清华大学本科生考试试题纸	
考试课程：操作系统（A卷）	时间：2021年12月06日下午03:20~05:20

系别：_____	班级：_____	学号：_____	姓名：_____

答卷注意事项:	1.答题前在试题纸和答卷本上写明A卷或B卷、系别、班级、学号和姓名。
	2.在答卷本上答题时, 要写明题号, 不必抄题。
	3.答题时, 要书写清楚和整洁。
	4.请注意回答所有试题。本试卷有22个小题，共7页。
	5.考试完毕, 必须将试题纸和答卷本一起交回。

一、填空题（15分）

1. 虚拟机监控程序（VMM, Hypervisor）在CPU虚拟化时，需要使用模拟方式执行_____。
2. 并行（Parallel）、并发（Concurrent）和异步（Asynchronous）是三个相关的技术，可以应用于单核CPU上的技术是_____。
3. 软链接和硬链接都是文件别名的实现方法，可以跨不同文件系统使用的是_____。
4. FAT、EXT3、EXT4、NTFS和ZFS是几种常用的文件系统，其中_____是支持掉电保护的文件系统。
5. 分层结构、微内核（microkernel）、外核（Exokernel）是几种常见的操作系统结构，其中_____把减少内核功能作为设计目标。
6. 中断、异常和系统调用与操作系统中的意外事件处理相关，其中_____是与CPU的当前执行指令流相关的。
7. uCore和rCore是我们教学操作系统，其中_____的内核和应用程序是两个不同的进[redacted]进程
8. SLAB 、SLOB和SLUB三种内存对象分配器，其中_____分配器是针对嵌入式系统的_____简化版本。
9. 在RISC-V CPU上执行指令是会出现多种异常，其中_____异常只会在MMU启用后
10. 在RISC-V Sv32把页表项分成第一级页表项（L1PTE）和第二级页表项（L2PTE），其中_____中保存的物理页号不能是指向自己所在页的物理页号。
11. 屏蔽中断和原子指令都可以实现临界区的访问控制。_____无法在多核场景中用于临界区的访问控制。
12. 进程、线程和协程是三个与处理机调度相关的概念，其中处理机执行流切换性能最好的是_____，执行环境隔离功能由_____来实现。
13. 优先级继承（Priority Inheritance）和优先级天花板协议（priority ceiling protocol）是用于解决“优先级反置(Priority Inversion)”的方法，其中_____只在占有资源的低优先级进程被阻塞

时,才提高占有资源进程的优先级。

14. 公平调度算法（CFS, Completely Fair Scheduling）和BFS(Brain Fuck Scheduler)调度算法是两种多处理机调度算法，其中_____是不需要负载均衡机制的。
15. 平均无故障时间(Mean Time To failures,MTTF)、平均修复时间 (Mean Time To Repair, MTTR)和平均失效间隔 (Mean Time Between Failure，MTBF)是几种常用的系统可靠性度量指标。其中表示容灾能力的指标是_____。

二、简答题

16. （24分）

1）试描述自旋锁的基本工作原理；

2）在 x86/arm 等 ISA 中包含了 `compare_and_exchange` 这条原子指令，简化定义如下：

```
cmpxchg rd, rs1, rs2, (rs3)  # 若 rs2 == *rs3, 则 rd = 1,  *rs3 = rs1; 否则 rd = 0, *rs3 不变
```

所有其他的原子操作(如 `test_and_set` `atomic_add` 等都可以基于 `cmpxchg` 实现)。

RISC-V A 拓展规定了一些内存原子指令：

```
lr rd, (rs1)  # rd = *rs1, 同时留下记录
sc rd, rs1, (rs2)   # 若上一条LR到这条SC之间没有其他核写 *rs2, 则: rd = 1, *rs2 = rs1; 否则 rd = 0, *rs2 不变
```

备注：`LR` `SC` 对有无其他核访问的记录是通过硬件上的核间一致性协议保证的，可以不关注这些细节。

RISC-V A 拓展中并没有直接包含 `compare_and_exchange` 原子指令。事实上，RISCV 的 `cmpxchg` 是通过 `lr` `sc` 来实现的。

试使用 `LR` `SC` 实现 `cmpxchg`。为了方便，使用 C 风格伪代码即可。

- C 风格 `LR` `SC` `CMPXCHG` 定义如下：

```
int LR(int* addr);
bool SC(int value, int* addr);
bool CMPXCHG(int val, int test_val, int* addr) {
    // YOUR WORK
}
```

3）试使用 `LR` `SC` 实现一个自旋锁的锁定和解锁操作。为了方便，使用 C 风格伪代码即可。

- C 风格 `lock` `unlock`定义如下，不考虑出错的情况：

```
typedef int lock_t;
// 假定 lock = 1 为上锁状态，lock = 0 为解锁状态。
void spin_lock_init(lock_t* lock) {
    *lock = 0;
}
void spin_lock_acquire(lock_t* lock) {
    // YOUR WORK
}
void spin_lock_release(lock_t* lock) {
    // YOUR WORK
}
```

4）试简单解释你的自旋锁实现是否能保证多线程按申请顺序获得自旋锁。

5）假定CPU硬件支持如下原子指令：

```
int lr(int* addr); // 同上
int sc(int val, int* addr);  // 同上
int cmp_and_exchange(int val, int test_val, int* addr);  // 同上
int test_and_set(int* addr); //若 *addr == 0 则 返回 1, *addr = 1; 否则: 返回 0, *addr 不变
int test_and_clear(int* addr); //若 *addr == 1 则 返回 1, *addr = 0; 否则: 返回0, *addr 不变
int load_and_add(int* addr); // *addr += 1, 返回原始值
int add_and_load(int* addr); // *addr += 1, 返回 +1 后的值
```

针对spinlock无法保证申请顺序的问题，令牌锁（ticket lock）是一种解决方法。它的做法是，线程在竞争某个锁时，先领一张令牌（ticket）（提示：可使用原子操作来保证每个线程领到令牌序号是递增发放的），然后监听当前允许进入临界区的令牌编号（head），发现head被更新为自己的tail时，进入临界区。

假定令牌锁数据结构为

```
typedef struct _ticketlock {
    uint32_t head;
    uint32_t tail;
} ticket_t;
```

考虑一个多核的机器，多个尝试获取锁的线程将tail的值保存在自己的一个寄存器中，同时将tail + 1。开锁的时候将head + 1，然后释放。真正获取锁是将head和自己保存的tail值进行比较，相等时获取。

试基于上面提供的原子指令，通过补全下面的代码，实现令牌锁机制。

```
ticketlock_t ticket;
ticket_lock_init(ticketlock *ticket) {
    ticket->head = ticket->tail = 0;
}

ticket_lock_acquire(ticketlock *ticket) {
    /* YOUR WORK*/;
}

ticket_lock_release(ticketlock *ticket) {
    /* YOUR WORK*/;
}
```

6）试简单解释为何你的令牌锁实现是否能保证多线程按申请顺序获得令牌锁。要求分别就 (a) 没有锁竞争和 (b) 有3个核锁竞争同一个锁两种情况描述令牌锁变量的变化情况。

17. （10分）

1）简述死锁发生的四个条件。

2）请用较为规范的语言证明对于资源A_1, A_2, ..., A_n 如果进程P_1, P_2, ..., P_m都按资源的下标顺序对自己需要的资源进行加锁，则不会发生死锁。

3）第二小题的方式破坏了死锁发生的哪个条件。

18. （12分）

1）张同学想实现一个支持并发操作的栈，他的实现代码如下：

```
elem_t *head; // top of the stack

void push(int val) {
    elem_t *e = malloc(sizeof(*e));
    e->next = head;
    e->val = val;
    head = e; // put e on the top
}
```

```
elem_t* pop() {
    elem_t *e = head;
    if (e) head = e->next;
    return e;
}

elem_t *search(int key){
  for (elem_t *e = head; e; e = e->next) {
    if (e->val == key)
      return e;
  }
  return NULL;
}
```

请给出一种两个线程并发调用 `search`, `push` 或 `pop`，导致数据竞争，栈的状态与正确执行的结果不一致的例子。

2）张同学意识到使用读写锁可以正确支持并发，现在他的代码如下：

```
struct rwlock rwlock;

void locked_push(int key, int value) {
    acquire_write(&rwlock);
    push(key, value);
    release_write(&rwlock);
}

elem_t *locked_search(int key) {
    acquire_read(&rwlock);
    elem_t *e = search(key);
    release_read(&rwlock);
    return e;
}


// 读写锁的实现，你不需要完全理解实现细节
struct rwlock {
    spinlock_t l;
    volatile unsigned nreader;
};

void acquire_read(struct rwlock *rl) {
    spin_lock(&rl->l);
```

```
    atomic_increment(&rl->nreader);
    spin_unlock(&rl->l);
}

void acquire_write(struct rwlock *rl) {
    spin_lock(&rl->l);
    while (rl->nreader) /* spin */ ;
}
```

并发地调用 `locked_search` 在栈中进行搜索时，程序的性能明显好于使用自旋锁实现的版本，为什么?

3）另一方面，即使完全不修改栈， `locked_search` 也明显慢于 `search`，为什么?

19. （8分）

操作系统种有四种资源A,B,C,D，它们一共有(8,8,6,8)个。现在有4个进程，它们的需要的总资源数目如下：

进程名	A	B	C	D
K	6	3	2	3
L	1	4	1	2
M	3	4	3	1
N	2	2	3	4

它们目前还需要的资源如下：

进程名	A	B	C	D
K	1	2	0	1
L	1	3	1	0
M	1	3	2	1
N	2	0	3	3

假设只有这4个进程以及这些资源。请分别分析下述几种请求之后是否会死锁（各个请求独立）。

1）K请求(0,2,0,0)
2）L请求(1,1,0,0)
3）M请求(0,0,2,1)
4）N请求(1,0,1,0)

20. （12分）

Unix文件系统（UFS）的文件组织方式很巧妙地把多种索引方式组合在一起，从保证小文件的存储效率，并且可支持很大的文件。某UFS采用如下文件组织方式。

小于等于

1. 如果存放文件所需的数据块小于`m`块，则采用直接索引方式；
2. 如果存放文件所需的数据块超过`m`块，则采用一级间接索引方式；
3. 如果前面两种方式都不够存放大文件，则采用二级间接索引方式；
4. 如果二级间接索引也不够存放大文件，这采用三级间接索引方式

数据块大小为`n`字节，一个数据块中最多可存储`k`个索引。

试计算以下各情形下UFS支持的最大文件长度。

1）没有一级间接索引方式

2）没有二级间接索引方式

3）没有三级间接索引方式

4）该文件系统支持的最大文件的长度

21. （10分）

1）为了解决一致性问题，提出了fsck以及日志这些方法，但在ext3/4文件系统采用了日志。为什么我们不想主要使用fsck来解决一致性问题?

2）ext3采用了一种数据日志的方式来记录日志。之后有人提出了一种元数据日志。（1）分别简要描述数据日志文件系统和元数据日志文件系统的文件写入顺序。（2）在其他条件相同时，数据日志文件系统和元数据日志文件系统哪个写操作性能高？为什么?

22. （9分）

假设一个磁盘有 200 个磁道，编号为 0~199，当前磁头位置为 149，磁头运动方向为由小到大。系统当前磁盘请求序列为：88、147、95、177、94、150、102、175。对FIFO算法、最短服务时间优先 (SSTF) 算法、扫描算法 (SCAN) 这三种磁盘调度算法而言，试分别给出每一种算法下的磁头访问序列，并计算总的磁道移动距离。