Optional refresher module assignment

- send(): wait until the entire buffer is sent
- receive(): wait until some bytes are available
- send() and receive() recycle the shared memory (e.g., can implement circular buffer)
- The shared memory can be accessed only through the send and receive APIs

Group assignments

- All the group members are expected to understand the complete assignment

- This will also help you in preparing for the final examination

Difference between log_write and bwrite?

Difference between log_write and bwrite?

- bwrite ensures the atomic write of single disk sector
- log_write ensures the atomic write of a sequence of disk blocks
  - i.e., the atomicity of a complete operation

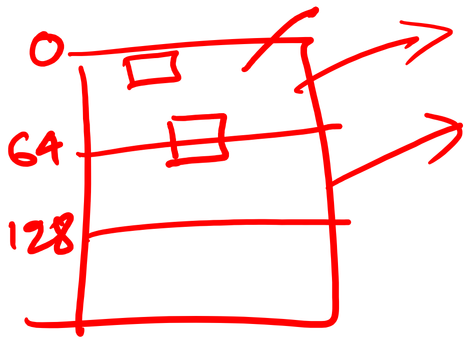
Does end_op immediately commit?

Can we commit just once, before the program termination?

Cache lines

- VA -> PA
- [4096 – 8192] -> [0 - 4096]

- Which cache line would be brought in the cache when the application access a virtual address (4096 + 8)



Does x86 reorder loads and stores?

Does x86 reorder loads and stores?

- Yes, on x86 a load can be reordered with a previous store on a different memory location

P1	P2
x = 1	y = 1
r1 = y	r2 = x

Initially, x == 0 y == 0

(r1 == 0 && r2 == 0)

Why does x86 reorder loads and stores?

Efficiency

Do we need mfence in a single threaded application?

Peterson's solution

```
volatile int turn;
volatile boolean flag[2];
acquire:                          → flag[j]
flag[i] = TRUE;
turn = j;
mfence →
while (flag[j] && turn == j);
release:
flag[i] = FALSE;
```

lock prefix

- Some instructions can additionally take a lock prefix to tell the hardware to execute the instruction atomically

lock add $1, 0x1000

Because of the lock prefix, the hardware will execute this instruction atomically

What does lock prefix do?

- drains the store buffer
- lock the cache-lines used by the instruction before executing the instruction
- loads/stores cannot be reordered across lock instruction

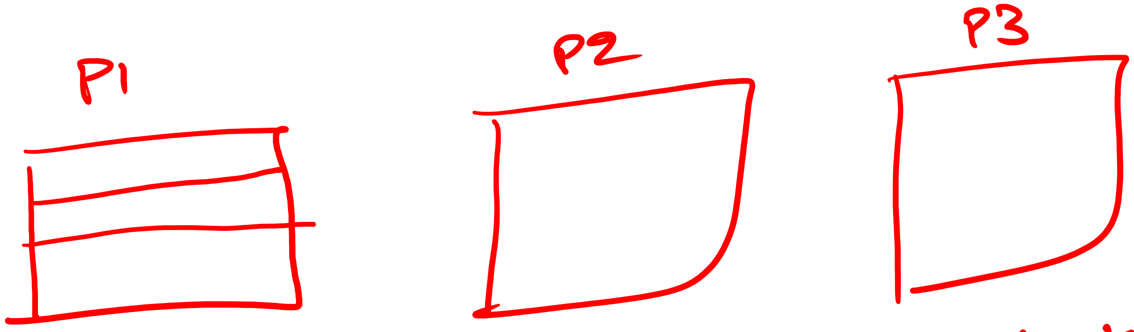
Disadvantage of using lock prefix

- Very slow
  - implicit memory barrier
  - lock the cache-lines (slow the execution on other cores with conflicting memory access)

- Aggressively invalidate cache lines of other CPUs if all CPUs try to modify the same cache line
  - also called cache line bouncing
  - very slow if you have large number of cores (>=32)

Cache line bouncing

```
label1:
lock add $1, lockvar
cmp $1, lockvar
je label2
lock sub $1, lockvar
jmp label1
label2:
```



On every update : invalidate cache line

On every read : fetch the cache line in shared mode

Spin lock

```
struct spinlock {
   volatile unsigned int locked;
};

void acquire (struct spinlock *lk) {
    while (lk->locked != 0);
    lk->locked = 1;
}

void release (struct spinlock *lk) {
   lk->locked = 0;
}
```

Spin lock

```
struct spinlock {
   volatile unsigned int locked;
};

void acquire (struct spinlock *lk) {
    pushcli();
    while (atomic_xchg(&lk->locked, 1) != 0);
    __sync_synchronize ();
}
```

mov $1 , %eax

lock xchg %eax , $lk->locked

Spin lock

```
struct spinlock {
    volatile unsigned int locked;
};

void release (struct spinlock *lk) {
    __sync_synchronize ();
    lk->locked = 0;
    popcli();
}
```

Spin lock

- pushcli and popcli ensures that this lock primitive can be used in an interrupt handler

- It is unsafe to use a spin lock, that does not disable interrupt, in an interrupt handler

Spin lock

- Can we improve this spin lock implementation?

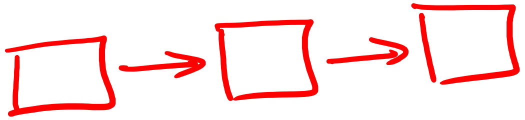
```
void acquire (struct spinlock *lk) {
    pushcli();
    while (atomic_xchg(&lk->locked, 1) != 0); { while (lk->locked != 0); }
    __sync_synchronize ();
}
```

Spin lock

- What is the problem with this spin lock?
  - not fair, anybody can acquire the lock irrespective of their arrival

```
void acquire (struct spinlock *lk) {
    pushcli();
    while (atomic_xchg(&lk->locked, 1) != 0) {
        while (lk->locked == 1);
    }
    __sync_synchronize ();
}
```

ticket spin lock



- atomic_xadd: atomically adds the input value to the input memory location and return the old value of the memory location

ticket spin lock

```
struct lock {
    volatile unsigned head;   // initially 0
    volatile unsigned tail;      // initially 0
};
```

acquire:

```
oldtail = atomic_xadd (&lockvar->tail, 1);
while (oldtail != lockvar->head);
```

release:

```
lockvar->head++;
```

head==1

head==0
tail==0

Thread 1
tail==1
oldtail == 0

Thread 2
tail=2
oldtail=1
while(oldtail != head

Thread 3
tail=3
oldtail =2
while (oldtail != head);

Readers-writer locks

- Multiple readers can concurrently execute in the critical section
- Only a single writer is allowed in the critical section

Readers-writer locks

```
volatile unsigned lockvar = 10000;
```

read_acquire:

```
while (atomic_sub(&lockvar, 1) < 0) {
    atomic_add (&lockvar, 1);
    while (lockvar <= 0);
}
```

read_release:

```
atomic_add (&lockvar, 1);
```

write_acquire:

```
while (atomic_sub (&lockvar, 10000) != 0) {
    atomic_add (&lockvar, 10000);
    while (lockvar != 10000);
}
```

write ~~read~~_release:

```
atomic_add (&lockvar, 1); 0000) ;
```

Readers-writer lock

- atomic_sub: atomically subtracts the input value from the input memory location and return the updated value

- atomic_add: atomically adds the input value to the input memory location and return the updated value

Problem with reader-writer lock

- Starvation

- Both readers and writer execute atomic instruction on a shared lock