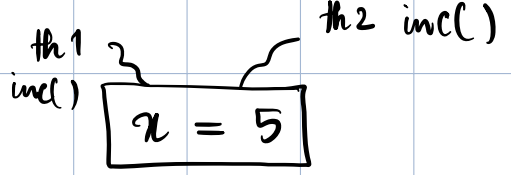
18 Feb 2025 - Operating Systems - II - Week 07

* New project assignment today

→ Atomic variables in C++ cppreference.com

→ increment ( ) without hardware support ⇝ uses compare_and_swap internally



drawback

→ ~~progress~~, bounded waiting

→ multiple threads: same threads keep getting the chance (CAS is non-deterministic)

1. int temp
2. do
   1. temp = *v

   while (temp != compare_and_swap(v, temp, temp + 1))

→ Solution → hardware level instruction fetch_add (C++)
returns the old value
and atomically increments

Mutex Locks

→ atomics may not always be easy → balanced BST
rotation requires modifying
multiple locations

→ Mutex locks :

→ multiple threads can
be doing lookups
and ocasional
updates
→ will need locks too

1. while (true)
    1. acquire lock → must be atomic
    2. critical section
    3. release lock → must be atomic
    4. remainder section

multiple threads
acquire locks

→ Fair locks and unfair locks

↓

entry section = CAS ⇒ not fair

→ Definitions (not implementations) → must be atomic

if not, two threads will enter CS.

→ acquire()

1. while (! available)

; // busy wait → not doing anything useful

2. available = false

→ alternative:

→ sleep: give up CPU

↓

more complex implementation

→ release()

1. available = true

→ Fix ? ① and ② in acquire() must be atomic

Thr 1	Thr 2
L6 : reads available as true	L6 : reads available as true
L7 : sets avail = false;	L7 : sets available = F ;

→ Only load and store being atomic is not sufficient.

→ Fix: Use CAS.

→ Bounded waiting ?

→ RISV ⇝ try running multiple threaded load_and_store
⇝ atomic class

→ Intel ⇝ load and store is atomic by default (test)
↓
array, multiple threads write; check logs and see if it makes sense.

→ Natural extension of a lock is a Semaphore

Semaphore

→ locks for a group of threads

→ proposed by Djikstra (1960s)

→ wait() and signal() : atomic operations
indivisible

→ wait (S) // Def^n ; not an implementation

1. while (S ≤ 0)
   ; // busy wait : spinning
2. S--;

→ signal (S)

1. S++;

→ Usage :

1. Counting semaphores
2. Binary semaphores ⇝ same as mutex lock

→ Can solve many synchronization problems

→ Ex: assume synch initialized to 0.

We want S_1 in P_1 to happen before S_2 in P_2

P_1:

S_1 ;

signal (synch);

P_2:

wait (synch);

S_2 ;

→ Counting semaphores can be implemented using binary semaphores.

→ Template, no busy waiting → block ( ) and wake up ( )

19 Feb 2025

→ Counting and Binary semaphores.

→ Semaphore implementation with no busy waiting. → keep looping / spinning

→ typedef struct {
    int value
    processes_list
} semaphore

→ wait and si

→ block() / sleep() ⇝ informs the kernel to make the thread give up the CPU and remove it from ready queue } thread stored in semaphore queue list.

→ wakeup( )

→ If sleeping time is not too long, busy waiting is okay.
in multicore system

→ spin lock ⇝ no context-switch

→ In single-core system, spin lock is obviously not allowed.

Semaphores
- spin lock: busy waiting
- lock: no busy waiting

Problems : semaphore code is not protected
entire wait ( ) and signal ( ) must be atomic.

Solution : protect code with CAS
atomic X { } We have a struct here

S = 1

Thr 1	Thr 2
L2 : s →val -- ; S = 0	
	L2 : S→val -- ; S = -1
	L3
	L4 Thr 2 gets added
L3 :	
L4 : Thr 2 gets added	

time ↓

Both of them get blocked, no one to wake up

Progress

not satisfied

S = 1

	Thr 1	Thr 2
5:	wait (* s)	wait (* s)
6:		Mutual exclusion (?)

- Try for S = 2 (three threads)

Book: Operating Systems : Three Easy Steps

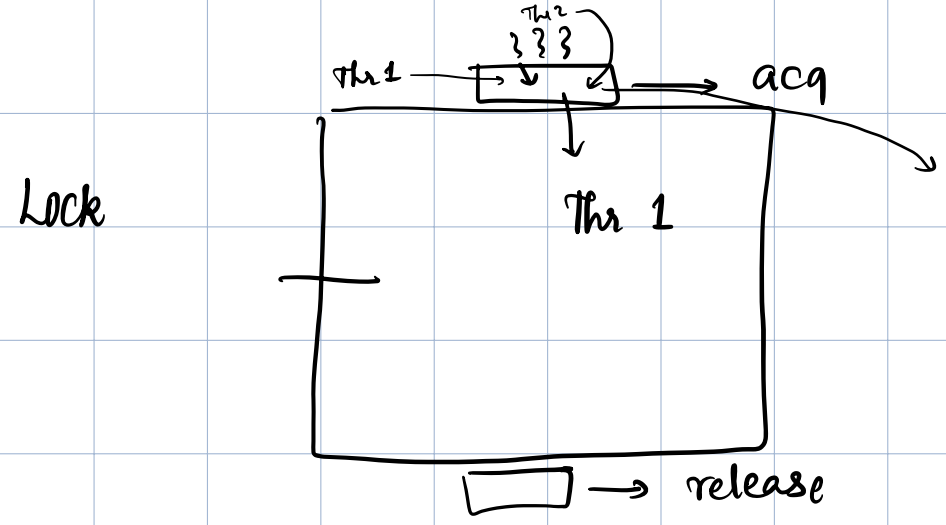
Locks

```
struct lock {
    flag, guard, queue
}
```

```
lock_init() {
}
```

lock () ≡ acqure

Read from book pg 348



Queues ⇝ also require locks

Bank Account

```
Lock lAcct;
lock_init(& lAcct);
Thr 1 ( ) {
    ...
5:  acq(& lAcct);
6:
```

```
Thr 2 ( ) {
    ...
5:  acq(& lAcct)
6:
```

lock needs C.S

} only one of them will reach 6

For C.S, you need C.S

To maintain flag and queue, you need guard.
C.S

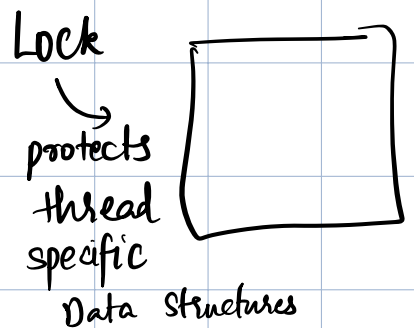
Why?

→ performance, hardware today is multicore

→ 1 program uses 1 of 64 cores ⟿ very inefficient

→ try parallelizing code as much as you can.

→ If not possible, execute sequential code using these locks and semaphores.



Test and Set → protects lock specific D.S
flag, guard
and queue

→ analogy: Hospital system

Ticket locks

fetch_and_add(int * 0, int v) → 1 in book

int temp = * 0

* 0 = * 0 + v

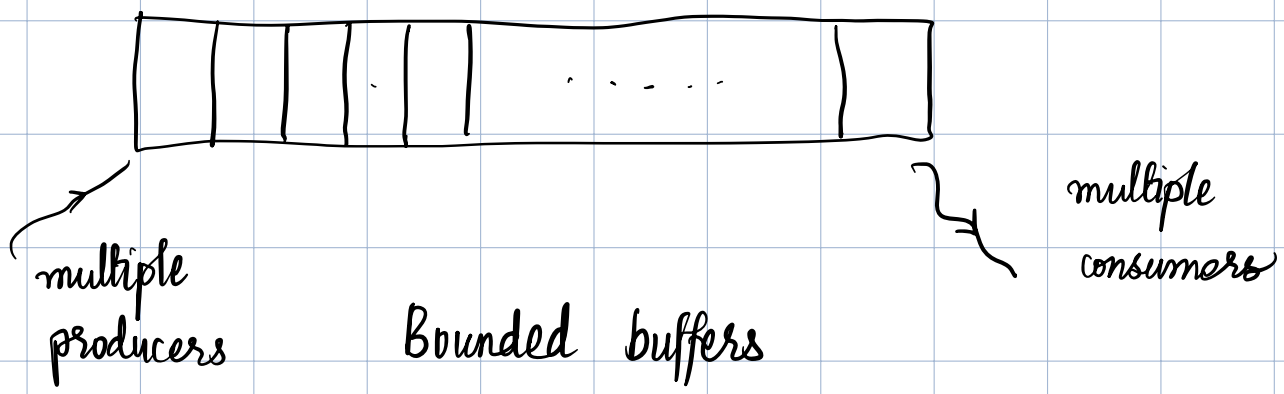
return temp

ticket

turn

Semaphores

Bounded buffer problem



n buffers, each can hold 1 item

Semaphore **mutex** initialized to 1

Semaphore **full** initialized to 0

Semaphore **empty** initialized to n

- produce - consumer . cpp .