Fakultet elektrotehnike i računarstva
Zavod za elektroničke sustave i obradbu informacija

Oblikovanje programske potpore za ugradbene računalne sustave

Operacijski sustav FreeRTOS

Hrvoje Džapo

ak. god. 2022/2023.

Fakultet elektrotehnike i računarstva

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.

Fakultet elektrotehnike i računarstva

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.

Fakultet elektrotehnike i računarstva

Podržane platforme

Proizvođač	Arhitektura	IDE
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

Fakultet elektrotehnike i računarstva

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)

Fakultet elektrotehnike i računarstva

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
- OpenRTOS
  - inačica koja se distribuira pod komercijalnom licencom (nema veze s GPL), a uključuje i dodatne module (USB, *file system*, TCP/IP itd.)
- www.highintegritysystems.com

Fakultet elektrotehnike i računarstva

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*

Fakultet elektrotehnike i računarstva

Konvencije korištene u kôdu

- varijable – kod imenovanja varijabli koriste se dogovorni prefiksi 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*”

Fakultet elektrotehnike i računarstva

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
```

Fakultet elektrotehnike i računarstva

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 prosljeđuju 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

Fakultet elektrotehnike i računarstva

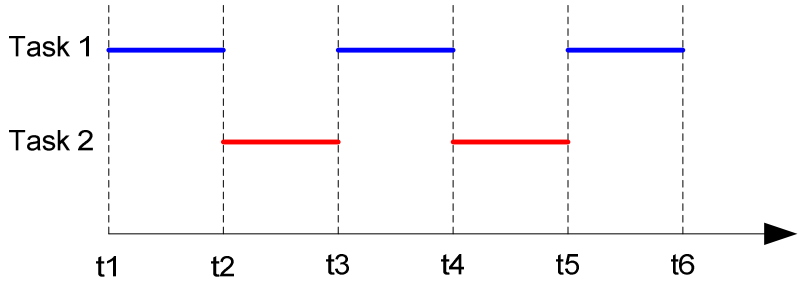
Primjer: dva zadatka s istim prioritetom

```
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
     }
}

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(
            vTask1,     // pokazivač na task funkciju
            "Task 1",   // ime zadatka (informativno)
            1000,       // dubina privatnog stoga zadatka
            NULL,       // ne prosljedjuju se podaci zadatku
            1,          // prioritet = 1
            NULL);      // ne koristi se task handler
    // kreiraj zadatak 2
    xTaskCreate(
            vTask2, "Task 2", 1000, NULL, 1, NULL );
    // pokreni OS
    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!

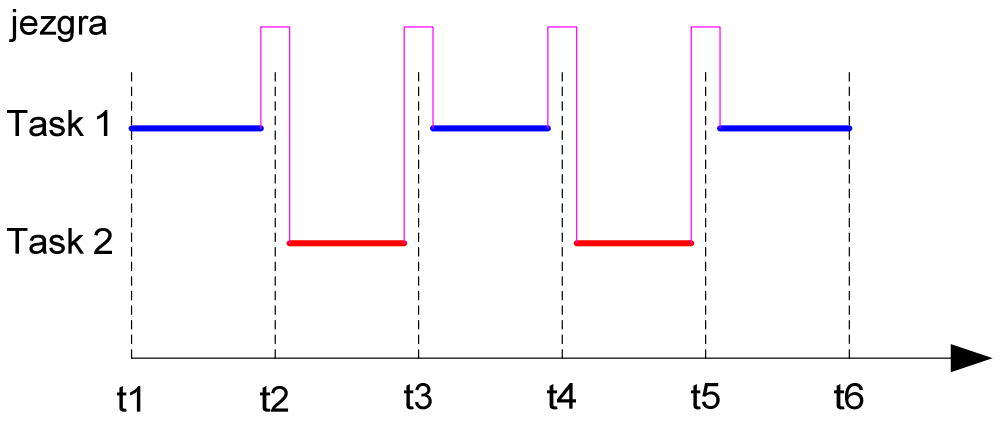
Fakultet elektrotehnike i računarstva

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?

Fakultet elektrotehnike i računarstva

Primjer: dva zadatka s različitim prioritetom

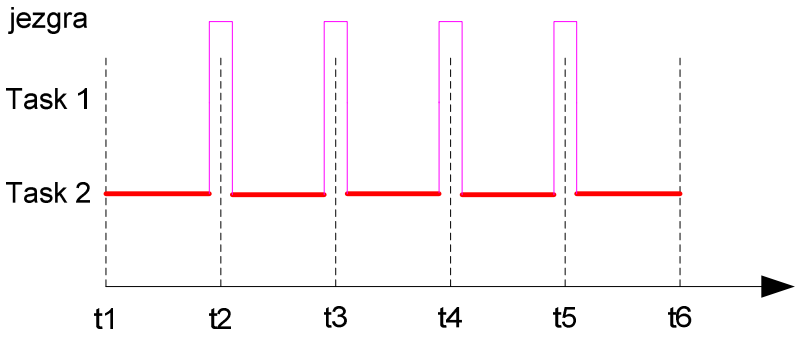
```
void vTaskFunc(void *pvParameters) {
char *pcTaskName;
volatile unsigned long ul;
    pcTaskName = (char*) pvParameters;
    while(1) {
       vPrintString(pcTaskName);
       for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {}
    }
}

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
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
...
```



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

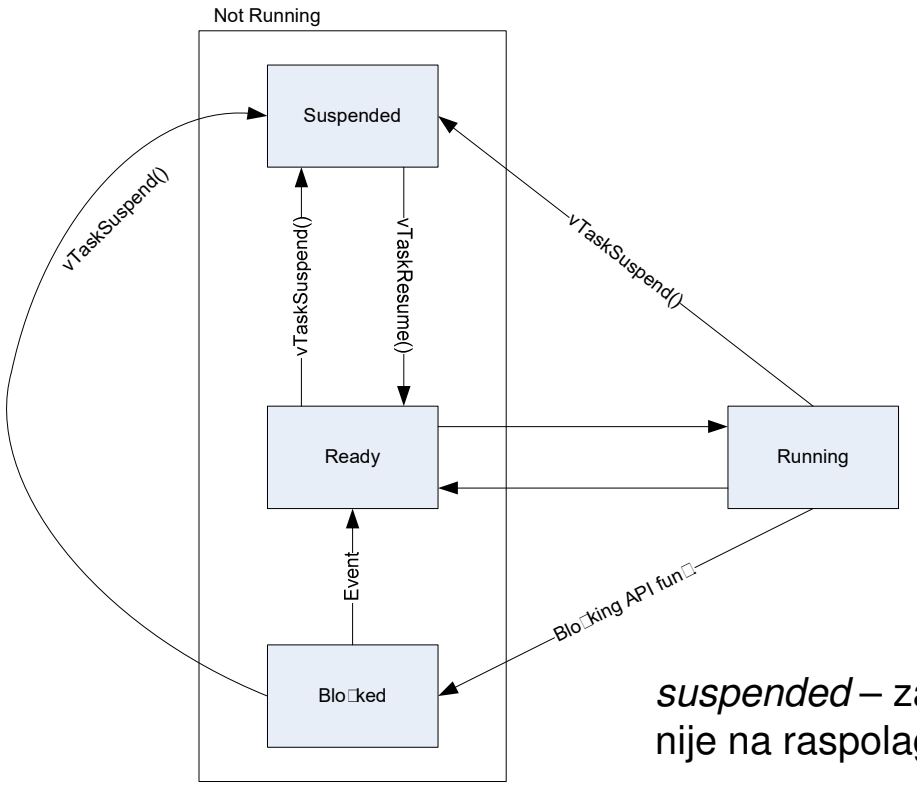
Fakultet elektrotehnike i računarstva

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

Fakultet elektrotehnike i računarstva

Blokiranje zadatka

- funkcija za vremensko blokiranje zadatka:

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

- *xTicksToDelay* – broj perioda raspoređivača zadataka (*tick count*) koliko zadatak treba ostati u blokiranom *(blocked)* stanju, prije nego što 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)

Fakultet elektrotehnike i računarstva

Primjer: dva zadatka s različitim prioritetom i vremenskim blokiranjem

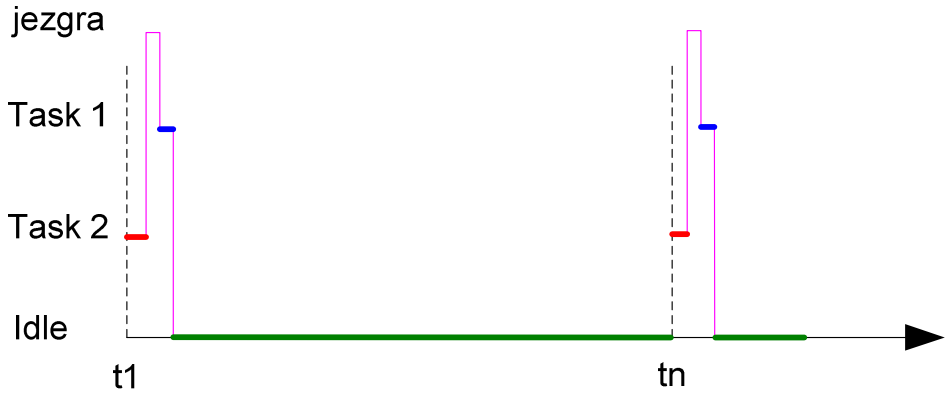
```
void vTaskFunc(void *pvParameters) {
char *pcTaskName;
volatile unsigned long ul;
    pcTaskName = (char*) pvParameters;
    while(1) {
       vPrintString(pcTaskName);
       vTaskDelay( 250 / portTICK_RATE_MS);
    }
}

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

Fakultet elektrotehnike i računarstva

Blokiranje zadatka

- *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

Fakultet elektrotehnike i računarstva

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 garatira da će se zadatak izvesti
          // za točno 250 ms (ovisno o prekidima i drugim zadacima 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 ) );
     }
}
```

Fakultet elektrotehnike i računarstva

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, mjerenje opterećenosti procesora, upravljanje stanjem niske potrošnje procesora i sl.
- deklaracija Idle hook funkcije (ime i prototip moraju egzaktno odgovarati!)

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

Fakultet elektrotehnike i računarstva

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

Fakultet elektrotehnike i računarstva

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 prioriteta 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

Fakultet elektrotehnike i računarstva

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);
}
```

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
...

Fakultet elektrotehnike i računarstva

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 izgladnjivanja *Idle* zadatka 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*)

Fakultet elektrotehnike i računarstva

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);
}
```

Fakultet elektrotehnike i računarstva

Primjer: Dinamičko kreiranje i brisanje zadatka

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!

Fakultet elektrotehnike i računarstva

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

Fakultet elektrotehnike i računarstva

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)

Fakultet elektrotehnike i računarstva

Kreiranje reda

- redovi se moraju eksplicitno 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

Fakultet elektrotehnike i računarstva

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 prosljeđen 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)

Fakultet elektrotehnike i računarstva

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!

Fakultet elektrotehnike i računarstva

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 prosljeđen prilikom kreiranja reda
- *xTicksToWait* – najdulje vrijeme koje zadatak može ostati blokiran prilikom čitanja praznog reda (parametar označava broj *tickova* raspoređivača zadataka); 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)

Fakultet elektrotehnike i računarstva

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

Fakultet elektrotehnike i računarstva

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*()

Fakultet elektrotehnike i računarstva

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);
}
```

Fakultet elektrotehnike i računarstva

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();
    }
}
```

Fakultet elektrotehnike i računarstva

Primjer: čitanje iz reda s blokiranjem

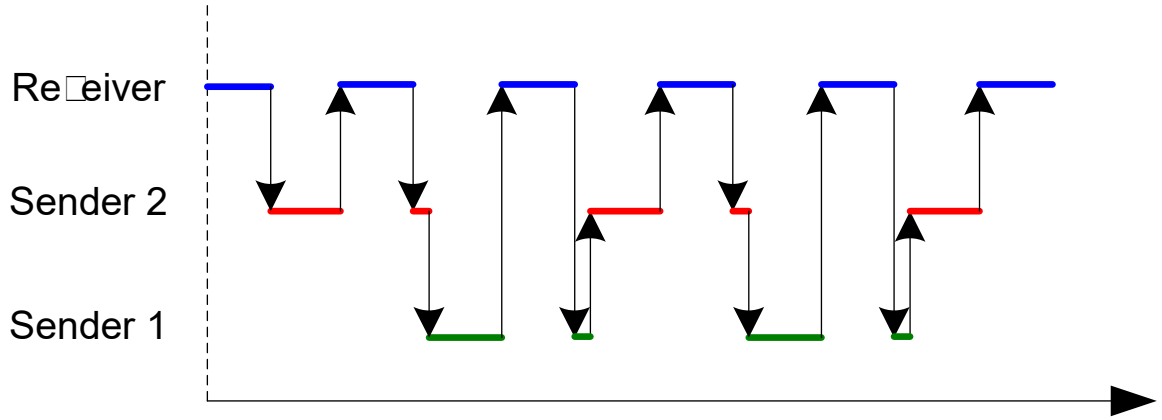
```
void vReceiverTask( void *pvParameters)
{
long lReceivedValue;        // memorijski prostor za prihvat podataka iz reda
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    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" );
         }
         // prihvat 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" );
         }
    }
}
```

Fakultet elektrotehnike i računarstva

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* višezadaćnosti

Primjer: čitanje iz reda s blokiranjem

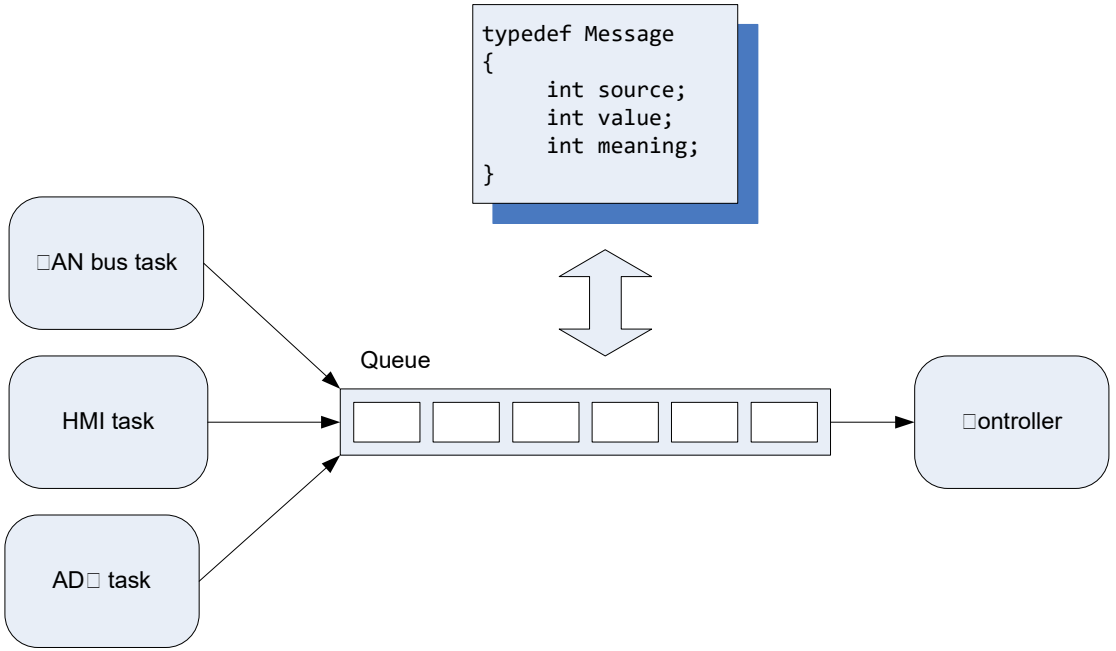


Ispis:

Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
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);
}
```

Fakultet elektrotehnike i računarstva

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

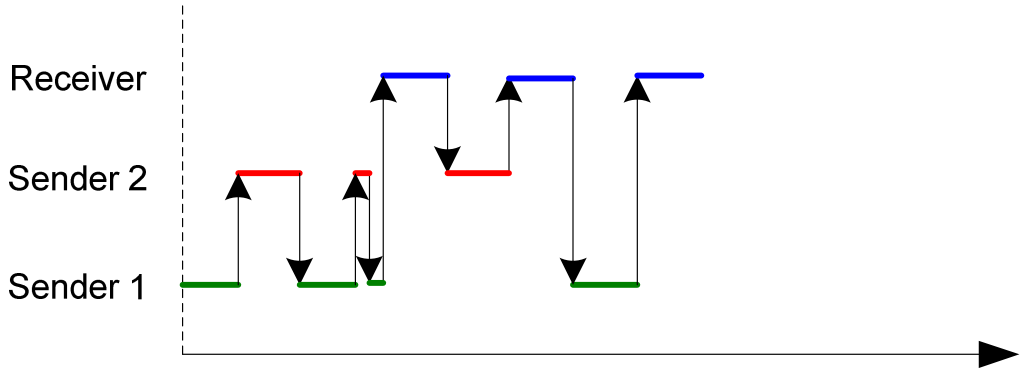
```
void vReceiverTask(void *pvParameters) {
xMESSAGE xReceivedStructure;     // memorija za pohranu poruke
portBASE_TYPE xStatus;
    while(1) {
           // obzirom da je receiver zadatak nize razine prioriteta, nece se probuditi sve dok
           // sender taskovi ne blokiraju na punom redu:
           if (uxQueueMessagesWaiting(xQueue) != 3) {
                     // pogreška!
                     vPrintString( "Queue should have been full!\r\n" );
           }
           // neblokirajuce citanje, jer se ocekuje 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" );
           }
    }
}
```

Fakultet elektrotehnike i računarstva

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 zadatka 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 zadatku, sve dok traje *time slice* tog zadatka
  - 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
...

Fakultet elektrotehnike i računarstva

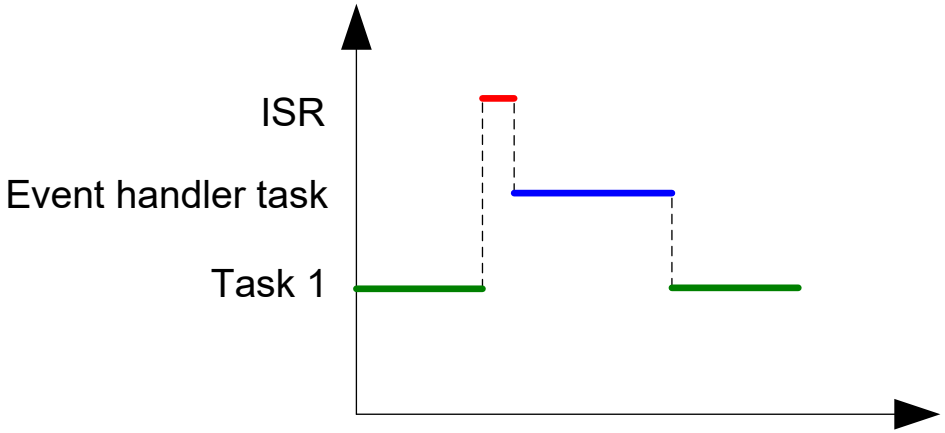
Upravljanje prekidima

- mogućnosti detekcije događaja:
  - prozivanje 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

Fakultet elektrotehnike i računarstva

Odgođena 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 osloboditi u prekidu i time zadatku signalizirati pojavu događaja
- prioritetni događaji – dati viši prioritet zadacima za obradu



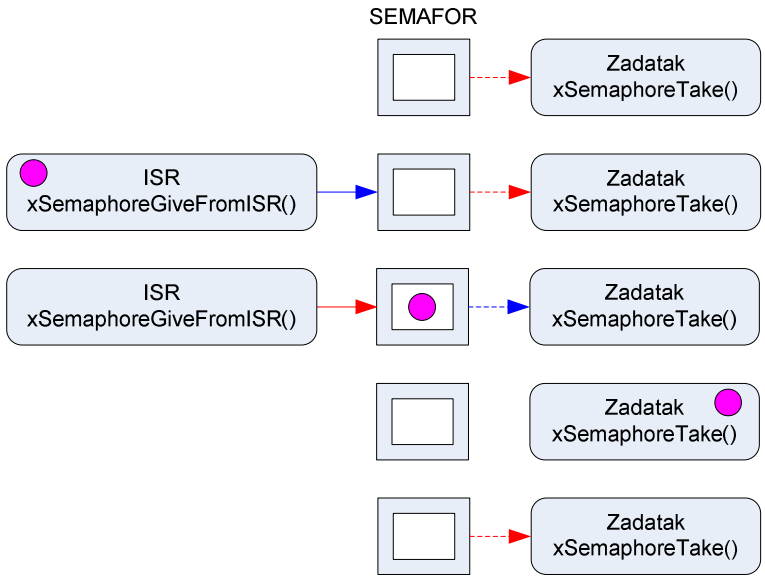
Binarni semafori

- kreiranje semafora:

```
xSemaphoreHandle xSemaphoreCreateBinary();
```

  - povratna vrijednost - handle na kreirani binarni semafor

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



Fakultet elektrotehnike i računarstva

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

Fakultet elektrotehnike i računarstva

Binarni semafori

- uzimanje semafora:

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

  - *xSemaphore* – handle semafora koji se želi zauzeti
  - *xTicksToWait* – najdulje 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

Fakultet elektrotehnike i računarstva

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)

Fakultet elektrotehnike i računarstva

Primjer

```
xSemaphoreHandle xBinarySemaphore;

int main(void)
{
   // kreiraj binarni semafor
   xBinarySemaphore = xSemaphoreCreateBinary();
   // ... konfiguriraj prekide
   // provjera da li je semafor kreiran:
   if (xBinarySemaphore != NULL) {
        // kreiraj 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);
}
```

Fakultet elektrotehnike i računarstva

Primjer

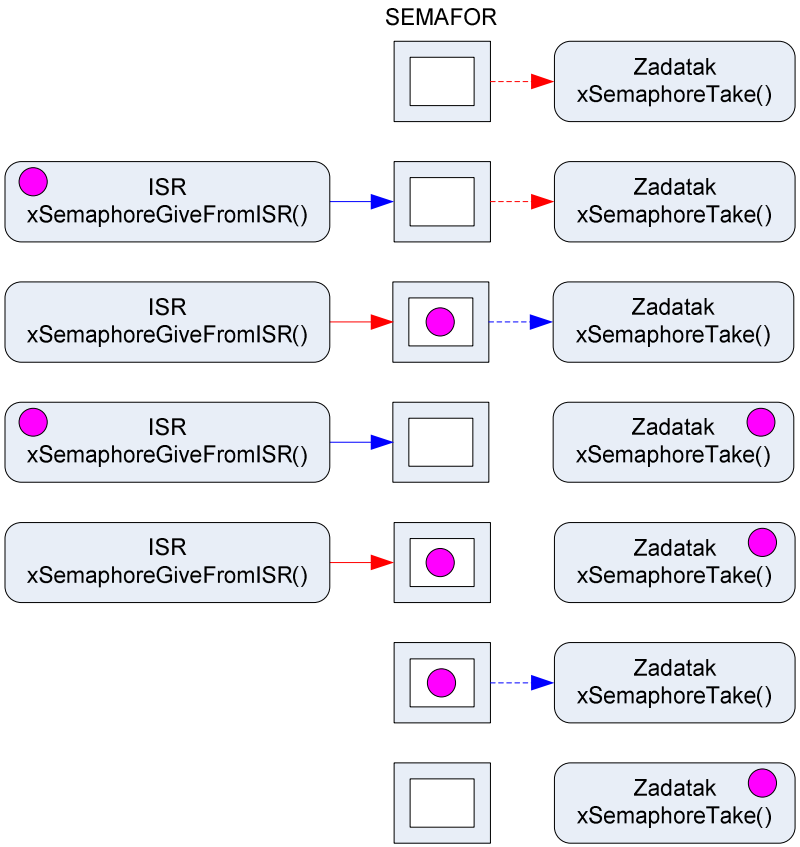
```
void vHandlerTask(void *pvParameters) {
   while(1) {
        // Zadatak ceka (beskonacno) na dogadjaj koristenjem
        // sinkronizacijskog semafora xBinarySemaphore.
        // Inicijalno kreirani semafor (prije pokretanja OS-a) je
        // neprolazan.
        xSemaphoreTake(xBinarySemaphore, 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();
   }
}
```

Fakultet elektrotehnike i računarstva

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

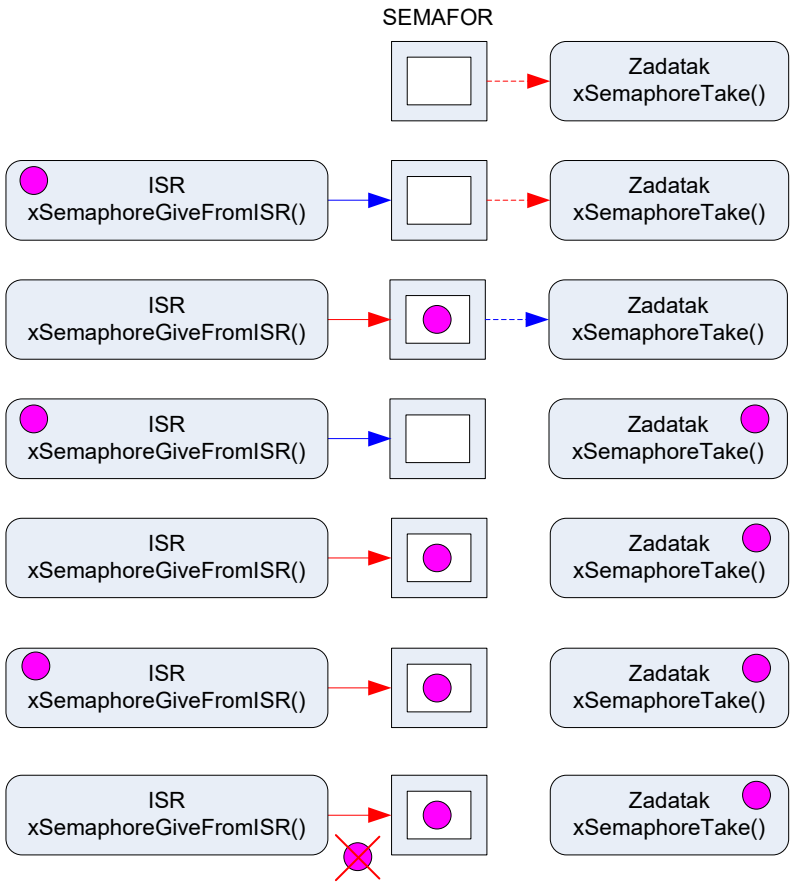
Binarni semafor – *event latching*



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

Binarni semafor ne može zapamtiti više od jednog dodatnog događaja!

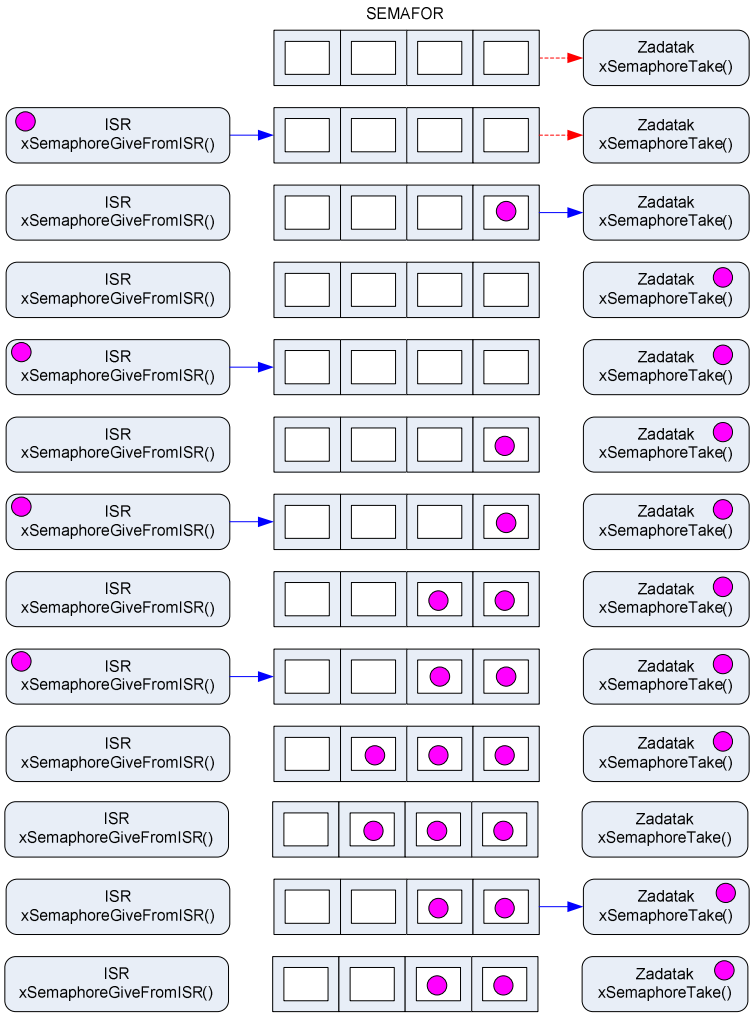
Binarni semafor – *event latching*



Fakultet elektrotehnike i računarstva

Brojeći semafori

- konceptualno proširenje binarnog semafora – može se promatrati kao red koji može primiti N “tokena”
- *event latching* – mogućnost prihvaćanja većeg broja događaja u red zadatka za obradu ako prebrzo stižu
- tipične primjene brojećih 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 oslobađanje resursa, tj. da vrijednost semafora postane pozitivna
    - u ovom slučaju semafor se tipično inicijalizira na vrijednost koja predstavlja broj raspoloživih resursa


SEMAFOR
Zadatak
xSemaphoreTake()
ISR
xSemaphoreGiveFromISR()

Fakultet elektrotehnike i računarstva

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

Fakultet elektrotehnike i računarstva

Primjer – *Event Latching*

```
xSemaphoreHandle xCountingSemaphore;
int main(void) {
   // kreiraj brojeci 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();
   }
}
```

Fakultet elektrotehnike i računarstva

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*:
  - *xQueueSendToFront**FromISR***()
  - *xQueueSendToBack**FromISR***()
  - *xQueueReceive**FromISR***()
- deklaracije ISR funkcija za komunikaciju putem redova su nešto drugačije od odgovarajućih “običnih” funkcija

Fakultet elektrotehnike i računarstva

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 zamjenu konteksta treba obaviti prije izlaska iz prekida
- povratna vrijednost – *pdPASS* – podatak je uspješno zapisan u red, *errQUEUE_FULL* inače

Fakultet elektrotehnike i računarstva

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 prosljeđuje je putem reda *xStringQueue* zadataku *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);
}
```

Fakultet elektrotehnike i računarstva

Primjer

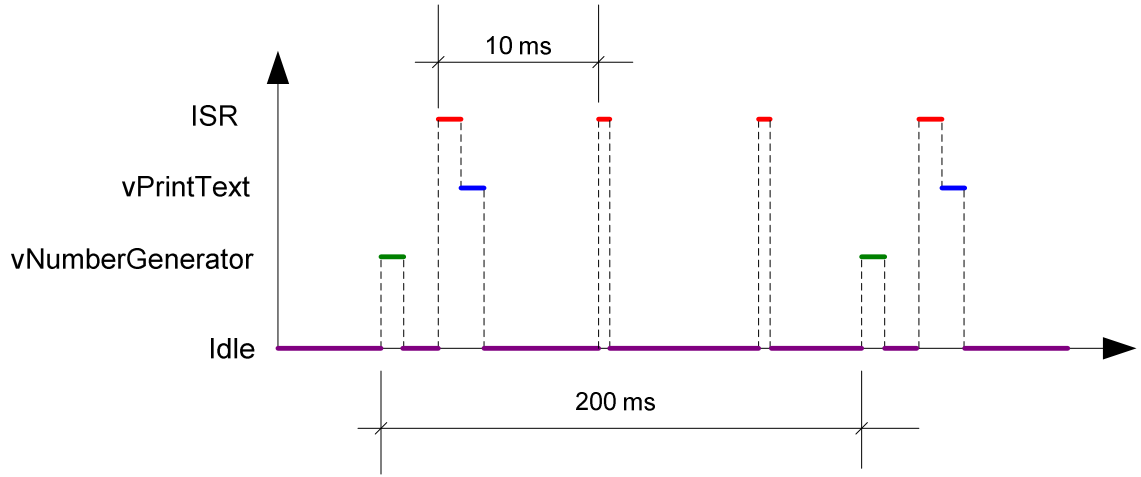
```
// zadatak koji generira cijele brojeve i salje ih u
// timer0 prekid putem xIntegerQueue reda
// (generira se 5 brojeva u 200 ms, a timer0 radi s frekvencijom 100 Hz)

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");
    }
}
```

Fakultet elektrotehnike i računarstva

```
// 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 zadataku 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 prosljedjuju 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);
    }
}
```



Fakultet elektrotehnike i računarstva

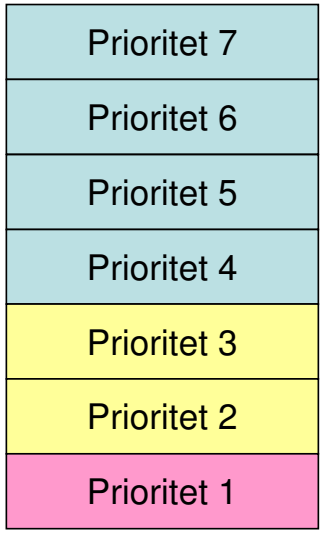
Prekidi i jezgra OS-a

- 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 sufiksu *_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

Fakultet elektrotehnike i računarstva

Upravljanje resursima

- problem pristupa dijeljenim resursima u višezadaćnom 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

Fakultet elektrotehnike i računarstva

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)!

Fakultet elektrotehnike i računarstva

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*!

Fakultet elektrotehnike i računarstva

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!

Fakultet elektrotehnike i računarstva

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);
```

Fakultet elektrotehnike i računarstva

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 stavljaju 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*()

Fakultet elektrotehnike i računarstva

Alternativna implementacija FreeRTOS kritične sekcije

- primjer alternativne implementacije kritične sekcije – ispis stringa:

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

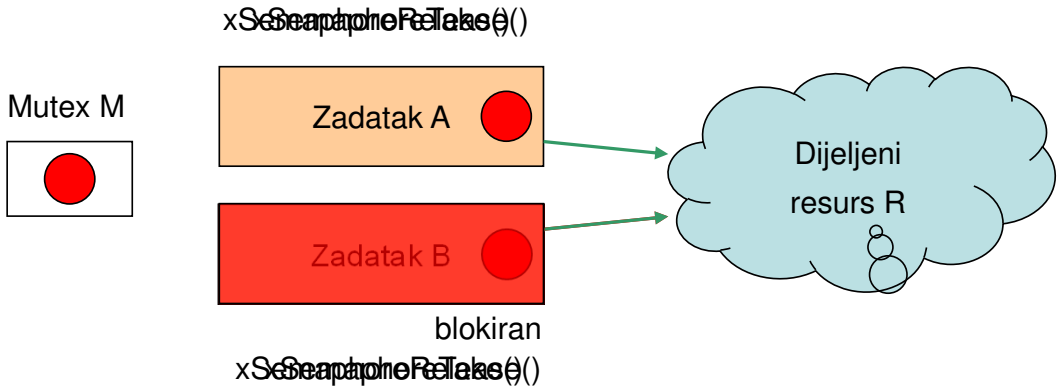
Fakultet elektrotehnike i računarstva

FreeRTOS i Mutex objekti

- običnibinarni 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 zadataka 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:



Fakultet elektrotehnike i računarstva

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*-u; 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

Fakultet elektrotehnike i računarstva

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
                                            // slucajnim 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);
}
```

Fakultet elektrotehnike i računarstva

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.

Fakultet elektrotehnike i računarstva

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);
}
```

Fakultet elektrotehnike i računarstva

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));
    }
}
```

Fakultet elektrotehnike i računarstva

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*()

Fakultet elektrotehnike i računarstva

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

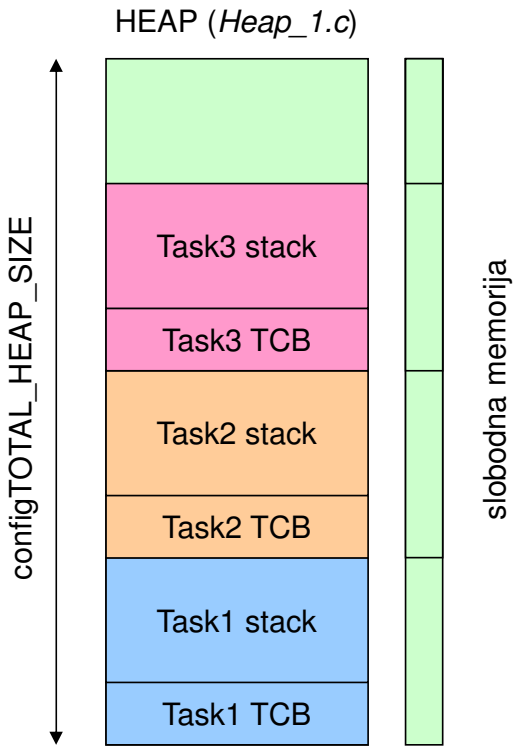
Fakultet elektrotehnike i računarstva

Algoritmi dinamičke alokacije memorije

- *Heap_1.c*
  - najjednostavniji algoritam izravne dodjele blokova bez mogućnosti oslobađanja 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 oslobađati
  - kompaktan kôd, nema problema s determinizmom izvođenja programa, fragmentacijom memorije i sl.

Fakultet elektrotehnike i računarstva

Primjer



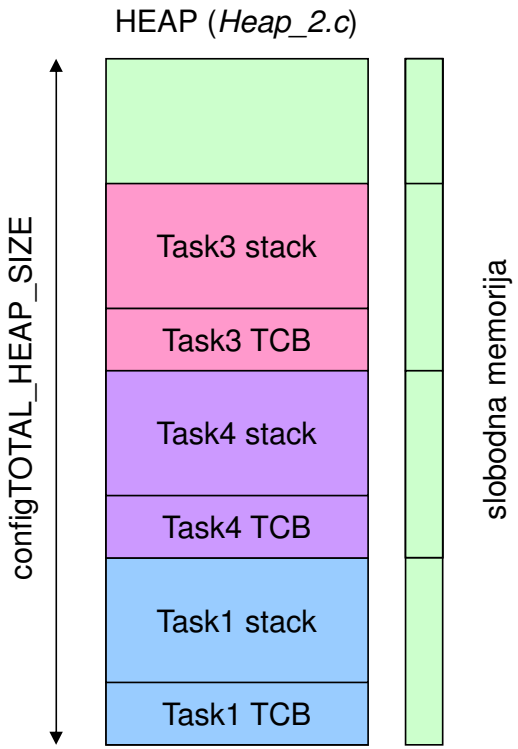
Fakultet elektrotehnike i računarstva

Algoritmi dinamičke alokacije memorije

- *Heap_2.c*
  - koristi tzv. "*best-fit algoritam*" dinamičke alokacije memorije, a omogućuje i oslobađanje 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!),
    - oslobađanje 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!

Fakultet elektrotehnike i računarstva

Primjer



Fakultet elektrotehnike i računarstva

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 inicijalizacije 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

Fakultet elektrotehnike i računarstva

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

Fakultet elektrotehnike i računarstva

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();
        }
}
```

Fakultet elektrotehnike i računarstva

Literatura

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