CppTss Izvršilac

UDŽBENIK, POGLAVLJE 7, STRANICE 110-146

Delovi CppTss izvršioca

- Deo CppTss-a koji ima ulogu **jezgra** operativnog sistema se naziva **izvršilac**.
- Funkcionalna sličnost CppTss izvršioca sa operativnim sistemom ima za posledicu sličnost njihovih izvedbi.
- Struktura CppTss izvršioca se može predstaviti pomoću istih slojeva kao i struktura operativnog sistema.
- Ključna razlika je da CppTss izvršilac ne sadrži modul za rukovanje datotekama, jer CppTss ne podržava pojam datoteke.

Delovi CppTss izvršioca

sistemske niti	thread_wake_up_deamon() thread_destroyer_deamon() thread_zero() klasa Delta
modul za rukovanje procesima	klasa thread klasa Thread_image
modul za rukovanje datotekama	-
modul za rukovanje random memorijom	klasa Memory_fragment
modul za rukovanje kontrolerima	klasa Timer_driver klasa Exception_driver klasa Driver
modul za rukovanje procesorom	klasa condition_variable klasa unique_lock klasa mutex klasa Kernel klasa Atomic_region klasa Ready_list klasa Descriptor klasa Permit klasa List_link klasa Failure

Klasa Failure

```
enum Failure_codes { MEMORY_SHORTAGE, NOTIFY_OUTSIDE_EXCLUSIVE_REGION };

class Failure {
        protected:
                const char* f_name;
                Failure_codes f_code;
public:
        Failure(const char* fname, const Failure_codes fcode)
                : f_name(fname), f_code(fcode) {};
        inline const char* name() const { return f_name; };
        inline Failure_codes code() const { return f_code; };
};

Failure failure_memory_shortage("MEMORY SHORTAGE!", MEMORY_SHORTAGE);
Failure failure_notify_outside_exclusive_region(
        "NOTIFY OUTSIDE EXCLUSIVE REGION!",
        NOTIFY_OUTSIDE_EXCLUSIVE_REGION
);
```

Klasa List_link

```
class List_link {
          List_link* left;
          List_link* right;
          List_link(const List_link&);
          List_link& operator=(const List_link&);
public:
          List_link() { right = this; left = this; };
          List_link* left_get() const { return left; };
          List_link* right_get() const { return right; };
          void insert(List_link* const link);
          List_link* extract();
          bool empty() const { return (this == right); };
          bool not_empty() const { return !empty(); };
};

void List_link::insert(List_link* const link)
{
          link->left = left;
          link->right = this;
          left->right = link;
          left = link;
}
```

Klasa List_link

```
List_link* List_link::extract()
{
        List_link* p = right;
        right->right->left = this;
        right = right->right;
        return p;
}
```

Klasa List_link

- Klasa **`List_link`** omogućuje obrazovanje **dvosmerne cirkularne** liste.
- Nju predstavlja objekat ove klase, koji tada istovremeno služi kao njen **početak** i **kraj**.
- Takođe se podrazumeva da se oni uvezuju na njen **kraj** i da se izvezuju sa njenog **početka**.
- Dvosmerna cirkularna lista se obrazuje pomoću polja **left** i **right**.
- Operacije klase List_link omogućuju preuzimanje vrednosti ovih polja: **`left_get()`**, **`right_get()`**

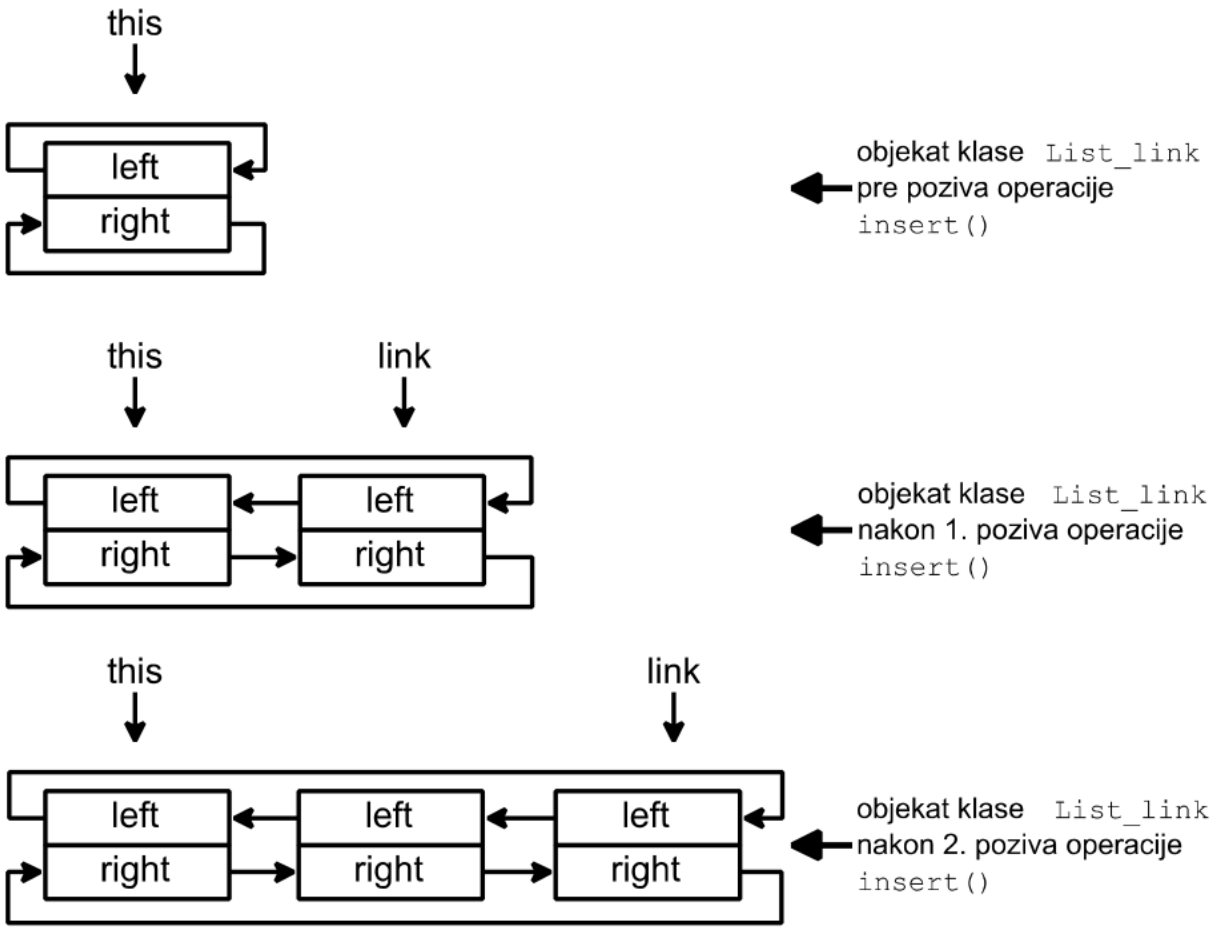
–**uvezivanje** novog elementa **na kraj** liste - **`insert()`**

–**izvezivanje** elementa **sa početka** liste - **`extract()`**

–proveru da li u listi ima uvezanih elemenata - **`not_empty()`**

–proveru da li je lista prazna - **`empty()`**

Klasa List_link



Klasa Permit

```
class Permit {
        bool free;
        Permit* previous;
        List_link admission_list;
        List_link fulfilled_list;
public:
        Permit() { free = true; previous = 0; };
        inline bool not_free() const { return(free == false); };
        inline void take() { free = false; };
        inline void release() { free = true; };
        inline void admission_insert(List_link* link)
                { admission_list.insert(link); };
        inline void fulfilled_insert(List_link* link)
                { fulfilled_list.insert(link); };
        inline List_link* admission_extract()
                { return admission_list.extract(); };
        inline List_link* fulfilled_extract()
                { return fulfilled_list.extract(); };
        inline bool admission_not_empty()
                { return admission_list.not_empty(); };
        inline bool fulfilled_not_empty()
                { return fulfilled_list.not_empty(); };
        friend class Descriptor;
};
```

Klasa Permit

- Klasa **`Permit`** opisuje rukovanje propusnicama. Polje **`free`** klase **`Permit`** čuva stanje propusnice.
- Lista (polje **`previous)`** je vezana za nit koja je dobila pomenute propusnice. Propusnice se uvezuju u ovu listu **u redosledu** u kome ih je nit dobila, a izvezuju iz nje **u obrnutom redosledu**, jer se u obrnutom redosledu propusnice vraćaju.
- Oko polja **`admission_list`** klase **`Permit`** se obrazuje **lista deskriptora** niti koje čekaju na propusnicu **da bi ušle u isključivi region**, a oko njenog polja **`fulfiled_list`** se obrazuje **lista deskriptora** niti koje čekaju na propusnicu **nakon ispunjenja uslova**.

Klasa Permit

- Operacije klase **`Permit`**:
  - **`not_free()`** - da li je propusnica zauzeta
  - **`take()`** - zauzimanje propusnice
  - **`release()`** - oslobadjanje propusnice
  - **`admission_insert()`**, **`admission_extract()`**, **`admission_not_empty()`**, **`fullfilled_insert()`**, **`fulfilled_extract()`**, **`fulfilled_not_empty()`** - za rukovanje listama deskriptora niti

Klasa Permit

- Pozivi operacija **`not_free()`** i **`take()`** moraju biti u **istom atomskom regionu ,** inače se može desiti da više niti jedna za drugom, proverom ustanovi da je ista propusnica slobodna i da zatim, jedna za drugom, zauzme istu propusnicu.
- Pozivi operacija za rukovanje listama deskriptora moraju biti u **atomskom regionu**.

Klasa Descriptor

```
typedef int Stack_item;

class Descriptor : private List_link {//nasledjuje se klasa List_link
        //da bi bilo moguce deskriptore niti uvezivati u liste
        protected:
                Stack_item* stack_top;   //pokazivac na vrh steka niti
                int priority;    //prioritet niti
                Permit* last;    //adresa poslednje dobijene propusnice
                unsigned tag;    //privezak deskriptora niti
        public:
                Descriptor();
                inline unsigned tag_get() const { return tag; };
                inline void tag_set(unsigned t) { tag = t; };
                inline void link_permit(Permit* const permit);//uvezivanje
 //propusnice u listu propusnica prilikom ulaska niti u kriticnu sekciju
                inline Permit* ulink_permit();//izvezivanje propusnice iz
 //liste propusnica niti prilikom njenog izlazka iz kriticne sekcije
                friend class Ready_list;
                friend class Kernel;
                friend class thread;
};
```

Klasa Descriptor

```
Descriptor::Descriptor()
{
        stack_top = 0;
        priority = 0;
        last = 0;
        tag = 0;
}

void Descriptor::link_permit(Permit* const permit)
{
        permit->previous = last;
        last = permit;
}

Permit* Descriptor::ulink_permit()
{
        Permit* permit = last;
        last = permit->previous;
        return permit;
}
```

Klasa Descriptor

- Klasa **Descriptor** određuje deskriptor niti.
- Ona nasleđuje klasu **List_link**, da bi bilo moguće deskriptore niti uvezivati u **liste**.
- Polje **stack_top** klase Descriptor sadrži pokazivač (adresu) **vrha steka niti**.
- **Prioritet** niti je sadržan u polju **priority** ove klase.
- Polje **last** klase Descriptor sadrži adresu **poslednje dobijene propusnice**.
- Polje **tag** ove klase je namenjeno za smeštanje **priveska deskriptora niti**.

Klasa Descriptor

- Klasa Descriptor nudi operacije za pristup nekim od njenih polja: **tag_get()**, **tag_set()**, kao i operacije za **uvezivanje propusnice** u listu propusnica niti prilikom njenog **ulaska** u kritični region: **link_permit()**, odnosno za **izvezivanje propusnice** iz liste propusnica niti prilikom njenog **izlaska** iz kritičnog regiona: **ulink_permit()**.

Klasa Ready_list

```
const unsigned PRIORITY_NUMBER = 32;

enum Priority {TERMINAL = -1,
                ZERO = 0,
                PR01, PR02, PR03, PR04, PR05, PR06, PR07, PR08, PR09, PR10,
                PR11, PR12, PR13, PR14, PR15, PR16, PR17, PR18, PR19, PR20,
                PR21, PR22, PR23, PR24, PR25, PR26, PR27, PR28, PR29, PR30,
                SYSTEM = 31};

class Ready_list {
        unsigned priority_bits;
        List_link ready[PRIORITY_NUMBER];
        Ready_list(const Ready_list&);
        Ready_list& operator=(const Ready_list&);
public:
        Ready_list() : priority_bits(0) {};
        int highest() const;
        void insert(Descriptor* d);
        Descriptor* extract();
        bool higher_than(Descriptor* d) const;
};
```

Klasa Ready_list

```
int Ready_list::highest() const
{
        int n = 0;
        if(priority_bits != 0)
                n = ad__get_index_of_most_significant_set_bit(priority_bits);
        return n;
}

void Ready_list::insert(Descriptor* d)
{
        if(d->priority != TERMINAL) {
                priority_bits = ad__set_bit(priority_bits, d->priority);
                ready[d->priority].insert(d);
        }
}
```

Klasa Ready_list

```
Descriptor* Ready_list::extract()
{
        Descriptor* d;
        d = (Descriptor*)(ready[highest()].extract());
        if(ready[d->priority].empty())
                priority_bits = ad__clear_bit(priority_bits, d->priority);
        return d;
}

bool Ready_list::higher_than(Descriptor* d) const
{
        return(highest() > d->priority);
}

static Ready_list ready;
```

Klasa Ready_list

- Klasa **`Ready_list`** omogućuje rukovanje spremnim nitima.
- Primer takvog rukovanja je brzo pronalaženje **najprioritetnije** niti među **spremnim** nitima, što je osnov za ispunjenje zahteva da je uvek aktivna najprioritetnija nit.
- Radi toga, svakom od **prioriteta** niti se dodeljuje **posebna** lista spremnih niti i podrazumeva se da se deskriptor spremne niti uvek **uvezuje na kraj** liste spremnih niti koja odgovara prioritetu dotične niti.
- Takođe se podrazumeva da se deskriptor spremne niti uvek **izvezuje sa početka** odabrane liste spremnih niti.
- Na ovaj način spremne niti istog prioriteta se uvek aktiviraju u redosledu u kome su postajale spremne.

Klasa Ready_list

- Sve liste spremnih niti zajedno formiraju multi-listu (**`Ready::ready`**).
- Rukovanje ovom multi-listom obuhvata:
  - Dobijanje prioriteta najprioritetnije neprazne liste spremnih niti: **`Ready::highest()`**
  - Uvezivanje deskriptora niti na kraj odgovarajuće liste spremnih niti: **`Ready::insert()`**
  - izvezivanje deskriptora niti sa početka **najprioritetnije neprazne** liste spremnih niti: **`Ready::extract()`**
  - poređenje prioriteta **najprioritetnije neprazne** liste spremnih niti sa prioritetom zadane niti: **`Ready::higher_than()`**

Klasa Ready_list

- Pošto je multi-lista niz listi spremnih niti, deskriptor spremne niti se uvezuje **na kraj** liste spremnih niti koju direktno **indeksira prioritet** ove **niti**.
- Međutim, za operaciju izvezivanja deskriptora iz najprioritetnije neprazne liste spremnih niti, potrebno je prvo pronaći **najprioritetniju nepraznu listu** spremnih niti.
- Brzo pronalaženje najprioritetnije neprazne liste spremnih niti se ostvaruje tako da se svaka lista spremnih niti reprezentuje **jednim bitom** koji sadrži 1 ako je lista neprazna, a 0 ako je lista prazna.

Klasa Ready_list

- Ove bite sadrži **`Ready::priority_bits`**, tako da se na značajnijim pozicijama nalaze biti prioritetnijih listi spremnih niti.
- Na najmanje značajnoj poziciji je bit nulte liste spremnih niti, sa prioritetom 0.
- U njoj se nalazi posebna **nulta nit**, sa prioritetom 0.
- Ona angažuje procesor **kada nema drugih spremnih niti**.
- Kada je nulta nit u stanju "spremna", tada je njen deskriptor uvezan u nultu listu spremnih niti. Nulta nit može biti još samo u stanju "aktivna".
- Ona u to stanje prelazi **kada ne postoji neka druga nit koja može da zaposli procesor**.

Klasa Ready_list

- Najniži prioritet spremnih niti 0 (ZERO) je rezervisan za **nultu nit**, a najviši prioritet spremnih niti 31 (SYSTEM) je rezervisan za **sistemske niti**.
- Između se nalaze prioriteti **korisničkih niti** (PR01, PR02, ..., PR30).
- Za **uništavane niti**, koje čekaju da budu uništene i koje više ne mogu biti spremne (a ni aktivne), uveden je poseban završni prioritet -1 (TERMINAL) koji omogućuje njihovo posebno tretiranje u okviru operacije **`Ready::insert()`**.

Klasa Atomic_region

```
class Atomic_region {
        bool flags;
        Atomic_region(const Atomic_region&);
        Atomic_region& operator=(const Atomic_region&);
public:
        Atomic_region();
        ~Atomic_region();
};

Atomic_region::Atomic_region()
{
        flags = ad__disable_interrupts();
}
Atomic_region::~Atomic_region()
{
        ad__restore_interrupts(flags);
}
```

Klasa Kernel

- Klasa **`Kernel`** omogućuje rukovanje procesorom. Rukovanje procesorom se svodi na prekljućivanje procesora sa jedne niti na drugu.
- Za prekljućivanje je neophodno imati adresu deskriptora aktivne niti sa koje se procesor prekljućuje (**`active`**), adresu deskriptora niti na koju se procesor prekljućuje (**`pretender`**), kao i adresu deskriptora niti sa koje se procesor prekljućio (**`former`**).
- Operaciju prekljućivanja poziva privatna operacija **`switch_to()`**, koja postavlja pokazivač deskriptora aktivne niti **`active`** i pokazivač deskriptora niti sa koje se procesor prekljućio **`former`**.

Klasa Kernel

- Preključivanje je nužno vezano za **raspoređivanje** (**scheduling**), odnosno za izbor niti na koju se procesor preključuje.
- Ciljevi raspoređivanja kod CppTss izvršioca su da uvek bude aktivna **najprioritetnija spremna nit** i da se **ravnomerno deli vreme** procesora između spremnih niti **istog prioriteta**. Do raspoređivanja dolazi:
  - kada se pojavi spremna nit sa višim prioritetom od aktivne niti (**`schedule()`**)
  - na kraju kvantuma (**`periodic_schedule()`**)

Klasa Kernel

- Klasa **`Kernel`** nasleđuje klasu **`Deskriptor`** da bi jedini objekat klase Kernel reprezentovao deskriptor **`main()`** niti.
- Konstruktor ove klase proglašava aktivnom **`main()`** nit. Pošto **`main()`** nit koristi stek konkurentnog programa kao svoj stek, za nju **nije potrebno zauzeti stek**.
- U nadležnosti klase **`Kernel`** nije samo preključivanje, nego i podrška **viših slojeva** iz hijerarhijske strukture CppTss izvršioca.

Klasa Kernel

- Funkcije:
  - **`make_ready()`** – omogućava aktivnost niti
  - **`expect()`** – omogućuje očekivanje dešavanja događaja
  - **`signal()`** – omogućuje objavu dešavanja događaja
  - **`exclusive_in()`** – za ulazak u isključivi region
  - **`exclusive_out()`** – za izlazak iz isključivog regiona
  - **`wait()`** – čekivanje ispunjenja uslova
  - **`notify_one()`** – objava ispunjenja uslova
- U operacijama deljene promenljive kernel se koriste **atomski regioni** radi zaštite njene konzistentnosti, kao i konzistentnosti argumenata iz poziva ovih operacija.

Klasa Kernel

```
class Kernel : public Descriptor {
          Descriptor* active;
          Descriptor* pretender;
          Descriptor* former;
          Kernel(const Kernel&);
          Kernel& operator=(const Kernel&);
          inline void switch_to(Descriptor* const d);
          inline void schedule();
          inline void periodic_schedule();
public:
          Kernel() : active(0),pretender(0), former(0)
                    { priority = PR15; active = this; };
          inline void make_ready(Descriptor* const d);
          inline void expect(List_link* const waiting_list);
          inline void signal(List_link* const waiting_list);
          inline void exclusive_in(Permit* const permit);
          inline void wait(const unsigned t,
                                                List_link* const waiting_list);
          inline void notify_one(List_link* const waiting_list);
          inline void exclusive_out();
          inline Descriptor* active_get() const { return active; };
          friend void yield();
          friend class Timer_driver;
};
```

Klasa Kernel

```
void Kernel::switch_to(Descriptor* const d)
{
        former = active;
        active = d;
        ad__stack_swap(&(former->stack_top), active->stack_top);
}

void Kernel::schedule()
{
        if(ready.higher_than(active)) {
                ready.insert(active);
                pretender = ready.extract();
                switch_to(pretender);
        }
}

void Kernel::periodic_schedule()
{
        ready.insert(active);
        pretender = ready.extract();
        if(active != pretender)
                switch_to(pretender);
}
```

Klasa Kernel

```
void Kernel::make_ready(Descriptor* const d)
{
        Atomic_region ar;
        ready.insert(d);
}

void Kernel::expect(List_link* const waiting_list)
{
        waiting_list->insert(active);
        pretender = ready.extract();
        switch_to(pretender);
}

void Kernel::signal(List_link* const waiting_list)
{
        if(waiting_list->not_empty()) {
                pretender =(Descriptor*) waiting_list->extract();
                ready.insert(pretender);
                schedule();
        }
}
```

Klasa Kernel

```
void Kernel::exclusive_in(Permit* const permit)
{
        Atomic_region ar;
        if(permit->not_free()) {
                permit->admission_insert(active);
                pretender = ready.extract();
                switch_to(pretender);
        } else {
                permit->take();
                active->link_permit(permit);
        }
}
```

Klasa Kernel

```
void Kernel::wait(const unsigned t,
                                  List_link* const waiting_list)
{
        Atomic_region ar;
        Permit* permit = active->ulink_permit();
        active->tag = t;
        if(permit->fulfilled_not_empty()) {
                pretender =
                        (Descriptor*)permit->fulfilled_extract();
                pretender->link_permit(permit);
                ready.insert(pretender);
        } else if(permit->admission_not_empty()) {
                pretender =
                        (Descriptor*)permit->admission_extract();
                pretender->link_permit(permit);
                ready.insert(pretender);
        } else
                permit->release();
        waiting_list->insert(active);
        pretender = ready.extract();
        switch_to(pretender);
}
```

Klasa Kernel

```
void Kernel::notify_one(List_link* const waiting_list)
{
        Permit* permit = active->last;
        if(permit == 0)
                throw &failure_notify_outside_exclusive_region;
        Atomic_region ar;
        if(waiting_list->not_empty()) {
                pretender =(Descriptor*) waiting_list->extract();
                permit->fulfilled_insert(pretender);
        }
}
```

Klasa Kernel

```
void Kernel::exclusive_out()
{
        Atomic_region ar;
        Permit* permit = active->ulink_permit();
        if(permit->fulfilled_not_empty()) {
                pretender = (Descriptor*)permit->fulfilled_extract();
                pretender->link_permit(permit);
                ready.insert(pretender);
        } else if(permit->admission_not_empty()) {
                pretender = (Descriptor*)permit->admission_extract();
                pretender->link_permit(permit);
                ready.insert(pretender);
        } else
                permit->release();
        schedule();
}

static Kernel kernel;

void yield()
{
        Atomic_region set_up;
        kernel.periodic_schedule();
}
```

Klasa mutex

```
class mutex : private Permit {
        mutex(const mutex&);
        mutex& operator=(const mutex&);
public:
        mutex() {};
        void lock();
        void unlock();
};

void mutex::lock()
{
        kernel.exclusive_in(this);
}

void mutex::unlock()
{
        kernel.exclusive_out();
}
```

Klasa unique_lock

```
template<class MUTEX> class unique_lock {
        unique_lock(const unique_lock&);
        unique_lock& operator=(const unique_lock&);
public:
        unique_lock(MUTEX& mx);
        ~unique_lock();
};

template<class MUTEX>
unique_lock<MUTEX>::unique_lock(MUTEX& mx)
{
        kernel.exclusive_in((Permit*)&mx);
}

template<class MUTEX>
unique_lock<MUTEX>::~unique_lock()
{
        kernel.exclusive_out();
}
```

Klasa condition_variable

```
class condition_variable {
        List_link list_head;
        List_link* position;
public:
        condition_variable() { position = &list_head; };
        void wait(unique_lock<mutex>& lock, unsigned t = 0);
        void notify_one();
        bool first(unsigned* t = 0);
        bool last();
        bool next(unsigned* t = 0);
        bool attach_tag(unsigned t);
};

void condition_variable::wait(unique_lock<mutex>& lock, unsigned t)
{
        kernel.wait(t, position);
        position = &list_head;
}
```

Klasa condition_variable

```
void condition_variable::notify_one()
{
        kernel.notify_one(&list_head);
        position = &list_head;
}

bool condition_variable::first(unsigned* t)
{
        bool r = false;
        if(list_head.not_empty()) {
                position = list_head.right_get();
                if(t != 0)
                        *t = ((Descriptor*) position)->tag_get();
                r = true;
        }
        return(r);
}
```

Klasa condition_variable

```
bool condition_variable::last()
{
        bool r = false;
        if(list_head.not_empty()) {
                position = &list_head;
                r = true;
        }
        return(r);
}

bool condition_variable::next(unsigned* t)
{
        bool r = false;
        if(position != &list_head) {
                position = position->right_get();
                if(position != &list_head) {
                        if(t != 0)
                                *t = ((Descriptor*) position)->tag_get();
                        r = true;
                }
        }
        return(r);
}
```

Klasa condition_variable

```
bool condition_variable::attach_tag(unsigned t)
{
        bool r = false;
        if(position != &list_head) {
                ((Descriptor*) position)->tag_set(t);
                r = true;
        }
        return(r);
}
```

Klasa Driver

- Klasa **`Driver`** omogućuje smeštanje adrese obrađivača prekida u tabelu prekida: **`Driver::start_interrupt_handling()`**.
- Pored toga, ova klasa uvodi definiciju klase **`Event`** koja omogućuje zaustavljanje aktivnosti niti do dešavanja događaja i objavu dešavanja događaja.

Klasa Driver

```
class Driver {
        protected:
                void start_interrupt_handling(Vector_numbers vector_number,
                                                                void (*handler)());
                class Event {
                        List_link list_head;
                public:
                        void expect();
                        void signal();
                };
};

void Driver::start_interrupt_handling(Vector_numbers vector_number,
void (*handler)())
{
        Atomic_region ar;
        ad__set_vector(vector_number, handler);
}

void Driver::Event::expect()
{
        kernel.expect(&list_head);
}
```

Klasa Driver

```
class Driver {
        protected:
                void start_interrupt_handling(Vector_numbers vector_number,void (*handler)());
                class Event {
                        List_link list_head;
                public:
                        void expect();
                        void signal();
                };
};
void Driver::start_interrupt_handling(Vector_numbers vector_number,
void (*handler)()){
        Atomic_region ar;
        ad__set_vector(vector_number, handler);
}

void Driver::Event::expect(){
        kernel.expect(&list_head);
}

void Driver::Event::signal(){
        kernel.signal(&list_head);
}
```

Klase Exception_driver i Timer_driver

- Iz klase Driver su izvedene klase **`Exception_driver`** i **`Timer_driver`**.
- Prva od njih omogućuje reakciju na pojavu hardverskih izuzetaka, radi izazivanja prevremenog kraja konkurentnog programa.
- Druga od ovih klasa omogućuje rukovanje vremenom.
- Klasa **`Exception_driver`** uvodi operaciju **`interrupt_handler()`**. Ova operacija zaustavlja izvršavanje konkurentnog programa.

Klase Exception_driver i Timer_driver

- Rukovanje vremenom obuhvata:
  - brojanje otkucaja sata, radi praćenja proticanja sistemskog vremena
  - odbrojavanje otkucaja sata preostalih do kraja kvantuma aktivne niti
  - odbrojavanja otkucaja sata preostalih do buđenja uspavane niti
- Kada broj otkucaja, preostalih do isticanja kvantuma aktivne niti, padne na nulu, potrebno je pokrenuti **periodično raspoređivanje**.

Klase Exception_driver i Timer_driver

- Takođe, kada broj otkucaja, preostalih do buđenja uspavane niti, padne na nulu, potrebno je probuditi **sve niti za koje je nastupio trenutak buđenja**.
- Svi prethodno pobrojani poslovi se nalaze u nadležnosti operacije **`interrupt_handler()`** koju uvodi klasa **`Timer_driver`**.
- Polje **`current_ticks`** ove klase sadrži sistemsko vreme, polje **`countdown`** sadrži broj otkucaja do buđenja, a polje **`rest`** broj otkucaja do isticanja kvantuma.
- Funkcija **`now()`** vraća sadržaj polja **`current_ticks`**, odnosno vraća sistemsko vreme.

Klase Exception_driver i Timer_driver

```
const unsigned long
QUANTUM = 2;

class Exception_driver : public Driver {
        static void interrupt_handler();
public:
        Exception_driver() { start_interrupt_handling(FP_EXCEPTION,
                                                      interrupt_handler); };
};

void Exception_driver::interrupt_handler()
{
        ad__report_and_finish("\nHARDWARE EXCEPTION!\n");
}

static Exception_driver exception_driver;
```

Klase Exception_driver i Timer_driver

```
class Timer_driver : public Driver {
        static unsigned long current_ticks;
        static unsigned long countdown;
        static unsigned long rest;
        static unsigned long quantum;
        static Event alarm;
        static void interrupt_handler();
public:
        Timer_driver() { start_interrupt_handling(TIMER,
                                                  interrupt_handler); };
        friend unsigned long now();
        friend class Delta;
};

unsigned long Timer_driver::current_ticks = 0;
unsigned long Timer_driver::countdown = 0;
unsigned long Timer_driver::rest = QUANTUM;
```

Klase Exception_driver i Timer_driver

```
Timer_driver::Event Timer_driver::alarm;

void Timer_driver::interrupt_handler()
{
        current_ticks++;
        if((--rest) == 0)
                rest = quantum;
        if((countdown > 0) && ((--countdown) == 0))
                alarm.signal();
        else if(rest == quantum)
                kernel.periodic_schedule();
}

static Timer_driver timer_driver;

unsigned long now()
{
        Atomic_region ar;
        return timer_driver.current_ticks;
}
```

Klasa Memory_fragment

- Klasa **`Memory_fragment`** omogućuje rukovanje slobodnom radnom memorijom. Slobodnu radnu memoriju obrazuje celi broj jedinica sastavljenih od **`UNIT`** bajta.
- Rukovanje slobodnom radnom memorijom podrazumeva da se uvek **zauzima**, odnosno da se uvek **oslobađa celi broj ovih jedinica**.
- Zauzimanja ovakvih zona slobodne radne memorije, odnosno njihova oslobađanja uzrokuju **iscepkanost** slobodne radne memorije **u odsečke**.
- Odsečci se zato uvezuju u **jednosmernu listu**, uređenu u rastućem redosledu njihovih početnih adresa.
- Radi toga, početak svakog odsečka sadrži svoju **veličinu**, izraženu u pomenutim jedinicama od po **`UNIT`** bajta, i **pokazivač narednog odsečka**.
- Veličinu odsečka i pokazivač narednog odsečka sadrže polja **`size`** i **`next`** klase **`Memory_fragment`**.

Klasa Memory_fragment

- Konstruktor klase **`Memory_fragment`** opisuje obrazovanje liste odsečaka slobodne radne memorije, sastavljene od **stalnog** odsečka čija veličina je **0** i od odsečka koji obuhvata **raspoloživu** slobodnu radnu memoriju.
- Stalnom (prvom) odsečku odgovara objekt **`memory`** klase **`Memory_fragment`**, koji je jedini objekt ove klase. Dodavanju drugog odsečka prethodi provera da li je obezbeđeno **dovoljno** radne memorije za potrebe konkurentnog programa.
- Ako nije, izvršavanje konkurentnog programa se završava uz poruku **`INITIAL MEMORY SHORTAGE`**.
- Pošto je **`UNIT`** jednak veličini stranice, uvek se zauzima toliko radne memorije da u nju može da stane traženi broj stranica, a da početak raspoložive slobodne radne memorije bude postavljen na početak prve stranice.

Klasa Memory_fragment

- Zauzimanje slobodne radne memorije omogućuje operacija **`take()`** klase **`Memory_fragment`**.
- Zauzima se jedna jedinica od **`UNIT`** bajta više nego što je traženo.
- Ona **prethodi** preostalim zauzetim jedinicama memorije i **sadrži ukupnu veličinu zauzete memorije**. Ova veličina se koristi prilikom kasnijeg oslobađanja zauzete memorije.
- Zauzimanju prethodi **pretraživanje liste odsečaka**, radi pronalaženja prvog **dovoljno velikog odsečka**.
- Pretraživanje uvek počinje od **stalnog odsečka**. Ako se pronađe dovoljno velik odsečak, traženi bajti se zauzimaju s **njegovog kraja**. Ako pronađeni odsečak obuhvata **baš traženi** broj bajta, tada se on isključuje iz liste i zauzimaju se **svi njegovi bajti**.

Klasa Memory_fragment

- Oslobađanje prethodno zauzete radne memorije omogućuje operacija **free()** klase **Memory_fragment**.
- Za oslobađanje je neophodno u listi odsečaka pronaći odsečak iza koga će se oslobađani odsečak uvezati u ovu listu.
- Pre uvezivanja proverava se da li oslobađani odsečak može da se spoji u **jedan odsečak** sa svojim **prethodnikom** i sa svojim **sledbenikom**.
- Odsečak se uvezuje u pomenutu listu samo ako ovo spajanje nije moguće.

Klasa Memory_fragment

- Prethodno opisane operacije klase **`Memory_fragment`** su namenjene za **zauzimanje** i **oslobađanje** radne memorije prilikom **stvaranja** i **uništavanja** objekata pojedinih klasa.
- Da bi se njihova namena ostvarila, neophodno je da ove operacije pozivaju globalni operatori **`new()`** i **`delete()`**.
- Ali, tada **razne niti** mogu da pozivaju operacije klase **`Memory_fragment`** posredstvom prethodna dva operatora i da tako **ugroze konzistentnost** liste odsečaka.
- Da bi se to sprečilo, ova klasa nasleđuje klasu **`mutex`** i tako omogućuje **zaključavanje** i **otključavanje** njenog jedinog **objekta memory**. To je obezbeđeno u definicijama funkcija operator **`new()`** i operator **`delete()`**, radi ostvarenja **međusobne isključivosti** različitih pristupanja listi odsečaka.

Klasa Memory_fragment

```
const size_t UNIT = PAGE_SIZE;

class Memory_fragment : public mutex {
        size_t size;
        Memory_fragment* next;
        Memory_fragment(const Memory_fragment&);
        Memory_fragment& operator=(const Memory_fragment&);
public:
        Memory_fragment();
        void* take(size_t size);
        void free(void* address);
};
```

Klasa Memory_fragment

```
Memory_fragment::Memory_fragment() : size(0), next(this)
{
        size_t free_memory = 2000 * UNIT;
        size_t beginning =(size_t) malloc(free_memory + UNIT-1);
        if(beginning == 0)
                ad__report_and_finish("INITIAL MEMORY SHORTAGE");
        else {
                beginning = (beginning + UNIT-1) & ~(UNIT-1);
                next =(Memory_fragment*) beginning;
                next->size = free_memory;
                next->next = this;
        }
}
```

Klasa Memory_fragment

```
void* Memory_fragment::take(size_t size)
{
        size += 2 * UNIT - 1;  //1 UNIT za broj zauzetih i drugi ako
        size -= size % UNIT;   //size nije ceo broj unita
        Memory_fragment* m = 0;
        Memory_fragment* p = this;
        while(p->next != this) {
                if((p->next->size) < size)
                        p = p->next;
                else if(p->next->size == size) {
                        m = p->next;
                        p->next = p->next->next;
                        break;
                } else {
                        p->next->size -= size;
                        m =(Memory_fragment*) ((size_t)(p->next) +
                                                (p->next->size));
                        break;
                }
        }
        if(m == 0)
                throw &failure_memory_shortage;
        m->size = size;
        return (void*) ((size_t)m + UNIT);
}
```

Klasa Memory_fragment

```
void Memory_fragment::free(void* address){
        if(address != 0) {
                Memory_fragment* a = (Memory_fragment*)((size_t)address - UNIT);
                Memory_fragment* p = this;
                while(p->next != this)
                        if(a > p->next)
                                p = p->next;
                        else
                                break;
                if((((size_t) p) + (p->size)) == ((size_t) a)) {
                        p->size += a->size;
                        if((((size_t) p) + (p->size)) == ((size_t)(p->next))) {
                                p->size += p->next->size;
                                p->next = p->next->next;
                        }
                } else if((((size_t) a) + (a->size)) == ((size_t)(p->next))) {
                        a->size += p->next->size;
                        a->next = p->next->next;
                        p->next = a;
                } else {
                        a->next = p->next;
                        p->next = a;
                }
        }
}
```

Klasa Memory_fragment

```
static Memory_fragment memory;

void* operator new(size_t size)
{
          void* memory_block;
          memory.lock();
          try {
                    memory_block = memory.take(size);
          }
          catch(...) {
                    memory.unlock();
                    throw;
          }
          memory.unlock();
          return memory_block;
}

void operator delete(void* address)
{
          memory.lock();
          memory.free(address);
          memory.unlock();
}
```

Klase Thread_image i thread

- Rukovanje nitima omogućuju klase **`Thread_image`** i **`thread`**, kao i definicije funkcija **`thread_destroyer_deamon()`**, **`destroy()`** i **`undetached_threads()`**.
- Klasa **`Thread_image`** opisuje **sliku** niti, sastavljenu od:
  - **deskriptora** niti
  - **uslova** (**`ended`**) koji omogućuje **čekanje završetka aktivnosti** niti
  - **oznake** da regularan kraj aktivnosti niti može da nastupi kao posledica **kraja aktivnosti procesa kome dotična nit pripada** (**`detached`**)
  - **steka** niti

Klase Thread_image i thread

- Klasa **`thread`** omogućuje:
  - međusobnu isključivost svojih operacija (**`mx`**)
  - **brojanje niti** za koje koje nije regularno da kraj njihove aktivnosti nastupi kao **posledica kraja aktivnosti procesa kome dotične niti pripadaju** (**`undetached_threads_number`**)
  - uništavanje niti (**`termination`**, **`terminating`**)
  - pristup slici niti (**`ti`**)

- Konstruktor klase **`thread`** omogućuje **kreiranje niti**. U toku kreiranja niti pripremi se njen **stek** za **preključivanje**, da bi automatski započelo izvršavanje **funkcije koja opisuje ponašanje niti** nakon prvog preključivanja na nit.
- Završetak aktivnosti niti se otkriva u okviru operacija **`join()`** i **`detach()`** na osnovu završnog prioriteta niti (**`TERMINAL`**).

Klase Thread_image i thread

- Na završetku aktivnosti niti, u toku izvršavanja funkcije **`destroy()`**, omogućuje se nastavak aktivnosti niti koja čeka dotični završetak i oslobađanje prostora koga zauzima **slika niti**, što je u nadležnosti sistemske niti **`thread_destroyer_deamon()`**.
- Funkcija **`undetached_threads()`** omogućuje proveru da li postoje niti za koje nije regularno da kraj njihove aktivnosti nastupi kao posledica kraja aktivnosti procesa kome dotične niti pripadaju, da bi se na kraju aktivnosti procesa ukazalo na **prevremeni** završetak ovakvih niti.

Klase Thread_image i thread

```
const unsigned DEFAULT_STACK_SIZE = 4096;

class Thread_image: public Descriptor {
          condition_variable ended;
          bool detached;
          Stack_item stack[DEFAULT_STACK_SIZE];
          Thread_image (const Thread_image &);
          Thread_image & operator=(const Thread_image &);
public:
          Thread_image(void (*thread_function)(), Priority p);
          friend class thread;
          friend void destroy();
};

Thread_image::Thread_image(void (*thread_function)(), Priority p) :
          detached(false)
{
          stack_top = &(stack[DEFAULT_STACK_SIZE]);
          ad__stack_init(&stack_top, (unsigned)thread_function);
          priority = p;
}
```

Klase Thread_image i thread

```
class thread {
          static mutex mx;
          static unsigned undetached_threads_number;
          static condition_variable termination;
          static Thread_image* terminating;
          Thread_image* ti;
          thread (const thread &);
          thread & operator=(const thread &);
public:
          thread(void (*thread_function)(), Priority p = PR15);
          void join();
          void detach();
          friend void thread_destroyer_deamon();
          friend void destroy();
          friend bool undetached_threads();
};

mutex thread::mx;

unsigned thread::undetached_threads_number = 0;

condition_variable thread::termination;

Thread_image* thread::terminating;
```

Klase Thread_image i thread

```
thread::thread(void (*thread_function)(), Priority p)
{
          ti = new Thread_image(thread_function, p);
          unique_lock<mutex> lock(mx);
          undetached_threads_number++;
          kernel.make_ready(ti);
}

void thread::join()
{
          unique_lock<mutex> lock(mx);
          if(ti->priority != TERMINAL)
                    ti->ended.wait(lock);
}

void thread::detach()
{
          unique_lock<mutex> lock(mx);
          if((ti->priority != TERMINAL) && (!ti->detached)) {
                    ti->detached = true;
                    undetached_threads_number--;
          }
}
```

Klase Thread_image i thread

```
void thread_destroyer_deamon()
{
          for(;;) {
                    unique_lock<mutex> lock(thread::mx);
                    thread::termination.wait(lock);
                    delete thread::terminating;
          }
}

void destroy()
{
          unique_lock<mutex> lock(thread::mx);
          thread::terminating = (Thread_image*)kernel.active_get();
          while(thread::terminating->ended.last())
                    thread::terminating->ended.notify_one();
          if(!thread::terminating->detached)
                    thread::undetached_threads_number--;
          thread::terminating->priority = TERMINAL;
          thread::termination.notify_one();
}

bool undetached_threads()
{
          return (thread::undetached_threads_number > 0);
}
```

Klasa Delta

- Klasa **`Delta`** i funkcije **`thread_wake_up_deamon()`**, **`sleep_for()`** i **`sleep_until()`** zajedno omogućuju uspavljivanje i buđenje niti, a funkcija **`thread_zero()`** opisuje aktivnost **nulte** (**sistemske**) niti.
  - **`sleep_for()`** omogućuje uspavljivanje aktivne niti **dok ne protekne zadani broj otkucaja sata**
  - **`sleep_until()`** omogućuje uspavljivanje aktivne niti **dok ne nastupi zadani trenutak** sistemskog vremena.

Klasa Delta

- Oko polja list klase **`Delta`** se formira lista **deskriptora** uspavanih niti.
- Da se za svaku uspavanu nit ne bi proveravalo, nakon svakog otkucaja, da li je nastupilo vreme njenog buđenja, deskriptori uspavanih niti se uvezuju u listu u **hronološkom redosledu buđenja** niti.
- Svakom od ovih deskriptora je dodeljen **privezak** koji pokazuje **relativno vreme buđenja** (relativni broj otkucaja do buđenja) u odnosu na **prethodnika u listi**.
- Ovakva lista se zove **delta lista**.
- Zahvaljujući delta listi, nakon svakog otkucaja potrebno je proveriti da li je nastupio trenutak buđenja **samo za nit koja se najranije budi**, odnosno samo za **prvi deskriptor** iz delta liste.
- Pošto može da bude **više niti**, čije buđenje je vezano za isti trenutak, unapred nije poznato koliko niti treba probuditi nakon otkucaja sata.

Klasa Delta

- Operaciju **`awake()`** klase **`Delta`** poziva sistemska nit **`Wake_up_daemon()`**. Vreme njenog buđenja je uvek jednako **najranijem vremenu buđenja korisničkih niti**.
- Nakon buđenja, **sistemska nit budi sve korisničke niti sa početka `delta` liste**, za koje je nastupio trenutak buđenja.
- Čekanje buđenja omogućuje poziv operacije **`Timer_driver::alarm.expect()`**.

Klasa Delta

- Dužinu čekanja određuje vrednost lokalne promenljive **`tag`**, kada ima uspavanih korisničkih niti (na čije prisustvo ukazuje vrednost lokalne promenljive **`sleeping`**).
- Dužina čekanja se skraćuje za vrednost lokalne promenljive **`passed_ticks`**, koja registruje vreme proteklo na buđenju korisničkih niti.

Klasa Delta

- Operaciju **`sleep()`** klase **`Delta`** poziva, posredstvom funkcije **`sleep_for()`**, korisnička nit, da bi se njen deskriptor uključio u **delta listu**, a njena aktivnost privremeno zaustavila.
- Konstruktor klase **`Delta`** omogućuje kreiranje sistemskih niti korišćenjem **bezimenih** (**privremenih**) objekata klase **`thread`**.
- Konzistentnost delta liste štite isključivi regioni u telima operacija **`awake()`** i **`sleep()`** klase **`Delta`**.

Klasa Delta

```
void thread_zero();

void thread_wake_up_deamon();

typedef unsigned long milliseconds;

class Delta {
          mutex mx;
          condition_variable list;
          inline void sleep(milliseconds duration);
          inline void awake();
          Delta (const Delta &);
          Delta & operator=(const Delta &);
public:
          Delta();
          friend void thread_wake_up_deamon();
          friend void sleep_for(milliseconds duration);
};

Delta::Delta()
{
          thread (thread_zero, ZERO).detach();
          thread (thread_destroyer_deamon, SYSTEM).detach();
          thread (thread_wake_up_deamon, SYSTEM).detach();
}
```

Klasa Delta

```
void Delta::awake()
{
        bool sleeping = false;
        unsigned long begining_moment = 0;
        unsigned long passed_ticks = 0;
        unsigned tag = 0;
        for(;;) {
                { Atomic_region ar;
                  if(!sleeping)  //ako nema nikoga u spavanju 1 deo
                          //ceka signal od Timer_driver IH
                          Timer_driver::alarm.expect();
                  else {    //provera da li treba buditi jos neki 3 deo
                          passed_ticks =
                                  Timer_driver::current_ticks - begining_moment;
                          if(tag > passed_ticks) {
                                  //postavljanje novog countdowna za timer drajver IH
                                  Timer_driver::countdown = tag - passed_ticks;
                                  Timer_driver::alarm.expect();
                          }
                  }
                  begining_moment = Timer_driver::current_ticks;
                }
                { unique_lock<mutex> lock(mx); //notifikacija dogod ima 2 deo
                  do {
                          passed_ticks = passed_ticks - tag;
                          list.notify_one();
                          sleeping = list.first(&tag);
                } while(sleeping && (tag <= passed_ticks));
            }
    }
}
```

Klasa Delta

```
void Delta::sleep(milliseconds duration)
{
        unsigned new_tag = duration;
        unsigned old_tag;
        {
                Atomic_region ar;
                old_tag = Timer_driver::countdown;
        }
        unique_lock<mutex> lock(mx);
        if(list.first())   //ako ima nesto u delta listi
                do {
                        if(old_tag > duration) { //ako je prvi u listi
                                list.attach_tag(old_tag - duration);
                                break;
                        } else if(old_tag == duration) { //a
                                duration = 0;
                                list.next();
                                break;
                        } else
                                duration -= old_tag;
                } while(list.next(&old_tag));
        if(duration == new_tag) {
                Atomic_region ar;
                Timer_driver::countdown = duration;
        }
        list.wait(lock, duration);
}
```

Klasa Delta

```
static Delta delta;

void thread_zero()
{
          for(;;)
                    ;
}

void thread_wake_up_deamon()
{
          delta.awake();
}

void sleep_for(milliseconds duration)
{
          if(duration > 0)
                    delta.sleep(duration);
}

void sleep_until(milliseconds moment)
{
          milliseconds difference = moment - now();
          if(difference <= (ULONG_MAX / 2))     //maksimalna vrednost longa
                    sleep_for(difference);
}
```