ch.7: Synchronization Examples

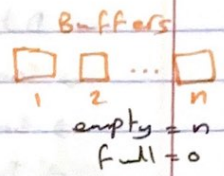
* Classical Problems of Synchronization

← في بعض المسائل في الـ synchronization :مثل

- Bounded-Buffer problem.
- Readers and writers Problem.
- Dining-Philosophers Problem.

* Bounded-Buffer Problem

- n buffers, each can hold one item.
- Semaphore mutex initialized to the value 1.
- Semaphore full initialized to the value 0.
- semaphore empty initialized to the value n.

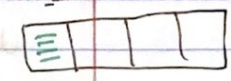


* Producer process

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait (empty);
    wait (mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal (mutex);
    signal (full);
} while (TRUE);
```

Assume n = 4
empty = 4
full = 0
mutex = 1

wait (empty); empty = 3, full = 0, mutex = 1
wait (mutex); empty = 3, full = 0, mutex = 0
/* add next produced to the buffer */ ⇒ Buffer ال



signal (mutex); empty = 3, full = 0, mutex = 1
signal (full); empty = 3, full = 1, mutex = 1

← ترتيب العمليات مهم في الكود هيك ↓

* Consumer process

```
do {
    wait (full);
    wait (mutex);
    /* remove item from buffer */
    signal (mutex);
    signal (empty);
    /* consume the item */
    ...
} while (TRUE);
```

wait (full); empty = 3, full = 0, mutex = 1
wait (mutex); empty = 3, full = 0, mutex = 0
signal (mutex); empty = 3, full = 0, mutex = 1
signal (empty); empty = 4, full = 0, mutex = 1

* Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers - only read.
  - Writers - can both read and write.
- Problem: allow multiple readers to read at the same time.

⇐ ما بنقدر ندخل reader و writer بنفس الوقت بلاش يصير خلط في البيانات
بس عادي ندخل أكثر من reader بنفس الوقت

- shared data:
  - ⇒ Data set
  - ⇒ Semaphore rw-mutex initialized to 1.
  - ⇒ Semaphore mutex initialized to 1.
  - ⇒ Integer read-count initialized to 0. ⇐ عدد الريدرز

* writer process

```
do {
    wait (rw-mutex);    rw-mutex = 0 (عشان ما يدخل writer ثاني)

    /* writing is performed */
    ...

    signal (rw-mutex);  rw-mutex = 1
} while (true);
```

* reader-process

```
do {
    wait (mutex);   mutex = 0
  [ read_count ++;   read-count = 1
  [ if (read_count == 1)  ⇐ إذا هذا أول ريدر
  [     wait (rw-mutex)   rw_mutex = 0 (عشان ما يدخل أي رايتر)
  [ signal (mutex)   mutex = 1
      ⇐ هاد الجزء بس أول ريدر بدخل عليه

    /* reading is performed */

  [ wait (mutex);   mutex = 0
  [ read count --;   read-count = 0
  [ if (read_count == 0)  ⇐ إذا هذا آخر ريدر
  [     signal (rw-mutex);   rw-mutex = 1 (اللي بده يكتب عادي يدخل)
  [ signal (mutex);   mutex = 1
      ⇐ هاد الجزء بس آخر ريدر بدخل عليه
} while (true);
```

ال mutex استخدمناها عشان ما نسمح لكل الريدرز يعبوا فيها

* Readers-writers Problem Variations

- First variation: No reader kept waiting unless writer has permission to use shared object.

⇐ الريدر مش لازم يستنى إلا في حال إنه الرايتر عنده البيرمشن إنه يستخدم السيرد داتا.

- Second variation: once writer is ready, it performs the write ASAP.

⇐ إذا الرايتر كان جاهز، لازم يأدّي عملية الكتابة في أسرع وقت ممكن.

- Both may have starvation.
  - ↳ problem is solved by kernel providing reader-writer locks.

* Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating.

⇐ القصة هون إنه عنا ٥ فلاسفة بعملوا بس شغلتين يا بفكروا أو بياكلوا، ولما ياكلوا لازم يستخدموا شوكتين أو "تشبستيكس chopsticks (عيدان). بس المشكلة إنه في بس ٥ شوك لكل الفلاسفة حدول فلازم يكون في تنسيق بالدوران.

- shared data (بالقصة هاي البروسيس عثلوا الفلاسفة & الشوك بثلوا الريسورسز).
  1. Bowl of rice.
  2. semaphore chopstick[5] initialized to 1. (يعني ما حدا بستخدمهم)

- A philosopher tries to grab a fork/chopstick by executing a wait() operation on that semaphore.
- He release his fork/chopstick by executing the signal() operation.

```
do {
    wait (chopstick[i]);
    wait (chopstick[(i+1) %5]);
      // eat
    signal (chopstick[i]);
    signal (chopstick[(i+1)%5]);
      // think
} while (TRUE
```

(على سطري الـ wait:) عشان يكونوا العيدان جنب بعض

⇐ المشكلة بهاد الألغوريثم إنه لسا ممكن يعمل deadlock، إنه مثلاً إذا كلهم مسكوا أول شوكة بنفس الوقت مش راح يقدروا يوصلوا باقي الكود.

* Monitor Solution to Dining Philosophers

- This solution imposes the restriction that a philosopher may pick up his chopstick only if both of them are available.

```
monitor DiningPhilosophers {
    enum { Thinking, hungry, EATING} state[5];
    condition self[5];

    Void pickup(int i){
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    Void putdown(int i){
        state[i] = THINKING;
        test((i+4)%5);
        test((i+1)%5);
    }

    Void test(int i){
        if ((state[(i+4)%5] != EATING) &&
            (state[(i+1)%5] != EATING) &&
            (state[i] == HUNGRY)){
            state[i] = EATING;
            self[i].signal();
        }
    }

    initialization_code(){
        for (int i=5; i<5; i++)
            state[i] = THINKING;
    }
}
```

- عنا ٣ حالات بكل فيلسوف
- بتعبر عن الفيلسوف نفسه
- إذا بدّه ياكل الشوك بكون جوعان
- أول شي بخلي الـ state HUNGRY
- بفحص إذا الشوك متاحات
- إذا طلعوا مش متاحات وما بقدر آكل بضل بستنى ليصيروا متاحات.
- إذا بدّه يخلص بخلي الشوك ويبلش يفكر
- بخلي الـ state THINKING
- بشوف اللي عن شماله إذا بدّه الشوكة
- بشوف اللي عن يمينه إذا بدّه الشوكة
- بدّه يفحص إذا الشوك متاحة أو لا
- بتأكد إنه اللي عن اليسار بوكله
- واللي عن اليمين
- وكمان هو أصلاً جوعان
- إذا تحققت الشروط بروح الحالة
- الحالة الافتراضية إنه كلهم يفكروا قاعدين يفكروا.