死锁代码

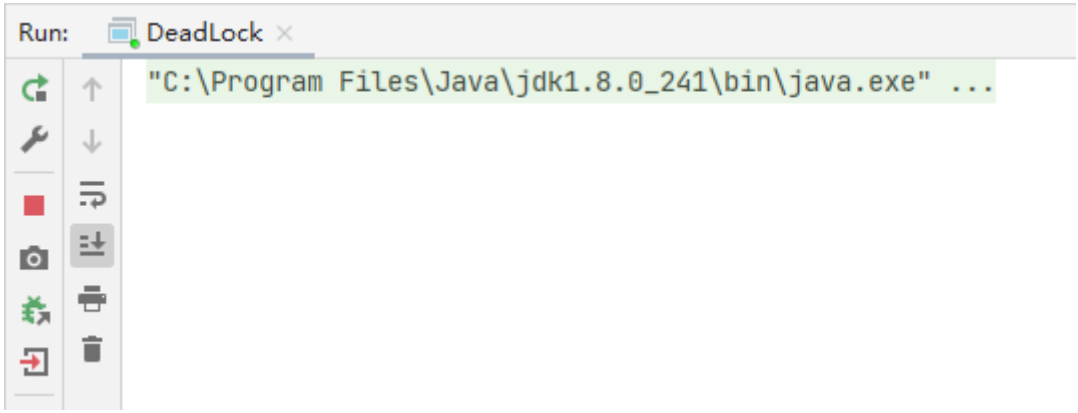
- **互斥、不可剥夺：** ReentrantLock是排他锁
- **请求和保持、环路等待：** `Lock1Lock2` 获得了 `lock1` 并且想获得 `lock2` ，而 `Lock2Lock1` 获得了 `lock2` 并且想获得 `lock1` ，且它们都不会释放已获得的锁

```
public class DeadLock {
    private static final Lock lock1= new ReentrantLock();
    private static final Lock lock2= new ReentrantLock();

    static class Lock1Lock2 implements Runnable{
        @Override
        public void run() {
            lock1.lock();
            /*
            休眠500ms，给Lock2Lock1能锁住的机会
            不然执行速度太快会直接退出
            */
            try {
                Thread.sleep( millis: 500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            lock2.lock();
            System.out.println("Lock1Lock2 break");
            lock2.unlock();
            lock1.unlock();
        }
    }
    static class Lock2Lock1 implements Runnable{
        @Override
        public void run() {
            lock2.lock();
            lock1.lock();
            System.out.println("Lock2Lock1 break");
            lock1.unlock();
            lock2.unlock();
        }
    }

    public static void main(String[] args) {
        new Thread(new Lock1Lock2()).start();
        new Thread(new Lock2Lock1()).start();
    }
}
```

执行时没有输出，说明死锁了



LRU算法实现

纠错：软件实现之前初见的时候说的是优先队列的实现，在此更正一下，用 哈希表+双端链表 的实现更好；硬件计数器实现说的没有问题，下面也给出参考资料

算法题参考文章： https://labuladong.gitbook.io/algo/shu-ju-jie-gou-xi-lie/shou-ba-shou-she-ji-shu-ju-jie-gou/lru-suan-fa

虽然LRU在理论上是可以实现的，但代价很高。为了完全实现LRU，需要在内存中维护一个所有页面的链表，最近最多使用的页面在表头，最近最少使用的页面在表尾。困难的是在每次访问内存时都必须要更新整个链表。在链表中找到一个页面，删除它，然后把它移动到表头是一个非常费时的操作，即使使用硬件实现也一样费时（假设有这样的硬件）。

然而，还是有一些使用特殊硬件实现LRU的方法。首先考虑一个最简单的方法，这个方法要求硬件有一个64位计数器C，它在每条指令执行完后自动加1，每个页表项必须有一个足够容纳这个计数器值的域。在每次访问内存后，将当前的C值保存到被访问页面的页表项中。一旦发生缺页中断，操作系统就检查所有页表项中计数器的值，找到值最小的一个页面，这个页面就是最近最少使用的页面。

管道相关

参考文章1-管道介绍： https://blog.csdn.net/qq_38410730/article/details/81569852

参考文章2-匿名管道和命名管道的区别： https://blog.csdn.net/qq_33951180/article/details/68959819

参考文章3-Linux 的进程间通信：管道 - 腾讯云技术社区的文章 - 知乎： https://zhuanlan.zhihu.com/p/58489873

匿名管道PIPE

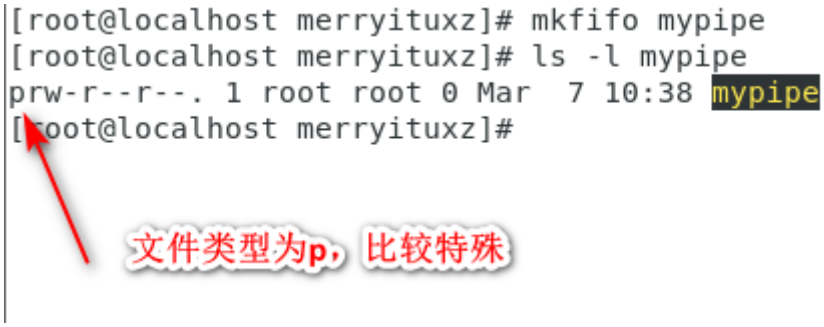
- **典型应用：** Linux shell中的 |
- **匿名性：** 只能在父子进程中使用

  父进程在产生子进程前必须打开一个pipe文件，然后fork出子进程，这样子进程通过拷贝父进程的地址空间就能获得到同一个管道文件的描述符。此时除了父子进程外，没有其他进程知道这个管道文件的描述符（匿名性的体现）

命名管道FIFO

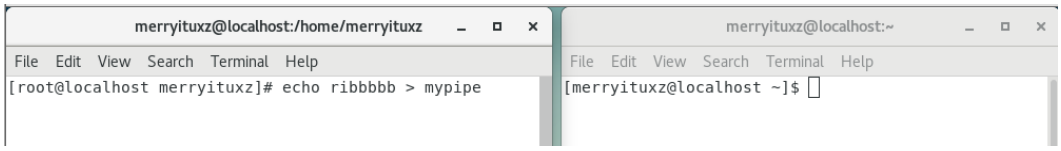
- 命名管道底层与匿名管道完全一样，只不过命名管道会提供一个文件名以供其他进程使用
- 使用 `mkfifo` 命令或 `mknod` 命令创建命名管道

  下面的命令创建了一个名字叫 `mypipe` 的命名管道

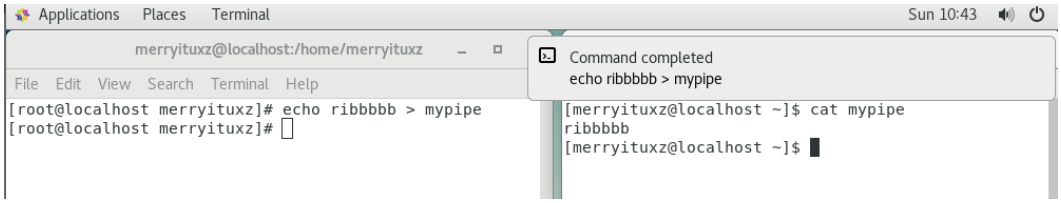


- 实操

1. 左边的 `进程1` 向 `mypipe` 写入内容，此时进程1的写操作被阻塞，因为另一端没有进程读取这个管道



2. 右边的 `进程2` 读取 `mypipe`，阻塞解除，`进程1` 的echo命令也成功返回



cache组索引位置

参考csapp 432-433页旁注

争议题2012-408真题-选择题17的争议，原因就是可能有这两种映射方式，但实际上基本都是中间位索引

旁注　为什么用中间的位来做索引

你也许会奇怪，为什么高速缓存用中间的位来作为组索引，而不是用高位。为什么用中间的位更好，是有很好的原因的。图 6-31 说明了原因。如果高位用做索引，那么一些连续的内存块就会映射到相同的高速缓存块。例如，在图中，头四个块映射到第一个高速缓存组，第二个四个块映射到第二个组，依此类推。如果一个程序有良好的空间局部性，顺序扫描一个数组的元素，那么在任何时刻，高速缓存都只保存着一个块大小

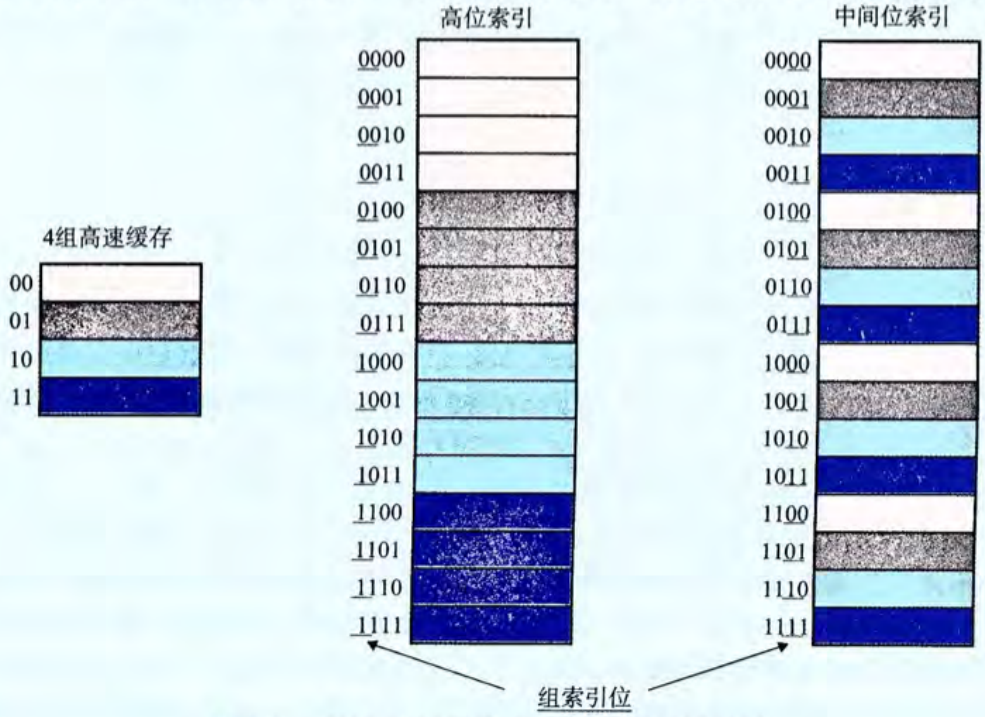


图 6-31　为什么用中间位来作为高速缓存的索引

的数组内容。这样对高速缓存的使用效率很低。相比较而言，以中间位作为索引，相邻的块总是映射到不同的高速缓存行。在这里的情况中，高速缓存能够存放整个大小为 C 的数组片，这里 C 是高速缓存的大小。