Pismeni ispit iz predmeta *Operacijski sustavi za ugrađena računala*, 28.6.'10.

1. (3 boda) Nadopuniti skriptu povezivača prikazanu desno. Odrediti vrijednosti svih varijabli korištenih u skripti.

```
code = 0x100000
data = code + SIZEOF(jezgra*(.text))  ***
bss = data + SIZEOF(jezgra*(.data))   ***
programi = code + SIZEOF(.jezgra)
ostatak = programi + SIZEOF(program1) +
          SIZEOF(program2)
```

```
SECTIONS {
        .jezgra 0x100000 : {
                code = .;
                jezgra*(.text)
                data = .;
                jezgra*(.data)
                jezgra*(.rodata.*)
                bss = .;
                jezgra*(.bss)
                . = ALIGN(4096);
        }
        programi = .;

        .program1 0 : AT (programi)
        {
                prog1*(.text)
                prog1*(.rodata.*)
                prog1*(.data)
                prog1*(.bss)
                . = ALIGN(4096);
        }
        .program2 0 :
        AT(programi + SIZEOF(program1))
        {
                prog2*(.text)
                prog2*(.rodata.*)
                prog2*(.data)
                prog2*(.bss)
                . = ALIGN(4096);
        }

        ostatak = programi +
                  SIZEOF(program1) +
                  SIZEOF(program2)
}
```

2. (3 boda) U nekom kodu (.c) nalazi se funkcija:

```
int zbroji ( int m, int n, int *p ) {
   int suma, paran;
   suma = m + n + p;
   paran = ~(suma & 1);
   *p = paran;
   return suma;
}
```

Prikažite stanje stoga (što se sve nalazi na stogu) dretve koja je pozvala navedenu funkciju u trenutku prije izvođenja zadnje linije koda (`return suma`) (uz pretpostavku da se optimizacije neće koristiti).

```
(vrh stoga)              paran
                         suma
                         stara kazaljka „okvira" stoga (ebp, nije potrebno)
                         povratna adresa (kamo se vraća nakon funkcije zbroji)
                         m
                         n
                         p
(tu je na stogu nešto što ovisi o širem kontekstu - ne da se iz primjera
   odrediti)
```

3. (3 boda) Što sve spada u (puni) kontekst dretve? Navedite primjer za x86 ili ARM7 sustave.

```
svi registri procesora koja dretva koristi:

za x86: eax, ebx, ..., registri „koprocesora", mmx registri, segmentni registri i ostali
koji se koriste za upravljanje spremnikom (cr0, cr3, cr4, ...) te registar stanja
(eflags)

za arm7: r0-r15 + CPSR (reg. stanja)
```

4. (6 bodova) Jedan projekt za ugrađene sustave sastoji se od tri datoteke: `pocetak.c`, `jezgra.c` i `programi.c` (te njihova zaglavlja, ekvivalentne `.h` datoteke). Sve se datoteke sastoje od `.text`, `.rodata` i `.data` odjeljaka. Datoteke je potrebno prevesti i povezati (*kompajlirati i linkati*) tako da se slika sustava može staviti u ROM na adresu 0. Obzirom da je ROM dovoljno velik samo za pohranu slike, kôd u `pocetak.c` (koji treba postaviti na adresu 0, a koji nema `.data` odjeljak) će odmah po početku izvođenja (*reseta*) premjestiti podatke „jezgre“ (`.data` odjeljak koji nastaje iz datoteke `jezgra.c`) na spremničku lokaciju 0x10000 (prva adresa RAM-a). Pri pokretanju programa (koji nastaju iz datoteke `programi.c`) njihovi se podaci (`.data` odjeljak) moraju kopirati u RAM. Za upravljanje spremnikom potrebno je pripremiti odgovarajuće dijelove sustava za predviđene lokacije. Za programe se koristi straničenje, tj. programe treba pripremiti tako da započinju s logičkom adresom nula. Jezgra radi u apsolutnim adresama.

   Pored toga, potrebne su slijedeće adrese:

`j_pod_rom`:	adresa podataka jezgre u ROMu (za kopiranje u RAM)
`j_pod_ram`:	adresa podataka jezgre u RAMu (za kopiranje iz ROMa)
`prog_rom`:	adresa instrukcija programa u ROMu (`.text` i `.rodata` odjeljaka)
`prog_d_rom`:	adresa podataka programa u ROMu (`.data` odjeljak)
`slobodno`:	prva adresa iza podataka jezgre, a koja će se koristiti za straničenje (za podatke programa, u RAM-u).

   Skicirati skriptu za povezivača (*linkera*) koji će napraviti datoteku slike sustava koja zadovoljava navedene zahtjeve. Sve se objektne datoteke nalaze u istom direktoriju (neka filter bude ime objektne datoteke). Po potrebi dodati i dodatne varijable u skriptu.

```
SECTIONS {

        .pocetak 0 : AT (0) {
                pocetak.o (.text .rodata)
        }

        .jezgra_kod : {
                jezgra.o (.text .rodata)
        }

        jezgra_pod_rom = .;
        j_pod_ram = 0x10000;
        .jezgra_pr j_pod_ram : AT ( jezgra_pod_rom ) {
                jezgra.o (.data)
                . = ALIGN(4096);
        }

        prog_rom = jezgra_pod_rom + SIZEOF ( jezgra_pr );
        .prog_kod 0 : AT ( prog_rom ) {
                programi.o (.text .rodata)
        }

        prog_d_rom = prog_rom + SIZEOF( prog_kod );
        .prog_pod 0 + SIZEOF(.prog_kod) : AT ( prog_d_rom ) {
                programi.o (.data)
        }

        slobodno = j_pod_ram + SIZEOF ( .jezgra_pr );

}
```

5. (3 boda) Nadopuniti navedene datoteke odgovarajućim ključnim riječima (`static`, `extern`, ...) i <u>slično</u> tako da se navedeno može prevesti i da ima smisla. Sve što ne treba ili ne smije biti „globalno“ označiti lokalnim. Dati i ostale „korisne preporuke“ prevoditelju.
   (2 boda) Napisati odgovarajući „`Makefile`“.

<u>N.h:</u>
```
inline
void dodaj_N ( int broj );
inline
int suma_N ();
```

<u>R.h:</u>
```
inline
void dodaj_R ( double broj );
inline
double suma_R ();
```

<u>C.h:</u>
```
typedef struct _C_ {
   double r, i;
} C;
inline
void dodaj_C ( C broj );
inline
C suma_C ();
```

<u>N.c:</u>
```
static
int suma = 0;

extern double naj;

inline static
void zbroji ( int broj ) {
   suma += broj;
   if ( broj > naj )
      naj = broj;
}
inline static
int provjeri ( int broj ) {
   if ( broj>0 && broj<1000 )
      return 1;
   else
      return 0;
}
inline
void zbroji_N ( int broj ) {
   if ( provjeri ( broj ) )
      zbroji ( broj );
}
inline
int suma_N () {
   return suma;
}
```

<u>R.c:</u>
```
static
double suma = 0;

extern double naj;

inline static
void zbroji ( double broj ) {
   suma += broj;
   if ( broj > naj )
      naj = broj;
}
inline static
int provjeri ( double broj ) {
   if ( broj>0 && broj<1000 )
      return 1;
   else
      return 0;
}
inline
void zbroji_R ( double broj ) {
   if ( provjeri ( broj ) )
      zbroji ( broj );
}
inline
double suma_R () {
   return suma;
}
```

<u>C.c:</u>
```
static
C suma = { 0, 0 };


inline static
void zbroji ( C broj ) {
   suma.r += broj.r;
   suma.i += broj.i;
}
inline static
int provjeri ( C broj ) {
   if(broj.r>0 && broj.r<1000)
      return 1;
   else
      return 0;
}
inline
void zbroji_C (  C broj ) {
   if ( provjeri ( broj ) )
      zbroji ( broj );
}
inline
C suma_C () {
   return suma;
}
```

<u>brojke.c:</u>
```
#include „N.h“ i „R.h“ i „C.h“
double naj = 0;

int main () {
   C broj;
   broj.r = 0;
   broj.i = 100;
   while ( broj.r < broj.i ) {
      zbroji_N ( (int)broj.i );
      zbroji_R ( broj.r );
      zbroji_C ( broj );
      broj.r += 1;
      broj.i -= 1;
   }

   broj = suma_c ();
   printf ("%d:%g:%g+%g:%g\n",
       suma_N (), suma_R (),
       broj.r, broj.i, naj );

   return 0;
}
```

<u>Makefile:</u>
```
brojke: brojke.o N.o R.o C.o
    gcc brojke.o N.o R.o C.o -o brojke

brojke.o: brojke.c N.h R.h C.h
    gcc -c brojke.c

brojke.o: N.c N.h
    gcc -c N.c

brojke.o: R.c R.h
    gcc -c R.c

brojke.o: C.c C.h
    gcc -c C.c
```

6. (5 bodova) Skicirajte ostvarenje nadzornog alarma (*watchdog timer*) korištenjem ili sučelja prikazanog sustava (okvirno) ili sučelja iz „standardnih sustava“. Prikazati korištenje sučelja na primjeru.

```
void istek_periode ( void *param )
{
    int id_dretve = (int) param;
    ispisi ( „istek nadzornog alarma za dretvu %d - dretva se prekida!“,
             id_dretve );
    prekini_dretvu ( id_dretve );
}

int postavi_nadzorni_alarm ( int id_alarma, vrijeme_t perioda, int id_dretve )
{
    return postavi_alarm ( id_alarma, perioda, istek_periode,
                           (void *) id_dretve, NEISTINA);
}

void neka_dretva ( void *param )
{
    ...
    id_na = stvori_nadzorni_alarm ( STVORI_NOVI, perioda, id_dretve () );
    ...
    (u nekoj petlji u kojoj dretva radi i mora raditi periodički poziva:)
    stvori_nadzorni_alarm (id_na, perioda, id_dretve () );
    (produžava se alarm za periodu)
    ...
}
```

7. (5 bodova) Neki sustav posjeduje sklop koji svakih 10ms generira prekid iz kojeg se poziva funkcija *j_prekid_sata*. Skicirati tu funkciju, funkciju *j_trenutno_vrijeme* (koja vraća trenutno vrijeme u preciznosti od 10ms – vrijeme od pokretanja sustava) te dodati potrebnu podatkovnu strukturu tako da se navedene funkcije mogu ostvariti. Vrijeme treba iskazati u milisekundama.

```
static int j_sat; //vrijeme sustava

void j_prekid_sata ()
{
    j_sat += 10;
}

int j_trenutno_vrijeme ()
{
    return j_sat;
}
```

8. (2 boda) Koje su prednosti i nedostaci upravljanja prekidima na niskoj razini (u „arh“ sloju) u odnosu na upravljanje na višoj razini (u „jezgri“, dok je u „arh“ sloju samo osnovno upravljanje kao u prikazanom sustavu)?

prednosti: brže (izravno i u IDT za x86; manje kućanskih poslova)
nedostaci: manja prenosivost (treba više mijenjati pri prijenosu na druge sustave), moguće više koda – veća jezgra (svaki prekid ima svoju pohranu i obnovu konteksta...)

9. (3 boda) Čemu služe programski prekidi? Može li se ista funkcionalnost napraviti i drugim načinima/mehanizmima (bar u posebnim slučajevima)? Kada i kako (ako da)?

za poziv jezgrine funkcije koja se obavlja pod većim privilegijama

sustavi koji nemaju „korisnički“ i „nadgledni“ način rada mogu istu funkcionalnost ostvariti izravnim pozivima funkcija (ne preko „skupih“ prekida)

10. (3 boda) Gdje se sve može koristiti ključna riječ *static*? Opišite značenja svake od uporaba.

- izvan funkcija, tada navedena varijabla je „globalna“ ali samo za navedenu datoteku (simboličko ime nije dostupno za druge .c datoteke, ni sa extern)
- funkcija može biti static – dostupna je (kao simboličko ime) samo u navedenoj datoteci
- varijabla u funkciji može biti static – ta se varijabla ponaša kao globalna – njena vrijednost se ne alocira na stogu kao ostale lokalne varijable, već statički je za nju rezervirano mjesto, ali za razliku od globalne njoj se može pristupiti samo iz te funkcije (slični je princip u objektnim jezicima s statičkim varijablama)

11. (2 boda) Uobičajene funkcije sinkronizacije uključuju neke inačice *čekaj* i *postavi*. Koja proširenja tih funkcija susrećemo u „standardnim“ sučeljima? Kratko opišite funkcionalnosti tih proširenja.

probaj_čekati - ne blokira dretvu ako ne može „proći normalno“ (npr. semafor nije prolazan)
čekaj_ograničeno - ako se dretva blokira, trajanje blokiranog stanja je ograničeno na vrijeme zadano kao parametar funkcije

12. (2 boda) Što su to signali (na razini OS-a, ne na električnoj razini)? Kako dretva može definirati svoje ponašanje po primitku signala (npr. u UNIX-u i sličnim sustavima)?

signali – sredstvo dojave asinkronih događaja koji se tiču dretve; sredstvo komunikacije između dretvi te između OS-a i dretvi
ponašanje za signal:
- ignoriraj signal
- obradi signal pretpostavljenom funkcijom (definirano OS-om, tj. bibliotekom pri stvaranju programa)
- obradi signal prethodno definiranom funkcijom (dretva definira)
- privremeno zadrži (blokiraj) signal (ne odbacuje se već čeka promjenu ponašanja za njega)

13. (3 boda) Operacijski sustav omogućava korisničkim dretvama *registraciju* na pojedine asinkrone događaje (npr. na istek alarma, na primitak signala) tako da se za te događaje pozovu odgovarajuće funkcije. Kako implementirati te funkcionalnosti u OS-u? Koji se problemi tu javljaju i kako ih riješiti (npr. kako su riješeni u prikazanom sustavu)?

Treba pozvati zadanu funkciju koju je dretva definirala, ali u kontekstu dretve! Problem je to implementirati kada u sustavu postoje različiti načini rada procesora i različiti procesi jer tada treba pripremiti okruženje za poziv zadane funkcije.
Jedan od načina je stvoriti novu dretvu unutar istog procesa, s istim svojstvima kao dotična dretva, te obradu funkcije obaviti u toj dretvi (ta je funkcija početna za dretvu).
Drugi je način „umetnuti“ obradu te funkcije zadanoj dretvi. Obzirom da se to radi u jezgrinoj funkciji kada je kontekst dretve na njenom stogu (ili drugom mjestu), može se taj kontekst pohraniti, a umjesto njega napraviti novi, tako da kada se vratimo u tu dretvu najprije idemo u obradu zadane funkcije (zadanog događaja)

14. (3 boda) Prijelaz s „flat“ modela upravljanja spremnikom („bez upravljanja“, svi vide sve u apsolutnim adresama, k0-k11) na straničenje zahtijeva znatne izmjene u implementaciji. Kratko opisati potrebne izmjene (gdje i koje).

za svaki proces treba izgraditi tablicu prevođenja
promjene su:
- u pohrani konteksta i obnovi konteksta (treba pohraniti/ažurirati potrebne registre koji upravljaju straničenjem i pokazuju na tablicu prevođenja);
- podsustav za upravljanje tablicama prevođenja (stvaranje, dodavanje zapisa, ...)
- komunikacija jezgra-dretva i obratno je „otežana“ jer treba interno prevoditi adrese (programski)

15. (2 boda) Što sve spada u opisnik procesa?

- id procesa, id korisnika, ...
- opis adresnog prostora procesa
- opis načina raspoređivanja i prioritete
- popis dretvi
- popis korištenih resursa sustava
- maske za signale (mada to je bolje u opisnike dretvi)