CH.6: Synchronization Tools

* Background

- Processes can execute concurrently

⇐ بما انه العمليات في منها بتم تنفيذها بنفس الوقت ممكن يأدّي الاشي هاد لـ data inconsistency
إذا كان في shared data (يعني الداتا ما تكون متناسقة مع انها من نفس المصدر)

- Concurrent access to shared data may result in data inconsistency.

⇐ منشان نحل هاي المشكلة بلزمنا آليات متبعة نحلها ونتبعها.
طبعاً مثال على المشاكل هاي هي مشكلة الـ consumer-producer.

⇐ منلاحظ shared variable هو الـ counter ⇐ بدل على عدد الـ Buffers الي فيهم بيانات

⇐ لما الـ producer يعبّي الـ Buffer بنتقل للـ Buffer الي بعده وبزيد الكاونتر واحد
طبعاً بشرط ما يتعدّى حجم الـ Buffer

⇐ الـ Consumer هو بقرأ الداتا، عا يكون الكاونتر صفر معناها ما في داتا يقرأها، واما إذا لا، فالـ consumer بحفظ القيم الي بالـ Buffer وبنقص الكاونتر واحد لأنه خلص البيانات الي في هاد المكان انقطوا

* Race Condition

هسا بالحالة الي فوق وين بتصير المشكلة؟ عا يجوا اثنين يقرؤوا قيمة الكاونتر ويعدّلوا عليها بنفس الوقت

```
. Consumer :  reg1 = counter          [ counter++ ]
              reg1 = reg1 + 1
              counter = reg

. Producer :  reg2 = counter          [ counter-- ]
              reg2 = reg2 - 1
              counter = reg2
```

reg1 = 5 | reg2 = 5
reg1 = 5+1 = 6 | reg2 = 5-1 = 4

قرؤوها بنفس الوقت صارت مشكلة

~~c = 5~~
~~c = 6~~
c = 4 → هي المفروض تكون 5

⇐ المفروض يشتغلوا بشكل منظم مش اثنين يعدلوا ويتقاطعوا ببعض

يعني بالمختصر الـ Race Cond. هو الحالة الي بتوصل فيه العمليات للداتا المشتركة وبتقوم بالتعديل عليها بصورة متزامنة (يعني العمليات بتصير تتسابق عشان تجيب الداتا)

⇐ الحل هو إنه كل عملية تنفذ بعدما العملية الثانية تقطعها

* Critical Section Problem

لو عنا سيستم فيه عدد من البروسسز وكان كل بروسس عنده critical section،

الـ critical section هي الجزئية المشتركة بين عدة بروسسز

.if Each process has critical section segment of code:

* process may be changing common variables, updating table, etc.

⇒ when one process in critical section, no other may be in its critical section.

⇐ المفروض ما يكون في بروسس بالكريتيكال سكشن، ولا بروسس تدخل هاي المنطقة إلا بإذن من الـ entry sec.، إذا كان في بروسس ممنوع بروسس ثانية تدخل.

⇒ general structure

```
do {
    entry section
        critical section
    exit section
        reminder section
} while (true);
```

* Algorithm for Process P_i

```
do {
    while (turn == j);      ⇐ إذا المفتاح مع j بنقعد نستنى j يفلت المفتاح (المفتاح الي بدخلنا ويعطينا إياه)
        critical section    ⇐ إذا لا بدخل على الكريتيكال سكشن
    turn = j;               ⇐ بعدين بنعطي المفتاح لـ j بس نخلص
        remainder section   ⇐ باقي الكود الي على Pi نفيذه
} while (true);
```

. The critical section problem: The problem of ensuring that when one process is executing in its critical section, no other processes is allowed to execute in its critical section.

* Solution to critical-section Problem (بدنا نفهم حلول نطبقها عليها الـ ٣ شروط)

. ① Mutual Exclusion

⇐ إذا البروسس Pi جوّا الـ critical sec. ممنوع ولا أي عملية تدخل هاي المنطقة

② Progress

⇐ إذا الـ sec. critical فاضية وفي بروسس بدها تدخل، لازم نتأكد إنه ماني حدا طالب الكريتيكال سكشن قبلها، إذا في بروسس اي طلبت أوّل إلها الأولوية وشي لازم إجي منها بتأجل (اي إلها الأولوية).

③ Bounded Waiting

⇐ لازم يكون في عدد محدود من مرّات الدخول للكريتيكال سكشن اي واحد (طالما طبعاً ماني حدا بالـ الكريتيكال سكشن) وإذا تجاوزت البروسس هذا العدد وقتها بضغطها لتجي بروسس ثانية تدخل وتطلع (يعني مارح نخلي بروسس تفل تستنى كتير بسبب إنه في بروسس ثانية قاعدة بتدخل وبتطلع)

* Critical-section Handling in OS

الـ approaches رح بتعتمد على نوع الـ kernal

① Preemptive : allows preemption of process when running in kernal mode.

⇐ بتسمح إنه يصير للبروسس interrupt وهو قاعد بيشتغل

② Non-preemptive : runs until exits kernal mode, blocks, or voluntarily yields CPU

⇐ إذا البروسس اشتغل رح يفل يشتغل لحد ما يخلص كل شي بايده (طبعاً هاي الحالة مافيا race condition في الكريتيكال صود.

* Peterson's Solution

↳ Two process Solution

⇐ هاد الحل بنفع في حالة يكون عنا بس عمليتين، هاد الحل بتفرض إنه يكون عنا:

⇒ Two shared variables : int turn; Boolean flag[2];

↳ Indicates whose turn is to enter the critical section

Array to indicate if a process is ready to enter the critical sec.

⇒ flag[i] = true (معناها البروسس Pi جاهزة عشان تدخل الكريتيكال سكشن)

*Algorithm for Process P_i

```
do {
    flag[i] = true;                       // entry section
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;                      // exit section
        reminder section
} while (true);
```

البروسس i بده يدخل الكريتيكال سكشن (البروسس جاهز يدخل الكريتيكال سكشن)

turn = j ← هاد بكون فوت للي P_j تدخل الكريتيكال سكشن

while (flag[j] && turn == j); ← البروسس j شغالة ودورها (لهذا البروسس i ستستنى تخلص)

flag[i] = false; ← (بس يخلص بحط ال flag تاعته false)

⇐ هنا بدنا نتأكد إذا هاد الحل لا حقق جميع الشروط الثلاثة؟

① Mutual exclusion is preserved

P_i enters CS only if:

flag[j] = false or turn = i

ال turn رح تكون إما لـ i أو لـ j مش رح تكون الإثنين سوا بنفس الوقت ⇐ يعني مستحيل العمليتين يدخلوا على الكريتيكال سكشن بنفس الوقت.

② Progress requirement is satisfied

البروسس مستحيل توقف دور مش إلها في هاد الحل لأنها بتظل واقفة عند ال loop عشان لحد ما البروسس الثانية ما تكون بدها الكريتيكال سكشن الي لحد ما يصير دورها.

③ Bounded-waiting requirement is met

البروسس الي بدها تدخل بتكون واقفة عند الوايل لوب وأول ما تطلع البروسس الثانية من بتدخل، يعني عدد المرات المسموح لكل بروسس تدخلهم ورا بعض هو مرة واحدة على الأكثر.

* الحل هاد حقق ال ٣ شروط ✓

* Synchronization Hardware

⇐ في كثير أنظمة بتوفر دعم من الهاردوير لحل هاي المشكلة.

. All solutions based on idea of locking

⇐ الطرق هاي مبدأها قائم على فكرة قفل الكريتيكال سكشن

. Uniprocessors - could disable interrupts

⇐ بالأنظمة الي فيها بروسيسور واحد بنقدر نوقف الإنتربتس على العمليات
بس بالعادة هاي العملية مش فعّالة إذا كان عنا أكثر من بروسيسور

. Modern machines provide special atomic hardware instructions.

⇐ فيه إنستركشنز معينة في البروسيسور بتمنع إنه يتم مقاطعة هاي البروسيس.

* Solution to critical-section Problem Using Locks

```
do {
    entry section ⇒ [acquire lock]
        critical section
    exit section ⇒ [release lock]
        reminder section
} while (true);
```

. بلزم القفل عشان البروسيس تدخل

. بس يخلص بترك القفل عشان هاي الثاني يأخذه

* test_and_set Instruction

```
boolean test_and_set(boolean *target){
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

⇐ هاد الفنكشن وظيفته إنه ياخذ القيمة الي دخلت بعطيها True (ويغيّرها بتغير) و برجّع القيمة الي دخلت أصلاً.

1. Executed atomically
2. Return the original value
3. Set the new value of passed parameter to "TRUE."

* Solution using test_and_set() ← (shared variable) قيمته الابتدائية False

```
do {
    while (test_and_set (&lock)); ⇒ أول عملية بترجع False بدخل الكريتيكال سكشن، وبتغير ل TRUE
        critical section
    lock = false;
        reminder section
} while (true);
```

⇐ لما تكون ال lock قيمتها false، والبروسس بدها تدخل بدخلها، وطبعاً بتغير قيمة ال lock لـ TRUE (معناها إنه في حدا شغال بالكريتيكال سكشن) فما حدا بقدر يدخل لأنه ال lock أصلاً قيمتها TRUE، بس البروسس تخلص العملية بتخلي ال lock قيمته false، والي بسكه أول بعدها هو الي بدخل، وهكذا.

* Compare_and_swap Instruction

```
int compare_and_swap (int *value, int expected, int new_value){
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

⇐ هذا الفنكشن بحفظ القيمة الأولى المدخلة في متغير اسمه temp، بعدين بتقارن إذا القيمة والقيمة المتوقعة متساويات، إذا آه بتغير القيمة للقيمة الجديدة، وبالآخر بترجع القيمة الأصلية المدخلة (اللي هي temp).

① Executed atomically
② Return the original value of "value"
③ set the variable "value" of the passed parameter "new_value" but only if "value" == "expected"

* Solution using compare_and_swap

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0);      → initially = 0
        critical section
    lock = 0;
        reminder section
} while (true);
```

⟸ بتكون بالأول قيمة ال lock تساوي 0، فالفنكشن بغيرها لواحد وبرجع 0 فالشرط بصير فولس و البروسس بتدخل الكريتيكال سكشن وبس تخلص بتفلت ال lock وبتغيرها لصفر عشان حدا ثاني يدخل على الكريتيكل سكشن بعد ما هي خلصت وهكذا

* Bounded-waiting Mutual execlusion with test_and_set

⟸ ممكن البروسس تدخل جوا الكريتيكال سكشن وتطوّل هناك وممكن ما تحقق شرط ال Bounded-waiting.

```
do {
    waiting[i] = true;          (معناها إنه البروسس بده يدخل الكريتيكال سكشن)
    key = true;
    while (waiting[i] && key)   → initially = false
        Key = test_and_set(&lock);      key = false / lock = true
    waiting[i] = false;         (دخل الكريتيكال سكشن خلص ما بستنى)
    /* critical section */
    j = (i+1) % n;              → No. of processes   (هون زي كأنه بروح يشوف البروسس اللي بعده)
    while ((j != i) && !waiting[j])
        j = (j+1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* reminder section */
} while (true);
```

[j = (i+1) % n … j = (j+1) % n]: هل البروسس اللي جنبي بده يدخل؟ إذا لأ روح شوف اللي جنبه لحد ما يلاقي بروسس بده تدخل

[if (j == i) lock = false;]: إذا ما حدا بده يدخل فرجّع ال lock فولس عشان إذا بروسس جبت تدخل بعدين تلاقي ال lock فولس

[else waiting[j] = false;]: بنعطي ز الصلاحية يدخل للكريتيكال سكشن

* Mutex Locks

بشكل عام، الحلول السابقة هي حلول معقدة، والمبرمجين بالعادة ما بقدروا يوصلولها

- OS designers build software tools to solve critical section problem ⇒ Mutex Locks
- Protect a critical section by first acquire() a lock then release() the lock.
  - Boolean variable indicating if lock is available or not.

    ← إذا كان هذا المتغير 1، إذن الـ lock مش متاح في حدا بستخدمه

    ← إذا كان 0، إذن الـ lock مفتوح ومتاح للاستخدام
- Calls to acquire() and release() must be atomic.

  ⇒ usually implemented via hardware atomic instructions.
- But this solution requires busy waiting.

  ← لما البروسيس تضل تستنى ليصير الكريتيكال سكشن متاح (بعمل delay).

  ⇒ This lock therefore called a spinlock.

* acquire() and release()

```
. acquire() {
    while (!available);
        /* wait */
    available = false;
}
```

بستنى ليصير متاح عشان ياخذه ويحوّله لغير متاح عشان ما حدا يستخدمه غيره

```
. release() {
    available = true;
}
```

بيخلّي الـ lock متاح

```
do {
    acquire lock
        critical
    release lock
        reminder
} while (true);
```

* Semaphore

- Synch. tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.

⇐ أداة مُزامنة متطورة أكثر، أفضل من الفكرة السابقة الـ Mutex بحيث إنه البروسس بتخلوها عشان يحققوا الـ synchronization.

- Semaphore S : integer variable

⇐ بس فنكشنين بقدروا يغيروا ويعدلوا على قيمة الـ semaphore

wait() and signal() ⇒ P() and V()

```
⇒ wait(S){
      while (S<=0);
        /* wait */
      S--;
  }

signal(S){
   S++;
}
```

* بتزيد قيمة الـ S بمقدار واحد

* بتنقص من الـ S واحد طالما ما وصلت قيمة الـ S للصفر

* Semaphore Usage

- Counting semaphore: Integer can range over unrestricted domain.
- Binary semaphore: Integer can range between 0 and 1.
  - ↳ same as mutex lock

⇐ السيمافور بتحل مشاكل السنكرونايزيشن المختلفة.

- Consider P_1 and P_2 that require S_1 to happen before S_2

```
synch = 0 (semaphore)
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

wait(synch); ⇒ مستحيل أنفذ S_2 إذا ما تنفذت S_1 (لأنه S_1 بتخلي synch = 1)

⇐ استخدمنا الـ semaphore هون عشان نضبط تنفيذ العمليات بالطريقة الي بدنا إياها

* Semaphore Implementation

. Must gurantee that no two processes can execute the wait() and signal() on the same semaphore at the same time.

⇐ السيمافور الوحدة ممنوع أكثر من تو بروسيس عيمسكوها بنفس الوقت.
لهيك بنحط الـ wait() , الـ signal() بالـ critical section جواتها يعني.

⇐ بهاي الحالة بوضوح بكون فيه busy waiting جوا الكريتيكال سكشن بس كتابة الكود بتكون أسهل، وعملية الانتظار نفسها قصيرة.

* Semaphore Implementation with no Busy waiting

. with each semaphore there is an associated waiting queue.

. Each entry in waiting queue has two data items:

① Value (of type integer)

② pointer to next record

. Two operations:

① Block ⇐ رح نوقف البروسيس P_1 مثلاً وأحطها بالـ waiting queue

② Wakeup ⇐ شيلها من الويتنغ كيو، نحطه بالريدي كيو

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

. Wait(semaphore *s){
    s->value--;
    if (s->value <0){
      add this process to s->list;  ( بحطها في الويتنغ كيو )
      block();
}
```

```
. signal (semaphore *S) {
    S-> value ++;
    if (S-> value <= 0) {        ← قاعدة بتستنى
        remove 1 a process P from S-> list;
        wakeup();                ← بنطلعها من الويتنغ ليو
    }
}
```

* Deadlock and starvation

Deadlock : two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting process

⇐ يعني أكثر من بروسس قاعدين بستنوا بحدث يصير من بروسس ثانية قاعدة بتستنى أصلاً

. let S & Q semaphores = 1

P_0	P_1	S = ~~1~~	Q = ~~1~~
wait(S);	wait(Q);	0	0
wait(Q);	wait(S);	stuck here	
...	...		
signal(S);	signal(Q);		
signal(Q);	signal(S);		

. starvation - indefinite blocking

A process may never be removed from the semaphore queue in which it is suspended.

⇐ يضل يأجل البروسس لأنه في بروسس ثانية أهم منه.

. Priority Inversion

scheduling problem when lower-priority process holds a lock needed by higher-priority process

. solved via priority-inheritance protocol

⇐ بروسس أقل أهمية معه ال lock الي بتاجه بروسس أعلى أهمية منه.

* Problems with semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) .. wait (mutex)

    ⟸ بالعادة القيمة بتكون 1 ولكن بوليس ما بنفع تخليها 12

  - wait (mutex) .. wait (mutex)

    ⟸ هاي الحالة بتسبب deadlock.

  - Omitting of wait (mutex) or signal (mutex) (or both)

    ⟸ لو نسيت اشي منهم أو الثنين هاد غلط جداً".

- Deadlock & starvation are possible.

* Monitors (لأنه السيمافورز أوقات بسببوا مشاكل زي الديدلوك، في عنا ⟸)

A high-level abstraction that provides a convenient and effective mechanism for process synchronization.

⟸ في حال إنه البروجرامر نقشه وهم البروسيس بطريقة غلط، فعكس يصير في مشاكل، فلهيك ال monitor موجود (مش شرط فيه، بإنها البروسس تكون بيرفكت عشان ما يصير مشاكل زي في حالة ال semaphores)

- The monitor itself programmed to ~~follow~~ provide the mutual exclusion, and when the processes wants to access some shared data, they will do it via the monitors and hence the monitor will provide the mutual exclusion in order to achieve the process synchronization.

  ⟸ لما البروسيس بدها تأثر على ~~[illegible]~~ بتوصله عن طريق ال monitor.

- Monitor contains the declaration of variables & the bodies of procedures that operates on those variables.

```
monitor monitor_name {
  // shared variables declarations
  Procedure P1 (...) {...}
  Procedure P2 (...) {...}
  Procedure Pn (...) {...}
  Initialization code (...) {...}
}
```

⟸ البروسيجرز عبارة عن عمليات بتم تنفيذها على ال shared variables

(Initialization code): بتعطي القيم الأولية للفاريبلز اللي بنتعامل معها

⇐ المونيتورز بوفروا حماية أعلى لأنه بس البروسيجرز الي جوا ال Monitor بقدروا يوصلوا للفاريبلز وما حدا غيرهم بقدر يوصل لهدول ال shared Variables.

- Only one process at a time can be active within the monitor

* Condition Variables

⇐ عشان نحقق الهدف من وجود ال monitor لازم نضيف شغلة مهمة عليه وهي ال condition construct.

condition x, y;

⇐ في بس عمليتين مسموح يصيروا على ال condition Var. وهم:

X.wait(): means that the process invoking this operation is suspended until another process invokes X.signal()

X.signal(): resumes one of processes (if any) that invoked x.wait().

لو في بروسس بدها تستخدم X وهي مش متاحة بتستخدم X.wait()، وبتضل هاي البروسس تستنى فيها لحد ما البروسس الي قاعدة بتستخدم فيها تنفذ X.signal() (والي معناها إنه X جاهزة).

* Condition Variables Choices

⇐ البروسسز ما بصيروا يتنفذوا بنفس الوقت، لهيك إذا في بروسس بتستنى ب X مثلاً وبروسس ثانية استخدمت X.signal() بس لسا بتشتغل [...] عليها شغلها، في عدة خيارات لهالإشي:

P ⇒ X.signal()   Q ⇒ X.wait()

- signal and wait: P waits until Q either leaves the monitor or it waits for another condition.
- signal and continue: Q waits until P either leaves the monitor or it waits for another condition.

* Monitor Implementation Using Semaphore

```
semaphore mutex;
semaphore next;
int next_count = 0;

wait (mutex);
    ...
    body of F;
    ...
if (next_counter > 0)
    signal (next);
else
    signal (mutex);
```

البروسس بستنى لحد ما القفل أو المفتاح يصير معه بحيث ينفذ الي بده ينفذه ، إذا البروسس بدها تستخدم الأشياء كمان مرة بتستخدم signal(next) ، إذا لا بتخلي شغلها مع ال mutex عن طريق ال signal(mutex).

* Monitor Implementation – Condition Variables

```
semaphore x_sem;   // بداية جزء
int x_count = 0;
```

لكل كونديشن فاريبل في عنا

```
x.wait():
    x_count++;
    if (next_counter > 0)
        signal (next);
    else
        signal (mutex);
    wait (x_sem);
    x_count--;

x.signal():
    if (x_count > 0) {
        next_count++;
        signal (x_sem);
        wait (next);
        next_count--;
    }
```

⇐ عيب هاد اني فاهم اشي ؟

* Resuming Processes within a Monitor

⇐ السؤال، إذا في أكثر من بروسس قاعدة بتستنى وتم تنفيذ X.signal()، أي وحدة نعطيها، نخليها تشتغل؟

- FCFS frequently not adequate.

⇐ في عنا إشي اسمه conditional-wait بكون بهاد الشكل:

X.wait(c);

بحيث إنه c هو أولوية هاي البروسس

- Process with lowest number (highest priority) is scheduled next.

* Single Resource allocation

⇐ الحصول على الريسورس إلي بدنا إياه بكون بناءً على الأولوية إلي هي أصلاً بتحدد قديش أكثر وقت العملية فيه هاد الريسورس. (بدها تستخدم)

```
R.acquire(t);
   ...
 access the resource;
   ...
 R.release();
```

⇒ R is an istance of type Resource Allocator.

```
monitor ResourceAllocator {
   boolean busy;
   condition x;
   void acquire (int time) {
          if (busy)
             x.wait (time);
          busy = TRUE;
   }
   void release() {
          busy = False;
          x.signal();
   }
   initialization code () {
      busy = False;
   }
}
```

⇐ إذا الريسورس مش متاح بتظل تستنى فيه، بس تمسكه بتخلي الـ Busy ترو طول

⇐ بس العملية تخلص بتطلعه راح المتغير.