第七章课后练习

7.8 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

当在尝试获取一个信号量的时候，进程会被阻塞挂起，放进等待队列；而当进程拥有一个spinlock的时候，是应当处于忙等待的状态，才可以不断判断spinlock的条件是否成立，获取进入临界区的机会。假如进程在尝试获取信号量时也拥有了一个spinlock，则有可能进程在等待信号量，不断被挂起放入等待队列的过程中，无法获取进入spinlock后的临界区的机会，造成饥饿现象。

7.11

fairness and throughput：在读者-写者问题的解决方案中，读者进程和写者进程的优先级是一样的，而且也没有抢占，读者进程可以并发地进行读操作，但是写者进程必须等所有的读者进程完成读操作后才可以进行写操作，且同时只能有一个写进程进行写操作。当同时有多个读进程进行读操作时，吞吐率可以得到提升。

starvation的产生：当始终不断有读进程在进行读操作时，会造成写进程的饥饿现象

method：①每次都是挑选等待队列中等待时间最长的进程进入临界区，这可以保证读进程不会发生饥饿；②当已有读进程在读取数据时，一个新的读进程想要进入临界区，需要先检查等待队列，若有写进程在等待，则读进程暂不进入，这可以保证写进程不会发生饥饿

7.16

在每个函数的开头加锁，返回前将锁返回，即每个函数的函数体都放在临界区内，这样既可保证在每个操作进行的过程中，其中的变量不会被别的进程所改变

修改后如下：

```c
/* * Stack containing race conditions */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Linked list node
typedef int value_t;

typedef struct Node {
    value_t data;
    struct Node *next;
} StackNode;

// Stack function declarations
void push(value_t v, StackNode **top);
value_t pop(StackNode **top);
int is_empty(StackNode *top);
```

```
18
19  pthread_mutex_t mutex;
20
21  int main(void) {
22      pthread_mutex_init(&mutex,NULL);
23
24      StackNode *top = NULL;
25      push(5, &top);
26      push(10, &top);
27      pop(&top);
28      push(15, &top);
29      pop(&top);
30      pop(&top);
31      push(20, &top);
32      push(-5, &top);
33      pop(&top);
34      push(-10, &top);
35      pop(&top);
36      pop(&top);
37      push(-15, &top);
38      pop(&top);
39      push(-20, &top);
40
41      return 0;
42  }
43
44  // Stack function definitions
45  void push(value_t v, StackNode **top) {
46      pthread_mutex_lock(&mutex);
47
48      StackNode *new_node = malloc(sizeof(StackNode));
49      new_node->data = v;
50      new_node->next = *top;
51      *top = new_node;
52
53      pthread_mutex_unlock(&mutex);
54  }
55
56  value_t pop(StackNode **top) {
57      pthread_mutex_lock(&mutex);
58
59      if (is_empty(*top)) return (value_t) 0;
60
61      value_t data = (*top)->data;
62      StackNode *temp = *top;
63      *top = (*top)->next;
64      free(temp);
65      return data;
66
67      pthread_mutex_unlock(&mutex);
68  }
69
70  int is_empty(StackNode *top) {
71      pthread_mutex_lock(&mutex);
72
73      if (top == NULL) return 1;
74      else return 0;
75
```

```
76      pthread_mutex_unlock(&mutex);
77  }
```