الباب الرابع

PRINCIPLES OF CONCURRENCY

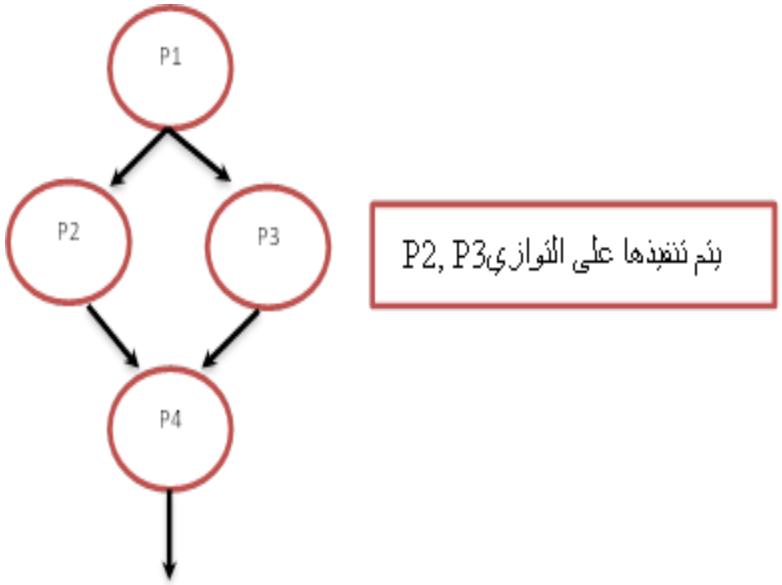
قواعد او اساسيات التزامن

3.2 مفهوم التوازي Concurrency

- **نقصد بالتوازي تنفيذ أكثر من عملية في وقت واحد**. قد تكون هذه العمليات المتوازية **مستقلة** عن بعضها البعض (Independent processes) أو **متعاونة** مع بعضها البعض (cooperative processes) .
- التوازي قد يطبق على **مستوى العمليات** (concurrent processes) أو على **مستوى الخيوط** (concurrent threads)
- **العملية المستقلة** هي التي لا تؤثر ولا تتأثر بالعمليات التي تعمل معها في نفس الوقت.
- العملية **تكون متعاونة** إذا أثرت و/ أو تأثرت بالعمليات الأخرى التي تعمل معها بالتوازي.

3.3 تعاون العمليات

- إذا احتاجت **العمليات المتوازية** أن تتشارك في بعض البيانات أو أن تؤدي عملا مشتركا، فلابد لها من أن تتعاون مع بعضها البعض .هذا التعاون يتطلب **اتصال** بين العمليات و**عملية تزامن** (**synchronization**).
- **الاتصال** بين العمليات يتم عن طريق **متغيرات مشتركة** أو **بتبادل الرسائل** (message passing).
- **والتزامن** نحتاجه ليتم الاتصال في الوقت المناسب (الشكل 3.1)، ويتحقق بوضع **نقاط تزامن للعمليات** بحيث إذا سبقت عملية بقية العمليات ووصلت نقطة تزامن، عليها أن تنتظر بقية العمليات الأخرى لتصل هذه النقطة، وإلا سيكون لدينا نتائج غير متوقعة.



الشكل 3.1 تسلسل توازي العمليات

هناك عدة اسباب تستدعي وجود آلية لضمان تعاون العمليات:

- **طلب عملية لخدمة من عملية أخرى**: في هذه الحالة على العملية التي طلبت الخدمة انتظار العملية التي طلب منها الخدمة حتى تفرغ من أدائها. مثلا إذا طلبت عملية من نظام التشغيل معلومة معينة من القرص الصلب، ستظل هذه العملية في انتظار المعلومة حتى تصلها من القرص إلى الذاكرة.
- **زيادة سرعة التنفيذ**: وذلك **بتقسيم مهمة واحدة كبيرة على عدة عمليات** يتم تنفيذها بالتوازي. مثلا إذا استخدمنا **الأنابيب الانسيابية (pipeline)** بين العمليات فهذا يعني أن **مخرج** العملية الأولى سيكون **مدخل** للعملية الثانية، في هذه الحالة لا يمكن للعملية الثانية أن تسبق العملية الأولى.

مثلا الأمر التالي (في UNIX)

$ ls | wc

هو أمر لتنفيذ عمليتين هما **ls** التي تعرض **محتويات الدليل الحالي**، و **wc لعد الكلمات**، هاتان العمليتان ستنفذان في وقت واحد (**بالتوازي**) ، وسيتم بينهم **تزامن** (ستنتظر العملية الثانية **wc** مخرجات العملية الأولى **ls** لتعمل عليها لتحسب عدد كلمات المخرج)

- مثلا في نظام **الوسائط المتعددة** يمكن أن تقوم عملية بإحضار أجزاء المادة الوسائطية من القرص ووضعها في **ذاكرة مؤقتة (خازن Buffer)** ، بينما عملية أخرى تأخذ المادة من **الخازن** وترسلها إلى جهاز الذي يصدر **الصوت/الصورة**. هنا تعتبر العملية الأولى **منتج (Producer)** بينما العملية الثانية **مستهلك (Consumer)** وعلى المنتج العمل على جعل الخازن ممتلئا دوما بحيث لا يتوقف المستهلك عن العمل (لا يجد الخازن فارغا)، لأن المستهلك إذا وجد الخازن فارغا سيتوقف عن العمل وينتظر وصول بيانات للخازن مما يتسبب في تقطع مقطع الفيديو/الصوت أو عدم وضوحه

ويمكن تلخيصها في النقاط التالية :

1. **المشاركة في البيانات (Information sharing)** : مثل الاحتياج لنفس الملف.
2. **زيادة سرعة الحسابات (Computation speedup)**: تقسيم المسألة الى عدة اجزاء يمكن تنفيذهم على التوازي (Parallel processes).
3. **الوحدات (Modularity)**: بناء النظام على هيئة **وحدات او طبقات** يمكن التعامل معها على حدة لتسهيل التوسع في بناء النظام ومعرفة اماكن الاخطاء وسهولة معالجتها.
4. **الملائمة (Convenience)**: يمكن لمستخدم واحد تشغيل اكثر من برنامج- طباعة، تحرير رسالة، ترجمة لغة برمجة،.. (editing, printing, compiling..).

أيا كان نوع التعاون بين العمليات، فيجب أن يكون هنالك:

- **تزامن** بينها.
- **اتصال** فيما بينها.

3.3.1 الجدولة الغير متوقعة للعمليات

- قد يقوم **المجدول** (في نظام التشغيل) بجدولة أي عملية في أي وقت ولأي سبب، هذه الجدولة ستوقف العملية التي يتم تنفيذها حاليا في المعالج، ويدخل عملية أخرى مكانها.
- هذا يعني أن البرنامج **قد يتوقف** في أي وقت (كنتيجة لإعادة الجدولة)، ثم فيما بعد قد يواصل من آخر نقطة توقف فيها **بالتالي قد تتسبب هذه الجدولة في مشكلة إذا كانت العملية المجدولة ضمن عمليات متعاونة**.

3.4 التنافس (competition)

- قد تكون **للعمليات المتعاونة بيانات مشتركة** تستطيع الوصول إليها. إذا لم يكن هنالك **تحكم في طريقة الوصول لهذه البيانات** ستكون النتيجة وصول غير منظم لها أو تنازع حولها وبالتالي نتوقع نتائج غير صحيحة. هذا الوصول غير المنظم للبيانات المشتركة يسمى **حالة السباق (Race condition)** حيث تتسابق العمليات في تغيير قيمة البيانات المشتركة.
- أيضا قد تكون **العمليات مستقلة** لكنها تتصل مع بعضها لاستخدام **موارد مشتركة**، مثلا افترض أن لدينا عمليتان، كل عملية تريد طباعة ملف على الطابعة (المشتركة)، لابد من وجود تنسيق ليتحدد من يستخدم الطابعة أولا، وعلى العملية الثانية الانتظار حتى تفرغ الأولى من الطباعة. هذا يسمى **تنافس العمليات (process competition)**

مشاكل التنافس

فيما يلي نسرد بعض الأمثلة التي تحدث فيها **مشاكل نتيجة للإيقاف غير المتوقع** لبعض العمليات بواسطة **المجدول**.

- **مشكلة تعديل ملف على خادم الملفات (file server) :** افترض أن لدينا شبكة حواسيب حيث يتم تخزين كل ملفات المستخدمين في **خادم الملفات**. إذا كان هنالك مستخدمين يريدان **تعديل ملف مشترك بينهما في نفس الوقت**، سيقوم كل مستخدم بفتح الملف في جهازه (نسخة من الملف)، ثم يقوم كل منهما بتعديل نسخته، **المشكلة** هي أنه عندما يقوم المستخدم الثاني بحفظ نسخة ملفه في الخادم سيكتب فوق تعديلات المستخدم الذي حفظ ملفه أولا (overwrite) هذه المشكلة ينتج عنها **مخرجات خاطئة**.

الصحيح هو أنه عندما تعمل عمليتان في وقت واحد يجب أن تكون النتيجة متشابهة بغض النظر عن أي عملية انتهت أول. **ولكن عندما يؤثر ترتيب تنفيذ العمليات في النتيجة يسمى هذا حالة السباق (Race condition) لأن ذلك يمثل سباق بين العمليات لنعرف من ينتهي أولا.**

- **مشكلة نظام الحجز الموزع :** إذا كان هنالك نظام حجز على خطوط طيران معينة، يعمل على عدة فروع. وكان هنالك مقعد واحد فارغ في رحلة ما، وجاء عميل لحجز هذا المقعد في **الفرع أ**، وفي نفس الوقت كان هنالك عميل **بالفرع ب** ، يريد حجز نفس المقعد، وقام الموظفين في الفرعين بعملية طلب حجز للمقعد في نفس الوقت،
- **ماذا يحدث ؟**

العملية الأولى في **الفرع أ** قامت بالتأكد من وجود مقعد فارغ، وهذا ما فعلته أيضا العملية التي نفذت في **الفرع ب**، فالعمليتان قامتا باختبار وجود مقعد فارغ، ثم أجاب النظام للعمليتين **"بنعم يوجد مقعد واحد فارغ"**، في هذه الإثناء قام المجدول (بنظام التشغيل) بتوقيف إحدى العمليتين لسبب ما، فقامت العملية الأخرى بحجز المقعد، ثم بعد فك توقيف العملية الأولى ستكمل عملها وتنفذ الأمر الذي يلي اختبار وجود مقعد فارغ (فهي اختبرت المقعد ووجدته فارغا قبل توقيفها). فتقوم العملية بحجز المقعد الذي حجز من قبل، **وبهذا يتم حجز المقعد مرتين.**

- **مشكلة الحساب المشترك :**

- إذا كان هنالك حساب مفتوح ببنك ما، وكان هذا الحساب مشترك بين عميلين، فقد يتفق وأن يطلب العميلين سحب مبلغ من المال من الحساب المشترك في وقت واحد من فرعين مختلفين. هنا ستنفذ عمليتين على الحساب المشترك، حيث ستقوم العمليتين باختبار الرصيد (نفترض أنه كان **1500** دولار)، ثم إذا أوقفت إحدى العمليتين بواسطة المجدول لسبب ما، ونفذت العملية الأخرى سحب المبلغ المطلوب (مثلا **100** دولار)، سيكون الرصيد الآن **1400** دولار. بعد أن يتم فك حجز العملية الثانية ستواصل من بعد آخر أمر كانت قد نفذته، وهو اختبار الرصيد (**1500** دولار)، وستنفذ السحب وتخصم المبلغ من الرصيد (مثلا إذا كان المبلغ المراد سحبه هو **200** دولار ، فسيكون الرصيد المتبقي هو(**1300** دولار). وتعتبر هذه العملية خطأ.

النتيجة الخاطئة التي وصلنا إليها كان **سببها الوصول غير المنظم لقاعدة البيانات وإجراء تعديل** على الحساب المشترك بصورة غير منسقة (غير تزامني) مما نتج عنه خطأ يسمى **حالة السباق**. حل مشكلة مثل هذه يتم بالتأكد من أن عملية واحدة فقط في لحظة معينة تقوم بتعديل البيانات المشتركة.

3.5 حالة السباق (race condition) والمنطقة الحرجة (critical section)

المشاكل التي سردناها مسبقا معظمها تعاني من **حالة السباق**، ولنوضح حالة السباق وطريقة حلها سنأخذ المثال التالي:

إذا كان لدينا متغير X، مشترك بين عمليتين، وقامت كل عملية بالآتي:

- قراءة قيمة المتغير X
- زيادة قيمة المتغير بواحد
- حفظ قيمة المتغير الجديدة.

```
Read (x)
x=x+1
Write(x)
```

المشكلة ستظهر عندما تقوم **أحد العمليات بقراءة قيمة المتغير**، ثم تقوم **العملية الثانية بقراءة قيمة المتغير** قبل أن تقوم العملية الأولى بحفظ القيمة الجديدة في المتغير.

فستكون النتيجة النهائية بعد تنفيذ العمليتان أن المتغير سيزيد بواحد بدلا من 2 **والحل هو أن نمنع أي عملية أخرى من الوصول للمتغير X إذا كانت هنالك عملية تستخدم هذا المتغير**.

المنطقة التي تتعامل مع **المتغير المشترك** في العملية نسميها **المنطقة الحرجة (critical section)** ستكون منطقة المتغير المشتركة هي المنطقة الحرجة، **ولحل مشكلة حالة السباق يجب أن نتأكد من أن هنالك عملية واحدة داخل المنطقة الحرجة.** ولا يسمح للعملية الثانية بالدخول إلى المنطقة الحرجة إلا بعد خروج العملية الأولى منها.

3.6 مشاكل التحكم بالعملية (Process Control Problems)

1. **الاقصاء المتبادل (Mutually exclusion) :** ضمان عدم تواجد اكثر من عملية في الجزء الحرج (**نتناوله لاحقا**).
2. **التزامن (Synchronization) :**ضمان عدم تجاوز العملية نقطة محددة بدون اشارة تسمح لها بذلك على ان تكون الاشارة خارجية.
3. **الايصاد او الاختناق (Deadlock)** : يقال لعملتين انهما في حالة اختناق اذا لم تسطع أيا منها الاستمرار حتى تستطيع الاخرى الاستمرار، ويقال ان النظام في حالة اختناق اذا كان كل العمليات في حالة اختناق.
4. **الاتصالات (Communication)** : وهي الطرق التي تمكن العمليات من الاتصال ببعضها البعض بما في ذلك طرق **الاقصاء المتبادل و التزامن**.

3.7 مفهوم الاقصاء/المنع المتبادل (Mutually exclusion)

الاقصاء المتبادل : هو ضمان **عدم استخدام الجزء الحرج الا من قبل عملية واحدة** في اي وقت ولنفس المورد. بمعنى انه **شرط معين** يضمن عدم استخدام مورد ما الا من قبل عملية واحدة.

من المعلوم ان الموارد قد تكون **مادية (Physical)** مثل المعالج, لوحة المفاتيح,...او **معنوية** – جزء من الذاكرة (متغير, برنامج،..), ملف,....

- الموارد التي **يمكن اعادة تخصيصها** تسمى **Preemptable**.
- الموارد التي **لا يمكن اعادة تخصيصها** الا اذا انتهت العملية التي تستخدمها من استخدامها، تسمى مثل هذه الموارد بالـ **nonpreemptable**.

- لكي تتعامل العمليات (**Processes**) مع بعضها البعض فلابد من وجود بعض **التعليمات** التي **تستخدم مورد ما جزئيا او كليا**, تعرف هذه التعليمات **بالجزء الحرج Critical section ويعرف كالاتي:**

تعريف الجزء (او الاجزاء) الحرج (Critical sections) : هو مجموعة من التعليمات التي نتيجة تنفيذها يؤدي الى نتائج غير متوقعة اذا تم استخدامها من قبل عمليات متوازية.

مثل هذه الاجزاء (**Critical-sections**) تستوجب **حماية خاصة**، وهي تستوجب هذه الحماية لكونها بدون هذه الحماية فإن استخدامها يؤدي الى نتائج غير متوقعة, وذلك اذا كان المتغير او المورد (بصفة عامة) يتعرض الى عملية تغيير من قبل العمليات المتوازية (**Parallel processes**)

3.7.1 المتطلبات الاساسية للإقصاء المتبادل (Mutually exclusion) وهي تتمثل في :

1. فرض الاقصاء المتبادل نتيجة التشارك في استخدام الموارد (جزء من الذاكرة, طابعة,..) **لابد من ضمان عدم تواجد اكثر من عملية** (**Process**) في الجزء الحرج (**Critical section**)
2. عندما تكون العملية خارج الجزء الحرج يجب **الا تعيق استمرار اي عملية اخرى**.
3. لا يجب **تأخير العملية** التي تطلب الجزء الحرج لفترة غير محددة، حتى لا يسمح بحدوث الايصاد (**Deadlock**) او الحرمان (**Starvation**)
4. بقاء العملية في الجزء الحرج يجب ان يكون **لزمن محدود**.

- **ضمان تحقيق الاقصاء المتبادل**
  - **خوارزمية ديكر Dekkers Algorithm** (الشكل 3.2): وهي استخدام **متغير كمؤشر (flag)** لاختبار ما اذا كان المؤشر **متاح (flag=0)** او **غير متاح (flag=1)**

```
begin
    repeat
    read flag;
    until  flag=0
    set flag to 1;  // the resource not available
    <critical section>
    set flag to 0;   // free the resource
end
```

الشكل 3.2 : خوارزمية ديكر

الاختبار المتتالي **للمؤشر (flag)** يتحقق عندما تصبح قيمة **المؤشر 0**, بعد ذلك يتم تغيير قيمة المؤشر الى **القيمة 1** لضمان عدم دخول اي عملية اخرى (flag=1), ثم تدخل العملية في الجزء الحرج. مثل هذه الحلقة (loop) تسمى **الانتظار المشغول (busy-waiting).**

يوجد بهذه الطريقة عيوب صعبة الاكتشاف وهي موضحة في السيناريو الاتي:

نفرض انه لدينا عمليتين P_1، P_2

1. العملية P_1 تختبر قيمة المؤشر فتجدها **0** وبالتالي المورد متاح فتنتقل العملية من الحلقة الى تنفيذ تعليمة تغيير قيمة المؤشر وقبل تنفيذ التغيير تحدث مقاطعة فيتم انتقال التحكم للعملية P_2 التي تريد المورد.
2. تختبر العملية P_2 المؤشر فتجده **0**, تبعا لذلك يتم تغيير قيمة المؤشر الى القيمة **1** وتدخل العملية الى الجزء الحرج, فاذا حدثت مقاطعة قبل ان تكمل العملية P_2 الخروج من الجزء الحرج واعدنا جدولة العملية P_1.
3. العملية P_1 الان تستمر من حيث توقفت باستكمال تغيير قيمة المؤشر الى القيمة **1** وتدخل الجزء الحرج.

الان العمليتان **P1, P2** داخل الجزء الحرج لكل منهما, مما يؤدي الى عدم تحقيق **الاقصاء التبادلي**.

- **ما هو السبب/الاسباب التي تؤدي لمثل هذه الحالة؟.**

لتفادي مثل هذا العيب تم اتباع الطرق الاتية كحلول:

1. **ابطال كل المقاطعات (turn off all interrupts) :** في الانظمة احادية المعالج لا يوجد تزامن بالمعنى الحرفي في تنفيذ العمليات ولكن يمكنها التبادل في التنفيذ (Interleaving execution)، حيث تستمر العمليات في التنفيذ الى ان تستدعي خدمة من نظام التشغيل (System call) او تصدر مقاطعة.

لضمان **الاقصاء المتبادل** يكفي **حماية العملية من أي مقاطعة** وذلك بتمكين نظام التشغيل (OS- Kernel) من القدرة على ابطال او تفعيل المقاطعة.

```
while (true) {

 /* disable interrupts */; ايقاف المقاطعات
 /* critical section */; دخول الجزء الحرج
 /* enable interrupts */; تنشيط المقاطعات
 /* remainder */;
 }
```

الشكل 3.3 : ايقاف المقاطعات

من عيوب هذه الطريقة

- **كفاءة التنفيذ تتناقص** : اعطاء مثل هذه القدرات للمستخدم يؤدي الى قصور في كفاءة نظام التشغيل، فاذا اوقف مستخدم ما المقاطعات ونسي تنشيطها فان نظام التشغيل سيتوقف.
- هذه الطريقة لا تؤثر على بقية المعالجات (في حالة المعالجات المتعددة).
- الاتجاه العام حاليا هو انتاج انظمة متعددة المعالجات الامر الذي اضعف من استخدام مثل هذه الطريقة.

2. **الحل الثاني لإبطال مفعول المقاطعة بطريقة غير مباشرة** كما على سبيل المثال في **MC68000** اعتمدت **طريقة تغيير الاسبقية الى الدرجة الاعلى وهي المستوى السابع** في هذه الحالة بتعليمة واحدة **(ORI.W #0700, SR)** الا ان هذه التعليمة لا يمكن تطبيقها على برامج المستخدم ولكن تطبق على برامج النظام لأنها تعليمة امتياز **(privileged instruction)** لأنها تقوم بتغيير قيم المسجلات **(SR)** ولذلك تعتبر حلا جزئيا.

3. **الحل الثالث** هو جعل عملية **القراءة**, **والتعديل**, **والكتابة للمؤشر** تتم دفعة واحدة غير قابلة للقسمة (Indivisible) يسمى هذا الحل "**اختبر واعطي القيمة**"

(**Test and Set -TAS**) وهو مستخدم في **MC68000**

الحل الثاني قد يؤدي الى حدوث ما يعرف بالاختناق (Deadlock) وفقا للسيناريو الاتي:

- **العملية P_L** ذات الاسبقية **الادنى** موجودة في **الجزء الحرج**. بمعنى انها حجزت المورد.
- **المرسل (Dispatcher)** يجدول العملية **P_H** ذات الاسبقية **الاعلى**.
- يستمر المرسل في اعادة الجدولة دون جدوى لان العملية **P_H** يتم اعادة جدولتها باستمرار لأنها الاعلى اسبقية ومؤشر المورد في **القيمة 1** اي ان المورد غير متاح. وبالتالي تسمى مثل هذه الحالة **بالإيصاد او الاختناق (Deadlock)**

هناك ايضا بعض الطرق الأخرى والتي سنتناولها لاحقا مثل **السيمافور (Semaphore)**، **والمراقب (Monitor)** ، **وتمرير الرسالة (Message passing)** .

3.8 التزامن (Synchronization)

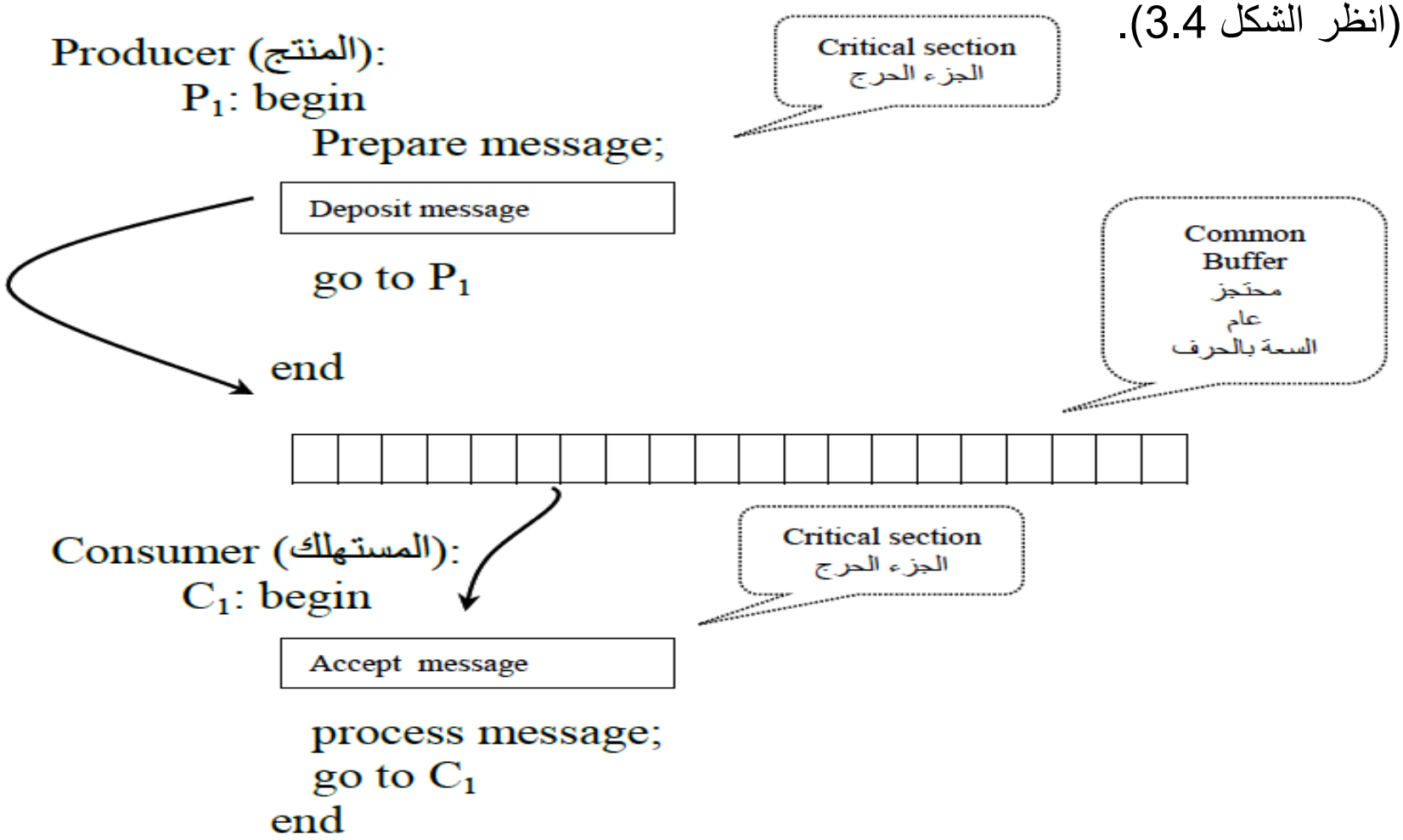
إذا احتاجت العمليات المتوازية أن تتشارك في بعض البيانات أو أن تؤدي عملا مشتركا، فلابد لها من أن تتعاون مع بعضها البعض. هذا التعاون يتطلب **اتصال** بين العمليات وعملية **تزامن** (**Synchronization**).

- **الاتصال** بين العمليات يتم عن طريق **متغيرات مشتركة** (**Shared variables**) أو **بتبادل الرسائل** (**Message passing**)
- **والتزامن** نحتاجه ليتم الاتصال في الوقت المناسب، ويتحقق بوضع نقاط تزامن للعمليات بحيث إذا سبقت عملية بقية العمليات ووصلت الى نقطة تزامن، عليها أن تنتظر بقية العمليات الأخرى لتصل هذه النقطة، وإلا سيكون لدينا نتائج غير متوقعة.

اذا **التزامن** هو ضمان عدم استمرار العملية بعد نقطة معينة الا عبر اشارة خارجية. هذه الاشارة قد تكون من عملية اخرى او من خارج النظام.

مثال:

- **مشـكلة المنـتج والمسـتهلك (Producer-Consumer)** : أحيانـا تسـمى مشـكلة المحتجـز المحـدود Bounded-buffer هنا يكون لدينا **عمليتان** تستخدمان محتجز (Buffer)مشترك، **العملية الأولى تضع به بيانات بينما تأخذ العملية الأخرى البيانات منه**. وعلى العمليتين التنسيق فيما بينهما بحيث لا تحاول العملية الأولى وضع بيانات في الخازن إذا كان ممتلئ ولا تحاول العملية الثانية أخذ بيانات من الخازن إذا كان فارغا (انظر الشكل 3.4).



الشكل 3.4 : مشكلة المنتج والمستهلك

حل مشكلة المنتج والمستهلك :

- يمكن حل هذه المشكلة بإجبار **عملية المنتج** على التوقف عن إحضار البيانات للخازن إذا كان ممتلئا (**sleep**) وعندما يفرغ الخازن تقوم **عملية المستهلك** بإيقاظ المنتج ليبدأ في تعبئة الخازن .
- بنفس الطريقة **تتوقف عملية المستهلك (sleep)** إذا كان الخازن فارغا، وعندما يحضر المنتج بيانات للخازن، يقوم بإيقاظها لتبدأ في أخذ البيانات من الخازن. يمكن تطبيق هذا الحل باستخدام **السيمافور (semaphore)** الذي يعتمد على الاتصال بين العمليتين (**inter-process communication**).

3.9 السيمافور (Semaphore)

تعتمد العمليات المتعاونة على **اشارات بسيطة**، وذلك بإيقاف عملية ما في نقطة معينة حتى تستقبل اشارة معينة وذلك باستخدام متغير يسمى **بالسيمافور** . والفكرة الاساسية هي بدلا من الاختبار المستمر **(Busy waiting)** لحالة المؤشر ، فان **السيمافور (S)** اذا وجد المورد محجوز فانه يرسل العملية الى **حالة الانتظار** الخاصة بالسيمافور (**sleep**)، ويتم ايقاظها بإشارة خارجية عندما يصبح المورد **متاح (wakeup)**

ويستخدم السيمافور ثلاث عمليات اساسية – عملية التهيئة **initialization**، **Signal**، **Wait**.

- يخزن السيمافور في مكان تابع لنظام التشغيل ويمكن الوصول اليه عبر عمليتين اساسيتين هما **Signal, Wait**.

- ينفذ نظام التشغيل هاتان العمليتين بدون انقطاع **(Indivisible, atomic)** لضمان اتمام عملية تنفيذهم لتفادي التداخل بين العمليات. ويمكن فعل ذلك عن طريق ايقاف المقاطعات او استخدام تعليمة **TAS: Test And Set**

عمليات السيمافور يمكن تعريفهم كالاتي (الشكل 3.5):

```
struct semaphore {
     int count;
    queueType queue;
};
void semWait (semaphore s)
{
    s.count--;
    if (s.count < 0) {
    /* place this process in s.queue  ;/* وضع العملية في رتل الانتظار
    /* block this process */;
    }
}
void semSignal (semaphore s)
{
    s.count++;
    if (s.count<= 0) {
    /* remove a process P from s.queue   */;
    /* place process P on ready list ;/* وضع العملية في رتل الجاهزية
   }
}
```

الشكل 3.5 : عمليات السيمافور

يوجد نوعان من السيمافور- العام والثنائي

- **السيمافور العام** : تعريف عمليات (انظر الشكل 3.5)

1. يتم تهيئة السيمافور بقيمة موجبة (0-ن).
2. اشارة **semWait(s)** تخفيض قيمة السيمافور **بمقدار 1**. اذا اصبحت القيمة **اقل من الصفر**, فان العملية الحالية يتم وضعها في قائمة **الانتظار** الخاصة بالسيمافور, وخلاف ذلك تستمر العملية في تنفيذ الجزء الحرج.
3. اشارة **semSignal(s)** تزيد قيمة السيمافور **بمقدار 1**. اذا كانت النتيجة **اقل او تساوي الصفر** فمعنى ذلك ان العملية الحالية انتهت من الجزء الحرج وان هناك عمليات اخرى في قائمة الانتظار وعليه يتم **نقل عملية ما من قائمة الانتظار الى قائمة الجاهزية**.

3.9.1 كيفية استخدام السيمافور لتحقيق الاقصاء المتبادل

لنفرض ان لدينا السيمافور **s (s=1)** وأن العملية **P1** هي اول عملية يتم تنفيذها وقبل ان تدخل الجزء الحرج, تَنفِذ عملية **semWait(s)** التي تتطلب تخفيض قيمة **s بمقدار 1** بحيث تصبح **قيمتها 0**، وبالتالي لا يتحقق الشرط **s<0** فتدخل العملية للجزء الحرج, فاذا كان هناك عملية اخرى منافسة ولتكن **P2** قد تم جدولتها, فإنها تنفذ عملية **semWait(s)** وتخفض قيمة **(s=0)** فتصبح **s=-1** فيتحقق الشرط فتوضع العملية **P2** في قائمة الانتظار.

عند اعادة جدولة **P1** فإنها تكمل الجزء الخاص بها وترسل اشارة **semSignal(s)** فتزيد قيمة السيمافور فتصبح قيمة **s=0** هذا يجعل العملية **P2** تنتقل الى قائمة الجاهزية, لإعادة جدولتها فيما بعد وبالتالي تستطيع الدخول الى الجزء الحرج. وعند انتهائها تكون قائمة الانتظار فارغة.

P_1
semWait(s);
<critical section 1>
semSignal(s);
more statement;

P_2
semWait(s);
<critical section 2>
semSignal(s);
more statement;

س3/ كيف يتغلب السيمافور على حلقة الانتظار المشغول؟

- **السيمافور الثنائي : تعريف عمليات**

1. يتم تهيئة السيمافور الثنائي بقيمة 0 او 1.
2. اشارة (**semwaitB(s** تختبر قيمة السيمافور, اذا كانت **القيمة = 0**, فان العملية الحالية يتم وضعها في قائمة الانتظار, وخلاف ذلك (**قيمتها 1**) تستمر العملية في الدخول الجزء الحرج.
3. اشارة (**semsignalB(s** تختبر وجود عمليات في قائمة الانتظار (**قيمة السيمافور = 0**) , اذا كانت كذلك فان العملية الموجودة بقائمة الانتظار تنتقل الى قائمة الجاهزية. وخلاف ذلك تصبح قيمة **السيمافور = 1**.

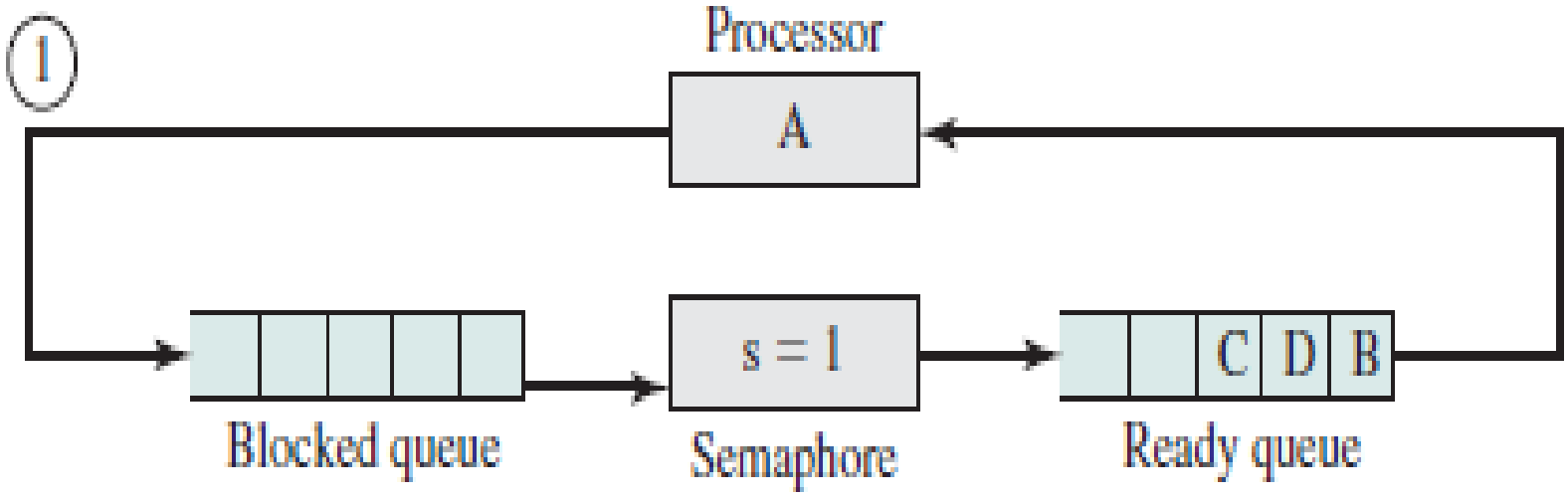
```
struct binary_semaphore {
        enum {zero, one} value;
        queueType queue;
};
void semWaitB(binary_semaphore s)
{
        if (s.value == one)
                s.value = zero;
        else {
                /* place this process in s.queue */;
                /* block this process */;
        }
}
void semSignalB(semaphore s)
{
        if (s.queue is empty())
                s.value = one;
        else {
                /* remove a process P from s.queue */;
                /* place process P on ready list */;
        }
}
```

ان استخدام **السيمافور الثنائي او العام** يؤدي لنفس النتيجة.

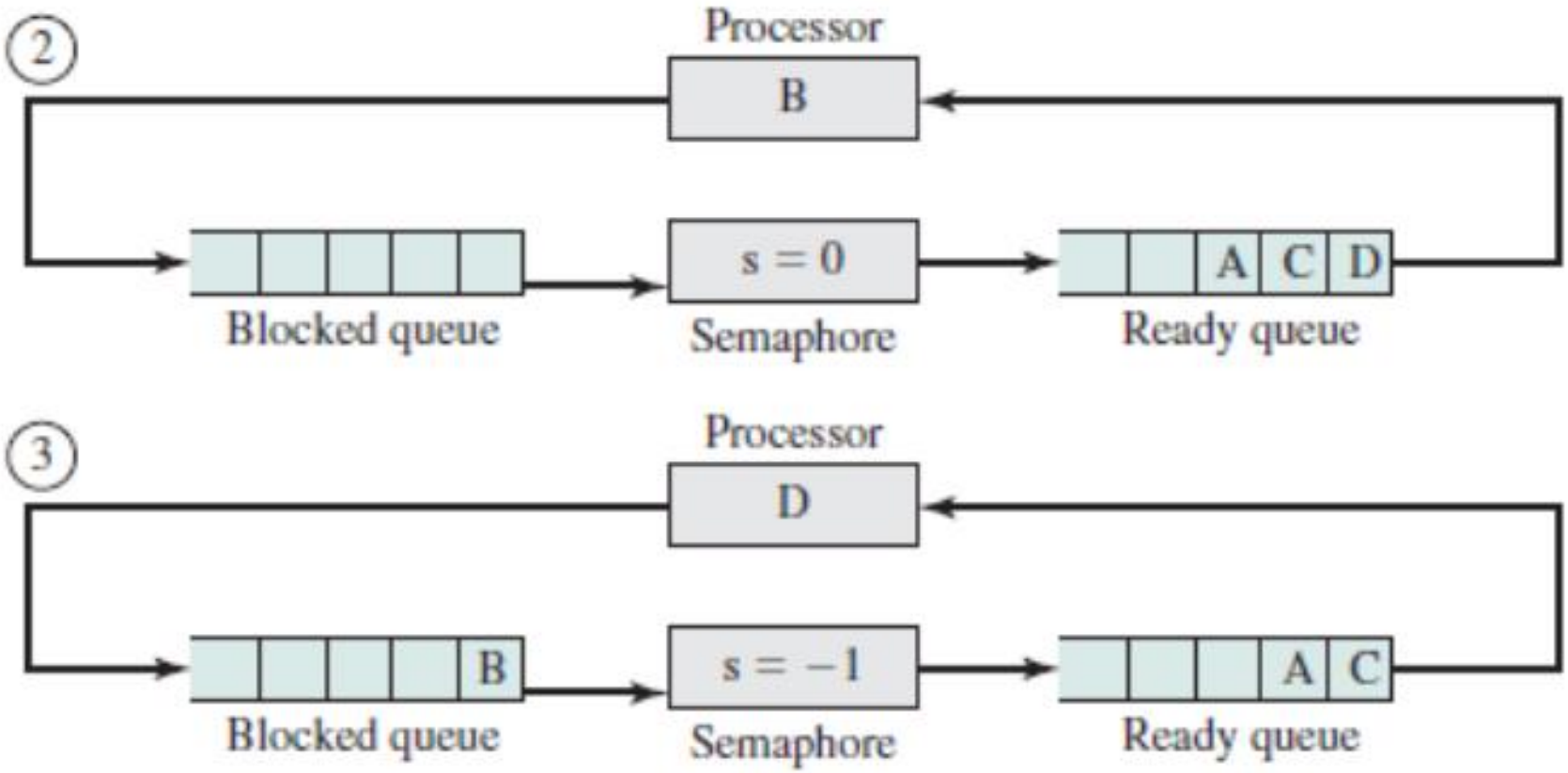
لعله من المفيد معرفة كيف يتم اختيار العمليات من طابور السيمافور فمن العادة ان يتم اختيارهم حسب الوصول (**FCFS**) ويعرف السيمافور في هذه الحالة **بالسيمافور القوي** (Strong semaphore) فاذا لم يتم تحديد طريقة الاختيار فانه يعرف **بالسيمافور الضعيف** (Weak semaphore)

في المثال التالي نبين استخدام **السيمافور القوي (Strong semaphore)**، حيث لدينا اربعة عمليات (**A, B, C, D**) موجودة بقائمة الجاهزية (Ready-list) وتتأثر العمليات (**A, B, C**) بنتيجة العملية (**D**) و لنفرض ان قيمة السيمافور **s=1** يمكن تتبع تنفيذ العمليات وفقا للخطوات التالية:

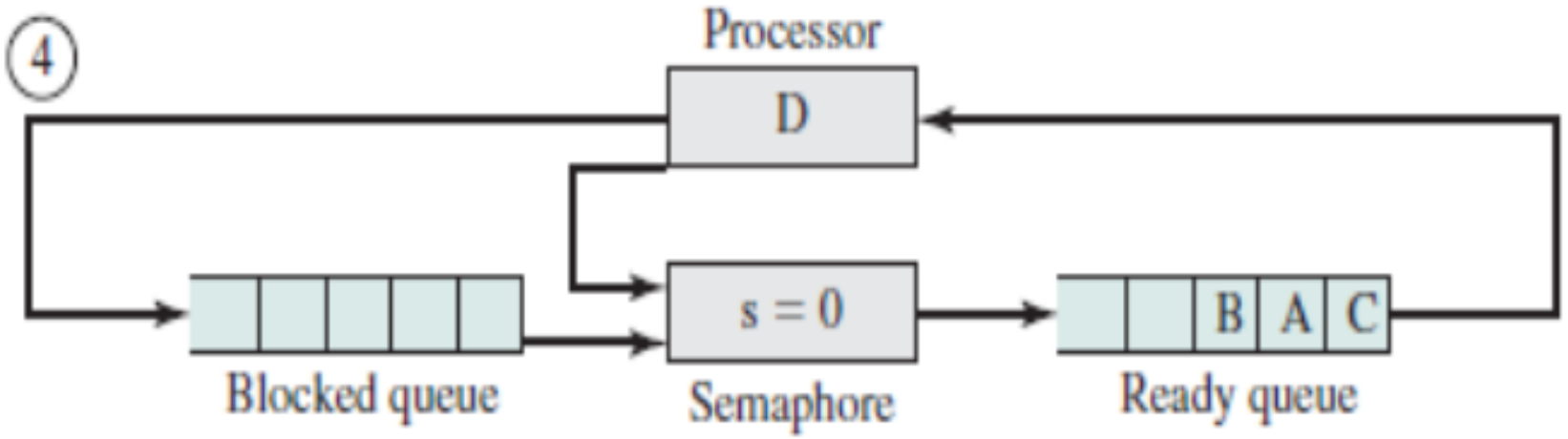
1. **A** قيد التنفيذ وقبل الدخول للجزء الحرج تختبر قيمة **s =1** باستخدام **semWait()** التي تخفض قيمة السيمافور **s=0** فتختبر الشرط وتدخل الجزء الحرج وذلك لعدم تحقق الشرط. لنفرض انه لسبب ما انتقلت الى قائمة الجاهزية.



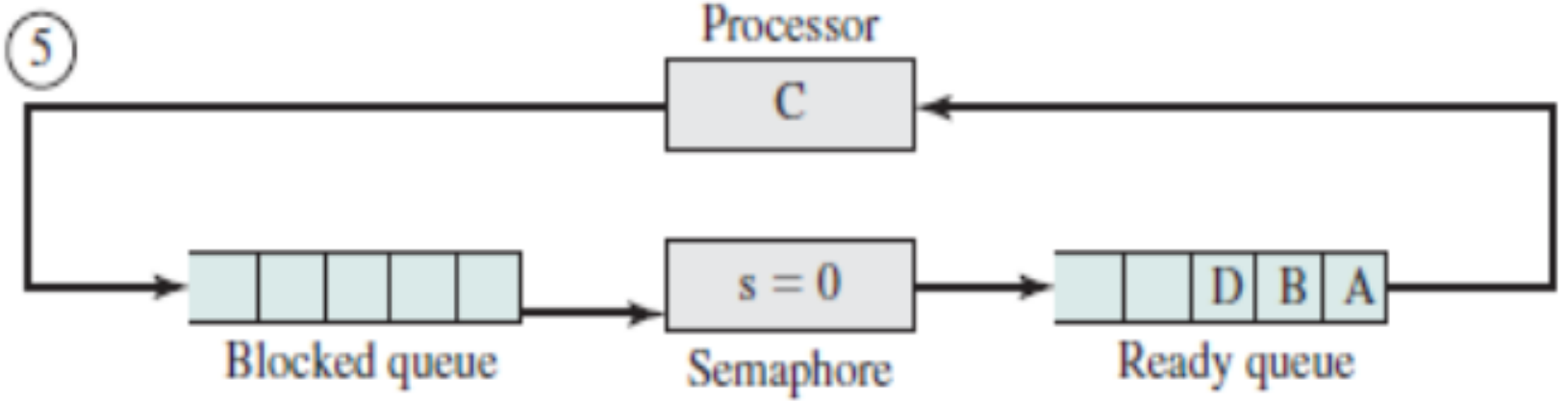
2. تدخل العملية **B** حالة التنفيذ لأنها العملية الاولى على قائمة الجاهزية (FCFS) حيث قيمة السيمافور **s=0**، حيث يتم تخفيضها (**semWait**) الى القيمة **s=-1**، فتدخل العملية **B** الى قائمة الانتظار.
3. تدخل العملية **D** لحالة التنفيذ، حيث قيمة السيمافور **s=-1**، لنفرض ان **D** ارسلت **semSignal** فالنتيجة تكون **s=0**



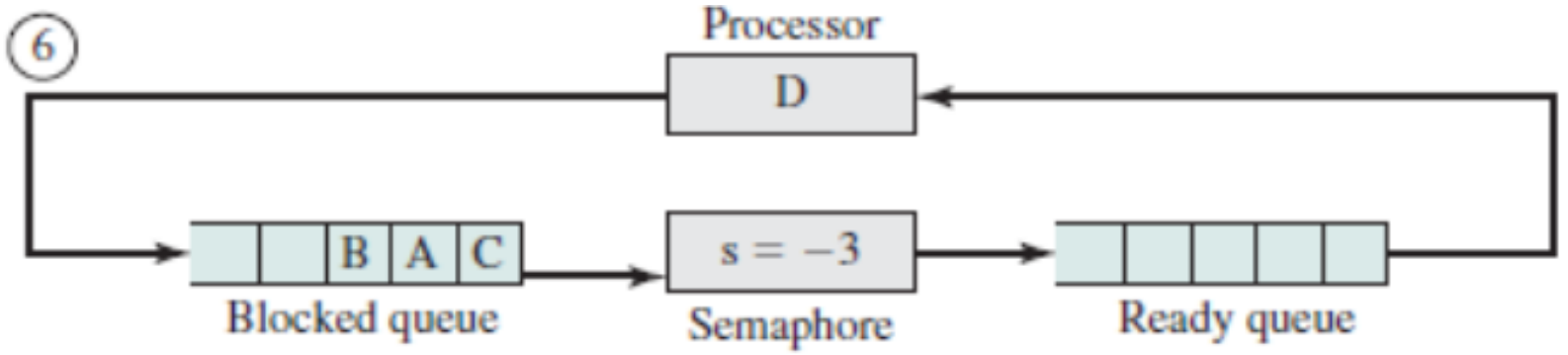
4. ويتم نقل العملية **B** من قائمة الانتظار الى قائمة الجاهزية (موضحة في الخطوة 4).



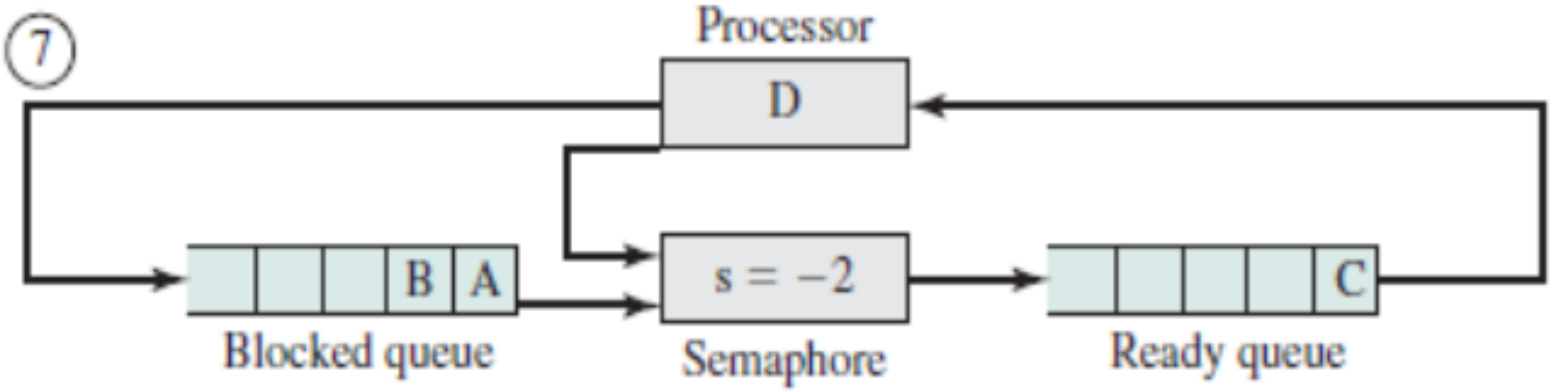
5. تعود العملية **D** الى قائمة الجاهزية الامر الذي يؤدي الى دخول العملية **C** حالة التنفيذ.



6. وبالتالي يتم تخفيض قيمة السيمافور **s=-1** وتدخل العملية **C** الى قائمة الانتظار (Blocked queue) وبالمثل للعمليتين **B ,A** ام العملية **D** فتدخل حالة التنفيذ.



7. لنفرض ان العملية **D** تكرر تنفيذ الامر **semSignal** وبذلك تعمل على اعادة العمليات **C, B, A** الى قائمة الجاهزية.



تطبيق السيمافور في حالة مسألة منتج واحد- مستهلك واحد

```
Semaphore ItemAvailable=0  /* عدد العناصر المتاحة للسحب */
semaphore SlotFree = BUFFER_SIZE  /* حجم المحتجز */
procedure producer() {
while (true) {
        item = produceItem()
        semWait(SlotFree)
        putItemIntoBuffer(item)
        NextIn++
        semSignal(ItemAvailable)
}

}
procedure consumer() {
while (true) {
        semWait (ItemAvailable)
        item = removeItemFromBuffer()
        NextOut++
        semSignal (SlotFree)
        consumeItem(item)
}
```

التزامن Synchronization

- عندما يكون **الخازن فارغ** ، فان **المستهلك** سوف يوضع في **قائمة الانتظار** الخاصة بالسيمافور **ItemAvailable**، وقيمته 0. ولن يخرج من القائمة الا عندما يقوم **المنتج** بإعطاء الاشارة **semSignal** الخاصة بالسيمافور **ItemAvailable** و التي تحدث كنتيجة لعملية لإيداع عنصر في الخازن.
- وعندما يكون **الخازن ممتلئ**، فان **المنتج** سوف يوضع في **قائمة الانتظار** التابعة للسيمافور **SlotFree**، والذي قيمته 0 في هذه الحالة. وتبقى عملية **المنتج** في قائمة الانتظار () الى ان يقوم **المستهلك** بسحب عنصر من الخازن فيصبح مكان العنصر فارغ، ثم يقوم بإعطاء اشارة **semSignal (SlotFree)** والتي تسمح للمنتج بالانتقال الى حالة الجاهزية لاخذ دوره في التنفيذ من جديد.

الاقصاء المتبادل Mutually exclusion

في حالة **المنتج الواحد والمستهلك الواحد** لا نحتاج **للإقصاء المتبادل** ولتوضيح ذلك :

- الطريقة الوحيدة للوصول لنفس العنصر هو عندما يكون **NextIn=NextOut**.
- تحدث هذه الحالة بطريقتين – عندما يكون الخازن **فارغ** و عندما يكون **ممتلئ**.
- ففي حالة **الخازن فارغ** فان **المستهلك** لن يدخل الى هناك **ItemAvailable=0**
- وفي حالة **الخازن ممتلئ** فلن يدخل **المنتج** هناك **SlotFree=0**
- وبالتالي تحقق **الاقصاء المتبادل**.

تعدد المنتجين و تعدد المستهلكين Multiple producers, multiple consumes

تعدد المنتجين والمستهلكين يؤدي لتواجد اكثر من منتج او مستهلك في الجزء الحرج لإنتاج عنصر لنفس الموقع الامر الذي يتطلب وضع سيمافور اخر لضمان الاقصاء المتبادل كما هو موضح في الشكل

```
Semaphore ItemAvailable=0  /* عدد العناصر المتاحة للسحب */
semaphore SlotFree = BUFFER_SIZE  /* حجم المحتجز */
procedure producer() {
while (true) {
        item = produceItem()
        semWait(SlotFree)
        semWait(Guard)
        putItemIntoBuffer(item)
        NextIn++
        semSignal(Guard)
        semSignal(ItemAvailable)
}

}
procedure consumer() {
while (true) {
        semWait (ItemAvailable)
        semWait(Guard)
        item = removeItemFromBuffer()
        NextOut++
        semSignal(Guard)
        semSignal (SlotFree)
        consumeItem(item)
}
```

عيوب السيمافور

- استخدامه ليس الزاميا بمعنى ان المبرمجين لا يستخدمونه (بقصد او بغير قصد) الامر الذي يخالف شرط الاقصاء المتبادل.
- الاستخدام الخاطئ يؤدي الى حدوث الاختناق.
- في حالة العملية ذات الاسبقية الادنى فانه يمكنها احداث الاقصاء المتبادل ومنع العمليات الاخرى مهما كانت اسبقيتهم.
- الإشارات **semWait, semSignal** قد تتوزع وتنتشر بداخل البرنامج وبالتالي ليسٌ من السهل مشاهدة التأثيرٌ الكل لهذه العملياٌت على السيمٌافور الذي يؤٌثرون فيه.

المونيتر Monitors

المونتير هو تركيبٌة من **تراكيبٌ لغات البرمجة** أي انه برنامج يحٌتوى مجموعة من (**الاجزاء** – **دوال** , **الإجراءات**) لها بيانات محليةٌ وهو ينٌاظر السيمٌافور (في قدرته بالتحكم باستعمال المورد) ولكن **المونتير** سهل التحكم.

المونتير طبق في لغات برمجة عديدٌة مثل البسكال المطورة, والجافا. وبالتالي المونتير تسمح للمبرمجينٌ بوضع **المونتير** للتحكم في اي جزء بالبرنامج

اهم خصائص المونيتر

- متغيرٌات البيأنات المحليةٌ يتٌم الوصول اليهٌا عبر المونتير ودواله وليسٌ عن طريقٌ الدوال الخارجيةٌ
- العمليةٌ تدخل المونتير عن طريقٌ استدعاء احد دواله او أحدى مكوناته.
- عمليةٌ واحدة فقط يتٌم تنفيذٌها بالمونتير في فترة زمنيةٌ محددة. ايٌ عمليةٌ أخرى تطلب المونتير يتٌم تجميدٌها حتى يصٌبح المونتير متوفرا
- اي لغة متطورة او نظام تشغيلٌ يمٌكنه تطبيقٌ المونتير ككيأٌن بصفات خاصة
- بفرض الترتيبٌ (النظام) عمليةٌ واحدة في زمن معينٌ, يمٌكن للمونيتر من توفيرٌ المنع المتبادل
- متغيرٌات البيانات بالمونتير يمٌكن استعمالها من قبل عمليةٌ واحدة فقط خلال فترة زمنيةٌ محددة.
- هيأٌكل البيانات المشتركة يمٌكن حمايتٌها بوضعها داخل المونتير.
- اذا كانت البيانات الموجودة بالمونيتر ترمز لمورد معينٌ, اذا المونيتر يقٌوم بتوفيرٌ الاقصاء المتبادل كوسيلٌة للوصول للمورد

تمارين غير محلولة

1. ما هو مفهوم التزامن ؟
2. ما المقصود بالتنافس ؟
3. ما الفرق بين العملية المستقلة والعملية المتعاونة ؟
4. لماذا تحتاج العمليات المتوازية التعاون فيما بينها ؟
5. ما المقصود بحالة السباق race condition ؟ وما هي مسبباتها ؟ وكيف يتم حلها ؟
6. عرف المنطقة او الجزء الحرجة/الحرج Critical section ؟
7. أذكر ثلاث من مشاكل التزامن الكلاسيكية ؟
8. تتصل العمليات المتعاونة فيما بينها بطريقتين، ما هما ؟