

KERTTULIN LUKIO

ICT-LINJAN PÄÄTTÖTYÖ

**UNIX-tyylinen käyttöjärjestelmäydin
sekä ISO/IEC 9899:1999 ja
POSIX.1-2008 -standardien mukainen
C-kirjasto: RazOS**

Eetu Pesonen ja Iiro Rastas

ICT -linja

28. huhtikuuta 2016

UNIX-TYYLINEN KÄYTTÖJÄRJESTELMÄYDIN SEKÄ ISO/IEC 9899:1999 JA POSIX.1-2008 -STANDARDIEN MUKAINEN C-KIRJASTO: RAZOS

Pesonen, Eetu & Rastas, Iiro

Kerttulin Lukio

ICT -linja

28. huhtikuuta 2016

Sivumäärä: 28 (38)

Liitteitä: 1

Asiasanat: Käyttöjärjestelmä, UNIX, POSIX, Ohjelmointi

Käyttöjärjestelmät ovat monimutkaisia kokonaisuuksia. Työn tarkoitus on selventää käyttöjärjestelmän toimintaa ja toteutusta. Työssä kuvataan UNIX-tyylisen käyttöjärjestelmän teknistä toteutusta erityisesti prosessien, niiden hallinnan ja ajanjaon, muistinhallinnan sekä käyttäjän käytettävissä olevien rajapintojen kannalta. Teknisen toteutuksen kuvausta varten laadittiin UNIX-tyylinen käyttöjärjestelmä, RazOS. Toteutuksen kuvaukseen käytetään apuna lähdekoodia sekä teknisiä yksityiskohtia selventäviä kaavioita.

RazOS ohjelmoitiin C ja Assembly -ohjelmointikielillä, jotka ovat perinteisiä ohjelmointikieliä käyttöjärjestelmän ohjelmointiin. Alunperin C kehitettiin UNIX-käyttöjärjestelmän kehittämistä varten, joten se on erityisen sopeva ohjelmointikieli työn tarkoituksen kannalta. Alustaksi valittiin laajalti käytössä oleva Intelin x86 -arkkitehtuuri, koska yleisyytensä takaa sille on helpompi laatia käyttöjärjestelmä kuin esimerkiksi ARM -arkkitehtuureille. Toinen syy x86:n valintaan oli sen ikä; sitä on käytetty lähes UNIXin alkua ajoista asti, aina vuodesta 1978.

Ohjelmointityön tuloksena saatiin aikaiseksi käyttöjärjestelmä, joka tukee osittain POSIX ja C99 -standardien mukaista C-kirjastoa. Sille tehtiin muutama ohjelma sekä komentotulkki, Rash. RazOSin kehitystyö jatkuu edelleen, ja tavoitteena on täysin standardien mukainen, UNIX-tyylinen käyttöjärjestelmä.

UNIX-LIKE OPERATING SYSTEM KERNEL AND ISO/IEC 9899:1999 AND POSIX.1-2008 COMPLIANT C-LIBRARY: RAZOS

Pesonen, Eetu & Rastas, Iiro

Kerttulin Lukio

ICT -linja

28. huhtikuuta 2016

Number of pages: 28 (38)

Appendices: 1

Keywords: Operating System, UNIX, POSIX, Programming

Operating systems are complex entities. The purpose of this work is to examine the workings and the implementation of an operating system. In this work we describe the technical implementation of a UNIX-like operating system, drawing particular attention to processes, their control and time-sharing, as well as to memory management and the interfaces available to the user. To illustrate the technical implementation, we have developed a UNIX-like operating system, RazOS. We use source code and figures of technical details to help illustrate this implementation.

RazOS was programmed in traditional system programming languages, namely C and Assembly. C was originally created for the development of the UNIX operating system, so it is particularly suitable for work of this nature. We chose the widely used Intel x86 architecture as our target platform, since its prevalence makes it easier to develop an operating system to than some other architectures, such as ARM. The second reason for choosing x86 was its age; it has been in use since the infancy of UNIX, ever since 1978.

Our development work has resulted in an operating system that partially supports the C library defined by the POSIX and C99 standards. We have developed some programs, as well as a shell, Rash. RazOS is actively developed, and our goal is a completely standards-compliant, UNIX-like operating system.

Sisällysluettelo

Tiivistelmä	iii
Abstract	v
Käytetyt merkinnät ja lyhenteet	ix
1 Johdanto	1
1.1 Käyttöjärjestelmän tehtävät	1
2 Käyttöjärjestelmäydin	3
2.1 Yleistä	3
2.2 Käynnistys	3
2.2.1 Ytimen ensimmäiset askeleet	3
2.3 Virtuaalimuisti ja muistinhallinta	4
2.3.1 Virtuaalimuistin hallinta	5
Sivutus	5
Muistin varaaminen	7
Muistin vapauttaminen	8
2.4 Keskeytykset	9
2.4.1 Poikkeukset	10
Sivutusvirhe	10
2.4.2 Laitteistokeskeytykset	10
2.5 Moniajo	11
2.5.1 Moniajo UNIX-varianteissa	11
fork-exec	11
Ohjelman lataus	12
Prosessin vaihto	12
2.6 Tiedostojen hallinta	14
2.7 Järjestelmäkutsut	15
3 Käyttäjätila	17
3.1 Standardityökalut (stdlib.h)	17
3.1.1 Dynaaminen muistinvaraus (malloc)	18
3.1.2 Ympäristömuuttuja (environ)	18
3.1.3 Satunnaisluvut (rand)	18
3.2 Syöte ja tulostus (stdio.h)	18
3.3 Komentotulkki	19
3.4 Esimerkki standardinmukaisuudesta	20
4 Yhteenveto	21
4.1 Ajankäyttö ja työnjako	21
4.2 Mitä opimme	21
Liite A RazOSin järjestelmäkutsut	23
Kirjallisuus ja lähteet	27

Käytetyt merkinnät ja lyhenteet

API Application Programming Interface, ohjelmointirajapinta
0x Etuliite heksadesimaaleille

1 Johdanto

Käyttöjärjestelmiä on useita, joista tunnetuimpia ovat Microsoft Windows, Applen Mac OS X sekä erinäiset Linux-versiot. Linux ja Mac OS X pohjautuvat teknisesti 1970-luvulla kehitettyyn UNIX-käyttöjärjestelmään, jota kehittivät Bell Labs -tutkimuslaitoksessa Dennis Richie ja Ken Thompson. Tässä työssä tarkastellaan käyttöjärjestelmän tehtäviä ja teknistä toteutusta UNIXin näkökulmasta tämän työn ohessa kehitetyn UNIX-tyylisen käyttöjärjestelmän, RazOSin, avulla.

Ajatus tarkemmasta käyttöjärjestelmän toiminnan tutkimisesta syntyi, kun käyttäjätason ohjelmia tehdessä käyttöjärjestelmä ja ohjelmointikirjastot vaikuttivat ”mustalta laatikolta”. Halusimme selvittää, miten laitteisto, käyttöjärjestelmä, kirjastot ja ohjelmat toimivat yhdessä. Tähän projektiin ei siis kuulu juurikaan käyttäjätason ohjelmia, vaan käyttöjärjestelmä itsessään, ja lisäksi osittain standardi C-kirjasto. Joitain käyttäjämaailman ohjelmia on tosin tehty, testausmielessä. Periaatteessa POSIX ja C99 -standardien mukaisten ohjelmien pitäisi RazOSissa toimia, hyvin pienellä muokkauksella.

RazOSin kehitys tapahtui GNU/Linux ympäristössä. Koodi kirjoitettiin GNU Emacs ja Vim -editoreilla, ja kääntämiseen käytettiin GNU Compiler Collectionin versiota 5.2.0 ja linkitykseen GNU ld -työkalua GNU Binutils -kokoelman versiosta 2.25.1. Kääntämisen apuna oli GNU Make. RazOSia ei ole vielä ajettu ”oikeassa” tietokoneessa, vaan testaamiseen on käytetty emulaattoreita Qemu ja Bochs.

RazOSin kehitystyö jatkuu edelleen, ja tavoitteena on täysin (*POSIX.1-2008*) ja (*ISO/IEC 9899:1999*) -standardeja tukeva käyttöjärjestelmä ohjelmointikirjastoineen. Suunnitelmissa on jatkaa kehitystä, kunnes RazOS on niin sanottu ”self-hosting”-käyttöjärjestelmä, eli se pystyy kääntämään itsensä ja kehitystyö voidaan siten siirtää GNU/Linux -ympäristöstä RazOSiin. Se vaatii vielä paljon työtä, mutta perusasiat ovat kunnossa: RazOS tukee monipuolista muistinhallintaa, keskeytyksiä, moniajoa ja tiedostoja sekä osaa käsitellä aikaa. Lisäksi siinä on erilliset ydin- ja käyttäjätilat, mikä on modernille käyttöjärjestelmälle tärkeä ominaisuus. Paljon laitteistoajureita puuttuu vielä, ja kirjastoista on suuria osia kesken, mutta niiden kehitys jatkuu. Kuten sanottu, tähän työhön ei varsinaisesti kuulu käyttäjän käytettävissä olevat ohjelmat, mutta niiden teko standardeja noudattaen on mahdollista; kokeilemamme ohjelmat toimivat niin GNU/Linux -ympäristössä kuin RazOSissakin.

1.1 Käyttöjärjestelmän tehtävät

Käyttöjärjestelmä on ohjelmisto, joka mahdollistaa muiden ohjelmien toiminnan hallinnoimalla tietokoneen resursseja, kuten muistia. Se antaa ohjelmien käyttöön yhtenäisen rajapinnan, API:n, joka on riippumaton alustasta. Rajapintaan kuuluu monia järjestelmäkutsuja, jotka toimivat erinäisten abstrahointien, kuten tiedostojärjestelmän ja virtuaalimuistin, kanssa.

UNIX-maailmassa The Open Groupin ja IEEE Computer Society:n julkaisema POSIX-standardi määrittelee järjestelmäkutsuja ja laajentaa niiden avulla International Organization for Standardizationin ja International Electrotechnical Commissionin standardisoimaa C-kirjastoa. Standardisoidun C-kirjaston avulla ohjelmoijien on mahdollista kehittää ohjelmia, jotka toimivat kaikissa standardien mukaisissa käyttöjärjestelmissä, laitteistosta riippumatta.

Käyttöjärjestelmä myös pitää huolta käyttäjän ja ohjelmien turvallisuudesta. Se esimerkiksi tarkistaa, onko käyttäjällä ja/tai ohjelmalla oikeus käyttää pyytämäänsä tiedostoa, ja suojelee muiden ohjelmien muistialueita virtuaalimuistinhallinnan keinoin, etteivät ohjelmat käytä muistia, johon niillä ei ole lupaa koskea.

Virheidenhallinta ja niistä selviäminen ovat myös käyttöjärjestelmän tehtäviä. UNIXeissa virheidenhallinta on tavallisesti hoidettu *signaaleilla*, joita käyttöjärjestelmä antaa C-kirjastolle virheen sattuessa. Kirjasto hoitaa yleensä virheet itse, usein lopettamalla ohjelman suorituksen. Käyttäjä voi myös itse määrittää signaaleille käsittelyfunktion ISO-standardiin kuuluvalla `signal()`-funktioilla.

2 Käyttöjärjestelmäydin

2.1 Yleistä

Perinteisesti käyttöjärjestelmäytimet, kernelit, jaetaan monoliittisiin- ja mikroytimiin. Monoliittinen ydin sisältää kaikki ytimen toiminnot yhdessä osoiteavaruudessa, yhdessä kokonaisuudessa. Se on usein tehokkaampi ja helpompi kehittää kuin mikroydin, joka koostuu monesta osasta. Mikroytimeen perustuva käyttöjärjestelmä koostuu *moduuleista*, joita pelkistetty *mikroydin* ohjaa. Moduulit ovat usein käyttäjätilassa, englanniksi userspace, ydintilan (kernel space) sijaan. Koostuessaan useasta moduulista kokonaisuus on usein vakaampi kuin monoliittinen ydin, mutta hitaampi ja monimutkaisempi. UNIX-johdannaiset, kuten Linux, ovat yleensä monoliittisiä.

2.2 Käynnistys

X86-tietokoneen käynnistyessä prosessori alkaa suorittaa koodia BIOSin ROM -muistista. BIOSin avulla etsitään ja otetaan käyttöön muisti ja kaikki laitteet ja saatetaan laitteisto sellaiseen tilaan, että ohjaus voidaan siirtää käynnistyslataajalle (bootloader) ja edelleen käyttöjärjestelmän ytimelle. Käynnistyslataajan toimintaan ei tässä tutustuta tarkemmin.

2.2.1 Ytimen ensimmäiset askeleet

Käynnistyslataaja siirtää ohjauksen ytimelle, eli paremmin sanottuna siirtää prosessorin suorittamaan ytimen koodia ladattuaan sen massamuistista RAM-muistiin. RazOSin tapauksessa prosessori siirtyy suorittamaan koodia osoitteesta 0x100000, eli yhden megatavun kohdalta muistin alusta laskeen. Aluksi luodaan ja alustetaan pino, stack, otetaan käyttöön liukulaskeentaprosessori, FPU, ja lopuksi siirrytään suorittamaan C-koodia (listaus 1).

Siirryttyään C-koodin puolelle, alkaa prosessori suorittaa kmain()-funktia. Sen avulla alustetaan laitteistoa ja otetaan käyttöön virtuaali-muisti. Lopuksi siirrytään käyttäjätilaan.

```

start:
    mov esp, stack_end    ; Luo ja alusta pino eli stack
    push dword 0
    push dword 0
    mov ebp, esp

    ;; Anna pinon alku ja multiboot-tiedot kmain():lle
    push esp
    push ebx

    fninit                ; Ota käyttöön FPU
    mov eax, cr0
    or  eax, 1 << 5
    mov cr0, eax

    cli                    ; Poista käytöstä keskeytykset
    call kmain             ; Siirry suorittamaan C-koodia

```

LISTAUS 1: Käynnistyskoodia; razos/kernel/src/boot.s

2.3 Virtuaalimuisti ja muistinhallinta

Käynnistuksen aikana prosessori on ns. *real mode*-tilassa, eli se käyttäytyy kuin 8086. Muistia ei ole käytettävissä kuin yksi megatavu, koska muistiosoitteet ovat 20-bittisiä. Käynnistyslataaja, tässä tapauksessa GNU GRUB, asettaa prosessorin 80386:n mukaiseen *protected mode*-tilaan, jossa mukaan tulee uusia ominaisuuksia, tärkeimpinä virtuaalimuisti ja sivutus (paging). Muistiosoitteet kasvavat 32-bittisiksi, jolloin suurin mahdollinen määrä muistia on neljä gigatavua. Tietokoneessa ei kuitenkaan aina ole neljää gigatavua muistia, mutta koko osoiteavaruus on silti käytettävissä *virtuaalisuutensa* takia. Fyysinen muisti on jaettu paloiksi, sivuiksi, ja virtuaaliosoitteavaruuden käytössä olevat muistialueet on ”mapattu” tiettyihin sivuihin. Virtuaalimuisti mahdollistaa sen, että prosessorin osoiteavaruuteen voidaan liittää muutakin kuin vain RAM-muistia eli käyttömuistia. Esimerkiksi laitteita, kuten näytönohjain, voidaan mapata joihinkin muistiosoitteisiin, jolloin niihin osoitteisiin kirjoitettaessa tieto kulkeekin näytönohjaimelle eikä RAM-muistiin.

Toinen virtuaalimuistin etu on se, että osoiteavaruuden kirjanpito voidaan tehdä prosessikohtaiseksi. Tällöin jokainen prosessi voi olla täysin tietämätön muista prosesseista, eikä sen tarvitse tietää mihin kohtaan *fyysistä* muistia se on ladattu, koska jokaisen prosessin osoiteavaruus alkaa nolasta ja päättyy neljään gigatavuun, osoitteeseen 0xFFFFFFFF. Muistia on myös helppompi suojella, kun käyttöjärjestelmä saa tarkistettua jokaisen luku- ja kirjoitusoperaation ja voi määrittää sivukohtaisesti mitä prosessi saa muistilla tehdä. Fyysisen muistin sivusta käytetään usein englanninkielistä nimitystä *frame*, ja virtuaalisen muistin sivusta nimitystä *page*.

Physical Memory	
00x	H E L L
01x	R L D !
02x	O W O
03x	H A V E
04x	F U N
05x	L O T
06x	S O F
07x	; -)

Process A	
Page Table	Virtual Memory
00x 00	00x H E L L
01x 02	01x O W O
02x 01	02x R L D !
03x n.a.	03x #####
04x n.a.	04x #####
05x 07	05x ; -)

Process B	
Page Table	Virtual Memory
00x 03	00x H A V E
01x 05	01x L O T
02x 06	02x S O F
03x 04	03x F U N
04x n.a.	04x #####
05x 07	05x ; -)

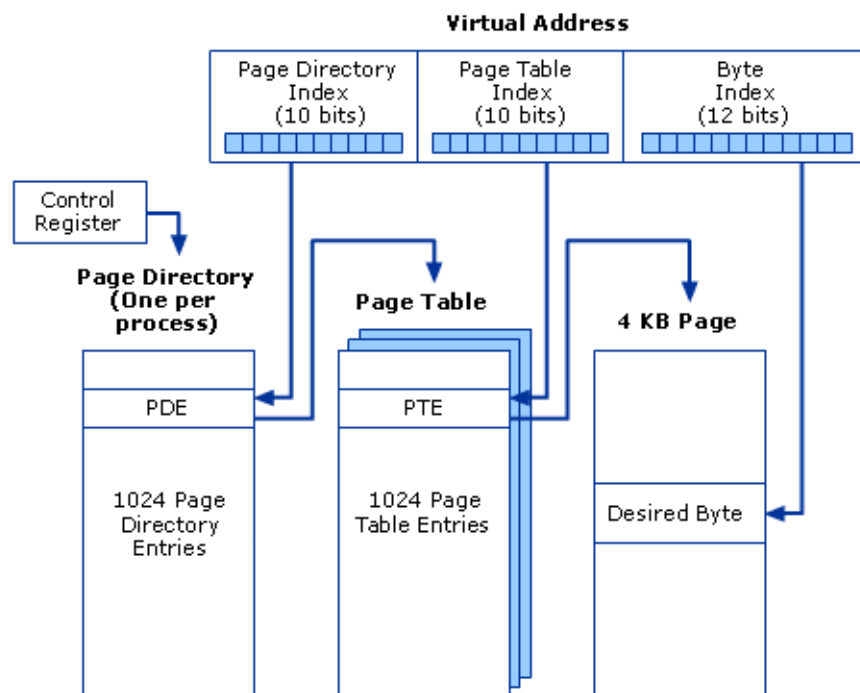
KUVA 2.1: Virtuaalimuisti havainnollistettuna

2.3.1 Virtuaalimuistin hallinta

Muistinhallinnan perustehtäviä on varata ja vapauttaa muistia ohjelmien tarpeen mukaan, mahdollisimman tehokkaasti.

Sivutus

X86-arkkitehtuuri toteuttaa virtuaalimuistin sivutuksella, kun prosessori on protected mode -tilassa. Sivutuksen kirjanpito on kolmetasoinen:



KUVA 2.2: Sivutuksen kirjanpito

Prossessorin ohjausrekisteri CR3 sisältää kulloinkin käytössä olevan *page directoryn* fyysisen muistiosoitteen. Page directory sisältää 1024 *page directory entryä* (PDE), joiden *flageilla* voi määrätä ominaisuuksia kutakin entryä

vastaavaan *page tableen*, joissa on jokaisessa puolestaan 1024 *page table entryä* (PTE). PTE sisältää yhden neljän kilotavun sivun fyysisen osoitteen ja joitakin flageja, joiden avulla määritellään, onko esimerkiksi sivulle kirjoittaminen sallittua.

Virtuaalinen muistiosoite on 32 bittiä pitkä. Sen 10 suurinta bittiä (bitit 31..21) ovat indeksi *page directoryyn*, eli kertovat mikä *page table* sisältää kyseisen sivun. Seuraavat kymmenen bittiä puolestaan ovat indeksi *page tableen*, eli kertovat mikä sivu on kyseessä. Loput 12 bittiä ovat halutun tavun etäisyys sivun alusta. RazOSissa *page directory entryä* kuvataan structilla *pg_dir_entry_t*, ja *page table entryä* structilla *page_t*.

```
struct page_t
{
    uint32_t present : 1;    /* Page present in memory */
    uint32_t rw : 1;        /* Page writable */
    uint32_t user : 1;      /* User-accessible */
    uint32_t wt_caching : 1; /* Write-through caching */
    uint32_t nocache : 1;   /* Caching disabled */
    uint32_t accessed : 1;  /* Has been accessed */
    uint32_t dirty : 1;     /* Has been written to */
    uint32_t zero : 1;      /* Always zero */
    uint32_t global : 1;    /* Global: do not flush in TLB flush */
    uint32_t avail : 3;     /* Bits not used by CPU */
    uint32_t frame : 20;    /* Physical address of the frame */
};

struct pg_dir_entry_t
{
    uint32_t present : 1;    /* Page present in memory */
    uint32_t rw : 1;        /* Page writable */
    uint32_t user : 1;      /* User-accessible */
    uint32_t wt_caching : 1; /* Write-through caching */
    uint32_t nocache : 1;   /* Caching disabled */
    uint32_t accessed : 1;  /* Has been accessed */
    uint32_t zero : 1;      /* Always zero */
    uint32_t size : 1;      /* 0 = 4K, 1 = 4M pages */
    uint32_t global : 1;    /* Global: do not flush in TLB flush */
    uint32_t avail : 3;     /* Bits not used by CPU */
    uint32_t table : 20;    /* Physical address of the page table */
};
```

LISTAUS 2: Sivutuksen tietorakenteita RazOSissa

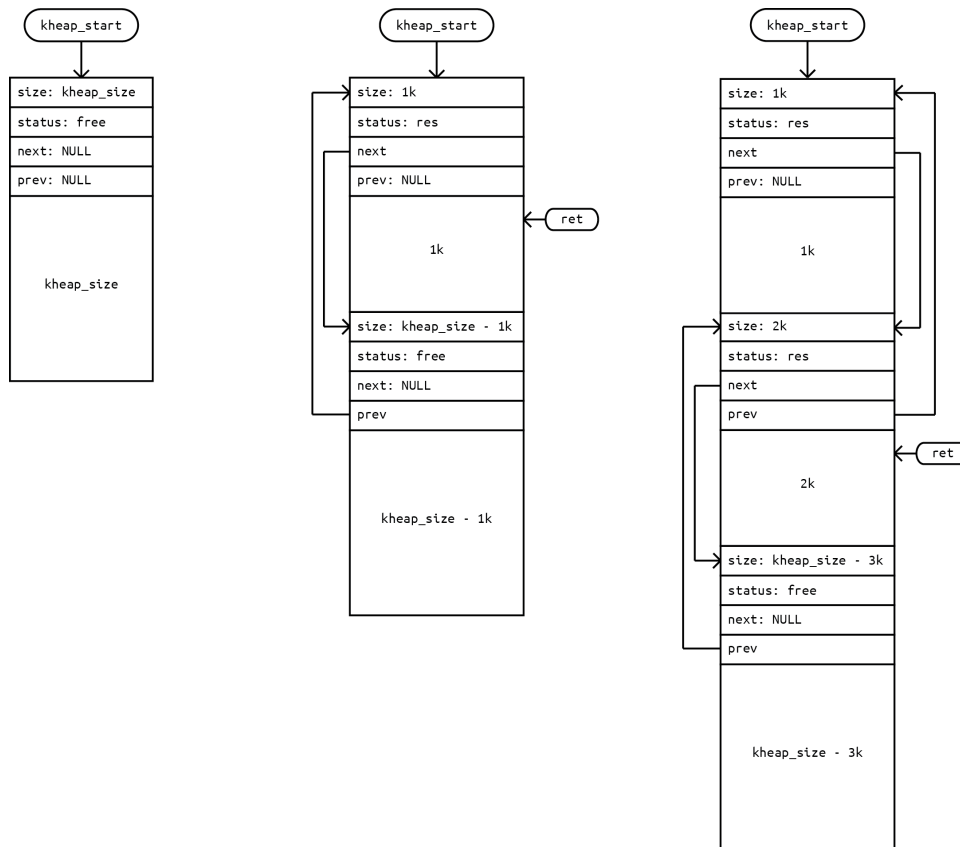
Kun tarvitaan lisää muistia, aloitetaan etsimällä vapaa fyysinen sivu. RazOSissa sen hoitaa *frame_alloc()*, joka palauttaa sivun fyysisen osoitteen. Tämän jälkeen sivu mapataan haluttuun virtuaaliosoitteeseen funktion *page_map()* avulla. Se asettaa myös halutut flagit, esimerkiksi onko sivu vain ytimen käytössä vai saako käyttäjätason ohjelmatkin käyttää sitä. Tämän jälkeen tehdään "TLB Flush", eli tyhjennetään prosessorin välimuistista kyseistä virtuaaliosoitetta vastaava fyysinen osoite, jotta prosessori laskee uuden, jolloin muistioperaatiot tapahtuvat oikeassa, uudessa kohdassa fyysistä muistia. Muistin vapauttaminen toimii samalla tavalla, ensin poistetaan mappaus kyseisestä virtuaaliosoitteesta, merkitään sivu poissaolevaksi ja lopuksi merkitään fyysinen sivu vapaaksi funktiolla *frame_free()*.

Muistin varaaminen

Kun muistia halutaan käyttää, täytyy sitä varata. Muistia käytetään pääosin kolmella tavalla: on pino (stack), dynaaminen muistialue (heap) sekä ohjelmakoodille varattu, suoritettava alue. Yksittäiset muuttujat tallennetaan yleensä pinoon, ja esimerkiksi funktiota kutsuttaessa sen parametrit ja paluusoite tallennetaan pinoon. Heappiä käytetään dynaamiseen varaukseen ja vapautukseen esimerkiksi suurten listojen kanssa. Pino on kooltaan hyvin pieni, kun heap puolestaan on todella suuri: RazOSissa pinon koko on 64 kilotavua, heapille on varattu virtuaaliosoitteita yli kolmen gigatavun edestä.

Muistin varaaminen heapiltä tapahtuu C-standardin mukaan `malloc()`-perheen funktioilla. Malloc tarvitsee tosin heap-alueen toimiakseen, ja sen käsittelyyn, laajentamiseen ja pienentämiseen, on perinteisesti kaksi funktiota; `brk()` ja `sbrk()`, jotka POSIX ennen määritteli. Funktiot `brk()` ja `sbrk()` käyttävät funktiota `page_map()` sivujen varaamiseen ja vapauttamiseen. Nykyään standardiin kuuluu niiden sijaan `mmap()`, mutta sen tarkastelu sivuutetaan. Malloc -implementaatioita on useita, joista ehkä tunnetuin on Doug Lean malloc, `dlmalloc`. `Dmalloc` on melko joustava, ja siksi sitä käytetään paljon (*A Memory Allocator*). Muitakin implementaatioita on tehty, joissakin on tarkoituksena säästää muistia ja toisissa prosessoriaikaa.

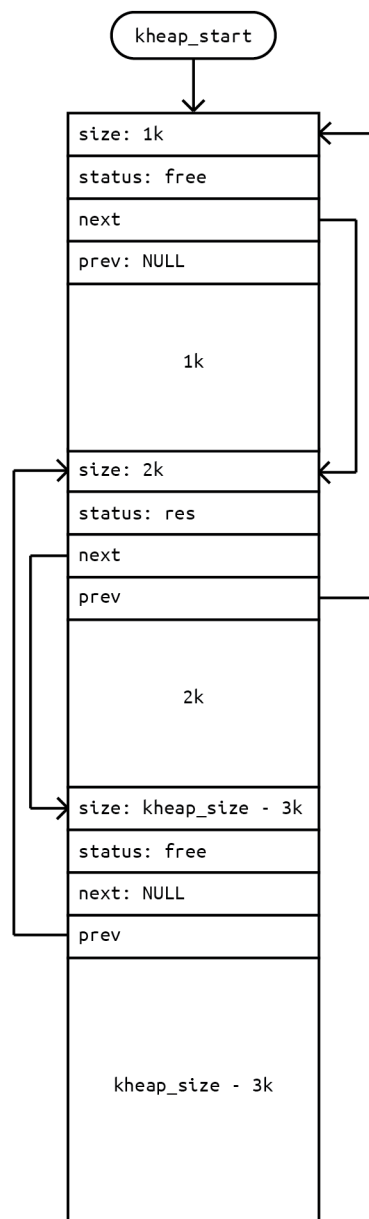
RazOSin `malloc`, `rmalloc`, lainaa joitain `dlmallocin` ajatuksia, ja on hyvin yksinkertainen. `Rmallocista` on kaksi eri versiota; toinen ytimen käyttöön ja toinen käyttäjämmailman prosessien käyttöön. Käyttäjämmailman toimintaa tarkastellaan myöhemmin. Funktioiden ero on siinä, miten varattu muisti sijoitetaan osoiteavaruuteen: ytimelle on monissa tapauksissa tärkeää, että osoitteet ovat esimerkiksi jaollisia yhden sivun koolla, neljällä kilotavulla. Käyttäjämmailmassa ei tällaista rajoitusta ole, joten käyttäjämmailman `rmalloc` on hieman yksinkertaisempi.



KUVA 2.3: Graafi `rmalloc`in toiminnasta. Alkutilanne, yksi yhden kilotavun varaus ja viimeisessä toinen, kahden kilotavun varaus.

Muistin vapauttaminen

Kun halutaan vapauttaa muistia heapiltä, käytetään perinteisesti funktiota `free()`. Se merkitsee `malloc`illa varatun muistin vapaaksi, yhdistää läheiset vapaat muistialueet (coalescing) ja mikäli mahdollista, pienetää heap-alueita `sbrk()`:n avulla.



KUVA 2.4: Kuvan 2.3 viimeisen tapauksen mukaisesta heapistä on vapautettu ensimmäinen, yhden kilotavun varaus.

2.4 Keskeytykset

Keskeytys (interrupt) on joko laitteiston tai ohjelman aikaansaama signaali, joka keskeyttää prosessorin toiminnan. Keskeytyksen tullessa suoritin tallentaa tilansa pinoon, ja siirtyy suorittamaan keskeytyskäsittelijää (interrupt handler). Kun käsittelijän suoritus on saatu päätökseen, palauttaa prosessori tilansa pinosta ja jatkaa siitä, mihin jäi keskeytyksen tullessa. Keskeytyksiä käytetään yleensä siihen, että tietokone saa tiedon erinäisistä tapahtumista, kuten näppäimistön napin painalluksista tai ajastimen (PIT,

programmable interrupt timer) laukeamisesta. Tällöin prosessorin ei tarvitse koko ajan tarkistaa, onko jotain tapahtunut, jolloin prosessoriaikaa säästyy.

X86-arkkitehtuurilla keskeytyskäsittelijöiden sijainti on tallennettu IDT-taulukkoon (Interrupt Descriptor Table). Jokainen keskeytys on numeroitu, ja IDT:stä löytyy jokaista numeroa vastaavan käsittelijän osoite, johon prosessori hyppää keskeytyksen tullessa.

2.4.1 Poikkeukset

Poikkeukset (exception) ovat keskeytyksiä, joita prosessori itse aiheuttaa. Tyypillisiä poikkeus ovat nollalla jako ja sivutusvirhe (page fault). Niiden tullessa, on prosessori yrittänyt suorittaa jotain käskyä siinä onnistumatta.

Sivutusvirhe

Sivutusvirhe tulee, kun yritetään käyttää muistia, jota ei ole mapattu tai jota ei saa käyttää. Sivutusvirheen virhekoodi kertoo, mikä meni pieleen; eikö muistia oltu mapattu, eikö muistiin kirjoittaminen ollut sallittua tai eikö käyttäjätason ohjelma saanut siihen koskea. Virheen käsittelijä voi olla hyvinkin monimutkainen, sillä se voi yrittää korjata tilannetta. Esimerkiksi Linux tukee niin sanottua *demand pagingiä*, jossa virtuaaliosoitetta ei mapata fyysiseen sivuun ennen ensimmäistä käyttöyritystä. Tällöin säästetään fyysistä muistia muiden prosessien käyttöön.

Yksinkertaisimmillaan käsittelijä vain ilmoittaa ongelmasta ja lopettaa ongelman aiheuttaneen ohjelman suorituksen. Yleensä ongelma tosin pyritään korjaamaan, mutta se ei aina ole mahdollista. Silloin UNIX-variantit lähettävät SIGSEGV -signaalin, jolla prosessi lopetetaan. RazOS ei vielä tue signaaleja, joten RazOSin sivutusvirhekäsittelijä vain lopettaa suorituksen.

2.4.2 Laitteistokeskeytykset

Laitteistokeskeytykset (Interrupt request, IRQ) aiheutuvat laitteiston toiminnasta. Esimerkiksi näppäimistön nappia painettaessa lähtee prosessorille tieto keskeytyksestä numero 33, jolloin prosessori etsii IDT:stä oikean käsittelijän ja hyppää siihen. Aluksi tallennetaan prosessorin tila, kerrotaan näppäimistölle, että keskeytys on huomattu ja jatketaan keskeytyksen käsittelyä.

```

isr_33:
    pusha                ; Prosessorin tilan tallennus pinoon

    ;; Ilmoitetaan nappaimistolle keskeytyksen huomaamisesta
    push ax
    mov al, 0x20
    out 0xa0, al
    out 0x20, al
    pop ax

    call kb_handler ; Hypataan C-koodin puolelle
    popa            ; Palautetaan prosessorin tila
    iret            ; Jatketaan suoritusta siita, mihin jaatiin

```

LISTAUS 3: Näppäimistökeskeytyksen käsittely;
razos/kernel/src/drivers/isr.s

2.5 Moniajo

Moderni käyttöjärjestelmä kykenee moniajoon, eli sillä voi käyttää useampaa ohjelmaa samaan aikaan. Myös UNIX ja sen variantit kykenevät siihen, myös RazOS. Moniytiminen prosessori kykenee suorittamaan useaa prosessia samaan aikaan, mutta tässä tarkastellaan vain yksiytimisiä prosessoreita, joissa moniajo toteutetaan jakamalla prosessoriaika usean prosessin (säikeen, thread of execution) välillä. Oiva työkalu siihen on ajastimen aiheuttamat keskeytykset; ajastimen keskeytyskäsittelijä vaihtaa suoritettavan prosessin aina ajastimen lauetessa.

2.5.1 Moniajo UNIX-varianteissa

fork-exec

UNIX ja sen variantit, mukaan lukien RazOS, hoitavat moniajon ja uuden prosessin aloituksen niin sanotulla *fork-exec* -menetelmällä. Siinä käytetään funktiota `fork()` kopiaamaan jonkin prosessin (vanhempi, parent) osoiteavaruus, jolloin syntyy toinen, samaa koodia suorittava prosessi (lapsi, child). Lapsi jatkaa suoritusta samasta kohdasta vanhempansa kanssa, mutta sillä on oma kopionsa muistista käytössä, ja se siis työskentelee omassa osoiteavaruudessaan, jolloin sillä on oma *page directory*. Tämän jälkeen voidaan kutsua jotain *exec*-perheen funktiota, RazOSSa `execve()` tai `execv()`, jolla vaihdetaan suoritettava koodi (process image), eli siirrytään suorittamaan eri ohjelmaa. Prosesseista käytetään UNIX-termiä *task*. Kun lapsi on suoritettu loppuun, kutsutaan C-kirjaston funktiota `exit()`, joka siirtää suorituksen ytimelle. Ydin vapauttaa lapsen varaaman muistin ja ilmoittaa sen vanhemmalle lapsen ”kuolleen” ja kertoo sen palautusarvon. Tämän jälkeen kaikki jäänteet lapsesta siivotaan muistista, ja palataan suorittamaan muita prosesseja.

RazOSSa jokaista prosessia kuvaa `struct task_t`, joka sisältää kaiken oleellisen tiedon prosessista:

```

struct task_t
{
    uint8_t fpu_state[512];           /* Liukulukuprosessorin tila */
    uint32_t esp;                     /* Pinon alku */
    uint32_t eip;                     /* Suoritettavan kaskyn osoite */
    struct page_dir_t* page_dir;

    struct registers_t* syscall_regs; /* Prosessorin rekisterit */

    pid_t pid;                         /* Prosessin numero */
    pid_t ppid;                       /* Vanhemman numero */

    uint32_t state;
    uint32_t exit_status;             /* Palautusarvo */

    uint32_t children;                /* Lapsien maara */

    struct task_t* wait_queue;       /* ``Kuolleiden`` lasten lista */

    void* uheap_begin;                /* Kayttajapuolen heap-alueen alku */
    void* uheap_end;

    struct fildes_t files[OPEN_MAX]; /* Kaytossa olevat tiedostot */
    int* errno_loc;                  /* errno-virhekoodin osoite */
};

```

LISTAUS 4: task_t; razos/kernel/src/mm/task.h

Funktion `fork()` implementaatioon voi tutustua tiedostojen `razos/kernel/src/mm/task.c` ja `razos/kernel/src/mm/task.s` avulla.

Ohjelman lataus

Tietokoneohjelmia tehdään monessa muodossa. UNIX-maailmassa niistä yleisin on ELF (Executable and Linkable Format). ELF-tiedosto pitää sisälleen ohjelmakoodin, etukäteen alustetut muuttujat (kuten merkkijonot) ja ohjeet käyttöjärjestelmälle siitä, miten tiedosto kuuluu ladata (virtuaali-)muistiin. Lataaja avaa tiedoston ja kopioi sen sisällön ohjeiden mukaan oikeisiin kohtiin osoiteavaruutta. Tämän jälkeen suoritus siirtyy uuden ohjelman alkuun.

Ohjelman lataukseen käytetään jotain `exec()`-funktioista. Ne antavat ohjelmalle lisäksi argumentteja ja ympäristömuuttujia (environment variable). Funktio vapauttaa käyttäjän muistialueet, sulkee avoimet tiedostot ja vaihtaa ohjelmakoodin. RazOSissa `exec()`-funktioihin kaikki funktiot perustuvat funktioon `execve()`. Sen lähdekoodi löytyy kansioista `razos/kernel/src/loader/`.

Prosessin vaihto

Ajastimen lauetessa kutsutaan vuorontajaa (scheduler), joka valitsee seuraavan suoritettavan prosessin. Aluksi vuorontaja tallentaa vanhan prosessin tilan, RazOSissa `task_t`:hen, jotta sen suoritukseen voidaan

myöhemmin palata ongelmista. Sen jälkeen valitaan uusi prosessi funktiolla `sched_next()`, vaihdetaan osoiteavaruus eli kirjoitetaan CR3-ohjausrekisteriin uuden prosessin page directoryn osoite ja lopuksi palautetaan uuden prosessin tila `task_t`:stä. Tämän jälkeen vuorontajan suoritus loppuu ja hypätään `task_t`:stä otettuun suoritussosiitteeseen (rekisteri EIP, [Extended] Instruction Pointer), josta palataan usein käyttäjämaailmaan ja siellä jatketaan prosessin suoritusta seuraavaan ajastimen laukeamiseen asti.

```

sched_switch:
    ;; Tallenna vanhan prosessin tila
    pusha
    mov eax, [cur_task]
    fxsave task_fpu_state(eax)
    mov task_esp(eax), esp
    mov task_eip(eax), dword .return ; Osoite, josta suoritus jatkuu

    ;; Valitse uusi prosessi
    call sched_next
    mov [cur_task], eax

    ;; Hae page directoryn fyysinen osoite
    push dword task_page_dir(eax)
    call get_page_dir_phys
    add esp, 4

    ;; Lataa se CR3:een
    mov cr3, eax

    mov eax, [cur_task]

    ;; Palauta prosessin tila
    fxrstor task_fpu_state(eax)
    mov esp, task_esp(eax)
    jmp task_eip(eax)

.return:
    popa
    ret

```

LISTAUS 5: Vuorontajan assemblyllä kirjoitettu osa;
razos/kernel/src/mm/task.s

Vuoronnusalgoritmeja on useita. Yleensä prosesseille voidaan määrittää prioriteetti, jonka mukaan määrätään mikä prosessi saa eniten suoritusaikaa. Yksinkertaisissa algoritmeissa ei priorisointia ole. Esimerkiksi RazOS käyttää nk. kiertovuorottelumenetelmää (round robin scheduling), jossa jokainen prosessi saa käyttää aikaa korkeintaan jonkin tietyn aikaviipaleen (time slice) verran. Tämän jälkeen prosessia vaihdetaan ja ajossa ollut prosessi laitetaan jonon viimeiseksi. Menetelmä on *irrottava*, koska ydin voi määrätä milloin prosessin aikaviipale loppuu ja on vaihdettava prosessia. Irrottavan vuoronnuksen vastakohta on ei-irrottava vuoronnus, jossa prosessilta ei voida viedä vuoroa pois, vaan se luovuttaa sen itse. Sekin on usein mahdollista irrottavaa vuoronnusta käyttävissä järjestelmissä. Esimerkiksi RazOSissa prosessi antaa suorituvuoronsa pois, jos se joutuu odottamaan esimerkiksi lukuoperaation valmistumista. Siihen käytetään funktiota `sched_yield()`.

2.6 Tiedostojen hallinta

Yksi käyttöjärjestelmän tärkeistä osista on tiedostojen hallitseminen. Se on laaja kokonaisuus, jossa nousee abstraktiotasolta toiselle, kunnes C-kirjaston yleiset menetit (tiedostossa `stdio.h`) ovat käytettävissä. POSIX tuo omia keinojaan, jotka ovat abstraktioltaan paljon alempana C-kirjastoa. Abstraktion tarkoituksena on, että ohjelmoija voi samalla tavalla käyttää mitä tahansa tiedostoksi ajateltavaa kokonaisuutta: kiintolevyllä olevaa tiedostoa, sarjaporttia tai vaikkapa USB-väylää. Abstraktiota kutsutaan virtuaalitiedostojärjestelmäksi (Virtual File System, VFS). RazOSissa VFS kuvaa tiedostoja ja niihin verrattavia kokonaisuuksia structeilla `vfs_node_t` ja `stat` ja käsittelee avoimia tiedostoja structin `fildes_t` avulla.

```
struct vfs_node_t
{
    char name[64];          /* Tiedoston nimi */
    struct stat status;

    /* Kyseisen tiedostojärjestelman käsittelyfunktiot,
     * kuuluvat laitteen/tiedostojärjestelman ajuriin */
    read_t read;
    write_t write;
    open_t open;
    creat_t creat;
    close_t close;
    lseek_t lseek;

    /* VFS on linkitetty lista, osoitin seuraavaan alkioon */
    struct vfs_node_t* next;
};

struct fildes_t
{
    struct vfs_node_t* vfs_node;
    off_t at;              /* Kertoo missä kohdassa tiedostoa ollaan */
    uint32_t oflag;        /* Tiedoston avausflagit */
};

LISTAUS      6:      vfs_node_t      ja      fildes_t;
              razos/kernel/src/fs/vfs.h
```

Structi `stat` löytyy tiedostosta `razos/razos_kernel_headers/sys/stat.h`. Se pitää sisällään tiedostoa koskevaa tietoa, kuten millä laitteella se sijaitsee (vai onko se laite itsessään), mitä sille saa tehdä, ja koska sitä on viimeksi muokattu.

Kun jotain tiedostoa halutaan käyttää, tulee se ensin avata funktiolla `open()`. Se palauttaa numeron, file descriptorin (`fd`), joka viittaa johonkin prosessin `task_t`:ssä olevista `fildes_t`-structeista. Se pitää sisällään avausflagit ja pitää kirjaa missä kohdassa tiedostoa ollaan. Siinä on myös osoitin tiedostoa vastaavaan `vfs_node_t`-structiin, jonka kautta löydetään sopivat funktiot tiedoston käyttöön. Funktioista löytyy tietoa standardeista (POSIX.1-2008) ja (ISO/IEC 9899:1999) ja niiden implementaatiot löytyvät kansioista `razos/kernel/src/fs/`.

Tiedostoja on monenlaisia. Tavallisten tiedostojen ja laitteiden lisäksi UNIX käsittelee putkia (pipe), joiden kautta prosessit voivat lähettää toisilleen tietoa. Putki on sisäisesti jono (FIFO, first-in, first-out), jonka päät ovat eri prosesseissa. Lisäksi on näytölle tulostamista varten `stdout` ja `stderr` ja näppäimistöltä lukemista varten `stdin`. Niitä voidaan korvata putkillla, jolloin yksi prosessi voi lähettää tavallisesti näytölle tulostettavan tekstin toiselle prosessille, joka käsittelee sen kuin se olisi tullut näppäimistöltä.

2.7 Järjestelmäkutsut

Järjestelmäkutsut eli system callit (syscall) mahdollistavat käyttäjätason ohjelmien pääsyn ytimen resursseihin turvallisesti. X86-alustalla järjestelmäkutsuja voidaan toteuttaa usealla tavalla, joista yleisimmät ovat keskeytysten käyttö tai `sysenter` ja `sysexit` -käskyjen käyttö. Keskeytyksien käyttö on usein hitaampaa, ja RazOS käyttääkin jälkimmäistä keinoa. Suorittaessaan `sysenter`-komennon, siirtyy prosessori lupatasoista (current privilege level, CPL) suurimmalle, tasolle nolla. Käyttäjämääailman koodia ajetaan tasolla kolme. Tasojen erot ovat siinä, että 0-tasolla on käytettävissä joitain käskyjä, jotka saattavat saada vahinkoa aikaan, jos esimerkiksi haittaohjelma pystyisi niitä suorittamaan. Toinen ero on se, että sivutuksen avulla voidaan määritellä, mitä muistia 3-tason koodi saa käyttää. Suoritustason muutoksen lisäksi prosessori siirtyy suorittamaan koodia, jonka osoite on tallennettu MSR-rekisteriin. Sieltä suoritus etenee haluttuun järjestelmäkutsuun, ja lopuksi palataan käyttäjämääailmaan, tasolle 3, käskyllä `sysexit`. Käskyjen toimintaan ja lupatasoihin voi tutustua tarkemmin Intelin manuaalin avulla (*Intel® 64 and IA-32 Architectures Software Developer's Manual*).

Järjestelmäkutsut on numeroitu, ja niiden numerot löytyvät RazOSissa tiedostosta `razos/razos_kernel_headers/api/razos.h`. Kun käyttäjätason koodissa halutaan käyttää jotain järjestelmäkutsua, käytetään siihen tiedostosta `razos/rlibc/arch/i386/crt0.s` löytyviä `__syscall` -funktioita. Niille annetaan argumenteiksi järjestelmäkutsun numero ja mahdolliset muut parametrit, maksimissaan kolme.

```

    ;; Parametrit ja syscall-numero System V ABI:n mukaisesti pinossa
    ;; razos.h:ssa uint32_t __syscall3(num, arg1, arg2, arg3)
__syscall3:
    ;; Tallennetaan callee-save rekisterit (System V ABI)
    push ebx
    push edi
    push esi

    ;; Parametrit ja syscall-numero pinosta rekistereihin
    mov esi, [esp+12+16]
    mov edi, [esp+12+12]
    mov ebx, [esp+12+8]
    mov eax, [esp+12+4]

    ;; sysexit vaatii käyttäjämääntä pinoon (esp)
    ;; ja osoitteen, josta suoritusta jatketaan kutsun
    ;; jälkeen rekistereihin ecx ja edx
    push ecx
    push edx
    mov ecx, esp
    mov edx, .ret
    sysenter

    ;; Kutsusta palataan tahaan sysexitin avulla
.ret:
    ;; Palautetaan rekisterien arvot pinosta
    pop edx
    pop ecx
    pop esi
    pop edi
    pop ebx

    ;; Palataan sinne, mistä syscallia kutsuttiin
    ret

```

LISTAUS 7: __syscall -funktioista yksi, kolme argumenttia ottava

Ylläolevasta koodista hypätään sysenterillä ytimeen, funktioon `syscall_entry`, tiedostossa `razos/kernel/src/syscall/syscall.s`. Siellä alustetaan kernelin pino ja kutsutaan C-koodilla kirjoitettu jatkokäsittelijä, `syscall_dispatch()`. Se etsii `syscall_table` -taulukosta järjestelmäkutsun numeroa vastaavan funktion, ja suorittaa sen. Lopuksi siivotaan pino ja suoritetaan `sysexit` -komento, jonka myötä palataan käyttäjämääntä.

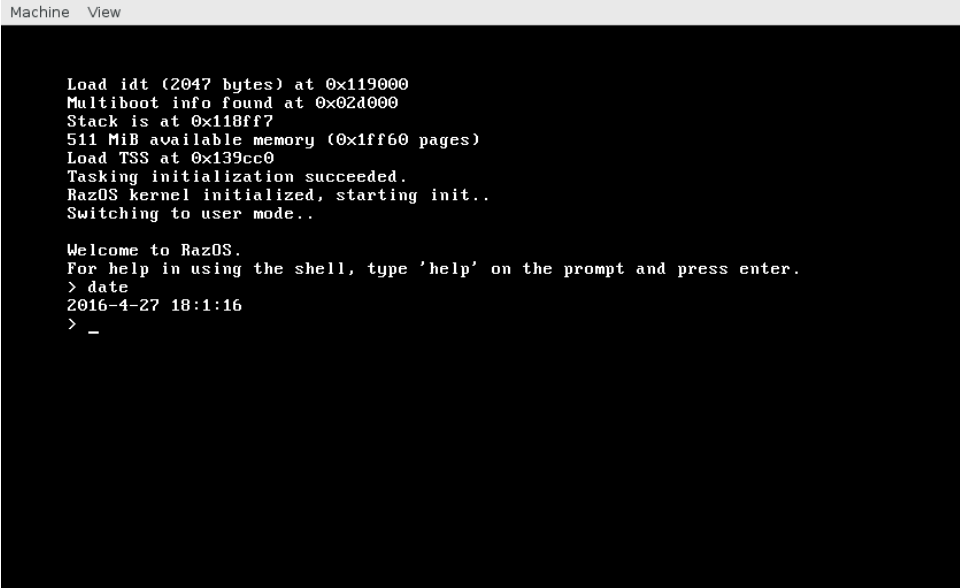
Järjestelmäkutsujen avulla voidaan kirjoittaa C-kirjaston ne osat, jotka tarvitsevat jotain kernelin resursseja, kuten tiedostoja. Ilman niitä olisi kirjaston teko hyvin vaikeaa, ja ohjelmien pitäisi olla osa ydintä. Se veisi mennessään huomattavan paljon joustavuutta ja toisi mukanaan suuria tietoturvaongelmia, koska kaikki koodi voisi käyttää mitä vain ytimen osaa hyväkseen, ja kaikki koodi suoritettaisiin lupatasolla nolla.

3 Käyttäjätila

Unix-käyttöjärjestelmissä virtuaalimuisti jaetaan ydintilan ja käyttäjätilan välillä. Käyttäjätilan määrittelee sen sisältämät ohjelmointikirjastot sekä kernelin tarjoamat järjestelmäkutsut. Näiden komponenttien päälle rakentuvat käyttöjärjestelmän käyttöä hallitsevat ohjelmat sekä näitä ohjelmia tukevat kirjastot.

RazOS:in kehitystyössä on kiinnitetty erityistä huomiota oleellisiin standardeihin. Toteutetuilta osin ohjelmointikirjastot noudattavatkin C99- ja POSIX- standardien vaatimuksia. Tavoitteena on ollut helpottaa kehitystyön jatkamista, ja toisaalta mahdollistaa ulkopuolisten standardinmukaisien ohjelmien käyttö osana RazOS:ia.

RazOS:iin kuuluu myös joitakin käyttäjäohjelmia, tärkeimpänä komentotulkki (shell), Rash. Vaikka ohjelmat eivät toistaiseksi muodosta kovinkaan käytännöllistä kokonaisuutta, osoittavat ne käyttöjärjestelmän standardinmukaisuuden: esimerkiksi komentotulkki Rash toimii niin RazOS:issa kuin Linuxissakin.



```
Machine View

Load idt (2047 bytes) at 0x119000
Multiboot info found at 0x02d000
Stack is at 0x118ff7
511 MiB available memory (0x1ff60 pages)
Load TSS at 0x139cc0
Tasking initialization succeeded.
RazOS kernel initialized, starting init..
Switching to user mode..

Welcome to RazOS.
For help in using the shell, type 'help' on the prompt and press enter.
> date
2016-4-27 18:1:16
> _
```

3.1 Standardityökalut (stdlib.h)

C-kirjasto `stdlib.h` määrittelee erilaisia yleishyödyllisiä funktioita, tyyppejä ja makroja, tärkeimpinä dynaamiseen muistinvaraukseen käytettävä `malloc`, ympäristömuuttujan hallitsemiseen käytettävät funktiot, `getenv` ja `setenv`, sekä satunnaislukujen luomiseen käytettävä `rand()`.

3.1.1 Dynaaminen muistinvaraus (malloc)

Dynaaminen muistinvaraus toimii RazOS:issa hyvin samalla tavoin niin ytimessä kuin käyttäjätilassakin. Käyttäjätilassa jokaisella ohjelmalla on kuitenkin oma virtuaalimuistiavaruus, jossa muistia hallitaan. Koska käyttäjätilalla ei myöskään ole oikeutta käyttää järjestelmäresursseja suoraan, varataan varsinaiset muistisivut `brk` ja `sbrk` -järjestelmäkutsujen avulla.

3.1.2 Ympäristömuuttuja (environ)

Ympäristömuuttuja (environment variable, `environ`) on osoitin, joka osoittaa merkkijonoista muodostuvaan vektoriin. Nämä merkkijonot sisältävät tietoa ohjelman ympäristöstä, kuten esimerkiksi käytössä olevasta komentotulkista. Kun prosessi kopioidaan `fork`-järjestelmäkutsulla tai ohjelma aloitetaan `exec`-järjestelmäkutsulla, ympäristömuuttujan arvot säilyvät.

Koska `environ`-muuttujan merkkijonojen manuaalinen muuttaminen on virhealtista, POSIX-standardiin kuuluu sen muuttamiseen ja noutamiseen käytettävät `setenv`- ja `getenv`-funktiot. Ne on toteutettu myös RazOS:issa.

3.1.3 Satunnaisluvut (rand)

C-kirjaston `rand`-funktio on näennäissatunnaislukugeneraattori, joka pyrkii matemaattisilla operaatioilla luomaan annetusta siemenarvosta mahdollisimman satunnaisen luvun. RazOS:in `rand`-funktio täyttää yksinkertaisuudestaan huolimatta C99-standardin minimivaatimukset, ja perustuu kin standardin ehdotukseen.

```
#include <stdlib.h>

static unsigned long next = 1; /* For the rand implementation. */

int rand(void)
{
    next = next * 1103515245 + 12345;
    return ((unsigned) (next/65536) % 32768);
}

void srand(unsigned int seed)
{
    next = seed;
}
```

LISTAUS 8: `stdlib.h`

3.2 Syöte ja tulostus (stdio.h)

Tärkeimmät syötteeseen ja tulostukseen liittyvät funktiot löytyvät C-kirjastossa `stdio`-ylätunnisteen alta. Kirjaston toiminta perustuu täysin käyttöjärjestelmäytimen `read`- ja `write`-järjestelmäkutsuihin, sillä vain ydin

voi hallita tietokoneen laitteistoja, kuten näyttöä ja näppäimistöä. Käyttäjätilan ohjelmat lukevat ja tulostavat muiden mekanismien avulla, joita Unixissa ovat tekstivirrat ja tiedostodeskriptorit. Jokaiseen tiedostodeskriptoriin liittyy tekstivirta, ja toisaalta jokaista tekstivirtaa vastaa tiedostodeskriptori. Myös standardisyöte (stdin) ja -tuloste (stdout) ovat tiedostodeskriptoreja, joten niitä voidaan käsitellä kuten muitakin tiedostoja. Järjestelmän käsittäminen tiedostojen avulla onkin yksi Unixin periaatteellisista kulmakivistä.

Toinen tärkeä osa Unixin syöte- ja tulostustoimintoja on tekstivirtojen uudelleenohjaus (redirection): ohjelman tuloste voidaan ohjata johonkin tiedostoon tai toisen ohjelman syötteeksi. Samaten syöte voi tulla näppäimistöltä, toiselta ohjelmalta tai tiedostosta. Ohjelman itse ei tarvitse kuin lukea stdin-tiedostodeskriptoria, sillä järjestelmä hoitaa tekstivirtojen uudelleenohjauksen. Tämä monipuolisuus on syy siihen, miksi Unix käyttää tekstivirtoja yleisenä rajapintana ohjelmien, tiedostojen ja käyttäjien välillä.

RazOS ei syöte- tai tulostustoiminnoiltaan eroa muista POSIX-yhteensopivista Unix-käyttöjärjestelmistä. Käytännön tasolla tekstivirtojen uudelleenohjaus toteutetaan pipe-järjestelmäkutsulla, ja tiedostodeskriptorien luonti vastaavasti dup- ja dup2-järjestelmäkutsuilla. Itse stdio-kirjasto vastaa myös toteutetuilta osin POSIX-standardia, sisältäen esimerkiksi printf-funktion eri muunnelmia.

3.3 Komentotulkki

Komentotulkki on yksi tärkeimmistä ohjelmista Unix-käyttöjärjestelmässä. Se vastaa ohjelmien käynnistämisestä ja prosessinhallinnasta, tekstivirtojen uudelleenohjaamisesta sekä automaatiosta. RazOS:in komentotulkki, Rash, tukee Shell-ohjelmointia lukuun ottamatta näitä komentotulkin perustoimintoja. Erikoisasemastaan huolimatta komentotulkki on Unixissa, kuten RazOS:issakin, vain ohjelma muiden joukossa, eikä sen tiiviimpi osa käyttöjärjestelmää kuin mikään muukaan ohjelma.

Korkealla tasolla komentotulkin toimintaperiaate on varsin yksinkertainen, josta osoituksena alla oleva Rashin syöte-tuloste -silmukka:

```

static void rash_loop(void)
{
    char *line;
    char **args;
    int status;
    int args_len;
    int i;

    do
    {
        printf("> ");
        line = rash_read_line();
        args = rash_split_line(line, &args_len);
        status = rash_execute(args);

        free(line);
        for (i = 0; i < args_len; i++)
            free(args[i]);
        free(args);
    } while (status);
}

```

LISTAUS 9: Rashin silmukka

Komentotulkin kehitystyö tehtiin GNU/Linux-ympäristössä, josta se siirrettiin pienin muutoksin osaksi RazOS:ia.

3.4 Esimerkki standardinmukaisuudesta

Alla on pieni C-ohjelma, joka tulostaa näytölle päivämäärän ja kellonajan.

```

#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t t = time(NULL);
    struct tm tm_now = *gmtime(&t);

    printf("%d-%d-%d %d:%d:%d\n",
           tm_now.tm_year + 1900,
           tm_now.tm_mon + 1,
           tm_now.tm_mday,
           tm_now.tm_hour,
           tm_now.tm_min,
           tm_now.tm_sec);

    return 0;
}

```

LISTAUS 10: Esimerkkiohjelma; date.c

Ohjelma toimii ilman muutoksia niin RazOS:issa, muissa Unixeissa kuin kaikissa C99-standardin ohjelmointikirjastot sisältävissä käyttöjärjestelmissä.

4 Yhteenveto

RazOS:in käytettävyydessä on vielä parannettavaa. Suurin puute on kunnon päätteen eli terminaaliemulaattorin puuttuminen. Tämä tekee näyttötulosteen tarkasta hallinnasta käyttäjätilassa vaikeaa. Pääteohjelma on kuitenkin jo olemassa, ja se tullaan käyttöjärjestelmän kehitystyön jatkuessa pian lisäämään osaksi järjestelmää.

Unix tunnetaan ehkä parhaiten komentorivityökaluistaan, kuten komennoista `grep`, `awk` ja `cat`. RazOS:issa ei toistaiseksi ole näitä Unixin tunto-merkkejä, mutta niiden lisääminen osaksi järjestelmää on jo nyt täysin mahdollista. Lisäksi mikä tahansa POSIX-yhteensopiva toteutus näistä ja muista ohjelmista voi suoraan toimia myös RazOS:issa.

Lähdekoodi kokonaisuudessaan löytyy GitHubista, osoitteesta <https://github.com/Razbit/razos>. Sieltä löytyy myös ohjeet RazOSin "asentamiseen" ja käyttämiseen, tiedostosta `README.md`. Oikealla tietokoneella käyttämistä emme suosittele. Jos tiedoston `floppy.img` laittaa johonkin emulaattoriin, kuten Qemuun, levykkeeksi, voi käyttöjärjestelmää kokeilla.

4.1 Ajankäyttö ja työnjako

Projekti alkoi jo syksyllä 2014. Sen jälkeen on ollut kaksi täydellistä uudelleenkirjoitusta, ja nykyinen versio sai alkunsa syksyllä 2015. Talven aikana muistinhallinta meni kokonaan uusiksi. Aikaa on siis mennyt todella paljon. GitHubin statistiikat kertovat, että koodia on noin 18000 riviä.

Iiro tuli projektiin mukaan vasta alkusyksystä 2015. Silloin oli kernelin toinen uudelleenkirjoitus alkamassa, ja tehtiin sellainen työnjako, että Iiro tekee käyttäjäpuolta ja Eetu kerneliä. Toki vähän meni ristiin välillä, mutta se oli ajatus ja siinä on melko hyvin pysytty.

4.2 Mitä opimme

Jo ennen projektia meillä oli molemmilla suhteellisen kattava ohjelmointitaitausta; C oli jo tuttu kieli ja monia ohjelmointiprojekteja oli takana. Tämän kaltainen projekti toi ohjelmointiin aivan uudenlaisen näkökulman: ennen saattoi luottaa siihen, että kirjasto tarjosi kaikenlaisia valmiita funktioita ja tietotyyppejä, mutta käyttöjärjestelmää tehdessä ne on määriteltävä itse. Assymbyllä ohjelmointi oli vierasta, mutta se tuli hyvin tutuksi projektin aikana. Binäärimuotoon käännettyjen ohjelmien rakenteesta saimme valtavan määrän uutta tietoa, kuten myös laitteiston ja ohjelmiston toimimisesta yhdessä. Laitteiston läheinen ohjelmointi oli uutta siinä mittakavassa, jossa sitä piti käyttöjärjestelmän alimpia osia tehdessä harjoittaa.

Lopputulos, RazOSin toiminnallisuus kirjoitushetkellä, yllätti tekijänsäkin; se toimii yllättävän hyvin, vaikka monesti on meinannut usko loppua. Tekemistä vielä toki on, että pääsisimme tavoitteeseemme: käyttöjärjestelmään, jolla voisi jatkaa sen itsensä kehitystä.

Mitä tekisimme toisin? Montakin asiaa. RazOS on pyritty pitämään mahdollisimman yksinkertaisena, eikä liialliseen optimointiin ole ryhdytty. Se näkyy esimerkiksi siinä, että muistinhallinta on melko hidas, ja heap-alueet fragmentoituvat helposti. Nämä asiat tosin ovat omalla tavallaan projektin tarkoituksen ulkopuolella; nykyiset algoritmit selittävät verrattain yksinkertaisesti käyttöjärjestelmän toimintaa. Ehkä suurin muutos, mikä tulisi tehdä, on virtuaalisen tiedostojärjestelmän toteutus siten, että se tukisi kansioita.

A RazOSin järjestelmäkutsut

RazOSissa on kirjoitushetkellä 18 järjestelmäkutsua. Ne on määritelty tiedostossa `razos/razos_kernel_headers/api/razos.h`.

sys_exit

Numero: 0

Kutsuminen: `void exit(int status)`

Implementaatio: `razos/kernel/src/syscall/sys_tasking.c`

Kuvaus: lopettaa prosessin

sys_sched_yield

Numero: 1

Kutsuminen: `void sched_yield(void)`

Implementaatio: `razos/kernel/src/syscall/sys_tasking.c`

Kuvaus: luovuttaa suoritusvuoron

sys_fork

Numero: 2

Kutsuminen: `pid_t fork(void)`

Implementaatio: `razos/kernel/src/syscall/sys_tasking.c`

Kuvaus: jakaa prosessin kahteen samanlaiseen

sys_wait

Numero: 3

Kutsuminen: `uint32_t wait(int* stat_loc)`

Implementaatio: `razos/kernel/src/syscall/sys_tasking.c`

Kuvaus: asettaa vanhemman odottamaan lapsiprosessin valmistumista

sys_read

Numero: 4

Kutsuminen: `ssize_t read(int fd, void* buf, size_t size)`

Implementaatio: `razos/kernel/src/syscall/sys_fs.c`

Kuvaus: lukee tiedostosta

sys_write

Numero: 5

Kutsuminen: `ssize_t write(int fd, const void* buf, size_t size)`

Implementaatio: `razos/kernel/src/syscall/sys_fs.c`

Kuvaus: kirjoittaa tiedostoon

sys_open

Numero: 6

Kutsuminen: `int open(const char* name, int oflag, ...)`

Implementaatio: `razos/kernel/src/syscall/sys_fs.c`

Kuvaus: avaa tiedoston

sys_close

Numero: 7

Kutsuminen: `int close(int fd)`

Implementaatio: razos/kernel/src/syscall/sys_fs.c

Kuvaus: sulkee avoimen tiedoston

sys_creat

Numero: 8

Kutsuminen: `int creat(const char* name, mode_t mode)`

Implementaatio: razos/kernel/src/syscall/sys_fs.c

Kuvaus: luo uuden tiedoston

sys_lseek

Numero: 9

Kutsuminen: `off_t lseek(int fd, off_t offset, int whence)`

Implementaatio: razos/kernel/src/syscall/sys_fs.c

Kuvaus: siirtyy tiedostossa *offset* tavua

sys_fcntl

Numero: 10

Kutsuminen: `int fcntl(int fd, int cmd, ...)`

Implementaatio: razos/kernel/src/syscall/sys_fs.c

Kuvaus: hakee tai asettaa monia tiedoston asetuksia/tietoja

sys_fstat

Numero: 11

Kutsuminen: `int fstat(int fd, struct stat* buf)`

Implementaatio: razos/kernel/src/syscall/sys_fs.c

Kuvaus: hakee tiedoston status-tietoja

sys_setup

Numero: 12

Kutsuminen:

Implementaatio: razos/kernel/src/syscall/sys_setup.c

Kuvaus: asettaa tai hakee kernelin asetuksia

sys_pipe

Numero: 13

Kutsuminen: `int pipe(int fd[2])`

Implementaatio: razos/kernel/src/syscall/sys_pipe.c

Kuvaus: luo pipen

sys_brk

Numero: 14

Kutsuminen: `int brk(void* addr)`

Implementaatio: razos/kernel/src/syscall/sys_uvm.c

Kuvaus: asettaa käyttäjän heap-alueen lopun osoitteeseen *addr*

sys_sbrk

Numero: 15

Kutsuminen: `void* sbrk(intptr_t incr)`

Implementaatio: razos/kernel/src/syscall/sys_uvm.c

Kuvaus: kasvattaa tai pienetää käyttäjän heap-aluetta *incr* tavun verran

sys_execve

Numero: 16

Kutsuminen: `int execve(const char* path, char* const argv[], char* const envp[])`

Implementaatio: razos/kernel/src/syscall/sys_execve.c

Kuvaus: vaihtaa suoritettavan ohjelman

sys_time

Numero: 17

Kutsuminen: time_t time(time_t* timer)

Implementaatio: razos/kernel/src/syscall/sys_time.c

Kuvaus: palauttaa ajan sekunteina UNIXin Epochista, 1.1.1970 UTC00:00:00

Kirjallisuus ja lähteet

- A Memory Allocator*. Doug Lea. 2000. URL: <http://g.oswego.edu/dl/html/malloc.html>.
- Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. 2016. URL: <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>.
- ISO/IEC 9899:1999*. International Organization for Standardization, International Electrotechnical Commission. 2005. URL: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>.
- LevOS*. levex. 2016. URL: <https://github.com/levex/osdev>.
- OSDev, Wiki. 8259 PIC*. 2016. URL: http://wiki.osdev.org/8259_PIC.
- *Context Switching*. 2016. URL: http://wiki.osdev.org/Context_Switching.
 - *Creating a C library*. 2016. URL: http://wiki.osdev.org/Creating_a_C_Library.
 - *ELF*. 2016. URL: <http://wiki.osdev.org/ELF>.
 - *Exceptions*. 2016. URL: <http://wiki.osdev.org/Exceptions>.
 - *Interrupt Descriptor Table*. 2016. URL: http://wiki.osdev.org/Interrupt_Descriptor_Table.
 - *Interrupt Service Routines*. 2016. URL: <http://wiki.osdev.org/ISR>.
 - *Interrupts*. 2016. URL: <http://wiki.osdev.org/Interrupts>.
 - *Memory management*. 2016. URL: http://wiki.osdev.org/Memory_management.
 - *Page Fault*. 2016. URL: http://wiki.osdev.org/Page_fault.
 - *Paging*. 2016. URL: <http://wiki.osdev.org/Paging>.
 - *Processes and threads*. 2016. URL: http://wiki.osdev.org/Processes_and_Threads.
 - *Programmable Interval Timer*. 2016. URL: http://wiki.osdev.org/Programmable_Interval_Timer.
 - *Protected Mode*. 2016. URL: http://wiki.osdev.org/Protected_mode.
 - *PS2 Keyboard*. 2016. URL: http://wiki.osdev.org/PS2_Keyboard.
 - *Scheduling algorithms*. 2016. URL: http://wiki.osdev.org/Scheduling_Algorithms.
 - *SYSENTER*. 2016. URL: <http://wiki.osdev.org/Sysenter>.
 - *System calls*. 2016. URL: http://wiki.osdev.org/System_Calls.
 - *System initialization (x86)*. 2016. URL: [http://wiki.osdev.org/System_Initialization_\(x86\)](http://wiki.osdev.org/System_Initialization_(x86)).
 - *System V ABI*. 2016. URL: http://wiki.osdev.org/System_V_ABI.
 - *Task state segment*. 2016. URL: http://wiki.osdev.org/Task_State_Segment.

- POSIX.1-2008*. IEEE Std 1003.1-2008 and The Open Group Technical Standard Base Specifications, Issue 7. The Open Group, IEEE Computer Society. 2013. URL: <http://pubs.opengroup.org/onlinepubs/9699919799/>.
- Radium*. Charlie Somerville. 2016. URL: <https://github.com/charliesome/radium>.
- sv6 Operating System*. aclements. 2016. URL: <https://github.com/aclements/sv6>.
- The Basekernel Operating System*. dthain. 2016. URL: <https://github.com/dthain/basekernel>.
- Wikipedia. *Memory management*. 2016. URL: https://en.wikipedia.org/wiki/Memory_management.
- *Paging*. 2016. URL: <https://en.wikipedia.org/wiki/Paging>.
 - *Virtual memory*. 2016. URL: https://en.wikipedia.org/wiki/Virtual_memory.
- xv6 Operating System*. guilleiguaran. 2016. URL: <https://github.com/guilleiguaran/xv6>.