需要1M个页表项(32位虚拟地址空间都需要对应的页表项).

\* 假设一个进程只使用了4MB的连续内存(位于0x80000000-0x80400000),在二级页表中: \* 需要1个完整的页目录(1024项,4KB)

\* 只需要1个二级页表(1024项,4KB,对应使用的4MB区域)

\* 其余1023个二级页表(对应未使用的地址空间)根本不需要创建

\* 总计只需8KB内存,而不是线性页表的4MB,节省了约99.8%的空间

\* 缺点:每次地址翻译需要多次内存访问(可通过TLB缓解).

1. 反转页表(Inverted Page Table): \* 思想:不再为每个进程维护一个页表,而是为整个物理内存建立一个全局页表.页表项 i 对应物理页框 i.

\* 内容:每个页表项记录(进程ID,虚拟页号),表示哪个进程的哪个虚拟页映射到了这个物理页框.

\* 地址转换:给定(进程ID,虚拟页号),需要搜索整个反转页表找到匹配项,得到其索引(即物理页框号).

\* 优化:使用哈希表(Hash Table)加速查找.将(进程ID,虚拟页号)哈希到一个索引,指向反转页表中的一个桶(可能需要链表解决冲突).

\* 优点:页表大小与物理内存大小成正比,与虚拟地址空间大小和进程数量无关.

\* 缺点:查找可能较慢(即使有哈希),实现共享比较困难.

\* 应用:PowerPC,UltraSPARC,IA-64 等.

### 3.3 内存管理单元(MMU)

\* 定义:CPU中的硬件单元,负责将虚拟地址转换为物理地址.

\* 过程:接收CPU发出的虚拟地址,查询页表(优先查TLB),生成物理地址或触发缺页异常.

### 3.4 地址转换(Address Translation)

\* 硬件机制: 1. CPU发出虚拟地址.

2. MMU 从虚拟地址中提取 虚拟页号(VPN)和 页内偏移(Offset).

3. MMU 使用 VPN(可能结合多级页表结构)查找页表(先查TLB).

4. 检查 PTE: \* Case 1:PTE 有效(Valid/Present bit = 1)且权限允许: \* 从 PTE 中获取 页框号(PFN).

\* 将 PFN 与页内偏移拼接,形成 物理地址.

\* 访问内存.

\* 硬件根据访问类型(读/写)可能设置 访问位(A)或 修改位(D).

\* Case 2:PTE 无效(Valid/Present bit = 0)或权限不足: \* MMU 产生 页错误(Page Fault)异常,将控制权交给操作系统.

\* 页错误处理(OS):(详见 3.6)

### 3.5 快表(Translation Look-aside Buffer - TLB)

\* 问题:多级页表导致每次地址翻译需要多次内存访问,显著降低性能.

\* 原理:利用 程序访问的局部性原理(Locality of Reference).最近访问过的页面很可能再次被访问.

\* 什么是TLB: \* 一种高速的、容量小的 相联存储器(Associative Memory).

\* 特点:按 内容 并行查找,速度极快.

\* 存储内容:缓存近期使用过的 虚拟页号(VPN)到 页框号(PFN)的映射(即部分活跃的页表项).

\* 工作流程: 1. MMU 收到虚拟地址后,首先并行查找 TLB.

2. TLB Hit(命中):如果在 TLB 中找到匹配的 VPN,直接获取 PFN,快速完成地址转换.跳过页表查找.

3. TLB Miss(未命中): \* MMU 需要访问内存中的页表进行查找.

\* 找到 PFN 后,将(VPN -> PFN)的映射关系装入 TLB(可能需要替换掉 TLB 中的一个旧条目,使用LRU等策略).

\* 完成地址转换.

\* TLB 刷新问题: \* 问题:进程切换时,原进程的 TLB 条目对新进程无效,需要刷新 TLB,导致新进程初期 TLB Miss 增多,性能下降.

\* 解决: \* PCID(Process Context Identifier)/ ASID(Address Space Identifier):给 TLB 条目打上进程标识符.切换进程时,只需加载新进程的 PCID/ASID,TLB 中带有不同 ID 的条目不会被匹配,无需完全刷新.

\* 关键参数:TLB 的大小、位置(通常集成在MMU或CPU核心内)、替换策略.

### 3.6 缺页异常(Page Fault)处理

\* 触发:地址转换过程中,MMU 发现所需页面的 PTE 无效(P=0)或访问权限不足.

\* 本质:一种硬件中断/异常,将控制权交给操作系统内核的 缺页异常处理程序(Page Fault Handler).

\* 处理流程(典型情况 - 页面不在内存): 1. 保存现场:保存用户进程的状态(PC,寄存器等).

2. 确定原因:操作系统分析是真缺页(P=0),还是保护性错误(权限不足).如果是后者,可能终止进程.

3. 定位磁盘地址:如果是真缺页,查找该虚拟页在磁盘(交换空间或文件)上的位置.

4. 查找空闲页框:在物理内存中寻找一个空闲的页框.

5. 处理无空闲页框: \* 若无空闲页框:执行 页面置换算法,选择一个牺牲页框(Victim Frame).

\* 写回脏页:如果牺牲页框中的页面是 "脏" 的(D=1),则需要将其内容 写回磁盘.

1. 调入页面:启动磁盘 I/O 操作,将所需的页面从磁盘读入选定的(空闲或牺牲)页框.

2. 更新页表:页面调入完成后,修改该虚拟页对应的 PTE(Page Table Entry,页表项):设置 P=1,填入 PFN,清除 D 位,可能设置 A 位.

3. 恢复现场:恢复用户进程的状态.

4. 重新执行指令:重新执行导致缺页异常的指令.此时地址转换可以成功.

### 3.7 驻留集管理(Resident Set Management)

\* 驻留集(Resident Set):进程当前在物理内存中的页面集合.

\* 驻留集大小管理:决定给每个进程分配多少页框.

\* 固定分配策略(Fixed Allocation): \* 在进程创建时确定分配的页框数量.

\* 分配依据可以是: \* 进程类型(交互式、批处理、应用类型).

\* 程序员指定的需求.

\* 系统管理员设置的策略.

\* 特点:简单但缺乏灵活性,无法适应进程工作集大小的动态变化.

\* 可变分配策略(Variable Allocation): \* 根据进程的 缺页率 动态评估其局部性表现.

\* 调整机制: \* 缺页率高 → 增加页框数(扩大驻留集).

\* 缺页率低 → 减少页框数(缩小驻留集).

\* 优点:能够适应程序局部性的变化,提高内存利用率.

\* 缺点:实现复杂,需要监控缺页率,调整策略可能引入系统开销.

\* 系统开销考量: \* 驻留集管理本身会消耗CPU时间和内存资源.

\* 过于频繁的调整可能导致系统开销超过收益.

\* 需要在响应性和开销之间找到平衡点.

## 4. 页面置换算法(Page Replacement Algorithms)

### 4.1 置换问题

\* 背景:当发生缺页异常且没有空闲物理页框时,需要选择一个当前在内存中的页面换出,为新页面腾出空间.

\* 目标:选择一个 最近最不可能被访问 的页面进行置换,以最小化未来的缺页次数.

\* 约束:不能置换被 锁定(Locked/Pinned)的页框(如内核代码、I/O 缓冲区等).

\* 页框锁定:通过 PTE 中的锁定位或特殊机制,防止 OS 将某些关键页面换出内存,避免 I/O 操作期间页面被换出导致错误,或保证实时任务的响应时间.

### 4.2 置换范围

\* 局部置换策略(Local Replacement):仅在引发缺页的那个进程自己的 驻留集(Resident Set)中选择牺牲页.

\* 优点:进程间的隔离性好.

\* 缺点:可能无法利用系统中其他进程不活跃的页框.

\* 全局置换策略(Global Replacement):可以在内存中所有未锁定的页框中选择牺牲页,无论它属于哪个进程.

\* 优点:更灵活,可能提高系统整体吞吐率.

\* 缺点:一个行为不良的进程可能挤占其他进程的页框；进程的缺页率受其他进程影响,难以控制.

\* 与分配策略的关系: \* 固定分配通常配合局部置换.

\* 可变分配可以配合局部或全局置换.

### 4.3 典型置换算法

1. 最优置换算法(Optimal - OPT / MIN): \* 思想:置换 未来最长时间内不会被访问 的页面.

\* 实现:无法实现,因为需要预知未来.

\* 作用:作为性能比较的 基准(Benchmark).

2. 先进先出算法(First-In,First-Out - FIFO): \* 思想:置换 在内存中驻留时间最长 的页面.

\* 实现:维护一个页面进入内存的队列,替换队首页面.

\* 优点:实现简单.

\* 缺点:性能较差,可能换出常用页面.存在 Belady 异常.

3. 第二次机会算法(Second Chance - SCR): \* 思想:FIFO 的改进.检查队首页面的 访问位(A/R).

\* 流程: \* 检查队首页面 PTE 的 A 位.

\* 如果 A=0,置换该页.

\* 如果 A=1,给它 "第二次机会":将 A 位清零,并将该页移到队尾,然后检查下一个队首页面.

\* 优点:比 FIFO 好,避免了立即换出刚被访问过的页面.

4. 时钟算法(Clock): \* 思想:Second Chance Replacement(SCR,第二次机会算法)的高效实现,避免了频繁移动页面.

\* 实现:将所有物理页框组织成一个 循环链表(缓冲区),用一个指针指向下一个要检查的候选页框.

\* 流程: \* 发生缺页时,从指针当前位置开始扫描.

\* 检查当前页框 PTE 的 A 位.

\* 如果 A=0,选择该页框进行置换,将新页面放入,指针前移.

\* 如果 A=1,将 A 位清零,指针前移,继续检查下一个页框.

\* 优点:实现相对简单,性能优于 FIFO,接近 LRU.

5. 最近未使用算法(Not Recently Used - NRU): \* 思想:优先淘汰近期 既未被访问(A=0)也未被修改(D=0)的页面.

\* 实现:利用 PTE 中的 访问位(A)和 修改位(D)将页面分为四类: \* 第 0 类:(A=0,D=0)- 未访问,未修改

\* 第 1 类:(A=0,D=1)- 未访问,已修改

\* 第 2 类:(A=1,D=0)- 已访问,未修改

\* 第 3 类:(A=1,D=1)- 已访问,已修改

\* 流程:OS 定期将所有页面的 A 位清零.发生缺页时,随机 从编号最小的非空类别中选择一个页面进行置换.

\* 优点:实现简单,性能尚可.

6. NRU 的时钟实现:(一种变体)

\* 扫描 1:找第一个(A=0,D=0)的页框,找到即置换.此过程不清 A 位.

\* 扫描 2(若扫描 1 失败):重新扫描,找第一个(A=0,D=1)的页框.此过程中,跳过的页框(A=1)的 A 位被清零.找到即置换.

\* 扫描 3(若扫描 2 失败):此时所有页框 A 位都为 0.重复扫描 1(必然能找到 A=0,D=0 或 A=0,D=1),然后如有必要重复扫描 2.

\* 特点:优先换出干净页(D=0),节省写回磁盘的时间.

7. 最近最少使用算法(Least Recently Used - LRU): \* 思想:置换 过去最长时间未被访问 的页面.基于局部性原理,认为最久未用的页面,近期也最不可能被使用.

\* 实现: \* 时间戳法:每个 PTE 记录上次访问时间,置换时间戳最小的.(硬件开销大)

\* 栈/链表法:维护一个按访问时间排序的页面栈/链表,每次访问将页面移到栈顶/链表头,置换栈底/链表尾的页面.(软件开销大)

\* 优点:性能非常好,接近 OPT.

\* 缺点:实现开销大,纯硬件或纯软件实现都困难.

8. 最不经常使用算法(Not Frequently Used - NFU): \* 思想:置换 过去访问次数最少 的页面.LRU 的一种软件近似.

\* 实现:每个 PTE 关联一个软件计数器,初值为 0.每次时钟中断,检查 A 位,若 A=1,则对应计数器加 1,并将 A 位清零.缺页时置换计数值最小的页面.

\* 缺点:不能很好地区分早期频繁访问但近期不用的页面和近期才开始访问的页面.

9. 老化算法(Aging): \* 思想:模拟 LRU.改进 NFU,使计数器能反映访问的时间远近.

\* 实现:每个 PTE 关联一个多位计数器(e.g.,8-bit).每次时钟中断: 1. 将每个计数器 右移 1 位(模拟时间流逝,旧的访问权重降低).

2. 将当前 PTE 的 A 位 加到计数器的 最左边(最高位).

3. 将 A 位清零.

\* 缺页时,置换计数值最小的页面.计数值小的页面表示近期访问较少或很久未访问.

\* 优点:较好地模拟了 LRU,实现开销适中.

\* 与LRU的区别: \* 精确度:LRU 精确记录每次访问的时间顺序,而 Aging 只能近似反映访问频率和时间远近.

\* 实现开销:LRU 需要在每次内存访问时更新数据结构,开销大；Aging 只在时钟中断时更新计数器,开销小.

\* 历史长度:LRU 可以无限追溯历史访问记录；Aging 受计数器位数限制(如8位只能记录最近8个时间窗口的访问情况).

\* 硬件支持:LRU 需要专门硬件支持才能高效实现；Aging 只需要访问位支持,更易于实现.

### 4.4 算法示例与现象

\* FIFO,LRU,OPT 缺页次数计算: \* 例子:页面访问序列 2 3 2 1 5 2 4 5 3 2 5 2,分配 3 个页框.

\* Belady 异常(Belady's Anomaly): \* 现象:对于某些置换算法(如 FIFO),增加分配给进程的物理页框数,缺页次数 反而增加 的反常现象.

\* 例子:序列 1 2 3 4 1 2 5 1 2 3 4 5,FIFO算法.

\* 原因:FIFO 只考虑进来的时间,不考虑进来之后的访问情况.增加页框可能导致一个"坏"的页面(未来会用到)驻留更久,从而在后面挤掉了更有用的页面.

\* LRU 和 OPT 不存在 Belady 异常:因为它们满足 栈属性(Stack Property):即 m 个页框时的内存内容总是 m+1 个页框时内存内容的子集.增加页框只会包含更多有用的页,不会导致缺页增加.

### 4.5 影响缺页次数的因素

\* 页面置换算法:好算法 ≈ 少缺页.

\* 分配给进程的物理页框数:太少会导致频繁缺页,过多则浪费内存.存在一个最佳范围.

\* 页面尺寸问题: \* 确定页面大小对于分页的硬件设计非常重要,而对于操作系统是个可选的参数.

\* 要考虑的因素: \* 内部碎片:页面越大,内部碎片越多；页面越小,内部碎片越少.

\* 页表长度:页面越大,页表越小；页面越小,页表越大.

\* 辅存的物理特性:磁盘传输特性影响最佳页面大小选择.

\* TLB 覆盖范围:影响地址转换性能.

\* 小页面优缺点: \* 优点:减少内部碎片,更适合程序局部性.

\* 缺点:页表变大,TLB 效率可能降低,磁盘 I/O 效率低.

\* 大页面优缺点: \* 优点:页表小,TLB 覆盖范围大,磁盘 I/O 效率高.

\* 缺点:内部碎片增加,可能不适合小局部性.

\* 最优页面大小:理论上可以用公式 P = √(2se)来计算,其中 $s$ 是页表项的大小(表示页表开销),$e$ 是平均程序段大小(表示内部碎片开销).这个公式平衡了页表大小和内部碎片之间的权衡.

\* 实际实现: \* Intel 80x86/Pentium:支持 4KB 或 4MB 页面大小.

\* 现代系统:通常支持多种页面大小(如 4KB,2MB,1GB),为有效使用TLB带来灵活性,但给操作系统带来复杂性.

\* OS 和应用可根据需求灵活选择不同页面大小.

\* 程序的编制方法:访问模式影响局部性.

\* 例子:只分配了一个 4KB页框,访问按行存储的二维数组 A[1024][1024](4KB页面).

\* 按行访问(方法2):for(i)... for(j)... A[i][j].空间局部性好,每次访问都在同一页或下一页,缺页少( 1024 次,每行开始时缺页).

\* 按列访问(方法1):for(j)... for(i)... A[i][j].空间局部性差,每次访问 A[i][j] 和 A[i+1][j] 会跨越多个页面(1024\*4 bytes ≈ 1 page),导致大量缺页(1024 \* 1024 次).

\* 颠簸/抖动(Thrashing): \* 定义:当系统内存严重不足,分配给进程的页框远小于其活跃页面所需时,进程会不断地发生缺页,大部分时间都用于页面换入换出,而不是真正执行计算.导致系统效率急剧下降.

\* 原因:并发度过高,或进程所需工作集大于可用内存.

\* 表现:CPU 利用率很低,但磁盘 I/O 非常繁忙.

## 5. 高级内存管理策略

### 5.1 工作集模型(Working Set Model)

\* 提出者:Denning(1968)

\* 基本思想:基于 程序访问的局部性原理.一个进程在任何时刻都倾向于访问一个相对较小的页面集合,称为 活跃页面(Active Pages).如果能将这些活跃页面都保留在内存中,就能显著减少缺页.

\* 工作集(Working Set)W(t,Δ):在当前时间 t 之前的 时间窗口 Δ 内,进程实际访问过的 虚拟页面 的集合.

\* Δ:工作集窗口大小,是一个关键参数.

\* 工作集大小 |W(t,Δ)| 随时间动态变化.

\* 与驻留集的关系: \* 驻留集(Resident Set):当前时刻,进程 实际 驻留在物理内存中的页面集合.由 OS 分配策略和置换算法决定.

\* 理想状态:进程的驻留集应包含其当前的工作集(Resident Set >= Working Set).

\* 工作集策略应用: \* 置换:换出不在当前工作集中的页面.

\* 加载控制:只有当一个进程的工作集能够被完全调入内存时,才激活该进程运行,否则挂起.防止 Thrashing.

### 5.2 工作集算法(实现工作集置换)

\* 基本思路:识别并换出不在当前工作集(W(t,Δ))中的页面.

\* 一种实现: \* PTE 增强:每个 PTE 增加一个字段,记录该页面的 最后访问时间(Last Use Time).

\* 参数:设置一个时间阈值 T(近似 Δ).

\* 扫描过程(类似时钟): 1. 定期或缺页时扫描页框.

2. 检查 PTE 的 A 位: \* 若 A=1:表示在当前时钟滴答内被访问.记录 当前虚拟时间 到 PTE 的 "最后访问时间" 字段,并将 A 位清零.

\* 若 A=0:表示在当前滴答内未被访问.计算 页面年龄(Age)= 当前虚拟时间 - 最后访问时间.

1. 判断与置换: \* 如果 Age > T(页面“老”,不在工作集内): \* 如果页面是干净的(D=0),则该页面是 最佳 牺牲页,直接置换.

\* 如果页面是脏的(D=1),先记录下来,继续扫描,希望能找到一个干净的老页面.如果找不到干净的老页面,最后回来置换这个脏的老页面(需要写回磁盘).

\* 如果 Age ≤ T(页面“年轻”,在工作集内):保留该页面,继续扫描.

\* 讨论: \* 精确实现工作集算法开销较大(需要记录和比较时间).

\* 缺页率算法(Page Fault Frequency - PFF):一种近似方法.通过监控进程的缺页率来动态调整其驻留集大小.

\* 设置缺页率上限和下限.

\* 缺页率 > 上限:增加进程的页框数.

\* 缺页率 < 下限:减少进程的页框数.

### 5.3 清除策略(Cleaning Policy)

\* 问题:当需要空闲页框时,如果选中的牺牲页是 "脏" 的,需要先写回磁盘,增加了缺页处理时间.

\* 目标:保持一定数量的 干净(Clean)空闲页框可用.

\* 实现:使用 分页守护进程(Paging Daemon)(如 kswapd in Linux).

\* 该进程周期性(或在内存不足时)被唤醒.

\* 检查内存状态,如果空闲页框低于某个阈值.

\* 使用页面置换算法(如 Clock 或 LRU 近似)选择一些页面.

\* 如果选中的页面是脏页,则启动 I/O 将其 提前写回(Write Back)磁盘,并将其标记为干净.

\* 这样,未来需要空闲页框时,可以直接使用这些已变干净的页框,或者快速换出它们.

\* 双指针时钟(Two-Handed Clock): \* 前指针(Cleaning Hand):由分页守护进程控制.扫描页框,遇到脏页就启动写回,然后将其标记为干净；遇到干净页则跳过.前指针不断“清洁”页面.

\* 后指针(Eviction Hand):由缺页处理程序控制.用于实际选择牺牲页.由于前指针的工作,后指针更有可能遇到干净页面,从而加速缺页处理.

### 5.4 页缓冲技术(Page Buffering)

\* 目的:进一步提高性能,减少因页面换出又立即换回造成的开销.

\* 思路: \* 被置换出的页面 不立即 丢弃或覆盖.

\* 维护两个链表: \* 空闲页链表(Free Page List):存放被置换出的 干净 页面.

\* 修改页链表(Modified Page List):存放被置换出的 脏 页面.

\* 这些页面 暂时保留在内存中.

\* 优点: \* 快速回收(Soft Fault):如果进程很快又要访问刚被“置换”到这两个链表中的页面,可以直接将其重新链回进程的驻留集,无需磁盘 I/O.

\* 簇写回(Cluster Write):修改页链表中的脏页可以累积起来,成簇地(in clusters)写回磁盘,而不是一次只写一页,提高了磁盘 I/O 效率.

\* Page Cache(如 OSTEP 23章 提及):现代 OS 中广泛使用的技术,用于缓存文件数据和匿名页(包括上述缓冲的页面).

\* 基本概念:操作系统在物理内存中维护的一个缓存区域,用于存储最近访问的文件数据和元数据.

\* 工作原理: \* 当进程读取文件时,数据首先从磁盘加载到 page cache,然后再传递给进程.

\* 当进程写入文件时,数据先写入 page cache,标记为"脏",稍后由后台进程(如 pdflush/flush/kswapd)异步写回磁盘.

\* 后续对相同文件数据的访问可直接从 page cache 获取,避免磁盘 I/O.

\* 管理策略: \* 使用类似 LRU 的替换算法(如 Linux 的 2Q)决定哪些页面保留在缓存中.

\* 通过 readahead 机制预读文件数据,提高顺序访问性能.

\* 支持 write-back(延迟写)和 write-through(直接写)两种写入策略.

\* 优势: \* 减少磁盘 I/O:大幅降低文件操作的延迟,提高系统整体性能.

\* 统一缓存:在现代系统中,page cache 通常与 buffer cache 统一,形成统一缓存管理.

\* 内存利用:未使用的物理内存自动用于缓存,提高内存利用率.

\* 2Q 算法:一种近似 LRU 的页面缓存替换算法,但开销更低且能应对 LRU 不擅长的场景.

\* 基本结构:维护两个队列: \* 非活跃队列(A1):存放首次访问的页面.

\* 活跃队列(Am):存放多次访问的页面.

\* 工作流程: \* 当页面第一次被访问时,放入非活跃队列 A1 的头部.

\* 当页面在 A1 中再次被访问时,将其从 A1 移除并放入活跃队列 Am 的头部.

\* 当页面在 Am 中被访问时,将其移到 Am 的头部(类似 LRU).

\* 需要置换页面时,总是从非活跃队列 A1 的尾部选择牺牲页.

\* 定期将活跃队列 Am 尾部的页面移回非活跃队列 A1,以保持整个缓存中约 2/3 的页面在活跃队列中.

\* 优势:能有效应对扫描型工作负载(如顺序读取大文件)导致的频繁页面互换问题,这类场景下传统 LRU 表现不佳.

### 5.5 加载控制(Load Control)

\* 问题:系统中并发运行的进程过多,总内存需求超过物理内存容量,导致 Thrashing.

\* 目标:控制系统的 并发度(Multiprogramming Level),即同时驻留在内存中(活跃)的进程数量.

\* 解决方案: \* 进程挂起/交换(Process Suspension/Swapping):当系统负载过高(如通过高缺页率或低 CPU 利用率检测到 Thrashing)时,选择一个或多个进程,将其 所有页面 换出到磁盘(交换区),并将其置于挂起状态.

\* 选择标准:选择哪些进程挂起?通常选择低优先级进程、长时间阻塞的进程,或者导致最多缺页的进程.

\* 效果:释放大量内存,降低活跃进程的内存竞争,使剩余进程能够获得足够的工作集空间,恢复系统效率.

## 6. 内存映射文件(Memory-Mapped Files)

### 6.1 基本思想

\* 允许进程将一个 文件 或文件的一部分直接 映射 到其 虚拟地址空间 的一个区域.

\* 映射后,进程可以像访问普通内存(如数组)一样 通过内存读写指令 来访问文件内容,而无需使用 read()/ write()等系统调用.

### 6.2 工作机制

\* 系统调用:如 POSIX 的 mmap().

\* 映射建立:mmap()调用并不立即读取文件内容.它只是在进程的虚拟地址空间中建立一个区域(vm\_area\_struct in Linux),并设置相应的页表项指向 文件 作为后备存储(Backing Store).PTE 初始标记为无效.

\* 按需调页:当进程 首次访问 映射区域中的某个地址时,会触发 缺页异常.

\* 缺页处理:OS 识别出这是一个映射文件的缺页,计算出该虚拟地址对应文件中的偏移量,然后从 磁盘文件 读取相应的 数据块(页)到一个物理页框,并更新 PTE 使其有效.

\* 写回: \* 如果映射是 共享的(MAP\_SHARED),对内存区域的修改 最终会写回 磁盘上的原始文件(通常在页面换出时、或调用 msync()、或解除映射 munmap()时).其他映射同一文件的进程也能看到修改.

\* 如果映射是 私有的(MAP\_PRIVATE),使用 写时复制(Copy-on-Write).首次写入时,会为该进程创建一个私有的页面副本,后续修改只影响此副本,不影响原始文件或其他进程.

一些问题:\* 内核空间共享:在现代操作系统中,内核空间通常在所有进程间共享

\* 实现方式:每个进程的页表中,映射到内核空间的部分是相同的,指向同一组物理页框

\* 优势:\* 减少内存占用:避免为每个进程复制一份内核代码和数据

\* 提高效率:进程切换时无需切换内核部分的地址映射

\* 简化内核访问:系统调用时可以直接访问内核数据结构

\* Linux实现:通过将所有进程的高地址部分映射到相同的内核物理页面实现

\* 减轻页表增长压力的方式: \* 稀疏地址空间处理: \* 多级页表(Multi-level Page Tables):将页表分为多级,只为实际使用的地址区域分配页表项,避免为整个虚拟地址空间分配连续页表

\* 倒排页表(Inverted Page Tables):以物理页框为索引建立表项,每个表项记录映射到该物理页的虚拟页信息,页表大小与物理内存成正比而非虚拟地址空间

\* 大页模式(Huge Pages/Large Pages): \* 解决的问题: \* 减少TLB缺失:使用大页可以增加TLB覆盖范围,单个TLB表项可以映射更大内存区域(如2MB或1GB而非4KB)

\* 减少页表层级:减少地址转换时的页表遍历层数,降低内存访问延迟

\* 减少页表大小:相同大小的内存区域需要更少的页表项,节省页表空间

\* 提高内存密集型应用性能:数据库、科学计算等应用可显著受益

\* 代价: \* 内部碎片增加:如果应用只使用大页的一小部分,会造成内存浪费

\* 内存分配挑战:需要连续的物理内存块,在系统运行一段时间后可能难以满足

\* 页面换出复杂化:换出一个大页需要更多I/O操作,可能增加延迟

\* 细粒度保护受限:无法为大页内的不同区域设置不同的访问权限

\* 内存管理复杂性增加:系统需要同时管理标准页和大页,增加内存管理复杂度

### 6.3 mmap()函数(POSIX 示例)

void \*mmap(void \*start,size\_t length,int prot,int flags,int fd,off\_t offset);

\* start:建议的映射起始虚拟地址(通常设为 NULL,由内核选择).

\* length:映射的字节数.

\* prot:内存保护标志(指定映射区域的访问权限).

\* PROT\_READ:可读.

\* PROT\_WRITE:可写.

\* PROT\_EXEC:可执行.

\* PROT\_NONE:不可访问.

\* flags:映射类型和选项.

\* MAP\_SHARED:共享映射,修改会写回文件.

\* MAP\_PRIVATE:私有写时复制映射.

\* MAP\_ANONYMOUS(或 MAP\_ANON):匿名映射,不关联任何文件,用于分配内存(类似 malloc).常与 MAP\_PRIVATE 结合用于进程的堆、栈、BSS段.

\* fd:要映射的文件描述符.对于匿名映射,此参数忽略(通常设为 -1).

\* offset:文件内的映射起始偏移量(必须是页面大小的倍数).

### 6.4 mmap 与 shm(共享内存)对比

\* mmap(基于文件): \* 通信方式:通过映射 同一个磁盘文件 到不同进程的地址空间实现共享.

\* 持久性:共享内容与磁盘文件关联,可以是持久的.

\* 使用场景:共享大文件,IPC,加载动态库,程序加载器.

\* shm(System V Shared Memory): \* 通信方式:使用 shmget()创建一个内核管理的 纯内存 共享区域,然后用 shmat()将其附加到进程地址空间.

\* 持久性:通常是临时的,与进程生命周期或显式删除(shmctl with IPC\_RMID)相关,不直接关联磁盘文件.

\* 性能:可能比基于文件的 mmap 更快,因为它不涉及文件系统开销(除非发生交换).

\* 大小限制:受可用物理内存/交换空间限制.

### 6.5 mmap 相关思考

\* mmap 比物理内存+Swap空间大,是否有问题?

\* 解答:mmap 本身 可以 映射比物理内存+Swap 大得多的文件.mmap 只是建立了虚拟地址到文件内容的 潜在 映射.只有当进程 实际访问 映射区域的页面时,才需要将其调入物理内存.这里需要区分两种情况: \* 文件映射:对于映射到文件的页面,文件本身就是这些页面的"后备存储".当内存不足时,如果这些页面没有被修改过(非脏页),可以直接丢弃,需要时再从文件重新读取；如果被修改过(脏页),则需要先写回文件再释放.

\* 匿名映射:没有关联文件的映射区域(如堆),必须使用Swap空间作为后备存储.Swap是专门用于存储从内存中换出的页面的磁盘区域.

因此,文件映射可以超过物理内存+Swap的总大小,因为它使用原始文件作为后备存储.但如果工作集(实际访问的页面总数)超过了物理内存+Swap,系统会发生颠簸(Thrashing),性能急剧下降,尽管映射本身是合法的.

\* 使用 mmap 代替 read/write 进行文件读写的优势?

\* 解答: 1. 减少数据拷贝:read/write 通常涉及数据在内核缓冲区和用户缓冲区之间的拷贝.mmap 允许进程直接访问内核的页缓存(或直接从磁盘调页),避免了这次拷贝,提高效率,尤其对于大文件或频繁读写.

2. 简化随机访问:对于需要频繁在文件中随机定位读写的场景,mmap 将文件视为内存数组,可以通过指针运算直接访问任意位置,代码更简洁,无需管理文件指针和复杂的 lseek 调用.

3. 内核优化:内核可以更有效地管理 mmap 区域的页面缓存和预读.

### 6.6 内存映射文件应用示例

\* 程序加载:加载可执行文件和动态链接库(DLLs/SOs)时,代码段和只读数据段通常通过私有映射(MAP\_PRIVATE)加载,数据段通过写时复制加载.

\* 进程间通信(IPC):通过共享映射(MAP\_SHARED)同一个文件(或匿名映射),实现高效的数据共享.

\* 数据库:像 LMDB(Lightning Memory-Mapped Database)这样的内存映射数据库,将整个数据库文件映射到内存,利用 OS 的虚拟内存管理进行数据缓存和访问,简化了缓冲管理,提高了 I/O 性能.

## 7. 虚拟内存管理全貌(结合进程结构)

\* 进程控制块(PCB / task\_struct in Linux):包含指向内存描述符(mm\_struct)的指针.

\* 内存描述符(mm\_struct):描述进程的整个虚拟地址空间,包含指向页表的指针(如 CR3 指向的页目录物理地址)和指向虚拟内存区域链表/树(vm\_area\_struct list/tree)的指针.

\* 虚拟内存区域(vm\_area\_struct - VMA):描述进程地址空间中一段 连续 的、具有 相同属性(如权限、映射文件)的虚拟内存区域.例如,代码段、数据段、堆、栈、每个内存映射文件、每个共享库都对应一个或多个 VMA.

\* 关键字段:起始地址(vm\_start),结束地址(vm\_end),访问权限(vm\_prot),标志(vm\_flags,如 VM\_READ,VM\_WRITE,VM\_EXEC,VM\_SHARED),指向映射文件信息的指针(vm\_file)等.

\* 页表(Page Tables):将 VMA 内的虚拟页号映射到物理页框号或标记为不在内存.

\* 物理内存(Page Frames):实际存储数据的内存块.

\* 后备存储(Backing Store):磁盘上的文件(可执行文件、库、数据文件)或交换空间(Swap Area),用于存放不在物理内存中的页面.

交互过程:访问虚拟地址 -> 查找 VMA -> 查找页表(TLB first)-> 访问物理内存 / Page Fault ->(缺页处理)-> 访问后备存储.

## 8. 写时复制(Copy-on-Write - COW)

\* 目的:优化资源(特别是内存)的复制过程,推迟实际的物理复制,直到真正需要时才进行.

\* 应用场景: \* fork()系统调用创建子进程.

\* 私有内存映射(MAP\_PRIVATE).

\* 机制: 1. 共享初始副本:当创建副本时(如 fork()创建子进程),并不立即复制父进程的物理内存页面.而是让子进程的页表项指向与父进程 相同 的物理页框.

2. 标记为只读:同时,将这些共享页框在 父子进程的页表项中都标记为只读(Read-Only),即使它们原本是可写的.

3. 写操作触发异常:如果任何一个进程(父或子)尝试 写入 这些共享的页面,会触发一个 保护性页错误(Protection Fault).

4. 真正复制:操作系统捕获此异常,识别出是 COW 机制.此时,内核会: \* 分配一个新的物理页框.

\* 将原始页框的内容 复制 到新页框.

\* 修改 触发写入操作的那个进程 的页表项,使其指向 新复制的页框,并将该页表项的权限 恢复为可写(Read-Write).

1. 恢复执行:进程继续执行写操作,现在写入的是它自己的私有副本.

\* 优点: \* fork()效率高:如果子进程立刻调用 exec()加载新程序,那么之前的大部分复制就白费了.COW 避免了这种不必要的开销.

\* 节省内存:只要页面不被修改,父子进程可以一直共享同一物理副本.

# 2018秋-期中

## 一、 关于CPU状态 (8分)

1. 为什么要将指令集合划分为不同的特权级别?

答 : 将指令集划分为不同特权级别是为了保护系统安全和稳定运行。高特权级(内核态)执行关键操作，低特权级(用户态)执行普通应用程序，防止用户程序直接访问硬件资源或执行危险操作，避免系统崩溃和安全漏洞。

2. 下列指令哪些指令只能在内核态（管态）下执行? (请在你要选择的指令上划勾)

算术运算、内存读写、读日期时间、启动I/O、修改程序状态字、访管指令、陷入指令(如sysenter)、允许/禁止中断、取数指令

答 : 只能在内核态下执行的指令：启动I/O、修改程序状态字、允许/禁止中断。

## 二、 关于中断、异常和系统调用 (15分)

1. 在实现系统调用时, 需要硬件体系结构提供哪些支持? 在操作系统中应完成哪些工作? 请以 IA32 体系结构为例, 描述执行系统调用时硬件和软件的工作流程。

(可以参考考图1给出的IA32体系结构中相关的信息。涉及到的寄存器包括: EAX、EBX、ECX、EDX、ESI、EDI、ESP、EBP等。)

中断描述符格式

| 段选择符(高16位) | DPL | P | 门类型 | D00 | 空闲 | 段选择符 | 段内偏移(低16位) |

\* IDTR -> IDT

\* GDTR -> GDT

图 1 IA32 体系结构中相关信息

段描述符格式

| 段基址 | 段界限 | 属性(包括DPL) | 段基址 |

| :----: | :----: | :-----------: | :----: |

答 : 硬件支持：特权级机制、中断门机制、寄存器保存现场。IA32提供特殊指令(如int、sysenter)、IDT和GDT表、特权级切换机制。

操作系统工作：系统调用表维护、参数传递接口、内核态处理逻辑实现。

工作流程：用户程序将系统调用号放入EAX，参数放入其他寄存器(EBX/ECX/EDX等)，执行int 0x80指令；CPU检查特权级，切换到内核态，保存现场；根据IDTR找到IDT，根据中断号找到对应处理程序；系统调用处理程序从EAX获取调用号，从寄存器获取参数；执行内核服务；返回用户态，通过EAX返回结果。

2. 系统调用与函数调用有什么区别? 系统调用与API是什么关系?

答 : 系统调用与函数调用区别：系统调用涉及特权级切换(用户态→内核态)，函数调用在同一特权级；系统调用使用特殊指令(如int 0x80)，函数调用使用call指令；系统调用开销较大，需保存更多现场信息。

系统调用与API关系：API是应用程序编程接口，为应用程序提供标准化功能；系统调用是操作系统内核提供的服务；API通常封装了系统调用，提供更友好的接口，一个API可能涉及多个系统调用。

## 三、 关于进程线程模型 (18分)

1. 进程控制块(PCB) 的作用是什么? 它与进程控制操作(创建、撤销、阻塞、唤醒)的关联是什么? 它与进程切换的关联是什么? 它与进程调度的关联是什么? 它是怎么描述进程地址空间的?

答 : PCB作用：记录进程状态信息，是进程存在的唯一标志，包含进程标识、状态、优先级、程序计数器、寄存器值、内存信息等。

与进程控制操作关联：创建进程时分配PCB并初始化；撤销时回收PCB及资源；阻塞/唤醒时修改PCB状态字段。

与进程切换关联：进程切换时保存当前进程寄存器等上下文信息到PCB，并从下一进程PCB恢复上下文；PCB是上下文切换的核心数据结构。

与进程调度关联：调度器根据PCB中优先级、状态等信息决定调度顺序；PCB中包含调度所需的时间片、优先级等信息。

描述地址空间：PCB记录进程页表基址或段表基址；记录代码段、数据段、堆栈段等内存区域信息；维护虚拟地址到物理地址的映射关系。

2. 操作系统对用户级线程和内核级线程的支持有什么不同点? 各列举一个支持用户级线程和内核级线程的实例操作系统名字。

答 : 用户级线程由用户空间线程库管理，内核不感知；切换开销小但一个线程阻塞会导致整个进程阻塞；无法利用多处理器。

内核级线程由内核直接支持和管理；线程阻塞不影响其他线程；可利用多处理器；切换开销较大。

支持用户级线程的系统：早期的Solaris(Green Threads)

支持内核级线程的系统：Linux(NPTL)

## 四、 关于进程调度 (18分)

1. 应用哪一种进程调度算法会导致优先级反转(置)问题? 请举例描述"优先级反转(置)"的现象。如何解决这一问题?

答 : 优先级调度算法(尤其是抢占式优先级调度)会导致优先级反转问题。

现象：低优先级进程L持有某资源，高优先级进程H需要该资源而被阻塞，但中优先级进程M可以继续执行，间接导致优先级H < M，违反原本优先级设计。

解决方法：优先级继承(低优先级进程临时继承被阻塞的高优先级进程的优先级)；优先级天花板(持有资源时提升到可能访问该资源的最高优先级)；避免共享资源(使用无锁算法或复制资源)。

2. 请叙述多级反馈队列调度算法的设计思想。请说明这一设计是如何处理 I/O 密集型和 CPU 密集型进程的? 这一设计是否会带来"饥饿"问题? 如何解决这一问题?

答 : 多级反馈队列设计思想：根据进程行为动态调整优先级；设置多个就绪队列，不同队列优先级和时间片不同；新进程先进入高优先级队列，用完时间片后降级；优先调度高优先级队列进程。

对I/O密集型和CPU密集型处理：I/O密集型进程在I/O操作前往往用不完时间片即主动让出CPU，下次调度时仍在高优先级队列；CPU密集型进程会用完时间片被降级到低优先级队列，减少对系统的影响。

饥饿问题：高优先级队列任务过多时，低优先级队列任务可能长时间得不到执行。

解决方法：设置老化机制(等待时间超过阈值自动提升优先级)；定期将所有进程重新放入最高优先级队列；保证每个队列都有最小执行时间份额。

## 五、 关于同步互斥机制 (5分)

针对单处理器和多处理器, 请说明自旋锁(忙等锁)的优势和劣势。

答: 单处理器环境：

优势：实现简单；无需上下文切换，对于锁竞争少且临界区短的情况效率高。

劣势：忙等浪费CPU资源；单处理器上一个线程自旋时，持有锁的线程无法执行，可能导致死锁；适用场景有限。

多处理器环境：

优势：避免上下文切换开销；临界区短时效率高；适合多核处理器并行执行。

劣势：多线程争用激烈时浪费处理器资源；需要硬件原子操作支持；可能导致缓存一致性问题。

## 六、 综合题 (17分)

1. 假设一个用户在命令行执行了程序 A, A 在执行过程中需要调用系统调用 read 到磁盘上读数据。又假设调度程序选择的下一个要执行的是进程 Z。请叙述从用户在命令行输入执行程序 A 的命令, 到系统执行完程序 A 的全过程。

答: 用户在命令行输入程序A名称并回车；shell进程执行fork创建子进程；子进程通过exec加载程序A；程序A开始执行，初始化环境；执行read系统调用读取磁盘数据；CPU从用户态切换到内核态；内核验证参数，查找文件；发起磁盘I/O请求；由于I/O操作耗时，内核将进程A状态改为阻塞；调度器选择进程Z运行；磁盘读取完成后产生中断；中断处理程序将数据复制到A的缓冲区；将进程A状态改为就绪；当进程A再次获得CPU时，从系统调用返回点继续执行；进程A完成所有操作后调用exit退出；父进程(shell)接收到子进程退出信号，回收子进程资源。

2. 设计操作系统时要对"时间与空间"、"公平与性能"进行权衡, 请各举一个例子进行说明。

答: 时间与空间权衡：分页系统中页面大小的选择。小页面节省内存空间(减少内部碎片)但增加页表开销和换页频率；大页面减少页表开销但可能增加内存浪费。操作系统设计者需根据系统特性选择合适页面大小。

公平与性能权衡：进程调度算法选择。时间片轮转算法公平性好但上下文切换开销大；优先级调度性能高但可能导致低优先级进程饥饿。现代操作系统常采用多级反馈队列等混合算法在公平性和性能间寻求平衡。

# 2019秋-期中

## 一、 运行环境和运行机制 (30 分)

1. (4 分) CPU 状态是软件实现的还是硬件实现的? 为什么这么实现?

CPU状态是硬件实现的。因为CPU状态需要快速响应硬件事件、保证运行时的高效性，同时需要防止软件篡改特权状态，确保系统安全性。硬件实现使得状态切换可以由物理机制保证，不依赖于软件可靠性。

2. (4 分) 举例说明中断/异常的区别。

中断是来自CPU外部的事件引起的，如时钟中断、I/O中断、键盘中断等，是异步发生的；而异常是CPU执行指令时内部产生的事件，如除零错误、页错误、非法指令等，是同步发生的。中断通常可屏蔽，而异常一般不可屏蔽需立即处理。

3. (10 分) 以时间中断为例, 描述处理中断/异常的过程 (包括硬件实现的部分和操作系统实现的部分)。

时钟中断处理过程：

- 硬件部分：

1.时钟芯片计数器达到预设值，向CPU发送中断信号

2. CPU完成当前指令执行后检测到中断信号

3. CPU保存当前程序状态（PC、寄存器值等）到内核栈

4. CPU切换到内核态，将控制权转移到中断向量表中对应时钟中断的处理程序入口

- 操作系统部分：

1. 保存进一步的上下文信息

2. 执行时钟中断服务例程：更新系统时间，检查进程时间片是否用完

3. 若时间片用完，触发进程调度，选择下一个要运行的进程

4. 恢复新进程的上下文

5. 从中断处理程序返回，CPU根据恢复的状态继续执行

4. (4 分) 系统调用有什么作用? 它和函数调用有什么区别?

系统调用作用：为用户程序提供访问操作系统服务的接口，使应用程序能够请求操作系统执行需要特权级别的操作，如I/O、进程创建等。

与函数调用区别：系统调用涉及从用户态到内核态的特权级切换，而普通函数调用在同一特权级内进行；系统调用通过特殊指令（如INT）实现，而函数调用通过CALL指令实现；系统调用需要参数传递机制（如寄存器或内存）来跨越特权边界，而函数调用直接通过栈传参。

5. (8 分) 操作系统是如何实现系统调用的?---------------

1. 用户程序将系统调用号和参数存入寄存器或栈中

2. 执行特权指令（如x86的INT 0x80或SYSCALL）触发从用户态到内核态的切换

3. CPU保存用户态上下文到内核栈中 (之后可能进一步保存在 PCB 中的数据结构)，切换到内核态

4. 硬件跳转到系统调用处理例程入口

5. 内核根据系统调用号查找系统调用表，找到对应的内核函数

6. 内核验证参数，执行相应的内核服务

7. 执行完毕，将结果存放在约定位置（通常是特定寄存器）

8. 通过特权指令（如IRET或SYSRET）恢复用户态上下文，返回用户程序 (此时可能触发程序调度)

## 二、 进程线程模型 (25 分)

1. (8 分) PCB 的作用是什么? PCB 里保存了哪些内容? PCB 和进程的调度/切换有什么关系? PCB 是如何保存进程地址空间的?

PCB（进程控制块）作用：是操作系统管理进程的核心数据结构，记录进程的所有信息，使操作系统能够对进程进行管理和控制。

PCB内容：

- 进程标识信息（PID、用户ID）

- CPU状态信息（寄存器值、程序计数器、PSW）

- 进程调度信息（进程状态、优先级、调度相关的统计信息）

- 内存管理信息（页表或段表指针，内存限制）

- I/O状态信息（打开文件列表，I/O请求）

- 会计信息（CPU时间、时钟时间）

PCB与进程调度/切换关系：进程调度器通过PCB中的调度信息选择下一个运行的进程；进程切换时，通过PCB保存当前进程状态并恢复新进程的状态，是上下文切换的核心数据结构。

PCB保存进程地址空间：PCB不直接保存进程地址空间内容，而是保存指向内存管理数据结构（如页表、段表）的指针，这些数据结构描述了进程的虚拟地址到物理地址的映射关系。

2. (4 分) 进程是如何创建的?

1. 通过系统调用（如fork、exec）请求创建新进程

2. 操作系统分配新的PID，创建新的PCB结构

3. 分配必要的资源（内存空间、初始化页表）

4. 复制或设置初始化数据（若是fork则复制父进程数据，若是exec则加载新程序）

5. 初始化PCB中的各项信息（状态、优先级、程序计数器等）

6. 将新进程加入就绪队列等待调度

7. 返回到父进程，同时新进程等待被调度执行

3. (9 分) 画出五状态进程模型 (包括状态名称和转移条件)。

五状态进程模型：

- 新建状态：进程被创建但尚未加入就绪队列

- 转移到就绪：系统完成初始化并有足够资源

- 就绪状态：进程已准备好运行，等待CPU分配

- 转移到运行：调度器选择该进程执行

- 运行状态：进程正在CPU上执行

- 转移到就绪：时间片用完或更高优先级进程抢占

- 转移到阻塞：进程等待某事件（如I/O操作）

- 转移到终止：进程执行完毕或出现错误

- 阻塞状态：进程等待某事件发生，暂不能执行

- 转移到就绪：等待的事件发生（如I/O完成）

- 终止状态：进程执行结束，释放资源前的过渡状态

- 完全终止：系统回收所有资源

4. (4 分) 用户线程和核心线程有什么区别?

用户线程与核心线程的区别：

- 实现层次：用户线程由用户空间的线程库实现，核心线程由操作系统内核实现

- 系统感知：操作系统不感知用户线程存在，而直接管理核心线程

- 调度方式：用户线程由线程库在用户空间调度，核心线程由内核调度器调度

- 系统调用：用户线程的切换不需要系统调用，而核心线程切换需要陷入内核

- 阻塞影响：一个用户线程阻塞会导致其所属进程的所有线程都阻塞，而核心线程可以独立阻塞

- 并行能力：用户线程不能利用多处理器并行，核心线程可以在多处理器上并行执行

## 三、 进程线程调度 (20 分)

1. (10 分) (作业题) 对某系统进行监测后发现, 在阻塞 I/O 之前, 平均每个进程运行时间为 T。一次进程切换需要的时间为 S, 这里 S 实际上就是开销。对于采用时间片长度为 Q 的轮转调度, 请给出以下各种情况的 CPU 利用率的计算公式:

(a) Q = ∞

CPU利用率 = T/(T+S)

当时间片无限大时，进程会一直运行到阻塞，因此只有在阻塞时才发生上下文切换。

(b) Q > T

CPU利用率 = T/(T+S)

当时间片大于平均运行时间时，进程在用完时间片前就会因I/O请求而阻塞，结果同(a)。

(c) S < Q < T

CPU利用率 = Q/(Q+S)

当时间片小于平均运行时间但大于切换开销时，进程会因为时间片用完而被切换，平均每Q时间就发生一次切换。

(d) Q = S

CPU利用率 = S/(S+S) = 1/2

当时间片等于切换开销时，系统有一半时间用于有效计算，一半时间用于切换。

(e) Q 趋近于 0

CPU利用率 → 0

当时间片趋近于0时，系统几乎所有时间都花在进程切换上，几乎没有时间执行有效计算。

2. (10 分) 请基于多级反馈调度算法设计一种调度算法, 并说明你设计的算法是如何使 I/O 密集型的进程优先调度的? 是如何避免"饥饿"发生的?

基于多级反馈队列的优化调度算法设计：

1. 队列设置：设置多个优先级队列（Q0、Q1、...、Qn），每个队列的时间片大小从高到低依次增加（如Q0为20ms，Q1为40ms...）

2. 新进程处理：新进程首先进入最高优先级队列Q0

3. 时间片用完处理：若进程在当前队列用完时间片，降低其优先级（进入下一级队列）

4. I/O处理：当进程因I/O阻塞而释放CPU时，回到原优先级队列的头部，而不是降级

5. 队列调度：优先调度高优先级队列中的进程，只有高优先级队列为空时才调度低优先级队列

6. 优先级提升：定期（如每隔一段时间T）将所有进程重新放入最高优先级队列

7. 资源占用惩罚：检测CPU密集型进程，如连续多次用完时间片的进程，可额外降低其优先级

I/O密集型进程优先调度的实现：

I/O密集型进程通常在发出I/O请求前只运行很短时间就阻塞，不会用完时间片，因此当I/O完成后回到原队列头部，获得较高优先级。而CPU密集型进程会频繁用完时间片被降级，从而使I/O密集型进程获得优先调度。

避免饥饿的实现：

通过定期的优先级提升（老化机制），确保长期处于低优先级队列的进程也能定期获得CPU时间，防止饥饿现象。同时，队列的时间片增长策略也确保低优先级队列中的进程获得更长执行时间，提高其完成工作的机会。

# 2021春-期中

## 一、 环境和运行机制、系统调用 (30 分)

1. 请列举五条需要在保护模式下运行的指令。"访管指令"需要在保护模式下执行吗？为什么？ (10 分)

需要在保护模式下运行的指令：LGDT（加载全局描述符表寄存器）、LIDT（加载中断描述符表寄存器）、MOV CR0（修改控制寄存器）、CLI（清中断标志）、LTR（加载任务寄存器）。

访管指令必须在保护模式下执行。因为访管指令涉及特权级转换，需要保护模式提供的特权级机制来确保安全，实模式下无法提供这种保护。

2. 描述从接收中断/异常到中断处理程序运行结束，处理中断/异常的过程，并指出哪些是硬件实现的？哪些是操作系统实现的？中断处理程序执行结束后，需要执行哪一个模块（哪一过程）？操作系统初始化和中断有什么关系？ (10 分)

处理中断/异常的过程：

1. 硬件保存当前上下文（保存标志寄存器、CS、IP等）

2. 硬件根据中断向量查找IDT，获取中断处理程序入口地址

3. 硬件切换到内核态，跳转到中断处理程序

4. 操作系统保存其他现场（寄存器等）

5. 操作系统执行具体中断处理程序

6. 操作系统恢复现场

7. 硬件执行IRET指令，恢复标志寄存器、CS、IP等，返回被中断的程序

硬件实现：1-3和7步骤；操作系统实现：4-6步骤。

中断处理程序结束后执行中断返回例程。

操作系统初始化需要设置中断描述符表，初始化中断控制器，为系统提供中断处理的基础设施。

3. 为了实现系统调用机制，系统要做哪些工作？ (10 分)

实现系统调用机制需要：

1. 定义系统调用接口和编号

2. 实现用户态到内核态的特权级切换（通常通过软中断实现）

3. 提供系统调用表，将调用编号映射到对应的内核函

4. 实现参数传递机制（通过寄存器或栈）

5. 实现返回值传递机制

6. 提供用户库封装系统调用（如libc）

7. 实现权限检查和安全控制

## 二、 进程线程模型 (20 分)

1. 请画出五状态进程模型，包括状态的名称以及转移条件。 (10 分)

五状态进程模型：

- 新建（New）：进程刚被创建

- 就绪（Ready）：进程等待被调度

- 运行（Running）：进程正在执行

- 阻塞（Blocked）：进程等待某事件发生

- 终止（Terminated）：进程执行完毕

状态转移：

- 新建→就绪：进程创建完成，系统接纳

- 就绪→运行：调度器选择执行

- 运行→就绪：时间片用完或高优先级进程抢占

- 运行→阻塞：进程等待资源或I/O操作

- 阻塞→就绪：等待事件发生或资源可用

- 运行→终止：进程执行完毕或被强制终止

2. 线程和进程有什么区别？线程有什么属性？为什么线程要有自己的栈？线程的实现方式有哪几种？典型的操作系统是通过什么方式实现线程的？ (10 分)

区别：

- 进程是资源分配的基本单位，线程是CPU调度的基本单位

- 同一进程内的线程共享地址空间和资源，进程间相互独立

- 线程切换开销小于进程切换

- 线程没有独立的地址空间，进程有

线程属性：线程ID、程序计数器、寄存器集合、栈、状态、优先级

线程需要自己的栈是因为：每个线程执行不同的函数调用序列，需要独立的栈来保存局部变量、返回地址和函数参数，确保函数调用的正确执行。

线程实现方式：用户级线程（在用户空间实现）、内核级线程（在内核中实现）、混合实现（两种方式结合）。

典型操作系统通过内核级线程实现线程机制，如Windows、Linux等。

## 三、 进程调度 (25 分)

1. 现在有三个进程，A、B 和 C。A 和 B 的运行时间是 1000ms（全部在 CPU 上进行计算），C 进程是一个无限循环，在每个循环中，会先使用处理器 1ms，然后再花 10ms 使用 I/O。请写出在以下几种调度算法下进程的调度状态，并对每个调度算法对本任务的调度进行分析和评价。调度算法分别为：先来先服务算法、时间片轮转算法（时间片分别为 100ms 和 1ms）、SRTN（最短剩余时间优先算法）。 (10 分)

先来先服务(FCFS)：

假设顺序为A→B→C

- A运行1000ms完成

- B运行1000ms完成

- C无限循环运行

评价：A、B进程有固定完成时间，但C的响应性很差，要等待2000ms才能首次运行。对I/O密集型进程不友好。

时间片轮转(RR)，时间片=100ms：

- A运行100ms，还剩900ms

- B运行100ms，还剩900ms

- C运行1ms后进入I/O等待

- I/O过程中可调度A、B

- 以此类推

评价：公平性好，A和B需要20轮才能完成，总完成时间约为2000ms。C获得较好响应性。

时间片轮转(RR)，时间片=1ms：

- A运行1ms，还剩999ms

- B运行1ms，还剩999ms

- C运行1ms后进入I/O等待

- 循环往复

评价：频繁切换造成巨大开销，但C进程响应性最佳，可以充分利用I/O和CPU并行。

SRTN：

- C的CPU时间短(每次1ms)，优先运行C

- C执行后I/O时，A和B竞争

- 当A和B剩余时间相等时，随机选择

评价：C获得最好的响应性，但A和B可能出现饥饿，因为C无限循环。

2. 请设计一个基于多级反馈队列的调度算法，描述你的算法并回答以下问题： (15 分)

(1) 你设计的算法是抢占的还是非抢占的？为什么？

(2) 你设计的算法对 I/O 密集型和 CPU 密集型进程哪个更友好？说明你的原因。

(3) 你是如何处理进程等待 I/O 后再次被调度上 CPU 时要继续运行的时间片大小的？

(4) 你设计的算法会导致"饥饿"问题吗？为什么会有这个问题？如果产生了"饥饿"的问题，要用什么方法解决？

多级反馈队列调度算法设计：

- 设置三个队列：Q0、Q1、Q2，优先级依次降低

- Q0时间片为4ms，Q1为8ms，Q2为16ms

- 新进程先进入Q0

- 进程在当前队列未用完时间片时被抢占，重新排队

- 用完时间片的进程进入下一级队列

- 最后一级队列(Q2)采用FCFS策略

- 每隔100ms将所有进程重新放入Q0

(1) 该算法是抢占式的。因为新进入高优先级队列的进程可以抢占低优先级队列中运行的进程，提高系统响应性和交互性能。

(2) 该算法对I/O密集型进程更友好。I/O密集型进程通常在短时间内就会因等待I/O而主动放弃CPU，未用完时间片，当它们的I/O完成后，会重新进入高优先级队列，获得更高的调度优先权；而CPU密集型进程通常会用完时间片，逐渐降到低优先级队列。

(3) 最高优先级进程等待I/O后再次被调度上CPU时，会进入最高优先级队列(Q0)，重新获得短时间片。这样设计是因为刚完成I/O的进程可能立即需要CPU进行处理，赋予较高优先级可提高系统吞吐量。

(4) 该算法可能导致饥饿问题。CPU密集型进程可能被持续推至最低优先级队列，而频繁的I/O操作进程一直占据高优先级队列。解决方法是定期将所有进程提升到最高优先级队列（已在算法中设计为每100ms一次），确保每个进程都有公平的执行机会。

## 附加题：(6 分)

请描述一个现实生活中遇到的 IPC (进程间通信) 问题的场景，要求不与已经见到过的 IPC 问题的场景重复。

现实生活中的IPC场景：智能家居系统控制

在一个智能家居系统中，多个独立的控制程序（相当于进程）需要协同工作：

- 温度监控进程收集传感器数据并发送给中央控制系统

- 照明控制进程管理家中灯光

- 安全监控进程检测异常并发送警报

- 用户界面进程接收用户指令

这些进程需要高效通信以确保系统协调运行。例如，当用户通过手机APP（用户界面进程）调整温度设置时，该指令需要传递给温度控制进程；当安全监控检测到异常，需要通知照明控制打开所有灯光并通知用户界面显示警报。

这种场景可通过消息队列实现进程间通信，各进程订阅相关主题，实现去中心化的事件驱动架构。与传统生产者-消费者或读写者问题不同，这是一个多对多的通信模型，各进程既可能是信息生产者也可能是消费者，通信关系更为复杂。

# 补充

## Lecture0 课程介绍

1.helloworld程序的执行

(1)用户告诉操作系统执行helloworld程序

(2)操作系统找到helloworld程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置，计算出对应的磁盘块地址

(3)操作系统之后创建一个新的进程，并将helloworld可执行文件映射到该进程结构（完成进程虚拟地址空间的初始化），表示由该进程执行helloworld程序

(4)完成进程初始化后，操作系统将该进程加入到某种形式的就绪队列中，运行调度程序，进行进程调度

(5)若调度程序选中hello程序，操作系统为helloworld程序设置CPU上下文环境，并跳转到程序开始处

(6)CPU执行helloworld程序的第一条指令，MMU进行地址转换时发生缺页异常（PTE无效）

(7)操作系统为该进程分配一页物理内存，并将代码从磁盘读入内存，然后继续执行helloworld程序

(8)CPU重新执行引发缺页异常的指令，之后helloworld程序执行puts函数（系统调用），在显示器上写一字符串

(9)操作系统找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以操作系统将要写的字符串送给该进程

(10)控制设备的进程告诉设备的窗口系统它要显示字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区

(11)视频硬件将像素转换成一组显示器可接收的控制/数据信号；显示器解释信号，激发液晶屏，我们在屏幕上看到了“hello world”

2.操作系统的地位

3.操作系统的主要工作

(1)程序的执行

a)启动程序、执行程序以及程序结束的工作

(2)完成个性的工作（为硬件提供抽象）

a)在各种硬件之上为用户提供一个抽象层，使用户不用直接和复杂、不一致的物理机器界面交互

b)增强了硬件的可移植性，将有限的硬件资源抽象为接近无限的虚拟资源（如对CPU和物理内存的抽象）

c)为硬件提供保护

d)实现该工作的过程代码（驱动程序）和硬件因素密切相关，即需要设置与测试、使用物理地址、设备接口寄存器等等

•复杂繁琐、代码量大

•代码不通用

(3)完成共性的工作（为用户提供服务）

a)任何一个程序都需要的、最基本的工作。这些工作具有共性，运行过程相同，且与具体应用无直接关系（与用户所关心的应用目标无直接关系）

b)易于使用，基本服务，统一性

## Lecture1 操作系统概述

1.操作系统的定义、作用、设计目标与特征

(1)定义

操作系统是计算机系统中的一个系统软件，是一些程序模块的集合

a)进程的角度：操作系统由一些可同时独立运行的进程和一个对这些进程进行协调的核心组成

b)资源管理的角度：是资源的管理者，有效、合理的管理资源。能以尽量有效、合理的方式（如资源分配策略与算法）组织和管理计算机的软硬件资源，实现资源在时间和空间上的共享、提高资源利用率

c)用户的角度：是服务的提供者，便于用户使用。合理地组织计算机的工作流程，控制程序的执行并向用户提供各种服务功能，使得用户能够灵活、方便的使用计算机，使整个计算机系统能高效地运行

d)虚机器的角度：可以将操作系统看作一个分层结构，每一层完成其特定构成一个虚机器，对上一层提供支持；通过逐层功能扩充，最终完成整个操作系统虚机器，而操作系统虚机器向用户提供各种功能，完成用户请求

(2)作用

a)虚拟化：操作系统将物理资源(如处理器、内存或磁盘)，转换成更通用、更强大、更易用的虚拟形式

b)提供API：为了满足用户的需求并且利用虚拟机的特性(比如运行程序、分配内存或访问文件)，操作系统还提供了一些可供用户调用的接口(APIs)以及应用程序的标准库

c)资源管理器：虚拟化允许多程序运行(如共享CPU)，允许多程序并发地访问自己的指令和数据(如共享内存)，允许多程序访问设备(如共享磁盘等等)

(3)设计目标

抽象、性能、保护、可靠、安全、移动性

(4)特征

a)并发（concurrency）

处理多个同时进行的活动的能力

b)共享（sharing）

操作系统与多个用户程序共享计算机系统中的资源，在共享的过程中操作系统需要尽量最优化资源的分配与对资源的保护

c)虚拟（Virtual）

将一个物理实体映射为若干个逻辑实体，通过分时或分空间使用逻辑实体实现对物理实体的共享

d)随机性（不确定性）

操作系统必须准备好随时对以不可预测的次序发生的事件进行响应

2.批处理操作系统的工作方式、分类

(1)工作方式

a)用户将作业交给系统操作员

b)系统操作员将许多用户的作业组成一批作业，输入到计算机系统中，在系统中形成一个自动转接的连续作业流

c)系统操作员启动操作系统

d)系统自动、依次执行每个作业

e)由系统操作员将作业结果交给用户

(2)分类

a)单道批处理系统

b)多道批处理系统

3.批处理操作系统中采用的SPOOLing技术

(1)定义

Simultaneous Peripheral Operation On-Line

同时的外围设备联机操作——假脱机技术

(2)基本思想

利用磁盘/磁带作缓冲，将输入、计算、输出分别组织成独立的任务流（进程），分离I/O和计算，使二者真正并行

(3)工作原理

a)作业进入到磁盘上的输入井

b)按某种调度策略选择几种搭配得当的作业，并调入内存

c)作业运行的结果输出到磁盘上的输出井

d)结果从磁盘上的输出井送到打印机

## Lecture2 中断异常机制（运行环境与机制）

1.寄存器

(1)分类

a)用户可见寄存器

数据寄存器、地址寄存器、条件码寄存器

机器语言在任何特权级别下均可以直接饮用

b)控制和状态寄存器

程序计数器、指令寄存器、程序状态字（PSW）寄存器

用于控制处理器，在某种特权级别下可以访问、修改

2.CPU状态（特权级别）

(1)定义

处理器具有不同的运行模式，在不同的运行模式（特权级别）下运行的指令集合不同

(2)实现

在程序状态字寄存器PSW中专门设置一位，根据运行程序对资源和指令的使用权限而设置不同的CPU状态

(3)不同特权级下的指令

a)特权指令

只能由操作系统使用、用户程序不能使用的指令

b)非特权指令

用户程序可以使用的指令

(4)状态的切换

a)用户态 → 内核态/监督模式/管态

只能通过中断/异常/陷入机制

陷入指令（访管指令）：提供给用户程序的接口，用于调用操作系统的功能（服务）；例如int，trap，syscall，sysenter/sysexit，ecall

b)内核态 → 用户态

设置程序状态字PSW

3.中断异常机制

操作系统是由中断/异常等事件驱动的

(1)事件的分类

(2)中断/异常的概念和区别

a)CPU对系统发生的某个事件作出的反应，事件的发生改变了处理器的控制流

b)中断：来自CPU外，为了支持CPU和设备之间的并行操作而引入

例子：I/O与CPU计算的并行

c)异常：表示CPU执行指令时本身出现的问题

例子：算术溢出、除零、取数时的奇偶错误，访存地址时越界或执行

了“陷入指令”等

(3)特点

a)随时发生

b)自动处理

c)被中断程序状态可恢复（在以后的某个时刻继续执行）

(4)作用

a)及时处理设备发来的中断请求

b)可使OS可以捕获用户程序提出的服务请求

c)防止用户程序执行过程中的破坏性活动

(5)工作原理

硬件和软件相互配合

a)硬件：中断/异常响应

捕获中断源发出的中断/异常请求，以一定方式响应，将处理器控制权交给特定的处理程序

b)软件：中断/异常处理

识别（在系统初始化时设置）中断/异常类型并完成相应的处理

(6)工作过程

a)框架

CPU暂停正在执行的程序，保留现场后自动转去执行相应事件的处理程序，处理完成后返回断点（在此之前需要先调度该程序对应的进程），继续执行被打断的程序

b)中断响应流程

i.硬件

•在每一个指令执行周期中，CPU取出指令并执行

•若中断被禁用，则在指令执行结束后直接取下一条指令，开始下一个指令执行周期；

•若允许中断，则在指令执行周期的最后时刻扫描中断寄存器，查看是否有中断（当有多个中断时有中断处理逻辑来决定多个中断怎么处理）。

若有中断，则中断硬件将该中断寄存器的内容按规定编码后（称为中断码）送入PSW的相应位，通过“交换中断向量”引出中断处理程序（见4）

若无中断，则取下一条指令，开始下一个指令执行周期

•中断向量表是一个内存单元，存放各中断处理程序入口地址和程序运行所需的处理机状态字（PSW）。计算机开机时操作系统按硬件要求初始化该表（Q：操作系统初始化与中断/异常有哪些关联？）

硬件执行流程按中断号/异常类型的不同，通过中断向量表转移控制权给中断处理程序

ii.软件

•保存相关寄存器（如通用寄存器，硬件只保存PSW和PC）

•分析中断/异常的具体原因

•执行对应的处理功能

•恢复现场，返回被事件打断的程序

4.中断异常响应流程实例

以设备输入输出中断为例

(1)硬件

a)打印机给CPU发中断信号

b)CPU处理完当前指令后检测到中断，判断出中断来源并向相关设备发确认信号

c)CPU开始为软件处理中断做准备

•处理器状态被切换到内核态

•在系统栈中保存被中断程序的重要上下文环境，主要是程序计数器PC和程序状态字PSW

d)CPU根据中断码查中断向量表，获得与该中断相关的处理程序的入口地址，并将PC设置成该地址，新的指令周期开始时，CPU控制转移到中断处理程序

(2)软件

a)中断处理程序开始工作，在系统栈中保存现场信息

b)检查I/O设备的状态信息，操纵I/O设备或者在设备和内存之间传送数据等

(3)硬件

a)中断处理结束时，CPU检测到中断返回指令，从系统栈中恢复被中断程序的上下文环境 ，CPU状态恢复成原来的状态，PSW和PC恢复成中断前的值，CPU开始一个新的指令周期（也可以调度其他进程，恢复的是被调度进程的上下文环境）

5.IA32体系结构的基于x86处理器的中断异常机制

(1)有关硬件

a)中断控制器（PIC或APIC）

b)实模式：中断向量表（Interrupt Vector）

存放中断服务程序的入口地址

•入口地址＝段地址左移4位＋偏移地址

•不支持CPU运行状态切换

•中断处理与一般的过程调用相似

c)保护模式：中断描述符表（Interrupt Descriptor Table）

采用门(gate)描述符数据结构描述中断向量

(2)中断/异常的硬件处理过程

a)确定与中断或异常关联的向量i

b)通过IDTR寄存器找到IDT表，获得中断描述符（表中的第i项）

c)从GDTR寄存器获得GDT的地址；结合中断描述符中的段选择符，在GDT表获取对应的段描述符；从该段描述符中得到中断或异常处理程序所在的段基址

d)特权级检查

e)检查是否发生了特权级的变化，如果是，则进行堆栈切换(必须使用与新的特权级相关的栈)

f)硬件压栈，保存上下文环境；如果异常产生了硬件出错码，也将它保存在栈中

g)如果是中断，清IF位

h)通过中断描述符中的段内偏移量和段描述符中的基地址，找到中断/异常处理程序的入口地址()入口地址 = 段基址＋偏移，执行其第一条指令

6.系统调用

(1)定义

用户在编程时可以调用的操作系统功能

(2)系统调用的作用（系统调用与C函数调用的区别）

a)系统调用是操作系统提供给用户的唯一接口

b)系统调用是操作系统提供给编程人员的唯一接口

(3)执行过程

a)CPU执行特殊的陷入指令，引发异常，完成用户态到内核态的切换

b)中断/异常机制：硬件保护现场；通过查中断向量表把控制权转给系统调用总入口程序

c)系统调用总入口程序：保存现场；将参数保存在内核栈里；通过查系统调用表把控制权转给相应的系统调用处理例程或内核函数

d)执行对应的系统调用例程

e)恢复现场，返回用户程序

(4)基于x86处理器的Linux系统调用实现

a)CPU执行系统调用指令int $0x80（%eax存放系统调用号，%ebx, %ecx, %edx, %esi, %edi…），产生异常

•改变特权级，切换栈：用户栈→内核栈

从任务状态段TSS中装入新的栈指针（SS︰ESP），该指针指向内核栈

•用户栈的信息（SS︰ESP）、EFLAGS(PSW)、代码段寄存器CS 、EIP 寄存器的内容压栈（返回用）

•将EFLAGS压栈后，复位TF，IF位保持不变

•在IDT中找到该门描述符，从中找出段选择符装入代码段寄存器CS

•代码段描述符中的基地址 + 陷阱门描述符中的偏移量 → 定位system\_call()的入口地址

•执行system\_call中的指令

b)system\_call()将参数保存在内核栈中，检查是否是合法的系统调用号，若是则根据系统调用号检索系统调用表，找到系统调用程序入口，比如sys\_func()

c)sys\_func()执行完后返回system\_call()，将返回值从内核栈转移到%eax中，经过ret\_from\_sys\_call()例程（调度，处理信号）返回用户程序

## Lecture3 进程线程模型

1.进程模型

(1)基本信息

a)特点

•是正在运行程序的抽象，是程序的一次执行过程

•是对CPU的抽象，将一个CPU变换成多个虚拟的CPU，操作系统将CPU调度给进程

•每个进程具有独立的地址空间，系统资源以进程为单位分配

b)定义

•进程是具有独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的独立单位

•如何查看当前系统中有多少个进程？Linux: ps -e

c)分类

•系统进程、用户进程

•前台进程、后台进程

•CPU密集型进程、 I/O密集型进程

d)进程与程序的区别

•程序是静态的，进程是动态的：进程有生命周期的，有诞生有消亡，是短暂的；而程序是相对长久的

一个程序可对应多个进程

进程具有创建其他进程的功能

•进程更能准确刻画并发，而程序不能

(2)进程的状态

a)进程的三种基本状态

•运行态（Running）

占有CPU，并在CPU上运行

•就绪态（Ready）

已经具备运行条件，但由于没有空闲CPU，而暂时不能运行

•等待/阻塞态（Waiting/Blocked）

因等待某一事件（如完成I/O，等待读盘结果DMA）而暂时不能运行，可以细化为阻塞态、封锁态、睡眠态

b)进程状态之间的转换

进程在消亡前处于且仅处于某一状态

不同系统设置的进程状态数目不同

•三状态模型

•进程的其他状态

创建（new）

已完成创建一进程所必要的工作（如PID、PCB）

但尚未同意执行该进程（因为资源有限）

ps. 创建与就绪的关系

终止（terminated）

终止执行后，进程进入该状态

可完成一些数据统计工作与资源回收

挂起（suspend）

把一个进程从内存转到磁盘（进程不占用内存空间，其进程映像交换到磁盘上）——用于调节负载

•五状态模型与七状态模型

(3)进程的数据结构

a)进程控制块PCB

•PCB: Process Control Block

进程描述符、进程属性

操统系统中表示进程的数据结构

记录进程的各种属性，描述进程的动态变化过程

•操作系统通过 PCB 控制和管理进程

进程与PCB一一对应

PCB是系统感知进程存在的唯一标志

进程表：所有进程的PCB的集合

b)PCB主要内容

•进程描述信息

进程标识符（process ID），唯一，通常是整数

进程名，通常基于可执行文件名（不唯一）

用户标识符（user ID）

进程组关系

•进程控制信息

进程当前状态

优先级（priority）

代码执行的入口地址

对应的可执行文件名（磁盘地址）

运行统计信息（e.g.执行时间片页面调度）

进程间同步和通信；进程阻塞的原因

进程的队列指针

进程的消息队列指针

•资源使用情况

虚拟地址空间的现状

打开文件列表（文件描述符…）

•CPU现场信息（上下文）

寄存器值（通用寄存器，程序计数器PC，进程状态字PSW…）

指向该进程段/页表的指针

c)内核对进程的组织与管理

•进程表（队列）

操作系统为每一类进程建立一个或多个队列，队列元素为PCB

伴随进程状态的改变，其PCB从一个队列进入另一个队列

注： 多个等待队列等待的事件不同；就绪队列也可以多个；单CPU情况下，运行队列中只有一个进程

•五状态进程模型的进程队列模型

就绪队列无优先级（e.g. FIFO）

事件发生时，进程从对应队列移动到就绪队列中

(4)进程的地址空间

一致、私有、巨大、稀疏

a)操作系统给每个进程都分配了一个地址空间，地址空间是对物理内存的抽象

b)进程地址空间的布局

•Linux：cat /proc/PID/maps

显示进程映射的内存区域和访问权限

在内核中，进程的段用vm\_area\_struct表示，所有地址空间存储在task(PCB)->mm->mmap链表中

一个文件可以映射到进程的一段内存区域中，映射的文件描述符保存在vm\_area\_struct->vm\_file域中，这种内存区域叫做有名内存区域，另一个是匿名映射内存区域

(5)进程的控制

a)简介

•进程控制操作完成进程各状态之间的转换，由具有特定功能的原语完成

•原语（primitive）：完成某种特定功能的一段程序，具有不可分割性或不可中断性，即原语的执行必须是连续的，在执行过程中不允许被中断（atomic）

•进程创建原语；进程撤消原语；阻塞原语；唤醒原语；挂起原语；激活（解挂）原语；改变进程优先级

b)进程的创建

•创建时机

系统初始化

操作系统提供的服务

交互用户登录系统（？）

现有进程派生出一个新进程

提交一个程序执行（e.g. 命令行）

•创建过程

UNIX： fork/exec; WINDOWS： CreateProcess

给新进程分配一个唯一标识(pid)以及进程控制块(PCB)，并为进程分配地址空间

初始化进程控制块，设置默认值 (如：状态为New，......)

设置相应的队列指针，如把新进程加到就绪队列的链表中

创建或扩充其他数据结构

c)进程的终止

UNIX: exit; WINDOWS：ExitProcess

•终止时机

正常退出（自愿的）

出错退出（自愿的）

严重错误（非自愿）

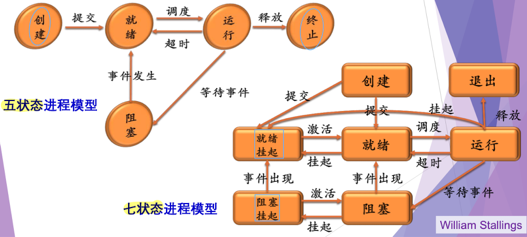
被其他进程杀死（非自愿）

•终止/撤销过程

结束子进程或线程（zombie in Linux）

收回进程所占有的资源（关闭打开的文件，断开网络连接，回收分配的内存…）

撤消该进程的PCB

d)进程阻塞和进程唤醒

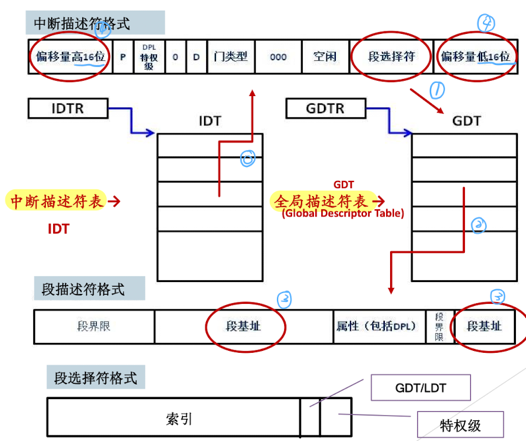
UNIX: wait; WINDOWS: WaitForSingleObject

•时机/过程

处于运行状态的进程，在其运行过程中期待某一事件发生，如等待键盘输入、等待磁盘数据传输完成、等待其它进程发送消息，当被等待的事件未发生时，由进程自己执行阻塞原语，使自己由运行态变为阻塞态

e)其他事件

正常结束； 给定时限到；缺少内存；存储器出界；保护性出错（写只读文件）；算术错；超出时间（进程等待超过对某事件的最大值）； I/O 失败；无效指令（如试图执行数据）；特权指令；操作系统干预（如当死锁发生时）；父进程请求中止某一子进程；父进程中止（子进程也中止）

2.Unix系统的进程控制操作

(1)fork()

通过复制调用进程来建立新的进程，是最基本的进程建立过程

a)fork的实现

•为子进程分配一个空闲的进程描述符（proc结构）,并分配给子进程唯一标识pid

•以一次一页的方式复制父进程地址空间（优化：Linux的解决方案是利用存储管理模块中的“写时复制技术”COW（Copy-On-Write）对fork()进行了优化<虚拟内存机制>）

•从父进程处继承共享资源，如打开的文件和当前工作目录等

•将子进程的状态设为就绪，插入到就绪队列

•对子进程返回标识符 0，对父进程返回子进程的 pid

b)fork与COW

•创建新的进程mm\_struct、vm\_area\_struct、页表的精确副本

•将两个进程中的每个页面均标记为只读，两个进程的每个vm\_area\_struct都标记为“私有的COW”

•于是返回时每个进程都有虚拟内存的精确副本，后续对内存的写使用COW机制创建新页面

(2)exec()

包括一系列系统调用，它们都是通过用一段新的代码覆盖原来的内存空间，实现进程执行代码的转换

(3)wait()

A diagram of a computer system

AI-generated content may be incorrect.提供初级的进程同步措施，能使一个进程等待，直到另外一个进程结束为止

(4)exit()

用来终止一个进程的运行

3.线程模型

(1)定义

进程中的一个运行实体，是CPU的调度单位，也叫轻量级进程

(2)特点

a)有状态及状态转换 → 需要提供一些操作

b)不运行时需要保存上下文（相比进程而言更少e.g.程序计数器等寄存器）

c)有自己的栈和栈指针（与进程栈“不交”）

d)线程间共享所在进程的地址空间和其他资源

e)可以创建、撤消另一个线程（程序开始时是以一个单线程进程方式运行的）

(3)用户级线程（User Level Thread）

a)基本信息

•在用户空间实现：在用户空间建立线程库，提供一组管理线程的函数

•运行时系统完成线程的管理工作（操作、线程表）

•内核管理进程，不知道（用户）线程的存在，线程切换不需要内核态特权

b)实例：POSIX Pthreads

•POSIX(Portable Operating System Interface)1003.4a

•多线程编程接口，以线程库方式提供给用户

c)优点

•线程切换快

•应用程序可以指定调度算法

•可运行在任何操作系统上（只需要实现线程库）

d)缺点

•内核阻塞进程（如进程进行系统调用）会导致进程中所有线程也被阻塞

•内核只将处理器分配给进程，因此同一进程中的两个线程不能同时运行于两个处理器上

•解决方案

修改系统调用为非阻塞（Jacketing/wrapper）

重新实现对应系统调用的I/O库函数（syscall -> 切换线程而非进程）

(4)核心级线程

a)在内核中实现，内核管理所有线程，维护进程和线程的上下文，并向应用程序提供API接口

b)线程的切换需要内核支持，线程是调度的基本单元

c)例子： Windows

(5)混合式线程

a)在内核中实现，支持用户线程。线程创建在用户空间完成，而线程调度等在核心态（Kernel thread）完成

b)例子：Solaris

4.协程（coroutine）

(1)协程的定义

允许在一个进程/线程中实现多个执行流，并在这些执行流之间进行协作

(2)引入协程的原因

改善并发编程的性能，用于处理异步任务和大量并发连接，支持高并发A screenshot of a grid

AI-generated content may be incorrect.