# الدرس الخامس
# تدبير الذاكرة: مخصص الذاكرة الحقيقية Physical Memory

في الدرس السابق رأينا كيف يمكن تدبير الانقطاعات و الاستثناءات في حاسبات x86. درسنا مثالا للاستثناءات و برمجنا منبه النظام كمثال للانقطاعات الصلبة.

ابتداء من هذا الدرس سنتطرق إلى "الأمور الجادة". و سنبدأ بدراسة كيفية إرساء نظام لتدبير الذاكرة.

نظام تدبير الذاكرة هو في قلب برنامج النواة و أحد أهم مكوناتها بجانب مدبر المهام Task Sheduler. و قبل إرسائه تظل عملية برمجة النواة صعبة لعدم توفر وسيلة لصنع البنيات التي نحتاجها في برامجنا. عند كتابة برامجك بسي لا بد أنك استعملت الدوال malloc و free للتعامل مع مدبر الذاكرة. و إذا برمجت قبلا بسي++ فلا بد أنك تعرف المعاملين new و delete.

ما سنحاول القيام به في هذه المرحلة الجديدة هو توفير مثل هذه الدوال في نواتنا. حيث يتسنى فيما بعد استعمالها من طرف مكونات النواة و في مرحلة لاحقة برامج المستعمل.

الحقيقة هو أن تنجيز دوال من فئة malloc و أخواتها ليس بالسهولة التي يتصورها البعض. إذ أن الأمر لا يقتصر على مجرد كتابة شفرة هذه الدوال. بل يتعداه إلى ما هو أعقد (و سنحاول تبسيطه إن شاء الله): تدبير الذاكرة بين مختلف المهام tasks التي سيتم تنفيذها في الحاسب.

بدون إسهاب، فنظام الذاكرة يعتمد عادة على ثلاث أنظمة متراتبة و مرتبطة بينها كما يوضح الشكل التالي

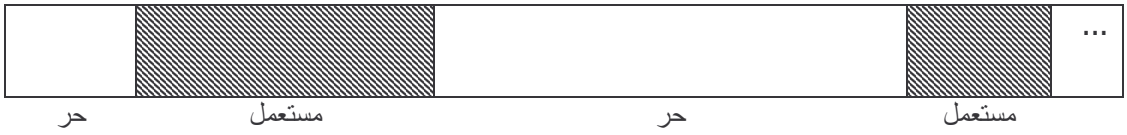

مجموع هذه الأنظمة و تفاعلها فيما بينها يشكل نظام تدبير الذاكرة بأكمله. لاحظ أن النظام مرتب على شكل طبقات، و كل طبقة تعتمد على وجود الطبقة التي أسفلها. فمدبر الذاكرة الافتراضية يحتاج قبلا إلى وجود مدبر الذاكرة الحقيقية. بينما يحتاج النظام الذي يوفر خدمة تدبير الذاكرة لباقي البرامج للمدبر الافتراضي.

في هذا الدرس سنقوم:

أولا بالتحدث عن مشكل تدبير الذاكرة بوجه عام و بعض السياسات المستعملة لمعالجته.
ثم في مرحلة ثانية سننجز مدبر الذاكرة الحقيقية في نواتنا و الذي سيشكل دعامة لباقي الأنظمة التي سنراها في الدروس اللاحقة بإذن الله.

## 1-سياسات تخصيص الذاكرة

يمكن تلخيص تدبير الذاكرة في المشكل التالي: كيفية تخصيص قطاعات من الذاكرة حسب الطلب، مع مراعاة ألا يخصص قطاع أو أي جزء داخل هذا القطاع لأكثر من مرة. مهمة مُخَصّص الذاكرة تنحصر إذن في تذكر القطاعات المستعملة و القطاعات الحرة.



لا بد أنكم لاحظتم استخدام مصطلح "سياسات" بدل "تقنيات" تخصيص الذاكرة. هذا لأنه لا توجد وصفة مُثلى تصلح لجميع الحالات و تحل مشكل الذاكرة بشكل نهائي. و في مجال تدبير الذاكرة بالذات، يكون على المبرمج دائما الحسم بين مجموعة من الخيارات و اختيار الأولويات : مثلا بين التضحية ببعض المساحات الزائدة في الذاكرة من أجل سرعة أفضل في الأداء أو العكس. ما يزيد الأمر صعوبة هو أن تدبير الذاكرة - من الناحية النظرية – موضوع غير محسوم بالمرة، و رغم الأبحاث الكثيرة التي قام بها مختلف الباحثون، و برغم التجارب المعملية المختلفة. إلا أن المقياس الحقيقي لأي حل يبقى هو "العالم الواقعي". و يبقى ملاذ المبرمج في هذه الحالة هو الحلول التي تم تنجيزها في أنظمة حقيقية و أثبتت نجاعتها على أرض الواقع.

مما سبق فإن السؤال الآتي : ما هو أفضل نظام لتدبير الذاكرة؟ سؤال ليس في محله.
السر يكمن في كيفية تفاعل نظام تدبير الذاكرة مع البرامج أو الأنظمة التي تستعمله. هذا الأمر مهم جدا وينبغي أن يؤخذ في الاعتبار من قبل كل من يريد أن يكتب مخصصا للذاكرة: **نجاعة نظام تدبير الذاكرة مرتبطة ارتباطا وثيقا بالوسط الذي يعمل فيه و يتفاعل معه**.

أهمية نظام تدبير الذاكرة في نواة نظام التشغيل تنبع من كثافة استعماله من طرف باقي الأنظمة الأخرى، سواء تلك المدمجة في النواة مثل مدبر المهام Task Scheduler أو برامج المستعمل المثبتة على نظام التشغيل. لذلك فتنجيز مخصص الذاكرة ينبغي أن يكون فعالا من حيث سرعة الأداء و استعمال الذاكرة معا. هذا بالضبط هو التحدي الذي يواجه المبرمجين في أي نظام تشغيل.

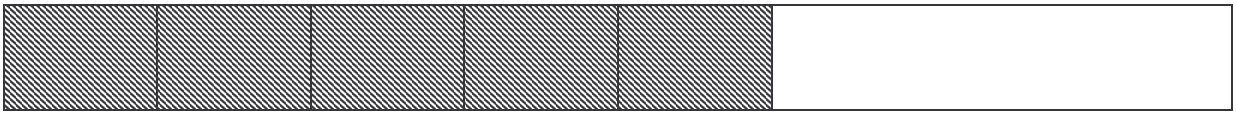
فيما يلي سنتحدث عن نظم تخصيص الذاكرة المعروفة. بعد ذلك سنرى كيف ينجز مخصص الذاكرة الحقيقية.

## 1-1 تعريف بعض المفاهيم:

مثال تمهيدي: لنتخيل أننا نتوفر على مساحة حرة من الذاكرة حجمها 64 بايت كما في الشكل التالي.

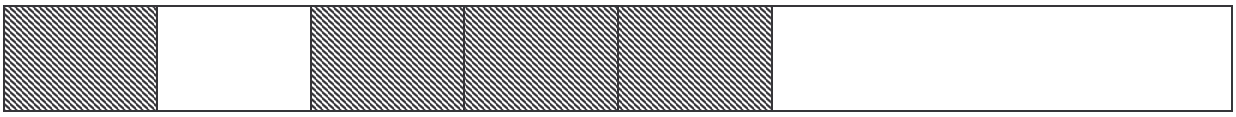
0→ 63

الآن لنمض أكثر في مثالنا و نتخيل 5 طلبات متتالية ل 8 بايت. في مثالنا، يقوم المخصص باقتطاع 8 بايت من المساحة لإرضاء طلب الذاكرة



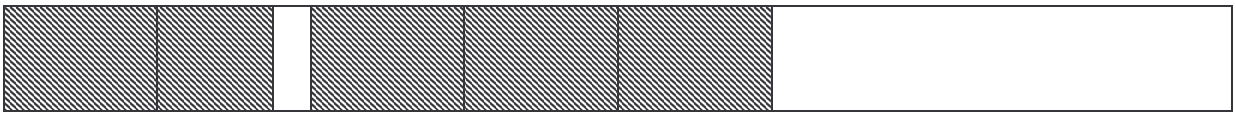
0→

الآن لننظر شكل الذاكرة بعد تحرير القطاع الثاني



0→

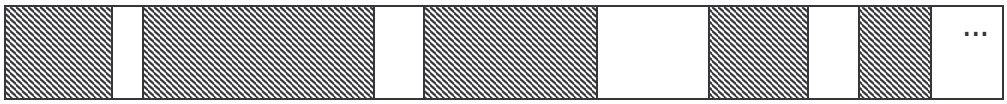
الآن نطلب 6 بايتات من الذاكرة. عندما يقوم المخصص بمسح القطاعات الحرة. سيصادف القطاع الأول بحجم 8 بايت. لكن الحجم المطلوب هو 6 بايت. لنفرض أن المخصص سيقتطع 6 بايت من ال8 بايت الحرة. هذا يعني توفر 2 بايت حرة.




لنتصور ما سيحدث بعد طلب 6 بايتات أخرى: يقوم المخصص بمسح القطاعات الحرة، هناك قطاع حر أول لكن حجمه – 2 بايت - صغير جدا بالمقارنة مع الحجم المطلوب – 6 بايت -. إذن المخصص مجبر على مواصلة البحث حتى العثور على قطاع كبير كفاية. مما ينتج عنه زيادة في وقت البحث. و قد يكون هناك ضياع في الذاكرة في حال كانت طلبات الذاكرة دائما أكبر من 2 بايت. طبعا ضياع الوقت و الذاكرة في مثالنا صغير جدا، لكن تخيل ما يحدث في ذاكرة حجمها 128 ميغابايت و طلبات في الذاكرة بعدد مهول.

المثال السابق سيمكننا من إدخال بعض المفاهيم المهمة:

1- الانقسام : عادة عندما يبدأ مُصَرّف الذاكرة عمله، تكون الذاكرة عبارة عن مساحة فارغة و متصلة. لكن على مر الوقت يقوم المُخَصّص باقتطاع أجزاء منها لأجل إرضاء طلبات الذاكرة (التي تأتي مثلا عبر الدالة malloc في سي). بعد فترات متباينة تقوم البرامج المستفيدة بتحرير القطاعات المستعملة (مثلا عبر الدالة free في سي). في الغالب تكون فترات استعمال قطاعات الذاكرة متفاوتة في البرنامج الواحد وبين مختلف البرامج. هذا يعني أن تخصيص الذاكرة و تحريرها يتم على مدى فترات متباينة. ينتج عما سبق أن القطاعات الحرة تصبح بعد فترة معينة من الاستعمال متفرقة على الذاكرة و بأحجام مختلفة. هذه الظاهرة هي ما يعرف بالانقسام و هي أهم عامل يجب اعتباره في سياسة التخصيص لأنه يؤثر على المدى الطويل في سرعة الأداء نظرا لتراكم قطاعات الذاكرة الحرة و تشتتها. قد يتسبب الانقسام أيضا في ضياع مساحات من الذاكرة (مثلا إثر تراكم قطاعات صغيرة ذات أحجام جد صغيرة كما رأينا في المثال السابق).
2- الانقسام الخارجي: عادة عند تلبية طلب للذاكرة بحجم معين ((malloc(size)، في حالة لم يجد المخصّص قطاعا حرا بالحجم المطلوب، يقوم بتقسيم قطاع بحجم أكبر إلى قسمين، فيقوم بتخصيص قسم لتلبية طلب الذاكرة و يضيف القسم المتبقي إلى قائمة القطاعات الحرة. في حالات معينة قد يكون القسم المتبقي صغيرا إلى درجة استحالة استعماله لتلبية طلبات الذاكرة. تسمى هذه الظاهرة بالانقسام الخارجي.




3- الانقسام الداخلي: في بعض الحالات قد يفضل مخصص الذاكرة ألا يقسم قطاعا بحجم أكبر من الحجم المطلوب بل تخصيصه بأكمله لتلبية طلب الذاكرة (مثلا عند طلب 6 بايت لا تقتطع من الـ8 بايت الحرة بل تخصص هذه الأخيرة بأكملها) على أمل أن يتم تحرير هذا القطاع في وقت وجيز. في هذا الحالة تضيع المساحة الزائدة مع القطاع المخصص. تسمى هذه الظاهرة بالانقسام الداخلي. في الشكل التالي تظهر المربعات المشطوبة المساحات المستعملة فعليا من طرف البرنامج بينما تظهر المربعات الرمادية المساحات المخصصة و الضائعة


الآن كل مشاكل تخصيص الذاكرة (بالأساس) تعود إلى هذين العنصرين: الانقسام الداخلي و الخارجي. إذا اختار المخصص تقسيم القطاعات الحرة لإرضاء الطلب بالحجم المطلوب بدقة يتزايد عدد القطاعات

الحرة الصغيرة في الذاكرة، و يُضَيّع المخصص وقتا أكثر في البحث. في حال اختار عدم تقسيم القطاعات لتجنب تراكم القطاعات الصغيرة (= الانقسام الخارجي) يتناقص وقت البحث لكن تضيع الذاكرة (الانقسام الداخلي).

لمحاربة الانقسام الداخلي، يلجأ المُخَصّص إلى تقسيم (splitting) القطاع المراد تخصيصه إلى قسمين: قسم لتلبية طلب الذاكرة و قسم يضاف إلى قائمة القطاعات الحرة. في بعض الأحيان قد يقوم المُخَصّص بدمج (coalescing) قطاعات حرة متجاورة و اعتبارها قطاعا واحدا أكبر حجما.

كما سنرى فيما بعد، فبعض أنظمة التخصيص تتعمد الانقسام الداخلي كسياسة مقصودة لمحاربة الانقسام الخارجي.

## 2-1 تصنيف لبعض أنظمة التخصيص :

بوجه عام، تستلزم تلبية طلب الذاكرة عمليتين رئيسيتين، أولا يجب البحث عن قطاع في الذاكرة الحرة كبير كفاية، تأتي بعده قرار تقسيم هذا القطاع للحصول على الحجم المطلوب. عكسيا، عند تحرير قطاع ما، قد يلجأ المُخَصّص إلى دمج هذا القطاع مع قطاع آخر حر بجواره. هذا بالطبع يقودنا إلى سؤالين رئيسيين: الأول يتعلق بطريقة البحث التي ينتهجها المُخَصّص للعثور على قطاع حر. و الثاني بالطريقة التي يخزن بها المُخَصّص معلوماته عن القطاعات الحرة. الجواب على هذين السؤالين هو ما يميز أنظمة التخصيص عن بعضها البعض.

### أ- تصنيف بالنظر لطرق البحث عن القطاعات

في هذه الطرق يتم الاعتماد على قائمة List بالقطاعات الحرة (مثل قائمة متصلة Linked List أو قائمة متصلة من الجهتين Doubly linked list). لتلبية الطلبات يقوم المُخَصّص بمسح هذه القائمة. يمكن تمييز عدة طرق للمسح و العثور على قطاع حر:

**طريقة أول متسع First fit:** يتم البحث بدءا من أول القائمة و التوقف عند العثور على أول متسع، يعني أول قطاع بحجم كاف لتلبية طلب الذاكرة. إذا كان هذا الأخير أكبر من الحجم المطلوب، يتم تقسيمه و إضافة الحجم الباقي إلى قائمة القطاعات الحرة. الهدف الأول من هذه الطريقة هو تلبية طلب الذاكرة في أسرع وقت ممكن. من انعكاسات هذه الطريقة على الانقسام أنه مع التقسيمات المتتالية للذاكرة تتراكم مجموعة من القطاعات الصغيرة في أول القائمة تعرف عند الباحثين ب splinters، وفي حال ما كانت هذه الأخيرة صغيرة جدا فإنها تعطل البحث لأن مسح القائمة يمر عليها في كل مرة.

هناك مشكل آخر يتعلق بالموقع – داخل القائمة – الذي تُرجع إليه القطاعات المحررة: يمكن وضعها في أول القائمة مما يحبذ إعادة استعماله في المرات القادمة في هذه الحالة التحرير لا يتطلب وقتا بعكس البحث. من جهة أخرى يمكن وضع القطاع المحرر حسب ترتيب عنوانه في الذاكرة مما يعطينا قائمة مرتبة بحسب عناوين القطاعات. في هذه الحالة التحرير مثل التخصيص يتطلب وقتا لكنه بالمقابل يسهل عملية دمج القطاعات المتجاورة Coalescing في الذاكرة و تخفيض نسبة الانقسام الخارجي. حسب الباحثين فإن استعمال هذه الطريقة بترتيب القائمة حسب العناوين يعطي نسبة انقسام معقولة.

**طريقة أفضل متسع Best fit:** هنا يتم البحث عن أصغر قطاع حجما و في نفس الوقت كاف لاستيعاب الحجم المطلوب، هدف هذه الطريقة هو تخفيض نسبة الانقسام الخارجي و ذلك بجعل القطاعات الباقية بعد التقسيم أصغر ما يكون مما يعني ضياع أقل ما يمكن للذاكرة. في هذه الطريقة يتطلب البحث وقتا أكبر لأنه يجب مسح القائمة كاملة أو حتى العثور على الحجم المطلوب.

المشكلة تأتي عندما يكون حجم الذاكرة كبيرا، في هذه الحالة و مع تراكم القطاعات بفعل الانقسام الخارجي فإن البحث الخطي (Linear search) في القائمة يصبح غير فعال و يتطلب الكثير من الوقت. عمليا يمكن معالجة هذا الأمر باستعمال بنيات معطيات أكثر فعالية من القائمة التقليدية (مثل أنواع متطورة من أشجار البحث الثنائي Binary search trees أو Balanced trees). بينت التجارب أن استعمال هذه الطريقة مع بنيات معطيات فعالة تؤدي إلى استعمال جيد للذاكرة.

**طريقة المتسع القادم Next fit:** بدل البحث من أول القائمة يتم البدء من الموقع الذي توقف فيه البحث في المرة السابقة. تهدف هذه الطريقة لتخفيض الوقت المتوسط للبحث في القائمة. عمليا، لوحظ أن هذه الطريقة قد تؤدي إلى نسبة انقسام أكبر من الطريقتين السابقتين.

كما قلنا فهذه الطرق في صيغتها التقليدية تتناقص فعاليتها مع تزايد حجم الذاكرة. لاحظ أيضا أن نسبة الانقسام الخارجي قد تؤثر على سرعة البحث ذلك أنها ترفع من عدد القطاعات في القائمة.

باستعمال بنيات معطيات متطورة، فإن طريقتي أفضل متسع و "أول متسع- مع ترتيب القائمة حسب العناوين" تعملان كلاهما بشكل جيد على أرض الواقع من حيث استعمال الذاكرة، و ربما من حيث تحسين الأداء.

## ب- تصنيف بالنظر لسياسة التقسيم

بينما تقوم المخصصات التقليدية كالتي رأينا في الفقرة السابقة بتقسيم القطاعات الحرة للحصول على الحجم المطلوب بدقة، تلجأ بعض المخصصات إلى فرض "قيود" على طريقة التقسيم. بعبارة أخرى: قد تقسم الذاكرة الحرة – مسبقا – إلى قطاعات متساوية من نفس الحجم – لنقل مثلا 8 بايتات -. لتلبية طلب ذاكرة من حجم معين – مثلا 5 بايتات – يقوم المخصص بتخصيص أحد القطاعات من 8 بايتات بأكمله لهذا الغرض. لكن ماذا يحدث ل 3 البايتات المتبقية؟ لا شيء سوى أنها تضيع مع القطاع المخصص (تذكر مفهوم الانقسام الداخلي) و لا تعود إلا بعد تحريره من طرف البرنامج المستفيد.

المثال البسيط الذي ذكرنا غرضه فقط تبسيط الأمور. فطلبات الذاكرة تأتي في الواقع بأحجام مختلفة و قد تفوق الحجم الذي استعمل كمعيار للتقسيم (ال8 بايتات في مثالنا). في واقع الأمر، بدل استعمال قياس واحد لتقسيم الذاكرة، فإن المخصص يستعمل عدة أحجام-معايير. كمثال آخر: يقسم جزء من الذاكرة الحرة إلى عدد من قطاعات من سعة 2 بايت، جزء آخر لعدد من قطاعات من سعة 4 بايت و هكذا...


في العادة تستعمل عدة قوائم لتذكر مواقع القطاعات الحرة، تحديدا قائمة لكل فئة من الأحجام المستعملة: في حال مثلا استخدمت 10 قياسات لتقسيم الذاكرة الحرة فإننا نحتاج ل 10 قوائم.

عندما يطلب من المخصص ذاكرة بحجم من 2 بايت فإنه يستعمل مخزونه من قطاعات 2 بايت. عندما يطلب منه ذاكرة ب 4 بايت يستعمل مخزون ال 4 بايت. لكن إذا طلب منه ذاكرة مثلا من 3 بايت فإنه يستعين بأقرب قطاع يستوعب هذا الحجم الذي هو في مثالنا قطاعات 4 بايت.

ما المقصود بهذه السياسة التي تتعمد إضاعة الذاكرة و التسبب في الانقسام الداخلي؟ الجواب يكمن في :

أولا، البحث الآن لا يتم في قائمة واحدة طويلة بل فقط في القائمة التي تتوفر على الحجم المناسب مما يعني وقتا أقل.
ثانيا، رغم التسبب في الانقسام الداخلي فإن هذا يجنبنا تراكم القطاعات الصغيرة التي تبقى من جرّاء الانقسام الخارجي.

الآن كل الاختلاف بين هذا النوع من المخصصات – التي تعرف بالمخصصات التجميعية Segregated allocators – هو في اختيار القياسات المعتمدة لتقسيم الذاكرة الحرة. في هذا الدرس سنكتفي بدراسة أحد أشهر هذه المخصصات و هو المخصص التجاوري Buddy Allocator.

لفهم عمل هذا المخصص لنعتبر أن لدين المساحة التالية من الذاكرة.

أولا تقسم هذه المساحة إلى قسمين متساويين

| | |
|---|---|
| 2 | 1 |

بعد ذلك لنقسم القسم رقم 2 إلى قسمين متساويين

| | | |
|---|---|---|
| 2-2 | 2-1 | 1 |

الآن نقسم القطاع 2-2 هو الآخر إلى قسمين متساويين

| | | | |
|---|---|---|---|
| 2-2-1 | 2-2-2 | 2-1 | 1 |

يمكن الاستمرار بهذه الوتيرة لغاية الوصول إلى أصغر حجم مسموح به مثلا 8 بايت.

لنفترض أن المخصص يريد تلبية طلب بحجم مكافئ (بعد تسطيره إذا لم يكن مساويا لأحد الأحجام أعلاه) للفئة 2-1. و لنفترض أنه بعد تخصيصه هذا القطاع تلقى طلبا آخر من نفس الحجم. بما أنه لم يعد قطاع حر من هذه الفئة فإن المخصص يستعمل الفئة ذات الحجم الأكبر الموالي – في مثالنا الفئة 1 -. أولا يقوم المخصص بتقسيم القطاع 1 إلى قسمين متساويين (ويزيحه بالتالي من قائمة القطاعات الحرة لهذه الفئة) ، ثم يخصص أحد القسمين لتلبية طلب الذاكرة بينما يضيف القسم الآخر إلى القائمة المكافئة لهذه الفئة.

يمكن هنا ملاحظة أنه بعد تقسيم قطاع من فئة ما إلى قسمين متساويين فإن حجمهما يندرج آليا في فئة أصغر. هكذا يضمن المخصص أنه لا قطاع في الذاكرة يضيع من جراء التقسيم و بالتالي يمكن استعماله مستقبلا لتلبية أحد طلبات الذاكرة.

المثال الذي درسناه فوق حالة خاصة للمخصص التجاوري يكون فيها حجم فئة ما ضعف الفئة السابقة.

هناك عدة أنظمة أخرى للتقسيم. أحدها تستعمل ما يسمى في الرياضيات بمتتاليات فيبروناتشي. في هذه المتتالية تكون قيمة كل عدد مساوية لمجموع العددين السابقين. مثلا إذا أخذنا 2 كأصغر حجم ممكن فإن المتتالية تصبح كالتالي

2 2 4 6 10 16 26 ...

المغزى من هذه الطريقة أن هناك دائما إمكانية لتقسيم قطاع من فئة ما ليعطي قطاعين من الفئتين الأصغر في الترتيب. مثلا يمكننا تقسيم قطاع من حجم 26 إلى قطاعين واحد من حجم 16 و آخر 10. بحيث يمكن إضافتهما للقائمتين المكافئتين الأصغر في الترتيب.

لاحظ أنه كلما صعدنا في الترتيب فإن المسافة بين أحجام الفئات تزداد تباعدا. تخيل أنه طلب من المخصص ذاكرة بحجم 17. الحل الوحيد يكون حينئذ تخصيص قطاع من فئة 26. مما يعني ضياع 26-17=9 بايتات. بصفة عامة فإن توزيع المتتالية التي نختارها يؤثر بصفة مباشرة على الانقسام الداخلي.

هناك نظام آخر في المخصص التجاوري يعطي فئات أحجام متباعدة أكثر و هو النظام الثنائي. هذا الأخير يستعمل أحجاما تكون قيمتها دائما قوة مرفوعة ل 2. أي يمكن كتابتها على شكل أس مرفوع

ل 2 : $2^x$ . مثلا الحجم 8 يمكن كتابته على شكل $2^3$ . في هذه الحالة تسمى 3 رتبة القطاع.
لننظر إلى الفئات التي ينتجها هذا النظام:

$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$

يمكن ملاحظة أن النظام ينتج فئات أحجام متباعدة أكثر من نظام فيبروناتشي، و بالتالي فهو أكثر عرضة للانقسام الداخلي. لكنه من جهة أخرى يسهل عملية البحث نظرا لقلة الفئات و بالتالي القوائم المستعملة. كما أنه يسهل عمليات الحساب لاستعماله قيما يمكن التعامل معها بفعالية في النظام الثنائي الذي تعتمد عليه الحواسب.

للإشارة فنظام التجاور الثنائي هو المستعمل في مخصص الذاكرة الحقيقية Physical Memory لنواة لينوكس.

المخصص التجاوري لا يسهل تدبير الذاكرة من حيث التقسيم فقط بل كذلك من حيث الدمج Coalescing. عندما تقوم بدمج قطاعين حرين متجاورين في الذاكرة و من نفس الحجم فإن الحاصل يكون دائما قطاعا من فئة أكبر حجما. يؤدي دمج القطاعات المتجاورة إلى تخفيف القوائم من عدد القطاعات المخزنة و بالتالي الرفع من سرعة الأداء

### ت- نظام خارطة البيتات

لختم هذه الفقرة، سنتحدث عن نظام فريد نوعا ما. بخلاف الأنظمة التي تعتمد عادة على قوائم خطية أو بنيات معطيات أكثر فعالية مثل بعض فصائل أشجار البحث Search trees. فهذا النظام يستعمل خارطة من البيتات Bitmap. كل بيت في هذه الخارطة يكافئ موقعا في الذاكرة. عندما تكون الذاكرة حرة في موقع ما تكون قيمة البيت 0 في حين تصبح 1 بعد تخصيص القطاع. مثلا إذا كانت الصورة التالية تمثل حالة الذاكرة


فإن البيتات المكافئة لهذه المواقع تكون على شكل 10011010

من ميزات هذا النظام الاقتصاد الذي يوفره لتخزين حالة الذاكرة، بيت لكل موقع، مما يعني أنه لتخزين حالة N موقع يمننا استعمال N/8 بايت. لكنه ليس بنفس الفعالية من حيث العثور القطاعات الحرة لأنه يعتمد طريقة البحث الخطي في خارطة البيتات. مما يعني أن كلفة البحث من حيث تتصاعد بتزايد حجم الذاكرة. هذا يضع مشكلا في الذواكر ذات الحجم الكبير. نظريا، يمكن الرفع من سرعة الأداء بإدخال بعض التحسينات عليه. لكن لا يمكن الجزم بنجاعته على أرض الواقع لأنه لم يستعمل في أنظمة واقعية بصفة رئيسية. بالمقابل يمكن استعماله بصفة ثانوية لتحسين أداء أحد المخصصات السالفة الذكر (كما يفعل لينوكس).

## 2- التنجيز

كما قلنا سابقا فإن نجاعة نظام تدبير الذاكرة مرتبطة ارتباطا وثيقا بالوسط الذي ستوضع فيه. ما هي إذن مميزات الوسط الذي سيعمل فيه مخصص الذاكرة في حالتنا؟

في درسنا نحن بصدد إنجاز مخصص للذاكرة الحقيقية Physical RAM أي الذاكرة المادية المثبتة على الحاسب. بطبيعة الحال لن نقوم بتدبير الذاكرة على صعيد البايتات المنفردة لأن حجم الذاكرة هائل و

سيكلفنا ذلك مساحة و وقتا كبيرين جدا. لاعتبارات خاصة ببيئة حواسب X86 سنقوم بتقسيم الذاكرة الحقيقية إلى قطع من 4 كيلوبايت (4096 بايت): تسمى هذه القطع بصور الصفحات Page Frames.

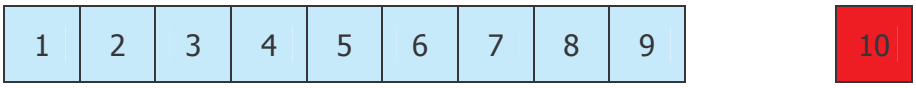
وظيفة مخصص الذاكرة الحقيقية توفير مساحة مادية من الذاكرة للذاكرة الافتراضية. في الدرس القادم سنعرف بالتفصيل ما هي الذاكرة الافتراضية، يكفي أن تعرف الآن أنها ذاكرة غير موجودة فعليا على الجهاز بل يتم "إيهام المعالج بوجودها". ما يهمنا معرفته في الوقت الحالي هو كيف تأتي طلبات الذاكرة من هذا النظام.

مدبر الذاكرة الافتراضي يطلب عدة صور لصفحات منفردة، و تخصيص صور متجاورة للصفحات في الذاكرة الحقيقية أمر غير ضروري في الغالب (هناك استثناءات مثل DMA لكن ذلك غير مطروح لنا في الوقت الراهن). و إذن كل المطلوب الآن هو التركيز على تخصيص الصفحات في وقت قياسي و عدم الاهتمام بانقسام الذاكرة.

الطريقة التي اخترتها في هذا الدرس تعتمد على بنية معطيات شبيهة بالكومة. هذه التقنية تخزن صور الصفحات غير المستعملة في جدول. عند طلب صورة لصفحة يتم تخصيص صورة الصفحة التي في رأس الجدول. بالمقابل عندما يتم تحرير صورة صفحة يتم وضعها في رأس الجدول: هكذا تعطى الأولوية لصور الصفحات التي تم تحريرها مؤخرا: تعرف هذه الطريقة ب "آخر دخول-أول خروج" Last in First out.

مثلا في البداية لدينا


طلب تخصيص لصورة صفحة




3 طلبات تخصيص أخرى متتالية


تحرير صورة الصفحة رقم 8


كما يظهر من المثال فإن صور الصفحات قد تصبح مبعثرة بعد فترة من الاستعمال و التحرير، لكن كما أوضحنا فإن ذلك لا يسبب مشكلا في الوقت الحالي.

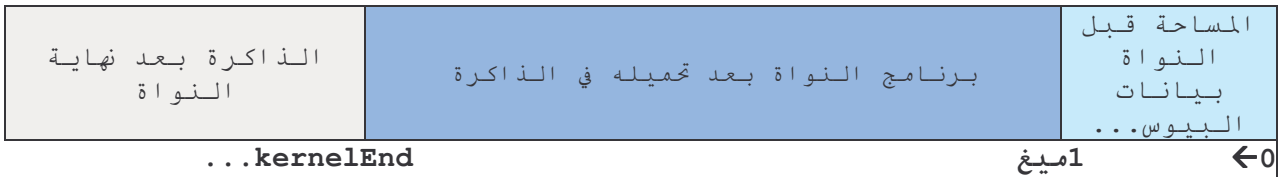
ملاحظة أخرى بخصوص هذا المخصص هي أن وقت التخصيص ثابت و لا يتأثر بحجم الذاكرة. بمعنى أن المخصص لا يضيع وقتا في البحث عن صور صفحات فارغة في الجدول و إنما يرجع آليا الصورة التي في رأس الجدول. لو استعملنا مثلا بنية أخرى مثل خارطة البيتات Bitmap فإن كلفة البحث كانت ستكون خطية (أي أن وقت البحث سيتزايد خطيا بتزايد حجم الذاكرة).

قبل المرور إلى تنجيز المخصص يجب علينا حل مشكل مهم: نحن لا نتوفر بعد على دعامة تمكننا من تخصيص الذاكرة المتغيرة Dynamic memory. حتى الآن التجأنا فقط لصنع المتغيرات بطريقة ثابتة (أي داخل ملف برنامج النواة) و صنع العناصر داخل كومة النواة. لكن لا يمكننا استعمال هذه الطريقة لتنجيز

المخصص لأننا لا نعرف كم من المساحة سيحتل مسبقا (حجم البنيات التي سيديرها المخصص مرتبط بحجم الذاكرة المثبتة على الجهاز وهذا المعطى غير متوفر لدينا ساعة الترجمة).

المطلوب إذن دعامة مسبقة لتخصيص الذاكرة المتغيرة. لكننا – بالضبط – بصدد إنجاز هذه الدعامة و لم نكملها بعد. كيف السبيل إذن لتجاوز هذه الحلقة المسدودة؟

الحل الذي اخترته – و هو المستعمل عادة و إن بطرق مختلفة – هو استعمال المساحة التي تبدأ بعد نهاية النواة. الصورة التالية توضح جيدا ما أريد قوله




المتغير kernelEnd يخزن العنوان الذي تنتهي عنده النواة. إذا كنت قد قرأت الدرس الثاني – عن برمجة النواة بسي++ - فلا بد أنك تذكر الرموز التي تم تعريفها في مخطوط محرر الروابط Linker script و بخاصة الرمز end الذي يشير إلى نهاية ملف النواة.

سنعرف إذن متغيرا kernelEnd لتخزين حدود نهاية النواة في الذاكرة. عندما يبدأ برنامج النواة التنفيذ نقوم بتهيئة القيمة الابتدائية لهذا المتغير

```
extern char end;
kernelEnd = (DWORD) &end;
```

عندما نريد صنع عنصر في الذاكرة ما علينا سوى تخصيص المساحة له انطلاقا من kernelEnd ثم بعد ذلك نغير هذا الأخير ليشير إلى نهاية الحد الجديد. من أجل ذلك نقوم بتنجيز دالة bootAlloc

```
void *bootAlloc(DWORD size) {

	DWORD addr = kernelEnd;

	نقوم بحساب الحد الجديد للنواة

	DWORD newEnd = kernelEnd + size;
	نتأكد من أننا نتوفر على ما يكفي من الذاكرة
	if(memorySize <= newEnd)
		return 0;

	ثم نمدد نهاية النواة
	kernelEnd = newEnd;
	return (void *)addr;
}
```

نحن الآن نتوفر على دالة شبيهة ب malloc المستعملة في سي. لكننا نشتغل في سي++ وهذه الأخيرة تعتمد بالأساس على المعامل new لصنع العناصر. لحسن الحظ، فسي++ تتيح لنا تعريف معاملنا new الخاص بنا.

```
inline void* operator new(DWORD size) {
	void *tmp = bootAlloc(size);
	return tmp;
}
```

بهذه الطريقة يمكننا استعمال المعامل new بطريقة طبيعية. مثلا إذا أردنا صنع العناصر التي استعملناها في الدروس السابقة مثل Exception

```
Exception *exc = new Exception();
```

الآن بعد أن حللنا مؤقتا مشكل تخصيص الذاكرة عند الإقلاع. يمكننا المرور إلى تنجيز مخصص الذاكرة الحقيقية.

لتنجيز الكومة التي ستخزن صور الصفحات سننشئ فئة جديدة نسميها PFrameStack

```
class PFrameStack
{     DWORD top;
      DWORD *frames;

public:

      PFrameStack(DWORD nbFrames);

            تقوم بإرجاع الصفحة التي في رأس الجدول. إذا كان الجدول فارغا تعيد 0

      inline DWORD pop() {
            return (top)?(frames[--top]):0;
      };

   إضافة صفحة إلى رأس الجدول و تحديث مؤشر الكومة. index هو رقم الصفحة في الذاكرة
                              عنوان الصفحة = رقم الصفحة x حجم الصفحة (4096 بايت)

      inline void push(DWORD index) {
            frames[top++] = index;
      };

                                          تعطينا عدد العناصر في الكومة

      inline DWORD size() {
            return top;
      }

};
```

كما تلاحظ فكل طرق الفئة – باستثناء المُصنّع – عرفت بطريقة سطرية inline. هذه التقنية تستعمل لتحسين الأداء حيث تدمج شفرة هذه الطرق مباشرة في المواقع التي يتم نداؤها منها. إذا استعملنا التعريف العادي سنخسر وقتا أكبر لأننا سنضطر للمرور عبر call كلما تم نداء هذه الطرق.

في المُصنّع نقوم بتهيئة الجدول الذي سيحتوي على عناوين صور الصفحات الحرة.

```
PFrameStack::PFrameStack(DWORD nbFrames) {

        تخصيص المساحة اللازمة لاحتضان عناوين الصفحات = عدد الصفحات * حجم العنوان

      top = 0;
      frames = (DWORD *) bootAlloc(nbFrames*sizeof(DWORD));
}
```

بعد تنجيز الكومة. سننشئ فئة ستشكل المخصص الفعلي الذي ستتعامل معه باقي الفئات. الفئة PhysMManager هي التي ستكون الواجهة المستخدمة لتخصيص الذاكرة الفعلية

(سؤال: لماذا لا نستخدم PFrameStack كواجهة مباشرة بدل المرور عبر PhysMManager؟ الجواب: هكذا يمكننا عزل تنجيز المخصص عن الخارج. فيما بعد يمكننا استعمال بنيات غير الكومة أو حتى عدة مخصصات في آن واحد بدون أن نضطر لتغيير أي شفرة خارج PhysMManager. مبرمجو سي++ و جافا يعرفون جيدا هذه التقنية: الحجب Encapsulation).

```
class PhysMManager
{
	PFrameStack *_pfStack;
	DWORD nbFrames;

	DWORD memTop;
	DWORD kernelTop;


public:
		هذه الشارة flag تحدد ما إذا كان يلزم ملئ صورة الصفحة ب0 قبل تخصيصها
	static const DWORD ZEROED = 1;

	PhysMManager();

		تخصيص صورة صفحة و احدة
	DWORD allocFrame(DWORD flags);

		تقوم بتحرير الصفحة رقم index (address = index*4096)
	void freeFrame(DWORD index);

		عدد الصفحات الحرة في الذاكرة
	DWORD freePages();

};
```

التنجيز:

```
		يشير إلى حدود برنامج النواة
extern DWORD kernelEnd;
		حجم الذاكرة بالبايت
extern DWORD memorySize;

PhysMManager::PhysMManager() {

		رقم آخر صفحة في الذاكرة
	memTop = pageBase(memorySize+1);

	nbFrames = memTop / PAGE_SIZE;
		صنع مخصص الكومة
	_pfStack = new PFrameStack(nbFrames);

		تسطير نهاية النواة على 4096 بايت
	kernelTop = pageTop(kernelEnd);

		إضافة صور الصفحات في الذاكرة السفلى إلى القائمة الحرة
	for (int i = START_PAGE; i < START_VGA; i+=PAGE_SIZE) {
		_pfStack->push(i>>PAGE_SHIFT);
	}

		إضافة صور الصفحات بعد النواة إلى القائمة الحرة

	for (int i = kernelTop; i < memTop; i+=PAGE_SIZE) {
		_pfStack->push(i>>PAGE_SHIFT);
```

```
	}

}

DWORD PhysMManager::allocFrame(DWORD flags) {

	استرجاع صورة صفحة من الكومة

	BYTE *addr = (BYTE *)_pfStack->pop();

	ثم في حالة تم التخصيص (addr != 0) و الشارة ZEROED flag محددة نملئ الصفحة ب0
	DWORD val = ((DWORD) addr) * PAGE_SIZE;

	if(addr && (flags & ZEROED)) {
		memset(val, 0, PAGE_SIZE);
	}

	نعيد عنوان الصفحة = رقم الصفحة * حجم الصفحة
	return val;
}

void PhysMManager::freeFrame(DWORD index) {
	_pfStack->push(index);
}

DWORD PhysMManager::freePages() {
	return _pfStack->size();
}
```

الآن وقد أنجزنا مخصص الذاكرة الحقيقية. يتبقى لنا تجريبه و التأكد من أنه يعمل كما ينبغي. برغم بساطته فقد ارتأيت أن أضيف دالة صغيرة لتجريب المخصص. فيما يلي سأقدم الدالة kmain حيث نقوم بصنع العناصر اللازمة باستخدام bootAlloc.

أريد فقط أن ألفت انتباه القارئ الذي تابع معي الدروس منذ البداية وحتى الآن أنني قد قمت ببعض التنظيم في الشفرة: لم أغير أيا من الدوال التي تم إنجازها في الدروس السابقة بل فقط قُمِت بتحويل مكانها داخل فئات جديدة حتى لا تتراكم لدينا مجموعة من الدوال و المتغيرات العامة التي تُعَقّد قراءة الشفرة. إذا كان القارئ قد قرأ الدروس السابقة فلن يجد مشكلة في قراءة الشفرة بالشكل الجديد. لبعض التوضيح أقدم عرضا عاما للبنية الجديدة للشفرة:

system.h/system.cpp: تحتوي على الشفرة القاعدية و الضرورية لعمل جميع الفئات الأخرى. أقصد التعريفات ...WORD-DWORD – الدوالmemcopy-memset-intToString بالإضافة إلى bootAlloc و المعامل new.

الفئة Machine تحتوي على الشفرة اللازمة لإعداد ج.م.ع GDT و ج.م.إ IDT. تعريفات الدوال inb و outb لقراءة البوابات الفرعية و cli() - sti() بالإضافة إلى بنيات reg .... الفئة تعرف الطريقة init() التي تقوم بنداء الدالتين setupGDT() و setupIDT().

الفئة Kernel تحتوي على طريقتين: init() تقوم أساسا بالتأكد من أن الإقلاع تم بكروب باستعمال بيانات البنية multiboot_info التي يمدنا بها كروب عند الإقلاع.

```
void Kernel::init(DWORD magic, multiboot_info *mbi) {

	// is magic number valid?
	if(magic != MULTIBOOT_MAGIC) {
		video->printf("Invalid magic number %x\n", magic);
```

```
            Machine::hlt();
      }
}
```

الطريقة initPhysMManager تقوم بصنع عنصر من الفئة التي رأينا سابقا PhysMManager و تخزينه في العضو physMManager.

```
void Kernel::initPhysMManager() {

      physMManager = new PhysMManager();
}
```

لاحظ أننا نستعمل المعامل new بطريقة طبيعية لأننا قمنا بإعادة تعريفه بما يلائم احتياجنا.

و أخيرا الفئة FATAL_ERROR تقوم بطباعة جملة على الشاشة و توقيف عمل النواة

```
extern Video *video;

void Kernel::FATAL_ERROR(const char *s) {
      video->printf(s);
      Machine::hlt();
}
```

الفئة IntManager تحتوي على الدالتين setHandler و unsetHandler. هاتان الدالتان كما رأينا في الدرس الخاص يتدبير الانقطاعات تقومان يتسجيل مدبرات الانقطاعات من فئة Interrupt.

الفئة PIC تضم جميع الدوال الخاصة ببرمجة محكم الانقطاعات الصلبة IRQ التي رأيناها في الدرس السابق.

باقي الفئات Video – Exception و Timer لم يلحقها أي تغيير.

أود فقط أن أشير أن هذا النموذج لا يعكس أي تصميم Design مسبق (هذا عمل يتجاوز بكثير نطاق هذا الدرس) بل غرضه فقط تنظيم مكونات الشفرة بغرض تسهيل صيانتها و تبديلها مستقبلا.

قبل المرور إلى شفرة Kmain. سأشرح بإيجاز كيف سنقوم بتجريب المخصص: التجربة تتم على مرحلتين في مرحلة أولى سنطلب تخصيص كل الصفحات الحرة. في كل صفحة مخصصة سنكتب بعض البيانات التي ستشكل "العلامة" على أن هذه الصفحة قد تم تخصيصها. عند تخصيص صفحة جديدة نقوم بالتأكد أن المخصص لم يعطنا صفحة تم تخصيصها مسبقا و ذلك عبر التأكد من أنها لا تحتوي على العلامة السابق ذكرها. في المرحلة الثانية، نقوم بتحرير جميع الصفحات و نتأكد من أن جميع الصفحات قد حررت من طرف المخصص عبر مقارنة عدد الصفحات الحرة بعد التحرير بعددها قبل التحرير.

و الآن بعد هذا التوضيح ننتقل إلى شفرة الدالة kmain.

```
extern "C" void kmain(DWORD magic, multiboot_info *mbi) {
      extern char end;
      kernelEnd = (DWORD) &end;

      العضو mem_upper في multiboot_info يعطي حجم "الذاكرة العليا" أي الموجود بعد
            أول ميغ. الحجم مخزن بالكيلوبايت لذلك نضربه في 1024 و نضيف عليه 1 ميغ.
      memorySize = mbi->mem_upper*1024+0x100000;

      video = new Video();
```

```
	video->clear();

	kernel::init تقوم بالتأكد من أن الإقلاع تم بكروب بالاستعانة ب multiboot_info
	Kernel::init(magic, mbi);
	printMemInfo(mbi);

	الفئة Machine تضم الشفرة اللازمة للتعامل مع المعالج
	inb, outb, cli, sti ...
	struct regs, GDTR, IDTR...
	setupGDT, setupIDT

	Machine::init();
	PIC::init() تقوم بإعداد محكم الانقطاعات
	PIC::init();
	ثم نقوم بملئ ج.م.إ IDT بمدبرات المرحلة الأولى (المكتوبة بالأسمبلي)
	initISRS();
	Machine::sti();

	pageTest ستستعمل كما سنرى لتخزين عناوين صور الصفحات التي سنطلبها من المخصص
	أثناء تجريبه. بما أننا أثناء التجريب سنقوم بتخصيص كل الصفحات الحرة فإننا نحتاج
	لمساحة = عدد الصفحات * حجم العنوان (DWORD)

	pageTest = (DWORD*) bootAlloc(
			(memorySize/PAGE_SIZE)*sizeof(DWORD)
			);

	Exception *exc = new Exception();
	for (int i = 0; i < 32; ++i) {
		IntManager::setHandler(i, exc);
	}

	Timer *t = new Timer();
	t->setPhase(20);
	IntManager::setHandler(32,t);
	PIC::setPIC1Mask(0xFE);

	video->printf("Init Physical Memory Manager");
	Kernel::initPhysMManager();
	video->printf("      [OK]\n");

	هذه الدالة التي تقوم بتجريب المخصص الحقيقي. أنظر أسفله
	testPhysMem();

	while(1);
}
```

مجرب المخصص الحقيقي يتكون من ثلاث دوال.

الدالة allocateAll تقوم بتخصيص جميع الصفحات الحرة و تخزين عناوينها في الجدول pageTest

```
void allocateAll() {
	DWORD *addr;
	while(1) {

		نقوم بطلب صفحة من المخصص

		addr = (DWORD *)physMManager->allocFrame(0);

		إذا لم تعد هناك صفحة حرة نخرج من الحلقة
```

```
            if(!addr)
                  break;

                              نتأكد من أن الصفحة لم يتم تخصيصها من قبلنا مسبقا

            if(*addr == (DWORD)addr && addr[(PAGE_SIZE-1)/4] == 0x55) {
                  video->printf("Error, Page at %u already allocated\n",
                                    (DWORD)addr);
                  Machine::hlt();
            }

                                                نكتب علامة التخصيص في الصفحة
            addr[0] = (DWORD)addr;
            addr[(PAGE_SIZE-1)/4] = 0x55;
            pageTest[nbAllocated] = (DWORD)addr;
            nbAllocated++;
            nbFree--;
            printMemState();
      }
}
```

عكسا، الدالة freeAll تقوم بتحرير كل الصفحات التي تم تخصيصها من قبل allocateAll

```
void freeAll() {
      DWORD *addr;
      while(1) {
            if(!nbAllocated)
                  break;
                              نستعيد عنوان الصفحة التالية من الجدول pageTest

            addr = (DWORD *)pageTest[nbAllocated-1];

                        ثم نتأكد من أن الصفحة تحتوي على علامة التخصيص (الحذر واجب ☺)

            if(addr[0] != (DWORD)addr || addr[(PAGE_SIZE-1)/4] != 0x55) {
                  video->printf("Error, Page at %u corrupted\n",
                                    (DWORD)addr);
                  Machine::hlt();
            }

                                    نزيح علامة التخصيص ثم نعيد الصفحة إلى المخصص

            addr[0] = 0;
            addr[(PAGE_SIZE-1)/4] = 0;
            physMManager->freeFrame(((DWORD)(addr)>>PAGE_SHIFT));
            nbFree++;
            nbAllocated--;
            printMemState();
      }
}
```

و الآن الدالة الأساسية testPhysMManager

```
void testPhysMManager() {
      nbAllocated = 0;
      physMManager = Kernel::physMManager;

                                    قبل كل شيء لنسجل عدد الصفحات الحرة

```

```
	DWORD initialNbFree = nbFree = physMManager->freePages();

	video->printf("testing Physical Memory Manager  \n");

                                                  نخصص جميع الصفحات

	allocateAll();
                                                  نتأكد من أنه تم فعلا تخصيص كل الصفحات

	if(nbAllocated != initialNbFree) {
		video->printf("There were %u free pages but\
                        only %u were allocated",
                         physMManager->freePages(), nbAllocated);

		Machine::hlt();
	}

                                                  ثم نعيد للمخصص جميع الصفحات

	freeAll();

                                  و نتأكد من أنه قد استعاد منا جميع الصفحات و لم يضع شيئا

	if(initialNbFree != physMManager->freePages()) {
		video->printf("There were %u allocated pages but\
                        only %u were freed",
                         physMManager->freePages(), nbAllocated);

		Machine::hlt();
	}

	video->printf("Physical Memory Manager test Ok\n");
}
```

للترجمة نستعمل makefile التالي

```
CPPFLAGS = -c -fno-builtin -fno-rtti -fno-exceptions

.PHONY:  clean

OBJS = multiboot.o system.o Machine.o pic.o Kernel.o\
 IntManager.o main.o video.o isrs.o\
  exception.o timer.o PFrameStack.o PhysMManager.o


all: kernel.elf

kernel.elf: $(OBJS) link.lds
	ld -T link.lds $(OBJS) -o kernel.tmp
	objcopy -O elf32-i386 kernel.tmp kernel.elf
	$(RM) kernel.tmp
	"C:\Program Files\WinImage\winimage.exe" floppy.img /i kernel.elf /h
	 objdump.exe -D kernel.elf > log.txt

clean:
	$(RM) $(OBJS)


%.o: %.cpp
	g++ $(CPPFLAGS) $< -o $@
```

```
%.o: %.s
      nasm -f elf  -DLEADING_USCORE $< -o $@
```

ملحوظة: لم أدرج الترابطات dependencies بين الملفات لحفظ المساحة (و أيضا لتكاسلي في إضافتها إلى الماكفايل. أرجو المعذرة ☺).

الشاشة التالية تعطي صورة لتنفيذ النواة في المحاكي بوكس Bochs

```
Assalamou Alaikoum from grub
memory = 32768KB
Boot device = floppy A
Init Physical Memory Manager      [OK]
testing Physical Memory Manager




6440 Frames allocated
1619 free Frames









10 secondes elapsed
```

في الدرس القادم إن شاء الله سنرتفع إلى أعلى قليلا لنرى كيف تعمل الذاكرة الافتراضية Virtual Memory. إذا أردت النسخة العربية من البرنامج اقرأ الملحق أسفله.

*ياسين الوافي طنجة - المغرب 20 فبراير 2006 الساعة 23:35*

# ملحق الدرس الخامس: تعريب الشاشة النصية

في هذا الدرس قررت أن أضيف ملحقا عن موضوع "تعريب" الشاشة النصية. في الحقيقة، و قبل أن نبدأ، أحب أن أوضح أن موضوع التعريب، و بصفة عامة موضوع التدويل Internationalization، هو مجال لوحده و يلزمه كتاب أو أكثر للإحاطة بجوانبه. لكن هذا لن يمنعنا بطبيعة الحال من مقاربتنا له بصفة أولية في هذه المرحلة.

الآن إلى المهم، الهدف هو طباعة الحروف العربية على الشاشة النصية. لكن هذه المهمة ليست بسيطة، بمعنى لا يكفي التوفر على رسوم للحروف وطباعتها على الشاشة. فاللغة العربية كما يعلم الكل، لا تكتب ببساطة كمثيلاتها اللاتينية. وهناك بعض المشاكل المتعلقة بالحروف العربية التي ينبغي معالجتها. في هذا الملحق سنشير إلى بعض من هذه المشاكل و كيفية التغلب عليها. بدون إطالة، فيما يلي المراحل اللازمة لإضافة العربية إلى نواتنا التدريبية.

## 1- كيف نرسم الحروف العربية

أولا لا بد من نظرة عامة على كيفية عمل الفيجا.
تعتمد الفيجا على مجموعة من المكونات. كل مكون يقوم بعمل محدد و تمهيدي لعمل المكون الذي بعده:

أولا، الSequencer، يقوم بقراءة البيانات التي قمنا بكتابتها في ذاكرة الفيجا. ثم يحول هذه الأخيرة إلى صيغة قابلة للفهم من قبل المكونات التالية (نحن نكتب فقط أرقام الحروف لكن هذا المكون يقوم بتحويلها إلى نقط Pixels كما سنرى فيما بعد)، تسمى هذه العملية ب Rasterization.

الAttribute Controller يقوم باستعمال بيانات الألوان التي نكتبها في الذاكرة بالإضافة إلى البيانات التي ينتجها الSequencer و من ثم إنتاج عدد من 8 بيت تقدم للمكون الموالي DAC.

الDAC يقوم بتحويل ال8 بيتات – التي تعمل كمؤشر على أحد مسجلا الألوان الخاصة به - ة إلى إشارة تماثلية Analog Signal.

ال CRTC يحدد الصيغة المطلوبة لإرسال الإشارة السابقة إلى الشاشة. مثل التردد، Timing...

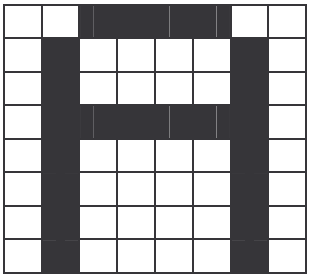
هناك أيضا مكون آخر يتوفر على مسجلاته الخاصة و هو الGraphics. يمكن برمجته للتحكم في الطريقة التي يقرأ بها المعالج ذاكرة الفيجا.

كل من المكونات المذكورة يوجد على بوابتين فرعيتين و مجموعة مسجلات. للتواصل معه ينبغي أولا كتابة رقم المسجل الذي نريد الولوج إليه في البوابة الفرعية الأولى (بوابة العنونة) ثم كتابة (أو قراءة) البيانات في البوابة الفرعية الثانية (بوابة البيانات).

ما يهمنا من هذا الأمر هو جعل الSequencer يستعمل وصفنا الخاص للحروف لإنتاج رسوم للحروف العربية.

في المرحلة الأولى علينا تحميل رسوم الحروف إلى ذاكرة الفيجا. حيث أن هذا الأخيرة تعتمد لرسم الحروف على الشاشة النصية على مجموعة من الرسوم Bitmap التي تصف شكل الحروف. هذه الرسوم مخزنة في ذاكرة الفيجا و مرتبة على شكل جدول من مداخل بحجم 32 بايت: كل مدخل يصف الحرف المقابل له. مثلا إذا كتبت في ذاكرة الشاشة النصية كما رأينا في الدروس السابقة الرقم 65 الذي يكافئ رقم الحرف A فإن بطاقة الفيجا تقوم باستعمال المدخل 65 (بدءا من 0) لرسم الحرف A على الشاشة.

تقسم ال32 بايت المشكلة لرسم الحرف الواحد إلى سطور. كل بايت يمثل سطرا واحدا: و كل بيت في هذا البايت يشكل نقطة في هذا السطر. تماما كما لو كنت تستعمل جدولا للرسم. كما في الشكل التالي الذي يعطي مثالا لرسم الحرف A



في هذا الجدول من 8x8 فإن البايتات المشكلة له يمكن أن تكتب على شكل

00111100
01000010
01000010
01111110
01000010
...

رسوم حروف الفيجا تكون دائما بعرض 8 بيتات (هناك وضع يستعمل 9 بيتات على الشاشة لكن الرسوم تظل بعرض 8 بيتات). لكن ارتفاعها ( وبالتالي عدد البايتات ما دام كل بايت يشكل سطرا واحدا) يختلف ويتم تحديد قيمته في مسجل Maximum scan line و هو أحد مسجلات CRTC ( تذكر الدرس الثالث عن النواة بسي++) تحديدا المسجل رقم 9 في ال5 بيتات الأضعف منه. مما يجعل أقصى قيمة له هي 32. لهذا السبب فكل رسم لحرف يشغل 32 بايت في الذاكرة و يبدأ في عنوان مسطر على 32 بايت. في حال ما كان الارتفاع أقل من 32، و معه رسم الحرف أقل من 32 بايت، فإن الفيجا تستعمل فقط عدد السطور (=البايتات) المطلوبة و تتجاهل السطور المتبقية.

عندما يقلع الحاسب فإن البيوس يقوم بإعداد خارطة لرسوم الحروف – مسبقا – مما يمكن البرامج المنفذة بعده من استعمال الشاشة النصية مباشرة. البيوس يمنح أيضا انقطاعات لتغيير رسم حرف أو مجموعة حروف في ذاكرة الفيجا. لكننا طبعا لا نستطيع استعمالها في الوضع المحمي. مما يحتم علينا التعامل مباشرة مع الفيجا.

بفرض أننا نتوفر على خارطة لرسوم الحروف كيف يمكننا استعمالها؟ الجواب بسيط : عليك بكتابة خارطة البايتات في ذاكرة الفيجا تحديدا في الذاكرة 0xA0000.

الآن السهولة مرتبطة بفهم كيف تعمل ذاكرة الفيجا. و هذا الأمر يتطلب قدرا من التوضيح برغم أننا في ملحق. ذاكرة الفيجا المستعملة للعرض من حجم 256 كيلوبايت و تبدأ كما قلنا في 0xA0000. في الوضع النصي Text Mode – حيث نكتب حروفا و ندع الفيجا ترسمها بدل الرسم مباشرة - تقسم هذه الذاكرة إلى أربعة قطع من 64 كيلوبايت. تسمى هذه القطع مستويات Planes.

الآن حان الوقت لتعلم أنه عندما نقول أن ذاكرة الفيديو تبدأ في 0xA0000 فهذا لا يعني أننا نتكلم عن الذاكرة الحقيقية RAM. لا يمكننا استعمال الذاكرة الحقيقة في هذا الموقع لأن الفيجا تحجبه. مزيد من التوضيح: تذكر الدرس الثاني، عندما يريد المعالج الولوج إلى موقع في الذاكرة يقوم بإرسال العنوان على ناقلة العناوين ثم يستقبل البيانات من الذاكرة الحية على ناقلة البيانات. لكن يحدث في بعض الأحيان أن بعض الفروع تقوم بمراقبة ناقلة العناوين و قطع طلبات الولوج إلى مناطق معينة من الذاكرة ثم ترسل ما تريد للمعالج كما لو أن البيانات أتت من الذاكرة الحية للجهاز. عندما يريد المعالج كتابة معلومة في الذاكرة فإنها تتعرض لها أيضا و تحول مسارها نحوها كما لو أن المعالج قام بالكتابة في الذاكرة الحية. بعبارة أخرى فإن هذه الفروع تستولي على جزء من فضاء عنونة المعالج. هدف هذه التقنية Memeory Mapped I/O هو أنه يمكنك التواصل مع الفروع عبر قراءة و كتابة المناطق الخاصة بها في الذاكرة. و هذه الطريقة أكثر فعالية من التواصل عبر البوابات الفرعية Port Mapped I/Oحيث أن هذه الأخيرة بطيئة نسبيا، بالإضافة إلى أنها تستلزم تعليمات الأسمبلي in و out. بينما التقنية الأولى تتطلب فقط مؤشرا على الموقع المناسب في الذاكرة.

الآن للعودة إلى موضوعنا فإن الفيجا تستولي على فضاء العنونة من 0xA0000. لكن للمواءمة مع البطاقات الأقدم منها فإن العنوان قد يختلف: مثلا في الوضع النصي بالألوان الذي نستعمله تعنون الفيجا من 0xB8000. لتحديد عنونة الفيجا نستعمل المسجل Miscellaneous Graphics Register و هو المسجل رقم 6 في مسجلات الGraphics. البيتات 2 و 3 من هذا المسجل تحدد عنونة الفيجا:
00: من 0xA0000 إلى 0xBFFFF.
01: من 0xA0000 إلى 0xAFFFF
10: من 0xB0000 إلى 0xB7FFF
11: من 0xB8000 إلى 0xBFFFF

لتحميل الحروف علينا استعمال الوضع 00.

كما قلنا فذاكرة الفيجا تقسم إلى 4 مستويات. كل مستوى من 64 كيلوبايت. تذكر أننا عندما نريد كتابة الحروف في الذاكرة كنا نكتب رقم الحرف متبوعا ببيانات اللون. ما يحدث في الواقع هو أن نظام الGraphics كان يحول مسار العناوين الزوجية إلى المستوى الأول حيث تخزن أكواد الحروف. في حين تحول العناوين الفردية إلى

المستوى الثاني حيث تخزن خصائص الحروف من اللون و ما شابه. هذه الطريقة في التأويل تسمى ODD/EVEN و يمكن برمجتها في مسجلات الSequencer (تحديدا البيت رقم 2 من المسجل رقم 4 عندما يوضع على 1 فإنه يعطل هذه الطريقة في التأويل).
لتحميل الحروف سنحتاج لتعطيل هذا الوضع. هناك أيضا مجموعة من المسجلات في الGraphics التي تجعل الفيجا تطبق مجموعة من التحويلات على البيانات من و إلى المعالج و ينبغي أيضا تعطيلها (بوضعها على 0).

رسوم الحروف في الوضع النصي تكون مخزنة في المستوى الثالث (رقم 2 بدءا من 0)، و لتمكيننا من الكتابة في هذا المستوى دون غيره يوجد المسجل رقم 2 في الSequencer. الـ4 بيتات الأضعف تتحكم في المستويات التي تكتب فيها البيانات الواردة من المعالج. لا حظ أننا يمكننا الكتابة في أكثر من مستوى دفعة واحدة إذا وضعنا البيتات المكافئة لها على 1. في حالتنا نحتاج لتنشيط المستوى 2 دون غيره مما يعني وضع البيتات على 4 (0100).

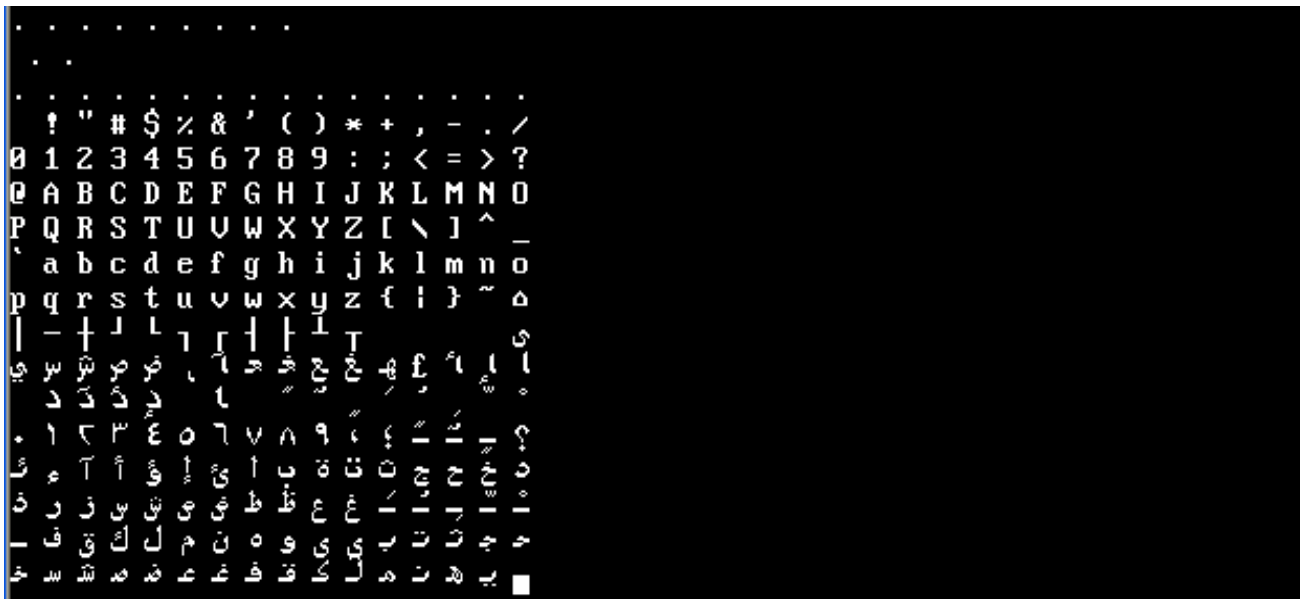
للتلخيص:
- ذاكرة الفيجا مقسمة إلى 4 مستويات كل مستوى من 64 كيلوبايت.
- المستوى 0 يستعمل لتخزين أرقام الحروف في خارطة الرسوم، المستوى 1 يخزن خصائص الطباعة من اللون و ما شابه. المستوى 2 يخزن خارطة (أو أكثر) رسوم الحروف.
- في الوضع النصي، تؤول العناوين بطريقة زوجي/فردي Odd/Even حيث ترسل العناوين الزوجية للمستوى 0 والفردية للمستوى 1.

عمليا لتحميل خارطة حروف جديدة و استعمالها بدل القديمة:
- تحديد خارطة العنونة لتبدأ من 0xA0000 بدلا من 0xB8000.
- تعطيل العنونة ب Odd/Even.
- اختيار المستوى 2 للكتابة فيه دون غيره.
- وضع القيم الصحيحة في مسجلات الGraphics لكي لا تؤثر على البيانات التي سنكتبها.
- كتابة خارطة رسوم الحروف في الذاكرة بدءا من 0xA0000.

في نواتنا التدريبية. سننشئ فئة جديدة FontMap و إسناد هذه المهمة لها عبر الطريقة loadFontMap.

```
void FontMap::loadDefaultMap(char *fmap) {
        char *mem = (char *) 0xA0000;
                                        أولا نقوم بحفظ قيمة المسجلات قبل تغييرها
        saveVGARegs();

                                        إعداد الفيجا للكتابة في المستوى رقم 2
        outVGA(VGA_SEQ, 0, 1);  // Synchronous reset
        outVGA(VGA_SEQ, 2, 4);  // only write to map 2
        outVGA(VGA_SEQ, 4, 7);  // Disable O/E, enable Extended memory
        outVGA(VGA_SEQ, 0, 3);  // clear synchronous reset

        outVGA(VGA_GR, 1, 0);   // disable set/reset
        outVGA(VGA_GR, 3, 0);   // logical operation=replace,
                                // rotate count=0
        outVGA(VGA_GR, 4, 2);   // selects map 2 for cpu reads
        outVGA(VGA_GR, 5, 0);   // write mode=0, read mode=0
        outVGA(VGA_GR, 6, 0);

الآن يمكننا نقل خارطة رسوم الحروف. في مثالنا نستعمل حروف بحجم 8x16 أي أن حجم رسم حرف واحد 16 بايت بينما رسم الحرف في الفيجا ينبغي أن يكون مسطرا على 32 بايت.
        for (int i = 0; i < 256; ++i)  {
                for (int j = 0; j < 16; ++j) {
                        mem[i*32+j] = fmap[i*16+j];
                }
        }
                                        إعادة القيم الأصلية للمسجلات.

        restoreVGARegs();

}
```

للتأكد من أن خارطتنا قد حملت فعلا للذاكرة يمكننا طباعة الحروف من 0 إلى 255 على الشاشة النصية لنرى هل تستعمل الفيجا الرسوم التي قمنا بتحويلها. الصورة التالية تظهر النتيجة



جيد، الآن يمكننا المرور إلى المرحلة التالية.

بعد أن قمنا بتجميل الحروف يمكننا كتابتها بوضع رقم الحرف الذي نريد في ذاكرة الشاشة النصية. لكن هذا لا يحل كل مشاكلنا. لأن الحروف العربية لها مميزات يعرفها الكل. أولا تكتب من اليسار إلى اليمين في حين أن الفيجا تفترض أن الحروف تكتب من اليسار إلى اليمين لدعم اللغة الإنجليزية. ثانيا الحرف العربي الواحد قد يأخذ أشكالا متعددة حسب وضعه في الكلمة.

عمليا نريد أن نتمكن من القيام بشيء من هذا القبيل في شفرتنا

```
video->printf("السلام عليكم");
```

و نريد أن تتم معالجة المشاكل بطريقة شفافة لمستعمل printf أو الدوال المتعلقة بالشاشة النصية مثل moveTo(x,y).

هذا يتطلب منا تعديل الشفرة المسؤولة عن التعامل مع الشاشة النصية. في نواتنا الأمر يتعلق بالفئة Video.

فيما يلي سنرى المشاكل التي ترافق مشكلة طباعة الحروف العربة و الحلول التي اخترتها لمعالجتها. أريد فقط أن أشير إلى أنني استعرت بعض التقنيات من شفرة acon في لينوكس.

أولا علينا تعديل شفرة الفئة Video. كما رأينا في الدرس الثالث، فإن هذه الفئة تقوم بطباعة الجمل أو الحروف التي نمدها لها مباشرة في الشاشة النصية. لكننا الآن ملزمون بمعالجة الجمل قبل طباعتها على الشاشة. لأجل ذلك سنقوم بوضع الحروف قبل طباعتها في جدول مسبق.

```
static CHAR screen[WIDTH*HEIGHT];
```

هذا الأخير سيمثل الشاشة المنطقية Logical screen أي الحروف كما يتم إمدادنا بها في دوال printf و put. بعد معالجة هذه الحروف سيتم نقلها إلى الشاشة المرئية Visual screen. قبل استعمال العربية، كان هناك تماثل بين الشاشة المنطقية و الشاشة المرئية، لكن الوضع يختلف الآن.

الدالة put لا تكتب الحروف الآن مباشرة في الشاشة المرئية بل في الشاشة المنطقية.

```
else if(c >= ' ')
{
     screen[_y*width + _x] = c;
```

```
    _x++;
} else
    screen[_y*width + _x] = '.';
```

عندما ننتهي من معالجة الحروف و تحويلها نقوم بنقل النتيجة إلى الشاشة المرئية عبر نداء updateScreen.

```
void Video::updateScreen() {

        for (int i = yfrom; i <= _y; ++i) {
                renderLine(i,
    كما سنرى فيما بعد، فالطريقة process في الفئة FontMap تاخذ سطرا من الشاشة المنطقية و تعيد صيغته المرئية بعد معالجة الحروف
                map->process(screen+(i*WIDTH),  writeMode)
                );
        }
        //reinit yfrom
        yfrom = _y;
}
```

هذه الدالة تأخذ الصيغة المرئية للسطر y من FontMap::process و تنقله إلى الشاشة المرئية

```
inline void renderLine(int y, CHAR *line) {
        for (int j = 0; j < width; ++j) {
               int loc2 = ((y*width)+translate(j))*2;
               mem[loc2] = (line[j]!=0)?line[j]:' ';
               mem[loc2+1] = _attribute;
        }
}
```

لاحظ أنه لحساب البعد في الشاشة النصية لا نستعمل الصيغة العادية 2*(j+(i*80)). بل نمرر الإحداثي j الذي يمثل الموقع داخل السطر إلى دالة tranaslate. هذه الأخيرة تقوم بحساب البعد المناسب حسب الوضع الذي نوجد فيه عربي أو لاتيني

```
enum WriteMode { Arabic, Latin };
…

class Video {
…
                    الخاصية writeMode تخزن وضع الكتابة المستعمل: عربي أو لاتيني
WriteMode writeMode;
FontMap map;
…

inline int translate(int x) {
        return (writeMode != Arabic)?x:width-x-1;
}
```

للتلخيص فإننا عندما نطلب طباعة جملة من الفئة Video تقوم هذه الأخيرة بما يلي

1- تقوم بطباعة الحروف قبل معالجتها في الشاشة المنطقية
2- تمرر السطور التي تم تغييرها إلى الطريقة FontMap::process لمعالجة الحروف العربية و ترتيب الحروف.
3- تطبع النتيجة في الشاشة المرئية مع الأخذ بعين الاعتبار تغيير الإحداثي x في الشاشة المرئية.

لمعالجة سطر من الحروف هناك مجموعة من المشاكل التي ينبغي مواجهتها.
أول هذا المشاكل هو ترجمة أكواد الحروف التي تمت كتابتها في جمل الشفرة إلى مثيلاتها في خارطة الرسوم التي نستعملها. لفهم المشكل تصور أننا كتبنا التعليمة التالية في الشفرة

```
video->printf(“Hello”),
```

السؤال يتعلق بكيف تخزن سلسلة الحروف Hello في ملفنا التنفيذي بعد الترجمة؟ الجواب هو
48 65 6C 6C 6F 00
تحويل الحروف الإنجليزية إلى أرقام يتم طبقا لمعيار ASCII و هذا لا يتغير ما دمنا نستعمل 1 بايت لتخزين الحروف.

الآن ماذا سيحدث إذا كتبنا التعليمة التالية؟

```
video->printf(“مرحبا”),
```

في هذه الحالة الجواب قد يختلف تبعا للمعيار الذي سيستخدمه معالج النصوص المستعمل لتشفير الحروف العربية. و هنا يكمن الاختلاف. في الحواسب الشخصية المعايير المعروفة لتشفير الحروف العربية على بايت واحد هي iso-8859-6 و cp1256 هذا الأخير هو المستعمل في ويندوز.

لكن تذكر أن أرقام الحروف التي تعطى لشاشة النصية هي مؤشرات على خارطة الرسوم. مما يلزم علينا تحويل أرقام الحروف المكتوبة في الشفرة إلى ما يكافئها في خارطة الرسوم. لهذا الغرض سنستعمل جدولا يسمى صفحة الأكواد يحتفظ بالأرقام في نظام تشفير معين مثل iso و نظيراتها في خارطة الرسوم. في هذا الجدول سنخزن أيضا بعض المعلومات الإضافية عن الحروف العربية و التي ستفيدنا فيما بعد

```
                        نحدد أنواع الحروف العربية من حيث تفاعلها مع الحروف المجاورة
                    AFT_BREAK  للحروف التي لا تلتصق مع تاليها مثل الواو، الدال...
#define AFT_BREAK       1
                      BEF_BREAK  للحروف التي لا تلتصق أيضا مع سابقها مثل "ء"
#define BEF_BREAK       2
                  JOIN للحروف التي تلتصق مع تاليها و سابقها مثل الباء، السين...
#define JOIN            3
#define NUMBER          5
#define OTHER           255

                                هذه البنية تستعمل لتخزين بيانات صفحة الأكواد
struct CodePage {
        DWORD code;
        CHAR glyph;
};


        جدول صفحة الأكواد . الحروف مرتبة و مقسمة إلى فئات. كل فئة تبدأ ببنية على شاكلة {الفئة, 255}.
CodePage iso_8859_6[] = {

// After brekqers
       {255,AFT_BREAK},
       {0xc2,0xc2},
       {0xc3,0xc3},
...
       {255,JOIN},
       {0xc6,0xc6},
       {0xc8,0xc8},
       {0xca,0xca},
…
…
    {255,OTHER}  // تشير إلى نهاية الجدول
};
```

لترجمة الأكواد نستعمل الدالة

```
static CHAR types[WIDTH];     // لتخزين أنواع الحروف
static CHAR newLine[WIDTH];   // هنا توضع الحروف بعد معالجتها
```

```
static CHAR natif[WIDTH];     // لتخزين الحروف بعد ترجمتها من صفحة الأكواد
static CHAR *line;            // يضم الحروف المطلوب معالجتها



  هذه الدالة تحول الحروف من التشفير المستعمل إلى نظيراتها ي خارطة الرسوم. و تخزن نوع
                                                        كل حرف في الجدول Types
inline void findCharAndType(CHAR c, int i) {
        for (int j = 0; j < 255; ++j) {
                CodePage *page = defcp+j;
                if(page->code == 255) {
                        إذا كان المدخل يضم 255 فهذه علامة على بداية فئة جديدة
                        types[i] = page->glyph;
                }
                else if(page->code == c) {
                        natif[i] = newLine[i] = page->glyph;
                        return;
                }
        }
}
```

الآن حللنا مشكل الترجمة. نمر إلى مشكل تبديل شكل الحرف حسب وضعه في الكلمة : البداية، الوسط... لأجل ذلك نستعمل جدولا يضم الحروف التي يتغير شكلها حسب وضعها في الكلمة و أرقام الرسوم المكافئة لكل وضع

```
struct GlyphMap {
        CHAR chr, alone, first, middle, last;
};

GlyphMap glyphMap[] = {
        {0xa8,0xa8,0xa8,0xbc,0xa8},
        {0xa9,0xa9,0xa9,0xbd,0xa9},
...
```

مهمة تغيير شكل الحروف موكولة إلى الدالة

```
inline void processShapes() {
        for (int i = 0; i < WIDTH; ++i) {
                if(isArabic(natif[i]))
                        reshapeCharAt(i);
        }

}
```

كل ما تقوم به هذه الدالة هو المرور على كل الحروف المطلوب معالجتها و نداء الدالة reshapeCharAt.

```
i inline void reshapeCharAt(int i) {
   لمعرفة شكل الحرف في الموقع i نحتاج لمعرفة نوع الحرفين المجاورين. في حال كنا في أول أو
                                                آخر السطر فقط نضع OTHER بدل نوع الحرف
        CHAR xt = (i!=0)?types[i-1]:OTHER;
        CHAR zt = (i!=WIDTH-1)?types[i+1]:OTHER;

                                                نبحث عن المدخل المكافئ في الجدول glyphMap
        int gm = getGlyphMap(natif[i]);
                                                                لا يوجد لا نغير شكل الحرف
        if(gm < 0)
                return;

                                                الآن ننظر هل بإمكاننا ربط الحرف مع الحرف السابق
        if( xt!=AFT_BREAK && xt != OTHER ) {
```

```
                        نعم، يمكننا الربط معه: إذن إما رسم الوسط أو رسم الأخير
                                                الآن هل يمكننا الربط مع الحرف التالي
                if(zt!=BEF_BREAK && zt != OTHER)
                                                نعم، يمكننا الربط : نختار رسم الوسط
                        newLine[i] = glyphMap[gm].middle;
                else
                                                لا يمكننا الربط مع التالي: نختار رسم الأخير
                        newLine[i] = glyphMap[gm].last;
        }
  إذا وصلنا هنا معنى لا يمكننا الربط مع الحرف السابق: إما رسم الأول أو رسم المنعزل
                                                نظر هل يمكننا الربط مع الحرف التالي
        else if(zt!=BEF_BREAK  && zt != OTHER) {
                                                يمكننا الربط معه، نختار رسم الأول
                        newLine[i] = glyphMap[gm].first;
        } else
                                                لا يمكننا الربط معه، نختار رسم المنعزل
                        newLine[i] = glyphMap[gm].alone;
}
```

حللنا مشكل الربط بين الحروف. المشكل التالي: الحروف التي تجمع في حرف واحد مثل "لا"

```
    هذه البنية ستتيح لنا تحويل حرفين مثل لام و ألف إلى "لا". في مثالنا سنستعمل رسمين
                                                منفصلين إذا تم ربطهما يعطيان شكلا أشبه ب "لا"
struct Pairs {
        CHAR old1, old2, new1, new2;
};


    سنستغل أيضا هذه الطريقة لرسم الحروف كبيرة الحجم مثل الشين في آخر السطر متبوعة
                                                بفراغ. سنستغل جزء من الفراغ التالي لتكميل رسم الحرف.
Pairs pairs[] = {
        {0xfa,0x96,0xfa,0xa2},
        {0xfa,0x9d,0xfa,0xa3},
…


inline void transformPairs() {
        CHAR c1, c2;
        for (int i = 0;  i < WIDTH-1; i++) {
    هذه الدالة تمر على جميع الحروف و تنادي الدالة changePair على كل حرف مع الذي
      يليه. إذا نجحت هذه الأخيرة فإنها تعيد 1 مما يعني أنه لا حاجة لمعالجة الحرف الموالي
                i+changePair(i, newLine[i], newLine[i+1]);
        }
}

inline int changePair(int ind, CHAR c1, CHAR c2) {
        if(!c2) c2 = ' ';
        for (int j = 0; pairs[j].old1!=0; ++j) {
                        if(pairs[j].old1 == c1 && pairs[j].old2 == c2) {
                                newLine[ind] = pairs[j].new1;
                                newLine[ind+1] = pairs[j].new2;
                                return 1;
                        }
        }
        return 0;
}
```

أيضا علينا حل مشكل الأقواس حيث أنه علينا تبديل اتجاهها في اللغة العربية.

```
هذه البنية تتيح لنا معرفة القوس المقابل لكل قوس آخر
struct Braces {
        CHAR in, out;
};

Braces braces[] = {
        {'{','}'} , {'}','{'} , {'[',']'} , {']','['} , {'(',')'} ,
        {')','('}, {'<','>'} , {'>','<'} ,
        {0,0}
};



inline CHAR getBrace(CHAR chr) {
        for (int i = 0;braces[i].in!=0 ; ++i) {
                if(braces[i].in == chr) {
                        return braces[i].out;
                }
        }
        return chr;
}

inline void transformBraces() {
        for (int i = 0; i < WIDTH; ++i) {
                newLine[i] = getBrace(newLine[i]);
        }
}
```

الآن يبقى لنا أصعب مشكل و هو تبديل ترتيب الحروف في السطر تبعا لوضع الكتابة. سأعطي مثالا بسيطا لتوضيح المشكل. تصور أن لدينا الكلمة التالية **"مرحبا"**. هذه الكلمة تكون مخزنة بالشكل التالي في الذاكرة

ا ب ح ر م

إذا قمنا بطباعة هذه الجملة في الوضع اللاتيني من اليسار إلى اليمين بدو تبديل حروفها ستطبع على الشكل

ابحرم

عكسا لنتصور أن لدينا كلمة لاتينية **"Hello"** و قمنا بطباعتها في الوضع العربي من اليمين إلى اليسار بدون تبديل. هذا ما سيظهر على الشاشة

olleH

المطلوب إذن هو عكس ترتيب الحروف العربية عندما نكون في الوضع اللاتيني. و عكس ترتيب الحروف اللاتينية عندما نكون في الوضع العربي. ما يزيد الأمر صعوبة ليس فقط أن السطر الواحد يمكن أن يضم خليطا من الكلمات اللاتينية و العربية. بل أن هناك أيضا مجموعة من الرموز التي لا تتبع لغة معينة مثل + و - ... هناك أيضا مشكل الفراغات بين الكلمات التي تنتمي لنفس اللغة. تصور أن لدينا جملة مثل **"السلام عليكم Hello world"** إذا لم نأخذ بعين الاعتبار الفراغات بين الكلمات من نفس اللغة فإن هذه الجملة ستطبع كما يلي

السلام عليكم world Hello

هناك أيضا مشكل الأعداد التي تكتب دائما من اليسار إلى اليمين. و هذه ليست سوى لمحة من المشاكل التي تواجه المبرمج عندما يريد دعم النص الثنائي الاتجاه Bidirectional Text.

من اجل ترتيب أفكارنا علينا إذن و ضع مجموعة من القواعد الواضحة

- الحروف العربية تعكس في الوضع اللاتيني
- الحروف اللاتينية تكس في الوضع العربي
- الفراغات و الرموز التي لا تنتمي للغة معينة تتبع – مسبقا – اتجاه الكتابة في الوضع الجاري. لكن إذا كانت محصورة بين حروف لغة معينة فإنها تتبع حروف هذه اللغة.
- الأرقام تعالج مثل الحروف اللاتينية في الوضع العربي. لكن في الوضع اللاتيني إذا كانت محصورة بين حروف عربية تظل وسط الحروف من دون أن تقلب ترتيب الكلمات العربية. مثلا إذا كان لدينا الجملة التالية في الوضع اللاتيني **"حجم الذاكرة 32568 كيلوبايت"** و لم نراعي هذه القاعدة فإن النتيجة ستكون

كيلوبايت 32568 حجم الذاكرة

الآن لننظر كيف يمكننا تنجيز هذه القواعد. أولا نحتاج لبعض الدوال المساعدة

```
                                تخبرنا ما إذا كانت الحروف عربية أو لاتينية أو أرقام
inline bool isArabic(CHAR c) {
        return ( (c >= 0xc1 && c <= 0xda) ||
                        (c>=0xe1 && c<=0xea) );
}

inline bool isLatin(CHAR c) {
        return ( (c >= 'A' && c <= 'Z') ||
                                (c>='a' && c<='z'));
}

inline bool isNumber(CHAR c) {
        return (c >= '0' && c <= '9');
}

                                                تقلب ترتيب len حرف ابتداء من buf
inline void reverse(CHAR *buf, int len) {
        CHAR *head, *tail;

        head = buf;
        tail = buf+len;
        while (head < tail) {
                char tmp = *head;
                *head = *tail;
                *tail = tmp;
                head++;
                tail--;
        }
}

                                تقلب ترتيب الحروف العربية و ما داخلها من فراغات و رموز
                                                الأرقام تظل بترتيبها الأصلي
inline void reverseAr(CHAR *buf, int len1) {
                                        أولا نقلب ترتيب كل الحروف بما فيهل الأرقام
        reverse(buf, len1);
                                    ثم نبحث عن الأرقام لإعادتها إلى ترتيبها الأصلي
        for (int i = 0; i < len1; ++i) {
                if(isNumber(buf[i])) {
                        int x = i;
          نمر على جميع الأرقام و ما يمكن أن تحتويه داخلها من فراغات و فواصل و نقط
                        for (; x < len1 &&
                          ( isNumber(buf[x]) || buf[x] == ' '
                            || buf[x] == '.' || buf[x] == ',')
                                ; ++x);
                            بعد الخروج من الحلقة x-1 تشير إلى آخر رقم أو نهاية الجملة
```

```
                        reverse(&buf[i], x-i-1);
                        i = x-1;
                }
        }
}
```

الشفرة الرئيسية تكمن في الطريقة FontMap::process التي تنادى من Video->updateScreen لمعالجة سطر من سطور الشاشة المنطقية قبل نقله على الشاشة المرئية.

```
                                                تقوم بمعالجة سطر من الحروف.
                                                buf مؤشر على السطر المطلوب معالجته
                                                mode الوضع الجاري للكتابة
CHAR* FontMap::process(CHAR *buf, WriteMode mode) {
                                                نخزن buf في المتغير العام line
        line = buf;
        CHAR c;
                                        1- ترجمة صفحة الأكواد + معرفة أنواع الحروف
        for (int i = 0; i < WIDTH; ++i) {
                c=line[i];
                if(c < 127) {
                        natif[i] = newLine[i] = c;
                        types[i] = OTHER;
                } else
                        findCharAndType(c, i);
        }

                        2- اختيار الشكل الصحيح للحروف حسب أوضاعها داخل الكلمة
        processShapes();

                                                3- معالجة الحروف المترابطة مثل لام ألف
        transformPairs();

        if(mode == Arabic) {
                                        4- في الوضع العربي نحتاج لتبديل اتجاه الاقواس
                transformBraces();

                                                5- قلب ترتيب الحروف اللاتينية و الأرقام
                processBidiAr();
        }
        else
                                        4- في الوضع اللاتيني نحتاج فقط لقلب الحروف العربية
                processBidiLat();

                                                نعيد السطر لطباعته في الشاشة المرئية
        return newLine;
}

                                                تعالج نصا ثنائي الاتجاه في الوضع العربي
inline void processBidiAr() {
        int endLatin = 0;    // لتخزين قيمة آخر حرف لاتيني

        for (int i = 0; i < WIDTH; ++i) {

                                        عند مصادفة بداية كلمة لاتينية أو عدد نقلب ترتيبها

                if (isLatin(natif[i]) || isNumber(natif[i])) {
                        int j = endLatin = i;
                                                نستمر في الحلقة حتى أول حرف عربي
                        for (; j < WIDTH && !isArabic(natif[j]) &&
                                natif[j]!=0; ++j) {
```

```
                                if(isLatin(natif[j]) || isNumber(natif[j]))
                                        endLatin = j;
                        }
                                                endLatin تشير إلى موقع آخر كلمة لاتينية
                                                        j-1 تشير إلى موقع أول كلمة عربية
                        reverse(&newLine[i], endLatin-i);
                        i = j-1;
                }
        }
}

                                                معالجة النص في الوضع اللاتيني
inline void processBidiLat() {
        int endArabic = 0;  // لتخزين موقع آخر حرف عربي

        for (int i = 0; i < WIDTH; ++i) {
                if(isArabic(natif[i])) {
                        int j = endArabic = i;
                        for (; j < WIDTH && !isLatin(natif[j])
                        && natif[j]!=0 ; ++j) {
                                if(isArabic(natif[j]))
                                        endArabic = j;
                        }
                                                endArabic تشير إلى موقع آخر حرف عربي
                                                        j-1 تشير إلى موقع أول حرف لاتيني
                نستعمل النسخة العربية من reverse لقلب ترتيب الحروف مع الحفاظ على ترتيب الأرقام
                                                                داخل الجملة العربية
                        reverseAr(&newLine[i], endArabic-i);
                        i = j-1;
                }
        }
}
```

الآن أتممنا تنجيزنا للتعريب في الشاشة النصية بشكل مقبول. قبل المرور إلى التجربة كلمة أخيرة بخصوص صفحات الأكواد المستعملة: في الشفرة المرافقة للدرس أدمجت صفحتين للأكواد هما iso-8859-6 و cp1256. بما أنني أعمل في Eclipse في بيئة ويندوز (أتمنى ألا يغضب مني مستعملو لينوكس). و محرر Eclispse يتيح لي فقط الكتابة ب cp1256. فجميع الجمل العربية التي كتبت مشفرة بهذا المعيار. لذلك فبرنامج النواة يستعمل – مسبقا – cp1256. إذا أردت تجريب الشفرة و إضافة الجمل إليها باستعمال iso-8859-6 فقد قمت بإدخال التعليمة التالية في مصنع الفئة FontMap

```
FontMap::FontMap()
{
        #ifdef ISO
                defcp = iso_8859_6
        #else
                defcp = cp1256;
        #endif
}
```

هذا يعني أنه عند الترجمة عليك أن تعرف ISO. بإضافة DISO– على تعليمة الترجمة ب gcc.  لا تنسى أيضا تبديل الجمل الحالية باستعمال محرر يدعم iso-8859-6.

منظر العربية دائما مميز و إن كان على الشاشة النصية. ما رأي عشاق لغة القرآن؟


ndows - Display
USER
Copy
Paste
snapshot
CONFIG
Reset
Power
السلام عليكم التحميل تم من GRUB والحمد لله
حجم الذاكرة = 32768 كيلوبايت
تم الإقلاع من القرص المرن A
إعداد المخمص الحقيقي للذاكرة.....[نجح]
بدء تجريب المخمص الحقيقي
تجريب التخميص
تم تخميص 2423 صورة صفحة
تم تحرير 5633 صورة صفحة
44 ثانية مرت منذ الإقلاع
enables mouse
A:
B:
HD:0-M
NUM
CAPS
SCRL

المراجع

1-

Memory Management: Algorithms and Implementation in C/C++
by Bill Blunden ISBN:1556223471
Wordware Publishing © 2003 (360 pages)