Recall

Section of A: x=x+1

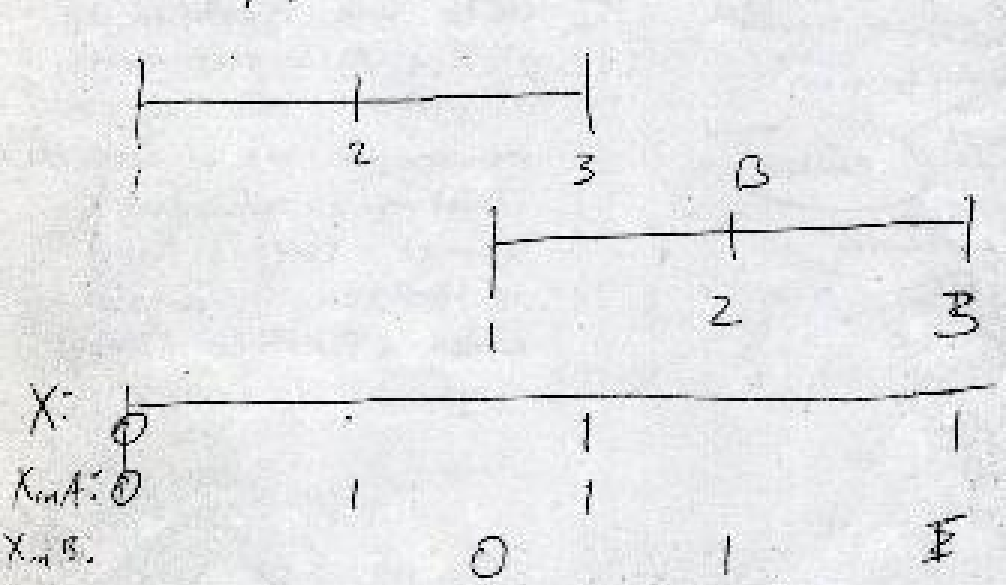
1 Load x
2 Add x, 1
3 Store x

x shared

Section of B: x=x+1

Load x 1
Add x, 1 2
Store x 3

A,B = x := x+1 (write, read, modify)



Clearly, A & B must not execute concurrently. This is the critical section prob.

How to force this sequentiality?

We need to "guard" the critical section.

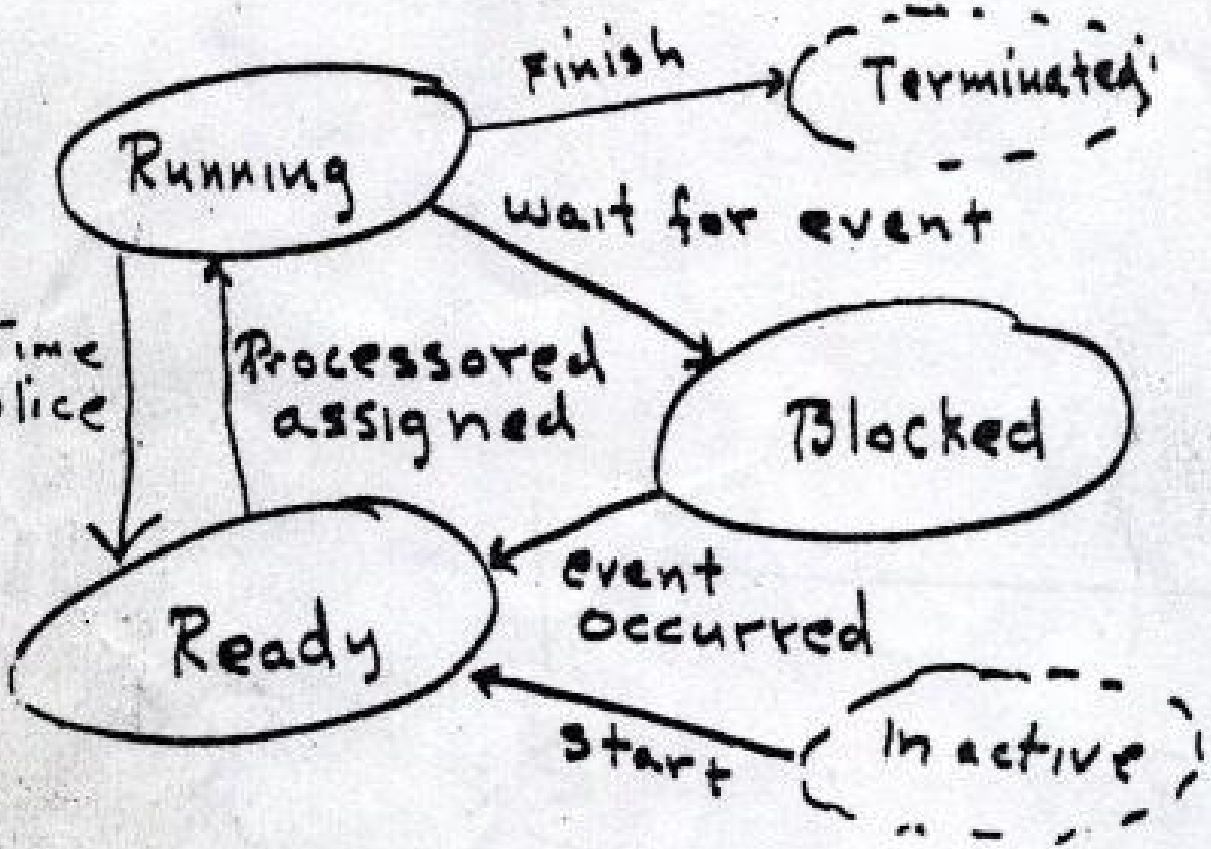
There must be mutual exclusion enforced.

State of a Process

Running: Instructions being executed

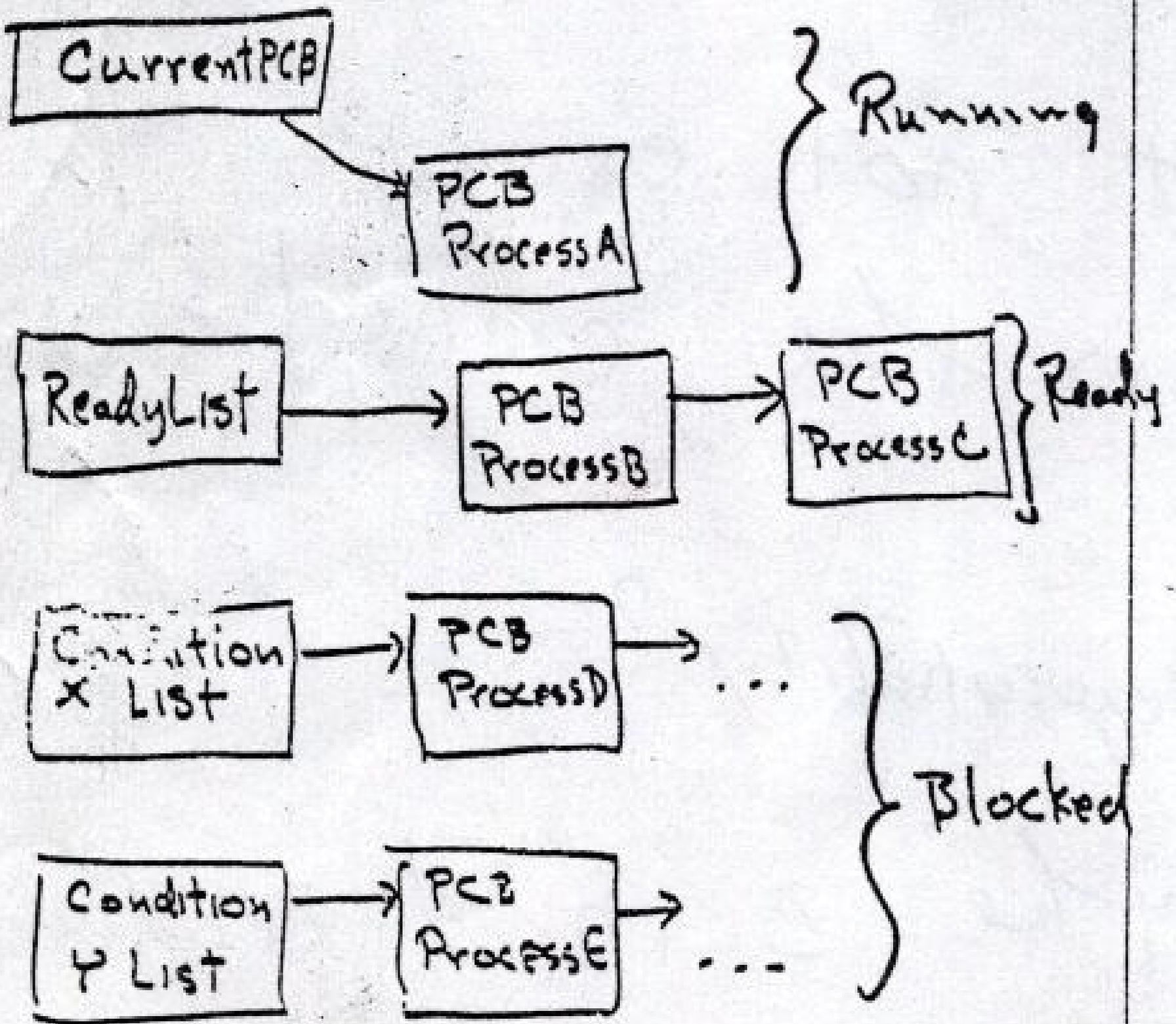
Blocked: The process is waiting for some event to occur.

Ready The process is active but not blocked & not Running.

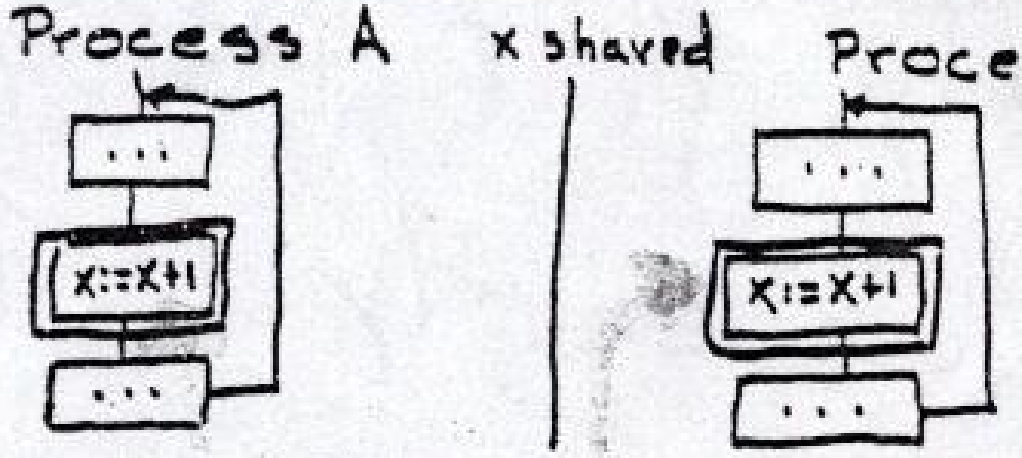


Every process has code to execute but that code does not identify the process. (several processes may be executing the same code)

A process and its state are identified by all the data that identify its progress. This may be memory locations or computer registers or secondary storage. This identity is focused on a structure called a Process Control Block (PCB).


CurrentPCB
PCB
Process A
Running
ReadyList
PCB
Process B
PCB
Process C
Ready
Condition
X List
PCB
Process D
...
Condition
Y List
PCB
Process E
...
Blocked

Critical-Section Problem (5.2)



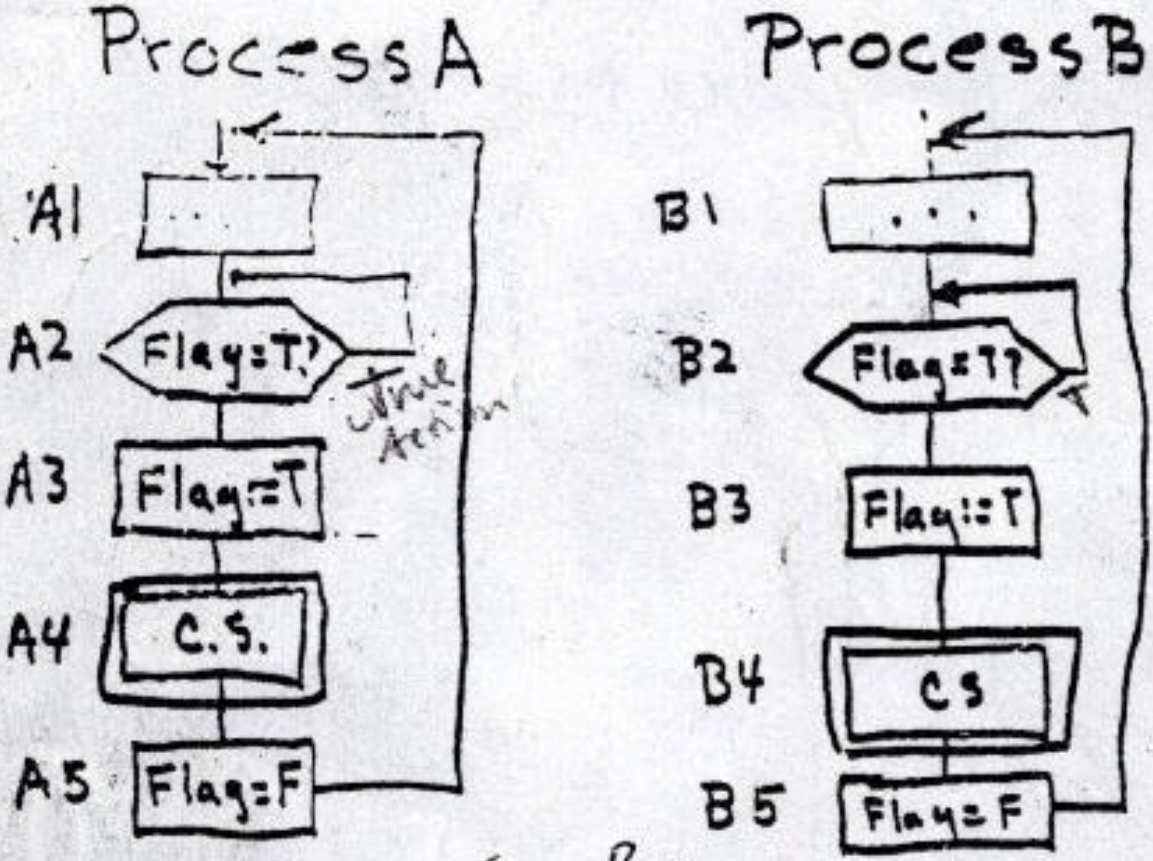
Trouble

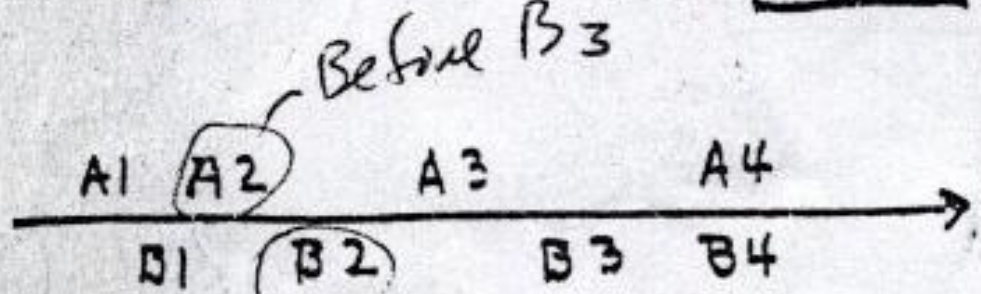
Let the blue box ▭ enclose the area of trouble, i.e. the place of sharing. Call this a Critical Section of execution. If these sections are not concurrent then at least this part of the program would behave in a predictable way.

Dijkstra's Examples

Flag Algorithm

ProcessA & ProcessB share a common variable Flag = False



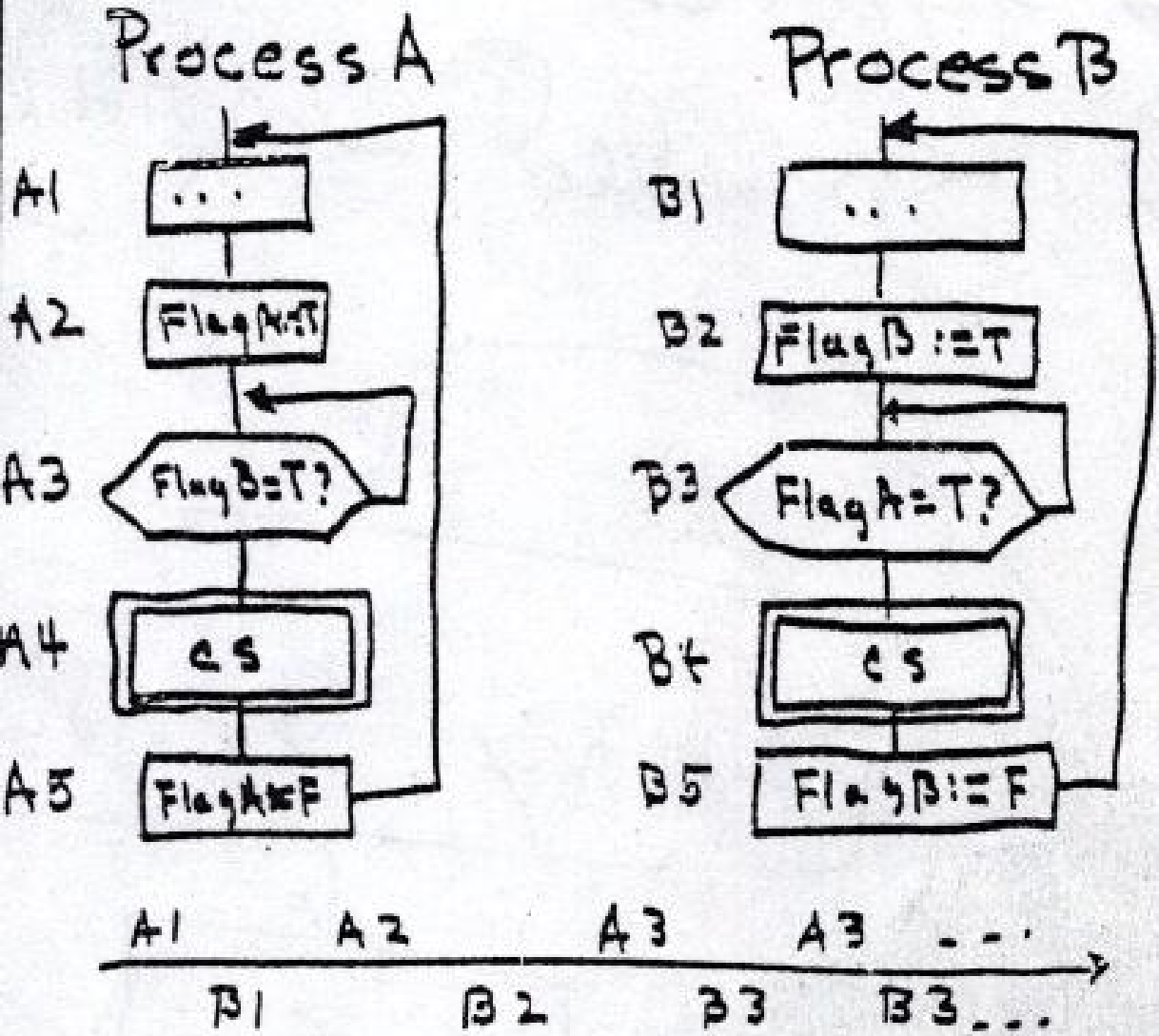


Not Mutual Excusion

before A3

Conservative Flag Algorithm

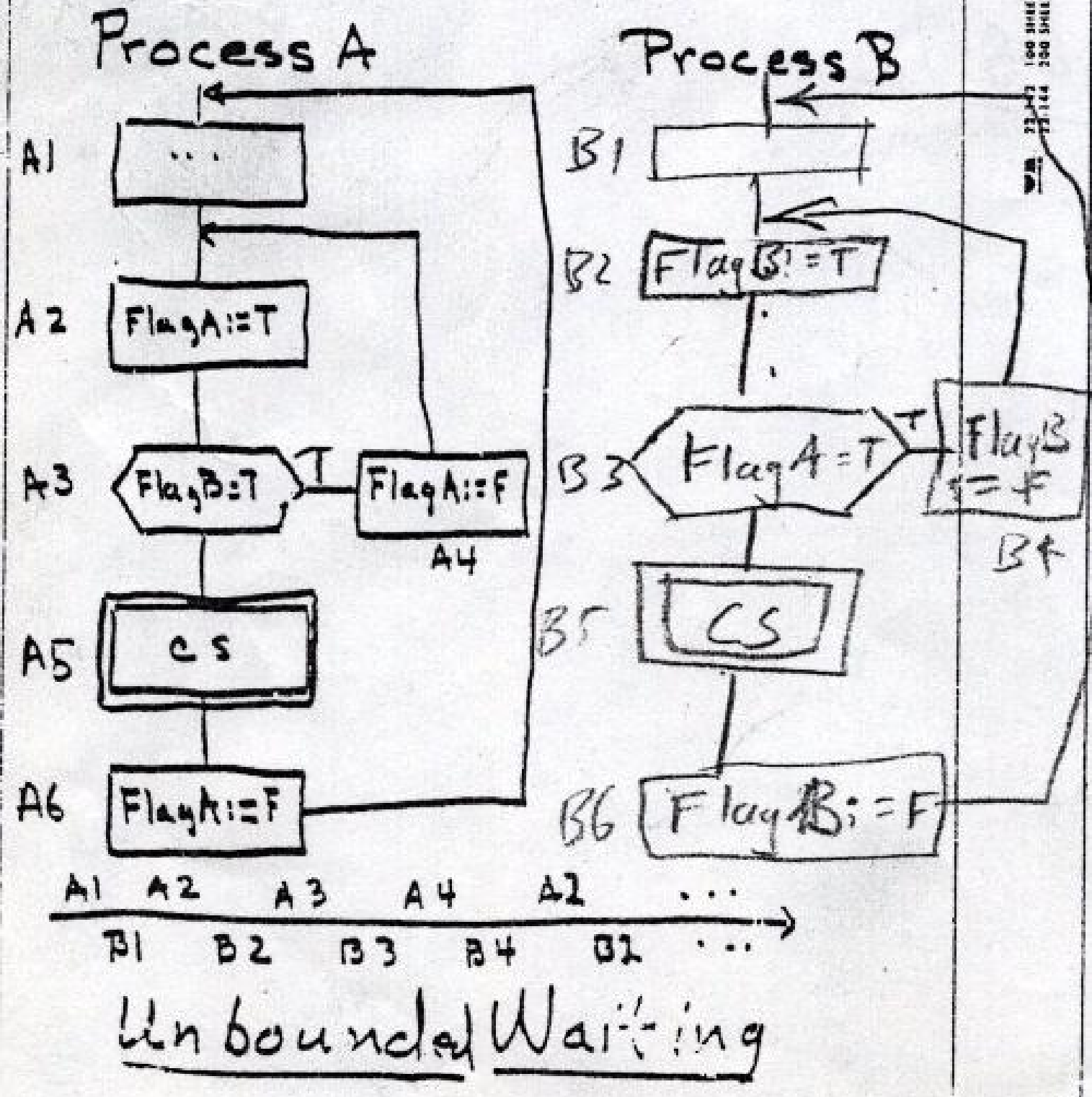
ProcessA & ProcessB have their own FlagA & FlagB which they set respectively and which the other may read (= False).



Deadlock - Unbounded Waiting

Try Try Again Algorithm

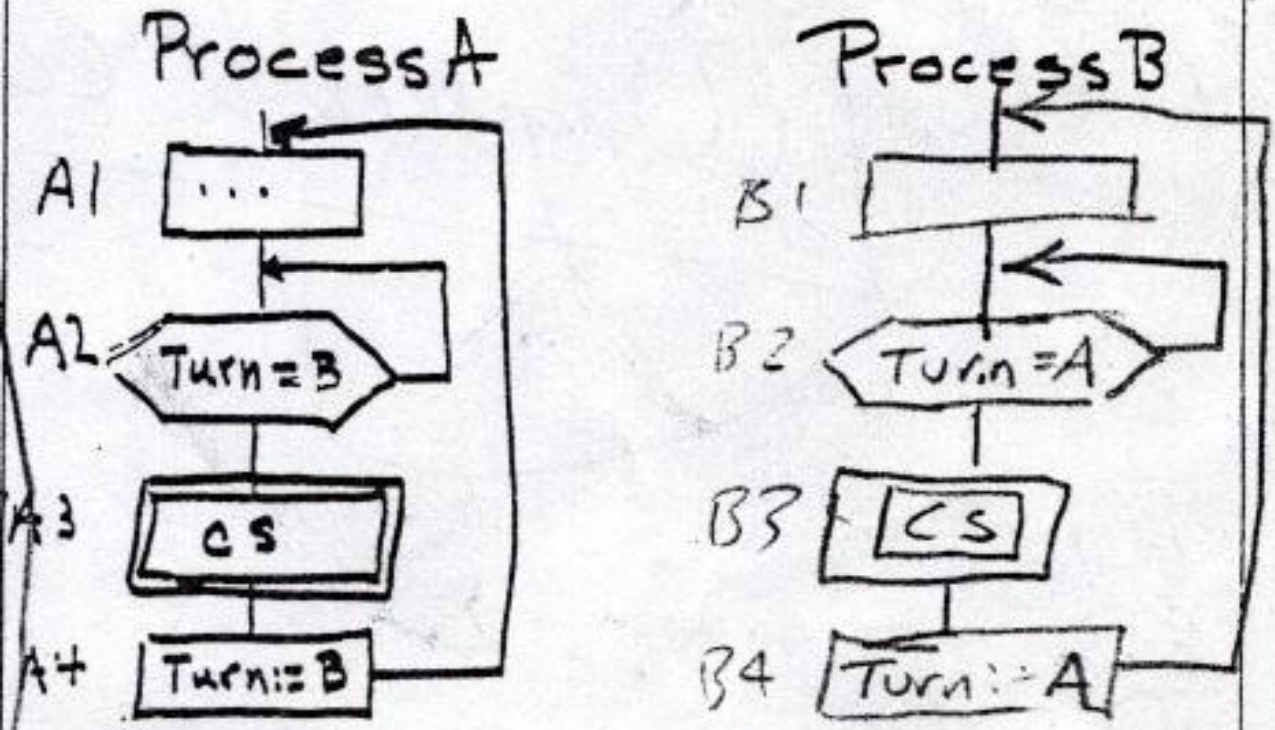
ProcessA, ProcessB, FlagA, FlagB as before



Unbounded Waiting

Take Turns Algorithm

ProcessA, ProcessB, shared variable Turn = A.



A1 A2 A3 A4 A1 A2 A2...

B1

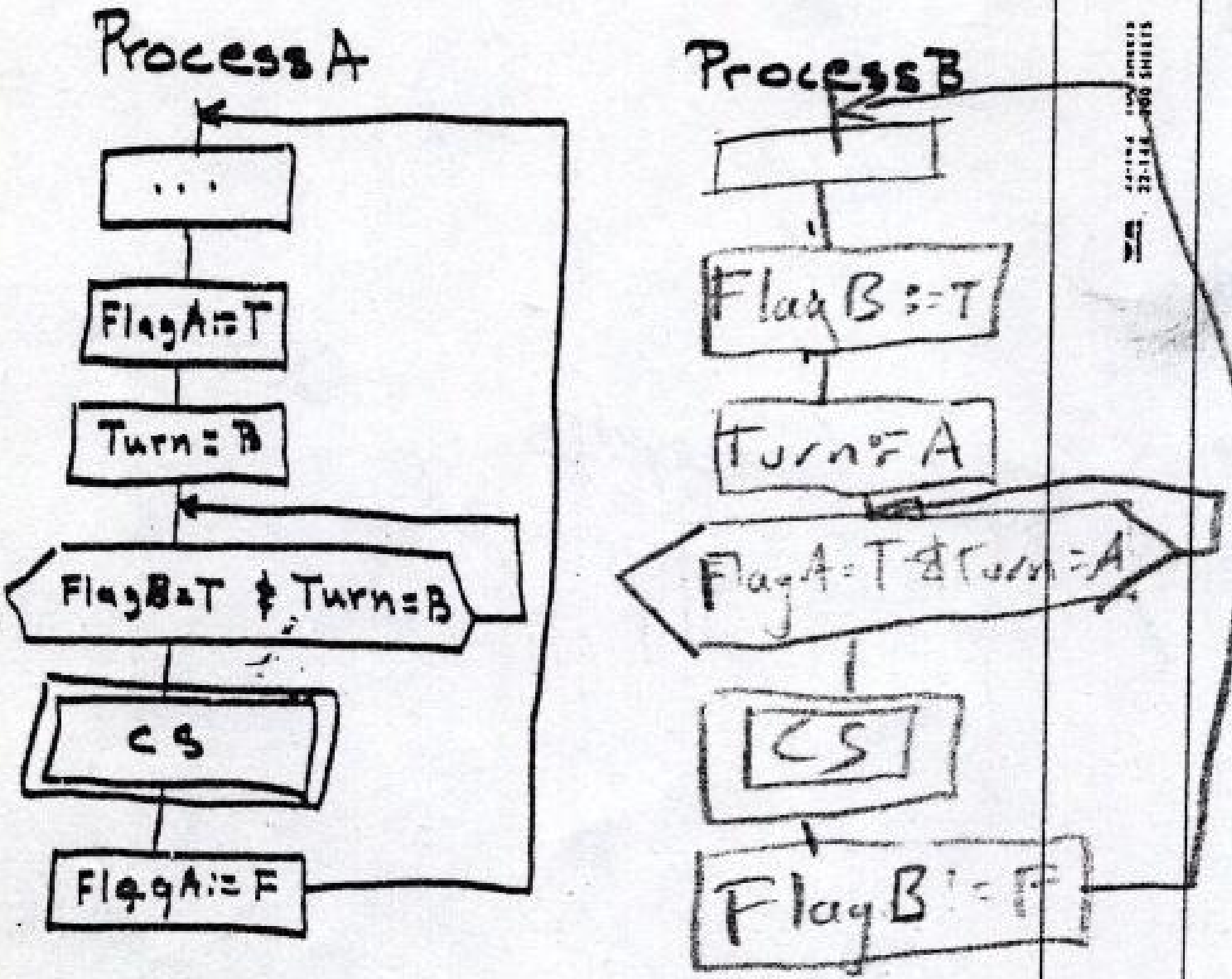
If B1 takes a long time...

No Progress

i.e. enforcing alternating critical section access forces A to wait for B to do its

Peterson's Algorithm

Process A, Process B, Flag A, Flag B, Turn as before



Key: Last one in set Turn to the other Process.

One Idea

Bakery Algorithm (Informal)

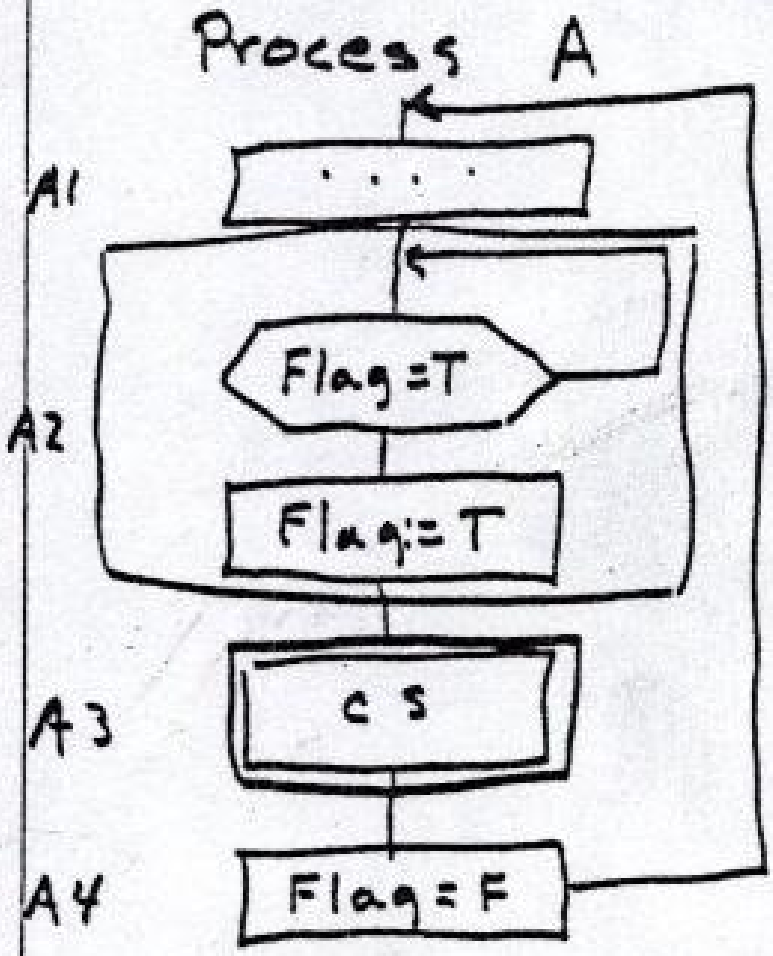
Each process takes a numbered ticket at least one larger than every ticket held by ~~another~~ processes (This may result in ties) whenever it wishes to enter the critical section. As soon as no process is choosing a ticket, the process with the lowest ticket number can enter the critical section. (Ties are broken arbitrarily) The process throws away it's ticket when leaving.

The Bad Apple Principle

One bad timing is enough to ruin a critical section solution.

We can make our life easier by reducing the number of timings. We do this by combining events. We call this making the operations indivisible. Consider the Flag Algorithm. Make the testing of the Flag, the setting of the Flag, and the loop back a single machine instruction (called TestAndSet)

(must be atomic

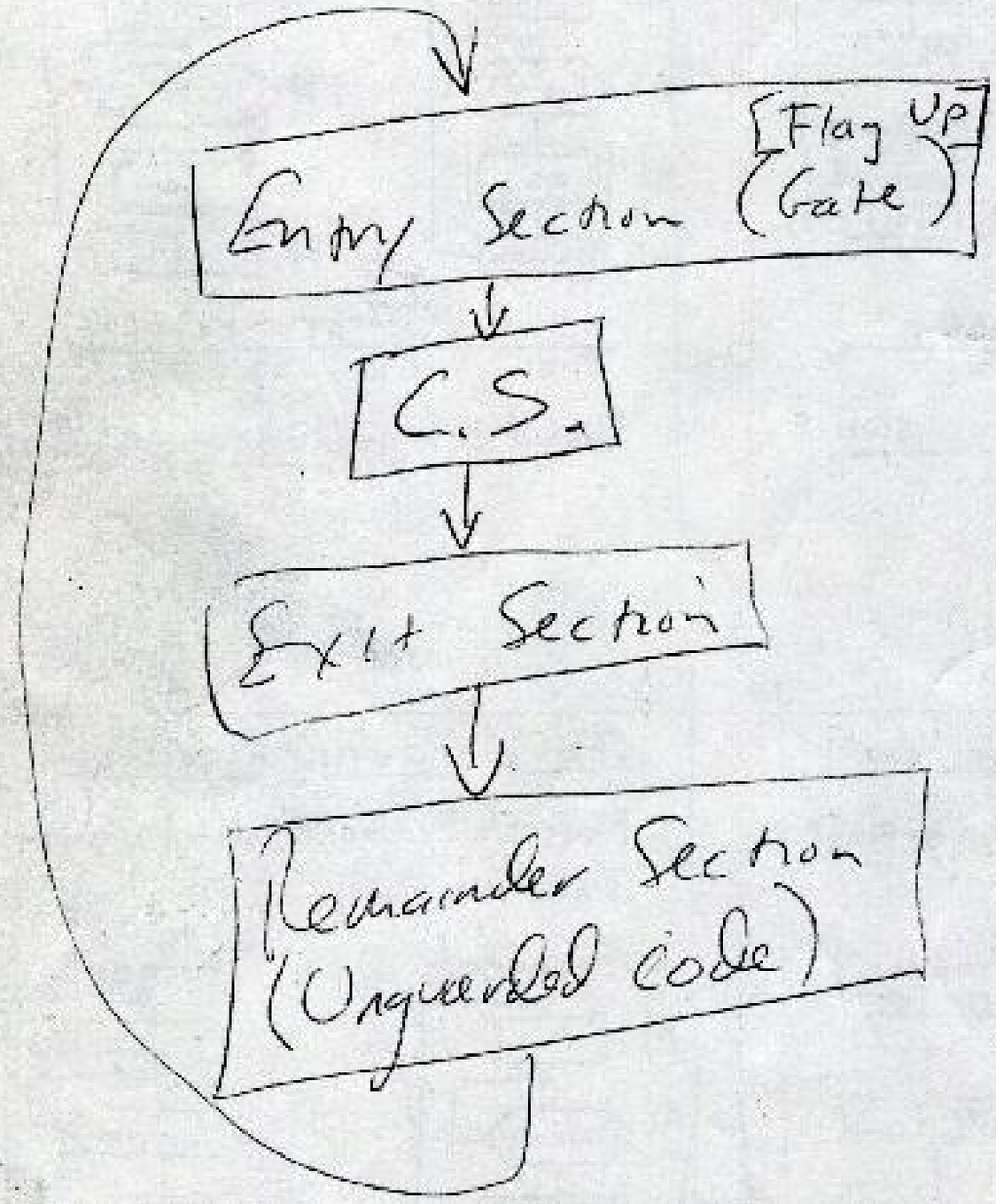


This solution now provides Mutual Exclusion. It may still allow UnBounded Waiting.

Operations may be made atomic (indivisible) by turning off interrupts. (in a single processor configuration)

Requirements for a critical section:

1. Mutual Exclusion
2. Progress
3. Bounded Waiting (Each process gets its chance)



Semaphores

The solutions to the C.S. problem discussed before involve Busy Waiting i.e. using the processor to poll a flag when no work can be done. Dijkstra saw that this was not appropriate in an operating system context and proposed a higher level construct based on the idea that an operating system can put a process to sleep by not scheduling it.

A Semaphore S is an integer variable together with two indivisible operations:

P(s) wait

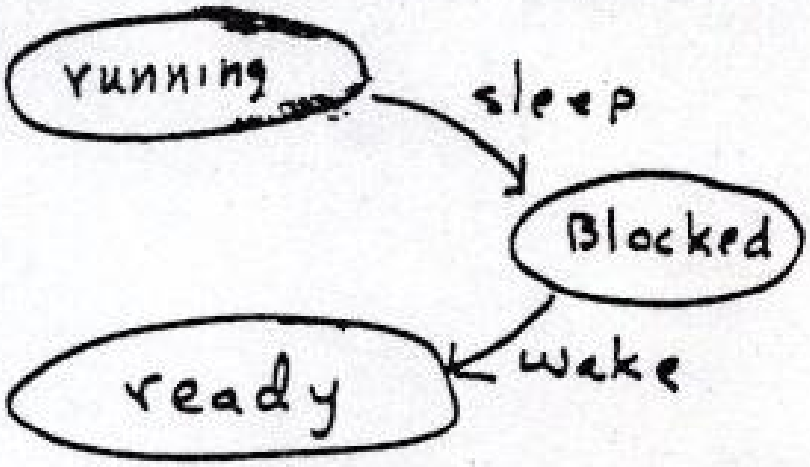
V(s) signal

P(s): S := S - 1

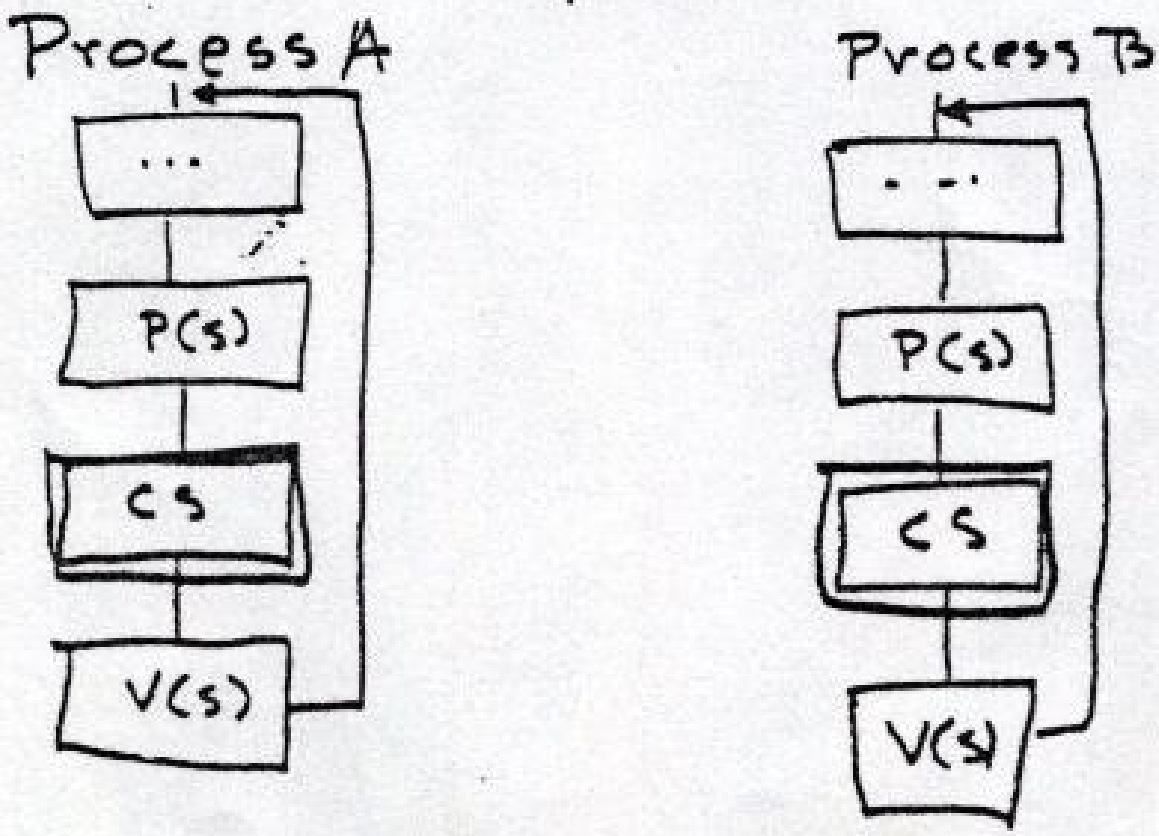
if (S < 0) sleep

V(s): S := S + 1

if (S ≤ 0) wake a process



Dijkstra did not specify the implementation but normally we associate a separate queue with each Semaphore where the PCB of the blocked process is parked till it is waked.



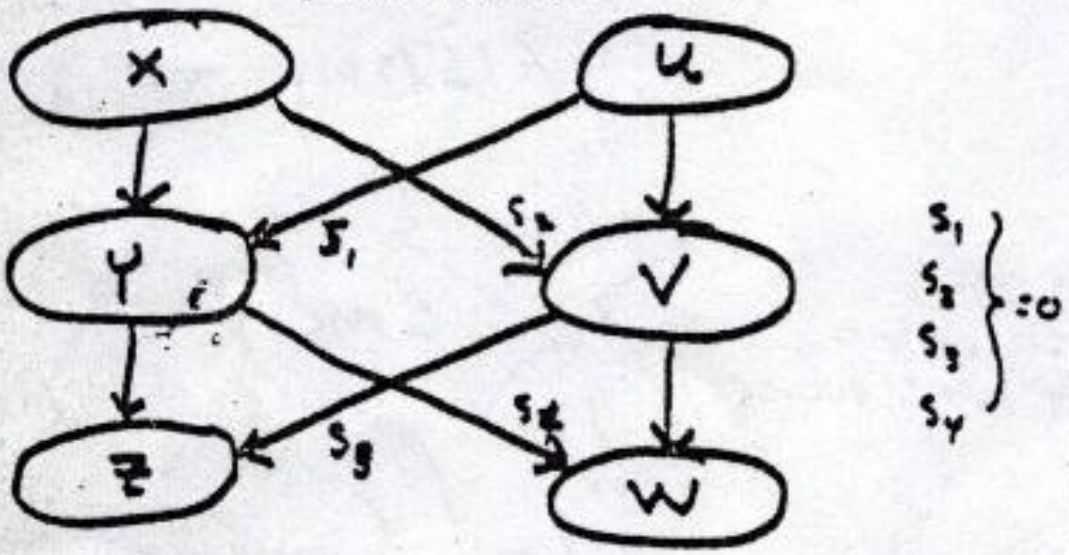
The C.S. problem has been solved with Semaphores.

1. The solution is simple and correct (almost by definition).
2. The code is linear without alternatives as far as the computational effects of this process
3. All processes (more than two are allowed) are the same and they need not know about each other.
4. There is no busy-waiting.

Semaphore have 3 modes of common usage:

1. Critical Section Guards (s=1)
2. Precedence Enforcers (s=0)
3. Resource Counters (s=n)

Precedence Graph



Process A

X
V(S_2)
P(S_1)
Y
V(S_4)
P(S_3)
Z

Process B

U
V(S_1)
P(S_2)
V
V(S_3)
P(S_4)
W