Operating Systems 2024/2025

T Class 04 – Synchronization

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

operating system
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

Abbreviation: OS

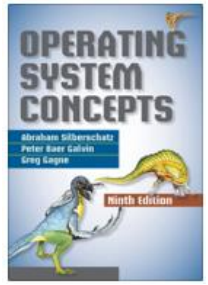
Source: Dictionary.com

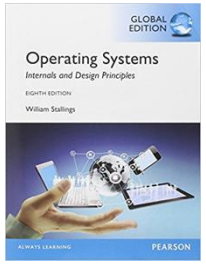
1290 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
dei departamento de engenharia informática

Updated on
22 February 2025

Disclaimer

- This slides and notes are based on the companion material [Silberschatz13]. The original material can be found at:
  - http://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/
- In some cases, material from [Stallings15] may also be used. The original material can be found at:
  - http://williamstallings.com/OS/OS5e.html
  - http://williamstallings.com/OperatingSystems/
- These slides also use materials from [McHoes2011] "Understanding Operating Systems, 6th Edition, Ann McIver McHoes and Ida M. Flynn"
- The respective copyrights belong to their owners.





Note: Some slides are also based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva (Operating Systems classes of DEI-FCTUC).

Synchronization
Outlines

- Multiprocessing
- Concurrency and synchronization
  - Critical region
  - Semaphores
  - Classical problems

MULTIPROCESSING

From one to multiple CPUs

- Multiprogramming
  - One CPU shared by several jobs or processes
- Multiprocessing
  - Several processors working together
    - Includes:
      - single computers with multiple cores
      - linked computing systems that share processing

Parallel Processing

- One form of multiprocessing
  - When >=2 processors work together
- Benefits
  - Increase in reliability
    - If one processor fails, then the others can continue to operate and absorb the load – needs careful design!
  - Faster Processing
    - Instructions can be executed in parallel

Parallel Processing (2)

- More flexibility => More complexity:
  - How can processor interact?
  - How to orchestrate interaction?

Example

Fast food drive-through lunch stop

Originator	Action	Receiver
Processor 1 (the order clerk)	Accepts the query, checks for errors, and passes the request on to =>	Processor 2 (the bagger)
Processor 2 (the bagger)	Searches the database for the required information (the hamburger)	
Processor 3 (the cook)	Retrieves the data from the database (the meat to cook for the hamburger) if it's kept off-line in secondary storage	
Processor 3 (the cook)	Once the data is gathered (the hamburger is cooked), it's placed where the receiver => can get it (in the hamburger bin)	Processor 2 (the bagger)
Processor 2 (the bagger)	Passes it on to =>	Processor 4 (the cashier)
Processor 4 (the cashier)	Routes the response (your order) back to the originator of the request =>	You

- 4 "processors" working together to deliver the order in 6 steps – synchronization is crucial!

Evolution of Multiprocessors

- Multiprocessing can occur in different levels, each with different synchronization needs

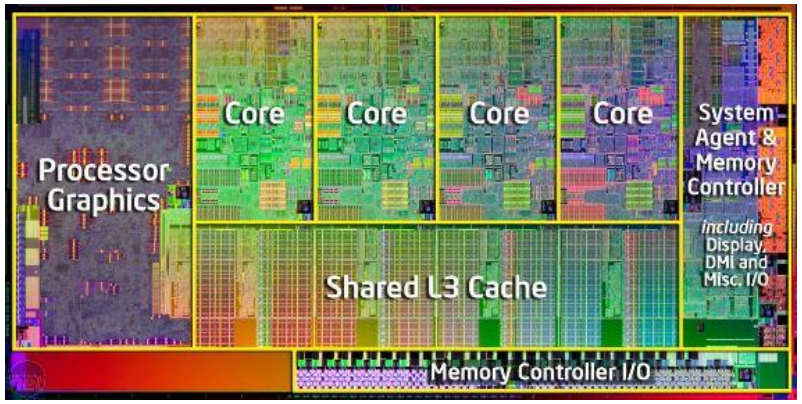
Parallelism Level	Process Assignments	Synchronization Required
Job level	Each job has its own processor and all processes and threads are run by that same processor.	No explicit synchronization required.
Process Level	Unrelated processes, regardless of job, are assigned to any available processor.	Moderate amount of synchronization required to track processes.
Thread Level	Threads are assigned to available processors.	High degree of synchronization required, often requiring explicit instructions from the programmer.

[McHoes2011]

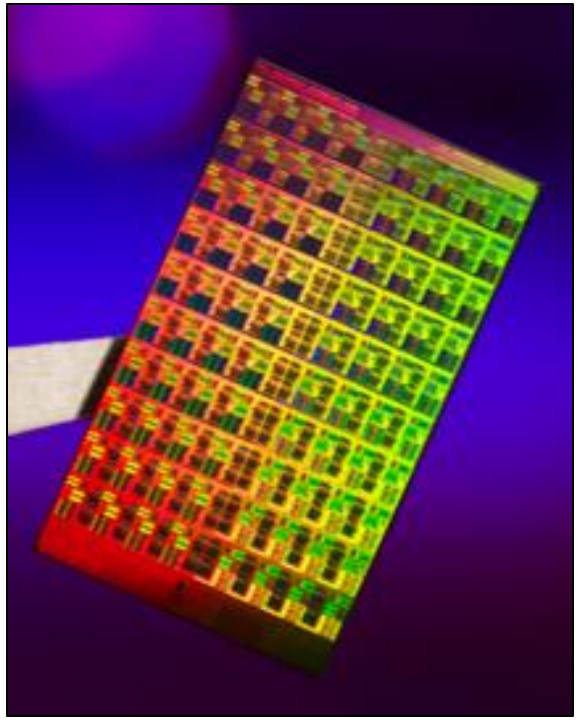
Multi-core Processors

- Multi-core processors have several processors on a single chip.
- Motivation
  - As processors are reduced in size and getting faster…
    - More heat
    - Tunneling effect (electrons “tunnel” from a transistor to another)
- Does this affect the OS? Yes!!
  - Manage multiple processors, RAMs, multiple tasks at once
- Is a multicore machine always faster than a single-core machine with the same clock speeds?
  - NO, depends on what is being computed (multi-threaded vs sequential tasks)

Multi-core Processors



Quad-core Intel Sandy Bridge



Intel – 80-core

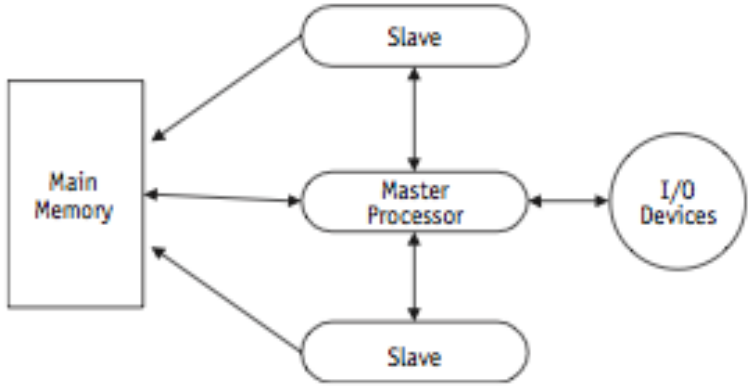
Typical Multiprocessing configurations

- Master/slave
- Loosely Coupled
- Symmetric

Multiprocessing configurations

Master/slave

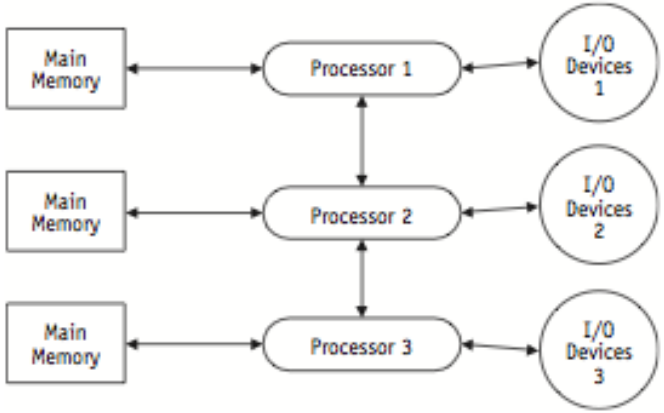
- Asymmetric
- One master processor controls slave processors
  - The master is responsible for managing the entire system (files, devices, processors, etc.)
- Well suited for environments where processing time is divided between backend and frontend processing



- Advantages
  - Simplicity
- Disadvantages
  - Poor reliability - if master fails, all fails
  - Poor resource use (slaves wait for master to get work)
  - Many interrupts as slaves interrupt master whenever they need OS intervention (e.g., I/O)

Multiprocessing configurations
Loosely Coupled

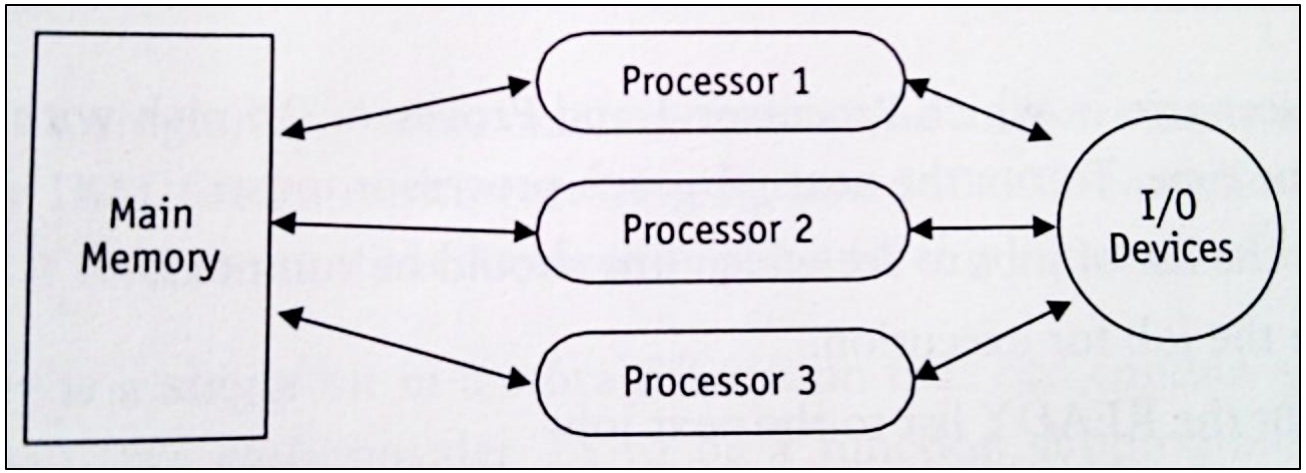
- Each processor controls its own resources: files, memory, I/O devices
- Not completely independent systems because processors communicate and cooperate with others
- Jobs stick to their assigned processor
  - Processor assignment to ensure well balanced use of resources
- Not prone to catastrophic failures as the failure of one processors does not imply the failure of the all system



Multiprocessing configurations

Symmetric (tightly coupled)

- More reliable than loosely coupled configurations, more effective using resources, good load balancing, degrades better in the event of a failures
- However,…
  - Prone to races and deadlocks
  - Requires **synchronization!**



CONCURRENCY & SYNCHRONIZATION

Applications of concurrent programming

- Example…

Sequential operations

$A = 3 * B * C + 4 / (D + E) ** (F - G)$

Step No.	Operation	Result
1	(F – G)	Store difference in T_1
2	(D + E)	Store sum in T_2
3	(T_2) ** (T_1)	Store power in T_1
4	4 / (T_1)	Store quotient in T_2
5	3 * B	Store product in T_1
6	(T_1) * C	Store product in T_1
7	(T_1) + (T_2)	Store sum in A

In parallel

Step No.	Processor	Operation	Result
1	1	3 * B	Store product in T_1
	2	(D + E)	Store sum in T_2
	3	(F – G)	Store difference in T_3
2	1	(T_1) * C	Store product in T_4
	2	(T_2) ** (T_3)	Store power in T_5
3	1	4 / (T_5)	Store quotient in T_1
4	1	(T_4) + (T_1)	Store sum in A

Applications of concurrent programming (2)

- Array operations:

```
for (j = 1; j <= 3; j++)
  a(j) = b(j) + c(j);
```

- Calculating array using 3 processors
  - Processor #1

```
a(1) = b(1) + c(1)
```

  - Processor #2

```
a(2) = b(2) + c(2)
```

  - Processor #3

```
a(3) = b(3) + c(3)
```

Applications of concurrent programming (3)

- Other applications
  - Database searches
  - Sorting and merging files
  - Graphical computing
  - …

Why synchronize?

Real life example

Buying milk - unsynchronized

	Roomate A	Roomate B
3:00	Arrive home: no milk	
3:05	Leave for store	
3:10	Arrive at store	Arrive home: no milk
3:15	Leave store	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Leave store
3:30		Arrive home: too much milk!

Copyright: Stanford 2012

The milk problem but with processors

- To select the next job to execute each processor must:
  - Consult the list of jobs to see which one should be run next
  - Retrieve the job for execution
  - Increment the READY list to the next job
  - Execute it

	Processor 1	Processor 2
1	Consult list	
2	Retrieve job 74	
3		Consult list
4		Retrieve job 74
5		Increment list (set 75 to top)
6	Increment list (set 76 to top)	
7	Execute job 74	
8		Execute job 74

When is 75 retrieved?

74 executes twice

And with concurrent programs in general...

```
void send_to_printer(int user_id, char* document)
{
    printer_write("Job from user --- %d ---", user_id);
    printer_write("%s", document);
    printer_write("-----------------------");
}
```

Process A

send_to_printer(59, “What a beautiful day!”);

Process B

send_to_printer(12, “I hate going to school!”);

Resulting in...

```
Job from user --- 59 ---
What a beautiful day!
------------------------------
Job from user --- 12 ---
I hate going to school!
------------------------------
```

What I expected!

```
Job from user --- 59 ---
Job from user --- 12 ---
What I hate a beautiful day!
------------------------------
going to school!
------------------------------
```

A possible result….

- **Race condition:** The situation where several processes access and manipulate shared data concurrently. The result critically depends on the timing of these processes (on the order of execution of instructions in the multiple processes), which are “racing”.

Mutual Exclusion!

- Instructions must be accessed in mutual exclusion!

```
void send_to_printer(int user_id, char* document)
{
    printer_write("Job from user --- %d ---", user_id);
    printer_write("%s", document);
    printer_write("------------------------");
}
```

Atomically
Mutual Exclusion
Critical Section

Critical Sections

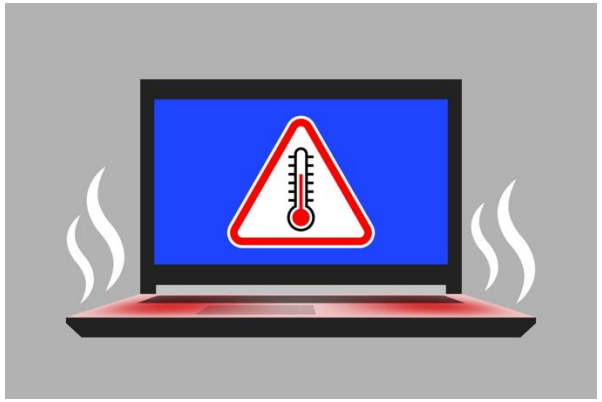
- Processes within a critical region cannot be interleaved without threatening integrity
- Synchronization can be achieved through lock-and-key arrangements
- Some locking mechanisms developed
  - Test-and-set
  - WAIT and SIGNAL
  - Semaphores

Locking mechanisms
Test-and-Set (TS)

- Single, indivisible machine instruction
- Introduced in IBM System 360/370 computers
- TS semantics
  - Saves the key in a storage location – the key is `0` (resource represented by the key is free) or `1` (it is busy)
  - If the key is `0` changes it to `1` and returns free to the process that issued the instruction; the process is allowed to proceed
  - If the key value is `1` (resource busy), the process awaits in cycle (!)
- Can cause starvation
  - A process might never get the resource if many are testing to enter the critical region!
- Can cause busy waiting
  - The process keeps testing the value!

Busy Waiting

AVOID IT!!



```
/* PROCESS 0 /*
•
•
while (turn != 0)
    /* do nothing */ ;
/* critical section*/;
turn = 1;
•
```

```
/* PROCESS 1 */
•
•
while (turn != 1)
    /* do nothing */;
/* critical section*/;
turn = 0;
•
```

Busy Waiting:
MOSTLY FRYING YOUR CPU

Client asks for a meal at a take-away restaurant…

- With BUSY WAITING
  - Client stands at the desk and keeps asking if meal is ready
  - Recepcionist will go crazy
  - Other clients are not happy
  - Client gets tired
- Without BUSY WAITING
  - Client gets a ticket with a number
  - When the number is called, the client goes and gets the meal; meanwhile can take a nap
  - Everybody is happy

However...

- In some VERY specific cases it might be used (WILL NOT BE YOUR CASE DURING OS COURSE!!) – spinlocks
  - In real-time systems
    - In some real-time systems, where timing is crucial, busy waiting might be used to ensure that a process responds immediately to an event.
  - In multiprocessor systems
    - When the expected waiting time is less than the quantum of a thread, to avoid the overhead of context switching
      - The general rule is to use a spinlock if the lock will be held for a duration of less than two context switches
  - In OSs kernels
    - When context switching is too expensive and waiting time is expected to be short

Busy Waiting

- Some problems created by busy-waiting
  - Wastes CPU
  - Decreases global system performance
  - Increases power comsumption
  - Has scalability issues
  - …

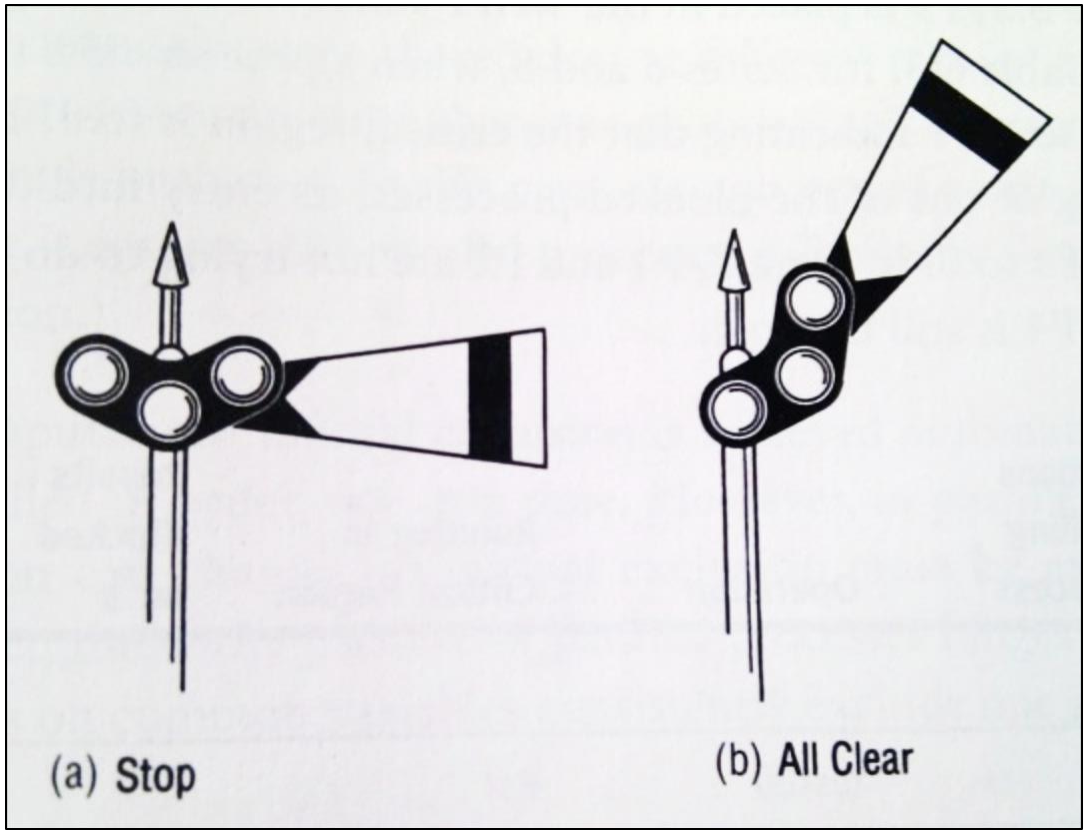
Locking mechanisms
WAIT and SIGNAL

- TS without busy waiting

- Semantics
  - On a busy condition set the process to waiting status (WAIT)
  - When a process exits the critical region, SIGNAL is activated, and the OS moves the next process waiting to enter that critical region to the ready list
    - Removes busy waiting!

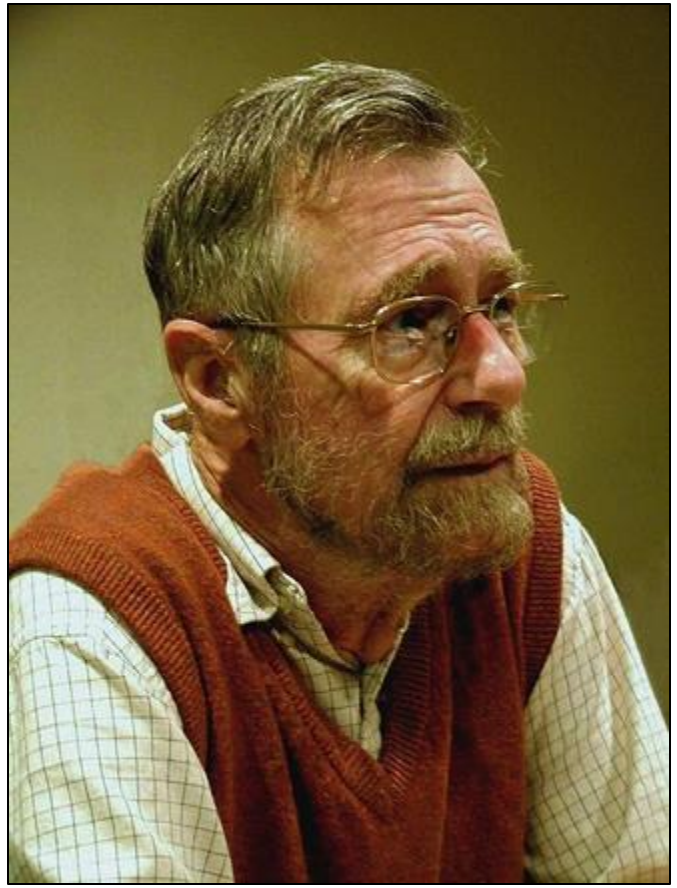
Locking mechanisms
Semaphores

- In railroads…



The inventor of the "Semaphore"

- Edsger Dijkstra was one of the most brilliant computer scientists ever!
- Inventor of the semaphore, one of the key contributions for modern operating systems; developed the THE operating system
  - Used in all operating systems today!
- Created the Dijkstra algorithm for finding the shortest path in a graph
- Wrote "A Case against the GO TO Statement", and was one of the fathers of Algol-60
  - Which introduced the revolution of structured programming!



Edsger Dijkstra

Semaphores

- In computers…
  - Dijkstra (1965)
    - P (*proberen* – 'to test' in Dutch)
    - V (*verbogen* – 'to increment' in Dutch)
  - *V(s): s := s + 1*
    - Fetch, increment and store in one cycle – indivisible action!
  - *P(s): If s > 0 then s := s – 1*
    - Test, fetch, decrement, and store in one cycle – indivisible action!

Semaphores: practical examples

- Imagine the resource is as a shower cabin in a bathroom…
  - Shower cabin = resource
  - Key to bathroom = semaphore to control access

  - Control access to 1 shower cabin:
    - One shower = resource
    - One key to access shower = semaphore initialized with 1
    - Only one person can take the key, enter the room and take a shower
  - Control access to a bathroom with 2 shower cabins
    - Two showers = two instances of the same resource
    - Two equal keys to access shower = semaphore initialized with 2
    - There are 2 keys that allow access to showers to a maximum of 2 persons at the same time

Semaphores: practical examples

- Imagine the resource is a parking space in a closed park…
  - Parking space = resource
  - Parking gate = semaphore to control access
  - Screen with spaces availability = resource counter

  - Control access to park:
    - On arriving
      - A driver waits on the parking gate **(wait on semaphore**) and takes a card;
      - If there are available spaces, the gate opens; otherwise, he must wait;
      - If he enters, the total number of available spaces decreases 1;

    - On leaving
      - The driver inserts the card on the gate (**post on semaphore**);
      - The gate opens and the number of available spaces increases 1.

Semaphores: practical examples

- Imagine a warehouse and an office (multiple resources), and a stock to count…
  - Warehouse = resource
  - Stock book in office = resource
  - 1 warehouse key = semaphore controlling access to one resource
  - 1 office key = semaphore controlling access to one resource

  - To count stocks:
    - In order to count stocks, an employee must access the office to get the stock book and to the warehouse; without both he cannot fulfill the job!
    - If two employees want to count stocks and one takes the key to the office, while the other takes the key to the warehouse, they become blocked by each other!

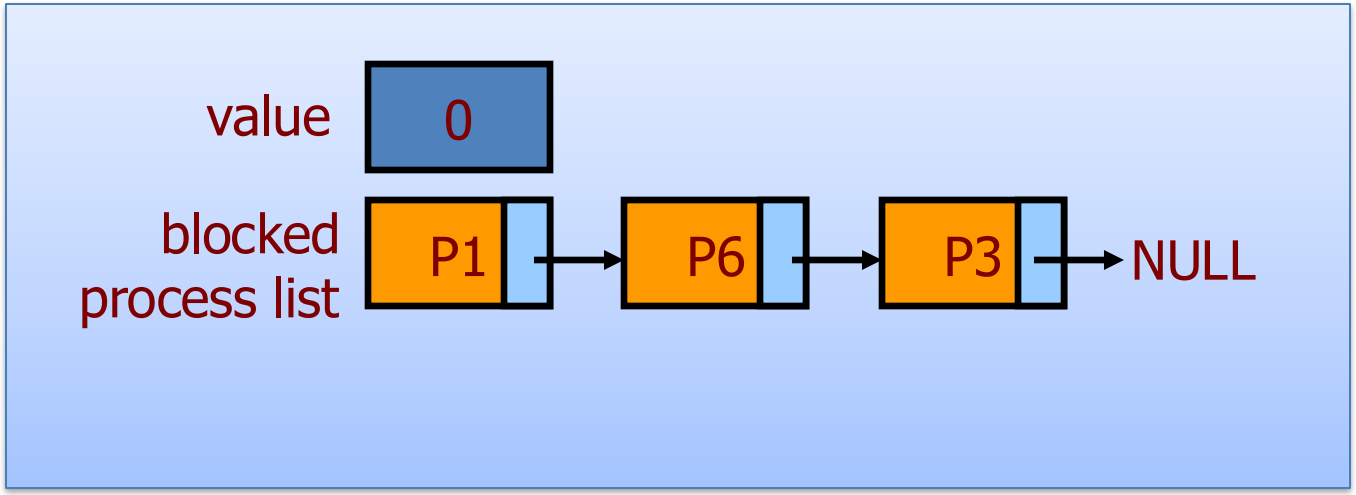
Semaphores in programming

- A semaphore is a synchronization object
  - Controlled access to a counter (a value)
  - Two operations: **`wait()`** and **`post()`**
  - Can also be used as a resource counter – to control access to finite resources!
- **`wait()`**
  - If the semaphore is positive, decrement it and continue
  - If not, block the calling thread (process)
- **`post()`**
  - Increment the semaphore value
  - If there was any (process) thread blocked due to the semaphore, unblock one of them.

- Blocked Processes in a semaphore do not consume resources (CPU)!

The anatomy of a Semaphore

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```



"A semaphore"

Semaphores implementations

- Unix System V Semaphores (aka Process Semaphores)
  - Works with semaphore arrays
  - `semget(), semctl(), semop()`
  - A little bit hard to use by themselves
    → Use a library to encapsulate them!

- POSIX Semaphores
  - Quite easy to use
  - `sem_init(), sem_close(), sem_post(), sem_wait()`

See more on TP Classes slides!

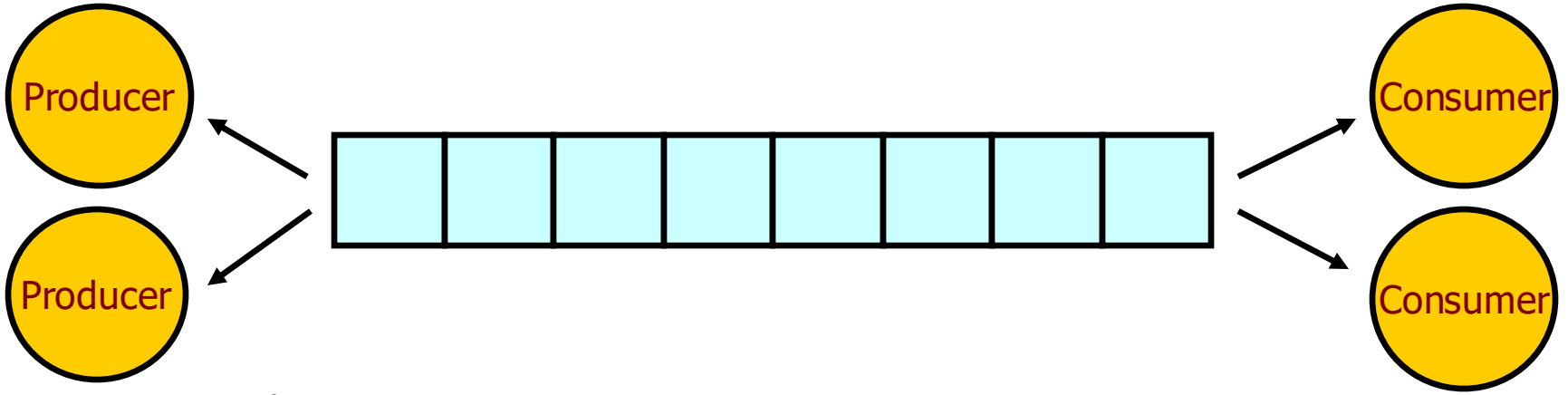
Process Cooperation

- Classical examples of multi-process synchronization
  - Producers and Consumers
  - Readers and Writers

Classical Producer/Consumer

Problem

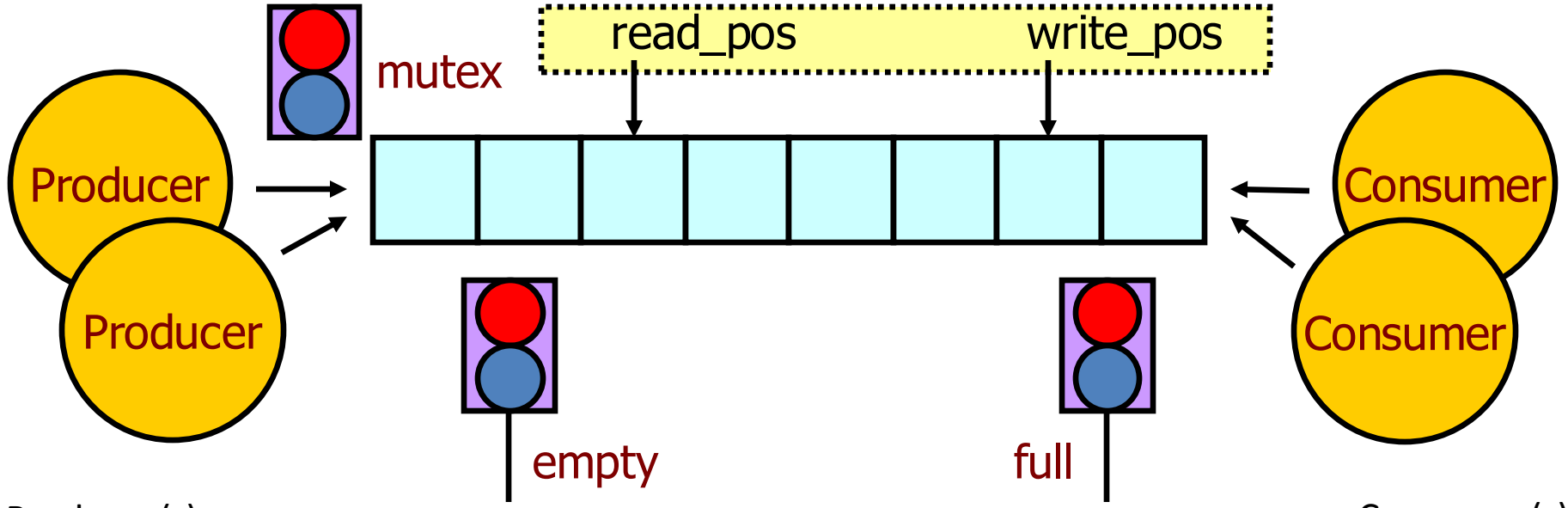
- A producer puts elements on a finite buffer. If the buffer is full, it blocks until there is space.
- The consumer retrieves elements. If the buffer is empty, it blocks until something comes along.



- We will need three semaphores
  - One to count the empty slots
  - One to count the full slots
  - One mutex (or binary semaphore) to provide for mutual exclusion to the shared variables that control access to bufer
    - Note: if there was only 1 producer and 1 consumer, the mutex was not needed

Classical Producer/Consumer

Basic Implementation



Producer(s)

```
put_element(e) {
   sem_wait(empty);
   sem_wait(mutex);
   buf[write_pos] = e;
   write_pos = (write_pos+1) % N;
   sem_post(mutex);
   sem_post(full);
}
```

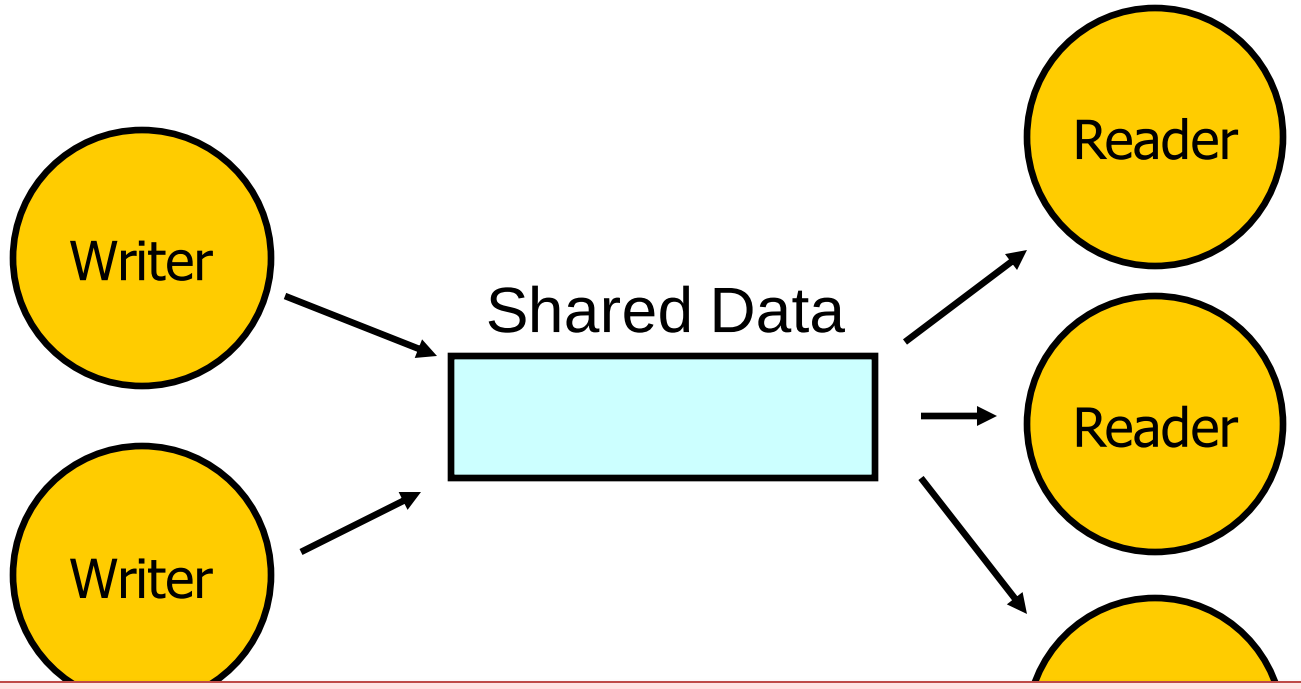
Consumer(s)

```
get_element() {
   sem_wait(full);
   sem_wait(mutex);
   e = buf[read_pos];
   read_pos = (read_pos+1) % N;
   sem_post(mutex);
   sem_post(empty);
   return e;
}
```

Classical Readers/Writers

Problem

- Writer processes must update shared data.
- Reader processes have to check the values of the data. They should all be able to read at the same time.

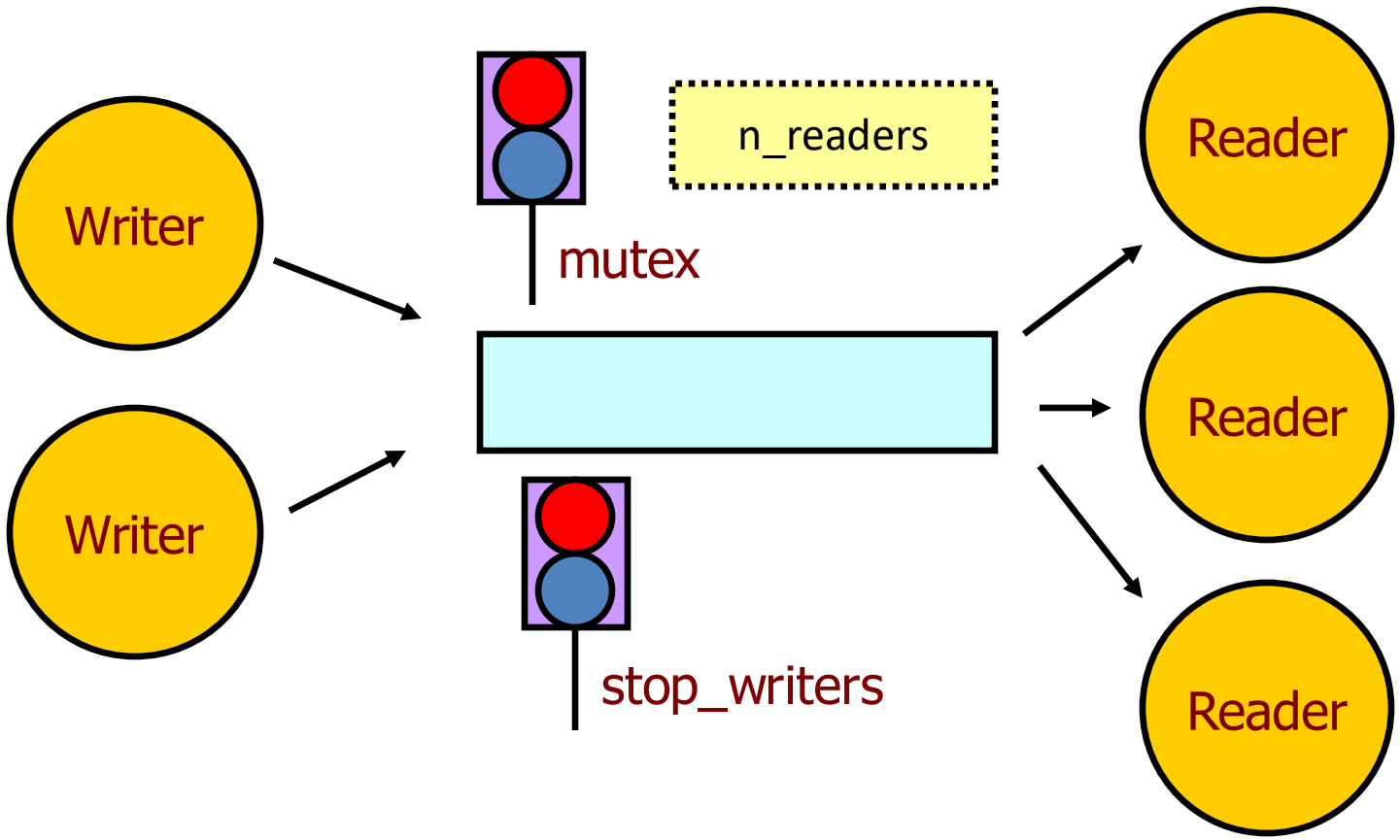


- Why is this different from the Producer/Consumer problem?
- Why not use a simple mutex?

Classical Readers/Writers

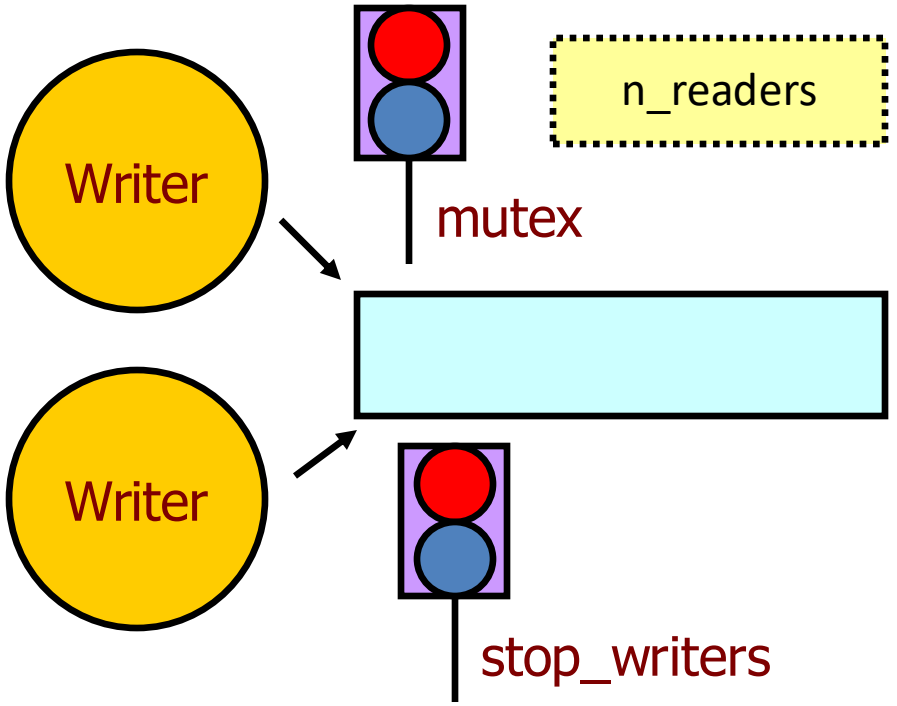
Synchronization

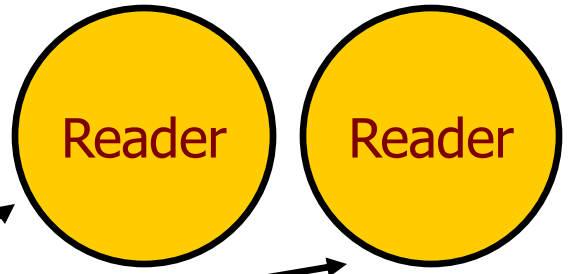
- We will need two semaphores:
  - One to stop the writers and guarantying mutual exclusion when a writer is updating the data
  - One to protect mutual exclusion of a shared variable that counts readers



Classical Readers/Writers

Basic Implementation: Priority to Readers






```
write(e) {
   sem_wait(stop_writers);
   buffer = e;
   sem_post(stop_writers);
}
```

```
read() {
   sem_wait(mutex);
   ++n_readers;
   if (n_readers == 1)
     sem_wait(stop_writers);
   sem_post(mutex);

   e = buffer;

   sem_wait(mutex);
   --n_readers;
   if (n_readers == 0)
     sem_post(stop_writers);
   sem_post(mutex);
   return e;
}
```

Classical Readers/Writers

Some Considerations on the algorithms

- The previous algorithm gives priority to readers
  - Not always what you want to do
- There is a different version that gives priority to writers in the original paper

- Why should I care?
  - This algorithm is the fundament of all database systems! Concurrent reads of data; single update.
  - … One bank agency deposits some money in an account; at the same time, all over the country, many agencies can be reading it
  - … You are booking a flight. Although someone in England in also booking a flight, you and thousands of people can still see what are the available places in the place.

Reader/Writer

- Both versions can suffer from starvation problems
  - Priority to readers
    - Writers can starve
  - Priority to writers
    - Readers can starve

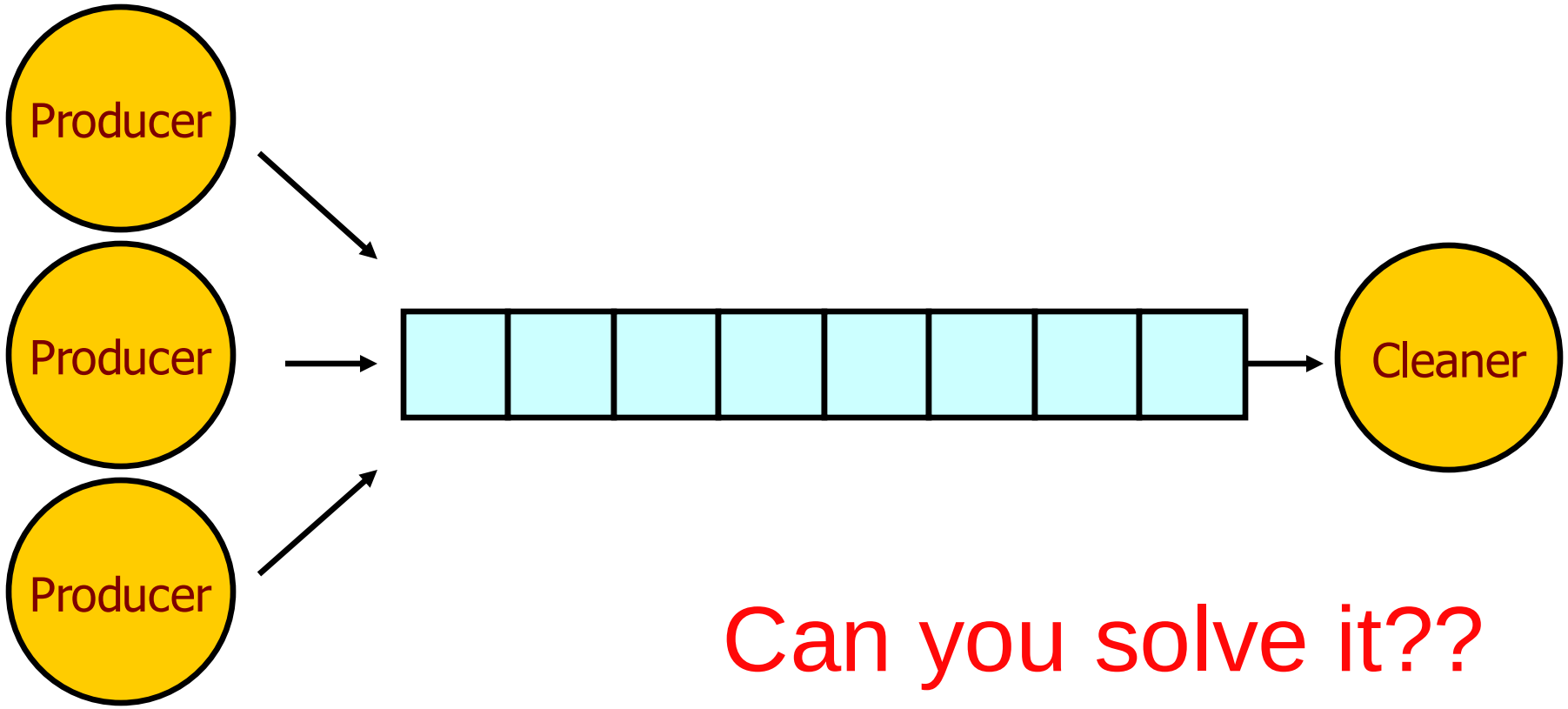
- **Let's write a version that handles the starvation problem ☺**

More problems

- … to solve ☺

Buffer Cleaner

- A buffer can hold a maximum of N elements. When it is full, it should be immediately emptied. While the buffer is being emptied, no thread can put things into it.



Can you solve it??

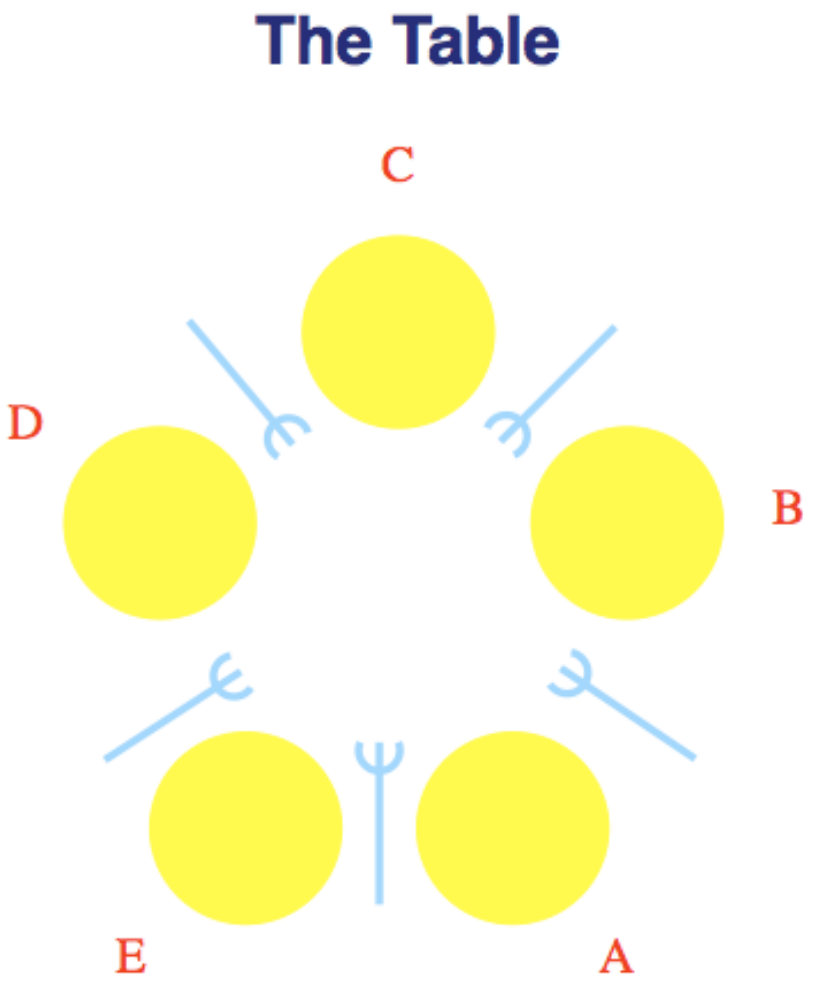
Sleeping Barber

- A barbershop has N waiting chairs and a barber chair
- If there are no customers, the barber goes to sleep
- When a customer enters:
  - If all chairs are occupied, he leaves
  - If the barber is busy but there are chairs, then the customer sits and waits
  - If the barber is sleeping, the customer wakes him up

Can you solve it??

Dining Philosophers

- Philosophers sitting around a table (A to E in the picture)
- There are forks between each pair of plates
- Philosophers need two forks to eat



Can you solve it??

Baboons crossing a canyon

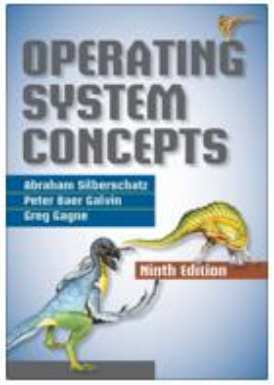
- Western-Baboons have food, but do not have water
- Eastern-Baboons have water, but do not have food
- To get the missing element they must cross the canyon using a rope
  - Passage along the rope follows these rules:
    - Several baboons can cross at the same time, if they go in the same direction.
    - If eastward and westward baboons get onto the rope at the same time, a deadlock will result (the baboons will get stuck in the middle)
    - If a baboon wants to cross the canyon, he must check to see that no other baboon is currently crossing in the opposite direction.

Baboons crossing a canyon

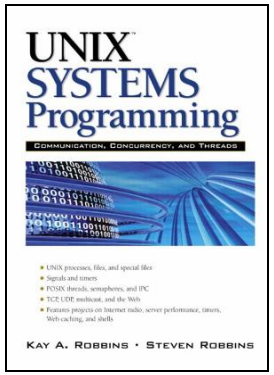
- **Avoid starvation**:
  - When a baboon that wants to cross to the east arrives at the rope and finds baboons crossing to the west, the baboon waits until the rope is empty, but no more westward moving baboons are allowed to start until at least one baboon has crossed the other way.

Can you solve it??

References



- [Silberschatz13]
  - Chapter 5: Process Synchronization
    - 5.5, 5.6, 5.7



- [Robbins03]
  - Chapter 12: POSIX Threads
  - Chapter 13: Thread Synchronization
  - Chapter 14: Critical Sections and Semaphores

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.
—Rudyard Kipling