Synchronization

Concurrent Processes

- Concurrent processes can:
  - Compete for shared resources
  - Cooperate with each other in sharing the global resources
- OS deals with competing processes
  - carefully allocating resources
  - properly isolating processes from each other
- OS deals with cooperating processes
  - providing mechanisms to share resources

Processes: *Competing*

- Processes that do not work together cannot affect the execution of each other, but they can compete for devices and other resources
- **Example:** Independent processes running on a computer
- **Properties:** Deterministic; reproducible
  - Can stop and restart without side effects
  - Can proceed at arbitrary rate

Processes: *Cooperating*

- Processes that are aware of each other, and directly (by exchanging messages) or indirectly (by sharing a common object) work together, may affect the execution of each other
- **Example**: Transaction processes in an airline reservation system
- **Properties**:
  - Share a common object or exchange messages
  - Non-deterministic; May be irreproducible
  - Subject to *race conditions* – coming up!

Why Cooperation?

- Cooperation clearly presents challenges – why do we want it?
- We may want to share resources:
  - Sharing a global task queue
- We may want to do things faster:
  - Read next block while processing current one; divide a job into pieces and execute concurrently
- We may want to solve problems modularly

UNIX example:

```
cat infile | tr ‘ ‘ ‘\012’ | tr ‘[A-Z]’ ‘[a-z]’ | sort |     uniq –c
```

See next slide for an alternate explanation

Problem with C

Original application..

A = 1
B = 2
A = B + 1
B = B * 2

- Hard to understand or executing programs
- Instructions of the prog arbitrarily
  - For cooperating processes, the order of (some) instructions are irrelevant
  - However, certain instruction combinations must be avoided, for example:

Process A	Process B	concurrent access
`A = 1;`	`B = 2;`	*does not matter*
`A = B + 1;`	`B = B * 2;`	*important!*

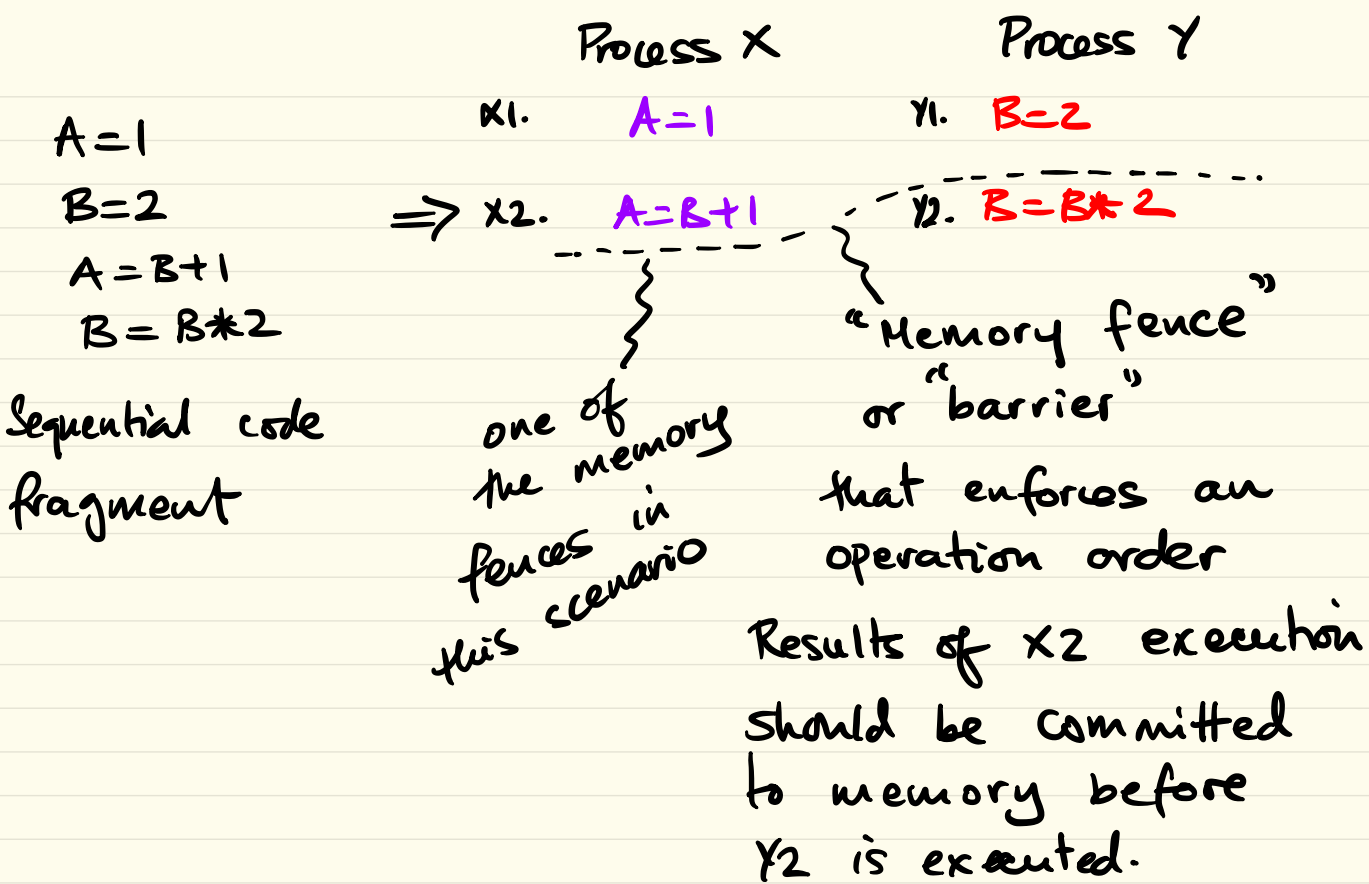
Consider a sequential program fragment

```
A=1
B=2
A=B+1
B=B*2
```

Suppose we are interested in concurrently executing this program fragment.

Using concurrent execution we could exploit hardware parallelism to run faster where appropriate

→ For correctness, the outcome of such an execution should match the sequential execution.


Process X
Process Y
A=1
B=2
A=B+1
B=B*2
Sequential code fragment
X1. A=1
X2. A=B+1
Y1. B=2
Y2. B=B*2
one of the memory fences in this scenario
"Memory fence" or "barrier" that enforces an operation order
Results of X2 execution should be committed to memory before Y2 is executed.

Race Conditions

- When two or more processes are reading or writing some shared data and final result depends on who runs precisely when is a *race condition*
- How to avoid race conditions?
  - Prohibit more than one process from reading and writing shared data at the same time
  - Essentially we need *mutual exclusion*
- Mutual exclusion:
  - When one process is reading or writing a shared data, other processes should be prevented from doing the same
  - Providing mutual exclusion in OS is a major design issue

See the power point slides for the animated version of this slide.

An Example

Correct balance = $100
($50 withdrawal and $50 deposit)

A & B are updating a
drawing and B depositing.

Initial balance = $100

Process A
R1 <- $100
R2 <- $50
R1 <- $100 - $50
balance <- R1 = $50
store R1, balance

Process B
R3 <- $100
R4 <- $50
R3 <- $100 + $50
balance <- R3 = $150
store R3, balance

Possible values for balance are $50, $150, and $100!

Banking Problem – we have two processes or threads that are modifying a shared variable – the account balance

Withdraw Process

// Retrieve balance in R1
R1 ← $100 (balance)
// substract the amount
R1 ← R1 – $50
// Update the balance
R1 → balance

Deposit Process

// Retrieve balance in R1
R1 ← $100 (balance)
// Add. the dep. amount
R1 ← R1 + $50
// update the balance
R1 → balance.

How could we end up with $150 and $50 due to race conditions?

The following execution sequence gives $50.

R1 ← $100 (balance)

R1 ← R1 − $50

R1 ← $100 (balance)

R1 ← R1 + $50

R1 → balance ← $150 in balance.

R1 → balance ← $50 is written into balance by the resumed deposit process.

← we switch from withdraw to deposit before withdraw was able to commit its computed R1

Solution: To perform the balance update in a critical section.

An Example....

- Suppose we have the following code for the account transactions

  authenticate user
  open account
  load R1, balance
  load R2, amount (-ve for withdrawal)
  add R1, R2
  store R1, balance
  close account
  display account info

An Example....

- Suppose we have the following code for the account transactions

```
        authenticate user
        open account
Critical section
        load R1, balance
        load R2, amount (-ve for withdrawal)
        add R1, R2
        store R1, balance
        close account
        display account info
```

Question: Why would the introduction of critical section solve the race condition introduced by the "balance update" problem?

Critical section use prevents an instruction interleaving like the one shown before that causes a race condition. It does not allow another thread to start executing a statement in the CS while a thread is still executing one of it.

This prevents problematic interleaving of statements.

Critical Section

- Part of the program that accesses shared data
- If we arrange the program runs such that no two ~~programs~~ processes are in their critical sections at the same time, race conditions can be avoided

Critical Section

Requirements of a Critical Section

- For multiple programs to cooperate correctly and efficiently:
  - No two processes may be **simultaneously in** their critical sections
  - No assumptions be made about **speeds** or number of CPUs
  - No process running **outside** its critical section may block other processes
  - No process should have to **wait forever** to enter its critical section

Critical Sections

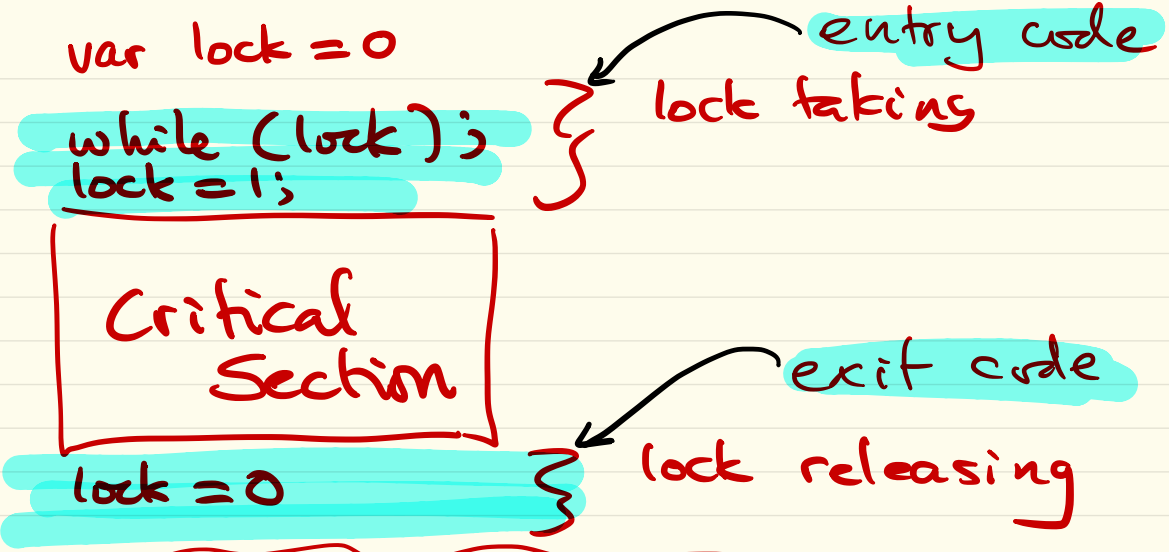
- Critical region execution should be *atomic* i.e., its runs “all or none”
- Should not be interrupted in the middle
- For efficiency, we need to minimize the length of the critical sections
- Most inefficient scenario
  - Whole program is a critical section – no multiprogramming!

Road to a Solution

- Simplest solution:
  - Disable all interrupts
  - This should work in a single processor situation
  - Because interrupts cause out-of-order execution due to interrupt servicing
  - With interrupts disabled, a process will run until it yields the resource voluntarily
- Not practical:
  - OS won't allow a user process to disable all interrupts – OS operation will be hindered too!
  - Does not work on multiprocessors

Road to a Solution

- Idea: To come up with a software solution
- Use “lock” variables to prevent two processes entering the critical section at the same time – all along we are talking about a single critical section
- Use variable `lock`
- If `lock == 0` set `lock =1` and enter_region
- If `lock == 1` wait until lock becomes `0`
- Does not work, Why??



lock = 0

while (lock); ~ empty statement

≡

lock = 0

The simple idea of using a "lock" variable does not wok due to the same problem we encountered with the bank balance update problem.

Strict Alternation

- Two processes take turns in entering the critical section
- Global variable turn set either to 0 or 1

```
Process 0                          Process 1


while (TRUE) {              while (TRUE) {
    while (turn !=0);           while (turn !=1);
    critical_section();         critical_section();
    turn = 1;                   turn = 0;
    non_critical();             non_critical();
}                           }
```

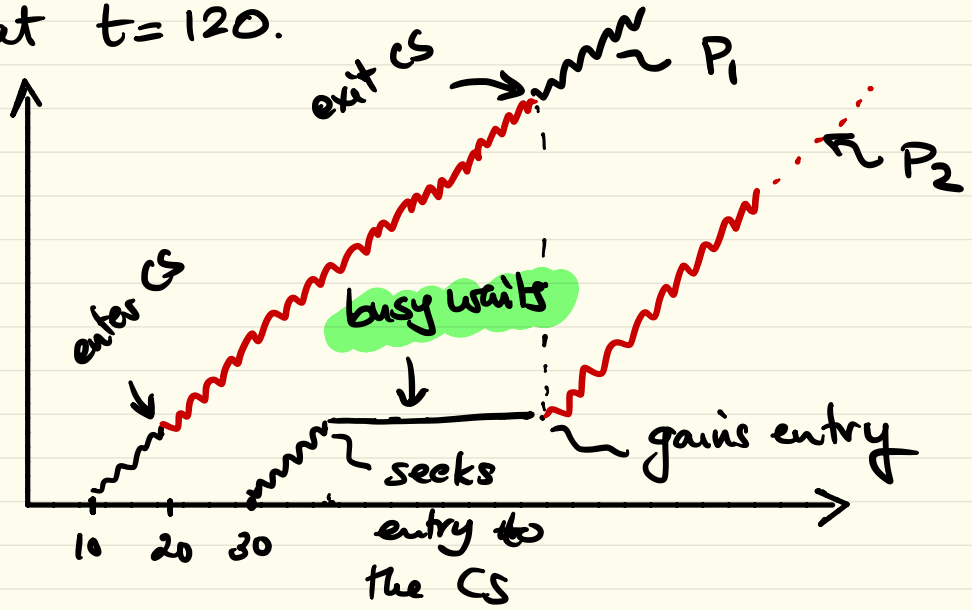
Strict Alternation

- What is the problem with strict alternation?
- We can have starvation, why??
- What are the other drawbacks?
  - Continuously testing a variable until some value appears is called **busy waiting**
  - Busy waiting wastes CPU time – should be used when the expected wait is short
  - A lock that uses busy waiting is called **spin lock**

Lets digress a bit. Suppose we assume that we have got a solution for the critical section problem that uses "busy waiting". We want to understand the pros and cons of this approach.

As the name implies with busy waiting the processes that are unable to access the critical section because another process is currently in it will wait for their turn while continuing to consume CPU cycles. This can cause slowdowns or even deadlocks.

Consider a simple scenario, where process P_1 is created at $t = 10$. It enters the critical section at $t = 20$ and leaves the CS at $t = 120$.



P_2 was busy waiting — running on the CPU and failing to grab the lock needed to enter the CS.

Because P_1 and P_2 are both running on the CPU, with a fair scheduling algorithm that evenly splits the CPU resources among the competing processes, P_1 and P_2 get half the CPU each — assuming no other process is in the system.

Example: We have a process that is running a task that takes 10 s.
For 4s (first 4s), the task is in a CS.
Suppose we are running two instances of the task in two different processes, what is the exec. time?

Lets make the following assumptions:

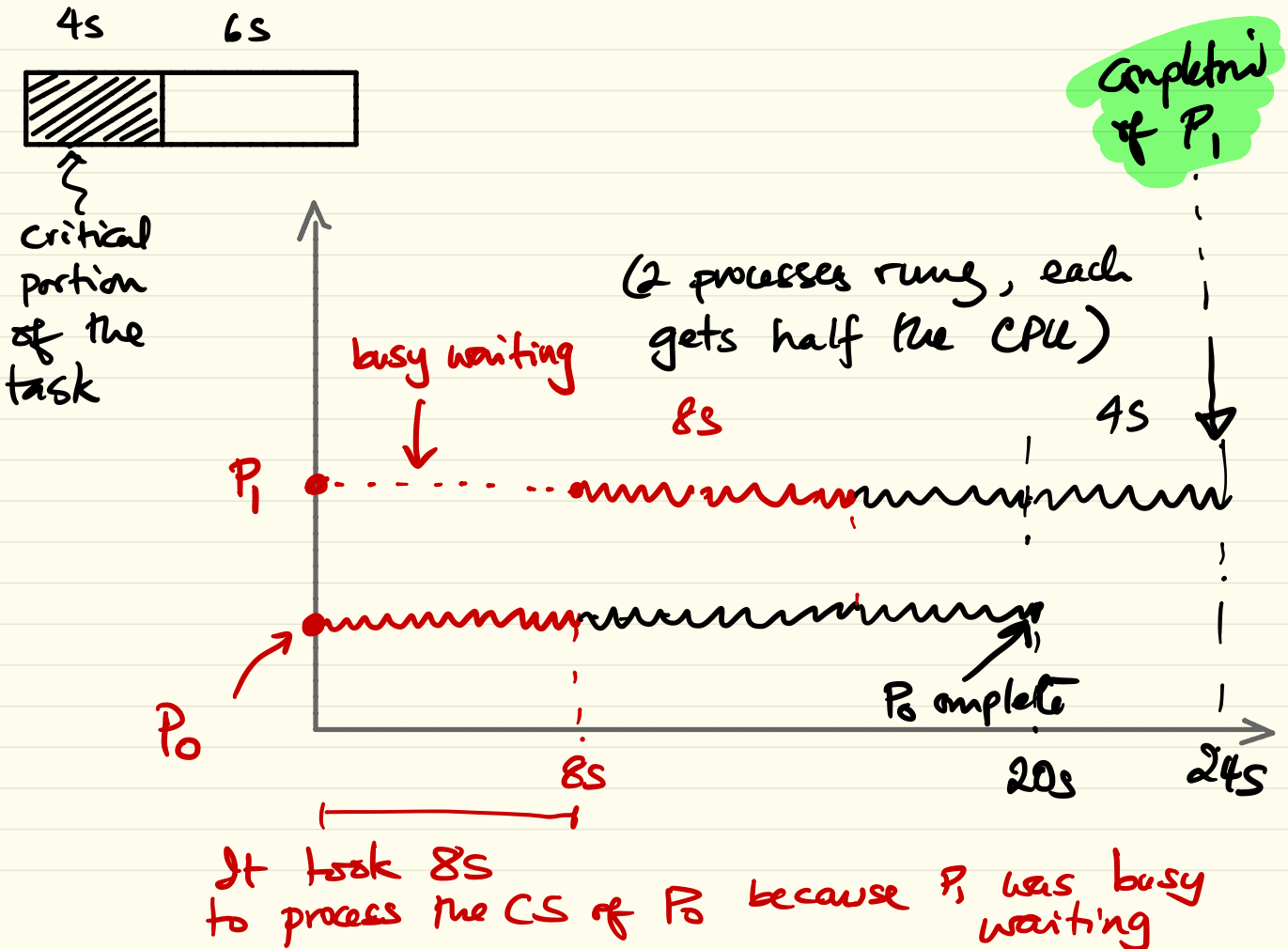
- single CPU, single core.
- Ideal time sharing, no other tasks besides these two tasks.

Simple argument: 2 tasks 20s (total workload)

So we need at least 20seconds.

Due to busy waiting, we have wasted cycles.

One task gets into the CS, and for 4s the next task will be busy waiting. So the second task will end up with a 14s workload. So, we need 24s to complete both tasks.


4s
6s
critical portion of the task
Completion of P_1
(2 processes running, each gets half the CPU)
busy waiting
8s
4s
P_1
P_0
P_0 completes
8s
20s
24s
It took 8s to process the CS of P_0 because P_1 was busy waiting

Question: Suppose the length of the CS is 1.0s, what is the total run time?

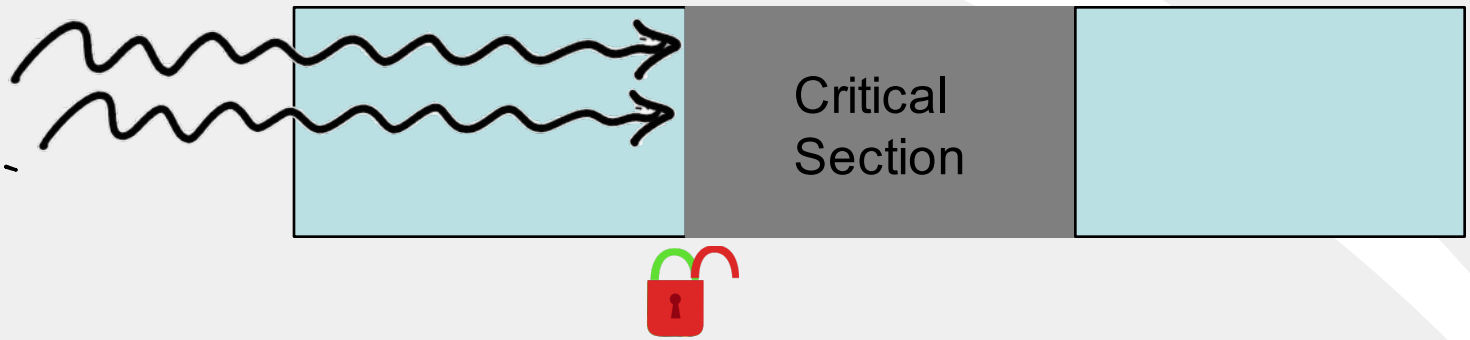
What can you say about the use of busy waiting? Would you recommend its use in certain situations?

Example

Kernel level multi-threads are used for concurrent processing in a data-parallel application. The application has a critical section. If a single thread of execution is used the critical section takes 2 seconds and other parts take 6 seconds. We have a large dataset so we use 2 threads. What is the run time? We have an even larger dataset so we use 8 threads. What is the run time?

Assume single CPU and ideal time sharing (no overhead). All threads starting at the same time.

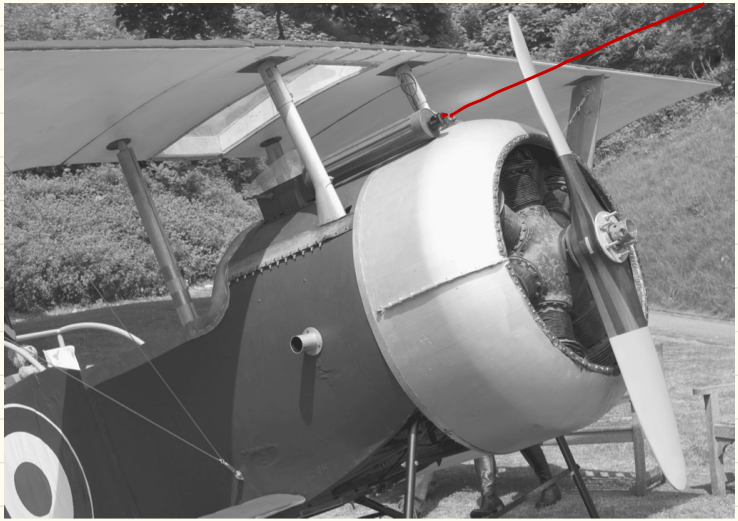
Locks – An Illutration



See the animation in the powerpoint slides

According to a book on Operating Systems, this is the first documented use of critical sections.

The "space" at the intersection of the propeller path and bullet path is the CS that needs to be protected so that only one is crossing it at any given time!



Mutual Exclusion: First Attempt

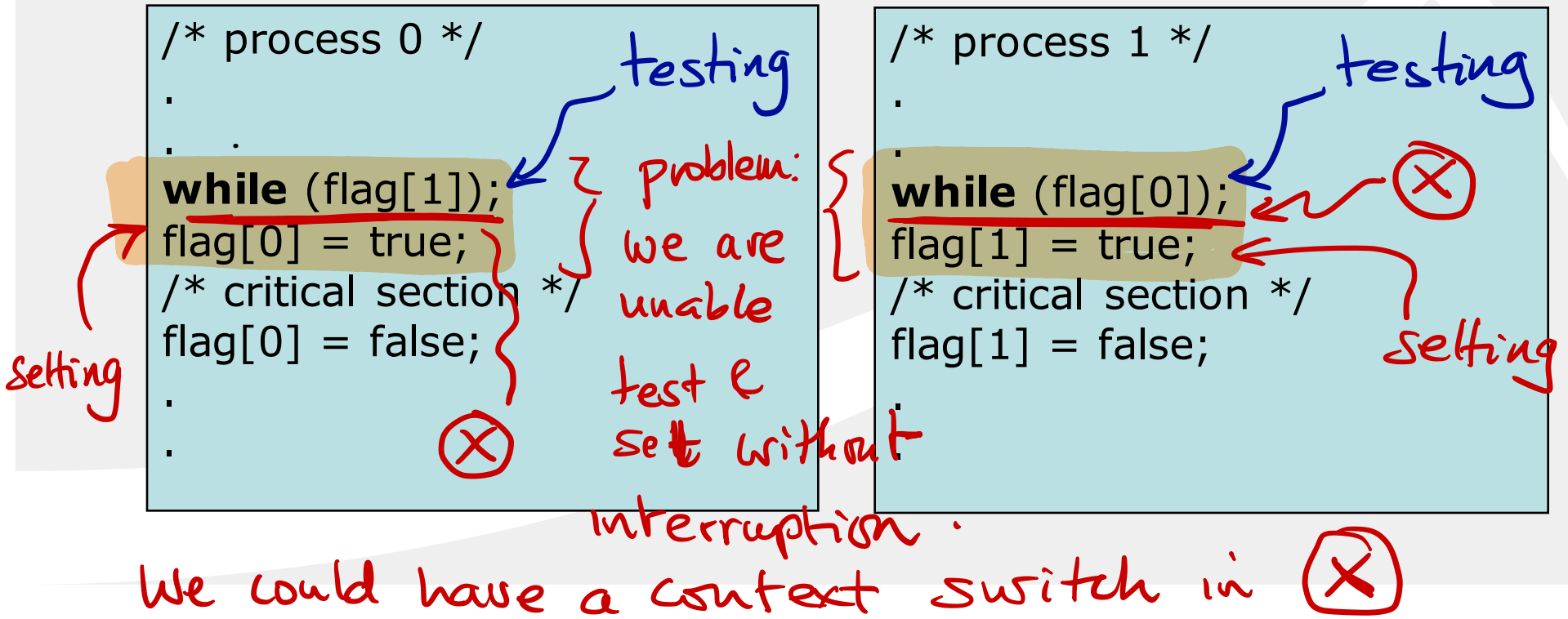
- The simplest mutual exclusion strategy is taking turns as considered earlier

```
/* process 0 */
.
.
while (turn != 0);
/* critical section */
turn = 1;
.
.
```

```
/* process 1 */
.
.
while (turn != 1);
/* critical section */
turn = 0;
.
.
```

Mutual Exclusion: Second Attempt

- First attempt problem – single key shared by the two processes
- Each have their own key to the critical section
- Solution does not work! Why??



Mutual Exclusion: Third Attempt

- This works, i.e., provides mutual exclusion
- Has **deadlock** – why?

In the previous scenario, we have no CS due to context switches in between Testing & Setting

```
/* process 0 */
.
.
flag[0] = true;
while (flag[1]);
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
flag[1] = true;
while (flag[0]);
/* critical section */
flag[1] = false;
.
.
```

Here, we switch the order, we can get deadlock! No process moves forward.

Mutual Exclusion: Fourth Attempt

- This works
- Has **livelock** – why?

One way of attempting to solve deadlock is by "being nice". Where we give up the demand and try to re-grab.

```
/* process 0 */
.
.
flag[0] = true;
while (flag[1])
{
        flag[0] = false;
        /* random delay */
        flag[0] = true;
}
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
flag[1] = true;
while (flag[0])
{
        flag[1] = false;
        /* random delay */
        flag[1] = true;
}
/* critical section */
flag[1] = false;
.
.
```

This can lead to live lock if both parties re-grab at the same instance.

Dekker's Algorithm

```
/* process 0 */
…
flag[0] = true;
while (flag[1])
    {
       if (turn == 1)
       {
          flag[0] = false;
          while (turn == 1);
          flag[0] = true;
        }
     }
     / * critical section */
     turn = 1;
     flag[0] = false;
     …

   …
```

```
/* process 1 */
…
flag[1] = true;
while (flag[0])
    {
       if (turn == 0)
       {
          flag[1] = false;
          while (turn == 0);
          flag[1] = true;
        }
     }
     / * critical section */
     turn = 0;
     flag[1] = false;
     …

  …
```

Peterson's Algorithm

- Dekker's algorithm is complicated
  - hard to prove the correctness
- Peterson's algorithm is much simpler
- Based on the same idea of using the *turn* variable to arbitrate and an array of flags to express interest to enter the critical section

Peterson's Algorithm

```
/* process 0 */
…
flag[0] = true;
turn = 1;
while (flag[1] &&
          turn == 1);
/ * critical section */
flag[0] = false;
/* remainder */
…
```

```
/* process 1 */
…
flag[1] = true;
turn = 0;
while (flag[0] &&
          turn == 0);
/* critical section */
flag[1] = false;
/* remainder */
…
```

Can Hardware Help?

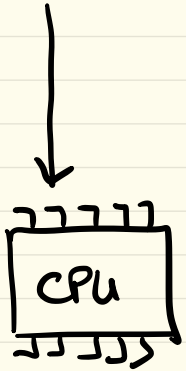
- Dekker's and Peterson's algorithms are pure software solutions
- Can hardware provide any help?
  - make the solution more efficient
  - make it scalable to more processes?
- Yes!
- Current microprocessors have hardware instructions supporting mutual exclusion

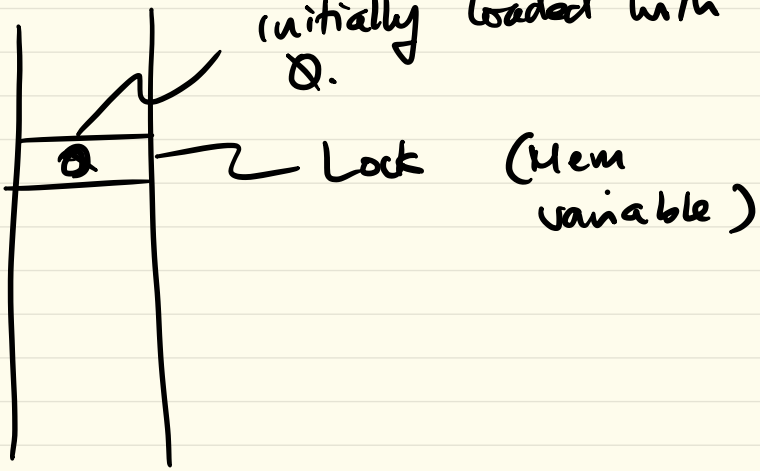
Recall from a earlier discussion that a simple idea of using a "lock" variable, could have worked for providing a critical section if we could do the Testing and Setting atomically. That is once the value is tested and ascertained to contain a certain value we perform a certain modification on the value, without interruption.

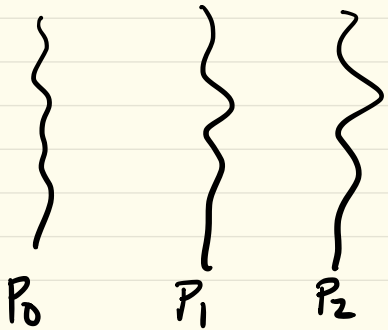
This is the basis of hardware implementations. There are various forms with subtle variations in their capabilities.

Here, we consider Test & Lock (R, Lock)

TRL (R, Lock)







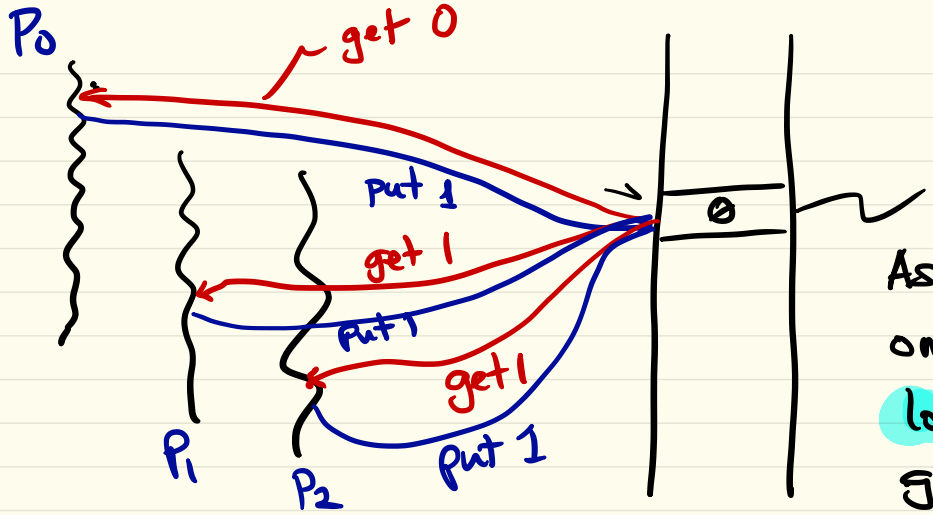
Suppose P_0, P_1, and P_2 execute TRL (R, L) simultaneously, we claim only one process will get 0 in its register.

Test and Lock

- `TSL RX, LOCK` is a typical CPU instruction providing support for mutual exclusion
  - read the contents of memory location `LOCK` into register `RX` and stores a non-zero value at memory location `LOCK`
  - operation of reading and writing are indivisible - atomic

```
enter_section:
    TSL REGISTER, LOCK
     // copy lock to reg and set lock to 1
    CMP REGISTER, #0
    // was lock zero??
    JNE enter_section
    // if non zero, lock was set
    RET
```

```
leave_region:
    MOVE LOCK, #0
    RET
```



Assumption: Only one memory location for the given value.

P_0 gets the 0. It gets the chance to get into the CS. Others wait (busy wait).

The location — Lock — was initialized to 0. All processes are trying to grab this 0 and store a non-zero (1) value in there. Only one process will succeed in getting the 0.

Properties of Machine Instruction Approach

- Advantages:
  - applicable to any number of processes
  - can be used with single processor or multiple processors that **share a single memory**
  - simple and easy to verify
  - can be used to support multiple critical sections, i.e., define a separate variable for each critical section

Properties of Machine Instruction Approach

- Disadvantages:
  - **Busy waiting is employed** – process waiting to get into a critical section consumes CPU time
  - **Starvation is possible** – selection of entering process is arbitrary when multiple processes are contending to enter

Starvation happens when a process is indefinitely delayed in getting its resources - because the system fails to choose it from a competing set of processes

Busy Waiting Approaches

- Mutual exclusion schemes discussed so far are based on busy waiting
- Busy waiting not desirable:
- Suppose a computer runs two processes H: high priority and L: low priority
  - scheduler always runs H when it is in ready state
  - at a certain time L is in its critical section and H become runnable
  - H begins busy waiting to enter the critical section
  - L is never scheduled to leave the critical section
  - there is a **deadlock**
  - this situation is sometimes referred to as the **priority inversion problem**

To illustrate priority inversion, we use the following scenario.

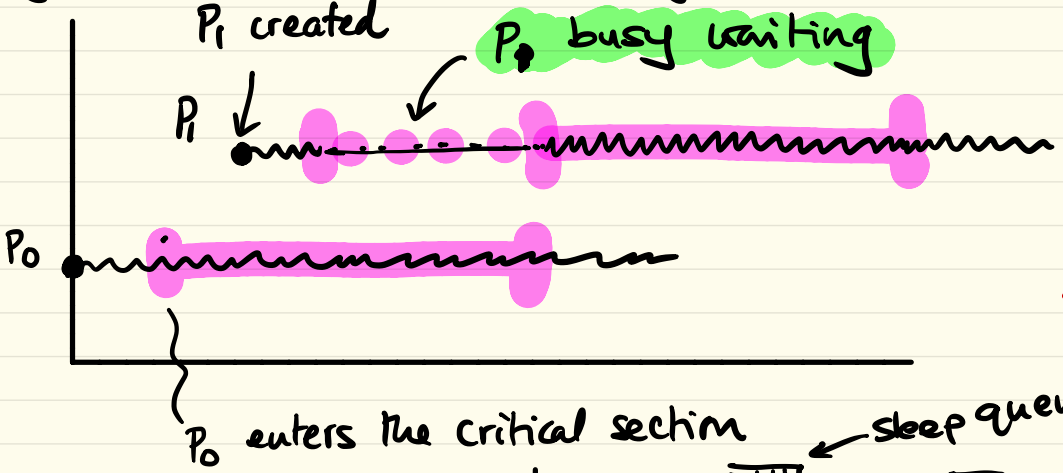
P_1 is unable to get into the critical section X because P_0 is still holding it. P_0 is unable release it because it is unable to run on the CPU.

Sleep/Wakeup Approach

- Alternative to busy waiting that is inefficient and deadlock prone is to use a sleep/wakeup approach
- Implemented by the Semaphores

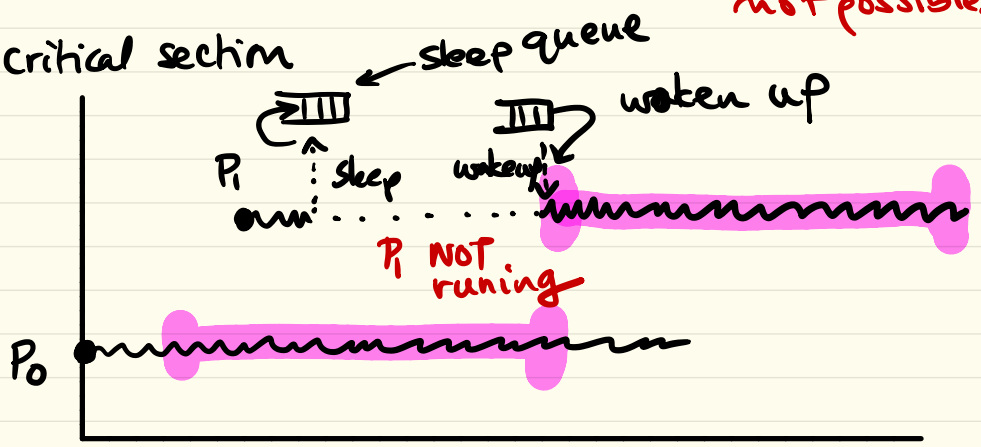
OS support is needed to implement process "sleep" and "wakeup". Most OSes provide semaphores as part of their system call offering.

Lets contrast busy waiting and sleep/wakeup to get a better understanding.



If the critical section is tiny, busy waiting could be a better alternative provided priority inversion is not possible.

There is overhead associated with putting a process to sleep and waking it up.



Semaphores

- Fundamental principle:
  - two or more processes can cooperate by sending simple messages
  - a process can be forced to stop at a specific place until it receives a specific message
  - complex coordination can be satisfied by appropriately structuring these messages
  - for messaging a special variable called semaphore ***s*** is used
  - to transmit a message via a semaphore a process executes signal(s)
  - to receive a message via a semaphore a process executes wait(s)

Semaphores

- Operations defined on a semaphore:
  - can be initialized to a nonnegative value – set semaphore
  - wait – decrements the semaphore value – if value becomes negative, process executing wait is blocked
  - signal – increments the semaphore value – if value is not positive, a process blocked by a wait operation is unblocked

Semaphores

- A definition of semaphore primitives…

```
struct semaphore {
        int count;
        queueType queue;
}
```

```
void wait(semaphore s)
{
    s.count--;
    if (s.count < 0)
    {
        place this process in
        s.queue;
        block this  process
    }
}
```

```
void signal(semaphore s)
{
    s.count++;
    if (s.count <= 0)
    {
        remove a process P
        from s.queue;
        place process P on
        ready list;
     }
}
```

Semaphores

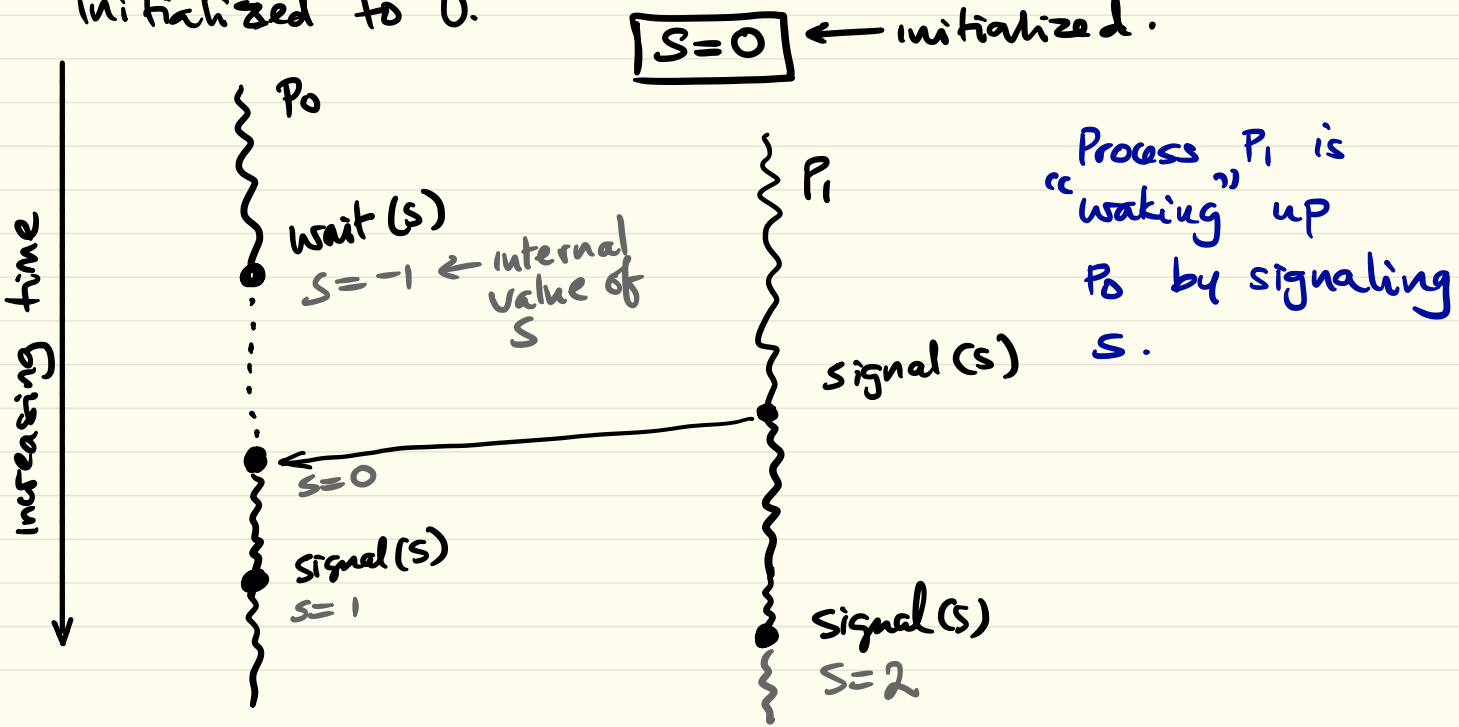
- Wait and Signal primitives are assumed to be atomic
  - they cannot be interrupted and treated as an indivisible step
- A queue is used to hold the processes waiting on a semaphore
- How are the processes removed from this queue?
  - FIFO: process blocked longest should be released next – ***strong semaphore***
  - Order not specified – ***weak semaphore***

Type of sema

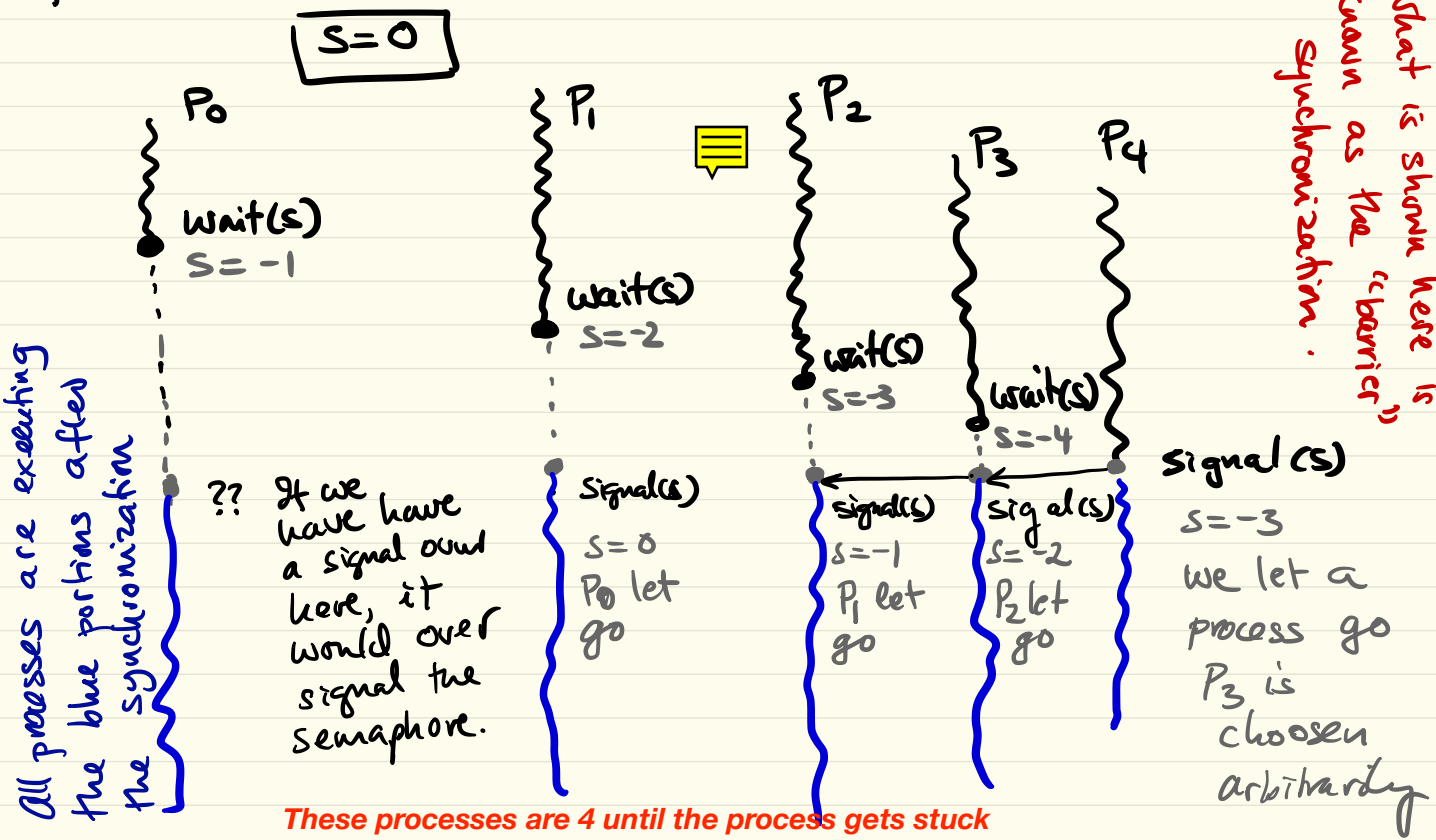
- Semaphores are usuall
  ways:
  - **binary**—is a semaphore wi
    of 0 or 1
    - Or a value of FALSE o RUE if you prefer
    - Initialized to 1 for mutex applications
  - **counting**—is a semaphore with an integer value ranging between 0 and an arbitrarily large number - initial value might represent the number of units of the critical resources that are available - also known as a **general** semaphore

These are initial values.
A semaphore can have
negative value
at any time .. many
threads or processes
waiting on it

To understand the operation of the semaphore operations lets consider a bunch of processes that share a single semaphore s that is initialized to 0.

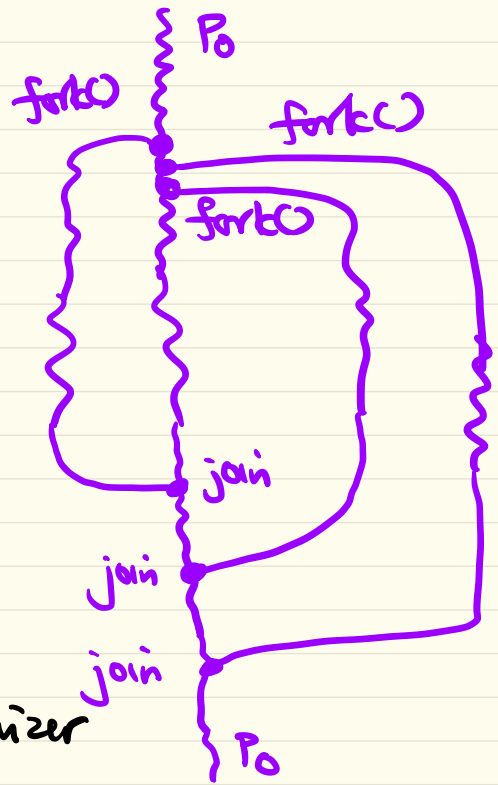


Lets consider another scenario. Here we have 5 processes. We want to synchronize their executions.



These processes are 4 until the process gets stuck because of the memory aspect of semaphores.

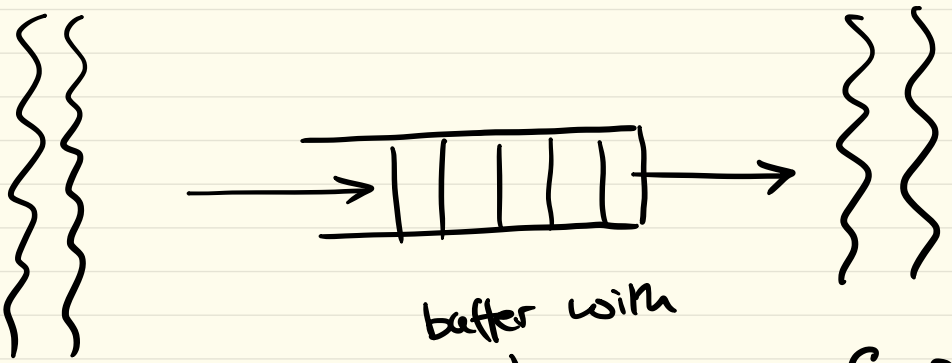
As another example consider a fork-join graph like what is shown next.



The "join" is a synchronization between two different processes. The process that arrives first at the join point needs to wait for the other process until its arrival.

So we can implement a join using a "barrier" type synchronizer for two processes.

Producer – consumer Problem. (Bounded Buffer Version)



1. Buffer can overflow
2. Buffer can underflow (trying to read from empty buffer)
3. Buffer modified by both producer & consumer

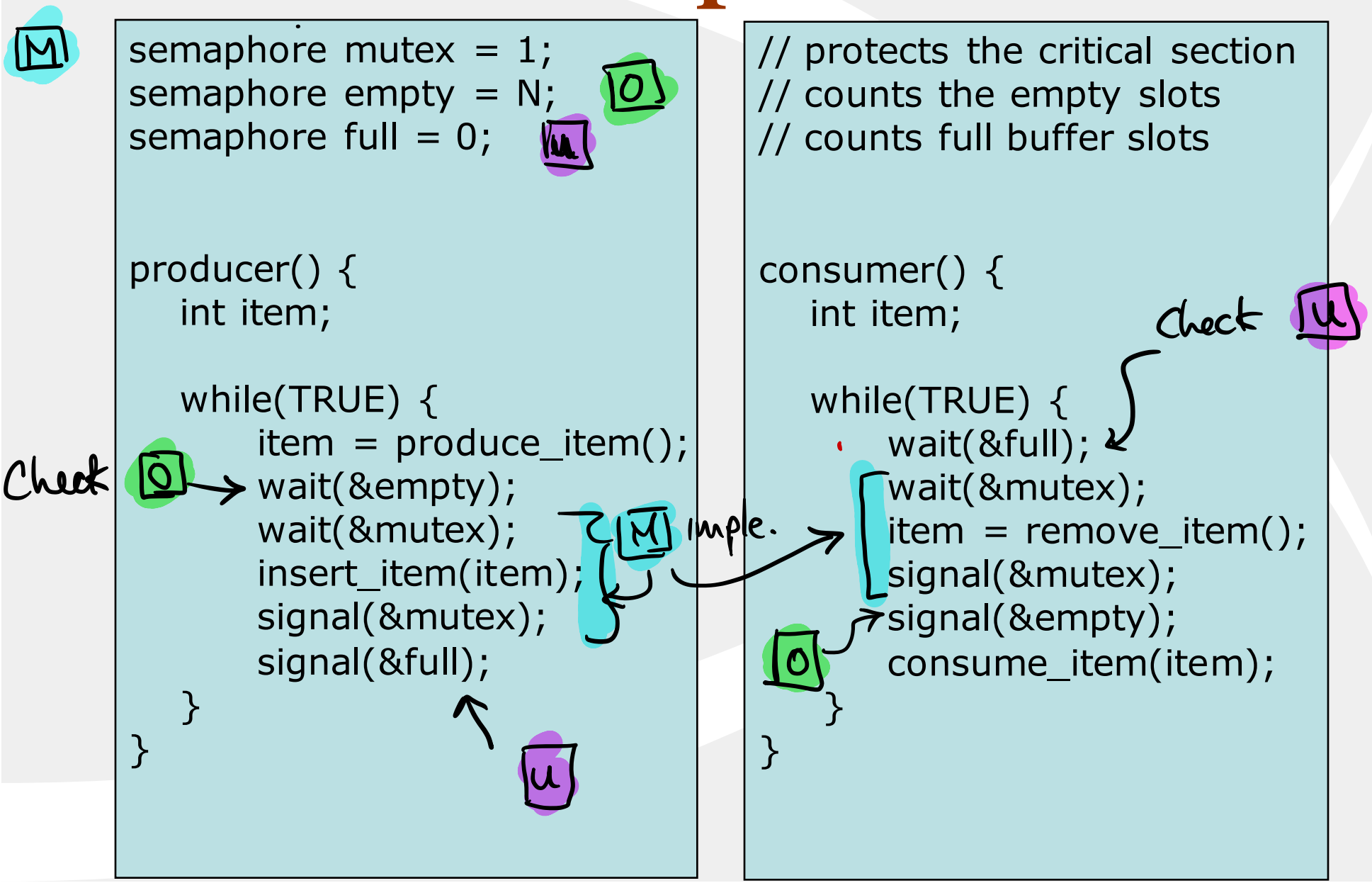
So we have three synchronization requirements.

– overflow synchronization. That is, we want to prevent overflow and the resulting loss of data.

– underflow synchronization. We want to stop underflow. Otherwise, we will read corrupted or duplicated data.

– mutual exclusion synchronization. We want any one process modifying the shared buffer structure for race-free operation.

Producer-Consumer: Semaphores



Producer-Consumer: Semaphores

- One binary semaphore **mutex** to ensure only one process is manipulating the buffers at a time
- **Empty** and **Full** counting semaphores, to count the number of empty or full buffer slots respectively
  - Empty initialized to N, so waiting on empty means waiting if there are no empty slots (*buffer full!*)
  - Full initialized to 0, so waiting on full means waiting if there are no full slots (*buffer empty!*)

Primitives Revisited

- The synchronization primitives discussed so far are all of the form:

```
entry protocol
    < access data >
exit protocol
```

- Semaphores give us some abstraction: implementation of the protocol to access shared data is transparent to the user
- More abstraction would be of additional benefit (as it often is!)

Monitors

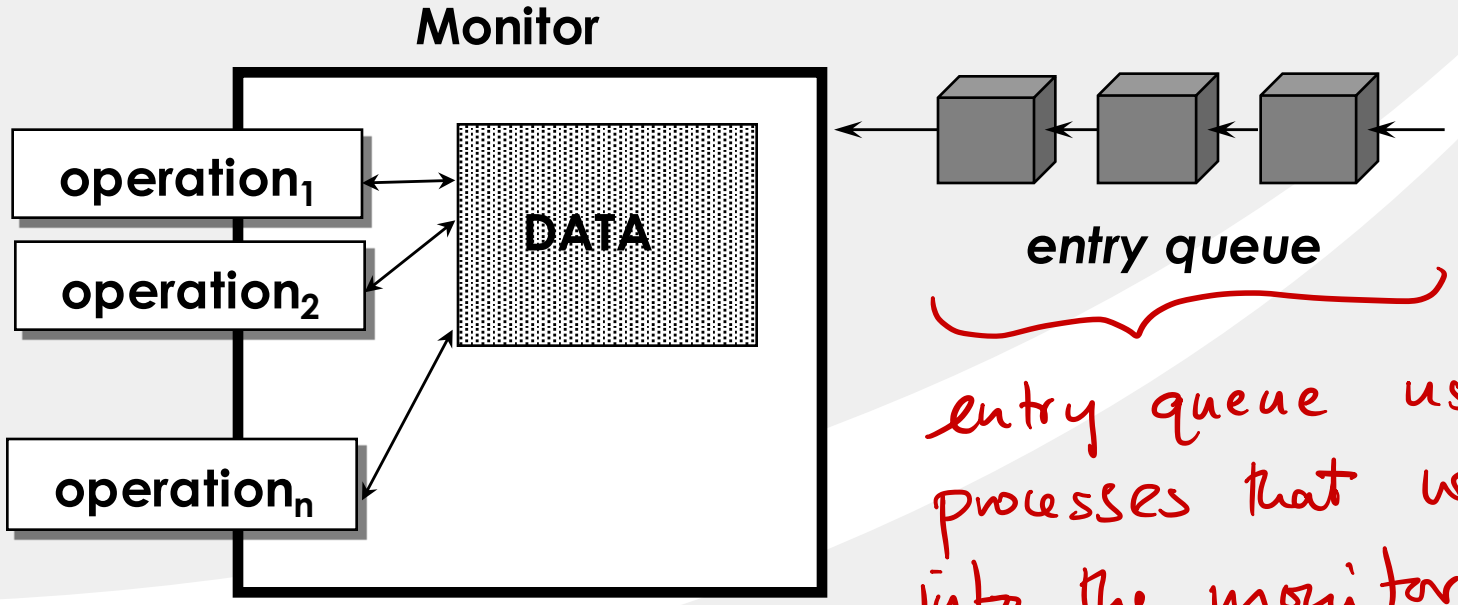
- A monitor is a high-level (programming language) abstraction that combines (and hides) the following:
  - shared data
  - operations on the data
  - synchronization using condition variables

Monitors

- Mutual exclusion --- not the only thing we need for concurrency
- Abstraction provided by monitors allows us to deal with many different issues in concurrency
  - block processes when a resource is busy
- With a monitor data type, we define a set of variables that define the state of an instance of the type, and a set of procedures that define the externally available operations on the type

Monitors

- Monitors are more than just objects
- A monitor ensures that only one process at a time can be active within the monitor – so you don't need to code this explicitly



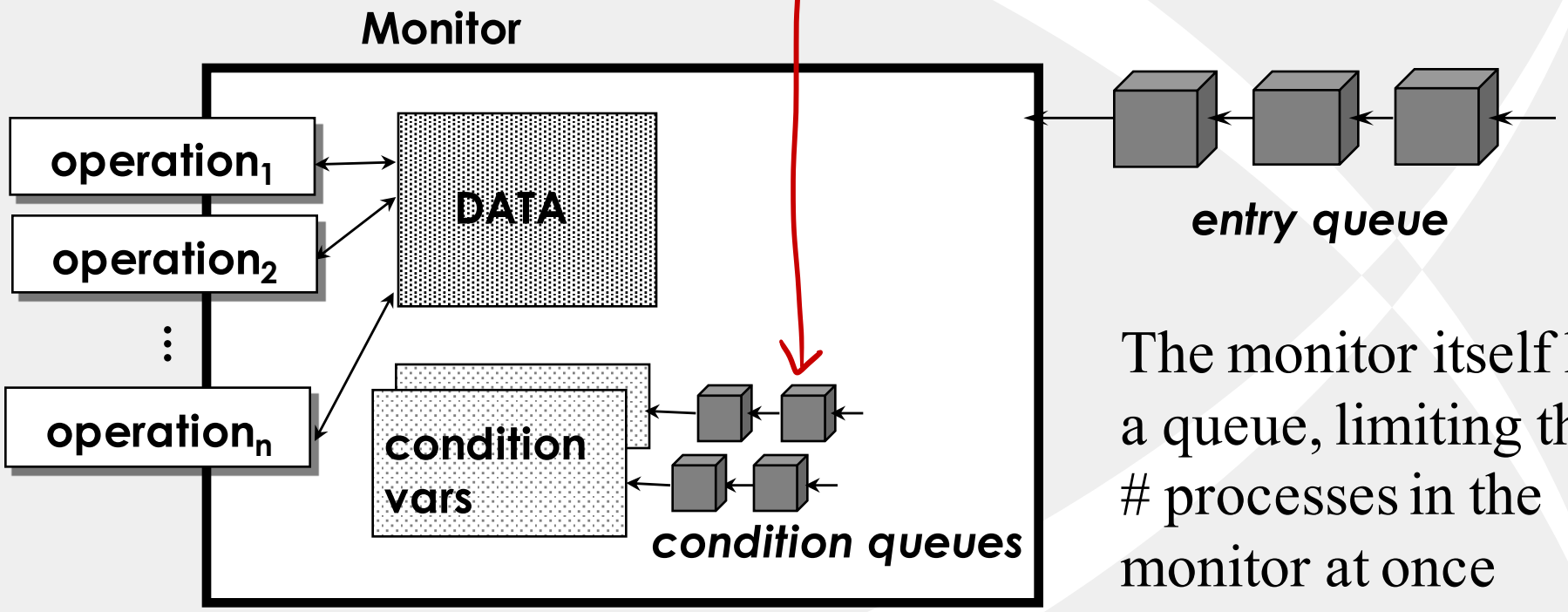
entry queue used by processes that want to get into the monitor but unable because the monitor is busy.

Monitors

- Just this on its own though isn't enough – need the additional synchronization mechanisms
- Monitors use **condition variables** to provide user-tailored synchronization and manage each with a separate queue for each condition
- The only operations available on these variables are **WAIT** and **SIGNAL**
  - **wait** suspends process that executes it until someone else does a **signal** (different from semaphore wait!)
  - **signal** resumes one suspended process. no effect if nobody's waiting (different from semaphore free/signal)!

Monitor Abstraction

Condition variables are used to halt processes within the monitor



The monitor itself has a queue, limiting the # processes in the monitor at once

A process gets "into" the monitor by running an operation that is exposed by the monitor.

How Monitors Work

- Remember only one process can be active in the monitor at once. Some will be waiting to get in, others will be waiting on conditions in the monitor
- If P executes x.signal (signal for condition x), and there's a process Q suspended on x, both can't be active at once. 2 possibilities:
  - P waits on some condition till Q leaves monitor
  - Q waits on some condition till P leaves monitor
- Also compromises: if P executes signal, it leaves monitor automatically, resuming Q

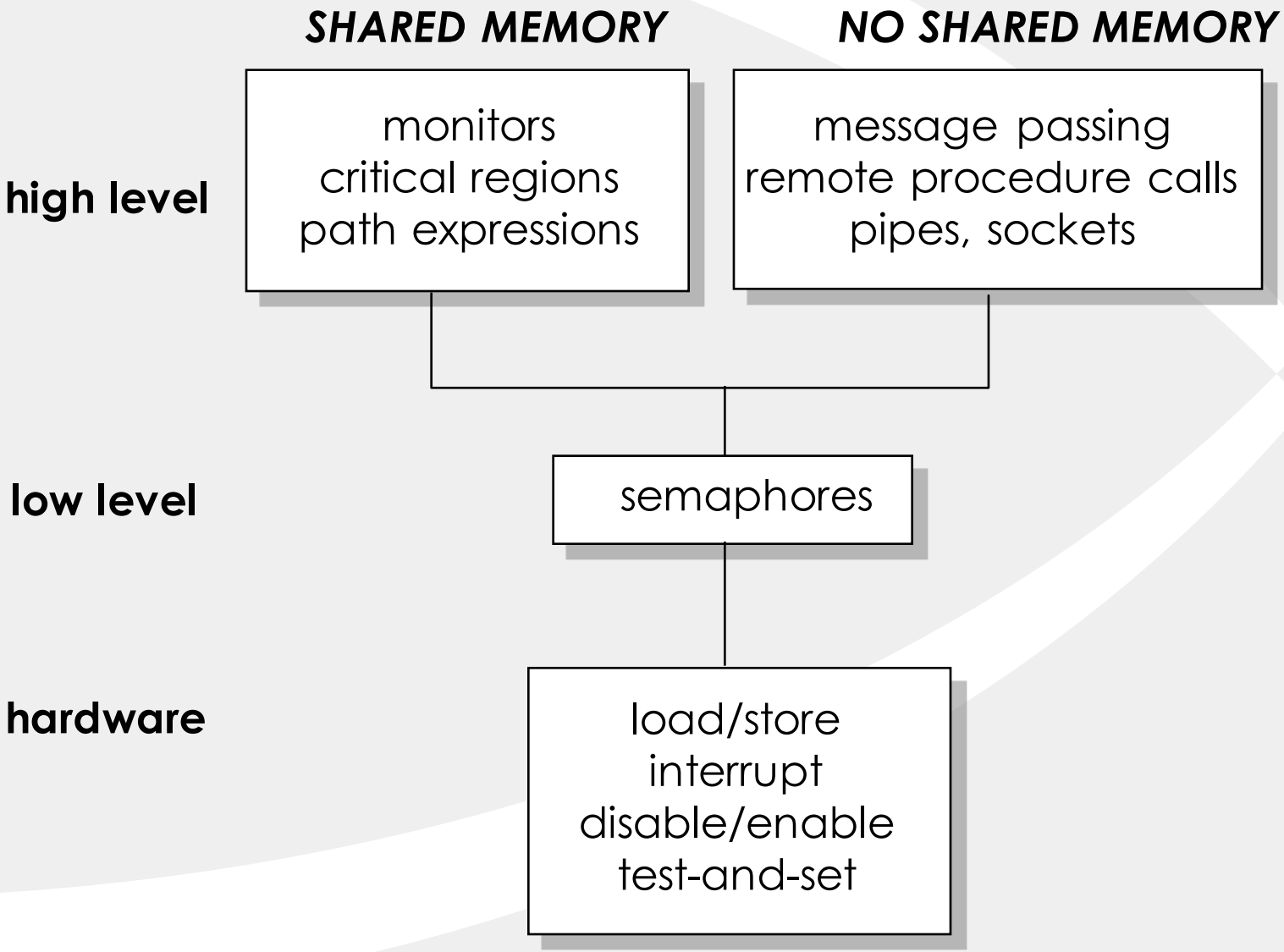
Problems with *synch* Primitives

- *Starvation*: the situation in which some processes are making progress toward completion but some others are locked out of the resource(s)
- *Deadlock*: the situation in which two or more processes are locked out of the resource(s) that are held by each other

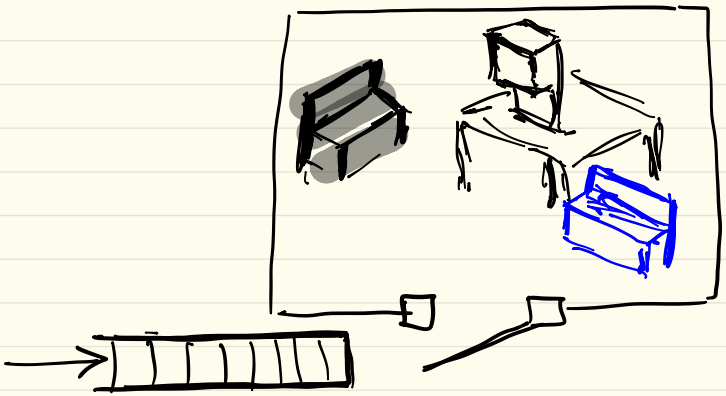
Problems with *synch* Primitives

- The most important deficiency of the primitives discussed so far is that they were all designed on the concept of one or more CPUs accessing a ‘‘common’’ memory
  - Hence, these primitives are not applicable to distributed systems. **Solution**? Message passing...

Synch Primitives—Summary



Monitor Ville



Ice creme maker

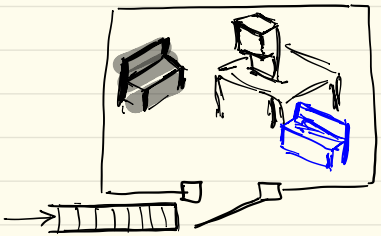
- enter the shop
- put ice creme
  * if drinkers on black bench wake them
- leave the shop.

Icecreme drinker

- enter the shop *
- get ice creme
  if no ice creme
  * goto sleep on black bench
  if got ice creme
  * leave the shop

Details are missing at *

Monitor Ville



Ice creme maker
- enter the shop
- put ice creme
  * if drinkers on black bench wake them
- leave the shop.

Icecreme drinker
- enter the shop *
- get ice creme
  if no ice creme
  * goto sleep on black bench
  if got ice creme
  * leave the shop

Details are missing at *

enter the shop:
- only one drinker or maker can be in the shop at any given time
- sleeping drinkers or makers don't count.

wake up a drinker:
- wake up one drinker and immediately goto sleep on the blue bench.

goto sleep:
be nice()
sleep on the black bench

leave:
be nice();
leave shop;

be nice':
if someone on blue bench wakeup that persn
else
let a new persn into the shop

How about complex scenarios?

- Ice creme maker never sleeps on the black bench.
- what modification to the shop would require a black bench for the ice creme maker?

Go into the ice creme shop, you get ice creme and ketchup?

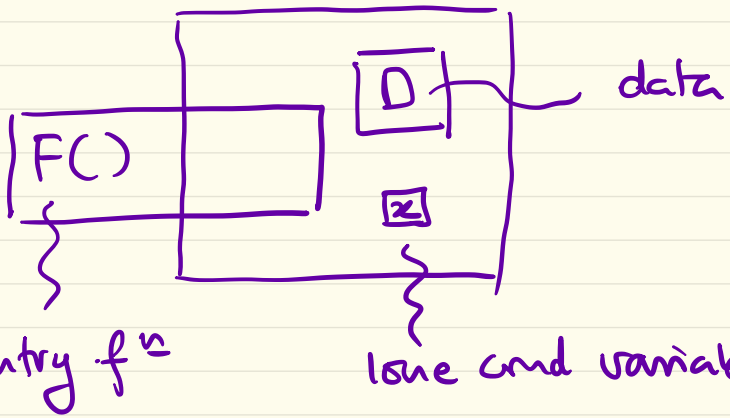
- First ice creme - always.
- Second ketchup - optionally - some drinkers want both some just ice creme

- What modifications?

After giving away ice creme for free for a long time, the ice creme maker closed shop. However, the shop was reused by the city to setup a water fountain.

- Still the same rules apply
- Only one drinker (water) in the shop at any given time. — always finds water
- No need for a black bench. — never run out of water's
- The blue bench comes with the shop and remains unused.

Implement a monitor using Semaphores



Need three semaphores : mutex = 1
x_sem = 0
next = 0

Some count variables : next_count = 0
x_count = 0

(ints)

m.F();
enter the
monitor.

x.signal()
do signal while
inside the monitor

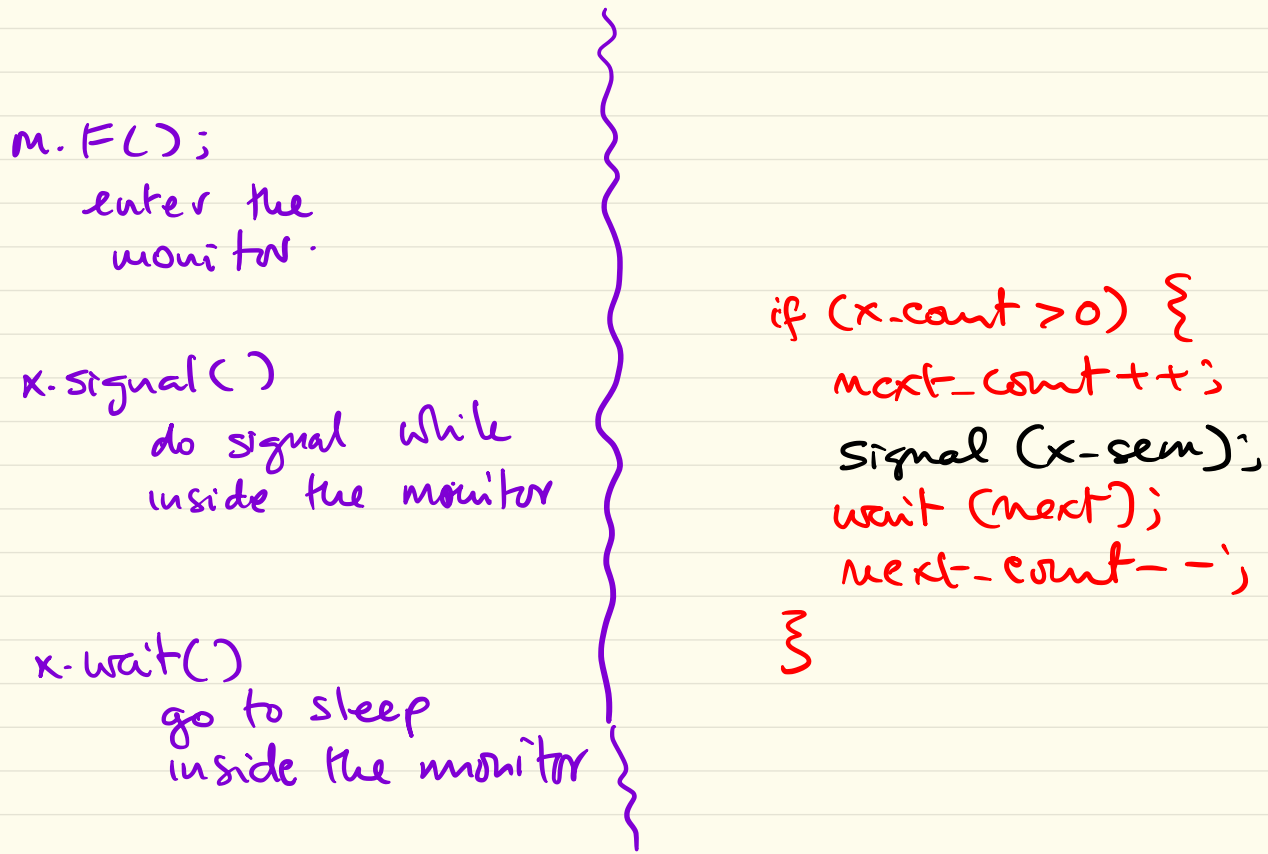
x.wait()
go to sleep
inside the monitor

m.F();
enter the monitor.

x.signal()
do signal while inside the monitor

x.wait()
go to sleep inside the monitor

```
wait (mutex);
    F();
if (next_count > 0)
    signal (next)
else
    signal (mutex);
```


M.F();
enter the monitor.
x.signal()
do signal while inside the monitor
x.wait()
go to sleep inside the monitor
if (x_count > 0) {
next_count++;
signal(x_sem);
wait(next);
next_count--;
}

m.F();
enter the monitor.

x.signal()
do signal while inside the monitor

x.wait()
go to sleep inside the monitor

```
x_count++;
if (next_count > 0)
    signal(next)
else
    signal(mutex);
wait(x_sem);
x_count--;
```

We have seen two types of synchronization mechanisms

To show that they are equally expressive we show how one mechanism could be implemented using the other mechanism.

1. Show semaphores can be simulated using monitors
2. Monitors can be simulated using semaphores.

Lets look at the first simulation next. We already examined how the second simulation can be done in the preceding slides.

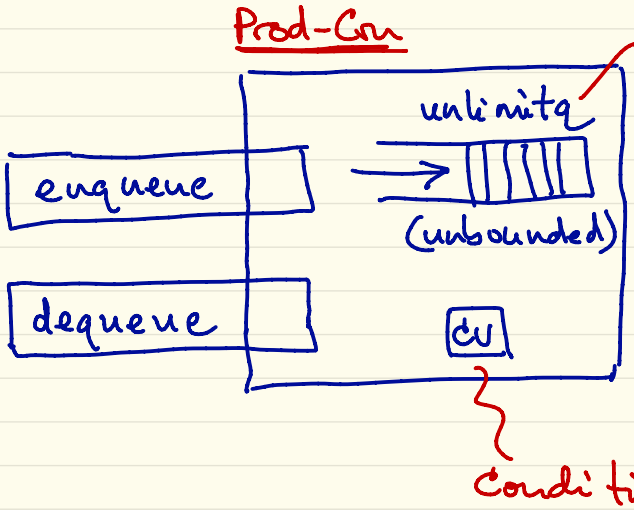
```
Monitor Semaphore {
    condvar semwait;
    int count;

    signal() {
        count++;
        if (count <= 0)
            semwait.signal();
    }
    wait() {
        count--;
        if (count < 0)
            semwait.wait();
    }
    init(int val) {
        count = val;
    }
}
```

Implement an unbounded bounded buffer producer-consumer using monitors.

With unbounded buffer, we don't have a buffer overflow problem. Just the underflow and the shared buffer update problem remain.

We need a condvar for the underflow problem and the shared buffer update problem is handled by the monitor.

So we are going to create a monitor to solve the problem.



The monitor we want to develop looks like the above. It has two methods and has one condition variable for underflow protection.

We are going to use such a monitor as follows:

```
Monitor Prod-Con pc;

Producer () {
    while (1) {
        produce() → item
        pc.enqueue(item);
    }
}

Consumer () {
    while (1) {
        pc.dequeue() → item
        consume(item);
    }
}
```

use of our monitor

use of our monitor

Now we need to implement the monitor to realize the intended operations.

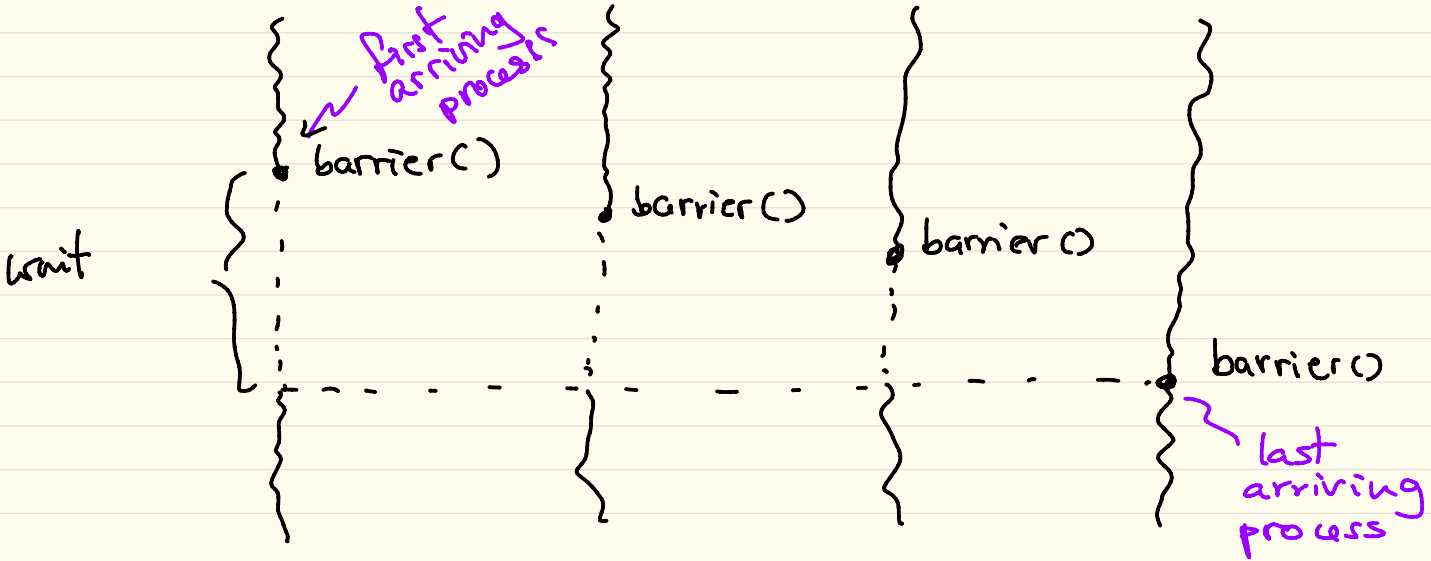
```
Monitor Prod-Con {
    condvar emptybuf;
    queue unlimitq;

void enqueue (item) {
    unlimitq.push (item);
    emptybuf.signal();
}

item dequeue () {
    if (unlimitq.size() == 0)
        emptybuf.wait();
    return unlimitq.pop();
}
```

Any process that may be sleeping are woken up – only one is woken up.

A process trying dequeue on an empty buffer goes to sleep here on the condvar emptybuf

EXERCISE: Implement a barrier() synchronization function for n processes using monitors.

Barrier() is a synchronization primitive where early arrivers are blocked until the late arrivers show up (call the barrier() function).



Your barrier() implementation can be of two different types:

- Reuseable implementation
- Non-reuseable implementation.

In the reuseable implementation, the barrier() will be in the initial state after all the synchronization of a barrier instance is complete. So it could be invoked again to perform another synchronization point.

EXERCISE: Implement a qbarrier() function with a quorum. In the barrier we saw previously, the barrier stopped all but the last process. The last process, unlocked all processes that arrived earlier.

With barrier(), we have computation phases, A and B, where we want all processes to have completed A before any one can start B. So we call the barrier() like the following:

```
A();
barrier();
B();
```

in each process that is engaged in the computing activity.

With the qbarrier(), we want to solve a slightly different problem.

We have n processes in the system. All of them are computing in phase A(). We can only enter phase B() after at least k processes have completed phase A(). The processes that arrive at the barrier the arrival of the kth process are not stopped at the barrier at all. So the barrier is open after the arrival of the first k processes.

```
A();
qbarrier (n, k);
B();
```

You could solve this problem using an existing barrier() implement.

EXERCISE: Implement a "pri tized synchronizer" using monitors.

The prioritized synchronizer psync object has two methods: pwait(), psignal().

A process can call pwait() with an argument like psync.pwait(5) to specify that it wants to wait at level 5. Levels with smaller numbers have higher priority. So Level 0 has the highest priority. Like the pwait(), the psignal() also has a priority.

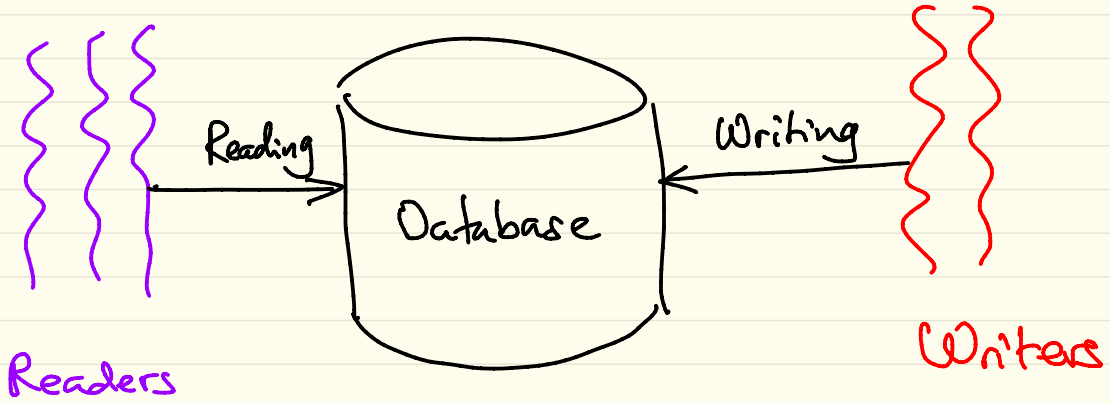
psignal(5) will release a waiting process with the highest process below Level 5. For example, if 3 processes are waiting at levels 3, 6, and 8.

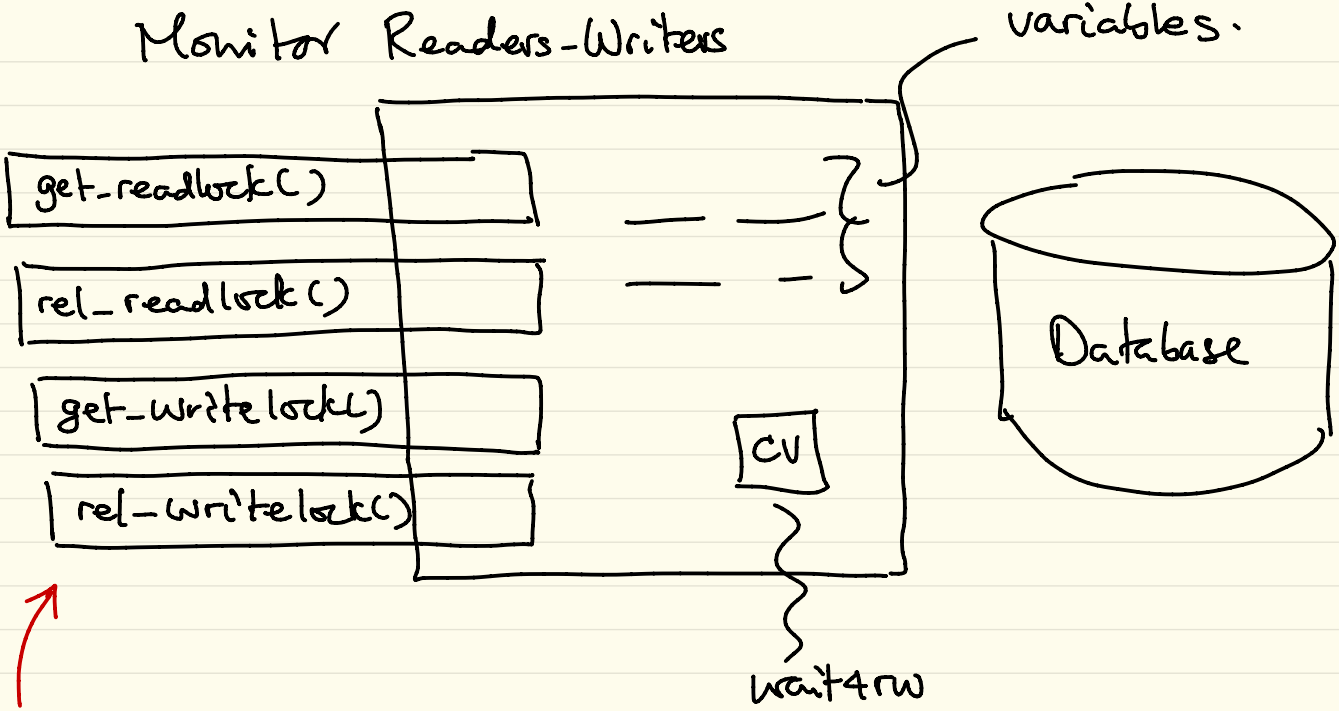
psignal (5) will release the process at level 6. So a signal at level 0 is bound to wake up a process if one is waiting. A signal at any other level can wake up no process because all of the waiting processes are at an higher level than the signalling level.

A signal has no other side effect besides waking up a process if one is waiting at the correct level.

EXERCISE: Implement the readers-writers problem with readers having priority.

This is common interaction pattern that takes place in databases. In this pattern two or more readers can access the database at the same time. However, a reader and writer cannot access at the same time. Either the reader or the writer have to wait. Same condition applies to two writers. No two writers can go at the same time. With the readers having priority, all readers who have arrived should complete before a writer is let to write. If there is a continuous inflow of readers, writers can starve.


Reading
Database
Writing
Readers
Writers



methods to be implemented as part of the Readers-Writers monitor.

```
Monitor Readers-Writers {
    int readcnt;
    bool writing, reading;
    condvar wait4rw;

    get_readlock() {

    }

    release_readlock() {

    }
```

```
    init() {
        readcnt = 0;
        reading = false;
        writing = false;
    }

    get_writelock() {

    }

    release_writelock() {

    }
```

R
- get_lock()
- Read DB
- rel_lock()

W
- get_wlock()
- Write
- rel_wlock()

```
Monitor Readers-Writers {
    int readcnt;
    bool writing, reading;
    condvar wait4rw;

    get_readlock(){

    }
    release_readlock(){

    }

    get_writelock(){

    }
    release_writelock(){

    }
}
```

```
get_readlock() {
    readcnt++;
    if (reading)
        return;
    if (readcnt == 1
        & !writing ){
        return;
    }
    wait4rw.wait()
}
```

reading = true

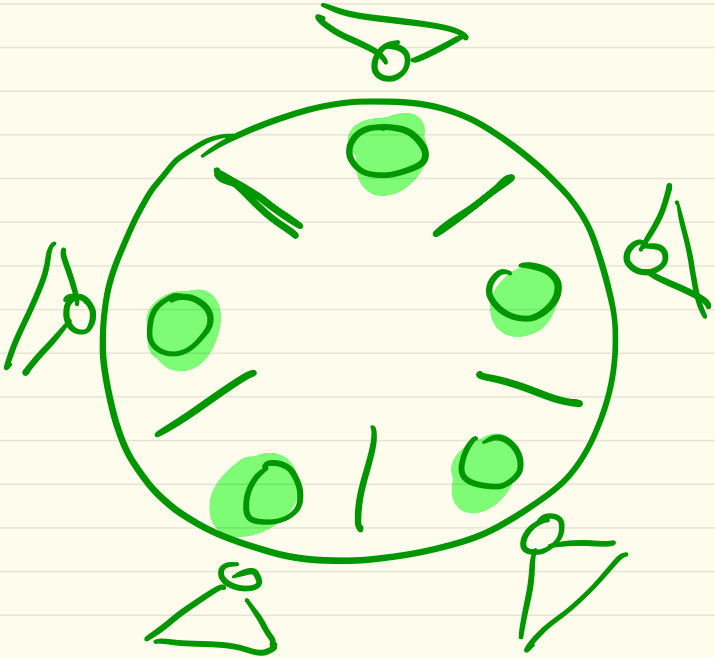
```
rel_readlock() {
    readcnt--;
    if (readcnt == 0){
        reading = false
        wait4rw.signal()
    }
}
```

```
Monitor Readers-Writers {
    int readcnt;
    bool writing, reading;
    condvar waitQrw;

    get_readlock(){
    }

    release_readlock(){
    }

    get_writelock(){
    }

    release_writelock(){
    }
```

```
get_writelock() {
    if (reading ||
        writing)
        waitQrw.lock();
    writing = true;
}
```

```
rel_writelock() {
    writing = false;
    waitQrw.signal();
}
```

Implement a monitor to solve the dining philosophers problem.



eat
think