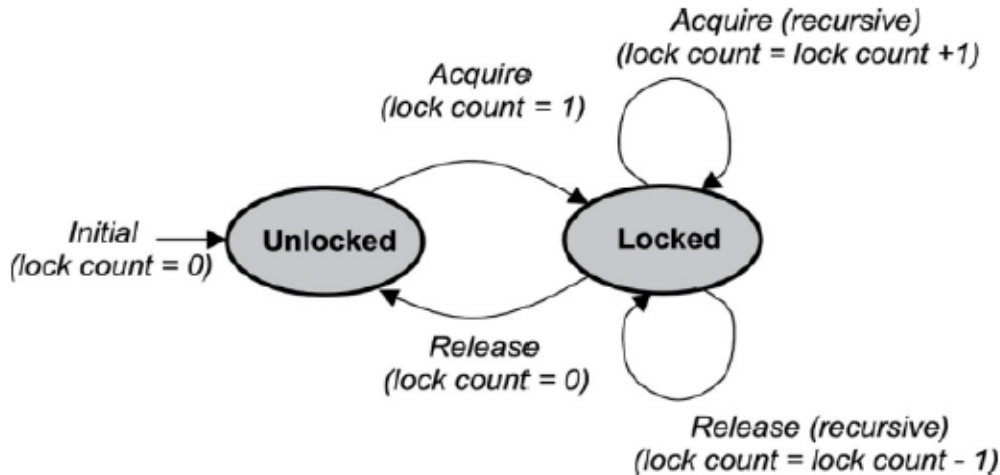


Jesenski rok:

PISMENI:

1. Nacrtati model mutexa, tj. oba stanja i sve prijelaze označiti



Skraćeno od **mutual exclusion object**. Mutex je programski objekt koji dozvoljava da više programskih threadova koristi isti resurs, kao što je pristup datoteci, ali ne istovremeno.

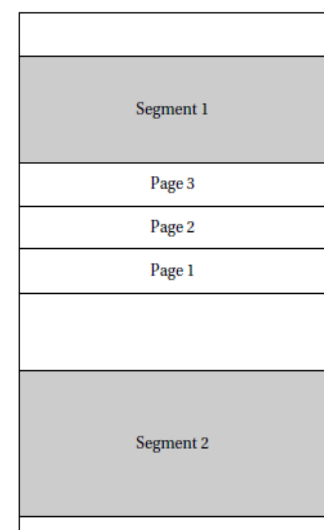
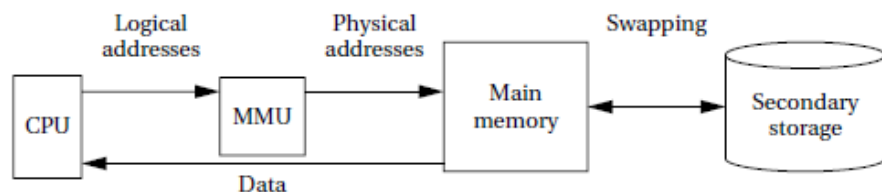
Kada se program pokrene, mutex se stvori sa jedinstvenim imenom. Svaki thread koji treba resurs mora zaključati mutex od drugih threadova dok koristi resurs. Mutex se otključa kada je rutina završena ili kada podaci nisu više potrebni.

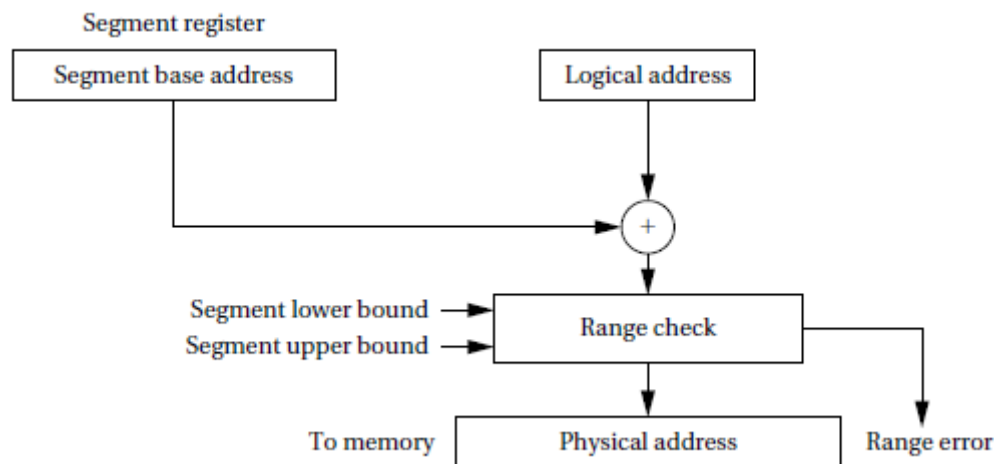
Ponovnim pristupanjem u **Locked** neće sam sebe zaključati već će samo povećati *lock count*.

2. Nacrtati sliku sheme straničenja memorije (memory translation, memory paging)

Paging se koristi kao tehnika za realizaciju virtualne memorije. Nemamo dovoljno interne memorije da bi učitali

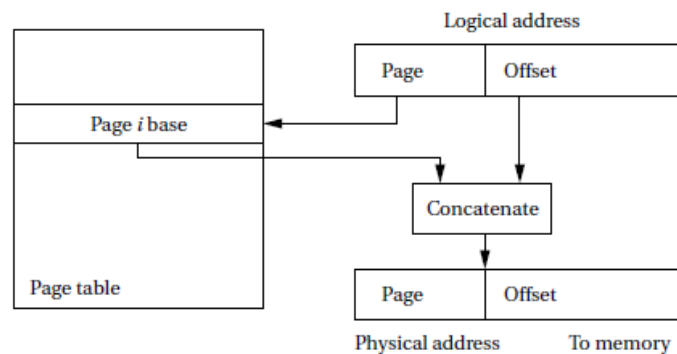
cijeli proces pa koristimo virtualnu memoriju (ona je veća od fizičke memorije). Proces misli da se i dalje izvršava, pristupa memorijskim lokacijama koje fizički i ne postoje, tj. pristupa virtualnim adresama generiranim iz virtualnog adresnog prostora. Gore je opisan tijek zauzimanja memorije: logička adresa ide u MMU koji virtualnu adresu pretvara u fizičku. Dodavanjem vanjske pohrane, možemo osloboditi djelove programa iz glavne memorije. MMU prati sve adrese u glavnoj memoriji te ako CPU zatraži nevaljanu adresu, izbacuje se **page fault**. Virtualni adresni prostor je podjeljen na *Segment* većih veličina memorije te *Pages* manjih, jednako podjeljenih regija, uniformnih veličina. Straničenje omogućava fragmetnaciju jer Segmenti su razbacani po memoriji.





Iznad vidimo primjer Segmentacije, MMU zadrži registar sa informacijom gdje je baza aktivnog segmenta. Fizička adresa se radi da se baznoj adresi doda offset te provjeri se veličina dobivene adrese.

Kod *paging* logička adresa se dijeli na dva segmenta: broj page-a i offset. Page number se koristi kao index u Page table (sadrži fizičku adresu početka svakog Pagea). S obzirom da su svi Pageovi istih veličina, MMU mora "ulančati" gornje bitove početne adrese Pagea sa donjim bitovima Page offseta da bi se dobila fizička adresa.



3. Navedi 5 tvrdnji vezanih uz RMS i EDF (ako krivu zaokružiš, poništava ti točnu, ali nema negativnih; mislim da su točne bile dvije; jedna koje se sjećam jest da je EDF optimalan i uz $U=100\%$, jer u uvjetu piše da U mora biti manje ili jednako 1, odnosno 100%)

RMS

- svi procesi se periodički izvršavaju na istom CPU
- zanemareno contex switching
- nema data dependencies između procesa
- konstantno vrijeme izvođenja procesa
- svi deadline su na kraju perioda
- porces sa najvišim prioritetom se uvijek izvršavaju (a priori se znaju!)- jedini takav algoritam

Prioriteti se dodjeljuju na principu da se procesu sa najmanjim periodom da najviši prioritet te oni se unaprijed znaju i statički su.

Algoritam:

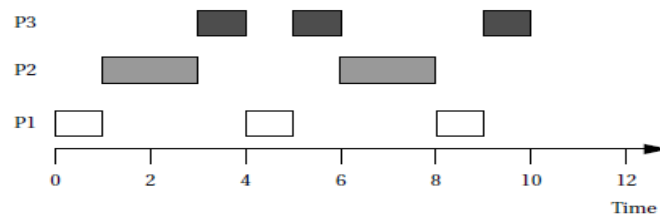
Za svaki zadatak J_i zna se period ponavljanja T_i .

Svakom J_i unaprijed se dodjeljuje prioritet (najviši – najkraći period)

Zadaci višeg prioriteta mogu prekidati zadatke nižeg prioriteta.

Primjer:

Process	Execution time	Period
P1	1	4
P2	2	6
P3	3	12



Prvo provjerimo je li raspored moguće ostvariti: ukupno vrijeme trajanja je 12 perioda (P3 najduži period). U tih 12 perioda P1 će se izvršiti 3 puta (3*1 trajanja), P2 2 puta (2*2 trajanja) te P3 1 puta (1*3 trajanja). Sve skupa 3+4+3=10 < 12 što je odlično, dakle rasporedivo je.

P1 najkraće traje pa ima najviši prioritet, potom P2 pa P3. Crtamo prvo P1 koji traje od 0-1, nakon toga dolazi P2 koji traje od 1-3, te počinje nakon njega P3 koji bi trebao trajati od 3-6 međutim, sljedeći period P1 koji je višeg prioriteta je u 4 te tada on prekida P3. Zbog toga P3 se izvršava 3-4 pa onda P1 4-5, pa ponovno P3 5-6. Zašto? Zato što P2 tek u 6 ponovno počinje. Nakon toga P2 6-8, pa opet nam P1 ima period, P1: 8-9 te preostao je P3:9-10.

Dovoljan uvjet rasporedivosti:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n \left(2^{\frac{1}{n}} - 1 \right)$$

Dovoljan, ali ne i nužan, ako je zadovoljena: rasporediv, nije zadovoljena: može biti rasporedivo u određenim slučajevima!
 U – faktor opterećenja procesora
 n-broj zadataka

Rasporedivost je moguća ako je opterećenje ispod donje granice $U < 0.69$

Kritičan trenutak: najgori mogući trenutak pojavljivanja zadatka koji će prouzročiti najdulje vrijeme odgovora. (tipa kad nam se pojavi zadatak višeg prioriteta pa se vrijeme izvršavanja početnog zadatka mora pomaknuti za to vrijeme izvršavanja višeg prioriteta)

EDF

- prioriteti se dodjeljuju dinamički (za vrijeme izvođenja)
- $D_i \leq T_i$
- preemptive algoritam (zadatak može biti prekinut za vrijeme izvođenja, zadatakom višeg prioriteta)
- znamo vrijeme izvođenja i preostalo vrijeme do deadlinea

Algoritam

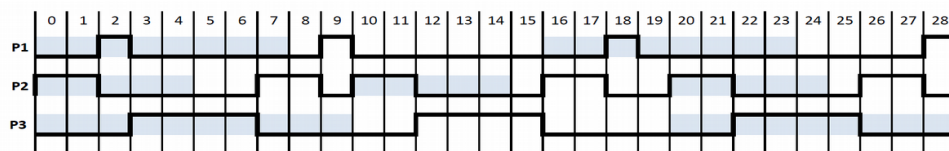
Trenutni zadatak J_i se prekida čim se pojavi zadatak s krajnjim rokom završetka bližim od trenutno aktivnog zadatka. Najviši prioritet u svakom trenutku ima onaj zadatak čiji je deadline najbliži.

Je li rasporediv?

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

Primjer:

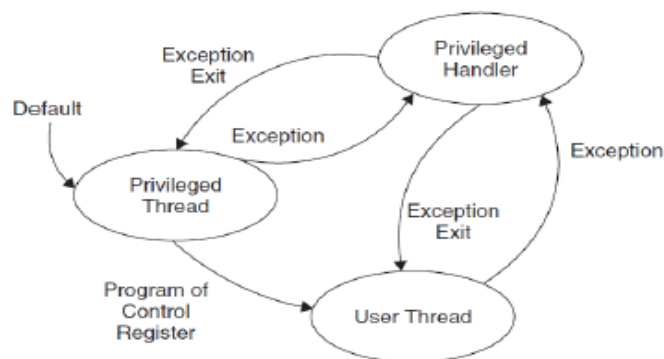
Process	Execution Time	Period
P1	1	8
P2	2	5
P3	4	10



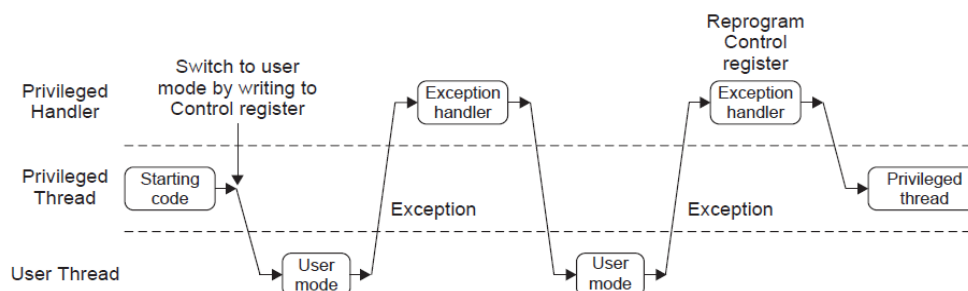
Prvo označimo periode ponavljanja svakog procesa, onda krećemo od trenutka 0. Prvo se izvodi P2 jer zbog najkraćeg perioda prvi će završiti. Onda slijedi P1 jer najprije će on završiti, pa preostaje još P3. U trenutku 5, to je već dio novog perioda P2 pa prvi će on završiti, nakon toga od slobodnih perioda jedino ostaje P1. U trenutku 10 ide P2 jer u novom periodu on prvi završava te još ostaje P3. U 16 vidimo da po periodima, P2 će prvi završiti pa on ide, nakon njega P1 i onda u 19 **idle**. Zašto? Zato što P1 je obavio svoj proces u periodu, P2 je isto kao i P3. U trenutku 20 P2 je najkraći završetku, pa P3 i onda logički P2 pa P1 završavaju.

4. načini rada i privilegirana stanja u Cortex-M-u

	Privileged	User
When running an exception	Handle Mode	
When running main program	Thread Mode	Thread Mode



Nakon reseta, procesor je u Thread modu sa Privileged rights. Sa Control registrom prelazi se u User Thread. Nakon iznimke, program uvijek prelazi u Privileged mode i vratiti se nazad kada izlazi iz exception handlera. Opet se vraća u Privileged handler te nakon što je reprogramiran Control registar, ulazi preko Exception handlera u Privileged thread.



5. prioriteti iznimki

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented			Not implemented, read as zero				

Iznimke višeg prioriteta mogu istisnuti one sa nižim prioritetom. Može biti do 256 *priority levela* i do 128 *preemption levela*.

Sa slike vidimo implementaciju sa 3 prioriteta bita. Bitovi 4-0 se očitavaju kao 0 jer nisu implementirani. I onda pisanjem vrijednosti kao 0x20, 0x40 do 0xE0 samo upisujemo u gornja tri bita.

Sa slike desno opet vidimo kako je to realizirano za primjer od 3 bita.

Taj 8 bitni registar podjeljen je na dva dijela: *preempt priority level* (gornji dio) te *subpriority level* (donji dio) implementiranog!

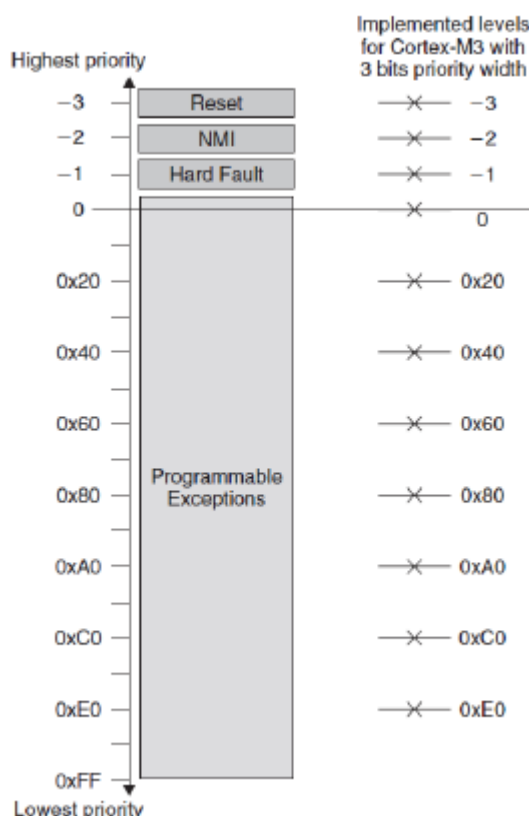
Preempt govori može li prekinuti trenutno izvođena iznimka (veća prekida manju po prioritetu)

Subpriority se koristi ako se pojave dvije preempt istog prioriteta, ona koja ima veći prioritet (nižu vrijednost u ovom polju), prva će se obraditi.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preempt priority		Sub priority					

Figure 7.4 Definition of Priority Fields in a 3-bit Priority Level Register with Priority Group Set to 5

Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

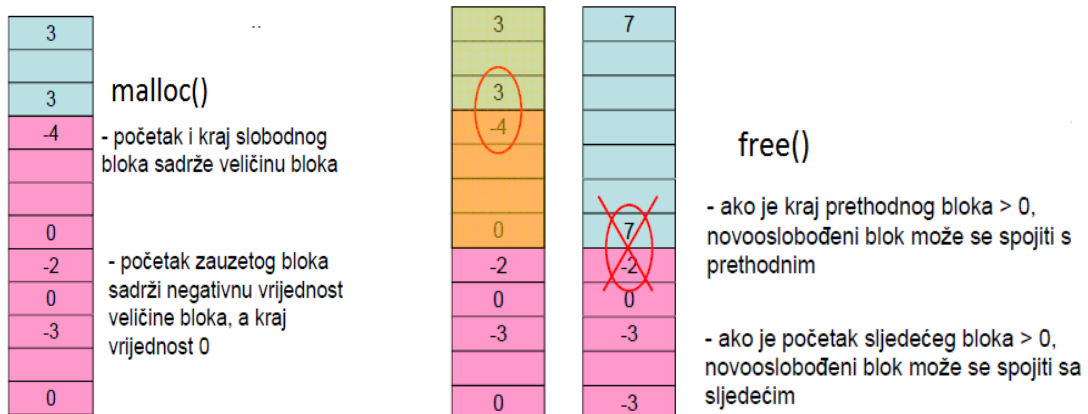
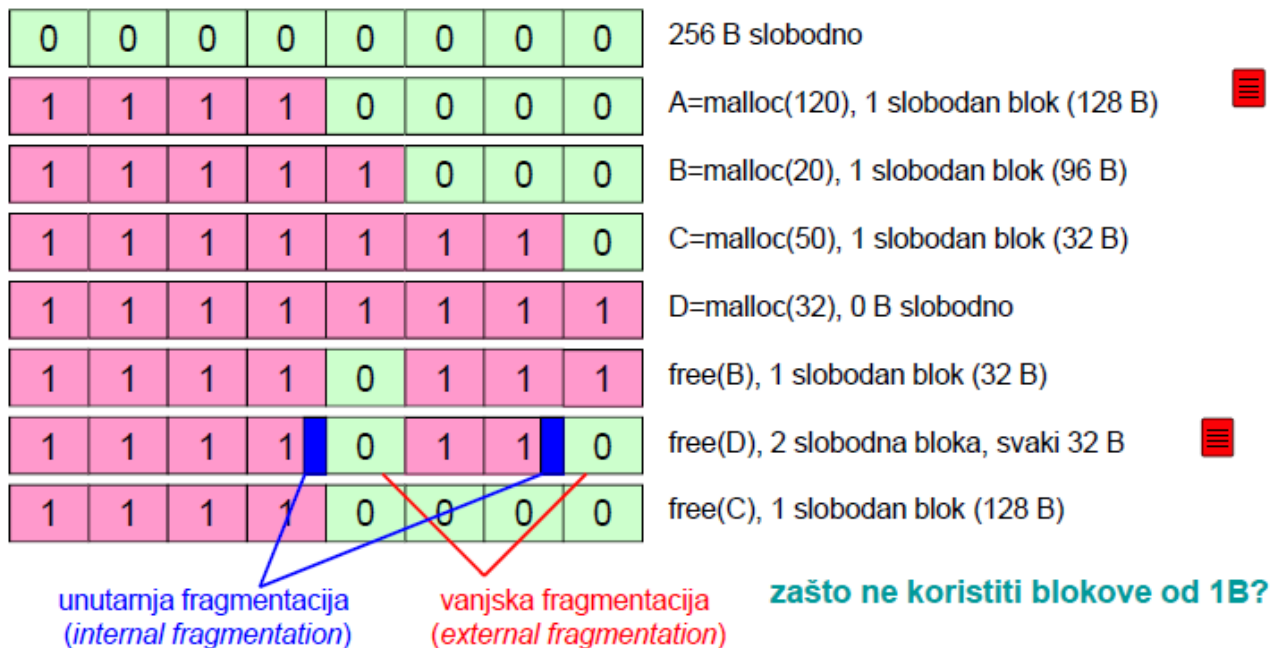


6. zadatak s malloc funkcijom, a) koliko je ostalo slobodnih blokova što se tiče unutarnje fragmentacije i pod b) što se tiče vanjske fragmentacije.

Blokovi su fiksne veličine, npr $B=32$. U početku je dostupno 256B to je 8 blokova te zbog efikasnosti pišemo u blokove umjesto opisivanja svakog bajta. Malloc će uvijek tražiti najveći kontinuirani blok za alokaciju memorije. Pod free(D) linijom, ne možemo alocirati više od 32B jer najveći kontinuirani blok je 32B.

Vanjska fragmentacija: oslobodimo jedan cijeli blok od 32B i to je slobodno

Unutarnja fragmentacija: kada smo sa A zauzeli 4 bloka, uzeli smo 128, a treba nam 120. Tih 8 je zapravo višak te predstavlja vanjsku fragmentaciju, isto je sa C kad smo uzeli 64, a treba nam 50.



7. zadatak s protokolom nasljeđivanja prioriteta kao na 40. slajdu 4. predavanja - dobiješ tri zadatka i vremena pojave događaja i vremena u kojem se semafori (mutex) zauzimaju i onda moraš napraviti ovaj graf, tj. označiti od kojeg do kojeg trenutka je pojedini zadatak aktivan. s obzirom da se radi o mutex-u malo je ipak kompliciranije radi rekurzivnog zaključavanja.

7a. protokol nasljeđivanja prioriteta - također sve što znaš; treba opisati i razliku između bounded i unbounded slučaja inverzije prioriteta i također je pitanje što je deadlock.

Većina RTOS sustava radi na multitaskingu te kod preemptive multitasking okruženja, dijeljenje resursa je stvar prioriteta Taska. Kako viši prioritetni taskovi imaju prednost u odnosu na niže, tako mogu uvijek uzimati resurse te tako izazvati *starvation* ili da niži taskovi se ne završavaju. Dva su najučestalija problema **deadlock** i **priority inversion problem**.

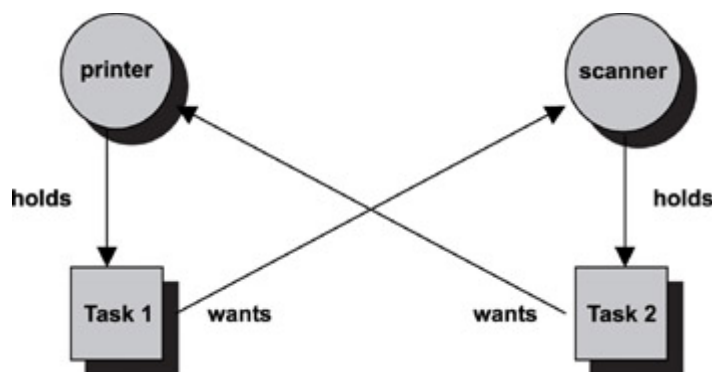
Resursi mogu biti preemptible (resurs može biti privremeno slučajno maknut bez narušavanja Taskovog stanja izvršavanja i rezultata) i nonpreemptible (dobrovoljno maknut od strane aktivnog taska ili nepredvidivi error)

Deadlock:

Situacija kad više istovremenih threadova koji se izvršavaju su blokirani za stalno zbog toga što se zahtjevi resursa ne mogu zadovoljiti)

Deadlock nastaje kad:

- resursu može pristupiti samo jedan task odjednom
- nema preemptiona (ne može se resurs na silu odvojiti od taska)
- drži i čekaj-task drži resurs dok čeka drugi da postane slobodan
- cirkularno čekanje-kao kružno čekanje, dva ili više taska izlazi, u kojem svaki task traži jedan ili više resursa koji isto to radi sljedeći u nizu.



Kada sustav uđe u stabilno stanje deadlocka, jedino pomoću vanskog utjecaja može biti riješen. Jedno od načina izbjegavanja deadlocka je detekcija, koja je implementirana na razini algoritma RTOS-a. Oporavak od deadlocka nema pravila, jedno od metoda je preemption resursa gdje se resurs oduzima od taska, dok kod nonpreemption resursa može doći da jedan task se izvršava, drugi želi pristupiti tom resursu. Prvi task će morati sačekati dok se drugi ne obavi, vrati se nazad na obavljanje i završi task. Tu zna doći do situacije da je onda i jedan i drugi task uzaludno napravljen posao. (Task piše u datoteku, drugi task hoće pročitati, ovaj mora stati sa pisanjem, dati da ovaj pročita i onda kasnije vratiti se i završiti napisano. Niti jedno nije napravilo posao što je trebalo). Nonpreempt se može oporaviti od deadlocka i pomoću checkpointa da se jednostavno vrati na zadnji poznati checkpoint.

Prevenција:

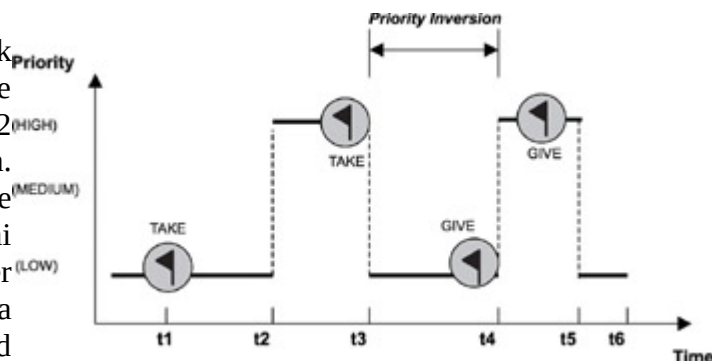
- eliminirati hold-wait stanje (task odjednom kaže koje će resurse koristiti)
- eliminirati no-preemption (task mora otpustiti postojeće resurse ako je novi zahtjev odbijen)
- eliminirati cirkularno čekanje (zahtjev za resursom treba biti broj više)

Priority inversion

Situacija kada task nižeg prioriteta se izvršava dok task visokog prioriteta čeka zbog kolizije u resursima. U real time sustavima raspored se određuje deterministički. Kernel preempta trenutni task koji se izvršava i prebaci na izvršavanje taska sa najvišim prioritetom koji je upravo postao

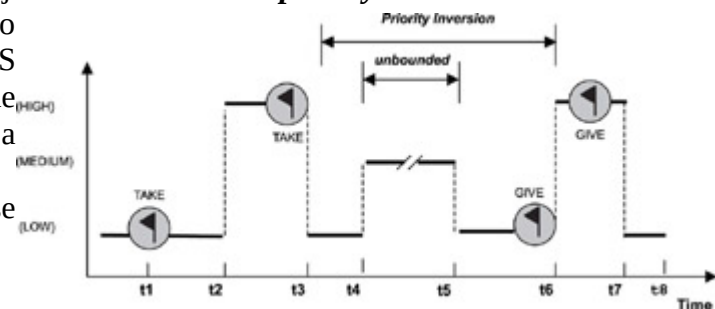
dostupan sa poznatim intervalom izvršavanja. To je moguće jer su taskovi nezavisni i to je neizbježno ako taskovi dijele resurse i sinkroniziraju aktivnosti. Priority inversion nastupa kada nezavisnost taska izlazi sa drugačijim prioritetima.

Kao na slici desno, visoko prioritetni task, dijeli resurse sa niskim koji uzima te zaključava resurs. Niski izvršava do t2 trenutka kada visoki postane dostupan. Scheduler preempta Niskog te prebacuje na Visokog. Visoki u t3 treba dijeljeni resurs, on se zablokira te Scheduler prebacuje na Niskog koji u t4 oslobađa resurs, događa se preempt, vraća se nazad na Visokog koji završava u t5 te vraća na Niskog koji završava u t6. Ovo se još naziva **bounded priority inversion**. Poznaje se vrijeme držanja resursa kod Niskog. Kad bi imali Srednji, on bi mogao preemptati niskog te tako natjerati Visokog da beskonačno čeka. Takav scenarij se zove **unbounded priority inversion**.



Opet je priority inversion nastupio u t3 samo što sada u t4 Srednji preempta Niskog. S obzirom da Srednji ne djeli resurse, vrijeme od t4 do t5 je nepoznato i povezano je sa trajanjem Srednjega.

Inversion se ne može izbjeći ali može se minimizirati sa protokolima.



Priority Inheritance protocol

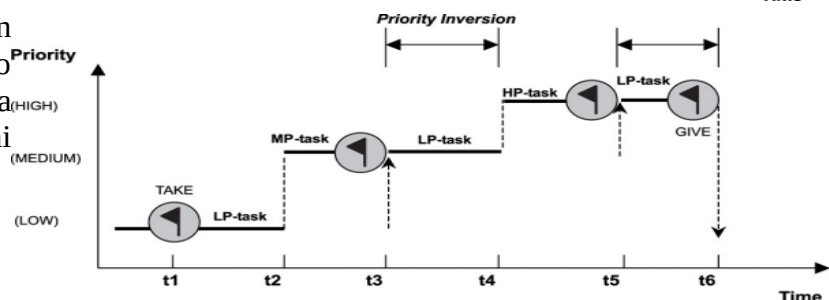
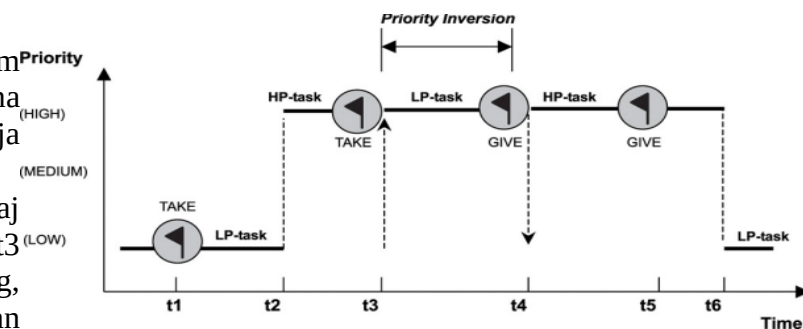
Protokol koji podiže prioritet taskova, ako taj task drži resurs koji treba task višeg prioriteta.

Pravila:

1. ako se R koristi, T je blokiran
2. ako je R slobdan, R je alociran T-u
3. kada task višeg prioriteta traži isti resurs, T-ov prioritet se diže na prioritet traženog taska
4. task se vraća na prijašnji prioritet kada oslobodi R

Sa desnih slika vidimo da u prošlom vremenu kada je bila aktualna Inverzija, sada dolazi do podizanja Niskog na razinu visokog.

Isto tako vidimo primjer za slučaj kada su tri razine izvršavanja. U t3 Niski se podiže na razinu Srednjeg, izvršava dok je Srednji zablokiran te u t4 resurs treba Visoki. On koristi resurs do t5 kada dolazi do podizanja prioriteta Niskog te kada on završi vraća ga na izvorni prioritet.



Deadlock ovdje može nastupiti ako Srednji treba još resursa koje treba i Visoki.

Priority Ceiling protokol

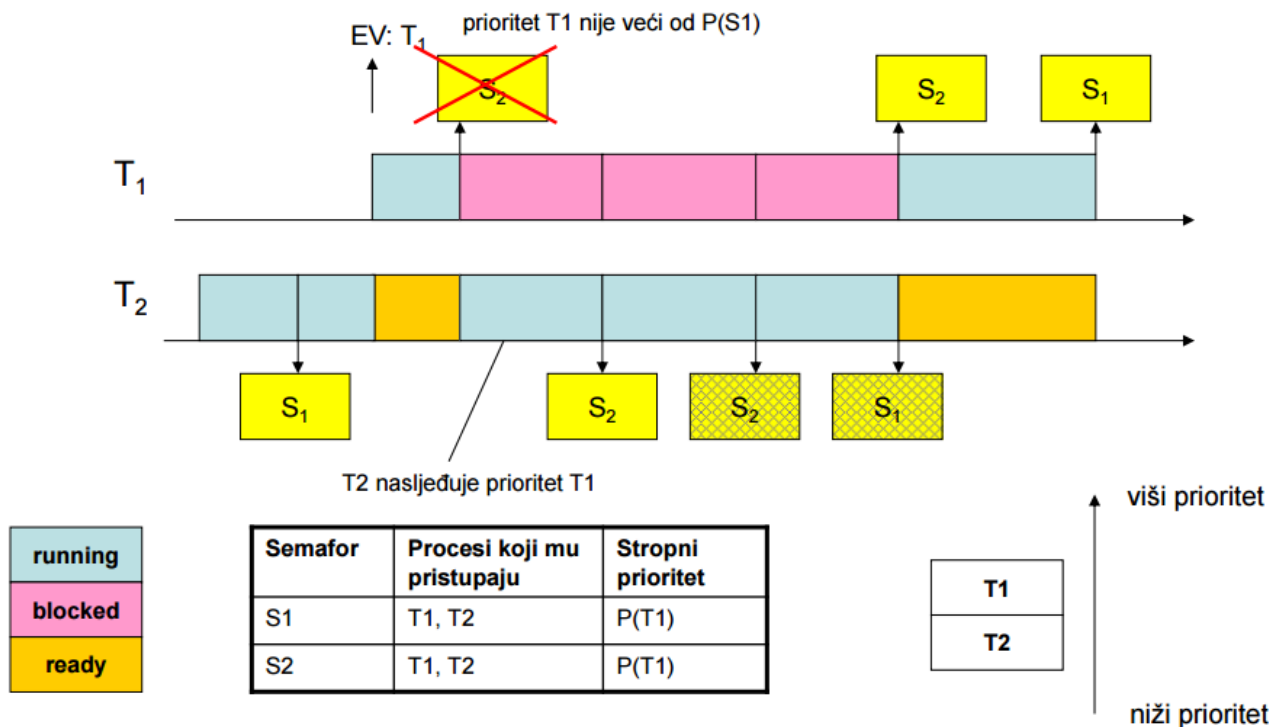
Znamo prioritet svakog taska kao i koje resurse trebaju. Trenutni priority ceiling je najviši priority ceiling svih resursa koji su u upotrebi.

Primjer: R1 ima stropni prioritet 4, R2: 9, R3: 10, R4 : 8, trenutni stropni prioritet sustava je 10.

Pravila:

1. ako se R koristi, T je blokiran
2. ako je R slobodan i prioritet T-a je viši od trenutnog stropnog prioriteta, R se dodjeljuje T-u.
3. ako trenutni stropni prioritet pripada jednom od resursa koje T trenutno drži, R se dodjeljuje T-u, u suprotnom slučaju T je blokiran
4. Task koji blokira T-a, nasljeđuje T-ov prioritet ako je viši i izvršava se na tom prioritetu sve dok ne oslobodi svaki resurs čiji stropni prioritet je viši ili jednak T-ovom prioritetu. Task tada vraća se na svoj prvobitni prioritet.

Deadlock nikad ne nastupa pod Priority Ceiling protokolom!



**8. bilo je još relativno dosta pitanja iz gradiva RTOS-a i Linuxa, s obzirom da su dva predavanja u pitanju, dosta bodova ovaj dio nosi tako da obavezno (za razliku od mene) dobro preči i skužiti ovaj dio. ne dolaze pitanja s prepoznavanjem kodova (bar meni nisu došla), tako da se valja na teoriju skoncentrirat

USMENI:

- usmeni je u pismenom obliku dobiješ tri pitanja i pišeš sve šta znaš i onda zoveš profesora da prokomentirate

1. algoritmi RMS i EDF - doslovno sve živo o njima

2. protokol nasljeđivanja prioriteta - također sve što znaš; treba opisati i razliku između bounded i unbounded slučaja inverzije prioriteta i također je pitanje što je deadlock.

Preporučam http://www.e-reading.club/bookreader.php/102147/Li%2C_Yao_-_Real-Time_Concepts_for_Embedded_Systems.html službenu literaturu iz koje su vadene sheme za slajdove, tamo su dosta dobro objašnjeni ovi primjeri s protokolima stropnog prioriteta, nasljeđivanja, deadlocka i sl.

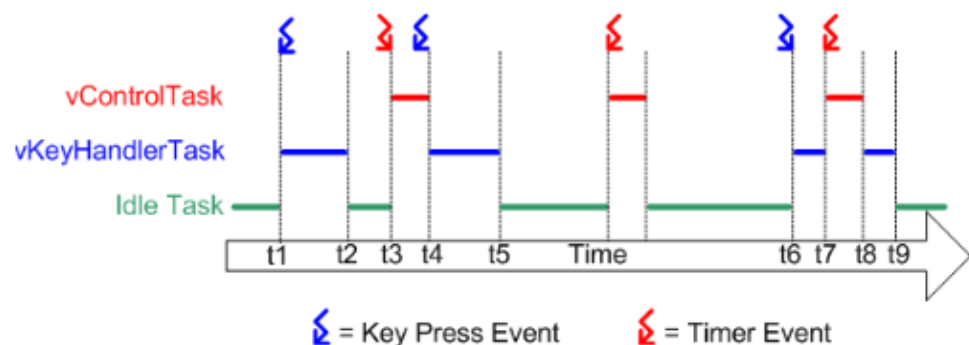
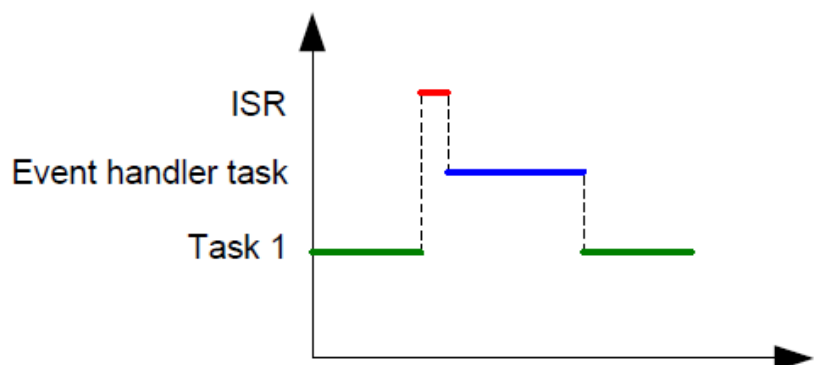
-opisano gore

3. odgođena obrada prekida u RTOS-u - znači sve živo o vanjskim prekidima (ISR) u usporedbi s task schedulingom. (56. slajd pa nadalje, 5. predavanja) ovo sam slabo znao jer sam općenito zadnja dva predavanja slabo prešao.

Odgođena obrada prekida (deferred interrupt processing) omogućava da nakon što je završio Task 1 sa izvođenjem, podigne se kao "zastavica" da bi se moglo prebaciti u drugo stanje ili obavijestiti drugo stanje o određenim podacima. Npr. radimo ADC pretvorbu. Kada je ona završena pokrene se Interrupt Service Routines (ISR) i pripremi se za ostale zadatke da su podaci spremni. Nakon što smo gotovi sa ISR, resetiramo zastavicu da se može ponovno upaliti na sljedećem pozivu. ISR obavlja minimalni potrebni posao i priprema zadatak za naknadnu obradu događaja.

Možemo za odgodu koristiti i binarni semafor.

S druge strane imamo **task**

**scheduling**

Sistem si stvori Idle task kada niti jedan drugi se ne izvršava. Vidimo sa slike gore da vKey čeka na tipku, a vControl čeka određeno vrijeme. U t1 tipka je stisnuta, vKey je višeg prioriteta pa se izvršava i završava u t2, sustav je opet u idle stanju. vControl se ponovno aktivira u t3 i u t4 završava kada je opet pritisnuta tipka.

Prvi dio gradiva:

1. PRIMASK, FAULTMASK, BASEPRI, STIR

PRIMASK- onemogućiti sve prekide osim NMI i HardFault

FAULTMASK – onemogućiti sve prekide osim NMI

BASEPRI – onemogućiti sve prekide određenog prioriteta

STIR – (software trigger interrupt register), prekid od strane softwarea

2. Bus fault, Memory management fault, Usage fault, Hard fault

Bus fault – pogreška na AHB sabirnici (pristup nemogućoj lokaciji, uređaj nije spreman za transfer...)

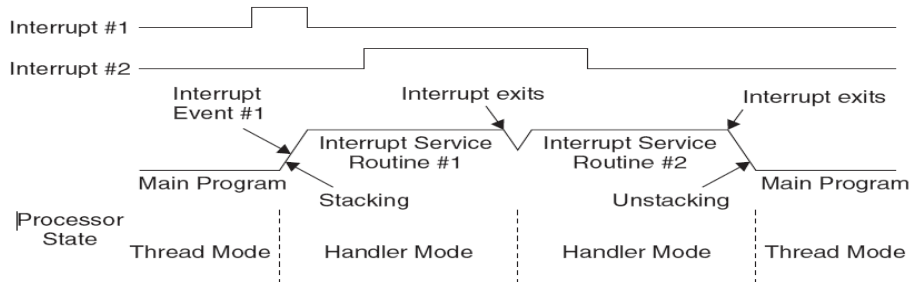
Memory management fault – MPU iznimke i pristup regiji koja nije definirana u MPU, pisanje u krivu regiju ...

Usage fault- nedefinirana/nepodržana instrukcija, pokušaj prebacivanja u krivo stanje, dijeljenje sa 0 ...

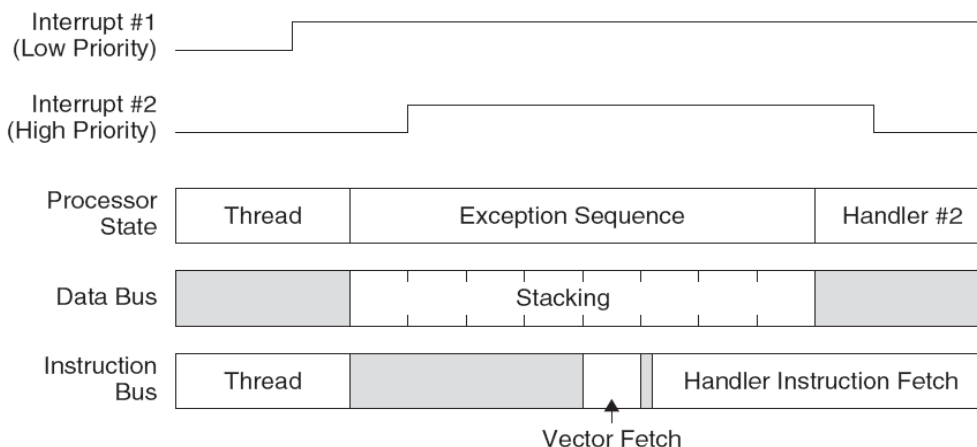
Hard Fault – prouzročen nekim od predhodnih slučajeva, ako se njihov handler ne može pokrenuti

3. Tail-chaining, Late arrival interrupts

Tail chaining, kada nastupi nova iznimka, a stara se još obrađuje koja je istog ili višeg prioriteta, nova se stavlja u stanje pended. Kada procesor završi sa trenutnom, umjesto POP, registri se vraćaju u stog i PUSH-aju te tako se preskače unstacking/stacking ciklus nakon obrade iznimke. Tako se smanjuje vrijeme između dvije iznimke te poboljšava latencija iznimki nižeg reda.



Late arrival interrupts, poboljšava latenciju iznimki višeg reda. Za vrijeme Stackinga iznimke niže razine, pojavi se iznimka više razine. Prvo će se obraditi iznimka više razine, a za to vrijeme niža iznimka će se staviti u stanje *pending*



4. **Priority level registar za 32 razine iznimki; preempt, subpriority, i napisati onaj registar priority lever registar**
-pogledati pitanje broj 5. sa jesenskog!

5. **Volatile – objasniti i programski kod napisati kratki odsjecak u c-u**

Volatile se koristi prilikom definiranja varijable koja se može mijenjati u kodu bez da smo mi direktno djelovali na nju. Na taj način govori se compileru da se vrijednost varijable može promijeniti bilo kad, bez da okolni kod djeluje na nju. Primjeri za nas:

Čitanje stanja registara:

```
uint8_t* pReg = (uint8_t *) 0x1234;
...
while (*pReg == 0) { ... }
...

mov ptr, #0x1234
mov a, @ptr
loop:
bz loop
```

```
uint8_t volatile * pReg = (uint8_t
volatile *) 0x1234;
...
while (*pReg == 0) { ... }
...

mov ptr, #0x1234
loop:
mov a, @ptr
bz loop
```

Prekidna rutina i globalne varijable

```
int etx_rcvd = false;
interrupt void rx_isr(void)
{
    ...
    if (ETX == rx_char) {
        etx_rcvd = true;
    }
    ...
}
void main() {
    ...
    // cekaj na znak
    while (!etx_rcvd) { }
    ...
}
```

```
volatile int etx_rcvd = false;
interrupt void rx_isr(void)
{
    ...
    if (ETX == rx_char) {
        etx_rcvd = true;
    }
    ...
}
void main() {
    ...
    // cekaj na znak
    while (!etx_rcvd) { }
    ...
}
```

Višezadačnost

```
int cntr;
void task1(void)
{
    cntr = 0;

    while (cntr == 0)
    {
        sleep(1);
    }
    ...
}
void task2(void)
{
    ...
    cntr++;
    sleep(10);
    ...
}
```

```
volatile int cntr;
void task1(void)
{
    cntr = 0;

    while (cntr == 0)
    {
        sleep(1);
    }
    ...
}
void task2(void)
{
    ...
    cntr++;
    sleep(10);
    ...
}
```

6. Kooperativna višezadaćnost bez istiskivanja: korutine, nedostaci naspram višezadaćnosti s istiskivanjem, programski kod

Kooperativna višezadaćnost je vrsta multitaskinga u kojoj OS nikad ne inicira contex switch (promjenu stanja i vraćanje nazad) od nekog drugog procesa. Umjesto toga proces dobrovoljno daje kontrolu periodično ili u stanju mirovanja – da se aplikacije vrte istovremeno. Nedostatak: složeno programiranje, ne može se istiskivati, ne znamo kada će se što dodjeliti CPU, sa procesom se komunicira putem globalnih varijabli "coroutines" funkcije se izvode paralelno, ali ne mogu prekinuti jedna drugu već u određenom trenutku dobrovoljno prepuštaju kontrolu.

```
void ProcessA() {
while(true) {
switch(StateProcessA) {
case 1: phase1A(); break;
case 2: phase2A(); break;
case 3: phase3A(); break;
}
cswitch(); // call task dispatcher
}
}

void ProcessB() {
while(true) {
switch(StateProcessB) {
case 1: phase1B(); break;
case 2: phase2B(); break;
case 3: phase3B(); break;
}
cswitch(); // call task dispatcher
}
}
```

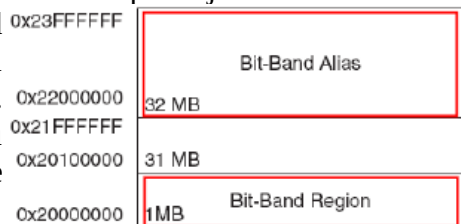
Višezadaćnost s istiskivanjem (preemptive multitasking) – najmoćniji oblik multitaskinga- potreban je Task Scheduler te OS odlučuje kada dolazi do contex switchinga (prekine niže prioritete, da višima) te OS odlučuje koji će se zadatak izvesti nakon zamjene.

7. Bit-banding, objasniti, kod u C-u

Postavljanje vrijednosti bita u jednoj load/store operaciji.

Sustav uzme dio memorije (bit-band region) i mapira svaki bit u toj regiji na drugoj regiji (bit-band alias). Prednost je što pisanje i čitanje u Regiju, radi automatski pisanje/čitanje u alias.

U C-u nema posebnih naredbi za bit banding već se definira kao desno navedeno



The diagram illustrates the SRAM memory layout. It shows a vertical stack of memory regions. The top region is labeled 'Bit-Band Alias' and spans from address 0x23FFFFFF to 0x22000000, with a size of 32 MB. Below it is a region labeled '31 MB' spanning from 0x21FFFFFF to 0x20100000. The bottom region is labeled 'Bit-Band Region' and spans from 0x20000000 to 0x20000000, with a size of 1 MB. To the right of the diagram, the text 'SRAM regija' is written.

```
#define DEVICE_REG0 ((volatile unsigned long *) (0x40000000))
#define DEVICE_REG0_BIT0 ((volatile unsigned long *) (0x42000000))
#define DEVICE_REG0_BIT1 ((volatile unsigned long *) (0x42000004))
...
// izravan pristup registru putem adrese - upis okteta
*DEVICE_REG0 = 0xAB;
...
// postavljanje bita 1 bez korištenja bit-bandinga
*DEVICE_REG0 = *DEVICE_REG0 | 0x2;
...
// postavljanje bita 1 korištenjem bit-bandinga (bit-band alias re-
gija)
*DEVICE_REG0_BIT1 = 0x1;
```

8. Bilo je pitanje vezano za neke registre sta koji predstavlja vezano za prekide

9. Na zaokruzivanje: o onim kritičnim sekcijama i tome

Kritična sekcija je situacija kada istoj varijabli žele pristupiti dva procesa. Recimo proces A(b=x+5), proces B(x=3+z). Kada B završi možemo dati A da očita vrijednost "x".

10. Na zaokruzivanje: o Cortexu

11. setPend, clrPend, clrena, setENA- sta rade

setPend i clrPend – registri za postavljanje i resetiranje pending statusa pojedinih prekida (kao omogućivanje i onemogućivanje prekida)

SETENA – omogućavanje prekida

CLRENA – micanje enable bita prekida

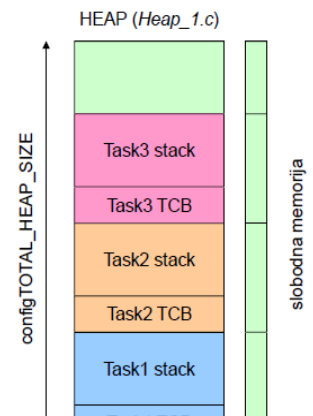
Drugi dio gradiva

1. Praktični zadatak s onim Heap_1,2,3 - moraš znat kak rade..tipa ovaj Heap_2 ne spaja slobodne blokove

Heap_1

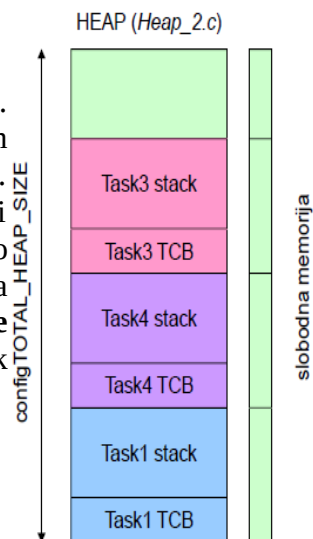
Najjednostavniji algoritam izravne dodjele blokova bez mogućnosti oslobađanja memorije. Ako želimo mijenjati veličinu heapa, preko configTOTAL_HEAP_SIZE definira se veličina polja za heap. Pogodno ga je koristiti prilikom pokretanja OS da bi inicijalizirali zadatke i objekte bez da ih naknadno brišemo:

- unaprijed rezerviramo prostor koji kasnije ne oslobađamo
- kompaktan kod i nema problema sa determinizmom izvođenja programa, fragmentacijom...



Heap_2

"**Best fit algoritam**" dinamički alokira memoriju, a može ju i oslobađati. On traži slobodan blok koji je po veličini najbliži traženom memorijskom bloku (npr. imamo tri bloka od 50, 250 i 1000B, nama je potrebno 200B. Algoritam uzima blok od 250 i zauzima 200B, a od preostalih 50B pravi novi blok (tako smanjuje unutarnju fragmentaciju). Ako želimo osloboditi blok od 200B, dobivamo dva slobodna bloka 200B i 50B. Za razliku od free() funkcije, Heap_2 **ne spaja susjedne slobodne blokove** (problem eksterne fragmentacije)). Pogodan algoritam ako ćemo uvijek iste veličine blokove oslobađati i zauzimati.



Heap_3

malloc() i free() implementacija na thread-safe način (privremeno se onemogućuje TaskScheduler).

Ovdje configTOTAL_HEAP_SIZE nema utjecaja, određuje se kroz konfiguraciju linkera.

2. Praktični s protokolom nasljeđivanja prioriteta i stropnog prioriteta

-opisano gore

3. Crtanje sad ne znam više nekog semafora ili redova poruka.

Objekt jezgre OS nad kojim zadaci (tasks) mogu obavljati dvije osnovne operacije (zuzmi i otpusti), služi za sinkronizaciju i međusobno isključivanje zadataka kod pristupa resursima. On nam omogućuje realizaciju kritičnih sekcija, mutexa, signalizaciju između procesa...

Mehanizam koji nam omogućava koordinaciju komunikacijskog procesa.

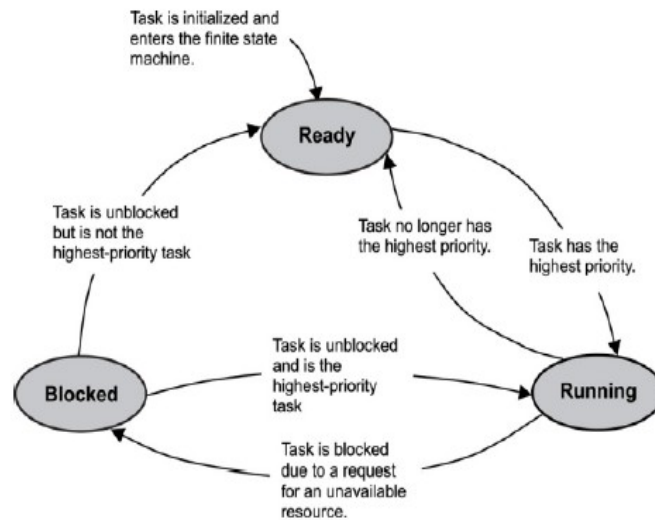
```
void *pvPortMalloc(size_t xWantedSize) {
void *pvReturn;
    vTaskSuspendAll();
    pvReturn = malloc(xWantedSize);
    xTaskResumeAll();
    return pvReturn;
}
```

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

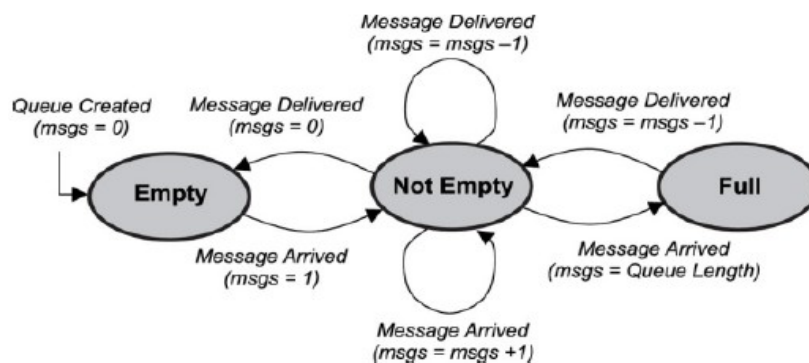
```
/* some nonprotected operations here */
P(); /* wait for semaphore */
/* do protected work here */
V(); /* release semaphore */
```

4. Crtanje state machine kod FreeRTOSa (21. slajd 5. predavanja) i Oni automati svi zivi.

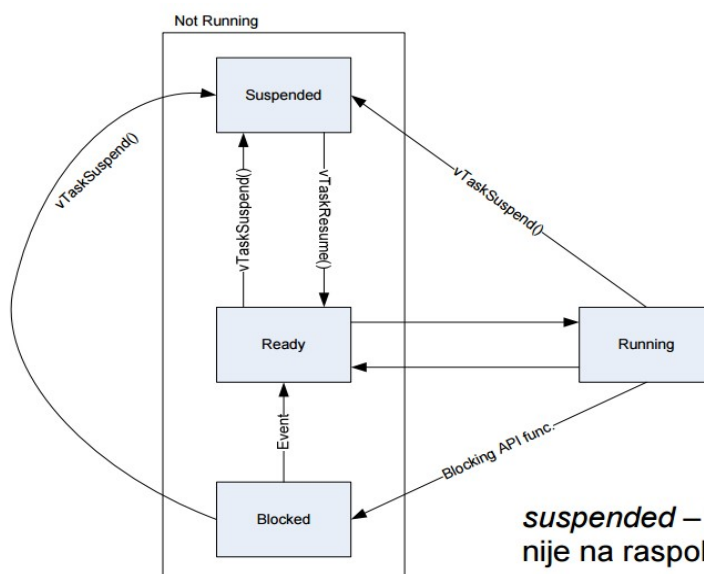
FSM stanja zadataka



FSM za redove poruka / isti model za pipe FSM samo message=data, queue=pipe

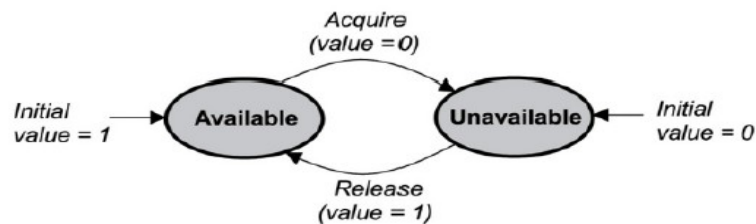


Dijagram stanja zadataka



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

FSM binarnog semafora



5. Direktoriji kod linuxa

chmod <permissions><files>

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

/	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

6. 5 Razlika mutex i semafor

Mutex je objekt koji posjeduje thread – vlasništvo u mutexu!

On dopušta samo jedan thread da pristupi resursu

Semafor je "signalni" mehanizam. Može dopustiti veći broj threadova da pristupa djeljnom resursu.

7. Zasto linux nije RTOS

Linux nema preemptive jezgru kakvu ima RTOS.

Paging- vrijeme zamjena stranica i RAM-a nije ograničeno - nedeterminizam

Sistemske pozivi jezgre mogu trajati dosta dugo i nije moguće ga prekinuti.

Linux koristi Complete Fair Scheduling koji je dobar za sustave sa GUI-jem, ali u potpunoj suprotnosti ciljevima RTOS-a.

Request reordering – prioritetniji zadaci mogu čekati zbog toga što su prednost dobile druge jedinice. Tako se učinkovitije iskorištava sklopovlje.

Batching – uzastopno se obavlja niz istovrsnih operacija da bi se povećala srednja brzina odziva sustava

8. One dodjele zadataka kod linuxa

Standardna jezgra linuxa podržava SCHED_OTHER (normalni procesi) te SCHED_FIFO, SCHED_RR (real time procesi). Svakom procesu dodjeljuje se prioritet 0-99.

SCHED_OTHER: procesi koji imaju 0 prioritet raspoređuju se po Fair Scheduler principu

SCHED_FIFO: procesi prioriteta 1 i višeg, može istisnuti procese 0, i on sam može biti istisnut od strane višeg procesa. Nakon prepuštanja procesora, postavlja se na kraj FIFO strukture i izvodi se sve dok dobrovoljno ne prepusti procesor

SCHED_RR: isto kao FIFO, samo što po *round robin* principu proces se može prekinuti i postaviti na kraj FIFO strukture i prije nego što eksplicitno prepusti kontrolu, a nakon određenog vremenskog intervala

9. Zadatak s redom (dobijes program i pitanje ti je sta se ispisuje)

Globalna struktura za razmjenu podataka između zadataka. S obzirom da su FIFO strukture, podaci se mogu umetati s obje strane reda. Zadaje se broj i veličina elementa prilikom kreiranja reda. Proces je da se podaci prilikom stavljanja u red kopiraju iz memorije polaznog zadatka te stavljaju se u memoriju reda. Iz memorije reda prebacuje se onda u određeni zadatak.

Kad se piše ili čita u red, zadatak se automatski stavlja u stanje blokiranja, logično ako je više zadataka u redu, onaj s najvišim prioritetom ima prednost, ako su svi istog – onda onaj koji najdulje čeka.

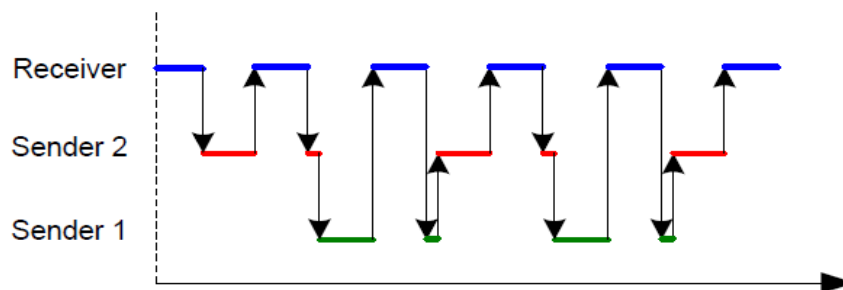
xQueueSendToBack()- šalje podatke na kraj reda

xQueueSendToFront()- šalje podatke na početak reda

!Oboje se ne smije pozivati iz prekida, za to postoji xQueueSendToBackISR()

xQueueReceive()- čitamo podatak

uxQueueMessagesWaiting()-čitamo broj poruka u redu



Ispis:

```
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
```

10. Sta je configKERNEL_INTERRUPT_PRIORITY, a sta configMAX_SYSCALL_INTERRUPT_PRIORITY

configKERNEL_INTERRUPT_PRIORITY – konstanta za definiranje prioriteta prekida raspoređivača zadataka

configMAX_SYSCALL_INTERRUPT_PRIORITY – konstanta za definiranje najveće vrijednosti prioriteta prekida koji koriste interrupt-safe API funkcije OS-a (to su one funkcije koje su prilagođene pozivu is prekida, imaju na kraju _FromISR)

11. Bilo je ono s promjenom vlasništva u Linuxu (19-21 slajd 6. predavanje)

12. Isto predavanje 23-24 slajd - što sadrži određena datoteka (došlo ih je 4)

Cross-compilation toolchain – prevoditelj što se izvodi na radnom računalu, generira kod za drugu programsku platformu

Bootloader- izvodi se odmah nakon restarta, početna inicijalizacija, učitava memoriju, pokreće jezgru OS-a (nije Grub, već U-Boot)

Linux Kernel – izvorni kod koji se prevodi za određenu platformu, upravljanje procesorima i memorijom ...

C library – sučelje između jezgre i korisničke aplikacije

13. Onda na zaokruživanje je bilo svega o freertosu i o linuxu to su bila jedno 4 zadatka**14. Onda moras nadopuniti sta koja funkcija radi Od linuxa****15. onda si moro znat nesta za one dvije funkcije ENTER_critical()**

Kritična sekcija počinje sa taskENTER_critical(), a završava sa taskEXIT_critical(). Između ta dva poziva, omogućeni su onemogućeni (mogući su samo oni koji su veći od configMAX_SYSCALL_INTERRUPT_PRIORITY), isključuje mogućnost zamjene konteksta

16. on ti da semafor ili mutex i kad se pali imas takav primjer tamo kod stropnog**17. bio je jedan zadatak na kraju s funkcijom koji se zablokira jer svi uzimaju semafor a on te pita kolka je vrijednost varijabe i u 1050 ms****18. i ovo ostalo je bilo na zaokruživanje sta je točno sta ne****19. greske su te kao na slajdovima****20. configUSE_IDLE_HOOK****configUSE_TICK_HOOK**

Kada nema taskova većeg prioriteta, možemo staviti procesor u low power stanje. Ako su svi zadaci blokirani, izvodi se isto idle zadatak sa prioritetom 0. To omogućavamo sa opcijom u configUSE_IDLE_HOOK 1 u FreeRTOSConfig.h

Ako želimo implementirati i timer u to sve, onda omogućimo i configUSE_TICK_HOOK