جامعة حلب - كلية الهندسة المعلوماتية
سنة رابعة – كافة الأقسام

Parallel Programming Course

Lecture (9)

Locks and Barriers

تمهيد

- تستخدم البرمجة المتواقتة نوعين من التزامن:
  - Mutual exclusion
    - ويبنى بواسطة الأقفال (*locks*) التي تحمي المقاطع الحرجة من البرنامج.
  - Condition synchronization
- في هذه المحاضرة سنختبر مسألتين مهمتين هما المقطع الحرج والحاجز (*critical section and barrier*).
  - الـ*critical section* فيهتم في بناء الأحداث الجزيئية في البرنامج.
  - الـbarrier فهي نقطة تزامن والتي يجب على جميع الإجرائيات الوصول إليها قبل أن يؤذن لأحدها بالمتابعة.

مسألة المقطع الحرج (*Critical section problem*)

- تعتبر من المسائل الكلاسيكية ولقد كانت من المسائل الأولى التي دُرست بشكل موسع ولازالت لأن معظم البرامج المتواقتة تحوي مقاطع حرجة.
- حل هذه المسألة يستخدم لبناء عبارات *await* عشوائية.
- في هذه الفقرة نعرف المسألة ونقوم بتطوير حل على المستوى الخشن (*coarse grain*).
- في مسألة المقطع الحرج يوجد n إجرائية تكرر تنفيذ مقطع حرج بعدها المقطع الغير حرج.
- المقطع الحرج يتابع بواسطة برتوكول دخول يتبعه بروتوكول خروج.

وهكذا نفترض بأن الإجرائيات تملك الشكل التالي:

```
Process CS[i = 1 to n] {
    while (true) {
        entry protocol;
        critical section;
        exit protocol
        nocritical section
        }
    }
```

- كل قسم حرج عبارة عن سلسلة من العبارات التي تخترق بعض العناصر المشتركة.
- كل مقطع غير حرج هو سلسلة أخرى من العبارات.
- نفترض بأن الإجرائية التي تدخل مقطعها الحرج سوف تخرج منه في النتيجة. لذلك فإن الإجرائية يمكن لها أن تتم فقط خارج قسمها الحرج.

- هدفنا هو تصميم بروتوكولات دخول وخروج تحقق الخواص التالية:

1. Mutual exclusion
   - بمعنى أنه في كل مرة إجرائية واحدة على الاكثر تنفذ قسمها الحرج.
2. Absence Deadlock (livelock)
   - إذا حاولت إجرائيتين أو أكثر الدخول إلى قسمها الحرج، ستنجح واحدة على الأقل.
3. Absence of unnecessary Delay
   - إذا حاولت إجرائية الدخول إلى قسمها الحرج والإجرائيات الأخرى تنفذ أقسامها غير الحرجة أو أنها تمت، فإن الإجرائية الأولى لا تمنع من دخولها لقسمها الحرج.
4. Eventual Entry
   - الإجرائية التي تحاول دخول قسمها الحرج سوف تنجح في النتيجة.

- الخاصيات الثلاثة الأولى فهي للأمان(*safety*)، لكن الرابعة خاصية للحياة(*liveness*).

- من أجل مبدأ استبعاد التشارك(*ME*)، الحالة السيئة هي الحالة التي يكون فيها إجرائيتين في قسمهما الحرج.
- أما تغيب القفل الميت(*Absence of Deadlock*)، الحالة السيئة هي الحالة التي تكون فيها جميع الإجرائيات في حالة انتظار للدخول، ولكن أحداً لن يستطيع الدخول.
  - تدعى هذه الحالة غياب الحياة في مشغولية الانتظار(*busy waiting*)، لأن الإجرائيات على قيد الحياة ولكن في حلقة لانهائية.
- أما غياب التأخير اللاضروري(*Absence of Unnecessary Delay*)، الحالة السيئة هي التي يكون فيها أحدى الإجرائيات التي تريد الدخول لا تستطيع ذلك، حتى لو كان لا يوجد اجرائية أخرى في قسمها الحرج.
- الدخول المنتج(*Eventual Entry*) هي خاصية للحياة بما أنها تعتمد على سياسة الشدولة.

- الطريقة التافهة لحل مسألة القسم الحرج بإدراج كل قسم حرج ضمن أقواس الزاوية، باستخدام عبارة await غير مشروطة.
- ME يتبع مباشرة من قواعد الأقواس الزاوية.
- أما الخصائص الثلاثة الأخرى يمكن تحقيقها إذا كانت الشدولة عادلة بدون شروط بما أن سياسة الشدولة تلك تؤكد أن محاولة احدى الإجرائيات لتنفيذ حدثها الجزيئي التابع لقسمها الحرج ستحصل عليه بالنتيجة، مهما كان عمل الإجرائيات الأخرى.
- لكن هذا الحل يرجو مسألة كيف يتم بناء الأقواس الزاوية.

- على الرغم من أن الخصائص الأربعة مطلوبة، إلا أن خاصية *ME* تعتبر الأكثر أهمية. ولذلك، سنركز عليها أولاً بعدها سنرى كيف نحقق الخصائص الاخرى.
- لكي نوصف مبدأ *ME* ، نحتاج طريقة تخبرنا فيما إذا كانت الإجرائية في قسمها الحرج.
- لتبسيط ملاحظتنا قمنا بتطوير حل لإجرائيتين CS1 و CS2؛ يمكن تعميم الحل على n إجرائية.
- نفرض أن in1 و in2 متحولين بوليين مهيئين بـF في البداية.
- عندما يكون CS1(CS2) في قسمه الحرج، نهيئ in1(in2) بـT.
- الحالة السيئة التي نريد التغلب عليها هي عندما in1 و in2 هي T.

- وهكذا نريد من كل الحالات أن ترفض الحالة السيئة:

$$\text{MUTEX: } \neg(in1 \wedge in2)$$

- الـMUTEX predicate يجب أن تكون لامتغير عام ولكي تكون كذلك يجب أن تتحقق في الحالة البدائية وبعد كل تكليف لكل من in1 و in2.
- أي قبل أن تدخل الإجرائية CS1 قسمها الحرج ،عندئذ تهيئ in1 بـT، تحتاج للتأكد من أن in2 هي F.
- يتحقق هذا باستخدام حدث جزيئي مشروط :

$$\langle \text{await } (!in2)\ in1 = true; \rangle$$

- الإجرائيتين متماثلتين، لذلك نستخدم نفس النوع من الحدث الجزيئي المشروط لبروتوكول الدخول في الإجرائية CS2.
- ماذا عن بروتوكول الخروج ؟
  - لا ضرورة مطلقاً للتأخير عند مغادرة المقطع الحرج، لذلك لا نحتاج إلى حراسة التكليف الذي يهيئ in1 و in2 بـF.

تحقيق البرنامج للخصائص الأربعة المطلوبة

- مبدأ ME محقق بسبب بنية البرنامج.
- مبدأ تفادي القفل الميت لأنه إذا تم حجز كل إجرائية عند برتوكول الدخول، عندها كلا in1 و in2 يكون محققاً T، وهذا يتعارض مع الحقيقة بأن كليهما يجب أن يكون Fعند هذه النقطة من الكود.
- تم تفادي التأخير غير الضروري لأن أحد الإجرائيات يكون محجوزاً عندما لا تكون الإجرائية الأخرى في قسمها الحرج.
- أخيراً، اعتبر خاصية الحياة عندما تحاول إجرائية الدخول إلى قسمها الحرج تكون في آخر الأمر قابلة لفعل ذلك.
- إذا كانت cs1 تحاول الدخول لكنها لا تستطيع، عندها in2 تكون محققة، لذا cs2 تكون في قسمها الحرج.

- بفرض أن الإجرائية في قسمها الحرج ستخرج في النتيجة، in2 ستصبح بالنتيجة غير محققة وبالتالي حارس الدخول لـcs1 يصبح محققاً.
- إذا بقيت CS1 غير مسموح دخولها، فإنه إما بسبب المشدول غير عادل أو بسبب CS2 مرة ثانية ربحت الدخول إلى قسمها الحرج.
- في الحالة الأخيرة، السيناريو أعلاه يكرر، وهكذا بالنتيجة in2 يصبح F.
- وهكذا، in2 يبقى F لفترة غير محددة غالباً (أو CS2 يتوقف، بهذه الحالة in2 تصبح وتبقى F).
- مطلوب سياسة شدولة حكيمة جداً للتأكد بأن CS1 بالنتيجة سيربح الدخول في أي من الحالات. المناقشة بالنسبة لـ CS2 مشابهة. وعلى أية حال فإن المشدول القوي العادل غير قابل للتطبيق.

Critical section problem: Coarse-grained solution

```
bool in1 = false, in2 = false;
## MUTEX: ¬(in1 ∧ in2) – global invariant
process cs1 {
    while (true) {
        ⟨await (!in2) in1 = true;⟩      /* entry */
        critical section;
        in2 = false;                    /* exit */
        noncritical section:
    }
}
```

Critical section problem: Coarse-grained solution

```
process cs2        {
    while (true)   {
        ⟨await (!in1) in2 = true;⟩    /* entry */
        critical section;
        in2 = false;                  /* exit */
        noncritical section;
    }
}
```

Critical Sections: Spin Locks

- يستخدم البرنامج متحولين. لتعميم الحل على n إجرائية، علينا استعمال n متحول.
- على أية حال هناك حالتين من اهتمامنا:
  - أحدى الإجرائيات في قسمها الحرج.
  - ولا إجرائية في قسمها الحرج.
- متحول واحد كاف للتمييز بين الحالتين، مستقل عن عدد الإجرائيات.
- نفترض أن lock عبارة عن متحول بولي والذي يشير متى تكون الإجرائية في القسم الحرج.
- بمعنى lock يكون T عندما يكون أي من in1 أو in2 هو T، و F في الحالات الأخرى.
- لدينا المتطلبات التالية:

$$lock == (in1 \vee in2)$$

- باستخدام القفل Lock مكان in1 و in2 ، فإن برتوكولات الدخول والخروج يصبح البرنامج بالشكل التالي:

Critical sections using locks

```
bool lock = false;
process cs1 {
    while (true) {
        ⟨await (!lock) lock = true;⟩   /* entry */
        critical section;
        lock = false;                  /* exit */
        noncritical section:
    }
}
```

Critical sections using locks

```
process cs2 {
    while (true)        {
        ⟨await (!lock) lock = true;⟩     /* entry */
        critical section;
        lock = false;                    /* exit */
        noncritical section;
    }
}
```

- مزية بروتوكولات الدخول والخروج في البرنامج الجديد بالنسبة لها في البرنامج السابق هو أنه يمكن استخدامها لحل مسألة القسم الحرج لأي عدد من الإجرائيات، ليس فقط اثنتان.
- أي عدد من الإجرائيات يمكن أن تشارك lock وتنفذ نفس البروتوكول.

Test and Set

- أهمية التغير في المتحولات هو أنه في الغالب جميع النظم خاصة متعددة المعالجات، تملك تعليمات خاصة يمكن أن تستعمل لبناء الأحداث الجزيئية المشروطة.
- نستخدم هنا واحدة منها تدعى Test and Set (TS) وفي الفقرة القادمة سنستخدم Fetch and Add.
- تعليمة TS تأخذ متحول مشترك هو lock كـargument ترجع نتيجة بولية (F/T).
- كحدث جزيئي، فإن TS يقرأ ويخزن قيمة الـlock، يهيئه بـT، بعدها يرجع القيمة البدائية المخزنة لـlock.

- تأثير التعليمة TS يستحصل من التابع التالي:

```
bool TS(bool lock) {
        ⟨ bool initial = lock;  /* save initial value */
        lock = true;            /* set lock */
        return initial; ⟩       /* return initial value */
    }
```

- وهكذا باستعمال TS، نستطيع بناء الحل السابق بالخوارزمية التي سترد في الشريحة التالية.
- بمعنى أن الأحداث الجزيئية المشروطة في الحل السابق يستبدل بحلقات لا تتوقف حتى يصبح الـlock هو F ، عندها ترجع TS المتحول F.
- بما أن جميع الإجرائيات تنفذ نفس البروتوكولات، فالحل كما ترونه يعمل من أجل أي عدد من الإجرائيات.
- عندما نستعمل متحول lock كما في البرنامج التالي، فإنه يدعى نموذجياً بـspin lock.
  - هذا لأن الإجرائيات تحافظ على دورانها بينما تنتظر lock ليصبح F.

Critical Sections using Test and Set

```
bool lock = false;                /* shared lock */
process CS[i = 1 to n] {
    while (true) {
        while (TS(lock)) skip;  /* entry protocol */
        critical section;
        lock = false;             /* exit protocol */
        noncritical section;
        }
    }
```

- البرنامج السابق يحمل الصفات التالية:
- *Mutual Exclusion* مؤكدة لأنه، إذا ما حاولت إجرائيتين أو أكثر الدخول إلى قسمهم الحرج، فقط واحدة منهما ستنجح كونها الأولى في تغيير قيمة الـ*lock* من *F* إلى *T*؛ لذلك، فإن إجرائية واحدة فقط سُتتم برتوكول دخولها.
- *Absence of Deadlock* ينتج من الحقيقة، أنه إذا كانت كلا الإجرائيتين في بروتوكول دخولها، *lock* يكون *F*، وبالتالي واحدة من الإجرائيتين ستنجح في الدخول إلى قسمها الحرج.
- *Unnecessary Delay* يُتجنب بسبب، إذا كانت كلا الإجرائيتين خارج قسمها الحرج، *lock* هو *F*، ولذلك فإنها أحداها يمكن أن تدخل قسمها الحرج إذا كانت الآخرى تنفذ قسمها غير الحرج، أو أنها تمت.

- *eventual entry* بالمقابل ليس مضموناً بالضرورة.
  - إذا كانت الجدولة مناسبة بشكل قوي، فإذا حاولت إجرائية دخول قسمها الحرج ستنجح بالنهاية، لأن lock يصبح في حالة F لفترة في الغالب غير محدودة.
  - إذا كانت الجدولة مناسبة بشكل ضعيف، وهي الحالة العامة، عندها يمكن للإجرائية أن تدور في مكانها إلى مالا نهاية في بروتوكول دخولها.
    - لكن هذا يمكن أن يحدث فقط إذا كان يوجد دائماً إجرائيات أخرى تحاول ومن ثم تنجح بالدخول إلى قسمها الحرج، ولن تكون هذه هي الحالة العملية.
  - وبهذا يكون الحل السابق مناسباً.

- الحل السابق يمكن استخدامه في أي آلة تملك تعليمات تختبر وتعدل متحولات مشتركة كحدث جزيئي مفرد.
  - مثال تعليمة الزيادة تقوم بزيادة عدد صحيح وأيضاً تهيئ شرط يشير إلى النتيجة موجبة أو سالبة.
    - باستخدام هذه التعليمة فإن بروتوكول الدخول يمكن أن يعتمد على التحول من الصفر إلى الواحد.
  - حلول الـspin-lock السابقة تحمل خاصية برتوكول خروج وهو ببساطة إعادة المتحولات المشتركة إلى قيمها الابتدائية.

Test and Test and Set protocol (TTS)

- على الرغم من أن الحل السابق صحيحاً، التجارب في انظمة المعالجات المتعددة أظهرت أن تعطي أداءً متدنياً إذا تنافست عدة إجرائيات للدخول إلى القسم الحرج.
  - لأن *lock* متحول مشترك وكل إجرائية مؤخرة ترجع باستمرار إلى هذا المتحول. مما يسبب نزاع ذاكري، والذي يشوه أداء وحدات الذاكرة وشبكات الربط بين المعالج والذاكرة.
  - إضافةً إلى أن، تعليمة *TS* تكتب إلى المتحول *lock* كلما ُتنفذ، حتى عندما لا تتغير قيمتها.
- بما أن الأنظمة متعددة المعالجات ذات الذاكرة المشتركة تستخدم الذاكرة cache لتخفيض الاتصالات بالذاكرة الأولية، هذا يجعل *TS* أكثر كلفةً من تعليمة فقط تقرأ متحول مشترك.
  - لأنه عندما يكتب إلى متحول بواسطة أحدى المعالجات، فإن الـ*cache* في المعالجات الأخرى يحتاج إلى الغاء أو تغيير إذا ما احتوت على نسخة منه.

- كلفة التصادم الذاكري وإلغاء الـ*cache* يمكن أن يخفض عن طريق تعديل برتوكول الدخول للقسم الحرج.
- بدلاً من الدوران حتى تصبح تعليمة *TS* محققة، يمكننا زيادة الاحتمالية التي تعود بتحقيق *TS* باستخدام البرتوكول التالي:

```
While (lock) skip;          /* spin while lock set */
While (TS(lock)) {          /* try to grab the lock */
    While (lock) skip;      /* spin again if fail */
    }
```

- هذا البروتوكول يسمى *Test-and-Test-and-Set* لأن الإجرائية تختبر فقط *lock* حتى يكون هناك امكانية أن *TS* ينجح.
- في الحلقتين الإضافيتين، يختبر *lock*، لذا يمكن قراءة قيمته من *cache* محلي بدون تأثير على المعالجات الأخرى. لذا فإن النزاع الذاكري ينخفض، ولكنه لا يختفي.
- بمعنى أنه عندما *lock* يصبح *T* فإن إجرائية واحدة على الأقل ويمكن لجميع الإجرائيات المنتظرة تنفذ *TS*، على الرغم من أن إجرائية واحدة فقط ستتابع.

- الشكل التالي يعرض الحل الكامل لمسألة القسم الحرج باستعمال بروتوكول الدخول *TTS* أما بروتوكول الخروج فهو فقط يعيد الـ *lock* إلى حالة *F*.

```
bool lock = false;                    /* shared lock */
process CS[i = 1 to n] {
    while (true) {
        while (lock) skip;            /* entry protocol */
        while (TS(lock)) {
                while (lock) skip;
                }
        critical section;
        lock = false;                 /* exit protocol */
        noncritical section;
        }
    }
```

Implementing Await Statements

- إن أي حل لمسألة القسم الحرج يمكن أن يستعمل لبناء حدث جزيئي غير مشروط ⟨ S; ⟩ عن طريق تغطية نقاط التحكم الداخلية عن الإجرائيات الأخرى.
- ليكن *CSenter* بروتوكول دخول المقطع الحرج وأن *CSexit* بروتوكول الخروج المقابل. عندها

```
CSenter;
S;
CSexit;
```

- يفترض هذا الحل أن جميع أكواد الحلول في جميع الإجرائيات التي ترجع أو تغير متحولات تتغير عن طريق S، أو بأنها تغير متحولات ترجع إلى S، هي أيضاً محمية ببروتوكولات دخول وخروج مشابه.
  - بشكل أساسي الرمز ⟩ استبدل بـ*CSenter* و الرمز ⟨ استبدل بـ*CSexit*.

- يمكن استخدام الكود أعلاه لبناء حدث جزيئي مشروط ⟨await (B) S;⟩ حيث أن الحدث الجزيئي المشروط يتأخر حتى يصبح B محققاً، عندها ينفذ S. أيضا B يجب أن يكون محققاً عندما يبدأ تنفيذ S.
- للتأكد من أن كامل الحدث هو جزيئي، نستطيع استعمال بروتوكول الحدث الجزيئي لتغطية الحالات الانتقالية في S. يمكننا بعدها استعمال حلقة لكي تختبر B تكرارياً حتى تصبح محققة بالشكل التالي:

```
CSenter;
While (!B) { ???}
S;
CSexit;
```

- أيضاً هنا نفترض بأن الأقسام الحرجة في جميع الإجرائيات التي تغير المتحولات المرجعية في B أو S أو التي ترجع بمتحولات تعدل في S محمية ببرتوكولات دخول وخروج مشابه.

- بقي أن نعرف كيف يمكن بناء جسم الحلقة.
- إذا نفذ جسم الحلقة، بمعنى أن B كان غير محقق. لذلك، الطريق الوحيد الذي يكون فيه B محققاً هو إذا ما غيرت أحد الإجرائيات الأخرى متحول ما يعود بمرجعيته إلى B.
- بما أننا نفترض أن أي حالة في إجرائية أخرى التي تغير متحول يرجع في B يجب أن يكون في مقطع حرج، علينا الخروج من القسم الحرج أثناء انتظارنا كي تصبح B محققة.
- لكن لتأكيد الجزيئية في تحقيق B وتنفيذ S، يجب علينا إعادة دخول القسم الحرج قبل إعادة تحقيق B. وبهذا يصبح البرتوكول بالشكل:

```
CSenter;
While (!B) { CSexit; CSenter; }
S;
CSexit;
```

- هذه الطريقة تحافظ على المعنى الدلالي للأحداث الجزيئية المشروطة، بافتراض أن بروتوكولات المقطع الحرج تضمن ME.
  - إذا كان المجدول الزمني ضعيف الإنصاف، فإن الإجرائية التي تنفذ البرتوكول السابق سوف تتم الحلقة بالنتيجة، على افتراض أن B بالنتيجة ستتحقق وتبقى محققة.
  - إذا كان المجدول الزمني قوي الإنصاف، فالحلقة ستتم إذا أصبح B محققاً لفترة غير محددة في الغالب.
- على الرغم من أن البرتوكول صحيح، إلا أنه غير فعال.
  - هذا لأن الإجرائية تنفذ البرتوكول السابق تدور في حلقة صعبة
    - تخرج باستمرار، بعدها تعاود دخول مقطعها الحرج
  - حتى لو انها لن تستطيع المتابعة حتى يغير أحد الإجرائيات الأخرى على الأقل متحول يعود بمرجعيته إلى B.
  - هذا يقود إلى نزاع ذاكري بما أن كل إجرائية منتظرة تدخل إلى المتحول المستخدم في بروتوكول المقطع الحرج والمتحولات في B باستمرار.

- لإنقاص نزاع الذاكرة، يفضل تأخير الإجرائية لفترة من الزمن قبل معاودة الدخول إلى المقطع الحرج. يمكن أن يكون التأخير عبارة كود يبطئ الإجرائية عندها يصبح البروتوكول بالشكل:

```
CSenter;
While (!B) { CSexit; Delay; CSenter;}
S;
CSexit;
```

- كود التأخير يمكن أن يكون حلقة مفرغة تتكرر عدد من المرات العشوائية.
  - لتجنب النزاع الذاكري في هذه الحلقة، فإن كود التأخير يجب أن يدخل إلى متحولات محلية فقط.

- هذا النوع من البروتوكول يستخدم ضمن بروتوكول الدخول *CSenter*
  - مثال يمكن استخدامها في مكان *skip* في حلقة تأخير في بروتوكول الدخول *TS*.
- إذا كانت S هي فقط عبارة *skip*، يمكن عندها ببساطة حذف S.
- إذا كانت B أيضاً تحقق متطلبات خاصية *at-most-once* عندها ⟨await (B);⟩ تصبح بالشكل:

```
While (!B) skip;
```

- كما ذكرنا التزامن في الانتظار المشغول عادة ما يستخدم بالقسم الصلب.
- في الحقيقة فإن برتوكول مشابه للسابق يستخدم في تحقيق تزامن الدخول إلى الـ*Ethernet* (شبكة اتصال محلية).
- لإرسال رسالة، يقوم متحكم *Ethernet* بإرسالها عبر *Ethernet* ثم ينتظر مستمعاً ليرى إذا كانت قد اصطدمت مع رسالة أخرى عند نفس اللحظة من قبل متحكم آخر.
- إذا لم يتم التصادم، فإن الإرسال ُيفترض أن يكون قد تم.
- إذا حدث التصادم، يقوم المتحكم بالانتظار برهة، ثم يحاول اعادة ارسال الرسالة.
- للتغلب على شرط السباق الذي فيه يتصادم المتحكمين بشكل متكرر كونهما يؤخران دائماً بنفس الفترة الزمنية، يتم اختيار التأخير عشوائياً وتضاعف في كل مرة يحدث فيها التصادم ويدعى هذا البروتوكول *binary exponential back-off-protocol*.
- أثبتت التجربة أن هذا البرتوكول مفيد في بروتوكولات دخول المقطع الحرج.

Critical sections: Fair Solutions
المقاطع الحرجة : الحلول المنصفة (الخالية من العيوب)

- إن حلول القفل الدوار *spin-lock* لمسألة المقطع الحرج تؤكد خلوها من *ME* و *Deadlock* (*livelock*) وتجنبها للتأخير اللاضروري.
- لكنها تتطلب مجدول مهام زمني شديد الإنصاف لتأكيد خاصية *Eventual Entry*.
- كما لوحظ بأن سياسة الجدولة العملية تحقق فقط ضعيفة الإنصاف.
- على الرغم من أنه من غير المحتمل أن إجرائية تحاول الدخول إلى قسمها الحرج أن لا تنجح أبداً، يمكن أن يحدث إذا كانت إجرائيتين أو أكثر تتنافسان دائماً على الدخول.
- حلول القفل الدوار لا تتحكم بترتيب الذي فيه الإجرائيات المتأخرة تدخل أقسامها الحرجة عندما يكون هناك اثنتين أو اكثر تحاول فعل ذلك.

- نستعرض فيما يلي ثلاث حلول عادلة لمسألة القسم الحرج:
  - Tie-Breaker
  - Ticket algorithm
  - Bakery Algorithm
- تعتمد فقط على مجدول ضعيف الإنصاف مثل round-robin، والذي يتأكد فقط من أن كل إجرائية تستمر في حصولها على فرصة في التنفيذ وأن شروط التأخير، حالما تصبح محققة، تبقى محققة.
- خوارزمية Tie-breaker بسيطة لإجرائيتين ولا تعتمد على تعليمات آلة معينة، لكنها معقدة من أجل n إجرائية.
- خوارزمية Ticket بسيطة من أجل أي عدد من الإجرائيات، لكنها تتطلب تعليمات آلة خاصة تدعى Fetch-and-Add.
- خوارزمية Bakery تختلف عن ticket بأنها لا تتطلب تعليمات آلة خاصة، لكنها بالنتيجة أكثر تعقيداً (على الرغم من أنها تبقى أبسط من Tie-Breaker لـn إجرائية.

الوظيفة