Sistemski softver - Domaći zadatak - Emulator

Miloš Aćimović 2014/0146 RTI

am140146d@student.etf.bg.ac.rs
Elektrotehnički fakultet,
Univerzitet u Beogradu

31. maj 2017

Sadržaj

1 Opis projekta

Napraviti interpretativni emulator za arhitekturu opisanu u prilogu. Ulaz emulatora je izlaz asemblera koji je moguće učitati i pokrenuti pod uslovom da nema nedefinisanih simbola i da sekcije mogu da se učitaju na predviđene adrese bez preklapanja. Naziv predmetnog programa se zadaje kao argumenti komandne linije. Dodatni preduslov za pokretanje je i da je definisan globalni simbol START od kojeg emulator počinje izvršavanje. Prekidi:

- mehanizam prekida je vektorisan, pri čemu IVT počinje od adrese 0 i zadrži 32 ulaza
- u toku izvršavanja prekida nisu dozvoljeni ugneždeni prekidi
- izvršavanjem instrukcije INT 0 se završava rad emulatora
- ulaz 0 u IVT sadrži početnu vrednost SP registra
- ulaz 3 sadrži adresu prekidne rutine koja se izvršava u slučaju bilo koje greške
- ulaz 4 sadrži adresu prekidne rutine koju je potrebno izvršavati periodično sa periodom 0.1s realnog vremena
- ulaz 5 sadrži adresu prekidne rutine koja se izvršava svaki put kada je pritisnut neki taster emuliranog računara.

U memorijski adresni prostor su mapirana i dva registra, neposredno posle IVT. Prvi registar je mapiran na adresu 32 i predstavlja adresu registra podataka za ispis na standardni izlaz. Registar je veličine bajta. Vrednost upisana na ovu adresu predstavlja ASCII kod znaka koji treba ispisati na standardni izlaz. Drugi registar je mapiran na adresu 36 i predstavlja registar podataka ulazne periferije. Nakon što se dogodi prekid na ulazu 5, ovaj registar sadrži ASCII kod znaka koji odgovara pritisnutom tasteru. Vrednost se može čitati i više puta, sve dok se ne dogodi naredni prekid na ulazu 5, kada u tom registru postaje dostupna naredna vrednost. Na nivou emulatora obezbediti sinhronizaciju da se ne generiše novi prekid pre nego se pročita ranije postavljena vrednost. Obzirom da je adresni prostor emulirane arhitekture 4GB, jasno je da nije moguće emulirati čitav adresni prostor, već je neophodno implementirati rešenje koje će obezbediti alokaciju dovoljne količine memorije. Količina alocirane memorije ne bi trebala da bude značajno veća od neophodne za izvršavanje programa.

2 Opis rešenja

3 Emulator

3.1 Uputstvo za prevođenje i pokretanje emulatora

Emulator se prevodi komandom:

```
make
```

Emulatoru se kao parametar prosleđuje ime predmetnog programa.

```
./emulator putanja_do_fajla/naziv_predmetnog_fajla.o
```

Primer pokretanja bi bio(za predmetni program u istom direktorijumu kao i emulator):

```
./emulator sample.o
```

3.2 Opis emulatora

Ulaz emulatora je izlaz asemblera koji je moguće učitati i pokrenuti pod uslovom da nema nedefinisanih simbola i da sekcije mogu da se učitaju na predviđene adrese bez preklapanja. Dodatni preduslov za pokretanje je i da je definisan globalni simbol START od kojeg emulator počinje izvršavanje. Emulator čita ceo predmetni program izgrađujući prvo tabelu simbola, a zatim relokacione tabele za svaku sekciju i sadržaj svake sekcije. I to radi u sledećoj funkciji:

```
void read_object_file(FILE* input,
std::vector<std::vector<Symbol*>>& symtbl,std::vector<RelTable*>& reltbls);
```

Po čitanju predmetnog fajla, emulator prvo smešta sekcije za koje je predefinisana adresa u adresnom prostoru emuliranog računara, proveravajući svaki put da li postoji preklapanje sa bilo kojom drugom predefinisanom(u smislu adrese na kojoj se nalazi) sekcijom. Zatim emulator raspoređuje ostale sekcije redom kako su navedene u predmetnom programu.

3.3 Emuliranje memorije

Pošto je adresni prostor emuliranog računara iste veličine kao i adresni prostor host mašine koristi se mehanizam stranične organizacije emulirane memorije, pa se svakoj sekciji dodeljuje veličina_sekcije/veličina_stranice stranica zaokružen na gore. Veličina stranice je uzeta fiksno - 4096B. Za vođenje evidencije o svakoj stranici napravljena je sledeća struktura:

```
typedef struct page_t{
        uint32_t     base;
        std::vector<char>      flags;
        uint8_t*    mem;
        struct page_t* next;
        page_t():base(0), flags(0), mem(NULL), next(NULL){}
}page_t;
```

Gde je

- *base* pokazivač na početak stranice;
- *flags* niz karaktera (W - označava da je upis u stranicu dozvoljen);
- *mem* pokazivač na alociranu memoriju u adresnom prostoru emulatora;
- *next* pokazivač na sledeći element deskriptora stranice;

Ulančana sortirana lista po polju *base* se održava. Za upis u memoriju i čitanje iz memorije su napravljene posebne funkcije.

```
void free_memory(page_t* page_table);
```

Ova funkcija oslobađa i alociranu memorju i *page_t* strukture.

```
page_t* memory_page_alloc();
```

Ova funkcija alocira jednu *page_t* strukturu.

```
void  memory_page_add(page_t** page_table, page_t* p);
```

Ova funkcija dodaje alociranu strukturu *p* u tabelu *page_table*.

```
page_t* get_last_page_entry(page_t* p);
```

Ova funkcija dohvata poslednju *page_t* strukturu u sortiranoj listi struktura po polju *base*.

```
int memory_read_byte(page_t* page_table, uint32_t address, uint8_t& byte, Core* core);

int memory_write_byte(page_t* page_table, uint32_t address, uint8_t byte);

int memory_read_word(page_t* page_table, uint32_t address, uint16_t& word);

int memory_write_word(page_t* page_table, uint32_t address, uint16_t word);

int memory_read_dword(page_t* page_table, uint32_t address, uint32_t& dword);

int memory_write_dword(page_t* page_table, uint32_t address, uint32_t dword);
```

Ovo su funkcije za čitanje i upisivanje bajta, reči i duple reči, respektivno. Za upis i čitanje lista se pretražuje za polje base koje odgovara traženoj adresi.

3.4 Emuliranje procesora

Glavna petlja emulatora je urađena u sledećoj funkciji:

```
void emulate(Emulate emul);
```

Kada se pripremi emulirana memorija pravljenjem tabele stranica pronalazi se simbol *START* i njegova vrednost se ugrađuje u polje *pc* strukture *Core*. Struktura *Core* izgleda:

```
typedef struct core{
        uint32_t pc;
        uint32_t sp;
        page_t* page_table;
        std::mutex mutex;
        std::condition_variable cv;
        cpu_regs regs;
        bool interrupted;
        bool in_reg_read;
        bool waiting_input;
        bool key_pressed;
        bool running;
        core(uint32_t start, uint32_t s) : pc(start), sp(s), page_table(NULL), interrupted(false),
  in_reg_read(true), waiting_input(false), mutex(), cv(), running(true), key_pressed(false) {}
}Core;
```

- *pc* polje predstavlja tekuću vrednost programskog brojača;
- *sp* polje predstavlja tekuću vrednost stek pokazivača;
- *page_table* polje predstavlja pokazivač na prvi element sortirane liste *page_t* struktura;
- *mutex* polje predstavlja semafor inicijalizovan na 1;
- *cv* polje predstavlja uslovnu promenljivu;
- *regs* polje predstavlja niz opštenamenskih registara;
- *interrupted* polje označava da li je prekidna rutina u toku jer ugnježđavanje prekida nije dozvoljeno;
- *in_reg_read* polje označava da li je ulazni registar pročitan ili nije;
- *waiting_input* polje označava da li jedna od niti čeka na ulaz;
- *key_pressed* polje označava da treba da dođe do prekida koji opslužuje pritisnut taster;
- *running* polje označava uslov nastavka ulazka u glavnu petlju emulatora.

Uvođenje strukture *Emulate* ima poseban značaj koji će biti kasnije pojašnjen. U glavnoj petlji emulatora nakon što se pročita jedna dupla reč sa adrese *core->pc* interpretira se svaka instrukcija na osnovu operacionog koda.

```
void int_inst(uint32_t instruction1,page_t* page_table, Core& core, bool is_inst, uint32_t int_num);

void jmp_inst(uint32_t instruction1,page_t* page_table,Core& core);

void call_inst(uint32_t instruction1,page_t* page_table,Core& core);

void ret_inst(uint32_t instruction1,page_t* page_table,Core& core);

void jz_inst(uint32_t instruction1,page_t* page_table,Core& core);
```

```
void jnz_inst(uint32_t instruction1,page_t* page_table,Core& core);

void jgz_inst(uint32_t instruction1,page_t* page_table,Core& core);

void jgez_inst(uint32_t instruction1,page_t* page_table,Core& core);

void jlz_inst(uint32_t instruction1,page_t* page_table,Core& core);

void jlez_inst(uint32_t instruction1,page_t* page_table,Core& core);

void load_inst(uint32_t instruction1,page_t* page_table,Core& core);

void store_inst(uint32_t instruction1,page_t* page_table,Core& core);

void push_inst(uint32_t instruction1,page_t* page_table,Core& core);

void pop_inst(uint32_t instruction1,page_t* page_table,Core& core);

void add_inst(uint32_t instruction1,page_t* page_table,Core& core);

void sub_inst(uint32_t instruction1,page_t* page_table,Core& core);

void mul_inst(uint32_t instruction1,page_t* page_table,Core& core);

void div_inst(uint32_t instruction1,page_t* page_table,Core& core);

void mod_inst(uint32_t instruction1,page_t* page_table,Core& core);

void and_inst(uint32_t instruction1,page_t* page_table,Core& core);

void or_inst(uint32_t instruction1,page_t* page_table,Core& core);

void xor_inst(uint32_t instruction1,page_t* page_table,Core& core);

void not_inst(uint32_t instruction1,page_t* page_table,Core& core);

void asl_inst(uint32_t instruction1,page_t* page_table,Core& core);

void asr_inst(uint32_t instruction1,page_t* page_table,Core& core);
```

3.5 Obrada prekida

Provera prekida u slučaju pritisnutog tastera ili u slučaju periodičnog prekida se vrši na kraju *switch-case* strukture glavne petlje emulatora. Da bi se omogućilo istovremeno čekanje na unos pritisnutog tastera i nesmetano emuliranje datog programa pokrenute su dve niti nad glavnom petljom emulatora koja je zaštićena *core->mutex* poljem *Core* strukture. Jedna nit, ona koja pre stigne, će ili čekati na uslovnoj promenljivoj dok se prethodno uneta vrednost ne pročita ili će čekati blokirana na pozivu *getchar()* funkcije, čekajući da se na standardni ulaz unese karakter. Zbog kreiranja niti korišćena je promenljiva tipa *vector<thread>* i pozivom *threads.emplace_back(emulator, emul)* se incijalizovala kontrolna struktura niti i pokrenuta je nad funkcijom *emulator* i sa argumentom *emul* koja je strkutura tipa:

```
typedef struct emulate {
        Core* core;
}Emulate;
```

tj. sadrži samo jednu promenljivu tipa pokazivač na *Core* koja je deljena između dve niti. Iz razloga što emplace_back kreira novu strukturu tipa *Emulate* za inicijalizaciju instance niti korišćenjem *copy constructor*-a koji u slučaju strukture kopira svako polje(*plitka kopija*) preko pokazivača na *Core* se postiglo deljenje strukture.

4 Testovi

4.1 test

Pokretanje(iz bin direktorijuma):

```
./emulator ../tests/test1.o
```

```
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
sadtimer
ftimer
timer

pritisnut s
pritisnut a
pritisnut d
pritisnut f
timer
timer
timer
timer
```

```
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
timer
^C
```

4.2 test

Pokretanje(iz bin direktorijuma):

```
./emulator ../tests/test2.o
```

```
123456789:
```

4.3 test

Pokretanje(iz bin direktorijuma):

```
./emulator ../tests/test3.o
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

4.4 test

Pokretanje(iz bin direktorijuma):

```
./emulator ../tests/testerror1.o
```

```
Error: symbol START needs to be defined.
```

4.5 test

Pokretanje(iz bin direktorijuma):

```
./emulator ../tests/testerror2.o
```

```
Error: there are undefined symbols.
```

4.6 test

Pokretanje(iz bin direktorijuma):

```
./emulator ../tests/testerror3.o
```

```
Error: two sections: .data and .text collide with their respective ORG directives
```