1. Race condition & critical -Section

2. user-level solutions

3. OS level solutions

4. Classical Problems of Synchronization

Process 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.

user → 3 algos

OS → hw sync → TestAndSet; atomic instruction

OS → semaphore → integer shared variable

SOLUTION

STRUCTURE

$\boxed{\text{entry}}$
critical section
$\boxed{\text{exit}}$
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 cant 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 isnt forever stuck in the entry section

ME : assume a process is in the critical section, can another process enter it?

Prog. no process is in the critical section, will it stop another process from entering

BW : infinite cutting in?

for data inconsistency

2. User-level Solution

see that OS intervention is needed.

Algorithm 1

shared variable int 'turn'

when turn = i , P_i can enter critical section

entry : while (turn != i);

exit : turn = k

for 2 processes, k = other process number
P_0, P_1

cross-check.

→ mutual exclusion : ✓

→ progress : X , if turn = 0 & P_0 is in a long remainder section (eg 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 flag[2]

initially, flag[0] = flag[1] = false

flag[i] = true , P_i can enter

entry : flag[i] = true ← show interest
while (flag[k]); ← wait for P_k to leave

exit : flag[i] = false ;

→ ME : ✓

→ Progress : X , if after P_0 → flag[0], context switches and P_1 → flag[1]=0

→ BW : once P_0 → flag[0] = true, P_1 cannot enter critical section

Algorithm 3

- a combination:

flag[i] = true ⇒ P_i ready to enter critical section

turn = i ⇒ P_i can enter critical section

```
entry : flag[i] = true;
        turn = k;
        while (flag[k] and turn = k);
exit  : flag[i] = false;
```

→ ME : ✓

→ Prog : ✓

→ BW : ✓ assuming turn is updated fairly low level → high level

synchronization hardware ← 3. OS-level solutions → monitor

semaphore

Synchronization Hardware

- atomic hardware instructions :
  - ↳ non-interruptible (no context-switches)
  - ↳ executed inorder 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)) ;     if lock = false, proceed
exit    lock = false ;                   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

wait & signal also are kernel operations?

1. Wait(S) : if S>0, S--; ~~else~~ Wait(); ← enter waiting state or busy waiting

↳ S <= 0

2. Signal(S) : S++ ;

busy waiting: practice that allows a process to use CPU time continuously while waiting

pros so, no context switch overhead

cons : inefficient if critical sections are long as waste of CPU time.

- 2 types :

Counting semaphore : integer value can range over an unrestricted domain

when shared resource has many instances

Binary Semaphore : integer value can never be greater than one

usually used for a lock for mutual exclusion

solution

- Variable shared : binary semaphore mutex; (initially mutex = 1)

if mutex = 1 , process executes wait(mutex) and then it enters critical section

entry : Wait (mutex); → no while loop as it is implemented in Wait already

exit : Signal (mutex);

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

increments S (S=1)

busy wait, keeps polling value of S

good for short time ← advantage that no context switch occurs mutex

Semaphore PROPERTY ANALYSIS

proof

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

P_0		P_1
if S = 1	→	
		S = 1
	←	S = 0
S = 0		
S--;		

race condition

??)

???

wait + signal use busy waiting which isn't the best idea w/ single processor systems, so instead we use blocking

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

- Consider the following with S>0 initially
  1. Process P_0 executes **while (S <= 0)** and exits busy wait
  2. **Context switch from P_0 to P_1**
  3. Process P_1 executes **while (S <= 0)** and **S--;** → both
  4. **Context switch from P_1 to P_0** → both
  5. Process P_0 executes **S--;**
  6. *Mutual Exclusion?* ☹

in CS.

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 wont consume the critical section, unlike the busy waiting of the previous implementation

- **operations** :

imp note :
context switch ⇒ shift kernel mode
kernel mode ⇏ context switch

① block()
→ dequeue current process from ready queue
→ enqueue the process to list L
→ change state to waiting ← leads to context switch
↳ kernel
→ change things
user mode
↳ next process

② wakeup()
→ dequeue a process from L
→ enqueue the process to ready queue
→ change state to ready

picked using different policies (like in process scheduling)

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

Implementation

waiting to enter has a separate queue (L)

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

wait(S) :

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

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

Signal(S) :

if S.val <= 0 wakeup

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

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

Atomic? Yes

Case 1

1. initially S.value = 1
2. P_0 executes S.value--;
3. Context switch from $P_0 \rightarrow P_1$
4. Process P_1 executes S.value--; and blocks ∴ S.value = -1
5. Context switch from $P_1 \rightarrow P_0$
6. Process P_0 also blocks
7. no progress for P_0 & P_1

Case 2

1. Initially S.value = -1 (P_2 blocked)
2. P_0 executes S.value++ ;
3. Context switch from P_0 to P_1
4. P_1 executes S.value++; and exits No wakeup(P_2) ∵ S.value = 1 instead of S.value = 0
5. context switch back to P_1 doesnt wake up ∵ S.value=1 exits.
6. no progress for P_2

- 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(P_0). 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

Step 1.

- shared resources ⇒ counting semaphores
- here, the shared resource is buffer [n]
- semaphores : full, empty, mutex ← binary for mutual exclusion to update buffer

full = 0 ← consumer check if empty
empty = n ← prod check if full
mutex = 1

Producer

```
item nextProduced; while(1){
  produce nextProduced;
  wait (empty) ;      → consume one empty slot
  wait (mutex) ;
  add nextProduced to buffer;
  signal (mutex);
  signal (full) ;     → 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
  - dining → only if 2 chopsticks
- 5 plates, 5 chopsticks
  - 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 phil: left then right
   even phil: right 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)
```