

Fakultet elektrotehnike i računarstva
Zavod za elektroničke sustave i obradbu informacija

Programska potpora industrijskih ugradbenih sustava

Operacijski sustavi za rad u stvarnom vremenu

Hrvoje Džapo, Mario Cifrek

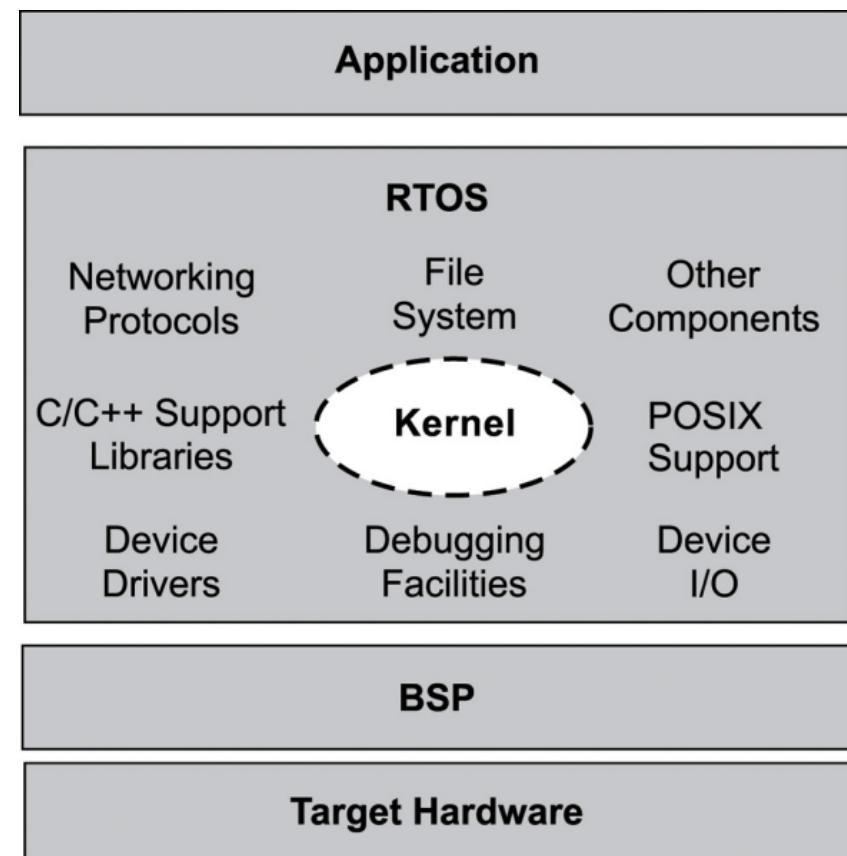
ak. god. 2014./2015.

Operacijski sustavi za rad u stvarnom vremenu

- *Real-Time Operating System (RTOS)*
- programsko okruženje koje omogućuje raspoređivanje zadaća s vremenskim ograničenjima, upravljanje resursima sustava i konzistentni temelj za razvoj aplikacijskog koda
- jezgra (*kernel*) – minimalna implementacija OS-a
- komponente jezgre:
 - *scheduler* – raspoređivač zadataka
 - jezgrini objekti (*kernel objects*) – zadaci (*tasks*, *processes*), semafori (*semaphores*) i redovi poruka (*message queues*)
 - servisi (*services*) – npr. vremenski servisi, prekidi, upravljanje resursima itd.

Operacijski sustavi za rad u stvarnom vremenu

- Real-Time Operating System (RTOS)
 - VxWorks, VRTX, pSOS, Nucleus, QNX, AMX, FreeRTOS ...
- General-Purpose Operating System (GPOS)
 - UNIX, Windows



RTOS vs GPOS

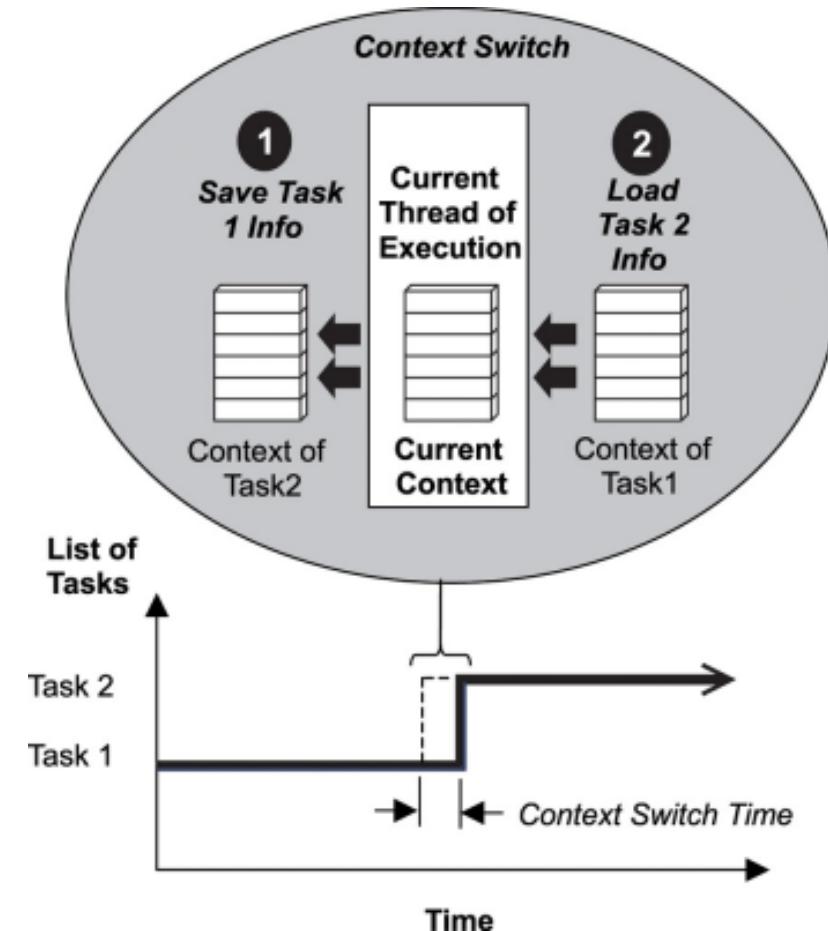
- sličnosti
 - podrška za *multitasking*
 - upravljanje SW/HW resursima sustava
 - OS service API
 - HAL (*hardware abstraction layer*) (*device drivers*)
- razlike (RTOS):
 - pouzdanost
 - skalabilnost
 - odziv u stvarnom vremenu (pogotovo za *hard-real time* sustave)
 - manji memorijski zahtjevi
 - algoritmi raspoređivanja zadataka optimirani za sustave za rad u stvarnom vremenu
 - dizajnirani tako da ne trebaju medije za masovnu pohranu podataka (npr. hard disk)
 - portabilnost

Raspoređivač zadataka (scheduler)

- važan dio jezgre, odlučuje kada se izvodi i prekida svaki zadatak – osnova za jednoprocесорски *multitasking*
- *schedulable entity* – objekt koji se može “natjecati” za CPU vrijeme (*task/process*, ali ne i semafor, *message queue* i sl.)
- *task* – neovisni slijed instrukcija čiji se redoslijed izvođenja može raspoređivati (tipično jedna funkcija s beskonačnom upravljačkom petljom kod jednostavnijeg RTOS-a)
- *process* – slično kao i *task*, ali se koristi kod složenijih operacijskih sustava
 - bolje međusobno razdvajanje procesa i zaštita memorije (uz korištenje virtualne memorije)
 - *thread* – neovisni slijed izvođenja unutar jednog procesa (*multithreading*) (“*lightweight process*”)

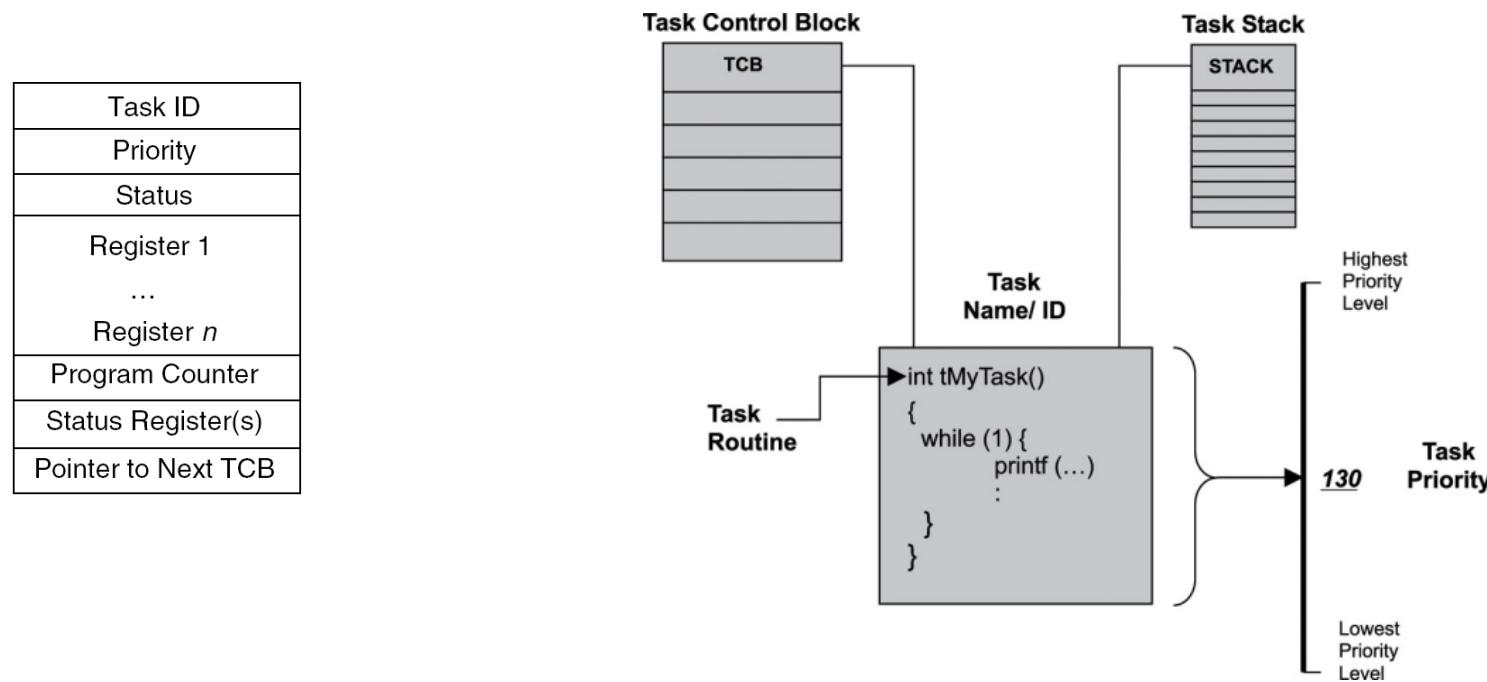
Raspoređivač zadataka (scheduler)

- zamjena konteksta (*context switch*)
- *dispatcher* – dio raspoređivača zadataka koji obavlja zamjenu konteksta i promjenu tekućeg zadatka
- tijek izvođenja:
 - task (aplikacijski kôd)
 - prekid (*interrupt*)
 - jezgra OS-a



Zadaci (*tasks*)

- neovisni slijed izvođenja programskih instrukcija koji se “bori” za procesorsko vrijeme s drugim paralelnim *taskovima*
- konkurentni dizajn programske potpore - dekompozicija u *malene, rasporedive, sekvencialne* programske celine
- TCB (*task control block*) model:

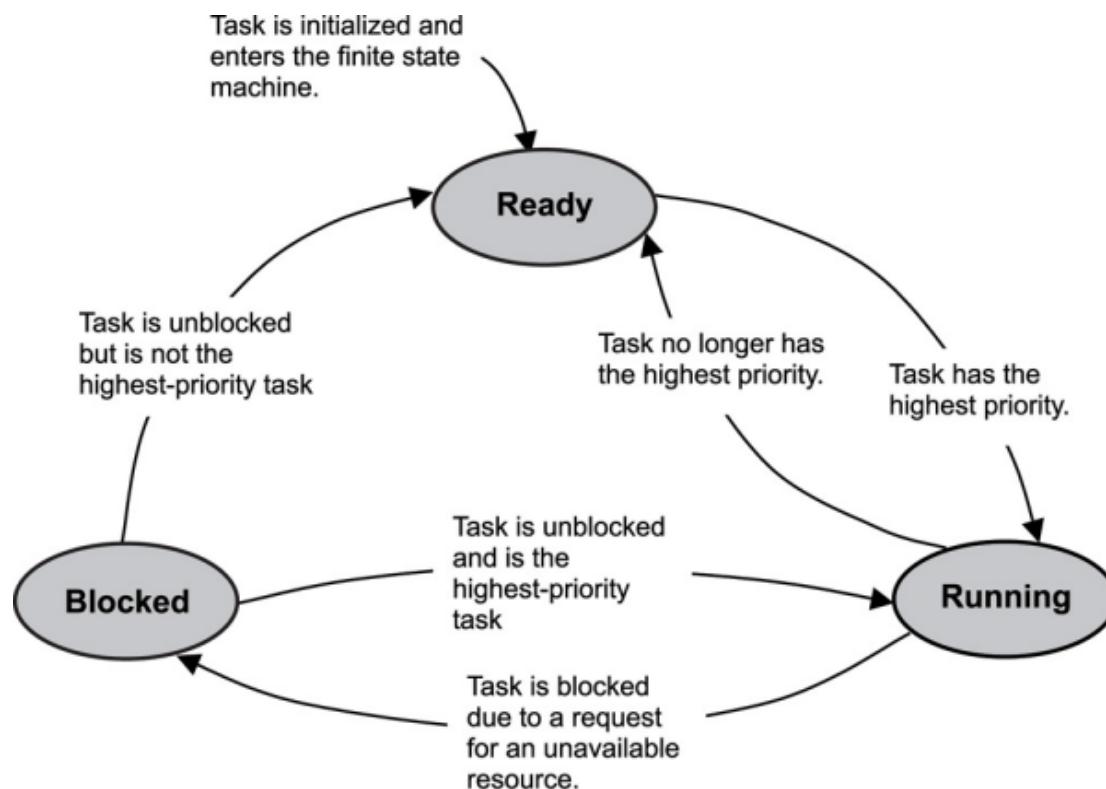


Zadaci (*tasks*)

- stanja zadataka:
 - **pripravan** (*ready*) – zadatak je spreman za izvršavanje, ali ne može dobiti CPU dok se izvršava zadatak višeg prioriteta
 - **blokiran** (*blocked*) – zadatak je neaktivan i čeka na vanjski događaj da se aktivira (ili da npr. prođe zadani vremenski interval)
 - **u izvođenju** (*running*) – zadatak se trenutno izvodi i ima na raspolaganju CPU dok ga ne prekine *scheduler* jezgre OS-a ili se dobrovoljno prebaci u *blocked state*
- neki OS-ovi nude i dodatna stanja – *suspended*, *pended*, *delayed* i sl.

Zadaci (*tasks*)

- stanja zadataka – FSM (*finite-state machine*) model:



Zadaci (*tasks*)

- stanje pripravnosti (*ready state*)
 - nakon kreiranja zadatak automatski ulazi u stanje *ready*
 - nije moguć prijelaz *ready* → *blocked*, samo *ready* → *running*
 - *Task-Ready List*
 - raspored izvođenja ovisi o *scheduling* algoritmu i dodjeli prioriteta (RMS, EDF i sl.)

1 First-Step: State of Task -Ready List

Task 1 Priority=70	Task 2 Priority=80	Task 3 Priority=80	Task 4 Priority=80	Task 5 Priority =90
-----------------------	-----------------------	-----------------------	-----------------------	------------------------

2 Second-Step: State of Task -Ready List

Task 2 Priority=80	Task 3 Priority=80	Task 4 Priority=80	Task 5 Priority=90	
-----------------------	-----------------------	-----------------------	-----------------------	--

3 Third-Step: State of Task -Ready List

Task 3 Priority=80	Task 4 Priority=80	Task 5 Priority=90		
-----------------------	-----------------------	-----------------------	--	--

4 Fourth-Step: State of Task - Ready List

Task 4 Priority=80	Task 5 Priority=90			
-----------------------	-----------------------	--	--	--

5 Fifth-Step: State of Task -Ready List

Task 4 Priority=80	Task 2 Priority=80	Task 5 Priority=90		
-----------------------	-----------------------	-----------------------	--	--

Zadaci (*tasks*)

- stanje izvršavanja (*running state*)
 - na jednoprocesorskom sustavu može se izvršavati samo jedan zadatak u jednom trenutku
 - kontekst zadataka – stanje registara procesora pridruženo zadataku koji se izvodi, pripadni stog zadataka
 - *task* funkcija može pozivati i druge funkcije, pa zato svaki *task* treba imati odvojeni *privatni* stog
 - *preemption* – prekidanje zadataka usred izvođenja
 - može biti i u funkciji koju poziva osnovna *task* funkcija
 - potrebna pohrana konteksta – TCB
- prijelaz *running* → *blocked*:
 - zahtjev za resursima koji trenutno nisu dostupni, čekanje događaja, vremenska pauza
- prijelaz *running* → *ready*: task scheduler!

Zadaci (*tasks*)

- blokirano stanje (*blocked state*)
 - vrlo važna mogućnost jer omogućuje zadacima višeg prioriteta da dobrovoljno prepuste procesor zadacima nižeg prioriteta dok čekaju na neki događaj
 - izgladnjivanje (CPU *starvation*) – pojava kada zadaci nižeg prioriteta ne mogu dobiti CPU na raspolaganje jer ga u potpunosti koriste zadaci s višim prioritetom
- prijelaz *running* → *blocked*:
 - čekanje na semafor (događaj ili signal od drugog zadatka)
 - čekanje poruke (komunikacija kroz *message queue*)
 - čekanje na protekne predefinirana vremenska pauza

Zadaci (*tasks*)

- tipična struktura *taska*:

```
void EndlessLoopTask()
{
    inicializacija
    while(true)
    {
        ...
        tijelo zadatka
        ...
        jedan ili vise blokirajucih poziva
    }
}
```

- sinkronizacija i komunikacija između zadataka:
 - *intertask primitives* – semafori, redovi poruka (*message queue*), signali, cjevovodi (*pipe*) itd.

Primjer

```
struct { // podaci za svaki spremnik
    float level;
    int time;
} level_data[NUMBER_OF_TANKS];
void vButtonTask() {
    while(true) {
        wait(button_pressed); // blokiranje zadatka
        i = button_pressed_id;
        printf("%f, %d\n", level_data[i].level, level_data[i].time);
    }
}
void vCalculateLevels() {
    while(true) {
        // procitaj razine za svaki spremnik
        // proracun...
        level_data[i].level = ...
        level_data[i].time = ...
        i = (i + 1) % NUMBER_OF_TANKS;
    }
}
void main() {
    InitRTOS();
    StartTask(vButtonTask, HIGH_PRIORITY);
    StartTask(vCalculateLevels, LOW_PRIORITY);
    StartRTOS();
}
```

Da li program radi ispravno?

Primjer

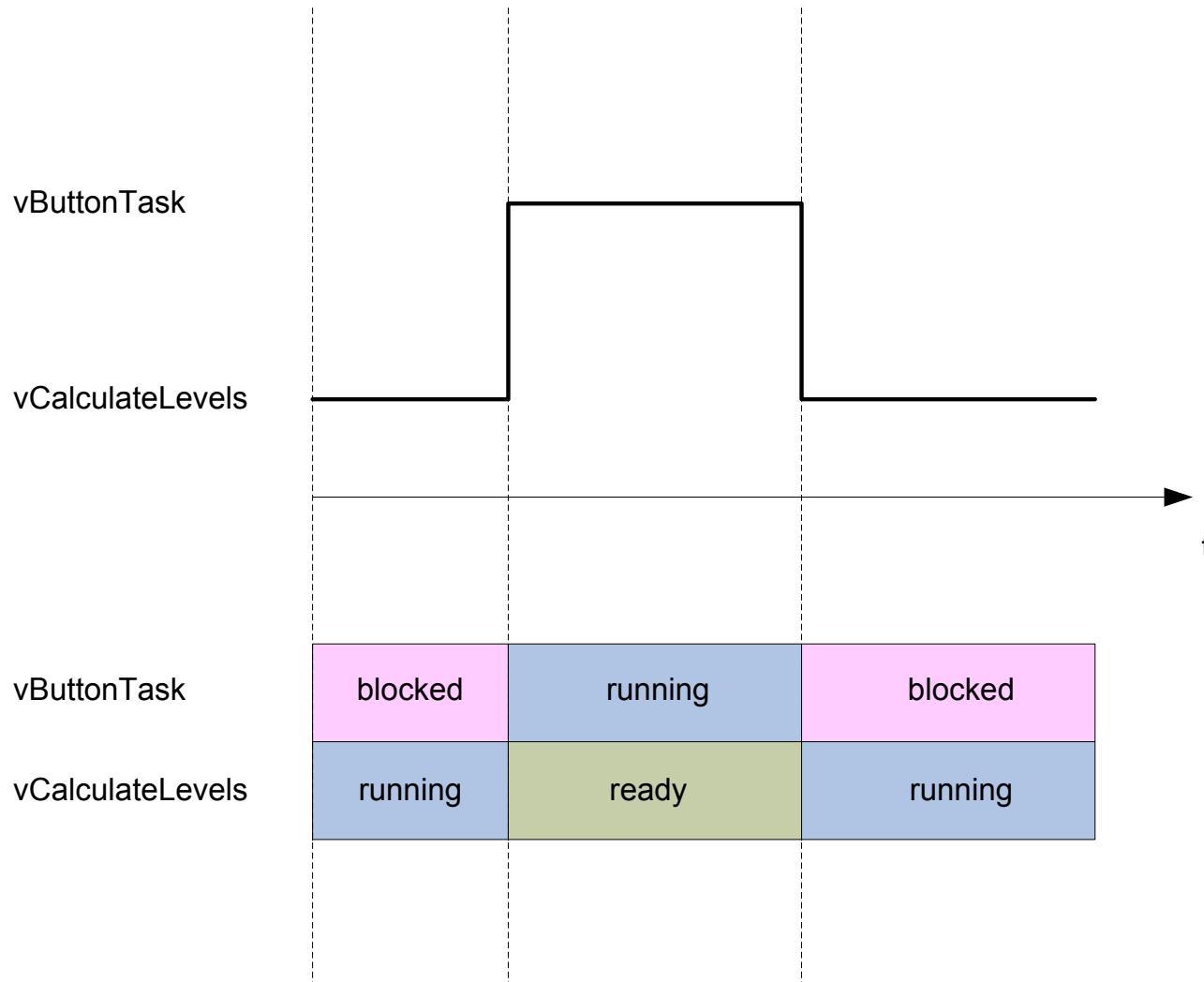
```

struct { // podaci za svaki spremnik
    float level;
    int time;
} level_data[NUMBER_OF_TANKS];
void vButtonTask() {
    while(true) {
        wait(button_pressed); // blokiranje zadatka
        i = button_pressed_id;
        printf("%f, %d\n", level_data[i].level, level_data[i].time);
    }
}
void vCalculateLevels() {
    while(true) {
        // procitaj razine za svaki spremnik
        // proracun...
        level_data[i].level = ...
        level_data[i].time = ...
        i = (i + 1) % NUMBER_OF_TANKS;
    }
}
void main() {
    InitRTOS();
    StartTask(vButtonTask, HIGH_PRIORITY);
    StartTask(vCalculateLevels, LOW_PRIORITY);
    StartRTOS();
}
  
```

Prekinuto izvođenje atomarne operacije!

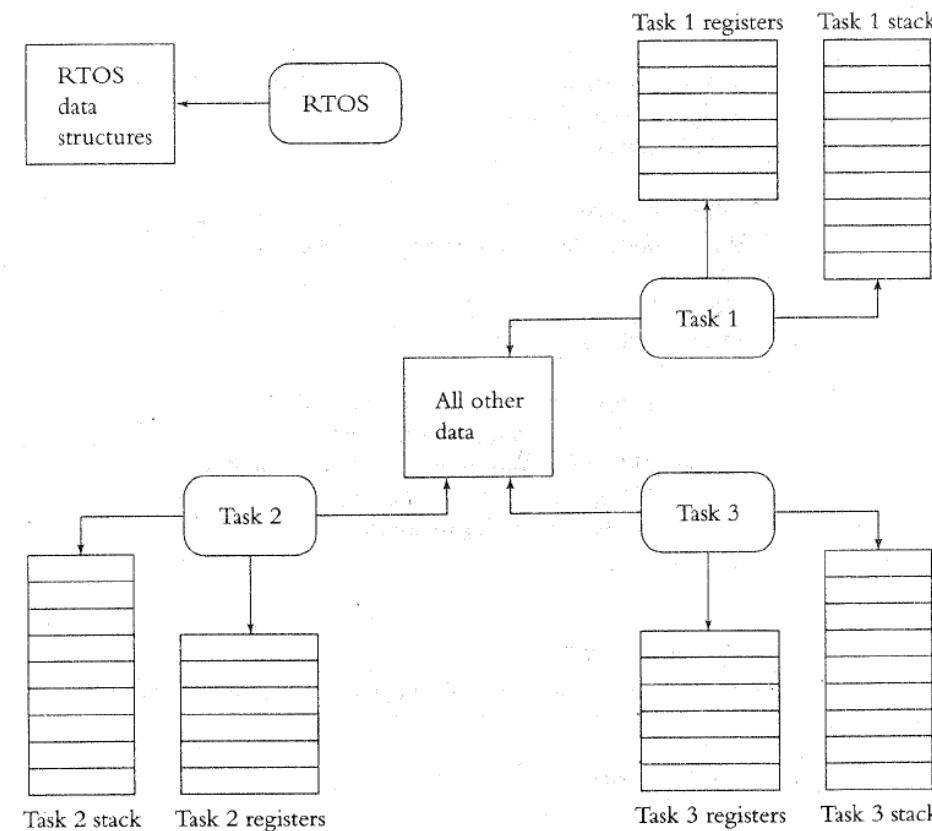
Kako realizirati kritičnu sekciju?

Primjer



Zadaci i podaci

- privatni kontekst – registri, PC, stog



Problem dijeljenih resursa (shared-data problem)

- primjer: funkcija koja se poziva iz više taskova

```

int eventCounter = 5;
void Task1() {
    ...
    IncEvCnt(9);
    ...
}
void Task2() {
    ...
    IncEvCnt(11);
    ...
}
void IncEvCnt(int value){
    eventCounter += value;
}
  
```

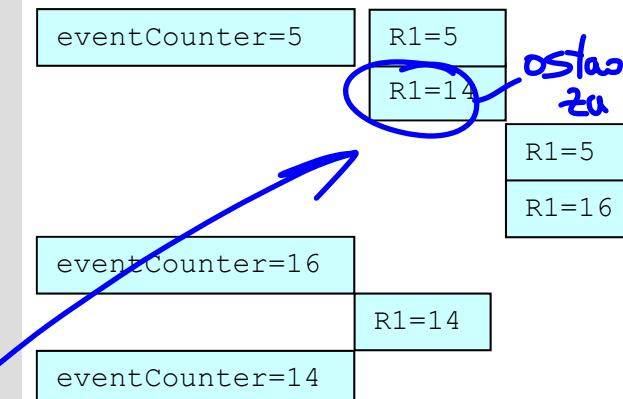
```

; void IncreaseEventCounter(int value) {
; .eventCounter += value;
MOV R1, (eventCounter)
ADD R1, (value)
MOV (eventCounter), R1
RETURN
  
```

```

; Task1:IncEvCnt (9)
MOV R1, (eventCounter)
ADD R1, (9)
; RTOS->Task2
; Task2:InvEvCnt (11)
MOV R1, (eventCounter)
ADD R1, (11)
MOV (eventCounter), R1
; RTOS->Task1
; Task1:InvEvCnt (9)
MOV (eventCounter), R1
  
```

task vracá



Očekivani rezultat: $5+9+11=25$

Dobiveni rezultat: 14

Ponekad!
0,01%

Reentrant funkcije

- funkcija *IncEvCnt()* u prethodnom primjeru poziva se iz više taskova 
- **reentrant** funkcija – funkcija koja može biti pozvana od strane više procesa, a da pri tome uvijek ispravno funkcionira
- uvjeti koje moraju zadovoljavati *reentrant* funkcije:
 - funkcija ne smije rukovati s podacima na neatomaran način, osim ako se radi o *privatnim* podacima taska (koji se nalaze npr. na stogu taska)  
 - globalne i statičke varijable dijeljene su između taskova!
 - funkcija ne smije pozivati *non-reentrant* funkcije
 - funkcija ne smije pristupati sklopolju na ne-atomaran način
- koji uvjet ne zadovoljava funkcija *IncEvCnt()*?

Reentrant funkcije

- primjer: da li je ova funkcija *reentrant*?

```
bool bError;  
void ShowLCDMessage(int x, int y, char* msg)  
{  
    if (!bError) {  
        lcdprintxy(x, y, msg);  
        bError = true;  
    }  
    else {  
        lcdprintxy(0, 0, "Error!");  
        bError = false;  
    }  
}
```

nezaštićeni pristup globalnim resursima

stog procesa

ispravno samo ako je funkcija koja se poziva reentrant

Primjer

```

struct { // podaci za svaki spremnik
    float level;
    int time;
} level_data[NUMBER_OF_TANKS];
void vButtonTask() {
    while(true) {
        wait(button_pressed); // blokiranje zadatka
        i = button_pressed_id;
        printf("%f, %d\n", level_data[i].level, level_data[i].time);
    }
}
void vCalculateLevels() {
    while(true) {
        // procitaj razine za svaki spremnik
        // proracun...
        level_data[i].level = ... - .prezid
        level_data[i].time = ...
        i = (i + 1) % NUMBER_OF_TANKS;
    }
}
void main() {
    InitRTOS();
    StartTask(vButtonTask, HIGH_PRIORITY);
    StartTask(vCalculateLevels, LOW_PRIORITY);
    StartRTOS();
}

```

Kako riješiti shared data bug?

Kako primijeniti rješenje na reentrant funkcije?

Semafori (Semaphores)

- **semafor** – objekt jezgre OS-a nad kojim zadaci (*tasks*) mogu obavljati dvije osnovne operacije (“zauzmi” i “otpusti”), a služe za sinkronizaciju i međusobno isključivanje zadataka kod pristupa resursima
 - programska apstrakcija koja omogućuje realizaciju kritičnih sekcija, mutex-a (*mutual exclusion kernel object*), signalizaciju između procesa itd.
- terminologija operacija nad semafora varira u različitim OS-ovima/API-jima:
 - *raise/lower, get/give, take/release, pend/post, p/v, wait/signal, decrement/increment* itd.

Varijable VS SEMAPHOR

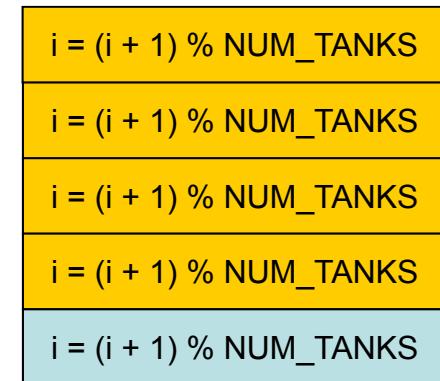
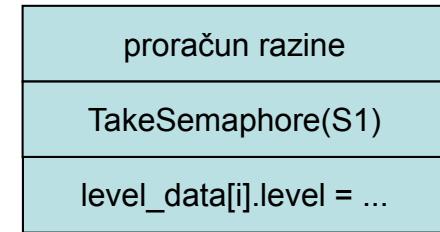
Primjer

```

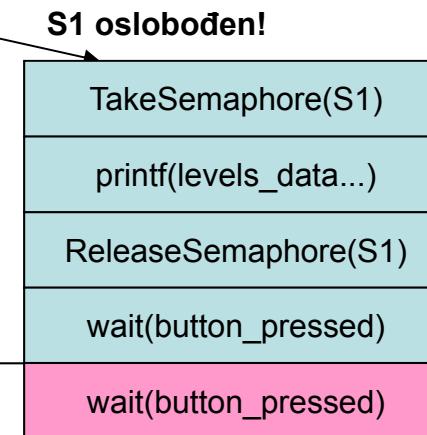
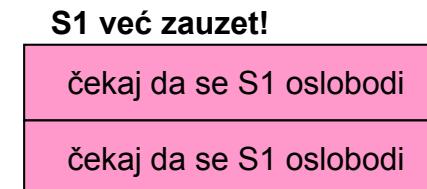
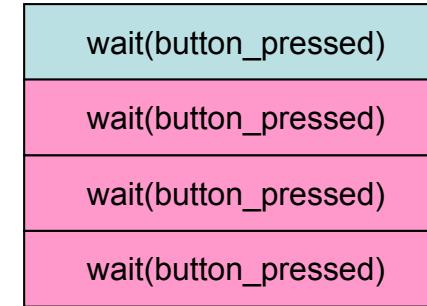
void vButtonTask() {                                // HIGH-PRIORITY
    while(true) {
        wait(button_pressed);                      // blokiranje zadatka
        i = button_pressed_id;
        TakeSemaphore(S1);
        printf("%f, %d\n", level_data[i].level, level_data[i].time);
        ReleaseSemaphore(S1);
    }
}
void vCalculateLevels() {                           // LOW-PRIORITY
    int i = 0;
    while(true) {
        // procitaj razine za svaki spremnik
        // proracun...
        // proracun...
        TakeSemaphore(S1);
        level_data[i].level = ...
        level_data[i].time = ...
        ReleaseSemaphore(S1);
        i = (i + 1) % NUMBER_OF_TANKS;
    }
}
  
```

*- može se imati
 koliko god semafora*

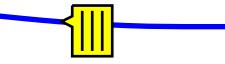
vCalculateLevels



PRIORITETNIJI vButtonTask



Primjer – semafori i reentrant funkcije



```

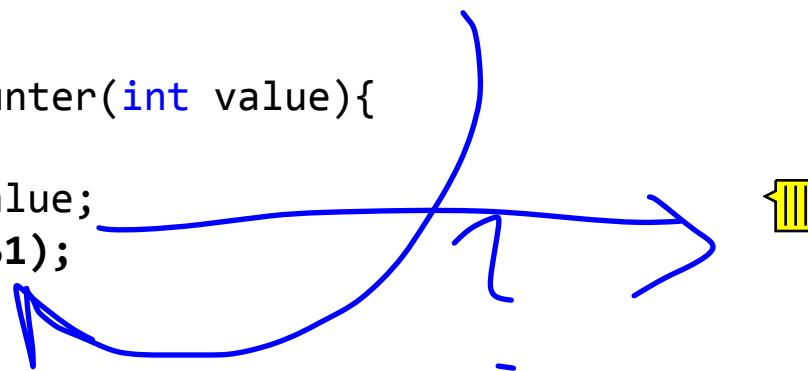
int eventCounter;
Semaphore S1;
void Task1() {
    ...
    IncreaseEventCounter(9);
    ...
}
void Task2() {
    ...
    IncreaseEventCounter(11);
    ...
}
void IncreaseEventCounter(int value){
    TakeSemaphore(S1);
    eventCounter += value;
    ReleaseSemaphore(S1);
}
  
```

ugradbeni

Uočiti: *Take/ReleaseSemaphore()* funkcije uzimaju objekt kao parametar!
 ⇒ broj semafora nije ograničen!

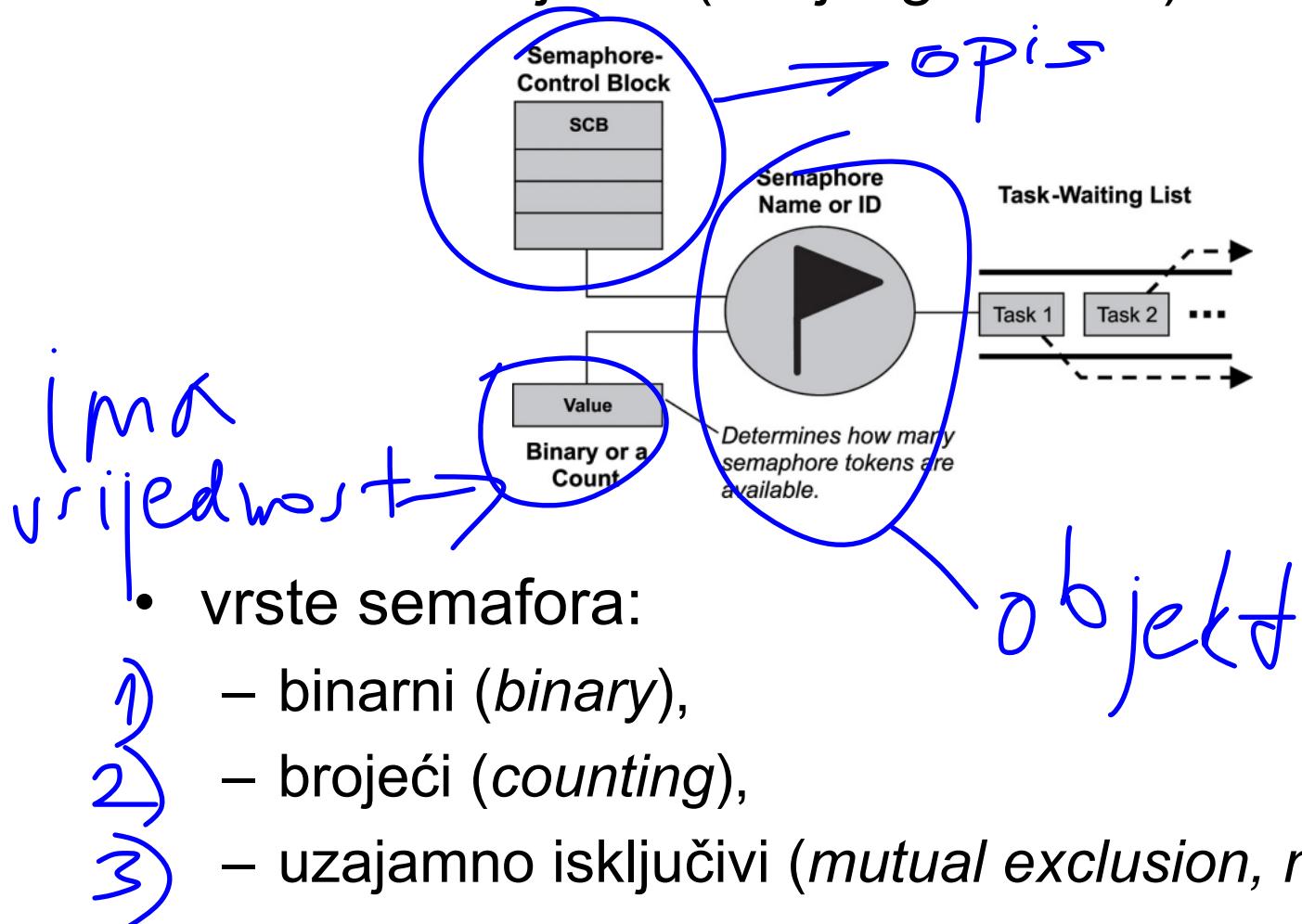
Koja je prednost korištenja više semafora?

Kako OS zna koji semafor štiti koji resurs?



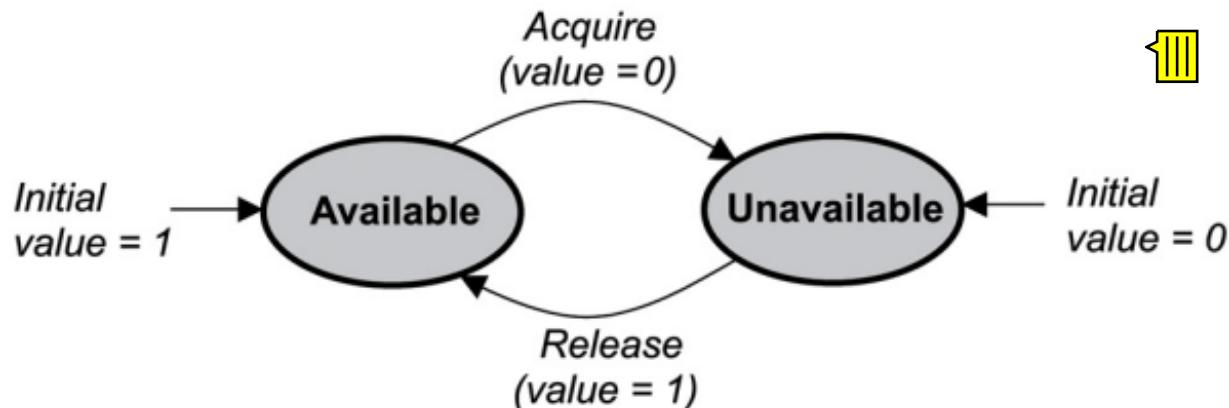
Implementacija semafora

- struktura objekta (dio jezgre OS-a):



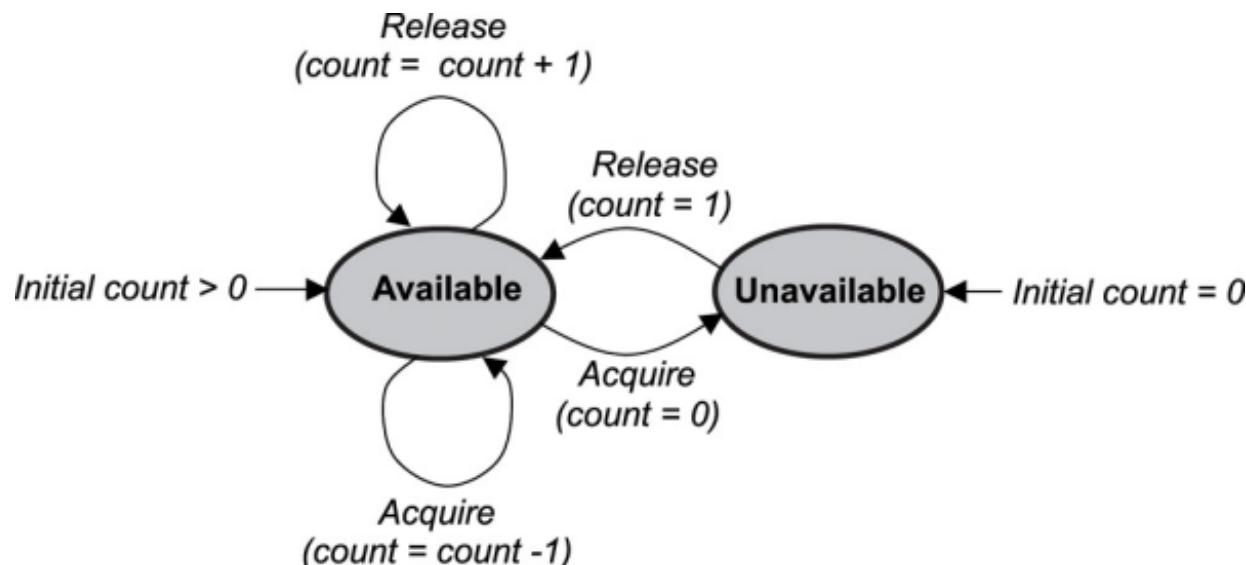
Binarni semafor

- moguće vrijednosti: 0 (zauzet), 1 (slobodan)
- inicijalna vrijednost može biti 0 ili 1; bilo koji proces može pristupiti semaforu i mijenjati mu vrijednost (globalni resurs, nije zaštićen sam po sebi!)
- FSM:



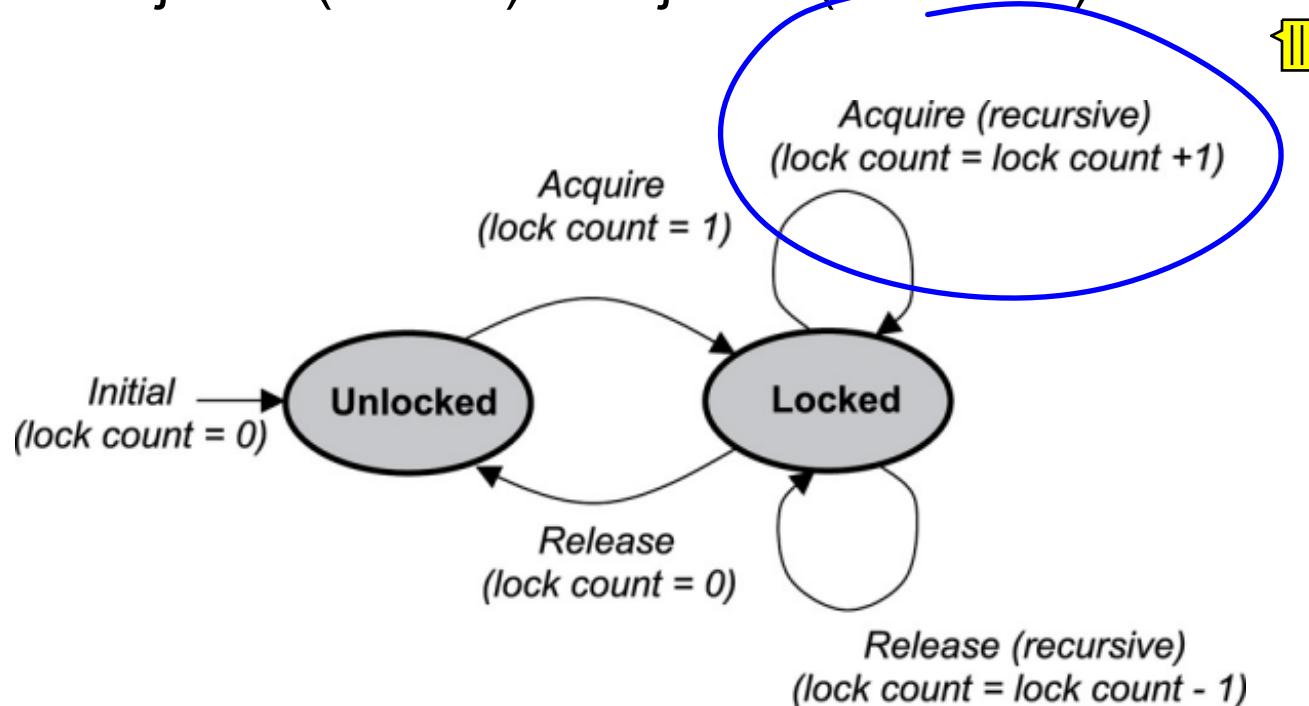
Brojeći semafor

- semafor se može zauzeti više puta
- vrijednosti: 0 (zauzet), >0 (slobodan)
- proces koji uzima semafor *token* dekrementira vrijednost (najviše dok se ne dođe do nule)
- otpuštanje semafora povećava mu vrijednost za jedan



Mutex *sličan binarnom*

- posebna vrsta binarnog semafora koja podržava vlasništvo na objektom, rekurzivno zaključavanje, sigurno brisanje zadataka, posebne protokole izbjegavanja problema svojstvenim uzajamnom isključivanju itd.
- stanje: zaključan (*locked*) i otključan (*unlocked*)



Mutex

- **vlasništvo** – samo proces koji je zaključao *mutex* može ga i otključati
- **rekurzivno zaključavanje** – zadatku koji je zaključao *mutex* omogućava se višestruko zaključavanje dijeljenog resursa – izbjegavanje *deadlocka*
 - razlika prema brojećem semaforu – rekurzivno zaključavanje dozvoljeno je samo procesu-vlasniku, dok kod brojećeg semafora bilo koji proces može uzeti *token*
- **sigurno brisanje zadataka** – zbog pojma vlasništva nad semaforom, moguće je implementirati zaštitu od preuranjenog brisanja procesa-vlasnika
- **protokoli** izbjegavanja problema svojstvenim uzajamnom isključivanju:
 - *priority inversion, priority ceiling*

Primjeri korištenja semafora

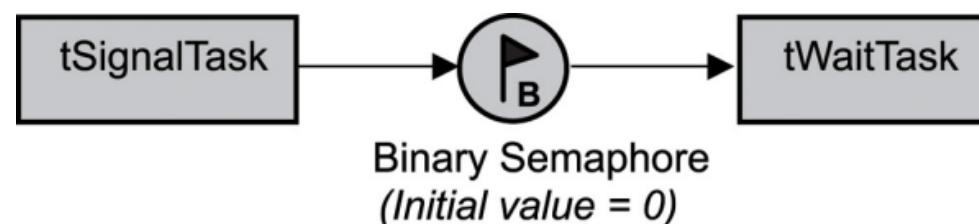
- “wait-and-signal” sinkronizacija zadataka

```

void vWaitTask()          // HIGH_PRIORITY
{
    TakeSemaphore(S1);   // blokiran, dok
                          // vSignalTask() ne
                          // osloboди S1
    ... do some work ...
}
void vSignalTask()        // LOW_PRIORITY
{
    ... do some work ...
    ReleaseSemaphore(S1); // oslobadja semafor
}
  
```



R D / H A R N |



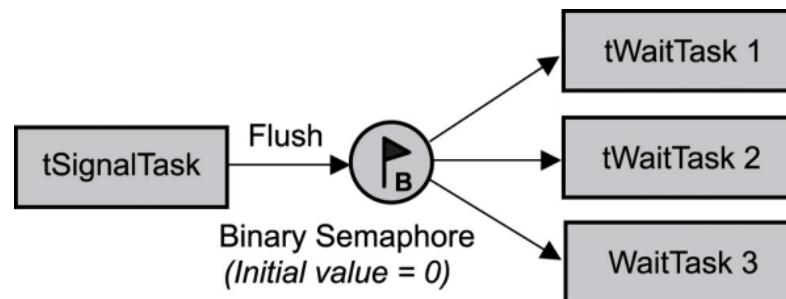
Primjeri korištenja semafora

B / NARNI

- “multiple-task wait-and-signal” sinkronizacija

```

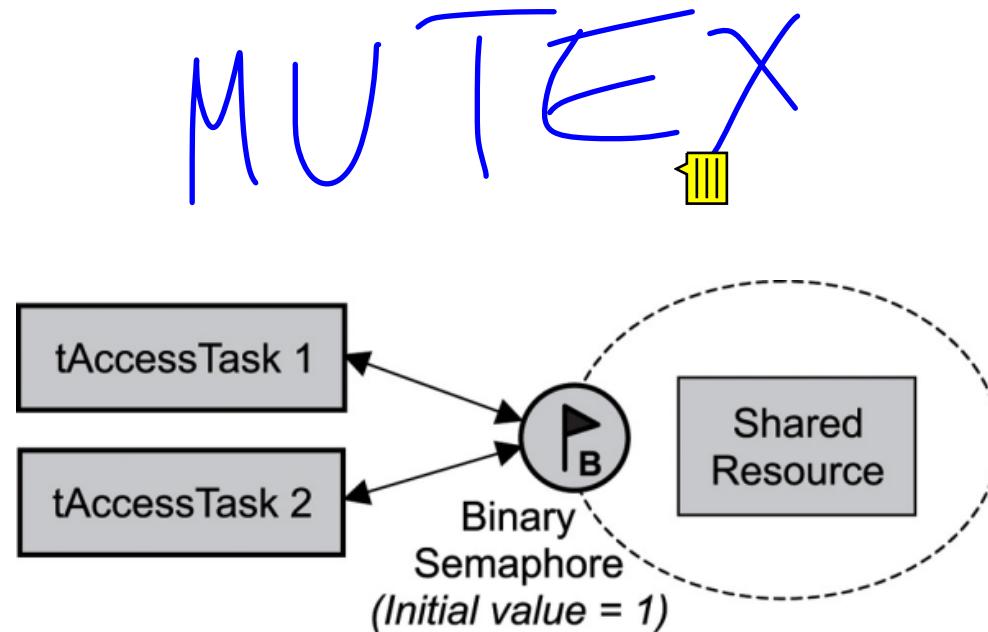
void vWaitTask1() { // HIGH_PRIORITY
    ... do some work ...
    TakeSemaphore(S1); // blokiran, dok vSignalTask() ne osloboди S1
}
void vWaitTask2() { // HIGH_PRIORITY
    ... do some work ...
    TakeSemaphore(S1); // blokiran, dok vSignalTask() ne osloboди S1
}
void vWaitTask3() { // HIGH_PRIORITY
    ... do some work ...
    TakeSemaphore(S1); // blokiran, dok vSignalTask() ne osloboди S1
}
void vSignalTask() { // LOW_PRIORITY
    ... do some work ...
    ReleaseSemaphore(S1); // oslobadja semafor i propusta druge zadatke
}
  
```



Primjeri korištenja semafora

- “Single Shared-Resource-Access” sinkronizacija

```
void vAccessTask1() {  
    ...  
    TakeSemaphore(S1);  
    R/W shared resource  
    ReleaseSemaphore(S1);  
    ...  
}  
void vAccessTask2() {  
    ...  
    TakeSemaphore(S1);  
    R/W shared resource  
    ReleaseSemaphore(S1);  
    ...  
}
```



Primjeri korištenja semafora

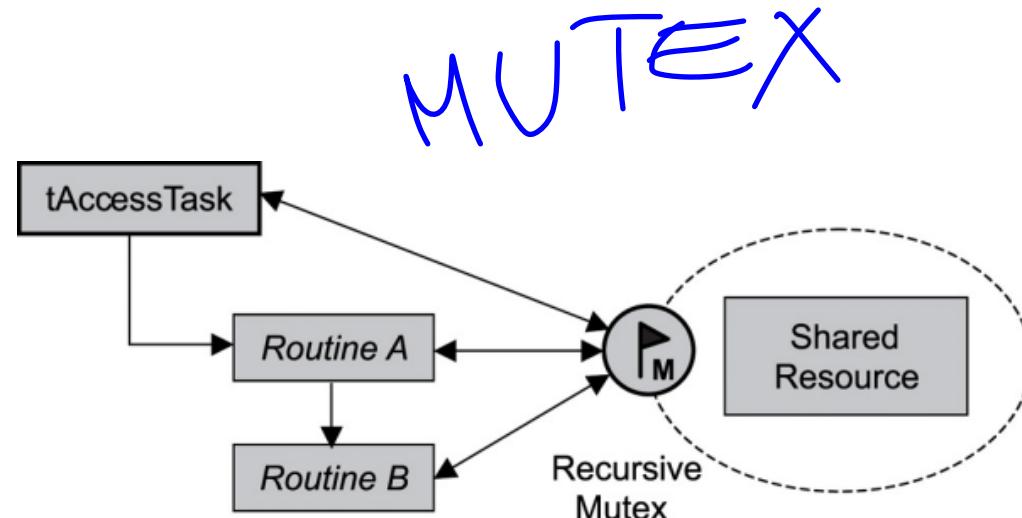
- “Recursive Shared-Resource-Access” sinkronizacija

```

void vAccessTask () {
    ...
    Acquire mutex
    Access shared resource
    Call Routine A
    Release mutex
    ...
}

void Routine A() {
    ...
    Acquire mutex
    Access shared resource
    Call Routine B
    Release mutex
    ...
}

void Routine B () {
    ...
    Acquire mutex
    Access shared resource
    Release mutex
    ...
}
  
```



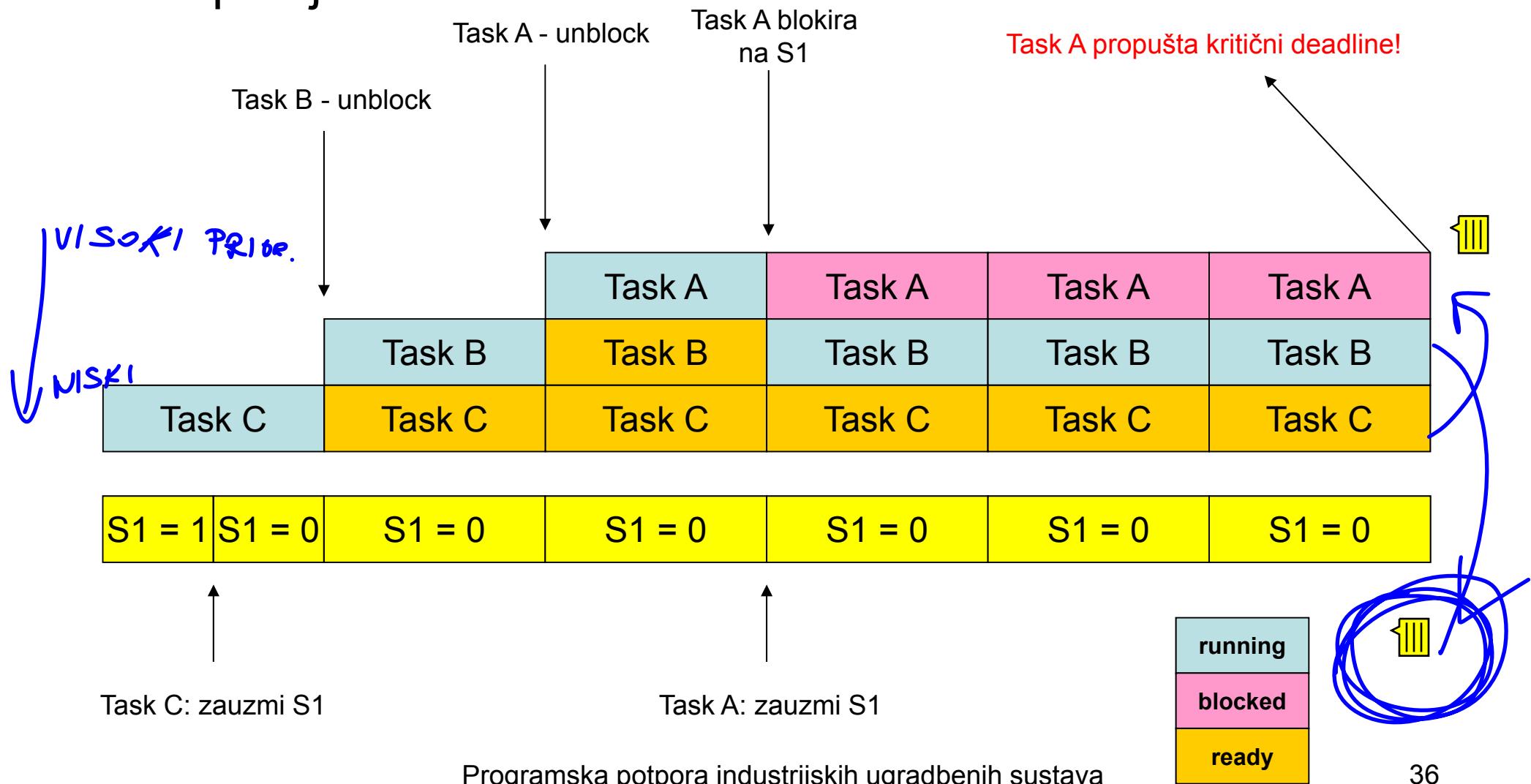
Što bi se dogodilo kada bi se koristio obični binarni semafor?

Problemi sa semaforima

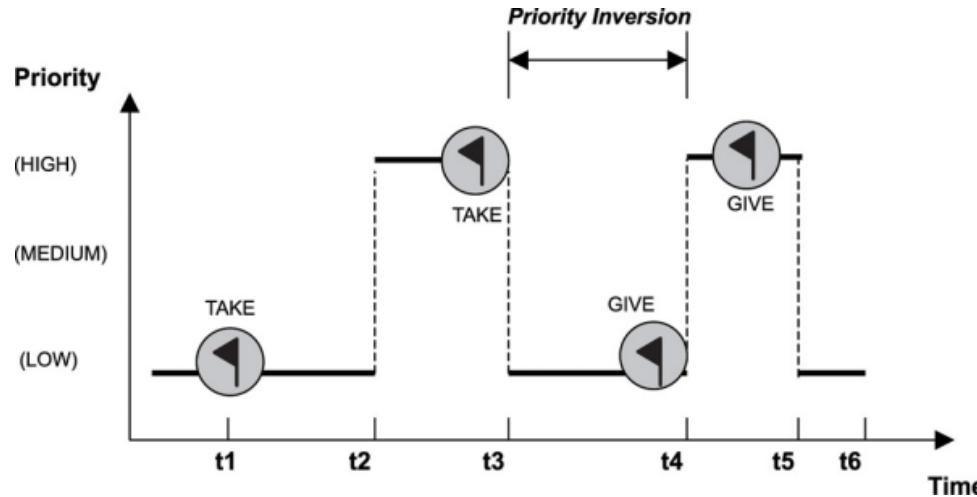
- tipične pogreške:
 - semafor nije ispravno zauzet
 - zaštita dijeljenih podataka funkcionirat će samo ako se eksplicitno i na pravom mjestu zauzme odgovarajući semafor
 - semafor nije ispravno otpušten
 - npr. zbog uvjetnog izvođenja programa, preranog izlaska iz funkcije, programerske pogreške itd.
 - zauzet pogrešan semafor
 - moguće kod sustava s više semafora
 - semafor se drži predugo
 - problem kod sustava za rad u stvarnom vremenu – ako semafor predugo blokira druge zadatke, može se dogoditi da se ne zadovolje postavljena vremenska ograničenja (*deadlines*)
 - **inverzija prioriteta (*priority inversion*)**
 - **deadlock**

Inverzija prioriteta (*Priority Inversion*)

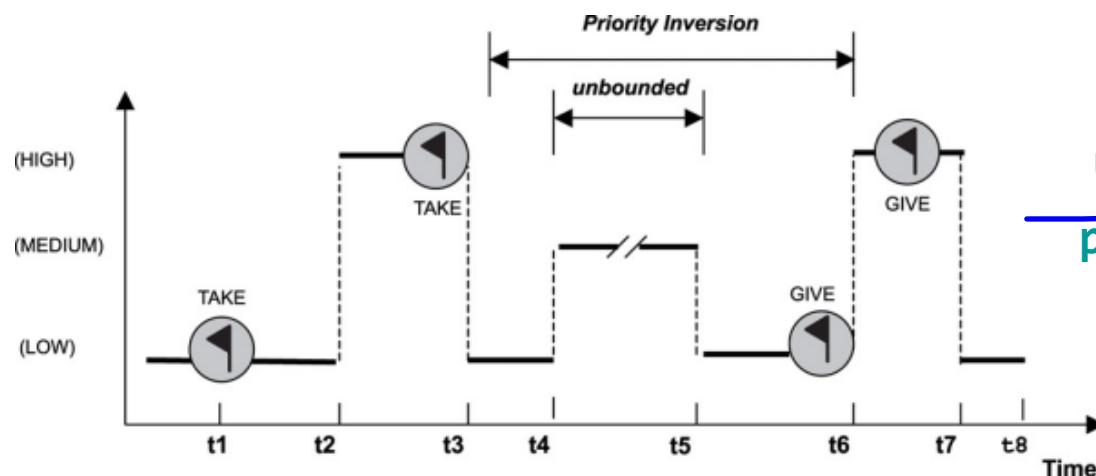
- primjer



Priority Inversion



bounded priority inversion



unbounded priority inversion
 poseban problem - medium priority task!

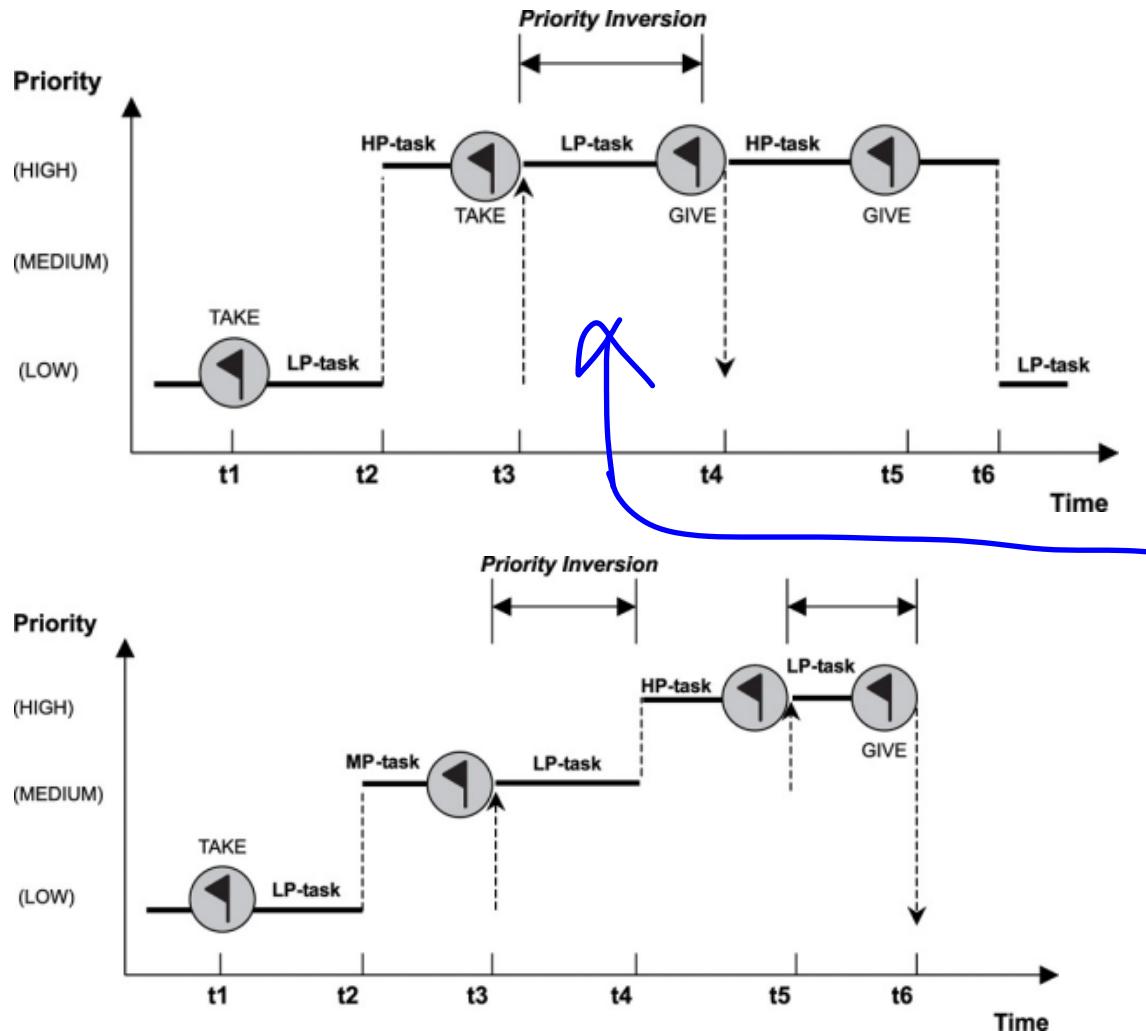
Task A
 Task B
 37

Protokol nasljeđivanja prioriteta (Priority Inheritance Protocol)



- umanjuje problem inverzije prioriteta, ali ga ne uklanja u potpunosti
- protokol:
 - neka je R dijeljeni resurs i T zadatak koji mu želi pristupiti
 - ako je R zaključan, zadatak T blokira dok se R ne oslobodi
 - ako je R slobodan, zaključava se i dodjeljuje T
 - ako zadatak T' višeg prioriteta od T (koji je prekinuo izvođenje T) zahtjeva resurs R dok ga drži T, *privremeno* (dinamički) se podiže *prioritet* zadatka T na razinu zadatka T' i zadatak T istiskuje T' dok ne završi operaciju nad resursom R
 - nakon što oslobodi R, zadatku T se vraća izvorni prioritet
- rješava *unbounded priority inversion* problem

Protokol nasljeđivanja prioriteta (Priority Inheritance Protocol)



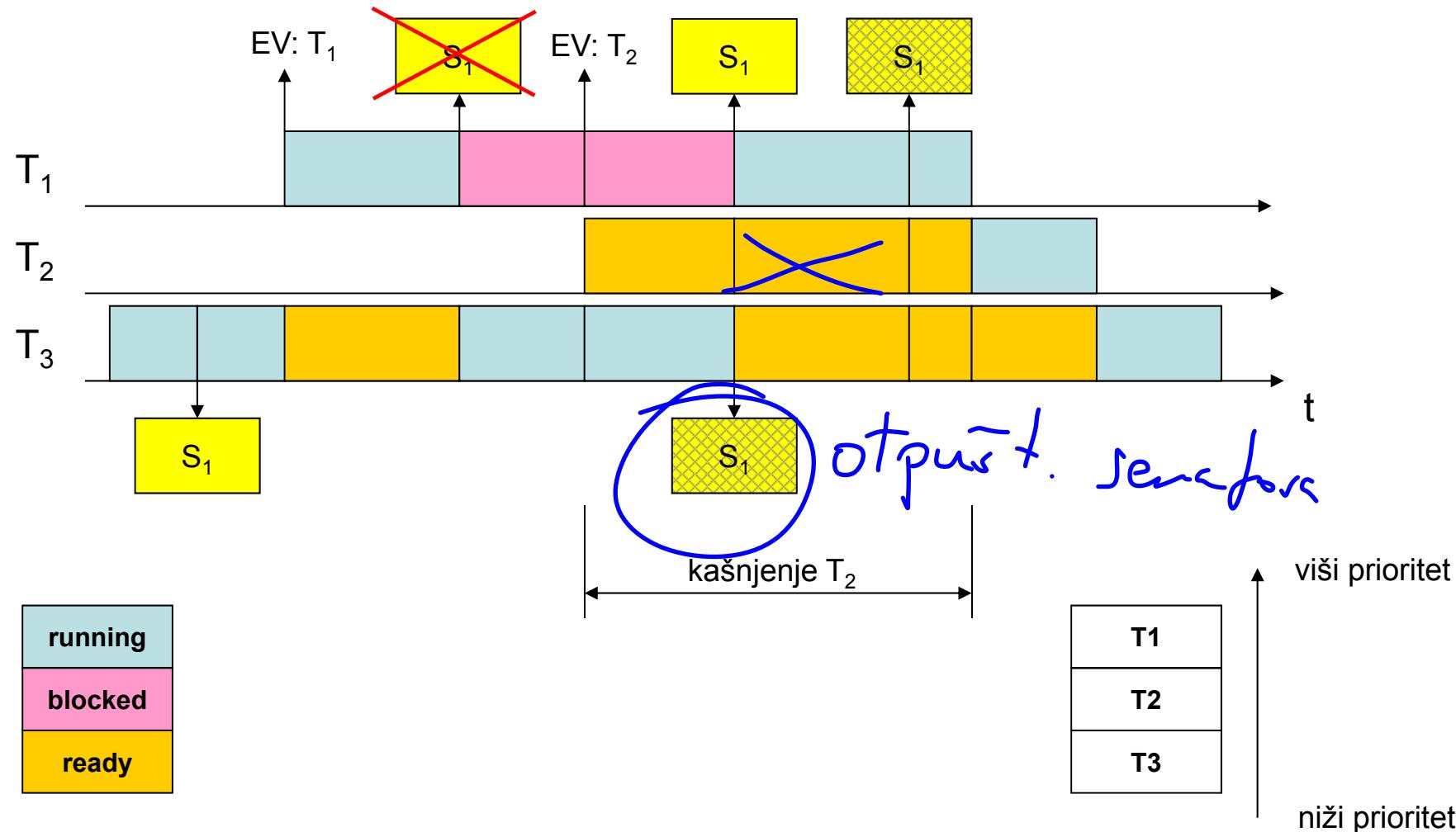
PIP za dva zadatka

Task B 
ne može prekinuti

propagiranje
nasljeđivanja
prioriteta za više
zadataka

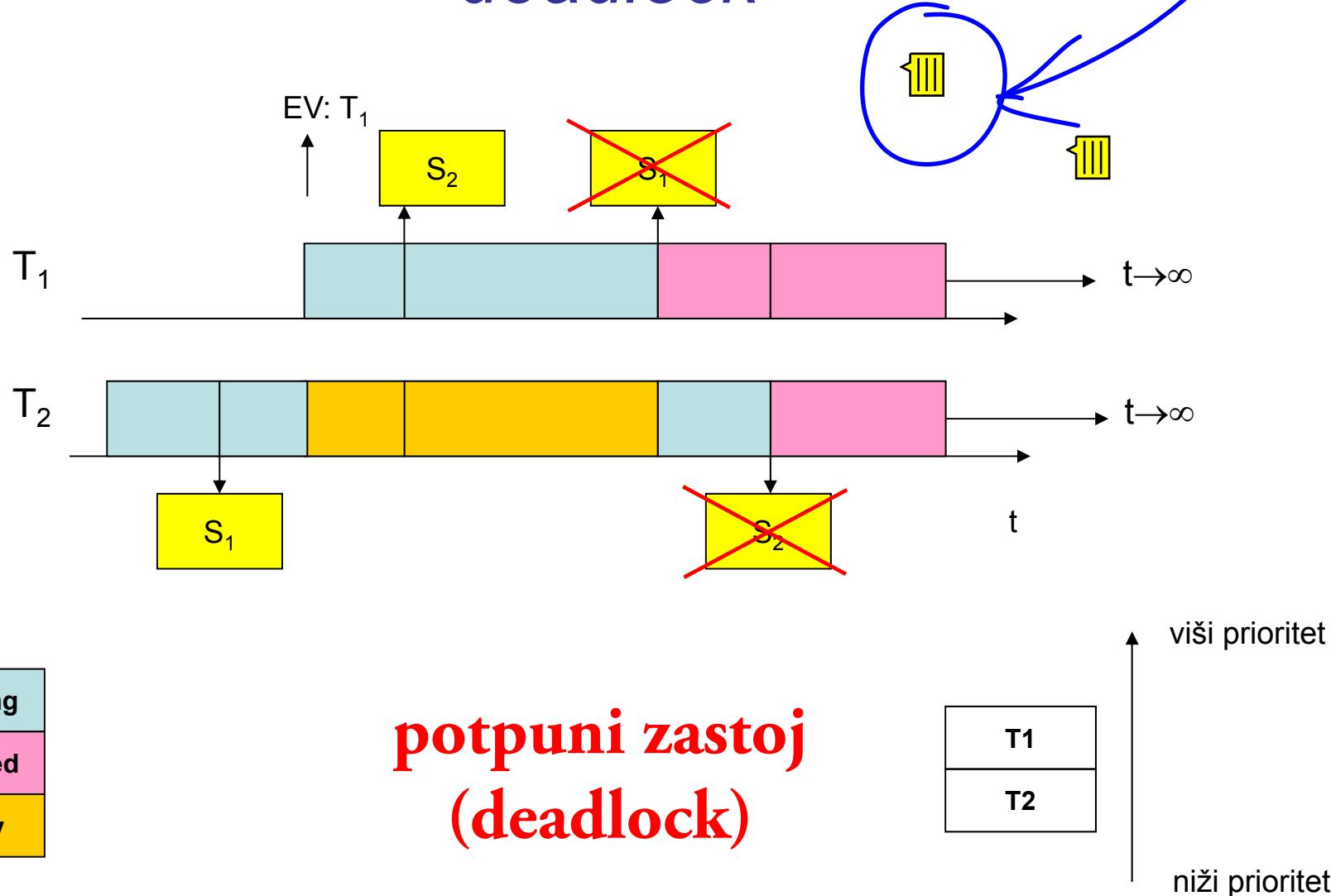
Protokol nasljeđivanja prioriteta

(Priority Inheritance Protocol)



NOVI PROBLEM

Protokol nasljeđivanja prioriteta – deadlock

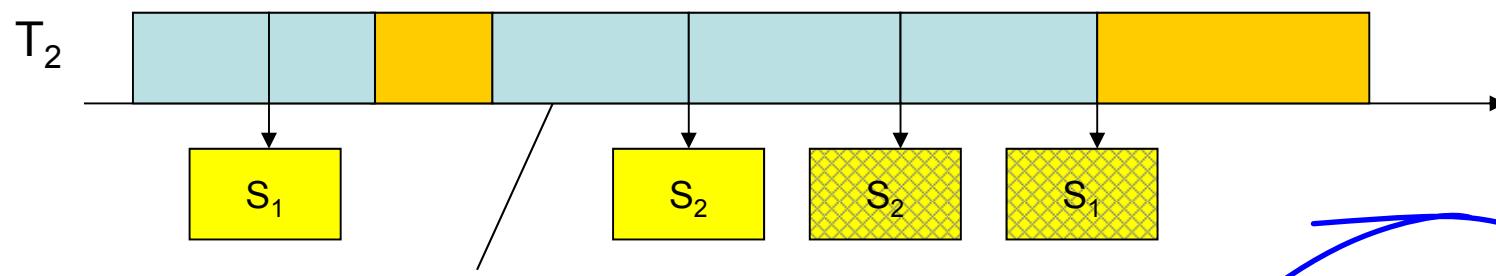
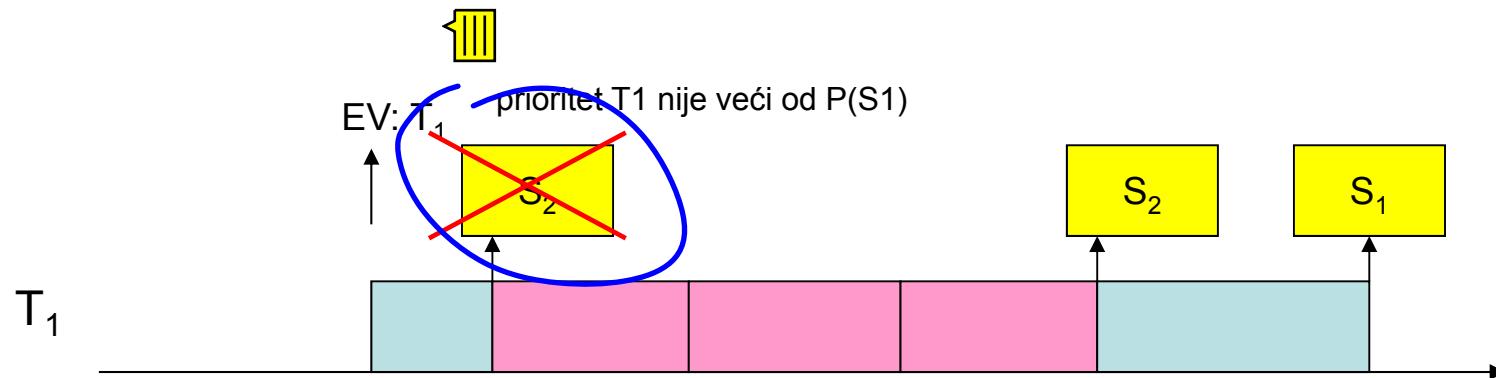


Protokol stropnog prioriteta (Priority Ceiling Protocol)

ima
nek
osnovi

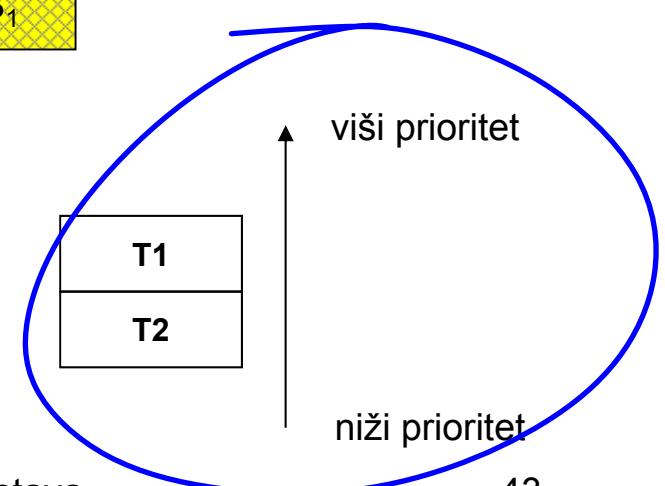
- nadogradnja *priority inheritance* protokola tako da se spriječi ulazak u kritičnu sekciju koja može biti kasnije blokirana i uzrokovati potpuni zastoj
- svakom resursu R dodjeljuje se prioritet (**stropni prioritet**) koji je jednak najvećem prioritetu zadatka koji mu može pristupiti
 - pretpostavka: *unaprijed* se zna koji zadaci mogu pristupiti kojem resursu
- zadatak T može biti dodatno blokiran na ulazu u kritičnu sekciju ako postoji neki semafor koji je trenutno zaključan i koji ima stropni prioritet veći ili jednak zadatku T

Protokol stropnog prioriteta – izbjegavanje deadlocka



running
blocked
ready

Semafor	Procesi koji mu pristupaju	Stropni prioritet
S1	T ₁ , T ₂	P(T ₁)
S2	T ₁ , T ₂	P(T ₁)



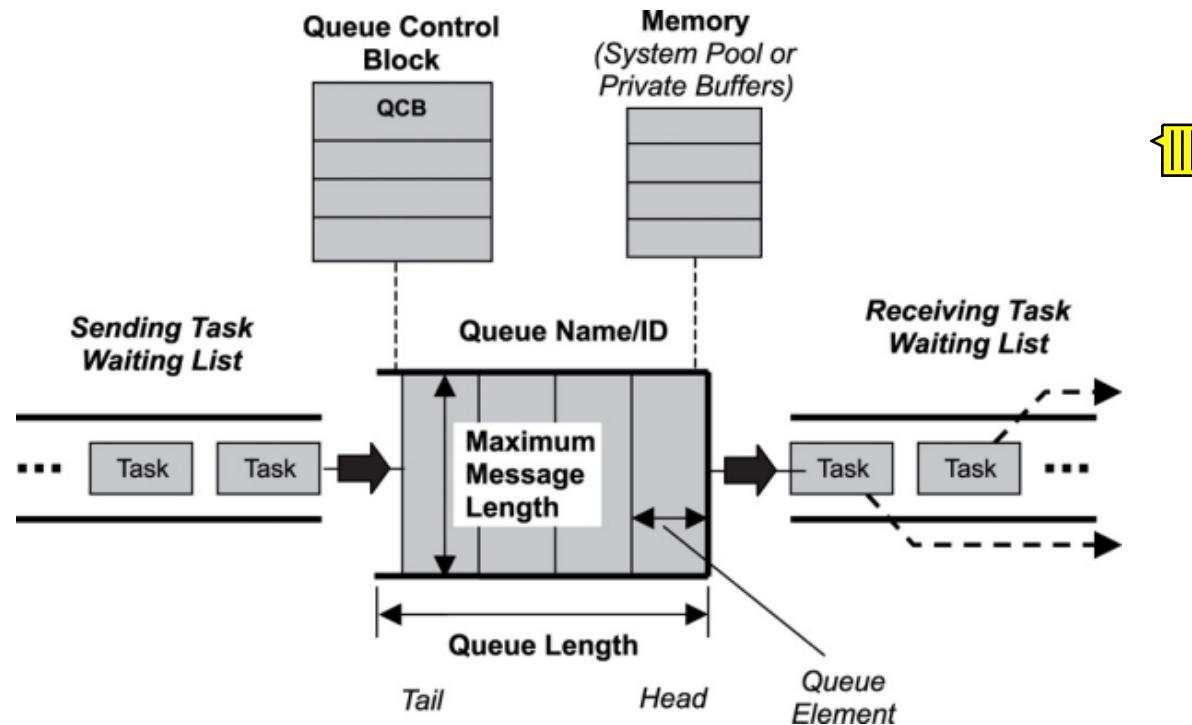
Ostali objekti jezgre RTOS-a

- redovi poruka (*message queues*)
- cjevovodi (*pipes*)
- registri događaja (*event registers*)
- signali (*signals*)
- ostali gradivni blokovi OS-a:
 - upravljanje memorijom (*memory management*),
 - mrežna potpora (*networking*),
 - datotečni sustav (*filesystem*) itd.



Redovi poruka (Message Queues)

- struktura podataka (red, *queue*) putem koje zadaci i prekidi mogu komunicirati i razmjenjivati poruke
- IPC – *Inter-Process Communication*



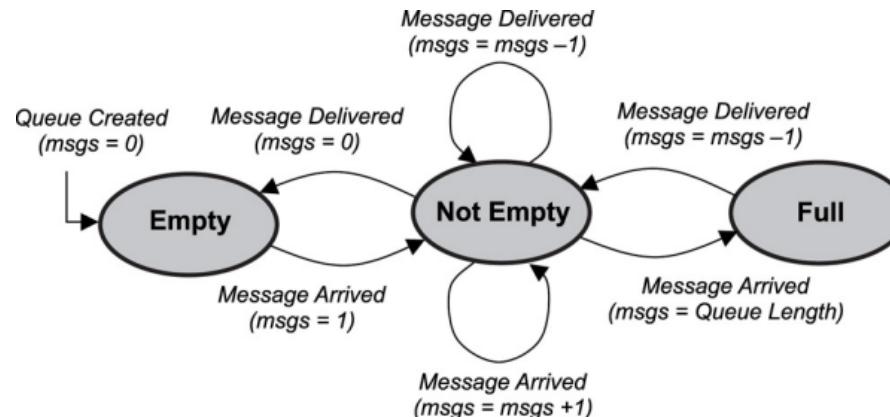
Redovi poruka (Message Queues)



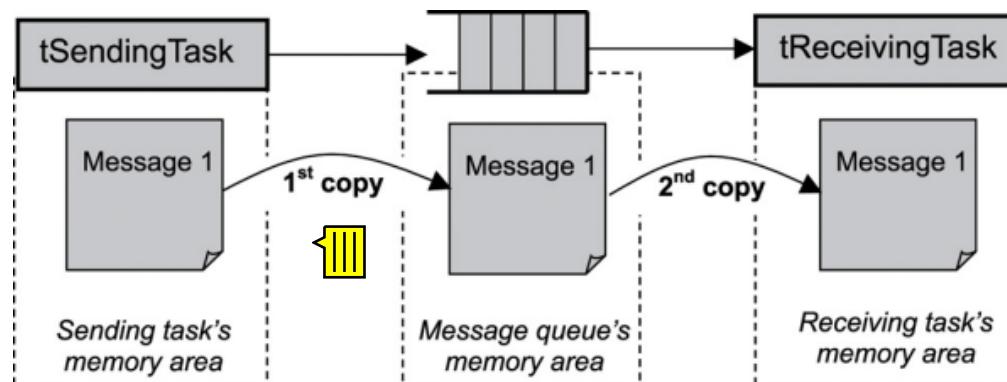
- globalni resursi dostupni svim zadacima
- prilikom kreiranja zadaje se: jedinstveni identifikator, memorijski resursi, broj elemenata, prioritet zadataka na čekanju, itd.
- liste čekanja zadataka (*task waiting lists*) – razdvojeno za slanje i primanje poruka
 - FIFO, LIFO, prioritetni red ...
- posebni slučajevi:
 - pokušaj pisanja zadatka u puni red – neblokirajući povratak iz funkcije (poruka o pogrešci) ili blokiranje zadatka
 - pokušaj čitanja praznog reda – slično kao i kod pisanja

Redovi poruka (Message Queues)

- automat s konačnim brojem stanja (FSM):



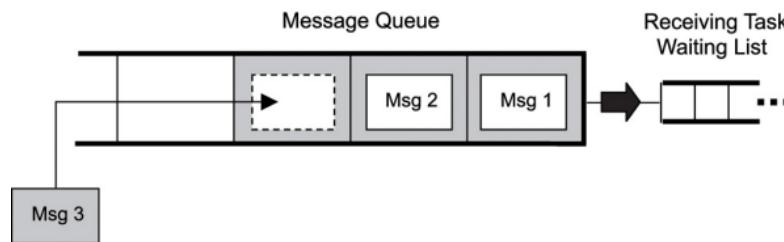
- razmjena podataka između procesa:



Programska potpora industrijskih ugradbenih sustava

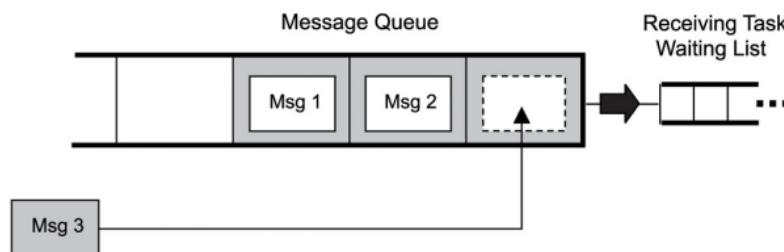
Slanje poruka

Sending Messages – First-In, First-Out (FIFO) Order



FIFO – tipična implementacija reda poruka

Sending Messages – Last-In, First-Out (LIFO) Order

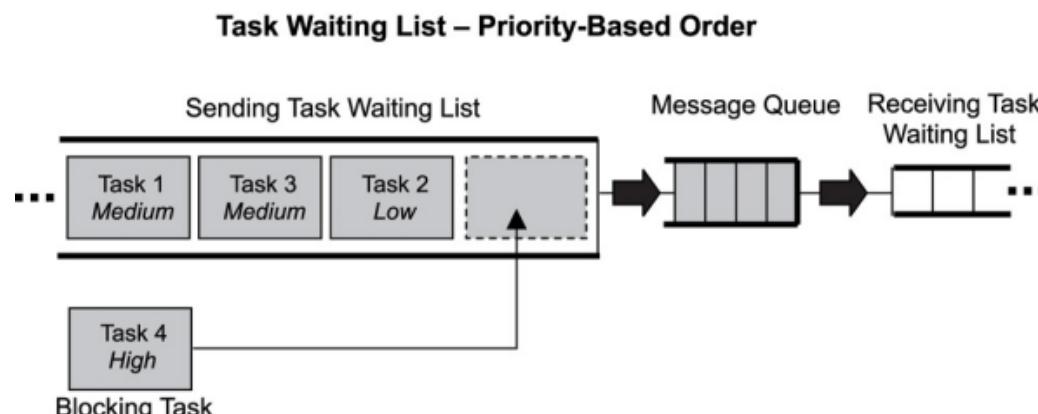
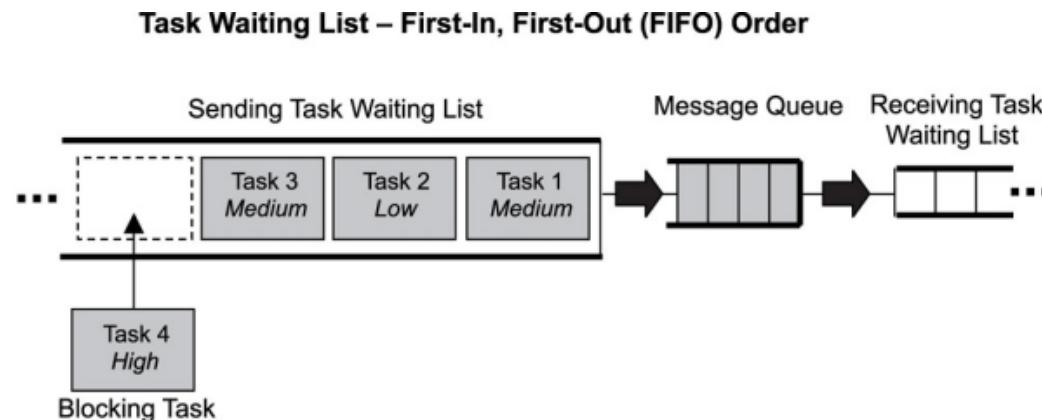


LIFO – npr. kada je uvijek zadnja poruka najprioritetnija (može se koristiti i prioritetni red)

- slanje poruka:
 - neblokirajuće (ISR, task) 
 - blokirajuće s timeoutom (task)
 - blokirajuće (task) 

Slanje poruka

- blokiranje zadataka u slučaju popunjenošću reda:



Primanje poruka

- primanje poruka:
 - neblokirajuće
 - blokirajuće s timeoutom
 - blokirajuće
- slične mogućnosti zadavanja prioriteta zadataka u listi zadataka na čekanju poruke kao i u slučaju slanja
- čitanje poruke
 - destruktivno (tipična realizacija) *(pop)* *def*
 - nedestruktivno (nije podržano kod svih OS-ova) *(peek)*

Redovi poruka (Message Queues)

- tipični načini upotrebe redova poruka:
 - jednosmjerno slanje bez rukovanja (*non-interlocked one-way data communication*)
 - jednosmjerno slanje s rukovanjem (*interlocked one-way data communication*)
 - dvosmjerna komunikacija (*two-way data communication*)
 - *broadcast* komunikacija

Redovi poruka (Message Queues)

- jednosmjerno slanje bez rukovanja (*non-interlocked one-way data communication*)

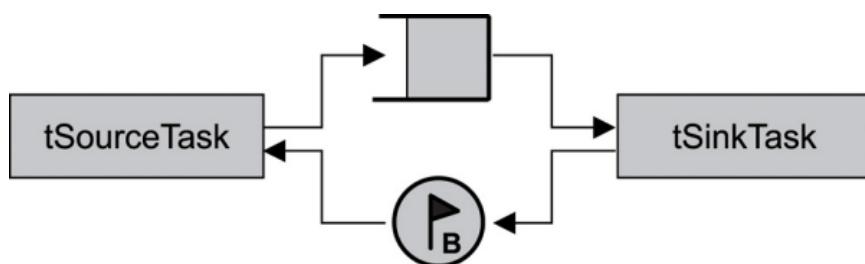


```
void vSourceTask() {  
    ...  
    pošalji poruku u red // blokiraj ako je red pun  
    ...  
}  
void vSinkTask() {  
    ...  
    primi poruku iz reda // blokiraj ako je red prazan  
    ...  
}
```

- tipična primjena – slanje podataka iz ISR-a u glavni program
 - pažnja: unutar ISR-a ne smije se koristiti blokiranje!

Redovi poruka (Message Queues)

- jednosmjerno slanje s rukovanjem (*interlocked one-way data communication*)



semafor je inicialno neprolazan (Queue.S1=0)

struktura se zove još i *mailbox*

```

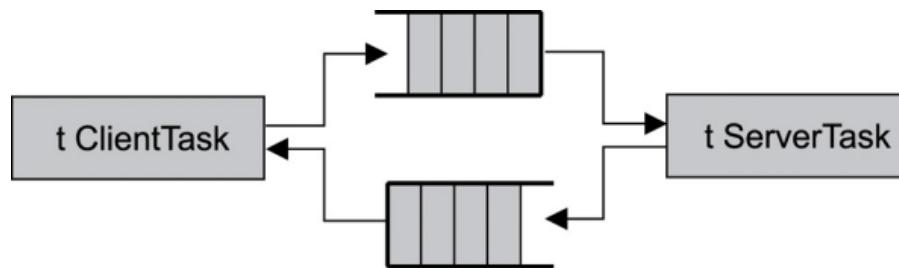
void vSourceTask() {
    ...
    pošalji poruku u red // blokiraj ako je red pun
    1) TakeSemaphore(Queue.S1);
    ...
}
void vSinkTask() {
    ...
    primi poruku iz reda // blokiraj ako je red prazan
    2) ReleaseSemaphore(Queue.S1);
    ...
}
  
```

*rucza preko
semafora*

- komunikacija s potvrdom primitka poruke – pouzdaniji IPC

Redovi poruka (Message Queues)

- dvosmjerna komunikacija (*two-way data communication*)



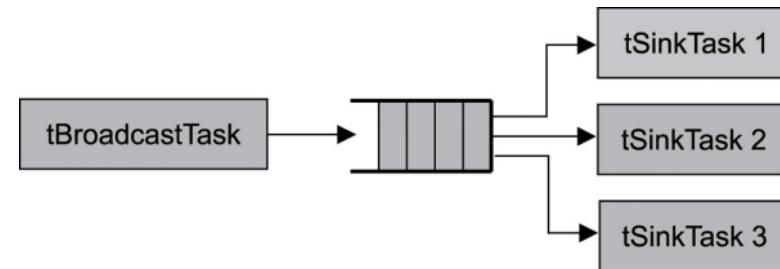
```
void vClientTask() {  
    ...  
    pošalji poruke u red zahtjeva  
    čekaj poruke iz reda odgovora  
    ...  
}  
void vServerTask() {  
    ...  
    primi poruku iz reda zahtjeva  
    pošalji poruke u red odgovora  
    ...  
}
```

- tipični primjer – *client/server* arhitektura

Redovi poruka (Message Queues)

- *broadcast* komunikacija:

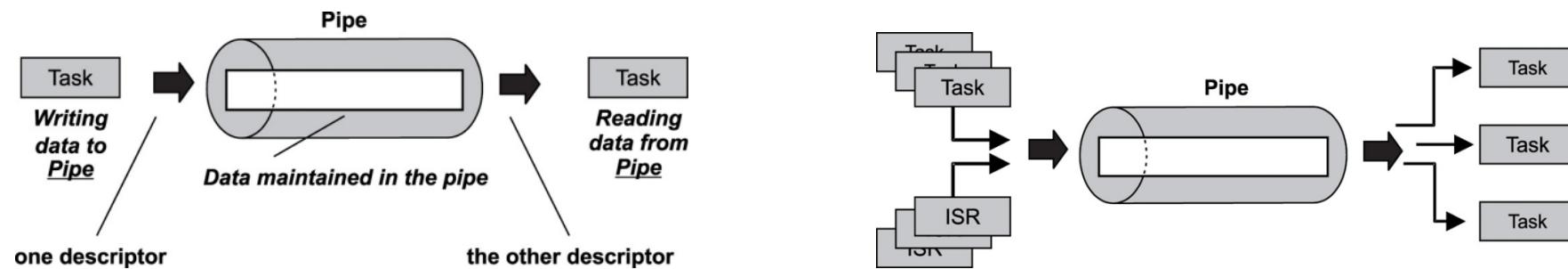
```
void vBroadcastTask() {  
    ...  
    pošalji poruku u broadcast red  
    ...  
}  
void vSinkTask1() {  
    ...  
    primi poruku iz broadcast reda  
    ...  
}  
void vSinkTask2() {  
    ...  
    primi poruku iz broadcast reda  
    ...  
}
```



- ne podržavaju svi RTOS-ovi redove poruka s mogućnošću *broadcasta*

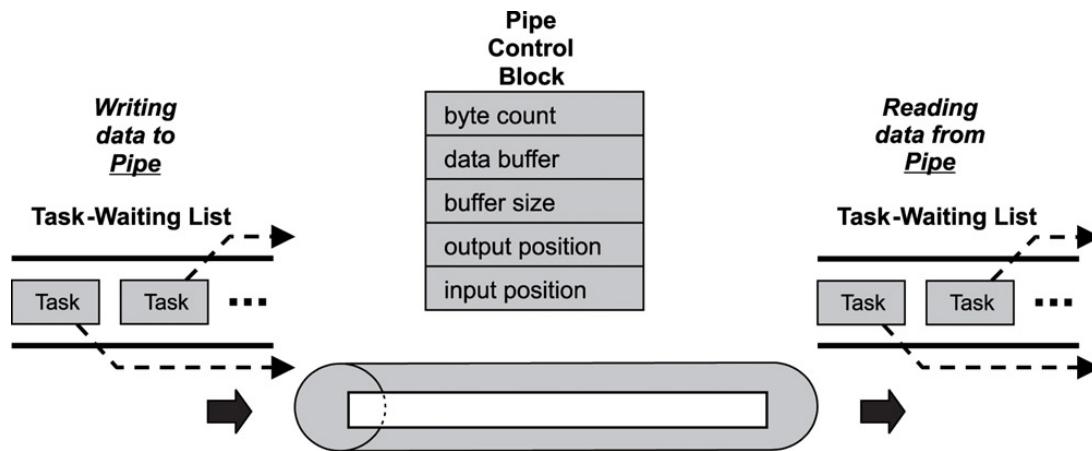
Cjevovodi (Pipes)

- objekti jezgre OS-a koji omogućuju *nestrukturiranu razmjenu podataka između procesa (FIFO byte-stream)*

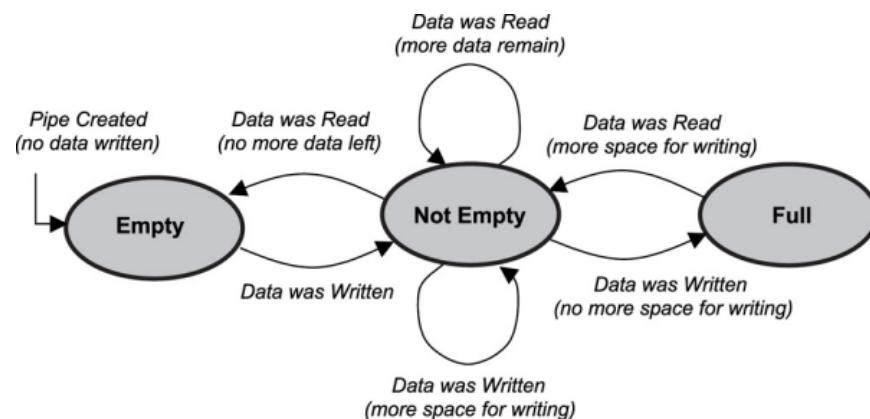


- procesi mogu biti blokirani kod pisanja u puni cjevovod i kod čitanja praznog cjevovoda

Cjevovodi (*Pipes*)

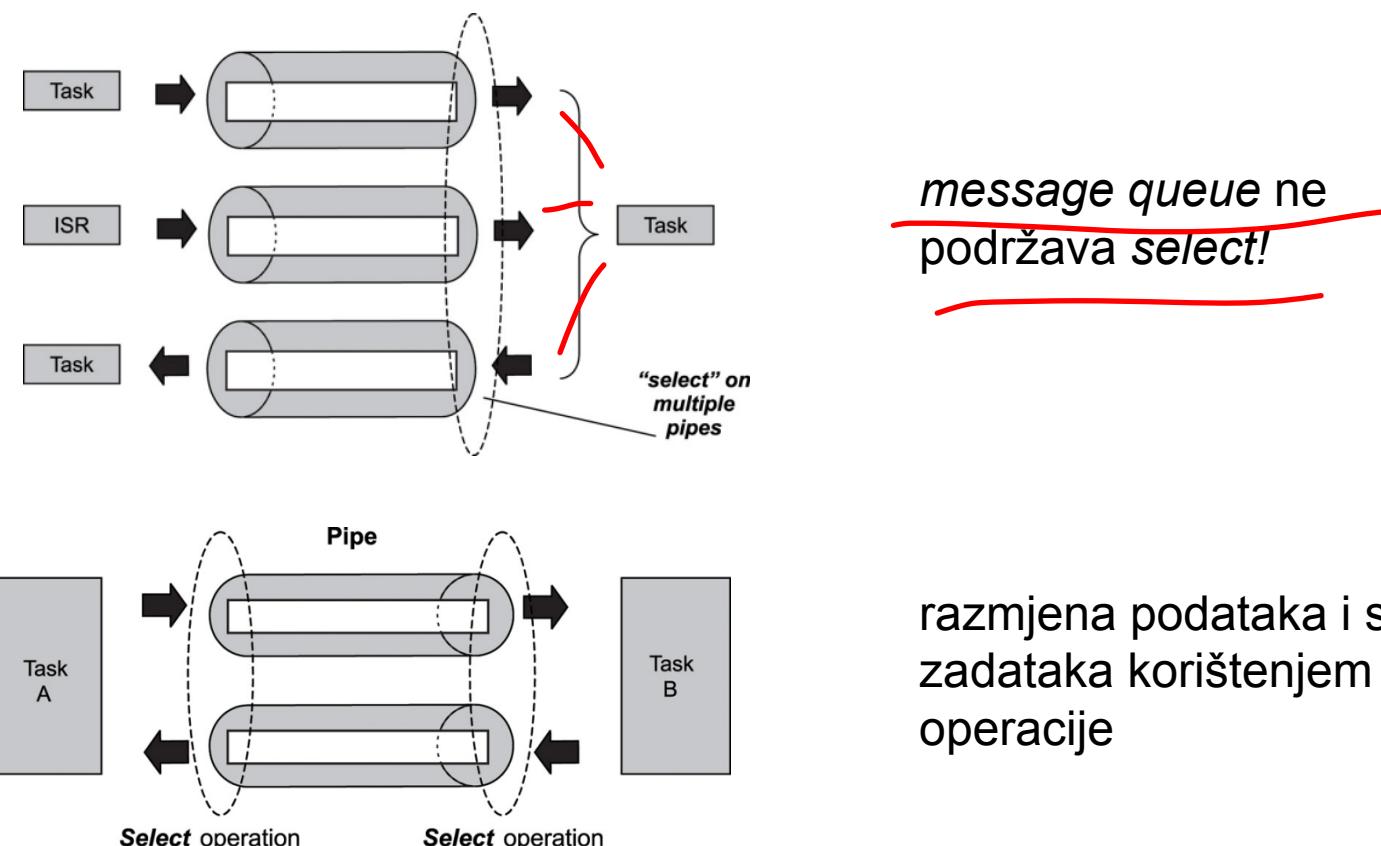


struktura *pipe*
OS objekta



Cjevovodi (Pipes)

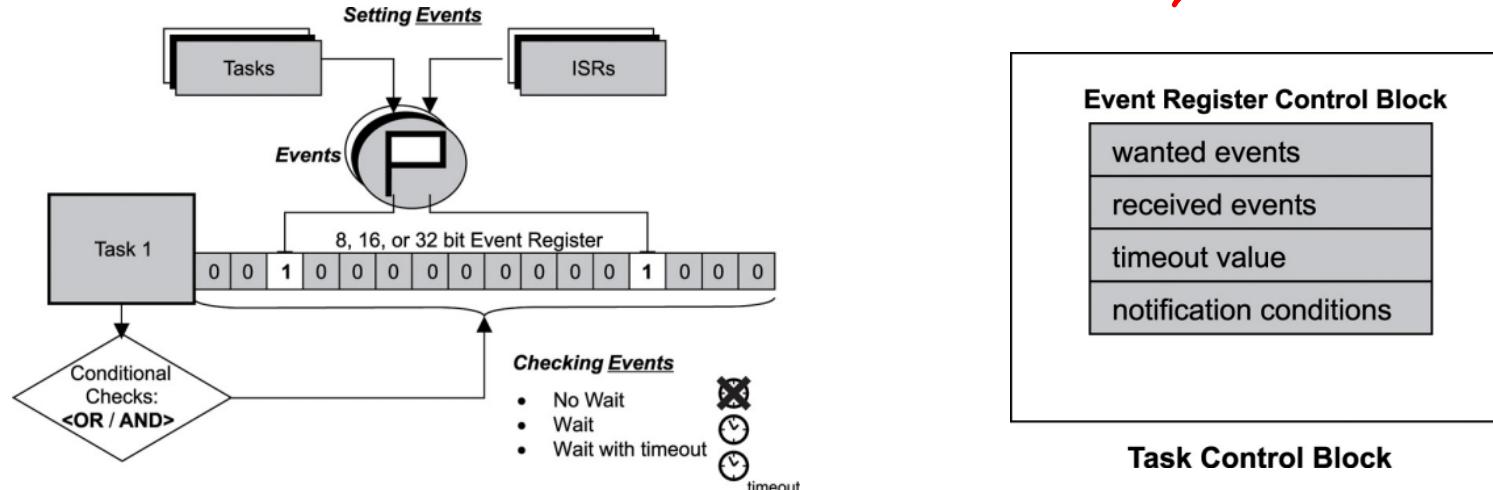
- *select* operacija: proces je blokiran dok se na barem jednom cjevovodi ne pojave podaci



razmjena podataka i sinkronizacija zadataka korištenjem *select pipe* operacije

Registri događaja (*Event Registers*)

- neki OS-ovi imaju posebne *event registre* kao dio TCB-a

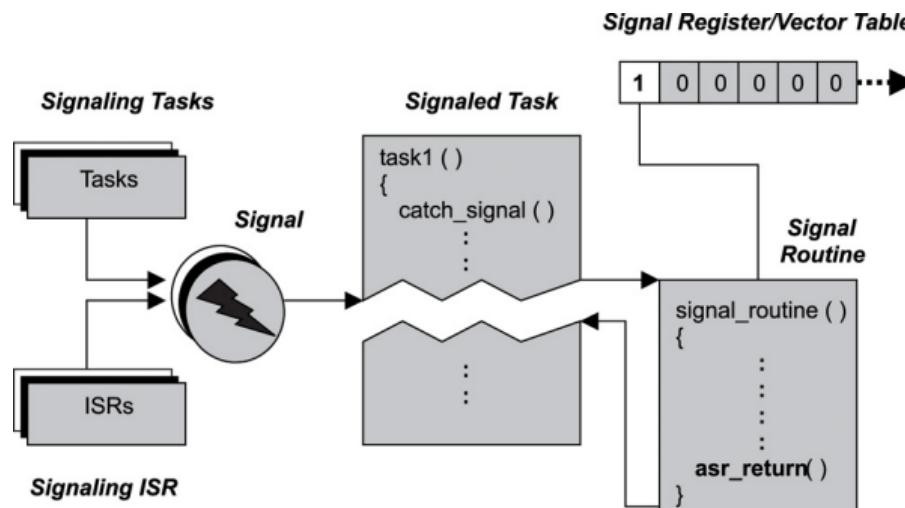


- maskiranje zastavica, logički uvjeti nad zastavicama
- služe za jednostavnu sinkronizaciju (čekanje jednog zadatka na neki događaj)
- niža fleksibilnost u odnosu na sinkronizaciju semaforima
 - nema podataka povezanih s registrom događaja, nemogućnost određivanja uzroka događaja, istovrsni događaji se ne mogu akumulirati, ograničeni broj bitova itd.

Usp. sa
semaf.

Signali (Signals)

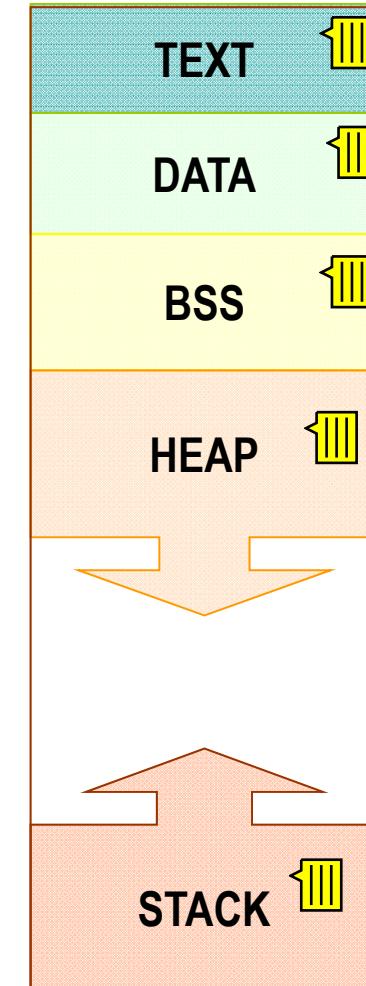
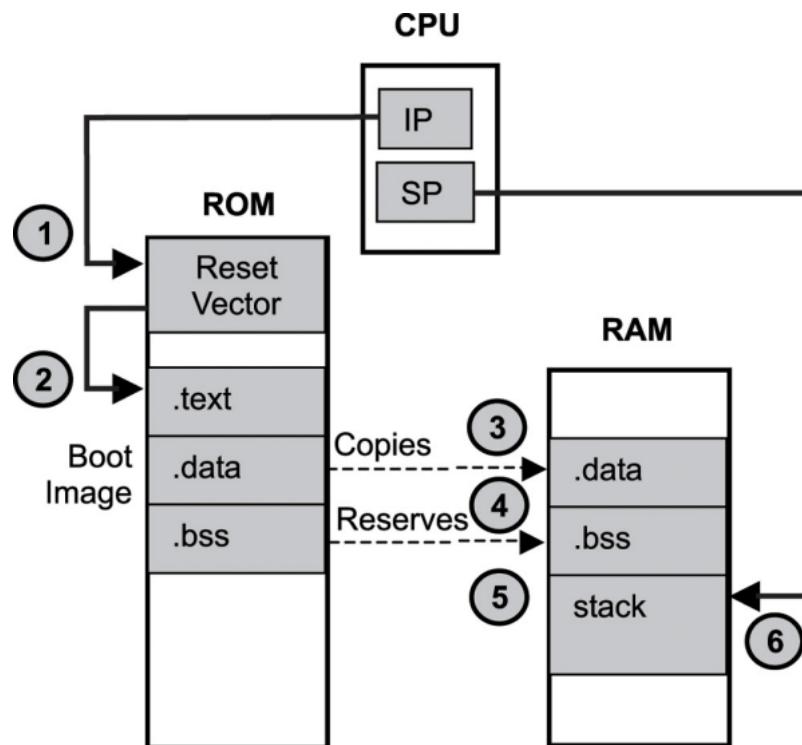
- *signal* u kontekstu RTOS-a predstavlja *programski prekid* (*software interrupt*) koji može prekinuti izvođenje zadatka u bilo kojem trenutku i preusmjeriti tijek programa u SSR (*signal service routine*)
- vrlo slično ISR-u, ali je okidač isključivo *softverski* (npr. ~~promjena stanja u registru događaja, ilegalna instrukcija i sl.~~)





Upravljanje memorijom

- dinamičko upravljanje memorijom kod sustava s ograničenim sklopovskim resursima – *heap* (funkcija *malloc*)



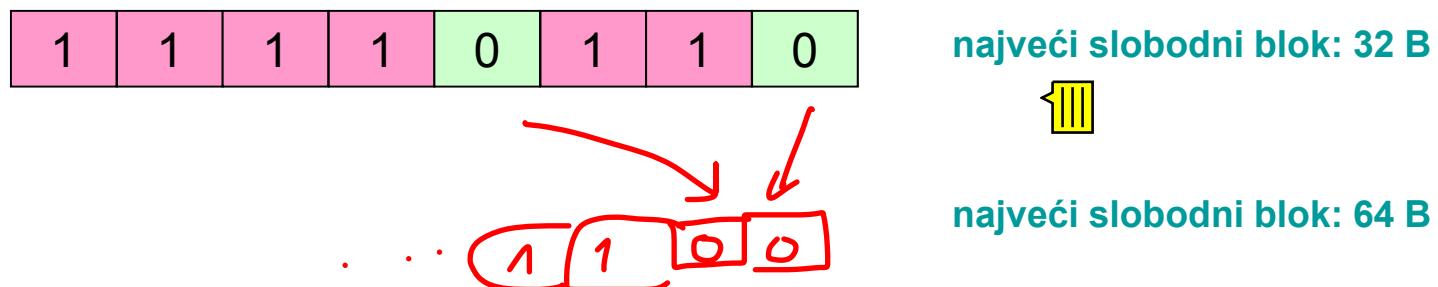
Upravljanje memorijom

- primjer: fragmentacija memorije kod *heap-a* s blokovima fiksne veličine ($B=32$)



Upravljanje memorijom

- algoritam: funkcija *malloc()* uvijek traži najveći kontinuirani blok slobodne memorije za alokaciju
- problem – *fragmentacija memorije* ograničava maksimalnu veličinu bloka koja se može dinamički alocirati u određenom trenutku, a ne ukupni zbroj veličina raspoloživih blokova na heapu!
- kompaktiranje memorije (*memory compaction*) – preslagivanje (kopiranje) alociranih memorijskih blokova tako da se dobije kontinuirani blok slobodne memorije maksimalnog kapaciteta
 - vremenski zahtjevan nedeterministički proces, neprihvativ za URS za rad u stvarnom vremenu (koristi se kod sustava s virtualnom memorijom, gdje nije potrebno kopirati sadržaj memorije, već samo indekse stranica)

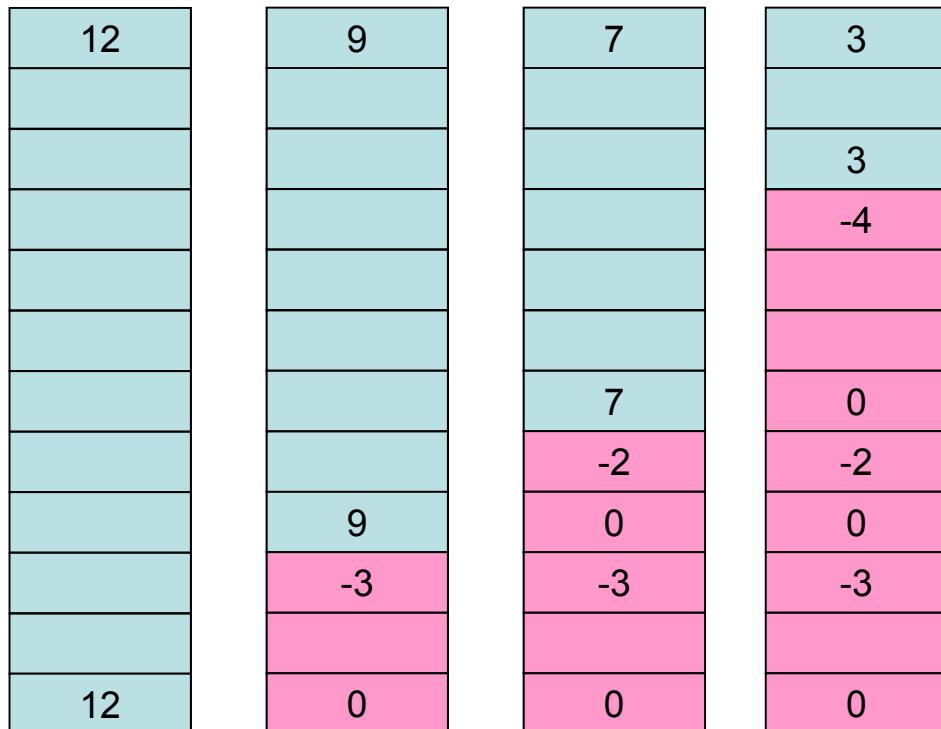


Dinamička alokacija memorije

Funkcija malloc():

vjeroj zad za mi

Alokacijska tablica

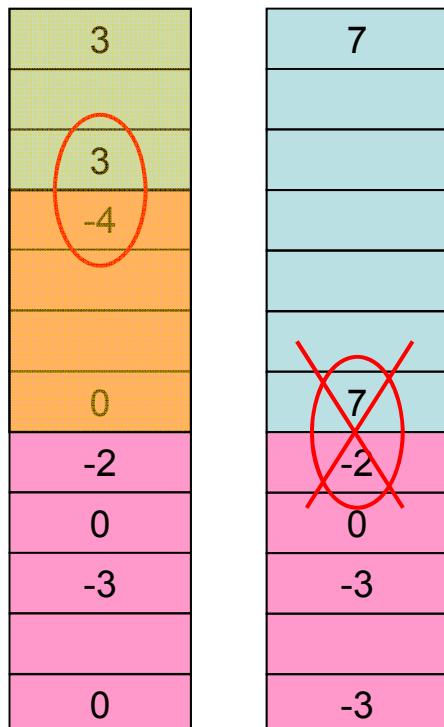


- početak i kraj slobodnog bloka sadrže veličinu bloka

- početak zauzetog bloka sadrži negativnu vrijednost veličine bloka, a kraj vrijednost 0

Dinamička alokacija memorije

Funkcija free() – minimizacija vanjske fragmentacije memorije:

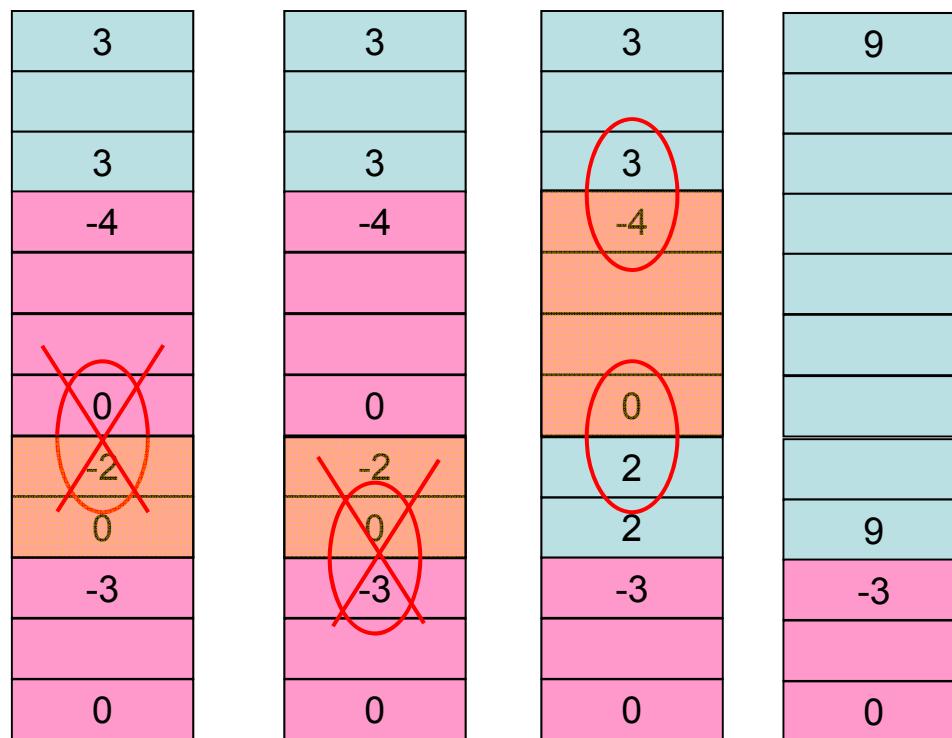


- ako je kraj prethodnog bloka > 0 , novooslobođeni blok može se spojiti s prethodnim

- ako je početak sljedećeg bloka > 0 , novooslobođeni blok može se spojiti sa sljedećim

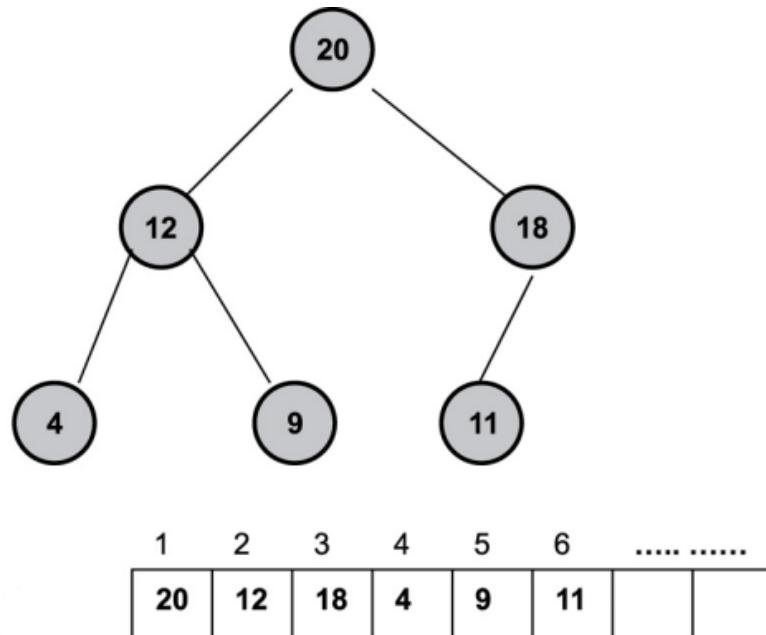
Dinamička alokacija memorije

Funkcija free() – minimizacija vanjske fragmentacije memorije:



Dinamička alokacija memorije

- kako brzo pronaći najveći blok?
 - koristi se struktura podataka *heap* implementirana statičkim poljem (otuda naziv *heap* za dinamički memorijski segment):



The left child of a node k is at position $2k$.
 The right child of a node k is at position $2k+1$.

učinkovitiji prikaz alokacijske tablice, jer
 bi niz trebalo linearno pretraživati –
 loše performanse!

Dodavanje novog bloka u *heap*?

O(log N)

**Dohvat najvećeg slobodnog bloka
 iz *heap*-a?**

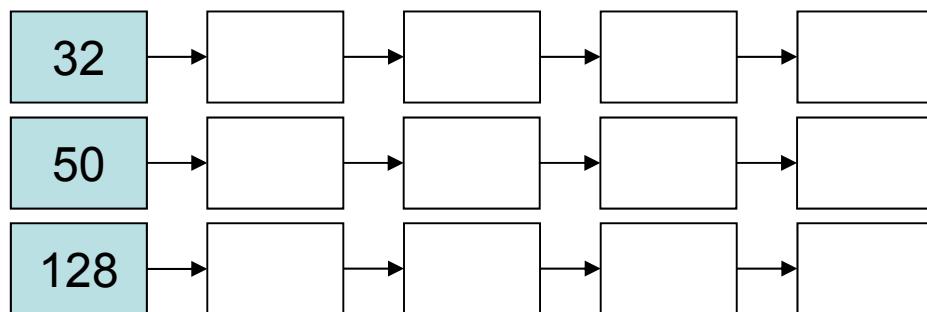
O(1)

**Osvježavanje stanja slobodnih
 blokova na *heap*-u nakon alokacije?**

O(log N)

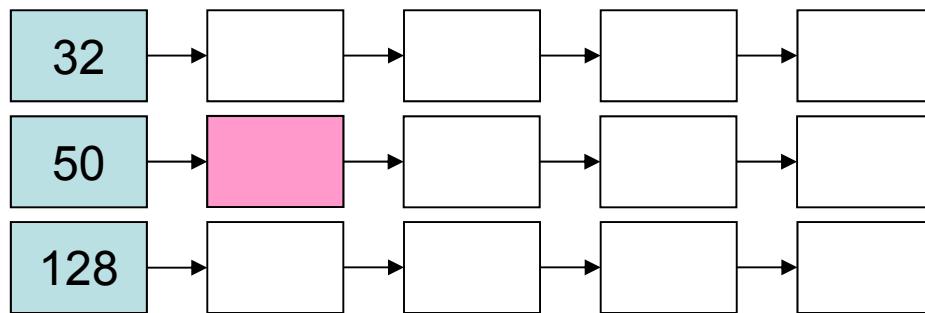
Dinamička alokacija memorije korištenjem fiksnih međuspremnika

- dinamička alokacija memorije pomoću *heap-a* može biti problematična kod RT-URS zbog nedeterminističkog ponašanja algoritma i potrebe za kompaktiranjem memorije nakon određenog broja alokacija
- alternativa: *fixed-sized memory management*
 - korištenje više *memory pool-ova* s blokovima fiksne veličine ovisno o trenutnoj potrebi

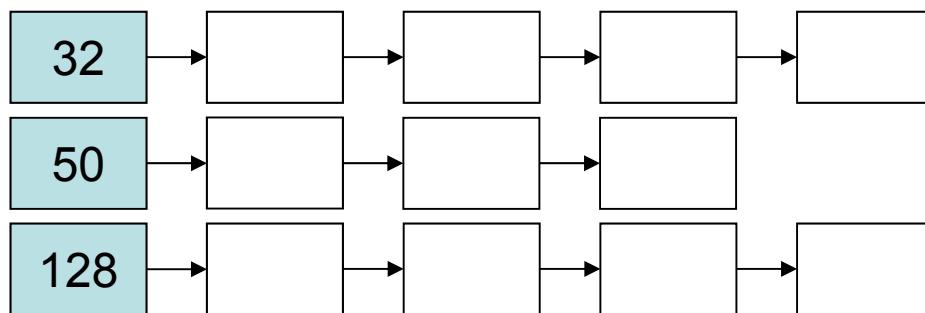


Primjer: tri *memory pool-a* s blokovima veličine 32, 50 i 128 bajtova

Dinamička alokacija memorije korištenjem fiksnih međuspremnika



alociraj 48 okteta



interna fragmentacija



alocirano

alociraj 256 okteta?

Dinamička alokacija memorije korištenjem fiksnih međuspremnika

- prednosti:
 - determinističko ponašanje algoritma
 - brzina
 - jednostavnost: koriste se jednostruko povezane liste
 - nema potrebe za kompaktiranjem memorije
- nedostaci:
 - nefleksibilnost – u općem slučaju je nemoguće unaprijed znati kolike su sve potrebne veličine blokova i koliko će ih u nekom trenutku najviše trebati
 - neučinkovitost – neki *pool*-ovi mogu biti preopterećeni, neki podopterećeni
- pogodan za jednostavnije aplikacije kod kojih se unaprijed dobro mogu prepostaviti potrebe za dinamičkom memorijom

Virtualna memorija

- omogućuje procesu da cijeli adresni prostor vidi kao da je dodijeljen samo njemu
- translacija logičkih u fizičke adrese – MMU
- prednosti: jednostavniji programski model, neovisnost izvođenja programa o stvarnoj veličini fizičke memorije (*back-storage swap*), kompaktiranje stranica bez potrebe za kopiranjem sadržaja memorije, zaštita memorije (na razini procesa i između više procesa) itd.
- nedostaci: zahtijeva složenije sklopolje (MMU), uglavnom nije podržana kod RTOS ...

RTOS - buy vs build?

- najčešće je optimalno rješenje da se za razvoj proizvoda koristi gotovi komercijalni RTOS:
 - pouzdanost i robusnost, jednostavna implementacija aplikacijske programske potpore, prenosivost kôda, brzi razvoj sustava (*time-to-market*) itd.
 - razvoj vlastitog RTOS-a je vrlo složen i zahtjevan proces koji se izbjegava kada je god moguće
- nedostaci gotovih rješenja: brzina i memorijski zahtjevi (zbog overhead-a TCB modela), otpornost na pogreške (*fault tolerance*), nemogućnost izbacivanja nepotrebnih mogućnosti, troškovi licenciranja po proizvedenom komadu, itd.

Kriteriji odabira RTOS-a

- Neki od kriterija koje je potrebno razmotriti prilikom odabira gotovog rješenja:
 - minimalna latencija prekida
 - vrijeme zamjene konteksta
 - broj podržanih zadataka
 - mehanizam raspoređivanja zadataka (*preemptive, non-preemptive*)
 - komunikacija između zadataka (*message queues, pipelines* itd.)
 - memorijski zahtjevi (ROM, RAM, vanjska memorija)
 - podržane sklopovske platforme
 - podržani mrežni protokoli
 - dostupnog izvornog koda (i komercijalni RTOS-ovi imaju tu mogućnost)
 - cijena (fiksna, po razvojnom mjestu, *royalty free* - da li se plaćaju licencni troškovi za svaki komad ...)
 - korisnička potpora

itd.

Primjer specifikacije RTOS-a

- primjer: Micrium µC/OS-III™

Who should use this RTOS?	Developers who want to save time on their current and future embedded system projects, and who require a robust RTOS built on clean, easy-to-implement code.
Supported architectures	ARM7/9, Cortex-MX, Nios-II, PowerPC, Coldfire, i.MX, Microblaze, RX600, H8, SH, M16C, M32C, Blackfin, ...
Maximum ROM Footprint (Unscaled)	24 Kbytes
Minimum ROM Footprint (Scaled)	6 Kbytes
Number of Kernel Services	10 different using 80 API calls
Multitasking Model	Preemptive
Code Execution Entities	Tasks, ISRs
Dynamic Objects	Static and Dynamic
Data Movement	Message Queues (unlimited)
Semaphores - Full Counting	Yes (unlimited)
Mutexes - With Priority Inheritance	Yes (priority ceiling)
Event Flags	Yes (unlimited), configurable for 8, 16, or 32 bits
Memory Partitions - RAM Management	Yes (unlimited)
Timers	Yes (unlimited)
Number of tasks	Unlimited
Interrupt Disable Time	Near Zero

<http://micrium.com/page/products/rtos/os-iii>

Neki komercijalni i besplatni RTOS-ovi

RTOS	Tip licence	Platforme
Contiki	BSD	MSP430, AVR
eCos/eCosPro	modified GNU GPL/eCosPro License	ARM/XScale, CalmRISC, 68000/Coldfire, fr30, FR-V, H8, IA32, MIPS, MN10300, OpenRISC, PowerPC, SPARC, SuperH, V8xx
FreeRTOS/ OpenRTOS	modified GNU GPL/proprietary	ARM, AVR, AVR32, Freescale ColdFire, HCS12, IA32, MicroBlaze, MSP430, PIC, Renesas H8/S, 8052, STM32
Nucleus OS	proprietary	AMD Au1100, ARM, Atmel AT91 series, Altera Nios II, Freescale iMX, Freescale MCF, Freescale MPC, Marvell PXA series, MTI, NEC uPD6111x, Sharp LH7 series, ST, TI OMAP, TI TMS320 series, Xilinx Microblaze
QNX Neutrino	proprietary	ARM, MIPS, PowerPC, SH-4, x86
RTLinux	GNU GPL	same as Linux
ThreadX	proprietary	ARC, ARM/Thumb, AVR32, BlackFin, ColdFire/68K, H8/300H, Luminary Micro Stellaris, M-CORE, MicroBlaze, PIC24/dsPIC, PIC32, MIPS, V8xx, Nios II, PowerPC, SH, SHARC, StarCore, STM32, StrongARM, TMS320C54x, TMS320C6x, x86/x386, XScale, Xtensa/Diamond, ZSP
µC/OS-II, µC/OS-III	proprietary	ARM7/9/11/Cortex M1/3, AVR, HC11/12/S12, Coldfire, Blackfin, Microblaze, NIOS, 8051, x86, Win32, H8S, M16C, M32C, MIPS, 68000, PIC24/dsPIC33/PIC32, MSP430, PowerPC, SH, StarCore, STM32, ...
VxWorks	proprietary	ARM, 68K/CPU32, ColdFire, MIPS, PowerPC, SH, SPARC, x86/Pentium/IA-3, XScale

Literatura

- P. A. Laplante: Real-Time Systems Design and Analysis, John Wiley & Sons, 2004.
- Q. Li, C. Yao: Real-Time Concepts for Embedded Systems, CMP Books, 2003.
- D. E. Simon: An Embedded Software Primer, Addison Wesley, 1999.

Fakultet elektrotehnike i računarstva
Zavod za elektroničke sustave i obradbu informacija

Programska potpora industrijskih ugradbenih sustava

Operacijski sustav FreeRTOS

Hrvoje Džapo, Mario Cifrek

ak. god. 2014./2015.

Operacijski sustav FreeRTOS

- operacijski sustav za rad u stvarnom vremenu otvorenog koda namijenjen ugradbenim računalnim sustavima s ograničenim resursima
- stranica projekta: www.freertos.org
- prednosti FreeRTOS-a:
 - malen, kompaktan i prenosiv RTOS,
 - jednostavan za korištenje,
 - programiranje u C-u, funkcionalnost osnovnog koda jezgre sadržana u svega tri C datoteke,
 - veliki broj gotovih primjera koji olakšavaju učenje,
 - tipična veličina izvršnog kôda jezgre 4kB do 9kB,
 - velika i aktivna razvojna i korisnička zajednica,
 - mogućnost besplatnog korištenja OS-a u komercijalne svrhe itd.

Osnovne mogućnosti

- načini rada jezgre - *preemptive, cooperative & hybrid*,
- podržava 30-tak procesorskih arhitektura,
- podržava redove, binarne i brojeće semafore, mutexe (s nasljeđivanjem prioriteta), programske timere, detekciju preljeva stoga itd.
- podržava Cortex-M MPU,
- neograničeni broj zadataka i razina prioriteta,
- više zadataka može imati isti prioritet,
- *tasks/co-routines* itd.

Podržane platforme

Proizvođač	Arhitektura	
Actel	SmartFusion	IAR, Keil, GCC
Altera	Nios II	Nios II IDE with GCC
Atmel	SAM3 (Cortex M3), SAM7 (ARM7), SAM9 (ARM9), AT91, AVR32 UC3, AVR	IAR, GCC, Keil, Rowley CrossWorks
Cypress	PSoC 5 Cortex-M3	GCC, ARM Keil and RVDS, PSoC Creator IDE
Freescale	Kinetis Cortex-M4, Coldfire V2, Coldfire V1, other Coldfire families, HCS12, PPC405 & PPC440	Codewarrior, GCC, Eclipse, IAR
Microchip	PIC32, PIC24, dsPIC, PIC18	MPLAB C32, MPLAB C30, (MPLAB C18 and wizC)
Renesas	RX62N, RX210, SuperH, RL78, H8/S, RX600	GCC, HEW, IAR Embedded
NXP	LPC1700 (Cortex M3), LPC2000 (ARM7)	GCC, Rowley CrossWorks, IAR, Keil, Red Suite, Eclipse
Silicon Labs	Super fast 8051 compatible microcontrollers	SDCC
ST	STM32 (Cortex M3), STR7 (ARM7), STR9 (ARM9)	IAR, Atollic TrueStudio, GCC, Keil, Rowley CrossWorks
TI	MSP430, MSP430X, Stellaris (Cortex-M3)	Rowley CrossWorks, IAR, GCC, Code Composer Studio 4
Xilinx	PPC405 running on a Virtex4 FPGA, PPC440 running on a Virtex5 FPGA, Microblaze	GCC
x86	Any x86 compatible running in Real mode only, plus a Win32 simulator	Studio 2010 Express, MingW, Open Watcom, Borland, Paradigm

Licenciranje

- modificirana GPL licenca – dozvoljava zatvoreni kôd kod komercijalnih rješenja koja koriste FreeRTOS, ali samo pod uvjetom da se **ne mijenja** javno dostupni izvorni kôd samog OS-a:

“The exception permits the source code of applications that use FreeRTOS solely through the API published on this website **to remain closed source**, thus permitting the use of FreeRTOS in commercial applications without necessitating that the whole application be open sourced. The exception can only be used if you wish to combine FreeRTOS with a proprietary product and you comply with the terms stated in the exception itself.”

(cjelokupni tekst licence dostupan na stranicama projekta)

Komercijalne inačice

- SafeRTOS
 - komercijalna inačica OS-a koja je funkcionalno identična FreeRTOS-u, ali čija implementacija kôda zadovoljava standarde vezane uz pouzdanost softvera (functional safety IEC 61508)
 - provedeni postupci analize i verifikacije bitni za ugradnju u kritične sustave
 - npr. Texas Instruments ugradio je inačicu SafeRTOS-a u ROM mikrokontrolera ~~LM3S9B96~~ – nije potrebno plaćati dodatne komercijalne licence za korištenje SafeRTOS-a
- OpenRTOS
 - inačica koja se distribuira pod komercijalnom licencem (nema veze s GPL), a uključuje i dodatne module (USB, *file system*, TCP/IP itd.)
- www.highintegritysystems.com

Organizacija kôda

- FreeRTOS distribucija sastoji se od dva glavna poddirektorija:
 - */Demo* – demo projekti vezani uz pojedine arhitekture i razvojne okoline
 - */Source* – izvorni kôd (portabilne) jezgre OS-a
- unutar */Source* direktorija svega tri datoteke sadrže osnovnu jezgru OS-a:
 - tasks.c, queue.c i list.c
- jezgra OS-a za svaku specifičnu arhitekturu i prevoditelj zahtijeva i prilagodni dio kôda koji se nalazi u */Source/portable*

Konvencije korištene u kôdu

- varijable – kod imenovanja varijabli koriste se dogovorni prefksi za određene tipove podataka:
 - c (char), s (short), l (long), e (enum), x (ostalo), p (pokazivači), u (unsigned)
 - moguće su i kombinacije, npr. “pus”
- funkcije
 - funkcije koje se koriste samo unutar iste .c datoteke (modula) imaju prefiks prv (private)
 - API funkcije – prefiks određuje tip povratne vrijednosti
 - ime funkcije općenito započinje imenom datoteke u kojoj se nalazi
 - npr. vTaskDelete – void funkcija, definirana u *Task.c*, koja obavlja operaciju “*Delete*”

Tipovi podataka

- *char* – dozvoljeno korištenje isključivo potpuno kvalificiranih *char* tipova podataka (*signed char* ili *unsigned char*, različiti prevodioci mogu imati različite podrazumijevane varijante)
- *int* – korištenje *int* tipa nije dozvoljeno, treba koristiti *short* i *long*
- *portTickType* – ako je postavljena konstanta configUSE_16_BIT_TICKS <> 0, *portTickType* je 16-bitni cijeli broj bez predznaka (inače je 32-bitni cijeli broj bez predznaka)
- *portBASE_TYPE* – često se koristi u definicijama API funkcija; određuje najprirodniji i najefikasniji osnovni tip podataka za neku arhitekturu; npr. za 32-bitnu arhitketuru je to 32-bitni cjelobrojni tip podataka

Konfiguriranje FreeRTOS-a

- konfiguriranje OS-a obavlja se u *include* datoteci “FreeRTOSConfig.h”, koja se tipično nalazi u projektnom direktoriju zajedno s aplikacijskim kôdom u “main.c”
- primjeri konfiguracijskih direktiva:

```
#define configUSE_PREEMPTION          1
#define configUSE_IDLE_HOOK            0
#define configCPU_CLOCK_HZ             ((unsigned long)47923200 )
#define configTICK_RATE_HZ              ((portTickType) 1000 )  
#define configMAX_PRIORITIES          ((unsigned portBASE_TYPE )5)
#define configMINIMAL_STACK_SIZE        ((unsigned short )100)
#define configTOTAL_HEAP_SIZE           ((size_t)14200)

/* Set the following definitions to 1 to include the API function, or
zero to exclude the API function. */
#define INCLUDE_vTaskPrioritySet        1
#define INCLUDE_uxTaskPriorityGet        1
#define INCLUDE_vTaskDelete             0
#define INCLUDE_vTaskCleanUpResources   0
#define INCLUDE_vTaskSuspend            1
#define INCLUDE_vTaskDelayUntil         1
#define INCLUDE_vTaskDelay              1
```

Implementacija zadatka - *task function*

- deklaracija funkcije:

```
void vTaskFunction(void *pvParameters);
```

- tipična implementacija funkcije zadatka:

```
void vTaskFunction(void *pvParameters)
{
    /* Lokalne varijable funkcije pohranjuju se na privatnom
       stogu svake instance zadatka, ali ne i statičke lokalne
       varijable! */
    int i = 0;
        while (1)
    {
        // ... tijelo zadatka
    }
    /* zadatak ne bi u normalnim okolnostima trebao
       kod izvoditi ovdje; ukoliko dođe, potrebno
       je eksplicitno obrisati instancu zadatka */
    vTaskDelete(NULL); // NULL - zadatak briše samog sebe
}
```

Kreiranje zadatka

```
portBASE_TYPE xTaskCreate(  
    pdTASK_CODE pvTaskCode,  
    const signed portCHAR * const pcName,  
    unsigned portSHORT usStackDepth,  
    void *pvParameters,  
    unsigned portBASE_TYPE uxPriority,  
    xTaskHandle *pxCreatedTask  
) ;
```

- *pvTaskCode* – pokazivač na task funkciju (ime funkcije)
- *pcName* – deskriptivno ime zadatka
- *usStackDepth* – veličina privatnog stoga zadatka (u riječima, ne oktetima!)
- *pvParameters* – (*void**) pokazivač na parametre koji se proslijeđuju zadatku
- *uxPriority* – prioritet (veći broj označava viši prioritet)
 - od 0 do (*configMAX_PRIORITIES* - 1)
 - configMAX_PRIORITIES* – korisnički definirana konstanta
- *pxCreatedTask* – *handler* kreiranog zadatka (NULL – ako se ne koristi)
- povratna vrijednost:
 - *pdTRUE* – zadatak uspješno kreiran
 - *errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY* – zadatak nije uspješno kreiran

Primjer: dva zadatka s istim prioritetom

```

1) void vTask1(void *pvParameters) {
    const char *pcTaskName = "Task 1 is running\r\n";
    volatile unsigned long ul;
    while(1) {
        vPrintString(pcTaskName); // ispisi ime zadatka
        for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {} // pauza
    }
}

2) void vTask2(void *pvParameters) {
    const char *pcTaskName = "Task 2 is running\r\n";
    volatile unsigned long ul;
    while(1) {
        vPrintString(pcTaskName); // ispisi ime zadatka
        for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {} // pauza
    }
}

int main( void ) {
    // kreiraj zadatak 1
    xTaskCreate(
3)        vTask1,      // pokazivač na task funkciju
        "Task 1",    // ime zadatka (informativno)
        1000,       // dubina privatnog stoga zadatka
        NULL,       // ne proslijedjuju se podaci zadatku
        1,          // prioritet = 1
        NULL);     // ne koristi se task handler
    // kreiraj zadatak 2
4)    xTaskCreate(
        vTask2, "Task 2", 1000, NULL, 1, NULL );
    // pokreni OS
5)    vTaskStartScheduler();
    // do ovdje program nikada ne bi smio doći
    while(1);
}
  
```

Primjer

Ispis:

Task 1 is running

Task 2 is running

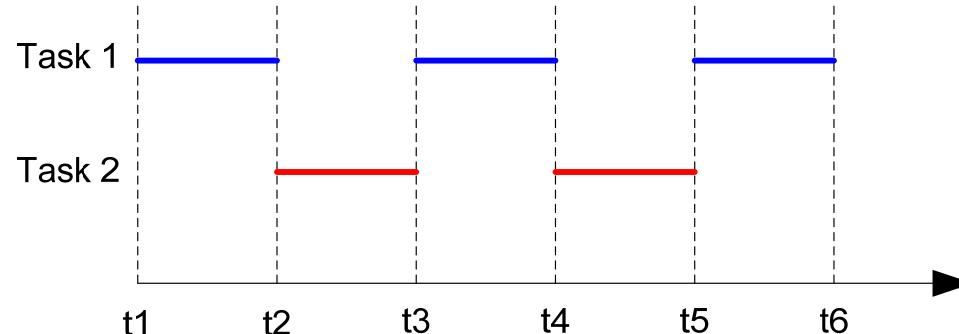
Task 1 is running

Task 2 is running

Task 1 is running

Task 2 is running

...



- u trenutku t_1 "Task 1" kreće s izvođenjem, ispisuje poruku i ulazi u petlju vremenskog čekanja
- u trenutku t_2 raspoređivač zadataka prekida "Task 1" i prepušta procesor zadatku "Task 2", koji nakon ispisa također ulazi u petlju čekanja
- čekanje implementirano na prikazani način vrlo neučinkovito koristi procesor – iako ne radi koristan posao, procesor se ne dodjeljuje drugim zadacima!

Primjer: Instanciranje dva zadatka korištenjem iste task funkcije

```
void vTaskFunc(void *pvParameters) {
    char *pcTaskName;
    volatile unsigned long ul;
    pcTaskName = (char*) pvParameters;
    while(1) {
        vPrintString(pcTaskName);
        for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {}
    }
}

static const char *pcTask1Message = "Task 1 is running\r\n";
static const char *pcTask2Message = "Task 2 is running\t\n";

int main( void ) {
    xTaskCreate(vTaskFunc, "Task 1", 1000, (void*)pcTask1Message, 1, NULL );
    xTaskCreate(vTaskFunc, "Task 2", 1000, (void*)pcTask2Message, 1, NULL );
    vTaskStartScheduler();      // pokreni OS
    // do ovdje program nikada ne bi smio doći
    while(1);
}
```

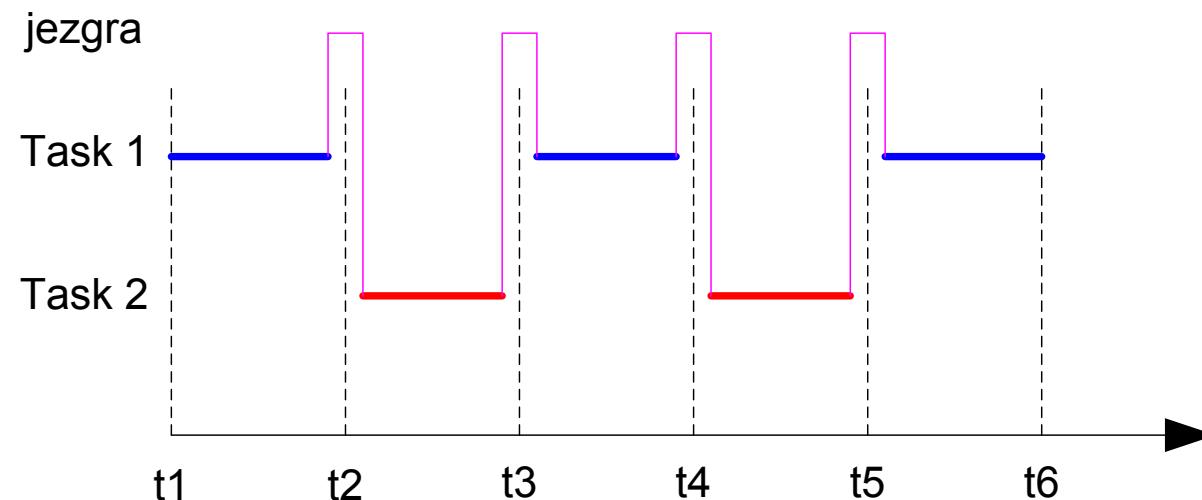
- identična funkcionalnost kao u prethodnom primjeru, ali bez potrebe za višestrukim pisanjem istog programskog koda!

Prioriteti zadataka

- *uxPriority* – inicijalni prioritet zadatka, kojeg je moguće naknadno mijenjati
- raspon: od 0 do *configMAX_PRIORITIES* - 1 (FreeRTOSConfig.h)
- koliko često RTOS obavlja zamjenu konteksta?
 - duljina vremenskog odsječka izvođenja zadatka (*time slice*) definirana je konstantom *configTICK_RATE_HZ* (FreeRTOSConfig.h), koja određuje period timera (*tick period*)
 - npr. *configTICK_RATE_HZ* = 100 Hz => time slice = 10 ms
 - vremenske konstante u FreeRTOS API-ju definiraju se u broju perioda raspoređivača zadataka (“*tick periods*”), a razlučivost ovisi o definiranoj konstanti *configTICK_RATE_HZ*
 - nakon isteka vremenskog odsječka generira se prekid u kojem jezgra OS-a odabire sljedeći zadatak koji će biti u *running* stanju, u ovisnosti o prioritetima zadataka u listi čekanja

Prioriteti zadataka

- izvođenje raspoređivača zadataka u vremenskom prekidu s učestalošću configTICK_RATE_HZ



- što bi se dogodilo ako bi se u prethodnom primjeru promijenili prioriteti zadataka?

Primjer: dva zadatka s različitim prioritetom

```
void vTaskFunc(void *pvParameters) {
    char *pcTaskName;
    volatile unsigned long ul;
    pcTaskName = (char*) pvParameters;
    while(1) {
        vPrintString(pcTaskName);
        for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {}
    }
}

static const char *pcTask1Message = "Task 1 is running\r\n";
static const char *pcTask2Message = "Task 2 is running\t\n";

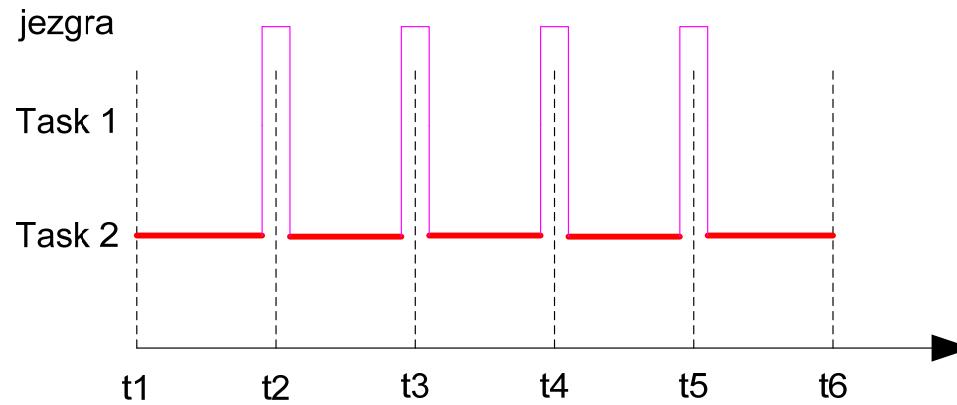
int main( void ) {
    xTaskCreate(vTaskFunc, "Task 1", 1000, (void*)pcTask1Message, 1, NULL );
    xTaskCreate(vTaskFunc, "Task 2", 1000, (void*)pcTask2Message, 2, NULL );
    vTaskStartScheduler(); // pokreni OS
    // do ovdje program nikada ne bi smio doći
    while(1);
}
```

- zadatak “Task 2” ima sada viši prioritet

Primjer

Ispis:

Task 2 is running
...



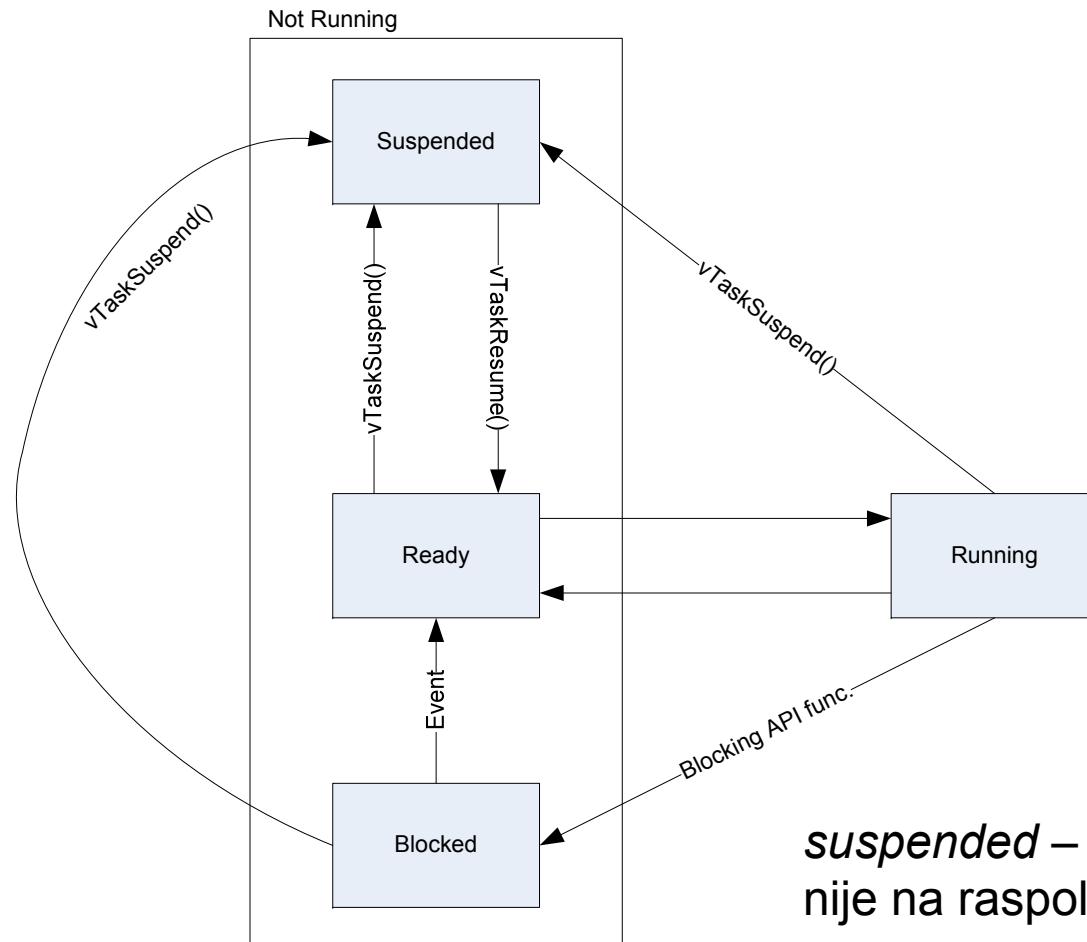
- primjer izgladnjivanja (“*starvation*”) zadatka niže razine prioriteta
- kako riješiti taj problem?

Blokiranje zadatka

- dobrovoljno blokiranje zadataka nužno je da bi zadaci niže razine prioriteta dobili procesorsko vrijeme
- mogućnosti:
 - vremensko blokiranje (čekanje da protekne vremenski interval)
 - sinkronizacijski događaji (signal od strane drugog zadatka ili prekida – semafor, red poruka i sl.)
- koja sve stanja mogu poprimiti zadaci kojima upravlja FreeRTOS operacijski sustav?

Dijagram stanja zadataka

- FreeRTOS stanja: *running, ready, blocked, suspended*



suspended – zadatak postoji, ali je neaktivan i nije na raspolaganju raspoređivaču zadataka

Blokiranje zadatka

- funkcija za vremensko blokiranje zadatka:

```
void vTaskDelay(portTickType xTicksToDelay);
```

- *xTicksToDelay* – broj perioda rasporedioca zadataka (*tick count*) koliko zadatak treba ostati u blokiranom (*blocked*) stanju, prije nego sto se opet vrati u pripravno (*ready*) stanje
- radi bolje razumljivosti kôda i preglednosti, često se koristi konstanta *portTICK_RATE_MS* za konverziju između milisekundi i broja perioda sistemskog timera
 - definirana u “*portmacro.h*” (unutar /Source/portable direktorija)

Primjer: dva zadatka s različitim prioritetom i vremenskim blokiranjem

```
void vTaskFunc(void *pvParameters) {
    char *pcTaskName;
    volatile unsigned long ul;
    pcTaskName = (char*) pvParameters;
    while(1) {
        vPrintString(pcTaskName);
        vTaskDelay( 250 / portTICK_RATE_MS);
    }
}

static const char *pcTask1Message = "Task 1 is running\r\n";
static const char *pcTask2Message = "Task 2 is running\t\n";

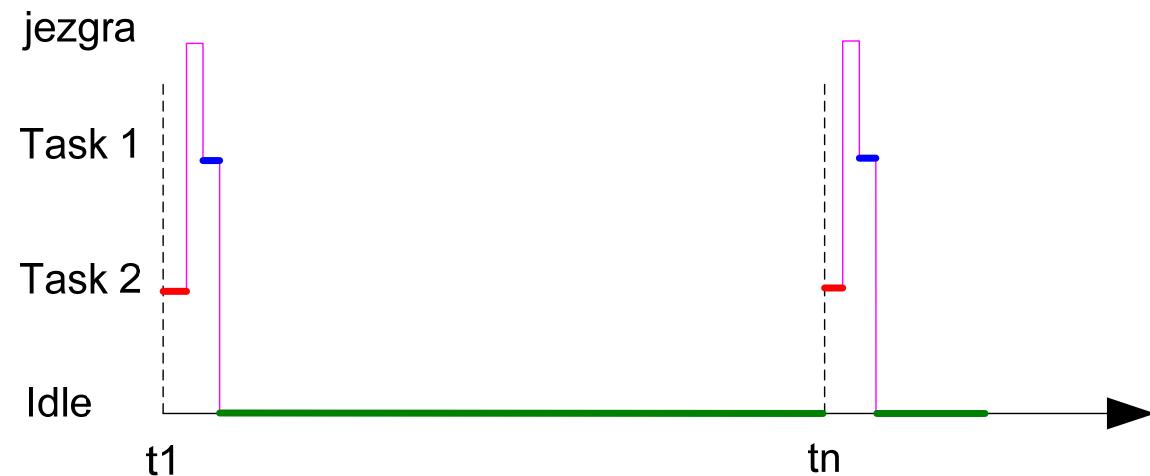
int main( void ) {
    xTaskCreate(vTaskFunc, "Task 1", 1000, (void*)pcTask1Message, 1, NULL );
    xTaskCreate(vTaskFunc, "Task 2", 1000, (void*)pcTask2Message, 2, NULL );
    vTaskStartScheduler(); // pokreni OS
    // do ovdje program nikada ne bi smio doći
    while(1);
}
```

- u ovom primjeru ostvaruje se blokiranje od 250 ms

Primjer

Ispis:

Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
...



- riješen problem izgladnjivanja zadatka niže razine prioriteta
- učinkovitije iskorištenje procesora – IDLE zadatak može npr. postaviti procesor u stanje niske potrošnje

Blokiranje zadatka

razlika

- *vTaskDelay()* funkcija ima nedostatak da se čekanje obavlja *relativno* u odnosu na prethodni poziv
 - akumuliranje procesorskog kašnjenja kroz dulji vremenski period nepovoljno utječe na repetitivne zadatke s konstantnom frekvencijom izvođenja
 - nije moguće postići stabilnu frekvenciju ponavljanja zadatka (npr. za uzorkovanje signala i sl.)
- rješenje - koristiti funkciju *vTaskDelayUntil()* kada se želi postići izvođenje zadatka konstantnom frekvencijom s trenucima buđenja koji se referiraju *apsolutno* prema početku izvođenja:

```
void vTaskDelayUntil(portTickType* pxPreviousWakeTime, portTickType xTimeIncrement);
```

- *pxPreviousWakeTime* – trenutak u kojem je zadatak zadnji put napustio *blocked* stanje; služi samo kao informativna pomoćna referenca i u pravilu korisnički program ne bi smio mijenjati ovu vrijednost!
- *xTimeIncrement* – fiksni vremenski pomak između dva buđenja, koji se ne računa relativno prema trenutku zadnjeg buđenja, već ukupnom broju perioda od početka izvođenja pomnoženim s željenim periodom pozivanja periodičkog zadatka

Primjer: korištenje funkcije vTaskDelayUntil

```
void vTaskFunction(void *pvParameters)
{
    char *pcTaskName;
    portTickType xLastWakeTime;

    pcTaskName = (char*) pvParameters;
    /* xLastWakeTime je potrebno inicijalizirati prije prvog poziva
     funkcije vTaskDelayUntil(). Ovo je ujedno i jedini slučaj kada
     korisnik piše u varijablu xLastWakeTime. Vrijednost inicijalizacije
     odgovara trenutnom broju tick-ova. U svakom pozivu funkcije
     vTaskDelayUntil() vrijednost varijable xLastWakeTime će
     se ažurirati interno. */
    xLastWakeTime = xTaskGetTickCount();
    while(1) {
        vPrintString( pcTaskName );
        // repeticija zadatka 250 ms; iako to ne garantira da će se zadatak izvesti
        // za točno 250 ms (ovisno o prekidima i drugim zadatacima više razine prioriteta
        // koji se trenutno izvode), ipak je eliminiran problem akumuliranog kašnjenja
        // koji postoji s API funkcijom vTaskDelay()
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

Idle task / Idle task hook

- u svakom trenutku barem jedan zadatak mora biti u stanju izvođenja (*running state*)
- ako su svi zadaci blokirani, izvodi se *Idle* zadatak (FreeRTOS ga automatski kreira nakon poziva *vTaskStartScheduler()*)
 - *Idle* task dobiva najniži prioritet 0
- na koji način je moguće povezati aplikacijski kôd s *Idle* taskom?
 - korištenjem *Idle hook* ili *Idle callback* funkcije!
- to je funkcija koja se automatski poziva u svakoj iteraciji *Idle* zadatka
- može poslužiti za:
 - kontinuirano procesiranje niske razine prioriteta, pozadinske zadatke, mjerjenje opterećenosti procesora, upravljanje stanjem niske potrošnje procesora i sl.
- deklaracija *Idle hook* funkcije (ime i prototip moraju egzaktno odgovarati!)

```
void vApplicationIdleHook(void);
```

Primjer: Korištenje *Idle hook* funkcije

```

unsigned long ulIdleCycleCount = 0UL;

void vApplicationIdleHook(void)
{
    ulIdleCycleCount++;
}

void vTaskFunc( void *pvParameters )
{
    char *pcTaskName;
    pcTaskName = (char* ) pvParameters;
    while(1) {
        vPrintStringAndNumber(pcTaskName, ulIdleCycleCount );
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}

int main( void ) {
    xTaskCreate(vTaskFunc, "Task 1", 1000, (void*)pcTask1Message, 1, NULL );
    xTaskCreate(vTaskFunc, "Task 2", 1000, (void*)pcTask2Message, 2, NULL );
    vTaskStartScheduler();
    while(1);
}

```

Ispis:

Task 2 is running
 IdleCount = 0
 Task 1 is running
 IdleCount = 0
 Task 2 is running
 IdleCount = 1343
 Task 1 is running
 IdleCount = 1343
 Task 2 is running
 IdleCount = 2642
 Task 1 is running
 IdleCount = 2642
 ...

- u *FreeRTOSConfig.h* potrebno *configUSE_IDLE_HOOK* postaviti u 1
- *idle hook* ne smije raditi pozive koji rezultiraju prelaskom u *blocked/suspended* stanja

Promjena prioriteta zadatka

- promjena prioriteta zadatka moguća je dinamički pozivom funkcije `vTaskPrioritySet()`:

```
void vTaskPrioritySet(  
    xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority);
```

- *pxTask* – task handler dobiven pozivom funkcije `xTaskCreate()`; ako se proslijedi NULL vrijednost, zadatak mijenja vlastiti prioritet!
- *uxNewPriority* – novi prioritet zadatka; ako je veći od `(configMAX_PRIORITIES-1)`, FreeRTOS će automatski limitirati prioritet na maksimalni mogući

- informacija o trenutnom prioritetu zadatka može se dobiti pozivom funkcije `uxTaskPriorityGet()`:

```
unsigned portBASE_TYPE uxTaskPriorityGet(xTaskHandle pxTask);
```

- *pxTask* – task handler dobiven pozivom funkcije `xTaskCreate()`; ako se proslijedi NULL vrijednost, zadatak ispituje vlastiti prioritet
- povratna vrijednost – prioritet zadatka

Primjer: Dinamička promjena prioriteta zadataka

```

/* Zadatak se izvodi prvi jer je u main() zadan visi prioritet;
ne koristi se mehanizam blokiranja za prepustanje procesora drugom
zadatku, vec dinamicka promjena prioriteta */

void vTask1(void *pvParameters ) {
    unsigned portBASE_TYPE uxPriority;
        uxPriority = uxTaskPriorityGet( NULL );
        while(1) {
            vPrintString("Task1 is running\r\n" );
            vPrintString("About to raise the Task2 priority\r\n" );
            vTaskPrioritySet(xTask2Handle, (uxPriority + 1 ));
        }
}
void vTask2(void *pvParameters ) {
    unsigned portBASE_TYPE uxPriority;
        uxPriority = uxTaskPriorityGet( NULL );
        while(1) {
            vPrintString("Task2 is running\r\n" );
            vPrintString("About to lower the Task2 priority\r\n" );
            vTaskPrioritySet(NULL, (uxPriority - 2));
        }
}

xTaskHandle xTask2Handle;
int main(void) {
    xTaskCreate(vTask1, "Task 1", 1000, NULL, 2, NULL);
    xTaskCreate(vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle);
    vTaskStartScheduler();
    while(1);
}

```

Prvi put kad se izvodi zadatak 2 ima prioritet 3, jer se prije nije izveo



Primjer: Dinamička promjena prioriteta zadataka



Ispis:

Task 1 is running
About to raise Task 2 priority
Task 2 is running
About to lower Task 2 priority
Task 1 is running
About to raise Task 2 priority
Task 2 is running
...

Brisanje zadatka

- zadaci se mogu obrisati pozivom funkcije `vTaskDelete()`:

```
void vTaskDelete(xTaskHandle pxTaskToDelete);
```

- `pxTaskToDelete` – task handler dobiven pozivom funkcije `xTaskCreate()`; ako se proslijedi NULL vrijednost, zadatak briše samog sebe!

- obrisani zadaci brišu se iz memorije i nisu dostupni raspoređivaču zadataka
- dealokacija dinamičke memorije alocirane za TCB (*task control block*), stog i ostale prateće OS strukture vezane uz zadatak odvija se automatski u *Idle* tasku
 - zato je važno da ne dođe do izglađnjivanja *Idle* zadataka ukoliko se koristi `vTaskDelete()` funkcija! 
- pažnja - `vTaskDelete()` će uzrokovati samo automatsku dealokaciju onih memorijskih resursa ~~koje je alocirao OS!~~
 - svu dinamički alociranu memoriju na *heapu* koju je zauzeo sam zadatak (pozivom `malloc()` funkcije i sl.) mora i eksplicitno osloboditi (pozivom `free()` funkcije), inače dolazi do problema curenja memorije (*memory leak*)

Primjer: Dinamičko kreiranje i brisanje zadatka

```
void vTask1(void *pvParameters) {
    while(1) {
        vPrintString("Task1 is running\r\n");
        /* Kreiraj Task 2 s višim prioritetom. Raspoređivač zadataka
         će odmah proslijediti kontrolu nad procesorom prioritetnijem
         zadataku. */
        xTaskCreate(vTask2, "Task 2", 1000, NULL, 2, NULL);
        /* Task 2 će brzo obaviti posao i obrisati samog sebe; Task 1
         čeka da prođe 100 ms prije nastavka */
        vTaskDelay(100 / portTICK_RATE_MS); // 100 ms
    }
}

void vTask2(void* pvParameters) {
    vPrintString("Task2 is running and about to delete itself\r\n");
    vTaskDelete(NULL);
}

int main(void) {
    xTaskCreate(vTask1, "Task 1", 1000, NULL, 1, NULL);
    vTaskStartScheduler();
    while(1);
}
```

Primjer: Dinamičko kreiranje i brisanje zadataka



Ispis:

Task 1 is running

Task 2 is running and about to delete itself

Task 1 is running

Task 2 is running and about to delete itself

Task 1 is running

Task 2 is running and about to delete itself

...

Kooperativna višezadaćnost

- FreeRTOS podržava “*Fixed Priority Preemptive Scheduling*” model
 - fiksni prioriteti – jezgra OS-a ne mijenja automatski prioritete zadataka
 - *preemptive scheduling* – istiskivanje zadataka
- FreeRTOS također podržava i “*Co-operative Multitasking*” model
 - nema zamjene konteksta sve dok se zadatak dobrovoljno ne blokira (npr. timeout) ili pozove **eksplicitno** funkciju za zamjenu konteksta:

```
void taskYIELD();
```

- isključivanje istiskivanja: configUSE_PREEMPTION = 0
- pažnja – kod kooperativne višezadaćnosti nema automatskog dijeljenja procesora između zadataka istog prioriteta!

Redovi (Queue)

- globalne OS strukture za razmjenu podataka između zadataka (zadaci ne mogu biti “vlasnici” resursa)
- FIFO strukture u koje se podaci mogu umetati s obje strane reda, prema potrebi i prioritetu poruke
- zadan broj i veličina svakog elementa prilikom kreiranja reda
- podaci se prilikom stavljanja u red *kopiraju* iz memorije izvorišnog zadatka u privatnu memoriju reda, a zatim još jedanput iz memorije reda u memoriju odredišnog zadatka

Redovi (Queue)

- kod čitanja praznog reda i upisa u puni red zadatak se automatski stavlja u stanje blokiranja
 - ugrađena podrška za blokiranje s timeoutom
- ako se više zadataka nalazi u listi čekanja, prednost za operaciju nad redom uvijek ima *najprioritetniji* zadatak
- u slučaju da su svi zadaci istog prioriteta, prednost ima onaj koji *najdulje* čeka (FIFO načelo)

Kreiranje reda

- redovi se moraju eksplisitno kreirati pozivom funkcije prije korištenja:

```
xQueueHandle xQueueCreate(  
    unsigned portBASE_TYPE uxQueueLength,  
    unsigned portBASE_TYPE uxItemSize);
```

- *uxQueueLength* – najveći broj elemenata reda
- *uxItemSize* – veličina podatkovnog elementa reda (u oktetima)
- povratna vrijednost – *queue handle*, NULL ako nema dovoljo prostora na *heap-u*

- operacijski sustav se brine oko dinamičke alokacije memorije za red
- *handle* reda se pohranjuje tipično u globalnu varijablu kako bi bio dostupan svih zadacima jer redovi upravo služe za komunikaciju između zadataka

Upis podataka u red

```
portBASE_TYPE xQueueSendToFront(xQueueHandle xQueue,  
        const void * pvItemToQueue,  
        portTickType xTicksToWait);
```

```
portBASE_TYPE xQueueSendToBack( xQueueHandle xQueue,  
        const void * pvItemToQueue,  
        portTickType xTicksToWait);
```

- *xQueue – queue handle*
- *pvItemToQueue* – pokazivač na memorijsku lokaciju koja sadrži element; automatski se kopira broj okteta zadan veličinom podatka koji je proslijeđen kao parametar prilikom kreiranja reda
- *xTicksToWait* – najdulje vrijeme koje zadatak može ostati blokiran prilikom pisanja u puni red (parametar označava broj *tickova raspoređivača zadataka*); ako se postavi vrijednost 0, zadatak odmah izlazi i ne blokira na punom redu; ako se postavi vrijednost *portMAX_DELAY*, zadatak će trajno ostati blokiran (bez timeouta), ali pod uvjetom da je definirano *INCLUDE_vTaskSuspend = 1* u *FreeRTOSConfig.h*
- povratna vrijednost – pdPASS (ako je upis uspio prije isteka timeouta), errQUEUE_FULL (inače)

Upis podataka u red

- funkcija `xQueueSendToBack()` upisuje podatak na kraj reda (uobičajeno ponašanje za FIFO strukturu), dok `xQueueSendToFront()` podatak upisuje na početak reda (npr. poruka visokog prioriteta)
- potreban je oprez kod upisa u red iz prekidne rutine: funkcije `xQueueSendToFront()` i `xQueueSendToBack()` **ne smiju** se pozivati **iz prekidnih rutina!**
- umjesto tih funkcija, potrebno je koristiti posebne verzije `xQueueSendToFrontFromISR()` i `xQueueSendToBackFromISR()`, koji imaju istu funkcionalnost, ali implementaciju prilagođenu za poziv iz prekida
 - funkcije koje se pozivaju iz prekida ne smiju ni pod kojim okolnostima blokirati čekajući na resurs – prekidne rutine se moraju izvesti što brže!

Čitanje podataka iz reda

```
portBASE_TYPE xQueueReceive(xQueueHandle xQueue,
    const void * pvBuffer,
    portTickType xTicksToWait);
portBASE_TYPE xQueuePeek(xQueueHandle xQueue,
    const void * pvBuffer,
    portTickType xTicksToWait);
```

- *xQueue* – queue handle
- *pvBuffer* – pokazivač na memorijsku lokaciju na koju se kopira element koji se pročita iz reda; automatski se kopira broj okteta zadan veličinom elementa reda koji je kao parametar proslijeđen prilikom kreiranja reda
- *xTicksToWait* – najdulje vrijeme koje zadatak može ostati blokiran prilikom čitanja praznog reda (parametar označava broj tickova raspoređivača zadatka); ako se postavi vrijednost 0, zadatak odmah izlazi i ne blokira na praznom redu; ako se postavi vrijednost *portMAX_DELAY*, zadatak će trajno ostati blokiran (bez timeouta), ali pod uvjetom da je definirano INCLUDE_vTaskSuspend = 1 u FreeRTOSConfig.h
- povratna vrijednost – pdPASS (ako je čitanje uspjelo prije isteka timeouta), errQUEUE_EMPTY (inače)

Čitanje podataka iz reda

- funkcija `xQueueReceive()` čita podatak i uklanja ga iz reda (destruktivno čitanje), dok funkcija `xQueuePeek()` čita podatak bez brisanja iz reda (nedestruktivno čitanje)
- potreban je oprez kod čitanja iz prekidne rutine: funkcije `xQueueReceive()` i `xQueuePeek()` **ne smiju** se pozivati iz prekidnih rutina!
- umjesto tih funkcija, potrebno je koristiti posebne verzije `xQueueReceiveFromISR()` i `xQueuePeekFromISR()`, koji imaju istu funkcionalnost, ali implementaciju prilagođenu za poziv iz prekida

Određivanje broja poruka u redu

- broj poruka koje se trenutno nalaze u redu moguće je odrediti pozivom funkcije:

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

- *xQueue – queue handle*
- povratna vrijednost – broj poruka

- za provjeru broja poruka u redu iz prekida koristi se posebna verzija funkcije *uxQueueMessagesWaitingFromISR()*

Primjer: čitanje iz reda s blokiranjem

```
xQueueHandle xQueue;           // red je globalni resurs

int main(void) {
    // kreiraj red za 5 varijabli tipa long
    xQueue = xQueueCreate(5, sizeof(long));
    if(xQueue != NULL) {
        // kreiraj dvije instance zadataka za slanje s istim prioritetom (1);
        // svaka instanca šalje u red drugačiju vrijednost (100 ili 200)
        xTaskCreate(vSenderTask, "Sender1", 1000, (void *)100, 1, NULL );
        xTaskCreate(vSenderTask, "Sender2", 1000, (void *)200, 1, NULL );
        // kreiraj zadatak koji čita iz reda, s višim prioritetom
        xTaskCreate(vReceiverTask, "Receiver", 1000, NULL, 2, NULL);
        // pokreni OS
        vTaskStartScheduler();
    }
    else
    {
        // nema dovoljno memorije na heapu!
    }
    while(1);
}
```

Primjer: čitanje iz reda s blokiranjem

```
void vSenderTask(void *pvParameters) {
    long lValueToSend;
    portBASE_TYPE xStatus;

    /* U primjeru se kreiraju dvije instance zadatka koje salju razlicite
     * poruke (odredjene preko pvParameters) */
    lValueToSend = ( long ) pvParameters;
    while(1) {
        /* posalji poruku bez blokiranja; red nikada ne bi trebao sadrzavati u
         * ovom primjeru vise od jednog elementa */
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
        if(xStatus != pdPASS) {
            vPrintString( "Could not send to the queue.\r\n" ); // pogreška!
        }
        // prepusti EKSPLICITNO kontrolu drugom zadatku - kooperativna
        // višezadaćnost (poziv funkcije taskYIELD() moze se koristiti i ako
        // se koristi višezadaćnost s istiskivanjem, jer rezultira
        // prijevremenim pozivom raspoređivaču zadataka; kod kooperativne
        // višezadaćnosti se mora obavezno koristiti!
        taskYIELD();
    }
}
```

Primjer: čitanje iz reda s blokiranjem

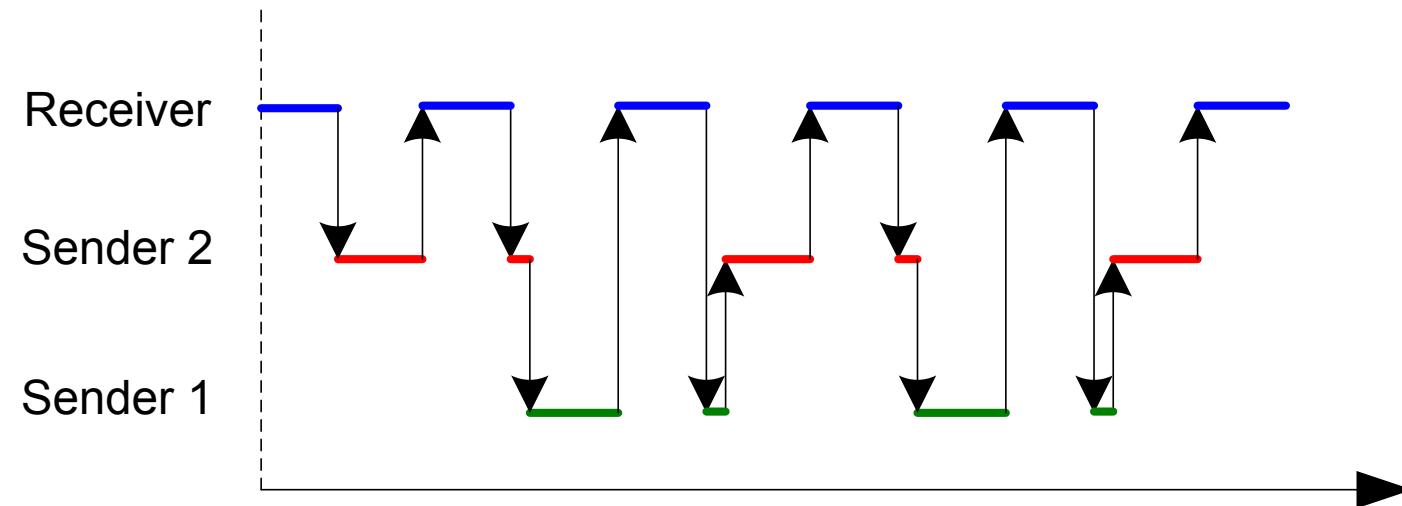
```
void vReceiverTask( void *pvParameters)
{
    long lReceivedValue;           // memorijski prostor za prihvat podataka iz reda
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    while(1) {
        // red bi uvijek na ovom mjestu trebao biti prazan, jer ovaj zadatak
        // prazni red cim se u njega upisu podaci
        if(uxQueueMessagesWaiting(xQueue) != 0 ) {
            vPrintString( "Queue should have been empty!\r\n" );
        }
        // prihvati podataka s timeoutom
        xStatus = xQueueReceive(xQueue, &lReceivedValue, xTicksToWait);
        if(xStatus == pdPASS {
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
        else {
            // ovo se ne bi smjelo dogoditi u primjeru
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}
```

Primjer: čitanje iz reda s blokiranjem

- u primjeru zadatak više razine prioriteta čita red na način da blokira (s timeoutom) na praznom redu
- dva zadatka koja šalju podatke imaju nižu razinu prioriteta – čim se podatak nađe u redu, zadatak za čitanje istiskuje zadatak za slanje i prazni red!
- što bi se dogodilo da se u primjeru nije koristila funkcija `taskYield()`?
 - nakon prihvata podatka procesor bi se vratio zadnjem aktivnom zadatku za slanje; sve dok traje *time slice* tog zadatka ne bi došlo do zamjene konteksta s drugim zadatkom
 - na ovaj način postignuta je zamjena konteksta između zadataka niže razine prioriteta *prije* isteka perioda kada bi to napravio raspoređivač zadataka
 - primjer kombiniranja *pre-emptive* i *co-operative* višezadačnosti

Primjer: čitanje iz reda s blokiranjem



Ispis:

Received = 100

Received = 200

Received = 100

Received = 200

Received = 100

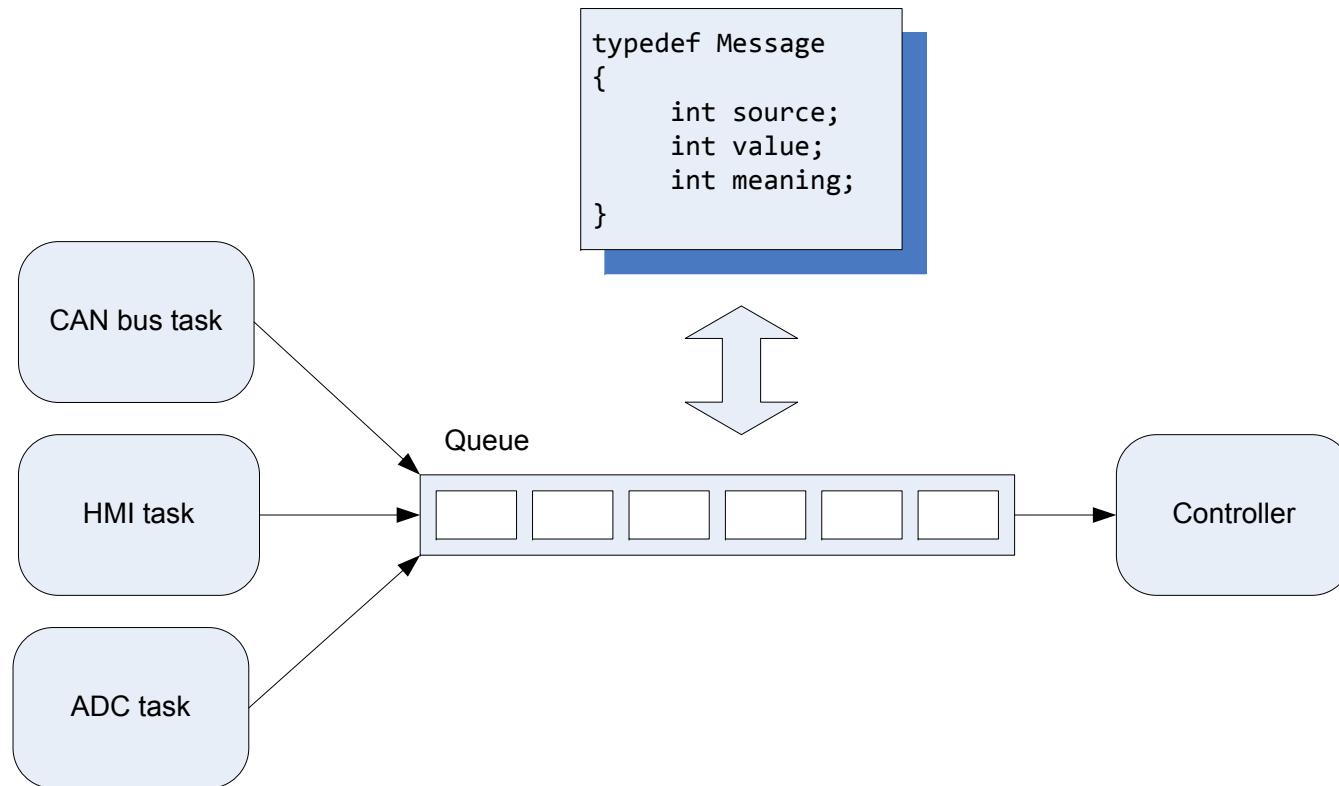
Received = 200

Received = 200

...

Transfer složenih tipova podataka pomoću reda

- tipično su poruke koje se šalju između zadataka složene strukture podataka:



Primjer: slanje složenih poruka između zadataka korištenjem reda

```
// opis poruke
typedef struct {
    unsigned char ucValue;
    unsigned char ucSource;
} xMESSAGE;
// statički niz poruka
static const xMESSAGE xMsgsToSend[2] = {
    { 100, mainSENDER_1 },
    { 200, mainSENDER_2 }
};
xQueueHandle xQueue;           // globalni resurs
int main(void) {
    xQueue = xQueueCreate(3, sizeof(xMESSAGE));
    if(xQueue != NULL) {
        // kreiraj sender taskove, više razine prioriteta, i proslijedi kao
        // parametar pokazivač na statičku poruku koju svaki zadatak šalje:
        xTaskCreate(vSenderTask, "Sender1", 1000, &(xStructsToSend[0]), 2, NULL);
        xTaskCreate(vSenderTask, "Sender2", 1000, &(xStructsToSend[1]), 2, NULL);
        // kreiraj zadatak za čitanje iz reda nižeg prioriteta:
        xTaskCreate(vReceiverTask, "Receiver", 1000, NULL, 1, NULL );
        vTaskStartScheduler(); } else { // nema dovoljno memorije na heapu!
    }
    while(1);
}
```

Primjer: slanje složenih poruka između zadataka korištenjem reda

```
void vSenderTask(void *pvParameters) {
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    while(1) {
        xStatus = xQueueSendToBack(xQueue, pvParameters,
                                   xTicksToWait);
        if( xStatus != pdPASS ) {
            // pogreška - receiver je trebao unutar 100 ms
            // pročitati poruku!
            vPrintString( "Could not send to the queue.\r\n" );
        }
        taskYIELD();
    }
}
```

Primjer: slanje složenih poruka između zadataka korištenjem reda

```

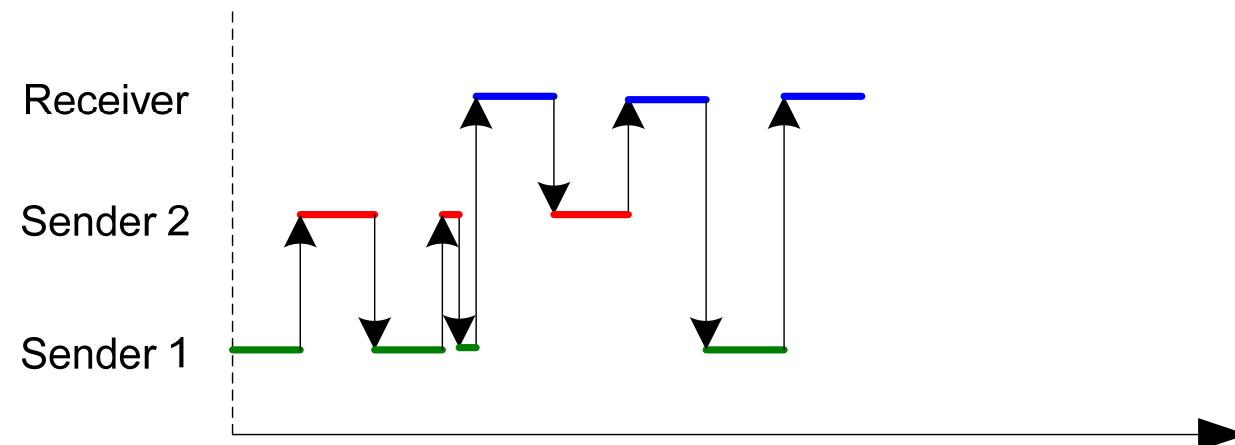
void vReceiverTask(void *pvParameters) {
    xMESSAGE xReceivedStructure;      // memorija za pohranu poruke
    portBASE_TYPE xStatus;
    while(1) {
        // obzirom da je receiver zadatak nize razine prioriteta, neće se probuditi sve dok
        // sender taskovi ne blokiraju na punom redu:
        if (uxQueueMessagesWaiting(xQueue) != 3) {
            // pogreška!
            vPrintString( "Queue should have been full!\r\n" );
        }
        // neblokirajuće citanje, jer se očekuje da je red uvijek pun kada se dođe do ovdje:
        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );
        if( xStatus == pdPASS ) {
            // ispisi primljeni podatak
            if( xReceivedStructure.ucSource == mainSENDER_1 ) {
                vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
            }
            else {
                vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
            }
        }
        else
        {
            // pogreška
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}
  
```

2x se kopira, ne radi automatski

Primjer: slanje složenih poruka između zadataka korištenjem reda

- u primjeru zadaci više razine prioriteta upisuju podatke u red na način da blokiraju (s timeoutom) na punom redu
- zadatak za čitanje ima nižu razinu prioriteta – postaje aktivan tek kada oba zadataka za slanje blokiraju na punom redu!
 - tada se čita jedan podatak i zadaci za slanje odmah istiskuju zadatak za primanje podataka
- što bi se dogodilo da se u primjeru nije koristila funkcija `taskYield()`?
 - nakon slanja podatka u red, slanje bi se nastavilo u istom aktivnom zadataku, sve dok traje *time slice* tog zadataka
 - na ovaj način postignuta je zamjena konteksta između zadataka odmah nakon što jedan zadatak upiše podatak u red

Primjer: slanje složenih struktura između zadataka korištenjem reda



Ispis:

From Sender 1 = 100

From Sender 2 = 200

From Sender 1 = 100

From Sender 2 = 200

From Sender 1 = 100

From Sender 2 = 200

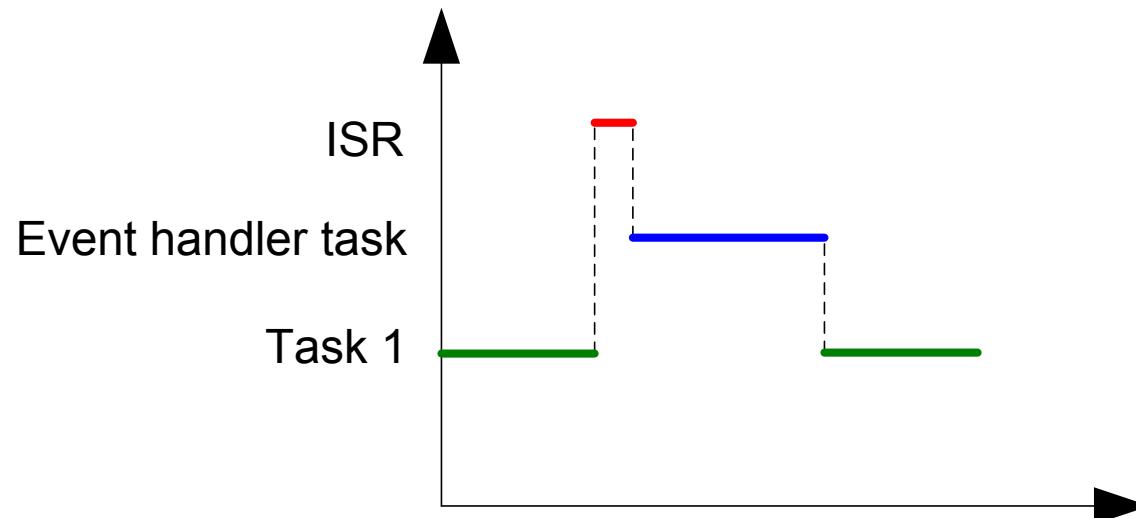
...

Upravljanje prekidima

- mogućnosti detekcije događaja:
 - pozivanje u upravljačkoj petlji
 - prekidne rutine
- odgođena obrada prekida (*deferred interrupt processing*):
 - prekidne rutine trebaju se izvoditi što brže – nije poželjno raditi cijelu obradu događaja u ISR!
 - odgođena obrada prekida:
 - ISR obavlja minimalni potrebni posao i priprema zadatak za naknadnu, sveobuhvatniju obradu događaja
 - obrada događaja odvija se u *interrupt handler tasku* – obični task

Odgodena obrada prekida

- za sinkronizaciju između prekida i zadatka za odgođenu obradu prekida može se koristiti binarni semafor
- zadatak je blokiran i čeka na događaj – pokušavati zauzeti semafor, koji će se oslobođiti u prekidu i time zadatku signalizirati pojavu događaja
- prioritetni događaji – dati viši prioritet zadacima za obradu



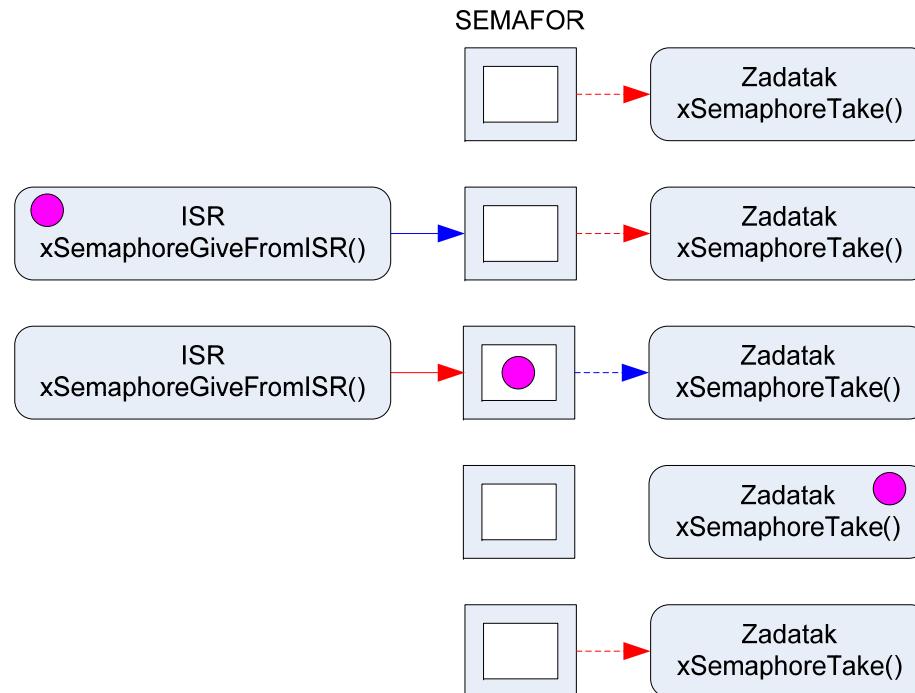
Binarni semafori

- kreiranje semafora:

```
void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

- *xSemaphore* – handle na kreirani binarni semafor

- model komunikacije između ISR-a i zadatka za obradu događaja:



Binarni semafori

- u scenariju sinkronizacije između ISR-a i zadatka za obradu događaja, ponašanje binarnog semafora može se usporediti s redom s jednim elementom:
 - zadatak blokira na praznom redu (binarni semafor)
 - ISR šalje “token” u red – odblokira zadatak
 - zadatak uzima “token” iz reda, koji je sada prazan
 - sljedeći put kada zadatak želi uzeti novi “token”, blokirat će na praznom redu
 - zadatak će se probuditi ponovo kada ISR stavi “token” u red, omogućavanjem binarnog semafora

Binarni semafori

- uzimanje semafora:

```
portBASE_TYPE xSemaphoreTake(  
    xSemaphoreHandle xSemaphore, portTickType xTicksToWait );
```

- *xSemaphore* – handle semafora koji se želi zauzeti
 - *xTicksToWait* – naj dulje vrijeme čekanja da semafor postane raspoloživ; ako je parametar jednak 0, funkcija neće čekati da semafor postane raspoloživ u slučaju da je zauzet prilikom poziva; ako je parametar jednak konstanti *portMAX_DELAY*, funkcija će beskonačno dugo čekati na semafor
 - povratna vrijednost – *pdPASS* – semafor uspješno zauzet; *pdFALSE* - inače
- funkcija *xSemaphoreTake* ne smije se koristiti u prekidnoj rutini!
 - može se koristiti za sve tipove semafora

Binarni semafori

- davanje semafora:

```
portBASE_TYPE xSemaphoreGiveFromISR(  
    xSemaphoreHandle xSemaphore,  
    portBASE_TYPE *pxHigherPriorityTaskWoken);
```

- *xSemaphore* – handle semafora koji se želi dati
- *pxHigherPriorityTaskWoken* – u slučaju da je poziv funkcije (davanje semafora) uzrokovao prelazak nekog zadatka višeg prioriteta od trenutnog izvođenog (prekinutog prekidom) iz blokiranog u pripravno stanje, vrijednost ovog *byref* parametra nakon izlaska iz funkcije bit će pdTRUE (inače je pdFALSE); u slučaju da je vraćena vrijednost pdTRUE, poželjno je obaviti zamjenu konteksta **prije** izlaska iz prekida, kako bi zadatak koji je odblokiran što prije mogao krenuti s izvođenjem
- povratna vrijednost – pdPASS – semafor uspješno “predan”; pdFAIL – u slučaju da se semafor nije mogao “osloboditi”, jer je već “slobodan”
- funkcija *xSemaphoreGiveFromISR()* može se koristiti za sve tipove semafora
- namijenjena je korištenju u prekidnoj rutini (ISR)

Primjer

```
xSemaphoreHandle xBinarySemaphore;

int main(void)
{
    // kreiraj binarni semafor
    vSemaphoreCreateBinary(xBinarySemaphore);
    // ... konfiguriraj prekide
    // provjera da li je semafor kreiran:
    if (xBinarySemaphore != NULL) {
        // kreiraj handler task (obično mu se zadaje visi prioritet,
        // npr. 3 u ovom slučaju)
        xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );
        // pokreni OS
        vTaskStartScheduler();
    }
    while(1);
}
```

Primjer

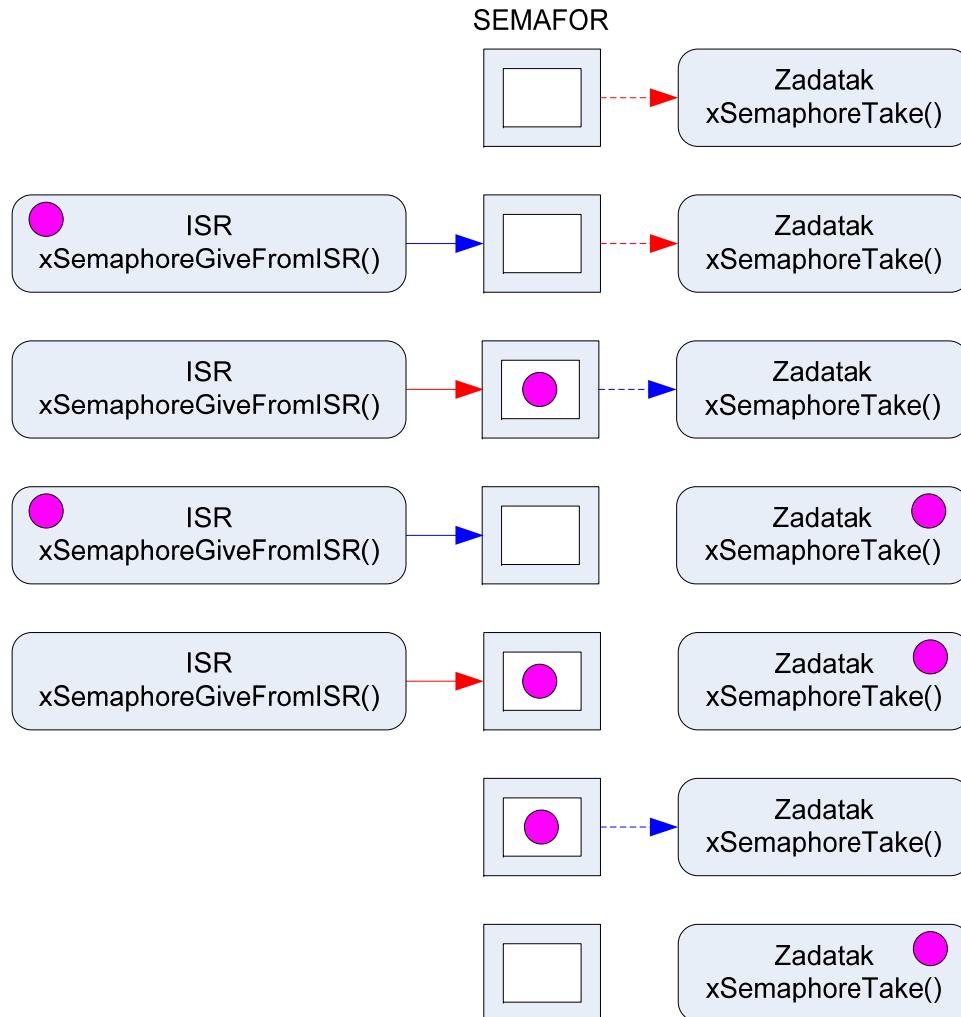
```
void vHandlerTask(void *pvParameters) {
    while(1) {
        // Zadatak ceka (beskonacno) na dogadjaj koristenjem
        // sinkronizacijskog semafora xBinarySemaphore.
        // Inicijalno kreirani semafor (prije pokretanja OS-a) je
        // neprolazan.
        xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
        // ...
        vPrintString( "Handler task - Processing event.\r\n" );
    }
}

void __interrupt vExampleISR(void) {
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;
    // otpusti semafor - signal događaja handler tasku:
    xSemaphoreGiveFromISR(
        xBinarySemaphore, &xHigherPriorityTaskWoken);
    if (xHigherPriorityTaskWoken == pdTRUE) {
        // obavi zamjenu konteksta prije izlaska iz prekida!
        portSWITCH_CONTEXT();
    }
}
```

Binarni semafor – *event latching*

- binarni semafori korisni su za implementaciju procesiranja događaja
- što ako frekvencija događaja raste?
 - može se dogoditi da za vrijeme obrade događaja u *event handler tasku* dođe novi događaj (npr. na U/I liniji), koji će pokrenuti prekid u kojem će se postaviti semafor prije nego što handler task završi s poslom
 - u tom slučaju, handler task neće ući u blokirano stanje i odmah će nastaviti s procesiranjem novog događaja
 - *event latching* – binarni semafor je zapamtio da je novi događaj stigao prije nego se završilo s obradom prethodnog

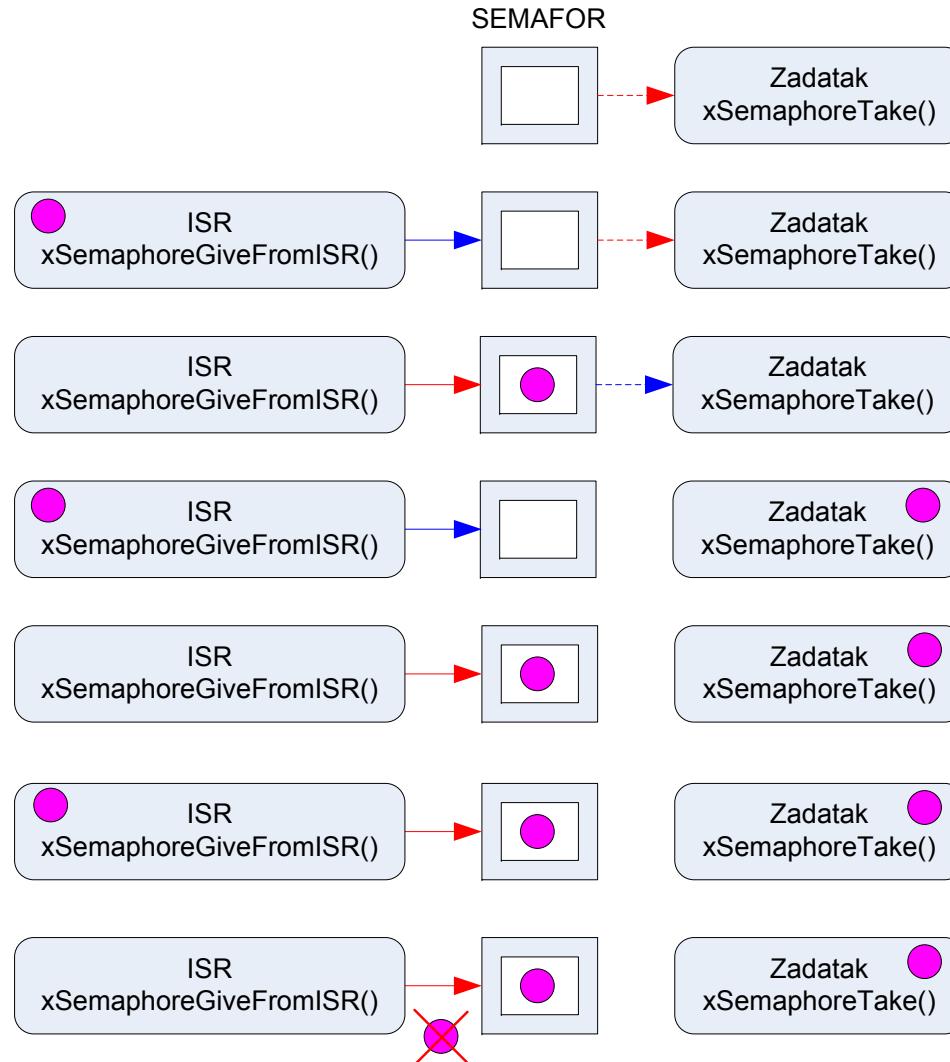
Binarni semafor – event latching



Problem: što ako umjesto jednog dođe više istovrsnih događaja dok se prvi još obrađuje?

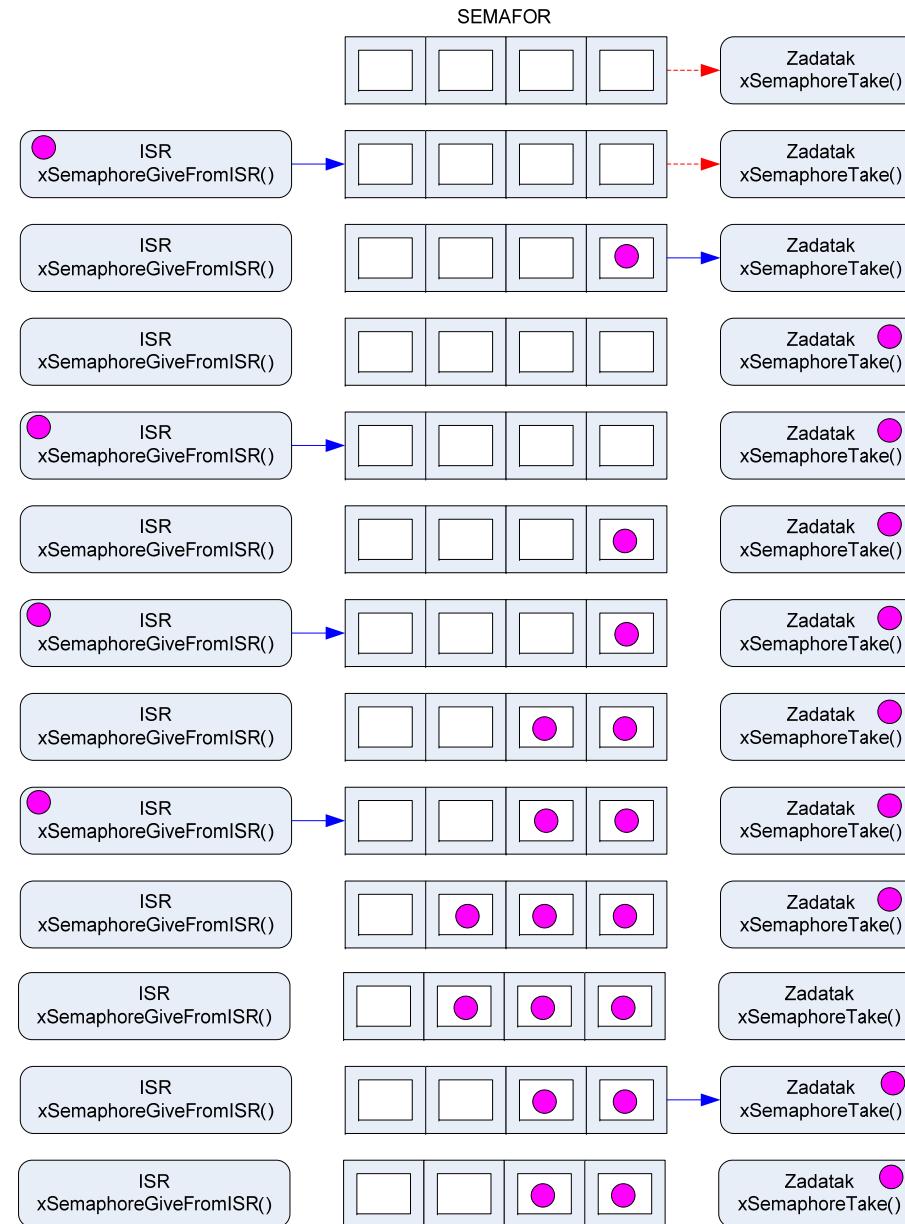
Binarni semafor ne može zapamtiti više od jednog dodatnog događaja!

Binarni semafor – event latching



Brojeći semafori

- konceptualno proširenje binarnog semafora – može se promatrati kao red koji može primiti N “tokena”
- *event latching* – mogućnost prihvatanja većeg broja događaja u red zadatka za obradu ako prebrzo stižu
- tipične primjene broječih semafora:
 - brojanje događaja – brojač služi kod podrške za *event latching* i označava broj događaja koji čekaju u redu na obradu
 - upravljanje resursima – scenarij gdje vrijednost semafora označava broj raspoloživih resursa; svako zauzimanje resursa umanjuje vrijednost; kada je vrijednost jednaka 0, prvi sljedeći zadatak mora čekati na oslobođanje resursa, tj. da vrijednost semafora postane pozitivna
 - u ovom slučaju semafor se tipično inicijalizira na vrijednost koja predstavlja broj raspoloživih resursa



Brojeći semafori

- kreiranje brojećeg semafora:

```
xSemaphoreHandle xSemaphoreCreateCounting(  
    unsigned portBASE_TYPE uxMaxCount,  
    unsigned portBASE_TYPE uxInitialCount);
```

- *uxMaxCount* – najveća vrijednost brojećeg semafora
- *uxInitialCount* – inicijalna vrijednost semafora; 0 predstavlja neprolazan semafor; ako se semafor koristi za brojanje događaja (*event latching*), inicijalna vrijednost treba biti 0 (dolazak novog događaja odgovara inkrementiranju vrijednosti); ako se semafor koristi za koordiniranje pristupa resursima, inicijalna vrijednost treba biti postavljena na maksimalnu moguću vrijednost (zauzeće resursa odgovara dekrementiranju vrijednosti)
- povratna vrijednost – *handle* na novokreirani brojeći semafor; NULL ako nema dovoljno memorije na *heap-u*

Primjer – *Event Latching*

```
xSemaphoreHandle xCountingSemaphore;  
int main(void) {  
    // kreiraj brojaci semafor s maksimalnom vrijednoscu 10 i pocetnom  
    // vrijednoscu 0:  
    xCountingSemaphore = xSemaphoreCreateCounting(10, 0);  
    // ... konfiguriraj prekide  
    // provjera da li je semafor kreiran:  
    if (xCountingSemaphore != NULL) {  
        // handler task (obicno mu se zadaje visi prioritet, npr. 3 u  
        // ovom slucaju)  
        xTaskCreate(vHandlerTask, "Handler", 1000, NULL, 3, NULL );  
        // pokreni OS  
        vTaskStartScheduler();  
    }  
    while(1);  
}
```

Primjer – *Event Latching*

```
void vHandlerTask(void *pvParameters) {
    while(1) {
        // Zadatak ceka (beskonacno) na dogadjaj koristenjem
        // sinkronizacijskog semafora xCountingSemaphore.
        // Inicijalno kreirani semafor (prije pokretanja OS-a) je
        // neprolazan.
        xSemaphoreTake(xCountingSemaphore, portMAX_DELAY);
        // ...
        vPrintString("Handler task - Processing event.\r\n");
    }
}

void __interrupt vExampleISR(void) {
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;
    // otpusti semafor - signal događaja handler tasku:
    xSemaphoreGiveFromISR(
        xCountingSemaphore, &xHigherPriorityTaskWoken);
    if (xHigherPriorityTaskWoken == pdTRUE) {
        // obavi zamjenu konteksta prije izlaska iz prekida!
        portSWITCH_CONTEXT();
    }
}
```

Korištenje redova unutar prekidnih rutina

- za komunikaciju između prekida i zadataka mogu se koristiti redovi
- primjene:
 - razmjena podataka između prekida i zadataka
 - sinkronizacija između prekida i zadataka
 - semafori se mogu koristiti samo za sinkronizaciju!
- funkcije koje se mogu koristiti u prekidima imaju sufiks *FromISR*:
 - *xQueueSendToFrontFromISR()*
 - *xQueueSendToBackFromISR()*
 - *xQueueReceiveFromISR()*
- deklaracije ISR funkcija za komunikaciju putem redova su nešto drugačije od odgovarajućih “običnih” funkcija

Funkcije za slanje podataka

```
portBASE_TYPE xQueueSendToFrontFromISR(  
    xQueueHandle xQueue,  
    void *pvItemToQueue  
    portBASE_TYPE *pxHigherPriorityTaskWoken);  
  
portBASE_TYPE xQueueSendToBackFromISR(  
    xQueueHandle xQueue,  
    void *pvItemToQueue  
    portBASE_TYPE *pxHigherPriorityTaskWoken);
```

- *xQueueHandle* – handle reda
- *pvItemToQueue* – pokazivač na memorijski međuspremnik s podatkom koji se dodaje u red (broj okteta se zadaje prilikom kreiranja reda)
- *pxHigherPriorityTaskWoken* – slično kao i kod semafora, operacija slanja u red može rezultirati prebacivanjem stanja zadatka razine prioriteta više od razine prekinutog zadatka iz *blocked* u *ready*; u tom slučaju, vrijednost ovog *byref* parametra je pdTRUE, a zamjenju konteksta treba obaviti prije izlaska iz prekida
- povratna vrijednost – *pdPASS* – podatak je uspješno zapisan u red, *errQUEUE_FULL* inače

Primjer

Zadatak *vNumberGenerator()* periodički generira niz cijelih brojeva i šalje ih u red *xIntegerQueue* (5 puta u 200 ms). Timer prekid provjerava i prazni red *xIntegerQueue* svakih 10 ms. Na temelju vrijednosti cijelog broja pročitanog iz reda *xIntegerQueue*, formira tekstualnu poruku i proslijedi je putem reda *xStringQueue* zadatku *vPrintText()*, koji ispisuje primljene tekstualne poruke.

```
xQueueHandle xIntegerQueue;
xQueueHandle xStringQueue;

int main(void) {
    // kreiraj redove:
    xIntegerQueue = xQueueCreate(10, sizeof(unsigned long));
    xStringQueue = xQueueCreate(10, sizeof(char *));
    // zadatak koji salje niz brojeva (prioritet 1)
    xTaskCreate(vNumberGenerator, "Generator", 1000, NULL, 1, NULL);
    // zadatak koji ispisuje tekstualne poruke indeksirane cijelim
    // brojem iz vNumberGenerator (prioritet 2)
    xTaskCreate(vPrintText, "Printer", 1000, NULL, 2, NULL);
    vTaskStartScheduler();           // start OS
    while(1);
}
```

Primjer

```
// zadatak koji generira cijele brojeve i salje ih u
// timer0 prekid putem xIntegerQueue reda
// (generira se 5 brojeva u 200 ms, a timer0 radi s frekvencijom 100 Hz)

void vNumberGenerator(void *pvParameters) {

    portTickType xLastExecutionTime;
    unsigned portLONG ulValueToSend = 0;
    int i;

    xLastExecutionTime = xTaskGetTickCount();
    while(1) {
        vTaskDelayUntil(&xLastExecutionTime, 200 / portTICK_RATE_MS);
        for(i = 0; i < 5; i++) {
            // red ce sigurno biti prazan jer je prosjecna brzina
            // slanja manja od prosjecne brzine citanja u ISR:
            xQueueSendToBack(xIntegerQueue, &ulValueToSend, 0);
            ulValueToSend++;
        }
        vPrintString("Generator task - data sent\r\n");
    }
}
```

```

// timer ISR koji okida svakih 10 ms, prima podatke iz xIntegerQueue
// (ako ih ima), koji predstavljaju indekse tekstualnih poruka
// koje se preko reda xStringQueue salju zadatku vPrintText
void __interrupt vTimer0InterruptHandler(void) {
    static portBASE_TYPE xHigherPriorityTaskWoken;
    static unsigned long ulReceivedNumber;
    static const char *pcStrings[] = { // static - nisu na stogu!
        "String 0\r\n", "String 1\r\n", "String 2\r\n", "String 3\r\n"};

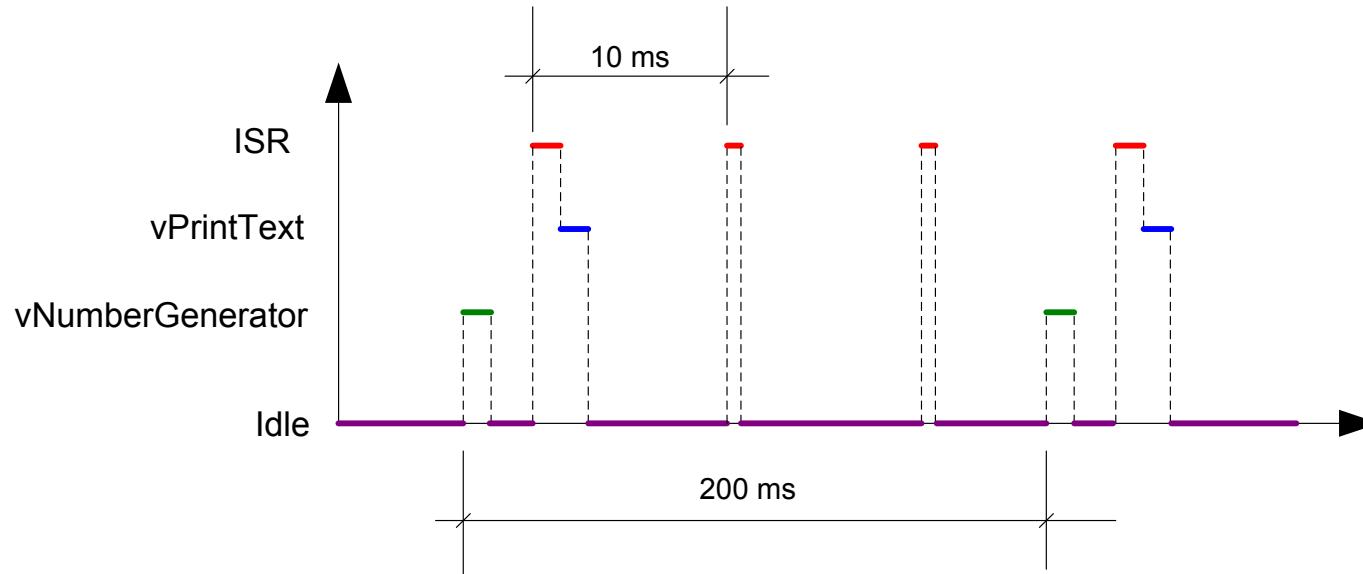
    xHigherPriorityTaskWoken = pdFALSE;
    // prekidna rutina mora se izvesti brzo - nema blokiranja na redu
    while(xQueueReceiveFromISR(xIntegerQueue, // isprazni red
                                &ulReceivedNumber, &xHigherPriorityTaskWoken) != errQUEUE_EMPTY) {
        ulReceivedNumber &= 0x03; // svedi primljene brojeve na interval
                                  // [0,3]
        // salji *adrese* pocetka odgovarajucih statickih poruka u
        // red xStringQueue; *ne kopiraju* se cijeli stringovi, vec
        // se samo proslijeduju pokazivaci (nema opasnosti od problema
        // dijeljenih resursa jer se ovdje radi o const nizu stringova)
        xQueueSendToBackFromISR(xStringQueue,
                               &pcStrings[ulReceivedNumber], &xHigherPriorityTaskWoken);
    }
    // da li je operacija slanja uzrokovala budjenje zadatka visoke
    // razine prioriteta?
    if( xHigherPriorityTaskWoken == pdTRUE ) {
        // eksplicitno pozovi zamjenu konteksta prije izlaska iz prekida!
        portSWITCH_CONTEXT();
    }
}

```

Primjer

```
// zadatak koji ispisuje primljene poruke
void vPrintText(void *pvParameters) {
    char *pcString;

    while(1)
    {
        // cekanje na tekstualnu poruku s beskonacnim blokiranjem
        xQueueReceive(xStringQueue, &pcString, portMAX_DELAY);
        vPrintString(pcString);
    }
}
```



Prekidi i jezgra OS-a

- FreeRTOS omogućuje fleksibilnost korištenja prekidnog sustava
- dvije su važne konstante kod definiranja razina prioriteta prekida:
 - *configKERNEL_INTERRUPT_PRIORITY* – prioritet prekida raspoređivača zadataka (jezgra OS-a, *tick interrupt*)
 - *configMAX_SYSCALL_INTERRUPT_PRIORITY* – najveća vrijednost prioriteta prekida koji koriste *interrupt-safe* API funkcije OS-a (funkcije koje su prilagođene pozivu iz prekida, a prepoznaju se po sufiksima *_FromISR*)
- važno je razlikovati prioritete zadataka od prioriteta prekida:
 - prioritete zadataka određuje jezgra OS-a
 - prioritete prekida određuju registri mikrokontrolera!

Primjer

```
configKERNEL_INTERRUPT_PRIORITY = 1
```

```
configMAX_SYSCALL_INTERRUPT_PRIORITY = 3
```



- prekidi razine prioriteta 1-3 neće se izvršavati unutar kritične sekcije i mogu pozivati *interrupt-safe* API funkcije
- prekidi razine 4-7 moći će se izvoditi bez obzira na kritičnu sekciju; tipična primjena je kada je potrebno održati vrlo striktne vremenske odnose (uzorkovanje, upravljanje motorima i sl.), uz pretpostavku da se ručno vodi računa da ti prekidi ne pristupaju globalnim resursima na neatomaran način
- prekidi koji ne pristupaju FreeRTOS API-ju i dijeljenim resursima mogu imati bilo koju razinu prioriteta

Upravljanje resursima

- problem pristupa dijeljenim resursima u više zadatačnom sustavu
 - zadatak započinje operaciju nad dijeljenim resursom i prije kraja operacije mijenja stanje iz *running* u *ready*
- primjeri:
 - pristup periferijskim jedinicama,
 - dijeljene globalne varijable,
 - pozivanje *non-reentrant* funkcija iz više zadataka ili zadatka i prekida itd.
- tehnike uzajamnog isključivanja – sinkronizacija pristupa dijeljenim resursima i očuvanje integriteta podataka

FreeRTOS kritične sekcije

- primjer realizacije kritične sekcije – pristup ulazno-izlaznom portu:

```
// onemogući ulazak drugih zadataka u kritičnu sekciju dok se upisuje vrijednost na
// I/O port
taskENTER_CRITICAL();
// zamjena konteksta onemogućena je između taskENTER_CRITICAL() i
// taskEXIT_CRITICAL(); prekidi su mogući, ali samo oni čiji je prioritet veći od
// configMAX_SYSCALL_INTERRUPT_PRIORITY; to su prekidi koji se smatraju sigurnima
// jer ne bi smjeli pristupati dijeljenim resursima i pozivati FreeRTOS API
// funkcije
PORTA |= 0x01;
// nakon obavljene operacije nad I/O portom - izađi iz kritične sekcije
taskEXIT_CRITICAL();
```

- pristup sličan primjeru prikazanom za sustav bez RTOS-a gdje je kritična sekcija realizirana maskiranjem prekida; u ovom slučaju RTOS automatski vodi računa o gniježđenju kritičnih sekcija (preko posebne interne varijable)!

FreeRTOS kritične sekcije

- primjer kritične sekcije – ispis stringa:

```
void vPrintString(const portCHAR *pcString) {
    taskENTER_CRITICAL();
    {
        printf("%s", pcString);
        fflush(stdout);
    }
    taskEXIT_CRITICAL();
}
```

- funkcija vPrintString() mogla bi se istovremeno pozivati iz više zadataka; kada se ne bi koristio mehanizam uzajamnog isključivanja pomoću kritične sekcije, funkcija ne bi bila *re-entrant!*

FreeRTOS kritične sekcije

- FreeRTOS kritična sekcija započinje pozivom funkcije `taskENTER_CRITICAL()`, a završava pozivom funkcije `taskEXIT_CRITICAL()`
- između dva poziva uparenih funkcija prekidi su onemogućeni (do određene razine), pa ne može niti doći do zamjene konteksta
- jednostavna i robusna metoda, ali ima nedostatak što u potpunosti isključuje mogućnost zamjene konteksta
- iz tog razloga nepogodna za primjenu u npr. slučaju ispisa niza znakova na serijski port, jer je to operacija koja može razmjerno dugo trajati i time značajno utjecati na odziv u *hard real-time* sustavu!

Alternativna implementacija FreeRTOS kritične sekcije

- kod realizacije k.s. pozivima taskENTER_CRITICAL()/
taskEXIT_CRITICAL() onemogućena je promjena tekućeg
zadatka i izvođenje prekida
- kritične sekcije i prekidi bi se trebali izvoditi što je moguće
brže
- u slučaju da se očekuje razmjerno dugo izvođenje kritične
sekcije, moguće je onemogućiti samo promjenu tekućeg
zadatka, uz omogućene prekide
- to je moguće postići *suspendiranjem raspoređivača zadataka*
pozivom API funkcije vTaskSuspendAll()

```
void vTaskSuspendAll(void);
```

Alternativna implementacija FreeRTOS kritične sekcije

- prekidi su omogućeni, ali ne mogu obaviti zamjenu konteksta (npr. pozivom funkcije *taskYield()*)
- raspoređivač zadataka se na kraju k.s. opet može omogućiti pozivom funkcije *xTaskResumeAll()*:

```
portBASE_TYPE xTaskResumeAll(void);
```

- povratna vrijednost – za vrijeme dok je raspoređivač zadataka onemogućen, moguće je da se pojave zahtjevi za zamjenom konteksta, koji se stavljuju u *pending* stanje; ukoliko je takvih zahtjeva za promjenom konteksta bilo i ako su izvršeni prilikom poziva funkcije *xTaskResumeAll*, funkcija vraća vrijednost pdTRUE, a pdFALSE inače
- FreeRTOS podržava ugniježđene pozive *vTaskSuspendAll()*/
xTaskResumeAll()

Alternativna implementacija FreeRTOS kritične sekcije

- primjer alternativne implementacije kritične sekcije – ispis stringa:

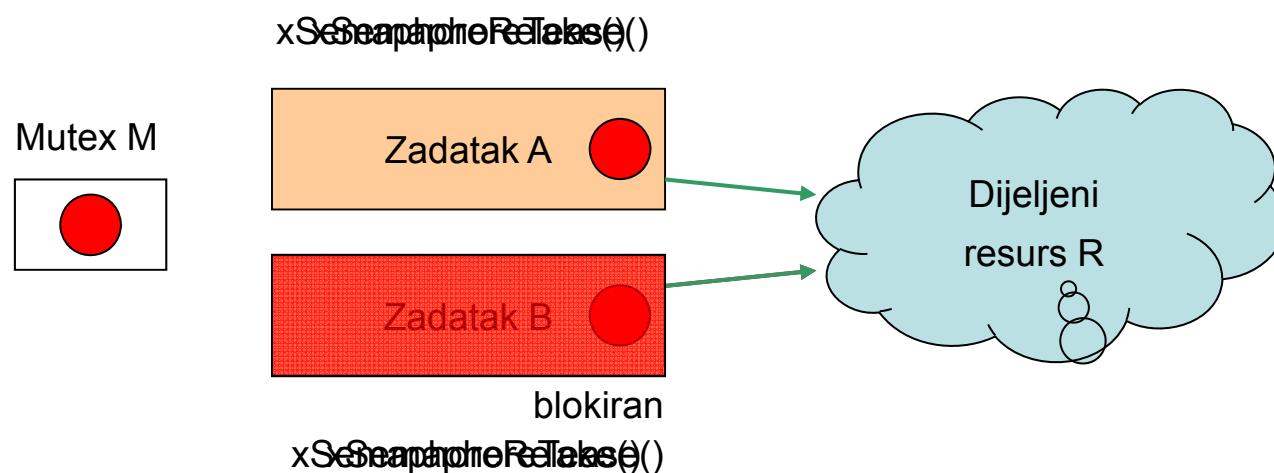
```
void vPrintString( const portCHAR *pcString )
{
    vTaskSuspendAll();
    {
        printf("%s", pcString);
        fflush(stdout);
    }
    xTaskResumeAll();
}
```

FreeRTOS i Mutex objekti

- obični binarni semafori pogodni su za sinkronizaciju aktivnosti između zadataka i implementaciju događaja, ali nisu idealni za zaštitu dijeljenih resursa:
 - problem rekurzivnog zauzimanja semafora,
 - problem vlasništva nad semaforom – bilo koji zadatak može pogreškom osloboditi semafor
- *mutex* objekt – dogovorni pristup zadataku dijeljenim resursima: samo zadatak koji je (rekurzivno) zaključao mutex može pristupati dijeljenim resursima, dok ostali čekaju da se (obavezno) otključa mutex od strane zadatka koji ga je zaključao

FreeRTOS i Mutex objekti

- primjer: dijeljenom resursu R pristupaju dva zadataka A i B; za sinkronizaciju pristupa resursu R koristi se mutex M:



FreeRTOS i Mutex objekti

- za pohranu handlea na mutex koristi se isti tip podataka kao i za obične semafore (`xSemaphoreHandle`)
- kreiranje mutex-a obavlja se pozivom API funkcije `xSemaphoreCreateMutex()`:

```
xSemaphoreHandle xSemaphoreCreateMutex(void);
```

- *povratna vrijednost* – *NULL* ako mutex nije kreiran zbog nedostatka memorije na *heapu*; vrijednost različita od *NULL* inače
- za zaključavanje i otključavanje mutex-a koriste se iste API funkcije kao i za obične semafore (`xSemaphoreTake()` i `xSemaphoreGive()`)
- FreeRTOS podržava rekurzivno zaključavanje mutex-a
- FreeRTOS mutex podržava protokol nasljeđivanja prioriteta

Primjer

```
void vPrintString(const portCHAR *pcString) {
    // beskonacno cekanje na mutex
    xSemaphoreTake(xMutex, portMAX_DELAY);
    // resurs "stdout" (npr. UART port) može se dobiti samo ako je uspješno zaključan mutex
    printf("%s", pcString);
    fflush(stdout);
    // mutex se mora obavezno otključati!
    xSemaphoreGive( xMutex );
}

void prvPrintTask(void *pvParameters) {
    char *pcStringToPrint;
    pcStringToPrint = (char *) pvParameters;
    while(1) {
        vPrintString(pcStringToPrint);    // pozovi reentrant funkciju za ispis
        vTaskDelay((rand() & 0x1FF));   // pricekaj vremenski period odredjen
                                         // slučajnim brojem
    }
}

xSemaphoreHandle xMutex;
int main(void) {
    xMutex = xSemaphoreCreateMutex(); // kreiraj mutex
    if(xMutex != NULL) {
        xTaskCreate(prvPrintTask, "Print1", 1000, "Task 1 message\r\n", 1, NULL);
        xTaskCreate(prvPrintTask, "Print2", 1000, "Task 2 message\r\n", 2, NULL);
        vTaskStartScheduler();
    }
    while(1);
}
```

Gatekeeper Task

- alternativni način rješavanja problema pristupa dijeljenim resursima, bez korištenja mutexa
- ideja – zadaci ne pristupaju izravno dijeljenom resursu, već posredno preko *gatekeeper* zadatka, kojem šalju poruke (putem OS *queue-a*)
- *gatekeeper* zadatak jedini pristupa izravno dijeljenom resursu i vodi računa od tome da svaku primljenu poruku riješi na atomaran način pa nema potrebe za posebnim sinkronizacijskim mehanizmom
- izbjegnuti problemi poput inverzije prioriteta i sl.

Primjer

- rješenje problema pristupa “stdio” sučelju iz više zadataka korištenjem *gatekeeper* zadatka

```
static char *pcStringsToPrint[] = {
    "Task 1 message\r\n",
    "Task 2 message\r\n"
};

xQueueHandle xPrintQueue;

int main(void) {
    xPrintQueue = xQueueCreate(5, sizeof(char *));
    if(xPrintQueue != NULL) {
        // kreiraj dva zadatka za ispis poruka i posalji kao parametre indekse
        // u globalnom polju poruka
        xTaskCreate(prvPrintTask, "Print1", 1000, (void *)0, 1, NULL);
        xTaskCreate(prvPrintTask, "Print2", 1000, (void *)1, 2, NULL);
        // kreiraj gatekeeper task (prioritet 0)
        xTaskCreate(vPrintStringGatekeeperTask, "Gatekeeper", 1000, NULL, 0, NULL );
        vTaskStartScheduler();
    }
    while(1);
}
```

Primjer

```
void vPrintStringGatekeeperTask(void *pvParameters)
{
    char *pcMessageToPrint;
    // ovo je jedini zadatak koji izravno pristupa "stdio"
    while(1) {
        // blokiraj na poruke drugih zadataka za pristupom stdio
        xQueueReceive(xPrintQueue, &pcMessageToPrint, portMAX_DELAY);
        // ispisi poruku na stdio
        printf("%s", pcMessageToPrint);
        fflush(stdout);
    }
}

void prvPrintTask(void *pvParameters) {
    int iIndexToString;
    iIndexToString = (int) pvParameters;
    while(1) {
        // posredni ispis tekstualne poruke na nacin da se ne zove izravno
        // funkcija koja ispisuje tekst na stdio, vec se salje poruka o tome
        // sto se zeli ispisati stdio gatekeeper tasku (pokazivac na string)
        xQueueSendToBack(xPrintQueue, &(pcStringsToPrint[iIndexToString]), 0);
        vTaskDelay((rand() & 0x1FF));
    }
}
```

Dinamičko upravljanje memorijom

- objekti jezgre OS-a alociraju se **dinamički** na heap-u (zadatak, red, semafor...)
- standardne implementacije *malloc()/free()* funkcija nisu uvijek dobar odabir kod sustava s vrlo ograničenim resursima:
 - nedeterminističko ponašanje funkcija, fragmentacija memorije, implementacije uglavnom nisu *thread-safe*, zauzeće memorije (kôd za implementaciju funkcija), složenija konfiguracija linkera itd.
- FreeRTOS - nudi mogućnost odabira optimalnog načina dinamičkog upravljanja memorijom u ovisnosti o konkretnom slučaju, preko portabilnih API funkcija *pvPortMalloc()* i *pvPortFree()*

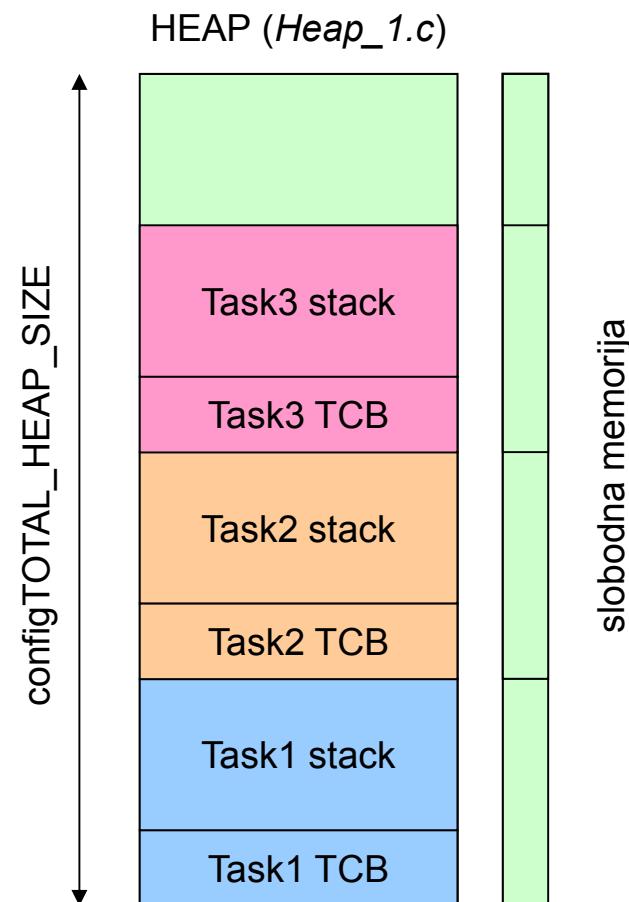
Dinamičko upravljanje memorijom

- funkcije *pvPortMalloc()* i *pvPortFree()* imaju isti prototip kao *malloc()* i *free()* funkcije
- FreeRTOS nudi tri implementacije (koje pokrivaju neke tipične slučajeve) u datotekama *heap_1.c*, *heap_2.c* i *heap_3.c* u direktoriju “*FreeRTOS\Source\Portable\MemMang*”
 - u svakoj datoteci drugačije su implementirane funkcije *pvPortMalloc()* i *pvPortFree()*
 - korisnik kod konfiguriranja projekta određuje koja se implementacija koristi
 - korisnik može jednostavno dodati i vlastite implementacije algoritama za dinamičko upravljanje memorijom
- rane verzije FreeRTOS-a koristile su *memory pool block allocation* pristup, koji se ne koristi kod novijih verzija

Algoritmi dinamičke alokacije memorije

- *Heap_1.c*
 - najjednostavniji algoritam izravne dodjele blokova bez mogućnosti oslobođanja memorije
 - funkcija pvPortFree() nije implementirana!
 - deterministički algoritam
 - configTOTAL_HEAP_SIZE u FreeRTOSConfig.h definira (statički) veličinu polja za implementaciju heap-a
- ovakav pristup pogodan je za slučajeve kada se prilikom pokretanja OS-a inicijaliziraju svi zadaci i objekti OS-a, bez naknadnog brisanja
 - unaprijed se rezervira prostor na heapu kojeg kasnije nije potrebno oslobođati
 - kompaktan kôd, nema problema s determinizmom izvođenja programa, fragmentacijom memorije i sl.

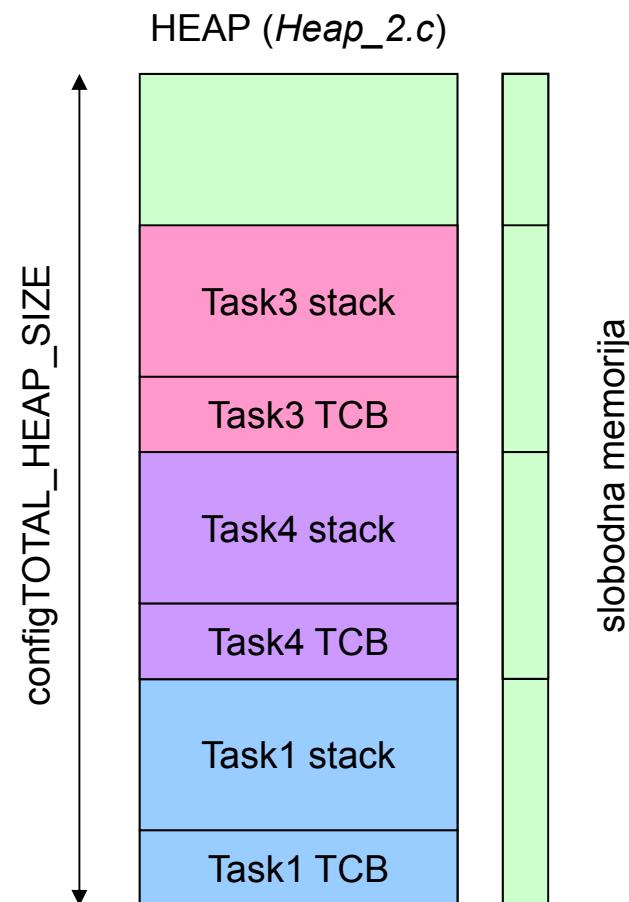
Primjer



Algoritmi dinamičke alokacije memorije

- *Heap_2.c*
 - koristi tzv. “*best-fit algoritam*” dinamičke alokacije memorije, a omogućuje i oslobođanje memorije
 - “*best-fit*” algoritam: traži slobodni blok koji je po veličini najbliži zahtijevanom memorijskom bloku
 - primjer: raspoloživa su tri slobodna bloka (50, 250 i 1000 okteta), a traži se memorija za niz od 200 okteta:
 - algoritam će zauzeti 200 okteta iz bloka s 250 slobodnih okteta i od preostalih 50 formirati novi slobodni blok (smanjenje interne fragmentacije!),
 - oslobođanje zauzetog bloka od 200 okteta rezultirat će s dva slobodna bloka - 200 okteta i 50 okteta
 - za razliku od standardne implementacije *free()* funkcije, FreeRTOS Heap_2.c algoritam **ne spaja** susjedne slobodne blokove! – problem eksterne fragmentacije!

Primjer



Algoritmi dinamičke alokacije memorije

- *Heap_2.c*
 - algoritam je pogodan u situacijama kada se dinamička memorija koristi na način da se uvijek oslobađaju i zauzimaju blokovi memorije **iste veličine**
 - npr. nakon početne inicializacije OS-a zadaci se naknadno dinamički kreiraju i brišu
 - TCB i stog mogu biti odabrani tako da za sve zadatke zauzimaju isti prostor na heap-u
 - *heap_2* algoritam vrlo je učinkovit u tom slučaju – nema problema s fragmentacijom memorije
 - iako *heap_2* algoritam nije deterministički, pokazuje puno bolje rezultate od korištenja standardnih implementacija *malloc()* i *free()* funkcija

Algoritmi dinamičke alokacije memorije

- *Heap_3.c*
 - algoritam koji koristi standardnu implementaciju *malloc()* i *free()* funkcija, ali na *thread-safe* način tako da se prilikom poziva tih funkcija privremeno onemogući raspoređivač zadataka
 - za razliku od *Heap_1* i *Heap_2* algoritama, koji cijelu dinamičku memoriju upravljaju unutar statičkog polja veličine `configTOTAL_HEAP_SIZE`, ta konstanta nema utjecaja na *Heap_3* algoritam, već se veličina heap-a određuje izravno kroz konfiguraciju linkera

Implementacija Heap_3 algoritma

```
void *pvPortMalloc(size_t xWantedSize) {
    void *pvReturn;
    vTaskSuspendAll();
    pvReturn = malloc(xWantedSize);
    xTaskResumeAll();
    return pvReturn;
}

void vPortFree(void *pv) {
    if(pv != NULL) {
        vTaskSuspendAll();
        free(pv);
        xTaskResumeAll();
    }
}
```

Literatura

- R. Barry: Using the FreeRTOS Real Time Kernel - Standard Edition, FreeRTOS.org, 2009, ISBN 978-1446169148
- www.freertos.org

Fakultet elektrotehnike i računarstva
Zavod za elektroničke sustave i obradbu informacija

Programska potpora industrijskih ugradbenih sustava

Embedded Linux

Hrvoje Džapo, Mario Cifrek

ak. god. 2014./2015.

Operacijski sustavi za ugradbena računala

- operacijski sustavi za rad u stvarnom vremenu (RTOS)
 - tipična primjena: *hard real-time, kritični sustavi*
 - vrlo ograničeni sklopovski resursi (RAM, ROM, procesorska snaga, potrošnja, dimenzije, upravljanje memorijom, komunikacije itd.)
 - tipični minimalni memorijski otisak – nekoliko kB do nekoliko 10 kB (jezgra)
 - primjeri: FreeRTOS, RTX, µC/OS-II, µC/OS-III, QNX Neutrino, Contiki, Nucleus OS, ThreadX, VxWorks, eCos...
- operacijski sustavi opće namjene (GPOS)
 - tipična primjena: *feature-rich* aplikacije, GUI, potrošačka elektronika, industrija itd.
 - ugradbeni računalni sustavi temeljeni na “*high-end*” sklopovlju
 - Linux, Windows CE, iOS, uLinux, Symbian itd.
 - bitniji širok raspon mogućnosti i interoperabilnost sustava nego rad u okviru *hard real-time* ograničenja

Linux

- pojam koji se koristi za **jezgru (kernel)**, **sustav** i **distribucije**
- Linux jezgra – projekt započet 1991. (Linus Torvalds) (kernel.org)
 - implementacija Unix operacijskog sustava otvorenog koda
 - velika i aktivna razvojna zajednica
- sustav – jezgra nije dovoljna kao cjelina bez pratećih programske komponenti
 - npr. bootloader, biblioteke, pomoćni programi, korisničke aplikacije, grafičko sučelje (X window system) itd.
 - visoko modularan pristup izgradnji sustava – složenost!
- distribucije – paketi s različitim komponentama sustava koji olakšavaju instalaciju OS-a na određenu platformu
 - jezgra, bootloader, biblioteke, pomoćni programi i sl.
 - npr. Red Hat, Debian, Ubuntu, SuSE, Slackware, Caldera, MontaVista, BlueCat, itd.

Linux

- FSF (*Free Software Foundation*), FOSS (*Free and Open Source Software*)
- licenca slobodnog otvorenog koda na kojoj je temeljen Linux:
GPL (GNU General Public License) (Richard Stallman)
 - kôd temeljen na drugom GPL kôdu mora biti dostupan pod istom licencicom
 - ideja: omogućiti korisniku uvid u ponašanje sustava i mogućnost nadogradnji te ispravljanja pogrešaka
 - GPL licenca – *copyleft* (kôd nije public domain, shareware i sl.)
 - otvoreni kôd ne znači nužno i besplatan
 - kôd koji nije izведен iz GPL-a može ostati zatvoren
- druge licence otvorenog koda: **LGPL (Library/Lesser GPL** (npr. za C biblioteke), **BSD**, **MIT** itd.)

Embedded Linux

- Linux nije izvorno zamišljen kao operacijski sustav za URS, već kao OS opće namjene kojeg je zbog otvorenosti koda moguće prilagoditi različitim arhitekturama i sklopovskim konfiguracijama
- razvoj složenijih, bržih i jeftinijih procesora, kao i pad cijena memorijskih modula, omogućio je upotrebu takvog sustava i u *embedded* okruženju
 - minijaturni sustavi s niskom potrošnjom, ali najčešće ne i *hard real-time* zahtjevima
- nema posebne “*embedded*” jezgre Linux-a!
 - prilagodba jezgre posebnostima i ograničenja ugradbenih računalnih sustava
 - fleksibilnost u izgradnji cjelokupnog OS-a (mogućnost konfiguriranja parametara jezgre OS-a i pratećih programske modula koji se uključuju u jezgru i korijenski datotečni sustav)
- prednost – programska potpora otvorenog koda lako se može prevesti za različite ciljne ugradbene arhitekture (*cross-compiling*)
 - dramatične uštede u vremenu razvoja uz korištenje standardnih i dobro testiranih programske rješenja!

uClinux

- grana Linux kernela za mikrokontrolere bez MMU jedinice
 - značenje imena: **MicroController Linux** [“you-see-Linux”]
 - izvorno razvijen za Motorolin 68k (68EZ328 DragonBall) 1998. na inačici 2.0.33 Linux kernela
- podržane platforme: Altera NIOS, Blackfin, ARM ARM7TDMI, Freescale m68k (Motorola 68000), Hitachi H8, MIPS, Motorola ColdFire, Xilinx MicroBlaze itd.
- razvijeni portovi temeljeni na 2.4 i 2.6 verzijama jezgre Linuxa
- od verzije jezgre 2.5.46 integracija s glavnom razvojnom linijom kernela koji sada podržava i MMU-less procesore
- osim jezgre, u okviru projekta razvijena i zasebna verzija C biblioteka (uClibc) i dodaci jezgri koji čine cijeli OS (“uClinux-dist”)
 - uClinux-dist sadrži biblioteke, pomoće aplikacije i alate



NetBSD

- <http://netbsd.org>
- temeljen na BSD (*Berkeley Software Distribution*) varijantama Unix implementacija
 - u tu grupu spadaju npr. FreeBSD, OpenBSD itd.
 - BSD-style FOSS licenca – razlikuje se od GPL-a po tome što se kôd temeljen na BSD kôdu ne mora objavljivati pod istom licencem – lakša upotreba kôda u npr. zatvorenim komercijalnim sustavima
 - Mac OS X , iOS bazirani na FreeBSD-u, a ne na Linuxu (na temelju kojeg je npr. razvijen Android OS)!
- NetBSD podržava 57 platformi (16 procesorskih arhitektura)
- sklopovski zahtjevi slični kao i za Embedded Linux
- manja, ali vrlo aktivna razvojna zajednica

eCos

- <http://ecos.sourceforge.org/>
- eCos - inicijalni razvoj *Cygnus Solutions* (kasnije Red Hat); prebačen u FSF 2005. (pod GPL-kompatibilnom licencom)
 - eCosPro – komercijalna grana (eCosCentric)
- RTOS koji podržava scenarij **jednog višenitnog procesa** (aplikacije) (*single multithreaded process*)
- memorijski zahtjevi: nekoliko 10-100 kB (min. 50 kB!), podrška za 16-bitne procesore
 - Linux: 1-2 MB minimalno!
- rad u stvarnom vremenu!
- HAL model (*device drivers*)
- podržane platforme: ARM, FR-V, Hitachi H8, IA-32, Motorola 68000, MIPS, NEC V8xx, Nios II, PowerPC, SPARC, SuperH itd.

Zašto (embedded) Linux?

- lakši razvoj – dostupan veliki broj gotovih programskih modula (u izvornom kodu!), npr.:
 - standardne C biblioteke (nisu integralni dio GCC-a, već je moguće odabrati implementaciju koja najviše odgovara ciljnoj primjeni)
 - *Berkley sockets* (TCP/IP *networking*) i gotovo svi postojeći komunikacijski protokoli
 - web serveri (apache, boa, lighttpd, micro_httpd...)
 - SSL/SSH (sigurna komunikacija)
 - itd.

Zašto (embedded) Linux?

- sukladnost standardima i interoperabilnost
 - *Portable Operating System Interface for Unix (POSIX)*
IEEE standardizirani API za upravljanje nitima i komunikacijom između procesa (*pthreads*)
- upravljanje procesima
 - proces – zasebni program koji se izvodi u *multitasking* okruženju
 - izolacija procesa međusobno i od jezgre (korištenjem virtualne memorije) – stabilnost i pouzdanost
 - višenitni procesi (*multithreading*)
 - više paralelnih linija izvođenja unutar jednog procesa (slično taskovima kod RTOS-a)

Zašto (embedded) Linux?

- upravljanje memorijom
 - virtualna memorija – zahtijeva MMU
 - svaki proces “vidi” da mu je dodijeljena sva memorija i da započinje na adresi 0x00..000
 - zaštita memorije (procesi + jezgra) – procesi ne mogu izravno pristupati fizičkim memorijskim lokacijama
 - omogućuje korištenje *backing store-a* u slučaju da je potrebno više radne memorije od fizički raspoložive
 - pristup stranici koja nije u memoriji uzrokuje njezino učitavanje s diska
 - ipak, URS najčešće ne koriste tu mogućnost, uglavnom koriste translaciju adresa i zaštitu memorije

Zašto (embedded) Linux?

- uniformi pristup resursima
 - Linux koristi datotečni sustav kao uniformnu apstrakciju pristupa različitim resursima (uredajima, statusnim podacima jezgre i sl.)
 - filozofija “*everything is a file*”
- sistemski pozivi
 - korisnički programi mogu pozivati servise operacijskog sustava putem sistemskih poziva (*syscall*)
 - korisnički programi ne mogu utjecati izravno na izvođenje jezgrinih servisa i interne podatkovne strukture OS-a

Zašto (embedded) Linux?

- ugrađena podrška za veliki broj različitih periferija
- sigurnost (duboko ugrađena u OS – izvorno *multiuser* OS)
- komercijalni razlozi:
 - niži troškovi razvoja (velika količina dostupnog kvalitetnog koda i programskih paketa)
 - portabilnost, kontrola i fleksibilnost
 - razvoj za različite platforme (dostupan izvorni C kod)
 - izgradnja sustava za vrlo različite sklopovske platforme (*desktop, server, embedded – iz istog koda!*)
 - royalty-free
- napomena - potreban oprez kod korištenja GPL koda u komercijalnim primjenama!
 - sve nadogradnje moraju biti otvorenog koda!

Minimalni sklopovalski zahtjevi

- CPU podržan od GCC prevoditelja i Linux kernela
 - obično nije problem, GCC i kernel imaju podršku za veliki broj procesorskih arhitektura
 - tipični odabir - 32-bitni procesor s MMU jedinicom
 - MMU jedinica neophodna za realizaciju virtualne memorije, razdvajanje jezgre i korisničkih aplikacija itd.
 - procesori bez MMU – uCLinux ili novije verzije kernela
 - jeftiniji sustavi, moguće i brže izvođenje programa (jer nema dodatnog overhead-a s virtualnom memorijom), ali slabija pouzdanost cijelog sustava
- radna memorija (RAM) – min. 2-4 MB
- medij za pohranu podataka – min. 2-4 MB

Osnovne značajke i mogućnosti OS-a

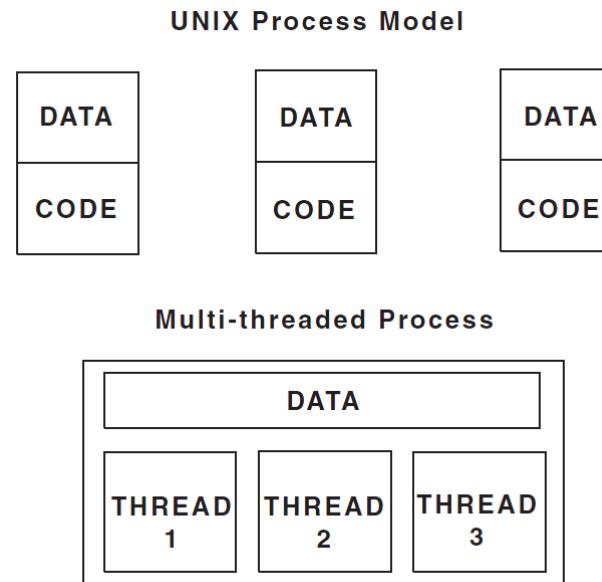
- *Multitasking* – mogućnost istovremenog izvođenja više programa (“procesa” u kontekstu Linux-a)
 - *preemptive multitasking* – **procesi** više razine prioriteta mogu istisnuti procese niže razine
 - znači li to da se Linux može smatrati RTOS-om?
 - NE! – **procesi** ne mogu istisnuti jedan drugog za vrijeme izvođenja sistemskog poziva!
 - Linux nema *preemptive* jezgru kakvu ima tipični RTOS!
 - sistemski poziv jezgri može trajati prilično dugo (za pojmove *hard real-time* sustava) i nije ga moguće prekinuti!
 - koristi tzv. CFS (*Complete Fair Scheduling*) algoritam, koji je pogodan za npr. desktop sustave s GUI-jem, ali je u potpunoj suprotnosti s ciljevima sustava za rad u stvarnom vremenu!
 - postoje *real-time* ekstenzije Linux-a – nije dio standardne jezgre

Osnovne značajke i mogućnosti OS-a

- *Multiuser* – višekorisnički OS, paradigma duboko ugrađena u OS (od ranih dana UNIX se koristio za pristup više korisnika oskudnim računalim resursima)
- *Multiprocessing (SMP – symmetric multiprocessing)*
 - dva ili više procesora priključeno na istu dijeljenu memoriju u sustavu upravljanom istim operacijskim sustavom
- *Virtualna memorija*
- *Hijerarhijski datotečni sustav*
 - jedan od najvažnijih UNIX koncepata
 - osim za datoteke, koristi se i za pristup U/I uređajima i jezgri

Model Linux procesa

- proces – kôd + resursi (podaci, opisnici datoteka i sl)
- zaštita procesa (virtualna memorija)
- IPC (inter-process communication) – dijeljena memorija (*shared-memory regions*), *pipes*
- nit (*thread*) – unutar memorijskog prostora procesa (samo kod)
 - manji *overhead*, “*lightweight multitasking*”



Linux datotečni sustav

- hijerarhijski datotečni sustav, s ishodištem u korijenskom čvoru (“root”) “/”
 - case-sensitive imena datoteka
 - ekstenzije pod Linuxom nemaju čvrsto značenje
 - “.” na početku – “skrivene” datoteke (može ih se vidjeti putem “ls -a”)
- *Linux* korisnike pridružuje jednoj ili više grupa
 - u skladu s tim, svaka datoteka u datotečnom sustavu ima vlasnika (*ownership*) i dozvole (*permissions*)

Linux datotečni sustav

- provjera dozvola: “ls -l”
- tri tipa dozvola:
 - read access (**r**), write access (**w**), exec rights (**x**)
- tri razine dozvola:
 - user (**u**), group (**g**), others (**o**)

Tip	r	w	x	r	w	x	r	w	x
	user			group			others		

- primjeri:
 - rw-r--r-- vlasnik ima R/W prava, ostali samo R
 - rw-r----- vlasnik ima R/W prava, članovi grupe samo R, ostali ništa
 - drwx----- direktorij kojem može pristupiti samo vlasnik

Linux datotečni sustav

- promjena dozvola:

chmod <permissions> <files>

- npr.

chmod 554 <file> (101 101 100 = -r-x r-x r--)

chmod 777 <file> (sva prava svima) (-rwxrwxrwx)

chmod go+r <file> (*group, others* – dodaj R pravo)

chmod u-w <file> (*user* – oduzmi W pravo)

chmod a-x <file> (*all* – oduzmi X pravo)

Linux datotečni sustav

- promjena vlasništva:

```
chown <user> <files>
```

```
chown <user>:<group> <files>
```

- npr.

```
chown -R user /home/linux/src
```

```
chown -R user:group /home/apps
```

Root korisnik

- ima sva prava – bez obzira na dozvole datoteka
 - administracija sustava
- nije preporučljivo da se obične aplikacije pokreću od strane *root* korisnika
- kada treba obaviti sistemsku operaciju:
 - su (switch user)
 - sudo (omogućuje korisniku pokretanje programa s privilegijama drugog korisnika (tipično root))

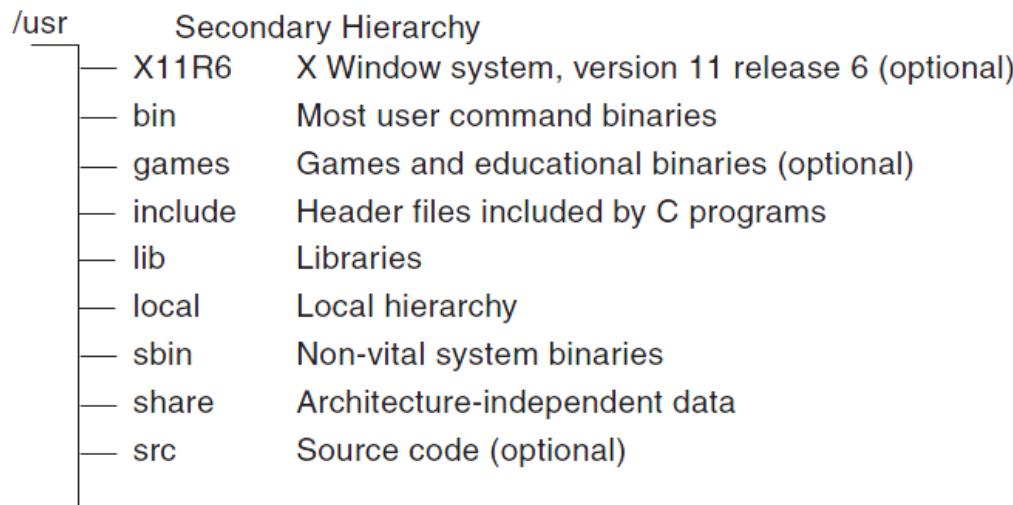
File System Hierarchy (FSH)

- standard za “UNIX-like” operacijske sustave
 - u različitim distribucijama postoje određena manja odstupanja
- glavna hijerarhija:

/	the root directory
bin	Essential command binaries
boot	Static files of the boot loader
dev	Device "inode" files
etc	Host-specific system configuration
home	Home directories for individual users (optional)
lib	Essential shared libraries and kernel modules
mnt	Mount point for temporarily mounting a filesystem
opt	Additional application software packages
root	Home directory for the root user (optional)
sbin	Essential system binaries
tmp	Temporary files
usr	Secondary hierarchy
var	Variable data

File System Hierarchy (FSH)

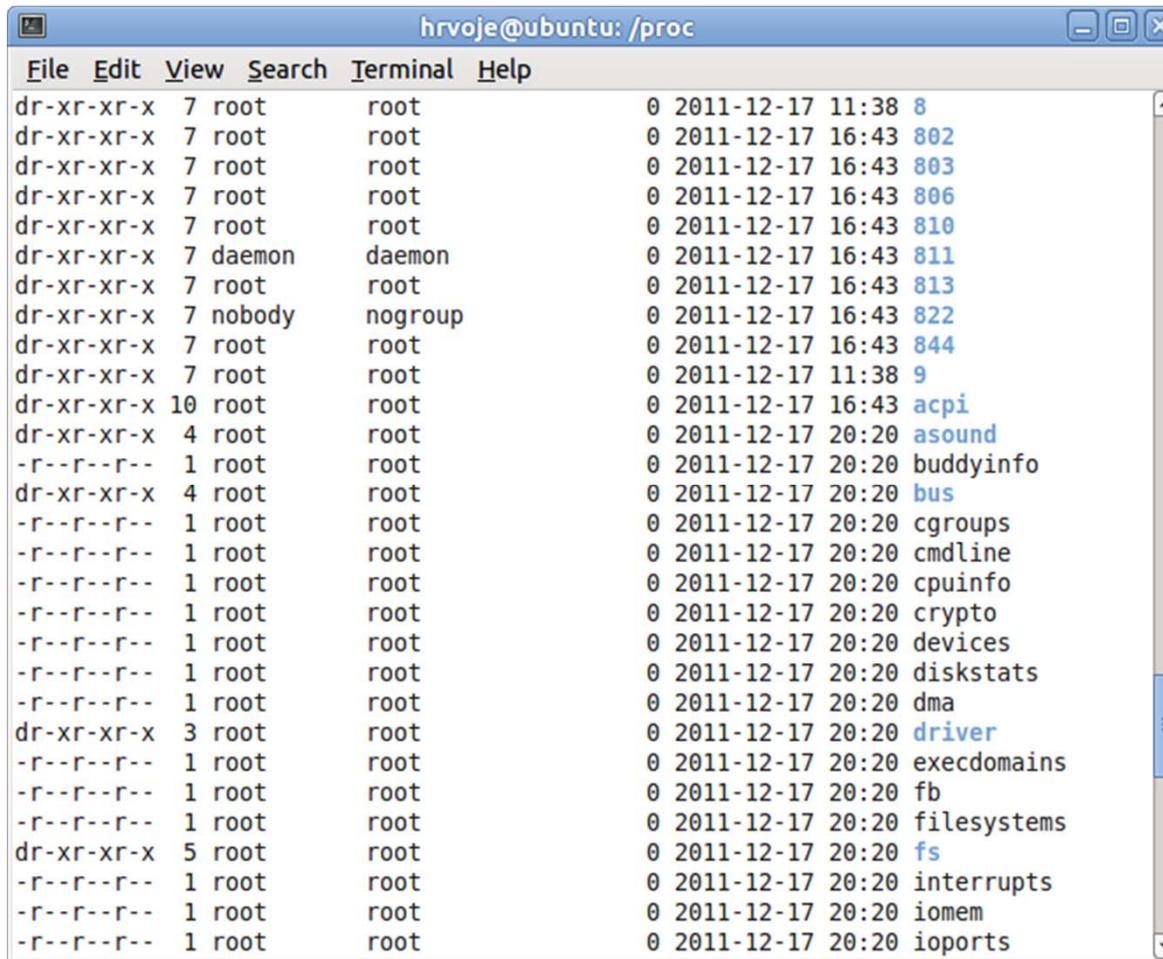
- sekundardna hijerarhija:



- /usr/src – izvorni kod jezgre

/proc filesystem

- /proc direktorij – omogućuje izravan uvid u jezgru preko *on-the-fly* generiranog virtualnog datotečnog sustava



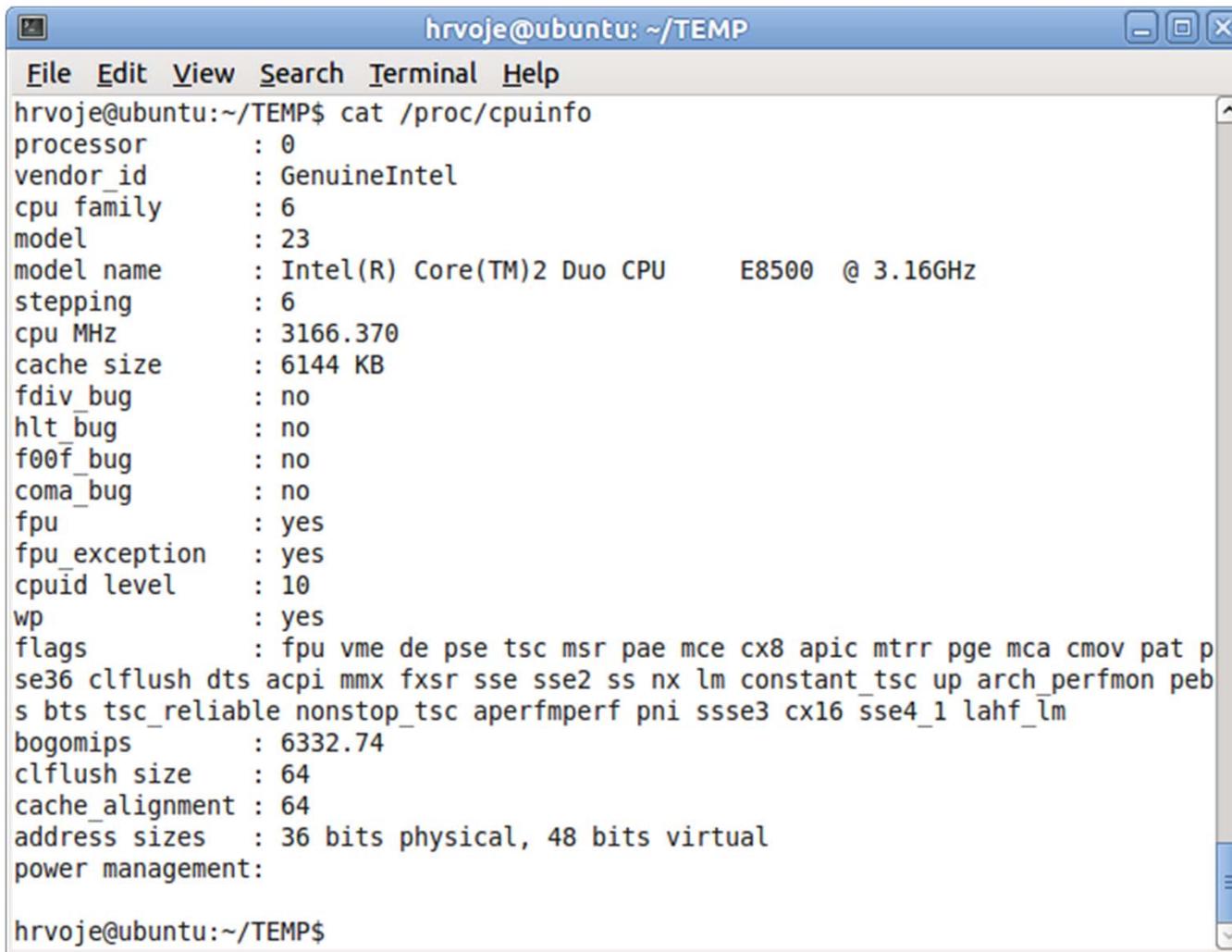
```

hrvoje@ubuntu: /proc
File Edit View Search Terminal Help
dr-xr-xr-x 7 root      root          0 2011-12-17 11:38 8
dr-xr-xr-x 7 root      root          0 2011-12-17 16:43 802
dr-xr-xr-x 7 root      root          0 2011-12-17 16:43 803
dr-xr-xr-x 7 root      root          0 2011-12-17 16:43 806
dr-xr-xr-x 7 root      root          0 2011-12-17 16:43 810
dr-xr-xr-x 7 daemon    daemon        0 2011-12-17 16:43 811
dr-xr-xr-x 7 root      root          0 2011-12-17 16:43 813
dr-xr-xr-x 7 nobody    nogroup      0 2011-12-17 16:43 822
dr-xr-xr-x 7 root      root          0 2011-12-17 16:43 844
dr-xr-xr-x 7 root      root          0 2011-12-17 11:38 9
dr-xr-xr-x 10 root     root         0 2011-12-17 16:43 acpi
dr-xr-xr-x 4 root     root          0 2011-12-17 20:20 asound
-r--r--r-- 1 root     root          0 2011-12-17 20:20 buddyinfo
dr-xr-xr-x 4 root     root          0 2011-12-17 20:20 bus
-r--r--r-- 1 root     root          0 2011-12-17 20:20 cgroups
-r--r--r-- 1 root     root          0 2011-12-17 20:20 cmdline
-r--r--r-- 1 root     root          0 2011-12-17 20:20 cpuinfo
-r--r--r-- 1 root     root          0 2011-12-17 20:20 crypto
-r--r--r-- 1 root     root          0 2011-12-17 20:20 devices
-r--r--r-- 1 root     root          0 2011-12-17 20:20 diskstats
-r--r--r-- 1 root     root          0 2011-12-17 20:20 dma
dr-xr-xr-x 3 root     root          0 2011-12-17 20:20 driver
-r--r--r-- 1 root     root          0 2011-12-17 20:20 execdomains
-r--r--r-- 1 root     root          0 2011-12-17 20:20 fb
-r--r--r-- 1 root     root          0 2011-12-17 20:20 filesystems
dr-xr-xr-x 5 root     root          0 2011-12-17 20:20 fs
-r--r--r-- 1 root     root          0 2011-12-17 20:20 interrupts
-r--r--r-- 1 root     root          0 2011-12-17 20:20 iomem
-r--r--r-- 1 root     root          0 2011-12-17 20:20 ioports

```

/proc filesystem

- npr. sadržaj datoteke cpuinfo: cat /proc/cpuinfo



```
hrvoje@ubuntu:~/TEMP$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 23
model name     : Intel(R) Core(TM)2 Duo CPU      E8500 @ 3.16GHz
stepping        : 6
cpu MHz        : 3166.370
cache size     : 6144 KB
fdiv_bug       : no
hlt_bug        : no
f00f_bug       : no
coma_bug       : no
fpu            : yes
fpu_exception  : yes
cpuid level   : 10
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic mttr pge mca cmov pat p
se36 clflush dts acpi mmx fxsr sse sse2 ss nx lm constant_tsc up arch_perfmon peb
s bts tsc_reliable nonstop_tsc aperfmpf perfmon_pni ssse3 cx16 sse4_1 lahf_lm
bogomips      : 6332.74
clflush size   : 64
cache_alignment : 64
address sizes  : 36 bits physical, 48 bits virtual
power management:

hrvoje@ubuntu:~/TEMP$
```

Interaktivna komandna ljudska (*Shell*)

- interpreter naredbi, podrška za skripte (*shell scripting*)
- važno kod ugradbenih računalnih sustava – mogućnost kontrole sustava putem serijskog terminala (npr. RS-232) ili udaljeno putem TCP/IP veze (telnet, openssh)
- npr. *Bourne Again Shell (bash)*, *C Shell (csh)*, *Z Shell (zsh)* itd.

Embedded Linux razvojni scenariji

- vrlo složeni sklopovski i programski sustavi
- tri razine razvoja pod Embedded Linux-om:
 - *BSP (Board Support Package) Development*
 - prilagodba jezgre Linux-a ciljnom sklopovlju
 - tipično rade proizvođači procesora i razvojnih ploča, rijetko inženjeri koji rade rješenja za krajnje korisnike
 - *System Integration*
 - integracija bootloadera, jezgre, biblioteka, aplikacije i datotečnog sustava za ciljno sklopovlje i primjenu
 - nužan korak u razvoju rješenja za krajnjeg korisnika – korisno poznавање elemenata BSP razvoja
 - *Application Development*
 - razvoj aplikacija na konačnom uhodanim sustavu (C/C++, skriptni jezici i sl.)
 - zahtijeva minimalno poznавање implementacijskih detalja Linux-a niske razine, ne razlikuje se puno od razvoja „*non-embedded*“ aplikacija (osim po prirodi zahtijeva i specifikacija)

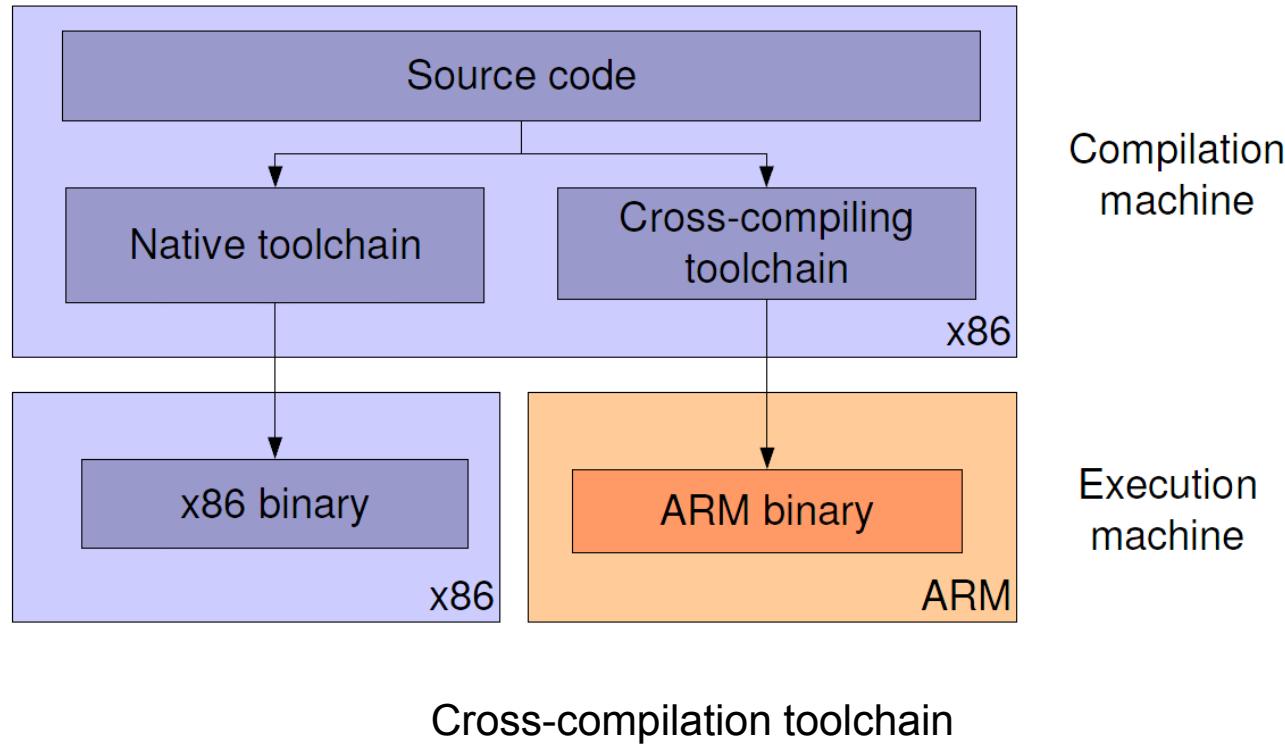
Programske komponente za izgradnju Embedded Linux sustava

- *Cross-compilation toolchain*
 - prevoditelj koji se izvodi na radnom računalu, a generira izvršni kod za drugu procesorsku platformu
 - neophodan za izgradnju jezgre, modula i aplikacija
- *Bootloader*
 - program koji se izvodi odmah nakon reseta sklopoljja, provodi početnu inicijalizaciju sklopoljja, učitava u memoriju i pokreće jezgru OS-a
 - zbog vrlo ograničenih resursa, ne koriste se uobičajeni Linux *bootloaderi* (npr. LILO, GRUB), već posebno prolagodeni za URS-eve (npr. U-Boot)

Programske komponente za izgradnju Embedded Linux sustava

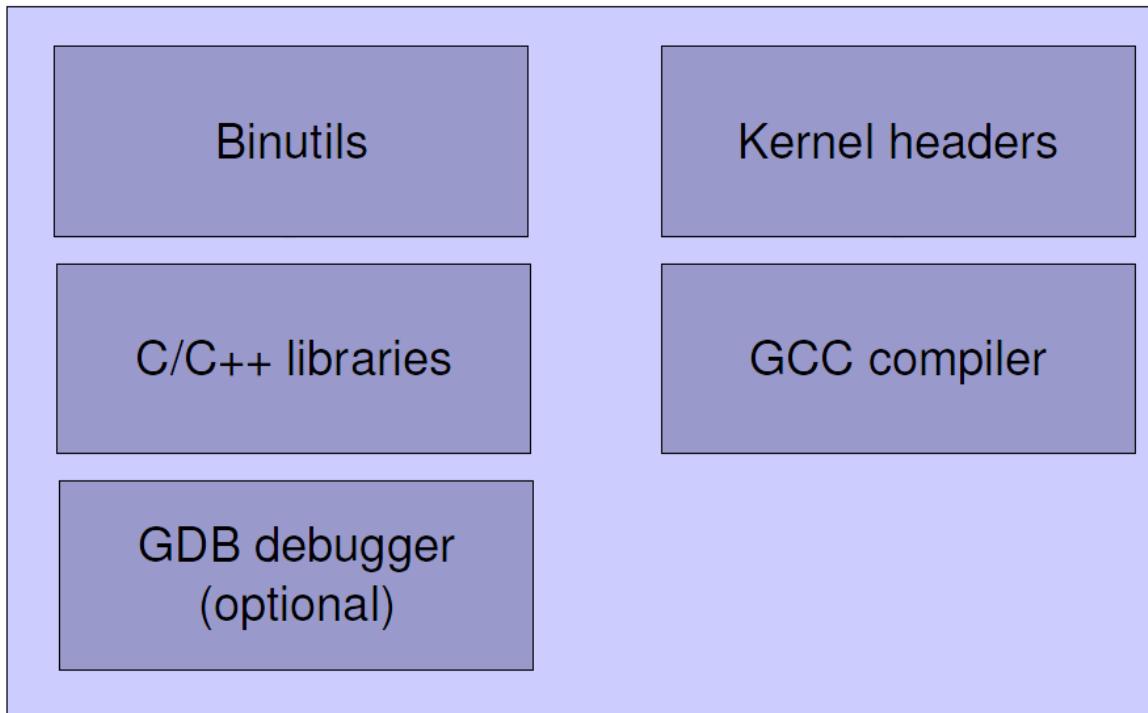
- *Linux Kernel*
 - izvorni kôd jezgre koja se prevodi za ciljnu platformu
 - upravljanje procesima i memorijom, komunikacija između procesa, mrežni protokoli, upravljački programi, sistemski servisi za korisničke aplikacije itd.
- *C library*
 - sučelje između jezgre i korisničkih aplikacija
- Biblioteke i dodatne aplikacije

Programske komponente



- izgradnja vlastitog toolchain-a – konfigurabilnost i fleksibilnost
- gotovi toolchain (*prebuilt*) (npr. CodeSourcery)

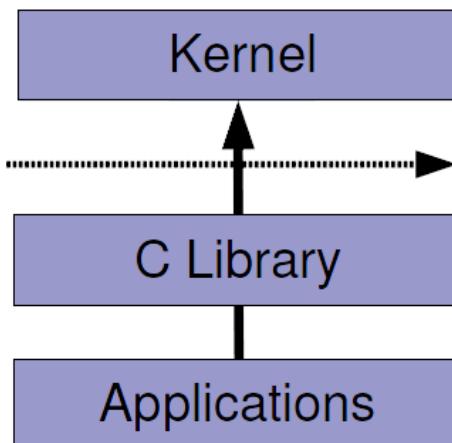
Programske komponente



- sve komponente su razdvojene – npr. debugger nije dio GCC-a, postoji nekoliko implementacija standardnih C/C++ biblioteka i sl.

Programske komponente

- *binutils* - alati za generiranje i manipulaciju binarnih izvršnih datoteka (asembler, linker, *strip*, *objdump*...)
- C biblioteke – nisu sastavni dio GCC-a!
 - sučelje aplikacija-jezgra
 - standardni C API – lakši razvoj aplikacija
 - nekoliko implementacija standardne C biblioteke: glibc, uClibc, eglIBC, dietlibc, newlib itd.

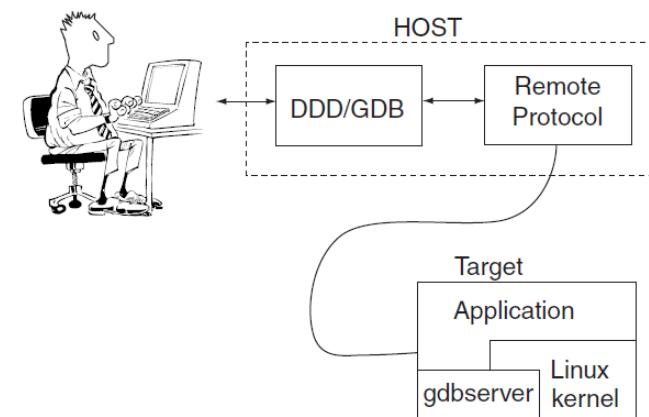
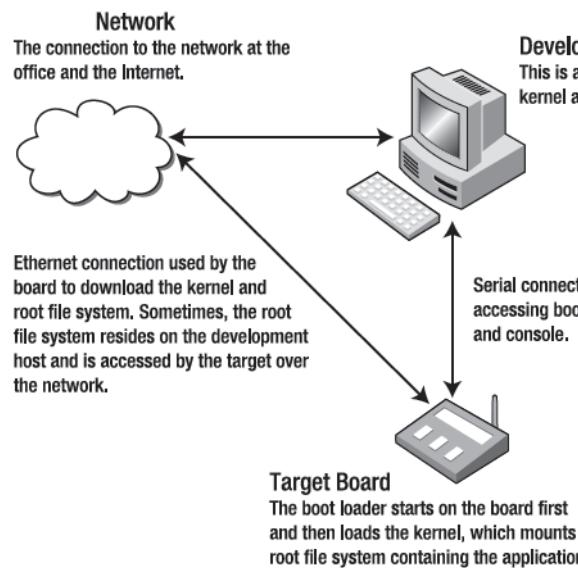


C biblioteke

- glibc - GNU projekt, najrasprostranjenija biblioteka
 - zauzeće koda: oko 2.5 MB (ARM)
 - najaktivnija podrška razvojne zajednice
 - uClibc – konfigurable, optimirana za URS
 - zauzeće koda: oko 600 kB
 - podrška - MontaVista, TimeSys i Wind River
 - eglIBC (“*embedded*” glibc) – varijanta glibc prilagođena URS
 - podrška - Freescale, MIPS, MontaVista i Wind River
 - dietlibc
 - zauzeće koda: oko 70 kB
 - newlib
 - klibc
- itd.

Razvojno okruženje

- računala koja čine razvojni sustav
 - **build machine** (izgradnja alatnog lanca)
 - **host machine** (razvojno računalo)
 - **target machine** (ciljno (*embedded*) sklopljje)



Struktura minimalnog Embedded Linux sustava

- *Bootloader* – pokretanje sustava
 - koriste se posebni bootloaderi, npr. U-Boot
- *Kernel* – jezgra Linux-a
 - u produkcijskoj verziji tipično uključuje sve *device drivers*, dok se za razvoj često koriste *loadable kernel modules*
- *Root filesystem* – korijenski datotečni sustav
 - različite mogućnosti, ovisno o memorijskim značajkama URS-a
- *Application(s)* – programska potpora za specifičnu primjenu ciljnog sklopoljja

Bootloader

- učitava u radnu memoriju jezgru OS-a (sam bootloader nalazi se u FLASH-u)
- U-Boot – malen, dobro dokumentiran, bogat mogućnostima, podrška za različite platforme
 - PPC, ARM, MIPS, AVR32, x86, 68k, Nios, MicroBlaze
- interakcija putem komandne linije (putem serijskog porta), podrška za skripte
- podrška za udaljeno učitavanje jezgre (DHCP, TFTP, NFS – korisno prilikom razvoja)
- poseban format slike jezgre (kernel image)
 - podržava i kompresirane jezgre

Jezgra

- prilagodba jezgre cilnjom sklopoljju i primjeni
 - danas razvojni inženjer rijetko mora raditi *razvoj* unutar izvornog koda jezgre ili obavljati *prilagodbu cilnoj arhitekturi* (to tipično rade proizvođači sklopoških platformi), ali postoji velika potreba za **konfiguriranjem i izgradnjom prilagođene jezgre** za konkretan projekt (veličina koda, performanse, mogućnosti i sl.)
- konfiguriranje i izgradnja jezgre iz izvornog koda korištenjem pomoćnih alata (*make*)
- nomenklatura verzija jezgre, npr. `linux-2.4.18-rthal5`
 - prvi broj (2) – verzija jezgre (vrlo rijetko se mijenja)
 - drugi broj (4) – “*patch level*” (podverzija)
 - parni brojevi koriste za stabilne, a neparni za razvojne podverzije
 - treći broj (18) – “*sub-level*” (ispravke (*bug fixed*) koje ne utječu na kernel API)
 - sufiks – “*additional features*” (npr. “rthal5” - RTAI *kernel patch abstraction layer* 5)

Struktura izvornog koda jezgre

Arch	Contains a directory dedicated to each architecture that Linux supports. The Linux kernel has very little architecture-specific code; most of what's here focuses on boot-time, power management, and the like. Any assembler code for a given architecture is found in this directory. Under each architecture, the organization varies a little, but each contains folders for architecture subtypes and board-specific code.
Block	Contains core code for managing block devices. The driver code resides elsewhere; this is the API the kernel presents to block devices.
Crypto	Contains the cryptographic API for the kernel.
Documentation	Contains kernel documentation. This is a key part of the kernel: the documentation is well written and well-organized. Use the 00-INDEX file as the table of contents for the documentation.
Drivers	Contains directories for each major type of driver: USB, keyboard, display, and network drivers, to name a few.
Firmware	Provides the API For devices that have firmware loaded into memory as part of the device driver load.
Fs	Contains the general code for handling file systems in Linux and a directory for each supported file system.
Include	Contains header files used throughout the kernel project. The kernel sets the search path this directory when building the kernel.

Struktura izvornog koda jezgre

Init	Contains the code that's run as part of the kernel's initialization. The entry point for the kernel (after the assembler parts of the code are executed) is in this directory.
Ipc	Contains the API for interprocess communications. This directory contains the code for the semaphores and pipes.
Kernel	Contains infrastructure code for the kernel. The scheduler code resides in this directory, as does the timer and process-management code.
Lib	Contains helper code that's shared across all other parts of the kernel.
Mm	Contains the memory-management code for handling virtual memory and paging.
Net	Contains the TCP stack and networking infrastructure code. The kernel supports much more than TCP in terms of networking, and each networking technology has a directory with some of the shared code in the top level
Samples	Contains sample code for a few of the newer kernel constructs like kobjects and tracepoints.
Scripts	Contains the kernel configuration and build code. None of the code in this directory is included in the kernel when it's compiled—it's there to make the compilation process happen.

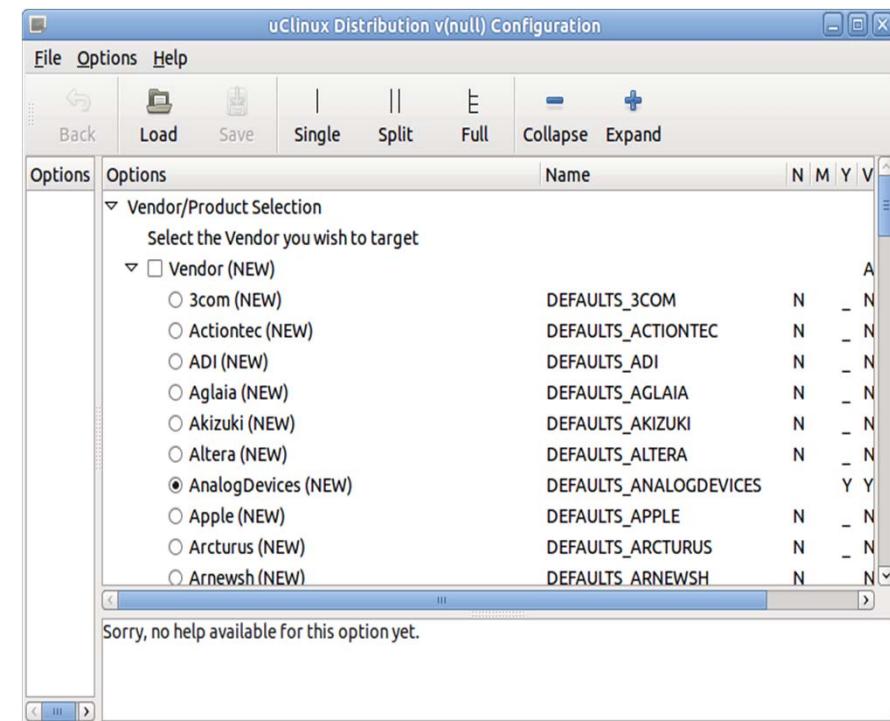
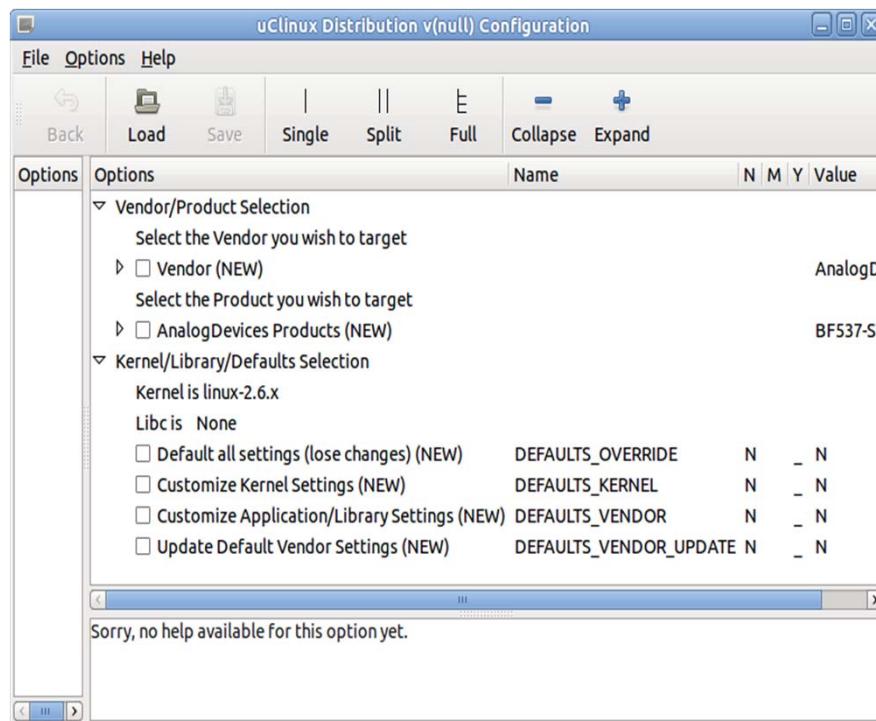
Struktura izvornog koda jezgre

Security	Contains Security Enhanced Linux (SELinux) and smack, an alternate access control system. The top-level directory contains some code shared by both of these modules.
Sound	Contains the drivers and codecs for audio.
Usr	Contains the stub initramfs file system when one isn't included in the kernel during the build process.
virt	Contains the kernel-level virtualization code for the kernel. This isn't code for QEMU virtualization; it's the drivers for the kernel-level x86 virtualization used in server devices.

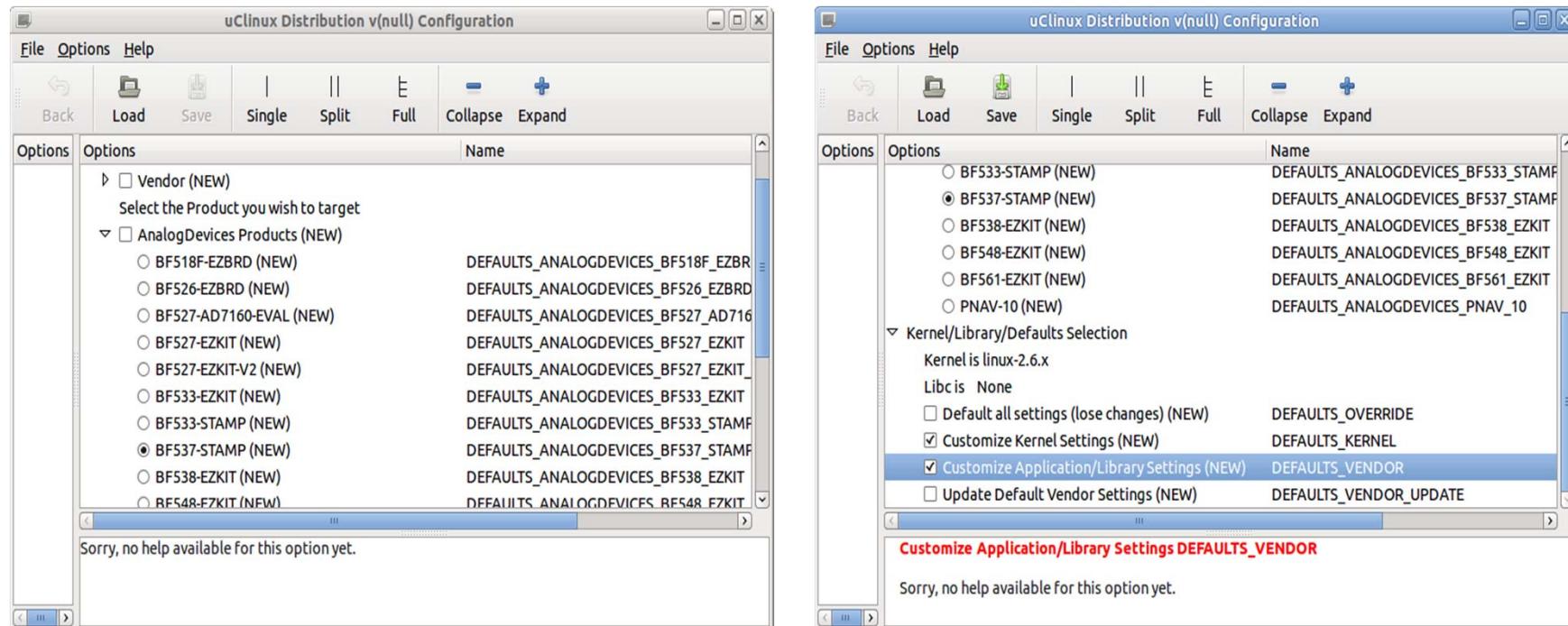
Postupak izgradnje jezgre

- koraci:
 - konfiguracija jezgre – zadavanje parametara i opcija tipično putem sustava izbornika
 - izgradnja jezgre iz izvornog koda
 - izgradnja dodatnih jezgrenih modula koje je moguće naknadno učitati u jezgru, nakon pokretanja OS-a (*loadable kernel modules*)
- sučelja za konfiguriranje parametara:
 - make menuconfig (tekstualno), make xconfig (Qt grafičko), make gconfig (GTK grafičko)

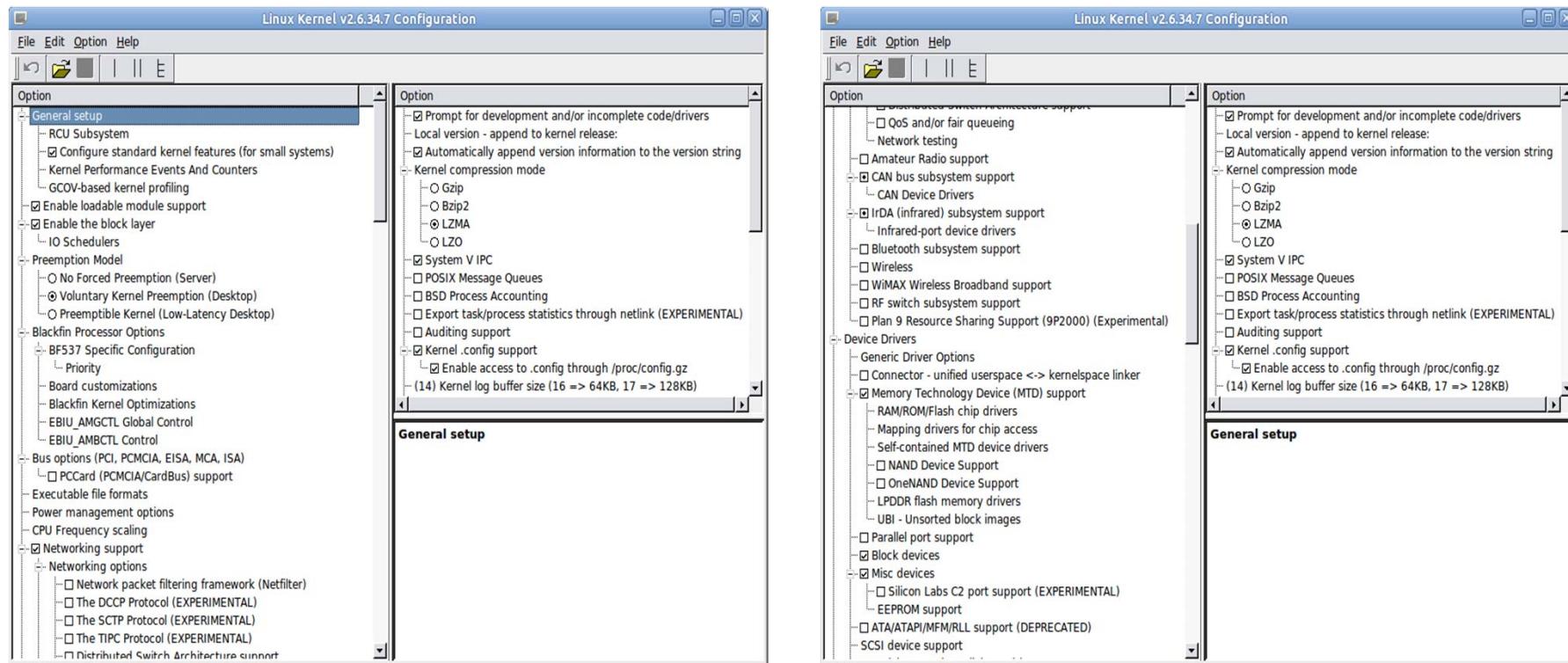
Primjer – konfiguriranje jezgre



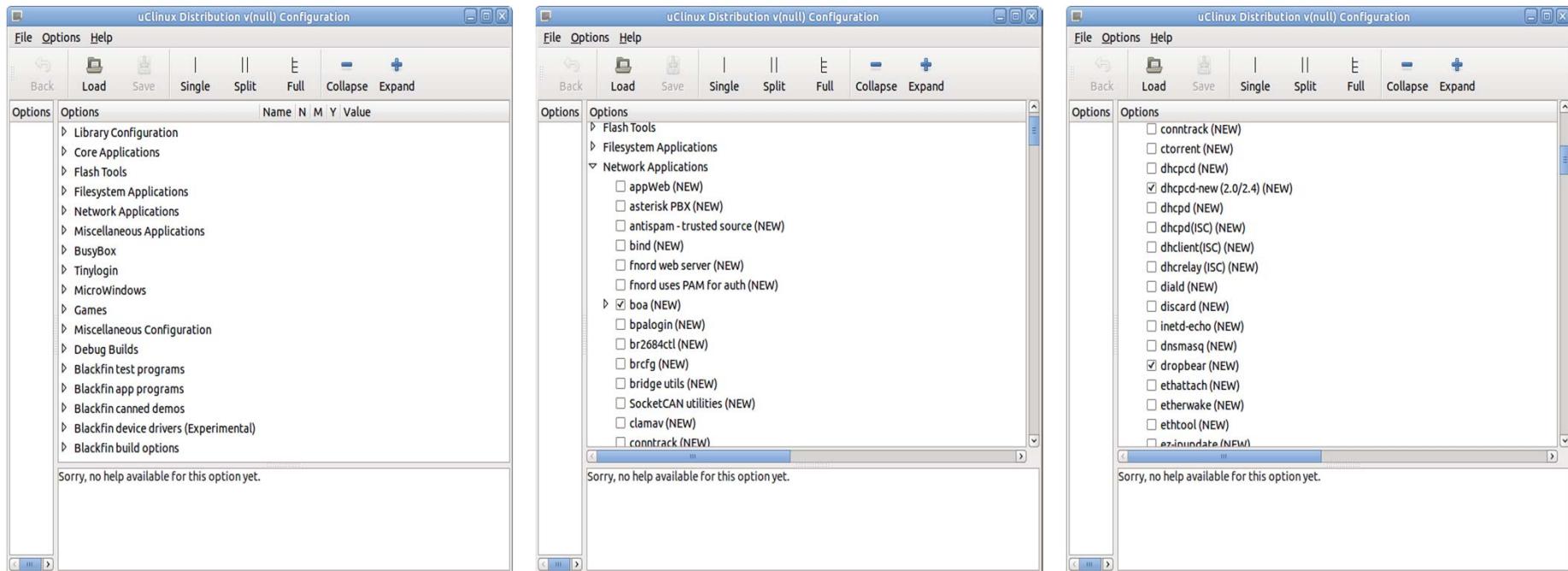
Primjer – konfiguriranje jezgre



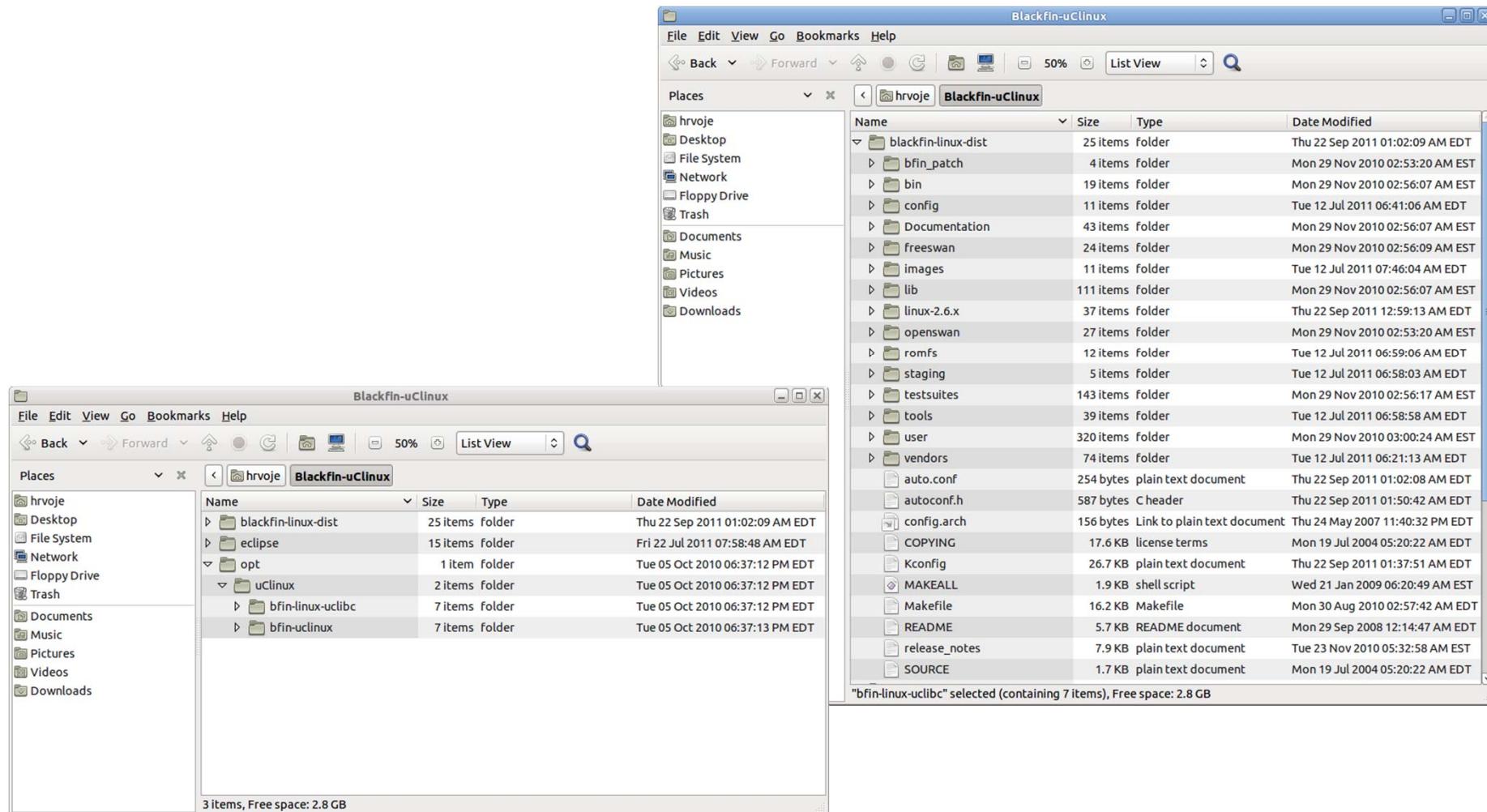
Primjer – konfiguriranje jezgre



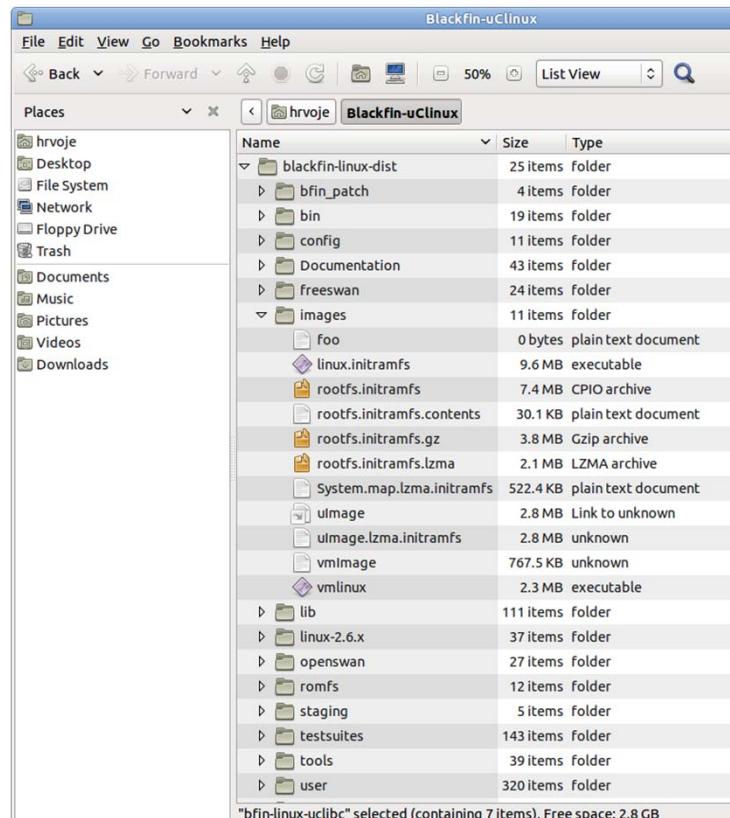
Primjer – konfiguriranje jezgre



Primjer – alati i distribucija

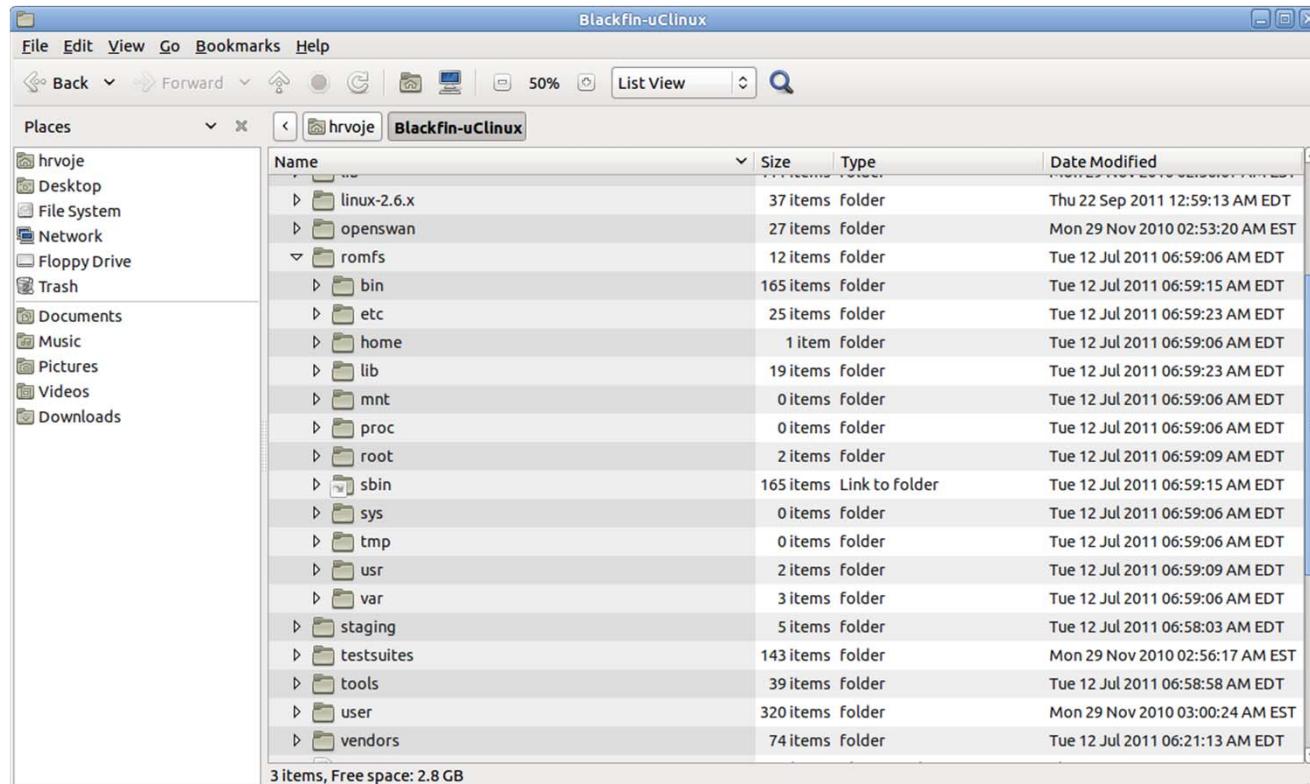


Primjer – izlazne datoteke (jezgra, FS)

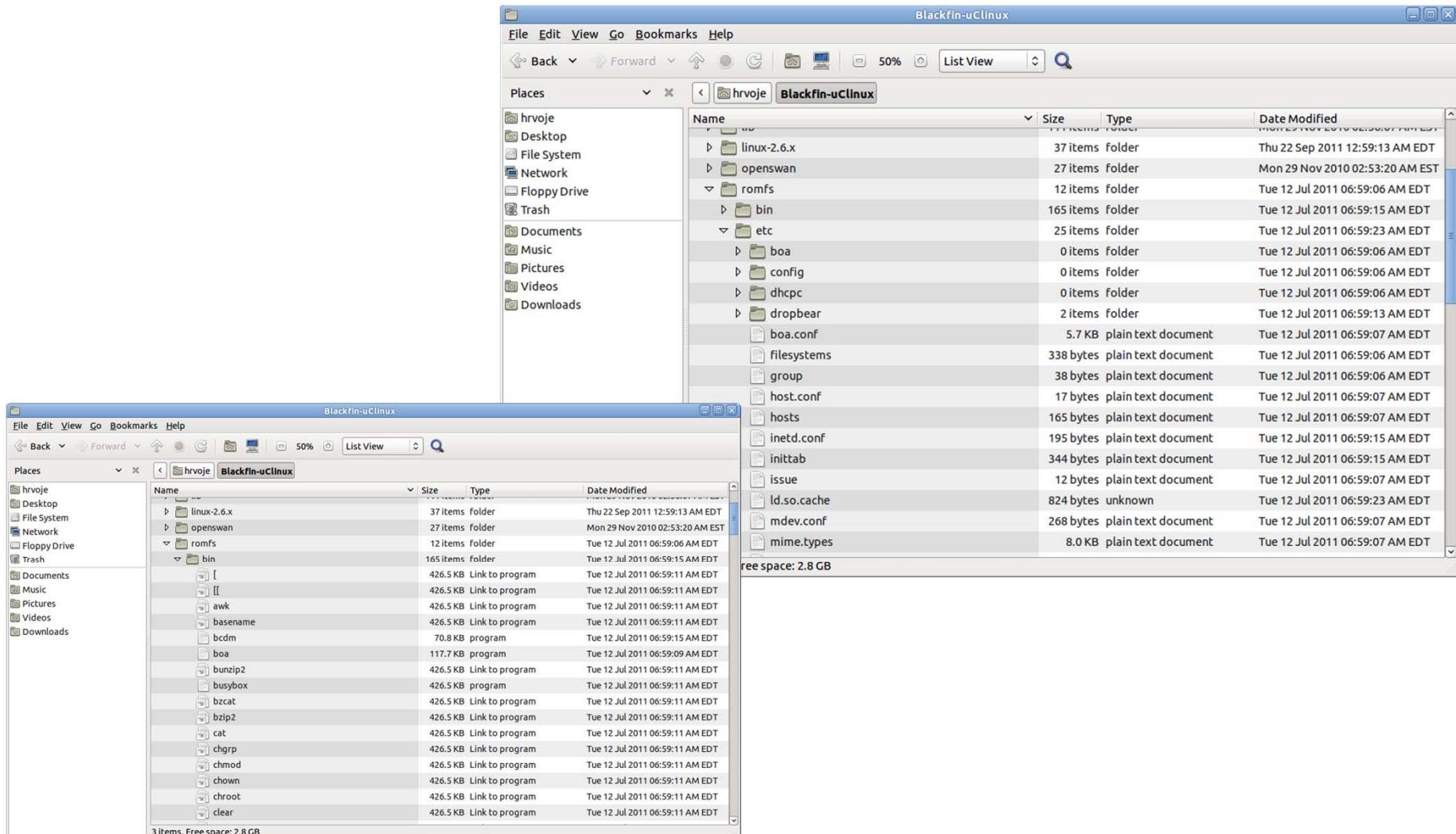


Datoteka	Značenje
linux.initramfs	Linux kernel, ELF format, with cpio archive filesystem attached
rootfs.initramfs	Root filesystem formatted in the cpio archive format
rootfs.initramfs.gz	Root filesystem formatted in the cpio archive format and compressed with gzip
rootfs.initramfs.lzma	Root filesystem formatted in the cpio archive format, compressed with lzma
uImage	Symlink to the “default” uImage.rootfs file
uImage.lzma.initramfs	Linux kernel in bootable U-Boot format, with initramfs filesystem attached, lzma compression
vmImage	Linux kernel in bootable U-Boot format, no root filesystem
vmlinux	Linux kernel, ELF format, no root filesystem

Primjer – root filesystem

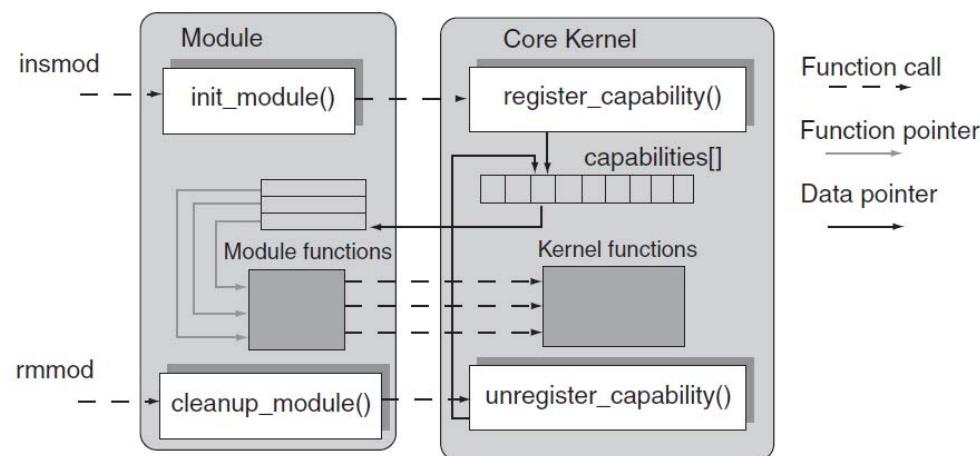


Primjer – root filesystem



Jezgrini moduli i upravljački programi

- *kernel module* – proširenje funkcionalnosti jezgre OS-a bez potrebe za rekomplajliranjem jezgre
 - implementacija servisa OS-a kojeg mogu pozivati korisničke (*user-space*) aplikacije
 - mogućnost dinamičkog učitavanja (*loadable kernel module*)
 - shell komande `insmod` / `rmmmod`
 - API funkcije – `init_module()`, `cleanup_module()`, `register_capability()`, `unregister_capability()`



Jezgrini moduli i upravljački programi

- *device driver* – upravljački program za kontrolu sklopovlja
- apstrakcija pristupa sklopovlju od strane aplikacijskog koda
- u sustavima bez OS-a ili s jednostavnijim RTOS-om, *device driver* može biti obična C biblioteka s funkcijama za pristup sklopovlju
- kod sustava opće namjene, *device driveri* su tipično posebni programi koji se izvršavaju u zaštićenom prostoru jezgre:
 - pristup I/O jedinicama mora biti zaštićen na razini jezgre,
 - kod sustava s podrškom za virtualnu memoriju, problem kod I/O operacija u korisničkom programu može nastati prilikom *swap* operacije (mijenjaju se fizičke lokacije stranica u memoriji)
- pod Linuxom upravljački programi (*device drivers*) realizirani su kao *kernel moduli*

Linux upravljački programi

- perifernim jedinicama pod Linuxom pristupa se putem datotečnog sustava
- definiraju se tri tipa perifernih jedinica (*device*) kojima se pristupa preko uniformnog API-ja:
 - *character* – pristup uređaju putem *sekvencijalnog* tijeka individualnih znakova:
 - najjednostavniji tip uređaja; primjeri: tipkovnica, miš, serijski i paralelni port, IrDA, konzole, zvuk i sl.
 - *block* – pristup uređaju putem blokova s mogućnošću *proizvoljnog* načina pristupa (*random access*)
 - hard disk, floppy, ram disk, loop device itd.
 - *network* – pristup putem tijeka paketa u skladu s odabranim komunikacijskim protokolom

Linux upravljački programi

- /dev direktorij – mjesto u datotečnom sustavu u kojem se nalaze svi dostupni *device driveri*
 - slično kao i /proc, radi se o “virtualnom” datotečnom sustavu
- primjer:

```
> ls /dev -l

crw-rw-rw- 1 root tty      5,   0 2011-12-21 18:43  tty
crw--w---- 1 root root    4,   0 2011-12-21 18:18  tty0
crw-rw---- 1 root dialout 4,  64 2011-12-21 18:18  ttyS0
crw-rw---- 1 root dialout 4,  65 2011-12-21 18:18  ttyS1
brw-rw---- 1 root disk    1,   0 2011-12-21 18:18  ram0
brw-rw---- 1 root disk    1,   1 2011-12-21 18:18  ram1
brw-rw---- 1 root disk    1,  10 2011-12-21 18:18  ram10
brw-rw---- 1 root disk    1,  11 2011-12-21 18:18  ram11
lrwxrwxrwx 1 root root    3 2011-12-21 18:18  cdrom -> sr0
crw----- 1 root root    5,   1 2011-12-21 18:18  console
drwxr-xr-x 2 root root   60 2011-12-21 18:18  cpu
lrwxrwxrwx 1 root root   13 2011-12-21 18:18  fd -> /proc/self/fd
brw-rw---- 1 root floppy  2,   0 2011-12-21 18:18  fd0
crw-rw-rw- 1 root root    1,   3 2011-12-21 18:18  null
crw----- 1 root root  254,   0 2011-12-21 18:18  rtc0
lrwxrwxrwx 1 root root   15 2011-12-21 18:18  stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root   15 2011-12-21 18:18  stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root   15 2011-12-21 18:18  stdout -> /proc/self/fd/1
```

Pristup device driveru

- user-space korisničke aplikacije pristupaju *device driveru* putem *low-level stream API*-ja:

```
int open(const char* path, int oflags, mode_t mode);  
size_t read(int fid, void* buf, size_t count);  
size_t write(int fid, void* buf, size_t count);  
close(int fid);  
int ioctl(int fid, int cmd, ...);
```

- uređajima se pristupa kao datotekama
- *ioctl()* funkcija omogućuje specifične operacije nad uređajima, npr. čitanje i pisanje u upravljačke sklopovske registre
- primjer kreiranja čvora uređaja u datotečnom sustavu:
`mknod /dev/ttyS1 c 4 65`
(kreira serijski port Com2 (standardna oznaka /dev/ttyS1), kao *character device* (c), koji koristi *major device number* 4 (klasa chrdev[]) i *minor device number* 65)

Root Filesystem

- prilikom izgradnje jezgre stvaraju se datoteke koje sadrže sliku (*image*) jezgre (*kernel*) i korijenskog datotečnog sustava (*root filesystem*)
 - ovisno o postavkama, slika jezgre može i ne mora uključivati *root filesystem*
- datotečni sustav slijedi FSH preporuke i obično implementira minimalni potrebni skup direktorija
- ovisno o memorijskim resursima URS-a, postoje tri kategorije datotečnih sustava (koje se mogu i kombinirati)
 - *Block-based file systems*
 - *MTD file systems (Memory Technology Device)*
(podrazumijevaju se *FLASH* memorije)
 - *RAM buffer-based file systems*

Block-based File Systems

- datotečni sustavi za jedinice za pohranu podataka koje upravljaju sekvencama blokova; tipični primjer je tvrdi disk, ali se ovi datotečni sustavi mogu koristiti i kod drugih uređaja
- moguće ih je koristiti i za MTD uređaje, ali samo u R/O modu
- datotečni sustavi:
 - **Ext2** – najstariji Linux datotečni sustav; potpuna implementacija dozvola, vlasništva, tipova datoteka i vremenskih oznaka; često se koristi i kod R/O FLASH uređaja zbog jednostavnosti i kompletnosti implementacije
 - **Ext3** – unaprijeđenje Ext2 datotečnog sustava; dodaje mogućnosti poput *journalinga* (sve promjene se najprije pišu u privremeno područje prije njihovog prihvaćanja) i podršku za velike kapacitete (8 TB); voditi računa od *overheadu journalinga* kod URS, a za R/O sustave nije potreban

Block-based File Systems

- datotečni sustavi:
 - **SquashFS** – *read-only* datotečni sustav s podrškom za kompresiju podataka (LZMA, LZO)
 - **CramFS** – *read-only* datotečni sustav s podrškom za kompresiju podataka, koji metapodatke o datotekama ima nekompresirane radi lakšeg pristupa; veličine datoteka ograničene su na 16 MB, a ukupni adresni prostor na 272 MB
 - **Romfs** – najmanji datotečni sustav (veličina koda svega 4 KB), ali i najmanjih mogućnosti; *read-only*, ne sadrži informacije o vlasništvu, atributima i vremenskim oznakama

MTD File Systems

- Flash uređaji za pohranu podataka ne mogu se koristiti na isti način kao blok-uređaji (npr. hard disk)
- problem ograničenog broja ciklusa brisanja Flash memorije (10^4 - 10^5)
- rješenje – *wear leveling* – raspoređivanje korištenja Flash memorije na način da se svi blokovi koriste ravnomjerno
 - npr. USB diskovi imaju ugrađenu FTL (*Flash Translation Layer*) podršku
- kada bi se koristio ext2 datotečni sustav koji ne vodi računa o *wear-levelingu*, moglo bi puno brže doći do uništenja blokova u memoriji
 - naravno, to nije problem ako se Flash koristi kao R/O uređaj

MTD File Systems

- datotečni sustavi:
 - **JFFS2** – od verzije jezgre Linuxa 2.4; kompresija podataka i metapodataka o datotekama, što ima pozitivni efekt na zauzeće prostora, ali pristup podacima čini sporijim
 - **YAFFS2** – MTD datotečni sustav koji ne komprimira podatke, što rezultira većim zauzećem Flash memorije, ali bržim pristupom

RAM Buffer-Based File Systems

- datotečni sustavi optimirani za RAM; sadržaj se gubi nestankom napajanja
- datotečni sustavi:
 - **Ramfs** – vrlo jednostavan i brz datotečni sustav, s malim memorijskim zauzećem; problem može nastati kod pisanja jer nema ograničenja sve dok se ne zauzme cijela radna memorija; zato je prvenstveno pogodan za R/O sustave
 - **Tmpfs** – sličan Ramfs-u, s tom razlikom da omogućava limitiranje zauzeća radne memorije (npr. kod korištenja za logove, `/var/log/messages`); za razliku od Ramfs-a, podržava i swap memoriju, što nije najčešće bitno kod URS
 - **Initramfs** – poseban slučaj Ramfs-a, koji je optimiran za postupak učitavanja jezgre (služi kao privremeni korijenski datotečni sustav prije montiranja konačnog)
- često se koriste u kombinaciji s drugim sustavima (npr. kod MTD sustava korisno je koristiti za `/tmp Tmpfs i sl.`)

Korisničke aplikacije

- *cross-compiler* – izrada aplikacijskog binarnog koda
- prebacivanje aplikacije *host* → *target* – putem datotečnog sustava
 - više mogućnosti: NFS, tftp, ftp, rcp, ssh (sftp, scp) itd.
- odabir strategije *debuggiranja*
- prednosti pisanja korisničkih aplikacija pod Embedded Linuxom:
 - zaštita memorijskog prostora i jezgre,
 - *device-independant* pristup sklopolju, korištenje servisa OS-a
 - apstrakcija sklopolja – paradigma bliža “non-embedded” pristupu
 - korištenje standardnih biblioteka i paketa
 - korištenje gotove infrastrukture OS-a (datotečni sustav, umrežavanje, sigurnost, komunikacija među procesima itd).
 - niz gotovih rješenja koja se na fleksibilan i modularan način mogu uključiti itd.

Pristupi debuggiranju

- interaktivno – kontrola izvođenja programa korištenjem prekidnih točaka
 - kod debugiranja aplikacija na Linuxu ne koriste se standardna JTAG sučelja kao kod mikrokontrolera, jer se debugira aplikacija koja se izvodi u *user spaceu* (uobičajena JTAG sučelja pogodna npr. verifikaciju bootloadera, punjenje sadržaja FLASH memorije i sl.)
 - kod Embedded Linux-a potreban poseban *remote debugging* sustav, jer se ciljna Linux aplikacija izvodi na *targetu*, a debugger na *hostu*
 - problem – teže određivanje pogrešaka uslijed *timinga*
- “*post mortem*” – ispitivanje log-a stanja procesa u slučaju neočekivanog prekida (“*crash*”)
- “*instrumentation*” – ispis poruka za vrijeme izvođenja aplikacije (npr. na stdout konzolu korištenjem *printf()* funkcije)
 - usporavanje aplikacije

Debuggiranje (embedded) Linux aplikacija

- GDB – GNU Debugger
 - omogućuje *remote debugging* – aplikacija se izvodi se na *target* računalu, a debugira na *hostu*
 - GDB server mora biti instaliran na *target* računalu (~100 kB)
 - tri elementa: program kojeg se debugira (*target*), debug monitor (*gdbserver* – *target*), debugger (*host*)
 - mrežno povezivanje GDB klijenta (*host*) i servera (*target*) (TCP/IP protokol, serijska komunikacija)
 - aplikacija mora biti kompajlirana s debugging informacijama (npr. u DWARF formatu – *Debugging With Attributed Records Format*)
- sučelje za debugiranje:
 - komandna linija
 - grafički *front-end* (*Data Display Debugger*, DDD)
 - IDE s ugrađenim debuggerom (npr. Eclipse)

Linux i rad u stvarnom vremenu

- višekorisnički operacijski sustav opće namjene – sustav nije pogodan za rad u stvarnom vremenu
- razlozi zašto Linux nije pogodan za rad u RT:
 - *non-preemptible kernel* – iako je istiskivanje procesa podržano, sistemske pozive nije moguće istiskivati (bez posebnih modifikacija jezgre OS-a!)
 - *paging* – vrijeme zamjene stranica između radne memorije i diska nije ograničeno – nedeterminizam!
 - *completely fair scheduler* (CFS) – “poštena” raspodjela procesorskog vremena između procesa; algoritam nastoji dodijeliti prosječno jednak vijeme svakom procesu; to znači da će proces niže razine dobiti procesor na raspolaganje ako je dugo čekao, čak i u slučaju da je proces više razine prioriteta u stanju spremnosti za izvođenje

Linux i rad u stvarnom vremenu

- razlozi zašto Linux nije pogodan za rad u RT:
 - *request reordering* – radi učinkovitijeg iskorištenja sklopolja, OS može prerasporediti zahtjeve U/I jedinica; npr. prednost mogu dobiti zahtjevi za višestrukim čitanjem podataka s tvrdog diska kako bi se izbjegla latencija kod pozicioniranja glave diska, iako na obradu čekaju prioritetniji zahtjevi s drugih U/I jedinica
 - *batching* – uzastopno obavljanje niza istovrsnih operacija radi povećanja srednje brzine odziva sustava (npr. zamjena niza stranica između memorije i diska efikasnija je ako se ne radi pojedinačno); problem determinizma odziva sustava
- zbog navedenih razloga Linux nije pogodan za primjenu u *hard real-time* sustavima!

Prilagodba Linux-a zahtjevima za rad u stvarnom vremenu

- standardna Linux jezgra podržava sljedeće algoritme raspoređivanja zadataka na razini procesa (*scheduling policy*):
 - SCHED_OTHER (*default*) (za “normalne” procese)
 - SCHED_FIFO, SCHED_RR (za “real-time” procese)
- svakom procesu dodjeljuje se prioritet u rasponu od 0 do 99
 - 0 predstavlja najniži prioritet i imaju ga procesi koji se raspoređuju prema SCHED_OTHER kriteriju
 - SCHED_FIFO i SCHED_RR se odnose na procese 1-99
- procesi se mogu međusobno istiskivati na način da proces više razine prioriteta istisne proces niže razine
 - *scheduling policy* određuje samo ponašanje između procesa iste razine prioriteta!
 - pažnja – istiskivanje zadataka nije moguće za vrijeme dok se izvodi sistemski poziv unutar jezgre! – nije pravi RTOS!

Prilagodba Linux-a zahtjevima za rad u stvarnom vremenu

- SCHED_OTHER:
 - standardni Linux raspoređivač za procese s prioritetom 0
 - implementira *fair scheduling* pristup raspoređivanju procesa
- SCHED_FIFO:
 - samo za procese prioriteta ≥ 1
 - može istisnuti proces prioriteta 0 (SCHED_OTHER) ili biti istisnut procesom višeg prioriteta
 - nakon prepuštanja procesora (*sched_yield*) postavlja se na kraj FIFO strukture – izvodi se sve dok dobrovoljno ne prepusti procesor
- SCHED_RR (*Round-Robin*):
 - isto kao i SCHED_FIFO, s tom razlikom da se proces može prekinuti i postaviti na kraj FIFO strukture i prije nego što eksplicitno prepusti kontrolu, a nakon što istekne definirani vremenski interval izvođenja procesa (*time slice*)

Prilagodba Linux-a zahtjevima za rad u stvarnom vremenu

- iako SCHED_FIFO i SCHED_RR algoritmi poboljšavaju brzinu odziva sustava za “*real-time*” zadatake, i dalje ne predstavljaju dobro rješenje za *hard real-time*, jer se ne mogu istisnuti sistemski pozivi unutar jezgre
- dva su osnovna pristupa modifikaciji Linux-a u cilju postizanja determinističkog odziva definiranog zahtjevima za rad u stvarnom vremenu:
 - *preemption improvement* – poboljšanje brzine odziva sustava modifikacijom jezgre tako da i sama jezgra podržava mogućnost istiskivanja
 - *interrupt abstraction* – korištenje RTOS-a u kojem se Linux izvodi kao jedan proces niske razine prioriteta, dok se zadaci s ograničenjima za rad u stvarnom vremenu izvode pod kontrolom jezgre RTOS-a

Prilagodba Linux-a zahtjevima za rad u stvarnom vremenu

- *preemption improvement* pristup
 - poboljšanje latencije odziva sustava uvođenjem mogućnosti prekida sistemskih poziva
 - korištenje Linux *spinlock* objekata koji već postoje na razini OS-a i omogućuju realizaciju kritičnih sekcija kod SMP-a (*symmetric multiprocessing*)
 - *spinlock* – čekanje na lock za vrijeme kojeg se proces izvodi u *idle* petlji
 - standardna Linux jezgra osigurava prosječnu latenciju od više desetaka ms, dok jezgra s *preemption improvement* mogućnostima omogućuje latencije reda veličine par ms

Prilagodba Linux-a zahtjevima za rad u stvarnom vremenu

- *preemption improvement* pristup
 - prednosti:
 - aplikacije za rad u stvarnom vremenu su obični Linux *user-space* procesi, s potpunom zaštitom memorije koju nudi jezgra OS-a (sigurnost, stabilnost rada)
 - nedostaci:
 - modifikacija jezgre Linux-a je složen i zahtjevan postupak
 - spomenute ekstenzije uvode se u glavnu distribuciju od verzije 2.5.4-pre6
 - poboljšanje latencija na ovaj način i dalje ne čini Linux pravim operacijskim sustavom za rad u stvarnom vremenu!
 - zbog unutarnje složenosti jezgre i dalje je teško osigurati determinizam i pouzdanost kakve zahtijevaju *hard real-time* kritični sustavi!
- Embedded Linux distribucije temeljene na *preemption improvement* pristupu:
 - Mona Vista, TimeSys

Prilagodba Linux-a zahtjevima za rad u stvarnom vremenu

- *interrupt abstraction* pristup
 - u većini praktičnih slučajeva, samo mali dio sustava mora zadovoljavati *hard real-time* ograničenja
 - Linux već pruža infrastrukturu za mnoge složene programske operacije koje se ne trebaju izvoditi u RT
 - ideja: mala jezgra RTOS-a upravlja zadacima koji se moraju izvoditi u stvarnom vremenu, dok se Linux izvodi kao zadatak niske razine prioriteta
 - *interrupt abstraction* – prekidima upravlja jezgra RTOS-a, što je za Linux transparentno

Prilagodba Linux-a zahtjevima za rad u stvarnom vremenu

- *interrupt abstraction* pristup
 - prednosti:
 - potrebne puno manje modifikacije same jezgre Linuxa
 - vrlo niska latencija prekida i zamjene konteksta – reda veličine nekoliko desetaka mikrosekundi!
 - nedostaci:
 - *real-time* zadaci izvode se u *kernel spaceu*
 - nema zaštite memorije i *real-time* zadatak može zablokirati cijeli sustav!
 - uvođenje dodatnog API-ja
- Embedded Linux distribucije temeljene na *interrupt abstraction* pristupu:
 - RTLinux, RTAI

Literatura

- D. Abbott: Linux for Embedded and Real-time Applications, Newnes, 2006, 978-0750679329
- G. Sally: Pro Linux Embedded Systems, APRESS ACADEMIC, 2010, ISBN 978-1430272274
- Embedded Linux Training materials, <http://free-electrons.com/>, 2011.