Race conditions

Sunday, April 19, 2020 9:34 AM

When several threads access and manipulate the same data concurrently, the outcome of the execution depends on the particular order in which the access takes place.

This is called a **race condition**.

Bugs in multi-threaded programs caused by race conditions are hard to debug and are called "Heisenbugs."

Examples: race.c and no_race.c on BBLearn

$\boxed{\text{Counter} = 5}$

S_0

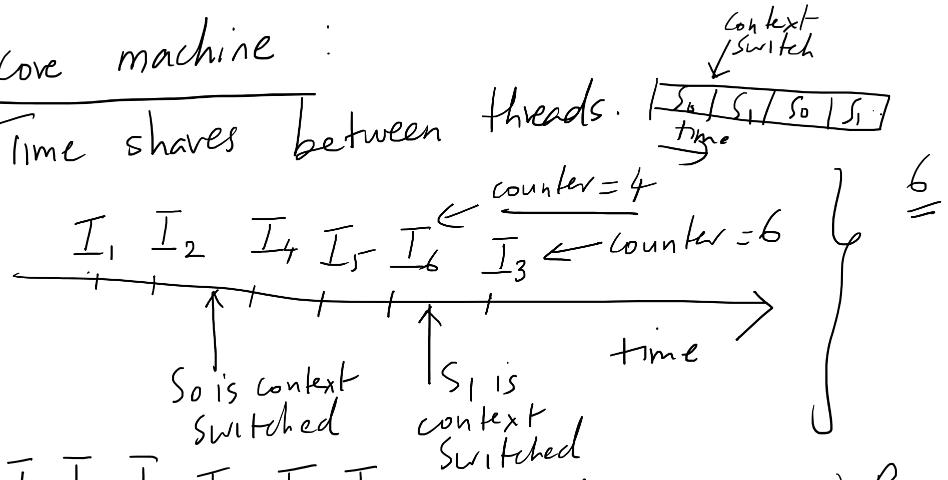
I_1 lw $r1, <counter>
(6) I_2 addi $r1, $r1, 1   counter++; ← not atomic
I_3 sw $r1, <counter>

S_1

counter--;
lw $r2, <counter> I_4
(4) subi $r2, $r2, 1 I_5
sw $r2, <counter> I_6

Single core machine:

· Time shares between threads. | S_0 | S_1 | S_0 | S_1 |



· $I_1 \, I_2 \, I_4 \, I_5 \, I_6 \, I_3 \rightarrow 6$ ✗

· $I_4 \, I_5 \, I_1 \, I_2 \, I_3 \, I_6 \rightarrow 4$ ✗

· $\underbrace{I_1 \, I_2 \, I_3}_{S_0} \, \underbrace{I_4 \, I_5 \, I_6}_{S_1} \rightarrow 5$ ✓

· $\underbrace{I_4 \, I_5 \, I_6}_{S_1} \, \underbrace{I_1 \, I_2 \, I_3}_{S_0} \rightarrow 5$ ✓

Race condition:

$\cancel{4}$, 5 ✓, $\cancel{6}$

Multi-core machine

time ↓

S_0 (core 0)

```
lw $r1, <counter>
addi $r1, $r1, 1
sw $r1, <counter> |
```

S_1 (core 1)

```
lw $r2, <counter>
subi $r2, $r2, 1
sw $r2, <counter> |*
```

Whoever writes last wins. (4 or 6)

S_0 S_1 or S_1 S_0 } ✓

Condition:

→ Atomicity → indivisible

counter ++ (or) counter --

operations must be atomic

Most machine instructions are not atomic

The critical section problem

Sunday, April 19, 2020 9:38 AM

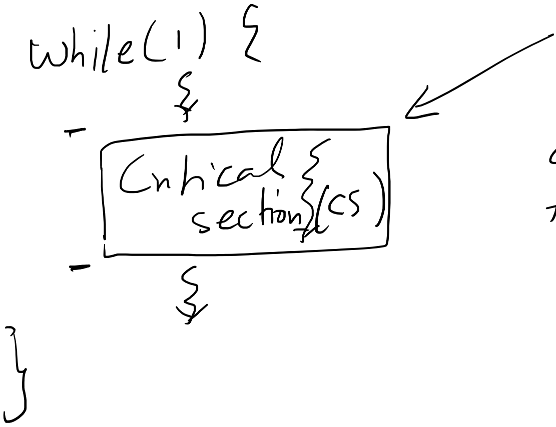
Critical section: segment of code in which threads may be modifying shared variables

Any solution to the critical section problem must satisfy the following three conditions:

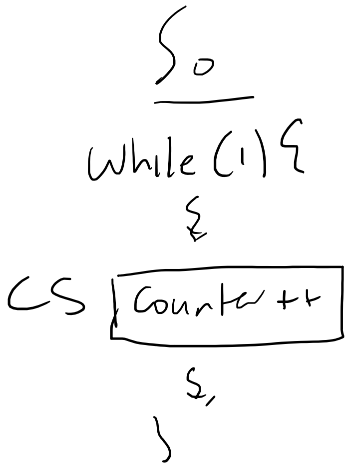
Mutual exclusion: if thread T_i is executing in its critical section, then no other threads can be executing in that critical section.

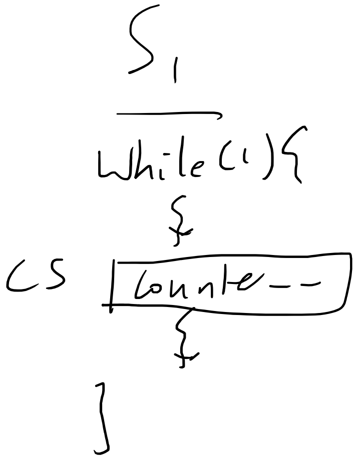
Progress: all threads must be making progress towards their overall objective. This means that the system is free of **deadlocks**.

Bounded waiting: there exists a bound or limit on the number of times other threads are allowed to enter their critical sections after a thread has made a request to enter its critical section and before the request is granted.



set of instructions that are operating on data shared between threads





Hardware support for synchronization

Sunday, April 19, 2020 9:44 AM

- Instruction ~~Set~~ architecture (ISA)
  - ↳ exchange (XCHG)
  - ↳ atomic CAS ( )

↑ compare and swap

} ← atomic instructions

↑ implement lock or mutex

Simplest way to achieve mutex:

→ Microcontrollers (MSP432 TI)

↳ single core → time sharing

↳ CPU is time shared between threads

↓ time quanta

↓ timer

↓ interrupt

↓ ISR()

Interrupt service routine → ISR()

```
While(1){
    /* Disable interrupts */
    Critical section
    /* Enable interrupts */
}
```

keep short

Will not work for multi-cores.

↳ Each core has its own interrupt set up.

atomic CAS

Behavior of atomic CAS:

shared memory

mutex

atomic

```
int atomicCAS(int *mutex, int compareVal,
                          int newVal)
{
      int oldVal = *mutex;
      if (*mutex == compareVal)
              *mutex = newVal;
      return oldVal;
}
```

Lock implementation:

mutex — Shared variables

int mutex = 0

S_0

```
While(1){
  While(atomicCAS(&mutex,0,1)!=0);
    Critical section {
      mutex = 0;
}
```

short

S_1

```
while(1){
  While(atomicCAS(&mutex,0,1)!=0);
    Critical section
      mutex = 0;
}
```

spin lock

Exchange:

atomic

```
void exchange (int *mutex, int registerVal)
{
        int temp;
        temp = *mutex;
        *mutex = registerVal;
        registerVal = temp;
}
```

mutex = 0

S0

```
int key = 1;
while ( key != 0)
    exchange(&mutex, key);

}, | critical section |

    exchange(&mutex, key);
```

S1

```
int key = 1;
while (key != 0)
    exchange(&mutex,
             key);

| critical section | {

exchange(&mutex,
         key );
```

spin lock

Recap:

- Atomicity → operations in CS should be indivisible
- Isolation → when multiple threads operate concurrently on a data structure, the final state should be the same as though the operations were performed sequentially.

The pthread mutex

Sunday, April 19, 2020 9:45 AM

- `pthread_mutex_t mutex;`

  ↑

  pthread mutex type

- Initialize mutex:

  `pthread_mutex_init(&mutex, NULL);`

- Destroy mutex:

  `pthread_mutex_destroy(&mutex);`

- Lock mutex:

  `pthread_mutex_lock(&mutex);`

- Unlock mutex:

  `pthread_mutex_unlock(&mutex);`

Semaphores

Sunday, April 19, 2020 9:45 AM

- Signalling semaphores
- mutex

Semaphore : Integer

Counting semaphore can take on any integer value (0, 1, . . r − 1)

Binary semaphore ⇓ mutex

Basic operations

- Probe (P)
- Signal (V)

} → atomic

S : semaphore

```
Probe:
  P() or sem_wait():
    If (S==0)
      put ourself in a wait queue
      for S;
    S--;
```

```
Signal or sem_post():
  prev_val = S;
  S++;
  if((prev_val ==0) && (threads are blocked on S))
    Wake up one thread that is
    blocked on S;
```

Binary semaphore

Values : 0, 1 $\boxed{S=1}$

$\boxed{\text{mutex}}$

S_0

sem_wait(S); → P(S);

CS

sem_post(S) V(S);

S_1

P(S); sem_wait(S)

CS

V(S); sem_post(S)

Example of signalling :

I_2 I_1 . . . .

$S = 0$

S_0

P(S);

I_1 : —

S_1

I_2 : —

V(S);

I_2 is executed first

I_1 is executed next

Recap

Tuesday, April 21, 2020 4:20 PM

Race conditions due to multiple threads modifying shared variables in unprotected fashion

Critical section problem: mutual exclusion and freedom from deadlocks

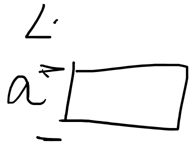
Hardware support for synchronization: atomic (indivisible) low-level instructions supported by the ISA; examples: atomicCAS(), exchange(), testSet()

Semaphores: counting and binary; basic atomic operations: probe (P) and signal (V)

Spinning versus context switching

"Smart" locks

Granularity of locking



S_0

```
lock(L);
/* CS */
unlock(L);
```

S_1

```
Lock(L);
/* CS */
unlock(L);
```

Uniprocessor case:

- Block the thread if it cannot acquire lock.

Multiprocessor case:

- Let threads spin on the locks
- Keep sizes of critical sections small

"Smart" locks:

↳ S_0 has lock
S_1 wants lock

If S_0 is currently executing on some core,
let S_1 spin

If S_0 is currently blocked, block S_1

Deadlocks

Sunday, April 19, 2020 11:08 AM

The use of locks serializes execution through critical sections -> loss of parallelism.

To reduce the impact on performance, the size of critical sections must be kept small (few hundreds of instructions or fewer) -> granularity of locking should be small.

Examples: non-preemptive kernel versus preemptive kernel.

Large applications (such as an OS kernel) may have thousands of locks in them.

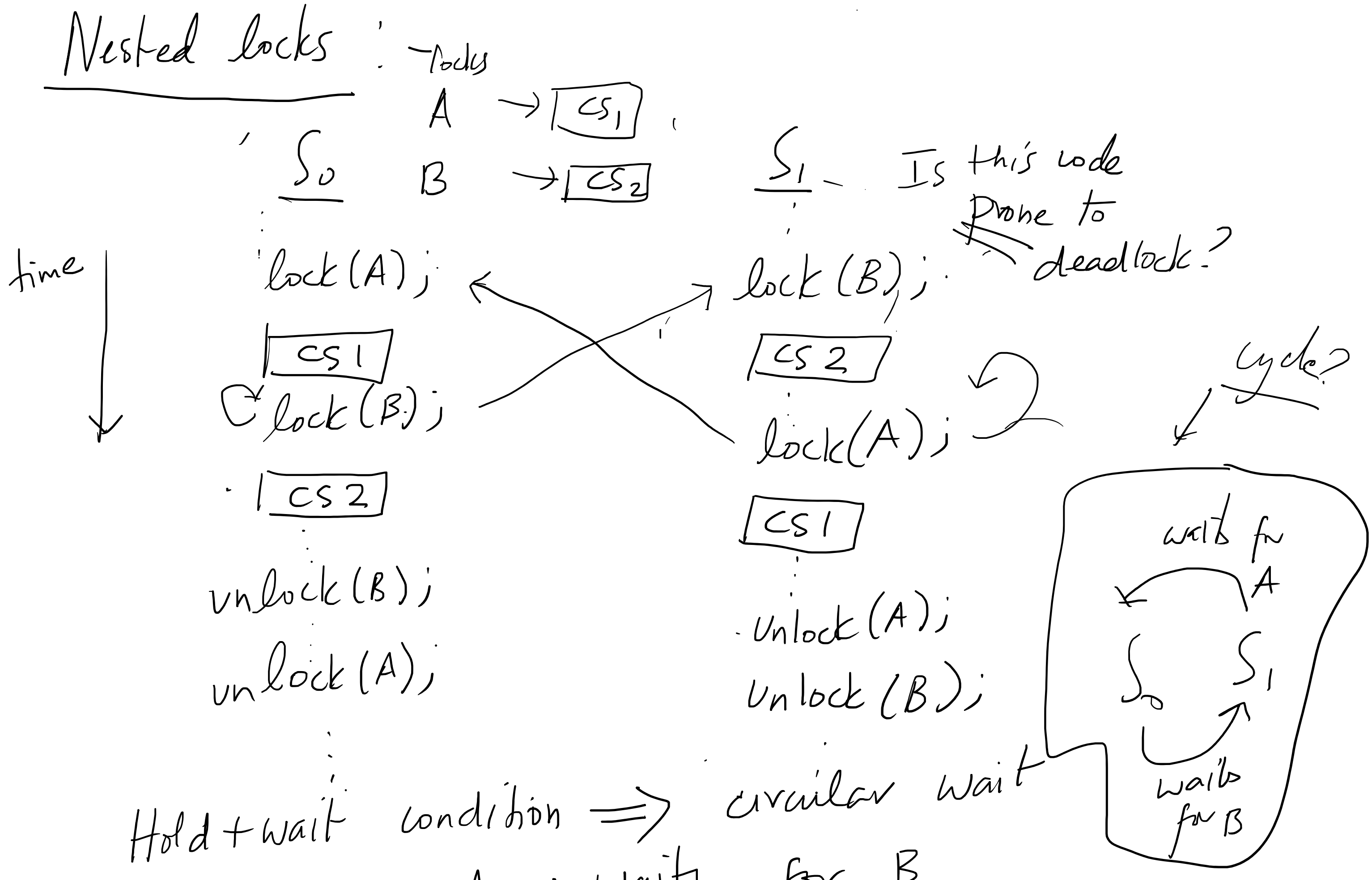
Must be careful to avoid deadlocks.

Examples…

How to handle deadlocks? Deadlock avoidance, deadlock prevention, deadlock detection, … or the Ostrich approach

Lock ordering protocol for deadlock avoidance

Nested locks :



S_0

```
lock(A);
CS 1
lock(B);
CS 2
unlock(B);
unlock(A);
```

S_1

```
lock(B);
CS 2
lock(A);
CS 1
Unlock(A);
Unlock(B);
```

Hold + wait condition ⟹ circular wait

S_0 holds A ; waits for B

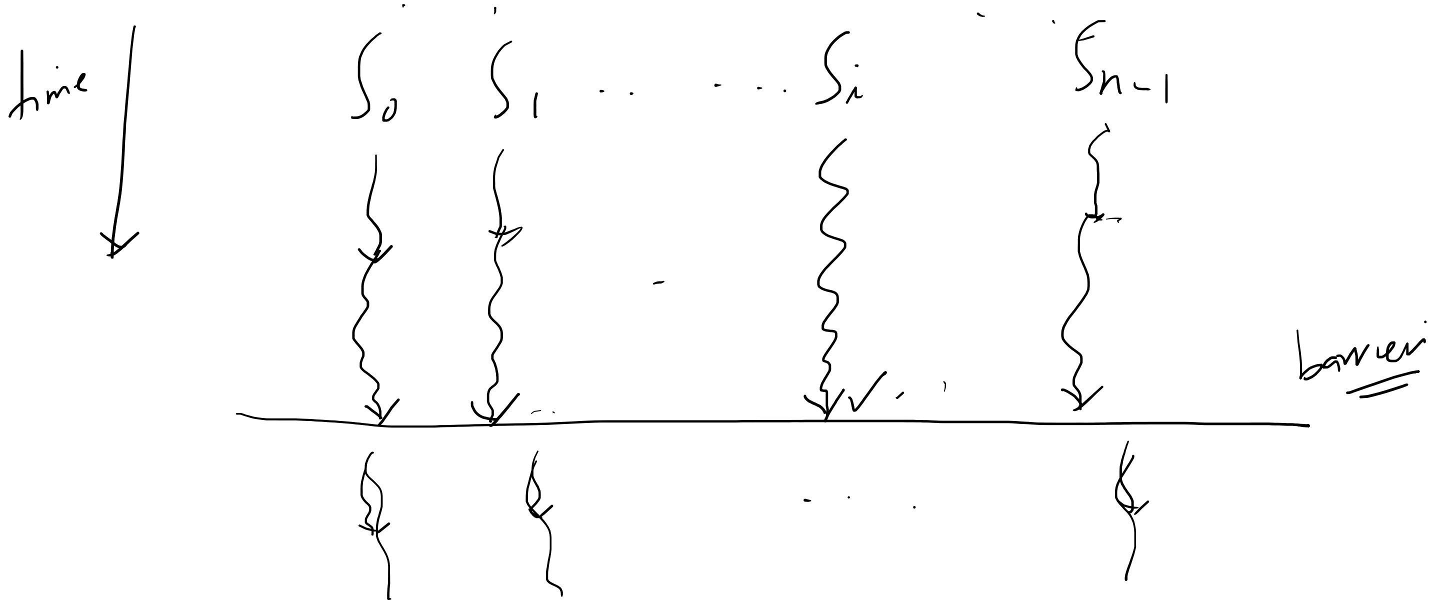
S_1 holds B ; waits for A

Code examples: Thread synchronization

Sunday, April 19, 2020 9:46 AM

1) Semaphores

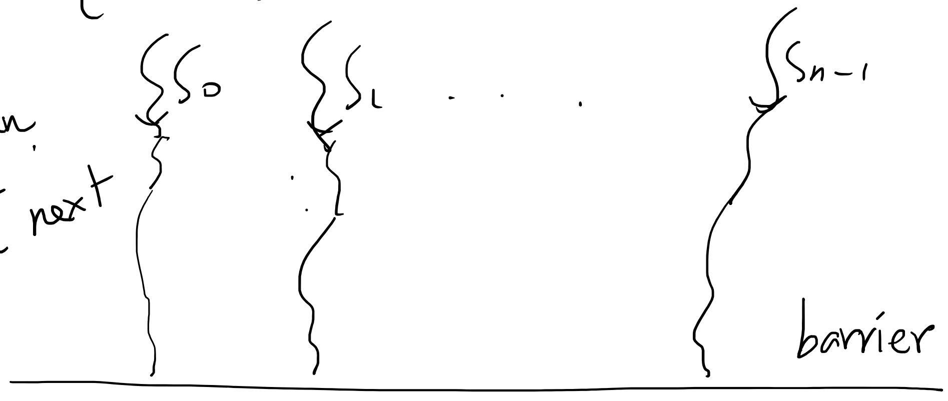
2) Barrier synchronization using semaphores



Iterative methods:

for (i = 0; i < num_iter; i++) {

Barrier ensures that values modified by threads during an iteration are stable before starting the next iteration.



}

If values calculated during iteration i-1 must be used during iteration i (example: Jacobi solver), then barrier is needed to force all threads to finish iteration i_1 before any thread is allowed to start iteration i.

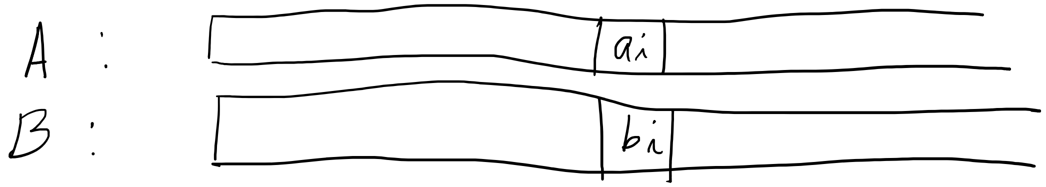
Pthread functions for barrier synchronization:

pthread_barrier_init();
pthread_barrier_wait();
pthread_barrier_destroy();

↳ Not all implementations may support.

Code examples: Vector dot product

Wednesday, April 22, 2020 11:51 AM

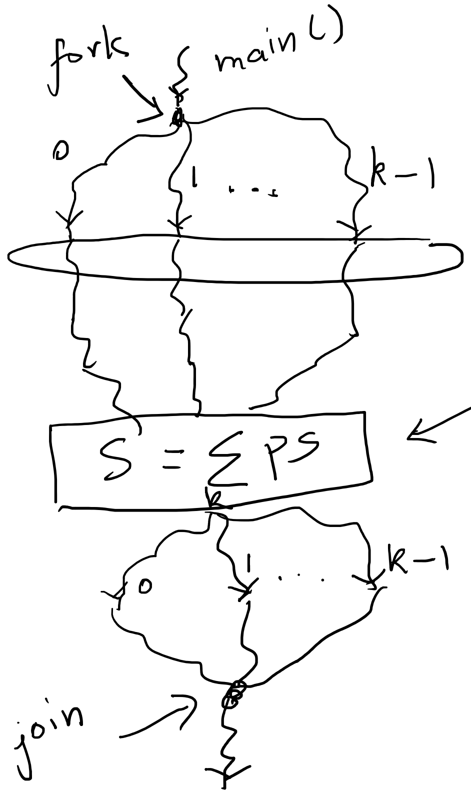


$$S = \sum_i a_i \cdot b_i$$

Use : chunking or striding

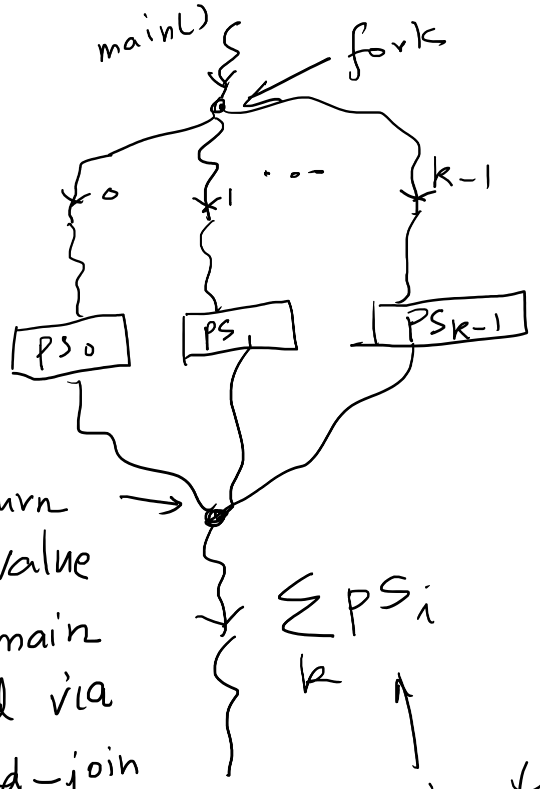
1) Lock to protect shared variable S when threads accumulate their partial sums.

parallel region: threads calculate partial sums (ps)



serialization into critical section

2) Lock free method



Store ps locally

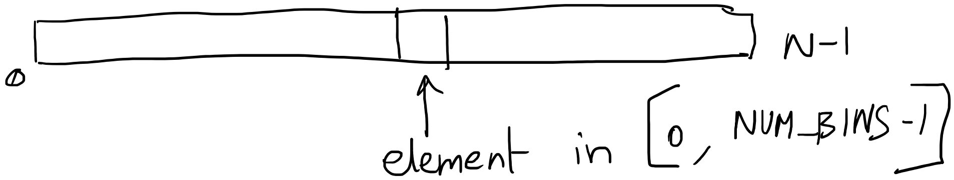
Return ps value to main thread via pthread_join

main thread calculates final sum.

Code examples: Histogram generation

Tuesday, April 21, 2020 4:20 PM

int bins[NUM_BINS]; input:



Serial code:

```
for(i=0; i<N; i++)
    bins[input[i]]++;
```

Parallelization:

Bad: have one histogram data structure and let threads constantly write to it.

Good: threads calculate their local histograms independently
Then, main thread can accumulate the local histograms into the final histogram.