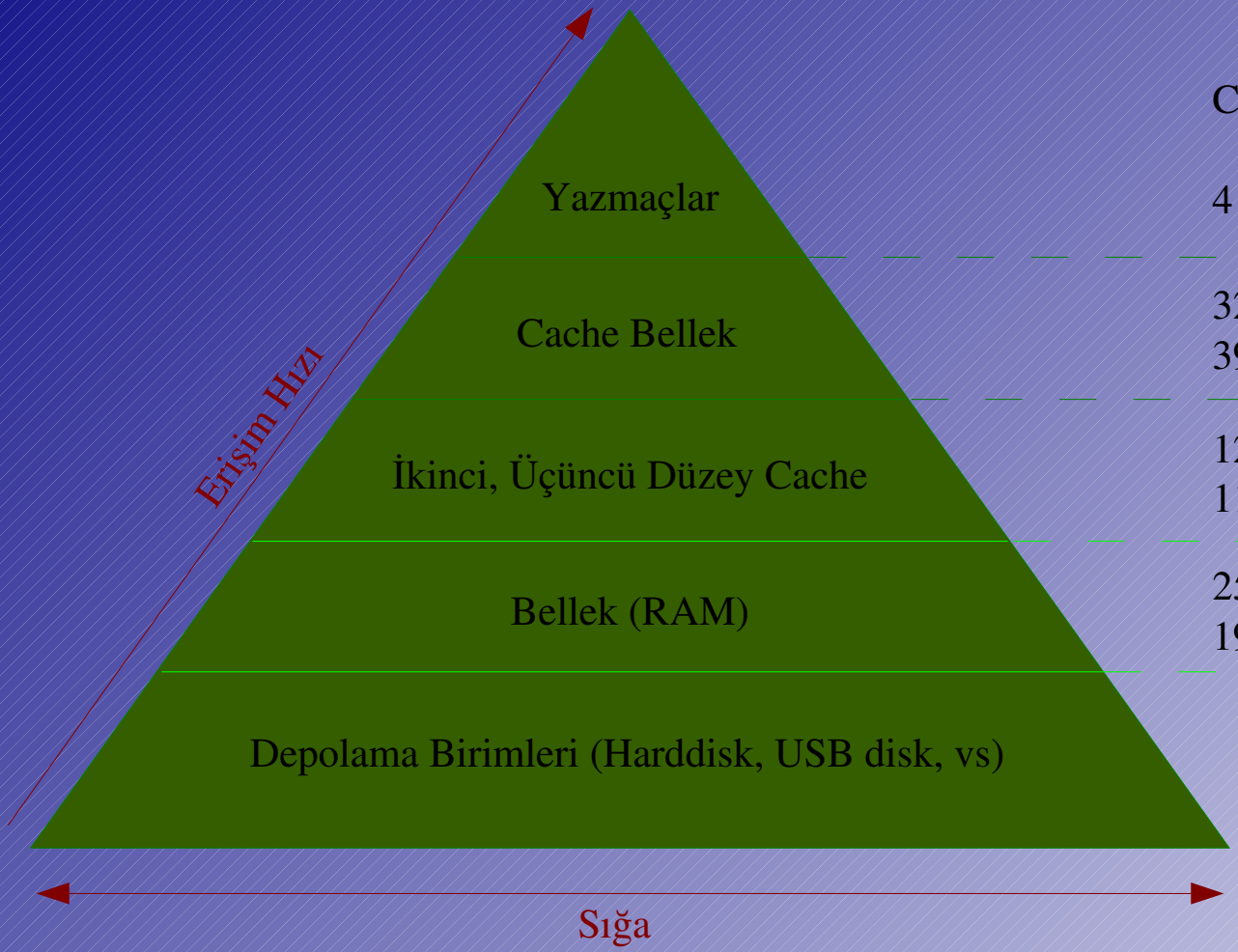
Linux'ta Bellek Yönetimi

Gürer Özen
madcat@e-kolay.net

Bellek Çeşitleri



	Celeron 766	PPC 604e 200
Yazmaçlar	4 adet 32bit	32 adet 32bit
Cache Bellek	32KB 3970 MB/s	64KB 730 MB/s
İkinci, Üçüncü Düzey Cache	128KB 1100 MB/s	512KB 120 MB/s
Bellek (RAM)	256MB SDRAM 198 MB/s	32MB EDO 87 MB/s
Depolama Birimleri (Harddisk, USB disk, vs)	40GB IDE 22 MB/s	

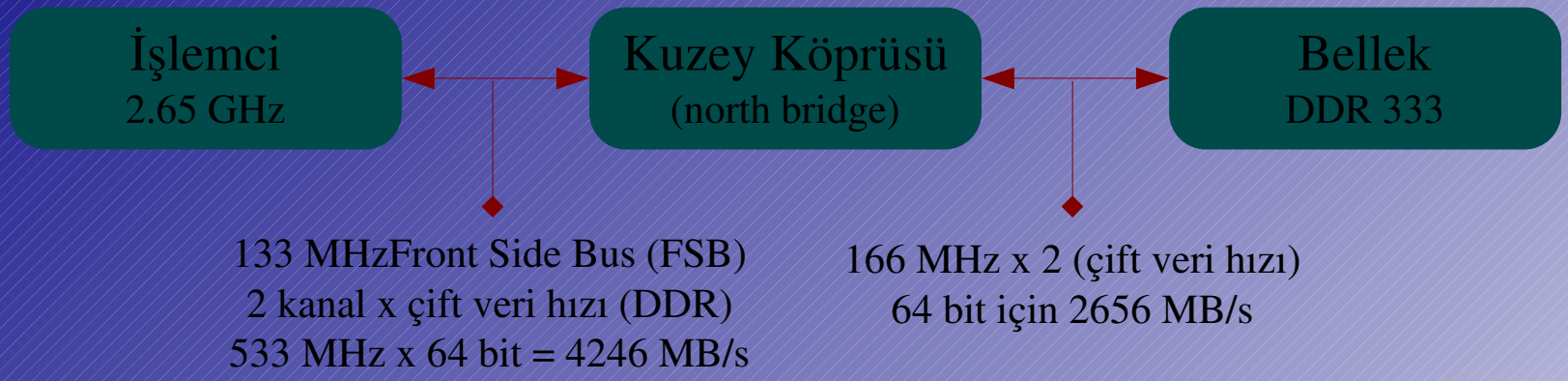
Yazmaçlar

- İşlemcinin çalışma hızında erişilebilirler
- Verimli kullanımları derleyicilerin ve assembler programcılarının elindedir

Cache

- Çok kullanılan verilere hızlı erişim sağlar
- Başlıca iki tiptir:
  - Harvard mimarisi: verim için komut ve veriler ayrı tutulur, toplam cache alanı yarıya iner
  - Birleşik mimari: tek bir cache alanı vardır
- Bellekten okunabilecek minimum veri miktarı cache satırıdır (cache line). Verileri cache satırı boyunda hizalamak (alignment) ve satır boyutunda okumak cache'den maksimum verim sağlar.
- En basit cache yapısında cache'deki her satır bellekteki belli bir grup adrese ait verileri taşıyabilir (direct mapping). Bir satırın değişik adres gruplarını taşıyabilmesi işlemci yapısını karmaşıklaştırırken performansı arttırır (2, 4, 8 way associative).
- Belleğe geri yazım
  - İşlemci veriyi cache ve belleğe yazar (write-through)
  - İşlemci veriyi cache'ye yazar, daha sonra cache veriyi belleğe taşır (write-back). Bu durumda bellek ile cache arasındaki farklılığın problem yaratmaması için sistemin yapabilecekleri:
    - Gerektiğinde cache'nin belleğe yazılmasını zorlamak
    - Belli veriler üzerince cache'lemeyi ya da write-back özelliğini kapatmak
    - Verilerin uygulamalar ve işlemciler arasında paylaşımını önlemek / senkronize etmek
- Donanıma ait olup işletim sistemi ve uygulamalara şeffaf olmakla birlikte, maksimum verim için özellikle cache satırı boyu ve belleğe geri yazımın dikkate alınması gerekir.

Bellek

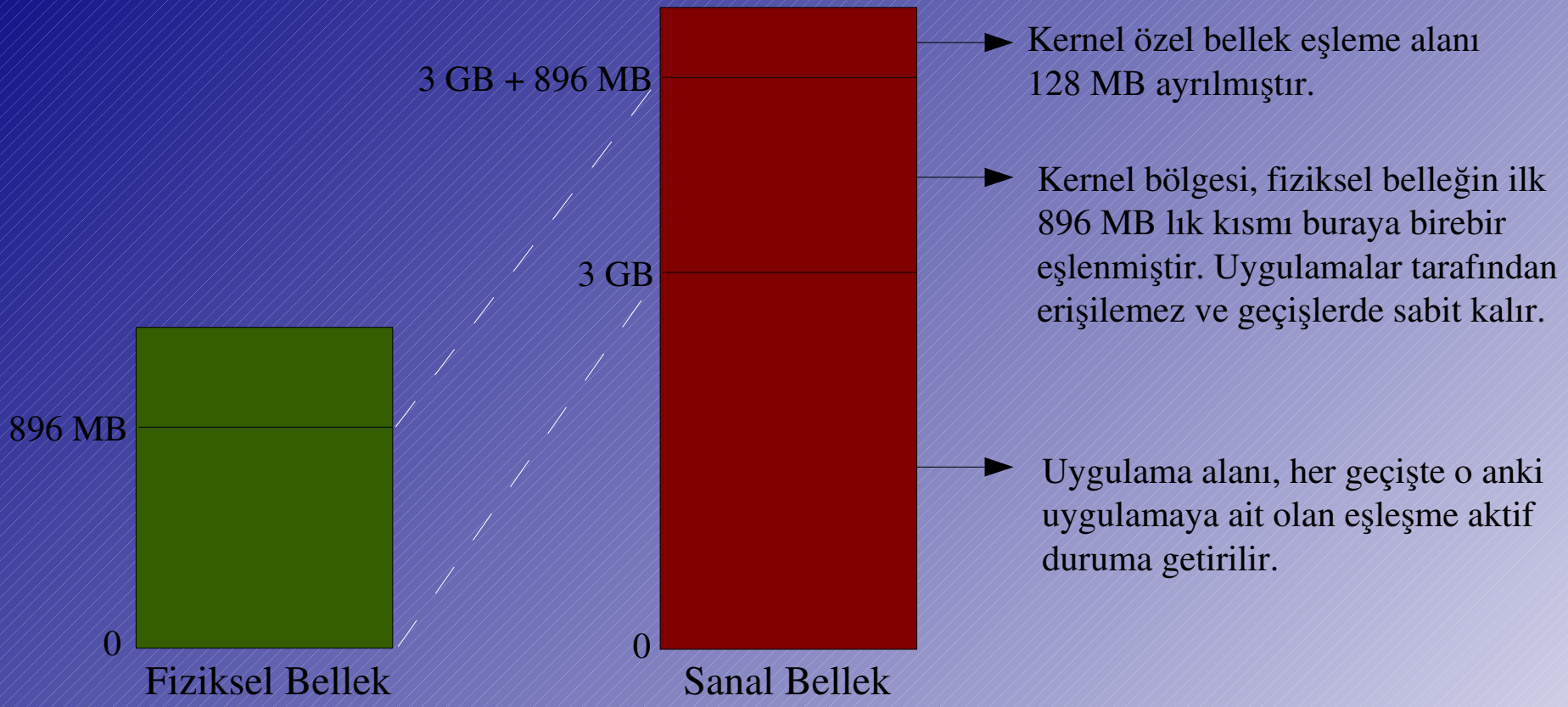


- İşletim sistemi tarafından yönetilir.
- Sayfa (page) adı verilen, mimari ve bellek boyuna göre 4-64 KB boyunda birimler bazında kullanılır
- MMU (memory management unit) adlı işlemci birimi yardımıyla oluşturulan sanal adresleme ile:
  - Varolandan daha büyük bir bellek alanı kullanılabilir
  - Belleğe sığandan daha fazla uygulama çalıştırılabilir
  - Her uygulamanın kendine ait bir sanal adres alanı olur
- Uygulamanın erişmek istediği sanal adres gerçek bellekte yoksa
  - İşlemci bir sayfa hatası (page fault) verir
  - İşletim sistemi kontrolü alır, gerekirse bellekte boş yer açıp, istenen sayfayı takas alanından yükler
  - Uygulama kaldığı yerden devam eder
- Gerçek bellek dolmaya başladığında işletim sistemi az erişilen sayfaları takas alanına taşır
- Sayfa hataları büyük yavaşlamaya neden olduğundan, en aza indirgenmeye çalışılır

Depolama

- Kalıcı depolama sağlar
- Uygulamaları, verileri, meta veriyi (hangi dosya nerde, ne özellikleri var) ve takas alanını depolar
- Sabit diskler eksenleri etrafında dönen disk şeklinde plakalar ile, bu plakalar üzerinde sabit bir hat boyunca merkez ile kenar arasında gidip gelebilen okuyucu/yazıcı kafalardan oluşur
- Dairesel hareket dolayısıyla dönme hızı plakların kenarında yüksek, ortasında yavaştır.
- Okuma hızı yaklaşık 5-40 MB/s (belleğin yüzde biri) ile aslında hızlı olmakla birlikte, kafanın belirli bir pozisyona gelmesi 5-10 milisaniye (belleğin yüzbinde biri) harcadığından maksimum verim için bazı optimizasyonlar yapılmalıdır:
  - Veri okunurken istenen miktardan fazla okuma (readahead)
    - Uygulama tekrar veri istediğinde, veri direk bellekten verilir
    - Okunan fazla veri bellekte yer tutar
    - Uygulamanın veri erişim biçimi tahmin edilemezse performans kaybına neden olur
  - Kafa hareketlerini azaltma
    - Dosya sistemi en az hareketle dosyalara erişilebilinecek biçimde tasarlanır
    - Dosyaların meta verileri bellekte tutulur
    - Disk üzerine yapılacak okuma/yazma işlemleri biriktirilir, uygulanacakları konuma göre sıralanır ve toplu halde uygulanırlar.
- Sabit diskin plak ve kafa sayısının fazla olması erişim süresini kısaltmakla birlikte pahalıdır. Bunun yerine düşük sığalı birden fazla sabit diskten oluşan RAID (redundant array of inexpensive disks) sistemleri kullanılabilir.

32 bit Mimaride Linux Adresleme Alanı



Kernel bölgesi adresi bellek başlangıcından PAGE_OFFSET (include/asm-i386/page.h) değeri kadar ilerdedir. PAGE_OFFSET 32 bit mimarilerde genellikle `0xC0000000` (3 GB) olarak ayarlanmıştır.

Uygulama 1 işletiliyor	Uygulama	`0x08048000`	-> Fiziksel `0x010c0000`
	Kernel	`0xc00e00000`	-> Fiziksel `0x000e0000`
Uygulama 2 işletiliyor	Uygulama	`0x08048000`	-> Fiziksel `0x011fe000`
	Kernel	`0xc00e00000`	-> Fiziksel `0x000e0000`

Bellek Sayfaları

Fiziksel bellekteki her sayfa için, o sayfanın durumunu içeren bir struct page yapısı tutulur.

```
struct page {
        page_flags_t flags;             Sayfanın durumu (sıcak, kirli, ayrılmış, kilitli, vs)
        atomic_t count;                 Kullanıcı sayısı, sıfır ise boş
        struct list_head list;          Sayfayı çeşitli kernel listelerinde tutmak için
        . . .
}
```

Tüm sayfa bilgi yapıları bir mem_map dizisinde tutulur. 32 bit mimaride sayfa bilgisi 44 byte boyundadır, 256 MB bellek (65536 adet 4KB sayfa) için bu dizi yaklaşık 2.75 MB (%1) yer kaplar.

```
struct page *mem_map;
```

include/asm-i386/page.h

```
#define PAGE_SHIFT          12
#define __PAGE_OFFSET       (0xC0000000UL)
#define __pa(x)             ((unsigned long) (x) – PAGE_OFFSET)
#define virt_to_page(kaddr) (mem_map + (__pa(kaddr) >> PAGE_SHIFT))
```

Verilen bir sanal adresin ait olduğu sayfanın bilgi yapısına virt_to_page() fonksiyonu ile ulaşılabilir.

Bellek Bölgeleri

Bellekteki sayfalar kullanım amacı ve şeklinde göre üç ana bölgeye ayrılmıştır:

- ZONE_DMA: Donanımın DMA için özel ihtiyaç duyduğu alan, intel mimarisinde ISA donanımlarının erişebileceği belleğin ilk 16 MB kısmı
- ZONE_NORMAL: Çekirdeğin kendi adres alanına direk eşleştirebildiği bellek, 32 bit mimaride 896 MB
- ZONE_HIGHMEM: Direk eşleştirilen belleğin dışında kalan fiziksel bellek

include/linux/mmzone.h:

```
struct zone {
        spinlock_t  lock;
        unsigned long  free_pages;          Toplam boş sayfa sayısı
        unsigned long  pages_min;           Daha az boş sayfa kaldığında sayfa atma başlar
        unsigned long  pages_low;           Sayfa atma arka planda başlatılır
        unsigned long  pages_high;          Bu kadar sayfa açılana kadar devam eder
        . . .
        struct free_area free_area[MAX_ORDER];   Badi sayfa ayırıcının boş sayfa listesi
        . . .
        struct list_head  active_list;
        struct list_head  inactive_list;
        . . .
        struct page *zone_mem_map;          Bölge sayfalarının mem_map içindeki yeri
        . . .
        unsigned long present_pages;        Toplam sayfa sayısı
}
```

Bellek Düğümleri

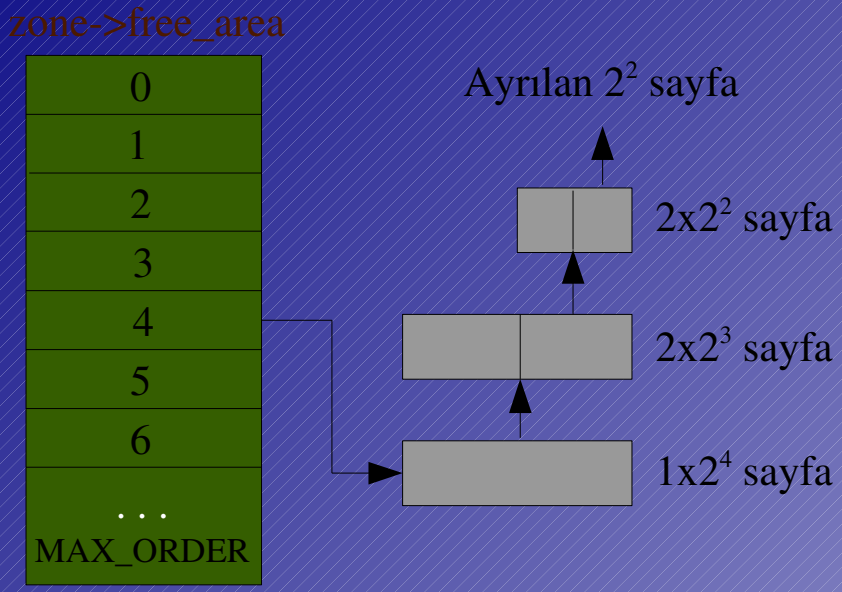
NUMA: Tekdüze olmayan bellek erişimi (Non-Uniform Memory Access)

- Bellek işlemciden uzaklığına göre farklı hızlarda erişilen ayrık alanlardan oluşmuştur
  - Çok işlemcili sistemlerde her işlemcinin kendine ait bellek alanı
  - Donanımın DMA ile daha hızlı kullanabileceği yakın alanlar
- Bu alanlara "düğüm" (node) adı verilir ve bilgileri pg_data_t yapılarında tutulur
- Sistemdeki tüm düğümlere pgdat_list listesiyle ulaşılabilir
- Standart tek işlemcili sistemlerde tek bir sabit config_page_data düğümü bulunmaktadır

include/linux/mmzone.h:

```
struct zonelist {
        struct zone *zones[MAX_NUMNODES * MAX_NR_ZONES + 1];
};
typedef struct pglist_data {
        struct zone node_zones[MAX_NR_ZONES];          Düğümün bölgeleri
        struct zonelist node_zonelists[MAX_NR_ZONES];  Bölgelerin yer ayırma öncelik sırası
        int nr_zones;                                  Düğümün bölge sayısı (1-3)
        struct page *node_mem_map;                     Düğüm sayfalarının mem_map içindeki yeri
        . . .
        struct pglist_data *pgdat_next;                Sonraki düğüm
        . . .
} pg_data_t;
```

Badi Sayfa Ayırıcı



Çekirdek içi arabirim:

```
alloc_page (gfp_mask);
alloc_pages (gfp_mask, order);
void *get_free_page (gfs_mask);

__free_page (page);
__free_pages (page, order);
free_page (void *addr);
```

- Kalbi mm/page_alloc.c içinde

```
__alloc_pages (  unsigned int gfp_mask,
                 unsigned int order,
                 struct zonelist *zonelist  )
__free_pages_ok ( struct page *page,
                  unsigned int order  )
```

- Verilen bölge listesinde 2^{order} sayfalık boşluk arar
- Her bölgeye ait boş sayfa listesi free_area yapısındadır

```
struct free_area {
        struct list_head   free_list;
        unsigned long      *map;
}
```

- Bu yapı 0 ile MAX_ORDER arasındaki her değer için $2^{değer}$ boyutundaki boş alanların listesini ve hangi badilerin kullanıldığını gösteren bit haritasını tutar.
- Sayfa istendiğinde, istenen sıradaki boş sayfa verilir
- Eğer istenen sırada boş sayfa yoksa, daha üst sıralardaki sayfalar gerektiği kadar ikiye bölünerek alt sıralara taşınır.
- Geri verilen sayfanın diğer çifti (badisi) de boş ise iki sayfa birleştirilip bir üst sıraya taşınır.
- 2^n sırasındaki bir sayfanın badisinin adresi sayfanın adresinde sağdan n. bit değiştirilerek hızla bulunabilir.

kswapd

Linux çekirdeği açılışı tamamlayıp çalışmaya başladığında kswapd adında bir işlevi çalıştırmaya başlar. Bu işlev (mm/vmscan.c dosyasında kswapd() fonksiyonu) şu işlemleri yapar:

- Bellek bölgeleri yapılarını dolaşıp bir düzenleme isteği olup olmadığına bakar
- Varsa her bölge için zone->pages_high kadar boş sayfa açılana kadar try_to_free_pages() fonksiyonunu çağırır, gerekirse sayfaların takas alanına yazılabilmesi için ilgili işlevleri uyandırır
- Uykuya yatar

Badi sayfa ayırıcısı herhangi bir bölgede zone->pages_low kadar boş sayfa kaldığında kswapd'yi uyandırır. Eğer boş sayfa sayısı zone->pages_min den az ise ayırıcı try_to_free_pages() fonksiyonunu kendisi çağırır.

Öntanımlı minimum boş bellek miktarı 128K ile 16MB arasındadır ve toplam normal belleğin karekökü alınıp sayfa adedine çevrilerek hesaplanır. Çalışan bir sistemde değiştirilebilir:

```
$ cat /proc/sys/vm/min_free_kbytes
338
$ echo “500” >/proc/sys/vm/min_free_kbytes
```

Bu değer taban alınarak her bölgenin min, low, high değerleri hesaplanır. Normal bellek bölgeleri toplam büyüklükleri ile orantılı olarak minimum boş bellekten paylarına düşen oranı min değeri olarak alırlar. HIGHMEM bölgelerinde ise en fazla 128 olmak üzere bölgenin toplam alanının 1024 te biridir. low değeri bunun iki katı, high değeri ise üç katı olarak ayarlanır.

Bellek Yetersizliği Katili

try_to_free_pages() yeterli alan açmayı başaramadığında mm/oom_killer.c içinden out_of_memory() fonksiyonunu çağırır. Bu fonksiyon,

- Eğer yeterli boş takas alanı varsa
- Son 5 saniye içinde en az 10 kere çağrılmadıysa
- Bir uygulamayı kapattıktan sonra en az 5 saniye geçmediyse geri döner.

Aksi takdirde çalışmakta olan uygulamalar içinden birini seçip kapatarak bellek kazanmaya çalışır.

Kullanıcıya en az zararı vermek için tüm uygulamalar aşağıdaki kritelere göre puanlanır ve puanı en yüksek uygulama kapatılır.

- Taban puan olarak uygulamanın kapladığı toplam bellek alanı verilir
- Bu puan saniye cinsinden işlemci kullanım süresinin karesine bölünür
- Dakika cinsinden toplam çalışma süresidir 4. köküne bölünür
- Öncelikleri azaltılmış uygulamaların puanı ikiyle çarpılır
- Uygulama root kullanıcıya aitse puan dörde bölünür
- Uygulama direk donanım erişimi yapıyorsa puan tekrar dörde bölünür

Böylece

- Tek bir kapatma işlemiyle en fazla alan kazanılmaya
- Uzun süredir çalışan uygulamaları kapatıp iş kaybına yol açmamaya
- Sistemin ve donanımın çalışmasını etkileyecek bir uygulamayı kapatmamaya çalışılır.

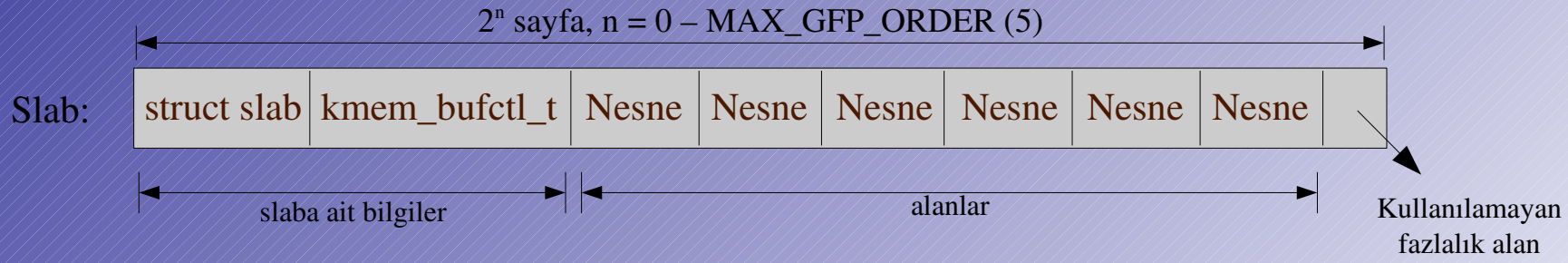
SLAB

Badi sayfa ayırıcı sayfa bloklarını gereksiz yere bölmediği, kolayca geri kazandığı için belleğin parçalanması (dış fragmentasyon) çok azdır. Üstelik sıralı yer ayırma algoritmaları (ilk-uygun, en-uygun) gibi blokların çoğalmasıyla birlikte büyük bir yavaşlama göstermez. Ancak ayırma büyüklüğü sayfa bazında olduğu için değişik boylarda çok sayıda ayırma çok büyük boşa yer harcanmasına (iç fragmentasyon) yol açacaktır.

Çekirdeğin bellek ihtiyaçları,

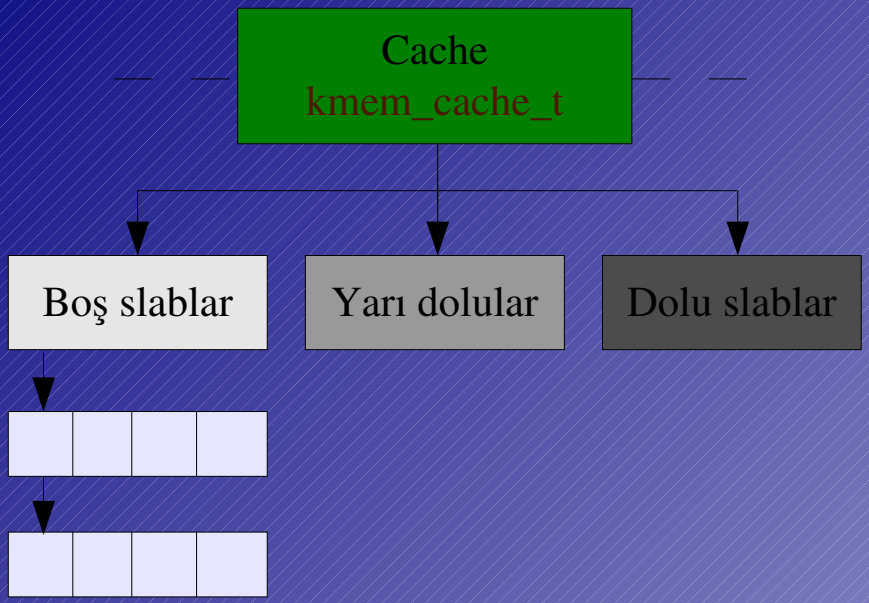
- Sabit boyda yapılardan (dosya sistemi düğümleri, ağ yapıları, vs) çok sayıda ayrılması ve geri verilmesi
- Bu yapıların kurulması ve yıkılması çok zaman aldığı için, gruplar halinde hazırlanmaları ve ihtiyaç olduğunda yeni bir tane ayırmak yerine hazır tutulan gruptan bir yapı verilmesi
- Kısa ömürlü çeşitli boylarda ufak ayırma ve geri vermeler

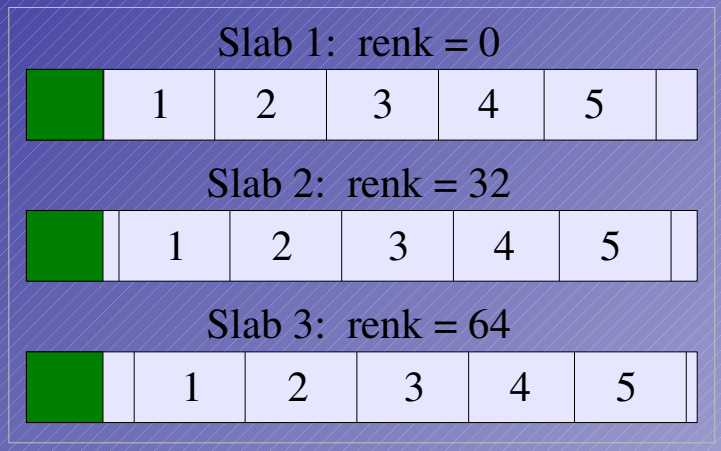
Bu ihtiyaçları karşılamak için badi ayırıcı üzerinde tek satır ayırma bloğu (Single Line Allocation Block) adı verilen daha esnek bir yer ayırıcı inşa edilmiştir. Bu ayırıcı mm/slab.c dosyası içindedir. Badiden aldığı sayfaları sabit boyda parçalara bölererek slablar oluşturur.



Slab bilgilerini içeren yapılar cache yerelliğini sağlamak için slab ile aynı sayfada tutulur, ancak nesne boyunun 512 byte ve daha büyük olması durumunda slab ayırıcının kendisine ait ayrı bir cache'de tutulur.

SLAB Yapıları





72 byte fazlalıkta renklendirme

- Slab sistemi çekirdeğin istediği her nesne tipi/boyu için bir cache açar. Bu cacheler bir cache_chain listesine eklenir. Cachelerin kendi yapıları açılışta hazırlanan bir cache_cache cachesinde tutulur.
- Ayrılacak ve geri verilecek nesneleri çabuk bulmak için her cacheye ait slablar üç ayrı grupta toplanır:
  - Sayfaları sisteme geri verilebilecek boş slablar
  - Öncelikli olarak kullanılacak yarı dolu slablar
  - Tüm nesneleri kullanımda olan slablar
- Yeni slablar oluştururken badi ayırıcıdan kullanılmayan alanı sayfa boyutunun sekizde birinden az bırakacak büyüklükte bir sayfa bloğu alınır.
- Badi ayırıcıdan ayrılan sayfalar üzerine slablar oluşturulurken, her slab için o slabtaki nesneler slabın sonunda kalan ufak boşluk ölçüsünde ve sistem hizalama (alignment) değeri kadar kaydırılır. Böylece her slabta nesnelerin ayrı bir işlemci cache satırının taşıma alanına girmesi sağlanır, bu tekniğe renklendirme (colouring) adı verilir
- Cacheye ait kurucu fonksiyon varsa her nesne üzerinde çağrılarak nesnelerin ilk hallerinde kurulmuş duruma gelmesi sağlanır.

SLAB Arabirimi

```
kmem_cache_create (const char *name, size_t size, unsigned long flags, void (*ctor)(), void (*dtor)());
kmem_cache_alloc (kmem_cache_t *cachep, int flags);
kmem_cache_free (kmem_cache_t *cachep, void *objp);
kmem_cache_destroy (kmem_cache_t *cachep);
```

```
typedef struct kmem_cache_s {
        struct list_head slabs_full;
        struct list_head slabs_partial;
        struct list_head slabs_free;
        unsigned int objsize;
        unsigned int num;
        unsigned int flags;
        unsigned int gfporder;
        unsigned int gfpflags;
        size_t colour;
        unsigned int colour_next;
        unsigned int colour_off;
        void (*ctor)();
        void (*dtor)();
        char *name;
        struct list_head next;
        . . .
} kmem_cache_t;
```

```
struct slab {
        struct list_head list;
        unsigned long colouroff;
        void *s_mem;
        unsigned int inuse;
        kmem_bufctl_t free;
};

typedef unsigned int kmem_bufctl_t;
```

kmalloc - kfree

Aşağıdaki komutla çalışmakta olan bir sistemdeki cacheler ve durumları ile ilgili bilgi alınabilir:

$ cat /proc/slabinfo

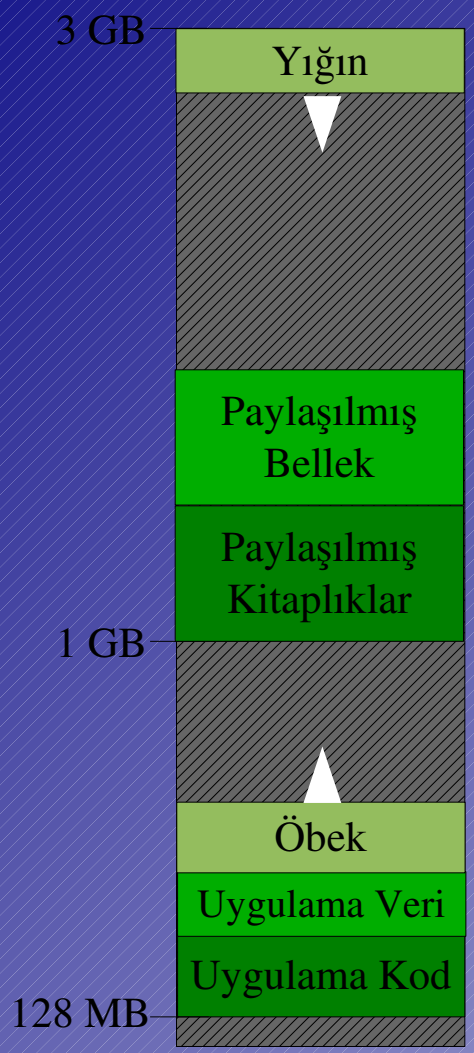
# name	<active_objs>	<num_objs>	<objsize>	<objperslab>	<pagesperslab>
tcp_bind_bucket	2	202	16	202	1
tcp_sock	6	8	960	4	1
ext2_inode_cache	7831	9603	448	9	1
inode_cache	720	726	352	11	1
task_struct	55	55	1440	5	2
. . .					
size-96	1040	1040	96	40	1
size-64	1045	1062	64	59	1
size-32(DMA)	0	0	32	113	1
size-32	1529	1582	32	113	1
kmem_cache	99	99	116	33	1
(slab adı)	(ayrılmış nesne)	(toplam nesne)	(n.boyu)	(kaç obje var)	(kaç sayfa)

Ufak boyutlu çeşitli alanları kolayca ayırıp kullanabilmek için sabit bir dizi cache standart olarak açılır (listede size-N ve size-N(DMA) olarak gözükenler). Bunların listesi include/linux/kmalloc_sizes.h de.

Bu standart cacheleri kullanarak kolayca bellek ayırmak için kmalloc() arabirimi geliştirilmiştir:

```
kmalloc (size_t size, int flags);
kfree (const void *objp);
```

Uygulama Adresleme Alanı



- Uygulamalara ait adresleme alanının ilk 128 MB kısmı, sıfır değerli adres işaretçilerinden (null pointer) kaynaklanan hataları yakalamak için bellekle eşleştirilmez.
- Hemen ardından programın kodu (code) ve yüklenme sırasında değerleri atanan veri (data+bss) kısımları eşleştirilir.
- Uygulamaların malloc() ile ayırdığı bellek öbek kısmında adreslenir.
- Öbeğin bitim noktası sys_brk() sistem çağrısı ile yükseltilerek bellekten kullanıma hazır boş alanlar ayrılır. Bu alanın genelde sıralı arama tarzı algoritmalar ile ayrılması ve uygulamanın çağrılarına verilmesi uygulamanın kullandığı dil ve standart kitaplığına aittir.
- 1 GB noktasında uygulamaların ortak kullandığı kitaplıklar gelir.
- Hemen ardından paylaşılmış bellek gelir.
- 3 GB noktasında yığın (stack) başlar. Yığın son giren ilk çıkar (LIFO) ilkesine göre çalışır ve uygulamaların çok kısa ömürlü değişkenlerini ve işlemci yazmaçlarını saklamak için kullanılır. Bellekte aşağı doğru genişler.

```
char buf[128];
int main (int ac, char **av) {
        int i, j;
        char *a = malloc (64000);
}
```

Uygulama veri alanı
Uygulama kod alanı
Yığın
Öbek

Uygulama Bellek Yapıları

Her uygulamanın bellek kullanımını tanımlayan bir mm_struct yapısı vardır (include/linux/sched.h):

```
struct mm_struct {
        // adresleme bölgeleri listesi ve sayısı:
        struct vm_area_struct * mmap;
        int map_count;
        // mm_structların listesi
        struct list_head mmlist;
        . . .
        // uygulamanın çeşitli bellek yerleşimleri
        unsigned long start_code, end_code;
        unsigned long  start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end;
        unsigned long env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
        . . .
        // mimariye özel bilgiler
        mm_context_t context;
        . . .
};
```

Uygulamanın adresleme bölgeleri: (include/linux/mm.h)

```
struct vm_area_struct {
        // ait olduğu yapı
        struct mm_struct * vm_mm;
        // başlangıç ve bitiş
        unsigned long vm_start;
        unsigned long vm_end;
        // bölge listesi
        struct vm_area_struct *vm_next;
        // koruma bayrakları (yazma, okuma, vs)
        pgprot_t vm_page_prot;
        . . .
        // paylaşılıyorsa paylaşım yapısı
        struct list_head shared;
        . . .
        // bu bölgeye ait işlemler
        struct vm_operations_struct * vm_ops;
        . . .
};
```

Kernel işlevleri her adreslemede belleğe direk erişebildikleri için kendilerine ait böyle bir yapıları yoktur.

Uygulama Bellek Yapıları

Her adresleme bölgesi kendine ait işlemlere ait bir fonksiyon listesi taşır.

```
struct vm_operations_struct {
        void (*open)(struct vm_area_struct * area);

        void (*close)(struct vm_area_struct * area);

        struct page * (*nopage)(struct vm_area_struct * area, unsigned long address, int *type);

        int (*populate)(struct vm_area_struct * area, unsigned long address, unsigned long len,
                pgprot_t prot, unsigned long pgoff, int nonblock);
};
```

Bu liste farklı adresleme bölgeleri oluşturabilmek için nesne tabanlı bir sistem sağlar.

Örneğin belleğe adreslenmiş dosya ve donanımlar için nopage fonksiyonu sayfaların ayrılıp, verilerin diskten yüklenmesi görevlerini yapan bir fonksiyon iken, normal bölgeler için sadece sayfa ayırımı yapan basit bir fonksiyon olabilir.

Sayfa Hataları

- Uygulama işlevlerinin adreslenmiş bellek alanlarına ait tüm sayfalar bellekte hazır bulunmayabilir
  - Linux ayrılan belleğe, gerçekten kullanılana kadar sayfa vermez
  - Sayfalar düşük bellek yüzünden takas alanına taşınmış olabilir
- Bir işlev ayrılmamış bir sayfaya erişmeye kalktığında işlemci (MMU) bir sayfa hatası (page fault) verir
- Hata üzerine işletim sistemi devreye girer, hatanın oluş şekline göre:
  - Bölge uygulamaya adreslenmiş ama sayfa yoksa, boş bellekten bir sayfa ayrılıp verilir
  - Bölge geçersiz ama genişleyebilir bi alanın altındaysa (yığın) alan genişletilir
  - Sayfa mevcut ama takas alanına taşınmışsa, alandan geri alınır
  - Uygulamanın erişimine izin verilmeyen bir bölgedeyse uygulama sonlandırılır (segfault)
  - Hata kernel modundayken oluşmuşsa ciddi bir kernel hatası söz konusudur
  - Eğer sadece okunabilir bir sayfa yazma erişimi yapılmışsa ve sayfa yazma-ise-kopyala (copy-on-write – COW) modunda ise sayfanın bir kopyası çıkarılır ve erişen işleve yazabileceği şekilde geri verilir.

COW sayfaları bir işlev fork() çağrısı ile kendisini çoğalttığında oluşur. Linux işlevin tüm sayfalarını çoğaltmak yerine sadece işaretler, diğer işlev herhangi bir sayfaya yazma işlemi yaptığında sadece o sayfa çoğaltılır.

Çekirdek / Uygulama Veri Taşıma

Çekirdeğin sistem çağrılarında uygulamadan aktarılan verileri sayfa hatalarını ve güvenlik erişimlerini dikkate alarak kendi alanına/uygulamaya taşıyabilmesi için iki adet yardımcı fonksiyon mevcuttur:

```
copy_from_user (void *to, const void *from, unsigned long n);
copy_to_user (void *to, const void *from, unsigned long n);
```

Sorular