操作系统原理

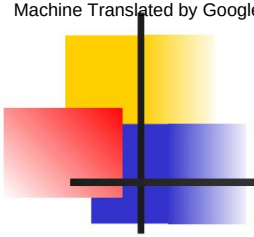
操作系统原理

进程和线程概念

李旭东

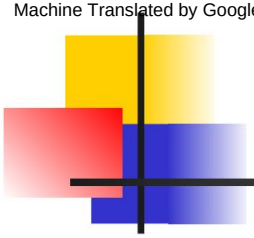
leexudong@nankai.edu.cn南

开大学



目标

程序的执行进程概念线程概念

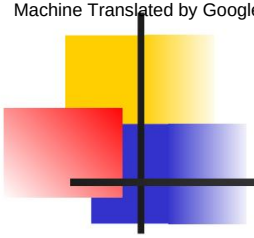


程序执行

程序 I（输入）、C（计算）、P（打印） I 类顺序过程

程序执行

程序 I（输入）、C（计算）、P（打印） 第 II 类并发进程



多道编程

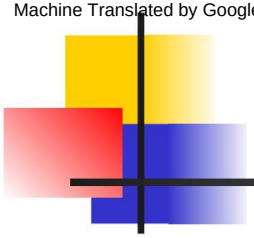
程序程序作业

任务

…

并发

单 CPU:伪并行多处理器:并行



过程

进程只是

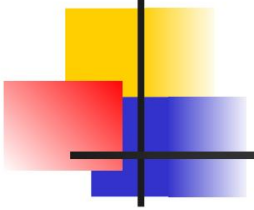
执行程序

包括程序计数器、寄存器和变量的当前值

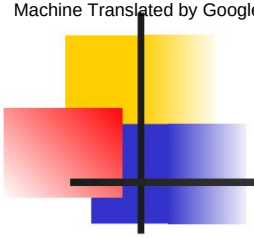
虚拟 CPU

主存储器

内核对象



内存中进程



函数堆栈框架

（上一个功能0）

（功能1）

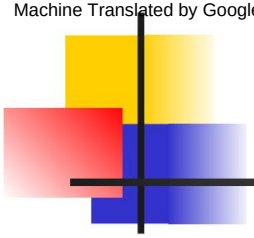
(功能2)

(功能1)

（功能1）

多进程

图 2-1。(a)四个程序的多道程序设计。(b)四个独立的、连续的过程的概念模型。(c)一次只能激活一个程序。



程序与流程

程序 vs 过程　　食谱菜

谱 vs 烹饪烹饪调

流程创建

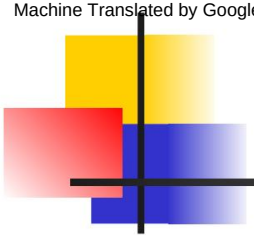
四个主要事件系统初始化正在运行的进程执行进程创建系统调用

用户请求创建新进程启动批处理作业函数　UNIX：fork、execve

Windows(Win32):CreateProcess

C 程序分叉独立进程

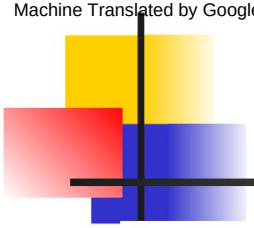
```
int 主要()
{
pid_t pid; /* fork 另
        一个进程 */ pid = fork(); if (pid < 0) { /* 发生错误 */


                fprintf（stderr，“Fork失败”）;退出（-1）；

        }
        else if (pid == 0) { /* 子进程 */ execlp(   /bin/ls   ,   ls   , NULL);

        }
        else { /* 父进程 */
                /* 父进程将等待子进程完成 */ wait (NULL);


                printf (   孩子完成   );

        } printf(“确定”);
}
```



进程终止

导致进程终止的事件

正常退出（自愿） 错误退出（自愿） 致命错误（非自愿） 被他人杀死（非自愿）

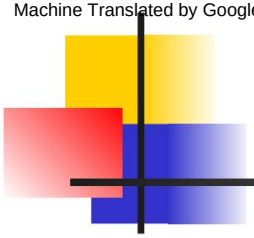


进程层次

UNIX

树

进程组



进程层次

Windows 无

层次结构所有进程

都是平等的句柄

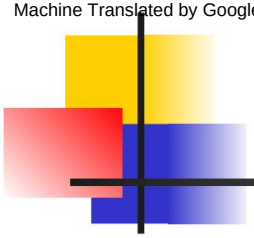
父进程用它来控制子进程

进程状态

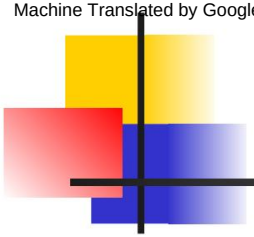
例如

cat 文件1 文件2 文件3 | grep osp | wc -l

图 2-2。进程可以处于运行、阻塞或就绪状态。
这些状态之间的转换如图所示。

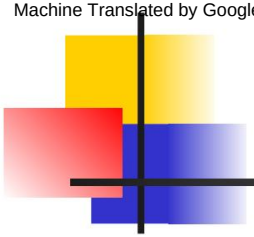


进程状态



流程实施

进程控制块 (PCB) 进程表:PCB 数组



流程实施

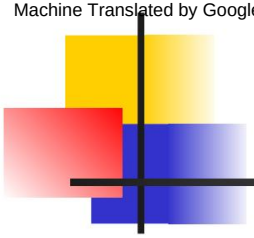
调度器

流程实施

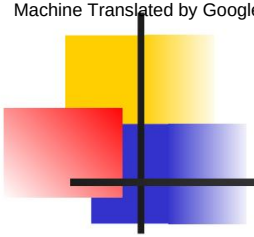
通过中断切换进程

图 2-5. 操作系统最低层框架

发生中断时系统会执行的操作。



CPU 在进程间切换



多道编程建模

CPU 利用率

=1 – pn

P:等待 I/O 完成的时间

多道编程建模

例如

一台计算机　512MB

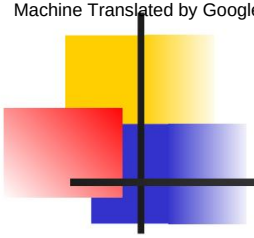
内存　128MB 用于操作

系统,128MB/进程　80% 时间用于等待 I/O

CPU 利用率 = 1-0.83 = 49%

添加第二条内存:512MB　CPU利用率 =1-0.87=79%

添加第三条内存:512MB　CPU利用率=?

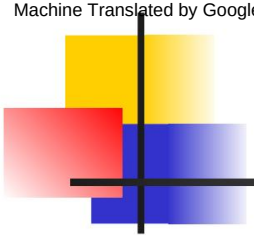


线

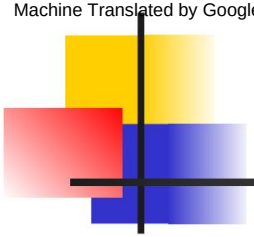
动机

在传统操作系统中,每个进程都有一个
地址空间和单个执行控制单元

但是…:文字处理器



单线程和多线程进程



多线程的好处

多线程编程

响应性资源共享经

济性多处理器架构的

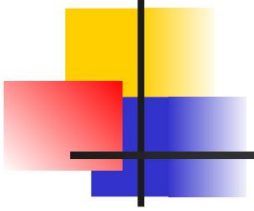
利用率

多线程用例

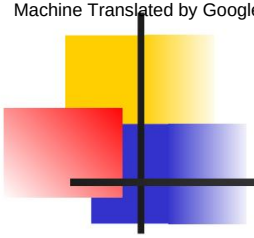
图 2-8。多线程 Web 服务器。

多线程用例

图 2-9。图 2-8 代码的粗略轮廓。(a)调度线程。(b)工作线程。



构建服务器的三种方式

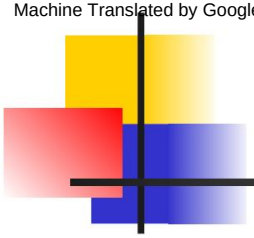


有限状态机（FSM）

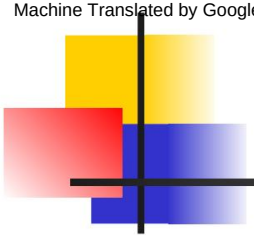
例如旋转闸门

经典线程模型

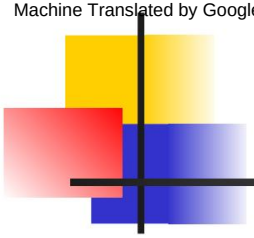
图 2-12。第一列列出了进程中所有线程共享的一些项目。第二列列出了每个线程私有的一些项目。



每个线程都有自己的堆栈



POSIX 线程



示例 1:POSIX 线程

示例 2a:POSIX 线程

```
#include <pthread.h>
typedef struct ST_ThreadArgs{ int
  m_index; char
  *m_path;
  pthread_t m_id; }
THREADARGS, *PTHREADARGS;

void * my_thread (void *aArgs)
  { PTHREADARGS mArgs = (PTHREADARGS) aArgs; if
  (mArgs!=NULL)
{ mArgs->m_id=pthread_self();
printf (   在线程中:你好,第 %d 个线程（%ld）,m_path 的值为 %s\n   , mArgs->m_index,
  mArgs->m_id, mArgs->m_path); }else{ fprintf(stderr,   在线
  程中
   （%ld）:args 为 null\n   ,pthread_self()); } pthread_exit (NULL); }
```

示例 2b:POSIX 线程

```
int main（int argc,char *argv []){
  int i;
  pthread_t *threadsArray;
  THREADARGS *argsArray; int
  status;
  void* res=NULL;
  printf (   父级 %d: begin\n   , getpid ()); printf
  (   %s 的参数列表:\n   , argv[0]); for (i = 0; i < argc;
  i++)

    { printf (   %3d %s\n   , i，  argv[i]); }

  如果 (argc <= 1) 返回
  0;threadsArray = (pthread_t *) malloc (sizeof (pthread_t) * (argc - 1));argsArray =
  (THREADARGS *) malloc (sizeof (THREADARGS) * (argc - 1));
```

示例 2c:POSIX 线程

```
  对于 (i = 1; i < argc; i++)
{ argsArray[i-1].m_index=i-1;
argsArray[i-1].m_path=argv[i];
*(threadsArray+i-1)=-1;
status = pthread_create(threadsArray+i-1, NULL,
my_thread, (void*)(argsArray+i-1)); if
(status!=0)
{ fprintf(stderr,   无法创建第 %d 个线程,代码为 %d\n   ,i-1,
status); break; }



    }
```

示例 2d:POSIX 线程

```
i=1;
while(i<argc)
{ status=pthread_join(threadsArray[i-1],&res); if
(status!=0)
{ fprintf(stderr,    第 %d 个线程连接失败\n    , i-1); }

else{ printf(    第 %d 个线程 (%ld) 返回 %ld\n    , i-1,
argsArray[i-1].m_id, (long)status); } i++; }
```

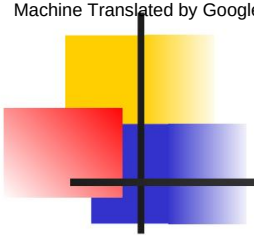
示例 2e:POSIX 线程

```
释放（threadsArray）；
释放（argsArray）；

printf (　父级 %d: 已退出\n　, getpid ()); 返回
EXIT_SUCCESS; }
```

·编译:gcc -o multithread multithread.c -lpthread

示例 2f:POSIX 线程

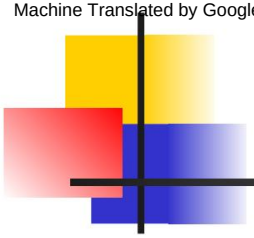
```
$ ./multithread /home /usr abc 父级
26643:开始 ./multithread
的参数列表:0 ./multithread 1 /home 2 /
   usr 3 abc



在线程:你好,第二个线程（140133858793024)中,m_path 的值为 abc
在线程:hello第0个线程（140133875578432)中,m_path的值为/home

在线程:你好第 1 个线程（140133867185728)中,m_path 的值为 /usr
第 0 个线程（140133875578432)返回 0
第 1 个线程（140133867185728)返回 0
第 2 个线程（140133858793024)返回 0
父级 26643:已退出
$
```



不同类型的线程

用户级线程

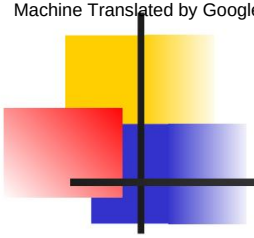
在用户空间



不同类型的线程

内核级线程

在操作系统内核中



不同类型的线程

混合实现线程内核线程和用户线程

混合实现线程I

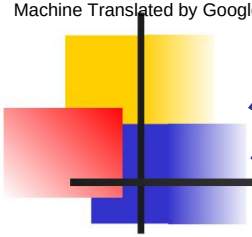
男:女（男:1、1:1、男:女）

混合实现线程 II

❖协程 协程 (M:N) ❖ Golang

GoLang 的调度器内部有三个重要的结构:M、P、S　　M:代表内核级 OS 线程；　❖ G:代表一个goroutine,即用户级线程　　,它有自己的栈、指令指针和其他信息（通道等）；

P: 进程在用户级别的调度器中,使 go 代码在一个线程上运行,它是实现从 M:1 到 M:N 的映射。



混合实现线程 II

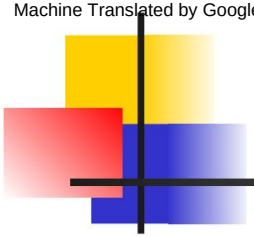
- 协程 协程 (M:N)
- Golang

示例:goroutine

包主要导入“fmt”

```
func main()
{ done := make(chan bool, 2)
fmt.Println(    Hello, 世界    ) go
loop(    A    , done)
loop(    B    , done) <-
done <-
done
close(done)
fmt.Println(    \nEnd.    ) }
```

```
func loop(s_pre string, ch chan bool) { for i := 0; i <
20; i++ { fmt.Printf(    %s_%d     ,
s_pre, i+1) } ch <- true }
```

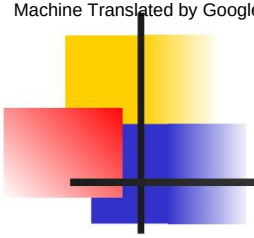


线程问题

fork() 和 exec() 系统调用的语义线程取消

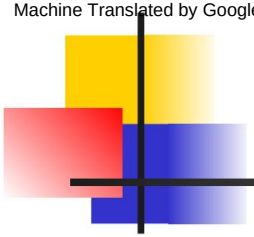
信号处理线程池线程特定

数据调度程序激活



fork() 和 exec() 的语义

fork() 是否只重复调用
线程还是所有线程?



线程取消

在线程完成之前终止它

两种通用方法：

同步取消立即终止目标线程

延迟取消允许目标线程定期检查是否应该取消

信号处理

在 UNIX 系统中,信号用于通知进程发生了特定事件信号处理程序
用于处理信号信号由特定事件生成信号被传

递给进程信号被处理选项： 将信号传递给该信号适用
的线程将信号传递给进程中的每个线程将
信号传递给进程中的某些线程分配
特定线程来接收进

程的所有信
号

线程池

在线程池中创建一定数量的线程,用于等待工作

优点:

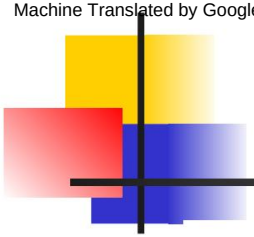
使用现有线程处理请求通常比创建新线程稍快

允许线程数

要绑定到池大小的应用程序

弹出主题

图 2-18。消息到达时创建新线程。(a)消息到达之前。(b)消息到达之后。

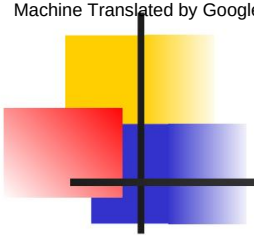


TSD:线程特定数据

- 允许每个线程拥有自己的数据副本
  - 当你无法控制线程创建过程时很有用（即,当使用线程池时）

使单线程代码变为多线程

图 2-19。线程之间因使用全局变量而产生的冲突。



使单线程代码变为多线程

图 2-20。线程可以拥有私有全局变量。

线程调度程序激活

调度许可

目标

模拟内核线程的功能， 但具有更好的性能和更大的

灵活性通常与在用户空间中实现的线程包相关

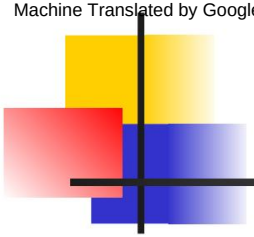
避免用户空间和内核空间之间不必要的转换

Upcall 上行

虚拟处理器

运行时系统

? 第 n 层不能调用第 n + 1 层中的过程



概括

流程

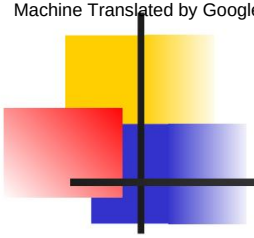
线程

进程与线程

内核线程与用户线程

重量级进程 vs 轻量级进程 vs Fiber

问答?



测验

下列哪个操作系统不支持线程？

麦金塔

Windows NT

Windows95~Windows2000

Solaris

红外

AIX

OS/2

数字 UNIX　　Linux