CH.6: Synchronization Tools

* Background

- Processes can execute concurrently

⇐ بما انه العمليات في منها بتم تنفيذها بنفس الوقت ممكن يأدي الا شي هاد ل 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 ال consumer-producer.

⇐ هنا ال shared variable هو ال counter ⇐ بدل على عدد ال Buffers الي فيهم بيانات

⇐ لما ال producer يعبي ال Buffer بنتقل لل Buffer الي بعده ، بزيد الكاونتر واحد
طبعاً بشرط ما يتعدى حجم ال Buffer

⇐ ال consumer هو بقرأ الداتا ، عايكون الكاونتر صفر معناها ما في داتا يقرأها ، اما اذا
لا ، فال consumer بحفظ القيم الي بال Buffer وبنقص الكاونتر واحد لأنه خلص
البيانات الي في هاد المكان انحفظوا

* Race Condition

هسا بالحالة الي فوق وين بتصير المشكلة؟ عايجوا اثنين يقرؤوا قيمة الكاونتر ويعدلوا
عليها بنفس الوقت

- Consumer : reg1 = counter
  reg1 = reg1 + 1
  counter = reg
  [counter++]

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

reg1 = 5
reg1 = 5+1 = 6

قرؤوها بنفس الوقت صارت مشكلة

reg 2 = 5
reg 2 = 5-1 = 4

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

⇐ المفروض يشتغلوا بشكل منظم مش اثنين يعدلوا ويقاطعوا بعض

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

⇐ الحل هو إنه كل عملية تنفذ بدون ما العملية الثانية تقاطعها

* Critical Section Problem

لو عنا سيستم فيه عدد من البروسيسز وكان كل بروسيس عنده critical section،
الـ critical section هي الجزئية المشتركة بين عدة بروسيسز

. 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 بروسيس بالكريتيكال سكشن، ولا بروسيس تدخل هاي المنطقة إلا بإذن من الـ entry sec، إذا كان في بروسيس ممنوع بروسيس 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 يفلّت المفتاح ويعطينا إياه، (المفتاح الي بدخّلنا)
        critical section    ⇐ إذا لا بدخل على الكريتيكال سكشن
    turn = j;               ⇐ بعدين بنعطي المفتاح لـ j بس نخلّص
        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 نصمم حلول تنطبق عليها الـ ٣ شروط)

① Mutual Exclusion

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

② Progress

⇐ إذا الـ critical sec. فاضية وفي بروسس بدها تدخل، لازم نتأكد إنه ما في حدا طلب الكريتيكال سكشن قبلها، إذا في فالبروسس اللي طلبت أول إلها الأولوية ومش لازم إشي فيها يتأجل (اللي إلها الأولوية).

③ Bounded Waiting

⇐ لازم يكون في عدد محدود من مرات الدخول للكريتيكال سكشن اللي ورا بعض (طالما طبعاً ما في حدا بده الكريتيكال سكشن)، وإذا تجاوزت البروسس هذا العدد وقتها بنمنعها لتيجي بروسس ثانية تدخل وتطلع (يعني ما رح نخلي بروسس تضل تستنى كثير بسبب إنه في بروسس ثانية قاعدة بتدخل وبتطلع)

* Critical-Section Handling in OS

الـ approaches رح تعتمد على نوع الـ Kernel

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

⇐ بتسمح إنه يصير للبروسس interrupt وهو قاعد بشتغل

② Non-Preemptive: runs until exits kernel mode, blocks, or voluntarily yields CPU

⇐ إذا البروسس اشتغل رح يضل يشتغل لحد ما يخلص الإشي اللي بده (طبعاً هاي الحالة ما فيها race condition في الكيرنال مود.)

* Peterson's Solution

↳ Two process solution

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

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

↳ turn: Indicates whose turn is to enter the critical section

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

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

*Algorithm for Process P_i

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

entry section: flag[i] = true; turn = j; while (flag[j] && turn == j);

exit section → flag[i] = false;

flag[i] = true; (البروسس جاهز يدخل الكريتيكال سكشن) — البروسس i بده يدخل الكريتيكال سكشن

turn = j; ← هاد بكون فرصة P_j يدخل الكريتيكال سكشن

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

flag[i] = false; (بس يخلص بحط ال flag تبعته 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 ل i أو ل j مش رح تكون الاثنين 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 بتضل واقفة عند ال loop عشانك لحد ما البروسس الثانية ما تكون بدها الكريتيكال سكشن أو لحد ما يصير دورها.

③ Bounded-waiting requirement is met

البروسس اللي بدها تدخل بتكون واقفة عند الوايل لوب وأول ما تطلع البروسس الثانية من بتدخل، يعني عدد المرات المسموح لكل بروسس تدخل 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 فعالة 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 {
    acquire lock        // entry section
        critical section
    release lock        // exit section
        reminder section
} while (true);
```

. بلزم القفل عشان البروسيس تدخل

. بس يخلص بترك القفل عشان ثاني يأخذه

* test_and_set Instruction

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

⇐ هاد الفنكشن وظيفته إنه يوخذ القيمة الي دخلت ويغيرها ويحطها True بغض النظر عنها، ويرجع 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 مرة بترجع false وبدخل الكريتكال سكشن، وبتغير الـ lock
        critical section
    lock = false;
        reminder section
} while (true);
```

⇐ لما تكون الـ lock قيمتها false، والبروسس بدها تدخل بدخلها، وطبعاً بتغير قيمة الـ lock لـ TRUE (معناها إنه في حدا شغّال بالكريتكال سكشن) فما حدا بقدر يدخل لأنه الـ lock أصلاً قيمتها TRUE، بس البروسس تخلص العملية بتخلّي الـ lock قيمته false، والي بستناه 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 (الي 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 excclusion with test_and_set

⇐ ممكن البروسس تدخل جوا الكريتيكال سكشن وتطول هناك وهيك 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;   ] إذا ما حدا بده يدخل [illegible] فـ رجّع ال lock فولس عشان إذا بروسس حبت تدخل بعدين تلاقي ال lock فولس
    else
        waiting [j] = false;   ] بنعطي ز الصلاحية يدخل للكريتيكال سكشن
    /* reminder section */
}while (true);
```

(تعليق على الأسطر j = (i+1) % n ... j = (j+1) % n: هل البروسس اللي جنبي بده يدخل؟ إذا لا، أروح أشوف اللي جنبه لحد ما ألاقي بروسس بده يدخل)

* Mutex Locks

بشكل عام، الحلول السابقة هي حلول معقدة، المبرمجين بالعادة ما بقدروا يوصلوا

- 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 البروسس تظل 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 يوخذه ويحوله لغير متاح عشان ما حدا يستخدمه غيره)

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

(بس يخلص بخلي 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 lock، بحيث إن البروسس بتخلّوها عشان يحققوا الـ synchronization.

- Semaphore S : integer variable

⇐ بس فنكشنين بقدروا يغيروا و يوصلوا 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 إلي 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 بروسس عم يكولها بنفس الوقت. فبلك نحط ال wait() , ال signal() ~~بال~~ بال 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 البروسس P_i مثلاً وأحطها بال waiting queue

② Wakeup ⇐ نشيلها 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; ( نحطها في الويتنغ كيو )
      block();
}
```

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

* 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 بعض يصير مع 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 تأجل 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 الـ lock اللي بتحتاجه بروسس أعلى أهمية منه.

* Problems with semaphores

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

    ⇐ بالعادة القيمة بتكون 1 وهين بوليس، ما بنفع تخليها 2!
  - wait (mutex) .. wait (mutex)

    ⇐ هاي الحالة بتسبب deadlock.
  - Omitting of wait (mutex) or signal (mutex) (or both)

    ⇐ "لو نسيت أي منهم أو اثنينهم هاد غلط جداً."
- Deadlock & starvation are possible.

* Monitors (لأنه السيمافورز أوقات بيعمل مشاكل زي الديدلوك، في عنا ⇐)

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

⇐ في حال إنه البروجرامر نفسه فهم البروسس بطريقة خطأ، فممكن يصير في مشاكل، فهيك الـ monitor موجود (مش شرط فيه، إنما البروسس تكون بيرفكت، عشان ما يصير مشاكل زي في حالة الـ 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.

~~[illegible]~~

⇐ لما البروسس بدها أي شي بتوصله عن طريق الـ monitor.

- Monitor contains the declaration of variables & the bodies of procedures that operates on those variables.
- monitor monitor_name {
  // shared variables declarations
  procedure P_1 (...) {...}
  procedure P_2 (...) {...}
  procedure P_n (...) {...}
  Initialization code (...) {...}
  }

⇐ البروسيجرز عبارة عن عمليات بتم تنفيذها على الـ shared variables

(تعطي القيم الأولية للفاريبلز اللي بنتعامل معها)

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

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

* Condition Variables

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

condition x, y;

⇐ في بس عمليتين مسموح يصيروا على الـ 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 بصير يتنفذوا بنفس الوقت، لهيك إذا في بروسس بتستنى بـ X مثلاً وبروسس ثانية استخدمت X.signal() بس لسا بدها تكمل شغلها، في عدة خيارات لهالشي:

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

* Monitor Implementation – Condition Variables

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

لكل كوندشن 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 هاي 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 اشي 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 اللي بدنا إياه بكون بناءً على الأولوية اللي هي أصلاً بتحدد قديش أكتر وقت العملية بدها تستخدم فيه هاد الريسورس.

```
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، وبس تمسكه بتخلي الـ Busy ترو طول

⇐ بس العملية تخلص تطلعه راح المتغير.