Sistemi v realnem času

Zapiski predavanj

Saša Divjak, Ljubljana 2000

Kazalo

Kazalo....1
Kaj je to: realni čas....2
Uvod....2
Odzivanje na realno okolje....2
Programska oprema za delo v realnem času....2
Procesi in realni čas....3
Osnovni pojmi....3
Kritične sekcije....4
Medsebojno izobčenje (mutual exclusion)....4
Aparaturno zagotavljanje medsebojnega izobčenja....5
Smrtni in živi objem 6
Semaforji....8
Zgled varovanja kritične sekcije s semaforji:....8
Sinhronizacija procesov s semaforji....9
Semaforji in problem proizvajalec - porabnik....10
Števni semaforji....11
Monitorji 11
Preprosto dodeljevanje virov z monitorji:....12
Monitorji in problem krožnega medpomnilnika....12
Zmenki in problem proizvajalec – porabnik....15
Kanali....15
Drugi mehanizmi 16
Razvrščanje procesov 17
Algoritem Round Robin....17
Večnivojski algoritem razvrščanja....18
Programski jeziki za delo v realnem času....20
Zgled jezika za konkurenčno programiranje: Modula-2....20
Značilnosti OS za delo v realnem času....27
Podpora operacijskega sistema za delo v realnem času....28
Operacijski sistem CHIMERA....28
Načrtovanje sistemov v realnem času....30
Proces definiranja zahtev....32
Testiranje sistemov v realnem času....40
Zanesljivost sistemov v realnem času....41
Vrste izpadov 41
Tipi diagnostik....41
Postopki za obravnavo izpadov 42
Odprava okvar 42
Postopki v primeru preobremenitev 42
Postopki za obravnavo trajnih izpadov 43

Kaj je to: realni čas

Uvod

Sistem v realnem času je tisti sistem, ki prejema iz svojega okolja vhodne podatke in signale in generira v to okolje izhodne signale in podatke sproti (čim hitreje) in na predvidljiv način. Posebno je pomen sistemov v realnem času zaznamoval pojav mikroprocesorjev. Danes najdemo take sisteme že vgrajene (embedded systems) v različna vozila, instrumente in druge naprave.

Interakcija med procesorjem in zunanjim svetom oziroma njegova reakcija na spremembe mora biti predvsem pravočasna. V primeru nekaterih fizikalnih in tehnoloških pojavov pomeni tak odzivni čas nekaj tisočink sekunde ali še manj. V primeru interakcije človeka s takim sistemom je tak čas morda nekoliko večji, vendar še znosen in v predvidljivem času. Zato nekateri klasični operacijski sistemi, ki znajo neko programsko aplikacijo tudi začasno prekiniti na račun drugih, niso primerni za delo v realnem času. Časovni zamik odziva sistema (temu pravimo latentnost) namreč ne sme biti tako dolg, da odgovor računalnika nima več nobenega smisla.
Dodatna značilnost sistemov v realnem času je še, da pogosto delujejo v omejenem okolju, ki ga lahko obvladuje relativno preprost mikroprocesorski računalnik.

Odzivanje na realno okolje

Signali iz zunanjega sveta (poznamo jih kot prekinitve), opozorijo procesor, da nekdo potrebuje pozornost. Programska oprema računalnika določa, kakšna bo reakcija na tak prekinitveni signal. Normalno sprejme ali pošlje kakšen podatek in nato skuša nadaljevati računanje čim hitreje, vsekakor pa predvidljivem času.

Sistemn za delo v realnem času lahko periodično preverja oziroma bere vhodne signale in ugotavlja njihove spremembe (polling). Tak način dela terja od programerja, da predvidi, kdaj lahko do sprememb pride, če naj sistem nanje pravočasno reagira.

Ko nek vhodni signal preberemo, običajno z njim nekaj naredimo. Hitrost odziva
določimo s prioriteto programskih modulov. Dogodke z višjo prioriteto moramo
obravnavati pred dogodki z nižjo, manj kritično prioriteto. Pri tem moramo upoštevati
kasnitve celotnega sistema in njegovo splošno propustnost.

Programska oprema za delo v realnem času

Da dosežemo determinističen odziv in kratke zakasnitve, moramo pri programiranju pogosto uporabljati trike, ki so pri razvoju navadnih programov nepotrebni. Tako programiranje je bolj težko. K sreči imamo orodja, ki ga podpirajo. Tako lahko uporabimo operacijske sisteme za delo v realnem času (RTOS). Tak sistem

zagotavlja upravljanje s pomnilnikom, z vhodom oziroma izhodom računalnika in celo z omrežjem. Med drugimi uporabnimi orodji zasledimo še različne programske knjižnjice.

Programiranje sistemov za delo v realnem času terja poznavanje aparaturne opreme, saj lahko tako uspešno integriramo nadzor naprav realnega sveta z danim procesorjem. Zato imajo mnogi programerji takih sistemov tudi dobro predznanje s področja elektronike.

Procesi in realni čas

V naslednjih poglavjih se bomo usmerili predvsem v področje paralelnih procesov in v probleme njihove medsebojne konkurence in sodelovanja. To problematiko zasledimo pri avtomatizacijah različnih bolj ali manj kompleksnih industrijskih procesov, v robotiki itd. Srečamo pa se z njo v čisto računalniški stroki, pri načrtovanju večuporabniških in predvsem večprocesnih ter večprocesorskih operacijskih sistemov. Tu nam proces predstavlja pravzaprav izvajanje nekega programa skupaj z okoljem, ki ga ta program za svoje življenje potrebuje. Posamezni **programski procesi** konkurirajo za računalniški čas in za aparaturne komponente računalniškega sistema. Zanimive pojave spoznamo že pri tako preprostih aktivnostih, ki se skrivajo za ukazom za branje ali izpisovanje.

Za računalnikarje pa so posebno zanimivi porazdeljeni računalniški sistemi. V tej domeni so mehko ali trdno sklopljeni računalniki in računalniške mreže. Komponente takih sistemov so lahko preprosti transputerji, delovne postaje ali večji računalniki. V takem okolju pa pride do izraza vsa problematika porazdeljenega računanja, porazdeljenih podatkovnih baz in operacijskih sistemov, ki podpirajo take sisteme.

Osnovni pojmi

Uporabimo definicijo, ki pravi,da je programski proces pravzaprav odvijanje nekega programa skupaj z okoljem, ki je za to odvijanje potrebno.

V to okolje sodijo predvsem potrebne programske strukture, kot je njegova koda, podatkovne strukture in sklad. V to okolje sodijo tudi vsi ostala potrebna sredstva (resources). Tako potrebujemo za omenjene programske strukture zadostno količino pomnilnika, dostop do perifernih naprav itd.

Procesi so paralelni, če obstojajo istočasno. Paralelni procesi so lahko drug od drugega neodvisni, lahko pa so **asinhroni**, kar pomeni, da zahtevajo občasno sinhronizacijo in sodelovanje. Interakcija asinhronih programov je včasih zelo kompleksna.

Pri obravnavi paralelnih procesov zasledimo vrsto zanimivih pojavov in pred programerjem takih (programskih) procesov se pojavijo dodatni problemi, na kakršne ni naletel niti pri zelo kompliciranih klasičnih programskih sistemih.

Tako si lahko posamezni programi med seboj konkurirajo za sredstva (resources), ki so v njihovem "delovnem" okolju na voljo. Pravimo, da so taki procesi **konkurenčni**. Preprost zgled za tako konkurenčnost so na primer sočasne zahteve po dostopu do diska v mnogouporabniškem računalniškem sistemu.

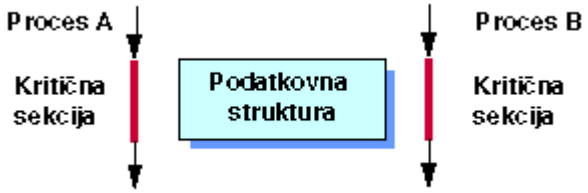
Pogosto se pojavi potreba po medsebojni časovni koordinaciji posameznih paralelnih procesov. In ne nazadnje, Posamezni procesi se morajo pogosto med seboj pogovarjati. Govorimo o medprocesni **koordinaciji** in **komunikaciji**.

Prav zaradi konkurenčnosti procesov in tudi potreb po nihovi medsebojni sinhronizaciji moramo uvesti pojem **stanja procesa**. Zaenkrat omenimo le, da je proces, ki mora čakati na nek drug proces, na sprostitev nekega sredstva (resource) ali na nek dogodek, v **blokiranem** stanju. Poenostavljeno naj velja, da je proces, ki ni blokiran, **tekoč**. V resnici so pri gradnji posameznih operacijskih sistemov uvedli še več procesnih stanj.

V nadaljevanju bomo spoznavali različne pristope, ki pomagajo reševati problematiko paralelnih procesov. Pri tem nam bodo kot rdeča nit služili nekateri tipični problemi.

Kritične sekcije

Ko ima proces dostop do skupnih podatkovnih struktur, je v takoimenovani kritični sekciji .



Zagotoviti moramo, da je v času, ko je nek proces v kritični sekciji, drugim procesom dostop v to sekcijo prepovedan. Ostali procesi lahko medtem paralelno potekajo, v primeru zahteve po vstopu v zasedeno kritično sekcijo pa čakajo. Ob sprostitvi kritične sekcije lahko vanjo vstopi le po en proces.
Medsebojno izobčenje konkurenčnih programskih procesov lahko zagotovimo s posebnimi konstrukti- primitivi.

Medsebojno izobčenje (mutual exclusion)

Vzemimo sistem z več terminali. Sistem naj "šteje število vrstic, vtipkanih v enem dnevu". V ta namen imamo skupno sistemsko spremenljivko *lineCount*. Delo posameznih uporabnikov za terminali nadzorujejo ločeni procesi. Kadarkoli nek uporabnik na svojem terminalu vtipka ENTER, se izvedejo v sklopu njegovega programskega procesa stavki naslednje vsebine:

```
load lineCount
```

```
add 1
store lineCount,
```

Vzemimo primer, da *lineCount* `e vsebuje vrednost 1234 in da proces *A* pravkar izvaja ukaz *add*, ki vrednost v akumulatorskem registru poveča na 1235. Naj v tem trenutku proces *B* (zaradi enakega štetja) prav tedaj izvede operacijo *load*. Podatke 1235 v akumulatorju se zato spet pokvari na 1234 in, če bi v tem hipu proces *A* izvedel še zadnjo operacijo *(store)*, bi očitno shranil v spremenljivko *lineCount* zgrešeno vrednost.

Problem lahko rešimo, če je **dostop** posameznih programskih procesov do takih skupnih spremenljivk, kot je v našem primeru *lineCount*, **ekskluziven**. Podobne probleme zasledimo tudi na primer pri večuporabniškem dostopu do podatkov v podatkovnih bazah, pri (sistemskem) dostopu do skupnih podatkovnih struktur (operacijskega sistema) v večprocesnih operacijskih sistemih itd.

Aparaturno zagotavljanje medsebojnega izobčenja

Ključ do uspeha je v enotni strojni instrukciji, ki hkrati bere vrednost neke spremenljivke in ji hkrati priredi določeno vrednost. To so instrukcije tipa *"Test-and-set"*.

Zgled take instrukcije bi bil naslednji:

testandset(a,b)

Instrukcija prebere vrednost logične spremenljivke *b*, jo kopira v *a* in istočasno nastavi *b* na *TRUE*.
Zgled dveh procesov, ki se ščitita s tako instrukcijo:

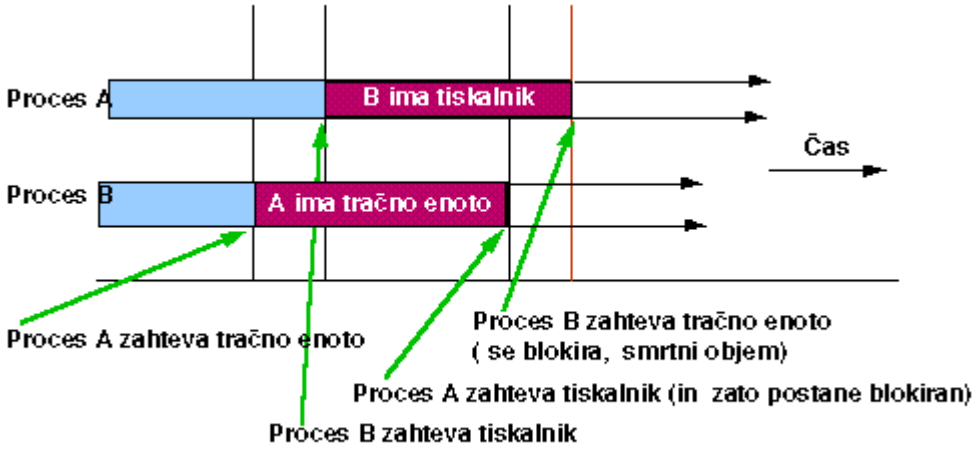
```
program example;
var active:boolean;
procedure process_A;
var waiting:boolean;
begin
  repeat
    waiting:=TRUE;
    while waiting do testAndSet(waiting,active);
    {...kriticna sekcija...}
   active:=FALSE;
until FALSE;

end;

procedure process_B;

var waiting:boolean;
begin
  repeat
    waiting:=TRUE;
    while waiting do testAndSet(waiting,active);
```

```
    {...kriticna sekcija...}
    active:=FALSE;
  until FALSE;
end;

begin
  active:=FALSE;
  parbegin
    process_A;
    process_B;
  parend;

end.
```

Ščitenje kritičnih sekcij s primernimi strojnimi instrukcijami pride v poštev predvsem pri večprocesorskih sistemih. Pri enoprocesorskih sistemih pa si lahko pomagamo s programskimi mehanizmi ali protokoli. Nedeljivost takega mehanizma zagotavlja začasna preprečitev prekinitev (interrupts) procesa, ko uporablja tak mehanizem. Take mehanizme si bomo ogledali v poglavju, ki je posvečeno medprocesni sinhronizaciji.

Smrtni in živi objem

Obravnavali bomo različne tehnike medsebojne sinhronizacije paralelnih programskih procesov. Ta se doseže tako, da proces, ki naj bo sinhroniziran z nekim drugim, v danem trenutku počaka na nek **dogodek** (event). Prav možno je, da do tega dogodka iz različnih razlogov ne more priti. Pravimo, da se je en ali več paralelnih procesov znašlo v **smrtnem objemu** (deadlock). Oglejmo si nekaj možnih vzrokov za tak pojav ter ugotovimo, kako ga lahko preprečujemo.



Do smrtnega objema lahko pride, ko na primer dva procesa čakata drug na (dogodek) drugega. Ker pa sta oba s tem blokirana, do odrešilnega dogodka ne v enem ne v drugem procesu ne pride. Preprost zgled je lahko naslednji. Imamo dva programska procesa, ki za svoje delo potrebujeta dve periferiji. Vsak si že lasti eno in čaka na drugo. Problem ponazoruje slika.

Sorodni problem smrtnemu objemu je problem **nedoločenega zapostavljanja**. Včasih mu tudi rečemo problem **živega objema** (livelock). Nek proces čaka na neko sredstvo, vendar je (morda zaradi nižje prednosti) stalno zapostavljen in nikdar ne pride na vrsto. Tipičen primer živega objema lahko zasledimo pri slabo zasnovanem algoritmu razvrščanja procesov (scheduling). Nek proces z nizko prioriteto morda nikdar ne pride na vrsto, ker imajo prednost procesi z višjo prednostjo. Ta problem lahko rešujemo na primer tako, da s staranjem prednost takemu procesu zvišujemo.
Navedene probleme blokiranja nekega procesa (nedoločeno zapostavljanje, smrtni objem) zasledimo na primer pri operacijskih računalniških sistemih. Operacijski sistem je predvsem "upravnik sredstev" (resource manager) računalniškega sistema. Za ta sredstva pa istočasno lahko konkurira več paralelnih programskih procesov.

Za nastop smrtnega objema so potrebni naslednji štirije pogoji:

- Procesi zahtevajo exskluzivno "lastništvo" sredstva,
- Procesi imajo že dodeljena sredstva in hkrati zahtevajo in čakajo na dodatna,
- Procesom ne moremo odvzeti sredstev, ki si jih lastijo, dokler jih sami ne vrnejo,
- Obstoja krožna veriga procesov, od katerih si vsak lasti kakšno sredstvo, ki ga zahteva tudi naslednji proces v tej verigi.

Problem smrtnega objema lahko rešujemo na več načinov. Tako lahko sredstva oštevilčimo in dovolimo, da posamezni procesi lahko zahtevajo le-ta zgolj po naraščajočih številkah. To seveda verjetno ne ustreza vsem procesom. Druga možnost je, da morajo v kritičnem primeru posamezni procesi vračati sredstva. Tako strategijo plačamo s tem, da je bilo dotedanje delo takih procesov morda zaman in ga bodo morali ponoviti.
Obe navedeni metodi preprečujeta smrtni objem. Blažje strategije se temu pojavu le izogibajo. Za primere, ko več procesov kandidira za resurse istega tipa (na primer bloke v medpomnilniku) lahko uporabimo "**bančni algoritem**", ki ga je predlagal Dijkstra. Ime je dobil po strategiji, ki jo uporabljajo tudi v bankah. Poslovanje teh sloni na predpostavki, da vsi varčevalci ne dvignejo istočasno vseh svojih vlog in ima zato banka lahko v danem trenutku manj gotovine, kot je vsota vseh vlog.

Po tej strategiji poznamo **varno** in **nevarno stanje** sistema.
Varno stanje imamo tedaj, če lahko (operacijski) sistem vsem tekočim uporabnikom zagotovi zaključek posla v nekem končnem času.

Uporabniki (procesi) morajo sistem vnaprej informirati, koliko sredstev (istega tipa) bodo maksimalno potrebovali v istem času. Za primer vzemimo procese, ki operirajo s tračnimi enotami.

Primer varnega stanja:

Proces:	**Trenutno v lasti:**	**Maksimalna zahteva:**

Proces(1)	1	4
Proces(2)	5	8
Proces(3)	4	6

Proste enote: 2

Primer nevarnega stanja:

Proces:	Trenutno v lasti:	Maksimalna zahteva:
Proces(1)	1	3
Proces(2)	8	10
Proces(3)	2	5

Proste enote: 1

Nevarno stanje še ne pomeni, da bo do smrtnega objema v resnici prišlo. Ker pa taka nevarnost, ob neugodnem razpletu dogodkov, grozi, se takemu stanju raje izognemo. Sistem zato dodeljuje sredstva le, če pri tem prehaja v varna stanja.

Semaforji

Dijkstra je uvedel koncept **semaforjev** in takoimenovanih operacij ***P*** in ***V***. Oznake teh operacij izhajajo iz holandskih besedic *vhod* in *izhod*. Semafor je ščitena spremenljivka (pravzaprav podatkovna struktura), ki jo lahko inicializiramo ter dosežemo le z operacijami *P* in *V*. **Binarni semaforji** lahko zavzamejo le vrednosti 0 in 1, **števni semaforji** pa poljubne nenegativne vrednosti.
Operacijo *P* na semaforju *S* zapišemo kot ***P(S)*** in deluje tako:

```
if S0
   then S:= S-1
   else (* pocakaj na S v ustrezni vrsti *);
```

Operacijo *V* na semaforju *S* pa zapišemo kot ***V(S)*** in deluje tako

```
if (* eden ali vec procesov caka v vrsti  na S *)
   then (* naj vstopi en proces *)
else S:=S+1;
```

Procesi naj čakajo na zaključek *P(S)* v čakalni vrsti tipa *FIFO* (first in - first out). Zato je vsak semafor podatkovna struktura, ki vsebuje en integer in eno vrsto.
V literaturi označujejo operaciji *P(S)* in *V(S)* tudi z imeni ***wait(semaphore)*** in ***signal(semaphore)***, ki imata v bistvu enak pomen.
Podobno kot *"test-and-set"* morata biti tudi operaciji *P* in *V* **nedeljivi**. Same konkretne implementacije semaforjev so lahko različne. Če imamo na voljo instrukcije tipa *test-and-set*, jih uporabimo, sicer pa ne dovolimo prekinitev (disable interrupts) med samim izvajanjem operacij *wait* in *signal.*

Zgled varovanja kritične sekcije s semaforji:

```
program example;
var active:semaphore;
```

```
procedure process_A;

begin
  repeat
     wait(active);
     š...kriticna sekcija...}
     signal(active);
  until FALSE;
end;

procedure process_B;
begin
 repeat
   wait(active);
   š...kriticna sekcija...}
   signal(active);
 until FALSE;
end;

begin
 initSemaphore(active,1);
 parbegin
   process_A;
   process_B;
 parend;
end.
```

Sinhronizacija procesov s semaforji

Naslednji zgled uporabe semaforjev je sinhronizacija procesov. Vzemimo za primer vhodno-izhodne računalniške operacije. Proces, ki sproži vhodno-izhodno operacijo, se običajno sam blokira , da bi tako počakal na njen zaključek. Blokirani proces kasneje obudi nek dogodek v nekem drugem procesu. Pri taki interakciji imamo torej protokol *blokiranje- bujenje*. Ta je uporaben, kadarkoli želimo nek proces sinhronizirati z danim dogodkom v drugem procesu. Zgled je podoben prejšnjemu:

```
program sinhronizacija;
var dogodek:semaphore;
procedure process_A;
begin
 (*..predhodne operacije...*)
 wait(dogodek); (* proces_A caka na dogodek v procesu B *)
 (*...nadaljevanje procesa...*)
end;

procedure process_B;
begin
 (*...predhodne operacije ..*)
 signal(dogodek);
 (*..nadaljevanje procesa..*)
end;
```

```
begin
  initSemaphore(dogodek,0);
  parbegin
    process_A;
    process_B;
  parend;
end.
```

Semaforji in problem proizvajalec - porabnik

Oglejmo si še uporabnost semaforjev za reševanje problema *proizvajalec - porabnik*. Proces *A* tvori podatke, ki jih proces *B* uporablja. Hitrost izvajanja posameznih procesov *A* in *B* se v splošnem ne ujema. Zagotoviti moramo, da ne prihaja niti do izgube podatkov niti do večkratnega branja in izpisa istega podatka. Oba procesa moramo torej sinhronizirati, kot to kaže naslednji zgled:

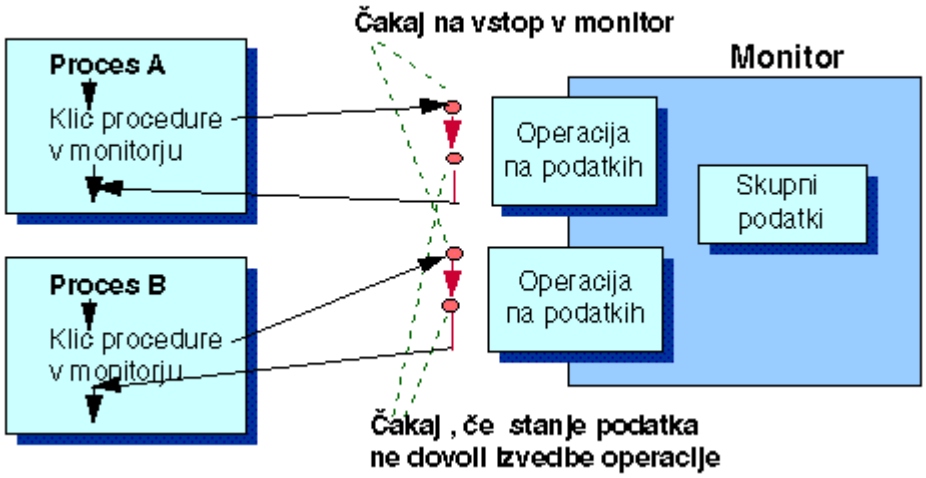
```
program proizvajalec_porabnik;
var zasedeno:semaphore;
    vpisano:semaphore;
    buffer: integer;

procedure proizvajalec;
var rezultat: integer;
begin
  repeat
    racunaj(rezultat);
    wait(zasedeno);  (* vstop v kriticno sekcijo *)
    buffer:=rezultat;
    signal(vpisano);  (* sinhronizacija *)
    signal(zasedeno); (* sprostitev kriticne sekcije *)
  until FALSE;
end;

procedure porabnik;
var podatek:integer;
begin
  repeat
     wait(vpisano); (* blokiran proces, ce podatka se ni *)
     wait(zasedeno);  (* vstop v kriticno sekcijo *)
    podatek:=buffer;
    signal(zasedeno);  (* sprostitev kriticne sekcije *)
    izpis(podatek);
  until FALSE;
end;

begin
  initSemaphore(zasedeno,1);
  initSemaphore(vpisano,0);
  parbegin
    proizvajalec;
    porabnik;
  parend;
```

end.

Števni semaforji

Števne semaforje s pridom uporabimo, če želimo alokacijo enega sredstva (resource) iz zaloge enakih sredstev. Semafor inicializiramo na število sredstev v zalogi. vsaka operacija *P* (oziroma *wait*) zmanjša semafor za 1 in tako pokaže, da se je zaloga zmanjšala. Vsaka operacija *V* (oziroma *signal*) poveča semafor za 1 in tako nakaže povečanje zaloge za 1. Če uporabimo operacijo *wait* potem, ko je semafor dosegel 0, mora tak proces počakati, da se vsaj eno sredstvo sprosti z ustrezno operacijo *signal*.
Semaforji so zelo primitivni in z njimi težko izrazimo zelo kompleksne konkurenčne probleme. Tako si težko zapomnimo, kateri ščiteni strukturi smo priredili dani semafor. Mimogrede pozabimo uporabiti kakšno instrukcijo wait in prezgodaj vstopimo v kritično sekcijo. Procesi lahko tudi ostanejo neomejeno blokirani, ker (morda zaman) čakajo na odrešilno operacijo *signal*, ki bi jo moral uporabiti nek drug proces.

Monitorji



Tveganje zaradi morebitne napačne uporabe semaforjev je veliko. Uvedli so zato **monitorje**. Ti so konkurenčni konstrukti, ki vsebujejo tako podatke kot procedure (tudi več procedur), ki omogočajo alokacijo skupnega sredstva (na primer nekega medpomnilnika, "bufferja"). Alokacijo doseže proces s klicem ustreznega **monitorskega vhoda** (monitor entry). Taki klici so podobni klicem funkcij ali procedur. Istočasno lahko vstopa v monitor le en proces, ostali morajo počakati. Čakanje procesov poteka izven monitorja (v ustrezni čakalni vrsti).
Podatki, ki so znotraj monitorja, so skriti vsem ostalim procesom. Sama implementacija monitorja mora preprečevati, da je v njem aktiven več kot en proces!! Zato je vsakemu monitorju prirejen en semafor. Vklučitev ustreznih operacij *wait* in *signal* je stvar prevajalnika, ne programerja, ki monitorje uporablja. Tako imamo zagotovljeno *medsebojno izobčenje* (mutual exclusion). To pa ni dovolj za

sinhronizacijo pri konkurenčnih procesih. Tako lahko morda želimo uporabiti nek podatek (na primer "buffer"), ki še ni definiran. Četudi smo že "v monitorju", moramo počakati. Uvedli so zato dodaten pojem **pogojne spremenljivke** (condition variable). Operaciji na taki, pogojni spremenljivki sta prav tako operaciji **wait** in **signal**, podobno kot pri semaforjih.

Če nek proces želi alociranje nekega sredstva, ki je že zasedeno, kliče monitor proceduro *wait.*

Ko proces, ki zaseda neko sredstvo, želi le-tega vrniti, spet pokliče monitor za ta poseg. Monitor s pomočjo procedure *signal* nato omogoči dostop do sredstva naslednjemu procesu iz čakalne vrste.

Preprosto dodeljevanje virov z monitorji:

Imejmo več procesov, ki kandidira za isto sredstvo, hkrati pa lahko to sredstvo uporablja le en proces. Preprost monitor, ki bi omogočal dodeljevanje in sproščanje tega sredstva, bi lahko imel obliko:

```
monitor allocResource;
var zaseden:boolean;
   prost: condition;

procedure getResource;
begin
  if zaseden then wait(prost);
  zaseden:=TRUE;
end;

procedure returnResource;
begin
  zaseden:=FALSE;
  signal(prost);
end;
begin
  zaseden:=FALSE;
end;
```

Zgled bi morali dopolniti še z definicijo dveh ali več procesov (na primer *procesA, procesB, procesC*,..), ki bi v svojem telesu uporabljali klice procedur *getResource* in *returnResource*.

Monitorji in problem krožnega medpomnilnika

Naslednji tipičen zgled, primeren za uporabo monitorjev, je primer krožnega medpomnilnika. Oglejmo si torej tehniko komunikacije preko krožnega medpomnilnika in z uporabo monitorjev. Krožni medpomnilnik (ringBuffer) naj vsebuje *N* elementov namišljenega tipa *"data"*.

```
monitor ringBuffer;
var Buffer: array[0..N-1] of data;
   numUsed, nextFree, nextToUse: 0..N-1;
   nonempty, nonfull: condition;

procedure putData(value:data);
  begin
    if numUsed= N then wait(nonfull);
    Buffer[nextFree]:= value;
    numUsed:= numUsed+1;
    nextFree:= (nextFree + 1) mod N;
    signal(nonempty);
  end;

procedure getData(var value:data);
  begin
    if numUsed=0 then  wait(nonempty);
    value:= Buffer[nextToUse];
    numUsed:=numUsed-1;
    nextToUse:= (nextToUse + 1) mod N;
    signal(nonfull);
  end;

begin
  numUsed:=0; nextFree:=0; nextToUse:=0;
end;
```

Monitorji in problem Bralci - pisalci

Končno si oglejmo uporabnost monitorjev pri problemu *bralci - pisalc*i. Več procesov- bralcev bere podatke, ki jih več procesov- pisalcev zapisuje. Ta problem srečamo na primer pri sistemih za delo s podatkovnimi bazami.
Velja, da

- lahko ima istočasno dostop do skupnega podatka več bralcev,
- je v danem trenutku lahko aktiven kvečjemu le en pisalec,
- ne sme biti istočasno aktiven pisalec in en ali več bralcev.

Poglejmo si listanje programske rešitve problema z monitorji:

```
monitor readersWriters;
var
  readers:integer;
  someonewriting:boolean;
  readingAllowed,writingAllowed:condition;

procedure beginReading;
  begin
    if someoneWriting or queue(writingAllowed)
      then wait(readingAllowed);
```

```
      readers:=readers+1;
      signal(readingAllowed);
  end;
procedure finishedReading;
  begin
    readers:=readers-1;
    if readers=0 then signal (writingAllowed);
  end;

procedure beginWriting;
  begin
    if readers0 or someoneWriting
      then wait(writingAllowed);
    someoneWriting:=TRUE;
  end;

procedure finishedWriting
  begin
    someoneWriting:=FALSE;
    if queue(readingAllowed)
      then signal(readingAllowed)
      else signal(writingAllowed);
  end;

begin
  readers:=0; someoneWriting:=FALSE;
end;
```

Zmenki

Naslednja tehnika, ki naj zagotovi medsebojno izobčenje ter sinhronizacijo, je tehnika **zmenkov**, ki so jo uvedli v programskem jeziku ADA. Srečamo se s pojmoma **strežnik** in **klijent** (kličoči in servisni proces). V razliko od monitorjev je to aktiven mehanizem. Tudi strežnik je namreč aktiven proces, ne le neka pasivna procedura. Servisni proces vsebuje več takoimenovanih **vhodov** (entry), ki jim ustrezajo stavki ***"accept"***. Take vhode pa lahko kliče proces, ki potrebuje zmenek s servisnim programom. Ne pozabimo, da je tudi strežnik proces, ki "živi". Če pride strežnikdo nekega stavka *"accept"*, tu počaka, da bo nek paralelni proces zahteval zmenek pri tem vhodu. Če pa nek paralelni proces kliče *vhod* strežnika, ko ta še ni prišel do stavka *"accept"*, mora prav tako počakati.
Ko pride do zmenka, kličoči proces posreduje strežniku morebitne parametre in počaka, da strežnik izvede stavke znotraj telesa *"accept"*. Strežnik še posreduje kličočemu programu morebitne rezultate in zmenek je končan. Oba procesa se nato odvijata naprej neodvisno.
Zanimivo je, da morajo kličoči programi poznati vstopne točke v strežnik, slednji pa lahko "servisira" klic kateregakoli kličočega procesa (takih procesov pa je lahko tudi več). V tem smislu je tehnika zmenkov **asimetrična**.

Zmenki in problem proizvajalec – porabnik

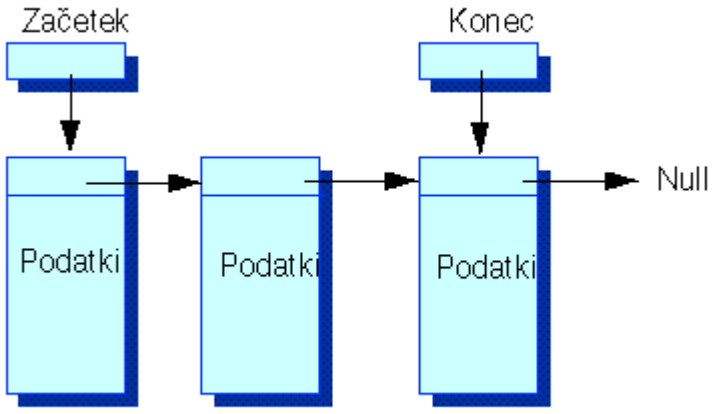
Demonstracijski primer spet obravnava problem *proizvajalec- porabnik*. Komunikacija med procesom- proizvajalcem in procesom- porabnikom tu poteka posredno preko strežnega procesa. Proizvajalec pošilja zaporedje znakov ASCII, ki jih nato porabnik prevzema. To tehniko lahko praktično uporabimo pri programiranju komunikacije računalnika z nekim terminalom ali oddaljenim računalnikom.

```
task simpleBuffer is
  entry write(ch:in character);
  entry read(ch:out character);
end simpleBuffer;
task body simpleBuffer is
  char:character;
begin
   loop
      accept write(ch:in character); do
        char:=ch;
      end write;
      accept read(ch:out character); do
         ch:=char;
      end read;
   end loop;
 task producer;
task body producer is
    c1:character;
    begin
        loop
            produce c1;
           simpleBuffer.write(c1);
       end loop
end producer;
task consumer;
task body consumer is
    c2:character;
begin
   loop
       simpleBuffer.read(c2);
       use c2;
   end loop
end consumer;
```

Kanali

Smisel kanala je, da zagotovi informacijski cevovod med dvema procesoma. Oba procesa potekata asinhrono, zato mora sistem predvidevati tudi pomnjenje prenašanih podatkov (buffering). Prenašana informacija je normalno posredovana v nekem fiksnem podatkovnem formatu. Ta normalno vsebuje identifikacijske podatke (kot je

na primer ime procesa - pošiljatelja). Poleg tega vsebuje še podatke ali pa vsaj kazalec na podatke.



Zelo pogosto so kanali implementirani kot **vrste** tipa **FIFO**. Podatek, ki ga posredujemo v kanal, bo prvi, ki ga bomo iz kanala prevzeli. Primer take vrste prikazuje slika.Vrste s sporočili niso nujno tipa FIFO. Posamezna sporočila namreč lahko imajo tudi različno **prednost**.

Proces- prejemnik jemlje sporočila z začetka take vrste, Proces- pošiljatelj pa vpisuje sporočila na konec vrste. Ker so to običajno dinamične podatkovne strukture, mora normalno najprej zaprositi operacijski sistem, da mu dodeli prostor za novo sporočilo. Glede na to, lahko pride tudi do zasičenja sistema. Take situacije moramo preprečiti, lahko tudi tako, da dobi obdelava pretirano dolgih vrst večjo prednost v računalniškem sistemu.

Tak koncept komunikacije je izvedljiv tudi v porazdeljenem, večprocesorskem okolju.

Drugi mehanizmi

Zalogo običajno predstavljajo razne sistemske tabele, skupni (shared) podatki in skupne datoteke. Tako shranjujemo na primer tiste podatke, ki so pomembni za več procesov. Format podatkov v takih zalogah je odvisen od posamezne aplikacije. Pri uporabi takih mehanizmov moramo zagotoviti konsistentnost podatkov. Preprečiti moramo branje podatkov medtem, ko jih nek proces spreminja. To je tipični problem, znan pod imenom *bralci - pisalci*.
Kot mehanizem za medprocesno komunikacijo služijo tudi **monitorji** in **zmenki**, saj je pri njih predvideno tudi posredovanje (in sprejemanje) parametrov.

Razvrščanje procesov

Pri tej problematiki moramo upoštevati pojem posla. (task). To je v bistvu naloga, ki jo zastavimo računalniku in za katero pričakujemo, da bo končana v primernem času. Pri tem moramo upoštevati razpoložljivost sredstev (resources), ki jih tak posel zahteva.

Po svoji naravi so posli periodični (ponavljajoči se) in aperiodični (prehodni). Parametri, ki jih morda še imamo na voljo, so:

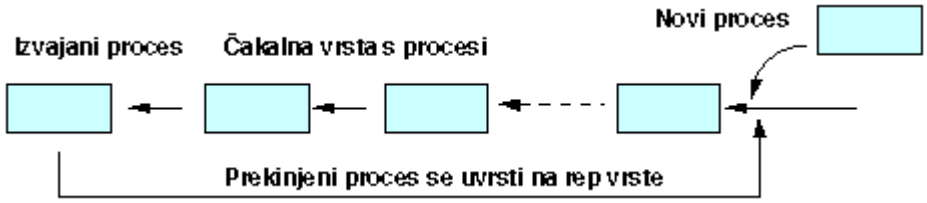
- Čas prihoda posla (arrival time)
- Najzgodnejši čas, ko se posel lahko začne izvajati (ready time),
- Čas računanja v najslabšem primeru (worst case computation time)
- Čas, do katerega se mora posel končati (deadline)

Poznamo statično in dinamično dodeljevanje časa v sistemih za delo v realnem času.

Statične razporede lahko računamo posredno (off line). Tako razporejanje je primerno predvsem za paketno obdelavo poslov. Tako razporejanje terja dobro poznavanje karakteristik poslov in je nefleksibilno. Če se nek posel obnaša drugače od pričakovanj, ga lahko sistem po poteku planiranega časa prekine (vse doslej opravljeno delo bo potrebno ponoviti) ali pa nadaljuje (in za takega uporabnika uvede dodatno kazensko tarifo).

Pri dinamičnem načinu se vrstni red razporejanja procesov računa sproti glede na dogovorjene kriterije optimalnosti in glede na spremembe v stanju sistema.
Po drugi strani ločimo principe razporejanja na centralizirane in decentralizirane. Prvi so primerni za delo na enoprocesorskih sistemih, drugi pa pridejo v poštev pri porazdeljenih računalniških sistemih.

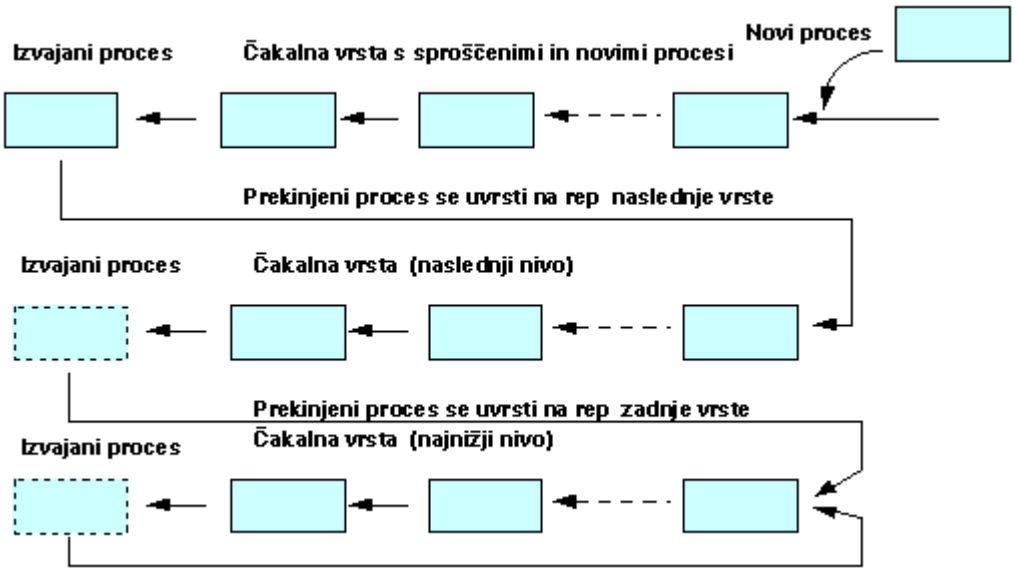
Algoritem Round Robin



Med najbolj znanimi in enostavnimi postopki centraliziranega razporejanja omenimo dodeljevanje po principu **round-robin**. Ta je po ideji podoben primitivnemu algoritmu FCFS (first come- first served) s tem, da po preteku nekega časovnega intervala tekoči proces (ali posel) prekinemo in ga uvrstimo na rep čakalne vrste procesov. Na konec te vrste dodaja sistem tudi vse novoprispele posle.
Algoritem lahko še izboljšamo, če upoštevamo tudi prednost posameznih procesov. Procese z večjo prednostjo uvrščamo v sredino čakalne vrste (proporcionalno njihovi prednosti). Pri tem moramo rešiti nevarnost živega objema za procese z nizko prednostjo. Problem lahko odpravimo na primer z uvedbo **dinamične prednosti**. Čakajočim procesom postopoma dvigujemo prednost in prej ali slej bodo uvrščeni.

Večnivojski algoritem razvrščanja

Učinkovitost algoritma ***round robin*** je seveda zelo odvisna od velikosti časovne rezine, ki jo vsakokrat dodelimo izbranemu procesu. Tu velja tudi upoštevati značilno obnašanje posameznih procesov. Nekateri so zadovoljni s krajšimi časovnimi rezinami. Takim lahko damo višjo prednost. Po drugi strani vpliva na obnašanje procesa tudi sama programska aplikacija. Poznamo računsko intenzivne procese (na primer razne zamudne obdelave podatkov). Za take procese običajno ni pomemben hiter odziv na zunanje dogodke. Druga vrsta procesov pa terja intenzivno komunikacijo s perifernimi napravami. Tu je odzivni čas na zunanje dogodke pomemben, same procedure pa računsko pogosto niso intenzivne. V to skupino sodijo predvsem različni interaktivni sistemi ter sistemi za avtomatizacijo tehnoloških procesov. Poznamo algoritme, ki dopuščajo dovolj uravnovešeno razporejanje računalniškega časa za sožitje obeh vrst procesov. Med njimi je tudi **večnivojski algoritem razvrščanja**, ki ga ponazarja spodnja slika.



Posli, ki so pravkar dobili status "ready to run" se najprej uvrstijo v čakalno vrsto na prvem nivoju. To so novi posli ter procesi, ki so zaradi nekega dogodka prečli iz stanja "blokiran" v stanje "pripravljen" (ready). Ko pride tak proces na vrsto, mu sistem dodeli časovno rezino. Ta zadošča le za res kratke posle in za periferno intenzivne procese. Bolj verjetno je, da bo proces prekinjen in preide na rep čakalne vrste drugega nivoja.

Sistem dodeli časovno rezino procesu iz čakalne vrste drugega nivoja le, če je čakalna vrsta prvega nivoja prazna (če torej ne čaka na uvrstitev noben, po možnosti kratek posel). Prej ali slej (odvisno od zasedenosti sistema) pride na vrsto tudi proces iz druge čakalne vrste. Spet mu sistem dodeli novo časovno rezino. Če posla ne uspemo zaključiti tudi v tej rezini, proces prekinemo in ga uvrstimo na rep naslednje čakalne vrste. Tako imamo lahko kar več nivojev čakalnih vrst. Procese iz čakalne vrste na najnižjem nivoju lahko sistem uvrsti le, če so čakalne vrste višjih nivojev prazne. V čakalni vrsti najnižjega nivoja uveljavljamo na primer že omenjeni algoritem *round robin*.

Ugotovimo lahko, da v tej, zadnji čakalni vrsti čakajo predvsem procesi, ki so računsko intenzivni in dolgotrajni.

Pri porazdeljenih sistemih moramo rešiti predvsem problem razporejanja (task allocation), kar naj zagotovi uravnovešeno obremenitev procesorjev. in minimizira stroške komunikacij med posli. **Dinamično razvrščanje v porazdeljenih sistemih** je običajno razdeljeno na dva dela. En modul ugotavlja, kam prestaviti posel, ki se ne more razporediti na danem vozlišču. Drug del je lokalno razvrščanje, ki poteka po enakih principih, kot jih zasledimo pri centraliziranih, enoprocesorskih razvrščevalnikih.

Programski jeziki za delo v realnem času

Zgled jezika za konkurenčno programiranje: Modula-2

Modula-2 je preprost, vendar močan programski jezik, ki ga lahko uporabljamo v vrsti aplikacij. Temelji na pascalu, uvedel pa ga je leta 1980 Niklaus Wirth. Uvedel je pojem programske enote- modula, kar pa danes zasledimo tudi pri drugih programskih jezikih oziroma orodjih. Zaradi njegove izredne podobnosti s z njegovim prednikom - pascalom se ne bomo spuščali v pretirane podrobnosti njegove sintakse. Spoznajmo ga raje na nekaj zgledih, ki kažejo njegovo uporabnost pri programiranju procesov v realnem času.

Kot prvi zgled vzemimo že znani primer krožnega medpomnilnika. Programski modul naj vsebuje proceduri za zapis (*putData*) in branje (*getData*) znaka v krožno polje *Buffer*; Programski modul za ta primer bi bil v Moduli-2 zapisan takole:

```
DEFINITION MODULE RingBuffer;
CONST BufSize=128;
PROCEDURE putData (c:CHAR);
PROCEDURE getData():CHAR;
VAR cnt:CHAR;
END RingBuffer;

IMPLEMENTATION MODULE RingBuffer;
FROM InOut IMPORT Writestring, Writeln;
CONST
   BufEnd= BufSize-1;
   EmptyErr= 1; (* possible errors *)
   FullErr= 2;
VAR
   Buffer:ARRAY[0..BufEnd] OF CHAR;
   head, tail: CARDINAL;
PROCEDURE err(cause:CARDINAL);(* private *)
BEGIN
   CASE cause OF
      EmptyErr:Writestring("Buffer is empty");
      FullErr: Writestring("Buffer is full");
    END; (* case *)
   WriteLn
END err;

PROCEDURE putData(ch:CHAR);
BEGIN
   IF cnt< BufSize THEN
      INC(cnt)
   ELSE
      err(FullErr);
     RETURN
   END;
```

```
Buffer[head]:=ch;
IF head= BufEnd THEN
   head:=0;
else
   INC(head); END
END putData;

PROCEDURE getData(): CHAR;
BEGIN
   IF cnt0 THEN
      DEC(cnt);
   ELSE
      err(EmptyErr);
     RETURN 0C
  END
  IF tail=BufEnd THEN
      tail:= 0;
  ELSE
     INC(tail);
  END;
   RETURN Buffer(tail);
END getData;

BEGIN (* initialization for RingBuffer Module *)
   head:=0;
   tail:=BufEnd;
   cnt:=0;
END RingBuffer.
```

Opazimo klasično zgradbo modula, ki vsebuje deklarativni, vmesniški del, implementacijo procedur ter iniciacijo spremenljivk.

Modula-2 ima v primerjavi s pascalom nekaj dodatnih tipov spremenljivk. V našem primeru zasledimo tip ***CARDINAL***, ki pomeni celoštevilčno nenegativno spremenljivko.

Posebnost je tudi stavek

FROM *InOut* ***IMPORT*** *Writestring, Writeln;*

Pri tem stavku se lahko malo ustavimo. Modula-2 nima vhodno- izhodnih stavkov, kot so *write, writeln, read, readln* v pascalu. Namesto tega uporabljamo procedure, ki so programirane v drugih modulih. To hkrati pomeni, da si procedure za vhodno-izhodne operacije lahko tudi sami pi{emo (v skladu s potrebami perifernih naprav).
Zgoraj navedeni stavek pove, da bomo uporabljali proceduri *Writestring* in *Writeln*, ki jih vsebuje modul *InOut*.

Stavek ***IMPORT*** v splošnem pove, katere procedure ali spremenljivke, ki jih modul uporablja, so definirane v drugih modulih. Analogno obstaja še konstrukt ***EXPORT***, ki pove, katere spremenljivke oziroma procedure, ki so definirane v danem modulu, so dostopne tudi drugim modulom.

Na istem zgledu si lahko ogledamo sposobnost Module-2 za obravnavo kritičnih sekcij. Zamislimo si, da sta programska procesa, ki ta modul uporabljata (proizvajalec in porabnik) krmiljena z zunanjimi prekinitvami (interrupt handlers). Dostop do krožnega medpomnilnika mora biti ekskluziven, kar rešujemo tako, da v implementaciji modula *RingBuffer* določimo tako prioriteto modula, ki v kritičnih trenutkih prepove prekinitve. Oglejmo si del ustrezno predelanega modula *RingBuffer*.

```
IMPLEMENTATION MODULE RingBuffer;
FROM InOut IMPORT Writestring, Writeln;
CONST
   BufEnd= BufSize-1;
VAR
   Buffer:ARRAY[0..BufEnd] OF CHAR;
   head, tail: CARDINAL;

MODULE crit[4];
IMPORT cnt,Buffer,head,BufSize,BufEnd;
EXPORT putData,DecCnt;
PROCEDURE putData(ch:CHAR);
BEGIN
   IF cnt< BufSize THEN INC(cnt) ELSE  RETURN   END;
  Buffer[head]:=ch;
 IF head= BufEnd THEN
      head:=0;
 ELSE
    INC(head);
 END
END putData;

PROCEDURE DecCnt;
BEGIN
   DEC(cnt);
END DecCnt;
END crit; (* module *)

PROCEDURE getData(): CHAR;
BEGIN
   IF cnt0 THEN DecCnt ELSE RETURN 0C END
   IF tail=BufEnd THEN
      tail:= 0;
   ELSE
      INC(tail);
   END;
   RETURN Buffer(tail);
END getData;

BEGIN (* initialization for RingBuffer Module *)
   head:=0;
   tail:=BufEnd;
```

```
   cnt:=0;
END RingBuffer.
```

V tem zgledu smo namerno opustili javljanje napak, ki je v primeru prekinitveno krmiljenih proizvajalcev in porabnikov izredno težavno.
V Moduli-2 se srečamo še s pojmom **korutine** (coroutine). Ta konstrukt je podoben (ne pa enak) podprogramu (subroutine). Omogoča definiranje paralelnih procesov in predvsem njihove interakcije na nivoju programskega jezika.
V razliko od podprogramov korutine niso hierarhična, gnezdena struktura. Klicanje korutin si lahko predstavljamo kot skakanje iz enega procesa v drugega in (čez nekaj časa) spet nazaj. Klic korutine pomeni, da bo ta nadaljevala svoje delo tam, kjer je prej prekinjena (izjema je prvi klic). Za razliko s podprogrami se spomnimo, da se podprogram začne izvajati vedno pri svojem prvem stavku. Običajno se izvajajo korutine določen časovni interval, nato pa so prekinjene do ponovnega klica. Podprogrami pogosto uporabljajo parametre, to pa pri korutinah ni možno (ker pri klicu ne začenjajo pri prvem stavku). Namesto tega v korutinah uporabljamo globalne spremenljivke.
Korutine tvorimo in upravljamo s klici ***NEWPROCESS*** oziroma ***TRANSFER*** in pri tem uporabljamo spremenljivke tipa ***PROCESS*.Spremenljivka** tega tipa opisuje **stanje** izvajanja (status) korutine. To stanje lahko vsebuje različne elemente, normalno pa vsebuje:

- naslov instrukcije, kjer se mora korutina nadaljevati,
- lokacijo sklada korutine (coroutine's stack),
- lokacijo lokalnih podatkov korutine,

Ko zahtevamo prehod iz ene korutine v drugo (transfer), se stanje kličoče korutine pomni v procesno spremenljivko in obnovi stanje klicane korutine iz njene procesne spremenljivke.

Poglejmo si preprost primer dveh korutin. Prva (glavni program *"primer"*) piše na zaslon terminala črko *"A"*, druga (korutina *PisiB*) pa črko *"B"*. Obe korutini izmenoma kličeta druga drugo.

```
MODULE primer;
FROM SYSTEM IMPORT PROCESS, TRANSFER, NEWPROCESS,SIZE,ADR;
FROM InOut IMPORT Write;
VAR
   co,main:PROCESS;
   WorkSpace: ARRAY[1..200] OF CARDINAL;

PROCEDURE PisiB; (*korutina*)
BEGIN
   LOOP
      Write('B');
      TRANSFER(co,main)
   END (*loop*)
END PisiB;

BEGIN
```

```
   NEWPROCESS(PisiB,ADR(WorkSpace),SIZE(WorkSpace),co);
   LOOP Write('A');
   TRANSFER(main,co);
   END (*loop*)
END primer.
```

Navedeni primer ponazoruje tehniko rezerviranja procesnih spremenljivk (co in main), tvorbo nove korutine (stavek *NEWPROCESS*) in vzajemno klicanje obeh korutin (stavka *TRANSFER*).

Mimogrede smo spoznali še eno posebnost Module-2. To je sistemski modul, ki ga programer normalno ne sestavlja sam. V razliko od klasičnih modulov vsebuje skupino spremenljivk in operacij, ki jih prevajalnik posebno obravnava. V našem zgledu smo tako zahtevali definicije struktur *PROCESS*, *ADR* in *SIZE* ter procedur *NEWPROCESS* in *TRANSFER*).

Za zaključek poglavja o Moduli-2 si poglejmo program za krmiljenje komunikacije z oddaljenim računalnikom in velja za računalnik PC s operacijskim sistemom *MS-DOS*. Klice *DOS* zato izvajamo z s klici procedure *DOSCALL*.

Uporabili bomo medprocesno komunikacijo po principu *"pooling"*. Imeli bomo dva procesa oziroma ustrezni korutini. Ena bo upravljala z lokalnim terminalom, druga bo krmilila komunikacijo z oddaljenim računalnikom.

Za pošiljanje in sprejem podatkov z oddaljenega računalnika uporabljamo korutino *RemIO* (Remote Input- Output). Ta bere statusni register serijskega vmesnika. Če smo ji pripravili znak in če je "transmit buffer" prazen, tedaj rutina znak odpošlje. Podobno velja za branje znaka z vmesnika.

Spet se srečamo s primerom *proizvajalec-porabnik* in z ustreznim medpomnilnikom. Vlogo medpomnilnika (za obe smeri prenosa) imata spremenljivki *KbdCh* in *RemCh*. V spremenljivki *KbdCh* pomnimo podatek, ki smo ga vtipkali na lastni tipkovnici. Ko ta znak odpošljemo, vstavimo v *KbdCh ASCII* kodo 0 (To se v Moduli-2 zapiše kot 0C). *RemCh* vsebuje znak, ki smo ga sprejeli iz vmesnika. Ko ta znak prikažemo na našem zaslonu, vpišemo v *RemCh* kodo 0.

```
MODULE TermPoll;
(*Virtual terminal program for IBM/PC running MS-DOS*)
FROM SYSTEM IMPORT
PROCESS, TRANSFER, NEWPROCESS, SIZE,ADR,WORD, (*standards*)
DOSCALL, INBYTE,ONBYTE; (*DOS specific*)
CONST
   Para= 16; (*16 bytes in a paragraph*)
VAR
   main, remote:PROCESS;
   WorkSpace: ARRAY[0..127] OF CARDINAL;
   KbdCh, RemCh: CHAR;
   ready: BOOLEAN;
```

```
MODULE REMOTE;
IMPORT TRANSFER, INBYTE, OUTBYTE, WORD,
   KbdCh, RemCh, remote, main;
EXPORT RemIO;
PROCEDURE RemIO;
   (* Pooling handler for an 8250 Async Serial I/O chip *)
   CONST
      DataGer= 03F8H;
      StatReg= 03FDH;
      RcvRdy= 0;
      XmtRdy= 5;
   VAR
      stat: WORD;
   BEGIN
      LOOP
         TRANSFER(remote,main);
         INBYTE(StatReg,stat);
         IF (KbdCh # 0C) AND (XmtRdy IN BITSET(stat))
            THEN OUTBYTE(DataReg, KbdCh);
         KbdCh:= 0C
      END;
      IF RcvRdy IN BITSET(stat) THEN
         INBYTE(DataGer, RemCh)
      END;
   END
END RemIO; (* procedure *)
END REMOTE; (* module *)

BEGIN
   KbdCh:= 0C; RemCh:= 0C;
   NEWPROCESS(RemIO,ADR(WorkSpace),
   SIZE(WorkSpace) DIV Para, remote);
   LOOP
      TRANSFER (main, remote);
      (* pool kbd *)
      IF KbdCh = 0C THEN
      DOSCALL(6H,0FFH, KbdCh, ready)
   END;
   (* exit on escape *)
   IF KbdCh = 33 THEN EXIT END;
   (* write screen as necessary *)
   IF RemCh # 0C THEN
      DOSCALL(6H, RemCh);
      RemCh:= 0C;
      END;
END
END Termpoll.
```

Programski jezik ADA

Programski jezik ADA je bil namenjen predvsem za programiranje sprotnih vgnezdenih računalniških sistemov. Danes lahko smatramo, da je ADA eden starejših jezikov. Marsikatero posebnost, ki naj bi lajšala "programiranje kompleksnih sistemov", najdemo pri vrsti sodobnih jezikov. Omejili se bomo zato le na nekatere karakteristike, ki so značilne za problematiko sistemov v realnem času.

- V ADI so uvedeni jezikovni konstrukti za **procese** kot enote asinhronega izvajanja
- Uvedeno je eksplicitno in sistematično obravnavanje izrednih situacij (**Exceptions**)
- Možno je preprosto kombiniranje visokonivojskih in nizkonivojskih vhodno-izhodnih operacij.
- Program, pisan v ADI, je lahko implementiran tudi na večprocesorskih sistemih.

Posamezni procesi pretežno komunicirajo preko procesnih **vhodov** (entry), kar je razvidno že iz zgleda, ki ilustrira mehanizem medprocesne sinhronizacije z **zmenki**. Torej mehanizem zmenkov enotno zagotavlja tako sinhronizacijo kot komunikacijo. Glede na uporabo tega mehanizma, bi lahko delili procese na aktivne (pravzaprav klijente) in pasivne (strežne procese).
Med posebnostmi ADE lahko omenimo tudi izbirni čakajoči stavek **select**. Oblika tega stavka je v splošnem naslednja:

select
*[**when** condition =]*
izbirna_alternativa
*{**or** [when condition =]*
izbirna_alternativa }
*[**else***
zaporedje_stavkov]
***end select**;*

Pri tem je "when condition" varovalo. Kadar je pred alternativo prisotno varovalo, je alternativa "odprta" za izbiro, če je izpolnjen navedeni pogoj. Tako so v danem trenutku lahko nekatere izbire odprte, druge pa blokirane. Sama izbirna alternativa pa ima lahko eno od naslednjih oblik:

accept_stavek [zaporedje_stavkov]
***delay** enostaven_izraz [zaporedje stavkov]*
terminate

Delay uvaja odgoditveni stavek, ki suspendira izvajanje procesa najmanj za čas, izražen z enostavnim izrazom. Tipičen primer uporabe odgoditvenega stavka je realizacija "time-out" postopkov pri različnih gonilnikih ipd. Z njim namreč lahko časovno omejimo čakanje na enega izmed dogodkov, ki ga izražajo prejemne izbire (alternative v istem stavku accept).

Poglavje o ADI zaklju~imo še z enim primerom, ki predstavlja modul za nadzor neke periferne naprave:

```
-- example in ADA
task SERVER is
-- remark, suppose that type LEVEL is (URGENT,NORMAL);
entry REQUEST (LEVEL'FIRST..LEVEL'LAST) (D:DATA);
end;
task body SERVER is
.......
select
accept REQUEST(URGENT)(D:DATA)
do
......
end;
or
when(REQUEST(URGENT)'COUNT=0) =
accept REQUEST(NORMAL)(D:DATA)
do
....
end;
or
delay 100.0;
end select;
end SERVER;
```

Klic takega strežnega programa bi lahko bil tudi časovno omejen, kot to ponazoruje spodnji primer:

```
...
select
SERVER.REQUEST(URGENT)(SOME_DATA);
or
delay 50.0;
--request not satisfied within 50 seconds, do some else...
end select;
...
```

Značilnosti OS za delo v realnem času

Vrste opravil v realnem času

- **Periodična**: izvajajo se regularno v skladu z vnaprej pripravljenim razporedom.
- **Sporadična**: Prožijo jih zunanji dogodki in pogoji. Imamo neko predvideno maksimalno frekvenco izvajanja.
- **Spontana**: Izvajajo se kot odziv na zunanje dogodke, če so viri za to na voljo.
- **Tekoča opravila**: Tista, ki se sicer izvajajo po principu souporabe časa (time sharing).

Razvrščanje

Politika razvrščanja je izredno pomembna in kaže na uporabnost danega operacijskega sistema za delo v realnem času. Razvrščevalnik namreč določa, kateri posel ali proces bo v danem trenutku lahko dejansko tekel. Večina splošno namenskih operacijskih sistemov uporablja politiko delitve časa (time sharing). V praksi je zaželeno, da imajo lahko posamezni procesi različne prioritete in s tem drugačne časovne odmerke. Dodatno izboljšavo pomeni to, da imajo procesi različna stanja. Lahko so speči (sleeping, dormant), čakajoči na nek vir (blocked), pripravljeni na izvajanje (inactive), itd.. Razvrščevalnik (scheduler) torej pri dodeljevanju časa upošteva tudi stanje in prioriteto procesov.

Podpora operacijskega sistema za delo v realnem času

S programerskega stališča lahko od operacijskega sistema zahtevamo različne storitve, Najbolj pogosto zahtevamo dostop do posameznih sredstev (na primer perifernih naprav), lahko pa tudi vplivamo na stanje procesa, zahtevamo medprocesno sinhronizacijo in komunikacijo. To praviloma zahtevamo s takoimenovanimi **sistemskimi klici**. Pri sodobnih, večprocesnih operacijskih sistemih je to sploh edini možen način, ki preprečuje nepooblaščene dostope do skupnih podatkovnih struktur in zagotavlja varno uporabo sistema.

Čeprav zasledimo take mehanizme tudi pri popularnem operacijskem sistemu UNIX, pa velja omeniti, da je za resnično delo v realnem času potrebno, da se sistem na različne dogodke tudi pravočasno odzove. Za tak operacijski sistem torej ne zadošča, da ima mehanizem dodeljevanja računalni{kega časa posameznim programskim procesom (time sharing). Tak operacijski sistem mora biti **dogodkovno krmiljen** (event driven). To pomeni, da reagira na različne programske in aparaturne dogodke. Dogodek je lahko na primer zaključek kakšne vhodno- izhodne operacije ali urin prekinitveni zahtevek.

To so bili primeri **asinhronih** dogodkov, saj ne vemo, v kateri fazi izvajanja tekočega programskega procesa bo do dogodka prišlo. Poznamo še dogodke kot posledico programskih potekov. V to skupino sodijo na primer zaključek ali blokiranje kakšnega programskega procesa, napačna računalniška instrukcija ali nemogoč operand itd. Ti dogodki pa se pri ponavljanju izvajanja programov pojavljajo stalno na istem mestu (v razliko od asinhronih dogodkov). Dogodek je tudi zahtevek za alokacijo kakšnega vira.

Operacijski sistem CHIMERA

Operacijski sistem CHIMERA so razvili na univerzi Carnegie Mellon v Pittsburghu. Razvit je bil za sisteme v realnem času. Tipičen primer takega sistema so senzorsko podprte tehnološke aplikacije (na primer robotizirani sistemi).

Med glavnimi značilnostmi sistema CHIMERA lahko omenimo:

- Tako statično kot dinamično razvrščanje nalog in ločitev teh mehanizmov, kar omogoča preprosto spreminjanje razvrščevalnih algoritmov s strani uporabnika.
- Rokovalnike izpadov oziroma izjem, ki delujejo po drugih prioritetah kot pa same naloge.
- Podporo perifernim napravam, podobno organizirano kot so gonilniki naprav pri Unixu. To zagotavlja aparaturno neodvisnost posameznih nivojev.
- Tudi prekinitveni strežni programi (interrupt handlers) so pisani v jeziku C.
- Aparaturno neodvisnost prevedenih programov

Uvaja več **nivojev**, ki jih grobo lahko razdelimo na (**strežni** (servo), **nadzorni** (supervisory) in **planerski** (planning). Razlika med nivoji je tudi v uporabljenih časovnih intervalih. Čas je posebno kritičen faktor na strežnem nivoju. To zagotavlja visokozmogljivo jedro s komunikacijskimi mehanizmi z malo režijo (low overhead),hitrimi preklopi konteksta ter kratkimi odzivnimi časi na prekinitve. Na nadzornem nivoju so uporabljeni mehanizmi s posredovanjem obvestil (message passing). Sistem predvideva tudi **integracijo simulacijskih** programov v realno okolje. Na voljo pa so tudi orodja za **testiranje** (debuging) razvitega sistema v realnem času. Sistem temelji na interakciji delovnih postaj s profesionalno razvojno programsko opremo in namenskimi enotami za delo v realnem času (RTPU, Real Time Processing Unit). Kot sistemsko arhitekturo so privzeli model NASREM, ki ga uporablja NASA za daljinsko upravljanje robotiziranih postaj. V sistemu zasledimo module za modeliranje sveta, preko katerih je možen dostop do globalne podatkovne baze. Sistem vsebuje tudi modul za **dekompozicijo nalog** (task decomposition). Med posameznimi moduli istega, horizontalnega nivoja poteka sinhrona komunikacija, mednivojska komunikacija pa je asinhrona. Programski moduli za medprocesorsko komunikacijo omogočajo transparentno delo, in uporabnost kodiranih programov, četudi pri tem menjamo tip procesorja.

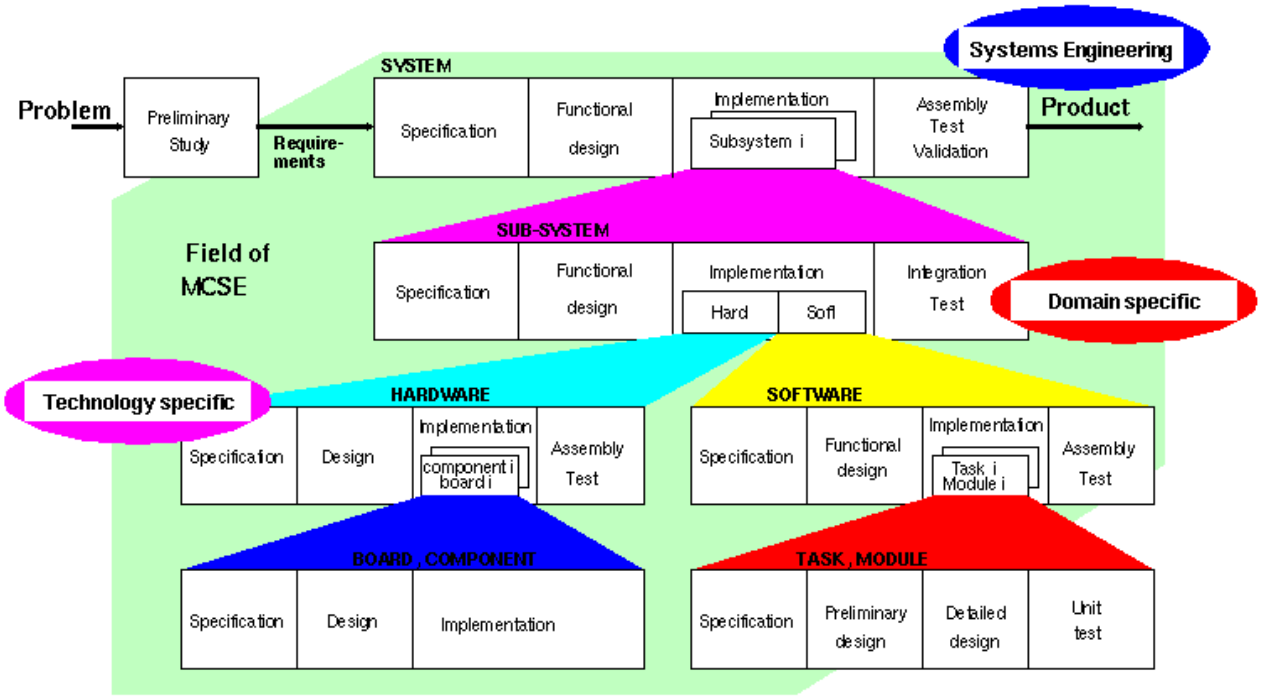
Vsaka RTPU uporablja svoje jedro (real time kernel) in lahko gosti več nalog. Zaradi časovne kritičnosti je opuščena sicer koristna zaščita med nalogami (intertask protection). Opustili so tudi idejo virtualnega pomnilnika, ki poleg tega, da poslabša performanse sistema, vnaša tudi nepredvidljivost. To pa je v nasprotju s kriteriji sistemov v realnem času.

Normalni razvrščevalnik sistema CHIMERA uporablja **dinamični kriterij** "Najprej naloge z najvišjo urgenco" (naloge, ki imajo bli`je časovni limit) , za naloge, ki nimajo časovnega limita (deadline) pa upošteva **statični kriterij** najvišje fiksne prioritete.

Uporabniški procesi so klasični Unixovi procesi, katerim je navezana dodatna knjižnjica CHIMERA (za medprocesorsko komunikacijo).

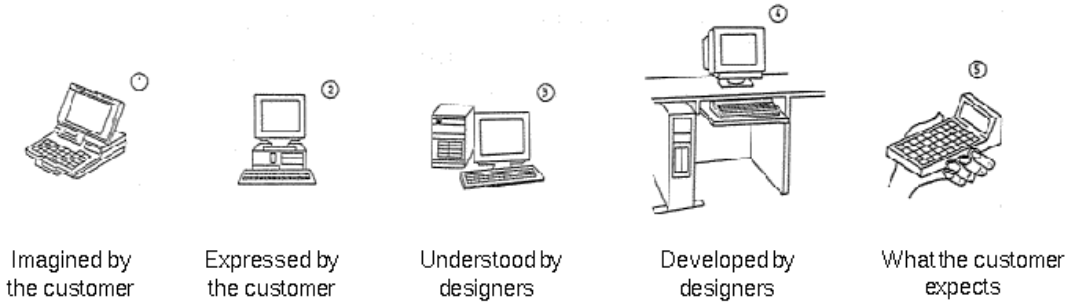
Načrtovanje sistemov v realnem času

Vsako načrtovanje kompleksnih sistemov je težavno, ker pogosto ne vemo vnaprej vseh (tehnoloških) podrobnosti.

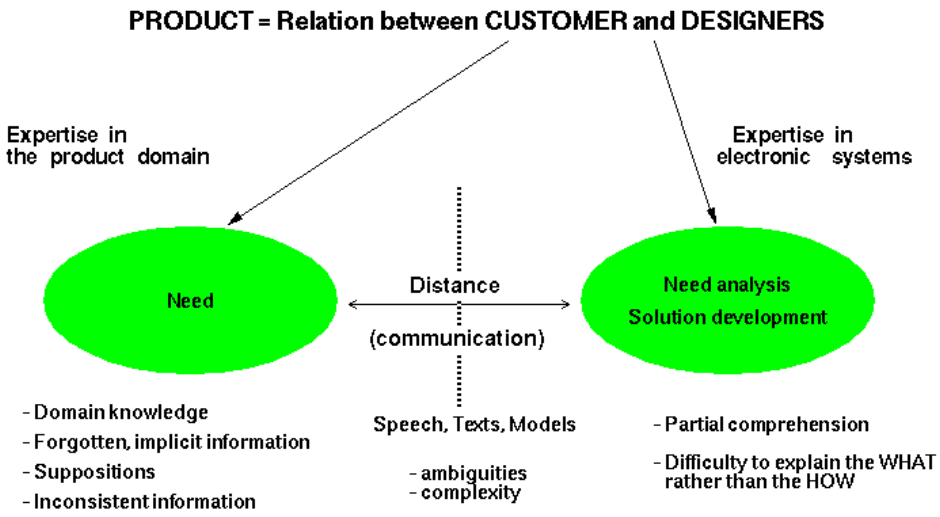


Hierarchical model for a complex system development process

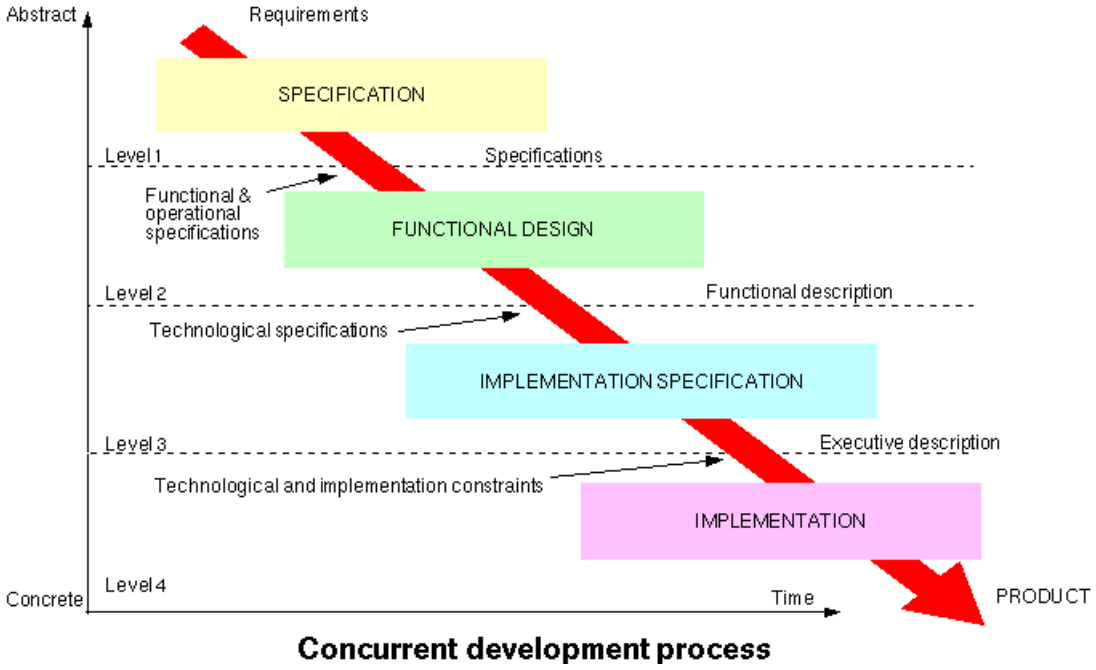
Še posebej to velja za kompleksne dinamične sisteme, ki vsebujejo sožitje večjega števila procesov. Bodoči uporabnik sistema pogosto sploh ne ve, kaj pravzaprav hoče in ne zna tega povedati razvijalcem.



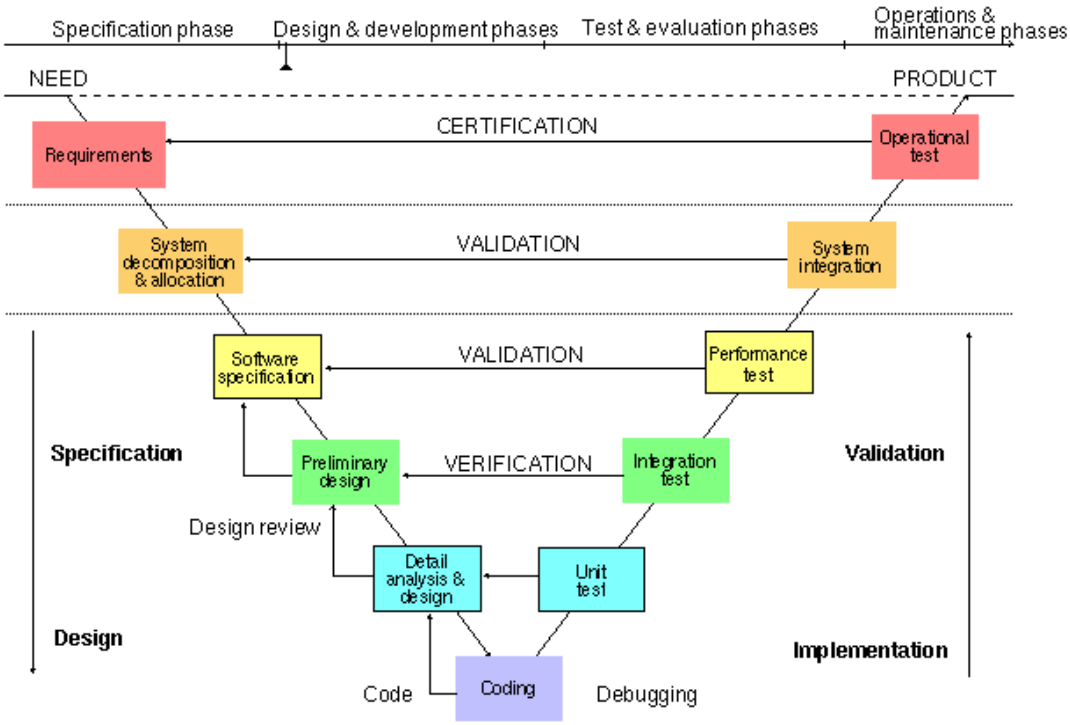
Tako uporabnik kot razvijalec imata vsak svoje izkušnje in predznanje, Pri razvoju novega produkta pa je problem njuna medsebojna komunikacija.



Postopek od ugotovitve po potrebi novega (avtomatiziranega) produkta do realizacije poteka preko več faz. Govorimo o življenskem ciklu projekta.



Skoraj v vsaki fazi lahko pride do novih spoznanj in morebitnih korektur predhodnih faz. To pa pomeni vračanje nazaj in daljšanje projekta.

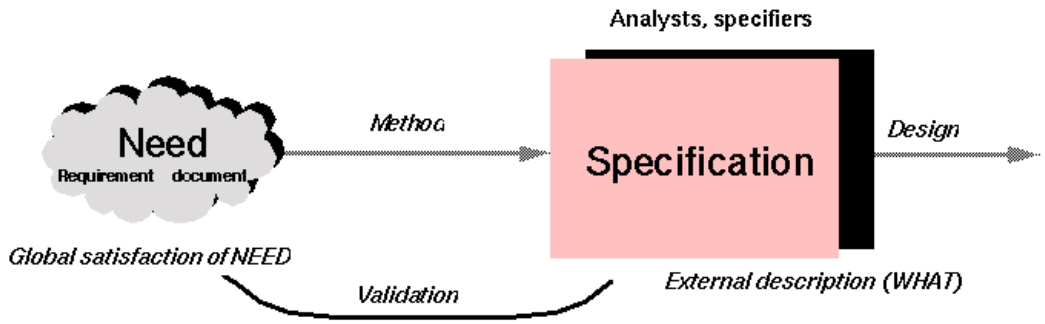


Proces definiranja zahtev

Ena prvih faz je priprava dokumenta z opisom zahtevkov, ki ga lahko kasneje, s pridobivanjem novih spoznanj (vendar sporazumno z uporabnikom avtomatizacije) dopolnjujemo.

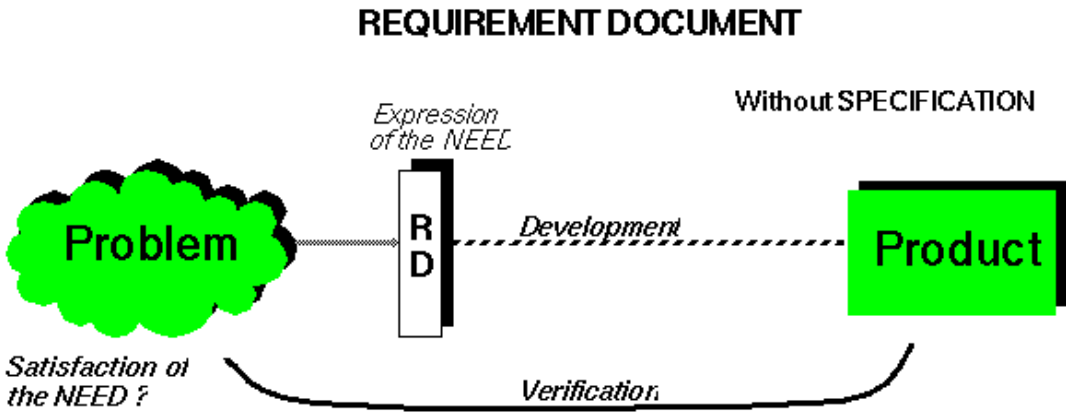
Koraki	**Pripombe**
• Potreba bo formulaciji (funkcionalna analiza, analiza vrednosti) • Priprava prvega dokumenta o zahtevkih • Možnostna študija, dodatne podrobnosti o zahtevah • Odločitev o razvoju, končni dokument o zahtevah	• Potreba po pogovoru in sodelovanju • Predvsem pojasniti, ZAKAJ in nato KAJ • Opisati problem in ne sistema, ki ga rešuje

Priti moramo torej do specifikacij, ki čimbolj ustrezajo zahtevam. Pri tem lahko upoštevamo obnašanje bodočega produkta oziroma sistema, njegove aktivnosti ali njegove podatke in relacije.



MODELS FOR SPECIFICATION:

- Behavior viewpoint (FSM, Petri nets, StateChart,...)
- Activity viewpoint (SADT, Data-Flow Diagram,...)
- Data/object viewpoint (Jackson diagram, Entity/Relation diagram, Object model(s))



Funkcionalna analiza produkta (oziroma sistema) v danem okolju privede do funkcionalnih specifikacij, te pa do operacijskih in tehnoloških specifikacij.Tako dobimo dokument z opisom oziroma specifikacijo sistema.

Operacijske specifikacije	Tehnološke specifikacije
• Pogoji delovanja • Območja vseh parametrov, natančnost,.. • Performanse: odzivni časi, propustnost, transakcije/sec • Operacijski postopki • Rešitve in metode načrtovanja	• Prostorska porazdelitev • Električne specifikacije • Vmesnik človek - računalnik • Vzdrževanje, odvisnost • Električne specifikacije • Druge specifikacije

Dokument s specifikacijo	Kaj s tem dobimo
Funkcionalna analiza okolja Seznam funkcionalnih enot in obnašanja le-teh Razmejitev funkcijskih vhodov in izhodov Funkcionalne specifikacije Seznam vseh sistemskih funkcionalnosti Obnašanje aplikacije za te funkcije Operacijske specifikacije Tehnološke specifikacije	Popolno razumevanje problema Dokument, ki ga lahko verificiramo in je osnova pogodbi Lažje načrtovanje sistema Dobro osnovo za oceno potrebnega časa in stroškov Ponovno uporabljivost za druge podobne projekte

Poznavanje problematike lahko postopoma dograjujemo. Pridobivanje spoznanj in s tem pospeševanje projekta pa lahko dosežemo z uporabo vsaj približno delujočih rešitev, takoimenovanih **prototipov**. Izkušnje z le-temi so dodatni vir znanja in vodijo v še boljše rešitve.

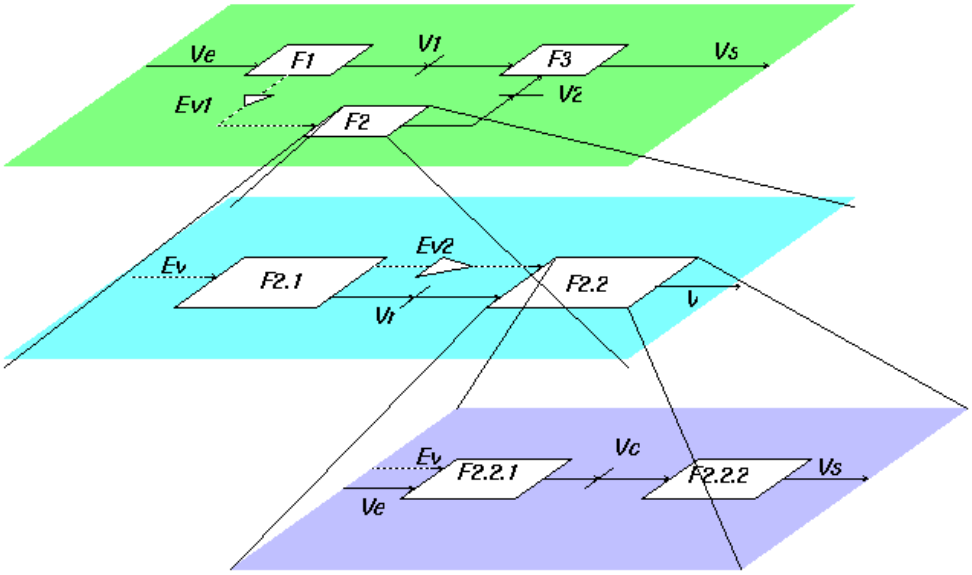
Možnost pridobivanja izkušenj in preverjanja pravilnosti rešitev daje takoimenovano **modeliranje sistemov**. Vsako modeliranje predstavlja bolj ali manj abstraktno ponazoritev realnosti. V našem primeru govorimo o modeliranju procesov. Predstavitev procesa je precej bolj nazorna, če si pri tem pomagamo z grafiko.
Opozoriti velja še na eno značilnost tovrstnega modeliranja: simbiozo č**asovno zveznega** in č**asovno diskretnega** obnašanja oziroma potekov. Hkrati opazimo lahko tesno povezanost med časovno zveznimi oziroma časovno diskretnimi **poteki** ter **podatki** oziroma **nadzorom**. Za modeliranje časovnih karakteristik sistema, ki deluje v realnem času, moramo imeti primeren jezik, v katerem lahko razlikujemo oba tipa obnašanja ter modeliramo interakcijo med njima.
Proces razvoja nekega sistema odraža tradicionalno razločevanje med **problemom** in **rešitvijo**. Poznamo delitev na **logične** in **fizične** modele sistema. Logični model govori o tem, kaj mora sistem v svojem danem okolju narediti in katere podatke mora pri tem hraniti. Fizični ali implementacijski model pa pove, kako je ta sistem realiziran s konkretno tehnologijo.
Razvijalec naj ne bi takoj pohitel z implementacijo. Raje naj bi razvil ustrezni model in ga postopoma dopolnjeval ter končno uporabil pripravljeni model za

implementacijo sistema. Koraki, ki jih razvijalci narede pri razvoju delujočega sistema, predstavljajo razvojni življenski cikel sistema in v vsakem koraku tega cikla naj bi bil na voljo "pregledljiv" (reviewable) model tega sistema. Vsak naslednji model dobimo iz predhodnega z vključevanjem dodatne informacije, ki je v predhodnem nismo upoštevali.

Poznamo več pristopov k načrtovanju sistemov v realnem času. Vsak od le-teh pa ima tudi svoje omejitve. Pristop z analizo podatkovnih tokov (data flow) je zelo primeren in poznan v sklopu **strukturne analize in načrtovanja**.

Strukturirana analiza uporablja diagrame pretoka podatkov in funkcije oziroma **transforme**, ki se na teh podatkih odvijajo. Pomembna značilnost pristopa je uporaba **podatkovnih slovarjev** (data dictionary) in **hierarhične dekompozicije** transformov. Postopno tako ugotovimo funkcionalne zahteve sistema in določimo funkcije podsistemov. Pri tem si lahko pomagamo s z grafično ponazoritvijo sistema, v kateri posamezne transforme predstavimo z mehurčki.

Strukturirano načrtovanje (structured design) temelji na dveh kriterijih (kohezija in povezovanje) za ocenjevanje kvalitete načrtovanja, ter na dekompoziciji modulov od zgoraj navzdol. Smisel takega pristopa je realizacija **modulov**, ki imajo **visoko stopnjo kohezije** in **majhno stopnjo povezovanja**.



Spomnimo se, da je pri načrtovanju sistemov v realnem času zelo pomembno definiranje posameznih procesov (oziroma nalog, tasks) in njihove medsebojne sinhronizacije in komunikacije. Hkrati je koncept skrivanja podatkov (data hiding) pomemben kriterij za dekompozicijo sistema v module . S tem naj bi bila souporaba podatkov (data sharing) zmanjšana na minimum. Tak sistem nedvomno lažje vzdržujemo in dopolnjujemo.
Ko so znane vse funkcije in podatkovne relacije v sistemu, moramo določiti sočasnost posameznih nalog. Pri tem moramo upoštevati asinhrono naravo sistemskih funkcij. Tako ugotovimo, kateri transformi morajo potekati paralelno, in kateri sekvenčno. Na ta način lahko ugotovimo posamezne programske procese.

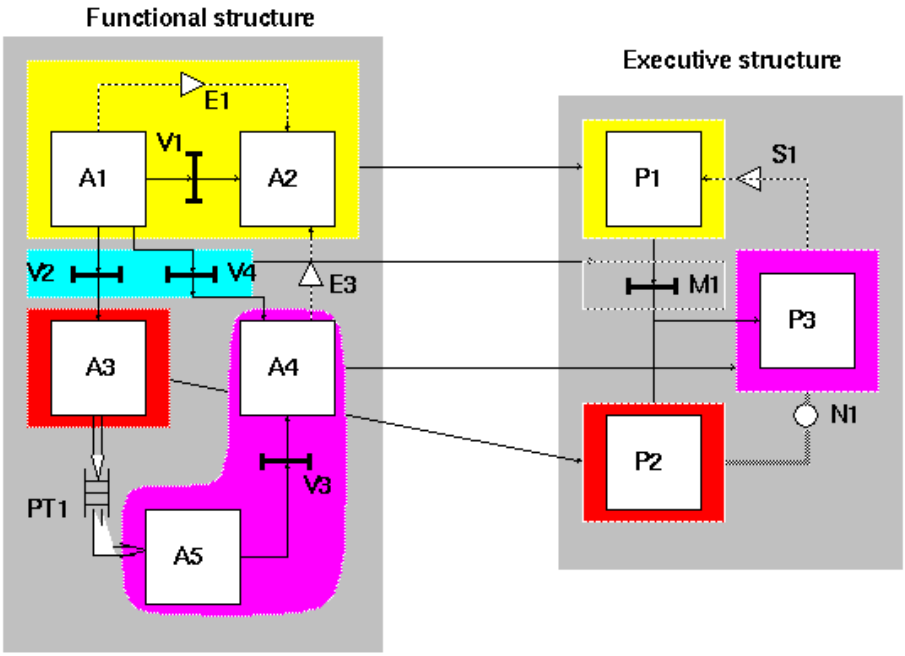
Kriteriji, ki priporočajo, da so določeni transformi posebni procesi, so naslednji:

- Vhodno izhodne relacije in s tem povezane zahtevane hitrosti izvedbe transformov,
- Časovna kritičnost posameznih funkcij zahteva njihovo realizacijo kot ločen proces z visoko prioriteto.
- Računsko intenzivne operacije so lahko poseben proces na nizkem prednostnem nivoju.
- Funkcionalna kohezija skupine transformov kaže na smiselnost njihovega združevanja v skupnem procesu, saj bi bila sicer režija sistema (overhead) previsoka.
- Časovna kohezija (sočasnost izvajanja) skupine transformov spet kaže na smiselnost njihovega združevanja v sklopu enega procesa, saj so verjetno prožene z enakim stimulusom.
- Periodično izvajanje. Transform, ki naj se izvaja ob določenih intervalih, je smiselno vgraditi v ločen proces.

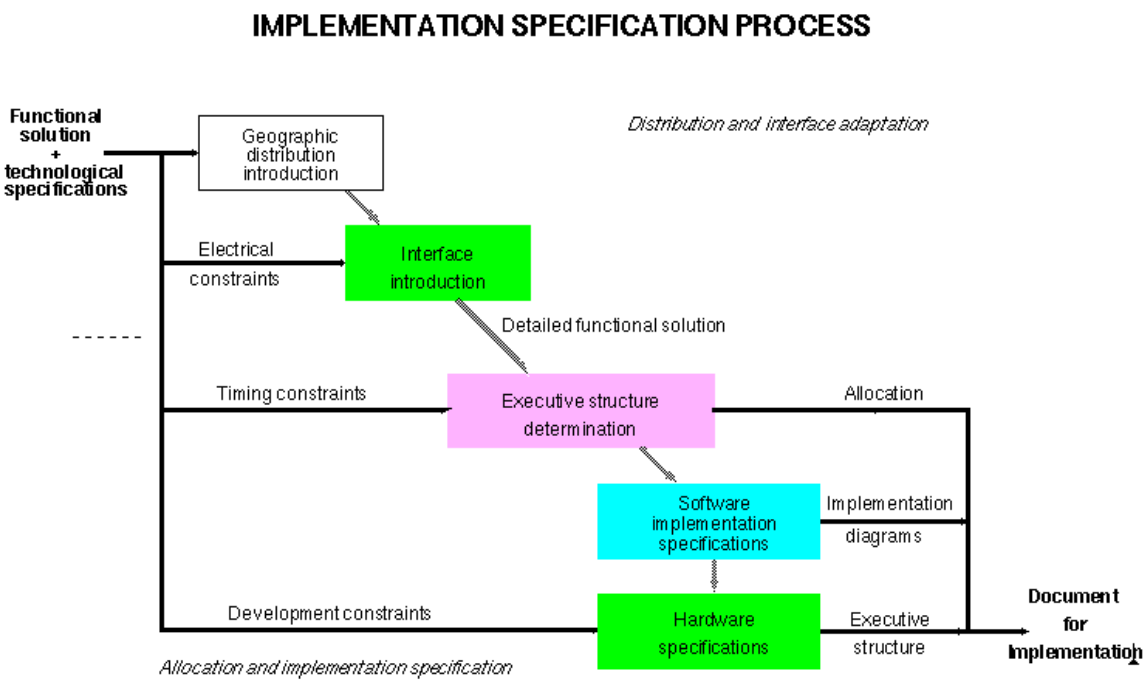
Vsak proces nato predstavimo v obliki ustreznega sekvenčnega programa. Poseben problem in posebnost sistemov v realnem času je, da **odvijanje** posameznih **procesov** ni odvisno le od vhodnega toka **podatkov**, ampak tudi od **stanja** celotnega sistema (state dependent real time system). Ena od rešitev tega problema predvideva uvedbo posebnega modula, ki vsebuje tabelo prehodov stanj in upo{teva trenutno stanje sistema (State Transition Manager). Do določene mere je pri operacijskih sistemih to vloga razvrščevalnika procesov (process scheduler).

Ko imamo opredeljene posamezne procese, lahko predvidimo njihovo izvajanje na enem procesorju ali pa v porazdeljenem okolju.

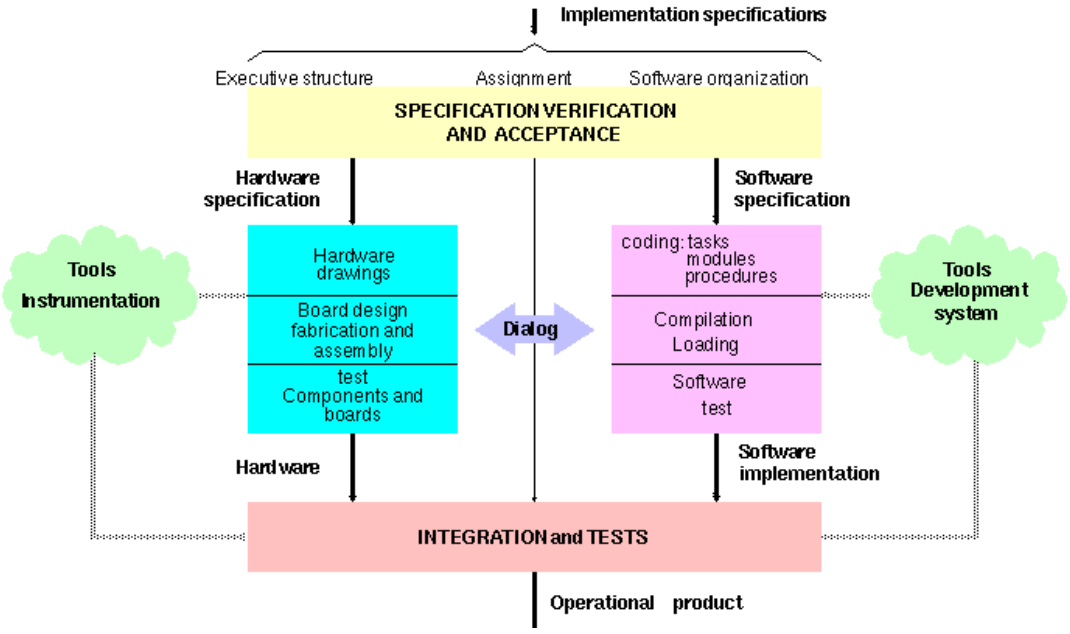
MAPPING: FUNCTIONAL->EXECUTIVE (PARTITIONING - ALLOCATION)



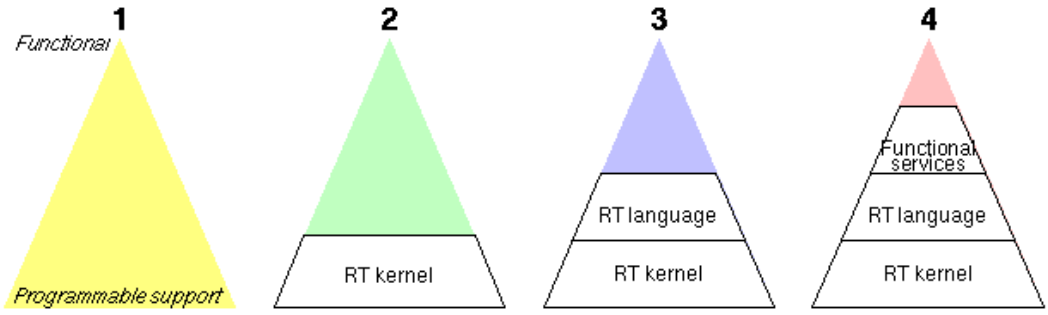
Pri tem upoštevamo različne omejitve:



Sledi definiranje medprocesnih vmesnikov, ki predvidevajo medprocesno komunikacijo s pomočjo sporočil, ali pa le njihovo medsebojno sinhronizacijo s pomočjo dogodkov. In končno sledi implementacija



Programsko opremo lahko implementiramo na različne načine oziroma na različnih nivojih



	Development cost	Reproduction cost	Applications categories
Solution 1	fairly important	nul	small applications heavy constraints
Solution 2	medium	executive	medium
Solution 3	medium	executive	medium and complex
Solution 4	fairly low	executive+services	medium and complex

Testiranje sistemov v realnem času

V sklopu vzpostavljanja sistema za delo v realnem času moramo preveriti pravilnost sožitja aparaturne opreme s posameznimi programskimi moduli. Programer, navajen razvijanja klasičnih programov, se sreča tu z novimi problemskimi dimenzijami. Tako časovno odvijanje programskih procesov ni več nepomembno. Pogosto pa mora upoštevati še **sočasnost** več različnih programskih in drugih **procesov**. Pri svojem delu največkrat uporablja terminale kot klasično diagnostično komunikacijsko sredstvo. Uporaba le-teh pri servisiranju je lahko prepočasna. Prav tako je sam č**lovek** pogosto **neprimeren akter** pri takem testiranju. Njegove reakcije so lahko prepočasne, njegov nadzor sistema in čakanje na naključne, nepredvidljive dogodke je včlasih praktično nemogoče.

Pomagamo si s simulacijo vhodov (na primer branje podatkov namesto s terminalov iz datotek). Prav tako je zelo koristno, če pomembne dogodke sistema beležimo v primerno datoteko (log file). To omogoča kasnej{o (off line) analizo sistema in ugotavljanje vzrokov nepravilnosti.

V vsakem primeru pa velja, da vsak dodatno **vgrajeni testirni program** sam po sebi predstavlja **motnjo v sistemu**. Ta motnja je proporcionalna količini zbiranih podatkov. Torej: če zbiramo več (testnih) informacij, bo njihova točnost slabša!!
Včasih predstavlja posebnost testiranega sistema tudi uporaba posebnih, neklasičnih periferij. Razvijalec se mora torej spoznati tako na skrivnosti programiranja na

sistemskem nivoju, kot tudi na krmiljenje oziroma programiranje navezanih perifernih naprav.

Testiranje sistemov otežuje še delo v okolju, ki je podprto z multiprogramiranjem in multiprocesiranjem in v katerem nastopa kompleksna soodvisnost programov. Vse to povečuje nepredvidljivost sistema in terja dodatno opremo za povečanje zanesljivosti.

Zanesljivost sistemov v realnem času

Vrste izpadov

Kot izpad (failure) razumimo nepravilnost v delovanju nekega sistema. To naj ne pomeni, da sistem v celoti preneha delovati. Izpadi so lahko **trajni** (trdni) ali pa tudi **občasni**. Trajni izpadi so pravzaprav najbolj preprosti oziroma jih najlažje odkrivamo. Občasne izpade težko reproduciramo in je zato tudi nihovo odkrivanje težje. So lahko posledica določenih kombinacij podatkov, frekvenc, zaporedja pogojev. Lahko so tudi posledica staranja komponent sistema.

Tretji tip izpadov nastopi le enkrat in se ne ponovi. To so **prehodni izpadi.**

Izpade lahko delimo tudi glede na njihov izvor. Začnimo z **operaterskimi napakami**, ki so posledica nepravilnega rokovanja s sistemom. Sledijo **instrumentacijske napake** oziroma okvare različnih perifernih naprav. K tej skupini lahko prištejemo okvare v samem računalniku (CPE, pomnilnik itd). Posebej omenimo **izpade pri prenosu podatkov**. Običajno jih razrešimo s ponovnim pošiljanje (dela) sporočila, v katerem je bila ugotovljena napaka. Končno omenimo še **programske napake**, ki skrivajo v sebi slabo načrtovanje programskega sistema. Morda jih nikdar ne odkrijemo, lahko pa pride do izpada ob kakšni nepredvideni kombinaciji podatkov, časovnega zaporedja posameznih dogodkov in programskih instrukcij.

Primarni pogoj sistemov za delo v realnem času je pogosto **zanesljivost**. Kritični podatki so "Mean time to failure: in 'Mean time to repair". Prvi napoveduje verjetni čas prvega izpada, drugi pa trajanje tega izpada oziroma čas za ugotovitev in odpravo napake, ki je izpad povzročila. Potrebne so učinkovite metode diagnostike in preventivnega vzdrževanja.

Tipi diagnostik

Sprotna (on line) diagnostika poteka v času, ko je **sistem delujoč**. Lahko poteka na sistemu sočasno, vendar z nižjo prioriteto, torej predvsem izven konic obremenitve sistema. V to skupino sodi tudi diagnosticiranje sistema na daljavo, preko terminala.
V skupino **sistemske diagnostike** sodijo metode, ki vzpostavijo **najtežje razmere** in maksimizirajo uporabo vseh elementov sistema. Pri tem delajo naključno. Ti sistemi ne odkrivajo napak temveč predvsem slabo delovanje sistema.
Omenimo še **diagnostiko, ki sodeluje s sistemskimi programi.** Diagnostični programi te vrste so programi, ki jih pišemo na isti način kot normalne aplikacijske programe. Diagnostika je na ta način omejena. Vhodno izhodne operacije na primer

potekajo preko klicev v supervizor. Slabost takega diagnosticiranja je, da ga lahko uporabimo šele, ko imamo razvit in stestiran operacijski sistem.
Za odkrivanje napak okvar pogosto uporabljamo postopek rokovanja (**handshaking**). V določenih intervalih si dva procesa pošiljata sporočila tipa "Še sem buden". Če na primer proces A v danem intervalu ne dobi sporočila od procesa B, lahko sklepa na njegovo okvaro, ali pa da je izpadla povezava med obema. Poizkusi lahko {e s posredovanjem sporočila "Ali si buden?" preko neke pomožne poti. Vendar v primeru, da tudi tedaj ne dobi pritrdilnega sporočila, še vedno ne ve, kaj je vzrok izpadu komunikacije (Izpad procesa B, izpad normalne in pomožne komunikacijske povezave)

Postopki za obravnavo izpadov

- Prvi ukrep naj bi bila pravočasna diagnostika. Pri sistemih za delo v realnem času lahko navedemo več vzrokov za uporabo sprotne diagnostike. Ti so:
- Uporaba oddaljenih terminalov; Vzdrževalec lahko iz svojega centra preverja stanje sistema,
- Testiranje, ko sistem deluje, prihrani čas,
- Če okvara(napaka) ni usodna, jo lahko popravimo med delovanjem sistema,
- Možno je takoj sprožiti korekturno akcijo, še med delovanjem sistema.

Odprava okvar

V primeru, da med delovanjem sistema odkrijemo nepravilnosti, lahko ukrepamo na več načinov:

- V primeru prehodne napake lahko s ponovitvijo (instrukcije) ali operacije program normalno nadaljujemo,
- Morda ni možen ponoven zagon sistema (restart) na mestu nastopa napake temveč moramo sprožiti pomožne programske rutine (restart address, emergency action), To metodo pogosto uporabljajo operacijski sistemi, ko na primer odkrijejo, da se neka periferna naprava ne odziva (time-out routine).
- Morda je napaka pokvarila del podatkovne strukture ali celo programske kode. Potrebno je ponovno nalaganje programa,
- Morda je okvara takšna, da sistem ne more niti naložiti programa, potreben je zagon paralelnega, rezervnega računalnika ali pa mora programe naložiti ponovno operater,
- Napaka je lahko "trajna", kar pomeni, da tudi ponovitev programa ne pomaga, morda lahko sistem deluje v omejenem obsegu ali pa sploh ne deluje več,

Postopki v primeru preobremenitev

Eden od možnih vzrokov izpadov sistema je njegova **preobremenitev**. Ta lahko doseže take razsežnosti, da sistem **blokira**. Tako na primer pride do preobremenitve komunikacijskih kanalov. Sistem nima niti dovolj sredstev, da bi zaključil obstoječa sporočila, niti ne more posredovati novih, ki so morda potrebna pri korekturnih akcijah. Problem rešujemo s **predvidevanjem** (anticipating procedures).
Glede na stopnjo izpada lahko uporabimo naslednje korekturne akcije:

- (Pravočasno) blokiranje sprejema novih sporočil (razen izredno nujnih),

- Opuščanje tistih aplikacijskih segmentov (v pomnilniku), ki trenutno niso v rabi, Ta strategija ni nujno dobra, ker sproščanje spomina na tak način daljša vrste čakajočih programov.
- Drastičen ukrep predstavlja opustitev vseh sprejetih podatkov in zahtevek za ponovitev zbiranja (sprejemanja) podatkov. Ta strategija ni primerna, če sistem deluje blizu maksimuma svojih zmogljivosti, saj bi bilo lahko zahtevkov za ponovitev preveč.

Delo v bližini maksimuma je tvegano tudi zato, ker časi odziva oziroma izvajanja posameznih procedur preveč narastejo.

Postopki za obravnavo trajnih izpadov

V primeru, da okvare ne moremo odpraviti, lahko sistem **rekonfiguriramo**. Nek drug podsistem prevzame nalogo izpadle komponente, to informacijo pa morajo prejeti vse komponente, ki so zadolžene za komunikacijo z izpadlim podsistemom. Seveda pa moramo izpadli sistem izobčiti, sicer lahko pride do neljubih situacij (če morda vendarle deluje). Zamislimo si samo, da v nekem porazdeljenem sistemu istočasno delujeta dva koordinatorja.
Postopke za reševanje v primeru trajnih okvar delimo v tri skupine:

- Delitev obremenitve,
- Podvajanje,
- Postopki "fall back"

V prvem primeru sistem ob okvari še vedno deluje, vendar upočasnjeno, saj en podsistem prevzema še naloge okvarjenega podsistema.V to skupino sodi tudi uporaba večračunalniških sistemov, pri katerih je eden normalno rezervni in namenjen preventivnemu vzdrževanju.

V primeru podvajanja uporabljamo dva sistema in v primeru okvare pride do preklopa. To je draga rešitev.

Cenejša je tretja metoda (fall back), pri kateri sistem predvidi v primeru okvare spremenjen način delovanja in ga torej prilagodi okvari. Pogosto govorimo o **mehki degradaciji** sistema, ki, namesto da bi popolnoma izpadel, uporablja alternativne metode ali podkomponente sistema.