

كلية الهندسة المعلوماتية

نظم تشغيل 1

المحاضرة الرابعة ((FORK)) عدد المصفحات : 15

السنة الثالثة



Greateness Achieves Through Excellence

الزراعة - مقابل باب السكن الجامعي - شارع مسايا



04 2496237

0960 066 016



www.facebook.com/groups/ITech.GATE

استدعاء النظام (fork) بالتفصيل



يستخدم استدعاء النظام (fork) لإنشاء الإجرائيات processes.

بحيث أنه لا يأخذ أي وسطاء ويرد رقم العملية PID الناتجة.

الهدف الرئيسي من (fork) هو إنشاء إجرائية جديدة تكون ابن child للإجرائية التي قامت باستدعاء هذا

التابع (استدعاء النظام fork)، بحيث أنه بعد أن تكون الإجرائية الابن child process قد ولدت فإن كلا الإجرائيتين سوف تنفذان معاً كل ما هو موجود من تعليمات بعد هذا الاستدعاء.

(أي يبدأ من نفس النقطة في التنفيذ)، لكن في هذه الحالة فنحن يجب أن نميز بين الإجرائية الاب parent والإجرائية الابن child وهذا يتم ببساطة عن طريق اختبار القيمة المعادة من الاستدعاء نفسه.

توضيح :

عندما نستدعي التابع (fork) فإن الكود يتكرر نفسه في إجرائيتين مختلفتين أحدهما الأب والأخر الابن لكن !! في الإجرائية الأب سوف يعيد هذا التابع رقم الإجرائية الابن التابعة للإجرائية الأب وهو رقم PID معين، بينما في الإجرائية الابن فإن هذا التابع سوف يعيد صفرًا وبالتالي نستطيع كتابة كود يعالج الحالتين (من أجل الأب ومن أجل الابن) (يعنى كود 1 in 2 ☺).

ملاحظة برمجية: الاستدعاء (fork) هو تابع يرد قيمة ، وبالتالي لاستدعائه يوجد طريقتان :

1_ أن نستدعيه بالشكل ;fork(); وبالتالي نحصل على القيمة المعادة من خلال التابع ;getpid();

2_ أن نخزن القيمة المعادة في متتحول من النمط pid_t

بالنهاية نحصل على القيمة المعادة من خلال المتتحول نفسه . ←

نظم تشغيل المحاضرة الرابعة (Fork())

عندما يرد القيمة -1 هذا يعني حدوث فشل failure ولن يتولد الابن وسيتم تحديد رقم الخطأ errno بشكل مناسب

القيم المعاادة من الاستدعاء (fork())

- $fork() < 0 \Leftarrow$ فشل عملية التوليد للإجرائية الابن .
- $fork() = 0 \Leftarrow$ نحن في العملية الابن .
- $fork() > 0 \Leftarrow$ نحن في العملية الاب والرقم المعاد هو رقم العملية الاب .

ملاحظة : بعد تنفيذ الاستدعاء (fork) يتم إنشاء نسخة طبق الأصل من فضاء عناوين الأب parent's address space

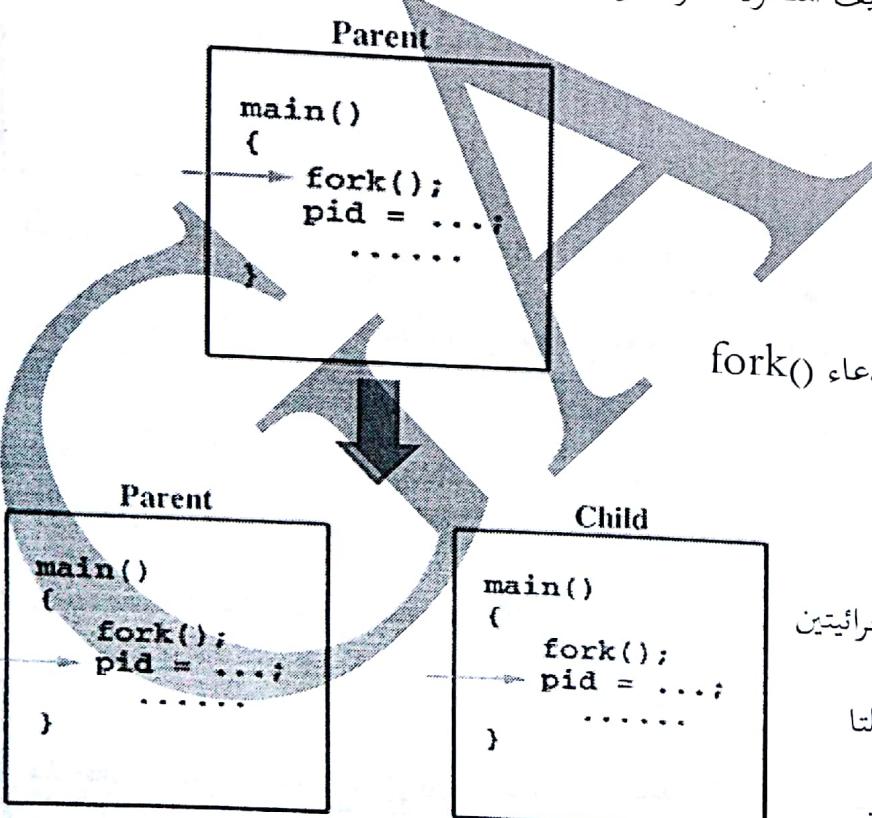
فضائي عناوين (نسخة من المتغيرات وقيمها لكن في موقع آخر من الذاكرة).

أي أنه أي تعديل لقيم المتغيرات في الإجرائية الاب لن يؤثر على قيم المتغيرات الموجودة في الإجرائية الاب والعكس صحيح ...

الآن لنأخذ المثال التالي بغض النظر عن تعريف المتغيرات واستيراد المكتبات المهم الآن كيفية العمل ،

لنفترض الآن لدينا البرنامج التالي :

(أثناء التنفيذ هو عبارة عن إجرائية)



عند بدء التنفيذ فإن أول تعلية هي الاستدعاء (fork())

أي أنه الآن سيتم إنشاء الإجرائية الابن:

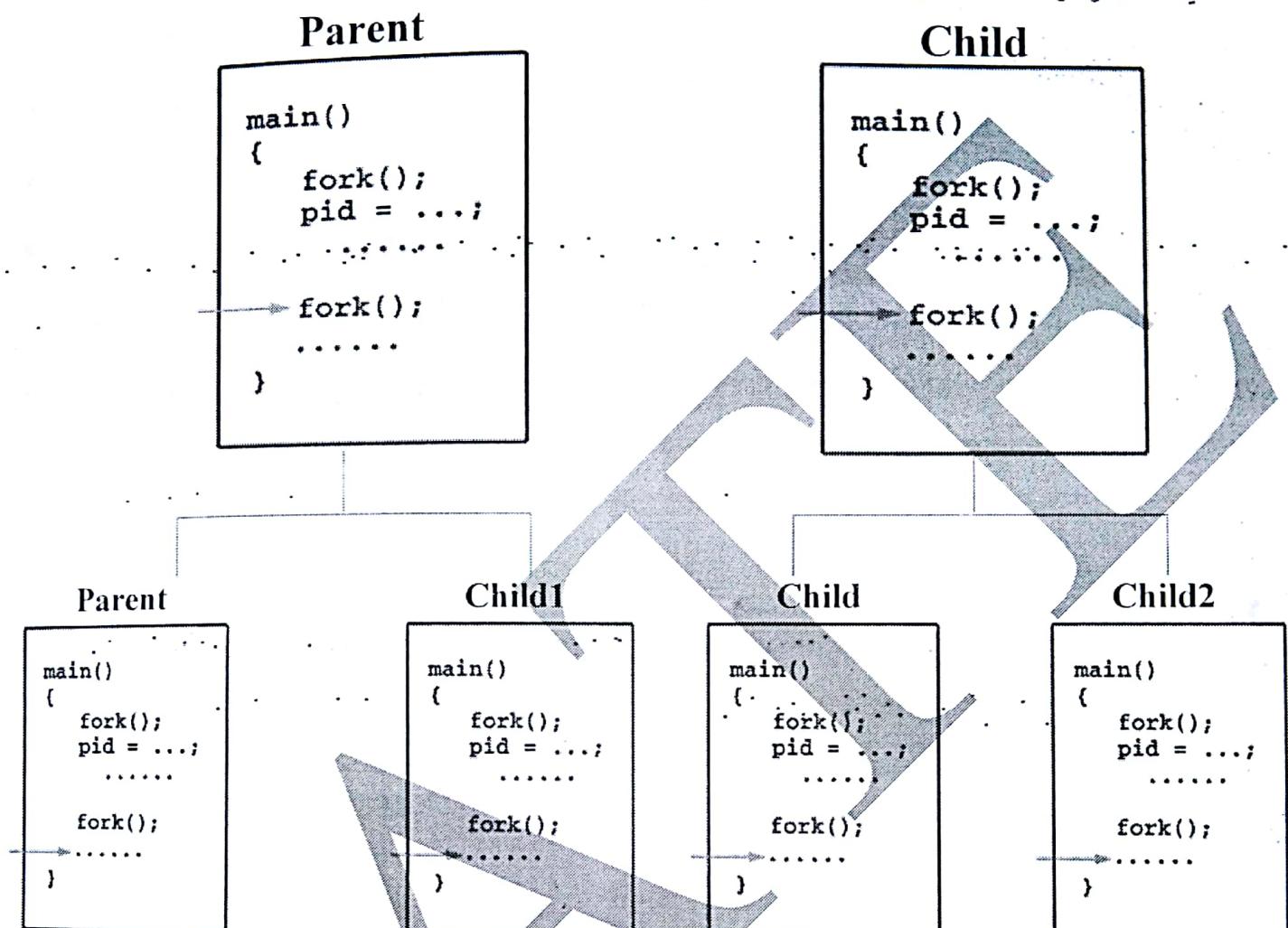
الآن أصبح لدينا البرنامج نفسه ينفذ في إجرائيتين

متختلفتين child و Parent بحيث أن كلتا

الإجرائيتين تبدآن التنفيذ من التعليمة التالية

والتي هي ;...=pid (بغض النظر عن ماهيتها المهم الفكرة الآن).

الآن في حال وجود استدعاء `fork()` آخر وبالتالي :



وبالتالي حصلنا الآن على 4 إجراءات بسبب وجود استدعاء `fork()` آخر 😊

نتيجة : عدد الإجراءات المولدة من برنامج يحوي على n استدعاء للتابع `(fork())` يساوي :

```
int main() {  
    fork();  
    fork();  
}
```

$$n \Rightarrow 2^n$$

بالتالي إذا كان لدينا التابع التالي :

فإن الجواب هو بالتأكيد 4 إجراءات

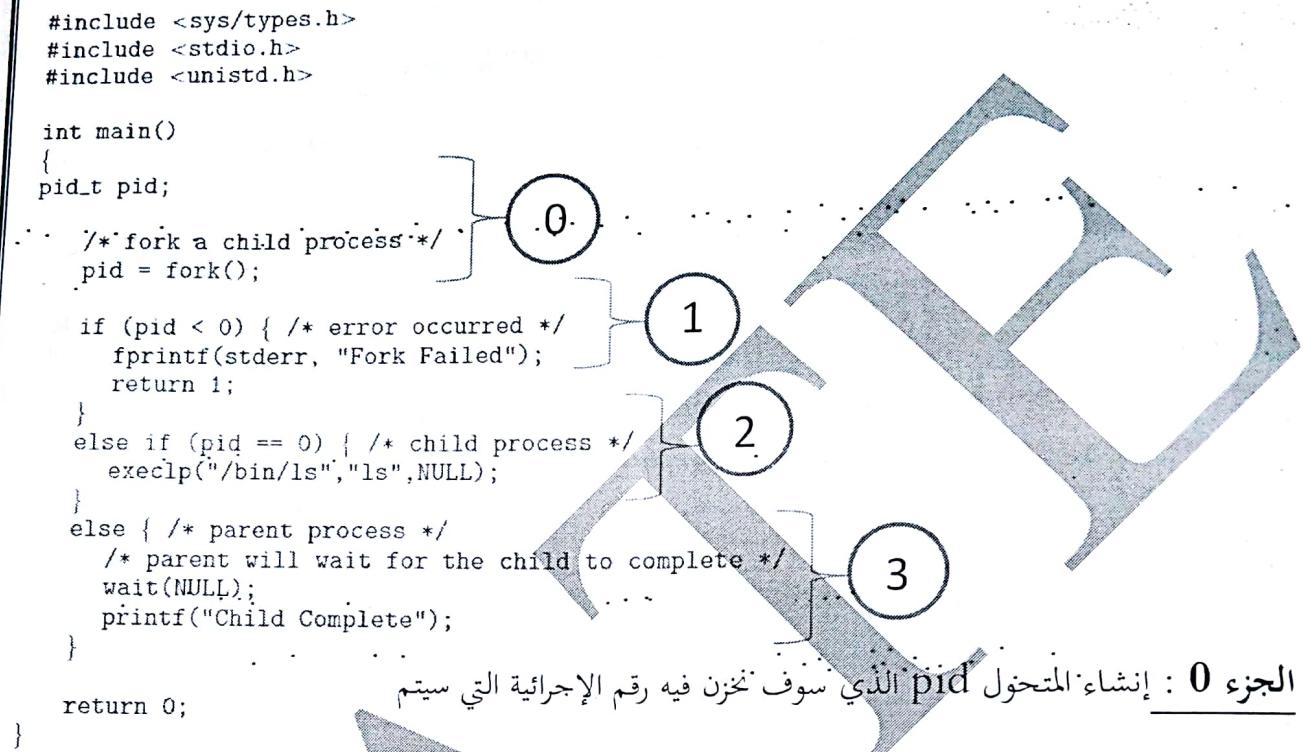
وإن كان 3 استدعاءات وبالتالي 8 وهكذا

ملاحظة خطيرة : عدد الإجراءات الأبناء = عدد الإجراءات الكلية 2^n مطروحاً منه 1 لأن الإجرائية

الاب يجب إنقاذه (ليست أبناء بالتأكيد) {`childrens = total - 1`}

نظم تشغيل المحاضرة الرابعة (fork())

يمكنا تنفيذ استدعاء النظام (fork()) من خلال واجهة UNIX :



الآن إذا تمت عملية إنشاء الإجرائية بنجاح فإننا أمام حالتين :

أ_ الإجرائية لها الرقم 0 (pid=0) وبالتالي هذه الإجرائية هي الإجرائية البنت هذا يعني أنه تم تنفيذ الجزء (2).

ومن ثم سيتم تنفيذ استدعاء النظام exec() لكي يتم تحميل الكود الثاني ووضعه في الإجرائية البنت.

ب_ الإجرائية تملك رقمًا أكبر من صفر (pid>0) وبالتالي هذه الإجرائية هي الإجرائية الأم أي أنه تم تنفيذ

الجزء (3) ومن ثم سيتم تنفيذ استدعاء النظام wait(NULL) لكي تنتهي الإجرائية الأم حتى تنتهي

الإجرائية البنت ذات الرقم صفر من تنفيذ كل تعليماتها وعندما تنتهي يتم طباعة الرسالة

"child complete" أي أن الابن تم إنشائه وانتهى التنفيذ بشكل صحيح.

لكن !!! في حال كانت القيمة المعادة من التابع fork() هي قيمة سالبة وبالتالي حدث خطأ أثناء إنشاء

الإجرائية وسوف يتم طباعة الرسالة "fork failed".

سؤال هام : كيف يمكننا جعل الإجرائية الابن أن تنفذ كود خاص بها مختلف عن

كود الإجرائية الاب المستدعي التابع (fork())

بعد استدعاء التابع fork لا يمكننا معرفة أين نحن الآن ...

أي هل نحن بحالتيون في الإجرائية الأب أم في الإجرائية الابن ???

المقصود أنه لو قمنا بتنفيذ تعليمية ما ، كيف نعلم أنها تنفذ الآن في الأب أم في الابن ???

بساطة لسنا أبو العريف ... ولا كنا وقت الحادثة !!!

بالتالي بالاستفادة من القيمة المعادة من التابع fork نستطيع تحديد أين نحن الآن ؟

وبناءً عليها (للقيمة المعادة) نستخدم if-else و نضع ضمن جسم ال if كود أول

و ضمن جسم ال else نضع كود ثانٍ وبهذا يجعل الابن ينفذ كود مختلف عن الاب 😊

ليكن لدينا التابع ChildProcess() ينفذ كود أول (خاص بالابن)

وليكن لدينا التابع ParentProcess() ينفذ كود ثانٍ (خاص بالاب)

معروfan كما يلي (بغض النظر عن الجسم body) :

```
void ChildProcess()
{
    ....
}
```

```
void ParentProcess()
{
    ....
}
```

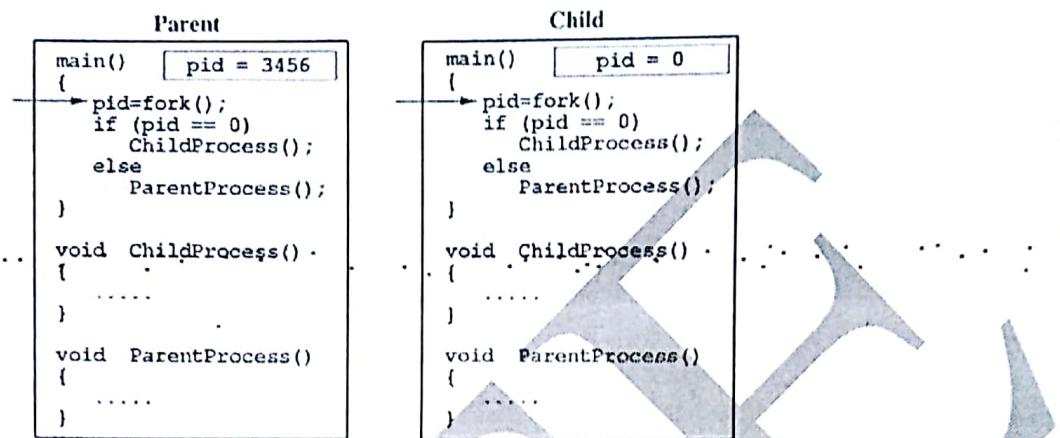
ولدينا تابع رئيسي main() ونقوم باستدعاء النظام fork()

وسنقوم بجعل الابن ينفذ التابع ChildProcess() والاب ينفذ ParentProcess()

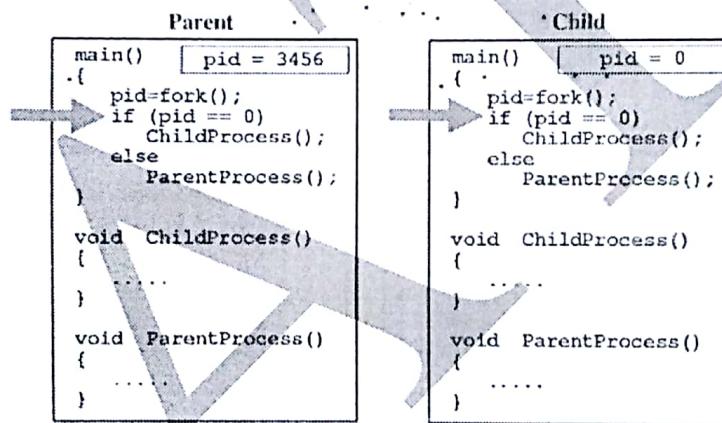
ل التابع معاً كيف سنقوم بذلك ...

نظم تشغيل المحاضرة الرابعة (Fork())

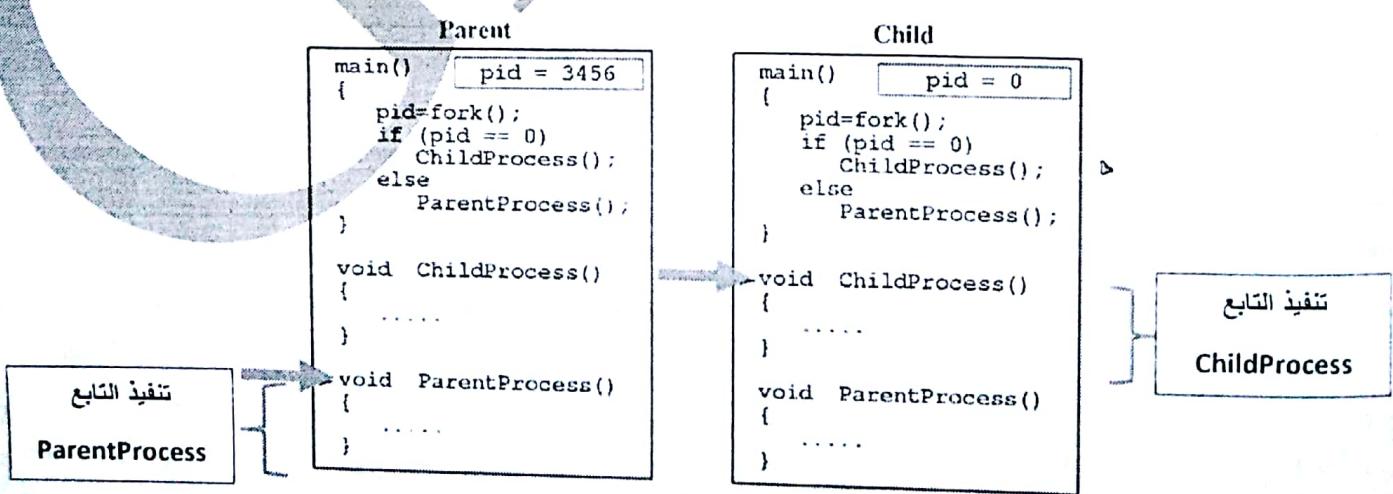
لحل المشكلة سنقوم ب تخزين القيمة المعاادة من التابع fork() في متتحول اسمه pid



الابن سيكون فيه $pid=0$ (ليس له أبناء) ولكن هذا ليس رقم الإجرائية الخاص بالابن والاب سيكون فيه $pid=3456$ (رقم عشوائي) وهو رقم عملية الابن التي تم إنشائها . بالتالي بعد ذلك سيتم اختبار قيمة الـ pid في الشرط `if` من أجل الإجرائيتين وعلى أساسه سيتم استدعاء التابع الموفق ...

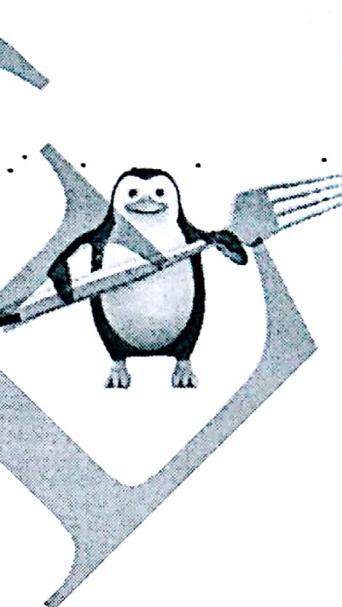


مثلاً الإجرائية ذات قيمة المتتحول $pid=3456$ سوف تنفذ التابع `ParentProcess()` والإجرائية ذات الرقم 0 سوف تنفذ التابع `ChildProcess()`



ليكن لدينا الكود التالي :

```
//Assume more lines before this
cout<<"Forking ... : "<<endl;
int pid = fork();
cout<<"forked !!! "<<endl;
cout<<"pid = "<<pid<<endl;
//Assume more lines after this
```



أول سطر وآخر سطر مجرّد تعليق comment للإشارة إلى وجود بقية الكود في الأعلى والأسفل ...

نقوم أولاً بطباعة ... Forking ثم النزول سطر ، هنا نحن لدينا إجرائية واحدة تنفذ (لم ننفذ fork)

بالتالي الذي ينفذ تعليمة cout هي إجرائية وحيدة وهي إجرائية الأب Parent (تنفيذ cout مرة واحدة)

ثم لدينا استدعاء لـ fork وتخزين القيمة المعادة في المتغير الصحيح pid ← الآن أصبح لدينا إجرائيتين تنفذ الكود بدءاً من هذه التعليمة (من عند الطريق أبو الشوكة) ...

إجرائية الأب سيكون فيها pid = 572 (وهو رقم إجرائية الابن التي تم إنشاؤها)

بينما إجرائية الاب سيكون فيها pid = 0 (لماذا ؟؟؟)

في هذه المعادلة السابقة نعلم فقط رقم إجرائية الاب ... لكننا لا نعلم رقم إجرائية الاب ☺

انسى موضوع رقم إجرائية الاب لنركز على الكود

نظم تشغيل المحاضرة الرابعة (fork())

الآن لدينا التعليمية " cout<<"forked !!! ولدينا 2 إجرائية تنفذان هذه التعليمية

سيتم طباعة !!! forked مرتاًان ←

ثم لدينا التعليمية; cout<<"pid = " << pid وما زال لدينا 2 إجرائية قيد التنفيذ

بالنالي سيتم طباعة pid = 0 بالنسبة لإجرائية الآرين ، وسيتم أيضاً طباعة pid = 572 بالنسبة لإجرائية الآخرين !!!

الفكرة الجديدة هنا أنها لا نعرف هل سيتم طباعة pid = 0 أولاً أم سيتم طباعة pid = 572 ثانياً ...

وهذا يعتمد على المعالج CPU نفسه وماذا ينفذ في اللحظة

وينكون الخرج النهائي كما يلي :

Output:

```
Forking...
Forked!
Forked!
pid = 572
pid = 0
```

قد نحصل على أي ترتيب آخر 4 تعليمات طباعة (2 تعليمية و 2 إجرائية)
أي قد نحصل على! Forked! ثم pid = 0 ثم pid = 572 أو قد نحصل على pid = 0 ثم pid = 572
أو قد نحصل على! Forked! ثم pid = 572 ثم pid = 0 أو قد نحصل على pid = 0 ثم pid = 572

لماذا لن نحصل على الخرج التالي ???

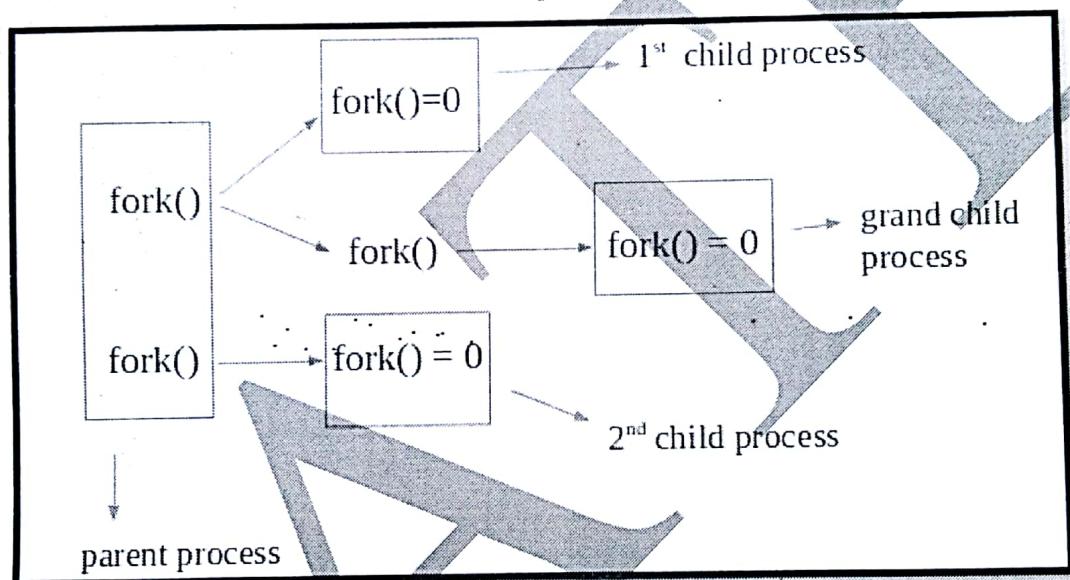
Forked! pid = 572 pid = 0 Forked!

إذا لم تعرف فهي كارثة كبيرة

مثال توضيحي حول fork

```
int main() {
    fork();
    fork();
    ...
}
```

من أجل الكود السابق سوف يكون لدينا الشكل التالي :



— من أجل أول fork سوف نحصل على ابن أول (pid=0) والاب نفسه

الآن تم تنفيذ أول fork

— الآن الابن الأول 1st child سوف ينفذ أيضاً fork ⇐ سوف نحصل على ابن حفيد (pid=0) child والاب نفسه .

— والأب أيضاً parent سوف ينفذ أيضاً fork ⇐ سوف نحصل على ابن ثاني (pid=0) child والاب نفسه .

الآن انتهي تنفيذ ثاني fork

نظم تشغيل المحاضرة الرابعة (fork())

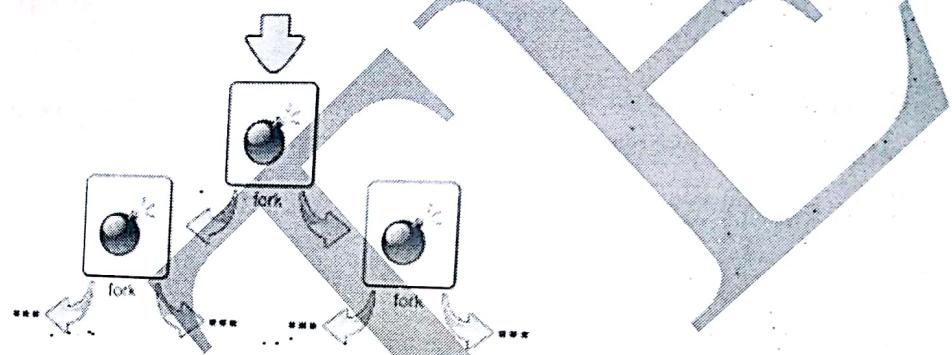
استخدام fork كقنبلة موقوتة Bomb



: fork() bomb

عبارة عن برنامج يؤذى نظام التشغيل (أنظمة تشغيل Linux فقط) عن طريق استهلاك الذاكرة RAM بشكل كامل ، بحيث أنه يقوم بإنشاء عدد لا تهاوي من الإجراءات لكنه يتم ملئ الذاكرة .

أوعي تكره نظام
DOS التشغيل



يعتبر شكل من أشكال الهجمات على الضحية وينسمى :-

Denial-of-service (DOS) "هجوم الحرمان من الخدمة"

و عندما يتم تفعيل "القنبلة fork" بنجاح فإنه من المستحيل إكمال العمل على نظام التشغيل بشكل طبيعي ويجب أن نقوم بإعادة تشغيل النظام reboot لكي يتم التخلص من كل الإجراءات التي تم إنشائها .

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    while(true)
        fork();
    return 0;
}
```

نفس الشيء while(1)

ملاحظة : لا تقم بتجربة fork() bomb على حاسوبك إلا إذا كنت مستعداً لخسارة بعض المعلومات ولربما تلف بعض القطاعات في القرص الصلب HDD لديك ☺ .

نظم تشغيل المحاضرة الرابعة (Fork())



Windows fork Bomb

طريقة شريرة :

1_ إنشاء ملف نصي جديد و كتابة التعليمات التالية فيه باستخدام المفكرة :

```
:run
start %0
goto run
```



ان windows في start %0
نكافع (fork() في linux

2_ نذهب إلى ملف file في الأعلى ونختار حفظ باسم save as ونكتب :

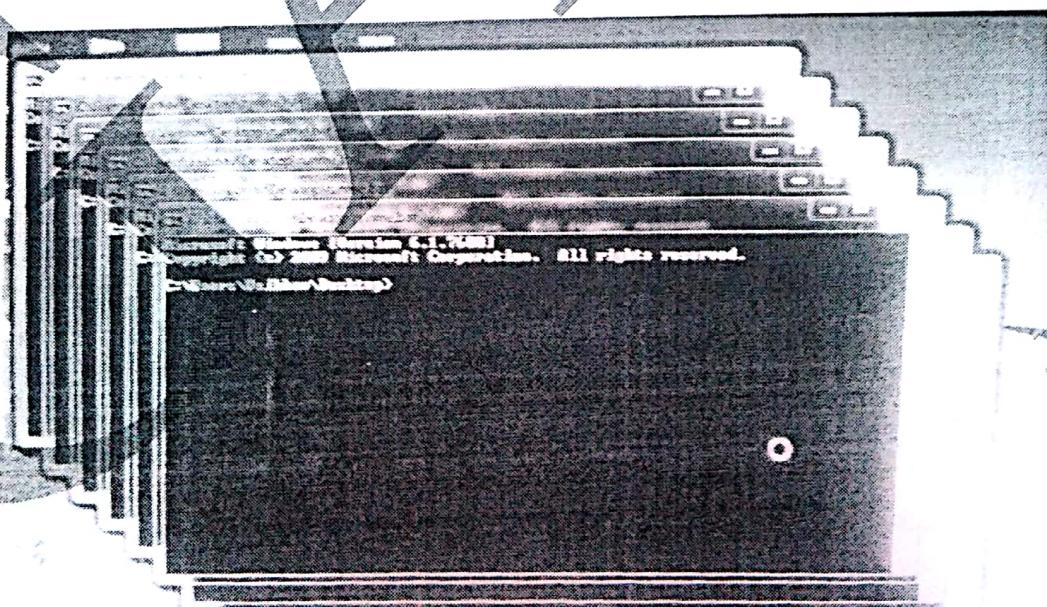


run.bat

3_ نقوم بتشغيل الملف ونشاهد الكوارث ...

ملاحظة : الامتداد .bat هو اختصار ل batch file (ملف دفعي) ماذا يعني ???

بعد تشغيل الملف السحري يجب أن نشاهد شيء يشبه هذا :



BOOM!

إنهاء الإجرائية Process Termination

كما قلنا سابقاً ، الإجرائية الأب parent قد تنتظر الإجرائية الابن child ان تنتهي عن terminate . طريق استدعاء الإجرائية الأب للتابع (system call) wait (استدعاء نظام

· استدعاء النظام (wait بالتفصيل :

عند استدعاء $\Leftarrow \text{call}$

يقوم بإيقاف block الإجرائية المستدعاة إلى حين انتهاء أحد أبنائها (الابن سيستدعي exit).

$\Leftarrow \text{parameter واحداً}$

وهو عبارة عن عنوان address متتحول صحيح int (مؤشر pointer) وفائدة هذا الوسيط أنه يتم تخزين أي معلومات information قد تمرر تصف حالة الإنتهاء الخاصة بالابن child .

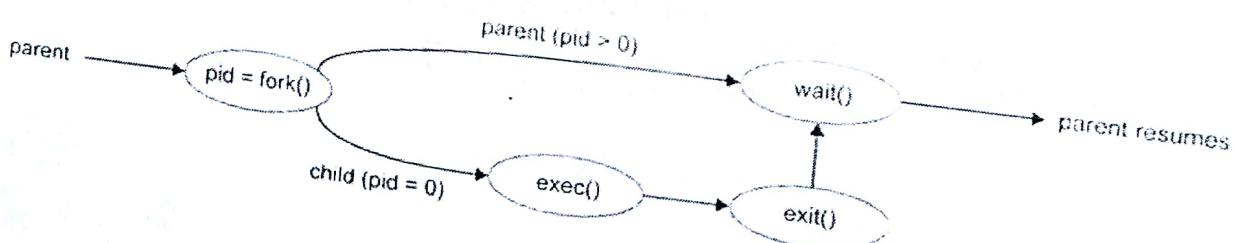
$\Leftarrow \text{return} \text{ يرد هذا الاستدعاء}$

رقم الإجرائية process ID الابن الذي انتهت (exit)

ملاحظة : يجب بشكل عام على كل إجرائية قامت بإنشاء

أبناء لها (من المحب) أن تكون التعليمية الأخيرة (قبل الخروج)

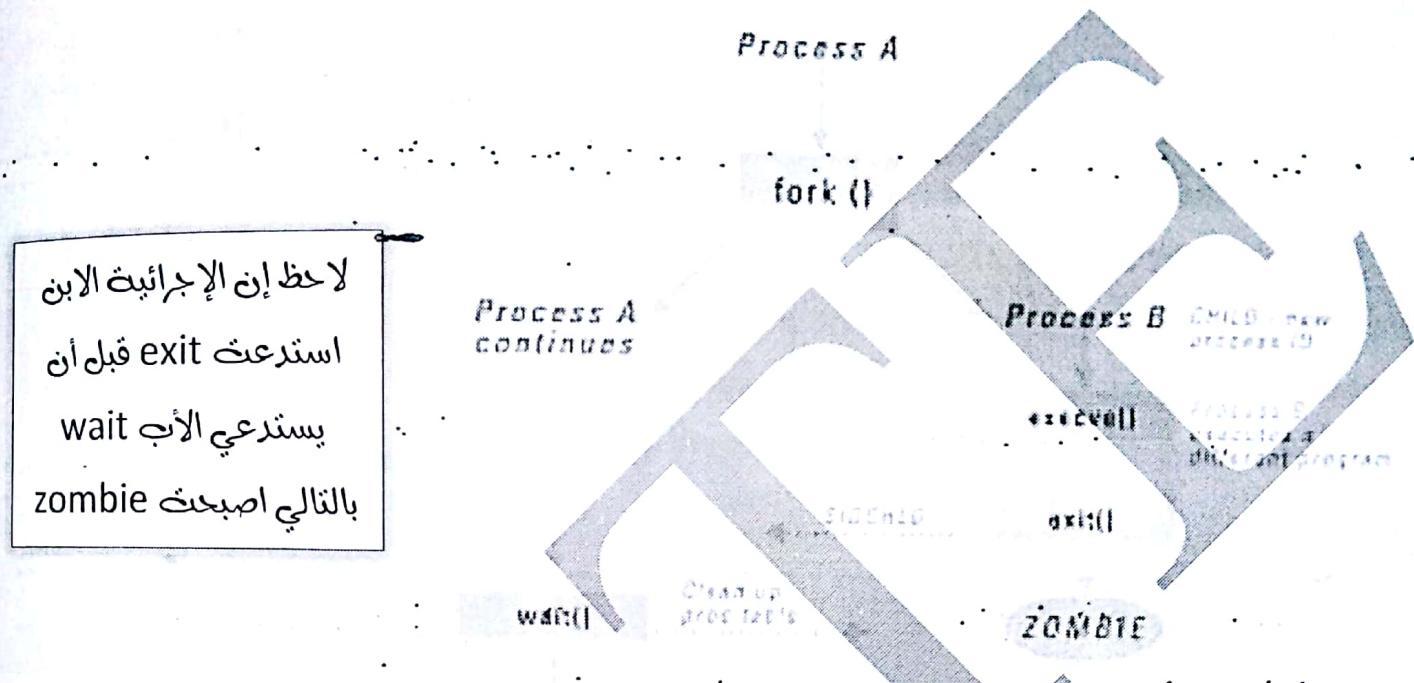
هي الاستدعاء wait (وذلك لكي تنتظر أبنائها).



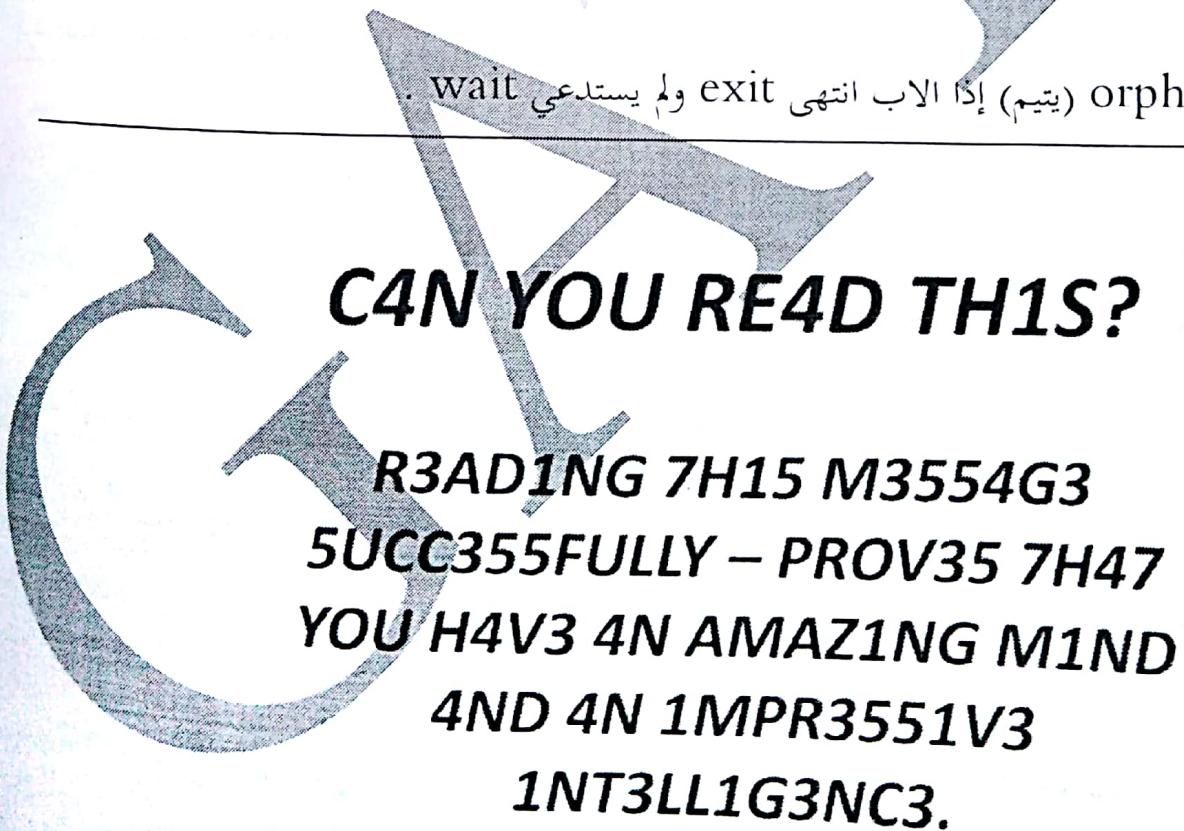
نظم تشغيل المحاضرة الرابعة (fork())

بالنسبة للإجرائية الابن child نصادف حالتين :

1 _ تكون zombie (مومياء) إذا لم يكن الاب ينتظرا wait (لم يستدعى)



2 _ تكون orphaned (يتيم) إذا انتهى wait ولم يستدعي exit



نظم تشغيل المحاضرة الرابعة (fork())



تمرين هام جداً :

ليكن لدينا البرنامج الآتي :

```
#include <stdio.h>
#include <sys/types.h>
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

ان التابع printf موجود في لغة C
وهو النسخة القديمة من cout

المطلوب :

-1 ما هو الخرج المتوقع عند تنفيذ البرنامج .

-2 ارسم شجرة الإجرائيات مع توضيح ما هي الإجرائيات التي تم توليدها من أجل كل استدعاء

ل التابع fork()

الحل :

الطلب الأول :

كما اتفقنا سابقاً فإن عدد الإجرائيات الكلية يساوي 2^n حيث ان $n=3$ (عدد مرات استدعاء التابع fork()) وبالتالي بعد تنفيذ التعليمات الثلاثة والوصول إلى تعلية الطباعة printf سوف يكون لدينا 8 إجرائيات (بما فيهم إجرائية الأب) سوف تنتهي هذه التعلية (تعلية الطباعة) وبالتالي سوف يتم طباعة الكلمة hello على الشاشة 8 مرات .

((إن لم تصدق جرب على نظام تشغيل linux يلي اسا ما ثبتو هلا))

نظم تشغيل المحاضرة الرابعة (fork())

الطلب الثاني :

على اعتبار أن الإجرائية الرئيسية (الأب) اسمها P0 .

أول استدعاء للتابع fork() : سوف يفتح لدينا P0 (نفسها) و P1 .

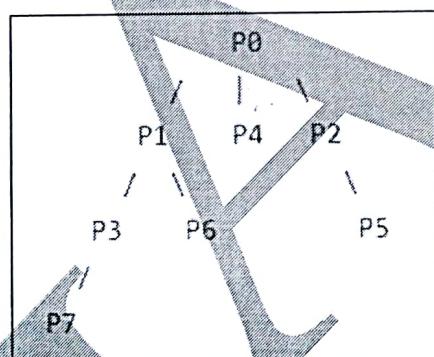
Processes created by the 1st fork: P1

ثاني استدعاء : لدينا P0 سوف تنتج P0 و P2 وكان لدينا P1 قد ولدت منذ قليل وبالتالي فإن P1 سوف تنتج P1 و P3 .

Processes created by the 2nd fork: P2, P3

ثالث استدعاء : لدينا P0 سوف تنتج P0 و P4 وكان لدينا P2 سوف تنتج P2 و P5 وكان لدينا أيضاً P1 سوف تنتج P1 و P6 ولا تنسب P3 بحسب P3 و P7 .

Processes created by the 3rd fork: P4, P5, P6, P7



قد يختلف ترتيب الإجراءات لكن
الشكل يجب أن يبقى نفسه

Special thanks to Ahmad Mokayes

انتهت المحاضرة

بالتوفيق



Mohammed Moulla