北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

并行编程原理与实践

5. 多线程编程

王一拙、计卫星

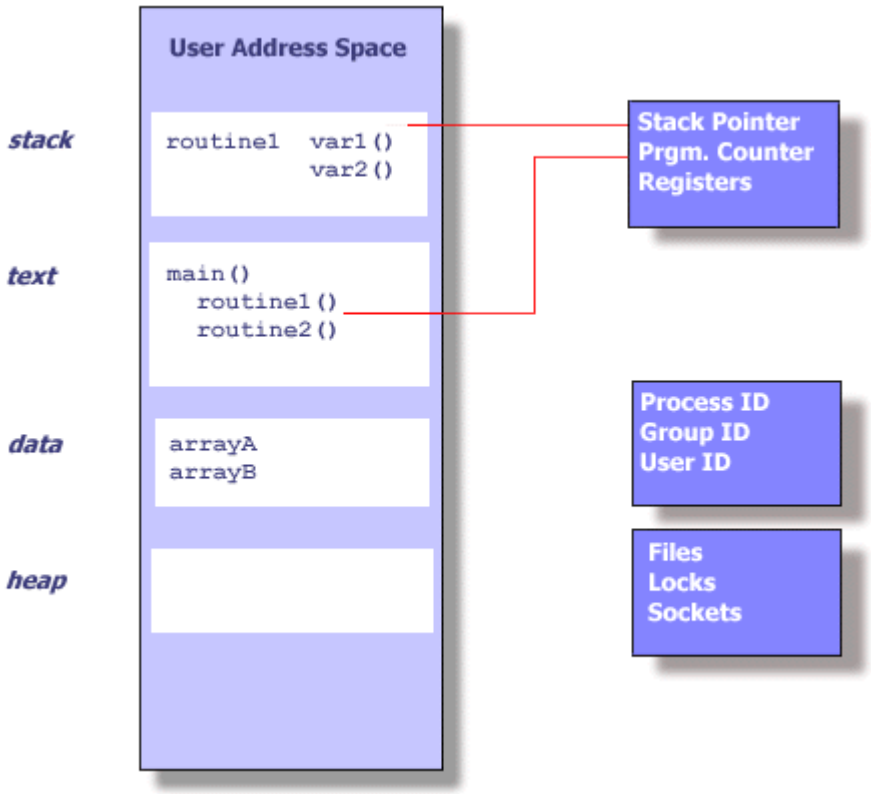
北京理工大学计算机学院

目录 | CONTENTS

1 多线程的基本概念

进程（Process）与线程（Thread）

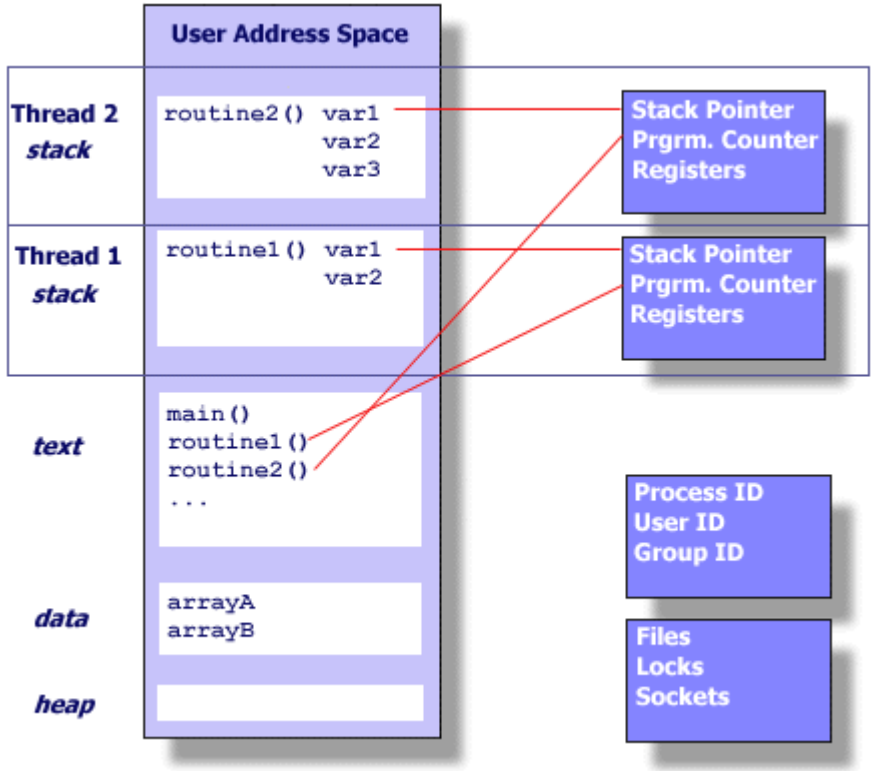
- 进程是程序在操作系统里的一次执行
- 进程是操作系统资源分配的基本单位
- UNIX进程包含：
  - 进程ID，用户ID，组ID
  - 工作目录
  - 程序指令
  - 寄存器，堆区，栈区，数据区
  - 文件描述符
  - 动态库
  - 环境变量
  - ....



https://computing.llnl.gov/tutorials/pthreads/

进程（Process）与线程（Thread）

- 线程是存在于进程内且可被操作系统独立调度的指令流实体
- 一个进程可以拥有多个线程；一个进程至少需要一个线程作为它的指令执行体。
- 进程的所有信息对该进程的所有线程都是共享的包括：可执行的程序文本、程序的全局内存、堆内存、文件描述符等。
- 线程独有的包括：线程ID、寄存器、栈、等待或挂起的信号、调度属性（策略和优先级）、线程私有数据等。



https://computing.llnl.gov/tutorials/pthreads/

为什么要用线程？

- 线程开销小。据统计，一个进程的创建开销大约是一个线程开销的30倍左右，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。
- 线程间通信更方便、快捷。同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用。
- 使多核系统更加有效。操作系统会保证当线程数不大于核心数目时，不同的线程运行于不同的处理核上。
- 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

Platform	fork()	pthread_create()
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.9
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.7
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.2
AMD 2.4 GHz Opteron (8 cores/node)	17.6	1.4
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	1.6
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	1.7
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	2.1
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.6
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	2.0

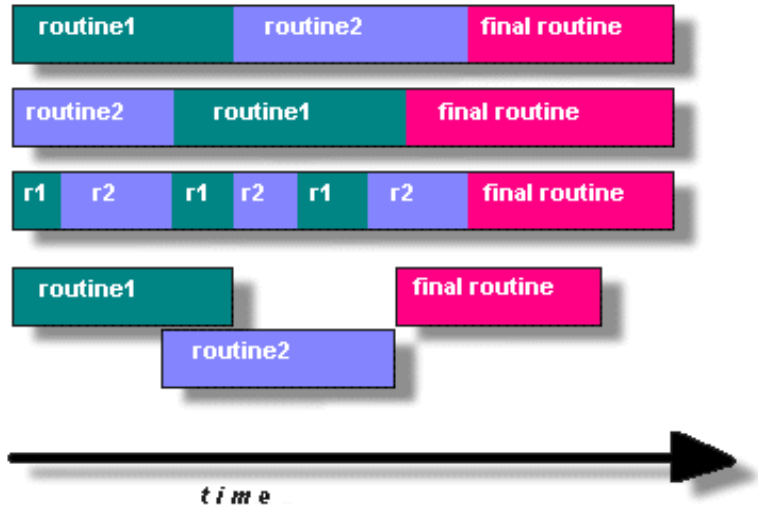
https://computing.llnl.gov/tutorials/pthreads/

https://computing.llnl.gov/tutorials/pthreads/

可多线程实现的典型任务

- 数据或任务可以分片并发执行
- I/O等待时间很长的 (数据库的大查询)
- 某些部分是CPU密集型，其它部分则不是

核心概念是任务需要可并发执行，同时机器应为多处理器和多核处理器

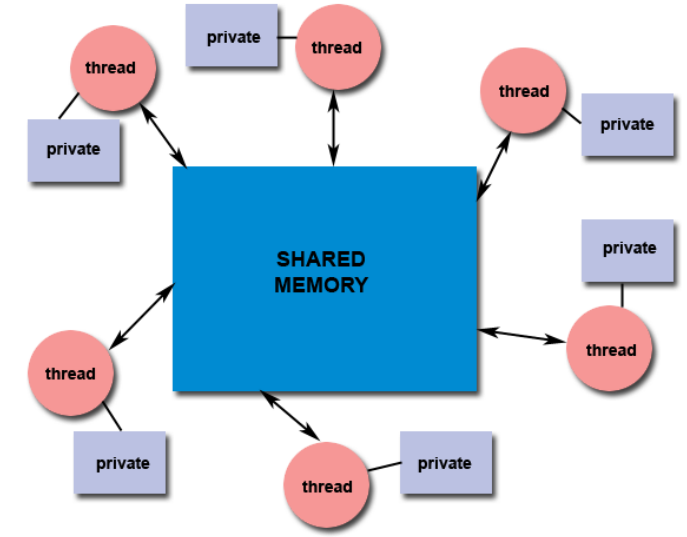


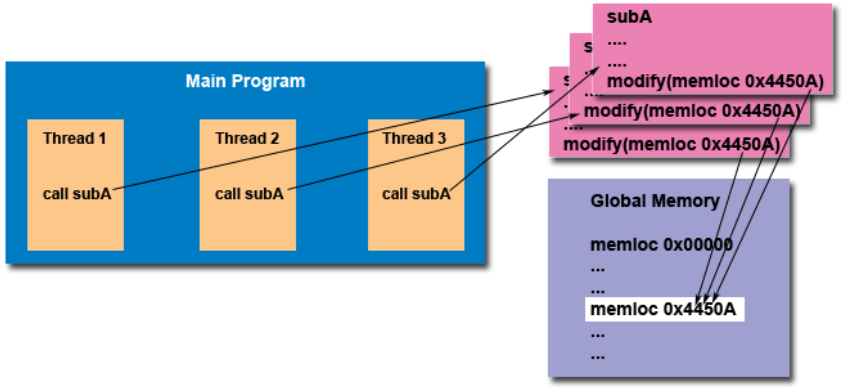
多线程协作模型

- Manager/worker：主线程（管理者）打包任务，放进任务列表，子线程（工作者）从任务列表提取任务，执行任务
- Pipeline：每个任务被分为几个子阶段，流水线执行
- Peer：跟Manager/worker类似，只是主线程在打包完任务后也参与到任务的执行

线程安全

- 所有线程共享进程的全局资源，程序员需要负责保证对共享数据的同步访问
- 从每个线程的入口函数开始调用的一系列函数都必须是线程安全的函数，包括库函数
- 在未确定某个库支持多线程的情况下，需假定该库不支持多线程，使用“顺序”方式使用该库的函数
- 线程的使用限制：
  - 最大线程数量
  - 线程堆栈大小

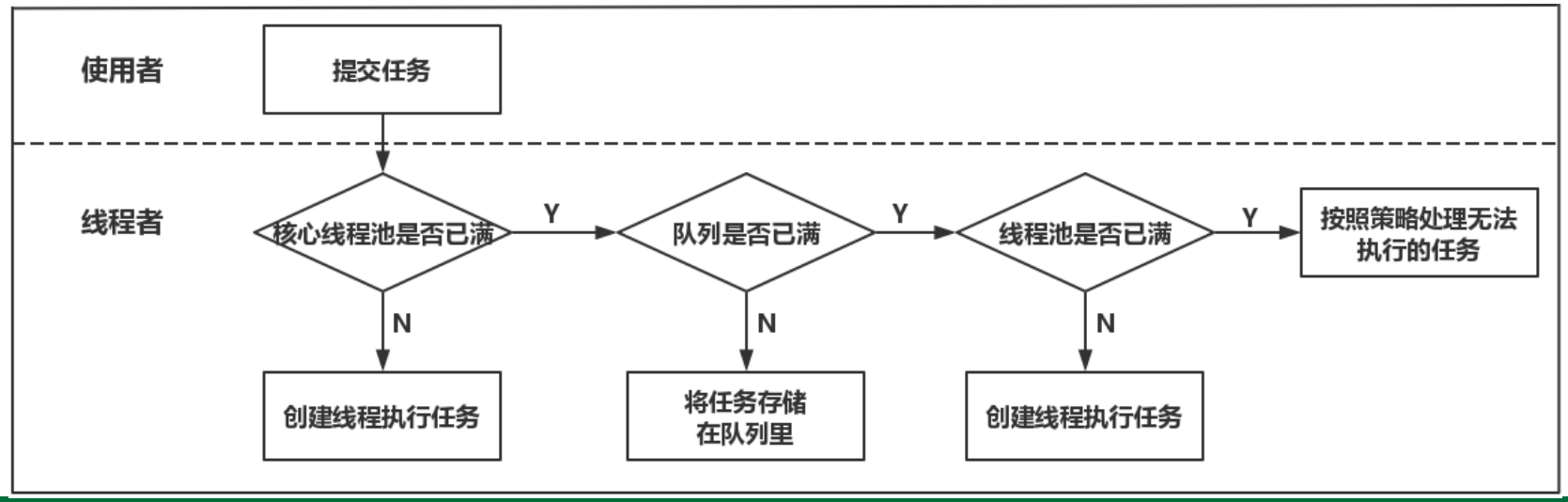




https://computing.llnl.gov/tutorials/pthreads/

线程池技术

- 在程序设计中，并不是来一个任务/请求就创建一个线程，执行完毕线程退出。
- 程序初始化时创建一个线程池，线程池中有n个线程。
- 当一个请求来时，从线程池中取一个线程，完成请求后线程再放回线程池。
- 节省了频繁创建线程和销毁线程的开销。



2 Pthreads线程库

```
#include <pthread.h>
gcc  main.c  –pthread
```

Pthreads线程库简介

- 历史上，计算机软硬件厂商都开发了各自私有的多线程实现
- 开发可移植的多线程程序非常困难
- IEEE于1995年完成编程接口的标准化，POSIX 1003.1c standard
- 遵循该标准的多线程实现称为POSIX threads或Pthreads.
- Pthreads是一组C语言数据类型和库函数的集合，编译时包含pthread.h头文件，链接时包含-pthread选项即可。

Compiler / Platform	Compiler Command	Description
INTEL Linux	`icc -pthread`	C
	`icpc -pthread`	C++
PGI Linux	`pgcc -lpthread`	C
	`pgCC -lpthread`	C++
GNU Linux, Blue Gene	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++
IBM Blue Gene	`bgxlc_r / bgcc_r`	C (ANSI / non-ANSI)
	`bgxlC_r, bgxlc++_r`	C++

- Pthreads线程库提供的函数一般都以pthread开头

前缀	功能集合
pthread_	线程或子线程
pthread_attr_	线程对象属性
pthread_mutex_	互斥量
pthread_mutexattr_	互斥量对象属性
pthread_cond_	条件变量
pthread_condattr_	条件变量属性

Pthreads线程库

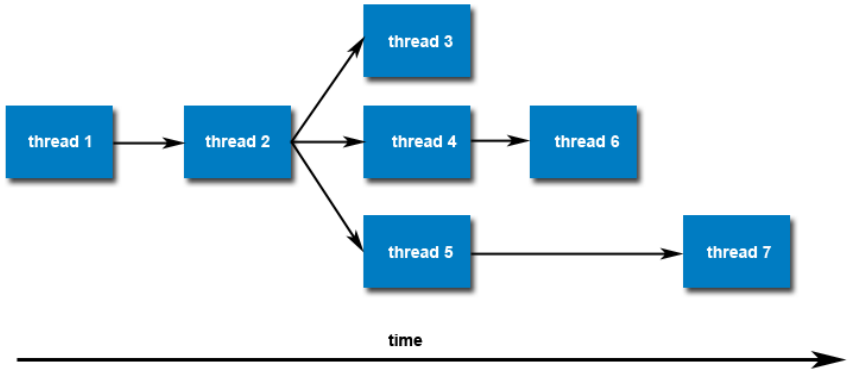
- **Pthreads线程库总共有100多个API，其中常用的3类API是：**
  - 线程管理：创建线程，终止线程，分离线程，等待线程，线程属性的设置与查询
  - 互斥变量（mutex）： 创建/销毁互斥变量，对互斥变量进行加解锁
  - 条件变量：解决线程间需要基于特定的条件进行互相通知的问题，创建和销毁条件变量，基于特定值等待某个条件变量，通知某个条件变量

3 线程的管理

线程创建：pthread_create()

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

- 参数和返回值
  - thread：当创建成功时，将线程ID存储在thread指向的内存区域
  - attr：用于定制各种不同的线程属性，将在后面部分讨论。通常可设为NULL，采用默认线程属性
  - start_routine：线程的入口函数，即新创建的线程从该函数开始执行。该函数只有一个参数，即arg，返回一个指针
  - 成功返回0，出错时返回各种错误码



线程终止函数：

```
void pthread_exit(void * retval)
int pthread_cancel(pthread_t thread);
```

- 线程终止的几种情况：
  - 线程的“main”函数正常结束
  - 线程调用pthread_exit函数退出
  - 被其它线程以pthread_cancel方式取消了
  - 整个进程终止了，如主线程调用exit()/exec()
  - 主线程main函数结束，未调用pthread_exit

3 Pthreads线程管理

- 创建和结束线程示例

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

3 Pthreads线程管理

- 创建和结束线程示例
- 给线程传递参数

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long taskids[NUM_THREADS];

   for(t=0; t<NUM_THREADS; t++)
   {
      taskids[t] = t;
      printf("Creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
      ...
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

```c
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];
```

线程属性

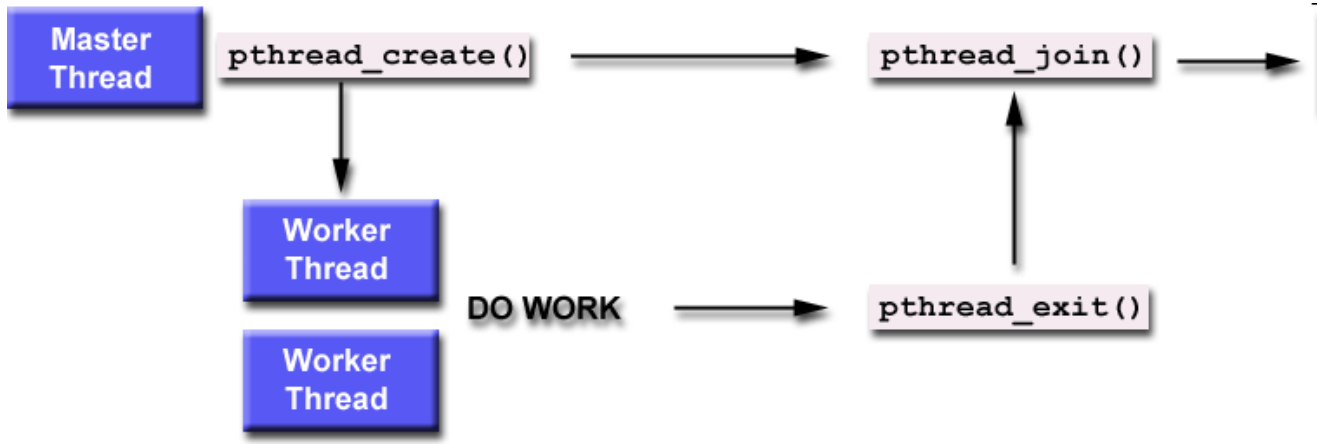
```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- 线程有默认属性，可以通过属性对象设置线程属性
- 常用属性：
  - 分离（Detached）或可等待（Joinable）状态
  - 堆栈大小
  - 堆栈地址
  - 调度策略
  - ……

线程的等待和分离

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

- pthread_join函数会挂起当前线程的执行，直到等待到想要等待的子线程结束。等待的线程必须在可加入的状态，即没有对线程调用pthread_detach函数，或者创建线程时把其属性设置为PTHREAD_CREATE_DETACHED。



线程的等待和分离

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

- pthread_join函数会挂起当前线程的执行，直到等待到想要等待的子线程结束。等待的线程必须在可加入的状态，即没有对线程调用pthread_detach函数，或者创建线程时把其属性设置为PTHREAD_CREATE_DETACHED。
- 主线程创建子线程后可调用pthread_detach分离子线程，子线程自行结束，自我回收内存资源
- 设置和查询线程分离属性：

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

堆栈管理

```
int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
                              void **restrict stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

- 默认线程堆栈大小变化很大，可获得的最大堆栈大小也相差很大，并且可能取决于处理器核数。
- 常用属性：
  - 分离（Detached）或可等待（Joinable）状态

Node Architecture	#CPUs	Memory (GB)	Default Size (bytes)
Intel Xeon E5-2670	16	32	2,097,152
Intel Xeon 5660	12	24	2,097,152
AMD Opteron	8	16	2,097,152
Intel IA64	4	8	33,554,432
Intel IA32	2	4	2,097,152
IBM Power5	8	32	196,608
IBM Power4	8	16	196,608
IBM Power3	16	16	98,304

线程调度策略

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
int sched_getscheduler(pid_t pid);
```

- POSIX 标准中的线程调度策略：
  - 先入先出策略 (SCHED_FIFO)
    - SCHED_FIFO 是基于队列的调度程序，对于每个优先级都会使用不同的队列。
  - 循环策略 (SCHED_RR)
    - SCHED_RR 与 FIFO 相似，不同的是前者的每个线程都有一个执行时间配额。
  - 自定义策略 (SCHED_OTHER)
    - SCHED_OTHER 是缺省的Linux时间共享调度策略，保证线程调度公平。

线程绑定 — CPU亲和力

```
int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset);
int pthread_getaffinity_np(pthread_t thread, size_t cpusetsize, cpu_set_t *cpuset);
int sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask);
int sched_getaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask);

//初始化，设为空
void CPU_ZERO (cpu_set_t *set);
//将某个cpu加入cpu集中
void CPU_SET (int cpu, cpu_set_t *set);
//将某个cpu从cpu集中移出
void CPU_CLR (int cpu, cpu_set_t *set);
//判断某个cpu是否已在cpu集中设置了
int CPU_ISSET (int cpu, const cpu_set_t *set);
```

线程优先级

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

- 线程优先级参数param是一个struct sched_param结构，目前仅含一个sched_priority整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即SCHED_RR或SCHED_FIFO）时才有效，缺省为0

```
#include <pthread.h>
#include <sched.h>
pthread_attr_t attr;
pthread_t tid;
sched_param param;
int newprio=20;
pthread_attr_init(&attr);
pthread_attr_getschedparam(&attr, &param);
param.sched_priority=newprio;
pthread_attr_setschedparam(&attr, &param);
pthread_create(&tid, &attr, (void *)myfunction, myarg);
```

3 Pthreads线程管理

- **获得当前线程ID**：pthread_t pthread_self( );
- **判断两个线程是否相同**：int pthread_equal(tid1, tid2);
- **让出当前线程的CPU占有权**：int sched_yield( );
- **初始化函数**：在多线程调用时仅会被执行一次

  int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));

  - pthread_once声明一个初始化函数，某个线程第一次调用init_routine时执行这个函数，以后的调用将被忽略
  - pthread_once_t once = PTHREAD_ONCE_INIT; 其实就是一个互斥量，初始为0

4 线程的同步

多个线程为什么需要同步？

- 多个线程同时读写共享变量，可能出现数据不一致的问题

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

https://computing.llnl.gov/tutorials/pthreads/

互斥量（Mutex）

- 可以使用Pthreads库提供的互斥量来确保同一时间只有一个线程访问数据
- 互斥量mutex，本质上就是一把锁
  - 在访问共享资源前，对互斥量进行加锁
  - 在访问完成后释放互斥量上的锁
  - 对互斥量进行加锁后，其它试图再次对互斥量加锁的线程将会被阻塞，直到锁被释放

Thread 1	Thread 2	Thread 3
Lock	Lock	
A = 2	A = A+1	A = A*B
Unlock	Unlock	

互斥量的创建和销毁

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- 有两种方法创建互斥锁，静态方式和动态方式
  - mutex：即互斥量，类型是pthread_mutex_t
  - attr：设置互斥量的属性，通常可采用默认属性，即可将attr设为NULL
- pthread_mutex_destroy用来销毁有init函数创建的互斥量，如果是静态创建的则无需调用destroy函数
- 创建和销毁函数成功返回0，出错返回错误码

互斥量的加锁和解锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- 函数pthread_mutex_trylock的语义与函数pthread_mutex_lock类似，不同的是在锁已经被占据时返回EBUSY，而不是阻塞等待
- 死锁的情况：
  - 线程试图对同一个互斥量加锁两次
  - 线程A先对A互斥量加锁，线程B对B互斥量加锁；然后线程A又去对B互斥量加锁，而线程B又去对A互斥量加锁

读写锁

- 读写锁与互斥量类似，不过读写锁允许更高的并行性
- 互斥量只有两种状态：锁住、不加锁
- 读写锁三种状态
  - 读模式下加锁状态
  - 写模式下加锁状态
  - 不加锁状态
- 一次只有一个线程可以占有写模式的读写锁、但多个线程可以同时占用读模式的读写锁

Pthreads读写锁接口

```
pthread_rwlock_t rwlock = PTHREASD_RWLOCK_INITIALIZER;
int pthread_rwlock_init(pthread_rwlock_t *restrict   rwlock, const
          pthread_rwlockattr_t   *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_tryrdlock(pthread_rwlock_t   *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t   *rwlock);
```

条件变量

- 条件变量是另一种同步机制。多个线程可以等待同一个条件，条件满足时被唤醒。可以唤醒一个线程，也可以唤醒所有等待的线程
- 互斥变量可以实现对一个共享数据的独占性访问，条件变量可以实现基于某个特定数值的停等-通知同步
- 没有条件变量，程序员可用使用轮询某个变量来实现停等-通知同步，但是非常消耗系统资源
- 条件变量可以使线程处于等待状态而不消耗资源
- 条件变量跟互斥量mutex一起使用
- 条件变量的类型是pthread_cond_t

条件变量的创建和销毁

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *restrict cond, pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

- 有两种方法创建条件变量，静态方式和动态方式
  - attr：条件变量属性，通常设为NULL，即采用默认属性
- 创建和销毁函数成功返回0，出错返回错误码

等待条件变量

```
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t  *restrict cond,
                pthread_mutex_t  *restrict mutex, const struct timespec  *restrict timeout);
```

- pthread_cond_wait函数将使调用线程进入阻塞状态，直到条件为真
- 在调用pthread_cond_wait前，需要使互斥量处于锁住状态
- pthread_cond_wait函数内部的特殊操作
  - 在线程阻塞前，调用pthread_mutex_unlock
  - 在线程唤醒后，调用pthread_mutex_lock
- timeout指定了等待时间，如果时间到了还没被唤醒，那么返回ETIMEDOUT

```
pthread_mutex_lock
pthread_cond_wait
pthread_mutex_unlock
```

给条件变量发信号

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- pthread_cond_signal唤醒某一个等待该条件的线程
- pthread_cond_broadcast唤醒等待该条件的所有线程
- 如果当时没有线程等待在条件变量上，这个唤醒动作是不会被保留的

Thread A

```
pthread_mutex_lock(&lock);
count ++ ;
if (count >=MAX_COUNT){
    pthread_cond_signal(&cond);
}
pthread_mutex_unlock(&lock)
```

Thread B

```
pthread_mutex_lock(&lock);
while (count < MAX_COUNT) {
   pthread_cond_wait(&cond, &lock);
}
count--;
pthread_mutex_unlock(&lock)
```


北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY
谢谢!
德以明理 学以精工