



Dragan Milićev, Bojan Furlan

Programiranje u realnom vremenu - Skripta sa praktikumom i rešenim zadacima

Univerzitet u Beogradu – Elektrotehnički fakultet
Beograd, 2011.

Drugo izdanje, 2019.

Sadržaj

Sadržaj.....	2
Predgovor.....	8
Predgovor drugom izdanju.....	8
I UVOD U SISTEME ZA RAD U REALNOM VREMENU.....	9
Definicija sistema za rad u realnom vremenu.....	10
Podela i terminologija RT sistema.....	11
Primeri RT sistema.....	12
Karakteristike RT sistema.....	13
Primeri struktura pogodnih za RT implementacije.....	15
Kolekcija implementirana kao dvostruko ulančana dinamička lista.....	15
Kolekcija kao ulančana lista sa vezama ugrađenim u objekte.....	22
FIFO Red.....	29
Red sa prioritetom.....	30
Efikasna alokacija memorije.....	35
II POUZDANOST I TOLERANCIJA OTKAZA.....	39
Pouzdanost i tolerancija otkaza.....	40
Pouzdanost, padovi i otkazi.....	40
Sprečavanje i tolerancija otkaza.....	41
Sprečavanje otkaza.....	41
Tolerancija otkaza.....	42
Programiranje u N verzija.....	44
Dinamička softverska redundansa.....	46
Detekcija greške.....	46
Izolacija i procena štete.....	48
Oporavak od greške.....	49
Blokovi oporavka.....	50
Poređenje između programiranja u N verzija i blokova oporavka.....	51
Izuzeci i njihova obrada.....	53
Obrada izuzetaka bez posebne jezičke podrške i sa njom.....	53
Izvori izuzetaka.....	55
Reprezentacija izuzetaka.....	56
Obrada izuzetka.....	57
Propagacija izuzetka.....	59
Tretman izuzetaka i programiranje po ugovoru.....	60
Zadaci.....	62
2.1. Programiranje u N verzija.....	62
Rešenje:.....	62
2.2. BER tehnike.....	63
Rešenje:.....	64
2.3. Blokovi oporavka.....	65
Rešenje:.....	65
Zadaci za samostalan rad.....	65
2.4.....	65
2.5.....	66

2.6.....	66
----------	----

III OSNOVE KONKURENTNOG PROGRAMIRANJA.....67

Konkurentnost i procesi.....	68
Konkurentno programiranje.....	68
Pojam procesa.....	70
Predstavljanje procesa.....	72
Procesi na jeziku Ada.....	72
Niti na jeziku Java.....	73
Niti na jeziku C++.....	74
POSIX niti.....	75
Interakcija između procesa.....	76
Implementacija niti.....	77
Promena konteksta.....	78
Raspoređivanje.....	80
Kreiranje niti.....	81
Ukidanje niti.....	83
Pokretanje i gašenje programa.....	84
Realizacija.....	85
Zadaci.....	95
3.1.....	95
Rešenje.....	95
3.2. Obrada izuzetaka.....	95
Rešenje.....	95
Zadaci za samostalan rad.....	96
3.3.....	96
3.4.....	97
3.5.....	97
Sinhronizacija i komunikacija pomoću deljene promenljive.....	98
Međusobno isključenje i uslovna sinhronizacija.....	98
Međusobno isključenje.....	98
Uslovna sinhronizacija.....	100
Uposleno čekanje.....	100
Sinhronizacija korišćenjem hardverske podrške.....	105
Semafori.....	107
Implementacija.....	108
Međusobno isključenje i uslovna sinhronizacija pomoću semafora.....	110
Binarni i drugi semafori.....	113
Monitori.....	114
Uslovna sinhronizacija u monitoru.....	115
Problemi vezani za monitore.....	117
Zaštićeni objekti u jeziku Ada.....	117
Sinhronizovane operacije u jeziku Java.....	119
Klasifikacija poziva operacija.....	122
Implementacija sinhronizacionih primitiva.....	122
Semafor.....	122
Događaj.....	123
Monitor.....	124
Ograničeni bafer.....	125
Zadaci.....	127
4.1.....	127
Rešenje - Ada.....	127
4.2 Cigarette smokers.....	128
Rešenje.....	128
4.3.....	131
Rešenje.....	131
4.4.....	133
Rešenje.....	133

Zadaci za samostalan rad.....	134
4.4.....	134
4.5.....	134
4.7.....	134
Sinhronizacija i komunikacija pomoću razmene poruka.....	135
Sinhronizacija procesa.....	135
Imenovanje procesa.....	138
Struktura poruke.....	140
Randevu u jeziku Ada.....	141
Zadaci.....	144
5.1. Randevu na jeziku Ada.....	144
Rešenje.....	144
Zadaci za samostalan rad.....	145
5.2.....	145
5.3.....	145
Pristup deljenim resursima.....	146
Modeli pristupa deljenim resursima.....	146
Čitaoci i pisci.....	146
Filozofi koji večeraju.....	148
Problemi nadmetanja za deljene resurse.....	148
Utrkivanje.....	149
Mrtvo blokiranje.....	150
Živo blokiranje.....	158
Izgladnjivanje.....	159
Jedno rešenje problema filozofa.....	160
Zadaci za samostalan rad.....	161
6.1.....	161
6.2.....	161
6.3.....	161
6.4.....	161
6.5.....	161
6.6.....	161
6.7.....	161
6.8.....	161
6.9.....	162
6.10.....	162
6.11.....	162

IV SPECIFIČNOSTI RT PROGRAMIRANJA.....163

Realno vreme.....	164
Časovnik realnog vremena.....	164
Časovnik u jeziku Ada.....	165
Časovnik u jeziku Java.....	166
Merenje proteklog vremena.....	167
Kašnjenje procesa.....	169
Vremenske kontrole.....	170
Deljene promenljive i vremenske kontrole.....	171
Komunikacija porukama i vremenske kontrole.....	172
Vremenske kontrole u školskom jezgri.....	174
Specifikacija vremenskih zahteva.....	175
Periodični procesi.....	176
Sporadični procesi.....	178
Kontrola zadovoljenja vremenskih zahteva.....	179
Detekcija prekoračenja roka.....	180
Detekcija prekoračenja ostalih vremenskih ograničenja.....	181
Proračun vremenskih parametara.....	181

Implementacija u školskom jezgri.....	187
Mehanizam merenja vremena.....	187
Obrada prekida.....	193
Zadaci.....	195
7.1 Konstrukt Delay.....	195
Rešenje.....	195
7.2 Timer.....	196
Rešenje.....	197
7.3 PeriodicTask.....	197
Rešenje.....	198
7.4 State.....	199
Rešenje.....	199
7.5 A/D konverzija.....	203
Rešenje.....	204
7.6 Timed Semaphore.....	205
Rešenje.....	205
7.7 Sigurnosni kočioni sistem.....	206
Rešenje.....	206
Zadaci za samostalan rad.....	208
7.8.....	208
7.9.....	208
7.10.....	208
7.11.....	209
7.12.....	209
7.13.....	209
7.14.....	209
7.15.....	210
7.16.....	210
7.17.....	210
7.18.....	211
7.19.....	211
7.20.....	211
7.21.....	212
Raspoređivanje i rasporedivost.....	213
Pojam raspoređivanja i rasporedivosti.....	213
Jednostavan model procesa.....	214
Ciklično izvršavanje.....	214
Raspoređivanje procesa.....	216
FPS i RMPO.....	218
EDF.....	220
Testovi rasporedivosti.....	221
Test rasporedivosti za FPS zasnovan na iskorišćenju.....	221
Test rasporedivosti za EDF zasnovan na iskorišćenju.....	223
Test rasporedivosti za FPS zasnovan na vremenu odziva.....	224
Procena WCET.....	226
Opštiji model procesa.....	228
Sporadični procesi.....	228
Aperiodični procesi.....	228
Hard i soft procesi.....	228
Sistem procesa sa $D < T$	229
Interakcija procesa i blokiranje.....	230
Tehnike za rešavanje inverzije prioriteta.....	233
Zadaci.....	236
8.1 Ciklično izvršavanje i FPS.....	236
Rešenje.....	236
.....	237
8.2 Test za FPS zasnovan na vremenu odziva.....	237
Rešenje.....	237
8.3.....	238

Rešenje.....	238
8.4.....	238
Rešenje.....	239
Zadaci za samostalan rad.....	240
8.5.....	240
8.6.....	240
8.7.....	240
8.8.....	241
8.9.....	241
8.10.....	241
V MODELOVANJE RT SISTEMA.....	242
Uvod.....	243
Istorijat.....	243
Principi.....	243
Jednostavan ROOM model.....	245
Definicija interfejsa aktera.....	245
Definicija ponašanja aktera.....	246
Izvršavanje modela.....	250
Hijerarhijski model sa više aktera.....	252
Komunikacija između aktera.....	252
Sistemske servisne pristupne tačke.....	257
Interne sekvence poruka.....	258
Hijerarhijske mašine stanja.....	259
Izvršavanje modela sa više aktera.....	260
Nivo detalja.....	261
Nasledivanje.....	262
VI PRAKTIKUM.....	263
Rational Rose Technical Developer.....	264
Podešavanje okruženja.....	265
Postavka zadatka.....	266
Pretpostavke.....	266
Kreiranje modela.....	268
Kreiranje novog modela.....	268
Kreiranje dijagrama slučajeva upotrebe.....	270
Kreiranje kapsula (aktera).....	270
Kreiranje dijagrama strukture za kapsule.....	271
Kreiranje strukture kapsule Alarm.....	271
Kreiranje dijagrama sekvence.....	274
Kreiranje protokola.....	274
Kreiranje signala za dati protokol.....	274
Kreiranje portova i konektora.....	275
Kreiranje portova.....	275
Dokumentovanje.....	277
Dodavanje ponašanja kapsuli.....	277
Kreiranje stanja.....	278
Kreiranje tranzicija.....	278
Kreiranje okidača.....	279

Dodavanje akcije.....	280
Kompajliranje i izvršavanje.....	281
Kreiranje komponente.....	281
Kompajliranje.....	282
Izvršavanje.....	282
Testiranje.....	285
Proširenje modela.....	288
Kreiranje <i>timing</i> porta.....	289
Povezivanje portova.....	290
Dodavanje atributa.....	291
Kreiranje stanja.....	292
Testiranje.....	293
Kreiranje traga.....	295
Samostalan rad.....	296
Korisni linkovi.....	296
Literatura.....	297

Predgovor

Ovaj nastavni materijal je proizašao iz materijala sačinjenih za predavanja i vežbe za predmet Programiranje u realnom vremenu koji se dugi niz godina drži na osnovnim i akademskim master studijama na Elektrotehničkom fakultetu u Beogradu. Nivo složenosti i način izlaganja podrazumeva neophodno predznanje iz objektno orijentisanog programiranja, algoritama i struktura podataka, operativnih sistema, kao i konkurentnog i distribuiranog programiranja.

U cilju boljeg i lakšeg usvajanja izložene problematike, kao i potpunijeg sagledavanja ove savremene inženjerske oblasti u skriptama su izloženi rešeni problemi i zadaci koji su dati na kraju svake oblasti, a poslednje poglavlje predstavlja praktikum koji se odnosi na upotrebu jednog savremenog aplikativnog okruženja za modelovanje i razvoj softverskih sistema za rad u realnom vremenu.

Skripta su prvenstveno namenjena studentima Elektrotehničkog fakulteta koji slušaju pomenuti predmet na akademskim master studijama modula Softversko inženjerstvo ili Računarska tehnika i informatika. Ona u potpunosti obuhvata gradivo predviđeno za predavanja i vežbe, kao i za laboratorijske vežbe. Autori se nadaju da će tekst biti od koristi i drugim studentima, inženjerima i praktičarima koji imaju dodira sa ovom oblašću.

Na kraju, autori su svesni da i pored uloženog truda, ovaj materijal može sadržati greške, stoga biće zahvalni za sve sugestije, korekcije i primedbe čitalaca.

Beograd, maj 2011.

Autori

Predgovor drugom izdanju

U drugom izdanju mnogi delovi su izmenjeni ili dopunjeni, a neki su izbačeni, u skladu sa potrebama izvođenja nastave. Ispravljene su i uočene greške.

Beograd, decembar 2019.

I Uvod u sisteme za rad u realnom vremenu

Definicija sistema za rad u realnom vremenu

- Sa razvojem hardvera – procesori, memorija, ulazno/izlazni uređaji postajali su vremenom sve brži, veći po kapacitetu a manji po gabaritu, i sve jeftiniji i dostupniji – širile su se i mogućnosti primene računara i softvera u najrazličitijim oblastima.
- Jedna od najbrže razvijanih oblasti primene jesu one aplikacije koje nemaju kao svoj primarni cilj obradu informacija, nego je obrada informacija samo sredstvo i neophodan preduslov za ispunjenje njihove osnovne namene – *nadzor i upravljanje* nekog većeg sistema odnosno procesa, tipično preko odgovarajućeg hardverskog podsistema.
- Primeri:
 - najjednostavniji uređaji, poput onih u domaćinstvu, već odavno imaju ugrađene procesore koji izvršavaju softver za upravljanje funkcijama tih uređaja, umesto nikakvog ili analognog automatskog upravljanja; npr. mašina za pranje veša nekada je imala analogni programator koji upravlja složenim elektromehaničkim podsistemima mašine (motor, pumpa, ventili, grejač), a sada ima digitalni računar, ali i drugi, jednostavniji kućni uređaji danas imaju procesore;
 - „pametni“ lični uređaji, poput satova i telefona su odavno kompletni računari;
 - moderni automobili imaju na desetine mikroprocesorskih modula koji upravljaju posebnim podsistemima automobila;
 - složenija prevozna sredstva odavno imaju ogroman broj ovakvih podsistema (vozovi, uključujući i one bez vozača, avioni itd.).
- Ovakav softverski sistem koji služi za nadzor i upravljanje određenog većeg inženjerskog (hardverskog) okruženja i koji ispunjava svoj cilj obradom informacija, ali pri čemu obrada informacija jeste samo sredstvo, a ne njegov primarni cilj, nazivaju se *ugrađeni* (engl. *embedded*) sistemi.
- Druga, nešto novija i opštija kategorija jesu tzv. *kibernetičko-fizički sistemi* (engl. *cyber-physical systems*, CPS): dok ugrađeni sistem podrazumeva softver ugrađen u mikroprocesorski hardver koji je deo nekog većeg sistema, CPS podrazumeva mrežu ovakvih uređaja u interakciji sa fizičkim (npr. mehatroničkim) komponentama i stavlja akcenat na interakciju ovih podsistema i posmatra ih kao celinu. Primer je paradigma *interneta stvari* (engl. *Internet of Things*, IoT).
- Postoji procena da od ukupne svetske proizvodnje mikroprocesora, 99% njih radi u ovakvim sistemima.
- Navedeni sistemi često spadaju u kategoriju *sistema za rad u realnom vremenu* (engl. *real-time system*, kratko RT sistem), odnosno imaju karakteristike RT sistema, iako ove kategorije nisu uvek ekvivalentne: može postojati i ugrađen sistem koji nije RT sistem, ili obrnuto, mada je to izuzetno retko. Zato se vrlo često ove kategorije poistovećuju.
- Nekoliko definicija RT sistema koje se sreću u literaturi [1]:
 - RT sistem je sistem za koji je vreme za koje se proizvede odgovor značajno. To je najčešće zbog toga što ulaz predstavlja neku promenu u realnom okruženju, uključujući i protok vremena, a izlaz treba da odgovara toj promeni. Kašnjenje od ulaza do izlaza mora da bude dovoljno malo da bi izlaz bio prihvatljiv.
 - RT sistem je sistem koji obrađuje informacije i koji treba da odgovori na spoljašnje pobude u konačnom i specifikovanom *vremenskom roku* (engl. *deadline*).

- RT sistem je sistem koji reaguje na spoljašnje pobude (uključujući i protok vremena) u okviru vremenskih intervala koje diktira okruženje.
- *RT sistem je sistem koji obrađuje informacije i čije korektno funkcionisanje ne zavisi samo od logičke korektnosti rezultata, nego i od njihove pravovremenosti.* Drugim rečima, rezultat isporučen neblagovremeno ili uopšte neisporučen je isto tako loš kao i pogrešan rezultat.
- Treba primetiti to da se u svim definicijama pominje *vremenska* odrednica funkcionisanja sistema, ali da se ona uvek posmatra relativno. Ne postoji apsolutna odrednica koja bi definisala koliko malo vreme odziva treba da bude da bi sistem bio klasifikovan kao RT, već to vreme zavisi od konkretnog sistema, odnosno primene – bitno je samo da je to vreme definisano. Ono treba samo da bude "dovoljno malo", posmatrano relativno za dati sistem. To može biti od nekoliko milisekundi (npr. sistem za vođenje rakete), desetina ili stotina milisekundi (npr. telefonska centrala), do čak nekoliko sekundi (npr. kontrola inertnog industrijskog procesa) – RT sistem ne mora uvek biti i „brz“.

Podela i terminologija RT sistema

- Tradicionalna podela RT sistema je na sledeće tipove:
 - "Tvrdi" (engl. *hard*): RT sistemi za koje je apsolutni imperativ da odziv stigne u zadatom vremenskom roku (engl. *deadline*), jer prekoračenje roka (engl. *deadline miss*) ili potpuno neisporučenje rezultata može da dovede do katastrofalnih posledica po živote i zdravlje ljudi, materijalna sredstva ili životnu okolinu. Primeri: sistem za kontrolu nuklearne elektrane, sistem za upravljanje vozom ili letom aviona itd.
 - "Meki" (engl. *soft*): RT sistemi kod kojih su vremenski rokovi važni i treba da budu poštovani, ali se povremeno mogu i prekoračiti, sve dok *performanse* sistema (propusnost i kašnjenje) *statistički* ulaze u zadate okvire. Primeri: telefonska centrala, uređaj za kablovsku TV, sistem za prikupljanje podataka u industriji itd.
- Prema ovim definicijama, važne karakteristike RT sistema su sledeće:
 - Za *hard* sisteme bitno je teorijski i unapred dokazati njihovu *izvodljivost* (engl. *feasibility*), tj. pokazati da se zadati rokovi neće prekoračiti ni u kom slučaju, pri zadatim uslovima i sa raspoloživim resursima. Ova analiza izvodljivosti najčešće podrazumeva analizu *rasporedivosti* (engl. *schedulability*) definisanih poslova na raspoložive procesne jedinice (najčešće procesore), uz uslov zadovoljenja definisanih vremenskih rokova.
 - Za *soft* sisteme bitno je teorijski, simulaciono ili praktično pokazati da su *performanse* sistema zadovoljavajuće, tj. u zadatim granicama pod svim uslovima. To podrazumeva statističku analizu parametara performansi (npr. srednje vrednosti i disperzije), kao što su kašnjenje (engl. *delay*, *latency*) ili propusnost (engl. *throughput*).
- "Stvarnim" RT sistemom (engl. *real real-time*) se naziva *hard* RT sistem kod koga su vremenski rokovi apsolutno kratki (reda milisekundi).
- "Strogim" RT sistemom (engl. *firm real-time*) se naziva *soft* RT sistem kod koga je zakasneli odgovor beskoristan.
- Mnogi sistemi u praksi imaju više svojih komponenta koje spadaju u različite navedene kategorije. Vrlo retko je jedan sistem u celini *hard*, već ima svoje podsisteme ili funkcionalnosti koji spadaju u ovu kategoriju, dok su ostali van nje.

Primeri RT sistema

- Sistemi za kontrolu procesa (engl. *process control systems*): upravljanje cevovodima, namešavanje supstanci, praćenje sagorevanja, nadzor električne centrale itd.
- Sistemi za proizvodnju (engl. *manufacturing systems*): pokretna traka za sastavljanje delova, računarski upravljane mašine (CNC) itd.
- Sistemi za komunikaciju, upravljanje i nadzor (engl. *communication, command, and control systems*, CCC): kontrola leta, upravljanje šinskim saobraćajem, upravljanje projektilima, avionski sistem itd.
- Telekomunikacioni sistemi: telefonska centrala (javna, kućna, za mobilnu telefoniju), mobilni telefon, komunikacioni uređaji (*router, switch*, itd.), uređaji za reprodukciju zvuka i slike (TV, uređaj za kablovsku TV) itd.
- Razni drugi ugrađeni (engl. *embedded*) sistemi: medicinski sistemi, uređaji u domaćinstvu itd.
- Kibernetско-fizički sistemi: roboti, autopilotska vozila (vozovi, automobili, letilice) itd.
- Simulacioni sistemi: simulacija leta aviona, simulacija borbenih dejstava itd.

Karakteristike RT sistema

- RT sistemi su često vrlo veliki i složeni softverski sistemi i, kao takvi, podležu svim principima softverskog inženjerstva složenih sistema. Njihova funkcionalnost je složena, sastoji se od stotina „funkcionalnih tačaka“, a implementacija može da varira od nekoliko stotina linija asemblerkog koda, pa do miliona linija koda na nekom višem programskom jeziku. Teško je ili nemoguće da takav sistem razume, napravi ili održava jedna osoba, već to rade timovi. Zbog toga se ovakvi sistemi moraju specifikovati, projektovati, imlementirati, testirati i dokumentovati uz poštovanje svih opštih principa softverskog inženjerstva, a posebno primenom apstrakcije i dekompozicije, kao i principa raspodele odgovornosti, enkapsulacije, lokalizacije projektnih odluka itd.
- RT softver neposredno interaguje sa hardverom, pa je zato neophodno da programski jezik ili okruženje za programiranje omoguće programski pristup do specijalizovanih hardverskih uređaja u koji je softver ugrađen i koji nadzire ili upravlja.
- RT sistemi su najčešće konkurentni programi, jer to odgovara uporednom, kontinualnom i asinhronom dešavanju promena, procesa i događaja u njihovom realnom okruženju. Iako RT sistem ne mora biti konkurentan, niti konkurentan sistem mora biti RT, te dve karakteristike vrlo često pojavljuju se zajedno. Zbog toga i koncepti za RT programiranje treba da obuhvataju koncepte za konkurentnost, jer je lakše modelovati konkurentne prirodne procese i asinhrono događaje tako, nego pomoću sekencijalnih koncepata. Drugim rečima, programiranje RT sistema najčešće podrazumeva principe i tehnike, ali i nosi probleme konkurentnog programiranja.
- RT sistemi često nadziru ili upravljaju analognim fizičkim veličinama iz svog okruženja, koje se u računarima mogu predstavljati samo diskretnim, racionalnim vrednostima koje zato predstavljaju aproksimacije analognih veličina ograničenog opsega i tačnosti. Algoritmi ugrađeni u RT softver moraju da uzmu u obzir ograničenu tačnost ovih aproksimacija i mogućnost neregularnih ili netačnih operacija sa racionalnim brojevima.
- RT sistemi vrlo često moraju da budu izuzetno pouzdani i sigurni, jer operišu u realnom okruženju pod različitim uticajima koji mogu da dovedu do otkaza ili neispravnog funkcionisanja, a koje može uzrokovati štetu ili ugrožavanje života i okoline. Zato oni često moraju imati ugrađene mehanizme i tehnike *tolerancije otkaza* (engl. *fault tolerance*).
- RT sistemi reaguju na uporedne događaje i promene u svom okruženju, ali i na protok vremena, pa RT programi moraju da koriste podršku jezika ili operativnog sistema za predstavu o protoku vremena i druge usluge vezane za realno vreme.
- RT sistemi treba da poštuju zadate vremenske rokove i druga vremenska ograničenja, kako bi svoj odziv davali pravovremeno i svoju funkciju obavljali kvalitetno. Za *soft* sisteme važne su *performanse*, dok je za *hard* sisteme važno obezbediti poštovanje vremenskih rokova u svim, pa i situacijama *najgoreg scenarija izvršavanja*.
- Varijante konfiguracije softvera za RT sisteme:
 - Aplikacija se instalira direktno na hardver, bez postojanja operativnog sistema. Aplikacija tada mora sama da preuzme sve funkcije upravljanja hardverskim i logičkim resursima (raspoređivanje procesora, alokaciju memorije, upravljanje drugim uređajima, komunikacija sa drugim sistemima, po potrebi i pristup fajlovima). Ovakvu konfiguraciju imaju samo krajnje jednostavni, po pravilu stariji sistemi.

-
- Aplikacija se instalira kao program na postojećem operativnom sistemu, koji onda obezbeđuje sve usluge upravljanja resursima. Današnji sistemi su uglavnom ovakvi, jer bi pravljenje svih funkcionalnosti operativnog sistema, uključujući i komunikacione protokole, pristup fajlovima i slično, i to na standardan način, interoperabilan sa drugim sistemima, bilo teško i neisplativo.
 - Operativni sistemi za RT sisteme:
 - Za *soft* sisteme dovoljni su operativni sistemi opšte namene, ukoliko poseduju osnovne usluge vezane za realno vreme (časovnik realnog vremena, merenje vremenskih intervala, vremenske kontrole, suspenziju na zadato vreme itd.).
 - Za *hard* sisteme potrebni su posebni operativni sistemi koji nemaju elemente koji unose nepredvidivost u vreme odziva, da bi analiza izvodljivosti bila uopšte moguća. Ovi sistemi moraju da imaju i druge potrebne usluge i zadovolje posebne uslove (posebni algoritmi raspoređivanja, kontrole prekoračenja vremenskih rokova i vremena izvršavanja).

Primeri struktura pogodnih za RT implementacije

Kolekcija implementirana kao dvostruko ulančana dinamička lista

Koncepcija

- Za mnoge primene u RT i drugim sistemima potrebna je struktura koja predstavlja kolekciju pokazivača na objekte nekog tipa. *Kolekcija* je linearna struktura elemenata u koju se elementi mogu ubacivati, iz koje se mogu izbacivati, i koji se mogu redom obilaziti.
- Jedna jednostavna implementacija oslanja se na dinamičku, dvostruko ulančanu listu, čiji su elementi strukture koje sadrže veze (pokazivače) prema susednim elementima i sam sadržaj (pokazivač na objekat u kolekciji). Ove strukture se dinamički alociraju i dealociraju prilikom umetanja i izbacivanja elemenata.

Implementacija

```
// Project:   Real-Time Programming
// Subject:   Data Structures
// Module:    Collection
// File:      collection.h
// Date:      October 2002
// Author:    Dragan Milicev
// Contents:
//           Class: Collection
//           CollectionIterator

#ifndef _COLLECTION_
#define _COLLECTION_

////////////////////////////////////
// class Collection
////////////////////////////////////

class Object;
class CollectionElement;
class CollectionIterator;

class Collection {
public:

    Collection ();
    ~Collection ();

    void      append (Object*);
    void      insert (Object*, int at=0);
    void      remove (Object*);
    Object*   remove (int at=0);
    Object*   removeFirst() { return remove(0); }
    Object*   removeLast()  { return remove(size()-1); }
    void      clear  ();
};
```

```

int      isEmpty ()      { return sz==0; }
int      size      ()    { return sz; }
Object*  first      ();
Object*  last       ();
Object*  itemAt     (int at);
int      location (Object*);

CollectionIterator* createIterator ();
CollectionIterator* getIterator ()    { return internalIterator; }

protected:

    void remove (CollectionElement*);

private:

    friend class CollectionIterator;
    CollectionElement* head;
    CollectionElement* tail;
    int sz;

    CollectionIterator* internalIterator;

};

////////////////////////////////////
// class CollectionIterator
////////////////////////////////////

class CollectionIterator {
public:

    CollectionIterator (Collection* c) : col(c), cur(0) { reset(); }

    void      reset() { if (col!=0) cur=col->head; }
    int      next ();

    int      isDone() { return cur==0; }
    Object*  currentItem();

private:

    Collection* col;
    CollectionElement* cur;

};

#endif

// Project:  Real-Time Programming
// Subject:  Data Structures
// Module:   Collection
// File:     collection.cpp
// Date:     October 2002

```

```

// Author:   Dragan Milicev
// Contents:
//          Class: Collection
//          CollectionIterator

#include "collection.h"

////////////////////////////////////
// class CollectionElement
////////////////////////////////////

class CollectionElement {
public:
    Object* cont;
    CollectionElement *prev, *next;

    CollectionElement (Object*);
    CollectionElement (Object*, CollectionElement* next);
    CollectionElement (Object*, CollectionElement* prev, CollectionElement*
next);
};

inline CollectionElement::CollectionElement (Object* e)
    : cont(e), prev(0), next(0) {}

inline CollectionElement::CollectionElement (Object* e, CollectionElement*
n)
    : cont(e), prev(0), next(n) {
    if (n!=0) n->prev=this;
}

inline CollectionElement::CollectionElement (Object* e, CollectionElement*
p, CollectionElement* n)
    : cont(e), prev(p), next(n) {
    if (n!=0) n->prev=this;
    if (p!=0) p->next=this;
}

////////////////////////////////////
// class Collection
////////////////////////////////////

Collection::Collection ()
    : head(0), tail(0), sz(0),
    internalIterator(new CollectionIterator(this))
{}

Collection::~~Collection () {
    clear();
    delete internalIterator;
}

```

```
void Collection::remove (CollectionElement* e) {
    if (e==0) return;
    if (e->next!=0) e->next->prev=e->prev;
    else tail=e->prev;
    if (e->prev!=0) e->prev->next=e->next;
    else head=e->next;
    if (internalIterator && internalIterator->currentItem()==e->cont)
        internalIterator->next();
    delete e;
    sz--;
}
```

```
void Collection::append (Object* e) {
    if (head==0) head=tail=new CollectionElement(e);
    else tail=new CollectionElement(e,tail,0);
    sz++;
}
```

```
void Collection::insert (Object* e, int at) {
    if (at<0 || at>size()) return;
    if (at==0) {
        head=new CollectionElement(e,head);
        if (tail==0) tail=head;
        sz++;
        return;
    }
    if (at==size()) {
        append(e);
        return;
    }
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    new CollectionElement(e,cur->prev,cur);
    sz++;
}
```

```
void Collection::remove (Object* e) {
    if (tail && tail->cont==e) {
        remove(tail);
        return;
    }
    for (CollectionElement* cur=head; cur!=0; cur=cur->next)
        if (cur->cont==e) remove(cur);
}
```

```
Object* Collection::remove (int at) {
    Object* ret = 0;
    if (at<0 || at>=size()) return 0;
    if (at==0) {
        ret = head->cont;
        remove(head);
        return ret;
    }
    if (at==size()-1) {
        ret = tail->cont;
```

```

        remove(tail);
        return ret;
    }
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    ret = cur->cont;
    remove(cur);
    return ret;
}

void Collection::clear () {
    for (CollectionElement* cur=head, *temp=0; cur!=0; cur=temp) {
        temp=cur->next;
        delete cur;
    }
    head=0;
    tail=0;
    sz=0;
    if (internalIterator) internalIterator->reset();
}

Object* Collection::first () {
    if (head==0) return 0;
    else return head->cont;
}

Object* Collection::last () {
    if (tail==0) return 0;
    else return tail->cont;
}

Object* Collection::itemAt (int at) {
    if (at<0 || at>=size()) return 0;
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    return cur->cont;
}

int Collection::location (Object* e) {
    int i=0;
    for (CollectionElement* cur=head; cur!=0; cur=cur->next, i++)
        if (cur->cont==e) return i;
    return -1;
}

CollectionIterator* Collection::createIterator () {
    return new CollectionIterator(this);
}

```

```

////////////////////////////////////
// class CollectionIterator
////////////////////////////////////

```

```
int CollectionIterator::next () {
    if (cur!=0) cur=cur->next;
    return !isDone();
}

Object* CollectionIterator::currentItem () {
    return cur?cur->cont:0;
}
```

Primer upotrebe

```
#include "collection.h"
#include <iostream.h>

class Object {
    //...
};

class X : public Object {
public:
    X(int ii) : i(ii) {}
    int i;
    //...
};

void main () {
    X* x1 = new X(1);
    X* x2 = new X(2);
    X* x3 = new X(3);
    X* x4 = new X(4);
    X* x5 = new X(5);

    Collection* coll = new Collection;
    coll->append(x1);
    coll->append(x2);
    coll->insert(x3);
    coll->insert(x4,2);
    X* x = (X*)coll->removeFirst();
    coll->append(x);
    coll->insert(x5,3);

    CollectionIterator* it = coll->getIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        X* x = (X*)it->currentItem();
        cout<<x->i<<" ";
    }
    cout<<"\n";

    Collection* col2 = new Collection;
    col2->append(x1);
    col2->append(x2);
    col2->append(x3);
    col2->append(x4);
    col2->append(x5);
    it = col2->getIterator();
    for (it->reset(); !it->isDone(); it->next()) {
```

```

        X* x = (X*)it->currentItem();
        cout<<x->i<<" ";
    }
    cout<<"\n";

    it = coll->createIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        X* x = (X*)it->currentItem();
        cout<<x->i<<" ";
        CollectionIterator* it = coll->createIterator();
        for (it->reset(); !it->isDone(); it->next()) {
            X* x = (X*)it->currentItem();
            cout<<x->i<<" ";
        }
        delete it;
        cout<<"\n";
    }
    delete it;
    cout<<"\n";

    coll->clear();
    delete coll;
    col2->clear();
    delete col2;
    delete x1;
    delete x2;
    delete x3;
    delete x4;
    delete x5;
}

```

Analiza kompleksnosti

- Neka je kompleksnost algoritma alokacije prostora ugrađenog alokatora (ugrađene operatorske funkcije `new`) C_a , a algoritma dealokacije C_d . Neka je n veličina kolekcije (broj elemenata u kolekciji). Tada je kompleksnost najznačajnijih operacija prikazane implementacije sledeća:

Operacija	Kompleksnost
<code>Collection::append(Object*)</code>	C_a
<code>Collection::insert(Object*)</code>	C_a
<code>Collection::insert(Object*, int)</code>	$C_a + O(n)$
<code>Collection::remove(Object*)</code>	$C_d + O(n)$
<code>Collection::remove(int)</code>	$C_d + O(n)$
<code>Collection::removeFirst()</code>	C_d
<code>Collection::removeLast()</code>	C_d
<code>Collection::clear()</code>	$C_d \cdot O(n)$
<code>Collection::isEmpty()</code>	$O(1)$
<code>Collection::size()</code>	$O(1)$
<code>Collection::first()</code>	$O(1)$
<code>Collection::last()</code>	$O(1)$
<code>Collection::itemAt(int)</code>	$O(n)$
<code>Collection::location(Object*)</code>	$O(n)$
<code>CollectionIterator::reset()</code>	$O(1)$
<code>CollectionIterator::next()</code>	$O(1)$

- Prema tome, najkritičnije i najčešće korišćene operacije za umetanje i izbacivanje elemenata jako zavise od kompleksnosti algoritma ugrađenog alokatora i dealokatora. U zavisnosti od implementacije struktura koje vode računa o zauzetom i slobodnom prostoru u dinamičkoj memoriji, ova kompleksnost može različitā. Tipično je to $O(k)$, gde je k broj zauzetih i/ili slobodnih segmenata dinamičke memorije, ili u boljem slučaju $O(\log k)$. Što je još gore, operacija `Collection::remove(Object*)` ima i dodatnu kompleksnost $O(n)$, zbog sekvencijalne pretrage elementa koji se izbacuje.
- Zbog ovakve orijentacije na ugrađene alokatore, čija se kompleksnost teško može proceniti i kontrolisati, kao i na sekvencijalnu pretragu kod izbacivanja, ova implementacija nije pogodna za RT sisteme.

Koncepcija

* Rešenje koje eliminiše ove probleme oslanja se na to da struktura veza bude ugrađena u same objekte koji se smeštaju u kolekciju. Zbog toga nema potrebe za alokaciju i dealokaciju struktura za veze.

* Potencijalni problem sa ovakvim pristupom je to što može doći do greške ukoliko se objekat koji je već element neke kolekcije pokuša ubaciti u drugu kolekciju, korišćenjem iste strukture za veze, čime dolazi do korupcije prve kolekcije. Zato je u ovu implementaciju ugrađena zaštita od ovakve pogrešne upotrebe.

[illegible]

```

class CollectionElement {
public:

    CollectionElement (Object* holder);

    Object*      getHolder()      { return holder; }
    Collection*  getContainer () { return container; }

private:

    friend class Collection;
    friend class CollectionIterator;
    void set (CollectionElement* prev, CollectionElement* next);
    void setContainer (Collection* col) { container = col; }

    CollectionElement *prev, *next;

    Collection* container;
    Object* holder;

};

inline CollectionElement::CollectionElement (Object* h)
    : container(0), holder(h), prev(0), next(0) {}

inline void CollectionElement::set (CollectionElement* p,
CollectionElement* n) {
    prev = p;
    next = n;
    if (n!=0) n->prev=this;
    if (p!=0) p->next=this;
}

////////////////////////////////////
// class Collection
////////////////////////////////////

class CollectionIterator;

class Collection {
public:

    Collection ();
    ~Collection ();

    void      append (CollectionElement*);
    void      insert (CollectionElement*, int at=0);
    void      insertBefore (CollectionElement* newElem, CollectionElement*
beforeThis);
    void      insertAfter  (CollectionElement* newElem, CollectionElement*
afterThis);

```

```

void      remove (CollectionElement*);
Object*   remove (int at=0);
Object*   removeFirst() { return remove(0); }
Object*   removeLast()  { return remove(size()-1); }

void      clear  ();

int       isEmpty ()      { return sz==0; }
int       size    ()      { return sz; }

Object*   first   () { return head->getHolder(); }
Object*   last    () { return tail->getHolder(); }
Object*   itemAt  (int at);

int       location(CollectionElement*);

CollectionIterator* createIterator ();
CollectionIterator* getIterator () { return internalIterator; }

private:

    friend class CollectionIterator;
    CollectionElement* head;
    CollectionElement* tail;
    int sz;

    CollectionIterator* internalIterator;

};

////////////////////////////////////
// class CollectionIterator
////////////////////////////////////

class CollectionIterator {
public:

    CollectionIterator (Collection* c) : col(c), cur(0) { reset(); }

    void      reset() { if (col!=0) cur=col->head; }
    int       next () { if (cur!=0) cur=cur->next; return !isDone(); }

    int       isDone() { return cur==0; }

    Object*   currentItem()      { return cur?cur->getHolder():0; }
    CollectionElement* currentElement() { return cur; }

private:

    Collection* col;
    CollectionElement* cur;

};

```



```

#endif

// Project:   Real-Time Programming
// Subject:   Data Structures
// Module:    Collection
// File:      collect.cpp
// Date:      October 2002
// Author:    Dragan Milicev
// Contents:
//      Class: CollectionElement
//      Collection
//      CollectionIterator

#include "collect.h"

////////////////////////////////////
// class Collection
////////////////////////////////////

Collection::Collection ()
: head(0), tail(0), sz(0),
  internalIterator(new CollectionIterator(this))
{}

Collection::~~Collection () {
    clear();
    delete internalIterator;
}

void Collection::append (CollectionElement* e) {
    if (e==0 || e->getContainer()!=0) return;
    if (head==0) {
        e->set(0,0);
        head=tail=e;
    }
    else {
        e->set(tail,0);
        tail=e;
    }
    e->setContainer(this);
    sz++;
}

void Collection::insert (CollectionElement* e, int at) {
    if (e==0 || e->getContainer()!=0 || at<0 || at>size()) return;
    if (at==0) {
        e->set(0,head);
        e->setContainer(this);
        head=e;
        if (tail==0) tail=head;
        sz++;
        return;
    }
    if (at==size()) {
        append(e);
        return;
    }
}

```

```

    }
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    e->set(cur->prev,cur);
    e->setContainer(this);
    sz++;
}

void Collection::insertBefore (CollectionElement* newElem,
CollectionElement* beforeThis) {
    if (newElem==0 || newElem->getContainer()!=0) return;
    if (beforeThis==0) { append(newElem); return; }
    if (beforeThis->prev==0) { insert(newElem); return; }
    newElem->set(beforeThis->prev,beforeThis);
    newElem->setContainer(this);
    sz++;
}

void Collection::insertAfter (CollectionElement* newElem,
CollectionElement* afterThis) {
    if (newElem==0 || newElem->getContainer()!=0) return;
    if (afterThis==0) { insert(newElem); return; }
    if (afterThis->next==0) { append(newElem); return; }
    newElem->set(afterThis,afterThis->next);
    newElem->setContainer(this);
    sz++;
}

void Collection::remove (CollectionElement* e) {
    if (e==0 || e->getContainer()!=this) return;
    if (e->next!=0) e->next->prev=e->prev;
    else tail=e->prev;
    if (e->prev!=0) e->prev->next=e->next;
    else head=e->next;
    e->set(0,0);
    e->setContainer(0);
    if (internalIterator && internalIterator->currentItem()==e->getHolder())
        internalIterator->next();
    sz--;
}

Object* Collection::remove (int at) {
    CollectionElement* ret = 0;
    if (at<0 || at>=size()) return 0;
    if (at==0) {
        ret = head;
        remove(head);
        return ret?ret->getHolder():0;
    }
    if (at==size()-1) {
        ret = tail;
        remove(tail);
        return ret?ret->getHolder():0;
    }
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    ret = cur;

```

```

    remove(cur);
    return ret?ret->getHolder():0;
}

void Collection::clear () {
    for (CollectionElement* cur=head, *temp=0; cur!=0; cur=temp) {
        temp=cur->next;
        cur->set(0,0);
        cur->setContainer(0);
    }
    head=0;
    tail=0;
    sz=0;
    if (internalIterator) internalIterator->reset();
}

Object* Collection::itemAt (int at) {
    if (at<0 || at>=size()) return 0;
    int i=0;
    for (CollectionElement* cur=head; i<at; cur=cur->next, i++);
    return cur?cur->getHolder():0;
}

int Collection::location (CollectionElement* e) {
    if (e==0 || e->getContainer()!=this) return -1;
    int i=0;
    for (CollectionElement* cur=head; cur!=0; cur=cur->next, i++)
        if (cur==e) return i;
    return -1;
}

CollectionIterator* Collection::createIterator () {
    return new CollectionIterator(this);
}

```

Primer upotrebe

```

#include "collect.h"
#include <iostream.h>

class Object {
    //...
};

class X : public Object {
public:
    X(int ii) : i(ii), ceForC1(this), ceForC2(this) {}
    int i;

    CollectionElement ceForC1;
    CollectionElement ceForC2;
    //...
};

void main () {
    X* x1 = new X(1);
}

```

```

X* x2 = new X(2);
X* x3 = new X(3);
X* x4 = new X(4);
X* x5 = new X(5);

Collection* coll = new Collection;
coll->append(&x1->ceForC1);
coll->append(&x2->ceForC1);
coll->insert(&x3->ceForC1);
coll->insert(&x4->ceForC1, 2);
X* x = (X*)coll->removeFirst();
coll->append(&x->ceForC1);
coll->insert(&x5->ceForC1, 3);

CollectionIterator* it = coll->getIterator();
for (it->reset(); !it->isDone(); it->next()) {
    X* x = (X*)it->currentItem();
    cout<<x->i<<" ";
}
cout<<"\n";

Collection* col2 = new Collection;
col2->append(&x1->ceForC2);
col2->append(&x2->ceForC2);
col2->append(&x3->ceForC2);
col2->append(&x4->ceForC2);
col2->append(&x5->ceForC2);
col2->append(&x3->ceForC1); // Tolerant Error
it = col2->getIterator();
for (it->reset(); !it->isDone(); it->next()) {
    X* x = (X*)it->currentItem();
    cout<<x->i<<" ";
}
cout<<"\n";

it = coll->createIterator();
for (it->reset(); !it->isDone(); it->next()) {
    X* x = (X*)it->currentItem();
    cout<<x->i<<" ";
    CollectionIterator* it2 = coll->createIterator();
    for (it2->reset(); !it2->isDone(); it2->next()) {
        X* x2 = (X*)it2->currentItem();
        cout<<x2->i<<" ";
    }
    delete it2;
    cout<<"\n";
}
delete it;
cout<<"\n";

coll->clear();
delete coll;
col2->clear();
delete col2;
delete x1;
delete x2;
delete x3;
delete x4;
delete x5;
}

```

Analiza kompleksnosti

- Kompleksnost ove implementacije više ne zavisi od kompleksnosti algoritma alokatora i dealokatora. Kompleksnost najznačajnijih operacija je sada značajno smanjena, naročito za najčešće operacije stavljanja i izbacivanja elementa, i to na početak ili kraj:

Operacija	Kompleksnost
<code>Collection::append(CollectionElement*)</code>	$O(1)$
<code>Collection::insert(CollectionElement*)</code>	$O(1)$
<code>Collection::insert(CollectionElement*, int)</code>	$O(n)$
<code>Collection::insertBefore(...)</code>	$O(1)$
<code>Collection::insertAfter(...)</code>	$O(1)$
<code>Collection::remove(CollectionElement*)</code>	$O(1)$
<code>Collection::remove(int)</code>	$O(n)$
<code>Collection::removeFirst()</code>	$O(1)$
<code>Collection::removeLast()</code>	$O(1)$
<code>Collection::clear()</code>	$O(n)$
<code>Collection::isEmpty()</code>	$O(1)$
<code>Collection::size()</code>	$O(1)$
<code>Collection::first()</code>	$O(1)$
<code>Collection::last()</code>	$O(1)$
<code>Collection::itemAt(int)</code>	$O(n)$
<code>Collection::location(CollectionElement*)</code>	$O(n)$
<code>CollectionIterator::reset()</code>	$O(1)$
<code>CollectionIterator::next()</code>	$O(1)$
<code>CollectionIterator::isDone()</code>	$O(1)$
<code>Collection::currentItem()</code>	$O(1)$
<code>Collection::currentElement()</code>	$O(1)$

FIFO Red*Koncepcija*

- FIFO (*First-In First-Out*) red (engl. *queue*) je struktura u koju se elementi mogu umetati i vaditi, ali sa sledećim protokolom: operacija vađenja elementa uvek vraća element koji je najdavnije stavljen u red, tako da se elementi vade po redosledu stavljanja.

Implementacija

- * Implementacija se u potpunosti oslanja na realizovanu kolekciju:

```
// Project:  Real-Time Programming
// Subject:  Data Structures
// Module:   FIFO Queue
// File:     queue.h
// Date:     October 2002
// Author:   Dragan Milicev
// Contents:
//          Class:
//          Queue
```

```
#ifndef _QUEUE_
#define _QUEUE_
```

```
#include "collect.h"

////////////////////////////////////
// class Queue
////////////////////////////////////

class Queue {
public:

    void    put (CollectionElement* e) { col.append(e); }
    Object* get ()                    { return col.removeFirst(); }
    void    clear ()                  { col.clear(); }

    int     isEmpty ()                { return col.isEmpty(); }
    int     isFull ()                 { return 0; }
    int     size ()                   { return col.size(); }

    Object* first ()                  { return col.first(); }
    Object* last ()                   { return col.last(); }
    Object* itemAt (int at)           { return col.itemAt(at); }

    int     location(CollectionElement* e) { return col.location(e); }

    CollectionIterator* createIterator () { return col.createIterator(); }
    CollectionIterator* getIterator ()    { return col.getIterator(); }

private:

    Collection col;

};

#endif
```

Primer upotrebe

- Najčešće upotrebljavane operacije ove apstrakcije su operacije stavljanja (`put()`) i uzimanja elementa (`get()`). Data klasa koristi se slično kao i ranije realizovana kolekcija.

Analiza kompleksnosti

- Kako se sve operacije ove klase svode na odgovarajuće operacije klase `Collection`, time je i njihova kompleksnost identična. Treba uočiti da je kompleksnost najčešće korišćenih operacija `put()` i `get()` jednaka $O(1)$.

Red sa prioritetom

Koncepcija

- *Red sa prioritetom* (engl. *priority queue*) je linearna struktura u koju se elementi mogu smeštati i izbacivati, ali pri čemu elementi imaju svoje *prioritete*. Prioritet je veličina koja se može upoređivati (tj. za koju je definisana neka relacija totalnog uređenja, na primer prirodan broj).
- Najkritičnije operacije ove strukture su operacije smeštanja i izbacivanja elementa i operacija vraćanja (bez izbacivanja) elementa koji ima trenutno najviši prioritet.

Implementacija

- Jedna moguća, ovde prikazana implementacija se oslanja na postojeću implementaciju kolekcije.
- Da bi kompleksnost obe operacije vraćanja trenutno najprioritetnijeg elementa i operacije umetanja elementa bila manja od $O(n)$, potrebno je da nijedna od njih ne uključuje linearnu pretragu po eventualno uređenoj listi. Zbog toga ova implementacija sadrži pokazivač na trenutno najprioritetniji element, koji se ažurira prilikom promene strukture reda ili promene prioriteta nekog elementa.

```
// Project: Real-Time Programming
// Subject: Data Structures
// Module: Priority Queue
// File: pqueue.h
// Date: October 2002
// Author: Dragan Milicev
// Contents:
//     Class:
//         PriorityElement
//         PriorityQueue
//     Type:
//         Priority

#ifndef _PQUEUE_
#define _PQUEUE_

#include "collect.h"

/////////////////////////////////////////////////////////////////
// class PriorityElement
/////////////////////////////////////////////////////////////////

typedef unsigned int Priority;
const Priority MinPri = 0;

class PriorityQueue;

class PriorityElement : public CollectionElement {
public:
    PriorityElement (Object* holder, Priority p = 0)
        : CollectionElement(holder), pri(p), container(0) {}

    Priority getPriority () { return pri; }
    void setPriority (Priority newPri);

    PriorityQueue* getContainer () { return container; }

private:
    Priority pri;

    friend class PriorityQueue;
    void setContainer (PriorityQueue* c) { container = c; }
    PriorityQueue* container;
};
```

```

/////////////////////////////////////////////////////////////////
// class PriorityQueue
/////////////////////////////////////////////////////////////////

class PriorityQueue {
public:

    PriorityQueue () : col(), highest(0) {}

    Object* first ()    { return highest?highest->getHolder():0; }

    void add (PriorityElement*);
    void remove(PriorityElement*);
    void clear  ();

    void notifyPriorityChange (PriorityElement*);

    int  isEmpty () { return col.isEmpty(); }
    int  size   () { return col.size(); }

    CollectionIterator* createIterator ()    { return col.createIterator(); }
    CollectionIterator* getIterator  ()      { return col.getIterator(); }

private:

    Collection col;
    PriorityElement* highest;

};

#endif

// Project:  Real-Time Programming
// Subject:  Data Structures
// Module:   Priority Queue
// File:     pqueue.cpp
// Date:     October 2002
// Author:   Dragan Milicev
// Contents:
//          Class:
//              PriorityElement
//              PriorityQueue

#include "pqueue.h"

/////////////////////////////////////////////////////////////////
// class PriorityElement
/////////////////////////////////////////////////////////////////

void PriorityElement::setPriority (Priority newPri) {
    if (pri==newPri) return;
    pri = newPri;
    if (container!=0) container->notifyPriorityChange(this);
}

```



```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// class PriorityQueue
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void PriorityQueue::add (PriorityElement* e) {
    if (e==0 || e->getContainer()!=0) return;
    col.append(e);
    e->setContainer(this);
    notifyPriorityChange(e);
}

void PriorityQueue::remove(PriorityElement* e) {
    if (e==0 || e->getContainer()!=this) return;
    col.remove(e);
    e->setContainer(0);
    if (highest!=e) return;

    Priority maxPri = MinPri;
    highest = 0;
    CollectionIterator* it = getIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        PriorityElement* pe = (PriorityElement*)it->currentElement();
        if (pe->getPriority()>=maxPri) {
            maxPri = pe->getPriority();
            highest = pe;
        }
    }
}

void PriorityQueue::clear () {
    CollectionIterator* it = getIterator();
    for (it->reset(); !it->isDone(); it->next()) {
        PriorityElement* pe = (PriorityElement*)it->currentElement();
        pe->setContainer(0);
    }
    col.clear();
    highest = 0;
}

void PriorityQueue::notifyPriorityChange (PriorityElement* e) {
    if (e==0 || e->getContainer()!=this) return;
    if (highest==0 || highest->getPriority()<e->getPriority()) {
        highest = e;
        return;
    }
    if (highest==e) {
        Priority maxPri = e->getPriority();
        CollectionIterator* it = getIterator();
        for (it->reset(); !it->isDone(); it->next()) {
            PriorityElement* pe = (PriorityElement*)it-
>currentElement();
            if (pe->getPriority()>maxPri) {
                maxPri = pe->getPriority();
                highest = pe;
            }
        }
    }
}

```

```

        }
        return;
    }
}

```

Primer upotrebe

```

#include "pqueue.h"
#include <iostream.h>

class Object {
    //...
};

class X : public Object {
public:

    X(int ID, Priority pri) : id(ID), peForPQ(this)
    { peForPQ.setPriority(pri); }
    int id;

    PriorityElement* getPriorityElement () { return &peForPQ; }

    Priority getPriority () { return
peForPQ.getPriority(); }
    void setPriority (Priority pri) { peForPQ.setPriority(pri); }

private:

    PriorityElement peForPQ;

    //...
};

void main () {
    X* x1 = new X(1,1);
    X* x2 = new X(2,2);
    X* x3 = new X(3,3);
    X* x4 = new X(4,4);
    X* x5 = new X(5,5);

    PriorityQueue* pq = new PriorityQueue;
    pq->add(x1->getPriorityElement());
    pq->add(x3->getPriorityElement());
    pq->add(x4->getPriorityElement());
    pq->add(x2->getPriorityElement());
    pq->add(x5->getPriorityElement());

    X* top = 0;

    top = (X*)pq->first();
    cout<<top->getPriority()<<" "<<top->id<<"\n";

    x1->setPriority(6);
    top = (X*)pq->first();
    cout<<top->getPriority()<<" "<<top->id<<"\n";

    x5->setPriority(3);
    x1->setPriority(0);
    top = (X*)pq->first();
}

```

```

cout<<top->getPriority()<<" "<<top->id<<"\n";

pq->remove(top->getPriorityElement());
top = (X*)pq->first();
cout<<top->getPriority()<<" "<<top->id<<"\n";

CollectionIterator* it = pq->getIterator();
for (it->reset(); !it->isDone(); it->next()) {
    X* x = (X*)it->currentItem();
    cout<<x->id<<" ";
}
cout<<"\n";

pq->clear();
delete pq;
delete x1;
delete x2;
delete x3;
delete x4;
delete x5;
}

```

Analiza kompleksnosti

Operacija	Kompleksnost
PriorityElement::getPriority()	$O(1)$
PriorityElement::setPriority()	Najčešći slučaj: $O(1)$ Najgori slučaj: $O(n)$
PriorityQueue::first()	$O(1)$
PriorityQueue::add(PriorityElement*)	$O(1)$
PriorityQueue::remove(PriorityElement*)	Najgori slučaj: $O(n)$
PriorityQueue::clear()	$O(n)$
PriorityQueue::notifyPriorityChange(PriorityElement*)	Najgori slučaj: $O(n)$
PriorityQueue::isEmpty()	$O(1)$
PriorityQueue::size()	$O(1)$

Efikasna alokacija memorije

Koncepcija

- Intenzivno korišćenje dinamičkih objekata u OOP podrazumeva oslanjanje na algoritme alokacije i dealokacije ugrađenih menadžera dinamičke memorije. Kao što je već rečeno, kompleksnost ovih algoritama je tipično $O(n)$ i $O(1)$, odnosno $O(\log n)$ za obe operacije u najboljem slučaju. Zbog toga kreiranje i uništavanje dinamičkih objekata klasa može da bude režijski jako skupo u RT programima.
- Osim toga, dugotrajno korišćenje (tipično za ugrađene sisteme) jedinstvene dinamičke memorije za objekte svih klasa može da dovede do fragmentacije dinamičke memorije.
- Ideja jednog mogućeg rešenja oba problema zasniva se na tome da se dealocirani objekti ne vraćaju na korišćenje ugrađenom menadžeru, već se čuvaju u listi pridruženoj datoj klasi kako bi se njihov prostor ponovo koristio za nove objekte iste klase ("reciklaža"). Za neku klasu X za koju se želi brza alokacija i dealokacija objekata, treba obezbediti jedan objekat klase *RecycleBin*. Ovaj objekat funkcioniše tako što u svojoj listi čuva sve dealocirane objekte date klase. Kada se traži alokacija novog objekta, *RecycleBin* prvo vadi iz svoje liste već spreman reciklirani objekat, tako da je alokacija u tom slučaju

veoma brza. Ako je lista prazna, tj. ako nema prethodno recikliranih objekata date klase, *RecycleBin* pri alokaciji koristi ugrađeni alokator *new*. Pri dealokaciji, objekat se ne oslobađa već se reciklira, tj. upisuje se u listu klase *RecycleBin*.

Implementacija

```
// Project:   Real-Time Programming
// Subject:   Data Structures
// Module:    Recycle Bin
// File:      recycle.h
// Date:      October 2002
// Author:    Dragan Milicev
// Contents:
//      Class:
//          RecycleBin
//      Macro:
//          RECYCLE_DEC(X)
//          RECYCLE_DEF(X)
//          RECYCLE_CON(X)

#ifndef _RECYCLE_
#define _RECYCLE_

#include <stdlib.h>
#include "collect.h"

////////////////////////////////////////////////////////////////
// class RecycleBin
////////////////////////////////////////////////////////////////

class RecycleBin {
public:

    void  recycle (CollectionElement* e) { col.append(e); }
    void* getNew  (size_t size);

    int    isEmpty ()                { return col.isEmpty(); }
    int    size    ()                { return col.size(); }

private:

    Collection col;

};

////////////////////////////////////////////////////////////////
// macro RECYCLE_DEC(X)
////////////////////////////////////////////////////////////////

#define RECYCLE_DEC(X) \
    private: static RecycleBin myRecycleBin; \
    public: static void* operator new (size_t); \
    public: static void operator delete (void*); \
    private: CollectionElement forRecycleBin;
```

```

/////////////////////////////////////////////////////////////////
// macro RECYCLE_CON
/////////////////////////////////////////////////////////////////

#define RECYCLE_CON(X) \
    forRecycleBin(this)

/////////////////////////////////////////////////////////////////
// macro RECYCLE_DEF(X)
/////////////////////////////////////////////////////////////////

#define RECYCLE_DEF(X) \
    RecycleBin X::myRecycleBin; \
    \
    void* X::operator new (size_t sz) { \
        return myRecycleBin.getNew(sz); \
    } \
    \
    void X::operator delete (void* p) { \
        myRecycleBin.recycle(&((X*)p)->forRecycleBin); \
    }

#endif
// Project:   Real-Time Programming
// Subject:   Data Structures
// Module:    Recycle Bin
// File:      recycle.cpp
// Date:      October 2002
// Author:    Dragan Milicev
// Contents:
//           Class:
//           RecycleBin

#include "recycle.h"

/////////////////////////////////////////////////////////////////
// class RecycleBin
/////////////////////////////////////////////////////////////////

void* RecycleBin::getNew (size_t size) {
    Object* obj = col.removeFirst();
    if (obj!=0) return obj;
    return new char[size];
}

```

Primer upotrebe

```

#include "recycle.h"

class Object {};

class Y : public Object {
public:
    Y() : RECYCLE_CON(Y) {}
    int i,j,k;

```

```
        RECYCLE_DEC(Y)
};

// To be put into a .cpp file:
RECYCLE_DEF(Y);

void main () {
    Y* p1 = new Y;
    Y* p2 = new Y;
    delete p1;
    Y* p3 = new Y;
    delete p3;
    delete p2;
}
```

Analiza kompleksnosti

- Implementacija klase `RecycleBin` se oslanja na implementaciju kolekcije. U većini slučajeva, posle dovoljno dugog rada programa i ulaska u stacionarni režim kreiranja i uništavanja dinamičkih objekata neke klase, logično je očekivati da se alokacija objekta svodi na uzimanje elementa iz kolekcije, što je izuzetno efikasno. U najgorem slučaju, alokacija se svodi na ugrađeni algoritam alokatora. Dealokacija se u svakom slučaju svodi na ubacivanje elementa u kolekciju:

Operacija	Kompleksnost u najčešćem slučaju	Kompleksnost u najgorem slučaju
<code>X::new()</code> / <code>RecycleBin::getNew()</code>	$O(1)$	C_a
<code>X::delete()</code> / <code>RecycleBin::recycle()</code>	$O(1)$	$O(1)$

II Pouzdanost i tolerancija otkaza

Pouzdanost i tolerancija otkaza

- RT sistemi obično moraju da ispunjavaju mnogo strožije zahteve u pogledu pouzdanosti nego druge aplikacije. Na primer, interaktivni uslužni program za obradu teksta, ukoliko se dogodi bilo kakav otkaz, može jednostavno da se prekine (nasilno ugasi) i pokrene iznova, uz ograničenu štetu u obliku izgubljenog dela unetog teksta. Sa druge strane, ne treba naglašavati značaj pouzdanosti sistema za upravljanje letom aviona ili onog za signalizaciju u šinskom saobraćaju, ili za neki opasan industrijski proces. Takav sistem ne sme sebi da dozvoli da tek tako prestane da radi zbog neke greške, odnosno otkaza. Umesto toga, on treba da nastavi da radi bezbedno, eventualno sa degradiranom funkcionalnošću i/ili performansama, ili da preduzme kontrolisanu operaciju gašenja celog sistema, kako ne bi došlo do opasnih posledica.
- Softver se „ne kvari“ vremenom i kao posledica upotrebe, ali izvori grešaka u njegovom izvršavanju mogu biti:
 - Nekorektna specifikacija softvera: na osnovu nepravilne specifikacije će čak i ispravna implementacija dovesti do nepouzdanog sistema.
 - Greške u realizaciji softverskih komponenata, unete u samom razvoju tog softvera.
 - Otkazi procesorskih i drugih hardverskih komponenata u sistemu, npr. otkaz memorijskog modula, ulazno-izlaznog uređaja i slično.
 - Tranzijentni ili permanentni uticaji na komunikacioni podsistem, koji dovode do povremenog ili trajnog prekida u prenosu poruka ili grešaka u prenetim porukama (gubitak i izobličenje poruka).
 - Neadekvatno rukovanje sistemom od strane korisnika, a koje sistem nije predvideo i za koje nije napravljena adekvatna zaštita i reakcija.

Pouzdanost, padovi i otkazi

- *Pouzdanost* (engl. *reliability*) je stepen uspešnosti kojom se sistem pridržava autoritativne specifikacije svog ponašanja. Ta specifikacija bi, u idealnom slučaju, trebalo da bude kompletna, konzistentna, razumljiva i nedvosmislena.
- Za RT sisteme, važan deo te specifikacije jesu vremena odziva sistema na spoljašnje događaje i procese, a koja se određuju tokom analize problema i procesa koje RT sistem treba da kontroliše. Ako ova analiza nije dobra, i određeni vremenski parametri neće biti korektni, pa ni sistem neće biti pouzdan, jer će potencijalno davati neblagovremene odzive.
- *Pad* (engl. *failure*) je slučaj u kom ponašanje sistema odstupa od navedene specifikacije.
- Pad je spoljašnja manifestacija ponašanja prouzrokovana internim *greškama* (engl. *error*) u sistemu. Fizički ili softverski uzroci grešaka nazivaju se *otkazi* (engl. *fault*).
- Složeni sistemi se sastoje iz komponenata koje su opet složeni podsistemi. Zbog toga se otkaz u jednom podsistemu manifestuje kao njegova greška, ova kao pad tog podsistema, a to opet kao otkaz na višem nivou hijerarhije itd.
- Prema vremenu pojavljivanja i trajanja, otkazi mogu biti:
 - Permanentni (engl. *permanent faults*) su otkazi koji nastaju u nekom trenutku i ostaju prisutni u sistemu sve dok se ne uklone popravkom sistema. Primeri: prekinut kontakt, trajno pregorevanje uređaja ili greška u softveru.

- Tranzijentni (engl. *transient faults*) su otkazi koji nastaju u nekom trenutku, postoje neko vreme i onda nestaju. Primer je otkaz hardverske komponente koja reaguje na prisutnost spoljašnjeg elektromagnetskog zračenja ili vibracija. Kada uzrok otkaza nestane, nestaje i greška.
- Intermitentni (engl. *intermittent faults*) su tranzijentni otkazi koji se ponavljaju, odnosno dešavaju s vremena na vreme. Primer je hardverska komponenta (npr. kontakt) koja je osetljiva na zagrevanje; ona radi neko vreme i kada se zagreje, isključi se, zatim se ohladi i ponovo uključi itd.

Sprečavanje i tolerancija otkaza

- Dva osnovna pristupa povećanju pouzdanosti su sprečavanje i tolerancija otkaza.
- *Sprečavanje otkaza* (engl. *fault prevention*) je strategija koja tokom projektovanja i izgradnje sistema pokušava da eliminiše mogućnost pojave otkaza tog sistema; ovo podrazumeva aktivnosti kojima se greške eliminišu i pre nego što se on stavi u pogon i postane operativan.
- *Tolerancija otkaza* (engl. *fault tolerance*) je strategija koja omogućava da sistem nastavi da funkcioniše i u prisustvu otkaza, uz eventualno degradiranu funkcionalnost i/ili performanse, ili da preduzme akcije za njegovo kontrolisano gašenje, kako bi se sprečile loše posledice koje bi u suprotnom mogle da nastanu.

Sprečavanje otkaza

- Dva stepena sprečavanja otkaza: *izbegavanje otkaza* (engl. *fault avoidance*) i *otklanjanje otkaza* (engl. *fault removal*).
- Izbegavanje otkaza podrazumeva primenu principa i tehnika kojima se teži da se izbegne ili barem ograniči unošenje grešaka tokom konstrukcije samog sistema:
 - korišćenje najkvalitetnijih i najpouzdanijih hardverskih i gotovih softverskih komponenata u okviru datih ograničenja u pogledu cene i performansi;
 - korišćenje pouzdanih tehnika za interkonekciju komponenata i sklapanje podsistema;
 - pakovanje i zaštita hardvera tako da bude zaštićen od spoljašnjih štetnih uticaja (npr. elektromagnetskog zračenja, vibracija, vlage, prašine);
 - rigorozna ili čak potpuno formalna specifikacija zahteva;
 - upotreba dokazanih i pouzdanih, a poželjno i rigoroznih, formalnih metoda projektovanja softvera i jezika koji podržavaju apstrakciju, modularnost i strogu enkapsulaciju;
 - upotreba formalnih metoda verifikacije softvera za formalno dokazivanje konzistentnosti specifikovanih uslova i zahteva, kao i njihovog ispunjenja od strane implementiranog softvera;
 - upotreba alata za softversko inženjerstvo pri projektovanju i testiranju, npr. alata za modelovanje softvera, automatsko generisanje koda, formalnu verifikaciju softvera, automatsko testiranje softvera itd;
 - upotreba dokazanih i strogo upravljanih procesa razvoja softvera, počev od analize i specifikacije, preko razvoja, testiranja, dokumentacije i kontrole konfiguracije (engl. *configuration control*).
- Jedan matematički formalizam za specifikaciju i verifikaciju vremenskih ograničenja i uslova jeste *temporalna logika* (*temporal logic*): kvantifikatorska predikatska logika proširena kvantifikatorima i operatorima koji se odnose na vremenske odnose. Njome se

može proveravati konzistentnost specifikacije vremenskih zahteva (nepostojanje kontradikcija).

- Uprkos svim ovim naporima, greške u dizajnu i implementaciji hardvera i softvera uvek će biti moguće.
- Otklanjanje otkaza podrazumeva procedure za pronalaženje a potom i uklanjanje grešaka u hardveru i softveru, pre njegovog stavljanja u pogon. Te procedure su npr. pregled projekta, pregled koda (engl. *code review*), verifikacija programa i testiranje sistema.
- Za *hard* RT sisteme, jedna formalna metoda verifikacije ispunjenja vremenskih zahteva jeste *analiza rasporedivosti* (engl. *schedulability analysis*).
- Za *soft* RT sisteme, jedan matematički formalizam za analizu performansi jeste *teorija redova čekanja* (engl. *queuing theory*).
- Nažalost, formalne metode specifikacije, dizajna i verifikacije softvera, iako veoma pouzdane, po pravilu nisu skalabilne, tj. primenjive za velike sisteme i projekte, pa su vrlo nepraktične i teško primenjive ili sasvim neprimenjive u praksi.
- Osim toga, sve ove metode verifikacije vremenskih zahteva i performansi podrazumevaju poznavanje određenih vremenskih parametara izvršavanja programa, kao što je npr. vreme izvršavanja delova programa u najgorem slučaju. Ove parametre je nemoguće uvek pouzdano proceniti i u toj proceni moguće su greške.
- Testiranje sistema, iako važna i obavezna procedura tokom razvoja softvera, nikada ne može biti potpuno iscrpno da otkrije sve greške zato što:
 - test nikada ne može da dokaže da grešaka nema, nego samo da ih otkrije ako ih ima (a uvek ih ima!);
 - RT sisteme je često nemoguće testirati u realnim uslovima: nemoguće je obezbediti realno okruženje ili vanredne realne okolnosti;
 - zato se većina testova vrši na sistemu koji radi u simulacionom režimu, a teško je garantovati da je simulacija adekvatna realnim uslovima;
 - testiranje i simulacija često podrazumevaju aktivaciju delova softvera (npr. za merenje performansi, praćenje izvršavanja, beleženje traga izvršavanja) koje unose dodatne režijske troškove (vreme i memoriju) koje onda čini simulaciju ili neizvodljivom, ili neadekvatnom;
 - greške uzrokovane pogrešnom specifikacijom zahteva se ne mogu manifestovati sve dok sistem ne postane operativan.
- Greške koje nisu otkrivene verifikacijom i testiranjem se po pravilu ne manifestuju dok se ne dogodi neka specijalna situacija u eksploataciji.
- I pored upotrebe svih tehnika softverske verifikacije i testiranja, hardverske komponente će otkazivati u radu, a greške u softveru će se provlačiti.
- Zbog svega toga, da bi sistem u eksploataciji bio pouzdan, neophodna je i *tolerancija otkaza* (engl. *fault tolerance*).

Tolerancija otkaza

- *Tolerantnost na otkaze* (engl. *fault tolerance*) je svojstvo sistema koje mu omogućava da u slučaju otkaza, odnosno pada neke njegove komponente, umesto da kao celina potpuno i nekontrolisano otkaze,
 - nastavi da funkcioniše, eventualno sa degradiranim funkcionalnostima, dostupnošću (engl. *availability*) i/ili performansama, tipično smanjenjem propusnosti (engl. *throughput*) ili povećanjem kašnjenja (engl. *latency*); ovakav nastavak funkcionisanja naziva se *blagonaklona degradacija* (engl. *graceful degradation*), ili

- preduzme kontrolisan i bezopasan proces gašenja, ukoliko je to prihvatljivo (engl. *graceful shutdown, fail gracefully, fail safe*).
- Tehnike tolerancije otkaza podrazumevaju da u sistemu ne postoji tzv. *jedinstvena tačka otkaza* (engl. *single point of failure*), odnosno zahtevaju *redundansu* (engl. *redundancy*) – postojanje dodatnih hardverskih i softverskih komponenata ili kopija podataka uvedenih u sistem radi detekcije i oporavka od otkaza. Ovi elementi ne bi bili potrebni u idealnom sistemu, odnosno kada se otkazi ne bi dešavali.
- Uvođenje redundanse po pravilu ima negativne posledice po sistem:
 - dodatne hardverske komponente ili softverske (algoritamske) provere korektnosti sa ciljem detekcije i korekcije grešaka unose dodatna kašnjenja i smanjuju propusnost i povećavaju zauzeće memorije;
 - dodatne hardverske i softverske komponente usložnjavaju njegovo projektovanje i izvođenje i povećavaju njegovu cenu;
 - redundandne kopije podataka utiču na trajanje operacija ažuriranja ili zahtevaju smanjen nivo njihove konzistentnosti (engl. *consistency*).
- Prilikom projektovanja sistema, cilj je minimizovati redundansu a maksimizovati pouzdanost, pod zadatim ograničenjima u pogledu zahtevanih performansi, konzistentnosti, pouzdanosti i cene.
- Pristupi u hardverskoj redundansi: statička i dinamička redundansa.
- Statička redundansa podrazumeva da su pri konstrukciji sistema uvedene dodatne hardverske komponente koje treba da sakriju efekte grešaka, odnosno prevaziđu otkaze.
- Jedan poznat primer statičke redundanse je tzv. *trostruko modularna redundansa* (engl. *triple modular redundancy, TMR*), ili u opštem slučaju *N-modularna redundansa* (NMR): tri identične hardverske komponente (ili *N* takvih) rade nezavisno i proizvode svaka svoj rezultat; posebna komponenta upoređuje proizvedene rezultate i usvaja onaj većinski, ukoliko se proizvedeni rezultati razlikuju; rezultat koji odstupa se odbacuje, a komponenta koja ga je proizvela smatra se privremeno ili trajno otkazanom. Ovaj pristup pretpostavlja da greška nije zajednička (npr. greška u dizajnu) nego je npr. tranzijentna. Detekcija greške iz više od jedne komponente zahteva NMR.
- Dinamička redundansa podrazumeva elemente i tehnike koje se ugrađuju u komponentu kako bi se pojava greške mogla *detektovati* (engl. *error detection*) za vreme funkcionisanja komponente. Primeri: razne tehnike provere korektnosti i detekcije grešaka u podacima, pri prenosu ili skladištenju (memorija, disk), kao što su bit parnosti (engl. *parity bit*), kontrolne sume (engl. *checksums*). Ove tehnike omogućavaju samo detekciju, ne i korekciju grešaka.
- Tehnike za *korekciju grešaka* (engl. *error correction*) omogućavaju i sakrivanje postojanja greške nakon njene detekcije. Primer: CRC kodovi (engl. *cyclic redundancy check*).
- Ukoliko postoje ugrađene tehnike detekcije grešaka za neku komponentu (npr. jednostavno izostanak odziva komponente u očekivanom, ograničenom vremenu), samo dvostruka statička redundansa može da obezbedi toleranciju otkaza. Dve identične komponente mogu da rade u različitim režimima:
 - obe komponente rade ravnopravno i raspoređuju opterećenje (engl. *workload*) tokom normalnog rada, dok su obe ispravne; ako jedna otkaze, druga preuzima celo opterećenje i nastavlja rad bez prekida;
 - jedna, glavna komponenta obrađuje svo opterećenje i u regularnom radu, dok je druga komponenta samo redundantna rezerva koja preuzima opterećenje samo u slučaju da glavna otkaze (engl. *fail over*); ova rezerva u normalnom režimu (dok je glavna ispravna) može raditi na dva načina: kao *vruća rezerva* (engl. *hot standby*), što znači da prati i prima sve zahteve upućene glavnoj komponenti i održava isto stanje kao i glavna komponenta, kako bi što pre preuzela opterećenje u slučaju

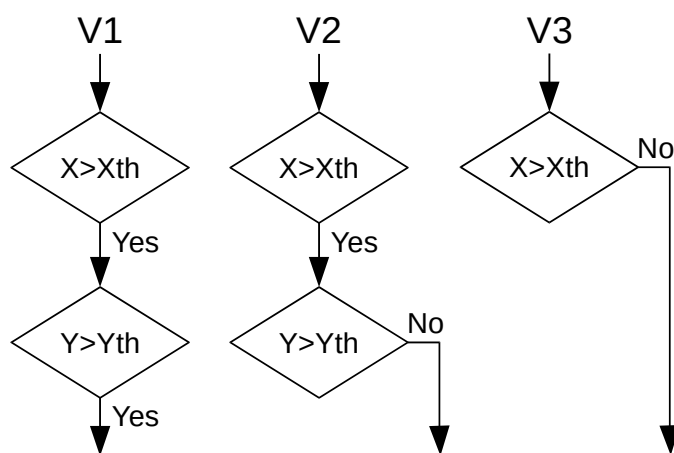
otkaza glavne komponente, ali ne proizvodi odziv, ili kao *hladna rezerva* (engl. *cold standby*), što znači da u normalnom režimu ne održava potrebno stanje, već to stanje uspostavlja tek nakon otkaza glavne komponente.

- Softverska redundansa:
 - Statička: programiranje u N verzija (engl. *N-version programming*)
 - Dinamička: detekcija greške, *oporavak od greške unazad* (engl. *backward error recovery*) i *oporavak od greške unapred* (engl. *forward error recovery*).

Programiranje u N verzija

- Po uzoru na tehnike statičke redundanse u hardveru (NMR) može se obezbediti i softverska statička redundansa, u smislu postojanja N (gde je $N > 1$) softverskih komponentata koje obavljaju istu funkciju, odnosno proizvode neki rezultat, a čiji se većinski rezultat smatra ispravnim – tzv. *programiranje u N verzija* (engl. *N-version programming*). Ova redundansa je statička, jer su ove komponente ugrađene u sistem u samoj njegovoj konstrukciji i sve one obavljaju svoj posao uvek, čak i kada otkazi ne postoje.
- Međutim, razlika u odnosu na hardver jeste u tome što se softver ne haba tokom upotrebe, odnosno ne kvari nakon nekog vremena iako je u početku bio ispravan, već je njegovo ponašanje nepromenjeno: greške u softveru nastaju u njegovoj proizvodnji, a ne eksploataciji, kada se samo ispoljavaju kao padovi, po pravilu nakon nekog vremena upotrebe (jer nisu otkriveni tokom testiranja). Na primer, problem tzv. „curenja memorije“ (engl. *memory leak*) ili „viseće reference“ (engl. *dangling reference*) je problem u implementaciji softvera, a može se ispoljiti tek nakon dužeg rada programa (u prvom slučaju) ili samo ponekad, kada izvršavanje programa dođe do tačke u kojoj se greška nalazi (u drugom slučaju).
- Zbog toga se osnovna ideja ovog programiranja zasniva na *nezavisnoj* realizaciji N funkcionalno ekvivalentnih verzija programske komponente (potprograma, procesa, modula) iz iste početne specifikacije, oslanjajući se na pretpostavku da će nezavisno razvijene komponente imati i nezavisno ugrađene greške.
- Te komponente se izvršavaju uporedo (npr. kao uporedni procesi, ili potprogrami koji se pozivaju), sa istim ulaznim podacima, dok njihove izlazne rezultate upoređuje tzv. *drajver* (engl. *driver*) komponenta (npr. proces ili pozivajući potprogram). Drajver ima zadatak da proziva svaku od verzija komponentata, prikupi sve njihove rezultate, uporedi te rezultate i donese odluku. U principu, njihovi rezultati bi trebalo da budu identični. Ukoliko postoji razlika, usvaja se većinski rezultat, pod uslovom da takav postoji.
- Ukoliko postoje razlike u rezultatu, drajver može da donese odluku da nastavi izvršavanje svih komponentata, pa i onih koje su dale manjinske rezultate, ili da neku od njih ugasi, odnosno prekine njenu dalju upotrebu.
- Iako na prvi pogled deluje kao jednostavan pristup, programiranje u N verzija poseduje neke probleme i nedostatke.
- Problem u odlučivanju može da bude poređenje rezultata dobijenih iz različitih verzija. Naime, ukoliko su rezultati egzaktno uporedivi, npr. celobrojni ili stringovi (nizovi znakova), rezultati mogu i treba da budu identični da bi se smatrali istim, tj. prihvatljivim. Međutim, ukoliko komponente proizvode samo racionalne aproksimacije realnih brojeva (npr. vrednosti nekih fizičkih veličina), te aproksimacije mogu biti različite, ali ipak prihvatljive, odnosno korektne, jer su nastale kao (korektne) aproksimacije primenom različitih algoritama izračunavanja ili zaokruživanja u različitim komponentama.

- Zbog toga je u nekim slučajevima potrebno *neegzaktno glasanje* (engl. *inexact voting*). Jedno jednostavno rešenje jeste da se u takvim slučajevima rezultati koji međusobno ne odstupaju više od neke zadate margine smatraju istovetnim.
- Problem kod neegzaktnog glasanja može da bude i tzv. *problem nekonzistentnog poređenja* (engl. *inconsistent comparison problem*), kada svaka komponenta treba da donosi odluke na osnovu približnih rezultata. Na primer, ako komponenta treba da donese binarnu odluku (da/ne) na osnovu veličine neke racionalne vrednosti, ukoliko ona jeste ili nije veća/manja od nekog praga te vrednosti, ili čak da donosi više takvih odluka na osnovu više takvih vrednosti: ako su te vrednosti, npr. izmerene kao fizičke veličine, blizu svog praga, neke ili sve verzije komponenata mogu doneti potpuno različite odluke od više mogućih, iako sve rade ispravno i sve te odluke se na neki način mogu smatrati ispravnim.



- Drugi važan uslov jeste *nezavisnost* izrade verzija komponenata. Nezavisna realizacija N verzija podrazumeva da N pojedinaca ili grupa proizvodi N verzija softvera *bez* međusobnog uticaja (interakcije), kako bi ispunile pretpostavku nezavisnosti eventualnih grešaka. Zbog toga se ovaj pristup naziva i *raznolikost dizajna* (engl. *design diversity*).
- Da bi ovakav pristup imao smisla, treba da bude ispunjena osnovna pretpostavka da se program može u potpunosti, konzistentno i nedvosmisleno specifikovati, kao i da programi koji su razvijeni nezavisno i otkazuju nezavisno. Drugim rečima, da ne postoje korelacije između grešaka u nezavisno razvijenim programima.
- Ovo obično zahteva da se nezavisne verzije realizuju od strane nezavisnih timova, na što različiti način, recimo u različitim programskim jezicima i/ili u različitim okruženjima (prevodioci, biblioteke, izvršna okruženja itd.), jer su programi na različitim programskim jezicima tipično osetljivi na različite vrste programerskih grešaka.
- Međutim, neki eksperimenti koji su imali zadatak da provere hipotezu o nezavisnosti otkaza nezavisno realizovanih programa dali su kontradiktorne rezultate. Osim toga, u slučaju kada je neki deo specifikacije složen, dvosmislen, ili nejasan, postoji velika verovatnoća da će on dovesti do nerazumevanja zahteva od strane većine nezavisnih timova. U tom slučaju će sve njihove realizacije biti pogrešne. Naravno, ako u samoj specifikaciji postoji greška, sve realizacije će najverovatnije biti pogrešne.
- Konačno, realizacija više nezavisnih verzija iste komponente značajno povećava troškove razvoja i testiranja softvera. Postavlja se pitanje da li je investicija u proizvodnju i testiranje više verzija iste komponente opravdana, ili je bolje ta sredstva uložiti na drugi način, recimo na pouzdanije tehnike dizajna i implementacije, ili temeljnije testiranje i proveru kvaliteta softvera.

- Zbog svega ovoga, iako programiranje u N verzija može značajno da doprinese povećanju pouzdanosti sistema, ono se upotrebljava samo u slučajevima visokih zahteva za pouzdanošću, odnosno veoma kritičnih sistema (engl. *mission-critical systems*), i to po pravilu u saradnji sa drugim tehnikama povećanja pouzdanosti.

Dinamička softverska redundansa

- Kod *dinamičke* redundanse redundantna komponenta se aktivira samo ukoliko dođe do otkaza u osnovnoj komponenti. Ovaj pristup ima sledeće faze:
 - *Detekcija greške* (engl. *error detection*): greška mora biti najpre otkrivena (detektovana) u samom softveru, kako bi dalje bila tretirana, odnosno da bi sistem uopšte mogao da se od nje oporavi.
 - *Izolacija i procena štete* (engl. *damage confinement and assessment*): od trenutka nastanka do trenutka detekcije greške može proći neko vreme, odnosno greška može nastati u jednom delu programa, a detektovana u nekom sasvim drugom, znatno kasnije. Kašnjenje od trenutka nastanka otkaza do trenutka detekcije greške znači da su se pogrešna informacija i drugi negativni efekti greške možda proširili i na druge delove sistema. Da bi sistem bio robusniji, potrebno je da se ovakva propagacija greške što više suzi, odnosno greška izoluje, a nakon njene detekcije, po mogućstvu proceni njen opseg uticaja i šteta koja je nastala.
 - *Oporavak od greške* (engl. *error recovery*): tehnike oporavka od greške teže da transformišu sistem koji je oštećen u stanje u kome će sistem nastaviti normalno operisanje (možda sa degradiranom funkcionalnošću ili performansama).
 - *Tretman otkaza* (engl. *fault treatment*): Iako je greška detektovana i šteta od nje možda nadoknađena, njen uzrok možda još uvek postoji i može uzrokovati novu grešku u daljem radu sistema, sve dok se sistem ne popravi.

Detekcija greške

- Na osnovu mesta na kom se greška otkriva, tehnike detekcije greške se mogu klasifikovati u sledeće kategorije:
 - detekcije grešaka u okruženju u kome se program izvršava;
 - detekcije grešaka unutar same aplikacije.
- Greške detektovane u okruženju:
 - Greške koje detektuje hardver (procesor), kao što su greške tipa "*illegal instruction executed*" ili "*arithmetic overflow*" ili "*memory access violation*". Ovakve greške detektuje procesor i signalizira grešku hardverskim izuzetkom (engl. *exception*) koji obrađuje odgovarajuća prekidna rutina koja pripada operativnom sistemu. Operativni sistem može proslediti tzv. *signal* procesu koji se izvršavao kada je ova greška detektovana, odnosno u čijem kontekstu je greška nastala. Reakciju na ovakve signale program može da definiše kao procedure koje se pozivaju od strane operativnog sistema na pojavu ovakvih signala, tzv. *obrađivače signala* (engl. *signal handler*). Obrađivači se definišu odgovarajućim sistemskim pozivima, a aktiviraju mehanizmom sličnim obradi prekida u hardveru.
 - Greške koje uočava izvršno okruženje (engl. *runtime environment*) ili virtuelna mašina (eng. *virtual machine*) datog programskog jezika, ili sam kod koji je generisao prevodilac i koji proverava odgovarajuće uslove i signalizira grešku ukoliko oni nisu ispunjeni. Primeri su greške tipa "*value out of range*" ili "*array bound error*" ili "*null pointer dereferencing*" ili "*type casting exception*".

- Neki primeri tehnika detekcije greške u aplikaciji:
 - Provere pomoću replika (engl. *replication checks*): poređenje vrednosti ili rezultata različitih replika, kao u programiranju u N verzija.
 - Provere pomoću kodova (engl. *coding checks*) se koriste za detekciju grešaka u podacima, npr. *parity*, *checksum*, CRC i slično.
 - Strukturne provere (engl. *structural checks*) se odnose na provere integriteta struktura podataka, npr. korektnost povezanosti listi. Oslanjaju se na tehnike kao što su brojanje referenci na objekte, redundantni pokazivači (dva pokazivača koji bi morali da budu isti, a ako nisu, detektuje se greška), konzistentnost pokazivača (npr. `this->next->prev==this`) itd.
 - Reverzne provere (engl. *reversal checks*) se mogu primeniti u situacijama u kojima postoji jedan-na-jedan preslikavanje ulazne na izlaznu vrednost; u tom slučaju se izlazni rezultat može proveriti inverznom funkcijom i poređenjem dobijenog argumenta sa ulaznim. Npr. funkcija koja izračunava kvadratni koren nekog broja može se proveriti prostim kvadriranjem izlaza (primetiti da je potrebno neegzaktno poređenje zbog zaokruživanja).
 - Provere razumnosti (engl. *reasonableness checks*) se oslanjaju na znanje o internoj konstrukciji sistema i logičkim uslovima koje stanje sistema treba da zadovolji u nekim tačkama izvršavanja. Ove uslove provere razumnosti (engl. *assertions*) u programe ugrađuju sami programeri. To su logički izrazi nad programskim varijablama koji treba da budu zadovoljeni u određenim trenucima izvršavanja programa, a ako nisu, detektuje se greška.
 - Vremenske provere (engl. *timing checks*):
 - *Watchdog timer* je softverski proces ili specijalizovan hardverski uređaj (tajmer) koji aplikacija, ukoliko ispravno radi, restartuje povremeno, stalnim i redovnim prolaskom kroz neke kontrolne tačke (npr. u glavnoj petlji svoje obrade), tako da njegovo vreme koje meri ne ističe u slučaju ispravnog rada softvera. Ukoliko dođe do greške u softveru, tako što aplikacija "odluta" (recimo skok na „instrukcije“ koje to nisu zbog korumpiranih pokazivača, npr. pokazivača na virtuelne funkcije), procesor interpretira kao instrukcije potpuno slučajan sadržaj u memoriji i zato ne restartuje ovaj tajmer (jer ne izvršava posebno zahtevane instrukcije njegovog restartovanja), tako da tajmeru ističe vreme i on generiše signal o nastanku greške, npr. kao hardverki prekid. Ovakva tehnika je praktično neizostavna u svim ugrađenim sistemima koji treba da rade neprekidno i bez ljudskog nadzora i intervencije. Moguće su i razne čisto softverske varijante ove tehnike, tako što se kao tajmer koristi deo programa ili poseban proces, dok ostali procesi povremeno ostavljaju „tragove“ kao mrvice u bajci o Ivici i Marici (engl. *breadcrumbs*) i povremeno se javljaju ovom tajmeru. Ukoliko se neki proces ne javi u određenom vremenu, tajmer može zaključiti da je taj proces otkazao („zaglavio“ se ili „odlutao“).
 - Kontrola isteka vremena (engl. *timeout*): neka aktivnost (npr. neka obrada, proces) ili stanje čekanja (na odgovor na poruku, na sinhronizacionu primitivu, na resurs koji se traži) može biti ograničen vremenski tzv. tajmaut kontrolom. Ukoliko ovo vreme istekne, a ograničena aktivnost ili stanje čekanja se do tada ne završi, istek vremena ukazuje na grešku.
 - Kontrola prekoračenja rokova (engl. *deadline miss*). Ukoliko se raspoređivanje radi u operativnom sistemu, što je tipično slučaj, ovo se može smatrati greškom detektovanom u okruženju, jer je detektuje operativni sistem uz pomoć hardvera.

- Dinamičke ili vremenske provere razumnosti (engl. *dynamic reasonableness checks*) se oslanjaju na činjenicu da se neka vrednost ne može promeniti preko neke mere u određenom vremenskom intervalu, zbog svoje fizičke prirode i ograničenja fizičkim zakonima. Na primer, dva očitana odbirka neke veličine dobijena sa nekog analognog ulaza u dva bliska vremenska trenutka ne mogu se mnogo razlikovati zbog prirode ulaznog signala, odnosno njegove konačne i poznate brzine promene u vremenu.

Treba primetiti da se mnoge od ovih provera mogu vršiti u hardveru, ili zahtevaju podršku hardvera ili operativnog sistema, pa se tada mogu smatrati greškama iz okruženja.

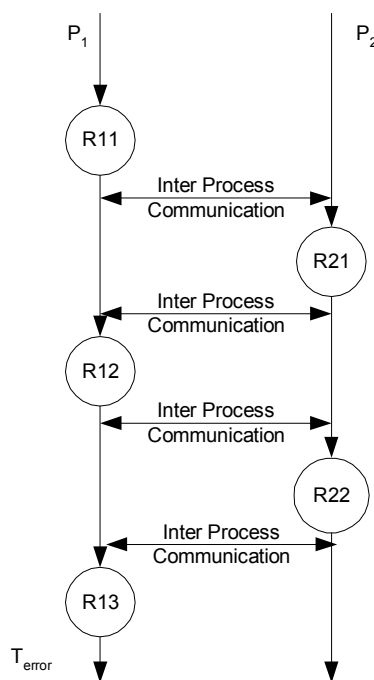
Izolacija i procena štete

- Od trenutka kada greška nastane, pa do onog kada se ona detektuje, može proći određeno vreme, a program napredovati u svom izvršavanju, pa se efekat greške može proširiti na veći deo programa. Na primer:
 1. U jednom delu programa, u jednom potprogramu, prekorači se opseg nekog niza prilikom upisa vrednosti u elemente tog niza (npr. prekorači se bafer pri učitavanju podataka). Ukoliko jezik ne obezbeđuje detekciju takve greške (C/C++), ovaj deo programa možda neće otkazati, tj. završiće svoju obradu ispravno (upisaće i možda i pročitati sve te iste upisane vrednosti).
 2. Međutim, ovaj upis „pregaziće“ vrednosti varijabli koje su slučajno smeštene u memoriji neposredno iza datog niza. Na primer, upisaće nekorektnu vrednost u pokazivač koji treba da ukazuje na neki objekat.
 3. Neki sasvim drugi deo programa, neki potprogram, čita tu vrednost pokazivača, pristupa podacima tog „objekta“, poziva potprograme koje koriste te vrednosti. Možda ni ti potprogrami neće detektovati grešku, već će „tiho“ obrađivati te nekorektne vrednosti.
 4. Tek možda mnogo kasnije, u nekoj narednoj proceduri, neregularna vrednost može biti detektovana kao pogrešna, npr. nakon što se ta vrednost protumči kao pokazivač i procesor generiše izuzetak zbog neregularnog pristupa memoriji.
- Zbog toga su veoma važne, i sa tehnikama detekcije greške u tesnoj vezi, tehnike *izolacije* greške (engl. *isolation*): strukturiranje sistema tako da se ograniči širenje posledica nastanka greške, odnosno minimizuje šteta.
- Jedna tehnika izolacije je *modularizacija* uz *enkapsulaciju* (engl. *encapsulation*), koja podrazumeva da se sistem deli na module koji interaguju samo kroz jasno definisane i kontrolisane interfejse, dok su njihove implementacije sakrivene i nedostupne spolja. Ovo doprinosi tome da se greška nastala u jednom modulu teže širi i na druge. Ovo je *statički* pristup izolaciji.
- *Dinamički* pristup izolaciji podržavaju *atomične akcije* (engl. *atomic actions*), ili *transakcije* (engl. *transaction*). Atomična akcija (transakcija) je skup elementarnih operacija sa sledećim svojstvima:
 1. transakcija je atomična ili *nedeljiva* (engl. *indivisible*), u smislu da su efekti njenog izvršavanja uvek ili takvi da je ona u celini izvršena (uspela), ili kao da nije ni započinjala izvršavanje (ukoliko se tokom njenog izvršavanja dogodila greška); drugim rečima, transakcija prevodi sistem iz jednog konzistentnog stanja u drugo takvo stanje, a ne može uraditi samo delimičnu promenu stanja u slučaju da se tokom njenog izvršavanja dogodi otkaz;
 2. *izolovana* (engl. *isolated*) u smislu da je njen efekat takav kao da za vreme njenog izvršavanja nema interakcija sa drugim transakcijama koje se izvršavaju uporedno

(konkurentno); drugim rečima, efekti transakcija su takvi kao da se one izvršavaju sekvencijalno, jedna po jedna, čak i kada se u stvarnosti izvršavaju uporedno.

Oporavak od greške

- Nakon što je greška detektovana, sistem koji je tolerantan na otkaze preduzima proces *oporavka od greške* (engl. *error recovery*) koji treba da prevaziđe tu grešku, odnosno da sistem sa greškom dovede u stanje normalnog operisanja, mada možda uz degradiranu funkcionalnost ili performanse.
- Postoje dva pristupa oporavku od greške: *oporavak unapred* (engl. *forward error recovery*, FER) i *oporavak unazad* (engl. *backward error recovery*, BER).
- FER tehnike pokušavaju da prevaziđu postojanje greške i nastave izvršavanje od stanja sa greškom, selektivno korigujući stanje sistema i dovodeći ga do delimično ili potpuno ispravnog stanja, ali svakako do bezbednog stanja. To uključuje aktivnosti koje treba da svaki aspekt kontrolisanog okruženja koji je oštećen ili opasan učine sigurnim. Ove aktivnosti su sasvim specifične za dati sistem i zavise od preciznosti procene lokacije i uzroka greške, kao i njenih posledica. Na primer, korišćenje redundantnih pokazivača koji se koriste kao rezerva za oštećene, ili samokorigujućih kodova.
- BER tehnike se zasnivaju na sledećem:
 - na određenim specificiranim mestima, u kojima je stanje sistema ispravno, bezbedno i konzistentno, tj. u tzv. *tačkama oporavka* (engl. *recovery point*), pamti se stanje sistema tako da se može kasnije restaurirati;
 - oporavak od detektovane greške se vrši povratkom sistema u posledenje zapamćeno bezbedno stanje („razmotavanjem“, engl. *rollback*), odnosno u poslednju tačku oporavka, odakle se pokušava ista ili neka druga, alternativna aktivnost.
- Prednost ovakve tehnike je to što ona ne zavisi od uzroka i lokacije greške, odnosno od konkretne specifičnosti nastale greške i aplikacije, već se efekat greške jednostavno poništava restauracijom poznatog ispravnog stanja sistema, pa je ovaj mehanizam krajnje generički. Zato se ova tehnika može koristiti i za oporavak od grešaka koje se ne mogu predvideti, kao što su greške u implementaciji.
- Međutim, BER tehnike imaju i značajne nedostatke:
 - Ne može se poništiti efekat promene okruženja (npr. jednom poslata poruka ili ispaljen projektil ne može se zaustaviti i vratiti).
 - Pamćenje stanja sistema zavisi od semantike programskog jezika i može biti logički veoma složeno.
 - Implementacija pamćenja stanja može da bude veoma složena i zahtevna po pitanju vremena u toku izvršavanja i zauzeća memorijskog prostora.
 - Kod konkurentnih procesa koji međusobno komuniciraju moguć je tzv. *domino efekat* (engl. *domino effect*): u primeru na slici, ukoliko greška u naznačenom trenutku nastane u procesu P1, samo on se vraća do tačke oporavka R13, jer od te tačke do nastanka greške nije bilo uticaja na drugi proces; međutim, ukoliko greška nastane u procesu P2, zbog potrebe poništavanja efekta međusobne razmene informacija i uticaja na drugi proces, nastaje domino efekat jer se oba procesa moraju vraćati unazad do jako daleke tačke; u najgorem slučaju, svi procesi koji interaguju se moraju potpuno poništiti, što može biti skupo.



- Jedna klasična BER tehnika koja se često primenjuje porazumeva korišćenje već opisanih transakcija.
- Zbog svega što je rečeno, obe tehnike FER i BER imaju svoje prednosti i nedostatke. FER tehnike su krajnje zavisne od aplikacije i neće biti dalje razmatrane.

Blokovi oporavka

- *Blok oporavka* (engl. *recovery block*) predstavlja jezički konstrukt za podršku BER.
- Blok oporavka je *blok naredbi* kao u standardnom programskom jeziku, osim što se ulazak u ovaj blok implicitno smatra *tačkom oporavka* (engl. *recovery point*), a njegov izlaz je *test prihvatljivosti* (engl. *acceptance test*), i što ima jednu *primarnu* i jednu ili više *alternativnih akcija*, odnosno grana.
- Moguća sintaksa ovog konstrukta:

```

ensure <acceptance test>
by
  <primary action>
else by
  <alternative action 1>
else by
  <alternative action 2>
...
else by
  <alternative action n>
else error;

```

- Semantika izvršavanja bloka oporavka je sledeća. Pri nailasku izvršavanja na ovaj blok, najpre se čuva stanje u ulaznoj tački oporavka. Zatim se izvršava njegova *primarna akcija*. Nakon njenog završetka, izračunava se *test prihvatljivosti*. Ukoliko je on zadovoljen, ceo blok se završava. U suprotnom, restaurira se stanje tačke oporavka ovog bloka i dalje izvršava prva alternativna akcija bloka. Na njenom kraju se ponovo proverava test prihvatljivosti itd. sve dok se blok ne završi sa zadovoljenim testom prihvatljivosti, ili dok se ne iscrpe sve alternativne grane. Ako nijedna alternativna akcija ne dovede do

zadovoljenja testa, oporavak se preduzima na višem nivou (u eventualnom okružujućem bloku oporavka).

- Test prihvatljivosti je logički uslov koji ispituje da li je sistem u *prihvatljivom* stanju posle izvršavanja bloka. Treba naglasiti da se ovde radi o testu *prihvatljivosti*, a ne *korektnosti*, što znači da rezultat ne mora da bude potpuno tačan, ali mora da bude prihvatljiv tako da dati deo sistema nastavi bezbedno funkcionisanje eventualno sa degradiranim funkcionalnostima ili tačnošću.
- Za formiranje testova prihvatljivosti mogu se koristiti sve navedene tehnike detekcije greške. Treba obratiti pažnju na to da pogrešno napravljen test prihvatljivosti može da dovede do situacije u kojoj se greške provlače nezapaženo.
- Test prihvatljivosti svakako unosi dodatne režijske troškove u izvršavanje (tj. predstavlja redundansu), pa je potrebno pronaći balans između ugrađivanja razumnih testova prihvatljivosti u program, tako da se postigne potreban nivo pouzdanosti, i troškova njihovog izvršavanja.
- Na primer [1], rešavanje diferencijalnih jednačina može se obaviti tzv. eksplicitnim Kutta metodom, koji je brz ali neprecizan za određene jednačine, ali i implicitnim Kutta metodom, koji je zahtevniji ali uspešno rešava složene jednačine. Sledeći primer obezbeđuje da se efikasniji metod upotrebljava sve dok daje zadovoljavajuće rezultate, a da se u složenim situacijama koristi složeniji i pouzdaniji metod; osim toga, ovaj pristup može i da toleriše potencijalne greške u implementaciji eksplicitne metode, ukoliko je test prihvatljivosti dovoljno fleksibilan:

```
ensure RoundingErrorWithinAcceptableTolerance
by
    ExplicitKuttaMethod
else by
    ImplicitKuttaMethod
else error;
```

- Nijedan popularan programski jezik ne podržava direktno ovaj konstrukt, ali se ovaj koncept može realizovati postojećim konstruktima i mehanizmima (npr. izuzecima i transakcijama). Ključan problem u realizaciji jeste pamćenje i restauriranje stanja za tačke oporavka.

Poređenje između programiranja u N verzija i blokova oporavka

- Programiranje u N verzija (NVP) se oslanja na statičku redundansu u kojoj N verzija nekog modula radi uvek, čak i kada ne postoji otkaz. Blokovi oporavka (RB) se oslanjaju na dinamičku redundansu u kojoj se redundantni delovi aktiviraju samo u slučaju postojanja otkaza, jer se alternativne akcije izvršavaju samo ukoliko test prihvatljivosti ne prođe.
- Tokom izvršavanja, NVP generalno zahteva N puta veću količinu resursa nego jedna varijanta, za svaku verziju modula. Iako RB zahtevaju samo jedan skup resursa, potrebni su dodatni resursi i vreme izvršavanja za uspostavljanje i restauraciju tačaka oporavka.
- NVP koristi glasanje za detekciju greške, dok RB koristi test prihvatljivosti. Glasanje je generalno jeftinije i jednostavnije nego izvršavanje testa prihvatljivosti, pod uslovom da je glasanje (egzaktno ili neegzaktno) moguće izvesti. Međutim, u slučaju kada glasanje nije moguće izvesti, testovi su bolje rešenje.
- Obe tehnike uključuju posebne troškove razvoja, jer obe zahtevaju realizaciju N različitih varijanti implementacije funkcionalno iste stvari. NVP još zahteva i realizaciju drajvera, a RB zahteva i realizaciju testa prihvatljivosti.

-
- Obe tehnike se oslanjaju na raznovrsnost izrade da bi tolerisale nepredvidive greške. Međutim, nijedna od njih ne može da se bori protiv grešaka uzrokovanih neispravnom specifikacijom zahteva.
 - Ni NVP ni RB ne mogu poništiti efekat na okruženje, ali se mogu strukturirati tako da se promene okruženja ne rade u okviru njihovih modula, odnosno akcija. Verzije modula u NVP se u principu prave tako da ne vrše promenu okruženja, već se ta promena radi tek nakon donošenja odluke.

Izuzeci i njihova obrada

- Koncept *izuzetka* (engl. *exception*) i mehanizam njegove obrade predstavljaju važnu podršku praktično svih danas popularnih programskih jezika. Ova podrška je veoma slična u svim tim jezicima, uz manje razlike.
- *Izuzetak* je nastanak nenormalnih, neregularnih ili neočekivanih uslova koji zahtevaju poseban tretman, različit od onog predviđenog normalnim, regularnim tokom programa. Drugim rečima, izuzetak se može definisati i kao ukazatelj na detektovanu grešku.
- Ukazivanje na detektovanu grešku naziva se *podizanje* (engl. *raising*) ili *signaliziranje* (engl. *signalling*) ili *bacanje* (engl. *throwing*) izuzetka.
- Odgovor na podignut izuzetak naziva se *obrada* (engl. *handling*) ili *hvatanje* (engl. *catch*) izuzetka.
- Obrada izuzetka je FER mehanizam oporavka od greške, zasnovan na dinamičkoj softverskoj redundansi, jer nakon detekcije greške ne postoji implicitni povratak sistema u sačuvano bezbedno i konzistentno stanje. Umesto toga, nakon detekcije greške, kontrola toka se prebacuje na deo programa koji je namenjen za obradu te greške na unapred definisan i kontrolisan način. Međutim, mehanizam obrade izuzetaka može se iskoristiti za implementaciju BER mehanizama.

Obrada izuzetaka bez posebne jezičke podrške i sa njom

- Ukoliko sam jezik ne podržava obradu izuzetaka, onda se neregularne situacije mogu rešavati pomoću posebnih povratnih vrednosti iz potprograma, kao što je to uobičajeno u starijim, proceduralnim jezicima:

```
if (functionCall(someParams)==OK) {  
    // Normal operation  
} else {  
    // Error handling code  
}
```

- Na primer:

```
enum DeviceStatus {ok, deviceFaulty, communicationFailed};  
DeviceStatus readTemperature (Temperature* value); DeviceStatus readPressure  
(Pressure* value); DeviceStatus readHumidity (Humidity* value);  
...  
  
void readMeteo () {  
    Temperature temp; Pressure press; Humidity hum;  
    DeviceStatus status = readTemperature(&temp);  
    if (status==ok) {  
        status = readPressure(&press);  
        if (status==ok) {  
            status = readHumidity(&hum);  
            if (status==ok) {  
                // Finally, do the job with temp, press, and hum  
            } else {  
                // Handle readHumidity exception  
            }  
        } else {  
            // Handle readPressure exception  
        }  
    }  
}
```

```

    }
} else {
    // Handle readTemperature exception
}
}

```

- Iako je ovaj mehanizam jednostavan, on ima sledeće probleme:
 - kod postaje nepregledan čim postoji nekoliko koraka koji mogu da rezultuju izuzetkom (duboko i nepregledno ugneždivanje naredbi *if-then-else*);
 - funkcije koje vraćaju status ne mogu da vraćaju svoje “prirodne” rezultate, koji se zato moraju prenositi kao izlazni argumenti po referenci/preko pokazivača;
 - pozivalac potprograma koji vraćaju greške može slučajno ili namerno da ignoriše te greške: da ih ne proverava i ne obrađuje, ili da ih ne propagira svom pozivaocu;
 - ispitivanje povratnih statusa, tj. postojanja grešaka, troši vreme i resurse tokom izvršavanja najčešće bespotrebno, jer greška u većini slučajeva ne postoji;
 - pitanje je kako na ovaj način obrađivati greške detektovane u okruženju.
- Uzrok ovih problema je u tome što je obrada izuzetaka učešljana u regularan tok programa, isprepletana je sa osnovnim tokom, pa ga čini nepreglednim i težim za programiranje.
- Bolji pristup podrazumeva sledeće:
 - regularni tok programira se kao da je sve u redu, tj. da nema izuzetaka i nije opterećen njihovom obradom, s tim da određeni koraci regularnog toka mogu da bace (engl. *throw*) izuzetak;
 - obrada izuzetaka se piše kao obrađivač ili *hvatač* izuzetka (engl. *exception handler*) pridružen bloku sa regularnim tokom, koji hvata izuzetak (engl. *catch*), ali nije izmešan sa njim.
- Mehanizam obrade izuzetaka nije direktno vezan za objektno orijentisanu paradigmu, ali je podržan u svim današnjim popularnim OO programskim jezicima na vrlo sličan način, po istom principu.

```

enum DeviceException {deviceFaulty, communicationFailed};

Temperature readTemperature ();
Pressure readPressure ();
Humidity readHumidity ();
...
void readMeteo () {
    try {
        Temperature temp = readTemperature(&temp);
        Pressure press = readPressure(&press);
        Humidity hum = readHumidity(&hum);
        // Finally, do the job with temp, press, and hum
    }
    catch (DeviceException& de) {
        // Handle exceptions
    }
}

```

- *try* blok uokviruje regularan tok u kom se može baciti izuzetak. Ukoliko se *try* blok završi bez izuzetka, završava se ceo konstrukt *try-catch* (*catch* blokovi se preskaču). Ukoliko se tokom izvršavanja *try* bloka baci izuzetak, prekida se izvršavanje tog *try* bloka i kontrola prebacuje na *catch* blok koji prihvata izuzetak tog tipa kao svoj argument.

Izvori izuzetaka

- Kao što je ranije rečeno, postoje dva izvora izuzetaka: okruženje programa (hardver, operativni sistem ili izvršno okruženje jezika) i sam program (aplikacija).
- Izuzetak se može podići *sinhrono*, kao neposredan rezultat izvršavanja dela koda koji je pokušao neku neregularnu operaciju, ili *asinhrono*, što znači nezavisno od operacije koja se trenutno izvršava, u nepredvidivim trenucima.
- Sinhroni izuzeci:

- Detektovani od strane okruženja: na primer, izuzeci tipa deljenja nulom ili prekoračenje granica niza.

Na jeziku C++, generisanje nekih ovakvih izuzetaka nije uvek garantovano, već zavisi od implementacije i platforme. Kod koji je generisao prevodilac ne generiše ovakve izuzetke (nema nikakve provere u vreme izvršavanja), pa je sama aplikacija odgovorna za detekciju ovakvih grešaka.

Jezik Java dosledno generiše sve tipove ovakvih izuzetaka.

Na primer, sledeći pogrešan kod na jeziku C++ može (ali ne mora) da podigne hardverski izuzetak koji nije uhvaćen od strane programa, dok se u jeziku Java podiže određeni izuzetak:

```
int a[N];
...
for (int i = 0; i<=N; i++)
    ...a[i]...
```

Slično, eksplicitna statička konverzija pokazivača na osnovnu klasu u pokazivač na izvedenu klasu u jeziku C++ ne generiše izuzetak ukoliko se iza pokazivača ne krije objekat željenog tipa, već će program imati nepredvidivo ponašanje i stoga verovatno nezgodnu grešku detektovanu kasnije. Java generiše izuzetak u ovom slučaju u vreme izvršavanja. Na primer, sledeći kod na jeziku C++ neće podići izuzetak, već će uzrokovati nezgodnu grešku, dok će ekvivalentan kod na jeziku Java podići izuzetak:

```
Base* pb = new Base;
...
Derived* pd = (Derived*)pb;
```

Sa druge strane, mnoge funkcije iz standardne biblioteke jezika C++ mogu da bace izuzetke, npr. `vector::at`, `string::substr` itd, čime signaliziraju greške (nemogućnost da urade zahtevano npr. zbog nekorektnosti argumenta). Osim toga, neki operatori ugrađeni u jezik C++ mogu da bace izuzetke, npr. `dynamic_cast` (ako rezultat izraza koji se konvertuje nije zahtevanog odredišnog tipa za reference) ili `new` (ako ne može da alocira prostor u memoriji za objekat koji se kreira), opet signalizirajući grešku.

- Izuzeci detektovani od strane aplikacije su oni eksplicitno podignuti u programskom kodu aplikacije (operatorom `throw`). Ovi izuzeci signaliziraju nezadovoljenje uslova definisanih u samom programu. C++ i Java nemaju ugrađene mehanizme za implicitnu proveru uslova, već se oni jednostavno eksplicitno programiraju. Na primer:

```
if (! some_assertion) throw anException;
```

Na primer:

```
template<typename T>T& vector<T>::at (int pos) {
    if (pos<0 || pos>=this->size) throw out_of_range;
```

```
return this->array[pos];
}
```

- Unutar obrađivača se može ponovo baciti isti izuzetak koji se obrađuje prostim izrazom `throw` bez parametra.
- Asinhroni izuzeci se uvek podižu implicitno i u nepredvidivim trenucima vremena, nezavisno od operacije koju aplikacija trenutno izvršava. Ovakve izuzetke po pravilu detektuje okruženje aplikacije, iako njihov uzrok može biti i sama aplikacija. Na primer:
 - Signal o padu napona napajanja uređaja, ispražnjennoj bateriji, neispravnosti nekog hardverskog uređaja ili alarmni signal iz okruženja.
 - Prekoračenje vremenskog roka: uzrok je ponašanje same aplikacije, ali je vremensko prekoračenje detektovao operativni sistem uz pomoć hardverske podrške za merenje vremena (prekid nakon isteka vremena).
 - Prekoračenje maksimalno dozvoljenog vremena izvršavanja dela programa ili vremenske kontrole (engl. *timeout*).
- Asinhroni izuzeci se obično nazivaju *asinhronim signalima* (engl. *asynchronous signal*). Ovakav signal programu može poslati operativni sistem ili neki drugi proces konkurentnog programa.
- I sinhroni izuzeci detektovani od strane operativnog sistema po pravilu se prosleđuju programu istim mehanizmom signala. Na ovaj način program koji prima i obrađuje signal može na jednobrazan način da obrađuje izuzetke ili druge poruke iz okruženja ili drugih procesa.

Reprezentacija izuzetaka

- Reprezentacija signala (asinhronih i sinhronih) je po pravilu definisana od strane operativnog sistema i standarda jezika. U principu, to su uvek jednostavni celi brojevi koji ne nose nikakvu dodatnu informaciju sem identifikacije vrste izuzetka.
- Za sinhronu izuzetke koje podiže sama aplikacija postoje dva najčešća pristupa reprezentaciji izuzetka:
 - izuzetak je eksplicitno deklarisan simbolička konstanta (jezik Ada);
 - izuzetak je objekat nekog tipa (C++ i Java).
- U jeziku C++, izuzetak može biti instanca bilo kog tipa, i ugrađenog i korisničkog (objekat klase). U trenutku podizanja izuzetka, kreira se jedan privremeni, bezimni objekat (smešta se u statički alociranu memoriju) i inicijalizuje izrazom iza naredbe `throw`. U trenutku pokretanja bloka koji obrađuje izuzetak (engl. *handler*) iza naredbe `catch`, formira se argument tog bloka kao automatski lokalni objekat tog bloka koji se inicijalizuje navedenim privremenim objektom. Semantika inicijalizacije u oba slučaja je ista kao i kod prenosa argumenata u funkciju. Na primer, izuzetak se može podići ovako:

```
if (...) throw HighTemperatureException(...);
```

ili ovako:

```
HighTemperatureException* preparedException = new HighTemperatureException;
...
if (...) throw preparedException;
```

ili ovako:

```
if (...) throw new HighTemperatureException(...);
```

a onda uhvatiti ovako:


```
catch (HighTemperatureException e) {
    ...
}
```

ili ovako:

```
catch (HighTemperatureException& e) {
    ...
}
```

odnosno ovako:

```
catch (HighTemperatureException* e) {
    ...
    delete e;
}
```

- Funkcije iz standardne biblioteke jezika C++ bacaju izuzetke koji su bezimeni objekti klase izvedenih iz osnovne klase `exception`; sve ove izvedene klase imaju sledeća osnovna svojstva:
 - objekti ovih klasa mogu se kreirati pozivom konstruktora sa argumentom koji predstavlja niz znakova koji daje objašnjenje za izuzetak:


```
throw out_of_range("Argument pos in function vector::at out of range.");
```
 - objekti ovih klasa mogu se kopirati (prilikom inicijalizacije ili operatorom dodele);
 - polimorfnu operaciju `what` koja vraća niz znakova koja predstavlja objašnjenje (zadat inicijalizacijom).
- Po pravilu, i korisnički definisani izuzeci treba da slede isti obrazac, odnosno da budu objekti klase (izvedenih iz klasa) iz ove hijerarhije.
- U jeziku Java, izuzetak je instanca klase direktno ili indirektno izvedene iz klase `Throwable`. Njegov životni vek je isti kao i životni vek bilo kog drugog objekta klase (nastaje eksplicitno i uništava se implicitno). Od izvora izuzetka do onoga ko ga hvata prosleđuje se, zapravo, samo referenca na taj objekat. Na primer, izuzetak se može podići ovako:

```
HighTemperatureException preparedException = new HighTemperatureException;
...
if (...) throw preparedException;
```

ili ovako:

```
if (...) throw new HighTemperatureException(...);
```

a uhvatiti ovako:

```
catch (HighTemperatureException e) {
    ...
}
```

Obrada izuzetka

- Mnogi operativni sistemi omogućavaju da program definiše hvatač signala kao potprogram koji se poziva kada se pojavi signal. Na primer, u standardnoj biblioteci jezika C++ postoji funkcija koja postavlja zadatu funkciju kao hvatač (obrađivač) definisanog signala:


```
std::signal(int signal, void (*handler)(int signal));
```

- Ovako definisan hvatač signala, zbog toga što signal može biti asinhron, ima puno ograničenja u pogledu toga šta sme da koristi od delova jezika i poziva od bibliotečnih funkcija.
- U nekim programskim jezicima kod za obradu izuzetka (engl. *exception handler*) može se pridružiti pojedinačnoj naredbi. Međutim, u većini blok-strukturiranih jezika, kod za obradu izuzetaka je vezan za blok.
- U jezicima C++ i Java, blok koji može da izazove izuzetak koji treba uhvatiti deklarise se kao `try` blok.

```
try {
    // Some code that can throw an exception
}
catch (ExceptionType e) {
    // Exception handler for exceptions of type ExceptionType
}
```

- Kada se izuzetak baci, prekida se izvršavanje prvog dinamički okružujućeg *try* bloka koji je započet, a nije završen, i u njemu traži *catch* blok koji može prihvatiti tip bačenog izuzetka; pravila uparivanja (skoro) su ista kao za argumente funkcija (uz sprovođenje implicitnih konverzija).
- *catch* blokovi pridruženi istom *try* bloku pretražuju se redom kako su navedeni (može ih biti više); bira se prvi koji po tipu argumenta odgovara bačenom izuzetku (može obraditi taj izuzetak) i započinje se njegovo izvršavanje.
- Ukoliko u datom *try* bloku nijedan *catch* blok ne može prihvatiti bačeni izuzetak, traži se sledeći dinamički okružujući *try* blok (mogu se ugnežđivati), koji može biti i van funkcije koja se izvršava - funkcija tako baca izuzetak (prosleđuje ga svom pozivaocu), i tako redom po steku ugnežđenih poziva funkcija u tekućoj niti.
- *catch* blok može baciti isti ili neki novi izuzetak, npr. tako što funkcija koja se unutar njega poziva baca izuzetak.
- Ako se *try* blok završi bez bačenog izuzetka, ili se *catch* blok koji je aktiviran završi bez bačenog izuzetka, izvršavanje nastavlja iza *try-catch* konstrukta:

```
void readMeteo () {
    try {
        ...
    }
    catch (ThermometerException& e) {
        ...
    }
    catch (ManometerException& e) {
        ...
    }
}

void calcMeteo () {
    try {
        ...
        ...readMeteo()...
        ...
    }
    catch (DeviceException& e) {
        ...
    }
}
```

- Zbog ovog redosleda, *catch* blok koji prima objekat izvedene klase (po vrednosti ili preko reference ili pokazivača na izvedenu klasu) mora da bude ispred onog koji hvata objekat osnovne klase (po vrednosti ili preko reference ili pokazivača na osnovnu klasu), jer u

suprotnom nikada neće biti aktiviran; na primer, ovakav deo programa nije dobar, jer drugi *catch* blok nikada neće biti izvršavan:

```
try {  
    ...  
}  
catch (Derived& e) {  
    ...  
}  
catch (Base& e) {  
    ...  
}
```

- *catch*(...) hvata izuzetke bilo kog tipa i, shodno tome, mora uvek biti poslednji u nizu (inače ostali nikada ne bi bili aktivirani); ovakav hvatač može poslužiti kao obezbeđenje za to da nijedan izuzetak ne bude bačen iz date funkcije:

```
try {  
    ...  
}  
catch (ThermometerException& e) {  
    ...  
}  
catch (ManometerException& e) {  
    ...  
}  
catch (...) {  
    ...  
}
```

Propagacija izuzetka

- Pitanje je kako tretirati situaciju kada bloku u kome može nastati izuzetak nije pridružen blok za njegovu obradu. U raznim jezicima postoji nekoliko pristupa:
 - Tretirati to kao grešku u programu koja se može otkriti u vreme prevođenja. Ovo je dosta restriktivan pristup, jer vrlo često nije moguće obraditi izuzetak tamo gde nastaje; na primer, izuzetak koji nastaje zbog neregularnih argumenata procedure. Današnji jezici nemaju ovakvo ograničenje.
 - Propagirati izuzetak u vreme izvršavanja u okružujuću proceduru (pozivaoca), tj. redom naviše, sve dok se ne nađe pozivalac koji ima kod za obradu izuzetka. Ovaj pristup imaju svi današnji popularni jezici.
- Sledeće pitanje kod obrade izuzetaka jeste to da li podizač izuzetka treba da nastavi svoje izvršavanje posle obrade izuzetka ili ne. Postoje dva generalna pristupa: *model povratka* (engl. *resumption model*) i *model terminacije* (engl. *termination model*).
- Opisani model koji ima većina danas popularnih OO programskih jezika jeste model terminacije: podizač izuzetka prekida svoje izvršavanje i kontrola toka prebacuje se na hvatač izuzetka, a potom nastavlja izvršavanjem okruženja tog hvatača; izvršavanje se nikad ne vraća na prekinut kod u kom je bačen izuzetak.
- Jezici C++ i Java podržavaju model terminacije na potpuno isti način. Jedina specifičnost jezika C++ je to da se pri prenosu kontrole na kod za obradu "razmotava stek", tj. propisno uništavaju (pozivom destruktora) svi automatski objekti klasa kreirani od trenutka ulaska u blok čiji je kod za obradu pronađen, do trenutka nastanka izuzetka. U jeziku Java za ovo nema potrebe, jer ne postoje automatski objekti klasa.
- Kod modela povratka, podizač izuzetka nastavlja svoje izvršavanje posle obrade izuzetka, od mesta na kom je nastao izuzetak. U takvom slučaju, kod za obradu izuzetka može se

shvatiti kao procedura koja se poziva sinhrono ili asinhrono, na pojavu izuzetka, i iz koje se izvršavanje vraća na kod u kom je nastao izuzetak. Obrada signala može biti po ovakvom modelu, ako se hvatač izuzetka regularno završi i vrati kontrolu na mesto poziva.

- Moguć je i hibridni model, kod koga kod za obradu izuzetka može da odluči da li je greška popravljiva ili ne i da li će se izvršavanje nastaviti na mestu nastanka izuzetka ili ne. Kombinovano korišćenje signala i izuzetaka je zapravo korišćenje oba modela.
- U ranijim verzijama jezika C++ deklaracija funkcije mogla je da uključuje i spisak tipova izuzetaka koje ta funkcija može da podigne, odnosno propagira, bilo direktno (iz svog koda), bilo indirektno (iz ugnežđenih poziva drugih funkcija, tranzitivno). Ukoliko ovaj spisak postoji, funkcija ne može podići izuzetak nijednog drugog tipa osim onih sa spiska, ni direktno ni indirektno. Ukoliko taj spisak ne postoji, funkcija može podići izuzetak bilo kog tipa (zbog kompatibilnosti sa jezikom C i starijim verzijama jezika C++). Na primer:

```
void checkTemperature () throw (HighTemperatureException*);
```

- Ovaj pristup je prevaziđen u novijim verzijama standarda jezika. Funkcija može biti označena kao `noexcept`, što znači da neće podizati izuzetke: ili ih sve obrađuje, ili će, ako neki ne obradi, biti pozvana predefinisana funkcija `terminate`; drugim rečima, funkcija označena kao `noexcept` ima jedan implicitni *try-catch* blok koji uokviruje celo njeno telo i hvata sve izuzetke (`catch(...)`).
- U jeziku Java postoji sličan princip, osim što ukoliko funkcija ne sadrži spisak izuzetaka, onda ona ne može podići nijedan. Ovaj pristup je značajno restriktivniji, ali time i pouzdaniji. Na primer:

```
public void checkTemperature () throws HighTemperatureException {...}
```

- U jeziku Java, hijerarhija izuzetaka počinje klasom `Throwable`. Izuzeci koji se ne proveravaju (engl. *unchecked*) ne moraju da se navode u listi izuzetaka koje funkcija podiže, oni se uvek podrazumevaju. Ove izuzetke podiže okruženje (npr. linker ili virtuelna mašina, ili su to greške u izvršavanju, npr. neregularna konverzija reference ili prekoračenje granica niza). Ostali tipovi izuzetaka se moraju deklarirati u deklaraciji funkcije ukoliko ih ona direktno ili indirektno podiže (engl. *checked*). Korisnički definisani tipovi izuzetaka realizuju se kao klase izvedene iz klase `Exception`.

Tretman izuzetaka i programiranje po ugovoru

- Šta je uopšte izuzetak? Šta je izuzetna situacija, a šta nije? Kada koristiti izuzetke, a kada prosto ispitivati stanje i rezultate?
- Primer: procedura koja učitava znakove iz nekog ulaznog znakovnog toka (npr. fajla) i obrađuje ih. Da li je nailazak na kraj fajla izuzetak?
- Ako se ne tretira kao izuzetak:

```
ifstream f("input.txt");
while (!f.eof()) {
    char c = f.getc();
    // ...
}
```

- Ako se tretira kao izuzetak:

```
ifstream f("input.txt");
while (true) {
    try {
        char c = f.getc();
```

```
    // ...  
} catch (...) {  
    break;  
}  
}
```

- Naravno da je ovo drugo potpuno besmisleno, jer kraj fajla nije neregularna, nego sasvim očekivana i ispravna situacija. Osim toga, kod koji bi tako tretirao kraj fajla bio bi potpuno nepregledan i nelogičan.
- Međutim, šta ako procedura treba da učitava složenije strukture, recimo zapise koji imaju odgovarajući format i strukturu; šta ako se tada nađe na kraj fajla u sledećim slučajevima: a) tačno nakon završetka celog zapisa; b) pre završetka zapisa?
- Sledeći primer: procedura pretražuje neku kolekciju elemenata i traži zadati element: a) koji može, a ne mora da bude u njoj; b) koji bi morao da bude u njoj, jer je to regularno stanje programa? Kako se tretira situacija ako ta procedura ne nađe traženi element?
- Paradigma programiranja i projektovanja softvera pod nazivom *programiranje po ugovoru* (engl. *programming by contract/design by contract*, Bertrand Meyer, 1986, programski jezik Ajfel, *Eiffel*) pretpostavlja preciznu, formalnu specifikaciju interfejsa softverskih komponenata (npr. potprograma ili klase) za koje se definišu logički uslovi (kao logički izrazi) sledećih vrsta:
 - preduslovi (engl. *precondition*): logički uslovi koji moraju da budu zadovoljeni prilikom poziva potprograma da bi taj potprogram mogao da obavi svoj zadatak; ove uslove mora da zadovolji pozivalac pri pozivu potprograma; na primer, uslovi u pogledu stanja globalnih promenljivih ili argumenata poziva potprograma (npr. indeks mora biti unutar granica niza);
 - postuslovi (engl. *postcondition*): logički uslovi koji moraju da budu zadovoljeni na kraju izvršavanja potprograma; ove uslove mora da zadovolji pozvani potprogram; na primer, uslovi u pogledu stanja globalnih podataka ili validnosti povratne vrednosti funkcije;
 - invarijante (engl. *invariant*): logički uslovi vezani za neku komponentu (npr. instancu apstraktnog tipa podataka, objekat klase, modul, strukturu podataka) koji moraju da budu zadovoljeni uvek, osim u tranzijentnim periodima kada se ta komponenta prevodi iz jednog validnog stanja (kada invarijanta važi) u drugo validno stanje (kada invarijanta ponovo važi); na primer, metode klase koje prevode objekat iz jednog regularnog u drugo regularno stanje, tokom koje invarijanta privremeno ne mora da važi.
- Prema tome, preduslovi su obaveze koje mora da izvrši pozivalac prema pozvanom potprogramu, da bi taj potprogram mogao da izvrši svoju obavezu; postuslovi su obaveze koje potprogram mora da ispunji ukoliko je pozivalac ispunio svoje obaveze; zato oni predstavljaju ugovor (engl. *contract*) između ove dve strane.
- Preporučeni stil tretiranja izuzetaka jeste taj da se, prema ovoj paradigmi, izuzetkom tretira nemogućnost zadovoljenja nekog od ovih logičkih uslova, i samo to (sve drugo nisu izuzeci):
 - preduslovi: pozvani potprogram, ukoliko nije ispunjen njegov preduslov (npr. neregularan argument) treba da podigne izuzetak; na primer, neispravan parametar;
 - postuslovi: ukoliko potprogram nije u mogućnosti da ispunji svoje postuslove, treba da signalizira izuzetak svom pozivaocu; na primer, funkcija ne može da kreira povratnu vrednost ili ne može da uspostavi njenu invarijantu, ne može da pronađe ono što bi morala da pronađe i vrati, i slično;

- invarijante: ukoliko metoda klase ne može da očuva invarijantu, treba da baci izuzetak; invarijante su zapravo preduslovi i postuslovi koji važe za objekat na ulazu i izlazu svake metode klase.
- Jedna tehnika za apstrahovano ispitivanje uslova - tzv. tvrdnje (engl. *assertion*):

```
assert(i>=0 && i<this->size);
```
- U standardnoj biblioteci za C++, `assert` je definisan kao makro koji zavisi od drugog definisanog makroa `NDEBUG` koji nije definisan u biblioteci, već se može definisati (uključiti ili isključiti) u okruženju prevodioca:
 - ako je `NDEBUG` definisan (uključen), makro `assert` nema nikavog efekta (zamenjuje se sa `((void)0)`);
 - u suprotnom, zamenjuje se implementacijom koja ispituje vrednost datog skalarnog izraza; ako je ta vrednost nula, na standardni izlaz za greške ispisuje informacije o mestu u programu i završava program.
- Ovakvi slični makroi ili funkcije mogu da se koriste za ispitivanje preduslova, postuslova i invarijanti, ali tako da podižu izuzetke ukoliko uslov nije ispunjen.
- Tretman izuzetaka, odnosno izuzetnih situacija koje su detektovane, može da bude na jednom od sledećih nivoa sigurnosti (engl. *safety*), od najstrožijeg ka najlabavijem:
 - *Nothrow* (ili *nofail*) garancija: potprogram nikada ne baca izuzetak (funkcije označene kao `noexcept`); to znači da nema preduslove i uvek će sigurno očuvati sve invarijante i ispuniti svoje postuslove, ili pak greške sakriva ili tretira na drugi način; ovakvo ponašanje očekuje se od destruktora (oni su implicitno `noexcept`) i *move* konstruktora i operatora dodele, kao i drugih funkcija koje se implicitno pozivaju tokom prosleđivanja bačenog izuzetka do hvatača.
 - Jaka garancija sigurnosti od izuzetaka (engl. *strong exception safety guarantee*): ako potprogram baci izuzetak, stanje programa biće vraćeno na ono pre poziva tog potprograma (tzv. *rollback*) - sam potprogram je napravljen tako da to stanje povрати ili očuva.
 - Osnovna garancija sigurnosti od izuzetaka (engl. *basic exception safety guarantee*): ako potprogram baci izuzetak, stanje programa biće validno (iako može biti različito od onog pre poziva tog potprograma), tj. sve invarijante biće očuvane.
 - Nikakva garancija: ukoliko se baci izuzetak, program može ostati u nekorektnom stanju; ovakve pojave predstavljaju bagove (engl. *bug*), odnosno nepravilnosti u programu koje treba rešavati.
- Preporuka je da svaki potprogram podrži najstrožiji od ovih redom iznesenih nivoa koji je moguće ispuniti.

Zadaci

2.1. Programiranje u *N* verzija

Korišćenjem programiranja u *N* verzija realizovati pouzdan potprogram koji sortira niz celih brojeva.

Rešenje:

```
void quickSort(int arr[], int size);  
void insertSort(int arr[], int size);  
void bubbleSort(int arr[], int size);
```

```
void arrcpy (const int from[], int to[], int size) {
    for (int i=0; i<size; i++) to[i]=from[i];
}

int arrcmp (const int arr1[], const int arr2[], int size) {
    for (int i=0; i<size; i++)
        if (arr1[i]<arr2[i]) return -1;
        else
            if (arr1[i]>arr2[i]) return 1;
    return 0;
}

const int* vote (const int arr1[], const int arr2[], const int arr3[], int
size) {
    if (arrcmp(arr1,arr2,size)==0) return arr1;
    if (arrcmp(arr1,arr3,size)==0) return arr1;
    if (arrcmp(arr2,arr3,size)==0) return arr2;
    return nullptr;
}

void safeSort (int arr[], int size) {
    int *arr1 = nullptr, *arr2 = nullptr, *arr3 = nullptr;
    try {
        arr1 = new int[size];
        arr2 = new int[size];
        arr3 = new int[size];

        arrcpy(arr,arr1,size);
        arrcpy(arr,arr2,size);
        arrcpy(arr,arr3,size);

        quickSort(arr1,size);
        insertSort(arr2,size);
        bubbleSort(arr3,size);

        const int* result = vote(arr1,arr2,arr3,size);
        if (!result)
            throw std::exception("Array sort failed: Voting impossible.");

        arrcpy(result,arr,size);
        delete [] arr1;
        delete [] arr2;
        delete [] arr3;
    }

    catch (...) {
        delete [] arr1;
        delete [] arr2;
        delete [] arr3;
        throw;
    }
}
```

2.2. BER tehnike

Korišćenjem tehniku BER, nalik na blokove oporavka, realizovati pouzdan potprogram koji sortira niz celih brojeva.

Rešenje:

```

void quickSort(int arr[], int size);
void insertSort(int arr[], int size);
void bubbleSort(int arr[], int size);

void arrcpy (const int from[], int to[], int size) {
    for (int i=0; i<size; i++) to[i]=from[i];
}

bool isSorted (const int arr[], int size, int checksum) {
    int cs = 0;
    for (int i=0; i<size; checksum+=arr[i], i++)
        if (i<size-1 && arr[i]>arr[i+1]) return false;
    if (cs!=checksum) return false;
    return true;
}

void safeSort (int arr[], int size) {
    int *arrOld = nullptr, checksum = 0;
    try {
        arrOld = new int[size];
        for (int i=0; i<size; i++) checksum+=arr[i];

        // Establish a recovery point
        arrcpy(arr,arrOld,size);

        // Primary action
        quickSort(arr,size);
        // Acceptance test
        if (isSorted(arr,size,checksum)) { delete [] arrOld; return; }

        // Restore recovery point
        arrcpy(arrOld,arr,size);
        // Alternative action
        insertSort(arr,size);
        // Acceptance test
        if (isSorted(arr,size,checksum)) { delete [] arrOld; return; }

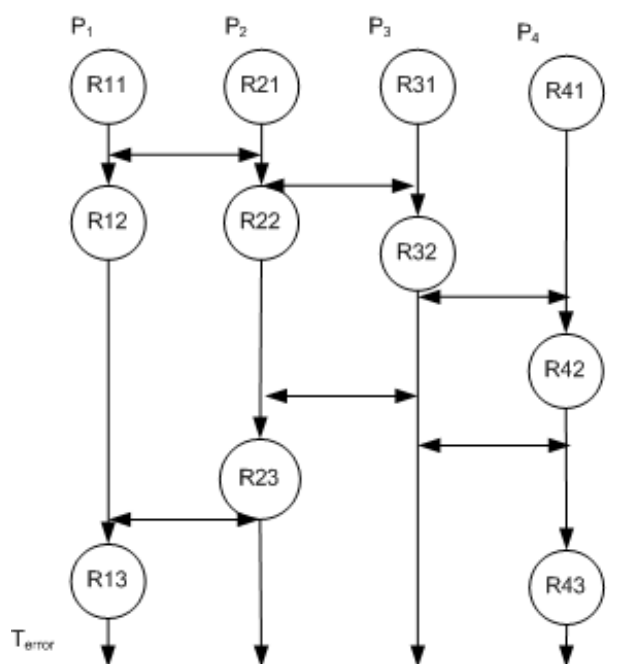
        // Restore recovery point
        arrcpy(arrOld,arr,size);
        // Alternative action
        insertSort(arr,size);
        if (isSorted(arr,size,checksum)) { delete [] arrOld; return; }

        throw std::exception("Array sort failed: Acceptance test cannot be
passed.");
    }
    catch (...) {
        delete [] arrOld;
        throw;
    }
}

```


2.3. Blokovi oporavka

Na slici je šematski prikazano izvršavanje četiri procesa P_1 do P_4 sa svojim tačkama oporavka R_{ij} i međusobnom komunikacijom. Objasniti šta se dešava u slučaju nastanka greške u naznačenom trenutku T_{error} , ako je greška nastala u jednom od ta četiri procesa. Odgovor dati za svaki pojedinačni slučaj od ta četiri.



Rešenje:

	P1	P2	P3	P4
P1	R13	-	-	-
P2	R12	R23	-	-
P3	R12	R22	R32	R41
P4	-	-	-	R43

Objašnjenje: Ako je u trenutku T_{error} nastala greška u procesu označenom vrstom u tabeli, procesi označeni u kolonama će se vratiti u datu tačku oporavka.

Zadaci za samostalan rad

2.4.

Dat je interfejs klase koja omogućava čitanje znakova sa terminala, a poseduje i operaciju za odbacivanje svih preostalih znakova na tekućoj liniji. Operacije ove klase podižu izuzetak tipa `IOError`.

```
class InputStream {
public:
    char get    ();
    void flush ();
};
```

Klasa `Look` sadrži funkciju `read()` koja pretražuje tekuću ulaznu liniju i traži sledeće znakove interpunkcije: zarez, tačku i tačku-zarez. Ova funkcija će vratiti prvi sledeći znak interpunkcije na koji naiđe ili podići izuzetak tipa `IllegalPunctuation` ukoliko naiđe na znak koji nije alfanumerički. Ako se tokom učitavanja znakova sa ulazne linije dogodi izuzetak tipa `IOException`, on će biti prosleđen pozivaocu funkcije `read()`. Kada vrati regularan znak interpunkcije, ova funkcija treba da odbaci preostale znakove na liniji.

Realizovati klasu `Look` korišćenjem klase `InputStream`. U klasi `Look` zatim realizovati i funkciju `getPunctuation()` koja će uvek vratiti sledeći znak interpunkcije, bez obzira na postojanje neregularnih znakova i grešaka na ulazu. Pretpostaviti da je ulazni tok neograničen, da uvek sadrži neki znak interpunkcije i da se greške na ulazu dešavaju slučajno.

2.5.

Korišćenjem redundantnih pokazivača implementirati ulančanu listu otpornu na korupciju strukture.

2.6.

Posmatra se neki sistem za kontrolu procesa u kome se gas zagreva u nekoj posudi. Posuda je okružena hladnjakom koji smanjuje njegovu temperaturu odvođenjem toplote preko tečnosti za hlađenje. Postoji takođe i slavina čijim se otvaranjem gas ispušta u atmosferu. Interfejs klase koja upravlja ovim procesom dat je u nastavku. Zbog sigurnosnih razloga, klasa prepoznaje nekoliko neregularnih uslova koji se korisniku dojavljuju putem izuzetaka. Izuzetak tipa `HeaterStuckOn` signalizira da grejač nije moguće isključiti jer se prekidač zaglavio. Izuzetak tipa `TemperatureStillRising` signalizira da hladnjak nije u stanju da snizi temperaturu gasa povećanjem protoka tečnosti za hlađenje. Konačno, izuzetak tipa `ValveStuck` signalizira da se slavina zaglavila i da je nije moguće otvoriti. Operacija `panic()` podiže znak za uzbunu.

```
class TemperatureControl {
public:
    void heaterOn ();
    void heaterOff () throw (HeaterStuckOn);
    void increaseCoolant () throw (TemperatureStillRising);
    void openValve () throw (ValveStuck);
    void panic();
};
```

Korišćenjem ove klase napisati funkciju koja, kada se pozove, pokušava da isključi grejač. Ukoliko se on zaglavio, treba povećati protok hladnjaka. Ukoliko temperatura i dalje raste, potrebno je otvoriti slavinu za izbacivanje gasa. Ukoliko ni to ne uspe, treba dići znak za uzbunu.

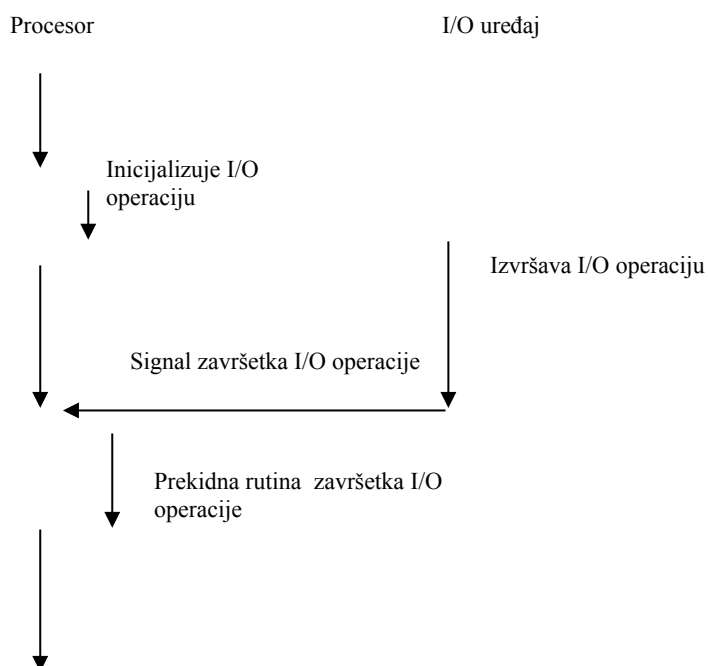
III Osnove konkurentnog programiranja

Konkurentnost i procesi

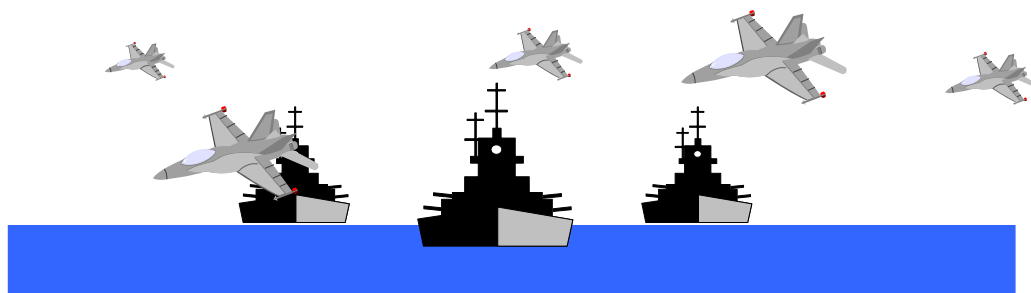
Konkurentno programiranje

- Praktično svi RT sistemi su inherentno *uporedni* (*konkurentni*, engl. *concurrent*), jer oni najčešće nadziru ili upravljaju okruženjem u kom se procesi ili događaji dešavaju uporedo, istovremeno ili u nepredvidivim trenucima.
- Klasično, sekvencijalno programiranje ne obezbeđuje koncepte kojim se u programu može neposredno i apstraktno specificovati način obrade uporednih procesa i događaja iz okruženja, ili uporedno izvršavanje različitih programskih aktivnosti, već podrazumeva semantiku isključivo sekvencijalnog izvršavanja pojedinačnih aktivnosti (naredbi, potprograma).
- Zbog toga programiranje softvera koji treba da nadzire okruženje i upravlja njime ili izvršava uporedne aktivnosti klasičnim, sekvencijalnim programiranjem može da bude izuzetno teško i nepogodno iz sledećih razloga:
 - Programer mora sam da konstruiše i implementira sistem tako da ciklično izvršava uporedne aktivnosti odnosno reakcije na asinhrono događaje.
 - Kako se pojedina aktivnost ne bi predugo izvršavala i tako neprihvatljivo odložila izvršavanje drugih aktivnosti ili hitnih reakcija na druge događaje, izvršavanje pojedine aktivnosti u svakom ciklusu mora se ograničiti, a duže aktivnosti deliti na manje delove, od kojih se svaki izvršava u posebnom ciklusu.
 - Ovakva konstrukcija program čini mnogo težim za pravljenje, razumevanje i održavanje, jer se u program ugrađuju kontrolne strukture koje nisu deo suštine zadataka samog sistema, a izdvojene aktivnosti je teže pratiti i programirati nastavak izvršavanja (prenos konteksta) iz jednog ciklusa u drugi.
 - Uključivanje obaveze ispunjenja vremenskih rokova i provera tog ispunjenja postaje mnogo teže.
 - Ugradnja koda za obradu grešaka je problematičnija.
 - Teško je obezbediti paralelno izvršavanje programa na više procesora, ukoliko za to postoji mogućnost i potreba.
- Da bi projektovanje i programiranje ovakvih aplikacija bilo lakše, a programi lakši za razumevanje i održavanje, programski jezik treba da bude *konkurentan*, odnosno da podržava koncepte *konkurentnog programiranja*.
- *Konkurentno programiranje* (engl. *concurrent programming*) je programska paradigma koja obuhvata koncepte, notaciju i tehnike za izražavanje *potencijalnog paralelizma* dešavanja i izvršavanja, kao i za rešavanje problema sinhronizacije i komunikacije koji nastaju zbog takvog paralelizma.
- *Konkurentnost* označava *potencijalni, logički* paralelizam: izvršavanje aktivnosti koje su u programu označene kao konkurentne može se fizički implementirati na bilo koji od sledećih načina:
 - Fizički sekvencijalno, *multiprogramiranjem* na jednom procesoru, čime se delovi uporednih aktivnosti izvršavaju naizmenično, ali sekvencijalno, a ne istovremeno (paralelno), pri čemu je to deljenje na podaktivnosti i čuvanje i restauracija konteksta njihovog izvršavanja od jedne do druge aktivacije odgovornost implementacije i potpuno je transparentno za program i programera.

- Fizički istovremeno, tj. *paralelno* (paralelizam označava stvarnu uporednost u fizičkom vremenu): a) *multiprocesiranjem*, na sistemu sa više procesora sa zajedničkom operativnom memorijom ili b) *distribuiranim procesiranjem*, na distribuiranom sistemu sa više procesora bez zajedničke memorije (povezanih mrežom preko koje mogu da razmenjuju poruke).
- Način implementacije konkurentnosti je odgovornost platforme (izvršnog okruženja jezika ili operativnog sistema, uz odgovarajuću podršku hardvera) i potpuno je transparentno za program i programera, odnosno nezavisno od konkurentnog programiranja.
- Zbog toga konkurentni jezici pružaju koncepte i konstrukte za neposredno izražavanje potencijalnog paralelizma, pa su stoga izražajniiji nego sekvencijalni jezici, a njihovim korišćenjem konkurentni problem se lakše modeluje, a programiranje postaje lakše.
- Polazna osnova koja važi u konkurentnom programiranju je, dakle, da se ništa ne pretpostavlja o stvarnoj implementaciji potencijalnog paralelizma, tj. da se ne pretpostavlja da se konkurentni procesi zaista izvršavaju fizički paralelno u vremenu ili ne, i na koji se način oni sekvencijalizuju ili prepliću, ukoliko se ne izvršavaju fizički paralelno (nego se izvršavaju na samo jednom procesoru), osim ako to nije eksplicitno ili implicitno definisano semantikom programa ili programskih konstrukata u vidu ograničenja koja se nazivaju *sinhronizacija* (engl. *synchronization*).
- Konkurentno programiranje obezbeđuje način da se:
 - iskoristi procesorsko radno vreme, kako bi procesor mogao da radi neki koristan posao dok čeka da okolni sistemi, tipično ulazno-izlazni uređaji, koji imaju daleko veće vreme odziva, izvrše svoje zadatke: dok jedna aktivnost čeka na završetak duže ulazno-izlazne operacije sa uređajem (koji tipično imaju mnogo duže vreme odziva nego procesor i memorija), procesor može da izvršava druge aktivnosti;
 - na ovaj način se koristi paralelizam u radu procesora i ulazno-izlaznih uređaja; pri tome, ovakva *promena konteksta* (prelazak procesora sa izvršavanja jedne aktivnosti na izvršavanje druge aktivnosti, uz sinhronizaciju sa ulazno-izlaznim uređajima) je transparentna za program i programera, jer je odgovornost implementacije (operativnog sistema):



- kada se u programu izrazi potencijalni paralelizam, platforma koja implementira konkurentnost (tipično operativni sistem) može da iskoristi raspoloživi paralelizam hardvera (postojanje više procesora), kako bi, umesto multiprogramiranja, uporedne aktivnosti izvršio paralelno, a time i mnogo efikasnije; ovo je opet transparentno za program i programera koji ne moraju da vode računa o ovome; ako u programu nije izražen potencijalni paralelizam, onda nema mogućnosti da se ovaj paralelizam u hardveru iskoristi; na primer, obilazak neke složene strukture zbog pretrage neke vrednosti može da se uradi sekvencijalno ili konkurentno (podelom te strukture na podstrukture i konkurentan obilazak tih podstrukture):
- koncepti konkurentnog programiranja olakšavaju modelovanje paralelizma u realnom svetu, odnosno fizikih pojava i aktivnosti objekata koje se u realnosti odvijaju uporedo:



Pojam procesa

- Jedan od najstarijih i najčešće korišćenih modela konkurentnog programiranja, ali ne i jedini, jeste onaj u kom je konkurentan program skup autonomnih sekvencijalnih *procesa* koji se izvršavaju uporedo, odnosno (logički) paralelno.
- *Proces* (engl. *process*) je sekvenca akcija koja se izvršava uporedo sa ostalim takvim sekvencama, odnosno procesima.
- Svaki proces ima svoj sopstveni i nezavisan *tok kontrole* (engl. *thread of control*): sekvencu izvršavanja instrukcija/naredbi, sa semantikom koja je uobičajena u sekvencijalnim jezicima (naredna akcija vidi rezultate prethodno izvršenih naredbi), upravljanjem redosledom izvršavanja tih akcija (uslovna grananja, petlje), uključujući i pozive potprograma (sa prenosom argumenata i potencijalnom rekurzijom). Osim ako to nije drugačije uređeno programom, kontrola toka svakog procesa (tj. izvršavanje njegove sekvence akcija) napreduje nezavisno od drugih procesa i uporedo sa njima.
- Proces predstavlja deo programskog koda zajedno sa strukturama podataka koje omogućuju uporedno (konkurentno) izvršavanje tog programskog koda sa ostalim procesima. Koncept procesa omogućuje izvršavanje dela programskog koda tako da su svi podaci koji su definisani kao lokalni za taj deo programskog koda zapravo lokalni za jedno izvršavanje tog koda, i da se njihove instance razlikuju od instanci istih podataka istih delova tog koda, ali različitih procesa. Ova lokalnost podataka procesa pridruženih jednom izvršavanju datog koda opisuje se kao izvršavanje datog dela koda u *kontekstu* nekog procesa.
- Stvarna implementacija (tj. izvršavanje) skupa procesa obično ima jedan od sledeća tri oblika:

1. *Multiprogramiranje* (engl. *multiprogramming*): izvršavanje procesa se multipleksira na jednom procesoru.
 2. *Multiprocesiranje* (engl. *multiprocessing*): izvršavanje procesa se multipleksira na više procesora koji imaju zajedničku operativnu memoriju (tzv. *multiprocessorski sistem*).
 3. *Distribuirano procesiranje* (engl. *distributed processing*): izvršavanje procesa se multipleksira na više procesora koji nemaju zajedničku memoriju, nego mogu da razmenjuju poruke preko komunikacione mreže koja ih povezuje (tzv. *distribuirani sistem*).
- U terminologiji konkurentnog programiranja razlikuju se obično dve vrste procesa:
 1. Proces na nivou operativnog sistema (engl. *process*). Ovakvi procesi nazivaju se ponekad "teškim" (engl. *heavy-weight*) procesima. Ovakav proces kreira se obično nad celim programom. Pri tome svaki proces ima *sopstveni adresni prostor* (engl. *address space*): ceo memorijski adresni prostor koji vide sve instrukcije datog procesa, u kome se nalaze i instrukcije i podaci svih vrsta skladišta (statički, automatski - lokalni za potprograme, dinamički), uključujući i trag izvršavanja sa ugnežđenim pozivima potprograma (kontrolni stek). Ovaj adresni prostor je nezavisan od prostora drugih takvih procesa (možda kreiranim nad istim programom): ista logička (virtuelna) adresa jednog procesa može imati drugi i nezavisan sadržaj od iste te adrese drugog procesa.
 2. Proces u okviru jednog programa. Ovakvi procesi nazivaju se "lakim" (engl. *light-weight*) ili *nitima* (engl. *thread*). Niti se kreiraju nad delovima jednog programa, najčešće kao tok izvršavanja koji polazi od jednog potprograma. Svi dalji ugnežđeni pozivi ostalih potprograma izvršavaju se u kontekstu date niti. To znači da sve niti unutar jednog programa dele statičke (globalne) i dinamičke podatke. Ono što ih razlikuje je lokalnost automatskih podataka: *svaka nit poseduje svoj kontrolni stek* na kome se kreiraju automatski objekti (alokacioni blokovi potprograma). Kaže se zato da sve niti poseduju *zajednički adresni prostor*, ali različite *tokove kontrole*, odnosno različite kontrolne stekove i pozicije do koje je izvršavanje stiglo u sekvenci izvršavanja instrukcija. Niti mogu da međusobno sarađuju preko istog zajedničkog prostora i podataka u njemu bez ikakvog učešća operativnog sistema.
 - Termin *proces* upotrebljava se često u oba značenja, kao opšti pojam i kao teški proces, u zavisnosti od konteksta. U mnogim kontekstima su analize koje se vrše nezavisno od toga da li ti procesi dele isti ili imaju odvojene adresne prostore.
 - Termin *zadatak* (engl. *task*) se upotrebljava u različitim značenjima, u nekom kontekstu označava pojam procesa kao opšti pojam, u nekim operativnim sistemima označava teški proces, a u nekim jezicima označava nit.
 - Postoji dugogodišnja debata o tome da li programski jezik treba da uključi koncepte konkurentnog programiranja ili da to ostavi operativnom sistemu:
 - Ukoliko jezik uključuje konkurentnost, onda multipleksiranje izvršavanja procesa obavlja "virtuelna mašina" koja je sastavni deo izvršnog okruženja programa (engl. *runtime environment*, *runtime support system*) koga je proizveo prevodilac, uz podršku operativnog sistema ili bez nje. Na primer, jezici Ada i Java imaju ugrađenu konkurentnost.
 - Ukoliko jezik ne podržava konkurentnost, onda se ona može dograditi posebnim delom koda, ili osloniti na usluge operativnog sistema (sistemske pozivi). Jezici C i C++ (u ranijim verzijama) nemaju ugrađenu konkurentnost. Ona se može obezbediti:
 - Izgradnjom sopstvenog izvršnog okruženja, kao što će to biti izvedeno za primer u ovom kursu.
 - Oslanjanjem na usluge operativnog sistema preko nekog programskog interfejsa (engl. *application programming interface*, API) za sistemske pozive iz aplikativnog programa. Primer jednog standardnog API-a za jezik

C/C++ za konkurentno programiranje na operativnim sistemima koji to podržavaju jeste POSIX (*Portable Operating System based on Unix*).

- Raspoređivanje izvršavanja procesa na procesoru (engl. *scheduling*) svakako utiče na vremensko ponašanje programa. Međutim, *logičko ponašanje dobro konstruisanog programa ne sme da zavisi od implementacije raspoređivanja u izvršnom okruženju*, tj. od sledećih parametara:
 - da li se procesi izvršavaju na jednom procesoru (miltiprogramiranjem) ili na više procesora (paralelno);
 - kada se može dogoditi promena konteksta (prelazak izvršavanja sa jednog procesa na drugi), tj. da li se promena konteksta može izvršiti samo u predvidivim momentima, kada sam proces pozove neku uslugu okruženja (sinhrono), ili u nepredvidivim trenucima, nezavisno od toga šta tekući proces trenutno radi, kao posledica spoljašnjih signala zahteva za prekid (asinhrono);
 - kada dođe do promene konteksta, koji proces će nastaviti izvršavanje (raspoređivanje procesa na procesoru, engl. *scheduling*).
- Drugim rečima, posmatrano sa strane programa, podrazumeva se da izvršno okruženje raspoređuje procese na nepredvidiv način. Nikakva pretpostavka o raspoređivanju se ne sme uzeti u obzir ukoliko se želi korektan, prenosiv program. Sva neophodna međuzavisnost između procesa mora se rešiti logikom samog programa i to:
 - *sinhronizacijom procesa* (engl. *process synchronization*): sinhronizacija procesa podrazumeva specifikaciju ograničenja koja moraju biti ispoštovana tokom izvršavanja konkurentnih procesa u pogledu načina i redosleda izvršavanja odnosno preplitanja nekih akcija tih procesa, npr. u smislu da neke grupe akcija različitih procesa ne smeju da se prepliću/izvršavaju uporedo/paralelno, ili neke akcije jednog procesa ne smeju da se izvrše pre nego što je neki uslov ispunjen, npr. ako neki drugi proces nije izvršio neke svoje akcije ili neki objekat nije u određenom stanju i slično; sinhronizacija ograničava skup dozvoljenih načina uporednog izvršavanja procesa, kako bi rezultat njihovog izvršavanja bio logički korektan u svim situacijama, nezavisno od načina implementacije i konkretnog redosleda izvršavanja njihovih pojedinačnih akcija.
 - *komunikacijom između procesa* (engl. *interprocess communication*, IPC): komunikacija podrazumeva razmenu informacija između uporednih procesa.

Predstavljanje procesa

Procesi na jeziku Ada

- Jezik Ada poseduje pojam procesa koji se eksplicitno deklariše kao programski modul nalik potprogramu, samo što je njegovo izvršavanje konkurentno.
- Na jeziku Ada se proces naziva *task* i definiše slično potprogramu, a semantički predstavlja jednu nit:

```
task Process;           -- Task declaration

task body Process is    -- Task definition
  -- Task's local variables
begin
  ...  -- Statements executed in the task
end;
```


- Izvršavanje ovakvog procesa implicitno počinje kad izvršavanje nekog toka kontrole uđe u opseg deklaracije ovakvog procesa (može biti blok naredbi ili telo potprograma ili drugog procesa). Ovakvi procesi su tipično globalni, pa njihovo izvršavanje počinje pokretanjem programa, ali mogu biti definisani i unutar procedure ili drugog procesa.
- Izvršavanje okružujućeg opsega se ne završava dok se ne završe svi u njemu ugrađeni procesi. Izvršavanje samog programa je jedan vrhunski implicitni proces.
- Navedeni oblik procesa omogućava pokretanje jedne instance, odnosno jednog izvršavanja za datu deklaraciju. Ada dozvoljava i definiciju *tipa procesa*, za koji se može kreirati više instanci (npr. kao elementi niza), odnosno više uporednih izvršavanja istog koda, pa i dinamički:

```
task type P;
...
```

```
task body P is
begin
  ...
end;
```

```
ptrP : access P; -- A reference/pointer to the task type P
```

```
...
```

```
ptrP := new P; -- Dynamic creation/activation of the process of type P
```

- Tip procesa može biti i parametrizovan (kao i ostali tipovi). Na primer:

```
type Matrix is array (1..M, 1..N) of Real;
```

```
procedure matAdd (a, b : Matrix, c : out Matrix) is
```

```
  task type MatrixAdder (row : Integer) body is
  begin
    for j in Integer range 1..N loop
      c(row,j) := a(row,j) + b(row,j);
    end loop;
  end;
```

```
  adders : array (1..M) of access MatrixAdder;
```

```
begin
```

```
  for i in Integer range 1..M loop
    adders(i) := new MatrixAdder(i);
  end loop;
```

```
end; -- procedure matAdd waits for completion of all adders(i)
```

Niti na jeziku Java

- Jezik Java podržava niti (engl. *thread*) na potpuno objektno orijentisan način. Skup niti istog tipa (nad istim programskim kodom) koje se mogu kreirati u toku izvršavanja deklariše se kao klasa izvedena iz bibliotečne klase `Thread`. Kod (telo) niti definiše se kao polimorfna operacija `run()` ove klase.
- U toku izvršavanja programa, mogu se kreirati objekti ove klase na uobičajen način. Međutim, izvršavanje niti se mora eksplicitno pokrenuti pozivom operacije `start()` objekta koji predstavlja tu nit.
- Klasa čiji objekti predstavljaju procese, tj. imaju sopstvenu nit toka kontrole, naziva se često *aktivna klasa* (engl. *active class*).
- Primer:

```
public class Actor extends Thread {
    public Actor(...) { // Constructor
        ...
    }

    public void run() {
        ... // Thread body
    }
}
```

```
...
Actor a1 = new Actor(...);
Actor a2 = new Actor(...);
Actor a3 = new Actor(...);
```

```
a1.start();
a2.start();
a3.start();
```

- Nit u Javi završava izvršavanje kada se dogodi nešto od sledećeg:
 - kada se završi operacija `run()`, bilo normalno, bilo zbog neobrađenog izuzetka;
 - kada se pozove operacija `stop()` klase `Thread` za taj objekat; ovoj operaciji se može proslediti referenca na objekat tipa `Throwable`, koji će biti podignut kao izuzetak u određenoj niti; operacija `stop()` nije sigurna, jer bezuslovno oslobađa sve objekte koje je nit zaključala; ovu operaciju zato u principu ne treba pozivati.

Niti na jeziku C++

- U standardnoj biblioteci jezika C++ (počev od verzije C++11) postoji klasa `std::thread` definisana u zaglavlju `<thread>`. Kreiranjem objekta ove klase aktivira se nova nit koja se izvršava nad funkcijom datom u konstruktoru ove klase.
- Konstruktor ove klase kao svoj prvi parametar očekuje pokazivač na funkciju nad kojom će se kreirati nit (može se dostaviti i objekat klase koja ima preklopljen operator poziva funkcije). Opcioni preostali parametri (može ih biti proizvoljno) se prosleđuju kao argumenti poziva te funkcije nad kojom se kreira nit.
- Nit se izvršava nezavisno od niti u kojoj je kreirana. Čekanje na završetak pokrenute niti može se uraditi pozivom operacije `join` te niti (nije dobro pozivati tu funkciju iz više različitih niti).
- Na primer:

```
#include <thread>
using std::thread;

const int N = ...;

void rowAdder (const double a[], const double b[],
               double c[], int size)
{
    for (int i=0; i<size; i++) c[i]=a[i]+b[i];
}

void matrixAdder (const double a[][N], const double b[][N],
                  double c[][N], int size)
{
    thread** adders = new (thread*)[size];
    for (int i=0; i<size; i++) adders[i] = new thread(rowAdder, a, b, c, N);
    for (int i=0; i<size; i++) adders[i]->join();
}
```

```

    for (int i=0; i<size; i++) delete adders[i];
    delete [] adders;
}

```

POSIX niti

- U biblioteci POSIX niti se kreiraju takođe nad funkcijom, kojoj se može dostaviti samo jedan argument tipa `void*`. Ukoliko je funkciji potrebno više argumenata, potrebno ih je složiti u neku strukturu na koju će ukazivati ovaj argument.
- Nit se kreira pozivom sledeće bibliotečne funkcije:

```

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void*(*thread_body)(void*), void *arg);

```

- Prvi parametar ukazuje na strukturu tipa `pthread_t` koja je deskriptor kreirane niti i koja će kasnije biti korišćena za sve druge operacije u kojima treba identifikovati tu nit. Drugi parametar je pokazivač na strukturu kojom se mogu podešavati atributi niti (npr. vreme aktivacije); ukoliko je ovaj argument *null*, nit se kreira sa podrazumevanim atributima. Treći parametar je pokazivač na funkciju nad kojom se kreira nit, a kojoj se dostavlja, kao jedini argument, parametar `arg`.
- Čekanje na završetak niti može se vršiti pozivom funkcije `pthread_join`. Prvi parametar ove funkcije identifikuje nit čiji se završetak čeka. Ovoj funkciji može se dostaviti i drugi argument koji je pokazivač na tip `void*`; u ovaj pokazivač tipa `void*` funkcija koja je izvršavala nit može da prosledi svoj rezultat niti koja izvršava *join*.
- Na primer:

```

#include <pthread.h>

const int N = ...;

struct RowAdderArgs {
    const double *a, *b;
    double* c;
    int size;
};

void* rowAdder (void* arguments) {
    RowAdderArgs* args = (RowAdderArgs*)arguments;
    for (int i=0; i<args->size; i++) args->c[i]=args->a[i]+args->b[i];
    return nullptr;
}

void matrixAdder (const double a[][N], const double b[][N],
                  double c[][N], int size)
{
    RowAdderArgs* args = new RowAdderArgs[size];
    pthread_t* adders = new pthread_t[size];
    for (int i=0; i<size; i++) {
        args[i] = {a, b, c, N};
        pthread_create(&adders[i], nullptr, rowAdder, &args[i])
    }
    for (int i=0; i<size; i++) pthread_join(&adders[i], nullptr);
    delete [] adders;
    delete [] args;
}

```

Interakcija između procesa

- Ukoliko su procesi potpuno nezavisni, odnosno nikako ne interaguju, rezultat njihovog izvršavanja nikako ne zavisi od načina i redosleda izvršavanja njihovih akcija, pa problema nikada nema u bilo kom izvršavanju na bilo kojoj platformi (multiprogramiranje ili multiprocesiranje, način promene konteksta i raspoređivanja).
- Međutim, procesi su retko nezavisni, već međusobno interaguju razmenom informacija ili pristupom istim deljenim resursima (podacima, logičkim objektima ili uređajima). Na jednom računaru procesi nikad nisu nezavisni, odnosno uvek makar implicitno interaguju, jer koriste iste fizičke i logičke resurse računara (procesor, memorija, uređaji, fajlovi). Osnovni problemi povezani sa konkurentnim programiranjem upravo i proističu iz interakcije uporednih procesa.
- Da bi se obezbedila korektnost izvršavanja uporednih procesa u svim mogućim izvršavanjima, nezavisno od platforme (načina izvršavanja), potrebno je da procesi interaguju na definisan i kontrolisan način. Ovakva kontrolisana interakcija podrazumeva sledeće:
 - *sinhronizaciju procesa* (engl. *process synchronization*): sinhronizacija procesa podrazumeva specifikaciju ograničenja koja moraju biti ispoštovana tokom izvršavanja konkurentnih procesa u pogledu načina i redosleda izvršavanja odnosno preplitanja nekih akcija tih procesa, npr. u smislu da neke grupe akcija različitih procesa ne smeju da se prepliću/izvršavaju uporedo/paralelno, ili neke akcije jednog procesa ne smeju da se izvrše pre nego što je neki uslov ispunjen, npr. ako neki drugi proces nije izvršio neke svoje akcije ili neki objekat nije u određenom stanju i slično; sinhronizacija ograničava skup dozvoljenih načina uporednog izvršavanja procesa, kako bi rezultat njihovog izvršavanja bio logički korektan u svim situacijama, nezavisno od načina implementacije i konkretnog redosleda izvršavanja njihovih pojedinačnih akcija.
 - *komunikacijom između procesa* (engl. *interprocess communication*, IPC): komunikacija podrazumeva razmenu informacija između uporednih procesa.
- Pojmovi sinhronizacije i komunikacije su međusobno povezani, jer neki oblici komunikacije podrazumevaju prethodnu sinhronizaciju, dok se sinhronizacija može smatrati nekom vrstom komunikacijom bez razmene sadržaja.
- Međuprocena komunikacija se obično zasniva na jednom od dva *logička* modela, odnosno programske paradigme:
 - *deljena promenljiva* (engl. *shared variable*) je objekat/podatak/promenljiva kome može pristupati više procesa; komunikacija se obavlja tako što neki proces ili procesi upisuju u deljeni podatak/promenljivu, odnosno menjaju stanje deljenog objekta, dok drugi proces ili procesi čitaju taj deljeni podatak/promenljivu, odnosno ispituju stanje deljenog objekta;
 - *razmena poruka* (engl. *message passing*) podrazumeva eksplicitnu razmenu informacija između procesa u vidu slanja i prijema poruka koje putuju od jednog do drugog procesa eventualno preko nekog posrednika.
- Izbor modela komunikacije je stvar konstrukcije programskog jezika ili operativnog sistema. On ne implicira nikakav poseban metod implementacije. Naime, deljene promenljive je jednostavno implementirati ukoliko procesori koji izvršavaju uporedne procese imaju zajedničku memoriju. Međutim, deljene promenljive se mogu, doduše nešto teže, implementirati i na distribuiranom sistemu bez deljene memorije. Slično, razmena poruka se može implementirati i na multiprocesorskim i na distribuiranim sistemima.
- Osim toga, aplikacija iste funkcionalnosti se u principu može programirati korišćenjem bilo kog od ova dva modela, s tim da je za neke probleme neki od ovih modela pogodniji.

- Oba modela se široko koriste u praksi i pojavljuju se u izuzetno mnogo pojava u različitim programskim jezicima, bibliotekama za programiranje i operativnim sistemima. Štaviše, po pravilu se koncept koji podržava jedan model, na jednom nivou apstrakcije, implementira konceptima po drugom modelu na nižem nivou apstrakcije. Na primer, u nekoj aplikaciji procesi mogu komunicirati po modelu deljenog objekta, npr. preko deljenog fajla ili baze podataka. Međutim, taj fajl ili baza podataka mogu biti na udaljenom računaru, pa operativni sistemi na različitim računarima komuniciraju razmenom poruka, kako bi obezbedili udaljen pristup fajlu ili bazi podataka. Implementacija protokola za razmenu poruka između udaljenih računara, unutar operativnog sistema, opet uključuje komunikaciju između internih niti ili procesa preko zajedničkih bafera ili drugih struktura podataka, kao deljenih objekata.
- Ova dva modela detaljnije su obrađena u naredna dva poglavlja.

Implementacija niti

- U ovom kursu biće prikazana realizacija jednog jezgra (engl. *kernel*) višeprocenog sistema sa nitima (engl. *multithreaded kernel*) na jeziku C++, koji može da služi kao izvršno okruženje za konkurentni korisnički program. Ova realizacija biće nazivana "školsko jezgro".
- Kod ovog sistema aplikativni sloj softvera treba da se poveže sa kodom Jezgra da bi se dobio kompletan izvršni program koji ne zahteva nikakvu softversku podlogu (nije mu potreban operativni sistem). Ovo je pogodno za ugrađene sisteme. Prema tome, veza između višeg sloja softvera i Jezgra je na nivou izvornog koda i zajedničkog povezivanja, a ne kao kod složenih operativnih sistema, gde se sistemski pozivi rešavaju u vreme izvršavanja, najčešće preko softverskih prekida.
- Na nivou aplikativnog sloja softvera, želja je da se postigne semantika analogna onoj na jeziku Java: nit je aktivan objekat koji poseduje sopstveni tok kontrole (sopstveni stek poziva).
- Nit se može kreirati nad nekom globalnom funkcijom. Pri tome se svi ugnežđeni pozivi, zajedno sa svojim automatskim objektima, odvijaju u sopstvenom kontekstu te niti. Na primer, korisnički program može da izgleda ovako:

```
#include "kernel.h" // uključivanje deklaracija Jezgra
#include <iostream.h>
```

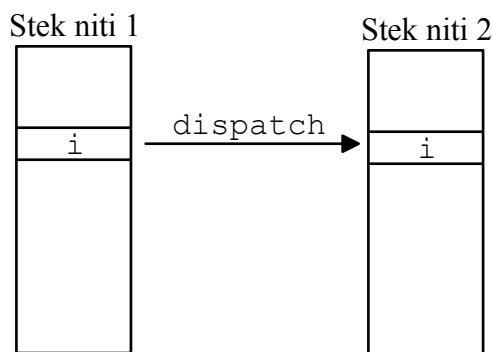
```
void threadBody () {
    for (int i=0; i<3; i++) {
        cout<<i<<"\n";
        dispatch();
    }
}
```

```
void userMain () {
    Thread* t1=new Thread(threadBody);
    Thread* t2=new Thread(threadBody);
    t1->start();
    t2->start();
    dispatch();
}
```

- Funkcija `threadBody()` predstavlja telo (programski kod) niti. Funkcija `dispatch()` predstavlja eksplicitni zahtev za preuzimanje (dodelu procesora drugoj

niti), slično kao kod koncepta korutina, samo što se ne imenuje nit koja preuzima izvršavanje.

- Funkcija `userMain()` predstavlja početnu nit aplikativnog, korisničkog dela programa. Funkcija `main()` nalazi se u nadležnosti Jezgra, pa korisniku nije dostupna. Jezgro inicijalno kreira jednu nit nad obaveznom funkcijom `userMain()`.
- U ovom primeru obe niti imaju isti kod, ali svaka poseduje svoj stek poziva, na kome se kreira automatski objekat `i`. Kada dođe do preuzimanja u funkciji `dispatch()`, Jezgro obezbeđuje pamćenje konteksta tekuće niti i povratak konteksta niti koja je izabrana za tekuću, što znači da se dalje izvršavanje odvija na steku nove tekuće niti. Ovo prikazuje sledeća slika:



Promena konteksta

- Promena konteksta (engl. *context switch*) podrazumeva da procesor napušta kontekst izvršavanja jednog procesa i prelazi u kontekst izvršavanja drugog procesa.
- Tipično se operativni sistemi konstruišu tako da postoje dva režima rada, koja su obično podržana i od strane procesora: sistemski (privilegovani) i korisnički (neprivilegovani). U sistemskom režimu dozvoljeno je izvršavanje raznih sistemskih operacija, kao što je pristup do nekih područja memorije koji su zaštićeni od korisničkih programa. Osim toga, kada postoji mogućnost da se pojavi prekid kao posledica nekog spoljašnjeg događaja na koji sistem treba da reaguje, potrebno je da se sistemski delovi programa izvršavaju neprekidivo, bez promene konteksta, kako ne bi došlo do poremećaja sistemskih delova podataka. U realizaciji ovog Jezgra naznačena su mesta prelaska u sistemski i korisnički režim. Prelaz na sistemski režim obavlja funkcija `lock()`, a na korisnički funkcija `unlock()`. Sve kritične sistemske sekcije uokvirene su u par poziva ovih funkcija. Njihova realizacija je zavisna od platforme i za sada je prazna:

```
void lock    () {} // Switch to kernel mode
void unlock () {} // Switch to user mode
```

- Kada dolazi do promene konteksta, u najjednostavnijem slučaju eksplicitnog pomoću funkcije `dispatch()`, Jezgro treba da uradi sledeće:
 1. Sačuva kontekst niti koja je bila tekuća (koja se izvršavala, engl. *running*).
 2. Smesti nit koja je bila tekuća u red niti koje su spremne za izvršavanje (engl. *ready*).
 3. Izabere nit koja će sledeća biti tekuća iz reda niti koje su spremne.
 4. Povrati kontekst novoizabrane niti i nastavi izvršavanje.
- Čuvanje konteksta niti znači sledeće: vrednosti svih relevantnih registara procesora čuvaju se u nekoj strukturi podataka da bi se kasnije mogle povratiti. Ova struktura naziva se najčešće PCB (engl. *process control block*). Povratak konteksta znači smeštanje sačuvanih vrednosti registara iz PCB u same registre procesora.

- U registre spada pokazivač adrese naredne instrukcije PC (engl. *program counter*), koji čuva informaciju o lokalnosti toka izvršavanja niti (vezano za instrukcijski kod), i pokazivač steka (engl. *stack pointer*, SP), koji čuva informaciju vezanu za lokalnost podataka niti. Kada se u PC i SP povrate vrednosti sačuvane u PCB-u, dalje izvršavanje nastaviće od mesta gde je to izvršavanje prekinuto, tj. gde ukazuje sačuvani PC, i koristiće upravo stek na koji ukazuje SP, čime se postiže svojstvo konkurentnosti niti: lokalnost automatskih podataka, odnosno sopstveni tok kontrole.
- U standardnoj biblioteci jezika C (pa time i C++) definisane su dve funkcije koje "sakrivaju" neposredno baratanje samim registrima procesora, pa se njihovim korišćenjem može dobiti potpuno prenosiv program.
- Deklaracije ovih funkcija nalaze se u `<setjmp.h>` i izgledaju ovako:

```
int  setjmp (jmp_buf context);
void longjmp (jmp_buf context, int value);
```

- Tip `jmp_buf` deklarisan je u istom zaglavlju i predstavlja zapravo PCB. To je struktura koja čuva sve relevantne, programski dostupne registre procesora čije su vrednosti bitne za kontekst izvršavanja C programa prevedenog pomoću datog prevodioca na datom procesoru.
- Funkcija `setjmp()` vrši smeštanje vrednosti registara u strukturu `jmp_buf`. Pri tom smeštanju ova funkcija vraća rezultat 0. Funkcija `longjmp()` vrši povratak konteksta sačuvanog u `jmp_buf`, što znači da izvršavanje vraća na poziciju steka koja je sačuvana pomoću odgovarajućeg `setjmp()`. Pri tome se izvršavanje nastavlja sa onog mesta gde je pozvana `setjmp()`, s tim da sada `setjmp()` vraća onu vrednost koju je dostavljena pozivu `longjmp()` (to mora biti vrednost različita od 0).
- Prema tome, pri čuvanju konteksta, `setjmp()` vraća 0. Kada se kontekst povrat iz `longjmp()`, dobija se efekat da odgovarajući `setjmp()` vraća vrednost različitu od 0. Veoma je važno da se pazi na sledeće: od trenutka čuvanja konteksta pomoću `setjmp()`, do trenutka povratka pomoću `longjmp()`, izvršavanje u kome je `setjmp()` *ne sme* da se vrati iz funkcije koja neposredno okružuje poziv `setjmp()`, jer bi se time stek narušio, pa povratak pomoću `longjmp()` dovodi do kraha sistema.
- Tipična upotreba ovih funkcija za potrebe realizacije korutina može da bude ovakva:

```
if (setjmp(runningThread->context)==0) {

    // Sačuvan je kontekst.
    // Može da se pređe na neki drugi,
    // i da se njegov kontekst povratu sa:
    longjmp(runningThread->context,1);

} else {
    // Ovde je povraćen kontekst onoga koji je sačuvan u setjmp()
}
```

- U realizaciji Jezgra ovi pozivi su "upakovani" u OO okvire. Nit je predstavljena klasom `Thread` koja poseduje atribut tipa `jmp_buf` (kontekst). Funkcija članica `resume()` vrši povratak konteksta jednostavnim pozivom `longjmp()`. Funkcija članica `setContext()` čuva kontekst pozivom `setjmp()`. Kako se iz ove funkcije ne sme vratiti pre povratka konteksta, ova funkcija je samo logički okvir i *mora* biti prava *inline* funkcija, kako prevodilac ne bi generisao kod za poziv i povratak iz ove funkcije `setContext()`:

```
// WARNING: This function MUST be truely inline!
inline int Thread::setContext () {
    return setjmp(myContext);
}
```

```
}
```

```
void Thread::resume () {
    longjmp(myContext,1);
}
```

- Klasa `Scheduler` realizuje raspoređivanje. U njoj se nalazi red spremnih niti (engl. *ready*), kao i protokol raspoređivanja. Funkcija `get()` ove klase vraća nit koja je na redu za izvršavanje, a funkcija `put()` stavlja novu spremnu nit u red.
- Klasa `Scheduler` poseduje samo jedan jedini objekat u sistemu (engl. *Singleton*). Ovaj jedini objekat sakriven je unutar klase kao statički objekat. Otkrivena je samo statička funkcija `Instance()` koja vraća pokazivač na ovaj objekat. Konstruktor ove klase sakriven je od prisupa spolja. Na ovaj način korisnici klase `Scheduler` ne mogu kreirati objekte ove klase, već je to u nadležnosti same te klase, čime se garantuje jedinstvenost objekta. Osim toga, korisnici ove klase ne moraju da znaju ime tog jedinog objekta, već im je dovoljan interfejs same klase i pristup do statičke funkcije `Instance()`. Ovakav projektni šablon (engl. *design pattern*) naziva se *Singleton*.
- Najzad, statička funkcija `Thread::dispatch()` koju jednostavno poziva globalna funkcija `dispatch()` izgleda jednostavno:

```
void Thread::dispatch () {
    lock ();
    if (runningThread->setContext()==0) {

        // Context switch:
        Scheduler::Instance()->put(runningThread);
        runningThread = Scheduler::Instance()->get();
        runningThread->resume();

    } else {
        unlock ();
        return;
    }
}
```

- Treba primetiti sledeće: deo funkcije `dispatch()` iza poziva `setContext()`, a pre poziva `resume()`, radi i dalje na steku prethodno tekuće niti (pozivi funkcija klase `Scheduler`). Tek od poziva `resume()` prelazi se na stek nove tekuće niti. Ovo nije nikakav problem, jer taj deo predstavlja "đubre" na steku iznad granice koja je zapamćena u `setContext()`. Prilikom povratka konteksta prethodne niti, izvršavanje će se nastaviti od zapamćene granice steka, ispod ovog "đubreta".

Raspoređivanje

- Kao što je opisano, klasa `Scheduler` realizuje apstrakciju koja obavlja skladištenje spremnih niti, kao i raspoređivanje. Pod raspoređivanjem se smatra izbor one niti koja je na redu za izvršavanje. Ovo obavlja funkcija članica `get()`. Funkcija `put()` smešta novu nit u red spremnih.
- U ovoj realizaciji obezbeđen je samo jednostavan kružni (engl. *round-robin*) raspoređivač, korišćenjem realizovanog reda čekanja.
- Klasa `Scheduler` je realizovana kao *Singleton*, što znači da ima samo jedan objekat. Ovaj objekat je zapravo lokalni statički objekat funkcije `Instance()`:

```
Scheduler* Scheduler::Instance () {
    static Scheduler instance;
```



```
    return &instance;  
}
```

Kreiranje niti

- Nit je predstavljena klasom `Thread`. Kao što je pokazano, korisnik kreira nit kreiranjem objekta ove klase. U tradicionalnom pristupu nit se kreira nad nekom globalnom funkcijom programa. Međutim, ovaj pristup nije dovoljno fleksibilan. Naime, često je potpuno beskorisno kreirati više niti nad istom funkcijom ako one ne mogu da se međusobno razlikuju, npr. pomoću argumenata pozvane funkcije. Zbog toga se u ovakvim tradicionalnim sistemima često omogućuje da korisnička funkcija nad kojom se kreira nit dobije neki argument prilikom kreiranja niti. Ipak, broj i tipovi ovih argumenata su fiksni, definisanim samim sistemom, pa ovakav pristup nije u duhu jezika C++.
- U realizaciji ovog Jezgra, pored navedenog tradicionalnog pristupa, omogućen je i OO pristup u kome se nit može definisati kao aktivan objekat. Taj objekat je objekat neke klase izvedene iz klase `Thread` koju definiše korisnik. Nit se kreira nad polimorfnom operacijom `run()` klase `Thread` koju korisnik može da redefiniše u izvedenoj klasi. Na ovaj način svaki aktivni objekat iste klase poseduje sopstvene attribute, pa na taj način mogu da se razlikuju aktivni objekti iste klase (niti nad istom funkcijom). Suština je zapravo u tome da jedini (doduše skriveni) argument funkcije `run()` nad kojom se kreira nit jeste pokazivač `this`, koji ukazuje na čitavu strukturu proizvoljnih atributa objekta.
- Prema tome, interfejs klase `Thread` prema korisnicima izgleda ovako:

```
class Thread {  
public:  
  
    Thread ();  
    Thread (void (*body)());  
    void start ();  
  
protected:  
  
    virtual void run () {}  
  
};
```

- Konstruktor bez argumenata kreira nit nad polimorfnom operacijom `run()`. Drugi konstruktor kreira nit nad globalnom funkcijom na koju ukazuje pokazivač-argument. Funkcija `run()` ima podrazumevano prazno telo, tako da se i ne mora redefinisati, pa klasa `Thread` nije apstraktna.
- Funkcija `start()` služi za eksplicitno pokretanje niti. Implicitno pokretanje moglo je da se obezbedi tako što se nit pokreće odmah po kreiranju, što bi se realizovalo unutar konstruktora osnovne klase `Thread`. Međutim, ovakav pristup nije dobar, jer se konstruktor osnovne klase izvršava pre konstruktora izvedene klase i njenih članova, pa se može dogoditi da novokreirana nit počne izvršavanje pre nego što je kompletan objekat izvedene klase kreiran. Kako nit izvršava redefinisanu funkciju `run()`, a unutar ove funkcije može da se pristupa članovima, moglo bi da dođe do konflikta.
- Treba primetiti da se konstruktor klase `Thread`, odnosno kreiranje nove niti, izvršava u kontekstu one niti koja poziva taj konstruktor, odnosno u kontekstu niti koja kreira novu nit.
- Prilikom kreiranja nove niti ključne i kritične su dve stvari: 1) kreirati novi stek za novu nit i 2) kreirati početni kontekst te niti, kako bi ona mogla da se pokrene kada dođe na red.
- Kreiranje novog steka vrši se prostom alokacijom niza bajtova u slobodnoj memoriji, unutar konstruktora klase `Thread`:

```
Thread::Thread ()
: myStack(new char[StackSize]), //...
```

- Obezbeđenje početnog konteksta je mnogo teži problem. Najvažnije je obezbediti trenutak "cepanja" steka: početak izvršavanja nove niti na njenom novokreiranom steku. Ova radnja se može izvršiti direktnim smeštanjem vrednosti u SP. Pri tom je veoma važno sledeće. Prvo, ta radnja se ne može obaviti unutar neke funkcije, jer se promenom vrednosti SP više iz te funkcije ne bi moglo vratiti. Zato je ova radnja u programu realizovana pomoću makroa (jednostavne tekstualne zamene), da bi ipak obezbedila lokalnost i fleksibilnost. Drugo, kod procesora i8086 SP se sastoji iz dva registra (SS i SP), pa se ova radnja vrši pomoću dve asemblerske instrukcije. Prilikom ove radnje vrednost koja se smešta u SP ne može biti automatski podatak, jer se on uzima sa steka čiji se položaj menja jer se menja i (jedan deo registra) SP. Zato su ove vrednosti statičke. Ovaj deo programa je ujedno i jedini mašinski zavisani deo Jezgra i izgleda ovako:

```
#define splitStack(p) \
static unsigned int sss, ssp; \ // FP_SEG() vraća segmentni, a FP_OFF() \
sss=FP_SEG(p); ssp=FP_OFF(p); \ // ofsetni deo pokazivača; \
asm { \ // neposredno ugrađivanje asemblerskih \
mov ss,sss; \ // instrukcija u kod; \
mov sp,ssp; \ \
mov bp,sp; \ // ovo nije neophodno; \
add bp,8 \ // ovo nije neophodno; \
}
```

- Početni kontekst nije lako obezbediti na mašinski nezavisani način. U ovoj realizaciji to je urađeno na sledeći način. Kada se kreira, nit se označi kao "započinjuća" atributom `isBeginning`. Kada dobije procesor unutar funkcije `resume()`, nit najpre ispituje da li započinje rad. Ako tek započinje rad (što se dešava samo pri prvom dobijanju procesora), poziva se globalna funkcija `wrapper()` koja predstavlja "omotač" korisničke niti:

```
void Thread::resume () {
if (isBeginning) {
isBeginning=0;
wrapper();
} else
longjmp(myContext,1);
}
```

- Prema tome, prvi poziv `resume()` i poziv `wrapper()` funkcije dešava se opet na steku prethodno tekuće niti, što ostavlja malo "đubre" na ovom steku, ali iznad granice zapamćene unutar `dispatch()`.
- Unutar statičke funkcije `wrapper()` vrši se konačno "cepanje" steka, odnosno prelazak na stek novokreirane niti:

```
void Thread::wrapper () {
void* p=runningThread->getStackPointer(); // vrati svoj SP
splitStack(p); // cepanje steka

unlock ();
runningThread->run(); // korisnička nit
lock ();

runningThread->markOver(); // nit je gotova,
runningThread = Scheduler::Instance()->get(); // predaje se procesor
runningThread->resume();
}
```

- Takođe je jako važno obratiti pažnju na to da ne sme da se izvrši povratak iz funkcije `wrapper()`, jer se unutar nje prešlo na novi stek, pa na steku ne postoji povratna adresa. Zbog toga se iz ove funkcije nikad i ne vraća, već se po završetku korisničke funkcije `run()` eksplicitno predaje procesor drugoj niti.
- Zbog ovakve logike, neophodno je da u sistemu uvek postoji bar jedna spremna nit. Uopšte, u sistemima se to najčešće rešava kreiranjem jednog "praznog", besposlenog (engl. *idle*) procesa, ili nekog procesa koji vodi računa o sistemskim resursima i koji se nikad ne može blokirati, pa je uvek u redu spremnih (tzv. demonski procesi, engl. *daemon process*). U ovoj realizaciji to će biti nit koja briše gotove niti, opisana u narednom odeljku.
- Na ovaj način, startovanje niti predstavlja samo njeno upisivanje u listu spremnih, posle označavanja kao "započinjuće":

```
void Thread::start () {
    //...
    fork();
}

void Thread::fork () {
    lock();
    Scheduler::Instance()->put(this);
    unlock();
}
```

Ukidanje niti

- Ukidanje niti je sledeći veći problem u konstrukciji Jezgra. Gledano sa strane korisnika, jedan mogući pristup je da se omogući eksplicitno ukidanje kreirane niti pomoću njenog destruktora. Pri tome se poziv destruktora opet izvršava u kontekstu onoga ko uništava nit. Za to vreme sama nit može da bude završena ili još uvek aktivna. Zbog toga je potrebno obezbediti odgovarajuću sinhronizaciju između ova dva procesa, što komplikuje realizaciju. Osim toga, ovakav pristup nosi i neke druge probleme, pa je on ovde odbačen, iako je opštiji i fleksibilniji.
- U ovoj realizaciji opredeljenje je da niti budu zapravo aktivni objekti, koji se eksplicitno kreiraju, a implicitno uništavaju. To znači da se nit kreira u kontekstu neke druge niti, a da zatim živi sve dok se ne završi funkcija `run()`. Tada se nit "sama" implicitno briše, tačnije njeno brisanje obezbeđuje Jezgro.
- Brisanje same niti ne sme da se izvrši unutar funkcije `wrapper()`, po završetku funkcije `run()`, jer bi to značilo "sečenje grane na kojoj se sedi": brisanje niti znači i dealokaciju steka na kome se izvršava sama funkcija `wrapper()`.
- Zbog ovoga je primenjen sledeći postupak: kada se nit završi, funkcija `wrapper()` samo označi nit kao "završenu" atributom `isOver`. Poseban aktivni objekat (nit) klase `ThreadCollector` vrši brisanje niti koje su označene kao završene. Ovaj objekat je nit kao i svaka druga, pa ona ne može doći do procesora sve dok se ne završi funkcija `wrapper()`, jer završni deo ove funkcije izvršava u sistemskom režimu.
- Klasa `ThreadCollector` je takođe *Singleton*. Kada se pokrene, svaka nit se "prijavi" u kolekciju ovog objekta, što je obezbeđeno unutar konstruktora klase `Thread`. Kada dobije procesor, ovaj aktivni objekat prolazi kroz svoju kolekciju i jednostavno briše sve niti koje su označene kao završene. Prema tome, ova klasa je zadužena tačno za brisanje niti:

```
void Thread::start () {
    ThreadCollector::Instance()->put(this);
    fork();
}

class ThreadCollector : public Thread {
public:

    static ThreadCollector* Instance ();

    void put (Thread*);
    int count ();

protected:

    virtual void run ();

private:

    ThreadCollector ();

    Collection rep;

    static ThreadCollector* instance;

};

void ThreadCollector::run () {
    while (1) {

        int i=0;
        CollectionIterator* it = rep.getIterator();

        for (i=0,it->reset(); !it->isDone(); it->next(),i++) {
            Thread* cur = (Thread*)it->currentItem();
            if (cur->isOver) {
                rep.remove(i);
                delete cur;
                it->reset(); i=0;
                dispatch();
            }
        }

        if (count()==1)
            longjmp(mainContext,1); // return to main

        dispatch();
    }
}
```

Pokretanje i gašenje programa

- Poslednji veći problem pri konstrukciji Jezgra jeste obezbeđenje ispravnog pokretanja programa i povratka iz programa. Problem povratka ne postoji kod ugrađenih (engl. *embedded*) sistema jer oni rade neprekidno i ne oslanjaju se na operativni sistem. U okruženju operativnog sistema kao što je PC DOS/Windows, ovaj problem treba rešiti jer je želja da se ovo Jezgro koristi za eksperimentisanje na PC računaru.

- Program se pokreće pozivom funkcije `main()` od strane operativnog sistema, na steku koji je odvojen od strane prevodioca i sistema. Ovaj stek nazivaćemo glavnim. Jezgro će unutar funkcije `main()` kreirati nit klase `ThreadCollector` (ugrađeni proces) i nit nad korisničkom funkcijom `userMain()`. Zatim će zapamtiti kontekst glavnog programa, kako bi po završetku svih korisničkih niti taj kontekst mogao da se povрати i program regularno završi:

```
void main () {

    ThreadCollector::Instance()->start();

    Thread::runningThread = new Thread(userMain);
    ThreadCollector::Instance()->put(Thread::runningThread);

    if (setjmp(mainContext)==0) {
        unlock();
        Thread::runningThread->resume();
    } else {
        ThreadCollector::destroy();
        return;
    }
}
```

- Treba još obezbediti "hvatanje" trenutka kada su sve korisničke niti završene. To najbolje može da uradi sam `ThreadCollector`: onog trenutka kada on sadrži samo jednu jedinu evidentiranu nit u sistemu (to je on sam), sve ostale niti su završene. (On evidentira sve aktivne niti, a ne samo spremne.) Tada treba izvršiti povratak na glavni kontekst:

```
void ThreadCollector::run () {
    //...
    if (count()==1)
        longjmp(mainContext,1); // return to main
    //...
}
```

Realizacija

- Zaglavlje `kernel.h` služi samo da uključi sva zaglavlja koja predstavljaju interfejs prema korisniku. Tako korisnik može jednostavno da uključi samo ovo zaglavlje u svoj kod da bi dobio deklaracije Jezgra.
- Prilikom prevođenja u bilo kom prevodiocu treba obratiti pažnju na sledeće opcije prevodioca:
 1. Funkcije deklarisanе kao *inline* moraju tako i da se prevode. U Borland C++ prevodiocu treba da bude isključena opcija `Options\Compiler\C++ options\Out-of-line inline functions`. Kritična je zapravo samo funkcija `Thread::setContext()`.
 2. Program ne sme biti preveden kao *overlay* aplikacija. U Borland C++ prevodiocu treba izabrati opciju `Options\Application\DOS Standard`.
 3. Memorijski model treba da bude takav da su svi pokazivači tipa *far*. U Borland C++ prevodiocu treba izabrati opciju `Options\Compiler\Code generation\Compact` ili `Large` ili `Huge`.
 4. Mora da bude isključena opcija provere ograničenja steka. U Borland C++ prevodiocu treba da bude isključena opcija `Options\Compiler\Entry/Exit code\Test stack overflow`.
- Sledi kompletan izvorni kod opisanog dela Jezgra.
- Datoteka `kernel.h`:

```
// Project:   Real-Time Programming
// Subject:   Multithreaded Kernel
// Module:    Kernel
// File:      kernel.h
// Created:   November 1996
// Revised:   August 2003
// Author:    Dragan Milicev
// Contents:  Kernel Interface

#include "thread.h"
#include "semaphor.h"
#include "msgque.h"
#include "timer.h"

inline void dispatch () { Thread::dispatch(); }

*      Datoteka krnl.h:

// Project:   Real-Time Programming
// Subject:   Multithreaded Kernel
// Module:    Kernel
// File:      krnl.h
// Created:   November 1996
// Revised:   August 2003
// Author:    Dragan Milicev
// Contents:  Kernel module interface
//           Helper functions:
//               lock
//               unlock

#ifndef _KRNL_
#define _KRNL_

#include <setjmp.h>

void lock ();    // Switch to kernel mode
void unlock (); // Switch to user mode

extern void userMain(); // User's main function

extern jmp_buf mainContext; // Context of the main thread

#endif

*      Datoteka thread.h:

// Project:   Real-Time Programming
// Subject:   Multithreaded Kernel
// Module:    Thread
// File:      thread.h
// Created:   November 1996
// Revised:   August 2003
// Author:    Dragan Milicev
// Contents:  Threading and context switching
//           Class: Thread

#ifndef _THREAD_
#define _THREAD_

#include "krnl.h"
#include "collect.h"
```

```

#include "recycle.h"

/////////////////////////////////////////////////////////////////
// class Thread
/////////////////////////////////////////////////////////////////

class Thread : public Object {
public:

    Thread ();
    Thread (void (*body) ());

    void start ();
    static void dispatch ();

    static Thread* running ();

    CollectionElement* getCEForScheduler ();
    CollectionElement* getCEForCollector ();
    CollectionElement* getCEForSemaphore ();

protected:

    virtual void run ();

    void markOver ();

    friend class ThreadCollector;
    virtual ~Thread ();

    friend class Semaphore;

    inline int setContext ();
    void resume ();
    char* getStackPointer () const;

    static void wrapper ();
    void fork();

private:

    void (*myBody) ();
    char* myStack;

    jmp_buf myContext;

    int isBeginning;
    int isOver;

    friend void main ();
    static Thread* runningThread;

    CollectionElement ceForScheduler;
    CollectionElement ceForCollector;
    CollectionElement ceForSemaphore;

    RECYCLE_DEC(Thread)

};

```

```
// WARNING: This function MUST be truely inline!
inline int Thread::setContext () {
    return setjmp(myContext);
}

inline void Thread::markOver () {
    isOver=1;
}

inline void Thread::run () {
    if (myBody!=0) myBody();
}

inline Thread* Thread::running () {
    return runningThread;
}

inline Thread::~~Thread () {
    delete [] myStack;
}

inline CollectionElement* Thread::getCEForScheduler () {
    return &ceForScheduler;
}

inline CollectionElement* Thread::getCEForCollector () {
    return &ceForCollector;
}

inline CollectionElement* Thread::getCEForSemaphore () {
    return &ceForSemaphore;
}

#endif
```

* **Datoteka schedul.h:**

```
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Scheduler
// File:     schedul.h
// Created:  November 1996
// Revised:  August 2003
// Author:   Dragan Milicev
// Contents:
//         Class: Scheduler

#ifndef _SCHEDUL_
#define _SCHEDUL_

////////////////////////////////////
```



```
// class Scheduler
////////////////////////////////////

class Thread;

class Scheduler {
public:

    static Scheduler* Instance ();

    virtual void    put (Thread*) = 0;
    virtual Thread* get () = 0;

protected:
    Scheduler () {}
};

#endif

*      Datoteka thrcol.h:

// Project:   Real-Time Programming
// Subject:   Multithreaded Kernel
// Module:    Thread Collector
// File:      thrcol.h
// Created:   November 1996
// Revised:   August 2003
// Author:    Dragan Milicev
// Contents:  Thread Collector responsible for thread deletion
//           Class:  ThreadCollector

#ifndef _THRCOL_
#define _THRCOL_

#include "collect.h"
#include "thread.h"

////////////////////////////////////
// class ThreadCollector
////////////////////////////////////

class ThreadCollector : public Thread {
public:

    static ThreadCollector* Instance ();

    void put (Thread*);
    int  count ();

protected:

    friend void main ();
    static void create ();
    static void destroy ();

    virtual void run ();

private:

    ThreadCollector () {}
};
```

```

~ThreadCollector () {}

Collection rep;

static ThreadCollector* instance;

};

inline void ThreadCollector::put (Thread* t) {
    if (t) rep.append(t->getCEForCollector());
}

inline int ThreadCollector::count () {
    return rep.size();
}

#endif

```

* **Datoteka kernel.cpp:**

```

// Project: Real-Time Programming
// Subject: Multithreaded Kernel
// Module: Kernel
// File: kernel.cpp
// Created: November 1996
// Revised: August 2003
// Author: Dragan Milicev
// Contents: Kernel main module
//           Helper functions: lock
//                               unlock
//           Functions:         main

#include "krnl.h"
#include "thread.h"
#include "schedul.h"
#include "thrcol.h"

/////////////////////////////////////////////////////////////////
// Helper functions lock () and unlock ()
/////////////////////////////////////////////////////////////////

void lock () {} // Switch to Kernel mode
void unlock () {} // Switch to User mode

/////////////////////////////////////////////////////////////////
// Function: main ()
/////////////////////////////////////////////////////////////////

jmp_buf mainContext; // Context of the main thread

void main () {
    ThreadCollector::create();
    ThreadCollector::Instance()->start();
}

```

```

Thread::runningThread = new Thread(userMain);
ThreadCollector::Instance()->put(Thread::running());

if (setjmp(mainContext)==0) {
    unlock();
    Thread::running()->resume();
} else {
    ThreadCollector::destroy();
    return;
}
}

*      Datoteka thread.cpp:

// Project:   Real-Time Programming
// Subject:   Multithreaded Kernel
// Module:    Thread
// File:      thread.cpp
// Created:   November 1996
// Revised:   August 2003
// Author:    Dragan Milicev
// Contents:  Threading and context switching
//           Class: Thread

#include <dos.h>
#include "thread.h"
#include "thrcol.h"
#include "schedul.h"

/////////////////////////////////////////////////////////////////
// class Thread
/////////////////////////////////////////////////////////////////

const int StackSize = 4096;

RECYCLE_DEF(Thread);

Thread::Thread ()
: RECYCLE_CON(Thread), myBody(0), myStack(new char[StackSize]),
  isBeginning(1), isOver(0),
  ceForScheduler(this), ceForCollector(this), ceForSemaphore(this) {}

Thread::Thread (void (*body)())
: RECYCLE_CON(Thread), myBody(body), myStack(new char[StackSize]),
  isBeginning(1), isOver(0),
  ceForScheduler(this), ceForCollector(this), ceForSemaphore(this) {}

void Thread::resume () {
    if (isBeginning) {
        isBeginning=0;
        wrapper();
    } else
        longjmp(myContext,1);
}

Thread* Thread::runningThread = 0;

```

```

void Thread::start () {
    ThreadCollector::Instance()->put(this);
    fork();
}

void Thread::dispatch () {
    lock ();
    if (runningThread && runningThread->setContext()==0) {

        Scheduler::Instance()->put(runningThread);
        runningThread = (Thread*)Scheduler::Instance()->get();
        // Context switch:
        runningThread->resume();

    } else {
        unlock ();
        return;
    }
}

////////////////////////////////////
// Warning: Hardware/OS Dependent!
////////////////////////////////////

char* Thread::getStackPointer () const {
    // WARNING: Hardware\OS dependent!
    // PC Stack grows downwards:
    return myStack+StackSize-10;
}

// Borland C++: Compact, Large, or Huge memory Model needed!
#ifdef __TINY__ || defined(__SMALL__) || defined(__MEDIUM__)
    #error Compact, Large, or Huge memory model needed
#endif

#define splitStack(p) \
    static unsigned int sss, ssp; \
    sss=FP_SEG(p); ssp=FP_OFF(p); \
    asm { \
        mov ss,sss; \
        mov sp,ssp; \
        mov bp,sp; \
        add bp,8 \
    }

////////////////////////////////////
// Enf of Dependencies
////////////////////////////////////

void Thread::fork () {
    lock();
    Scheduler::Instance()->put(this);
    unlock();
}

```

```

}

void Thread::wrapper () {
    void* p=runningThread->getStackPointer();
    splitStack(p);

    unlock ();
    runningThread->run();
    lock ();

    runningThread->markOver();
    runningThread=(Thread*) Scheduler::Instance()->get();
    runningThread->resume();
}

*      Datoteka schedul.cpp:

// Project:   Real-Time Programming
// Subject:   Multithreaded Kernel
// Module:    Scheduler
// File:      schedul.cpp
// Created:   November 1996
// Revised:   August 2003
// Author:    Dragan Milicev
// Contents:
//           Classes: Scheduler
//                   RoundRobinScheduler

#define _RoundRobinScheduler

#include "schedul.h"
#include "queue.h"
#include "thread.h"

////////////////////////////////////
// class RoundRobinScheduler
////////////////////////////////////

class RoundRobinScheduler : public Scheduler {
public:

    virtual void    put (Thread* t) { if (t) rep.put(t->getCEForScheduler()); }
}

    virtual Thread* get ()          { return (Thread*)rep.get(); }

private:
    Queue rep;
};

////////////////////////////////////
// class Scheduler
////////////////////////////////////

Scheduler* Scheduler::Instance () {
    #ifdef _RoundRobinScheduler
        static RoundRobinScheduler instance;
    #endif

```

```

    return &instance;
}

*      Datoteka thrcol.cpp:
// Project:  Real-Time Programming
// Subject:  Multithreaded Kernel
// Module:   Thread Collector
// File:     thrcol.cpp
// Created:  November 1996
// Revised:  August 2003
// Author:   Dragan Milicev
// Contents: Thread Collector responsible for thread deletion
//          Class:  ThreadCollector

#include "thrcol.h"

////////////////////////////////////
// class ThreadCollector
////////////////////////////////////

ThreadCollector* ThreadCollector::instance = 0;

ThreadCollector* ThreadCollector::Instance () {
    if (instance==0) create();
    return instance;
}

void ThreadCollector::create () {
    instance = new ThreadCollector;
}

void ThreadCollector::destroy () {
    delete instance;
}

void ThreadCollector::run () {
    while (1) {

        int i=0;
        CollectionIterator* it = rep.getIterator();

        for (i=0,it->reset(); !it->isDone(); it->next(),i++) {
            Thread* cur = (Thread*)it->currentItem();
            if (cur->isOver) {
                rep.remove(i);
                delete cur;
                it->reset(); i=0;
                dispatch();
            }
        }

        if (count()==1)
            longjmp(mainContext,1); // return to main

        dispatch();
    }
}

```

```

    }
}

```

Zadaci

3.1.

Komentarisati kako se propagira izuzetak podignut u nekoj niti konkurentnog programa realizovanog u školskom Jezgru. Šta se dešava ukoliko se izuzetak ne obradi u korisničkom kodu jedne niti? Rešiti ovaj problem odgovarajućom modifikacijom Jezgra.

Rešenje

```

void Thread::wrapper () {
    void* p=runningThread->getStackPointer();//dohvati svoj SP
    splitStack(p); // cepanje steka
    unlock ();

    try{
        runningThread->run(); // korisnička nit
    } catch(...){
        // obrada izuzetka
    }

    lock ();

    // predaja procesora drugoj niti
}

```

3.2. Obrada izuzetaka

Komentarisati da li i kako bi se mogla realizovati sledeća semantika propagacije izuzetaka u programu realizovanom pomoću datog školskog jezgra: ukoliko izuzetak nije obrađen u datoj niti, on se propagira u nit koja je pokrenula tu nit ("roditeljska" nit).

Rešenje:

U klasu `Thread` potrebno je dodati, kao atribut, pokazivač na roditeljsku nit, koji se inicijalizuje pri kreiranju ili najkasnije pokretanju date niti i obezbediti implementaciju operacije `handleChildException()` kojom će roditeljska nit obrađivati neobrađene izuzetke.

```

// thread.h

class Thread : public Object {
private:
    Thread* parentThread;

protected:
    virtual void handleChildException(std::exception e) {}
    ...
};

// thread.cpp

#include <thread.h>

```

```

void Thread::start() {
    ThreadCollector::Instance()->put(this);

    parent = Thread::runningThread;
    fork();
}

void Thread::wrapper () {

    void* p=runningThread->getStackPointer();

    splitStack(p);

    unlock ();

    try {
        runningThread->run();
    }
    catch (std::exception e) {
        runningThread->parent->handleChildException(e);
    }

    lock ();

    runningThread->markOver();
    runningThread=(Thread*) Scheduler::Instance()->get();
    runningThread->resume();
}

```

Međutim, ovo je samo naizgled odgovarajuće rešenje. Problem je što se obrada izuzetka obavlja u kontekstu (na steku) iste niti koja je bacila izuzetak (i koji je razmotan do dna). Mehanizam koji bi „dobacio“ izuzetak roditeljskoj niti, tako da se njeno izvršavanje prekine, a onda izuzetak obradi u njenom kontekstu (i na njenom steku koji se razmotava), kao da je izuzetak nastao u njoj, zahteva drugačiji pristup. Ovde su ukratko opisani elementi tog pristupa, a čitaocu se prepušta implementacija:

- Izuzetak uhvaćen u funkciji `wrapper` kao što je pokazano, treba sačuvati u objektu klase `Thread` koji predstavlja roditeljsku nit.
- Na svim mestima na kojima Jezgro dobija kontrolu, odnosno u trenucima kada se kontekst izvršavanja prebacuje na neki nit, treba proveriti da li ta nit ima ovako postavljen izuzetak. U tom slučaju, odmah po povratku u funkciju kojom se prešlo u Jezgro i kojom je ta nit izgubila procesor, treba baciti ovaj izuzetak ugrađenim mehanizmom jezika C++.

Na ovaj način, roditeljska nit „vidi“ izuzetak ka „svoj“, bačen iz Jezgra, ali u ovoj izvedbi isključivo sinhrono (iz neke funkcije Jezgra koju je pozvala sama korisnička nit). Ta nit potom obrađuje ovaj izuzetak na standardan način.

Zadaci za samostalan rad

3.3

U prikazanoj realizaciji postoji problem ukoliko kreator niti ima referencu (pokazivač) na tu nit, jer se po završetku niti ona implicitno uništava, pa referenca ostaje viseća (engl. *dangling reference*). Rešiti ovaj problem tako da se:

- može ispitati da li je izvršavanje niti gotovo ili ne, pozivom operacije `isDone()`;
- nit ne uništava implicitno, već eksplicitno, pozivom operacije `destroy()`, ali samo pod uslovom da je njeno izvršavanje gotovo.

3.4

Potrebno je da klasa `x` ima operaciju `do()` čija implementacija treba da ima sledeću semantiku:

```
void X::do(Y* y, int i) {  
    int j = this->helper(i);  
    y->do(j);  
}
```

Korišćenjem školskog jezgra realizovati klasu `x`, ali tako da se svaki poziv operacije `X::doSomething()` izvršava u sopstvenom kontekstu, a ne u kontekstu pozivaoca.

3.5

Korišćenjem niti iz školskog jezgra, skicirati strukturu programa koji kontroliše parking za vozila. Parking ima jedan ulaz i jedan izlaz, kao i znak da na parkingu više nema slobodnih mesta.

Sinhronizacija i komunikacija pomoću deljene promenljive

Međusobno isključenje i uslovna sinhronizacija

Međusobno isključenje

- Deljena promenljiva je koncept međuprocene komunikacije jednostavan za upotrebu, jer procesi mogu pristupati deljenoj promenljivoj na isti način kao i svojoj lokalnoj promenljivoj, jednostavnim čitanjem i upisom. Uprkos tome, njeno korišćenje od strane uporednih procesa bez potrebne sinhronizacije, odnosno bez odgovarajućih ograničenja u redosledu izvršavanja delova uporednih procesa koji joj pristupaju, može da dovede do potpuno nepredvidivih (pa time i nekorektnih) rezultata.
- Na primer, neka dva procesa pristupaju celobrojnoj deljenoj promenljivoj x tako što uporedo izvršavaju sledeću jednostavnu naredbu:

```
x := x+1
```

Na mnogim procesorima ova naredba biće prevedena u sekvencu tri mašinske instrukcije:

- učitaj vrednost x iz memorije (sa lokacije promenljive x) u neki registar procesora
- uvećaj vrednost tog registra za jedan
- upiši vrednost tog registra u memoriju (u lokaciju promenljive x).

Čak i ako je svaka pojedinačna instrukcija *nedeljiva* (engl. *indivisible*), tj. *atomična* (engl. *atomic*), jer atomičnost čitanja ili upisa skalarne promenljive iz memorije, odnosno u memoriju obezbeđuje hardver, cela grupa instrukcija ne izvršava se nedeljivo, odnosno izolovano. Zbog toga mogu da se dogode različita preplitanja njihovog izvršavanja u kontekstu dva uporedna procesa koja mogu da proizvedu različite rezultate: ili zbog asinhronne promene konteksta (kao posledica spoljašnjeg prekida) pri multiprogramiranju na jednom procesoru, ili paralelnim izvršavanjem na više procesora. Na primer, ukoliko je prethodna vrednost x bila 0, konačna vrednost za x može biti i 1 (tako što oba procesa, jedan po jedan, najpre učitaju u registar vrednost 0, a zatim oba, opet jedan po jedan, upišu u memoriju vrednost 1) ili 2, što odgovara rezultatu u kom se ove dve sekvence izvršavaju jedna po jedna, bez međusobnog preplitanja, tzv. *serijalizovano* (engl. *serializable*). Osim u slučaju kada je nepredvidivo ponašanje baš željeno (što je izuzetno retko), konkurentan program koji ima nepredvidivo ponašanje nije dobar, jer logička ispravnost njegovog rezultata zavisi od nepredvidivih parametara platforme (multiprogramiranje ili multiprocesiranje, sinhrona ili asinhrona promena konteksta, algoritam raspoređivanja).

- Posmatrajmo još jedan primer. Jedan proces izračunava koordinate za navođenje nekog automatski navođenog vozila, dok drugi proces očitava izračunate koordinate i upravlja tim vozilom. Ove dve obrade odvojene su u procese recimo sa ciljem paralelizacije na više procesora: one se tada mogu vršiti istovremeno i time popraviti performanse. Neka deljena struktura podataka koja čuva izračunate koordinate izgleda ovako:

```
type Coord = record {  
  x : integer;  
  y : integer;  
};  
  
var sharedCoord : Coord;
```

Procesi izgledaju ovako:

```

process Controller
var nextCoord : Coord;
begin
  loop
    computeCoord(nextCoord);
    sharedCoord := nextCoord;
  end;
end;

process Driver
begin
  loop
    moveTo(sharedCoord);
  end;
end;

```

Procedura `computeCoord` obavlja proračun koordinata tokom kog nema interakcije sa drugim procesom (lokalna, privatna obrada). Interakcija nastaje kada proces `Controller` upisuje u deljenu promenljivu. Proces `Driver` čita iz deljene promenljive (implicitan pristup vrednosti `sharedCoord` kod definisanja stvarnog argumenta poziva procedure `moveTo`), a potom obavlja svoju privatnu obradu u proceduri `moveTo`, tokom koje opet nema interakcije između procesa. Kako se može očekivati da se operacija upisa podataka u strukturu (naredba `sharedCoord:=nextCoord`) implementira pomoću (bar) dve mašinske instrukcije koje nisu nedeljive, može se dogoditi da proces `Driver` koji upravlja vozilom, iako ga proces `Controller` ispravno navodi, pročita nekorektne vrednosti koordinata (samo delimično promenjenu, a delimično staru), zbog sledećeg preplitanja izvršenih instrukcija dva procesa:

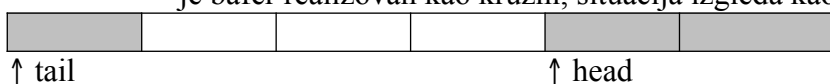
Proces Controller upisuje u sharedCoord:	Vrednost u sharedCoord:	Proces Driver čita iz sharedCoord:
	0,0	
x:=1	1,0	
y:=1	1,1	
	1,1	x=1
	1,1	y=1
x:=2	2,1	
y:=2	2,2	
	2,2	x=2
	2,2	y=2
x:=3	3,2	
	3,2	x=3
	3,2	y=2
y:=3	3,3	

- Prema tome, neki delovi koda procesa koji pristupaju deljenim promenljivim moraju da se izvršavaju nedeljivo (atomično) u odnosu na druge takve delove drugih procesa. U suprotnom, mogu da izazovu konflikte, odnosno nepredvidive rezultate koji se razlikuju od onih koji bi postojali kada bi se ovi delovi izvršavali serijalizovano (jedan po jedan, izolovano).
- Deo koda (sekvenca naredbi) procesa koji se mora izvršavati nedeljivo (engl. *indivisible*), odnosno izolovano u odnosu na druge takve delove koda drugih procesa naziva se *kritična sekcija* (engl. *critical section*).
- Sinhronizacija koja je neophodna da bi se obezbedila atomičnost izvršavanja kritičnih sekcija naziva se *međusobno isključenje* (engl. *mutual exclusion*).

- Međusobno isključenje je tipično potrebno kada uporedni procesi pristupaju (na čitanje i upis) složenim deljenim promenljivim ili objektima koji imaju složeno stanje, kao u navedenim primerima.
- Pretpostavlja se da je barem nekakva atomičnost, recimo atomičnost operacije čitanja ili upisa vrednosti skalarne promenljive obezbeđena na nivou pristupa memoriji. Na primer, ukoliko jedan proces izvršava naredbu $x:=1$ a drugi naredbu $x:=2$, onda će vrednost x biti ili 1 ili 2, a ne neka druga vrednost. Naravno, ukoliko se vrši upis u nescikalnu promenljivu, atomičnost u opštem slučaju nije obezbeđena, osim na nivou jedne mašinske reči.
- Problem međusobnog isključenja je prvi opisao E. Dijsktra 1965. godine. Ovaj problem je od izuzetnog teorijskog i praktičnog značaja za konkurentno programiranje.

Uslovna sinhronizacija

- Međusobno isključenje nije jedina vrsta sinhronizacije od interesa. Druga je *uslovna sinhronizacija* (engl. *condition synchronization*). Uslovna sinhronizacija je potrebna kada jedan proces želi da izvrši neku akciju, a koja ima smisla ili je ispravna samo ako je neki uslov ispunjen od strane drugog procesa, recimo tako što je neki drugi proces već izvršio neku svoju akciju ili se taj proces ili deljena promenljiva nalazi u nekom definisanom stanju, odnosno zadovoljava neki uslov.
- Najpoznatiji, školski primer je problem *ograničenog bafera* (engl. *bounded buffer*). Dva procesa razmenjuju podatke preko bafera koji je ograničenog kapaciteta. Prvi proces "proizvodi" podatke i upisuje ih u bafer; on se naziva *proizvođačem* (engl. *producer*). Drugi proces uzima (čita) podatke iz bafera i konzumira ih na neki način; on se naziva *potrošačem* (engl. *consumer*).
- Ovakva indirektna komunikacija između procesa obezbeđuje njihovo nezavisnije izvršavanje koje dozvoljava male fluktuacije u brzini kojom proizvode, odnosno konzumiraju podatke. Na primer, ukoliko je u nekom periodu potrošač nešto sporiji, proizvođač može puniti bafer proizvedenim podacima, ne čekajući na to da potrošač bude spreman da ih preuzima istom brzinom; slično, u nekom drugom periodu potrošač može biti nešto brži i trošiti zalihu podataka iz bafera.
- Ovakav sistem razmene informacija između dva procesa preko bafera se često naziva još i sistem *proizvođač-potrošač* (engl. *producer-consumer*).
- Ukoliko je bafer ograničenog kapaciteta, što je često slučaj zbog ograničenosti resursa u sistemu (čak i ako je bafer logički neograničen, memorijski prostor u koji se on smešta je uvek fizički ograničen), onda je potrebna sledeća uslovna sinhronizacija između procesa:
 - proizvođač ne sme da stavi podatak u bafer ukoliko je bafer pun;
 - potrošač ne može da uzme podatak iz bafera ukoliko je bafer prazan;
 - ukoliko su moguće simultane operacije stavljanja i uzimanja podatka, onda se mora obezbediti i međusobno isključenje, kako ne bi došlo do korupcije internih struktura koje pamte poziciju "prvog" i "poslednjeg" stavljenog elementa; ukoliko je bafer realizovan kao kružni, situacija izgleda kao na sledećoj slici:



Uposleno čekanje

- Jednostavan i očigledan način za implementaciju sinhronizacije, korišćenjem samo tehnika klasičnog, sekvencionalnog programiranja, jeste taj da procesi postavljaju i proveravaju

indikatore (logičke promenljive, „zastavice“, engl. *flags*), odnosno u opštijem slučaju izračunavaju i ispituju logičke izraze koji za svoje operande imaju deljene promenljive. Indikator, kao deljena promenljiva logičkog tipa, ili u opštijem slučaju izraz takvog tipa, označava to da li neki proces može da nastavi izvršavanje dalje od neke tačke ili ne, odnosno da li je odgovarajući uslov nastavka izvršavanja zadovoljen; taj proces ispituje indikator ili izračunava izraz u petlji, iznova čitajući vrednost deljene promenljive i izračunavajući vrednost izraza, sve dok ona postane odgovarajuća zbog promene vrednosti deljenih promenljivih od strane drugog procesa.

- Na primer, za jednostavnu uslovnu sinhronizaciju kod koje jedan proces treba da nastavi izvršavanje od neke tačke samo ako je neki uslov ispunjen, a neki drugi proces signalizira ispunjenje tog uslova, rešenje je jednostavno: proces koji signalizira ispunjenje uslova postavlja indikator – deljenu promenljivu; proces koji čeka da uslov bude ispunjen, proverava indikator u petlji:

```
shared var flag : boolean = false;

process P1;  (* Waiting process *)
begin
    ...
    while flag = false do
        null
    end;
    ...
end P1;

process P2;  (* Signalling process *)
begin
    ...
    flag := true;
    ...
end P2;
```

- Za primer ograničenog bafera (bez međusobnog isključenja), skica rešenja na jeziku C++ izgleda ovako:

```
const int N = ...;  // Capacity of the buffer
class Data;

class BoundedBuffer {
public:

    BoundedBuffer ();

    void  append (Data*);
    Data* take ();

private:
    Data* buffer[N];
    int head, tail;
    int count;
};

BoundedBuffer::BoundedBuffer () : head(0), tail(0), count(0) {}

void BoundedBuffer::append (Data* d) {
    while (count==N); // Wait while the buffer is full
    buffer[tail] = d;
    tail = (tail+1)%N;
    count++; // Signal that the buffer is not empty
```

```

}

Data* BoundedBuffer::take () {
    while (count==0); // Wait while the buffer is empty
    Data* d = buffer[head];
    head = (head+1)%N;
    count--; // Signal that the buffer is not full
    return d;
}

```

- Ovakva realizacija sinhronizacije, u kojoj proces koji čeka na ispunjenje uslova izvršava petlju sve dok taj uslov ne bude ispunjen, iznova čitajući vrednost jedne ili više deljenih promenljivih i izračunavajući vrednost logičkog izraza koji predstavlja uslov nastavka izvršavanja, naziva se *uposlano čekanje* (engl. *busy waiting*) ili *vrtenje* (engl. *spinning*).
- Algoritam za uslovnu sinhronizaciju uposlenim čekanjem je relativno jednostavan. Međutim, međusobno isključenje nije jednostavno sasvim korektno realizovati uposlenim čekanjem.
- Jedan (neispravan) pristup rešavanju međusobnog isključenja pomoću uposlenog čekanja je sledeći: proces najavljuje svoju želju da uđe u kritičnu sekciju, a potom ispituje da li je drugi proces uradio to isto, u kom slučaju čeka da drugi proces izađe iz kritične sekcije:

```
shared var flag1, flag2 : boolean = false;
```

```

process P1
begin
    loop
        flag1 := true;          (* Announce intent to enter *)
        while flag2 = true do    (* Busy wait if the other process is in *)
            null
        end;
        <critical section>      (* Critical section *)
        flag1 := false;         (* Exit protocol *)
        <non-critical section>
    end
end P1;

process P2
begin
    loop
        flag2 := true;          (* Announce intent to enter *)
        while flag1 = true do    (* Busy wait if the other process is in *)
            null
        end;
        <critical section>      (* Critical section *)
        flag2 := false;         (* Exit protocol *)
        <non-critical section>
    end
end P2;

```

Ovo rešenje ima problem jer se može desiti sledeći scenario: jedan proces najavi svoj ulazak u kritičnu sekciju postavljajući svoj indikator, onda to uradi i drugi proces, a zatim oba procesa zauvek ostaju u petljama čekajući na obaranje indikatora onog drugog procesa. Dakle, nijedan proces neće moći da uđe u kritičnu sekciju, iako se oba procesa izvršavaju. Ovakvo neispravno stanje sistema naziva se *živo blokiranje* (engl. *livelock*).

- Drugi pristup može da bude promena redosleda najave ulaska u sekciju i postavljanja indikatora:

```

process P1
begin

```

```

loop
  while flag2 = true do          (* Busy wait if the other process is in *)
    null
  end;
  flag1 := true;
  <critical section>             (* Critical section *)
  flag1 := false;               (* Exit protocol *)
  <non-critical section>
end
end P1;

process P2
begin
  loop
    while flag1 = true do        (* Busy wait if the other process is in *)
      null
    end;
    flag2 := true;
    <critical section>           (* Critical section *)
    flag2 := false;             (* Exit protocol *)
    <non-critical section>
  end
end P2;

```

Ovakvo rešenje ne obezbeđuje međusobno isključenje, jer se može dogoditi sledeći scenario: oba indikatora su `false`, oba procesa ispituju tuđ indikator i pronalaze da je on `false`, pa zatim oba postavljaju svoj indikator i ulaze u kritičnu sekciju. Problem je u tome što se operacije ispitivanja tuđeg indikatora i postavljanja svog ne obavljaju atomično. Ovakav problem naziva se *utrkivanje* (engl. *race condition*): zbog nepostojanja odgovarajuće sinhronizacije, procesi se „pretiču“, odnosno neki proces se „zatrčava“ i izvršava neke svoje akcije iako to ne bi smeo da uradi, jer za to nije ispunjen odgovarajući uslov.

- Sledeće rešenje uvodi promenljivu `turn` koja ukazuje na koga je došao red da uđe u sekciju:

```

shared var turn : integer = 1;

process P1
begin
  loop
    while turn = 2 do
      null
    end;
    <critical section>
    turn := 2;
    <non-critical section>
  end
end P1;

process P2
begin
  loop
    while turn = 1 do
      null
    end;
    <critical section>
    turn := 1;
    <non-critical section>
  end
end P2;

```

Ovo rešenje obezbeđuje međusobno isključenje i nema problem živog blokiranja (izvesti dokaze!), ali ima problem zato što nasilno uvodi nepotreban naizmeničan redosled izvršavanja procesa, jer se procesi ovde uvek naizmenično smenjuju u kritičnoj sekciji. To je neprihvatljivo u opštem slučaju konkurentnih procesa: ako neki proces uopšte ne želi da uđe u svoju kritičnu sekciju, nije dobro i nema nikakve potrebe sprečavati neki drugi proces da to uradi, kada to svakako neće izazvati bilo kakve probleme. Prema tome, ovakvo rešenje nepotrebno ograničava konkurentnost uvodeći suvišnu sinhronizaciju.

- Konačno, sledeće rešenje, Petersonov algoritam, nema ovaj problem nasilnog uslovljavanja redosleda procesa, kao ni živog blokiranja, a obezbeđuje međusobno isključenje (Peterson, 1981). Ukoliko oba procesa žele da uđu u kritičnu sekciju, onda dozvolu dobija onaj na kome je red (definisan promenljivom `turn`, koja tada ima neku od dve vrednosti, bilo koju, makar i neodređeno koju od te dve); ukoliko samo jedan proces želi da uđe u sekciju, on to može da uradi, bez obzira na vrednost promenljive `turn`, jer indikator koji ukazuje na to da onaj drugi proces želi da uđe u kritičnu sekciju nije postavljen:

```
process P1
begin
  loop
    flag1 := true;
    turn := 2;
    while flag2 = true and turn = 2 do
      null
    end;
    <critical section>
    flag1 := false;
    <non-critical section>
  end
end P1;
```

```
process P2
begin
  loop
    flag2 := true;
    turn := 1;
    while flag1 = true and turn = 1 do
      null
    end;
    <critical section>
    flag2 := false;
    <non-critical section>
  end
end P2;
```

- Postoji i opštije rešenje za proizvoljan broj procesa n , ali je ono komplikovanije.
- Nedostatak ovog algoritma je u tome što procesi moraju da „znaju jedan za drugog“, odnosno što njihov kod zavisi od onog drugog i deljene promenljive koju on postavlja: proces `P1` koristi deljenu promenljivu `flag2` kojom `P2` najavljuje nameru da uđe u kritičnu sekciju, i obratno. Mora se unapred znati i koliko ovakvih procesa za koje treba obezbediti međusobno isključenje ima. U praksi često postoje opštiji slučajevi u kojima je potrebno obezbediti međusobno isključenje kritične sekcije nekog procesa nezavisno od toga koji još procesi postoje sa kojima treba obezbediti međusobno isključenje (npr. izvršavanje iste procedure isključivo, u kontekstu neodređenog broja procesa koji je pozivaju).
- Osnovni problem algoritama uposlenog čekanja je njihova neefikasnost: proces troši procesorsko vreme na nekoristan rad, čekajući u petlji da uslov bude ispunjen, stalno čitajući vrednosti deljenih promenljivih i izračunavajući uslov, što može biti neodređeno

dugo. Za to vreme, procesor bi mogao da radi neki drugi, koristan rad (izvršava instrukcije nekog drugog procesa). Čak i na multiprocesorskim sistemima sa deljenom memorijom, intenzivan i repetitivan pristup deljenim promenljivim koji ovi algoritmi podrazumevaju mogu da uzrokuju intenzivan saobraćaj na memorijskoj magistrali, odnosno povećane konflikte na magistrali koja povezuje procesore i zajedničku memoriju, a koju u jednom trenutku može koristiti samo jedan procesor (takvo međusobno isključenje obezbeđuju hardverski uređaji i protokoli), pa se procesori onda nepotrebno zaustavljaju u izvršavanju svojih instrukcija koje možda i ne pristupaju deljenim promenljivim.

- Zbog svega navedenog se uposlano čekanje veoma retko koristi u realnim sistemima.

Sinhronizacija korišćenjem hardverske podrške

- Ukoliko se konkurentni procesi izvršavaju na jednom procesoru, multiprogramiranjem, kada jedan proces uđe u kritičnu sekciju, jedini način da neki drugi proces takođe uđe u tu kritičnu sekciju jeste taj da se, pre nego što prvi proces iz kritične sekcije izađe, dogodi promena konteksta i procesor pređe sa izvršavanja instrukcija procesa koji je već u kritičnoj sekciji na instrukcije procesa koji u kritičnu sekciju potom ulazi. Ovo se može dogoditi samo na jedan od sledećih načina:
 - Sinhrono, ukoliko proces koji je ušao u kritičnu sekciju izvrši neki sistemski poziv, odnosno samoinicijativno pređe u izvršavanje koda koji vrši promenu konteksta.
 - Sinhrono, ukoliko proces koji se izvršava izazove neki hardverski izuzetak, pa se dogodi isto što i u narednom slučaju.
 - Asinhrono, ukoliko stigne spoljašnji zahtev za prekid i procesor prekine izvršavanje tekuće kontrole toka i pređe na obradu prekida, u kojoj se izvrši promena konteksta i kontrola iz prekidne rutine vrati drugom procesu.
- Ukoliko kod kritične sekcije ne sadrži nikakav sistemski poziv, niti neka njegova instrukcija izaziva izuzetak, što se može obezbediti pod određenim uslovima, jedina preostala mogućnost jeste pojava asinhronog, spoljašnjeg prekida. Zbog toga se međusobno isključenje u ovakvom okruženju može obezbediti *maskiranjem* (zabranom) spoljašnjih prekida – mehanizmom kojim hardver procesora prelazi u režim u kom ignoriše spoljašnje prekide.
- Prema tome, na ulasku u kritičnu sekciju potrebno je maskirati (zabraniti) prekide, a po izlasku iz nje ih demaskirati (ponovo dozvoliti), recimo nekom procedurom koja se prevodi u odgovarajuće procesorske instrukcije koje to rade:

```
...
disable_interrupts();
<critical section>
enable_interrupts();
...
```

- Međutim, ovakav pristup nije pogodan za korišćenje u aplikacijama, tj. korisničkim procesima na višem nivou apstrakcije, iz sledećih razloga:
 - Dok su prekidi maskirani računar ne reaguje na spoljašnje događaje, pa postaje „neosetljiv“ na njih: odziv na takve događaje se odlaže dok prekidi ponovo budu omogućeni i softver na njih reaguje, što može biti neprihvatljivo dugo. Nema nikakve garancije da kod korisničkih procesa ne drži prekide neograničeno dugo (recimo zbog izvršavanja beskonačne petlje), ili prosto neprihvatljivo dugo (zbog neke duge obrade ili čak čekanja na neki uslov).

- Zlonamerno ili zbog slučajne greške u programu, proces može da maskira prekide, a onda ih nikada ne demaskira (izostane poziv procedure za omogućavanje prekida). Isto važi i za slučaj kada proces otkáže usred izvršavanja kritične sekcije.
- Osim toga, ovo rešenje obezbeđuje samo to da jedan procesor „ne uskoči sam sebi“ u kritičnu sekciju, ali ne sprečava da se to dogodi izvršavanjem na više procesora: kada jedan proces, čije instrukcije izvršava jedan procesor, uđe u kritičnu sekciju, ništa ne sprečava to da i drugi procesor izvrši instrukcije drugog procesa koje pripadaju njegovoj kritičnoj sekciji.
- Za međusobno isključenje pri multiprocesiranju potrebna je posebna hardverska podrška koju procesori obezbeđuju u obliku posebnih procesorskih instrukcija. Jedna od njih je instrukcija tipa *test and set* („testiraj i postavi“). Ova instrukcija ispituje (i vraća kao rezultat) vrednost sadržaja neke memorijske lokacije (u kojoj je neka deljena promenljiva, indikator, smeštena u zajedničku operativnu memoriju), a potom tu istu vrednost postavlja na 1. Pritom su ove dve akcije čitanja i upisa garantovano atomične, odnosno nedeljive: ovu atomičnost obezbeđuje hardver, tipično tako što procesor izvršava ciklus čitanja memorijske lokacije i naredni ciklus upisa u istu memorijsku lokaciju preko magistrale računara držeći sve vreme tu magistralu „zaključanu“, tako da nijedan drugi procesor ne može da je koristi za sve to vreme, pa time ni da pristupa memoriji dok se oba ciklusa ne završe. Drugim rečima, hardverski uređaj i protokol obezbeđuju atomičnost dva ciklusa na magistrali, međusobnim isključenjem pristupa različitih procesora istoj deljenoj magistrali (pa time i memoriji) – atomičnost na višem nivou apstrakcije (u softveru) uvek se dobija korišćenjem raspoložive atomičnosti na nižem nivou apstrakcije (ovde u hardveru).
- Ako je opisana instrukcija „upakovana“ u odgovarajuću funkciju, onda međusobno isključenje izgleda ovako:

```
int lock = 0;
...
while test_and_set(lock);
    <critical section>
lock = 0;
...
```

- Svakoj kritičnoj sekciji, tačnije deljenom objektu ili deljenoj strukturi podataka za koju se želi obezbediti međusobno isključenje pristupa, pridruži se jedna celobrojna (zapravo logička) deljena promenljiva (ovde `lock`) inicijalizovana na 0. Vrednost 0 ove promenljive označava da kritična sekcija nije „zaključana“, odnosno da pristup deljenoj strukturi nije zatvoren. Instrukcija tipa *test and set* atomično čita i vraća vrednost ove promenljive, a ujedno je i postavlja na 1. Ako je kritična sekcija bila zaključana, ova instrukcija neće promeniti njenu vrednost (bila je i ostaje 1). Ako je kritična sekcija otvorena, ova instrukcija će je zaključati, a vratiti 0, i to uraditi atomično (tako da nijedan drugi procesor ne može istovremeno da uradi to isto). Sve dok ova instrukcija ne vrati 0, proces mora da čeka, i to radi u petlji *while* – petlji uposlenog čekanja.
- Sličan mehanizam obezbeđuje procesorska instrukcija *swap* koja atomično zamenjuje vrednost *lock* bita u memoriji i operanda instrukcije. Preostala logika je potpuno ista kao prethodna.
- Naravno, opisani algoritam je ponovo uposlono čekanje (tzv. *spin lock*) i ima iste opisane nedostatke. Osim toga, ukoliko proces koji je zaključao kritičnu sekciju u njoj ostane predugo, ili je greškom ili zbog otkaza nikada ne otključa, sekcija ostaje trajno zatvorena.
- Zbog navedenih problema, ni ove tehnike se ne koriste u aplikacijama, tj. u procesima na višem nivou apstrakcije, ali se koriste za implementaciju kritičnih sekcija operativnih sistema. U tom slučaju, navedenih problema nema, jer je:
 - kod operativnog sistema pisan bez zle namere i bez poznatih grešaka (pouzđano);

- kod kritičnih sekcija uvek ograničenog, i to kratkog trajanja, pošto su kritične sekcije vrlo ograničene (tipično nekoliko instrukcija do nekoliko desetina ili stotina procesorskih instrukcija); čekanje je stoga ograničeno tipično na vreme dok drugi procesor ne izađe iz kritične sekcije u koju je ušao;
- operativni sistem ili otporan na određene otkaze, ili, ukoliko nije, ti otkazi postaju fatalni za ceo računar.
- U navedenom slučaju, ove dve tehnike se koriste zajedno: maskiranje prekida obezbeđuje isključenje procesora „od samog sebe“, a „vrtenje“ (*spin lock*) za međusobno isključenje više procesora. Za svaku deljenu strukturu podataka i njen „ključ“ `lck`, i svaku kritičnu sekciju koja joj pristupa, međusobno isključenje se obezbeđuje ovako:

```
void lock (int& lck) {
    disable_interrupts();
    while test_and_set(lck);
}

void unlock (int& lck) {
    lck = 0;
    enable_interrupts();
}

// Mutual exclusion:
...
lock(lck);
<critical section>
unlock(lck);
...
```

Semafori

- Semafori predstavljaju jednostavan i efikasan koncept za programiranje međusobnog isključenja i uslovne sinhronizacije, široko rasprostranjen i dostupan u praktično svim operativnim sistemima i u mnogim jezicima i bibliotekama za konkurentno programiranje. Predložio ih je Dijkstra 1968. godine.
- Semafor je deljeni objekat sa internim stanjem predstavljenim jednom celobrojnomo nenegativnom promenljivom, nad kojim se, osim inicijalizacije, mogu vršiti još samo sledeće dve operacije:
 - `wait(S)`: (Dijkstra je originalno zvao P) Ako je vrednost semafora `s` veća od nule, ta vrednost se umanjuje za jedan i proces koji je izvršio ovu operaciju nastavlja svoje izvršavanje; u suprotnom, proces koji je izvršio ovu operaciju mora da čeka sve dok vrednost semafora ne postane veća od nule, a tada se ta vrednost takođe umanjuje za jedan.
 - `signal(S)`: (Dijkstra je originalno zvao V) Vrednost semafora se uvećava za jedan.
- Važno je uočiti da su operacije `wait` i `signal` atomične (nedeljive), pri čemu je ta atomičnost definisana i garantovana semantikom semafora, a obezbeđuje je implementacija semafora (npr. operativni sistem koji nudi koncept semafora i operacije nad njima kao sistemске pozive), pa programer ne mora o tome da brine niti da zbog toga preduzima posebne radnje. Prema tome, dva procesa koja uporedo izvršavaju neku od ovih operacija međusobno ne interaguju tokom izvršavanja ovih operacija, pa na semaforu (tačnije, njegovoj celobrojnoj vrednosti) nema konflikata (utrivanja).

Implementacija

- Kada je vrednost semafora nula, proces koji je izvršio operaciju `wait()` treba da čeka da neki drugi proces izvrši operaciju `signal()`, kako bi vrednost semafora postala veća od nule i proces koji je izvršio `wait` mogao da nastavi izvršavanje. Iako je ovo čekanje moguće implementirati uposlenim čekanjem, takva implementacija je, kao što je objašnjeno, neefikasna i neprimenjiva za slučaj proizvoljno mnogo procesa koji se mogu sinhronizovati pomoću semafora. Umesto toga se ne samo implementacija čekanja na semaforu, nego na praktično svim drugim sinhronizacionim primitivama, oslanja na neki vid *suspenzije* (engl. *suspension*) procesa – zaustavljanja izvršavanja procesa dok se ne ispuni uslov nastavka izvršavanja, tako što proces neće ni dobiti procesor do tada. Suspenzija se ponekad naziva i *blokiranje* (engl. *blocking*).
- Ovaj pristup sastoji se u sledećem: kada proces izvršava operaciju `wait()`, kontrolu preuzima kod koji pripada operativnom sistemu ili izvršnom okruženju (u zavisnosti od toga gde je implementirana sinhronizaciona primitiva). Ako proces treba suspendovati, onda izvršno okruženje/operativni sistem ne smešta strukturu sa kontekstom tog procesa (tj. njegov PCB) u listu (skup, red) spremnih procesa (engl. *ready queue*), već u posebnu listu pridruženu svakom semaforu (lista procesa suspendovanih na semaforu). Na taj način suspendovani proces ne može dobiti procesor sve dok ga sistem ponovo ne vrati u listu spremnih, pa zbog toga on ne troši procesorsko vreme dok čeka, kao kod uposlenog čekanja.
- Implementacija semafora zato može da bude sledeća: semafor sadrži red procesa koji čekaju na semaforu i jednu celobrojnu promenljivu `val` koja ima sledeće značenje:
 - 1) `val > 0`: još `val` procesa može da izvrši operaciju `wait` a da se ne blokira, a na semaforu nema blokiranih procesa;
 - 2) `val = 0`: na semaforu nema blokiranih procesa, ali će se proces koji naredni izvrši `wait` blokirati;
 - 3) `val < 0`: ima `-val` blokiranih procesa, a svaki poziv `wait` izaziva blokiranje pozivajućeg procesa.

- o Algoritam operacije `wait` je tada sledeći:

```
procedure wait(S)
  val:=val-1;
  if val<0 then
    begin
      suspend the running process by putting it into the suspended queue of S
      take another process from the ready queue and switch the context to it
    end
  end;
end;
```

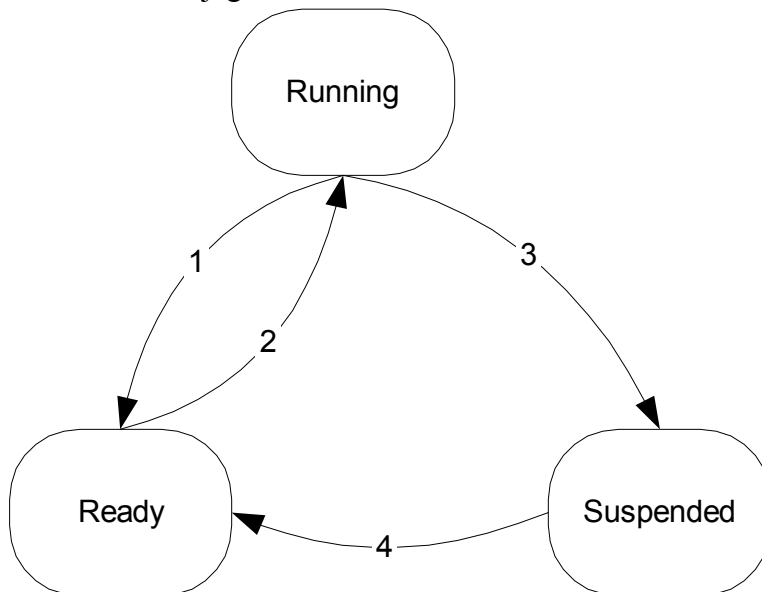
- o Algoritam operacije `signal` je sledeći:

```
procedure signal(S)
  val:=val+1;
  if val<=0 then
    begin
      take one process from the suspended queue of S
      and deblock it by putting it into the ready queue
    end
  end;
end;
```

- Treba primetiti da ovi algoritmi ne definišu redosled po kome se procesi ređaju u listi suspendovanih (ili *blokiranih*, engl. *blocked*) na semaforu. Obično implementacije podrazumevaju FIFO redosled, mada programer treba da smatra da je taj redosled nepoznat, pri čemu se može (i mora) podrazumevati da je taj algoritam *pravedan* (engl. *fair*), što znači da neće dozvoliti da neki proces neograničeno dugo čeka jer ga na

semaforu stalno pretiču neki drugi procesi koji se suspenduju nakon njega, a deblokiraju pre njega. Ovakav problem neograničenog čekanja naziva se *izgladnjivanje* (engl. *starvation*). FIFO redosled jeste (trivijalno) pravedan, ali nije jedini takav.

- Prema tome, osnovna stanja kroz koje proces prolazi tokom svog života prikazana su na sledećem dijagramu:



gde prelazi označavaju sledeće situacije:

1. Proces gubi procesor i prelazi u stanje *Ready*, ali ne zbog suspenzije, nego zato što je to sam eksplicitno tražio nekim sistemskim pozivom (npr. operacijom `dispatch()` ili neblokirajućom operacijom na semaforu), ili se u sistemu pojavio novi spreman proces koji treba da preuzme procesor, ukoliko je sistem sa *preuzimanjem* (engl. *preemptive*), pa do promene konteksta dolazi i asinhrono (kao posledica spoljašnjeg prekida).
 2. Izabrani proces iz liste spremnih dobija procesor prilikom promene konteksta.
 3. Proces gubi procesor i postaje suspendovan zato što se blokira na sinhronizacionoj primitivi (npr. semaforu, po izvršavanju operacije `wait()`).
 4. Proces prelazi iz stanja suspenzije u listu spremnih, jer je tekući (engl. *running*) proces izvršio operaciju `signal()` na semaforu.
- Nedeljivost operacija `wait` i `signal`, kao i drugih sličnih atomičnih primitiva, izvršno okruženje ili operativni sistem obezbeđuje korišćenjem hardverske podrške, kako je to ranije opisano (maskiranje prekida i *spin lock*). Treba primetiti da u ovom slučaju jedan proces izvršava uposlono čekanje na *lock* bitu. Međutim, to uposlono čekanje je veoma kratko i traje samo dok jedan proces ne završi operaciju `wait` ili `signal` na semaforu, što je ne samo ograničenog, nego i sasvim predvidivog i vrlo kratkog trajanja. Atomičnost se ne može dobiti "ni iz čega", već se ipak mora nekako podržati na nižem nivou apstrakcije, a u ovom slučaju na hardverskom nivou.
 - U zavisnosti od implementacije, granularnost kritične sekcije, odnosno zaključavanja ovim mehanizmom može da bude jedan od sledećih:
 - Ceo kod jezgra (kernela) operativnog sistema može da bude jedna jedinstvena kritična sekcija, odnosno svaki ulazak u kod implementacije sistemskog poziva (npr. operacije nad semaforom) zaključava celu tu sekciju i nijedan drugi proces ne može da u nju uđe (izvrši bilo šta drugo u jezgru) dok izvršavanje ne izađe iz koda kernela i kontrola se vrati kodu korisničkog procesa. U tom slučaju promena konteksta nije moguća tokom izvršavanja koda kernela, pa se kernel naziva kernelom „bez preotimanja“ (engl. *non preemptive*).

- Kritične sekcije su finije granularnosti, i formiraju se oko operacija nad pojedinačnim deljenim strukturama podataka unutar kernela. U tom slučaju promena konteksta je moguća i dok se izvršavaju neki delovi kernela (engl. *preemptive kernel*), pa je sistem reaktivniji (sa bržim odzivom, engl. *responsive*). Ovo, međutim, otežava programiranje kernela i unosi potencijalne rizike od žive i mrtve blokade koji se moraju sprečiti dizajnom kernela.

Međusobno isključenje i uslovna sinhronizacija pomoću semafora

- Međusobno isključenje je jednostavno obezbediti pomoću semafora koji, kao deljeni objekat, koriste uporedni procesi koji se međusobno isključuju:

```
var mutex : Semaphore = 1; // Initially equal to 1
```

```
process P1;
  loop
    wait(mutex);
    <critical section>
    signal(mutex);
    <non-critical section>
  end
end P1;
```

```
process P2;
  loop
    wait(mutex);
    <critical section>
    signal(mutex);
    <non-critical section>
  end
end P2;
```

- Treba primetiti da je kod koji okružuje kritičnu sekciju jednostavan i uvek isti, bez obzira na to koliko procesa treba ovako sinhronizovati. Pritom, skup procesa koji se međusobno isključuju ne mora da bude unapred poznat niti fiksni.
- Atomičnost izvršavanja operacija na semaforu obezbeđuje da će samo jedan od više procesa koji žele da uđu u kritičnu sekciju prvi uspeti da prođe operaciju *wait* bez suspenzije, jer nailazi na vrednost semafora jednaku 1. Taj proces će tom operacijom spustiti vrednost semafora na 0, pa će svi ostali procesi biti suspendovani pre ulaska u svoje kritične sekcije, sve dok ovaj proces ne izađe iz nje i podigne vrednost semafora na 1. Tada će samo jedan od procesa koji čekaju na tom semaforu ući u kritičnu sekciju. Ako takvih nema, vrednost semafora ostaje 1, pa ulaz u kritičnu sekciju ostaje otvoren.
- Inicijalna vrednost semafora predstavlja zapravo maksimalan broj procesa koji može istovremeno ući u kritičnu sekciju, pa u opštem slučaju može biti i veći od 1.
- Primer međusobnog isključenja korišćenjem semafora iza POSIX API:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

// Initialization:

const char* mutexName = "/myprogram_mutex"
sem_t* mutex = sem_open(mutexName, O_CREAT, 1);
// O_CREAT|O_EXCL creates an exclusive (non-shared, private) semaphore

...
```

```
// Use for mutual exclusion:
sem_wait(mutex);
// Critical section
sem_post(mutex); // signal
...
// Release the semaphore when it is no longer needed:
sem_close(mutex);
```

Uporedni procesi koji treba da dele isti semafor identifikuju taj semafor simboličkim imenom koje je proizvoljan niz znakova koji počinje znakom /. Da bi koristio semafor, proces mora da ga, kao i svaki drugi resurs koji dobija od operativnog sistema, „otvori“ pozivom funkcije `sem_open`. Bit postavljen maskom `O_CREAT` u drugom argumentu modifikuje semantiku ovog poziva tako da prvi proces koji izvrši ovu operaciju kreira semafor u sistemu, a svaki drugi ga samo „otvara“ (u suprotnom, ovaj poziv bi vratio grešku ako semafor već ne postoji). Poslednji argument ove operacije zadaje inicijalnu vrednost semafora. Operacije `sem_wait` i `sem_post` (*signal*) identifikuju semafor deskriptorom tipa `sem_t` koji je vratila operacija otvaranja. Na kraju svaki proces mora osloboditi semafor operacijom zatvaranja `sem_close`.

- Uslovnu sinhronizaciju je takođe jednostavno obezbediti semaforima:

```
var sync : Semaphore(0); // Initially 0

process P1; // Waiting process
...
wait(sync);
...
end P1;

process P2; // Signalling process
...
signal(sync);
...
end P2;
```

- Primer realizacije ograničenog bafera pomoću semafora:

```
const int N = ...; // Capacity of the buffer
class Data;

class BoundedBuffer {
public:

    BoundedBuffer ();

    void append (Data*);
    Data* take ();

private:
    Semaphore mutex;
    Semaphore spaceAvailable, itemAvailable;

    Data* buffer[N];
    int head, tail;
};

BoundedBuffer::BoundedBuffer () :
    mutex(1), spaceAvailable(N), itemAvailable(0),
    head(0), tail(0) {}

void BoundedBuffer::append (Data* d) {
    spaceAvailable.wait();
```

```
mutex.wait();
buffer[tail] = d;
tail = (tail+1)%N;
mutex.signal();
itemAvailable.signal();
}

Data* BoundedBuffer::take () {
    itemAvailable.wait();
    mutex.wait();
    Data* d = buffer[head];
    head = (head+1)%N;
    mutex.signal();
    spaceAvailable.signal();
    return d;
}

class Producer : public Thread {
public:
    Producer (BoundedBuffer* bb) : myBuffer(bb) {...}

protected:
    virtual void run ();

    Data* produce(); // Produce an item

private:
    BoundedBuffer* myBuffer;
};

void Producer::run () {
    while (1) {
        Data* d = produce();
        myBuffer->append(d);
    }
}

class Consumer : public Thread {
public:
    Consumer (BoundedBuffer* bb) : myBuffer(bb) {...}

protected:
    virtual void run ();

    void consume(Data*); // Consume an item

private:
    BoundedBuffer* myBuffer;
};

void Consumer::run () {
    while (1) {
        Data* d = myBuffer->take();
        consume(d);
    }
}
```


- Treba primetiti sledeći detalj: iako su u oba slučaja sinhronizacije, i za međusobno isključenje i za uslovnu sinhronizaciju, operacije *wait* i *signal* nad istim semaforom uparene, u prvom slučaju obe te operacije izvršavaju se u kontekstu istog procesa (proces koji zaključava kritičnu sekciju je i otključava), odnosno različitih procesa (jedan proces čeka na uslov, a drugi signalizira njegovo ispunjenje).
- Potencijalni problem koji može da se pojavi pri nekorektnoj upotrebi semafora je *mrtvo* ili *kružno blokiranje* (engl. *deadlock*): stanje sistema u kome je nekoliko procesa suspendovano (blokirano) međusobnim uslovljavanjem. Na primer, u ovom primeru oba procesa mogu uporedo izvršiti svoju prvu *wait* operaciju, a onda se međusobno „zaglaviti“ na drugom semaforu:

```
process P1;
  wait(S1);
  wait(S2);
  ...
  signal(S2);
  signal(S1);
end P1;
```

```
process P2;
  wait(S2);
  wait(S1);
  ...
  signal(S1);
  signal(S2);
end P2;
```

- Iako su semafori jednostavan i efikasan koncept, oni nisu sasvim pogodni za programiranje složenih sistema. Samo jedna greška u uparivanju operacija *wait* i *signal* kod međusobnog isključenja ili uslovne sinhronizacije dovodi do potpuno nekorektnog ponašanja programa: ukoliko se greškom izostavi *wait* ili njemu odgovarajući *signal*, sinhronizacija će ili izostati (utrkiivanje), ili će kritična sekcija ostati trajno zaključana. Osim toga, kao što je pokazano, čak i ako su ove operacije propisno uparene, njihov redosled može biti bitan, ukoliko se kritične sekcije ugnežđuju, jer može doći do mrtve blokade.
- Semafori su važni iz istorijskih razloga i zbog toga što predstavljaju jednostavnu, elementarnu sinhronizacionu primitivu, pomoću koje je moguće realizovati mnoge druge. Zato su oni dostupni u praktično svim operativnim sistemima i u mnogim bibliotekama.
- Međutim, semafori su koncept suviše niskog nivoa apstrakcije. Način poziva operacija *wait* i *signal* nikako ne sugerise svrhu upotrebe semafora (za pristup deljenom objektu ili za uslovnu sinhronizaciju), niti je na bilo koji način ograničena njihova upotreba u određenim delovima koda, niti se garantuje njihova uparenost. Tako ove operacije mogu lako postati „raštrkane“ po kodu konkurentnog programa, u različitim procedurama čak i istog procesa. Zato takav kod postaje težak za razumevanje, proveru i održavanje, a i vrlo lako podložan greškama opisanog tipa. Zato se nijedan složeniji sistem ne zasniva isključivo na korišćenju semafora, nego su potrebni strukturirani konstrukti višeg nivoa apstrakcije.

Binarni i drugi semafori

- Opisani semafor se naziva *brojačkim* (engl. *counting*) ili *n-arnim*. Za mnoge primene (npr. za međusobno isključenje ili jednostavnu uslovnu sinhronizaciju) dovoljno je da semafor ima najveću vrednost 1, pa je dovoljna samo binarna vrednost semafora (0 ili 1). Takvi semafori nazivaju se *binarnim* (engl. *binary*).

- Kod binarnih semafora, semantika operacija je neznatno izmenjena. Operacija `wait` suspenduje proces, ukoliko vrednost binarnog semafora 0, a postavlja vrednost semafora na 0 (umesto dekrementiranja), ako je njegova vrednost bila 1. Operacija `signal` deblokira proces koji je suspendovan, ako ga ima, odnosno postavlja vrednost događaja na 1, ako suspendovanog procesa nema (umesto inkrementiranja).
- Mnoga okruženja (operativni sistemi ili biblioteke) nude različite varijante binarnih semafora, uz odgovarajuću semantiku i eventualna ograničenja u upotrebi, pa njihova implementacija može biti jednostavnija i efikasnija u određenim slučajevima kada takva ograničenja postoje.
- Na primer, koncept zvan *mutex* je binaran semafor namenjen isključivo za upotrebu kod međusobnog isključenja, pa je prirodno i često ograničenje da je njegova podrazumevana inicijalna vrednost 1, kao i da samo proces koji je „zaključao“ ovaj semafor operacijom `wait` može da izvrši operaciju `signal` na njemu (poziv te operacije iz konteksta drugog procesa smatra se greškom).
- Drugi primer je koncept *događaja* (engl. *event*), u smislu da binarna vrednost semafora označava da se neki događaj ili desio, ili nije desio, ili *uslova* (engl. *condition*) – uslov je ispunjen ili nije. Ovi koncepti služe za uslovnu sinhronizaciju, pa su u skladu sa tim primerena i neka ograničenja. Na primer, na događaj po pravilu čeka samo jedan proces, pa je semantika događaja nedefinisana ako postoji više suspendovanih procesa. Zato se u nekim sistemima događaj proglašava kao vlasništvo nekog procesa, i jedino taj proces može izvršiti operaciju `wait`, dok operaciju `signal` može vršiti bilo koji drugi proces.
- U mnogim sistemima postoje složene operacije čekanja na više binarnih ili čak brojačkih sema, po kriterijumu "i" i "ili".
- Postoje i varijante operacija `wait` i `signal` *brojačkih semafora* u kojima se parametrom zadaje inkrement odnosno dekrement vrednosti semafora (može biti veći od 1).

Monitori

- Jedan koncept višeg nivoa apstrakcije od semafora, namenjen za sinhronizaciju i komunikaciju po modelu deljenog objekta, koji je odavno osmišljen, ali i dalje izuzetno prisutan u popularnim programskim jezicima jeste koncept *monitora* (engl. *monitor*).
- Monitor je:
 - Apstraktni tip podataka (u terminologiji OO programiranja – klasa) koji grupiše *podatke* i *operacije* nad tim podacima (procedure). Instance monitora su deljeni objekti kojim mogu da pristupaju uporedni procesi pozivajući operacije monitora.
 - Monitor podrazumevano enkapsulira svoje podatke, a u interfejsu ima samo operacije (procedure) dostupne spolja.
 - Procedure se podrazumevano izvršavaju međusobno isključeno (serijalizovano), pa nije potrebna nikakva eksplicitna sinhronizacija u tom cilju o kojoj bi programer morao da brine. Drugim rečima, međusobno isključenje procedura je implicitno garantovano semantikom monitora: ako neki proces trenutno izvršava neku proceduru monitora, garantovano je da nijedan drugi proces ne izvršava istovremeno bilo koju proceduru monitora.
- Na razvoju koncepta monitora radili su mnogi autori, a najznačajniji radovi su sledećih autora: Dijkstra (1968), Brinch-Hansen (1973) i Hoare (1974). Monitori u svom izvornom obliku postoje u jezicima Modula 1, Concurrent Pascal i Mesa. Naprednije varijante monitora postoje i u jezicima Ada i Java.

- Osnovne navedene (prve dve) karakteristike monitora su inherentno objektno orijentisane, pa je to razlog zbog kog je koncept monitora, iako veoma star i zastupljen u mnogim zamrlim jezicima, i danas u širokoj upotrebi u živim jezicima.
- Podrazumevano međusobno isključenje, zajedno sa enkapsulacijom, značajno olakšava programiranje procedura monitora jer garantuje izolovanost pristupa deljenim podacima unutar monitora, pa je utrkivanje sprečeno.
- Primer monitora koji realizuje ograničeni bafer (jezik Concurrent Pascal):

```
monitor buffer;
  export append, take;

  var ... (* Declaration of necessary variables *)

  procedure append (i : integer);
    ...
  end;

  procedure take (var i : integer);
    ...
  end;

begin
  ... (* Initialization of monitor variables *)
end;
```

Uslovna sinhronizacija u monitoru

- Iako monitor implicitno obezbeđuje međusobno isključenje svojih procedura, potrebna je i uslovna sinhronizacija (npr. kod ograničenog bafera). Mnogi autori su predlagali različite varijante koncepata za uslovnu sinhronizaciju unutar monitora, a različiti jezici su podržavali neke od njih.
- Kod monitora koje je predložio Hoare (1974), sinhronizaciona primitiva za uslovnu sinhronizaciju unutar monitora se naziva *uslovna promenljiva* (engl. *condition variable*). Uslovna promenljiva je član monitora. Nad njom se mogu vršiti dve operacije sa sledećom semantikom:
 - *wait*: proces koji je izvršio *wait* se (bezuslovno) suspenduje (blokira) i smešta u red čekanja pridružen ovoj uslovnoj promenljivoj; proces potom oslobađa svoj ekskluzivni pristup do monitora i time dozvoljava da drugi proces uđe u neku proceduru monitora;
 - *signal*: kada neki proces izvrši ovu operaciju, sa reda blokiranih procesa na ovoj uslovnoj promenljivoj oslobađa se (deblokira) jedan proces, ako takvog ima; ako takvog procesa nema, onda operacija *signal* nema nikakvog efekta.
- Prirodno i posledično, operacije *wait* i *signal*, kao operacije nad uslovnom promenljivom kao privatnim (enkapsuliranim) podatkom monitora, mogu da se izvrše samo unutar procedura monitora, i samo u kontekstu procesa koji izvršava neku operaciju monitora.
- Primetiti razlike semantike semafora i uslovnih promenljivih: semafor ima svoje stanje predstavljeno celobrojnem promenljivom, dok uslovna promenljiva to nema; operacija *wait* na semaforu uslovno blokira proces, dok ga ista operacija na uslovnoj promenljivoj bezuslovno blokira (zato se ova operacija po pravilu izvršava kao posledica provere nekog uslova u proceduri monitora); operacija *signal* na semaforu inkrementira vrednost semafora čak i ako blokiranih procesa nema, dok ista operacija na uslovnoj promenljivoj tada nema efekta.
- Primer ograničenog bafera sa uslovnom sinhronizacijom:

```

monitor buffer;
  export append, take;

  var
    buf : array[0..size-1] of integer;
    head, tail : 0..size-1;
    numberInBuffer : integer;
    spaceAvailable, itemAvailable : condition;

  procedure append (i : integer);
  begin
    while numberInBuffer = size do
      wait(spaceAvailable);
    end;
    buf[tail] := i;
    tail := (tail+1) mod size;
    numberInBuffer := numberInBuffer+1;
    signal(itemAvailable);
  end;

  procedure take (var i : integer);
  begin
    while numberInBuffer = 0 do
      wait(itemAvailable);
    end;
    i := buf[head];
    head := (head+1) mod size;
    numberInBuffer := numberInBuffer-1;
    signal(spaceAvailable);
  end;

begin (* Initialization *)
  numberInBuffer := 0;
  head := 0; tail := 0
end;

```

- Postavlja se pitanje šta se dešava kada se operacijom `signal` deblokira neki proces: tada postoje dva procesa koja konkurišu za pristup monitoru, onaj koji je izvršio `signal` i onaj koji je deblokiran, pri čemu ne smeju oba nastaviti izvršavanje, zbog garantovanog međusobnog isključenja? Postoje različite varijante definisane semantike operacije `signal` koje ovo rešavaju:
 - Operacija `signal` je dozvoljena samo ako je poslednja akcija procesa pre napuštanja monitora (kao u primeru ograničenog bafera), pa problema zapravo i nema. Međutim, ovo je prilično restriktivna opcija.
 - Operacija `signal` ima sporedni efekat izlaska procesa iz procedure monitora (implicitni `return`), pa proces koji izvrši `signal` implicitno napušta monitor. I ovo može da bude suviše restriktivno, slično prethodnom.
 - Operacija `signal` koja deblokira drugi proces implicitno blokira proces koji je izvršio `signal`, tako da on može da nastavi izvršavanje tek kada monitor postane slobodan, odnosno kad deblokirani proces napusti svoju proceduru monitora (čije izvršavanje nastavlja iza `wait` na kom je bio blokiran), ali svakako pre bilo kog drugog procesa koji tek čeka da uđe u proceduru monitora.
 - Operacija `signal` koja deblokira drugi proces ne blokira proces koji je izvršio `signal`, ali deblokirani proces može da nastavi izvršavanje tek kada proces koji je izvršio `signal` napusti monitor (a opet svakako pre bilo kog drugog procesa koji tek čeka da uđe u monitor). Ova varijanta je potencijalno osetljivija na greške nego prethodna iz sledećeg razloga: proces koji je izvršio `signal`, po pravilu zato što je

stanje varijabli monitora postavio tako da je određeni uslov, na koga je čekao deblokirani proces, ispunjen, nastavlja svoje izvršavanje procedure monitora; ako se iza operacije `signal` nalaze još neke akcije, one mogu slučajno promeniti stanje varijabli monitora i dovesti do toga da posmatrani uslov više nije ispunjen. Proces koji je čekao na taj uslov, nakon deblokade, tako nastavlja izvršavanje sa ponovo narušenim uslovom, što nije dobro.

- Ukoliko se želi nezavisna logika monitora od navedene semantike u trećem ili četvrtom slučaju, operacija `wait` može da se ugradi u petlju tipa `while`, umesto u naredbu `if`, kako je to urađeno u primeru ograničenog bafera (iako u tom primeru ne postoji opisani problem, jer iza `signal` nema više naredbi, pa je bila dovoljna i naredba `if`). Tada će uslov uvek ponovo biti proveren kada se proces koji je na njemu čekao deblokira. Treba primetiti i to da između provere uslova i poziva operacije `wait` uslov ne može da se izmeni, jer ne može nastati utrkanje, pošto je proces koji izvršava bilo koji deo procedure monitora uvek jedini takav.

Problemi vezani za monitore

- Jedan od osnovnih problema vezanih za koncept monitora jeste pitanje kako razrešiti situaciju kada se proces koji je napravio ugnežđeni poziv operacije drugog monitora iz operacije jednog monitora suspenduje na uslovnoj promenljivoj ili sličnoj primitivi unutar tog drugog monitora? Zbog semantike `wait` operacije, pristup drugom monitoru biće oslobođen, ali neće biti oslobođen pristup monitoru iz čije procedure je napravljen ugnežđeni poziv. Tako će procesi koji pokušavaju da uđu u taj prvi monitor biti blokirani dugo, što smanjuje konkurentnost.
- Najčešći pristup ovom problemu jeste taj da se spoljašnji monitori drže zaključanim (Java, POSIX, Mesa), odnosno da se ovaj problem ignoriše i njegovo rešavanje prepusti programeru. Drugi pristup je da se potpuno zabrani ugnežđivanje poziva operacija monitora (Modula-1), što je previše restriktivno. Treći pristup je da se obezbede konstrukti kojima bi se definisalo koji monitori se oslobađaju u slučaju blokiranja na uslovnoj promenljivoj u ugnežđenom pozivu.
- Iako su monitori dobar koncept visokog nivoa apstrakcije, koji uspešno obezbeđuje enkapsulaciju i međusobno isključenje, uslovna sinhronizacija se i dalje obavlja primitivama relativno niskog nivoa apstrakcije.

Zaštićeni objekti u jeziku Ada

- Jezik Ada podržava koncept monitora pojmom *zaštićenog objekta* (engl. *protected object*).
- Zaštićeni objekat obezbeđuje grupisanje podataka i potprograma koji nad tim podacima operišu, kao kod klasičnog monitora. Interfejs zaštićenog objekta specificira koji su potprogrami dostupni spolja, dok je implementacija (uključujući i podatke) enkapsulirana. Potprogrami zaštićenog objekta nazivaju se *zaštićenim procedurama i funkcijama*.
- Procedure i funkcije u jeziku Ada se razlikuju po tome što funkcije mogu vraćati vrednost, dok procedure ne mogu. Obe vrste potprograma mogu imati ulazne, izlazne i ulazno-izlazne parametre.
- Zaštićene procedure mogu da čitaju ili upisuju vrednosti u zaštićene podatke objekta, s tim da se to radi međusobno isključivo, tako da najviše jedan proces može u datom trenutku izvršavati zaštićenu proceduru. Međusobno isključenje je opet obezbeđeno semantikom jezika, pa programer ne treba da obezbeđuje ovu sinhronizaciju eksplicitno.

- Sa druge strane, zaštićene funkcije dozvoljavaju samo čitanje zaštićenih podataka. Dozvoljava se da više procesa uporedo izvršava zaštićene funkcije, jer to uporedo izvršavanje ne pravi konflikte (jer ne menja vrednosti podataka).
- Obezbeđeno je takođe da se zaštićena funkcija izvršava samo ako se ne izvršava nijedna zaštićena procedura istog objekta, i obratno: pozivi zaštićenih procedura i funkcija su međusobno isključivi. Ovakav protokol naziva se *više čitalaca-jedan pisac* (engl. *multiple readers-single writer*).
- Zaštićeni objekat može da se definiše ili kao pojedinačna instanca, ili kao tip koji se može instancirati u vreme izvršavanja, slično kao što važi i za procese u jeziku Ada.
- Primer:

```
protected type SharedCoord (initX, initY : Real) is
  function read (x, y : out Real) return Boolean;
  procedure write (x, y : in Real);
private
  myX : Real := initX;
  myY : Real := initY;
end SharedCoord;

protected body SharedCoord is
  function read (x, y : out Real) return Boolean is
  begin
    x := myX; y := myY;
    return True;
  end read;

  procedure write (x, y : in Real) is
  begin
    myX := x; myY := y;
  end write;
end SharedCoord;
```

- Umesto uslovnih promenljivih, za uslovnu sinhronizaciju koriste se tzv. *čuvvari* (engl. *guards*): logički izrazi koji se deklarativno pridružuju procedurama monitora, a izvršavanje procedure dopušta pozivajućem procesu samo ako je uslov definisan tim izrazom ispunjen; u suprotnom, pozivajući proces se suspenduje dok uslov ne bude ispunjen.
- Čuvvari se u jeziku Ada nazivaju *barijerama* (engl. *barrier*), a zaštićene procedure kojima su pridružene barijere nazivaju se *ulazi* (engl. *entry*).
- Ako je rezultat barijere *False* kada proces poziva neki ulaz, pozivajući proces se suspenduje sve dok barijera ne promeni vrednost i dok drugi procesi ne napuste sve funkcije i procedure zaštićenog objekta.
- Primer ograničenog bafera:

```
bufferSize : constant Integer := ...;
type Index is mod bufferSize;
subtype Count is Natural range 0..bufferSize;
type Buffer is array (Index) of DataItem;

protected type BoundedBuffer is
  entry append (item : in DataItem);
  entry take (item : out DataItem);
private
  head : Index := 0;
  tail : Index := 0;
  numberInBuffer : Count := 0;
  buffer : Buffer;
end BoundedBuffer;
```

```

protected body BoundedBuffer is
  entry append (item : in DataItem) when numberInBuffer /= bufferSize is
  begin
    buf(tail) := item;
    tail := tail + 1;
    numberInBuffer := numberInBuffer + 1;
  end append;

  entry take (item : out DataItem) when numberInBuffer /= 0 is
  begin
    item := buf(head);
    head := head + 1;
    numberInBuffer := numberInBuffer - 1;
  end take;

end BoundedBuffer;

```

- Barijere se izračunavaju kada:
 1. proces poziva neki zaštićeni ulaz i pridružena barijera referencira neku promenljivu ili podatak objekta koji se možda promenio od kada je barijera poslednji put izračunavana;
 2. proces napušta zaštićenu proceduru ili ulaz, a postoje procesi koji čekaju na ulazima čije barijere referenciraju promenljive ili podatke objekta koji su se možda promenili od kada je barijera poslednji put izračunavana.

Treba primetiti da se barijera ne izračunava kada neki proces napušta zaštićenu funkciju, jer ona ne može da promeni vrednost promenljivih.

- Uslovna sinhronizacija barijerama, odnosno čuvarima, olakšava programiranje uslovne sinhronizacije: dovoljno je da se deklarativno specifikuje preduslov za ispravno funkcionisanje procedure monitora, a izvršno okruženje jezika obezbeđuje odgovarajuću sinhronizaciju, uz ponovno izračunavanje uslova samo kada je potrebno. Naravno, ovakav pristup može da uzrokuje veliki režijski trošak za vreme izvršavanja, zbog nepotrebnih ponovljenih izračunavanja barijera.

Sinhronizovane operacije u jeziku Java

- Jezik Java podržava koncept monitora na sledeći način. Svaka klasa u jeziku Java je implicitno, bilo direktno ili indirektno, izvedena iz ugrađene klase `Object`. Svakom objektu klase pridružen je implicitno jedan (i samo jedan) *ključ* (engl. *lock*), kome se ne može pristupiti direktno, već samo implicitno, a kojim se može obezbediti ekskluzivan pristup nad tim objektom.
- Ključ može u jednom trenutku držati najviše jedna nit (zaključavanje je ekskluzivno). Nit traži ključ implicitno na jedan od sledeća dva načina:
 - pozivom operacije klase označene kao `synchronized`
 - ulaskom u blok naredbi koji je označen kao `synchronized`.
 Ukoliko je ključ zauzet, pozivajuća nit mora da čeka (suspenduje se): kada jedan proces (nit) dobije ključ, ostali to ne mogu, odnosno moraju da čekaju pre ulaska u sinhronizovanu metodu ili blok tog objekta.
- Operacije označene kao `synchronized` su međusobno isključive, kao kod klasičnih monitora: izvršavanje ovakve operacije omogućeno je pozivajućem procesu (niti) samo ukoliko on može da dobije ključ pridružen objektu čiju operaciju poziva. Pristup do operacija koje nisu označene kao `synchronized` je uvek omogućen (ne zahteva ključ). Na primer:

```

class SharedCoord {

    private double myX, myY;

    public SharedCoord (double initX, double initY) {
        myX = initX; myY = initY;
    };

    public synchronized Pair<double,double> read () {
        return new Pair<double,double>(myX,myY);
    };

    public synchronized void write (double x, double y) {
        myX = x; myY = y;
    };
}

```

- Sinhronizacijom na nivou bloka se ključ nad označenim objektom (kao parametrom bloka označenog kao `synchronized`) zahteva na ulasku u blok kao sekvencu naredbi, tako da se ta sekvenca izvršava isključivo sa drugim izvršavanjima koja drže ključ nad istim tim objektom:

```

synchronized(object) {
    ...
}

```

- Na primer, operacija označena kao `synchronized` se implicitno implementira na sledeći način:

```

public Pair<double,double> read () {
    synchronized(this) {
        return new Pair<double,double>(myX,myY);
    }
}

```

- Sinhronizaciju na nivou bloka treba izbegavati, odnosno upotrebljavati krajnje pažljivo i ograničeno, jer inače ona dovodi do nepreglednog koda, pošto se kod za sinhronizaciju vezanu za jedan objekat tada ne nalazi obavezno samo unutar operacije iste klase tog objekta, nego je potencijalno rasut po programu, pa se sinhronizacija vezana za jedan objekat ne može ni razumeti samo posmatranjem jedne klase.
- Ovakvo zaključavanje ne utiče na statičke podatke članove klase. Međutim, zaključavanje statičkih podataka članova klase može se postići na sledeći način. Naime, u jeziku Java, za svaku klasu u programu postoji odgovarajući objekat ugrađene klase `Class`. Zaključavanjem ovog objekta, zapravo se zaključavaju statički podaci članovi:

```

synchronized(this.getClass()) {...}

```

- Za uslovnu sinhronizaciju služe sledeće operacije klase `Object` (iz koje su implicitno izvedene sve klase, direktno ili indirektno):

```

public void wait      ();
public void notify    ();
public void notifyAll ();

```

- Ove operacije smeju da se pozivaju samo iz konteksta niti koja drži ključ nad objektom; u suprotnom, baca se izuzetak tipa `IllegalMonitorStateException`.
- Operacija `wait()` bezuslovno blokira pozivajuću nit i oslobađa ključ nad objektom. Ukoliko je poziv napravljen iz ugneždenog poziva metode drugog monitora, oslobađa se ključ samo za taj unutrašnji monitor.

- Operacija `notify()` deblokira jednu nit blokiranu sa `wait()`. Osnovna verzija jezika Java ne definiše koja je to nit od blokiranih, ali Real-Time Java to definiše. Operacija `notify()` ne oslobađa ključ koji pozivajuća nit ima nad objektom, pa deblokirana nit mora da čeka da dobije ključ pre nego što nastavi izvršavanje, kao kod klasičnih uslovnih promenljivih.
- Slično, operacija `notifyAll()` deblokira sve blokirane niti koje potom, jedna po jedna, nastavljaju izvršavanje metoda u kojima su pozvale `wait()`. Ukoliko blokiranih niti nema, ove dve operacije nemaju efekta.
- Najvažnija razlika između opisanog mehanizma i koncepta klasičnih uslovnih promenljivih je u tome što jedan objekat (kao monitor) u jeziku Java ima uvek jednu i samo jednu, i to implicitnu svoju uslovnu promenljivu (nad kojom deluju opisane operacije `wait` i `notify/notifyAll`), za razliku od klasičnih uslovnih promenljivih kojih može biti više u istom objektu-monitoru i koje se eksplicitno deklarišu i imenuju. Zbog toga se niti koje čekaju na ispunjenje različitih uslova (npr. u ograničenom baferu, proizvođači i potrošači čekaju na različite uslove) ne mogu razvrstati čekanjem na različite, imenovane uslovne promenljive, već se oni suspenduju na istoj (implicitnoj i bezimenoj) uslovnoj promenljivoj pridruženoj objektu, i čekaju u istom redu čekanja. Kao posledica toga, kada se operacijom `notify` deblokira jedan proces, taj proces ne može uvek da računa na to da je uslov na koji je on čekao ispunjen, pošto je možda deblokiran zato što je neki drugi uslov ispunjen i zbog toga pozvan `notify()`. Za mnoge slučajeve ovo nije problem, pošto su uslovi međusobno isključivi. Na primer, kod ograničenog bafera, ukoliko neki proces čeka na jedan uslov (npr. da se u baferu pojavi element jer je bafer prazan), onda sigurno nema procesa koji čekaju na suprotan uslov (da se u baferu pojavi slobodno mesto jer je bafer pun). U suprotnom, ako u redu čekanja postoje procesi koji čekaju na različite uslove, proces koji je deblokiran mora ponovo da ispita svoj uslov i ponovo se blokira ako taj uslov nije ispunjen, a da pritom deblokira naredni proces pozivom `notify()` itd. Zato je u tim slučajevima lakše koristiti `notifyAll()`, ali tada svi deblokirani procesi moraju ponovo da provere svoje uslove i ponovo se blokiraju ako uslov na koji čekaju nije ispunjen, pa se tada poziv `wait` ugrađuje u petlju tipa `while` koja ispituje uslov (umesto naredbe `if`). Ovo pak podrazumeva mnogo veće nepotrebne režijske troškove (vreme) tokom izvršavanja, pa je uslovna sinhronizacija u Javi ponekad mnogo manje efikasna nego kod klasičnih uslovnih promenljivih.
- Primer ograničenog bafera (u ovom slučaju, umesto `while` može da stoji i `if`, a umesto `notifyAll` da stoji `notify`):

```
public class BoundedBuffer {  
  
    private int buffer[];  
    private int head = 0, tail = 0;  
    private int size, numberInBuffer = 0;  
  
    public BoundedBuffer (int sz) {  
        buffer = new int[size = sz];  
    }  
  
    public synchronized void append (int item) throws InterruptedException {  
        while (numberInBuffer == size) wait();  
        buffer[tail] = item;  
        tail = (tail + 1) % size ;  
        numberInBuffer++;  
        notifyAll();  
    }  
  
    public synchronized int take () throws InterruptedException {  
        while (numberInBuffer == 0) wait();
```

```

    int data = buffer[head];
    head = (head + 1) % size ;
    numberInBuffer--;
    notifyAll();
    return data;
}
}

```

Klasifikacija poziva operacija

- Posmatrano sa strane pozivaoca, odnosno kontrole toka (procesa, niti) u čijem kontekstu se poziv operacije, ulaza, ili nekog drugog servisa klijenta izvršava, taj poziv može biti:
 - *sinhron* (engl. *synchronous*), što znači da pozivalac, odnosno pozivajuća kontrola toka (proces, nit) čeka dok se pozvana operacija (servis) ne završi, pa tek onda nastavlja dalje svoje izvršavanje;
 - *asinhron* (engl. *asynchronous*), što znači da pozivalac ne čeka na završetak pozvane operacije (servisa), već odmah po upućenom pozivu nastavlja svoje izvršavanje uporedo i nezavisno od izvršavanja pozvane operacije.

Implementacija sinhronizacionih primitiva

- U nastavku je opisana realizacija sinhronizacionih primitiva (semafora i događaja) u školskom jezgru. Takođe je prikazana i realizacija monitora na jeziku C++ korišćenjem ovih koncepata školskog jezgra.

Semafor

- Klasom `Semaphore` realizovan je koncept standardnog brojačkog semafora.
- Operacija `signalWait(s1, s2)` izvršava neprekidivu sekvencu operacija `s1->signal()` i `s2->wait()`. Ova operacija je pogodna za realizaciju uslovnih promenljivih.
- Operacije `lock()` i `unlock()` zaključavaju kritičnu sekciju – celo jezgro; pretpostavlja se da su one implementirane na opisani način, odgovarajućom hardverskom podrškom.
- Izvorni kod klase `Semaphore`:

```

class Semaphore {
public:
    Semaphore (int initValue=1) : val(initValue) {}
    ~Semaphore ();

    void wait    ();
    void signal ();

    friend void signalWait (Semaphore* s, Semaphore* w);

    int value () { return val; };

protected:

    void block ();
    void deblock ();

    int val;

private:

```

```

    Queue blocked;

};

Semaphore::~Semaphore () {
    lock();
    for (IteratorCollection* it=blocked->getIterator();
        !it->isDone(); it->next()) {
        Thread* t = (Thread*)it->currentItem();
        Scheduler::Instance()->put(t);
    }
    unlock();
}

void Semaphore::block () {
    if (Thread::runningThread->setContext()==0) {
        blocked->put(Thread::runningThread->getCEForSemaphore());
        Thread::runningThread = Scheduler::Instance()->get();
        Thread::runningThread->resume();          // context switch
    } else return;
}

void Semaphore::deblock () {
    Thread* t = (Thread*)blocked->get();
    Scheduler::Instance()->put(t);
}

void Semaphore::wait () {
    lock();
    if (--val<0)
        block();
    unlock();
}

void Semaphore::signal () {
    lock();
    if (val++<0)
        deblock();
    unlock();
}

void signalWait (Semaphore* s, Semaphore* w) {
    lock();
    if (s && s->val++<0) s->deblock();
    if (w && --w->val<0) w->block();
    unlock();
}

```

Događaj

- Događaj je ovde definisan kao binarni semafor. Izvorni kod za klasu Event izgleda ovako:

```

class Event : public Semaphore {
public:
    Event ();

    void wait ();
    void signal ();

};

Event::Event () : Semaphore(0) {}

void Event::wait () {
    lock();
    if (--val<0)
        block();
    unlock();
}

void Event::signal () {
    lock();
    if (++val<=0)
        deblock();
    else
        val=1;
    unlock();
}

```

Monitor

- Na jeziku C++, međusobno isključenje neke operacije (funkcije članice) može da se obezbedi na jednostavan način pomoću semafora:

```

class Monitor {
public:
    Monitor () : sem(1) {}
    void criticalSection ();
private:
    Semaphore sem;
};

void Monitor::criticalSection () {
    sem.wait();
    //... telo kritične sekcije
    sem.signal();
}

```

- Međutim, opisano rešenje ne garantuje ispravan rad u svim slučajevima. Na primer, ako funkcija vraća rezultat nekog izraza iza naredbe `return`, ne može se tačno kontrolisati trenutak oslobađanja kritične sekcije, odnosno poziva operacije *signal*. Drugi, teži slučaj je izlaz i potprograma u slučaju izuzetka. Na primer:

```

int Monitor::criticalSection () {
    sem.wait();
    return f()+2/x; // gde pozvati signal()?
}

```

- Opisani problem se jednostavno rešava na sledeći način. Potrebno je unutar funkcije koja predstavlja kritičnu sekciju, na samom početku, definisati lokalni automatski objekat koji

će u svom konstruktoru imati poziv operacije *wait*, a u destrukturu poziv operacije *signal*. Semantika jezika C++ obezbeđuje da se destruktork ovog objekta uvek poziva tačno na izlasku iz funkcije, pri svakom načinu izlaska (izraz iza `return` ili izuzetak).

- Jednostavna klasa `Mutex` obezbeđuje ovakvu semantiku:

```
class Mutex {
public:

    Mutex (Semaphore* s) : sem(s) { if (sem) sem->wait(); }
    ~Mutex ()                  { if (sem) sem->signal(); }

private:
    Semaphore *sem;
};
```

- Upotreba ove klase je takođe veoma jednostavna: ime samog lokalnog objekta nije uopšte bitno, jer se on i ne koristi eksplicitno.

```
void Monitor::criticalSection () {
    Mutex dummy(&sem);
    //... telo kritične sekcije
}
```

Ograničeni bafer

- Jedna realizacija ograničenog bafera u školskom jezgru može da izgleda kao što je prikazano u nastavku.
- Treba primetiti sledeće: operacija oslobađanja kritične sekcije (`mutex.signal()`) i blokiranja na semaforu za čekanje na prazan prostor (`notFull.wait()`) moraju da budu nedeljive, inače bi moglo da se dogodi da između ove dve operacije neki proces uzme poruku iz bafera, a prvi proces se blokira na semaforu `notFull` bez razloga (problem utrivanja). Isto važi i u operaciji *receive*. Zbog toga je upotrebljena neprekidiva sekvenca `signalWait()`.
- Pomoćna operacija `receive()` koja vraća `int` je neblokirajuća: ako je bafer prazan, ona vraća 0, inače smešta jedan element u argument `i` i vraća 1. Kompletan kod izgleda ovako:

```
class MsgQueue {
public:

    MsgQueue () : mutex(1), notEmpty(0), notFull(0) {}
    ~MsgQueue () { mutex.wait(); }

    void      send      (CollectionElement*);
    Object*   receive   ();                // blocking
    int       receive   (Object**);        // nonblocking
    void      clear     ();

    Object*   first     ();
    int       isEmpty   ();
    int       isFull    ();
    int       size      ();

private:

    Queue rep;
    Semaphore mutex, notEmpty, notFull;
};
```

```
void MsgQueue::send (CollectionElement* ce) {
    Mutex dummy(&mutex);
    while (rep.isFull()) {
        signalWait(&mutex, &notFull);
        mutex.wait();
    }
    rep.put(ce);
    if (notEmpty.value() < 0) notEmpty.signal();
}
```

```
Object* MsgQueue::receive () {
    Mutex dummy(&mutex);
    while (rep.isEmpty()) {
        signalWait(&mutex, &notEmpty);
        mutex.wait();
    }
    Object* temp=rep.get();
    if (notFull.value() < 0) notFull.signal();
    return temp;
}
```

```
int MsgQueue::receive (Object** t) {
    Mutex dummy(&mutex);
    if (rep.isEmpty()) return 0;
    *t=rep.get();
    if (notFull.value() < 0) notFull.signal();
    return 1;
}
```

```
void MsgQueue::clear () {
    Mutex dummy(&mutex);
    rep.clear();
}
```

```
Object* MsgQueue::first () {
    Mutex dummy(&mutex);
    return rep.first();
}
```

```
int MsgQueue::isEmpty () {
    Mutex dummy(&mutex);
    return rep.isEmpty();
}
```

```
int MsgQueue::isFull () {
    Mutex dummy(&mutex);
    return rep.isFull();
}
```

```
int MsgQueue::size () {
    Mutex dummy(&mutex);
    return rep.size();
}
```

Zadaci

4.1

Potrebno je napraviti sistem za kontrolu pristupa nekoj podzemnoj garaži. U garažu ne može stati više od približno N vozila. Semafor na ulazu u garažu kontroliše ulaz vozila, dok detektori vozila na ulazu i izlazu iz garaže prate tok vozila. Potrebno je napisati dva procesa i monitor kojima se kontroliše tok saobraćaja u garaži. Prvi proces nadzire ulazni, a drugi izlazni detektor saobraćaja. Monitor kontroliše semafor na ulazu u garažu. Pretpostavlja se da su realizovane sledeće globalne funkcije:

```
int carsExited (); // Vraća broj vozila koja su napustila garažu
                  // od trenutka poslednjeg poziva ove funkcije

int carsEntered (); // Vraća broj vozila koja su ušla u garažu
                  // od trenutka poslednjeg poziva ove funkcije

void setLights (Color); // Postavlja svetlo semafora na zadatu boju:
                        // enum Color {Red, Green};

void delay10milisec (); // Blokira pozivajući proces na 10 milisekundi
```

Program treba da očitava senzore (preko funkcija `carsExited()` i `carsEntered()`) svakih 10 milisekundi, sve dok garaža ne postane prazna ili puna. Kada garaža postane puna (i na semaforu se upali crveno svetlo), proces koji nadgleda ulaz u garažu ne treba više da poziva funkciju `carsEntered()`. Slično, kada garaža postane prazna, proces koji nadgleda izlaz iz garaže ne treba više da poziva funkciju `carsExited()`. Prikazati rešenje korišćenjem:

- klasičnih monitora i uslovnih promenljivih
- zaštićenih objekata u jeziku Ada
- koncepata iz jezika Java
- školskog jezgra.

Rešenje - Ada

```
N : constant Natural := ...;
type Color is (red, green);

protected type Garage is
  entry getIn (num : Natural);
  entry getOut (num : Natural);
private
  counter : Integer := 0;
end Garage;

protected body Garage is

  entry getIn (num : Natural) when counter < N is
  begin
    counter := counter + num;
    if counter >= N then
      setLights(red);
    end if;
  end getIn;

  entry getOut (num : Natural) when current > 0 is
```

```

begin
  counter := counter - num;
  if counter < N then
    setLights(green)
  end if;
end getOut;

end Garage;

gar : Garage;

task type In;
task body In is
  c : Natural := 0;
begin
  loop
    c := carsEntered();
    if c > 0 then t.getIn(c);
    delay10milisec();
  end loop
end In;

task type Out;
task body Out is
  c : Natural := 0;
begin
  loop
    c := carsExited();
    if c > 0 then t.getOut(c);
    delay10milisec();
  end loop
end Out;

```

4.2 Cigarette smokers

Posmatra se sistem od tri procesa koji predstavljaju pušače i jednog procesa koji predstavlja agenta. Svaki pušač ciklično zavija cigaretu i puši je. Za cigaretu su potrebna tri sastojka: duvan, papir i šibica. Jedan pušač ima samo duvan, drugi papir, a treći šibice. Agent ima neograničene zalihe sva tri sastojka. Agent postavlja na sto dva sastojka izabrana slučajno. Pušač koji poseduje treći potreban sastojak može tada da uzme ova dva, zavije cigaretu i puši. Kada je taj pušač popužio svoju cigaretu, on javlja agentu da može da postavi nova dva sastojka, a ciklus se potom ponavlja. Realizovati deljeni objekat koji sinhronizuje agenta i tri pušača. Prikazati rešenje korišćenjem:

- zaštićenih objekata u jeziku Ada
- klasičnih monitora i uslovnih promenljivih.

Rešenje

(a) Zaštićeni objekti u jeziku Ada:

```

protected Agent is
  entry  takeTobaccoAndPaper ();
  entry  takePaperAndMatch   ();
  entry  takeTobaccoAndMatch ();
  procedure finishedSmoking ();
private
  tobaccoAvailable : Boolean := False;
  paperAvailable   : Boolean := False;

```



```

    matchAvailable : Boolean := False;
end Agent;

protected body Agent is

    procedure putItems () is
    begin
        ... -- Randomly select two items and put them on the table
        -- by setting two Boolean variables to True
    end putItems;

    entry takeTobaccoAndPaper () when tobaccoAvailable and paperAvailable is
    begin
        tobaccoAvailable := False;
        paperAvailable := False;
    end;

    entry takePaperAndMatch () when paperAvailable and matchAvailable is
    begin
        paperAvailable := False;
        matchAvailable := False;
    end;

    entry takeTobaccoAndMatch () when tobaccoAvailable and matchAvailable is
    begin
        tobaccoAvailable := False;
        matchAvailable := False;
    end;

    procedure finishedSmoking () is
    begin
        putItems();
    end;

begin
    putItems();
end Agent;

task SmokerWithPaper;

task body SmokerWithPaper is
begin
    loop
        Agent.takeTobaccoAndMatch();
        -- Smoke
        Agent.finishedSmoking();
    end loop;
end SmokerWithPaper;

```

(b) Klasični monitori i uslovne promenljive:

```

monitor Agent;
    export takeTobaccoAndPaper,
           takePaperAndMatch,
           takeTobaccoAndMatch,
           finishedSmoking;
    var
        tobaccoAvailable : boolean;
        paperAvailable   : boolean;
        matchAvailable   : boolean;
        waitTobaccoAndPaper : condition;

```

```
waitPaperAndMatch : condition;
waitTobaccoAndMatch : condition;

procedure putItems ();
begin
    ... (* Randomly select two items and put them on the table
        by setting two Boolean variables to True *)
end;

procedure takeTobaccoAndPaper ();
begin
    if not (tobaccoAvailable and paperAvailable) then
        wait(waitTobaccoAndPaper);
    tobaccoAvailable := false;
    paperAvailable := false;
end;

procedure takePaperAndMatch ();
begin
    if not (paperAvailable and matchAvailable) then
        wait(waitPaperAndMatch);
    paperAvailable := false;
    matchAvailable := false;
end;

procedure takeTobaccoAndMatch ();
begin
    if not (tobaccoAvailable and matchAvailable) then
        wait(waitTobaccoAndMatch);
    tobaccoAvailable := false;
    matchAvailable := false;
end;

procedure finishedSmoking ();
begin
    putItems();
    if tobaccoAvailable and paperAvailable then
        signal(waitTobaccoAndPaper);
    if paperAvailable and matchAvailable then
        signal(waitPaperAndMatch);
    if tobaccoAvailable and matchAvailable then
        signal(waitTobaccoAndMatch);
end;

begin
    putItems();
end;

process SmokerWithPaper;
begin
    loop
        Agent.takeTobaccoAndMatch();
        -- Smoke
        Agent.finishedSmoking();
    end
end;
```

4.3

U školskom jezgru treba realizovati događaj sa mogućnošću složenog logičkog uslova čekanja. Događaj treba da bude vlasništvo neke niti, a ne nezavisan objekat. Samo nit koja sadrži dati događaj može čekati na tom događaju. Jedna nit sadrži više događaja (konstantan broj N). Osim proste operacije čekanja na događaju, može se zadati i složeni uslov čekanja na događajima: operacija `waitAnd(e1,e2)` blokira nit sve dok se ne pojavi signal na oba događaja `e1` i `e2`.

Rešenje

Postojeću klasu `Thread` iz Jezgra treba proširiti na sledeći način:

```
typedef unsigned int EventID;
const EventID NumOfEvents = 32;

class IllegalEventIDException {...};
class IllegalContextException {...};

class Thread {
public:
    ...
    void wait      (EventID)
        throw (IllegalEventIDException*,IllegalContextException*);
    void waitAnd   (EventID e1, EventID e2)
        throw (IllegalEventIDException*,IllegalContextException*);
    void signal    (EventID)
        throw (IllegalEventIDException*);

protected:
    ...
    void checkEvent (EventID) throw (IllegalEventIDException*);
    void checkContext () throw (IllegalContextException*);
    int  isWaitingForEvent ();
    void block();
    void deblock();

private:
    ...
    struct Event {
        Event() : val(0), waitingOn(0) {}
        unsigned int val, waitingOn;
    };

    Event events[NumOfEvents];
};

...

void Thread::checkEvent (EventID e) throw (IllegalEventIDException*) {
    if (e<0 || e>=NumOfEvents)
        throw new IllegalEventIDException("Thread::checkEvent()",this,e);
}

void Thread::checkContext () throw (IllegalContextException*) {
    if (runningThread!=this) throw new
        IllegalContextException("Thread::checkContext()",this,runningThread);
}
```

```
void Thread::block () {
    if (runningThread->setContext()==0) {
        runningThread = Scheduler::Instance()->get();
        runningThread->resume();          // context switch
    } else return;
}

void Thread::deblock () {
    Scheduler::Instance()->put(this);
}

int Thread::isWaitingForEvent () {
    for (EventID e=0; e<NumOfEvents; e++) {
        if (events[e].waitingOn && events[e].val==0) return 1;
    }
    return 0;
}

void Thread::wait (EventID e)
    throw (IllegalEventIDException*,IllegalContextException*)
{
    checkEvent(e);
    checkContext();
    lock();
    if (events[e].val==0) {
        events[e].waitingOn=1;
        block();
    }
    events[e].waitingOn=events[e].val=0;
    unlock();
}

void Thread::waitAnd (EventID e1, EventID e2)
    throw (IllegalEventIDException*,IllegalContextException*)
{
    checkEvent(e1); checkEvent(e2);
    checkContext();
    lock();
    events[e1].waitingOn=events[e2].waitingOn=1;
    if (isWaitingOnEvent())
        block();
    events[e1].waitingOn=events[e1].val=0;
    events[e2].waitingOn=events[e2].val=0;
    unlock();
}

void Thread::signal (EventID e) throw (IllegalEventIDException*) {
    checkEvent(e);
    lock();
    events[e].val=1;
    if (!isWaitingForEvent())
        deblock();
    unlock();
}
```

4.4

Multicast je konstrukt koji omogućava da jedan proces pošalje istu poruku (podatak) grupi procesa koji čekaju na poruku. Data je sledeća specifikacija apstrakcije *Multicast*:

```
class Multicast {
public:
    void send (Data* d);
    Data* receive ();
};
```

Proces-primalac izražava svoju želju da primi podatak pozivajući operaciju `receive()`. Ovaj poziv je blokirajući. Proces-pošiljalac šalje poruku pozivom operacije `send()`. Svi procesi koji su trenutno blokirani čekajući da prime poruku oslobađaju se kada se pozove `send()`, a podatak koji je poslat im se prosleđuje. Kada se završi operacija `send()`, svi naredni pozivi operacije `receive()` blokiraju procese do narednog slanja podatka. Realizovati apstrakciju *Multicast* korišćenjem:

- koncepata iz jezika Java
- školskog jezgra.

Rešenje

(a) Java

```
class Multicast {

    public synchronized void send (Data d) {
        msg = d;
        notifyAll();
    }

    public synchronized Data receive () {
        wait();
        return msg;
    }

    private Data msg;

}
```

(b) C++ i školsko jezgro:

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

    void send (Data* d);
    Data* receive ();

private:
    Data* msg;
    Semaphore mutex, cond;
};

Multicast::Multicast () : mutex(1), cond(0), msg(0) {}

void Multicast::send (Data* d) {
    Mutex dummy(&mutex);
    msg = d;
    while (cond.value() < 0) cond.signal();
}
```

```
Data* Multicast::receive () {  
    Mutex dummy(&mutex);  
    signalWait(&mutex, &cond);  
    mutex.wait();  
    return msg;  
}
```

Zadaci za samostalan rad

4.4

Korišćenjem školskog jezgra, realizovati ograničeni bafer (engl. *bounded buffer*) koji će konkurentni aktivni klijenti koristiti kao čuvani (engl. *guarded*) jedinstveni (engl. *singleton*) objekat. Proizvođači (engl. *producers*), odnosno potrošači (engl. *consumers*) mogu da stave, odnosno izvade nekoliko elemenata u toku svog jednog pristupa baferu, ali u jednoj istoj nedeljivoj transakciji. Broj elemenata koji se stavlja, odnosno uzima u jednoj transakciji određuje sam proizvođač, odnosno potrošač sopstvenim algoritmom, pa zato bafer ima samo operacije za stavljanje i uzimanje po jednog elementa. Treba obezbediti i uobičajenu potrebnu sinhronizaciju u slučaju punog, odnosno praznog bafera. Prikazati klase za bafer, proizvođač i potrošač.

4.5

Projektuje se konkurentni sistem za modelovanje jednog klijent/server sistema. Server treba modelovati jedinstvenim (*singleton*) sinhronizovanim objektom (monitorom). Klijenti su aktivni objekti koji ciklično obavljaju svoje aktivnosti. Pre nego što u jednom ciklusu neki klijent započne svoju aktivnost, dužan je da od servera traži dozvolu u obliku "žetona" (*token*). Kada dobije žeton, klijent započinje aktivnost. Po završetku aktivnosti, klijent vraća žeton serveru. Server vodi računa da u jednom trenutku ne može biti izdato više od N žetona: ukoliko klijent traži žeton, a ne može da ga dobije jer je već izdato N žetona, klijent se blokira. Prikazati rešenje korišćenjem:

- klasičnih monitora i uslovnih promenljivih
- zaštićenih objekata u jeziku Ada
- koncepata iz jezika Java
- školskog jezgra.

4.7

Broadcast je sličan konceptu *multicast*, osim što se poruka šalje *svim* učesnicima u sistemu koji mogu da prime poruku. Intefrejs apstrakcije *Broadcast* izgleda kao *Multicast* u pokazanom zadatku, ali je razlika u tome što se pošiljalac blokira prilikom slanja poruke sve dok *svi* procesi-primaoci (a ima ih tačno N u sistemu) ne prime poruku. Ako se u međuvremenu pošalje nova poruka, ona se stavlja u red čekanja. Realizovati apstrakciju *Broadcast* korišćenjem:

- klasičnih monitora i uslovnih promenljivih
- zaštićenih objekata u jeziku Ada
- koncepata iz jezika Java
- školskog jezgra.

Sinhronizacija i komunikacija pomoću razmene poruka

- *Prosleđivanje* (ili *razmena*) *poruka* (engl. *message passing*) podrazumeva upotrebu konstrukata za *slanje* i *prijem* poruke: jedan proces na određeni način eksplicitno traži *slanje* poruke, dok neki drugi proces ili procesi traže *prijem* poruke.
- Konstrukti za slanje i prijem poruke po pravilu implicitno uključuju i određenu sinhronizaciju, pa se ovim mehanizmom procesi sinhronizuju, ali i razmenjuju informacije (kao sadržaj poruke).
- U različitim programskim jezicima, bibliotekama i operativnim sistemima, konstrukti za slanje i prijem poruke realizovani su na mnogo različitih načina, sa različitom semantikom i notacijom, a koji u suštini variraju u odnosu na:
 - model sinhronizacije procesa koja je implicitno uključena u konstrukte slanja i prijema poruka,
 - način imenovanja procesa kojima se šalje ili od kojih se prima poruka, i
 - strukturu same poruke.
- Jezik Java ne podržava razmenu poruka neposrednim jezičkim konstruktima. Jezik Ada ima napredne konstrukte za razmenu poruka, ali će oni ovde biti prikazani samo ukratko.

Sinhronizacija procesa

- Na prijemnoj strani, odnosno u procesu koji želi da primi poruku, konstrukt kojim se zahteva prijem poruke može biti:
 - *sinhron* (engl. *synchronous*), odnosno *blokirajući* (engl. *blocking*): ukoliko poruka nije stigla do primaoca i zbog toga ne može biti isporučena (recimo zato što nije ni poslata), pozivajući proces se suspenduje dok poruka ne stigne; na primer:

```
...
receive(&message);
..
```

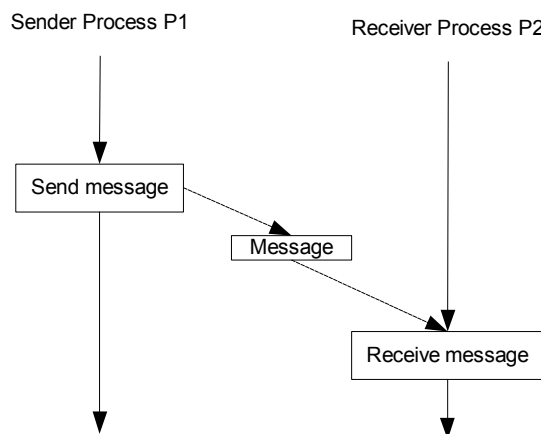
- *asinhron* (engl. *asynchronous*), odnosno *neblokirajući* (engl. *non-blocking*): ukoliko poruka nije stigla, pozivajućem procesu se vraća kontrola i on nastavlja izvršavanje, ali uz povratnu informaciju da poruka nije isporučena (npr. preko povratnog statusa ili podignutog izuzetka); pozivajući proces tada mora da preduzme odgovarajuću alternativnu akciju u odnosu na onu koju bi uradio kada je poruka isporučena (npr. obrada greške):

```
...
if (receive(&message) != 0) {
    ... // Message not delivered
}
..
```

- Treba primetiti da u svim varijantama postoji implicitna sinhronizacija u smislu da primalac ne može da primi poruku pre nego što je ona uopšte poslata, odnosno da poruka ne može biti isporučena pošiljaocu ako ta poruka uopšte nije ni poslata. Iako ovo deluje očigledno, treba ovo uporediti sa međuprocesnom komunikacijom pomoću deljene

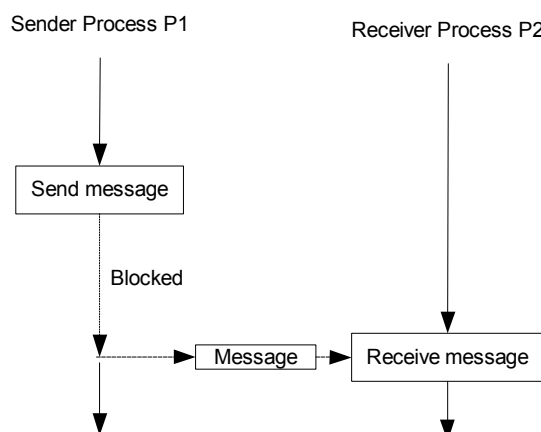
promenljive, gde primalac informacije može da pročita vrednost deljene promenljive ne znajući da li je ta vrednost upisana od strane pošiljaoca, odnosno čak i pre nego što je ta vrednost upisana (tj. informacija poslata), pa analogna sinhronizacija tamo ne postoji.

- Na strani pošiljaoca, konstrukt slanja se u pogledu sinhronizacije operacije može generalno klasifikovati u sledeće kategorije:
 - *Asinhrono* ili *neblokirajuće* slanje, *bez čekanja* (engl. *asynchronous, no-wait*): proces pošiljalac dolazi do tačke u kojoj izvršava konstrukt slanja poruke i odmah nastavlja svoje izvršavanje posle slanja, ne čekajući da poruka bude primljena. Proces primalac, kao uporedni tok kontrole, izvršava prijem poruke pre ili kasnije (možda mnogo kasnije ili nikada). Ovakav model podržavaju neki programski jezici (npr. CONIC) i POSIX. Analogija: slanje pisama običnom ili elektronskom poštom, slanje SMS poruke (bez povratne informacije o uručanju poruke).



Ovakav pristup zahteva odgovarajuće bafere za prihvatanje poruka i amortizaciju različitih brzina slanja i prijema poruka. Problem je što je prostor za bafere generalno ograničen, pa je pitanje šta se dešava u slučaju punih bafera: pošiljalac tada mora da se suspenduje ili će slanje poruke biti odbijeno i tretirano kao izuzetak.

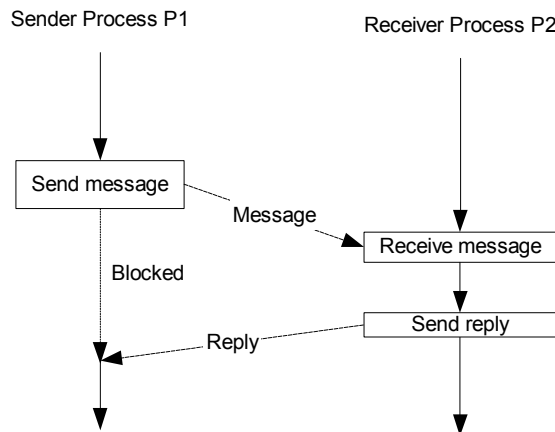
- *Sinhrono*, *blokirajuće* slanje, ili *randevu* (engl. *synchronous, rendez-vous*): pošiljalac izvršava konstrukt slanja poruke i suspenduje se (ne nastavlja svoje izvršavanje) sve dok poruka nije primljena i tek tada nastavlja svoje izvršavanje. Neki jezici podržavaju ovaj koncept (npr. CSP i occam2). Analogija: telefonski poziv.



Ovakav model u principu ne zahteva bafere za smeštanje poruka.

- *Udaljeni poziv* ili *prošireni randevu* (engl. *remote procedure call, RPC, extended rendez-vous*): pošiljalac izvršava konstrukt slanja poruke i suspenduje se sve dok ne dobije odgovor od primaoca; primalac dolazi do tačke prijema, prihvata poruku,

obrađuje je, priprema odziv i šalje odgovor pošiljaocu. Dakle, pošiljalac i primalac organizuju „susret“ (randevu): prvi koji od njih dođe do tačke susreta (konstrukt slanja ili prijema), čeka onog drugog; tokom susreta razmenjuju informacije (pošiljalac šalje poruku, a primalac vraća odgovor), pri čemu primalac može da izvrši proizvoljnu obradu tokom obrade poruke i pripreme odgovora; nakon toga se učesnici razilaze. Ovo se može shvatiti i kao poziv udaljene procedure (procedure koja se izvršava u kontekstu procesa primaoca): poslata poruka šalje ulazne argumente, obrada poruke je procedura koja se izvršava po prijemu poruke, a odgovor sadrži izlazne argumente/rezultate pozvane procedure; prijemni proces je taj koji prihvata zahteve za „pozivima“ procedure i opslužuje te zahteve, jedan po jedan. Ovakav koncept podržavaju razni jezici (npr. Ada, SR, CONIC).



- Sinhrono slanje se može realizovati pomoću asinhronog slanja i sinhronog prijema, tako što pošiljalac pošalje poruku i odmah čeka na prijem potvrde o uručanju poruke („povratnice“, engl. *acknowledgement*):

Sender process P1:	Receiver process P2:
<code>async_send(message);</code>	<code>sync_receive(message);</code>
<code>sync_receive(ack);</code>	<code>async_send(ack);</code>

Slično, udaljeni poziv se može realizovati pomoću sinhronog slanja i sinhronog prijema:

Sender process P1:	Receiver process P2:
<code>sync_send(message);</code>	<code>sync_receive(message);</code>
<code>sync_receive(reply);</code>	<code>...</code>
	<code>construct reply;</code>
	<code>...</code>
	<code>sync_send(reply);</code>

- Kako je, prema tome, asinhrono slanje (uz sinhroni prijem) zapravo elementarni konstrukt pomoću koga se mogu realizovati i ostali, može se pomisliti da je on dovoljan i jedini potreban u programskim jezicima.
- Međutim, asinhrono slanje ima sledeće nedostatke:
 - potrebni su baferi za prihvatanje poruka koje su poslate a još nisu primljene; da bi se odložio momenat kada asinhrono slanje nije više moguće zbog punog bafera, baferi moraju da budu dovoljno veliki ili čak (logički) neograničeni, što može biti previše zahtevno, režijski skupo ili neprihvatljivo;
 - programiranje samo korišćenjem asinhronog slanja je generalno teško, upravo zbog toga što pošiljalac često treba da obrađuje poruke od primaoca koje vraćaju odgovore i rezultate na ranije poslate poruke, a te poruke i odgovori se mogu međusobno učešljavati, pa i preticati; zato je ponekad potrebno uparivati takve

poruke i odgovore koji im odgovaraju, odnosno uvoditi njihovu identifikaciju, ali i obradu situacija u kojima se one učešljavaju i pretiču; u tim situacijama se često pribegava sinhronom slanju, iako ono smanjuje konkurentnost;

- asinhrona komunikacija uzrokuje "bajate" poruke: kada primalac primi poruku, pošiljalac je možda već daleko odmakao u svom izvršavanju i informacija u poruci možda više nije validna, a u programima često postoje situacije u kojima to nije dobro.
- Kada ovi nedostaci asinhronog slanja postanu neprihvatljivi, potrebno je implementirati sinhrono slanje (u kom će pošiljalac čekati na prijem poruke ili čak povratak odgovora, a time primalac biti siguran da je poruka „sveža“, odnosno da pošiljalac nije odmakao dalje u svom izvršavanju). Ukoliko se sinhrona komunikacija ne obezbedi kao poseban konstrukt, implementiran generički, a onda koristi u svim ovakvim situacijama, jednobrazno (npr. kao poziv potprograma), nego se direktno upotrebljava samo asinhrono slanje, program lako može da postane glomazan, nepregledan i težak za razumevanje i proveru korektnosti.
- Asinhrono slanje može biti bolje rešenje i neophodno kada je potrebno procese „dekuplovati“, odnosno učiniti nezavisnijim (pošiljalac ne mora da čeka da primalac bude spreman na prijem poruke), a kada ne postoje opisani problemi asinhronog slanja. U takvim situacijama asinhrono slanje značajno povećava konkurentnost.
- U slučaju da jezik ili okruženje podržava samo sinhrono slanje, asinhrono slanje se može realizovati uvođenjem bafera poruka kao posrednika između pošiljaoca i primaoca. Ovaj bafer može biti implementiran kao proces koji prima poruke od pošiljaoca i šalje poruke primaocu: umesto da direktno pošalje poruku primaocu, što ga potencijalno suspenduje, pošiljalac ostavi poruku u bafer iz kog će je primalac pokupiti onda kada njemu to bude odgovaralo. Na taj način se pošiljalac i primalac „dekupluju“. Ovakva implementacija pomoću procesa koji implementira bafer može da utiče na degradaciju performansi zbog većeg broja procesa u sistemu.
- Primetiti da razmena poruka preko bafera implementiranog kao deljeni objekat ne spada u model komunikacije razmenom poruka, jer deljeni objekat nije proces, već struktura kojoj pristupaju procesi ili čije procedure izvršavaju ti procesi, ali svako u svom kontekstu. Iz ugla procesa pošiljaoca i primaoca, međutim, ne mora da bude vidljive razlike između toga da li poruke ostavljaju u bafer/uzimaju iz bafera koji je implementiran kao deljeni objekat ili kao proces. Ovo je upravo tako u jeziku Ada, kako će biti opisano kasnije.

Imenovanje procesa

- Imenovanje procesa uključuje dva aspekta:
 - direkciju ili indirekciju u imenovanju
 - simetriju ili asimetriju u imenovanju.
- *Direktno imenovanje* podrazumeva da učesnik u komunikaciji neposredno imenuje drugog učesnika u komunikaciji (pošiljalac imenuje primaoca ili obratno); na primer:

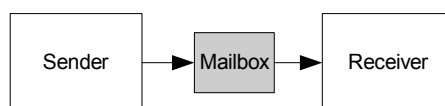
```
send (message, P2) ;           receive (message, P1) ;
```

- Iako jednostavno, direktno imenovanje uvodi tesnu zavisnost između pošiljaoca i primaoca: u kod tih procesa unosi se čvrsto definisan drugi učesnik u komunikaciji, pa su promene u konfiguraciji sistema u kom ovi učesnici razmenjuju poruke teže, odnosno sistem je manje fleksibilan.

- *Indirektno imenovanje* podrazumeva da se učesnici ne imenuju neposredno, već da između pošiljaoca i primaoca postoji neki međumedium koga pošiljalac, odnosno primalac imenuje u konstruktima za slanje i prijem poruke.
- U različitim programskim jezicima, operativnim sistemima i bibliotekama za programiranje postoji mnogo različitih koncepata za međuprocenu komunikaciju razmenom poruka sa indirektnim imenovanjem; u nekim slučajevima oni imaju istu ili vrlo sličnu semantiku, ali različite nazive; u drugim slučajevima je obratno – isti nazivi imenuju semantički različite koncepte. Recimo, pojavljuju se sledeći nazivi: *kanal* (engl. *channel*), *poštansko sanduče* (engl. *mailbox*), *veza* (engl. *link*), *cevovod* (engl. *pipe*), *vrata* (engl. *port*), *kapija* (engl. *gate*) i slično.
- Na primer, posrednik u komunikaciji procesa može biti *poštansko sanduče*, koje zapravo predstavlja bafer kao deljeni objekat za razmenu poruka; pošiljalac i primalac imenuju sanduče u koje stavljaju poruku, odnosno iz koje uzimaju poruku, a ne krajnjeg učesnika u komunikaciji. Analogija – poštanski fah u klasičnoj pošti. Na primer:

```
send(message, MBX);
```

```
receive(message, MBX);
```



- Sličan koncept je koncept *standardnog ulaznog i izlaznog znakovnog* toka nekog programa koji se izvršava kao proces u operativnom sistemu: program može učitavati znak po znak sa svog standardnog ulaznog toka, i izbacivati znak po znak na svoj izlazni tok, ne znajući odakle ti znakovi originalno dolaze, odnosno kuda konačno odlaze. Onaj ko pokreće taj proces (recimo roditeljski proces ili interpreter komandne linije) može da definiše šta će biti taj tok – može biti konzola (tastatura/ekran), tekstualni fajl, ili čak može povezati više takvih procesa u *cevovod* (engl. *pipe*), tako da izlaz jednog procesa bude ulaz drugog procesa (komunikacija je najčešće baferisana); na taj način procesi mogu da rade sekvencijalnu obradu. Ovakav mehanizam podržavaju svi operativni sistemi nalik sistemu Unix.
- Još jedna vrsta koncepta sa indirektnim imenovanjem podrazumeva da proces (ili objekat ili modul) poseduje izlazne ili ulazne tačke (pod nazivom vrata, engl. *port*, ili kapija, engl. *gate*), koji čine njegov interfejs. Proces može primiti poruke imenujući samo neki svoj ulazni port i slati poruke samo imenujući neki svoj izlazni port. Na taj način je proces (ili objekat, modul) potpuno enkapsuliran i izolovan od okruženja, jer okruženje ne može pristupiti njegovoj unutrašnjosti drugačije osim slanja poruka na njegove ulazne portove, ali je i okruženje zaštićeno od procesa, jer proces (modul) ne može uticati na okruženje direktno, nego samo slanjem poruka na svoje izlazne portove. Onaj ko konfiguriše procese, odnosno objekte, recimo onaj ko ih kreira, može da poveže njihove ulazne i izlazne portove *kanalima* ili *vezama* koji prosleđuju poruke sa izlaznog na ulazni port. Ovakav koncept podržavaju jezici ROOM i UML.
- Neki konstrukti indirektnog imenovanja dozvoljavaju da se međumedium konfiguriše tako da ima sledeće veze između pošiljalaca i primalaca:
 - jedan-u-jedan: samo jedan pošiljalac može da šalje i samo jedan primalac da prima;
 - jedan-u-više: jedan pošiljalac može da šalje, a više primalaca da prima;
 - više-u-jedan: više pošiljalaca može da šalje, a samo jedan primalac da prima;
 - više-u-više: više pošiljalaca može da šalje i više primalaca da prima.
- Treba primetiti da upotreba indirektnog imenovanja ne implicira obavezno metod sinhronizacije prilikom slanja: slanje sa indirektnim imenovanjem je često asinhrono, ali i dalje može biti sinhrono (pošiljalac čeka da poruka bude primljena).
- Prednosti indirektnog imenovanja su sledeće:

- procesi/moduli koji učestvuju u komunikaciji su međusobno nezavisni, jer njihov programski kod ne zavisi od drugih učesnika, već samo od međumeditijuma ili sopstvenog interfejsa; ta zavisnost je izmeštena u odgovornost onoga ko učesnike povezuje, odnosno konfiguriše;
- sistem je fleksibilniji, jer se učesnici (procesi, moduli) mogu konfigurisati na različite načine, kako je opisano u navedenim primerima;
- enkapsulacija je bolja, kako je opisano u nekim datim primerima.
- *Simetrično imenovanje* podrazumeva da i pošiljalac i primalac imenuju drugu stranu, makar i indirektno. Pošto po pravilu pošiljalac mora da imenuje odredište poruke (direktno ili indirektno), ovo se tipično odnosi na to da primalac imenuje izvor iz kog želi da primi poruku. To znači da konstrukt za prijem vraća samo poruku pristiglu sa imenovanog izvora (od pošiljaoca ili iz međumeditijuma):

```
send(message, P2);           receive(message, P1);
```

ili:

```
send(message, MBX);          receive(message, MBX);
send(message, port);         receive(message, port);
```

- *Asimetrično imenovanje* podrazumeva da primalac ne imenuje izvor poruke, već prihvata poruku iz bilo kog izvora:

```
send(message, P2);           receive(message);
```

ili:

```
send(message, MBX);          receive(message);
send(message, port);         receive(message);
```

- Asimetrično imenovanje odgovara paradigmi klijent/server, gde primalac, kao server, obrađuje zahteve od bilo kog klijenta, ne praveći razliku između klijenata od kojih poruke stižu.

Struktura poruke

- Programski jezici koji podržavaju međuprocenu komunikaciju pomoću poruka uglavnom dozvoljavaju da poruka bude objekat bilo kog tipa koji taj jezik podržava, pa i korisnički definisanih tipova (klasa), ili eventualno samo nekog iz ograničenog skupa tipova.
- Operativni sistemi koji podržavaju komunikaciju razmenom poruka po pravilu dozvoljavaju samo poruke jednostavne, sekvencijalne strukture: prosti nizovi bajtova (ili eventualno znakova) određene dužine.
- Ukoliko se poruka (objekat) razmenjuje između procesa pisanih na različitim programskim jezicima, uslugom operativnog sistema, npr. preko mreže na udaljeni računar, onda je potrebno izvršiti serijalizaciju objekta izvornog tipa u niz bajtova na mestu slanja (engl. *marshalling*) i deserijalizaciju na mestu prijema (engl. *unmarshalling*). Problem u kompatibilnosti i prenosivosti mogu da predstavljaju različite implementacije formata čak i jednostavnih, skalarnih tipova na različitim arhitekturama, operativnim sistemima i programskim jezicima.
- U rešavanju ovog problema značajnu ulogu igraju standardi, kao što su standardi kodovanja (npr. UTF-8), ali i složeniji formati zapisa (npr. XML, JSON), kao i programske biblioteke koje olakšavaju serijalizaciju i deserijalizaciju objekata tipova iz programskog jezika u te formate.

Randevu u jeziku Ada

- Ada podržava prošireni randevu između procesa po principu klijent/server. Serverski proces deklariše svoje servise koje nudi klijentima kao javne *ulaze* (engl. *entry*) u specifikaciji interfejsa procesa. Svaki ulaz definisan je nazivom i parametrima, koji mogu biti ulazni i izlazni (povratni rezultati), na isti način kao za procedure ili ulaze zaštićenih objekata. Na primer:

```
task type Restaurant is
  entry bookTable (person:in Name; time:in Time; booked:out Boolean);
  -- Other entries
end Restaurant;
```

- Klijentski proces poziva ulaz serverskog procesa u notaciji običnog poziva procedure. Na primer:

```
chezJean : Restaurant;

-- Client task:
task Guest;
task body Guest is
  booked : Boolean := False;
begin
  ...
  chezJean.bookTable("John Smith",nextFriday8pm,booked);
  ...
end Guest;
```

- Serverski proces navodi mesto svoje spremnosti da prihvati poziv servisa pomoću naredbe *accept*. Na primer:

```
task type body Restaurant is
  ...
begin
  ...
  -- Prepare to accept next booking request
  accept bookTable (person:in Name; time:in Time; booked:out Boolean) do
    -- Look up a free table
    booked := ... ; -- Prepare the reply
  end bookTable;
  ...
end Restaurant;
```

- Semantika randevua u jeziku Ada je sledeća. Oba procesa moraju biti spremna da uđu u randevu (i klijent koji izvršava poziv ulaza, i server koji treba da izvrši *accept*). Ukoliko neki proces nije spreman za randevu, onaj drugi mora da čeka. Kada su oba procesa spremna za randevu, ulazni parametri se prosleđuju od klijenta ka serveru, kao kod poziva procedure. Zatim server nastavlja izvršavanje tela naredbe *accept*. Na kraju izvršavanja tela naredbe *accept*, izlazni parametri se vraćaju pozivaocu, a zatim oba procesa nastavljaju svoje izvršavanje konkurentno.
- Treba primetiti da je poziv na strani klijenta apsolutno isti i u slučaju poziva obične procedure, i u slučaju poziva procedure ili ulaza zaštićenog objekta („pasivni“ server), i u slučaju poziva ulaza procesa („aktivni“ server), i to u pogledu:
 - imenovanja i zapisa: klijent imenuje serverski objekat/proces, kao i njegovu prioceduru/ulaz na isti način;
 - prenosa ulaznih i izlaznih argumenata: način definisanja ulaznih i izlaznih argumenata, kao i semantika njihovog prenosa je ista;

- sinhronizacije: poziv je uvek sinhron, potencijalno blokirajući, jer pozivalac nastavlja svoje izvršavanje tek kada se zahtev obradi na serverskoj strani i odgovor od servera vrati, zajedno sa izlaznim argumentima.

Prema tome, klijent ne trpi nikakvu izmenu u notaciji ili semantici izvršavanja ako se implementacija serverske strane promeni iz oblika pasivnog objekta u aktivni proces ili obratno.

- U slučaju da server želi da prihvati *bilo koji* poziv bilo kog od nekoliko ulaza od strane proizvoljnih klijenata u nekom trenutku, moguće je upotrebiti naredbu `select`. Ova naredba omogućuje izbor bilo kog postojećeg poziva za randevu; ukoliko takvih postavljenih, a neopsluženih poziva ima više, bira se jedan od njih nedeterministički. Na primer:

```
task type Restaurant is
  entry bookTable (person:in Name; time:in Time; booked:out Boolean);
  entry enter (person:in Name; canEnter:out Boolean; table:out Number);
  entry exit (person:in Name);
end Restaurant;

task type body Restaurant is
begin
  ...
  loop
    select
      accept bookTable(person:in Name; time:in Time; booked:out Boolean) do
        -- Look up a free table
        booked := ... ; -- Prepare the reply
      end bookTable;
    or
      accept enter(p:in Name; canEnter:out Boolean; table:out Number) do
        -- Look up the table booked by p or a free one
        canEnter := ... ; -- Prepare the reply
        table := ...;
      end enter;
    or
      accept exit(p:in Name) do
        -- Free the table booked by p
      end exit;
    end select;
  end loop;
  ...
end Restaurant;
```

- Naredba `accept` može biti uslovljena *čuvarem* (engl. *guard*) koji predstavlja Bulov izraz iza klauzule `when`. Randevu na toj naredbi `accept` može biti ostvaren samo ukoliko je rezultat ovog izraza `True`. Na primer:

```
task type body Restaurant is
begin
  ...
  loop
    select
      when isOpen =>
        accept bookTable(person:in Name; time:in Time; booked:out Boolean) do
          -- Look up a free table
          booked := ... ; -- Prepare the reply
        end bookTable;
    or
      when isOpen =>
        accept enter(p:in Name; canEnter:out Boolean; table:out Number) do
          -- Look up the table booked by p or a free one
```

```

        canEnter := ... ; -- Prepare the reply
        table := ...;
    end enter;
or
    accept exit(p:in Name) do
        -- Free the table booked by p
    end exit;
end select;
end loop;
...
end Restaurant;

```

- Alternativa u naredbi `select` može biti i naredba `delay`. Ona se bira ukoliko se randevu ne uspostavi na nekoj od `accept` naredbi u definisanom roku. Na primer:

```

task type body Restaurant is
    isOpen : Boolean := True;
begin
    ...
    loop
        select
            when isOpen =>
                accept bookTable(person:in Name; time:in Time; booked:out Boolean) do
                    -- Look up a free table
                    booked := ... ; -- Prepare the reply
                end bookTable;
            or
                when isOpen =>
                    accept enter(p:in Name; canEnter:out Boolean; table:out Number) do
                        -- Look up the table booked by p or a free one
                        canEnter := ... ; -- Prepare the reply
                        table := ...;
                    end enter;
            or
                accept exit(p:in Name) do
                    -- Free the table booked by p
                end exit;
            or
                delay until closingTime;
                isOpen := False;
        end select;
    end loop;
    ...
end Restaurant;

```

- Nešto preciznije, semantika izvršavanja naredbe `select` je sledeća. Svaka od alternativa ove naredbe (alternative su razdvojene operatorom `or`) je omogućena ili ne, na sledeći način:
 - ulaz `accept` je omogućen ako je vrednost izraza-čuvara `True`, i ako ima zahteva za randevu na tom ulazu od strane klijenata;
 - ulaz sa naredbom `delay` je omogućen ako je došao ili prošao apsolutni vremenski trenutak definisan u naredbi `delay until`, odnosno ako je prošao zadati vremenski interval počev od nailaska na naredbu `select`, ako je u pitanju `delay` bez `until`.

Ukoliko omogućenih alternativa nema, serverski proces se suspenduje dok se neka od alternativa ne omogući. Ako ima više omogućenih alternativa, bira se jedna, neterministički. Kada se jedna alternativa prihvati, izvršavanje ulazi u odgovarajuću granu `select` naredbe i po njenom završetku se cela ta naredba završava, dok se merenja vremena u `delay` granama aktivirano po nailasku na `select` zaustavljaju i odbacuju.

Zadaci

5.1. *Randevu na jeziku Ada*

Posmatra se sistem od tri procesa koji predstavljaju pušače i jednog procesa koji predstavlja agenta. Svaki pušač ciklično zavija cigaretu i puši je. Za zavijanje cigarete potrebna su tri sastojka: duvan, papir i šibica. Jedan pušač ima samo duvan, drugi papir, a treći šibice. Agent ima neograničene zalihe sva tri sastojka. Agent postavlja na sto dva sastojka izabrana slučajno. Pušač koji poseduje treći potreban sastojak može tada da uzme ova dva, zavije cigaretu i puši. Kada je taj pušač popužio svoju cigaretu, on javlja agentu da može da postavi nova dva sastojka, a ciklus se potom ponavlja. Realizovati procese pušača i agenta korišćenjem koncepata procesa i randevua u jeziku Ada.

Rešenje

```
task Agent is
  entry takeTobaccoAndPaper ();
  entry takePaperAndMatch ();
  entry takeTobaccoAndMatch ();
  entry finishedSmoking ();
end Agent;

task body Agent is
  tobaccoAvailable : Boolean := False;
  paperAvailable : Boolean := False;
  matchAvailable : Boolean := False;

  procedure putItems is
  begin
    ... -- Randomly select two items and put them on the table
        -- by setting two Boolean variables to True
  end putItems;

begin
  putItems();

  loop

    select
      when tobaccoAvailable and paperAvailable =>
        accept takeTobaccoAndPaper () do
          tobaccoAvailable := False;
          paperAvailable := False;
        end takeTobaccoAndPaper;

    or

      when paperAvailable and matchAvailable =>
        accept takePaperAndMatch () do
          paperAvailable := False;
          matchAvailable := False;
        end takePaperAndMatch;

    or

      when tobaccoAvailable and matchAvailable =>
        accept takeTobaccoAndMatch () do
          tobaccoAvailable := False;
          matchAvailable := False;
        end takeTobaccoAndMatch;
```



```
end select;

accept finishedSmoking () do
    putItems();
end finishedSmoking;

end loop;
end Agent;

task SmokerWithPaper;
task body SmokerWithPaper is
begin
    loop
        Agent.takeTobaccoAndMatch();
        -- Smoke
        Agent.finishedSmoking();
    end loop;
end SmokerWithPaper;
```

Zadaci za samostalan rad

5.2

Korišćenjem koncepata procesa i randevua u jeziku Ada, implementirati sistem koji se sastoji od proizvođača, potrošača i ograničenog bafera.

5.3

Korišćenjem realizovanih koncepata iz školskog jezgra, koncipirati podršku za randevu u jeziku C++.

Pristup deljenim resursima

- Veliki deo logike konkurentnih programa leži u međusobnom nadmetanju procesa za pristup deljenim resursima; deljeni resursi mogu biti logički i fizički, kao što su deljeni podaci, baferi, memorijski prostor, eksterni uređaji, semafori, datoteke i slično.
- *Kooperativni* (engl. *cooperating*) procesi su oni koji međusobno sarađuju, tj. sinhronizuju se i komuniciraju, odnosno razmenjuju informacije potrebne za njihovu obradu, ili pristupaju deljenim resursima (logičkim ili fizičkim).
- Čak i kada su procesi logički nezavisni, odnosno napravljeni tako da međusobno ne razmenjuju informacije vezane za logiku njihove obrade, oni često pristupaju resursima za koje možda nisu ni svesni da ih dele sa drugim procesima (npr. zato što su pod kontrolom izvršnog okruženja jezika ili operativnog sistema), ili su prosto potrebni za njihovu obradu. Tada takvi procesi postaju kooperativni, jer moraju da se sinhronizuju u slučaju da ti deljeni resursi nisu deljivi (tj. nije dozvoljena njihova uporedna upotreba od strane različitih procesa). Na jednom računaru praktično nema nezavisnih procesa, jer svi oni konkurišu za iste resurse (procesor, memoriju, uređaje, fajlove itd.), kojima upravlja operativni sistem.
- Kao što je do sada pokazano, implementacija pristupa deljenim resursima u programu obično podrazumeva postojanje nekog agenta za kontrolu pristupa resursu. Ukoliko je taj agent *pasivan* (nema svoj tok kontrole), naziva se *zaštićenim* (engl. *protected*) ili *sinhronizovanim* (engl. *synchronized*) objektom. Ako je taj agent *aktivan* (ima svoj tok kontrole), on se obično naziva *serverom* (engl. *server*).
- Ovo poglavlje najpre prikazuje nekoliko čestih modela kontrole resursa koji se koriste u konkurentnom programiranju, a zatim diskutuje probleme koji mogu da nastupe kod nadmetanja za deljene resurse.

Modeli pristupa deljenim resursima

- Jedan od standardnih problema kontrole konkurentnosti jeste problem ograničenog bafera koji je do sada bio detaljno analiziran. Međutim, pored njega, još nekoliko problema se često koristi u teoriji i praksi konkurentnog programiranja za analizu i demonstraciju koncepata konkurentnih jezika i konkurentnih algoritama, kao obrasci (modeli) po kojima konkurentni procesi u praksi često pristupaju deljenim resursima. Ovde će biti razmatrana dva takva problema, problem *čitalaca i pisaca* (engl. *readers-writers*) i problem *filozofa koji večeraju* (engl. *dining philosophers*).

Čitaoci i pisci

- Koncept monitora obezbeđuje međusobno isključenje pristupa konkurentnih procesa do deljenog resursa, uz eventualnu uslovnu sinhronizaciju. Međutim, koncept potpunog međusobnog isključenja kod monitora ponekad predstavlja suviše restriktivnu politiku koja smanjuje konkurentnost programa.
- Naime, veoma često se operacije nad deljenim resursom mogu svrstati u dve grupe, i to operacije koje:
 - samo čitaju deljene podatke, odnosno ne menjaju stanje resursa (operacije čitanja)
 - upisuju u deljene podatke, tj. menjaju stanje resursa (operacije upisa).

- Koncept monitora ne dozvoljava nikakvu konkurentnost ovih operacija. Međutim, konkurentnost se može povećati ukoliko se dozvoli da:
 - proizvoljno mnogo procesa izvršava operacije čitanja (tzv. *čitaoci*, engl. *readers*), jer oni ne smetaju jedan drugom, pošto ne menjaju deljene podatke;
 - najviše jedan proces izvršava operaciju upisa (tzv. *pisac*, engl. *writer*), međusobno isključivo sa drugim piscima, ali i sa čitaocima (jer pisac može da „smeta“ čitaocima i drugim piscima, kako je ranije pokazano).
- Na taj način, deljenom resursu u datom trenutku može pristupati ili samo jedan pisac, ili više čitalaca, ali ne istovremeno i jedni i drugi. Zato se ovaj koncept naziva *više čitalaca-jedan pisac* (engl. *multiple readers-single writer*).
- Koncept zaštićenog objekta u jeziku Ada inherentno podržava protokol više čitalaca i jednog pisca. Međutim, ukoliko taj koncept nije direktno podržan u jeziku, on se mora realizovati pomoću drugih koncepata.
- Postoje različite varijante ove šeme koje se razlikuju u pogledu prioriteta koji se daje procesima koji čekaju na pristup resursu. Ovde je realizovana sledeća varijanta: prioritet imaju pisci koji čekaju, tj. čim postoji pisac koji čeka, svi novi čitaoci biće zaustavljeni sve dok svi pisci ne završe.
- Implementacija opisane varijante korišćenjem monitora je da monitor poseduje četiri operacije: `startRead`, `stopRead`, `startWrite` i `stopWrite`. Čitaoci i pisci, odnosno operacije čitanja i upisa moraju da budu strukturirani na sledeći način:

Reader:

```
startRead();
... // Read data structure
stopRead();
```

Writer:

```
startWrite();
... // Write data structure
stopWrite();
```

- Na jeziku Java, monitor može da izgleda ovako:

```
public class ReadersWriters {

    private int readers = 0;
    private int waitingWriters = 0;
    private boolean writing = false;

    public synchronized void startWrite () throws InterruptedException {
        while (readers>0 || writing) {
            waitingWriters++;
            wait();
            waitingWriters--;
        }
        writing = true;
    }

    public synchronized void stopWrite () {
        writing = false;
        notifyAll();
    }

    public synchronized void startRead () throws InterruptedException {
        while (writing || waitingWriters>0) wait();
        readers++;
    }

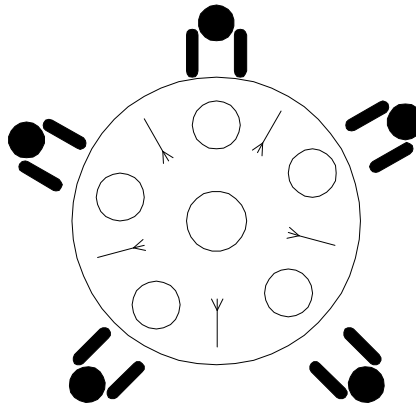
    public synchronized void stopRead () {
        reader--;
        if (readers==0) notifyAll();
    }
}
```

```
}
}
```

- Problem sa ovim rešenjem je što se pri svakoj značajnoj promeni (pisac ili poslednji čitalac završava svoju operaciju) svi blokirani procesi deblokiraju, pa svi moraju ponovo da izračunaju svoje uslove. Iako se svi deblokiraju, mnogi od njih će ponovo biti blokirani, pa je rešenje neefikasno.

Filozofi koji večeraju

- Jedan od najstarijih problema koji se najčešće koriste za proveru izražajnosti i upotrebljivosti koncepta konkurentnog programiranja, kao i korektnosti konkurentnih algoritama, jeste problem *filozofa koji večeraju* (engl. *dining philosophers*). Predložio ga je E. Dijsktra.
- Pet filozofa sedi za okruglim stolom na kom se nalazi posuda sa špagetama. Ispred svakog filozofa je njegov tanjir, a sa leve i desne strane tanjira, odnosno između svaka dva susedna tanjira je samo po jedna viljuška. Svaki filozof naizmenično razmišlja neko vreme, a onda od razmišljanja ogladni i želi da se posluži špagetama i jede. Pretpostavlja se da su svakom filozofu, da bi se poslužio, potrebne dve viljuške, kao i da može da koristi samo one koje se nalaze levo i desno od njegovog tanjira. Ako je jedna od njih zauzeta, on mora da čeka. Kad završi sa jelom, filozof spušta obe viljuške na sto i nastavlja da razmišlja. Posle nekog vremena, filozof ogladni i ponovo pokušava da jede i tako ciklično. Potrebno je definisati protokol (pravila ponašanja, algoritam) koji će obezbediti ovakvo ponašanje filozofa i pristup do viljušaka. U principu, filozofe predstavljaju procesi, a viljuške su deljeni resursi.



Problemi nadmetanja za deljene resurse

- Da bi konkurentan program bio ispravan, mora da zadovolji sledeće uslove:
 - logička ispravnost njegovog rezultata ne sme da zavisi od toga kojim redosledom se nezavisne akcije uporednih procesa izvršavaju (osim ako to nije ograničeno sinhronizacijom), odnosno od toga na kakvoj platformi se uporedni procesi izvršavaju;
 - konkurentan program mora da poseduje svojstvo *živosti* (engl. *liveness*), tj. *napredovanja* (engl. *progress*): sve što je programom predviđeno da se u konačnom vremenu dogodi ili završi (npr. da se dobije pristup deljenom resursu), mora i da se dogodi u konačnom vremenu; u konkurentnom programiranju bitno je samo da to vreme bude konačno (ograničeno bilo kojom veličinom), dok je u RT programiranju bitna i apsolutna veličina tog vremena (vremenski rok).

- Neispravna logika konkurentnih programa može da dovede do različitih *pogrešnih stanja* programa (engl. *error condition*), od kojih su najvažniji:
 - *utrkivanje* (engl. *race condition*)
 - *izgladnjivanje* (engl. *starvation*) ili *neograničeno odlaganje* (engl. *indefinite postponement*)
 - *živo blokiranje* (engl. *livelock*)
 - *mrtvo (kružno) blokiranje* (engl. *deadlock*).
- Postojanje prvog problema, utrkivanja, znači da nije ispunjen prvi gore navedeni uslov (logička korektnost u svim izvršavanjima), odnosno da nije obezbeđena potrebna sinhronizacija. Postojanje nekog od ostalih problema znači da nije ispunjen drugi uslov (živost). Da bi bio logički korektan, konkurentan program ne sme da dozvoli mogućnost za nastajanje bilo kog od ovih pogrešnih stanja.

Utrkivanje

- Pretpostavimo da jezik, okruženje ili operativni sistem podržava dve jednostavne primitive:
 - `suspend`: bezuslovno suspenduje (blokira) pozivajući proces;
 - `resume`: bezuslovno deblokira imenovani proces, ukoliko je on suspendovan.
- Pretpostavimo da uslovnu sinhronizaciju između dva procesa P1 (koji čeka na uslov) i P2 (koji signalizira ispunjenje uslova) treba obaviti pomoću ovih primitiva i deljene promenljive `flag` tipa `Boolean` na sledeći način:

```
flag : Boolean := false;
```

```
Process P1:
```

```
...
if not flag then suspend;
flag := false;
...
```

```
Process P2:
```

```
...
flag := true;
P1.resume;
...
```

- Problem ovog rešenja je u tome što može da se dogodi sledeći scenario: proces P1 ispita vrednost promenljive `flag` (koja je `false`), a odmah potom se dogodi preuzimanje i procesor dodeli procesu P2 (ili se on može izvršavati paralelno na drugom procesoru). Proces P2 postavi `flag` na `true` i izvrši `resume` procesa P1 (bez efekta, jer P1 još nije suspendovan). Kada P1 nastavi izvršavanje, on će biti nekorektno suspendovan i čekati na uslovu koji je ispunjen. Tako ova dva procesa ispadaju iz sinhronizacije.
- Ovakva neispravna situacija naziva se *utrkivanje* (engl. *race condition*) i posledica je toga što je neki proces izvršio akciju koju nije smeo da izvrši, pa su se procesi „pretekli“. U opštem slučaju, utrkivanje znači da nije obezbeđena potrebna sinhronizacija (npr. međusobno isključenje).
- U navedenom primeru, utrkivanje je nastalo tako što se odluka o promeni stanja procesa (suspenciji) donosi na osnovu ispitivanja vrednosti deljene promenljive, pri čemu ta dva koraka nisu nedeljiva, pa može doći do preuzimanja, tj. "utrkivanja" od strane drugog procesa koji pristupa istoj deljenoj promenljivoj. Ovo je karakteristična situacija na koju u praksi treba obratiti pažnju: odluka o nekoj promeni (upisu podatka, suspenciji procesa ili nekoj drugoj akciji) se donosi na osnovu nekog uslova koji uključuje deljene podatke, a te dve akcije nisu nedeljive, pa u međuvremenu uslov može biti promenjen.
- Sličan problem bi postojao u implementaciji ograničenog bafera u školskom jezgru, ukoliko operacije `signal` i `wait` koje obezbeđuju uslovnu sinhronizaciju ne bi bile nedeljive:

```

void MsgQueue::send (CollectionElement* ce) {
    Mutex dummy(&mutex);
    if (rep.isFull()) {
        mutex.signal();          // Race condition!
        notFull.wait();
        mutex.wait();
    }
    rep.put(ce);
    if (notEmpty.value() < 0) notEmpty.signal();
}

```

```

Object* MsgQueue::receive () {
    Mutex dummy(&mutex);
    if (rep.isEmpty()) {
        mutex.signal();          // Race condition!
        notEmpty.wait();
        mutex.wait();
    }
    Object* temp=rep.get();
    if (notFull.value() < 0) notFull.signal();
    return temp;
}

```

Mrtvo blokiranje

- Posmatrajmo sledeći algoritam po kome postupa svaki filozof u primeru filozofa koji večeraju. Pretpostavlja se da su viljuške resursi za koje je obezbeđena ekskluzivnost pristupa, tako da se proces filozofa blokira ukoliko je viljuška zauzeta. Svaki filozof najpre uzima svoju levu viljušku, ako je slobodna, a potom desnu, ako je ona slobodna:

```

task type Philosopher
loop
    think;
    take left fork;
    take right fork;
    eat;
    release left fork;
    release right fork;
end;
end;

```

- Ovaj algoritam može biti implementiran korišćenjem semafora na sledeći način:

```
Semaphore forks[5];
```

```

class Philosopher : public Thread {
public:
    Philosopher (int orderNumber)
        : myNum(orderNumber), left(orderNumber), right((orderNumber+1)%5) {}

protected:
    virtual void run ();

    void eat();
    void think();

private:
    int myNum, left, right;
};

```

```

void Philosopher::run () {
    while (true) {
        think();
        forks[left].wait();
        forks[right].wait();
        eat();
        forks[left].signal();
        forks[right].signal();
    }
}

...
Philosopher p0(0), p1(1), p2(2), p3(3), p4(4);
p0.start(); p1.start(); p2.start(); p3.start(); p4.start();

```

- Problem ovog rešenja je u tome što može nastati sledeći scenario: svi filozofi uporedo uzmu po jednu viljušku, svaki svoju levu, a onda se blokiraju prilikom pristupa do svoje desne viljuške, jer su sve viljuške zauzete. Tako svi procesi ostaju trajno kružno blokirani.
- Ovakav neregularan uslov koji nastaje tako što se grupa procesa koji konkurišu za deljene resurse međusobno kružno blokiraju, naziva se *mrtvo* (ili *kružno*) *blokiranje* (engl. *deadlock*).
- U opštem slučaju, mrtvo blokiranje nastaje tako što se grupa procesa nadmeće za ograničene resurse, pri čemu proces $P1$ drži ekskluzivan pristup do resursa $R1$ i pri tom čeka blokiran da se oslobodi resurs $R2$, proces $P2$ drži ekskluzivan pristup do resursa $R2$ i pri tom čeka blokiran da se oslobodi resurs $R3$, itd., proces Pn drži ekskluzivan pristup do resursa Rn i pri tom čeka blokiran da se oslobodi resurs $R1$ (n je prirodan broj, uključujući i 1). Tako procesi ostaju neograničeno suspendovani u cikličnom lancu blokiranja.
- Mrtvo blokiranje je jedan od najtežih problema koji mogu da se pojave u konkurentnim programima.
- Postoje četiri neophodna uslova za nastanak mrtvog blokiranja:
 - *međusobno isključenje* (engl. *mutual exclusion*): mora postojati resurs do kog je pristup ekskluzivan, odnosno takav da samo jedan proces može koristiti resurs u jednom trenutku; drugim rečima, resurs nije deljiv ili je pristup do nega ograničen;
 - *držanje i čekanje* (engl. *hold and wait*): mora postojati proces koji drži zauzete neke resurse i pri tom čeka na neke resurse;
 - *nema preuzimanja resursa* (engl. *no preemption*): samo proces koji je zauzeo resurs može ga osloboditi, resurs se ne može preoteti procesu koji ga drži;
 - *kružno čekanje* (engl. *circular wait*): mora postojati cikličan lanac procesa tako da svaki proces u lancu drži resurs koga traži naredni proces u lancu.
- Ovi uslovi su neophodni: mrtva blokada ne može nastupiti ako neki od ovih uslova nije ispunjen. Treba primetiti da ovi uslovi nisu nezavisni (npr. „kružno čekanje“ uključuje „držanje i čekanje“), pa ni minimalni, ali su ovako formulisani jer se određene tehnike sprečavanja mrtve blokade oslanjaju na ukidanje nekog od ovih uslova.
- Da bi program bio pouzdan, mora da bude zaštićen od mrtvog blokiranja (engl. *deadlock-free*). Postoji nekoliko pristupa rešavanju ovog problema:
 - *sprečavanje mrtvog blokiranja* (engl. *deadlock prevention*);
 - *izbegavanje mrtvog blokiranja* (engl. *deadlock avoidance*);
 - *detekcija i oporavak od mrtvog blokiranja* (engl. *deadlock detection and recovery*).

Sprečavanje mrtvog blokiranja

- Sprečavanje mrtvog blokiranja pretpostavlja da se prilikom izvedbe samog sistema, njegovom konstrukcijom obezbedi da neki od neophodnih uslova nastanka mrtve blokade nije ispunjen. Tako je garantovano da mrtvo blokiranje ne može nastati.
- Za svaki od navedenih neophodnih uslova postoje određene tehnike koje se u nekim situacijama mogu primeniti, čime se sprečava mrtvo blokiranje. Nijedna od tih tehnika nije primenjiva uvek, u svakoj situaciji, ali su situacije u praksi u kojima se neka od ovih tehnika može primeniti relativno česte. Posebno su česte situacije u kojima se neka od ovih tehnika može primeniti na deo sistema, tj. samo na neke procese ili neke resurse, i tada je garantovano da barem ti procesi ili resursi ne mogu učestvovati u mrtvoj blokadi. U svakom slučaju, primena neke od ovih tehnika zavisi od konkretnog programa, odnosno situacije.
- *Međusobno isključenje.* Ako su resursi deljivi, odnosno takvi da je procesima dozvoljen upredni pristup do njih, onda oni ne mogu da izazovu mrtvo blokiranje. Ako je neki resurs deljiv, onda on sigurno ne može učestvovati u mrtvoj blokadi. Na primer, ako procesi samo rade operacije čitanja nad deljenim resursom, nema potrebe tražiti njihovo međusobno isključenje, što može da utiče i na ceo sistem ako se time raskida mogući kružni lanac, ali u svakom slučaju se povećava konkurentnost. Zato ne treba uvoditi preteranu sinhronizaciju kod pristupa deljenim resursima, već dozvoliti uporedni pristup kad god je to moguće. Nažalost, to često nije moguće.
- *Držanje i čekanje.* Neke tehnike koje ukidaju ovaj uslov su sledeće:
 - Konstrukcijom procesa ili njegovim restrukturiranjem obezbediti da proces nikada ne koristi više resursa istovremeno, već najviše jedan (jedan po jedan, tj. prethodni oslobodi pre nego što sledeći zatraži).
 - Konstrukcijom procesa ili njegovim restrukturiranjem obezbediti da proces uzima sve resurse koji su mu istovremeno potrebni zajedno (videti primer filozofa u kasnijem odeljku o izgladnjivanju), tj. da ne uzima jedan po jedan tako što traži naredni (i potencijalno se blokira) ako je već zauzeo drugi resurs.
 - Konstrukcijom procesa ili njegovim restrukturiranjem obezbediti da kada taj proces traži nov resurs i ne može da ga dobije, oslobodi resurse koje drži (videti primer filozofa u kasnijem odeljku o živoj blokadi).
 - Ograničiti vreme čekanja na resursu: ako proces čeka blokiran na resursu, ili ako to radi onda kada već drži neke resurse, odbiti mu zahtev za alokacijom resursa nakon isteka nekog maksimalnog vremena čekanja; proces to onda tretira kao izuzetak.
- *Nema preuzimanja.* Neke tehnike koje ukidaju ovaj uslov su sledeće:
 - Ako proces traži resurs, a ne može da ga dobije, preoteti taj resurs od procesa koji ga drži i dodeliti ga procesu koji ga traži.
 - Ako proces traži resurs, a ne može da ga dobije, preoteti tom procesu sve resurse koje on već drži.

Problem sa ovim tehnikama je u tome što je potrebno ili sačuvati kontekst korišćenja resursa od strane procesa kome se taj resurs preotima, kako bi mu kasnije bio vraćen na korišćenje, uz restauraciju konteksta, ili procesu kom se resurs preotima signalizirati izuzetak ili ga ugasiti i ponovo pokrenuti. Čuvanje i restauracija konteksta može biti režijski skupa, komplikovana ili neizvodljiva. Signalizacija izuzetka ili restartovanje procesa može da uzrokuje mnogo ponovnih pokušaja ili čak izgladnjivanje procesa.

- *Kružno čekanje.* Da bi se sprečilo kružno čekanje, moguće je uvesti totalno uređenje svih resursa; formalno, svakom resursu R_i dodeljuje se vrednost $F(R_i)$ iz skupa za koji je definisana relacija totalnog uređenja $<$. Najjednostavnije, to može biti prosto numeracija resursa ($F(R_i)$ je jedinstven redni broj resursa R_i). Funkcija F može da se odredi i prema

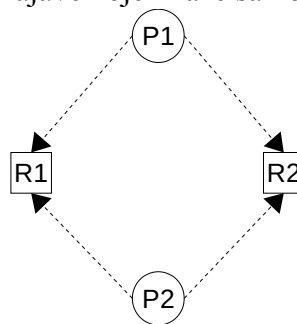
načinu korišćenja resursa. Kružno čekanje se može sprečiti na sledeće načine (izvesti dokaz da je tada kružno čekanje nemoguće):

- Konstrukcijom procesa ili njegovim restrukturiranjem obezbediti da proces koji je zauzeo neke resurse sme da traži, ili mu se dozvoljava da zauzme nov resurs R_j samo ukoliko je $F(R_i) < F(R_j)$ za svaki resurs R_i koji taj proces drži.
- Konstrukcijom procesa ili njegovim restrukturiranjem obezbediti da proces može da zauzme novi resurs R_j samo ukoliko prethodno oslobodi sve resurse R_i za koje je $F(R_i) > F(R_j)$.
- Drugi način za sprečavanje kružnog blokiranja jeste da se konkurentan program najpre konstruiše tako da je kružno blokiranje nemoguće, a onda program formalno analizira i ovo svojstvo i dokaže. Takva formalna analiza se po pravilu svodi na ispitivanje skupa svih diskretnih stanja u koja program može da uđe, pri čemu se stanjima smatraju pozicije izvršavanja procesa u diskretnim tačkama sinhronizacije. Nažalost, ovo je često veoma teško ili praktično nemoguće izvesti za realne sisteme zbog ogromnog broja ("eksplozije") mogućih stanja. Izvodljivost takve analize zavisi i od od modela konkurentnosti u programskom jeziku.

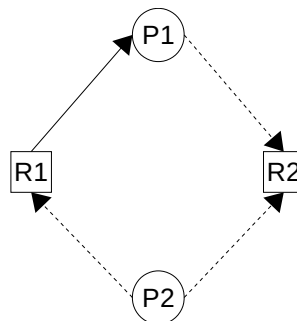
Izbegavanje mrtvog blokiranja

- Izbegavanje mrtve blokade podrazumeva primenu algoritma koji u vreme izvršavanja sistema, dinamički, kontroliše alokaciju resursa tako da do mrtve blokade sigurno ne može da dođe, iako su konstrukcijom sistema ispunjeni svi neophodni uslovi za mrtvu blokadu.
- Algoritmi izbegavanja mrtve blokade obezbeđuju da se sistem stalno drži u tzv. *bezbednom stanju* (engl. *safe state*); to se čini tako što se čak i ako postoje slobodni resursi, ne odobrava korišćenje tih resursa procesu koji ih je zatražio, ako bi to potencijalno moglo da odvede sistem u mrtvu blokadu, a da to sistem više ne može da izbegne.
- Da bi ovi algoritmi bili sprovodivi, sistem koji upravlja resursima mora unapred znati koliko kojih resursa svaki proces može najviše istovremeno da koristi, odnosno da traži tokom svog izvršavanja. Prema tome, svaki proces mora da *najavi* korišćenje svih resursa koje će potencijalno tražiti, kako bi sistem mogao da sprovodi algoritme izbegavanja mrtve blokade. Ova najava uključuje potencijalno korišćenje maksimalne količine resursa, u najgorem slučaju, što ne mora uvek biti i ostvareno u konkretnom izvršavanju. Proces mora najaviti korišćenje svih resursa najkasnije pre nego što zatraži prvi resurs.
- Stanje sistema u nekom trenutku izvršavanja, u smislu alokacije resursa, definiše se kao skup sledećih podataka:
 - skup raspoloživih (slobodnih) resursa
 - skup alociranih resursa za svaki proces
 - skup maksimalno zahtevanih (najavljenih) resursa za svaki proces.
- Sistem je u *bezbednom stanju* ukoliko se može pronaći način, tj. redosled alokacije resursa od strane procesa, takav da se svakom procesu može dodeliti još onoliko resursa koliko on maksimalno može da zahteva (i koliko je najavio), a da se pri tome ipak izbegne mrtvo blokiranje.
- Jedan algoritam izbegavanja mrtve blokade zasniva se na *grafu alokacije resursa* (engl. *resource allocation graph*) – strukturi podataka koja vodi evidenciju o stanju zauzeća resursa. Ovaj pristup pretpostavlja da je svaka instanca resursa entitet za sebe, tj. da proces eksplicitno identifikuje konkretnu instancu resursa koju traži. Graf alokacije se formira na sledeći način:
 - čvorovi u grafu su procesi (označeni kružnim čvorovima na slikama ovde) i resursi (označeni kvadratnim čvorovima);

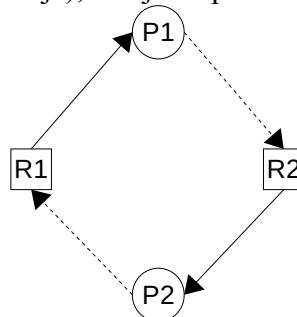
- kada proces P_i najavi korišćenje resursa R_j , uvodi se usmerena grana najave (označena isprekidanom linijom) od čvora P_i do čvora R_j ;
 - kada proces P_i zatraži korišćenje resursa R_j , uvodi se usmerena grana potražnje (označena punom linijom) od čvora P_i do čvora R_j ;
 - kada proces P_i dobije na korišćenje resurs R_j , grana potražnje obrće smer i pretvara se u granu korišćenja (označena punom linijom) od čvora R_j do čvora P_i ;
 - kada proces P_i oslobodi resurs R_j , grana zauzeća zamenjuje se granom najave (jer taj proces može ponovo zatražiti korišćenje tog istog resursa).
- Stanje sistema je bezbedno ako i samo ako opisani graf nema petlju (uzimajući u obzir sve tipove grana i njihovo usmerenje).
 - Na primer, neka u sistemu postoje samo dva procesa i dva resursa, pri čemu su oba procesa najavila korišćenje oba resursa. Početno stanje je dato sledećim grafom i svakako je bezbedno (i u opštem slučaju početno stanje je sigurno bezbedno jer ne poseduje petlju, pošto su sve grane samo grane najave koje izlaze samo iz procesa, a ulaze samo u resurse):



Ako sada proces P_1 zatraži korišćenje resursa R_1 , koji je slobodan, a sistem mu taj resurs dodeli, grana najave zamenjuje se granom zauzeća (menja smer) i graf izgleda ovako, a stanje je još uvek bezbedno:



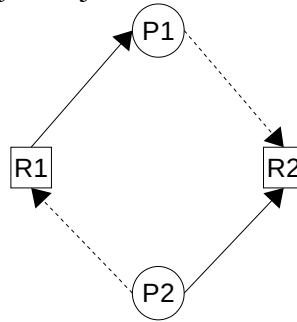
Ako u ovom stanju proces P_2 zatraži korišćenje resursa R_2 , koji je slobodan, i sistem mu taj resurs alocira (dozvoli korišćenje), stanje će postati nebezbedno, jer će sadržati petlju:



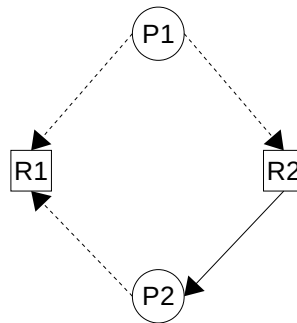
Ovo stanje *nije* stanje mrtve blokade (jer su oba procesa aktivna, nisu blokirana i ne postoji cikličan lanac držanja i čekanja), ali je nebezbedno, jer se može dogovoditi da

procesu traže nove resurse, u skladu sa svojom najavom, a što sistem ne može da spreči, a da ti resursi ne mogu da budu dodeljeni jer su zauzeti, i da se time procesi upetljaju u mrtvu blokadu. Za dato stanje, mrtva blokada može nastati ako i P_1 i P_2 (pošto su oba aktivna) zatraže drugi najavljeni resurs, što mogu da urade a da sistem to ne može da izbegne. Prema tome, nebezbedno stanje je stanje koje *može da odvede sistem* u stanje mrtve blokade, a da sistem to više ne može da izbegne.

Zbog toga sistem neće dozvoliti alokaciju resursa R_2 procesu P_2 , čak i ako je on slobodan, već će ga blokirati sve dok se stanje sistema ne promeni tako da bi alokacija traženog resursa zadržala sistem u bezbednom stanju, a to znači do onda kada neki od procesa oslobodi neki resurs. Tako stanje ostaje ovakvo:



Dakle, proces P_2 i dalje čeka na resurs R_2 koji je slobodan. Stanje će se promeniti kada proces P_1 oslobodi resurs R_1 . Tada P_2 može dobiti resurs R_2 , kada je stanje ponovo bezbedno:



- Treba primetiti da su grane najave i grane potražnje istog smera, upravo zato da bi odslikale činjenicu da sistem ne može da spreči aktivan proces da zatraži resurs koji je najavio, pa ta potražnja ne menja prirodu stanja sistema. Zato stanje mora biti bezbedno (bez postojanja petlje) i pre nego što taj proces zatraži resurs. Grana menja smer alokacijom resursa, koju sistem može da spreči, kao i oslobađanjem resursa, i zato se ispitivanje postojanja petlje, odnosno aktivacija algoritma izbegavanja mrtve blokade sprovodi kada neki proces može da dobije slobodan resurs i kada proces oslobađa resurs.
- Kada postoji više identičnih instanci istog tipa resursa, proces može zatražiti (u jednom zahtevu), jednu ili više instanci nekog tipa resursa (ili čak i više tipova resursa). Pretpostavlja se da su sve instance istog tipa identične i ravnopravne, pa se procesu može dodeliti bilo koja slobodna instanca traženog tipa. (Sistem sa po jednom instancom svakog tipa resursa je samo specijalan slučaj ovog opštijeg modela.)
- Za ovakve situacije primenjiv je tzv. *bankarev algoritam* (engl. *Banker's algorithm*). Ovaj algoritam proverava da li je dato stanje bezbedno ili nije. Kada neki proces traži neke resurse, i ako su ti resursi slobodni, sistem ovim algoritmom proverava da li bi alokacija tih resursa odvela sistem u bezbedno ili nebezbedno stanje. Ako bi ga odvela u nebezbedno stanje, alokacija se ne odobrava, a proces se blokira dok se stanje ne promeni i alokacija postane bezbedna (kada neki drugi proces oslobodi neke svoje resurse).

- Ovaj algoritam ispituje da li je stanje bezbedno pronalaženjem tzv. *sigurne sekvence* (engl. *safe sequence*): zamišljenog redosleda izvršavanja procesa, jednog po jednog, tako da svaki proces može da zadovolji svoju maksimalnu najavljenju potražnju sa resursima koji su ukupno slobodni nakon što su svi procesi u sekvenci pre njega fiktivno završili svoje izvršavanje i oslobodili resurse.
- Stanje je bezbedno ako i samo ako sigurna sekvenca postoji.
- Na primer, neka u sistemu postoji 9 identičnih jedinica istog tipa nekog resursa i tri procesa: P_1 , koji je najavio, tj. maksimalno zahteva 4 jedinice tog resursa, P_2 koji maksimalno zahteva 6 jedinica i P_3 koji zahteva 8 jedinica. Neka u datom trenutku proces P_1 zauzima 2, P_2 zauzima 3, a P_3 zauzima 2 jedinice resursa. Stanje sistema dato je na sledeći način:

Proces	Zauzeo	Traži
P_1	2	4
P_2	3	6
P_3	2	8
Ukupno zauzeto: 7		
Slobodnih: 2		

Ovo stanje je bezbedno, jer postoji sigurna sekvenca izvršavanja $\langle P_1, P_2, P_3 \rangle$ koja omogućava da svi procesi završe svoje izvršavanje tim redom: P_1 može da zadovolji svoju maksimalnu potražnju od 4 resursa sa 2 koja već zauzima i 2 trenutno slobodna, završi i oslobodi sva 4 resursa; potom P_2 može da zadovolji svoju maksimalnu potražnju sa još 3 resursa od 4 slobodna, i konačno, P_3 može da dobije još 6 slobodnih resursa nakon što je P_2 oslobodio sve resurse koje drži.

Ako u ovom stanju, u nekom narednom trenutku proces P_1 zahteva i dobije još jednu jedinicu (jer je ona raspoloživa), stanje postaje:

Proces	Zauzeo	Traži
P_1	3	4
P_2	3	6
P_3	2	8
Ukupno zauzeto: 8		
Slobodnih: 1		

Ovo stanje više nije bezbedno, jer sa slobodnim resursima samo P_1 može da zadovolji svoju maksimalnu potražnju. Da je stanje nebezbedno, tj. da može odvesti u stanje mrtve blokade pokazuje to da i P_2 i P_3 mogu da zatraže resurse do svoje maksimalne potražnje (3, odnosno 6), što sistem ne može da spreči, i da nijedan ne može da ih dobije (jer ih nema dovoljno slobodnih), pa procesi tada ostaju trajno blokirani u mrtvoj blokadi.

Prema tome, ukoliko sistem ima ugrađen algoritam za izbegavanje mrtvog blokiranja, on će odbiti zahtev procesa P_1 i blokirati ga dok tražena alokacija ne bude bezbedna. To se utvrđuje nakon što neki proces oslobodi resurse i time promeni stanje sistema.

- Kada postoji više tipova resursa, svaki tip posmatra se na isti način, odvojeno, odnosno broj alociranih i najavljenih resursa svakog procesa, kao i broj slobodnih resursa posmatra se kao vektor.
- Prema tome, ako se posmatra skup svih mogućih stanja zauzeća resursa (potencijalno ogroman), i podskupovi bezbednih i nebezbednih stanja, važi sledeće:
 - svako stanje je ili bezbedno ili nebezbedno, tj. podskupovi bezbednih i nebezbednih stanja dele skup svih stanja na dve disjunktne particije;
 - stanje mrtve blokade je nebezbedno stanje, ali nije svako nebezbedno stanje stanje mrtve blokade: skup stanja sa mrtvom blokadom je podskup skupa nebezbednih stanja;

- iz stanja mrtve blokade sistem više ne može preći u bezbedno stanje niti u nebezbedno stanje koje nije stanje mrtve blokade;
- iz nebezbednog stanja sistem *može* da „sklizne“ u stanje mrtve blokade, a da se to više ne može izbeći, ali se to *ne mora* uvek dogoditi, jer procesi ne moraju da traže resurse u poretku koji bi odveo u mrtvu blokadu; drugim rečima, iz nebezbednog stanja koje nije stanje mrtve blokade ima mogućnosti povratka u bezbedno stanje;
- početno stanje je bezbedno stanje ako i samo ako svaki proces ima maksimalnu potražnju koja se može zadovoljiti ukupnom količinom resursa; krajnje (konačno) stanje, kada su svi procesi završili, je trivijalno bezbedno stanje;
- iz svakog bezbednog stanja može se preći u neko drugo bezbedno stanje, i direktno ili indirektno u konačno stanje;
- sistem sprovodi algoritam izbegavanja mrtve blokade tako što ne dozvoljava alokaciju resursa procesu čak i ako je on slobodan, ako bi to sistem prevelo u nebezbedno stanje; na taj način sistem izbegava mrtvu blokadu sve do kraja izvršavanja svih procesa.
- Problem svih algoritama izbegavanja mrtvog blokiranja je u sledećem:
 - strukture podataka za evidenciju stanja alokacije resursa zahtevaju memorijski prostor;
 - algoritmi održavanja ovih struktura zahtevaju režijsko vreme pri svakoj promeni;
 - algoritam izbegavanja mrtve blokade (ispitivanje postojanja petlje u grafu ili bankarev algoritam traženja sigurne sekvence) zahteva režijsko vreme u situacijama kada se aktivira (kada postoje slobodni resursi koje proces traži i kada proces oslobađa resurse);
 - suspenzija procesa čak i kada ima slobodnih resursa, sa ciljem izbegavanja mrtve blokade, može nepotrebno da koči procese i usporava njihovo napredovanje; ovo je utoliko značajnije ukoliko procesi imaju preveliku maksimalnu potražnju u najgorem slučaju, odnosno ako se ta procena najgoreg slučaja uradi pesimistično i restriktivno (konzervativno), a u stvarnom izvršavanju je količina alociranih resursa značajno manja;
 - mora biti unapred poznata maksimalna potražnja resursa, što nije uvek moguće.

Detekcija i oporavak od mrtvog blokiranja

- Kao što je već objašnjeno, tehnike sprečavanja mrtve blokade nisu primenjive u opštem slučaju, dok su tehnike izbegavanja mrtve blokade ili neprimenjive (zbog nepoznavanja maksimalne potražnje), ili režijski skupe.
- Ukoliko se mrtva blokada niti sprečava, niti izbegava, onda ona sigurno može nastati. U takvim slučajevima može se pribeći *detekciji i oporavku* od mrtvog blokiranja.
- Detekcija stanja mrtve blokade zasniva se na jednostavnim modifikacijama već opisanih algoritama izbegavanja mrtve blokade, s tim što ti algoritmi sada utvrđuju stvarno postojanje mrtve blokade na osnovu iskazane potražnje (zahteva koje su procesi postavili), a ne na osnovu najavljene potencijalne potražnje:
 - na grafu alokacije i utvrđivanju postojanja petlje, ali ovoga puta petlje koja uključuje samo grane potražnje i grane korišćenja (nema grana najave);
 - sprovođenju bankarevog algoritma, ali ovoga puta na stvarno postavljene zahteve (nema maksimalne najave).
- Osnovni problemi i dalje ostaju isti kao kod algoritama izbegavanja: ove strukture podataka zahtevaju poseban memorijski prostor, a algoritmi njihovog održavanja režijsko vreme tokom izvršavanja.
- Međutim, osnovno pitanje jeste kada pokretati algoritam utvrđivanja postojanja mrtve blokade:

- aktivacija algoritma svaki put kada se zahtev za alokaciju resursa ne može ispuniti, jer nema dovoljno slobodnih resursa, detektuje mrtvu blokadu čim ona nastane, ali zahteva veliko režijsko vreme koje u većini slučajeva može biti nepotrebno (jer se proces prosto blokira čekajući na oslobađanje zauzetog resursa, a mrtve blokade zapravo nema);
- reda aktivacija algoritma, povremeno, smanjuje režijske troškove, ali se detekcija mrtve blokade, kao jedne vrste otkaza, odlaže u odnosu na trenutak kada ona zaista nastaje; od trenutka kada je mrtva blokada nastala do trenutka kada je detektovana može sve više procesa biti „upetljivano“ u „klupko“ onih koji su u blokadi, pa se time šteta potencijalno povećava i širi kroz sistem: svi procesi koji budu dalje zahtevali resurse koji učestvuju u mrtvoj blokadi takođe se uključuju u tu mrtvu blokadu.
- Detektovano stanje mrtve blokade može se smatrati otkazom sistema, pa i tretirati odgovarajućim mehanizmom oporavka od tog otkaza.
- Oporavak se može izvesti preotimanjem nekog zauzetog resursa, ukidanjem pojedinih procesa, ili odbijanjem njegovog zahteva za alokaciju (otkaz se tada lokalizuje u jednom procesu). Posebno pitanje je koje procese „žrtvovati“ (ugasiti ih ili im odbiti zahteve). Međutim, svaki od ovih pristupa može imati svoje nedostatke: ili nije primenjiv, ili su posledice (šteta) velike ili neprihvatljive.

Živo blokiranje

- Posmatrajmo algoritam rada filozofa koji pokušava da izbegne mrtvo blokiranje tako što filozof uzima najpre svoju levu, pa onda svoju desnu viljušku, ali da bi se sprečila mrtva blokada, ako desna viljuška nije slobodna, filozof oslobađa levu viljušku i pokušava ceo postupak ispočetka:

```
task type Philosopher
loop
  think;
  loop
    take_left_fork;
    if can_take_right_fork then
      take_right_fork;
      exit loop;
    else
      release_left_fork;
    end if;
  end;
  eat;
  release_left_fork;
  release_right_fork;
end;
end;
```

- Problem ovog rešenja je u tome što može nastati sledeći scenario. Svi filozofi istovremeno uzmu svoju levu viljušku. Zatim svi zaključe da ne mogu da uzmu svoju desnu viljušku, jer je ona zauzeta, pa spuštaju svoju levu viljušku. Teorijski, ovaj postupak se može beskonačno ponavljati, što znači da se svi procesi izvršavaju, tj. ne postoji mrtvo blokiranje, ali nijedan proces ne nastavlja dalje svoj koristan rad.
- Ovakva neregularna situacija u konkurentnom programu, kod koje se grupa procesa izvršava, ali nijedan ne može da napreduje jer u petlji čeka na neki uslov, naziva se *živo blokiranje* (engl. *livelock*).
- Treba razlikovati živo od mrtvog blokiranja. Iako se u oba slučaja procesi nalaze "zaglavljeni" čekajući na ispunjenje nekog uslova, kod mrtvog blokiranja su oni

suspendovani, dok se kod živog izvršavaju, tj. uposlono čekaju. Obe situacije su neispravna stanja u kojima program ne može da napreduje u korisnom smeru, tj. nije obezbeđena njegova živost (engl. *liveness*).

- Ako proces ne vrši uposlono čekanje, živa blokada ne može nastati (ali možda može mrtva blokada). Ostali uslovi za nastanak žive blokade, osim navedenog, su praktično isti kao i za mrtvu blokadu.
- Suština nastanka žive i mrtve blokade je zato u kružnom čekanju, pa se vrlo često ove dve stvari tretiraju na isti način i ponekad i nazivaju istim nazivom (mrtva blokada), ako razlika nije od značaja (da li se proces vrti u petlji čekanja ili je suspendovan).
- Isti problem postoji kod ranije pokazane varijante međusobnog isključenja pomoću uposlenog čekanja:

```
process P1
begin
  loop
    flag1 := true;          (* Announce intent to enter *)
    while flag2 = true do   (* Busy wait if the other process is in *)
      null
    end;
    <critical section>      (* Critical section *)
    flag1 := false;        (* Exit protocol *)
    <non-critical section>
  end
end P1;

process P2
begin
  loop
    flag2 := true;          (* Announce intent to enter *)
    while flag1 = true do   (* Busy wait if the other process is in *)
      null
    end;
    <critical section>      (* Critical section *)
    flag2 := false;        (* Exit protocol *)
    <non-critical section>
  end
end P2;
```

Izgladnjivanje

- Posmatrajmo algoritam rada filozofa koji pokušava da izbegne mrtvo i živo blokiranje tako što svaki filozof uzima obe svoje viljuške zajedno, što znači da uzima ili obe, ili nijednu (a ne jednu po jednu), tj. ako obe nisu slobodne, mora da čeka na sledeći način:

```
task type Philosopher
  loop
    think;
    take_both_forks;
    eat;
    release_both_forks;
  end;
end;
```

Podrazumeva se da je operacija uzimanja obe viljuške atomična.

- Problem ovog rešenja je u tome što može nastati sledeći scenario. Posmatrajmo filozofa X. Neka je njegov levi sused označen sa L, a desni sa D. U jednom trenutku L može da uzme obe svoje viljuške, što sprečava filozofa X da uzme svoju levu viljušku. Pre nego što L spusti svoje viljuške kad završi sa jelom, a što će se dogoditi u konačnom vremenu (jer taj

filozof sigurno ima obe viljuške i konačno završava sa jelom), D može da uzme svoje, što opet sprečava filozofa X da počne da jede. Teorijski, ovaj postupak se može neograničeno ponavljati, što znači da filozof X nikako ne uspeva da zauzme svoje viljuške (resurse) i počne da jede, jer njegovi susedi naizmenično uzimaju njegovu levu, odnosno desnu viljušku.

- Ovakva neregularna situacija u konkurentnom programu, kod koje jedan proces ne može da dođe do željenog resursa jer ga drugi procesi neprekidno pretiču i zauzimaju te resurse, naziva se *izgladnjivanje* (engl. *starvation*), ili *neograničeno odlaganje* (engl. *indefinite postponement*).
- Isti problem izgladnjivanja postoji kod protokola više čitalaca i jednog pisca u kom čitaoci imaju prioritet nad piscima koji čekaju, jer u slučaju da su neki čitaoci aktivni, moguće je da novi čitaoci stalno pristiju, tako da pisac ostaje da čeka neograničeno. Ukoliko pak pisci imaju prioritet u odnosu na čitaoce (novi čitaoci se ne puštaju da krenu sa čitanjem ako postoji pisac koji čeka), čitaoci mogu da budu izgladnjivani ako nov pisac stiže pre nego što tekući pisac završi svoju operaciju.
- Svaki algoritam dodele resursa koji se zasniva na nekoj vrsti prioriteta ima potencijalni problem izgladnjivanja, jer procesi sa prioritetom mogu stalno preticati procese nižeg prioriteta.
- Izgladnjivanje se rešava nekom tehnikom ograničenja čekanja na deljeni resurs: ili se postepeno (sa protokom vremena) procesima koji čekaju povećava prioritet (tzv. *starenje*, engl. *aging*), pa oni vremenom postaju najprioritetniji, ili im se na drugi način vremenski ograniči čekanje, tako da po isteku tog vremena svakako dobijaju resurs na koji čekaju.

Jedno rešenje problema filozofa

- Jedno moguće rešenje problema filozofa koji večeraju, a koje ne poseduje probleme mrtvog i živog blokiranja, kao ni izgladnjivanja, jeste sledeće. Svaki filozof uzima najpre svoju levu, pa onda svoju desnu viljušku. Da bi se sprečilo mrtvo blokiranje, uzimanje leve viljuške dozvoljava se samo prvoj četvorici koji to pokušaju; ukoliko i peti filozof u nekom trenutku želi da uzme svoju levu viljušku, on se blokira. (Taj peti nije uvek jedan isti, unapred određen, što bi moglo da uzrokuje njegovo izgladnjivanje, nego bilo koji koji se zadesio kao peti koji uzima svoju levu viljušku.)
- Ovo rešenje se jednostavno implementira pomoću semafora, pri čemu pet semafora predstavlja viljuške, a još jedan semafor, koji ima inicijalnu vrednost 4, služi za izbegavanje mrtvog blokiranja i sprečavanje petog filozofa da uzme svoju viljušku:

```
Semaphore forks[5]; // Initially set to 1
Semaphore deadlockPrevention(4);
```

```
...
```

```
void Philosopher::run () {
    while (1) {
        think();
        deadlockPrevention.wait();
        forks[left].wait();
        forks[right].wait();
        eat();
        forks[left].signal();
        forks[right].signal();
        deadlockPrevention.signal();
    }
}
```


Zadaci za samostalan rad

6.1

Navesti još neke varijante prioritiranja čitalaca i pisaca osim opisane. Modifikovati datu implementaciju monitora tako da podrži te varijante. Prodiskutovati eventualne probleme koji mogu da nastanu u tim varijantama (eventualno mrtvo blokiranje i izglednjivanje).

6.2

Implementirati opisanu varijantu čitalaca i pisaca korišćenjem standardnih monitora i uslovnih promenljivih `okToRead` i `okToWrite`.

6.3

Realizovati najjednostavniju varijantu čitalaca i pisaca korišćenjem školskog jezgra.

6.4

Prikazati realizaciju sistema čitalaca i pisaca na jeziku Java, pri čemu se prioritet daje čitaocima, a piscima se garantuje pristup u FIFO (*First-In-First-Out*) redosledu.

6.5

Precizno formulisati bankarev algoritam i implementirati ga za slučaj više tipova resursa.

6.6

U cilju izbegavanja mrtvog blokiranja, primenjuje se tehnika čekanja sa vremenskim ograničenjem (engl. *timeout*) na zauzeti resurs. Kako procesu koji zahteva resurs dojaviti da resurs nije zauzet, već da je isteklo vreme čekanja? Prikazati kako izgleda deklaracija operacije zauzimanja resursa i deo koda procesa koji zahteva taj resurs na jeziku Java ili C++. Odgovor prikazati na primeru proizvođača koji periodično proizvodi podatke i šalje ih u bafer, pri čemu u slučaju isteka vremena čekanja na smeštanje u bafer proizvođač povećava svoju periodu rada. (Pretpostaviti da funkcija `delay(int)` blokira pozivajući proces na vreme zadato argumentom.)

6.7

Posmatra se sistem sa pet procesa P_1, P_2, \dots, P_5 i sedam tipova resursa R_1, R_2, \dots, R_7 . Postoji po jedna instanca resursa 2, 5 i 7, a po dve resursa 1, 3, 4 i 6. Proces 1 je zauzeo jednu instancu R_1 i zahteva jednu instancu R_7 . Proces 2 je zauzeo po jednu instancu R_1, R_2 i R_3 i zahteva jednu instancu R_5 . Proces 3 je zauzeo po jednu instancu R_3 i R_4 i zahteva jednu instancu R_1 . Proces 4 je zauzeo po jednu instancu R_4 i R_5 i zahteva jednu instancu R_2 . Proces 5 je zauzeo jednu instancu R_7 . Da li je ovaj sistem u mrtvoj blokadi?

6.8

Neki sistem je u stanju prikazanom u tabeli. Da li je ovaj sistem u bezbednom ili nebezbednom stanju?

Proces	Zauzeo	Traži
P_0	2	12

P_1	4	10
P_2	2	5
P_3	0	5
P_4	2	4
P_5	1	2
P_6	5	13
Slobodnih: 1		

6.9

Neki sistem izbegava mrtvu blokadu algoritmom zasnovanim na grafu alokacije. U sistemu je aktivno četiri procesa ($P_1..P_4$). Procesi su inicijalno najavili korišćenje resursa $R_1..R_3$ prema sledećoj tabeli.

Proces	P_1	P_2	P_3	P_4
Najavio korišćenje resursa	R_1, R_2	R_1, R_3	R_1, R_3	R_2, R_3

Procesi su već izdali sledeće zahteve za alokaciju resursa: P_1-R_1, P_4-R_2 .

- Nacrtati graf zauzeća resursa u ovom stanju, nakon izvršenih navedenih zahteva.
 - Nakon zahteva pod a) proces P_2 traži resurs R_3 .
 - Nakon zahteva pod b) proces P_3 traži resurs R_1 .
 - Nakon zahteva pod c) proces P_4 oslobađa R_2 .
- Nacrtati graf alokacije nakon svakog od zahteva b-d.

6.10

Implementirati prikazani algoritam filozofa koji poseduje problem živog blokiranja, korišćenjem proizvoljnih koncepata za sinhronizaciju.

6.11

Implementirati prikazani algoritam filozofa koji poseduje problem izgladnjivanja, korišćenjem semafora.

IV Specifičnosti RT programiranja

Realno vreme

- Pojam *realno vreme* (engl. *real time*) odnosi se na protok fizičkog vremena, izvan računarskog sistema i nezavisno od njegovog rada, njegovog internog časovnika ili relativnog napredovanja konkurentnih procesa.
- Računarski sistemi zasnovani su na diskretizaciji vremena, tj. na periodično taktovanim internim časovnicima koji mere protok vremena, a jedinu predstavu o protoku spoljašnjeg, realnog vremena softver može imati pomoću ovih časovnika ili drugih, eksternih elektronskih sistema koji obezbeđuju takvu uslugu, ali opet sa ograničenom rezolucijom.
- Za RT sisteme je veoma važno da programski jezik ili okruženje za programiranje obezbeđuje usluge vezane za realno vreme. Te usluge tipično uključuju sledeće aspekte:
 - predstavu o protoku vremena, kao što je pristup časovniku realnog vremena (tj. informacija o apsolutnom datumu i vremenu), merenje proteklog vremena (između dva događaja ili dužine trajanja neke aktivnosti), suspenziju („uspavljivanje“) procesa na zadato vreme, programiranje vremenskih kontrola (engl. *timeout*), itd.;
 - specifikaciju vremenskih zahteva, npr. zadavanje krajnjih rokova ili periode procesa, kao i kontrolu zadovoljenja i detekciju prekršaja vremenskih zahteva.

Časovnik realnog vremena

- Prva usluga vezana za realno vreme odnosi se na dobijanje informacije o *apsolutnom realnom vremenu* (engl. *absolute real time*), tj. o tekućem datumu i vremenu u realnom, fizičkom svetu.
- Informacija o realnom vremenu u računaru može da se obezbedi na sledeće načine:
 - postoji periodični hardverski prekid, a izvršno okruženje/operativni sistem "odbrojava" te prekide; ovo je najjednostavniji mehanizam, korišćen u veoma jednostavnim i starijim sistemima, u kojima tačnost merenja vremena nije od presudnog značaja;
 - postoji izdvojen hardverski uređaj (časovnik) koji obezbeđuje dovoljno tačnu aproksimaciju protoka vremena; izvršno okruženje/operativni sistem pristupa tom časovniku kao što inače pristupa uređajima, očitavajući informaciju o vremenu;
 - postoji hardverski uređaj koji neposredno pristupa nekom eksternom servisu koji obezbeđuje informaciju o globalnom vremenu (UTC), npr. preko radio veze ili drugog telekomunikacionog kanala (npr. mobilna mreža ili GPS); izvršno okruženje/operativni sistem pristupa tom časovniku kao što inače pristupa uređajima.
- Na osnovu neke od navedenih varijanti hardverske podrške, okruženje obezbeđuje aplikaciji informaciju u odgovarajućem formatu, preko programskog interfejsa ili koncepta direktno ugrađenog u programski jezik. Ovakva usluga, nezavisno od vrste hardverske podrške, naziva se „časovnik realnog vremena“.
- Poseban problem predstavljaju razlike u časovnicima realnog vremena u distribuiranim sistemima: različiti časovnici ne mogu biti apsolutno sinhronizovani, pa distribuirane aplikacije moraju uzimati u obzir ove razlike - nije moguće apsolutno tačno definisati totalno hronološko uređenje nezavisnih događaja koji su registrovani (u smislu vremenske odrednice) na različitim čvorovima u distribuiranom sistemu.
- Mnogi programski jezici, a među njima i Ada, Java i C/C++, nemaju neposredne jezičke konstrukte za očitavanje realnog vremena, već obezbeđuju standardne bibliotečne usluge.

Časovnik u jeziku Ada

- Pristup časovniku realnog vremena u jeziku Ada omogućen je preko standardnog paketa `Ada.Calendar`. Ovaj paket realizuje apstraktni tip podataka `Time`, koji predstavlja vremenski trenutak u realnom vremenu, apstraktni tip podataka `Duration`, koji predstavlja vremenski interval protekao između dva vremenska trenutka u realnom vremenu izražen u sekundama, funkciju `Clock`, koja vraća trenutno vreme, kao i niz operacija nad tim tipovima za konverziju, poređenje i aritmetiku:

```

package Ada.Calendar is
  type Time is private;
  subtype Year_Number is Integer range 1901..2099;
  subtype Month_Number is Integer range 1..12;
  subtype Day_Number is Integer range 1..31;
  subtype Day_Duration is Duration range 0.0..86_400.0;

  function Clock return Time;

  function Year (Date:Time) return Year_Number;
  function Month (Date:Time) return Month_Number;
  function Day (Date:Time) return Day_Number;
  function Seconds (Date:Time) return Day_Duration;

  procedure Split (Date:in Time; Year:out Year_Number;
    Month:out Month_Number; Day:out Day_Number;
    Seconds:out Day_Duration);
  function Time_Of (Year:Year_Number; Month:Month_Number;
    Day:Day_Number; Seconds:Day_Duration := 0.0) return Time;

  function "+"(Left:Time; Right:Duration) return Time;
  function "+"(Left:Duration; Right:Time) return Time;
  function "-"(Left:Time; Right:Duration) return Time;
  function "-"(Left:Time; Right:Time) return Duration;
  function "<="(Left,Right:Time) return Boolean;
  function "<="(Left,Right:Time) return Boolean;
  function ">="(Left,Right:Time) return Boolean;
  function ">="(Left,Right:Time) return Boolean;

  Time_Error:exception;
  -- Time_Error may be raised by Time_Of,
  -- Split, Year, "+" and "-"
private
  implementation-dependent
end Ada.Calendar;
```

- Tip `Duration` je predefinisani racionalni tip u fiksnom zarezu. Tačnost i opseg tipa `Duration` zavisi su od implementacije, ali taj opseg mora biti najmanje od -86400.00 do 86400.00 , što pokriva broj sekundi u danu, a granularnost mora biti bar 20 milisekundi.
- Opcioni paket `Ada.Real_Time` u aneksu jezika koji se naziva Real-Time Ada obezbeđuje sličan pristup satu realnog vremena, ali sa finijom granularnošću. Tip `Time` predstavlja apsolutni trenutak u realnom vremenu, ali izražen kao period protekao od nekog predefinisano početka (može biti početak izvršavanja programa ili neki drugi trenutak u istoriji). Opseg tipa `Time` mora biti takav da pokrije najmanje 50 godina od početka izvršavanja programa. Tip `Time_Span` predstavlja vremenski interval protekao između dva data vremenska trenutka. Konstanta `Time_Unit` predstavlja najmanji interval vremena koji se može predstaviti tipom `Time`, izražen u sekundama. Vrednost konstante `Tick` je granularnost funkcije `Clock`, odnosno prosečan interval u kom se vrednost koju ova funkcija vraća ne menja; ova konstanta ne sme biti veća od jedne milisekunde:

```

package Ada.Real_Time is
  type Time is private;
  Time_First: constant Time;
  Time_Last: constant Time;
  Time_Unit: constant := implementation_defined_real_number;

  type Time_Span is private;
  Time_Span_First: constant Time_Span;
  Time_Span_Last: constant Time_Span;
  Time_Span_Zero: constant Time_Span;
  Time_Span_Unit: constant Time_Span;

  Tick: constant Time_Span;
  function Clock return Time;

  function "+" (Left: Time; Right: Time_Span) return Time;
  function "+" (Left: Time_Span; Right: Time) return Time;
  -- similarly for "-", "<", etc.

  function To_Duration (TS: Time_Span) return Duration;
  function To_Time_Span (D: Duration) return Time_Span;
  function Nanoseconds (NS: Integer) return Time_Span;
  function Microseconds (US: Integer) return Time_Span;
  function Milliseconds (MS: Integer) return Time_Span;
  type Seconds_Count is range implementation-defined;
  procedure Split (T : in Time; SC: out Seconds_Count; TS : out Time_Span);
  function Time_Of (SC: Seconds_Count; TS: Time_Span) return Time;

private
  -- not specified by the language
end Ada.Real_Time;

```

Časovnik u jeziku Java

- U standardnom paketu `java.lang` jezika Java postoji klasa `System`, čija statička operacija `currentTimeMillis()` vraća (prirodan) broj milisekundi proteklih od ponoći, 1. januara 1970. po Griniču. Klasa `Date` iz standardnog paketa `java.util` obezbeđuje apstraktni tip podataka za datum i vreme, uz operacije za konverziju. Podrazumevani konstruktor klase `Date` pravi objekat koji sadrži trenutni datum i vreme.
- Specifikacija pod nazivom RT Java još pruža i usluge vezane za časovnik realnog vremena i vremenske tipove visoke rezolucije. Apstraktna klasa `HighResolutionTime` predstavlja generalizaciju takvih tipova:

```

public abstract class HighResolutionTime implements java.lang.Comparable {
  ...
  public boolean equals(HighResolutionTime time);

  public final long getMilliseconds();
  public final int getNanoseconds();
  public void set(HighResolutionTime time);
  public void set(long millis);
  public void set(long millis, int nanos);
  ...
}

```

- Tri izvedene klase jesu `AbsoluteTime`, `RelativeTime` i `RationalTime`. Klasa `AbsoluteTime` predstavlja trenutak u realnom vremenu (kao `Time` u jeziku Ada), ali izražen relativno u odnosu na 1. januar 1970. Klasa `RelativeTime` predstavlja interval

vremena (kao `Duration` u jeziku Ada). Klasa `RationalTime` predstavlja relativno vreme (interval), koje se odnosi na periodu dešavanja nekih događaja (npr. periodičnih procesa):

```
public class AbsoluteTime extends HighResolutionTime {
    public AbsoluteTime(AbsoluteTime t);
    public AbsoluteTime(long millis, int nanos);
    ...
    public AbsoluteTime add(long millis, int nanos);
    public final AbsoluteTime add(RelativeTime time);
    ...
    public final RelativeTime subtract(AbsoluteTime time);
    public final AbsoluteTime subtract(RelativeTime time);
}

public class RelativeTime extends HighResolutionTime {
    public RelativeTime(long millis, int nanos);
    public RelativeTime(RelativeTime time);
    ...
    public RelativeTime add(long millis, int nanos);
    public final RelativeTime add(RelativeTime time);
    public void addInterarrivalTo(AbsoluteTime destination);
    public final RelativeTime subtract(RelativeTime time);
    ...
}
```

- Apstraktna klasa `Clock` predstavlja generalizaciju časovnika koji se mogu kreirati u programu. Jezik dozvoljava formiranje proizvoljno mnogo vrsta časovnika. Pritom, uvek postoji jedan časovnik realnog vremena kome se pristupa statičkom operacijom `getRealTimeClock()`:

```
public abstract class Clock {
    public Clock();

    public static Clock getRealtimeClock();

    public abstract RelativeTime getResolution();
    public AbsoluteTime getTime();
    public abstract void getTime(AbsoluteTime time);
    public abstract void setResolution(RelativeTime resolution);
}
```

Merenje proteklog vremena

- Sledeći važan element vezan za vreme u RT programima jeste merenje proteklog vremena, npr. vremena izvršavanja nekog dela koda ili trajanja neke aktivnosti, ili vremena proteklog između dva događaja, ili vremena čekanja na neki događaj.
- Ukoliko jezik ili okruženje omogućava pristup časovniku realnog vremena, kao i odgovarajuće operacije nad apstraktnim tipovima podataka kojima se predstavljaju vremenski trenuci i intervali, informacija o proteklom vremenu dobija se jednostavno, očitavanjem vremenskog trenutka na početku i na kraju merenja i izvršavanjem operacije koja izračunava protekli interval između ta dva očitana trenutka.
- Na primer, na jeziku Ada to se može uraditi na sledeći način:

```
with Ada.Calendar; use Ada.Calendar;
declare
    start, finish : Time;
    interval : Duration;
begin
```

```

start := Clock; -- start measuring time interval
... -- some activity
finish := Clock; -- end measuring
interval := finish - start; -- measured time interval
end;

```

ili, korišćenjem paketa `Real_Time`:

```

with Ada.Real_Time; use Ada.Real_Time;
declare
  start, finish : Time;
  maxDuration : Time_Span := To_Time_Span(1.7);
begin
  start := Clock;
  ... -- some activity
  finish := Clock;
  if finish-start > maxDuration then
    raise Time_Exceeded; -- a user-defined exception
  end if;
end;

```

- Sličan pristup je moguć i na jeziku Java:

```

{
  AbsoluteTime start, finish;
  RelativeTime interval;
  Clock clock = Clock.getRealtimeClock();

  start = clock.getTime();
  ... // some activity
  finish = clock.getTime();
  interval = finish.subtract(start); // measured interval
}

```

- Merenje proteklog vremena u školskom jezgru podržano je apstrakcijom `Timer`. Ova apstrakcija predstavlja vremenski brojač kom se može zadati početna vrednost i koji odbrojava po otkucajima sata realnog vremena unazad, do nule, kada se sam zaustavlja. Brojač ne može meriti intervale duže od onih zadatih konstantom `maxTimeInterval`. Jedinica vremena je zavisna od implementacije. Brojač se može zaustaviti (operacija `stop()`), pri čemu vraća vreme proteklo od kada je pokrenut:

```

typedef Time ...;
const Time maxTimeInterval = ...;

class Timer {
public:

  Timer ();

  void start (Time period=maxTimeInterval);
  Time stop ();
  void restart (Time=0);

  Time elapsed ();
  Time remained();
};

```

- Funkcija `restart()` ponovo pokreće brojač za novozadatom početnom vrednošću, ili sa prethodno zadatom vrednošću, ako se nova ne zada. Funkcije `elapsed()` i `remained()` vraćaju proteklo, odnosno preostalo vreme.

- Ovakav pristup omogućava kreiranje proizvoljno mnogo objekata klase `Timer` koji će nazavisno i uporedo moći da mere svoje intervale, npr. na sledeći način:

```
Timer* timer = new Timer();

timer->start();
// some activity
timer->stop();
Time t = timer->elapsed();
delete timer;
```

Kašnjenje procesa

- Potrebno je ponekad da proces zahteva svoje "uspavljivanje", odnosno odlaganje nastavka svog izvršavanja na određeno vreme, bilo relativno u odnosu na tekući trenutak (na zadati interval od momenta uspavljivanja), bilo do određenog apsolutnog trenutka.
- U jeziku Ada, ovakva suspenzija obavlja se konstruktom `delay` koji zadaje relativno kašnjenje, odnosno vremenski interval (tipa `Duration`) za koji se odlaže nastavak daljeg izvršavanja:

```
delay 10.0;  -- relative delay for 10 seconds
```

- Konstrukst `delay until` definiše apsolutni trenutak (tipa `Time`) do kog se odlaže nastavak daljeg izvršavanja; naravno, ukoliko je taj trenutak već prošao, izvršavanje se ne odlaže:

```
start := Clock;
... -- some action
delay until start+10.0;  -- absolute delay
...
```

- U jeziku Java, odlaganje izvršavanja relativno u odnosu na tekući trenutak obavlja se operacijom `sleep()` klase `Thread`, kojoj se zadaje vreme odlaganja u milisekundama:

```
public final void sleep (long delayTime);
```

- Treba naglasiti da ovi konstrukti mogu da garantuju jedino da proces neće nastaviti izvršavanje *pre* zadatog vremena. Koliko će stvarno odlaganje izvršavanja biti, odnosno kada će zaista proces nastaviti svoje izvršavanje, naravno zavisi od implementacije platforme i ostalih spremnih procesa koji konkurišu za procesor. Sve u svemu, faktori koji mogu dodatno da odlože izvršavanje preko zadatog vremena i utiču na stvarno kašnjenje u odnosu na vreme zadato konstruktom jesu:
 - razlika granularnosti između časovnika realnog vremena i specifikacije vremena u konstruktu: računar ne može delovati preciznije od granulacije ovog časovnika, bez obzira na to koliko je precizno zadato vreme kašnjenja u konstruktu;
 - vreme za koje su prekidi maskirani, jer za to vreme procesor neće reagovati na prekide, pa neće biti ni asinhrona promene konteksta;
 - režijskog vremena koje je potrebno da se proces vrati iz suspenzije, a potom i restaurira njegov kontekst izvršavanja;
 - vreme za koje je proces spreman, ali se ne izvršava, jer se izvršavaju drugi, eventualno prioritetniji procesi.
- Odlaganje stvarnog nastavka izvršavanja procesa u odnosu na vreme zadato konstruktom, a koje je posledica navedenih faktora, naziva se *lokalno plivanje* (engl. *local drift*) i ne može se eliminisati.

- Međutim, kod implementacije periodičnih procesa, važno je eliminisati akumulaciju ovih grešaka, tj. *kumulativno plivanje* (engl. *cumulative drift*), pri kome se lokalna pomeranja mogu superponirati.
- Na primer, sledeća naivna implementacija periodičnog procesa pati i od lokalnog i od kumulativnog plivanja, jer se trenutak aktivacije svaki put pomera za dužinu izvršavanja tela ovog procesa, kao i za dodatnu vrednost lokalnog plivanja:

```
task PeriodicTask;

task body PeriodicTask is
  period : constant Duration := ...;
begin
  loop
    ... -- periodic action
    delay period;
  end loop;
end PeriodicTask;
```

- Bolje rešenje, koje eliminiše kumulativno plivanje (ali i dalje ima lokalno plivanje koje se ne može eliminisati), jer se naredni apsolutni trenutak aktivacije uvek izračunava u odnosu na prethodni takav dodavanjem konstante periode, jeste sledeće:

```
task body PeriodicTask is
  period : constant Duration := 5.0;
  nextActivation : Time;
begin
  nextActivation := Clock + period;
  loop
    delay until nextActivation;
    nextActivation := nextActivation + period;
    ... -- periodic action
  end loop;
end PeriodicTask;
```

Vremenske kontrole

- Jedan od najčešćih zahteva vezanih za vreme u RT programima jeste ograničenje vremena čekanja na neki događaj ili trajanja neke aktivnosti:
 - na prolazak kroz sinhronizacionu tačku kod međuprocene komunikacije pomoću deljenog objekta: ulazak u kritičnu sekciju, pristup deljenom objektu ili neki uslov;
 - na uspostavu ili završetak komunikacije: sinhrono slanje ili prijem poruke, završetak randevua, ili povratni odgovor na poslatu poruku;
 - na završetak izvršavanja neke aktivnosti ili dela koda;
 - na neki drugi događaj, na primer spoljašnji signal od uređaja.
- Svaki od navedenih slučajeva zapravo uvodi vremensku kontrolu kao mehanizam detekcije greške, jer bi u idealnom slučaju navedeno čekanje moralo biti ograničeno, i to završeno u zadatom vremenskom roku prema konstrukciji sistema i vremenskim zahtevima. Međutim, zbog mogućnosti otkaza, čekanje ili trajanje aktivnosti može biti i neograničeno, pa je zato potrebno uvoditi vremenska ograničenja tog čekanja ili aktivnosti.
- Jedan od uzroka neograničenog čekanja jeste i mrtva blokada, pa se ovaj mehanizam, kao što je već navedeno, može koristiti i kao mehanizam za sprečavanje (ili oporavak) od mrtve blokade.

- U opštem slučaju, *vremenska kontrola* (engl. *timeout*) je ograničenje vremena za koje je neki proces spreman da čeka na neki događaj ili uspostavljanje komunikacije.
- Ukoliko to čekanje ili izvršavanje aktivnosti traje duže od predviđenog, može se smatrati da je nastala neregularna situacija, i to čekanje ili izvršavanje prosto treba prekinuti, i eventualno preduzeti akcije oporavka od tako detektovanog otkaza.
- Zbog stalne mogućnosti otkaza, u RT sistemima je neophodno da sinhronizacioni i komunikacioni konstrukti podržavaju vremenske kontrole, tj. čekanje (suspenciju, blokiranje) ograničenog trajanja.

Deljene promenljive i vremenske kontrole

- Kao što je ranije izneseno, komunikacija i sinhronizacija pomoću deljene promenljive uključuje:
 - međusobno isključenje
 - uslovnu sinhronizaciju.
- Bez obzira na to kakav se konstrukt koristi, međusobno isključenje podrazumeva potencijalno blokiranje procesa koji želi da uđe u zauzetu kritičnu sekciju ili dobije pristup deljenom objektu/resursu. Vreme tog čekanja u opštem slučaju zavisi od vremena koje je potrebno da proces koji je ušao u kritičnu sekciju iz nje izađe, ali i da drugi procesi koju su stigli ranije ili imaju viši prioritet to urade. U slučaju da nema otkaza, izgladnjivanja niti mrtve ili žive blokade, ovo vreme je ograničeno, pa zato nije uvek neophodno obezbediti vremensku kontrolu pridruženu čekanju na ulaz u kritičnu sekciju ili pristup deljenom objektu. Međutim, u slučaju mogućnosti otkaza procesa koji je zauzeo deljeni objekat, ili kao zaštita od prekoračenja zadatih vremenskih rokova, ili kao mehanizam zaštite od mrtve blokade, vremenska kontrola može biti neophodna.
- Uslovna sinhronizacija u opštem slučaju takođe može da blokira proces na neodređeno vreme, u zavisnosti od prirode aplikacije i mogućnosti otkaza. Na primer, proizvođač koji čeka na slobodno mesto u ograničenom baferu može nograničeno dugo da ostane suspendovan ukoliko je proces potrošača otkazao ili nije u stanju da preuzme element iz bafera neodređeno ili neprihvatljivo dugo vreme. Zbog toga je i u ovakvim slučajevima potrebno obezbediti vremenski ograničeno čekanje.
- Ovakve vremenske kontrole treba da budu moguće za sve prikazane koncepte uslovne sinhronizacije: semafore, uslovne promenljive u monitorima i ulaze u zaštićene objekte.
- Na primer, POSIX podržava operaciju `wait()` na semaforu uz zadavanje vremenske kontrole:

```
if (sem_timedwait(&sem, &timeout) < 0) {
    if (errno == ETIMEDOUT) {
        /* timeout occurred */
    }
    else { /* some other error */ }
} else {
    /* semaphore passed */
};
```

- Jezik Ada tretira pozive ulaza u zaštićene objekte na isti način kao i randevu na klijenstkoj strani, pa je njima moguće pridružiti vremenske kontrole kao što će to biti opisano u nastavku.
- U jeziku Java, poziv operacije `wait()` može da bude sa vremenskom kontrolom zadatom u milisekundama, ili u milisekundama i nanosekundama:

```
public final void wait (long timeout);
public final void wait (long millis, int nanos);
```

Komunikacija porukama i vremenske kontrole

- Kod sinhronog slanja, pošiljalac se potencijalno blokira dok poruka ne bude primljena, pa je ovo čekanje potrebno ograničiti vremenskom kontrolom iz već navedenih razloga mogućnosti otkaza u komunikaciji ili na strani primaoca, ili zbog neprihvatljivo dugog odlaganja trenutka kada primalac bude u stanju da poruku primi.
- I kod sinhronog prijema, primalac se blokira dok ne dobije poruku, pa je i ovo čekanje potrebno vremenski kontrolisati iz sličnih razloga.
- U jeziku Ada čekanje na prijem poruke, tj. uspostavljanje randevua na serverskoj strani moguće je vremenski kontrolisati pomoću naredbe `delay` u jednoj grani nedeterminističke naredbe `select`. Na primer, sledeći primer prikazuje server koji prima pozive drugih procesa-klijenata, pri čemu se odsustvo poziva u roku od 5 sekundi posebno tretira:

```

task Server is
  entry call1 (...);
  entry call2 (...);
  ...
end Server;

task body Server is
  ...
begin
  loop
    select
      accept call1 (...) do
        ...
      end call;
    or
      accept call2 (...) do
        ...
      end call;
    or
      delay 5.0;
      ... -- actions for timeout
    end select;
  end loop;
end Server;

```

- Grana iza `delay` se aktivira ako je od trenutka nailaska na `select` prošlo zadato vreme; kada istekne to vreme, a u međuvremenu nije bilo poziva na ostalim ulazima u `select`, izvršavanje bira tu granu i nastavlja se izvršavanje naredbi iza naredbe `delay`.
- Moguće je specifikovati i apsolutno vreme ograničenja prijema poruke, konstruktom `delay until`; tada vreme nije zadato relativno, kao interval od nailaska na `select`, nego kao apsolutni trenutak (tipa `Time`):

```

task body Restaurant is
  isOpen : Boolean := True;
  closingTime : constant Time := ...;
begin
  ...
  loop
    select
      when isOpen =>
        accept bookTable(person:in Name; time:in Time; booked:out Boolean) do
          -- Look up a free table
          booked := ... ; -- Prepare the reply
        end bookTable;
      or
        when isOpen =>

```

```

    accept enter(p:in Name; canEnter:out Boolean; table:out Number) do
      -- Look up the table booked by p or a free one
      canEnter := ... ; -- Prepare the reply
      table := ...;
    end enter;
  or
    accept exit(p:in Name) do
      -- Free the table booked by p
    end exit;
  or
    delay until closingTime;
    isOpen := False;
  end select;
end loop;
...
end Restaurant;

```

- U jeziku Ada moguće je vremenski kontrolisati uspostavu komunikacije i na strani pozivaoca, kako za pozive ulaza u procese (randevu), tako i za pozive ulaza u zaštićene objekte, jer je na strani klijenta notacija i semantika potpuno ista. Na primer:

```

select
  Server.call(p);
or
  delay 1.0;
  ... -- actions for timeout
end select;

```

Ovo je poseban oblik naredbe `select`, koji ne može imati više od jedne alternative poziva. Vremenska kontrola predstavlja rok za uspostavljanje komunikacije, a ne za njen završetak.

- U slučaju da pozivalac želi da uspostavi komunikaciju samo ukoliko je pozvani odmah spreman da je prihvati, onda je poziv sledeći:

```

select
  Server.call(p);
else
  -- alternative action
end select;

```

- Vremenske kontrole mogu da budu pridružene i aktivnostima, odnosno izvršavanju bloka naredbi koje treba prekinuti ukoliko traju duže od zadatog vremena. Oblik je tada:

```

select
  delay 0.1;
  -- actions for timeout
then abort
  -- time-constrained activity
end select;

```

Ovaj konstrukt izvršava se na sledeći način. Po nailasku izvršavanja na `select`, aktivira se merenje vremena zadato u `delay` (može biti zadato relativno ili apsolutno, sa `until`). Uporedo se započinje izvršavanje naredbi iza `then abort` (osim ako apsolutno vreme zadato u `delay until` već nije prošlo). Ako se blok naredi iza `then abort` završi pre isteka merenog vremena, odnosno apsolutno zadatog trenutka (tj. vremenske kontrole), biće završen ceo konstrukt i merenje vremena zaustavljeno i odbačeno. Ukoliko pak vremenska kontrola istekne pre završetka ovih naredbi, prekida se njihovo izvršavanje, a nastavlja se izvršavanje naredbi iza `delay`.

- Sledeći primer pokazuje kako se može vršiti neko izračunavanje sa postepenim povećanjem tačnosti rezultata, sve dok ograničeno vreme ne istekne. Prikazani proces ima

obavezni deo koji izračunava neki početni, približni rezultat, a onda ulazi u iterativno popravljavanje rezultata sve dok vremensko ograničenje ne istekne:

```
declare
  preciseResult : Boolean := False;
  completionTime : Time := ...;
  ...
begin
  completionTime := ...;
  ... -- compute approximate result quickly
  select
    delay until completionTime;
  then abort
    while canBeImproved loop
      ... -- improve result
    end loop;
    preciseResult := True;
  end select;
end;
```

Vremenske kontrole u školskom jezgru

- Ista apstrakcija `Timer` iz školskog jezgra može se koristiti i za vremenske kontrole. Apstrakcija `Timer` predstavlja vremenski brojač kome se zadaje početna vrednost i koji odbrojava po otkucanjima sata realnog vremena unazad, dok ne dobroji do nule.
- Ako je potrebno vršiti vremensku kontrolu, onda korisnička klasa treba da implementira interfejs (tj. bude izvedena iz jednostavne apstraktne klase) `Timeable` koja poseduje čistu virtuelnu funkcije `timeout()`. Ovu funkciju korisnik može da redefiniše, a poziva je `Timer` kada zadato vreme istekne, odnosno kada dobroji do nule. Opisani interfejsi izgledaju ovako:

```
class Timeable {
public:
  virtual void timeout () = 0;
};

class Timer {
public:

  Timer (Time period=maxTimeInterval, Timeable*=0);
  ~Timer ();

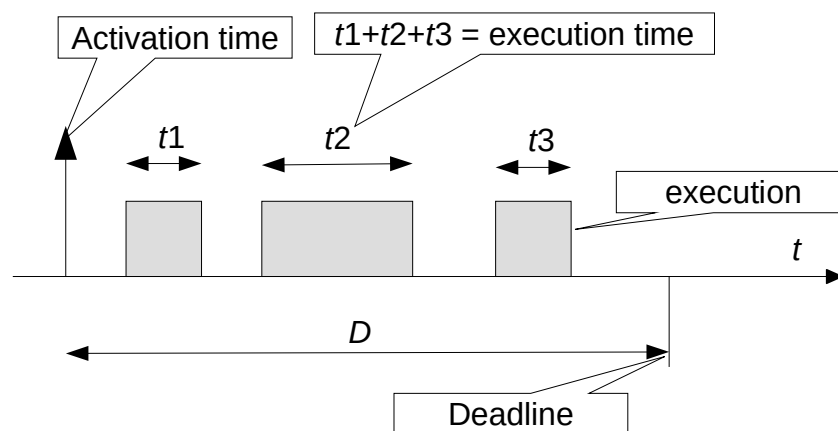
  void start (Time period=maxTimeInterval);
  Time stop ();
  void restart (Time=0);

  Time elapsed () const;
  Time remained() const;
};
```

- Drugi argument konstruktora predstavlja pokazivač na objekat kome treba poslati poruku `timeout()` kada zadato vreme istekne. Ako se ovaj pokazivač ne zada, brojač neće poslati ovu poruku pri isteku vremena.

Specifikacija vremenskih zahteva

- Jedan koncept koji olakšava specifikaciju različitih vremenskih zahteva u RT aplikacijama jeste koncept *temporalnih opsega* (engl. *temporal scope*). Temporalni opseg je skup naredbi u programu, najčešće neki konstrukt u postojećem sekvencijalnom ili konkurentnom jeziku (npr. blok ili proces odnosno nit), za koji se mogu u programu specifikovati neka od sledećih vremenskih ograničenja i karakteristika, a koja izvršno okruženje treba da obezbedi:
 - trenutak aktivacije* (engl. *activation time*), odnosno način aktivacije, koji može biti *periodičan* (engl. *periodic*) ili *aperiodičan* (engl. *aperiodic*);
 - vremenski rok* (engl. *deadline*): trenutak do kog se izvršavanje opsega mora završiti; može biti zadat u relativnom odnosu, kao interval od trenutka aktivacije (u oznaci D), ili kao apsolutni trenutak;
 - minimalno kašnjenje* (engl. *minimum delay*): minimalno vreme koje mora proteći od trenutka aktivacije dok ne započne izvršavanje opsega;
 - maksimalno kašnjenje* (engl. *maximum delay*): maksimalno vreme koje može proteći od trenutka aktivacije dok ne započne izvršavanje opsega;
 - maksimalno vreme izvršavanja* (engl. *maximum execution time*): maksimalno procesorsko vreme koje izvršavanje opsega sme iskoristiti;
 - maksimalno proteklo vreme* (engl. *maximum elapse time*): maksimalno vreme koje može proteći od trenutka započinjanja do trenutka završetka izvršavanja opsega; u opštem slučaju, ovo vreme je različito od maksimalnog vremena izvršavanja (veće je), jer izvršavanje opsega može biti isprekidano zbog izvršavanja drugih procesa.



- U odnosu na način, odnosno trenutak aktivacije, temporalni opseg može biti definisan kao:
 - periodičan* (engl. *periodic*): trenuci aktivacije su periodično raspoređeni sa periodom T ; takvi su tipično poslovi koji periodično uzimaju odbirke neke fizičke veličine i obrađuju ih ili reaguju na promenu te veličine, ili izvršavaju kontrolne petlje; ovakvi procesi tipično imaju svoje vremenske rokove koji moraju biti zadovoljeni; ovi vremenski rokovi zadaju se relativno u odnosu na trenutak svake periodične aktivacije; ako je perioda interval T , vremenski rok je interval D za koji tipično mora važiti $D \leq T$, jer se izvršavanje u svakoj aktivaciji mora završiti do trenutka sledeće aktivacije ili ranije, inače nema smisla;
 - aperiodičan* (engl. *aperiodic*): takvi su obično poslovi koje treba obaviti kao reakciju na asinhronu spoljašnje *događaje* (engl. *event*), odnosno koji treba da reaguju na signale sa spoljašnjih uređaja, ili su trenuci aktivacije unapred određeni vremenski trenuci (u realnom vremenu), i koji obično imaju definisano *vreme odziva* (engl.

response time), odnosno maksimalno dozvoljeno vreme reakcije na događaj; upravo to vreme može se uzeti kao vremenski rok D aperiodičnog temporalnog opsega.

- U opštem slučaju, može se smatrati da aperiodični događaji stižu slučajno, prema nekoj slučajnoj raspodeli. Takva raspodela, teorijski, dozvoljava nalete događaja u određenom periodu, i to u proizvoljnoj gustini. Drugim rečima, verovatnoća da dva događaja stignu u proizvoljno malom vremenskom razmaku je uvek veća od nule. Međutim, ovakav teorijski slučaj najčešće ne odgovara realnoj situaciji, a ni hardver nije u stanju da registruje događaje koji su proizvoljno bliski. Pored toga, to ne dozvoljava analizu rasporedivosti u najgorem slučaju.
- Zbog toga se uvek uvodi pretpostavka o *minimalnom mogućem vremenskom razmaku* između asinhronih događaja (i time trenutaka aktivacije njima pridruženih poslova, odnosno temporalnih opsega). Takvi aperiodični događaji i temporalni opsezi, koji imaju definisan minimalni (najgori) vremenski razmak T , nazivaju se *sporadičnim* (engl. *sporadic*). Minimalno vreme T razmaka između dve pojave sporadičnog događaja može se smatrati periodom odgovarajućeg sporadičnog procesa u najgorem slučaju, pa se ta veličina može koristiti u analizi rasporedivosti.
- U praksi, specifikacija temporalnih opsega, koji se najčešće vezuju za procese odnosno niti programa, svodi se najčešće na specifikaciju sledećih vremenskih ograničenja:
 - pokretanje periodičnih procesa sa odgovarajućom periodom T ;
 - završavanje svih procesa do njihovog roka D u odnosu na trenutak aktivacije; za periodične procese, tipično je $D \leq T$.
- Tako se problem zadovoljenja vremenskih zahteva svodi na problem raspoređivanja procesa tako da zadovolje svoje vremenske rokove (engl. *deadline scheduling*):
 - ukoliko su rokovi strogi i ne smeju se propustiti, sistem se naziva *hard RT sistemom*; najčešće se pod tim podrazumeva to da se prekoračenje vremenskog roka mora detektovati kao greška, a sistem mora da reaguje odgovarajućom akcijom oporavka od otkaza;
 - ukoliko sistem toleriše povremeno prekoračenje rokova, naziva se *soft RT sistemom*.

Periodični procesi

- U jeziku Ada, periodičan proces koji ne pati od kumulativnog plivanja (engl. *cumulative drift*) može da se definiše na sledeći način:

```
task body PeriodicTask is
  period : Duration := ...; -- or
  period : Time_Span := Milliseconds(...);
  nextActivation : Time;
begin
  nextActivation := Clock + period;
  loop
    delay until nextActivation;
    nextActivation := nextActivation + period;
    ... -- periodic action
  end loop;
end PeriodicTask;
```

- Za neke algoritme raspoređivanja (npr. algoritam EDF), kako će biti pokazano kasnije, izvršno okruženje, odnosno raspoređivač procesa, mora da ima informaciju o vremenskom roku procesa. U osnovnoj verziji jezika Ada, ova informacija o vremenskom roku ne može biti eksplicitno definisana kada se proces definiše na način kako je prikazano gore. RT

proširenje jezika Ada, međutim, ima podršku za eksplicitno definisanje vremenskog roka u paketu `Ada.Dispatching.EDF`. Periodičan proces se tada može definisati ovako:

```
with Ada.RealTime; use Ada.RealTime;
with Ada.Dispatching.EDF; use Ada.Dispatching.EDF;

task body PeriodicTask is
  period : Time_Span := Milliseconds(...);
  deadline : Time_Span := Milliseconds(...);
  nextActivation : Time;
begin
  nextActivation := Clock + period;
  loop
    Delay_Until_And_Set_Deadline(nextActivation,deadline);
    nextActivation := nextActivation + period;
    ... -- periodic action
  end loop;
end PeriodicTask;
```

Procedura `Delay_Until_And_Set_Deadline` iz navedenog paketa suspenduje tekući proces do apsolutnog trenutka zadatog prvim argumentom (isto kao konstrukt `delay until`). Kada proces bude aktiviran („probuđen“), odnosno ponovo spreman za izvršavanje, apsolutni trenutak vremenskog roka biće postavljen na `nextActivation + deadline`. U navedenom paketu postoje i drugi uslužni potprogrami, npr. procedura `Set_Deadline` koja eksplicitno postavlja vremenski rok tekućem ili nekom drugom imenovanom procesu i druge.

- U jeziku RT Java, periodičan proces može da se definiše na sledeći način:

```
public class Periodic extends RealtimeThread {
  public Periodic (PriorityParameters pp, PeriodicParameters p) { ... };

  public void run() {
    while(true) {
      ... // periodic action
      waitNextPeriod(); // delay until next periodic activation
    }
  }
}

PeriodicParameters per = new PeriodicParameters(
  new AbsoluteTime(...), // Start time
  new RelativeTime(100,0), // Period in milliseconds, nanoseconds
  new RelativeTime(20,0), // Maximum execution time in ms,ns
  new RelativeTime(60,0) // Deadline in ms,ns
  null, // Overrun handler
  null // Deadline miss handler
);

PriorityParameters pri = new PriorityParameters(...);
Periodic myThread = new Periodic(pri,per); // Create thread
myThread.start(); // and activate it
```

- Treba primetiti da se ovde vremenski parametri, kao što su perioda i (relativan) vremenski rok, zadaju eksplicitno kroz objekat tipa `PeriodicParameters` koji se dostavlja kao argument konstruktora periodičnog procesa.
- Periodični procesi u školskom Jezgru mogu se konstruisati pomoću koncepta niti (klasa `Thread`) i vremenskog brojača (klasa `Timer`). Videti zadatke.

Sporadični procesi

- U jeziku Ada, sporadičan proces može da se definiše tako da koristi zaštićeni objekat na kom se međusobno sinhronizuju prekid, kao spoljašnji tok kontrole koji poziva jedan ulaz tog zaštićenog objekta i time signalizira događaj, i sporadični proces, koji čeka na drugi ulaz tog zaštićenog objekta:

```
protected SporadicController is
  procedure interruptHandler; -- mapped onto interrupt
  entry waitForNextInterrupt;
private
  eventOccurred : Boolean := False;
end SporadicController;
```

```
protected SporadicController is

  procedure interruptHandler is
  begin
    eventOccurred := True;
  end interruptHandler;

  entry waitForNextInterrupt when eventOccurred is
  begin
    eventOccurred := False;
  end waitForNextInterrupt;

end SporadicController;
```

```
task SporadicTask;
task body SporadicTask is
begin
  loop
    SporadicController.waitForNextInterrupt;
    -- sporadic action
  end loop;
end SporadicTask;
```

- U jeziku RT Java, sporadični procesi se konstruišu slično kao i periodični, osim što proces čeka na sledeći događaj umesto na sledeću periodičnu aktivaciju.
- U školskom Jezgru sporadični procesi pobuđuju se *prekidom* (engl. *interrupt*). Klasa `InterruptHandler` predstavlja generalizaciju prekida. Njen interfejs izgleda ovako:

```
typedef unsigned int IntNo; // Interrupt Number

class InterruptHandler : public Thread {
protected:

  InterruptHandler (IntNo num, void (*intHandler) ());

  virtual int handle () { return 0; }
  void interruptHandler ();

};
```

- Korisnik iz ove apstraktne klase treba da izvede sopstvenu klasu za svaku vrstu prekida koja se koristi. Korisnička klasa treba da bude *Singleton*, a prekidna rutina definiše se kao statička funkcija te klase (jer ne može imati argumente, pa ni `this`). Korisnička prekidna rutina treba samo da pozove funkciju jedinog objekta `InterruptHandler::interruptHandler()`. Dalje, korisnik treba da redefiniše virtuelnu

funkciju `handle()`. Ovu funkciju će pozvati sporadični proces kada se dogodi prekid, pa u njoj korisnik može da navede proizvoljan kod. Treba primetiti da se taj kod izvršava svaki put kada se dogodi prekid, pa on ne treba da sadrži petlju, niti čekanje na prekid.

- Osim navedene uloge, klasa `InterruptHandler` obezbeđuje i implicitnu inicijalizaciju interapt vektor tabele: konstruktor ove klase zahteva broj prekida i pokazivač na prekidnu rutinu. Na ovaj način ne može da se dogodi da programer zaboravi inicijalizaciju, a ta inicijalizacija je lokalizovana, pa su zavisnosti od platforme svedene na minimum.
- Primer upotrebe:

```
// Timer interrupt entry:
const int TimerIntNo = 0;

class TimerInterrupt : public InterruptHandler {
protected:

    TimerInterrupt () : InterruptHandler(TimerIntNo,timerInterrupt) {}

    static void timerInterrupt () { instance->interruptHandler(); }

    virtual int handle () {
        ... // User-defined code for one release of the sporadic process
    }

private:
    static TimerInterrupt* instance;
};

TimerInterrupt* TimerInterrupt::instance = new TimerInterrupt;
```

Kontrola zadovoljenja vremenskih zahteva

- Uključivanje vremenskih ograničenja u RT programe podrazumeva i mogućnost njihovog narušavanja, što se može smatrati otkazom u RT programu.
- Iako narušavanje nekog vremenskog zahteva, prvenstveno vremenskog roka, u *soft* RT sistemima ne mora da se smatra otkazom, već prihvatljivom pojavom (do izvesne mere), ovakav sistem ponekad treba da bude svestan prekoračenja roka, kako bi preduzeo odgovarajuću akciju (npr. skupljao statistiku o prekoračenjima, beležio ih ili preduzeo neku drugu akciju).
- Sa druge strane, prekoračenje vremenskog roka u *hard* RT sistemima tretira se kao otkaz, pa sistem mora da detektuje takve pojave, kako bi preduzeo odgovarajuće postupke oporavka od takvog otkaza, npr. neke od ranije opisanih tehnika oporavka od otkaza (BER ili FER).
- Iako se za *hard* RT sisteme pretpostavlja dokazivanje izvodljivosti unapred, pri konstrukciji sistema, odnosno to da je unapred izveden „dokaz“ da se vremenski rokovi nikada neće prekoračiti, čak i u najgorim scenarijima izvršavanja, prekoračenja vremenskih rokova se ipak mogu dogoditi iz sledećih razloga:
 - sve analize rasporedivosti zasnivaju se na proceni *vremena izvršavanja procesa u najgorem slučaju* (engl. *worst-case execution time*, WCET); ova procena u većini slučajeva ne može biti sasvim precizna i pouzdana, pa se može dogoditi da stvarna vremena izvršavanja prekorače te procene, a time i rasporedivost više nije garantovana;

- analiza rasporedivosti može biti pogrešno izvedena ili sprovedena na nekoj drugoj pogrešnoj pretpostavci, recimo na zanemarivanju nekih efekata koji u realnosti nisu zanemarivi;
- sistem funkcioniše izvan opsega za koji je projektovan, npr. tako što se pojavi neočekivano prekomerno opterećenje, recimo pojava sporadičnih događaja češće nego što je predviđeno.
- Zbog toga je u RT sistemima potrebno detektovati sledeće pojave narušavanja vremenskih ograničenja:
 - prekoračenje vremenskog roka (engl. *overrun of deadline*, *deadline miss*)
 - prekoračenje vremena izvršavanja u najgorem slučaju (engl. *overrun of WCET*)
 - pojavu sporadičnih događaja frekventnije nego što je predviđeno
 - istek vremenskih kontrola (engl. *timeout*).
- Pojava nekog od poslednja tri navedena prekoračenja ne mora obavezno da znači da će neki vremenski rok biti prekoračen, odnosno da će se dogoditi otkaz. Na primer, ako se jedan proces izvršava nešto duže od svog procenjenog WCET, ne znači da će on ili neki drugi proces prekoračiti svoj vremenski rok: možda procesor ima sasvim dovoljno vremena da izvrši sve druge procese na vreme, recimo zato što se drugi procesi izvršavaju daleko kraće od svog WCET, ili sporadični događaji ne stižu uopšte ili stižu veoma retko, mnogo ređe od najgore procenjene gustine. Međutim, ako se prekoračenje roka ipak dogodi, i to možda u nekom sasvim drugom procesu koji nije prekoračio svoj WCET (nego nije dobio procesor na vreme), otkaz će biti detektovan u tom drugom procesu ili procesima kao prekoračenje vremenskog roka, dok je uzrok greške u procesu koji je prekoračio svoj WCET. Zbog ograničenja propagacije te greške, može biti korisno detektovati je na samom mestu nastanka.

Detekcija prekoračenja roka

- U jeziku Ada, detekcija prekoračenja roka periodičnog ili sporadičnog procesa može se obaviti pomoću `select-then-abort` konstrukta:

```

task body PeriodicTask is
  period : constant Time_Span := Milliseconds(...);
  deadline : constant Time_Span := Milliseconds(...);
  nextActivation : Time;
  nextDeadline : Time;
begin
  nextActivation := Clock + period;
  loop
    delay until nextActivation;
    nextDeadline := nextActivation + deadline;
    nextActivation := nextActivation + period;
    select
      delay until nextDeadline;
      -- Deadline miss detected here, perform recovery
    then abort
      -- periodic action
    end select;
  end loop;
end PeriodicTask;

```

- Jedan od potencijalnih problema kod ovog pristupa jeste to što se aktivnost procesa koji je prekoračio svoj rok prekida, a nastavlja se izvršavanje koda za oporavak unutar istog procesa, čime proces i dalje zauzima procesor i potencijalno ugrožava izvršavanje drugih procesa. Drugačiji pristup bi bio, recimo, da proces nastavi svoje izvršavanje pod drugih

uslovima, npr. sa drugim prioritetom, odnosno da se obrada ove greške prepusti nekom drugom procesu.

- U jeziku RT Java, izvršno okruženje (Java virtuelna mašina) će asinhrono signalizirati prekoračenje roka procesa i biće pozvana polimorfna metoda `handleAsyncEvent()` klase izvedene iz klase `AsyncEventHandler`. Objekat ovakve klase zadaje se kao parametar `overrunHandler` konstruktora `PeriodicParameters`, kao što je pokazano. Sporadični procesi u jeziku RT Java nemaju eksplicitni vremenski rok, pa se smatraju *soft* procesima.

Detekcija prekoračenja ostalih vremenskih ograničenja

- Ukoliko izvršavanje jednog procesa nije prekidano od trenutka kada je taj proces dobio procesor (engl. *preempted*), onda je vreme izvršavanja tog procesa jednako proteklom fizičkom vremenu, pa se detekcija prekoračenja WCET može obaviti slično kao i detekcija prekoračenja roka. Međutim, to je veoma retko slučaj u konkurentnim programima.
- Paket `Ada.Execution_Time` jezika Ada obezbeđuje usluge za pristup „časovniku“ koji meri vreme izvršavanja tekućeg procesa na procesoru. Funkcija `Clock` iz ovog paketa vraća vreme izvršavanja tekućeg (ili nekog drugog imenovanog) procesa:

```
package Ada.Execution_Time is
  type CPU_Time is private;
  CPU_Time_First : constant CPU_Time;
  CPU_Time_Last  : constant CPU_Time;
  CPU_Time_Unit  : constant := implementation-defined-real-number;
  CPU_Tick       : constant Time_Span;

  function Clock return CPU_Time;

  function "+" (Left : CPU_Time; Right : Time_Span) return CPU_Time;
  function "+" (Left : Time_Span; Right : CPU_Time) return CPU_Time;
  ... -- similar for "-", ">", "<", etc.
  procedure Split (T:in CPU_Time; SC:out Seconds_Count; TS:out Time_Span);
  function Time_Of (SC:Seconds_Count; TS:Time_Span:=Time_Span_Zero) return
CPU_Time;
private
  ... -- not specified by the language
end Ada.Execution_Time;
```

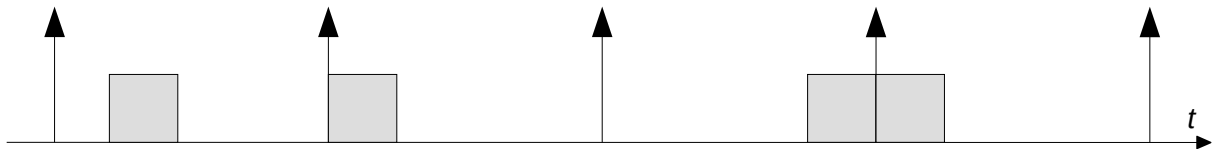
- Jezik Java dozvoljava kontrolu prekoračenja WCET za objekte tipa `RealTimeThread` na isti način kao i za prekoračenje roka: izvršno okruženje (Java virtuelna mašina) će asinhrono signalizirati prekoračenje WCET procesa i biće pozvan kod definisan u klasi izvedenoj iz klase `AsyncEventHandler`. Objekat ovakve klase zadaje se kao parametar `costHandler` konstruktora `PeriodicParameters`.
- Pojava sporadičnih događaja češće nego što je predviđeno se može ili sprečiti, ili detektovati. Jedan pristup sprečavanju prečestih pojava sporadičnih događaja jeste kontrola učestanosti hardverskih prekida uticajem na same registre hardverskih uređaja koji generišu te prekide (interrupt kontroleri), čime se oni mogu maskirati u određenim intervalima vremena, ili se može drugačije podesiti njihovo registrovanje. Drugi pristup vezan je za jednu tehniku primenjivu kod raspoređivanja koja će biti opisana kasnije.

Proračun vremenskih parametara

- RT sistemima se postavljaju veoma raznovrsni vremenski zahtevi. Nažalost, postojeća inženjerska praksa uglavnom primenjuje *ad-hoc*, a ne rigorozne i formalne metode za projektovanje i implementaciju sistema koji ispunjavaju te zahteve. To znači da se sistem

najpre konstruiše tako da bude logički ispravan, a zatim se koristi i testira na ispunjenje vremenskih zahteva. Ukoliko neki zadati vremenski zahtevi nisu zadovoljeni, vrše se fina podešavanja, unapređenja i ispravke sistema. Ovakav pristup nije rigorozan i ne garantuje ispunjenje zahteva u svim slučajevima, pa je dobijeni sistem nepouzdan, ali i nepregledan i težak za održavanje.

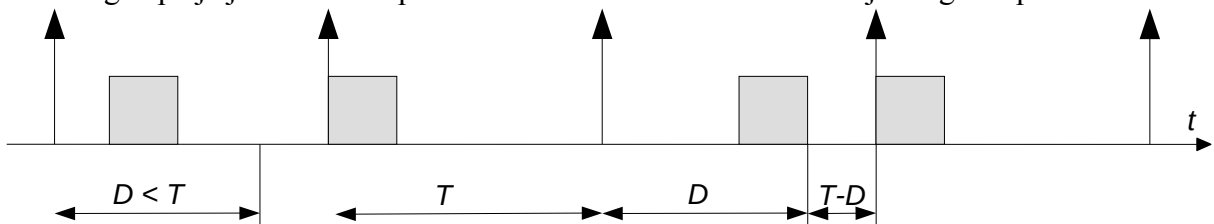
- Postoje ipak i strožije, pouzdanije metode za specifikaciju i verifikaciju zadovoljavanja vremenskih ograničenja. Istraživanja i metode u tom domenu uglavnom idu u dva pravca:
 - Upotreba formalnih jezika sa preciznom semantikom koja uključuje i vremenske karakteristike koje se mogu definisati i analizirati, tako da se definisan i implementiran sistem može verifikovati na ispunjenje zadatih vremenskih zahteva. Nažalost, ove formalne tehnike nisu u široj upotrebi u praksi zbog nedovoljne zrelosti, nedostupnosti alata i problema sa skalabilnošću (tipično problem eksplozije stanja pri verifikaciji sistema u svim mogućim situacijama).
 - Upotreba tehnika za utvrđivanje performansi realizovanog sistema i dokazivanje izvodljivosti, odnosno rasporedivosti zahtevanog opterećenja na postojeće resurse (procesore i drugo). Neke od ovih tehnika biće prikazane u narednom poglavlju.
- Formalna verifikacija RT sistema uključuje dve vrste aktivnosti:
 - Verifikaciju konzistentnosti specifikovanih vremenskih zahteva: pod pretpostavkom da je na raspolaganju idealan, proizvoljno brz procesor, treba proveriti da li su vremenski zahtevi koherentni i konzistentni, tj. da li je uopšte teorijski moguće ispuniti ih. Na primer, sledeći zahtevi su kontradiktorni, pa ih nikako nije moguće ispuniti ako je $t_1 > t_2$: B ne sme početi pre nego što prođe t_1 vremena od kad se završi A , C ne sme početi pre nego što prođe t_2 vremena od kad se završi A , a B mora da se završi pre nego što počne C .
 - Verifikacija implementacije: da li se vremenski zahtevi mogu ispuniti na raspoloživom skupu konačnih resursa, konačne i raspoložive brzine.
- U nastavku će biti prikazane neke tehnike konstrukcije RT sistema na osnovu postavljenih zahteva i proračun odgovarajućih vremenskih parametara periodičnih i sporadičnih procesa za neke najčešće obrasce zahteva u praksi.
- Jedna tipična potreba jeste to da se neka radnja radi periodično, sa periodom T_0 . Na primer, da se očitava odbirak neke fizičke veličine ili ispituje ispravnost nekog podsistema.
- Ako se ovaj posao poveri periodičnom procesu sa periodom T , otvara se pitanje kakav vremenski rok D postaviti tom procesu. Najrelaksiraniji uslov, koji ostavlja najviše prostora za raspoređivanje procesa i povećava mogućnost rasporedivosti tih procesa na procesoru, jeste da D bude maksimalno moguće, $D = T$. Ovakav uslov omogućava i jednostavnije modele rasporedivosti, kao što će biti pokazano kasnije.
- Međutim, proces se može raspoređivati na različite načine i izvršavati u različitim intervalima vremena, počev od svog trenutka aktivacije, pa sve dok ne istekne njegov vremenski rok: raspoređivač je dužan samo da obezbedi zadato ispunjenje vremenskog roka, dok je sve drugo nepredvidivo. Zbog toga se proces može izvršavati, pa time i odbirak uzimati u bilo kom trenutku između trenutaka dve susedne aktivacije. Prema tome, njegove aktivacije mogu izgledati i ovako (slika prikazuje neprekidna izvršavanja, radi jednostavnosti; izvršavanja mogu biti isprekidana, ako je sistem sa preotimanjem, engl. *preemptive*, ali se suština ne menja):



$$D = T$$

To znači da se izvršavanja u dve susedne aktivacije mogu „slepiti“, odnosno aktivnosti izvršiti odmah jedna iza druge, ili pak „rastaviti“ na rastojanje (skoro) dvostruke periode (ako je dužina izvršavanja jako kratka u odnosu na periodu, što lako može biti slučaj). Ovo može da bude neprihvatljivo, recimo onda kada posao uzima odbirke neke fizičke veličine koju prati: odbirci te vrednosti u jako bliskim trenucima neće se mnogo razlikovati, ili se neće razlikovati uopšte, pa takvi mogu biti beskorisni, dok će odbirci uzeti u slučaju druge krajnosti biti uzeti u vremenskom razmaku od približno $2T$.

- Prema tome, da bi se zadati posao *sigurno* uradio najmanje jednom u svakom intervalu T_0 , ako je $D = T$, perioda procesa mora se definisati tako da je $T \leq T_0/2$: periodičan proces mora biti barem dvostruko veće učestanosti $f = 1/T \geq 2f_0$ nego što je zadata učestanost $f_0 = 1/T_0$ (ovaj rezultat podseća na Teoremu odabiranja iz teorije obrade signala i telekomunikacija). Naravno, dati posao će tada biti obavljan nekada i češće, ali se samo ovako garantuje da će sigurno biti obavljan najmanje jednom u svakom intervalu T_0 .
- Druga opcija jeste ta da se postavi neko $D < T$. Tada se aktivacije mogu rasporediti ovako:



- Kao što se vidi iz slike, najveći razmak između dva susedna izvršavanja ovog posla, ako se njegovo trajanje zanemari, odnosno u najgorem slučaju smatra zanemarljivo malim (ovaj uslov se postavlja kao sigurno, pesimistično ograničenje u najgorem slučaju za koji se proračun uvek vrši) je $T + D$, pa T i D moraju biti izabrani tako da je:

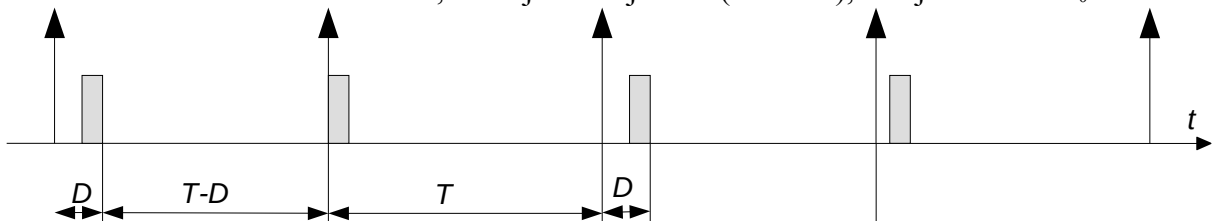
$$T + D \leq T_0$$

Treba primetiti da se za $T = D$ ova nejednakost svodi na onu prethodno izvedenu.

- Dakle, u opštem slučaju, kada je $D \leq T$, dve susedne aktivacije su razmaknute za Δt tako da je:

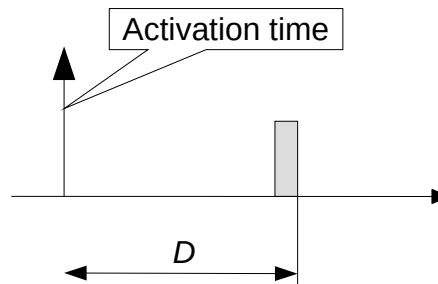
$$T - D \leq \Delta t \leq T + D$$

- Treba primetiti i to da se razmak približan potrebnom T_0 može postići i tako što se postavi veoma tesan vremenski rok D , značajno manji od T ($D \ll T$), i da je tada $T \approx T_0$:

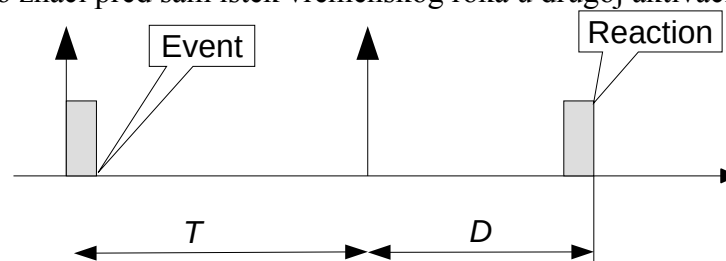


- Drugi tipičan zahtev koji se postavlja jeste to da sistem treba da reaguje na neki događaj u zadatom vremenskom roku T_c .
- Ako je taj događaj neki diskretan, sporadičan događaj, reakcija se može poveriti sporadičnom procesu koji se aktivira na taj događaj, pa je onda vremenski rok tog procesa jednak zadatom vremenu reakcije:

$$D = T_c$$



- *Reakcija na sporadičan događaj može se poveriti i periodičnom procesu koji će u svakoj aktivaciji proveriti da li se događaj dogodio i, ako jeste, preduzeti reakciju, a ako nije, „uspavati“ do sledeće aktivacije. Da bi reakcija bila garantovana u vremenu T_c , mora se analizirati najgori slučaj, odnosno najnepovoljniji scenario. Taj scenario je sledeći: u jednoj aktivaciji, periodičan proces može da zaključi da se događaj nije dogodio, a da se događaj dogodi neposredno nakon toga. Zbog toga će tek naredna aktivacija periodičnog procesa detektovati događaj i preduzeti reakciju. Najnepovoljniji slučaj (za najkasniju reakciju) je taj da se izvršavanje u prvoj aktivaciji (kada je događaj propušten) dogodi na samom početku periode aktivacije, a u drugoj, kada se detektuje događaj, dogodi najkasnije, a to znači pred sam istek vremenskog roka u drugoj aktivaciji:*



- Zbog toga mora da bude zadovoljeno:

$$T + D \leq T_c$$

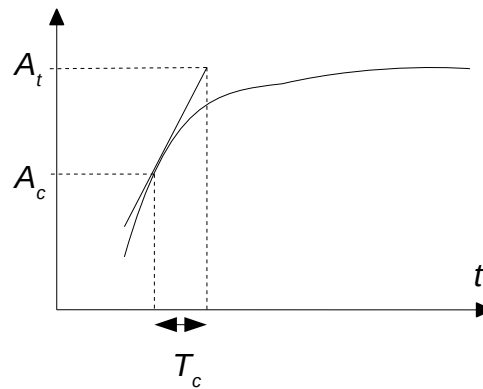
Ako se ovaj uslov maksimalno relaksira, nejednakost postaje jednakost, a ako se D maksimalno relaksira i postavi na $D = T$, dobija se relacija koja je analogna onoj ranije izvedenoj, kojom se zahteva da se sporadičan događaj nadgleda sa periodom dvostruko manjom od potrebnog vremena reakcije:

$$T = D = T_c/2$$

- Jedan tipičan slučaj događaja na koji treba reagovati jeste prekoračenje neke granične vrednosti praćene fizičke analogne veličine: sistem treba da prati tu fizičku veličinu i reaguje ako ona pređe neki prag A_t . Kako je za reakciju sistema svakako potrebno neko vreme, potrebno je odrediti maksimalnu dozvoljenu veličinu vremena reakcije T_c . Ono se može odrediti recimo na sledeći način. Neka je, prema fizičkim karakteristikama, maksimalna brzina promene te fizičke veličine (maksimum apsolutne vrednosti prvog izvoda te veličine po vremenu) ΔA_m . Da bi sistem reagovao za vreme T_c , reakcija mora da usledi ako praćena veličina prekorači neku vrednost A_c , tako da važi:

$$\frac{|A_t - A_c|}{T_c} \geq \Delta A_m, \text{ pa je}$$

$$T_c \leq \frac{|A_t - A_c|}{\Delta A_m}$$



- Ovo dozvoljava da se A_c i T_c izaberu prilično proizvoljno, ali je njihov izbor stvar suprotstavljenih uslova koje treba balansirati (engl. *trade off*): da bi se sistemu ostavilo više vremena za reakciju, što relaksira i uslove rasporedivosti, T_c treba da bude veće, ali to onda zahteva i pesimističniju reakciju na kritičnu vrednost A_c koja je tim više različita od one koja je realni prag reakcije A_t ; zbog toga reakcija na A_c može biti i lažna, jer se reakcija može izazvati preuranjeno i bespotrebno, pošto praćena veličina uopšte neće dostići graničnu vrednost A_t koja je daleko. U suprotnom, sistemu se ostavlja kraće vreme za reakciju, pa time i strožiji i teže ostvarivi uslovi rasporedivosti.
- Dakle, periodičan proces sada treba da očitava vrednost praćene analogne veličine A i da reaguje na događaj kada ta veličina prekorači vrednost A_c i preduzme reakciju najkasnije za vreme T_c .
- Uređaj za uzimanje odbirka analogne veličine sa nekog senzora i njenu konverziju u digitalni oblik nazivaju se *analogno-digitalni (A/D) konvertori*. Oni često funkcionišu po sledećem principu. Procesor mora najpre da zada uzimanje odbirka signala sa analognog senzora i početak njegove konverzije u digitalni oblik. To zadavanje obično se izvodi upisom neke vrednosti u upravljački (kontrolni) registar uređaja, tj. A/D konvertora. Posle toga, uređaj vrši potrebno očitavanje i konverziju, što zahteva određeno vreme. Kada završi sa konverzijom i pripremi podatak za očitavanje u svom registru za podatke, uređaj može ili generisati prekid procesoru, ili samo postaviti odgovarajući indikator u svom statusnom registru. U svakom slučaju, taj podatak za očitavanje može biti spreman tek posle određenog vremena od pokretanja konverzije, a o spremnosti tog podatka procesor (tj. aplikacija) može biti obavestena na jedan od sledeća tri načina:
 - stalnim očitavanjem statusnog registra i ispitivanjem indikatora (engl. *polling*), tj. uposlenim čekanjem (engl. *busy waiting*) da taj indikator postane postavljen;
 - prekidom koji generiše postavljeni indikator (engl. *interrupt*);
 - puštanjem da protekne odgovarajuće vreme za koje će ova vrednost sigurno biti raspoloživa, odnosno za koje će konverzija biti završena; maksimalno vreme trajanja konverzije T_{AD} je poznat parametar samog uređaja.

U slučaju da konverzija nije uspeła ili je došlo do bilo kakve greške, podaci o grešci se mogu očitati iz statusnog registra uređaja.

- Periodično uzimanje odbirka može se onda realizovati ovako:

```
task body SensorReader is
  period : Time_Span := ...;
  nextActivation : Time;
  durationAD : Time_Span := ...; -- Maximal duration of A/D conversion
  value : Real;
begin
  nextActivation := Clock + period;
  loop
    delay until nextActivation;
```

```

nextActivation := nextActivation + period;
startConversion(); -- Start A/D conversion
delay durationAD; -- Wait for A/D conversion to complete
value := readAD(); -- Read A/D
... -- process the value
end loop;
end SensorReader;

```

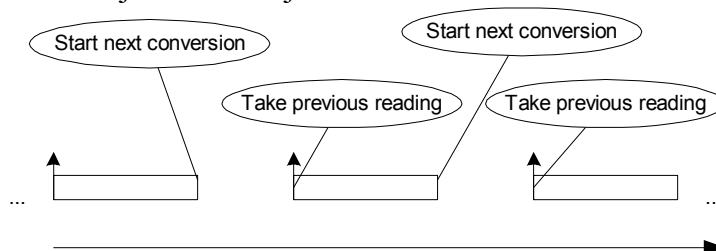
- Za ovakav pristup, najgori mogući scenario je ponovo sledeći: u jednoj aktivaciji, odmah nakon aktivacije, uzima se odbirak merene veličine koji je izpod granične, ali veličina prelazi graničnu odmah nakon toga. Ta veličina biće očitana u narednoj aktivaciji. Vreme od trenutka kada je veličina prekoračila graničnu do reakcije sistema je najduže kada je izvršavanje u prvoj aktivaciji bilo najranije moguće, tj. odmah nakon aktivacije, a u drugoj najkasnije moguće, neposredno pred istek vremenskog roka. Tako nejednakost koja definišu vremenske parametre izgleda isto kao ranije:

$$T + D \leq T_c$$

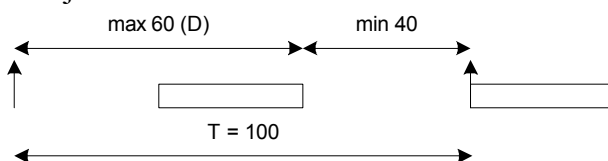
Ako se ovaj uslov maksimalno relaksira, nejednakost postaje jednakost, a ako se D maksimalno relaksira i postavi na $D = T$, dobija se ista ranija relacija:

$$T = D = T_c/2$$

- Međutim, nepotrebno je da periodični proces u istoj aktivaciji i zada konverziju i očitava vrednost, jer to zahteva dodatnu suspenziju (pauzu) tokom istog izvršavanja procesa zbog čekanja na završetak konverzije.
- Zato se u ovakvim situacijama može primeniti tehnika tzv. *pomeranja periode* (engl. *period displacement*). Ona se sastoji u tome da se na samom kraju jednog izvršavanja (aktivacije) periodičnog procesa koji proziva senzor zapravo zada očitavanje i pokrene konverzija (upisom odgovarajuće vrednosti u upravljački registar uređaja), a na početku narednog izvršavanja konvertovana vrednost očitava iz registra podataka. Na taj način se rad konvertora paralelizuje sa radom procesora na drugim procesima u sistemu, pa nema potrebe za dodatnom suspenzijom procesa zbog čekanja na konverziju. Ova tehnika prikazana je na sledećoj slici:

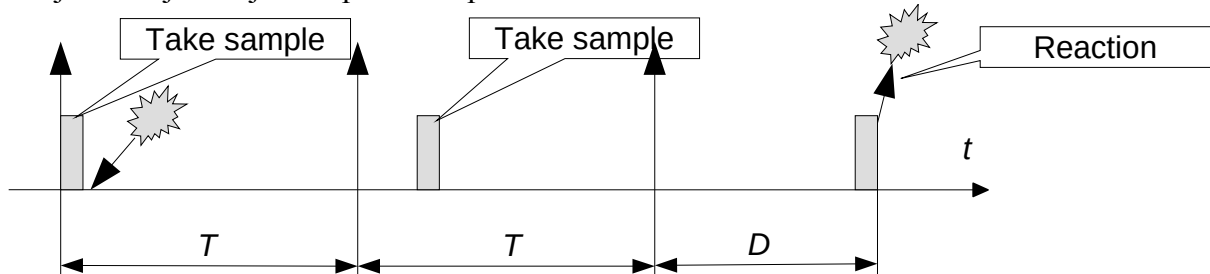


- Sa druge strane, međutim, mora biti ispoštovano vreme trajanja konverzije T_{AD} , što znači da najmanji razmak između kraja jednog izvršavanja procesa (tj. pokretanja A/D konverzije) i početka drugog (kada se očitava konvertovana vrednost) mora biti najmanje jednaka trajanju konverzije u najgorem slučaju, T_{AD} . Ovo se može ispuniti zadavanjem vremenskog roka periodičnog procesa $D = T - T_{AD}$. Na primer, ako je $T = 100$, a $T_{AD} = 40$, onda je $D = 60$:



- Sada najgori scenario za reakciju na prekoračenje očitane vrednosti u roku T_c izgleda ovako. U nekoj prvoj aktivaciji, na kraju izvršavanja koje je veoma kratko, uzima se odbirak veličine u trenutku kada je ona tik ispod granične. Odmah nakon tog trenutka uzimanja odbirka veličina prekoračuje graničnu. U narednoj aktivaciji vrednost očitana sa

A/D konvertora tako neće biti preko granične, pa sistem neće reagovati u toj aktivaciji. Na kraju te aktivacije, zadata konverzija uzeće odbirak koji je veći od granične, pa će taj odbirak biti očitán u narednoj, trećoj aktivaciji, u kojoj će sistem preduzeti odgovarajuću reakciju. Najduže vreme koje može proteći je u slučaju kada je izvršavanje u prvoj aktivaciji najranije moguće, odmah nakon aktivacije, a u trećoj najkasnije moguće, tj. kada je reakcija učinjena neposredno pred istek roka:



- Kako je od trenutka kada je vrednost prešla graničnu, do trenutka reakcije sistema tada proteklo vreme jednako (najgora procena) $2T + D$, nejednakosti koje postavljaju vremenske uslove sada izgledaju ovako:

$$2T + D \leq T_c$$

$$T - D \geq T_{AD}$$

- Najrelaksiraniji uslovi dobijaju se kada ove nejednakosti postanu jednakosti. Rešavanjem sistema takve dve jednačine dobija se:

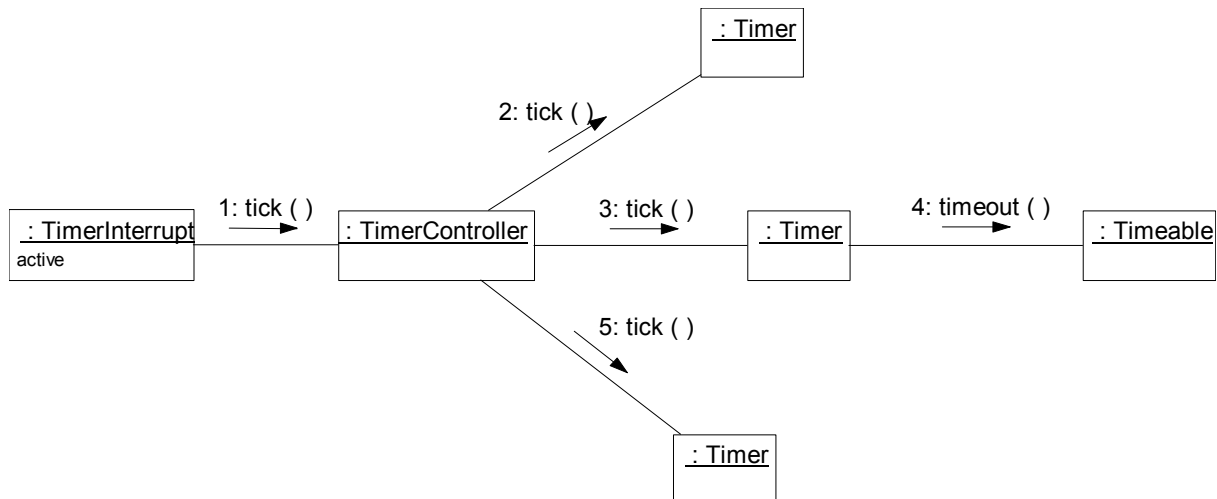
$$T = \frac{T_c + T_{AD}}{3}$$

$$D = T - T_{AD} = \frac{T_c - 2T_{AD}}{3}$$

Implementacija u školskom jezgru

Mehanizam merenja vremena

- Mehanizam merenja vremena u školskom jezgru može se jednostavno realizovati na sledeći način. Od hardvera se očekuje da obezbedi (što tipično postoji u svakom računaru) brojač (sat) realnog vremena koji periodično generiše prekid sa zadatim brojem. Ovaj prekid kontrolisaće objekat klase `TimerInterrupt`. Ovaj objekat pri svakom otkucaju sata realnog vremena, odnosno po pozivu prekidne rutine, prosleđuje poruku `tick()` jednom centralizovanom *Singleton* objektu tipa `TimerController`, koji sadrži spisak svih kreiranih objekata tipa `Timer` u sistemu. Ovaj kontroler će proslediti poruku `tick()` svim brojačima.
- Svaki brojač tipa `Timer` se prilikom kreiranja prijavljuje u spisak kontrolera (operacija `sign()`), što se obezbeđuje unutar konstruktora klase `Timer`. Analogno, prilikom ukidanja, brojač se odjavljuje (operacija `unsign()`), što obezbeđuje destruktora klase `Timer`.
- Vremenski brojač poseduje atribut `isRunning` koji pokazuje da li je brojač pokrenut (odbrojava) ili ne. Kada primi poruku `tick()`, brojač će odbrojati samo ako je ovaj indikator jednak 1, inače jednostavno vraća kontrolu pozivaocu. Ako je prilikom odbrojanja brojač stigao do 0, šalje se poruka `timeout()` povezanom objektu tipa `Timeable`.
- Opisani mehanizam prikazan je na sledećem dijagramu interakcije:



- Kako do objekta `TimerController` stižu konkurentne poruke sa dve strane, od objekta `InterruptHandler` poruka `tick()` i od objekata `Timer` poruke `sign()` i `unsign()`, ovaj objekat mora da bude sinhronizovan (monitor). Slično važi i za objekte klase `Timer`.
- Međutim, ovakav mehanizam dovodi do sledećeg problema: može se dogoditi da se unutar istog toka kontrole (niti) koji potiče od objekta `TimerInterrupt`, pozove `TimerController::tick()`, čime se ovaj objekat zaključava za nove pozive svojih operacija, zatim odatle pozove `Timer::tick()`, brojač dobrojava do nule, poziva se `Timeable::timeot()`, a odatle neka korisnička funkcija. Unutar ove korisničke funkcije može se, u opštem slučaju, kreirati ili brisati neki `Timer`, sve u kontekstu iste niti, čime se dolazi do poziva operacija objekta `TimerController`, koji je ostao zaključan. Na taj način dolazi do kružnog blokiranja (engl. *deadlock*) i to jedne niti same sa sobom.
- Čak i ako se ovaj problem zanemari, ostaje problem eventualno predugog zadržavanja unutar konteksta niti koja ažurira brojače, jer se ne može kontrolisati koliko traje izvršavanje korisničke operacije `timeout()`. Time se neodređeno zadržava mehanizam ažuriranja brojača, pa se gubi smisao samog merenja vremena.
- Problem se može rešiti na sledeći način: potrebno je na nekom mestu prekinuti kontrolu toka i razdvojiti kontekste uvođenjem niti kojoj će biti signaliziran događaj, zapravo operaciju `timeout()` pozvati asinhrono. Ovde je to učinjeno tako što objekat `Timer`, ukoliko poseduje pridružen objekat `Timeable`, poseduje i jedan aktivni objekat `TimerThread` koji poseduje nezavisan tok kontrole u kome se obavlja poziv operacije `timeout()`. Objekat `Timer` će, kada vreme istekne, samo signalizirati događaj pridružen objektu `TimerThread` i vratiti kontrolu objektu `TimerController`. `TimerThread` će, kada primi signal, obaviti poziv operacije `timeout()`. Na ovaj način se navedeni problemi eliminišu, jer se sada korisnička funkcija izvršava u kontekstu sopstvene niti, odnosno poziva asinhrono.
- Opisana implementacija ima problem što je kompleksnost ažuriranja vremenskih brojača srazmerna broju tih brojača ($O(n)$), pa može biti neefikasna za veliki broj objekata. Ostavlja se čitaocu da osmisli efikasniju implementaciju.
- Kompletan kod za ovaj podsistem dat je u nastavku.

```

// Project:   Real-Time Programming
// Subject:   Multithreaded Kernel
// Module:    Timer
// File:      timer.h
// Created:   November 1996
// Revised:   August 2003
// Author:    Dragan Milicev
// Contents:  Timers
//           Types:      Time

```

```

//      Classes:      Timer
//      Interfaces:   Timeable

#ifndef _TIMER_
#define _TIMER_

#include "collect.h"
#include "semaphor.h"

////////////////////////////////////
// type Time
// constant maxTimeInterval
////////////////////////////////////

typedef unsigned long int Time;
const Time maxTimeInterval = ~0;

////////////////////////////////////
// interface Timeable
////////////////////////////////////

class Timeable {
public:
    virtual void timeout () = 0;
};

////////////////////////////////////
// class Timer
////////////////////////////////////

class TimerThread;
class Semaphore;

class Timer : public Object {
public:

    Timer (Time period=maxTimeInterval, Timeable* toNotify=0);
    ~Timer ();

    void start    (Time period=maxTimeInterval) { restart(period); }
    Time stop     ();
    void restart  (Time=0);

    Time elapsed () { return initial-counter; }
    Time remained() { return counter; }

protected:

    friend class TimerController;
    CollectionElement* getCefForController () { return &cefForController; }
    void tick ();

private:

    Timeable* myTimeable;
    TimerThread* myThread;

    Time counter;

```

```

    Time initial;

    int isRunning;

    Semaphore mutex;
    CollectionElement ceForController;

    RECYCLE_DEC(Timer)

};

#endif

// Project:   Real-Time Programming
// Subject:   Multithreaded Kernel
// Module:    Timer
// File:      timer.cpp
// Created:   November 1996
// Revised:   August 2003
// Author:    Dragan Milicev
// Contents:  Timers
//           Classes:
//               Timer
//               TimerThread
//               TimerController
//               TimerInterrupt

#include "timer.h"
#include "semaphor.h"

////////////////////////////////////
// class TimerThread
////////////////////////////////////

class TimerThread : public Thread {
public:

    TimerThread (Timeable*);

    void signal ();
    void destroy ();

protected:

    virtual void run ();

private:

    Event ev;
    Timeable* myTimeable;
    int isOver;

    RECYCLE_DEC(TimerThread)

};

RECYCLE_DEF(TimerThread);

```

```
TimerThread::TimerThread (Timeable* t) : myTimeable(t), isOver(0),
RECYCLE_CON(TimerThread) {}
```

```
void TimerThread::signal () {
    ev.signal();
}
```

```
void TimerThread::destroy () {
    isOver=1;
    ev.signal();
}
```

```
void TimerThread::run () {
    while (1) {
        ev.wait();
        if (isOver)
            return;
        else
            if (myTimeable) myTimeable->timeout();
    }
}
```

```
////////////////////////////////////
// class TimerController
////////////////////////////////////
```

```
class TimerController {
public:
```

```
    static TimerController* Instance();
```

```
    void tick ();
```

```
    void sign    (Timer*);
```

```
    void unsign (Timer*);
```

```
private:
```

```
    TimerController () {}
```

```
    Collection rep;
```

```
    Semaphore mutex;
```

```
};
```

```
TimerController* TimerController::Instance () {
    static TimerController instance;
    return &instance;
}
```

```
void TimerController::tick () {
```

```

    Mutex dummy(&mutex);
    CollectionIterator* it = rep.getIterator();
    for (it->reset(); !it->isDone(); it->next())
        ((Timer*)it->currentItem())->tick();
}

void TimerController::sign (Timer* t) {
    Mutex dummy(&mutex);
    if (t) rep.append(t->getCEForController());
}

void TimerController::unsign (Timer* t) {
    Mutex dummy(&mutex);
    if (t) rep.remove(t->getCEForController());
}

////////////////////////////////////
// class TimerInterrupt
////////////////////////////////////

// Timer interrupt entry:
const int TimerIntNo = 0;

class TimerInterrupt : public InterruptHandler {
protected:

    TimerInterrupt () : InterruptHandler(TimerIntNo,timerInterrupt) {}

    static void timerInterrupt () { instance->interruptHandler(); }
    virtual int handle () { TimerController::Instance()->tick(); return 1; }

private:

    static TimerInterrupt* instance;
};

TimerInterrupt* TimerInterrupt::instance = new TimerInterrupt;

////////////////////////////////////
// class Timer
////////////////////////////////////

RECYCLE_DEF(Timer);

Timer::Timer (Time t, Timeable* tmb1) : RECYCLE_CON(Timer),
myTimeable(tmb1), myThread(0),
counter(t), initial(t), isRunning(0),
mutex(1), ceForController(this) {

```



```

    if (myTimeable!=0) {
        myThread=new TimerThread(myTimeable);
        myThread->start();
    }
    TimerController::Instance()->sign(this);
}

Timer::~Timer () {
    mutex.wait();
    TimerController::Instance()->unsign(this);
    if (myThread!=0) myThread->destroy();
}

Time Timer::stop () {
    Mutex dummy(&mutex);
    isRunning=0;
    return initial-counter;
}

void Timer::restart (Time t) {
    Mutex dummy(&mutex);
    if (t!=0)
        counter=initial=t;
    else
        counter=initial;
    isRunning=1;
}

void Timer::tick () {
    Mutex dummy(&mutex);
    if (!isRunning) return;
    if (--counter==0) {
        isRunning=0;
        if (myThread!=0) myThread->signal();
    }
}

```

Obrada prekida

- Posao koji se obavlja kao posledica prekida logički nikako ne pripada niti koja je prekinuta, jer se u opštem slučaju i ne zna koja je nit prekinuta: prekid je za softver signal nekog asinhronog spoljašnjeg događaja. Zato posao koji se obavlja kao posledica prekida treba da ima sopstveni kontekst, tj. da se pridruži sporadičnom procesu, kao što je ranije rečeno.
- Osim toga, ne bi valjalo dopustiti da se u prekidnoj rutini, koja se izvršava u kontekstu niti koja je prekinuta, poziva neka operacija koja može da blokira pozivajuću nit.
- Drugo, značajno je da se u prekidnoj rutini vodi računa o tome kako dolazi do preuzimanja, ako je to potrebno.
- Treće, u svakom slučaju, prekidna rutina treba da završi svoje izvršavanje što je moguće kraće, kako ne bi zadržavala ostale prekide.
- Prema tome, opasno je u prekidnoj rutini pozivati bilo kakve operacije drugih objekata, jer one potencijalno nose opasnost od navedenih problema. Ovaj problem rešava se ako se na suštinu prekida posmatra na sledeći način.

- Prekid zapravo predstavlja obaveštenje (asinhroni signal) softveru da se neki događaj dogodio. Pri tome, signal o tom događaju ne nosi nikakve druge informacije, jer prekidne rutine po pravilu nemaju argumente. Sve što softver može da sazna o događaju svodi se na softversko čitanje podataka (eventualno nekih registara hardvera). Prema tome, prekid je *asinhroni signal događaja*.
- Navedeni problemi rešavaju se tako što se obezbedi jedan događaj koji će prekidna rutina da signalizira, i jedan proces koji će na taj događaj da čeka. Na ovaj način su konteksti prekinutog procesa (i sa njim i prekidne rutine) i sporadičnog procesa koji se prekidom aktivira potpuno razdvojeni, prekidna rutina je kratka jer samo obavlja signal događaja, a prekidni proces može da obavlja proizvoljne operacije posla koji se vrši kao posledica prekida.
- Ukoliko operativni sistem treba odmah da odgovori na prekid, onda operacija signaliziranja događaja iz prekidne rutine treba da bude sa preuzimanjem (engl. *preemptive*), pri čemu treba voditi računa kako se to preuzimanje vrši na konkretnoj platformi (maskiranje prekida, pamćenje konteksta u prekidnoj rutini i slično).
- Kod za opisano rešenje dato je u nastavku:

```
typedef unsigned int IntNo; // Interrupt Number

class InterruptHandler : public Thread {
protected:

    InterruptHandler (IntNo num, void (*intHandler)());

    virtual void run ();

    virtual int handle () { return 0; }
    void interruptHandler ();

private:

    Event ev;

};

void initIVT (IntNo, void (*)() ) {
    // Init IVT entry with the given vector
}

InterruptHandler::InterruptHandler (IntNo num, void (*intHandler)()) {
    // Init IVT entry num by intHandler vector:
    initIVT(num,intHandler);

    // Start the thread:
    start();
}

void InterruptHandler::run () {
    for(;;) {
        ev.wait();
        if (handle()==0) return;
    }
}
```

```
void InterruptHandler::interruptHandler () {
    ev.signal();
}
```

Zadaci

7.1 *Konstrukt Delay*

Pomoću raspoloživih koncepata školskog jezgra i korišćenjem apstrakcije `Timer`, potrebno je realizovati opisani konstrukt *delay* koji postoji u mnogim operativnim sistemima i programskim jezicima. Korisnički program može pozvati na bilo kom mestu operaciju `delay(Time)` koja suspenduje tekuću nit (u čijem se kontekstu ova operacija izvršava) na vreme dato argumentom. Posle isteka datog vremena, sistem sam (implicitno) deblokira datu nit. Data nit se može deblokirati i iz druge niti, eksplicitnim pozivom operacije `Thread::wakeUp()` date suspendovane niti. Navesti precizno koje izmene treba učiniti i gde u postojećem jezgru i dati realizaciju operacija `delay()` i `wakeUp()`.

Prikazati upotrebu ovog koncepta na primeru niti koje kontrolišu deset svetiljki koje se pale i gase naizmenično, svaka sa svojom periodom ugašenog i upaljenog svetla.

Rešenje

(a)

Izmene u klasi `Thread`:

```
class Thread : ..., public Timeable {
public:

    void wakeUp () { timeBlocking.signal(); }

protected:

    virtual void timeout () { timeBlocking.signal(); }

private:
    friend void delay (Time);
    Timer myTimer;
    Event timeBlocking;
};

Thread::Thread (...) : ..., myTimer(0,this) {...}

void delay (Time t) {
    Thread::running->myTimer.start(t);
    Thread::running->timeBlocking.wait();
}
```

(b)

```
class LightControl : public Thread {
public:
    LightControl (int num, Time period) : myNum(num), myPeriod(period) {}
protected:
    virtual void run ();
```

```
private:
    int myNum;
    Time myPeriod;
};

void LightControl::run () {
    while (1) {
        lightOn(myNum);
        delay(myPeriod);
        lightOff(myNum);
        delay(myPeriod);
    }
}

...
const int N = 10;
Time periods[N] = {...};
LightControl* lights[N];
for (int i=0; i<10; i++) {
    lights[i] = new LightControl(i, periods[i]);
    lights[i]->start;
}
```

7.2 Timer

Projektuje se optimizovani podsistem za merenje vremena u nekom RT operativnom sistemu. Podsistem se zasniva na konceptu vremenskog brojača realizovanog klasom `Timer` poput onog u postojećem školskom jezgru, ali u kome vremenski brojači služe samo za kontrolu isteka zadatog vremenskog intervala (*timeout*) i u kome je implementacija mehanizma praćenja isteka tog intervala drugačija. Vremenski brojači, kao objekti klase `Timer`, uvezani su u jednostruko ulančanu listu, uređenu neopadajuće prema trenutku isteka intervala koje brojači mere. Pritom, prvi objekat u listi (onaj kome vreme najpre ističe), u svom atributu čuva relativno vreme u odnosu na sadašnji trenutak za koje dolazi do isteka intervala koji meri, a ostali objekti klase `Timer` u tom svom atributu čuvaju samo relativno vreme u odnosu na prethodni brojač u listi (može biti i 0 ukoliko dva susedna brojača ističu u istom trenutku). Na primer, ukoliko brojači u listi imaju vrednosti ovog atributa redom: 1, 0, 0, 2, 5, onda to znači da prva tri brojača ističu za 1, sledeći za 3, a poslednji za 8 jedinica od sadašnjeg trenutka. Prema tome, prilikom smeštanja novog brojača u listu, on se umeće na odgovarajuću poziciju, a njegov atribut se podešava u odnosu na prethodnike, kao što je opisano. U svakom trenutku otkućaja sata realnog vremena, ažurira se samo prvi brojač u listi. Ukoliko pri tome prvi brojač u listi padne na nulu, vremenski istek se signalizira onim brojačima sa početka liste koji u svom atributu imaju nulu. Realizovati na jeziku C++ klasu `Timer` prema opisanom mehanizmu. Ne koristiti gotovu strukturu podataka za ulančanu listu, već tu strukturu inkorporirati u klasu `Timer`. Realizovati i dve ključne operacije ove klase:

- (a) `Timer::start(Time t)`: pokreće brojač sa zadatim vremenom isteka u odnosu na sadašnji trenutak.
- (b) `Timer::tick()`: statička funkcija klase koju spolja poziva prekidna rutina sata realnog vremena na svaki otkućaj (ne treba realizovati ovu prekidnu rutinu). Ova operacija treba da pokrene operaciju isteka vremena `timeout()` za one brojače koji su istekli.

Prilikom kreiranja, brojač se ne umeće u navedenu listu; tek prilikom startovanja se to radi. Nije potrebno realizovati nijednu drugu pomoćnu operaciju (npr. restart, očitavanje proteklog vremena itd.), već samo opisane operacije koje služe za kontrolu isteka zadatog intervala.

Takođe nije potrebno voditi računa o cepanju konteksta prilikom poziva operacije `timeout()` pridruženog objekta koji se krije iza interfejsa `Timeable`.

Rešenje

```
class Timer {
public:
    Timer (Time initial = maxTimeInterval, Timeable* timeable = 0);

    void start (Time time = 0);

protected:

    static void tick ();

private:
    Time initial;
    Time relative;
    Timeable* myTimeable;
    Timer* next;

    static Timer* head;
};

Timer* Timer::head = 0;

Timer::Timer (Time init, Timeable* timeable) :
    initial(init), relative(0), myTimeable(timeable), next(0) {}

void Timer::start (Time time) {
    this->relative = (time==0)?initial:time;

    Timer* nxt = head;
    Timer* prv = 0;
    while (nxt!=0 && this->relative >= nxt->relative) {
        this->relative -= nxt->relative;
        prv = nxt;
        nxt = nxt->next;
    }

    if (nxt) nxt->relative -= this->relative;
    this->next = nxt;
    if (prv!=0) prv->next = this;
    else head = this;
}

void Timer::tick () {
    if (head) head->relative--;
    while (head!=0 && head->relative<=0) {
        Timer* timer = head;
        head = head->next;
        timer->next = 0;
        if (timer->myTimeable!=0) timer->myTimeable->timeout();
    }
}
```

7.3 PeriodicTask

Korišćenjem školskog jezgra, potrebno je koncept *periodičnog posla* realizovati klasom `PeriodicTask`. Korisnik može da definiše neki periodični posao tako što definiše svoju klasu

izvedenu iz klase `PeriodicTask` i redefiniše njenu virtuelnu funkciju `step()` u kojoj navodi kod za posao koji treba da se uradi u svakoj periodi. Konstruktoru klase `PeriodicTask` zadaje se perioda posla tipa `Time`, a funkcijom `start()` ove klase pokreće se posao. Korišćenjem ovog koncepta dati kompletan program kojim se kontroliše n sijalica koje trepću periodično, svaka sa svojom periodom (periode su zadate u nekom nizu), pri čemu su poluperiode ugašenog i upaljenog svetla jednake. Sijalica se pali i gasi pozivima funkcije `lightOnOff(int lighNo, int onOff)`.

Rešenje

(I)

```
// PeriodicTask1.h

class PeriodicTask : public Thread {
public:
    PeriodicTask(Time per) : period(per) {}

protected:
    virtual void run();
    virtual void step() = 0;

private:
    Time period;
};

// PeriodicTask1.cpp

#include <PeriodicTask.h>

void PeriodicTask::run() {
    while(1) {
        step();
        delay(period);
    }
}
```

Problem: vreme izvršavanja `step()` metoda ne mora biti zanemarljivo malo, pa ovo rešenje pati od kumulativnog plivanja.

(II)

```
// PeriodicTask2.h

class PeriodicTask : public Thread {
public:
    PeriodicTask(Time per) : period(per) {}

protected:
    virtual void run();
    virtual void step() = 0;

private:
    Time period;
    Timer t;
};
```

```

void PeriodicTask::run() {
    Time t_start, t_end, sleepTime, t_drift = 0;

    t.start();
    while(1) {
        t_start = t.elapsed();
        step();
        t_end = t.elapsed();
        // execution time = t_end - t_start;
        sleepTime = period - (t_end - t_start) - t_drift;
        delay(sleepTime);
        t_drift = t.elapsed() - (t_end + sleepTime);
    }
}

```

Napomena: Ovo rešenje u generalnom slučaju (kada je podržano preuzimanje) ne rešava problem kumulativnog kašnjenja jer je moguća situacija da nit bude prekinuta (izgubi procesor) u toku računanja vremena uspavljivanja. Jedno rešenje ovog problema je korišćenjem poziva usluge okruženja `delayUntil(AbsoluteTime t)` koja uspavljuje nit do apsolutnog trenutka na vremenskoj osi, kao na jeziku Ada.

7.4 State

Potrebno je napisati program za kontrolu semafora za vozila sa tri svetla (crveno, žuto, zeleno). Semafor menja stanja na uobičajeni način: crveno, crveno-žuto, zeleno, žuto i tako ciklično. Vremena trajanja ovih stanja su TR, TRY, TG i TY, respektivno. Kada se semafor pokvari, treba da trepće žuto svetlo, sa jednakim poluperiodama upaljenog i ugašenog svetla jednakim TF; tada semafor trepće sve dok ne dođe serviser, otkloni kvar i ne resetuje semafor (semafor startuje od svog početnog stanja - start periode crvenog svetla).

Svetla se kontrolišu pozivom funkcije `light(Light which, int onOff)`; argument `which` (tipa `enum Light {R,G,Y}`) ukazuje na svetlo, a argument `onOff` da li svetlo treba upaliti ili ugastiti. Neispravnost semafora uzrokuje generisanje prekida. Osim toga, poseban prekid dolazi kada se iz kontrolnog centra vrši sinhronizacija svih semafora; tada semafor treba da pređe u unapred definisano stanje (koje se zadaje konfigurisanjem semafora).

Prikazati dijagram stanja semafora, a zatim realizovati sve potrebne klase za upravljanje semaforom po datom dijagramu stanja, korišćenjem školskog Jezgra.

Rešenje

Primena projektnog obrasca *State* implementirana je odgovarajuća mašina stanja.

```

////////////////////////////////////
// class State
////////////////////////////////////

class TrafficLight;

class State : public Object {
public:

    State (TrafficLight * fsm) : myFSM(fsm) {}

    virtual State* fix () { return this; }
    virtual State* err () { return this; }
    virtual State* sync() { return this; }
    virtual State* timeout () { return this; }

```

```

    virtual void entry () {}
    virtual void exit () {}

protected:

    TrafficLight* fsm () { return myFSM; }

private:

    TrafficLight* myFSM;

};

/////////////////////////////////////////////////////////////////
// classes StateR, StateG, StateRY, StateY
/////////////////////////////////////////////////////////////////

class StateR : public State {
public:
    StateR (TrafficLight* fsm) : State(fsm) {}
    virtual State* err ();
    virtual State* sync();
    virtual State* timeout ();
};

class StateG : public State {
public:
    StateG (TrafficLight* fsm) : State(fsm) {}
    virtual State* err () ;
    virtual State* sync() ;
    virtual State* timeout () ;
};

class StateRY: public State {
public:
    StateRY (TrafficLight* fsm) : State(fsm) {}
    virtual State* err () ;
    virtual State* sync() ;
    virtual State* timeout () ;
};

class StateY : public State {
public:
    StateY (TrafficLight* fsm) : State(fsm) {}
    virtual State* err () ;
    virtual State* sync() ;
    virtual State* timeout () ;
};

class StateE : public State {
public:
    StateA (TrafficLight* fsm) : State(fsm) {}
    virtual State* fix () ;
    virtual State* sync() ;
    virtual State* timeout () ;
};

```



```

////////////////////////////////////
// class TrafficLight
////////////////////////////////////

enum Event { TO, ERR, FIX, SIN };
enum Light { R, G, Y };
enum InitState { R, RY, G, Y };
const int TR=..., TRY=..., TG=..., TY=..., TF=...;

class TrafficLight : public Thread, public Timeable {
public:

    static TrafficLight* Instance ();
    setInitState (InitState init) { syncState = init; }

    void event (Event);

protected:

    virtual void run ();
    virtual void timeout ();

    TraficLight();

    friend class StateR;
    friend class StateG;
    friend class StateRY;
    friend class StateY;
    friend class StateE;

    void r() { light(Y,0); light(G,0); light(R,1); myTimer.start(TR); }
    void ry() { light(Y,1); light(G,0); light(R,1); myTimer.start(TRY); }
    void y() { light(Y,1); light(G,0); light(R,0); myTimer.start(TY); }
    void g() { light(Y,0); light(G,1); light(R,0); myTimer.start(TG); }
    void e() {
        YellowOn = YellowOn?0:1;
        light(Y, YellowOn); light(G,0); light(R,0); myTimer.start(TF);
    }
    void s() {
        switch (syncState) {
            case R: currentState = &stateR; break;
            case RY: currentState = &stateRY; break;
            case G: currentState = &stateG; break;
            case Y: currentState = &stateY; break;
        }
    }

private:

    StateR stateR;
    StateRY stateRY;
    StateY stateY;
    StateE stateE;
    StateG stateG;

    State* currentState;
    InitState syncState;

```

```

    Timer myTimer;
    MsgQueue que; // Implemented to accept Event

    int YellowOn;
};

TrafficLight * TrafficLight::Instance () {
    static TrafficLight instance;
    return &instance;
}

TrafficLight::TrafficLight() : syncState(R), stateR(this), stateRY(this),
stateY(this), stateE(this), stateG(this), currentState(&stateR),
myTimer(0,this), YellowOn(0) {}

void TrafficLight::event (Event e) {
    que.put(e);
}

void TrafficLight::timeout () {
    que.put(TO);
}

void X::run () {
    while (1) {
        Event e = que.get();
        currentState->exit();
        switch (e) {
            case TO: currentState = currentState->timeout(); break;
            case ERR: currentState = currentState->err(); break;
            case SYN: currentState = currentState->sync(); break;
            case FIX: currentState = currentState->fix(); break;
        }
        currentState->entry();
    }
}

////////////////////////////////////
// Implementation
////////////////////////////////////

State* StateR::timeout() {
    fsm()->ry();
    return &(fsm()->stateRY);
}

State* StateR::err() {
    fsm()->e();
    return &(fsm()->stateE);
}

State* StateR::sync() {
    fsm()->s();
    return &(fsm()->currentState);
}

// Analogno za StateG, StateRY, StateY

State* StateE::fix() {
    fsm()->r();
    return &(fsm()->stateR);
}

```

```

}

State* StateE::timeout() {
    fsm()->e();
    return &(fsm()->stateE);
}

////////////////////////////////////
// Interrupt handlers
////////////////////////////////////

// Timer interrupt entry:
const int ErrorIntNo = ...;

class ErrorInterrupt : public InterruptHandler {
protected:

    ErrorInterrupt () : InterruptHandler(ErrorIntNo,void (*errorInterrupt)())
    {}

    static void errorInterrupt () { instance->interruptHandler(); }

    virtual int handle () {
        // User-defined code for one release of the sporadic process
        TrafficLight::Instance()->event(ERR);
    }

private:

    static ErrorInterrupt* instance;

};

ErrorInterrupt * ErrorInterrupt::instance = new ErrorInterrupt;
// Analogno za FIX i SYN

```

7.5 A/D konverzija

Na ulaz računara vezan je A/D konvertor koji dati analogni signal konvertuje u digitalnu vrednost periodično, sa dovoljno velikom frekvencijom da se sa strane softvera konvertovana digitalna vrednost može smatrati kontinualno dostupnom. Konvertovana digitalna vrednost očitava se funkcijom `readValue():double`. Ovaj signal treba modulisati primenom adaptivne delta modulacije na sledeći način. Vrednost na ulazu treba očitavati (uzimati odbirke) u vremenskim intervalima čija je veličina obrnuto srazmerna veličini promene signala (veća promena – češće odabiranje). Preciznije, ako je ΔA vrednost promene signala (razlika poslednje i preposlednje učitane vrednosti), onda narednu vrednost treba očitati kroz $\Delta T = T_k * (A_k / \Delta A)$, gde su T_k i A_k odgovarajuće konstante. Modulirana vrednost ΔA predstavlja razliku poslednje i preposlednje učitane vrednosti. Tako modulirane vrednosti ΔA treba slati periodično, sa periodom T_s , na izlaz računara. Jedna vrednost šalje se na izlaz funkcijom `sendValue(double)`. Korišćenjem školskog jezgra napisati program na jeziku C++ koji obavlja opisanu modulaciju.

Rešenje

```

typedef double Amplitude;
const Time Ts = ..., Tk = ...;
const Amplitude Ak = ...;

class Transmitter : public Timeable {
public:
    static Transmitter* Instance ();
    void send (Amplitude);
protected:
    Transmitter ();
    virtual void timeout ();
private:
    Timer myTimer;
    MsgQueue que; // Implemented to accept Amplitude
};

Transmitter* Transmitter::Instance () {
    static Transmitter instance;
    return &instance;
}

Transmitter::Transmitter () : myTimer(Ts,this) {
    myTimer.start();
}

void Transmitter::send (Amplitude a) {
    que.put(a);
}

void Transmitter::timeout () {
    Amplitude a = que.get();
    sendValue(a);
    myTimer.restart();
}

class Modulator : public Timeable {
public:
    static Modulator* Instance ();
protected:
    Modulator ();
    virtual void timeout ();
private:
    Timer myTimer;
    Time period;
    Amplitude oldReading;
};

Modulator* Modulator::Instance () {
    static Modulator instance;
    return &instance;
}

Modulator::Modulator () : myTimer(Tk,this), period(Tk), oldReading(Ak) {
    myTimer.start();
}

void Modulator::timeout () {
    Amplitude newReading = readValue();

```

```

    Amplitude delta = newReading - oldReading;
    oldReading = newReading;
    Transmitter::Instance()->send(delta);
    period = Tk*Ak/delta;
    myTimer.restart(period);
}

```

7.6 Timed Semaphore

Korišćenjem školskog jezgra, realizovati klasu `TimedSemaphore` koja obezbeđuje logiku standardnog semafora, ali uz vremenski ograničeno čekanje.

Rešenje

```

class TimedSemaphore : protected Semaphore {
public:
    TimedSemaphore (int initValue=1) : Semaphore(initValue) {}
    virtual ~TimedSemaphore ();

    void wait      (Time timeout) throws TimeoutException;
    void signal    () { Semaphore::signal(); }

private:
    friend SemaphoreTimer;
};

class SemaphoreTimer : public Timer, public Timable {
public:
    SemaphoreTimer (Time initial, Thread* t, TimedSemaphore* s):
        Timer(initial,this), owner(t), timedOut(0), sem(s) {}
    virtual ~SemaphoreTimer ();

    int counted () { return timedOut; }

protected:
    virtual void timeout();

private:
    TimedSemaphore* sem;
    Thread* owner;
    int timedOut;
};

void TimedSemaphore::wait (Time timeout) throws TimeoutException {
    lock();

    SemaphoreTimer timer(timeout,Thread::runningThread,this);
    if (--val<0){
        timer.start();
        block();
    }

    timer.stop();
    if (timer.counted()){
        unlock();
        throw TimeoutException();
    }
}

```

```

    unlock();
}

void SemaphoreTimer::timeout () {
    timeOut = 1;

    // Deblocking:
    lock();
    sem->val++;
    Thread* t = (Thread*) sem->blocked->remove(owner->getCEForScheduler());
    Scheduler::Instance()->put(t);
    unlock();
}

```

Primer upotrebe:

```

...
try {
    notFull.wait(timeout);
}
catch(TimeoutException){
    //... Code for error recovery
}

```

7.7 Sigurnosni kočioni sistem

U nekom brzom vozu postoji sistem za nadzor ispravnosti kočionog sistema koji aktivira rezervni, sigurnosni kočioni sistem u slučaju otkaza glavnog kočionog sistema. Ovaj kontrolni sistem treba da proveri da li po aktiviranju glavne kočnice voz počinje da usporava u roku od $t_d = 0,5$ s od trenutka aktiviranja kočnice. Ovo kašnjenje od t_d podešeno je prema inerciji voza i osetljivosti senzora ubrzanja/usporenja voza, što znači da rezervnu kočnicu ne treba aktivirati u tom roku čak i ako se ne detektuje usporenje voza. Ako detektuje da voz ne usporava nakon isteka tog roka, ovaj sistem treba da aktivira pomoćni kočioni sistem u roku od $t_c = 0,7$ s nakon što je aktivirana glavna kočnica.

Softver ovog kontrolnog sistema implementiran je kao jedan periodičan proces koji je stalno aktivan i koji radi na sledeći način. Ako u nekoj aktivaciji prvi put pronade da je glavna kočnica aktivirana, ne radi ništa u toj i u još nekoliko narednih aktivacija (ukupno njih n), kako bi obezbedio protok vremena t_d , a onda u $n+1$ -oj aktivaciji, ukoliko pronade da voz i dalje ne usporava a glavna kočnica je još uvek aktivirana, aktivira rezervnu kočnicu.

Na raspolaganju su sledeće funkcije interfejsa prema uređajima:

<code>isMainBrakeOn()</code> : Boolean	Vraća True ako je glavna kočnica aktivirana
<code>isDecelerating()</code> : Boolean	Vraća True ako je detektovano usporenje voza
<code>emergencyBrakeOn()</code>	Uključuje pomoćnu kočnicu.

(a)(10) Na jeziku Ada implementirati ovaj proces tako da ne pati od kumulativnog plivanja (engl. *cumulative drift*), uzimajući periodu kao simboličku konstantu.

(b)(10) Napisati i objasniti nejednakosti koje predstavljaju uslove za periodu T , vremenski rok D i broj aktivacija n u funkciji t_d i t_c , a potom odrediti ove parametre (T , D i n).

Rešenje

a)

```

task body EmergencyBrakeTask is
  nextActivation : Time;
  period : constant Time_Span := Milliseconds(...);
  i : Natural := 1;
  N : constant Natural := ...;
begin
  nextActivation := Clock + period;
  loop
    delay until nextActivation;
    nextActivation := nextActivation + period;

    if isMainBrakeOn() then
      if i < N+1 then
        i := i+1;
      else
        if not isDecelerating() then
          i := 1;
          emergencyBrakeOn();
        end if;
      end if;
    else
      i:=1;
    end if;
  end loop;
end EmergencyBrakeTask;

```

b) Prvi uslov zahteva da od trenutka kada se aktivira glavna kočnica, do trenutka kada sistem odluči da treba aktivirati pomoćnu kočnicu jer voz i dalje ne usporava, prođe najmanje t_d vremena. Najgori scenario za ovaj zahtev jeste taj da se u prvoj aktivaciji izvršavanje procesa dogodi najkasnije moguće, dakle neposredno pred istek vremenskog roka D , pri čemu se aktivacija glavne kočnice dogodila neposredno pre toga, a u $n+1$ -voj, kada sistem donosi ovu odluku, izvršavanje obavi odmah po aktivaciji, na početku periode. Odatle sledi:

$$(n-1)T + (T-D) \geq t_d$$

odnosno:

$$nT - D \geq t_d$$

Drugi zahtev je taj da od trenutka aktivacije glavne kočnice do trenutka aktivacije pomoćne kočnice prođe najviše t_c vremena. Nagori scenario za ovaj uslov je taj da se izvršavanje u prvoj aktivaciji i izvršavanje u $n+1$ -voj aktivaciji maksimalno udalje:

$$nT + D \leq t_c$$

Odatle sledi:

$$t_d + D \leq nT \leq t_c - D$$

Ako se uslovi maksimalno relaksiraju, odnosno dozvoli $D = T$ i nejednakosti pretvore u jednakosti, oduzimanjem dve jednačine po n i $T = D$ se dobija:

$$2T = t_c - t_d$$

odnosno:

$$T = D = \frac{t_c - t_d}{2} = 0,1s$$

$$n = \frac{t_d}{T} + 1 = 6$$

Zadaci za samostalan rad

7.8

Potrebno je proširiti školsko jezgro tako da podržava detekciju prekoračenja roka i prekoračenja WCET. Prodiskutovati:

- (a) Kako bi se u aplikativnom (korisničkom) kodu specifikovala ova ograničenja i koje su modifikacije jezgra potrebne za prihvatanje ovih specifikacija?
- (b) Kako modifikovati jezgro da bi se obezbedila detekcija ovih prekoračenja? Kako dojaviti ovaj otkaz korisničkoj aplikaciji?
- (c) Prikazati primerom način korišćenja predloženih koncepata.

7.9

Korišćenjem klase `TimedSemaphore` iz ranijeg zadatka, prikazati realizaciju ograničenog bafera u školskom jezgru, pri čemu su vremenske kontrole pridružene i čekanju na ulaz u kritičnu sekciju i čekanju na uslovnu sinhronizaciju. Ilustrovati upotrebu ovakvog bafera primerom proizvođača i potrošača.

7.10

Korišćenjem školskog jezgra, realizovati klasu `TimeoutActivity` čiji je interfejs prema izvedenim klasama i klijentima dat. Ova klasa predviđena je za izvođenje, tako da aktivna korisnička izvedena klasa može da pokrene neku aktivnost sa zadatom vremenskom kontrolom (*timeout*). Kada pokrene datu aktivnost, korisnička klasa poziva funkciju `waitWithTimeout()` sa zadatim vremenom. Pozivajuća korisnička nit se tada blokira. Kraj aktivnosti neki klijent zadaje spolja pozivom funkcije `endActivity()`. Tada se korisnička nit deblokira. Korisnička klasa može posle toga ispitati poreklo deblokade pozivom funkcije `status()` koja vraća `EOA` u slučaju okončanja aktivnosti pre isteka vremenske kontrole, odnosno `TO` u slučaju isteka roka pre okončanja aktivnosti. Predvideti da se kraj aktivnosti može signalizirati i posle isteka roka.

```
class TimeoutActivity {
public:

    void endActivity ();

protected:

    TimeoutActivity ();
    enum Status { NULL, EOA, TO };

    void waitWithTimeout (Time timeout);
    Status status ();

};
```

Prikazati upotrebu ovog koncepta na sledećem primeru. Program treba da posle slučajnog intervala vremena u opsegu $[0..T_1]$ (slučajan broj u opsegu $[0..1]$ dobija se pozivom bibliotečne funkcije `rnd()`) ispiše korisniku poruku "Brzo pritisni taster!" Posle toga program očekuje da korisnik pritisne taster (pritisak na taster generiše prekid koji treba u programu obraditi) u roku od T_2 jedinica vremena. Ukoliko korisnik pritisne taster u tom roku, program ispisuje poruku "Ala si brz, svaka čast!" i ponavlja isti postupak posle slučajnog vremena. Ukoliko ne pritisne, program ispisuje poruku "Mnogo si, brate, spor!" i ponavlja

postupak. Pretpostaviti da je na raspolaganju biblioteka funkcija `delay(Time t)` kojom se pozivajuća nit blokira na vreme `t`.

7.11

Korišćenjem školskog jezgra, realizovati apstraktnu klasu `MultipleTimeouts` koja je namenjena za nasleđivanje i čiji je interfejs prema izvedenim klasama dat u nastavku. Klasa je predviđena da podrži višestruku kontrolu do N kontrolnih intervala vremena (*timeout*). Iz koda izvedene klase može se startovati pojedinačno merenje kontrolnog vremena pozivom operacije `startTiming()`, čiji parametar `i` ukazuje na to koji od N intervala se startuje ($0 \leq i < N$), a parametar `time` zadaje kontrolni vremenski interval. Merenje i -tog intervala može se zaustaviti pozivom operacije `stopTiming()`. Kada i -ti interval istekne, sistem treba da pozove apstraktnu operaciju `timeout()` čiji parametar ukazuje na to da je istekao interval i . Obezbediti da se eventualni istovremeni signali isteka više intervala sekvencijalizuju, tj. međusobno isključe.

```
const int N = ...;

class MultipleTimeouts {
protected:
    void startTiming (int i, Time time);
    void stopTiming  (int i);
    virtual void timeout (int i) = 0;
};
```

7.12

Svaki od više objekata klase `Client` periodično, sa svojom periodom koja mu se zadaje pri kreiranju, upućuje asinhronu poruku `accept()` sa sadržajem tipa `int` *Singleton* objektu klase `Server`. Ovaj objekat obrađuje pristigle poruke tako što njihov sadržaj ispisuje redom na ekran. Sadržaji poruka su brojevi koji označavaju redni broj poruke svakog klijenta. Realizovati klase `Client` i `Server` korišćenjem školskog jezgra. Dati definicije klase, a u glavnom programu konfigurisati sistem tako da poseguje N objekata klase `Client` i pokrenuti rad ovih objekata.

7.13

Klasa `Poller` je *Singleton*. Njen jedini objekat je zadužen da periodično proziva sve prijavljene objekte klase `DataSource` tako što poziva njihovu apstraktnu operaciju `poll()`. Perioda prozivanja treba da bude konstanta u programu koja se lako definiše i menja. Klasa `DataSource` je apstraktna. Konkretno izvedene klase treba da definišu funkciju `poll()`. Svaki objekat ove klase prilikom kreiranja prijavljuje se objektu funkcijom `sign(DataSource*)` i odjavljuje prilikom gašenja funkcijom `unsign(DataSource*)`. Realizovati klase `Poller` i `DataSource` korišćenjem školskog jezgra.

7.14

Mnogi digitalni uređaji, npr. muzički aparati ili mobilni telefoni, zasnivaju svoju interakciju sa korisnikom prilikom podešavanja funkcionalnosti na sledećem principu. Ako korisnik kratko pritisne neki taster, onda taj taster ima jednu funkciju. Ako pak korisnik nešto duže drži pritisnut isti taster (recimo više od 2 sec), onda taster ima drugačiju funkciju. Korišćenjem školskog jezgra, napisati na jeziku C++ klasu čiji je interfejs prema izvedenoj klasi prikazan dole. Ova klasa služi kao osnovna apstraktna klasa iz koje se može izvesti neka konkretna

klasa koja definiše dve funkcije nekog tastera. Konkretna izvedena klasa treba da bude *Singleton*, vezana za jedan taster. Svaki pritisak ili otpuštanje tastera generiše prekid; informacija o tome da li se radi o pritisku ili otpuštanju tastera može se dobiti očitavanjem zadatog registra (vrednost $\neq 0$ znači pritisak, a $= 0$ znači otpuštanje). Broj prekida kome se pridružuje izvedena klasa i adresa porta sa koga se čita navedena informacija parametri su konstruktora osnovne klase. Granično vreme koje određuje funkciju pritiska tastera određeno je takođe parametrom konstruktora, a može se i posebno postaviti odgovarajućom funkcijom `setTime()` ili očitati funkcijom `getTime()`. Konkretna izvedena klasa može da redefiniše dve virtualne funkcije `onShortPress()` i `onLongPress()` koje se pozivaju ako je detektovan kratki, odnosno duži pritisak, respektivno.

```
class DblFunKey {
protected:

    DblFunKey (IntNo interruptNo, int portAddr, Time borderTime = 0);

    void setTime (const Time& borderTime);
    Time getTime ();

    virtual void onShortPress () {};
    virtual void onLongPress  () {};
};
```

7.15

Korišćenjem školskog jezgra, potrebno je realizovati podsistem kojim se može izmeriti prosečna perioda pojave prekida na ulazu `IntN`. Uslugu ovog merenja korisnik može da dobije pozivom funkcije:

```
Time getIntPeriod (Time timeInterval, Time maxPeriod);
```

Ova funkcija startuje merenje broja prekida koji se dogode za zadato vreme `timeInterval`. Argumentom `maxPeriod` se zadaje maksimalno očekivano rastojanje između pojave dva uzastopna prekida. Funkcija vraća količnik proteklog vremena merenja `timeInterval` i broja prekida koji su se dogodili. Ukoliko se desi da u roku od `maxPeriod` posle jednog prekida vremena ne stigne novi prekid, ova funkcija vraća 0. Korisnički proces se blokira unutar ove funkcije sve do završetka merenja.

7.16

Korišćenjem školskog jezgra, realizovati klasu `InterruptTimer` koja je zadužena da meri vremenski interval između dva susedna prekida i da maksimalnu zabeleženu vrednost tog intervala čuva u svom atributu. Pretpostavlja se da maksimalna dužina tog intervala može biti znatno veća od maksimalne dužine intervala koji može da meri objekat sistemske klase `Timer` pre nego što generiše signal *timeout*.

7.17

U sistemu postoje spoljni događaji koji se registruju prekidima. Za svaki događaj potrebno je izvršiti odgovarajući posao predstavljen prekidnim, sporadičnim procesom. Posmatrano na velikom intervalu vremena, prosečan vremenski razmak između događaja dovoljno je veliki da sistem uspe da odradi prekidni proces. Međutim, na kraćem vremenskom intervalu može se desiti da vremensko rastojanje između nekoliko događaja bude dosta kraće od vremena potrebnog da se obavi prekidni proces. Sistem treba u svakom slučaju da za svaki događaj koji se desio obavi po jedan ciklus prekidnog procesa (sekvencijalno jedan po jedan ciklus, za

svaki događaj), predstavljen jednim pozivom funkcije `handle()` koju obezbeđuje korisnik. Za ovakve potrebe, treba realizovati klasu `InterruptHandler` koja će biti nalik na onu koja je data u školskom jezgru, ali modifikovanu tako da zadovolji navedene zahteve. Osim toga, potrebno je da ova klasa obezbedi i zaštitu od preopterećenja sistema na sledeći način. Ukoliko se registruje da je broj trenutno neobrađenih događaja prešao odgovarajuću zadatu kritičnu granicu, treba da aktivira odgovarajući proces koji korisnik treba da obezbedi za tu namenu. Ovaj proces može, recimo, da ukloni spoljašnje uzroke prečestog nastajanja događaja. Realizovati ovako modifikovanu klasu `InterruptHandler` korišćenjem školskog jezgra.

7.18

Neki prekid u sistemu stiže sporadično. Kada stigne prekid, potrebno je posle 10 jedinica vremena signalizirati semafor koji je pridružen tom prekidu. Ukoliko prekid ne stigne u roku od 50 jedinica vremena od prethodnog prekida, potrebno je uključiti sijalicu koja trepće periodično. Kada stigne novi prekid, potrebno je isključiti treptanje. Sijalica se pali ili gasi pozivom funkcije `lightOnOff(int onOff)`. Realizovati ovo korišćenjem školskog jezgra.

7.19

Tri tastera su vezana tako da svaki pritisak na bilo koji od njih generiše isti prekid procesoru. U svakom trenutku, u bitima 2..0 registra koji je vezan na 8-bitni port TAST nalazi se stanje tri tastera (1-pritisnut, 0-otpušten). Ovaj sistem koristi se za suđenje boks-meča, kod kojeg tri sudije prate meč i broje udarce koje zadaje jedan bokser. Kada se dogodi udarac, svaki sudija pritiska svoj taster ukoliko je primetio taj udarac i smatra da ga treba odbrojati. Kako sudije mogu da pritiskaju tastere ili istovremeno (teoretski), ili sa vremenskom zadržkom, ali i tako da jedan još uvek drži taster dok je drugi izvršio pritisak, sistem treba da se ponaša na sledeći način. U periodu od 500 ms od prvog pritiska tastera, svaki pritisak jednog sudije smatra se za isti registrovani udarac. Dakle, eventualni ponovni pritisci ili duže držanje tastera jednog sudije u datom periodu se ignorišu. Posle isteka datog perioda od 500 ms, novi pritisak nekog tastera smatra se početkom novog perioda u kome se prate tastere (smatra se da sudija ne može da registruje dva različita udarca u razmaku manjem od 500 ms). Pošto se može dogoditi da neki sudija greškom pritisne taster, ili da neki sudija ne primeti udarac, odluka se donosi većinskom logikom: ako i samo ako je u navedenom periodu od 500 ms registrovano da su dvojica ili trojica sudija pritisnuli svoje tastere, udarac se broji. Treba obezbediti i kontrolu trajanja runde od 3 min: pritisci na tastere se uzimaju u obzir samo tokom trajanja runde. Realizovati ovaj sistem na jeziku Ada.

7.20

Svetlosna dioda (LED) se pali i gasi pozivom funkcije `ledOnOff(int onOrOff)`, a na računar je vezan i taster koji generiše prekid procesoru kada se pritisne. Na jeziku Ada potrebno je realizovati sistem za merenje motoričke sposobnosti korisnika. Sistem treba da, kada se pokrene pozivom operacije `start()`, upali svetlosnu diodu i drži je upaljenu 10 jedinica vremena. U toku tog perioda, dok je dioda upaljena, korisnik treba da što brže i češće pritiska i otpušta taster, a sistem treba da broji koliko puta je taster pritisnut. Kada se dioda ugasi, svi dalji pritisci na taster se ignorišu. Nakon 5 jedinica vremena od gašenja diode, ukoliko je broj pritiskova veći od zadatog broja N , sistem treba da upali diodu još jednom i drži je upaljenu 5 jedinica vremena, kao znak da je korisnikova motorika zadovoljavajuća. Ukoliko je broj pritiskova manji ili jednak N , diodu više ne treba paliti.

7.21

Svetlosna dioda (LED) se pali i gasi pozivom funkcije `ledOnOff(int onOrOff)`, sirena se pali i gasi pozivom funkcije `alarmOnOff(int onOrOff)`, a na računar je vezan i taster koji generiše prekid procesoru kada se pritisne. Pomoću ovakvog sistema kontroliše se budnost mašinovođe. Sistem treba na svakih 5 min da upali diodu. Ukoliko mašinovođa pritisne taster u roku od 5 s od uključanja diode, sistem se "povlači" i ponavlja postupak za 5 min. Ukoliko mašinovođa ne pritisne taster u roku od 5 s, sistem treba da uključi sirenu, koja treba da radi sve do pritiska na taster. Voditi računa da se eventualni pritisak na taster pre nego što je dioda uključena ignoriše. Ponašanje ovog sistema modelovati mašinom stanja i nacrtati taj model, a zatim realizovati ovaj sistem na jeziku Ada.

Raspoređivanje i rasporedivost

Pojam raspoređivanja i rasporedivosti

- Kao što je do sada razmatrano, logička ispravnost konkurentnog programa ne zavisi od preciznog redosleda izvršavanja konkurentnih procesa ili njihovih delova. Zbog toga nije ni neophodno definisati taj precizan redosled, osim potrebnih lokalnih ograničenja u pogledu redosleda izvršavanja ili preplitanja delova konkurentnih procesa, koja se definišu odgovarajućimn sinhronizacijama u konkurentnom programu.
- Na taj način izvršavanje konkurentnog programa i dalje podrazumeva značajnu dozu proizvoljnosti u redosledu izvršavanja. Na primer, n nezavisnih procesa bez ikakve međusobne sinhronizacije, aktiviranih u istom trenutku, se na jednom procesoru bez preuzimanja mogu izvršiti na $n!$ načina, odnosno redosleda (permutacija). U opštem slučaju, uz postojanje mogućnosti asinhronne promene konteksta i zavisnosti između procesa, skup načina na koji se procesi mogu redom izvršavati brzo postaje ogroman. Ako je program korektan, onda će on dati isti i ispravan izlaz bez obzira na detalje implementacije izvršnog okruženja i nepredvidivost događaja koji određuju ovaj redosled.
- Međutim, *vremensko* ponašanje konkurentnog programa može značajno da zavisi od redosleda izvršavanja procesa. Za navedeni primer n nezavisnih procesa aktiviranih u istom trenutku, ukoliko neki od tih procesa ima tesan vremenski rok, nije sasvim svejedno kojim će se od $n!$ redosleda oni izvršiti, jer će možda samo neki od tih redosleda, i to oni u kojima su procesi sa tesnim rokovima izvršeni ranije, dovesti do ispunjenja svih vremenskih rokova i korektnog ponašanja RT programa, dok će ostali redosledi biti nekorektni u tom smislu. Zbog toga je kod RT sistema važno ograničiti proizvoljnost izvršavanja konkurentnih programa.
- Postupak kojim se definiše redosled izvršavanja konkurentnih procesa naziva se *raspoređivanjem* (engl. *scheduling*). Raspoređivanje predstavlja algoritam za definisanje redosleda korišćenja ograničenih sistemskih resursa (uglavnom procesorskog vremena) od strane konkurentnog programa.
- Za dati algoritam raspoređivanja, potrebno je posedovati sredstvo kojim se može predvideti ponašanje sistema i to u *najgorem slučaju* (engl. *worst case*) i pokazati da će vremenski zahtevi, prvenstveno vremenski rokovi, biti ispoštovani i u tom najgorem slučaju, pa samim tim i u svakom drugom, povoljnijem slučaju. Mogućnost zadovoljenja vremenskih zahteva za dati skup procesa i algoritam raspoređivanja u svakom, pa i najgorem mogućem slučaju, naziva se *rasporedivost* (engl. *schedulability*).
- Skup procesa je *rasporediv* (engl. *schedulable*) na datom ograničenom skupu resursa, ukoliko se može definisati raspored izvršavanja tih procesa koji zadovoljava sva postavljena vremenska ograničenja, u prvom redu vremenske rokove.
- Tehnika kojom se može ispitati rasporedivost za dati skup procesa i dati algoritam raspoređivanja naziva se *test rasporedivosti* (engl. *schedulability test*).
- Rasporedivost je ključni problem kod *hard* RT sistema i predstavlja centralnu temu ovog poglavlja.
- Ovde će biti razmatrano raspoređivanje i rasporedivost procesa na jednom procesoru. Raspoređivanje na više procesora daje više stepeni slobode i postaje značajno kompleksnije. Ukoliko se skup procesa podeli na disjunktne podskupove procesa između kojih nema interakcije, ti podskupovi se onda mogu dodeliti različitim procesorima i na svaki od njih sprovesti ovde opisane tehnike.

Jednostavan model procesa

- Analizu rasporedivosti nije nimalo jednostavno izvoditi za složene sisteme konkurentnih procesa i algoritme raspoređivanja.
- Zato će ovde biti uveden najpre jedan veoma jednostavan model RT programa, sa nizom pojednostavljenja i ograničenja koja omogućavaju jednostavnu teorijsku analizu. Taj jednostavan i apstraktan model će kasnije biti uopštavan postepenim ukidanjem pojedinih pojednostavljenja i ograničenja.
- Čak i sa navedenim vrlo velikim ograničenjima, i ovaj jednostavan model je često primenjiv na male skupove jednostavnih i nezavisnih procesa, kakva su ponekad *hard* jezgra RT sistema: iako je čitav sistem možda mnogo složeniji, kritično jezgro tog sistema se može sastojati od svega nekoliko veoma jednostavnih procesa koji zadovoljavaju ova ograničenja, ali koji imaju *hard* RT zahteve, pa se na njega mogu primeniti teorijski rezultati koji važe za ovaj model.
- Ovaj model podrazumeva sledeća ograničenja:
 - Program se sastoji iz konačnog i fiksnog skupa procesa.
 - Svi procesi imaju poznato i fiksno vreme izvršavanja u najgorem slučaju (WCET).
 - Svi procesi su periodični, sa unapred poznatom periodom.
 - Svi procesi imaju rok (engl. *deadline*) jednak svojoj periodi ($D = T$); to znači da svaki periodični proces mora da se završi pre početka naredne periode.
 - Procesi su međusobno potpuno nezavisni, odnosno ne interaguju međusobno (ne sinhronizuju se, ne pristupaju deljenim resursima).
 - Sva režijska vremena, kao što je vreme promene konteksta, se zanemaruju.
- Jedna posledica nezavisnosti procesa je to da se može pretpostaviti da će u nekom trenutku svi periodični procesi biti istovremeno aktivirani, tj. da će se trenuci njihovih periodičnih aktivacija nekad poklopiti i svi oni zajedno postati spremni za izvršavanje. Taj trenutak naziva se *kritični trenutak* (engl. *critical instant*) i predstavlja najnepovoljniji scenario, jer zadaje maksimalno opterećenje procesora.

Ciklično izvršavanje

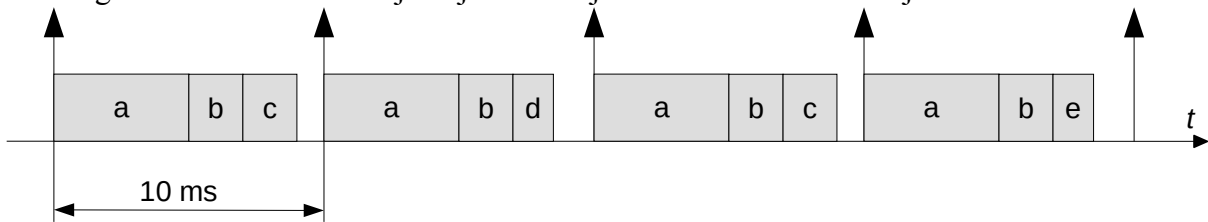
- Jedan krajnje jednostavan, ali ipak u praksi i dalje često korišćen način raspoređivanja u *hard* RT sistemima, jeste tzv. *ciklično izvršavanje* (engl. *cyclic executive*).
- Iako se sistem konstruiše kao konkurentan program, njegov fiksni skup isključivo periodičnih procesa se raspoređuje tako da se ti procesi izvršavaju ciklično, kao najobičnije procedure. Te procedure se preslikavaju na tzv. *male cikluse* (engl. *minor cycle*) koji opet sačinjavaju tzv. *veliki ciklus* (engl. *major cycle*).
- Na primer, neka se sistem sastoji od sledećih periodičnih procesa sa zadatim periodama T i vremenom izvršavanja (WCET) označenim sa C :

Proces	T [ms]	C [ms]
<i>a</i>	10	5
<i>b</i>	10	2
<i>c</i>	20	2
<i>d</i>	40	1
<i>e</i>	40	1

- Takt (npr. periodičan prekid od tajmera) nailazi svake male periode, u ovom slučaju svakih 10 ms, i predstavlja trenutak aktivacije jedne obrade koja se sastoji od nekoliko

poziva onih procedura koje su predviđene u tom malom ciklusu. Ceo veliki ciklus se sastoji od nekoliko malih ciklusa, u ovom primeru četiri.

- U ovom primeru, perioda malog ciklusa može da bude 10 ms, a velikog 40 ms, a jedno moguće ciklično izvršavanje koje zadovoljava vremenske zahteve je sledeće:



- *Ovakvo izvršavanje zadovoljava vremenske zahteve jer je svaki od procesa aktiviran tačno jednom u svakoj svojoj periodi T , a u svakom malom ciklusu mogu da se izvrše svi u nju raspoređeni procesi čak i u slučaju da svi traju koliko mogu da traju u najgorem slučaju (WCET).*
- Kod koji obezbeđuje ovakvo izvršavanje je:

```
loop
  wait_for_activation;
  procedure_a;
  procedure_b;
  procedure_c;
  wait_for_activation;
  procedure_a;
  procedure_b;
  procedure_d;
  wait_for_activation;
  procedure_a;
  procedure_b;
  procedure_c;
  wait_for_activation;
  procedure_a;
  procedure_b;
  procedure_e;
end loop;
```

- Neke važne osnovne karakteristike ovog pristupa jesu sledeće:
 - Raspored je potpuno nepromenljiv i unapred određen, pa nema nikakvih neodređenosti u vreme izvršavanja.
 - Ukoliko se može konstruisati ovaj raspored tako da zadovolji vremenske zahteve, rasporedivost je ispunjena samom konstrukcijom sistema i ne treba je posebno dokazivati. Rasporedivost je krajnje predvidiva i veoma otporna na zanemarivanja u procenama.
 - U vreme izvršavanja zapravo ne postoje stvarni, uporedni procesi; svaki mali ciklus je samo obična sekvenca poziva procedura.
 - Procedure dele zajednički adresni prostor preko koga mogu da razmenjuju podatke. Ti podaci ne moraju da budu zaštićeni međusobnim isključenjem, jer zapravo nema konkurentnog pristupa, pa nema ni konflikata.
 - Sve periode procesa moraju da budu umnošci periode malog ciklusa: perioda malog ciklusa mora biti delilac (recimo najveći zajednički delilac) periode svakog od procesa.
 - Režijski troškovi u vreme izvršavanja su minimalni, jer zapravo nema izvršavanja nikakvog algoritma raspoređivanja niti promene konteksta, već ima samo prostih poziva potprograma.
- Nedostaci ovog pristupa su:

- Navedeno ograničenje u veličini periode malog ciklusa može biti problem u konstrukciji.
- Teško je uklopiti procese sa veoma velikim periodama, jer to onda zahteva veoma dugačak veliki ciklus.
- Proces sa velikim vremenom izvršavanja (WCET) mora da se podeli na komade fiksne i manje veličine, što smanjuje preglednost programa i otežava njegovo održavanje: procedure koje obavljaju delove poslova ovog procesa moraju da čuvaju svoje stanje (kontekst) između susednih poziva, kako bi prekinule i nastavile svoje izvršavanje. Ovo otežava njihovo programiranje, a potom i održavanje u slučaju promene implementacije.
- Uopšte, teško je konstruisati, a potom i održavati raspored kada dođe do promene u konstrukciji procesa (recimo, promeni se WCET nekog procesa zbog promene njegove implementacije).
- Teško je uklopiti sporadične procese u raspored.
- Iako je ovo jednostavan i dosta korišćen pristup za jednostavnije podsisteme procesa, njegovi nedostaci ukazuju na potrebu za opštijim pristupima koji ne moraju biti tako strogo i fiksno određeni, ali i dalje moraju obezbediti predvidivost.

Raspoređivanje procesa

- Ciklično izvršavanje nije dovoljno opšte i ne podržava neposredno koncept konkurentnih procesa. Raspoređivanje treba zato direktno da podrži koncept procesa i da u svakom trenutku preuzimanja obezbedi odluku o tome koji će proces biti sledeći izvršavan.
- U opštem slučaju, algoritam raspoređivanja (procesa na procesoru) deluje svaki put kada se vrši promena konteksta. Promena konteksta može se vršiti *sinhrono*, kao posledica operacije koju je izvršio tekući proces, ili *asinhrono*, nezavisno od toga šta radi tekući proces koji se izvršava i u potencijalno proizvoljnom i nepredvidivom trenutku. Promena konteksta može se tako izvršiti u sledećim situacijama:
 - Kada tekući proces eksplicitno traži promenu konteksta, tj. "dobrovoljno" se odriče procesora, npr. pozivom funkcije `dispatch()` u školskom jezgru.
 - Kada proces izvrši neki blokirajući sistemski poziv, npr. suspenduje se na nekom sinhronizacionom elementu, npr. semaforu ili se uspava konstruktom `delay`, recimo suspenduje se do trenutka svoje sledeće periodične aktivacije;
 - Kada proces izvrši neki neblokirajući sistemski poziv: iako se pozivajući proces ne suspenduje, već može da nastavi izvršavanje, izvršno okruženje ili operativni sistem dobija kontrolu i može izvršiti promenu konteksta. Npr. operacija *signal* semafora, ili operacija koja je potencijalno blokirajuća (npr. *wait* semafora), nit se ne blokira jer nisu zadovoljeni uslovi za to, ali okruženje ipak implicitno vrši preuzimanje.
 - Kada proces izvrši neku instrukciju koja izaziva hardverski izuzetak koji obrađuje operativni sistem ili izvršno okruženje i kao posledicu toga izvrši promenu konteksta.
 - Kada dođe vreme za aktivaciju nekog periodičnog procesa ili istekne vreme čekanja nekog suspendovanog procesa.
 - Kada se dogodi neki spoljašnji asinhroni događaj koji se manifestuje npr. kao signal zahteva za prekid. Kao posledica tog događaja može se aktivirati neki sporadičan proces, pa se vrši preuzimanje.
 - Kada istekne vreme dodeljeno datom procesu, npr. ako postoji ograničenje WCET ili mehanizam raspodele procesorskog vremena (engl. *time sharing*).

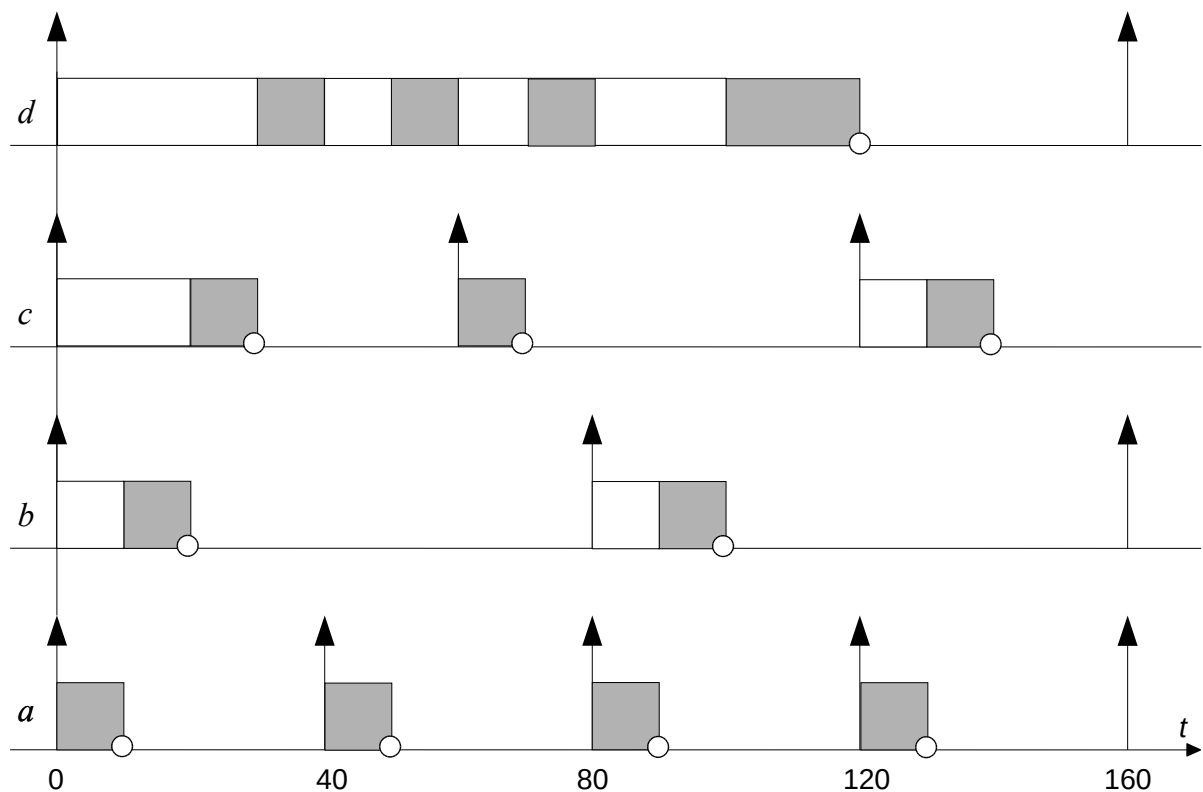
- Prva četiri slučaja su sinhrona preuzimanja, jer se dešavaju kao posledica operacije koju je izvršio sam proces koji je prekinut. Druga tri slučaja su asinhrona preuzimanja, jer se dešavaju potpuno nezavisno od operacije koju tekući proces izvršava.
- Kada sistem vrši ne samo sinhronu, nego i asinhronu promenu konteksta, naziva se sistemom *sa preuzimanjem* (engl. *preemptive*). U suprotnom je sistem *bez preuzimanja* (engl. *non-preemptive*).
- Svaki put kada se vrši promena konteksta, sprovodi se algoritam raspoređivanja. On treba da odluči koji će od trenutno aktivnih, spremnih procesa dobiti procesor. Pretpostavljajući nezavisnost procesa u jednostavnom modelu (bez interakcije procesa), proces se može naći u jednom u sledećih stanja: izvršava se (engl. *running*), spreman je za izvršavanje (engl. *ready*), suspendovan je jer čeka na vremenski događaj (za periodične procese), ili je suspendovan jer čeka na ne-vremenski događaj (za sporadične procese). Svi procesi koji se mogu izvršavati (onaj koji se izvršava i oni koji su spremni), odnosno oni koji ulaze u izbor algoritma raspoređivanja se nazivaju *izvršivim* (engl. *runnable*).
- Ovde će biti razmatrani postupci raspoređivanja *bazirani na prioritetima* (engl. *priority-based*) koji se i najčešće primenjuju. To znači da se procesima dodeljuju prioriteti i da svaki put kada se sprovodi, algoritam bira onaj izvršiv proces koji ima najviši prioritet. Tako se u svakom trenutku izvršava proces najvišeg prioriteta, osim ako je on suspendovan.
- U RT sistemima, prioriteti procesa su posledice njihovih vremenskih karakteristika, odnosno vremenskih ograničenja, a ne nekih drugih karakteristika, npr. njihovog značaja za ispravnu funkcionalnost sistema ili njegov integritet.
- U teoriji i praksi RT sistema postoji veliki broj algoritama raspoređivanja. Ovde će biti razmatrana samo dva najznačajnija i najčešće primenjivana:
 - *Raspoređivanje na osnovu fiksnih prioriteta* (engl. *Fixed-Priority Scheduling*, FPS).
 - *Najkraći-rok-prvi* (engl. *Earliest Deadline First*, EDF).
- Oba ova algoritma su zasnovana na prioritetima, jer se uvek za izvršavanje bira onaj proces iz skupa trenutno izvršivih koji ima najviši prioritet. Međutim, razlika između ovih algoritama je upravo u tome na koji način se procesima dodeljuju prioriteti:
 - kod FPS, prioriteti su procesima dodeljeni unapred, fiksno i statički, konstrukcijom sistema, ali se i to radi na osnovu vremenskih karakteristika procesa, kao što će biti prikazano kasnije;
 - kod EDF, prioriteti se dodeljuju dinamički, tokom izvršavanja sistema, tako da je uvek trenutno najprioritetniji onaj proces kom najskorije ističe vremenski rok.
- Oba algoritma mogu biti primenjena i sa i bez preotimanja (engl. *preemptive* i *non-preemptive*):
 - kod *preemptive* varijante, sistem će izvršiti promenu konteksta ako se u nekom trenutku pojavi prioritetniji spreman proces, recimo zato što je stigao trenutak njegove aktivacije; on će preoteti procesor od procesa nižeg prioriteta koji se izvršavao;
 - kod *non-preemptive* varijante, aktivirani proces višeg prioriteta će dobiti procesor tek kada se proces koji se izvršavao završi, odnosno suspenduje.
- *Preemptive* sistemi su zbog toga reaktivniji (imaju brži odziv), pa su time i poželjniji, ali su teži za implementaciju i analizu rasporedivosti.
- Postoje i međuvarijante, kod kojih proces nižeg prioriteta nastavlja svoje izvršavanje, ali ne obavezno do svog završetka, već do isteka nekog ograničenog vremena, kada se dešava preuzimanje. Ovakav pristup naziva se *odloženo preuzimanje* (engl. *deferred preemption*).

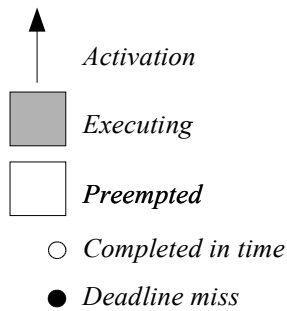
FPS i RMPO

- Kod FPS raspoređivanja, prioritet je svojstvo procesa koje se dodeljuje pri konstrukciji sistema, statički, i u osnovi se ne menja tokom izvršavanja. Algoritam raspoređivanja je krajnje jednostavan: iz skupa izvršivih procesa bira se onaj sa najvišim prioritetom.
- Na primer, neka je dat sledeći skup periodičnih procesa sa $D = T$, dužinom izvršavanja u najgorem slučaju (u oznaci C) i prioritetima (u oznaci P , uz konvenciju da veći broj označava viši prioritet):

Proces	Prioritet, P	Perioda, T	WCET, C
a	4	40	10
b	3	80	10
c	2	60	10
d	1	160	50

- Jasno je da promene konteksta u ovakvom sistemu nastaju onda i samo onda kada se neki proces aktivira ili kada završi svoje izvršavanje, jer se procesi ne suspenduju drugačije, niti se dešavaju neke druge pojave koje bi uzrokovale preuzimanje, a prioriteti procesa su uvek isti. Ako bi se svi procesi izvršavali najduže što mogu (C), raspoređivanje po FPS bi, počev od kritičnog trenutka 0, izgledalo kao na sledećem vremenskom dijagramu:





Oznake na dijagramu znače sledeće: Preempted označava interval u kome je proces izvršiv, ali se ne izvršava, jer se izvršava neki drugi proces višeg prioriteta koji mu je preteo procesor. Ovo je prirodna situacija kod raspoređivanja po prioritetima, jer se i zahteva da se uvek izvršava proces najvišeg prioriteta, dok proces nižeg prioriteta tada mora da čeka iako je izvršiv.

- Treba primetiti da kod ovog raspoređivanja uvek važi to da neki proces može trpeti tzv. *inteferenciju* (engl. *interference*), odnosno biti „nadjačan“ (engl. *preempted*) samo od strane procesa višeg prioriteta, u smislu da mu samo oni mogu preotimati procesor i odlagati njegovo izvršavanje. Drugim rečima, proces „ne trpi“ nikakvo odlaganje izvršavanja zbog procesa prioriteta nižih od njegovog, oni su za njega „nevidljivi“. Shodno tome, proces najvišeg prioriteta ne trpi nikakvu interferenciju, odnosno izvršava se uvek kao da je jedini. Zbog ovoga je vremenske dijagrame poput pokazanog na slici najlakše crtati unošenjem procesa po redosledu prioriteta, od najvišeg ka najnižem.
- Ostaje pitanje kako procesima treba dodeliti prioritete. Naravno, raspodela prioriteta može da utiče na rasporedivost datog skupa procesa: dati skup procesa može biti rasporediv po jednoj raspodeli prioriteta, a nerasporediv po drugoj (u smislu da se mogu dogoditi prekoračenja vremenskih rokova u nekim situacijama).
- Postoji jedan veoma jednostavan, ali *optimalan* način za dodelu prioriteta periodičnim procesima po FPS šemi, tzv. *dodela prioriteta monotono po učestanostima* (engl. *Rate-Monotonic Priority Ordering*, RMPO): periodičnim procesima treba dodeliti jedinstvene prioritete monotono prema učestanosti, tako da proces sa kraćom periodom ima viši prioritet. Drugim rečima, za svaka dva procesa i i j treba da važi: $T_i < T_j \Rightarrow P_i > P_j$.
- Na primer, za gore dat skup procesa, raspored prioriteta po RMPO šemi bi bio sledeći:

Proces	Prioritet, P	Perioda, T	WCET, C
a	4	40	10
b	2	80	10
c	3	60	10
d	1	160	50

- Razlog za ovakav raspored daje sledeća teorema:

Teorema: (O optimalnosti RMPO, Liu & Layland, 1973) Ako je dati skup nezavisnih periodičnih procesa sa $D = T$ rasporediv pomoću FPS *preemptive* raspoređivanja i uz neku (bilo koju) raspodelu prioriteta tim procesima, onda je taj skup procesa sigurno rasporediv i uz RMPO raspodelu prioriteta.

Dokaz ove teoreme biće dat kasnije, kao specijalni slučaj jedne opštije teoreme.

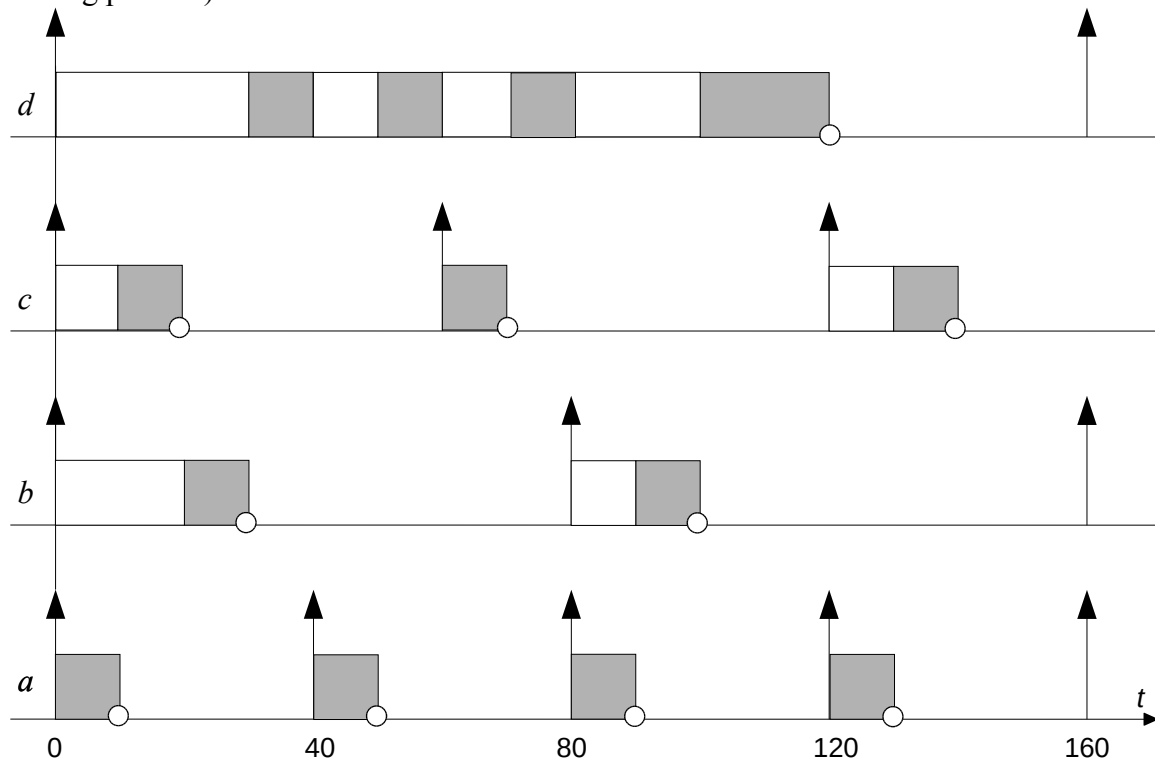
- Dakle, ako je neki skup nezavisnih procesa sa $D = T$ rasporediv po FPS kada im se prioritete dodele proizvoljno, na bilo koji od $n!$ mogućih načina (permutacija, gde je n broj procesa), sigurno je da će taj isti skup procesa biti rasporediv i kada im se prioritete dodele baš po RMPO, odnosno po učestanosti aktivacije. Obrnuto, ako dati skup procesa nije

rasporediv po FPS ako im se prioritete dodele po RMPO, onda taj skup procesa *sigurno nije rasporediv* po FPS ni po *bilo kojoj drugoj* raspodeli prioriteta.

- Zbog toga je potrebno i dovoljno procesima dodeliti prioritete po RMPO i ispitati rasporedivost samo u tom slučaju: ako je skup procesa tada rasporediv, time ujedno znamo i barem neku raspodelu prioriteta po kojoj je taj skup rasporediv. Ako pak dati skup procesa nije rasporediv kada im se prioritete dodele po RMPO, onda *nema potrebe* ispitivati bilo koju drugu (od $n!$) raspodelu prioriteta, jer sigurno ne postoji neka po kojoj bi ti procesi bili rasporedivi. To je upravo smisao optimalnosti RMPO raspodele prioriteta.

EDF

- Kod EDF raspoređivanja se prioritete dinamički menjaju tokom izvršavanja i srazmerni su blizini (skorosti) isticanja vremenskog roka: u trenutku preuzimanja, za izvršavanje se bira onaj izvršiv proces koji ima najskorije apsolutno vreme svog vremenskog roka.
- Iako se obično vremenski rok određuje statički i to kao relativno vreme u odnosu na aktivaciju, apsolutno vreme roka se izračunava dinamički, u vreme izvršavanja. Zbog toga ovaj algoritam podrazumeva veći vremenski trošak tokom raspoređivanja nego FPS.
- I kod ovog raspoređivanja nezavisnih periodičnih procesa promene konteksta nastaju onda i samo onda kada se neki proces aktivira ili kada završi svoje izvršavanje, jer se procesi ne suspenduju drugačije. Osim toga, ako je u nekom trenutku dati proces najprioritetniji, jer mu vremenski rok najskorije ističe, on će sa protokom vremena ostati najprioritetniji sve dok se ne aktivira neki proces kome vremenski rok ne ističe pre njegovog.
- Treba primetiti da je, kako je već pomenuto, za sprovođenje EDF potrebno da raspoređivač ima informaciju o vremenskom roku svakog izvršivog procesa, dok za FPS to nije slučaj (ali mora znati prioritet).
- Za isti prethodni skup nezavisnih procesa sa $D = T$, i ako bi se svi procesi izvršavali najduže što mogu (C), raspoređivanje po EDF bi, počev od kritičnog trenutka 0, izgledalo kao na sledećem vremenskom dijagramu (vremenski rok je trenutak naredne aktivacije datog procesa):



Testovi rasporedivosti

Test rasporedivosti za FPS zasnovan na iskorišćenju

- U svom važnom radu pod naslovom „Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment“ (Journal of the ACM, Vol. 20, No. 1, 1973), Liu i Layland su definisali jedan veoma jednostavan kriterijum provere rasporedivosti skupa periodičnih procesa sa $D = T$ pomoću FPS raspoređivanja, zasnovan samo na ispitivanju iskorišćenja procesora od strane procesa.
- Pritom se iskorišćenjem procesora od strane procesa smatra količnik ukupnog vremena izvršavanja tog procesa na procesoru i ukupnog proteklog vremena. Kako se analiza uvek radi za najgori slučaj, za periodičan proces je iskorišćenje procesora u najgorem slučaju (u oznaci U) jednako količniku WCET (C) i periode T tog procesa:

$$U = \frac{C}{T}$$

- Ovaj kriterijum definiše sledeća teorema.

Teorema: (Test rasporedivosti za FPS zasnovan na iskorišćenju, Liu & Layland, 1973) Ako je sledeći uslov ispunjen, onda je dati skup periodičnih procesa (za koje važi $D = T$) rasporediv pomoću FPS raspoređivanja:

$$\sum_{i=1}^N \left(\frac{C_i}{T_i} \right) \leq N(2^{1/N} - 1).$$

- Suma sa leve strane nejednakosti predstavlja sumu iskorišćenja svih procesa, odnosno ukupno iskorišćenje procesora U u najgorem slučaju.
- Izraz sa desne strane nejednakosti predstavlja graničnu vrednost iskorišćenja procesora za N procesa preko koje ovaj uslov više ne važi. Ova granična vrednost monotono opada sa porastom N i teži ka $\ln 2 \approx 69,3\%$ kada N teži beskonačnosti. Zbog toga svaki skup procesa koji ima ukupno iskorišćenje manje od $69,3\%$ *sigurno jeste* rasporediv pomoću *preemptive* RMPO šeme. Vrednosti ove granice za nekoliko malih N su sledeće:

N	Granično U
1	100,0%
2	82,8%
3	78,0%
4	75,7%
5	74,3%
10	71,8%

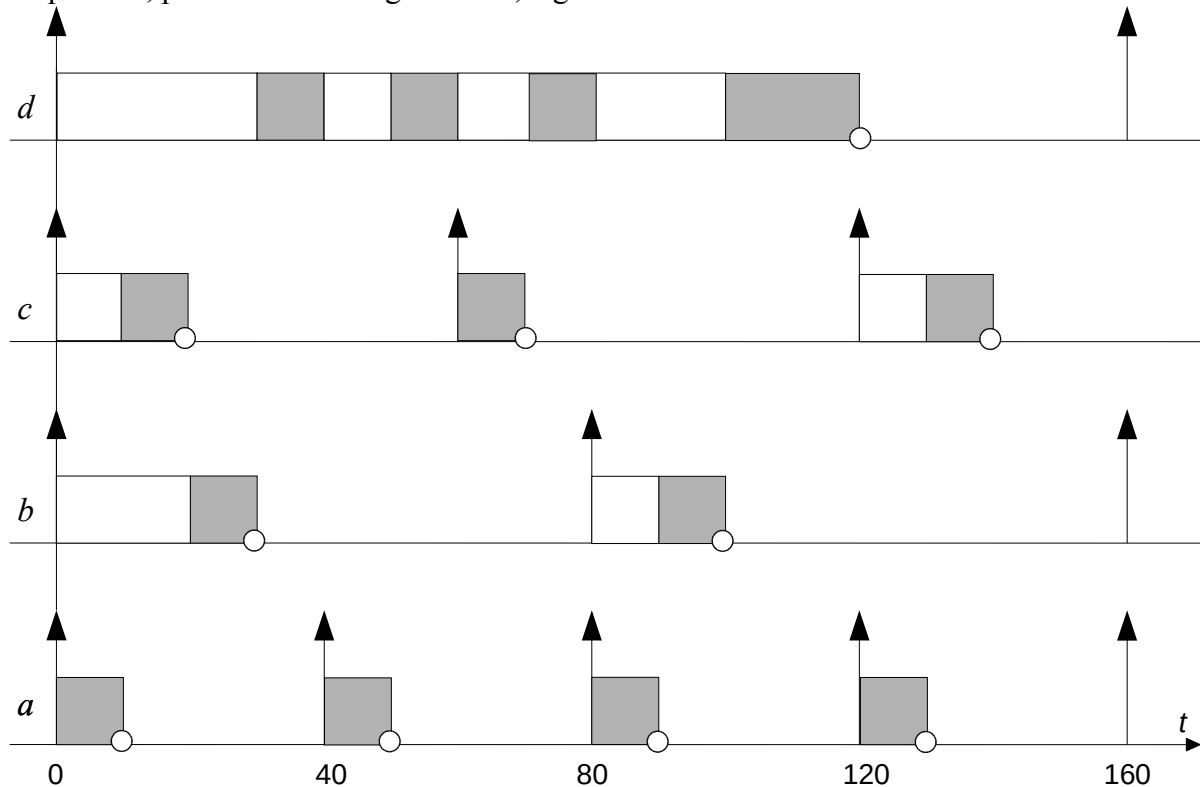
- Ovaj uslov je *dovoljan* za rasporedivost: ako za dati skup procesa važi ova nejednakost, onda je taj skup procesa *sigurno* rasporediv po FPS kada im se prioriteta dodele po RMPO. Međutim, on nije i *neophodan*: ukoliko ovaj uslov nije zadovoljen, skup procesa može biti i nerasporediv, ali može biti i rasporediv. Zbog toga ovaj test nije sasvim tačan, ali je siguran ukoliko je ispunjen.
- Iako ovaj test nije egzakatan, jer predstavlja samo dovoljan, ali ne i potreban uslov rasporedivosti, veoma je popularan zbog svoje jednostavnosti: ispitivanjem ovog kriterijuma, koji je složenosti $O(N)$, može se brzo i lako dobiti rezultat (ukoliko test prolazi). Ukoliko skup procesa ne zadovoljava ovaj test, mora se sprovesti neki drugi, pouzdaniji ali i složeniji test rasporedivosti.
- Na primer, sledeći skup procesa prolazi ovaj test, jer je ukupno iskorišćenje 65% , što je manje od granične vrednosti za tri procesa od 78% , pa je ovaj skup procesa sigurno rasporediv uz RMPO raspodelu prioriteta i FPS raspoređivanje:

Proces	Perioda, T	WCET, C	Iskorišćenje, U
a	10	2	0,2
b	20	5	0,25
c	40	8	0,2

- Primer koji je ranije dat je upravo primer skupa procesa koji ne prolaze ovaj test rasporedivosti, a ipak je rasporediv. Ako se ovim procesima prioriteti dodele po RMPO:

Proces	Prioritet, P	Perioda, T	WCET, C	Iskorišćenje, U
a	4	40	10	0,25
b	2	80	10	0,125
c	3	60	10	0,167
d	1	160	50	0,3125

- Ukupno iskorišćenje je ovde nešto veće od 85,4%, što je veće od granične vrednosti za 4 procesa 75,7%, pa ovaj test ne prolazi. Vremenski dijagram raspoređivanja po FPS ovih procesa, počev od kritičnog trenutka, izgleda ovako:



- Konstrukcija vremenskih dijagrama počev od kritičnog trenutka može da posluži kao test provere rasporedivosti. Međutim, postavlja se pitanje koliko dugo mora da se konstruiše raspored u vremenu da bi bilo garantovano da se u budućnosti neće dogoditi slučaj da neki proces prekorači svoj rok? Za procese koji počinju u istom trenutku (kritični trenutak) može se pokazati da je dovoljno posmatrati vreme do isteka prve najduže periode procesa (Liu & Layland, 1973). To znači da je dovoljno pokazati da svi procesi zadovoljavaju svoj prvi vremenski rok, počev od kritičnog trenutka, pa je sigurno da će zadovoljiti i svaki naredni.

Test rasporedivosti za EDF zasnovan na iskorišćenju

- U istom pomenutom radu, Liu i Layland (1973) su formulisali i test rasporedivosti za EDF zasnovan na iskorišćenju procesora od strane procesa:

Teorema: (Test rasporedivosti za EDF zasnovan na iskorišćenju, Liu & Layland, 1973) Dati skup periodičnih procesa (za koje važi $D = T$) je rasporediv pomoću EDF raspoređivanja ako i samo ako je sledeći uslov ispunjen:

$$\sum_{i=1}^N \left(\frac{C_i}{T_i} \right) \leq 1.$$

- Kao što se vidi, ovaj test je najpre jednostavniji nego za FPS, jer nema izraza zavisnog od N sa desne strane nejednakosti, već je dovoljno jednostavno da ukupno iskorišćenje procesa bude ispod onog teorijski maksimalnog mogućeg (100%). Osim toga, on je sasvim tačan, jer predstavlja *potreban i dovoljan* uslov rasporedivosti (očigledno je potreban, jer je jasno da skup procesa sa ukupnim iskorišćenjem većim od 100% nije rasporediv): ako dati skup procesa prolazi ovaj test, on je rasporediv pomoću EDF raspoređivanja; ako ga ne prolazi, nije.
- Prema tome, EDF raspoređivanje se u ovom smislu čini superiornijim u odnosu na FPS, jer EDF može rasporediti svaki skup procesa koji može rasporediti FPS, dok obrnuto ne važi.
- I pored ovih prednosti EDF šeme, FPS se ipak češće primenjuje iz sledećih razloga:
 - EDF raspoređivač zahteva poznavanje vremenskog roka svakog procesa, dok za FPS to ne važi.
 - FPS je lakše implementirati jer je prioritet (kao atribut procesa) statičan i izračunava se pre izvršavanja, a ne tokom izvršavanja kao kod EDF. EDF ima nešto veće režijske troškove tokom izvršavanja zbog praćenja vremenskog roka.
 - Lakše je inkorporirati procese bez vremenskog roka u FPS šemu nego u EDF, jednostavnim davanjem najnižeg prioriteta takvom procesu, dok davanje proizvoljnog roka za EDF takvom procesu nije sasvim prirodno.
 - Vremenski rok ponekad nije jedini parametar od važnosti. Opet je lakše inkorporirati druge parametre raspoređivanja u pojam prioriteta (za FPS) nego u pojam vremenskog roka (za EDF).
 - Tokom perioda preopterećenja (engl. *overload*) koji su uzrokovani npr. otkazima, odnosno izvršavanjem koda za obradu grešaka i oporavak od otkaza, ponašanje FPS je predvidljivije, jer će uvek prvo trpeti procesi najnižeg prioriteta. Naime, kod FPS raspoređivanja, pošto neki proces trpi interferenciju samo od procesa višeg prioriteta, ako se neki proces izvršava duže nego što je procenjeno, on može ugroziti samo svoj vremenski rok i izvršavanje procesa nižeg prioriteta od njegovog. Dakle, vremenske rokove će možda prekoračivati samo taj proces i procesi nižeg prioriteta od njegovog, dok oni višeg prioriteta *sigurno* neće prekoračivati svoje rokove. EDF je u ovakvim situacijama nepredvidljiv jer jedan proces koji prekorači svoje vreme izvršavanja može ugroziti bilo koji drugi proces, pa tako može da dovede do domino efekta kada procesi počnu da prekoračuju svoje rokove.
 - Neki skup procesa može biti rasporediv i po FPS iako ne prolazi test rasporedivosti zasnovan na iskorišćenju. Zbog toga se i za FPS mogu postizati veća iskorišćenja od onih graničnih za taj test rasporedivosti u nekim slučajevima rasporedivih procesa.

Test rasporedivosti za FPS zasnovan na vremenu odziva

- Test rasporedivosti za FPS zasnovan na iskorišćenju ima dva bitna nedostatka: nije uvek tačan i nije primenjiv na opštije modele procesa. Opštiji pristup je zasnovan na analizi vremena odziva procesa (engl. *response-time analysis*, RTA).
- *Vremen odziva* (engl. *response time*) je vreme koje protekne od trenutka aktivacije nekog procesa (periodičnog ili sporadičnog), do trenutka kada taj proces završi svoje izvršavanje u toj aktivaciji.
- Ovaj test podrazumeva da se za svaki proces izračuna vreme odziva u najgorem slučaju, odnosno u najnepovoljnijem scenariju (u oznaci R), a onda se proveriti da li je to vreme odziva u okviru zadatog vremenskog roka ($R \leq D$).
- Kod FPS raspoređivanja, vreme odziva najprioritetnijeg procesa jednako je njegovom vremenu izvršavanja u najgorem slučaju ($R = C$) jer taj proces, kao što je već pokazano, ne trpi nikakvu interferenciju drugih procesa i izvršava se kao da je jedini.
- Svi ostali procesi će trpeti interferenciju od strane procesa višeg prioriteta od sopstvenog. Interferencija predstavlja vreme provedeno u izvršavanju procesa višeg prioriteta dok je posmatrani proces spreman za izvršavanje (engl. *preempted*). U opštem slučaju, za proces i tako važi:

$$R_i = C_i + I_i,$$

gde je I_i maksimalna interferencija koju trpi proces i u bilo kom vremenskom intervalu $[t, t + R_i)$.

- Posmatrajmo neki proces j koji je višeg prioriteta od procesa i . Tokom nekog intervala $[t, t + R_i)$, proces j će u najgorem slučaju biti aktiviran bar jednom, a možda i više puta. Maksimalan broj aktivacija K_j procesa j u intervalu $[t, t + R_i)$ iznosi:

$$K_j = \left\lceil \frac{R_i}{T_j} \right\rceil$$

Na primer, ako je R_i jednako 15 a T_j jednako 6, onda će proces j biti aktiviran najviše tri puta u intervalu dužine 15 (npr. u trenucima 0, 6 i 12).

- Svaka aktivacija procesa j uzrokuje interferenciju dužine C_j . Zbog toga je maksimalna interferencija koju proces i trpi od procesa j :

$$I_{ij} = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Tako je ukupna interferencija koju trpi proces i :

$$I_i = \sum_{j \in hp(i)} I_{ij} = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

gde je $hp(i)$ skup svih procesa višeg prioriteta od procesa i .

- Konačna jednačina izgleda ovako:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Ova jednačina ima R_i na obe svoje strane, ali ju je teško rešiti zbog funkcije „plafon vrednosti“ (gornjeg zaokruživanja) za R_i/T_j . U opštem slučaju, može postojati mnogo vrednosti R_i koje zadovoljavaju ovu jednačinu. Najmanja takva vrednost predstavlja vreme odziva procesa u najgorem slučaju.
- Najjednostavniji način za rešavanje navedene jednačine jeste formiranje rekurentne relacije:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

gde se za početnu vrednost w_i^0 uzme C_i , a svaka naredna w_i^{n+1} izračuna pomoću prethodne w_i^n datom jednačinom.

- Niz ovako dobijenih vrednosti $\{w_i^0, w_i^1, w_i^2, \dots, w_i^n, \dots\}$ je monotono neopadajući. Kada se postigne da je $w_i^{n+1} = w_i^n$, rešenje jednačine je nađeno. Ako jednačina nema rešenja, onda će niz w nastaviti da raste. To se dešava za proces niskog prioriteta ukoliko je ukupno iskorišćenje veće od 100%. Ako se u nekom trenutku dobije vrednost veća od vremenskog roka D_i , može se smatrati da proces neće ispoštovati svoj rok, pa se dalje izračunavanje može prekinuti.
- Oдавde se može formulisati algoritam za izračunavanje R_i :

```

for each process i do
  W_old := C_i;
  loop
    Calculate W_new from W_old using the given equation;
    if W_new = W_old then
      R_i := W_new;
      exit {Value found};
    end if;
    if W_new > D_i then
      exit {Value not found, deadline missed};
    endif;
    W_old := W_new;
  end loop;
end for;

```

- Iako je redosled kojim se izračunavaju vremena odziva za procese nebitan, najlakše je izračunavati ih po redosledu prioriteta tih procesa.
- Sprovedimo ovaj postupak za isti ranije već analizirani primer skupa procesa i raspodelu njihovih prioriteta po RMPO:

Proces	Prioritet, P	Perioda, T	WCET, C
a	4	40	10
b	2	80	10
c	3	60	10
d	1	160	50

Proces a : $R_a = C_a = 10 < D_a = T_a$, pa ovaj proces zadovoljava svoj vremenski rok.

Za proces c , za interferenciju se uzima u obzir samo proces a koji je jedini višeg prioriteta:

$$w_c^0 = C_c = 10$$

$$w_c^1 = C_c + \left\lceil \frac{w_c^0}{T_a} \right\rceil C_a = 10 + \left\lceil \frac{10}{40} \right\rceil 10 = 20$$

$$w_c^2 = C_c + \left\lceil \frac{w_c^1}{T_a} \right\rceil C_a = 10 + \left\lceil \frac{20}{40} \right\rceil 10 = 20 = w_c^1 = R_c < D_c = T_c = 60$$

Za proces b , za interferenciju se uzimaju u obzir procesi a i c koji su višeg prioriteta:

$$w_b^0 = C_b = 10$$

$$w_b^1 = C_b + \left\lceil \frac{w_b^0}{T_a} \right\rceil C_a + \left\lceil \frac{w_b^0}{T_c} \right\rceil C_c = 10 + \left\lceil \frac{10}{40} \right\rceil 10 + \left\lceil \frac{10}{60} \right\rceil 10 = 30$$

$$w_b^2 = C_b + \left\lceil \frac{w_b^1}{T_a} \right\rceil C_a + \left\lceil \frac{w_b^1}{T_c} \right\rceil C_c = 10 + \left\lceil \frac{30}{40} \right\rceil 10 + \left\lceil \frac{30}{60} \right\rceil 10 = 30 = w_b^1 = R_b < D_b = T_b = 80$$

Za proces d , za interferenciju se uzimaju u obzir procesi a , b i c koji su višeg prioriteta:

$$w_d^0 = C_d = 50$$

$$w_d^1 = C_d + \left\lceil \frac{w_d^0}{T_a} \right\rceil C_a + \left\lceil \frac{w_d^0}{T_c} \right\rceil C_c + \left\lceil \frac{w_d^0}{T_b} \right\rceil C_b = 50 + \left\lceil \frac{50}{40} \right\rceil 10 + \left\lceil \frac{50}{60} \right\rceil 10 + \left\lceil \frac{50}{80} \right\rceil 10 = 90$$

$$w_d^2 = C_d + \left\lceil \frac{w_d^1}{T_a} \right\rceil C_a + \left\lceil \frac{w_d^1}{T_c} \right\rceil C_c + \left\lceil \frac{w_d^1}{T_b} \right\rceil C_b = 50 + \left\lceil \frac{90}{40} \right\rceil 10 + \left\lceil \frac{90}{60} \right\rceil 10 + \left\lceil \frac{90}{80} \right\rceil 10 = 120$$

$$w_d^3 = C_d + \left\lceil \frac{w_d^2}{T_a} \right\rceil C_a + \left\lceil \frac{w_d^2}{T_c} \right\rceil C_c + \left\lceil \frac{w_d^2}{T_b} \right\rceil C_b = 50 + \left\lceil \frac{120}{40} \right\rceil 10 + \left\lceil \frac{120}{60} \right\rceil 10 + \left\lceil \frac{120}{80} \right\rceil 10 = 120 = R_d < D_d = T_d = 160$$

Kako su za sve procese izračunata vremena odziva kraća ili jednaka njihovim vremenskim rokovima, dati skup procesa je prošao ovaj test, pa je rasporediv.

- Treba primetiti da su izračunata vremena odziva upravo ona prikazana na ranije datom vremenskom dijagramu, i to za prvu aktivaciju svakog procesa, koja je i najnepovoljnija, jer je u kritičnom trenutku.
- Ovaj test je superiorniji, ali i složeniji od testa za FPS zasnovanog na iskorišćenju. On je, naime, i potreban i dovoljan: ako skup procesa prođe ovaj test, onda će svi procesi zadovoljiti svoje rokove; ako skup procesa ne prođe ovaj test, neki proces će prekoračiti svoj rok, osim ako procena WCET (C) nije bila previše pesimistička i neprecizna.

Procena WCET

- Svi opisani testovi raporedivosti zasnivaju se na poznavanju procene vremena izvršavanja u najgorem slučaju (WCET) svakog procesa.
- Procena WCET se može izvesti na dva načina, analizom koda i merenjem.
- Analiza koda podrazumeva tehnike kojima se statički procenjuje dužina izvršavanja svakog programskog konstrukta u najgorem slučaju.
- Najpre se kod nekog temporalnog opsega (bloka, potprograma ili procesa) dekomponuje na delove sekvencijalnog koda (bez skokova) i sa samo jednom ulaznom tačkom (kroz koju kontrola toka može ući u taj deo koda) koji se nazivaju *bazičnim blokovima* (engl. *basic block*). Mogućnost prelaska toka kontrole sa kraja jednog bazičnog bloka na početak drugog predstavlja se usmerenom granom. Povezivanjem bazičnih blokova (kao čvorova) granama koje predstavljaju tok kontrole programa formira se tako usmereni graf toka kontrole.
- Zatim se analizira mašinski kod pridružen svakom bazičnom bloku i model procesora, i pokušava da predvidi najduže vreme izvršavanja svakog pojedinačnog bazičnog bloka, sabiranjem dužina izvršavanja svake naredbe u sekvenci tog bazičnog bloka.
- Kada su vremena izvršavanja bazičnih blokova poznata, vrši se redukcija grafa tako što se više čvorova zamenjuje jednim sa najdužim vremenom. Na primer, dva bloka koji čine dve grane *if-then-else* konstrukta zamenjuju se onim koji ima duže vreme izvršavanja.

- Poseban problem čine složenije strukture, recimo petlje sa nepoznatim brojem iteracija (tj. broj iteracija određen vrednošću neke promenljive). U takvim situacijama mora se predvideti najgori mogući slučaj.
- Na primer, posmatrajmo sledeći deo koda:

```
for i in 1..N loop
  if cond then
    -- Basic block of cost 100
  else
    -- Basic block of cost 10
  end if;
end loop;
```

Bez ikakve posebne dodatne informacije, procena WCET za ovaj kod bila bi $N_{\max} \times 100$ plus vreme potrebno za režiju petlje, gde je N_{\max} procenjena najveća vrednost N . Međutim, može se dogoditi da uslov `cond` može biti tačan samo u malom broju iteracija (npr. samo jednoj, npr. prvoj ili poslednjoj). Tako se može dobiti potpuno nerealna, odnosno preterano pesimistična procena od realnosti.

- Zato su moguće (i poželjne) sofisticiranije tehnike semantičke analize koda koje mogu da daju precizniju procenu.
- Nažalost, sasvim preciznu procenu moguće je dati samo u jednostavnim slučajevima, dok je problem sa iole složenijim slučajevima u sledećem:
 - semantička analiza nije moguća ili ne može da da valjan i pouzdan rezultat;
 - model procesora nije precizan ili nije poznat, ili sadrži nepredvidive ili teško procenjive elemente transparentne za softver, kao što su keš memorije, instrukcijski paralelizam, predikcija grananja i mnoge druge.
- U takvim situacijama, procena mora da sprovodi pesimistično zaključivanje, što može dovesti do daleko preceñjenih vremena koje onda mogu da dovedu do negativnih ishoda testova rasporedivosti, iako su procesi zapravo rasporedivi. Tada se poseže za naprednijim hardverom, kako bi se vreme izvršavala skratilo, ali koji može zapravo biti potpuno nepotreban.
- Sa druge strane, nedovoljno restriktivna procena može da dovede do pogrešno sprovedenih analiza rasporedivosti koje pokazuju da je sistem rasporediv, ali se u praksi mogu dogoditi prekoračenja vremenskih rokova. Upravo zato se moraju ugrađivati mehanizmi tolerancije ovakvih otkaza.
- Procena dužine izvršavanja u najgorem slučaju merenjem pretpostavlja izvršavanje datog temporalnog opsega (npr. procesa) više puta i merenje proteklog vremena, podešavajući parametre izvršavanja tako da se program izvrši po najdužoj mogućoj putanji.
- Problem sa merenjem je što se ne može uvek biti siguran da je u probnom izvršavanju postignut baš najgori slučaj. Osim toga, kao i sa analizom koda, problem je što hardver unosi mnoge nepredvidive elemente, pa se ne može biti siguran da je izvršavanje zaista pokazalo najgore moguće vreme.
- Zato se merenja sprovode mnogo puta, uz pažljivu analizu rezultata i pesimističku procenu najgoreg. Međutim, najgore izmereno vreme može biti dobijeno uz grešku u merenju.
- Zbog toga se u praksi uglavnom upotrebljavaju kombinovane tehnike analize (uglavnom pesimistične, koje zanemaruju uticaj optimizacija u hardveru i softveru) i detaljnog testiranja.

Opštiji model procesa

U nastavku će biti pokazano kako se pojedini uslovi mogu relaksirati, odnosno ukinuti, čime se opisani jednostavni model procesa uopštava.

Sporadični procesi

- Sporadični procesi se mogu uključiti u opisani jednostavni model i analizu rasporedivosti tako što se za njih veličina T uzme tako da predstavlja njihovo minimalno vreme između susednih pojavljivanja, odnosno vreme između dva susedna pojavljivanja sporadičnog događaja u najgorem slučaju. Na primer, ako se za neki sporadičan proces zna da se ne može pojaviti više od jednom u svakih 10 ms, za njega se uzima da je $T = 10$ ms u opisanim modelima. Iako sporadičan proces u stvarnosti može da se pojavljuje znatno ređe, analiza vremena odziva će dati rezultat za najgori slučaj.
- Drugi element koji treba definisati za sporadične procese jeste vremenski rok D . U jednostavnom modelu procesa uzima se da je $D = T$. Međutim, ovo za sporadične procese može biti neprihvatljivo, jer oni često predstavljaju obradu otkaza ili nekih vanrednih situacija. Tada se oni aktiviraju relativno retko (pa je T veliko), ali su onda veoma hitni (pa D mora biti relativno malo). Zato model mora da dozvoli postojanje $D < T$, što će biti pokazano kasnije.

Aperiodični procesi

- Jedan jednostavan pristup za FPS raspoređivanje aperiodičnih procesa (koji nemaju definisano minimalno vreme između susednih pojava) jeste taj da se oni izvršavaju sa prioritetom ispod svih *hard* procesa (onih procesa čiji se vremenski rokovi uvek moraju ispoštovati), a iznad *soft* procesa (onih procesa čiji se vremenski rokovi ponekad smeju prekoračiti). Na taj način aperiodični procesi nikako neće ugrožavati *hard* procese, pa će *hard* procesi svakako ispunjavati svoje vremenske rokove, ako su sami za sebe rasporedivi.
- Drugi pristup jeste tehnika *servera* (engl. *server*). POSIX podržava jednu varijantu servera.
- Jedan pristup za realizaciju servera je sledeći. U skup procesa i analizu rasporedivosti uvrsti se i jedan poseban, fiktivan periodičan proces sa najvišim prioritetom, tzv. *server*, koji ima definisan T_s i C_s tako da svi *hard* procesi ostanu rasporedivi kada se *server* izvršava sa periodom T_s i vremenom izvršavanja C_s . C_s predstavlja "kapacitet" dodeljen aperiodičnim procesima, odnosno vreme raspoloživo za njihovo izvršavanje, tako da ono ne ugrozi *hard* procese. U vreme izvršavanja, kapacitet se svakih T_s jedinica vremena postavlja na veličinu C_s . Kada se aktivira aperiodičan proces i pod uslovom da ima preostalog kapaciteta, on se izvršava (sa najvišim prioritetom) sve dok se ne završi ili dok ne potroši kapacitet. U ovom drugom slučaju se suspenduje dok se kapacitet ne dopuni ponovnim postavljanjem na C_s .

Hard i soft procesi

- Analiza rasporedivosti uzima u obzir vreme izvršavanja procesa (pa i sporadičnih) u najgorem slučaju, kao i razmak između susednih pojava sporadičnih događaja u najgorem slučaju. Međutim, veoma često su ta vremena značajno drugačija (nepovoljnija) od onih u prosečnom ili najčešćem slučaju. Da bi dati sistem bio rasporediv u najgorem slučaju, potrebno je da hardver (procesor) bude takav da vremena izvršavanja u najgorem slučaju budu dovoljno kratka da bi analiza rasporedivosti dala pozitivan rezultat. Zbog toga

provera rasporedivosti koja uzima u obzir vrednosti za najgori slučaj može da dovede do znatno slabijeg iskorišćenja procesora u stvarnom izvršavanju sistema, odnosno potrebu za korišćenjem hardvera koji je značajno moćniji (i skuplji) od realno potrebnog.

- Da bi se ovaj problem ublažio, može se primeniti sledeći pristup projektovanju RT sistema:
 - Svi procesi moraju biti rasporedivi uzimajući u obzir *prosečne* vrednosti C i T svih procesa (pod prosečnim T misli se na sporadične procese).
 - Svi *hard* procesi moraju biti rasporedivi uzimajući u obzir vrednosti C i T svih procesa u *najgorem slučaju*.
- Na ovaj način će *hard* procesi uvek, pa i u najgorem slučaju, poštovati svoje vremenske rokove, dok će u većini slučajeva (ali ne uvek) i *soft* procesi poštovati svoje rokove. Situacija u kojoj neki *soft* procesi neće ispoštovati sve vremenske rokove naziva se *tranzijentno preopterećenje* (engl. *transient overload*).

Sistem procesa sa $D < T$

- Test rasporedivosti za FPS zasnovan na iskorišćenju nije primenjiv za sistem sa $D < T$.
- Analiza rasporedivosti za FPS zasnovana na vremenu odziva ni na koji način se ne oslanja na ograničenje da je $D = T$, jer u proračunu vremena odziva ne figuriše vremenski rok procesa. Vremenski rok koristi se samo za proveru prekoračenja od strane izračunatog vremena odziva. Zbog toga je ova analiza potpuno primenjiva i za sistem u kom je $D < T$, za FPS raspoređivanje i neku usvojenu rasporedelu prioriteta.
- Test rasporedivosti baziran na iskorišćenju za EDF raspoređivanje takođe ne važi za sistem sa $D < T$. Međutim, i dalje važi da je EDF efikasniji u smislu da svaki sistem koji je rasporediv po FPS jeste rasporediv i po EDF, dok obratno ne važi. Tako se uslov za rasporedivost po FPS može smatrati dovoljnim (ali ne i potrebnim) i za rasporedivost po EDF.
- Naravno, proračun vremena odziva za FPS, pa time i rasporedivost, zavisi od načina raspodele prioriteta procesima: za neku raspodelu skup procesa može biti rasporediv, dok za neku drugu taj isti skup procesa može prekoračivati vremenske rokove. Postavlja se pitanje kriterijuma raspodele prioriteta procesima za $D < T$ i FPS raspoređivanje. Kao što je ranije pokazano, za sistem procesa sa $D = T$ i FPS raspoređivanje, RMPO je optimalan.
- Za sistem procesa sa $D < T$ postoji sličan princip raspodele prioriteta koji je takođe optimalan. To je tzv. *monotono uređenje prioriteta prema vremenskom roku* (engl. *Deadline-Monotonic Priority Ordering*, DMPO). Kod ove šeme, fiksni, statički prioriteti dodeljuju se procesima prema kratkoći njihovog vremenskog roka, tako da važi: $D_i < D_j \Rightarrow P_i > P_j$. Dakle, procesu sa kraćim vremenskim rokom treba dodeliti viši prioritet.
- Optimalnost DMPO šeme iskazuje sledeća teorema:

Teorema: (O optimalnosti DMPO, Leung & Whitehead, 1982) Ako je dati skup periodičnih procesa rasporediv pomoću *preemptive* FPS raspoređivanja i neke (bilo koje) raspodele prioriteta, onda je on sigurno rasporediv i pomoću DMPO raspodele prioriteta.

Dokaz: Dokaz će pokazati sledeće: ukoliko je neki skup procesa S rasporediv po nekoj raspodeli (permutaciji) prioriteta procesima W , onda se ta raspodela prioriteta W može transformisati tako da se procesima preraspodele prioriteti prema DMPO, a da se rasporedivost očuva.

Ako W nije DMPO, onda postoje dva procesa i i j iz S susedni u permutaciji W tako da važi $P_i > P_j$ i $D_i > D_j$. Definišimo raspodelu W' koja je ista kao i W , osim što su prioriteti procesima i i j međusobno zamenjeni (i i j međusobno zamenjuju mesta u permutaciji). Posmatrajmo rasporedivost po W' :

- Svi procesi sa prioritetima višim od P_i neće biti nikako ugroženi ovom promenom, jer oni ne trpe interferenciju od ostalih procesa koji su nižeg prioriteta od njih.
- Svi procesi sa prioritetima nižim od P_j neće biti nikako ugroženi ovom promenom, jer će i dalje trpeti istu interferenciju od strane procesa viših prioriteta.
- Proces j , koji je bio rasporediv po šemi W , sada ima viši prioritet, pa će trpeti manje interferencije i zato će sigurno biti rasporediv i po šemi W' .

Ostaje da se pokaže da je proces i , kome je smanjen prioritet, još uvek zadovoljava svoj rok. Prema raspodeli W važi: $R_j < D_j$, jer je S raspodediv po W ; $D_j < D_i$, po pretpostavci; $D_i \leq T_i$, što važi za svaki periodičan proces. Prema tome, važi $R_j < T_i$. Zato se proces i može aktivirati najviše samo jednom tokom izvršavanja procesa j po šemi W , pa proces j trpi interferenciju od procesa i samo jednom C_i . Kako proces j ima vreme odziva R_j , koje je manje od njegovog vremenskog roka D_j , pa time i od periode T_j , proces j može biti aktiviran samo jednom u tom vremenu R_j . Prema tome, vreme R_j uključuje samo jedno C_j , jedno C_i kao interferenciju koju trpi od i , kao i interferenciju od strane svih ostalih procesa viših prioriteta od P_i tokom tog R_j . Kada se prioriteti ova dva procesa zamene u W' , proces i će, osim interferencije svih ostalih procesa višeg prioriteta, trpeti samo još dodatnu interferenciju od procesa j . Novo vreme odziva procesa i , R'_i , postaje jednako starom vremenu odziva procesa j , R_j , jer ono uključuje jedno izvršavanje procesa i dužine C_i , samo jedno C_j jer je pokazano da proces j može biti aktiviran samo jednom tokom R_j , i vreme interferencije ostalih procesa višeg prioriteta koje je isto kao i u W , jer je isti interval R_j za koje se ta interferencija posmatra. Odatle sledi:

$$R'_i = R_j \leq D_j < D_i.$$

Odatle sledi da je proces i rasporediv po šemi W' .

Na isti način se šema W' može transformisati u neku novu šemu W'' zamenom prioriteta druga dva procesa čiji susedni prioriteti nisu u skladu sa DMPO, pri čemu će svi procesi i dalje biti rasporedivi itd, sve dok takvi procesi postoje. Kada više ne postoje, raspodela je DMPO. Prema tome, DMPO je optimalna.

- Treba primetiti da isti ovaj dokaz važi i za specijalni slučaj procesa sa $D = T$, pa je ovo ujedno i dokaz optimalnosti RMPO; RMPO je zapravo samo specijalan slučaj DMPO za $D = T$.
- Prema tome, skupu procesa sa $D \leq T$ treba dodeliti prioritete po DMPO i sprovesti analizu rasporedivosti izračunavanjem vremena odziva. Ako svi procesi zadovoljavaju svoje vremenske rokove, dobijena je jedna odgovarajuća raspodela prioriteta. Ako procesi nisu rasporedivi po toj raspodeli prioriteta, sigurno nisu ni po jednoj drugoj.

Interakcija procesa i blokiranje

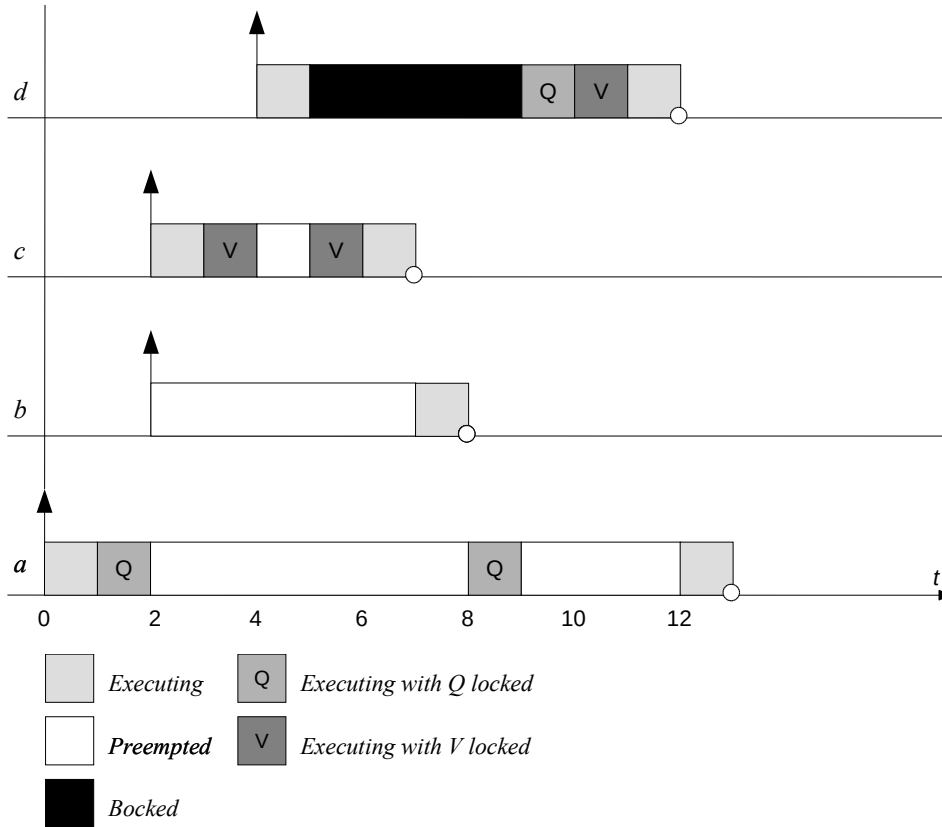
- Pretpostavka da su procesi nezavisni, tj. da ne interaguju, je očigledno preterano restriktivna, jer praktično svaka realna aplikacija zahteva interakciju između nekih procesa.
- Kao što je ranije pokazano, procesi mogu da interaguju ili pomoću neke forme deljenih podataka (koristeći, na primer, semafore, monitore ili zaštićene objekte), ili razmenom poruka (npr. pomoću randevua). Svi ovi koncepti podrazumevaju to da neki proces može da bude suspendovan sve dok se ne ispuni neki uslov (npr. čekanje na semaforu ili uslovnoj promenljivoj, na ulazu u monitor ili zaštićeni objekat, ili čekanje saučesnika za randevu). Zbog toga analiza rasporedivosti mora da uzme u obzir i vreme suspenzije (čekanja na takav događaj) u najgorem slučaju.
- Osim toga, navedeni slučajevi suspenzije mogu da znače i to da neki proces višeg prioriteta mora da čeka, suspendovan na nekom od navedenih kontruktata, dok se uslov ne ispuni od strane nekog drugog procesa nižeg prioriteta, recimo dok taj drugi proces ne oslobodi deljeni resurs (monitor, kritičnu sekciju) ili ne ispuni neki drugi uslov. Za to

vreme čekanja izvršavaju se potencijalno neki drugi procesi, pa i ovaj navedeni drugi proces, koji mogu biti i nižeg prioriteta od onog suspendovanog.

- Na primer, ako je proces p višeg prioriteta suspendovan jer je deljeni resurs pre njega već zauzeo proces q koji je od njega nižeg prioriteta, proces p će morati da čeka da proces q nastavi izvršavanje i oslobodi deljeni resurs. Međutim, proces q može da trpi interferenciju izvršivih procesa višeg prioriteta od njega, pa neće nastaviti izvršavanje dok se oni ne završe. Što se tiče suspendovanog procesa p , izvršavanje procesa prioriteta većeg od njegovog nije problem, jer je njihova interferencija svakako inkorporirana u raspodelu prioriteta, ali su problem svi procesi nižeg prioriteta od p a višeg od q . Proces p mora svakako sačekati da proces q oslobodi resurs, ali njegovo izvršavanje ne mora biti odloženo i zbog izvršavanja procesa koju su po prioritetima izmdeu ova dva procesa.
- U takvoj situaciji pojam prioriteta na neki način gubi smisao, jer nije obezbeđena njegova osnovna svrha, a to je da proces višeg prioriteta neće trpeti zaustavljanje od strane procesa nižeg prioriteta, odnosno da se neće dogoditi situacija u kojoj se izvršava proces nižeg prioriteta, a da je aktiviran proces višeg prioriteta. Prioriteti su, kao što je pokazano, po pravilu dodeljeni po hitnosti vremenskih rokova, pa ova pojava može da utiče na ispunjenje tih rokova.
- Ovakva situacija, kada je proces suspendovan čekajući da neki drugi proces nižeg prioriteta završi svoje izračunavanje ili oslobodi resurs, naziva se *inverzija prioriteta* (engl. *priority inversion*). Za takav proces višeg prioriteta koji čeka na proces nižeg prioriteta kaže se u ovom kontekstu da je *blokiran* (engl. *blocked*).
- U idealnom slučaju, inverzija prioriteta ne bi smela da se desi. U realnim slučajevima nju je nemoguće izbeći, ali je cilj da se ona ograniči i minimizuje, a svakako da bude predvidiva, kako bi mogla da se izvrši analiza rasporedivosti.
- Analiza u narednom izlaganju odnosi se isključivo na FPS raspoređivanje.
- Kao ilustraciju inverzije prioriteta, posmatrajmo primer četiri procesa (a , b , c i d) kojima su prioriteti dodeljeni kao što je prikazano u donjoj tabeli. Pretpostavimo da procesi d i a dele resurs (ili kritičnu sekciju), zaštićen međusobnim isključenjem, označen sa Q , a procesi d i c dele resurs V . U datoj tabeli simbol E označava nezavisno izračunavanje dužine jedne jedinice vremena, a simboli Q i V označavaju izvršavanje u kritičnoj sekciji Q odnosno V , dužine jedne jedinice vremena.

Proces	Prioritet	Sekvenca izvršavanja	Vreme aktivacije
a	1	EQQE	0
b	2	E	2
c	3	EVVE	2
d	4	EQVE	4

- Na sledećem dijagramu pokazan je redosled izvršavanja ovih procesa po FPS:



Kao što se vidi, proces d biva suspendovan u trenutku $t = 5$ kada pokuša da pristupi resursu Q , jer je taj resurs već zauzet od strane procesa a . Ovo se dogodilo zato što je proces a ranije aktiviran od procesa d , jer bi se u suprotnom, kao prioritetniji, izvršavao proces d i on prvi zauzeo ovaj resurs, pa ne bi trpeo blokadu. Zbog toga proces d ostaje suspendovan sve dok proces a ne oslobodi resurs Q . Međutim, proces a je najnižeg prioriteta i zbog toga se ne izvršava sve dok procesi b i c , kao procesi višeg prioriteta od njega, ne završe. Tek kad proces a nastavi izvršavanje i oslobodi resurs Q , proces d nastavlja svoje izvršavanje. Interval u kom je proces d čekao dok su se izvršavali procesi nižeg prioriteta od njegovog je vreme njegove blokade, odnosno pojava inverzije prioriteta. Proces d tako trpi blokadu ne samo od procesa a , koji ga je sprečio da zauzme resurs Q , nego i od procesa c i b . Blokada od strane procesa a je neizbežna, jer a mora da oslobodi zajednički resurs Q . Međutim, blokada od strane drugih procesa se može i treba izbeći.

Prema ovom izvršavanju, d završava u trenutku 12, pa ima vreme odziva jednako 8, iako mu je vreme izvršavanja samo 4. Ovo ukazuje na to da blokada može ugroziti ispunjenje vremenskih rokova, jer bi, da nema inverzije prioriteta, najprioritetniji proces morao da ima vreme odziva jednako svom WCET. Proces c ima vreme odziva jednako 5, b jednako 6, a a jednako 13. Zbog inverzije prioriteta, procesi b i c ne trpe najgoru moguću interferenciju, jer proces d , višeg prioriteta od njih, trpi blokadu i završava se nakon njih.

- Kada je poznato vreme blokade u najgorem slučaju B_i za svaki proces i , onda je njegovo vreme odziva jednako:

$$R_i = C_i + B_i + I_i,$$

odnosno može se dobiti prema formuli:

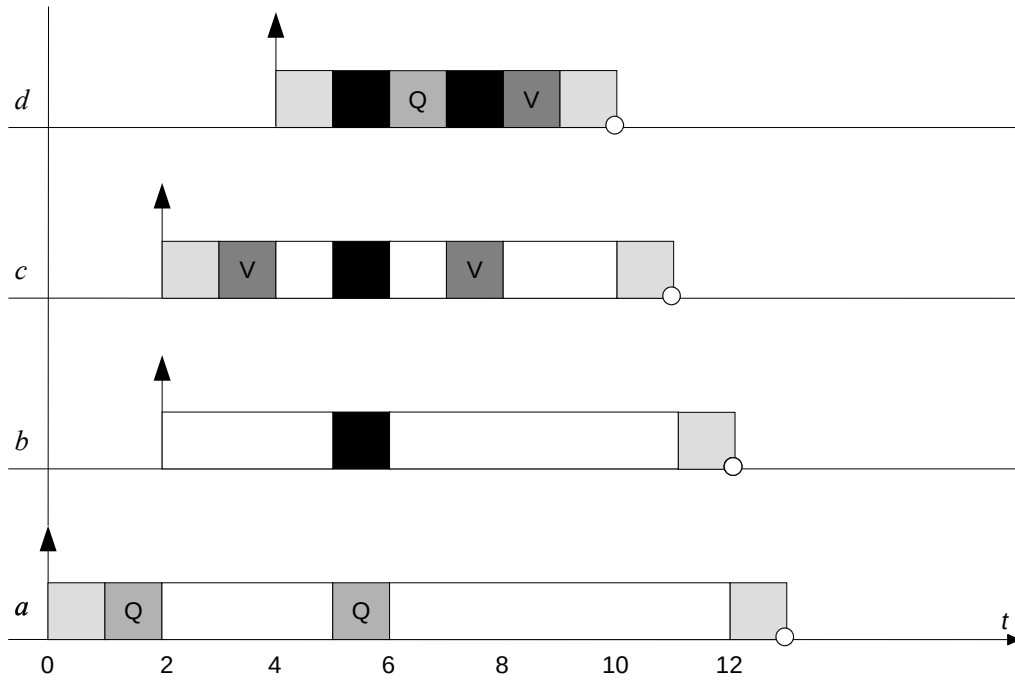
$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_j}{T_j} \right\rceil C_j$$

koja se opet izračunava na isti iterativan način kao i ranije.

- Da li će proces zaista trpeti blokadu zavisi od toga kako se procesi aktiviraju, tj. kako su fazno pomereni trenuci njihovih aktivacija. Inverzija prioriteta, kao što je pokazano, nastaje kada se proces nižeg prioriteta aktivira pre procesa višeg prioriteta, pa stigne da zauzme deljeni resurs pre procesa višeg prioriteta. Ako se to ne događa, recimo zato što su svi procesi iste periode i „u istoj fazi“, odnosno aktiviraju u istom trenutku, inverzije prioriteta neće biti. Zbog toga navedena procena vremena izvršavanja u najgorem slučaju ne mora više biti egzaktna, tj. potrebna, već može biti preterano pesimistična.

Tehnike za rešavanje inverzije prioriteta

- Tehnike kojima se smanjuje uticaj inverzije prioriteta, odnosno vreme blokiranja, zasnivaju se na dinamičkoj promeni prioriteta. Naime, prioriteti procesa nisu više fiksni, već se menjaju u vreme izvršavanja. Svaki proces ima svoj podrazumevani prioritet, dodeljen statički, recimo u skladu sa DMPO šemom. Međutim, u vreme izvršavanja, njegov tekući, dinamički prioritet može povremeno da se promeni, a potom vrati na ovaj statički, koji je podrazumevan. Ovu promenu vrši izvršno okruženje. I dalje se podrazumeva da je algoritam raspoređivanja svakako FPS i on ostaje isti, samo što u obzir uzima dinamički prioritet procesa.
- Jedan tehnika smanjenja inverzije prioriteta jeste *nasleđivanje prioriteta* (engl. *priority inheritance*, Cornhill et al, 1987). Kod ove tehnike prioritet procesa p se menja ukoliko on drži neki resurs na kog čeka neki proces q višeg prioriteta; tada se procesu p dodeljuje taj viši prioritet procesa q koji je zbog njega blokiran: p nasleđuje prioritet od q .
- Ova dodela prioriteta nije ograničena samo na jedan par procesa, već se može raditi i tranzitivno: na primer, ako proces p čeka na proces q , a q opet čeka na proces r , onda će proces r naslediti prioritet procesa p , ako je viši od njegovog tekućeg prioriteta.
- Kada proces oslobodi resurs, vraća mu se njegov podrazumevani prioritet (ili prethodni dinamički prioritet, ako i dalje postoje procesi višeg prioriteta koji su zbog njega blokirani).
- Tako je prioritet procesa u vreme izvršavanja jednak maksimumu njegovog podrazumevanog prioriteta, koji mu je statički dodeljen, i prioriteta svih procesa koji su od njega zavisni u datom trenutku (čekaju ga, tj. blokirani su zbog njega). Zato će u vreme izvršavanja prioriteti procesa biti često menjani, pa o tome treba voditi računa pri implementaciji raspoređivača.
- Prema ovoj tehnici, izvršavanje procesa iz prethodnog primera izgleda ovako:



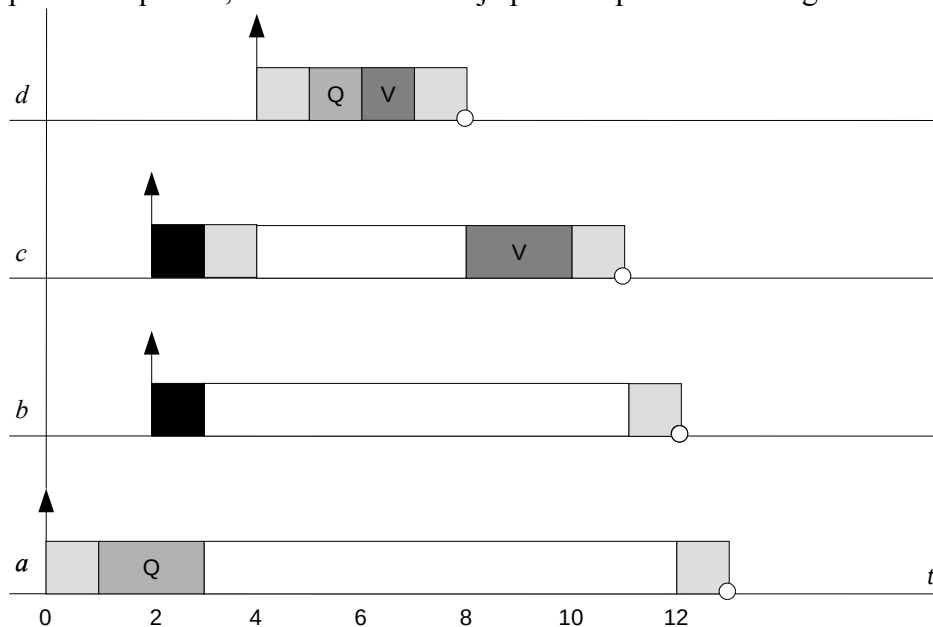
U trenutku kada proces d pokuša da pristupi resursu Q , prioritet procesa a koji je zauzeo Q se povećava na vrednost 4, čime on postaje najprioritetniji izvršiv proces, pa on nastavlja izvršavanje i tako oslobađa resurs Q znatno ranije. Kada a oslobodi resurs, njegov prioritet se vraća na podrazumevani, pa proces d onda nastavlja izvršavanje kao najprioritetniji. Slična situacija dešava se i kada proces d zahteva pristup resursu V koji je zauzet od strane procesa c . Tako proces d sada završava znatno ranije: njegovo vreme odziva je sada 6 umesto ranijih 8.

- Implementacija ove tehnike zahteva da se, osim međusobnog isključenja i potrebne suspenzije kod pristupa deljenom resursu, prati i to koji proces je trenutno zauzeo taj resurs i obezbedi dinamička promena prioriteta tom procesu ukoliko se u redu čekanja pojavi proces višeg prioriteta, kao i naknadni povraćaj prethodnog tekućeg prioriteta procesu koji je oslobodio resurs.
- Ovo nije uvek moguće izvesti jer koncepti za sinhronizaciju i komunikaciju u konkurentnom jeziku nisu uvek pogodni za to. Na primer, izvršavanje operacije *wait* na semaforu, koja blokira proces, ne mora uvek da garantuje da će semafor osloboditi baš onaj proces koji je poslednji prošao kroz semafor: upotreba semafora razlikuje se za potrebe međusobnog isključenja i uslovne sinhronizacije. Zato su pogodniji koncepti koji eksplicitno služe za pristup deljenom resursu.
- Kod šeme sa nasleđivanjem prioriteta postoji gornja granica broja blokada koje neki proces može da trpi. Ukoliko proces prolazi kroz m kritičnih sekcija, onda je maksimalni broj blokada koje mogu da mu se dogode u najgorem slučaju jednak m . Ukoliko postoji samo $n < m$ procesa nižeg prioriteta od njegovog, onda je taj broj jednak n .
- Ukupno maksimalno vreme B_i koje neki proces i može da provede blokirano od strane procesa nižih prioriteta, u slučaju nasleđivanja prioriteta, dato je sledećom formulom:

$$B_i = \sum_{k=1}^K usage(k, i) C(k)$$

gde je K ukupan broj resursa (kritičnih sekcija); $usage(k, i) = 1$ ako resurs k koristi bar jedan proces nižeg prioriteta od procesa i i bar jedan proces višeg ili istog prioriteta od prioriteta procesa i , inače je $usage(k, i) = 0$. $C(k)$ je vreme izvršavanja kritične sekcije za resurs k u najgorem slučaju.

- Sledeći pristup koji još značajnije smanjuje vreme blokiranja predstavljaju *protokoli sa gornjom granicom prioriteta* (engl. *ceiling priority protocols*, Sha et al, 1990).
- Jedan od nekoliko takvih protokola jeste tzv. *protokol sa neposrednom gornjom granicom prioriteta* (engl. *Immediate Ceiling Priority Protocol*, ICPP). On je definisan na sledeći način:
 - svaki proces ima svoj statički dodeljen podrazumevani prioritet (dodeljen npr. DMPO šemom);
 - svaki resurs ima svoju statički dodeljenu *gornju vrednost prioriteta*, tzv. *plafon-vrednost* (engl. *ceiling value*), koja je jednaka maksimumu podrazumevanih prioriteta svih procesa koji koriste taj resurs;
 - u vreme izvršavanja, proces ima svoj dinamički prioritet koji je u svakom trenutku jednak maksimumu vrednosti njegovog statičkog prioriteta i plafon-vrednosti svih resursa koje u tom trenutku on drži zauzete.
- Procesu se tako u vreme izvršavanja prioritet potencijalno menja čim zauzme neki resurs i postavlja se na plafon-vrednost tog resursa, a koja je uvek veća ili jednaka podrazumevanoj vrednosti prioriteta tog procesa, ukoliko je prioritet tog procesa bio niži. Prilikom oslobađanja resursa, prioritet procesa se vraća na prethodnu dinamičku vrednost.
- Za isti prethodni primer, redosled izvršavanja procesa prema ICPP izgleda ovako:



Kako i resurs Q i resurs V koristi proces d podrazumevanog prioriteta 4, plafon-vrednosti ova dva resursa su 4. Proces a započinje svoje izvršavanje sa svojim podrazumevanim prioritetom 1, ali čim zauzme resurs Q , prioritet mu se podiže na 4, što je plafon-vrednost ovog resursa. Zbog toga procesi b , c i d ne mogu da započnu svoje izvršavanje sve dok a ne oslobodi resurs Q , kada mu se prioritet vraća na podrazumevanu vrednost. Dalje procesi nastavljaju izvršavanje prema svojim podrazumevanim prioritetima, jer blokade više nema. Tako je vreme odziva procesa d sada samo 4. Treba primetiti to da se blokiranje procesa dešava eventualno samo na početku njegovog izvršavanja.

- ICPP podržavaju i POSIX (gde se naziva *Priority Protect Protocol*) i RT Java (gde se naziva *Priority Ceiling Emulation*).
- Važno svojstvo ovog protokola u opštem slučaju, ali na jednoprocorskom sistemu, jeste to da će proces trpeti blokadu eventualno samo na početku svog izvršavanja. Kada proces započne svoje izvršavanje, svi resursi koje on traži biće raspoloživi. Zaista, ukoliko bi neki resurs bio zauzet, onda bi on bio zauzet od strane procesa koji trenutno ima viši ili

jednak prioritet od posmatranog procesa (jer prema ICPP taj prioritet sigurno nije manji od plafon-vrednosti tog resursa, a ta vrednost opet nije manja od prioriteta posmatranog procesa), pa bi izvršavanje posmatranog procesa bilo odloženo.

- Posledica ovog svojstva na jednoprocorskim sistemima jeste to da zapravo sam *protokol raspoređivanja obezbeđuje međusobno isključenje* pristupa resursima, pa to nije potrebno obezbeđivati na nivou sinhronizacionih primitiva. Kod ICPP, ako jedan proces zauzima neki resurs, onda će se taj proces izvršavati najmanje sa prioritetom jednakim plafon-vrednosti tog resursa, koja nije manja od prioriteta bilo kog drugog procesa koji koristi isti resurs. Tako nijedan drugi proces koji koristi isti resurs neće dobiti priliku za izvršavanjem od strane samog raspoređivača, pa je implicitno obezbeđeno međusobno isključenje. Zato sinhronizacione primitive za implementaciju međusobnog isključenja ne moraju da implementiraju suspenziju procesa i promenu konteksta, već samo rukovanje priritetom tekućeg procesa prema opisanom protokolu. Naravno, ovo ne važi za višeprocorske sisteme na kojima mora da se obezbedi međusobno isključenje.
- Posledično, takođe na jednoprocorskim sistemima, ovaj protokol *sigurno sprečava mrtvo blokiranje*, jer proces sigurno dobija resurse koje traži posle početka svog izvršavanja, pa nema držanja i čekanja. Zbog toga je primena ovog protokola jedan način za sprečavanje mrtvog blokiranja u RT sistemima.
- Vreme blokiranja procesa u najgorem slučaju predstavlja tako najduže izvršavanje neke kritičine sekcije i dato je sledećim izrazom:

$$B_i = \max_{k=1}^K (usage(k, i) C(k))$$

Zadaci

8.1 Ciklično izvršavanje i FPS

Tri procesa sa $D = T$ imaju sledeće karakteristike:

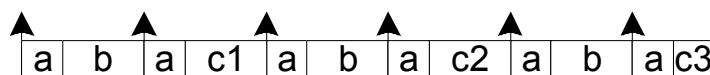
Proces	T	C
a	3	1
b	6	2
c	18	5

- Pokazati kako se može konstruisati ciklično izvršavanje ovih procesa.
- Pokazati kako izgleda raspoređivanje ovih procesa prema FPS, uz prioritete dodeljene prema RMPO, počev od kritičnog trenutka.

Rešenje

- Perioda malog siklusa: 3; perioda velikog ciklusa: 18.

Prema tome, u svakoj maloj periodi potrebno je izvršiti "proces" a u celini. U svakoj drugoj maloj periodi potrebno je izvršiti "proces" b u celini. "Proces" c treba izvršiti po jednom u svakoj velikoj periodi. Međutim, kako tek u svakoj drugoj maloj periodi ostaje još samo 2 jedinice vremena za izvršavanje "procesa" c , potrebno je "proces" c podeliti na tri dela-procedure, nazovimo ih $c1$, $c2$ i $c3$, pri čemu su dva maksimalne dužine 2 jedinice, a jedan dužine 1 jedinice vremena. Tako se dobija sledeći cikličan raspored:



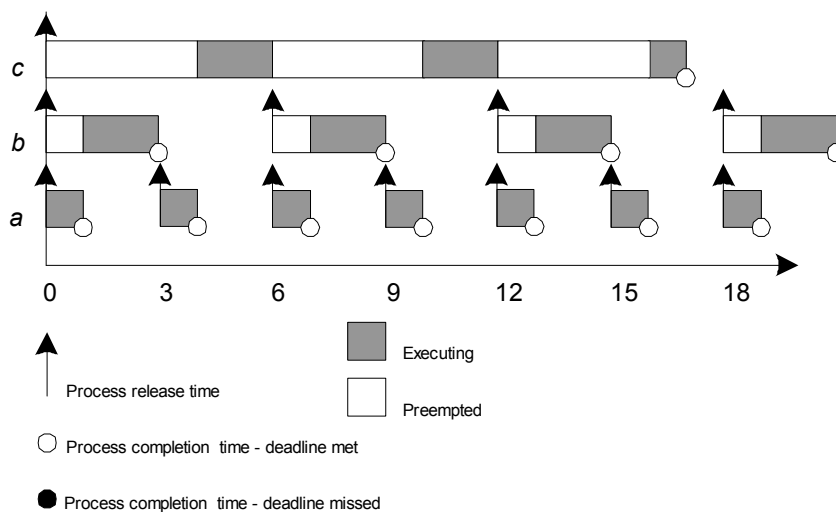
Izvršavanje se dobija sledećim programom:

```

loop
  wait_for_activation;
  procedure_a;
  procedure_b;
  wait_for_activation;
  procedure_a;
  procedure_c1;
  wait_for_activation;
  procedure_a;
  procedure_b;
  wait_for_activation;
  procedure_a;
  procedure_c2;
  wait_for_activation;
  procedure_a;
  procedure_b;
  wait_for_activation;
  procedure_a;
  procedure_c3;
end loop;

```

(b)



8.2 Test za FPS zasnovan na vremenu odziva

Da li je sledeći skup procesa rasporediv FPS raspoređivanjem?

Proces	T	D	C
a	50	40	10
b	40	30	10
c	30	20	9

Rešenje

Prioriteti prema DMPO: $P_c = 3$, $P_b = 2$, $P_a = 1$.

$R_c = C_c = 9 < D_c = 20$, rasporediv;

R_b :

$$R_b := C_b = 10$$

$$R_b := 10 + \lceil 10/30 \rceil * 9 = 19$$

$$R_b := 10 + \lceil 19/30 \rceil * 9 = 19 < D_b = 30, \text{ rasporediv};$$

R_a :

$$R_a := C_a = 10$$

$$R_a := 10 + \lceil 10/30 \rceil * 9 + \lceil 10/40 \rceil * 10 = 29$$

$$R_a := 10 + \lceil 29/30 \rceil * 9 + \lceil 29/40 \rceil * 10 = 29 < D_a = 40, \text{ rasporediv}.$$

8.3

Posmatra se skup asinhronih događaja zajedno sa vremenom izvršavanja potrebnim za reakciju na događaje. Ako se za nadzor svakog događaja koristi zaseban periodičan proces i ti procesi raspoređuju po FPS šemi, pokazati kako se može obezbediti da svi vremenski zahtevi budu ispunjeni.

Događaj	Ograničenje vremena reakcije	Vreme izvršavanja
A	36	2
B	24	1
C	10	1
D	48	4
E	12	1

Rešenje

Za svaki periodični proces pridružen jednom događaju mora da bude zadovoljeno (najgori slučaj):

$$T + D \leq T_c.$$

Izborom parametara T i D može da se utiče na rasporedivost ovih procesa. Kada se ovi parametri izaberu, potrebno je proveriti rasporedivost procesa nekim od pokazanih testova (u svakom slučaju je primenjiv test zasnovan na vremenu odziva). Međutim, kako je najlakše primeniti test zasnovan na iskorišćenju, može se najpre probati sa izborom parametara $T = D$ da bi ovaj test bio primenjiv.

Tako je izvršen izbor parametara gde je:

$$T + D = T_c, \text{ jer se time dobija najrelaksiraniji uslov};$$

$T = D$, jer se tako može primeniti test zasnovan na iskorišćenju. Odatle sledi da je izabrano $T = D = T_c/2$. U narednoj tabeli prikazani su izabrani parametri procesa pridruženih odgovarajućim događajima označenim istim slovima:

Proces	T_c	C	T	D	U
A	36	2	18	18	0,1111
B	24	1	12	12	0,0833
C	10	1	5	5	0,2
D	48	4	24	24	0,1667
E	12	1	6	6	0,1667

Ukupno iskorišćenje iznosi 0,7278, što je manje od granične vrednosti za pet procesa koja iznosi 0,743, pa je ovaj skup procesa sigurno rasporediv po FPS (svakako ako im se prioriteti dodele po RMPO).

8.4

Implementirati ICPP u školskom jezgru. Izvršiti potrebne modifikacije u programskom interfejsu koje su potrebne za zadavanje prioriteta niti, ali tako da kod koji je pravljnjen za

prethodnu verziju jezgra bude prevodiv i na novoj verziji. Kao jedinu primitivu za pristup do deljenih resursa predvideti realizovanu klasu `Mutex` koju treba modifikovati na odgovarajući način.

Rešenje

Izmene u klasi `Thread`:

```
const Priority MinPri = 0;

class Thread : public Object {
public:

    Thread (Priority pri=MinPri) : peForScheduler(this) {
        ...
        peForScheduler.setPriority(pri);
    }

    void start ();
    static void dispatch ();

    static Thread* running ();

    CollectionElement* getPEForScheduler ();
    ...

private:
    ...
    PriorityElement peForScheduler;
};
```

FPS raspoređivač:

```
class Scheduler {
public:

    static Scheduler* Instance ();

    virtual void put (Thread*) = 0;
    virtual Thread* get () = 0;

protected:
    Scheduler () {}
};

class FPScheduler : public Scheduler {
public:

    virtual void put (Thread* t) { if (t) rep.add(t->getPEForScheduler()); }
    virtual Thread* get () { return (Thread*)rep.remove(rep.first()); }

private:
    PriorityQueue rep;
};
```

Konstrukt `Mutex`:

```
class Mutex {
```

```

public:

    Mutex (Priority ceiling);
    ~Mutex ();

private:
    Priority ceiling;
    Priority runningThreadOldPri;
};

Mutex::Mutex (Priority c): ceiling(c) {
    runningThreadOldPri = Thread::runningThread->
        getPEForScheduler().getPriority();
    if (c>runningThreadOldPri)
        Thread::runningThread->getPEForScheduler().setPriority(c);
}

Mutex::~~Mutex () {
    Thread::runningThread->
        getPEForScheduler().setPriority(runningThreadOldPri);
}

```

Način upotrebe:

```

void Monitor::criticalSection () {
    Mutex dummy(ceilingValue);
    //... telo kritične sekcije
}

```

Zadaci za samostalan rad

8.5

Implementirati EDF raspoređivanje u školskom jezgru. Izvršiti potrebne modifikacije u programskom interfejsu (engl. *Application Programming Interface*, API) prema korisničkom kodu koje su potrebne za zadavanje vremenskih rokova niti, ali tako da kod koji je pravljen za prethodnu verziju jezgra bude prevodiv i na novoj verziji.

8.6

Implementirati raspoređivanje sa nasleđivanjem prioriteta (engl. *priority inheritance*) u školskom jezgru. Izvršiti potrebne modifikacije u programskom interfejsu (engl. *Application Programming Interface*, API) prema korisničkom kodu koje su potrebne za zadavanje prioriteta niti, ali tako da kod koji je pravljen za prethodnu verziju jezgra bude prevodiv i na novoj verziji. Kao jedinu primitivu za pristup do deljenih resursa predvideti realizovanu klasu *Mutex* koju treba modifikovati na odgovarajući način.

8.7

Tri procesa sa $D = T$ imaju sledeće karakteristike:

Proces	T	C
a	100	30
b	5	1
c	25	5

Pretpostavimo da a ima najveći značaj za sistem prema svojim funkcionalnostima, pa da zatim sledi b i potom c .

- (a) Pokazati kako izgleda raspoređivanje ovih procesa po FPS počev od kritičnog trenutka, pod uslovom da su prioriteti procesima dodeljeni prema njihovom značaju.
 (b) Kako treba dodeliti prioritete procesima da bi svi vremenski rokovi bili ispoštovani?

8.8

U sistem procesa iz prethodnog zadatka dodat je još jedan proces d čiji otkaz neće ugroziti sigurnost sistema i koji ima period 50 i vreme izvršavanja koje varira od 5 do 25 jedinica vremena. Diskutovati kako ovaj proces treba uključiti u prethodni sistem procesa.

8.9

Procesima sa datim karakteristikama dodeliti prioritete po RMPO i po DMPO, izračunati vreme odziva i ispitati rasporedivost u oba slučaja.

<i>Proces</i>	<i>T</i>	<i>D</i>	<i>C</i>
<i>a</i>	20	5	3
<i>b</i>	15	7	3
<i>c</i>	10	10	4
<i>d</i>	20	20	3

8.10

Prikazati vremenski dijagram FPS raspoređivanja sledećih procesa koji koriste deljene resursa Q i V na dati način:

- a) bez primene neke tehnike rešavanja inverzije prioriteta;
 b) sa primenom tehnike nasleđivanja prioriteta;
 c) primenom protokola ICPP.

<i>Proces</i>	<i>Prioritet</i>	<i>Sekvenca izvršavanja</i>	<i>Vreme aktivacije</i>
<i>a</i>	1	EQQQQE	0
<i>b</i>	2	EE	2
<i>c</i>	3	EVVE	2
<i>d</i>	4	EEQVE	4

V Modelovanje RT sistema

Uvod

- Ovo poglavlje daje veoma kratak i koncizan prikaz metode za modelovanje RT sistema ROOM (*Real-Time Object-Oriented Modeling*) [2]

Istorijat

- Krajem 1980-ih: razvoj u okviru Bell-Nothern Research, Otava, Kanada
- Početak 1990-ih: razvojni tim se izdvaja u sopstveno preduzeće, ObjectTime, Otava, Kanada i počinje razvoj metode ROOM i alata ObjectTime Developer
- Objavljena knjiga: Bran Selic, Garth Gullekson, Paul T. Ward, “Real-Time Object-Oriented Modeling,” John Wiley and Sons, 1994
- Oko 1997. ObjectTime i Rational sarađuju na izradi Rational Rose for RT – u UML inkorporiraju koncepte ROOM
- Krajem 1990-ih: Rational kupuje ObjectTime
- Početkom 2000-ih IBM kupuje Rational
- Sredinom 2000-ih: UML2 su inkorporirani svi ključni koncepti ROOM, sa veoma sličnom semantikom i notacijom, ali pod drugim nazivima; ROOM je ipak bolje definisan jezik, sa više precizno definisanih detalja prilagođenih RT domenu

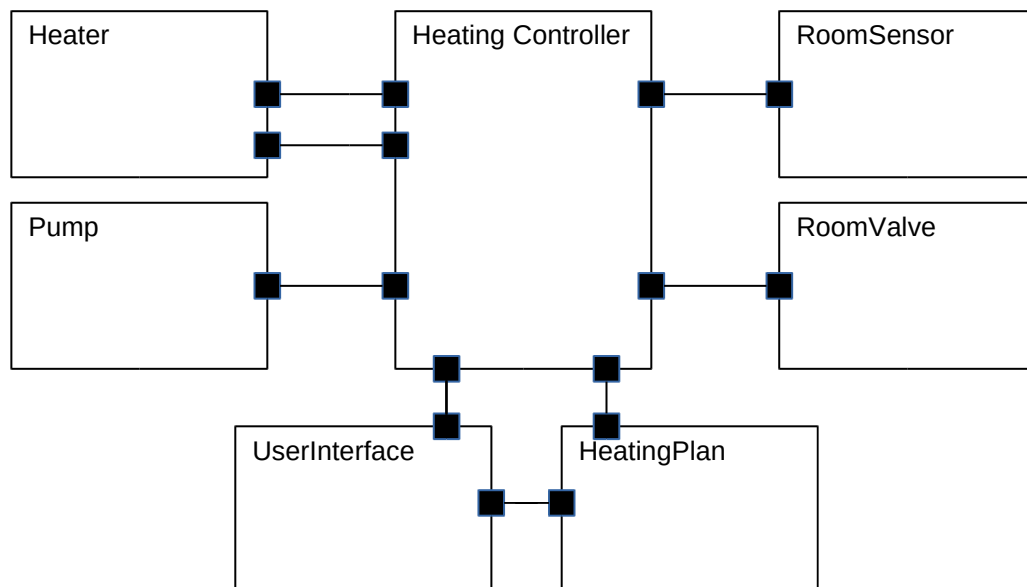
Principi

- ROOM je primenljiv na sisteme sledećih karakteristika:
 - obezbeđuju odziv na vreme (engl. *timeliness*)
 - imaju dinamičku internu strukturu (dinamička rekonfiguracija, kreiranje i uništavanje softverskih komponenta)
 - poseduju reaktivnost na događaje čiji redosled i vreme nisu uvek predvidivi, i to na vreme i u zavisnosti od internog stanja sistema
 - konkurentnost
 - distribuiranost
- Problem ranijih (ali i danas popularnih) načina razvoja softvera – diskontinuiteti [2]:
 - semantički diskontinuitet (engl. *semantic discontinuity*) zbog nedostatka formalne sprege između reprezentacija različitih *vrsta* povezanih detalja, npr. između elemenata koji opisuju strukturu i ponašanje sistema
 - diskontinuitet opsega (engl. *scope discontinuity*) zbog nedostatka formalne sprege između reprezentacija različitih *nivoa* detalja, npr. između elemenata koji opisuju arhitekturu sistema (visok nivo) i implementaciju koraka unutar akcija (nivo detalja)
 - fazni diskontinuitet (engl. *phase discontinuity*) zbog nedostatka formalne sprege između reprezentacija koje se koriste u različitim fazama razvoja, npr. specifikaciji zahteva, projektovanju, implementaciji
- Posledica diskontinuiteta opsega i semantike:
 - modeli i modelovanje se koriste samo u ranim fazama razvoja, specifikacije zahteva i projektovanja, a onda se prelazi na kodovanje, pri čemu kod nije

- formalno i čvrsto spregnut sa modelom, već se dobija ručnom interpretacijom modela ili eventualnom generisanjem parcijalnog kostura i ručnim dopunjavanjem
- “sindrom žurbe ka kodovanju” (engl. *rush to code syndrome*): model ne može da pruži čvrste, očigledne i uverljive dokaze da je sistem dobro zamišljen, kao što to može da pruži kod (ispravnim prevođenjem i izvršavanjem testova), pa je stalno prisutno uverenje da pravi posao nije započet dok nije napisan programski kod [2]
 - model je samo skica rešenja, ne i formalna, autoritarna specifikacija i konstrukcija sistema
 - kada je potrebno uneti ispravku u sistem ili njegovu dopunu, zbog nedostatka čvrste sprege sa implementacijom, iteracija u unapređenju ili ispravci implementacije ne ide od modela, već se to radi samo u kodu, pa model postaje samo neažurna skica sistema (pogrešna dokumentacija)
- Ovaj sindrom se ne može rešiti samo zamenom paradigme (npr. OO umesto proceduralne) – on je posledica navedenih diskontinuiteta
 - Rešenje – operacioni pristup:
 - korišćenje jezika za modelovanje sa *kompletnom, formalnom semantikom*, koja precizno i formalno definiše sprege između različitih nivoa i vrsta detalja
 - zbog formalne semantike, interpretacija modela je jednoznačna, pa je model *izvršiv* (engl. *executable model*) – model se može “prevesti” i “izvršiti” kao i svaki program
 - model je apstraktna specifikacija koja se može hijerarhijski dekomponovati i čiji se detalji mogu specifikovati ili izostaviti do proizvoljne mere
 - Kao posledica, model već u ranoj fazi analize može da posluži za verifikaciju projektnih odluka – nema potrebe za žurbom ka kodovanju; model je tako i brzi prototip sistema, uz moguću simulaciju okruženja
 - Model se može i prevoditi, ali i interpretirati, pa i vizualizovati (uključujući i izvršavanje)
 - Objektna paradigma primenjena u metodi ROOM:
 - Objekti kao instance apstraktnih tipova podataka: kolekcija podataka i pridruženih procedura sa skrivenom implementacijom i dostupnim interfejsom
 - Objekti kao softverske mašine: aktivni agenti implementirani u softveru
 - Objekti kao logičke mašine: aktivne komponente sistema koje mogu biti implementirane u softveru, hardveru, manuelno, ili kombinovano
 - Enkapsulacija: objekat je unutar ljuske u kojoj su „kapije“ kroz koje jedino može ići komunikacija objekta sa okolinom; zaštita unutrašnjosti objekta od okoline, ali i obratno – objekat nema neposredan pristup do svog okruženja, već sa njim interaguje samo preko svog interfejsa
 - Komunikacija između objekata je po modelu razmene poruka (engl. *message passing*), uz podršku i klijent-server relacijama
 - Klase i nasleđivanje (bez višestrukog nasleđivanja)
 - Demo primer koji se koristi u nastavku: automatizovani sistem za kućno grejanje

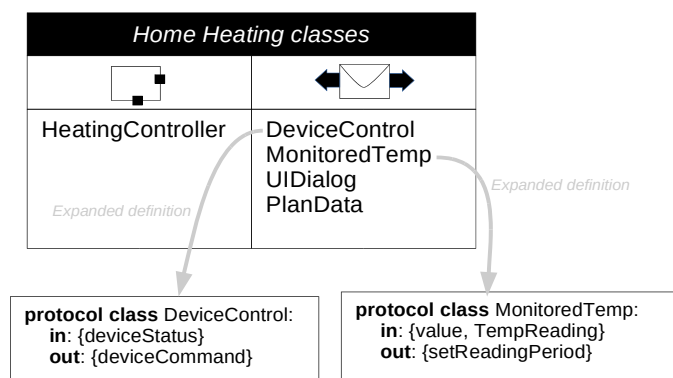
Jednostavan ROOM model

- *Akter* (engl. *actor*) je objekat kao logička mašina, aktivna konkurentno sa drugim akterima (ima svoj tok kontrole), sposobna da prima i šalje poruke kroz svoj *interfejs*
- *Klasa aktera* (engl. *actor class*) – definicija klase objekata (npr. *HeatingController*) koji dele isti *interfejs*, *internu strukturu* i *ponašanje*; osnovna komponenta ROOM modela
- Skica arhitekture sistema na visokom nivou apstrakcije (izostavljeni detalji) definisane preko aktera i njihovih veza preko kojih mogu razmenjivati poruke:

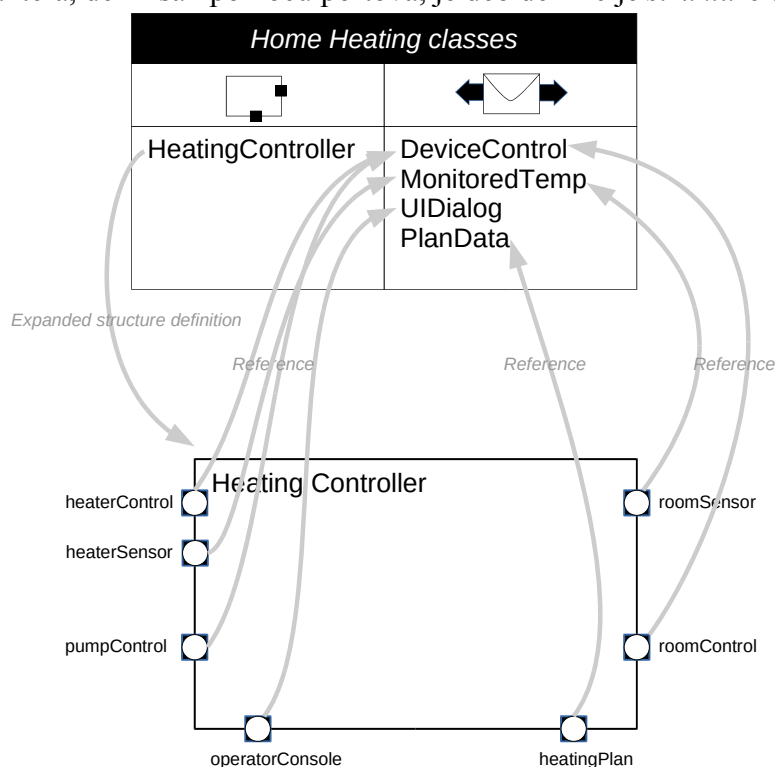


Definicija interfejsa aktera

- Individualne poruke koje se mogu razmenjivati između aktera grupišu se u skupove povezanih; tako identifikovani skupovi poruka koriste se za definisanje tzv. *protokola*
- *Klasa protokola* (engl. *protocol class*) definiše skup vrsta poruka, i za svaku vrstu poruke:
 - *smer poruke*: ulazni (prijem, *in*) ili izlazni (slanje, *out*)
 - *signal* – identifikator vrste poruke
 - *objekat sa podacima* (engl. *data object*) – struktura koja se šalje sa porukom (parametri); svaka poruka može nositi jedan opcioni objekat sa podacima



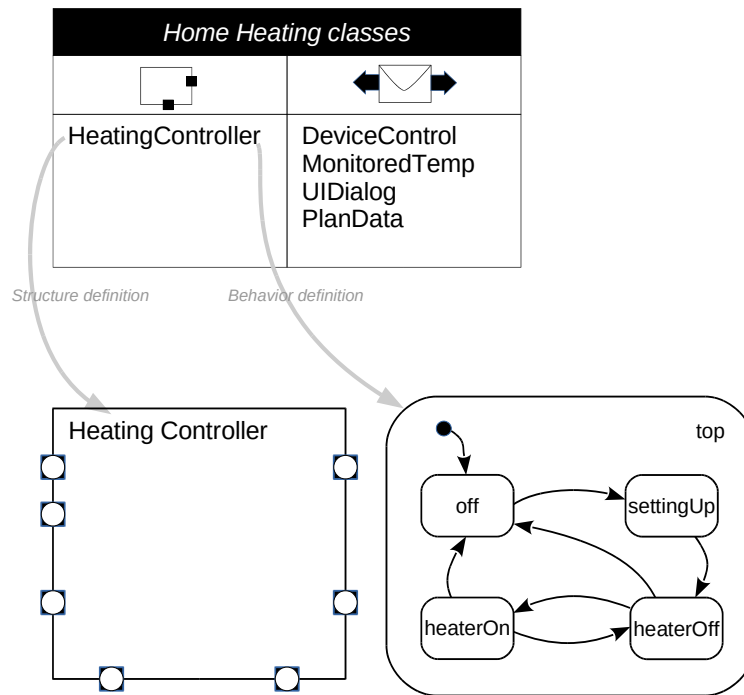
- Interfejs klase aktera sastoji se iz “vrata” (engl. *port*); *port* je referenca na klasu protokola iz klase aktera, deklaracija da skup poruka definisan klasom protokola čini deo interfejsa aktera date klase
- Interfejs aktera, definisan pomoću portova, je deo definicije *strukture* aktera:



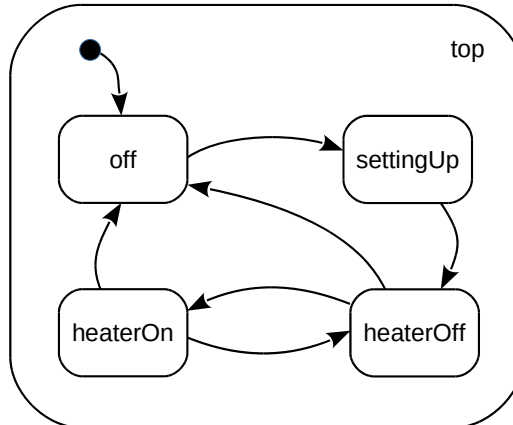
- Interfejs obezbeđuje potpunu i obostranu enkapsulaciju aktera:
 - poruke akteru mogu stizati spolja samo kroz portove, u odgovarajućem smeru ka unutra, u skladu sa definicijom protokola
 - akter može slati poruke svom okruženju samo kroz svoje portove, u odgovarajućem smeru ka spolja, u skladu sa definicijom protokola; akter iz svoje unutrašnjosti ne vidi spoljašnjost, već samo svoj interfejs

Definicija ponašanja aktera

- *Ponašanje* aktera se definiše *mašinom stanja* (engl. *state machine*) koja je deo definicije klase aktera (tzv. *ROOMcharts*):

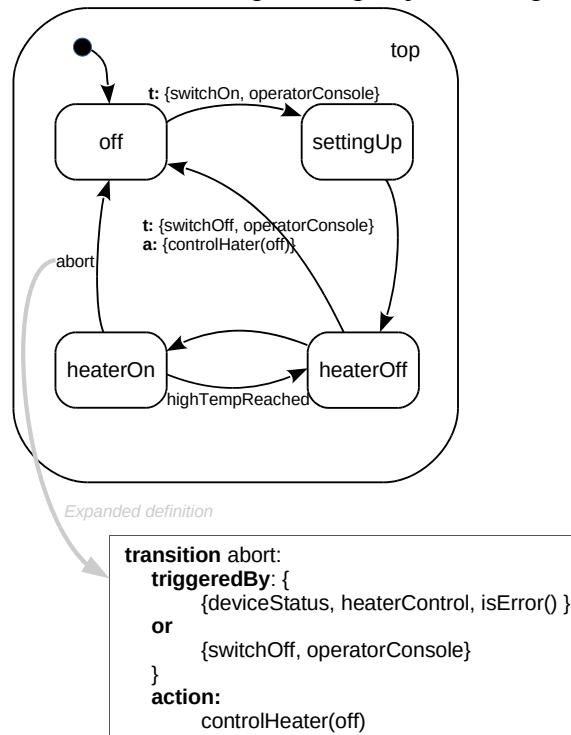


- Mašina stanja modeluje ponašanje u kom reakcija na pobudu zavisi od istorije pobuda, odnosno od trenutnog stanja objekta
- *Stanje* (engl. *state*) predstavlja period vremena tokom kog akter ispoljava određeno ponašanje (izvršava aktivnost i/ili čeka na pobudu na koju će reagovati)
- *Tranzicija* (engl. *transition*) predstavlja moguć prelaz iz jednog u drugo stanje

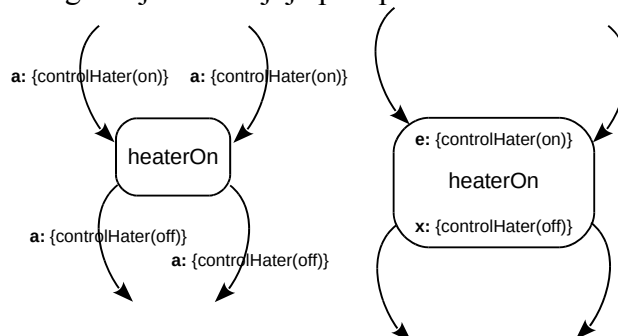


- Tranzicija može (ali ne mora) imati definisan svaki od sledećih elemenata:
 - *okidač* (engl. *trigger*): lista od jednog ili više tripleta – signal (identifikator vrste poruke iz klase protokola), port (sa koga se prima signal) i opcioni uslov (engl. *guard function*) – logički izraz koji mora biti ispunjen da bi tranzicija bila omogućena
 - *akcija* (engl. *action*) koja se preduzima kada se tranzicija okine

- Akcije i logički uslovi su zapisani na *jeziku detalja* (engl. *detail-level language*) koji je inkorporiran u ROOM; može biti neki opšti ili specijalizovan programski jezik (npr. C++)

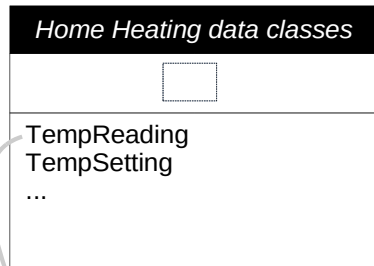


- Tokom izvršavanja, objekat čije je ponašanje definisano datom mašinom stanja nalazi se u jednom od stanja sve dok se ne pojavi ulazna poruka; poruke se obrađuju jedna po jedna. Od skupa tranzicija koje izlaze iz tekućeg stanja, omogućena je ona i samo ona koja ima okidač koji odgovara poruci koja se obrađuje (port i signal) i čiji je uslov ispunjen. Od svih omogućenih tranzicija bira se jedna (bilo koja) koja se okida (ako ih ima više, model nije dobro definisan). Tada se izvršava akcija pridružena toj tranziciji, objekat prelazi u odredišno stanje te tranzicije i obrada poruke se završava. Ako u tekućem stanju nema omogućenih izlaznih tranzicija, poruka se odbacuje (ignoriše) bez efekta.
- Akcija se može pridružiti i stanju i to kao:
 - ulazna akcija stanja (engl. *entry action*) – izvršava se pri svakom ulazu u stanje
 - izlazna akcija stanja (engl. *exit action*) – izvršava se pri svakom izlazu iz stanja
- Ovo predstavlja prečicu za akcije koje treba izvršiti pri svim ulaznim, odnosno izlaznim tranzicijama tog stanja – smanjuje pretrpanost modela:



- Akter može imati svoje enkapsulirane podatke, tzv. *proširene promenljive stanja* (engl. *extended state variables*) – svojstva (atributi) aktera koja imaju svoje vrednosti tipa *klasa podataka* (engl. *data class*)
- Generalno, objekti podataka (engl. *data objects*) su instance apstraktnih tipova podataka (tzv. *klasa podataka*, engl. *data class*)

- Jezik detalja inkorporiran u ROOM obezbeđuje osnovni skup predefinisanih tipova podataka (primitivnih i klasa) sa operacijama, koji se mogu koristiti neposredno, ili od kojih se onda konstruišu korisnički definisani tipovi



Expanded definition

```

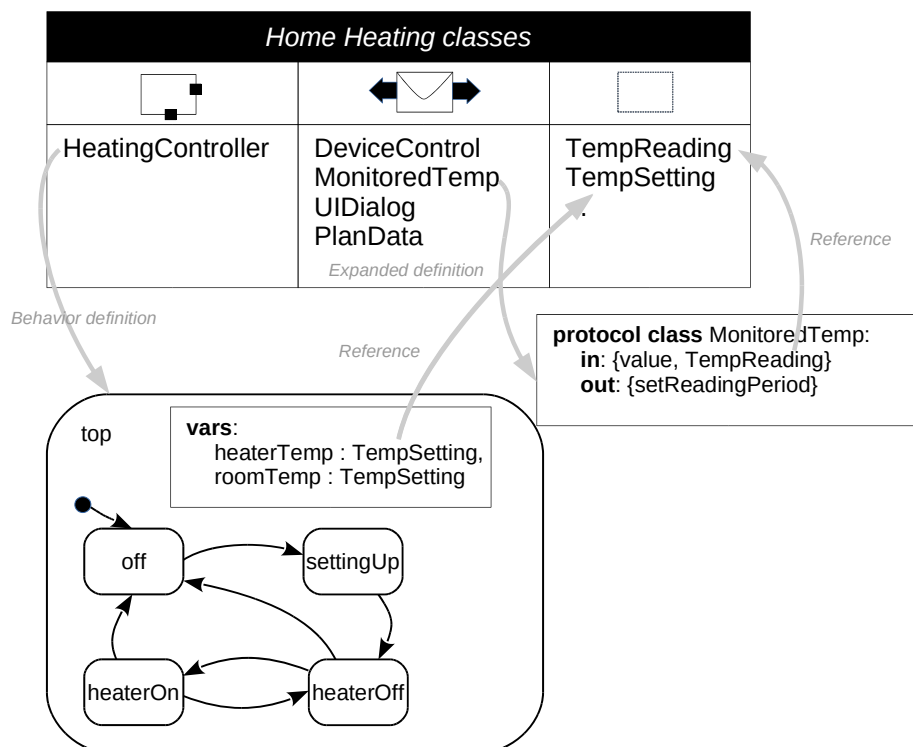
class TempReading {
public:
    TempReading (Temperature temp, DateTime time = DateTime::now());

    Temperature getTemp () const { return value; }
    DateTime     getTime () const { return timestamp; }

private:
    Temperature value;
    DateTime timestamp;
};

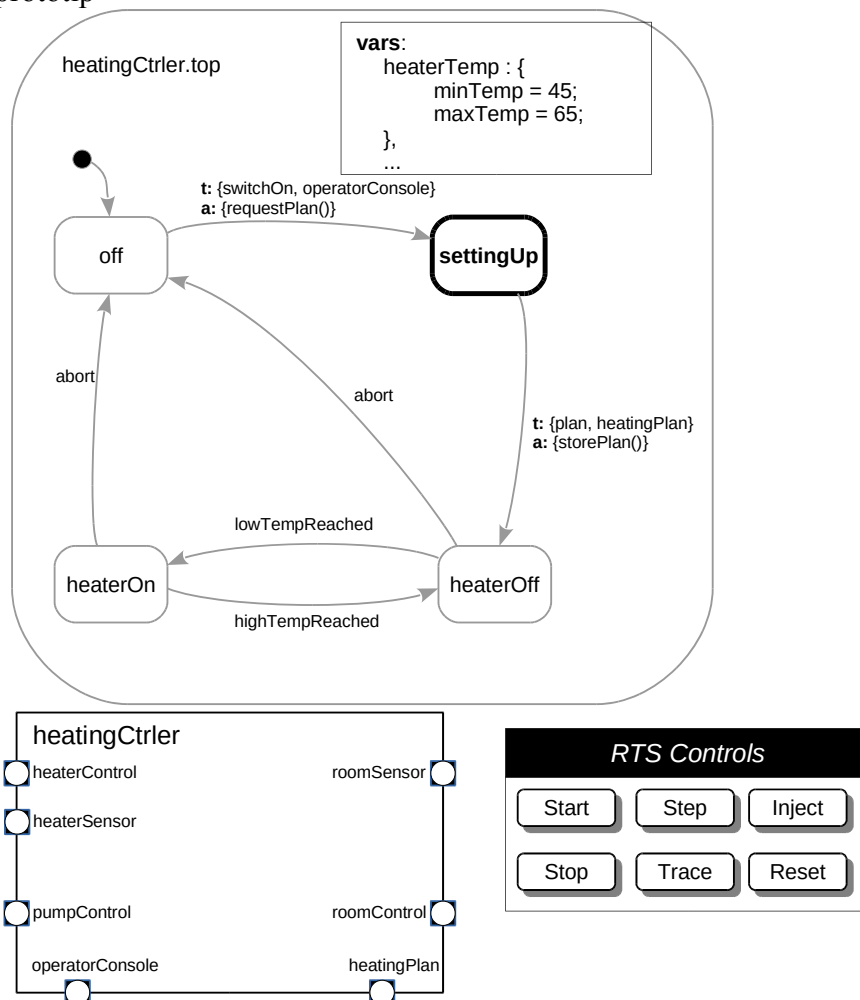
```

- Klase podataka se mogu koristiti za definisanje:
 - proširenih promenljivih stanja aktera
 - podataka koje prenose poruke iz klase protokola: svaka poruka može opciono da nosi jedan (neimenovani) objekat date klase podataka

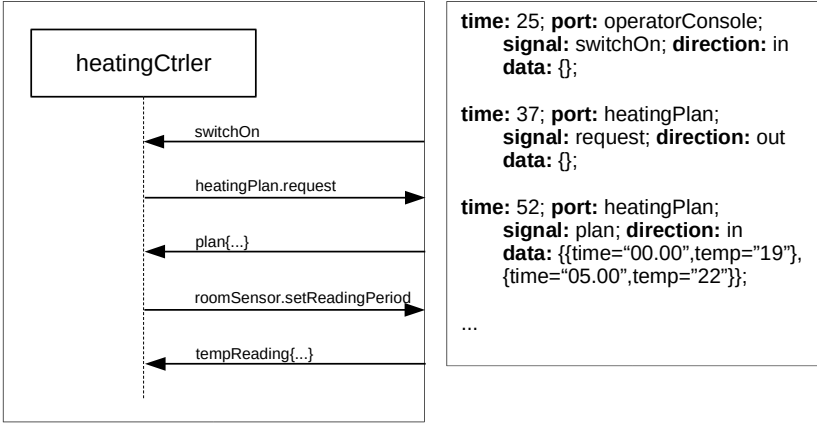


Izvršavanje modela

- Definicija svake klase aktera može se izvršiti, čak i ako nije u potpunosti definisan svaki detalj, pa predstavlja program na jeziku veoma visokog nivoa apstrakcije
- Izvršavanje podrazumeva kreiranje instance klase aktera i njeno izvršavanje u okviru ROOM izvršnog okruženja (virtuelne mašine, VM)
- Izvršavanje omogućava (i zahteva) sve mogućnosti uobičajene za klasične dibagere, ali specifične i za ovu vrstu modela:
 - opservabilnost: praćenje tekućeg stanja, animacija prelaza u mašini stanja, animacija toka poruka, praćenje vrednosti promenljivih, praćenje toka kontrole itd.
 - kontrolu stanja i toka: *start*, *stop*, *inject* (injekcija poruke na neki port aktera), *step*, *trace*, *reset*
- Na ovaj način se model može proveriti veoma rano, i bez definisanja mnogo detalja, kao rani prototip

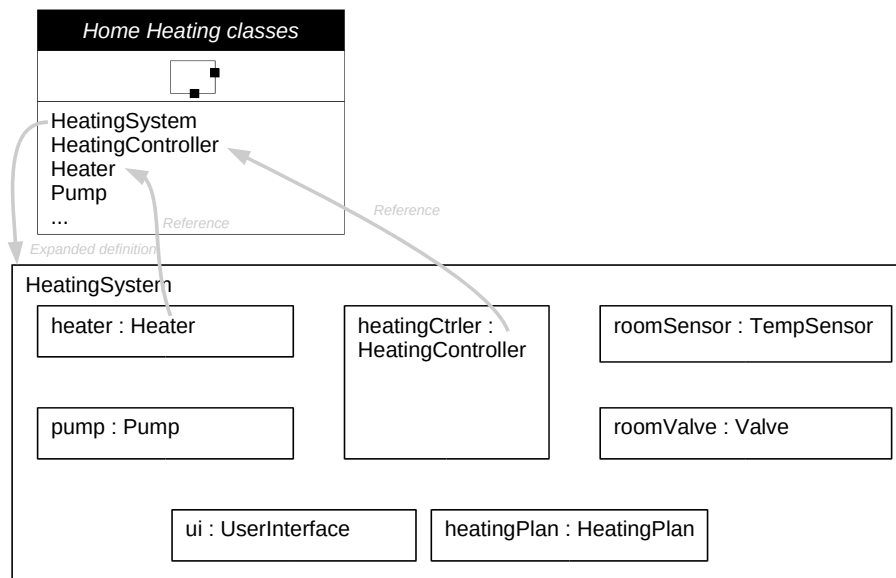


- Moguće je dobiti trag izvršavanja (engl. *trace*), odnosno događaja (razmena poruka, promena stanja):



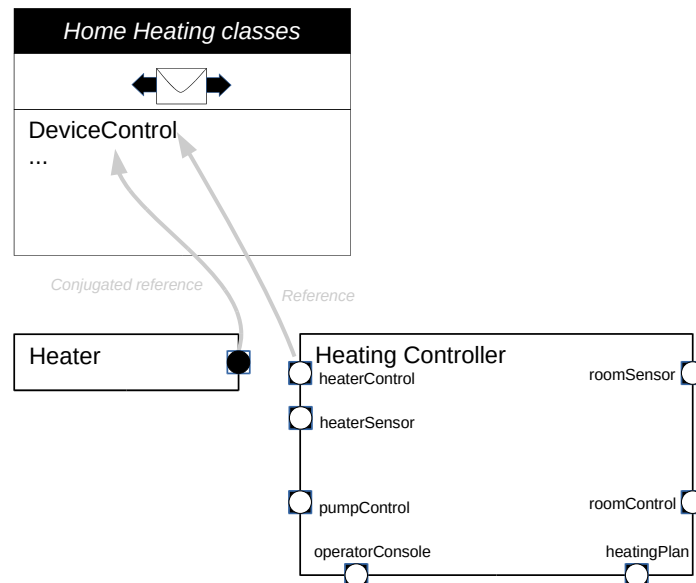
Hijerarhijski model sa više aktera

- Iole složeniji sistem praktično je nemoguće definisati kao jedinstvenu, monolitnu komponentu; ROOM primenjuje jednobrazan sistem hijerarhijske dekompozicije strukture aktera: akter može da sadrži druge aktere kao svoje komponente do proizvoljne dubine
- Akteri mogu da predstavljaju i neračunarske ili kombinovane komponente (npr. Valve, Pump), skupove podataka (npr. HeatingPlan) ili ceo heterogeni sistem (npr. HeatingSystem)
- Definicija strukture aktera sadrži svojstva (delove) kao reference na klase aktera koji definišu tip tog svojstva (npr. svojstvo `roomValve` tipa `Valve`). Podrazumevana semantika je sledeća: kada se kreira jedna instanca definisane klase aktera (`HeatingSystem`), ona će u sebi sadržati po jednu instancu referenciranog tipa za svako svojstvo (deo, podakter)

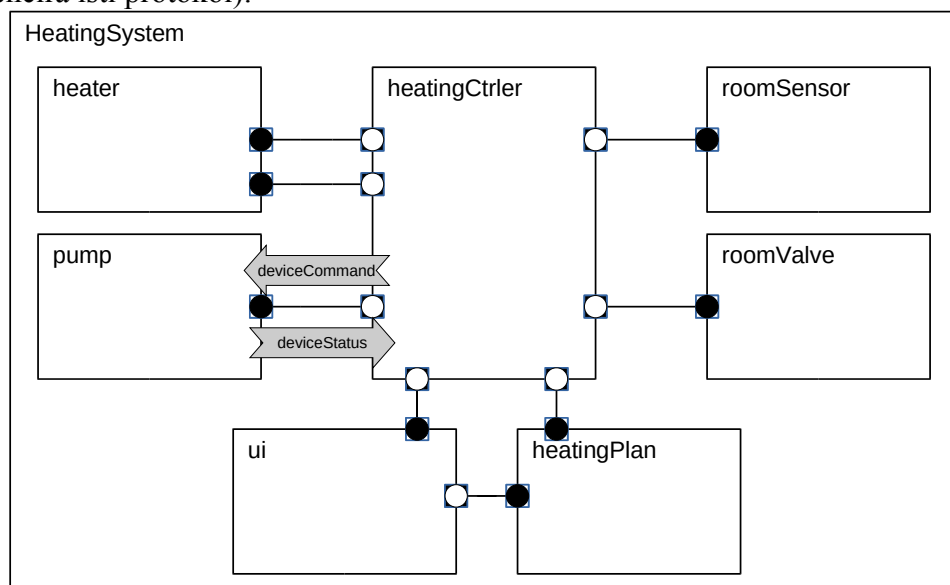


Komunikacija između aktera

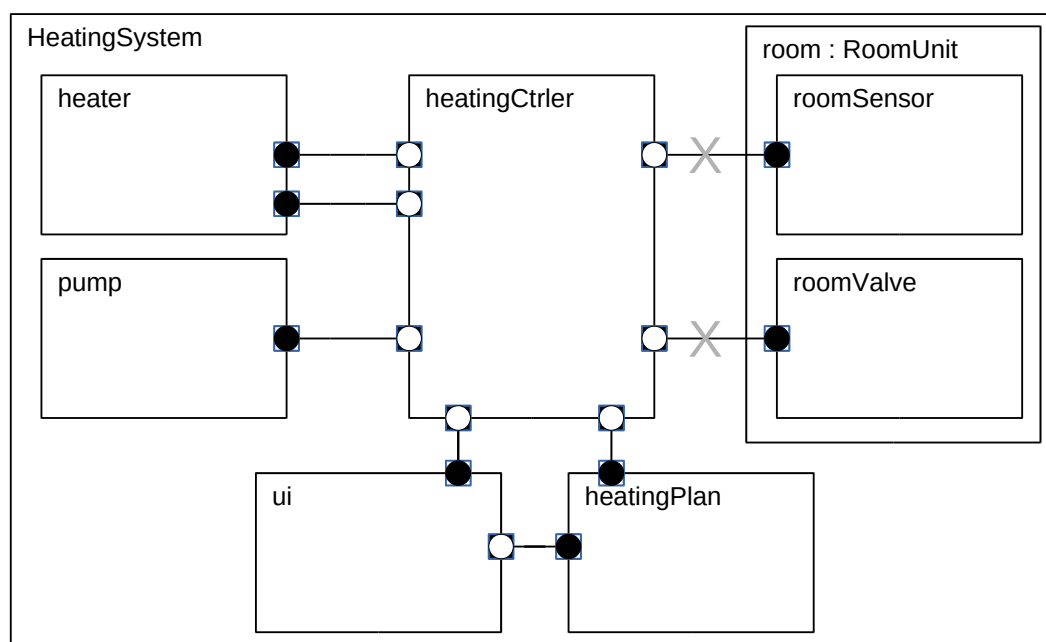
- Komunikacija između aktera-komponentata unutar okružujućeg aktera obavlja se preko portova i veza
- *Konjugovani port* (engl. *conjugated port*) je deo interfejsa klase aktera, referenca na klasu protokola, ali podrazumeva protok poruka u obrnutom smeru od onog definisanom u referenciranom protokolu:



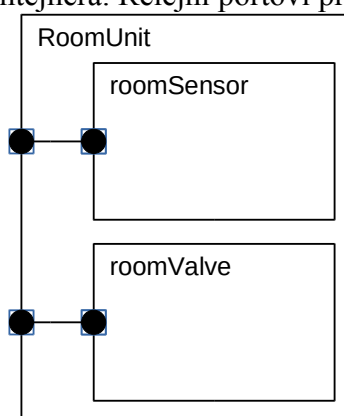
- *Veza* (engl. *binding*) je svojstvo klase aktera unutar koje je definisana i predstavlja komunikacioni kanal za protok poruka između portova koje povezuje
- Veza može biti uspostavljena između kompatibilnih portova aktera koji su sadržani unutar iste klase aktera (npr. port koji referencira jedan protokol i konjugovani port koji referencira isti protokol):



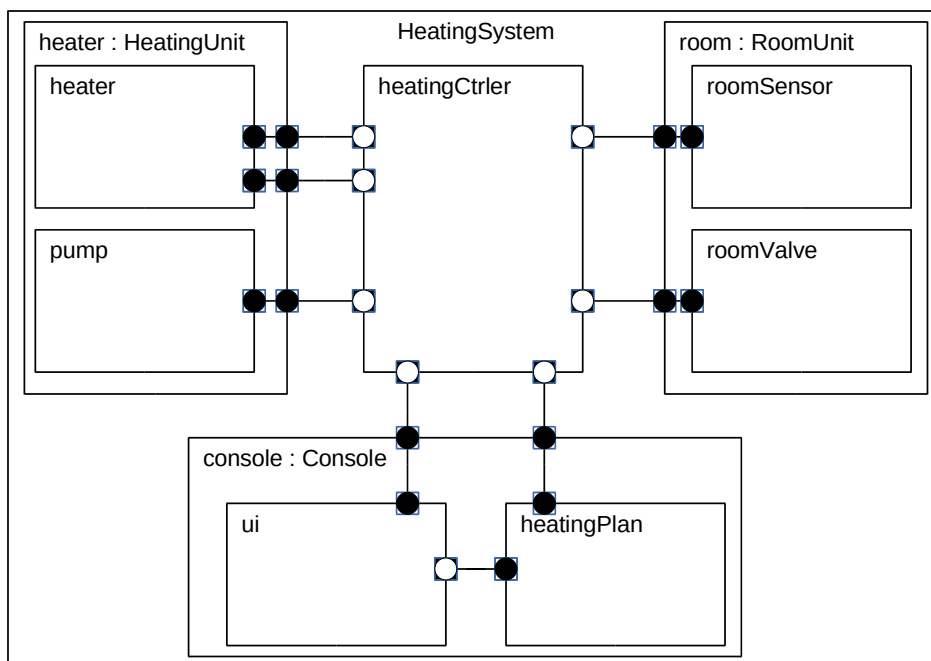
- Klasa aktera predstavlja apstrakciju koja sakriva svoju internu strukturu i ponašanje iza svog interfejsa. Kada akteri-komponente koji čine internu strukturu neke klase aktera treba da razmenjuju poruke kroz interfejs okružujuće klase aktera, potrebno je koristiti relejne portove, jer veza ne može prelaziti granice interfejsa klase aktera:



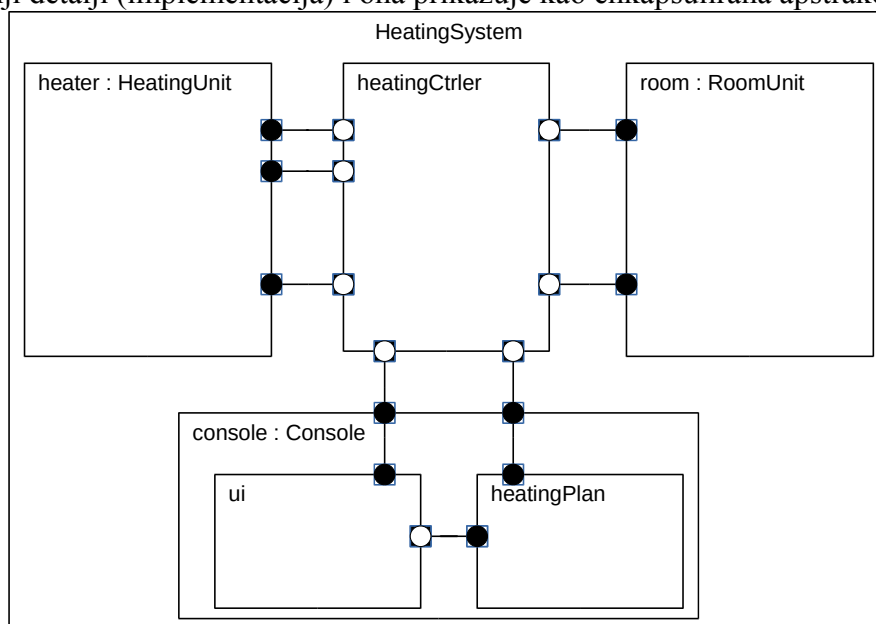
- Relejni portovi (engl. *relay port*) su način da se portovi komponenata aktera regularno “izvezu” u interfejs aktera kontejnera. Relejni portovi prosto prosleđuju sve poruke:



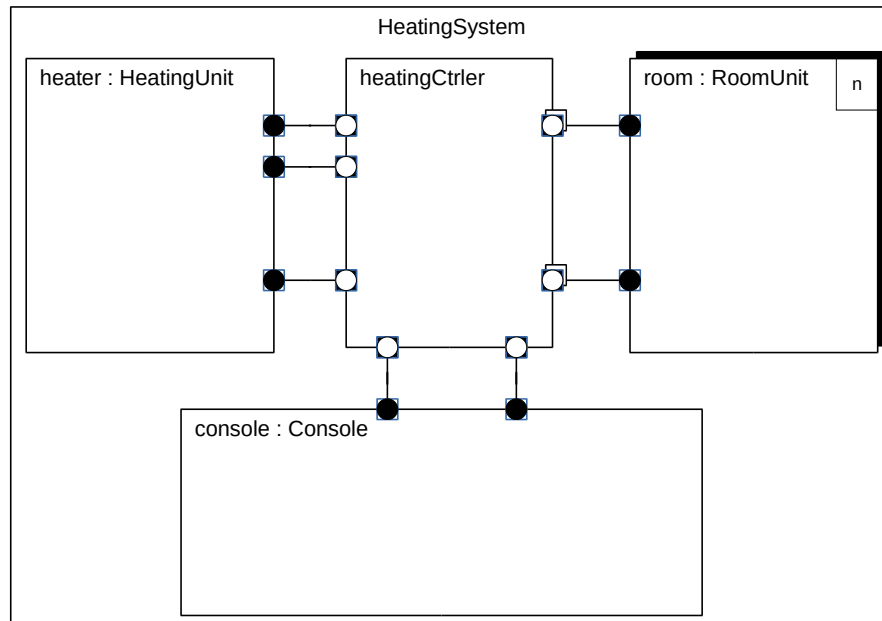
- Relejni portovi, kao i krajnji portovi (engl. *end port*) mogu biti konjugovani ili nekonjugovani; relejni port može biti povezan sa istorodnim interfejsnim portom aktera-komponente
- Poruka može da prolazi kroz proizvoljno mnogo relejnih portova od svog izvora do konačnog odredišta



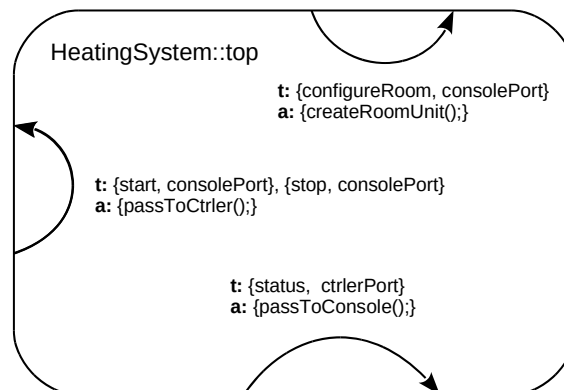
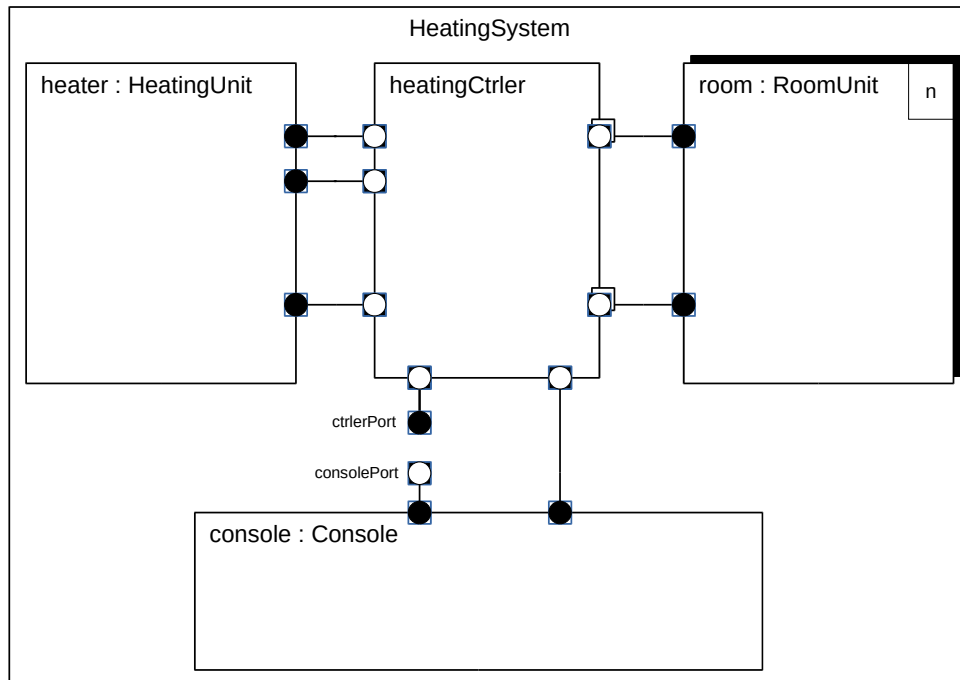
- Unutrašnja struktura aktera može se sakriti na dijagramu, čime se sakrivaju njeni unutrašnji detalji (implementacija) i ona prikazuje kao enkapsulirana apstrakcija:



- Kada je u klasi aktera potrebno imati višestruke instance iste klase aktera kao komponente, definiše se replikacija referenci na aktere i referenci na portove: svojstva kardinalnosti veće od jedan; na primer, imamo više soba u kojima je instalirana ista jedinica za kontrolu grejanja u sobi (`roomUnit` tipa `RoomUnit`)
- Replikacija reference na klasu aktera menja definiciju *kontejnerske* klase aktera, a ne *referencirane* klase aktera
- Replikacija reference na klasu protokola menja definiciju referencirane klase aktera (menja se njen interfejs), a ne kontejnersku klasu aktera:



- Moguće je definisati i opcione reference na aktere: donja granica kardinalnosti je 0; tada se instance komponente aktera kreiraju dinamički, eksplicitno, akcijama ponašanja (a ne implicitno pri kreiranju kontejnerske komponente)
- Moguće su različite definicije kardinalnosti:
 - opcioni i neograničeni: 0..*
 - samo opcioni: 0..1
 - obavezni i neograničeni: 1..*
- Akter može, ali ne mora imati i aktere-komponente i sopstvenu mašinu stanja. Primer kada akter ima i komponente i sopstvenu mašinu: kada je neka komponenta opcionalna, njeno kreiranje se obavlja akcijama u ponašanju (mašini stanja) kontejnera
- Prošireno stanje (engl. *extended state*) aktera u vreme izvršavanja sastoji se tako od:
 - trenutnog stanja mašine stanja (ponašanja) tog aktera
 - trenutne vrednosti varijabli tog aktera
 - proširenih stanja aktera-komponentata koje taj akter sadrži, rekurzivno
- Komunikacija između aktera-kontejnera i aktera-komponente moguća je preko *internih* portova kontejnera (port koji nije deo interfejsa, nego interne implementacije klase aktera); poruke primljene na interni port prosleđuju se mašini stanja aktera čiji je to port, a mašina stanja može slati poruke i na taj port:

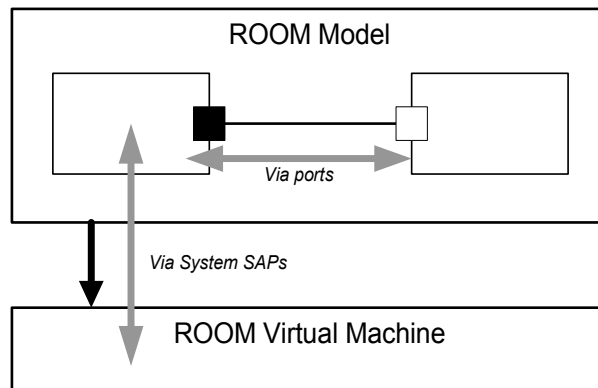


- Na slici je prikazana jednostavna mašina stanja klase aktera `HeatingSystem`, koja se sastoji samo od jednog jedinog stanja (`top`) i internih prelaza (prelazi koji ne napuštaju stanje, mašina ostaje u istom stanju): jednostavna obrada poruka u procedurama, bez promene stanja
- Poruka `configureRoom` primljena na interni port `consolePort` od komponente `console` prosleđuje se mašini stanja klase `HeatingSystem`, koja onda akcijom unutar potprograma `createRoomUnit` (nivo detalja) kreira akter `room` klase `RoomUnit`
- Poruke `start` i `stop` primljene na interni port `consolePort` od komponente `console` prosleđuju se mašini stanja klase `HeatingSystem`, koja ih onda akcijom unutar potprograma `passToCtrlr` (nivo detalja) prosleđuje na interni port `ctrlrPort` akteru-komponenti `heatingCtrlr`

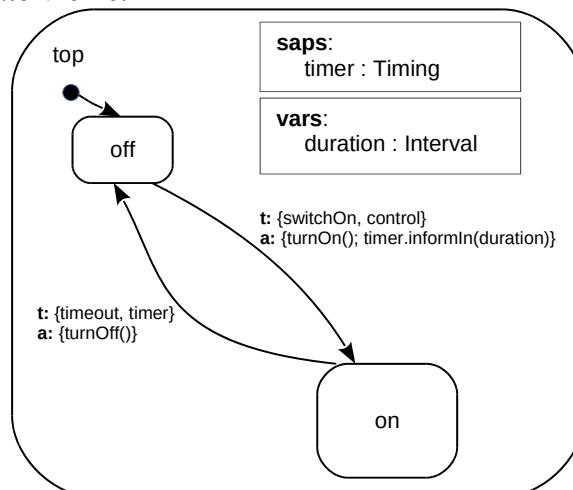
Sistemske servisne pristupne tačke

- Komunikacija između aktera-komponentata preko portova i veza je samo jedna, „horizontalna“ vrsta komunikacije

- Moguća je i „vertikalna“ komunikacija preko tzv. *sistemskih servisnih pristupnih tačaka* (*System Service Access Point, SAP*): pristupne tačke do ROOM virtuelne mašine (VM) i njenih usluga; SAP se ponašaju isto kao interni portovi unutar aktera, ali omogućavaju razmenu „sistemskih“ poruka sa VM

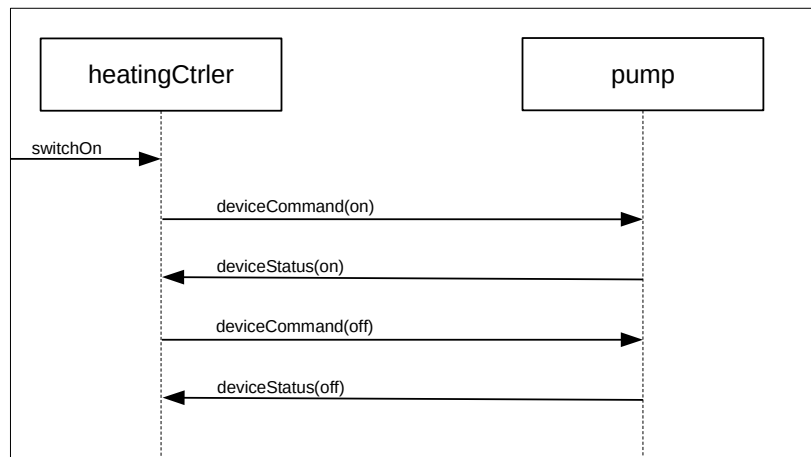


- ROOM VM obezbeđuje predefinisane protokole za svoje interfejse i usluge
- Na primer: protokol *Timing* iz interfejsta *Timing Service* uključuje predefinisanu poruku *timeout* koja se šalje kada istekne zadato vreme
- Primer mašine stanja koja koristi tajmer tako da se neki uređaj (npr. grejač, *Heater*) drži uključen zadato vreme:



Interne sekvence poruka

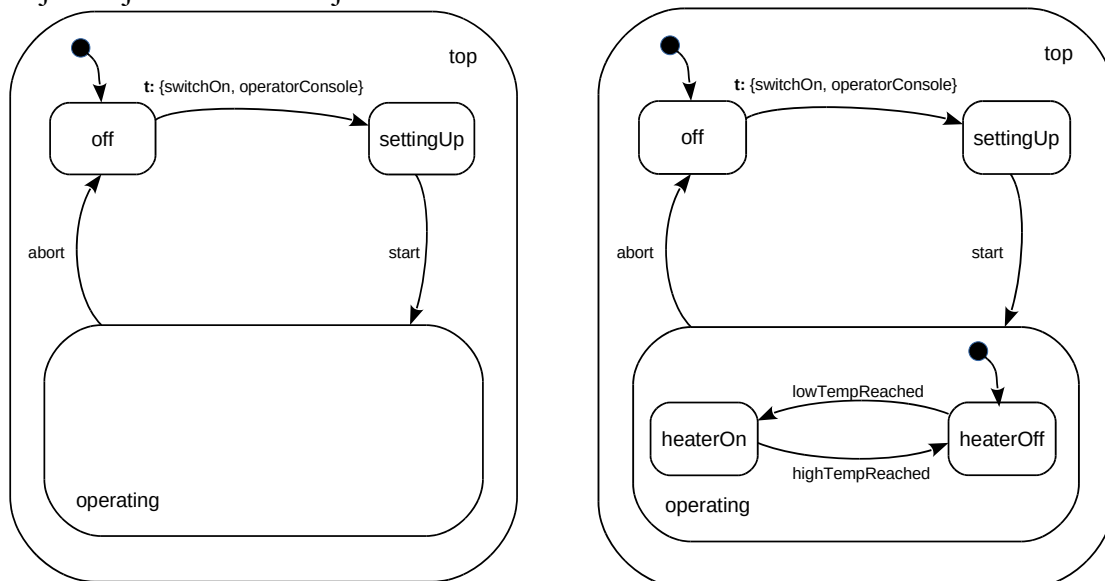
- Definicija klase aktera može da ima opcionu specifikaciju *interne sekvence poruka* (engl. *internal message sequence specification*) koja deklariše očekivanu sekvencu poruka između komponenata



- Puni opis poruke može da uključi vrednosti objekata podataka unutar poruke
- Sekvence specifikuju očekivane scenarije izvršavanja modela. Kada se akter izvrši, VM prijavljuje prekršaj stvarne sekvence u odnosu na specifikaciju. Može se koristiti za specifikaciju i verifikaciju zahteva

Hijerarhijske mašine stanja

- I stanje unutar mašine stanja predstavlja apstrakciju - vreme tokom kog mašina čeka na događaj ili vrši neku aktivnost, ali se i ta apstrakcija može hijerarhijski dekomponovati na podstanja koja definišu detalje ponašanja mašine unutar tog složenog stanja - hijerarhijske mašine stanja



- Tranzicija može da završava na nekom složenom stanju; okidanjem te tranzicije mašina stanja ulazi u podstanje tog složenog stanja na sledeći način (i dalje rekurzivno, ako je i to stanje složeno):
 - ako je složeno stanje označeno tako da se ulazi po istoriji (oznaka H u kružiću) i ako je mašina već boravila u tom složenom stanju (tj. u nekom njegovom podstanju), ulazi se u podstanje u kom se poslednji put boravilo kada je napušteno ovo složeno stanje
 - u suprotnom, ulazi se u inicijalno stanje (tj. okida se inicijalna tranzicija koja izlazi iz crnog kružića)

- Ako tranzicija polazi iz nekog složenog stanja, onda ona definiše podrazumevano ponašanje za svako podstanje unutar tog složenog stanja: kada se obrađuje poruka u datom podstanju, traži se tranzicija koja je omogućena i koja izlazi iz tog podstanja; ako takve nema, traži se tranzicija iz prvog okružujućeg stanja koja je omogućena i koja će biti okinuta, ako je ima; i tako dalje, ka spolja
- Na ovaj način okružujuće stanje definiše neko podrazumevano ponašanje na dati događaj u celom tom složenom stanju, ali se to ponašanje može (ali ne mora) redefinisati unutar podstanja (za isti događaj)
- Kada se odredi tranzicija koja se okida, izvršavaju se redom sve izlazne akcije pridružene stanjima koja se napuštaju (od ugnežđenih ka okružujućim), potom akcija pridružena tranziciji, i onda ulazne akcije pridružene stanjima u koja se ulazi, redom kako se ulazi (od spolja ka unutra)
- Kod internih tranzicija ne izvršavaju se izlazne i ulazne akcije stanja čija je tranzicija interna

Izvršavanje modela sa više aktera

- Ponašanja različitih aktera (reakcija na poruke) se izvršavaju konkurentno (multiprogramiranjem, multiprocesiranjem ili distribuirano)
- Hijerarhijska struktura aktera ne ograničava konkurentnost – komponente se izvršavaju konkurentno
- Poruke koje su poslate jednom akteru se izvršavaju jedna po jedna, svaki akter ima svoj red primljenih poruka; svaka poruka se obrađuje do kraja, nikada ne biva prekinuta (engl. *preempted*) obradom druge poruke u istom akteru, čak i ako je ona “višeg prioriteta” – tzv. *run-to-completion* semantika;
- Na taj način obrada poruke u jednom akteru je atomična, (logički) neprekidiva aktivnost, poruke istom akteru se sekvencijalizuju – nema konflikata unutar jednog aktera, što olakšava programiranje; zato nema potrebe brinuti o međusobnom isključenju unutar aktera, a svaki akter je svakako enkapsuliran, pa nema deljenih podataka sa drugim (konkurentnim) akterima
- Izvršavanje obrade poruke unutar jednog aktera može biti prekinuto obradom poruke (npr. višeg prioriteta) unutar nekog drugog aktera
- Okruženje (alat) omogućava opservabilnost i kontrolu izvršavanja više aktera uporedo, kao što je ranije pokazano za jedan akter

Nivo detalja

- Jezik nivoa detalja (engl. *detail-level language*) je konvencionalni OO programski jezik za specifikaciju detalja modela niskog nivoa inkorporiran u ROOM
- Da bi se rešili problemi diskontinuiteta opsega (engl. *scope discontinuity*), potrebno je da postoji mogućnost da se iz jezika nivoa detalja mogu koristiti usluge ROOM virtuelne mašine (ROOM API)
- Slanje poruke može se vršiti iz akcije mašine stanja neke klase aktera, a tu su dostupni (u opsegu su važenja) svi interfejsni i interni portovi te klase aktera
- Port se vidi kao objekat (npr. `port`) u jeziku detalja, a slanje poruke na port se radi kao poziv operacije tog porta:

```
port.send(value,currentTemp);
```

- Parametri ovog poziva su ime signala i opcioni objekat podataka čija se kopija šalje u poruci
- Unutar logičkog uslova tranzicije ili unutar akcije mašine stanja, dostupna je posebna podrazumevana varijabla `msg` koja predstavlja poruku koja se obrađuje u tom trenutku; u jeziku C++ deklarirana je kao pokazivač (`ROOMMessage*`):

```
bool HeatingController::isHighTempReached () {
    if (msg->data.getTemp() <= this->highTemp) return false;
    else return true;
}
```

- Ova funkcija je definisana u opsegu ponašanja date klase aktera, pa ima direktan pristup do varijabli i poruke svog aktera
- Podrazumevano slanje poruke je *asinhrono*: odmah nakon slanja poruke, procedura `send` vraća kontrolu kodu tranzicije koja nastavlja izvršavanje (ili se završava) nezavisno od obrade poslate poruke
- Slanje poruke može biti i *sinhrono*: slanje poruke sa `invoke` (umesto `send`) vraća kontrolu izvršavanju tranzicije tek kada je poslata poruka obrađena u odredišnom akteru (do tada je izvršavanje tranzicije blokirano)
- Definicija protokola je nezavisna od načina slanja poruke (sinhrono ili asinhrono)
- Pristup sistemskim SAP je slično kao za portove:

```
timer.informIn(duration);
```

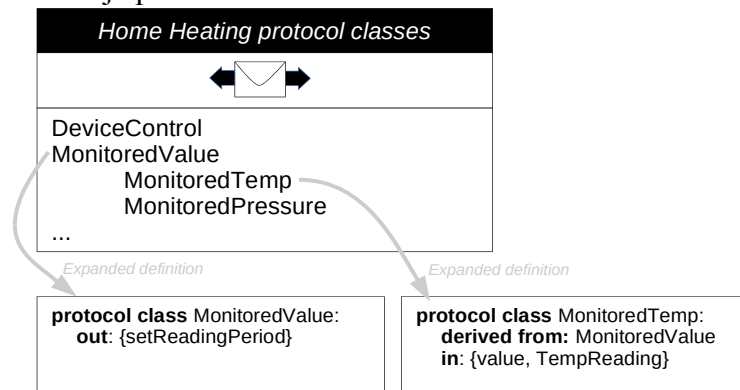
- SAP protokol `FrameService` pruža usluge kreiranja i uništavanja instanci klasa aktera; na primer, ako je u klasi aktera `HeatingSystem` definisan SAP frame tipa `FrameService`, kreiranje aktera `roomUnit` izgleda ovako:

```
frame.incarnate(roomUnit,initData);
```

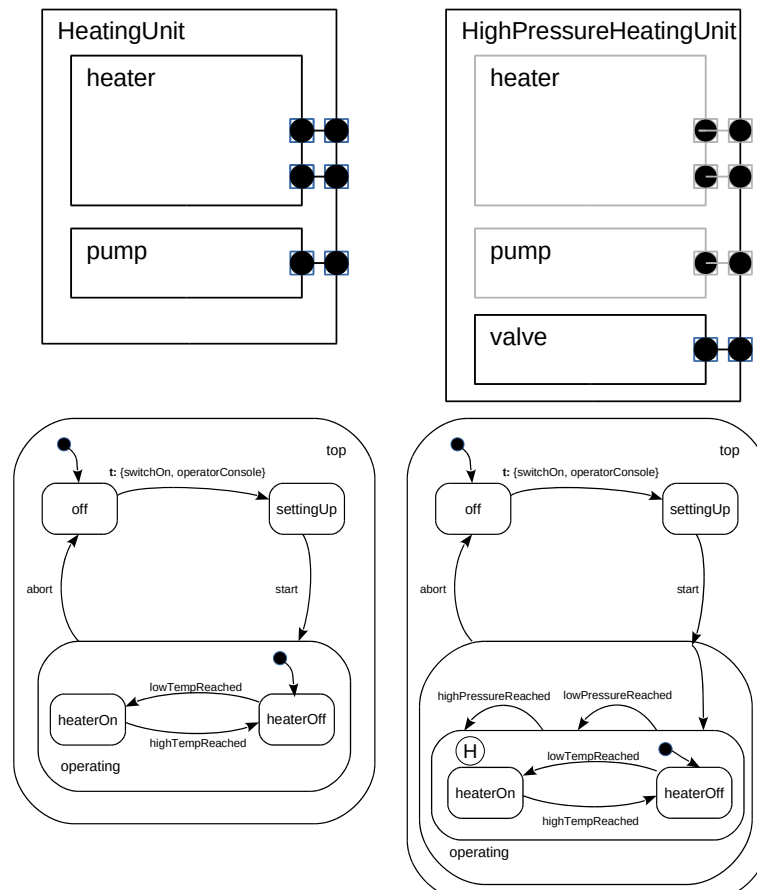
- Parametri ovog poziva su naziv aktera-komponente (dela, svojstva) i opcioni objekat podataka koji se prosleđuje u inicijalizacionoj poruci koja okira inicijalizacionu tranziciju mašine stanja aktera koji se kreira

Nasleđivanje

- ROOM dosledno podržava nasleđivanje svih vrsta klasa: aktera, protokola i podataka
- Podrazumevano, izvedena klasa nasleđuje, ali može i da redefiniše svojstva osnovne klase
- Primer nasleđivanja protokola:



- Klasa aktera nasleđuje interfejs, internu strukturu i ponašanje (mašinu stanja), ali može (a ne mora) da proširi interfejs dodavanjem novih portova (ali ne može da izbaci neki nasleđeni port)
 - proširi internu strukturu aktera, ili je potpuno promeni
 - proširi ili izmeni mašinu stanja
- Ovo obezbeđuje doslednu primenu pravila supstitucije: instanca izvedene klase aktera može se upotrebiti gde god se očekuje instanca osnovne klase aktera
- Na primer:



VI Praktikum

Rational Rose Technical Developer

- IBM Rational Rose Technical Developer (ranije pod nazivom Rational Rose Real Time) predstavlja integrisano MDD okruženje koje podržava automatsku translaciju između modela i koda za različite programske jezike, kao što su Java, C i C++. Neke od podržanih mogućnosti ovog programskog paketa su:
 - Mogućnosti izvršavanja, testiranja i vizuelnog debugovanja modela
 - Kolaboracija više učesnika projektnog tima
 - Integracija sa razvojnim okruženjem Eclipse
 - Mogućnost reverznog inženjerstva
 - Integracija sa velikim brojem RT operativnih sistema i *embedded* razvojnih okruženja
 - Instalacija pod operativnim sistemima Windows i Linux
- U daljem tekstu predstavljen je jedan jednostavna ugrađen (engl. *embedded*) sistem, na kome će biti prikazane mogućnosti razvoja jednog RT UML modela pomoću programskog okruženja Rational Rose Technical Developer ver. 7.0. Za jezik nivoa implementacije upotrebljen je Java programski jezik pod operativnim sistemom Windows.

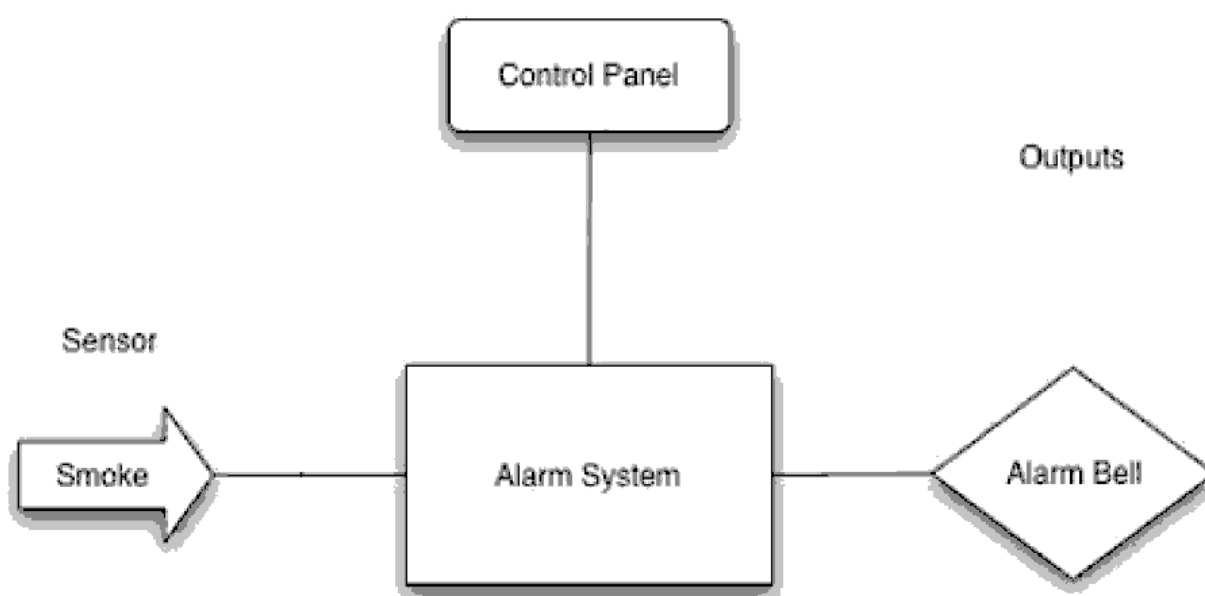
Podešavanje okruženja

- Pre pokretanju razvojnog okruženja potrebno je instalirati sleće alate:
 - Java JDK (engl. *Java Development Kit*) -
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
 - *nmake.exe* – postoji u okviru instalacije okruženja *Microsoft Visual Studio*
- Podesiti promenljive okruženja, tako da putanja bude podešena prema java JDK i *nmake.exe* alatu.
- Jedan jednostavan način je kreiranje skript fajla (*.bat*) koji izvršava sva neophodna podešavanja i koji pokreće Rose RT razvojno okruženje. U nastavku je dat primer jednog takvog skripta:

```
set Path=PATH=%PATH%;c:\Program Files\Java\jdk1.6.0_04\bin\  
  
call "c:\Program Files\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat"  
  
start /d "C:\Program Files\Rational\Rose RealTime\bin\win32\" roseRT.exe  
  
exit
```

Postavka zadatka

- Potrebno je projektovati jednostavan alarm sistem šematski prikazan na sledećoj slici.



- Sistem automatski obaveštava o potencijalnoj mogućnosti izbijanja požara. Senzor za detekciju dima generiše određeni prekid na koji sistem (u koliko je u aktivnom stanju) treba da reaguje tako što uključuje alarm. Dužina trajanja alarma kao i vrsta sirene kojom će se oglasiti mogu se zadati putem kontrolne table, npr. dužina trajanja alarma $t_a = 60s$, sirena periode $T_s = 1s$ ($T_{s1} = 400ms$ poluperioda naizmenično uključene i $T_{s2} = 600ms$ poluperioda isključene sirene).

Pretpostavke

- Kontrolna tabla se sastoji od jednog tastera i jednog svetlosnog signalizatora, pri čemu različita signalizacija označava drugo stanje sistema:
 - Crveno* – ukazuje da je sistem u neaktivnom stanju (za prelazak iz ovog stanja potreban je pritisak na taster čije je trajanje duže od 1s).
 - Zeleno* – $t_a = 60s$, $T_s = 1s$ ($T_{s1} = 400ms$, $T_{s2} = 600ms$)
 - Zeleno trepćuće* – $t_a = 90s$, $T_s = 500ms$ ($T_{s1} = 250ms$, $T_{s2} = 250ms$)
- Sistem prelazi iz jednog aktivnog stanja u drugo po gore navedenom redosledu kratkim pritiskom na taster čije je trajanje kraće od 1s.
- Sistem se inicijalno nalazi u aktivnom stanju.
- Sistem prelazi iz aktivnog u neaktivno stanje pritiskom na taster kontrolne table trajanja dužeg od 1s.

-
- Dužina pritiska tastera kontrolne table meri se na način tako što se generišu dva prekida, jedan pri pritisku i jedan pri otpuštanju tastera.
 - Alarm sirena se aktivira kratkim signalom i ostaje aktivna sve dok se ne dovede ponovo signal na njen ulaz.

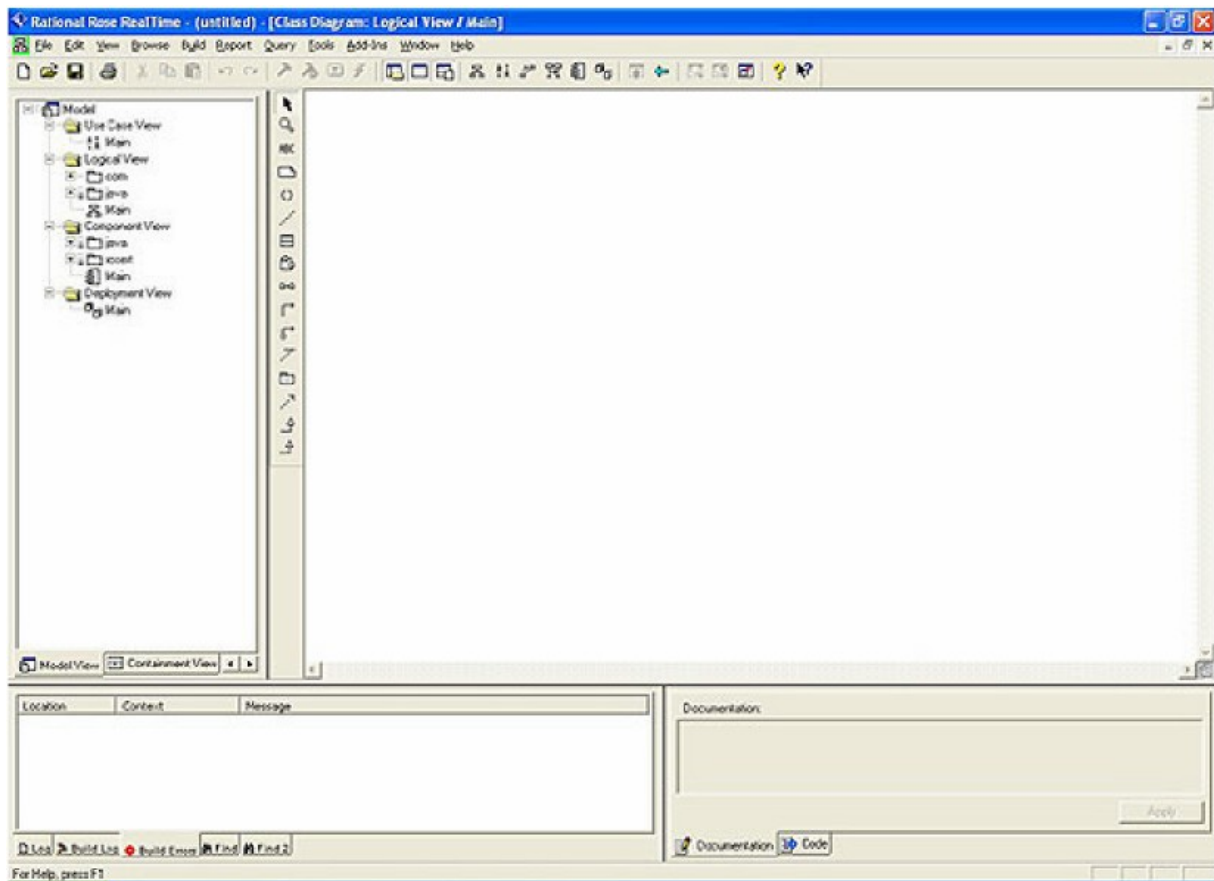
Kreiranje modela

Kreiranje novog modela

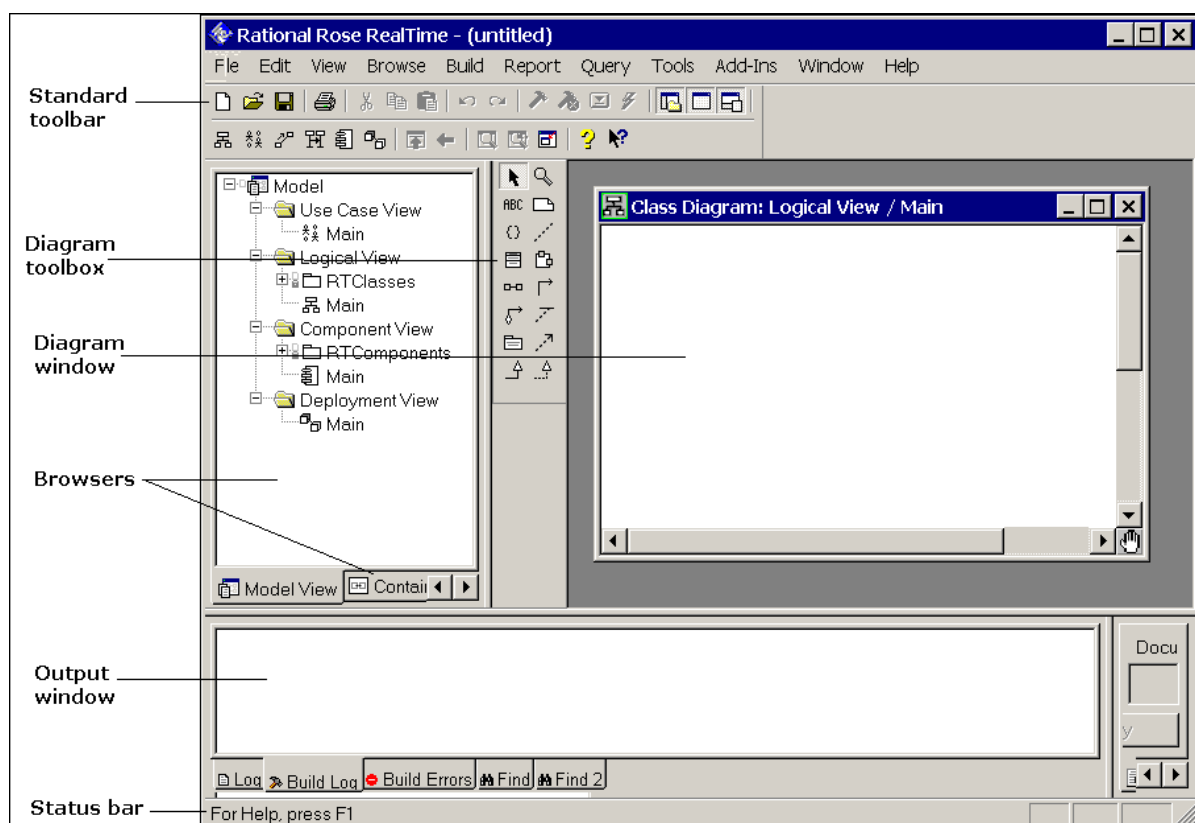
1. Pokrenuti Rational Rose Real Time prema prethodno datom uputstvu.
- Na početku će se pojaviti dijalog (kao na slici) za odabir ciljnog programskog okruženja. Odabrati **RTJava**.





- Pojaviće se prazan model kao na sledećoj slici.




- Osnovni prozor podeljen je u četiri sekcije:
 - Sa leve strane je *browser* koji služi za navigaciju kroz celokupan model.
 - Glavni panel, tj. dijagram prozor sa desne strane
 - Ispod se nalaze *output* i *code/documentation* prozori.

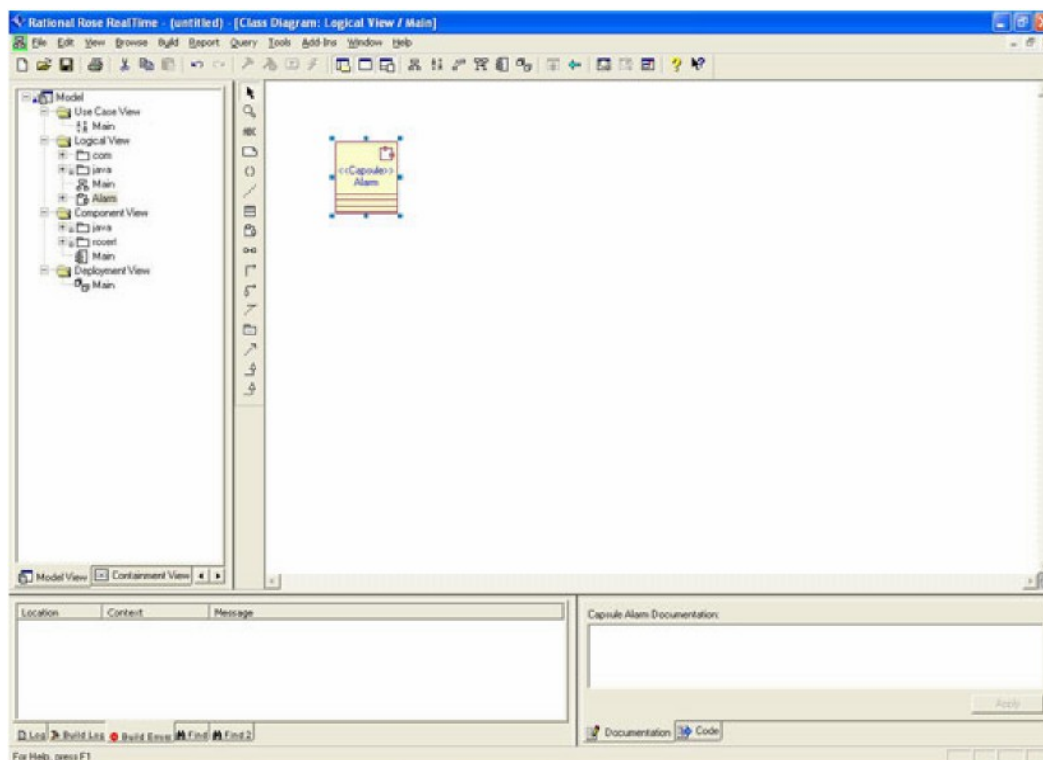


Kreiranje dijagrama slučaja upotrebe

1. U okviru strukture modela, proširiti **Use Case View** i otvoriti komponentu **Main**.
2. Izabrati alatku **Use Case**  iz kutije sa alatima i prevući na dijagram.
3. Preimenovati u *FireDetectorMainFlow*
4. Izabrati alatku **Text**  i uneti kratak opis sistema.

Kreiranje kapsula (aktera)

- Prvi zadatak je kreiranje kapsula koje sačinjavaju sistem. Postoje 4 osnovna aktera (*capsules*) u sistemu: senzor za detekciju dima, sirena (alarm), kontrolna tabla/tastatura i sam alarmni sistem.
- Kreiranje kapsule:
 1. U okviru brauzera modela, proširiti **Logical View** i otvoriti komponentu **Main**.
 2. Prevući alatku **Capsule**  iz **Diagram toolbox**-a na dijagram.
 3. Imenovati kapsulu sa **Alarm**.
- Trebalo bi dobiti dijagram prikazan na sledećoj slici:



- Ponoviti postupak za sledeće elemente:
 1. Keypad
 2. SmokeDetector
 3. FireAlarm
- U ovom trenutku definisani su osnovni akteri u sisemu, tri koja definišu aktore vezane za spoljne uređaje i jedan koji definiše ponašanje samog sistema.

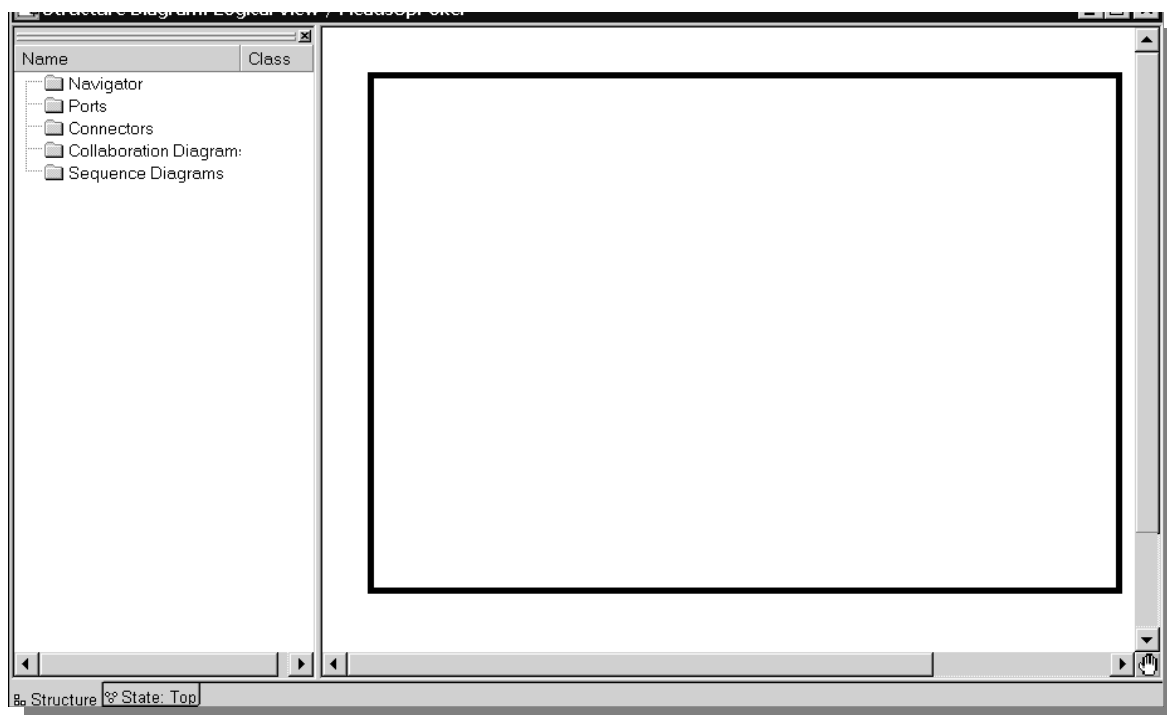
Kreiranje dijagrama strukture za kapsule

- Sada je potrebno kreirati strukturni dijagram koji implementira ponašanje opisano u *use case* dijagramu. Zatim je potrebno kreirati dijagram sekvence koji opisuje interakciju kapsula.
- S obzirom na to da kapsula **Alarm** opisuje tok događaja opisanih u osnovnom *use case* dijagramu koji se implementira, on takođe enkapsulira sve kapsule i klase relevantne za sistem.

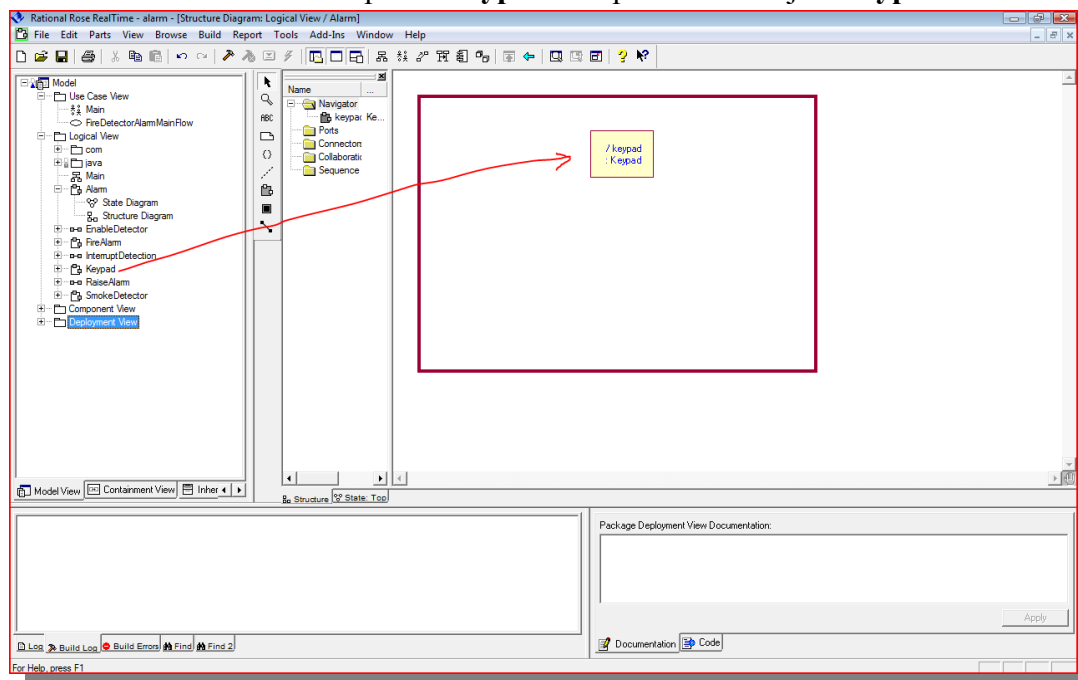
Kreiranje strukture kapsule Alarm

- Moguće je na dva načina agregirati kapsule u okviru neke druge kapsule:
 - Pomoću alatke **Aggregation** u klasnom dijagramu.
 - Prevlačenjem kapsule na strukturni dijagram druge kapsule.
- Ovde će biti upotrebljen drugi način:

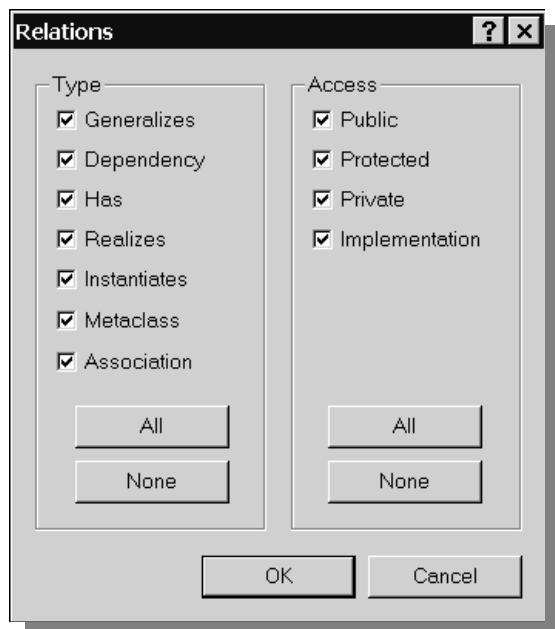
1. U okviru **Model View** taba brauzera kliknuti na znak plus [+] pored ikonice da bi se raširila kapsula **Alarm**.
2. Pojaviće se dve stavke: jedna koja reprezentuje dijagram stanja i druga vezana za dijagram strukture.
3. Dvostruki klik na ikonicu **Structure Diagram** kako bi se otvorio sledeći prozor dijagrama.



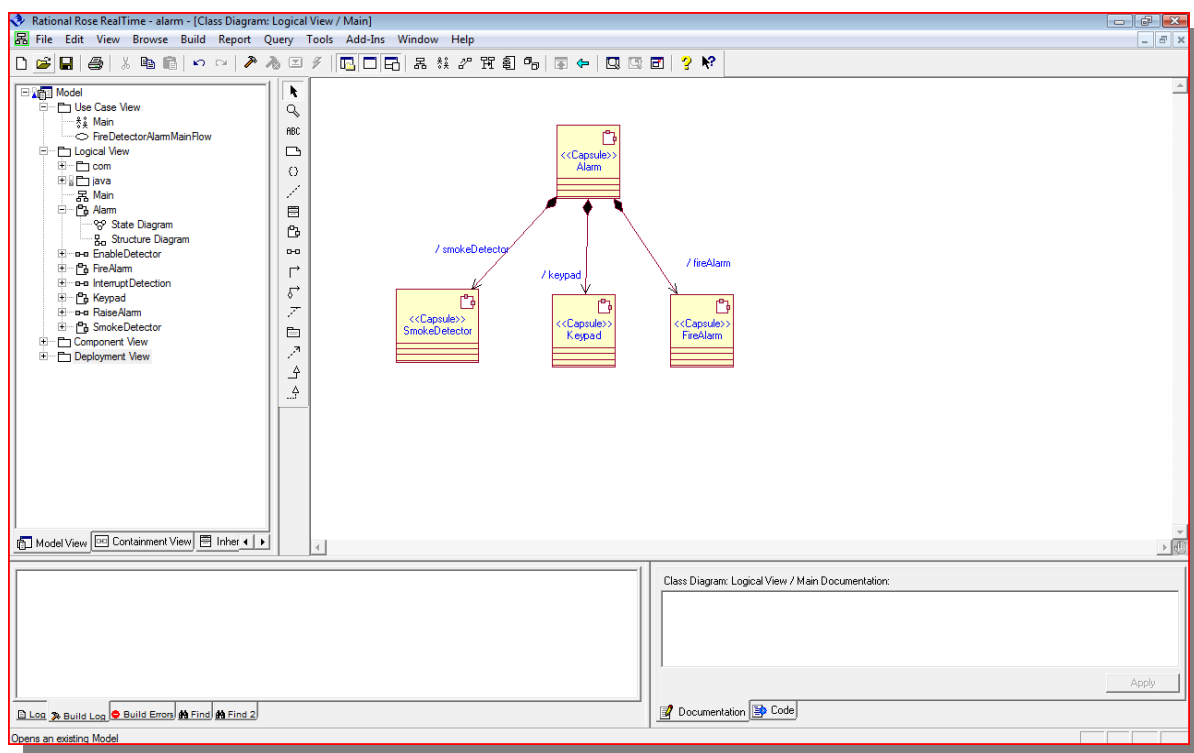
4. Prevući kasulu **Keypad** iz taba **Model View** brauzera na strukturni dijagram na **Alarm**.
5. Dvostruki klik na rolu kapsule **/keypadR1** i preimenovati je u **keypad**.



6. Ponoviti postupak za kapsule **SmokeDetector** i **FireAlarm**
7. U folderu **Logical View**, dvostruki klik na dijalog **Class Diagram: Logical View / Main**. Trebalo bi da se prikažu prethodno kreirane kapsule.
8. Iz menija **Query** odabrati **Filter Relationships**

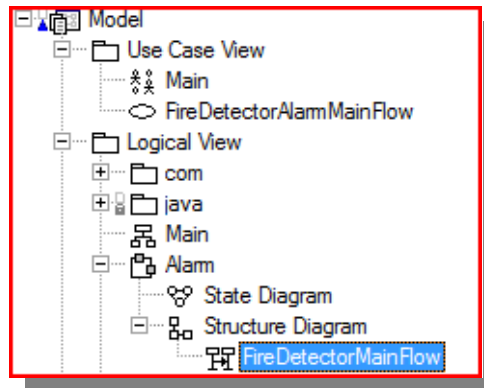


9. Proveriti da li su selektovane sve **Type** i **Access** grupe i kliknuti na **OK**.




Kreiranje dijagrama sekvence

- Potrebno je kreirati dijagram sekvence na kome će se prikazati interakcija određenih kapsula sistema, radi implementacije datog ponašanja opisanog u slučaju upotrebe **FireDetectorMainFlow**.
 - Tab Model View, proširiti Logical View, desni klik na **Alarm** i odabrati **Sequence Diagram**.
 - Preimenovati u **FireDetectorAlarmMainFlow**.



- Prevući alatku **Interaction Instance** i odabrati **SmokeDetector**
- Ponoviti za **FireAlarm** i **Keypad**
 - Samostalno opisati interakciju kapsula u sistemu pomoću datog dijagrama sekvence
(pomoć **Help: Contents: Toolset Guide: Creating Sequence Diagrams**)

Kreiranje protokola

- Otvoriti klasni dijagram **Main**
- Prevući latku **Protocol** .
- Imenovati u **EnableDetector**.

Kreiranje signala za dati protokol

- Desni klik na protokol i odabrati **Open Specification**.
- Odabrati tab **Signals**.



6. Desni klik na panel **In Signals**, odabrati **Insert**.
7. Preimenovati signal u **Enable**.
8. Kliknuti na kolonu **Data Class** pored signala **Enable** signala i pritisnuti F8.
9. Odabrati **boolean** iz liste.
10. Na isti način kreirati sledeće protokole:

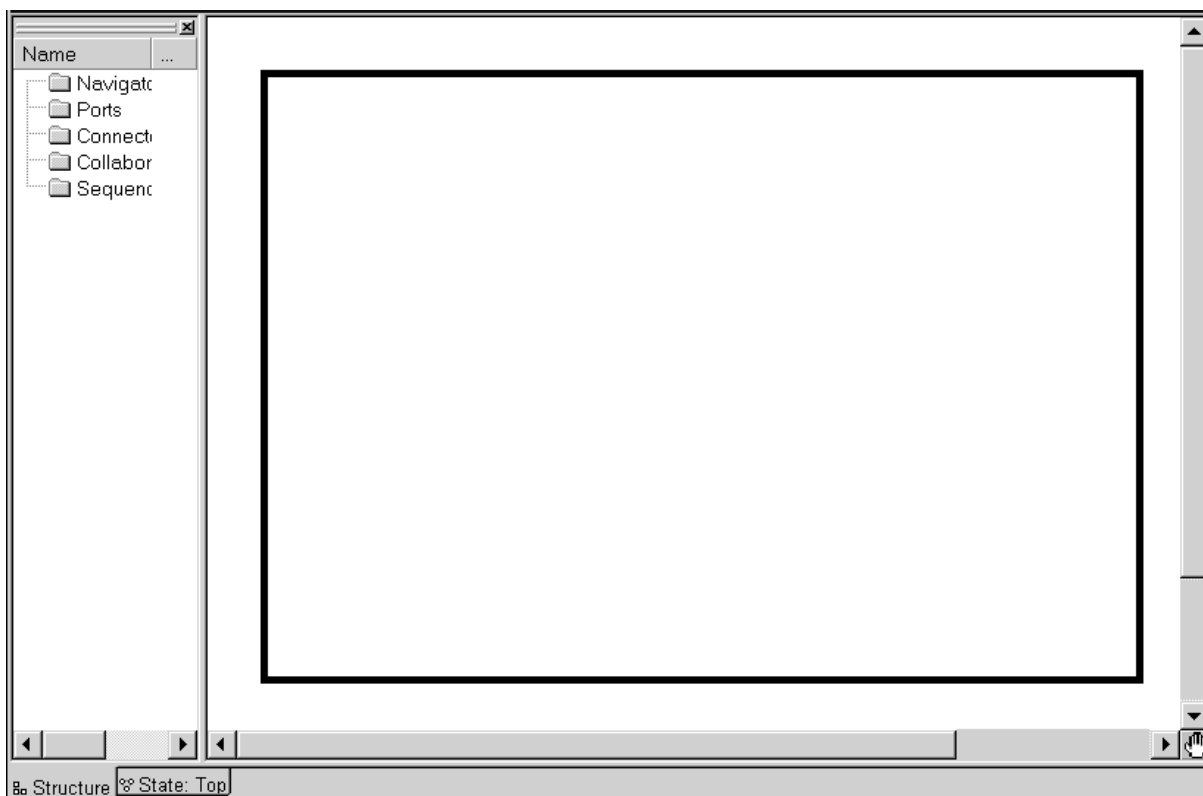
Protokol	InSignals	OutSignals
InterruptDetection	Interrupt:void	
RaiseAlarm		AlarmSignal:void


Kreiranje portova i konektora

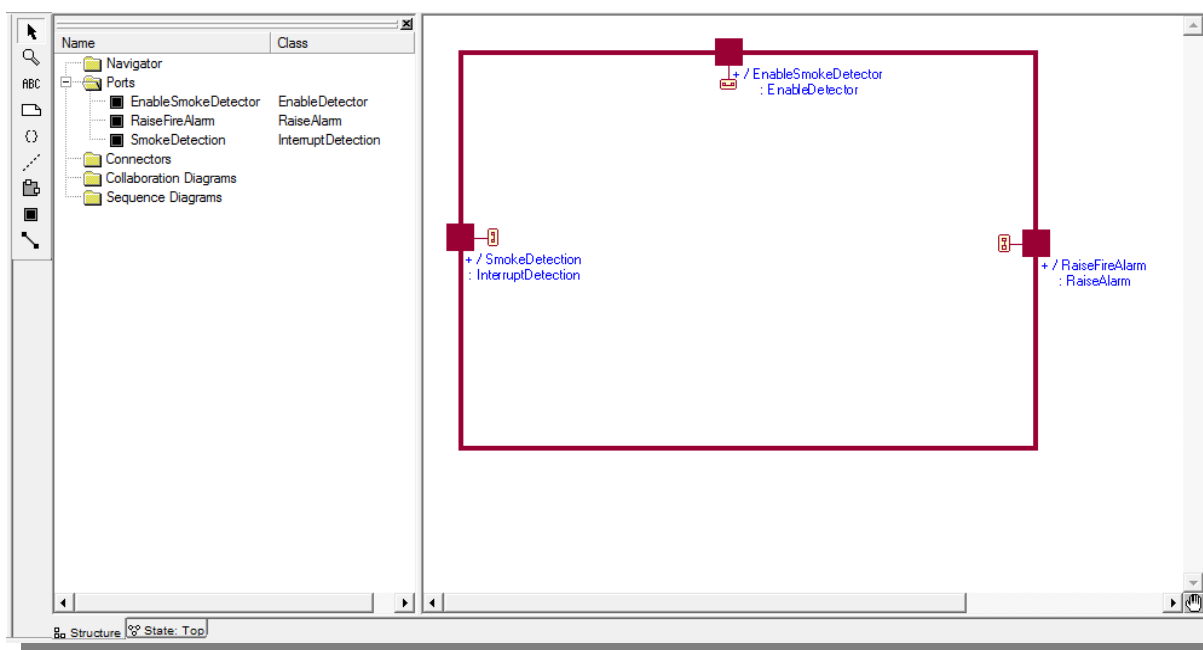
- Portovi su atributi kapsula, pa kao takve, sve role (instance) kapsula imaju dostupne iste portove. Česta greška je da se portovi dodaju rolama kapsula.

Kreiranje portova

1. Otvoriti **Structure Diagram** za kapsulu **SmokeDetector**.

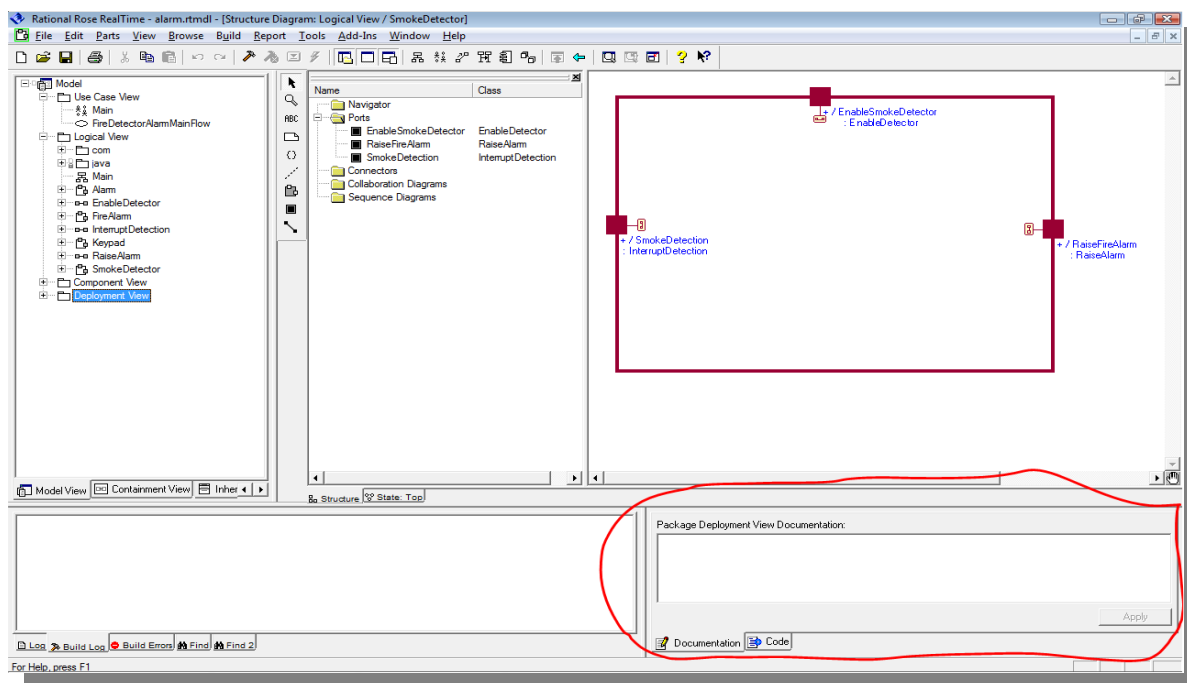


2. Iz taba **Model View** brauzera, prevući protokol **EnableDetector** na crnu ivicu strukturnog dijagrama kapsule **SmokeDetector**.
3. Desni klik na novokreirani port, odabrati **End Port**. Port će promeniti izgled u . Šta predstavlja **End Prprt**?
4. Preimenovati u **EnableSmokeDetector**.
5. Uraditi isto za **InterruptDetection** i **RaiseAlarm** kao na slici.



Dokumentovanje

- Pri projektovanju modela dobra praksa je dokumentovati svaki element. *Rose RealTime* omogućava da svaki element može sadržati dokumentaciju.
- Uneti za svaki element kratak opis ponašanja pomoću prozora **Documentation** označenog na sledećoj slici.



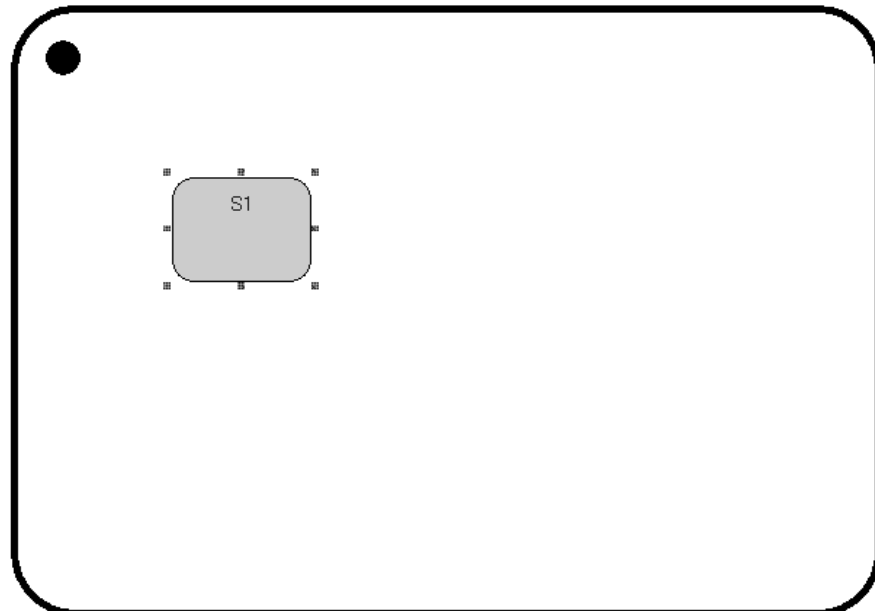
Dodavanje ponašanja kapsuli

- Ponašanje kapsule opisano je dijagramom stanja koji se automatski pridodaje pri njenom kreiranju.
 1. Dvostruki klik na **State Diagram** za otvaranje dijagrama stanja za kapsulu **SmokeDetector**.




Kreiranje stanja


2. Kliknuti na **State** tool  iz **State Diagram** alatki i prevući na dijagram stanja.



3. Preimenovati stanje u **Enabled**.
4. Na isti način dodati stanje **Disabled**.

Kreiranje tranzicija

5. U **State Diagram** kutiji sa alatkama kliknuti na alatku **State Transition** .

6. Prevući tranziciju sa **Initial Point**  na stanje **Enabled**.
7. Imenovati tranziciju **Initial**.
8. Na isti način dodati tranzicije između stanja **Enabled** i **Disabled** i obratno i nazvati ih **enabled** i **disabled**, respektivno.
9. Simbol $\overline{\text{T}}$ predstavlja da tranzicija nema definisan okidač (*trigger*).

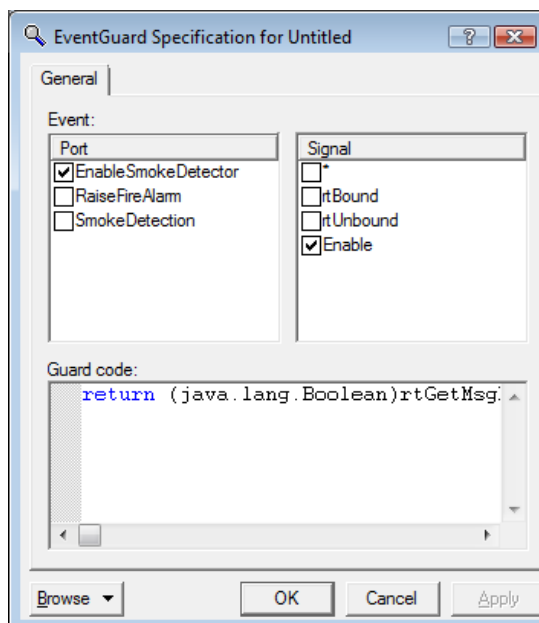
Kreiranje okidača

10. Dvostruki klik na tranziciju **enabled**.
11. Odabrati tab **Triggers**.



12. Desni klik na listu i odabrati **Insert**.
13. Odabrati **EnableSmokeDetector**
14. U polju za **Signal** štiklirati **Enable**
15. U polju **Guard code** uneti sledeći kod:

```
return (java.lang.Boolean) rtGetMsgData () == true;
```



16. Na isti način definisati okidač na tranziciji **disabled**, s tim što u **Guard code** treba uneti:

```
return (java.lang.Boolean)rtGetMsgData()==false;
```

Dodavanje akcije

17. U kutiji sa alatima **State Diagram** kliknuti na alatku **Transition to Self**.
18. Odabrati stanje **Enabled**.
19. Imenovati tranziciju **SmokeDetected**.
20. Dvostruki klik na tranziciju.
21. Dodati okidač na signal **Interrupt** porta **SmokeDetection**.
22. Odabrati tab **Action** i uneti sledeći kod:

```
RaiseFireAlarm.AlarmSignal().send();
```

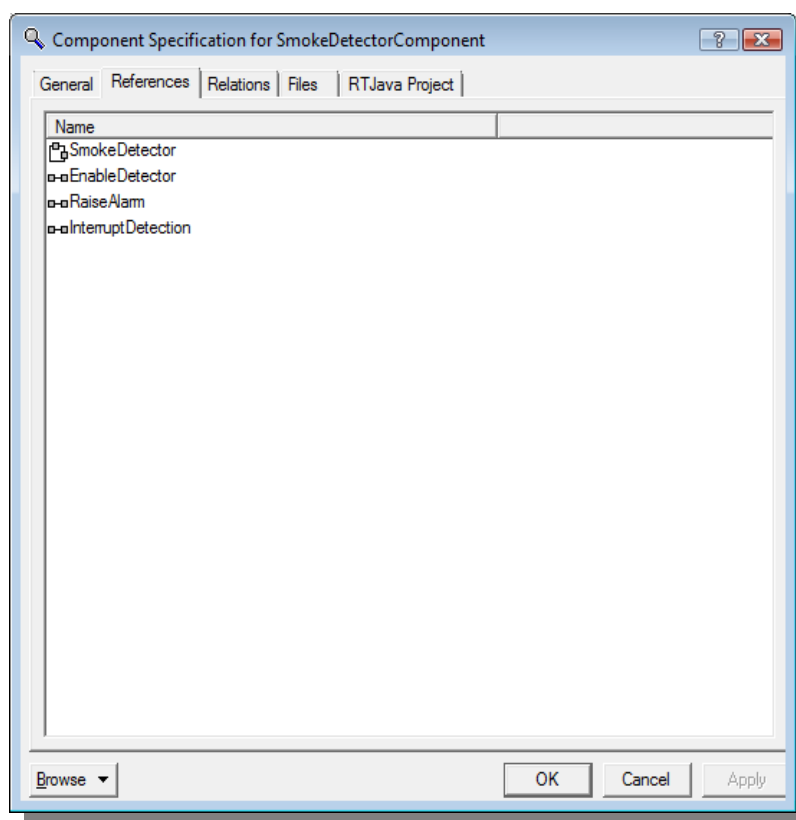
Šta predstavlja dati kod?

Kompajliranje i izvršavanje

- Šta je to MDD i zašto je bitno u što ranijoj fazi izvršiti kompajliranje, izvršavanje i testiranje modela?

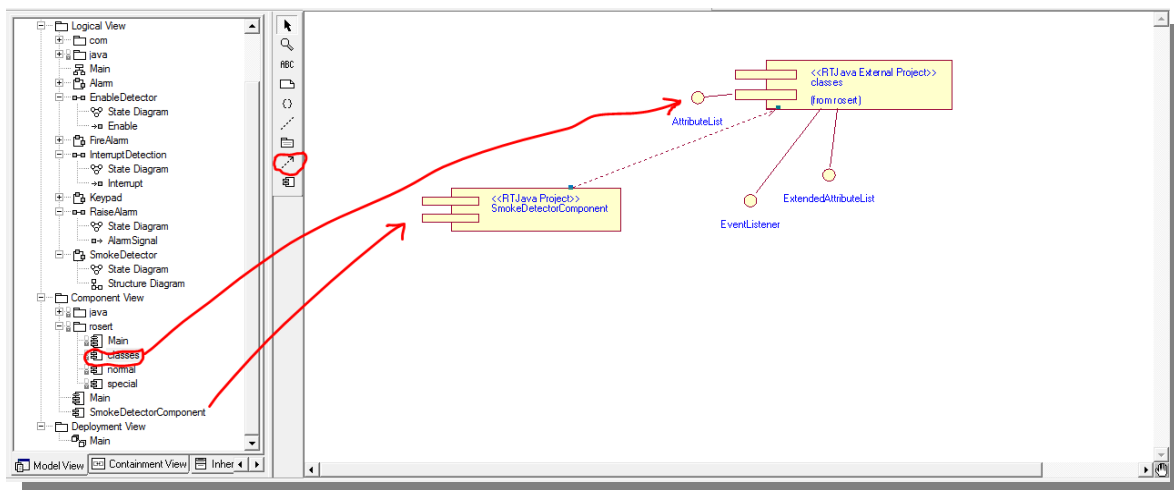
Kreiranje komponente

1. U brauzeru **Model View**, desni klik na **Component View**, zatim **New > Component**.
2. Preimenovati u **SmokeDetectorComponent**.
3. Dvostruki klik na ovu komponentu da bi se otvorio dijalog **Component Specification**.
4. **OBAVEZNO!** Pod tabom **General** postaviti polje **Type** na **RTJavaProject**.
5. Pod tabom **References** prevući iz brauzera **Logical View** kapsulu **SmokeDetector** kao i sve kreirane portove.
6. Kliknuti na OK



OBAVEZNO!

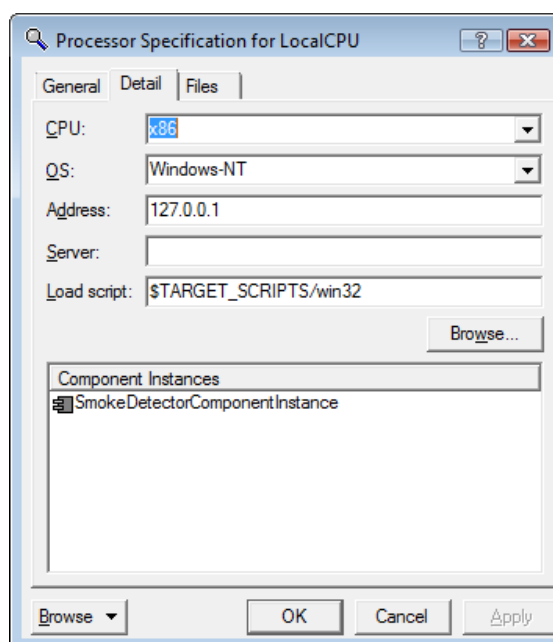
7. Otvoriti **Component View/Main** dijagram i prevući **SmokeDetectorComponent** i komponentu **classes** iz paketa **rosert** / **Component View**
8. Postaviti zavisnost (**Dependency**) od **SmokeDetectorComponent** ka komponenti **classes**.

**Kompajliranje**

1. Desni klik na **SmokeDetectorComponent** (u **Component View**).
2. **Build > Build...**
3. **OK**.
4. U koliko je kompajliranje uspešno nastaviti dalje, u suprotnom proveriti ceo postupak.

Izvršavanje

1. Desni klik na **Deployment View**
2. **New>Processor**
3. Imenovati **LocalCPU**
4. Dvostruki klik na **LocalCPU** i odabrati tab **Detail**
5. Desni klik na **Component Insatances**, odabrati **Insert**
6. Odabrati **SmokeDetectorComponent**.



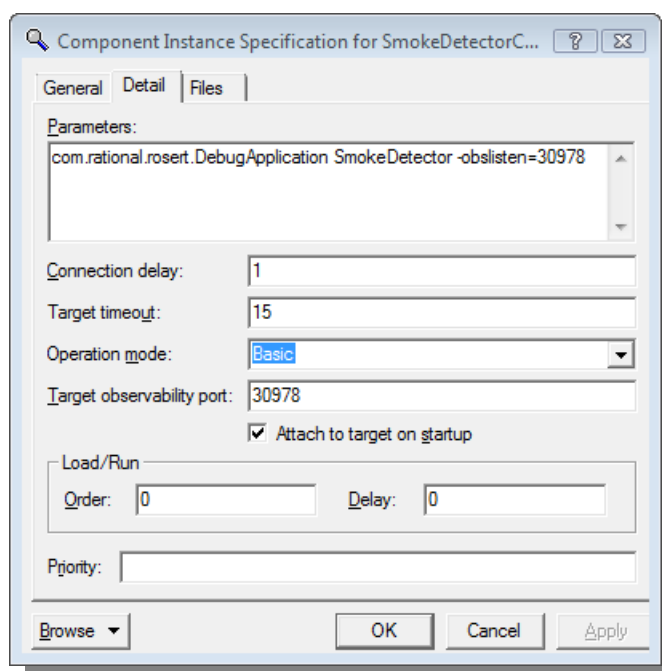
7. OK.

8. Dvostruki klik na **SmokeDetectorComponentInstatnce** u okviru **Deployment View / LokalCPU**

OBAVEZNO!

9. U polje **Parameters** uneti:

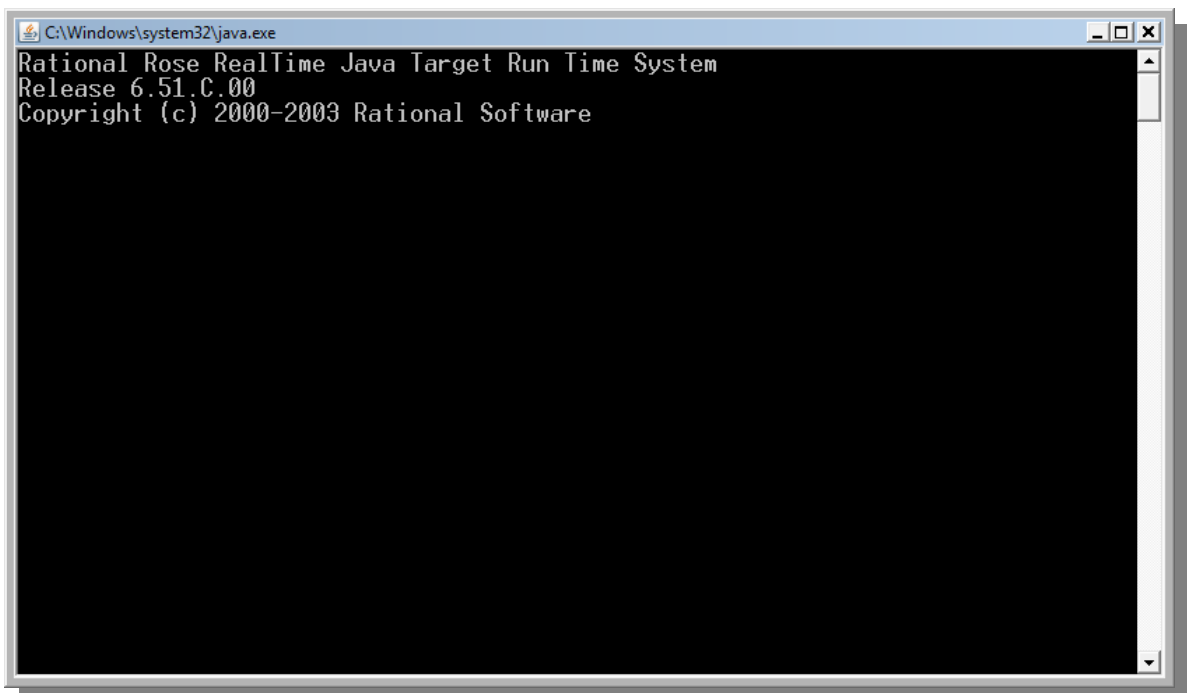
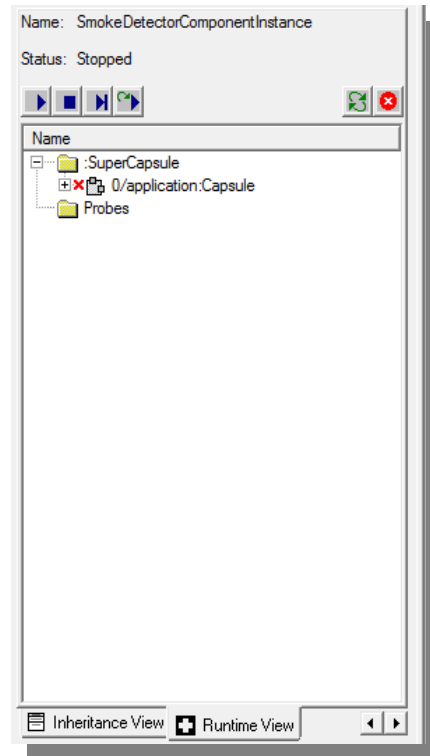
```
com.rational.rosert.DebugApplication SmokeDetector -  
obslisten= <br. porta koji se nalazi polju Target  
observabilitiy port>
```




Napomena: Prvi parametar označava ime osnovne (*top-level*) kapsule koja se pokreće (u ovom slučaju **SmokeDetector**), a drugi označava broj porta koji osluškuje okruženje (**uneti bez razmaka**).

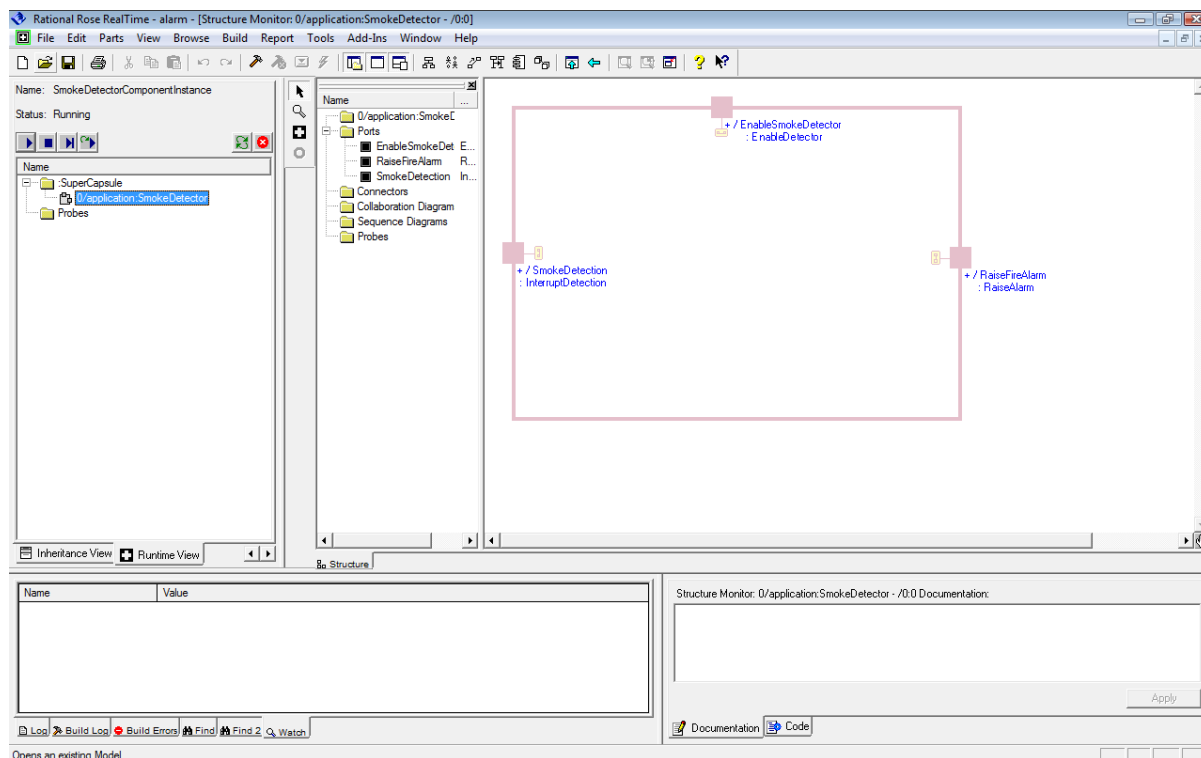
10. OK

11. Desni klik na **SmokeDetectorComponentInstance**, odabrati **Run**.

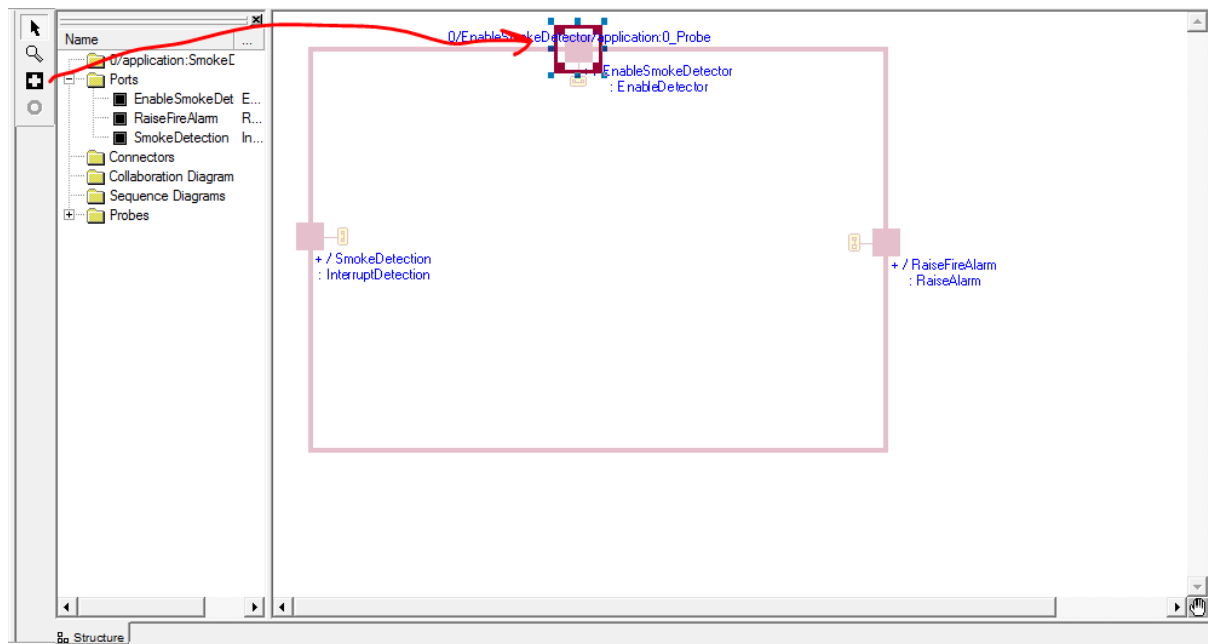


Testiranje

1. Pritisnuti **Start** dugme  da bi se učitala komponenta.
2. Desni klik na instancu kapsule **SmokeDetector** i odabrati **Open Structure Monitor**.

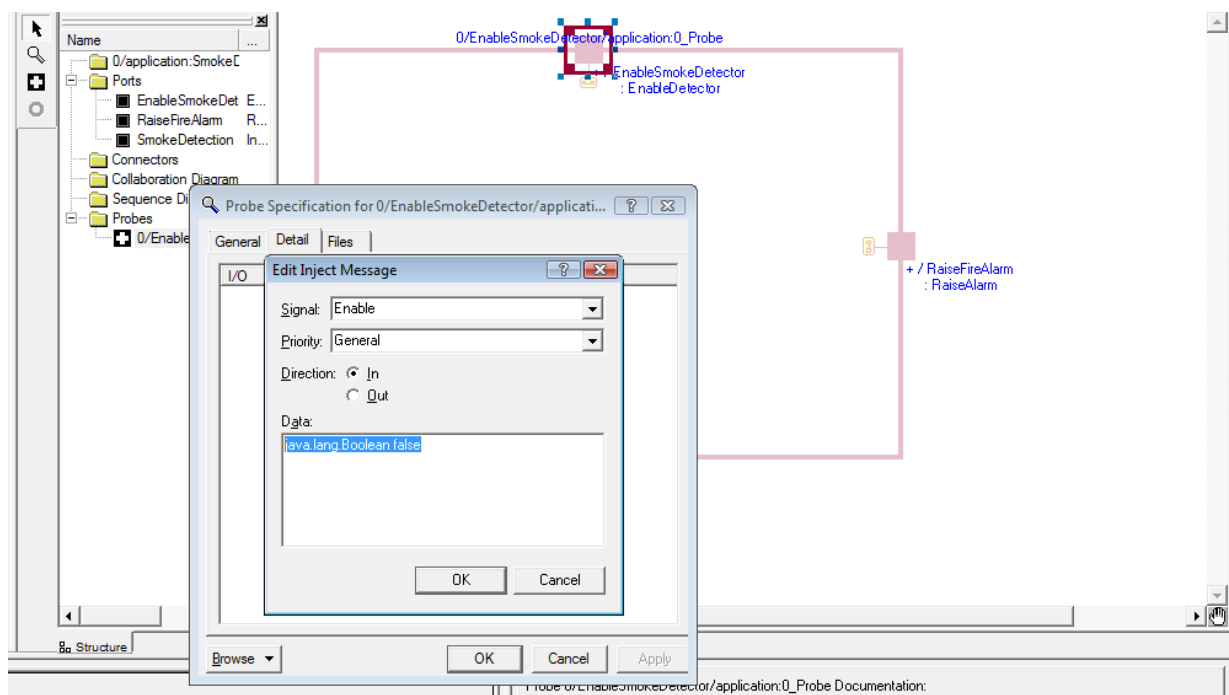


3. Odabrati alatku **Probe** iz kutije **Structure Monitor**.
4. Pozicionirati kurzor na port **EnableSmokeDetector** na dijagramu **Structure Monitor** i zatim kliknuti.



5. Na dijagramu **Structure Monitor** proširiti **Probes**, desni klik na jedinu ponuđenu sondu (*probe*) (za port **SmokeDetection**) i izabrati **Open Inject**.
6. Desni klik na listu pa **Insert**.
7. U **Data** polje uneti:

```
java.lang.Boolean false
```



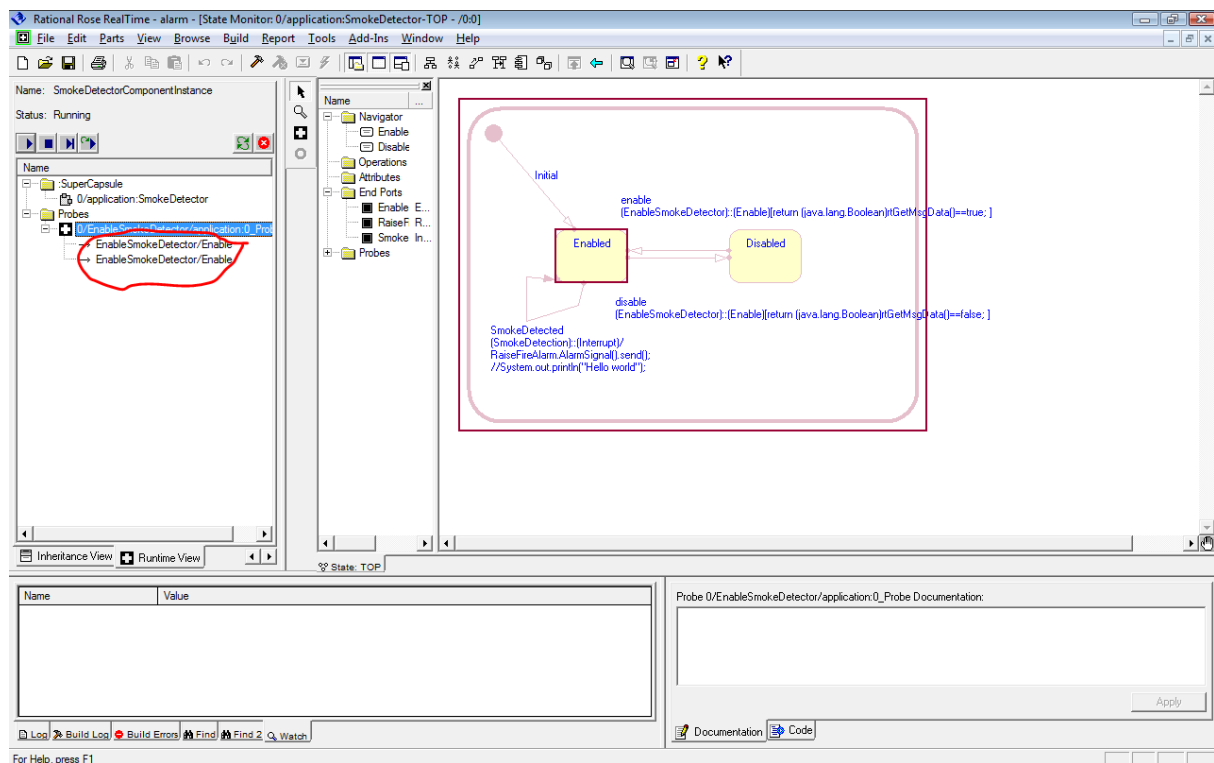
8. Ponoviti za vrednost

```
java.lang.Boolean true
```

9. **OK.**

10. Desni klik na instancu kapsule **SmokeDetector** i odabrati **Open State Monitor**.

11. Raširiti kreiranu sondu.



12. Desni klik na jednu od vrednosti i odabrati **Inject**.

13. Naizmenično zadavati jednu pa drugu injektovanu vrednost i posmatrati promene stanja sistema.

Šta će se dogoditi ako se dva puta zada ista vrednost? Zašto?

14. Zaustaviti izvršavanje pomoću dugmeta **Shutdown**.

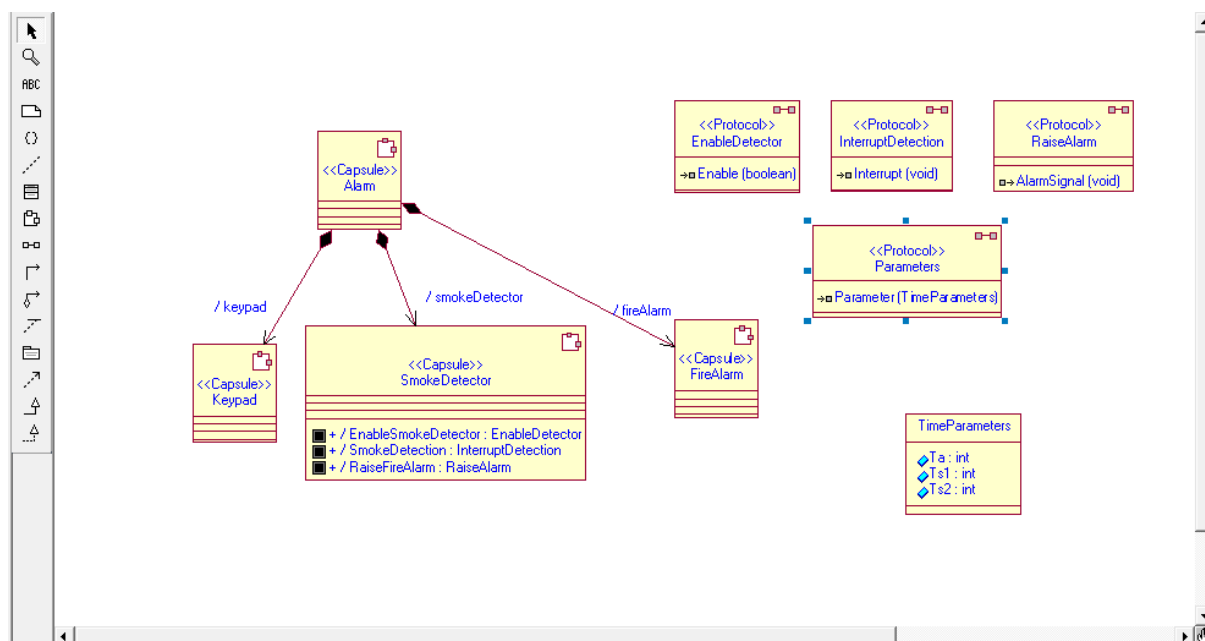
Proširenje modela

- Do sada je implementiran i testiran akter **SmokeDetector**. Potrebno je na isti način to uraditi i za aktere **FireAlarm** i **Keypad**.
 - U okviru dijagrama klasa **Logical View: Main** dodati klasu **TimeParameters**, koja predstavlja parametre koji se definišu pomoću kontrolne table, sa sledećim poljima:

Ime	Vidljivost	Tip
Ta	public	int
Ts1	public	int
Ts2	public	int

Napomena: Zbog jednostavnosti sva polja su definisana kao **public**.

- Na prethodno prikazan način definisati protokol **Parameters** sa jednim **In** signalom **Parameter** tipa **TimeParameters** prethodno definisane klase.



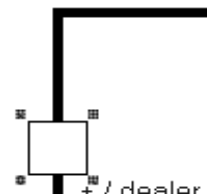
- Na prethodno prikazan način u okviru **Structure Diagram** kapsule **FireAlarm** definisati sledeće potove:

Ime

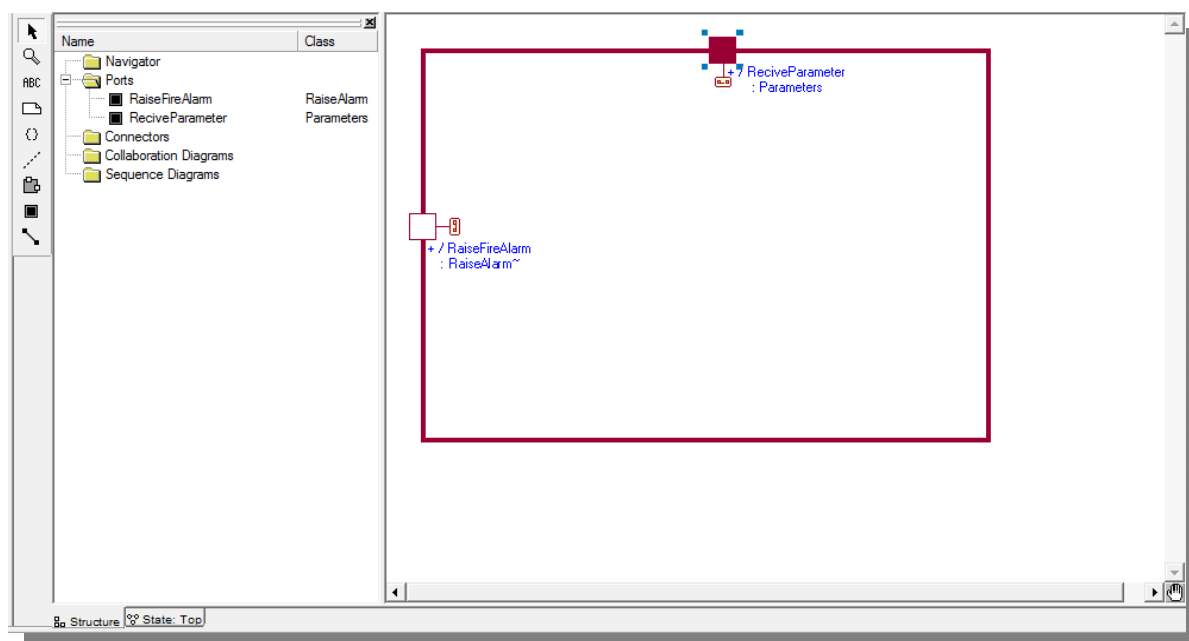
Tip

RaiseFireAlarm	RaiseAlarm
ReceiveParameter	Parameters


- Postaviti da su tipa **End Port**.



- Desni klik na port **RaiseFireAlarm**, odabrati **Conjugate**.
Zašto?



Kreiranje *timing* porta

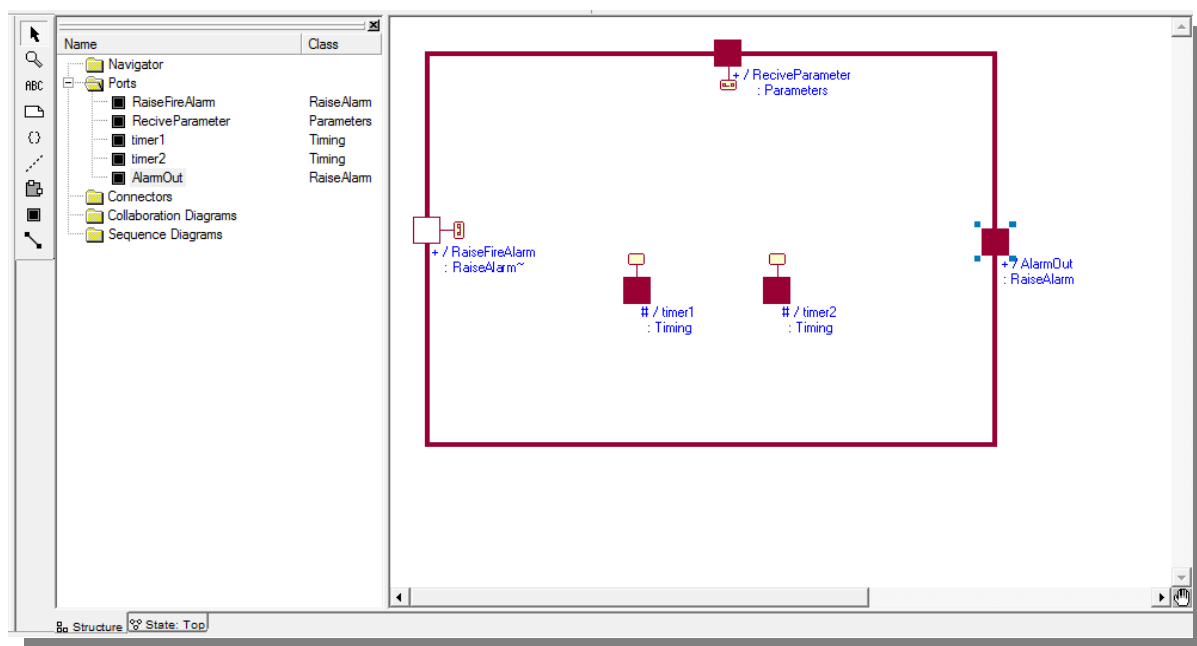
- Iz kutije **Structure Diagram** odabrati alatku **Port**  i prevući unutar strukturnog dijagrama kapsule **FireAlarm**.

Napomena: Ne prevlačiti na ivicu već unutar strukture kapsule **FireAlarm**. Dodavanje porta na ivicu kreira *public* port, dok dodavanje unutar strukture stvara *protected* port.
Zašto?

- Odabrati protokol **Timing** iz liste ponuđenih protokola.




3. Imenovati port **timer1**.
4. Na isti način dodati port **timer2**.
5. Dodati još jedan port tipa **RaiseAlarm** i imenovati ga **AlarmOut**, koji će predstavljati izlaz na alarm sirenu.

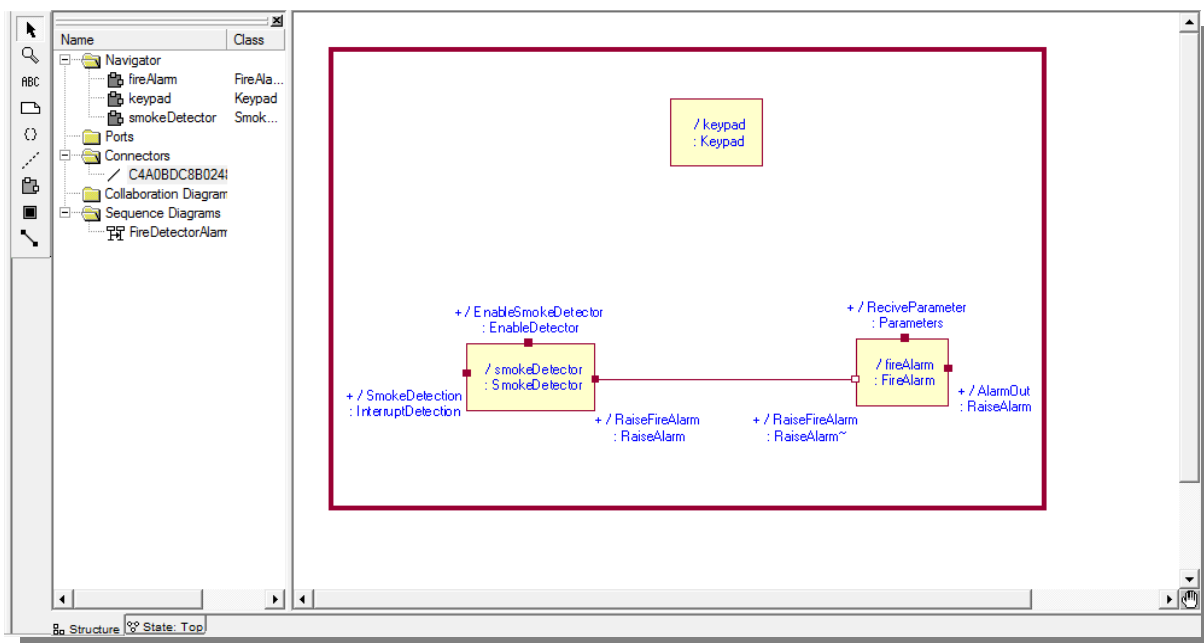


6. Postaviti da je tipa **EndPort**.

Povezivanje portova

1. Otvoriti **Structure Diagram** kapsule **Alarm**.

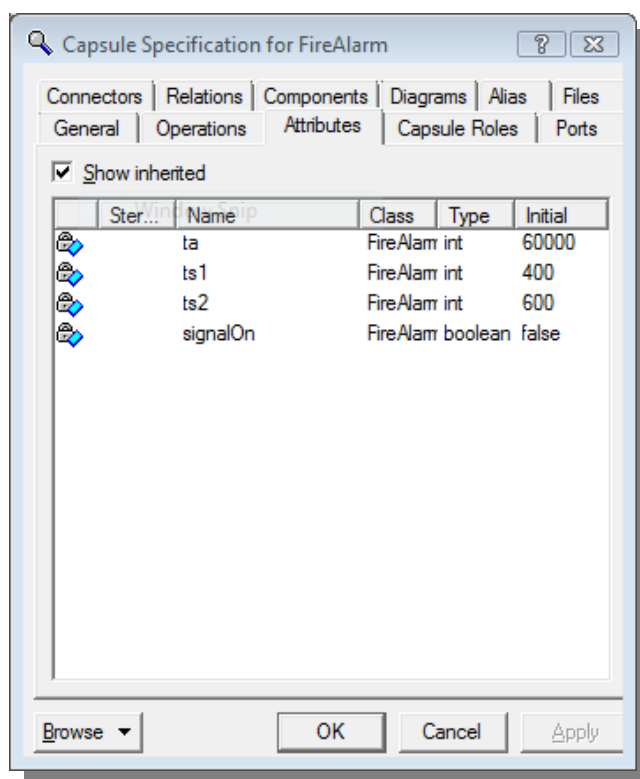
2. Pomoću alatke **Connector**  povezati portove **RaiseFireAlarm** kapsula **SmokeDetector** i **FireAlarm**



Dodavanje atributa

1. Desni klik na kapsulu **FireAlarm** (u Model brazeru) i odabrati **Open Specification**.
2. Odabrati **Attributes** tab.
3. Uneti sledeće attribute:

Ime	Tip	Inicijalna vrednost
ta	int	60000
ts1	int	400
ts2	int	600
signalOn	boolean	false

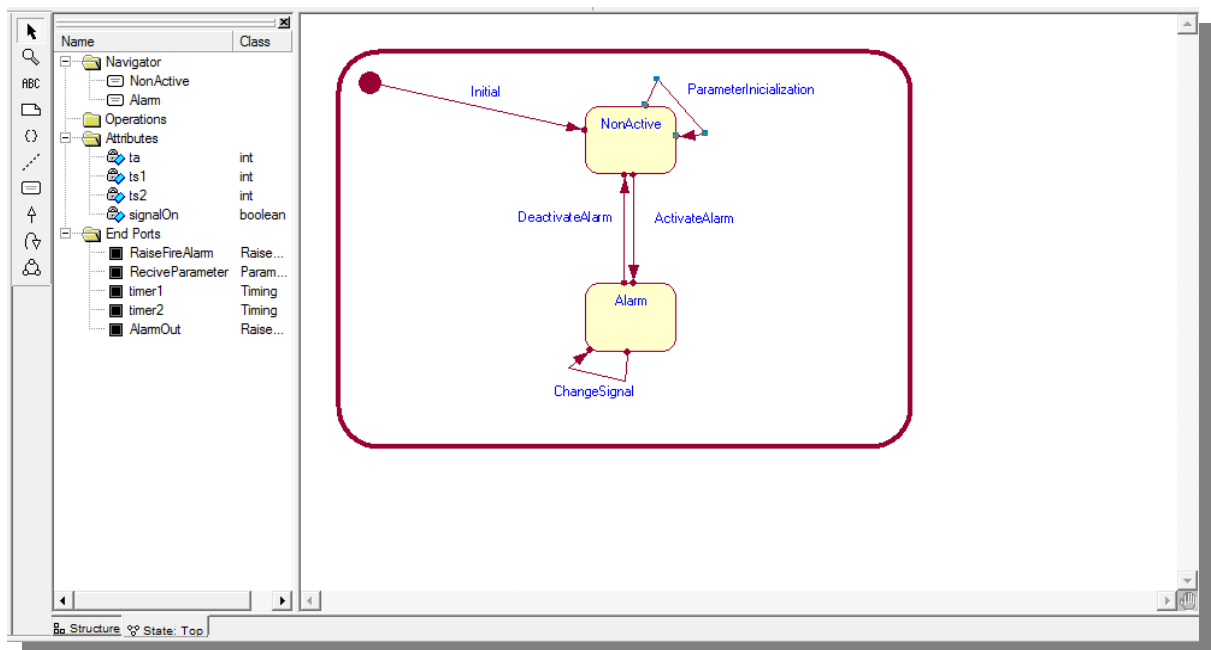


Kreiranje stanja

1. U okviru kapsule **FireAlarm** definisati stanja **Alarm** i **NonActive**.
2. Napraviti međusobne tranzicije.
3. Imenovati ih sa **ActivateAlarm** i **DeactivateAlarm**.
4. Napraviti tranziju u inicijalno stanje **NonActive**, imenovati je **Initial**.
5. Napraviti takođe **Transition to Self** za stanje **NonActive** i imenovati je **ParameterInicIALIZation**.
6. Napraviti **Transition to Self** za stanje **Alarm** i imenovati je **ChangeSignal**.
7. Implementirati tranzicije na sledeći način:

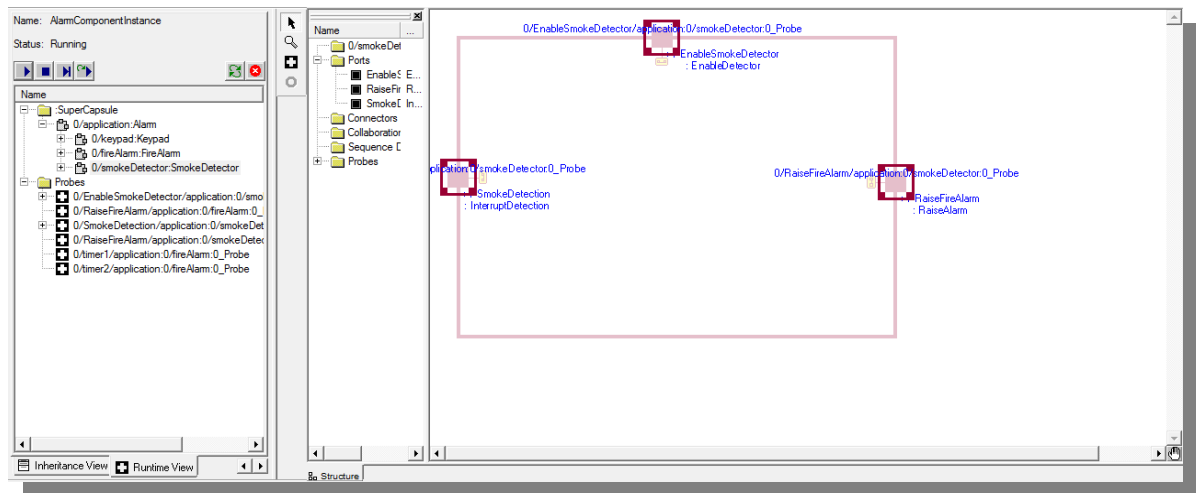
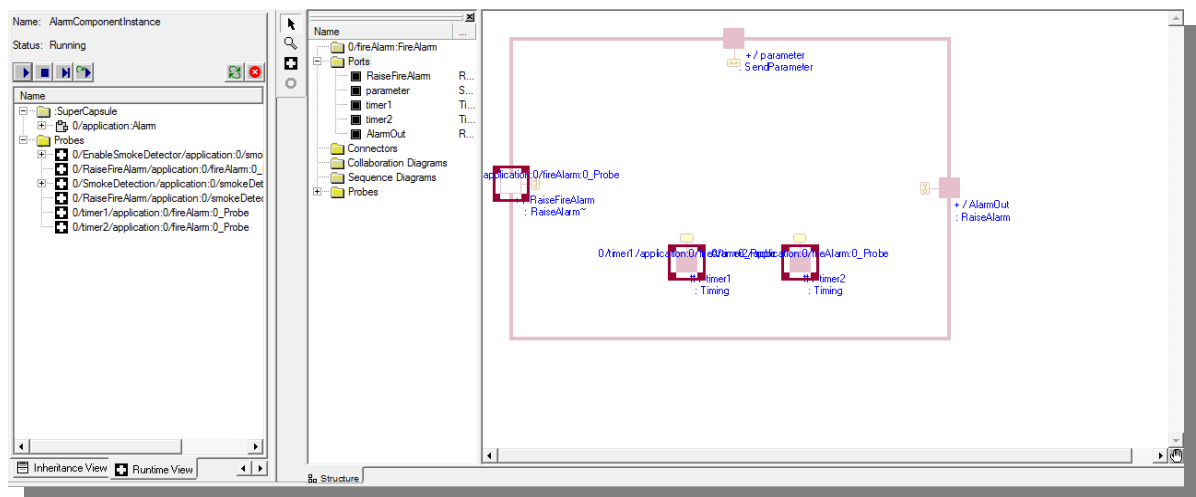
Tranzicija	Port	Signal	Akcija
Initial			ta=60000; ts1=600; ts2=400; signalOn = false;
ActivateAlarm	RaiseFireAlarm	AlarmSignal	timer1.informIn(ts1); timer2.informIn(ta); AlarmOut.AlarmSignal().send(); signalOn = true;
DeactivateAlarm	timer2	timeout	if(signalOn) { // stopAlarm; } AlarmOut.AlarmSignal().send();

			<pre> signalOn = false; } </pre>
ParameterInicijalizacija	ReceiveParameter	Parameter	<pre> ta=((TimeParameters)rtGetMsgData()).Ta; ts1=((TimeParameters)rtGetMsgData()).Ts1; ts2=((TimeParameters)rtGetMsgData()).Ts2; </pre>
ChangeSignal	timer1	timeout	<pre> signalOn = !signalOn; if(signalOn) { // startAlarm; AlarmOut.AlarmSignal().send(); timer1.informIn(ts1); } else { //stopAlarm AlarmOut.AlarmSignal().send(); timer1.informIn(ts2); } </pre>

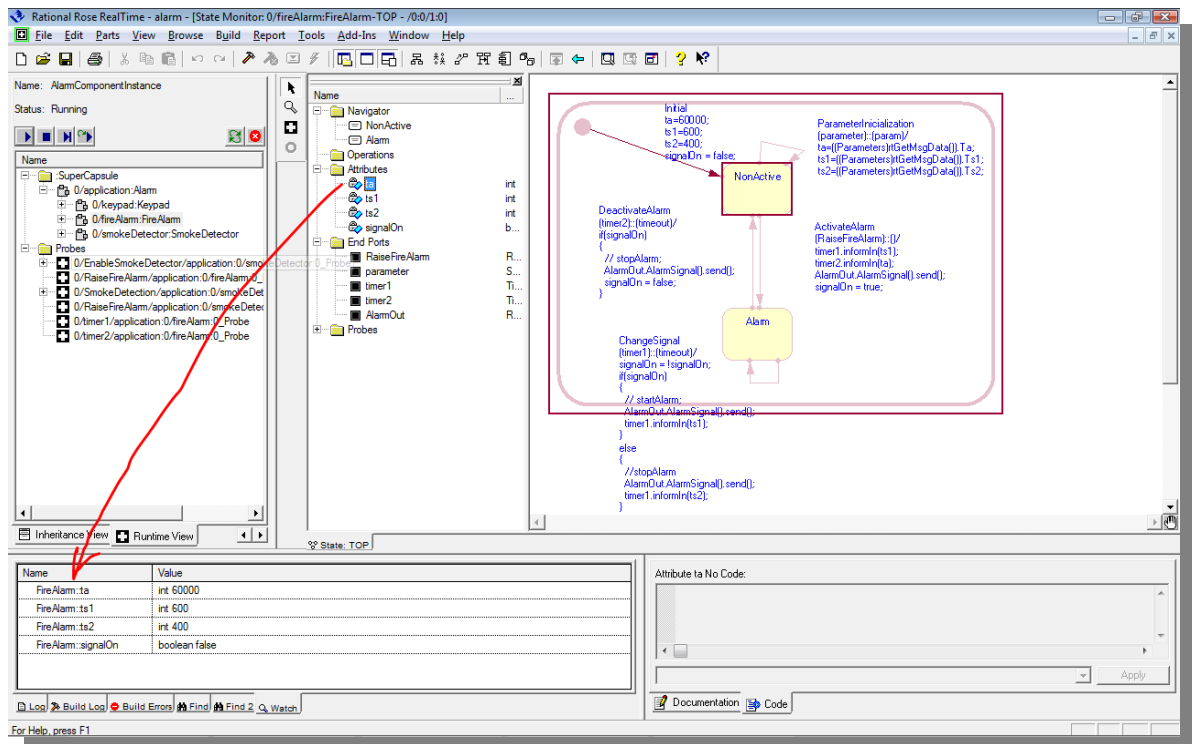


Testiranje

1. Na prethodno naveden način kompajlirati i pokrenuti aplikaciju za kapsulu **Alarm** (*top-level* kapsula koja sadrži sve ostale kapsule).
2. Postaviti *probe* na portove **RaiseFireAlarm**, **timer1** i **timer2** kapsule **FireAlarm**, kao i na sve portove kapsule **SmokeDetector**.



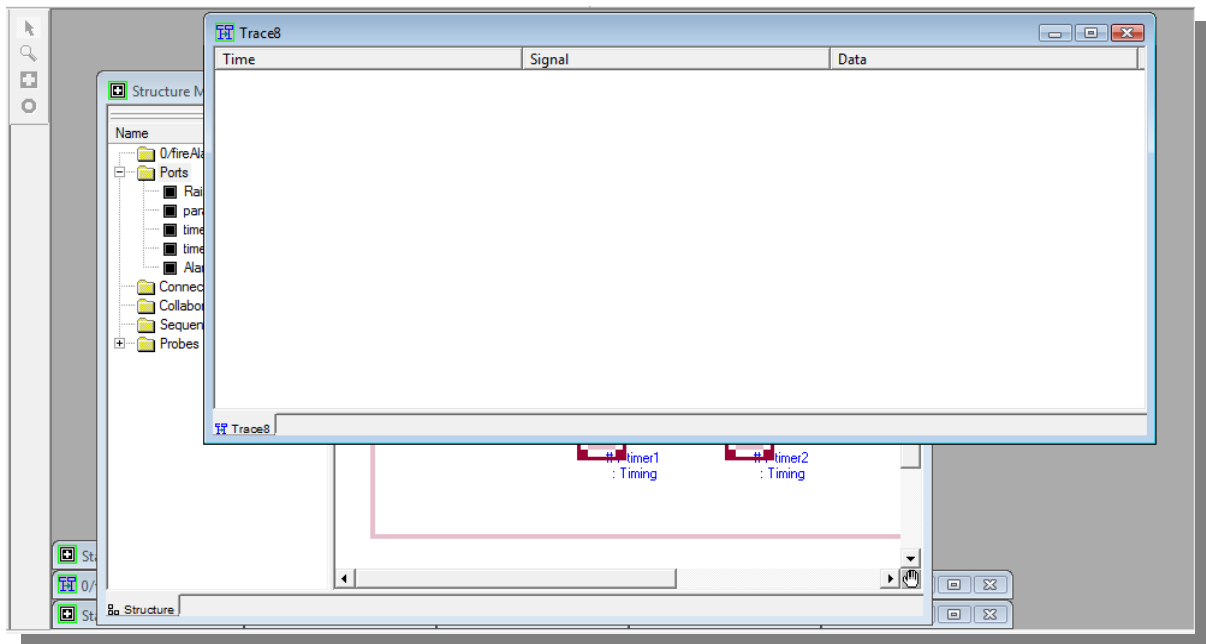
3. Desni klik na instancu kapsule **FireAlarm** i odabrati **Open State Monitor**.
4. Postaviti pogled na sve atribute kapsule **FireAlarm**, prevlačeći ih u prozor **Watch**.



5. Na prethodno pokazani način kreirati **Inject** za port **SmokeDetection** kapsule **SmokeDetector**. Takođe uraditi isto i za port **EnableSmokeDetector**.
6. Zadavati ove pobude i posmatrati promenu atributa i stanja u kapsuli **FireAlarm**.

Kreiranje traga

- Drugi način testiranja izvršavanja modela predstavlja kreiranje i praćenja događaja unutar traga (*trace*):
 1. Desni klik na sondu kreiranu nad portom **RaiseFireAlarm** u strukturnom dijagramu kapsule **FireAlarm** i odabrati **Open Trace**.



1. Ponoviti za portove **timer1** i **timer2**.
2. Ponovo zadati pobudu za port **SmokeDetection** kapsule **SmokeDetector** i posmatrati promene za zadate tragove.

Napomena: Za port **AlarmOut** nije moguće pratiti trag s obzirom na to da nije povezan ni na jedan drugi port. Testiranje se može izvršiti npr. dodavanjem naredbe `System.out.println()` u akciju tranzicije koja šalje poruku preko ovog porta.

Samostalan rad

- Prepraviti ponašanje kapsule **FireAlarm** pomoću metode **Timer.informAt()** (pogledati dokumentaciju) umesto **Timer.informIn()** tako da ovaj proces ne pati od kumulativnog plivanja (engl. *cumulative drift*).
- Impelmentirati i testirati kapsulu **Keypad**, a zatim testirati ceo sistem.

Korisni linkovi

- Rational RoseRT Help:
 - Concept Tutorials
 - QuickStart Tutorial
 - Card Game Tutorial (C++)

Za integrisanje modela sa eksternim projektom (npr. simulacionim ili GUI kodom) pogledati tutorijal [Java Reference>Getting Started with Rational Rose RealTime Java>Integrating external classes](#)

Literatura

1. Burns, A., Wellings, A., "Real-Time Systems and Programming Languages," 3rd ed., *Addison-Wesley*, 2001
2. Selic, B., Gullekson, G., Ward, P.T., "Real-Time Object-Oriented Modeling," *Wiley*, 1994
3. Dibble, D., "Real-Time Java Platform Programming", 2nd ed., *BookSurge Publishing*, 2008
4. Brown, M., "Modeling and developing embedded Java applications with Rational Rose RealTime," *developerWorks*®, 2004
<http://www.ibm.com/developerworks/rational/tutorials/rroserealttime/index.html>
5. "RATIONAL ROSE® REALTIME – Tutorials," *Rational Software Corporation*, Part Number: 800-025115-000, 2002
6. Silberschatz, A., Gagne, G., Galvin, P. B., "Operating System Concepts", 7th ed., *Wiley*, 2005