

# **KSE Sustav u operacijskom sustavu FreeBSD 5.0**

Seminarski rad iz kolegija

"Operacijski sustavi 1",

ak. god. 2002./2003.

Ivan Voras

(0036380923)

## Sažetak

Cilj ovog rada je prikazati mogućnosti postizanja višedretvenog rada u operacijskom sustavu FreeBSD 5.0, sa posebnim osvrtom na novorazvijeni sustav KSE: *Kernel Scheduled Entities* koji donosi mnoga poboljšanja u odnosu na prethodno dostupna rješenja, posebno u podršci za iskorištavanje više sistemskih procesora (ukoliko su prisutni) i konkurentnog izvršavanja poziva jezgre operacijskog sustava.

Praktični dio rada demonstrira korištenje različitih postojećih metoda postizanja višedretvenosti u aplikacijama, te njihove posljedice za aplikaciju i cijeli sustav.

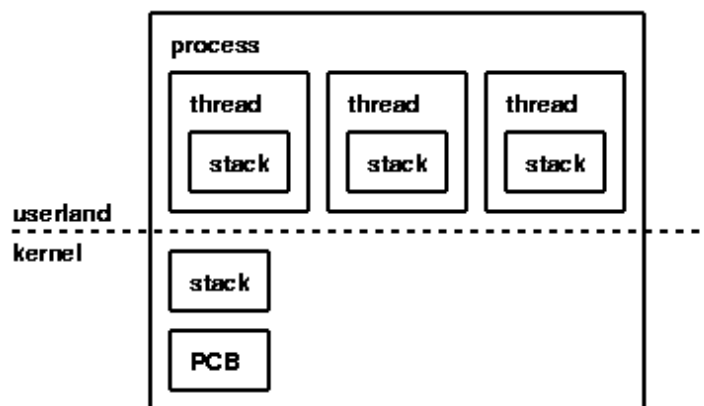
## Postizanje višedretvenosti

U FreeBSD 5.0 sustavu trenutno postoje tri mehanizma za postizanje višedretvenosti u korisničkim programima, s obzirom na odnos dretvi i procesa, te raspoređivanje i upravljanje redoslijedom izvršavanja dretvi (*scheduling*):

- “Korisničke dretve” (*user threads*)
- “Jezgrene dretve” (*kernel threads*)
- KSE Sustav (kernel scheduled entities)

### Korisničke dretve

Korisničke dretve su dobile naziv po tome što se o njihovom održavanju i raspoređivanju brine samo korisnička aplikacija (odn. obično korisnička biblioteka). Ovakav sustav dretvi može se danas implementirati na većini Unix sustava, pošto ne zahtjeva posebnu podršku od jezgre sustava.



1. Opis korisničkih dretvi

Biblioteka za podršku višedretvenosti sama kreira kontekste dretvi te upravlja njima, a jezgra operacijskog sustava “vidi” samo jedan proces (zbog toga se ovakav pristup često označuje kao **N:1**). Paraleliziranje, odn. privid istodobnog izvršavanja dretvi izvodi se korištenjem signala operacijskog sustava, konkretno SIGALARM kojim biblioteka postiže da se nakon određenog vremena (vremenskog prozora, “kvantuma”) izvrši procedura koja upravlja prelaženjem između konteksta pojedinih dretvi. Ta procedura se obično naziva “korisnički raspoređivač dretvi” (*userland thread scheduler, UTS*).

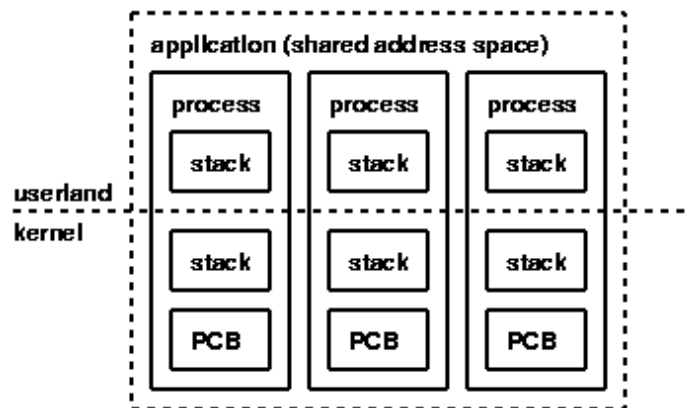
Prednosti ovakvog načina postizanja višedretvenosti su relativna jednostavnost implementacije, i vrlo dobre performanse u slučaju nezahtjevne uporabe. Dodatno, kako se održavanje dretvi obavlja u korisničkoj aplikaciji, moguće je iskoristiti znanje o potrebama aplikacije i po potrebi primijeniti specifične algoritme raspoređivanja izvršavanja.

Mane ovakvog pristupa su:

- Neučinkovitost izvođenja pri korištenju "blokirajućih" sistemskih poziva. Neki takvi sistemski pozivi postoje u više varijanti, te biblioteka može emulirati takav poziv korištenjem asinkrone verzije ako takva postoji. U takvom slučaju dolazi do manjeg pada performansi, ali preostaju pozivi za koje se ne može izbjeći "blokiranje". Tada jedna dretva koja koristi takav poziv sprječava izvršavanje svih ostalih, a u ovisnosti o tome kakav je to poziv, zbog ograničenja sustava, možda i svih drugih procesa. Programi koji malo koriste sistemske pozive (npr. koji rade numeričke proračune) obično nisu podložni ovome.
- Kako jezgra sustava vidi samo jedan proces (i nema informacija o unutrašnjoj organizaciji procesa), nema prilike da učinkovito raspoređuje njegovo izvršavanje u odnosu na resurse računala, posebno s obzirom na broj prisutnih fizičkih procesora. Proces koji koristi ovakav sustav višedretvenosti neće se moći izvršavati istovremeno na više procesora, čak ni kada ovi postoje i slobodni su.

## Jezgrene dretve

Jezgrene dretve (*kernel threads*) čine dijagonalno suprotan način ostvarivanja višedretvenosti. Dok su kod sustava korisničkih dretvi sve dretve "živjele" unutar jednog procesa, i jezgra operacijskog sustava nije prepoznavala pojedinačne dretve, sa svim posljedicama koje takav pristup donosi, ovdje se višedretvenost postiže tako da se svaka dretva odvaja u zasebni proces.



### 2. Opis jezgrenih dretvi

Biblioteka za podršku koristi sistemski poziv kojim se proces razdvaja (*rfork()*) pri svakom kreiranju nove dretve, uz posebnu osobinu da se memorija koju proces koristi proglašava zajedničkom i dostupnom iz svih novokreiranih procesa. Kreira se još i proces za sinkronizaciju, tako da ukupno postoji  $n+1$  procesa za  $n$  kreiranih dretvi. Na ovaj način se problem održavanja i raspoređivanja izvršavanja dretvi prepušta jezgri operacijskog sustava kao da se radi o "normalnim" procesima (pristup se često označuje kao **1:1**).

Prednosti ovakvog pristupa su što se izbjegava blokiranje svih dretvi u slučaju da jedna od dretvi napravi blokirajući sistemski poziv, te se omogućuje potpuno iskorištavanje višestrukih procesora u sustavu, ako su ovi prisutni.

Povijesno, pristup sličan ovome se tradicionalno koristi u praktički svim Unix i Unix-nalik sustavima, kroz sistemski poziv `fork()`, uz ograničenje da se memorijski prostor ne dijeli, već kopira između procesa.

U Linux baziranim sustavima ovo je "službeni" način na koji se postiže višedretveni rad, te se u novije vrijeme čak i koristi naziv "*Linux threads*", prema tako nazvanoj biblioteci za podršku, koja se koristi i u FreeBSD-u.

Mane ovakvog pristupa su:

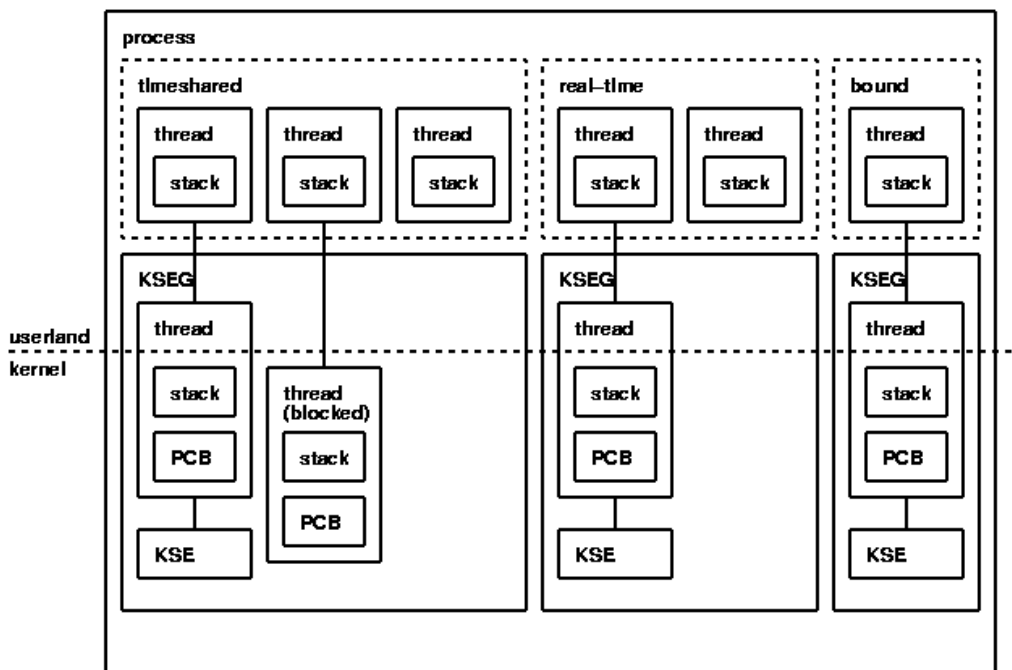
- Nesukladnosti sa POSIX standardom koji zahtjeva da sve dretve pojedinog procesa imaju zajednički *process ID*, te da se nad dretvama može vršiti raspoređivanje izvršavanja kao da pripadaju jednom procesu.
- Pri raspoređivanju izvršavanja, prijelaz između dretvi (izmjena konteksta) je mnogo dugotrajnija operacija nego u slučaju korisničkih dretvi, pošto zahtjeva intervenciju jezgre operacijskog sustava.
- Postoji neravnopravnost pri raspoređivanju izvršavanja procesa, pošto jedna aplikacija može uzrokovati kreiranje mnoštva procesa, koji se u jezgri sustava ravnopravno natječu za izvršavanje sa jednodretvenim aplikacijama.
- Mnoštvo procesa koji se moraju opsluživati rezultira velikom potrošnjom sistemskih resursa (npr. memorije koja je dodijeljena jezgri sustava, te tablica i identifikatora procesa). Ako su prisutna, moraju se ukloniti ili povećati ograničenja na broj procesa koje jedan korisnik smije pokrenuti.

## **Hibridni sustavi**

Razni sustavi, pogotovo komercijalne inačice Unixa, na razne načine kombiniraju osobine gore navedenih pristupa, sa ciljem izbjegavanja loših, i zadržavanja dobrih osobina. Primjeri takvih sustava su LWP (*light-weight processes*) kod Solarisa, i SA (*Scheduler Activations*) koji je utjecao na razvoj KSE sustava u FreeBSD-u.

## **KSE Sustav**

KSE sustav je "hibridni" s obzirom da se višedretvenost postiže kombinacijom funkcionalnosti jezgre operacijskog sustava uz suradnju korisničkog raspoređivača dretvi (*UTS*). Sustav omogućava potrebnu POSIX kompatibilnost i izbjegavanje negativnih osobina. Jezgra operacijskog sustava istovremeno vidi i procese i pojedinačne dretve unutar procesa (te se sustav označuje kao M:N), što uvodi dodatnu složenost u sustav.



### 3. Opis KSE sustava dretvi

Uvodi se pojam KSE grupe (KSEG), koja na određeni način predstavlja skup "korisničkih dretvi". KSE grupe se raspoređuju za izvršavanje u jezgri operacijskog sustava kao što se u klasičnim okolnostima raspoređuju procesi, a dretve unutar KSE grupe se raspoređuju na način na koji se raspoređuju dretve u sustavu korisničkih dretvi (koristeći UTS). Svaki KSEG sadrži pridružene mu dretve, i može sadržavati najviše onoliko pojedinačnih KSE objekata koliko ima procesora u sustavu. U slučaju klasičnog jednodretvenog procesa, proces sadrži jednu KSE grupu, jednu dretvu i jedan KSE objekt.

### **POSIX standard**

POSIX standard 1003.1c specificira API (*Application Programming Interface*) putem kojega se ostvaruje višedretvenost u korisničkim aplikacijama i svi navedeni sustavi posjeduju biblioteke sa POSIX standardnim funkcijama i strukturama. Osobito dobra strana ovakve standardizacije je da jednom napisani program može koristiti bilo koji od njih, u ovisnosti o tome koje biblioteke su korištenje pri kreiranju izvršne datoteke.

## KSE Sustav

Jedan KSE objekt predstavlja ekvivalent "malim procesima" (*Light-weight process*) u nekim drugim operacijskim sustavima, odn. apstrakciju sistemskog procesora. Prilikom raspoređivanja izvršavanja dretvi unutar jedne KSE grupe, konteksti dretvi se dodjeljuju (*attach*) KSE objektu, i tako proglašavaju "aktivnim". Dosljedno ovoj apstrakciji, može se reći da KSE objekt "izvršava" dretve, jednu po jednu, u poretku koji određuje korisnički raspoređivač dretvi (*UTS*). Pojedinačni KSE objekti nisu vidljivi korisničkoj aplikaciji, već služe u raspoređivanju izvršavanja unutar jezgre operacijskog sustava.

KSE sustav zahtjeva suradnju i koordinaciju jezgre operacijskog sustava i korisničke aplikacije, što se postiže pozivanjem korisničkog raspoređivača dretvi (*UTS*) izravno iz jezgre sustava (*upcalls*). Kako sada postoji mogućnost da jedan korisnik, odn. aplikacija kreira više KSE grupa, uvedena je mogućnost sistemskog ograničavanja broja KSE grupa koje jedan korisnik odn. aplikacija mogu kreirati (analogno ograničenju na broj procesa kod "jezgrenih dretvi").

Iako su KSE pozivi normalno dostupni iz korisničkih aplikacija, osim u vrlo specifičnim slučajevima, pojedinačne aplikacije ih vjerojatno neće koristiti, i obično neće biti svjesne njihovog postojanja. Razlog tome je što su pozivi sami po sebi namijenjeni izgradnji biblioteka, i ne mogu se korisno upotrijebiti bez dodatnog koda koji će upravljati održavanjem i raspoređivanjem dretvi, što je neuobičajena zadaća za aplikacijski kod. U razvoju su trenutno 2 biblioteke koje će aplikacijama nuditi podršku za višedretvenost:

- *libpthread*, koja će postati "službena" biblioteka za višedretvenost po POSIX standardu. Ova biblioteka implementira u potpunosti M:N hibridni model i trenutno se razvija pod imenom *libkse*. *libkse* biblioteka je uključena u osnovno stablo FreeBSD 5.0-RELEASE izvornog koda, ali se ne gradi i postavlja zajedno sa ostatkom sustava, već se mora posebno postaviti (iz `/usr/src/lib/libpthread`)
- *libthr*, koja je alternativna biblioteka, i trenutno služi za testiranje KSE sustava (kao *proof of concept*). Razvoj na biblioteci je počeo tek nakon izdanja verzije 5.0-RELEASE, te nije korištena ovdje. Biblioteka implementira 1:1 model višedretvenosti korištenjem KSE sustava (kreira se po jedna KSE grupa za svaku dretvu)

## KSE pozivi (API)

Dokumentacija (`man kse`) navodi sljedeće pozive za rad sa KSE sustavom:

```
int kse_create(struct kse_mailbox *mbx, int newgroup);
int kse_exit(void);
int kse_release(void);
int kse_wakeup(struct kse_mailbox *mbx);
int kse_thr_interrupt(struct kse_thr_mailbox *tmbx);
```

Uvodi se pojam pretinca "*mailbox*", koji označava memorijsku strukturu putem koje komuniciraju jezgra operacijskog sustava i korisnički raspoređivač dretvi. Svaki KSE posjeduje jedan "KSE pretinac" koji se održava sa strane korisničke aplikacije. Pretinac se sastoji od kontrolnih informacija, među kojim je pokazivač na korisničku funkciju raspoređivača dretvi (*upcall*), te stog korisničke aplikacije. Jezgra operacijskog sustava

poziva navedenu funkciju kada KSE postane odvojen (*detached*) od dretve, te korisnička funkcija treba odrediti daljnji tijek izvršavanja. Jezgra operacijskog sustava mijenja podatke u pretincu u ovisnosti o stanju sustava i raspoređivača izvršavanja.

Strukture korištene u pozivima imaju sljedeće definicije:

```

/* KSE mailbox, dodijeljen jednoj KSE grupi */
struct kse_mailbox {
    int km_version; /* Mailbox version */
    struct kse_thr_mailbox *km_curthread; /* Current thread */
    struct kse_thr_mailbox *km_completed; /* Completed threads */
    sigset_t km_sigscaught; /* Caught signals */
    unsigned int km_flags; /* KSE flags */
    kse_func_t *km_func; /* UTS function */
    stack_t km_stack; /* UTS context */
    void *km_udata; /* For use by the UTS */
    struct timespec km_timeofday; /* Time of upcall */
};

/* Thread mailbox, po jedan za svaku dretvu */
struct kse_thr_mailbox {
    ucontext_t tm_context; /* User thread context */
    unsigned int tm_flags; /* Thread flags */
    struct kse_thr_mailbox *tm_next; /* Next thread in list */
    void *tm_udata; /* For use by the UTS */
    unsigned int tm_uticks; /* User time counter */
    unsigned int tm_sticks; /* Kernel time counter */
};

```

Glavna struktura u operacijskom sustavu koja sudjeluju u postizanju višedretvenosti u KSE sustavu je `thread`, iz `sys/proc.h` datoteke:

```

struct thread {
    struct proc *td_proc; /* Associated process. */
    struct ksegrp *td_ksegrp; /* Associated KSEG. */
    TAILQ_ENTRY(thread) td_plist; /* All threads in this proc */
    TAILQ_ENTRY(thread) td_kglist; /* All threads in this ksegrp */

    /* The two queues below should someday be merged */
    TAILQ_ENTRY(thread) td_slpq; /* (j) Sleep queue. XXXKSE */
    TAILQ_ENTRY(thread) td_lockq; /* (j) Lock queue. XXXKSE */
    TAILQ_ENTRY(thread) td_runq; /* (j) Run queue(s). XXXKSE */

    TAILQ_HEAD(, selinfo) td_selq; /* (p) List of selinfos. */

    /* Cleared during fork1() or thread_sched_upcall() */
#define td_startzero td_flags
    int td_flags; /* (j) TDF_* flags. */
    int td_inhibitors; /* (j) Why can not run */
    struct kse *td_last_kse; /* (j) Previous value of td_kse */
    struct kse *td_kse; /* (j) Current KSE if running. */
    int td_dupfd; /* (k) Ret value from fdopen. */
    void *td_wchan; /* (j) Sleep address. */
    const char *td_wmesg; /* (j) Reason for sleep. */
    u_char td_lastcpu; /* (j) Last cpu we were on. */
    u_char td_inktr; /* (k) Currently handling a KTR. */
    u_char td_inktrace; /* (k) Currently handling a KTRACE. */
    short td_locks; /* (k) DEBUG: lockmgr count of locks */
    struct mtx *td_blocked; /* (j) Mutex process is blocked on. */
    struct ithd *td_ithd; /* (b) For interrupt threads only. */
    const char *td_lockname; /* (j) Name of lock blocked on. */
    LIST_HEAD(, mtx) td_contested; /* (j) Contested locks. */
    struct lock_list_entry *td_sleeplocks; /* (k) Held sleep locks. */
    int td_intr_nesting_level; /* (k) Interrupt recursion. */
    struct kse_thr_mailbox *td_mailbox; /* the userland mailbox address */
    struct ucred *td_ucred; /* (k) Reference to credentials. */
    void (*td_switchin)(void); /* (k) Switchin special func. */
    struct thread *td_standin; /* (?) use this for an upcall */

```

```

    u_int          td_usticks;      /* Statclock hits in kernel, for UTS */
    u_int          td_critnest;    /* (k) Critical section nest level. */
#define td_endzero td_base_pri

/* Copied during fork1() or thread_sched_upcall() */
#define td_startcopy td_endzero
    u_char         td_base_pri;    /* (j) Thread base kernel priority. */
    u_char         td_priority;    /* (j) Thread active priority. */
#define td_endcopy td_pcb

/*
 * fields that must be manually set in fork1() or thread_sched_upcall()
 * or already have been set in the allocator, contstructor, etc..
 */
struct pcb        *td_pcb;        /* (k) Kernel VA of pcb and kstack. */
enum {
    TDS_INACTIVE = 0x20,
    TDS_INHIBITED,
    TDS_CAN_RUN,
    TDS_RUNQ,
    TDS_RUNNING
} td_state;
register_t        td_retval[2];    /* (k) Syscall aux returns. */
struct callout    td_slpcallout;   /* (h) Callout for sleep. */
struct trapframe *td_frame;       /* (k) */
struct vm_object *td_kstack_obj;   /* (a) Kstack object. */
vm_offset_t      td_kstack;       /* Kernel VA of kstack. */
int              td_kstack_pages; /* Size of the kstack */
struct vm_object *td_alkstack_obj; /* (a) Alternate kstack object. */
vm_offset_t      td_alkstack;     /* Kernel VA of alternate kstack. */
int              td_alkstack_pages; /* Size of the alternate kstack */
struct mdthread  td_md;           /* (k) Any machine-dependent fields. */
struct td_sched  *td_sched;       /* Scheduler specific data */
};

```

Zanimljivo je još navesti strukturu proc koja sadrži podatke o pojedinim procesima u sustavu:

```

/*
 * The old fashioned process. May have multiple threads, KSEGRPs
 * and KSEs. Starts off with a single embedded KSEGRP, KSE and THREAD.
 */
struct proc {
    LIST_ENTRY(proc) p_list;        /* (d) List of all processes. */
    TAILQ_HEAD(, ksegrp) p_ksegrps; /* (kg_ksegrp) All KSEGs. */
    TAILQ_HEAD(, thread) p_threads; /* (td_plist) Threads. (shortcut) */
    TAILQ_HEAD(, thread) p_suspended; /* (td_runq) suspended threads */
    struct ucred *p_ucred;          /* (c) Process owner's identity. */
    struct filedesc *p_fd;          /* (b) Ptr to open files structure. */
    /* Accumulated stats for all KSEs? */
    struct pstats *p_stats;         /* (b) Accounting/statistics (CPU). */
    struct plimit *p_limit;         /* (m) Process limits. */
    struct vm_object *p_upages_obj; /* (a) Upages object. */
    struct procsig *p_procsig;     /* (c) Signal actions, state (CPU). */

    /*struct ksegrp p_ksegrp;
    struct kse p_kse; */

/*
 * The following don't make too much sense..
 * See the td_ or ke_ versions of the same flags
 */
    int p_flag;                     /* (c) P_* flags. */
    int p_sflag;                    /* (j) PS_* flags. */
    enum {
        PRS_NEW = 0,                /* In creation */
        PRS_NORMAL,                /* KSEs can be run */
        PRS_ZOMBIE
    } p_state;                      /* (j) S* process status. */
};

```

```

pid_t          p_pid;          /* (b) Process identifier. */
LIST_ENTRY(proc) p_hash;      /* (d) Hash chain. */
LIST_ENTRY(proc) p_pglst;     /* (g + e) List of processes in pgrp. */
struct proc     *p_pptr;      /* (c + e) Pointer to parent process. */
LIST_ENTRY(proc) p_sibling;    /* (e) List of sibling processes. */
LIST_HEAD(, proc) p_children;  /* (e) Pointer to list of children. */
struct mtx      p_mtx;        /* (k) Lock for this struct. */

/* The following fields are all zeroed upon creation in fork. */
#define p_startzero      p_oppid
pid_t          p_oppid;        /* (c + e) Save ppid in ptrace. XXX */
struct vmSPACE  *p_vmSPACE;    /* (b) Address space. */
u_int          p_swtime;       /* (j) Time swapped in or out. */
struct itimerval p_realtimer;  /* (h?/k?) Alarm timer. */
struct bintime  p_runtime;     /* (j) Real time. */
int            p_traceflag;    /* (o) Kernel trace points. */
struct vnode    *p_tracep;     /* (c + o) Trace to vnode. */
sigset_t       p_siglist;      /* (c) Sigs arrived, not delivered. */
struct vnode    *p_textvp;     /* (b) Vnode of executable. */
char           p_lock;         /* (c) Proclock (prevent swap) count. */
struct klist    p_klist;       /* (c) Knots attached to this proc. */
struct sigiolst p_sigiolst;    /* (c) List of sigio sources. */
int            p_sigparent;     /* (c) Signal to parent on exit. */
sigset_t       p_oldsigmask;   /* (c) Saved mask from pre sigpause. */
int            p_sig;          /* (n) For core dump/debugger XXX. */
u_long         p_code;         /* (n) For core dump/debugger XXX. */
u_int          p_stops;        /* (c) Stop event bitmask. */
u_int          p_stype;        /* (c) Stop event type. */
char           p_step;         /* (c) Process is stopped. */
u_char         p_pfsflags;     /* (c) Procfs flags. */
struct nlminfo *p_nlminfo;     /* (?) Only used by/for lockd. */
void           *p_aioinfo;     /* (c) ASYNC I/O info. */
struct thread   *p_singlethread; /* (j) If single threading this is it */
int            p_suspcount;     /* (j) # threads in suspended mode */
int            p_userthreads;   /* (j) # threads in userland */
/* End area that is zeroed on creation. */
#define p_endzero      p_sigmask

/* The following fields are all copied upon creation in fork. */
#define p_startcopy    p_endzero
sigset_t       p_sigmask;      /* (c) Current signal mask. */
stack_t        p_sigstk;       /* (c) Stack ptr and on-stack flag. */
u_int          p_magic;        /* (b) Magic number. */
char           p_comm[MAXCOMLEN + 1]; /* (b) Process name. */
struct pgrp    *p_pgrp;        /* (c + e) Pointer to process group. */
struct sysentvec *p_sysent;     /* (b) Syscall dispatch info. */
struct pargs   *p_args;        /* (c) Process arguments. */
rlim_t         p_cpulimit;     /* (j) Current CPU limit in seconds. */
/* End area that is copied on creation. */
#define p_endcopy      p_xstat

u_short        p_xstat;        /* (c) Exit status; also stop sig. */
int            p_numthreads;    /* (?) number of threads */
int            p_numksegrps;    /* (?) number of ksegrps */
struct mdproc  p_md;           /* (c) Any machine-dependent fields. */
struct callout p_itcallout;    /* (h) Interval timer callout. */
struct user    *p_uarea;       /* (k) Kernel VA of u-area (CPU) */
u_short        p_acflag;       /* (c) Accounting flags. */
struct rusage  *p_ru;          /* (a) Exit information. XXX */
struct proc    *p_peers;       /* (r) */
struct proc    *p_leader;      /* (b) */
void           *p_emuldata;     /* (c) Emulator state data. */
struct label   p_label;        /* process (not subject) MAC label */
struct p_sched *p_sched;       /* Scheduler specific data */
};

```

## Primjer korištenja

FreeBSD 5.0 sustav predstavlja gotovo idealnu okolinu za testiranje ovakvog sustava jer su istovremeno i bez kolizija dostupni svi navedeni mehanizmi postizanja višedretvenosti, odn. biblioteke za njihovu podršku:

- Korisničke dretve su podržane izravno od sustava u `libc_r` biblioteci
- Jezgrene dretve su podržane putem `linuxthreads` paketa
- KSE dretve su podržane putem eksperimentalne biblioteke `libkse` (nakon potpunog stabiliziranja sustava, biblioteka će ući u službenu uporabu kao `libpthread`)

Izvorni kod programa za testiranje je napisan po POSIX standardu (koristeći `pthread` pozive) te se generiranje izvršnog koda koji koristi različite načine postizanja višedretvenosti postiže navođenjem različitih parametara prevodiocu (*compiler*), odn. različitih biblioteka.

## Testni program

Testni program implementira jednostavni proizvođač/potrošač ("*producer/consumer*") slučaj sa parovima dretvi, jedna od kojih "proizvodi" podatke čitajući iz `/dev/random`, dok ih druga "koristi" zapisujući podatke u `/dev/null`, uz korištenje POSIX *mutex* i *condition* varijabli za sinkronizaciju. Ovim testom se ispituje kako višedretvenost utječe na propusnost sustava i IO-intenzivne aplikacije kakve se susreću u npr. raznim Internet serverima.

```
/*
    A threading consumer-producer example/benchmark, using
    portable pthread API

    by Ivan Voras <ivoras@fer.hr>
*/
#define _REENTRANT
#include <stdio.h>
#include <pthread.h>

#define THREAD_COUNT 100      // lo:100 mid:1000 hi:10000    actual thread count: THREAD_COUNT*2
#define BLOCK_SIZE 2048      // lo:2048 mid:5001 hi:10001
#define ITER_COUNT 200        // constant

/*
    One consumer/producer element, with appropriate control structures
    and buffers
*/
struct elem {
    int id;
    char *buf;
    int flag;
    int ccount, pcount;
    pthread_t consumer;
    pthread_t producer;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
};

/*
    Consumer function
    Writes buffer to /dev/null
*/
```

```

void *consumerfunc (struct elem* el) {
    FILE *f = fopen ("/dev/null", "w");

    while (1) {
        pthread_mutex_lock (&el->mutex);

        while (el->flag == 0)
            pthread_cond_wait (&el->cond, &el->mutex);

        fwrite (el->buf, BLOCK_SIZE, 1, f);

//         printf ("%3d consumed %3d\n", el->id, el->ccount++);

        el->flag = 0;
        pthread_cond_signal (&el->cond);
        pthread_mutex_unlock (&el->mutex);

    }
    fclose (f);
}

/*
    Producer function
    Reads data from /dev/random into a buffer
*/
void *producerfunc (struct elem* el) {
    FILE *f = fopen ("/dev/random", "r");
    int i;

    for (i = 0; i < ITER_COUNT; i++) {
        pthread_mutex_lock (&el->mutex);

        while (el->flag == 1)
            pthread_cond_wait (&el->cond, &el->mutex);

        fread (el->buf, BLOCK_SIZE, 1, f);

//         printf ("%3d produced %3d\n", el->id, el->pcount++);

        el->flag = 1;

        pthread_cond_signal (&el->cond);
        pthread_mutex_unlock (&el->mutex);

    }
    fclose (f);
}

/*
    main()
    Initialisation and management of consumer/producer elements
*/
int main() {
    struct elem threads[THREAD_COUNT];
    int i;

    puts ("Thread test.");

    for (i = 0; i < THREAD_COUNT; i++) {
        threads[i].id = i;
        threads[i].flag = 0;
        threads[i].ccount = 0;
        threads[i].pcount = 0;
        threads[i].buf = malloc (BLOCK_SIZE);
        pthread_cond_init (&threads[i].cond, NULL);
        pthread_mutex_init (&threads[i].mutex, NULL);
        pthread_create (&threads[i].producer, NULL, producerfunc, (void*)&threads[i]);
        pthread_create (&threads[i].consumer, NULL, consumerfunc, (void*)&threads[i]);
    }

    for (i = 0; i < THREAD_COUNT; i++)
        pthread_join (threads[i].producer, NULL);

    return 0;
}

```

Kako je program napisan tako da koristi standardne pthreads pozive, jedina razlika u prevođenju programa u izvršni kod se odnosi na korištene biblioteke, kako je i vidljivo iz Makefile datoteke.

Program je napisan namjerno jednostavno, i ne izvodi provjere događanja pogreški pri pozivu niti eksplicitno oslobađa resurse, već ostavlja te akcije operacijskom sustavu, odn. bibliotekama za podršku višedretvenosti.

## **Makefile**

```
all: thrtest_usr thrtest_kse thrtest_lin

thrtest_usr: thrtest.c
    gcc -o thrtest_usr -pthread -D_THREAD_SAFE thrtest.c

thrtest_kse: thrtest.c
    gcc -o thrtest_kse -D_LIBC_R_ -D_THREAD_SAFE -lkse thrtest.c

thrtest_lin: thrtest.c
    gcc -o thrtest_lin -D_THREAD_SAFE -I/usr/local/include/pthread/linuxthreads \
    -L/usr/local/lib -llthread -llgcc_r thrtest.c
```

Korištene su opcije prevodioca prema navodima iz dokumentacije i/ili primjera razvojnog koda. Izvršavanjem naredbe make kao rezultat se kreiraju tri izvršne datoteke: thrtest\_usr, thrtest\_kse i thrtest\_lin, koje sadrže program u koji je uključena podrška za korisničke dretve, KSE sustav i jezgrene dretve, respektivno.

## **Metodologija mjerenja**

Mjerenja su izvedena pomoću /usr/bin/time programa sa parametrima -l-h-p koji označavaju ispis detaljne evidencije potrošnje resursa, broja obrađenih sistemskih signala i sl. Putem navedenog time programa mjereno je trajanje izvršavanja testnog programa, za različiti broj dretvi te veličinu bloka podataka koji se prenosi između dretvi. Sva mjerenja su izvedena na istom računalu, Pentium III na 933MHz, sa 512Mb radne memorije, pod FreeBSD 5.0-RELEASE sustavom (GENERIC kernel), u jednokorisničkom načinu rada. Mjerenja su izvršena po tri puta, a ovdje su prikazane srednje vrijednosti mjerenja.

Kako se vidi na izvornom kodu programa, posebno su definirani parametri testiranja:

- THREAD\_COUNT – definira broj parova proizvođač/korisnik dretvi koji se kreiraju
- BLOCK\_SIZE – veličina bloka podataka koji se koristi prilikom testiranja (blok te veličine se čita iz /dev/random i zapisuje u /dev/null)
- ITER\_COUNT – broj iteracija "proizvodnja-potrošnja" koje se izvode

Od tih podatak, ITER\_COUNT je prilikom mjerenja držan konstantnim, a THREAD\_COUNT i BLOCK\_SIZE su mijenjani tako da čine tri slučaja:

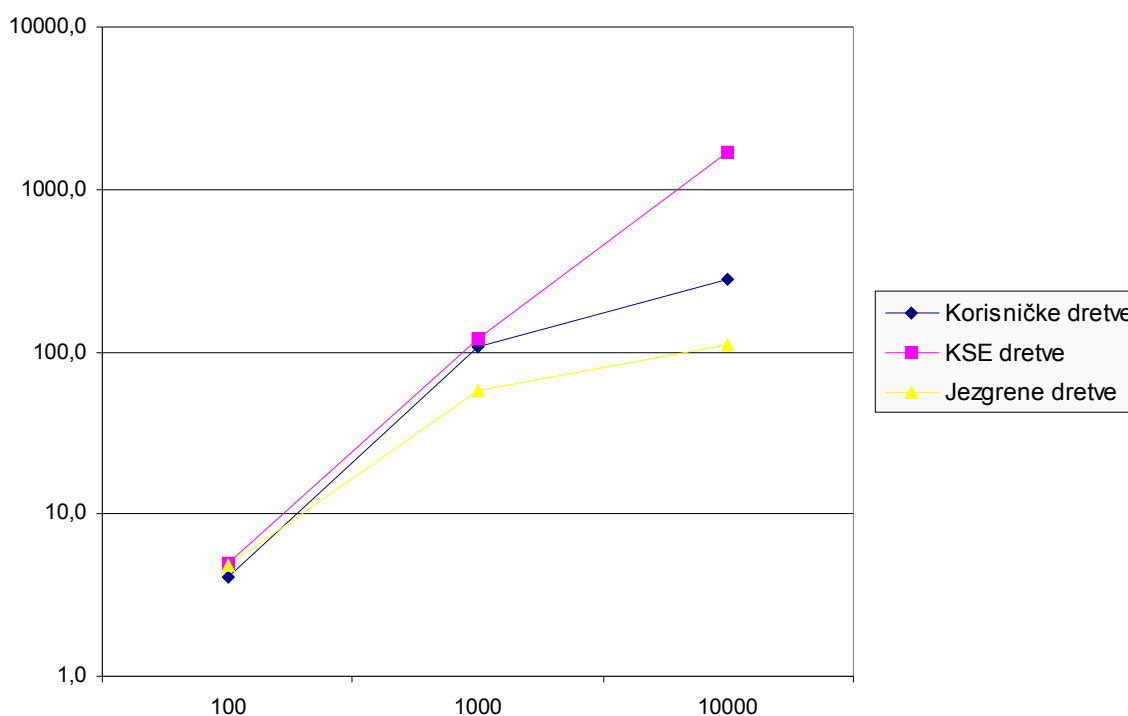
1. THREAD\_COUNT=100, BLOCK\_SIZE=2048 – predstavlja relativno malo opterećenje sustava s obzirom na broj kreiranih dretvi
2. THREAD\_COUNT=1000, BLOCK\_SIZE=5001 – malo veće opterećenje
3. THREAD\_COUNT=10000, BLOCK\_SIZE=10001 – veliko opterećenje s obzirom na broj dretvi. Neki od korištenih biblioteka za podršku višedretvenosti nisu mogle završiti testove pri ovim postavkama

## Rezultati mjerenja

Rezultati testova, ispisi naredbe time su (po redcima su slučajevi opterećenja):

	Korisničke dretve	KSE dretve	Jezgrene dretve
1	real 4.12 user 0.24 sys 3.86 . 2840 maximum resident set size 4 average shared memory size 1002 average unshared data size 128 average unshared stack size 602 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 412 signals received 0 voluntary context switches 423 involuntary context switches	real 4.99 user 0.94 sys 4.03 2640 maximum resident set size 4 average shared memory size 1093 average unshared data size 128 average unshared stack size 570 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 0 signals received 0 voluntary context switches 504 involuntary context switches	real 4.80 user 0.26 sys 4.52 2760 maximum resident set size 4 average shared memory size 688 average unshared data size 128 average unshared stack size 492 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 13928 signals received 13793 voluntary context switches 14230 involuntary context switches
2	real 107.49 user 11.45 sys 95.74 . 27852 maximum resident set size 4 average shared memory size 17301 average unshared data size 128 average unshared stack size 6829 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 10747 signals received 0 voluntary context switches 10866 involuntary context switches	real 121.19 user 20.01 sys 100.77 20392 maximum resident set size 3 average shared memory size 8918 average unshared data size 127 average unshared stack size 4995 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 0 signals received 0 voluntary context switches 12250 involuntary context switches	real 57.81 user 3.86 sys 53.77 13336 maximum resident set size 4 average shared memory size 3866 average unshared data size 128 average unshared stack size 2732 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 95642 signals received 94971 voluntary context switches 100763 involuntary context switches
3	real 277.22 user 6.95 sys 175.95 . 835632 maximum resident set size 4 average shared memory size 70543 average unshared data size 384 average unshared stack size 208990 page reclaims 0 page faults 0 swaps 5 block input operations 13038 block output operations 0 messages sent 0 messages received 17631 signals received 14195 voluntary context switches 30933 involuntary context switches	real 1708.30 user 232.07 sys 1403.12 600076 maximum resident set size 4 average shared memory size 7374 average unshared data size 384 average unshared stack size 150117 page reclaims 0 page faults 0 swaps 3 block input operations 9364 block output operations 0 messages sent 0 messages received 0 signals received 10199 voluntary context switches 174874 involuntary context switches time: command terminated abnormally	real 110.96 user 6.64 sys 102.45 16304 maximum resident set size 4 average shared memory size 5822 average unshared data size 640 average unshared stack size 3749 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 117375 signals received 116898 voluntary context switches 128656 involuntary context switches

Kako je u većini praktičnih primjena najvažniji podatak vrijeme izvršavanja, slijede grafovi ovisnosti vremena izvršavanja o broju dretvi (pretpostavlja se da veličina bloka podataka nema veliki utjecaj):



#### 4. Logaritamski graf ovisnosti duljine izvršavanja o broju dretvi

Zaključci na koje navodi ovaj graf jesu:

- Jezgrene dretve (linuxthreads) se pokazuju općenito najbolji izbor s obzirom na vrijeme izvršavanja
- Korisničke dretve su najbolji izbor ako je broj potrebnih dretvi mali
- Korisničke dretve i jezgrene dretve pokazuju gotovo linearno produljenje izvođenja programa s obzirom na porast broja dretvi (jezgrene dretve su puno bliže linearnom porastu), dok KSE sustav pokazuje gotovo eksponencijalno produljenje, pogotovo u slučaju sa velikim brojem dretvi, koji se nije niti mogao izvršiti do kraja). Ovo se može objasniti time što je KSE sustav još u razvoju i tek nadolazeća verzija FreeBSD 5.1-RELEASE će imati dovoljno stabilan KSE sustav.

Osim ovisnosti vremena izvođenja, iz tablice rezultata moguće je primijetiti i slijedeće podatke:

- Korisničke i jezgrene dretve koriste sustav jezgrenih signala za sinkronizaciju, dok KSE sustav ne (jezgrene dretve koriste najveći broj signala)
- Jezgrene dretve imaju najveću vrijednost "involuntary context switches" podatka, što, zajedno sa velikim korištenjem signala ukazuje na veliko opterećenje jezgre sustava

Druga vrsta podataka koja se može sakupiti odnosi se na ponašanje sustava pri izvršavanju testnog programa.

The image shows three screenshots of the 'top' command output in a terminal window. Each screenshot displays system statistics and a list of running processes. The first screenshot shows a system with 28 processes, 71.6% user CPU usage, and 172M total memory. The second screenshot shows 229 processes, 65.5% user CPU usage, and 172M total memory. The third screenshot shows 229 processes, 65.5% user CPU usage, and 172M total memory. The process list includes columns for PID, USERNAME, PRI, NICE, SIZE, RES, STATE, TIME, WCPU, CPU, and COMMAND.

## 5. Praćenje stanja sustava (top) tijekom izvršavanja testnog programa

Iz podataka o stanju sustava je jasno da je na sustavu normalno prisutno 27 procesa, te je testni proces 28. Ali pri korištenju jezgrenih dretvi, kreira se 200 procesa, za svaku dretvu po jedan, i još jedan proces koji služi za sinkronizaciju, te je ukupan broj procesa tada 229.

KSE sustav uzrokuje neočekivane podatke za parametre prioriteta i "nice" vrijednosti procesa, što nije objašnjivo specifikacijama sustava, i vjerojatno predstavlja neku nuspojavu ili pogrešku.

Neočekivano, u sva tri slučaja je odnos vremena provedenih u jezgri i u korisničkom kodu otprilike jednak. Slučaj korisničkih dretvi ima najpovoljniji odnos, ali i najviše vremena provedenom u jezgrenom kodu obrade prekida.

Slijedeći podaci koji pomažu shvaćanju implementacije različitih sustava dretvi odnose se na dinamiku izvođenja proizvođač/potrošač parova dretvi. Iako je tijekom mjerenja zakomentiran kod koji ispisuje poruke o tome koja iteracija koje dretve se trenutno izvodi, uključivanjem te opcije dobiju se zanimljivi rezultati:

Korisničke dretve	KSE dretve	Jezgrene dretve
0 produced 0	0 produced 0	0 produced 0
0 consumed 0	0 consumed 0	0 consumed 0
1 produced 0	0 produced 1	0 produced 1
1 consumed 0	0 consumed 1	1 produced 0
2 produced 0	0 produced 2	0 consumed 1
2 consumed 0	1 produced 0	1 consumed 0
3 produced 0	0 consumed 2	0 produced 2
3 consumed 0	0 produced 3	1 produced 1
4 produced 0	0 consumed 3	2 produced 0
4 consumed 0	1 consumed 0	0 consumed 2
5 produced 0	1 produced 1	1 consumed 1
5 consumed 0	0 produced 4	2 consumed 0
6 produced 0	1 consumed 1	0 produced 3
6 consumed 0	0 consumed 4	1 produced 2
7 produced 0	1 produced 2	2 produced 1
7 consumed 0	2 produced 0	0 consumed 3
8 produced 0	0 produced 5	1 consumed 2
8 consumed 0	0 consumed 5	2 consumed 1
9 produced 0	1 consumed 2	0 produced 4
9 consumed 0	1 produced 3	1 produced 3

Pri korištenju korisničkih dretvi, očito se parovi dretvi, barem na početku, izmjenjuju vrlo slijedno – kako proizvođač pošalje poruku da je proizveo djelić informacije, odmah se izvede dretva potrošač, a zatim se prelazi na slijedeći par. KSE sustav i jezgrene dretve kao da malo veći prioritet daju dretvama koje su prve kreirane, ali ovo je donekle očekivano, pošto se dretve puštaju u rad slijedno, odmah nakon što su kreirane.

## Implementacija bez korištenja mutex objekata

Zanimljivo je pogledati ponašanje testnog primjera kada se ne koriste *mutex* i *conditional variable* mogućnosti operacijskog sustava. Za tu svrhu su procedure koje se izvode kao dretve proizvođač i potrošač malo preuređene:

```

/*
    Consumer function
    Writes buffer to /dev/null
*/
void *consumerfunc (struct elem* el) {
    FILE *f = fopen ("/dev/null", "w");

    while (1) {
        while (el->who != 0)
            ;
        fwrite (el->buf, BLOCK_SIZE, 1, f);
        printf ("%3d consumed %3d\n", el->id, el->ccount++);
        el->who = 1;
    }
    fclose (f);
}

/*
    Producer function
    Reads data from /dev/random into a buffer
*/
void *producerfunc (struct elem* el) {
    FILE *f = fopen ("/dev/random", "r");
    int i;

    el->who = 1;

    for (i = 0; i < ITER_COUNT; i++) {
        while (el->who != 1)
            ;
        fread (el->buf, BLOCK_SIZE, 1, f);
        printf ("%3d produced %3d\n", el->id, el->pcount++);
    }
}

```

```

        e1->who = 0;
    }
    fclose (f);
}

```

Umjesto da raspoređivač procesa operacijskog sustava uz pomoć mehanizama *mutex* i *conditional variable* određuje spremnost neke dretve za izvođenje, koristi se metoda petlje čekalice koja konstanto provjerava mogućnost svoga izvođenja. Očekivano, izvršavanje je mnogo sporije te je broj aktivnih dretvi smanjen na 10.

Korisničke dretve	Jezgrene dretve
real 96.37 user 90.27 sys 2.88 944 maximum resident set size 2 average shared memory size 125 average unshared data size 87 average unshared stack size 123 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 5970 signals received 0 voluntary context switches 9935 involuntary context switches	real 119.05 user 0.03 sys 0.05 624 maximum resident set size 1 average shared memory size 13 average unshared data size 53 average unshared stack size 63 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 29 signals received 32 voluntary context switches 5 involuntary context switches

Uočljivo je nekoliko podataka:

- Iako se radi o samo 10 dretvi, izvršavanje je drastično sporije
- Korištenje korisničkih dretvi daje nešto brže rezultate, uz više vremena provedenog u jezgri za sinkronizaciju
- Prilikom izvršavanja izvršnog programa povezanog sa KSE bibliotekama, izvršavanje uđe u potpuni zastoj odmah nakon prve iteracije prve dretve (ne dozvoljava se uopće izvršavanje drugih dretvi, što je vjerojatno greška u sustavu).

Ovakav način izvršavanja, koristeći petlje čekalice je vrlo neučinkovit. Ali sa samo malom promjenom se izvršavanje može drastično ubrzati: dodavanjem poziva `pthread_yield()` u petlje čekalice, kojim signaliziramo sustavu da program više ne obavlja korisnu zadaću i sustav može preusmjeriti tok izvršavanja na slijedeću dretvu:

```

while (e1->who != 0)
    pthread_yield();

```

Rezultati izvršavanja su:

Korisničke dretve	KSE dretve	Jezgrene dretve
real 2.89 user 0.47 sys 0.87 944 maximum resident set size 2 average shared memory size 126 average unshared data size 90 average unshared stack size 123 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 86 signals received 0 voluntary context switches 462 involuntary context switches	real 3.36 user 0.54 sys 0.97 828 maximum resident set size 2 average shared memory size 115 average unshared data size 94 average unshared stack size 112 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 0 signals received 2 voluntary context switches 472 involuntary context switches	real 28.20 user 0.00 sys 0.02 268 maximum resident set size 2 average shared memory size 2 average unshared data size 64 average unshared stack size 63 page reclaims 0 page faults 0 swaps 0 block input operations 0 block output operations 0 messages sent 0 messages received 24 signals received 28 voluntary context switches 1 involuntary context switches

Ovaj puta:

- Korisničke dretve daju uvjerljivo najbolje rezultate (potpuno usporedive sa rezultatima dobivenim prilikom korištenja *mutex* i *conditional variable* mogućnosti)
- Prilikom izvršavanja programa koji koristi KSE sustav, u nekoliko navrata je došlo do blokiranja programa, mogućeg potpunog zastoja uslijed pogreške
- Prilikom pokretanja programa koji koristi jezgrene dretve, očito je bilo da se sistemski raspoređivač dretvi ne snalazi u ovom okruženju tako dobro, jer je usred izvršavanja dolazilo do čestog zastajkivanja i kratkotrajnih prekida izvršavanja

## **Zaključak**

Iako je KSE sustav po svojim svojstvima definitivno najbolji način postizanja višedretvenosti, trenutno se pokazuju problemi kod njegovog praktičnog korištenja, koji će ipak nesumnjivo biti uskoro riješeni.

Između korisničkih i jezgrenih dretvi, odabir ovisi prvenstveno o namjeni i složenosti aplikacije. Korištenjem standardnih API poziva moguće je izgraditi aplikaciju koja bez problema može koristiti bilo koji od navedenih sustava. Za manje aplikacije i nezahtjevne primjene, moguće je bez problema koristiti korisničke dretve, čak i bez korištenja sinkronizacijskih mehanizama (koristeći `pthread_yield()` poziv i metodu kooperativne višezadačnosti – obavještavajući sustav kada je najbolji trenutak da počne izvršavati neku drugu dretvu). Velike i zahtjevne aplikacije će vjerojatno odabrati jezgrene dretve, već i zbog toga što korisničke dretve ne omogućavaju iskorištavanje više procesora u sustavu.

## Literatura

Jason Evans, Julian Elischer: Kernel Scheduled Entities for FreeBSD

<http://www.aims.com.au/chris/kse/docbook/>

FreeBSD "man" pages

man kse

The FreeBSD Handbook

[http://www.freebsd.org/doc/en\\_US.ISO8859-1/books/handbook/index.html](http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html)

The FreeBSD developer's Handbook

[http://www.freebsd.org/doc/en\\_US.ISO8859-1/books/developers-handbook/index.html](http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/index.html)

The FreeBSD FAQ

[http://www.freebsd.org/doc/en\\_US.ISO8859-1/books/faq/index.html](http://www.freebsd.org/doc/en_US.ISO8859-1/books/faq/index.html)

The LinuxThreads Library

<http://pauillac.inria.fr/~xleroy/linuxthreads/>

# Sadržaj

<b>SAŽETAK.....</b>	<b>2</b>
<b>POSTIZANJE VIŠEDRETVENOSTI.....</b>	<b>3</b>
<b>KORISNIČKE DRETVE.....</b>	<b>3</b>
<b>JEZGRENE DRETVE.....</b>	<b>4</b>
<b>HIBRIDNI SUSTAVI.....</b>	<b>5</b>
<b>KSE SUSTAV.....</b>	<b>5</b>
<b>POSIX STANDARD.....</b>	<b>6</b>
<b>KSE SUSTAV.....</b>	<b>7</b>
<b>KSE POZIVI (API) .....</b>	<b>7</b>
<b>PRIMJER KORIŠTENJA.....</b>	<b>12</b>
<b>TESTNI PROGRAM.....</b>	<b>12</b>
<b>MAKEFILE.....</b>	<b>14</b>
<b>METODOLOGIJA MJERENJA.....</b>	<b>14</b>
<b>REZULTATI MJERENJA.....</b>	<b>15</b>
<b>IMPLEMENTACIJA BEZ KORIŠTENJA MUTEX OBJEKATA.....</b>	<b>18</b>
<b>ZAKLJUČAK.....</b>	<b>21</b>
<b>LITERATURA.....</b>	<b>22</b>
<b>SADRŽAJ.....</b>	<b>23</b>