Linux-käyttöjärjestelmän muistinhallinta

Jan Lindström
Linux-järjestelmän toteutus -seminaari
Helsingin yliopisto
Tietojenkäsittelytieteen laitos
Helsinki 16. maaliskuuta 1998

Muistinhallinta on käyttöjärjestelmän tärkeimpiä osia. Tietojenkäsittelyn alkuajoista lähtien on ollut tarvetta suuremmalle määrälle muistia kuin mitä järjestelmässä fyysisesti on. Fyysisen muistin rajoitukset ratkaisevia menetelmiä on kehitetty useita. Kaikkein onnistunein kehitetyistä menetelmistä on *virtuaalimuisti* (virtual memory). Virtuaalimuistin ideana on käyttää levyä keskusmuistin jatkeena, jolloin osa ohjelmista on levyllä ja osa muistissa. Virtuaalimuistin avulla järjestelmään saadaan näkyviin enemmän muistia kuin mitä fyysistä muistia on. Virtuaalimuistia jaetaan muistia tarvitsevien prosessien välillä. Muistinhallinnan muita tehtäviä on tarjota muistin suojaus, muistin varaus ja vapautus, tarjota yhteiskäyttöistä muistia sekä tarjota mekanismi muistiinkuvaukseen.

1 Johdanto

Muistinhallinta on ytimen monimutkaisimpia osia ja muisti on tärkeimpiä resursseja järjestelmässä. Siksi muistinhallinnan tehokkuus on ratkaisevaa järjestelmän tehokkaalle toiminnalle. Muistinhallinnan tehokkuuden varmistaminen on yksi Linuxin tärkeimmistä lähtökohdista [Tor97].

Muistinhallinnan avulla käyttöjärjestelmä muuntaa järjestelmän näyttämään kuin siinä olisi enemmän muistia kuin mitä siitä todellisuudessa on. Virtuaalimuisti voi olla useita kertoja suurempi kuin fyysinen muisti järjestelmässä.

Muistinhallinta tarjoaa jokaiselle prosessille oman *virtuaalimuistialueen* (virtual address space). Nämä virtuaalimuistialueet ovat täysin erillisiä. Prosessit eivät voi vaikuttaa toisten prosessien virtuaalimuistialueisiin eikä fyysisiinkään muistialueisiin. Lisäksi laitteistotason muistinhallinnan avulla voidaan suojata muistialueet kirjoitukselta. Lisää Intel x86 laitetason muistinhallinnasta ja muistinsuojauksesta löytää lähteestä [Int90, luku 5].

Muistinhallinta tarjoaa jokaiselle ajettavalle prosessille oman osuuden järjestelmän fyysisestä muistista. Vaikka muistinhallinta tarjoaa prosesseille erilliset virtuaalimuistialueet on tilanteita, joissa prosessit tarvitsevat *yhteiskäyttöistä muistia* (shared memory). Esimerkiksi järjestelmässä voi olla käynnissä useita bash-komentotulkkeja. Järjestelmän muistissa ei kannata kuitenkaan pitää useita kopioita kyseisestä ohjelmasta. Muistinkäytön kannalta on tehokkaampaa tallettaa muistiin yksi kopio ohjelmasta ja antaa kaikkien bash-ohjelmaa ajavien prosessien käyttää kyseistä muistialuetta.

Dynaamiset kirjastot (dynamic libraries) on toinen yleinen esimerkki, jossa suoritettava koodi on useiden prosessien yhteisessä käytössä. Yhteiskäyttöistä muistia voidaan käyttää myös prosessien väliseen kommunikointiin [Rus97].

Tässä paperissa kuvataan Linuxin muistinhallinnan toteutusta käyttäen lähteitä [Rus97, BBD$^+$98]. Muistinhallinnan toiminnan toteutuksen tutkiminen aloitetaan muistiin kuvauksesta. Seuraavaksi esitetään Linuxin sivutaulu, sivujen varaus ja sivujen vapautus muistista. Muistinhallinnan toiminnan tutkimista jatketaan tarvesivutuksella ja sivuvälimuistilla. Seuraavaksi käsitellään fyysisten sivujen vapautusta sekä sivutusta. Lopuksi esitetään lyhyt yhteenveto, jossa tutkitaan mitä asioita käsiteltiin ja mitkä jätettiin pois.

2 Muistiin kuvaus

Suoritettava *ohjelma* (image) täytyy lukea prosessin virtuaaliosoiteavaruuteen, jotta ohjelmaa voitaisiin suorittaa. Samalla tavalla yhteiskäyttöiset kirjastot pitää lukea prosessin virtuaaliosoiteavaruuteen. Suoritettavaa tiedostoa ei itseasiassa lueta fyysiseen muistiin vaan suoritettava tiedosto yhdistetään prosessin virtuaalimuistiin (ks. do_mmap() tiedostossa mm/mmap.c). Jos ohjelma viittaa muistisivuihin, joita ei ole fyysisessä muistissa, muistinhallinta hakee kyseiset sivut tiedostojärjestelmästä. Tätä suoritettavan ohjelman yhdistämistä prosessin osoiteavaruuteen kutsutaan *muistiinkuvaukseksi* (memory mapping).

Jokaisen prosessin virtuaalimuisti on kuvattu tietorakenteella *mm_struct* (kuva 1). Tietorakenne sisältää tiedot tällä hetkellä suoritettavasta ohjelmasta ja sisältää osoittimia *vm_area_struct* tietorakenteeseen (kuva 1).

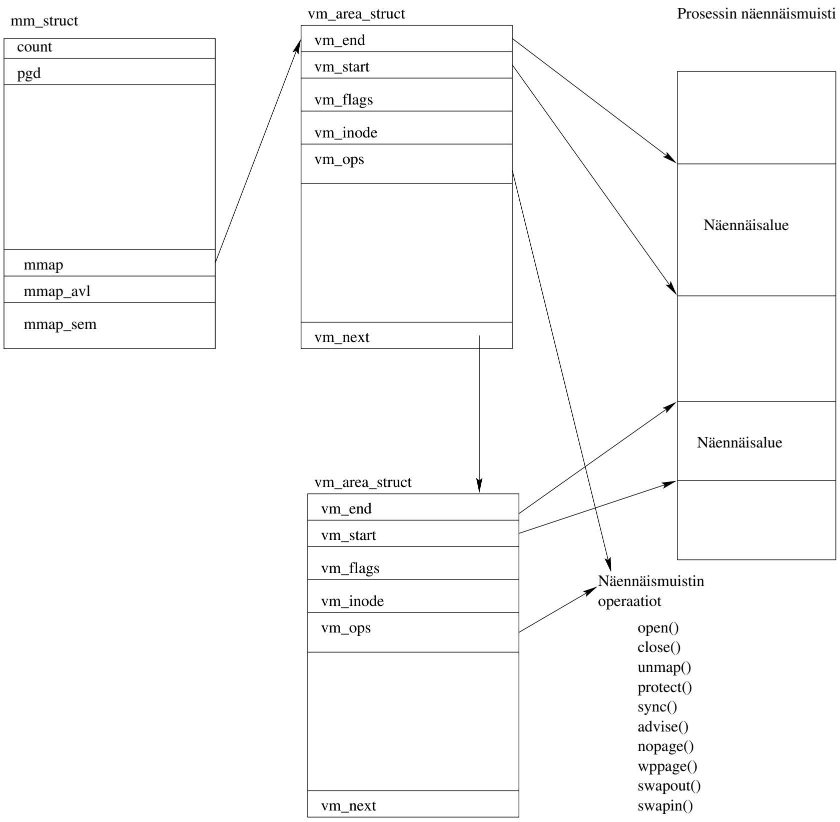
```
struct mm_struct {
    int count;   /* Käyttäjien lukumäärä                       */
    pgd_t * pgd; /* Ensimmäisen tason sivutaulun osoite        */
    unsigned long context;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack, start_mmap;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    struct vm_area_struct * mmap; /* Lista muistialueista      */
    struct vm_area_struct * mmap_avl;/* AVL-puu muistialueista */
    struct semaphore mmap_sem;
};

struct vm_area_struct {
    struct mm_struct * vm_mm; /* Suoritettavan ohjelman tiedot */
    unsigned long vm_start;   /* Alkuosoite                    */
    unsigned long vm_end;     /* Loppuosoite                   */
    pgprot_t vm_page_prot;    /* Muistialueen suojaus          */
    unsigned short vm_flags;
/* AVL-puu muistialueista osoitteen mukaan lajiteltuna         */
    short vm_avl_height;      /* Puun korkeus                  */
    struct vm_area_struct * vm_avl_left; /* Vasen alipuu       */
    struct vm_area_struct * vm_avl_right;/* Oikea alipuu       */
/* Lista muistialueista osoitteen mukaan lajiteltuna           */
    struct vm_area_struct * vm_next;
    struct vm_area_struct * vm_next_share;
    struct vm_area_struct * vm_prev_share;
    struct vm_operations_struct * vm_ops; /* Operaatiot        */
    unsigned long vm_offset;
    struct inode * vm_inode;
    unsigned long vm_pte;    /* Yhteiskäyttöinen muisti        */
};
```

Kuva 1: Tietorakenteet mm_struct ja vm_area_struct.

Jokainen *vm_area_struct* tietorakenne kuvaa virtuaalimuistialueen alun ja virtuaalimuistialueen lopun, prosessin oikeudet muistialueeseen ja joukon operaatioita kyseiseen muistialueeseen (kuvat 1 ja 2). Nämä operaatiot ovat joukko rutiineja, joita Linuxin täytyy käyttää, kun se käsittelee kyseistä muistialuetta.

Esimerkiksi virtuaalimuistialueen operaatio *nopage* suorittaa oikeat toimenpiteet, kun prosessi yrittää viitata muistialueeseen, mutta huomaa sivunpuutosvirheen kautta ettei sivu ole fyysisessä muistissa. Operaatiota *nopage* käytetään, kun muistinhallinta vaatii



Kuva 2: Virtuaalimuistin alueet.

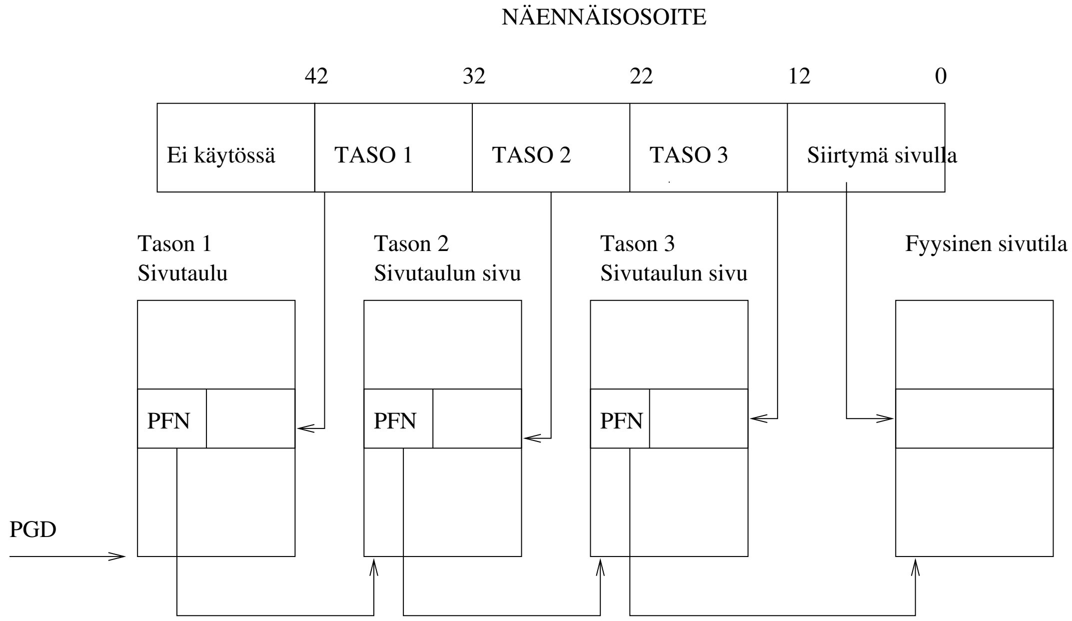
sivua tuotavaksi muistiin.

Suoritettava ohjelma kuvataan prosessin virtuaalimuistiin ja luodaan joukko *vm_area_ struct* tietorakenteita. Jokainen *vm_area_struct* tietorakenne kuvaa osaa suoritettavasta ohjelmasta; suoritettava koodi, alustetut muuttujat, alustamattomat muuttujat jne. Linux tukee joukkoa virtuaalimuistin standardioperaatioita ja luotaessa *vm_area_struct* tietorakennetta liitetään oikea joukko operaatioita tietorakenteeseen.

3 Linuxin sivutaulu

Virtuaalimuistissa osoitteet eivät ole osoitteita fyysiseen muistiin vaan ovat virtuaaliosoitteita. Prosessori muuntaa ajonaikana virtuaaliosoitteet fyysisiksi osoitteiksi käyttöjärjestelmän ylläpitämien tietorakenteiden avulla. Muunnoksen helpottamiseksi virtuaaliosoiteavaruus ja fyysinen osoiteavaruus jaetaan *sivuiksi* (pages). Sivut ovat kaikki samankokoisia. Alpha AXP järjestelmän Linux käyttää 8 kilotavun sivuja ja Intel x86 järjestelmässä käytetään 4 kilotavun sivuja [Int90, sivu 5-19].

Linux olettaa, että sivutaulussa on kolme tasoa. Jokainen sivutaulun alkio sisältää seuraavan tason *sivutilan numeron* (page frame number, PFN). Kuva 3 esittää kuinka virtuaaliosoite voidaan jakaa osiin ja kuinka jokainen osa tarjoaa siirtymän tiettyn sivutaulun sisällä (kuvassa esitetyt rajat ova Alphasta).



Kuva 3: Kolmitasoinen sivutaulu Linuxissa.

Jokainen sivutaulu on taulukko, joka sisältää osoitetun sivun perustiedot. Sivutaulun alkiolla on kuvassa 4 esitetty rakenne Intel x86 arkkitehtuurissa [Int90, sivu 5-20]. Sivutaulun alkion sisältö on kuvan 4(a) mukainen, jos sivu on fyysisesti muistissa. Sivutaulun alkion sisältö on kuvan 4(b) mukainen, jos sivu ei ole muistissa.

31	12	11							0
Sivutilan numero 31..12	AVAIL	0 0	D	A	PCD	PWT	U/S	R/W	P

P	Present-bitti
R/W	Read/Write -bitti
U/S	User/Superuser -bitti
PWT	Page write transparent
PCD	Page cache disable
A	Accessed
D	Dirty-bitti
AVAIL	Järjestelmäohjelmoijan käytettävissä

(a) Sivutaulun alkion rakenne, kun sivu on muistissa

AVAIL	0

(b) Sivutaulun alkion rakenne, kun sivu ei ole muistissa

Kuva 4: Sivutaulun alkion rakenne.

Present-bitti kertoo onko kyseinen sivu fyysisessä muistissa. Present-bitti on 0, jos sivu ei ole muistissa. Accessed-bitti on 1, jos sivuun on viitattu. Dirty-bitti on 1, jos sivun tietoja on muutettu. Read/Write-bittiä ja User-Supervisor-bittiä käytetään muistin suojauksessa, josta tarkemman kuvauksen löytää lähteestä [Int90, luku 5]. PCD-bittiä ja PWT-bittiä käytetään välimuistin hallintaan.

Prosessori muuntaa virtuaaliosoitteen fyysiseksi osoitteeksi kuvassa 5 esitetyllä algoritmillä.

```
offset = bits(41..32);
index = concat(PGD,offset);
if ( pagetable[index].p = 0 ) throw page_fault;
offset = bits(31..22);
index = concat(pagetable[index].PFN,offset);
if ( pagetable[index].p = 0 ) throw page_fault;
offset = bits(21..12);
index = concat(pagetable[index].PFN.offset);
if ( pagetable[index].p = 0 ) throw page_fault;
offset = bits(11..0);
index = concat(pagetable[index],offset);
return index;
```

Kuva 5: Virtuaaliosoitteen muunnos fyysiseksi osoitteeksi.

Jos järjestelmässä fyysisellä tasolla on vähemmän kuin kolme tasoa sivutaulussa (kuten x86), yksi taso piilotetaan toisen sisään [Tor97]. Koska osoitteenmuutos on tehtävä ajoaikana jokaiselle muistiviitteelle, on muunnoksen oltava nopea. Tämän vuoksi säilytetään viimeksi muunnoksessa tarvittuja tietoja *muistinhallintayksikön* (memory management unit, MMU) *osoitteemuunnospuskurissa* (translation lookaside buffer, TLB).

Järjestelmässä on useita tarpeita fyysisille sivutiloille. Esimerkiksi käyttöjärjestelmän täytyy varata sivuja luettaessa ohjelma muistiin. Sivut vapautetaan kun ohjelma on päättynyt. Muita tarpeita fyysisille sivutiloille on käyttöjärjestelmän tietorakenteiden talletus. Tällaisia tietorakenteita ovat sivutaulut itse. Algoritmit ja tietorakenteet sivutilojen varaukseen ja vapautukseen ovat kaikkein tärkeimpiä muistijärjestelmän tehokkuuden varmistamisessa.

Kaikki sivutilat on kuvattu Linux-järjestelmässä *mem_map* tietorakenteessa, joka on lista *mem_map_t* tyyppisiä alkioita (kuva 6) ja alustettu käynnistyksessä. Jokainen *mem_map_t* alkio kuvaa yhden fyysisen sivun järjestelmässä.

```
typedef struct page {
  struct page *next;              /* Seuraava sivu                      */
  struct page *prev;              /* Edellinen sivu                     */
  struct inode *inode;            /* Missäpäin levyä tässä sivutilassa */
  unsigned long offset;           /* oleva sivu sijaitsee               */
  struct page *next_hash;         /* Seuraava hajautustaulun alkio      */
  atomic_t count;                 /* Käyttäjien lukumäärä               */
  unsigned flags;                 /* Liput                              */
  unsigned dirty:16,              /* Onko sivua muutettu                */
    age:8;                        /* Sivun ikä                          */
  struct wait_queue *wait;        /* Lukittua sivua odottavat täällä    */
  struct page *prev_hash;         /* Edellinen hajautustaulun alkio     */
  struct buffer_head * buffers;   /* Puskurointitietorakenne            */
  unsigned long swap_unlock_entry;/* Sivutustilan sivunumero            */
  unsigned long map_nr;           /* Sivuntilan numero                  */
} mem_map_t;
```

Kuva 6: Sivutilojen tietorakenne.

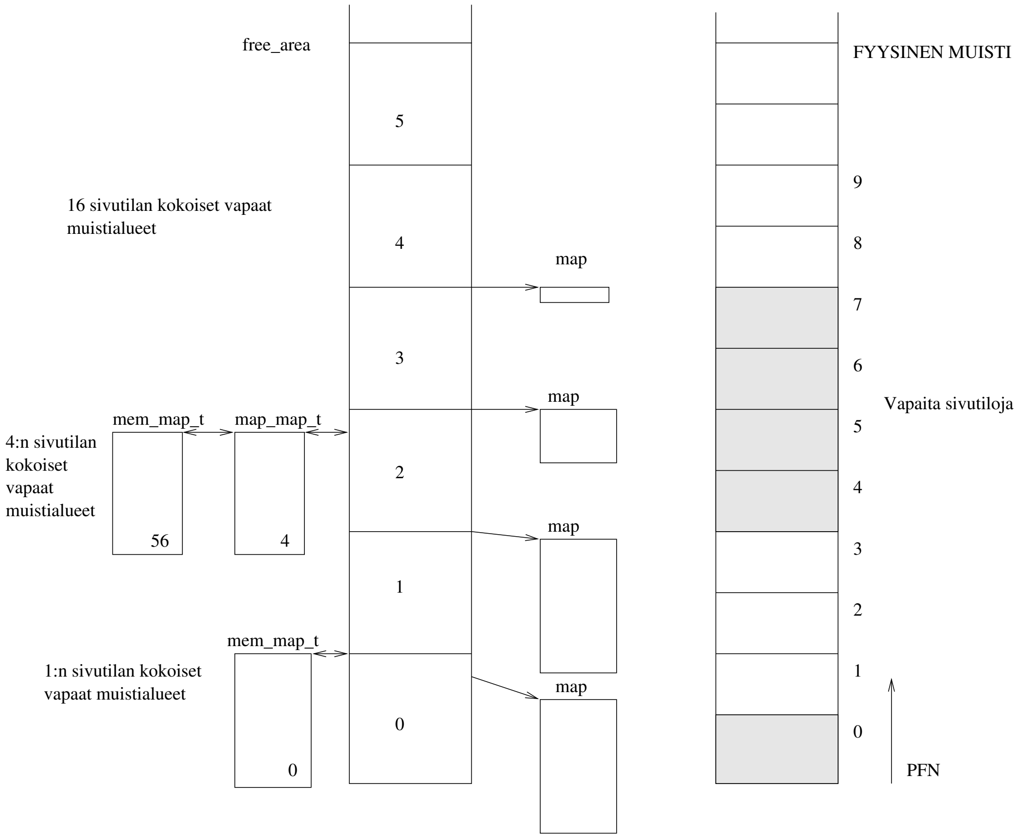
Taulukkoa *free_area* käytetään vapaiden sivutilojen etsimiseen (kuva 7). Koko puskurihallintamenetelmä tarjotaan tällä mekanismilla. Jokainen tietorakenteen *free_area* alkio sisältää tiedot muistialueista. Ensimmäinen alkio taulukossa kuvaa yksittäisiä sivutiloja. Seuraava kahden sivun muistialueita ja seuraava 4 sivun muistialueita jatkuen aina kahden potensseissa. Vapaat muistialueet ovat kahteen suuntaan linkitettynä jonona *list* tietorakenteessa. Osoitin *map* osoittaa bittikarttaan, joka pitää kirjaa varatuista muistialueista. Bitti N on asetettu, jos N:s muistialue on vapaa.

```
struct free_area_struct {
    struct page *next;      /* List tietorakenne jonoa varten       */
    struct page *prev;
    unsigned int * map;     /* Bittikartta vapaista muistialueista  */
};

static struct free_area_struct free_area[NR_MEM_LISTS = 6];
```

Kuva 7: Tietorakenne free_area.

Kuvassa 8 on tietorakenteen *free_area* rakenne. Alkiossa 0 on yksi vapaa sivutila ja alkiossa 2 on 2 4:n sivutilan kokoista aluetta. Ensimmäinen vapaa sivutila alkaa sivutilasta 4 ja toinen sivutilasta 56.



Kuva 8: Tietorakenne free_area.

3.1 Sivutilojen varaus

Linux käyttää Buddy-algoritmia [Kno65] tehokkaaseen muistialueiden varaukseen ja vapautukseen (ks. _get_free_pages() tiedostosta mm/page_alloc.c). Muistialuuen varauksessa varataan yksi sivutila tai useampia peräkkäisiä sivutiloja. Sivutilat varataan muistialueina, jotka ovat kooltaan 2 potensseja. Eli voidaan varata yksi sivutila, 2 sivutilaa, 4 sivutilaa jne. Järjestelmä varaa muistialueita, jos tarpeeksi vapaita sivutiloja on jäljellä (*nr_free_pages* > *min_free_pages*). Tietorakenteesta *free_area* haetaan pyydetty määrä sivutiloja. Jokainen *free_area* tietorakenteen alkio sisältää *kartan* (map) varatuista ja vapaista kyseisen kokoisista muistialueista. Esimerkiksi taulukon alkiossa 2 on muistikartta, joka kuvaa vapaat ja varatut 4 sivun pituiset muistialueet.

Ensin haetaan pyydetyn kokoista muistialuetta. Algoritmi läpikäy listan vapaista muistialueista, jotka on jonotettu *list* tietorakenteeseen tietorakenteessa *free_area*. Jos yhtään pyydetyn kokoista muistialuetta ei ole vapaana, jatketaan seuraavasta koosta. Jos löydetty vapaa muistialue on pyydettyä suurempi, täytyy löydetty muistialue jakaa osiin kunnes saadaan oikean kokoinen muistialue. Koska muistialueiden koot seuraavat 2 potensseja, on muistialueiden osiin jako helppoa. Muistialue jaetaan kahteen osaan tarvittavan monta kertaa. Jaetut vapaat muistialueet lisätään oikeaan paikkaan tietorakenteessa *free_area* ja varattu muistialue palautetaan kutsujalle.

Esimerkiksi jos kuvan 8 tilanteessa pyydetään kaksi sivutilaa, ensimmäinen neljän sivun muistialue jaettaisiin kahteen osaan. Ensimmäinen osa alkaen sivutilanumerosta 4 palautettaisiin kutsujalle varattuna muistialueena ja toinen vapaa muistialue alkaen sivutilanumerosta 6 lisättäisiin tietorakenteen *free_area* alkioon 1.

3.2 Sivutilojen vapautus

Muistialueiden varaus aiheuttaa muistin pirstoutumista, koska suurempia vapaita muistialueita jaetaan pienemmiksi. Sivujen vapautus kokoaa sivut suuremmiksi muistialueiksi aina kun kokoaminen on mahdollista (ks. free_pages() tiedostosta mm/page_alloc.c). Muistialueen vapauttamisen yhteydessä tarkistetaan onko viereinen muistialue vapaana. Jos viereinen muistialue on vapaana, yhdistetään se vapautetun muistialueen kanssa, jolloin saadaan muodostettua uusi yhtä kokoa suurempi vapaa muistialue. Yhdistämistä jatketaan aina suuremmiksi muistialueiksi, jos yhdistäminen on mahdollista. Tällä tavalla vapaat muistialueet ovat aina niin suuria kuin muistinkäyttö sallii.

Esimerkiksi jos kuvassa 8 sivutilanumero 1 vapautettaisiin, muistialue yhdistettäisiin jo vapaana olevan alueen 0 kanssa. Syntynyt muistialue lisättäisiin tietorakenteen *free_area* alkioon 1, jolloin muodostuu 2:n sivutilan kokoinen vapaa alue.

4 Tarvesivutus

Kun suoritettava ohjelma on kuvattu prosessin virtuaalimuistiin, voi ohjelman suoritus alkaa. Koska vain suoritettavan ohjelman alku on fyysisesti muistissa, ohjelma viittaa pian muistialueisiin, jotka eivät ole fyysesessä muistissa. Tämän seurauksena prosessori aiheuttaa sivunpuutoskeskeytyksen (ks. handle_mm_fault() tiedostossa mm/memory.c). Sivunpuutoksen käsittelyn tuloksena syntyy tietorakenne, joka kuvaa muistiosoitteen, missä sivuvirhe tapahtui ja käytetyn muistiosoituksen tyypin.

Muistinhallinnan pitää löytää *vm_area_struct* tietorakenne (ks. kuvat 1 ja 2), joka vastaa sivunpuutoksen tapahtumapaikan muistialuetta. Koska *vm_area_struct* tietorakenteen läpikäynti on sivunpuutosvirheiden käsittelyn tehokkuuden kannalta kriittistä on *vm_area_struct* tietorakenteet talletettu AVL-puuhun (Adelson-Velskii and Landis) [AVL62]. Jos *vm_ area_ struct* tietorakennetta ei löydy tälle sivunpuutosvirheen muistiosoitteelle, prosessi on viitannut virheelliseen osoitteeseen. Prosessille ilmoitetaan SIGSEGV-signaalilla ja jos prosessilla ei ole signaalikäsittelijää kyseiselle signaalille, prosessi keskeytetään.

Seuraavaksi muistinhallinta vertaa sivunpuutosvirheen tyyppiä kyseisen muistialueen sallittuihin operaatiotyyppeihin. Jos prosessi yrittää kiellettyä operaatiota, esimerkiksi kirjoittaa kirjoitussuojatulle muistialueelle, prosessille signaloidaan SIGSEGV-signaalilla.

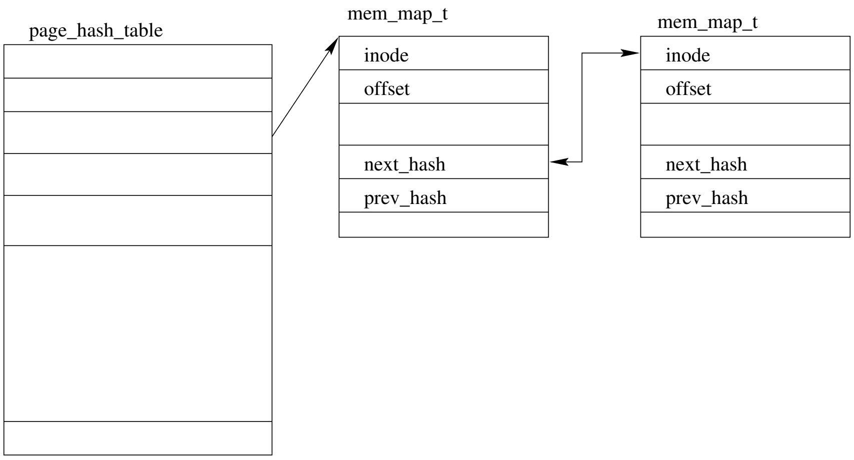
Jos muistinhallinta havaitsee sivunpuutosvirheen lailliseksi, täytyy sivunpuutosvirhe käsitellä (ks. do_no_page() tiedostossa mm/memory.c). Muistinhallinnan täytyy erottaa sekä sivut, jotka ovat *sivutustiedostossa* (swap file) että sivut, jotka ovat osa suoritettavaa ohjelmaa levyllä jossain. Sivut erotetaan käyttämällä kyseisen sivunpuutosvirheen muistiosoitteen sivutaulua.

Jos sivutaulun alkion mukaan sivu ei ole fyysisessä muistissa (present = 0), on sivu talletettu tällä hetkellä sivutustiedostoon. Tässä tapauksessa sivutaulun alkion sivutilanumero sisältää tiedon sivutustiedostosta, jossa kyseinen sivu sijaitsee. Se kuinka sivutustiedoston sivut käsitellään kuvataan myöhemmin.

Linuxin geneeristä *nopage()* operaatiota käytetään muistiin kuvattaviin ohjelmiin. Operaatio *nopage* hakee *sivuvälimuistista* (page cache) tarvitun sivun fyysiseen muistiin (ks. filemap_nopage() tiedostossa mm/filemap.c).

5 Sivuvälimuisti

Linuxin sivuvälimuistin tehtävänä on nopeuttaa levyllä oleviin tiedostoihin viittaamista. Muistiin kuvatut tiedostot luetaan sivu kerrallaan ja luetut sivut talletetaan sivuvälimuistiin. Kuvasta 9 nähdään sivuvälimuistin sisältävän taulukon *page_hash_table* osoittimia tietorakenteeseen *mem_map_t* (ks. kuva 6). Jokainen tiedosto Linuxissa tunnistetaan *virtuaalitiedostojärjestelmän* (virtual file system, VFS) tietorakenteen *inode* avulla. Jokainen virtuaalitiedostojärjestelmän tietorakenne *inode* on yksilöllinen ja kuvaa vain yhden tiedoston.



Kuva 9: Linuxin sivuvälimuisti.

Jos sivu luetaan muistiinkuvatusta tiedostosta, lukeminen tapahtuu sivuvälimuistin kautta. Jos sivu on jo sivuvälimuistissa, osoitin *mem_map_t* tietorakenteeseen palautetaan sivunpuutosvirheen käsittelyyn. Muussa tapauksessa sivu täytyy tuoda muistiin tiedostojärjestelmästä, jossa sivu sijaitsee. Muistinhallinta varaa fyysisen sivun ja lukee sivun tiedostosta.

Linux pyrkii aloittamaan seuraavan sivun lukemisen välittömästi, jos mahdollista. Yhden sivun ennakolta lukeminen merkitsee sitä, että prosessin lukiessa tiedoston sivuja peräkkäisjärjestyksessä on seuraava sivu valmiina odottamassa muistissa.

6 Muistitilan vapautus

Jos fyysisen muisti on käymässä liian pieneksi, muistinhallinnan täytyy yrittää vapauttaa fyysisiä sivutiloja. Ytimen *sivutusdaemonprosessin* (swap daemon) kswapd:n tehtävänä on säilyttää Linuxin käytössä riittävän paljon vapaita sivutiloja. Ytimen sivutusdaemonprosessi on erityinen prosessi, *ytimen säie* (kernel thread), jolla ei ole virtuaalimuistia (ks. kswapd() tiedostossa mm/vmscan.c). Sen sijaan se suoritetaan suojatussa tilassa fyysisellä muistiavaruudella. Sivutusdaemonin tehtävänä on pitää muistinhallintajärjestelmä tehokkaasti toiminnassa. Sivutusdaemon käynnistetään ytimen init-prosessissa käynnistyksen yhteydessä. Sivutusdaemon odottelee ytimen sivutusajastimen määräaikaista laukeamista. Kun ajastin laukeaa, sivutusdaemon tutkii onko vapaiden sivutilojen lukumäärä pudonnut liian alas. Vapaita sivutiloja on liian vähän jos:

```
(nr_free_pages + nr_async_pages < free_pages_low ) ||
(nr_free_pages + nr_async_pages < free_pages_high)
```

Muistinhallinta ei kirjoita liian montaa sivua sivutustiedostoon samalla kertaa. Muistinhallinta käyttää nr_async_pages muuttujaa säilyttämään tällä hetkellä sivutustiedostoon kirjoitettujen sivujen lukumäärää. Kun järjestelmä käynnistetään, alustetaan muuttujat $free\_pages\_low$ ja $free\_pages\_high$, jotka kuvaavat vapaiden sivutilojen lukumäärän järjestelmässä. Jos riittävästi vapaita sivutiloja on jäljellä, sivutusdaemon palaa nukkumaan. Muussa tapauksessa sivutusdaemon yrittää kolmella tavalla vähentää käytössä olevien fyysisten sivujen lukumäärää:

1. Vähentämällä puskurivälimuistien ja sivuvälimuistien kokoa,
2. sivuttamalla jaettuja sivuja ulos, tai

3. sivuttamalla tai hävittämällä sivuja.

6.1 Sivutus levylle

Sivutusdaemon tutkii järjestelmän jokaisen prosessin kerrallaan ja tutkii onko kyseinen prosessi hyvä ehdokas sivutukseen. Hyviä ehdokkaita ovat prosessit, jotka voidaan sivuttaa (esimerkiksi kswapd prosessia ei voi). Prosessi on hyvä ehdokas, jos sillä on yksi tai useampia sivuja, jotka voidaan sivuttaa tai hävittää muistista. Sivut sivutetaan vain jos, sivujen tietoja ei voida hakea muilla tavoilla.

Kun sivutusdaemon löytää prosessin sivutusta varten, sivutusdaemonin täytyy tutkia kyseisen prosessin kaikki muistialueet. Se etsii muistialueita, jotka eivät ole yhteiskäytössä tai lukittuja (ks. swap_out_vma() tiedostosta mm/vmscan.c). Jokaisella sivulla on ikä (age), joka kertoo sivutusdaemonille jonkinlaisen tiedon kannattaako sivu sivuttaa.

Sivut ikääntyvät jos niihin ei viitata. Sivutusdaemon sivuttaa vain vanhat käyttämättömät sivut (*age* = 0). Sivua varattaessa asetetaan oletusiäksi kolme. Joka kerta, kun sivuun viitataan ikä kasvaa kolmella kunnes maksimi 20 saavutetaan. Sivun ikää vähennetään yhdellä joka kerta, kun sivutusdaemonin käsittelee kyseistä sivua. Nämä oletusarvot voidaan vaihtaa ja siksi ne on talletettu tietorakenteeseen *swap_control*.

Sivutusdaemon varaa sivun sivutustiedostosta ja kirjoittaa sivun kyseiselle laitteelle. Sivun sivutaulun alkio korvataan alkiolla, jossa present-bitti on asetettu nollaksi ja sivutilanumero viittaa käytettyyn sivutustiedoston sivuun. Jos sivutiloja on riittävästi vapaana, sivutusdaemon palaa nukkumaan ja seuraavalla käynnistyksellä aloittaa tutkimisen seuraavasta prosessista.

6.2 Sivujen takaisinhaku

Sivutiedostoon kirjoitettua sivua saatetaan tarvita uudestaan, jos prosessi viittaa sivutettuun muistialueeseen sen jälkeen kun alue on talletettu sivutustiedostoon. Tapahtumasta seuraa sivunpuutosvirhe. Sivutaulun perusteella sivutusdaemon tietää, että kyseinen sivu on sivutettu

Sivutusdaemon käyttää sivutaulun avulla löytyvän muistialueen operaatiota *swapin*, jos operaatio löytyy (ks. do_swap_page() tiedostosta mm/memory.c). Muuten Linux käsittelee sivutetun sivun geneerisellä *swapin* operaatiolla (ks. swap_in() tiedostosta mmpage_alloc.c). Linux varaa vapaan fyysisen sivun ja lukee sivutetun sivun muistiin.

Jos sivunpuutosvirheen aiheuttaja ei ollut kirjoituspyyntö, sivua säilytetään *sivutusvälimuistissa* (swap cache). Sivua ei merkitä kirjoitettavaksi. Jos sivuun kirjoitetaan, syntyy uusi sivunpuutosvirhe ja tällöin sivu merkitään muutetuksi ja sivu poistetaan sivutusvälimuistista.

7 Yhteenveto

Muistinhallinta on ytimen monimutkaisimpia osia ja muisti on tärkeimpiä resursseja järjestelmässä. Siksi muistinhallinnan tehokkuus on ratkaisevaa järjestelmän tehokkaalle toiminnalle. Linuxin muistinhallinnassa on käytetty erittäin tehokkaita ratkaisuja muistinhallinnan toteutuksessa. Tässä paperissa on keskitytty kuvaamaan muistinhallinnan toteutusta ytimen tasolla. Muistinhallintaan liittyy myös laitteistotason toteutus, jonka tarkempi käsittely on jätetty tästä esityksestä pois.

Viitteet

[AVL62] G. M. Adelson-Velskii and E. M. Landis. An algorithm for organization of information. *Soviet Math. Doclady 3*, pages 1259–1263, 1962.

[BBD+98] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R Magnus, and D. Verworner. *Linux Kernel Internals, second edition*. Addison-Wesley, 1998. ISBN 0-201-33143-8.

[Int90] Intel. *i486 Programmers Reference Manual*. Osborne McGraw-Hill, 1990.

[Kno65] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, 1965.

[Rus97] D. A. Rusling. *The Linux Kernel*. DRAFT, VERSION 0.1-13(19), LinuxHq Project, ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel/, tarkistettu 16.2.1998, July 1997.

[Tor97] L. Torvalds. Linux: A portable operating system. Master's thesis, University of Helsinki, Department of Computer Science, Serie of Publications C, Report C-1997-12, 1997.