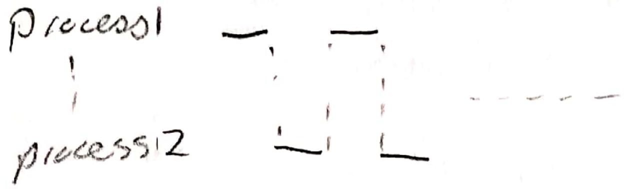
Overview

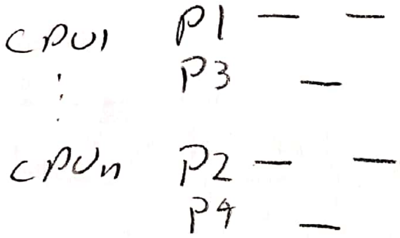
①

① Multiprogramming - multiple processes - single CPU



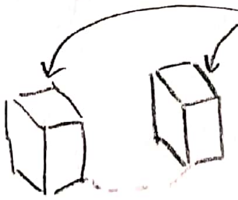
processor time
one at a time

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: multiple machines, multiple processors(co
Multiprocessing multiple processes

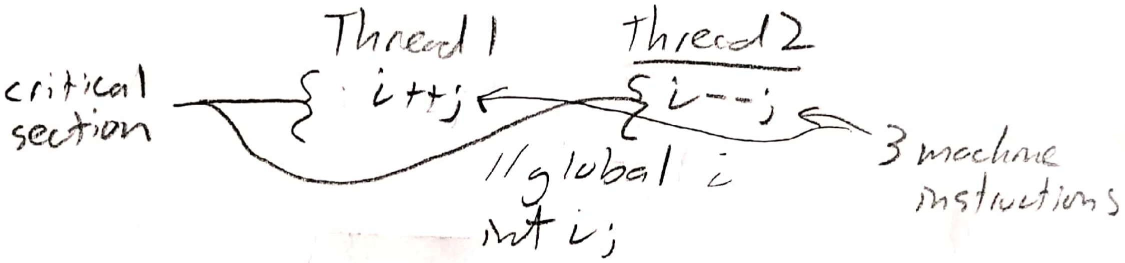


each machine running processes from same application

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 "Problems that threads can cause" for example race condition}



```
Thread 1        Thread 2
{ i++;          { i--;

//global i
int i;
```

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
```

skip if done already

```
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.

→ are single line only!

if you have this? 3 lines must complete. Cannot use atomics!

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

```
int func(int i){
    return i+27;
}
```

replacehere

is this func part of critical section? what if used as

```
int j = func(3);
```

still critical? the global makes it risky & globals are fine if single threaded

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	Thread 2
int j = i;	int k = i

```
//global
int i
```

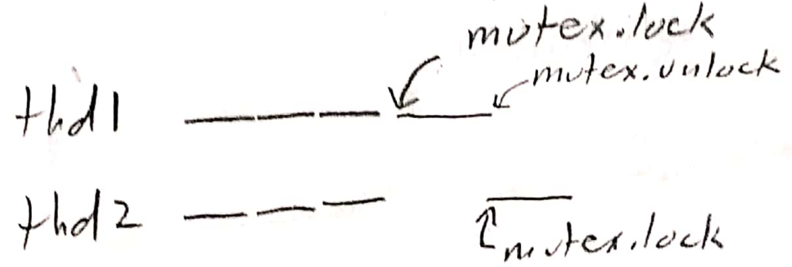
if all you do is read a variable then no critical section, no need to protect. The first time you write the var, even if 100 reads & 1 write, then all 101 operations are critical & must be protected.

4

mutex

utual Exclusion - traffic cop - one at a time. (solve previous prob using mutex.

But... No free lunch, performance suffers.



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

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

```
   Thread 1                     Thread 2

   get Red                      get Green
①  get Green                 ②  get Red
      ⋮                            ⋮
   give up Green                give up Red
   give up Red                  give up Green
```

Red & Green are global synchronization objects that only 1 thread can hold @ a time.

Deadlocks if Thread1 stops at ① & Thread2 starts @ ②

rule of thumb: always aquire synchronization objects in same order. Reverse either (but not both) of top 2 statements above & no deadlock!