 加锁和解锁的基本思想是，当某个进程进入临界区，它将持有一个某种类型的锁（UNIX里一般来说是semaphore，Linux里一般是信号量和原子量或者spinlock）。当其他进程在该进程没有释放该锁时试图进入临界区（加锁），它将会被设置成睡眠状态，然后被置入等待该锁的进程队列（某个优先级的）。当该锁被释放时，也就是解锁事件发生时，内核将从等待该锁的进程优先级队列中寻找一个进程并将其置为就绪态，等待调度（schedule）。在system v中，等待某一事件被称为sleep（sleep on an event），因此下文将统一使用睡眠（sleep）。等待某事件也可以成为等待某个锁。（注：本文中的sleep与sleep()系统调用不同）系统的实现将一组事件映射到一组内核虚拟地址（锁）；而且事件不区别对待到底有多少进程在等待。这就意味着两个不规则的事情： 一、当某个事件发生时，等待该事件的一组进程均被唤醒（而不是仅仅唤醒一个进程），并且状态均被设置成就绪（ready-to-run）。这时候由内核选择（schedule）一个进程来执行，由于system v内核不是可抢占的（Linux内核可抢占），因此其他的进程将一直在就绪状态等待调度，或者再次进入睡眠（因为该锁有可能被执行进程持有，而执行进程因为等待其他事件的发生而睡眠），或者等其他进程在用户态被抢占。 二、多个事件映射到同一个地址（锁）。假设事件e1和e2都映射到同一个地址（锁）addr，有一组进程在等待e1，一组进程在等待e2，它们等待的事件不同，但是对

应的锁相同。假如e2发生了，所有等待e2的进程都被唤醒进入就绪状态，而由于e1没有发生，锁addr没有被释放，所有被唤醒的进程又回到睡眠状态。貌似一个事件对应一个地址会提高效率，但实际上由于system v是非抢占式内核，而且这种多对一映射非常少，再加上运行态进程很快就会释放资源（在其他进程被调度之前），因此这种映射不会导致性能的显著降低。 下面简单阐述一下sleep和wakeup的算法。 //伪代码 sleep(地址（事件），优先级) 返回值：进程能捕获的信号发生导致的返回则返回1，当进程不能捕获的信号发生时返回longjmp算法，否则返回0。 { 提高处理器执行等级以禁用所有中断； //避免竞态条件 将进程的状态设置为睡眠； 根据事件将进程放入睡眠哈希队列； //一般来说每个事件都有一个等待队列 将睡眠地址（事件）及输入的优先级保存到进程表中； if (该等待是不可中断的等待) //一般有两种睡眠状态：可中断的和不可中断的。不可中断的睡眠是指进程除了等待的事件外， //不会被其他任何事件（如信号）中断睡眠状态，该情况不太常用。 { 上下文切换； //此处该进程执行上下文被保存起来，内核转而执行其他进程 //在别处进行了上下文切换，内核选择该上下文进行执行，此时该进程被唤醒 恢复处理器等级来允许中断； 返回0； } // 被信号中断的睡眠 if (没有未递送的信号) { 上下文切换； if (没有未递送的信号) { 恢复处理器等级来允许中断； 返回0； } } //有未递送的信号 若进程还在等待哈希队列中，将其从该队列移出； 恢复处理器等级来允许中断； if(进程捕获该信号) 返回1； 执行longjmp算法 ； //这一段我也不明白 } 100Test 下载频道开通，各类考试题目直接下载。详细请访问 www.100test.com