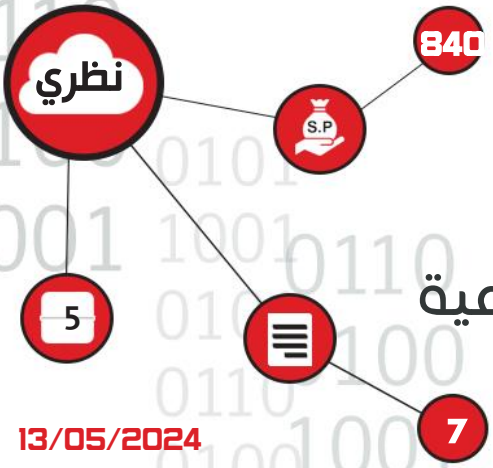
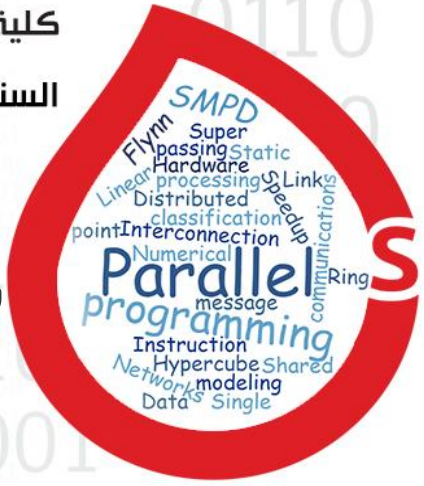


طرق حل المشكلات باستخدام البرمجة التفرعية

د. روان قرعوني

13/05/2024



RB Informatics;

البرمجة التفرعية

بسم الله الرحمن الرحيم

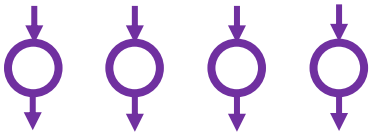
- تعرفنا في المحاضرة السابقة على أهم توابع pvm و mpi وطريقة عملهم، بالإضافة إلى بعض التوابع التجميعية المتقدمة سنتعرف اليوم على بعض الخوارزميات التفرعية (الأساسيات التفرعية) وهي الطرق المستخدمة لحل المشاكل البرمجية باستخدام البرمجة التفرعية ومن هذه الخوارزميات:
- الحسابات التفرعية الصريحة
- التجزئة
- فرق تسد
- pipeline

تذكرة : الاستجابة تتحسن بزيادة المعالجات لحد معين،

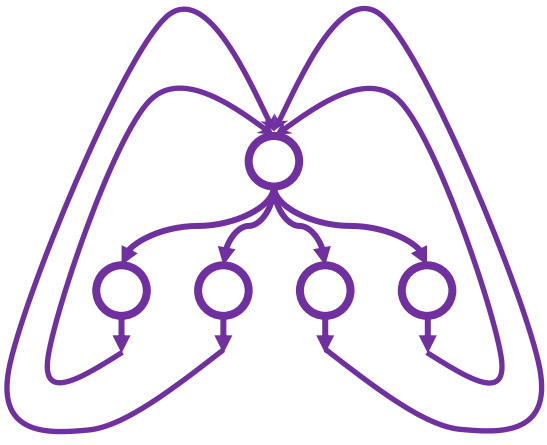
الفكرة الأولى: الحسابات التفرعية الصريحة

تستخدم هذه الطريقة مع المسائل التي تتفرع بشكل كامل، أي لا يوجد ترابطات بين البيانات،

مثال: مصفوفة مربعة $4 * 4$ ونريد تطبيق عملية على عناصرها، نقوم بتقسيم الـ Data، وتوزيع الأقسام على المعالجات المتاحة بدون الحاجة لتواصل المعالجات مع بعضها، فلنفرض أن المعالجات المتاحة عددها أربعة، سيقوم كل معالج بمعالجة 4 قيم من المصفوفة السابقة



كما يجب وجود مهمة مسؤولة عن توزيع الـ Data، وتجميعها بعد انتهاء المعالجات الأربعة من معالجة القيم .



- في القسم العملي يكون تحقيق المثال السابق:

- في pvm، يكون لدينا مهمة تمثل parent task، تقوم بتنفيذ التابع Spawn لخلق المهام المطلوبة، وباستخدام مصفوفة الـ id الناتجة تقوم الـ parent task بإرسال مواقع القيم الواجب معالجتها لكل معالج، بعد تنفيذ كل مهمة للقسم المُوكل به، تقوم المهمة الأب (parent task) بتجميع النتائج من المهام الأخرى، أما في mpi، في بداية المشروع (عند الضغط على زر run) نقوم بتحديد عدد المهام المطلوبة لتنفيذ البرنامج، ثم نقوم بتقسيم الـ Data وإرسالها للمهام حسب الـ rank الخاص بكل مهمة
- أي في الـ mpi، لا يوجد مهمة أب، لكن يوجد حاجة لمهمة تقوم بالتوزيع والتجميع وغالباً ما تكون المهمة التي قيمة الـ rank فيها يساوي الصفر.
- تذكرة: مفهوم rank مشابه لمفهوم id حيث يقوم بتمييز المهمة

- **لدينا 3 نقاط أساسية يجب مناقشتها .**

1. **المعالجات ليست بنفس السرعة دائما**

قد يوجد اختلاف في مواصفات المعالجات المستخدمة، كأن تختلف بعدد الـ RAMs أو بنوع النواة المستخدمة في تصميم المعالج ومنه يجب أن نلاحظ عند استخدام هذه النوع من الخوارزميات أنه قد تختلف السرعة بين المعالجات، وهذا يؤدي إلى إبطاء العملية بحال كان أحد المعالجات بطيء <u>لذلك كيف سنوجد الحل المناسب لذلك؟!</u>

1. الحل الأول: load balance

يستخدم عند معرفة سرعة المعالجات المتاحة، يستخدم لتوزيع العمل بحيث يتناسب مع سرعة المعالج لتحقيق أعلى استجابة ممكنة

- مثال : لدينا معالجين أحدهما سريع يستطيع إنجاز مهمة معينة بنصف الزمن اللازم للمعالج الثاني، في هذه الحالة أقوم بتوكيل المعالج السريع بمهمة أخرى ريثما ينتهي المعالج البطيء من تأدية مهمته
- أو بطريقة أخرى يمكن تقسيم المسألة إلى 3 أقسام وتوكيل المعالج السريع بقسمين ونوكل المعالج الثاني بالقسم الثالث.

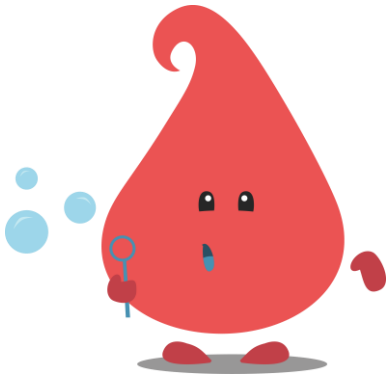
2. الحل الثاني : التجريب العشوائي random

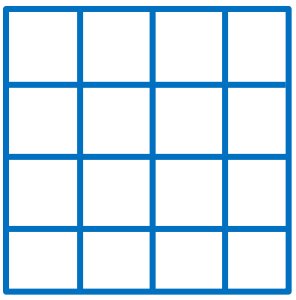
نقوم بتنفيذ البرنامج، وبعدها نقوم بالنظر إلى الإحصائيات لمعرفة أداء كل معالج، وعند تنفيذ البرنامج في المرات المُقبلة نعيد توزيع المهام حسب سرعة كل معالج، أي بتوزيع أصح.

- ما الفائدة من إعادة تنفيذ نفس البرنامج ؟
- وجود العديد من البرامج المستمرة على الدوام مثل برامج التنبؤ بالطقس، معالجة صور أقمار صناعية. كما تعد طريقة أوفر للجهد من طريقة load balance

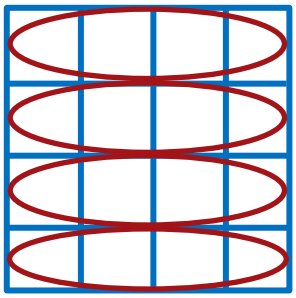
2. آلية التوزيع

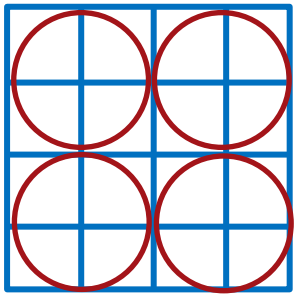
تعد واحدة من المعاملات المساهمة في سرعة البرنامج التفرعي مثال ليكن لدينا المصفوفة التالية ونريد توزيعها على 4 معالجات



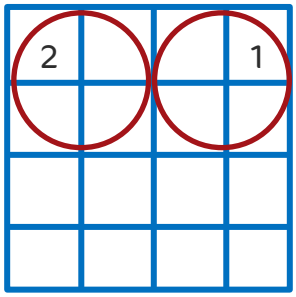


من الطرق الممكنة لتوزيعها





في هذا المثال لا يوجد فرق بين طريقتي التوزيع، ولكن لنفرض أنه يوجد اعتمادية بين العنصرين التاليين



عندها تكون طريقة التوزيع الثانية أسرع (على اليسار).

3. أحيانا يكون توزيع الـ Data مُكلف، مثلا في حالة مصفوفة حجمها (مليار*مليار) عملية إرسال سطر تكون مكلفة ومنه يجب مراعاة حجم البيانات المُرسلة

قد يكون الحل هو استخدام shared memory، أي وجود ذاكرة مشتركة تستطيع جميع المهام الوصول إليها، في هذه الحالة نقوم بإرسال موقع الداتا للمهام عوضا عن إرسالها .

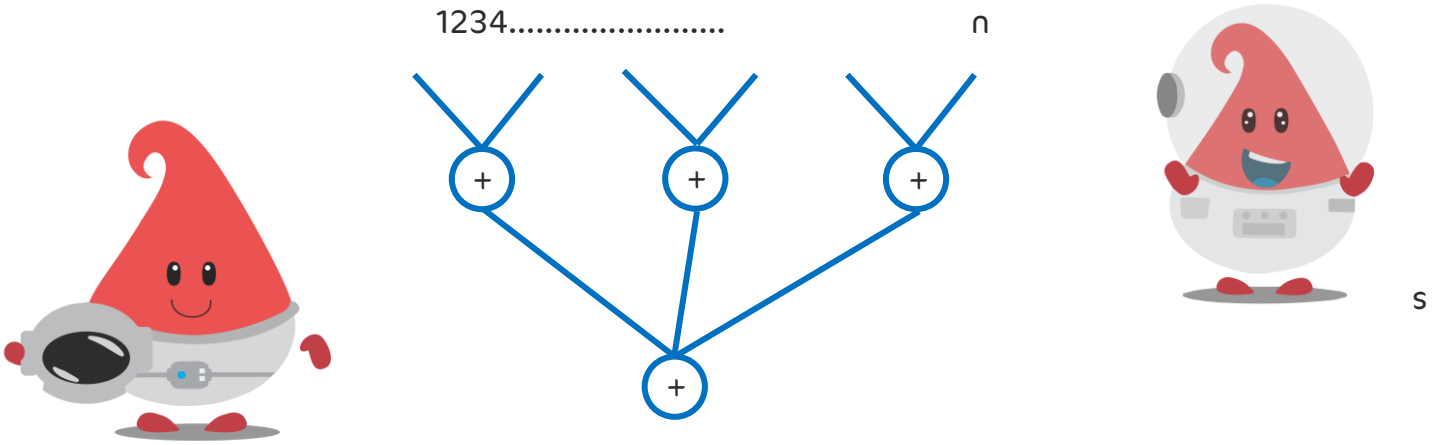
- هل يمكننا استخدام shared memory دائما ؟
- تعد الshared memory مُكلفة، فمثلا في حالة وجود سيرفرات في دول مختلفة سيكون من المُكلف استخدام ذاكرة مشتركة، بل إضافة إلى صعوبة إدارة التضاربات كما تحدثنا في المحاضرة الثانية.

التجزئة

تستخدم هذه الطريقة مع المسائل التي تكون فيها الداتا تُقسم لكتل وتُرسل كل كتلة إلى مهمة، دون استخدامها من مهمة أُخرى.

مثال :

- لنقم بكتابة برنامج يقوم بجمع الأعداد من الواحد و حتى رقم كبير، في هذه المثال نستطيع تقسيم الأعداد إلى مجموعات و إرسال كل مجموعة لمهمة , ثم جمع خرج هذه المهام



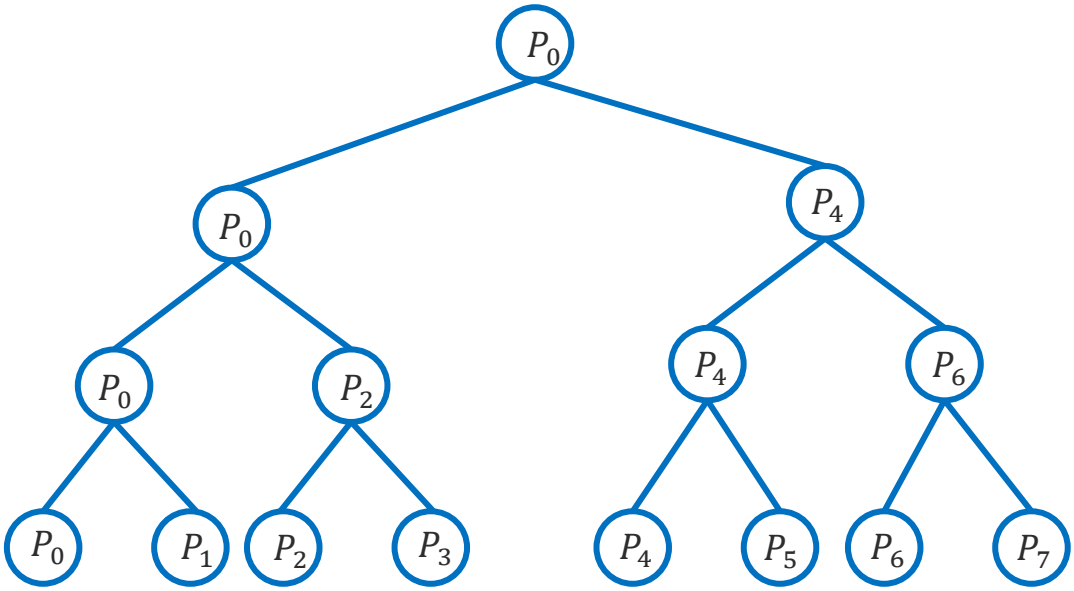
في التجزئة نستخدم أيضا الـ shared memory، كما أنه في هذا النوع من المسائل نستخدم التوابع التجميعية التي تعرفنا عليها في المحاضرة السابقة لتهيئة الاتصال مرة واحدة لكل القيم.

- ما الفرق بين طريقة التجزئة والطريقة السابقة " الحسابات التفرعية الصريحة "؟
- تعتبر الطريقتين من طرق التفكير التفرعي حيث:
- في الأولى يكون لدينا مهمة واحدة تقوم بتوزيع الداتا على مجموعة معالجات، ثم تقوم بتجميع خرج هذه المعالجات .
- أما في التجزئة، كل جزء من الداتا يطبق عليه نفس العملية، باستخدام اكثر من معالج، ثم يقوم اكثر من معالج بعملية التجميع.

فرق تسد

يكون لدينا ما يشبه شجرة المعالجات ويقوم كل معالج باستخدام نتيجة عمل المعالجات الأبناء في الشجرة،

- مثال:
ليكن لدينا 8 معالجات ولنقم بتمثليهم باستخدام شجرة ثنائية لتسهيل الفكرة فيكون لدينا الشكل التالي



في المخطط السابق كل معالج يقوم بعمله باستخدام المعالجات في المستوى الأدنى وتعبر طريقة التوزيع هذه طريقة شبكية، حيث يقوم المعالج ρ0 باستخدام النتيجة الخاصة به مع نتيجة المعالج ρ1، تعد بنية التوزيع هذه بنية شجرية. يشبه مبدأ هذه الطريقة مبدأ dived & conquer حيث نقوم بتجزئة المسألة إلى ابسط ما يمكن، وننطلق منه لحل المسألة كاملة

pipeline

الفكرة العامة لهذه الطريقة: يكون لدينا تسلسل من العمليات، كل عملية تستخدم خرج العملية التي سبقتها كما في المخطط التالي

في المخطط السابق يبدأ التنفيذ من المعالج ρ0 ثم يقوم بتمرير نتيجة التنفيذ إلى المعالج ρ1، وبعد تمريرها يكون المعالج ρ0 مُتاح لتنفيذ عمل آخر .

مثال :

- ليكن لدينا مصفوفة نريد أن نقوم بزيادة كل عنصر من عناصرها بـ x، ثم نضربه بـ y ، ثم نقسمه على z وسنوزع العمل على 3 معالجات .
- نقوم بتمرير العنصر الأول في المصفوفة على المعالج الأول ρ0 للقيام بالجمع، وبعد تمرير القيمة الناتجة للمعالج الثاني
- يقوم المعالج ρ0 بتطبيق عملية الجمع على العنصر الثاني في المصفوفة .
- هكذا نضمن ان تكون الاستجابة أعلى، لأن جميع المعالجات تقوم بالمعالجة في الوقت نفسه .

الـ pipeline 3 أنواع:

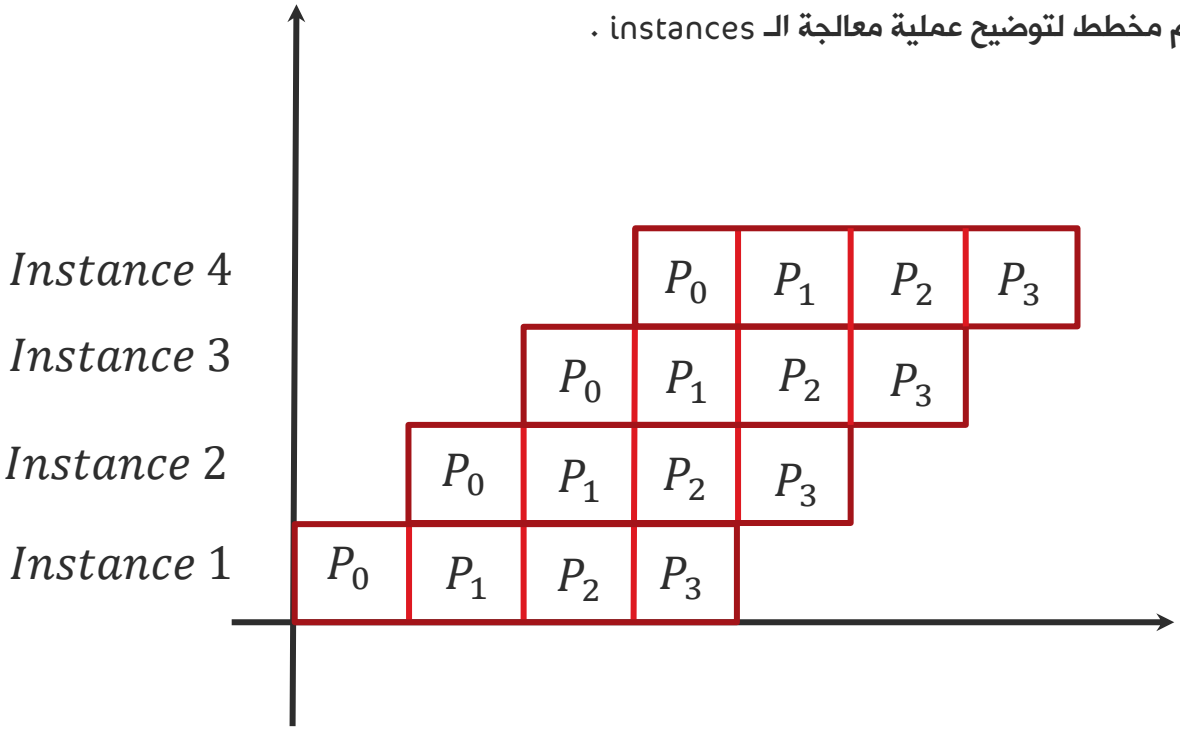
1. type 1:

يستخدم هذه النوع مع المسائل التي يكون لدينا فيها تطبيق نفس العمل على اكثر من instance

- مثال نريد عمل برنامج يقوم بتعديل قيمة العناصر في مصفوفة، وسوف يعالج اكثر من مصفوفة، في هذا المثال تكون المصفوفة هي الـ instance.
- مثال: نريد تنفيذ تعديل معين على مجموعة من الصور، في هذه الحالة تكون الصور هي الـ instance
- في كلا المثالين يجب ان يكون التعديل هو نفسه على جميع العناصر (الـ instances)

نلاحظ أن مبدأ عمله هو Single Instruction multiple Data .

لنقم برسم مخطط لتوضيح عملية معالجة الـ instances .



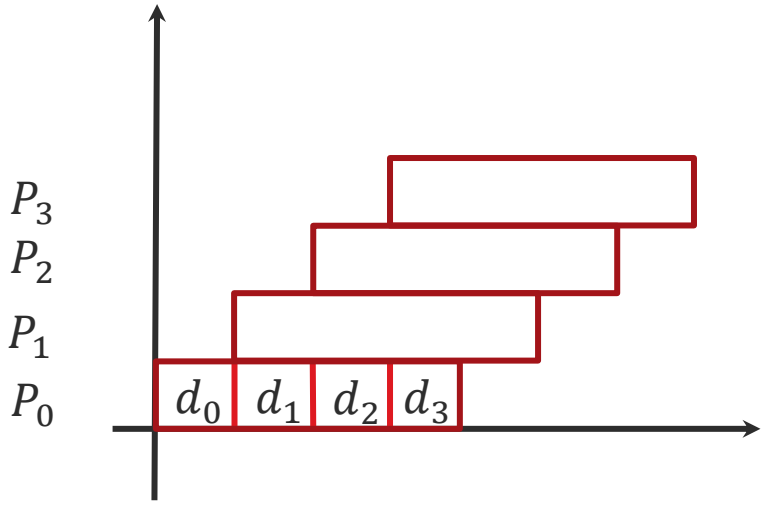
لاحظ انه في المخطط السابق، بعد انتهاء المعالج P_0 من إجراء العمل المُوكل إليه على الـ instanceالأولى، ينتقل مباشرة لمعالجة الـ instanceالثانية، وبهذا نقوم بزيادة الاستجابة من خلال تشغيل كل المعالجات في الوقت نفسه.

لنفرض انه في المخطط السابق كان كل معالج يحتاج ثانية لإنهاء عمله على الـ instanceالواحدة، يكون زمن التنفيذ 7 ثواني، بينما في حالة انتظرنا كل instanceلتنتهي قبل البدء بمعالجة التي تليها سيكون الزمن 16 ثانية.

2. type 2 :

بمبدأ مشابه للنوع السابق ، ولكن بتطبيقه على الـ Data.

توضيح : في حالة كان يجب مرور الـ Data ذاتها على جميع المعالجات، كما في أمثلة التعديل على عناصر مصفوفة نستخدم هذا النوع فلنلاحظ المخطط التالي..



بعد انتهاء المعالج p0 من معالجة الـ Data الـ d0 يرسلها إلى المعالج p1 ليقوم بعمله و ينتقل مباشرة لمعالجة d1.

3. type 3:

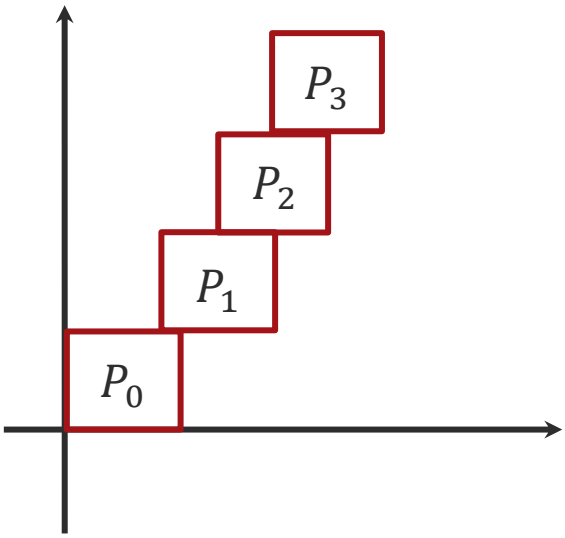
في بعض المسائل، قد لا تكون مضطر إلى انتظار المعالج السابق لتنفيذ كامل العمل المُوكل إليه، أي أنه قام بالجزء المطلوب ليتمكن المعالج التالي له من إجراء عمله بدون حدوث أخطاء،

مثال : ليكن لدينا المعالجين P_0, P_1 وكان المعالج P_1 يريد قيمة متحول x،

قد ينتهي المعالج p_0 من تطبيق كامل عمله على قيمة المتحول x،

في هذه الحالة يقوم المعالج P_1 ببدء التنفيذ قبل انتهاء المعالج p_0 بالكامل،

تحديدا في اللحظة التي ينتهي المعالج p_0 من إجراء كافة التعديلات على قيمة x.



The End...