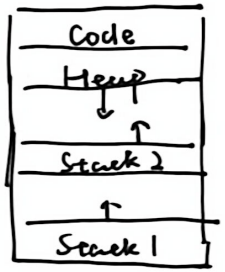
§ 26. Concurrency: Introduction & Threads.

§ Thread

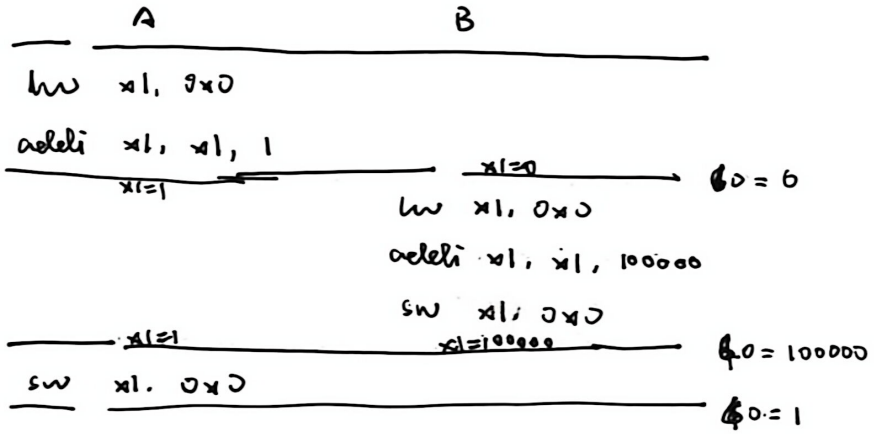
Multithreaded program

- has multiple point of execution (PC)
- share the same address space.

Context switch.



§ Shared Data.



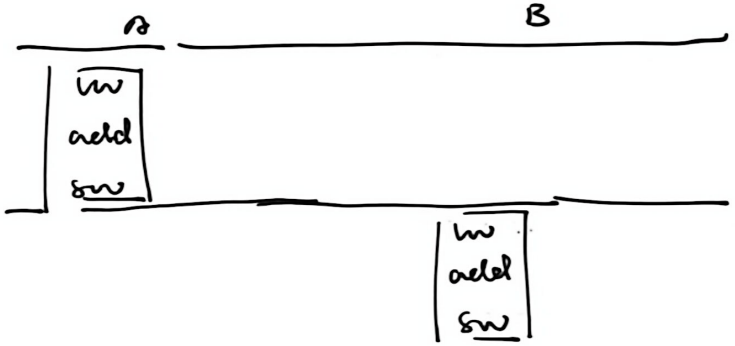
Race condition. Indeterminate.

```
mythread()
    for i in range(N)
        [counter++]  --> critical section.
```

Want: mutual exclusion.

§ Atomicity

As a unit / All or none.



Synchronization Primitives.

§ Waiting for Another

omitted.

§ 27. Thread API

omitted.

§ 28. Locks

```
lock_t mutex
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

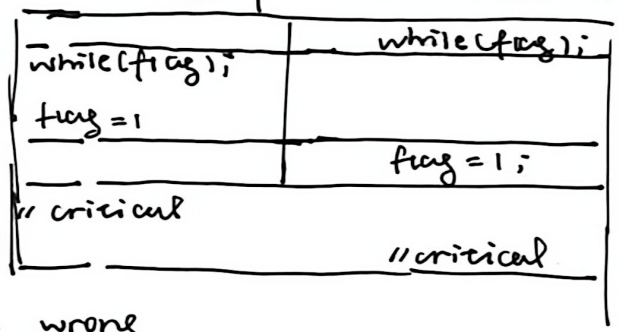
§ Evaluating Locks

1. mutual exclusion
2. fairness (starve?)
3. performance

§ Controlling Interrupts

Cons: 1. trust
2. doesn't work on multiprocessors
3. lost interrupts

§ A Failed Attempt: Loads / Stores.



1. wrong
2. performs bad (spin-waiting).

§ Test-And-Set Spinlock

x86 xchg

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr;
    *old_ptr = new;
    return old
}                                   // atomically!

while(TestAndSet(&flag, 1)); // lock
...
flag = 0.                           // unlock
```

Correctness ✓ Fairness ✗

performance O (singleCPU ✗ multiCPU ✓)

§ Compare-And-Swap

x86 compare-and-exchange

```
int CompareAndSwap(int* ptr, int expected, int new)
    int actual = *ptr;
    if (actual==expected) *ptr=new;
    return actual;
}
```

Wait-free sync.

§ Load-Linked & Store-conditional

optimistic

```
int LoadLinked(int *ptr){
    return *ptr;
}

int StoreConditional(int *ptr, int value){
    if( *ptr was updated since LoadLinked ){
        return 0;   // fail
    } else {
        *ptr = value;
        return 1;   // success
    }
}
```

```
Lock:
  while( LL(&flag) || !SC(&flag, 1) );
```

§ Fetch-And-Add. Ticket Lock.

```
lock:
    int myturn = FAA(&lock->ticket);
    while(myturn != lock->turn);
    //
unlock:
    FAA(&lock->turn);   //or ++
```

Ticket Lock. Fairness √. (FIFO)

§ Too much Spining

N-1 slices wasted.

1. Yield

   waste still substantial;

   still starvation.

2. Queue √.

3. Hybrid.

§

§ 29. Locked Data Structures

makes the structure threaded safe.

§ Concurrent Counters.

Simply add mutex: not scalable!

Sloppy counter.

S (sloppiness). threshold.

L1 L2 L3 L4. g.

§ Concurrent Linked-list.

```
void insert(list_t *L, int key) {
    ⋮                          // method is safe.
    lock();
    new->next = L->head;
    L->head = new;
    unlock();
}.

void lookup(___) {
    lock();
    node_t *curr = L->head;
    while(___) ___;
    unlock();
}
```

hand-over-hand locking / lock coupling.

R add a lock per node.

when traversing

1. grabs the next node's lock
2. releases the current node's lock.

Unlikely faster in fact.

§ Concurrent Queue.

Dummy.

Head-lock & Tail lock.

§ Concurrent Hash Table.

Simply uses concurrent list.

§ 30. Condition Variables

A thread wishes to check whether a condition is true before continuing. e.g. join();

1. Spin. 2. Condition variables.

§ Definition & Routines.

[lock(m)]
wait(&m). [unlock]

[lock(m)]
signal. [unlock]

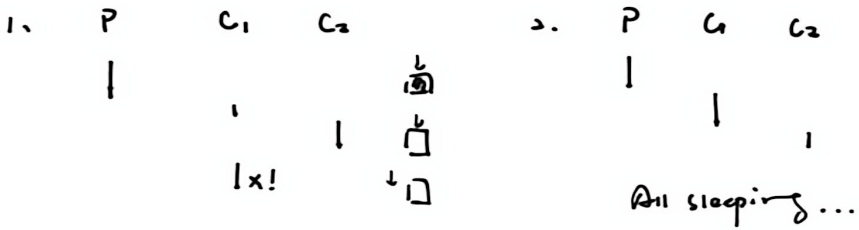
Hold the lock when calling signal!
(cond wait)

1. Enqueue
2. Unlock
3. Block
4. Lock

§ Producer & Consumer.

1. write. (Mesa semantic)

2. empty/fill 2 cond.



§ 31. Semaphores

§1. Definition.

```
sem_t s;
sem_init(&s, 0, 1);
                 ↑ value.

int sem_wait(sem_t *s) {
    decrement s.value--;
    wait if s.value<0.
}
int sem_post(sem_t *s) {
    s.value++;
    if s.value<=0 wake one;
}.
```

Conditional variables
that can
wait & wake.

§2. Binary Semaphores (Locks).

Initial value = 1.

§3. Semaphores for Ordering
(& Condition Variables).

Initial value = 0.

§4. The Producer/Consumer (Bounded Buffer) Problem

```
producer() {
    while(1) {
        sem_wait(&e);
        sem_wait(&m);
        push(&st, rand());
        sem_post(&m);
        sem_post(&f);
    }
}.
```

```
consumer() {
    while(1) {
        sem_wait(&f);
        sem_wait(&m);
        pop(&st);
        sem_post(&m);
        sem_post(&e);
    }
}.
```

§5. Reader-Writer Locks.

```
sem_t writelock;

// first reader acquires writelock
// last reader releases writelock.
```

§ 32. Concurrency Bugs

Non-Deadlock & Deadlock

§ 2. Non-Deadlock Bugs

1. Atomicity Violation

```
T1                                  T2
if(thd->proc_info){                 thd->proc_info
  ...                                 = NULL;
  fputs(thd->proc_info,...);
  ...
}
```

Solution: add mutex.

2. Order Violation

```
T1
mThread = PR_CreateThread(mMain, ...);

T2
mState = mThread->State;
```

Solution: add condition variables.

§ 3. Deadlock Bugs.

1. Mutual exclusion
2. Hold-and-wait
3. No preemption
4. Circular wait.

Circular wait

total / partial ordering.

Hold-and-Wait

acquiring all locks at once, atomically.

make a big lock

No Preemption

~~trylock~~: trylock.

Livelock.

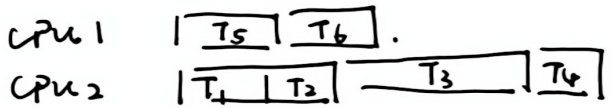
```
top:
  lock(L1);
  if(trylock(L2)){
    unlock(L1);
    goto top;
  }
```

Mutual Exclusion

Lock-free (/ wait-free) structures.

Deadlock Avoidance via Scheduling.

	T_1	T_2	T_3	T_4	T_5	T_6
L1	✓	✓	✓	✓	✓	
L2	✓	✓	✓	✓		✓



Dijkstra's Banker's Algorithm.

Detect and Recover

Deadlock detector runs periodically,

building a resource graph and checking it

for cycles.

If deadlocked, reboot.

§33. Event-based Concurrency.