s14

- s14
    - 线程(thread)
        - 线程的概念
        - 线程控制
            - 创建线程
            - 获取当前线程的id
            - 终止线程
    - 线程间同步
        - 互斥锁(mutex)
        - Mutex的init与destroy
        - 死锁(Deadlock)
            - 多个锁
        - 状态变量(Condition Variable)

        - 信号量(Semaphore)
        - 作业
            - 哲学家用餐问题
            - 停车场问题
    - 文件锁
        - 读锁(readlock)
        - 写锁(writelock)

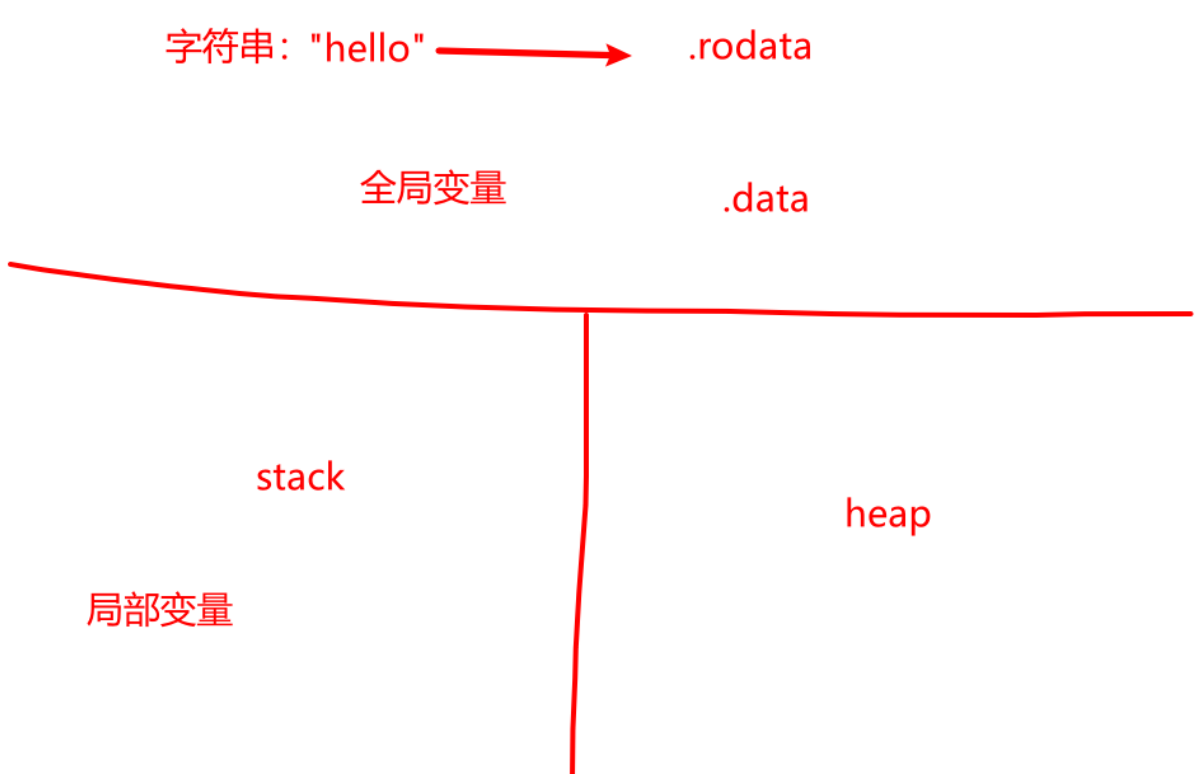
线程(thread)

是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务

线程的概念

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：

- 地址空间
    - .rodata(readonly data段,ELF严格来说不属于data段，在全局区域中



    - 堆空间 malloc返回的值可在函数间传递

1. 文件描述符表
2. 种信号的处理方式
3. 当前工作目录
4. 用户id和组id

但有些资源是每个线程各有一份的：

1. 线程id
2. 上下文，包括各种寄存器的值、程序计数器和栈指针
3. 栈空间
4. errno变量
5. 信号屏蔽字
6. 调度优先级

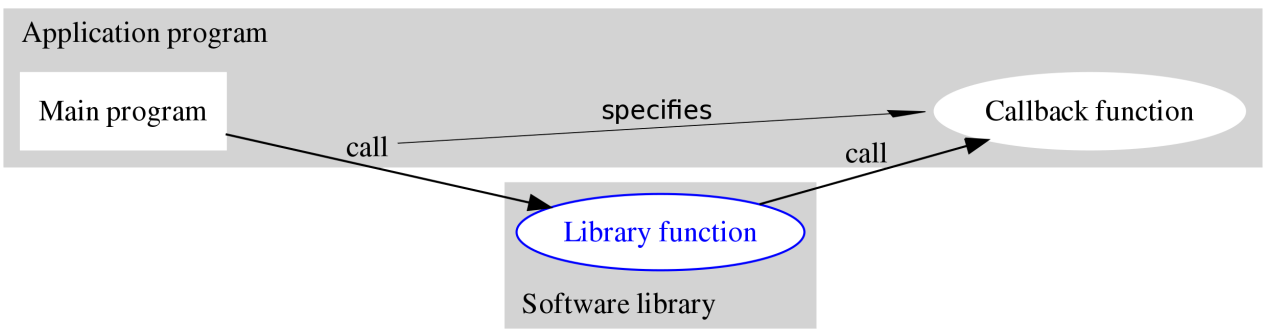
在Linux上线程函数位于libpthread共享库中，因此在编译时要加上-plthread

线程控制

创建线程

```
#include<pthread.h>

int pthread_create (pthread_t *restrict thread, \
/*pthread_t类似pid_t,返回一个“pid”，结果参数，记录子线程id，有返回值效果。restrict 加强安全性,内存只能通过该指针修改*/
const pthread attr_t *restrict attr, \
/*thread属性，课程中使用较少*/
void *(*start_routine)(void*), \
/*start_routine函数指针，(void *传啥都可以，可传入结构体等)，子线程入口地址*/
void *restrict arg/*传的参数列表*/);
```

- #a function,回调函数(call back):通过参数将函数传递到其它代码的，某一块可执行代码的引用。这一设计允许了底层代码调用在高层定义的子程序。https://zh.wikipedia.org/wiki/回调函数 可用于jsp前台窗口



```
/**/pcr(tid, NULL, func, argv)
```

返回值：成功返回0，失败返回错误号。以前学过的系统函数都是成功返回0，失败返回-1，而错误号保存在全局变量errno中，而pthread库的函数都是通过返回值返回错误号，虽然每个线程也都有一个errno，但这是为了兼容其它函数接口而提供的，pthread库本身并不使用它，通过返回值返回错误码更加清晰

获取当前线程的id

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Compile and link with -plthread.

返回值：总是成功返回，返回调用该函数线程ID

- man 3 pthread_create

The new thread inherits a copy of the creating thread's signal mask (pthread_sigmask(3)). The set of pending signals for the new thread is empty (sigpending(2)). The new thread does not inherit the creating thread's alternate signal stack (sigaltstack(2)).

- man 3 pthread_self
- createThread.c
    - ld链接器

```
youhuangla@Ubuntu s14 % gcc createThread.c
[0]
/tmp/ccnKCUYo.o: In function `main':
createThread.c:(.text+0x5d): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
youhuangla@Ubuntu s14 % gcc createThread.c -lpthread
[0]
youhuangla@Ubuntu s14 % ./a.out
[0]
maint thread
```

```c
/*************************************************************************
        > File Name: createThread.c
        > Author:
        > Mail:
        > Created Time: Wed 09 Feb 2022 11:08:08 PM CST
 ************************************************************************/

#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
void *thr_fn(void *arg) {
    printf("%s\n", (char *)arg);
    return NULL;
}
int main() {
    pthread_t ntid;
    int ret;
    ret = pthread_create(&ntid, NULL, thr_fn, "new thread");
    if (ret != 0) {
        //error
        printf("create thread err:%s\n", strerror(ret));
        exit(1);
    }
    sleep(1);//missing will only print maint thread, as main terminate, thread
terminate.
    printf("main thread\n");

    return 0;
}
```

```
youhuangla@Ubuntu s14 % ./a.out
[0]
new thread
main thread
```

Q：主线程在一个全局变量ntid中保存了新创建的线程的id，如果新创建的线程不调用pthread_self而是直接打印这个ntid，能不能达到同样的效果？

```c
/*************************************************************************
        > File Name: createThread.c
        > Author:
        > Mail:
        > Created Time: Wed 09 Feb 2022 11:08:08 PM CST
 ************************************************************************/

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

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

pthread_t ntid;
void printid(char *tip) {
    pid_t pid = getpid();
    pthread_t tid = pthread_self();
    printf("%s pid: %u tid: %lu (%p)\n", tip, pid, tid, (void *)tid);
    return ;
}
void *thr_fn(void *arg) {
    printid(arg);
    printf("%s ntid = %p\n", (char *)arg, (void *)ntid);
    return NULL;
}
int main() {
    int ret;
    ret = pthread_create(&ntid, NULL, thr_fn, "new  thread");
    if (ret != 0) {
        //error
        printf("create thread err:%s\n", strerror(ret));
        exit(1);
    }
    sleep(1);//missing will only print maint thread, as main terminate, thread terminate.
    printid("main thread");

    return 0;
}
```

```c
/*************************************************************************
> File Name: 1_createThread.c
> Author:
> Mail:
> Created Time: Thu 10 Feb 2022 05:03:39 PM CST
************************************************************************/

#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;
}
```

终止线程

如果需要只终止某个线程而不终止整个进程，可以有三种方法：

1. 从线程函数return。这种方法对主线程不适用，从main函数return相当于调用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。
3. 线程可以调用pthread_exit终止自己。

```
#include <pthread.h>void pthread_exit(void *value_ptr);
```

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。

```
#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。

```c
/*************************************************************************
        > File Name: pthread_exit.c
        > Author:
        > Mail:
        > Created Time: Thu 10 Feb 2022 05:03:26 PM CST
 ************************************************************************/

#include <stdio.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);
    return NULL;
}
void *thr_fn3(void *arg) {
    while (1) {
        printf("thread 3 sleeping\n");
        sleep(1);
    }
    return NULL;
}
int main() {
    pthread_t tid;
    void *sts;
    pthread_create(&tid, NULL, thr_fn1, NULL);
    pthread_join(tid, &sts);
    printf("thread 1 exit code %ld\n", (long)sts);

    pthread_create(&tid, NULL, thr_fn2, NULL);
    pthread_join(tid, &sts);
    printf("thread 2 exit code %ld\n", (long)sts);

    pthread_create(&tid, NULL, thr_fn3, NULL);
    sleep(3);

    pthread_cancel(tid);
    pthread_join(tid, &sts);
    printf("thread 3 exit code %ld\n", (long)sts);

    return 0;
}
```

```
youhuangla@Ubuntu s14 % vim pthread_exit.c
[0]
youhuangla@Ubuntu s14 % gcc pthread_exit.c -lpthread
[0]
```

```
youhuangla@Ubuntu s14 % ./a.out
[0]
thread 1 returning
thread 1 exit code 1
thread 2 exiting
thread 2 exit code 2
thread 3 sleeping
thread 3 sleeping
thread 3 sleeping
thread 3 exit code -1
youhuangla@Ubuntu s14 % nm a.out
000000000000090f T main
                 U printf@@GLIBC_2.2.5#u是未实现的函数，在动态库中实现
                 U pthread_cancel@@GLIBC_2.2.5
                 U pthread_create@@GLIBC_2.2.5
                 U pthread_exit@@GLIBC_2.2.5
                 U pthread_join@@GLIBC_2.2.5
```

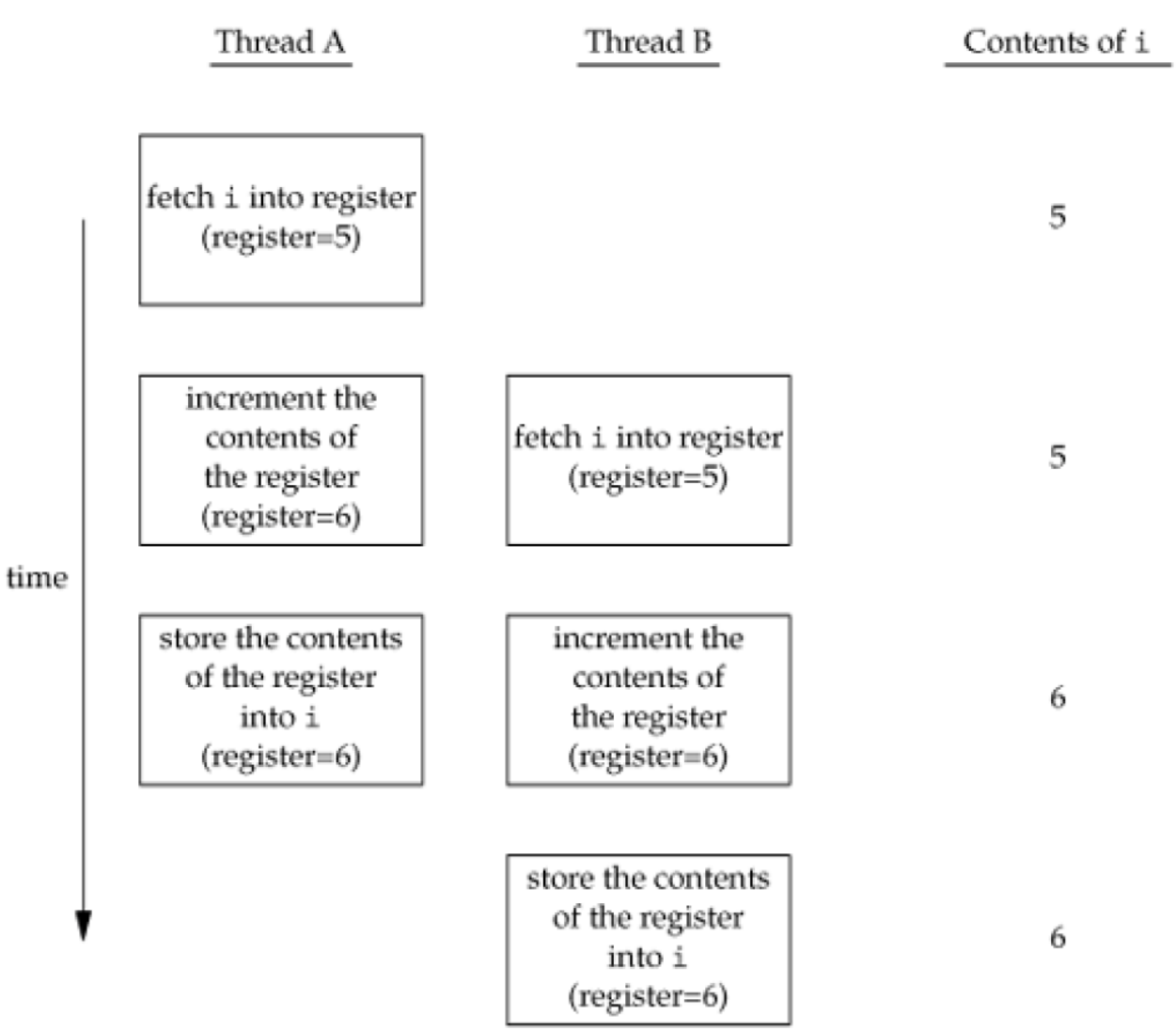
线程间同步

互斥锁(mutex)

多个线程同时访问共享数据时可能会冲突，这跟前面讲信号时所说的可重入性是同样的问题。比如两个线程都要把某个全局变量增加1，这个操作在某平台需要三条指令完成：

1. 从内存读变量值到寄存器
2. 寄存器的值加1
3. 将寄存器的值写回内存

假设两个线程在多处理器平台上同时执行这三条指令，则可能导致下图所示的结果，最后变量只加了一次而非两次，（下图出自[APUE2e]）。



- #以此编写下面的程序，有趣的是，隔了一段时间后运行a.out时，结果总是停在4999，连续不断运行才使得结果超过5000，而一站式编程中的程序(./my_addnum)总是>=5000

```c
/*************************************************************************
        > File Name: addnum.c
        > Author:
        > Mail:
        > Created Time: Thu 10 Feb 2022 08:49:05 PM CST
 ************************************************************************/

#include <stdio.h>
#include <pthread.h>
int cnt = 0;

void *cntadd(void *arg) {
    int val, i;
    for (i = 0; i < 5000; i++) {
        val = cnt;
        printf("%x: %d\n", (unsigned int)pthread_self(), val);
        cnt = val + 1;
    }
    return NULL;
}
```

```c
int main() {
    pthread_t tida, tidb;
    pthread_create(&tida, NULL, cntadd, NULL);
    pthread_create(&tidb, NULL, cntadd, NULL);
    pthread_join(tida, NULL);
    pthread_join(tidb, NULL);
    return 0;
}
```

```
youhuangla@Ubuntu s14 %./a.out
df801700: 1084
df000700: 2417
......
df000700: 3349
df801700: 1085
......
df000700: 5000
df000700: 5001
df000700: 5002
df000700: 5003
df000700: 5004
```

```c
/*************************************************************************
        > File Name: my_addnum.c
        > Author:
        > Mail:
        > Created Time: Thu 10 Feb 2022 09:04:42 PM CST
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NLOOP 5000

int counter; /* incremented by threads */
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++) {
       val = counter; printf("%x: %d\n", (unsigned int)pthread_self(), val + 1);
        counter = val + 1;
```

```
    }
    return NULL;
}
```

```
b27f8700: 4998
b27f8700: 4999
b27f8700: 5000
```

Mutex的init与destroy

```
#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;
```

- 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，本章不详细介绍Mutex属性，感兴趣的读者可以参考[APUE2e]。
- 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。

```
#include <stdio.h>
#include <pthread.h>
//#include <sys/types.h>

pthread_mutex_t add_lock = PTHREAD_MUTEX_INITIALIZER;

int cnt = 0;

void *cntadd(void *arg) {
    int val, i;
    for (i = 0; i < 5000; i++) {
        pthread_mutex_lock(&add_lock);
        val = cnt;
        printf("%x: %d\n", (unsigned int)pthread_self(), val);
        cnt = val + 1;
        pthread_mutex_unlock(&add_lock);
    }
    return NULL;
}
```

```
int main() {
    pthread_t tida, tidb;
    pthread_create(&tida, NULL, cntadd, NULL);
    pthread_create(&tidb, NULL, cntadd, NULL);
    pthread_join(tida, NULL);
    pthread_join(tidb, NULL);
    return 0;
}
```

“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。

死锁(Deadlock)

- 一般情况下，如果同一个线程先后两次调用lock，在第二次调用时，由于锁已经被占用，该线程会挂起等待别的线程释放锁，然而锁正是被自己占用着的，该线程又被挂起而没有机会释放锁，因此就永远处于挂起等待状态了，这叫做死锁（Deadlock）。
- 另一种典型的死锁情形是这样：线程A获得了锁1，线程B获得了锁2，这时线程A调用lock试图获得锁2，结果是需要挂起等待线程B释放锁2，而这时线程B也调用lock试图获得锁1，结果是需要挂起等待线程A释放锁1，于是线程A和B都永远处于挂起状态了。不难想象，如果涉及到更多的线程和更多的锁，有没有可能死锁的问题将会变得复杂和难以判断。
- 下列程序注释掉了for loop中的unlock，死锁。

```
#include <stdio.h>
#include <pthread.h>
//#include <sys/types.h>

pthread_mutex_t add_lock = PTHREAD_MUTEX_INITIALIZER;

int cnt = 0;

void *cntadd(void *arg) {
    int val, i;
    for (i = 0; i < 5000; i++) {
        pthread_mutex_lock(&add_lock);
        val = cnt;
        printf("%x: %d\n", (unsigned int)pthread_self(), val);
        cnt = val + 1;
        //pthread_mutex_unlock(&add_lock);
```

```
    }
    return NULL;
}

int main() {
    pthread_t tida, tidb;
    pthread_create(&tida, NULL, cntadd, NULL);
    pthread_create(&tidb, NULL, cntadd, NULL);
    pthread_join(tida, NULL);
    pthread_join(tidb, NULL);
    return 0;
}
```

```
youhuangla@Ubuntu s14 % vim mutex_addnum.c
[2]
youhuangla@Ubuntu s14 % gcc mutex_addnum.c   -lpthread -o mutex_addnu
[0]
youhuangla@Ubuntu s14 % ./mutex_addnu
[0]
a37f9700: 0
^C
```

多个锁

- 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。比如一个程序中用到锁1、锁2、锁3，它们所对应的Mutex变量的地址是锁1<锁2<锁3，那么所有线程在需要同时获得2个或3个锁时都应该按锁1、锁2、锁3的顺序获得。如果要为所有的锁确定一个先后顺序比较困难，则应该尽量使用pthread_mutex_trylock调用代替pthread_mutex_lock调用，以免死锁。

状态变量(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：
- #蹲坑(执行)没带纸(条件)

```
#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;
```

可见，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。

- #broadcast来了一张纸(状态)，所有人都被惊醒了
- #Makefile中如何link lpthread(以编译mutex_addnum为例)，参考 https://stackoverflow.com/questions/6332410/compile-link-error-using-pthread仿写，原本我写的是-lpthread

```
mutex_addnum : mutex_addnum.o
        gcc -lpthread mutex_addnum.o

mutex_addnum.o : mutex_addnum.c
        gcc -c mutex_addnum.c -o
```

- 结果如下

```
gcc -c mutex_addnum.c
gcc -lpthread mutex_addnum.o
mutex_addnum.o: In function `main':
mutex_addnum.c:(.text+0x99): undefined reference to `pthread_create'
mutex_addnum.c:(.text+0xb6): undefined reference to `pthread_create'
mutex_addnum.c:(.text+0xc7): undefined reference to `pthread_join'
mutex_addnum.c:(.text+0xd8): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
Makefile:2: recipe for target 'mutex_addnum' failed
make: *** [mutex_addnum] Error 1
```

- https://stackoverflow.com/questions/1662909/undefined-reference-to-pthread-create-in-linux得知-lpthread是library specification的意思，修改后编译出./a.out成功，makefile看起来挺难的

```
mutex_addnum : mutex_addnum.o
        gcc -pthread mutex_addnum.o

mutex_addnum.o : mutex_addnum.c
        gcc -c mutex_addnum.c -o
```

信号量(Semaphore)

- Mutex变量是非0即1的，可看作一种资源的可用数量，初始化时Mutex是1，表示有一个可用资源，加锁时获得该资源，将Mutex减到0，表示不再有可用资源，解锁时释放该资源，将Mutex重新加到1，表示又有了一个可用资源。
- 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，本节只介绍这种情况。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。

- #生产者与消费者，生产者有5个空位，每生产一个产品消耗一个空位，消费者每次消费一个资源，空出一个空位

```
```

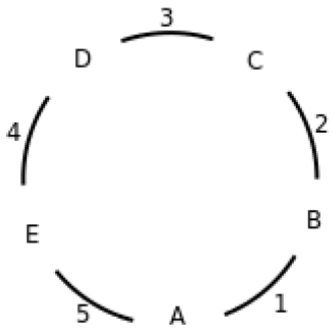
```c
/*************************************************************************
        > File Name: sem.c
        > Author:
        > Mail:
        > Created Time: Sun 13 Feb 2022 05:26:13 PM CST
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM 5

int q[NUM];
sem_t blank_number, goods_number;
void *producer(void *arg) {
    int i = 0;
    while (1) {
        sem_wait(&blank_number);//reduce a blank
        q[i] = rand() % 100 + 1;
        printf("produce %d\n", q[i]);
        sem_post(&goods_number);//increase a goods
        i = (i + 1) % NUM;
        sleep(rand() % 3);
    }
}

void *consumer(void *arg) {
    int i = 0;
    while (1) {
        sem_wait(&goods_number);
        printf("consume %d\n", q[i]);
        q[i] = 0;
        sem_post(&blank_number);
        i = (i + 1) % NUM;
        sleep(rand() % 3);
    }
}

int main(void) {
    srand(time(NULL));

    sem_init(&blank_number, 0, NUM);
    sem_init(&goods_number, 0, 0);//0 product

    pthread_t pid, cid;
    pthread_create(&pid, NULL, producer, NULL);
    pthread_create(&cid, NULL, consumer, NULL);
    pthread_join(pid, NULL);
    pthread_join(cid, NULL);

    return 0;
}
```

作业

哲学家用餐问题

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



- #picture smaller by typora

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。

停车场问题

编写一个程序，开启3个线程，这3个线程的ID分别为A、B、C，每个线程将自己的ID在屏幕上打印10遍，要求输出结果必须按ABC的顺序显示；如：ABCABC….依次递推

用信号量模拟一个停车场，停车场有3个进出口，总共可以停N辆车，每个出口都可以进车，出车。每个出口都是一个线程，先roll一个随机数，奇数表示进车，偶数表示出车，一个出口出车或进车后，停车场的容量发生变化，当停车场满的时候任何出口都不能再进车。

文件锁

在多进程对同一个文件进行读写访问时，为了保证数据的完整性，有时需要对文件进行锁定。可以通过fcntl()函数对文件进行锁定和解锁。

对于写锁（F__WRLCK独占锁），只有一个进程可以在文件的任一特定区域上享有独占锁。对于读锁（F__RDLCK 共享锁），许多不同的进程可以同时拥有文件上同一区域上的共享锁。为了拥有共享锁，文件必须以读或者读 / 写的方式打开。只要任一进程拥有共享锁，那么其他进程就无法再获得独占锁。

读锁(readlock)

```c
/*************************************************************************
        > File Name: readlock.c
        > Author:
        > Mail:
        > Created Time: Sun 13 Feb 2022 09:40:06 PM CST
 ************************************************************************/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>
int main(void) {
    int fd = open("./hello.txt", O_RDONLY);
    if (fd < 0) {
        perror("open");
        exit(1);
    }
    struct stat sta;
    fstat(fd, &sta);

    struct flock lock;
    lock.l_type = F_RDLCK;
    lock.l_pid = getpid();

    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = sta.st_size;

    printf("pid: %d ", lock.l_pid);//can also getpid(), but slow compare to struct member
    if (fcntl(fd, F_SETLK, &lock)) {
        perror("fcntl");
        exit(1);
    } else {
        printf("add read lock successfully\n");
    }

    sleep(10);
    return 0;
}
```

```
youhuangla@Ubuntu s16 % gcc readlock.c
youhuangla@Ubuntu s16 % ./a.out
[0]
pid: 13590 add read lock successfully
# in new copy
youhuangla@Ubuntu s16 % ./a.out
[0]
pid: 13589 add read lock successfully
```

写锁(writelock)

```c
/*************************************************************************
        > File Name: readlock.c
        > Author:
        > Mail:
        > Created Time: Sun 13 Feb 2022 09:40:06 PM CST
 ************************************************************************/
```

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>
int main(void) {
    int fd = open("./hello.txt", O_WRONLY);
    if (fd < 0) {
        perror("open");
        exit(1);
    }
    struct stat sta;
    fstat(fd, &sta);

    struct flock lock;
    lock.l_type = F_WRLCK;
    lock.l_pid = getpid();

    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = sta.st_size;

    printf("pid: %d ", lock.l_pid);//can also getpid(), but slow compare to struct
member
    while (fcntl(fd, F_SETLK, &lock) < 0) {
        perror("fcntl");

        struct flock lock_1;//new lock for test
        lock_1 = lock;
        lock_1.l_type = F_WRLCK;

        fcntl(fd, F_GETLK, &lock_1);
        switch (lock_1.l_type) {
            case F_UNLCK:
                printf("get no lock\n");
                break;
            case F_RDLCK:
                printf("get read lock of pid = %d\n", lock_1.l_pid);
                break;
            case F_WRLCK:
                printf("get write lock of pid = %d\n", lock_1.l_pid);
                break;
        }

        sleep(1);
    }

    printf("set write lock successfully\n");
    getchar();
    close(fd);

    return 0;
}
```

```
#one shell
youhuangla@Ubuntu s16 % ./readlock
[0]
pid: 21678 add read lock successfully
#another shell,wait for 10 sec to success.You press enter to execute getchar() and
end the program.
youhuangla@Ubuntu s16 % ./writelock
[0]
fcntl: Resource temporarily unavailable
pid: 21679 get read lock of pid = 21678
fcntl: Resource temporarily unavailable
get read lock of pid = 21678
fcntl: Resource temporarily unavailable
get read lock of pid = 21678
fcntl: Resource temporarily unavailable
get read lock of pid = 21678
fcntl: Resource temporarily unavailable
get read lock of pid = 21678
fcntl: Resource temporarily unavailable
get read lock of pid = 21678
fcntl: Resource temporarily unavailable
get read lock of pid = 21678
fcntl: Resource temporarily unavailable
get read lock of pid = 21678
set write lock successfully
```