SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 5909

# Operacijski sustav za rad u stvarnom vremenu

Luka Kelava

Zagreb, lipanj 2019.

*Umjesto ove stranice umetnite izvornik Vašeg rada.*
*Da bi ste uklonili ovu stranicu obrišite naredbu* `\izvornik`.

*Ovaj završni rad posvećujem svojoj obitelji i prijateljima*

# SADRŽAJ

# POPIS SLIKA

# 1. Uvod

Cilj ovoga rada bit će dati opći uvid u arhitekturu operacijskih sustava, proširiti pregled sustava u stvarnome vremenu i načina implementacije ključnih dijelova te naposljetku demonstrirati jednu takvu implementaciju s prikazom dijelova koda i raščlambom mehanizama samoga sustava. Osim toga, u ovom radu, prikazat će se i dijelovi arhitekture ARM procesora bitni za razumijevanje principa rada cjelokupnoga sustava.

Operacijski sustavi nastali su kao logičan korak između sklopovlja i željene funkcionalnosti, nužna apstrakcija koja će korisniku olakšati služenje računalom, automatizirati procese i ubrzati određene poslove i raspodijeliti računalne aktivnosti tako da osiguraju dobru iskorištenost raspoloživih resursa i da uspostave prikladno okruženje za pripremu i izvođenje programa. U rane dane računarstva, većina je ugradbenih sustava bila dizajnirana i namijenjena za usko specifična područja. Ugradbeni sustav uglavnom se sastoji, kao i nekad, od mikrokontrolera (u posljednje vrijeme uglavnom po nacrtima od većih proizvođača – Atmega, ARM, Texas Instruments...), ulazno/izlaznih jedinica, sustava napajanja i generatora signala. U davna vremena 80-ih počéla funkcioniranja programa u takvim sustavima bila su rudimentarna, većina programa sastojala se od velikih petlji koje su provjeravale neko stanje na vanjskim jedinica i, ovisno o promjeni stanja, izvršavale neke operacije. Kako je složenost ugradbenih sustava rasla, bilo je potrebno utećі se nekim drugim obrascima u dizajnu. Na to mjesto dolaze operacijski sustavi. Oni će nam dati privid istovremenosti, a ujedno će postati i platforma za razvoj tržišne utakmice koja će izroditi neke od proizvoda bez kojih bi naš današnji život uvelike bio nezamisliv. Neki od tih sustava su i oni najpoznatiji – Android, distribucije Linuxa, Windows Mac OS i mnogi drugi za koje ne znamo, ali su prisutni na svakome koraku – od našega automobila, televizora, zrakoplova pa sve do najmanjih kućnih uređaja poput perilica za rublje. [9]

# 2. Pregled ciljane arhitekture (ARM)

## 2.1. O ARM-u

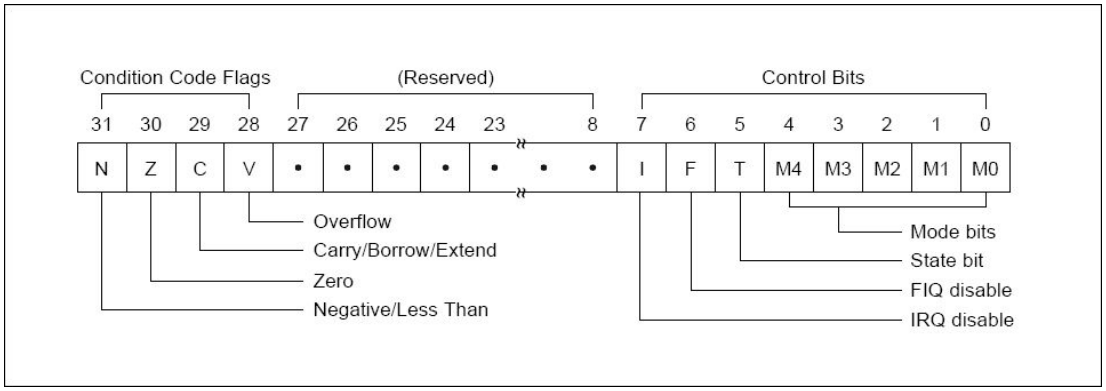
ARM pripada RISC procesorskoj familiji razvijanoj ciljano za ugradbene okoline. Zbog njihove male veličine i male potrošnje ARM je postao najraširenija porodica procesora u mobilnoj industriji i ugradbenim (eng. embedded) sustavima. U velikoj mjeri ARM danas gotovo pa da predstavlja sinonim za cjelokupnu mobilnu industriju, a programiranje za ARM procesore programiranje ugradbenih sustava. Zbog velikoga spektra upotrebe i već povelike povijesti potrebno klasificirati ARM-ove procesore. Ovisno o sposobnostima i ciljanoj primjeni, ARM Cortex može se podijeliti u tri kategorije:

- Cortex-M serija
  - MCU jedinice
  - SoC sustavi
- Cortex-R serija
  - Za ugradbene sustave s obradom signala u stvarnome vremenu
  - Za kontrolne primjene
- Cortex-A serija
  - sustavi opće namjene (ugradbeni sustavi s operacijskim sustavom)

ARM Cortex-A najmoćnija je ARM-ov 32-bitna i 64-bitna porodica procesora. 32-bitne ARM Cortex-A, osim Cortex-A32, implementiraju ARMv7-A profil ARMv7 arhitekture koji je nuždan za pokretanje mnogih modernih operacijskih sustava jer uključuje MMU (eng. memory management unit). 64-bitne ARM Cortex-A jezgre implementiraju ARMv8-A profil ARMv8 arhitekture. Također valja napomenuti da Raspberry PI 3 sadržava Broadcomov čip BCM 2837 koji sadržava quad-core ARM Cortex A53 (ARMv8) procesoru.[1]

## 2.2. Načini rada u procesoru ARM

ARM ima 7 načina rada koji su specificirani s 5 bitova u CPSR registru, što je vidljivo u ilustraciji.



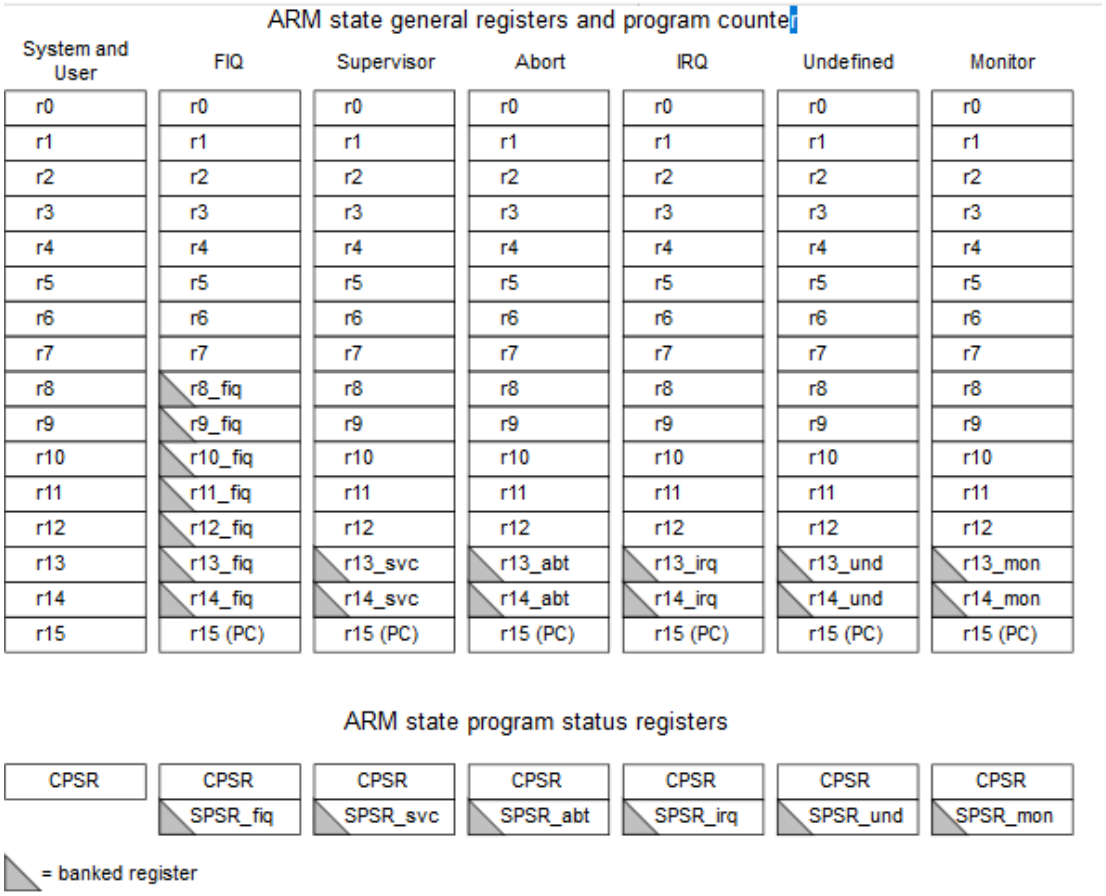

**Slika 2.1:** CPSR registar

Načini rada poredan od korisničkoga prema privilegiranima:

- USR – neprivilegirani korisnički način rada
- SYS – sistemski način rada koji koristi istovjetan set registara kao i USR
- FIQ – brzi prekidi (eng. fast interrupt)
- IRQ – prekidni način rada (eng. interrupt)
- SVC – nadzornički način rada (eng. supervisor), u njega sustav ulazi tijekom reseta
- ABT – način rada prilikom nailaska na iznimku (eng. abort)
- UND – nedefinirana instrukcija (eng. undefined instruction)

## 2.3. Registri u ARM-ovim procesorima

Kao što se već moglo iščitati iz podjele opisa načina rada procesora, ne dijele svi načini rada „iste“ registre. ARM-ov procesor uopće ima 37 registara, od kojih su svi 32-bitni na 32-bitnoj arhitekturi, a 64-bitni na 64-bitnoj. Od ti 37 registara 1 je PC i 1 CPSR (gore prikazan) i ti registri su zajednički svim načinima rada. Uz njih imamo i 5 SPSR registara (spremljeno stanje CPSR-a) i 30 procesora opće namjene. Razdioba registara po načinima rada, vidljiva je iz ilustracije 2.1.




**Slika 2.2:** Registri

## 2.4. Pregled naredbi u ARM-u

### 2.4.1. Cjevovod

Izvršavanje naredaba u ARM-u funkcionira po principu cjevovoda (eng. pipelina), što omogućuje da se više operacija izvodi istovremeno. Postoje tri faze izvođenja:

1. FETCH (PC) – PC pokazuje na trenutnu instrukciju koju smo dohvatili
2. DECODE (PC - 4) – instrukcija se dekodira
3. EXECUTE (PC - 8) – instrukcija se izvodi

### 2.4.2. Zastavice

- N – negativan rezultat
- Z – rezultat je 0
- V – preljev (eng. overflow)
- C – bit prijenosa (carry bit)

Zastavice se postavljaju tako da se u najviše bitove CPSR-a, ovisno o rezultatu, upiše 1. Zastavice su vidljive u 2.1.

### 2.4.3. Forma instrukcije

Ovdje ćemo sažeto prikazati neke od instrukcijskih forma ARM-a, 1. forma je RRR - koristi se kod aritmetičkih i logičkih operacija.

Na priliku:

```
1 ADD r0, r1, r2 -> r0 = r1 + r2
```

naredbe te forme imaju i sufiks -S na kraju koji signalizira da se mijenjaju vrijednosti zastavica u CPSR registru, pa možemo pisati i:

```
1 ADDS r0, r1, r2 -> r0 = r1 + r2 uz mijenjanje zastavica
```

Zatim imamo instrukcije grananja koje uzimaju kao argument samo adresu skoka (apsolutnu ili relativnu), a to je lokacija koju upisujemo u PC registar.

Skok može biti uvjetan i bezuvjetan, ovisno opet o sufiksu, tj. cond. dijelu. Ako je naredba za skok BL umjesto B, tada se u R14 zapisuje povratna adresa.

primjer:

```
1 BL mojaFunkcija
```

Od bitnijih instrukcija imamo još MOV, Load-Store i STMFD instrukcije

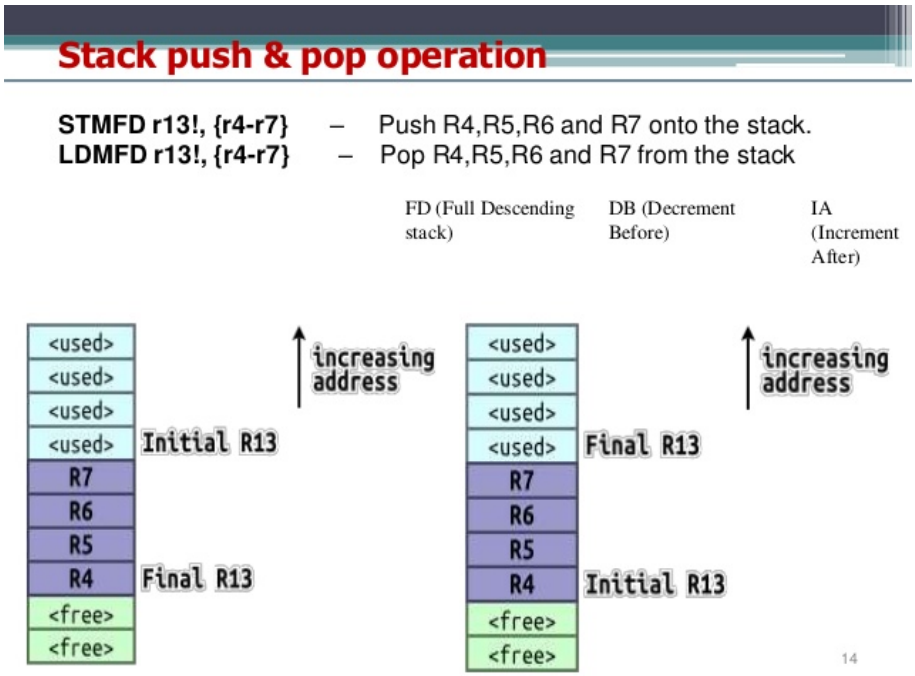
primjeri:

```
MOV r0, #0 -> r0 = 0
LDR r1, =Podatak -> r1 = &Podatak
STR r1, [r2] -> spremamo r1 na lokaciju r2
LDR r2, [r1] -> r2 = *r1
```

Za kraj ovoga potpoglavlja uzet ćemo zdravo za gotovo da stog "raste" prema dolje (kako uostalom uzima i C kompajler) pa ćemo ekskluzivno upotrebljavati naredbe STMFD i LDMFD.

primjer:

```
STMFD sp!, {r0-r12, lr} -> spremanje svih registara
i povratne adrese
LDMFD sp!, {r0-r12, pc} -> vracanje registara i povratak na lr
```




**Slika 2.3:** Stog

### 2.4.4. Primjeri s objašnjenjima

**Poziv funkcije, definicija globalnoga područja i obrada iznimke**

```
.text
.code 32
.global main, interrupt_handler

reset_entry:
    LDR sp, =svc_stack_addr -> postavljanje adrese SVC stoga
    BL move_interrupt_vectors -> skok na funkciju koja
    kopira vektore na adresu 0
    MSR cpsr, #0x92 -> ulazak u IRQ nacin rada
    LDR sp, =irq_stack_addr
    MSR cpsr, #0x13 -> vracanje u SVC


interrupt_handler:
    SUB lr, lr, #4 -> zelimo se vratiti se na LR - 4
    STMFD sp!, {r0-r12, lr} -> spremimo na stog registre
    i adresu povratka (u slucaju rekurzije ili poziva u pozivu)
    BL obrada_prekida
    LDMFD sp!, {r0-r3, r12, pc}^ -> povratak
```

## 2.5. Cross-kompajliranje koda za ARM

U procesu od pisanja do pokretanja neke izvršne datoteke postoji nekoliko koraka. Za početak potrebno je iskompajlirati kod, tj. prethodno proći kroz fazu preprocesora. Nakon toga slijedi povezivanje koda (eng. linking). Iako GCC može i bez naše intervencije povezati kod, potrebno je napisati svoju skriptu za povezivanje da bismo imali potpunu kontrolu.

Prvi korak je instaliranje programske potpore nužne za povezivanje i kompilaciju koga.

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

Ovi paketi dostupni su na KDE Neonu i bilo kojoj drugoj distribuciji temeljenoj na Ubuntu.

Nakon toga, potrebno je napisati skriptu za povezivanje. Za potrebe ovoga primjera, nazvat ćemo je link-example.ld.

```
ENTRY(reset_entry)

SECTIONS
{
/* Ishodiste */
. = 0x10000;
.text : { asm_prog.o *(.text) }
.data : { *(.data) }
.bss : { *(.bss) }
. = ALIGN(8);
. = . + 0x1000;
/* velicina SVC stoga je 4kB */
svc_stack_addr = .;
/* velicina IRQ stoga je 4kB */
. = . + 0x1000;
irq_stack_addr = .;
```

Nekoliko bitnih dijelova gornjega koda:

- .text - mjesto gdje se nalazi izvršni kod
- .data - mjesto gdje se nalaze globalne i statičke varijable koje su inicijalizirane prilikom kompilacije
- .bss mjesto gdje se nalaza neinicijalizirane globalne i statičke varijable
- ENTRY(ENTRY) - definiramo `reset_entry` kao ishodište u našem kodu

Na posljetku, portebno je taj kod asemblirati, prevesti i povezati. To ćemo učiniti s pomoću alata iz gcc-arm-none-eabi paketa.[8]

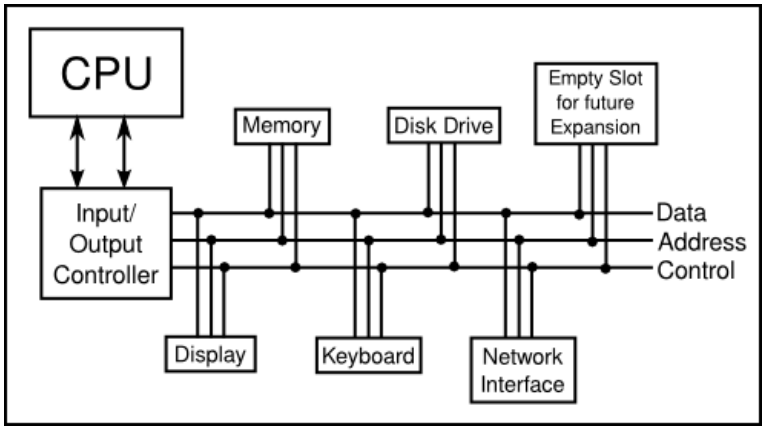
```
arm-none-eabi-gcc -fpic -ffreestanding -c boot.S -o boot.o
arm-none-eabi-gcc -fpic -ffreestanding -std=gnu99 -c kernel.c -o \
kernel.o -O2 -Wall -Wextra
arm-none-eabi-gcc -T linker.ld -o myos.elf -ffreestanding -O2 \
-nostdlib boot.o kernel.o
```

# 3. Operacijski sustav - terminologija i podjele

## 3.1. Način obrade događaja

### 3.1.1. Općenito o događajima

Računalni sustav koji se sastoji od više uređaja obično se organizira spajanjem tih uređaja na jednu ili vise sabirnica. Sabirnica je skup žica koje nose različite vrste informacija između uređaja spojenih na te žice. To mogu biti podatci, adrese i upravljački signali. Adresa usmjerava podatke na određeni uređaj, a možda i na određeni registar ili lokaciju unutar tog uređaja. Upravljački signali mogu se koristiti, na primjer, s pomoću jednog uređaja kako bi upozorili druge da su podatci dostupni za nj na podatkovnoj sabirnici. Prilično jednostavan računalni sustav može biti organiziran na sljedeći način:




**Slika 3.1:** Primjer spajanja sabirnica

Kako CPU zna da su podatci doista tamo? Jedna jednostavna zamisao, ali prilično skupa, jest da CPU neprestano provjerava dolazne podatke. Kad god pronađe podatke, obrađuje ih. Ova metoda se naziva prozivanje, budući da CPU neprestano ispituje ulazne uređaje kako bi provjerio imaju li ulazne podatke koji nam trebaju. Nažalost, iako je prozivanje (eng. polling) vrlo jednostavno, ono je također vrlo neučinkovito. CPU može potrošiti mnogo vremena samo čekajući ulaz.

Kako bi se izbjegla ta neučinkovitost, umjesto prozivanja obično se upotrebljavaju prekidi. Prekid je signal koji drugi uređaj šalje CPU-u. CPU reagira na prekidni signal stavljajući na stranu sve što radi kako bi odgovorio na prekid. Pošto je obradio prekid, vraća se onome što je radio prije nego je došlo do prekida. Na primjer, kada pritisnete tipku na tipkovnici računala, na procesor se šalje prekid tipkovnice. CPU odgovara na taj signal prekidajući ono što radi, čitajući znak koji ste pritisnuli, obrađujući ga, a zatim se vraćajući na zadatak koji je obavio prije nego što ste pritisnuli tipku.

Uređaji poput tipkovnice, miša i mrežnog sučelja mogu proizvesti ulazne podatke koje procesor treba obraditi. Bitno je razlučiti događaje na sinkrone, poput timera, one koje možemo predvidjeti i događaju se u predvidljivim vremenskim okvirima i asinkrone, poput korisničkoga unosa, pritiska tipke miša ili neke slične operacije.

## 3.2. Kategorizacija sustava

### 3.2.1. Općenito

Operacijski sustavi su tijekom svoga razvoja prošli različite faze i zahtjeve, od jednostavnih sustava koji su podržavali jednu procesorsku jezgru s rudimentarnim multitaskingom i bez zaštite memorije do velikih višejezgrenih multithreading sustava s virtualnom memorijom i preemptivnim raspoređivačem (eng. scheduler). Daljnje poglavlja dat će uvid u kompletnu kategorizaciju sustava.

### 3.2.2. Jednoprocesorski sustav

Jednoprocesorski sustav sastoji se samo od jedne jezgre, jednoga CPU-a. Svi procesi izvršavaju se na jednom CPU. Za promjenu trenutnoga procesa, potrebno je izvršiti izmjenu konteksta (eng. context switching) i pokazivač na trenutni proces usmjeriti na željeni proces.

### 3.2.3. Višeprocesorski sustav

Višeprocesorski sustavi sastoje se od više jezgara ili više CPU-a. U tom slučaju procesi se mogu pokretati istovremeno (bez privida kao u jednoprocesorskom sustavu) na više jezgara.

### 3.2.4. Sustav sa predefiniranim statičkim procesima

Statički procesi stvaraju se pri pokretanju sustava. U sustavu ostaju zauvijek, najčešće se radi o jednostavnoj Round Robin rotaciji bez preempcije.

### 3.2.5. Sustav s dinamičkim stvaranjem procesa

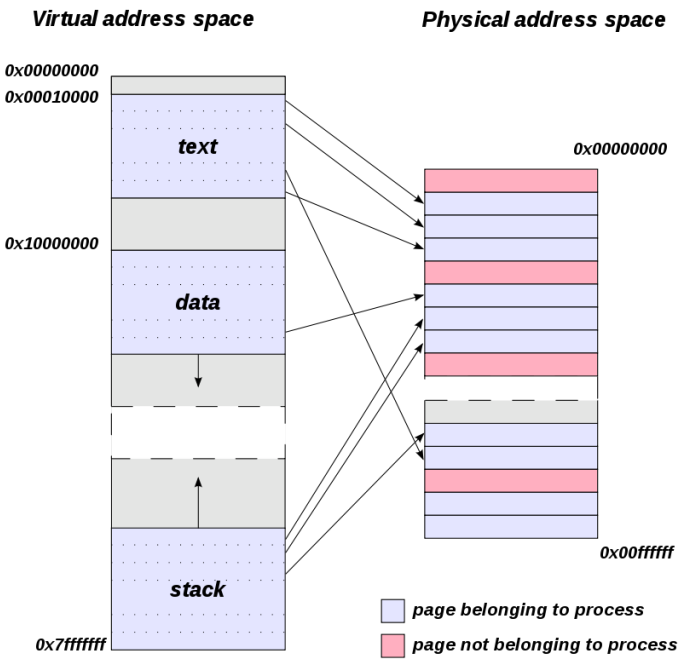
Za razliku od sustava sa statičkim stvaranjem procesa, u ovom sustavu procesi se stvaraju na zahtjev, zvanjem sistemskoga poziva fork().

### 3.2.6. Sustav s realnim adresnim prostorom

Za razliku od sustava s virtualnom memorijom gdje svaki sustav ima svoj virtualni prostor koji je u korisničkom načinu rada zaštićen od ostalih aplikacija i mapira se na realni, u sustavu s realnim adresnim prostorom ta distinkcija ne postoji. Svi procesi pokreću se u istom adresnom prostoru, prednost ovoga je brzina i lakoća implementacije, ali postoji opasnost od narušavanja integriteta podataka od procesa.

### 3.2.7. Sustav s virtualnim adresnim prostorom

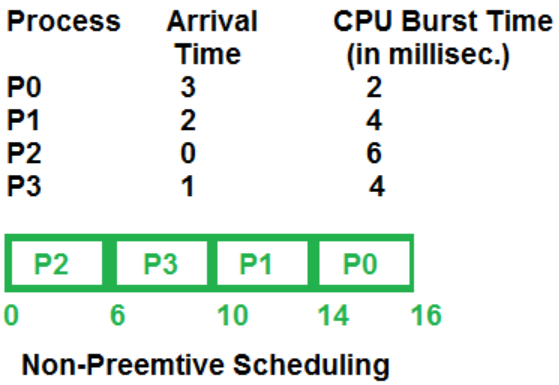
U sustavu s virtualnim adresnim prostorom postoji više tehnika mapiranja prostora vidljivoga procesa na realni adresni prostor. Proces logički vidi cijeli adresni prostor kao "svoj", iako zapravo dijelovi virtualnoga adresnoga prostora mapiraju neki dio realnog adresnoga prostora, na pr. nekoliko stranica.




**Slika 3.2:** Virtualni prostor

### 3.2.8. Nepreemptivni sustav

Nepreemptivno raspoređivanje procesa temelji se na dobrovoljnom otpuštanju resursa, točnije jedanput kada se resurs (CPU ciklus) alocira procesu, proces zadržava taj resurs sve dok se ne završi ili dok ne uđe u stanje čekanja. [5]

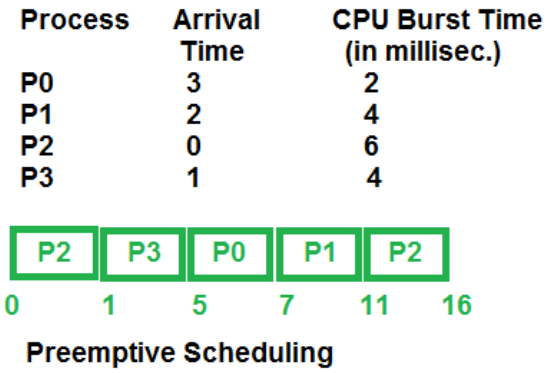



**Slika 3.3:** Nepreemptivno raspoređivanje

Algoritmi temeljeni na nepreemptivnom raspoređivanju: Round Robin (RR), Shortest Job First (SJF), Priority (nepreemptivna inačica)

### 3.2.9. Preemptivni sustav

Preemptivno raspoređivanje temelji se na dodjeljivanju resursa procesima u ograničenim vremenskim periodima i kad proces izgubi "pravo" na korištenje ciklusa CPU-a vraća se u red (eng. queue) procesa spremnih za izvršenje, gdje ostaje sve do idućega dodjeljivanja procesorskih resursa.




**Slika 3.4:** Preemptivno raspoređivanje

Algoritmi temeljeni na preemptivnom raspoređivanju su: Shortest Remaining Time First (SRTF), Priority (preemptivna inačica).

### 3.2.10. Realtime sustavi

Općenito govoreći, većina ugradbenih sustava ima neka vremenska ograničenja i ne će dopustiti procesu da iskorištava resurse neograničeno, međutim sustavi koji imaju stroga ograničenja u pogledu vremena odziva na prekid i završetak operacije zovu se operacijski sustavi za rad u realnom vremenu (eng. Real-time operting systems). Operacijski sustav koji može jamčiti maksimalno vrijeme za te operacije, naziva se „hard real-time“. S druge strane, postoje operacijski sustavi koji ne mogu u svim slučajevima jamčiti vrijeme izvođenja i oni se nazivaju „soft real-time“. U primjeni se pronalaze obje inačice RTOS-a, a prije upotrebe treba proučiti specifikacije.[7]

# 4. Operacijski sustav - implementacija i pregled

## 4.1. Arhitektonske odluke prilikom dizajniranja sustava

U sljedećem poglavljima predstavit će se i obrazložiti dizajnerske odluke pri izradi ovoga jednostavnoga OS i njihova implementacija unutar sustava te, ta analize algoritama koji se upotrebljavaju.

Cilj sustava jest postići preemptivno raspoređivanje s pomoću RMS (Rate Monotonic Scheduling) algoritma, efikasnu obradu prekida, stvaranje procesa i realni adresni prostor. Za sinkronizaciju koristit ćemo semafore.

## 4.2. Stvaranje procesa i procesni blok

Proces je osnovna jedinica u jednoprocesnome sustavu. Procesi će u našem slučaju imati jedinstveni identifikator s pomoću kojega ćemo ga moći identificirati, spremljenoga pokazivača (eng. pointer) na stoga, trenutnoga statusa, vremena pauze, vremena u kojem je proces spreman, spremljenih registara i pointera na funkciju koju ćemo izvršavati. [6]

```
1 struct proc
2 {
3     // za debbugiranje
4     char name[16];
5     volatile int pid;
6     struct proc *next;
7     int *ksp;
8     enum procstatus status;
9     int pause;
```

```
    int ready_time;
    int priority;
    int real_priority;
    int kstack[SSIZE];
}
```

### 4.2.1. Inicijalizacija procesa

Glavna funkcija u kodu, odnosno main, u jednom će trenutku pokrenuti inicijalizaciju jezgre (eng. kernel)

Sustav mora proći kroz listu (u našoj implementaciji u C-u to je array) procesa i svakom procesu pridijeliti pid - odnosno jedinstveni identifikator, postaviti vrijeme pokretanja na 0 i postaviti pokazivač na idući proces u nizu (implementacija vezane liste).

U tijeku rada sustava postojat će dvije liste - `free_list` i `ready_queue` kao red procesa koji su spremni da počnu s izvršavanjem i pokazivač na proces koji se trenutno izvršava `current_process`.

```
for (i = 0; i < PROC_NUM; i++) {
    p = &proc[i];
    p->pid = i;
    p->status = READY;
    p->ready_time = 0;
    p->next = p + 1;
    p->name = num_to_str(i);
}
```

Nakon toga pokazivač `free_list` postavlja se na prvi proces i time postaje HEAD vezane liste slobodnih procesa.

Potom je potrebno postaviti prvi proces kao `current_process` i njegov prioritet na 0.

### 4.2.2. Alokacija procesa sistemskim pozivom

Pošto smo popunili sve elemente niza struktura `struct proc` s osnovnim podatcima i smjestili ih u listu `free_list`, procese je potrebno alocirati i "kazati" im što će izvršavati. To radimo zvanjem sistemskoga poziva kfork(int body, int priority[6]

Sažetak funkcije int kfork(int body, priority):

```
new_proc->status = READY;
new_proc->priority = new_priority;
for (i=1; i<15; i++)
    new_proc->kstack[SSIZE-i] = 0; // registre postavljmo
    na inicijalne vrijednosti
new_proc->kstack[SSIZE-1] = body; // adresa povratka je pokazivac
na funkciju body
new_proc->ksp = &(new_proc->kstack[SSIZE-14]); // postavljamo
pokazivac na spremljeni stog
enqueue(&ready_queue, new_proc); // prebacujemo new_proc
u listu ready_queue
```

U gornjem isječku vidimo da je status procesa u stanju READY, prioritet je određujemo dinamički, a još moramo postaviti adresu povratne funkcije `body` na najgornje mjesto na stogu, jer stog, kao što je i napomenuto u prvome poglavlju o ARM-u raste prema dolje. Postavljamo pokazivač na stog. On se nalazi "ispod" spremljenih registara i povratne adrese.

Na posljetku, proces mećemo u `ready_queue` i time smo završili prvu fazu u životnom ciklusu procesa.

## 4.3. Semafori

U našem pokaznom primjeru za sinkronizaciju koristit ćemo semafore. Semafori su jednostavna struktura koja se sastoji od varijable brojača - tradicionalno nazvana S i reda blokiranih konzumenata - u našem slučaju procesa. Semafor omogućuje dvije operacije - blokirajuću operaciju zahtijevanja resursa i operaciju otpuštanja. Njihova tradicionalna imena su P i V. Međutim, radi lakšega snalaženja, zvat ćemo ih acquire i release, a S counter.

```
struct semaphore {
    int counter;
    struct proc *waiting_queue;
}
```

U kontekstu sinkronizacije procesa naše metode acquire i release izgledat će ovako: [10, 4]

```
int
acquire(struct semaphore *sem)
{
    disable_IRQ_interupts();
    sem->counter--;
    if (sem->counter < 0) {
        current_process->status = BLOCKED;
        enqueue(&sem->queue, current_process);
        enable_IRQ_interupts();
        switch_task();
    }
    enable_IRQ_interupts();
}
```

Ako proces iz bilo kojega razloga zapne, pozvat ćemo `taks_switch` koji će assemblerski pozvati raspoređivača i promijeniti kontekst.

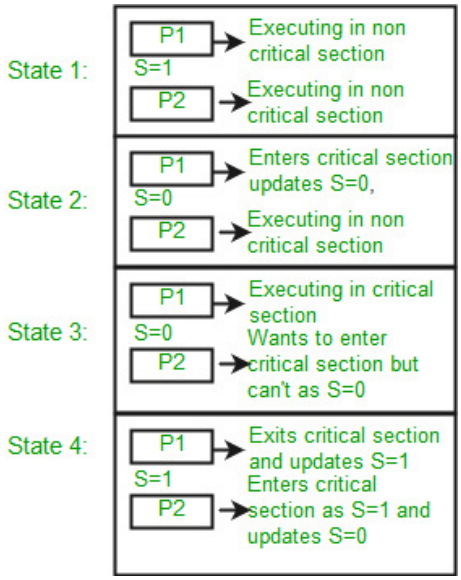
S druge strane release izgleda ovako:

```
int
release(struct semaphore *sem)
{
    struct proc *new_proc;
    disable_IRQ_interupts();
    sem->counter++;
    if (sem->counter <= 0) {
        new_proc = dequeue(&sem->queue);
        new_proc->status = READY;
        enqueue(&ready_queue, new_proc);
    }
    enable_IRQ_interupts();
}
```

Funkciju release pozivamo kad želimo osloboditi resurs, u našem slučaju, to je trenutak u početku perioda određenoga procesa u okviru RMS algoritma.

Semafori općenito mogu biti binarni (0 ili 1) ili mogu uzimati na se više vrijednosti. Može se reći da su semafori brojači (eng. counting semaphores) samo generalizacija binarnih semafora.

Primjer kontrole toka programa sa semaforima:




**Slika 4.1:** Primjer rada semafora

## 4.4. Algoritam za raspoređivanje

### 4.4.1. Opis i analiza

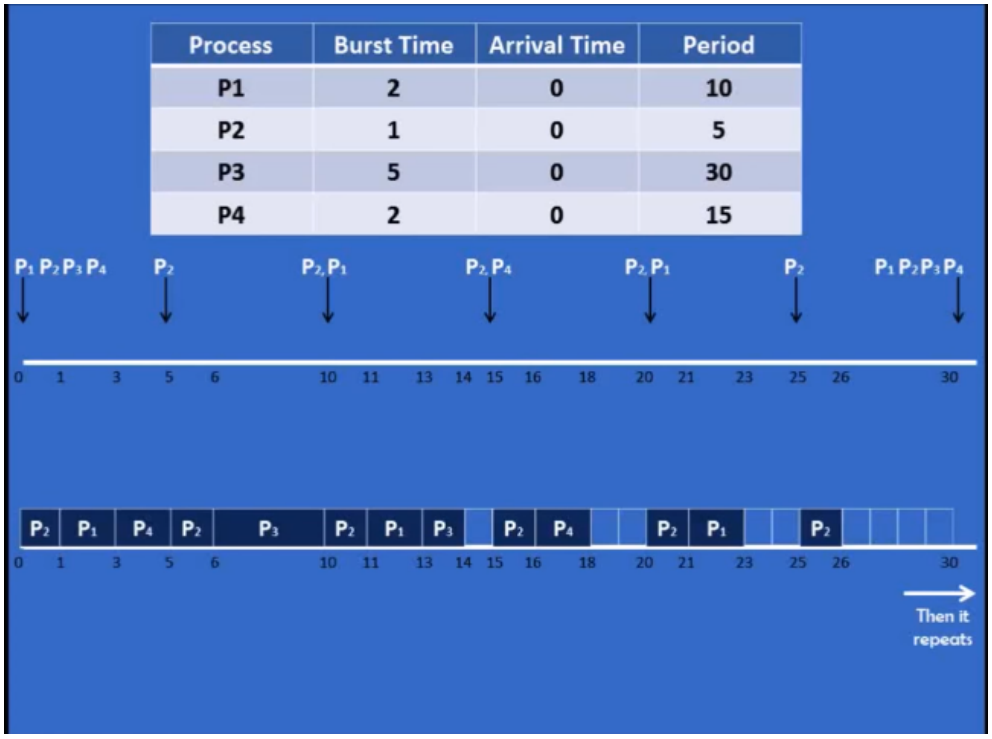
Rate-Monotonic scheduling (RMS) jest algoritam koji se temelji na statičkom dodjeljivanju prioriteta. RMS zahtijeva nekoliko pretpostavki da budu zadovoljene:

- Periodični zadatci: zadatci su periodični gdje je kraj perioda ujedno i nužno vrijeme izvršenja zadatka (eng. deadline)
- Prioretiziranje zadataka u skladu s nužnim vremenom izvršetka -> kraći period, veći prioritet
- Preempcija: zadatci s višim prioritetom odmah prekidaju zadatke s nižim prioritetom
- Zadatci ne dijele resurse koji bi ih blokirali
- Vrijeme izmjene konteksta jest 0

Asimptotskom analizom utvrđeno je da je utilizacija procesora kod RMS algoritma otprilike 69 %. Međutim, kod nekih kombinacija, ta utilizacija može biti i 100 %.[10, 3]. Utilizacija u slučaju da se zadovolje sva nužna vremena izvršenja jest:

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

**Slika 4.2:** RMS utilizacija




**Slika 4.3:** Primjer izvršavanja RMS algoritma

### 4.4.2. Scenarij

Recimo da imamo tri procesa P1, P2, P3.

| Proces | Kapacitet | Period |
|---|---|---|
| P1 | 3 | 20 |
| P2 | 2 | 5 |
| P3 | 2 | 10 |

Proces P2 ima najkraći period. Tj. on se mora (sic!) izvršiti jedanput po 2 jedinice unutar 5T. Zatim ide P3 i P1 kao proces s najdužim periodom.

Tijek izvođenja procesa izgledat će ovako:

1. Proces P2 izvodi se dvije vremenske jedinice, od 0 do 2
2. Proces P3 izvodi se 2 vremenske jedinice, od 2 do 4.
3. Proces P1 izvodi se 1 vremensku jedinicu i prekida ga proces P2 koji mora opet početi izvođenje da bi se završio u periodu od 5 do 10

4. Proces P2 izvrši se od 5 do 7
5. Izvrše se još dvije jedinice procesa P1
6. CPU je u stanju mirovanja od 9 do 10

### 4.4.3. Implementacija

U prvom dijelu koda vezanoga za raspoređivanje vidimo da ako je globalno vrijeme koje pratimo u našem sustavu s pomoću timera jednako početku perioda u kojem bi nas proces mogao početi (ali ne i završiti), uzimamo prvo onaj s najkraćim periodom i taj proces mećemo u `ready_queue`.

```
1  if ((global_time % (1*T)) == 0) {
2      release(&main_semphore[4]);
3      proc[4].ready_time = global_time;
4  }
5  if ((global_time % (2*T)) == 0) {
6      release(&main_semphore[3]);
7      proc[3].ready_time = global_time;
8  }
9  if ((global_time % (7*T)) == 0) {
10     release(&main_semphore[1]);
11     proc[1].ready_time = global_time;
12 }
```

Funkcija release pozvat će, kao što je i napomenuto u prošlome poglavlju, funkciju schedule koja će uz asemblerski poziv i promjenu konteksta dohvatiti prvi proces po prioritetu iz `ready_queuea` i time uspješno izvršiti promjenu procesa.

```
1  int
2  schedule()
3  {
4      if (ready_queue &&
5      ready_queue->priority > curret_process->priority) {
6          if (interrut_nesting == 0) {
7              switch_tasks();
8          } ...
9      }
10 }
```

## 4.5. Upravljački programi

U ovom dijelu ćemo se dotaknuti i teme koju dosad nismo načinjali, a to su upravljački programi, tj. driveri. Ovdje ćemo uz prikaz obrade prekida pokazati i strukturu drivera na primjeru timera.[2]

Asemblerski dio isti je kao i u potpoglavlju 2.4.4. u funkciji `obrada_prekida` pročitat ćemo izvor prekida i ovisno o tome pozvati funkciju za ispravnu obradu. Prethodno je potrebno timer inicijalizirati.

Recimo, jednostavnosti radi, da nas zanima samo jedan timer i da želimo na svaki prekid od timera okrenuti rutinu koja će povećati globalnu varijablu `global_time`.

```
1
2  struct timer
3  {
4      unsigned int *base_addr;
5  }
6
7  void
8  init_timer()
9  {
10     struct timer *new_timer;
11     global_time = 0;
12     //resetiranje valuea i interrupt status
13     i masked status registara
14     ...
15     //postavljamo Load registar na 0x100, tj. na decimalni 256
16     *(new_timer->base+0x00) = 0x100;
17     // Interrupt omogucen, reload omogucen, 32-bitni
18     *(new_timer->base+0x08) = 0x66;
19 }
```

U funkciji koja će se okidati kod prekida povećavat ćemo globalno vrijeme.

```
1  void
2  handle_timer_interrupt()
3  {
4      global_timer++;
5  }
```

# 5. Zaključak

Real-time operacijski sustavi jedna su od ključnih karika u radu s ugradbenim sustavima. Cilj ovoga rada bio je pokazati tijek razvoja jednoga takvoga operacijskoga sustava - od samoga početak, dakle proučavanja arhitekture sustava, zahtjeva koje imamo i određivanja ciljeva koje želimo postići. U ovom radu odabrali smo jednostavniji pristup i sustav s jednim procesorom i bez virtualizacije memorije. Također, odabrali smo algoritam koji odgovara svim potrebama real-time sustava i omogućuje nam da prioritiziramo procese sa skorijim nužnim rokom izvršenja. Na kraju, možemo reći da i takav jednostavan sustav sa samo nekoliko definiranih tipova (struktura) i nekoliko drivera može sasvim ispravno obaviti sve zadaće za koje je namijenjen, ali ujedno i ispuniti sve bitne zahtjeve RTOS-a, a to su obrada ugnjžeđenih prekida, ispunjavanje vremenskih rokova, izmjenu konteksta, komuniciranje s vanjskim jedinicama itd.

Možemo na kraju reći da je ovaj sustav uspio ispuniti te minimalne zadatke koje smo zacrtali prije izrade ovoga rada.

# LITERATURA


[1] Arm architecture. . URL https://en.wikipedia.org/wiki/ARM_architecture.

[2] Arm dual timer module. . URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0271d/index.html.

[3] Rate-monotonic scheduling. URL https://en.wikipedia.org/wiki/Rate-monotonic_scheduling.

[4] Operating system | semaphores in operating system. URL https://www.geeksforgeeks.org/semaphores-operating-system/.

[5] Remzi H. Arpaci-Dusseau i Andrea Arpaci-Dusseau. *Operating Systems Three Easy Pieces*. 2015.

[6] Russ Cox, Frans Kaashoek, i Robert Morris. *xv6 a simple, Unix-like teaching operating system*. 2014.

[7] Luka Kelava. Real-time operacijski sustavi. 2017.

[8] Jake Sandler. Building an operating system for the raspberry pi. URL https://jsandler18.github.io/.

[9] William Stallings. Embedded operating systems. URL https://www.unf.edu/public/cop4610/ree/Notes/PPT/PPT8E/CH13-OS8e.pdf.

[10] K.C. Wang. *Embedded and Real-Time Operating Systems*. 2016.

## Operacijski sustav za rad u stvarnom vremenu


### Sažetak

Rad započinje opisom općenito ARM arhitekture te osnovnih mogućnosti ARM-ova assemblera kako bi se mogao ispratiti ostatak rada. Nastavlja se opisom načina izvršavanja i prevođenja koda, početne točke u OS-u koja obrađuje reset. Nadalje, dan je prikaz osnovnih podjela operacijskoga sustava i na posljetku je prikazana implementacija zanimljivih dijelova OS-a, ključne operacije prilikom zamjene aktivnih procesa, algoritam zamjene i prioritiziranja i jednostavni driver.



**Ključne riječi:** Operacijski sustavi u stvarnom vremenu, ARM, raspoređivač, proces, prekid, ugradbeni sustav, RMS algoritam.


## Title


### Abstract

The work begins with a general description of the ARM architecture and the basic capabilities of the ARM assembler as kind of prerequisites for following rest of the paper. The description of the method of executing and translating the code, the starting point in the OS that handles the resets. Additionally, it is an overview of the basic operating system subdivisions and finally shows the implementation of interesting OS parts, key operations for replacing active processes, scheduling, replacing and prioritizing algorithms, and finally a simple driver.



**Keywords:** Real-time operating systems, ARM, scheduler, process, interrupt, embedded system, RMS algorithm.