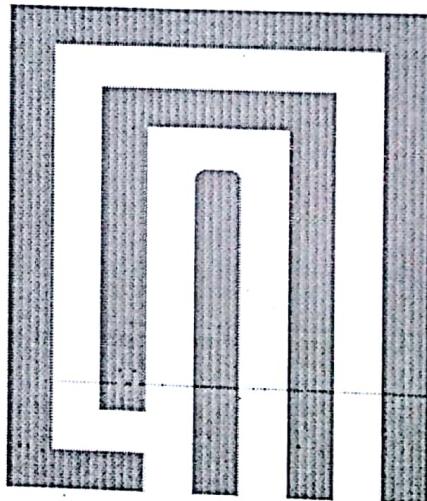


كلية الهندسة المعلوماتية

نظم تشغيل 1

المحاضرة التاسعة (تزامن الإجرائيات 2) عدد الصفحات : 29

السنة الثالثة



GATE

Greatness Achieves Through Excellence

الزراعة - مقابل باب السكن الجامعي - شارع مسايا

٠٤١ ٢٤٩٦٢٣٧ ٠٥٩٦٠ ٠٦٦ ٠١٦



www.facebook.com/groups/ITech.GATE

حلول مشكلة المقطع الحرج

1 حلول برمجية Software Solutions

وهي عبارة عن خوارزميات ، بحيث أن صحة correctness هذه الخوارزميات لا يتعلّق بأي افتراضات assumptions أو قيود أخرى (وجود قدرات أو مواصفات خاصة بالحاسوب أو أي شيء خارجي) .

2 حلول من قبل نظام التشغيل Operation System Solutions

نظام التشغيل يؤمّن بعض التوابع والخدمات وبني المعطيات التي تساعد المبرمج في العمل .

3 حلول عتادية Hardware Solutions

هي حلول تعتمد على بعض التعليمات الخاصة بلغة الآلة (تعتمد على قدرات حاسوبية محددة) مثل تعليمات Semaphores وـ Mutex test_and_set()

الآن يجب الإشارة إلى أهمية الإجابة عن الأسئلة الآتية :

- ما المقصود بالإجرائيات المعاونة ؟ وكيف تم تنظيمها ؟
- عدد نماذج تواصل الإجرائيات IPC ؟ ما الفرق بين هذه النماذج ؟
- ما هي حالة السباق ؟ ومتى تحدث ؟ ولماذا تحدث أصلاً ؟
- ما المقصود بالمقطع الحرج ؟ وما هو السبب الرئيسي في وجوده أصلاً ؟
- ما هي المقاطع الأربعية التي يجب أن تتألف منها كل إجرائية تسعى للعمل وفق البروتوكول المنشود ؟
- ما هي الشروط التي يجب أن يتحققها الحل المقترن لمشكلة المقطع الحرج حتى يكون حلاً مقبولاً ؟

☺ إن لم تستطع الإجابة عن هذه الأسئلة عد إلى المحاضرة السابقة ☺

نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)



حلول مشكلة المقطع الحرج البرمجية Software Solutions

يوجد لدينا 3 خوارزميات لكننا سوف نتعرف على خوارزمية واحدة فقط لكي نفهم الفكرة.

الخوارزمية الأولى (الأدوار)



من أجل إجرائيتين الأولى اسمها P_i والثانية اسمها P_j (أو إذا أردت P_0 و P_1)

لدينا مت حول صحيح $turn$ تشارك فيه الإجرائيتين بحيث يحدد الإجرائية

التي يجب أن تدخل مقطعها الحرج ويتم تحديده بالصفر (i) أو الواحد (j).

إذا كان $turn = i \Leftarrow$ حان دور الإجرائية P_i لتنفذ مقطعها الحرج

$turn = j \Leftarrow$ حان دور الإجرائية P_j لتنفذ مقطعها الحرج

do {

while ($turn == j$) ;

/*CS*/

$turn = j;$

/*RS*/

} while (true) ;

Process Pi

do {

while ($turn == i$) ;

.. /*CS*/

$turn = i;$

/*RS*/

} while (true) ;

Process Pj

نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)



سنتحقق بالنسبة للإجرائية Pi :

إن ; $j = \text{turn}$ تعني أنه طالما دور الإجرائية Pj انتظري يا Pi (نقذى التعليمية الحالية ;)
 وإن ; $j = \text{turn}$ تعني الآن أصبح دور الإجرائية Pj بما أنني أخفيت تنفيذ مقطعي الحرج .

إيجابيات وسلبيات الخوارزمية الأولى (الأدوار) :

☺ تتحقق الشرط الأول (الاستبعاد المتبادل Mutual Exclusion)

لاحظ أنه في لحظة واحدة ، فإن المتغير turn إما يحمل القيمة i أو يحمل القيمة j ...
 وبالتالي في لحظة واحدة لدينا إجرائية واحدة (إما Pi أو Pj) تنفذ في مقطعيها الحرج .

☹ لا تتحقق الشرط الثاني (التقدم Progress)

لأن هذا الحل يتطلب تناوب ثابت للإجراءات لتنفيذ المقطع الحرج

أي أن Pi تنفذ ثم Pj ثم Pi ثم بهذا الترتيب فقط \Leftarrow

لا يمكن أن تنفذ Pi مرتين على التبالي أو أكثر بنفس الأمر بالنسبة ل Pj

مثلاً إذا كان $\text{turn}=i$ (أي أنه دور الإجرائية Pi الآن) لكن Pi لا تريد أن تنفذ الآن !!!

وكانت Pj تريد دخول المقطع الحرج الخاص بها \Leftarrow الإجرائية Pj لا تستطيع أن تنفذ المقطع الحرج وذلك

لأنه دور الإجرائية Pi وليس دور الإجرائية Pj ☹

☺ تتحقق الشرط الثالث (الانتظار المحدود)

لاحظ أنه بعد أن تطلب مثلاً الإجرائية Pi الدخول إلى المقطع الحرج الخاص بها ...

فإنما ستنتظر حتى تدخل الإجرائية Pj مرة واحدة فقط إلى مقطعيها الحرج وبالتالي اقتبس من المحاضرة السابقة " يوجد حد على عدد المرات التي يسمح فيها لإجراءات بدخول مقاطعها الحرجة ... " وهذا محق هنا .

نظم تشغيل المحاضرة التاسعة (تزامن الإجرائيات 2)

مشاكل الحلول البرمجية

سوف نتعرف على مشكلة خطيرة في الحل البرمجي بشكل عام

وهي أن الإجرائية المتطرفة تنفذ حلقة while تحتوي على التعليمية الحالية (الفاصلة المنقوطة) ;
لكي تنتظر الإجرائية الأخرى التي تنفذ مقطعها المخرج ...

بالتالي أثناء تنفيذ هذه التعليمية الحالية التي تظن أنها بسيطة فإنها يجعل المعالج مشغولاً busy !!!

استهلاك في زمن المعالج وقدرته (?)

إن الحلقة ; while(condition) ;

نشير إليها بـ Busy Waiting أي الانتظار المشغول .



الحلول العتادية

إن معظم الحواسيب تؤمن دعم عتادي من أجل حل مشكلة المقطع المخرج critical section ، بحيث

أن كل هذه الحلول تعتمد على فكرة وحيدة وهي الإقفال locking

سوف نفهم فكرة الإقفال بالسلسل المنطقي ، انتبه جيداً :

من أجل كل متحول مشترك shared ← لدينا مقطع حرج خاص بكل إجرائية يمكن تعداً عليه .

ـ من أجل كل مقطع حرج خاص بمحول مشترك ما ← لدينا قفل lock خاص به .

ـ كل إجرائية تريد أن تدخل إلى المقطع المخرج ← تقوم بغلق هذا القفل (تحجزه) locking

ـ وأناء تنفيذها في المقطع المخرج يكون القفل مفتوحاً ولا يمكن لأحد أن يأخذه لأنه مغلق !!!

ـ كل إجرائية تريد أن تخرج من المقطع المخرج ← تقوم بفتح هذا القفل (تحريره) unlocking

ـ سامعة بذلك للإجراءات الأخرى التي تريد أن تدخل إلى المقطع المخرج بأن تأخذه وتغفله من جديد .

نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)

انتبه !!! عدد المقاطع الخرجية يتتناسب طرداً مع عدد المتغيرات المشتركة ... shared variables

وليس مع عدد الإجرائيات المتعاونة !!! cooperating

إن الأجهزة الحديثة تعلم لنا تعليمات خاصة على مستوى لغة الآلة ...



ميزة هذه التعليمات أنها ذرية atomic (دقيقة جداً) !!!

ذرة \Leftarrow لا يمكن فصلها إلى أجزاء أصغر \Leftarrow لا يمكن مقاطعة المعالج (تنفيذ تعليمية أخرى) أثناء

تنفيذ هذه التعليمية الذرية أي أنها not interruptible (غير قابلة للمقاطعة) لا يلعن متعدد ...

\Leftarrow تُلغي حق الشفاعة preemption (لا يمكن إخراجها إجبارياً).

الشكل العام للإجرائية باستخدام حل الإقفال:

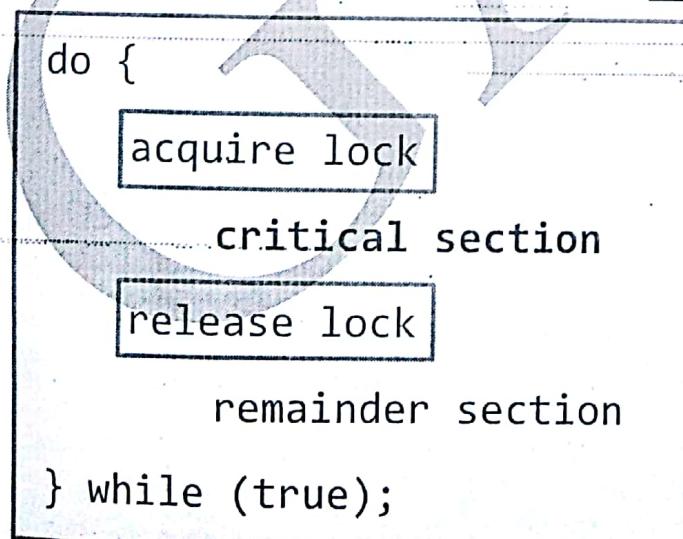
كل إجرائية Pi تعمل وفق بروتوكول الإقفال lock يجب أن تتألف من 4 مقاطع :

1 _ حجز القفل acquire lock : يتم الحصول على قفل (حجز).

2 _ المقطع الخرج critical section : تنفذ فيه تعديل على المتغير المشترك shared.

3 _ تحرير القفل release lock : يتم ترك القفل (تخمير).

4 _ المقطعباقي remainder section : تعليمات غير متعلقة بالمتغيرات المشتركة.



بالنالي لا بد من متغير مشترك إضافي يعبر عن حالة القفل (مدبوغ أو مدرر)

الحل العتادي باستخدام ال Mutex

إن مصممي أنظمة التشغيل قاموا ببناء أدوات برمجية (توابع مفيدة) يستطيع المبرمج استخدامها في برنامجه ...

أبسط هذه الأدوات هي ال mutex lock (قفل الاستبعاد المتبادل) حيث أن :

مجرد اختصار فقط

mutex = mutual exclusion

لدينا التابع acquire يستخدم لحجز القفل ، والتابع release يستخدم لتحرير هذا القفل .

حيث أن استدعاء هذه التابع (تنفيذها) يجب أن يكون ذري atomic يعني إن كان المعالج ينفذ تعليمة release أو acquire من أجل إجرائية ما \Leftarrow لا يمكن مقاطعته في هذه الأثناء .

((يتم استخدام تعليمات ذرية أبسط atomic instructions لتحقيق ذلك))

لدينا في ال mutex lock متاح بوليفي اسمه available يعبر عن توفر القفل ...

\Leftarrow available = false القفل غير متوفّر (محجوز)

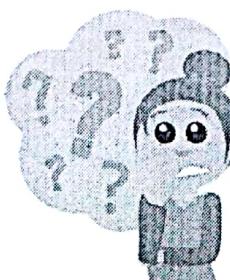
\Leftarrow available = true القفل متوفّر (متحرّر)

مبدأ العمل :

إذا كان القفل متوفّر "true" وبالتالي تستطيع الإجرائية دخول مقطعيها الحرج ويجب أن تجعله false أثناء دخولها (تنفيذ مقطعيها الحرج) ... وبعد انتهاء تنفيذ المقطع الحرج يجب أن تحرّر القفل أي يصبح القفل

متوفّر وبالتالي يجب أن تجعله true \Leftarrow لكي تسمح لإجرائية أخرى بدخول مقطعيها الحرج وهكذا ...

$$\begin{array}{r}
 \boxed{A} + \boxed{B} = \boxed{13} \\
 + + \\
 \boxed{C} \div \boxed{D} = \boxed{9} \\
 = = \\
 \boxed{7} \quad \boxed{10}
 \end{array}$$



حاول أن تخمن قيم المتغيرات

A , B , C , D

• التابع : acquire()

سوف يقوم بهمتيين : [1] انتظار توفر القفل .
[2] حجز القفل عند توفره .

```
void acquire () {  
    while ( ! available )  
        available = false ;  
}
```

شرط استمرار أكلقته هو كون المتبول
false قيمة available

• التابع : release()

سوف يقوم بمهمة واحدة وهي تحرير القفل .

```
void release () {  
    available = true;  
}  
... أطلق يد يا ... حرب
```

```
do {  
    acquire();  
    /* CS */  
    release();  
    /* RS */  
} while (true);
```



نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)

توضيح خطوات العمل :

يبدأ المتحول المشترك `available` بالقيمة الابتدائية `true` (`طازاً برأيك؟`) ...

إن أول إجرائية تريد أن تدخل إلى مقطعها الخرج فإنما ستتفقد أولاً التابع (`acquire()`) ، حيث أن قيمة المتحول `available` الآن تساوي `true` وبالتالي لن تنتظر أي شيء (شرط حلقة `while` غير محقق) .

وستقوم يجعل قيمة المتحول `available = false` ... الآن أي إجرائية تريد الدخول إلى مقطعها الخرج فلن تستطع لأنها ستتدخل في حلقة انتظار مشغول حتى تنتهي الإجرائية الأولى من تنفيذ مقطعها الخرج وتحرر القفل .

الآن الإجرائية التي تتفقد مقطعها الخرج انتهت من التنفيذ \rightarrow ستتفقد (`release()`) وبالتالي يجعل قيمة المتحول تساوي `true` من جديد ساخنةً بذلك لأحد الإجرائيات بدخول مقطعها الخرج وهكذا ...

ملاحظة غير مطلوبة : هناك حل عتادي باستخدام تعليمات (`test_and_set()`) لم يعطى هذه السنة !

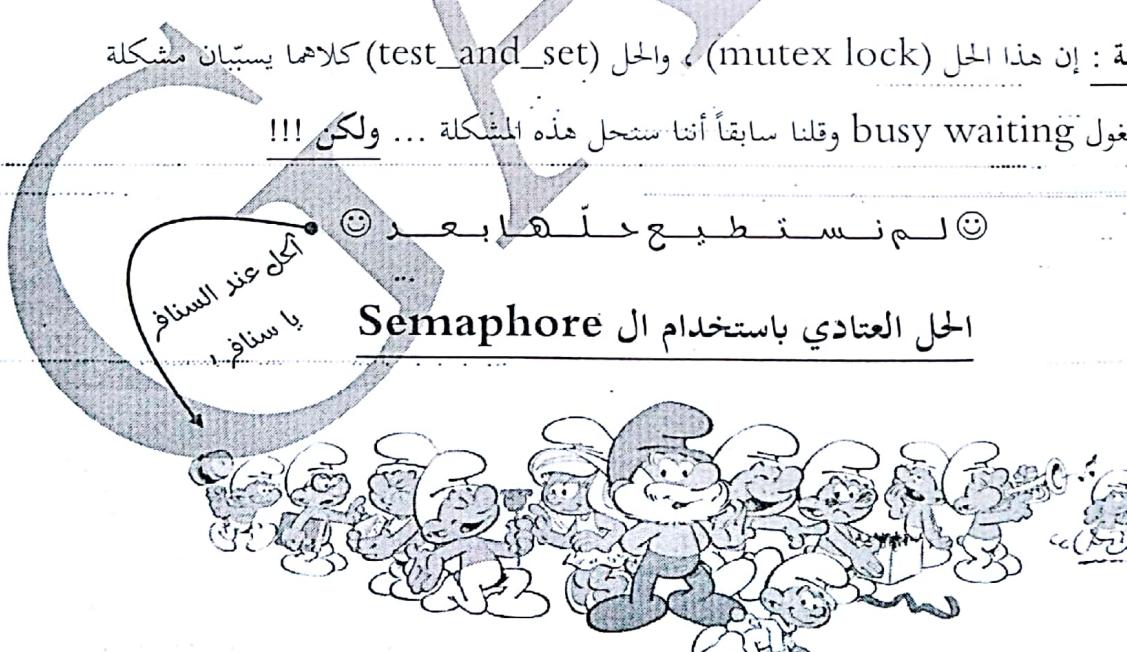
وهو حل يشبه ال `mutex lock` ولكن مشكلته أن المبرمج لا يستطيع استخدام هذه التعليمية بحيث أن نظام التشغيل هو الوحيد الذي يستطيع تنفيذها 😊

ملاحظة هامة : إن هذا الحل (`mutex lock`) ، والحل (`test_and_set()`) كلاهما يسببان مشكلة

الانتظار المشغول busy waiting وقلنا سابقاً أنها ستحل هذه المشكلة ... ولكن !!!

لم نستطيع حلها بعد 😊

الحل العتادي باستخدام ال Semaphore



جاءت الـ mutex بعد الـ test_and_set لحل مشكلة المبرمج (لكي يستطيع استخدامها) ...

الآن جاءت الـ semaphore بعد الـ mutex لحل مشكلة الانتظار المشغول

السيمافور : Semaphore

هو أداة تزامن تستخدم حل مشكلة المقطع الحرج ، وهو عبارة عن متغير صحيح integer اسمه S مثلاً يمكّنا التعامل معه من خلال عمليتين فقط هما :



`signal(S)`

&&

`wait(S)`

إن العمليتين `wait(S)` و `signal(S)` هما عمليتان ذريتان atomic ، بالإضافة إلى أنه في لحظة ما لا يمكننا تنفيذ سوى عملية واحدة فقط (إما `wait` أو `signal`) من أجل السيمافور نفسه.

يعتبر التعامل مع السيمافور أقل تعقيداً من التعامل مع إقفال الاستبعاد المتبادل mutex .

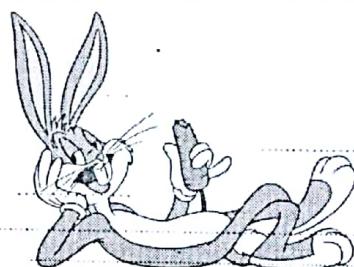


انتظار مشغول حتى يصبح المتغير S قيمته 1

التابع : `wait(S)`

```
void wait (int S) {
    while ( -S <= 0 ) ;
    S--;
}
```

إنقاص



التابع : `signal(S)`

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

زيادة

ملاحظة : تسمى العملية `V()` بـ `wait()` والعملية `P()` بـ `signal()`

نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)

وتكون الخوارزمية المشودة كما يلي :

```
do {
    wait(S);
    /* CS */
    signal(S);
    /* RS */
} while (true);
```



فيّز نوعين من السيمافورات :

1_ سيمافور عددي counting

وهو عبارة عن عدد صحيح تكون قيمته ضمن المجال $[0, n]$ حيث أن n عدد صحيح محدد.

2_ سيمافور ثنائي binary

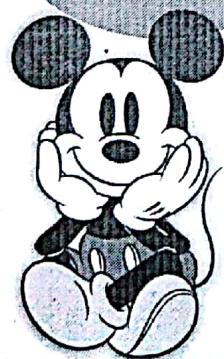
وهو عبارة عن عدد صحيح يأخذ القيمتين 0 أو 1 فقط (حالة خاصة من السيمافور العددي).

الآن سوف نتعامل مع السيمافور الثنائي ...

لاحظ أن السيمافور S لم أخذ القيم ضمن المجال $[1, 0]$ وبالتالي هذا يعني ...

قبل أن نستطيع معرفة ماذا يعني هذا ...

يجب أن نفهم ما الذي يحدث عند استدعاء التابع !!! wait



مثال توضيحي لفهم ماذا يحدث عند استدعاء التابع : wait

ليكن لدينا مثلاً إجرائيين P1 و P2 ولتكن القيمة الابتدائية للسيمافور $S=1$

ستأتي P1 وتتفقد أولاً $wait(S)$ وبالتالي ستختبر الشرط ($S \leq 0$)

وسيكون غير محقق وبالتالي لن تنتظر وستقوم بتنفيذ $S=0$ ويصبح لدينا $S=0$

الآن ستتمكن P1 من تنفيذ CS الخاص بها

الآن ستأتي P2 وتتفقد $wait(S)$ وبالتالي سيكون الشرط ($S \leq 0$) متحقق وبالتالي ستنتظر

الآن تكون P1 قد أنهت تنفيذ مقطعها الحرج CS

الآن ستتفقد $signal(S)$ ويصبح لدينا $S=1$

بالعودة إلى الإجرائية P2 التي كانت تنتظر

يكون الشرط ($S \leq 0$) غير متحقق

ينتهي الانتظار وتتفقد $S=0$ أي يصبح $S=0$

بحيث تستطيع الإجرائية P2 أن تنفذ مقطعها الحرج بأمان واطمئنان ☺

لو كان لدينا ثلاثة إجراءيات P1, P2, P3 وكان $S=1$ أيضاً

الإجرائية P1 ستتفقد $wait(S)$ وبالتالي $S=0$ (P1 تستطيع أن تنفذ مقطعها الحرج)

الإجرائية P2 ستتفقد $wait(S)$ وبالتالي ستبقى منتظرة ...

الإجرائية P3 ستتفقد $wait(S)$ وبالتالي ستبقى منتظرة ...

عندما تنتهي P1 من تنفيذ مقطعها الحرج CS عندها تستطيع إما P2 أو P3 أن تنفذ CS الخاص بها

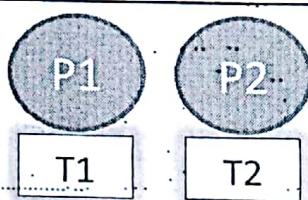
نستنتج بأن : عندما تكون قيمة $S=1$ هذا يعني أنه يمكن لـ إجرائية واحدة في لحظة ما أن تنفذ المقطع

الحرج الخاص بها \Leftarrow السيمافور الثنائي **binary mutex** يكفي قفل

ملاحظة هامة : إن الشرح لهذا لفهم عمل **Wait** وما هي القيم التي سيعاذ لها السيمافور **5** وهذه

ليس كذلك العامة الصيغة لكن لكي نستوعب الموضوع قليلاً كان لا بد من التوقف هنا ☺

نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)



مثال سريع (الشرح مهم جداً) :

ليكن لدينا الإجرائيات المعاونة $P1$ و $P2$...
 $P1$ تستطيع تنفيذ التعليمية $T1$ والإجرائية $P2$ تستطيع تنفيذ التعليمية $T2$.
الآن نحن نريد تنفيذ التعليمية $T1$ ثم التعليمية $T2$ بالترتيب تماماً. يعني أننا لا نريد أن تنفذ $T2$ قبل $T1$ أو
أن تنفذ $T2$ أثناء تنفيذ $T1$ ، نحن نريد أن تنفذ $T1$ وبعد أن تنتهي نريد تنفيذ $T2$ مباشرة ☺

السؤال هنا : كيف نستطيع فعل هذا بالاستفادة من السيمافورات ؟؟؟

لاحظ أنه لا مشكلة في حال $P1$ نفذت $T1 \leftarrow$ لن نضيف شيء إلى كود الإجرائية $P1$.
لكن المشكلة عند الإجرائية $P2$ أنها قد (رَكِزَ على الكلمة قد) تنفذ $T2$ قبل أو أثناء تنفيذ الإجرائية $P1$.

للتعميلية $T1 \leftarrow$ نحتاج أن نمنع الإجرائية $P2$ من تنفيذ $T2 \leftarrow$ نحن بحاجة إلى سيمافور ولتكن اسمه S
بحيث أنها سنضيف التعليمية $; wait(S)$ في أول سطر في $P2$ قبل تنفيذ التعليمية $T2$...

لكن السؤال هنا ما هي القيمة التي سنبدأ بها المتتحول S هل 1 أم 0 وماذا؟؟؟ انتبه !

لاحظ أنه لو بدأنا بالقيمة 1 لكن شرط الحلقة في التابع $wait$ غير محققاً \leftarrow لن تنتظر شيئاً ⚡
بالتالي يجب حتماً أن يبدأ السيمافور S بالقيمة الابتدائية 0 لكي نمنع الإجرائية $P2$ من تنفيذ أي شيء
حتى إشعار آخر \leftarrow الآن الإجرائية $P2$ تقوم بالانتظار المشغول ...

لكن إلى متى ستبقى $P2$ تنتظر ؟ إلى أن تنتهي الإجرائية $P1$ تنفيذها للتعليمية $T1 \leftarrow$ لا بد من حركة

نقوم بها لكي نخرج الإجرائية $P2$ أن $P1$ قد أنتهت التنفيذ (اللبيب من الإشارة يفهم).
بالتالي نجعل الإجرائية $P1$ ترسل إشارة $signal(S)$ بعد تنفيذ التعليمية $T1$ عن طريق استدعاء (S) $signal$ بال التالي

تردد قيمة المتتحول الواحد ليصبح $1 = S$ وبالتالي شرط الانتظار لدى الإجرائية $P2$ أصبح غير محققاً ⚡

\leftarrow تخرج $P2$ من الانتظار المشغول وتستطيع تنفيذ التعليمية $T2$ وبذلك تكون قد حصلنا على تنفيذ

مرتب للتعليميات $T1$ و $T2$ بالاستفادة من السيمافور S والعمليات $wait$ و $signal$.



السيمافور بالحالة العامة بدون الانتظار المشغول

Semaphore implementation without busy waiting



لاحظ أنه في المثال السابق ذو الثلاث إجرائيات ، كان هناك حلقة مفقودة ...

وهي أنه حين تنتهي الإجرائية P1 من تنفيذ مقطعها الحرج ثم تنفذ signal(S) بعدها

من هي الإجرائية التي يجب أن تنفذ P2 أم P3 علمًا أن P2 هي التي أتت أولاً؟ → الحلقة المفقودة

إن الجواب يعتمد على الصدفة 😊 فحسب الإجرائية التي ستخرج من الانتظار المشغول (حلقة while) بعد أن تكون الإجرائية P1 قد نفذت signal(S) هي التي تستطيع الفاصل إلى مقطعها الحرج CS وبالتالي إما P2 أو P3 (لا نعرف بالضبط).

On ne sait pas ???

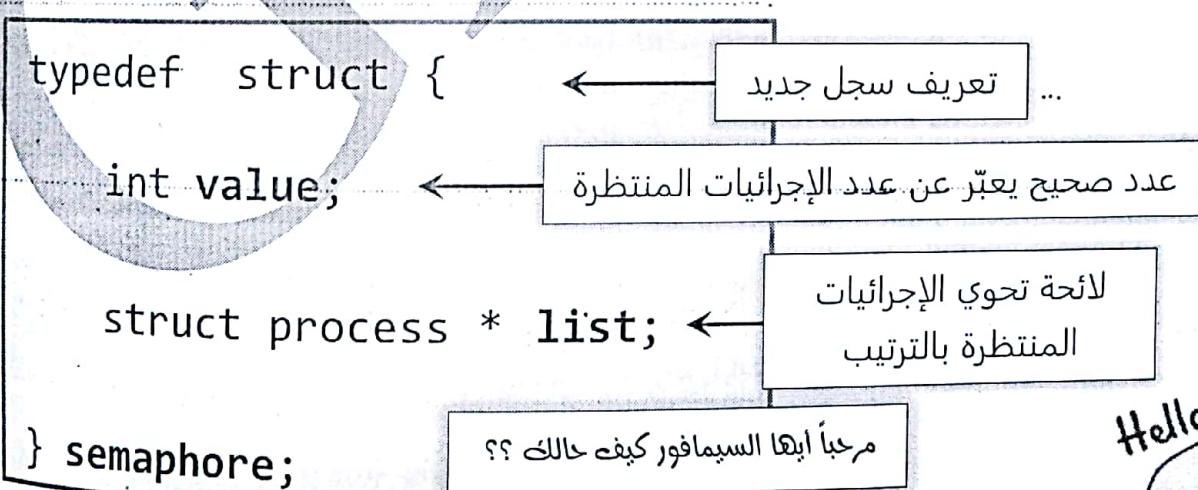
في العلم ... لا يوجد شيء اسمه لا نعرف !!!

ومن أجل ذلك (ومن أجل الانتظار المشغول) سوف نعرف السيمافور بطريقة مختلفة كلية ...

كل سيمافور S سيحتوي على شيئين هما (أصبح سجل struct):

عدد صحيح value يعبر عن القيمة العددية للسيمافور ، بالإضافة إلى رتل انتظار waiting queue

اسم list يعبر عن الإجرائيات المنتظرة (لائحة متربطة).



المتحول value عبارة عن عدد (يبدأ بالقيمة 1) بحيث أن هذا العدد يُعد بشكل عكسي !!!

اللائحة list عبارة عن رتل queue (يبدأ فارغة) بحيث يتم ترتيب الإجرائيات على مبدأ FIFO

`list = null`

`&&`

`value = 1`

أ_ عندما تريد إجرائية الحصول على قفل (تنفذ ... wait ...)

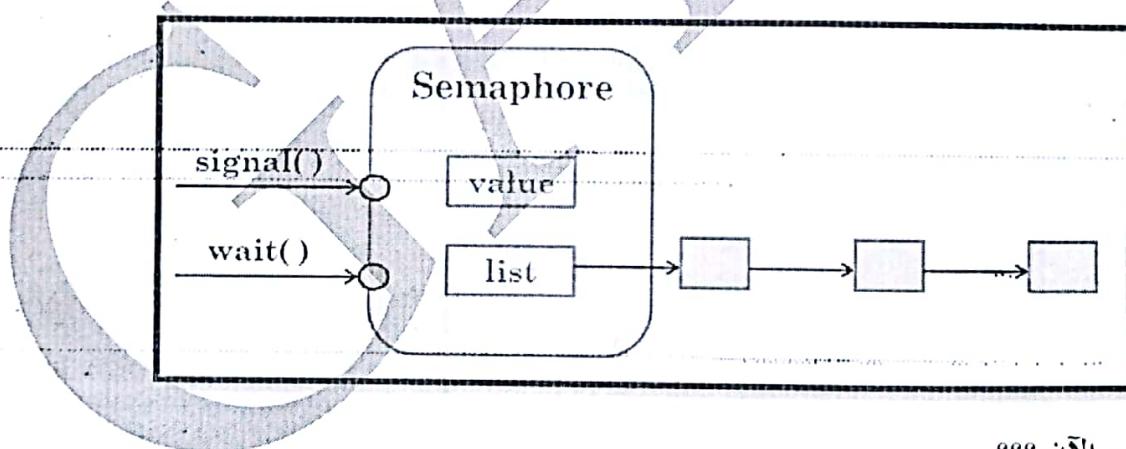
value ينقص بقدر واحد وتضاف هذه الإجرائية إلى بداية الرتل .

ب_ عندما تريد إجرائية تحرير قفل (تنفذ signal).

value تزداد بقدر واحد وتحذف الإجرائية الموجودة في نهاية الرتل ويتم تنفيذها .

بناء على ما سبق ... فإن كل سيمافور S لديه ما يلي :

- عدد صحيح value (عدد الإجرائيات المنتظرة) .
- لائحة متربطة list (الإجراءات المنتظرة بالترتيب) .
- عملية wait .
- عملية signal .



ماذا نستنتج الآن ؟؟؟

السيمافور بهذا التعريف يسمح لنا بمعالجة حالة المقطع الخرج من أجل n إجرائية ☺

لكن !!! هل تخلص من حلقة while التي تضيّع زمن المعالج (تخلص من الانتظار المشغول) ؟؟؟

هل نسيت ما
ذلك الانتظار وربما
أجاكيت ؟

من أجل التخلص من الانتظار المشغول (الحلم العربي) ...

☺ wakeup(P) و block() ← سوف نعرف العمليتين

تقوم بإيقاف الإجرائية المستدعاة لها (تصبح في waiting queue)

تقوم بإيقاظ الإجرائية P المطلوبة (تصبح في ready queue)

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

إنقاذهن القيمة بواحد

إن كانت سالبة (أكثر من إجرائية
تريد الدخول للمقطع الحرج)
سوف نضيفها للرتل ونوقفها

: signal(S) ثانياً : التابع

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

زيادة القيمة بواحد

إن كان هناك إجرائية أو أكثر ت يريد
الدخول للمقطع الحرج قم بإخراج
آخر إجرائية في الرتل ونفذها

ملاحظة : سنمرر wait و signal مؤشر على سجل semaphore * S وبالتالي للوصول إلى حقوق

هذا السجل ، نستخدم العملية > لا خراق العنوان والوصول إلى السجل الفعلي .

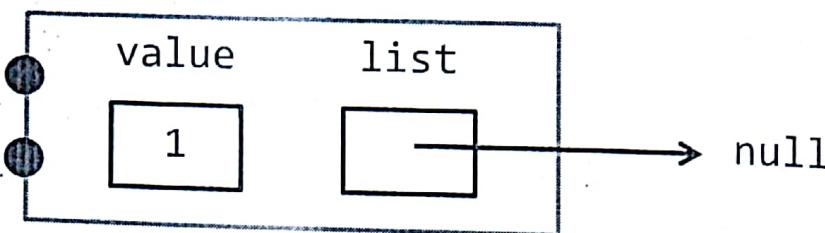
نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)



آلية العمل :

في البداية

لا يوجد أي إجرائية



أنت الآن الإجرائية P1 ونفذت $wait(S)$ وبالتالي ستنقص قيمة value وتصبح 0

ستختبر الشرط $value < 0$ وسيكون غير متحقق لأن $0 > 0$ تكافئ false وبالتالي لن تدخل في جسم الحلقة

إذ وسوف يتنهي تنفيذ $wait(S)$ وتنفذ مقطعها الخرج مباشرةً .



wait

في هذه الأثناء (تنفيذ P1 لمقطعها الخرج وقبل تنفيذها $signal(S)$) لنفرض أن الإجرائية P2 قد أنت

بالتالي ستنفذ P2 أيضاً $wait(S)$ وبالتالي ستنقص قيمة value وتصبح -1

ستختبر الشرط $value < 0$ وسيكون متحقق لأن $0 < -1$ تكافئ true وبالتالي ستدخل في جسم

block . وستضاف إلى الرتل وسيتم إيقافها باستخدام العملية

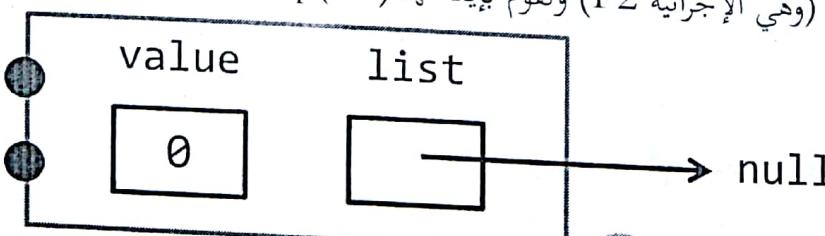


wait

في هذه اللحظات الإجرائية P1 تنفذ ... والإجرائية P2 في حالة

الآن أكملت الإجرائية P1 تنفيذ المقطع الخرج وبالتالي سوف تنفذ $signal(S)$ ل تقوم بزيادة value بواحد و

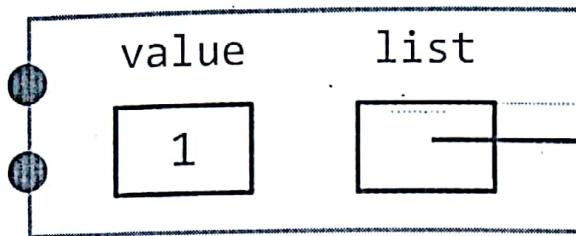
. $wakep(P2)$ وتقوم بإيقاظها



signal

الآن الإجرائية P2 تنفذ مقطعها الحرج ...

بعد أن تنهي التنفيذ سوف تنفذ signal(S) لتزيد قيمة value بواحد ، حيث أنه لا يوجد أي إجرائية في الرتل وبالتالي لن تقوم بإيقاظ أحد .



بالتالي عاد السيمافور S إلى الحالة الابتدائية التي بدأنا منها .

استخدامات السيمافورات :

أولاً : حل مشكلة المقطع الحرج من أجل n إجرائية وبدون الانتظار المشغول :

في هذه الحالة فإننا نسمي السيمافور بالاسم mutex ، بحيث يبدأ بالقيمة الابتدائية 1 .

```
semaphore mutex = 1;
while (true) {
    wait(mutex);
    /* critical section */
    signal(mutex);
    /* remainder section */
}
```

أثناء استدعاء الإجرائية التابع (wait(mutex))

- إذا كان $1 = \text{value} \Leftarrow$ تابع تفريذها ($\text{value} = 0$ الآن أصبحت 0) .

- إذا كان $0 \leq \text{value} \Leftarrow$ يتم إيقافها وتضاف إلى رتل الانتظار ($\text{value} < 0$ الآن أصبحت 0) .

أثناء استدعاء الإجرائية التابع (signal(mutex))

- إذا كان $0 = \text{value} \Leftarrow$ تابع تفريذها لوحدها ($\text{value} = 1$ الآن أصبحت 1) .

- إذا كان $0 < \text{value} \Leftarrow$ يتم تحرير إجرائية من رتل الانتظار وتتابع تفريذها معها حيث أن الإجرائية

المستدعاة للتابع signal المقطع الباقى RS بينما الإجرائية الحرة تنفذ المقطع الحرج CS .

نتائج هامة :


1_ القيمة الابتدائية للسيمافور ...

← تحدد عدد الإجرائيات التي يسمح لها بتنفيذ المقطع الحرج في نفس اللحظة .

2_ إن كانت قيمة السيمافور سالبة ...

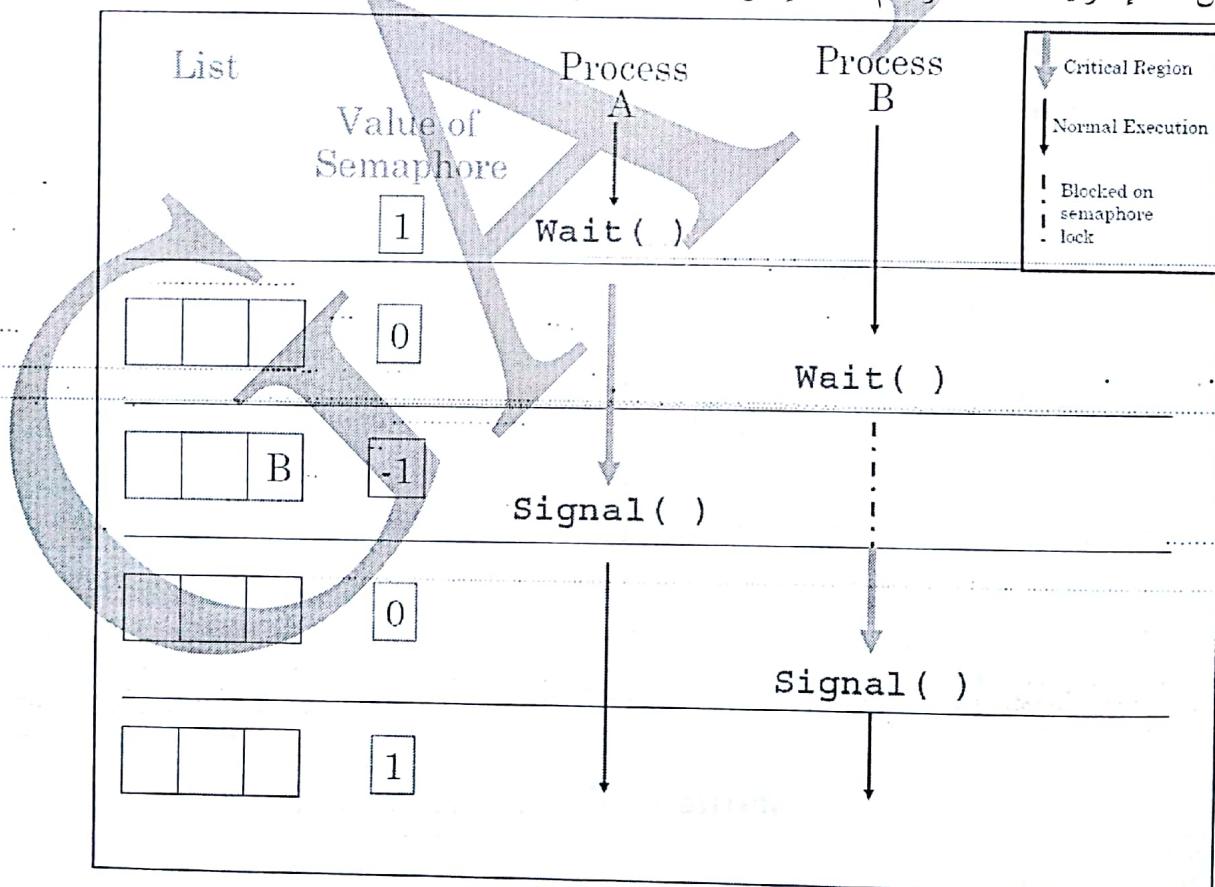
← فإن القيمة المطلقة تعبر عن عدد الإجرائيات المنتظرة .

3_ قد يتم إيقاف الإجرائية أثناء استدعائها ل ... wait

. لكن !!! لن يتم إيقافها أثناء استدعائها ل signal

مثال هام لتوضيح عمل السيمافور في حالة 2 إجرائية :

لنفرض أن الإجرائية A أتت أولاً ثم أتت الإجرائية B بعدها وكانت قيمة 1 و value=1



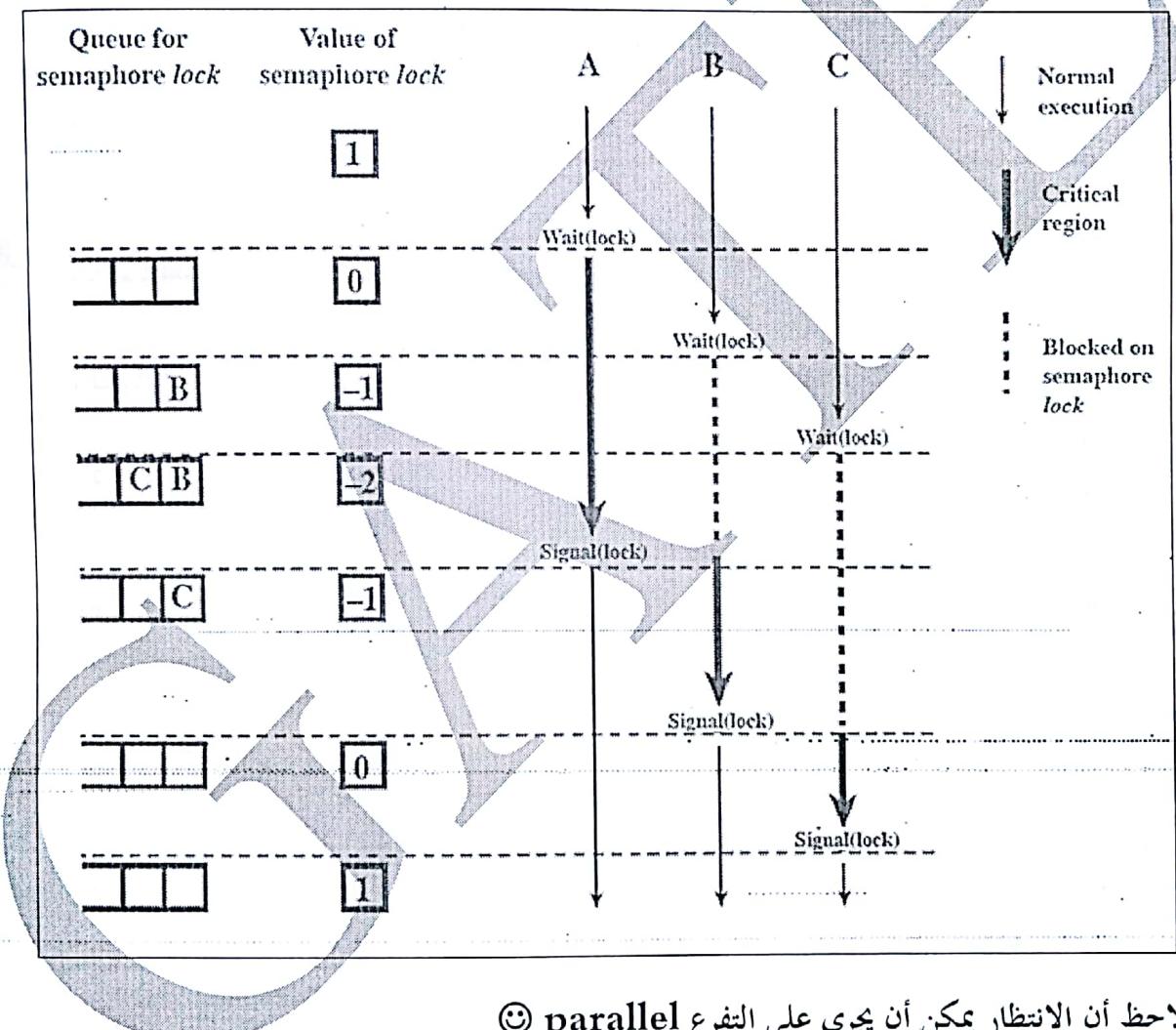
مثال لتوضيح عمل السيمافور في حالة 3 إجرائية :

اسم السيمافور هو lock
((عاشت الأسامي))

الخط العادي \leftarrow تنفيذ طبيعي (خارج المقطع الحرج) .

الخط العريض \leftarrow تنفيذ المقطع الحرج . CS

الخط المتقطّع \leftarrow انتظار (الإجرائية متجمدة) .



لاحظ أن الانتظار يمكن أن يجري على التفرع ☺ parallel

لكن !!! تنفيذ المقطع الحرج هو تسلسلي ... sequential

((أي إجرائية واحدة فقط يمكنها أن تفند في لحظة معينة))

الاستفادة من السيمافور **mutex** بشكل جزئي لحل مشكلة المنتج - المستهلك :

انتبه !!!
حل جزئي للمشكلة

كما نعلم ، بأن المقطع الحرج في مسألة المنتج - المستهلك هو :

اجزائية المنتج counter++ <= producer

اجزائية المستهلك counter-- <= consumer

لدينا المتحول المشترك counter يبدأ بالقيمة 0 (عدد المخازن الممتدة) .

سوف نكتب الكود لكل منهما كما يلي :

semaphore mutex = 1;

int counter = 0;

```
while (1) {
    wait(mutex);
    counter++;
    signal(mutex);
    /* RS */
}
```

Producer

```
while (1) {
    wait(mutex);
    counter--;
    signal(mutex);
    /* RS */
}
```

Consumer

نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)



ثانياً : عدّاد تنازلي count-down

يمكننا استخدام السيمافورات كعداد تنازلي وذلك بأن نبدأ قيمة السيمافور S بالقيمة n حيث $n \geq 1$,

العدد n يعبر عن عدد الإجرائيات المسموحة لها التواجد في مقطعها المخرج في نفس اللحظة ...

و عند وجود n إجرائية (العدد الأعظمي) تتفّق المقطع المخرج الخاص بها في نفس اللحظة ... عندها أي إجرائية جديدة ستأتي \Leftarrow سوف تنتظر إلى حين انتهاء إجرائية واحدة على الأقل .

مثال عن سيمافور كعداد تنازلي لثلاث إجرائيات :

semaphore $S = 3$;

انتبه !!!

while (1) {

wait(mutex);

while (1) {

wait(mutex);

while (1) {

wait(mutex);

ثلاث إجرائيات على الأكثر مسموحة تواجدها هنا في نفس اللحظة !!!

signal(mutex);

/* RS ... */

}

P1

signal(mutex);

/* RS ... */

}

P2

signal(mutex);

/* RS ... */

}

P3

الآن إن أتت إجرائية P4 مثلاً

وكانت P1 و P2 و P3 تتقاذن CS في نفس الوقت ...

بالتالي سوف تنتظر P4 حتى تنتهي إحداهنّ لكي تتبع معهم الحفلة 😊



ثالثاً : إبلاغ notification

يمكّنا استخدام السيمافورات لاستخدامها من قبل الإجرائيات كعلام أو إشارة من الإجرائية إلى الأخرى بحدوث شيء ما (تم شرح هذه الفكرة سابقاً) ...

مثال 1 : لدينا برنامج لحساب المتوسط الحسابي لعددين x, y .

فينا بتقسيم العمل على إجرائيتين $P1, P2$ بحيث أن :

- الإجرائية $P1$ تقوم بحساب المجموع أي $x+y$

- الإجرائية $P2$ تقوم بقسمة المجموع (النتائج) على 2 أي $\frac{x+y}{2}$

ما المشكلة يا عزيزي ؟؟؟ إنني لا أرى أي مشكلة !!!

المشكلة هي لو أن الإجرائية $P2$ نفذت أولاً قبل أن تكون الإجرائية $P1$ قد حسبت المجموع وبالتالي سوف

يكون ناتج المجموع صفرًا وبالتالي ستحسب الإجرائية $P2$ ما يلي $\frac{0}{2}$ ويساوي 0 ولكن هذا خاطئ !!!

لكن لو نفذت الإجرائية $P1$. ثم $P2$ بهذا التناوب ستكون الأمور بخير ☺

بالتالي يجب أن تعطي الإجرائية $P1$ إشارة (غمزة wink) إلى الإجرائية $P2$ بأن تكمل العمل ...

((((notification)) (هذه الإشارة نسميها إبلاغ



نظرة بالإشارة ... تعرف نبض الصرارة ...

synch = synchronization

لاحظ أن الإجرائية P2 تنتظر إشارة من P1

بينما الإجرائية P1 لا تنتظر شيء من P2 !!!

ماذا نستنتج ؟؟؟ نستنتج أنه تحتاج سيمافور واحد فقط وليكن اسمه **synch** وسيبدأ بالقيمة 0 (لماذا؟)

نفس المثال
ص 12

وسيكون لدينا أيضاً المتحول المشترك **sum** بحيث سيبدأ بالقيمة 0 (حيادي الجمع).

وسيكون الكود كالتالي :

semaphore **synch** = 0;

int **sum** = 0;

while (1) {

sum = **x1+x2**;

signal(synch);

}

while (1) {

wait(synch);

avg = sum/2;

}

P2

لاحظ أن الإجرائية P2 ستبقى متنبطة حتى تنهي الإجرائية P1 عملها وتعطي إبلاغ

عن طريق تنفيذ **signal** على السيمافور **synch** الذي تنتظره الإجرائية P2.

(تساؤل خارج عن المحاضرة) لاحظ أنه قد يحدث شيء لم نحسبه ما هو ؟؟؟

قد تنفذ P1 أكثر من مرة وبالتالي P2 قد لا تحسب كل القيم (P1 قد تنفذ أكثر من 2)

نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)

مثال 2 : لدينا برنامج يطبع 1 بشكل مستمر ...

قمنا بتقسيم العمل على إجرائيتين $P1, P2$ ، بحيث أن :

- الإجرائية $P1$ تقوم بطباعة العدد 1 أي ;
 $cout << 1$

- الإجرائية $P2$ تقوم بطباعة العدد 2 أي ;
 $cout << 2$

المشكلة هنا أعقد من السابقة ، وهو أنه لا يمكن طباعة العدد 2 إلا في حال طباعة 1 قبله

أضف إلى ذلك أنه لا يمكن طباعة العدد 1 إلا في حال طباعة 2 قبله ☺ (ما عدا أول مرة)

بالتالي أصبح لدينا إشارة من $P1$ إلى $P2$ لتخبرها بأنها قد أكملت عملها (طبعت 1)

بالإنجليزية

ولدينا أيضاً إشارة من $P2$ إلى $P1$ لتخبرها بأنها قد أكملت عملها (طبعت 2)

بالتالي نحتاج إلى 2 سيمافور أي ... $S1, S2$

حيث أنه لفرض تسلسل معين نحتاج إلى تعيينة هذه السيمافورات بالقيم الابتدائية المناسبة .

لكي نجعل $P1$ تبدأ أولاً سنجعل السيمافور الذي تنتظره يبدأ بالقيمة 1 والعكس من أجل $P2$.

إن $P1$ تنتظر إشارة $P2$. من خلال السيمافور $S1$

إن $P2$ تنتظر إشارة $P1$ من خلال السيمافور $S2$

بالتالي كي نجعل $P1$ تبدأ بالطباعة (طبع أول 1 على الشاشة)

سوف نجعل $S1=1$ و $S2=0$

وسوف نقوم بتنفيذ تصالب بين الإجرائيتين كما يلي ...

نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)



```
semaphore S1 = 1;
```

```
semaphore S2 = 0;
```

```
while (1) {  
    //do something  
    wait(S1);  
    cout<<"1";  
    signal(S2);  
    //do something  
}
```

```
while (1) {  
    //do something  
    wait(S2);  
    cout<<"2";  
    signal(S1);  
    //do something  
}
```

P1

P2

سيكون الخرج كما يلي :

1 2 1 2 1 2 1 2 1 2

هذا جيد إلى الآن ...

سؤال : كيف سنجعل الخرج يكون بالعكس أي إذا أردنا طباعة :

2 1 2 1 2 1 2 1 2 1

ماذا يجب أن نفعل ؟؟؟

إن لم تعرف الإجابة اتصلك
بالذهاب في رحلة خارج البلد

حلول مقترنة لمشاكل التزامن الشهيرة :

مشكلة القارئان والكتابات
لم تعطى السنة!

مسألة المنتج المستهلك : bounded buffer

إن مسألة المنتج - المستهلك تحيي ثلاثة مشاكل :

1_ المقطع الخرج counter++ و counter--

2_ الانتظار في حال كان المخزن فارغ buffer is empty

3_ الانتظار في حال كان المخزن ممتلئ buffer is full

إن المشكلة الأولى (المقطع الخرج) تم حلها سابقاً في الصفحة 20 (حل جزئي للمشكلة).

بالعودة إلى إجرائية المنتج والممستهلك (من المحاضرة السابقة) كان لدينا ما يلي :

```
do {
    while (counter==n) ;
    buffer[in]=nextProduced;
    in = (in+1) % n;
    counter++;
} while (true);
```

Producer

```
do {
    while (counter==0) ;
    nextConsumed=buffer[out];
    out = (out+1) % n;
    counter--;
} while (true);
```

Consumer

إن الحلقة while(counter==0) هي مشكلة كبيرة !!!

حيث هنا تفقد الإجرائية التعليمية الخالية ; (الانتظار المشغول) وهذا سوف نتخلص منه الآن ...

نظم تشغيل المحاضرة التاسعة (ترامن الإجراءات 2)

إن العملية التي سنقوم بها هي عملية عكسية للانتظار :

إجرائية المنتج Producer . تنتظر طلباً أن المخزن متليء ← سنجعلها تتوقف عندما يصبح المخزن متليء :

● سعر فسيمافور اسمه full يعبر عن عدد المخازن الممتلأة يبدأ بالقيمة 0.

إجراءات المستهلك Consumer تنتظر طلما أن المخزن فارغ ← ستتوقف عندما يصبح المخزن فارغ .

رَكَزُ الآنِ جيداً (هنا أصبح العكس فعلياً) :

أراد المنتج Producer إنتاج عنصر جديد ... فهو يتمنى وجود مخازن فارغة لكي يملؤها

بالناتي ينتظر السيمافور empty (إن كان 0) يعني أنه لا يوجد مخازن فارغة (ممتلئ) ← توقف).

ان أراد المستهلك Consumer استهلاك عنصر... فهو يتمنى وجود مخازن ممتلئة لكي يستهلكها

بالناتج، يتطلب السيمافور full (إن كان 0 فهذا يعني أنه لا يوجد مخازن ممتلئة (فاغ) ← توقف).

انت الان تفهم كل شيء

أضاف إلى ذلك أنه لم تعد بحاجة للمتحول **counter** أصلًا !!!

حيث أن القيمة الموجودة في السيمافور full تكافئ قيمة المتحول counter (عدد المخازن المختلفة).

لكن هنا تخلصنا من المشكلة الأولى وهي `counter++` و `counter--`

المواءب هو لا ... حيث أن المقطوع المخرج الآن أصبح عملية الإنتاج والاستهلاك نفسها ! لأن المخزن

buffer (المصفوفة) على اعتبار أنه متاحول مشترك لا يجب أن يتم تنفيذ عمليتين عليه في نفس الوقت

(انتاج واستهلاك) لذلك سوف نعرف سيمافور ثالث اسمه mutex لحماية ال buffer .

نظم تشغيل المحاضرة التاسعة (ترامن الإجرائيات 2)

ملاحظة : دوماً يجب أن يكون لدينا $\text{full} + \text{empty} = \text{BUFFER_SIZE}$

ملاحظة لتشييت المعلومات :

. notification mutex \leftarrow يستخدم في الاستبعاد المتبادل والإبلاغ السيمافور الثنائي

. count-down \leftarrow يستخدم كعداد تنازلي السيمافور العددي

. counting_semaphore \leftarrow يوجد نمط اسمه binary_semaphore ونقط آخر اسمه counting_semaphore لنفرض أنه يوجد نمط اسمه binary_semaphore

الحل المنشود لمشكلة المنتج - المستهلك باستخدام السيمافورات :

```
binary_semaphore mutex = 1;
```

```
counting_semaphore full = 0;  $\leftarrow$  عدد المخازن الممتلئة
```

```
counting_semaphore empty = n;  $\leftarrow$  عدد المخازن الفارغة
```

```
do {
    wait(empty);
    wait(mutex);
    buffer[in] = nextProduced;
    in = (in+1) % n;
    signal(mutex);
    signal(full);
} while (true);
```

```
do {
    wait(full);
    wait(mutex);
    nextConsumed = buffer[out];
    out = (out+1) % n;
    signal(mutex);
    signal(empty);
} while (true);
```

Producer

Consumer

مثال تجربى : لنفرض أن $n=4$ (المخزن يتسع ل 4 عناصر فقط) ، ولتبدا إجرائية المنتج بانتاج عنصر

واحد ثم تأتي إجرائية المستهلك وتستهلك هذا العنصر وتخلصنا بها !!!

ستكون القيم الابتدائية للسيمافورات كما يلى :

semaphore full 0
semaphore empty 4

BUFFER



لتبدأ إجرائية المنتج Producer بالتنفيذ :

- 1_ decrease empty buffers \rightarrow wait(empty).
- 2_ place the new item in the buffer.
- 3_ increase full buffers \rightarrow signal(full).

semaphore full 1
semaphore empty 3

BUFFER

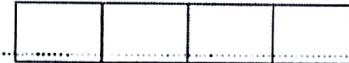


لتأتي بعدها إجرائية المستهلك Consumer وتنفذ :

- 1_ decrease full buffers \rightarrow wait(full).
- 2_ take an item from the buffer.
- 3_ increase empty buffers \rightarrow signal(empty).

semaphore full 0
semaphore empty 4

BUFFER



Special thanks to Ahmad Mokayes

انتهت المعاشرة

☺ بال توفيق ☺



Mohammed Moulla