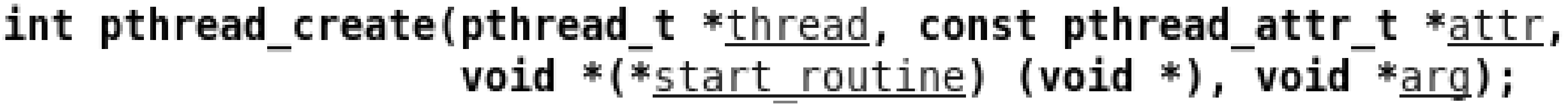
系统编程

基于TaiShan服务器/openEuler OS 的实践

第三讲：多线程编程 – 线程属性

深圳大学-冯禹洪，yuhongf@szu.edu.cn

线程属性 – 配置线程的状态和行为



```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

```
typedef struct
{
  int detachstate;                       //分离状态
  int schedpolicy;                       //调度策略
  struct sched_param schedparam;         //调度参数
  int inheritsched;                      //继承性
  int scope;                             //作用域
  size_t guardsize;                      //栈警戒缓冲区大小
  int stackaddr_set;                     //栈的设置
  void* stackaddr;                       //栈的启始地址
  size_t stacksize;                      //栈大小
}pthread_attr_t
```

配置属性流程

1. 声明类型为pthread_attr_t的属性对象变量

2. 调用函数pthread_attr_init()初始化线程属性对象

3. 调用属性设置函数配置属性对象

4. 属性对象作为参数2调用函数pthread_create()创建新线程

5. 调用函数pthread_attr_destroy()去除属性对象初始化和设置

① 该变量不是被内存回收

② 该变量可继续用于其他的线程属性设置

配置属性流程

- **调用属性设置函数设置对象属性**
  - 增加代码的可移植性
    - 隐藏属性配置细节
  - 简化线程属性管理规范
    - 一次性初始化线程属性
      - 创建时确定
      - 创建后不能修改
    - 针对线程（组）配置属性
      - 不同服务，不同的线程行为
- **属性对象 初始化 & 去初始化 成对出现**
  - 初始化属性对象（分配内存）
  - 去除属性对象初始化（释放内存）

内存泄露 ……

初始化属性

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *tattr);
```

- 初始化属性数据：缺省值；
- 分配存储空间

```
int pthread_attr_destroy(pthread_attr_t *tattr);
```

- 释放存储空间

属性	缺省值
scope	PTHREAD_SCOPE_PROCESS
detachstate	PTHREAD_CREATE_JOINABLE
stackaddr	NULL
stacksize	1M
priority	0
inheritsched	PTHREAD_EXPLICIT_SCHED
schedpolicy	SCHED_OTHER

■ 返回值

- 成功：0
- 失败：出错代码

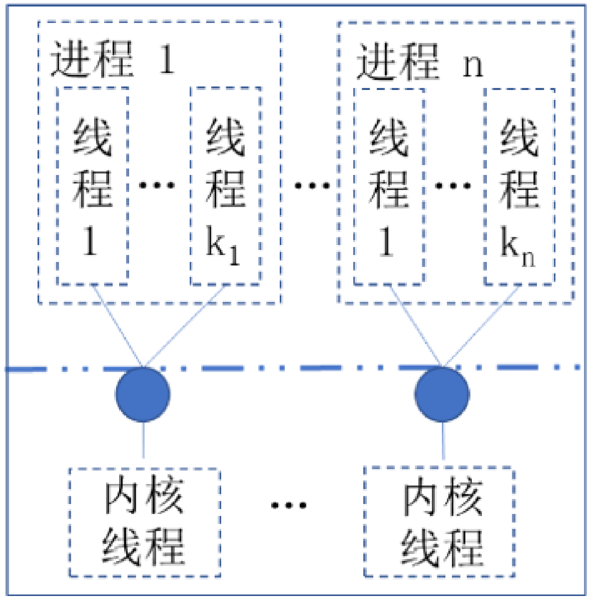
线程属性 - 作用域（是否绑定）

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope)
int pthread_attr_getscope(pthread_attr_t *attr, int *scope)
```

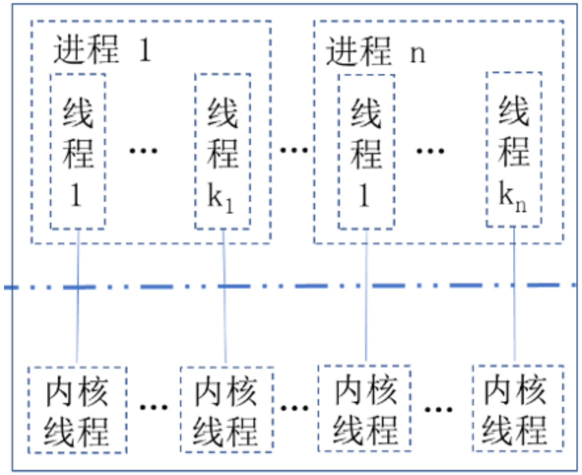
- **设置新线程将与哪些线程竞争CPU资源**
  - PTHREAD_SCOPE_PROCESS
    - 非绑定
    - 局部竞争（local contention scope）
    - 调度时：同一进程的线程之间竞争CPU
    - 线程模型：(M:1,多对1)
  - PTHREAD_SCOPE_SYSTEM
    - 绑定
    - 全局竞争（global contention scope）
    - 调度时：线程在系统级竞争CPU
    - 线程模型：(1:1，1对1)

线程属性 － 作用域

- **轻进程(LWP: Light Weight Process)**
  - 内核线程，内核的调度实体
  - 系统对线程资源的分配和对线程的控制单位
  - 一个轻进程可控制一个或多个线程
- **非绑定状态(默认状态),操作系统控制:**
  - 启动多少个轻进程
  - 轻进程与线程的映射关系
- **绑定状态**
  - 提高响应速度
  - 设置被绑定的轻进程的优先级和调度级来进一步提高

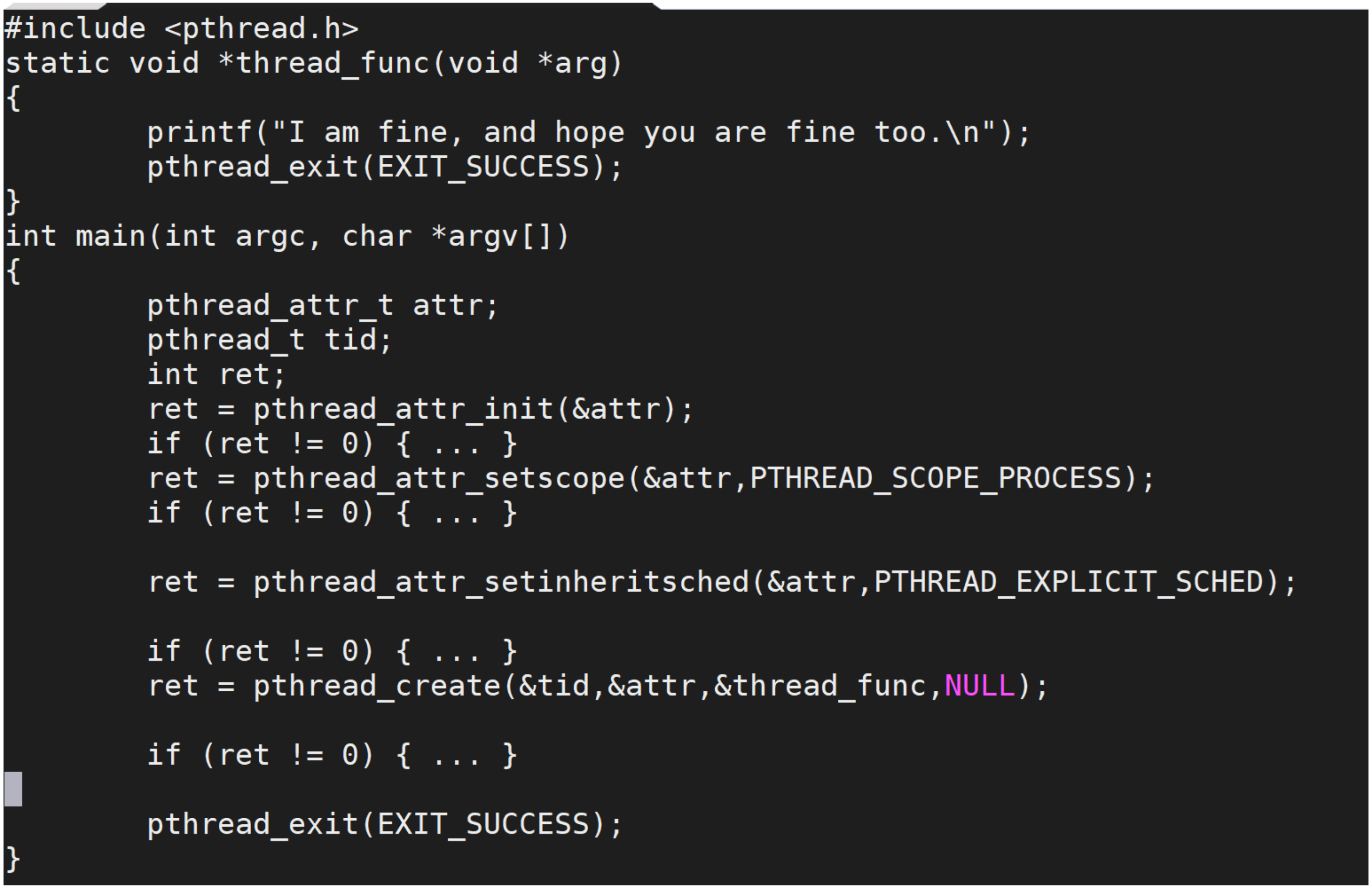


M:1



1:1

设置新线程的作用域例程



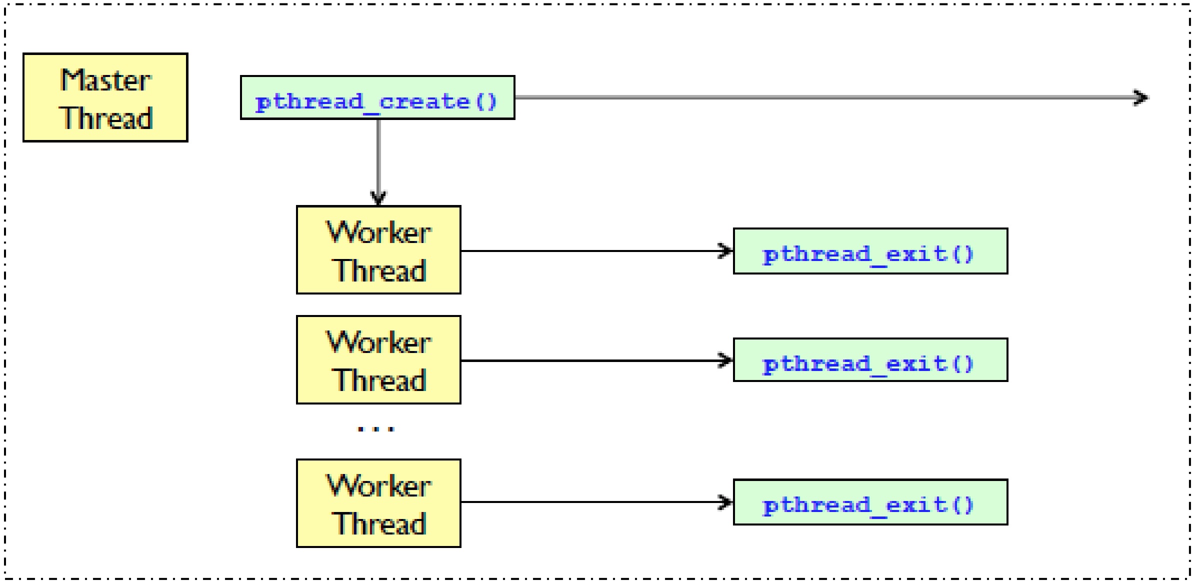
```
#include <pthread.h>
static void *thread_func(void *arg)
{
        printf("I am fine, and hope you are fine too.\n");
        pthread_exit(EXIT_SUCCESS);
}
int main(int argc, char *argv[])
{
        pthread_attr_t attr;
        pthread_t tid;
        int ret;
        ret = pthread_attr_init(&attr);
        if (ret != 0) { ... }
        ret = pthread_attr_setscope(&attr,PTHREAD_SCOPE_PROCESS);
        if (ret != 0) { ... }

        ret = pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);

        if (ret != 0) { ... }
        ret = pthread_create(&tid,&attr,&thread_func,NULL);

        if (ret != 0) { ... }

        pthread_exit(EXIT_SUCCESS);
}
```

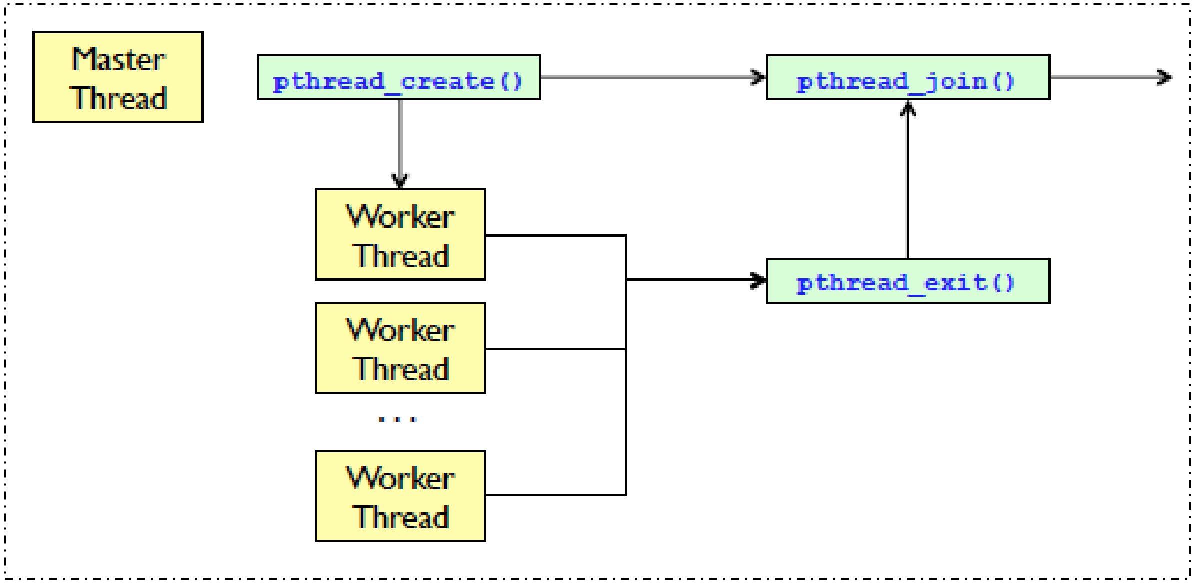
线程属性 - 分离状态

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
int pthread_attr_getdetachstate(pthread_attr_t *attr,
                                int *detachstate);
```

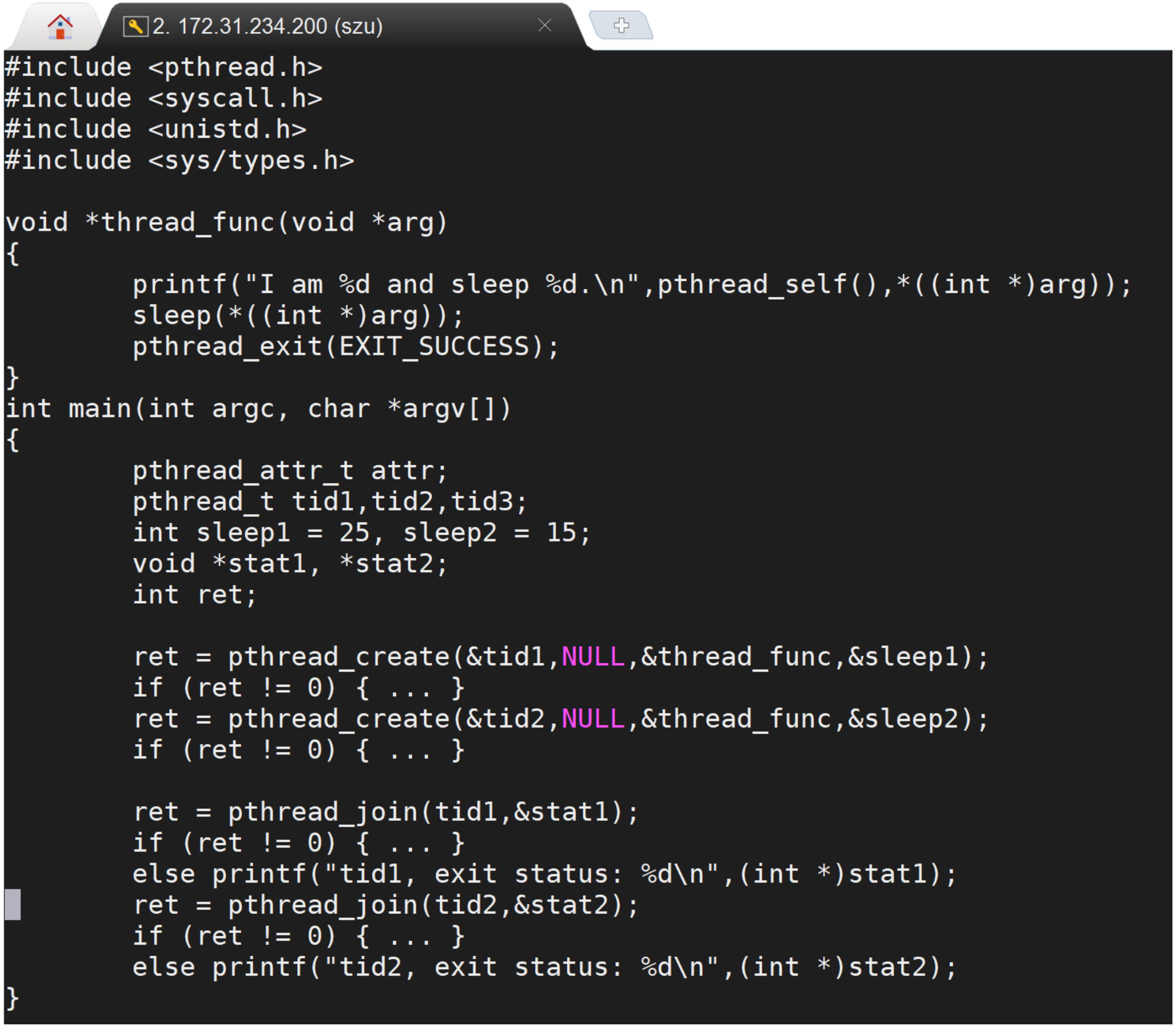
- **设置新线程是否与同一进程中其他线程同步**
  - PTHREAD_CREATE_DETACHED(分离状态)
    - 线程终止时,自动清除状态并释放系统资源
    - 其他线程不能调用pthread_join与其同步
  - PTHREAD_CREATE_JOINABLE(缺省,非分离状态)
    - 终止时不会自动清除状态,状态保留在系统中直至被获取或主线程退出
    - 可被另一线程调用pthread_join来获取其返回状态

Detached Threads vs. Joined Threads



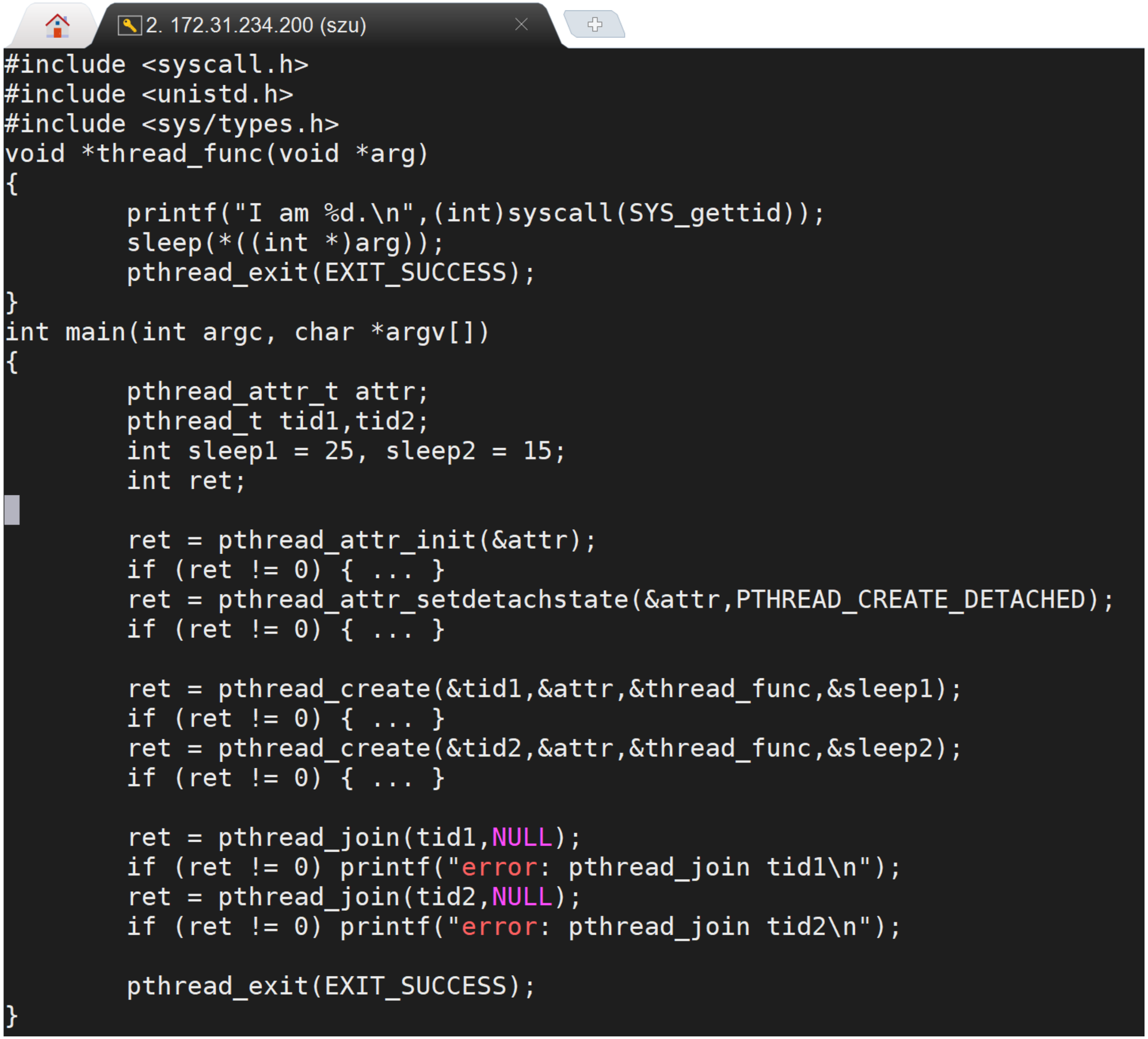


设置新线程的分离状态域例程 - PTHREAD_CREATE_JOINABLE



```
#include <pthread.h>
#include <syscall.h>
#include <unistd.h>
#include <sys/types.h>

void *thread_func(void *arg)
{
        printf("I am %d and sleep %d.\n",pthread_self(),*((int *)arg));
        sleep(*((int *)arg));
        pthread_exit(EXIT_SUCCESS);
}
int main(int argc, char *argv[])
{
        pthread_attr_t attr;
        pthread_t tid1,tid2,tid3;
        int sleep1 = 25, sleep2 = 15;
        void *stat1, *stat2;
        int ret;

        ret = pthread_create(&tid1,NULL,&thread_func,&sleep1);
        if (ret != 0) { ... }
        ret = pthread_create(&tid2,NULL,&thread_func,&sleep2);
        if (ret != 0) { ... }

        ret = pthread_join(tid1,&stat1);
        if (ret != 0) { ... }
        else printf("tid1, exit status: %d\n",(int *)stat1);
        ret = pthread_join(tid2,&stat2);
        if (ret != 0) { ... }
        else printf("tid2, exit status: %d\n",(int *)stat2);
}
```

设置新线程的分离状态域例程 - PTHREAD_CREATE_DETACHED



```
#include <syscall.h>
#include <unistd.h>
#include <sys/types.h>
void *thread_func(void *arg)
{
        printf("I am %d.\n",(int)syscall(SYS_gettid));
        sleep(*((int *)arg));
        pthread_exit(EXIT_SUCCESS);
}
int main(int argc, char *argv[])
{
        pthread_attr_t attr;
        pthread_t tid1,tid2;
        int sleep1 = 25, sleep2 = 15;
        int ret;

        ret = pthread_attr_init(&attr);
        if (ret != 0) { ... }
        ret = pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
        if (ret != 0) { ... }

        ret = pthread_create(&tid1,&attr,&thread_func,&sleep1);
        if (ret != 0) { ... }
        ret = pthread_create(&tid2,&attr,&thread_func,&sleep2);
        if (ret != 0) { ... }

        ret = pthread_join(tid1,NULL);
        if (ret != 0) printf("error: pthread_join tid1\n");
        ret = pthread_join(tid2,NULL);
        if (ret != 0) printf("error: pthread_join tid2\n");

        pthread_exit(EXIT_SUCCESS);
}
```

线程属性 - 调度策略

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr,
                                int *policy);
```

- 非实时调度策略
  - SCHED_OTHER 标准时间片轮转分时策略
  - SCHED_BATCH 用于批处理模式运行的进程
  - SCHED_IDLE 用于运行优先级非常低的后台作业
- 实时调度策略
  - SCHED_FIFO:先进先出
  - SCHED_RR:时间片轮转法

线程属性 － 调度参数

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr,
                               struct sched_param *param);
```

- **调度参数数据结构 － 目前只支持一个调度参数**

```
struct sched_param
{
    int sched_priority;
};
```

- **获得系统支持的线程优先权的最大和最小值**

```
 #include<sched.h>
int sched_get_priority_max(int policy)
int sched_get_priority_min(int policy)
```

线程属性 － 继承性

- **新线程是否继承创建者线程的调度策略**

```
int pthread_attr_getinheritsched(
        const pthread_attr_t *attr, int *inheritsched)
int pthread_attr_setinheritsched(
        pthread_attr_t *attr,int inheritsched)
```

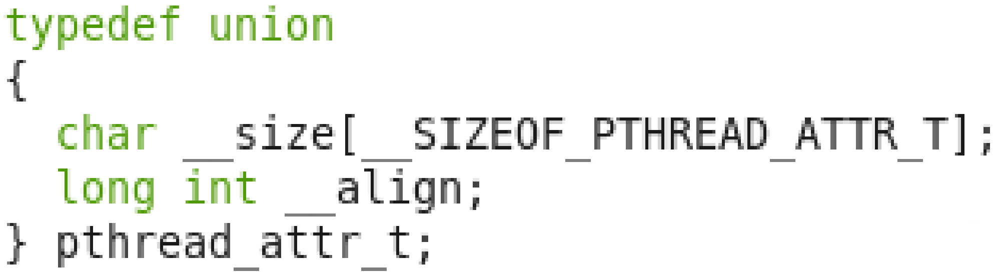
- PTHREAD_INHERIT_SCHED
  - 新线程将继承创建者线程的调度策略
  - 忽略pthread_create()调用中设置的调度属性

- PTHREAD_EXPLICIT_SCHED
  - 使用pthread_create()调用中设置的调度属性

线程属性 – 数据结构

```
typedef struct
{
    int detachstate;
    int schedpolicy;
    struct sched_param schedparam;
    int inheritsched;
    int scope;
    size_t guardsize;
    int stackaddr_set;
    void* stackaddr;
    size_t stacksize;
}pthread_attr_t
```

- 属性结构 /usr/include/bits/pthreadtypes.h



```
typedef union
{
  char __size[__SIZEOF_PTHREAD_ATTR_T];
  long int __align;
} pthread_attr_t;
```

线程属性的设置与获取（一）

属性操作注意事项

① 属性值不能直接赋值

② 使用相关函数进行操作

③ 属性初始化先于 pthread_create()

④ 去除初始化 pthread_attr_destroy()

```c
#include <stdio.h>
#include <pthread.h>
#include <sched.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>

// 显示线程属性信息
void *get_thread_sched_attr(void *arg) {
    int ipolicy, spolicy; char buf[100];
    pthread_t self;pthread_attr_t attr; struct sched_param param;

    pthread_attr_init(&attr); self = pthread_self(); pthread_getattr_np(self,&attr); //获取自己的属性
    pthread_attr_getinheritsched(&attr, &ipolicy); // 获得线程的继承性属性
    if (ipolicy == PTHREAD_EXPLICIT_SCHED) sprintf(buf,"Inheritsched: PTHREAD_EXPLICIT_SCHED;");
    if (ipolicy == PTHREAD_INHERIT_SCHED) sprintf(buf, "Inheritsched: PTHREAD_INHERIT_SCHED;");

    pthread_attr_getschedpolicy(&attr, &spolicy); // 获得线程的调度策略
    if (spolicy == SCHED_FIFO) strcat(buf, " Schedpolicy: SCHED_FIFO");
    if (spolicy == SCHED_RR) strcat(buf, " Schedpolicy: SCHED_RR");
    if (spolicy == SCHED_OTHER) strcat(buf, " Schedpolicy: SCHED_OTHER");

    int maxpri = sched_get_priority_max(spolicy);
    int minpri = sched_get_priority_min(spolicy);
    pthread_attr_getschedparam(&attr, &param);

    printf("%s\nMax priority: %u, Min priority: %u, sched_priority: %u\n\n",buf,maxpri,minpri,param.sched_priority);
    pthread_attr_destroy(&attr);
    return NULL;
}
```

线程属性的设置与获取（一）

```c
int main(int argc, char* argv[]) {
    pthread_t thread_FIFO, thread_RR, thread_OTHER;
    pthread_attr_t attr_FIFO, attr_RR, attr_OTHER;
    struct sched_param param_FIFO, param_RR, param_OTHER;
    pthread_attr_init(&attr_FIFO); /* 设置线程属性 */
    pthread_attr_setinheritsched(&attr_FIFO, PTHREAD_EXPLICIT_SCHED); // 设置线程继承性
    pthread_attr_setschedpolicy(&attr_FIFO, SCHED_FIFO); // 设置线程调度策略 及 调度参数,子线程输出自身属性
    param_FIFO.sched_priority = 5;
    pthread_attr_setschedparam(&attr_FIFO, &param_FIFO);
    pthread_create(&thread_FIFO, &attr_FIFO, get_thread_sched_attr, NULL);

    pthread_attr_init(&attr_RR);
    pthread_attr_setinheritsched(&attr_RR, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr_RR, SCHED_RR);
    param_RR.sched_priority = 10;
    pthread_attr_setschedparam(&attr_RR, &param_RR);
    pthread_create(&thread_RR, &attr_RR, get_thread_sched_attr, NULL);

    pthread_attr_init(&attr_OTHER);
    pthread_attr_setinheritsched(&attr_OTHER, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr_OTHER, SCHED_OTHER);
    // 标准时间片调度默认优先级最低，无法设置
    //param.sched_priority = 10;
    //pthread_attr_setschedparam(&attr_OTHER, &param);
    pthread_create(&thread_OTHER, &attr_OTHER, get_thread_sched_attr, NULL);

    pthread_join(thread_FIFO, NULL);
    pthread_join(thread_RR, NULL);
    pthread_join(thread_OTHER, NULL);
    pthread_attr_destroy(&attr_FIFO);
    pthread_attr_destroy(&attr_RR);
    pthread_attr_destroy(&attr_OTHER);
}
```

属性操作注意事项

① 属性值不能直接赋值

② 使用相关函数进行操作

③ 属性初始化先于 pthread_create()

④ 去除初始化 pthread_attr_destroy()

线程属性的设置与获取（一）

属性操作注意事项

① 属性不能直接赋值

② 使用相关函数进行操作

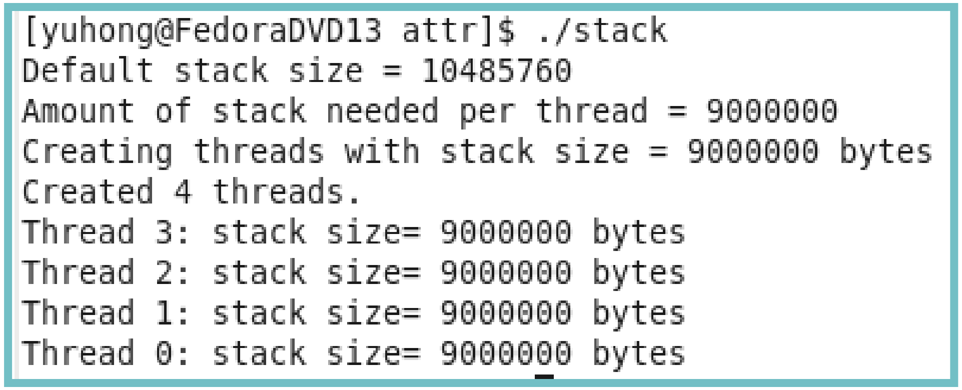
③ 属性初始化先于 pthread_create()

④ 去除初始化 pthread_attr_destroy()

```
[szu@taishan02-vm-10 threads]$ gcc -o attr attr.c -lpthread
attr.c: 在函数 ‘get_thread_sched_attr’中:
attr.c:13:54: 警告： implicit declaration of function ‘pthread_getattr_np’; did you mean ‘pthread_attr_init’? [-Wimplicit-function-declaration]
    pthread_attr_init(&attr); self = pthread_self(); pthread_getattr_np(self,&attr); //获取自己的属性
                                     ^~~~~~~~~~~~~~~~~
                                     pthread_attr_init
[szu@taishan02-vm-10 threads]$ sudo ./attr
Inheritsched: PTHREAD_EXPLICIT_SCHED; Schedpolicy: SCHED_FIFO
Max priority: 99, Min priority: 1, sched_priority: 5

Inheritsched: PTHREAD_EXPLICIT_SCHED; Schedpolicy: SCHED_RR
Max priority: 99, Min priority: 1, sched_priority: 10

Inheritsched: PTHREAD_EXPLICIT_SCHED; Schedpolicy: SCHED_OTHER
Max priority: 0, Min priority: 0, sched_priority: 0
```

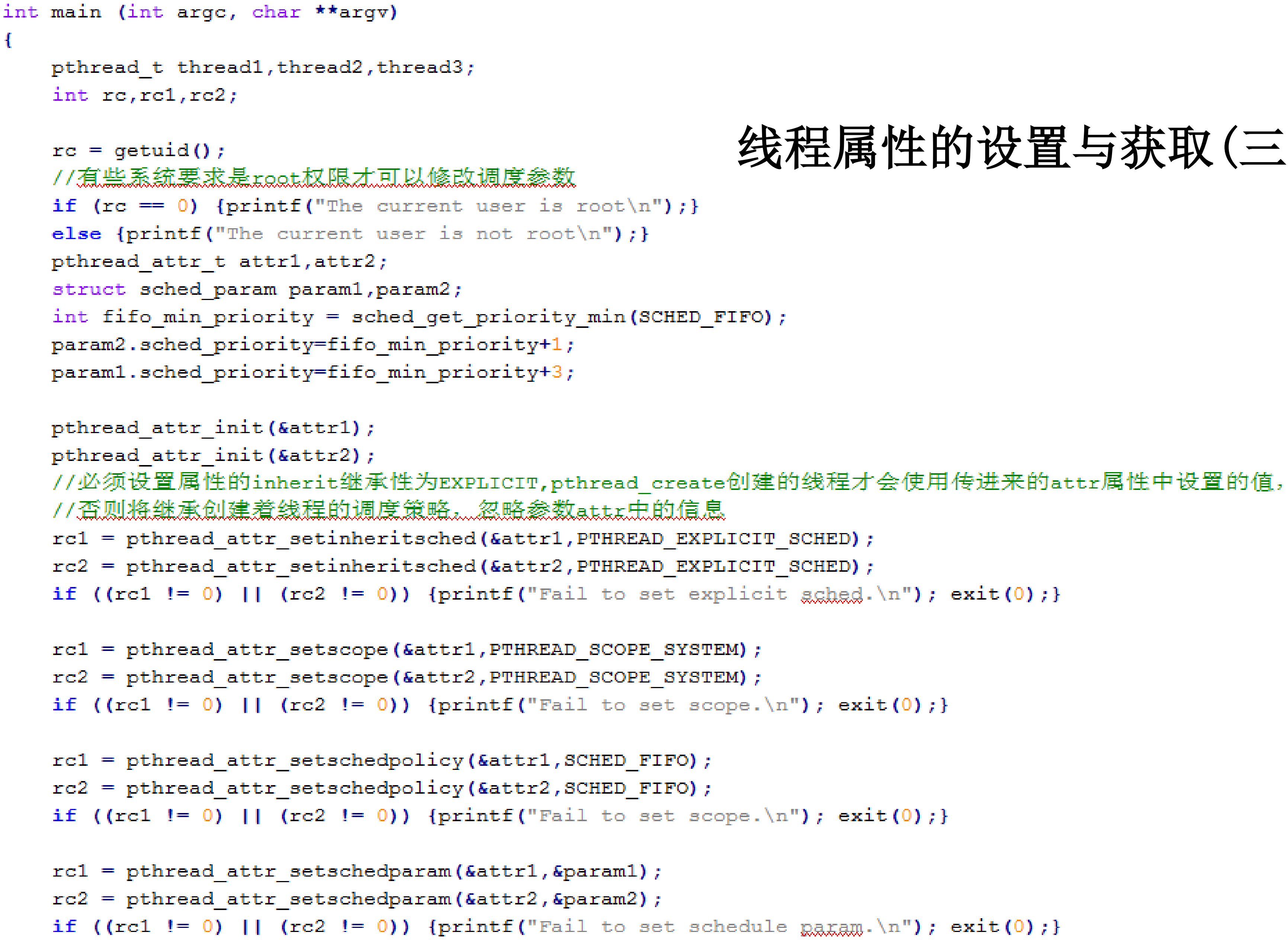
线程属性的设置与获取(二)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;
void *dowork(void *threadid)
{
        double A[N][N];
        int i,j;
        long tid;
        size_t mystacksize;
        tid =(long)threadid;
        pthread_attr_getstacksize(&attr,&mystacksize);
        printf("Thread %ld: stack size= %li bytes \n",tid,mystacksize);
        for (i=0; i < N; i++)
                for (j=0;j<N;j++)
                        A[i][j]=((i*j)/3.452)+(N-i);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        pthread_t threads[NTHREADS];
        size_t  stacksize;
        int rc;
        long t;
        pthread_attr_init(&attr);
        pthread_attr_getstacksize(&attr,&stacksize);
        printf("Default stack size = %li\n",stacksize);
        stacksize = sizeof(double)*N*N+MEGEXTRA;
        printf("Amount of stack needed per thread = %li\n",stacksize);
        pthread_attr_setstacksize(&attr,stacksize);
        printf("Creating threads with stack size = %li bytes\n",stacksize);
        for (t=0; t<NTHREADS;t++){
                rc = pthread_create(&threads[t],&attr,dowork,(void *)t);
                if (rc){
                        printf("ERROR: return code from pthread_create() is %d\n",rc);
                        exit(-1);
                }
        }
        printf("Created %ld threads.\n",t);
        pthread_exit(NULL);
}
```



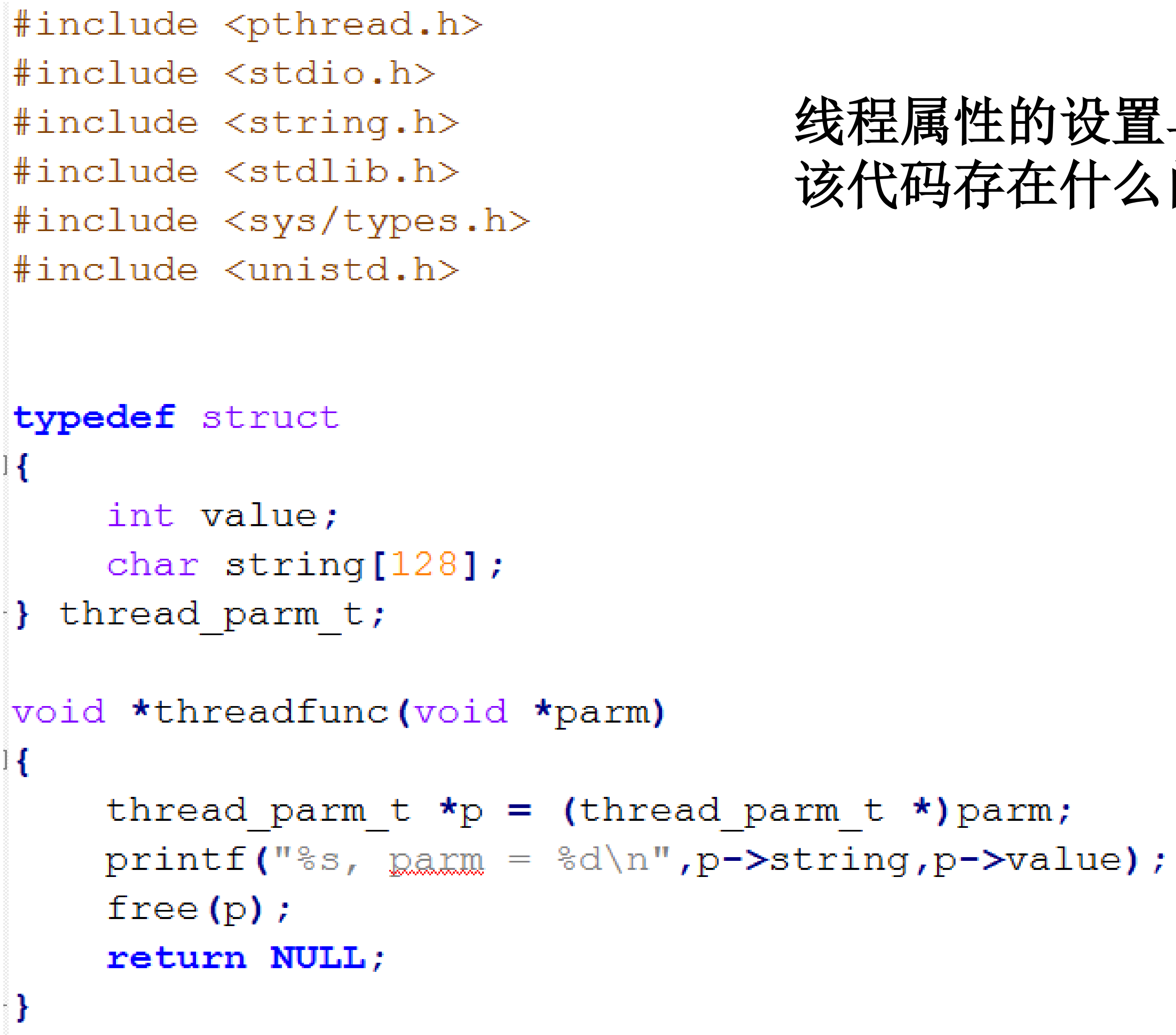
```
[yuhong@FedoraDVD13 attr]$ ./stack
Default stack size = 10485760
Amount of stack needed per thread = 9000000
Creating threads with stack size = 9000000 bytes
Created 4 threads.
Thread 3: stack size= 9000000 bytes
Thread 2: stack size= 9000000 bytes
Thread 1: stack size= 9000000 bytes
Thread 0: stack size= 9000000 bytes
```

线程属性的设置与获取(三)



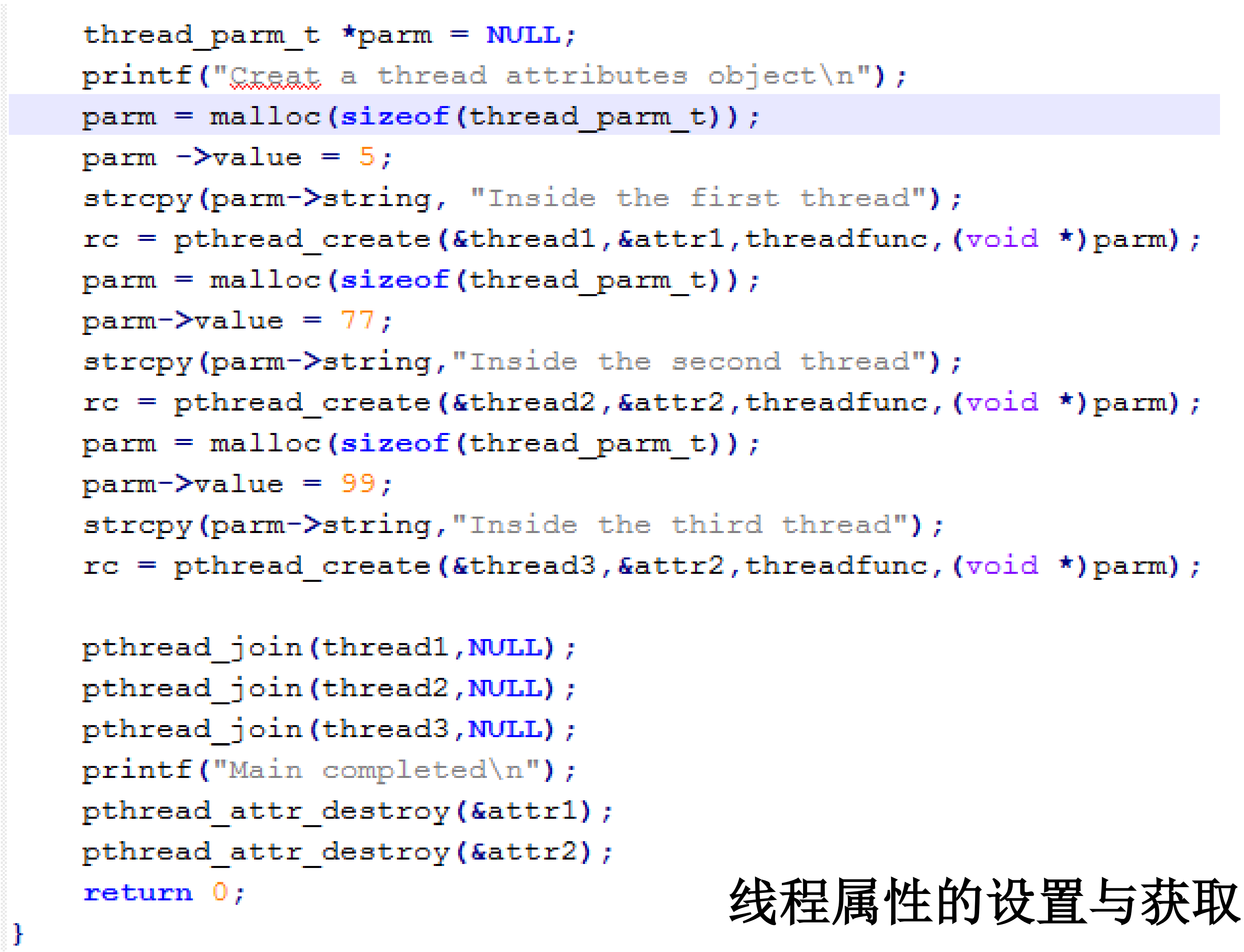
```c
int main (int argc, char **argv)
{
    pthread_t thread1,thread2,thread3;
    int rc,rc1,rc2;

    rc = getuid();
    //有些系统要求是root权限才可以修改调度参数
    if (rc == 0) {printf("The current user is root\n");}
    else {printf("The current user is not root\n");}
    pthread_attr_t attr1,attr2;
    struct sched_param param1,param2;
    int fifo_min_priority = sched_get_priority_min(SCHED_FIFO);
    param2.sched_priority=fifo_min_priority+1;
    param1.sched_priority=fifo_min_priority+3;

    pthread_attr_init(&attr1);
    pthread_attr_init(&attr2);
    //必须设置属性的inherit继承性为EXPLICIT,pthread_create创建的线程才会使用传进来的attr属性中设置的值,
    //否则将继承创建者线程的调度策略, 忽略参数attr中的信息
    rc1 = pthread_attr_setinheritsched(&attr1,PTHREAD_EXPLICIT_SCHED);
    rc2 = pthread_attr_setinheritsched(&attr2,PTHREAD_EXPLICIT_SCHED);
    if ((rc1 != 0) || (rc2 != 0)) {printf("Fail to set explicit sched.\n"); exit(0);}

    rc1 = pthread_attr_setscope(&attr1,PTHREAD_SCOPE_SYSTEM);
    rc2 = pthread_attr_setscope(&attr2,PTHREAD_SCOPE_SYSTEM);
    if ((rc1 != 0) || (rc2 != 0)) {printf("Fail to set scope.\n"); exit(0);}

    rc1 = pthread_attr_setschedpolicy(&attr1,SCHED_FIFO);
    rc2 = pthread_attr_setschedpolicy(&attr2,SCHED_FIFO);
    if ((rc1 != 0) || (rc2 != 0)) {printf("Fail to set scope.\n"); exit(0);}

    rc1 = pthread_attr_setschedparam(&attr1,&param1);
    rc2 = pthread_attr_setschedparam(&attr2,&param2);
    if ((rc1 != 0) || (rc2 != 0)) {printf("Fail to set schedule param.\n"); exit(0);}
```

线程属性的设置与获取(四)
该代码存在什么问题？



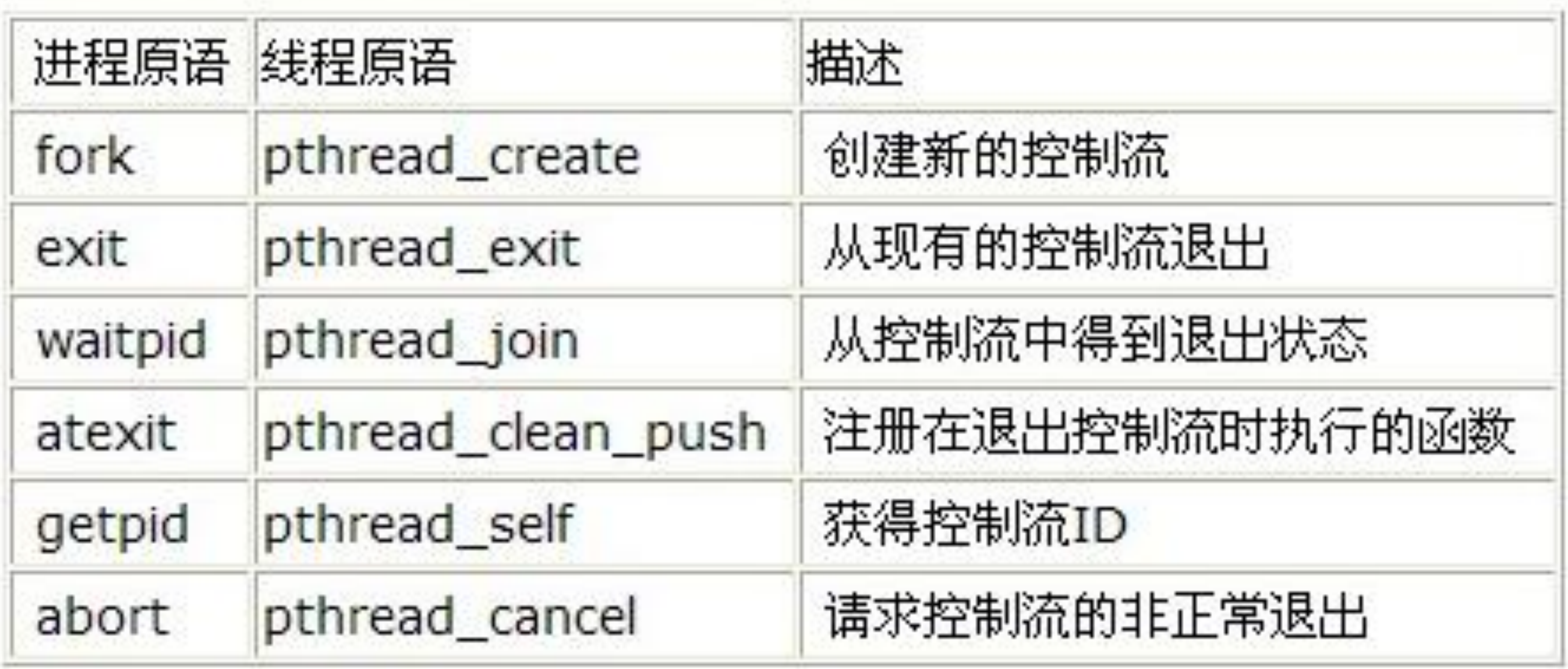
```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>


typedef struct
{
    int value;
    char string[128];
} thread_parm_t;

void *threadfunc(void *parm)
{
    thread_parm_t *p = (thread_parm_t *)parm;
    printf("%s, parm = %d\n",p->string,p->value);
    free(p);
    return NULL;
}
```

线程属性的设置与获取(四)



```
    thread_parm_t *parm = NULL;
    printf("Creat a thread attributes object\n");
    parm = malloc(sizeof(thread_parm_t));
    parm ->value = 5;
    strcpy(parm->string, "Inside the first thread");
    rc = pthread_create(&thread1,&attr1,threadfunc,(void *)parm);
    parm = malloc(sizeof(thread_parm_t));
    parm->value = 77;
    strcpy(parm->string,"Inside the second thread");
    rc = pthread_create(&thread2,&attr2,threadfunc,(void *)parm);
    parm = malloc(sizeof(thread_parm_t));
    parm->value = 99;
    strcpy(parm->string,"Inside the third thread");
    rc = pthread_create(&thread3,&attr2,threadfunc,(void *)parm);

    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);
    printf("Main completed\n");
    pthread_attr_destroy(&attr1);
    pthread_attr_destroy(&attr2);
    return 0;
}
```

Linux进程与线程序原语比较



进程原语	线程原语	描述
fork	pthread_create	创建新的控制流
exit	pthread_exit	从现有的控制流退出
waitpid	pthread_join	从控制流中得到退出状态
atexit	pthread_clean_push	注册在退出控制流时执行的函数
getpid	pthread_self	获得控制流ID
abort	pthread_cancel	请求控制流的非正常退出