线程

多进程的局限与多线程的优势所在

- 进程间通信较为复杂
  - 进程和进程之间有着完全不同的地址空间，即便是父、子进程。除去只读代码段共享以外，无其它内存共享，必须通过诸如信号量、共享内存或者对消息队列进行通信，加大了编码的难度与复杂度
  - 同一个进程间的线程完全共享进程空间，包括文件描述符、堆以及静态变量
- 创建子进程的代价不菲
  - 尽管子进程的创建(fork)采用写时复制的技术省去了一部分内存拷贝，但是像内存页表、文件描述符等资源仍然需要复制
  - 在创建线程时无需进行内存页表或者是文件描述符的复制，直接共享。clone()创建线程的速度可能要比fork()快上10倍左右

POSIX线程的创建与终止

- 线程的创建
  - *原型: int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg);*
    - 返回0表示调用成功，返回正值表示调用失败
    - 和大多数的C函数返回有所不同，需注意
  - pthread_t
    - 为线程标识(或线程ID)，通常是无符号长整型，定义成pthread_t类型是为了可一致性
    - 线程可使用pthread_self(void)调用来获取自身的线程ID，调用同样返回pthread_t类型
  - pthread_attr_t
    - 用于设置线程的各种属性，例如线程栈的位置和大小，线程调度策略和优先级等。该参数通常为NULL
  - start与arg
    - 线程所执行函数与参数，和clone()调用没什么区别
- 线程的终止
  - 线程函数运行完毕，执行return语句返回
  - 调用pthread_exit(void retval)退出
  - 其余线程调用pthread_cancel(pthread_t thread)
    - thread线程不一定立即终止
  - **主线程调用pthread_exit()，或者是exit()、_exit()，或是执行return语句，则进程中的所有线程立即终止**
    - 这里和Python、Java的线程实现稍有不同
- 连接(join)已终止的线程
  - *原型: pthread_join(pthread_t thread, void **retval)*
    - 返回0表示调用成功，返回正值表示调用失败
  - 函数pthread_join()等待由thread标识的线程终止，并获取其返回值，类似于waitpid()系统调用
  - 当线程处于未分离的状态时，必须调用pthread_join来进行连接，否则将会产生僵尸线程(即线程数据未被清空)，同僵尸进程一样，当积累了过多的僵尸线程时，将无法创建更多线程
- 线程的分离(detach)
  - 在默认情况下， 线程是可连接的，即已经运行结束的线程可通过pthread_join()获取其返回状态
    - 但是有时更希望线程结束后能够自动清理并移除不再使用的数据，而不是由用户调用pthread_join()进行移除
  - *int pthread_detach(pthread_t thread);*
    - 返回0表示调用成功，返回正值表示调用失败
    - 将某个线程标记为分离状态，当线程结束后自动清理相关内容
  - 当一个线程处于分离状态时，不能再使用pthread_join()对其进行连接，也没有办法将其状态变更为"可连接"。就是个一锤子买卖

POSIX线程同步

- 保护对共享变量的访问: 互斥量(mutex)
  - 为什么需要互斥量?

```
static int count = 0;

int main() {
    count++;
    return 0;
}
```

    - 这是一段及其简单的使用静态变量作为计数器的代码，使用 gcc -S 查看汇编语言
    - `count++`这一条C语句将会对应多条汇编语句，而每一个汇编语句都会对应一个CPU操作，故`count++`需要CPU的多次操作才能得到结果
    - CPU在执行逻辑时很有可能被中断，例如某线程的时间片到期，调度程序执行另一个线程。CPU会保存当前线程执行上下文，如堆栈、函数调用点等，以便后续的恢复
      - 在中断后，其余线程完全有可能去修改count这一静态变量，修改完成后，原有的线程恢复执行，从寄存器、堆栈中取出数据并继续执行，那么此时count的值就会出现混乱
    - 所以，我们需要一个机制，使得`count++`这条语句在执行时仅有一个线程对其操作，且在执行完毕之前，CPU不会中断执行
      - 通常称之为原子操作
  - 互斥量的初始化
    - 静态初始化
      - static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
      - PTHREAD_MUTEX_INITIALIZER只能初始化静态分配的互斥量，并且采用互斥量的默认属性
    - 动态初始化
      - *int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);*
        - 返回0表示调用成功，返回正值表示调用失败
      - 由init方法初始化的互斥量需要调用 pthread_mutex_destroy(pthread_mutex *mutex)进行销毁。因为动态初始化的mutex可能位于堆、栈或者是静态变量存储区中
    - 销毁动态分配的互斥量
      - *int pthread_mutex_destroy(pthread_mutex_t *mutex);*
  - 互斥量的属性
    - 属性初始
      - *int pthread_mutexattr_init(pthread_mutxattr_t *attr);*
    - 属性设置
      - *int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int kind);*
    - 属性销毁
      - *int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);*
  - 互斥量的加锁与解锁
    - 加锁
      - *int pthread_mutex_lock(pthread_mutex_t *mutex);*
      - 当互斥量处于未锁定状态时，调用会锁定互斥量并立即返回
      - 当互斥量处于锁定时，调用会一值阻塞，直到该互斥量被解锁
    - 解锁
      - *int pthread_mutex_unlock(pthread_mutex_t *mutex);*
- 通知状态的改变: 条件变量(condition variable)
  - 为什么需要条件变量?
    - 在消息队列领域中，有一个非常重要的应用就是发布-订阅: 当发布方将产生的消息发送给所有订阅该事件的主体
      - 那么当没有消息产生时，订阅方该怎么做?
        - 不停的轮询，一个while True解决所有问题，就是有点儿费CPU
        - 同样是轮询，但是在每次循环将要结束时，睡眠(sleep)一段时间，让出CPU。虽然省CPU，但是接收订阅事件不能做到实时
        - 直接阻塞，当有订阅事件发生时将其唤醒。这样既节省CPU，同时也能做到实时消费
    - 互斥量防止多个线程同时访问同一共享变量，条件变量允许一个线程就某个共享变量的状态变化通知其它线程，并让其它线程阻塞于这一通知
    - Python线程池以及Java线程池均采用条件变量实现，且一旦出现条件变量，互斥量必然出现
  - 条件变量的初始化
    - 静态初始化
      - static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
      - 与互斥量的初始化一样，静态初始化将采用默认的条件变量属性
    - 动态初始化
      - *原型: int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);*
    - 销毁动态分配的条件变量
      - *int pthread_cond_destroy(pthread_cond_t *cond);*
  - 通知和等待条件变量
    - 条件变量的主要操作是发送信号和等待
      - 发送信号
      - 通知一个或多个处于等待状态的线程，某个共享变量的状态已经改变
      - 等待信号
      - 在收到一个通知前将一直处于阻塞状态
      - *这里的发送信号与Linux Signal无关，将其理解为发送通知即可*
    - 发送信号
      - *int pthread_cond_signal(pthread_cond_t *cond);*
        - 仅保证唤醒至少一条遭到阻塞的线程
        - 当所有等待线程均在执行同一个任务时，采用该调用会更加高效
      - *int pthread_cond_broadcast(pthread_cond_t *cond);*
        - 唤醒全部造cond阻塞的线程
        - 当所有等待线程执行的任务不同，或者说，各线程关联于条件变量的判定条件不同时，调用此函数更加合理
          - 因为发送信号的线程也不知道哪条线程在等待哪一个条件变量，所以将等待线程全部唤醒，由线程自己去做判断
    - 等待信号
      - *int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);*
      - 当某一线程被唤醒时，做的事情基本上是检查共享变量的状态是否符合预期(例如队列是否真的有数据)。而诸如if-satisfied-then-do的语句是非线程安全的，因为可能刚检查完条件满足其余线程又改变了状态
      - 需要互斥量的原因
        - 所以，该逻辑语句必须是原子执行的，二者就需要互斥量的协助
        - 函数设计者从逻辑关联性上帮助用户把对共享变量的加锁和解锁做掉，用户拿来直接用就行
        - *条件变量和互斥量有着天然存在的关联关系*
    - 使用while，而不是if
      - 当某一个共享变量不满足需求时，线程会调用pthread_cond_wait()陷入阻塞
      - 当线程被唤醒时，不能保证共享变量一定满足需求
        - 因为可能会有其余线程先被唤醒，而改变了共享变量的状态
      - 所以，对pthread_cond_wait()的调用，一定会是在while循环中，而不是一次性执行的if语句中

```
while (队列长度 == 0) {    // not if
    pthread_cond_wait(&conditon, &mutex);
}
```

- 协同多个线程并行工作: 屏障(barrier)
  - 屏障允许每个线程等待，直到所有的合作线程都达到某一点，然后从该点继续运行
    - pthread_join()就是一种屏障，允许一个线程等待，直到另外一个线程退出
    - 但是屏障的范围更为广阔一些，允许任意数量的线程等待
  - 屏障常常应用于容器的并行化处理，当容器内的对象全部处理完毕后，主线程就并行处理完毕后的容器继续工作
  - Golang中的waitGroup就是一种屏障的应用

线程本地存储: ThreadLocal

- 利用线程ID在同一进程中唯一的特点，使用thread_id作为key，用户所需的数据结构作为value的一种存储方式
- 实现上基本使用哈希表，平衡树或者红黑树等结构也有应用
  - Python Flask Web框架是使用线程局部存储的一个比较典型的例子