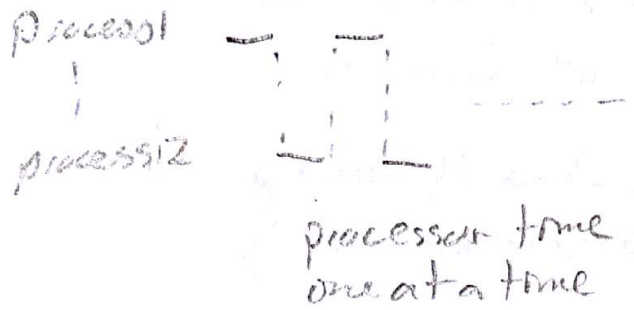
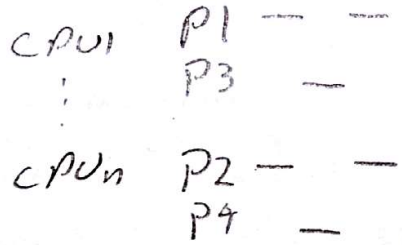
Chapter 5

① Multiprogramming - multiple processes - single CPU

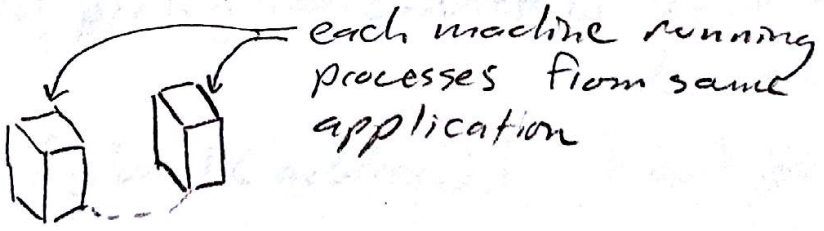


② Multiprocessing - multiple processes - multiple CPU's (or cores)



↑ notice that there are 2 processes running at once. (P1 & P2 are concurrent as are P3 & P4)

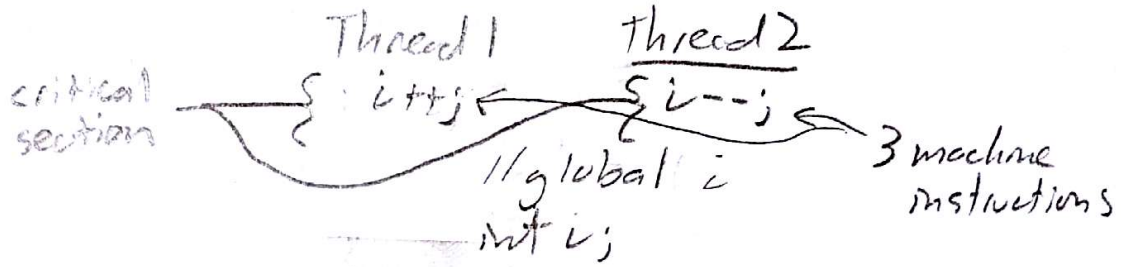
③ Distributed Multiprocessing: multiple machines, multiple processors (CPU) multiple processes



communication becomes much slower between processes (network latency) difficult to coordinate, not discussed in this course.

Race Condition - outcome depends on which thread finishes first. Often spurious, & tough to reproduce (may need exacting set of conditions). So usually tough to debug.

{see thread_problem_atomic_solution for example race condition}



Atomic Operation sequence of 1 or more operations that appear indivisible. No other process can see an intermediate state or interrupt it.

```
int i;
i++; ← is this atomic? No it compiles to this
```

```
mov, eax dword ptr [global (address)] // get it
add eax, 1                            // increment
mov dword ptr [global (address)]      // put back
```

can interrupt anywhere in the middle of above

3. {show how this leads to non-deterministic behavior in code}

---

solve with atomic<int> i, all 3 guaranteed to complete in 1 go.

but atomics are single line only!
what if you have this? 3 lines must complete. Cannot use atomics!

```
int iglobal=0;
void func(){
    int i;
    i = global;
    i = i + 27;
    global = i;
}
```

critical section ← called a

Critical Section code that accesses shared resource that must complete w/o interruption!

can be simple like above, can also be complex. Sometimes tricky

Thread 1

```
int j = i;
```

Thread 2

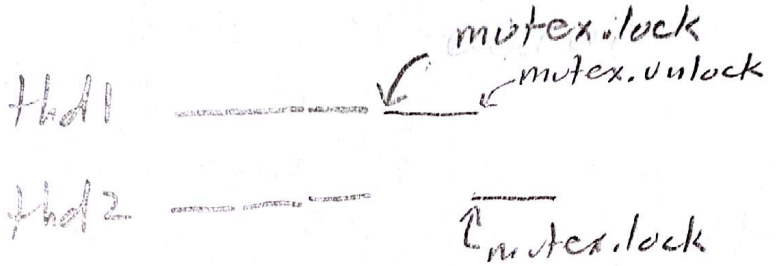
```
int k = i
```

// global
int i

if all you do is read a variable then no critical section, no need to protect. The first time you write the var, even if 100 reads & 1 write, then all 101 operations are critical & must be protected.

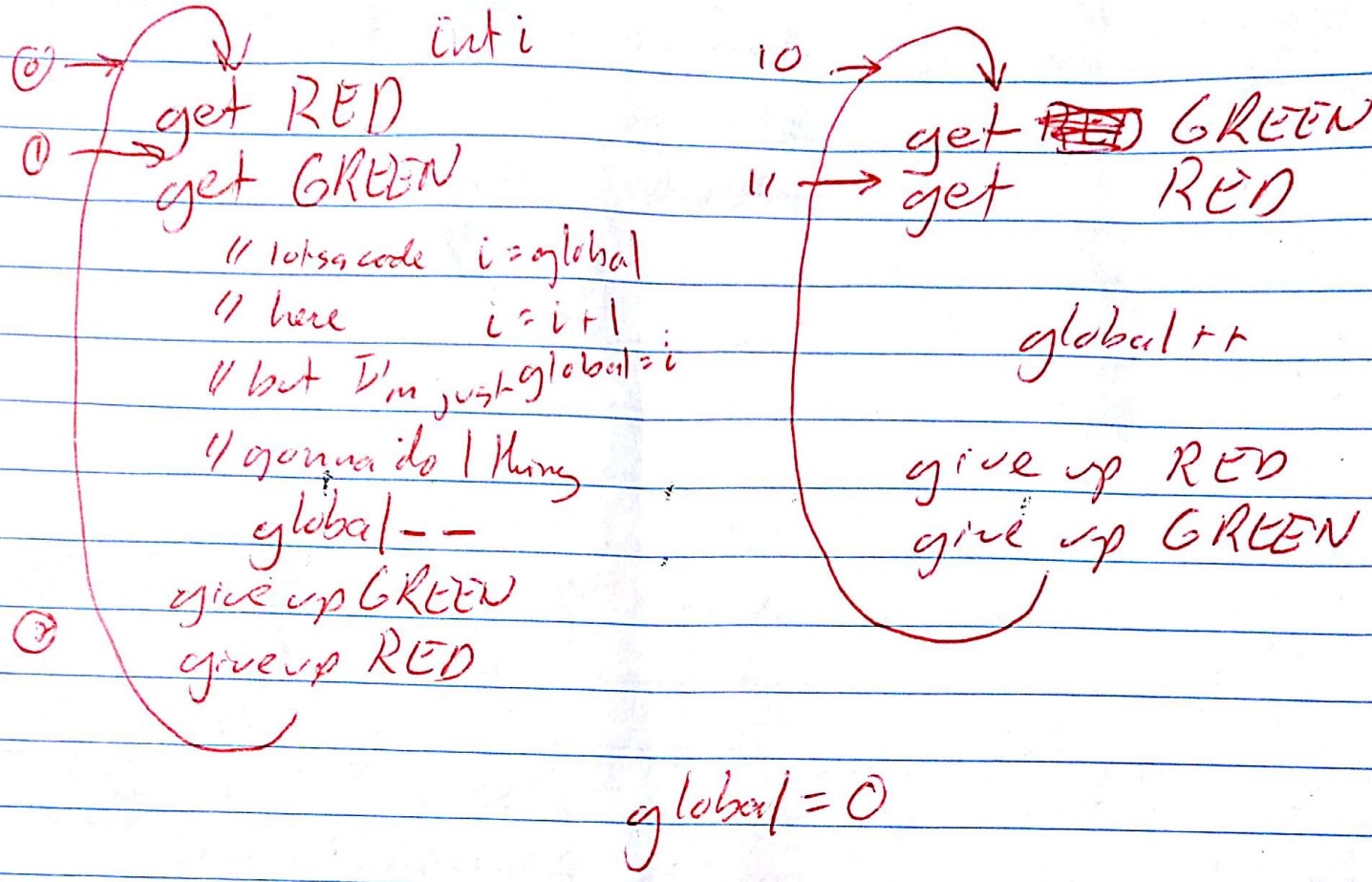
④ Mutual Exclusion ← mutex - traffic cop - one at a time. (solve previous prob using mutex.

But... No free lunch, performance suffers.



notice how thrds slow down. Go from 2 thds running simultaneously to 1 at a time. 50% reduction in utilization

⑤ Deadlock - 2 or more processes are waiting because each has something the other wants.



global != 0

- have it work a few times

- stop T_DEC @ (1) start T_inc @ (10)
  ~~stop~~ can't get red.

← go to 8.

Livelock

2 people in corridor each move to the side so other can pass. But end up swaying back & forth with no forward progress.

Diff to livelock - processes that constantly change w regard to 1 another but do no work. Can occur in some Deadlock avoidance algorithms. If more than 1 process takes action. Avoided if only 1 process (random or priority) takes action

(6) · Starvation - runnable process or thread is overlooked by scheduler. it is never chosen.

A priority 1
B 2
C 2

real world have priority queues & scheduler chooses from them. EX pure priority on uniprocessor.

P1 queue – A
P2 queue – B – C

scheduler always chooses these. A will not run until B & C done.