14.2 Neblokirajući Ulaz/Izlaz

Sistemski pozivi se dele u dve kategorije: spore i oni koji to nisu. Spori sistemski pozivi su oni koji mogu večno da budu blokirani. U njih spadaju:

- read pozivi koji čekaju podatke koji nikada nisu dostupni
- write pozivi koji ne mogu da upišu podatke iz nekog razloga (npr., nedostatak prostora)
- open pozivi koji blokiraju dok ne dođe do promene nekog stanja, itd.

Neblokirajući U/I omogućava nam da izvršimo U/I operaciju, npr. read, write ili open, a da ona ne bude večno blokirana. Ukoliko se operacija ne može izvršiti, funkcija se vraća sa porukom o greški.

Postoje 2 načina da se za dati deskriptor zada neblokirajući U/I.

1. Ako pozivamo open da dobijemo deskriptor, možemo da navedemo O_NONBLOCK fleg.
2. Za deskriptor koji je već otvoren možemo da pozovemo **fcntl** funkciju da bi uključili **O_NONBLOCK** fleg.

(PRIMER – nonblocking)
./nonblocking < ulaz.txt > izlaz.txt
ls -l izlaz.txt

./nonblocking < ulaz.txt 2>izlaz_za_greske.txt
less izlaz_za_greske.txt

Napomena: generisati fajl ulaz.txt da bude dovoljno veliki.

Pajpovi, FIFOi i neki uređaji (terminali, mreže) imaju sledeće dve osobine:

- read operacija može vratiti manje nego što je traženo, iako nismo stigli do kraja fajla. Ovo nije greška i u tom slučaju treba nastaviti sa čitanjem.
- write operacija takođe može vratiti manje nego što smo tražili. Ovo nije greška i možemo dalje da nastavimo sa pisanjem.

Ovo se ne dešava kada pišemo po fajlu na disku (izuzev kada nestane slobodnog prostora). Kada čitamo iz ili pišemo u pajp, mrežni uređaj ili terminal, moramo prethodne dve osobine uzeti u obzir.

14.3 Zaključavanje zapisa

Šta se dešava kada dva korisnika u isto vreme menjaju isti fajl? U UNIX-u će završno stanje odgovarati zadnjem procesu koji je menjao fajl. U bazama podataka proces mora biti siguran da u trenutku samo on piše u fajl. UNIX obezbeđuje mehanizam koji se zove z.z.

Z.z. označava mogućnost da proces A zaključa deo fajla kako ostali procesi ne bi mogli da promene taj deo dok proces A čita ili piše po njemu.

```
int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */ );
```

Argument *cmd* je F_GETLK, F_SETLK ili F_SETLKW.

```
struct flock {
```

```
    short l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
    off_t l_start;  /* offset in bytes, relative to l_whence */
    short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
    off_t l_len;    /* length, in bytes; 0 means lock to EOF */
    pid_t l_pid;    /* returned with F_GETLK */
  };
```

F_RDLCK – deljeni katanac za čitanje
F_WRLCK – ekskluzivni katanac za pisanje
F_UNLCK – otključavanje dela fajla
`l_pid` – ID procesa koji drži katanac koji može da blokira trenutni proces

Ako je `l_len` 0, to označava da se katanac proširuje do najvećeg mogućeg ofseta u fajlu. Ovo omogućava da zaključamo sve ono što je dodato na kraj fajla (a da ne pretpostavljamo koliko je bajtova dodato).

SLIKA 14.3
Ova pravila se odnose na više procesa, sam proces može promeniti regiju na kojoj je postavio katanac ako njen deo ne drže drugi procesi. Takođe, ukoliko već poseduje katanac za pisanje na nekom prostoru, proces može postaviti katanac za čitanje umesto prethodnog.

Da bi se dobio katanac za čitanje (pisanje), deskriptor mora biti otvoren za čitanje (pisanje).

F_GETLK – proverava da li prostor na koji se odnosi *flockptr* zauzet od strane nekog drugog procesa. Ukoliko postoji katanac koji nas sprečava da mi postavimo svoj, onda se informacije o tom katancu smeštaju u prostor na koji pokazuje *flockptr*. Ukoliko ne postoji katanac koji sprečava da postavimo svoj, struktura ostaje neizmenjena izuzev polja `l_type` koji se postavlja na F_UNLCK.
F_SETLK – Ukoliko pokušavamo da dobijemo katanac, i to ne uspeva, fcntl funkcija se vraća i postalja vrednost errno na EACCES i EAGAIN. Ova opcija se koristi i za oslobađanje katanca opisanog u *flockptr* (l_type je F_UNCLK).
F_SETLKW – blokirajuća verzija prethodne opcije. Proces se uspavljuje dok traženi resursi ne budu dostupni.

Testiranje da li je neki deo memorije slobodan i pokušaj da se dobije katanac nad tim prostorom NIJE atomična operacija.

(PRIMER locking)

Nasleđivanje i oslobađanje katanaca:

1) Kada se proces završi, svi katanci se oslobađaju. Kada se zatvori deskriptor svi katanci na fajlu sa kojim je povezan taj deskriptor se zatvaraju.
2) Katanci se nikad ne nasleđuju od strane deteta po pozivu fork funkcije. Ovo ima smisla jer katanci služe da onemoguće da dva procesa istovremeno menjaju fajl, što bi se inače desilo.
3) Katanci se nasleđuju od strane novog procesa po pozivu exec funkcije. Ukoliko je pak postavljen close-on-exec fleg, svi katanci se oslobađaju kada se deskriptor završi kao deo exec.

Postavljanje close-on-exec flega:

```
    if ((flags = fcntl (file, F_GETFD, 0)) < 0)
```

```
        error_fatal (argv[0]);
    flags |= FD_CLOEXEC;
    if (fcntl (file, F_SETFD, flags) < 0)
        error_fatal (argv[0]);
```

Obavezno naspram savetodavnog zaključavanja: Prethodno navedeno zaključavanje je **savetodavno**, tj. ako jedan proces drži katanac za pisanje na delu nekog fajla, drugi proces ne može dobiti katanac za pisanje na tom delu. Međutim, *drugi proces može pisati po zaključanom delu*!!!! Ovo zaključavanje ima smisla kada se radi o funkcijama koje npr. naizmenično pristupaju nekoj bazi, i samo one imaju pristup. Svaka izvodi odgovarajuću operaciju tek kad ima katanac na tom prostoru. Ako pak mnogo ljudi ima pristup istim podacima, onda je potrebno koristiti **obavezno** zaključavanje. U tom slučaju, kada je neki proces zaključao prostor, drugi prosesi ne mogu ni čitati ni pisati sa tog prostora. Obavezno zaključavanje prevazilazi okvire ovog kursa.

14.5 U/I multipleksing

```
while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        errror_fatal ("write error");
```

Na ovaj način možemo blokirati ulaz. Šta ukoliko imamo 2 fajl deskriptora? U ovom slučaju ne možemo da blokiramo čitanje na jednom deskriptoru jer se podaci mogu pojaviti na drugom.

Primer telneta : ima dva ulaza pa ne možemo da blokiramo jedan ulaz jer podaci mogu stići i sa drugog.

1. način rešavanja ovog problema je da se proces podeli u dva dela koristeći fork, i da svaka polovina obrađuje polovinu podataka. Ovako svaki proces može imati blokirajući read. Ovaj način ima nedostatak da se usložnjava kod, potrebno je sinhronizovati roditelja i dete (najčešće signalima).

2. način je da koristimo dve niti u okviru istog procesa. Ovo takođe povećava kompleksnost programa.

3. način je da oba deskriptora postavimo da budu neblokirajući, i da naizmenično proveravamo da li su na njima prisutni podaci. Ovo se zove *polling*. Nedostatak je što se troši procesorsko vreme.

4. način je korišćenjem asinhronog U/I. Tražimo od kernela da nas obavesti signalom kada je deskriptor spreman za korišćenje. Dva problema se ovde javljaju: prvi je da ne podržavaju svi sistemi ovu opciju, a drugi da kada dođe do signala moramo proveriti kod oba deskriptora koji je spreman.

5. način je U/I multipleksing. Pravimo listu deskriptora za koje smo zainteresovani, i pozivamo funkciju koja se ne vraća sve dok jedan od deskriptora nije spreman za U/I. Po povratku iz funkcije

znamo koji deskriptori su spremni za U/I.

3 funkcije koje nam ovo omogućavaju su **poll**,**pselect** i **select**.

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Select prima sledeće argumente:

- deskriptore za koje smo zainteresovani
- za šta smo zainteresovani kod kog deskriptora (čitanje, pisanje, pojavu izuzetka)
- koliko dugo hoćemo da čekamo

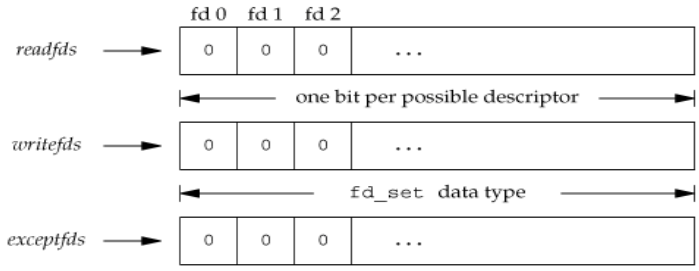
Pri povratku iz select kernel nam prenosi:

- broj deskriptora koji su spremni
- koji deskriptori su spremni za koji od 3 uslova (čitanje, pisanje, pojavu izuzetka)

Za deskriptor se kaže da je spreman kada nije blokiran: to je slučaj kada se na tom deskriptoru nalaze dostupni podaci ili kada se naiđe na EOF. Sa ovim informacijama možemo pozvati odgovarajuću U/I funkciju znajući da funkcija neće da blokira.

Ako je poslednji argument NULL to označava beskonačno čekanje. Prekida se oslobađanjem deskriptora ili pojavom signala. Ukoliko su obe vrednosti u strukturi timeval 0, uopšte ne čekati. Inače čekati naznačeni broj sekundi i mikrosekundi. Ukoliko naznačeno vreme istekne, funkcija vraća 0.

Središnji argumenti označavaju za koje smo deskriptore zainteresovani i za koje uslove. Ovo se može posmatrati kao niz bitova.



Sa tipom podataka *fd_set* možemo da radimo sa nekom od 4 funkcije (mogu biti implementirane i kao makroi). FD_ISSET, FD_CLR, FD_SET i FD_ZERO.

```
void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

Bilo koji od tri argumenta funkcije select može biti null pokazivač ako nismo zainteresovani za to stanje. Ako su sva tri postavljena na 0, onda imamo tajmer veće preciznosti od sleep funkcije.

Prvi argument select funkcije je najveći mogući fajl deskriptor na koji čekamo uvećan za 1. Stavljajući ovu vrednost sprečavamo kernel da prolazi i proverava hiljade neiskorišćenih bitova u 3 skupa.
Postoje 3 povratne vrednosti iz select:

1) -1 označava da je došlo do greške (uhvaćen signal pre nego sto je bilo koji od deskriptora spreman)
2) 0 označava da nema spremnih deskriptora. Tada se svi deskriptori koji su postavljeni menjaju na 0.
3) Pozitivna povratna vrednost označava broj spremnih deskriptora. Ovo je suma broja spremnih u svim skupovima, pa ako je neki postavljen za čitanje i pisanje, računa ce se dva puta pri povratku. Jedini bitovi koji će biti postavljeni u 3 skupa su oni koji su dostupni.

Kada je deskriptor u skupu izuzetaka dostupan? Ukoliko stanje izuzetka čeka na deskriptoru.

Važno je shvatiti da bez obzira da li deskriptor blokira ili ne, ne utiče na blokiranje select. Ako imamo neblokirajući deskriptor iz koga želimo da čitamo, i pozovemo select koji čeka 5 sekundi, onda će select blokirati na 5 sekundi.

(PRIMER select)
Funkcija *pipe* u ovom primeru otvara deskriptore filedes[0] i filedes[1] redom za čitanje i pisanje, tako da ono što bude upisano na filedes[1] može biti pročitano sa filedes[0]. Ako prethodno nismo otvarali/zatvarali deskriptore ove promenljive će dobiti vrednosti 3 i 4.

14.7 readv i writev funkcije

Omogućavaju čitanje iz više nepovezanih bafera u istom funkcijskom pozivu.

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);

struct iovec {
     void   *iov_base;   /* starting address of buffer */
     size_t  iov_len;    /* size of buffer */
   };
```

Writev funkcija skuplja podatke za izlaz redom iz bafera iov[0], iov[1] do iov[iovcnt]. Ova funkcija vraća ukupan broj bajtova koji se upisuju, sto bi trebalo da bude zbir svih veličina bafera.

Readv upisuje podatke u bafere redom, uvek puneći jedan bafer pre nego što pređe na sledeći. Vraća ukupan broj pročitanih bajtova. vraća se 0 ukoliko nema više podataka ili se stiglo do EOF.

Zadatak 1. Napisati program koji kreira dete proces i demonstrira komunikaciju između roditeljskog procesa i deteta procesa pomoću *pipe* sistemskog poziva. U roditelju se učitava jedno slovo sa standardnog ulaza. Ideja je da roditelj ne troši procesorsko vreme već da detetu šalje slovo, i dete

izvršava komandu (za slovo *d* izvršava naredbu *date* a za slovo *u* komadnu *users*, ostala slova se zanemaruju). Komande se mogu unositi i više puta. Jedan primer pokretanja programa je:

./komunikacija
d
Sat Jun 4 10:40:12 CEST 2011
u
andjelkaz gordana mirkos
d
Sat Jun 4 10:40:18 CEST 2011
K
No such command

Zadatak 2. Napisati program koji u datoteci koja se navodi kao prvi argument komandne linije zamenjuje sva pojavljivanja reči koja se navodi kao drugi argumenat komandne linije rečju koja se navodi kao treći argumenat komandne linije. Pretpostavka je da te dve reči imaju jednak broj karaktera. Svaki put kada se dođe do reči koju treba zameniti, postavlja se katanac za pisanje na tu reč, zamenjuje se novom rečju i pušta katanac. Proveriti da li je zaključavanje uspelo tako što se proces pri svakom upisu uspavljuje na 5 sekundi, i tada pokrenuti novu instancu programa. Ukoliko je saržaj fajla *1 2 3 1 2 3*, i program se pokrene sa

./zameni_reci fajl.txt 1 4&

a potom i sa

./zameni_reci fajl.txt 2 5&

sadržaj fajla *fajl.txt* po završetku treba biti *4 5 3 4 5 3* (jednostavnosti radi u primeru su reči jednocifreni brojevi).