操作系统第五章习题

5.4

自旋锁不适用于单处理器系统，因为只有通过执行不同的进程才能获得将进程从自旋锁中断的条件。如果进程没有放弃处理器，其他进程就没有机会设置第一个进程取得进展所需的程序条件。在多处理器系统中，其他进程在其他处理器上执行，从而修改程序状态，以便从自旋锁释放第一个进程。

5.5

wait()操作以原子的方式递减与信号量相关的值。如果一个信号量的值为1时，在该信号量上执行了两个等待操作，如果这两个操作不是原子执行的，那么这两个操作可能会继续减小信号量的值，从而违反互斥。

5.10

如果用户级程序具有禁用中断的能力，那么它可以禁用计时器中断并防止发生上下文切换，从而不让其他进程执行，达到独占处理器的目的。

5.11

如果多个进程运行在不同的CPU上，每个进程都试图进入一个临界区，即使禁止中断，其他进程依旧能够在其他处理器上进入临界区。

5.15

|  |
| --- |
| typedef struct { int available; } lock;    void init(lock mutex) {  *// available=0 -> lock is available, available=1 -> lock is unavailable*  mutex->available = 0;  }    void acquire(lock mutex) {  while (testandset(&(mutex->available))) *// TEST the available*  ; *// spin-wait (do nothing)*  }    void release(lock mutex) {  mutex->available = 0;  } |

5.17

(1) 如果锁要保持很短的时间，使用自旋锁更有意义，因为事实上它可能比使用互斥锁更快，因为互斥锁需要暂停-和唤醒-等待的进程。

(2) 如果要保持较长的时间，最好使用互斥锁，因为这允许处理核心在被锁进程等待时安排另一个进程。

(3) 如果线程在持有锁的时候可能会进入睡眠状态，那么互斥锁肯定是最好的，因为我们不希望等待的进程在等待另一个进程醒来的时候还在旋转。

5.18

就时间t而言，可以容忍的最大时间将是<2xT 因此，超过最大时间，将使用互斥锁而不是自旋锁，处于等待模式的线程被切换到睡眠模式。

5.20

(a)number\_of\_processes可能产生竞争

(b) 在进入每个函数时必须调用acquisition()，在退出每个函数前必须立即调用release()。

(c)不能解决。原因是竞赛发生在allocate process()函数中，其中进程数首先在if语句中被测试，但之后又根据测试值被更新。有可能在测试时进程数=254，但由于竞赛条件，在再次递增前被另一个线程设置为255。

5.23

|  |
| --- |
| typedef struct semaphore {  int value;  int lock;  } semaphore;    void wait(semaphore \*s) {  while (test\_and\_set(&s->lock) == 1);  s->value--;  if (s->value < 0) {  test\_and\_set(&s->lock);  } else {  test\_and\_set(&s->lock);  }  }    void signal(semaphore \*s) {  while (test\_and\_set(&s->lock) == 1);  s->value++;  if (s->value <= 0) {  }  test\_and\_set(&s->lock);  } |

5.28

相对于允许单个写作者专门访问共享值而言，偏向于多个读者，可以提高读者-写作者问题的吞吐量。另一方面，偏爱读者可能会导致写手的饥饿。当一个读者来到这里并注意到另一个读者正在访问 数据库，那么只有当没有等待的写手时，它才会进入关键部分。

读者/写作者问题中的饥饿现象可以通过保留与等待进程相关的时间戳来避免。当一个写手完成其任务时，它将唤醒等待时间最长的进程。