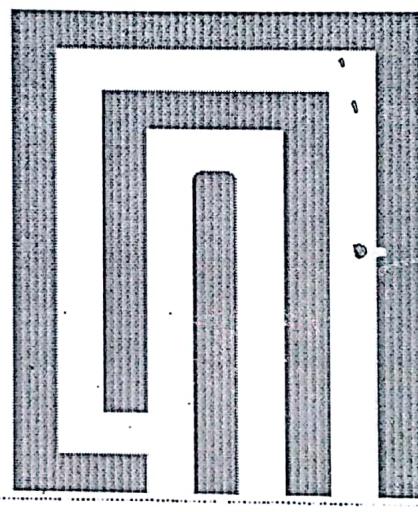


# كلية الهندسة المعلوماتية

## نظم تشغيل 1

المحاضرة الثامنة (تزامن الإجراءيات 1) عدد الصفحات: 15

السنة الثالثة



**GATE**

Greatness Achieves Through Excellence

الزراعة - مقابل باب السكن الجامعي - شارع مساوا

📞 ٠٤ ٢٤٩٦٢٣٧ ٠٩٦٠ ٠٦٦ ٠١٦



🌐 [www.facebook.com/groups/ITech.GATE](https://www.facebook.com/groups/ITech.GATE)

## مقدمة :

إن الإجرائيات المتسايرة concurrent (التي تنفذ بالتزامن ضمن إل CPU أي في نفس الوقت) ...  
الموجودة في النظام هي نوعين إما مستقلة independent أو متعاونة cooperating

### [1] الإجرائية المستقلة independent process

هي الإجرائية التي لا تؤثر can't affect ولا تتأثر not affected بتنفيذ الإجرائيات الأخرى.

كل إجرائية لا تشارك المعطيات مع إجرائيات أخرى تعتبر إجرائية مستقلة.

### [2] الإجرائية المتعاونة cooperating process

هي الإجرائية التي تؤثر can affect وتتأثر affected بتنفيذ الإجرائيات الأخرى.

كل إجرائية تشارك المعطيات مع إجرائيات أخرى تعتبر إجرائية متعاونة.

☞ ☺ الفكرة هنا هي وجود بيانات مشتركة shared data

إن الوصول المتزامن concurrent access للبيانات المشتركة

قد يسبب عدم اتساق البيانات data inconsistency

مثال للتوضيح : المتحوّل  $x$  مشترك بين إجرائيتين  $p_1$  و  $p_2$  وقيمتها يجب أن تكون 5 لكن بسبب

الوصول المتزامن لـ  $p_1$  و  $p_2$  على هذا المتحوّل المشترك أصبحت قيمته 4 عن طريق الخطأ وهذا يسمى

مشاكل في قيم البيانات تشير إليه بالعبارة "عدم اتساقية البيانات" (ارجع إلى فوائد المعطيات 1).

الحفاظ على البيانات متسقة consistency (كي تبقى الأمور تمام) يحتاج إلى آليات للتأكد على

التنفيذ المنظم orderly لتنفيذ الإجرائيات المتعاونة (لا يوجد مشكلة بالنسبة لـ الإجرائيات المستقلة).

إن Process تسمى هنا "إجرائية" ولكن في الامتحان تسمى "عملية"



من الآليات التي تسمح لنا بتنظيم تنفيذ الإجرائيات المتعاونة cooperating هي :

## ترامن الإجرائية : process synchronization

هي طريقة لتنسيق coordinate عمل الإجرائيات التي تستخدم بيانات مشتركة shared data

إن العمل في بيئة من الإجرائيات المتعددة المتعاونة multiple processes يحتاج إلى شيء يسمى بـ

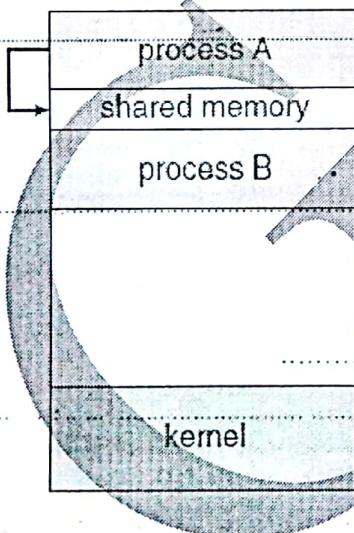
IPC وهي اختصار ل التواصل بين الإجرائيات :

**IPC = Inter-Process Communication**

IPC : هي طريقة تسمح للإجرائيات بتبادل البيانات بين بعضها البعض . exchange

إن إمكانيات ال IPC تتلخص في عمليتين :

- إرسال الرسائل send(message)
- استقبال الرسائل receive(message)



هناك ثوانيان من آليات تواصل الإجرائيات IPC :

### 1\_الذاكرة المشتركة shared memory

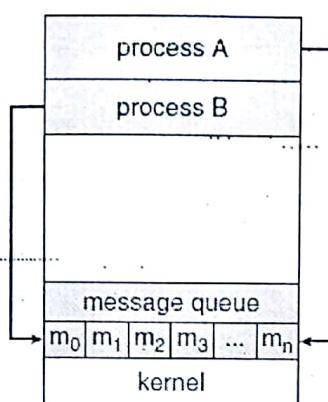
يتم تخصيص establish منطقة من الذاكرة وجعلها مشتركة بين الإجرائيات

المتعاونة ← تستطيع الإجرائيات تبادل المعلومات عن طريق

قراءة reading وكتابة writing كل البيانات إلى هذه المنطقة المشتركة .

بالتالي يقع العبء على المبرمج programmer كي ينظم العمل بين الإجرائيات

مع ملاحظة أن نظام التشغيل OS ليس له علاقة بهذا الموضوع أبداً



## ٢\_ تبادل الرسائل : message passing

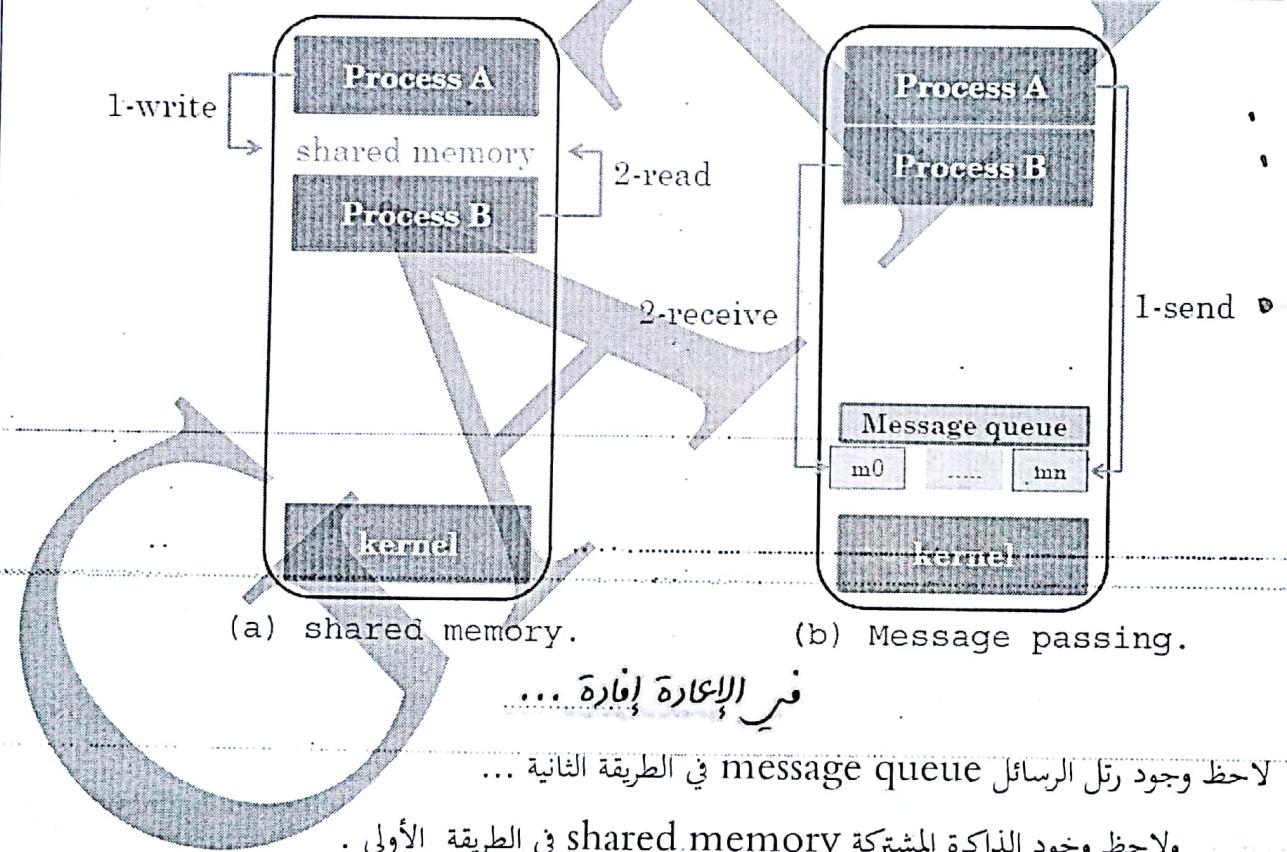
يتم عن طريق اتصال الإجرائيات فيما بينها عبر قناة اتصال

يتم فيها تبادل الرسائل exchange messages دون

الحاجة لوجود ذاكرة مشتركة ..

يقع العبء على نظام التشغيل لتأمين عملية الاتصال هذه

الشكل التالي يوضح الفرق بين الطريقتين :



فِرِّيْدَةِ إِعْلَانِهِ

لاحظ وجود رتل الرسائل message queue في الطريقة الثانية ...

ولاحظ وجود الذاكرة المشتركة shared memory في الطريقة الأولى .

سيكون حديثنا عن الذاكرة المشتركة shared memory وكيفية تنظيم عمل هذه الذاكرة

لكي تعمل الإجرائيات بشكل منظم دون حدوث أي مشاكل 😊

## نظم تشغيل المحاضرة الثامنة (ترامن الإجرائيات 1)



### المنتج - المستهلك producer-consumer

لدينا هنا إجرائيتين متعاونتين : cooperating

الأولى اسمها الإجرائية المنتجة producer والثانية اسمها الإجرائية المستهلكة consumer  
producer produces && consumer consumes

بحيث أن الأولى تقوم بإنتاج أشياء معينة ولتكن سكاكراً والثانية تقوم باستهلاك هذه الأشياء التي تنتجهما الأولى عن طريق أكل هذه السكاكرا ☺.



لكي تتم العملية السابقة (إنتاج-استهلاك) يجب أن يوجد مكان محدد يتم وضع (إنتاج) السكاكرا فيه ويتم تفريغ (استهلاك) السكاكرا منه يسمى هذا المكان بـ buffer (مخزن).

producer puts in buffer && consumer picks from buffer

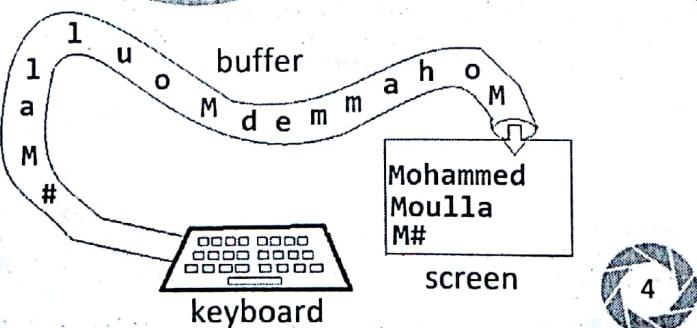
ملاحظة خارجية :

إن buffer ليس لها ترجمة دقيقة لكن يمكن اعتبارها بأكملها مخزن مؤقت ، مثلاً عندما تكتب على لوحة

المفاتيح keyboard فإن الأحرف التي تكتبهما يتم تخزينها في buffer لكي يتم استلامها فيما بعد من

قبل المتحكم في لوحة المفاتيح لأنه في لحظة ما قد يكون الحاسب مشغول بأمر آخر ولا يستطيع تلقي هذه

الأحرف مباشرة من لوحة المفاتيح وبالتالي لن تضيع الأحرف التي كتبتها وإنما سيحصل تأخير فقط



المخزن buffer عبارة عن  
مصفوفة احادية

• يوجد لدينا نوعين من مسائل المنتج-المستهلك :

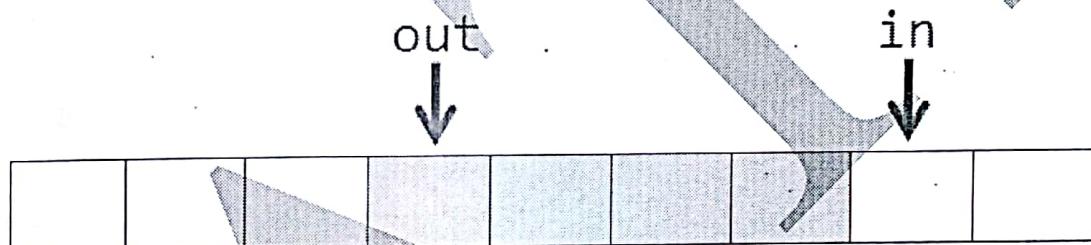
## 1\_ المنتج-المستهلك ذات المخزن اللاحدود unbounded-buffer

أي أنه لا يوجد حد لحجم المصفوفة أي أن  $n = \infty$  وبالتالي يستطيع المنتج توليد بند جديداً دائماً أي لا يوجد حدود لعدد البنود التي ستتتجها لكن !! يجب أن يتذكر المستهلك وجود البنود حتى يستهلاكها.

بشكل عام سنميز مؤشرين للقيام بعملية إنتاج - استهلاك هما :

المؤشر in  $\leftarrow$  يؤشر على المكان الذي يجب أن يتم وضع العنصر الذي تم إنتاجه (فارغ أصلاً).

المؤشر out  $\leftarrow$  يؤشر على المكان الحاوي للعنصر الذي يجب أن يتم استهلاكه (متلئ أصلاً).

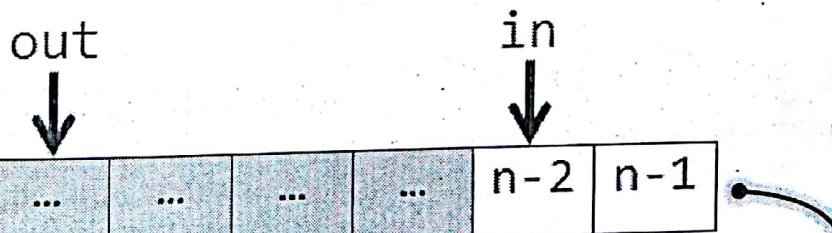


## 2\_ المنتج-المستهلك ذات المخزن المحدود bounded-buffer

أي أنه يوجد حد لحجم المصفوفة أي أن  $n = const$  وبالتالي يجب أن يتذكر المنتج في حال كان المخزن ممتلئاً حتى يتفرّغ ويجب أن يتذكر المستهلك في حال كان المخزن فارغاً حتى يمتلئ لكي يقوم بالاستهلاك.

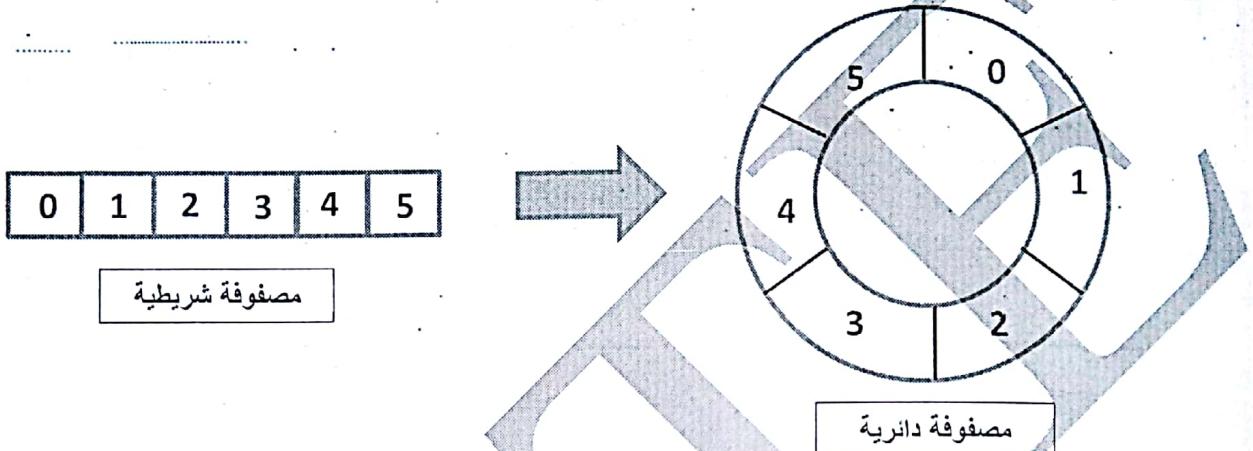
من أجل مصفوفة ب  $n$  عنصر تكون أدلة العناصر هي  $0, 1, 2, \dots, n-1$ .

عند الوصول إلى العنصر  $n-1$  سوف نعود إلى العنصر 0 بالشكل :



لتحقيق المصفوفة السابقة سنتعامل مع دليل المصفوفة **index** بطريقة مختلفة !!!

سنعتبر في هذه الحالة أن المصفوفة دائيرية **circular** ليس شريطية **string** بمعنى أنه عند الوصول إلى آخر عنصر فإن العنصر التالي هو العنصر الأول كما يلي :



ملاحظة : سوف نستخدم مفهوم باقي القسمة (mod) لكي نحصل على مصفوفة دائيرية حيث أنه لا يوجد شيء اسمه مصفوفة دائيرية وإنما مجرد خيال ، لكننا نحصل على مصفوفة دائيرية عن طريق التعليمية :

$i = i + 1$

بدلاً من

$i = (i + 1) \% n;$

الانتقال للعنصر التالي في  
المصفوفة الدائرية

حيث أن  $i$  تعتبر عن دليل العنصر و  $n$  تعبّر عن عدد العناصر وبالتالي إذا أردنا الوصول إلى العنصر التالي

نضيف للمتحول  $i$  واحد ثم نأخذ باقي القسمة على  $n$  لكي لا نخرج خارج المجال  $[0, n - 1]$ .

مثلاً لنكن  $n = 5$  وبالتالي سأخذ المتحول  $i$  القيم التالية بالترتيب :

$$i = 0 \Rightarrow i = (0 + 1) \% 5 = 1 \% 5 = 1$$

$$i = 1 \Rightarrow i = (1 + 1) \% 5 = 2 \% 5 = 2$$

$$i = 2 \Rightarrow i = (2 + 1) \% 5 = 3 \% 5 = 3$$

$$i = 3 \Rightarrow i = (3 + 1) \% 5 = 4 \% 5 = 4$$

$$i = 4 \Rightarrow i = (4 + 1) \% 5 = 5 \% 5 = 0$$

$$i = 5 \Rightarrow i = (5 + 1) \% 5 = 6 \% 5 = 1$$

أي أننا أصبحنا ندور في حلقة تبدأ بـ 0 ...

وتنتهي بـ 4 وهكذا ... وبالتالي حصلنا على

مصفوفة دائيرية 😊

بناءً على ما تقدم نستنتج ما يلي :

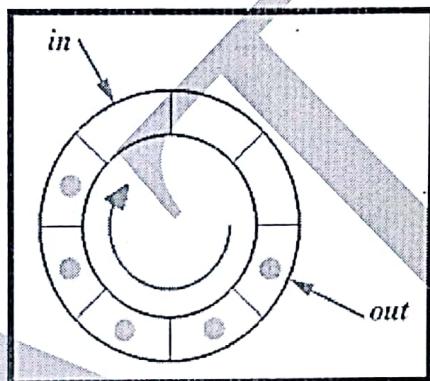
أفكار وأضحة

فكرة 1 : على المنتج التأكد من أن المخزن غير ممتلئ قبل أن ينتاج .

فكرة 2 : على المستهلك أن يتأكد بأن المخزن غير فارغ قبل أن يستهلك .

(( يستطيع المنتج توليد بند واحد فقط ويستطيع المستهلك استهلاكه بند واحد فقط ))

على اعتبار أن المخزن buffer أصبح مصفوفة دائيرية ، وبالتالي يمكننا تخيل الأمر كما يلي :



يجب أن نعلم في كل لحظة ما هو عدد المخازن الممتلئة filled buffer ، وبالتاليحتاج إلى متتحول آخر

لكي يعد المخازن الممتلئة وليكن اسمه counter

مستكون القيمة الابتدائية لهذا المتتحول تساوي 0 ومن ثم من أجل كل إنتاج جديد للمنتج producer

ستزداد قيمة هذا المتتحول بوحدة أي counter++ ... ومن أجل كل استهلاك للمستهلك

consumer ستنقص قيمة هذا المتتحول بوحدة أي counter-- .

الآن سوف نحاول تأمين حل مشكلة producer-consumer حيث نستطيع ملئ كل المخازن buffers

(عناصر المصفوفة) دون أن يتم كتابة شيء فوق شيء أو دون استهلاك شيء فارغ ...

☺ الآن قمنا بتوصيف المشكلة ... وبقى أن نبحث عن حل لها ☺

## نظم تشغيل المحاضرة الثامنة (ترامن الإجرائيات 1)



حل مقترح لمسألة المنتج - المستهلك :

قبل التعرف على حل المشكلة (الكود البرمجي اللازم) ...

سوف نتعرف على المتحولات الازمة (البيانات المشتركة) .

سنفرض أنه لدينا مخزن buffer (مصفوفة) بحجم 5 بنود ، بحيث أن هذه المصفوفة هي عبارة عن سجلات تحوي على بيانات DATA (بعض النظر عن محتواها المهم أنها تحوي شيء ما) .

```
#define BUFFER_SIZE 5
```

التصریح عن ثابت اسمه BUFFER\_SIZE وقيمته تساوی 5

```
typedef struct {
```

التصریح عن سجل اسمه item يحوي على حقل اسمه data

```
    DATA data ; } item ;
```

```
item buffer [BUFFER_SIZE];
```

مصفوفة اسمها buffer تعبر عن المخزن عدد عناصرها يساوی 5

```
int in = 0;
```

مؤشر للمكان الذي سنكتب فيه

```
int out = 0;
```

مؤشر للمكان الذي سنقرأ منه

```
int counter = 0;
```

عداد يعبر عن عدد المخازن الممتلئة

يكون المخزن فارغ عندما :

```
counter == 0
```

يكون المخزن ممتليء عندما :

```
counter == BUFFER_SIZE
```

لدينا الآن إجرائيتين المنتج consumer والمستهلك producer حيث أن لكل منهما برنامجهما

الخاص اللذان يقومان بتنفيذ طوال الوقت ...

يجب أن نعلم ما يلي :

- الإجرائيتين يقذدان في نفس الوقت concurrent execution

- الإجرائيتين متعاونتين فيما بينهما (يوجد متغيرات مشتركة المتحول counter مثلاً)

### إجرائية المنتج producer process

item nextProduced ; ← المُنتَج الذي سيتم إنتاجه وهو عبارة عن سجل يحوي بيانات DATA

while (true) { ← حلقة لا نهاية لكي يستمر في الإنتاج

while (counter == BUFFER\_SIZE) ← طالما أن المخزن ممتلىء نفذ التعليمية الخالية (لا شيء) (لا تفعل شيء انتظر فقط)

; ← التعليمية الخالية (لا شيء)

buffer[in] = nextProduced ; ← ضع المنتج الجديد في المكان in في المخزن

in = (in + 1) % BUFFER\_SIZE ; ← قم بتحريك دواراني in للمؤشر

counter++; ← زيادة عدد المخازن الممتلئة

}

سيستمر تكرار الحلقة لإنتاج منتجات جديدة ووضعها في المخزن ... وفي حال كان المخزن ممتلىء س يتم

انتظاره حتى يفرغ لكي يتم وضع منتج جديد فيه وهكذا ...

**إجراءات المستهلك consumer process**

```
item nextConsumed ;
```

المُنتَجُ الَّذِي سُيَتَّمْ أَسْتَهْلَكَهُ وَهُوَ عَبَارَةٌ عَنْ سُجْلٍ يَحْوِي بَيَانَاتٍ DATA

```
while (true) {
```

حَلْقَةٌ لَا نَهَايَةٌ لِكِي يَسْتَمِرُ فِي الْاسْتَهْلَاكِ

```
    while (counter == 0)
```

التعليمَةُ الْخَالِيَّةُ (لَا شَيْءٌ)

```
        nextConsumed = buffer[out];
```

طَالَمَا أَنَّ الْمَخْزُنَ فَارِغٌ  
نَفَذَ التَّعْلِيمَةُ الْخَالِيَّةُ  
(لَا تَفْعَلْ شَيْءٌ اِنْتَظِرْ فَقَطْ)

```
        out = (out + 1) % BUFFER_SIZE;
```

قَمْ بِتَحْرِيكِ دُورَانِيِّ  
لِلْمَؤْشِرِ out

```
        counter--;
```

إِنْقَاصُ عَدْدِ الْمَخَازِنِ الْمُمْتَلَّةِ

```
}
```

سِيسْتَمِرُ تَكَارُ الْحَلْقَةِ لِاسْتَهْلَاكِ مِنْتَجَاتِ جَدِيدَةٍ مِنَ الْمَخْزُنِ ... وَفِي حَالِ كَانَ الْمَخْزُنَ فَارِغًا سُيَتَّمْ اِنْتَظَارُ  
كِي يَمْتَلَئُ وَلَوْ بِعَنْصَرٍ وَاحِدٍ لِكِي يَتَمَّ اِسْتَهْلَاكُ مِنْتَجٍ جَدِيدٍ مِنْهُ وَعَكْنَا ...

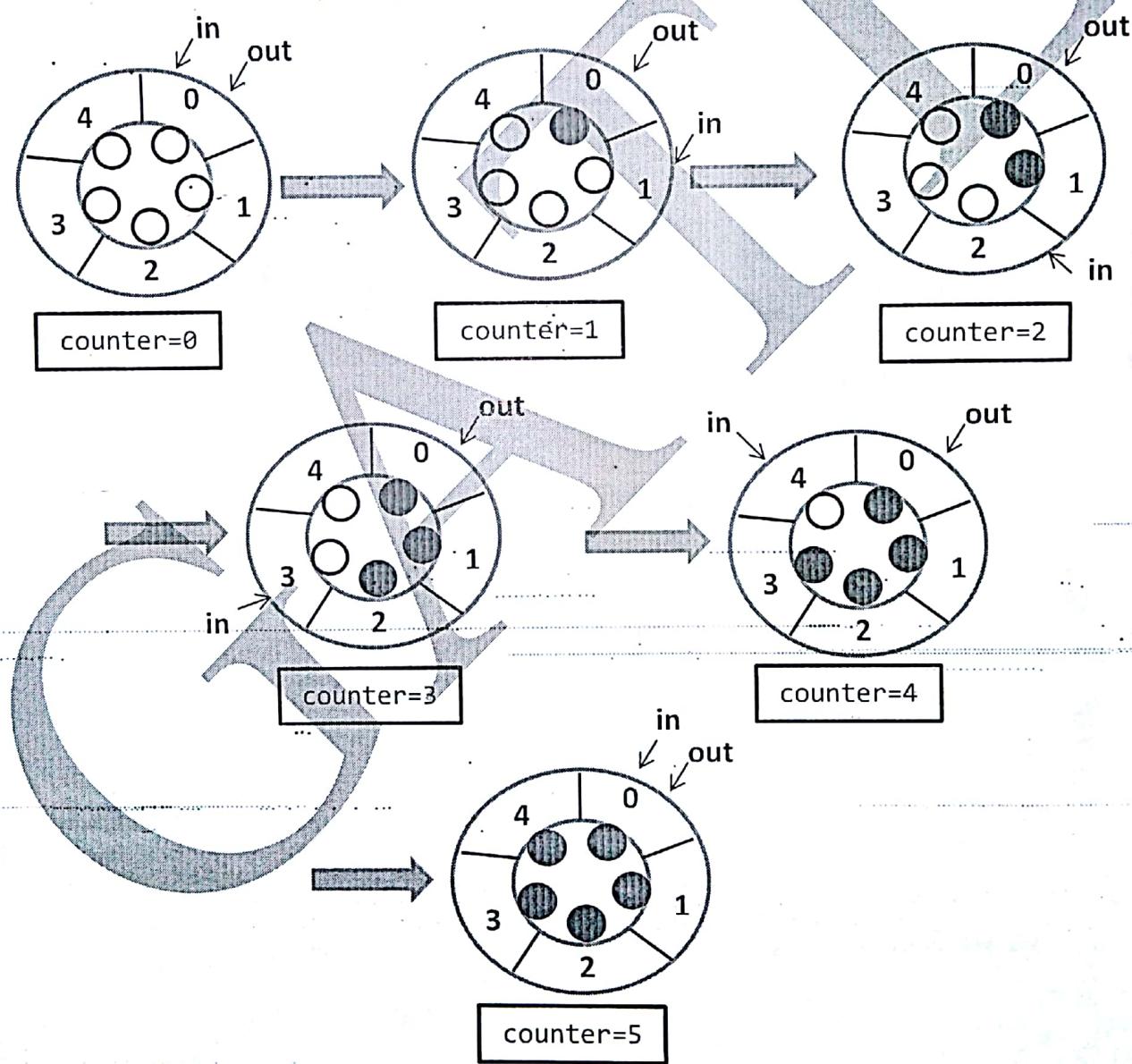
**تَوْضِيْحٌ لِبَعْضِ الْأَكْبَارِ**

عَلْمَيَّةُ الْاسْتَهْلَاكِ وَالْإِنْتَاجُ هُيَ مِجَرَّدُ عَمَلَيَّةٍ وَهُمْمَيَّةٍ ، وَهَذِهُ اِلْسَائِلَةُ هُيَ كَمِثْلُ بِرْمَجِيِّ لِلشَّيْءِ آخِرِ  
وَلَكِنْ نُسْتَعْدِمُ هَذِهِ التَّمَثِيلَ لِفَهْمِ اِلْمَوْضُوعِ بِشَكْلٍ أَقْرَبٍ إِلَىِ الْعَقْلِ ☺

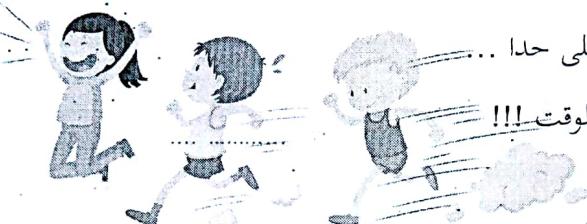
إِنَّ الْمُتَحَولَ nextProduced وَ nextConsumed عَبَارَةٌ عَنْ شَيْءٍ كَمِثْلِيِّ لِتَوْضِيْحِ فَكْرَةِ الإِنْتَاجِ  
وَالْاسْتَهْلَاكِ ، وَهُوَ فِيِ الْحَقِيقَةِ لَا يَتَمَّ إِنْتَاجُ أَوْ اِسْتَهْلَاكُ شَيْءٍ ☺

توضيح مبدأ العمل :

بما أن  $\text{BUFFER\_SIZE} = 5$  .  
 بال التالي المصفوفة (المخزن) تملك المواقع 0,1,2,3,4  
 سوف نبدأ المؤشرات بالقيمة الابتدائية  $\text{in}=0$  و  $\text{out}=0$  ولدينا المخزن فارغ بال التالي  $\text{counter}=0$   
 ولنقوم بتكرار تفاصيل عملية الإنتاج عدداً من المرات حتى يمتلي المخزن لتوضيح كيفية حدوث الإنتاج وتحريك  
 المؤشر  $\text{in}$  ، ستعبر عن المخزن الفارغ ب (دائرة فارغة ) وعن المخزن الممتلي ب ( دائرة ممتلة ) .



## Race Condition



نلاحظ أن إجرائي المتنج والمستهلك صحيحتان كلًا على حدا

لكن قد لا تعملان بشكل صحيح إذانفذتا في نفس الوقت !!!

لوضيح ذلك نأخذ المثال التالي :

لتفترض أن قيمة المتغير count حالياً تساوي 5 ، وأن إجرائي المتنج والمستهلك تنفذان التعليمتين

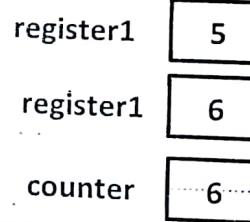
counter++ و counter-- في نفس الملحظة !

بعاً لتنفيذ هاتين العبارتين يمكن أن يأخذ المتغير counter إحدى القيم 4,5,6 ، كيف ذلك ???

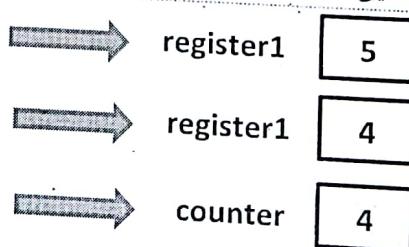
والقيمة الوحيدة الصحيحة هي counter=5 التي تتولد بشكل صحيح فقط عندما ينفذ المتنج والمستهلك بشكل تسلسلي (زيادة ثم إنفاص أو إنفاص ثم زيادة) وليس تفرعي (زيادة وإنفاص معاً).

يمكن تحقيق التعليمية counter++ بلغة الآلة كما يلي :

```
register1 = counter;
register1 = register1+1;
counter = register1;
```



```
register2 = counter;
...
register2 = register2-1;
counter = register2;
```



حيث register1,register2 عبارة عن مسجلات في وحدة المعالجة CPU

إن التنفيذ التفرعي للعبارات ذات المستوى counter-- يؤدي إلى تداخل العبارتين counter++ و

المنخفض فيما بينها بترتيب عشوائي .

## نظم تشغيل المحاضرة الثامنة (ترامن الإجرائيات 1)



وأحد الأمثلة على حدوث التداخل موضحة كما يلي (بفرض أن  $count = 5$  في البداية) :

T0 : register1 = counter



register1

5
---

↑ تدل على الزمن

T1 : register1 = register1 + 1



register1

6
---

T2 : register2 = counter



register2

5
---

T3 : register2 = register2 - 1



register2

4
---

T4 : counter = register1



counter

6
---

T5 : counter = register2



counter

4
---

↓ النتيجة النهائية

بال التالي وصلنا إلى حالة غير صحيحة وهي  $counter = 4$

الآن لو قمنا بتبدل آخر عبارتين نحصل على :

T4: counter = register2



counter

4
---

↑ النتيجة النهائية

T5: counter = register1



counter

6
---

بال التالي وصلنا إلى حالة غير صحيحة أيضاً وهي  $counter = 6$

لم نكن نصل إلى هذه الحالة غير الصحيحة لو لم نسمح للإجرائيتين بمعالجة المتحول **counter** في نفس

الوقت أي ظهر لدينا شيء اسمه حالة السباق .

(تسابق الإجرائيات فيما بينها لتعديل قيمة متحول مشترك حيث أن القيمة النهائية لهذا المتحول ستكون خاطئة وذلك بسبب تداخل تعليمات اللغة منخفضة المستوى مع بعضها)

**حالة السباق race condition** : هي الحالة التي تصل فيها عدة إجرائيات إلى البيانات نفسها

وتعالجها في نفس الوقت بحيث يعتمد ناتج التنفيذ النهائي على الترتيب الذي جرى وفقه التنفيذ .

لتجنب حالة السباق يجب أن نضمن أن

تعالج إجرائية واحدة فقط لمتغير **counter** في وقت واحد

## Critical Section المقطع الحرج

لناخذ نظاماً يحوي  $n$  إجرائية ، كل إجرائية تحوي على مقطع محدد من الكود نسميه مقطعاً حرجاً يمكن أن تقوم الإجرائية فيه بتغيير قيم بيانات مشتركة shared data مثل :

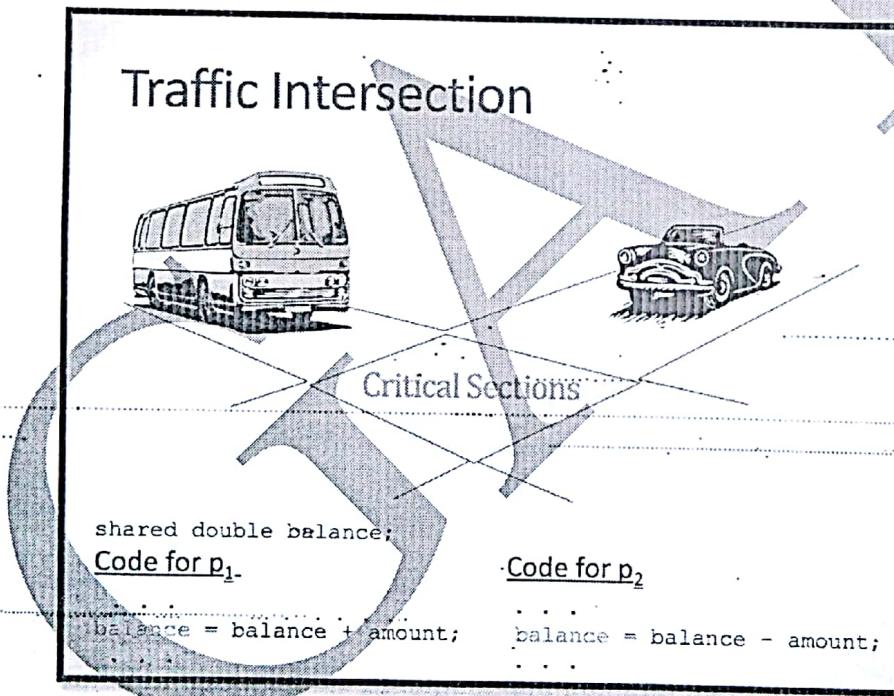
تحديث جدول ، الكتابة في ملف ، تغيير قيمة متتحول عام (مثل counter-- و counter++)

بالنالي عندما تنفذ إجرائية في المقطع الحرج الخاص بها

يجب ألا يسمح لأية إجرائية أخرى أن تنفذ ضمن المقطع الحرج الخاص بها

مثال للتوضيح لدينا إجرائيتين  $p_1$  (الحافلة) و  $p_2$  (السيارة) كلٌّ منهما يريدان الدخول إلى المقطع الحرج

حيث أحدهما يعدلان على المتتحول المشترك balance .



لاحظ أنه في لحظة واحدة يجب أن تكون إما السيارة أو الحافلة في المنطقة المظللة وإن حصل ذلك في نفس اللحظة سيخصل حادث اليم !!!

لاحظ أن كل إجرائية لها مقطع حرج خاص بها (المسألة نسبية)

تكمن المشكلة في تصميم بروتوكول (خوارزمية) معين لحل هذه المشكلة ، هذه الخوارزمية ستحمي المتتحول المشترك من النفاذ إليه في نفس الوقت في لحظة ما إجرائية واحدة فقط يجب أن تنفذ في مقطعها الحرج .

إن كل إجرائية يجب أن تعمل وفق هذا البروتوكول المنشود لضمان عدم حصول حالة سباق ...

ما هو المقطع الحرج في مسألة المتنج المستهلك ؟

## نظم تشغيل المحاضرة الثامنة (ترامن الإجرائيات 1)

```

do {
    entry section
    critical section
    exit section
    remainder section
} while (TRUE);
  
```

إذاً بشكل عام كل إجرائية p تتألف من 4 مقاطع كما يلي :

### 1\_ مقطع الدخول (Entry Section)

طلب فيه الإجرائية السماح لها بدخول مقطعها الحرج .

### 2\_ المقطع الحرج (CriticalSection=CS)

تقوم فيه الإجرائية بتغيير قيم متغيرات مشتركة (shared data) .

### 3\_ مقطع الخروج (Exit Section)

تغادر فيه الإجرائية أئمة تفاصيل مقطعها الحرج .

### 4\_ المقطعباقي (Remainder Section=RS)

توجد فيه تعليمات الإجرائية المتبقية التي ليس لها علاقة بالمتغيرات المشتركة .

سؤال هام جداً : عدد الشروط التي يجب أن يتحققها حل مشكلة المقطع الحرج ؟

### 1\_ استبعاد متبادل Mutual Exclusion

إذا كانت إجرائية ما تنفذ ضمن مقطعها الحرج ، فلا يجب لأية إجرائية أخرى أن تنفذ ضمن مقطعها الحرج .

### 2\_ تقدم Progress

إذا كان لا يوجد أي إجرائية تنفذ ضمن مقطعها الحرج ...

وووجدت إجرائيات أخرى ترغب بالدخول إلى مقطاعها الحرجة وبالتالي عملية اختيار واحدة من هذه

الإجراءات لدخول مقطاعها الحرج يجب ألا يؤجل إلى ما لا نهاية .

### 3\_ انتظار محدود Bounded Waiting

يوجد حد على عدد المرات التي يسمح فيها إجرائيات بدخول مقطاعها الحرجة بعد أن تكون إجرائية

أخرى قد طلبت الدخول إلى مقطاعها الحرج .

*Special thanks to Ahmad Mokayes*



Mohammed Moulla  
TATTOO STUDIO TATTOO STUDIO