s14

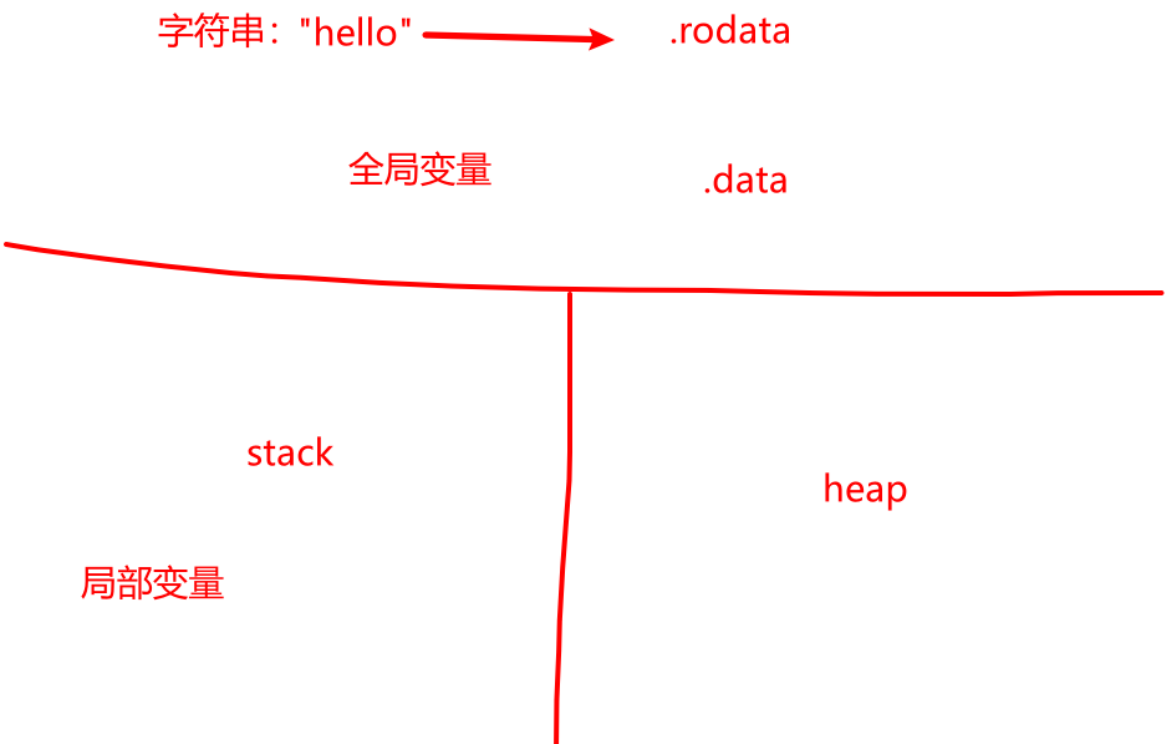
1. 线程(thread)

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

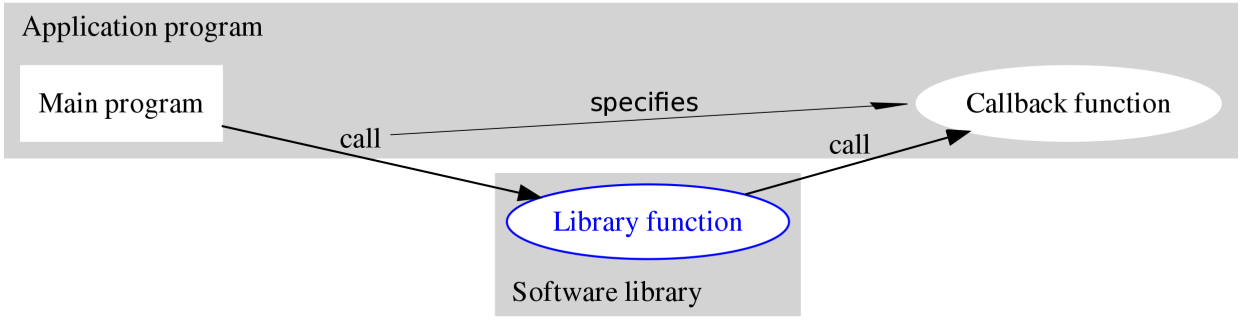
线程控制

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
```

```
/*************************************************************************
        > 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？

```
/*************************************************************************
        > 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>

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;
}
```

```
/*************************************************************************
> 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。

```
/*************************************************************************
        > 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
```

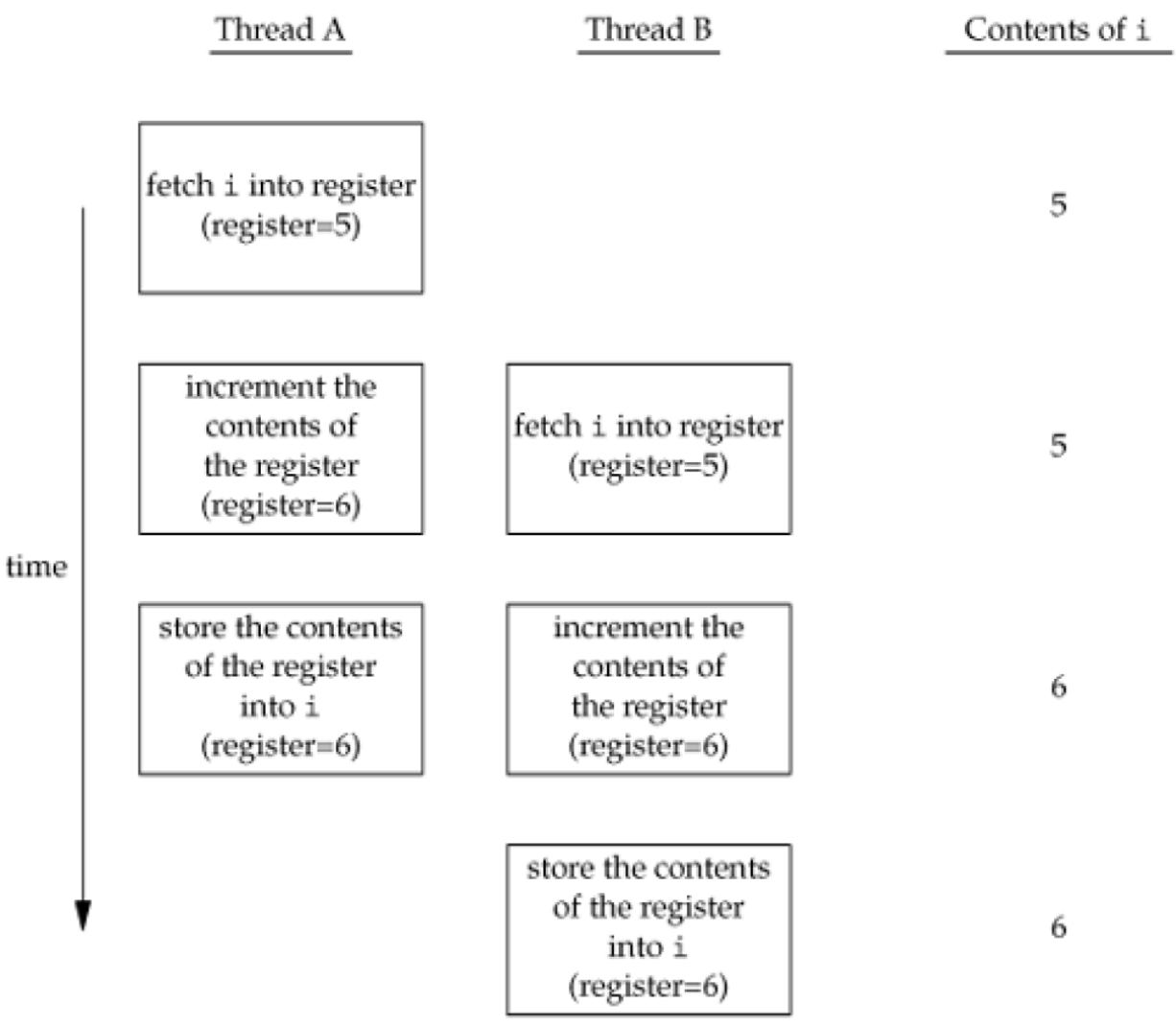
线程间同步

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

```
/*************************************************************************
        > 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;
}

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
```

```
/*************************************************************************
        > 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
```