

第三讲 多线程程 序设计

课程网站: <http://spoc.buaa.edu.cn>

主讲教师: 李云春
办公室: 新主楼G217
时间: 周四上午1-2节 (1-13周)
Email: lych@buaa.edu.cn
Phone: 82339268

Autumn 2023

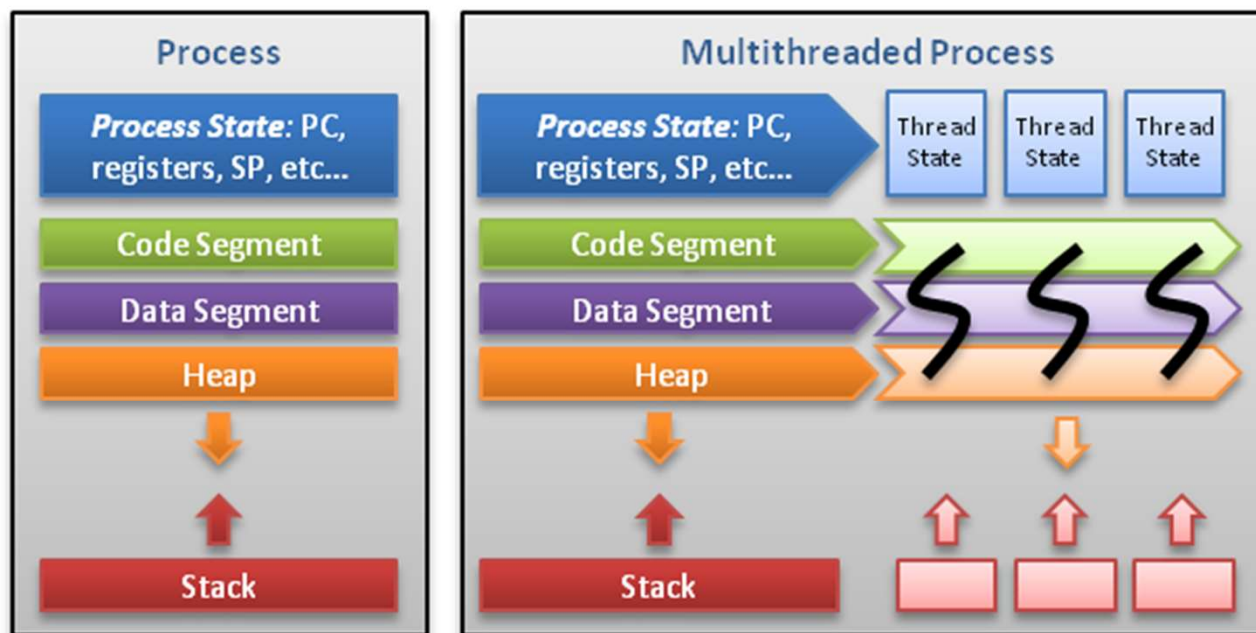
本章内容

- 3.1 **线程基础**
- 3.2 **线程的基本操作**
- 3.3 **多线程的共享变量**
- 3.4 **线程同步机制**
- 3.5 **多线程信号处理**
- 3.6 **并发常见问题**
- 3.7 **线程程序的性能分析**

3.1 线程基础

一. 线程定义

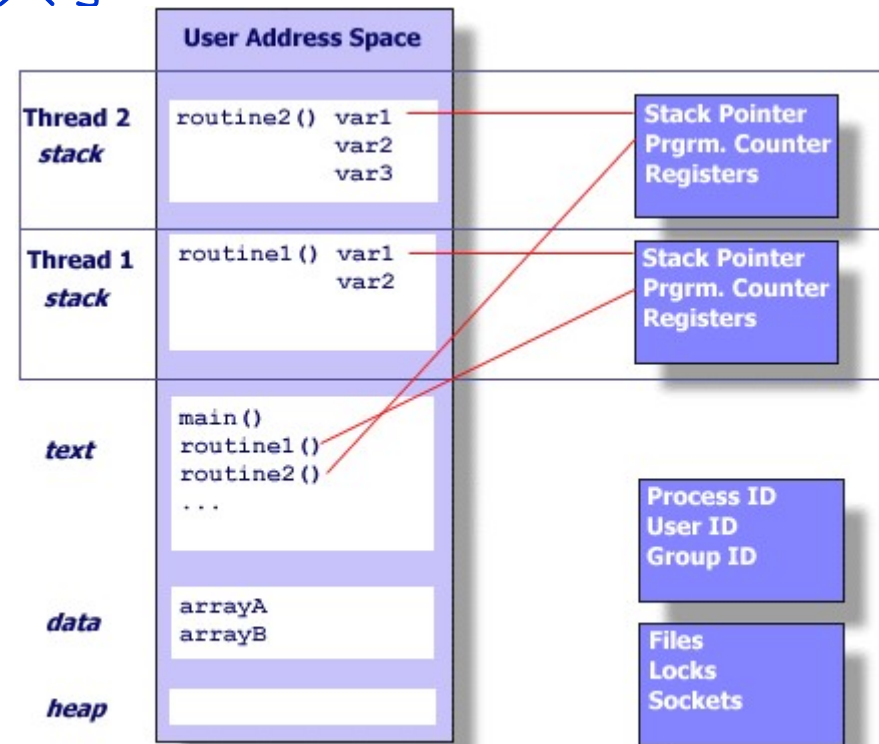
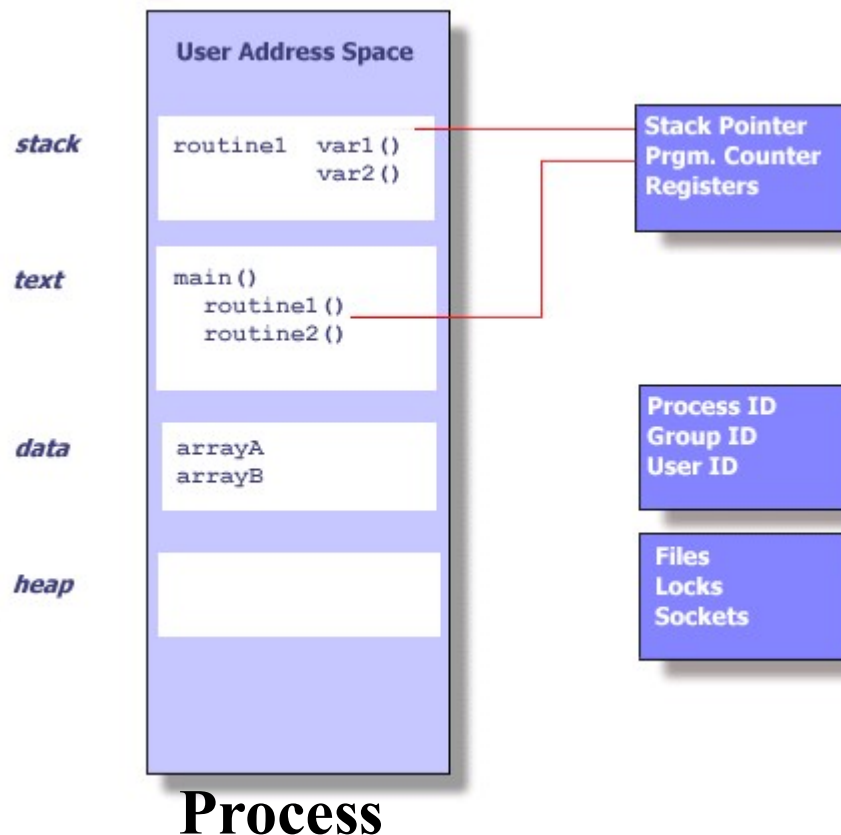
线程可认为是进程内部的执行流，一个进程内可包括多个线程，一个显著特点是线程间共享地址空间。



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

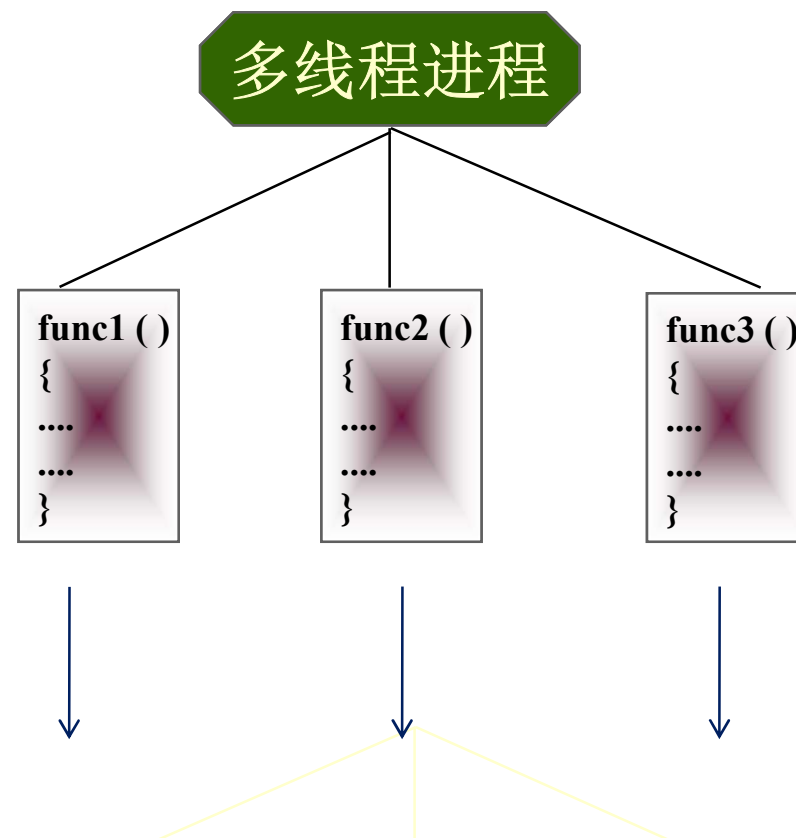
理解线程：进程内独立的执行流

- ✳ 进程开始执行时只有一个线程，称为主线程（main thread）
- ✳ 独立执行：线程间互相独立，与进程类似
- ✳ 共享地址空间：共享堆（指针）、数据段（静态变量、全局变量）、代码段
- ✳ 独立的栈：临时变量可间接共享



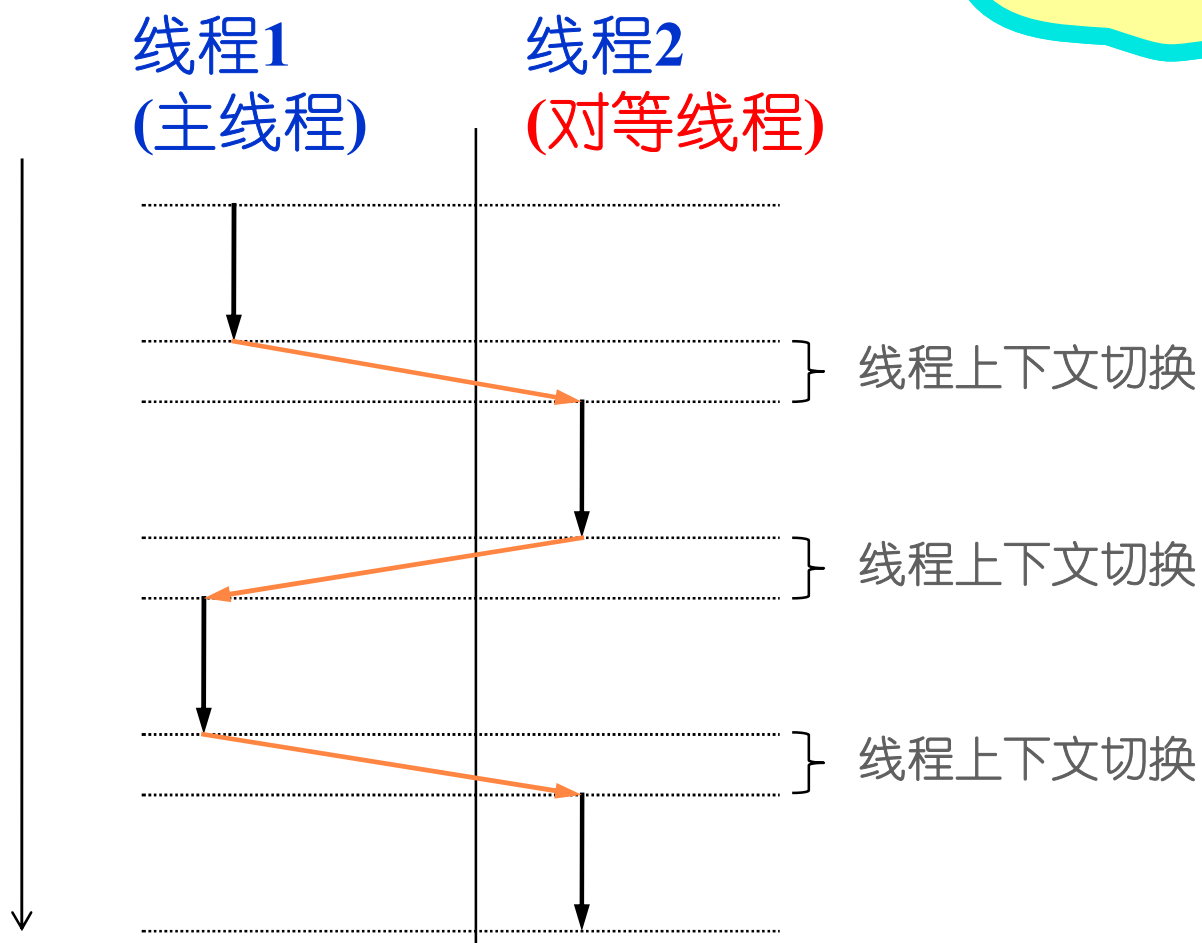
Threads Within a Process

理解线程：并行的函数



理解线程：并发线程执行

多线程间没有严格的父子关系



理解线程：轻量级进程

Linux线程也是进程，与父进程共享地址空间，称为轻量级进程

fork实现
(创建进程)

pthread_create实现
(创建线程)

```
int clone(int (*fn)(void *),  
void *child_stack, int flags, void  
*arg, ... pid_t *ptid, struct  
user_desc *tlsctid */);
```

clone系统调用可以指定子进程共享父进程的哪一部分进程上下文。
如果共享虚拟内存，则是线程

二. 线程优势与风险

50000次fork() 与pthread_create()的时间比较，单位秒

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.8 GHz Xeon 5660 (12cpus/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16cpus/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

二. 线程优势与风险

MPI和Pthreads数据传输比较

Platform	MPI Shared Memory Bandwidth(GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
AMD 2.4 GHz Opteron (8cpus/node)	1.2	5.3
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	4.1	16
IBM 1.5 GHz POWER4 (8cpus/node)	2.1	4
INTEL 2.4 GHz Xeon (2 cpus/node)	0.3	4.3
INTEL 1.4 GHz Itanium2 (4 cpus/node)	1.8	6.4

线程优势

1) 提高性能（相比进程）

- ✳ 线程创建快：与进程共享资源，因此，创建线程不需要复制整个地址空间
- ✳ 上下文切换快：从同一个进程内的一个线程切换到另一个线程时需要载入的信息比进程少

2) 便捷的数据共享（相比进程）

- ✳ 不必通过内核就可以共享和传递数据。线程间通信比进程间通信高效、方便

使用线程

- 1) 工作可以被多个任务同时执行，或者数据可以同时被多个任务操作
- 2) 阻塞与潜在的长时间I/O等待
- 3) 在某些地方使用很多CPU循环而其他地方没有
- 4) 对异步事件必须响应
- 5) 一些工作比其他的重要（优先级中断）

线程风险

- 1) 增加程序复杂性
- 2) 难于调试
竞态条件、死锁.....

3.2 线程的基本操作

一. 线程创建

创建线程

函数指针

```
#include <pthread.h>
```

```
typedef void * (start_routine)(void *);
```

```
int pthread_create(pthread_t * tid, pthread_attr_t * attr,  
                  start_routine* f, void * arg);
```

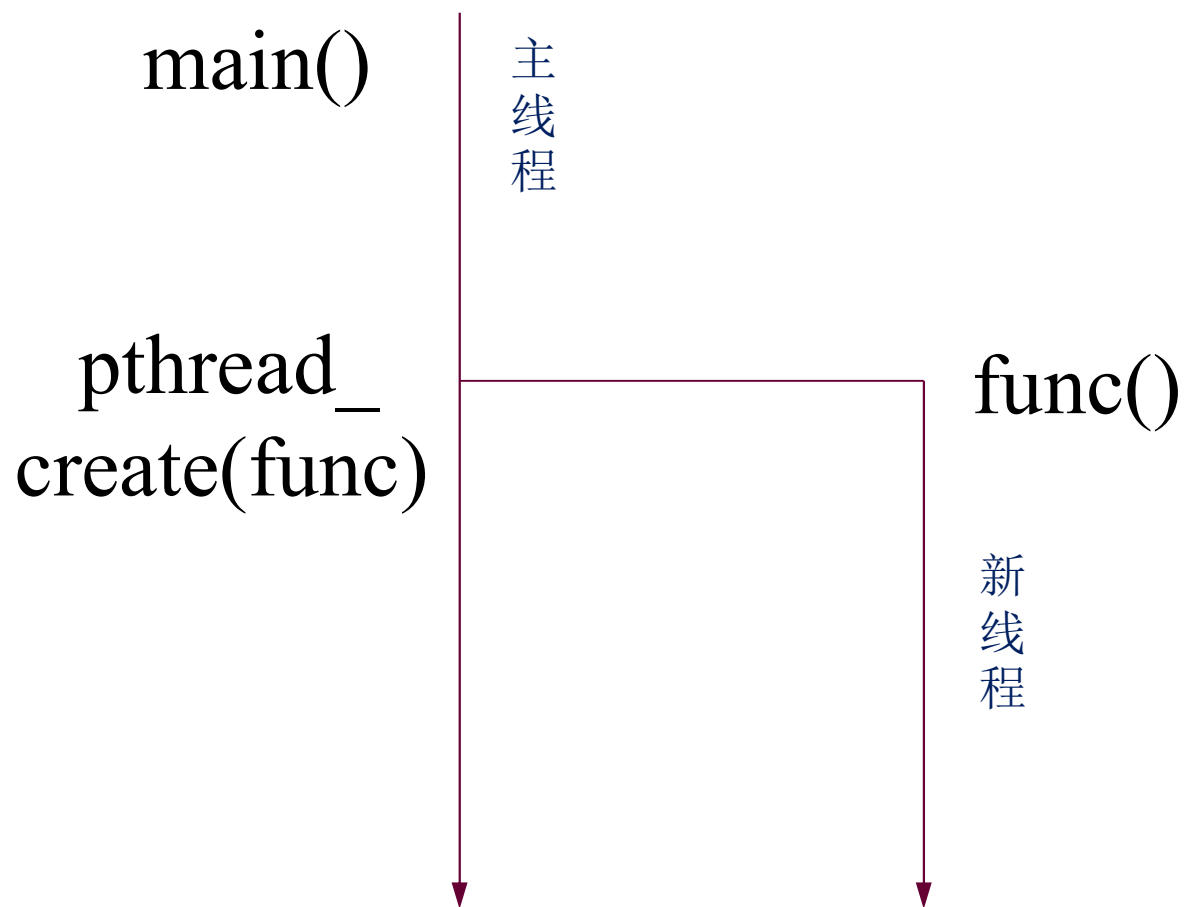
pthread_t* tid: 新创建线程的ID (传出参数)

pthread_attr_t* attr: 设置线程属性, 可以是NULL(传入参数)

start_routine* f: 在新线程中运行的函数地址(传入参数)

void* arg: 函数f的参数, 如果要传多个参数, 使用结构体指针

新线程的创建与执行



二. 线程标识

线程ID

```
#include <pthread.h>
```

```
pthread_t pthread_self(void);
```

获取当前线程的**ID**，在线程内调用该函数

pthread_t有可能是无符号整型(linux)、结构体、指针，为了可移植性，尽量不要打印。

例

```
void* print(void* str)
{
    pthread_t tid = pthread_self();
    if(str != NULL)
        printf("%u: %s\n", (unsigned int)tid, (char*)str);
    return NULL;
}
```

```
int main()
{
    pthread_t ht;
    char str[] = "helloworld";

    pthread_create(&ht, NULL, print, str);

    return 0;
}
```

线程ID 线程属性 入口函数 入口函数参数



进程结束，所有线程终止运行

原因：主线程(main)结束，调用了**exit**

所有线程并发执行

与多进程类似，但上下文切换开销小很多；

共享地址空间

共享静态变量、全局变量、文件描述符等；

三. 线程回收

等待(回收)线程

```
#include <pthread.h>
```

```
int pthread_join(pthread_t tid, void **thread_return);
```

等待tid线程终止运行，如果该线程已经终止，立即返回；
否则，阻塞直到该线程终止

与wait不同，pthread_join只能等待一个指定的线程
终止，不能等待任意一个线程终止

pthread_t tid: 要等待的线程ID (传入参数)

void thread_return**: 线程函数返回的 (void *)指针赋给
thread_return所指的地址，可以是NULL (传出参数)



例

```
void* print(void* str)
{
    pthread_t tid = pthread_self();
    if(str != NULL)
        printf("%u: %s\n", (unsigned int)tid, (char*)str);
    return NULL;
}
```

```
int main()
{
    pthread_t ht;
    char str[] = "helloworld";
    void * tret;

    pthread_create(&ht, NULL, print, str);
    pthread_join(ht, &tret);
    return 0;
}
```

/*线程函数*/

```
void * thr_fn1(void *arg) {  
    printf("thread 1 returning\n");  
    return((void *)1);  
}
```

/*线程函数*/

```
void * thr_fn2(void *arg) {  
    printf("thread 2 exiting\n");  
    pthread_exit((void *)2);  
}
```

```
int main(void) {  
    int      err;  
    pthread_t tid1, tid2;  
    void      *tret;  
    //创建线程  
    err = pthread_create(&tid1, NULL, thr_fn1, NULL);  
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);  
  
    //等待线程终止  
    err = pthread_join(tid1, &tret);  
    printf("thread 1 exit code %d\n", (int)tret);  
  
    err = pthread_join(tid2, &tret);  
    printf("thread 2 exit code %d\n", (int)tret);  
    exit(0);  
}
```

例

```
thread 1 returning  
thread 2 returning  
thread 1 exit code 1  
thread 2 exit code 2
```

四. 线程终止

线程终止的三种方式：

- (1) 从线程函数返回；
- (2) 被同进程内的其它线程取消 (pthread_cancel)；
- (3) 调用 pthread_exit 函数；

线程终止

```
#include <pthread.h>
```

```
void pthread_exit(void *rval_ptr);
```

```
int pthread_cancel(pthread_t tid);
```

从当前线程终止：自杀

终止其它线程：他杀

pthread_cancel并不会立即终止另一个线程，只是发送了请求，另一个线程在到达**cancellation point**（系统调用）的时候才终止

Figure 12-7. Cancellation points defined by POSIX.1

accept	mq_timedsend	putpmsg	sigsuspend
aio_suspend	msgrcv	pwrite	sigtimedwait
clock_nanosleep	msgsnd	read	sigwait
close	msync	readv	sigwaitinfo
connect	nanosleep	recv	sleep
creat	open	recvfrom	system
fcntl2	pause	recvmsg	tcdrain
fsync	poll	select	usleep
getmsg	pread	sem_timedwait	wait
getpmsg	pthread_cond_time dwait	sem_wait	waitid
lockf	pthread_cond_wait	send	waitpid
mq_receive	pthread_join	sendmsg	write
mq_send	pthread_testcancel	sendto	writew
mq_timedreceive	putmsg	sigpause	

pthread_cancel 调用风险

如果线程处于临界区的Cancellation points点时，线程取消，此时会引起死锁

.....

需要时学习

线程取消时执行

```
#include <pthread.h>
```

```
void pthread_cleanup_push(void (*rtn)(void *), void *arg);
```

```
void pthread_cleanup_pop(int execute);
```

五. 线程分离

线程默认属性

类似僵尸进程

线程
分离

在任何时间点上，线程是 **可结合的(joinable)** 或者是 **分离的(detached)**。可结合的线程如果没有被回收，**存储资源(栈等)** 不会被释放；分离的线程不能被回收或者杀死，线程终止时，存储资源由系统自动释放。

分离线程

分离可结合线程tid

```
#include <pthread.h>
```

```
void pthread_detach(pthread_t tid);
```

以pthread_self
返回值作参数，可
以分离自己



```
int main() {  
    while (true) {  
        /*等待请求到来(例如http请求)*/  
        waitQuest();//伪代码  
        pthread_create(&ht, NULL, print, str);  
        pthread_detach(&ht);  
    }  
    return 0;  
}
```

分离线程，
不阻塞

阻塞，等待
线程结束

```
int main() {  
    char* str = "helloworld"  
    pthread_create(&ht, NULL, print, str);  
    pthread_join(&ht, NULL);  
    return 0;  
}
```

3.3 多线程的共享变量

```
#define N 2
void *thread(void *arg);/*线程函数声明*/
char **ptr; /*全局变量*/

int main(){
    int i;
    pthread_t tid;
    char *msgs[N]= {"Hello from T1", "Hello from T2"};

    ptr=msg; /*本地自动变量*/
    for(i = 0; i < N; i++)
        pthread_create(&tid, NULL, thread, (void *)i);
    pthread_exit(NULL);
}

void *thread(void *arg) {
    int myid = (int)arg;
    static int cnt = 0; /*本地静态变量*/

    printf("[%d]: %s(cnt=%d)\n", myid, ptr[myid], ++cnt);
}
```

主线程

msgs是自动变量(栈内)

线程函数(例程)

共享变量

一个变量 v 是共享的，当且仅当它的一个实例被一个以上的线程引用。

C语言存储类型

- 全局变量

定义在函数之外的变量。运行时，一个全局变量在整个地址空间只有一个实例，任何线程都可以引用

- 本地静态变量

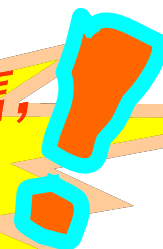
定义在函数内部有 `static` 属性的变量。和全局变量一样，运行时，在整个地址空间只有一个实例，任何线程都可以引用

- 本地自动变量

定义在函数内部没有 `static` 属性的变量。运行时，每一个线程栈内都有一份本地自动变量实例。其它线程可以间接引用本地自动变量

3.4 线程同步

利用共享变量进行线程间通信，
但容易引起“同步错误”



示例：利用多个线程计数

例

```
#define N 100000
volatile int cnt = 0; /* 计数器*/

void *thread(void *arg);

int main() {
    pthread_t tid1, tid2;

    /*创建线程并等待它们结束*/
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /*打印结果*/
    printf("cnt = %d\n", cnt);

    return 0;
}
```

```
/*线程函数*/
void * thread(void *arg) {
    int i;

    for ( i = 0; i < N; i++)
        cnt++;

    return NULL;
}
```

最后计数结果是



200000?

计数错误原因

```
for ( i = 0; i < N; i++)  
    cnt++;
```

将cnt(内存)放入寄存器eax
寄存器eax值加1
寄存器eax值存入cnt (内存)

汇编代码

.L9:

```
    movl    -4(%esp), %eax  
    cmpl    $99999, -4(%ebp)  
    jle     .L12  
    jmp     .L10
```

.L12:

```
    movl    cnt, %eax  
    leal    1(%eax), %eax  
    movl    %eax, cnt
```

Load cnt
Update cnt
Store cnt

.L11:

```
    movl    -4(%esp), %eax  
    leal    1(%eax), %eax  
    movl    %edx, -4(%ebp)  
    jmp     .L9
```

.L10:

IA32上的汇编代码

全局变量cnt最后的
计数结果可能是?
(cnt初始0)



最大:

$2 * N$

最小:

2

汇编代码

H _i 头	}	.L9:	movl -4(%esp), %eax	
		cmpl \$99999, -4(%ebp)		
		jle .L12		
		jmp .L10		
L _i	}	.L12:	movl cnt, %eax	Load cnt
U _i		leal 1 1(%eax), %eax	Update cnt	
S _i		movl %eax, cnt	Store cnt	
	}	.L11:	movl -4(%esp), %eax	
		leal 1 1(%eax), %eax		
T _i 尾		movl %edx, -4(%ebp)		
		jmp .L9		
		.L10:		

IA32上的汇编代码

正确的顺序

步骤	线程	指令	%eax ₁	%eax ₂	cnt
1	1	H ₁	—	—	0
2	1	L ₁	0	—	0
3	1	U ₁	1	—	0
4	1	S ₁	1	—	1
5	2	H ₂	—	—	1
6	2	L ₂	—	1	1
7	2	U ₂	—	2	1
8	2	S ₂	—	2	2
9	2	T ₂	—	2	2
10	1	T ₁	1	—	2

不正确的顺序

步骤	线程	指令	%eax ₁	%eax ₂	cnt
1	1	H ₁	—	—	0
2	1	L ₁	0	—	0
3	1	U ₁	1	—	0
4	2	H ₂	—	—	0
5	2	L ₂	—	0	0
6	1	S ₁	1	—	1
7	1	T ₁	1	—	1
8	2	U ₂	—	1	1
9	2	S ₂	—	1	1
10	2	T ₂	—	1	1

不正确的顺序

步骤	线程	指令	%eax ₁	%eax ₂	cnt
1	1	H ₁	—	—	0
2	1	L ₁	0	—	0
3	1	U ₁	1	—	0
4	2	H ₂	—	—	0
5	2	L ₂	—	0	0
6	1	S ₁	1	—	1
7	1	T ₁	1	—	1
	1	Thread1:2~99 999	99999		99999
	2	U ₂	—	1	1
	2	S ₂	—	1	1
	2	T ₂	—	1	1
	1	H1;L1	1		
	2	Thread2:2~10 0000	1	100000	100000
	1	U1;S1;T1	2		2

一. 互斥 (Mutex)

互斥 可以看作一把锁，保护共享资源同一时刻只能被一个线程访问。

互斥初始化和销毁

互斥变量类型

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *mutexattr);
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

加锁和解锁

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

如果mutex已经被锁，调用线程阻塞，直到mutex被解锁。

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

如果mutex已经被锁，不阻塞，返回**EBUSY**。

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

解锁

调用方式

```
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex);
```

初始化

```
THREAD A:  
pthread_mutex_lock(&mutex);  
cnt++;  
pthread_mutex_unlock(&mutex);
```

使用

```
THREAD B:  
pthread_mutex_lock(&mutex);  
cnt--;  
pthread_mutex_unlock(&mutex);
```

使用

```
pthread_mutex_destroy(&mutex);
```

释放

调用方式2

初始化

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

THREAD A:

```
pthread_mutex_lock(&mutex);  
cnt++;  
pthread_mutex_unlock(&mutex);
```

使用

THREAD B:

```
pthread_mutex_lock(&mutex);  
cnt--;  
pthread_mutex_unlock(&mutex);
```

使用



例

```
#define N 100000
volatile int cnt = 0; /* 计数器*/

void *thread(void *arg);
pthread_mutex_t mutex;

int main() {
    pthread_t tid1, tid2;
    pthread_mutex_init(&mutex, NULL);
    /*创建线程并等待它们结束*/
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_mutex_destroy(&mutex);
    /*打印结果*/
    printf("cnt = %d\n", cnt);

    return 0;
}
```

```
/*线程函数*/
void * thread(void *arg) {
    int i;

    for ( i = 0; i < N; i++) {
        pthread_mutex_lock(&mutex);
        cnt++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```


二. 信号量 (semaphore)

信号量初始化和销毁

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, 0, unsigned int value);
```

```
int sem_destroy(sem_t *sem);
```

信号量变量类型

信号量初始值

为0, 信号量可以被线程共享

非0, 信号量可以被进程共享

P和V

```
#include <semaphore.h>
```

int sem_wait (sem_t *sem);

P

如果信号量的值大于0, 那么进行减一的操作, 函数立即返回.
如果信号量当前等于0, 那么调用就会阻塞, 直到信号量大于0

int sem_trywait(sem_t *sem);

如果信号量的值大于0, 那么进行减一的操作, 函数立即返回.
如果信号量当前等于0, 函数立即返回, errno等于EAGAIN

int sem_post(sem_t *sem);

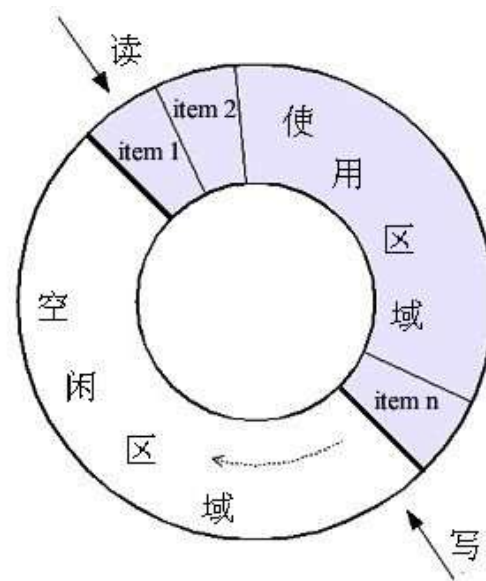
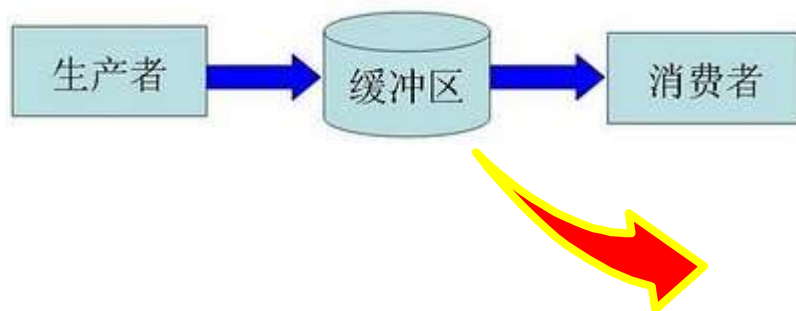
V

信号量的值增加1, 阻塞在这个信号量上的一个线程将会被唤醒

例

生产者-消费者

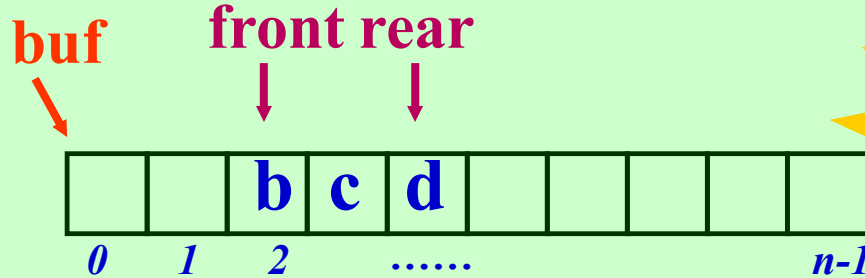
“生产者”线程不断向共享缓冲区写入数据（即生产数据），而“消费者”线程不断从共享缓冲区读出数据（即消费数据）；共享缓冲区共有 n 个；任何时刻只能有一个线程可对共享缓冲区进行操作。



设计要点（阻塞版本）：

- 缓冲区槽满，生产者要阻塞
- 缓冲区所有槽空，消费者要阻塞
- 同一时刻只能有一个线程读或者写缓冲区

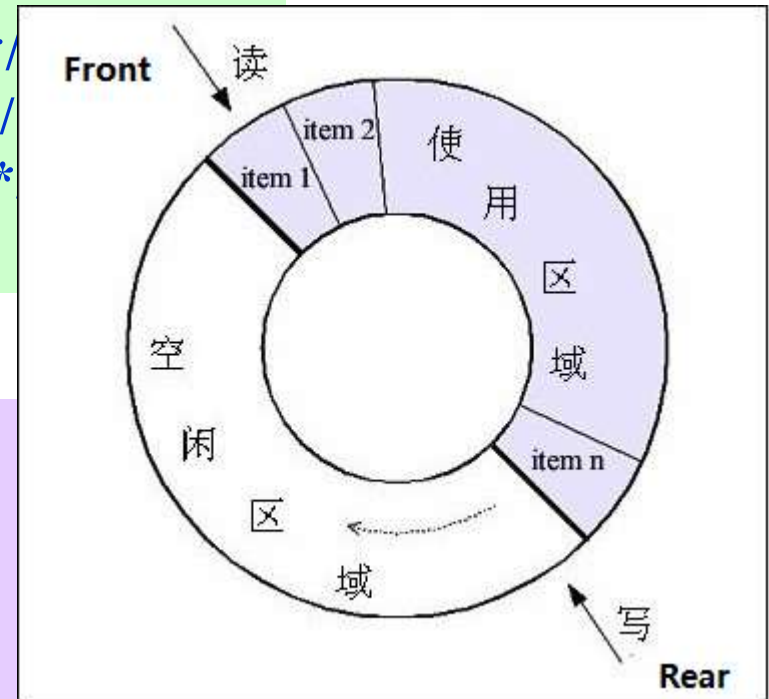
```
typedef struct {
    int *buf;
    int n;
    int front;
    int rear;
    sem_t mutex; /*protects accesses to buf*/
    sem_t nEmpty; /*counts available slots*/
    sem_t nStored; /*counts available items*/
} buf_t;
```



缓冲区数据结构

/*初始化缓冲区，创建n个槽的空间*/

```
void buf_init(buf_t *bp, int n) {
    bp->buf = calloc(n, sizeof(int));
    bp->n = n;
    bp->front = bp->rear = 0;
    sem_init(&bp->mutex, 0, 1);
    sem_init(&bp->nEmpty, 0, n);
    sem_init(&bp->nStored, 0, 0);
}
```



缓冲区初始化

共享缓冲区读写

/*写一项到缓冲区尾部*/

```
void buf_write(buf_t *bp, int item) {  
    int idx;  
    sem_wait(&bp->nEmpty);  
    sem_wait(&bp->mutex);  
    idx = (bp->rear)++;  
    bp->rear = (bp->rear) % (bp->n);  
    bp->buf[idx] = item;  
    sem_post(&bp->mutex);  
    sem_post(&bp->nStored);  
}
```

/*读并删除缓冲区头部数据*/

```
int buf_read(buf_t *bp) {  
    int item, idx;  
    sem_wait(&bp->nStored);  
    sem_wait(&bp->mutex);  
    idx = (bp->front)++;  
    bp->front =  
        (bp->front) % (bp->n);  
    item = bp->buf[idx];  
    sem_post(&bp->mutex);  
    sem_post(&bp->nEmpty);  
    return item;  
}
```

三. 条件变量

条件变量 (Condition Variables) 是一种等待某事件发生的一种同步机制。即，线程挂起，直到共享数据上的某些条件得到满足。

主要包括两个部分：

一部分线程等待“条件变量的条件成立”；

一部分线程判断“条件成立”，唤醒等待线程。(信号发送)

该线程判断
“条件成立”，
发出唤醒信号

```
lock(mutex_x);  
++x;  
if (x == condition to wake) {  
    unlock(mutex_x);  
    signal(cond_var);  
} else {  
    unlock(mutex_x);  
}  
  
lock(mutex_x);  
while (x != condition to wake) {  
    sleep(cond_var, mutex_x);  
}  
unlock(mutex_x);  
/* woken - do useful things */
```



signal to wake



a thread woken

等待被
唤醒

Sleeping
threads
waiting for
signal to
wake

注意要点：

- 条件变量是用来**等待**而不是用来上锁。它是通过一种能够挂起当前正在执行的进程或者放弃当前进程，直到在共享数据上的一些**条件**得到满足。
- 条件变量的操作过程是：首先通知条件变量，然后等待，同时挂起当前进程直到有另外一个进程通知该条件变量为止。

互斥锁一个明显的**缺点**是它只有两种状态：锁定和非锁定。**条件变量**通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，**常和互斥锁一起使用**。另外一个进程通知该条件变量为止。

条件变量初始化和销毁

```
#include <pthread.h>
```

条件变量类型

```
int pthread_cond_init(pthread_cond_t *cond,  
                      pthread_condattr_t *cond_attr)  
int pthread_cond_destroy(pthread_cond_t *cond);
```

一般设置为NULL，使用
缺省属性

常量PTHREAD_COND_INITIALIZER赋给静态分配的
条件变量

等待条件成立

条件
检查

线程进入休眠状态
等待条件改变

注意：互斥参数，为什么？

```
#include <pthread.h>
```

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

```
int pthread_cond_timedwait(pthread_cond_t *cond,  
                           pthread_mutex_t *mutex,  
                           const struct timespec *abstime);
```

限时等待

调用者把锁住的互斥量传递给函数：
1. 函数将调用线程放到等待条件的线程列表上
2. 然后对互斥量进行解锁
这两个操作是原子操作

等待线程

pthread_cond_wait用法:

```
pthread_mutex_lock(&mutex);  
while(condition_is_false)  
{  
    pthread_cond_wait(&cond, &mutex)  
}
```

线程挂起前, unlock mutex
被唤醒首先: lock mutex

condition_is_false=true; //此操作是带锁的, 也就是说
只有一个线程同时进入这块

pthread_mutex_unlock(&mutex); 为什么将mutex作为参数

```
{  
    %Waken do useful things%  
}
```



唤醒等待线程

```
#include <pthread.h>
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

唤醒一个等待 (条件成立) 线程

唤醒所有等待 (条件成立) 线程

解锁线程

```
pthread_mutex_lock(&mutex);  
condition_is_false=false;  
pthread_cond_signal(&cond)  
pthread_mutex_unlock(&mutex)
```

调用方式1

```
pthread_cond_t cond;  
pthread_mutex_t mutex;
```

保护共享变量

主线程:

```
pthread_cond_init(&cond);
```

线程 1、2.....:

```
pthread_mutex_lock(&mutex);  
++ count;  
pthread_cond_signal(&cond);  
pthread_mutex_unlock(&mutex);
```

解锁线程

线程 A、B.....:

```
pthread_mutex_lock(&mutex);  
while (count <= 0)  
    pthread_cond_wait(&cond, &mutex);  
pthread_mutex_unlock(&mutex);
```

等待线程

主线程:

```
pthread_cond_destroy(&cond);
```

调用方式2

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

线程 1、2.....:

```
pthread_mutex_lock(&mutex);  
++ count;  
pthread_cond_signal(&cond);  
pthread_mutex_unlock(&mutex);
```

线程 A、B.....:

```
pthread_mutex_lock(&mutex);  
while (count <= 0)  
    pthread_cond_wait(&cond, &mutex);  
pthread_mutex_unlock(&mutex);
```

```
#define MAXMSGSIZE 128      /*消息最大长度*/
typedef struct msg_node {
    struct msg_node    *m_next;
    char  data[MAXMSGSIZE]; /*存放消息*/
    int   msg_size;         /*实际消息长度*/
} msg_node_t;
```

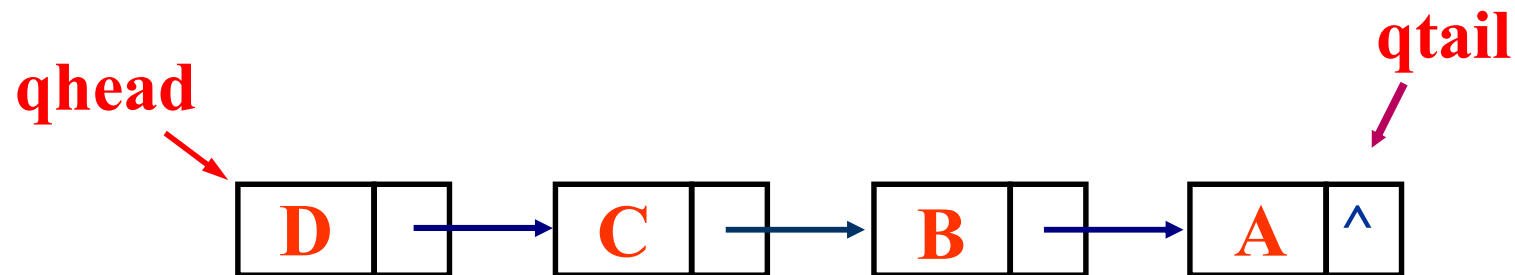
```
msg_node_t *qtail = NULL; /*queue tail*/
msg_node_t *qhead = NULL; /*queue head*/
```

```
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;
```

例

简单的消息
队列

队列节点



队列操作

/*写一项到队列尾部*/

```
void msg_send(const char *msg_ptr,  
              int msg_size)
```

```
{  
    smsg_node_t* msg = malloc(...);  
    memcpy(msg->data, msg_ptr, msg_size);  
    msg->msg_size = msg_size;  
    msg->m_next = NULL;
```

```
    pthread_mutex_lock(&qlock);
```

```
    if(qtail != NULL) {
```

```
        qtail->m_next = msg;
```

```
    }
```

```
    qtail = msg;
```

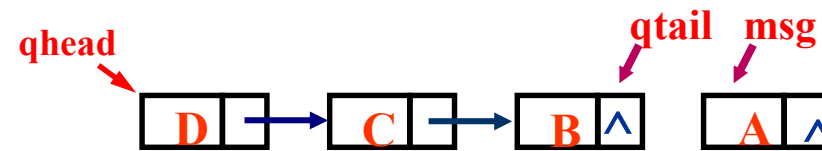
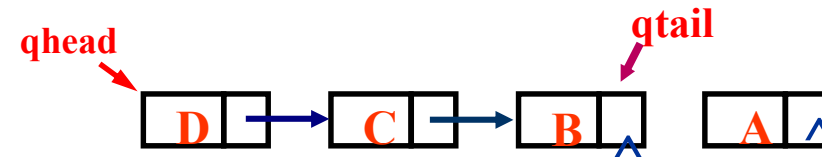
```
    if(qhead == NULL)
```

```
        qhead = msg;
```

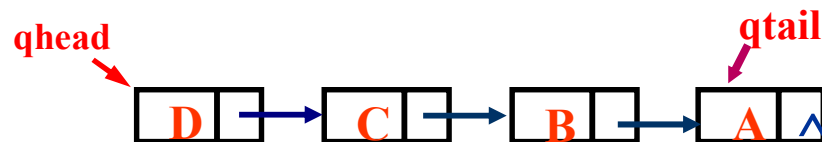
```
    pthread_mutex_unlock(&qlock);
```

```
    pthread_cond_signal(&qready);
```

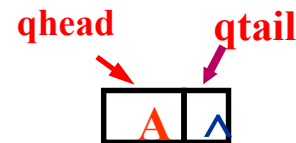
```
}
```



```
if(qtail != NULL)
```



```
if(qhead == NULL)
```



队列操作

```

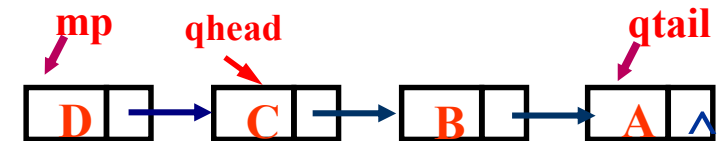
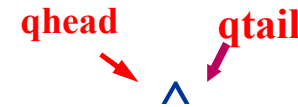
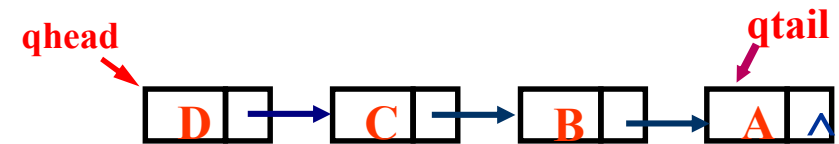
/*从队列头部取一个消息*/
void msg_receive(char * msg_ptr,
                 int* msg_size)
{
    smmsg_node_t *mp;

    pthread_mutex_lock(&qlock);
    while (qhead == NULL)
        pthread_cond_wait(&qready,
                        &qlock);

    mp = qhead;
    qhead = qhead->m_next;
    if(mp == qtail)
        qtail = NULL;
    pthread_mutex_unlock(&qlock);

    /* now process the message mp */
    *msg_size = mp->msg_size;
    memcpy(msg_ptr, mp->data, mp->msg_size);
    free(mp);
}

```





调用pthread_cond_signal, 必然会唤醒一个线程吗?

唤醒丢失问题

什么时候发生唤醒丢失?

1. 没有线程正在处在阻塞等待的状态下
2. 一个线程调用pthread_cond_broadcast() 函数, 另一个线程正处在**测试条件变量和调用pthread_cond_wait函数之间**

在线程未获得相应的互斥锁时调用pthread_cond_signal或pthread_cond_broadcast函数可能会引起唤醒丢失问题。

四. CAS指令

CAS指令：compare-and-swap指令（CMPXCHG），同步机制实现的基石。

指令语义：将寄存器（*reg）内的值与给定值(oldval)比较，当前且仅当相等时，寄存器内的值被赋一个新的值(newval)，返回寄存器内存放的旧值(*reg)。

C语言表达CAS指令

```
int compare_and_swap (int* reg, int oldval, int newval)
{
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        *reg = newval;
    return old_reg_val;
}
```

三步操作由CAS
一条指令完成

CAS指令： compare-and-swap指令。

变为返回bool值得形式

优点：可以调用者知道有没有更新成功

```
bool compare_and_swap (int* reg, int oldval, int newval)
{
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        { *reg = newval; return true }
    return old_reg_val; return false;
}
```

Fetch And Add，一般用来对变量做 +1 的原子操作

Test-and-set，写值到某个内存位置并传回其旧值。汇编指令BST

如何使用CAS实现互斥



```
int compare_and_swap (int* reg, int oldval, int newval)
```

```
int reg = 1; //对比pthread_mutex_t mutex

void mutex_lock() {
    int old_reg;
    do {
        old_reg = compare_and_swap(&reg, 1, 0);
    } while (reg == old_reg);
}

void mutex_unlock() {
    compare_and_swap(&reg, 0, 1);
}
```

如何使用CAS实现互斥

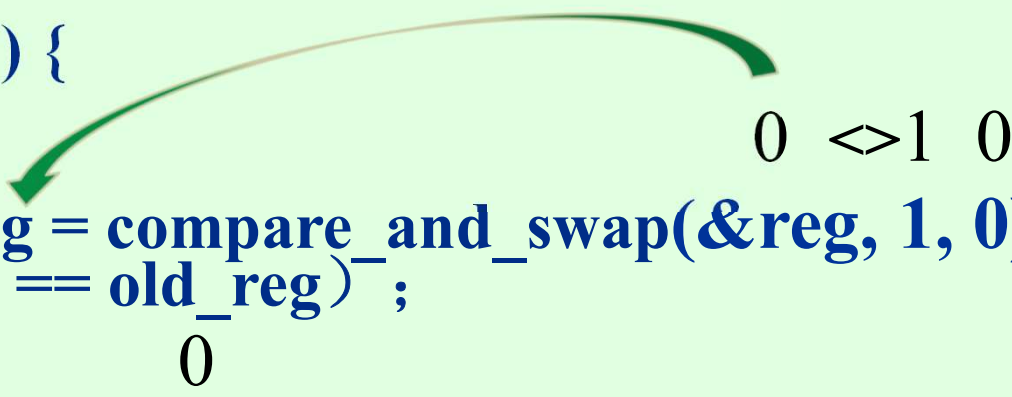


int compare_and_swap (int* reg, int oldval, int newval)

```
int reg = 1; //对比pthread_mutex_t mutex

void mutex_lock() {
    int old_reg;
    do {
        old_reg = compare_and_swap(&reg, 1, 0);
    } while (reg == old_reg);
}

void mutex_unlock() {
    compare_and_swap(&reg, 0, 1);
}
```



如何使用CAS实现互斥



int compare_and_swap (int* reg, int oldval, int newval)

```
int reg = 1; //对比pthread_mutex_t mutex

void mutex_lock() {
    int old_reg;
    do {
        old_reg = compare_and_swap(&reg, 1, 0);
    } while (reg == old_reg);
}

void mutex_unlock() {
    compare_and_swap(&reg, 0, 1);
}
```


使用CAS实现无锁(lock-free)队列

进队:

bool CAS (T* reg, T oldval, T newval)

算法:

EnQueue(x) { /*进队列*/

q = new record();

q->value = x;

q->next = NULL;

do {

p = tail; //取链表尾指针

} while(CAS(p->next, NULL, q) != true);

/*一旦线程T_i走到此处，在重置尾节点之前，其它线程一直在A处重试，直到T_i执行完语句B*/

→若线程T_i在此处退出？其它线程将会死锁

CAS(tail, p, q); //置尾节点

}

1. 创建新结点;
2. 取得的链表的尾指针p;
→此时tail可能被其它线程修改
3. 如果p是尾指针，将新节点缀到尾指针之后；否则，返回2；
4. 重置tail（利用CAS保证修改的原子性）

进队操作：版本2：

```
EnQueue(x) { /*进队列*/
```

```
    q = new record();
```

```
    q->value = x;
```

```
    q->next = NULL;
```

```
    p = tail;
```

```
    oldp = p;
```

```
    do {
```

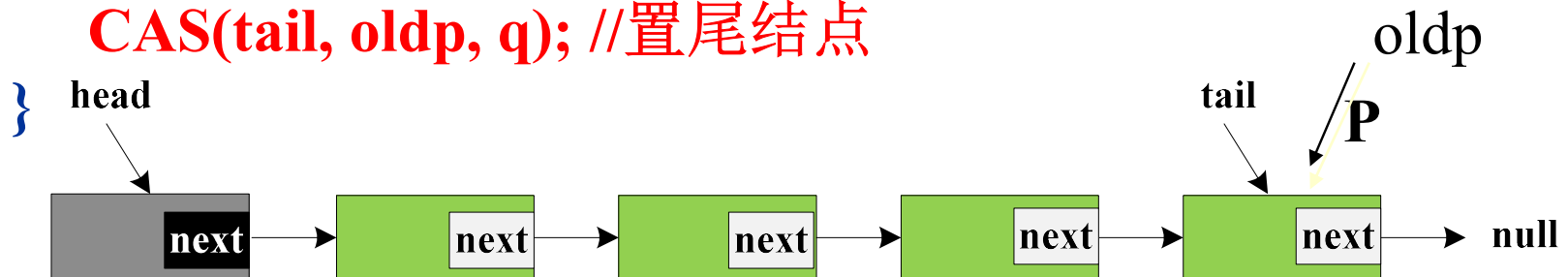
```
        while (p->next != NULL)
```

/*不等tail被重置，新
节点总能进队列*/

```
        p = p->next;
```

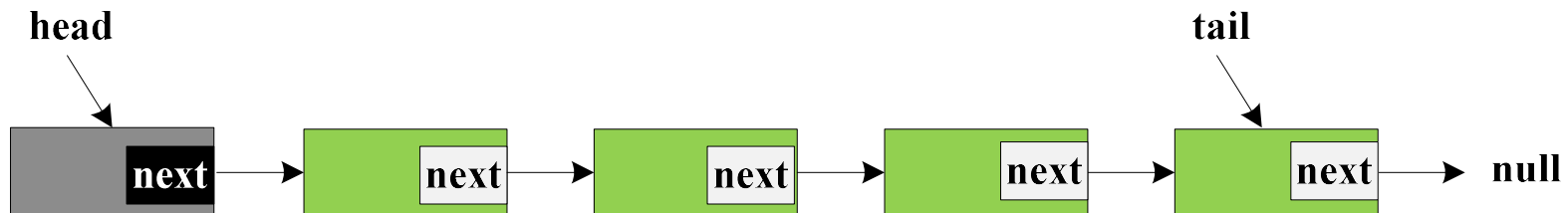
```
    } while( CAS(p->next, NULL, q) != true);
```

```
    CAS(tail, oldp, q); //置尾结点
```



出队操作

```
DeQueue() { /*出队列*/  
    do {  
        p = head;  
        if (p ->next == NULL) {  
            return ERR_QUEUE_EMPTY;  
        }  
    } while( CAS(head, p, p->next) != true);  
    return p->next->value;  
}
```



3.5 多线程信号处理

同步信号(synchronous signals): 进程(线程)的某个操作产生的信号, 例如SEGILL、SIGSEGV、SIGFPE等。

异步信号(asynchronous signals): 类似用户击键这样的进程外部事件产生的信号叫做异步信号, 例如kill命令、ctrl+c产生的信号。

异步信号由哪一个线程接收



某线程内产生的同步信号, 由谁接收



线程信号处理函数

```
#include <signal.h>
```

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

类似sigprocmask，阻塞或者取消阻塞信号

```
int pthread_kill(pthread_t thread, int sig);
```

向另外一个线程发送信号

```
int sigwait(const sigset_t *set, int *sig);
```

信号等待函数，挂起线程，直到set集合内的信号到达；

sigwait的特别之处：自动取消阻塞set集合内的信号

线程调用sigwait之前，必须阻塞那些它正在等待的信号。

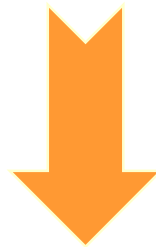
异步信号由哪一个线程接收



如果所有线程都未阻塞该信号，则接收线程不确定，可能是任意线程；

如果只有一个线程未阻塞该信号，则信号将送达该线程。

多线程“异步信号”处理模式



```
void * sig_thread(void *arg) {  
    sigset_t *set = (sigset_t *) arg;  
    int s, sig;  
    for (;;) {  
        sigwait(set, &sig);  
        printf("Signal handling thread got signal %d\n", sig);  
    }  
}
```

取消阻塞信号，并等待信号到达

```
int main() {  
    pthread_t thread;  
    sigset_t set;  
  
    /* 主线程阻塞信号，其它线程继承信号阻塞 */  
    sigemptyset(&set);  
    sigaddset(&set, SIGQUIT);  
    sigaddset(&set, SIGUSR1);  
    pthread_sigmask(SIG_BLOCK, &set, NULL);  
    pthread_create(&thread, NULL, &sig_thread, (void *) &set);  
    /* 主线程继续创建其它线程或者进行其它工作 */  
    pause();  
}
```

阻塞信号

信号处理线程，继承了阻塞的信号

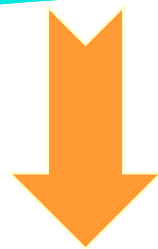
某线程内产生的同步信号，由谁接收



线程内产生的同步信号（SIGSEGV等）由本线程接收。

捕捉同步信号的用途之一：程序故障时，打印错误信息（堆栈、错误发生的位置），方便调试

示例




```
void* sig_thread(void* arg){  
    int i = 0;  
    int* nullptr = NULL;  
  
    signal(SIGSEGV, handler);  
  
    /*generate SIGSEGV signal*/  
    *nullptr = i;  
    return NULL;  
}
```

```
void handler(int sig) {  
    printf("Catch signal %d\n", sig);  
    exit(0);  
}
```



注册信号处理函数，处理本线程产生的同步信号

```
int main(){  
    pthread_t ht;  
  
    pthread_create(&ht, NULL, sig_thread, NULL);  
    pthread_join(ht, NULL);  
    return 0;  
}
```

3.6 并发常见问题

- 线程安全
- 可重入函数(reentrant function)
- 竞争
- 死锁
- 假共享
- 线程个数限制

一. 线程安全(thread safety)

线程安全：一段代码(函数)被称为线程安全的，当且仅当被多个线程反复调用时，一直产生正确的结果。

线程不安全函数分类：

1. 共享资源（变量）未做保护

1.1 不保护共享变量

```
int count;  
void * thread(void *arg){  
    .....  
    count++;  
    .....  
}
```

1. 利用同步操作保护共享变量
互斥、信号量等

2. 原子操作

Linux/Unix的atomic_set等、java的AtomicInteger等

1.2 意外共享

```
extern int errno;  
int open(const char *path, int oflag, ... ) {  
    .....  
    errno = EACCES;  
    .....  
}
```

例

__thread int errno;

使用线程本地存储（Thread-local storage）：同样的变量名，其实例在不同的线程中位于不同的存储位置

__thread关键字(编译器支持)或pthread Thread-Specific Data

__thread 关键字:使用于**global**变量，使每个线程都私有一份

pthread Thread-Specific Data

(1)"线程相关的数据"可以是一个全局变量,并且

(2)每个线程存取的"线程相关的数据"是相互独立的

1.3 共享资源操作未保护

```
int function() {  
    char *filename="/etc/config";  
    FILE *config;  
    if(file_exist(filename)){  
        config=fopen(filename);  
        fputs(config, ....);  
    }  
}
```

其它线程可能正在读该配置文件，
或者删除该文件

- 1.尽量在一个线程内进行共享资源的操作
- 2.其它办法？

2.返回指向静态变量的指针的函数

```
char * getenv(const char *name) {  
    static char envbuf[ARG_MAX];  
    .....  
    return envbuf;  
}
```

- 静态变量
- 返回一个指向一个结构的指针

1.重写函数

— 调用者传递存放结果的地址

2.lock-and-copy（加锁-拷贝）

3. 调用线程不安全的函数

```
void func() {  
    char username[64];  
    strcpy(username, getenv("USER"));  
    .....  
}
```

调用时采用“加锁-拷贝(lock-and-copy)”

例

```
char user [64];  
pthread_mutex_lock(&mutex);  
strcpy(user, getenv("USER"));  
pthread_mutex_unlock(&mutex);
```


二. 可重入函数 (reentrant function)

A function is called **reentrant** if it can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete executing.

例

```
int function() {  
    mutex_lock();  
    ...  
    function body  
    ...  
    mutex_unlock();  
}
```

显示可重入:所有的函数参数都是传值传递的, 并且所有的数据引用都是本地的自动栈变量

三. 竞争 (race)

竞争： 当一个程序的正确性依赖于一个线程在另一个线程到达y点之前，到达它的控制流中的x点时，就会发生竞争。

例

N=4

```
int main() {  
    pthread_t tid[N];  
    int i;  
    for(i = 0; i < N; i++)  
        pthread_create(&tid[i], NULL, thread, &i);  
    for(i = 0; i < N; i++)  
        pthread_join(tid[i], NULL);  
    exit(0);  
}  
  
void *thread(void * arg) {  
    int myid = *((int*)arg);  
    printf("Hello from thread %d\n", myid);  
    return NULL;  
}
```

Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3

三. 竞争 (race)

竞争： 当一个程序的正确性依赖于一个线程在另一个线程到达y点之前，到达它的控制流中的x点时，就会发生竞争。

例

```
pthread_create(&tid[1], NULL, thread, &1);  
int myid = 1;  
pthread_create(&tid[2], NULL, thread, &2);  
int myid = 2;  
pthread_create(&tid[3], NULL, thread, &3);  
int myid = 3;  
pthread_create(&tid[4], NULL, thread, &4);  
int myid = 4;
```

```
Thread1 pthread_create(&tid[1], NULL, thread, &1);  
Thread1 int myid = 1;  
Thread2 pthread_create(&tid[2], NULL, thread, &2);  
Thread3 pthread_create(&tid[3], NULL, thread, &3);  
Thread2 int myid = 3;  
Thread3 int myid = 3;
```

。 。 。

例

N=4

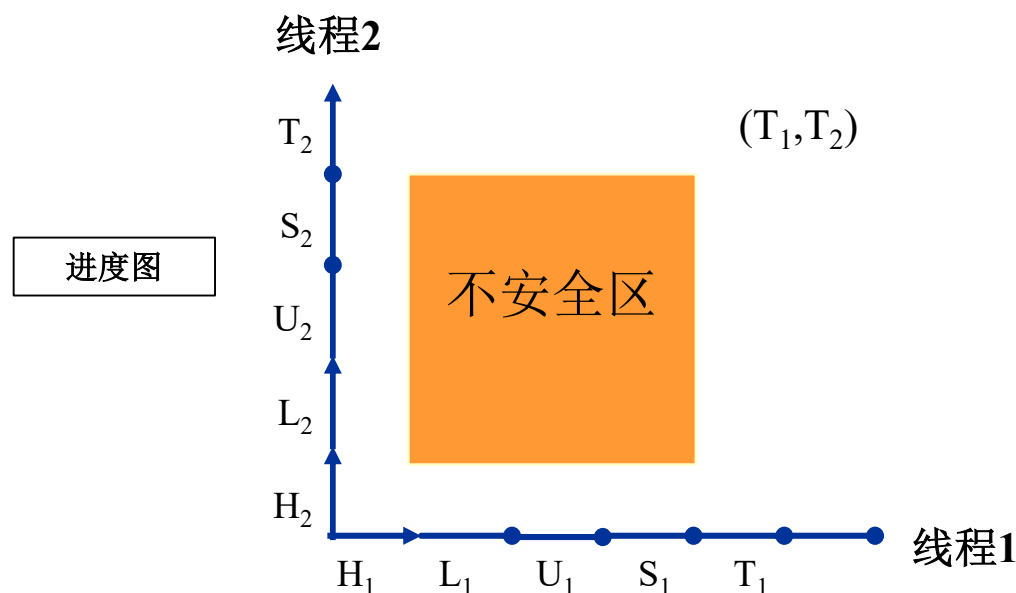
```
int main() {
    pthread_t tid[N];
    int i, *ptr;
    for(i = 0; i < N; i++) {
        ptr= Malloc(sizeof(int));
        *ptr=i;
        pthread_create(&tid[i], NULL, thread, ptr);
    }
    for(i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    exit(0);
}

void *thread(void * vargp) {
    int myid = *((int*)vargp);
    free (vargp)
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

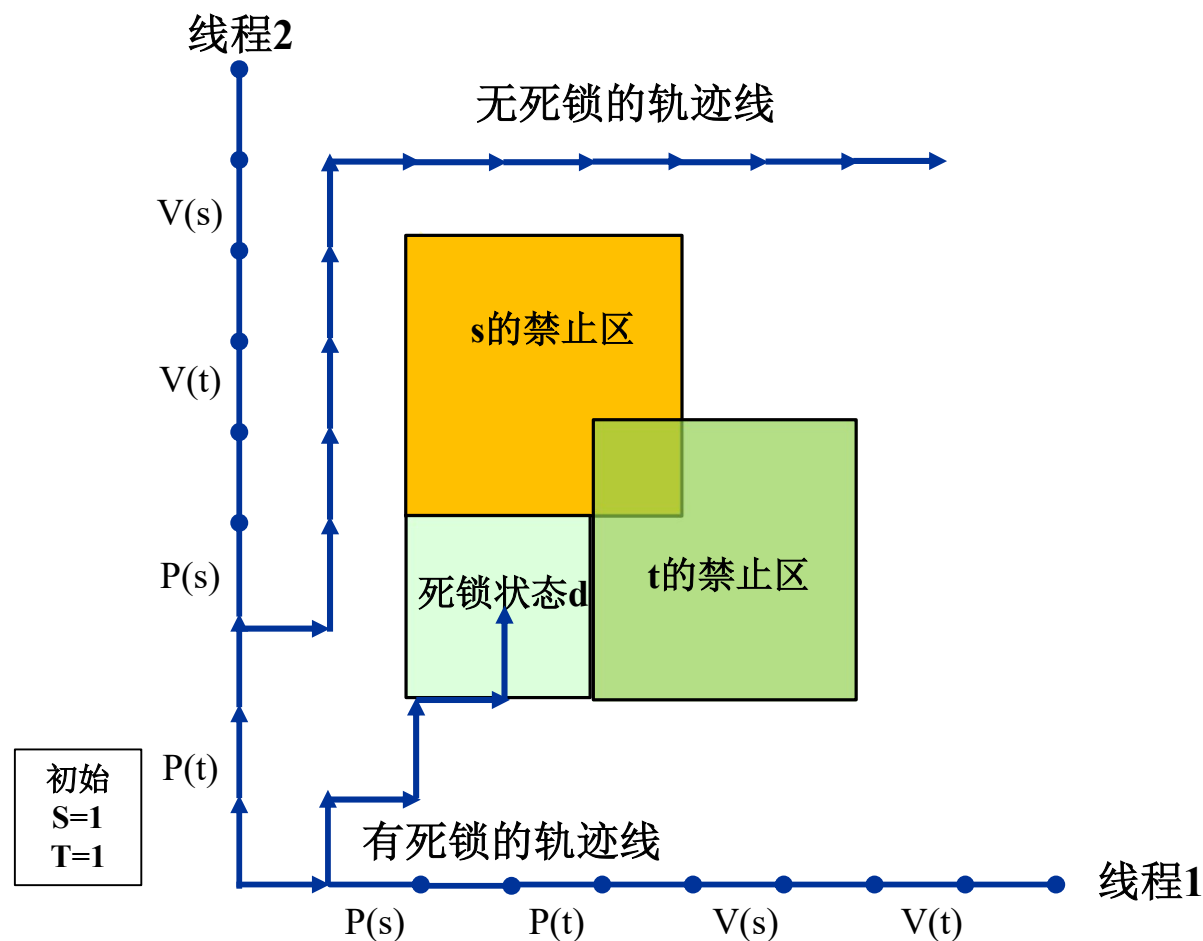
四. 死锁 (deadlock)

死锁：信号量引入死锁。一组线程被阻塞了，等待一个永远也不会为真的条件。

互斥锁加锁顺序规则：如果对于程序中每对互斥锁(s, t)，每个同时占用s和t的线程都按照相同的顺序对它们加锁，那么这个程序就是无死锁的。



四. 死锁 (deadlock)



互斥锁加锁顺序规则：如果对于程序中每对互斥锁(s, t)，每个同时占用s和t的线程都按照相同的顺序对它们加锁，那么这个程序就是无死锁的。

五. 假共享(false sharing)

假共享：分别被两个线程使用的变量，由于存储位置靠得太近，有可能被放到一个cache line（通常64字节）内，此时会引起假共享，严重影响性能。

例

```
struct foo {  
    volatile int x;  
    volatile int y;  
};  
struct foo f;
```

变量声明

线程0内

```
void inc_x(){  
    for (int i = 0; i < 1000000; ++i)  
        ++f.x;  
}
```

线程1内

```
int sum_y() {  
    int s = 0;  
    for (int i = 0; i < 1000000; ++i)  
        s += f.y;  
    return s;  
}
```

Cache line

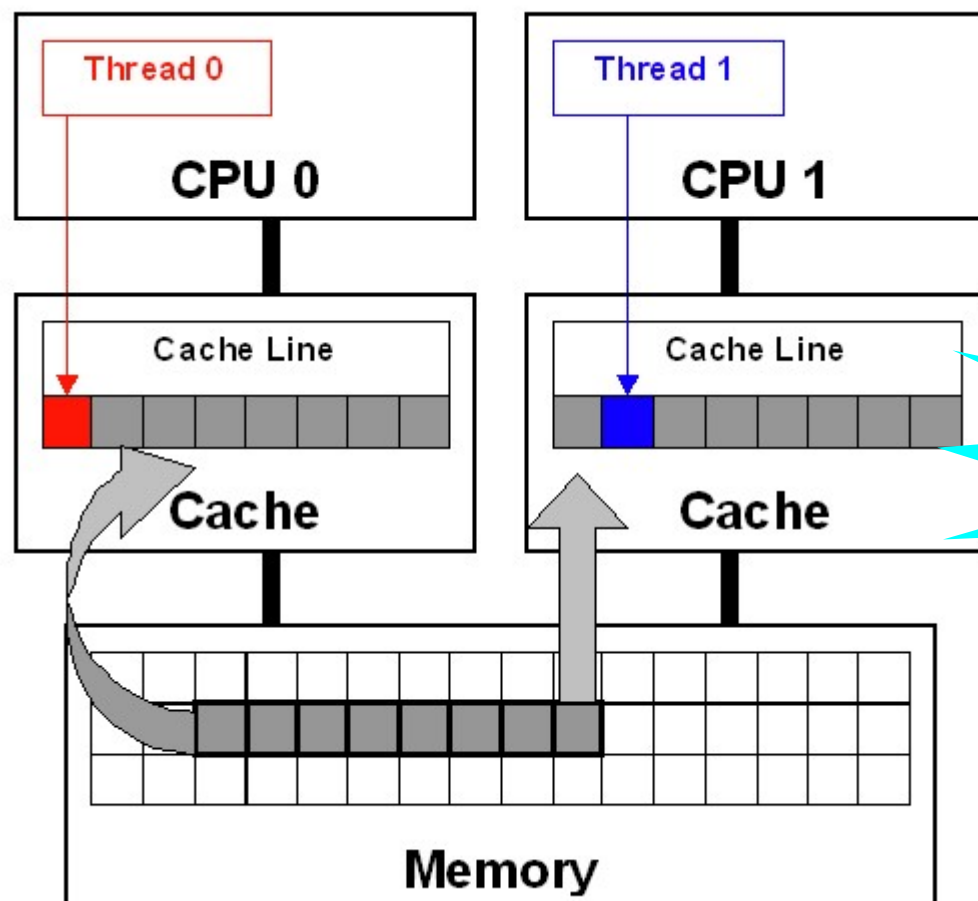
Cache中的数据组成以块为单位，该块称为 **cache line**, 典型大小是64~128字节，**cache line**是从内存读写的最小单位。

x(CPU 0)被修改, y(CPU 1)所在的cache line被置无效

缓存一致性

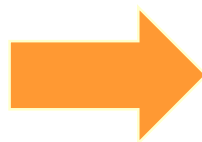
对于多核（多处理）CPU，如果一个**cache line**(L_i)内的数据被更改，其它Cache上 L_i 的副本，都会被标识为**无效**，需要重新从内存读取。

内存延迟**250**时钟周期，**L1 cache 3**个时钟周期



避免假共享

```
struct foo {  
    volatile int x;  
    volatile int y;  
};  
struct foo f;
```



```
struct foo {  
    volatile int x;  
    char padding[60];  
    volatile int y;  
};  
struct foo f;
```

x与y存储位置间填充数据，
确保x与y不处于一个cache line

六. 线程个数限制

创建多少个线程最为合适？
性能最高、充分发挥计算能力

1. 所有线程执行计算密集型计算

创建线程 == CPU核数，

创建线程过多，会引起频繁的上下文切换

2. 既有计算密集型线程，也有I/O线程

创建线程 == CPU核数 + n(个网络I/O线程) + 1(个磁盘I/O线程)

如果有多块磁盘，可以启用多个磁盘I/O线程，但要保证每个线程读不同磁盘；

3.7 线程程序的性能分析

问题：

统计整型数组中等于10的元素的个数



例

性能测试方法

1. 计时精确到**微秒**
2. 运行程序100次，取程序**平均运行时间**
3. 测试时，保证无其它作业在执行

实验环境1

(1) 硬件环境

台式机，一个Intel Core i7-950 CPU【四核（超线程）、64K L1 Cache、1M L2 Cache、共享8M L3 Cache】

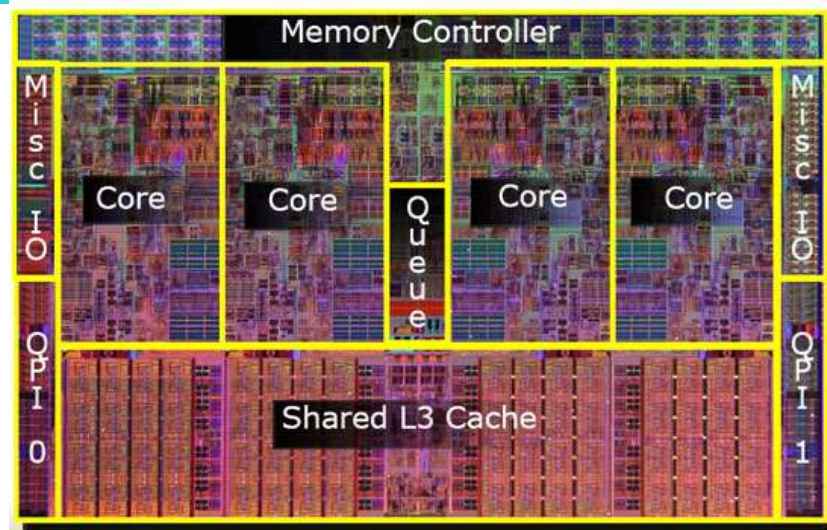
(2) 软件环境

Red Hat Enterprise Linux Server release 5.4【内核版本：2.6.18-164.el5】

gcc版本4.1.2，启用-O2优化

(3) 数据

80MB的整型数组，随机分布有31%值为10的元素。



实验环境2

(1) 硬件环境

IBM服务器，四个双核Xeon 7041 CPU【共有8个CPU核，32K L1 Cache、2M L2 Cache，没有L3 Cache】

(2) 软件环境

SUSE Linux Enterprise Server 10【内核版本：2.6.16.60-0.21-bigsm】

gcc版本4.1.2，启用-O2优化

(3) 数据

80MB的整型数组，随机分布有26%值为10的元素。



问题： 统计整型数组中等于10的元素的个数

版本1： 串行

```
int gCount;//全局变量
int count10() {
    int i;
    gCount = 0;

    for(i = 0; i < ARRAY_LEN; i++) {
        if(array[i] == 10) {
            gCount++;
        }
    }
    return gCount;
}
```

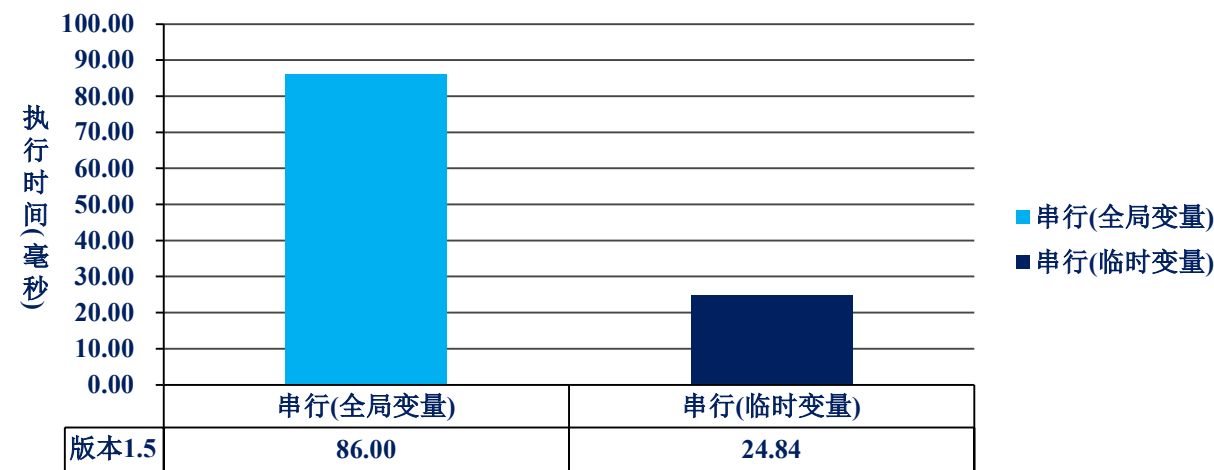
如何提高版本1的性能
(不利用并行技术)



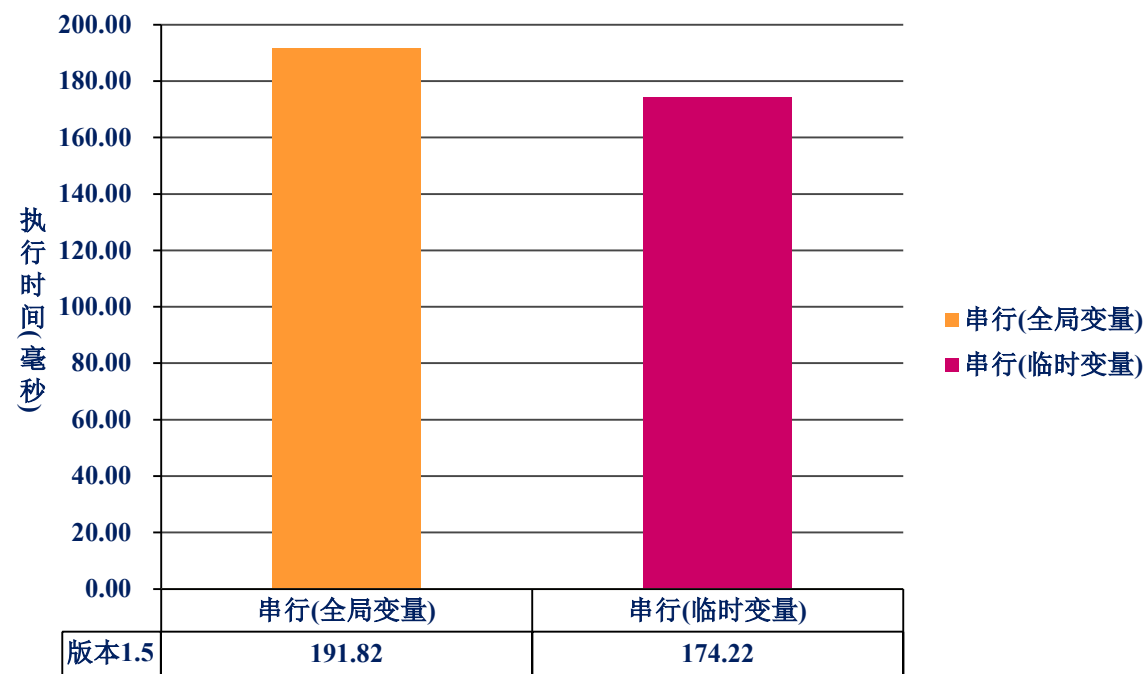
版本1.5:串行

```
int count10() {  
    int i;  
    int count = 0; //临时变量  
  
    for(i = 0; i < ARRAY_LEN; i++) {  
        if(array[i] == 10) {  
            count++;  
        }  
    }  
    return count;  
}
```

版本1.5的性能（一个四核Intel Core i7-950）

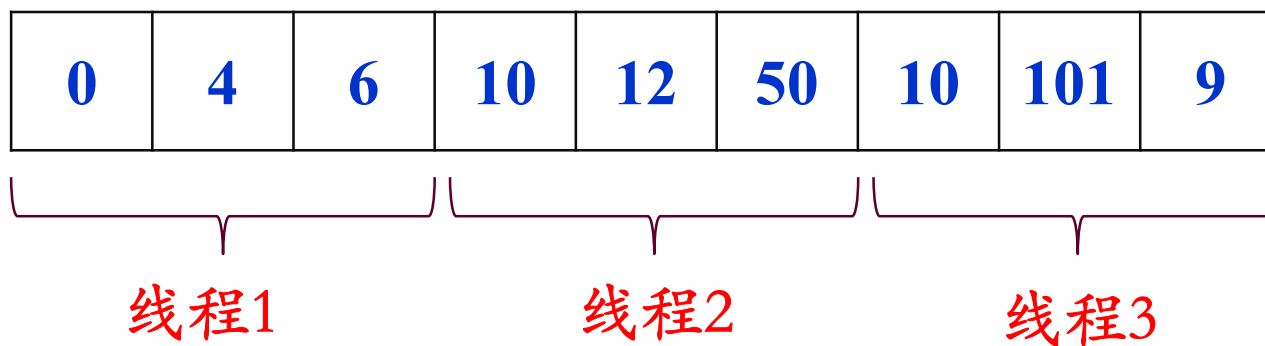


版本1.5的性能（四个双核Intel Xeon）





如何利用并行技术提高性能



版本2:

创建多个线程

```
int count10() {  
    int i;  
    pthread_t tids[MAX_THREADS];  
  
    gCount = 0;  
    for(i = 0; i < gThreadNum; i++) {  
        pthread_create(&tids[i], NULL, count10_thread, (void*)i);  
    }  
    for(i = 0; i < gThreadNum; i++) {  
        pthread_join(tids[i], NULL);  
    }  
    return gCount;  
}
```

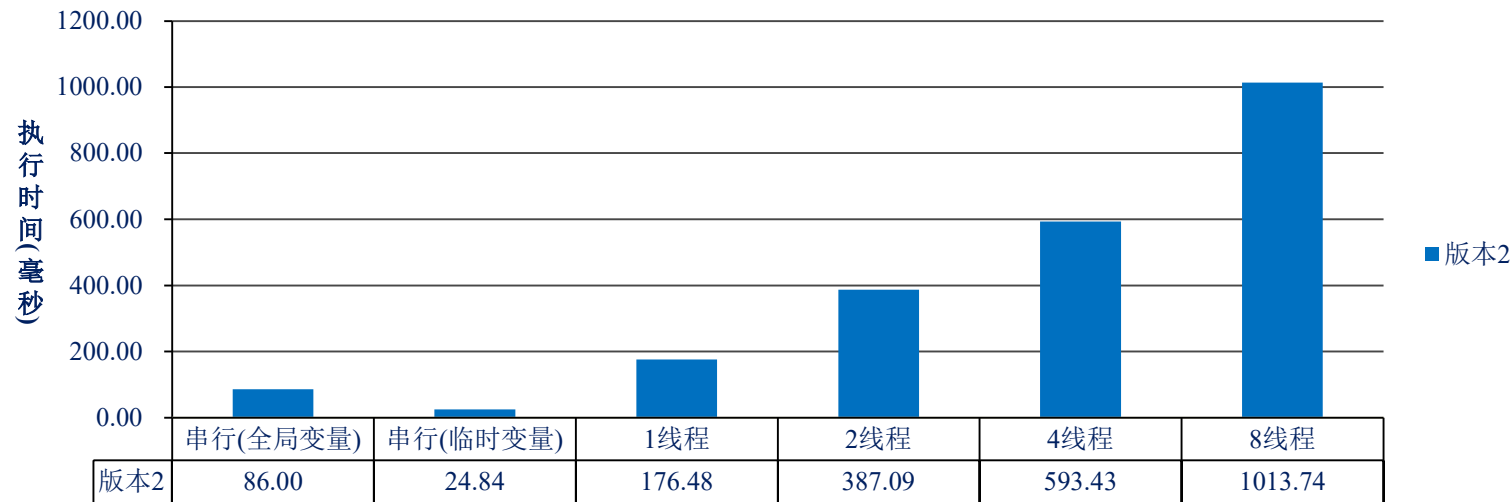
版本2:

每个线程统计一部分

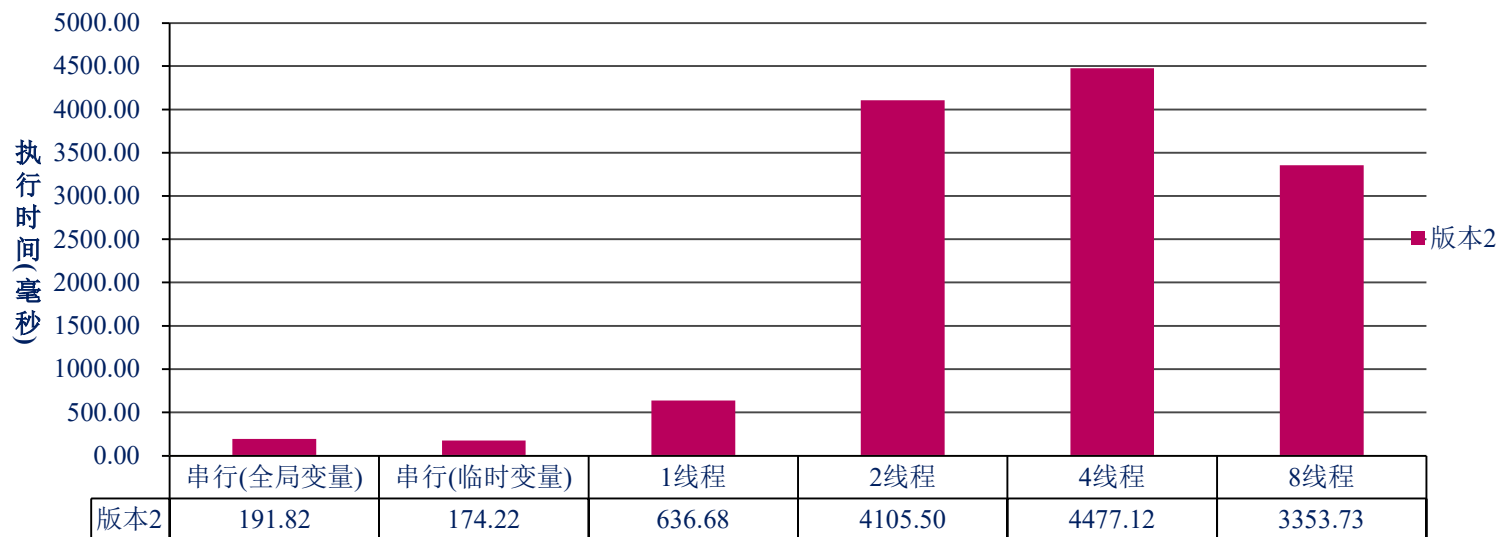
```
void * count10_thread(void * arg) {  
    int myid = (int)arg;  
    int len_per_thread = ARRAY_LEN / gThreadNum;  
    int start = myid * len_per_thread;  
  
    for(i = start; i < start + len_per_thread; i++) {  
        if(gArray[i] == 10) {  
            pthread_mutex_lock(&lock);  
            gCount++;  
            pthread_mutex_unlock(&lock);  
        }  
    }  
}
```

版本2的性能分析

版本2的性能（Intel Core i7-950）



版本2的性能（四个双核Intel Xeon）





如何改进版本2的性能



策略 增大并行的粒度，降低同步次数

版本3:

创建多个线程: 与版本2相同

```
int count10() {  
    int i;  
    pthread_t tids[MAX_THREADS];  
  
    gCount = 0;  
    for(i = 0; i < gThreadNum; i++) {  
        pthread_create(&tids[i], NULL, count10_thread, (void*)i);  
    }  
    for(i = 0; i < gThreadNum; i++) {  
        pthread_join(tids[i], NULL);  
    }  
    return gCount;  
}
```

版本3 :

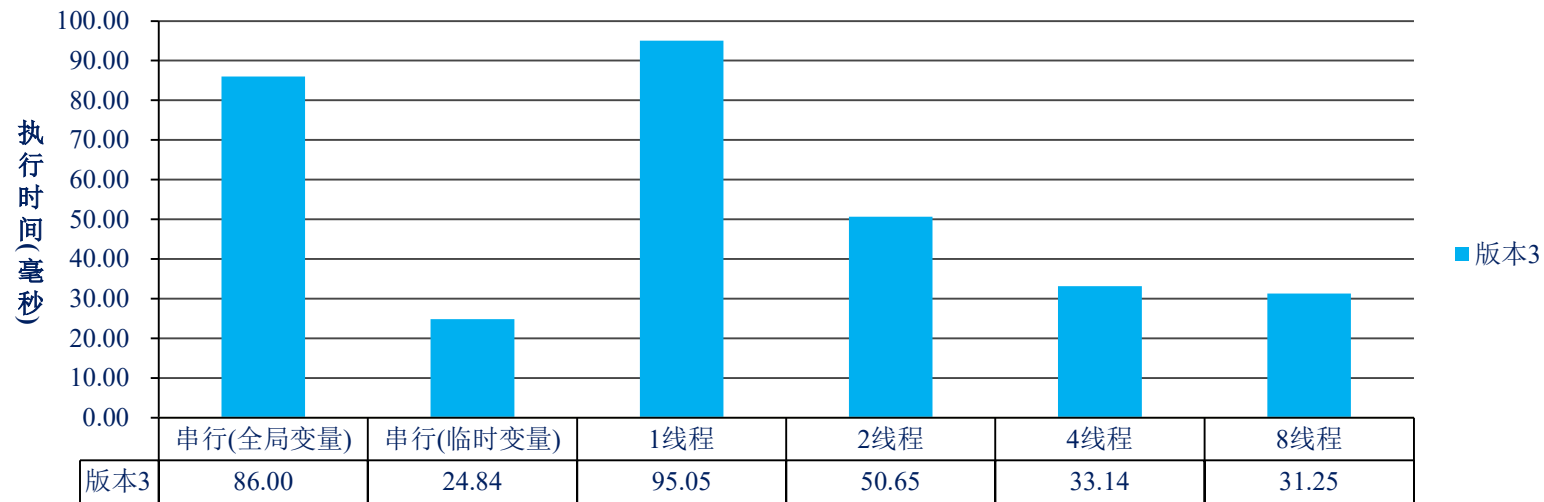
每个线程统计一部分

```
int  gPrivateCount[MAX_THTREADS];
void * count10_thread(void * arg) {
    int myid = (int)arg;
    int len_per_thread = ARRAY_LEN / gThreadNum;
    int start = myid * len_per_thread;

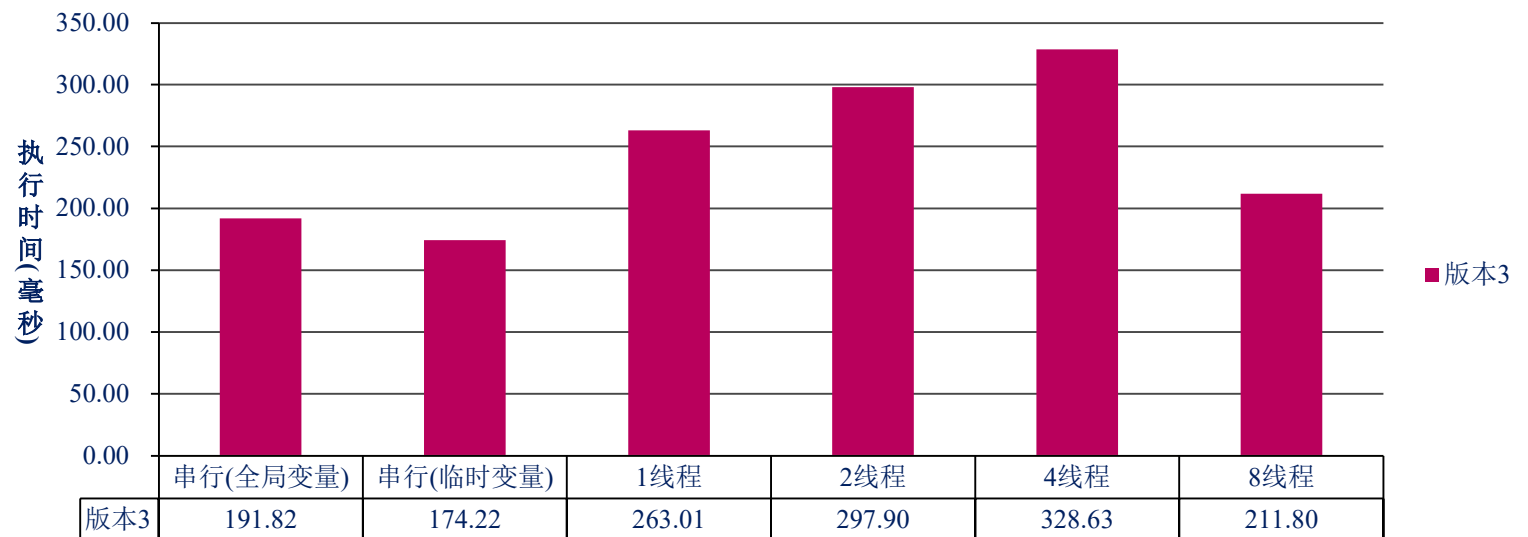
    gPrivateCount[myid] = 0;
    for(i = start; i < start + len_per_thread; i++) {
        if(gArray[i] == 10) {
            gPrivateCount[myid]++;
        }
    }
    pthread_mutex_lock(&lock);
    gCount += gPrivateCount[myid];
    pthread_mutex_unlock(&lock);
}
```

版本3的性能分析

版本3的性能（Intel Core i7-950）



版本3的性能（四个双核Intel Xeon）





如何改进版本3的性能



策略 消除假共享

回忆：假共享(false sharing)

假共享：分别被两个线程使用的变量，由于存储位置靠得太近，有可能被放到一个cache line（通常64字节）内，此时会引起假共享，严重影响性能。

例

```
struct foo {  
    volatile int x;  
    volatile int y;  
};  
struct foo f;
```

变量声明

线程0内

```
void inc_x(){  
    for (int i = 0; i < 1000000; ++i)  
        ++f.x;  
}
```

线程1内

```
int sum_y() {  
    int s = 0;  
    for (int i = 0; i < 1000000; ++i)  
        s += f.y;  
    return s;  
}
```

Cache line

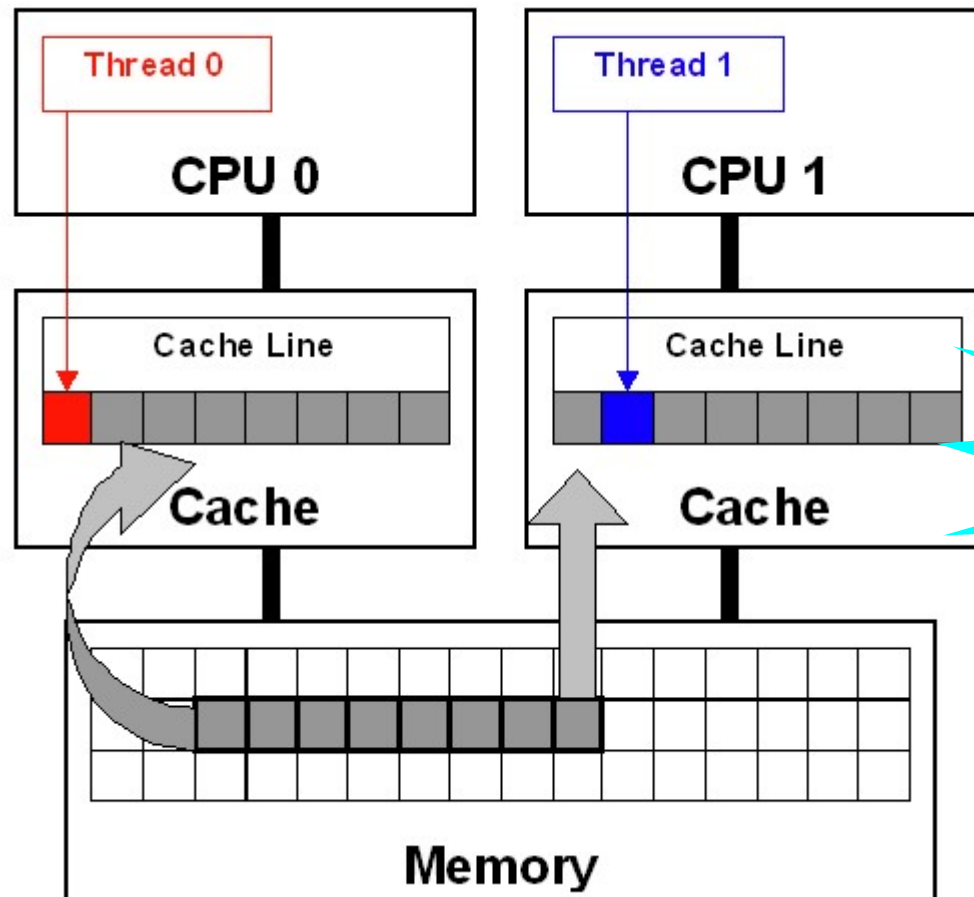
Cache中的数据组成以块为单位，该块称为 **cache line**, 典型大小是64~128字节，**cache line** 是从内存读写的最小单位。

x(CPU 0)被修改, y(CPU 1)所在的cache line被置无效

缓存一致性

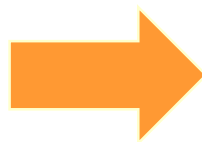
对于多核（多处理）CPU，如果一个**cache line**(L_i)内的数据被更改，其它 Cache 上 L_i 的副本，都会被标识为**无效**，需要重新从内存读取。

内存延迟**250**时钟周期，**L1 cache 3**个时钟周期



避免假共享

```
struct foo {  
    volatile int x;  
    volatile int y;  
};  
struct foo f;
```



```
struct foo {  
    volatile int x;  
    char padding[60];  
    volatile int y;  
};  
struct foo f;
```

x与y存储位置间填充数据，
确保x与y不处于一个cache line

版本4:

创建多个线程: 与版本3相同

```
int count10() {
    int i;
    pthread_t tids[MAX_THREADS];

    gCount = 0;
    for(i = 0; i < gThreadNum; i++) {
        pthread_create(&tids[i], NULL, count10_thread, (void*)i);
    }
    for(i = 0; i < gThreadNum; i++) {
        pthread_join(tids[i], NULL);
    }
    return gCount;
}
```

版本4:

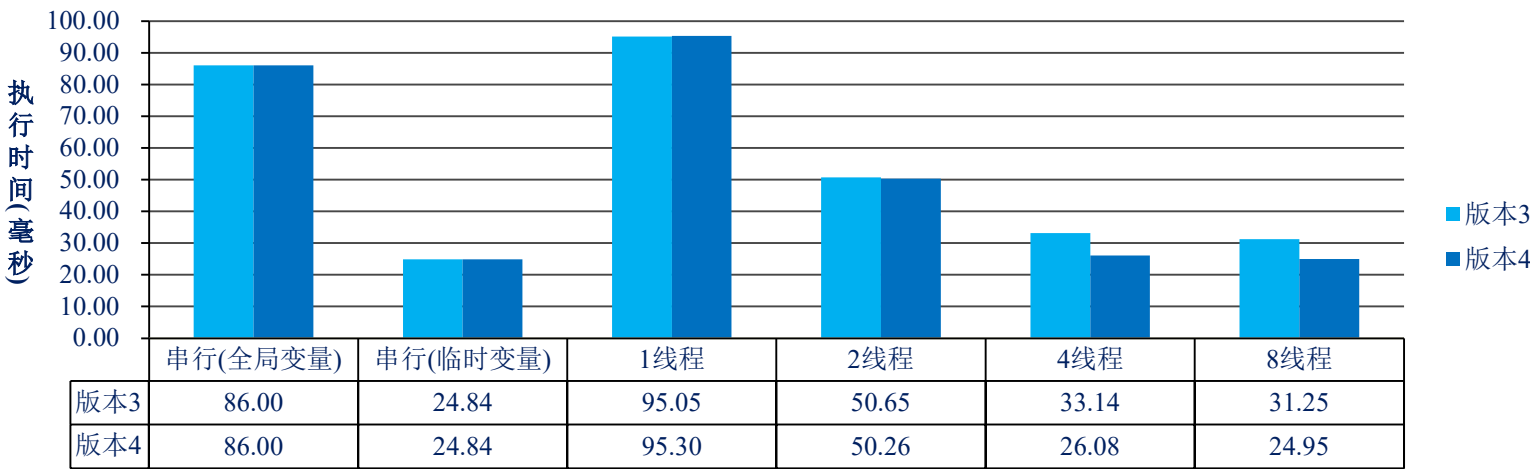
每个线程统计一部分

```
struct padded_int {
    int    value;
    char   padding[60];
} gPrivateCount[MAX_THREADS];
void * count10_thread(void * arg) {
    int myid = (int)arg;
    int len_per_thread = ARRAY_LEN / gThreadNum;
    int start = myid * len_per_thread;

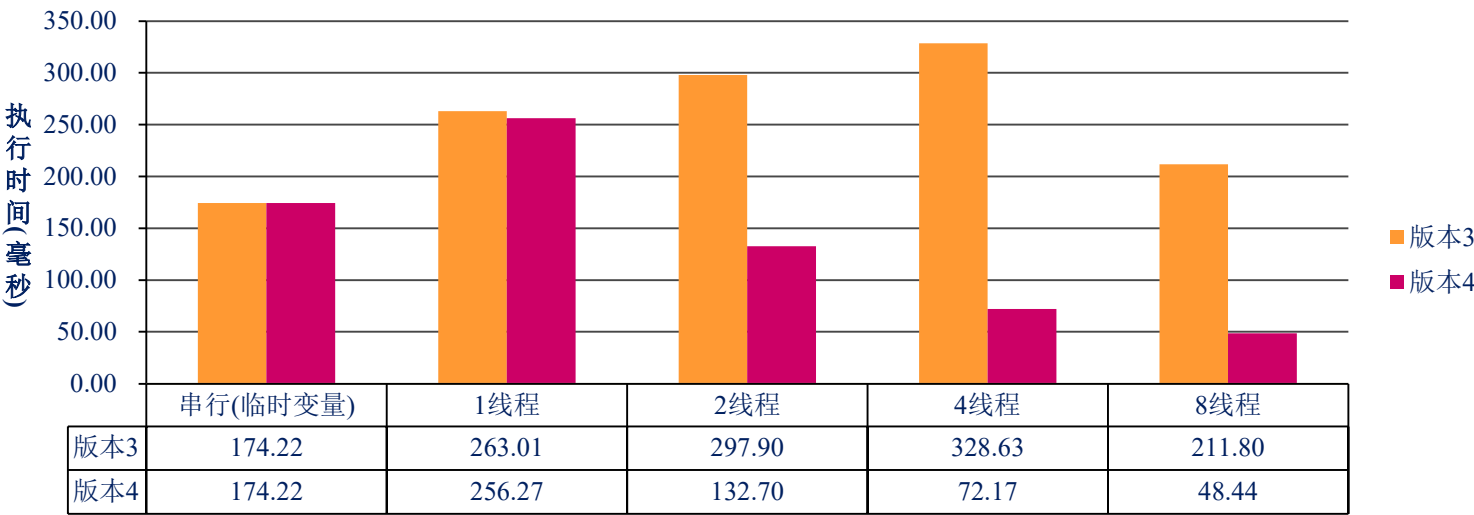
    gPrivateCount[myid].value = 0;
    for(i = start; i < start + len_per_thread; i++) {
        if(gArray[i] == 10) {
            gPrivateCount[myid].value++;
        }
    }
    pthread_mutex_lock(&lock);
    gCount += gPrivateCount[myid].value;
    pthread_mutex_unlock(&lock);
}
```

版本4的性能分析

版本3与版本4性能对比（Intel Core i7-950）



版本3与版本4性能对比（四个双核Intel Xeon）





如何改进4的基础上继续改进性能



策略 数据局部性

版本5:

创建多个线程: 与版本4相同

```
int count10() {  
    int i;  
    pthread_t tids[MAX_THREADS];  
  
    gCount = 0;  
    for(i = 0; i < gThreadNum; i++) {  
        pthread_create(&tids[i], NULL, count10_thread, (void*)i);  
    }  
    for(i = 0; i < gThreadNum; i++) {  
        pthread_join(tids[i], NULL);  
    }  
    return gCount;  
}
```

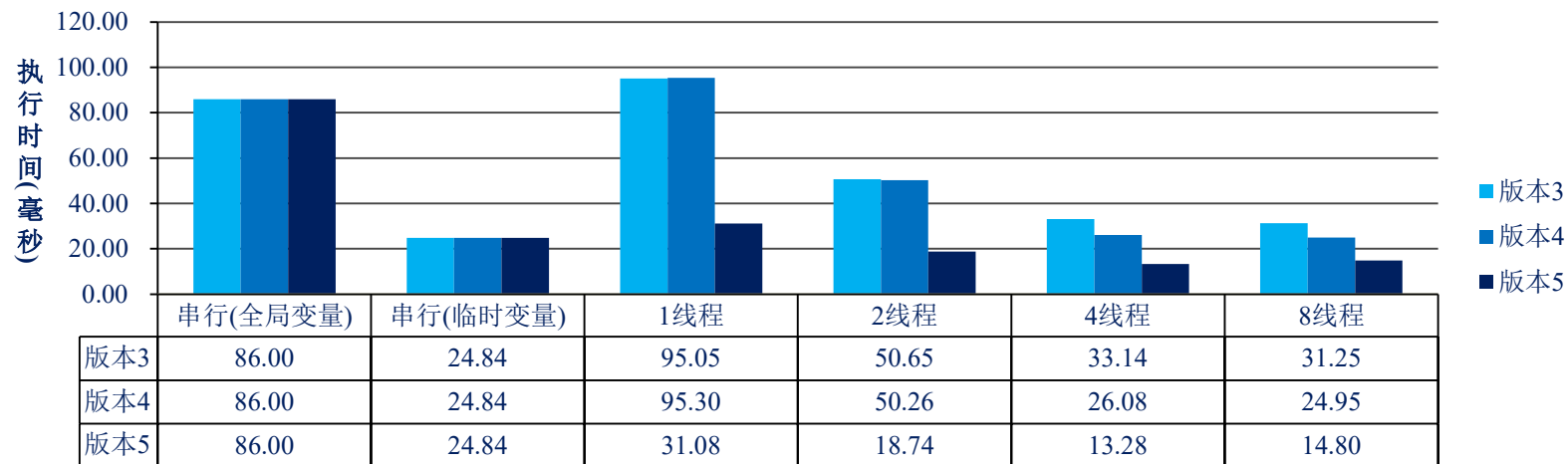
版本5:

每个线程统计一部分

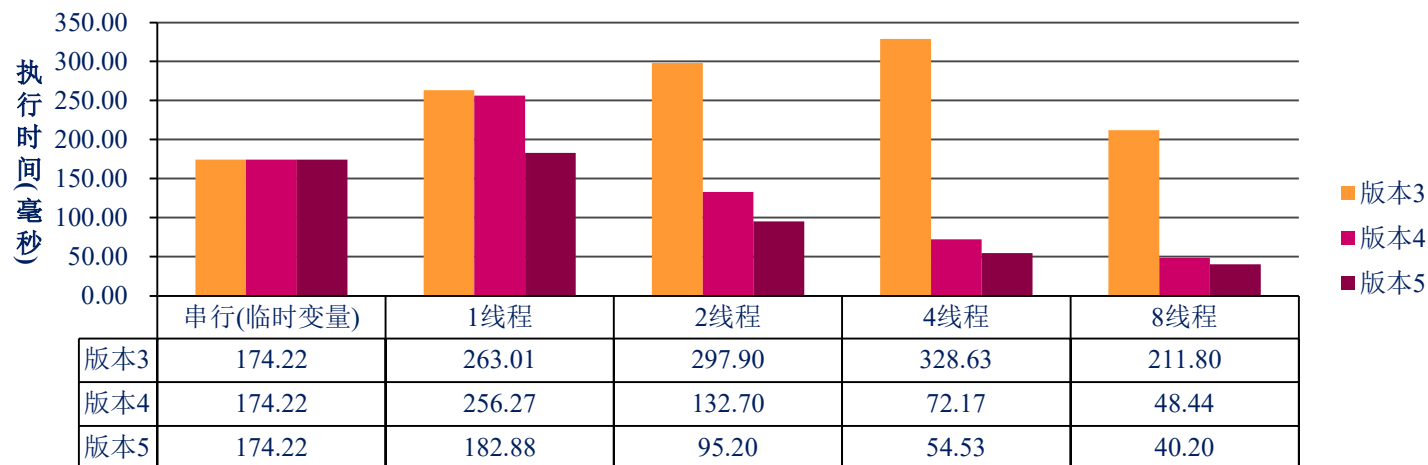
```
void * count10_thread(void * arg) {  
    int myid = (int)arg;  
    int i;  
    int len_per_thread = ARRAY_LEN / gThreadNum;  
    int start = myid * len_per_thread;  
  
    int tcount = 0;  
    for(i = start; i < start + len_per_thread; i++) {  
        if(gArray[i] == 10) {  
            tcount++;  
        }  
    }  
    pthread_mutex_lock(&lock);  
    gCount += tcount;  
    pthread_mutex_unlock(&lock);  
}
```

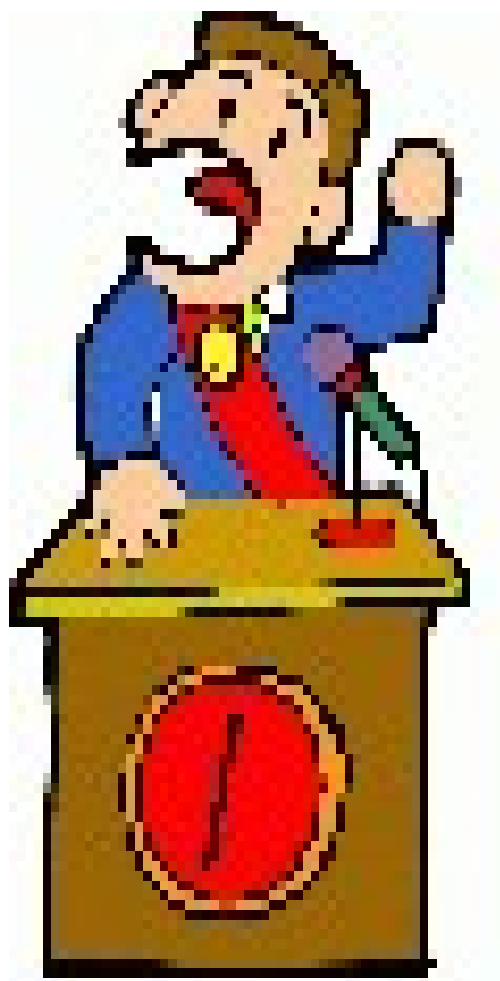
版本5的性能分析

版本3、4、5性能对比（Intel Core i7-950）



版本3、4、5性能对比（四个双核Intel Xeon）





本章内容小结

多线程编程

1. 理解线程

- ◆ 并行的函数、轻量级进程
- ◆ 共享地址空间
共享数据段、堆等

2. 线程操作

- ◆ 创建线程(pthread_create)
- ◆ 回收线程、分离线程
可结合的线程被回收之后才会释放占用的资源
可分离的线程终止运行, 占用资源自动释放
- ◆ 线程退出 (pthread_exit, pthread_cancel)

3. 线程同步

- ◆ 共享变量
- ◆ 互斥、信号量、条件变量、CAS

4. 线程信号处理

- ◆ 同步信号处理
- ◆ 异步信号处理

5. 线程常见问题及解决方法

6. 线程程序的性能分析