به نام خدا

نام : عماد

نام خانوادگی : آقاجانی

شماره دانشجویی : ۸۸۵۲۱۳۴۴

استاد درس: دکتر شریفی

Linux Memory Management

فهرست مندرجات :

توضیحات

با عرض سلام و خسته نباشید

تحقیق پیش رو چکیده ای از مطالعه و یادگیری بنده از منابع مختلف است .

علت کار اضافی انجام شده روی این تکلیف صرفا یادگیری و علاقه شخصی و ایجاد یک منبع مطالعاتی چکیده دررابطه با نحوه مدیریت حافظه در لینوکس برای (حداقل) خودم بوده که هنوز هم قصد کاملتر کردنشو دارم .

همچنین در صورت تمایل از ارایه حضوری این تکلیف در کلاس درس، استقبال میکنم .

در ضمن چون هدف نحوه مدیریت حافظه بوده ، نه تعریف **Heap Memory** و **Page File** و **External Fragmentation** ، فرض بر اینه که این اطلاعات رو داریم .

در مورد متن هم باید بگم که الکی نخواستم با کلمات جمله سازی کنم و برای همین به سبک **note** نوشته شده .

با تشکر از توجه شما

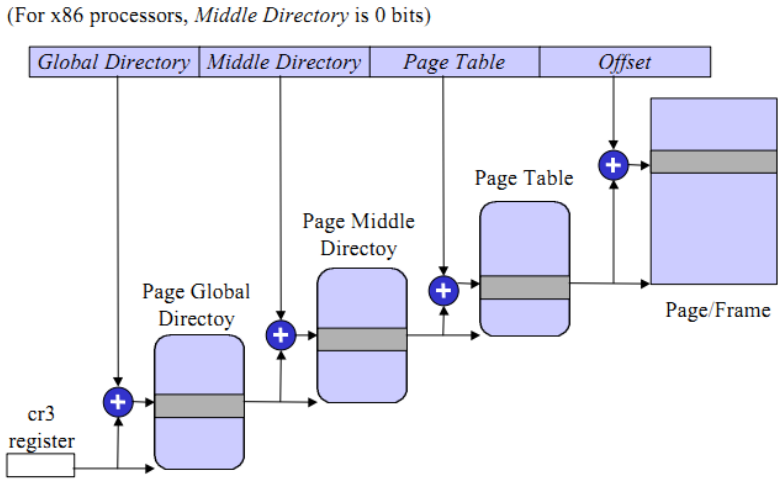
آقاجانی **=)**

مکانیزم **Paging**

- پردازنده هایی از قبیل **Intel x86** توانایی پشتیبانی از شیوه **Segment** را دارا می باشند.
- ولی در لینوکس از شیوه **Segment** استفاده نمیشود، چراکه :
  ۱) استفاده از مقدار ثبات **Segment** یکسان برای تمام پردازه ها(**Process**)، مدیریت حافظه را آسان میکند.
  ۲) حفظ قابلیلت استفاده بر روی سایر پردازنده ها (**Portability**).
- در شیوه **Paging** استفاده شده در لینوکس :
  ۱) اندازه هر **Page File** برابر 4K.
  ۲) یک **Page Table** سه لایه برای مدیریت ۶۴ بیت خط آدرس.
  ۳) در پردازنده هایی از قبیل **x86** :
    - تنها به دو لایه **Page Table** نیاز میباشد.
    - **Page Table** توسط سخت افزار پشتیبانی میشود.
    - امکان بهره گیری از **TLB(Translation lookaside buffer)** فراهم آمده است.

فضای آدرس (Addresses Space) منطقی/ فیزیکی

- شمای آدرس های منطقی در لینوکس :



- حدود ۲ مگابایت اول **حافظه فیزیکی** بمنظور معماری کامپیوتر جاری و اطلاعات و ... سیستم عامل رزرو شده است و مابقی برای انجام **Paging** در دسترس میباشد.
- **فضای منطقی** آدرس های یک پردازه به دو قسمت تقسیم شده است :

۱) که شامل `0X00000000` تا `PAGE_OFFSET-1` میشود و میتوان چه در **User mode** و چه در **Kernel mode** آن را آدرس دهی کرد

۲) شامل `PAGE_OFFSET` تا `0Xffffffff` میشود و تنها میتوان در **Kernel mode** آن را آدرس دهی کرد

- `PAGE_OFFSET` معمولا `0Xc0000000` میباشد

مدیریت **Page Frame** در لینوکس

- کرنل وضعیت کنونی هر **Page Frame** را در آرایه ای از `struct page` ها ، بنام `mem_map` ، نگه میدارد.
- همچنین کرنل تعداد استفاده از یک **Page Frame** را نگه میدارد . مقدار ۰ بمعنای آزاد بودن **Page Frame** و عدد بزرگتر از ۰ بمعنای تعداد استفاده از آن **Page Farme** میباشد.
- همچنین پرچم(**Flag**) هایی از قبیل **Dirty**، **Locked**، **Referenced** و غیره نیز برای **Page Frame** ها وجود دارد.
- کرنل **Page Frame** ها را بوسیله دستورات زیر گرفته و یا آزاد میکند :

```
__get_free_pages ( gfp _mask, order )
free_pages ( addr, order )
```

- در تئوری، شیوه **Paging** نیاز به تخصیص حافظه پیوسته را از بین برده است - بعضی از عملیات مانند **DMA** در روند **Paging** خلل ایجاد میکنند .
- از طرف دیگر باید توجه داشت که شیوه تخصیص حافظه پیوسته **Page Table** کرنل را تغییر نمیدهد که در نتیجه باعث حفظ **TLB** میگردد و زمان دسترسی را کاهش میدهد !
- در نتیجه لینوکس از شیوه ای بمنظور تخصیص پیوسته **Page Frame** (**allocating contiguous page frames**) استفاده میکند .

در این روش مشکل **External Fragmentation** نیز وجود نخواهد داشت.

انواع مدیریت حافظه های استفاده شده در لینوکس :

۱) تخصیص حافظه پیوسته (**contiguous memory allocation**)

۲) تخصیص حافظه ناپیوسته (**noncontiguous memory allocation**)

تخصیص حافظه پیوسته (**contiguous memory allocation**) [۴و۵و۶]

- در این شیوه مدیریت حافظه که یک شیوه سنتی محسوب میشود، تخصیص حافظه به پردازه ها از بلاک های متوالی حافظه صورت میگیرد.
- اشکال این شیوه : مشکل **External Fragmentation** .
- ویژگی این روش : در حافظه هایی با حجم بالا مشکل های مرتبط با کش(**Cache**) و تاخیر دسترسی (**Access Latency**) وجود ندارد.

الگوریتم **Buddy** [۷و۸]

- تمامی **page frame** های موجود به ۱۰ لیست از بلاک ها تقسیم‌بندی میشوند که شامل لیست هایی با ۱، ۲، ۴، ۸، ۱۶، ۳۲، ۶۴، ۱۲۸، ۲۵۶، ۵۱۲ **page frame** پیوسته میشود.
  - آدرس اولین **page frame** در یک بلاک ضریبی از سایز آن دسته میباشد. بعنوان مثال در یک بلاک ۱۶تایی، آدرس اولین **page frame** برابر $16x2^{12}$ خواهد بود.
- الگوریتم برای تخصیص مثلا یک بلاک ۱۲۸ تایی از **page frame** های پیوسته :
  - ابتدا به دنبال یک بلاک خالی در لیست ۱۲۸ تایی بگرد
  - اگر بلاک خالی پیدا نشد، در لیست ۲۵۶ تایی بدنبال یک بلاک خالی بگرد
  - اگر بلاکی پیدا کرد، کرنل ۱۲۸ تا از ۲۵۶ **page frame** موجود را تخصیص میدهد و ۱۲۸ تای باقیمانده را در لیست ۱۲۸تایی ها قرار میدهد
  - اگر در لیست های بزرگتر پیدا شد، انقدر بلاک را خرد کن (بصورت باینری) تا حافظه مورد نیاز را اختصاص بدهی و بلاک های بوجود آمده را در لیست مناسب قرار بده
  - اگر هیچ بلاکی قابل تخصیص نیست یک پیغام خطا بازگردان

- یک مثال از نحوه تخصیص حافظه (**Buddy System**) :

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							
7.1	A: 2^0	C: 2^0	2^1		2^1		2^1		2^3							
7.2	A: 2^0	C: 2^0	2^1		2^2				2^3							
8	2^0	C: 2^0	2^1		2^2				2^3							
9.1	2^0	2^0	2^1		2^2				2^3							
9.2	2^1		2^1		2^2				2^3							
9.3	2^2				2^2				2^3							
9.4	2^3								2^3							
9.5	2^4															

This allocation could have occurred in the following manner

1. The initial situation.
2. Program A requests memory 34K, order 0.
    1. No order 0 blocks are available, so an order 4 block is split, creating two order 3 blocks.
    2. Still no order 0 blocks available, so the first order 3 block is split, creating two order 2 blocks.
    3. Still no order 0 blocks available, so the first order 2 block is split, creating two order 1 blocks.
    4. Still no order 0 blocks available, so the first order 1 block is split, creating two order 0 blocks.
    5. Now an order 0 block is available, so it is allocated to A.
3. Program B requests memory 66K, order 1. An order 1 block is available, so it is allocated to B.
4. Program C requests memory 35K, order 0. An order 0 block is available, so it is allocated to C.
5. Program D requests memory 67K, order 1.
    1. No order 1 blocks are available, so an order 2 block is split, creating two order 1 blocks.
    2. Now an order 1 block is available, so it is allocated to D.
6. Program B releases its memory, freeing one order 1 block.
7. Program D releases its memory.
    1. One order 1 block is freed.
    2. Since the buddy block of the newly freed block is also free, the two are merged into one order 2 block.
8. Program A releases its memory, freeing one order 0 block.
9. Program C releases its memory.
    1. One order 0 block is freed.
    2. Since the buddy block of the newly freed block is also free, the two are merged into one order 1 block.
    3. Since the buddy block of the newly formed order 1 block is also free, the two are merged into one order 2 block.
    4. Since the buddy block of the newly formed order 2 block is also free, the two are merged into one order 3 block.
    5. Since the buddy block of the newly formed order 3 block is also free, the two are merged into one order 4 block.

As you can see, what happens when a memory request is made is as follows:

- If memory is to be **allocated**

1. Look for a memory slot of a suitable size (the minimal 2^k block that is larger or equal to that of the requested memory)
    1. If it is found, it is allocated to the program
    2. If not, it tries to make a suitable memory slot. The system does so by trying the following:
        1. Split a free memory slot larger than the requested memory size into half
        2. If the lower limit is reached, then allocate that amount of memory
        3. Go back to step 1 (look for a memory slot of a suitable size)
        4. Repeat this process until a suitable memory slot is found

- If memory is to be **freed**

1. Free the block of memory
2. Look at the neighboring block - is it free too?
3. If it is, combine the two, and go back to step 2 and repeat this process until either the upper limit is reached (all memory is freed), or until a non-free neighbour block is encountered

- وقتی یک بلاک آزاد میگردد، کرنل سعی بر ادغام جفت های مشابه(**free Buddy Block**) با سایز **b** به یک **2b** دارد.
- دو بلاک **buddy** تلقی میشوند اگر :
    ۱) اگر هم سایز باشند
    ۲) هر دو در آدرس فیزیکی متوالی واقع شده باشند
    ۳) آدرس فیزیکی **page** اول از بلاک اول ضریبی از $2bx2^{12}$ باشد
- عملیات ادغام متوالیا تکرار میشود.
- لینوکس از دو نوع **Buddy System** متفاوت برای مدیریت حافظه استفاده میکند.
    - یکی برای **page frame** های مناسب برای **DMA** (بعنوان مثال **page frame** هایی با آدرس هایی کمتر از **16MB**).
    - دیگری برای سایر.
- هر **Buddy System** مبتنی است بر ..
    - آرایه توصیفگر **frame** ها ، **mem_map**
    - آرایه ۱۰ تایی از **free_area_struct**، هر درایه برای یک سایز(۱، ۲، ... ، ۵۱۲)(هر درایه یک لینک لیست دوطرفه از بلاک های **page frame** با سایز مربوطه خود میباشد) .
    - ده بیت‌مپ (**Bitmap**) برای نگهداری وضعیت تخصیص بلاک های موجود.
- الگوریتم **Buddy** برای مدیریت حجم های بالای حافظه بخوبی عمل میکند. اما در رابطه با بخش های کوچک حافظه با مشکل **Internal Fragmentation** مواجه است !
- برای حل این مشکل از نسخه ۲.۲ لینوکس **slab allocator** معرفی گردید

مکانیزم Slab Allocator [۹و۱۰]

- ایده اصلی : نگهداری اشیا(**objects**) بتازگی استفاده شده در کش
- در این شیوه :
    - قسمت های مختلف حافظه بعنوان یک اشیا همراه با اطلاعات و متدها (مانند سازنده و مخرب و ...) دیده میشوند.
    - اشیا بصورت دسته بندی شده به کش فرستاده میشوند
    - یک مجموعه ویژه در کش برای عملیات های سیستم عامل در نظر گرفته میشود

- **Slab cache** شامل هیچ یا چند **slab** میشود. **slab** یک یا چند **page frame** پیوسته از **Buddy System** میباشد.
- اشیا بوسیله دستور `kmem_cache_alloc(cachep)` تخصیص می یابند (`cachep` یک اشاره‌گر به کش میباشد).
- اشیا بکمک دستور `kmem_cache_free(cachep, objp)` آزاد میشوند.
- یک گروه از کش های عمومی وجود دارد که با سایز های گسترده ای بین ۳۲ تا ۱۳۱۰۷۲ بایت توزیع شده است.
- برای گرفتن شی ای از این کش ها از دستور `kmalloc(size, flags)` استفاده میشود.
- برای آزاد کردن شی ای از این کش ها عمومی از دستور `kfree(objp)` استفاده میشود.

تخصیص حافظه ناپیوسته (noncontiguous memory allocation) [۶]

- در این شیوه از یک حافظه مجازی(**Virtual Memory**) برای پیاده سازی شیوه سنتی " تخصیص حافظه ناپیوسته " استفاده میگردد.
- یک فضا در فضای آدرس مجازی بین `VMALLOC_START` و `VMALLOC_END` رزرو شده است. مکان `VMALLOC_START` به میزان حافظه فیزیکی بستگی دارد ولی محدوده همیشه حداقل `VMALLOC_RESERVE` میباشد که بر روی یک **X86** برابر ۱۲۸ مگابایت میباشد.
- لینوکس از تخصیص حافظه ناپیوسته عموما پرهیز میکند. اما گاها برای اختصاص حافظه به گونه ای از درخواست های محدود استفاده از این روش کارامدتر میشود.
- لینوکس بیشتر آدرس های بالاتر از `PAGE_OFFSET` را بمنظور تخصیص ناپیوسته استفاده میکند.
- برای تخصیص و آزاد کردن حافظه به روش ناپیوسته از دستورات `vmalloc(size)` و `vfree(addr)` استفاده میگردد.

مرور : انواع تخصیص حافظه در کرنل لینوکس

- کرنل حافظه پویا خود را به یکی از ۳ روش زیر میگیرد :

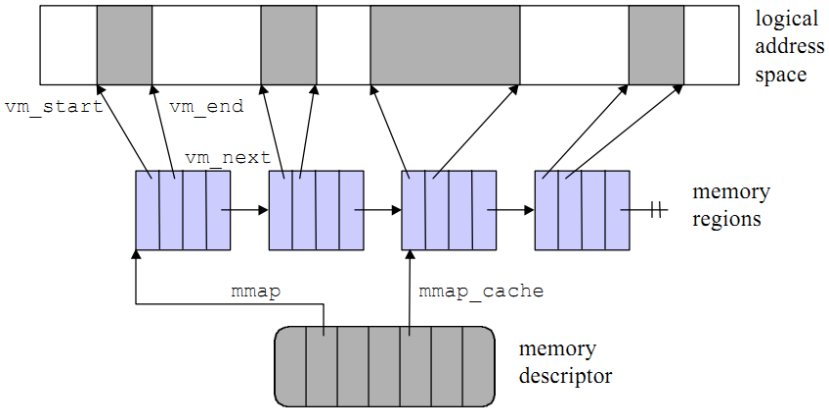
۱) استفاده از `__get_free_pages()` برای گرفتن **pages** از **Buddy System**

۲) `kmem_cache_alloc()` یا `kmalloc()` برای استفاده از تخصیص‌گر **slab** برای گرفتن اشیا خاص یا عمومی

۳) `vmalloc()` برای گرفتن حافظه ناپیوسته

فضای آدرس دهی پردازه ها

- فضای آدرس(**Address Space**) یک پردازه، شامل تمام آدرس های منطقی قابل دسترس توسط پردازه میشود.
- فضای آدرس هر پردازه مستقل از دیگری است (مگر اینکه مشترک باشد) .
- کرنل عملیات تخصیص آدرس منطقی برای پردازه ها را در یک بازه بنام "مناطق حافظه"(**Memory Reigns**) انجام میدهد. این ناحیه یک مقدار اولیه و یک طول دارد که مضربی از ۴۰۹۶ می باشد.
- برای کرنال، دو دسته درخواست حافظه اتفاق می افتد :
  ۱) درخواست های غیر ضروری :
     a. نیازی به تمام **Page** ها همیشه نیست
     b. تخصیص حافظه ممکن است برای لحظه ای امکان پذیر نباشد
  ۲) درخواست های نامطمئن
     a. کرنل باید آمادگی روبرو شدن با خطا را داشته باشد
- در نتیجه کرنل لینوکس با توجه به پردازه، عملیات تخصیص حافظه را گوناگون انجام میدهد.
- شرایطی که پروسه ها حافظه جدید میگیرند :
  ۱) ایجاد یک پروسه جدید (`fork()`)
  ۲) بارگزاری یک برنامه جدید (`execve()`)
  ۳) نگاشت حافظه به فایل (`mmap()`)
  ۴) رشد پشته(**Stack**)
  ۵) ایجاد حافظه اشتراکی (`shmat()`)
  ۶) گسترش **heap** (`malloc()`)
- تمامی اطلاعات مرتبط با فضای آدرس یک پردازه در بخش توصیفگر حافظه (**memory descriptor**) (`mm_struct`) قرار گرفته است که از طریق فیلد `mm` از توصیفگر پردازه(**process descriptor**) قابل دسترس میباشد. بعضی از اطلاعاتی که قابل دسترس میباشد :
  ۱) یک اشاره گر به بالاترین مرحله **Page Table** یا همان **Page Global Directory** از طریق `pgd`
  ۲) تعداد **page frame**های تخصیص یافته از طریق `rss`
  ۳) . . .
- توصیفگر حافظه از کش اختصاص‌گر **slab** (**slab allocator cache**) و با دستور **mm_alloc()** تخصیص می یابد.
- شمایی انتزاعی از توصیفگر حافظه، مناطق حافظه و فضای آدرس منطقی



- برای تخصیص و آزاد سازی یک آدرس مجازی، کرنل بترتیب از دستورات `Do_mmap()` و `do_munmap()` استفاده میکند.

رسیدگی کننده **Page Fault**

- زمانی که یک پردازه از کرنل تقاضای حافظه بیشتر میکند، پردازه تنها فضای آدرس منطقی بیشتر میگیرد و نه حافظه ی فیزیکی بیشتر .
- زمانی که پردازه قصد استفاده و دسترسی به فضای آدرس منطقی جدید خود را میکند، یک **Page Fault** رخ میدهد تا به کرنل بگوید که واقعا به این حافظه نیاز دارد.
- رسیدگی کننده **Page Fault** ، آدرس منطقی تقاضا شده را با ناحیه ای از حافظه که در اختیار پردازه هست مقایسه میکند تا یا متوجه خطا در تقاضا شود (عدم امکان دسترسی پردازه به آن آدرس مجازی) و یا متوجه نیاز پردازه به اختصاص از حافظه فیزیکی شود. همچنین آدرس تقاضا شده، ممکن است بدلایلی از روی حافظه فیزیکی به روی دیسک سخت منتقل شده باشد.

نحوه اختصاص حافظه به پردازه جدید

- زمانی که کرنل یک پردازه جدید ایجاد میکند، به آن یک فضای آدرس کاملا جدید نمیدهد و در ابتدا پردازه جدید از فضای آدرس پردازه پدر خود بصورت اشتراکی استفاده میکند. کرنل در این زمان از **page farme** های مشترک محافظت میکند.
  - سپس اولین مرتبه ای که پردازه پدر یا پردازه جدید قصد نوشتن بر روی **page frame** های مشترک را داشته باشند یک استثنا (**exception**) رخ میدهد.
  - کرنل این استثنا را دریافت کرده و یک نسخه کپی از آن **farme** ، برای پردازه ای که قصد نوشتن داشت، ایجاد میشود.

مدیریت حافظه Heap

- هر پردازه میتوانند بر روی **heap** خود، حافظه پویا جدید بدست بیاورد .
- دستورات زبان C بمنظور انجام تغییرات در **Heap** : `malloc()`, `calloc()` و `free()`.
- تمامی این دستورات از دستور اصلی `brk()` مشتق شده اند.
  - این تنها دستوری میباشد که یک فراخوانی سیستمی (**System Call**) را انجام میدهد.
  - این دستور مستقیبا میزان حافظه **Heap** را تغییر میدهد.
  - این دستور در حقیقت عمل تخصیص و رها سازی فضای آدرس منطقی را برعهده دارد.
- زمانی که پردازه یک **page frame** میگیرد، عمل تخصیص قطعه حافظه های کوچک (مانند: **malloc (sizeof(char)*50**) در **user space** صورت میگیرد.

منابع تحقیقاتی

منبع اصلی:

www.inf.fu-berlin.de/lehre/SS01/OS/Lectures/Lecture14.pdf

سایر منابع:

[1] http://tldp.org/LDP/tlk/mm/memory.html

[2] http://en.wikipedia.org/wiki/Translation_lookaside_buffer

[3] http://www.cs.umd.edu/~meesh/cmsc411/website/saltz/cs412/lect3.html

[4] http://www.techopedia.com/definition/3769/contiguous-memory-allocation

[5] http://siber.cankaya.edu.tr/ozdogan/OperatingSystems/ceng328/node180.html

[6] http://www.kernel.org/doc/gorman/html/understand/understand010.html

[7]http://en.wikipedia.org/wiki/Buddy_memory_allocation

[8]www.cs.purdue.edu/homes/hosking/690M/p421-peterson.pdf

[9]en.wikipedia.org/wiki/Slab_allocation

[10] http://www.kernel.org/doc/gorman/html/understand/understand011.html