Donia said
Ch.6 & Ch.7

Ch 6 and 7 Synchronization Tools

- mutual exclusion :- لايجب ان يدخل اكثر من process داخل ال critical sec تأخذ يعني مجرد ما كانت وحدة داخله في الحلقة الحرجة لا يجوز لغيرها بالدخول ايضا " إذا ما فيه أحد وأراد الدخول لها عندها لا يجب ان يؤجل القرار إلى اللانهائية ".

- process تلخيص الفكرة

- Bounded waiting → اذا في process أعلنت انها تريد الدخول الى critical فيجب ان يكون هناك حدا على الدخول وهذا الرقم يمنع ال starvation .

* the question → Does it solve the critical-section problem →

I check about three form →

① mautully exclusion

② progress

③ Bounded waiting

① mautual exc. ← يضمن أن عملية واحدة فقط يمكنها تنفيذ ال Critical Sec. في وقت واحد.

٢ ال progress ← اذا لم يكن هناك أي عملية داخل ال critcal sec. يجب أن تتمكن احدى العمليات التي تحاول الدخول من النجاح في النهاية

٣ Bounded waiting ← يضمن أن العملية التي تحاول الدخول إلى القسم الحرج لن تتأخر إلى أجل غير مسمى بسبب عمليات أخرى.

example

Solution using compare_and_swap

- Shared integer **lock** initialized to 0;
- Solution:

```
while (true){
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

① mautlaely exclustion →

compare-and-swap ~> one process can change the value of lock from 0 to 1 in one time.

→ another process still in busy-waiting to return the value lock to 0. ✓

② progress →

if the value of lock = 0 it mean that can any process chang the value of lock to 1 and entire the critical section. ✓

③ Bounded waiting →

the solution Does not gurntee that the process entire to critcal section →

احتمال تفوت في عدد محدود لكن في حال اذا كان النظام مزدحم هذا يعني أن رح يفوت في حالة starvation

لأن العملية Compre-and-sw لا تفترض ترتيب معين بين العمليات المتشاركة

the condition Does not satisfied

- the result :-

This solution is satisfies mutual exclusion and progress but the bounded waiting is fail → This Does not fully solve the critcal-section problem

* حرينا حل الهارد وير solution يعني ال process نفسو يضمن انه كل ال instruction يتم مرة وحدة

- Softwor solutions :-

1_ mutex Locks → هون نلتزم يضمنلنا ال process انه لحالها يمسك ال lock وتنفذ ويصير الاشي بشكل Atomaticly

↳ Taken from muatuly exclusion

عبارة عن varible يسمح لك أن تعمل release and acquire

```
while (true) {
    acquire lock

        critical section

    release lock

remainder section
}
```

هاي العملية فيها busy waiting يعني فعليا انت موجود في الانتظار بس مابتعمل اشي

mutex Lock = spin Lock

*Semaphore → فكرتها مجموعة رموز بعبر عنها من خلال الأرقام

↳ integer varible
« binary Semaphore حالة خاصة »

wait, signal → ال Semaphore بسمح لشغلتين انهم يشتغلو عليه

example from wait ():-

اصلا بنتظر ال (CS) وبيقوت على ال loop من اذا كان S=0 وبيقوت في منطقة ال busy wait وبضل لكن جد ما صارت S=1 بنعمل Decrement وبنطلع.

example from Signal():-
يعني فعلياً نفس فكرة ال wait لكن مابنعمل Decrement بنعمل increment

- Solution to the CS Problem
  - Create a semaphore “**mutex**” initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore “**synch**” initialized to 0

بدنا نضمن ان S2 رح تنفذ بعد ما تتنفذ S1

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

كيف يتم ضمان هذا الشيء :-
S2 → wait قبلها حط.
wait (synch);
S2;

وبس اخلص من S1 بعمل signal increament للمتغير الوسيط وترجع لقيمتها الاصلية وهاد الي خلي ان ال wait ما تنتظر

وتنفذ .

* يعني فعليا" هذه هي فكرة ال semaphore التي تجعل ال mutex lock يتنفذ بطريقة اخرى.

* أي process بيستنى request I/O ما بيضل داخل ال cpu تتذكر دكتور تيهل على waiting-queue العملية ـــ وهيك بيتسع لـ process الاخرى الى عندها excution حقيقي تفوت داخل ال cpu.

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

اذا ال value نزلت عن الصفر فهذا يعني ان ليس هناك مكان للتنفيذ → Decriment

لـ S->list: نحطها داخل ال waiting

ال process الي طلبت ال wait هي الي تتعمل add لنفسها ويعمل block لنفسها أيضاً.

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

S->value++ → increamt

ال process التي عملت request لـ Signal هي نفسها ال process التي تستعمل لـ Wakeup

* هذه الطريقة الخنيعة لـ Semaphor بدون busy wait

* Liveness :- كم في مؤشرات حيوية

ممكن أجزاء من ال System تتوقف أو أجزاء من النظام تعلق ـــ حضرافي ليشغل مشاريع مشكلة في حياة ال System

P0	P1
wait(s)	wait(Q)

اذا هدول التنتين أخذو بهاي الطريقة فعلياً حيكون عنا Dead lock.

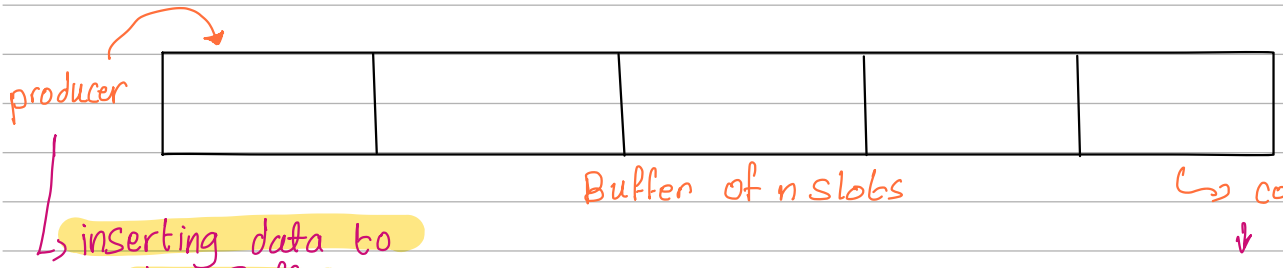
Ch.7 → Classic problems of Synchronization

« The Bounded - Buffer problem »

- The Bounded Buffer problem (producer Consumer problem), is one of the classical problem of Synchronization.

- There is a buffer of n slots and each slot is capable of storing one unit of Data.

- There are two processes running, namely, producer and consumer, which are operating on the buffer.



↳ inserting data to the Buffer

↳ removing Data from the buffer

Rols :-

1- The produce tries to insert data into an empty slot of the buffer.

2- The consumer tries to remove data from a filled slot in the buffer

3- The produce should not insert the Data in the buffer when the buffer is full

4- The consumer should not remove the Data from the buffer when the buffer is empty.

5- The producer and consumer should not insert and remove data simultoneously ← « Too »

* Solution to the Buffer problem using Semaphore :-

- we will make use of three Semaphores :-

1- m (mutex), a binary semaphore which is used to acquire and realse the Lock.

2- empty, acounting semaphore whose inital value is the number of slots

in the buffer, since, initially all slots are empty.

3- full, a counting semaphore whose inital value is 0

1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.

Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
               then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

NESO ACADEMY

((The readers - writers problem))

- A database is to be shered among several concurrent process.

- Some of these processes may want only to read the database, wheras other may want to updat (that it, to read and write) the database.

- we distingwish between these two type of process by referring to the former as Reades and the Letters as writers

- obviously, if two readers access the shared data simultaneously, no advese affectes will result.

- Howevere, if a writer and some other thread (Like Reads or writer) access the database simultanesouly, chaos may ensure.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.

*Solution to the Reades - writers problem using Semaphore :-

we will make use of two semaphore and an integer varible :-

1. mutex, a semaphore (initialized to 1) which is used to ensure mutual exclusion when readcount is updated i.e. when any reader enters or exit from the critical section.

2. wrt, is a semaphore (initialized to 1) common to both reader and writer process.

3 - readcount, an integer varible (initialized to 0) that keep track of how many process are currently reading the object.

Writer Process

```
do {
/* writer requests for critical
section */
   wait(wrt);
  /* performs the write */
   // leaves the critical section
   signal(wrt);
} while(true);
```

Reader Process

```
do {
  wait (mutex);
 readcnt++; // The number of readers has now increased by 1
 if (readcnt==1)
    wait (wrt); // this ensure no writer can enter if there is even one reader
  signal (mutex); // other readers can enter while this current reader is
                                inside the critical section
 /* current reader performs reading here */
  wait (mutex);
  readcnt--; // a reader wants to leave
  if (readcnt == 0)      //no reader is left in the critical section
     signal (wrt);       // writers can enter
     signal (mutex); // reader leaves
} while(true);
```

NESO ACADEMY

Section A: Multiple Choice Questions (10 marks)

1. Which of the following conditions is NOT a requirement for a solution to the critical-section problem?

- A) Mutual Exclusion
- B) Progress
- C) Race Condition
- D) Bounded Waiting

2. What is the primary issue with interrupt-based solutions to synchronization?

- A) They do not guarantee mutual exclusion.
- B) They can lead to deadlock in single-CPU systems.
- C) They are not scalable for multiprocessor systems.
- D) They require Peterson's solution to work.

3. In Peterson's solution, the turn variable:

- A) Indicates if the system is in a critical section.
- B) Specifies which process can enter the critical section.
- C) Prevents instruction reordering.
- D) Holds the flag values for all processes.

Section B: Short Answer Questions (20 marks)

4. Define the term race condition and provide an example from the slides where a race condition could occur. (5 marks)

5. Discuss the limitations of Peterson's solution in modern architectures. Why is a memory barrier required to ensure correctness? (5 marks)

6. Explain the difference between strongly ordered and weakly ordered memory models and their implications for synchronization. (5 marks)

7. How does a binary semaphore differ from a counting semaphore? Provide a use case for each. (5 marks)

Section A: Multiple Choice Questions (10 marks)

1. Which of the following conditions is NOT a requirement for a solution to the critical-section problem?
   - A) Mutual Exclusion
   - B) Progress
   - C) Race Condition
   - D) Bounded Waiting
2. What is the primary issue with interrupt-based solutions to synchronization?
   - A) They do not guarantee mutual exclusion.
   - B) They can lead to deadlock in single-CPU systems.
   - C) They are not scalable for multiprocessor systems.
   - D) They require Peterson's solution to work.
3. In Peterson's solution, the `turn` variable:
   - A) Indicates if the system is in a critical section.
   - B) Specifies which process can enter the critical section.
   - C) Prevents instruction reordering.
   - D) Holds the flag values for all processes.

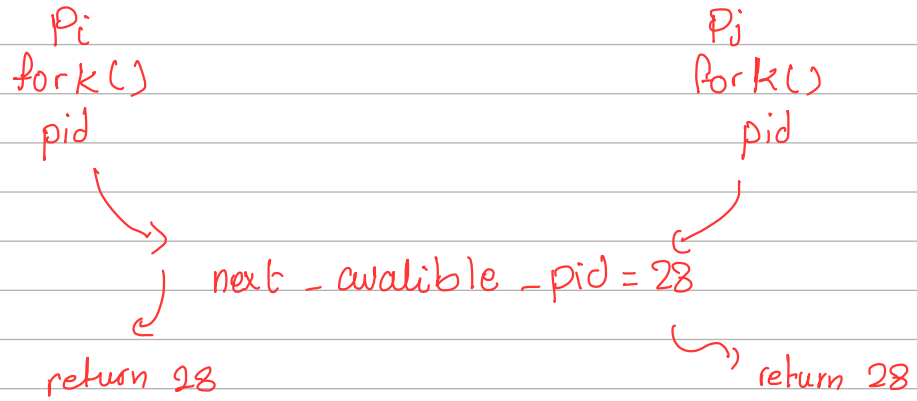
Section B: Short Answer Questions (20 marks)

4. Define the term **race condition** and provide an example from the slides where a race condition could occur. (5 marks)
5. Discuss the limitations of Peterson's solution in modern architectures. Why is a memory barrier required to ensure correctness? (5 marks)
6. Explain the difference between **strongly ordered** and **weakly ordered** memory models and their implications for synchronization. (5 marks)
7. How does a binary semaphore differ from a counting semaphore? Provide a use case for each. (5 marks)

Section C: Code Analysis (20 marks)

4. Define the term **race condition** and provide an example from the slides where a race condition could occur. (5 marks)

the race condition in Kernal with the next-~~child~~ avalible-pid wich creating a child is pid by using fork() which represent the next avalible process in pid



two process have the same value of child

5. Discuss the limitations of Peterson's solution in modern architectures. Why is a memory barrier required to ensure correctness? (5 marks)

the memory barrire use to ensure the puter solution is correct to modern arch.

the peterson's not gurentee to solution in moduren arch.

↳ in ~> improve performance, Compiler may be orderd operation that have no dependences

Does not understand why it will Does not work to usful undest work in race condition

6. Explain the difference between **strongly ordered** and **weakly ordered** memory models and their implications for synchronization. (5 marks)

Strongly orderded ~> should imidiatly visible to the other process

weakly orderded ~> should not imidiatly visble to the other process.

7. How does a binary semaphore differ from a counting semaphore? Provide a use case for each. (5 marks)

binary → integer value that does Counting only between 0,1 ~> Computer langue

use a mutex Lock!

Counting → integer value that does counting any number

models and their implications for synchronization. (5 marks)

7. How does a binary semaphore differ from a counting semaphore? Provide a use case for each. (5 marks)

Section C: Code Analysis (20 marks)

8. The following code is a synchronization solution using the `test_and_set` instruction:

```c
boolean lock = false;

void critical_section() {
    while (test_and_set(&lock)) {
        ; /* busy wait */
    }
    // critical section
    lock = false;
}
```

- Identify one potential issue with this solution. (3 marks)
- Suggest a modification to improve fairness. (3 marks)

9. Given the following `wait` and `signal` semaphore operations: