

Област вежби: *Multitasking*

ВИШЕПРОЦЕСНО ОКРУЖЕЊЕ НА АРМ АРХИТЕКТУРИ

Предуслови:

- RPI2 рачунар
- Микро СД картица са минималном величином 1GB и читач за СД картице
- Адаптер за серијску комуникацију (УАРТ)
- Преводиоц *GCC* за АРМ процесоре (*gcc-arm-none-eabi*) освежен на верзију 6.1 или новију,
- Познавање језика Це и материјала из вежби „УВОД У КОНКУРЕНТНО ПРОГРАМИРАЊЕ“ и „СИНХРОНИЗАЦИЈА И СИГНАЛИЗАЦИЈА ПРОГРАМСКИХ НИТИ“

Увод

Циљ ове вежбе јесте упознавање са једним примером вишепроцесног програмског окружења. Окружење које се користи у вежби, развијено је на одсеку за рачунарску технику и рачунарске комуникације у Новом Саду, иницијално намењено извршавању над оперативним системом МС-ДОС, у заштићеном режиму микропроцесора Интел 80x86. Накнадно је решење прилагођено извршавању на архитектури ARMv7 без додатног оперативног система. Целокупно решење је развијено употребом програмског језика Це. Детаљан опис реализације овог вишепроцесног окружења налази се у ФТН уџбенику бр. 299: „Системска програмска подршка у реалном времену 2“, у одељку 1.5.

Током ове вежбе кроз практичне примере ћете се упознати са основни, концептима и апликативном спрегом овог вишепроцесног окружења.

Припрема радног окружења

За разлику од досадашњих врежби, на RPI рачунару не постоји инсталиран оперативни систем. RPI рачунар користимо у тзв. огољеном (енг. *bare bone*) режиму рада. Због тога превођење кода потребно је извршити на рачунару, а тек потом прекопирати датотеку са машинксим кодом у меморију RPI рачунара.

Да би се омогућило превођење кода писаног у програмским језицима Ц и Ц++ неопходно је инсталирати скуп алата за превођење кода за АРМ процесоре. Алати се могу инсталирати позивом следеће наредбе из Линукс терминала:

```
sudo apt-get install gcc-arm-none-eabi
```

Уколико користите оперативни систем Виндоуз, могуће је преузети званични скуп алата на адреси: <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads>

За превођење вишепроцесног окружења за АРМ процесоре, као и свих примера потребно је користити скрипту *build.sh*. Као параметре скрипти је потребно проследити жељену циљну архитектуру (rpi0, rpi1, rpi1bp, rpi2, rpi3, rpi4 или rpi4b) и који пример желите да преведете (v8_0, v8_1a, v8_1b, v8_2a, v8_2b, v8_3 или v8_4).

Пример коришћења: *build.sh rpi2 v8_1a*

Након успешног превођења у директоријуму *out* направљена је датотека *kernel.img*. Ова датотека садржи преведен код вишепроцесног окружења укључујући наведени пример. Како бисте покренули добијени код потребно је прекопирати ову датотеку у претходно припремљену меморијску картицу.

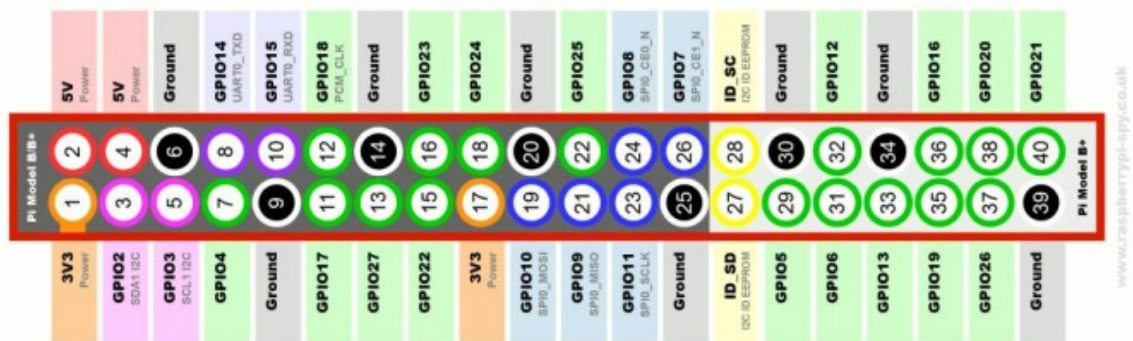
Картицу је пре коришћења потребно форматирати у формату FAT32 и на њу прекопирати садржај директоријума *bootfiles*. Ове датотеке садрже почетни код за RPI који врши почетно пуњење и иницијализацију уређаја.

Након копирања датотеке *kernel.img* потребно је убацити картицу у RPI и укључити га. Уколико је све исправно, зелена лампица на RPI ће бити уагешан током иницијализације система (неколико секунди), након чега ће се упалити непосредно пре позива функције *kernel_main*.

Како би се омогућила спрега са корисником током извршавања програма, вишепроцесно окружење је проширено руковаоцем за серијску комуникацију кроз пролаз UART. За потребе читања и писања података кроз серијски пролаз реализоване су функције:

```
void print_str(char * s); - шаље стринг кроз серијски пролаз  
void print_num(uint32_t n); - претвара број у стринг и шаље кроз  
серијски пролаз  
void print_char(char c); - шаље прослеђени карактер кроз серијски пролаз  
char getch(); - преузима карактер са серијског пролаза
```

Да би се омогућила комуникација са рачунаром неопходно је повезати рачунар са RPI користећи адаптер за серијску комуникацију. Серијски конектор Raspberry Pi рачунара се налази на 40-пинском конектору, као што је приказано на слици испод.



Док је плоча искључена, прикључите жице адаптера на приказани конектор на следећи начин: GND (црна) прикључите на пин означен са GND (пин 6), а RX (бела) и TX (зелена) жице на пинове TX и RX на плочи (пинови 8 и 10). Увек водите рачуна да TX жицу кабла (адаптера) спојите на RX пин плоче и обрнуто, који год кабел или плочу користите.



Када је „USB to Serial“ адаптер повезан на рачунар, нови серијски порт би требало да се појави као `/dev/ttyUSB0`. Да бисте комуницирали са плочом кроз серијски порт, инсталирајте најпре програм за серијску комуникацију попут `picocom`:

```
sudo apt-get install picocom
```

Ако покренете команду `ls -l /dev/ttyUSB0`, можете видети да само `root` и корисници који припадају `dialout` групи имају права приступа датотеци ради читања и писања. Стога, треба додати вашег корисника у `dialout` групу:

```
sudo adduser $USER dialout
```

Сада је још потребно да урадите `log out` и `log in` поново да бисте омогућили новој групи да буде свугде видљива.

Сада можете покренути:

```
picocom -b 115200 /dev/ttyUSB0,
```

да бисте започели серијску комуникацију на `/dev/ttyUSB0`, са параметром `baudrate` постављеним на 115200. Уколико желите да напустите `picocom`, притисните `[Ctrl][a]` праћено са `[Ctrl][x]`.

Вишепроцесно окружење

Иако је иницијално пројектовано са циљем подршке апликацијама за рад у реалном времену, ово окружење је реализовано као вишепроцесни систем опште намене.

Основне карактеристике овог система су:

- могућност динамичког формирање програмских процеса,
- могућност дефинисања модела по коме ће се вршити смена процеса: са истискивањем, без истискивања или комбиновани режим
- програмско дефинисање алгорита распоређивања процеса: редом у круг (енгл. *round robin*) на истом нивоу приоритета и/или побуђен догађајима (енгл. *event driven*), по приоритету догађаја,
- могућност издавања захтева за смену текућег процеса непосредно из прекидне рутине,
- брза смена активних процеса
- могућност одлагања извршења процеса назадати временски период,
- синхронизација процеса (семафорске примитиве, критичне секције, мутекси)
- могућност постављања и промене приоритета процеса)

Процеси се дефинишу као функције у програмском језику Це. Сваки процес описан је контролном табелом процеса (структура *ISA*). Ова структура декларисана је у датотеци *types-mt.h*. Контролне табеле свих процеси који постоје у окружењу увезане су у Табелу логичких ресурса. Индекс контролне табеле процеса у овој табели се назива број логичког ресурса (*LRN*). Он је уједно и јединствени идентификацион број сваког процеса. Испод је дат преглед поља *ISA* структуре:

Назив поља	Значење
I_REGISTRI	Копија регистара микропроцесора.
I_LRN	Број логичког ресурса.
I_PARENT	Број логичког ресурса претка.
I_STANJE	Стање процеса.
I_UZROK	Узрок последње промене стања.
I_VREME	Преостало време на које је процес временски одложен.
I_SWITCH	Величина временског одсечка или NO_TIME ако се могућност не користи.
I_PRIORITY	Приоритет процеса.
stack_mem	Показивач на меморијску зону, која се користи као стек процеса.
stack_len	Дужина меморијске зоне за стек.
next_ready	Ако је процес приправан ово је показивач на следећи у листи приправних.
next_tim_susp	Ако је процес временски одложен ово је показивач на следећи у листи временски одложених процеса.
next_sem_susp	Ако је процес блокиран на семафору ово је показивач на следећи листи временски одложених процеса.

mails	Поштанско сандуче процеса.
start_time	Време првог покретања процеса (у откуцајима РТЦ).
total_time_low	Укупно време извршавања процеса (unsigned long мање тежине).
total_time_hi	Укупно време извршавања процеса (unsigned long веће тежине).

Из претходне табеле може се видети да се сваки процес налази у одређеном стању. Могућа стања процеса су:

Назив стања	Значење
TASK_NOT_CREATED	Још увек није завршено прављење задатка.
TASK_BACKGND	Задатак најнижег приоритета (чека да сви остали заврше).
TASK_READY	Приправан (у реду процеса који чекају на извршавање).
TASK_WAIT_SUSP	Чека на блокирање.
TASK_SUSPENDED	Блокиран.
TASK_TERMINATED	Неактиван / завршен.

Када процес из листе приправних започне да се извршава, њега називамо ТЕКУЋИ процес. Поље стања (у структури ISA) текућег процеса се не ажурира. Провера да ли је процес текући може се извршити провером показивача на контролну табелу текућег процеса: *running*. За процес који је блокиран може се проверити узрок блокирања (семафор, временски одложен и сл.).

Три процеса која увек постоје у систему су основни процес, позадински процес и помоћни системски процес. Основни процес извршава функцију *main*, и он се увек формира одмах по иницијализацији окружења. Овај процес је подразумеваног приоритета. Позадински процес (*idle*) се такође покреће одмах након иницијализације окружења. Ово је процес најнижег приоритета и извршава се само онда када су сви остали процеси неактивни. Помоћни системски процес омогућава укидање осталих процеса (*sysMaintenance*).

Почетна тачка вишепроцесног окружења је функција *kernel_main*. Задатак ове функције је:

1. Иницијализација серијске везе,
2. Иницијализација хардверских прекида,
3. Иницијализација часовника који се користи за смену процеса у случају режима рада са истискивањем,
4. Постављање интервала часовника,
5. Иницијализација контролне јединице за динамичко заузимање меморије,
6. Иницијализација вишепроцесног окружења позивом функције *initmt*. Као параметар функцији се прослеђује режим рада: *TIME_SLICE*, *ROUND_ROBIN* или *TIME_AND_ROUND*. Ова функција је задужена за прављење позадинског и помоћног системског процеса,

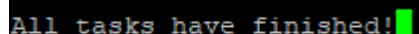
7. Омогућавање прекида, и
8. Прављење основног процеса (*main*) и његово покретање.

Додатно, за потребе ових вежби направљен је још један процес који служи да пријави када су сви остали кориснички процеси (укључујући и основни процес) завршени (*logger*).

Први пример – празна функција *main*

Како бисте испробали да ли је окружење за превођење и покретање подешено исправно, односно да ли сте на исправан начин повезали RPI и рачунар, приложен је пример *v8_0*. Овај пример садржи празну функцију *main*. Другим речима основни задатак се завршава одмах по прављењу.

Преведите и покрените пример. Очекивани излаз на приказу примљених порука путем серијске везе је испис процеса *logger*:



```
All tasks have finished!
```

Стварање и покретање процеса

Стварање новог процеса врши се позивом функције *create_task*. Ова функција прави контролну табелу процеса за нови процес, врши иницијализацију табеле и заузима меморију за стек новонаправљеног процеса. Направљени процес није активан (неће се извршавати пре позива функције за покретање извршавања процеса).

```
int create_task ( void(*fun)(void*), int steklen, void* params, int tasktime);
```

Повратна вредност функције је логички број процеса (позитиван цео број) или код грешке уколико је дошло до грешке (негативан број).

Функција	Опис
<i>create task</i>	Функција за формирање процеса.
Параметри	Опис
<i>fun</i>	Адреса функције коју процес извршава. Функција се декларише на следећи начин: void* TaskRoutine (void* param);
<i>steklen</i>	Величина стека резервисаног за процес (у бајтима).
<i>params</i>	Показивач који ће бити прослеђен процесу приликом покретања. Преко њега процес може приступити подацима које је програмер наменио.
<i>tasktime</i>	Величина временског одсечка у милисекундама. Користи се за постизање различите расподеле процесорског времена међу процесима који су истог приоритета (<i>NO_TIME</i> означава да се користи подразумевана вредност)
Повратна вредност	Опис
<i>int</i>	Логички број процеса, уколико је процес успешно направљен. У супртном код грешке која је спречила стварање процеса.

Након успешног прављења новог процеса, потребно је извршити његово покретање. Покретање процеса се врши позивом функције:

```
int sptsk (int task_handler, int priority);
```

Ова функција додељује приоритет процесу, врши иницијализацију процеса и додаје га у ред процеса спремних за извршавање. Пошто процес може бити највишег приоритета од свих текућих процеса, врши се одмах експлицитан позив распоређивача (не чека се на временски прекид).

Функција	Опис
<i>sptsk</i>	Функција за активирање неактивног процеса.
Параметри	Опис
<i>task_handler</i>	Логички број процеса
<i>priority</i>	Приоритет процеса (1 – највиши приоритет, 1000 – најнижи приоритет, <i>PRDEFAULT</i> – подразумевана вредност)
Повратна вредност	Опис
<i>int</i>	0 уколико је операција успешна. У супртном код грешке.

Након завршетка функције која се извршава у оквиру процеса, процес прелази у стање неактиван. Да би се процес у потпуности уклонио из табеле процеса, потребно је позвати функцију за укидање процеса:

```
int kill_task (int task_handler);
```

Било који процес може да укине себе, или било који други неактиван процес. Уколико процес покуша да укине неки други процес који није неактиван, функција ће вратити грешку.

Функција	Опис
<i>kill_task</i>	Укида тренутно извршавани, или неактиван процес чији логички број је прослеђен.
Параметри	Опис
<i>task_handler</i>	Логички број процеса.
Повратна вредност	Опис
<i>int</i>	0 уколико је операција успешна. У супртном код грешке.

Пример 1 (изворни код се налази у датотеци vezba v8 1a.c)

Овај пример представља програм са три процеса који исписују свој идентификатор на исти начин као што је то рађено са нитима у вежби 2. За испис вредности користи се УАРТ пролаз. Процес који извршава функцију *main* се након покретања 3 процеса потомка одмах завршава. Потребно је приметити да за разлику од рада са програмским нитима под оперативним системом Linux, иако се задатак *main* завршио, преостали задаци настављају извршавање. Након завршетка свих процеса систем прелази на извршавање позадинског процеса најнижег приоритета (*logger*).

```
#include "tasks.h"
#include "kernel.h"
#include "stdlib.h"
#include "types-mt.h"
#include "rpi-aux.h"

void print1(void* params){
    int i;

    for(i = 0; i < 1000; i++){
        print_str("1");
    }
}

void print2(void * params){
    int i;

    for(i = 0; i < 1000; i++){
        print_str("2");
    }
}

void print3(void * params){
    int i;

    for(i = 0; i < 1000; i++){
        print_str("3");
    }
}

void main(void * params){
```



```
int hPrint1, hPrint2, hPrint3;

hPrint1 = create_task(print1, STACK_SIZE, NULL, 0);
hPrint2 = create_task(print2, STACK_SIZE, NULL, 0);
hPrint3 = create_task(print3, STACK_SIZE, NULL, 0);

sptsk(hPrint1, PRDEFAULT);
sptsk(hPrint2, PRDEFAULT);
sptsk(hPrint3, PRDEFAULT);
}
```

У претходном примеру сви процеси су једнаког приоритета. Уколико променимо приоритет процеса 3 тако да је виши од приоритета друга два процеса, можете примерити да ће се по покретњу процеса 3 непрекидно извршавати овај процес све док не заврши предвиђени посао. Тек након тога процеси 1 и 2 настављају са извршавањем (пример v8_1b).

Објекат искључивог приступа

За потребе синхронизације између задатака и заштиту критичних секција кода окружење нуди објекте искључивог приступа - мутексе (енг. mutex = **mutual exclusion**). Објекат искључивог приступа имплементиран је као структура MUTEX која садржи следећа поља:

Назив поља	Значење
<i>S_COUNT</i>	Текућа вредност мутекса (има вредност 1 уколико је откључан)
<i>S_OWNCNT</i>	Број угњеждених секција.
<i>S_OWNER</i>	Логички број процеса који је закључао мутекс.
<i>S_QH</i>	Показивач на почетак листе процеса блокираних на мутексу.
<i>S_QTP</i>	Показивач на крај листе процеса блокираних на мутексу.

Мутекси су имплементирани као бинарни семафори са подршком за угњеждавање критичних секција. Процес који заузима мутекс постаје његов „власник“. Након тога исти процес може више пута извршити операцију закључавања. Други процеси уколико покушају да заузму мутекс који је већ заузет, биће блокирани. Свака операција заузимања мора бити праћена операцијом ослобађања од стране истог процеса да би мутекс прешао у стање „ослобођен“, односно како би неки други процес могао да изврши заузимање мутекса.

Функције које се користе у раду са објектом искључивог приступа су:

```
void dfmtx ( MTXPTR mtx )
```

Функција	Опис
<i>dfmtx</i>	Иницијализује објекат искључивог приступа.
Параметри	Опис
<i>mtx</i>	Показивач на објекат искључивог приступа. Функција подразумева да је меморија за тај објекат већ заузета од стране корисника, то јест, да је објекат пре тога дефинисан.

```
int rsvmtx( MTXPTR mtx )
```

Функција	Опис
<i>rsvmtx</i>	Заузимање објекта искључивог приступа.
Параметри	Опис
<i>mtx</i>	Показивач на објекат искључивог приступа. Функција подразумева да је објекат искључивог приступа претходно био иницијализован.
Повратна вредност	Опис
<i>int</i>	Повратна вредност је 1 у случају да је објекат искључивог приступа успешно заузет. У супротном, повратна вредност је 0.

```
int rlsmtx( MTXPTR mtx )
```

Функција	Опис
<i>rlsmtx</i>	Ослобађање објекта искључивог приступа.
Параметри	Опис
<i>mtx</i>	Показивач на објекат искључивог приступа. Функција подразумева да је објекат искључивог приступа претходно био иницијализован.
Повратна вредност	Опис
<i>int</i>	Повратна вредност је 1 у случају да је објекат искључивог приступа успешно ослобођен. У супротном, повратна вредност је 0.

Пример 2 (изворни код се налази у датотеци vezba v8 2a.c)

У овом примеру приказана је употреба мутекса за синхронизацију исписа између више процеса. Програм чине три процеса који исписују три различите реченице путем УАРТ пролаза. Постоје две верзије примера: у првој (v8_2a) се не користе објекти искључивог приступа и исписи реченица ће се преклапати, што је нежељена ситуација. До преклапања долази јер су сви процеси једнаког приоритета, и не зна се у ком тренутку извршавања ће доћи до смене процеса.

Избегавање преклапања реченица је урађено користећи мутекс у примеру v8_2b.

```
#include "tasks.h"
#include "kernel.h"
#include "types-mt.h"
#include "rpi-aux.h"
#include "stdlib.h"
```

```
static MUTEX cs_mutex;

void print1(void * params){
    int i;

    for(i = 0; i < 100; i++){
        rsvmtx(&cs_mutex);

        print_str("Zvezvda");
        print_str(" je");
        print_str(" prvak");
        print_str(" jsveta\n\r");

        rlsmtx(&cs_mutex);
    }
}

void print2(void * params){
    int i;

    for(i = 0; i < 100; i++){
        rsvmtx(&cs_mutex);

        print_str("Priprema");
        print_str(" vezbe");
        print_str(" iz");
        print_str(" systemske");
        print_str(" programske");
        print_str(" podrske\n\r");

        rlsmtx(&cs_mutex);
    }
}

void print3(void * params){
    int i;

    for(i = 0; i < 100; i++){
        rsvmtx(&cs_mutex);

        print_str("Danas");
        print_str(" je");
        print_str(" lep");
        print_str(" i");
        print_str(" suncan");
        print_str(" dan\n\r");

        rlsmtx(&cs_mutex);
    }
}

void main(void * params){

    int hPrint1, hPrint2, hPrint3;

    dfmtx(&cs_mutex);
```

```
hPrint1 = create_task(print1, STACK_SIZE, NULL, 0);  
hPrint2 = create_task(print2, STACK_SIZE, NULL, 0);  
hPrint3 = create_task(print3, STACK_SIZE, NULL, 0);  
  
sptsk(hPrint1, PRDEFAULT);  
sptsk(hPrint2, PRDEFAULT);  
sptsk(hPrint3, PRDEFAULT);  
}
```

Семафори

Поред мутекса, вишепроцесно окружење као алат за синхронизацију између процеса нуди и семафор. Семафор је имплементиран као структура SEM у програмском језику Це, на следећи начин:

Назив поља	Значење
<i>S_COUNT</i>	Текућа вредност семафора (бројач слободних ресурса).
<i>S_QH</i>	Показивач на почетак листе процеса блокираних на семафору.
<i>S_QTP</i>	Показивач на крај листе процеса блокираних на семафору.

Семафори функционишу на сличан начин као и код библиотеке POSIX. Дефинисане су недељиве операције УВЕЋАЈ и УМАЊИ над семафором. Када је вредност бројача *S_COUNT* већа од 0 семафор је сигнализирани. У супротном семафор није сигнализирани. Процес који покуша да умањи семафор чија је вредност 0 биће блокиран. Сви процеси који су блокирани и чекају да семафор буде сигнализирани смештају се у листу процеса за дати семафор.

Функције које се користе у раду са семафорима су:

```
void dfsm (SEMPTR sem, int count)
```

Функција	Опис
<i>dfsm</i>	Иницијализује семафор, тј. поставља почетну вредност семафора.
Параметри	Опис
<i>sem</i>	Показивач на структуру семафора. Функција подразумева да је меморија за тај објекат већ заузета од стране корисника, тј. да је објекат пре тога дефинисан.
<i>count</i>	Почетна вредност семафора.

```
int rsvsm (SEMPTR sem)
```

Функција	Опис
<i>rsvsm</i>	Умањује вредност семафора за 1. Уколико семафор није сигнализирани текући процес прелази у стање блокиран и додаје се у листу блокираних процеса за дати семафор.
Параметри	Опис
<i>sem</i>	Показивач на структуру семафора. Функција подразумева да је семафор претходно био иницијализован.
Повратна вредност	Опис
<i>int</i>	Повратна вредност је 1 у случају да је операција извршена успешно. У случају грешке, повратна вредност је 0.

```
int rlsm (SEMPTR sem)
```

Функција	Опис
<i>rlsm</i>	Увећава вредност семафора за 1. Поставља први процес из листе блокираних (уколико листа није празна) у стање приправан и додаје у листу активних процеса.
Параметри	Опис
<i>sem</i>	Показивач на структуру семафора. Функција подразумева да је семафор претходно био иницијализован.
Повратна вредност	Опис
<i>int</i>	Повратна вредност је 1 у случају да је операција извршена успешно. У случају грешке, повратна вредност је 0.

```
int tryrsvsm (SEMPTR sem)
```

Функција	Опис
<i>rsvsm</i>	Умањује вредност семафора за 1 уколико је семафор сигнализирани и враћа вредност 1. Уколико семафор није сигнализирани враћа 0.
Параметри	Опис
<i>sem</i>	Показивач на структуру семафора.
Повратна вредност	Опис
<i>int</i>	Повратна вредност је 1 у случају да је семафор успешно умањен. У супротном 0.

Пример 3 (изворни код се налази у датотеци vezba v8 3.c)

Пример илуструје рад семафора. Процес *main* покреће нови процес који чека на семафору. У оквиру направљеног процеса, сваки пут када се семафор сигнализира повећава се вредност бројача за 1 и исписује. Процес *main* након покретања направљеног процеса преузима карактере са серијског пролаза, и за сваки преузети карактер изврши операцију УВЕЋАЈ над семафором.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

static sem_t semaphore;
```

```
#include "tasks.h"
#include "kernel.h"
#include "types-mt.h"
#include "rpi-aux.h"
#include "stdlib.h"

static SEM semaphore;

static int counter;

void counter_thread(void * params){
    while(1){
        rsvsm(&semaphore);

        counter++;
        print_str("counter = ");
        print_num(counter);
        print_str("\n\r");
    }
}

void main(void * params){

    int hThread;

    dfsm(&semaphore, 0);

    hThread = create_task(counter_thread, STACK_SIZE, NULL, 0);

    sptsk(hThread, PRDEFAULT);

    while(1){
        if(getch() == 'q'){
            break;
        }
        rlsm(&semaphore);
    }

}
```

Пример 4 (изворни код се налази у датотеци vezba v8 4.c)

У овом примеру приказана је имплементација шаблона „произвођач – потрошач“ (претходно обрађен у вежби 3) у оквиру вишепроцесног окружења, користећи механизме које оно нуди. Реализација се састоји из 3 процеса: произвођач, радник и потрошач. За комуникацију између процеса користе се кружни бафери, тачније улази бафер за комуникацију између произвођача и радника и излазни бафер за комуникацију између радника и потрошача.

Задатак произвођача је да кад год има доступног места у улазном кружном баферу, преузме карактер са серијског пролаза, провери да ли је притиснути карактер слово ‘q’. Уколико јесте, сигнализира се семафор који означава завршетак програма (*semFinishSignal*). Уколико није, преузети карактер се уписује у кружни

бафер, при чему је операција додавања заштићена мутексом. Након тога се сигнализира семафор који представља бројач карактера у првом кружном баферу.

```
/* Funkcija procesa proizvođača */
void producer(void * param){
    char c;

    while(1){
        if(tryrsvsm(&semFinishSignal) == TRUE){
            break;
        }

        if(tryrsvsm(&semInEmpty) == TRUE){

            /* Funkcija za unos karaktera sa tastature. */
            c = getch();

            /* Ukoliko je unet karakter q signalizira se
            svim procesima zavrsetak rada. */
            if(c == 'q'){
                /* Zaustavljanje procesa; Semafor se oslobadja 3 puta
                da bi signaliziralo svim procesima.*/
                rlsn(&semFinishSignal);
                rlsn(&semFinishSignal);
                rlsn(&semFinishSignal);
            }

            /* Pristup kružnom baferu. */
            rsvmtx(&inputBufferAccess);
            ringBufPutChar(&inputBuffer, c);
            rlsmtx(&inputBufferAccess);

            rlsn(&semInFull);
        }
    }
}
```

Процес радник проверава да ли у улазном баферу постоје доступни карактери. Уколико постоје, карактер се преузима из бафера и сигнализира се семафор који представља бројач слободних места у улазном баферу. Уколико је карактер мало слово, врши се конверзија у велико слово. И потом се вредност уписује у излазни бафер. Уколико је излазни бафер препуњен, чека се док се не појави слободно место. Након уписа се сигнализира семафор који представља бројач елемената у излазном баферу.

```
/* Funkcija procesa radnika */
void converter(void){
    char c;

    while(1){
        if(tryrsvsm(&semFinishSignal) == TRUE){
            break;
        }

        if(tryrsvsm(&semInFull) == TRUE){
```

```
    rsvmtx(&inputBufferAccess);
    c = ringBufGetChar(&inputBuffer);
    rlsmtx(&inputBufferAccess);

    if(c >= 'a' && c <= 'z'){
        c -= LOWER_TO_UPPER_CASE;
    }

    /* Simulacija trajanja posla */
    simulate_processing();

    while(1){
        if(tryrsvsm(&semOutEmpty) == TRUE){
            rsvmtx(&outputBufferAccess);
            ringBufPutChar(&outputBuffer, c);
            rlsmtx(&outputBufferAccess);
            rlsm(&semOutFull);
            break;
        }
    }
    rlsm(&semInEmpty);
}
}
```

Процес потрошач чека на појаву карактера у излазном баферу, односно на семафор који представља бројач елемената за излазни кружни бафер. Када год у баферу постоји доступан карактер, он се преузима и исписује кроз серијски пролаз. Након преузимања карактера, сигнализира се семафор који означава број слободних места у излазном баферу.

```
/* Funkcija procesa potrosaca */
void consumer(void){
    char c;

    print_str("Znakovi preuzeti iz kruznog bafera:\n\r");

    while(1){
        if(tryrsvsm(&semFinishSignal) == TRUE){
            break;
        }

        if(tryrsvsm(&semOutFull) == TRUE){
            /* Pristup kruznom baferu. */
            rsvmtx(&outputBufferAccess);
            c = ringBufGetChar(&outputBuffer);
            rlsmtx(&outputBufferAccess);

            /* Ispis na konzolu. */
            print_char(c);

            rlsm(&semOutEmpty);
        }
    }
}
```


Главни процес, (*main*) врши иницијализацију семафора и мутекса. Потом прави 3 процеса и покреће их са подразумеваним нивоом приоритета. Након покретања процеса, врши се промена приоритета главног процеса, тако да буде нижег приоритета од три направљена процеса. Овим се обезбеђује да ће процес сачекати да поменута 3 процеса пређу у стање неактивни, пре него што изврши уништавање процеса.

Функције за добављање и постављање приоритета текућег процеса су:

```
int levelget ( void );
```

Функција	Опис
<i>levelget</i>	Враћа приоритет текућег процеса.
Повратна вредност	Опис
<i>int</i>	Цео број у опсегу 1-1000 који представља приоритет текућег процеса. -1 у случају грешке.

```
int levelchange ( int priority);
```

Функција	Опис
<i>levelchange</i>	Враћа приоритет текућег процеса.
Параметри	Опис
<i>priority</i>	Жељени приоритет (1-1000).
Повратна вредност	Опис
<i>int</i>	Уколико је промена успешна, стара вредност приоритета процеса. -1 у случају грешке.

```
/* Glavni proces koji formira preostale procese (proizvodjac, potrosac i  
radnik) i ceka njihovo gasenje. */  
void main(void * params){  
  
    /* Identifikatori procesa. */  
    int hProducer;  
    int hConverter;  
    int hConsumer;  
  
    /* Formiranje semEmpty, semFull i semFinishSignal semafora. */  
    dfsm(&semInEmpty, RING_SIZE);  
    dfsm(&semInFull, 0);  
    dfsm(&semOutEmpty, RING_SIZE);  
    dfsm(&semOutFull, 0);  
    dfsm(&semFinishSignal, 0);  
  
    /* Inicijalizacija objekta iskljucivog pristupa. */  
    dfmtx(&inputBufferAccess);  
    dfmtx(&outputBufferAccess);  
  
    /* Formiranje procesa: proizvodjac, potrosac i radnik */  
    hProducer = create_task(producer, STACK_SIZE, NULL, 0);  
    hConsumer = create_task(consumer, STACK_SIZE, NULL, 0);  
    hConverter = create_task(converter, STACK_SIZE, NULL, 0);  
  
    sptsk(hProducer, PRDEFAULT);  
    sptsk(hConsumer, PRDEFAULT);
```

```
sptsk(hConverter, PRDEFAULT);  
  
levelchange(PRDEFAULT+1);  
  
print_str("Oslobadjanje memorije i uklanjanje zadataka\n\r");  
  
kill_task(hProducer);  
kill_task(hConsumer);  
kill_task(hConverter);  
  
}
```