OPERATING SYSTEM

Lecture Notes

Dr. Professor, J.M. Khalifeh

قسم المعلوماتية

الوحدة الخامسة

النسخة العربية

Unit-5

Process Synchronization

ملخص

تزامن العمليات هو تنسيق تنفيذ عمليات متعددة في نظام متعدد العمليات لضمان وصول هذه العمليات إلى الموارد المشتركة بطريقة خاضعة للرقابة ويمكن التنبؤ بها. تهدف المزامنة إلى حل المشكلة التي تنشأ عن ظروف التسابق وقضايا المزامنة الأخرى في نظام متزامن.

في نظام متعدد العمليات، تكون المزامنة ضرورية لضمان اتساق البيانات وسلامتها، ولتجنب مخاطر حالات الجمود ومشاكل المزامنة الأخرى. تعد مزامنة العمليات جانبًا مهمًا من أنظمة التشغيل الحديثة، وتلعب دورًا حاسمًا في ضمان الأداء الصحيح والفعال للأنظمة متعددة العمليات.

على أساس المزامنة، يتم تصنيف العمليات على أنها واحدة من النوعين التاليين:

- العملية المستقلة Independent Process: لا يؤثر تنفيذ عملية واحدة على تنفيذ العمليات الأخرى.
- العملية المتعاونة: Cooperative Process: عملية يمكن أن تؤثر أو تتأثر بالعمليات الأخرى التي يتم تنفيذها في النظام.

تنشأ مشكلة مزامنة العملية في حالة العملية المتعاونة أيضًا بسبب مشاركة الموارد في العمليات المتعاونة.

أهداف الوحدة

- وصف مشكلة القسم الحرج وشرح حالة التسابق.
- توضيح حلول الأجهزة لمشكلة القسم الحرج باستخدام حواجز الذاكرة وعمليات المقارنة والمبادلة والمتغيرات الذرية.
- توضيح كيف يمكن استخدام أقفال المزامنة والسيمافورات والمونيتورات ومتغيرات الحالة لحل مشكلة القسم الحرج.
- تقييم الأدوات التي تحل مشكلة القسم الحرج في سيناريوهات مختلفة

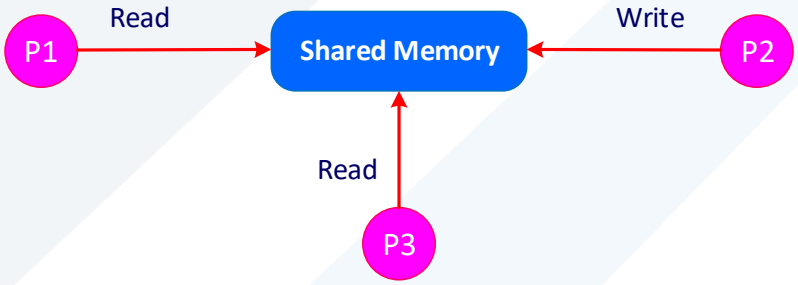
What is process synchronization in OS? ما هي مزامنة العمليات في نظام التشغيل

نظام التشغيل هو برنامج يدير جميع التطبيقات على الجهاز ويساعد بشكل أساسي في الأداء السلس لجهاز الكمبيوتر الخاص بنا. لهذا السبب، يجب أن يقوم نظام التشغيل بأداء العديد من المهام، وأحيانًا في وقت واحد. هذه ليست مشكلة عادة إلا إذا كانت هذه العمليات التي تحدث في وقت واحد تستخدم موردًا مشتركًا. ولتوضيح الأمر يمكننا مقارنته بما قد يحدث في المثال التالي:

لنأخذ في الاعتبار بنكًا يخزن رصيد حساب كل عميل في نفس قاعدة البيانات. افترض الآن أن لديك في البداية x ليرة في حسابك. الآن، تقوم بسحب مبلغ من المال من حسابك المصرفي، وفي نفس الوقت، يحاول شخص ما النظر في مقدار الأموال المخزنة في حسابك. نظرًا لأنك تقوم بسحب بعض الأموال من حسابك، بعد المعاملة، سيكون إجمالي الرصيد المتبقي أقل من x. لكن المعاملة تستغرق وقتًا، وخلال هذا الوقت يقرأ الشخص x كرصيد في حسابك مما يؤدي إلى بيانات غير متناسقة. إذا تمكنا بطريقة ما من التأكد من حدوث عملية واحدة فقط في كل مرة، فيمكننا ضمان اتساق هذه البيانات.

بالعودة إلى نظام التشغيل فإنه وبنفس الطريقة يمكن أن يحدث تضارب في البيانات عندما تشترك عمليات مختلفة في مورد مشترك في النظام وهذا هو سبب الحاجة إلى مزامنة العملية في نظام التشغيل.

دعونا نلقي نظرة على سبب حاجتنا بالضبط إلى مزامنة العملية. على سبيل المثال، إذا كانت عملية 1 في الشكل 1 تحاول قراءة البيانات الموجودة في موقع ذاكرة بينما تحاول عملية أخرى 2 تغيير البيانات الموجودة في نفس الموقع، فهناك احتمال كبير أن البيانات التي تقرأها العملية 1 ستكون غير صحيحة.



الشكل 1: مشاركة ثلاث عمليات بمورد في الذاكرة

يؤدي غياب التزامن في بيئة التواصل بين العمليات إلى المشاكل التالية:

- **عدم الاتساق Inconsistency:** عندما تصل عمليتان أو أكثر إلى بيانات مشتركة في الوقت نفسه دون تزامن سليم. قد يؤدي هذا إلى تغييرات متضاربة، حيث تُستبدل تحديثات إحدى العمليات بأخرى، مما يجعل البيانات غير موثوقة وغير صحيحة.
- **فقدان البيانات Loss of Data:** يحدث فقدان البيانات عندما تحاول عمليات متعددة كتابة أو تعديل نفس المورد المشترك دون تنسيق. إذا قامت إحدى العمليات باستبدال البيانات قبل انتهاء عملية أخرى، فقد تُفقد معلومات مهمة، مما يؤدي إلى بيانات غير كاملة أو تالفة.
- **الجمود Deadlock:** يؤدي نقص التزامن إلى الجمود، مما يعني توقف عمليتين أو أكثر عن العمل، حيث تنتظر كل منهما الأخرى لتحرير مورد. ونظرًا لعدم قدرة أي من العمليات على الاستمرار، يصبح النظام غير مستجيب ولا تتمكن أي منها من إكمال مهامها.

دعونا نلقي نظرة على الأقسام المختلفة لبرنامج العملية:

- **قسم الدخول Entry Section:** يقرر قسم الدخول دخول العملية.
- **القسم الحرج Critical Section:** يسمح القسم الحرج ويتأكد من أن عملية واحدة فقط تعدل البيانات المشتركة الشكل 2.

- **قسم الخروج Exit Section:** يتم التعامل مع إدخال العمليات الأخرى في البيانات المشتركة بعد تنفيذ عملية واحدة بواسطة قسم الخروج.
- **القسم المتبقي Remainder Section:** الجزء المتبقي من الكود الذي لم يتم تصنيفه على النحو الوارد أعلاه موجود في القسم المتبقي.

```
do
    {
  acquire lock

  critical section

  release lock

  remainder section

  } while (TRUE):
```

الشكل 2: البنية العامة لتنفيذ العملية

حالة التسابق Race Condition

عندما تقوم أكثر من عملية بتشغيل باستخدام نفس الرمز أو تعديل نفس الذاكرة أو التعامل مع أي بيانات مشتركة، فهناك خطر يتمثل في أن نتيجة أو قيمة البيانات المشتركة قد تكون غير صحيحة لأن جميع العمليات قد تحاول الوصول إلى هذا المورد المشترك وتعديله. وبالتالي، تتسابق جميع العمليات لتقول إن نتيجتي صحيحة. هذه الحالة تسمى حالة التسابق. نظرًا لأن العديد من العمليات تستخدم نفس البيانات، فقد تعتمد نتائج العمليات على ترتيب تنفيذها.

هذا هو في الغالب موقف يمكن أن ينشأ داخل القسم الحرج. في القسم الحرج، تحدث حالة التسابق عندما تختلف النتيجة النهائية لعمليات تنفيذ المسالك المتعددة اعتمادًا على التسلسل الذي يتم تنفيذ المسالك فيه.

مثال 1:

كمثال على تأثير حالة التسابق نورد هنا مشكلة المُنتِج–المستهلك، وهي نموذج تمثيلي للعديد من وظائف نظام التشغيل. حيث يوضح المثال كيفية استخدام مخزن مؤقت محدود Bounded Buffer لتمكين العمليات من مشاركة الذاكرة.

أحد الاحتمالات هو إضافة متغير عدد صحيح، count، بتهيئته إلى 0. يتزايد count في كل مرة نضيف فيها عنصرًا جديدًا إلى المخزن المؤقت، وينخفض في كل مرة نزيل فيها عنصرًا واحدًا. يمكن تعديل شيفرة عملية المُنتِج كما يلي:

- متغير العداد = 0
- يتم زيادة العداد في كل مرة نضيف فيها عنصرًا إلى عداد المخزن المؤقت ++

- يتناقص العداد في كل مرة ننقل فيها عنصرًا من عداد المخزن المؤقت –
- افترض أن قيمة العداد حاليًا 5
- تنفذ عمليات المنتج والمستهلك العبارتين "counter ++" و "counter –" في نفس الوقت.
- بعد تنفيذ هاتين العبارتين، قد تكون قيمة عداد المتغير 4 أو 5 أو 6!

ومع ذلك، فإن النتيجة الصحيحة الوحيدة هي العداد 5، والذي يتم إنشاؤه بشكل صحيح إذا نفذ المنتج والمستهلك بشكل منفصل.

يمكن تنفيذ "counter++" بلغة الآلة (على جهاز نموذجي) على النحو التالي:

$register_1$=counter
$register_1$ = $register_1$+1
counter =$register_1$

كما يمكن تنفيذ "counter––" بلغة الآلة (على جهاز نموذجي) على النحو التالي:

$register_2$=counter
$register_2$ = $register_2$-1
counter =$register_2$

T_0:	**producer**	**execute**	**$register_1$ = counter**	**{$register_1$ = 5}**
T_1:	producer	execute	$register_1$ = $register_1$+1	{$register_1$ = 6}
T_2:	consumer	execute	$register_2$ = counter	{$register_2$ = 5}
T_3:	consumer	execute	$Register_2$ = $register_2$-1	{$Register_2$ = 4}
T_4:	producer	execute	counter= $register_1$	{counter= 6}
T_5:	consumer	execute	counter= $register_2$	{counter= 4}

سنصل إلى هذه الحالة غير الصحيحة لأننا سمحنا لكلا العمليتين بمعالجة متغير العداد بشكل متزامن.

يُطلق على موقف مثل هذا، حيث تقوم العديد من العمليات بالوصول إلى نفس البيانات ومعالجتها بشكل متزامن وتعتمد نتيجة التنفيذ على الترتيب المعين الذي يحدث فيه الوصول، حالة التسابق.

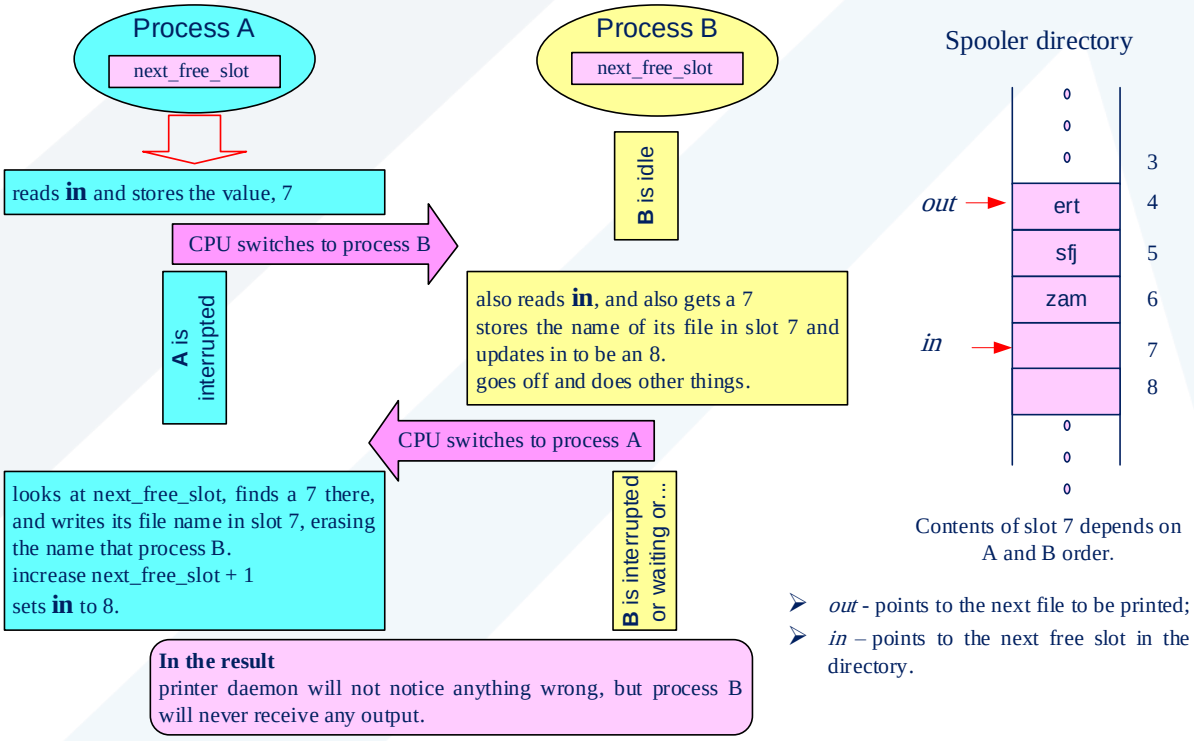
مثال 2:

عندما تريد عملية A طباعة ملف، تُدخل اسم الملف في دليل مُخزّن spooler طباعة خاص. وهناك عملية أخرى B، وهي برنامج الطابعة، تتحقق دوريًا من وجود أي ملفات للطباعة، وفي حال وجودها، تطبعها وتزيل أسمائها من الدليل كما في الشكل 3. تخيل أن دليل المُخزّن لدينا يحتوي على عدد كبير من الأماكن الفارغة، مرقمة من 0، 1، 2، ...، كل منها قادر على الاحتفاظ باسم ملف. تخيل أيضًا أن هناك متغيرين مشتركين، out الذي يُشير إلى الملف التالي المراد طباعته، inالذي يُشير إلى المكان الخالي التالي حيث يمكن الكتابة في الدليل. في لحظة معينة، تكون الفتحات من 0 إلى 3 فارغة (لأن الملفات قد طُبعت بالفعل) والفتحات من 4 إلى 6 ممتلئة (بأسماء الملفات المراد طباعتها). في وقت واحد تقريبًا، تقرر العمليتان A و B وضع ملف في قائمة انتظار للطباعة كما هو موضح في الشكل. تقرأ العملية أ القيمة 7 وتخزنها في متغير محلي يُسمى next_free_slot. بعد ذلك، تحدث مقاطعة مؤقتة، ويقرر المعالج أن العملية A قد

عملت لفترة كافية، فينتقل إلى العملية B. تقرأ العملية B أيضًا القيمة 7، وتحصل عليها أيضًا، فتخزن اسم ملفها في الفتحة 7 وتُحدّثها إلى 8. ثم تتوقف عن العمل وتقوم بأشياء أخرى. في النهاية، تعمل العملية A مرة أخرى، بدءًا من المكان الذي توقفت فيه في المرة السابقة. تبحث في next_free_slot، وتجد القيمة 7 هناك، وتكتب اسم ملفها في الفتحة 7، وتمحو الاسم الذي وضعته العملية B هناك. ثم تحسب next_free_slot + 1، وهو 8، وتضبطه إلى 8. أصبح دليل التخزين المؤقت الآن متسقًا داخليًا، لذا لن يلاحظ برنامج الطابعة أي خطأ، ولكن لن تتلقى العملية B أي مخرجات أبدًا.

لكن كيف نتجنب حالة التسابق هذه؟

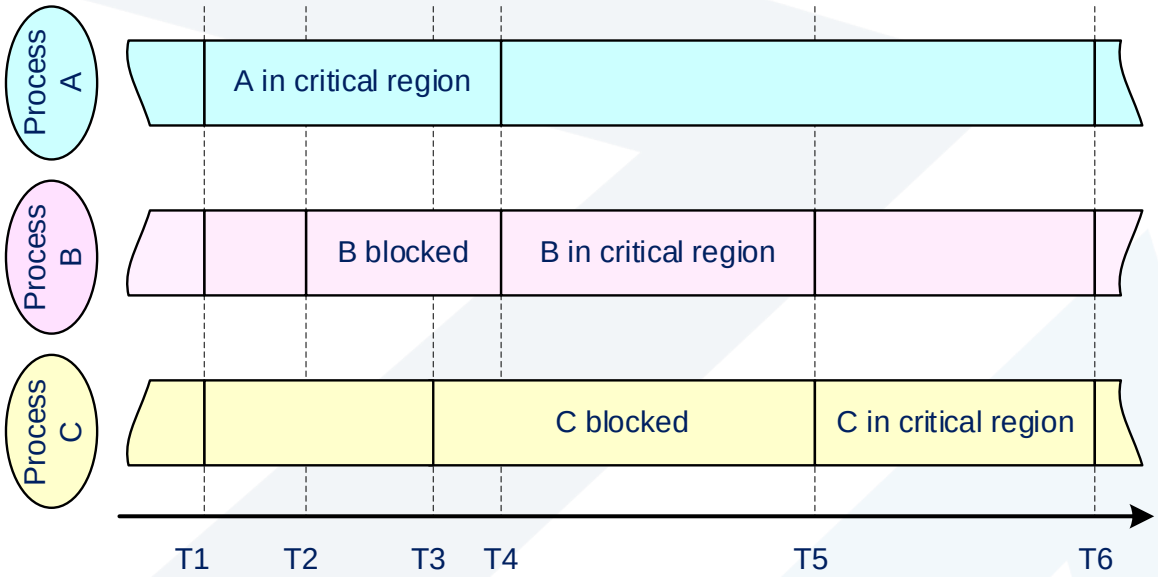
يكون تجنب ذلك من خلال التعامل مع القسم الحرج كقسم لا يمكن الوصول إليه إلا بعملية واحدة في كل مرة. يسمى هذا النوع من الأقسام القسم الذري. أي غير القابل للتجزئة.



الشكل 3: مثال آخر عن حالة التسابق بين عمليتي قراءة وكتابة على الطابعة

What is the critical section problem ما هي مشكلة القسم الحرج؟

لماذا نحتاج إلى قسم حرج؟ ما هي المشاكل التي تحدث إذا قمنا بإزالته؟



الشكل 4: التنفيذ المشترك للعمليات مع وجود القسم الحرج

يُعرف الجزء من الكود الذي لا يمكن الوصول إليه إلا من خلال عملية واحدة في أي لحظة باسم القسم الحرج. هذا يعني أنه عندما ترغب الكثير من البرامج في الوصول إلى بيانات مشتركة واحدة وتغييرها، فلن يُسمح إلا لعملية واحدة فقط بالتغيير في أي لحظة. ويبين الشكل 4 كيفية الدخول إلى القسم الحرج من قبل ثلاث عمليات بحيث لا يسمح لأي منها بالدخول إلى القسم الحرج طالما أن هناك عملية قد سبقتها إلى ذلك ولا بد من الانتظار أو الانشغال بتنفيذ مسارات أخرى طالما أن العملية الأخرى مازالت هناك.

- T1: تدخل A المنطقة الحرجة (القسم)
- T2: تحاول B دخول المنطقة الحرجة، فتمنع
- T3: تحاول C دخول المنطقة الحرجة، فتمنع
- T4: تغادر A المنطقة الحرجة، وتدخل B إليها، ولا تزال C في حالة مُنع
- T5: تغادر C المنطقة الحرجة.

يجب أن تنتظر العمليات الأخرى حتى تكون البيانات متاحة للاستخدام وذلك باستخدام تعليمات مثل wait( ) و signal( ) على سبيل المثال.

في هذه الحالة تتعامل wait( ) بشكل أساسي مع الإدخال إلى القسم الحرج، بينما تتولى signal( ) الخروج من القسم الحرج. إذا أزلنا القسم الحرج، فلا يمكننا ضمان اتساق النتيجة النهائية بعد انتهاء جميع العمليات في وقت واحد.

سننظر في بعض الحلول لمشكلة القسم الحرج ولكن قبل أن ننتقل إلى ذلك، دعونا نلقي نظرة على الشروط اللازمة لحل مشكلة القسم الحرج.

مزامنة العمليات Process Synchronization

مزامنة العمليات هي تنسيق تنفيذ عمليات متعددة في نظام متعدد العمليات لضمان وصولها إلى الموارد المشتركة بطريقة مُتحكم بها وقابلة للتنبؤ. وتهدف إلى حل مشكلة ظروف التسابق وغيرها من مشكلات المزامنة في نظام متزامن.

يجب تلبية المتطلبات الثلاثة التالية عن طريق حل مشكلة القسم الحرج:

- **الاستبعاد المتبادل Mutual exclusion:** إذا كانت هناك عملية قيد التشغيل في القسم الحرج، فلا ينبغي السماح بإجراء أي عملية أخرى في هذا القسم في ذلك الوقت.
- **التقدم Progress:** إذا لم تكن هناك عملية في القسم الحرج وكانت هناك عمليات أخرى تنتظر خارج القسم الحرج للتنفيذ، فيجب السماح لأي من مسالك هذه العملية بدخول القسم الحرج. وسيتم اتخاذ قرار العملية التي ستدخل القسم الحرج من خلال تلك العمليات التي لم تنتقل بعد إلى التنفيذ في القسم المتبقي فقط.
- **عدم المجاعة No starvation :** تعني المجاعة أن العملية تظل تنتظر إلى الأبد للوصول إلى القسم الحرج ولكن لا تحصل على فرصة أبدًا. تعني عدم المجاعة أيضًا الانتظار المحدود.
  - يجب ألا تنتظر العملية إلى الأبد للدخول داخل القسم الحرج.
  - عندما تقدم عملية ما طلبًا للوصول إلى قسمها الحرج، يجب أن يكون هناك حد أو تقييد، وهو عدد العمليات الأخرى المسموح لها بالوصول إلى القسم الحرج قبلها.
  - بعد الوصول إلى هذا الحد، يجب السماح لهذه العملية بالوصول إلى القسم الحرج.

دعونا الآن نناقش بعض الأدوات المستخدمة في حلول مشكلة القسم الحرج.

حل بيترسون

يتم استخدام حل بيترسون لمشاكل القسم الحرج على نطاق واسع في البرامج، فهو حل كلاسيكي للبرامج حين تتطلب ذلك. يعتمد الحل على فكرة أنه عندما يتم تنفيذ عملية في قسم حرج، فإن العملية الأخرى تنفذ بقية التعليمات البرمجية والعكس صحيح أيضًا، أي أن هذا الحل يتأكد من أن عملية واحدة فقط تنفذ القسم الحرج في أي وقت من الأوقات.

في حل بيترسون، لدينا متغيرين مشتركين تستخدمهما العمليات.

- boolean Flag[ ] : توضع حالته المسبقة FALSE وتصبح TRUE حين تصبح العملية بحاجة للدخول إلى القسم الحرج. أي تمثل مصفوفة Flag العمليات التي تريد الدخول في قسمها الحرج.
- int Turn: يشير هذا المتغير الذي تكون قيمته عددا صحيحا إلى العملية التي تصبح جاهزة للدخول في القسم الحرج.

Flag array

True	False	False	True		False

Process 1 2 3 4 n

```
do{
    //A process Pi wants to enter into the critical section

    //The ith index of flag is set
```

```
        Flag[i] = True;
        Turn = i;
        while(Flag[i] && Turn == i);

        { Critical Section };

        Flag[i] = False;
        // another process can go to Critical Section
        Turn = j;

            Remainder Section

    }  while ( True);
```

- هذا حل يعتمد على البرامج لحل مشكلة القسم الحرج.
- لا يعمل على البنى الحاسوبية الحديثة.
- سيتم هنا إيضاحه من أجل عمليتين تتناوبان التنفيذ بين القسم الحرج والقسم المتبقي. بفرض أن P1 هي العملية الأولى و P2 هي العملية الثانية.
- يجب أن تشارك العمليتان عنصري بيانات مع بعضهما البعض.
  - int turn
  - Boolean flag [2]
- Turn : يشير إلى العملية التي يجب أن تدخل في قسمها الحرج.
- flag: يخبرنا ما إذا كانت العملية جاهزة للدخول إلى قسمها الحرج. تشير flag[i] إلى العملية Pi، أي flag[i]=TRUE فإن Process Pi جاهزة للتنفيذ في قسمها الحرج. وتشير flag[j] إلى العملية Pj. إذا كانت العلامة flag[j]=TRUE فإن العملية Pj جاهزة للتنفيذ في قسمها الحرج.

الآن دعونا نلقي نظرة على خوارزمية بيترسون.

Peterson's Solution

```
while (true)
{
   flag[i] = TRUE;
   turn = j;
   while ( flag[j] && turn == j);
   //CRITICAL SECTION
   flag[i] = FALSE;
   //REMAINDER SECTION
}
```

Structure of process Pi	Structure of process Pj
While(true){	While(true){
flag[i] = true;	flag[j] = true;

turn = i; while(flag[i]&& turn==i); // Critical Section flag[j]= false; // Remainder Section }	**turn = j;** **while(flag[j]&& turn==j);** **// Critical Section** **flag[i]= false;** **// Remainder Section** **}**

- أولاً، تقوم Pi بتعيين flag[i]=TRUE، ثم تقوم بتحويل الدور إلى j أي turn=j لذلك إذا أرادت Pj الدخول إلى القسم الحرج، فيمكنها القيام بذلك.
- الخطوة السابقة تضمن أنه إذا حاولت كل من Pi، Pj الدخول إلى القسم الحرج في نفس الوقت، فسيتم تغيير الدور أولاً إلى i، ثم j أو العكس. لكن النقطة المهمة هي أنه يُسمح لواحدة فقط من هاتين العمليتين بالدخول إلى قسمها الحرج.

دعم الأجهزة للتزامن Hardware Support for Synchronization

الأدوات المستندة إلى البرامج ليست مضمونة للعمل على الكمبيوترات ذات البنى الحديثة. هنا، نقدم ثلاث تعليمات الأجهزة للعمليات البدائية التي توفر الدعم لحل مشكلة القسم الحرج. يمكن استخدام هذه العمليات البدائية لتشكيل أساس المزيد من آليات التزامن المجردة.

حواجز الذاكرة Memory Barriers

تُعرف الطريقة التي تحدد بها بنية الكمبيوتر ما تضمنه الذاكرة التي ستوفرها لبرنامج تطبيق باسم نموذج الذاكرة الخاص بها.

تختلف نماذج الذاكرة حسب نوع المعالج، لذلك لا يستطيع مطورو النواة وضع أي افتراضات فيما يتعلق بإمكانية رؤية التعديلات على الذاكرة المشتركة لمعالجات متعددة. لمعالجة هذه المشكلة، توفر بنى الكمبيوتر إرشادات يمكن أن تفرض نشر أي تغييرات في الذاكرة على جميع المعالجات الأخرى، وبالتالي ضمان أن تكون تعديلات الذاكرة مرئية للمسالك التي تعمل على المعالجات الأخرى. تُعرف هذه التعليمات باسم حواجز الذاكرة أو أسوار الذاكرة. عند تنفيذ تعليمات حاجز الذاكرة، يضمن النظام اكتمال جميع الأحمال والمخازن قبل تنفيذ أي عمليات تحميل أو تخزين لاحقة. لذلك، حتى إذا تم إعادة ترتيب التعليمات، يضمن حاجز الذاكرة أن عمليات المخزن قد اكتملت في الذاكرة ومرئية للمعالجات الأخرى قبل إجراء عمليات التحميل أو التخزين المستقبلية.

كمثال، ضع في اعتبارك البيانات التالية المشتركة بين مسلكين:

```
boolean flag = false;
int x = 0;
;
```

حيث ينفذ حيث Thread1 البيانات

```
while (!flag)
```

```
;
print x;
```

ويؤدي الموضوع 2

```
x = 100;
    flag = true
```

السلوك المتوقع، بالطبع، هو أن المسلك 1 ينتج القيمة 100 لـ متغير x. ومع ذلك، نظرًا لعدم وجود تبعيات للبيانات بين علامة المتغيرات و x، فمن الممكن أن يقوم المعالج بإعادة ترتيب الإرشادات الخاصة بـ Thread 2 بحيث يتم تعيين هذه العلامة صحيحة قبل تعيين x = 100. في هذه الحالة، من الممكن أن يكون Thread 1 سوف ينتج 0 للمتغير x. الأمر الأقل وضوحًا هو أن المعالج قد يعيد ترتيب البيانات الصادرة عن Thread 1 وتحميل المتغير x قبل تحميل قيمة العلم. إذا حدث هذا، فستخرج سلسلة الرسائل 1 0 للمتغير x حتى إذا لم يتم إعادة ترتيب التعليمات الصادرة عن سلسلة الرسائل 2.

إذا أضفنا عملية حاجز ذاكرة إلى المسلك 1

```
while (!flag)
memory barrier();
print x;
```

نحن نضمن تحميل قيمة العلم قبل قيمة x. وبالمثل، إذا وضعنا حاجزًا للذاكرة بين التخصيصات التي يقوم بها مسلك التنفيذ 2

```
x = 100;
memory barrier();
flag = true;
```

نتأكد من أن الإسناد إلى x يحدث قبل تعيين العلامة. فيما يتعلق بحل بيترسون، يمكننا وضع حاجز ذاكرة بين أول عبارتين للتخصيص في قسم الإدخال لتجنب إعادة ترتيب العمليات المعروضة. لاحظ أن حواجز الذاكرة تعتبر عمليات منخفضة المستوى ولا يستخدمها مطورو النواة إلا عند كتابة كود متخصص يضمن الاستبعاد المتبادل.

التزامن بالاعتماد على البنية الفيزيائية Synchronization Hardware

يمكن أن تساعد الأجهزة أحيانًا في حل مشكلات القسم الحرج. حيث توفر بعض أنظمة التشغيل ميزة القفل. وهو هنا، المتغير المشترك هو lock الذي تتم تهيئته إلى false.

تعمل خوارزمية TestAndSet (lock) بهذه الطريقة: فهي تُرجع دائمًا أي قيمة يتم إرسالها إليها وتضبط القفل على القيمة true. ستدخل العملية الأولى إلى القسم الحرج مرة واحدة حيث سيعود TestAndSet (lock) إلى القيمة false وستخرج من الحلقة while. لا يمكن للعمليات الأخرى الدخول الآن حيث تم ضبط القفل على "true" وبالتالي تستمر حلقة while في أن تكون صحيحة.

TSL Instruction

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

Mutual Exclusion implementation with TSL

```
while (true)
{
   while ( TestAndSet (&lock )) /*1. copy LOCK to register and set LOCK to 1
                                  2. was LOCK zero?
                                  3. if it was non zero, LOCK was set, so loop
                                  4. return to caller; critical region entered*/
   ;    /* do nothing
   //critical section
   lock = FALSE;                 //store a 0 in LOCK
   //remainder section           //return to caller
}
```

هنا الاستبعاد المتبادل مكفول: بمجرد خروج العملية الأولى من القسم الحرج، يتم تغيير القفل إلى "false". لذلك، يمكن الآن إدخال العمليات الأخرى واحدة تلو الأخرى.

التقدم: مضمون أيضا. ومع ذلك، بعد العملية الأولى، يمكن أن تدخل أي عملية. لا توجد قائمة انتظار محفوظة، لذلك يمكن أن تدخل أي عملية جديدة تجد القفل خاطئًا مرة أخرى. لذا فإن الانتظار المحدود غير مضمون.

Compare and swap قارن وبادل

تستخدم وظيفة المبادلة قفل ومفتاح متغيرين منطقيين. يتم في البداية تهيئة كل من متغيري القفل والمفتاح إلى خطأ. خوارزمية المبادلة هي نفسها خوارزمية الاختبار والتعيين. تستخدم خوارزمية Swap متغيرًا مؤقتًا لضبط القفل على "true" عندما تدخل العملية القسم الحرج من البرنامج.

دعونا نرى شبيه الكود لخوارزمية التبادل:

Swap Instruction

```
void Swap (boolean *a, boolean *b)
 {
   boolean temp = *a;
   *a = *b;
   *b = temp:
 }
```

Mutual Exclusion implementation with Sawp

```
while (true)
{
    key = TRUE;
     while ( key == TRUE)
          Swap (&lock, &key );
          //critical section
          lock = FALSE;
          //remainder section
}
```

في الكود أعلاه، عندما تدخل عملية P1 القسم الحرج من البرنامج، فإنها تقوم أولاً بتنفيذ حلقة while

```
while ( key == TRUE)
          Swap (&lock, &key );
```

بما أن قيمة المفتاح مضبوطة على "true" قبل حلقة for مباشرة، فإن التبديل (القفل، المفتاح) يبدل قيمة القفل والمفتاح. يصبح القفل صحيحًا ويصبح المفتاح زائفًا. في التكرار التالي لفواصل الحلقة والعملية، يدخل P1 القسم الحرج.

قيمة القفل والمفتاح عند دخول P1 إلى القسم الحرج هي lock = true و key = false.

لنفترض أن عملية أخرى، P2، تحاول الدخول إلى القسم الحرج بينما يكون P1 في القسم الحرج. دعنا نلقي نظرة على ما يحدث إذا حاول P2 الدخول إلى القسم الحرج.

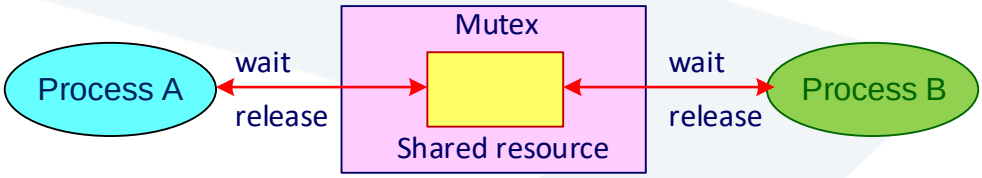
يتم تعيين المفتاح إلى true مرة أخرى بعد تنفيذ أول حلقة while loop، أي while (1). الآن، حلقة while الثانية في البرنامج، أي أثناء فحص (lock). بما أن المفتاح صحيح، تدخل العملية حلقة while الثانية. التبديل (قفل، مفتاح) يتم تنفيذه مرة أخرى. نظرًا لأن كل من المفتاح والقفل صحيحان، فسيكون كلاهما صحيحًا بعد التبديل أيضًا. لذا، فإن الوقت يستمر في التنفيذ وتستمر العملية P2 في تشغيل حلقة while loop حتى تخرج العملية P1 من القسم الحرج وتجعل القفل خاطئًا.

عندما تخرج العملية P1 من القسم الحرج، يتم تعيين قيمة القفل مرة أخرى على "False" حتى تتمكن العمليات الأخرى الآن من الدخول إلى القسم الحرج.

عندما تكون العملية داخل القسم الحرج، لا يتم الاحتفاظ بأي عملية واردة أخرى تحاول إدخال القسم الحرج بأي ترتيب أو قائمة انتظار. لذا فإن أي عملية خارج كل عملية الانتظار يمكن أن تحصل على فرصة لدخول القسم الحرج حيث يصبح القفل False. لذلك، قد تكون هناك عملية قد تنتظر إلى أجل غير مسمى. لذلك، لا يتم ضمان الانتظار المحدود في خوارزمية المبادلة أيضًا.

mutex

وهو كائن استبعاد متبادل يقوم بمزامنة الوصول إلى المورد المشترك. يتم إنشاؤه باسم فريد في بداية البرنامج. تضمن آلية قفل كائن المزامنة (mutex) أن مسلك تنفيذ واحد فقط يمكنه الحصول على كائن المزامنة (mutex) والدخول إلى القسم الحرج الشكل 5. يقوم هذا المسلك بتحرير كائن المزامنة عند الخروج في القسم الحرج فقط.



الشكل 5: الوصول إلى المورد المشترك باستخدام Mutex

مثال

```
wait (mutex);
.....
Critical Section
.....
signal (mutex);
```

يوفر كائن المزامنة (mutex) الاستبعاد المتبادل، سواء كان منتجًا أو مستهلكًا يمكنه الحصول على المفتاح (كائن المزامنة) والمضي قدمًا في عمله. طالما أن المنتج يملأ المخزن المؤقت، يحتاج المستخدم إلى الانتظار والعكس صحيح. لمسلك واحد فقط أن يمتلك قفل Mutex، في أي وقت.

عندما يبدأ البرنامج، فإنه يطلب من النظام إنشاء كائن المزامنة (mutex) لمورد معين. يقوم النظام بإنشاء كائن كائن المزامنة (mutex) باسم فريد أو معرف. عندما يريد مسلك التنفيذ البرنامج استخدام المورد، فإنه يقوم بقفل كائن المزامنة (mutex)، ويستخدم المورد وبعد الاستخدام، فإنه يحرر القفل. ثم يُسمح للعملية التالية بالحصول على قفل كائن المزامنة (mutex).

في غضون ذلك، وبعد أن حصلت عملية على قفل كائن المزامنة (mutex)، فإنه لا يمكن لأي مسلك تنفيذ أو عملية أخرى الوصول إلى هذا المورد. إذا كان كائن المزامنة (mutex) مغلقًا بالفعل، فإن العملية التي ترغب في الحصول على القفل على كائن المزامنة (mutex) يجب أن تنتظر ويتم وضعها في قائمة الانتظار بواسطة النظام حتى يتم تحرير قفل كائن المزامنة (mutex).

فيما يلي المزايا التالية لـ كائن المزامنة (mutex)، مثل:

- Mutex هو مجرد أقفال بسيطة تم الحصول عليها قبل الدخول إلى القسم الحرج ثم تحريره.
- نظرًا لوجود مسلك تنفيذ واحد فقط في قسمه الحرج في أي وقت معين، فلا توجد شروط تسابق، وتظل البيانات دائمًا متسقة.

لدى Mutex أيضًا بعض العيوب، مثل:

- إذا حصل مسلك التنفيذ على قفل وذهب إلى وضع السكون أو تم منعه مسبقًا، فقد لا يتحرك المسلك الآخر للأمام. هذا قد يؤدي إلى المجاعة.
- لا يمكن قفله أو إلغاء تأمينه من سياق مختلف عن السياق الذي حصل عليه.
- يجب السماح بمسلك تنفيذ واحد فقط في القسم الحرج في كل مرة.
- قد يؤدي التنفيذ العادي إلى الانتظار في حالة المشغولية، مما يضيع وقت وحدة المعالجة المركزية.

الإشارات Semaphores

في عام 1965، اقترح Dijkstra تقنية جديدة وهامة للغاية لإدارة العمليات المتزامنة باستخدام قيمة متغير عدد صحيح بسيط لمزامنة تقدم العمليات المتفاعلة. يسمى هذا المتغير الصحيح Semaphore. لذلك فهي في الأساس أداة مزامنة ولا يمكن الوصول إليها إلا من خلال عمليتين ذريتين قياسيتين **wait** and **signal** يشار إليهما بـ P(S) and V(S) على التوالي.

بكلمات بسيطة للغاية، السيمافور هو متغير يمكنه الاحتفاظ فقط بقيمة صحيحة غير سالبة، مشتركة بين جميع مسالك التنفيذ، مع تنفيذ **wait** and **signal**، والتي تعمل على النحو التالي:

```
P(S): if S >= 1 then S := S - 1
      else <block and enqueue the process>;
V(S): if <some process is blocked on the queue>
      then <unblock a process>
      else S := S + 1;
```

التعريف التقليدي لـ **wait** and **signal** هو:

- **wait**: تعمل هذه العملية على انقاص قيمة الوسيط S الخاصة بها بمجرد أن تصبح غير سالبة (أكبر من أو تساوي 1). تساعدك هذه العملية بشكل أساسي على التحكم في إدخال مهمة ما في القسم الحرج. في حالة القيمة السالبة أو الصفرية، لا يتم تنفيذ أي عملية. Wait() كانت العملية في الأصل تسمى P ؛ لذلك تُعرف أيضًا باسم عملية P(S). تعريف عملية الانتظار كما يلي:

```
wait(S)
{
    while (S<=0);//no operation
    S--;
}
```

ملحوظة:

عندما تقوم إحدى العمليات بتعديل قيمة السيمافور، فلا يمكن لأي عملية أخرى أن تعدل في نفس الوقت قيمة السيمافور نفسها. في الحالة المذكورة أعلاه، يجب تنفيذ القيمة الصحيحة (S <=0) S وكذلك التعديل المحتمل الذي هو S – دون أي انقطاع.

- **signal()**: تزيد من قيمة الوسيط S الخاصة بها، حيث لم تعد هناك عملية محظورة في قائمة الانتظار. تُستخدم هذه العملية بشكل أساسي للتحكم في خروج مهمة ما من القسم الحرج. لذلك تُعرف أيضًا باسم عملية V (S). تعريف عملية **signal** على النحو التالي:

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

لاحظ أيضًا أنه يجب تنفيذ جميع التعديلات على القيمة الصحيحة للإشارة في عمليات Wait() و **signal()** بشكل غير قابل للتجزئة.

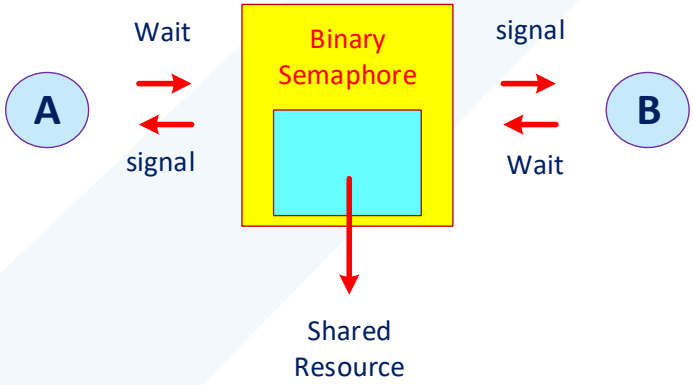
خصائص Semaphores

- إنها بسيطة ولها دائمًا قيمة عدد صحيح غير سالب.
- يعمل مع العديد من العمليات.
- يمكن أن يكون لها أقسام حرجة مختلفة مع إشارات مختلفة.
- كل قسم حرج له إشارات وصول فريدة.
- يمكن أن تسمح بعمليات متعددة في القسم الحرج دفعة واحدة، إذا كان ذلك مرغوبًا فيه.

أنواع السيمافور

السيمافور الثنائي Binary semaphore:

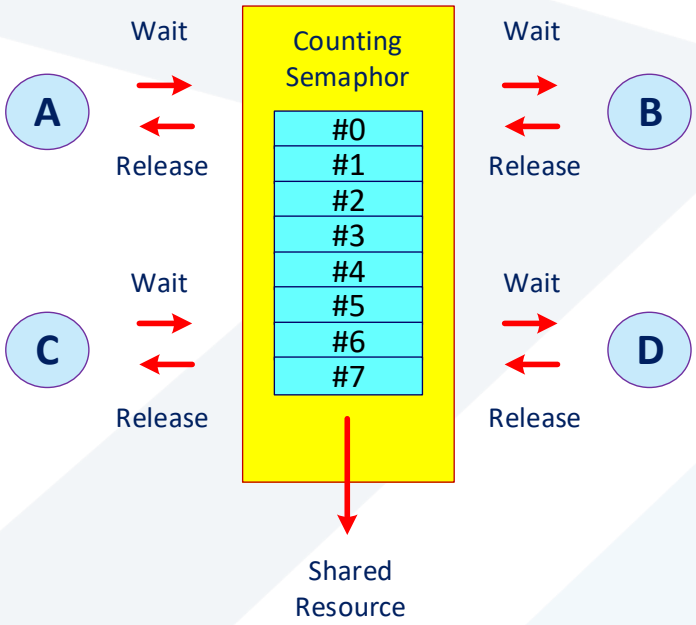
إنه شكل خاص من السمافور يستخدم لتنفيذ الاستبعاد المتبادل، ومن ثم يطلق عليه غالبًا اسم Mutex. تتم تهيئة السيمافور الثنائي إلى 1 ويأخذ القيمتين 0 و 1 فقط أثناء تنفيذ البرنامج كما في الشكل 6. في Binary Semaphore، لا تعمل عملية الانتظار إلا إذا كانت قيمة السيمافور = 1، وتنجح عملية الإشارة عندما تكون قيمة السيمافور = 0. يكون تنفيذ السيمافورات الثنائية أسهل في التنفيذ من السيمافور العداد.



الشكل 6: السيمافور الثنائي

السيمافور العداد Counting semaphore:

يمكن أن تتراوح سيمافورات العد عبر مجال غير مقيد. يمكن استخدامها للتحكم في الوصول إلى مورد معين يتكون من عدد محدود من الموارد كما في الشكل 7. هنا يتم استخدام عدد الإشارات للإشارة إلى عدد الموارد المتاحة. إذا تمت إضافة الموارد، فسيتم زيادة عدد السيمافورات تلقائيًا وإذا تمت إزالة الموارد، فسيتم إنقاص العدد تلقائيا. لا يوجد استبعاد متبادل في سيمافورات العد.



الشكل 7: السيمافور العداد

فوائد استخدام Semaphores كما هو موضح أدناه:

- بمساعدة السيمافور، هناك إدارة مرنة للموارد.
- السيمافورات مستقلة عن التجهيزات ويجب تشغيلها في الكود المستقل في النواة.
- لا تسمح Semaphores لعمليات متعددة بالدخول في القسم الحرج.
- أنها تسمح لأكثر من مسلك تنفيذ واحد للوصول إلى القسم الحرج.
- • حيث أن السيمافورات تتبع مبدأ الاستبعاد المتبادل بصرامة وهذه أكثر فاعلية من بعض طرق المزامنة الأخرى.
- • لا يوجد هدر للموارد في السيمافورات ناتج عن الانتظار المشغول في السيمافورات حيث لا يتم إهدار وقت المعالج دون داع للتحقق مما إذا كان أي شرط قد تم الوفاء به من أجل السماح للعملية بالوصول إلى القسم الحرج.

مساوئ السيمافورات هي

- أحد أكبر القيود هو أن الإشارات قد تؤدي إلى انعكاس الأولوية ؛ حيث قد تصل العمليات ذات الأولوية المنخفضة إلى القسم الحرج أولاً وقد تصل العمليات ذات الأولوية العالية إلى القسم الحرج لاحقًا.
- لتجنب حالات الجمود deadlocks في السيمافور، يجب تنفيذ عمليتي الانتظار والإشارة بالترتيب الصحيح.
- استخدام السيمافورات على نطاق واسع غير عملي. لأن استخدامها يؤدي إلى فقدان النمطية modularity وهذا يحدث لأن عمليات **wait** and **signal** تمنع إنشاء تخطيط منظم لمثل هذه الأنظمة.
- مع الاستخدام غير السليم، قد تتوقف العملية إلى أجل غير مسمى. مثل هذا الوضع يسمى الجمودdeadlocks .