计算机系统（2）课程项目

更新：2023 年 2 月 26 日

1 协程

线程和进程是操作系统中非常重要的一组抽象。从概念上说，进程拥有独立的地址空间，创建的若干个进程之间默认没有共享内存（注：这里指没有进行`mmap`等操作）。线程则只有独立的栈而共享所属进程的内存空间。由于进程间通信的开销并不小，所以一些计算任务需要通过多线程的方式完成。然而，线程仍然不够轻量。为此，操作系统领域提出了协程的概念用以实现更轻量级别的计算。我们知道，程序的执行可以看做一个状态机，涉及到一段代码的变量包括寄存器数值、内存地址空间和栈等等。

在本作业中，你需要先掌握`pthread`的用法，并且实现一套简单的协程。当然，在这一部分中你可以通过为协程增加轻量级别的隔离实现来获得额外的分数。

1.1 多线程初步尝试

1.1.1 任务 1: 创建多线程程序

在这个任务中，你需要使用多线程输出 Hello World。任务的期望目标是实践如何创建多线程程序，传递参数给各个线程，并且对多线程的共享变量有所了解。**你只能够按照注释的要求添加代码，而不可以对代码有任何的删减或修改**。

我们在这个任务中的重点不是编写在多核上运行更快的代码。而是编写正确的同步代码，这意味着无论线程的运行顺序如何，代码产生的结果都属于**正确**的结果。

当你完成必要的填空之后，你可以通过在`task1`目录下执行`make`以编译程序，并且在同一个目录下执行`make run`以测试你的结果。

值得注意的是，多线程之间此时并没有任何同步手段，这意味着输出可能是混乱的，但是不用在意。

1.1.2 任务 2: 带有同步原语的多线程

在任务 1 中，我们输出了不少的 Hello World，但是他们之间的顺序是混乱的，甚至每次运行的结果都不尽相同。为了保证多线程对于共享的资源或者变量的合理访问顺序，需要使用一些同步的手段对共享资源的访问进行同步，这样的机制我们称之为同步原语（Synchronization primitives）。关于同步原语的详细知识将会在未来的课程中涉及到。

pthread 库为我们提供了众多同步原语，其中包括互斥锁来保证多个线程不会同时进入到互斥锁所保护的区域之中。互斥锁正如其名字，用于保证程序的互斥访问。具体可以参考以下的例子。

```
thread1(){
  lock(lockA);
  shared_val++;
  printf("abcdabcd");
  unlock(lockA);
}
thread2(){
  lock(lockA);
  printf("efghefgh");
  shared_val = shared_val / 10;
  unlock(lockA);
}
```

在上述给出的例子中，`thread1()`和`thread2()`是两个同时开始运行的线程。`lock(lockA)`和`unlock(lockA)`表示尝试获取锁或者尝试释放锁。如果一个锁已经被获取，那么它在被持有者释放之前无法被其他线程获取。上述的例子中可以保证`printf("abcdabcd");`与`printf("efghefgh");`不会同时执行。

在这个任务之中，你需要使用 pthread 中的互斥锁保证两个函数之中的输出语句不会同时输出造成数据结果混乱。当你完成必要的填空之后，你可以通过在`task2`目录下执行`make`以编译程序，并且在同一个目录下执行`make run`以测试你的结果。此时程序输出可能仍然是随机的，但是不应当出现两句句子混合的情况。

1.1.3 任务 3: 带有 barrier 的多线程程序

聪明的你一定发现了，任务 2 中的代码运行仍然是不确定的，运行两次的结果很有可能是不相同的。不过，他们服从一定的特点，也就是先执行的线程很有可能先完成。现在请你在程序中添加一些语句（提示：`pthread_barrier`），使得各个创建的线程一定从同一个时刻开始运行。你可以通过在`task3`目录下执行`make`以编译程序，并且在同一个目录下执行`make run`以测试你的结果。

1.1.4 任务 4: 带有顺序控制的多线程

上一个任务完成之后，输出顺序仍然是随机的。现在我们希望程序中的多个线程可以按照某个确定的顺序运行，也就是让多线程在满足我们先前定义的顺序的情况下继续执行。在这里我们引出一种同步机制称为条件变量。条件变量允许线程阻塞到某个条件发生时再继续执行。以下给出的代码是一个具体的例子。

```
thread1(){
  lock(lockA);
  shared_val++;
  printf("abcdabcd");
  signal_and_unlock(cond, lockA);
}
thread2(){
  wait_and_lock(cond, lockA);
  printf("efghefgh");
  shared_val = shared_val / 10;
  unlock(lockA);
}
```

在上述给出的例子中，`thread1()`和`thread2()`是两个同时开始运行的线程。`signal_and_unlock(lockA)`和`wait_and_lock(lockA)`表示尝试释放锁并广播条件满足和等待条件满足后尝试获取锁。按照这一定义，`printf("abcdabcd");`与`printf("efghefgh");`不会同时执行，且`printf("abcdabcd");`一定在`printf("efghefgh");`之前被执行。

在这个任务中，你需要使用 pthread 中的线程创建（带参数）结合互斥锁和条件变量保证程序中最后一个创建的线程一定是最早执行的。你可以通过在`task4`目录下执行`make`以编译程序，并且在同一个目录下执行`make run`以测试你的结果。

1.1.5 任务 5: 绑定核心的线程执行

在上述所有任务完成之后，你对线程的基本创建、线程间通信等过程应该有了基本的了解。然而我们此时仍然无法控制某一个线程执行的核心。一个线程启动之后可能会在多个核心之间迁移，为了解决这个问题，你需要通过设置一些特殊的属性使得某一个线程被锁定在一个特定的核心上。（提示：`pthread_setaffinity_np`, `pthread_getaffinity_np`）

在这个任务中，你需要插入必要的代码实现特定核心的绑定。你可以通过在`task5`目录下执行`make`以编译程序，并且在同一个目录下执行`make run`以测试你的结果。运行时，你可以通过执行`htop`观察现象。

1.2 协程设计与实现

在 Linux 中，线程的切换仍然会引进较大的开销。针对一些轻量级的任务，切换进入内核的开销可能比任务本身执行开销还大。这些任务不需要自己的独立地址空间，只需要完成简单的任务即可。你需要完成的是实现一个简单的用户态协程管理库。在开始进行这项作业之前，你需要先对`setjmp`和`longjmp`有所了解。你可以通过`man setjmp`来获取更多的信息。本项作业要求实现至少以下函数：

1. `co_start`：开始一个协程。我们协程的最基本单位是函数。
2. `co_getid`：获得当前执行协程的 ID。为了方便测试，我们约定只有`co_start`才能创建协程且编号从 0 开始。
3. `co_getret`：获得给定 ID 协程的返回值。为了方便测试，函数返回值只有`int`和`void`两种。
4. `co_yield`：主动切换当前协程给另外一个协程，顺序可以由你自己决定。
5. `co_waitall`：等待所有协程执行结束，阻塞方法。
6. `co_wait`：等待特定协程执行结束，阻塞方法。
7. `co_status`：返回某个协程的状态，它应该是不可查询（UNAUTHORIZED）、结束执行（FINISHED）和正在执行（RUNNING）三者之一。不可查询的含义是待查询的协程不是当前协程的子协程，或该 ID 不存在。

你需要在用户态维护一个协程管理库，在其中维护类似于内核`task_struct`的数据结构并至少记录协程之间的关系、协程号和返回值。我们的测试中包括了单线程和多线程。单线程指的是所有协程都归属于一个线程。多线程指的是协程可能归属于不同的线程。在单线程模式下，`co_waitall`，`co_wait`可能不会有任何实际的效果。在多线程模式下，你需要手动维护好`task_struct`的访问冲突，避免出现多线程不安全。整个过程中内核只能感知到线程而无法感知到协程的存在。

注 本作业不可以使用 C++ 实现。

作为 Bonus 项目，协程之间由于所有资源共享，可能产生不安全的情况。因此请实现一种和内核协同的协程资源隔离机制，对协程的资源进行隔离，保证任意两个协程不会访问到对方的内核资源（在我们的测试中指的是文件标识符）。