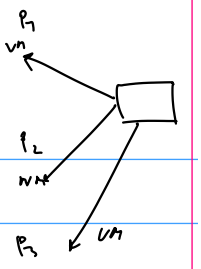


3. (10 poena)

Više procesa proizvođača i potrošača međusobno razmenjuju znakove (`char`) preko ograničenog bafera tako što svi ti procesi dele jedan zajednički logički segment memorije veličine `sizeof(bbuffer)` koji su alocirali sistemskim pozivom *mmap* kao *bss* segment (inicijalizovan nulama pri alokaciji). Svaki od tih procesa adresu tog segmenta konvertuje u pokazivač na tip `struct bbuffer` i dalje radi operacije sa ograničenim baferom pozivajući sledeće operacije iz biblioteke `bbuf` čije su deklaracije (`bbuf.h`) date dole:

- `bbuf_init`: svaki proces koji želi da koristi bafer mora najpre da pozove ovu operaciju kako bi ona otvorila potrebne semafore; ukoliko neki od sistemskih poziva vezanih za semafore nije uspeo, sve one već otvorene semafore treba osloboditi i vratiti negativnu vrednost (greška); ukoliko je inicijalizacija uspela, treba vratiti 0;
- `bbuf_close`: svaki proces koji koristi bafer treba da pozove ovu operaciju kada završi sa korišćenjem bafera;
- `bbuf_append`, `bbuf_take`: operacije stavljanja i uzimanja elementa (`char`) u bafer.

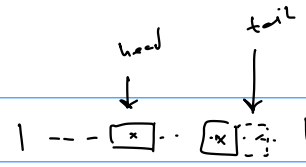
Sve ove operacije podrazumevaju da je argument ispravan pokazivač (ne treba ga proveravati). Sve operacije osim `bbuf_init` takođe podrazumevaju da je inicijalizacija pre toga uspešno završena – proces ih ne sme pozivati ako nije, pa ne treba proveravati ispravnost stanja bafera.

Implementirati biblioteku `bbuf` (`bbuf.cc`): dati definiciju strukture `bbuffer` i implementirati sve ove operacije korišćenjem POSIX semafora.

```
struct bbuffer;
int  bbuf_init (struct bbuffer*);
void bbuf_append (struct bbuffer*, char);
char bbuf_take (struct bbuffer*);
void bbuf_close (struct bbuffer*);
```

sem_open

[head, tail)




```
struct GCutter {
    char buff[BUFSIZE];
    int head, tail;
};

sem_t *slots, *items, *mtx;
int   buf_init (struct *cutter) {
        cutter->head = 0;
        cutter->tail = 0;
        slots = sem_open ("/cp_slots", O_CREAT, O_RDWR, BUFSIZE);
        if (slots == NULL)  return -1;
        items = sem_open ("/cp_items", O_CREAT, O_RDWR, 0);
        if (items == NULL) {
                sem_close(slots);
                return -1;
        }
        mtx = sem_open ("/cp_mtx", O_CREAT, O_RDWR, 1);
        if (mtx == NULL) {
                sem_close (items);
                sem_close (slots);
                return -1;
        }
}
```

pro. slots(BUFSIZE)

pot. items(0)

mtx(1)

```
void buff_append (struct cbuffer *buf, char c) {
        sem_wait (slots);
        sem_wait (mtx);
        buf->buff [buf->tail] = c;
        buf->tail = (buf->tail+1) % BUFSIZE;
        sem_post (mtx);
        sem_post (items);
}

char bBuf_take (struct bbuffer *buf) {
        sem_wait (items);
        sem_wait (mtx);
        char c = buf->buff [buf->head];
        buf->head = (buf->head+1) % BUFSIZE;
        sem_post (mtx);
        sem_post (slots);
        return c;
};

void bBuf_close (struct bbuffer *buf) {
        sem_close (items);
        sem_close (slots);
        sem_close (mtx);
}
```

3. (10 poena)

Pravi se softver za neki mali specijalizovani ugrađeni (*embedded*) računar sa malo RAM memorije i jednostavnim jezgrom operativnog sistema koje podržava POSIX niti i semafore. Ovaj računar treba da obrađuje podatke tipa `char` koje učitava sa nekog ulaznog uređaja kao tok znakova, a koji neprekidno stižu na ulazu, da ih obrađuje tako što transformiše svaki učitani znak u po jedan transformisan znak i da tako dobijene znakove isto tako, kao tok šalje na izlazni uređaj. Taj posao treba da obavlja bez prestanka.

Učitavanje podataka sa ulaznog uređaja radi funkcija `read_data` koja može da učita niz znakova zadate dužine na zadato mesto (u zadati bafer). Ova funkcija je napravljena i na raspolaganju je. Ona ulaznu operaciju obavlja prozivanjem uređaja (*polling*), uz mnogo uposlenog čekanja, a po potrebi uspavljuje pozivajuću nit na neko vreme ako ulazni podaci nisu duže raspoloživi jer je ulazni uređaj spor. Isto tako radi i data funkcija `write_data` koja prenosi niz znakova iz datog izlaznog bafera zadate veličine na izlazni uređaj. Obradu jednog niza znakova radi napravljena funkcija `process_data` koja te znakove čita iz datog ulaznog bafera i transformisani niz znakova iste dužine upisuje u dati izlazni bafer.

Zbog sporosti ulaznog i izlaznog uređaja i mnogo čekanja na njih, opisanu obradu treba organizovati tako da se opisane tri funkcije izvršavaju u tri uporedne niti koje sarađuju poput cevovoda (*pipeline*) razmenjujući podatke preko dva bafera, ulaznog i izlaznog, svaki veličine 256 znakova, a sinhronizuju se pomoću semafora, s tim da se ulazne i izlazne niti izvršavaju uporedo: dok ulazna nit učitava podatke u ulazni bafer, izlazna nit može da šalje prethodno obrađeni niz podataka iz izlaznog bafera na izlazni uređaj. Na raspolaganju su te funkcije:

```
extern void read_data (char* buffer, size_t size);
extern void write_data (const char* buffer, size_t size);
extern void process_data (const char* in_buf, char* out_buf, size_t size);
```

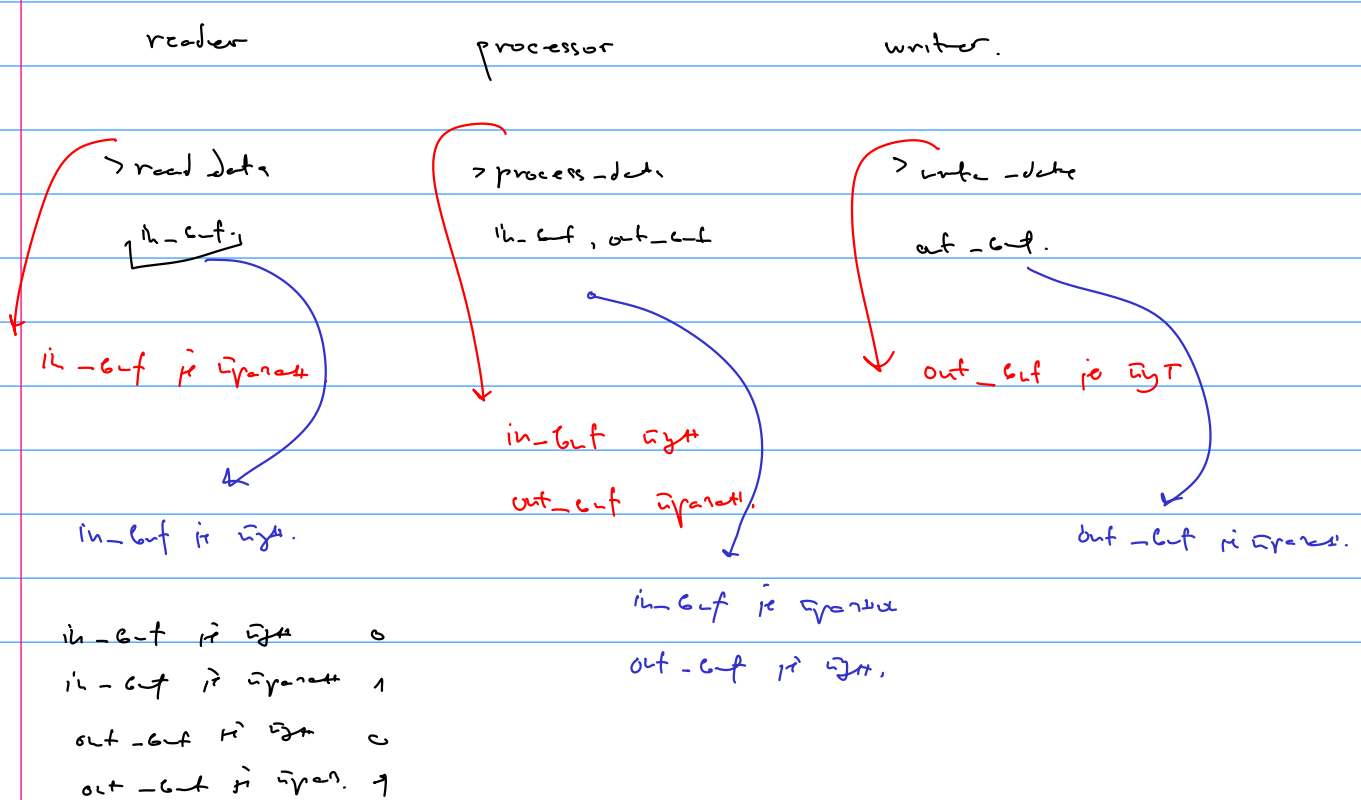
Napisati kompletan C/C++ program, sa svim potrebnim deklaracijama i funkcijom `main` koji obavlja opisani prenos i obradu. Ignorisati greške. Podsetnik na potpise POSIX funkcija vezanih za niti i semafore:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void*(*thread_body)(void*), void *arg);
sem_t *sem_open(const char *name,int oflag,mode_t mode,unsigned int value);
int sem_wait(sem_t *sem); int sem_post(sem_t *sem);
```

Rešenje:

char in_buf [256];

char out_buf [256];



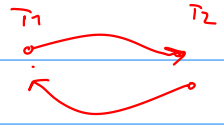
in_buf je ispunjen 0

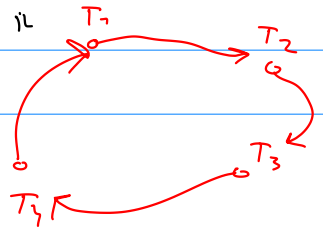
in_buf je ispunjen 1

out_buf je ispunjen 0

out_buf je ispunjen. 1

```
#define SIZE 256
char in_buf[SIZE], out_buf[SIZE];

sem_t *in_buf_full, *in_buf_empty;
sem_t *out_buf_full, *out_buf_empty;

void *reader(void *arg){
    while(1) {
        sem_wait(in_buf_empty);
        read_data(in_buf, SIZE);
        sem_post(in_buf_full);
    }
}

void *writer(void *arg) {
    while(1) {
        sem_wait(out_buf_full);
        write_data(out_buf, SIZE);
        sem_post(out_buf_empty);
    }
}

void *processor(void *arg){
    while(1){
        sem_wait(in_buf_full);
        sem_wait(out_buf_empty);
        process_data(in_buf, out_buf, SIZE);
        sem_post(in_buf_empty);
        sem_post(out_buf_full);
    }
}
```





mtx1 = semaphor(1)

mtx2 = sem (1).

1 { wait(mtx1) wait(mtx2)

2 { wait(mtx1) wait(mtx2)

deadlock

exclusive lock

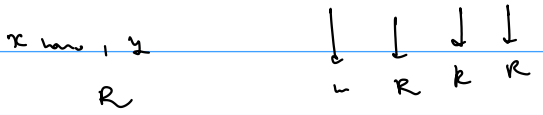
```
int main() {
    in_out_empty = sem_open("/in_out_empty", O_CREAT, O_RDWR, 1);
    in_out_full  = sem_open("/in_out_full",  O_CREAT, O_RDWR, 0);
    out_out_empty = sem_open("/out_out_empty", O_CREAT, O_RDWR, 1);
    out_out_full  = sem_open("/out_out_full",  O_CREAT, O_RDWR, 0);

    pthread_t r, w, p;
    pthread_create(&r, NULL, reader, NULL);
    pthread_create(&w, NULL, writer, NULL);
    pthread_create(&p, NULL, processor, NULL);

    pthread_join(r, NULL);
    pthread_join(w, NULL);
    pthread_join(p, NULL);
}
```

3. (10 poena)

Više uporednih niti-pisaca upisuje izračunate celobrojne dvodimenzionalne koordinate (x, y), ~~na koje treba pomeriti robota~~, u deljeni objekat klase `SharedCoord` čiji je interfejs dat dole; svaka ovakva nit nezavisno upisuje svoj par izračunatih koordinata pozivom operacije `write` ove klase. Jedna nit-čitalac periodično očitava par koordinata iz tog deljenog objekta i pomera robota na očitane koordinate; ova nit to radi pozivom operacije `read` ove klase, nezavisno od pisaca, svojim tempom, tako da svaki put čita poslednje upisane koordinate (može pročitati više puta isti par koordinata ili neke izračunate koordinate i preskočiti).

Implementirati klasu `SharedCoord` uz neophodnu sinhronizaciju korišćenjem najmanjeg broja semafora školskog jezgra.

```
class SharedCoord {
public:
  SharedCoord ();
  void read (int& x, int& y);
  void write (int x, int y);
};
```



class SharedCoord {

int m_x, m_y;

Semaphore mtx(1);

void read (int &x, int &y){ → [ne smeju se write]

mtx.wait() → ...

x = m_x;

y = m_y;

mtx.signal().

}

void write (int x, int y){ → ne smeju se read.

mtx.wait() → ne smeju write

m_x = x;

m_y = y;

mtx.signal().

}

}

2. (10 poena)

U školsko jezgro dodaje je se koncept *uslova* (engl. *condition*) za uslovnu sinhronizaciju, podržan klasom `Condition` čiji je interfejs dat dole. Uslov može biti ispunjen ili neispunjen; inicijalna vrednost zadaje se konstruktorom. Niti koje smeju da nastave izvršavanje iza neke tačke samo ako je uslov ispunjen treba da pozovu operaciju `wait`, koja ih suspenduje ako uslov nije ispunjen. Bilo koja nit koja ispuni uslov to objavljuje pozivom operacije `set`; sve niti koje čekaju na taj uslov tada nastavljaju izvršavanje. Kada neka nit promeni uslov tako da on više nije ispunjen, treba da pozove operaciju `clear`. Implementirati klasu `Condition`.

```
class Condition {
public:
  Condition (bool init = false);
  void set ();
  void clear ();
  void wait ();
};
```

Rešenje:

```
class Condition {

    bool cond;

    Condition (bool init = false) : cond(init) { }

    Queue blocked;

    void wait() {
        if (!cond) {
            blocked.put(Thread::running);
            Thread *old = Thread::running;
            Thread *new = Scheduler::get();
            Thread::running = new;
            yield(old->ctx, new->ctx);
        }
    }

    void set() {
        cond = true;
        while (!blocked.empty()) {
            Scheduler::put(blocked.get());
        }
    }

    void clear() {
        cond = false;
    }

};
```

1. (10 poena)

U školskom jezgru modifikuje se (uopštava) koncept brojačkog semafora prodržan klasom `Semaphore` na sledeći način. Operacija `wait()` prima jedan nenegativan celobrojni argument *n* sa podrazumevanom vrednošću 1 i sa sledećim značenjem. Trenutna nenegativna vrednost semafora *v* predstavlja broj raspoloživih „žetona“. Operacijom *wait* pozivajuća nit „traži“ *n* „žetona“ (kod standardnog brojačkog semafora *n* je uvek podrazumevano 1). Ako je trenutna vrednost *v* semafora veća ili jednaka argumentu *n* operacije *wait*, ta vrednost *v* će biti umanjena za *n*, a nit će nastaviti izvršavanje bez blokade, jer je „dobila“ svih traženih *n* „žetona“. U suprotnom, ova nit će „uzeti“ *v* „žetona“, i čekaće blokirana na semaforu dok se operacijama *signal* ne pojavi još *n-v* žetona; kada se to dogodi, nit može nastaviti sa izvršavanjem (jer je dobila svih traženih *n* „žetona“). Operacija *signal* „obezbeđuje“ uvek jedan „žeton“, kao i kod standardnog brojačkog semafora.

```
class Semaphore {
public:
  Semaphore (unsigned int init=1);
  void wait(unsigned int n=1);
  void signal();
  int val ();
};
```

Za podršku ovoj implementaciji, u klasi `Thread` postoji nenegativno celobrojno polje `waiting` koje pokazuje na koliko još preostalih „žetona“ čeka data nit, ukoliko je blokirana na nekom semaforu. Osim toga, klasa `Queue` kojom se implementira FIFO red čekanja na semaforu ima operaciju `first()` koja vraća prvu nit u tom redu, ako je ima (0 ako je red prazan), bez izbacivanja te niti iz reda. Pomoćne operacije `block()` i `deblock()` klase `Semaphore` su iste kao i u postojećoj implementaciji školskog jezgra.

Dati izmenjenu implementaciju operacija `wait()` i `signal()`.

```
void Semaphore::wait(int n) {
        lock();
        if (n > v) {
                v -= n;
        } else {
                Thread::running->waiting = n - v;
                v = 0;
                block();
        }
        unlock();
}

void Semaphore::signal() {
        lock();
        if (blocked.first()) {
                blocked.first()->waiting -= 1;
                if (blocked.first()->waiting == 0) deblock();
        } else {
                v += 1;
        }
        unlock();
}
```

1. (10 poena)

Školsko jezgro proširuje se konceptom *mutex* koji predstavlja binarni semafor, poput događaja (*event*), sa istom semantikom operacija *wait* i *signal* kao kod događaja, ali sa sledećim dodatnim ograničenjima koja su u skladu sa namenom upotrebe samo za međusobno isključenje kritične sekcije:

- inicijalna vrednost je uvek 1;
- operaciju *signal* može da pozove samo nit koja je zatvorila ulaz u kritičnu sekciju, odnosno koja je pozvala operaciju *wait*; u suprotnom, operacija *signal* vraća grešku;
- nit koja je već zatvorila *mutex* operacijom *wait*, ne može ponovo izvršiti *wait* na njemu.

Operacije *wait* i *signal* vraćaju celobrojnu vrednost, 0 u slučaju uspeha, negativnu vrednost u slučaju greške. Prikazati implementaciju klase `Mutex`, po uzoru na klasu `Semaphore` prikazanu na predavanjima (ne treba implementirati red čekanja niti, pretpostaviti da ta klasa postoji).

Rešenje:

```
class Mutex {
    Thread * owner = nullptr;
    Queue blocked;
    void wait() {
        int ret = 0;
        lock();
        if (owner == nullptr) {
            owner = Thread::running;
        } else if (owner == Thread::running) {
            ret = 1;
        } else {
            block();
        }
        unlock();
        return ret;
    }
    void block() {
        Thread *old = Thread::running;
        Thread *new = Scheduler::get();
        Thread::running = new;
        if (setjmp(old->ctx) == 0) {
            longjmp(new->ctx, 1);
        }
    }
```

```
void signal() {
    int ret;
    lock();
    if (Thread::running == owner) {
        owner = nullptr;
        deblock();
        ret = 0;
    } else {
        ret = -1;
    }
    unlock()
    return ret;
}

void deblock() {
    if (!blocked.empty()) {
        Scheduler::put(blocked.get());
    }
}
```

1. (10 poena)

Školsko jezgro treba proširiti podrškom za slanje i prijem poruka između niti, implementacijom sledeće dve operacije klase `Thread`:

- `void Thread::send(char* message)`: pozivajuća nit šalje poruku datoj niti (`this`); ukoliko je ovoj niti već stigla neka poruka koju ona nije preuzela, pozivajuća nit se suspenduje dok se prethodna poruka ne preuzme i tek onda ostavlja poruku i nastavlja izvršavanje;
- `static char* Thread::receive()`: pozivajuća nit preuzima poruku koja joj je poslata; ukoliko poruke nema, pozivajuća nit se suspenduje dok poruka ne stigne.

Rešenje:

```
class Thread {
    ...
    bool waiting = false;
    char *msg = null;
    Queue blocked;
    int lck = 0;
void Thread::send(char *message) {
    lock(lck);
    if (msg == nullptr) {
        msg = message;
```

message_full (0)

message_empty (1)

```
send(m) {
    message_empt.wait();
    msg = m.
    mess_full.signal();
}

receive() {
    ...
    running->message_full.wait
```

```
            if (waiting) {
                Scheduler::put(this);
            }
        } else {
            Blocked.put(running);
            Thread *new = Scheduler::get();
            Thread *old = running;
            running = new;
            if (setjmp(old->ctx) == 0) {
                longjmp(new->ctx, 1);
            } else {
                msg = message;
            }
        }
    }

    static char *receive() {
        char *m;
        lock(running->lock);
        if (running->msg == null) {
            running->waiting = true;
            Thread *new = Scheduler::get();
            Thread *old = running;
            running = new;
            if (setjmp(old->ctx) == 0) {
                unlock(running->lock);
                longjmp(new->ctx, 1);
            }
            lock(running->lock);
        }
        if (!Blocked.empty()) {
            Scheduler::get(Blocked.get());
        }
        m = running->msg;
        running->msg = nullptr;
        unlock(running->lock);
        return m;
    }
```

m = msg
msg = null
running->message->empty.signal
return m;
}

2. (10 poena)

Data je bibliotečna funkcija namenjena podršci optimističkom pristupu međusobnom isključenju bez eksplicitnog zaključavanja:

```
int cmpxchg(void** ptr, void* oldValue, void* newValue);
```

Ova funkcija kao prvi argument (`ptr`) prima adresu lokacije u kojoj se nalazi neki pokazivač bilo kog tipa (`void*`). Ona atomično poredi vrednost pokazivača koji je dostavljen kao drugi argument (`oldValue`) sa vrednošću na lokaciji na koju ukazuje `ptr`, i ako su te vrednosti iste, u lokaciju na koju ukazuje `ptr` upisuje vrednost datu trećim argumentom (`newValue`) i vraća 1; u suprotnom, ako su ove vrednosti različite, ne radi ništa, već samo vraća 0. Atomičnost je obezbeđena implementacijom pomoću odgovarajuće mašinske instrukcije.

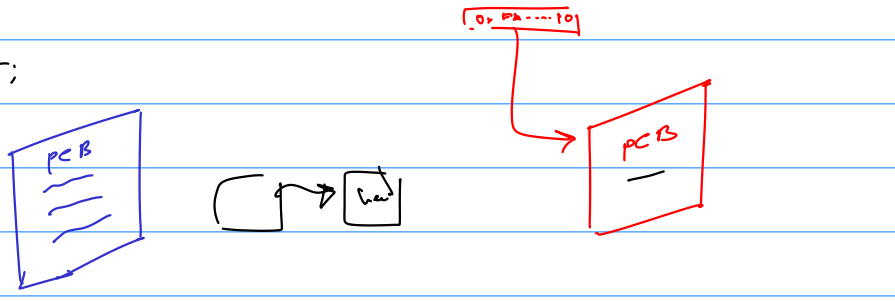
Struktura `Record` predstavlja zapis (jedan element) jednostruko ulančane liste. U toj strukturi polje `next` ukazuje na sledeći element u listi.

Korišćenjem date operacije `cmpxchg()` implementirati funkciju:

```
void insert (Record** head, Record* e);
```

Ova funkcija prima argument koji predstavlja adresu pokazivača na prvi element liste (adresu lokacije u kojoj je glava liste), a kao drugi argument dobija pokazivač na novi element u listi koga treba da umetne na početak date liste. Lista je deljena između više procesa, pa ova funkcija treba da bude sigurna za uporedne pozive iz više procesa, s tim da međusobno isključenje treba obezbediti optimističkom strategijom bez eksplicitnog zaključavanja. Zapis na koga ukazuje drugi element (e) je privatan samo za pozivajući proces (drugi procesi mu ne pristupaju pre umetanja u listu).

```
struct Record {
    ;
    Record * next;
};
```




```
*head -> Rec

void insert (Record **head, Record *e) {
    Record * old_head;
    do {
        old_head = *head;
        e -> next = old_head;
    while (0 == cmpxchg (head, old_head, e));
}
```

3. (10 poena)

Klasa `Data` predstavlja neku strukturu podataka i poseduje konstruktor kopije. Objekti ove klase mogu se praviti operatorom `new` i brisati operatorom `delete`. Pokazivač `sharedData` je deljen između uporednih niti i ukazuje na deljeni objekat ove klase. Klasa `OptimisticCCtrl` implementira optimistički pristup kontroli konkurentnosti nad objektom na koga ukazuje deljeni pokazivač i namenjena je da se koristi na dole dat način. Svaki pokušaj transakcije izmene deljenog objekta mora da se započne pozivom operacije `startTrans` kojoj se dostavlja adresa pokazivača na deljeni objekat klase `Data`. Ova operacija vraća pokazivač na kopiju objekta nad kojim transakcija može da radi izmene (upis). Na kraju treba pozvati operaciju `commit` koja vraća `true` ako je uspela, `false` ako je detektovan konflikt; u ovom drugom slučaju transakcija mora da se pokuša iznova sve dok konačno ne uspe.

```
Data* sharedData = ...
OptimisticCCtrl* ctrl = new OptimisticCCtrl();

bool committed = false;
do {
  Data* myCopy = ctrl->startTrans(&sharedData);
  myCopy->write(...); // Write to *myCopy
  committed = ctrl->commit();
while (!committed);
```

Implementirati klasu `OptimisticCCtrl` čiji je interfejs dat dole. Na raspolaganju je sistemska funkcija `cmp_and_swap` koja radi atomičnu proveru jednakosti vrednosti pokazivača `*shared` i `read` i ako su oni isti, u `*shared` upisuje vrednost `copy`.

```
class OptimisticCCtrl {
public:
  OptimisticCCtrl ();

  Data* startTrans (Data** shared);
  bool  commit ();

};

bool cmp_and_swap(void** shared, void* read, void* copy);
```

```
Data * startTrans (Data ** shared) {
        this → shared = shared;
        this → org = *shared;
        this → copy = new Data (*(this → org));
        return this → copy;
}.

bool commit() {
        Bool ret = cmp_and_swap (this → shared, this → org, this → copy);
        if (! ret) {
                delete this → copy;
        }
        return ret;
}.
```