《进程》章节练习

1. 在Linux中，若在一个多线程进程的某个线程中使用fork创建一个子进程，则在子进程中会存在几个线程？创建出的子进程可能会导致什么问题？Linux为何要如此设计Fork的语义？

几个线程：

子进程中只存在一个线程。其他线程均在子进程中立即停止并消失。

问题：

1. 子进程内存泄漏。因为其他线程在子进程中停止并消失时，并没有执行清理函数和针对线程局部存储数据的析构函数。另外，子进程中的线程可能无法访问（父进程中）由其他线程所创建的线程局部存储变量，因为（子进程）没有任何相应的引用指针。
2. 死锁。如果一个线程在fork()被调用前锁定了某个互斥量，且对某个全局变量的更新也做到了一半，此时fork()被调用，所有数据及状态被拷贝到子进程中，那么子进程中对该互斥量就无法解锁（因为其并非该互斥量的属主），如果再试图锁定该互斥量就会导致死锁。

为何如此设计：

fork的语义：为调用进程创建一个一模一样的新进程。

1. 接口简单，无需任何参数
2. 历史。先产生的`fork()`，20世纪90年代初期才引入线程
3. 性能 。锁实现在用户态最方便， 背后可能只需要一条原子操作指令即可。大多数 CPU 都支持的。fork 只管用户空间的复制，不会涉及其中的对象细节。

2. 对于分配大量内存的应用，为何及时采用了写时拷贝（Copy-on-write，COW）优化后，`fork`的性能仍然比`vfork`要差？请尝试利用`vfork`的思路对`fork`进行优化，从而在不改变`fork`语义的前提下，提升`fork`的性能。

为何fork性能更差：

- 因为`vfork`不会为子进程单独创建地址空间，不需拷贝映射，性能更好。
- 如果在`fork`之后父子进程都有写操作，则会产生大量`page fault`，得不偿失

提升fork性能：

`fork`后，先让父子进程共享统一地址空间，如果发生了写操作，再拷贝相应映射和内存

3. 在像`ChCore`一样的单进程多线程的操作系统中，操作系统通常以线程为粒度进行调度。在一些调度的实现中，在从属于同一个进程的两个不同线程之间进行上下文切换所消耗的时间比在从属于不同进程的两个不同线程之间进行切换耗时更低，试解释其原因。

同进程的线程共用页表，不同进程的线程不共用页表，即`TTBR0_EL1`不同

如果在调度的实现中，检查了线程是否为同一进程，就可避免页表切换这一步骤，比不同进程的线程切换要更快。

4. 在`ChCore`中，内核会在上下文切换的过程中切换属于不同进程的页表。于此同时，`ChCore`内核中同样使用虚拟地址进行内存访问，需要根据页表进行地址翻译。在这一设计下，如何确保上下文切换过程中对页表的切换不会影响到内核的正常执行？

因为所有线程映射的内核部分是相同的，所以在线程切换时，不需要改变`TTBR1_EL1`中的值，不会切换内核页表。所以在上下文切换的过程中，不会影响到内核的正常执行。

5. 对于像协程一样的用户态线程而言，由于所有的协程均从属于同一个内核态线程，因此同一时间仅能同时运行一个协程。在这一情况下，为何协程仍然能够提升应用性能？对于哪类应用，协程能够尽可能高的提升系统性能？

为何能提升性能：

1. 因为协程的调度是由开发人员决定的。开发人员对应用的语义和执行状态更加了解，因此可能做出更优的调度决策
2. 协程的切换是通过用户态线程库完成的，不需要操作系统参与，不会陷入系统调用，所以性能更好（也不用保存上下文）
3. 合作式，不用等锁

适用应用：

适合频繁切换任务，并明确切换语义的应用