中国科学院大学
University of Chinese Academy of Sciences

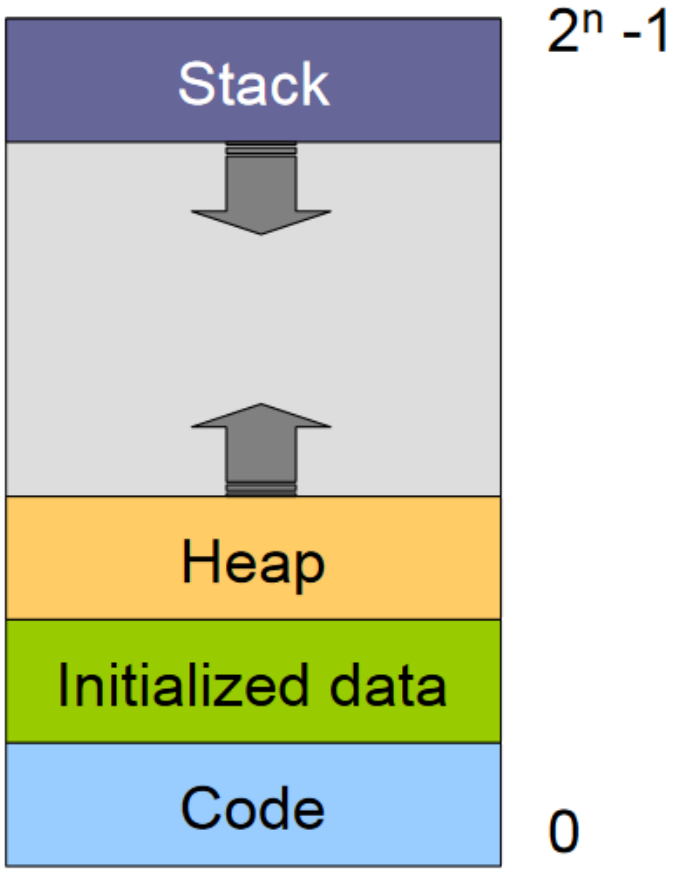
操作系统习题课

中国科学院大学计算机与控制学院
中国科学院计算技术研究所
2018-11

作业 1

- 请在一个POSIX兼容的环境下(Unix,Linux,Windows CMD,MacOS)等下面编译执行附件的小程序，并根据结果分析每个变量所属的段(section)。

课件



```
#include <stdio.h>
#include <stdlib.h>

char *myname=“Bao  Yungang”;          数据段  .data
char gdata[128];                      数据段（未初始化）  .bss
char bdata[16] = {1,2,3,4};           数据段 .data
main() {
        char * ldata[16];             栈
        char * ddata;                 栈

        ddata = malloc(16);           堆
        printf("gdata: %llX\nbdata:%llX\nldata:%llx\nddata:%llx\n",
                gdata,bdata,ldata,ddata);
        free(ddata);
}
```

注意点

- 需要注重理论和实际相结合，充分利用工具
- 注意区分指针变量自身的地址和分配的空间

作业 2

- 作业指导 Linux 下常见的3种系统调用方法包括有：

1. 通过glibc提供的库函数

2. 使用syscall函数直接调用相应的系统调用

3. 通过 int 80指令陷入（32bit）或者通过syscall指令陷入（64bit）

- 请研究Linux(kernel>=2.6.24) gettimeofday这一系统调用的用法，并且选择上述3种系统调用方法中的2种来执行，记录其运行时间。
- 提示：请思考一次系统调用的时间开销的量级，对比结果，并尝试解释其中原因。

注意点

- 在计算机系统中，统计程序的运行时间，经常会由于各种复杂的原因引入误差。因此，在实验过程中，需要**多次测量来消除误差**!

- 以clock_gettime为例（功能同gettimeofday，精确到纳秒级），第一次和第二次运行的时间差异:

  583.00 ns

  412.00 ns

  30%的误差!

注意点

- gettimeofday已经被Linux做了优化，并不会进行上下文切换并进入内核态。因此，通过glibc调用要比syscall函数或者汇编形式调用gettimeofday快得多。
- VDSO
- RTFM

Syscall（单位：ns）	通过glibc提供的库函数	使用syscall函数直接调用	使用syscall/int指令陷入
gettimeofday	33.70	385.74	387.44

作业 3

第1题 以下代码为Linux/Unix创建进程的例子，其中

* fork克隆出一个进程
* exec覆盖掉当前进程

```
If ((pid = fork()) == 0) {
        /* child process */
        exec(“foo”);    /* does not return */
else
        /* parent */
        wait(pid);       /* wait for child to die */
```

1. 在xv6中，PCB信息是如何表示的？是用什么数据结构？存放在哪个文件中？

2. 上述代码中，修改了哪些PCB中的信息？

注意点

- 重点掌握 fork、exec、wait
- 结合源码分析，而不是泛泛而谈

作业 3

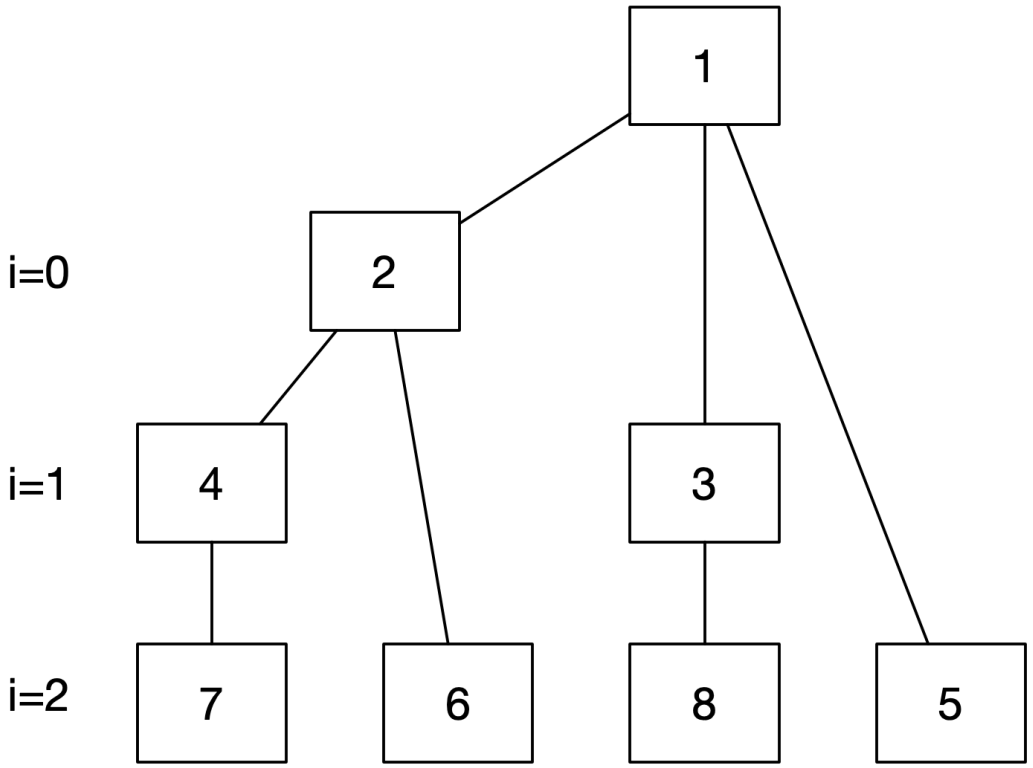
第2题

```
for  (i=0;  i<3;  i++) {
        pid = fork();
        if (pid == 0) { /* child process */
            fprintf(stdout, “i=%d,  pid=%d,  parent  pid=%d\n”,i,
            getpid() ,getppid());
        }
}
wait(NULL);
exit(0);
```

请问该程序最终一共生成几个进程？假设当前进程PID为1，生成的进程PID依次加1，请将生成进程关系图画出来。

注意点

- 答案：包括原进程共8个
  - fork炸弹
- 进程关系图
  - 最好画成不对称树



作业 4

- 第1题
- 写一个2线程的程序，首先生成一个从1到1000万的整数数组，然后用两个线程分别计算数组奇数部分和偶数部分的和，并打印出总的和。分别在单核和双核系统上运行该程序，计算加速比。（采用pthread API）

注意点

- 多次实验消除误差
- gettimeofday的卡点
- 2线程？3线程？
- 加速比太低甚至小于1，可能原因
  - 虚拟机性能问题，或多核调度算法问题
  - 程序本身执行时间太短
    - posix线程本身的overhead

作业 4

- 第2题
- Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and X. In what order should they be run to minimize average response time? (Your answer will depend on X.)
- 答案：9 6 3 5 X 从小到大排序

作业 4

第3题

Five batch jobs A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process **turn-around time**. Ignore process switching overhead.

(a) Round robin.

(b) Priority scheduling

(c) First-come, first-served (run in order 10, 6, 2, 4, 8).

(d) Shortest job first.

For (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b) through (d) assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

- (a) 10 6 2 4 8 round robin
- 答案：假设题目时间片很短，所以各任务完成时间之间间隔为C10 D8 B6 E4 A2，完成时间：C10 D18 B24 E28 A30,平均22

- (b) 10 6 2 4 8, priority 3 5 2 1 4 (5 highest)
- 答案：调度顺序 6 8 10 2 4，完成时间分别为6 14 24 26 30，平均20

- (c) FCFS 10 6 2 4 8
- 答案：完成时间分别为10 16 18 22 30，平均19.2

- (d) Shortest job first 10 6 2 4 8
- 答案：完成时间分别为2 6 12 20 30，平均14

作业 4

- 第4题
- A real-time system needs to handle two voice calls that each run every 5 msec and consume 1 msec of CPU time per burst, plus one video at 24 frames/sec, with each frame requiring 20 msec of CPU time. Is this system schedulable?
- 答案：计算任务1000ms内消耗的总时间，为(2*1*(1000/5)+24*20)=880ms，因此是可调度的

作业 5

写一个两线程程序,两线程同时向一个数组分别写入1000万以内的奇数和偶数，过程中共用一个偏移量。写完后打印出数组相邻两个数的最大绝对差值。

请分别按下列方法完成一个不会丢失数据的程序:

1) 请用Peterson算法实现上述功能。

2) 学习了解 pthread_mutex_lock/unlock()函数的功能，并实现上述程序功能。

3) 学习了解 atomic_add()(_sync_fetch_and_add（）for gcc 4.1+) 函数，实现上述程序功能。

要求方法1和方法2 每次进入临界区之后，执行一百次操作后离开临界区。

请找一个双核系统测试，分别列出三种方式的执行时间。

注意点

- 多次实验消除误差　（重要的事情说三遍）
- 需要验证数据是否丢失
- 数组相邻两个数的最大差值可以反映线程切换频度

注意点

- 进入临界区后执行一百次操作后离开临界区：

```
for (int i = 0; i < N; ) {
    pthread_mutex_lock(&mutex);
    for (int j = 0; j < 100; j++, i+=2) {
        num[shindex] = i;
        shindex++;
    }
    pthread_mutex_unlock(&mutex);
}
```

注意点

- 原子指令实现
- 正确版本
- num[__sync_fetch_and_add(&shindex, 1)] = i;

- 错误版本
- num[shindex] = i;
- __sync_fetch_and_add(&shindex, 1);

错误版本

- num[shindex] = i;
- __sync_fetch_and_add(&shindex, 1);

```
000000000000085a <odd>:
 85a:   55                      push   %rbp
 85b:   48 89 e5                mov    %rsp,%rbp
 85e:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
 862:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
 869:   eb 2c                   jmp    897 <odd+0x3d>
 86b:   8b 05 cf 61 82 02       mov    0x28261cf(%rip),%eax        # 2826a40 <shindex>
 871:   8b 55 fc                mov    -0x4(%rbp),%edx
 874:   8d 4a 01                lea    0x1(%rdx),%ecx
 877:   48 98                   cltq
 879:   48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
 880:   00
 881:   48 8d 05 b8 07 20 00    lea    0x2007b8(%rip),%rax        # 201040 <num>
 888:   89 0c 02                mov    %ecx,(%rdx,%rax,1)
 88b:   f0 83 05 ad 61 82 02    lock addl $0x1,0x28261ad(%rip)        # 2826a40 <shindex>
 892:   01
 893:   83 45 fc 02             addl   $0x2,-0x4(%rbp)
 897:   81 7d fc 7f 96 98 00    cmpl   $0x98967f,-0x4(%rbp)
 89e:   7e cb                   jle    86b <odd+0x11>
 8a0:   90                      nop
 8a1:   5d                      pop    %rbp
 8a2:   c3                      retq
```

正确版本

- num[__sync_fetch_and_add(&shindex, 1)] = i;

```
000000000000085a <odd>:
 85a:   55                      push   %rbp
 85b:   48 89 e5                mov    %rsp,%rbp
 85e:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
 862:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
 869:   eb 2e                   jmp    899 <odd+0x3f>
 86b:   b8 01 00 00 00          mov    $0x1,%eax
 870:   f0 0f c1 05 c8 61 82    lock xadd %eax,0x28261c8(%rip)        # 2826a40 <shindex>
 877:   02
 878:   89 c2                   mov    %eax,%edx
 87a:   8b 45 fc                mov    -0x4(%rbp),%eax
 87d:   8d 48 01                lea    0x1(%rax),%ecx
 880:   48 63 c2                movslq %edx,%rax
 883:   48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
 88a:   00
 88b:   48 8d 05 ae 07 20 00    lea    0x2007ae(%rip),%rax        # 201040 <num>
 892:   89 0c 02                mov    %ecx,(%rdx,%rax,1)
 895:   83 45 fc 02             addl   $0x2,-0x4(%rbp)
 899:   81 7d fc 7f 96 98 00    cmpl   $0x98967f,-0x4(%rbp)
 8a0:   7e c9                   jle    86b <odd+0x11>
 8a2:   90                      nop
 8a3:   5d                      pop    %rbp
 8a4:   c3                      retq
```

作业 6

第1题 A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	Maximum	Available
Process A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Process B	2 0 1 1 0	2 2 2 1 0	
Process C	1 1 0 1 0	2 1 3 1 0	
Process D	1 1 1 1 0	1 1 2 2 1	

What is the smallest value of x for which this is a safe state?

- x=0，没有进程可以申请最大资源，死锁
- x=1，仅D可以继续运行，D结束后，仍然死锁
- x=2，可以按照DCBA的顺序运行
  - 题目有瑕疵，A的最后一个资源无法满足

作业 6

- 第2题 The processes, A and B, each need three records, 1, 2, and 3, in a database. If A asks for them in the order 1, 2, 3, and B asks for them in the same order, deadlock is not possible. However, if B asks for them in the order 3, 2, 1, then deadlock is possible. With three resources, there are 3! or six possible combinations each process can request the resources. What fraction of all the combinations is guaranteed to be deadlock free?

- 假设A申请顺序为1 2 3
- B申请顺序，1 2 3和1 3 2不会发生死锁，其他顺序不行
- 因此比例为1/3

作业 7

第1题 设有两个优先级相同的进程P1，P2如下。令信号量S1，S2的初值为0，已知z=2，试问P1，P2并发运行结束后x=？y=？z=？

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁
- 确定性的步骤：

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

x = 2
y = 3
z = 2

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 2
y = 3
z = 2

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 2
y = 3
z = 4

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

①继续执行 P1

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 5
y = 3
z = 4

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

①继续执行 P1

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 5
y = 3
z = 4

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

①继续执行 P1

不确定点 2:
V(S2)唤醒被阻塞的进程P2之后，继续执行P2，还是切换到P1

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 5

y = 3

z = 9

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

①继续执行 P1

不确定点 2:
V(S2)唤醒被阻塞的进程P2之后，继续执行P2，还是切换到P1

①继续执行 P2

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 5

y = 12

z = 9

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

①继续执行 P1

不确定点 2:
V(S2)唤醒被阻塞的进程P2之后，继续执行P2，还是切换到P1

①继续执行 P2

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 5
y = 3
z = 4

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

①继续执行 P1

不确定点 2:
V(S2)唤醒被阻塞的进程P2之后，继续执行P2，还是切换到P1

①~~继续执行 P2~~
②切换到 P1

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 5

y = 7

z = 4

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

①继续执行 P1

不确定点 2:
V(S2)唤醒被阻塞的进程P2之后，继续执行P2，还是切换到P1

①~~继续执行 P2~~
②切换到 P1

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 5

y = 7

z = 9

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

①继续执行 P1

不确定点 2:
V(S2)唤醒被阻塞的进程P2之后，继续执行P2，还是切换到P1

①~~继续执行 P2~~
②切换到 P1

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 2
y = 3
z = 2

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

①继续执行~~P1~~

②切换到 P2

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 5
y = 3
z = 7

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2
① 继续执行~~P1~~

② 切换到 P2

注意点

- 信号量的初始值决定其如何作用!
  - 本题，S1、S2的初值为0，并不是互斥锁

x = 5

y = 7

z = 4

进程P1	进程P2
y:=1;	x:=1;
y:=y+2;	x:=x+1;
V(S1);	P(S1);
z:=y+1;	x:=x+y;
P(S2);	V(S2);
y:=z+y;	z:=x+z;

不确定点 1:
V(S1)唤醒被阻塞的进程P2之后，继续执行P1，还是切换到P2

① 继续执行~~P1~~

② 切换到 P2

作业 7

第2题 银行有n个柜员,每个顾客进入银行后先取一个号,并且等着叫号,当一个柜员空闲后,就叫下一个号.

请使用PV操作分别实现:

顾客取号操作 Customer_Service

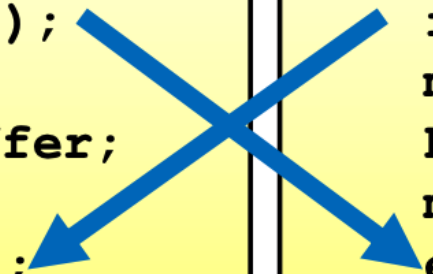
柜员服务操作 Teller_Service

讲义 用信号量实现生产者-消费者问题

```
Class BoundedBuffer {
    mutex = new Semaphore(1);
    fullBuffers = new Semaphore(0);
    emptyBuffers = new Semaphore(n);
}
```

```
BoundedBuffer::Deposit(c) {
    emptyBuffers->P();
    mutex->P();
    Add c to the buffer;
    mutex->V();
    fullBuffers->V();
}
```

```
BoundedBuffer::Remove(c) {
    fullBuffers->P();
    mutex->P();
    Remove c from buffer;
    mutex->V();
    emptyBuffers->V();
}
```



注意点

- 信号量的初始值决定其如何作用!
- 取号:
  - count ++， 访问共享变量，需要临界区

```
Mutex = 1;

getNumber();  // 同步控制了吗?
P(mutex);
// 进入等待队列
V(mutex);
```

注意点

- 信号量的初始值决定其如何作用!
- 取号:
  - count ++， 访问共享变量，需要临界区
  - 取号不应该被柜员状态阻塞

```
Mutex = 1;  teller_empty = n;

P(teller_empty)  // 柜员都在工作，就不能取号了?
P(mutex);
// 取号并进入等待队列
V(mutex);
```

注意点

- 信号量的初始值决定其如何作用!
- 取号:
  - count ++， 访问共享变量，需要临界区
  - 取号不应该被柜员状态阻塞
- 信号量不止能够作为互斥锁，还可以作为条件变量

```
Customer:
P(mutex)
// 取号 加入 等待队列
V(mutex)
```

```
Teller:
P(mutex)
// 叫号
V(mutex)
// 服务
```

注意点

- 信号量的初始值决定其如何作用!
- 取号:
  - count ++， 访问共享变量，需要临界区
  - 取号不应该被柜员状态阻塞
- 信号量不止能够作为互斥锁，还可以作为条件变量
- 临界区尽可能简短

```
Teller:
P(customer)
P(mutex)
// 叫号
// 服务  其他柜员还叫号吗?
V(mutex)
```

作业7

第3题 多个线程的规约(Reduce)操作是把每个线程的结果按照某种运算（符合交换律和结合律）两两合并直到得到最终结果的过程。

试设计管程monitor实现一个8线程规约的过程，随机初始化16个整数，每个线程通过调用 monitor.getTask 获得2个数，相加后，返回一个数monitor.putResult ，然后再getTask()直到全部完成退出，最后打印归约过程和结果。

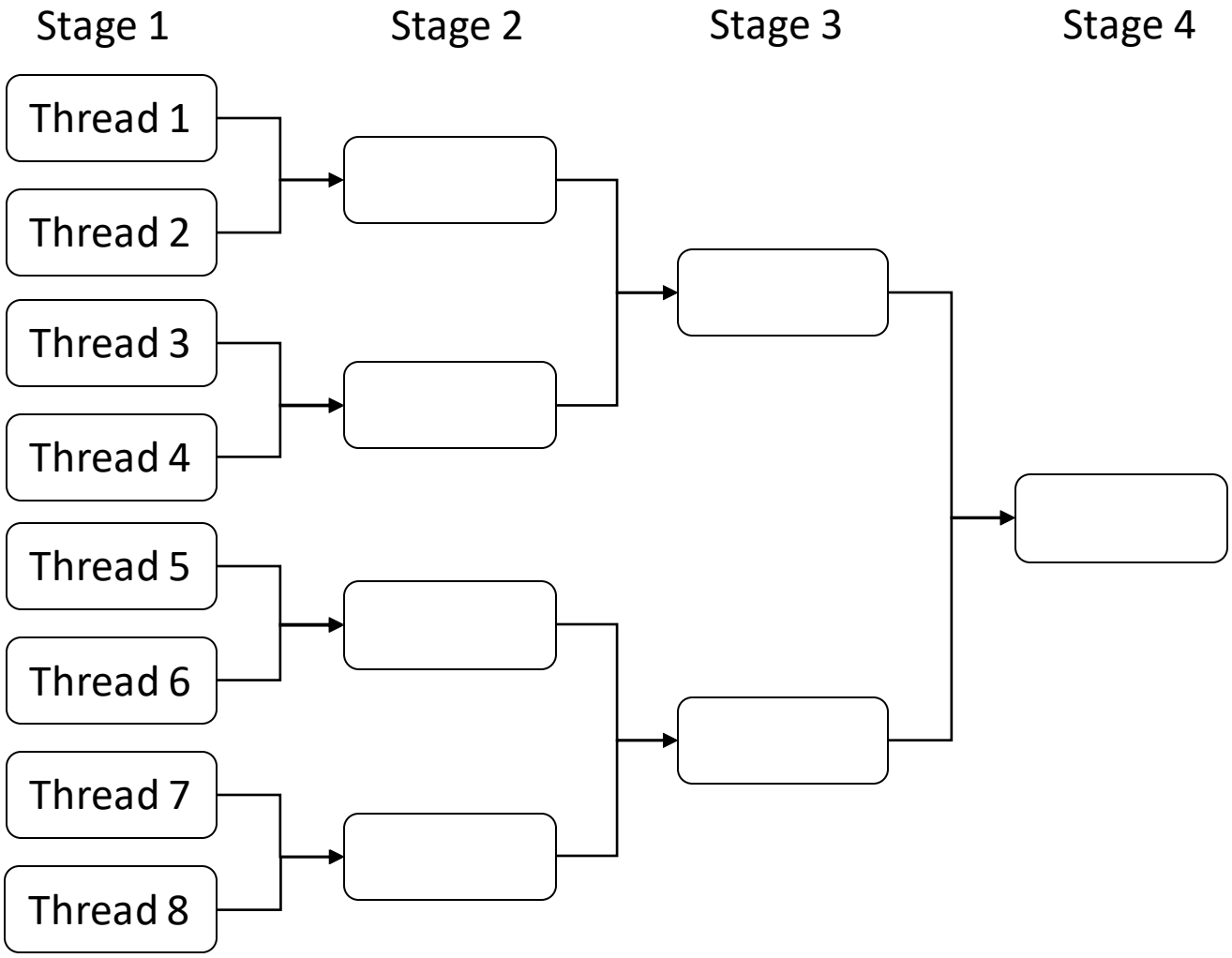
要求：为了模拟不均衡性，每个加法操作要加上随机的时间扰动，变动区间1~10ms。

提示：

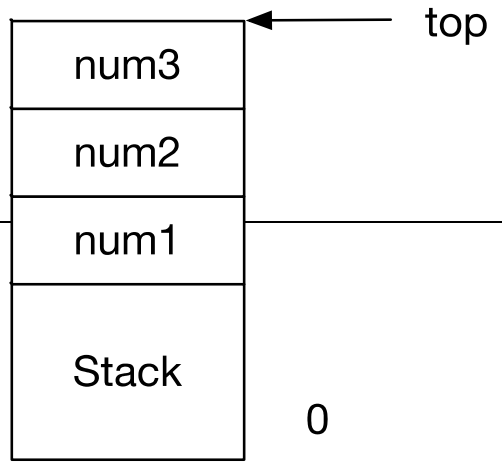
使用pthread_系列的 cond_wait, cond_signal, mutex 实现管程

使用rand()函数产生随机数，和随机执行时间。

Reduce

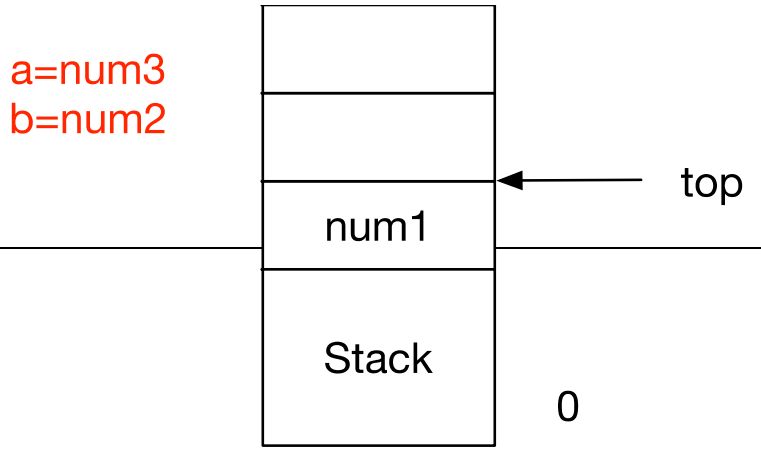

Stage 1
Stage 2
Stage 3
Stage 4
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8

注意点



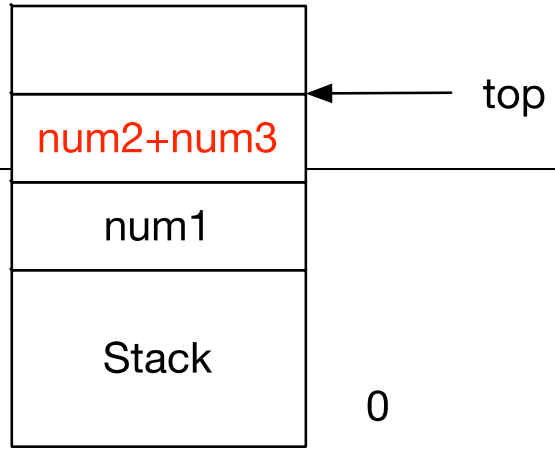

```
void thread_func() {
    int a, b, res;
    while(top > 0) {
        gettask(&a, &b);
        res = a+b;
        usleep(rand_time());
        putresult(res);
        printf("thread %d: %d + %d = %d\n", tid, a, b, res);
    }
}
```

注意点




```
void thread_func() {
    int a, b, res;
    while(top > 0) {
        gettask(&a, &b);
        res = a+b;
        usleep(rand_time());
        putresult(res);
        printf("thread %d: %d + %d = %d\n", tid, a, b, res);
    }
}
```

注意点




```
void thread_func() {
    int a, b, res;
    while(top > 0) {
        gettask(&a, &b);
        res = a+b;
        usleep(rand_time());
        putresult(res);
        printf("thread %d: %d + %d = %d\n", tid, a, b, res);
    }
}
```

期中考试注意事项

- 考试时间11.10周六下午18:00~19:40
  - 一班：阶二3
  - 二班：阶二4
- 考试题型：
  - 20道选择题，每题3分
  - 4道大题，每题10分
  - 有写代码题，有XV6代码阅读的题