Posix-säikeet

Saku Hujala

LuK-tutkielma 3.3.2004

Jyväskylän Yliopisto
Tietotekniikan laitos

Tekijä: Saku Hujala

Yhteystiedot: sphujala@cc.jyu.fi

Työn nimi: Posix-säikeet

Title in english: Posix Threads

Työ: LuK-tutkielma

Linja: Ohjelmistotekniikka

Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos

Sivuja: 18

Avainsanat: monisäikeinen ohjelmointi, Posix, rinnakkainen suoritus

Tiivistelmä: Tutkielmassa luodaan katsaus monisäikeseen ohjelmointiin, erityisesti Posix-standardin mukaisia säikeitä käyttäen. Säikeiden avulla voidaan ohjelmoida rinnakkain suoritettavia tehtäviä ohjelmiin.

Keywords: multithreaded programming, Posix, parallel computing

Summary: Study provides a compact overview of multithreaded programming, especially using Posix-threads. Threads allow for programming parallel tasks in programs.

Sisältö

1 Johdanto

LuK-tutkielma käsittelee ohjelmoinnissa käytettäviä säikeitä keskittyen erityisesti Posix-standardin mukaisiin säikeisiin. Posix määrittelee useita Unix-järjestelmiin liittyviä standardeja. Tutkielmassa kuvataan säikeiden toimintaa ja käyttöä ohjelmoinnissa. Tutkielma ei kuitenkaan esitä yksityiskohtaisesti säikeiden toteutuksen sisäistä toimintaa.

Suurissa ohjelmistoissa esiintyy usein tilanteita, joissa rinnakkaisella suorituksella voidaan saavuttaa etuja tavanomaiseen perättäiseen suoritukseen nähden. Näitä tilanteita voi syntyä esimerkiksi asiakas-palvelinjärjestelmissä tai suurissa laskutoimituksissa. Monisäikeinen ohjelmointi antaa keinon, jolla rinnakkainen suoritus voidaan toteuttaa.

Luvussa 2 tarkastellaan yleisesti säikeitä ja niihin liittyvää terminologiaa. Lisäksi luvussa luodaan katsaus säikeiden synkronointiin. Luku 3 esittelee tarkemmin säikeiden käyttöä C-kielellä, esitellen funktiokutsut ja niiden parametrit. Luvussa 3 esitellään myös pieni esimerkkiohjelma, joka hyödyntää säikeitä syötettyjen komentojen tausta-ajoon. Luvussa 4 esitellään säikeiden käyttökohteita sekä säikeisiin liittyviä hankaluuksia.

2 Säikeet ja niiden ominaisuudet

Luku käsittelee yleisesti säikeitä ja niiden ominaisuuksia. Luvussa ei vielä kuitenkaan syvennytä käytännön toteutukseen, jota luku 3 käsittelee. Luku perustuu pääasiassa lähteeseen [Lewis, luvut 3-6 ja 9].

2.1 Mikä on säie

Säikeet ohjelmoinnissa tarkoittavat "kevyitä prosesseja" (engl. *lightweight process*), jotka toimivat yhden prosessin "sisässä". Ne ovat siis ikäänkuin ohjelman sisällä toimivia ohjelmia. Säikeet jakavat saman muistialueen ja muut resurssit. Yksittäiset prosessit sen sijaan eivät yleensä näe toisten prosessien resursseja.

Säikeet suoritetaan periaatteessa samanaikaisesti, mutta käytännössä **samanaikaisuus** ei välttämättä ole mahdollista. Fyysisesti samanaikaisesti voidaan suorittaa korkeintaan yhtä monta säiettä, kuin tietokoneessa on suorittimia eli prosessoreita. Jos suoritettavia säikeita ja prosesseja on enemmän kuin prosessoreita, joudutaan suoritusaikaa vuorottelemaan säikeiden ja prosessien kesken. Kuitenkin on huomattavaa, että ohjelmoija ei voi yleensä ennustaa säikeiden suoritusjärjestystä.

Koska useat **säikeet käyttävät samaa muistialuetta**, on tärkeää suojata käytetyt resurssit päällekkäisiltä kirjoituksilta. Tähän tarkoitukseen on kehitetty erilaisia menetelmiä, joista yleisimmin käytetty on Posix-säikeiden mutex ja tämän laajennus semafori (katso luku 2.5). Näiden tarkoituksena on ilmaista muille säikeille, että resurssi on varattu, jos jokin muu säie käyttää kyseistä resurssia.

2.2 Säikeen luonti

Kun säie luodaan, se alustaa tarpeelliset tietorakenteet ja alkaa suorittamaan parametrina välitettyä funktiota. Tästä ajanhetkestä lähtien ei enää voida tietää, saako seuraavaksi suoritusaikaa uusi säie vai sen luonut säie.

Normaalisti säie on kuitenkin "kiinni" (engl. *attached*) sen luoneessa säikeessä siten, että luova säie voi odottaa luodun säikeen lopettamista, analogisesti Unix-prosessien kanssa. Säie voidaan kuitenkin haluttaessa **irrottaa luovasta säikeestä**, jolloin sen paluuarvoa ei enää voida tutkia. Tämä merkitsee myös sitä, että jos säie on kiinni luoneessa säikeessä, se ei vapauta varaamiaan resursseja ennenkuin sen paluuarvo on luettu jonkin toisen säikeen toimesta. Irrallinen säie sensijaan vapauttaa varaamansa resurssit heti loputtuaan.

2.3 Ominaisuuksista

Säikeen suoritukseen ja sen tilaan voidaan vaikuttaa muuttelemalla erinäisiä säikeen attribuutteja. Yleisesti attribuutit määrätään jo luontivaiheessa, mutta niitä on myös mahdollista muuttaa ajon aikana.

Kuten jo aikaisemmin mainittiin, säie voi olla kiinnitetyssä tilassa tai irrallisessa tilassa (kts. luku 2.2). Muut attribuutit liittyvät säikeen suorituksen skedulointiin, eli säikeiden suoritusjärjestykseen.

2.4 Säikeen suorituksen lopetus

Säikeen normaali lopetus tapahtuu samankaltaisesti kuin minkä tahansa ohjelman lopetus. Funktio `pthread_exit` aiheuttaa säikeen suorituksen lopettamisen. Toisaalta säikeen luonnissa annetun funktion palattua säikeen suoritus lopetetaan automaattisesti.

Säikeitä voidaan myös **käskeä lopettamaan suorituksensa** kutsulla `pthread_cancel`. Tämä on kuitenkin käyttökelpoinen menetelmä ainoastaan harvoissa tilanteissa, koska säikeen suorituksen kohtaa ei varsinaisesti voida tietää. Onkin parempi asettaa jokin muuttuja (esimerkiksi ehtomuuttuja). Sen perusteella säie itse voi tarkistaa, pitäisikö sen lopettaa suoritus.

2.5 Synkronointi

Jos säikeiden toimintaa tulee pystyä ohjaamaan toisista säikeistä käsin, tulee käyttää erilaisia **synkronointimenetelmiä**. Näitä ovat esimerkiksi ehtomuuttujat (engl. *condition variable*), mutex ja semafori.

Mutex (lyhenne engl. *Mutual Exclusion Lock*) lukitaan kutsulla `pthread_mutex_lock`. Tämän jälkeen sama kutsu ei enää onnistu ennenkuin sama säie avaa mutexin kutsulla `pthread_mutex_unlock`. On huomattava, että mutex sinänsä ei ole sidottu mihinkään resurssiin, vaan ohjelmoijan tulee luoda mutex jokaista yhteiskäytössä olevaa resurssia varten. **Semafori** toimii muuten samoin kuin mutex, mutta se sallii useamman yhtäaikaisen lukinnan. Semaforia käsitellään tarkemmin luvussa 3.4.

Ehtomuuttuja on nimensä mukaisesti muuttuja, jolta voidaan kysyä sen tilaa. Säie voi esimerkiksi odottaa jonkin itsestään riippumattoman tilan muuttumista kutsumalla `pthread_cond_wait`-funktiota, jolloin säie jää odottamaan kyseisen ehdon täyttymistä. Nyt jonkin muun säikeen muuttaessa kyseistä ehtoa, kutsuu tämä säie funktiota `pthread_cond_signal`, jolloin ensimmäinen säie herää ja tarkistaa kyseisen ehdon tilan ja jatkaa suoritustaan.

Edellämainitut ovat säikeiden perussynkronointimenetelmiä ja näistä onkin kehitetty erilaisia **kehittyneempiä menetelmiä,** kuten kirjoittaja-lukija -lukko (engl. *readers-writers -lock*). Se sallii mielivaltaisen määrän lukijoita käyttävän jotakin resurssia, mutta vain yhden kirjoittajan kerrallaan tietenkin siten, että kirjoitettessa ei myöskään saa lukea. On myös kehitetty "aita" (engl. *barrier*), joka pysäyttää säikeiden suorituksen, kunnes kaikki halutut säikeet saavuttavat aitakohdan ohjelmassa. Näin saadaan varmistettua, että kaikki säikeet jatkavat suoritusta samasta kohtaa ohjelmaa.

Näitä kehittyneempiä synkronointimenetelmiä ei usein löydy peruskirjastoista, mutta Posixin mutexien avulla nämä voidaan helpohkosti rakentaa.

3 Säikeiden käyttö

Luvussa tutustutaan säikeiden käyttöön käytännössä. Esimerkki ja kutsut ovat C-kielellä, mutta muissakin Posix-säikeitä tukevissa ohjelmointikielissä on vastaavanlaisia kutsuja. Luvun sisältö perustuu pääosin lähteeseen [Leroy], joka on nähtävissä Unix-järjestelmässä komennolla `man <funktio>`, mikäli Posix-säiekirjaston manuaalit ovat asennettuina.

Seuraavissa luvuissa esitetään säikeiden käytön kannalta vain olennaisimmat kutsut. Luvussa 3.6 on listattuna lyhyin selostuksin lisää säikeisiin liittyviä kutsuja.

3.1 Säikeen luonti ja parametrien välitys

Säie luodaan kutsumalla funktiota

```
int pthread_create(pthread_t *tid,
                   pthread_attr_t *attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

Funktion **parametrit** ovat seuraavat:

`tid` on osoitin säikeen identifioivaan rakenteeseen. Tätä osoitinta voidaan myöhemmin käyttää esimerkiksi säikeen pakotettuun lopetukseen tai sen suorituksen loppumisen odottamiseen.

`attr` on osoitin rakenteeseen, joka sisältää informaatiota halutuista säikeen ominaisuuksista, kuten ajastus, tai siitä, onko säie irrallinen.

`start_routine` on monimutkaisesta ulkonäöstään huolimatta pelkästään sen funktion nimi, jota säikeen halutaan suorittavan. Funktion tulee palauttaa ja vastaanottaa parametrinaan tyypiltään määräämätön osoitin.

`arg` on osoitin suoritettavalle funktiolle välitettävään parametriin (argumenttiin).

Emme käsittele yksityiskohtaisesti säikeen attribuutteja. Yleisestiottaen säikeet toimivat halutulla tavalla oletusattribuutein, jolloin säikeen luovassa kutsussa voidaan parametrina `attr` käyttää nollaosoitinta `NULL`.

On huomattava, että säikeen suorittama funktio ottaa parametrinsa `void`-tyyppisenä osoittimena. Tämä tarkoittaa sitä, että osoittimen tyyppi pitää (yleensä) määrätä uudelleen suoritettavassa funktiossa.

3.2 Säikeen lopetus

Säikeen lopetus tapahtuu joko säikeen suorittaman funktion loppumisella tai eksplisiittisesti kutsumalla funktiota `pthread_exit`. Säie voidaan myös pakottaa loppumaan kutsumalla `pthread_cancel`-funktiota seuraavasti:

```
pthread_exit(void *retval);
pthread_join(pthread_t thread,
             void **thread_return);
pthread_cancel(pthread_t thread);
```

Kutsu `pthread_exit` aiheuttaa **säikeen lopetuksen**. Jos säie oli kiinnitetyssä tilassa, saa säikeen lopetusta odottava toinen säie kutsulla `pthread_join` arvon `retval` osoittimeen `*thread_return`.

Kutsu `pthread_cancel` **pakottaa säikeen** `thread` lopettamaan suorituksensa. Tämä ei yleisesti ottaen ole järkevää, sillä ilman monimutkaisia synkronisaatioita ei voida varmuudella tietää, mitä säie on milloinkin tekemässä.

Säikeen lopetukseen voidaan vaikuttaa funktioilla `pthread_setcancelstate`. Sillä voidaan määrätä, totteleeko säie lopetuskäskyä `pthread_cancel`. Funktio `pthread_setcanceltype` määrittää, suoritetaanko pakotettu lopetus välittömästi vai vasta seuraavan sopivan funktion kohdalla. Posix-standardissa on määritelty tietyt funktiot sellaisiksi, että niiden kohdalla säikeen suoritus lopetetaan, mikäli säie on pakotettu lopettamaan kutsulla `pthread_cancel`. Ohjelmoija voi myös itse asettaa tällaisia kohtia ohjelmaansa funktiolla `pthread_testcancel`.

Säikeen lopettaessa halutaan toisinaan suorittaa joitain **"siivousfunktioita"**. Niitä voidaan määrittää komennolla `pthread_cleanup_push` ja poistaa komennolla `pthread_cleanup_pop`, jotka käyttäytyvät pinomaisesti. Kutsu `pthread_cleanup_push` siis lisää pinoon päällimmäiseksi funktion ja `pthread_cleanup_pop` poistaa päällimmäisen funktion. Funktio `pthread_once` estää halutun funktion suorittamisen useampaan kertaan, joka on hyödyllistä alustusfunktiolle.

3.3 Säikeen suorituksen ohjaus

Kuten luvussa 2.5 mainittiin, säikeen suoritusta voidaan ohjata synkronointimenetelmillä. Tässä ja luvussa 3.4 esitellään vain mutexin ja semaforin käyttöä ohjelmoinnissa. Muihin synkronointimenetelmiin voi tutustua Unixin manuaalisivuilta tai alan kirjallisuudesta (katso [Lewis, luvut 6 ja 7]).

Mutex täytyy ennen sen käyttöä **alustaa** kutsulla

```
pthread_mutex_init(pthread_mutex_t *mutex,
                   const pthread_mutexattr_t *mutexattr);
```

Mutex **poistetaan käytöstä** kutsulla

```
pthread_mutex_destroy(pthread_mutex_t *mutex);
```

kun sitä ei enää tarvita.

Edellisissä kutsuissa **parametrit** ovat seuraavat:

`mutex` on osoitin mutex-rakenteeseen. Tätä tarvitaan aina mutexia lukittaessa ja avattaessa.

`mutexattr` on taas osoitin mutexin ominaisuudet sisältävään rakenteeseen. Näitä ei yleisesti ottaen tarvitse, joten tämä yleensä korvataan nollaosoittimella `NULL`.

Mutexin käyttö tapahtuu seuraavilla kutsuilla:

- `pthread_mutex_lock(pthread_mutex_t *mutex)` odottaa, kunnes mutex vapautuu ja lukitsee sen
- `pthread_mutex_trylock(pthread_mutex_t *mutex)` yrittää lukita mutexin, mutta palauttaa virheen välittömästi, jos lukitus ei onnistu.
- `pthread_mutex_unlock(pthread_mutex_t *mutex)` vapauttaa lukitun mutexin.

3.4 Semafori

Semafori on yksi useimmin käytetyistä synkronointimenetelmistä. Semaforin toiminnan taustalla on **laskuri**. Sitä vähennetään semafori varattessa ja kasvatetaan semafori vapautettaessa. Laskurin arvo ei saa mennä negatiiviseksi, joten semaforin varaus epäonnistuu, jos laskurin arvo on nolla. Semaforin alustuksessa määrätään laskurin arvo määrittäen, montako kertaa semafori voidaan varata.

Semaforia käytetään seuraavilla **funktiokutsuilla** ([Lewis, sivut 355-357]):

```
int sem_init(sem_t *sem, int pshared, int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_getvalue(sem_t *sem);
```

Kaikissa kutsuissa parametri `sem` on osoitin kyseessä olevaan semaforiin. Kutsu `sem_init` alustaa semaforin arvoon `value`. Jos parametrin `pshared` arvo eroaa nollasta, voidaan semafori jakaa prosessien kesken.

Kutsu `sem_destroy` vapauttaa semaforin varaamat resurssit ja poistaa semaforin käytöstä. Kutsu `sem_post` kasvattaa semaforin laskurin arvoa yhdellä. Kutsu `sem_wait` vähentää laskuria yhdellä, mikäli laskurin arvo on nollaa suurempi. Jos näin ei ole, jää kutsu odottamaan laskurin arvon nousua. Kutsu `sem_trywait` sen sijaan koettaa varata semaforin, mutta palaa heti virheen kera, jos se ei onnistu. Kutsulla `sem_getvalue` voidaan kysyä semaforin laskurin arvo.

3.5 Unix-järjestelmän signaaleista

Sen tarkemmin yksityiskohtiin paneutumatta on syytä varoittaa signaaleista [Lewis, luku 10]. Tässä yhteydessä tarkoitettavilla Unix-järjestelmän signaaleilla voidaan ohjata ohjelman suoritusta. Monisäikeiset ohjelmat ja signaalit eivät toimi hyvin yhdessä. Koska kaikki säikeet toimivat saman prosessin sisässä ja signaalit on aina osoitettu tietylle prosessille, on epävarmaa, mikä säie saa signaalin käsiteltäväkseen.

Kyseiseen ongelmaan kuitenkin löytyy hieman apua Posix-säiekirjastosta. Säiekohtaisesti **voidaan asettaa signaalimaski** kutsulla `pthread_sigmask`, joka estää säiettä vastaamasta tiettyihin signaaleihin. Näin ollen, jos ohjelmassa on tarpeen käsitellä signaaleja, voidaan luoda säie, joka käsittelee pelkästään signaaleja. Tällöin mikään muu säie ei välitä signaaleista. Jos säikeistä halutaan antaa signaali, voidaan se tehdä kutsulla `pthread_kill`.

Tällainen signaalien koonti ei kuitenkaan ole kovinkaan käytännöllistä. Jos jokin säie aiheuttaa systeemikutsullaan signaalin (esim. SIGPIPE, katkennut putkitiedosto), ei ole mitään luotettavaa keinoa selvittää, mikä säie tämän signaalin aiheutti. Periaatteessa monimutkaisilla synkronointirakenteilla voitaisiin tutkia, mikä säie on milloinkin tehnyt jotain mahdollisesti signaaleja aiheuttavaa. Todennäköisesti tämänkaltainen lähestymistapa kuitenkin heikentäisi säikeiden käytön hyödyllisyyttä.

3.6 Muita funktioita

Luvussa esitellään muita säikeisiin liittyviä funktiokutsuja. Unix-järjestelmässä (johon on Posix-säiekirjastot asennettu) saa lisätietoa komennolla `man haluttu_funktio`, jossa `haluttu_funktio` on kyseisen funktion nimi. Funktiot löytyvät myös lähteestä [Lewis, liite E].

Funktio `pthread_self` **palauttaa kutsuvan säikeen tunnisteen** tyyppinä `pthread_t`. Tätä tunnistetta tarvitaan kaikissa säikeisiin liittyvissä kutsuissa.

Funktiot `pthread_attr_init(pthread_attr_t *attr)` luo ja `pthread_attr_destroy-(pthread_attr_t *attr)` ja vastaavasti tuhoaa **säikeen alustamisessa käytettävän attribuuttirakenteen** `attr`. Kyseistä attribuuttirakennetta voidaan muokata ja tutkia funktioilla `pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)` ja `pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate)`. Niistä ensimmäinen asettaa ja toinen tutkii, onko attribuuttirakenteessa `attr` määritetty säie irralliseen vai kiinnitettyyn tilaan (katso luku 2.2).

Säikeiden suoritusjärjestykseen vaikuttavat (`set`) ja suoritusjärjestystä tutkivat (`get`) funktiot `pthread_attr_[set/get]schedpolicy` ja `pthread_attr_[set/get]-schedparam`. Edellisistä funktioista ensimmäiset vaikuttavat suoritusjärjestyksen valinnan malliin ja toiset mallin parametreihin. Suoritusjärjestyksen periytymiseen vaikuttaa `pthread_attr_[set/get]inheritsched`. Funktioilla `pthread_attr_[set/get]scope` määritetään tai vastaavasti tutkitaan, kilpaileeko säie suoritusajasta tasavertaisesti prosessien kanssa vai ainoastaan samanarvoisten säikeiden kanssa.

3.7 Yksinkertainen esimerkki

Luvussa tutustutaan mahdolliseen säikeiden käyttöön. Esimerkki on erittäin yksinkertainen, eikä sisällä muuta kuin säikeiden käynnistämisen ja yksinkertaisen semaforin käytön. Esimerkistä ilmenee kuitenkin luonteva käyttötarkoitus säikeille.

Monessa ohjelmassa suoritetaan aikaavieviä operaatioita, joiden tulos ei ole välttämätön ohjelman jatkamiseen. Nämä operaatiot voisi siten suorittaa taustalla, kuten usein onkin tehty. Ohjelma siis lukee käyttäjältä komennon ja käynnistää säikeen tulkitsemaan ja suorittamaan sen.

Tämä yksinkertainen esimerkki näyttää yhden (ei erityisen hyvän) tavan toteuttaa kyseisen kaltaisen "tausta-ajon" sisältävä käyttöliittymä:

```
1  /* Esimerkkiohjelma esim.c säikeiden käytöstä */
2  /* Kääntö esim. gcc -o esim esim.c -lpthread */
3  #include <pthread.h> /* Säikeisiin liittyvät funktiot. */
5  #include <unistd.h> /* Sleep-funktio. */
6  #include <stdio.h> /* Tulostukset ja syötteen luku. */
7  #include <string.h> /* Merkkijonon käsittely. */
8  #include <semaphore.h> /* Semaforin kutsut. */
9  #include <stdlib.h> /* Funktiot malloc ja free. */
10
11 sem_t sem; /* Määritellään semafori globaaliksi muuttujaksi, jotta sitä
12               ei tarvitse erikseen viedä parametrina säikeille. */
13
14 void *suorita_komento(void *arg) {
15   int i,kertoma,viive;
16   double arvo=1.0; /* Lasketaan kertoman arvo reaalilukuun, koska
17                       kokonaislukujen lukualue ei ole riittävän laaja. */
18   /* Irroitetaan säie, jotta se vapauttaa resurssinsa lopetettuaan. */
19   pthread_detach(pthread_self());
20   if (!strncmp((char *)arg,"viive",5)) { /* Tulkataan komento. */
21     viive=atoi((char *)arg+5); /* Luetaan viiveen määrä. */
22     if (viive>0)
23       sleep(viive);
24     printf("Viive arvolla (%d) suoritettu.\n",viive);
25   } else if (!strncmp((char *)arg,"kertoma",7)) {
26     sleep(3); /* Kulutetaan hieman aikaa. */
27     kertoma=atoi((char *)arg+7); /* Luetaan haluttu kertoma. */
28     if (kertoma>1)
29       for (i=2;i<=kertoma;i++) arvo*=i; /* Lasketaan kertoma. */
30     printf("%d:n kertoma on %.0lf\n",kertoma,arvo);
31   } else {
32     printf("Tuntematon komento!\n");
33   }
```

```
34   free((char *)arg); /* Vapautetaan syötteen muisti. */
35   sem_post(&sem); /* Vapautetaan varattu semafori. */
36   return NULL;
37 }
38 int main(void) {
39   pthread_t tid;
40   char *komento;
41   int err;
42   printf("Syötä komento tai 'lopeta', jos haluat lopettaa \n");
43   printf("Komennot: \n");
44   printf(" KertomaXX laskee XX:n kertoman, kun xx <=170\n");
45   printf(" ViiveXX nukkuu xx sekuntia.\n");
46   printf("------------------------------------------------- \n");
47   if ((err=sem_init(&sem,0,5))) { /* Alustetaan semafori arvoon 5.*/
48     printf("Semaforia ei voitu alustaa. Virhe %d.",err);
49     return 1;
50   }
51   while (1) {
52     /* Varataan muisti syötteelle ja luetaan se käyttäjältä. */
53     komento=(char *)malloc(256);
54     fgets(komento,256,stdin);
55     /* Lopetusehdot. */
56     if (!strncmp(komento,"lopeta",6) || strlen(komento)==0) {
57       sem_destroy(&sem);
58       return 0;
59     }
60     /* Koetetaan varata semafori. */
61     if (sem_trywait(&sem)) {
62       printf("Liikaa suoritettavia komentoja! Odota valmistumista.\n");
63     } else {
64       /* Käynnistetään säie tulkitsemaan komento. */
65       pthread_create(&tid,NULL,suorita_komento,komento);
66     }
67   }
68   sem_destroy(&sem); /* Poistetaan semafori käytöstä. */
```

```
69   return 0;
70 }
71
```

Esimerkissä tutkielman kannalta kiintoisat kohdat ovat semaforin käyttö riveillä 35, 47, 57, 61 ja 68 sekä säikeiden käyttö riveillä 19 ja 65. Säie luodaan rivillä 65 kutsulla `pthread_create` ja sille annetaan suoritettavaksi funktioksi `suorita_komento` ja parametriksi `komento`. Funktiossa `suorita_komento` rivillä 19 säie asettaa itsensä (`pthread_self`) irroitettuun tilaan (katso luku 2.2) kutsulla `pthread_detach`.

Rivillä 47 semafori alustetaan (`sem_init`) arvoon `5`, mikä tässä tapauksessa tarkoittaa mahdollisuutta ajaa viittä komentoa yhtäaikaisesti. Rivillä 61 koetetaan varata semafori kutsulla `sem_trywait`. Rivillä 35 funktiossa `suorita_komento` semafori vapautetaan. Kutsu `sem_destroy` riveillä 57 ja 68 poistaa semaforin käytöstä ja vapauttaa sen varaaman muistin.

On huomattava, että tämä esimerkki on erittäin yksinkertainen ja oikeissa ohjelmissa joudutaan yleensä tekemään paljon monimutkaisempia ratkaisuja synkronointiin.

4 Sovelluksia ja käyttömahdollisuuksia

Luvussa kuvataan muutamia käyttökohteita, joissa säikeiden käytöstä saatetaan hyötyä. Lisäksi käsitellään myös hieman säikeiden käyttöön liittyviä hankaluuksia. Luku perustuu pääasiassa lähteeseen [Lewis, luku 2].

4.1 Säikeet palvelinohjelmissa

Palvelinohjelmat usein palvelevat yhtaikaisesti montaa asiakasohjelmaa. Tällöin yleensä joudutaan **yhtäaikaiseen suoritukseen**, jotta vältettäisiin mahdolliset pitkät palvelinvasteet. Monissa palvelinohjelmissa on turvauduttu perinteiseen Unix-tapaan käynnistää uutta palveltavaa varten uusi prosessi. Joissain tapauksissa kuitenkin prosessit saattavat kuluttaa turhan paljon resursseja (esim. muistia), jolloin säikeiden käytöstä voi olla etua.

Uuden asiakkaan ottaessa yhteyttä palvelimeen voidaan käynnistää prosessin sijaan säie. Säikeen käynnistymisaika (käyttöjärjestelmästä riippuen) on yleensä huomattavasti nopeampi kuin prosessin käynnistys, jonka lisäksi säie jakaa muistialueen ja muut resurssit sen luoneen ohjelman kanssa. Tämä ratkaisu voi tietenkin olla myös haitallista. Tapauksesta riippuen säikeiden käyttö voi parantaa palvelinohjelman vasteaikaa huomattavasti. Palvelinohjelmissa on varmasti olemassa myös tapauksia, joissa säikeiden käytöstä ei saada mitään etua tai säikeiden käyttö voi olla jopa haitallista.

4.2 Säikeet käyttöliittymissä

Monesti käyttöliittymäohjelmat tekevät aikaavieviä tehtäviä, esimerkiksi käyttäjän painaessa painiketta. Tällöin ohjelman käyttöliittymä "jumittuu", kunnes ohjelma on suorittanut aikaavievän tehtävän. Usein tämä onkin aivan oikea käyttäytyminen ohjelmalta. On kuitenkin myös tilanteita, jolloin käyttäjä voisi tehdä jotain muuta odotellessa aikaavievän tehtävän valmistumista. **Säikeet auttavat tähän asiaan**.

Käyttöliittymä voi käynnistää säikeen suorittamaan aikaavievän tehtävän, jolloin muu ohjelman suoritus jatkuu ennallaan. Tämä voidaan toteuttaa hyvinkin yksinkertaisesti, kuten luvun 3.7 esimerkki näytti. Tietenkin varsinaisissa ohjelmissa tarvitaan hieman parannuksia kyseiseen esimerkkiin verrattuna, kuten paluuarvon tallennus ja mahdollisia synkronointeja. Mutta periaate on kuitenkin melkein sama eli otetaan vastaan komento ja luodaan säie suorittamaan sitä.

4.3 Säikeiden hankaluudet

Säikeiden käyttö ohjelmistoissa tuo mukanaan myös joitain ongelmia, joihin kuuluu riittämätön synkronointi, umpikujat (engl. *deadlock*) ja kilpailutilanteet (engl. *race condition*). Näistä ehkäpä yleisin on riittämätön synkronointi.

Monisäikeisten ohjelmien suoritusta on yleensä erittäin vaikea hahmottaa. Tämä helposti johtaa tilanteisiin, joissa samaa resurssia käytetään yhtäaikaisesti useammasta säikeestä. Pääsääntöisesti **resurssin yhtäaikainen käyttö vaatii jonkinlaista synkronointia**. Jos kuitenkin resurssin käyttö ei ole synkronoitua, saattaa tämä johtaa virhetilanteisiin, joiden löytäminen ja selvittäminen voi olla vaikeaa. Niinpä on syytä synkronoida monisäikeisen ohjelman kaikki ne resurssit, joita on mahdollista käyttää useammasta säikeestä yhtäaikaa.

Umpikuja (engl. *deadlock*) tarkoittaa tilannetta, jossa säie A pitää mutexia 1 lukittuna ja yrittää lukita mutexia 2. Samanaikaisesti säie B pitää mutexia 2 lukittuna ja yrittää lukita mutexia 1. On selvää, että tämä tilanne ei johda mihinkään, sillä kumpikaan säikeistä ei pysty jatkamaan suoritustaan. Nämä tilanteet johtuvat useimmiten huonosta suunnittelusta tai ohjelmoijan virheestä. Useampia mutexeja (tai muita synkronointikeinoja) lukittaessa tulisi aina seurata tarkkaa hierarkkiaa tai järjestystä, jotta umpiku-

jilta vältyttäisiin. Yksinkertaisin umpikujatilanne onkin se, että säie yrittää lukita jo lukitsemansa mutexin toiseen kertaan.

Kilpailutilanteessa kaksi säiettä tekevät muutoksen samaan resurssiin. Tämä muutos riippuu resurssin edellisestä tilasta. Nyt säikeiden suoritusjärjestyksestä riippuu, mihin tilaan resurssi jää. Kilpailutilanne on usein seuraus huonosta synkronoinnista tai suunnittelusta.

Kuten aiemmista kappaleista käy ilmi, kaikki nämä hankaluudet voidaan välttää huolellisuudella ja hyvällä suunnittelulla. Monisäikeisiä ohjelmia suunnitellessa onkin syytä korostaa hyvän suunnittelun ja täsmällisen arkkitehtuurin merkitystä.

5 Yhteenveto

Tutkielmassa on kuvattu säikeiden yleistä toimintaa, säikeiden käyttöä ohjelmoinnissa sekä tilanteita, joissa säikeiden käytöstä voidaan saavuttaa etua.

Säikeiden käytöstä on havaittu saatavan etua varsinkin palvelinohjelmissa. Niistä on hyötyä myös yksittäisissä ohjelmissa, joissa on tarve tehdä pitkäkestoisia operaatioita. Säikeistä ei kuitenkaan ole pelkästään hyötyä, sillä säikeet vaativat usein monimutkaisia synkronointijärjestelmiä. Huolellisella ohjelmiston suunnittelulla ja toteutuksella kuitenkin pystytään estämään suurin osan säikeiden aiheuttamista ongelmista.

Tutkielman valmistuessa näyttää tilanne siltä, että Posix-säikeet ovat enää historian siipien havinaa. Monet modernimmat ja usein monipuolisemmat säiekirjastot ovat käytännössä täysin vallanneet “markkinat”. Monet näistä kuitenkin perustuvat ainakin osittain Posix-säikeisiin. Monet C++-kirjastot tarjoavat lähinnä vain helppokäyttöisemmän käyttöliittymän Posix-säikeisiin. On myös kirjastoja, jotka käyttävät omia menetelmiään ja metodejaan säikeiden toteutukseen.

Lähteet

Xavier Leroy, "LinuxThreads", manuaalisivut (`man -k pthread`) sekä `http://pauillac.inria.fr/~xleroy/linuxthreads/index.html`, 2002.

Lewis, Bil ja Berg, Daniel J., "Multithreaded Programming with Pthreads", Prentice Hall, 1998.