# الدرس السابع : مخصص الذاكرة في النواة
# Kernel Memory Allocator

في الدرسين السابقين تمكنا من إرساء الدعائم الأولى لمخصص الذاكرة. بحيث أصبح في إمكاننا الآن تخصيص صور صفحات من الذاكرة الحقيقية و ترسيمها أينما شئنا في الفضاء الافتراضي.

في هذا الدرس، سنتمم مخصص الذاكرة في النواة. بعد هذا الدرس، سيصبح في إمكاننا استعمال المعاملين new/delete في شفرة النواة مما سيسهل لنا لاحقا برمجة باقي مكونات النواة.

## 1- تقديم مخصص النواة المستعمل

كما رأينا في الدرس الخامس فإن مهمة مخصص الذاكرة تتلخص في توفيره لقطع من الذاكرة حسب الطلب مع مراعاة ألا تتقاطع القطع المخصصة مع بعضها البعض. و كما رأينا في الدرس السادس، فبعد تنشيط الفهرسة أصبحت النواة تشتغل في الفضاء الافتراضي. وقلنا أيضا أن هذا الفضاء يقسم عادة لقسمين : قسم تقيم فيه النواة و قسم تتناوب عليه برامج المستعمل. في حالتنا فالنواة ستستعمل الـ1جيغ الأولى من الفضاء الافتراضي. للتذكير نورد هنا خارطة للفضاء الافتراضي:


لتخصيص الذاكرة سنستعمل المساحة التي تبدأ مباشرة بعد نهاية النواة أو ما يعرف بالفناء Heap.

### اختيار سياسة التخصيص

كما رأينا سابقا هناك عدة طرق لتنجيز مخصص الذاكرة. و نجاعة مخصص الذاكرة مرتبطة أساسا بالوسط الذي يشتغل فيه و يتعامل معه. فيما يخص النواة، هناك مجموعة من الخصائص تجعل مشكل تخصيص الذاكرة مختلفا نوعا ما عن المخصصات التقليدية التي تستعمل عادة في برامج المستعمل.

فيما يلي بعض المميزات المطلوبة في مخصص النواة:

- سرعة الأداء، حيث أن طلب التخصيص يأتي من مكونات حساسة مثل منسق المهام Task Sheduler أو سائقي الفروع. فإن التخصيص يجب أن يتم في أسرع وقت ممكن، يستحسن أن تكون كلفة التخصيص ثابتة أي مستقلة عن حجم الذاكرة وعدد القطاعات الحرة.
- ترشيد استعمال الذاكرة، حيث أن تخصيص مساحات من الذاكرة الافتراضية تتبعها في أغلب الحالات تخصيص مساحات من الذاكرة الحقيقية (بعكس برامج المستعمل حيث تستعمل تقنيات مثل التخصيص عند الولوج Demand Paging و التبديل Swap). فإنه ينبغي مراعاة ألا تبقى مساحات كبيرة من الذاكرة غير مستعملة و مستهلكة للذاكرة الحقيقية.
- ينتج عما سبق أنه ينبغي تقليل الانقسام (الخارجي و الداخلي – أنظر الدرس الخامس) إلى أقصى حد ممكن.

كالعادة هناك عدة حلول لتحقيق هذه الأهداف. فيما يلي سنقدم نسخة مبسطة من أحد هذه الحلول: يتعلق بالأمر بالمخصص المستعمل في نظامي سولاريس Solaris و لينوكس و يعرف بمخصص السلاب Slab Allocator. هذا الأخير يتوفر على عدة مزايا تمكننا من تحقيق الأهداف السابقة: سرعة أداء جيدة و استعمال فعال للذاكرة.

في نواتنا سننجز مخصص السلاب على مرحلتين أو طبقتين Layers:

1- مخصص المناطق Region Allocator: يقوم بتخصيص قطع من فناء النواة بأحجام مسطرة على 4كلب. بمعنى أن هذا المخصص يقوم فقط بتخصيص الصفحات و ليس أحجام اعتباطية.
2- مخصص السلاب Slab Allocator: هذا الأخير يمكننا من تخصيص أحجام أصغر من 4كيلب. يعتمد في عمله على مخصص المناطق. في واقع الأمر مخصص السلاب يتكون من مجموعة مخصصات: كل واحد يتكفل بتخصيص حجم معين.

لفهم الأمور جيدا لا بد من الرجوع إلى بعض الأسس.

مخصص السلاب يصنف في فئة المخصصات التجميعية Segregated Allocators (رأينا مثالا لأحد هذه المخصصات في الدرس الخامس و هو المخصص التجاوري Buddy Allocator). كما رأينا فإن هذه المخصصات تقوم – في بادئ الأمر – بتقسيم الفناء المتوفر لها إلى مجموعة من القطع من أحجام معينة. هكذا تكون لدى المخصص عدة قوائم بالأحجام الممكن تخصيصها: مثلا قائمة لقطع بحجم 8بايت، قائمة لقطع بحجم 16بايت و هكذا... عندما يتم طلب حجم – مثلا 13 بايت – يتم استعمال قائمة الحجم الأقرب – في مثالنا 16بايت -. هكذا بدل أن تكون لدينا قائمة واحدة طويلة من عدة أحجام اعتباطية و مبعثرة و يتم البحث على طول هذه القائمة حتى العثور على الحجم المناسب كفاية، فإنه يتم التخصيص مباشرة من الحجم الأقرب مما يقلص كلفة البحث بنسبة مهمة.

الآن عيب هذه المخصصات في أنه يتم دائما تضييع مساحات من الذاكرة – في مثالنا فوق تم طلب 13 بايت و خصصت 16 بايت مما تسبب في ضياع 3 بايت-. هذا لأن المخصص لا تكون لديه قبلا معرفة بالأحجام التي سيتم استعمالها.

مخصص السلاب صمم خصيصا لتفادي هذه العيوب. بدلا من صنع قوائم – أو مخزونات - بأحجام من اختياره، فإن مستعمل هذا المخصص هو الذي يختار الأحجام التي ينبغي صنع مخزونات لها: مثلا نفترض أنه في نواتنا سنحتاج بكثرة لصنع عناصر Objects من فئة بحجم 14بايت، و عناصر من فئة أخرى بحجم 22بايت. سنخبر إذن مخصص السلاب بصنع مخزونين: واحد من حجم 14بايت و آخر بحجم 22بايت. عندما نريد تخصيص عنصر من الفئة الأولى سنطلب من المخصص أن يمدنا بأحد القطع من القائمة الأولى. بما أن المخزون يتوفر قبلا على عدة قطع فارغة من حجم هذه الفئة فإن المخصص لن يضطر للبحث و سيعيد لنا أول قطعة متوفرة في القائمة.

هذه الميزة تجعل مخصص السلاب مناسبا جدا لأغراض النواة: حيث سنستعمل أنواع فئات قليلة لكن بأعداد كبيرة. بالطبع هناك أيضا أحجام لا يمكن تصنيفها لكنها حالات نادرة بالمقارنة مع الحالة الأولى و سنرى لاحقا كيف يمكن استغلال مخصص السلاب لتنجيز مخصص عام يقبل تخصيص جميع الأحجام.

عندما يطلب تخصيص مساحة من أحد المخزونات، يحدث أحد الأمرين:
1- هناك بعد قطع حرة متوفرة في المخزون الخاص بهذه المساحة، في هذه الحالة يقوم المخصص بإرجاع أحد هذه القطع.
2- لم تعد هناك قطع متوفرة، يقوم المخصص بتخصيص صفحة أو عدة صفحات متجاورة – لاحقا سنقول منطقة - باستعمال مخصص المناطق، يقسم هذه المنطقة إلى عدة قطع متساوية من الحجم المحدد ثم يعيد أول قطعة حرة من هذه المنطقة.

مما سبق يتضح أن كل مخزون قد يتوفر على عدة مناطق (المنطقة=صفحة أو عدة صفحات متجاورة في الذاكرة الافتراضية). هذه المناطق تعرف بالسلابات Slabs و هي الأصل في تسمية المخصص. كل مخزون إذن يتوفر على مجموعة من السلابات وكل سلاب يتوفر على مجموعة من القطع – لاحقا سنقول عناصر حيث أننا مثل من يخزن عناصر لاستعمالها لاحقا -.

هناك ثلاث حالات للسلاب: قد يكون فارغا بمعنى أن كل العناصر داخله حرة – قد يكون فارغا جزئيا بمعنى انه يتوفر على عناصر مستعملة و عناصر حرة و قد يكون مليئا بمعنى أنه لا يتوفر على عناصر حرة.

كل عمل مخصص السلاب تكمن في إدارته لقوائم تضم السلابات التي توجد من نفس الحالة: بمعنى قائمة للسلابات الفارغة، قائمة للسلابات الجزئية و أخرى للسلابات المليئة. مع التأكد من وضع كل سلاب في القائمة الصحيحة إذا تغيرت حالته بعد تخصيص أو تحرير أحد العناصر.

فيما يلي سنقدم تفاصيل تنجيز مخصص المناطق و مخصص السلاب في نواتنا. أريد أن ألفت الانتباه أنه نظرا لحجم الشفرة فإنني لن أدرج الشفرة بأكملها هنا كما درجت العادة و سأكتفي بإدراج القطع المهمة منها هنا. يمكن للقارئ الرجوع إلى ملفات الشفرة المرافقة للدرس (الشفرة مرفقة بتعليقات وافية لتسهيل الفهم).

## 2- التنجيز

### 2-1 فئة لتدبير القوائم

سنحتاج من أجل تنجيز هذا المخصص لتدبير مجموعة من القوائم المتصلة Linked Lists. لذلك علينا قبلا توفير مجموعة من الدوال للتعامل مع القوائم المتصلة. من اجل ذلك تم إنشاء فئة DLList ( Doubly Linked List أي أن كل عنصر يتوفر على مؤشرين أو رابطين واحد للعنصر السالف و آخر للعنصر اللاحق).

سنحتاج لإدارة قوائم لأنواع مختلفة من البيانات. بطبيعة الحال لن نقوم بإنشاء فئة خاصة لكل نوع. بل سنقوم باستغلال وظيفة النماذج Tempates في سي++ لصنع فئة تستطيع التعامل مع جميع انواع البيانات.

```
template<class T>
class DLList {
      T* head;     //رأس القائمة

      // لتسهيل الاستعمال في الحلقات
      T* _next;
      bool iterate;

      // عدد العناصر في القائمة
      unsigned int count;
...
```

عندما نريد صنع قائمة لنوع معين مثلا VRegion فإنه يكفي تحديد قيمة النموذج T.

```
DLList<VRegion> regionList;
```

فتصبح لدينا فئة جديدة DLList قادرة على التعامل مع النوع VRegion.

لتنجيز الفئة تم اختيار قائمة دائرية بمعنى أن رأس القائمة يؤشر على العنصر التالي، هذا الاخير يؤشر على العنصر الذي بعده وهكذا إلى آخر عنصر في القائمة، هذا الأخير بدوره سيؤشر على رأس القائمة مما يعطينا شكلا أشبه بالحلقة الدائرية. هذا سيمكننا من إدخال عناصر في رأس القائمة و ذيلها في وقت ثابت.

DLList تتوفر على طرق لإدخال عناصر و إزاحتها من رأس القائمة، طرق لإدخالها و إزاحتها من ذيل القائمة، و طرق أخرى لإدخالها او إزاحتها من أي مكان داخل القائمة

```
void push(T* item);      إضافة عنصر إلى رأس القائمة
T* pop();                إزاحة العنصر من رأس القائمة

void queue(T* item);     إضافة عنصر إلى ذيل القائمة
T* dequeue();            إزاحة عنصر من ذيل القائمة
```

```
                                                    beforeItem إدخال item قبل
void insertBefore(T* beforeItem, T* item);
                                                     afterItem إدخال item بعد
void insertAfter(T* afterItem, T* item);

void remove(T* item);   إزاحة عنصر من وسط القائمة
```

الفئة توفر أيضا على طرق لعبور الفئة من الأول إلى الآخر

```
T* reset();         تعيد أول عنصر و تبدأ عملية العبور
bool atEnd();       تخبرنا ما إذا وصلنا لآخر القائمة
T* next();          تعيد لنا العنصر التالي أثناء العبور
```

مثال لعبور قائمة من VRegion

```
VRegion *tmp = regionList.reset();
while(!regionList.atEnd()) {
      …
      tmp = regionList.next();
}
```

لتنجيز المخصص سننشئ الفئات التالية:

| | |
|---|---|
| VRegion | لتمثيل المناطق التي يتم تخصيصها من قبل مخصص المناطق |
| KHeapStorage | مخصص المناطق |
| SelfBuffer | لتمثيل السلابات |
| ObjCache | مخصص السلابات |

## 2-2 مخصص المناطق

الفئة VRegion تمثل منطقة من الفضاء الافتراضي

```
class VRegion
{

private:
      virtAddr_t _start;      عنوان بداية المنطقة
      DWORD _size;            حجم المنطقة بالبايت
      DWORD _attribs;         خصائص المنطقة
      SelfBuffer *buffer;     السلاب الذي يوجد به هذا العنصر أنظر الشرح لاحقا
```

الفردين _start و _size تباعا لتحديد حدود المنطقة. _attribs لتحديد خصائص المنطقة: حاليا سنستعمل شارتين VR_USED للمناطق المستعملة و VR_MAP للمناطق التي تتوفر على صور صفحات مرسمة في حدودها.

الفرد الأخير buffer يحتاج إلى بعض الشرح عن كيفية عمل مخصص المناطق.

الفئة KHeapStorage مسئولة عن تدبير فناء النواة بتخصيص و تحرير مناطق من هذا الفناء. كل منطقة، كما رأينا بحجم صفحة أو عدة صفحات. عندما نقوم بتخصيص منطقة فإنه كما سنرى قد ينتج عن ذلك صنع عنصر جديد من الفئة VRegion لتمثيل هذه المنطقة. السؤال هو كيف سنصنع هذا العنصر الجديد؟ نحن الآن بصدد صنع دعامة ل new و delete. و نحتاج قبلا لخدمات هذين المعاملين لتنجيز هذه الدعامة. كما نرى فإن مشكل البيضة و الدجاجة يعود دائما في برمجة النواة. بلغة الكمبيوتر هذا المشكل يعرف بالإيقاع Bootstarppig. كيف إذن سنوقع بمخصص الذاكرة و نخرج من هذه الحلقة المفرغة؟

الحل المقترح هنا هو استعمال السلابات. انتبه لن نستعمل مخصص السلاب لأن هذا الأخير يحتاج قبلا لمخصص المناطق. هذا الأخير سيقوم بإدارة السلابات المعتمدة لصنع عناصر VRegion بنفسه. هذه الطريقة ممكنة لأن الفئة SelfBuffer صممت لتكون مستقلة عن مخصص السلابات ObjCache.

لكن هذا لا يحل لنا المشكل تماما، كيف سنصنع العناصر SelfBuffer لتمثيل السلابات المستعملة؟

الجواب هو : داخل المناطق نفسها التي سيستعملها السلاب (أعلم أن الأمور قد تبدو معقدة، لكن هذا التعقيد لا مناص منه لطبيعة المشاكل التي نواجهها في هذه المرحلة).

قلنا أن السلاب سيستعمل منطقة لتوفير مخزون العناصر الحرة. سنضع العنصر SelfBuffer الممثل لهذا السلاب في أول هذه المنطقة و استغلال ما تبقى لصنع المخزون.


لأجل ذلك سنستعمل صيغة خاصة للمعامل new تعرف ب Placement new. هذه الأخيرة تختلف عن الصيغة التقليدية بأنها تتيح لنا تحديد العنوان الذي سنضع فيه العنصر الذي نريد تخصيصه مباشرة مما يجنبنا الاحتياج إلى نداء مخصص قبلي لتخصيص المساحة اللازمة لصنع العنصر. لصنع عنصر مثلا في العنوان address يمكننا كتابة

```
SelfBuffer *buffer = new(address) SelfBuffer(...);
```

لاحظ استعمال (address) لإمداد المعامل new بالعنوان الذي سيوضع فيه العنصر المخصص. إذا أمددنا new بالعنوان فلا داعي بعد لتخصيص مساحة ما و كل ما يتبقى فعله لـnew هو نداء المصنع Constructor المناسب مع إعطاء العامل this قيمة address.

لكن لاستعمال هذه الصيغة لنداء new ينبغي لنا تعريفها قبلا في الفئة SelfBuffer.

```
void *operator new(size_t size, void *location){
	return location;
};
```

لا حظ أن دالة المعامل new تأخذ بالإضافة للعامل التقليدي size عاملا إضافيا و هو العنوان الذي تم إمداد new به كما رأينا سابقا.

قبل أن نمر إلى مخصص المناطق KHeapStorage سنلقي نظرة على عمل الفئة SelfBuffer.

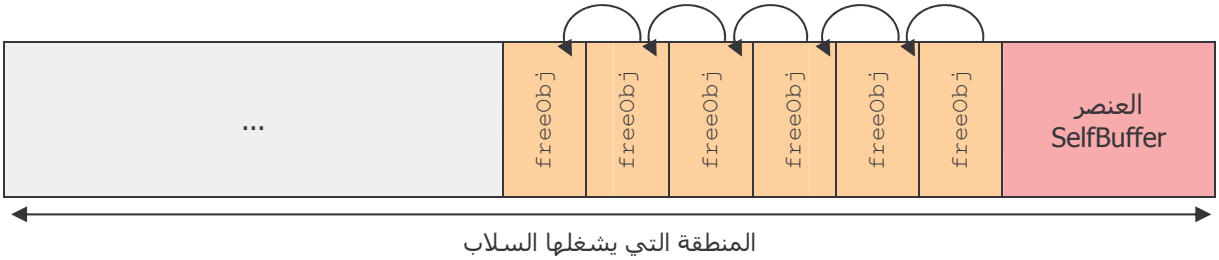
```
class SelfBuffer
{	DWORD size;		حجم العناصر المراد تخزينها
	DWORD nbFree;		عدد العناصر الحرة
	DWORD maxFree;		عدد العناصر كاملة
	DWORD nbAlloc;		عدد العناصر المستعملة
	freeObj *firstFree;		مؤشر على أول عنصر حر
...

للاستعمال من الفئة DLList
SelfBuffer *next, *prev;
لتخزين معلومة قد تفيد من طرف مستعملي هذه الفئة
DWORD info;
```

لربط العناصر الحرة مع بعضها البعض داخل السلاب سنستعمل البنية freeObj.

```
struct freeObj {
	freeObj *next;
};
```

هذه البنية ستخزن في كل عنصر حر مما يصنع لنا قائمة متصلة من العناصر الحرة. عندما نريد تخصيص عنصر من السلاب يكفي إزاحة رأس القائمة و تخصيصه. عكسا عندما نريد تحرير عنصر مستعمل يكفي إضافته إلى رأس القائمة.




مصنع الفئة Selfbuffer يقوم بصنع العناصر الفارغة في العنوان الذي يحدد له

```
                                        حجم العناصر المراد تخزينها objSize
                                    حجم المساحة المتوفرة لصنع العناصر بداخلها maxSize

SelfBuffer::SelfBuffer(DWORD objSize, DWORD maxSize):


                                    على objSize أن يكون كبيرا كفاية لاحتواء freeObj

size((objSize < sizeof(freeObj))?sizeof(freeObj):objSize)
{
                                            أول عنصر حر يأتي بعد نهاية SelfBuffer

	firstFree = (freeObj *)((DWORD)this+sizeof(SelfBuffer));

                                        حساب عدد العناصر الممكن تخزينها في هذا السلاب

	nbFree = maxFree = (maxSize-sizeof(SelfBuffer))/size;

                                                    صنع قائمة العناصر الحرة
	freeObj *curFree = firstFree;
	for (int i = 1; i < maxFree; ++i) {
		freeObj *nextFree = (freeObj*)((DWORD)curFree + size);
		curFree->next = nextFree;
		curFree = nextFree;
	}
	info = 0;
}
```

الطريقة alloc تقوم بإرجاع أول عنصر حر في القائمة

```
void *SelfBuffer::alloc() {
	if(nbFree) {
		freeObj *ret = firstFree;
```

```
            firstFree = firstFree->next;
            nbFree--; nbAlloc++;
            return ret;
      }
      return (void*)0;
};
```

عكسا الطريقة free تقوم يتحرير عنصر مستعمل

```
void SelfBuffer::free(void *object) {
      freeObj *toFree = (freeObj *)object;
      toFree->next = firstFree;
      firstFree = toFree;
      nbFree++; nbAlloc--;
};
```

بعد أن رأينا كيف يمكن تخزين العناصر في السلابات سنمر إلى الفئة KHeapStorage. هذه الأخيرة مسئولة عن تدبير فضاء النواة و تخصيص مساحات بحجم صفحة أو عدة صفحات.

الفئة KHeapStorage تضم قائمتين من VRegion قائمة بالمناطق المستعملة و أخرى بالمناطق الحرة

```
DLList<VRegion> regUsed;
DLList<VRegion> regFree;
```

القائمة regFree تحتفظ بالمناطق الحرة مرتبة حسب العناوين مما يسهل عملية دمج القطاعات الحرة المتجاورة كما سنرى فيما بعد.

الفئة تضم أيضا ثلاث قوائم لإدارة السلابات المستعملة لتخزين VRegion. تذكر أن الفئة KHeapStorage لا تستطيع الاعتماد على ObjCache لتخصيص العناصر. لذلك ستقوم بإدارة السلابات بنفسها.

```
DLList<SelfBuffer> sbEmpty;   قائمة السلابات الفارغة
DLList<SelfBuffer> sbFull;    قائمة السلابات المليئة
DLList<SelfBuffer> sbPart;    قائمة السلابات الجزئية
DWORD _nbFree;                عدد العناصر الحرة في جميع السلابات
```

الطريقة initHeap تقوم بتهيئة فناء النواة وقوائم السلابات

```
int KHeapStorage::initHeap(virtAddr_t heapBase, DWORD heapSize,
                  PhysMManager *pmem, Pager *pager)
{     KASSERT(heapBase && heapSize && pmem && pager);

      _heapBase = heapBase;   عنوان بداية الفناء
      _heapSize = heapSize;   حجم الفناء بالبايت
      _pmem = pmem;           مؤشر على مخصص الذاكرة الحقيقية
      _pager = pager;         مؤشر على مدبر الفهرسة
      _nbFree = 0;

                                                تخصيص صورة صفحة لصنع أول سلاب
      physAddr_t paddr = _pmem->allocFrame(0);
      if(!paddr)
            return NOMEM;
                                                ثم ترسيمها في أول الفناء
      int err = _pager->map(_heapBase, paddr, KERNEL_WRITE);
      if(err) {
            _pmem->freeFrame(paddr);
            return INVPARAM;
```

```
        }
        _heapFree = _heapSize;
  الدالة selfSplit تقوم بتقسيم الفناء إلى منطقتين: الأولى تضم المنطقة المستعملة من
                                طرف السلاب و الثانية تضم المنطقة الحرة المتبقية
        selfSplit(_heapBase, _heapSize, null);
        return SUCCESS;
}
```

للإيقاع بأول سلاب SelfBuffer و أول منطقتين VRegion نستعمل الدالة selfSplit هذه الأخيرة تقوم بصنع سلاب في أول صفحة في الفناء

```
SelfBuffer *sb = new (_heapBase) SelfBuffer(sizeof(VRegion), VRBUF_SIZE);
```

ثم صنع منطقتين: الأولى تمثل المنطقة التي يشغلها السلاب نفسه و الثانية تمثل المنطقة الحرة المتبقية بعد السلاب

```
                                                تخصيص عنصر من السلاب
void *tmp = sb->alloc();

                                ثم استعماله لتخصيص VRegion تمثل المنطقة المشغولة من السلاب

VRegion *sbuf = new(tmp) VRegion(start, VRBUF_SIZE, 0, sb);
regUsed.push(sbuf);
sbuf->_attribs |= VR_USED;
_heapFree -= VRBUF_SIZE;

                                            نفس الشيء للمنطقة الحرة المتبقية بعد السلاب
tmp = sb->alloc();
VRegion *remaind = new(tmp) VRegion(start + VRBUF_SIZE,
                                    size - VRBUF_SIZE, 0, sb);
                                    نخزن مؤشرا على المنطقة التي يشغلها السلاب في الفرد info
sb->info = (DWORD) sbuf;
```

الآن تم الإيقاع بأول سلاب و أول منطقة حرة. و يمكننا استعمال باقي الطرق على أسس هذه البيانات الأولى.

الطريقة alloc تقوم يتخصيص منطقة من الفضاء الحر.

```
VRegion *KHeapStorage::alloc(DWORD size, DWORD attribs)
```

هذه الأخيرة، بعد التأكد من صحة العوامل المقدمة لها، تبدأ بنداء الدالة findFirstFree للبحث عن منطقة حرة كافية لاحتواء الحجم size. findFirstFree تعتمد على طريقة أول متسع First fit (أنظر الدرس الخامس). كل ما تقوم به هو مسح قائمة القطاعات الحرة regFree للعثور على متسع كاف لاحتواء size

```
inline VRegion* findFirstFree(DWORD size) {
                                                نبدأ بعبور قائمة المناطق الحرة
        VRegion *free = regFree.reset();
        while(!regFree.atEnd()) {
                if(free->_size == size) {

    إذا حالفنا الحظ و كان حجم المنطقة الحالية نفس الحجم المطلوب نعيدها مباشرة بعد
                                                نقلها إلى قائمة المناطق المستعملة

                        regFree.remove(free);
                        regUsed.push(free);
                        return free;
                } else if(free->_size > size) {
```

```
أما إذا كان حجم المنطقة الحالية أكبر فيجب علينا تقسيمها إلى قسمين: و احدة بالحجم المطلوب يتم تخصيصها و أخرى بالحجم المتبقي. هذا سيتلزم صنع عنصر جديد VRegion

                    DWORD newStart = free->_start+size;
                    DWORD newSize = free->_size - size;
                    VRegion *rg = createVRegion(free->_start, size, 0);
                    if(rg) {
                          free->_start = newStart;
                          free->_size = newSize;
                          regUsed.push(rg);
                          return rg;
                    } else
                          return null;
              } else {
                    free = regFree.next();
              }
        }
        return null;
};
```

بعد أن تتم findFirstFree عملها تعود إلى alloc بمؤشر على منطقة بالحجم المطلوب. حينذاك تقوم هذه الأخيرة – في حالة ما تم تحديد الشارة VR_MAP – بتخصيص صور صفحات حقيقية و ترسيمها في حدود المنطقة المخصصة

```
if(attribs & VR_MAP) {
      for (DWORD start = rg->_start; start < rg->end();
                              start += PAGE_SIZE) {

                        نقوم بتخصيص صور الصفحات و ترسيمها في حدود المنطقة المخصصة

              physAddr_t paddr = _pmem->allocFrame(0);

انتبه في حال فشل تخصيص أو ترسيم صور الصفحات الحقيقية علينا إلغاء جميع التخصيصات السابقة من الذاكرة الحقيقية

              if(!paddr || _pager->map(start, paddr, KERNEL_WRITE)) {
                    free(rg);
                    if(paddr)
                          _pmem->freeFrame(paddr);
                    start -= PAGE_SIZE;
                    while(start >= rg->_start) {
                          _pager->unmap(start, true);
                          start -= PAGE_SIZE;
                    }
                    return null;
              }
        }
}
```

هناك أيضا عمل ينبغي ل alloc أن تنجزه بعد كل تخصيص. إذا أصبحت جميع السلابات مليئة فإنه لا يمكننا بعد صنع عناصر VRegion جديدة. لذلك ينبغي صنع سلاب جديد، لكن هذه العملية تتطلب صنع عنصر VRegion لتمثيل المنطقة التي يشغلها السلاب. أي أنه ينبغي توفر عنصر VRegion واحد على الأقل في السلابات القديمة. و هذا هو ما تقوم به الدالة alloc

```
if(_nbFree < 2) {
```

```
                                        إذا وصلنا إلى الحد الأدنى علينا صنع سلاب جديد

      VRegion *vr = findFirstFree(VRBUF_SIZE);
      ...
      SelfBuffer *sb = new (vr->_start)
                              SelfBuffer(sizeof(VRegion), VRBUF_SIZE);
      sb->info = (DWORD) vr;
      _heapFree -= VRBUF_SIZE;
      sbEmpty.push(sb);
      _nbFree += sb->freeObjs();
}
```

عكسا الطريقة free تقوم بتحرير منطقة مستعملة

```
void KHeapStorage::free(VRegion *region) {
      KASSERT(region);
      DWORD size = region->_size;

                                        إزاحة region من قائمة المناطق المستعملة

      regUsed.remove(region);

                                                    مسح الشارة VR_USED

      region->_attribs &= ~(VR_USED);

      if(region->_attribs & VR_MAP) {

                              إذا تم ترسيم صور صفحات في هذه المنطقة علينا تحريرها أيضا

            for(DWORD start = region->_start; start < region->end();
                              start += PAGE_SIZE)
                        _pager->unmap(start, true);

                                        مسح الشارة VR_MAP بعد تحرير صور الصفحات

            region->_attribs &= ~(VR_MAP);
      }

    بعد تحرير المنطقة علينا إضافتها إلى قائمة المناطق الحرة. الدالة insertVRFree
  تقوم بإدخال region إلى القائمة بترتيبها الصحيح حسب العناوين. تقوم أيضا بدمج
                                        region إذا كانت المنطقتان المجاورتان حرتين

      insertVRFree(region);
      _heapFree += size;
}
```

## 3-2 مخصص السلابات

الفئة ObjCache تقوم بصنع مخزون من العناصر بحجم معين. و هي الواجهة المباشرة لمستخدمي مخصص السلاب. قبل أن نمر إلى تنجيز الفئة سنوضح مثالا لكيفية استعمالها. نفرض مثلا أننا نريد أن نصنع مخزونا لعناصر من فئة ما Thraed. أولا علينا صنع عنصر جديد ObjCache لإدارة مخزون هذه الفئة

```
ObjCache *threadCache = new ObjCache("threadCache", sizeof(THread),
                                    VRBUF_SIZE);
```

العامل الأول هو إسم المخزون المراد صنعه، العامل الثاني يمثل حجم العناصر المراد تخزينها أما العامل الثالث فهو حجم السلابات التي سيستعملها المخزون.

بعد ذلك يمكن استعمال المخزون threadCache لتخصيص مساحات كافية لصنع عنصر جديد Thread

```
void *address = threadCache->alloc();
Thread *thread = new(address) Thread(...);
```

لتحرير العنصر thread

```
threadCache->free(thread);
```

يمكن أيضا تدمير المخزون threadCache ببساطة

```
delete threadCache;
```

فيما يلي تعريف للفئة ObjCache

```
class ObjCache
{     char name[20];                إسم المخزون
      DWORD _objSize;               حجم العناصر المخزنة
      DWORD _bufSize;               حجم السلابات

      DLList<SelfBuffer> sbEmpty;   قائمة السلابات الفارغة
      DLList<SelfBuffer> sbFull;    قائمة السلابات المليئة
      DLList<SelfBuffer> sbPart;    قائمة السلابات الجزئية

      DWORD _totObjSize;            الحجم الفعلي المخزن أنظر تحت
      DWORD _nbFree;                عدد العناصر الحرة في جميع السلابات
...
```

المهمة الأساسية ل ObjCache تكمن في إدارة قوائم السلابات بشكل صحيح. إي وضع السلابات في القائمة الصحيحة تبعا لحالتها.

ملاحظة صغيرة بخصوص الفرد totObjSize_. هذا الأخير يمثل الحجم الذي سيخزن فعليا في السلاب و الذي قد يكون اكبر من الحجم المطلوب تخزينه: لماذا؟ أولا لتمديد الأحجام لتكون مضاعفا ل 4بايت، حيث أداء المعالج يصبح أفضل. ثانيا لأن ObjCache تقوم بإضافة رأس head على كل عنصر مخزن. هذا الرأس يضم مؤشرا على السلاب الذي استعمل لتخزين العنصر. هكذا عندما يطلب تحرير عنصر في عنوان معين، تقوم ObjCache بالعثور على السلاب الذي يخزن هذا العنصر فقط باستعمال المؤشر المخزن في رأس هذا الأخير.

كما هي العادة، علينا الإيقاع أيضا بمخصص السلاب حيث أننا نحتاج قبلا لمخصص من أجل صنع عناصر ObjCache. هذه المهمة موكولة إلى الطريقة init و التي تقوم بصناعة "مخزون لتخزين ObjCache"

```
int ObjCache::init(KHeapStorage *kheap) {
      heap = kheap;

                                        أولا نخصص منطقة باستعمال KHeapStorage

      VRegion *rg = heap->alloc(VRBUF_SIZE, VR_MAP);
      if(!rg)
            return NOMEM;

                              ثم نصنع سلابا في هذه المنطقة لتخزين عناصر بحجم ObjCache

      SelfBuffer *sb = new(rg->start())
                  SelfBuffer(sizeof(ObjCache)+sizeof(objHead),
                  VRBUF_SIZE);
      sb->info = (DWORD) rg;

    نستعمل هذا السلاب لصناعة أول المخزون ccache باستعمال ccache .Placement new
                                        ستمكننا من صناعة مخزون لعناصر ObjCache
```

```
	void *tmp = sb->alloc();
	KASSERT(tmp);
	ccache = new(tmp) ObjCache("ccache", sizeof(ObjCache), VRBUF_SIZE);

	ccache->sbPart.push(sb);
	ccache->_nbFree += sb->freeObjs();
	return SUCCESS;
}
```

بعد ذلك نعرف المعاملين new و delete في الفئة ObjCache لاستعمال المخزون ccache

```
معامل Placement new المستعمل لصناعة ccache

void *operator new(size_t size, void *loc){
	return loc;
};

المعاملين التقليديين لصناعة عناصر ObjCache تقوم باستعمال ccache لتخصيص العناصر
و تحريرها

void *operator new(size_t size){
	return ccache->alloc(0);
};

void operator delete(void *address){
	ccache->free(address);
};
```

كما رأينا في مثال سابق، الطريقة alloc تقوم بتخصيص عنصر من أحد السلابات الجزئية أو الفارغة

```
void *ObjCache::alloc(DWORD attribs) {
	SelfBuffer *buf;
	void *ret;

نقوم باختيار أحد السلابات الجزئية أولا في حال كان موجودا
	if(!sbPart.isEmpty()) {
		buf = sbPart.Head();
		ret = buf->alloc();

إذا أصبح السلاب مليئا نضعه في القائمة المليئة

		if(buf->isFull()) {
			sbPart.remove(buf);
			sbFull.push(buf);
		}
	} else if(!sbEmpty.isEmpty() || grow()) {

إذا لم يكن هناك سلاب جزئي متوفر نختار أحد السلابات الفارغة. إذا كانت جميع
السلابات مليئة ننادي على الدالة grow انظر تحت

		buf = sbEmpty.pop();
		ret = buf->alloc();

ثم وضعه في القائمة الصحيحة تبعا لحالته

		if(!buf->isFull())
			sbPart.push(buf);
		else
			sbFull.push(buf);
```

```
        } else
                return null;
        _nbFree--;
        KASSERT(ret);

                                        تخزين مؤشر على السلاب في رأس العنصر المخزن

        ((objHead *)ret)->owner = buf;

                                        و إعادة العنوان الصحيح بعد الرأس

        return (void *) ((DWORD)ret+sizeof(objHead));
}

                        الدالة grow تقوم بصنع سلاب جديد في حال أصبحت جميع السلابات مليئة

inline SelfBuffer* grow() {
        VRegion *rg = heap->alloc(_bufSize, VR_MAP);
        if(!rg)
                return null;
        SelfBuffer *sb = new(rg->start()) SelfBuffer(_totObjSize, _bufSize);
        sb->info = (DWORD) rg;
        sbEmpty.push(sb);
        _nbFree += sb->freeObjs();
        return sb;
};
```

عكسا الطريقة free تقوم بتحرير عنصر مستعمل

```
int ObjCache::free(void *object) {

    أولا نستخرج السلاب الذي استعمل لتخزين هذا العنصر. نقوم بالنظر في رأس العنصر

        objHead *head = (objHead *)((DWORD)object-sizeof(objHead));
        SelfBuffer *buf = head->owner;
        if(!buf)
                return INVPARAM;

      علينا تحديث حالة السلاب و نقله من القائمة الأصلية إلى القائمة المناسبة في حال
                                                                تغيرت حالته
        DLList<SelfBuffer> *list;
        if(buf->isFull())
                list = &sbFull;
        else
                list = &sbPart;

        buf->free(object);
        _nbFree ++;
        if(buf->isEmpty()) {
                list->remove(buf);
                sbEmpty.push(buf);
        } else if(list != (&sbPart)) {
                list->remove(buf);
                sbPart.push(buf);
        }
}
```

### 3-2 المخصص العام Malloc

أصبح في استطاعتنا الآن صنع مخزونات للعناصر التي نود تخصيصها. لكن بقيت أمامنا مهمة أخرى: نحن لا نحتاج لصنع مخزونات كلما أردنا تخصيص عنصر ما، هذه الطريقة مفيدة فقط عندما نعرف مسبقا أننا سنحتاج لعدد كبير من هذه العناصر وليس في حالة التخصيصات العامة (مثلا تخصيص جدول من عشرة int). من أجل ذلك سننجز مخصصا عاما، هذا الأخير يقوم فقط بصنع مخزونات للأحجام الأكثر استعمالا، في حالتنا اخترت الأحجام: 8-16-32-64-128-256-512-1024-2048-4096-8192-12288.

عندما يتم طلب تخصيص لحجم ما فإن Malloc تقوم باختيار مخزون من الحجم الأقرب، مثلا إذا طلب تخصيص 27 بايت فإنها تقوم بالتخصيص باستعمال مخزون الـ32 بايت.

في حال تم طلب أحجام أكبر فإن Malloc تقوم بالتخصيص من مخصص المناطق مباشرة.

لأجل جعل خدمات هذا المخصص متوفرة لجميع العناصر نقوم بتعريف المعاملين العامين new و delete ليقوما بنداء المخصص Malloc مباشرة. هكذا يصبح بالإمكان التخصيص مباشرة باستعمال new. مثلا لتخصيص جدول من 10 int يمكننا كتابة شفرة عادية

```
int table[] = new int[10];
```

هذه التعليمة البسيطة تستعمل المخصص Malloc بطريقة شفافة عبر نداء الطريقتين alloc و free.

## 3- التجربة

تجربة هذا الدرس، كما هي العادة، بسيطة و تهدف فقط إلى تجريب المخصص بطريقتين مختلفتين.

أولا، الفئة TestObjCache تقوم بتجريب مخصص السلاب

```
مؤشر على مخزون هذه الفئة
ObjCache *cache;

class TestObjCache {
public:
	int nb;
	char name[20];

	TestObjCache(int n, char *aname) {
		nb = n;
		strncpy(name, aname, 19);
	};

	إعادة تعريف المعاملين new و delete بجعلهما يناديان على مخصص السلاب ObjCache

	void* operator new(DWORD size) {

		في حال لم يكن المخزون قد أنشئ بعد نقوم بإنشائه
		if(!cache)
			cache = new ObjCache("TestObjCache",
					sizeof(	TestObjCache),
					PAGE_SIZE);
		return cache->alloc(0);
	};

	void operator delete(void *loc) {
		cache->free(loc);
	};
};
```

ثانيا الفئة TestMalloc تقوم بتجريب المخصص العام Malloc. من أجل ذلك لا تحتاج لإعادة تعريف المعاملين new و delete. بل نستعمل المعاملين الشاملين في الملف System.h

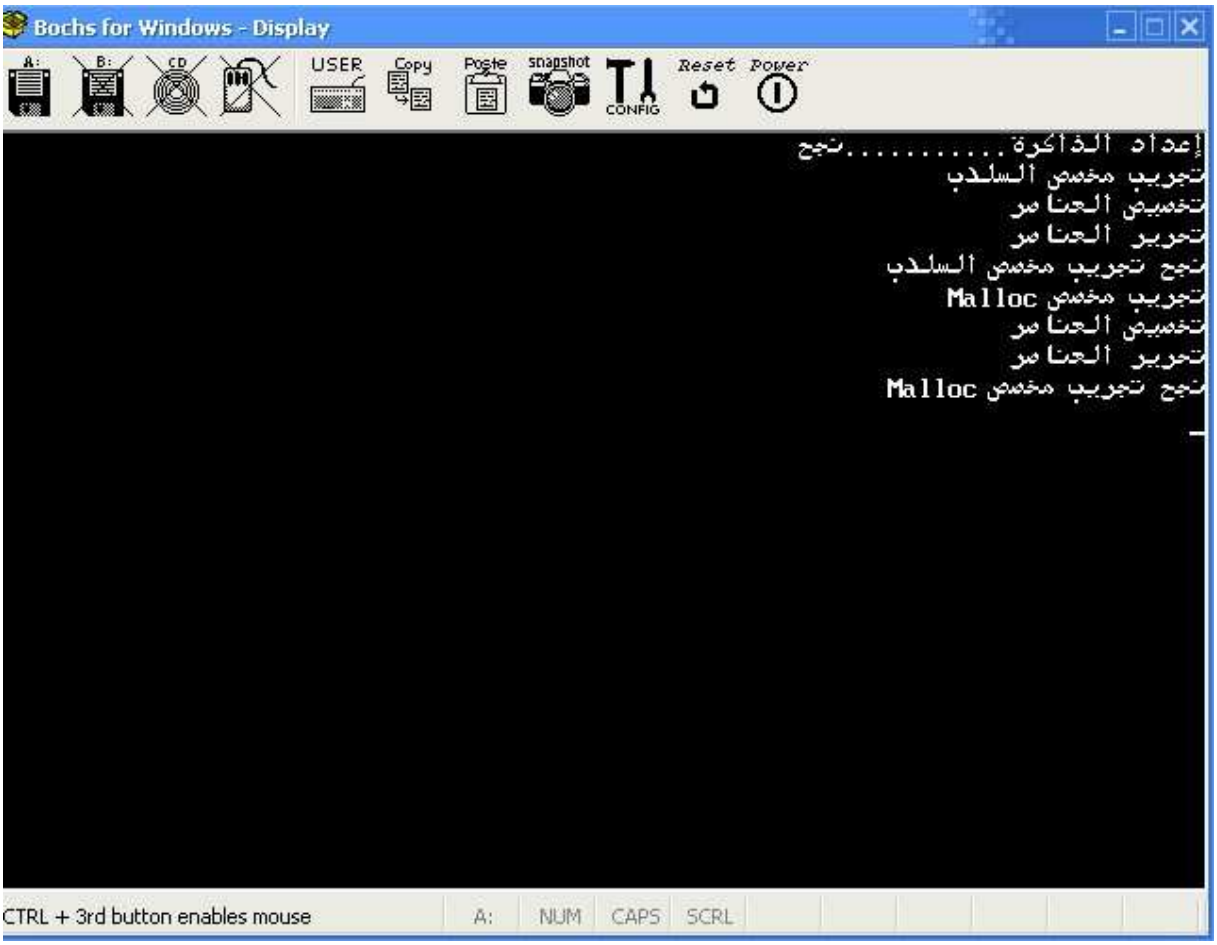
```
class TestMalloc {
public:
	int nb;

		نستعمل حجما كبيرا جدا لإرغام المخصص على استعمال مخصص المناطق مباشرة

	char name[5000];

	TestMalloc(int n, char *aname) {
		nb = n;
		strncpy(name, aname, 5000);
	};
};
```

بعد ذلك نقوم بإدخال شفرة في الدالة الرئيسية kmain تقوم بتخصيص 1000 عنصر من TestObjCache و 1000 من TestMalloc ثم تحريرها.




كما قلت سابقا. التجربة لا تقوم بشيء مميز سوى تجريب المخصص و كذا تبيان كيفية استعمال مخصص السلاب داخل فئة-مثال (TestObjCache).

النواة أصبحت الآن تتوفر على خدمات لتوفير الذاكرة في فضائها الافتراضي. ولا تستعمل مخصصا تقليديا. بل مخصص بكفاءة جيدة يمكننا من التقليل من الانقسام بنسبة مهمة إضافة إلى سرعة أدائه.

هناك عدة وظائف يمكن إضافتها إلى هذا المخصص لتحسن نسبة أدائه أكثر كما يفعل نظيراه في لينوكس و سولاريس مثلSlab coloring, static constructors & destructor. وتنجيز هذه الوظائف في حد ذاته لا يتطلب مجهودا كبيرا (بعض العمليات الحسابية و تغيير و اجهة المخصص). لكنني اخترت أن أقف عند هذا الحد لأن شرح مضمون و مزايا هذه الوظائف يتطلب درسا بأكمله.

على العموم، قطعنا الآن شوطا مهما منذ بدء الدروس. والنواة التدريبية تطورت بشكل ملحوظ بالمقارنة مع وقت كانت تكتفي فيه بطباعة "السلام عليكم" على الشاشة.

***ياسين الوافي طنجة-المغرب 06 أبريل 2006- 21h10mn***