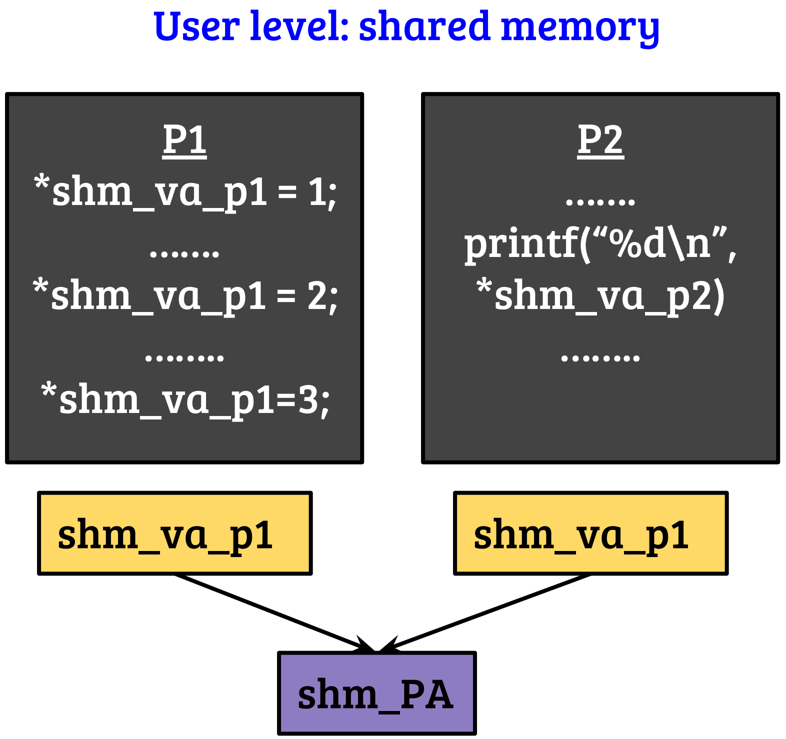
Operating Systems

Synchronization

Debadatta Mishra, CSE, IITK

Why synchronize?



- Shared memory mapped to two different processes, can be two different virtual addresses
- Both the VAs map to the same physical address through the page tables
- What will be the output of P2?
- In a uniprocessor system?
- Does the OS play any role?
- Does the hardware play any role?

Why synchronize?

User level: multithreading

```
static int count = 0;

void *thfunc( )
{
   int ctr=0;
   for(ctr=0; ctr<10000; ctr++)
         count++;
}
```

- If a single thread execute the program, what will be the final value of count?
- If two threads execute the program concurrently, what will be the output?
- What are the possible values if N threads execute the function in a concurrent manner?

Why synchronize?

User level: multithreading

```
static int count = 0;

void *thfunc( )
{
   int ctr=0;
   for(ctr=0; ctr<10000; ctr++)
         count++;
}
```

- If a single thread execute the program, what will be the final value of count?
- If two threads execute the program concurrently, what will be the output?
- What are the possible values if N threads execute the function in a concurrent manner?
- Any ideas how to guarantee count = N*10000?

Why synchronize?

OS level: sys_open( )

```
do_open( )
{
   ........
   alloc_inode( );
  .........
}
alloc_inode( )
{
   .........
   scan_and_find (inode_bmap);
  ..........
}
```

- If two programs execute open with O_CREAT for two different files, the inode bitmap require concurrent modification
- Issues?
- Crux: serialize write access to the inode bitmap
- How?

Why synchronize?

- How to serialize write accesses to the inode bitmap?
- Approach 1: Disable preemption during open( ) system call
    - correctness, efficiency, ease of implementation
    - Uniprocessor vs. Multiprocessor systems
- Approach 2: Disable preemption only during the access to the inode bitmap
- Approach 3: Implement locks
    - Before accessing the bitmap, acquire the lock
    - Before modifying the bitmap, acquire the lock

Why synchronize?

```
Queue *Q = INIT_QUEUE( );

do_NIC_irq( )
{
   ...........
   add_packet_to_queue(Q, pkt);
   wakeup_waiters(Q)
}
```

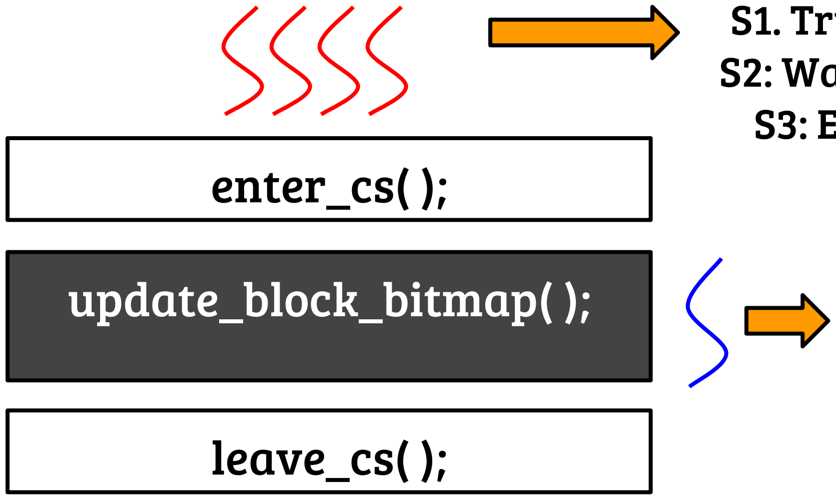
```
do_receive( )
{
     if(empty(Q))   WAIT(Q);
     while(!empty(Q)){
         pkt = next_packet(Q);
         copy_to_user(pkt)
     }
   /*Return to user*/
}
```

- Possible issues?
- Solution approaches: single CPU vs multi-processor systems

Some definitions

- Atomic operation: An operation is atomic if it is *uninterruptible* and *indivisible*
- Critical section: A section of code accessing one or more shared resource(s), mostly shared memory location(s)
- Mutual exclusion: Technique to allow exactly one execution entity to execute the critical section
- Race condition: Occurs when multiple threads are allowed to enter the critical section

Critical section



Contenders

S1. Try to enter CS

S2: Wait if required

S3: Enter the CS

Acquirer

S1. Enter the CS

S2: Leave and allow others to enter CS

- Solution requirements: *mutual exclusion, fairness and performance*
- Performance in terms of lock acquire latency vs. CPU overheads

Entry to critical section: design choices

Alternate 1

```
while(enter_cs( ) == false);
```

Alternate 2

```
while(enter_cs( ) == false)
        schedule( );
```

- Lock acquisition latency is small in an ideal case
- CPU cycle wastage
- Commonly known as spinlocks

- Lock acquisition latency depends on context switch overheads
- Efficient use of CPU cycles
- While leaving the CS, the context should wake up waiters

Critical sections in OS

Contexts executing critical sections	Uniprocessor systems	Multiprocessor systems
System calls	Disable preemption	Locking
System calls, Interrupt handler	Disable interrupts	Locking + Interrupt disabling (local CPU)
Multiple interrupt handlers	Disable interrupts	Locking + Interrupt disabling (local CPU)

- How to design a lock?
- How to hold the lock and disable interrupt at the same time?

First attempt: Wrong implementation of a spinlock

```
s_lock(int *lock )
{
  while(*lock);
  *lock=1;
}

s_unlock(int *lock )
{
   *lock = 0;
}
```

- What is wrong with this implementation?

Buggy spinlock in assembly (x86)

```
s_lock(int *lock )
{
   asm volatile(
   “loop: cmp $0x0, (%%rdi);\n ”
    “jne loop; \n”
    “mov $1, (%%rdi);”
     : : : “memory” );
}
s_unlock(int *lock )
{
   *lock = 0;
}
```

- Approaches to fix the code?
- If only comparison and assignment be done atomically!
- What about an atomic exchange between a CPU register and memory?

Spinlock using XCHG (x86)

```
s_lock(int *lock )
{
   asm volatile(
   “mov $1, %%rax;\n ”
   “loop: xchg %%rax, (%%rdi);\n ”
   “cmp $0, %%rax;\n”
    “jne loop; \n”
     : : : “memory” );
}
s_unlock(int *lock )
{
   *lock = 0;
}
```

- XCHG R, M → Atomically exchange contents of R and M
- Is the lock implementation correct?

CMPXCHG (x86)

cmpxchg source[Reg] destination [Mem/Reg]

implicit registers : rax and rflags

```
if rax == [destination]
 then
        rflags[ZF] = 1
        [destination] = source
 else
        rflags[ZF] = 0
       rax = [destination]
```

CMPXCHG (x86)

cmpxchg source[Reg] destination [Mem/Reg]

implicit registers : rax and rflags

```
if rax == [destination]
 then
        rflags[ZF] = 1
        [destination] = source
 else
        rflags[ZF] = 0
       rax = [destination]
```

Catch: cmpxchg is not atomic!

Spinlock using CMPXCHG (x86)

```
s_lock(int *lock )
{
   asm volatile(
   “mov $1, %%rcx;\n ”
   “loop: xor %%rax, %%rax;\n”
    “lock cmpxchg %%rcx, (%%rdi);\n ”
    “jnz loop; \n”
     : : : “rcx”,  “rax”, “memory”);
}
s_unlock(int *lock )
{
   *lock = 0;
}
```

- Note the “lock” prefix
- Is the implementation correct?
- How else “lock” prefix can be used?

Fairness in spinlocks

- Spinlock implementations discussed so far are not fair, no bounded waiting
- To ensure fairness, some notion of ordering is required
- What if the threads are granted the lock in the order of their arrival to the lock contention loop?
- A single lock variable may not be sufficient
- Possible with multiple variables and *atomic fetch and add*

Atomic fetch and add (xadd in X86)

xadd SRC[Reg] DST [Mem/Reg]

TMP ← SRC + DST
SRC ← DST
DST ← TEMP

- Require lock prefix to be atomic
- Example:

```
mov $100, M; mov $200, %rax

lock xadd  %rax, M
```

- Value of rax = 100, [M] = 300

Atomic fetch and add (xadd in X86)

xadd SRC[Reg] DST [Mem/Reg]

TMP ← SRC + DST
SRC ← DST
DST ← TEMP

- Require lock prefix to be atomic
- Example:

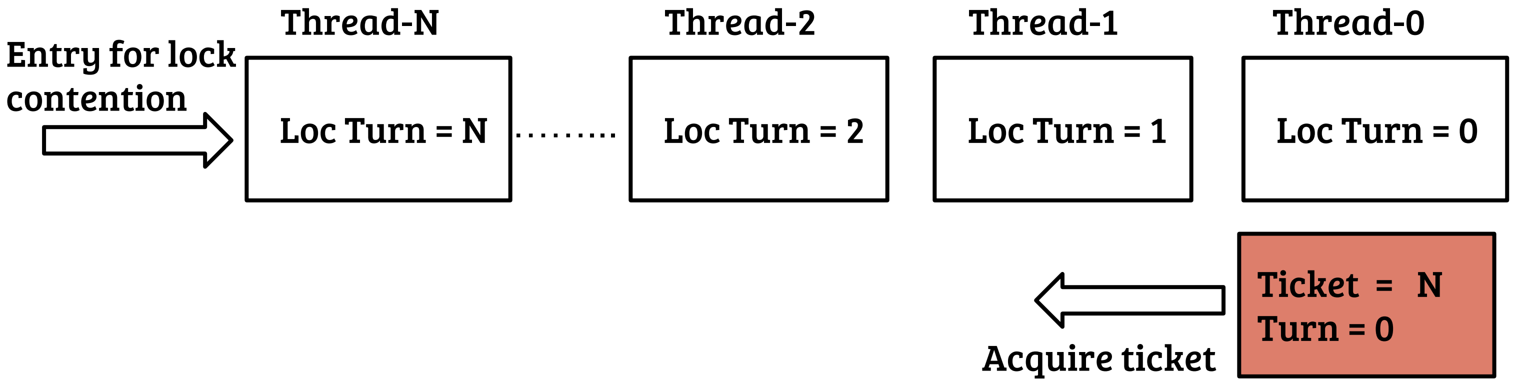
mov $100, M; mov $200, %rax

lock xadd %rax, M

- Value of rax = 100, [M] = 300

- Any ideas of how to implement a fair lock using two variables and all the instructions discussed?

Ticket spinlock



- Local variable “Loc Turn” is equivalent to the order of arrival
- Shared “Turn” reflects who owns the lock
- If a thread is in CS ⇒ Local Turn must be same as “Turn”
- Increment “Turn” during unlock
- Value of ticket incremented on a new thread arrival
- Which operations should be atomic?

Ticket spinlock using xadd (X86)

```
slock_ticket( int *ticket, int *turn)
{
  asm volatile( "mov $1, %%rax;\n"
              "lock xadd %%rax, (%%rdi);\n"
              "loop: cmp %%rax, (%%rsi);\n"
              "jne loop;"
              :::"rax", "memory"
              );
}
unlock_ticket(long *turn)
{
    (*turn)++;
}
```

- Any assumptions regarding fairness?
- How to ensure fairness?

Read-write locks

- Spinlock does not distinguish between read and write access to shared variables/data structures
- Many real life scenarios exhibit that behavior
- Example 1: Search and insert on a list
- Example 2: Search a file block in disk cache with concurrent insertions
- Allow multiple readers when no write is going on, how?

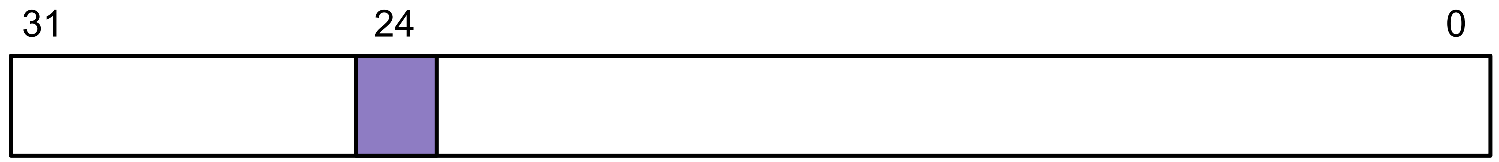
A simple read-write lock

```
struct rw_lock{
                                Spinlock R;          #define write_lock(L)      spin_lock(L->G)
                                Spinlock G;          #define write_unlock(L)  spin_unlock(L->G)
                                Int count;
                          };

read_lock (struct rw_lock *L){                               read_unlock (struct rw_lock *L){
    spin_lock(L->R);                                             spinlock(L->R);
    L->count++;                                                  L->count--;
    If (L->count == 1)                                           if(L->count == 0)
       spin_lock(L->G);                                              spin_unlock(L->G);
   spin_unlock(L->R);                                            spin_unlock(L->R);
}                                                            }
```

Improved read-write lock

- Simple R/W lock requires two spinlocks and read accesses are not fully concurrent
- How to improve? Can we get rid of the two locks?



- Example R/W lock with 32-bit integer
- 0x1000000 → Free, 0x0 → Acquired for write, [0xFFFFFF, 0x0] → Readers
- How to implement?

Software implementation of a lock (buggy #1)

```
int flag[2] = {0,0};
void lock (int id)   /*id = 0 or 1 */
{
   flag[id] = 1;
   while(flag[id ^ 1])); // ^ → XOR
}

void unlock (int id)
{
    flag[id] = 0;
}
```

- Solution for two threads, T_0 and T_1 with id 0 and 1, respectively
- What is the problem?
- What if we switch the two statements inside lock?

Software implementation of a lock (buggy #2)

```
int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   while(turn == id^1));
}

void unlock (int id)
{
    turn = id ^ 1;
}
```

- Solution for two threads, T_0 and T_1 with id 0 and 1, respectively
- Assuming T_0 invokes lock first, Does the solution provide mutual exclusion?
- What is the problem?

Software implementation of a lock (buggy #3)

```
int flag[2] = {0,0};
Int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   while(flag[id ^ 1]) && turn == (id ^1));
}

void unlock (int id)
{
    flag[id] = 0;
    turn  = id ^1;
}
```

- What is the problem scenario?
- What if we switch the two statements in unlock?
- Does it help if two statements in unlock are executed atomically?

Software implementation (Peterson’s solution)

```
int flag[2] = {0,0};
Int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   turn  = id ^1;
   while(flag[id ^ 1]) && turn == (id ^1));
}

void unlock (int id)
{
    flag[id] = 0;
}
```

- How does the solution guarantee mutual exclusion?
- How fair is this lock?
- What if first two statements of lock are swapped?
-

Software implementation (Peterson’s solution)

```
int flag[2] = {0,0};
Int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   turn  = id ^1;
   while(flag[id ^ 1]) && turn == (id ^1));
}

void unlock (int id)
{
     flag[id] = 0;
}
```

- Any assumptions regarding correctness?
- Software solutions are not used, why?