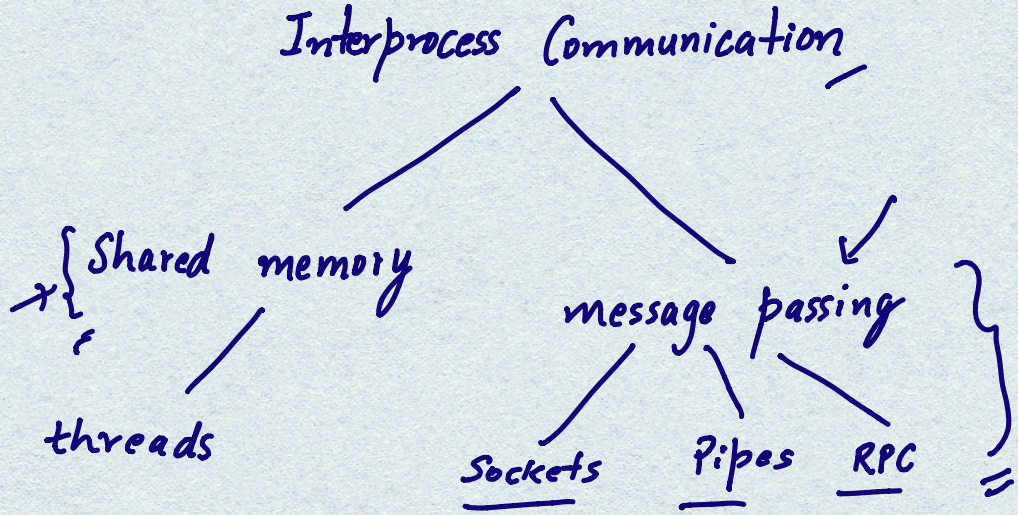
Process Synchronization



Challenges with shared memory

Concurrent access to shared data may lead to data inconsistency.

* Discuss various issues that can arise when shared data is accessed concurrently.
* Also, discuss solutions to these issues.

Race Condition

* We will now look at how Concurrent access to Shared data can lead to data inconsistency.

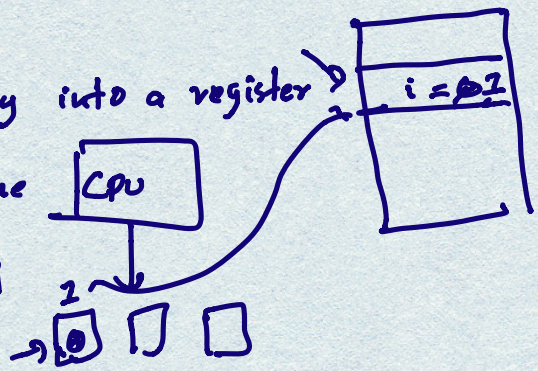
```
{ int i = 0;
Void increment() {
    i++;
}
int main() {
    thread t1, t2;
    t1(increment);
    t2(increment);
    Print(i);
}
```

i++ → critical

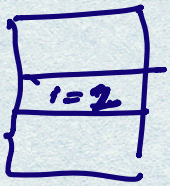
i++;
i = i + 4

i++

i) get the Current value and copy into a register
ii) add one to the register value
iii) write back to main memory



	T1	T2
t1	get i (0)	
t2	increment i (0→1)	
t3	write back i (1)	
t4	- — — .	get i (1)
t5	- — — .	increment i (1→2)
t6	. — -	write back i (2)



	→ T1	→ T2
t1	get i (0)	→get i (0)
t2	increment i (0→1)	. . . .
t3	. . . .	increment i (0→1)
t4	write back i (1)	- - .
t5	—— - .	write back i (1)

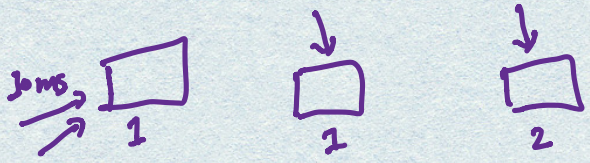
⇒ (1)

Race Condition

When several threads tries to access and manipulate the same data concurrently and the outcome of execution depends on the order in which access takes place these type of situations are called race conditions.

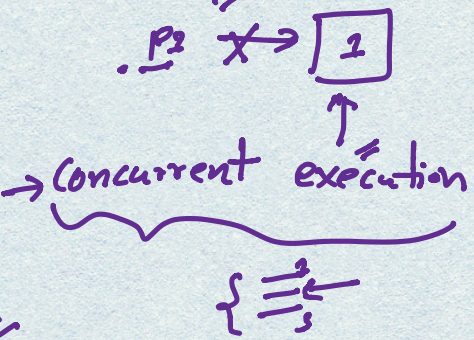
* Each thread is racing for the access to the shared data.

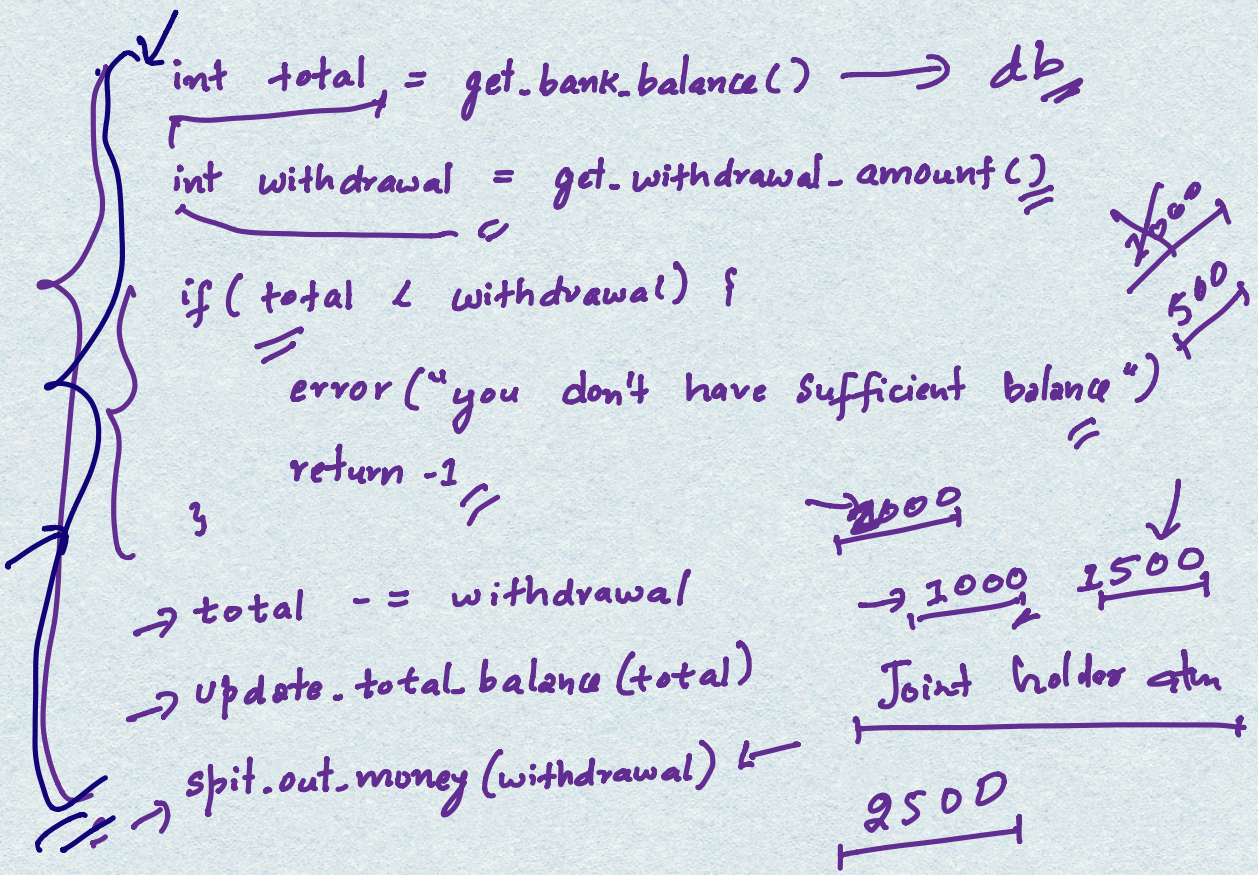
* Note the difference between Concurrent and parallel execution.

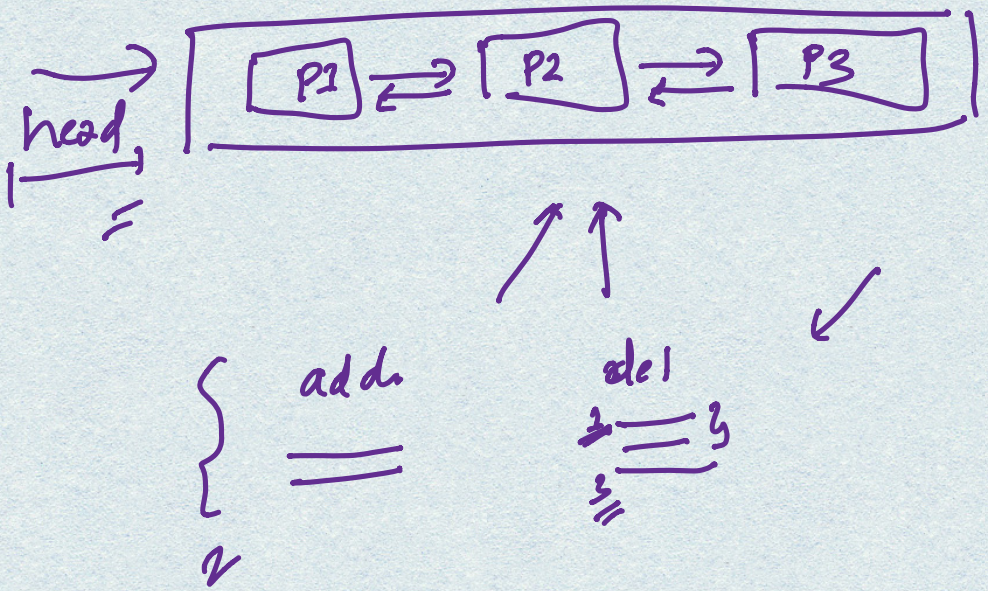


Different ways race conditions can happen:

- Interrupts
- Kernel preemption
- A task in kernel sleeps
- Parallel processing → symmetrical multiprocessing




int total = get_bank_balance() → db
int withdrawal = get_withdrawal_amount()
if (total < withdrawal) {
error ("you don't have sufficient balance")
return -1
}
total -= withdrawal
update_total_balance (total)
spit_out_money (withdrawal)
2000
1000
1500
Joint holder atm
2500


P1
P2
P3
head
add
del

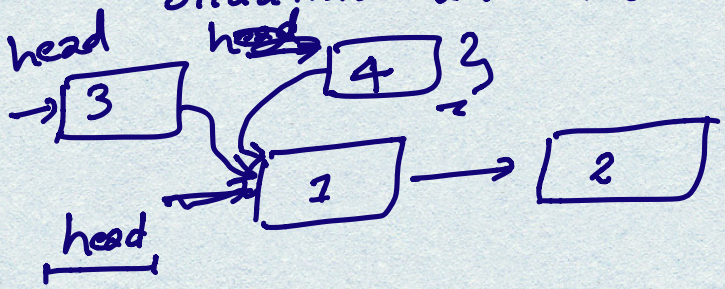
Critical Section Problem and its solutions

- Assume we have n processes or threads.
- And a Common Segment of code shared between each of these processes.

If multiple processes tries to execute this Common section of code and it leads to data inconsistency.

Common Section of code → Critical Section

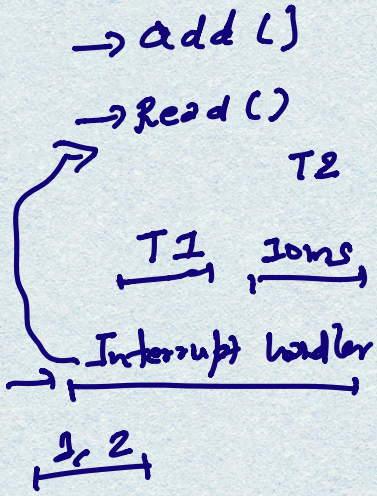
Situation mentioned above → Race Condition



newnode = create()

newnode → next = head

head = newnode



Ask yourself these questions before deciding if your code can lead to race condition or not.

- Is the data global ? Can other threads access it ?
- Is the data shared between process context and interrupt context ?
- If the process is preempted while accessing this data, can the newly scheduled process access it ?
- What happens if this function is called on another processor ?
- Can the current process sleep on anything ? If it does, in what state that leave the shared data ?

T1

P1 `g = g` P2

`if (g > 10) {` `g = g - 2`

f1 → 11

f2 → g

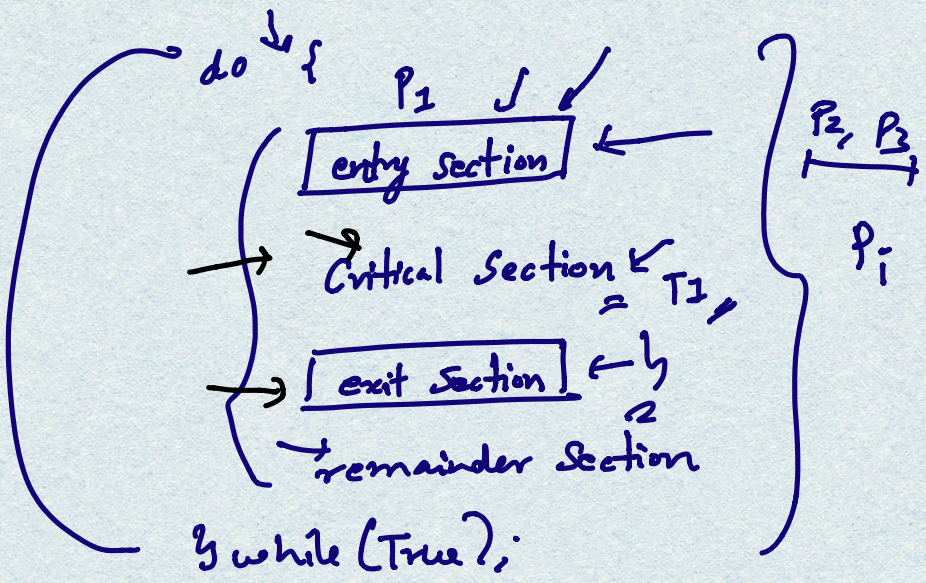
write (f2, g)

write (f2, g)

}

Critical Section Problem

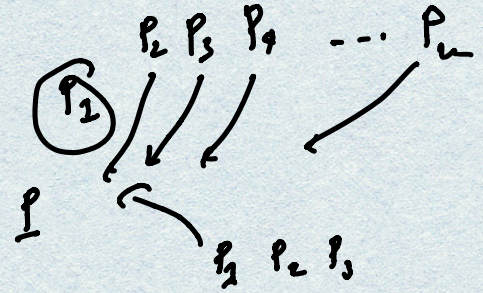
The critical section problem is to design a protocol that the processes can use to cooperate or that doesn't lead to race condition.



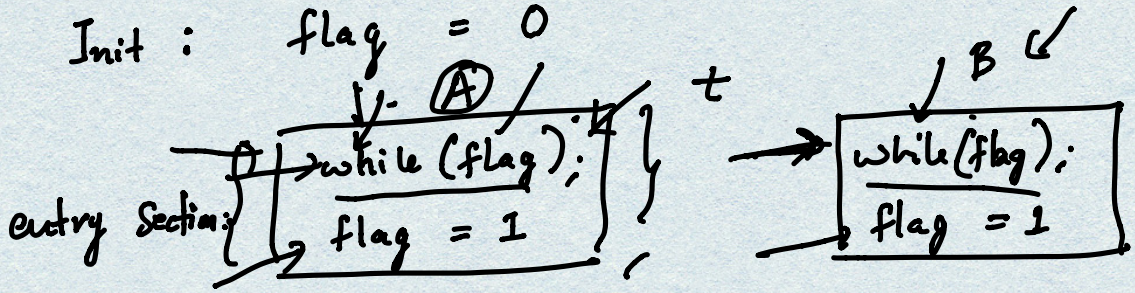

```
main()  {
   T1, T2,
}
```

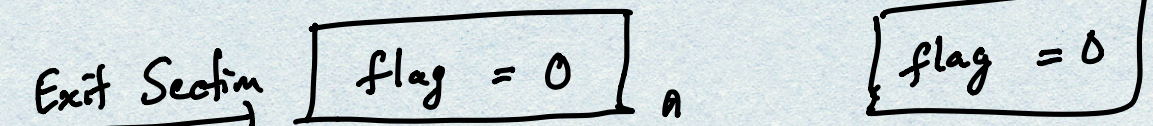
Requirements that a solution to CS problem must satisfy :—

i) Mutual exclusion
ii) Progress
iii) Bounded waiting



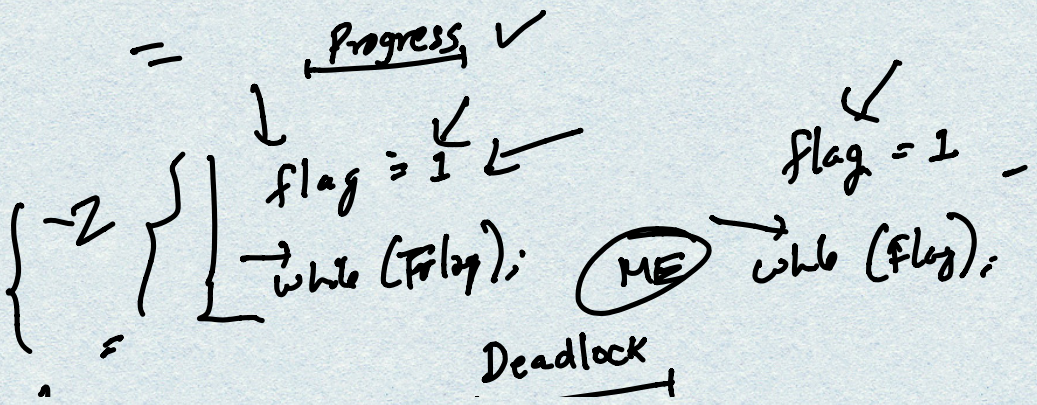
Init : flag = 0





ME ✗

Progress ✓



Init: FlagA = FlagB = False

A

Entry: while (FlagB);
FlagA = True

CS

Exit: FlagA = False

B

while (FlagA);
FlagB = True

CS

FlagB = False

→ ME ✗
→ Progress ✓

A

Entry: FlagA = True
while (FlagB);

t1

B

FlagB = True
while (FlagA);

Deadlock

ME
FlagA = False

Bounded waiting

Turn taking

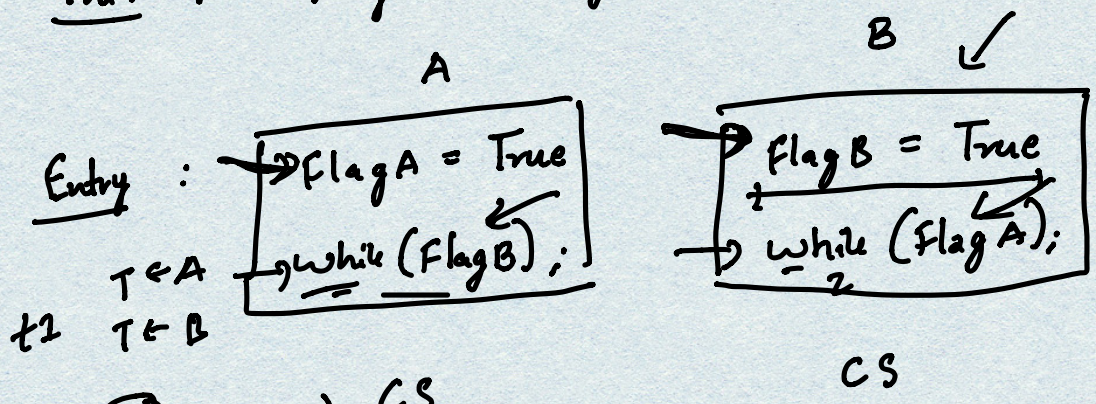
Peterson's Solution for Critical Section Problem

① Using two lock variables

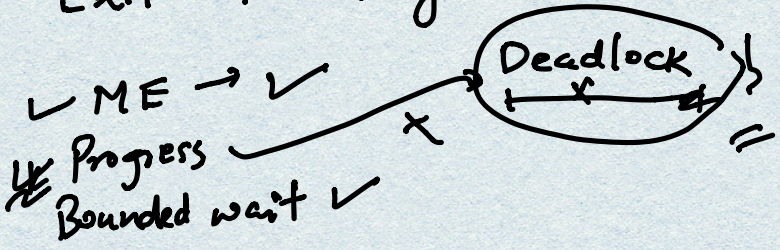
② Turn-taking approach

Two lock Variables

init : FlagA = FlagB = False



Exit : FlagA = False FlagB = False



Turn Taking approach

Deadlock ✗

init: Turn = A // B

	A	B
Entry:	while (turn != A);	while (turn != B);
	turn = A	turn = B
	CS	CS
Exit:	turn = B	turn = A

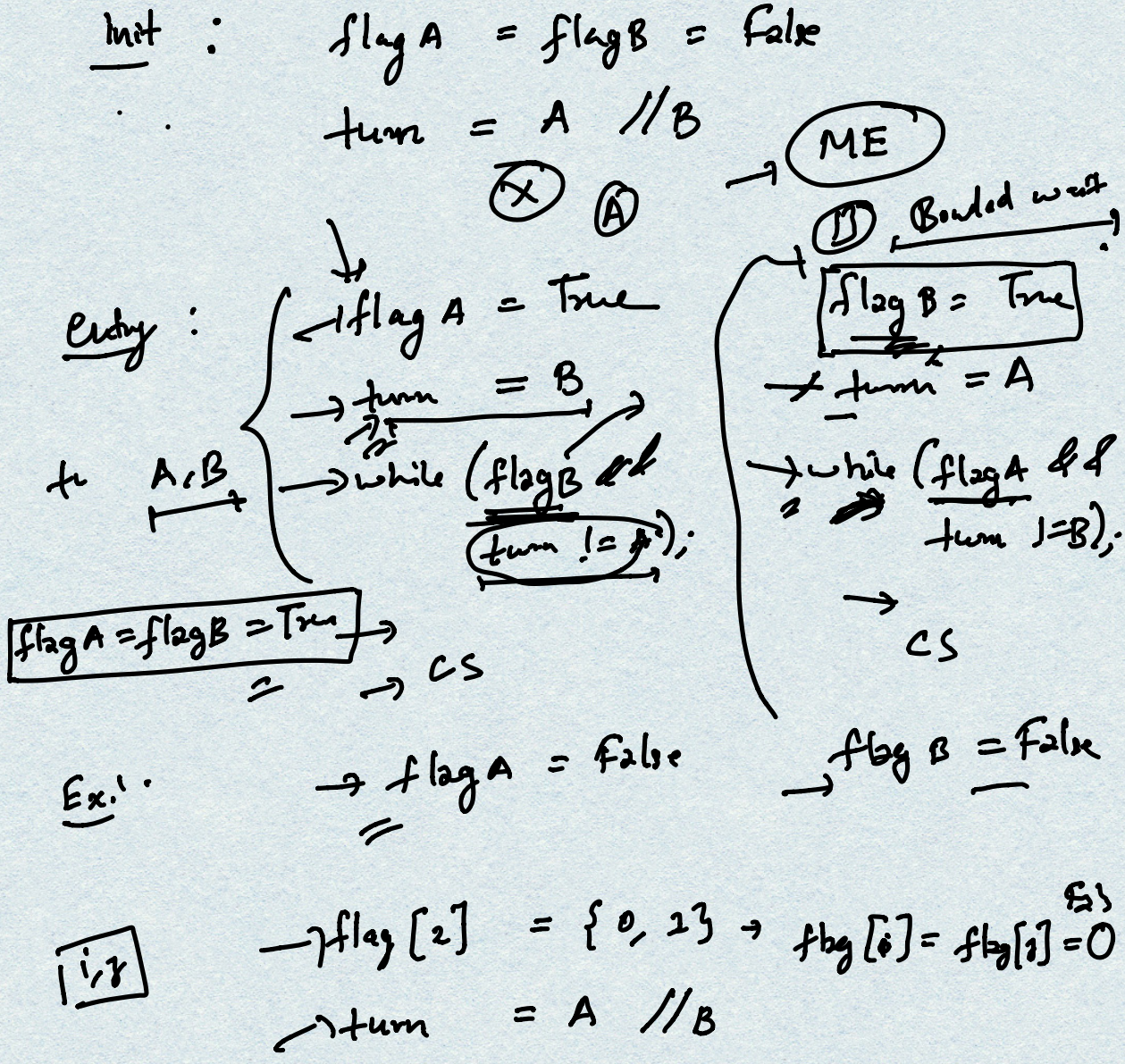
ME ✓

Progress ✓

Bounded wait ✗

A, B

	turn = A	turn = B
	while (turn != A);	while (turn != B);



i, j

flag [2] = {0, 1} → flag [i] = flag [j] = 0

turn = A // B

Limitations of Peterson's Solution :-

1. Spinlock - wastes cpu cycles (N)
2. Doesn't work with unknown number of processes.
3. Doesn't work well on modern operating systems which have multiple cores.

Init : flag[i] = flag[j] = False
turn = j // i

Entry : A → flag[i] = True → store
turn = j → load
→ while (flag[j] && turn == j);

B → flag[j] = True → store
turn = i
while (flag[i] && turn == i); ← load

Critical Section

Critical Section

Exit : flag[i] = False

flag[j] = False

Load → it loads the value from main memory to CPU registers

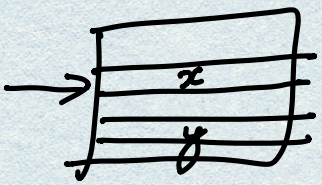
Store → it stores the value from CPU registers to main memory.

1 → x = 0 ← store

2 → if (x == 0) { ← Load

① x = 0 St

② if (y == 0) { → Load



A

turn = y

while (flag[y] && turn == y);

flag[i] = True

Store

CS

B

turn = i

while (flag[i] && turn == i);

flag[y] = True

CS