7. Okruženje procesa

7.2 main funkcija

Kada kernel izvršava C program uz pomoć neke od *exec* funkcija, specijalna start-up rutina se poziva pre poziva main funkcije. Izvršni program specifira ovu rutinu kao početnu adresu programa; ovo se postavlja od strane link editora kada ga pozove C kompajler. Ova start-up rutina uzima vrednosti od kernela, argumente komadne linije i okruženja, vrši neke neophodne inicijalizacije i poziva main funkciju.

7.3 Prekid procesa

Postoji 8 načina na koje se može prekinuti proces. Normalan prekid se dešava u 5 slučajeva:

1. povratak iz main-a
2. pozivanje exit
3. pozivanje _exit ili _Exit
4. povratak poslednje niti iz njene početne rutine
5. pozivanjem pthread_exit iz poslednje niti

Neplanirani prekid se desava u 3 slućajeva:

6. pozivom abort
7. prijemom signala
8. odgovorom poslednje niti na zahtev za prekid

Start-up rutina je napisana tako da ukoliko se završi main funkcija, poziva se exit funkcija. Ukoliko su start-up rutine kodirane u C-u, poziv main funkcije bi mogao da izgleda ovako:

```
exit( main( argc,argv));
```

Exit funkcije

Pomoću 3 funkcije se proces normalno prekida: **_exit** i **_Exit**, kojima se vraćamo u kernel trenutno (ne prazne se baferi dodeljeni tokovima), i ***exit***, koja obavlja izvesno čiscenje i onda se kontrola predaje kernelu (prazne se i baferi). Funkcija *exit* je obavljala neke funkcije za "čisto" završavanje funkcija standardne I/O biblioteke: *fclose* funkcija se poziva za sve otvorene tokove (bar za 3 koja su uvek otvorena). Sve 3 exit funkcije očekuju celobrojni argument, koji nazivamo **exit status**. Unix sistemi obezbeđuju način da se ispita exit status procesa.

exit(0); <=> return 0;

(PRIMER – hello world)

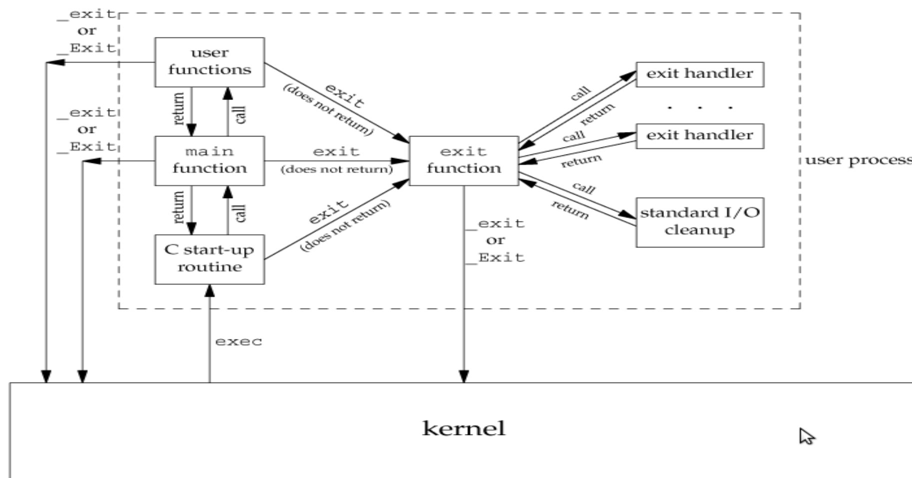
echo $? - dobijanje exit statusa poslednje pokrenutog programa

```
int atexit(void (*func)(void));
```

Proces može registrovati do 32 funkcije koje se automatski pozivaju od strane *exit*. Ove funkcije se nazivaju **exit handlers** i registruju se pozivom **atexit** funkcije. Ova funkcija kao argument prima adresu funkcije. Kada se ta funkcija pozove ona ne prima nikakve argumente i ne vraća nikakve vrednosti. Funkcija exit poziva ove funkcije u obrnutom redosledu od registrovanja, i svaka funkcija se

poziva onoliko puta koliko je registrovana. *exit* prvo poziva exit handlere i potom zatvara sve otvorene tokove.

Ovo je korisno za velike programe, jer da ne postoji *atexit* funkcija, morali bi pre svakog poziva exit da pozivamo odgovarajuću našu funkciju. Ovaj koncept je povezan sa programiranjem za kreiranje GUI-a. Registrujemo neke funkcije koje će se izvršavati kada se desi neki događaj (klikne se mišem, pritisne taster na tastaturi, itd.).



Program se izvršava pozivom neke od *exec* funkcija od strane kernela. Jedini način da se proces "dobrovoljno" završi jeste pozivom *_exit* ili *_Exit* funkcije, bilo eksplicitno ili implicitno (pozivom *exit* funkcije).

(PRIMER atexit)

7.4 Argumenti komandne linije

Kada se program izvršava, proces koji pokreće exec funkciju može da prosledi argumente komandne linije novom programu. Ovako funkcioniše shell.

argv[argc] je uvek null pointer, tako da se petlja može napisati kao:

```
for (i=0; argv[i] != NULL; i++)
```

7.6 Memorijska pozadina C programa

C program se sastoji iz sledećih delova:

- **tekst segment** – mašinske instrukcije koje procesor izvršava. Obično se ovaj deo često izvršavanih programa kao što su tekstualni editori, C kompajler, shell nalazi u memoriji. U ovom segmentu je obično dozvoljeno samo čitanje, da se ne bi desilo da program slučajno modifikuje instrukcije.
- Inicijalizovani segment podataka, obično samo **segment podataka**, sadrži promenljive koje su inicijalizovane u programu, npr: *int i = 0;* a da se pritom inicijlizacija nalazi van funkcija
- Neinicijalizovani segment podataka, obično se naziva **bss** segment (dobio ime po asemblerskom operatoru "block started by symbol"). Podaci u ovom segmentu su inicijalizovani na 0 ili na null pokazivače pre izvršavanja programa. C deklaracija koja se nalazi van funkcija: *int s[10];*

smešta podatke u ovaj segment.

- **Stek**, gde se smeštaju automatske promenljive, zajedno sa informacijama koje se čuvaju svaki put kada se pozove funkcija. Svaki put kada se funkcija pozove, adresa na koju treba da se izvrši povratak i određene informacije o okruženju pozivaoca (npr. neki mašinski registri) se čuvaju na steku. Nova funkcija zatim na steku alocira prostor za njene automatske i privremene promenljive. Ovako funkcionišu rekurzivne funkcije u C-u. Svaki put kada rekurzivna funkcija pozove samu sebe, novi **stek okvir** (stack frame) se koristi, tako da se skupovi promenljivih različitih instanci funkcije ne mešaju.
- **Hip**, u kome se vrši dinamička alokacija memorije.

SLIKA 7.6

Komanda **size** daje veličinu u bajtovima, za tekstualni, bss i segment podataka. Probati npr: *size atexit.* Zatim probati dodavanje po jedne inicijalizovane i neinicijalizovane globalne promenljive u atexit.c i posle kompilacije ponovo pokrenuti gornju komandu.

```
readelf -a program
```

štampa razne informacije o tom fajlu. Kada se pokrene program, on vidi neki virtuelni memorijski adresni prostor koji se na 32bitnim računarima prostire od adrese 0 do 2^32-1. Za većinu adresa ne postoje fizičke adrese koje stoje iza njih. Kada se program pokrene, kernel čita sekcije iz elf fajla i raspoređuje ih u okviru tog adresnog prostora.

Ukoliko kod sadrži liniju

```
char *k; *k='3';
```

često (ne uvek) se javlja segmentation fault jer je vrlo verovatno da pokušavamo da pristupimo nekoj virtuelnoj adresi kojoj ne odgovara neka fizička. Većina procesora ima *memory managment unit*, jedinicu koja služi za mapiranje, prepoznaje da tražimo nepostojeću adresu, obaveštava o.s., o.s. obaveštava naš program i šalje signal. Podrazumevana akcija je prekid programa.

U izvršnoj verziji programa na disku se čuvaju samo tekst segment i inicijalizovani segment podataka. Probati sa dodavanjem različitih globalnih promenljivih i pokretanjem *stat* komande na izvršnim verzijama programa kako bi se videla razlika u veličini (obično se samo dodavanjem prve promenljive u neinicijalizovani segment podataka se povećava veličina fajla – da bi se uradile neke inicijalizacije).

7.8 Alokacija memorije

```
void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);

void free(void *ptr);
```

Pokazivač koji vraćaju ove tri funkcije može se koristiti za bilo koji tip podataka. Ukoliko uključimo zaglavlje stdlib.h, ne moramo eksplicitno da kastujemo pokazivač koji vraćaju ove funkcije. Ukoliko postoji prostor iza već obezbeđene memorije onda *realloc* ne mora nista da pomera, jednostavno alocira memoriju i vraća isti pokazivač koji smo mu preneli. Ali ako ne postoji dovoljno prostora, onda *realloc* alocira novu memoriju, kopira već postojeće podatke i vraća novi pokazivač.

Pošto se ovaj prostor može pomeriti, ne bi trebali da imamo nikakve pokazivače u ovom prostoru. Poslednji argument je veličina novog prostora, a ne razlika novog i starog. Ukoliko je prvi argument ove funkcije null pokazivač, onda se ona ponaša kao *malloc*.

Ove funkcije su obično implementirane pomoću **sbrk** sistemskog poziva. On povećava ili smanjuje hip procesa. Obično se prostor koji se oslobodi ne vraća kernelu, vec se čuva u **malloc skladištu** za kasniju alokaciju.

Obično se za svaki alocirani prostor čuva nešto više podataka, kao što su veličina alociranog bloka, pokazivač na sledeći alocirani blok itd. To znači da se pisanjem posle alocirane memorije mogu presnimiti informacije koje se odnose na naredni blok. Ovo je vrlo opasna greška, i teška za pronalaženje, jer se može pokazati tek dosta kasnije.

Greška je i oslobađanje memorije koja je već oslobođena ili pozivanje *free* za pokazivač koji se nije dobio pozivanjem jedne od prethodne 3 funkcije. Ukoliko proces poziva *malloc* ali ne dealocira memoriju, memorijsko zauzeće se postepeno povećava. Ovo se zove **curenje memorije**. Danas se na većini sistema oslobađa dinamički alocirana memorija po završetku programa, ali ovo nije obuhvaćeno standardom i ne postoji garancija da će se oslobađanje izvršiti. Zato je potrebno da programer obezbedi oslobađanje ove memorije (i u slučaju greške).

7.5 Environment list

Svakom procesu se prosleđuje i EL. Kao i lista argumenata, EL je niz pokazivača na niske, tako da svaki pokazivač sadrži adresu stringa koji se zavrsava sa \0. Adresa niza pokazivača se čuva u globalnoj promenljivoj **environ**:

extern char **environ;

Npr, ukoliko E čine 5 stringova, environ može izgledati kao na SLICI 7.5 - APUE.
environ ćemo zvati E pokazivačem, niz pokazivača EL, a stringove na koje pokazuju E stringovi.
E se sastoji od stringova oblika

name = value

Većina UNIX sistema obezbeđuje 3. argument main funkciji – adresu EL.

int main(int argc, char *argv[], char *envp[]);

POSIX nalaže da pre treba koristiti environ nego treći argument main funkcije.

7.9 Environment promenljive

To su u stvari neka podešavanja za okruženje u kome se pokreće proces.
E stringovi su oblika:

name = value

Neke od ovih promenljivih se automatski postavljaju pri logovanju, kao HOME i USER. Možemo postaviti ove promenljive u shell start-up fajlu. Ukoliko postavimo promenljivu MAILPATH, onda shell zna gde treba da proveri mail.

```
char *getenv(const char *name);
```

Ova funkcija nam omogućava da pročitimao E promenljive. Vraća string oblika name=value.
TABELA
Pored dobijanja vrednosti ovih promenljivih, ponekad želimo da postavimo neku promenljivu ili da dodamo novu. To možemo da uradimo samo za trenutni proces i za decu procese koje pokrećemo, dok ne možemo za roditeljski proces – najčešće shell.

```
int putenv(char *str);

int setenv(const char *name, const char *value,int rewrite);

int unsetenv(const char *name);
```

putenv smešta string tipa name=value u E listu. Ukoliko name već postoji, prvo se uklanja prethodna definicija. Kasnije se došlo na ideju da se napravi funkcija koja je konfigurabilna.
setenv postavlja *name* na *value*. U zavisnosti od toga koja je vrednost ***rewrite*** vrši se zamena ili ne kada name već postoji u listi.
unsetenv briše definiciju name.

Razlika između prve dve funkcije je da setenv mora prvo da alocira memoriju. Često nije jednostavno dodati novu promenljivu, jer se promenljive nalaze na vrhu memorijskog prostora procesa, iznad steka. Ukoliko je potrebno povećati memoriju to se najćešće radi prenošenjem E promenljivih na hip.

7.11 getrlimit and setrlimit Functions

Ovo su vrlo korisne funkcije, jer se pomoću njih može pročitati i podesiti količina resursa koja je dostupna procesu.

Zadatak 1. Napisati program *three_newest_files.c* koji pronalazi 3 najskorije menjana fajla u direktorijumu datim prvim argumentom komandne linije. Pretraga treba da se vrši i u svim njegovim poddirektorijumima. Štampati ime svakog takvog fajla i vreme poslednje promene. Nije dozvoljeno korišćenje funkcija *ftw* i *nftw*. Ne postoji pretpostavka o dužini putanje fajla!

Zadatak 2. Napisati program *size_of_folder.c* koji izračunava zbir veličina svih fajlova u direktorijumu datim prvim argumentom komandne linije. Potrebno je koristiti funkciju *lseek* za dobijanje veličine fajla. Pretraga treba da se vrši i u svim njegovim poddirektorijumima. Nije dozvoljeno korišćenje funkcija *ftw* i *nftw*. Ne postoji pretpostavka o dužini putanje fajla!

Zadatak 3. Napisati program *environment_vars.c* koji štampa sve environment promenljive koje mu se prosleđuju.

Zadatak 4. Napisati program koji proverava da li se pri pokretanju bilo kog programa na sistemu proverava da li se taj program nalazi u tekućem direktorijumu (.). Ovo uraditi koristeći sadržaj environment promenljive PATH (tj., proveriti da li se '.' nalazi u promenljivoj PATH). Ako tekući direktorijum nije u PATH-u, pronaći pomoć Web-a način da se ubaci, i potom ponovo testirati program.