操作系统

Kajih Du

操作系统

1 引言
- 1.1 操作系统的功能
- 1.2 指令执行过程
- 1.3 硬件工作模式
- 1.4 系统调用

2 进程
- 2.1 进程的特征
- 2.2 进程的表示
- 2.3 线程
- 2.4 中断
- 2.5 进程管理的API
  - 2.5.1 fork()
  - 2.5.2 exit()
  - 2.5.3 exec()
  - 2.5.4 wait()
  - 2.5.5 信号
- 2.6 进程间通信（IPC）
  - 2.6.1 共享内存
  - 2.6.2 文件抽象

3 线程
- 3.1 并发和并行
- 3.2 线程的内容
- 3.3 线程库
  - 3.3.1 用户空间实现线程
  - 3.3.2 内核实现线程
- 3.4 多线程模型
  - 3.4.1 多对一模型
  - 3.4.2 一对一模型
  - 3.4.3 多对多模型
- 3.5 Linux线程实现
- 3.6 存在的问题

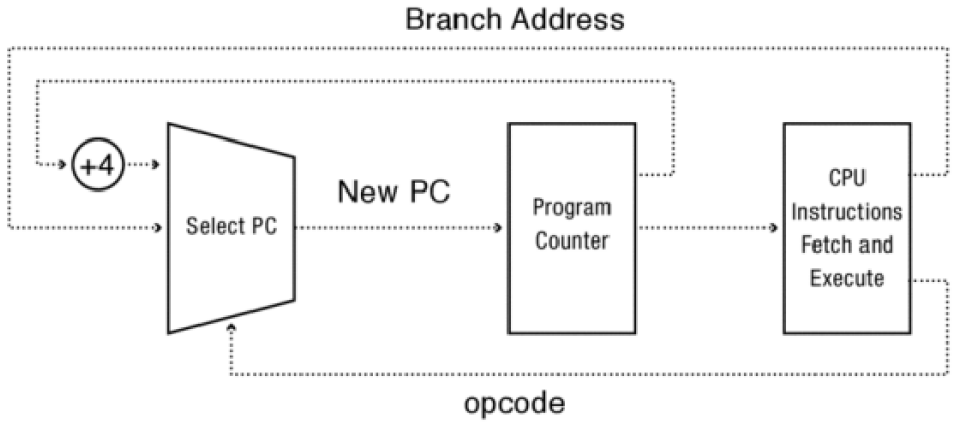
4 进程调度
4.1 进程状态
4.2 基本概念
4.3 调度算法
4.3.1 先来先服务（FCFS）
4.3.2 短作业优先（SJF）

1 引言

1.1 操作系统的功能

- 抽象、虚拟化：处理器-线程；内存-地址空间；磁盘-文件；网络-接口；进程（独立的地址空间，一个或多个线程）
- 资源保护
- 资源共享和分配

1.2 指令执行过程



- 取指：根据PC，从存储器中取指令到IR（指令寄存器）
- 解码：解译IR中的指令
- 执行：控制CPU部件，执行运算，产生结果并写回，同时在标志寄存器里设置运算结果标志；跳转指令根据跳转地址修改PC，其他指令递增PC
- 重复上述过程

1.3 硬件工作模式

硬件至少要支持两种工作模式：内核态和用户态

用户态禁止以下操作：

- 修改页表指针
- 禁止中断

- 直接操作硬件
- 修改内核内存

内核态和用户态切换条件：

- 系统调用：用户态进程主动切换到内核态
- 中断
- 异常

1.4 系统调用

系统调用：用户通过操作系统提供的接口（API）获得内核提供的一些服务

服务种类：进程管理、信号管理、文件管理、目录管理、权限管理、 时间管理

API和系统调用的关系：

- API是一些预定义的函数。跟内核没有必然的联系。
- 不是所有的API函数都对应一个系统调用，有时，一个API函数会需要几个系统调用来共同完成函数的功能也有一些API函数不需要调用系统调用（因此它所完成的不是内核提供的服务）
- 程序员只能使用API与系统交互， 不能直接使用系统调用
- 系统调用不与程序员进行交互，它根据API函数，通过一个软中断机制向内核提交请求，以获取内核服务的接口

2 进程

2.1 进程的特征

进程：一个具有一定独立功能的程序关于某个数据集合的一次活动。

特征：动态性、并发性、独立性、异步性、结构性。

2.2 进程的表示

进程控制块（PCB）：使一个在多道程序环境下不能独立运行的程序(含数据),成为一个**能独立运行**的基本单位、一个**能与其它进程并发执行**的进程

- Process number 进程号

- Process state 进程状态
- Program counter 程序计数器
- CPU registers 寄存器
- CPU scheduling information 调度信息
- Memory-management information 内存管理信息
- Accounting information 记账信息
- I/O status information 输入/输出状态

进程具有：1. “看上去”独立的CPU：记录了上下文，即当前的程序计数器及寄存器；2. “看上去”有全部内存：具有私有的地址空间

栈的作用：

- 存储临时结果
- 允许递归执行

2.3 线程

- 同一个进程内所有线程共享内存的内容（全局变量，堆），I/O状态
- 每一个线程有一个TCB，私有的CPU寄存器和执行栈

2.4 中断

中断处理程序需要栈，中断时使用**内核栈**（不能是用户栈）。

操作系统在创建进程时，同时要创建两个栈（用户栈和内核栈）

2.5 进程管理的API

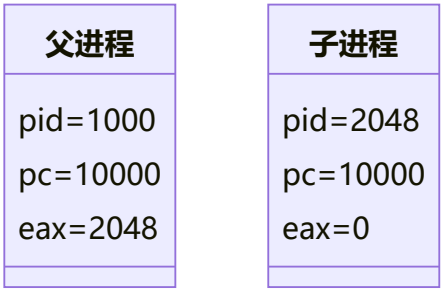
2.5.1 fork()

`pid_t fork()`
`pid_t`：整数，`fork()`的返回值类型。

```
1 int main()
2 {
3     pid_t pid;
```

```
4       int x = 1;
5       pid = fork();
6       if (pid == 0) { /* Child */
7           printf("child: x=%d\n"; ++x);
8           exit(0);
9       }
10      /* Parent */
11      printf("parent: x=%d\n", --x);
12      exit(0);
13  }
14  /*
15  一种可能的输出:
16  child: x=2
17  parent: x=0
18  */
```

复制的过程如下所示



图中父进程和子进程有同样的`pc`，寄存器`eax`存储`fork()`的返回值。对于父进程，其变量`pid`的值为子进程的`pid`2048，运行过程中直接绕过`if`语句输出`parent: x=0`；对于新创建的子进程，其`pc`和父进程的相同，执行`pid=fork();`，但`fork()`的子进程的返回值为`0`，进入`if`子句内，输出`child: x=2`。父进程和子进程**并发执行**，无法预测先后。

子进程几乎**复制了父进程全部的PCB**，继承了父进程的打开文件，除了有不同的`pid`，父进程和子进程在`fork`函数的返回值不同，子进程的返回值为0，父进程的返回值为子进程的`pid`。

子进程创建完未返回`pid`，与父进程共用页表，同一块物理内存；子进程返回`pid`写入时为子进程拷贝一块新的内存空间，再改变页表映射关系（**写时复制**）。

2.5.2 exit()

`return`和`exit`的区别：

- `return`是栈的返回，`exit`是进程的结束
- 从`main`函数退出会隐式调用`exit()`，并将`return`的返回值传给`exit()`
- 从`exit()`退出，先调用退出程序，刷新`stdio`流缓冲区，再执行`_exit()`系统调用
- `return`为关键字，`exit()`为库函数

2.5.3 exec()

作用：

- 用指定程序文件重新初始化一个进程
- 父进程`fork`子进程，子进程调用`exec`启动新的程序
- `exec`**不创建新进程**，只将当前进程重新初始化指令段和用户数据段、堆栈段以及CPU的PC指针
- 调用`exec`后不会返回，不再是父进程的程序

```
pid = fork();
if (pid == 0) // 子进程
    exec(...);
else // 父进程
    wait(&stat); // 监控子进程是否结束
```

执行上述代码后，子进程执行新的程序；父进程执行`wait()`等待子进程的结束。

2.5.4 wait()

作用:

- 调用`wait()`的进程会被挂起（阻塞），直到任意一个子进程结束
- 子进程结束时，**回收子进程所占用资源（PCB等）**，返回被回收的子进程的`pid`
- 没用运行的子进程，直接返回`-1`

注意：`exit()`不会回收PCB。

如果一个子进程执行`exit()`之后，父进程未执行`wait()`，此时子进程的PCB未被回收，成为**僵尸进程**。如果父进程未调用`wait()`就中止，子进程将成为**孤儿进程**。

Linux对孤儿进程的处理：将`init`进程作为孤儿进程的父进程，`init`进程定期执行`wait()`.

2.5.5 信号

`SIGINT`：中断进程（Del或Ctrl-C）
`SIGTERM`：软件中止，来自`kill`指令
`SIGCLD`：进程的一个子进程终止
`SIGFPE`：浮点数溢出

接收信号:

- 内核计算`pnb = pending & ~blocked`
- `if (pnb == 0)` 无信号需要处理
- `else`找到非零位，处理相应信号

处理信号:

- 缺省处理`signal(SIGINT, SIG_DFL);`
- 忽略`signal(SIGINT, SIG_IGN);`
- 捕捉信号并定义handler `signal(SIGINT, sig_handler);`

2.6 进程间通信（IPC）

2.6.1 共享内存

共享存储：允许两个或更多进程共享一个给定的存储区。**不需要复制数据**，最快的IPC。

```
#include <sys/shm.h>
// 创建或获取一个共享内存：成功返回共享内存ID，失败返回-1
int shmget(key_t key, size_t size, int flag);
// 连接共享内存到当前进程的地址空间：成功返回指向共享内存的指针，失败返回-1
void *shmat(int shm_id, const void *addr, int flag);
// 断开与共享内存的连接：成功返回0，失败返回-1
int shmdt(void *addr);
// 控制共享内存的相关信息：成功返回0，失败返回-1
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
```

消息传递与共享内存：

- 共享内存通信速度快（只需要访存）
- 消息传递在传递少量数据有用，因为不需要同步避免冲突
- 消息传递在机械间通信更容易实现
- 消息传递通过系统调用，开销更大

2.6.2 文件抽象

`cwd`：当前工作目录

通过系统调用切换`int chdir(const char *path); //change CWD`

高层次文件操作抽象：streams

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

`FILE*`：文件描述符、缓冲区队列、锁

3 线程

3.1 并发和并行

并发：同时在做，但不是同时执行（交替执行）
并行：多个处理器同时处理不同任务

3.2 线程的内容

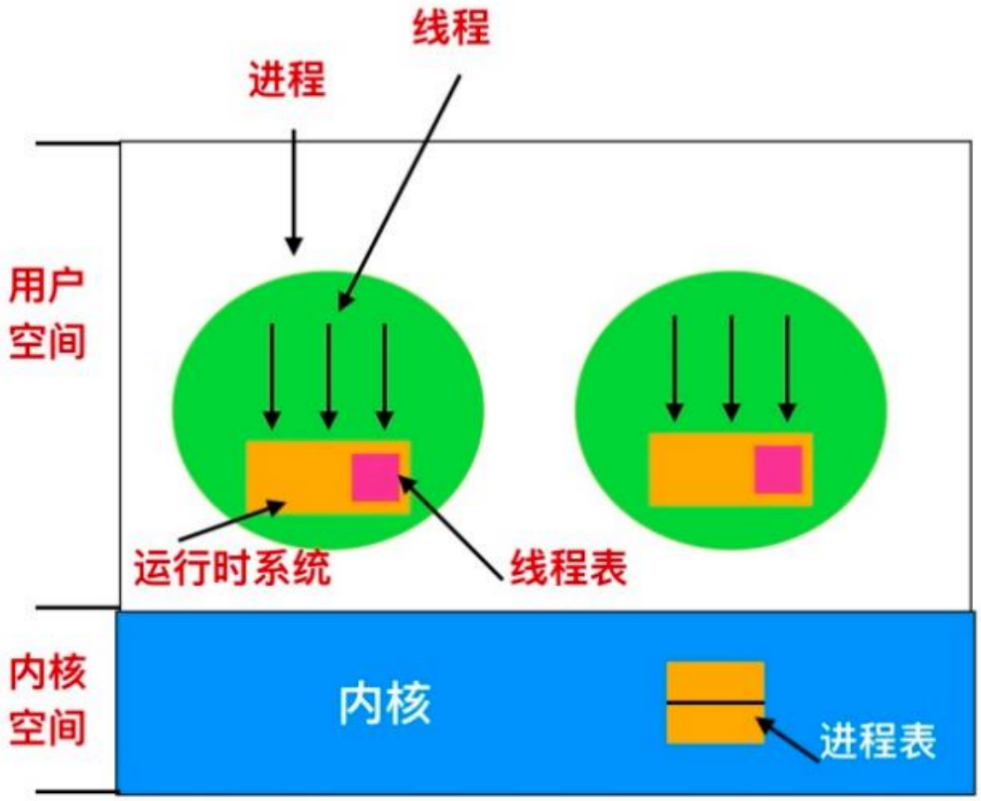
每个线程有独立的程序计数器、堆栈、寄存器、状态，共享代码、数据及打开的文件

线程和进程：

- 进程：1.独立；2.携带相对更多的状态信息；3.有独立的地址空间；4.通过IPC（进程间通信机制）通信；5.上下文切换速度较慢
- 线程：1.作为进程的子集存在；2.共享进程的状态、内存、资源；3.共享进程的地址空间；4.线程之间通信更方便；5.在同一个进程空间内的上下文切换，速度更快
- **一个进程至少是一个线程**

3.3 线程库

3.3.1 用户空间实现线程



不需要操作系统支持，进程维护线程表实现

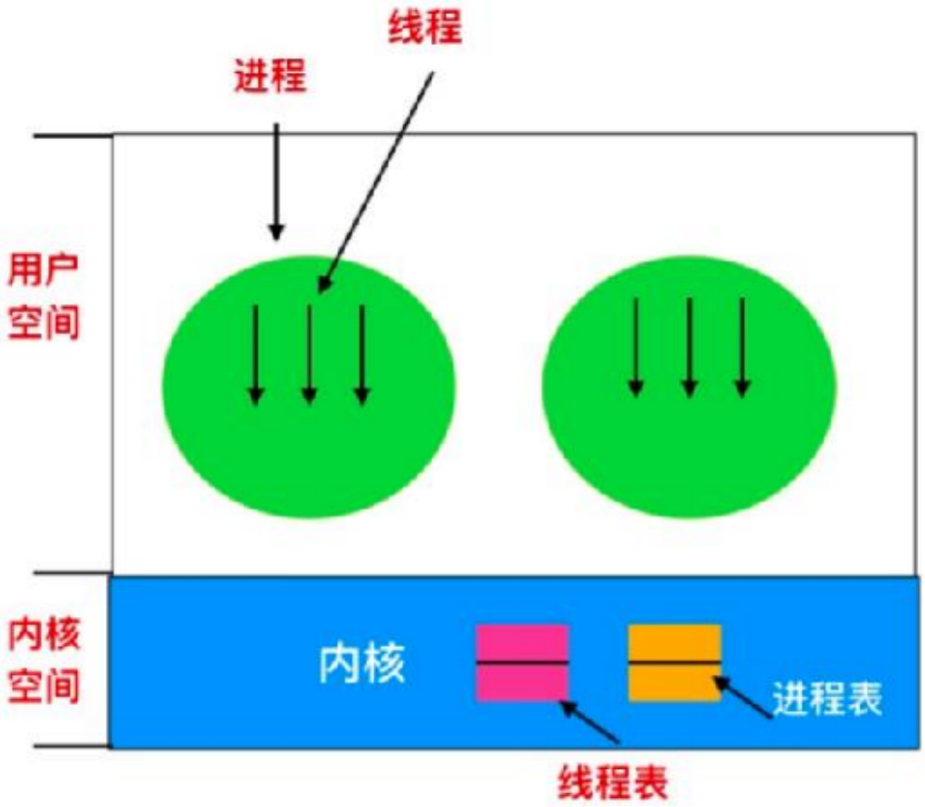
优点：

- 线程调度不需要内核直接参与，**控制简单**
- 可以在不支持线程的操作系统实现
- 创建、销毁线程核线程切换代价等线程管理的**代价比内核线程少**
- 允许每个线程定制自己的调度算法，**管理比较灵活**
- 线程能用的**堆栈空间比内核级线程多**

缺点：

- 同一进程中只能同时有一个线程在运行，如果有一个线程使用了系统调用而阻塞，那么整个进程都会被挂起。
- 资源调度按照进程进行，多个处理机下，同一个进程中的线程只能在同一个处理机下分时复用

3.3.2 内核实现线程

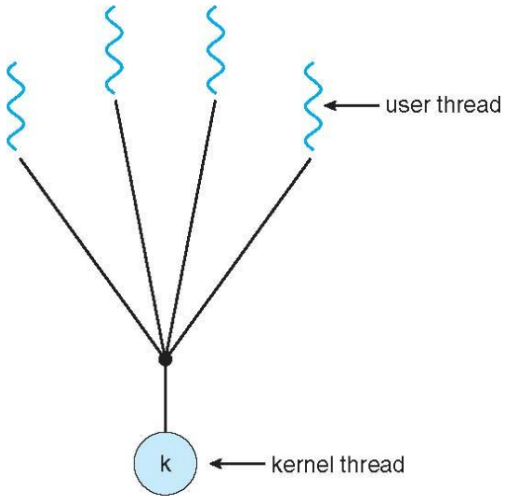


内核支持线程的创建、撤消和调度都需OS内核提供支持

内核支持线程是OS内核可感知的， 用户级线程是OS内核不可感知的

3.4 多线程模型

3.4.1 多对一模型

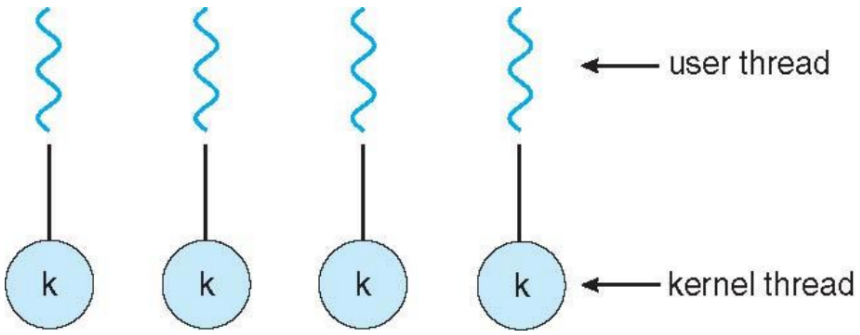


基本调度单位为内核线程，多个用户线程对应一个内核线程。

进程内的线程切换不会导致进程切换，用于不支持内核线程的操作系统

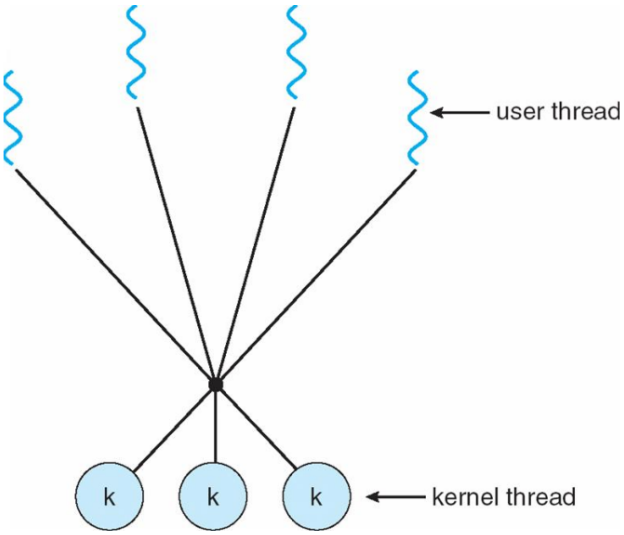
优点：简单
缺点：单个线程阻塞会造成整个进程；多个线程**无法运行在多处理器**上

3.4.2 一对一模型



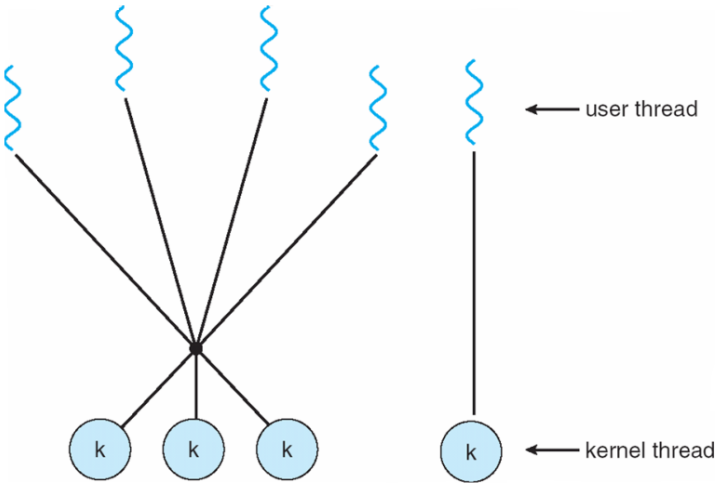
优点：并发性，可充分利用多处理器的优势；一个线程阻塞可以切换到同一个进程的另外一个线程
缺点：线程调度上下文切换的开销较大；操作系统限制内核线程数量，使得用户的线程数量受到限制

3.4.3 多对多模型



多路复用多个用户级线程到同样数量或更少数量的内核线程

优点：不限制应用的线程数；多个线程可以并发，当一个线程被阻塞时，内核可以调用另一个线程来执行



双层模型：多对多模型的变种，允许绑定某个用户线程到一个内核线程上

3.5 Linux线程实现

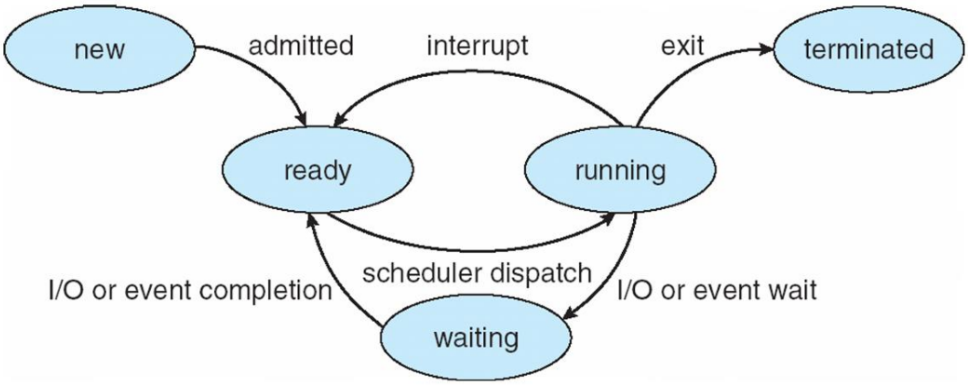
`clone()`系统调用，产生轻量级进程（Lightweight Process, LWP）（创建了一个进程，多了一个PCB），生成的PCB共享页表

3.6 存在的问题

- 线程中的`fork()`
- 线程退出：异步退出（立刻退出）、推迟退出（检测是否能退出后再退出）
- 信号处理：线程接受到信号，发送给指定线程还是进程下的所有线程
- 调度器激活

4 进程调度

4.1 进程状态



- 运行（running）：正在执行指令
- 就绪（ready）：等待被分配到处理器
- 等待/阻塞（waiting）：等待事件完成

4.2 基本概念

CPU区间：大部分时间较短

I/O区间

进程调度时机：

- 主动放弃：进程正常中止；运行过程中发生异常；主动请求阻塞
- 被动放弃：分给进程的时间片用完；更高优先级的进程进入队列；更紧急事件处理

4.3 调度算法

4.3.1 先来先服务（FCFS）

周转时间和响应时间较长，不适用一般的操作系统

4.3.2 短作业优先（SJF）