操作系统

第2章 2.5-2.6 线程

朱小军

南京航空航天大学计算机科学与技术学院

2025年春

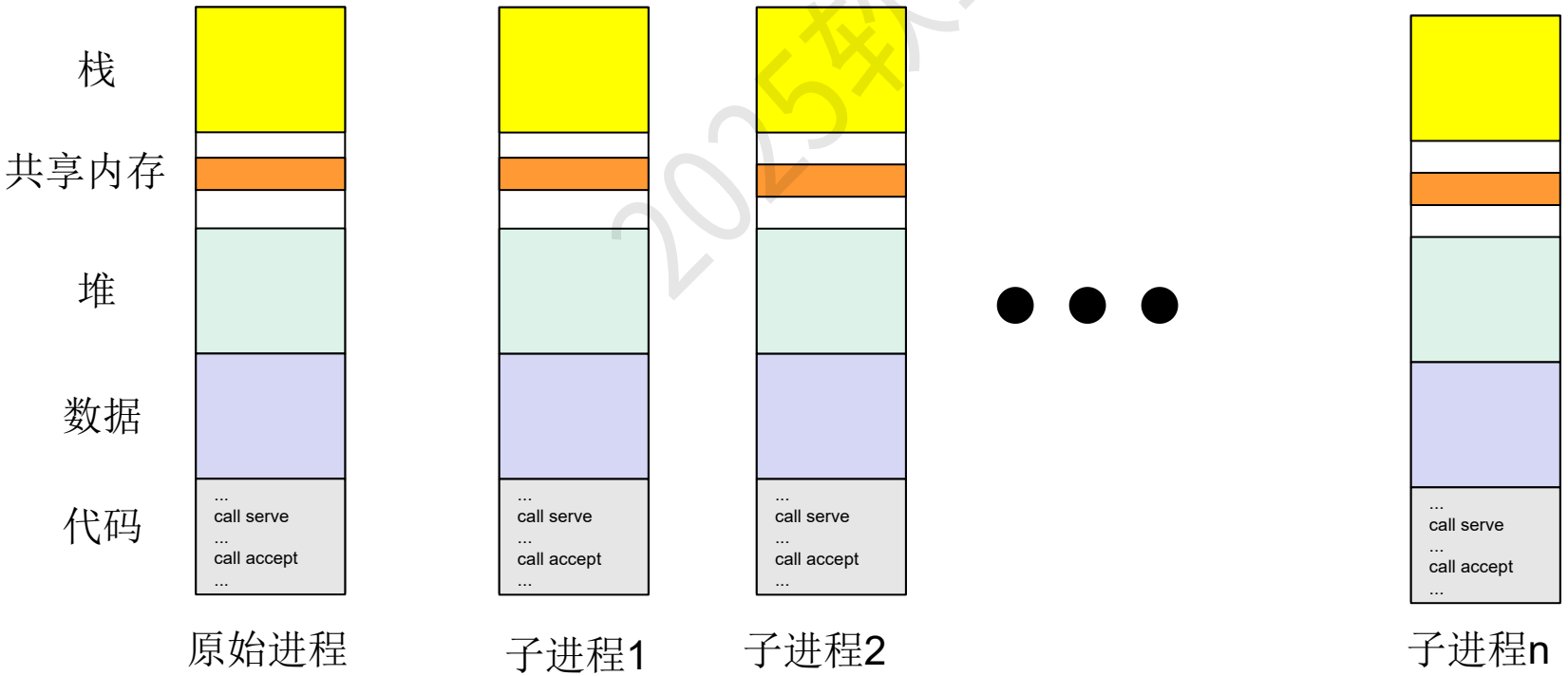
目录

为何要引入“线程”？

回顾：多进程网页服务器

多进程版本：

```
while( true){
        conn=accept( );
        创建一个新的进程执行serve( conn );}
```

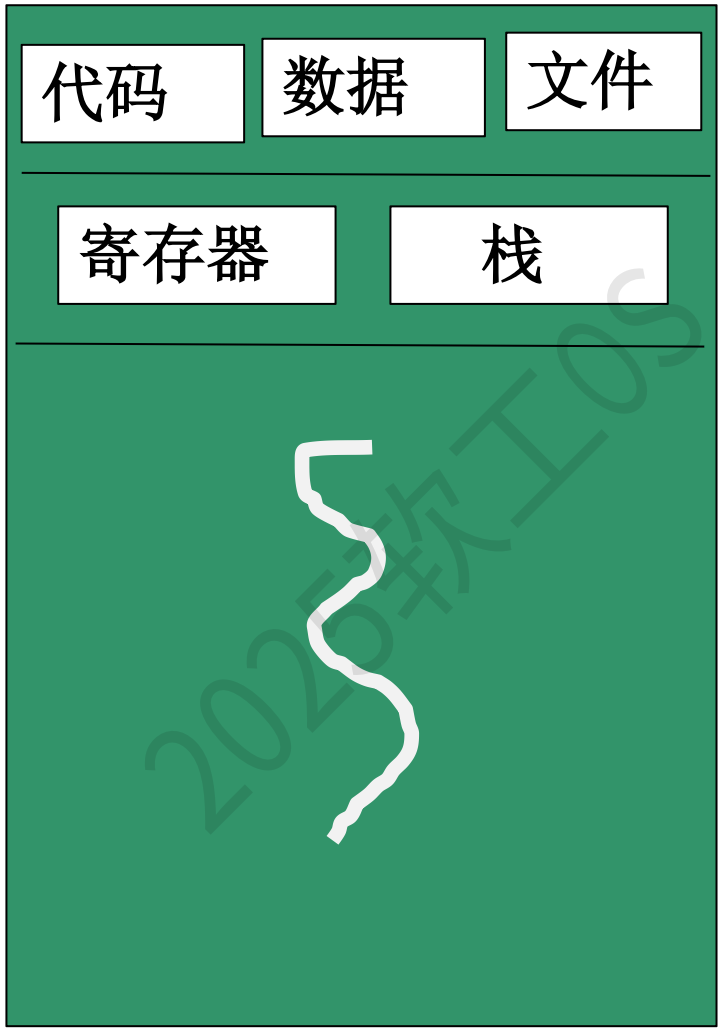


观察上面多进程方案的缺点

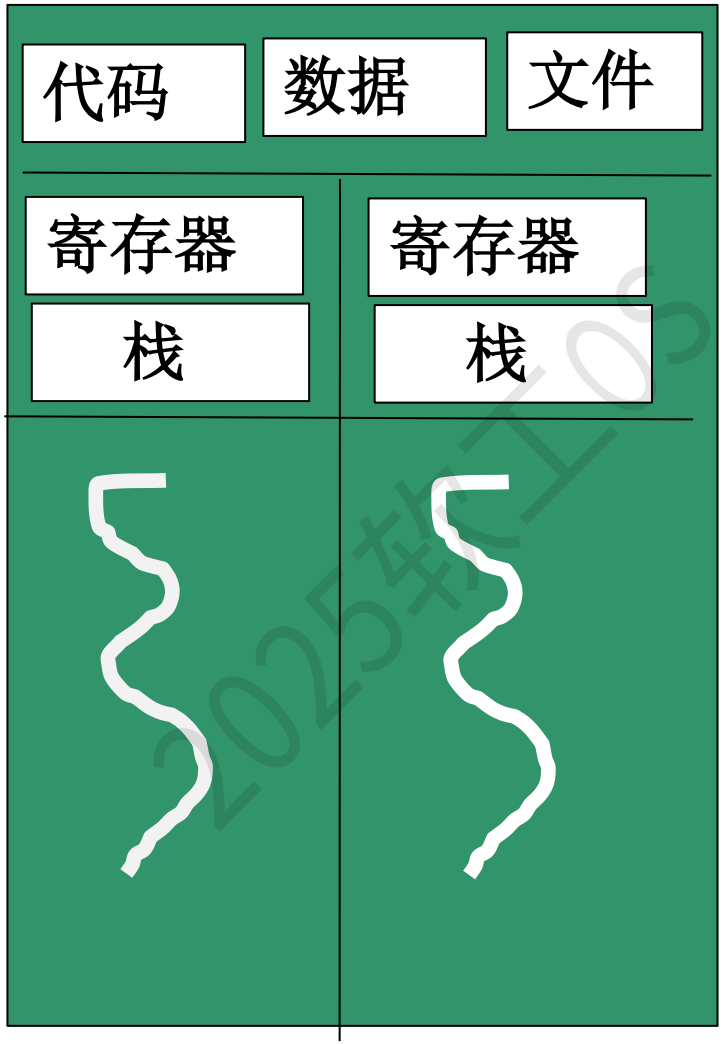
- 空间资源耗费大
  - 新的进程需要PCB
  - 地址空间（页表资源）等
- 时间资源
  - 创建进程本身消耗时间（如复制内存映像）
  - 切换进程开销大，需要切换地址空间以及内核栈
  - 进程之间传递数据需要内核介入（进程通信）
- 如果在某个编程比赛中，这种方式虽然能利用多核，但性能依然堪忧

优化思路

- 相同之处
  - 代码段完全一样；数据段分配一样（值可能变化）
  - 权限一样
  - 资源一样（文件等）
- 不同之处
  - 每个进程有不同执行状态PC、SP、通用寄存器
  - 栈不一样（用户栈、内核栈）
- 能否将进程概念中的“执行”进一步分离？
  - 进程：地址空间、权限、资源，等
  - 执行状态：PC，SP，通用寄存器，栈
- 执行状态命名为线程



（单线程）进程



（多线程）进程

线程：处理器调度的基本单位

- 同一个进程可以创建多个线程
- 多个线程共享地址空间、资源
  - e.g.，命令提示符窗口
- 多个线程在内存中的布局图
- 线程在运行过程中有三个状态
  - 就绪、执行、阻塞
- 操作系统如何管理线程？
  - TCB： Thread Control Block
    - 包含信息：所属进程、内核栈、线程状态等
      - 用户栈一般是隐含的（陷入内核前，sp寄存器指向的地方）
  - PCB也需改变，比如创建的线程列表

引入线程后，进程的状态？

- 平台相关，综合所有线程状态（Windows）、或者主线程、或者活跃线程（Linux）

目录

- 为何引入“线程”?
- 多线程与多进程的比较
- 线程的实现方式
- 线程库

多线程编程的优势

- 创建线程的开销比创建进程低
- (同一个进程内的)线程之间通信无开销
- (同一个进程的)线程之间切换时开销低
- 多进程程序可以改为多线程程序

多进程版本：

```
while( true){
        conn=accept( );
        创建一个新的进程
    执行serve( conn );
}
```

多线程版本：

```
while( true){
        conn=accept( );
        创建一个新的线程
    执行serve( conn );
}
```

多进程服务器的缺点弥补了吗？

- 空间资源耗费大
  - 新的进程需要PCB
  - 地址空间（页表资源）等
- 时间资源
  - 创建进程本身消耗时间（如复制内存映像）
  - 切换进程开销大，需要切换地址空间以及内核栈
  - 进程之间传递数据需要内核介入（进程通信）

多进程服务器

- 新的线程需要TCB
- 不需要新地址空间
- 创建线程不需要复制内存映像
- 切换线程只需要切换内核栈
- 线程之间传递数据不需要进程通信（全局变量即可）

多线程服务器

多线程的缺点？

- 若一个线程出错，则整个进程被强制终止
  - 有时可以通过提前注册信号避免，如捕获**SIGSEGV(segementation violation)**、**SIGFPE（floating-point exception）**
  - 用其他语言，如**Go**
- 共享地址空间，线程之间无防护，容易出现无意中的错误修改
  - 比如，一个线程损坏另一个线程的栈
  - 为什么无防护？做不到（技术上），一般也没必要（都是自己人）

多进程编程还是多线程编程？

- 二者本质上等价（解决同一个问题时），各有优缺点，适合不同场景
- 支持多线程的最主要理由
  - 创建线程的开销小
  - 通信无开销
- 支持多进程的最主要理由
  - 更好的资源隔离与保护：一个进程出错，其他正常运行
  - 更好的编程模式（强迫程序员思考：共享什么）
- 发展趋势
  - 创建进程的开销逐渐向创建线程靠拢

目录

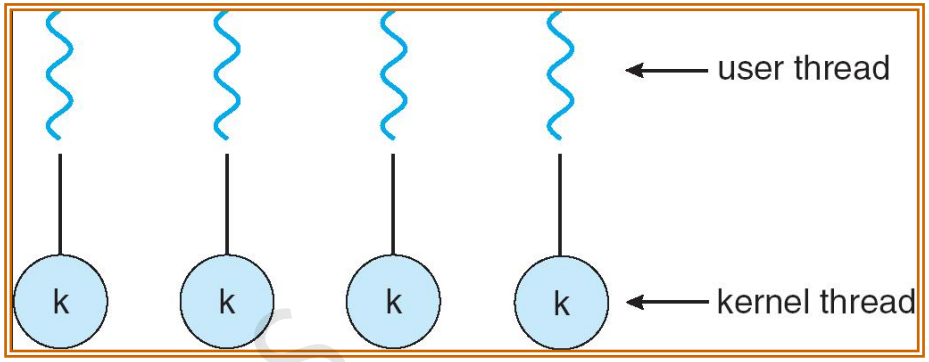
- 为何引入“线程”?
- 多线程与多进程的比较
- 线程的实现方式
- 线程库

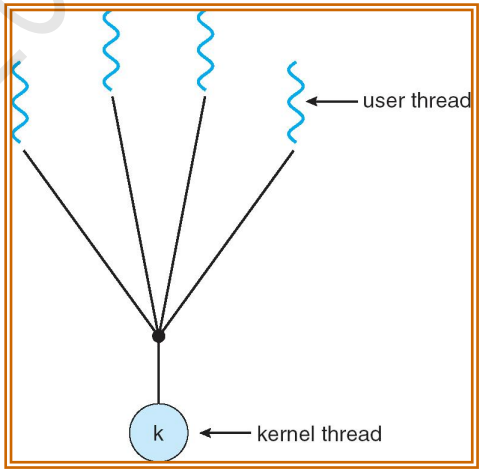
内核级线程 vs 用户级线程

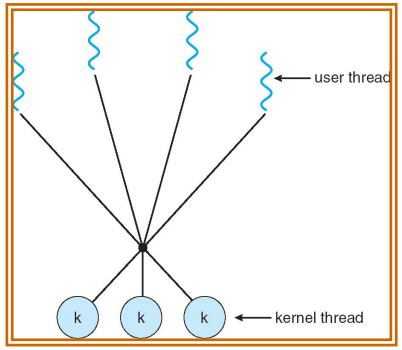
- 内核级线程（或内核支持线程）
  - 内核自身支持的线程（前面讲的内容都是内核级线程）
  - 内核知道线程的存在，有内核数据结构（TCB、内核栈），内核会调度它们执行
  - 可以实现多线程并行执行
- 用户级线程
  - 在内核之上，编程人员编写代码时使用，由库函数提供（比如协程，libco，腾讯开源）
  - 内核不知道用户级线程的存在，不管理它们
  - 早期OS不支持多线程时，用库函数提供线程支持，如Green（1997-2000年,Java）,名气大，导致用户级线程别名Green Thread，绿色线程
- 在具体的实现中，用户级线程必须映射到内核级线程上，有三种方式

三种映射方式

- 一对一
  - 大多数
- 多对一
  - 系统调用、阻塞问题需考虑：某线程调用scanf，内核认为是进程调用的，阻塞整个进程
  - 历史上有，现在很少见
- 多对多
  - Go语言的Goroutine

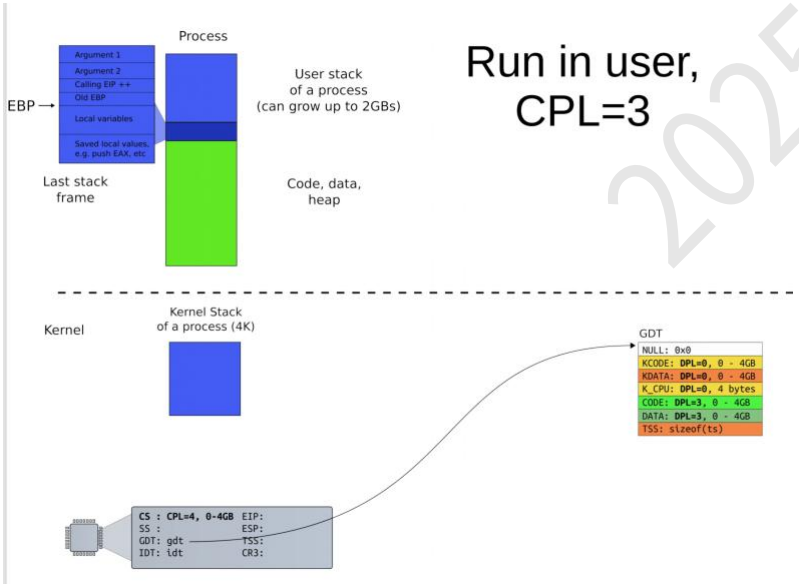




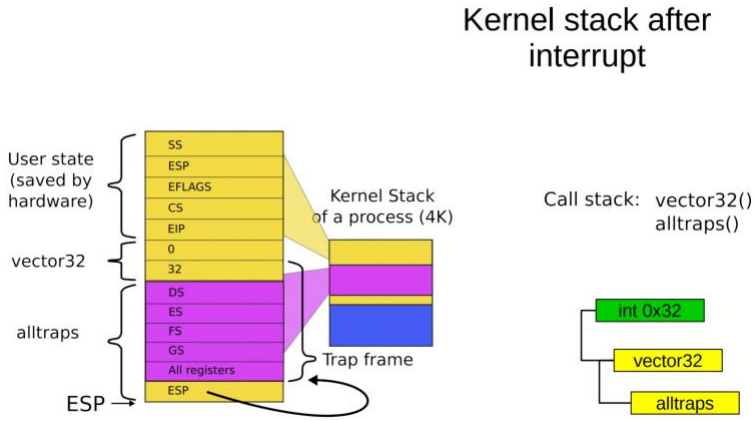


xv6的内核级线程

- xv6不支持多线程，每个进程对应一个线程
- 每个进程创建时，分配了内核栈，对应了内核级线程
- CPU在用户态运行时，内核栈为空
- 当中断、系统调用到达时，内核栈被填上内容，CPU运行在内核栈上



CPU在用户态时，内核栈为空



CPU在内核态时，内核栈中保存了进程陷入内核态前CPU各寄存器的值(trapframe)

目录

- 为何引入“线程”?
- 多线程与多进程的比较
- 线程的实现方式
- 线程库

线程库

线程库

- 线程库为程序员提供创建和管理线程的API
- 两种实现方法
  - 线程库完全在用户空间中（没有内核支持）
  - 内核级线程库，由操作系统支持
- 常用的线程库
  - Pthread
  - Win32线程
  - Java线程

Pthread

- 可能是用户级也可能是内核级
- **POSIX标准 (IEEE 1003.1c) 为线程创建和同步定义的API**
- 是规范，不是实现
- **API说明了线程的行为，具体实现交给操作系统**
- 在类**Unix**操作系统中很常见，如**Solaris**，**Linux**，**Mac OS X**

创建与退出

```
int pthread_create(
        pthread_t *thread, //保存thread ID
            const pthread_attr_t *attr, //传入属性
             void *(*start_routine)(void*),//执行函数
                                    void * arg//函数参数
 );
void pthread_exit(
                    void *value_ptr//返回值
  )
```

等待退出

```
int pthread_join(pthread_t thread, //等待的线程
        void **value_ptr  //*value_ptr=返回值
);
```

求和

```
#include<pthread.h>
#include<stdio.h>
int sum=0;
void *runner(void *param){
        for(int i=0;i<*(int *)param;i++)
                sum+=i;
        pthread_exit(0);
        return 0;
}
int main(){
        pthread_t tid;
        int end;
        scanf("%d",&end);
        pthread_create(&tid,null,runner,&end);
        pthread_join(tid,null);
        printf("sum=%d\n",sum);
         return 0;
}
```

多线程服务器的实现

- 如果用Pthread如何实现？
- 可以无限制创建线程吗？
- 线程池
  - 为何要引入？

小结（2025考研大纲）

- 线程的基本概念
- 线程的状态与转换
- 线程的实现方式
  - 内核支持的线程
  - 线程库支持的线程
- 线程的组织与控制