Namenski Računarski Sistemi

Laboratorijske vežbe: Sistemski pozivi, prekidi i preključenje procesora (context cwitch)

1 Uvod

U ovim vežbama obradićemo dve bitne stvari:

- Pisanje obrađivača sistemskog poziva,
- Smena slike procesa (context switch) između dva programa koja se istovremeno izvršavaju.

2 Obrađivač sistemskog poziva

Počećemo sa primerom obrađivača sistemskog poziva. U prethodnim vežbama smo videli kako da koristimo simulator *Keyboard and Display Simulator* tako što smo pristupali njegovim registrima i čitali/pisali u njih. Sad ćemo uvesti dva nova sistemska poziva:

Syscall	Naziv	Svrha
104	myprint_string	Slično kao print_string, ali koristi Display simulatora
108	myread_string	Slično kao read_string, ali koristi Keyboard simulatora

U praksi direktni pristup registrima *Keyboard and Display Simulator*-a ne bi trebalo da bude omogućen korisničkim programima. To znači da je ispravan način da se pristupi registrima *Keyboard and Display Simulator*-a i vrše U/I operacije zapravo preko sistemskog poziva.
Sistemski poziv sakriva detalje rukovanja uredjajem i pruža nam jednostavan API. Zato ćemo sada imati dva nova sistemska poziva: syscall 104 i syscall 108.

Da bismo to uradili trebaće nam dve asemblerske datoteke:

1. Korisnički program koji koristi sistemske pozive - Program radi u korisničkom režimu izvršenja.
2. Obrađivač sistemskih poziva koji obavllja pristup simuliranom uređaju - Obrađivač radi u kernel režimu izvršenja.

2.1 Analiza datoteka

Ako pogledamo *mips6use_new_syscalls.asm*, videćemo da je to običan korisnički program koji koristi sistemske pozive. Jedino se neki sistemski pozivi dupliraju radi potvrde da novi sistemski pozivi rade ispravno.

Zahtevnija datoteka za analizu jeste *mips6syscall_handler.asm*, zato ćemo je obraditi po stavkama.

- Imamo kod obrađivača sistemskih poziva, koji počinje od labele `handler:` , a završava se `eret`

instrukcijom.

- Imamo funkciju *myprint_string()*. Napisana je kao i svaka druga funkcija, uključujući rad sa stek frejmom i slično. To je zato što i program koji se piše za kernel režim rada procesora poštuje ista pravila kao i korisnički program. Ono što funkcija radi jeste da ispituje kontrolni registar uređaja dok uređaj ne javi da je spreman za prijem karaktera. Zatim se karakter koji trenutno ispisujemo smesta u registar podataka *Display* uređaja. Ova sekvenca se ponavlja za sve karaktere u stringu koji ispisujemo.
- Imamo funkciju *myread_string()*. Opet funkcija kao i svaka druga funkcija, samo što se izvršava u kernel režimu rada procesora. Funkcija ispituje kontrolni registar *Keyboard* uređaja sve dok uređaj ne javi da ima karakter spreman za preuzimanje. Zatim se karakter preuzima i smešta u bafer koji je funkcija dobila kao ulazni parametar. Prethodna sekvenca se ponavlja do detektovanja '\n' karaktera ili do popunjenosti bafera.

Treba primetiti da je sam kod ovde pisan kao i kod korisničkog programa. Štaviše ovaj kod je mogao biti generisan i od strane "C" kompajlera. Kada pišemo kod za operativni sistem, trudimo se da on u najvećoj meri odgovara načinu pisanja korisničkog programa.

Jedini izuzetak je kod samog obrađivača sistemskih poziva, Npr. kod od labele `handler:` pa nadole. Ovaj kod radi istu stvar kao i onaj koji smo videli na prošlim vežbama:

- Čuvanje $at registra
- Provera uzroka i ako nije radi *exit()*. U stvarnosti bismo terminirali proces (kill).
- Ako je u pitanju sistemski poziv 104, tada pozivamo funkciju *myprint_string()*, a zatim nastavljamo sa `restore:` kodom.
- Ako je u pitanju sistemski poziv 108, tada poozivamo funkciju *myread_string()*, a zatim nastavljamo sa `restore:` kodom.
- Pri restore-u vraćamo na zatečenu vrednost registar $at i zatim uvećavamo EPC vrednost da bismo preskočili syscall instrukciju koja nas je dovela u obrađivač sistemskih poziva.
- Resetujemo specijalni registar koji ukazuje na našu aktivnost na obradi sistemskog poziva, i pozivamo `eret` radi povratka na korisniči program.

2.2 Zadatak 1

1. Otvoriti u MARS simulatoru datoteke *mips6syscall_handler.asm* i *mips6use_new_syscalls.asm*
2. Prateći uputstvo iz prethodnih vežbi postavite *mips6syscall_handler.asm* za obrađivač izuzetaka.
3. implementirati nedostajucu funkciju "myread_string" u datoteci *mips6syscall_handler.asm*
4. Pokrenite simulator *Tools -> Keyboard and Display Simulator*. Povežite se iz simulatora na MARS i izvršite reset simulatora.
5. Pokrenuti program *mips6use_new_syscalls.asm* i uveriti se da sistemski pozivi 104 i 108 rade kao i sistemski pozivi 4 i 8. Pre ponovnog pokretanja programa korisno je resetovati *Keyboard and Display Simulator*.

3 Konkurentno izvršenje programa (*multitasking*) i prekljućenje procesora (*context switching*)

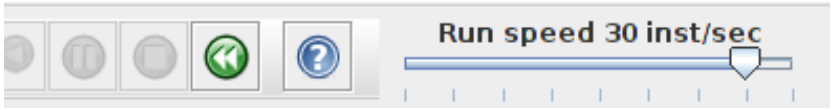
Sada ćemo obraditi prekljućivanje procesora, koje omogućava programima da na smenu koriste CPU za izvršenje svojih instrukcija. Za ovo nam je potrebno:

1. Dva korisnička programa. U ovom primeru će oba biti u istoj datoteci.
2. Izvor periodičnih prekida poznatih kao otkucaji časovnika (*clock ticks)*, koji nam omogućavaju periodični prelaz u kernel režim rada i izazivaju smenu procesa, tj. prekljućenje procesora na drugi program.
3. Obrađivač prekida koji "hvata" prekide izazvane otkucajima časovnika i radi prekljućivanje.

3.1 Zadatak 2

Postavke okruženja:

1. Prateći uputstvo iz prethodnih vežbi postavite *context_switcher.asm* za obrađivač izuzetaka.
2. Otvorite u MARS simulatoru program *mips6multitasking.asm*, i asemblirajte ga.
3. Podesite brzinu izvršavanja na 30 instrukcija u sekundi.



4. Iz MARS simulatora pokrenite *Tools -> Digital Lab Sim* i povežite ga na MIPS. Ovaj uređaj će slati otkucaje časovnika.
5. Na posetku pokrenite program *mips6multitasking.asm*. Trebalo bi da se pojavi ispis sličan ovom:

```
In main1, the counter is 0
In main1, the counter is 1
In main2, the counter is 10000
In main2, the counter is 10002
In main1, the counter is 2
In main1, the counter is 3
In main1, the counter is
In main2, the counter is 10004
In main2, the counter is 10006
In main2, the counter is 100084
In main1, the counter is 5
In main1, the counter is 6
In main2, the counter is 10010
In main2, the counter is 10012
```

3.2 Analiza datoteka iz 2. zadatka

Analiziraćemo obe datoteke iz prethodnog zadatka. Program u *mips6multitasking.asm* datoteci izgleda kao i svaki drugi program i ima tri funkcije: *main()*, *main1()* i *main2()*.

main1() i *main2()* su jednostavne. Svaka ispisuje vrednost brojača, uvećava ga i to radi u beskonačnoj petlji. Primetite i to da *main1()* nikada ne poziva *main2()*i obrnuto. Međutim, iako ne postoji kod u ovom fajlu koji to radi, iz ispisa programa je očito da se CPU prebacuje sa jednog programa na drugi!

Šta više, funkcija *main()* vrši sistemski poziv 100 i zatim izlazi sa *exit()* . Kako onda CPU pokreće funkcije *main1()* i *main2()* , ako nema grananja niti skoka iz funkcije *main()*?

Odgovor je, naravno, taj da smo započeli prekljućenje procesora između dva programa. Ovo zahteva periodične prekide časovnika poznate kao *clock tick*s. Prekidi časovnika se obrađuju u obrađivaču prekida, koji sačuva stanje programa, učita stanje drugog programa, i vrati se izvršenju **drugog** programa. Znači ne vraćamo se programu koji je prekinut!

3.3 Obrađivač sistemskog poziva 100

Sada ćemo analizirati kod iz datoteke *context_switcher.asm*.

Da bismo sačuvali stanje svakog programa treba nam PCB blok (Process Control Block) za svaki program. PCB ima dovoljno prostora da sačuva sve registre koje program koristi. U pravom operativnom sistemu bi PCB za MIPS bio veličine 32 reči, kako bi se sačuvala sva 32 registra. Mi ćemo čuvati samo 5 registara:

```
# Definisemo process control block za svaki program
        .kdata
p1pcb:  .space 20       # Prostor za a0, v0, t0, t1 i PC (ofseti 0, 4, 8, 12, 16)
p2pcb:  .space 20       # Isto to
```

Na početku asemblerskog koda u ovoj datoteci se nalazi generički *front-end* kod koji razlučuje da li se radi o sistemskom pozivu ili prekidu.

Obrađivač sistemskih poziva (*syscallhdlr:*) vrši proveru da li je upućen sistemski poziv 100 i samo se taj poziv obrađuje. Korisnički program je predao adrese prvih instrukcija za oba programa između kojih želimo da se vrši preključivanje procesora. Obrađivač sistemskih poziva samo popuni inicijalne vrednosti za sve registre u PCB blokove svakog programa. Program *main()* je prilikom sistemskog poziva 100 prosledio adrese funkcija *main1()* i *main2()*, tako da su PCB blokovi inicijalno:

Program	$a0	$v0	$t0	$t1	Program Counter
1	0	0	0	0	*main1()*'s address
2	0	0	0	0	*main2()*'s address

Kada se prethodno opisani deo koda izvrši, tada obrađivač omogućava prekide časovnika koje šalje *Digital Lab Sim* program, a zatim skače na kod koji obnavlja sadržaje registara iz PCB bloka i pokreće jedan od dva programa. Pokretanjem jednog od programa sistemski poziv se nikada ne vrati u funkciju *main()*!

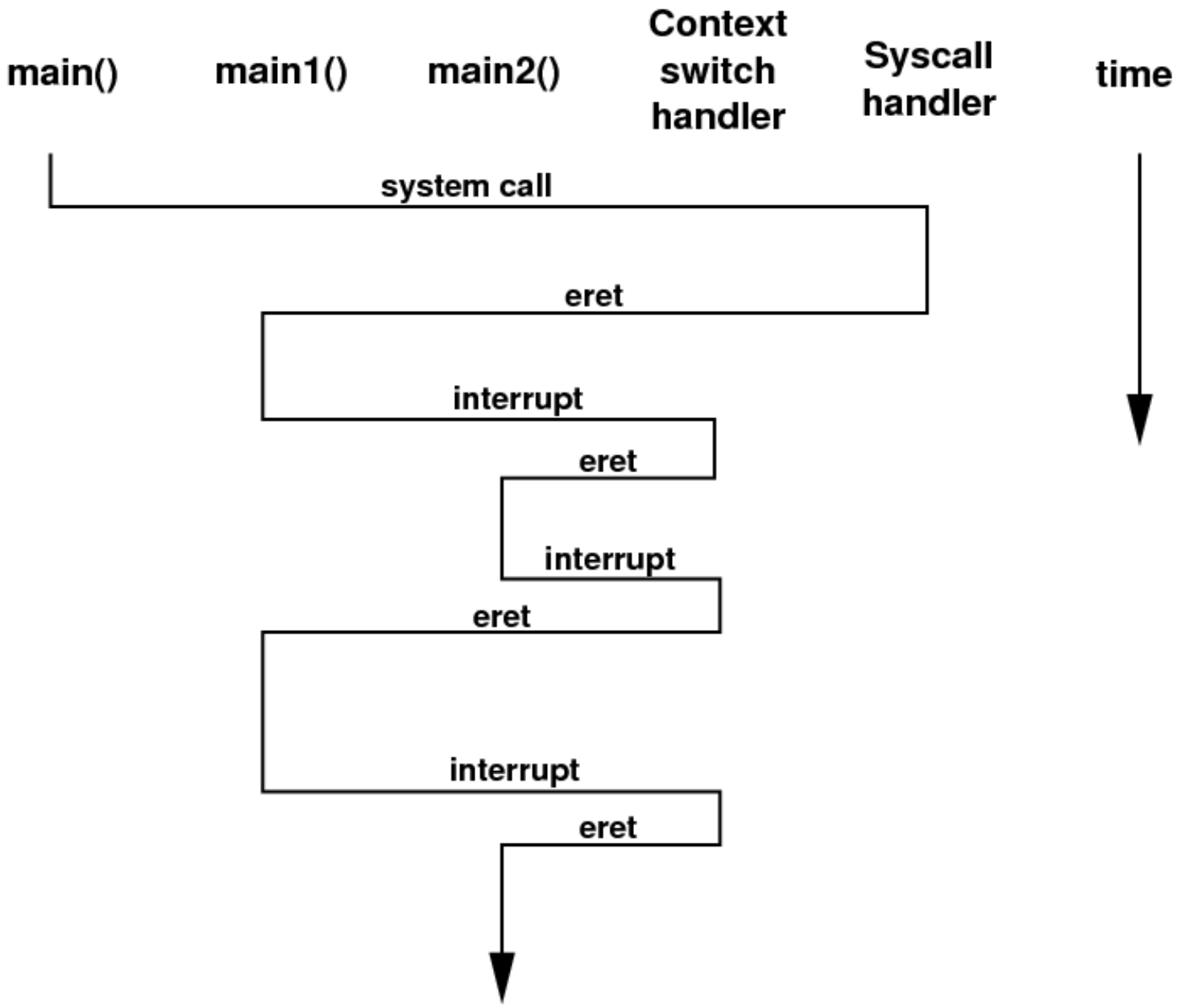
3.4 Obrađivač prekida izazvanih otkucajima časovnika

Obrađivač prekida ima 4 osnovna zadatka:

1. Sačuvati sve registre u tekući PCB blok.
2. Prebaciti se na drugi PCB blok.
3. Sve registre CPU-a popuniti vrednostima iz novog PCB bloka.
4. Izvršiti *eret* instrukciju, koja nas vraća iz kernel režima rada u korisnički režim rada. Povratak se vrši na instrukciju koja je bila na redu da se izvrši u trenutku nastanka prekida.

Nisu navedeni još neki koraci koje je potrebno obaviti, kao što je resetovanje nekih registara koprocesora0 i ponovno omogućavanje prekida, koji su postali onemogućeni prilikom pokretanja obrađivača.

Grafički se angažovanje CPU-a u vremenu može predstaviti sledećim diagramom::



3.5 Zadatak 3

Prođite kroz programski kod obe datoteke i uparite asemblerske instrukcije sa prethodnim objašnjenjem.

Pitanje koje se postavlja je: Da li preključenje procesora oduzima procesorsko vreme korisničkim programima?

Odgovor je naravno potvrdan! Oko 30 instrukcija se obradi u procesu preključivanja, a neke od njih su i pseudo instrukcije (znači oko 40 pravih instrukcija). Recimo da program, u proseku, uspe da izvrši oko 60 instrukcija posle instrukcije *eret* , a pre nego se desi prekid časovnika. Prema tome, od 100 instrukcija 60 instrukcija su korisne, a 40 instrukcija je potrošeno na preključenje procesora. Da bi se ovaj odnos popravio u korist korisničkog programa, potrebno je da interval otkucaja časovnika bude značajno duži u odnosu na vreme koje je potrebno da se preključenje izvrši.

3.6 Zadatak 4

Pogledajte izlaz iz ova dva programa. Možete li da objasnite zašto na nekim linijama ispisa postoje ispisi iz oba programa? Npr:

```
In main2, the counter is 10006
In main2, the counter is 100084
In main1, the counter is 5
```

3.7 Zadatak 5

Pokrenite program iz početka i pratite kako se procesor preključuje sa jednog procesa na drugi. Dok je izvršenje u toku, klikom na dugme "Disconnect from MIPS" na *Digital Lab Sim* simulatoru. Kada ovo uradite tada ste ukinuli izazivača preključenja procesora. To znači da će se proces koji se tenutno izvršava nastaviti da se izvršava u nedogled.

Pokrenite ponovo prekide časovnika klikom na dugme "Connect to MIPS" na *Digital Lab Sim* simulatoru. Program bi sada ponovo trebao da počne sa preključenjem između dva procesa.