Linux多线程概述

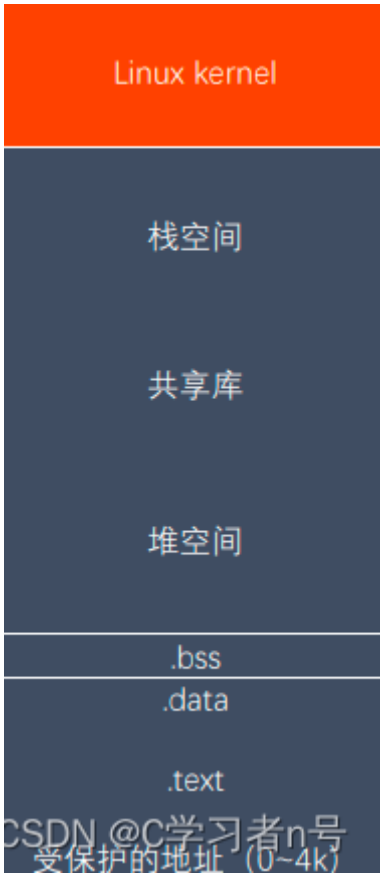
概述

■ 与进程（process）类似，线程（thread）是允许应用程序并发执行多个任务的一种机制。一个进程可以包含多个线程。同一个程序中的所有线程均会独立执行相同程序，且共享同一份全局内存区域，其中包括初始化数据段、未初始化数据段，以及堆内存段。（传统意义上的 UNIX 进程只是多线程程序的一个特例，该进程只包含一个线程）
■ 进程是 CPU 分配资源的最小单位，线程是操作系统调度执行的最小单位。
■ 线程是轻量级的进程（LWP：Light Weight Process），在 Linux 环境下线程的本质仍是进程。
■ 查看指定进程的 LWP 号：ps –Lf pid

进程和线程的区别

■ 进程间的信息难以共享。由于除去只读代码段外，父子进程并未共享内存，因此必须采用一些进程间通信方式，在进程间进行信息交换。
■ 调用 fork() 来创建进程的代价相对较高，即便利用写时复制技术，仍然需要复制诸如内存页表和文件描述符表之类的多种进程属性，这意味着 fork() 调用在时间上的开销依然不菲。
■ 线程之间能够方便、快速地共享信息。只需将数据复制到共享（全局或堆）变量中即可。
■ 创建线程比创建进程通常要快 10 倍甚至更多。线程间是共享虚拟地址空间的，无需采用写时复制来复制内存，也无需复制页表。

线程间的共享和非共享资源



- 共享资源
  - 进程 ID 和父进程 ID
  - 进程组 ID 和会话 ID
  - 用户 ID 和 用户组 ID
  - 文件描述符表
  - 信号处置
  - 文件系统的相关信息：文件权限掩码（umask）、当前工作目录
  - 虚拟地址空间（除栈、.text）
- 非共享资源
  - 线程 ID
  - 信号掩码
  - 线程特有数据
  - error 变量
  - 实时调度策略和优先级
  - 栈，本地变量和函数的调用链接信息

NPTL

■ 当 Linux 最初开发时，在内核中并不能真正支持线程。但是它的确可以通过 clone() 系统调用将进程作为可调度的实体。这个调用创建了调用进程（calling process）的一个拷贝，这个拷贝与调用进程共享相同的地址空间。LinuxThreads 项目使用这个调用来完成在用户空间模拟对线程的支持。不幸的是，这种方法有一些缺点，尤其是在信号处理、调度和进程间同步等方面都存在问题。另外，这个线程模型也不符合 POSIX 的要求。
■ 要改进 LinuxThreads，需要内核的支持，并且重写线程库。有两个相互竞争的项目开始来满足这些要求。一个包括 IBM 的开发人员的团队开展了 NGPT（Next-Generation POSIX Threads）项目。同时，Red Hat 的一些开发人员开展了 NPTL 项目。NGPT 在 2003 年中期被放弃了，把这个领域完全留给了 NPTL。
■ NPTL，或称为 Native POSIX Thread Library，是 Linux 线程的一个新实现，它克服了 LinuxThreads 的缺点，同时也符合 POSIX 的需求。与 LinuxThreads 相比，它在性能和稳定性方面都提供了重大的改进。
■ 查看当前 pthread 库版本：`getconf NU_LIBPTHREAD_VERSION`

线程操作函数

```
■ int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine) (void *), void *arg);
■ pthread_t pthread_self(void);
■ int pthread_equal(pthread_t t1, pthread_t t2);
■ void pthread_exit(void *retval);
■ int pthread_join(pthread_t thread, void **retval);
■ int pthread_detach(pthread_t thread);
■ int pthread_cancel(pthread_t thread);
```

函数详解

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine) (void *), void *arg);
    功能：创建一个线程。
        参数：
            thread：输出参数，输出创建的线程的id
            attr：线程的属性，传入NULL使用系统默认属性，也可以自定义
            start_routine：线程要执行函数。且要求函数的返回值和参数类型都是void *
            arg：线程执行函数的参数
        返回值：（以下函数的返回值类同于此）
            成功 返回 0
```

失败 返回错误号。和errno不一样。获取错误号对应的信息函数：char * strerror(int errnum)

```
pthread_t pthread_self(void);
```

功能：返回执行该函数的线程的id

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

功能：比较两个线程id是否相等。

不同的操作系统pthread_t的类型的实现不一样，有的是无符号的长整型，有的是使用结构体去实现的。

```
void pthread_exit(void *retval);
```

功能：终止一个线程，在哪个线程调用，就表示终止哪个进程。终止线程之后，后面的代码都不会执行。

通过该函数终止的线程不会影响到其他线程的执行。比如在main线程中结束一般使用return，那么它的结束就意味这子线程结束，

如果使用该函数终止main进程，那么子进程还是正常运行到结束的。

参数：

retval：是一个输入参数，作为一个返回值，可以在pthread_join函数中获取到这个参数。

```
int pthread_join(pthread_t thread, void **retval);
```

功能：连接一个已经终止的线程，目的是对它的资源进行回收。类似于进程的wait

该函数是阻塞函数，回收指定id的线程。通常在主线程中使用，回收子线程。

参数：

thread：需要回收的线程的id

**retval：二级指针类型，接收子进程退出时的返回值，也就是pthread_exit函数的参数。

返回值：

与pthread_create函数的返回值一样

```
int pthread_detach(pthread_t thread);
```

功能：分离一个指定的线程。被分离的线程在终止的时候，会自动释放资源返回给系统。

所以，可以通过分离线程的方式实现pthread_join函数的回收资源的功能。

不能多次分离，会产生不可预料的行为。

不能去连接（pthread_join）一个已经分离的线程，会报错。

参数：thread就是要分离的线程的id

返回值：与pthread_create函数的返回值一样

```
int pthread_cancel(pthread_t thread);
```

功能：取消参数指定的线程（让线程终止）

取消某个线程，可以终止某个线程的运行，但是并不是立马终止，而是当子线程执行到一个取消点，线程才会终止。

取消点：系统规定好的一些系统调用，我们可以粗略的理解为从用户区到内核区的切换，这个位置称之为取消点。

所以，每次执行同样的程序，程序结果可能是不一致的。

函数pthread_create

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

void * callback(void * arg) {
    printf("child thread...\n");
    printf("arg value: %d\n", *(int *)arg);
    return NULL;
}
```

```c
int main() {

    pthread_t tid;

    int num = 10;

    // 创建一个子线程
    int ret = pthread_create(&tid, NULL, callback, (void *)&num);

    if(ret != 0) {
        char * errstr = strerror(ret);
        printf("error : %s\n", errstr);
    }

    for(int i = 0; i < 5; i++) {
        printf("%d\n", i);
    }

    sleep(1);

    return 0;   // exit(0);
}
```

函数pthread_exit

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

void * callback(void * arg) {
    printf("child thread id : %ld\n", pthread_self());
    return NULL;    // pthread_exit(NULL);
}

int main() {

    // 创建一个子线程
    pthread_t tid;
    int ret = pthread_create(&tid, NULL, callback, NULL);

    if(ret != 0) {
        char * errstr = strerror(ret);
        printf("error : %s\n", errstr);
    }

    // 主线程
    for(int i = 0; i < 5; i++) {
        printf("%d\n", i);
    }

    printf("tid : %ld, main thread id : %ld\n", tid ,pthread_self());

    // 让主线程退出,当主线程退出时，不会影响其他正常运行的线程。
    pthread_exit(NULL);

    printf("main thread exit\n");
```

```
    return 0;   // exit(0);
}
```

函数pthread_join

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

int value = 10;

void * callback(void * arg) {
    printf("child thread id : %ld\n", pthread_self());
    // sleep(3);
    // return NULL;
    // int value = 10; // 局部变量
    pthread_exit((void *)&value);   // return (void *)&value;
}

int main() {

    // 创建一个子线程
    pthread_t tid;
    int ret = pthread_create(&tid, NULL, callback, NULL);

    if(ret != 0) {
        char * errstr = strerror(ret);
        printf("error : %s\n", errstr);
    }

    // 主线程
    for(int i = 0; i < 5; i++) {
        printf("%d\n", i);
    }

    printf("tid : %ld, main thread id : %ld\n", tid ,pthread_self());

    // 主线程调用pthread_join()回收子线程的资源
    int * thread_retval;
    ret = pthread_join(tid, (void **)&thread_retval);

    if(ret != 0) {
        char * errstr = strerror(ret);
        printf("error : %s\n", errstr);
    }

    printf("exit data : %d\n", *thread_retval);

    printf("回收子线程资源成功! \n");

    // 让主线程退出,当主线程退出时，不会影响其他正常运行的线程。
    pthread_exit(NULL);

    return 0;
}
```

函数pthread_detach

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

void * callback(void * arg) {
    printf("chid thread id : %ld\n", pthread_self());
    return NULL;
}

int main() {

    // 创建一个子线程
    pthread_t tid;

    int ret = pthread_create(&tid, NULL, callback, NULL);
    if(ret != 0) {
        char * errstr = strerror(ret);
        printf("error1 : %s\n", errstr);
    }

    // 输出主线程和子线程的id
    printf("tid : %ld, main thread id : %ld\n", tid, pthread_self());

    // 设置子线程分离,子线程分离后，子线程结束时对应的资源就不需要主线程释放
    ret = pthread_detach(tid);
    if(ret != 0) {
        char * errstr = strerror(ret);
        printf("error2 : %s\n", errstr);
    }

    // 设置分离后，对分离的子线程进行连接 pthread_join()会报错
    // ret = pthread_join(tid, NULL);
    // if(ret != 0) {
    //     char * errstr = strerror(ret);
    //     printf("error3 : %s\n", errstr);
    // }

    pthread_exit(NULL);

    return 0;
}
```

函数pthread_cancel

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

void * callback(void * arg) {
    printf("chid thread id : %ld\n", pthread_self());
    for(int i = 0; i < 5; i++) {
        printf("child : %d\n", i);
```

```
    }
    return NULL;
}

int main() {

    // 创建一个子线程
    pthread_t tid;

    int ret = pthread_create(&tid, NULL, callback, NULL);
    if(ret != 0) {
        char * errstr = strerror(ret);
        printf("error1 : %s\n", errstr);
    }

    // 取消线程
    pthread_cancel(tid);

    for(int i = 0; i < 5; i++) {
        printf("%d\n", i);
    }

    // 输出主线程和子线程的id
    printf("tid : %ld, main thread id : %ld\n", tid, pthread_self());


    pthread_exit(NULL);

    return 0;
}
```

线程属性

概述

```
/*
线程属性的相关函数：
    线程属性：对应了pthread_create函数的第二个参数。可以对线程的一些属性在线程创建之出就进行设置，
                同时也可以获取对应的属性（get和set）。比如线程的栈的大小，线程是否分离（相当于pthread_detach函数）等等

    线程属性的类型：pthread_attr_t

    属性的初始化和销毁：
    int pthread_attr_init(pthread_attr_t *attr);
        - 初始化线程属性变量
    int pthread_attr_destroy(pthread_attr_t *attr);
        - 释放线程属性的资源

    比如线程分离属性：
    int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
        - 获取线程分离的状态属性
    int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
        - 设置线程分离的状态属性
```

```
线程属性的相关函数，输入man pthread_attr_t + tabl键可以查看相关的函数
pthread_attr_destroy            pthread_attr_getstack
pthread_attr_setschedparam
pthread_attr_getaffinity_np     pthread_attr_getstackaddr
pthread_attr_setschedpolicy
pthread_attr_getdetachstate     pthread_attr_getstacksize
pthread_attr_setscope
pthread_attr_getguardsize       pthread_attr_init
pthread_attr_setstack
pthread_attr_getinheritsched    pthread_attr_setaffinity_np
pthread_attr_setstackaddr
pthread_attr_getschedparam    pthread_attr_setdetachstate
pthread_attr_setstacksize
pthread_attr_getschedpolicy     pthread_attr_setguardsize
pthread_attr_getscope           pthread_attr_setinheritsched
*/
```

示例

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

void *callback(void *arg)
{
    printf("子进程\n");
}

int main(int argc, char **argv)
{
    // Initialize
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    //设置线程分离和栈的大小
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_attr_setstacksize(&attr, 1024 * 1024);

    pthread_t thread;
    pthread_create(&thread, &attr, callback, NULL);

    size_t size;
    pthread_attr_getstacksize(&attr, &size);
    printf("size = %ld\n", size);

    //销毁属性对象
    pthread_attr_destroy(&attr);

    pthread_exit(NULL);
}
```