


1. Race condition & Critical-Section

2. user-level solutions

3. OS level solutions

Process Synchronization

4. Classical Problems of Synchronization

why is it necessary?

- concurrent processes share data → can lead to →
data inconsistency
↳ data values depend on the order of instruction executions from concurrent processes ⇒ depends on when context switches occur b/w concurrent processes
- to prevent that we need mechanisms to ensure the orderly execution of concurrent processes

1. Race Condition & Critical Section

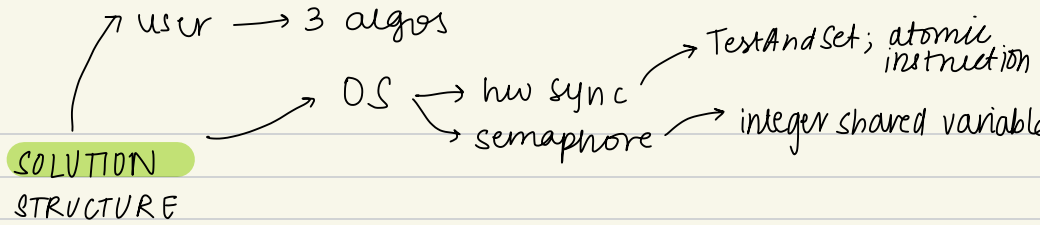
- race condition : a situation in which 2 threads are concurrently trying to change the value of a variable
- critical section : a section of code responsible for
└ changing data (basically where shared data is accessed)
└ the data is being modified
- protocol needed such that only one process can execute in its critical section at a time
↑
PROBLEM STATEMENT

SOLUTION →

Solution

1. Disabling interrupts when one process enters its critical section works only for single processor systems
↓

not feasible for multiprocessor environment as disabling interrupts can be time consuming (message must be sent to all cores), an system clock which is kept updated by the interrupts will also be affected.



entry

critical section

exit

remainder section ← non critical section
executions

PROPERTIES + ASSUMPTIONS

A. each process is guaranteed to make progress over time in the critical and remainder sections

P.

① mutual exclusion : only 1 thread or process can be executing code at once
↳ in the critical section

② progress : if no process is executing in its critical section and there exist processes that wish to enter their critical section, then the selection of the next process to enter the critical section cannot be postponed indefinitely
basically, a process not in the critical section can't block other processes from entering the critical section.

③ bounded waiting : after a process has requested to enter the critical section , other processes can enter their critical sections only a bounded number of times

2 & 3 ensure that a process isn't forever stuck in the entry section

ME : assume a process is in the critical section, can another process enter it? for data inconsistency
Prog. no process is in the critical section, will it stop another process from entering?
BW : infinite waiting in?

2. User-level Solution

see that OS intervention is needed.

Algorithm 1

shared variable int 'turn'

when $\text{turn} = i$, P_i can enter critical section

entry : while ($\text{turn} \neq i$);

exit : $\text{turn} = k$

for 2 processes, $k = \text{other process number}$
 P_0, P_1

cross-check.

→ mutual exclusion : ✓

→ progress : X, if $\text{turn} = 0$ & P_0 is in a long remainder section (while(1) loop or blocked I/O) and P_1 is done and wants to enter the critical section, it can't until P_0 does.

→ bounded waiting : ✓, assuming turn is updated in a fair manner.

algorithm 2 → Peterson's

shared variable boolean $\text{flag}[2]$

initially, $\text{flag}[0] = \text{flag}[1] = \text{false}$

$\text{flag}[i] = \text{true}$, P_i can enter

entry : $\text{flag}[i] = \text{true}$ ← show interest

while ($\text{flag}[k]$); ← wait for P_k to leave

exit : $\text{flag}[i] = \text{false}$;

→ ME : ✓

→ Progress : X, if after $P_0 \rightarrow \text{flag}[0]$, context switches and $P_1 \rightarrow \text{flag}[1] = 0$

→ BW : once $P_0 \rightarrow \text{flag}[0] = \text{true}$, P_1 cannot enter critical section

Algorithm 3

- a combination:

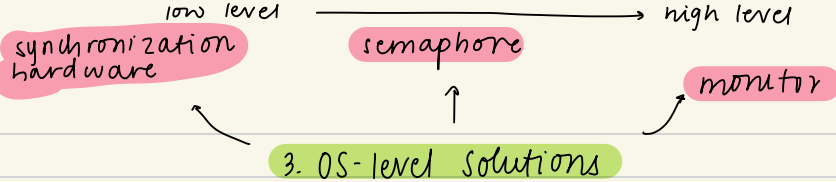
$\text{flag}[i] = \text{true} \Rightarrow P_i$ ready to enter critical section
 $\text{turn} = i \Rightarrow P_i$ can enter critical section

entry : $\text{flag}[i] = \text{true};$
 $\text{turn} = i;$
 while ($\text{flag}[k]$ and $\text{turn} = k$);
exit : $\text{flag}[i] = \text{false};$

→ ME : ✓

→ Prog : ✓

→ BW : ✓ assuming turn is updated fairly



Synchronization Hardware

- atomic hardware instructions :
 - ↳ non-interruptible (no context-switches)
 - ↳ executed in order without preemption
- TestAndSet : test and modify the content of a main memory word atomically

↓
when the operation is called, interrupt is disabled and it is enabled right before the operation is exited.

```

boolean TestAndSet (boolean *target) {
    boolean rv = *target;      ← get current value
    *target = true;           ← store true
    return rv;                ← return old value
}
  
```

- Variable shared : boolean lock ;
initially lock = false
if TestAndSet on lock returns false, then process can enter critical section

```

entry while (TestAndSet(&lock)),
exit   lock = false;
  
```

if lock = false, proceed
release lock
on entrance, lock = true.

- ME : ✓
- progress : if lock = false ⇒ none in critical sec., next process to execute TestAndSet will enter critical section
- bounded waiting : if process A is faster than B, it could keep holding the lock.

Semaphore

· widely used

· Semaphore S : integer shared variable



accessible by 2 atomic system calls

(1) $\text{Wait}(S)$: if $S > 0$, $S--$; else $\text{Wait}()$;
↳ $S \leq 0$

Wait also done
kernel & signal
operations?

enter waiting
state
or busy waiting

(2) $\text{Signal}(S)$: $S++$;

pros so, no context
switch overhead
cons : inefficient if critical
sections are long as waste of CPU time.

allows a process to use
CPU time continuously
while waiting

· 2 types :

when shared resource has many instances

Counting semaphore : integer value can range over an unrestricted domain

Binary Semaphore : integer value can never be greater than one

usually used for a lock for mutual exclusion

Variable shared : semaphore mutex; (initially mutex : 1)
if mutex = 1, process executes $\text{Wait}(\text{mutex})$ and then it enters critical section

entry : $\text{Wait}(\text{mutex})$;
exit : $\text{Signal}(\text{mutex})$;

→ no while loop as it is implemented in Wait already

→ busy wait, keeps polling value of S

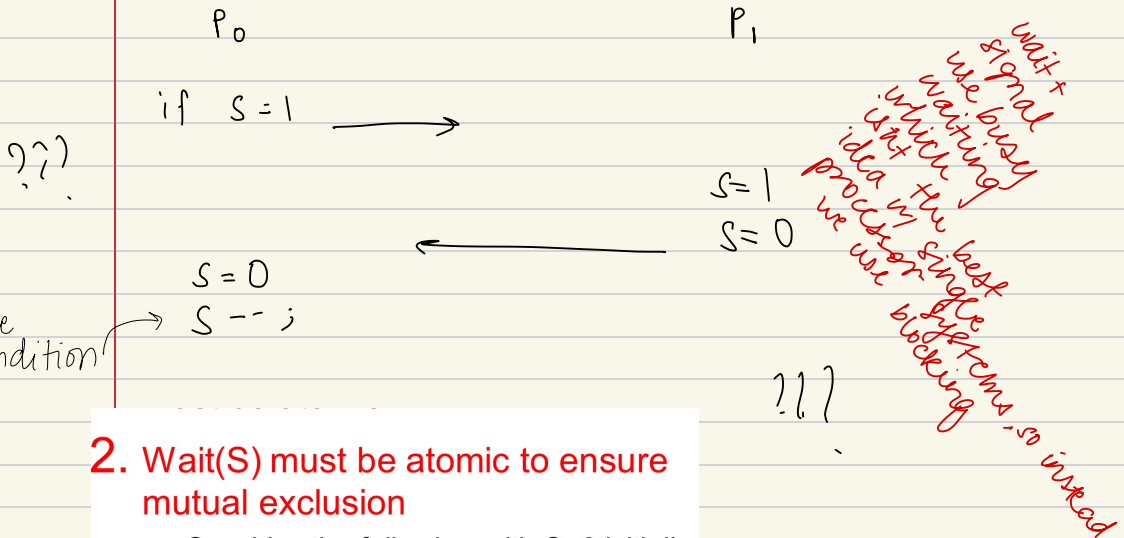
good for short time ← advantage that no context switch occurs

solution :

enters if
 $S=1$
decrements
 $S (S=0)$
increments
 $S (S=1)$

mutex Semaphore PROPERTY ANALYSIS

- proof* ↪ to avoid race condition, $S++$, $S--$ should be atomic
- Wait(S) must be atomic to ensure mutual exclusion



2. Wait(S) must be atomic to ensure mutual exclusion

- Consider the following with $S > 0$ initially
 - Process P_0 executes **while ($S \leq 0$)** and exits busy wait
 - Context switch from P_0 to P_1**
 - Process P_1 executes **while ($S \leq 0$)** and **$S--$**
 - Context switch from P_1 to P_0**
 - Process P_0 executes **$S--$**
 - Mutual Exclusion?** ☹

→ both in CS.

→ both

is blocking only for single processor?
is prev imple. only for multiprocessor?

Semaphore Blocking

Defining semaphore as a record

```
typedef struct {  
    int value,  
    struct process *L;  
} semaphore
```

queue that stores the
processes waiting on this
semaphore in the form of
a PCB list

- basic idea : when a process is in its CS, other processes wanting to enter will be blocked hence the waiting processes won't consume the critical section, unlike the busy waiting of the previous implementation

operations :

① block()

- dequeue current process from ready queue
- enqueue the process to list L
- change state to waiting

imp note :
context switch \Rightarrow shift
kernel mode \Rightarrow user mode
kernel mode \Rightarrow context switch

leads to context switch

② wakeup()

- dequeue a process from L
- enqueue the process to ready queue
- change state to ready

↳ kernel
 \Rightarrow change things
user mode
next process

picked
using
different
policies

(like in process scheduling)

entry : wait(S)
exit : Signal(S)

Implementation

wanting to enter has a separate queue (L)

so when one process is done, another is added to ready queue from L

wait(S) :

Signal(S) :

if S.val <= 0
↳ block
∴ we do S--, that condition changes to S.val < 0

S.value -- ;
if (S.value < 0) {
 block()
}

S.value ++ ;
if (S.value <= 0) {
 wakeup(P)
}

S.value indicates the number of waiting processes
(if = -2 \Rightarrow 2 waiting processes)

Atomic? Yes

Case 1

1. initially S.value = 1
2. P₀ executes S.value--;
3. Context switch from P₀ \rightarrow P₁
4. Process P₁ executes S.value-- and blocks : S.value = -1
5. Context Switch from P₁ \rightarrow P₀
6. Process P₀ also blocks
7. no progress for P₀ & P₁

Case 2

1. Initially S.value = -1 (P₂ blocked)
2. P₀ executes S.value++;
3. Context Switch from P₀ to P₁
4. P₁ executes S.value++ and exits No wakeup(P₂)
∴ S.value = 1 instead of S.value = 0
5. Context switch back to P₁ doesn't wake up ∴ S.value = 1 exits.
6. no progress for P₂

ALL UPDATES + CHECKS FOR S SHOULD BE ATOMIC

but if wait and signal are atomic then this won't be an issue?

• why `S--;` before blocking?

keep in mind: block leads to context switch

correct?

because we enter waiting state and the eventually a context switch occurs.

⇒ b/w `block()` and `S.value--`, a context switch would occur.

↓

if it does, mutual exclusion is violated

eg. `S.val = 0`

P_0 executes `block()`

P_1 executes `signal(S)` to increment `S.val` to 1, executes `wakeup(P0)`.

⇒ by making `S.val = 0`

P_2 executes `wait(S)`, locks `S`, enters critical section

P_0 executes `S.value--`; enters critical section

↳ ∴ it can't check what the value of `S.val` is due to `block()` being already executed.

1. Bounded Buffer
Producer
Consumer Problem

2. Dining-Philosophers
Problem

3. Readers & Writers
Problem

Classical Problems of Synchronization

1. Producer-Consumer with Bounded Buffer

multiple producers, multiple consumers, bounded buffer

shared resources \Rightarrow counting semaphores

here, the shared resource is buffer [n]

semaphores: full, empty, mutex \leftarrow binary for mutual exclusion to update buffer

$full = 0$ \leftarrow consumer check if empty
 $empty = n$ \leftarrow prod check if full
 $mutex = 1$

Producer

```
item nextProduced; while(1){  
    produce nextProduced;  
    wait(empty);  $\rightarrow$  consume one empty slot  
    wait(mutex);  
    add nextProduced to buffer;  
    signal(mutex);  
    signal(full);  $\rightarrow$  signal consumers  
}
```

Consumer

```
item nextConsumed; while(1){  
    wait(full);  
    wait(mutex);  
    nextConsumed = item from buffer;  
    signal(mutex);  
    signal(empty);  
    consume the item nextConsumed;  
}
```

2. Dining-Philosophers Problem

- 5 philosophers - dining OR thinking
→ only if 2 chopsticks
- 5 plates, 5 chopsticks

shared resource
hence 5 semaphores

wait(left)
wait(right)
eat
signal(left)
signal(right)
think

leads to possible deadlock

solutions

1. at most 4
simultaneously
hungry

2. Allow pickup
only if both
chopsticks are
available

3. Asymmetric
solution:
odd: left then
Phil right
even: right
Phil then left

3. Readers - Writers Problem

- Scenario : common file
 - writer needs exclusive access
 - readers can concurrently access
 - 1st reader : before reading, block writers
 - last reader : after reading, allow writers
 - readers are given preference
 - ↳ if reader reading, more readers can access
- Shared data
 - int readcount = 0 ; ← tracks no of readers in the database
 - database
- Semaphore
 - mutex = 1 ; → binary, protect access to readcount
 - wrt = 1 ; → binary, mutual exclusion to data base

Writer Process {

```
wait(wrt);
writing is performed
signal(wrt);
```

Reader Process {

```
.wait(mutex);
readcount++;
if (readcount == 1) wait(wrt-); } first reader things
signal(mutex);
reading is performed
wait(mutex);
readcount--;
if (readcount == 0) signal(wrt); } last reader things
signal(mutex)
```