

# 并发：进程与线程

蒋炎岩

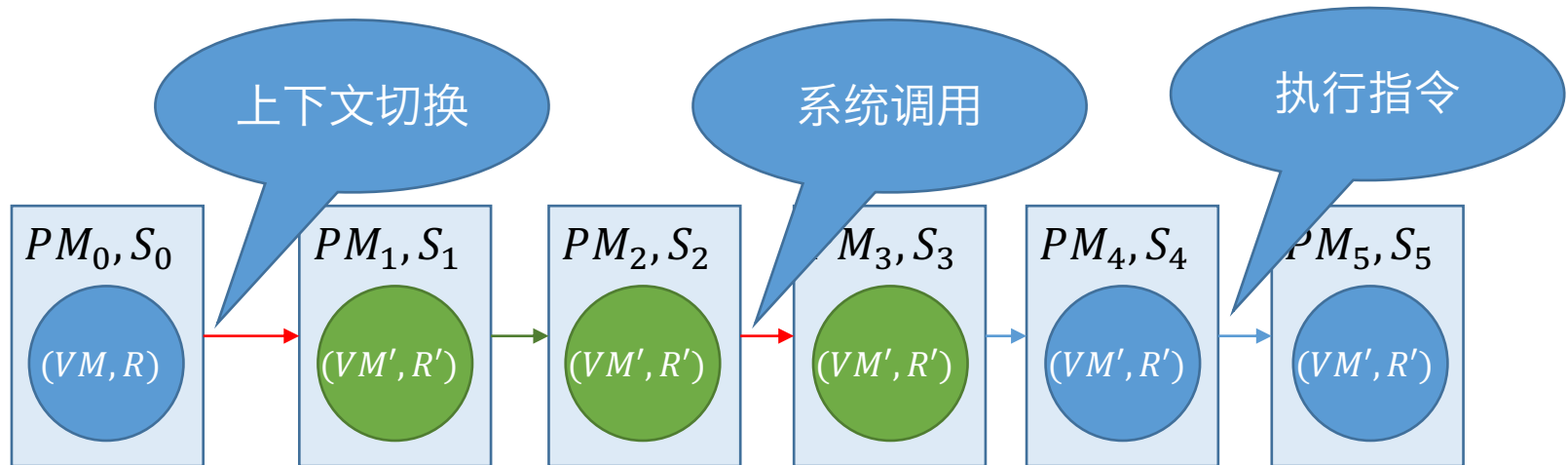
南京大学 | 计算机软件研究所 | 系统与软件分析研究组



# 多进程并发

## 复习：操作系统中的进程

- 每个进程可以看作是一个独立的状态机
- 通过系统调用访问操作系统中的资源



# 多进程并发

- 如果没有系统调用，进程之间互不合作，**完全独立**
  - 从状态机的角度，一个进程完全不能影响另一个进程的行为
- 但进程实际可能**分时共享处理器**或**同时执行**(多CPU)
  - 系统调用的执行也可能分时共享/同时执行
- 进程和我们一样
  - 有自己的想法、自己的行为(程序)
  - 能访问物理世界中公共的资源(系统调用)

# 多进程并发：简单的例子

- 启动两个进程：
  - `./a.out 1 & ; ./a.out 2 &` ← 这是什么？
  - 两个进程随时(也可能同时)请求系统调用

```
int main(int argc, char *argv[]) {  
    for (int i = 0; i < 10000; i++) {  
        printf("Hello from %s\n", argv[1]);  
    }  
}
```

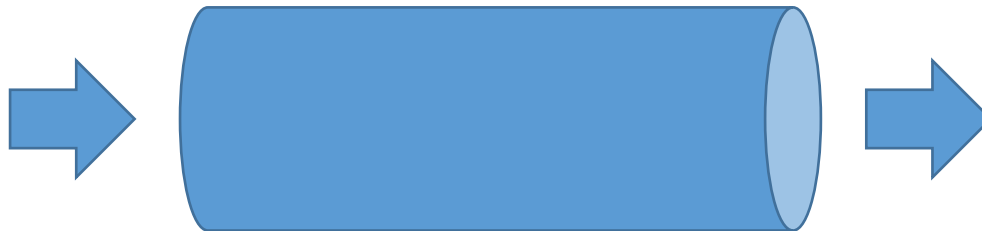
- 运行结果是不确定(non-deterministic)的
  - ... 1 1 1 2 1 2 1 2 2 2 1 1 1 1 2 1 ...
  - 我们是否可能看到Hello from Hello from 1 2?

# 系统调用的规约(Specification)

- 在我们理解系统调用时，都是假设系统中只有一个进程的
  - write(fd, buf, size) – 向文件描述符中写入数据
  - fork – 创建新的进程
  - execve – 替换当前进程
  - .....
- 但实际上，系统调用的规约包含多进程并发的行为
  - 任意多个系统调用可能并发执行
    - 几个进程同时从一个文件中读取/写入数据
    - mmap一个文件的时候有进程删除它

## 例子：管道

- 读口/写口同时可能有多个进程打开
  - 读管道时，若有数据则读出
  - 读管道时，若没有数据，则等待(阻塞)，读到数据后才返回
  - 写管道时，若管道已满，则等待(阻塞)，写入后才返回
  - 读管道时，若所有写口都被关闭，则返回size = 0
  - 写管道时，若所有读口都被关闭，则发送SIGPIPE信号



# 例子：日志文件

- 多个进程共享一个日志文件
  - 父子进程方式(write到同一个fd)
  - 重定向方式
    - 用O\_APPEND打开log文件写入
    - `seq 1 1000000 >> a.log`
- 思考题：并发写入的语义
  - write的并发行为是什么？
  - 上面的例子能保证正确输出到文件吗？



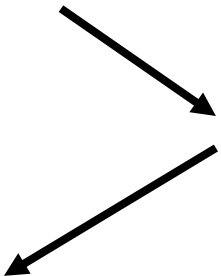
## 例子：目录管理

- 一个进程在扫描目录(readdir), 另一个进程把目录连同里面的文件都删了(rmdir).....
  - /proc/[pid]
  - pid可能刚才还存在, 下一秒(毫秒)就退出了
- 问题来了
  - 假设我在实现OJ的后台
  - 在程序运行结束后, 查看使用内存的峰值
  - 但程序结束了, /proc/[pid]/就没了.....

# 例子: TOCTTOU

- Time of check to time of use

```
if (access("file", W_OK) != 0) {  
    exit(1);  
}  
  
fd = open("file", O_WRONLY);  
write(fd, buffer, sizeof(buffer));  
  
symlink("/etc/passwd", "file");
```



# 综合例子

- 行为貌似非常明确的一组文件API
  - unlink/rmdir/rename – 删除文件/目录/重命名
  - open/read/write/close – 打开/读取/写入/关闭
- 那么.....
  - 如果有进程open了文件read/write; 然后文件被unlink; 又被创建(open)了同名文件, 各自会发生什么?
  - 同一个文件并发读/写的行为?
    - 多个进程append的行为
    - 父/子进程同时写入
    - 同时读/写的行为

# 多进程并发：操作系统视角

- 操作系统就是个C程序嘛
  - 管理系统中的程序执行：在中断驱动下进行上下文切换
  - 小case啦，就是课程作业
- 系统调用
  - 系统调用也用中断实现
  - 直接在中断里执行操作系统里write, fork, ...的代码(这时候可以访问操作系统里的数据啦)
- 但是处理中断(系统调用)的时候，还可能还有其他中断哦.....



# 多进程并发：操作系统视角

- 难题：并发系统调用的设计与实现

目前地球人还没有找到有效的方式解决这些问题

- 必须正确设计系统调用的并发语义

- 管道的读/写语义、文件的偏移量、写操作的原子性.....

- 必须正确实现并发系统调用

- 系统调用代码也可能被抢占(否则write巨大的数据机器就卡住了)
  - 操作系统必须在任意调度下都正确实现并发系统调用语义

- 必须在现代处理器上高效地实现并发系统调用

- 系统中可能有大量进程并发访问文件系统、I/O.....

# 多线程并发



# 我们已经有进程了.....

- 进程 = 执行的程序
  - 多进程分时共享处理器

早期UNIX  
的全部

- 进程之间可以通信
  - 文件、管道、套接字.....

- 但是在一个程序之内，我们有时也希望能管理多个共享内存的执行流

“Processes are the abstraction of running programs;  
Threads are the unit of execution in a process.”

—— Linux Kernel Development

# 多线程：为什么需要它？

- 利用多处理器的资源
  - 计算素数表：  $n$  个CPU，速度快  $n$  倍
- 分离程序中需要等待的部分
  - 考虑文本编辑器的三个功能：
    - 事件输入(等待I/O)
    - 语法高亮(消耗CPU)
    - 定时自动保存(写入磁盘)
  - 分为三个线程，在保存时能同时响应输入、完成语法高亮



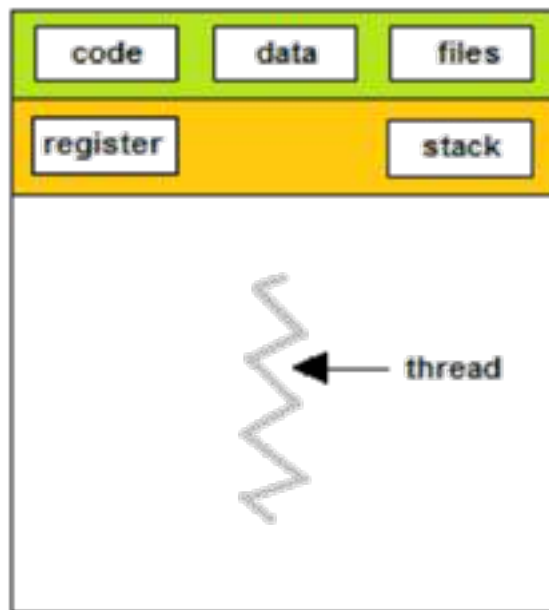


# 多线程的实现：共享地址空间的进程

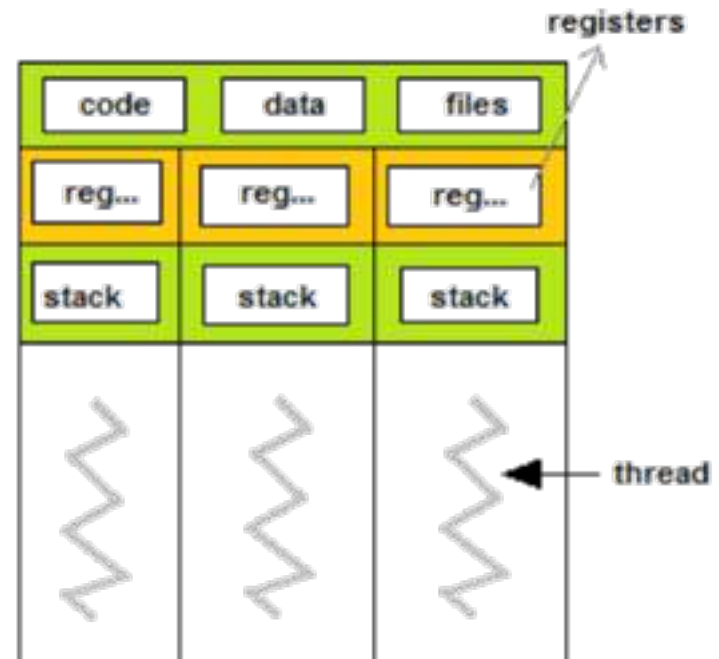
- 多个线程可以看成是系统中的若干进程
  - 每个进程都能独立执行
    - 拥有独立的寄存器 and 堆栈
  - 但通过  $VM(x)$  的设置共享同一份代码、数据
- 思考题：线程应该用哪种方式实现？
  - #1: 堆栈也在同一地址空间，但在不同位置
  - #2: 堆栈在不同地址空间，由  $VM(x)$  指定
- 这一句就说完并发部分的全部内容了(逃

# 进程与线程

- 线程  $\approx$  共享内存的进程
- 进程  $\approx$  单个线程



single-threaded process



multithreaded process

# 并发多线程：例子1 (假想中的保研面试题)

```
1 #define N 1000000000
2 int sum = 0;
3 void thread1() {
4     for (int i = 0; i < N; i++)
5         sum++;
6 }
7 void thread2() {
8     for (int i = 0; i < N; i++)
9         sum++;
10 }
```

- 运行结果：
  - -O0 (8.726s) sum = 1054212206
  - -O1 (0.388s) sum = 1000000000
  - -O2 (0.002s) sum = 2000000000

## 例子2: \$\$\$相关的问题

- `balance = 100, money = 100`

```
void deposit(int money) {  
    if (balance >= money) {  
  
        balance -= money;  
    }  
}
```

```
void deposit(int money) {  
    if (balance >= money) {  
        balance -= money;  
    }  
}
```

## 例子3：真实世界里的的问题

- 程序员真的也会犯这种错误耶！
  - 而且很多.....我们的自动工具都能找到.....

```
int ReadWriteProc(..) {  
    ...  
    PReadAsnc(&p);  
    io_pending = TRUE;  
    ...  
    while (io_pending) {  
        ...  
    }  
}
```

Diagram illustrating a race condition or bug in the code:

- The call to `PReadAsnc(&p);` in `ReadWriteProc` is linked by an arrow to the `DoneWaiting(...)` function.
- The assignment `io_pending = TRUE;` in `ReadWriteProc` is linked by an arrow to the `io_pending = FALSE;` line in `DoneWaiting`.
- The `while (io_pending)` loop in `ReadWriteProc` is linked by an arrow to the `io_pending = FALSE;` line in `DoneWaiting`.

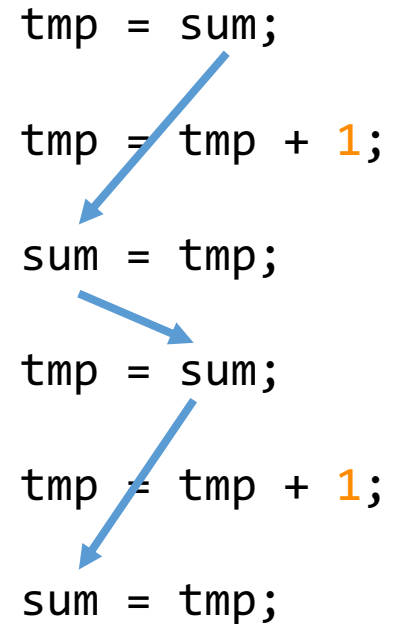
The `DoneWaiting(...)` function is defined as:

```
void DoneWaiting(...) {  
    // callback for PReadAsnc  
    ...  
    io_pending = FALSE;  
    ...  
}
```

# 共享内存并发：为什么这么麻烦？

- 顺序程序：数据流是直接确定的 `sum += 1;`  
`sum += 1;`
- 并发程序：共享变量数值可能来自其他线程
  - 世界观都被颠覆了！
  - 多年编程训练的技巧大多失效了

```
tmp = sum;  
tmp = tmp + 1;  
sum = tmp;  
tmp = sum;  
tmp = tmp + 1;  
sum = tmp;
```



这都是同一个sum

## 例子4：此处有黑人问号

- 实验结果(4 x Xeon X7460)

- 0.2%:  $x = 0, y = 0$
- 82.3%:  $x = 0, y = 1$
- 17.5%:  $x = 1, y = 0$
- 0%:  $x = 1, y = 1$

```
10 int volatile x = 0, y = 0;
11 atomic<int> tx, ty;
12
13 void thread1() {
14     x = 1;
15     ty = y;
16 }
17
18 void thread2() {
19     y = 1;
20     tx = x;
21 }
```

## 习题:

- 在刚才的例子中，我们希望反复运行
  - thread1() thread2() 同时执行10,000次
  - 打印出每次的tx和ty结果
  - 程序应该怎样写？

```
13 void test1() {  
14     x = 1;  
15     ty = y;  
16 }  
17  
18 void test2() {  
19     y = 1;  
20     tx = x;  
21 }
```

- 失败的尝试
  - T1: 循环100,000次调用test1()
  - T2: 循环100,000次调用test2()
- 加上sleep也不能保证test1() test2()并发执行