

# Threads

## Operating Systems

郑贵锋 博士  
中山大学计算机学院  
zhenggf@mail.sysu.edu.cn  
[https://gitee.com/code\\_sysu](https://gitee.com/code_sysu)



### Overview

2 / 74

#### ■ 目录

- 概述
- 多核编程
- 用户和内核级线程
- 多线程模型
- 线程库
- 隐式线程化实现
- 线程问题
  - fork()和exec()系统调用的语义
  - 信号处理
  - 线程取消
  - 线程本地存储
  - 调度程序激活
- Linux clone()
- 示例：Windows线程



## ■ 线程本地存储

- 线程本地存储（TLS）允许每个线程拥有自己的数据副本。
- 当我们无法控制创建线程的进程时，TLS非常有用。
  - 我们不能向创建的线程传递任何参数。
  - 例如，当使用线程池时。
- TLS不同于局部变量。
  - 局部变量仅在单个函数调用期间可见。
  - TLS的可见性跨越函数调用。
- 与静态数据类似：
  - TLS对于每个线程都是唯一的。
- TLS的实现
  - `__thread` `int` `tlsvar`; /\* 每个线程的tlsvar; 由语言编译器解释，  
TLS的语言级解决方案\*/
  - 或使用`pthread_key_create`实现



## ■ 线程本地存储

- 使用`__thread`实现TLS
  - 算法. 14-1-tls-thread.c (1)
 

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/syscall.h>
#include <pthread.h>
#define gettid() syscall(__NR_gettid)

__thread int tlsvar = 0; /* tlsvar for each thread; interpreted by language compiler,
a language level solution to 线程本地存储 */

static void* thread_worker(void* arg)
{
    char *param = (char *)arg;
    int randomcount;

    for (int i = 0; i < 5; ++i) {
        randomcount = rand() % 100000;
        for (int k = 0; k < randomcount; k++) ;
        printf("%s%d, tlsvar = %d\n", param, gettid(), tlsvar);
        tlsvar++; /* each thread has its local tlsvar */
    }

    pthread_exit(0);
}
```



## 线程本地存储

### 使用 `__thread` 实现 TLS

#### 算法. 14-1-tls-thread.c (2)

```
int main(void)
{
    pthread_t tid1, tid2;
    char para1[] = "          ";
    char para2[] = "          ";
    int randomcount;

    pthread_create(&tid1, NULL, &thread_worker, para1);
    pthread_create(&tid2, NULL, &thread_worker, para2);

    printf("parent          tid1          tid2\n");
    printf("=====          =====          =====\n");

    for (int i = 0; i < 5; ++i) {
        randomcount = rand() % 100000;
        for (int k = 0; k < randomcount; k++) ;
        printf("%ld, tlsvar = %d\n", gettid(), tlsvar);
        tlsvar++; /* main- thread has its local tlsvar */
    }

    sleep(1);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}
```



## 线程本地存储

### 使用 `__thread` 实现 TLS

#### 算法.14-1-tls-thread.c (2)

```
int main(void)
{
    pthread_t tid1, tid2;
    char para1[] = "          ";
    char para2[] = "          ";
    int randomcount;

    pthread_create(&tid1, NULL, &thread_worker, para1);
    pthread_create(&tid2, NULL, &thread_worker, para2);

    printf("parent          tid1          tid2\n");
    printf("=====          =====          =====\n");

    for (int i = 0; i < 5; ++i) {
        randomcount = rand() % 100000;
        for (int k = 0; k < randomcount; k++) ;
        printf("%ld, tlsvar = %d\n", gettid(), tlsvar);
        tlsvar++; /* main- thread has its local tlsvar */
    }

    sleep(1);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}
```

Terminal output:

```
isscg@ubuntu:/mnt/os-2020$ gcc alg.14-1-tls-thread.c -pthread
isscg@ubuntu:/mnt/os-2020$ ./a.out
parent          tid1          tid2
=====          =====          =====
20914, tlsvar = 0
20914, tlsvar = 1
20914, tlsvar = 2
20914, tlsvar = 3
20914, tlsvar = 4
20915, tlsvar = 0
20915, tlsvar = 1
20915, tlsvar = 2
20915, tlsvar = 3
20915, tlsvar = 4
20916, tlsvar = 0
20916, tlsvar = 1
20916, tlsvar = 2
20916, tlsvar = 3
20916, tlsvar = 4
isscg@ubuntu:/mnt/os-2020$
```



## ■ 线程本地存储

- 由 `pthread_key_create` 实现的 TLS

```
typedef unsigned int pthread_key_t
```

- 线程特定数据(TSD)的键
- 检查 `sysdeps/x86/bits/pthreadtypes.h` 中 TSD 的 **上限**
- 一个进程最多可以创建 1024 个密钥

```
PTHREAD_KEYS_MAX = 1024
```

```
int pthread_key_create(pthread_key_t *key,
                      void (*destructor)(void*));
/* tls_key 是一个标识符, 而不是一个数据结构 */
int pthread_setspecific(pthread_key_t key,
                      const void *value);
void *pthread_getspecific(pthread_key_t key);
int pthread_key_delete(pthread_key_t key);
```



## ■ 线程本地存储

- 由 `pthread_key_create` 实现的 TLS

- 算法. 14-2-tls-pthread-key-1.c (1)

```
/* gcc -pthread */
static pthread_key_t log_key;
/* 每个线程将全局 log_key 与线程的一个局部变量相关联, 通过使用 log_key, 关联的局部变量的行为类似于全局变量 */
void write_log(const char *msg);

static void *thread_worker(void *args)
{
    static int thcnt = 0;
    char fname[64], msg[64];
    FILE *fp_log; /* 一个局部变量 */

    sprintf(fname, "log/thread-%d.log", ++thcnt); /* ./log 目录必须存在 */
    fp_log = fopen(fname, "w");
    if(!fp_log) {
        printf("%s\n", fname);
        perror("fopen()");
        return NULL;
    }

    pthread_setspecific(log_key, fp_log); /* fp_log 与全局变量 log_key 关联 */

    sprintf(msg, "Here is %s\n", fname);
    write_log(msg);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <pthread.h>
#define gettid() syscall(__NR_gettid)
```



## ■ 线程本地存储

### ■ 由pthread\_key\_create实现的TLS

#### ■ 算法. 14-2-tls-pthread-key-1.c (2)

```
void write_log(const char *msg)
{
    FILE *fp_log;
    fp_log = (FILE *)pthread_getspecific(log_key); /* fp_log 从全局变量log_key获得 */
    fprintf(fp_log, "writing msg: %s\n", msg);
    printf("log_key = %d, tid = %ld, address of fp_log %p\n", log_key, gettid(),
        fp_log);
}

void close_log_file(void* log_file) /* the destructor */
{
    fclose((FILE*)log_file);
}
```



## ■ 线程本地存储

### ■ 由pthread\_key\_create实现的TLS

#### ■ 算法. 14-2-tls-pthread-key-1.c (3)

```
int main(void)
{
    const int n = 5;
    pthread_t tids[n];
    pthread_key_create(&log_key, &close_log_file);
    // or: pthread_key_create(&log_key, NULL); /* NULL for default destructor */

    printf("====tids and TLS variable addresses =====\n");
    for(int i = 0; i < n; i++) {
        pthread_create(&tids[i], NULL, &thread_worker, NULL);
    }

    for(int i = 0; i < n; i++) {
        pthread_join(tids[i], NULL);
    }

    pthread_key_delete(log_key); /* delete the key */

    printf("\ncommand: ls -l ./log\n");
    system("ls -l ./log"); /* 列出所有打开目录./log的实例 */
    printf("\ncommand: cat ./log/thread-1.log ./log/thread-5.log\n");
    system("cat ./log/thread-1.log ./log/thread-5.log");

    return 0;
}
```



## 线程本地存储

- 由 `pthread_key_create` 实现的 TLS
- 算法. 14-2-tls-pthread-key-1.c (3)

```

iisscg@ubuntu:/mnt/os-2020$ gcc alg.14-2-tls-pthread-key-1.c -pthread
iisscg@ubuntu:/mnt/os-2020$ ./a.out
=====tids and TLS variable addresses =====
log_key = 0, tid = 47249, address of fp_log 0x7f2a20000b20
log_key = 0, tid = 47250, address of fp_log 0x7f2a18000b20
log_key = 0, tid = 47251, address of fp_log 0x7f2a1c000b20
log_key = 0, tid = 47253, address of fp_log 0x7f2a14000b20
log_key = 0, tid = 47252, address of fp_log 0x7f2a10000b20

command: lsof +d ./log
writing msg: Here is log/thread-1.log
writing msg: Here is log/thread-5.log

iisscg@ubuntu:/mnt/os-2020$
    return 0;
}

```

不同线程的不同的fp\_log  
关联相同的log\_key



## 线程本地存储

- 由 `pthread_key_create` 实现的 TLS
- 算法. 14-2-tls-pthread-key-1.c (3)

```

iisscg@ubuntu:/mnt/os-2020$ gcc alg.14-2-tls-pthread-key-1.c -pthread
iisscg@ubuntu:/mnt/os-2020$ ./a.out
=====tids and TLS variable addresses =====
log_key = 0, tid = 47249, address of fp_log 0x7f2a20000b20
log_key = 0, tid = 47250, address of fp_log 0x7f2a18000b20
log_key = 0, tid = 47251, address of fp_log 0x7f2a1c000b20
log_key = 0, tid = 47253, address of fp_log 0x7f2a14000b20
log_key = 0, tid = 47252, address of fp_log 0x7f2a10000b20

command: lsof +d ./log
writing msg: Here is log/thread-1.log
writing msg: Here is log/thread-5.log

iisscg@ubuntu:/mnt/os-2020$
    return 0;
}

```

所有log文件都被析构函数“void  
close\_log\_file()”关闭了



## ■ 线程本地存储

### ■ 由 `pthread_key_create` 实现的 TLS

#### ■ 算法. 14-3-tls-pthread-key-2.c (1): 绑定数据结构

```
/* gcc -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <malloc.h>
#include <pthread.h>
#define gettid() syscall(__NR_gettid)

static pthread_key_t tls_key; /* 静态全局变量 */

void print_msg1(void);
void print_msg2(void);
static void *thread_func1(void *);
static void *thread_func2(void *);

/* msg1 和 msg2 有不同的数据结构 */
struct msg_struct1 {
    char stuno[9];
    char stuname[20];
};
struct msg_struct2 {
    int stuno;
    char nationality[20];
    char stuname[20];
};
```



## ■ 线程本地存储

### ■ 由 `pthread_key_create` 实现的 TLS

#### ■ 算法. 14-3-tls-pthread-key-2.c (2): 绑定数据结构

```
int main(void)
{
    pthread_t ptid1, ptid2;

    pthread_key_create(&tls_key, NULL);

    printf("    msg1 -->>      stuno    stuname");
    printf("    msg2 -->>      stuno    stuname  nationality\n");
    printf("=====  =====  =====");
    printf("=====  =====  =====\n");

    pthread_create(&ptid1, NULL, &thread_func1, NULL);
    pthread_create(&ptid2, NULL, &thread_func2, NULL);

    pthread_join(ptid1, NULL);
    pthread_join(ptid2, NULL);

    pthread_key_delete(tls_key);
    return EXIT_SUCCESS;
}
```



## ■ 线程本地存储

### ■ 由pthread\_key\_create实现的TLS

#### ■ 算法. 14-3-tls-pthread-key-2.c (3): 绑定数据结构

```
static void *thread_func1(void *args)
{
    struct msg_struct1 ptr[5]; /* 线程栈的局部变量 */
    printf("thread_func1: tid = %ld  ptr = %p\n", gettid(), ptr);
    pthread_setspecific(tls_key, ptr); /* 绑定ptr到全局tls_key */
    sprintf(ptr[0].stuno, "18000001"); sprintf(ptr[0].stuname, "Alex");
    sprintf(ptr[4].stuno, "18000005"); sprintf(ptr[4].stuname, "Michael");
    print_msg1();

    pthread_exit(0);
}

void print_msg1(void)
{
    int randomcount;
    struct msg_struct1 *ptr = (struct msg_struct1 *)pthread_getspecific(tls_key);
    printf("print_msg1:  tid = %ld  ptr = %p\n", gettid(), ptr);
    for (int i = 1; i < 6; i++) {
        randomcount = rand() % 10000;
        for (int k = 0; k < randomcount; k++) ;
        printf("tid = %ld  i = %2d  %s  %s\n",
               gettid(), i, ptr->stuno, 8, 8, ptr->stuname);
        ptr++;
    }
    return;
}
```



## ■ 线程本地存储

### ■ 由pthread\_key\_create实现的TLS

#### ■ 算法. 14-3-tls-pthread-key-2.c (4): 绑定数据结构

```
static void *thread_func2(void *args)
{
    struct msg_struct2 *ptr;
    ptr = (struct msg_struct2 *)malloc(5*sizeof(struct msg_struct2)); /* 保存在进程堆 */
    printf("thread_func2: tid = %ld  ptr = %p\n", gettid(), ptr);
    pthread_setspecific(tls_key, ptr); /* 绑定ptr到全局tls_key */
    ptr->stuno = 19000001; sprintf(ptr->stuname, "Bob");
    sprintf(ptr->nationality, "United Kingdom");
    (ptr+2)->stuno = 19000003; sprintf((ptr+2)->stuname, "John");
    sprintf((ptr+2)->nationality, "United States");
    print_msg2();
    free(ptr); ptr = NULL; pthread_exit(0);
}

void print_msg2(void)
{
    int randomcount;
    struct msg_struct2 *ptr = (struct msg_struct2 *)pthread_getspecific(tls_key);
    printf("print_msg2:  tid = %ld  ptr = %p\n", gettid(), ptr);
    for (int i = 1; i < 6; i++) {
        randomcount = rand() % 10000;
        for (int k = 0; k < randomcount; k++) ;
        printf("
        print_msg2:  tid = %ld  i = %2d  %d  %s  %s\n",
               gettid(), i, ptr->stuno, 8, 8, ptr->stuname, ptr->nationality);
        ptr++;
    }
    return;
}
```





## 线程本地存储

- 由 `pthread_key_create` 实现的 TLS
  - 算法. 14-3-tls-pthread-key-2.c (4): 绑定数据结构

```

isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-3-tls-pthread-key-2.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
msg1 -->          stuno      stuname      msg2 -->          stuno      stuname      nationality
=====
thread_func1: tid = 47333 ptr = 0x7fa8b549be50
print_msg1:  tid = 47333 ptr = 0x7fa8b549be50
tid = 47333 i = 1      18000001      Alex
tid = 47333 i = 2
tid = 47333 i = 3
tid = 47333 i = 4
tid = 47333 i = 5      18000005      Michael
thread_func2: tid = 47334 ptr = 0x7fa8b0000b20
print_msg2:  tid = 47334 ptr = 0x7fa8b0000b20
tid = 47334 i = 1      19000001      Bob      United Kingdom
tid = 47334 i = 2      0
tid = 47334 i = 3      19000003      John     United States
tid = 47334 i = 4      0
tid = 47334 i = 5      0
isscgy@ubuntu:/mnt/os-2020$
    print(
    printf("tid = %ld i = %2d %d %*.s %s\n",
           getpid(), i, ptr->stuno, 8, 8, ptr->stuname, ptr->nationality);
    ptr++;
  }
  return;
}

```

不同线程的不同数据结构指针ptr  
绑定到相同的全局tls\_key



## 线程本地存储

- 由 `pthread_key_create` 实现的 TLS
  - 算法. 14-4-tls-pthread-key-3.c (1): 不同的线程函数

```

/* gcc -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <pthread.h>
#define getpid() syscall(__NR_gettid)

static pthread_key_t tls_key; /* static global */

void print_msg(void);
static void *thread_func1(void *);
static void *thread_func2(void *);

struct msg_struct {
    char pos[80];
    char stuno[9];
    char stuname[20];
};

```



## 线程本地存储

### 由 `pthread_key_create` 实现的 TLS

#### 算法. 14-4-tls-pthread-key-3.c (2): 不同的线程函数

```
int main(void)
{
    pthread_t ptid1, ptid2;

    pthread_key_create(&tls_key, NULL);

    printf("      msg1 -->      stuno      stuname      msg2 -->      stuno
stuname\n");
    printf("=====      =====      =====      =====
=====\\n");

    pthread_create(&ptid1, NULL, &thread_func1, NULL);
    pthread_create(&ptid2, NULL, &thread_func2, NULL);

    pthread_join(ptid1, NULL);
    pthread_join(ptid2, NULL);

    pthread_key_delete(tls_key);
    return EXIT_SUCCESS;
}
```



## 线程本地存储

### 由 `pthread_key_create` 实现的 TLS

#### 算法. 14-4-tls-pthread-key-3.c (3): 不同的线程函数

```
static void *thread_func1(void *args)
{
    struct msg_struct ptr[5]; /* in thread stack */
    printf("thread_func1: tid = %ld  ptr = %p\\n", gettid(), ptr);

    pthread_setspecific(tls_key, ptr); /* binding its tls_key to address of ptr */
    for (int i = 0; i < 5; i++) {
        sprintf(ptr[i].pos, " ");
        sprintf(ptr[i].stuno, " ");
        sprintf(ptr[i].stuname, " ");
    }
    sprintf(ptr[0].stuno, "18000001");
    sprintf(ptr[0].stuname, "Alex");
    sprintf(ptr[4].stuno, "18000005");
    sprintf(ptr[4].stuname, "Michael");
    print_msg(); /* thread_func1 and thread_func2 call the same print_msg() */
    pthread_exit(0);
}
```



## ■ 线程本地存储

### ■ 由pthread\_key\_create实现的TLS

#### ■ 算法. 14-4-tls-pthread-key-3.c (4):不同的线程函数

```
static void *thread_func2(void *args)
{
    struct msg_struct ptr[5]; /* in thread stack */
    printf("thread_func2: tid = %ld  ptr = %p\n", gettid(), ptr);

    pthread_setspecific(tls_key, ptr); /* binding its tls_key to address of ptr */

    for (int i = 0; i < 5; i++) {
        sprintf(ptr[i].pos, "
        sprintf(ptr[i].stuno, "
        sprintf(ptr[i].stuname, "
    }
    sprintf(ptr[0].stuno, "19000001");
    sprintf(ptr[0].stuname, "Bob");
    sprintf(ptr[2].stuno, "19000003");
    sprintf(ptr[2].stuname, "John");
    print_msg(); /* thread_func1 and thread_fun2 call the same print_msg() */
    pthread_exit(0);
}
```



## ■ 线程本地存储

### ■ 由pthread\_key\_create实现的TLS

#### ■ 算法. 14-4-tls-pthread-key-3.c (5):不同的线程函数

```
void print_msg(void)
{
    int randomcount;

    struct msg_struct* ptr = (struct msg_struct *)pthread_getspecific(tls_key);
    /* ptr 由调用本函数的线程决定 */
    printf("print_msg:  tid = %ld  ptr = %p\n", gettid(), ptr);

    for (int i = 1; i < 6; i++) {
        randomcount = rand() % 10000;
        for (int k = 0; k < randomcount; k++) ;
        printf("%stid = %ld  i = %2d  %s  %*.s\n", ptr->pos, gettid(), i, ptr->stuno,
        8, 8, ptr->stuname);
        ptr++;
    }

    return;
}
```



## 线程本地存储

- 由 `pthread_key_create` 实现的 TLS

- 算法. 14-4-tls-pthread-key-3.c (5): 不同的线程函数

```
void print_msg(void)
{
    struct msg_struct *p;
    pthread_key_t key;
    pthread_once_t once = PTHREAD_ONCE_T;
    pthread_key_create(&key, &print_msg, &once);
    pthread_once(&once, &print_msg);
    p = pthread_getspecific(key);
    printf("msg1 -->> stuno stuname\n");
    printf("===== stuno stuname\n");
    printf("thread_func1: tid = 47508 ptr = 0x7fd4fe4b3cc0\n");
    printf("print_msg: tid = 47508 ptr = 0x7fd4fe4b3cc0\n");
    printf("thread_func2: tid = 47509 ptr = 0x7fd4fdbc2cc0\n");
    printf("print_msg: tid = 47509 ptr = 0x7fd4fdbc2cc0\n");
    printf("tid = 47508 i = 1 18000001 Alex tid = 47509 i = 1 19000001 Bob\n");
    printf("tid = 47508 i = 2 tid = 47509 i = 2\n");
    printf("tid = 47508 i = 3 tid = 47509 i = 3 19000003 John\n");
    printf("tid = 47508 i = 4 tid = 47509 i = 4\n");
    printf("tid = 47508 i = 5 18000005 Michael tid = 47509 i = 5\n");
}
```

线程函数func1、func2都调用了相同的 `print_msg()`，针对不同的线程，`print_msg()` 有两个 `ptr` 变量副本



## 线程本地存储

- 由 `pthread_key_create` 实现的 TLS

- 算法. 14-5-tls-pthread-key-4.c (1): 在子函数里设置 `tls_key`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <malloc.h>
#include <pthread.h>
#define getpid() syscall(__NR_gettid)
static pthread_key_t tls_key; /* static global */
static void *thread_func(void *);
void thread_data1(void);
void thread_data2(void);

struct msg_struct {
    char stuno[9];
    char stuname[20];
};

int main(void)
{
    pthread_t ptid;
    pthread_key_create(&tls_key, NULL);
    pthread_create(&ptid, NULL, &thread_func, NULL);
    pthread_join(ptid, NULL);
    pthread_key_delete(tls_key);
    return EXIT_SUCCESS;
}
```



## Threading Issues

25 / 74

## ■ 线程本地存储

## ■ 由pthread\_key\_create实现的TLS

## ■ 算法. 14-5-tls-pthread-key-4.c (2): 在子函数里设置tls\_key

```
static void *thread_func(void *args)
{
    struct msg_struct *ptr;

    thread_data1();
    ptr = (struct msg_struct *)pthread_getspecific(tls_key);
    /* get ptr from thread_data1() */
    perror("pthread_getspecific()");
    printf("ptr from thread_data1() in thread_func(): %p\n", ptr);
    for (int i = 1; i < 6; i++) {
        printf("tid = %ld i = %2d %s %*.s\n", gettid(), i, (ptr+i-1)->stuno, 8, 8,
(ptr+i-1)->stuname);
    }
    thread_data2();
    ptr = (struct msg_struct *)pthread_getspecific(tls_key);
    /* get ptr from thread_data2() */
    perror("pthread_getspecific()");
    printf("ptr from thread_data2() in thread_func(): %p\n", ptr);
    for (int i = 1; i < 6; i++) {
        printf("tid = %ld i = %2d %s %*.s\n", gettid(), i, (ptr+i-1)->stuno, 8, 8,
(ptr+i-1)->stuname);
    }
    free(ptr);
    ptr = NULL;
    pthread_exit(0);
}
```



## Threading Issues

26 / 74

## ■ 线程本地存储

## ■ 由pthread\_key\_create实现的TLS

## ■ 算法. 14-5-tls-pthread-key-4.c (3): 在子函数里设置tls\_key

```
void thread_data1(void)
{
    struct msg_struct ptr[5]; /* 保存在线程栈内存里 */
    pthread_setspecific(tls_key, ptr); /* binding the tls_key to address of ptr */
    printf("ptr in thread_data1(): %p\n", ptr);

    for (int i = 0; i < 5; i++) {
        sprintf(ptr[i].stuno, " ");
        sprintf(ptr[i].stuname, " ");
    }
    sprintf(ptr[0].stuno, "19000001");
    sprintf(ptr[0].stuname, "Bob");
    sprintf(ptr[2].stuno, "19000003");
    sprintf(ptr[2].stuname, "John");

    return;
    /* 当thread_data1()返回时, 线程栈空间被释放, 造成数据丢失 */
}
```



## ■ 线程本地存储

### ■ 由pthread\_key\_create实现的TLS

#### ■ 算法. 14-5-tls-pthread-key-4.c (4): 在子函数里设置tls\_key

```
void thread_data2(void)
{
    struct msg_struct *ptr;
    ptr = (struct msg_struct *)malloc(5*sizeof(struct msg_struct));
    /* 在进程堆里分配内存 */
    pthread_setspecific(tls_key, ptr); /* binding the tls_key to address of ptr */
    printf("ptr in thread_data2(): %p\n", ptr);

    for (int i = 0; i < 5; i++) {
        sprintf(ptr[i].stuno, "          ");
        sprintf(ptr[i].stuname, "          ");
    }
    sprintf(ptr->stuno, "19000001");
    sprintf(ptr->stuname, "Bob");
    sprintf((ptr+2)->stuno, "19000003");
    sprintf((ptr+2)->stuname, "John");

    return;
    /* 只要ptr不被主动释放, 进程堆空间继续有效 */
    /* 如果返回前free(ptr), 则空间被释放, 将造成数据丢失 */
    /* 需要在线程函数thread_func里释放空间, 否则会造成内存泄漏! */
}
```



## ■ 线程本地存储

### ■ 由pthread\_key\_create实现的TLS

#### ■ 算法. 14-5-tls-pthread-key-4.c (4): 在子函数里设置tls\_key

```
isscg@ubuntu:/mnt/os-2020$ gcc alg.14-5-tls-pthread-key-4.c -pthread
isscg@ubuntu:/mnt/os-2020$ ./a.out
ptr in thread_data1(): 0x7ffb277d9e10
pthread_getspecific(): Success
ptr from thread_data1() in thread_func(): 0x7ffb277d9e10
tid = 47607 i = 1
tid = 47607 i = 2
tid = 47607 i = 3
tid = 47607 i = 4
tid = 47607 i = 5
ptr in thread_data2(): 0x7ffb20001570
pthread_getspecific(): Success
ptr from thread_data2() in thread_func(): 0x7ffb20001570
tid = 47607 i = 1 19000001 Bob
tid = 47607 i = 2
tid = 47607 i = 3 19000003 John
tid = 47607 i = 4
tid = 47607 i = 5
isscg@ubuntu:/mnt/os-2020$
```

两种情况下全局tls\_key  
都能继续工作, 正常得  
到关联的本地存储。



## ■ 线程本地存储

- 由 `pthread_key_create` 实现的 TLS
  - 算法. 14-5-tls-pthread-key-4.c (4): 在子函数里设置 `tls_key`

```

isscgy@ubuntu:/mnt/os-2020$ gcc alg.14-5-tls-pthread-key-4.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
ptr in thread_data1(): 0x7ffb277d9e10
pthread_getspecific(): Success
ptr from thread_data1() in thread_func(): 0x7ffb277d9e10
tid = 47607 i = 1
tid = 47607 i = 2
tid = 47607 i = 3
tid = 47607 i = 4
tid = 47607 i = 5
ptr in thread_data2(): 0x7ffb20001570
pthread_getspecific(): Success
ptr from thread_data2() in thread_func(): 0x7ffb20001570
tid = 47607 i = 1 19000001 Bob
tid = 47607 i = 2
tid = 47607 i = 3 19000003 John
tid = 47607 i = 4
tid = 47607 i = 5
isscgy@ubuntu:/mnt/os-2020$

```

因子函数里本地存储空间释放，  
造成数据丢失

因子函数里本地存储空间未释放，  
数据有效



## ■ 调度程序激活

- 多对多和双层线程模型都需要通信来维持分配给应用程序的适当数量的内核线程，以获得更好的性能。
- 通常在用户线程和内核线程之间使用中间数据结构——轻量级进程（LWP）：
  - 似乎是一个虚拟处理器，进程可以在其上调度用户线程运行。
  - 每个LWP都连接到内核线程。
  - 要创建多少LWP？
- 调度程序激活提供了 *upcalls*——一种从内核到线程库中的 *upcall* 处理程序的通信机制。
  - 通过 *upcall*，内核通知一个应用程序相关的内核事件
  - 这种通信允许应用程序维持正确的内核线程数
  - 注意：系统调用即 *downcall*，是用户程序调用内核的服务，*upcall* 则相反



## Linux clone()

- Linux为fork()和vfork()系统调用提供了复制进程的传统功能。Linux还提供了使用clone()系统调用创建线程的能力。
  - 事实上，Linux使用术语“任务(task)”来表达程序中的控制流，而不是“进程”或“线程”。它不区分进程和线程。
  - clone()带有一组标志，允许子任务共享父任务的某些资源。这些标志确定父任务和子任务之间的共享量。
  - 如果在调用clone()时未设置这些标志，则不会发生共享，这与fork()系统调用提供的标志类似。

标志	意义
CLONE_FS	共享文件系统信息
CLONE_VM	共享相同的内存空间
CLONE_SIGHAND	信号处理程序是共享的
CLONE_FILES	打开的文件集是共享的



## Linux clone()

- 数据结构task\_struct
  - 系统中每个任务都有一个Linux内核数据结构struct task\_struct。此数据结构不存储任务的数据，而是包含指向存储这些数据的其他数据结构的指针。
    - 例如，表示打开文件列表、信号处理信息和虚拟内存的数据结构。
    - 检查你的系统里sched.h头文件中task\_struct。例如  
/usr/src/linux-headers-5.3.0-53/include/linux# vim sched.h
  - 调用fork()时，将创建一个新任务以及父进程的所有关联数据结构的副本
  - 进行clone()系统调用时，也将创建一个新任务。但是，与复制所有数据结构不同，新任务的task\_struct指针指向父级的真实或重复的数据结构，具体取决于传递给clone()的一组标志。





## Linux clone()

### clone()的原型

```
#define _GNU_SOURCE
#include <sched.h>
```

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

- 它实际上是一个位于底层clone()系统调用（以下称为sys\_clone系统调用）之上的库函数。
- clone()的主要用途是实现线程：在共享内存空间中并发运行的程序的多线程的控制。
- 使用clone()创建子进程时，它执行函数fn(arg)。
- 当fn(arg)函数应用程序返回时，子进程/任务终止。fn返回的整数是子进程的退出代码。子进程也可以通过调用exit或在收到致命信号后显式终止。



## Linux clone()

### clone()的原型

```
#define _GNU_SOURCE
#include <sched.h>
```

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

- child\_stack参数指定子进程使用的堆栈的位置
  - 由于子进程和调用进程可能共享内存，因此子进程不可能与调用进程在同一堆栈中执行。
  - 因此，调用进程必须为子堆栈设置内存空间，并将指向该空间的指针传递给clone()。
  - 所有运行Linux的处理器（HP PA处理器除外）上的堆栈都会向下增长，因此child\_stack通常指向为子进程堆栈设置的内存空间的最高地址。



## Linux clone()

### clone()的原型

```
#define _GNU_SOURCE
#include <sched.h>
```

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

- **flags** 的低位字节包含子进程终止时发送给父进程的终止信号的编号
  - 如果此信号被指定为SIGCHLD以外的任何值，则在使用wait()等待子进程时，父进程必须指定\_\_WALL或\_\_WCLONE选项。
  - 如果未指定信号，则当子进程终止时，不会向父进程发送信号
- **flags** 也可以是后面幻灯片中零个或多个常量的按位或，以指定调用任务和子任务之间共享的内容。



## Linux clone()

### 查看Linux手册：

```
#man 2 clone
https://linux.die.net/man/2/clone
http://man7.org/linux/man-pages/man2/clone.2.html
https://www.kernel.org/doc/man-pages/
```

### Linux手册页 (man-pages) 项目

1. **User commands**; 手册页包括极少数Section 1的页面，收录了GNU C库提供的程序。
2. **System calls** 收录了Linux内核提供的系统调用。
3. **Library functions** 收录了标准C库提供的函数。
4. **Devices** 收录各种设备的详细信息，其中大多数位于/dev中。
5. **Files** 描述了各种文件格式，包括记录/proc文件系统的proc(5)。
6. **Overviews**, 惯例和杂项。
7. **Superuser and system administration commands**; 手册页包括少数Section 8的页面，收录了GNU C库提供的程序。



## ■ Linux clone()

### ■ clone()中标志的常量

#### ■ CLONE\_PARENT

- 如果设置了CLONE\_PARENT，则新进程的父进程（由getppid(2)返回）将与调用进程的父进程相同（即新进程是调用进程的**新生兄弟**）
- 如果未设置CLONE\_PARENT，则（与fork(2)一样）子进程的父进程是调用进程
- 注意，当子进程终止时，会发出信号（SIGCHLD）给它的父进程（由getppid(2)返回）。设置CLONE\_PARENT将使调用进程的父进程（而不是调用进程本身）收到信号。



## ■ Linux clone()

### ■ clone()中标志的常量

#### ■ CLONE\_NEWPID

- 如果设置了CLONE\_NEWPID，则在**新的PID命名空间**中创建进程。如果未设置此标志，则将在与调用进程相同的PID命名空间中创建进程。此标志用于**实现容器**。
  - PID命名空间为PID提供了一个独立的环境：PID从1开始，有点像独立系统，对fork(2)、vfork(2)或clone()的调用生成的进程具有命名空间中唯一的PID。
- 在新命名空间中创建的第一个进程（即使用CLONE\_NEWPID标志创建的进程）具有PID 1，且是命名空间的“init”进程。命名空间中孤立的子进程将由此进程收养，而不是init(8)。与传统init进程不同，PID命名空间的“init”进程可以终止，如果终止，命名空间中的所有进程都将终止。



## ■ Linux clone()

### ■ clone()中标志的常量

#### ■ CLONE\_NEWPID (2)

- PID命名空间形成一个**层次结构**。
- 进程**仅对上层可见**
  - 子PID命名空间的进程在父PID命名空间中可见；类似地，子PID命名空间和父PID命名空间中的进程都将在祖父PID命名空间中可见
  - “子”PID命名空间中的进程看不到父命名空间中的进程
- 每个进程可能有**多个PID**：在它可见的每个命名空间中都有一个PID；每个PID在相应的命名空间中都是唯一的。对 `getpid(2)` 的调用始终返回与进程所在命名空间关联的PID。



## ■ Linux clone()

### ■ clone()中标志的常量

#### ■ CLONE\_NEWPID (3)

- 创建新命名空间后，子进程可以**更改其根目录**，并在 `/proc` 处装载新的 `procfs`（进程文件系统，伪文件系统）实例，以便 `ps(1)` 等工具正常工作。（如果 `CLONE_NEWNS` 也包含在标志中，则无需更改根目录：可以直接在 `/proc` 上装载新的 `procfs` 实例。）
- 使用此标志需要：一个配置了 `CONFIG_PID_NS` 选项的内核，并且该进程具有特权（`CAP_SYS_ADMIN`）。
- 此标志不能与 `CLONE_THREAD` 一起指定。



## ■ Linux clone()

### ■ clone()中标志的常量

#### ■ CLONE\_FS

- 如果设置了CLONE\_FS，则调用者和子进程**共享相同的文件系统信息**。
  - 这包括文件系统的根目录、当前工作目录和umask。
  - 调用进程或子进程对chroot(2)、chdir(2)或unmask(2)的调用会相互影响。
- 如果未设置CLONE\_FS，则在调用CLONE()时，子进程将在调用进程的文件系统信息的副本上执行。之后由其中一个进程执行的chroot(2)、chdir(2)、unmask(2)的调用不会影响另一个进程。



## ■ Linux clone()

### ■ clone()中标志的常量

#### ■ CLONE\_FILES

- 如果设置了CLONE\_FILES，则调用进程和子进程**共享同一个文件描述符表**。
  - 由调用进程或子进程创建的任何文件描述符在另一个进程中也有效。
  - 类似地，如果其中一个进程关闭文件描述符或更改其相关标志（使用fcntl(2) F\_SETFD操作），另一个进程也会受到影响。
- 如果未设置CLONE\_FILES，则在clone()时，子进程将继承调用进程中打开的所有文件描述符的副本。（子进程中的重复文件描述符与调用进程中的相应文件描述符指的是相同的打开文件描述。）
  - 之后由调用进程或子进程执行的打开或关闭文件描述符或更改文件描述符标志的操作不会影响另一进程。



## Linux clone()

43 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_NEWNS

- 每个进程都存在一个挂载命名空间，该命名空间是描述该进程所看到的文件层次结构的挂载集。
- 在未设置CLONE\_NEWNS标志的fork(2)或clone()之后，子进程与父进程位于同一挂载命名空间中。
  - 系统调用mount(2)和umount(2)会更改调用进程的挂载命名空间，因此会影响位于同一命名空间中的所有进程，但不会影响位于不同挂载命名空间中的进程。
- 在设置了 CLONE\_NEWNS 标志的clone()之后，将在**新的挂载命名空间**中启动克隆的子进程，并使用父进程挂载命名空间的副本进行初始化。
- 只有特权进程（具有CAP\_SYS\_ADMIN功能的进程）可以指定CLONE\_NEWNS标志。不允许在同一个clone()调用中同时指定CLONE\_NEWNS和CLONE\_FS。



## Linux clone()

44 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_SIGHAND

- 如果设置了CLONE\_SIGHAND，调用进程和子进程**共享同一个信号处理程序表**。
  - 如果调用进程或子进程调用sigaction(2)来改变与信号相关的行为，那么该行为也会在其他进程中发生改变。
  - 但是，调用进程和子进程仍然有不同的信号掩码和挂起的信号集。其中一个进程可以使用sigpromask(2)屏蔽或解除屏蔽一些信号，而不影响另一进程。
- 如果未设置CLONE和，则子进程将在调用clone()时继承调用进程的信号处理程序的副本。其中一个进程稍后执行的对sigaction(2)的调用对另一个进程没有影响。
- 自Linux 2.6.0-test6版起，如果指定了CLONE\_SIGHAND，则还必须同时指定CLONE\_VM



## Linux clone()

45 / 74

### ■ Linux clone()

#### ■ clone()中标志的常量

##### ■ CLONE\_PTRACE

- 如果指定了CLONE\_PTRACE，并且正在跟踪调用进程，则也要跟踪子进程。

##### ■ CLONE\_UNPTRACE

- 如果指定了CLONE\_UNTRACED，则跟踪进程无法强制跟踪子进程。



## Linux clone()

46 / 74

### ■ Linux clone()

#### ■ clone()中标志的常量

##### ■ CLONE\_VFORK

- 如果设置了CLONE\_VFORK，则调用进程的执行将挂起，直到子进程释放其虚拟内存资源，通过调用execve(2)或\_exit(2)（与vfork(2)类似）
- 如果未设置CLONE\_VFORK，则调用进程和子进程都可以在调用后进行调度，并且应用程序不应依赖于以任何特定顺序执行。



## Linux clone()

47 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_VM

- 如果设置了CLONE\_VM，则调用进程和子进程在**同一内存空间**中运行。特别是，由调用进程或子进程执行的内存写入也可以在另一个进程中看到。此外，子进程或调用进程使用mmap(2)或munmap(2)执行的任何内存映射或取消映射也会影响另一进程。
- 如果未设置CLONE\_VM，则子进程在调用进程的内存空间的**分离副本**中运行。与fork(2)一样，其中一个进程执行的内存写入或文件映射/取消映射不会影响另一个进程。



## Linux clone()

48 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_IO

- 如果设置了CLONE\_IO，则新进程将与调用进程**共享I/O上下文**。
  - I/O上下文是磁盘调度器的I/O作用域（即，I/O调度器用于为进程的I/O调度建模的范围）。如果进程共享相同的I/O上下文，I/O调度程序将它们视为一个，它们可以共享磁盘时间。对于某些I/O调度器，如果两个进程共享一个I/O上下文，则允许它们交错访问磁盘。如果多个线程代表同一进程执行I/O（如aio\_read(3)），应使用CLONE\_IO来获得更好的I/O性能
  - 如内核未配置CONFIG\_BLOCK选项，则此标志为no-op(没有操作，无效)
- 如果没有设置CLONE\_IO，那么（与fork(2)一样），新进程有自己的I/O上下文。





## Linux clone()

49 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_NEWUTS

- 如果设置了CLONE\_NEWUTS，则在新的UTS命名空间中创建进程，通过复制调用进程的UTS命名空间中的标识符来初始化其标识符。
- 如果未设置此标志，则（与fork(2)一样）在与调用进程相同的UTS命名空间中创建进程。此标志用于实现容器。
- UTS命名空间是uname(2)返回的标识符集；其中，域名和主机名可以分别通过setdomainname(2)和sethostname(2)修改。对UTS命名空间中的标识符所做的更改对同一命名空间中的所有其他进程可见，但对其他UTS命名空间中的进程不可见。（参见uts\_namespaces(7)）
- 使用此标志需要：一个配置了CONFIG\_UTS\_NS选项的内核，并且该进程具有特权（CAP\_SYS\_ADMIN）。



## Linux clone()

50 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_NEWUSER

- 如果设置了CLONE\_NEWUSER，则在新的用户命名空间中创建进程。如果未设置此标志，则（与fork(2)一样）在与调用进程相同的用户命名空间中创建进程。
- 有关用户命名空间的更多信息，请参阅namespaces(7)和user\_namespaces(7)。
- 在Linux 3.8之前，使用CLONE\_NEWUSER需要调用方具有三个功能：CAP\_SYS\_ADMIN、CAP\_SETUID和CAP\_SETGID。从Linux 3.8开始，创建用户命名空间不需要特权。
- 此标志不能与CLONE\_THREAD或CLONE\_PARENT一起指定。出于安全原因，不能与CLONE\_FS一起指定。



## Linux clone()

51 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_NEWIPC

- 如果设置了CLONE\_NEWIPC，则在**新的IPC命名空间**中创建进程。
- 如果未设置CLONE\_NEWIPC，则（与fork(2)一样），该进程将在与调用进程相同的IPC命名空间中创建。
- 此标志**用于实现容器**。IPC命名空间提供System V IPC对象和（自Linux 2.6.30）POSIX消息队列的隔离视图。这些IPC机制的共同特点是IPC对象由**非文件系统路径名**的机制标识。
- 在IPC命名空间中创建的对象对作为该命名空间成员的所有其他进程可见，但对其他IPC命名空间中的进程不可见。



## Linux clone()

52 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_NEWIPC(2)

- 当IPC命名空间被销毁（作为命名空间成员的最后一个进程终止）时，命名空间中的所有IPC对象将自动销毁。
- 使用此标志需要：一个配置了CONFIG\_SYSVIPC和CONFIG\_IPC\_NS选项的内核，并且该进程具有特权（CAP\_SYS\_ADMIN）。此标志不能与CLONE\_SYSVSEM一起指定。



## Linux clone()

53 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_NEWNET

- 该标志的实现仅在内核版本2.6.29左右时完成。
- 如果设置了CLONE\_NEWNET，则在新的网络命名空间中创建进程。
- 如果未设置CLONE\_NEWNET，则（与fork(2)一样），该进程将在与调用进程相同的网络命名空间中创建。
- 此标志用于实现容器。网络命名空间提供网络堆栈的隔离视图（网络设备接口、IPv4和IPv6协议堆栈、IP路由表、防火墙规则、/proc/net和/sys/class/net目录树、套接字等）。



## Linux clone()

54 / 74

## Linux clone()

### clone()中标志的常量

#### CLONE\_NEWNET(2)

- 物理网络设备只能存在于一个网络命名空间中。虚拟网络设备（“veth”）对提供了类似管道的抽象，可用于在网络命名空间之间创建隧道，并可用于在另一命名空间中创建到物理网络设备的网桥。
- 释放网络命名空间（命名空间中的最后一个进程终止）时，其物理网络设备将移回初始网络命名空间（而不是进程的父进程）。
- 使用此标志需要：配置了CONFIG\_NET\_NS选项的内核，并且该进程具有特权（CAP\_SYS\_ADMIN）



## ■ Linux clone()

### ■ clone()中标志的常量

#### ■ CLONE\_THREAD

- 如果设置了 CLONE\_THREAD，则子线程将与调用进程放在**同一个线程组中**。
  - 这里，术语“线程”用于指线程组中的进程。
- 线程组是Linux 2.4中添加的一项功能，它支持POSIX线程概念，即共享一个PID的一组线程。在内部，此共享PID是线程组的所谓线程组标识符（TGID）。从Linux2.4开始，对 `getpid(2)` 的调用将返回调用者的TGID。
- 组中的线程可以通过其（系统范围）唯一的线程ID（TID）来区分。新线程的TID可用作返回给 `clone()` 调用方的函数结果，线程可以使用 `gettid(2)` 获得自己的TID。



## ■ Linux clone()

### ■ clone()中标志的常量

#### ■ CLONE\_THREAD(2)

- 使用CLONE\_THREAD创建的新线程与 `clone()` 的调用方具有相同的父进程（即，与CLONE\_PARENT类似），因此对 `getppid(2)` 的调用为线程组中的所有线程返回相同的值。
- 当CLONE\_THREAD线程终止时，使用 `clone()` 创建它的线程不会发送SIGCHLD或其他终止信号；也不能使用 `wait(2)` 获得这样一个线程的状态。可以说这个线程已经分离(detached)了
- 线程组中的所有线程终止后，线程组的父进程将收到SIGCHLD或其他终止信号
- 在不指定CLONE\_THREAD的情况下调用 `clone()` 时，生成的线程将被放入一个新的线程组中，该线程组的TGID与线程的TID相同。此线程是新线程组的**leader**。



## Linux clone()

### clone()中标志的常量

#### CLONE\_THREAD(3)

- 如果线程组中的任何线程执行`execve(2)`，则终止除线程组leader以外的所有线程，并在线程组leader中执行新程序
- 如果线程组中的一个线程使用`fork(2)`创建一个子线程，那么组中的任何线程都可以`wait(2)`该子线程。
- 自Linux 2.5.35以来，如果指定了`CLONE_THREAD`，则标志还必须包括`CLONE_SIGHAND`
- 可以使用`kill(2)`将信号发送到整个线程组（即TGID），或者使用`tgkill(2)`将信号发送到特定线程（即TID）。
- 信号处理和操作是进程范围的：如果未经处理的信号被传递到线程，那么它将影响（终止、停止、继续、忽略）线程组的所有成员。



## Linux clone()

### clone()中标志的常量

#### CLONE\_THREAD (4)

- 每个线程都有自己的信号掩码，由`sigprocmask(2)`设置，但信号可以挂起：当与`kill(2)`一起发送时，对于整个进程（即，可传递给线程组的任何成员）；或者当与`tgkill(2)`一起发送时，对于单个线程。
- 对`sigpending(2)`的调用将返回一个信号集，它是整个进程的挂起信号和调用线程的挂起信号的联合。
- 如果使用`kill(2)`向线程组发送信号，并且线程组已为该信号设置了处理程序，则该处理程序将在未阻止该信号的线程组中的一个任意选择的成员中调用。如果一个组中的多个线程正在等待使用`sigwaitinfo(2)`接受相同的信号，内核将任意选择其中一个线程来接收使用`kill(2)`发送的信号。



## Linux clone()

### clone()中标志的常量

#### \*CLONE\_STOPPED

- 如果设置了CLONE\_STOPPED，则子进程最初会**被停止**（就像它被发送了SIGSTOP信号一样），并且必须通过发送SIGCONT信号来恢复。
- 此标志从Linux 2.6.25开始就被**弃用**，并在Linux 2.6.38中被完全删除。

#### \*CLONE\_PID

- 如果设置了CLONE\_PID，则子进程将使用与调用进程**相同的进程ID**创建。这有助于入侵系统，但在其他方面用处不大。自Linux 2.3.21，该标志只能由系统引导进程(PID 0)指定。它自Linux 2.5.16中**消失了**。



## Examples

### 算法.14-6-clone-demo.c:使用VM和VFORK标志clone() (1)

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sched.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/syscall.h>
#include <unistd.h>
#define gettid() syscall(__NR_gettid)
/* wrap the system call syscall(__NR_gettid), __NR_gettid = 224 */
#define gettidv2() syscall(SYS_gettid) /* a traditional wrapper */
#define STACK_SIZE 1024*1024 /* 1Mib.问题:什么是STACK_SIZE的上界? */

static int child_func1(void *arg)
{
    char *chdbuf = (char*)arg; /* type casting */
    printf("child_func1 read buf: %s\n", chdbuf);
    sprintf(chdbuf, "I am child_func1, my tid = %ld, pid = %d", gettid(), getpid());
    printf("child_func1 set buf: %s\n", chdbuf);
    printf("child_func1 sleeping and then exits ...\n");
    sleep(1);

    return 0;
}
```

子进程1读写参数；  
休眠并退出；



## Linux clone()

61 / 74

## Examples

### ■ 算法.14-6-clone-demo.c:使用VM和VFORK标志clone() (2)

```
static int child_func2(void *arg)
{
    char *chdbuf = (char*)arg; /* type casting */
    printf("child_func2 read buf: %s\n", chdbuf);
    sprintf(chdbuf, "I am child_func2, my tid = %ld, pid = %d", gettid(), getpid());
    printf("child_func2 set buf: %s\n", chdbuf);
    printf("child_func2 sleeping and then exists ...\n");
    sleep(1);

    return 0;
}

int main(int argc, char **argv)
{
    char *stack1 = malloc(STACK_SIZE*sizeof(char));
    /* allocating from heap, safer than stack1[STACK_SIZE] */
    char *stack2 = malloc(STACK_SIZE*sizeof(char));
    pid_t chdtid1, chdtid2;
    unsigned long flags = 0;
    char buf[100]; /* a global variable has the same behavior */

    if(!stack1 || !stack2) {
        perror("malloc()");
        exit(1);
    }
}
```

子进程2读写参数；  
休眠并退出；

父进程分配栈内存；



## Linux clone()

62 / 74

## Examples

### ■ 算法.14-6-clone-demo.c:使用VM和VFORK标志clone() (3)

```
/* set CLONE flags */
if((argc > 1) && (!strcmp(argv[1], "vm"))) {
    flags |= CLONE_VM;
}
if((argc > 2) && (!strcmp(argv[2], "vfork"))) {
    flags |= CLONE_VFORK;
}

sprintf(buf, "I am parent, my pid = %d", getpid());
printf("parent set buf: %s\n", buf);
printf("parent clone ...\n");

/* 创建子线程，子线程的顶部是stack+STACK_SIZE */
chdtid1 = clone(child_func1, stack1 + STACK_SIZE, flags | SIGCHLD, buf);
/* 没有SIGCHLD会发生什么 */
if(chdtid1 == -1) {
    perror("clone1()");
    exit(1);
}

chdtid2 = clone(child_func2, stack2 + STACK_SIZE, flags | SIGCHLD, buf);
if(chdtid2 == -1) {
    perror("clone2()");
    exit(1);
}
}
```

如果clone()有  
CLONE\_VM但没有  
SIGCHLD会发生什么？

根据命令行参数，  
父进程克隆子线程1、2；



## Linux clone()

63 / 74

### Examples

#### ■ 算法.14-6-clone-demo.c:使用VM和VFORK标志clone() (4)

```
printf("parent waiting ... \n");

int status = 0;
if(waitpid(-1, &status, 0) == -1) { /* 等待任何一个子线程退出, 可能会让某个子线程失效 */
    perror("wait()");
}

//waitpid(chdtid1, &status, 0);
//waitpid(chdtid2, &status, 0);

sleep(2);

printf("parent read buf: %s\n", buf);

system("ps");

free(stack1);
free(stack2);
stack1 = NULL;
stack2 = NULL;

return 0;
}
```

父进程等待一个子线程结束



## Linux clone()

64 / 74

### Examples

#### ■ 算法.14-6-clone-demo.c:使用VM和VFORK标志clone() (4)

```
lab@ByteDanceServer:~/os/lec14$ gcc alg.14-6-clone-demo.c
lab@ByteDanceServer:~/os/lec14$ ./a.out
parent set buf: I am parent, my pid = 153
parent clone ...
parent waiting ...
child_func1 read buf: I am parent, my pid = 153
child_func2 read buf: I am parent, my pid = 153
child_func1 set buf: I am child_func1, my tid = 154, pid = 154
child_func2 set buf: I am child_func2, my tid = 155, pid = 155
child_func1 sleeping and then exits ...
child_func2 sleeping and then exits ...
parent read buf: I am parent, my pid = 153

PID TTY          TIME CMD
 18 pts/0        00:00:00 bash
 153 pts/0        00:00:00 a.out
 155 pts/0        00:00:00 a.out <defunct>
 156 pts/0        00:00:00 sh
 157 pts/0        00:00:00 ps

lab@ByteDanceServer:~/os/lec14$ |
```

每个任务都有自己  
的内存空间。  
互不影响。





## Linux clone()

65 / 74

## ■ Examples

- 算法.14-6-clone-demo.c:使用VM和VFORK标志clone() (4)

```
lab@ByteDanceServer:~/os/lec14$ gcc alg.14-6-clone-demo.c
lab@ByteDanceServer:~/os/lec14$ ./a.out
parent set buf: I am parent, my pid = 153
parent clone ...
parent waiting ...
child_func1 read buf: I am parent, my pid = 153
child_func2 read buf: I am parent, my pid = 153
child_func1 set buf: I am child_func1, my tid = 154, pid = 154
child_func2 set buf: I am child_func2, my tid = 155, pid = 155
child_func1 sleeping and then exits ...
child_func2 sleeping and then exits ...
parent read buf: I am parent, my pid = 153
```

PID	TTY	TIME	CMD
18	pts/0	00:00:00	bash
153	pts/0	00:00:00	a.out
155	pts/0	00:00:00	a.out <defunct>
156	pts/0	00:00:00	sh
157	pts/0	00:00:00	ps

```
lab@ByteDanceServer:~/os/lec14$ |
```

父子任务异步执行



## Linux clone()

66 / 74

## ■ Examples

- 算法.14-6-clone-demo.c:使用VM和VFORK标志clone() (4)

```
lab@ByteDanceServer:~/os/lec14$ ./a.out (vm)
parent set buf: I am parent, my pid = 187
parent clone ...
parent waiting ...
child_func1 read buf: I am parent, my pid = 187
child_func2 read buf: I am parent, my pid = 187
child_func1 set buf: I am child_func1, my tid = 188, pid = 188
child_func1 set buf: I am child_func1, my tid = 188, pid = 188
child_func2 set buf: I am child_func2, my tid = 189, pid = 189
child_func1 sleeping and then exits ...
child_func1 sleeping and then exits ...
child_func2 sleeping and then exits ...
parent read buf: I am child_func2, my tid = 189, pid = 189
```

PID	TTY	TIME	CMD
18	pts/0	00:00:00	bash
187	pts/0	00:00:00	a.out
189	pts/0	00:00:00	a.out <defunct>
190	pts/0	00:00:00	sh
191	pts/0	00:00:00	ps

任务共享内存空间。  
父进程的内存被子线程改写。



## Linux clone()

67 / 74

## ■ Examples

- 算法.14-6-clone-demo.c:使用VM和VFORK标志clone() (4)

```
lab@ByteDanceServer:~/os/lec14$ ./a.out vm
parent set buf: I am parent, my pid = 187
parent clone ...
parent waiting ...
child_func1 read buf: I am parent, my pid = 187
child_func2 read buf: I am parent, my pid = 187
child_func1 set buf: I am child_func1, my tid = 188, pid = 188
child_func1 set buf: I am child_func1, my tid = 188, pid = 188
child_func2 set buf: I am child_func2, my tid = 189, pid = 189
child_func1 sleeping and then exits ...
child_func1 sleeping and then exits ...
child_func2 sleeping and then exits ...
parent read buf: I am child_func2, my tid = 189, pid = 189
```

PID	TTY	TIME	CMD
18	pts/0	00:00:00	bash
187	pts/0	00:00:00	a.out
189	pts/0	00:00:00	a.out <defunct>
190	pts/0	00:00:00	sh
191	pts/0	00:00:00	ps

父子任务异步执行



## Linux clone()

68 / 74

## ■ Examples

- 算法.14-6-clone-demo.c:使用VM和VFORK标志clone() (4)

```
lab@ByteDanceServer:~/os/lec14$ ./a.out vm vfork
parent set buf: I am parent, my pid = 192
parent clone ...
child_func1 read buf: I am parent, my pid = 192
child_func1 set buf: I am child_func1, my tid = 193, pid = 193
child_func1 sleeping and then exits ...
child_func2 read buf: I am child_func1, my tid = 193, pid = 193
child_func2 set buf: I am child_func2, my tid = 194, pid = 194
child_func2 sleeping and then exits ...
parent waiting ...
parent read buf: I am child_func2, my tid = 194, pid = 194
```

PID	TTY	TIME	CMD
18	pts/0	00:00:00	bash
192	pts/0	00:00:00	a.out
194	pts/0	00:00:00	a.out <defunct>
195	pts/0	00:00:00	sh
196	pts/0	00:00:00	ps

```
lab@ByteDanceServer:~/os/lec14$
```

父进程挂起了，  
因为使用了VFORK标志

■ 算法.14-7-clone-stack.c: 测试克隆线程堆栈的上界 (1)

■ 算法.14-7-clone-stack.c: 测试克隆线程堆栈的上界 (2)

35



## Linux clone()

71 / 74

## Examples

### ■ 算法.14-7-clone-stack.c: 测试克隆线程堆栈的上界 (2)

```
int main(int argc, char **argv)
{
    char *stack = malloc(STACK_SIZE); /* 从进程的堆中分配 */
    pid_t chdtid;
    char buf[40];
    if(!stack) {
```

```
isscg@ubuntu:/mnt/os-2020$ gcc alg.14-7-clone-stack.c
isscg@ubuntu:/mnt/os-2020$ ./a.out

main: my pid = 48422, I'm waiting for cloned child, his tid = 48423
test: my ptd = 48423, tid = 48423, ppid = 48422

iteration = 1936125
main: my pid = 48422, waitpid returns = 48423
isscg@ubuntu:/mnt/os-2020$
```

```
ret = waitpid(-1, &status, 0); /* wait for any child existing */
```

STACK\_SIZE是 $(524288-10000)*4096 = 2,106,523,648$

迭代了1936125次, 每次使用1024,  $1936125*1024 = 1,982,592,000$

相差123,931,648, 每次迭代的overhead = 64字节

```
stack = NULL;
return 0;
}
```



## Examples

72 / 74

## Windows线程

- Windows实现Windows API—Win 98、Win NT、Win 2000、Win XP和Win 7的主要API。
- 实现内核级的一对一映射。
- 每个线程包含：
  - 线程ID。
  - 表示处理器状态的寄存器集。
  - 当线程在用户模式或内核模式下运行时, 分开用户和内核堆栈
  - 运行时库和动态链接库 (DLL) 使用的专用数据存储区域。
- 寄存器集、堆栈和专用存储区域称为线程的上下文。



## ■ Windows线程

- 线程的主要数据结构包括：
  - ETHREAD（执行线程块）
    - 在内核空间中，包含指向线程所属进程和KTHREAD的指针
  - KTHREAD（内核线程块）
    - 在内核空间中，调度和同步信息，内核模式堆栈，指向TEB的指针。
  - TEB（线程环境块）
    - 在用户空间中，线程ID，用户模式堆栈，线程本地存储。



## ■ Windows线程

- 线程的主要数据结构

