المرحلة الثانية: المرور إلى النمط المحمي

في المرحلة السابقة قمنا بتحميل نواة مبسطة إلى الذاكرة، النواة في حد ذاتها لم تكن تقوم بالشيء الكثير، لكن المهم كان هو توضيح كيفية تحميل البرنامج و من ثم تسليمه مقاليد الأمور.

في هدا الدرس سنقوم بخطوة أخرى إلى الأمام، سنمر إلى ما يعرف بالوضع المحمي للمعالج. هدا الوضع هو المستعمل حاليا في أنظمة التشغيل الحديثة. وبدونه، لا أعتقد انه يمكنك تحقيق شيء يذكر في نظام تشغيلك مستقبلا .).

حسنا، قبل أن نمر إلى كتابة الكود، لا بد طبعا من الإجابة على سؤال ما هو الوضع المحمي؟ الشيء الذي يقودنا إلى تسليط الضوء على طريقة عمل المعالجات x86.

تذكير بطريقة عمل الحاسب الشخصي:

مكونات الحاسب الأساسية:

- المعالج
- الفروع : مثبتة على البطاقة الأم سواء مباشرة أو عبر بطاقات الامتداد
- البطاقة الأم

يتحكم المعالج في جميع العمليات الأساسية للجهاز بإرسال/استقبال 1- إشارات التحكم، 2- عناوين الذاكرة 3- بيانات.

هذه الإشارات تمر عبر شبكة إلكترونية مطبوعة على البطاقة الأم. تعرف هذه الشبكة بالناقلة. هدف هذه الأخيرة هو توصيل مختلف الرقاقات المثبتة على البطاقة الأم ببعضها البعض. وهي متصلة أيضا بالناقلات الأخرى مثل PCI و ISA.

على امتداد الناقلة، توجد البوابات الفرعية: هذه الأخيرة عبارة عن معابر تربط مختلف الذاكرات مع الفروع دائما عن طريق الناقلة.

كل فرع مثبت يمتلك رقاقة خاصة تسمى رقاقة التحكم، تكو ن هي الوسيط بين المعالج و الهاردوير.

مثال: عندما تقوم بالضغط على زر في لوحة المفاتيح، يقوم الهاردوير بإرسال إشارات كهربائية، تقوم رقاقة التحكم الخاصة بلوحة المفاتيح بتحويل الإشارة إلى رمز من 8 بيت يمثل الزر المضغوط.

تقسم خطوط الناقلة حسب وظائفها إلى أربعة أنواع:

1- خطوط لتوصيل الطاقة
2- خطوط لتوصيل إشارات التحكم: إشارات الانقطاع – إشارات المنبه
3- خطوط لتوصيل العناوين: عناوين الذاكرة أو البوابات الفرعية.
4- خطوط لتوصيل البيانات

ما يهمنا هنا هي الخطوط 3 و 4.

عندما يريد المعالج كتابة بيانات في موقع ما من الذاكرة، يقوم بإرسال عنوان الموقع على ناقلة العناوين (أي الخطوط المخصصة لنقل العناوين) بعدها مباشرة يرسل البيانات على ناقلة البيانات.

يحدد عدد الخطوط المخصصة لنفل العناوين مدى العناوين التي يمكن أن يلج إليها المعالج، هذا المدى هو ما يعرف بفضاء العنونة.

مهم جدا: فضاء العنونة ليس هو حجم الذاكرة الموجودة فعليا على الجهاز، بل هو يشكل المجال الذي يشمل كل العناوين الممكن إرسالها على ناقلة العناوين.

فمثلا، إذا تم رصد 32 خطا لنقل العناوين، و مع العلم أن كل خط يحتمل إما 1 أو 0، باستعمال الترميز الثنائي يمكننا كتابة 2^{32} عنوانا مختلفا = 4294967296 بايت (كل موقع في الذاكرة يمثل بايتا واحدا) أي يمكن للمعالج عنونة 4 جيغابايت.

من جهة أخرى، يحدد عدد الخطوط المخصصة لنقل البيانات حجم البيانات التي يمكن للمعالج إرسالها أو استرجاعها في نفس الآن (أي بدون إعادة إرسال إشارة أخرى على ناقلة العناوين).

- 8 خطوط: بايت واحد (Byte)
- 16 خطا: كلمة فردية (Word)
- 32 خطا: كلمة مزدوجة (Double word or dword)

الذاكرة: عنصر سلبي في الحاسب لا يتدخل و لا يغير مسار الإشارات بل يكتفي يتخزينها و إرجاعها عند الطلب.

من وجهة نظر الهاردوير، الذاكرة مجرد رقاقات مثبتة على البطاقة الأم وتستعمل تقنيات مختلفة للتخزين، في الحاسبات الشخصية نجد عادة:

1- ذاكرة حية تتميز بقدرتها على تخزين كميات كبيرة من البيانات وسرعتها الكبيرة في الولوج إلى هذه الأخيرة، لكن البيانات تنمحي حال انقطاع التوصيل الكهربائي عن الذاكرة.
2- ذاكرة ميتة: لها سعة بيانات و سرعة ولوج أقل لكنها تحتفظ بالبيانات حتى في حال وقف إمدادها بالطاقة. في الحاسبات الحديثة غالبا ما تستعمل ذاكرات من نوع فلاش للذاكرة الميتة.

من وجهة نظر المبرمج، الذاكرة هي مجموعة من الخانات المتتابعة، كل خانة بسعة بايت واحد، و تحدد بعنوان وحيد، الخانة رقم 0 تلها الخانة رقم 1 وهكذا...

في الحاسبات الشخصية عادة توضع الذاكرة الميتة في أعلى أول ميغابايت، مما يجعلها تحجب مواقع الذاكرة الحية في هذا المجال. سبب ذلك أن المعالج معد عند الانطلاق لتنفيذ التعليمة الموجودة على الموقع 0xFFF0.

كيف يتواصل x86: هناك ثلاث سبل يتواصل بها الx86 مع باقي مكونات الحاسب
1- عبر الذاكرة: بغرض قراءة/كتابة البيانات، يتم الولوج إلى الذاكرة إما عن طريق مسجلات المعالج أو عبر استعمال الولوج المباشر للذاكرة Direct Memory Access (DMA). يمكن أيضا استعمال مواقع الذاكرة للتواصل مع الفروع وذلك عن طريق التحويل المباشر لما يكتب على هذه المواقع إلى تلك الفروع، تعرف هذه التقنية ب

.Memory mapped I/O

2- عبر البوابات الفرعية: هذه هي الطريقة الأصلية المتبعة في الحاسبات الأصلية و تستعمل للتواصل مع رقاقات التحكم(تذكر، تلك المكونات التي تقوم بدور المترجم بين المعالج و الهاردوير). كل بوابة فرعية محددة بعدد من حرفين أو 16بيتس 16bits، مما يعني مجالا من 0 إلى 65535. كما أن كل رقاقة يمكنها استعمال أكثر من بوابة فرعية. تعرف هذه التقنية ب Port mapped I/O.

ملحوظة مهمة: عناوين البوابات الفرعية لا علاقة لها ب بعناوين الذاكرة.
البوابة رقم 0x3D0 لا تعني بأية حال العنوان 0x3D0 في الذاكرة.

3- عبر الانقطاعات: تستعمل لتبليغ الحاسب بوقوع حدث ما مثل ضغطة زر – تكة المنبه...
يمكن أن تأتي من عدة مصادر: الفروع، عبر تعليمة خاصة int، أو يمكن أن تنتج عن خطأ رصده المعالج أثناء القيام بعملية داخلية فيعلن ما يسمى استثناء.

كيف يخزن x86 البيانات: عندما يطلب من المعالج تخزين كلمة(مثلا 1234) على الذاكرة فإنه يقوم بإرسال البايت الأيمن- أو البايت الأضعف- (34) متبوعا البايت الأيسر -أو البايت الأقوى-(12). ينتج عن هذا أن الكلمة المخزنة ككل في موقع ما تكون في الواقع مقلوبة، كما لو نظر إليها عبر مرآة

…	…	…	34	12	…	…	…	…	…
0	1	3	4	5	6	7	8	9	10

الشكل 2 - 1

عكسيا، عندما يطلب من المعالج تحميلها إلى المسجل ax مثلا فإنه يسترجع 34 أولا ويضعها في أقصى اليمين متبوعة ب 12 لتوضع في أقصى اليسار لتشكل في النهاية الكلمة الأصلية 1234.
بالنسبة للمزدوجات (Double words) نفس الشيء، مثلا المزدوجة 12345678 تكون في الذاكرة على شكل 12-34-56-78
بالنسبة لك كمبرمج لا تأثير لهذه الطريقة على شفرتك ما دمت تقرأ البيانات كما كتبتها على الذاكرة. لكن في حالة ما خزنت بيانات على شكل كلمة مثلا و أردت استرجاعها بايتا ببايت عليك أن تقوم بإعادة ترتيب الحرفين في الوضع الصحيح بنفسك.
ملحوظة: التخزين المر آوي يخص البيانات التي في الذاكرة فقط وليس تلك التي في المسجلات.

2- أوضاع العنونة في x86 – Adressing modes

ما هو وضع العنونة: عندما نريد الولوج إلى موقع ما في الذاكرة، فإننا نحدد قيمة رقمية يستعملها المعالج من أجل تحديد هذا الموقع. لكن كيف يقوم المعالج بتأويل القيمة التي نعطيها له؟ ذلك هو ما يحدده وضع العنونة.

في معالجات ل x86 يمكن تحديد ثلاث أوضاع للعنونة:

1- الوضع الحقيقي (Real mode): الوضع الأصلي الذي ظهر مع المعالجات القديمة 8088 و 8086
2- الوضع المحمي (Protected mode): ظهر لأول مرة مع معالجات 80286
3- الوضع الافتراضي لـ8086 (V8086): ظهر مع 80386 لتمكين أنظمة التشغيل

من تنفيذ برامج 16 بيت جنبا إلى جنب مع برامج الوضع المحمي.

في هذا الدرس سنتحدث عن الوضعين 1 و 2.

الوضع الحقيقي: كما قلنا ظهر مع معالجات 8086. هذا الأخير كان يتوفر آنذاك على ناقلة للعناوين سعة 20 بيت، مما يمكنه من عنونة 2^{20} بيت = 1 ميغابايت. المشكلة المطروحة آنذاك أن مسجلات 8086 لها سعة 16 بيت، الشيء الذي يجعل عدد القيم الممكن وضعها فيها 2^{16}=64 كيلوبايت. تذكر أن كل ولوج للذاكرة يتم عبر المسجلات. في هذه الحال كيف يمكننا الولوج إلى المواقع الموجودة وراء الـ64 كيلوبايت؟

الحل الذي استعمل آنذاك هو ما يعرف بتقنية التقسيم. يتم اعتبار الذاكرة على شكل قطاعات متواصلة، كل قسم من سعة 64 كب (كيلوبايت).

قسم 1	قسم 2	...	...	قسم ن

الشكل 2 - 2

إذا تم وضع أن كل قسم عليه البدء من موقع قيمة عنوانه قابلة للقسمة على 16. فذلك يجعل الأربع بيتات في أقصى اليمين دائما 0.

لنأخذ أي قيمة قابلة للقسمة على16: مثلا 213328، لنكتبها بالترميز السداعشري: 0x3415**0**. لاحظ الصفر في أقصى اليمين، لنكتب الآن بالترميز الثنائي: 10100110101**0000**. وهذا صالح مع جميع القيم التي تقبل القسمة على 16.

نحن لا نستطيع أن نعطي القيمة 0x34150 مباشرة للمعالج لأنها تفوق أقصى قيمة يمكن لمسجل 16 بيت حملها وهي 0xFFFF. لكن بما أننا نعرف مسبقا أن أضعف أربع بيتات تساوي صفر يمكننا التغاضي عنها ببساطة وكتابة الرقم على شكل 0x3415 ثم ترك المعالج يقوم بعد ذلك بزيادة الصفر. هذه القيمة بعد وضعها في أحد المسجلات الخاصة بالأقسام تحدد العنوان الفعلي الذي يبتدأ القسم منه في الذاكرة.

الآن يمكننا وضع قيمة أخرى على مسجل 16 بيت: مثلا 0x55 لتحديد موقع نسبي ابتداء من بداية القسم، هذه الأخيرة تعطينا البعد داخل القسم (Offset). ويكتب العنوان كاملا على الشكل 0x3415:0x55

يسمى هذا الشكل (البعد:القسم) عنوانا مقسما (Segmented adress).

والآن لحساب العنوان الخطي (The linear adress)، يقوم المعالج بضرب قيمة القسم في 16 (بالترميز الثنائي هذا يكافئ الضرب في 2^4 أو إزاحة العدد 4 بيتات إلى اليسار)، بعد ذلك يضيف البعد ليحصل على قيمة من سعة 20 بيت يمكنه وضعها على ناقلة العناوين.

مثال: 0x3415:0x0055 للحصول على العنوان الخطي

```
3415 -> 34150
     +    55
     = 341A5
```

بهذه الحيلة يمكن يمكن لـ8086 الولوج إلى كامل فضاء الـ1 ميغابايت.

ملاحظة صغيرة هنا، بإمكان تحديد عنوان واحد بعدة طرق وذلك بتغيير قيمتي القسم و

البعد معا. فمثلا يمكن كتابة العنوان الخطي 341A5 أعلاه أيضا بالصيغة 341A:0005.

الوضع المحمي: ظهر لأول مرة مع 80286 واستمر مع ظهور المعالجات من سعة 32 بيت. سنتحدث هنا انطلاقا من 80386 و هو أول معالج 32 بيت.

في 80386، و أيضا في المعالجات التي ظهرت بعده، المسجلات و ناقلة العناوين معا من سعة 32 بيت (معالجات بنتيوم لها ناقلة عناوين من سعة 36 بيت لكن استخدامها كاملة يتطلب إعدادا خاصا). يمكن بذلك عنونة فضاء حجمه 2^{32} بيت = 4 جيغابايت. و يمكن تحديد هذه القيمة مباشرة على أحد المسجلات 32 بيت.

إذن هل يعني هذا أنه لا حاجة بعد للتقسيم؟

في الواقع، التقسيم سيستمر لكن هذه المرة ليس بغرض تمديد فضاء العنونة.

قلنا في الوضع الحقيقي العناوين تكتب على شكل **البعد**(16ب) : **القسم**(16ب).

في الوضع المحمي تصبح **البعد**(32ب) : **محدد-القسم**(16بيت).

لاحظ محدد القسم (Segment selector) وليس عنوان القسم، هذا لأن القيمة المذكورة لا تعطينا عنوان القسم مباشرة بل تعمل كمؤشر (index) على القيمة الحقيقية للعنوان و التي تكون مخزونة في مكان آخر.

لماذا يحدد العنوان بطريقة غير مباشرة؟ الجواب له علاقة بالوظائف الجديدة التي أدخلت مع 80286 و 80386. فبالإضافة لتقسيم الذاكرة، يمكننا المعالج من تحديد خصائص إضافية لقسم ما: مثل نوع المعلومات المخزنة به، سواء كود أو مجرد بيانات، أيضا يمكن تحديد حقوق للولوج إلى المواقع داخل القسم ... كل هذه الخصائص مع عنوان بداية القسم و طوله في الذاكرة تخزن في بنية معطيات (data structure) تسمى موصف القسم (segment descriptor). تجمع موصفات الأقسام في مكان في الذاكرة لتشكل جدولا يعرف بجدول الموصفات العام (Global descriptor table). لاحقا سنقول اختصارا ج.م.ع. (GDT).

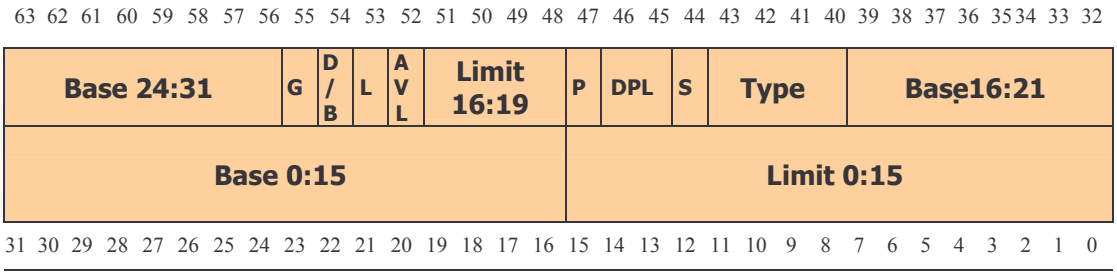
الموصف الأول	المدخل 0 --->
الموصف الثاني	المدخل 1 --->
...	...
...	
..	
...	
الموصف ن	المدخل ن-1 --->

الشكل 2 - 3

إذن، موصفات الأقسام تشكل مداخل في ج.م.ع. كل موصف من حجم 64 بيت. و محدد القسم يعمل كمؤشر على رتبة المدخل في هذا الجدول. كيف يتم تحديد الموقع الفعلي؟ يستعمل المعالج محدد القسم لمعرفة مكان الموصف في ج.م.ع. بعد

ذلك يقوم باستخراج عنوان البداية من الموصف ليضيف عليه قيمة البعد من حجم 32 بيت (انتبه، لا يوجد ضرب في 16). الحاصل يعطينا العنوان الخطي في الذاكرة.

والآن لننظر أكثر إلى محتوى موصف القسم:



الشكل 2 - 4

فيما يلي معنى الحقول المكونة لها:

Limit: يحدد طول القسم كاملا على 20 بيت و لأسباب ربما تتعلق بتحسين الأداء تم تفريق بيتات القيمة على جزأين متفرقين في ال64 بيت المشكلة للموصف، الجزء الأول من 0 إلى 15 والجزء الثاني من 48 إلى 51. نعرف أن أقصى قيمة ل 2^{20} هي 4 ميغابيت. لكن بوضع البيت G (البيت رقم 63) على1، يصبح الطول محسوبا لا بالبايتات بل بوحدات من 4كيلوبايت. مما يعني أن أقصى قيمة ممكنة هي 4 جيغابايت. بذلك يمكن صنع أقسام على 4 جيغابايت.

Base: يحدد العنوان كاملا على 32 بيتا و تم تفريقه أيضا، كما يرى من الشكل أعلاه، إلى ثلاث أجزاء موزعة على طول الموصف، الجزء الأول من 16 إلى 31، الجزء الثاني من 32 إلى 39 والجزء الثالث من 56 إلى 63.

Type: نوع القسم التأويل يختلف حسب بعض القيم الأخرى من أجل تبسيط الأمور يمكنك حاليا وضع القيمة 1011 بالنسبة لأقسام الشفرة و 0011 بالنسبة لأقسام البيانات و الكومة.

S- System: 0 تعني موصف لقسم نظام : هذا النوع من الأقسام يستعمل لأغراض أخرى سنتحدث عنها في دروس قادمة إن شاء الله، في الوقت الحالي يمكن وضع 1 مما يعني أنه موصف لقسم عادي أي شفرة أو بيانات.

DPL – Descriptor privilege level: يحدد درجة الأفضلية المطبقة على القسم، 0 تعني أعلى درجة بينما 3 تعني أضعف درجة. إذا حاول برنامج يعمل في درجة أفضلية أضعف الولوج إلى موقع داخل هذا القسم فإن المعالج يرفع ما يسمى استثناء من نوع General Protection fault. ببساطة يخبرنا بالأمر ويمنع البرنامج المتسبب في الخطأ من الولوج إلى الموقع. توظف هذه الميزة عادة لحماية الذاكرة التي تعمل فيها النواة من البرامج الأخرى.

P - Present : 1 تعني أن القسم موجود حاليا في الذاكرة بينما 0 تعني العكس. يمكنك وضع 1.

AVL - Available: متروك لنا لنضع فيه ما نشاء.

D/B: يختلف التأويل حسب نوع القسم، باختصار نضع 1 في قسم يشمل شفرة أو بيانات من سعة 32 بيت (1 في الوضع المحمي) و 0 بالنسبة ل 16 بيت.

G - Granularity: كما قلنا سابقا تحدد كيف يتم حساب القيمة الموجودة في طول القسم. 1 تعني أن الطول يحسب بوحدات 4 كيلوبايت و 0 تعني وحدات 1 بايت.

الآن يمكنك التقاط أنفاسك قليلا ☺.

تمرين بسيط: إصنع موصفا لقسم يحتوي على شفرة 32 بيت، يبدأ من 0 و طوله 4 جيغابايت، يتوفر على أعلى درجة حماية 0.

الحل:

Base 24:31	G	D / B	-	AVL	Lim 16:19	P	DPL	S	Type	Base 16:23
0	1	1	0	1	1111	1	00	1	1011	0
0						0xFFFF				
Base 0:15						Limit 0:15				

الشكل 2 - 5

أولا لنستخرج طول القسم، البايتين الأولين 0xFFFF ثم نضع إلى جانبهما البيتات 16 إلى 19 وهي 1111. بالسداعشري تساوي F نضيفها إلى ما سبق فنحصل على 0xFFFFF. الآن لننظر إلى قيمة البيت G. تساوي 1 مما يعني الطول يحسب بوحدات 4 كيلوبايت. حاصل العملية يخبرنا أن طول القسم هو 4 جيغابايت.

لصنع هذا الموصف في Nasm نستعمل التعليمة التالية:

```
0xFF, 0xFF,  0,0,     0,  1 00 1 1011b,     1 1 0 1 1111b,    0
```

حسنا، db لمن لا يعلم تصريح بأن ما يلي واحد أو مجموعة بايتات أي قيم على 8 بيت. الفواصل تفصل كل بايت عن الآخرين. كما ترى لدينا 8 بايتات مما يعطي في المجموع 64 بيتا (8*8). القيم التي تبدأ ب *0x* قيم سداعشرية بينما القيم التي تنتهي ب *b* قيم ثنائية، استعملت هنا لتسهيل قراءة حقول الموصف بوضوح.

و الآن كيف يتم المرور إلى الوضع المحمي:

أولا: يتم بناء ج.م.ع و إخبار المعالج عن موقعه أين يبدأ و أين ينتهي. يتم ذلك عن طريق بنية أخرى صغيرة مشكلة كالتالي

طول ج.م.ع	بداية ج.م.ع
0	16 48

الشكل 2 - 6

البيتات من 0 إلى 15: طول ج.م.ع في الذاكرة ناقص واحد.
البيتات من 16 إلى 48 : **العنوان الخطي** لبداية ج.م.ع في الذاكرة.

بعد بناء هذه البنية يجب تحميلها إلى المسجل
GDTR (Global Descriptor Table Register). يتم ذلك عبر التعليمة التالية

```
lgdt  gdt
```

حيث gdt هو عنوان البنية التي فوق.

ثانيا: يتم وضع 1 في البيت الأضعف (يسمى هذا البيت PE) في المسجل CR0.
مثال:

```
mov eax, cr0
or al, 1
mov cr0, eax
jmp next        ; short jump to flush the prefetch queue

next:
    ...
```

التعليمة الأخيرة jmp next ضرورية، ذلك أن المعالج من أجل رفع سرعة أدائه يقوم بتحميل عدة تعليمات في ذاكرة داخلية بينما لا تزال التعليمة الحالية قيد التنفيذ. بما أننا غيرنا الوضع فيجب إخلاء هذه الذاكرة و إجبار المعالج على إعادة تحميل التعليمات الموالية و تأويلها بالطريقة الصحيحة وفقا للوضع الجديد.

بعد هذه المقدمة النظرية أصبحنا نعرف ما يكفي لكتابة الشفرة.

فيما يلي النص الكامل للشفرة، كالعادة التغييرات المضافة بالخط البارز، عليك إزاحة السطر بالأحمر

```
[BITS 16]
[ORG 0x0]

%define DESC_ENTRIES 3       ; number of entries in GDT
%define DESC_LENGTH 8        ; entry length in GDT
%define LOAD_ADDR 0x7C00     ;   boot sector load adress

start:
        mov ax, 0x07c0
        mov ds, ax
        mov es, ax
        mov ax, 0x9000
        mov ss, ax
        mov sp, 0xffff

        mov si, msg
        call prints

        xor ax, ax
        int 0x13

        mov ax, 0x100   ; memory location to load data
        mov es, ax
        mov bx, 0
        mov dl, 0
        mov dh, 0  ; head
        mov ch, 0 ; track
        mov cl, 2   ; sector
        mov al, 1  ; number of sectors to read
        mov ah, 2
        int 0x13
```

```
   ;;          jmp 0x100:0          ;; you must remove this line if you use the older source

setup_gdt:
          cli
          lgdt[gdtptr]
          mov eax, cr0
          or ax, 1
          mov cr0, eax

          jmp next      ; jmp to clear the prefetch queu

next:
          mov ax,0x10                  ; update segment registers
          mov ds,ax
          mov fs,ax
          mov gs,ax
          mov es,ax
          mov ss,ax
          mov esp,0x9F000

          jmp dword 0x8:0x1000   ; long jump (to update cs) to kernel code
          jmp $


prints:
          push ax
          push bx
   .debut
          lodsb
          cmp al, 0
          jz .fin
          mov ah, 0x0e
          mov bx, 0x07
          int 0x10
          inc bx
          jmp .debut

   .fin:
          pop bx
          pop ax
          ret

;-----GDT table descriptors---------
gdt:
gdt_null:
          dw 0,0,0,0      ; first entry must be always null
gdt_cs:
          db 0xFF, 0xFF,  0,0,   0,     10011011b,     11011111b,     0
gdt_ds:
          db 0xFF, 0xFF,  0,0,    0,     10010011b,    11011111b,     0
gdtend:

gdtptr:
          dw        (DESC_ENTRIES  * DESC_LENGTH)-1         ; limite =( number_entries x
entry_length)-1
          dd        LOAD_ADDR +gdt                    ; base, linear address = load adress + offset

;-------------------------------------------------------------
msg db "Loading the kernel",13,10,0

;-------------------------------------------------------------

times 510-($-$$) db 144
dw 0xAA55
```

الآن لنتمعن في التعليمات الجديدة، لنبدأ من الأسفل حيث نقوم بتشكيل البنيات المطلوبة للمرور إلى الوضع المحمي.

```
gdt:
gdt_null:
        dw 0,0,0,0
gdt_cs:
        db 0xFF, 0xFF,  0,0,   0,    10011011b,    11011111b,    0
gdt_ds:
        db 0xFF, 0xFF,  0,0,    0,    10010011b,    11011111b,    0
gdtend:
```

هنا نقوم بإعداد ج.م.ع من ثلاث مداخل

- المدخل الأول كله 0 تبعا لتعليمات كتاب الأنتل (The intel manual).

- المدخل الثاني مخصص لقسم الشفرة، يبدأ من 0 و طوله 4 جيغابايت، درجة أفضليته 0.

- المدخل الثالث مخصص لقسم البيانات (و أيضا للكومة the stack)، يبدأ من 0 و طوله 4 جيغابايت، درجة أفضليته 0.

كتمرين، أترك القارئ لمطابقة الأرقام التي في الشفرة مع التوصيفات التي في الشكل 2-5

```
gdtptr:
        dw      (DESC_ENTRIES  * DESC_LENGTH)-1         ; limite =( number_entries x
entry_length)-1
        dd      LOAD_ADDR +gdt                  ; base, linear address = load adress + offset
```

هنا نقوم بإعداد البنية التي سيتم تحميلها إلى مسجل GDTR. الحقل الأول عبارة عن كلمة مزدوجة لتحديد طول ج.م.ع. نحسبها هنا بعملية بسيطة: طول الموصف الواحد مضروب في عدد الموصفات. ثم نطرح منها واحد دائما تبعا لتعليمات أنتل. الحقل الثاني هو **العنوان الخطي** لبداية ج.م.ع. بما أننا لا زلنا في الوضع الحقيقي فإننا نضيف على البعد gdt عنوان القسم مضروبا في 16: 0x7C0*0x10=0x7C00+gdt.

بعد إعداد البيانات اللازمة نقوم بتحميل المسجل GDTR بالبنية أعلاه

```
cli
lgdt    [gdtptr]

mov eax, cr0
or ax, 1
mov cr0, eax
jmp next

next:
```

أولا نقوم بتعطيل الانقطاعات عبر التعليمة cli، ثم نحمل المسجل GDTR بالتعليمة lgdt. للمرور إلى الوضع المحمي نقوم وضع 1 في أضعف بيت في المسجل CR0 (the PE flag). بعد ذلك نقوم بقفزة قصيرة لإخلاء الذاكرة الداخلية للمعالج.

الآن نحن في الوضع المحمي، لكن بقيت أمامنا مهمة أخرى وهي تغيير القيم الموجودة في مسجلات الأقسام لأنها لم تعد صالحة في الوضع المحمي (تذكر أن هذه المسجلات يجب أن تحتوي على محدد للقسم في الوضع المحمي).

```
next:
        mov ax,0x10             ; updating segment registers
        mov ds,ax
        mov fs,ax
        mov gs,ax
        mov es,ax
        mov ss,ax
        mov esp,0x9F000

        jmp dword 0x8:0x1000    ; long jump (to update cs) to kernel code
```

هنا نقوم بإعداد محدد لقسم البيانات 0x10 (التي هي المؤشر على قسم البيانات في ج.م.ع) ثم نقوم بتحميله إلى كل مسجلات أقسام البيانات و مسجل قسم الكومة SS. بعد ذلك نقوم بتغيير مسجل الكومة esp بعنوان آخر لأن بداية القسم SS أصبحت 0.

يبقى لنا مسجل قسم الشفرة CS، لتغييره نقوم بقفزة طويلة إلى بداية النواة فنحقق هدفين في آن واحد: تغيير CS ثم إعطاء اليد لبرنامج النواة.

برنامج النواة

في الوضع المحمي لا يمكننا استعمال انقطاعات البيوس بطريقة مباشرة و سهلة، لذلك من أجل طباعة جملتنا على الشاشة نحن مجبرون على التعامل مباشرة مع بطاقة الفيديو.

لن ندخل هنا في كل تفاصيل برمجة الفيديو، على أمل أن نخصص درسا قادما عن هذا المجال، و إذن كيف سنتمكن من طباعة جملتنا النصية؟

لحسن الحظ، فهذه البطاقات تمنحنا طريقة سهلة لطباعة جمل على الشاشة النصية. عند انطلاق الحاسب يتم تخصيص مجال في الذاكرة يبدأ من الموقع 0xB8000، بحيث أن ما تكتبه في هذا المجال ينعكس مباشرة على الشاشة النصية لبطاقة الفيديو. لكي تكتب حرفا في الشاشة ما عليك سوى وضع الرمز ascii المقابل لهذا الحرف في الذاكرة متبوعا ببايت لوصف لون الكتابة (foreground) و لون الخلفية (background).

الشكل التالي يوضح كيف يكون محتوى ذاكرة الشاشة النصية عندما نطبع جملة "Hello" (7 تعني أن الحرف يطبع بلون أبيض على خلفية سوداء):

....	'H'	7	'E'	7	'L'	7	'L'	7	'L'	7	'O'	7	....
	B8000		B8001		B8002....								

الشكل 2 - 7

تحديد لوني الكتابة والخلفية على بايت واحد مشروح في الجدول التالي:

لون الخلفية				لون الكتابة			
كثافة أو ومض	أحمر	أخضر	أزرق	كثافة	أحمر	أخضر	أزرق
7	6	5	4	3	2	1	0

الشكل 2 - 7

يتم تحديد كل لون بوضع كل بيت إما 1 أو 0، البيتات 4 الأولى خاصة بتحديد لون الكتابة بينما الباقي بتحديد الخلفية.

البيت 7 قد يحدد إما كثافة الخلفية، وفي حال ما إذا كان الومض نشطا، يحدد البيت هل تكون الخلفية وامضة بمعنى أن تظهر و تختفي بانتظام.

وهذا مثال لبعض القيم الممكنة

0000:	أسود
0111:	رمادي
1111:	أبيض
0001:	أزرق
0010:	أخضر
1110:	أصفر
0100:	أحمر

الشاشة التي في الذاكرة تحتمل سعة من 25 سطرا و 80 عمودا كل حرف يشغل عمودا، مما يعني مجموع 80*25=2000 موقع للكتابة، بما أن كل موقع يحدد على 2 بايت (واحد للحرف و آخر للون) فإن سعة الذاكرة تصبح 4000 بايت.

فيما يلي نص الشفرة الخاص بالنواة

```
[BITS 32]
[ORG 0x1000]

%define SCREEN_SIZE 80*25

        mov ebx, 0xB8000
        mov ecx, SCREEN _SIZE

;---------clear the screen by putting ' ' blank char in all the 80*25 ases
clear_screen:
        mov byte [ebx], ' '
        inc ebx
        mov byte [ebx], 0
        inc ebx
        loop clear_screen



prints:
        xor eax, eax
        xor edi, edi
        mov ebx, 0xB8000

begin:
        mov al, [msg+edi]       ; mov the char number edi in msg
        cmp al, 0
        je end                  ; end of msg?
        mov byte [ebx], al      ; for putting a charachter in screen, we must put 1-the char
```

```
ascii code
        inc ebx                 ; and 2- the char color attribute:
        mov byte [ebx], 0x7
        inc ebx
        inc edi
        jmp begin

end:
        jmp end;        ;infinite loop



msg  db "Now, we are in proected mode", 0
```

[Bits 32] - نقوم بإعلام nasm بأننا نشتغل في وضع 32بيت عبر التعليمة.

[ORG 0x1000]: عنوان البدء في الذاكرة حتى تكون العناوين المنتجة من طرف nasm صحيحة.

```
%define SCREEN_SIZE 80*25

mov ebx, 0xB8000
mov ecx, SCREEN _SIZE

;---------clear the screen by putting ' ' blank char in all the 80*25 ases
clear_screen:
        mov byte [ebx], ' '
        inc ebx
        mov byte [ebx], 0
        inc ebx
        loop clear_screen
```

التعليمات التالية تقوم بمحو الشاشة عبر طباعة ' ' (blank char) على كل خانات الشاشة، نستعمل المسجل ecx كعداد مع التعليمة loop، هذه الأخيرة تقوم بتكرار تعليمة المحو عدد المرات الموجود في ecx. في البدء نقوم بجعل ebx يشير إلى بدء الشاشة في الذاكرة. ثم نقوم بزيادته ليشير إلى الخانة الموالية مع كل دورة.

```
prints:
        xor eax, eax
        xor edi, edi
        mov ebx, 0xB8000

begin:
        mov al, [msg+edi]       ; edi is used to idex chars in msg
        cmp al, 0
        je end                  ; end of msg?
        mov byte [ebx], al      ; for putting a charachter in screen, we must put 1-the char
ascii code
        inc ebx                 ; and 2- the char color attribute: 4bits foreground & 4 bits
background
        mov byte [ebx], 0x7     ; 7: foreground=gray – background = black
        inc ebx
        inc edi
        jmp begin
```

```
end:
        jmp end;          ;infinite loop

msg  db "Now, we are in proected mode", 0
```

لطباعة الجملة نستعمل مثل المرة السابقة ebx كمؤشر على الشاشة في الذاكرة، بينما نستعمل msg+edi كمؤشر على الحرف الذي سنطبعه.

للترجمة نستعمل نفس التعليمة

nasm –f bin boot.s -o boot.bin
nasm –f bin kernel.s -o kernel.bin

لنقل الملفات إلى القرص المرن

cat boot.bin kernel.bin | dd of=/dev/fd0 bs=512

بالنسبة لمستعملي ويندوز

copy /b boot.bin+kernel.bin boot.img

لنقل الملف *boot.img* إلى القرص المرن نستعين بالبرنامج *ntrawrite* المرفق مع الدرس

ntrawrite –f boot.img

و كالعادة عند السؤال *please type the diskette drive* عليك الإجابة ب A

إذا أردت كتابة الملف على صورة للقرص

cat boot.bin kernel.bin | dd of=image_file bs=512 conv=notrunc

مع استبدال *image_file* باسم الملف الفعلي.

النتيجة:

إذا تم كل شيء على ما يرام و نجح المعالج في المرور إلى الوضع المحمي، ستطبع على الشاشة في أعلى الجملة المطلوبة.

ختاما، في هذا الدرس ألقينا نظرة على كيفية تواصل المعالج، وطريقة العنونة تبعا للوضع الذي يوجد فيه. رأينا كيف يتم المرور إلى الوضع المحمي و ألقينا نظرة عابرة على كيفية التعامل مع الشاشة النصية.

في الدرس القادم إن شاء الله سنخطو خطوة أو خطوات أخرى إلى الأمام والبرنامج

سيكون غنيا :

- تحميل النواة بواسطة كروب Grub.

- سنطور نواتنا بلغة من أعلى مستوى وهي C++. سنطور كذلك مكتبة صغيرة من العناصر خاصة للتعامل مع الشاشة النصية.

- سنتحدث عن كيفية تدبير تدبير الانقطاعات في الوضع المحمي.

إلى اللقاء.