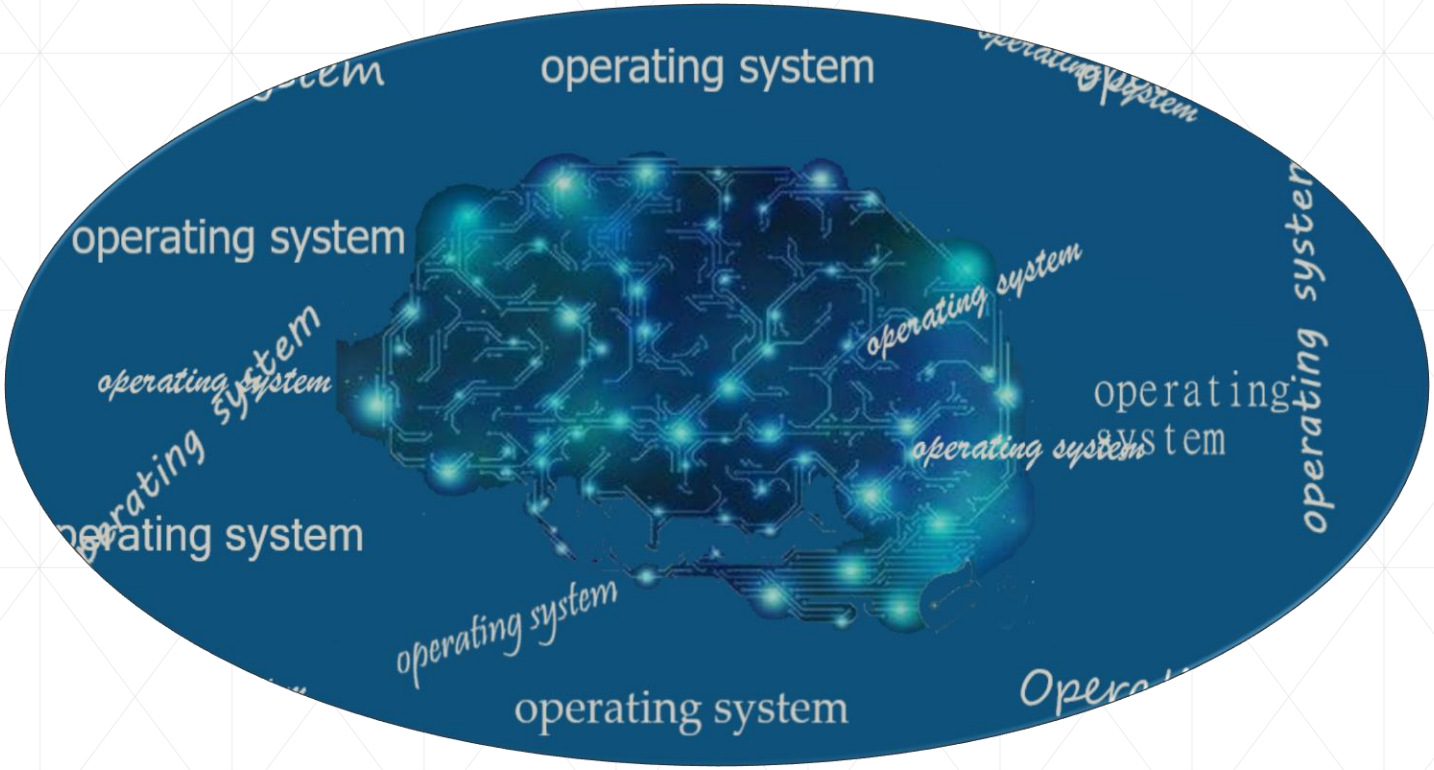


第十二章 线程与线程控制

授课教师

电子邮箱：

主要内容

12.1 线程概述

12.2 线程与进程的比较

12.3 线程的控制

线程的基本概念

1.线程的引入

- 在OS中一直都是以进程作为能拥有资源和独立运行的基本单位的。
- 直到20世纪80年代中期，提出了比进程更小的能独立运行的基本单位——线程（Threads）；试图用它来提高系统内程序并发执行的程度，从而可进一步提高系统的吞吐量。

线程的基本概念

- 在操作系统中引入进程的目的：使多个程序能并发执行，以提高资源利用率和系统吞吐量。
- 线程的目的：
  - 减少程序并发执行时所付出的时空开销
  - 使操作系统具有更好的并发性

线程的基本概念

2. 线程与进程

线程：一个动态对象，它是处理机调度的基本单位，表示进程的一个控制点，执行一系列的指令。

线程被称为轻型进程(Light-Weight Process)；

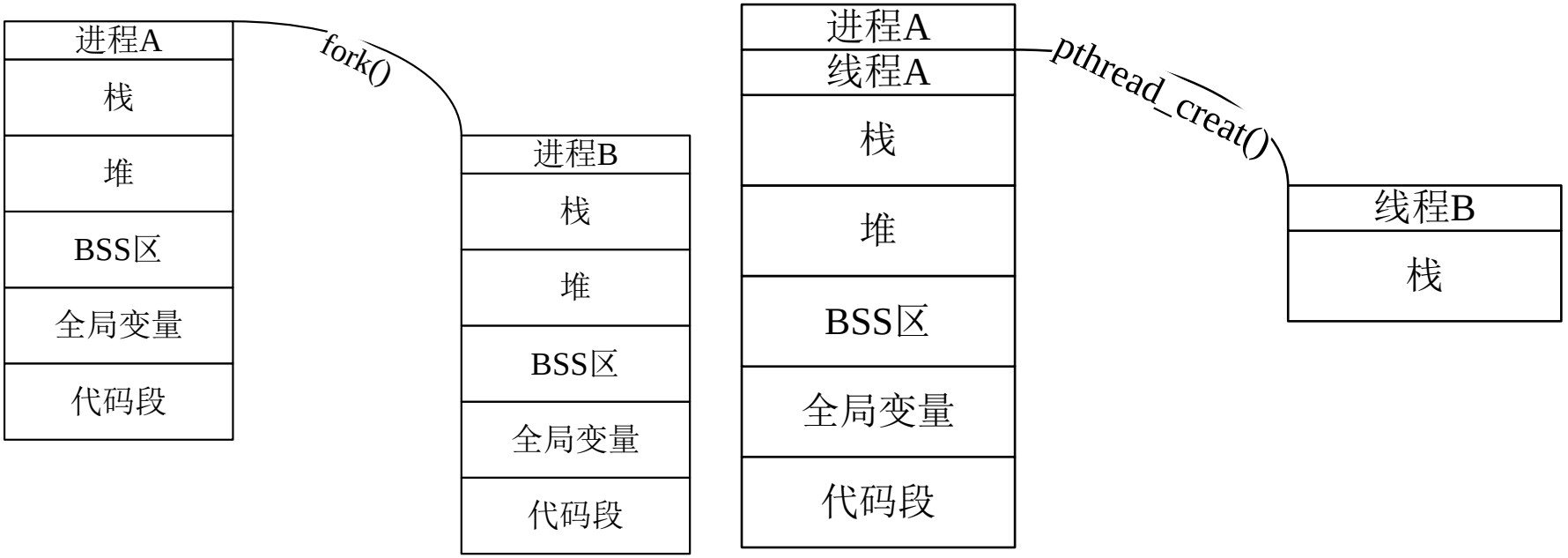
传统进程称为重型进程(Heavy-Weight Process)。

2. 线程与进程

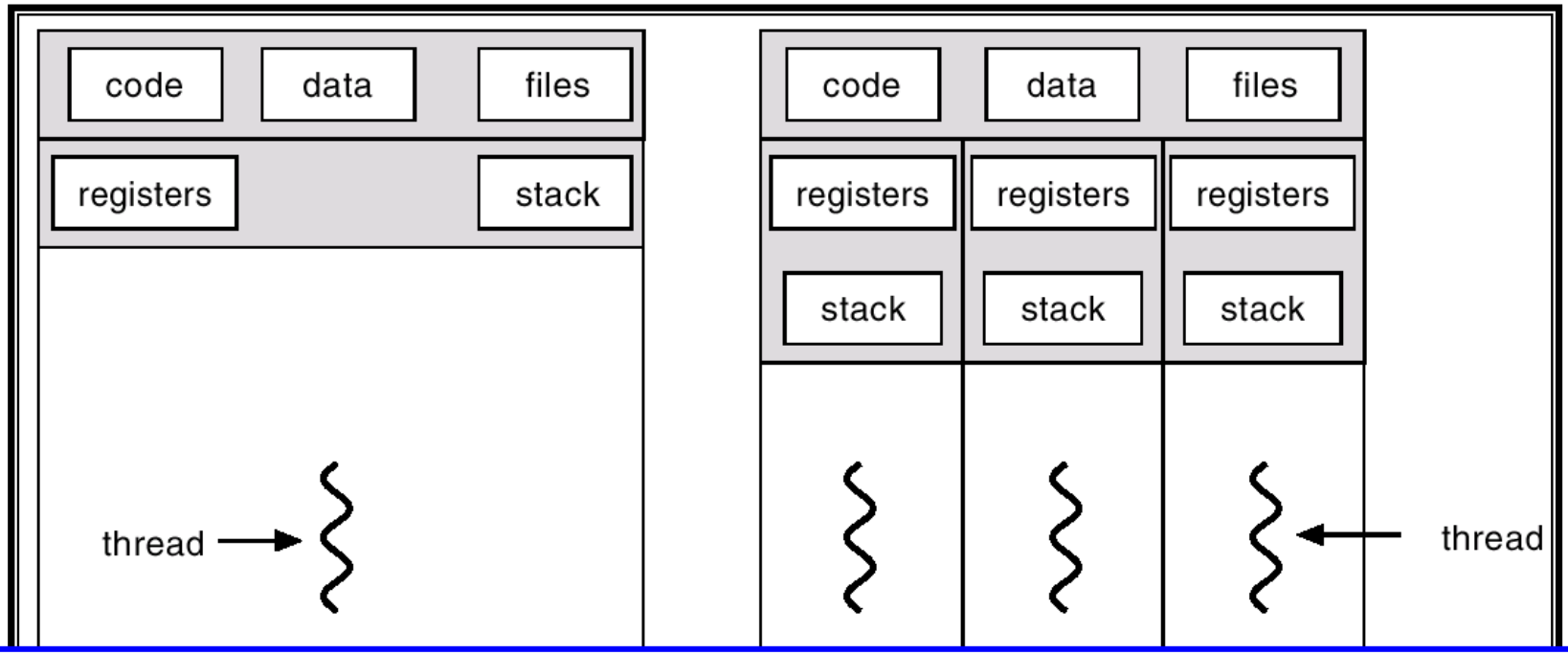
	进程	线程
引入目的	能并发执行,提高资源的利用率和系统吞吐量.	提高并发执行的程度，进一步提高资源的利用率和系统吞吐量.
并发性	较低	较高
基本属性(调度)	资源拥有的基本单位—进程 独立调度/分派的基本单位—进程	资源拥有的基本单位—进程 独立调度/分派的基本单位—线程
基本状态	就绪; 执行;等待	就绪;执行;等待
拥有资源	资源拥有的基本单位—进程	资源拥有的基本单位—进程
系统开销	创建/撤消/切换时空开销较大	创建/撤消/切换时空开销较小
系统操作	创建,撤消,切换	创建,撤消,切换
存在标志	进程控制块PCB	进程控制块PCB，线程控制块TCB

线程的基本概念

- 进程的概念体现出两个特点：资源（代码和数据空间、打开的文件等）以及调度执行。
- 线程是进程内的独立执行代码的实体和调度单元



线程的基本概念



- 线程可以看作是一个执行流，拥有记录自己状态和运行现场的少量数据（栈段和上下文），但没有单独的代码段和数据段，而是与其他线程共享。
- 多个线程共享一个进程内部的各种资源，分别按照不同的路径执行，同时线程也是一个基本调度单位，
- 可以在一个进程内部进行线程切换，现场保护工作量小。

线程的基本概念

系统开销

可以在一个进程内部进行线程切换，现场保护工作量小。一方面通过共享进程的基本资源而减轻系统开销，另一方面提高了现场切换的效率，因此，线程也被称为轻权进程或轻量级进程。

独立性

- 一个线程可以创建和撤消另一个线程
- 当一个线程改变了存储器中的一个数据项时，在其它线程访问这一项时它们能够看到变化后的结果
- 如果一个线程为读操作打开一个文件时，同一个进程中的其它线程也能够从该文件中读。

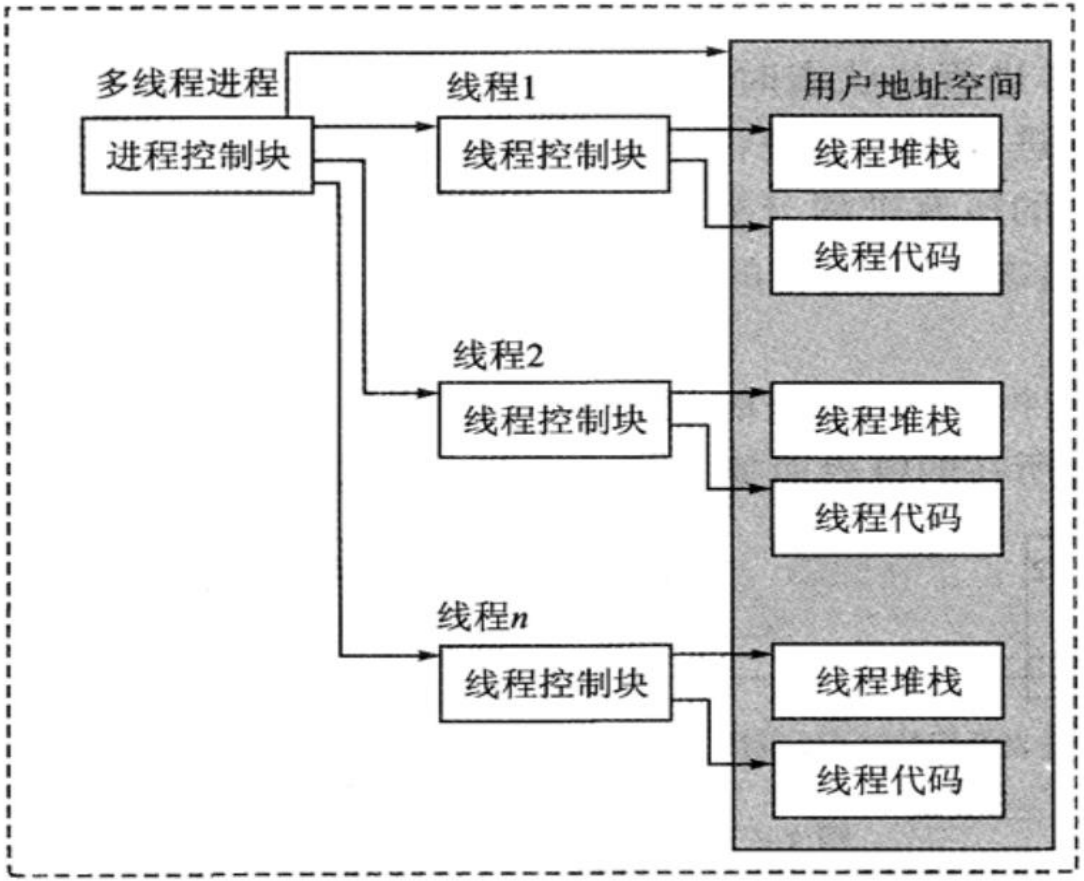
线程的基本概念

- 进程内的所有线程共享进程的很多资源（这种共享又带来了同步问题）

线程间共享	线程私有
进程指令	线程ID
全局变量	寄存器集合（包括PC和栈指针）
打开的文件	栈（用于存放局部变量）
信号处理程序	信号掩码
当前工作目录	优先级
用户ID	

线程的状态

- 线程也是系统中动态变化的实体，它描述程序的运行活动。
- 在OS中的每一个线程都可以利用线程标识符和一组状态参数进行描述，这些信息保存在线程控制块TCB中。



线程的实现方式

- **内核支持线程**（KST - Kernel Supported Threads）

  如Macintosh和OS/2操作系统

- **用户级线程**（ULT - User Level Threads）

  如一些数据库管理系统（如Infomix）

- 组合方式

  把ULT和KST两种方式进行组合，提供了组合方式ULT/KST线程

  如Solaris操作系统

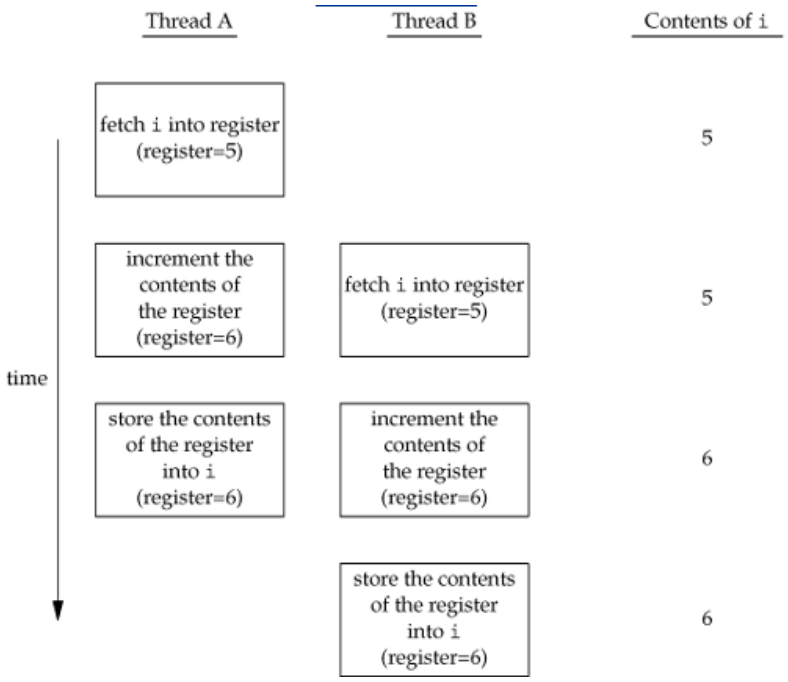
线程的应用场景

线程的出现，在多核心CPU架构下实现了真正意义上的并行执行。

1. 执行后台任务，特别是有一些定时的批量任务，比如定时发送短信、定时生成批量文件
2. 异步处理，如GUI界面刷新，可以通过异步的方式来执行，一方面提升主程序的执行性能；另一方面可以解耦核心功能，防止非核心功能对核心功能造成影响
3. 分布式处理，比如fork/join，将一个任务拆分成多个子任务分别执行
4. BIO模型中的线程任务分发，也是一种比较常见的使用场景，一个请求对应一个线程

线程的互斥问题

- 全局变量在数据段中（内存单元中）通常对一个全局变量的访问，要经历三个步骤
  - 将内存单元中的数据读入寄存器
  - 对寄存器中的值进行运算
  - 将寄存器中的值写回内存单元



主要内容

12.1 线程概述

12.2 线程与进程的比较

12.3 线程的控制

线程与进程的对比

- 线程只拥有少量在运行中必不可少的资源
  - **PC指针：标识当前线程代码执行的位置**
  - **寄存器：当前线程执行的上下文环境**
  - **栈：用于实现函数调用、局部变量**
  - **线程局部变量和私有数据（在栈中申请的数据）**
  - **线程信号掩码（可以设置每个线程阻塞的信号）**
- 进程占用资源多，线程占用资源少，使用灵活
- 线程不能脱离进程而存在，线程的层次关系，执行顺序并不明显，会增加程序的复杂度
- 没有通过代码显示创建线程的进程，可以看成是只有一个线程的进程

线程ID

- 同进程一样，每个线程也有一个线程ID
- 进程ID在整个系统中是唯一的，线程ID只在它所属的进程环境中唯一
- 线程ID的类型是pthread_t，在Linux中的定义如下：
  - **typedef unsigned long int pthread_t**
    (/usr/include/bits/pthreadtypes.h)

获取线程ID

- pthread_self函数可以让调用线程获取自己的线程ID
- 函数原型
  - 头文件： pthread.h
  - pthread_t pthread_self();
- 返回调用线程的线程ID

比较线程ID

- Linux中使用整型表示线程ID，而其他系统则不一定
- FreeBSD 5.2.1、Mac OS X 10.3用一个指向pthread结构的指针来表示pthread_t类型。
- 为了保证应用程序的可移植性，在比较两个线程ID是否相同时，建议使用pthread_equal函数

- 该函数用于比较两个线程ID是否相同
- 函数原型
  - 头文件：pthread.h
  - int pthread_equal(pthread_t tid1, pthread_t tid2);
- 若相等则返回非0值，否则返回0

进程/线程控制操作对比

应用功能	线程	进程
创建	pthread_create	fork,vfork
退出	pthread_exit	exit
等待	pthread_join	wait、waitpid
取消/终止	pthread_cancel	abort
读取ID	pthread_self()	getpid()
同步互斥/通信机制	互斥锁、条件变量、读写锁	无名管道、有名管道、信号、消息队列、信号量、共享内存

主要内容

12.1 线程概述

12.2 线程与进程的比较

12.3 线程的控制

线程的创建

- pthread_create函数用于创建一个线程
- 函数原型
  - 头文件： pthread.h
  - int pthread_create(pthread_t *restrict tidp,
    const pthread_attr_t *restrict attr,
    void *(*start_rtn)(void *),
    void *restrict arg);

线程的创建

- int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr, void *(*start_rtn)(void *), void *restrict arg);

参数

- tidp：指向线程ID的指针，当函数成功返回时将存储所创建的子线程ID
- attr：用于指定线程属性（一般直接传入空指针NULL，采用默认线程属性）
- start_rtn：线程的启动例程函数指针，创建的线程首先执行该函数代码（可以调用其他函数）
- arg：向线程的启动例程函数传递信息的参数

返回值

- 成功返回0，出错时返回各种错误码

创建子线程代码示例

```
void *childthread(void){
  int i;
  for(i=0;i<10;i++){
    printf( “childthread message\n” );
    sleep(100);}}

int main(){
  pthread_t tid;
  printf( “create childthread\n” );
   pthread_create(&tid,NULL,(void *) childthread,NULL);
   sleep(3);
   printf( “process exit\n” );
}
```

线程的终止

- 线程的三种终止方式
  - 线程从启动例程函数中返回，函数返回值作为线程的退出码
  - 线程被同一进程中的其他线程取消
  - 线程在任意函数中调用pthread_exit函数终止执行

线程终止函数

函数原型

- 头文件：pthread.h
- void pthread_exit(void *rval_ptr);

参数

- rval_ptr：该指针将传递给pthread_join函数（与exit函数参数用法类似）

父线程等待子线程终止

函数原型

- 头文件：pthread.h
- int pthread_join(pthread_t thread, void **rval_ptr);

- 将参数thread指定的子线程合入主线程，主线程阻塞等待子线程结束，然后回收子线程资源。
- 如果线程已经结束，那么该函数会立即返回。并且thread指定的线程必须是joinable的。

返回值

- 成功返回0，否则返回错误编号

pthread_join函数

- int pthread_join(pthread_t thread, void **rval_ptr);

- 参数
  - thread：需要等待的子线程ID
  - rval_ptr：（若不关心线程返回值，可直接将该参数设置为空指针NULL）
    - 若线程从启动例程返回，rval_ptr将包含返回码
    - 若线程被取消，rval_ptr指向的内存单元值置为PTHREAD_CANCELED
    - 若线程通过调用pthread_exit函数终止，rval_ptr就是调用pthread_exit时传入的参数

创建并等待子线程代码示例

```
void *childthread(void){
   int i;
   for(i=0;i<10;i++){
     printf( “childthread message\n” );
     sleep(100);}}

int main(){
     pthread_t tid;
     printf( “create childthread\n” );
      pthread_create(&tid, NULL, (void *) childthread, NULL);
     pthread_join(tid, NULL);
     printf( “childthread exit process exit\n” );
}
```

取消线程

- 线程调用该函数可以取消同一进程中的其他线程（即让该线程终止）

函数原型

- 头文件： pthread.h
- int pthread_cancel(pthread_t tid);

参数与返回值

- tid：需要取消的线程ID
- 成功返回0，出错返回错误编号

取消线程

- 在默认情况下，pthread_cancel函数与线程ID等于tid的线程自身调用pthread_exit函数（参数为PTHREAD_CANCELED）效果等同
- 线程可以选择忽略取消方式或者控制取消方式
- pthread_cancel并不等待线程终止，它仅仅是提出请求

pthread_detach函数

- 在任何一个时间点上，线程是可结合的（joinable）或者是分离的（detached）
  - 可结合的线程能够被父线程回收其资源和杀死。在被父线程回收之前，它的存储器资源（例如栈）是不释放的
  - 分离的线程是不能被父线程回收或杀死的，它的存储器资源在它终止时由系统自动释放
- 若线程已经处于分离状态，线程的底层存储资源可以在线程终止时立即被收回
- 当线程被分离时，并不能用pthread_join函数等待它的终止状态，此时pthread_join返回EINVAL
- pthread_detach函数可以使线程进入分离状态

pthread_detach函数

- 函数原型
  - 头文件：pthread.h
  - int pthread_detach(pthread_t tid);
- 参数与返回值
  - tid：进入分离状态的线程的ID
  - 成功返回0，出错返回错误编号

线程属性

- 前面讨论pthread_create时，针对线程属性，传入的参数都是NULL。
- 实际上，可以通过构建pthread_attr_t结构体，设置若干线程属性
- 要使用该结构体，必须首先对其进行初始化；使用完毕后，需要销毁它

线程属性

■ POSIX规定的一些线程属性

```
typedef struct{
    int                 detachstate;   // 线程的分离状态
    int                 schedpolicy;   // 线程调度策略
    structsched_param   schedparam;    // 线程的调度参数
    int                 inheritsched;  // 线程的继承性
    int                 scope;         // 线程的作用域
    size_t              guardsize;     // 线程栈末尾的警戒缓冲区大小
    int                 stackaddr_set; // 线程的栈设置
    void*               stackaddr;     // 线程栈的位置
    size_t              stacksize;     // 线程栈的大小
} pthread_attr_t;
```

初始化和销毁

- 函数原型

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

- 参数与返回值
  - 成功返回0，否则返回错误编号
  - attr：线程属性，确保attr指向的存储区域有效
  - 为了移植性，pthread_attr_t结构对应用程序是不可见的，应使用设置和查询等函数访问属性

线程属性操作示例代码

```
#include <pthread.h>
#include <sched.h>
int main(void)
{
  int ret;
  pthread_t pid;   /* 线程ID */
  pthread_attr_t pattr;  /* 线程属性结构体 */
  struct sched_param param; /* 线程优先级结构体 */
  pthread_attr_init(&pattr);  /* 初始化线程属性对象，这时是默认值 */
  pthread_attr_setscope(&pattr, PTHREAD_SCOPE_SYSTEM); /* 设置线程绑定 */
  pthread_attr_getschedparam(&pattr, &param); /* 修改线程优先级 */
  param.sched_priority = 20;
  pthread_attr_setschedparam(&pattr, &param);
  /* 使用设置好的线程属性来创建一个新的线程 */
  ret = pthread_create(&pid, &pattr, (void *)thread, NULL);
  ……
```

初始化线程属性对象

属性	缺省值	描述
scope	**PTHREAD_SCOPE_PROCESS**	新线程与进程中的其他线程发生竞争
detachstate	**PTHREAD_CREATE_JOINABLE**	线程可以被其它线程等待
stackaddr	**NULL**	新线程具有系统分配的栈地址
stacksize	**0**	新线程具有系统定义的栈大小
priority	**0**	新线程的优先级为0
inheritsched	**PTHREAD_EXPLICIT_SCHED**	新线程不继承父线程调度优先级
schedpolicy	**SCHED_OTHER**	新线程使用优先级调用策略

获取线程栈属性

- 函数原型

```
#include<pthread.h>
int pthread_attr_getstack(
        const pthread_attr_t *attr,
        void **stackaddr,   size_t *stacksize);
```

- 参数与返回值
  - attr：线程属性
  - stackaddr：该函数返回的线程栈的最低地址
  - stacksize：该函数返回的线程栈的大小
  - 成功返回0，否则返回错误编号

设置线程栈属性

- 函数原型

```
#include<pthread.h>
int pthread_attr_setstack(
        const pthread_attr_t *attr,
         void *stackaddr, size_t *stacksize);
```

- 当用完线程栈时，可以再分配内存，并调用本函数设置新建栈的位置

设置线程栈属性

- 参数与返回值
  - attr：线程属性
  - stackaddr：新栈的内存单元的最低地址，通常是栈的开始位置；对于某些处理器，栈是从高地址向低地址方向伸展的，stackaddr就是栈的结尾
  - stacksize：新栈的大小
  - 成功返回0，否则返回错误编号

多线程实例

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#define LEN 100000
```

```
int num = 0;
void* thread_func(void* arg) {
    for (int i = 0; i< LEN; ++i) {
        num += 1;
    }
    return NULL;
}
```

```
int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, (void*)thread_func, NULL);
    pthread_create(&tid2, NULL, (void*)thread_func, NULL);

    char* rev = NULL;
    pthread_join(tid1, (void *)&rev);
    pthread_join(tid2, (void *)&rev);

    printf("correct result=%d, wrong result=%d.\n", 2*LEN, num);
    return 0;
}
```

未进行线程同步，结果错误！！

多线程实例

使用互斥锁

```
#include <pthread.h>
 int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr); /*初始化互斥量*/
 int pthread_mutex_destroy(pthread_mutex_t *mutex); /*销毁互斥量*/
 int pthread_mutex_lock(pthread_mutex_t *mutex);
 int pthread_mutex_trylock(pthread_mutex_t *mutex);
 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

多线程实例

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

#define LEN 100000
int num = 0;
```

```
void* thread_func(void* arg) {
    pthread_mutex_t* p_mutex  = (pthread_mutex_t*)arg;
    for (int i = 0; i< LEN; ++i) {
        pthread_mutex_lock(p_mutex);
        num += 1;
        pthread_mutex_unlock(p_mutex);
    }
    return NULL;
}
```

```
int main() {
    pthread_mutex_t m_mutex;
    pthread_mutex_init(&m_mutex, NULL);

    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, (void*)thread_func, (void*)&m_mutex);
    pthread_create(&tid2, NULL, (void*)thread_func, (void*)&m_mutex);

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

    pthread_mutex_destroy(&m_mutex);
    printf("correct result=%d, result=%d.\n", 2*LEN, num);
    return 0;
}
```

课堂练习

1. 线程是调度的基本单位，但不是资源分配的基本单位（T）。

2. 隶属于同一进程的多个线程共享一组CPU寄存器值，并共享一个堆栈（F）。

3. 即便线程不作为资源分配单位，线程之间仍可能因为竞争影响并行执行（T）。

4. 进程控制块TCB中存储的内容是（C）：

A. User ID　　B. Memory map

C. The machine state(registers, program counter)

D. 打开的文件描述符

课后作业

用线程方式实现生产者和消费者问题