Ulazno/Izlazni Podsistem (IO)

Zad 1. Jun 2005.

Potrebno je realizovati drajver (*device driver*) za jedan izlazni, znakovno orijentisani uređaj, pri čemu korisnički proces može zadati prenos čitavog niza znakova koje će DMA upisivati na uređaj znak po znak. Korisnički proces koji zadaje izlaznu operaciju svoj zahtev predstavlja sledećom strukturom:

```
struct DeviceRequest {
  char* block; // Pokazivač na niz znakova koje treba preneti na uređaj
  unsigned long size; // Veličina niza znakova
  Semaphore* toSignal; // Semafor koji treba signalizirati po završetku op.
};
```

Korisnički proces svoj zahtev zapisan u ovakvoj strukturi upisuje u ograničeni bafer koji predstavlja red postavljenih zahteva, a potom se blokira na semaforu na koji ukazuje pokazivač `toSignal`. Semafor je standardni, brojački semafor sa uobičajenim operacijama `signal` i `wait`. Ograničenom baferu pristupa se sledećim operacijama koje su blokirajuće ukoliko je bafer pun, odnosno prazan:

```
void putDeviceRequest (DeviceRequest*); // Stavlja jedan zahtev u red
DeviceRequest* getDeviceRequest();  // Uzima jedan zahtev iz reda
```

Sama izlazna operacija obavlja se preko DMA uređaja kome se prenos niza znakova `block` veličine `size` iz memorije na uređaj zadaje i pokreće sledećom neblokirajućom operacijom:

```
void DMARequest (char* block, unsigned long size);
```

Kada završi prenos bloka podataka, DMA generiše prekid kome pripada ulaz 10h u vektor tabeli.
Korišćenjem navedenih operacija (za koje se pretpostavlja da su implementirane) i već realizovanih koncepta događaja i niti, sa interfejsima kao što je definisano projektnim zadatkom za domaći rad, napisati kompletan kod ovog drajvera uređaja.

Rešenje:

```
#include "event.h"
#include "thread.h"
extern DeviceRequest* getDeviceRequest();
extern void DMARequest (char* block, unsigned long size);

PREPAREENTRY(0x10,0);

Event deviceEvent(0x10);

class DeviceDriver : public Thread {
protected:
  DeviceDriver () : Thread() { start(); }
  virtual void run ();
private:
  static DeviceDriver instance;
};
```

```
DeviceDriver DeviceDriver::instance;

void DeviceDriver::run () {
  while (1) {
    DeviceRequest* dr = getDeviceRequest();
    DMARequest(dr->block,dr->size);
    deviceEvent.wait();
    dr->toSignal->signal();
  }
}
```

Zad 2.
Implementirati bibliotečnu funkciju `void print(char* str)` koja štampa niz karatkera `str`, na znakovno orijentisani uređaj opisan u prethodnom zadatku, kojisteći drajver implementiran takođe u prethodnom zadatku. Funkcija treba da bude blokirajuća i bezbedna za rad u okruženju sa više niti (*reentrant*, odnosno *thread-safe*).

Rešenje:

```
#include <string.h>

void print(char* str){
      DeviceRequest* r = new DeviceRequest();
      r->block = str;
      r->size = strlen(str);
      r->toSignal = new Semaphore(0);

      putDeviceRequest(r);
      r->toSignal->wait();
      delete r->toSignal;
      delete r;
}
```

Napomena: Pretpostavka je da su pozivi operatora `new` i `delete` bezbedni za rad u okruženju sa više niti, u suprotnom potrebno je i ove pozive uokviriti u kritične sekcije.

Zad 3. Kolokvijum – Maj 2006.

U nekom računarskom sistemu postoje spoljašnji maskirajući prekidi koje generišu ulazno/izlazni uređaji. Ovi prekidi maskiraju se operacijom `int_mask()`, a demaskiraju operacijom `int_unmask()`. Po prihvatanju prekida, procesor automatski maskira prekide.

Potrebno je realizovati koncept *spoljašnjeg događaja* (*external event*) koji se manifestuje pojavom spoljašnjeg maskirajućeg prekida. Ovaj koncept treba realizovati klasom `ExternalEvent`, po uzoru na realizaciju semafora pomoću operacija `setjmp()` i `longjmp()` datu na predavanjima. Pretpostaviti da operacije `setjmp()` i `longjmp()` čuvaju celokupan kontekst izvršavanja, tj. sve programski dostupne registre. Interfejs klase `ExternalEvent` treba da izgleda ovako:

```
const int NumOfInterruptEntries = ... ; // Number of interrupt entries

class ExternalEvent {
public:
  ExternalEvent(int interruptNo);
  void wait();

  static void interruptOccurrence(int interruptNo);
};
```

Konstruktor ove klase kreira jednu instancu događaja koji se signalizira prekidom sa brojem ulaza zadatim parametrom konstruktora. Operacija `wait()` blokira pozivajuću nit sve dok se prekid ne dogodi. Pojava prekida na ulazu *N* deblokira jednu nit blokiranu na događaju koji je vezan za prekid sa tim brojem ulaza *N*. Pretpostaviti da u sistemu već postoje prekidne rutine vezane za spoljašnje maskirajuće prekide tako da prekidna rutina za prekid na ulazu *N* poziva funkciju `ExternalEvent::interruptOccurrence(int)` sa argumentom koji predstavlja taj broj ulaza.

Rešenje:

```
// ExternalEvent.h

void int_mask();
void int_unmask();


const int NumOfInterruptEntries = ... ; // Number of interrupt entries

class ExternalEvent {
public:
  ExternalEvent(int interruptNo);
  void wait();

  static void interruptOccurrence(int interruptNo);

protected:
  void signal ();

  void block ();
  void deblock ();

  ~ExternalEvent();
  static ExternalEvent* events[NumOfInterruptEntries];
```

```
private:
  int val, intNo;
  Queue blocked;
};

// ExternalEvent.cpp

ExternalEvent::ExternalEvent(int interruptNo) : val(0),
                                                intNo(interruptNo){
   if (intNo>=0 && intNo<NumOfInterruptEntries && events[interruptNo]==0) {
      int_mask();
      // add the event to the event list
      events[intNo]=this;
      int_unmask();
   }
}

ExternalEvent::~ExternalEvent(){
   int_mask();
   events[intNo]=0;
   int_unmask();
}

void ExternalEvent::interruptOccurrence(int interruptNo){
   if (interruptNo>=0 && interruptNo < NumOfInterruptEntries &&
      events[interruptNo])
     events[interruptNo]->signal();
}

void ExternalEvent::block () {
  if (setjmp(Thread::runningThread->context)==0) {
    // Blocking:
    blocked->put(Thread::runningThread);
    Thread::runningThread = Scheduler::get();
    longjmp(Thread::runningThread->context);
  } else return;
}

void ExternalEvent::deblock () {
  // Deblocking:
  Thread* t = blocked->get();
  Scheduler::put(t);
}

void ExternalEvent::wait () {
  int_mask();
  if (--val<0) block();
  int_unmask();
}

void ExternalEvent::signal () {
  if (val++<0) deblock();
}
```

Napomena: S obzirom da se u zadatku zahteva realizacija po uzoru na semafore date na predavanjima potrebno je za dati ulaz voditi evidenciju o broju generisanih prekida, u suprotnom na kraju operacije `signal()` bi trebalo dodati sledeće `if(val>1) val=1;`

Zad 4. Kolokvijum – Maj 2006.

Korišćenjem koncepta spoljašnjeg događaja iz prethodnog zadatka, realizovati telo niti koja ciklično izvršava sledeći posao:

- uzima jedan zahtev iz liste postavljenih zahteva za ulaznom operacijom prenosa bloka podataka sa nekog ulaznog uređaja u memoriju pomoću DMA kontrolera, odnosno čeka blokirana ako takvog zahteva nema
- pokreće operaciju prenosa bloka podataka na zadato mesto u memoriji pomoću DMA kontrolera
- čeka (suspendovana) da DMA kontroler završi prenos, što DMA kontroler signalizira prekidom na ulazu N
- kada se završi prenos, označava zahtev ispunjenim i signalizira semafor koji je zadat u zahtevu, kako bi se nit koja je postavila zahtev deblokirala.

Zahtev za ulaznom operacijom i red zahteva izgledaju ovako:

```
struct IORequest {
  IORequest* next; // Next in the request queue
  void* buffer;  // Input buffer in memory
  long  size;  // Size of the data block
  int isCompleted; // Is this request completed?
  Semaphore* toSignal;  // The semaphore to signal on completion
};

extern IORequest* requestQueue;
extern Semaphore mutex;
extern Semaphore requestQueSize;
```

Za pristup do reda zahteva treba obezbediti međusobno isključenje pomoću klasičnog brojačkog semafora `mutex`. Brojački semafor `requestQueSize` ima vrednost koja je jednaka broju zahteva u redu zahteva `requestQueue`. Prenos bloka podataka sa DMA kontrolera pokreće se operacijom `void startDMA(void* buffer, long size)`.

Rešenje:

```
ExternalEvent dmaEvent(N);

while (1) {
  requestQueSize.wait();
  mutex.wait();
  IORequest* r = requestQueue;
  requestQueue = requestQueue->next;
  mutex.signal();

  startDMA(r->buffer, r->size);
  dmaEvent.wait();

  r->isCompleted = 1;
  r->toSignal->signal();
}
```

Zad 5. Septembar 2005.

Analogno-digitalna (A/D) konverzija je postupak konverzije odbirka nekog analognog signala na ulazu u računar u njegovu digitalnu aproksimaciju. A/D konvertor je ulazni uređaj koji vrši ovakvu konverziju. Da bi A/D konvertor izvršio konverziju, potrebno mu je najpre zadati (pokrenuti) operaciju konverzije upisom 1 u bit START njegovog upravljačkog registra. Da bi izvršio konverziju, A/D konvertoru je potrebno izvesno vreme. Po završetku konverzije, A/D konvertor upisuje digitalni rezultat u svoj registar za podatke i signalizira završetak operacije postavljanjem bita EOC (*end of conversion*) u svom statusnom registru. Ukoliko je ovaj indikator povezan na odgovarajući način, završetak konverzije generiše prekid procesoru. Poznato je da A/D konverzija svakako traje najviše 50 ms, ali može da traje i znatno kraće. Na raspolaganju su sledeće funkcije iz C API nekog operativnog sistema:

`void startAD()` pokreće A/D konverziju prostim upisom 1 u bit START A/D konvertora

`int getEOC()` čita statusni registar A/D konvertora i vraća vrednost bita EOC (0 ili 1)

`int getData()` čita registar podataka A/D konvertora i vraća pročitanu vrednost

`void waitAD()` suspenduje pozivajući proces sve dok se ne dogodi prekid zbog EOC

`void delay(unsigned ms)` suspenduje pozivajući proces na vreme zadato argumentom u ms

Korišćenjem ovih funkcija, realizovati sledeće funkcije:

(a)(5) `int adConvert()`: sinhrona, blokirajuća operacija koja treba da izvrši A/D konverziju i vrati dobijenu digitalnu vrednost, pod pretpostavkom da EOC jeste vezan na ulaz zahteva za prekid.

(b)(5) `int adConvert()`: sinhrona, blokirajuća operacija koja treba da izvrši A/D konverziju i vrati dobijenu digitalnu vrednost, pod pretpostavkom da EOC nije vezan na ulaz zahteva za prekid.

(c)(5) `int getADConv()`: sinhrona, neblokirajuća operacija koja treba da zada A/D konverziju, uposleno čeka neko kratko vreme (po Vašem izboru) i vrati dobijenu digitalnu vrednost (koja je uvek nenegativna) ako je konverzija završena, odnosno –1 ako konverzija nije završena u tom vremenu.

Rešenje:

(a)(5)

```
int adConvert () {
  startAD();
  waitAD();
  return getData();
}
```

(b)(5)

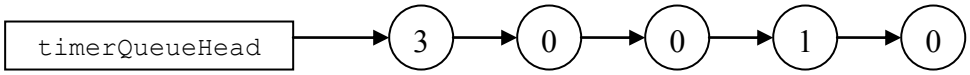
```
int adConvert () {
  startAD();
  delay(50);
  return getData();
}
```

(c)(5)

```
int getADConv () {
  startAD();
  for (int i=0; i<...; i++);
  if (getEOC()) return getData(); else return -1;
}
```

Zad 6. Jun 2010

U nekom sistemu obezbeđena je podrška procesima za usluge merenja vremena i uspavljivanja procesa na zadato vreme. U podsistemu za merenje vremena ovakvi zahtevi smeštaju se u jednostruko ulančanu listu elemenata tipa `TimerRequest` čija je deklaracija data dole. Na početak ove liste zahteva ukazuje globalni pokazivač `timerQueueHead`. Svaki zahtev tipa `TimerRequest` sadrži pokazivač `event` na događaj na kome je blokiran proces koji se uspavao na zadato vreme i koji treba „probuditi“ (signaliziranjem tog događaja) kada to vreme dođe. Poseban hardverski uređaj (tajmer) generiše periodične prekide. Zahtevi se u red smeštaju po hronološkom redosledu, tako da u svakom zahtevu polje `time` predstavlja dužinu intervala (izraženu u jedinici vremena koja je jednaka periodi periodičnog prekida sa hardverskog tajmera) koji treba da protekne od „buđenja“ prethodnog zahteva u listi do trenutka „buđenja“ posmatranog zahteva u listi. Prvi zahtev u listi treba „probuditi“ za `time` perioda u odnosu na trenutak poslednjeg prekida. Na primer, za red zahteva na donjoj slici, prva tri zahteva treba probuditi za 3 „otkucaja“ tajmera, a četvrti i peti za 1 nakon njih, odnosno za 4 od poslednjeg otkucaja.




```
struct TimerRequest {
  TimerRequest* next;  // Next in the list of timer requests
  unsigned long int time;
  Event* event;
};

extern TimerRequest* timerQueueHead;
```

Realizovati prekidnu rutinu `timer()` koja se poziva na svaki periodični prekid od vremenskog brojača i koja treba da odradi odgovarajući posao ažuriranja liste zahteva i buđenja procesa kojima je isteklo vreme. Operacija `signal()` događaja može da se pozove iz prekidne rutine, jer je za to i predviđena. Prekidna rutina je uvek atomična, pošto se radi o jednoprocesorskom sistemu koji hardverski maskira druge prekide pri prihvatanju prekida. Nakon obrade zahteva, ne treba ga dealocirati iz memorije, pošto je to odgovornost onoga ko je zahtev postavio.

Rešenje:

```
interrupt void timer () {
  if (timerQueueHead==0) return;
  timerQueueHead->time--;
  while (timerQueueHead && timerQueueHead->time==0) {
    timerQueueHead->event->signal();
    timerQueueHead=timerQueueHead->next;
  }
}
```

Zad 7. Oktobar 2005.

(a)(5) Kojom tehnikom se fizički nedeljivi izlazni uređaj može učiniti logički (virtuelno) deljivim između procesa koji ga uporedo koriste?

(b)(5) Kojom tehnikom se fizički sekvencijalni ulazni uređaj sa koga se učitava ograničeni niz podataka veličine znatno manje od virtuelnog adresnog prostora procesa može učiniti logički (virtuelno) uređajem sa direktnim pristupom?

Rešenje:

(a)(5) *Spooling*.

- operacije zadate uređaju od strane procesa smeštaju se u poseban fajl; kada završi, proces zatvara fajl i "predaje ga" OSu
- poseban proces ili nit OSa, spooler, uzima jedan po jedan fajl iz reda i prenosi njegov sadržaj na uređaj

(b)(5) Baferisanje.

Zad 8. April 2006.

a)(5) Ako sistem koristi *spooling*, kakva je po svojoj prirodi operacija slanja niza znakova na štampač koju poziva korisnički proces, posmatrana „sa kraja na kraj", tj. iz perspektive korisničkog procesa na operaciju koju zadaje i uređaj koji treba da je izvrši - asinhrona ili sinhrona (blokirajuća)?

Rešenje:

a)(5) Uvek asinhrona, jer proces zadaje operaciju i nastavlja svoje izvršavanje, ne čekajući da se uređaj završi zadatu operaciju. Operacija neće ni početi pre nego što dati proces „zatvori" svoj posao sa uređajem (a posao može da uključuje više ovakvih operacija) i kada taj posao dođe na red za obradu od strane *spooler*-a. Zato pozivajući proces nikada i ne treba da čeka na završetak ove operacije.

b)(5) Povećanje bafera i blokova prenosa prilikom upisa na disk ima mnoge pozitivne efekte, poput ređih i efikasnijih operacija sa uređajima. Međutim, kakav je negativan uticaj povećanja bafera, u pogledu posledica u slučaju otkaza?

Rešenje:

b)(5) Veći baferi znače i ređe slanje podataka na disk, što znači veću štetu u slučaju otkaza: više podataka duže vreme nije snimljeno na disk i ostaje trajno izgubljeno u slučaju otkaza računara.

Zad 9. Januar 2011

U nekom modularnom operativnom sistemu drajveri za raznovrsne ulazno/izlazne blokovski orijentisane uređaje registruju se kao moduli na sledeći način. Sistem održava jedinstvenu, centralizovanu tabelu registrovanih drajvera `blockIOMap` u kojoj svaki ulaz odgovara jednom registrovanom drajveru. Prilikom instalacije, svaki drajver u novi ulaz ove tabele upisuje adresu svoje tabele tipa `BlockIOVectorTable` sa adresama svojih funkcija zaduženih za odgovarajuće ulazno/izlazne operacije. Date su sledeće deklaracije:

```
// I/O request:
struct BlockIORequest {...};

// I/O operation:
typedef int (*BlockIOOp) (BlockIORequest*);

// I/O driver vector table:
struct BlockIOVectorTable {
  BlockIOOp open, close, read, write;
};

// I/O device driver map:
BlockIOVectorTable* blockIOMap[MAXDEVICES];
extern int numOfRegisteredDrivers;
typedef int HANDLE;
```

a)(5) Implementirati sledeće funkcije sistema kojima se zadaju operacije uređaju datom prvim argumentom:

```
int open  (HANDLE device, BlockIORequest* req);
int close (HANDLE device, BlockIORequest* req);
int read  (HANDLE device, BlockIORequest* req);
int write (HANDLE device, BlockIORequest* req);
```

Rešenje:

```
int open (HANDLE device, BlockIORequest* req) {
  if (device>=numOfRegisteredDrivers) return -1;
  BlockIOVectorTable* vt = blockIOMap[device];
  if (vt==0) return -1;
  BlockIOOp op = vt->open;
  if (op==0) return -1;
  return (*op)(req);
}
```

Analogno ostale.

b)(5) Implementirati operaciju za registraciju drajvera koja vraća identifikator uređaja koji će se koristiti kao „ručka“ u daljim operacijama sa tim uređajem, a predstavlja broj ulaza u tabeli registrovanih uređaja (-1 u slučaju greške):

```
HANDLE blockIODriverReg (BlockIOVectorTable*);
```

Rešenje:

```
HANDLE blockIODriverReg (BlockIOVectorTable* vt) {
  if (numOfRegisteredDrivers>=MAXDEVICES) return -1;
  blockIOMap[numOfRegisteredDrivers]=vt;
  return numOfRegisteredDrivers++;
}
```

Zad 10. Januar 2006.

Posmatra se neki fizički sekvencijalni ulazni uređaj sa koga se učitava ograničeni tok znakova veličine `SIZE` znatno manje od virtuelnog adresnog prostora procesa, a sa koga se jedan znak učitava funkcijom:

```
char getchar();
```

Potrebno je obezbediti programski interfejs (API) za ovaj uređaj koji će ga tehnikom baferisanja učiniti logički (virtuelno) uređajem sa direktnim pristupom. Treba realizovati sledeće funkcije ovog interfejsa (uz odgovarajuće strukture podataka):

```
void seek(int position);
int read(char* buf, int sz);
```

Funkcija `seek()` postavlja „kurzor“ za čitanje sa ovog virtuelnog uređaja na zadatu poziciju. Funkcija `read()` prenosi niz znakova sa virtuelnog uređaja počev od pozicije kurzora u niz `buf` (alociran od strane pozivaoca); maksimalno se prenosi `sz` znakova (to je veličina niza `buf`), odnosno onoliko koliko znakova preostaje u ulaznom toku (koji je veličine `SIZE`) počev od mesta kurzora; funkcija vraća broj stvarno učitanih znakova (`sz` ili manje). Funkcija `read()` treba da učitava znakove sa fizičkog uređaja u bafer samo koliko je potrebno: ako bafer ne sadrži sve znakove koji se traže u funkciji `read()`, ona treba da učita najmanje što može (ne odmah ceo ulazni tok).

Rešenje:

```
const int SIZE = ...;
char buffer[SIZE];
int cursor = 0;
int numOfLoaded = 0; // Broj znakova učitanih sa ulaznog toka

void seek (int pos) {
  if (pos>=0 && pos<SIZE) cursor=pos;
}

int read (char* b, int sz) {
  int toLoad = cursor+sz-numOfLoaded;
  if (toLoad>SIZE-numOfLoaded) toLoad=SIZE-numOfLoaded;
  for (; toLoad>0; toLoad--)  // Load
    buffer[numOfLoaded++]=getchar();
  int num=0;
  for (; num<sz && cursor+num<SIZE; num++)
    b[num]=buffer[cursor+num];
  return num;
}
```

Zad 11. Feb 2011.

Implementirati internu nit jezgra, zajedno sa odgovarajućom prekidnom rutinom, koja uzima jedan po jedan zahtev za ulaznom operacijom na nekom ulaznom uređaju iz ograničenog bafera i obavlja prenos bloka podataka zadat tim zahtevom korišćenjem dva raspoloživa kanala nekog DMA uređaja. Ova dva kanala omogućavaju dva uporedna prenosa sa istog uređaja. Ograničeni bafer, realizovan klasom `InputBuffer`, u sebi ima implementiranu sinhronizaciju proizvođača i potrošača. Zahtev ima sledeću strukturu:

```
struct InputRequest {
  char* buffer; // Buffer with data (block)
  unsigned int size; // Buffer (blok) size
  Semaphore* toSignal; // Semaphore to signal on request completion
};
```

Kada se završi prenos zadat zahtevom, potrebno je signalizirati semafor na koga ukazuje `InputRequest::toSignal` i obrisati zahtev. Date su deklaracije pokazivača preko kojih se može pristupiti registrima DMA uređaja, pošto su oni inicijalizovani adresama tih registara:

```
typedef unsigned int REG;
REG* dmaCtrl[2] =...; // control registers for 2 DMA channels
REG* dmaStatus[2] =...; // status registers for 2 DMA channels
REG* dmaBlkAddr[2] =...; // data block address registers for 2 DMA channels
REG* dmaBlkSize[2] =...; // data block size registers for 2 DMA channels
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće prenos na datom DMA kanalu, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je dati DMA kanal spreman za novi prenos podatka (inicijalno je tako postavljen). Postavljanje bilo kog od ova dva bita spremnosti kada neki kanal završi prethodni prenos generiše signal zahteva za prekid na istom IRQ ulazu.
Na raspolaganju za implementaciju potrebne sinhronizacije između prekidne rutine i niti su standardni brojački semafori, čija se operacija *signal* može pozivati iz prekidne rutine.

Rešenje:

```
class InputThread : public Thread { // Must be a singleton!
public:
  InputThread () : Thread (), endOfTransfer(2) {}
protected:
  virtual void run ();
private:
  friend void inputThreadInterrupt();
  static InputRequest* pendingRequest[2];
  static Semaphore endOfTransfer;
};

void InputThread::run () {
  pendingRequest[0]=pendingRequest[1]=0;
  int i=0;
  while (1) {
    endOfTransfer.wait(); // Wait for a DMA channel to become ready
    for (i=0; i<2; i++)  // Detect which DMA channel is ready
      if (*dmaStatus[i] & 1) break;
    if (i>=2) return; // Unexpected exception!
    if (pendingRequest[i]) { // Notify transfer completion
      if (pendingRequest[i]->toSignal)
        pendingRequest[i]->toSignal->signal();
      delete pendingRequest[i];  //destroys only request, not semaphore
      pendingRequest[i]=0;
    }
    pendingRequest[i] = InputBuffer::get(); // Get a new one, if exists,...
    *dmaBlkAddress[i] = pendingRequest[i]->buffer; // ...and start it
    *dmaBlkSize[i] = pendingRequest[i]->size;
    *dmaCtrl[i] = 1;
  }
}

interrupt void inputThreadInterrupt () {
  InputThread::endOfTransfer.signal();
}
```

Napomena: Obratiti pažnju da se ova nit blokira u dva slučaja, pozivom operacija `endOfTransfer.wait()` i `InputBuffer::get()`pa dok je jedan kanal slobodan, a drugi zauzet postoji dilema da li čekati na novi zahtev ili čekati na periferiju kako bi se signaliziralo korisniku da je prenos završen. U ovom slučaju je pretpostavljeno da zahtevi brže pristižu nego što ih periferija obrađuje, pa je izabran prvi pristup. Ovo može predstavljati problem jer

se signalizacija korisniku da je prenos završen nepotrebno se odlaže do trenutka kada pristigne novi zahtev pa je potrebno izmestiti ovu signalizaciju unutar prekidne rutine, na sledeći način:

```
int detectReadyDMAchannel () {
  int i=0;
  for (i=0; i<2; i++)  // Detect which DMA channel is ready
      if (*dmaStatus[i] & 1) break;
  return i;
}

void InputThread::run () {
  pendingRequest[0]=pendingRequest[1]=0;
  int i=0;
  while (1) {
    endOfTransfer.wait(); // Wait for a DMA channel to become ready
    i = detectReadyDMAchannel (); // Detect which DMA channel is ready
    if (i>=2) return; // Unexpected exception!
    if (pendingRequest[i]) { // transfer completed
      delete pendingRequest[i];  //destroys only request, not semaphore
      pendingRequest[i]=0;
    }
    pendingRequest[i] = InputBuffer::get(); // Get a new one, if exists,...
    *dmaBlkAddress[i] = pendingRequest[i]->buffer; // ...and start it
    *dmaBlkSize[i] = pendingRequest[i]->size;
    *dmaCtrl[i] = 1;
  }
}

interrupt void inputThreadInterrupt () {
  int i = detectReadyDMAchannel ();
  if (i<2 && pendingRequest[i]) { // Notify transfer completion
     if (pendingRequest[i]->toSignal)
        pendingRequest[i]->toSignal->signal();
  InputThread::endOfTransfer.signal();
}
```