目录

[目录 1](#_Toc65492064)

[多线程解决什么问题： 2](#_Toc65492065)

[线程的概念 2](#_Toc65492066)

[线程ID 3](#_Toc65492067)

[1 多线程生命周期 4](#_Toc65492068)

[1.1 线程的创建 4](#_Toc65492069)

[1.2线程终止方法： 5](#_Toc65492070)

[1.3 join线程 6](#_Toc65492071)

[1.4线程detach 7](#_Toc65492072)

[1.5多线程的信号处理 8](#_Toc65492073)

[2 多线程同步 8](#_Toc65492074)

[2.1 互斥锁 8](#_Toc65492075)

[2.2 Condition Variable 11](#_Toc65492076)

[2.3 信号量Semaphone 14](#_Toc65492077)

[2.4其它线程间同步机制 16](#_Toc65492078)

[2.5 如何正确的加锁？ 17](#_Toc65492079)

[2.6 race condition的调试： 17](#_Toc65492080)

[3线程安全和可重入 23](#_Toc65492081)

[3.1线程安全和可重入函数的重要性 23](#_Toc65492082)

[3.3 非学究可重入与线程安全注意事项 27](#_Toc65492083)

[3.4 多线程情况下的信号处理 27](#_Toc65492084)

[3.4 典型的不安全的Linux库函数 31](#_Toc65492085)

[4. 多线程与栈 31](#_Toc65492086)

[4.1 线程的栈 31](#_Toc65492087)

[4.2栈的Guard page 34](#_Toc65492088)

[5. 如何设计多线程程序 36](#_Toc65492089)

[5.1 划分线程的典型原则 36](#_Toc65492090)

[5.2 典型的多线程模型 37](#_Toc65492091)

[5.3 Amdahl定律 40](#_Toc65492092)

[5.4 多线程与I/O 41](#_Toc65492093)

[6. 多线程调试与优化 44](#_Toc65492094)

## 多线程解决什么问题：

1. 线程生命周期问题；
2. 线程如何划分的问题(性能问题);
3. 线程如何通信的问题；

信号的行为是进程级别的；对任何线程发信号，线程组都会响应；

多线程通信开销(不涉及大量数据交互)，远低于多进程；

多进程比多线程好调试；

关系不亲密，适合用多进程模型；否则用多线程模型；

## 线程的概念

由于同一进程的多个线程共享同一地址空间，因此Text Segment、Data Segment都是共享的，如果定义一个函数，在各线程中都可以调用，如果定义一个全局变量，在各线程中都可以访问到，除此之外，**各线程还共享以下进程资源和环境**：

文件描述符表

每种信号的处理方式（SIG\_IGN、SIG\_DFL或者自定义的信号处理函数）

当前工作目录

用户id和组id

**但有些资源是每个线程各有一份的：**

线程id

上下文，包括各种寄存器的值、程序计数器和栈指针

栈空间

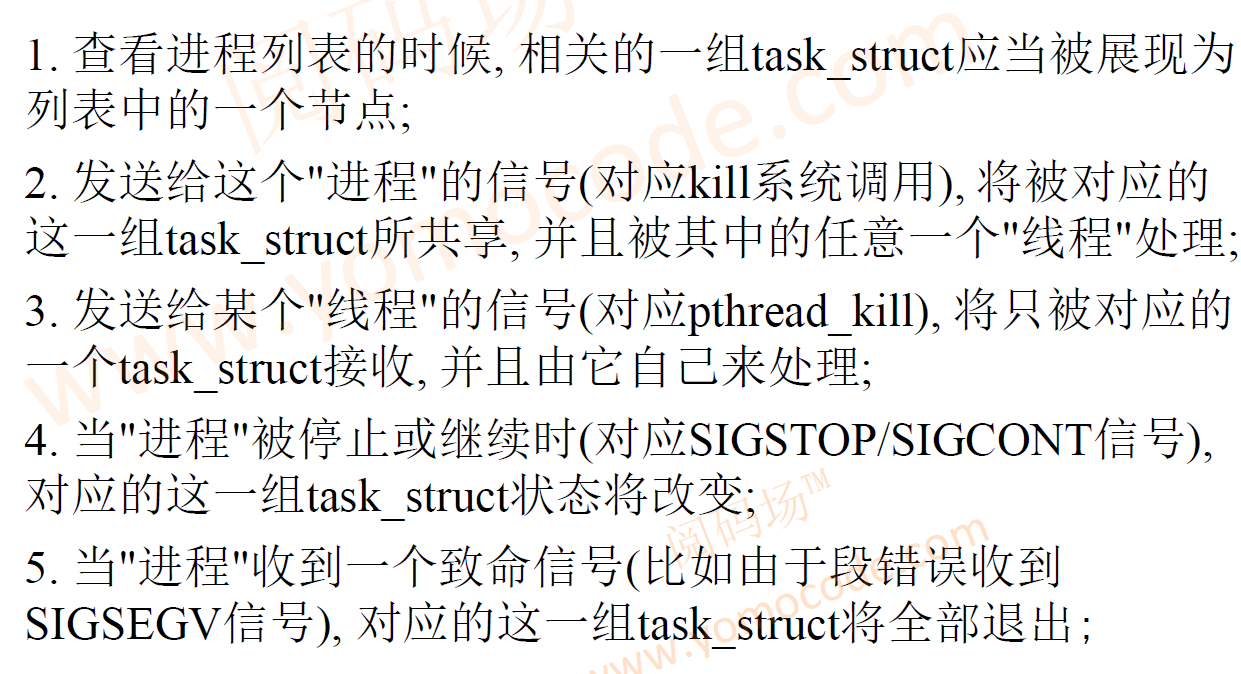
errno变量

信号屏蔽字

调度优先级

Linux使用的线程库是由POSIX标准定义的，称为POSIX thread或者pthread。线程函数位于libpthread共享库中，因此在编译时要加上-lpthread选项。

POSIX标准对线程的要求：



由上可见Linux内核通过clone创建线程并不能支持POSIX标准，Linux通过NPTL模型支持POSIX；

leon@pc:~$ getconf GNU\_LIBPTHREAD\_VERSION

NPTL 2.23

leon@pc:~$

在NPTL模型，同一个进程内的线程提供不同PID, 共用一个TGID；

Top命令看到的PID实际是TGID(等于主线程的PID)；

Top –H线程视角看到的是每个线程的PID(不同，内核存在的真正PID)；

### 线程ID

我们知道**进程id**的类型是pid\_t，每个进程的id在整个系统中是唯一的，调用getpid(2)可以获得当前进程的id，是一个正整数值。

**线程id**的类型是thread\_t，它只在当前进程中保证是唯一的，在不同的系统中thread\_t这个类型有不同的实现，它可能是一个整数值，也可能是一个结构体，也可能是一个地址，所以不能简单地当成整数用printf打印，调用pthread\_self(3)可以获得当前线程的id。

在Linux上，thread\_t类型是一个地址值，属于同一进程的多个线程调用getpid(2)可以得到相同的进程号，而调用pthread\_self(3)得到的线程号各不相同。

pthread\_self是在C库实现的数据结构(TCB)，与内核没关系；

## 1 多线程生命周期

### 1.1 线程的创建

#include <pthread.h>

int pthread\_create(pthread\_t \*restrict thread, const pthread\_attr\_t \*restrict attr, void \*(\*start\_routine)(void\*), void \*restrict arg);

返回值：成功返回0，失败返回错误号

在一个线程中调用pthread\_create()创建新的线程后，当前线程从pthread\_create()返回继续往下执行，而新的线程所执行的代码由我们传给pthread\_create的函数指针start\_routine决定。

start\_routine函数接收一个参数，是通过pthread\_create的arg参数传递给它的，该参数的类型为void \*，这个指针按什么类型解释由调用者自己定义。start\_routine的返回值类型也是void \*，这个指针的含义同样由调用者自己定义。start\_routine返回时，这个线程就退出了，其它线程可以调用pthread\_join得到start\_routine的返回值，类似于父进程调用wait(2)得到子进程的退出状态，稍后详细介绍pthread\_join。

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

pthread\_t ntid;

void printids(const char \*s)

{

pid\_t pid;

pthread\_t tid;

pid = getpid();

tid = pthread\_self();

printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid, (unsigned int)tid, (unsigned int)tid);

}

void \*thr\_fn(void \*arg)

{

printids(arg);

return NULL;

}

int main(void)

{

int err;

err = pthread\_create(&ntid, NULL, thr\_fn, "new thread: ");

if (err != 0)

{

fprintf(stderr, "can't create thread: %s\n", strerror(err));

exit(1);

}

printids("main thread:");

sleep(1);

return 0;

}

### 1.2线程终止方法：

(1).从线程函数return; 这个方法对主线程不适用，从main函数return，会调用exit();

(2)一个线程可以调用pthread\_cancel终止同一进程中的另一个线程。(会导致资源同步问题，在android中已经弃用)；

(3)线程可以调用pthread\_exit终止自己；

#include <pthread.h>

void pthread\_exit(void \*value\_ptr);

**整个进程退出**

(4)main函数返回；

(5)调用进程级别的API;exit(),\_exit();

(6)某一线程做了非常操作，引起segment fault；

value\_ptr是void \*类型，和线程函数返回值的用法一样，其它线程可以调用pthread\_join获得这个指针。

需要注意，pthread\_exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的，不能在线程函数的栈上分配，因为当其它线程得到这个返回指针时线程函数已经退出了。

如果任意一个线程调用了exit或\_exit，会调用exit\_group(), 是则整个进程的所有线程都终止；

从main函数return会调用exit,相当于调用exit\_group；

退出单个线程，用pthread\_exit();

exit()退出，调用exit\_group()，所有线程退出；

pthread\_exit()退出，调用exit()单个线程退出;

资源的单位是进程；

线程退出，可能引起memory leak等leak;

线程不pthread\_join可能引起leak;

### 1.3 join线程

#include <pthread.h>

int pthread\_join(pthread\_t thread, void \*\*value\_ptr);

返回值：成功返回0，失败返回错误号

调用该函数的线程将挂起等待，直到id为thread的线程终止。thread线程以不同的方法终止，通过pthread\_join得到的终止状态是不同的，总结如下：

(1)如果thread线程通过return返回，value\_ptr所指向的单元里存放的是thread线程函数的返回值。

(2)如果thread线程被别的线程调用pthread\_cancel异常终止掉，value\_ptr所指向的单元里存放的是常数PTHREAD\_CANCELED。

(3)如果thread线程是自己调用pthread\_exit终止的，value\_ptr所指向的单元存放的是传给pthread\_exit的参数。

如果对thread线程的终止状态不感兴趣，可以传NULL给value\_ptr参数。

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

void \*thr\_fn1(void \*arg)

{

printf("thread 1 returning\n");

return (void \*)1;

}

void \*thr\_fn2(void \*arg)

{

printf("thread 2 exiting\n");

pthread\_exit((void \*)2);

}

void \*thr\_fn3(void \*arg)

{

while(1)

{

printf("thread 3 writing\n");

sleep(1);

}

}

int main(void)

{

pthread\_t tid;

void \*tret;

pthread\_create(&tid, NULL, thr\_fn1, NULL);

pthread\_join(tid, &tret);

printf("thread 1 exit code %d\n", (int)tret);

pthread\_create(&tid, NULL, thr\_fn2, NULL);

pthread\_join(tid, &tret);

printf("thread 2 exit code %d\n", (int)tret);

pthread\_create(&tid, NULL, thr\_fn3, NULL);

sleep(3);

pthread\_cancel(tid);

pthread\_join(tid, &tret);

printf("thread 3 exit code %d\n", (int)tret);

return 0;

}

leon@pc:~/work/multi\_thread/day1$ ./a.out

thread 1 returning

thread 1 exit code 1

thread 2 exiting

thread 2 exit code 2

thread 3 writing

thread 3 writing

thread 3 writing

thread 3 exit code -1

可见在Linux的pthread库中常数PTHREAD\_CANCELED的值是-1。可以在头文件pthread.h中找到它的定义：

#define PTHREAD\_CANCELED ((void \*) -1)

pthread\_join()退出，释放线程资源；如果线程不做pthread\_join，线程栈不会释放，线程在堆上malloc的内存也不会释放(进程没有退出)；

### 1.4线程detach

如果线程被设置为detach状态，线程一旦终止就立刻回收它占用所有资源，

而不保留终止状态；

不能对一个已经处于detach状态的线程调用pthread\_join，这样的调用将返回EINVAL。对一个尚未detach的线程调用pthread\_join或pthread\_detach都可以把该线程置为detach状态，也就是说，不能对同一线程调用两次pthread\_join，或者如果已经对一个线程调用了pthread\_detach就不能再调用pthread\_join了。

Detach工程上很少用，只有调试线程用；

### 1.5多线程的信号处理

信号是进程级的概念，给某个进程发信号，该进程内所有线程都会响应；

对同一个信号，多个线程都设置处理函数，则最后一个设置生效；

POSIX要求可以对特定的线程发送信号，用pthread\_kill()实现；

线程组有共用的sispending， 每个线程有私有的sispending；pthread\_kill()对应私有sispending；

## 2 多线程同步

### 2.1 互斥锁

#### 1.pthread mutex

mutex用pthread\_mutex\_t类型的变量表示，可以这样初始化和销毁：

#include <pthread.h>

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

返回值：成功返回0，失败返回错误号。

pthread\_mutex\_init函数对Mutex做初始化，参数attr设定Mutex的属性，如果attr为NULL则表示缺省属性；

用pthread\_mutex\_init函数初始化的Mutex可以用pthread\_mutex\_destroy销毁。如果Mutex变量是静态分配的（全局变量或static变量），也可以用宏定义PTHREAD\_MUTEX\_INITIALIZER来初始化，相当于用pthread\_mutex\_init初始化并且attr参数为NULL。

Mutex的加锁和解锁操作可以用下列函数：

#include <pthread.h>

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

返回值：成功返回0，失败返回错误号。

一个线程可以调用pthread\_mutex\_lock获得Mutex，如果这时另一个线程已经调用pthread\_mutex\_lock获得了该Mutex，则当前线程需要挂起等待，直到另一个线程调用pthread\_mutex\_unlock释放Mutex，当前线程被唤醒，才能获得该Mutex并继续执行。

如果一个线程既想获得锁，又不想挂起等待，可以调用pthread\_mutex\_trylock，如果Mutex已经被另一个线程获得，这个函数会失败返回EBUSY，而不会使线程挂起等待。

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define NLOOP 5000

int counter; /\* incremented by threads \*/

pthread\_mutex\_t counter\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

void \*doit(void \*);

int main(int argc, char \*\*argv)

{

pthread\_t tidA, tidB;

pthread\_create(&tidA, NULL, doit, NULL);

pthread\_create(&tidB, NULL, doit, NULL);/\*wait for both threads to terminate \*/

pthread\_join(tidA, NULL);

pthread\_join(tidB, NULL);

return 0;

}

void \*doit(void \*vptr){

int i, val; /\* \* Each thread fetches, prints, and increments the counter NLOOP times. \* The value of the counter should increase monotonically. \*/

for (i = 0; i < NLOOP; i++)

{

pthread\_mutex\_lock(&counter\_mutex);

val = counter;

printf("%x: %d\n", (unsigned int)pthread\_self(), val + 1);

counter = val + 1;

pthread\_mutex\_unlock(&counter\_mutex);

}

return NULL;

}

这样运行结果就正常了，每次运行都能数到10000。

**Mutex的两个基本操作lock和unlock是如何实现的呢？**

lock和unlock的伪代码如下：

lock:

if(mutex > 0)

{

mutex = 0;

return 0;

}

else

挂起等待;

goto lock;

unlock:

mutex = 1;

唤醒等待Mutex的线程;

return 0

为了实现互斥锁操作，大多数体系结构都提供了swap或exchange指令，该指令的作用是把寄存器和内存单元的数据相交换，由于只有一条指令，保证了原子性，即使是多处理器平台，访问内存的总线周期也有先后，一个处理器上的交换指令执行时另一个处理器的交换指令只能等待总线周期。现在我们把lock和unlock的伪代码改一下（以x86的xchg指令为例）：

lock:

movb $0, %al

xchgb %al, mutex

if(al寄存器的内容 > 0)

{

return 0;

}

else 挂起等待;

goto lock;

unlock:

movb $1, mutex

唤醒等待Mutex的线程;

return 0;

unlock中的释放锁操作同样只用一条指令实现，以保证它的原子性。

**“挂起等待”和“唤醒等待线程”的操作如何实现？**

每个Mutex有一个等待队列，一个线程要在Mutex上挂起等待，首先在把自己加入等待队列中，然后置线程状态为睡眠，然后调用调度器函数切换到别的线程。

一个线程要唤醒等待队列中的其它线程，只需从等待队列中取出一项，把它的状态从睡眠改为就绪，加入就绪队列，那么下次调度器函数执行时就有可能切换到被唤醒的线程。

写程序时应该尽量避免同时获得多个锁，如果一定有必要这么做，则有一个原则：如果所有线程在需要多个锁时都按相同的先后顺序（常见的是按Mutex变量的地址顺序）获得锁，则不会出现死锁。

mutex等待时，涉及线程切换，spin\_lock不会做线程切换，就在原地等；

#### 2.用户空间spin\_lock:

#include <pthread.h>

int pthread\_spin\_lock(pthread\_spinlock\_t \* lock);

int pthread\_spin\_trylock(pthread\_spinlock\_t \* lock);

int pthread\_spin\_unlock(pthread\_spinlock\_t \* lock);

使用场景，

锁住的区间短；

区间经常发生；

区间可能成为性能瓶颈；

锁住大区间可能导致很高的CPU利用率和性能下降；

在较短区间，用spin\_lock效率更高；

spin\_lock自旋会消耗CPU；

内核的spin\_lock会屏蔽线程抢占，其主要作用在多核场景；

用户态的spin\_lock不会屏蔽抢占；

### 2.2 Condition Variable

线程间的同步还有这样一种情况：线程A需要等某个条件成立才能继续往下执行，现在这个条件不成立，线程A就阻塞等待，而线程B在执行过程中使这个条件成立了，就唤醒线程A继续执行。

在pthread库中通过条件变量（Condition Variable）来阻塞等待一个条件，或者唤醒等待这个条件的线程。Condition Variable用pthread\_cond\_t类型的变量表示，可以这样初始化和销毁：

#include <pthread.h>

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond, const pthread\_condattr\_t \*restrict attr);

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

返回值：成功返回0，失败返回错误号。

和Mutex的初始化和销毁类似，pthread\_cond\_init函数初始化一个Condition Variable，attr参数为NULL则表示缺省属性，pthread\_cond\_destroy函数销毁一个Condition Variable。如果Condition Variable是静态分配的，也可以用宏定义PTHEAD\_COND\_INITIALIZER初始化，相当于用pthread\_cond\_init函数初始化并且attr参数为NULL。Condition Variable的操作可以用下列函数：

#include <pthread.h>

int pthread\_cond\_timedwait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex, const struct timespec \*restrict abstime);

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex);

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

返回值：成功返回0，失败返回错误号。

一个Condition Variable总是和一个Mutex搭配使用的。一个线程可以调用pthread\_cond\_wait在一个Condition Variable上阻塞等待，这个函数做以下三步操作：

1. 释放Mutex

2. 阻塞等待

3. 当被唤醒时，重新获得Mutex并返回

pthread\_cond\_timedwait函数还有一个额外的参数可以设定等待超时，如果到达了abstime所指定的时刻仍然没有别的线程来唤醒当前线程，就返回ETIMEDOUT。

一个线程可以调用pthread\_cond\_signal唤醒在某个Condition Variable上等待的另一个线程，也可以调用pthread\_cond\_broadcast唤醒在这个Condition Variable上等待的所有线程。

下面的程序演示了一个生产者-消费者的例子，生产者生产一个结构体串在链表的表头上，消费者从表头取走结构体。

#include <stdlib.h>

#include <pthread.h>

#include <stdio.h>

struct msg {

struct msg \*next;

int num;

};

struct msg \*head;

pthread\_cond\_t has\_product = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;

void \*consumer(void \*p)

{

struct msg \*mp;

for (;;) {

pthread\_mutex\_lock(&lock);

while (head == NULL)

pthread\_cond\_wait(&has\_product, &lock);

mp = head;

head = mp->next;

pthread\_mutex\_unlock(&lock);

printf("Consume %d\n", mp->num);

free(mp);

sleep(rand() % 5);

}

}

void \*producer(void \*p)

{

struct msg \*mp;

for (;;) {

mp = malloc(sizeof(struct msg));

mp->num = rand() % 1000 + 1;

printf("Produce %d\n", mp->num);

pthread\_mutex\_lock(&lock);

if (head == NULL) {

head = mp;

mp->next = NULL;

} else {

for (struct msg \*it=head;it!=NULL;){

if (it->next==NULL) {

it->next = mp;

mp->next=NULL;

break;

} else {

it = it->next;

}

}

}

pthread\_mutex\_unlock(&lock);

pthread\_cond\_signal(&has\_product);

sleep(rand() % 3);

}

}

int main(int argc, char \*argv[])

{

pthread\_t pid, cid;

srand(time(NULL));

pthread\_create(&pid, NULL, producer, NULL);

pthread\_create(&cid, NULL, consumer, NULL);

pthread\_join(pid, NULL);

pthread\_join(cid, NULL);

return 0;

}

执行结果如下：

FIFO

### 2.3 信号量Semaphone

信号量（Semaphore）和Mutex类似，表示可用资源的数量，和Mutex不同的是这个数量可以大于1。

本节介绍的是POSIX semaphore库函数，详见sem\_overview(7)，这种信号量不仅可用于同一进程的线程间同步，也可用于不同进程间的同步。

#include <semaphore.h>

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

int sem\_wait(sem\_t \*sem);

int sem\_trywait(sem\_t \*sem);

int sem\_post(sem\_t \* sem);

int sem\_destroy(sem\_t \* sem);

semaphore变量的类型为sem\_t，sem\_init()初始化一个semaphore变量，value参数表示可用资源的数量，pshared参数为0表示信号量用于同一进程的线程间同步，这里只介绍这种情况。在用完semaphore变量之后应该调用sem\_destroy()释放与semaphore相关的资源。

调用sem\_wait()可以获得资源，使semaphore的值减1，如果调用sem\_wait()时semaphore的值已经是0，则挂起等待。如果不希望挂起等待，可以调用sem\_trywait()。

调用sem\_post()可以释放资源，使semaphore的值加1，同时唤醒挂起等待的线程。

上一节生产者－消费者的例子是基于链表的，其空间可以动态分配，现在基于固定大小的环形队列重写这个程序：

#include <stdlib.h>

#include <pthread.h>

#include <stdio.h>

#include <semaphore.h>

#define NUM 5

int queue[NUM];

sem\_t blank\_number, product\_number;

void \*producer(void \*arg)

{

int p = 0;

while (1) {

sem\_wait(&blank\_number);

queue[p] = rand() % 1000 + 1;

printf("Produce %d\n", queue[p]);

sem\_post(&product\_number);

p = (p+1)%NUM;

sleep(rand()%5);

}

}

void \*consumer(void \*arg)

{

int c = 0;

while (1) {

sem\_wait(&product\_number);

printf("Consume %d\n", queue[c]); queue[c] = 0;

sem\_post(&blank\_number);

c = (c+1)%NUM;

sleep(rand()%5);

}

}

int main(int argc, char \*argv[])

{

pthread\_t pid, cid;

sem\_init(&blank\_number, 0, NUM);

sem\_init(&product\_number, 0, 0);

pthread\_create(&pid, NULL, producer, NULL);

pthread\_create(&cid, NULL, consumer, NULL);

pthread\_join(pid, NULL);

pthread\_join(cid, NULL);

sem\_destroy(&blank\_number);

sem\_destroy(&product\_number);

return 0;

}

### 2.4其它线程间同步机制

读者写者锁（Reader-Writer Lock）的概念，Reader之间并不互斥，可以同时读共享数据，而Writer是独占的（exclusive），在Writer修改数据时其它Reader或Writer不能访问数据，可见Reader-WriterLock比Mutex具有更好的并发性。

用挂起等待的方式解决访问冲突不见得是最好的办法，因为这样毕竟会影响系统的并发性，在某些情况下解决访问冲突的问题可以尽量避免挂起某个线程，例如Linux内核的Seqlock、RCU（read-copy-update）等机制。

4. 编程练习

哲学家就餐问题。这是由计算机科学家Dijkstra提出的经典死锁场景。

原版的故事里有五个哲学家(不过我们写的程序可以有N个哲学家)，这些哲学家们只做两件事－－思考和吃饭，他们思考的时候不需要任何共享资源，但是吃饭的时候就必须使用餐具，而餐桌上的餐具是有限的，原版的故事里，餐具是叉子，吃饭的时候要用两把叉子把面条从碗里捞出来。很显然把叉子换成筷子会更合理，所以：一个哲学家需要两根筷子才能吃饭。

现在引入问题的关键：这些哲学家很穷，只买得起五根筷子。他们坐成一圈，两个人的中间放一根筷子。哲学家吃饭的时候必须同时得到左手边和右手边的筷子。如果他身边的任何一位正在使用筷子，那他只有等着。

假设哲学家的编号是A、B、C、D、E，筷子编号是1、2、3、4、5，哲学家和筷子围成一圈如下图所示：

图 35.2. 哲学家问题

每个哲学家都是一个单独的线程，每个线程循环做以下动作：思考rand()%10秒，然后先拿左手边的筷子再拿右手边的筷子（筷子这种资源可以用mutex表示），有任何一边拿不到就一直等着，全拿到就吃饭rand()%10秒，然后放下筷子。

编写程序仿真哲学家就餐的场景：

Philosopher A fetches chopstick 5

Philosopher B fetches chopstick 1

Philosopher B fetches chopstick 2

Philosopher D fetches chopstick 3

Philosopher B releases chopsticks 1 2

Philosopher A fetches chopstick 1

Philosopher C fetches chopstick 2

Philosopher A releases chopsticks 5 1......

分析一下，这个过程有没有可能产生死锁？调用usleep(3)函数可以实现微秒级的延时，试着用usleep(3)加快仿真的速度，看能不能观察到死锁现象。然后修改上述算法避免产生死锁。

### 2.5 如何正确的加锁？

要考虑线程安全性，可重入性

**加锁三要素：**

(1)同一把锁;

(2)语义整体(数据库中的事务的概念);

(2)粒度最小;增加并发性;

对于要加多个锁的情况，必须严格按照同样顺序加锁，这样可以避免死锁问题；

### 2.6 race condition的调试：

自动检测竞态的工具，ThreadSanitizer和helgrind

1.ThreadSanitizer引入编译选项-fsanitize=thread

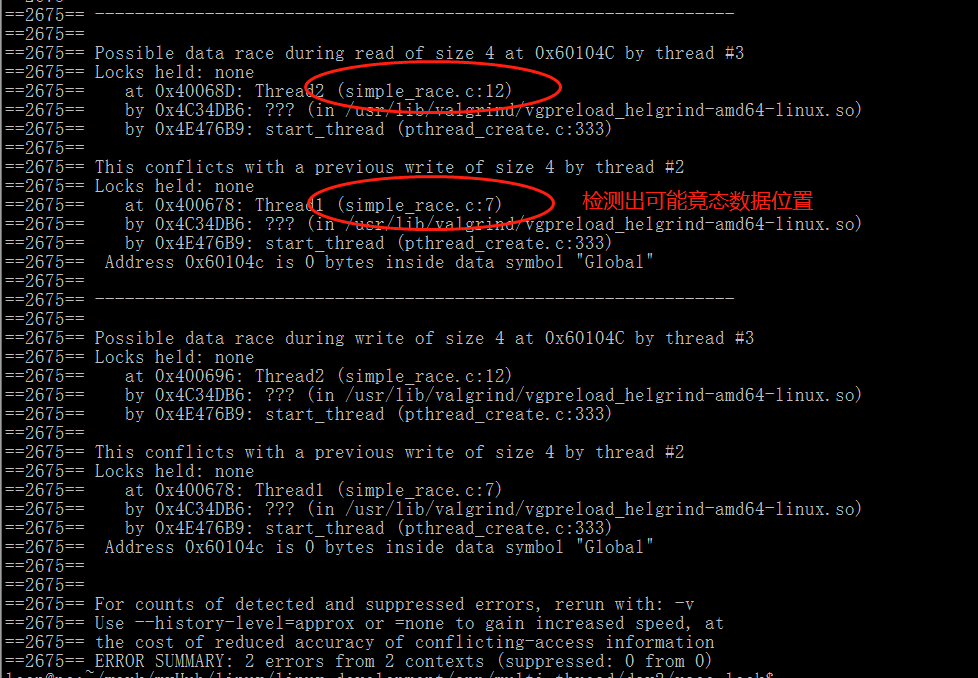
gcc simple\_race.c -fsanitize=thread -g -pthread

2. helgrind, old gcc version:

不依赖gcc fsanitize

gcc simple\_race.c -g -lpthread

valgrind --tool=helgrind ./a.out



http://github.com/google/sanitizers/wiki/ThreadSanitizeCppManual

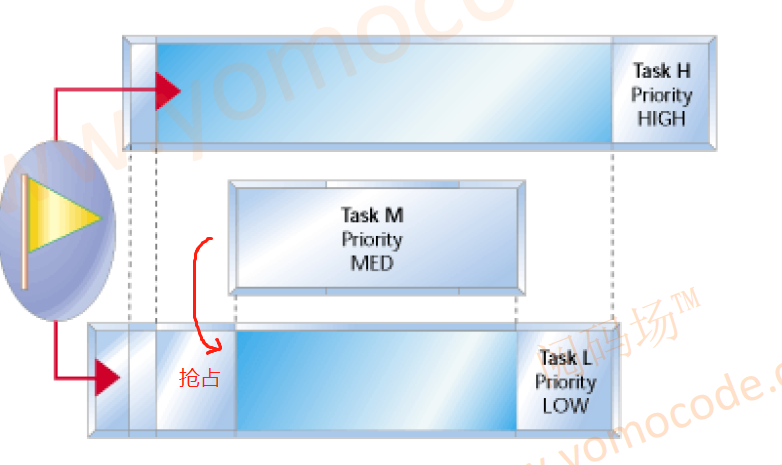
用sanitizers内存消耗增长5-10倍，执行时间增加2-20倍；

https://www.valgrind.org/docs/manual/hg-manual.html  
**2.7 pthread\_mutex与优先级继承**

高优先级线程等低优先级线程释放锁的过程中，中等优先级线程打断低优先级线程；

导致高优先级线程很大延迟；

设置优先级反转属性，低优先级就提搞到高优先级一样的优先级；



Int pthread\_mutexattr\_setprotocol(pthread\_mutexattr\_t \*attr, int protocol);

**PTHREAD\_PRIO\_NONE:**

**PTHREAD\_PRIO\_INHERIT: 优先级继承**

#define \_GNU\_SOURCE /\* See feature\_test\_macros(7) \*/

#include <sched.h>

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <time.h>

#include <stdlib.h>

#include <unistd.h>

#include <sched.h>

#define USE\_PRI\_INHERIT /\* high-priority task only wait LOW\_SPIN time \*/

#if 1

#undef USE\_PRI\_INHERIT /\* high-priority task wait LOW+MID\_SPIN time \*/

#endif

/\*

A simple demonstration of a 3-thread priority inversion that can

be improved with priority inheritance.

Written by Kevin Dankwardt, k@kcomputing.com

\*/

sem\_t low\_go, mid\_go, high\_go, high\_ready, mid\_ready;

pthread\_mutex\_t shared\_mutex;

const int minutes = 60;

const int seconds = 1;

pthread\_mutexattr\_t mutex\_attr;

#define LOW\_SPIN 2

#define MID\_SPIN 5

int gettime ()

{

return time(NULL);

}

void spin\_for (int n)

{

int now = gettime();

int counter=0;

while ( ( gettime() - now) < n)

{ counter++;

//if ((counter % 1000000) == 0) printf("gettime()-now = %d n=%d\n",gettime()-now,n);

}

//printf("done spinning for %d seconds\n",n);

}

void \*low (void \*n)

{

int now = gettime();

struct sched\_param the\_priority;

the\_priority.sched\_priority = 1;

pthread\_setschedparam(pthread\_self(), SCHED\_FIFO, &the\_priority);

sem\_wait(&low\_go);

pthread\_mutex\_lock(&shared\_mutex);

sem\_wait(&mid\_ready);

sem\_wait(&high\_ready);

sem\_post(&high\_go);

sem\_post(&mid\_go);

spin\_for(LOW\_SPIN\*seconds);

pthread\_mutex\_unlock(&shared\_mutex);

printf ("low took %d seconds wanted about %d (critical section + mid time)\n",gettime() - now,LOW\_SPIN+MID\_SPIN);

return NULL;

}

void \*mid (void \*n)

{

struct sched\_param the\_priority;

int now;

the\_priority.sched\_priority = 25;

pthread\_setschedparam(pthread\_self(), SCHED\_FIFO, &the\_priority);

sem\_post(&mid\_ready);

sem\_wait(&mid\_go);

now = gettime();

spin\_for(MID\_SPIN\*seconds);

printf ("mid took %d seconds wanted about %d\n",gettime() - now,MID\_SPIN);

return NULL;

}

void \*high (void \*n)

{

int now ;

struct sched\_param the\_priority;

the\_priority.sched\_priority = 50;

pthread\_setschedparam(pthread\_self(), SCHED\_FIFO, &the\_priority);

sem\_post(&high\_ready);

sem\_wait(&high\_go);

now=gettime();

pthread\_mutex\_lock(&shared\_mutex);

pthread\_mutex\_unlock(&shared\_mutex);

printf ("high took %d seconds wanted about %d (low critical section)\n",gettime() - now,LOW\_SPIN);

return NULL;

}

int main ()

{

pthread\_t tid1, tid2, tid3;

cpu\_set\_t thecpus;

CPU\_ZERO(&thecpus);

CPU\_SET(0, &thecpus);

if (sched\_setaffinity(getpid(), sizeof(cpu\_set\_t), &thecpus)< 0)

perror("set affinity");

if (pthread\_mutexattr\_init(&mutex\_attr))

{perror("mutex init"); exit(1);}

#if defined(\_POSIX\_THREAD\_PRIO\_INHERIT) && \_POSIX\_THREAD\_PRIO\_INHERIT != -1 && defined(USE\_PRI\_INHERIT)

printf("Using priority inheritance\n");

if (pthread\_mutexattr\_setprotocol(&mutex\_attr, PTHREAD\_PRIO\_INHERIT))

{perror("mutex init"); exit(1);}

#else

printf("Not Using priority inheritance\n");

#endif

if (pthread\_mutex\_init(&shared\_mutex, &mutex\_attr))

{perror("mutex init"); exit(1);}

// all initialized to zero. Must wait on a post

sem\_init (&low\_go,0,0);

sem\_init (&mid\_go,0,0);

sem\_init (&high\_go,0,0);

sem\_init (&high\_ready,0,0);

sem\_init (&mid\_ready,0,0);

pthread\_create(&tid1, NULL, low, NULL);

pthread\_create(&tid2, NULL, mid, NULL);

pthread\_create(&tid3, NULL, high, NULL);

sem\_post(&low\_go);

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

pthread\_join(tid3, NULL);

return 0;

}

## 3线程安全和可重入

### 3.1线程安全和可重入函数的重要性

安全访问竟态数据，涉及两个概念

**线程安全：**

不访问全局资源的函数；或者访问全局资源，但是加mutex的函数；

比如常见的malloc, free, printf等函数，会访问全局资源，其内部都添加了锁保护，因此是线程安全的；

特点，多线程对竟态的资源访问，加锁即可；

**信号安全：**

不访问全局资源的函数；访问全局资源，但是做保存和恢复的函数；

信号函数，属于线程内资源，不能加锁(可能引起死锁)；

信号函数打断线程后，信号函数会执行完，才会重新调度，因此只要对竟态资源添加保存和恢复，不会影响原线程数据；

特点，不能用mutex等锁；

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>

#include <ctype.h>

#include <sys/types.h>

char \*strtoupper(char \*string)

{

static char buffer[1000];

int index;

for (index = 0; string[index]; index++)

buffer[index] = toupper(string[index]);

buffer[index] = 0;

return buffer;

}

void \*thread\_fun(void \*param)

{

while (1) {

usleep(100);

printf("%s\n", strtoupper((char \*)param));

}

}

int main(void)

{

pthread\_t tid1, tid2;

int ret;

printf("main pid:%d, tid:%lu\n", getpid(), pthread\_self());

ret = pthread\_create(&tid1, NULL, thread\_fun, "hello world");

if (ret == -1) {

perror("cannot create new thread");

return 1;

}

ret = pthread\_create(&tid2, NULL, thread\_fun, "world hello");

if (ret == -1) {

perror("cannot create new thread");

return 1;

}

if (pthread\_join(tid1, NULL) != 0) {

perror("call pthread\_join function fail");

return 1;

}

if (pthread\_join(tid2, NULL) != 0) {

perror("call pthread\_join function fail");

return 1;

}

return 0;

}

**非学究的概括，可重入函数要满足两个条件**：

(1)函数是线程安全的；

(2)函数是可中断的(对于Linux, 异步的信号)，执行了中断处理后，再回来继续执行函数，结果仍然正确；

线程安全和信号安全的案例

<https://deadbeef.me/2017/09/reentrant-threadsafe>

线程安全，信号不安全案例：

\_\_thread int test = 123;

#include <threads.h>

// `t` is now local to each thread

thread\_local int t;//线程级全局变量，线程安全，但可以被信号访问

void swap(int \*x, int \*y) {

t = \*x;

\*x = \*y;

// `my\_func()` could be called here

\*y = t;

}

void my\_func() {

int x = 1, y = 2;

swap(&x, &y);

}

**信号安全，线程不安全案例**

int t;

//t可以恢复(信号函数执行完，才继续执行线程)，

void swap(int \*x, int \*y) {

int s;

// save global variable

s = t;

t = \*x;

\*x = \*y;

// `my\_func()` could be called here

\*y = t;

// restore global variable

t = s;

}

void my\_func() {

int x = 1, y = 2;

swap(&x, &y);

}

#include <signal.h>

#include <stdio.h>

struct two\_long { long a, b; } data;

void signal\_handler(int signum){

printf ("%d, %d\n", data.a, data.b);

alarm (1);

}

int main (void){

static struct two\_long zeros = { 0, 0 }, ones = { 1, 1 };

signal (SIGALRM, signal\_handler);

data = zeros;

alarm (1);

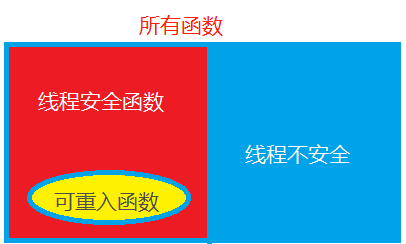
while (1)

{data = zeros; data = ones;}

}

线程不安全的通常不可重入，可重入的通常线程安全(极少例外)；

按经验可以做一下归类，



**典型的程序设计满足下面条件之一的多数是不可重入函数：**  
(1)使用了静态[数据结构](http://lib.csdn.net/base/31);  
(2)调用了malloc或free; 申请的内存在共同的堆里；  
(3)调用了标准I/O函数;标准io库很多实现都以不可重入的方式使用全局数据结构。  
(4)进行了浮点运算.许多的处理器/编译器中，浮点一般都是不可重入的 (浮点运算大多使用协处理器或者软件模拟来实现。)

### 3.3 非学究可重入与线程安全注意事项

如果一个函数中用到了全局或静态变量（const除外），那么它不是线程安全的，也不是可重入的；

如果我们对它加以改进，在访问全局或静态变量时使用互斥量或信号量等方式加锁，则可以使它变成线程安全的，但此时它仍然是不可重入的，因为**通常加锁方式是针对不同线程的访问**，而对同一线程可能出现问题(比如信号里调用加锁函数，可能出现死锁)；

如果将函数中的全局或静态变量去掉，改成函数参数等其他形式，则有可能使函数变成既线程安全，又可重入。

### 3.4 多线程情况下的信号处理

#### 方法1：替换掉不可重入函数

Signal\_handler()

{

Printf🡪write

}

打印，但是不会涉及线程安全问题；

#### 方法2：将信号的异步问题，转为同步化

在主线程将信号屏蔽掉，则其他同组线程也会屏蔽掉对应信号；

然后专门创建一个子线程，重新设置该信号，并且用sigwaitinfo函数，通过同步等待的方式获取信号，然后做信号处理

#include <signal.h>

#include <stdio.h>

struct two\_long { long a, b; } data;

void signal\_handler(int signum){

printf ("%d, %d\n", data.a, data.b);

alarm (1);

}

int main (void){

static struct two\_long zeros = { 0, 0 }, ones = { 1, 1 };

signal (SIGALRM, signal\_handler);

data = zeros;

alarm (1);

while (1)

{data = zeros; data = ones;}

}

leon@pc:~/work/multi\_thread/day3$ cat signalsync.c

#include <signal.h>

#include <errno.h>

#include <pthread.h>

#include <unistd.h>

#include <sys/types.h>

void sig\_handler(int signum)

{

static int j = 0;

static int k = 0;

pthread\_t sig\_ppid = pthread\_self();

// used to show which thread the signal is handled in.

if (signum == SIGUSR1) {

printf("thread %d, receive SIGUSR1 No. %d\n", sig\_ppid, j);

j++;

//SIGRTMIN should not be considered constants from userland,

//there is compile error when use switch case

} else if (signum == SIGRTMIN) {

printf("thread %d, receive SIGRTMIN No. %d\n", sig\_ppid, k);

k++;

}

}

void\* worker\_thread()

{

pthread\_t ppid = pthread\_self();

pthread\_detach(ppid);

while (1) {

printf("I'm thread %d, I'm alive\n", ppid);

sleep(10);

}

}

void\* sigmgr\_thread()

{

sigset\_t waitset, oset;

siginfo\_t info;

int rc;

pthread\_t ppid = pthread\_self();

pthread\_detach(ppid);

sigemptyset(&waitset);

sigaddset(&waitset, SIGRTMIN);

sigaddset(&waitset, SIGUSR1);

while (1) {

rc = sigwaitinfo(&waitset, &info);

if (rc != -1) {

printf("sigwaitinfo() fetch the signal - %d\n", rc);

sig\_handler(info.si\_signo);

} else {

printf("sigwaitinfo() returned err: %d; %s\n", errno, strerror(errno));

}

}

}

int main()

{

sigset\_t bset, oset;

int i;

pid\_t pid = getpid();

pthread\_t ppid;

// Block SIGRTMIN and SIGUSR1 which will be handled in

//dedicated thread sigmgr\_thread()

// Newly created threads will inherit the pthread mask from its creator

sigemptyset(&bset);

sigaddset(&bset, SIGRTMIN);

sigaddset(&bset, SIGUSR1);

//A new thread inherits a copy of its creator's signal mask.

if (pthread\_sigmask(SIG\_BLOCK, &bset, &oset) != 0)

printf("!! Set pthread mask failed\n");

// Create the dedicated thread sigmgr\_thread() which will handle

// SIGUSR1 and SIGRTMIN synchronously

pthread\_create(&ppid, NULL, sigmgr\_thread, NULL);

// Create 5 worker threads, which will inherit the thread mask of

// the creator main thread

for (i = 0; i < 5; i++) {

pthread\_create(&ppid, NULL, worker\_thread, NULL);

}

// send out 50 SIGUSR1 and SIGRTMIN signals

for (i = 0; i < 50; i++) {

kill(pid, SIGUSR1);

printf("main thread, send SIGUSR1 No. %d\n", i);

kill(pid, SIGRTMIN);

printf("main thread, send SIGRTMIN No. %d\n", i);

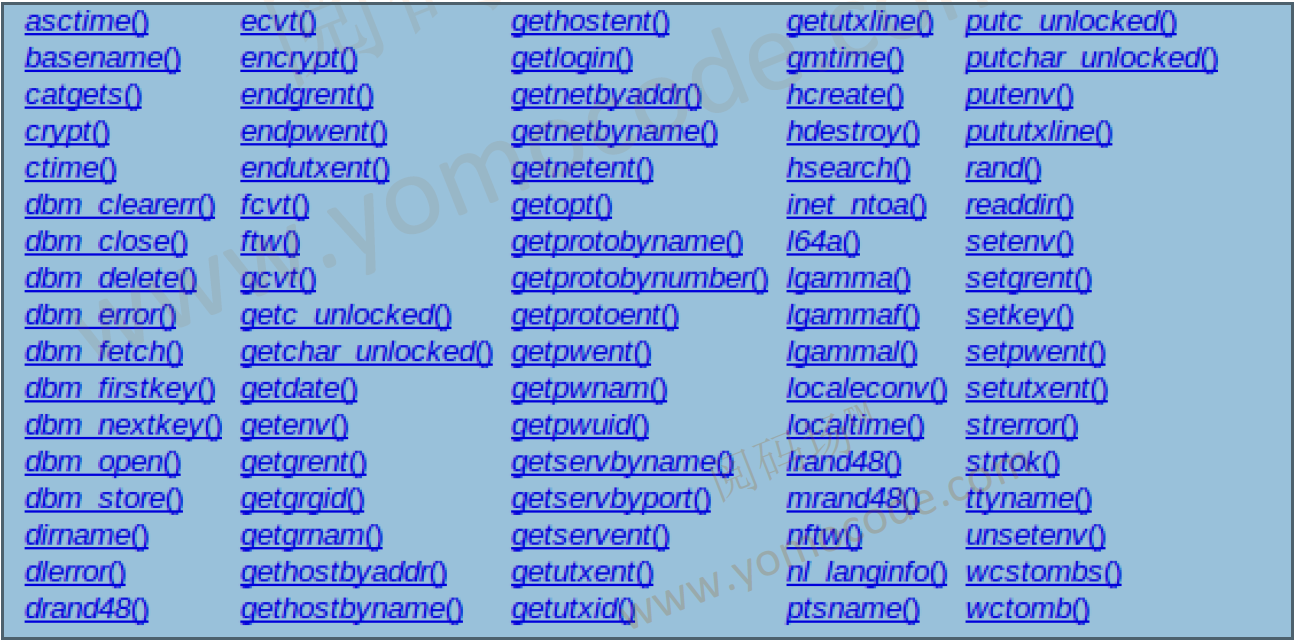
sleep(10);

}

exit (0);

}

### 3.4 典型的不安全的Linux库函数

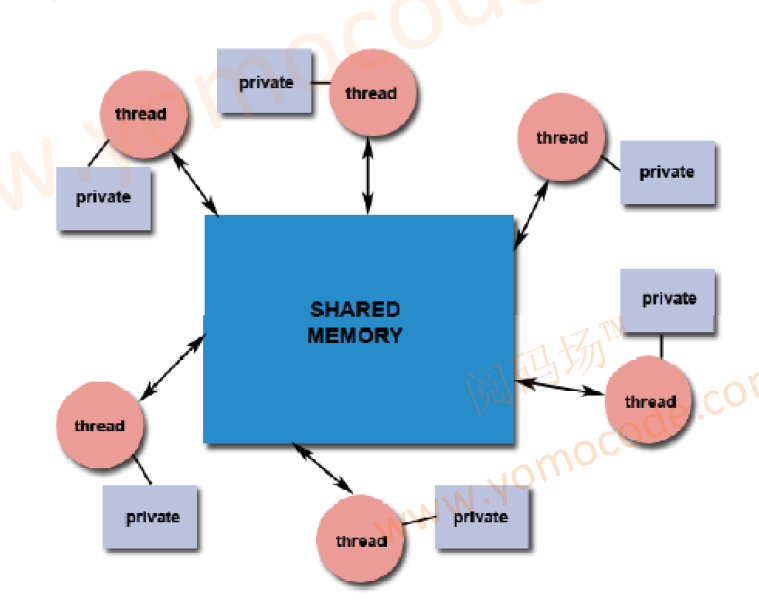


## 4. 多线程与栈

### 4.1 线程的栈

函数调用的回溯，保存在栈中；

函数临时变量，参数，保存在栈中；



**线程栈的双重特性：**

**从用法看**：每个线程一个栈；

**从进程资源管理角度**：所有线程的栈都属于整个进程的内存资源；

在一个线程访问另一个线程的栈，是合法的，Linux不会报错；但从用法看是错的；

#include <limits.h>

#include <pthread.h>

#include <string.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

static char \*p;//bss->0

static void \*thread\_routine(void \*arg)

{

char buf[10] = { 0 };

p = buf;

while (1) {

printf("buf is %s\n", buf);

sleep(1);

}

return NULL;

}

int main(int argc, char \*argv[])

{

pthread\_t thread\_id1;

pthread\_attr\_t thread\_attr;

size\_t stack\_size, guard\_size;

int status;

status = pthread\_attr\_init(&thread\_attr);

if (status != 0)

exit(-1);

status = pthread\_create(&thread\_id1, &thread\_attr, thread\_routine, NULL);

if (status != 0)

exit(-1);

int i = 0;

while (1) {

if (p) {

if (i++ % 2 == 0)

strcpy(p, "helloworld1");

else

strcpy(p, "world");

}

sleep(4);

}

pthread\_join(thread\_id1, NULL);

return 0;

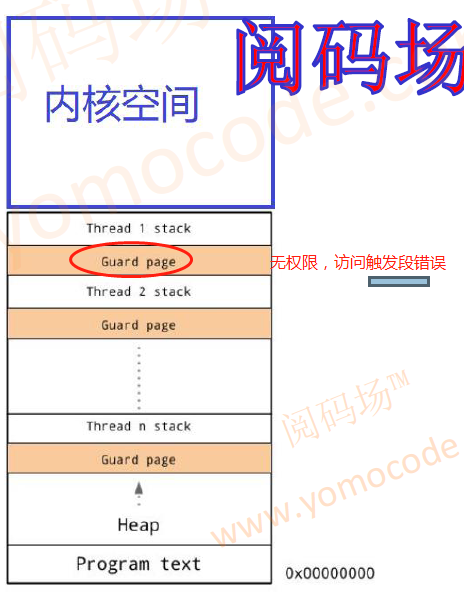
}  
4.2 guard page

越界sanitize可以检测，但是正常访问另一个线程地址，无法检测；

gcc -g -fsanitize=address stack\_corruption.c

### 4.2栈的Guard page

两个线程栈内存之间添加一个guard page（无权限，读取触发段错误）；



a->b->c->d->e，超过栈空间则触发段错误；

Guard page, 让出错现场更靠近真实原因，便于调试；

**堆和栈都是mmap实现**

Stack, heap都通过mmap得到vma

Pthread\_create创建的线程的栈，都是linux自己实现的；

Strace-readme， mmp分配stack内存

MAP\_GROWSDOWN，栈可以往下长；

Pthread\_create实现的栈，类似堆的实现，没有GROWSDOWN，划分一块内存，再按需分配；

内核提供的针对GROWSDOWN栈的guard page;

内核的改进

Mm:large stack guard gap, between vmas 提高到1M；

## 5. 如何设计多线程程序

### 5.1 划分线程的典型原则

《The Art of Concurrency》第四章Eight Simple Rules for Designing Multithread Applications.

(1).**区分真正独立的计算任务；**

(2).尽可能地在最高级别并发；

for..//并发

for …

for …

(3).尽早考虑多核个数；

(4).尽可能利用已有的线程安全库；

许多的库，例如Intel Math Kernel Library（Intel MKL）和Intel Integrated Performance Primitives (Intel IPP)，提供了能更好的利用多核处理器的并行版本的函数。

(5).使用合适的多线程模型；

有pthread, OpenMP 等，另外一本书

<https://maxim.int.ru/bookshelf/PthreadsProgram/htm/r_19.html>

Chapter 2 - Designing Threaded Programs

讲解基于pthread实现的多线程模型，难得的是1996年提出的模型，现今依然适用；

(6).永远不要假设多线程的执行顺序；

(7.)尽可能使用线程本地数据，或者对特定的数据加锁

数据越独立，更容易降低锁粒度；

(8)：敢于更换更易并行化的算法

并发：逻辑上的同时发出；

并行：物理上，实际同时执行；

以上规则，适合CPU型任务分解；

**补充**：**让I/O型任务与CPU计算型任务，同时运行而不是互相等待；**

### 5.2 典型的多线程模型

#### 1老板-工人(boss-worker)模型, w/o线程池；

(1)一个boss线程负责分派工作(delegation)，遇到具体任务，创建线程去干活；

main()

/\* The boss \*/

{

forever {

get a request

switch request:

case X: pthread\_create(...taskX)

case Y: pthread\_create(...taskY)

}

}

taskX() /\*Workers processing requests of type X\*/

{

perform the task, synchronize as needed if accessing shared resources

done

}

taskY() /\*Workers processing requests of type Y\*/

{

perform the task, synchronize as needed if accessing shared resources

done

}

(2)先创建好n个线程(对应n个worker)，遇到具体任务，放入队列，唤醒等待队列，准备好的线程干活；

main()

/\* The boss \*/

{

for the numnber of workers

pthread\_create(...poll\_base)

forever {

get a request

place request in work queue;

signal sleeping threads that work is available

}

}

poll\_base()

{

while(1) {

sleep until awoken by boss

dequeuea work request

switch (case)

case request X: taskX()

case request Y: taskY()

}

}

#### 2.PEER模型：

没有Boss分派工作的过程，每个线程都知道自己要做什么，有自己有输入(workcrew)

main()

{

pthread\_create(...thread1 ... task1)

pthread\_create(...thread2 ... task2)

...

signal all workers to start

wait for all workers fo finish

do any clean up

}

taks1()

{

wait for start

perform task, synchronize as needed if accessing shared resources

done

}

taks2()

{

wait for start

perform task, synchronize as needed if accessing shared resources

done

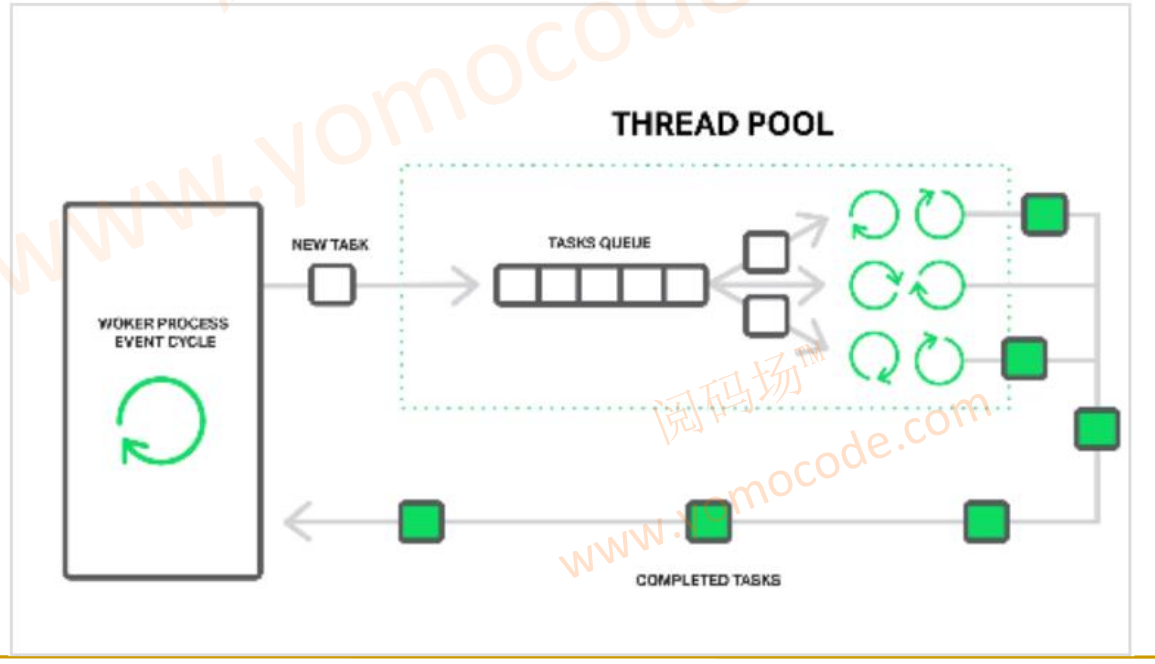
}

#### 3.线程池模型

避免频繁创建撤销，或者不确定数量的创建；线程的切换介于忙/闲之间，而非生死之间；

解偶作用：线程的创建与执行完全分开，方便维护；

放入一个池子中，可以给其他任务机型复用；



boss/worker模型，常与线程池模型结合实现；

code:

multiThread.cpp

#### 4.流水线模型：

流水线的每一个阶段应尽可能相等时间；

pthread\_t Thread[N]

Queues[N]

//initial thread

{

place all input into stage1 is queue

pthread\_create(&(Thread[1]...stage1...))

pthread\_create(&(Thread[2]...stage1...))

pthread\_create(&(Thread[3]...stage1...))

//...

}

void \*stageX(void \*X)

{

loop

suspend until input unit is in queue

loop while XQueue is not empty

dequeue intput unit

process input unit

enqueue input unit into next stage s queu

end loop

until done

return NULL;

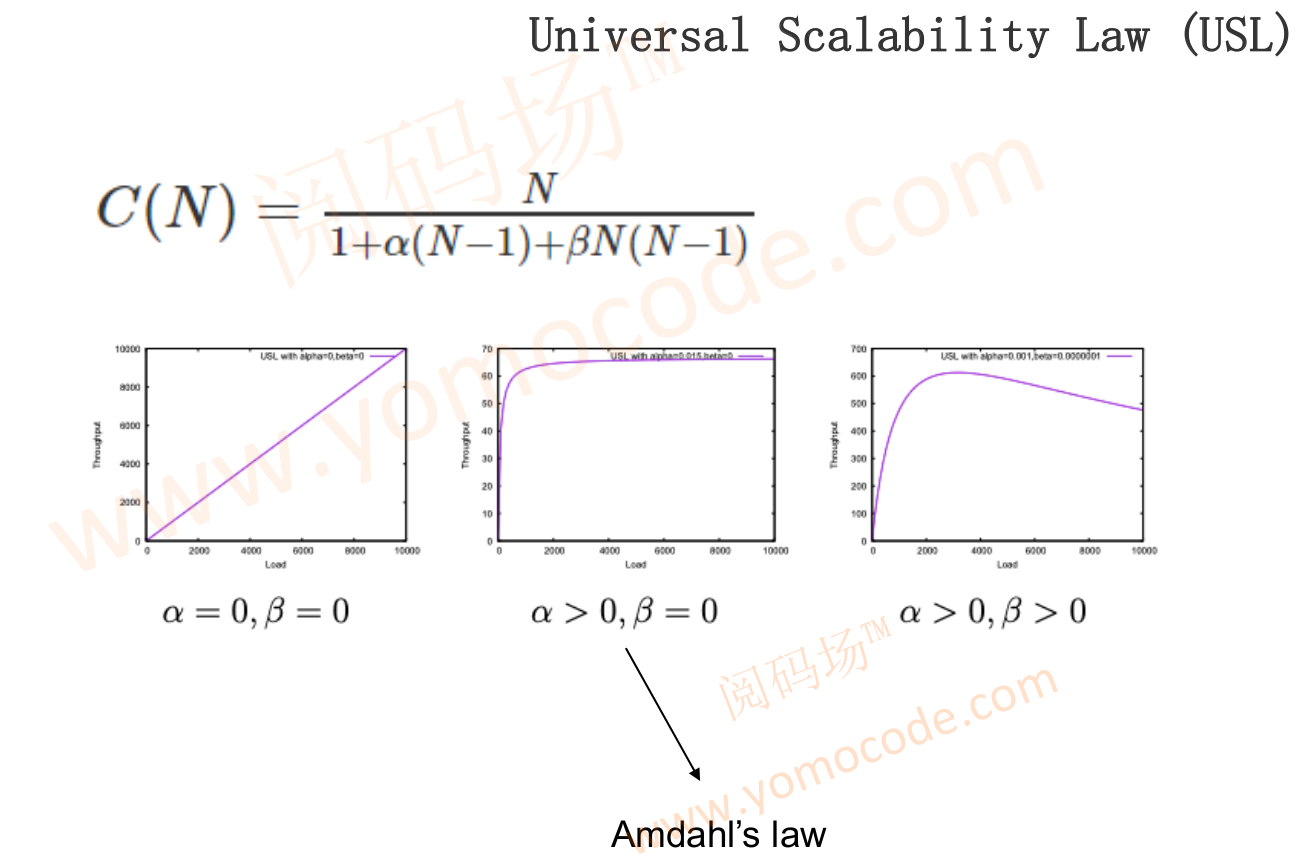
}

pipeline, 类似ARM流水线；

### 5.3 Amdahl定律

α: 串行化比例；

β多核间通信消耗；



当β=0时，就是Amdahl定律

### 5.4 多线程与I/O

#### 1异步IO

Aio\_read函数会立马返回，自动创建一个线程去干活；类似boos-worker模型的实现；

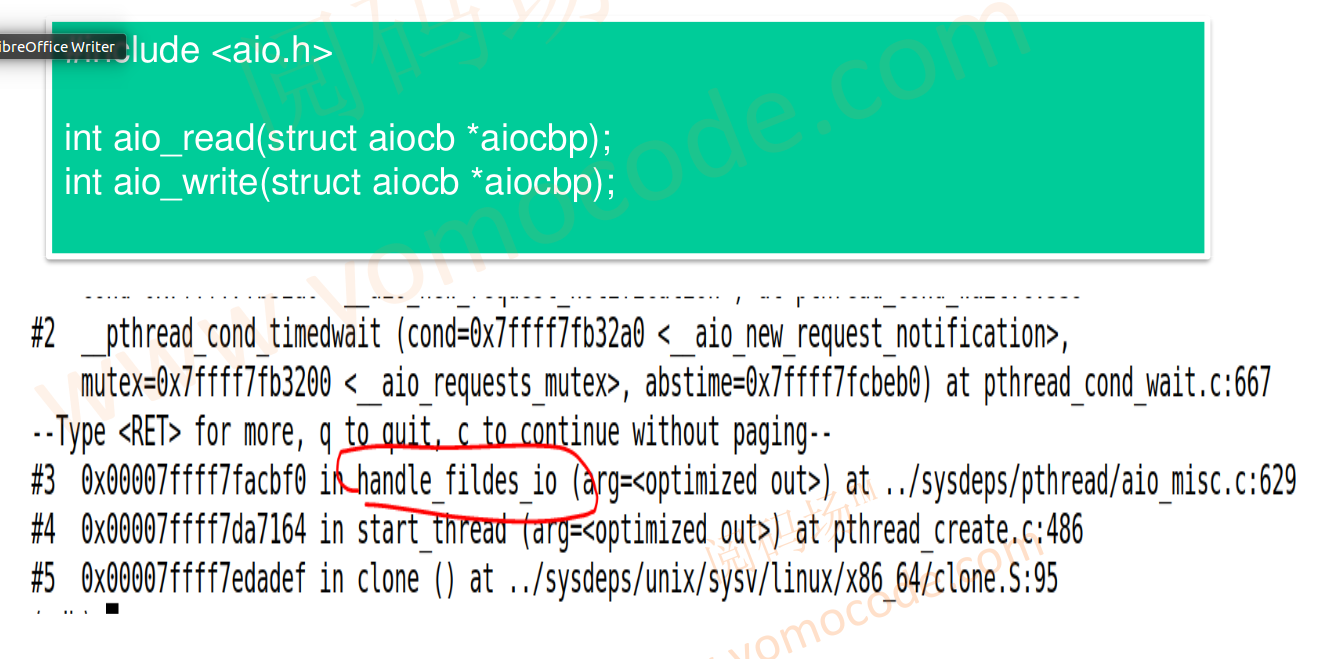
#include <aio.h>

int aio\_read(struct aiocb \*aiocbp);

int aio\_write(struct aiocb \*aiocbp);

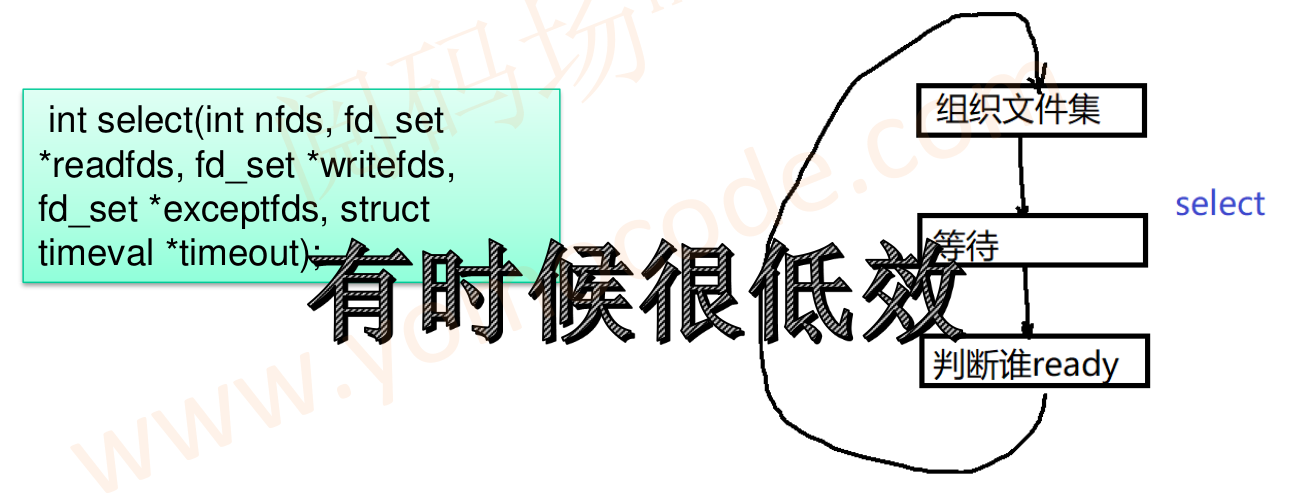
code:

...



#### 2多路复用

**(1)select**



因为存在一个唤醒文件查询过程，没调用一次，还需要重新设置观察文件集，当文件数量过大时，效率会地下；

while(1)死循环内添加fd到fdset, 等待select, 检查谁read-FD\_ISSET()

while(1){

FD\_ZERO(&rset);

for (int i=0; i <5; i++){

FD\_SET(fds[i],&rset);

}

puts("round again");

select(max+1, &rset, NULL, NULL, NULL);

for (int i=0; i < 5; i++) {

if (FD\_ISSET(fds[i],&rset)) {

memset(buffer,0,MAXBUF);

read(fds[i], buffer, MAXBUF);

puts(buffer);

}

}

}

**(2)epoll**

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

int epoll\_wait(int epfd, struct epoll\_event \*events,int maxevents, int timeout);

epoll在循环体外添加FD, 循环体内epoll\_wait,返回值知道ready的文件数量

for (int i=0; i <5; i++){

static struct epoll\_event ev;

memset(&client,0,sizeof(client));

addrlen = sizeof(client)

ev.data.fd = accept(sockfd, (struct sockaddr\*)&client, &addrlen);

en.events = EPOLLIN;

epoll\_ctl(epfd, EPOLL\_CTL\_ADD, ev.data.fd, &ev);

}

while(1){

puts("round again");

nfds = epoll\_wait(epfd, events, 5, 10000);

for (int i=0; i < nfds; i++) {

memset(buffer,0,MAXBUF);

read(events[i].data.fd,buffer,MAXBUF);

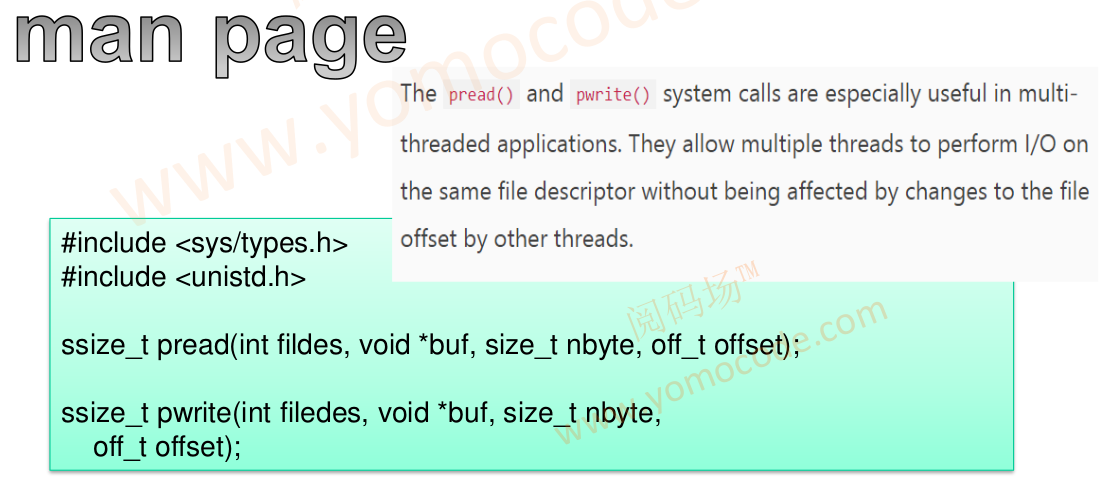
puts(buffer);

}

}

#### 3多线程文件冲突问题

文件是进程级的，一个线程访问文件，会导致另外一个线程访问文件的偏移位置等不可控；



C库提供的pread，pwrite函数解决多线程offset等冲突问题；(实现方法，共享数据加锁恢复)

## 6. 多线程调试与优化

6.1 gdb与多线程  
6.2 valgrind/callgrind与多线程  
6.3 perf  
6.4 top命令与多线程