CSL373/CSL633 Minor 2 Exam
Operating Systems
Sem II, 2012-13

Answer all 6 questions   Max. Marks: 30

1. **True or False**. Give short reasons. No marks for incomplete/wrong reasons. [10] (2 each)

a. Pages that are shared between two or more processes can never be swapped out to disk

False. Pages shared by two processes can be swapped out. Multiple page tables need to be updated while swapping out shared pages.

Full marks if have rough idea. No marks if no reason given

b. CLOCK algorithm (1-bit approximation to LRU) always has a poor hit ratio, when compared with LRU

No. Because all algorithms approximate past with future, It is always possible to construct a workload where any one algorithm is better than another.

Full marks if have rough idea. No marks if no reason given.

c. With hardware-managed TLBs (as on x86), every write to a virtual address results in a write to the page table entry corresponding to that virtual address (to set the dirty and accessed bits). In other words, every write to a virtual address results in two physical memory writes.

No. Hardware writes to the page table only when the values of these bits change. It obtains the old values of these bits from TLB.

Full marks if mention that writes need to happen only at value change or if mention that TLB absorbs repeat writes. No marks otherwise.

d. Pintos does not need to use spinlocks.

True. Being a uniprocessor system, it can implement critical sessions by disabling interrupts. Only blocking locks make sense because they yield CPU to other processes.

Full marks if specify that pintos is uniprocessor and spinlocks do not make sense on uniprocessor. No marks otherwise.

e. A blocking lock is always more efficient than a spinlock, as it relinquishes CPU.

No. The cost of blocking (context switch, save/restore state, add thread to queue) can be higher than the cost of spinning (if the critical sections are small).

Full marks if mention the cost of blocking, or small critical sections. No marks otherwise.

2. Implement an infinite queue with potentially multiple producer and multiple consumer threads. Use locks and sleep/wakeup for synchronization, if needed.   [2]

```
struct q {   int arr[];  int head; int tail; }
spinlock l;

initialize(): q.head = 0; q.tail = 0;

produce(elem):
acquire(l);
arr[arr.head] = elem;
arr.head++;
wakeup(q);
release(l);

consume(elem):
acquire(l);
while (arr.head == arr.tail) sleep(q, l);
retval = arr[arr.tail];
arr.tail++;
release(l);
```

No marks deducted for not specifying initialize()

0.5 marks deducted if make a silly error like forget to update tail or head properly.

Deduct one mark if:
1. forgot a call to wakeup()
2. Don't use the fact that it is an infinite queue (e.g., sleep in producer)

Zero if
1. Don't use a spinlock and use xchg, etc.

3. Consider the statement 'release(&ptable.lock)' at line 2487 in forkret() function. The comment before the line says that this releases the ptable.lock from scheduler. Where is the corresponding call to bracketing acquire(ptable.lock)? Is it true that the parent acquires the lock and the child releases it in forkret()? List the sequence of lock/unlock events (for ptable.lock) by

parent/child on a fork. (Also say in which function these events occurred). [6]

The corresponding call to acquire(ptable.lock) is at line 2417 (scheduler() in proc.c). Another possibility is for the bracketing call to be at line 2474 (yield() in proc.c). In this case, the scheduler switches to the child process without releasing the lock.

The lock is either acquired by the process that got context-switched out just before the child got to run (at line 2474) or by the scheduler thread itself (at line 2417). In the first case, it is possible that the previous process is the parent. It is also possible that this previous process is not the parent. Hence, this statement is not true in general, but true in some possible executions.
Here is one possible execution

1. parent acquires in allocproc()
2. parent releases in allocproc()
3. parent acquires in yield() and context switches to scheduler.
4. scheduler() releases lock and reacquires it before context switching to the child
5. child releases the lock in forkret

If you omit step 4, that is also a correct answer.

4. Consider the kernel stack (kstack) of a switched-out process (a process in RUNNABLE state but not currently running) in xv6. What is the minimum number of code pointers on the kernel stack (i.e., they point to an executable instruction)? List them. How many of these are user pointers (i.e., pointing into user address space)? [6]

There are at least two structures on kstack of a RUNNABLE process, namely “context” and “trapframe”. context.eip is a kernel eip. trapframe.eip is a user eip. So a minimum of two code pointers.

Plus, the kernel trapframe needs to be read by a function. The only function to read the trapframe is “trapret”. Trapret cannot be the context.eip pointer because, the only way to get to trapret is for a new process (where trapret is written to stack explicitly by allocproc) or by falling through from alltraps. Hence, there will always be one more code pointer pointing to trapret on kstack.

For a newly created process, there are three code pointers (trapframe.eip, context.eip, and trapret). Here, context.eip is set to forkret (see allocproc). Hence, the answer is 3. Of these, 1 is a user code pointer.

deduct 3 marks if wrong answer with good explanation
deduct 4 marks if wrong answer with poor explanation

deduct 1 mark for incorrect number of user pointers
deduct 1 mark if answer is correct and good rough idea but bad explanation

Zero if no idea.

5. There are two ways to access files on Pintos:
   a. open/read/write/close system calls
   b. mmap/munmap system calls

   List at least one advantage of each interface (read/write and mmap/munmap) of accessing files. [3]

Advantages of open/read/write/code:
1. No need to allocate address space
2. Good for small IOs, as mmap/munmap require at least page granularity IOs.
3. Potentially better sequential IO performance

Advantages of mmap/munmap:
1. Better performance and programmability for random IO
2. Allows a good interface for mapping dynamically linked libraries
3. Allows different mappings of the same address space (e.g., to local memory, to remote memory, to disk).

6. Implement a blocking lock using sleep and wakeup calls (as on xv6). In other words, implement ‘blocking_acquire()’ and ‘blocking_release()’. You are free to use other synchronization primitives like spinlocks. [3]

```
struct b_lock {
  int locked;
  struct spinlock l;
 } bl;

b_initialize(bl):  bl.locked = 0;

b_acquire(bl):
   acquire(l);
   while (bl.locked) sleep(bl, l);
   bl.locked = 1;
```

```
    release(l);

b_release(bl):
    acquire(l);
    bl.locked = 0;
    release(l);
```

Full marks if get it right. Deduct 1 mark for careless mistake. Zero marks if use xchg or any other primitives.