操作系统 Operating Systems

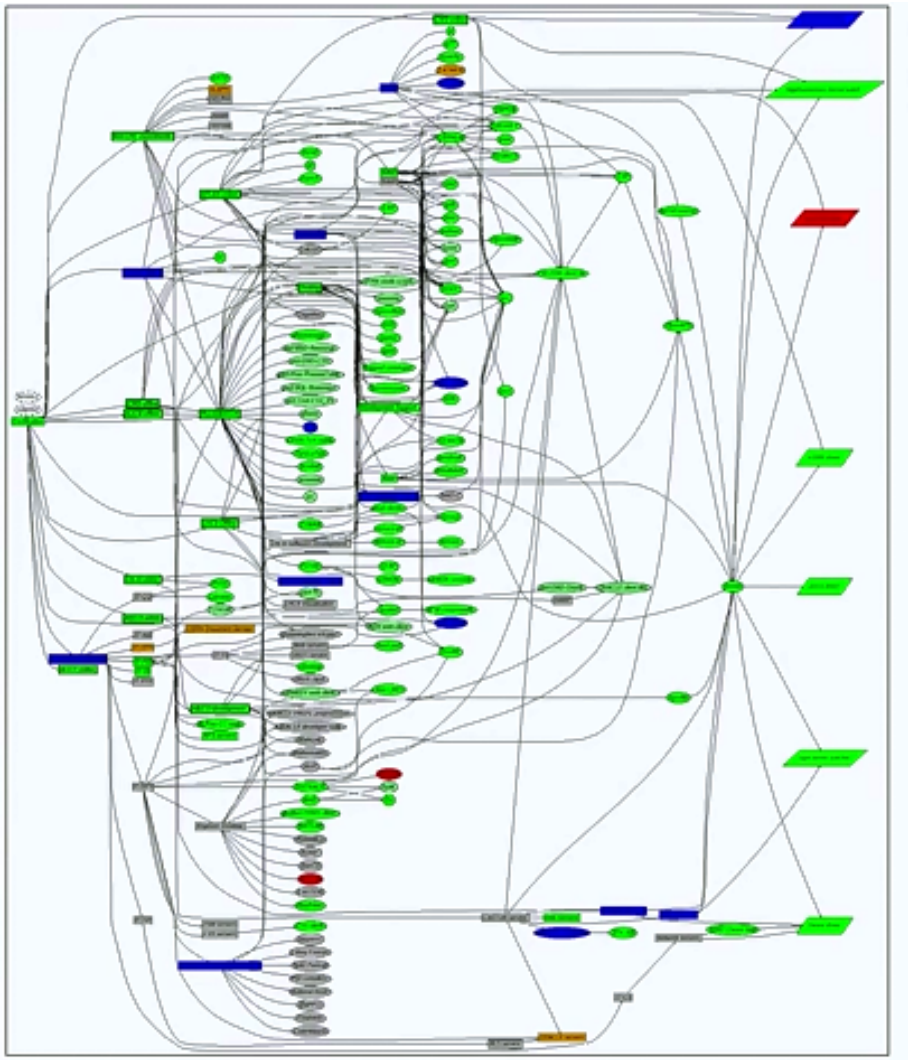
L13 操作系统的那棵“树”

L13 The Tree of OS

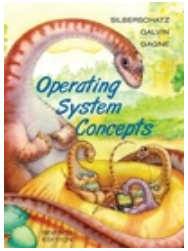
lizhijun_os@hit.edu.cn

授课教师：李治军 综合楼411室

Linux Kernel Source Tree

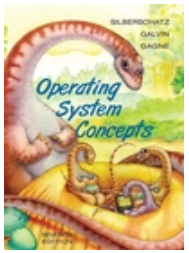


Linus Torvalds

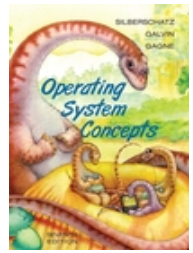


The mind is not a vessel that needs filling, but wood that needs igniting!

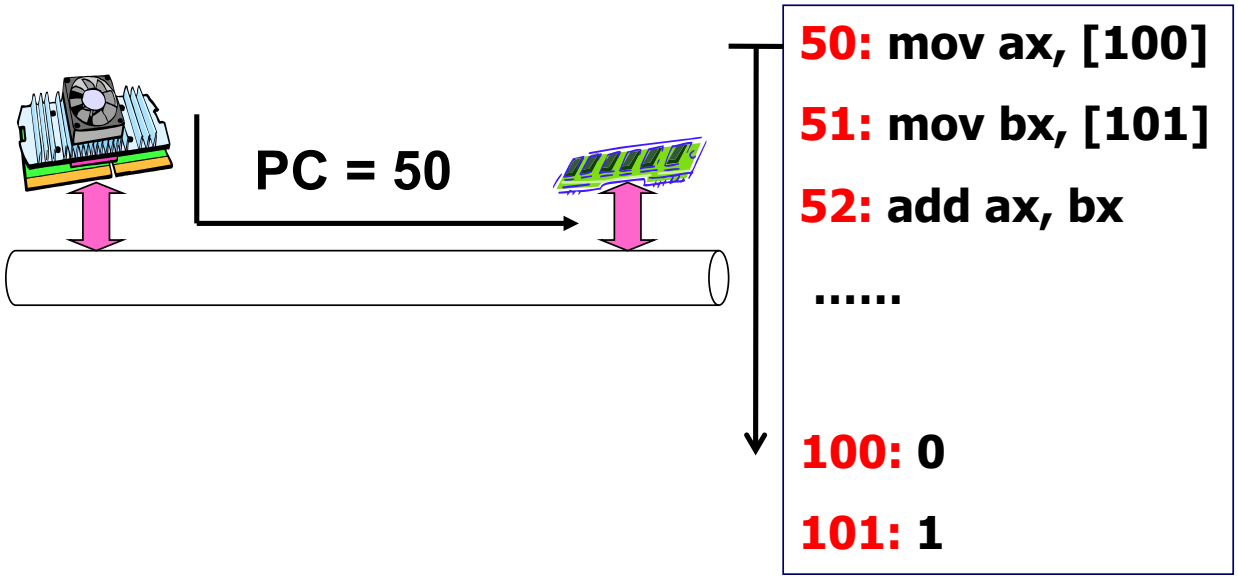
Plutarch quote



IDEA

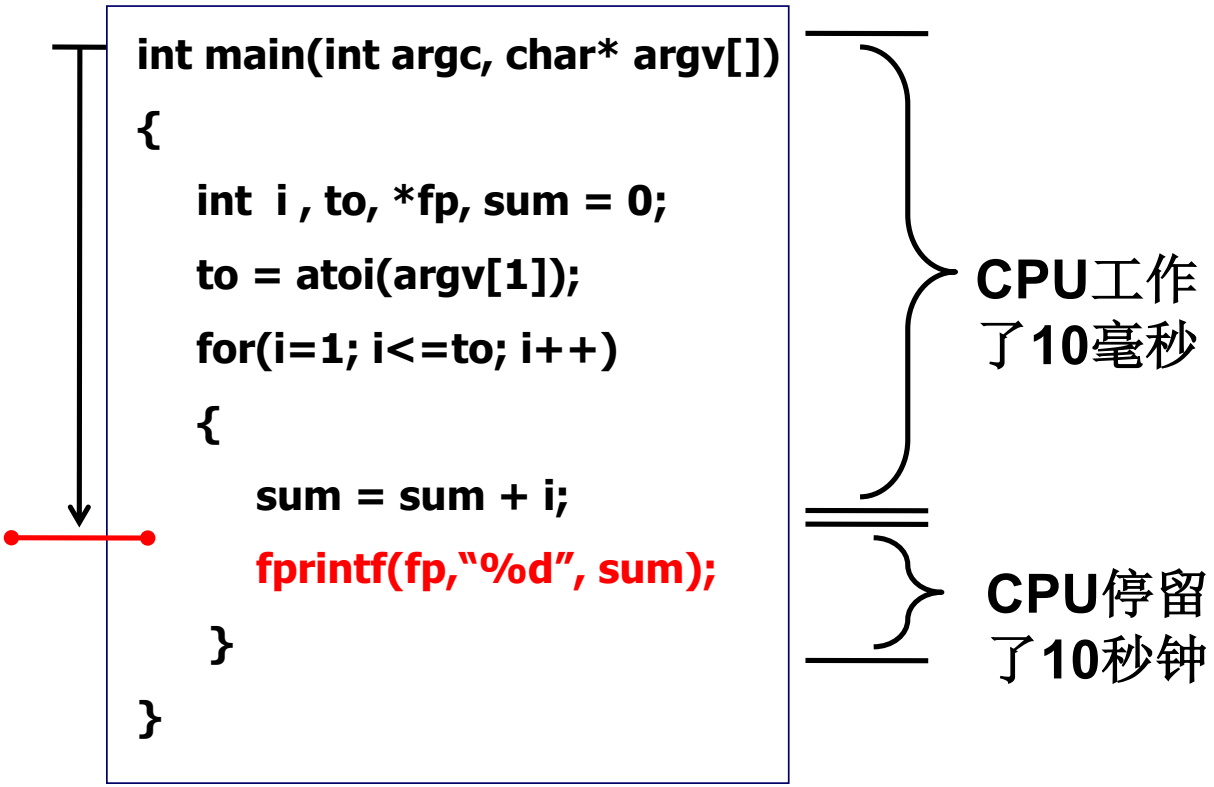


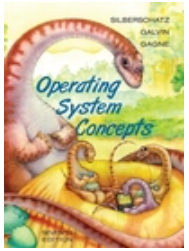
运转CPU


PC = 50
50: mov ax, [100]
51: mov bx, [101]
52: add ax, bx
.....
100: 0
101: 1

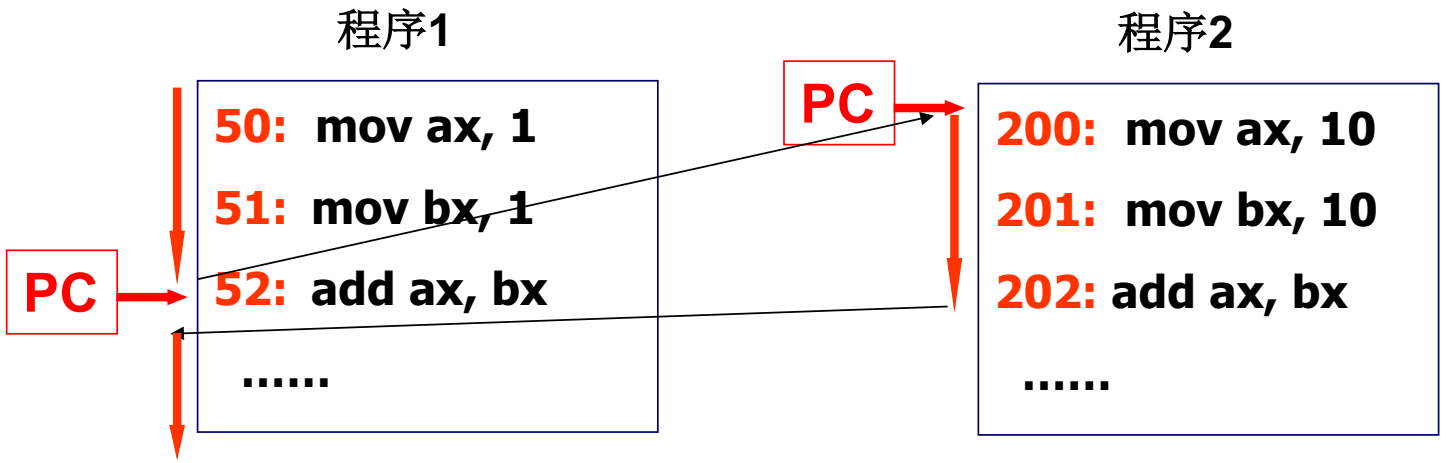

Operating System Concepts

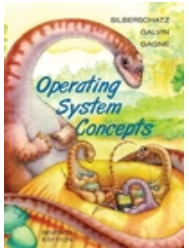
CPU没有好好运转


int main(int argc, char* argv[])
{
int i , to, *fp, sum = 0;
to = atoi(argv[1]);
for(i=1; i<=to; i++)
{
sum = sum + i;
fprintf(fp,"%d", sum);
}
}
CPU工作了10毫秒
CPU停留了10秒钟

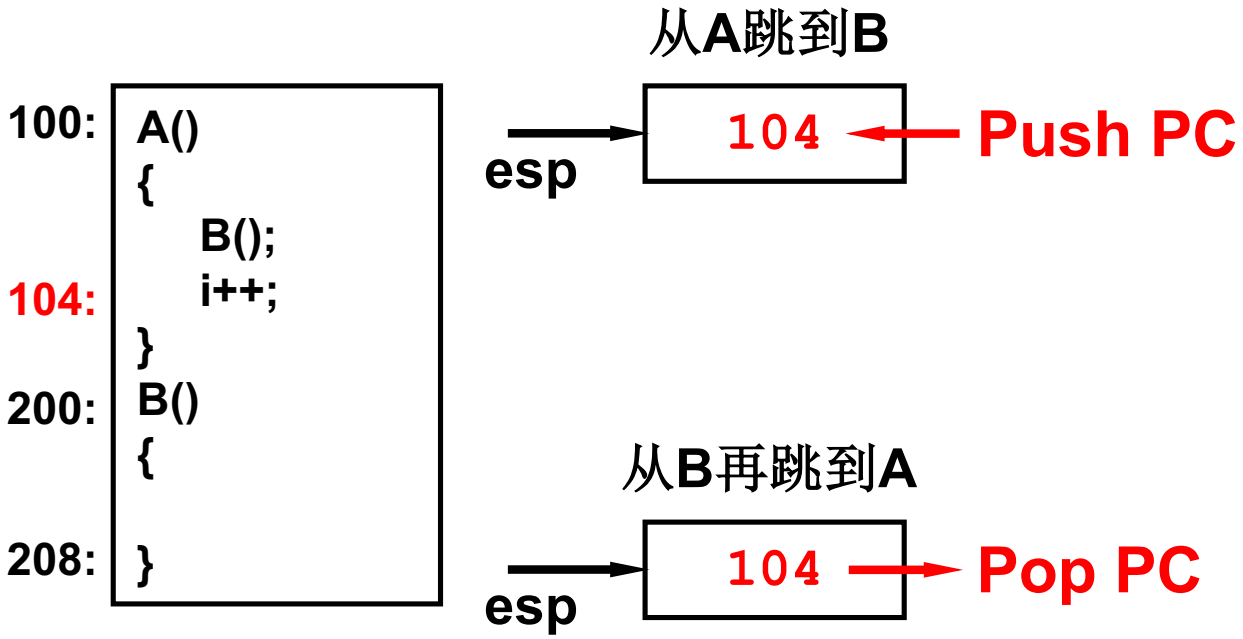

Operating System Concepts

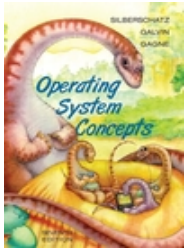
得让CPU好好运转


程序1
程序2
PC
PC
50: mov ax, 1
51: mov bx, 1
52: add ax, bx
.....
200: mov ax, 10
201: mov bx, 10
202: add ax, bx
.....

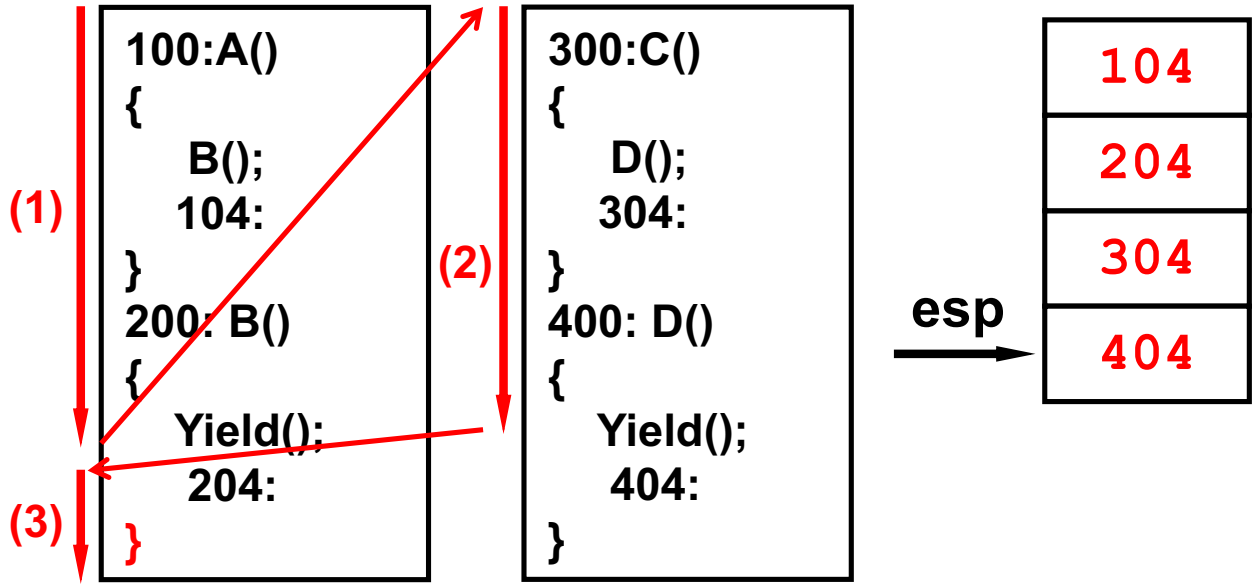

Operating System Concepts

从A跳到B我们并不陌生

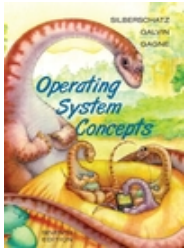




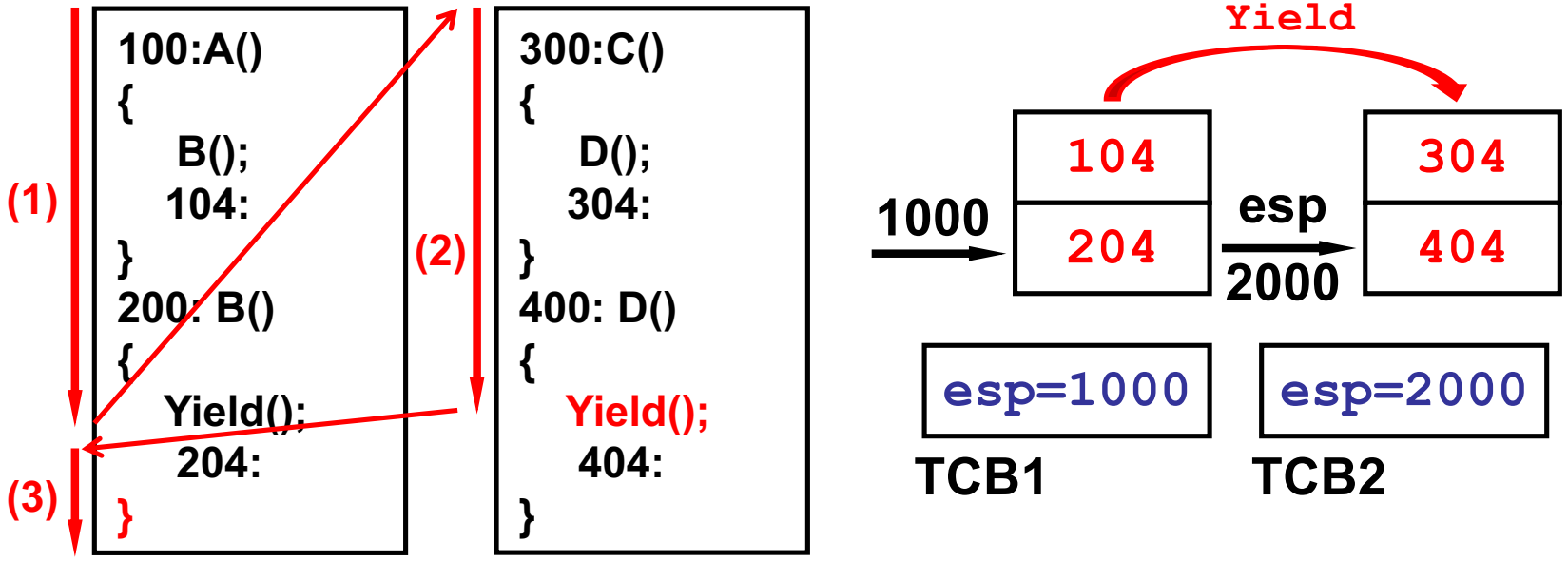
一个栈+Yield造成的混乱



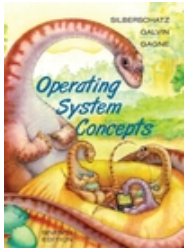
面对这样的栈你怎么可能从B顺利的回到A?



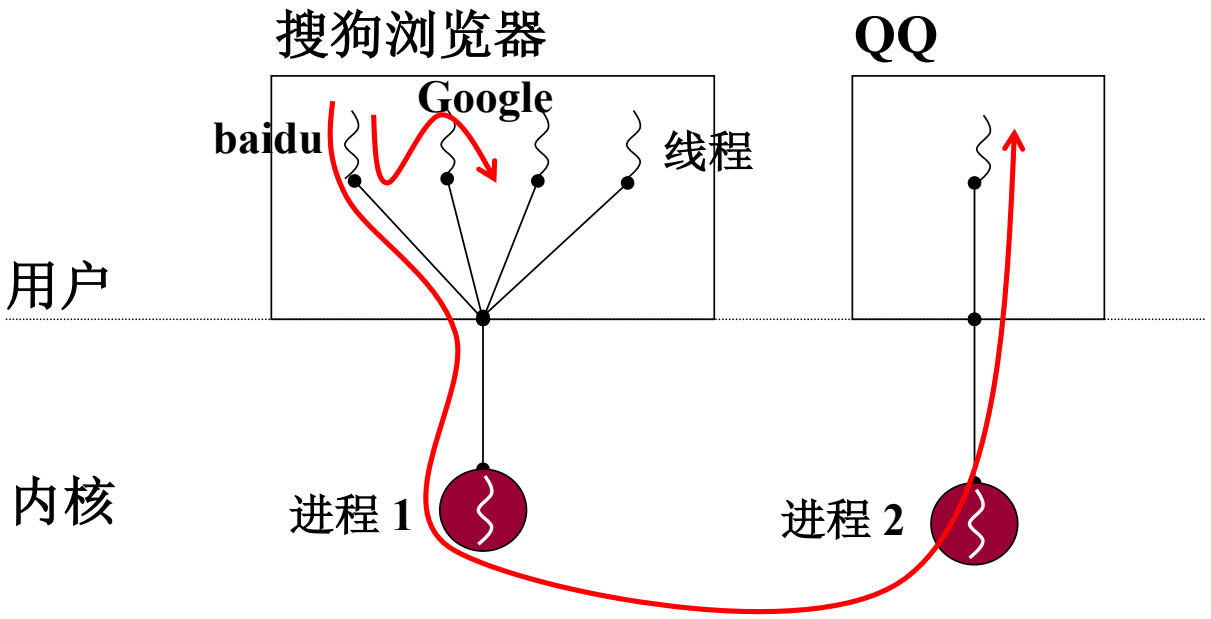
两个栈+两个用户TCB

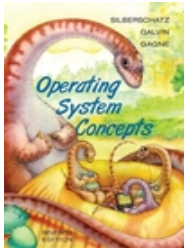


Yield()找到下一个TCB→找到新的栈
→切到新的栈

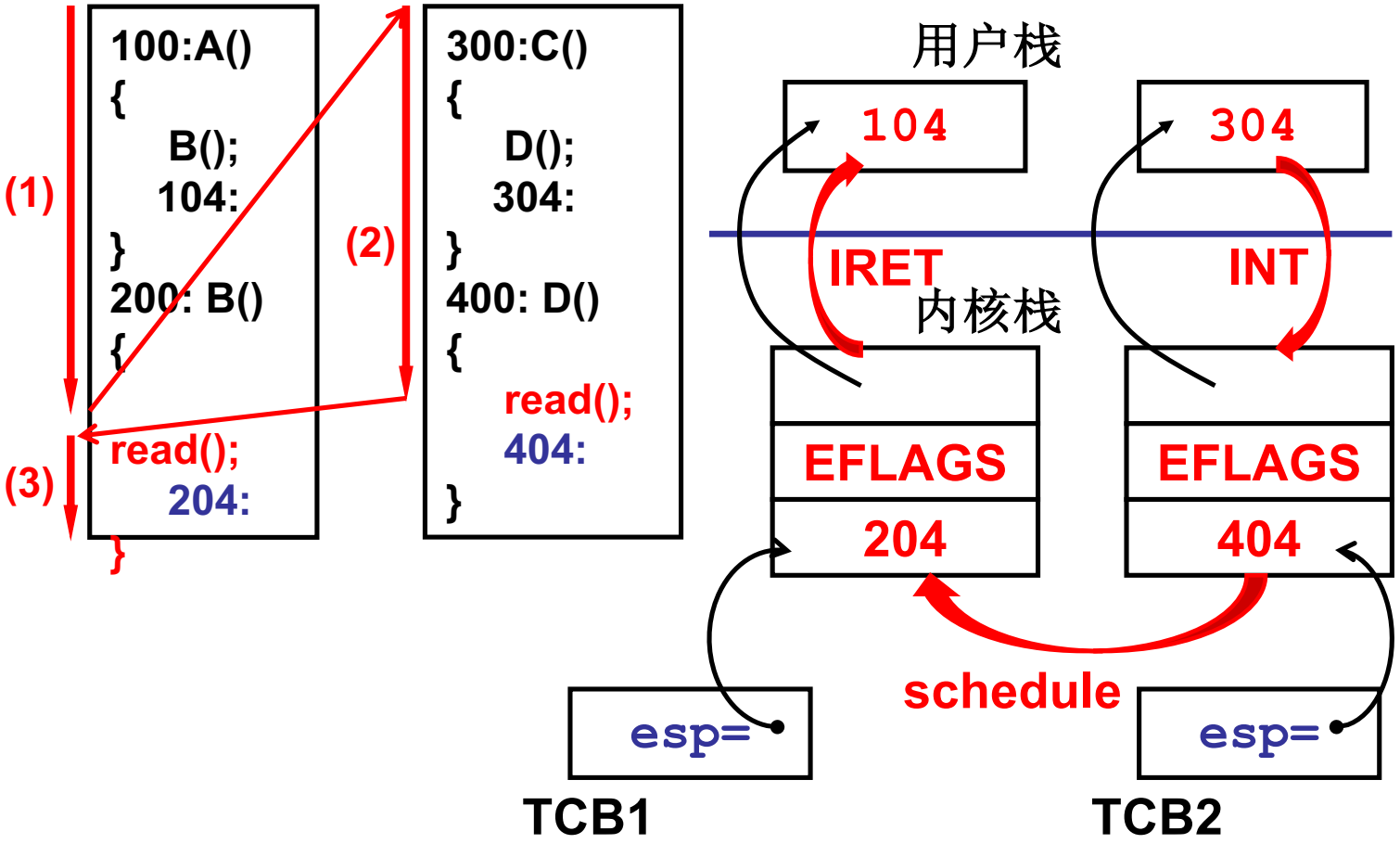


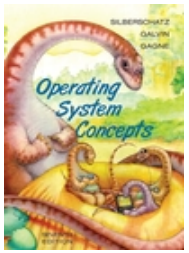
一直在用户态那怎么行?


搜狗浏览器
QQ
Google
baidu
线程
用户
内核
进程 1
进程 2


Operating System Concepts

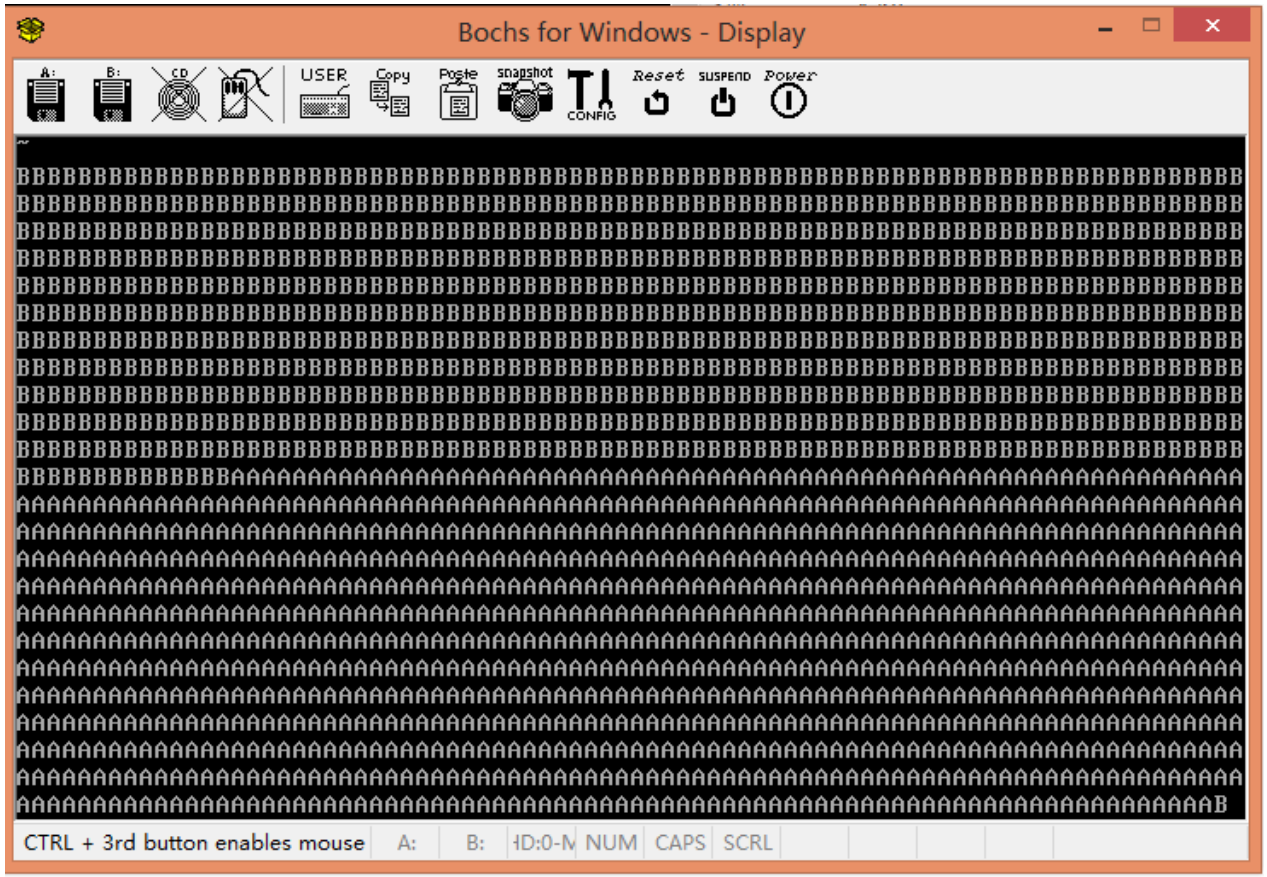
引入内核栈的切换

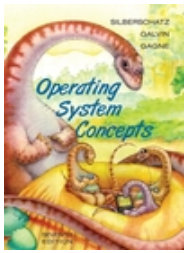

100:A()
{
B();
104:
}
200: B()
{
read();
204:
}
300:C()
{
D();
304:
}
400: D()
{
read();
404:
}
(1)
(2)
(3)
用户栈
104
304
IRET
INT
内核栈
EFLAGS
204
EFLAGS
404
schedule
esp=
esp=
TCB1
TCB2


Operating System Concepts

到实现idea的时候了

从一个简单、清晰、明确的目标开始

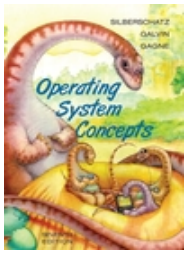




从用户代码开始

AB.c

```
main()
{
  if(!fork()){while(1)printf("A");}
  if(!fork()){while(1)printf("B");}
  wait();
}
```

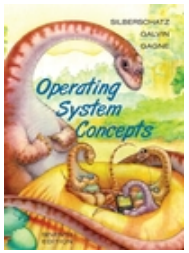


程序是什么?就是人的思维的C表达

```
main(){
      mov __NR_fork, %eax
      int 0x80
100:  mov %eax, res
      cmpl res,0
      jne 208
200:  printf("A")
      jmp 200
208:  ...
304:  wait()
}
```

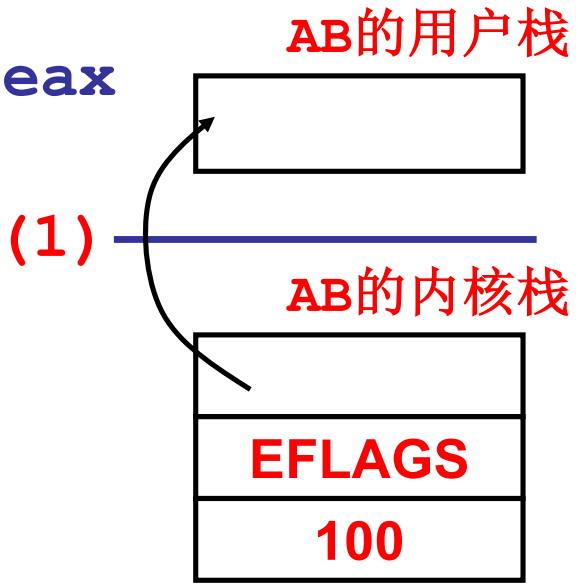
把自己变成计算机跑一遍

AB.c

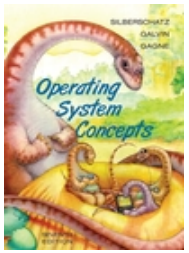


INT进入内核

```
main(){
      mov __NR_fork, %eax
      int 0x80
100:  mov %eax, res
      cmpl res,0
```

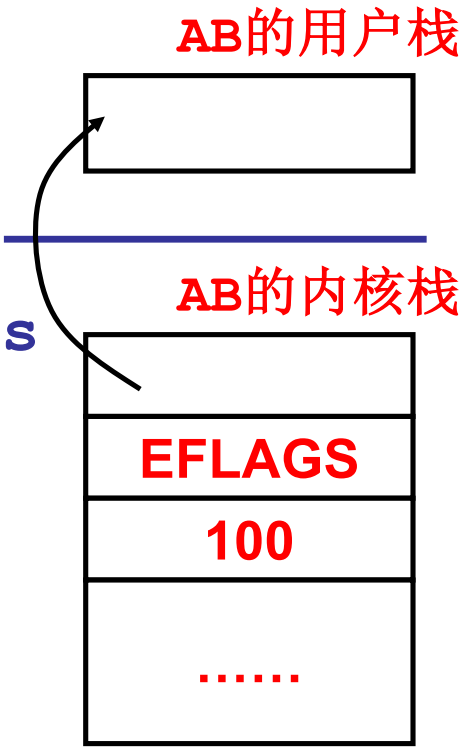



```
(2)set_system_gate(0x80,&system_
call);
(3)system_call:
      call sys_call_table(,%eax,4)
```



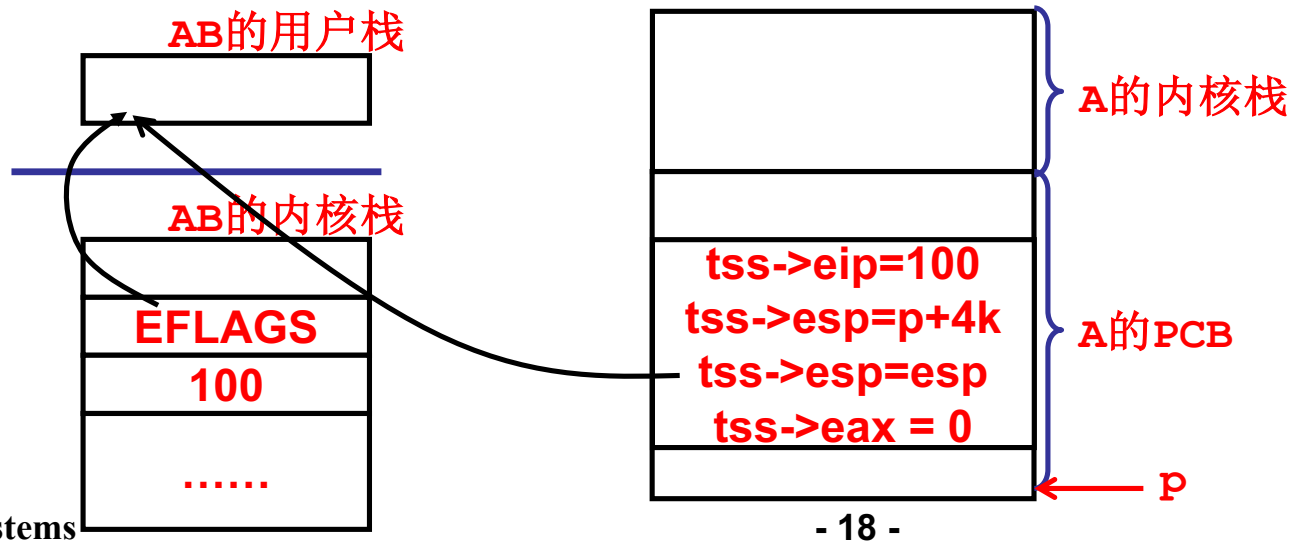
开始sys_fork

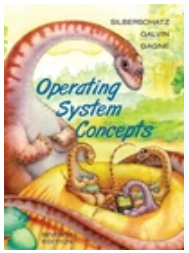
```
sys_fork:
  pushl ...
  call copy_process
  ret
```



开始copy_process

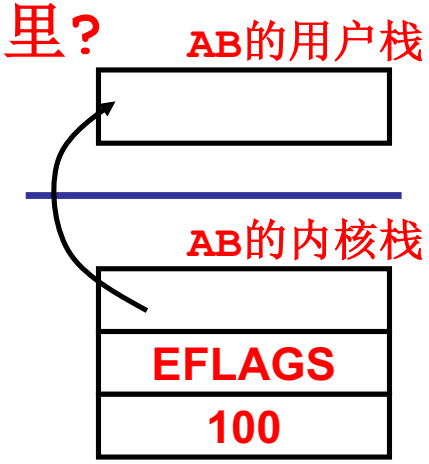
```
                                    =100
copy_process(...long eip, ...){
  p = (PCB *)get_free_page();
  p -> tss.esp0 = p+4k;
  p -> tss.esp = esp;
  p -> tss.eax=0; p -> tss.eip = eip;...}
```





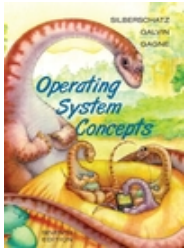
开始返回...

```
copy_process(){... }//ret到哪里?
sys_fork:
   ... call copy_process ...
   ret //到哪里?
system_call:
   ...call sys_call_table(,%eax,4)
   cmpl $0,state(current)
   jne reschedule
   iret //到哪里?
main: int 0x80
   100: mov %eax, res cmpl res,0
```




```
main()
{
  if(!fork()){while(1)printf("A");}
  if(!fork()){while(1)printf("B");}
  wait();
}
```

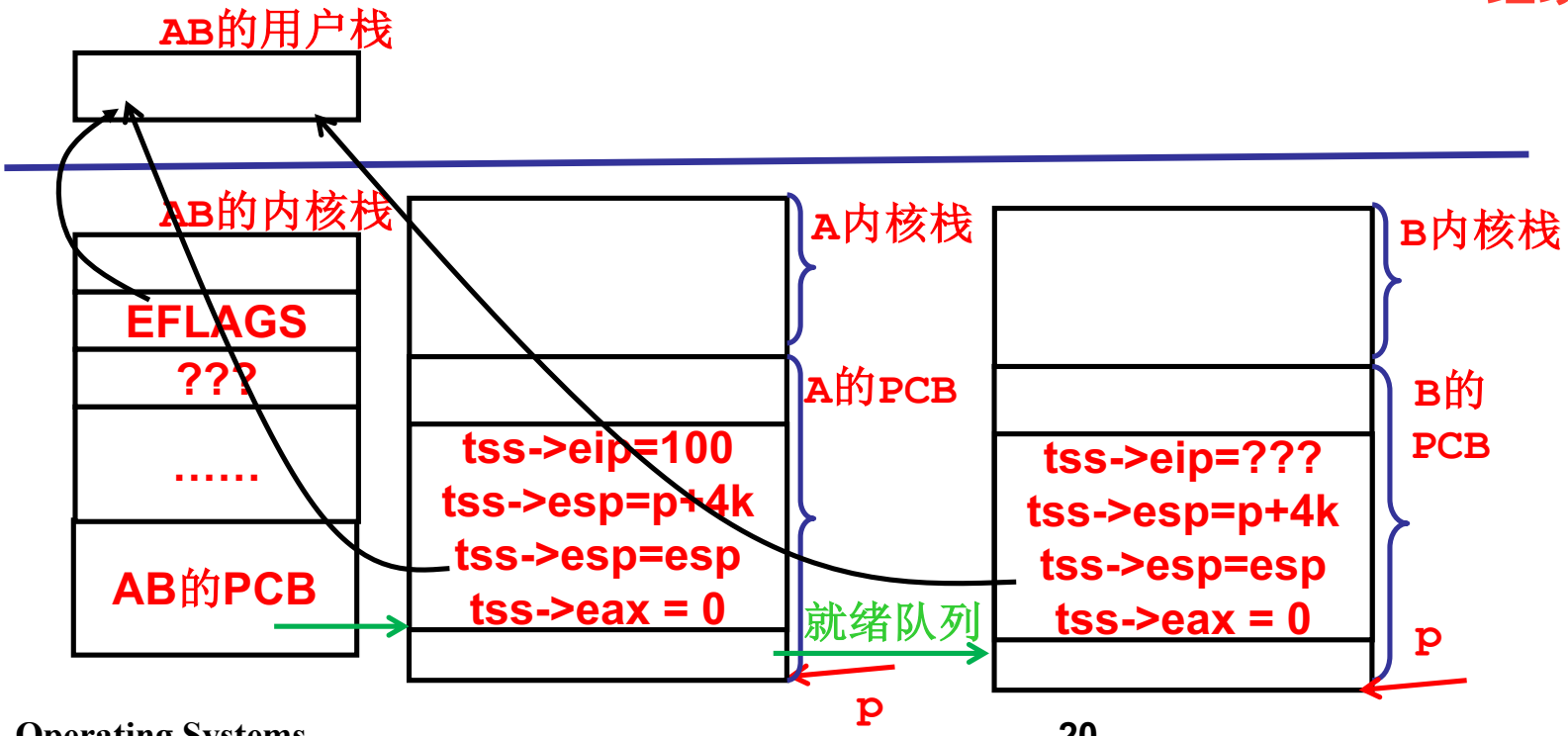
中断返回的时候
判断是否需要
进行schedule

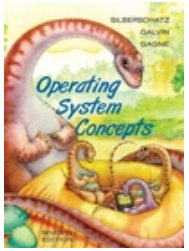


main继续执行，现在我们有了什么？

```
main(){
    又一个fork;再一次int 0x80;
    在内核中再产生一个PCB和内核栈...
```

继续将B进程创建出来

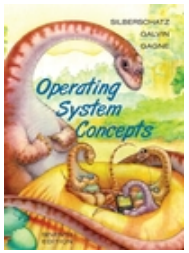




main继续，到了哪里？

```
main(){
    ...   wait(); //又是mov __NR_wait
                        int 0x80
system_call:
    call sys_waitpid
sys_waitpid() //exit.c中
    current->state=TASK_INTERRUPTIBLE;
    schedule();
```

将当期进程的状态变为阻塞态，
并调用调度函数

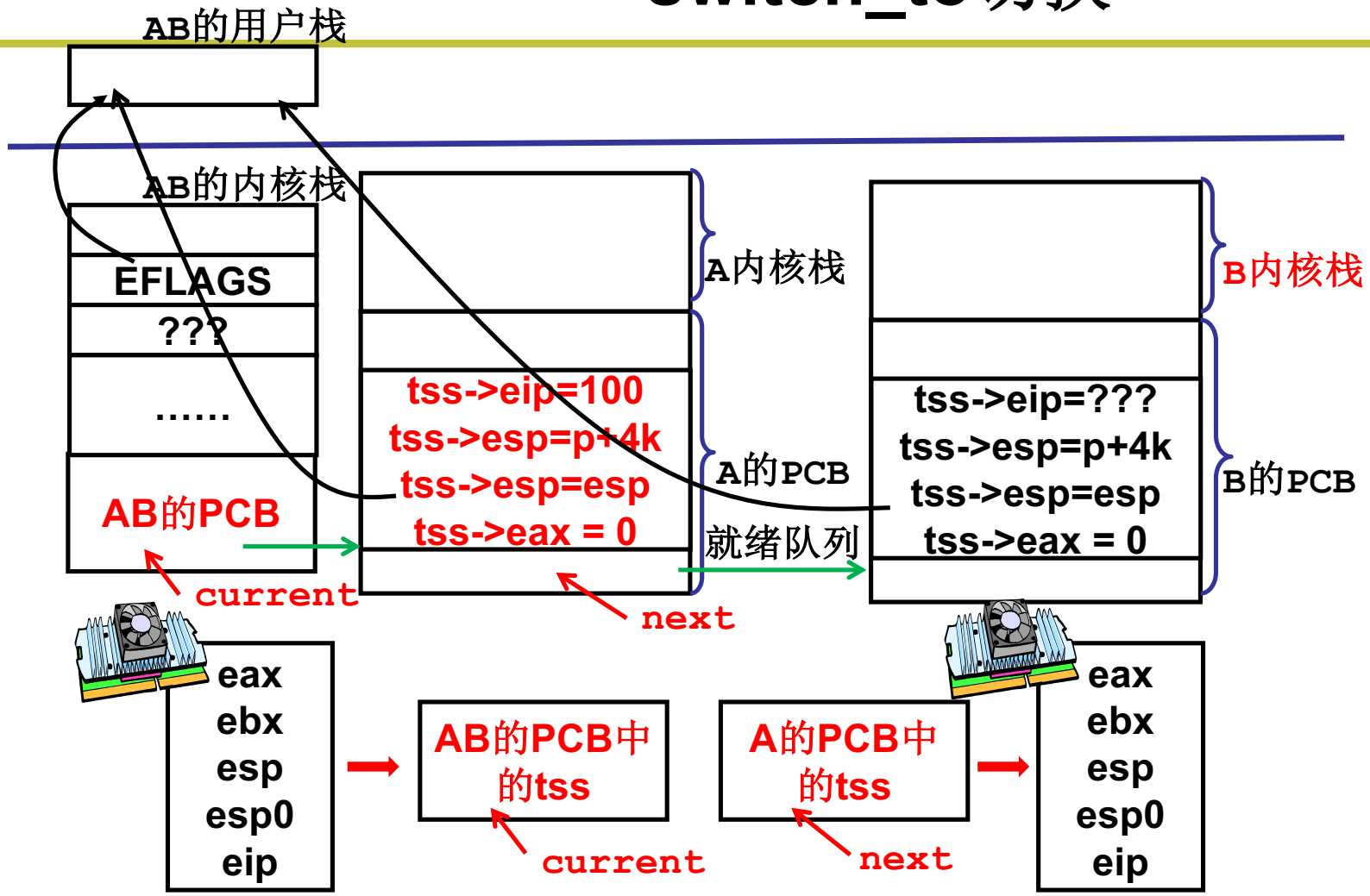


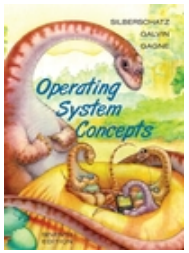
schedule

```
schedule(){
 if ((*p)->state == TASK_RUNNING &&
  (*p)->counter > c)
    c = (*p)->counter, next = i;
    ...
  switch_to(next);}
```

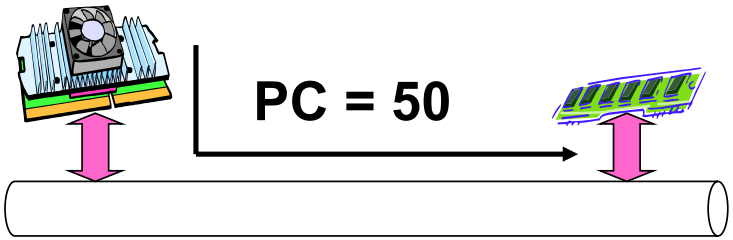
调度算法，然后切换过去!

switch_to切换


AB的用户栈
AB的内核栈
EFLAGS
???
.....
AB的PCB
current
tss->eip=100
tss->esp=p+4k
tss->esp=esp
tss->eax = 0
next
A内核栈
A的PCB
就绪队列
tss->eip=???
tss->esp=p+4k
tss->esp=esp
tss->eax = 0
B内核栈
B的PCB
eax
ebx
esp
esp0
eip
AB的PCB中的tss
current
A的PCB中的tss
next
eax
ebx
esp
esp0
eip


Operating System Concepts

接下来会怎么样?



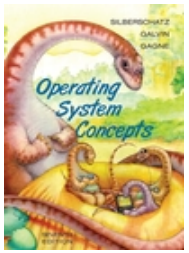
eip=100 eax=0 esp=...

这就是线程，代码始终不曾变过，一直在内存中...

fork的代码也不曾变过，所以实验8可以通过fork“伪造”

进程A开始不断的打A...

```
main(){
    mov __NR_fork, %eax
    int 0x80
100:mov %eax, res
    cmpl res,0
    jne 208
200: printf("A")
    jmp 200
208: ...
304: wait()
}
```



我们的目标是什么？

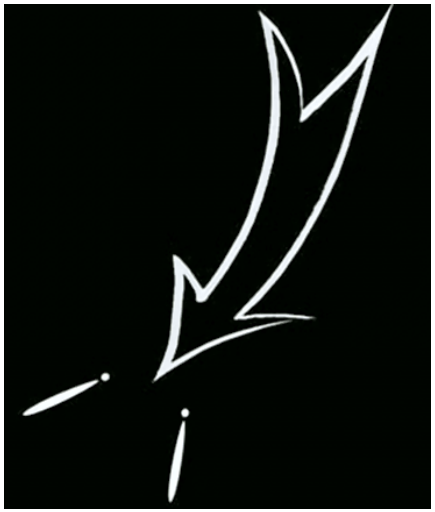
交替的打出A和B . . .

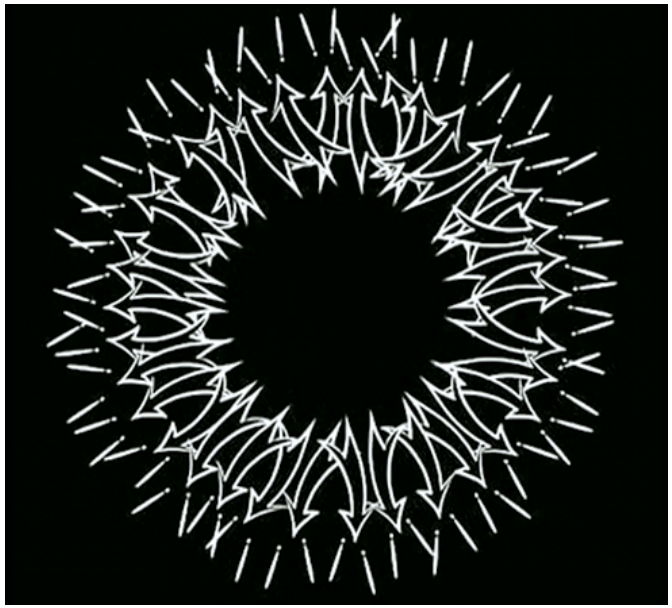
怎么可能打出B?//让B进程执行

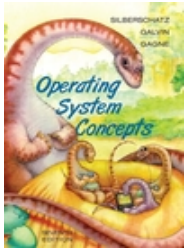
需要时钟中断来进行干预!
不能只依赖程序自己让出CPU

接下来会发生什么?把自己变成计算机想一想 . . .

中断 . . .什么中断?







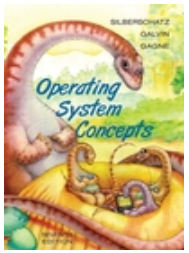
时钟中断

```
void sched_init(void)  //在sched.c中
{ set_intr_gate(0x20,&timer_interrupt);
```

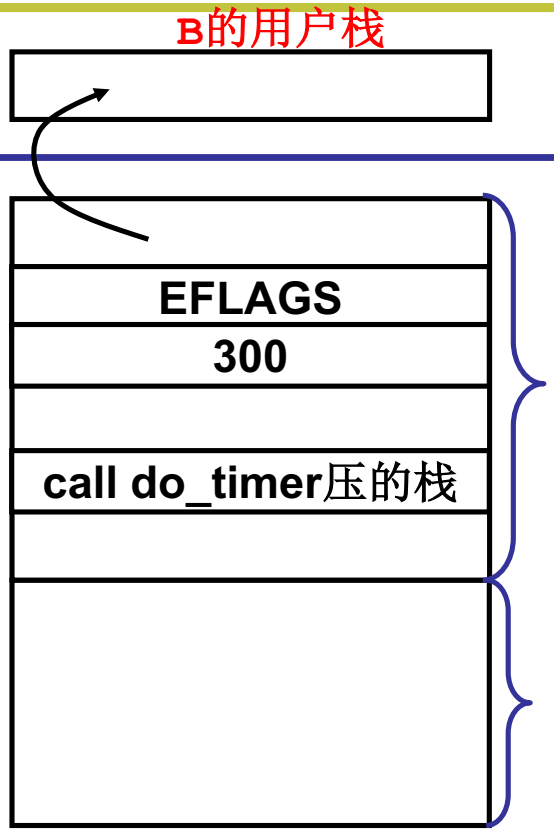
```
void _timer_interrupt:
   ...
   call do_timer
```

```
void do_timer(...)
{   if((--current->counter>0) return;
    current->counter=0;
    schedule(); }
```

每一次时钟中断，
就将counter--，
当counter变为零的时候，
当前进程的时间片用完，
进行调度！

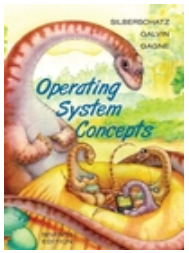


有那么一次时钟中断

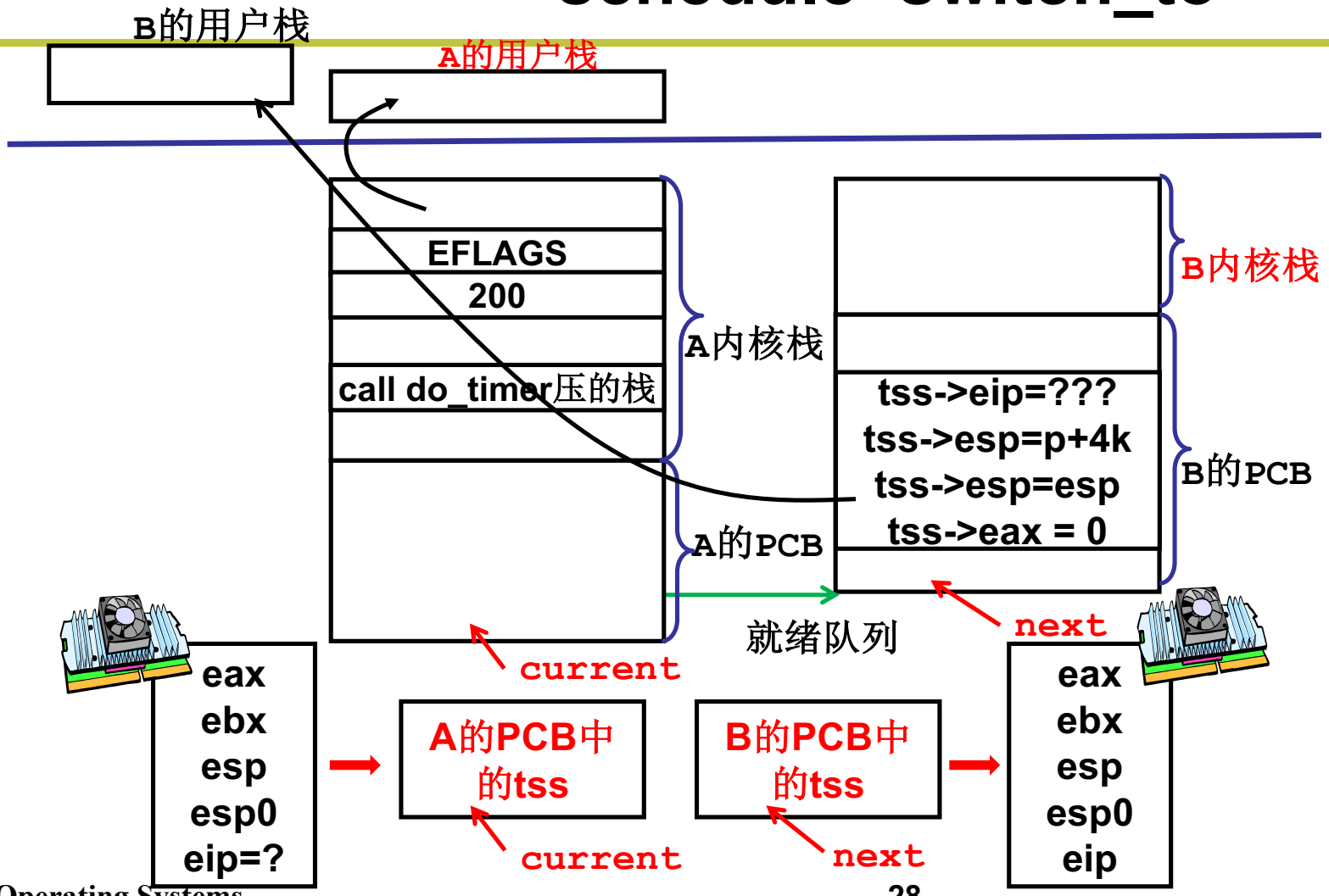


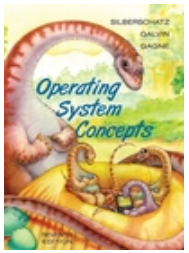

```
main(){
    mov __NR_fork, %eax
    int 0x80
100:mov %eax, res
    cmpl res,0
    jne 208
200: printf("A")
    jmp 200
208: ... 300:printf("B")
             jmp 300
308: wait()
}
```

```
do_timer(...) {
    current->counter=0; schedule(); }
```

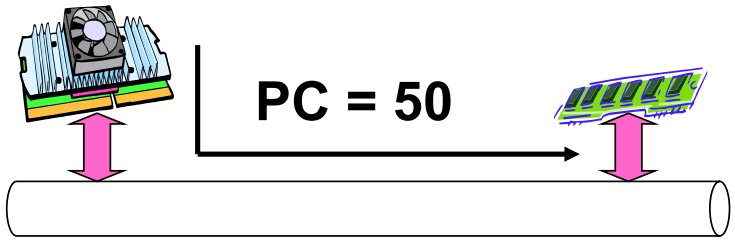


schedule+switch_to





接下来会怎么样?

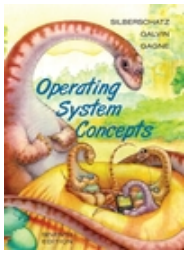


eip=??? eax=0 esp=...

代码始终不曾变过，
如同对操作系统的热爱...

进程B开始不断的打B...

```
main(){
    mov __NR_fork, %eax
    int 0x80
100:mov %eax, res
    cmpl res,0
    jne 304
200: printf("A")
    jmp 200
208: ...300:printf("B")
            jmp 300
304: wait()
}
```



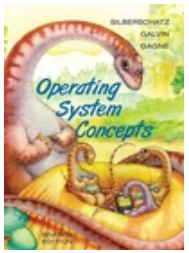
我们的目标达到了吗?

交替的打出A和B...

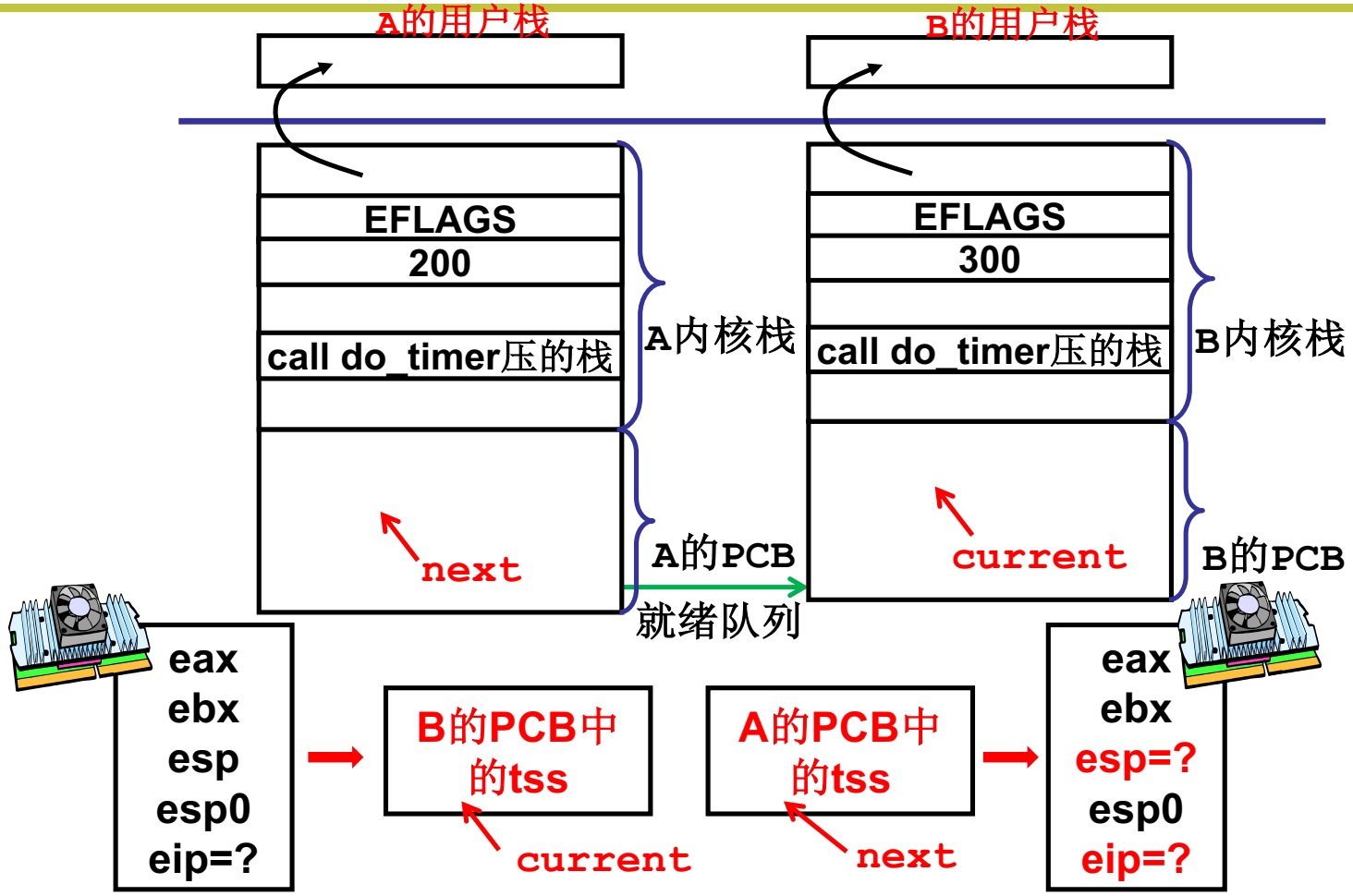
已经打出了B，完事了吗？ 何为交替？

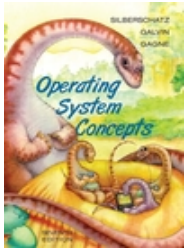
接下来会发生什么?把自己变成计算机想一想...

中断,仍然是中断...什么中断?

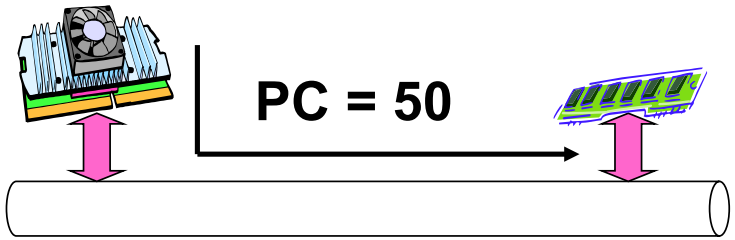


又有那么一次时钟中断, 再一次schedule+switch_to





接下来会怎么样?




```
void do_timer(...)
{   ...
    schedule(); }
```

```
void _timer_interrupt:
   ...
   call do_timer
   ...这后面是什么?
```