Vežba 7 : Napredni kernel objekti i međuprocesna komunikacija

1. Uvod

Na prošlim vežbama su opisani osnovni konstrukti pri dizajniranju drajvera za sekvencijalne uređaje kroz primer jednostavnog *storage* drajvera. Na ovim vežbama će se posvetiti pažnja modifikovanju drajvera za sekvencijalne uređaje, kako bi podržavali izvršavanje u modernim višeprocesnim operativnim sistemima koji potencijalno rade na procesorima sa više jezgara. Prvo će se posetiti pojam procesa i njihova reprezentacija u Linux operativnim sistemima, a zatim rukovanje procesima (blokiranje). Navešćemo najvažnije kernel objekte za interprocesnu komunikaciju, sinhronizaciju i razrešavanje hazarda koji se mogu pojaviti na modernim sistemima.
Za demonstraciju hazarda, kao i njihovih rešenja će biti korišten jednostavan drajver koji simulira rad LIFO (last in first out) registra. Ovaj drajver se sastoji od globalnog statičkog niza sa deset elemenata i globalnog indikatora pozicije pos.

int lifo[10];
int pos=0;

Promenljiva *pos* indeksira poziciju u nizu u koju je potrebno upisati sledeći podatak. Inicijalno je jednaka nuli te ukoliko se upiše vrednost u drajver pozivom komande ***echo ’3’ > /dev/lifo*** , na poziciju lifo[0] se upisuje vrednost 3, te se promenljiva pos inkrementira. Na ovaj način promenljiva pos je jednaka broju vrednosti koje se trenutno nalaze u nizu: 0 - lifo je prazan, 10 - lifo je pun. Pozivom komande ***cat /dev/lifo*** se *pos* dekrementira, te se u terminalu ispisuje poslednji podatak koji je upisan.

Implementacija lifo_write funkcije je data u nastavku:

```
ssize_t lifo_write(struct file *pfile, const char __user *buffer, size_t length, loff_t *offset)
{
	char buff[BUFF_SIZE];
	int value;
	int ret;

	ret = copy_from_user(buff, buffer, length);
	if(ret)
		return -EFAULT;
	buff[length-1] = '\0';

	if(pos<10)
```

```
	{
		ret = sscanf(buff,"%d",&value);
		if(ret==1)//one parameter parsed in sscanf
		{
			lifo[pos] = value;
			printk(KERN_INFO "Succesfully wrote value %d", value);
			pos++;
		}
		else
		{
			printk(KERN_WARNING "Wrong command format\n");
		}
	}
	else
	{
		printk(KERN_WARNING "Lifo is full\n");
	}

	return length;
}
```

Nakon kopiranja stringa iz korisničkog prostora u lokalni niz *buff*, proverava se da li u lifo baferu postoji mesta za poslati podatak (pos<10). Ukoliko je pun, ispisuje se poruka upozorenja i podatak se odbacuje. U suprotnom, pomoću *sscanf* funkcije se podatak iz stringa smešta u celobrojnu promenljivu *value*. Ukoliko je parsiranje stringa uspešno izvršeno (povratna vrednost *ret* = 1, jedan argument parsiran), možemo smestiti podatak u lifo. Budući da globalna promenljiva *pos* indeksira poziciju u nizu gde se treba smestiti nova vrednost, možemo jednostavno izvršiti liniju ***lifo[pos]=value***. Nakon toga se vrednost pos inkrementira kako bi indeksirala sledeće slobodno mesto u lifo baferu.

Implementacija lifo_read funkcije je data u nastavku:

```
ssize_t lifo_read(struct file *pfile, char __user *buffer, size_t
length, loff_t *offset)
{
	int ret;
	char buff[BUFF_SIZE];
	long int len = 0;
	if (endRead){
		endRead = 0;
		return 0;
	}
```

```
	if(pos > 0)
	{
		pos --;
		len = scnprintf(buff, BUFF_SIZE, "%d ", lifo[pos]);
		ret = copy_to_user(buffer, buff, len);
		if(ret)
			return -EFAULT;
		printk(KERN_INFO "Succesfully read\n");
		endRead = 1;
		return len;
	}
	else
	{
		printk(KERN_WARNING "Lifo is empty\n");
		return 0;
	}
}
```

Čitanje vrednosti iz lifo bafera se izvršava u dve iteracije *lifo_read* funkcije. U prvom pozivu funkcije globalna promenljiva *endRead* je jednaka nuli, te prva uslovna naredba *if(endRead)* neće biti zadovoljena. Sledeća provera ispituje da li lifo bafer ima upisan podatak (*pos>0*). Ukoliko ovaj uslov nije zadovoljen, to znači da je lifo prazan te se ispisuje poruka upozorenja i vraća se vrednost 0. Ukoliko postoje podaci u lifo baferu, promenljiva *pos* se dekrementira kako bi indeksirala poslednje upisani podatak. Pomoću *scanprintf* funkcije se ta celobrojna vrednost konvertuje u string, a zatim se kopira u korisnički prostor kako bi mogla biti pročitana. Promenljiva *endRead* se postavlja na 1 i vraća se dužina stringa *len*. Budući da je vraćena vrednost veća od nule, funkcija će ponovo biti pozvana. U drugoj iteraciji je *endRead* promenljiva jedanaka jedinici, uslov *if(endRead)* je zadovoljen, te se vraća nula. Ovime je čitanje podatka završeno.

2. Procesi

Procesi predstavljaju jedan od najvažnijih koncepata operativnih sistema. Program je niz instrukcija koje ostvaruju neki algoritam. Proces je program u statusu izvršavanja, zajedno sa svim resursima koji su potrebni za rad programa. To praktično znači da je program fajl (datoteka) na disku, a kada se taj fajl učita u memoriju i počne da se izvršava, dobijamo proces.

U operativnom sistemu u svakom trenutku postoji više procesa koji se trebaju izvršavati kako bi sistem funkcionisao kao celina. Koji će se proces izvršavati na procesoru i u kojem trenutku je posao planera (*scheduler*). Za samu zamenu procesa koji se trenutno izvršava (zamena konteksta, *eng. context switch*) je zadužen dispečer (*dispatcher*).

Sem memorije koja je dodeljena nekom procesu (programska, stek, heap), proces sadrži i podatke koji opisuju njegovu aktivnost i koji su neophodni za upravljanje njime. Ove podatke generiše i koristi dispečer, a u literaturi se pominju pod raznim nazivima: kontrolni blok procesa (*Process Control Block, PCB*), vektor stanja ili deskriptor procesa.

Na primer, posmatrajmo višeprocesni operativni sistem sa dva aktivna procesa (P1 i P2). Proces P1 koji se izvršava na procesoru, u jednom trenutku se blokira zbog čekanja da se neki resurs oslobodi. Nakon toga, nastavlja se izvršavanje procesa P2 koji posle nekog vremena takođe biva blokiran. U međuvremenu se odblokira resurs neophodan za izvršavanje procesa P1, pa proces P1 može nastaviti izvršavanje. Da bi OS znao gde treba da nastavi izvršavanje, svakom procesu se dodeljuju prateće informacije, odnosno jedinstven kontrolni blok.

Kontrolni blok je deo radne memorije sa osnovnim informacijama o procesu, koje operativni sistem koristi za upravljanje tim procesom. Zahvaljujući kontrolnom bloku, izvršavanje programa se može prekidati i nastavljati više puta. Informacije koje čine kontrolni blok su:

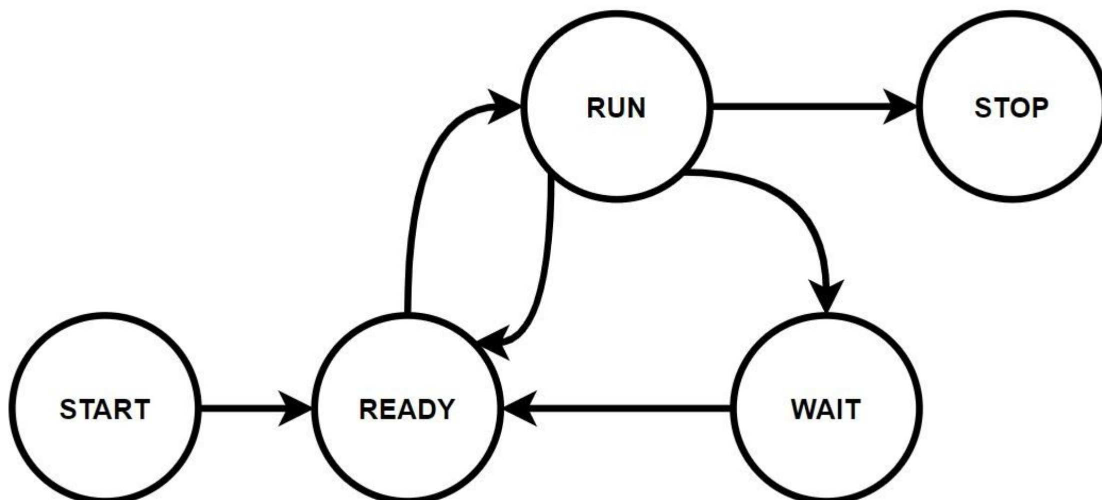
- Jedinstveni identifikator procesa
- Kontekst (okruženje) procesa
- Prioritet procesa
- Informacije o memoriji procesa
- Lista otvorenih datoteka
- Status zauzetih ulazno-izlaznih resursa
- Trenutno stanje procesa.

Kontekst procesa čine podaci koji se čuvaju prilikom oduzimanja procesora, a njih generiše sam hardver. U te podatke spadaju vrednost programskog brojača, vrednost registara i pokazivači na deo memorije koje proces koristi. Deo kontrolnog bloka u kome se čuva kontekst se naziva još i hardverski kontrolni blok ili hardverski deskriptor procesa.

Proces se sastoji od niza instrukcija koje slede jedna za drugom. Između dve instrukcije proces može biti prekinut, a njegovo izvršavanje se može nastaviti u drugom trenutku, na istom ili drugom procesoru. Svi procesi koji uđu u računarski sistem prolaze kroz niz stanja tokom svog boravka na računaru. Jednostavno rečeno, stanje procesa opisuje ono što se u datom trenutku događa sa procesom. Prevođenje procesa iz jednog u drugo stanje obavlja operativni sistem.

Proces se može naći u nekoliko stanja, a sledeća tri su najznačajnija:

- Stanje izvršenja (**RUN**) – procesor izvršava instrukcije ovog procesa
- Stanje čekanja na procesor (**READY**) – proces je dobio sve potrebne resurse osim procesora, spreman je za rad i čeka da mu se dodeli procesor.
- Stanje čekanja na resurs (**WAIT**) – proces čeka na neki događaj, jer su za dalje izvršenje procesa potrebni resursi koji trenutno nisu na raspolaganju.



Svaki proces počinje u stanju START, nakon čega se dovodi u stanje READY, tj. u stanje čekanja na dodelu procesora u procesorskom redu. Ovo je prva tranzicija u dijagramu stanja. Posle nje, proces prolazi kroz dodatne tranzicije u konačnom automatu:

- **READY-RUN** dodela procesora procesu koji je došao na početak reda čekanja. Proces prelazi iz stanja čekanja na procesor u stanje izvršavanja kada procesor prekine izvršavanje tekućeg procesa.
- **RUN-READY** oduzimanje procesora procesu nakon isticanja vremenskog perioda za koje mu je procesor dodeljen. Ako je u pitanju *preemptive* OS, procesor se može oduzeti ako naiđe proces višeg

prioriteta. Ova tranzicija je moguća samo u višeprocesnim operativnim sistemima.

- **RUN-WAIT** oduzimanje procesora procesu ukoliko je resurs koji je potreban za izvršavanje procesa zauzet. Ova tranzicija je moguća i u jednoprocesnim i u višeprocesnim OS. Do ove tranzicije se može doći i kada se čeka rezultat operacije koju obavlja drugi proces ili ako proces čeka određeni trenutak koji je unapred programiran i u kome se može nastaviti izvršavanje.
- **WAIT-READY** proces se vraća na kraj reda čekanja nakon oslobađanja resursa koji je neophodan za rad procesa. Napomena: tranzicija WAIT-RUN nije moguća u višeprocesnim OS. U takvim sistemima, proces se uvek prvo dovodi u stanje READY.
- **RUN-STOP** proces završava rad (prirodno ili nasilno). U ovom stanju proces oslobađa sve resurse koje je koristio.

U stanju RUN se u jednom trenutku može naći onoliko procesa koliko ima procesora u sistemu. Procesi koji se nalaze u ovom stanju su tekući procesi. Proces se ne može zadržati do svog završetka u ovom stanju, zato što se u opštem slučaju posmatra višeprocesno okruženje u kome se procesor dodeljuje na određeno vreme - po nekom algoritmu. Na taj način se sprečava da jedan proces crpi resurse računara, dok drugi procesi čekaju u redu.

3. Blokiranje (suspendovanje) procesa u Linux-u

Na koji način drajver uređaja reaguje ako ne može momentalno da zadovolji zahtev? Ovakva situacija može da nastane ukoliko zahtev za čitanjem stigne u trenutku kada podaci nisu dostupni. Slična situacija nastaje i kada proces pokušava da upiše podatke u uređaj, ali uređaj nije spreman da prihvati podatke, jer je ulazni bafer pun. Proces obično ne brine oko ovakvih događaja. U slučaju ovakvih događaja drajver bi trebao da blokira proces, da ga suspenduje (*sleep*), sve dok zahtev nije moguće izvršiti. Ovaj odeljak pokazuje kako se proces može suspendovati i kasnije opet aktivirati. Najpre je, ipak, neophodno objasniti nekoliko koncepata:

Kada je proces suspendovan, označen je kao da je se nalazi u specijalnom stanju i sklonjen je iz planerovog reda čekanja (*run queue*). Dok se ne desi događaj koji će mu promeniti to stanje, procesor neće biti dodeljen procesu, i stoga neće biti izvršavan. Može se smatrati da je suspendovani proces uklonjen iz sistema i čeka na neki budući događaj. Dovođenje procesa u suspendovano stanje je postupak koji drajver uređaja vrlo lako može da uradi. Postoji, međutim, nekoliko pravila kojih se treba pridržavati kako bi se postupak suspendovanja procesa odvijao bezbedno.

Prvo zlatno pravilo je: Nikad se proces ne sme staviti u suspendovano stanje ukoliko trenutno izvršava atomične operacije (*atomic context*). Atomične operacije predstavljaju sekvencu više koraka koji moraju biti izvršeni bez ikakvog vida konkurentnog pristupa. To znači da, sa aspekta suspendovanja, drajver ne sme spavati kada koristi mehanizme *spinlock-a*, *seqlock-a*, ili *RCU lock-a* (o čemu će biti više reči kasnije). Takođe, suspendovanje je zabranjeno ukoliko su prekidi onemogućeni. Moguće je suspendovati proces koji drži semafor, ali treba biti vrlo oprezan sa kodom koji to radi. Ako je prces suspendovan dok drži semafor, svaki drugi proces (nit ili thread) koji čekaju na semafor su takođe suspendovani. Usled toga, svako suspendovanje koje se dešava dok se drži semafor treba da bude što kraće i trebamo biti naročito oprezani da se držanjem semafora ne blokira proces koji bi kasnije trebao da probudi naš proces.

Još jedna važna činjenica kod suspendovanja je: kada se proces aktivira (tj. probudi), nije poznat interval vremena koji je protekao dok je proces bio suspendovan i šta se desilo u međuvremenu. Moguć je scenario u kome se neki drugi proces, koji je takođe bio suspendovan i čekao na isti događaj, aktivirao pre prvog i zauzeo resurs na koji čeka prvi proces. Krajnji rezultat je da se ne može pretpostaviti kakvo će biti stanje sistema nakon buđenja.

Treći važan detalj je da proces ne sme da ode u suspendovano stanje sve dok se sa sigurnošću ne zna da će nešto, negde, probuditi taj proces. Deo koda koji obavlja aktiviranje mora biti sposoban da pronađe proces koji spava da bi mogao da odradi aktiviranje. Treba napraviti kod koji će dovesti do niza događaja koji će prekinuti spavanje procesa.

3.1. Redovi čekanja (wait queue)

Da bi neki suspendovani proces mogao biti pronađen, Linux kernel obezbeđuje posebnu strukturu koja se zove **red čekanja (*wait queue*)**. Red čekanja je, kao što i samo ime kaže, lista procesa koji čekaju na određeni događaj. U *Linux* kernelu, red čekanja se kontroliše preko "glave reda čekanja" - strukture tipa *wait_queue_head_t*, koja je definisana u *<linux/wait.h>*. Glava reda čekanja može biti definisana i inicijalizovana statički sa:

DECLARE_WAIT_QUEUE_HEAD(my_queue);

ili dinamički:

wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);

Kada se proces suspenduje, očekuje se da će u budućnosti neki uslov zadovoljiti (npr. resurs postati slobodan). Svaki proces koji je suspendovan mora da proveri da li je uslov na koji se čekao istinit kada se ponovo aktivira. Najjednostavniji način za suspendovanje u Linux-ovom kernelu predstavlja makro sa imenom wait_event (sa nekim varijacijama). Ovim makroom se kombinuje upravljanje postupkom suspendovanja sa proverom stanja na koje proces čeka. Forme makroa wait_event su:

wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)

U prethodnim formama, *queue* je glava reda čekanja i prosleđuje se preko svoje vrednosti. *Condition* je proizvoljni bulov uslov koji se provera pri suspendovanju i buđenju procesa. Sve dok *condition* ne postane tačan iskaz, proces nastavlja da spava, tj. ostaje u suspendovanom stanju. Ako se koristi *wait_event*, proces se stavlja u suspendovano spavanje koje je nemoguće prekinuti spoljašnjim signalom. Ovaj poziv je poželjno izbeći jer na ovaj način nastaju *unkillable* procesi - koje je nemoguće prisilno završiti. Alternativa ovome je korišćenje *wait_event_interruptible*, što omogućava prekid pomoću odgovarajućih signala. Ova verzija vraća *integer* vrednost, koju je potrebno proveriti. Nenulta vrednost znači da je došlo do prekidanja nekim terminišućim signalom (npr. proces je terminiran od strane korisnika ili kernela). Drajver bi, u tom slučaju, verovatno trebao da vrati kod greške *-ERESTARTSYS*. Ova povratna greška govori procesu da je prekinuti sistemsmki poziv potrebno restartovati, tj. pozvati iznova. Poslednje dve verzije makroa navedenih gore (*wait_event_timeout* i *wait_event_interruptible_timeout*) čekaju ograničeno vreme, a posle tog perioda vremena, makroi vraćaju vrednost nula bez obzira na to da li se odgovarajući događaj desio ili ne.

Sa druge strane imamo, naravno, aktiviranje (buđenje) procesa. Neka druga procesorska nit, proces, ili prekid mora da odradi aktiviranje procesa koji spava, pošto je on naravno u suspendovanom stanju. Osnovna funkcija koja vrši aktiviranje procesa je *wake_up*. Na raspolaganju je nekoliko formi, a ovde će biti razmotrene samo dve:

void **wake_up**(wait_queue_head_t *queue);
void **wake_up_interruptible**(wait_queue_head_t *queue);

wake_up budi sve procese koji čekaju u redu. Drugi oblik (wake_up_interruptible) se odnosi samo na procese koji se nalaze u suspendovanom stanju sa mogućnošću prekidanja. Iako je implementacija obe funkcije slična, u praksi postoji konvencija da se koristi *wake_up* ako je za suspendovanje procesa korišćen *wait_event* i *wake_up_interruptible* ako je korišćen *wait_event_interruptible*. Pri buđenju procesa pozivom *wake_up*, on ponovno evaluira bulov izraz zadat u *wait_event* funckiji. Ukoliko bulov izraz nije zadovoljen, proces nastavlja da spava iako je pozvana funkcija wake_up, zato što resurs na koji je čekao nije slobodan.

3.2. Primer blokiranja u lifo drajveru

Sledi jednostavan primer suspendovanja i aktiviranja na prethodno opisnom lifo drajveru. U kodu koji je dat u prvom poglavlju, ukoliko proces pokuša pročitati prazan lifo, drajver ne vrati podatak već ispiše poruku greške.

[Nov26 21:24] Succesfully opened lifo
[+0.002142] **Lifo is empty**
[+0.001286] Succesfully closed lifo

Takođe, ukoliko proces pokuša upisati u pun *lifo*, podatak se odbacuje i ispisuje se poruka greške.

[+0.245784] Succesfully opened lifo
[+0.002164] **Lifo is full**
[+0.001156] Succesfully closed lifo

Ovo nije poželjan slučaj jer zahteva da proces periodično pravi sistemski poziv dok se resurs ne oslobodi (*polling*), što uveliko troši procesorsko vreme. U modifikovanom kodu koji sledi su implementirana dva reda čekanja, jedan za upis (*writeQ*) i jedan za čitanje (*readQ*). Ukoliko proces pokuša da upiše u pun *lifo*, u ovom sluičaju će se blokirati u redu za upis *writeQ* sve dok neki drugi proces ne oslobodi mesto u *lifo* baferu čitajući iz njega. Na sličan način, ukoliko proces pokuša da pročita iz praznog lifo bafera, biće blokiran u redu za čitanje readQ, sve dok neki drugi proces ne upiše novu vrednost.

Nakon što definišemo prethodna dva reda čekanja pomoću naredbi:

DECLARE_WAIT_QUEUE_HEAD(readQ);
DECLARE_WAIT_QUEUE_HEAD(writeQ);

Modifikovane linije su date u sledećem kodnom segmentu:

```
ssize_t lifo_read(struct file *pfile, char __user *buffer, size_t
length, loff_t *offset)
{
	int ret;
	char buff[20];
	long int len = 0;
	if (endRead){
		endRead = 0;
		return 0;
	}

	if(wait_event_interruptible(readQ,(pos>0)))
		return -ERESTARTSYS;
	if(pos > 0)
	{
		pos --;
		len = scnprintf(buff, strlen(buff), "%d ", lifo[pos]);
		ret = copy_to_user(buffer, buff, len);
		if(ret)
			return -EFAULT;
		printk(KERN_INFO "Succesfully read\n");
	}
	else
	{
			printk(KERN_WARNING "Lifo is empty\n");
	}

	wake_up_interruptible(&writeQ);
	endRead = 1;
	return len;
}

ssize_t lifo_write(struct file *pfile, const char __user *buffer, size_t
length, loff_t *offset)
{
	char buff[20];
	int value;
	int ret;

	ret = copy_from_user(buff, buffer, length);
```

```
	if(ret)
		return -EFAULT;
	buff[length-1] = '\0';

	if(wait_event_interruptible(writeQ,(pos<10)))
		return -ERESTARTSYS;
	if(pos<10)
	{
		ret = sscanf(buff,"%d",&value);
		if(ret==1)//one parameter parsed in sscanf
		{
			printk(KERN_INFO "Succesfully wrote value %d", value);
			lifo[pos] = value;
			pos=pos+1;
		}
		else
		{
			printk(KERN_WARNING "Wrong command format\n");
		}
	}
	else
	{
		printk(KERN_WARNING "Lifo is full\n");
	}

	wake_up_interruptible(&readQ);
	return length;
}
```

Nakon što pozovemo komandu insmod, lifo je prazan. Ukoliko pokušamo da pročitamo pomoću komande:

#cat /dev/lifo &

Dobijamo ispis:

[3] 8355

[+9.567036] Succesfully opened lifo

Ampersen (&) na kraju komande nam omogućava da pokrenemo komandu u odvojenom procesu kako bi u slučaju blokiranja i dalje mogli koristiti terminal. Iz prethodnog ispisa saznajemo da je komanda za čitanje pokrenuta u procesu sa brojem (ID) “8355”. Možemo primetiti da je uspešno otvoren fajl, ali da ništa nije pročitano, niti je fajl zatvoren. Budući da je lifo bio prazan (pos==0), proces je blokiran u liniji: `wait_event_interruptible(readQ,(pos>0))` ***lifo_read*** funkcije.

Ovaj proces čeka da se neka vrednost upiše u *lifo* kako bi je on mogao pročitati. Upisaćemo vrednost ‘1’ u lifo pozivom komande:

#echo '1' >/dev/lifo

Kao rezultat dobijamo sledeći niz poruka:

```
[ +28.347329] Succesfully opened lifo
[  +0.002146] Succesfully wrote value 1
[  +0.000028] Succesfully closed lifo
[  +0.000009] Succesfully read
[  +0.000054] Succesfully closed lifo
[3]+  Done                    cat /dev/lifo
```

Na osnovu prve tri linije primećujemo da je vrednost 1 uspešno upisana u *lifo*. Kada upišemo novu vrednost, na kraju ***lifo_write*** funkcije budimo sve procese koji su blokiranu u redu za čitanje pomoću `wake_up_interruptible(&readQ);` Sada proces koji je prethodno bio blokiran u redu za upis *readQ* nastavlja izvršavanje, te čita upisanu vrednost i zatvara fajl. Poslednja linija signalizira da je proces koji je izvršavao komandu “cat /dev/lifo” završen.

Slična situacija se dešava ako se lifo napuni sa 10 vrednosti a zatim se pokuša upisati dodatan podatak ‘3’ pomoću:

#echo '3' > /dev/lifo &

```
[2] 8358
[  +6.848524] Succesfully opened lifo
```

Primećujemo da se proces blokirao u lifo_write funkciji u pozivu `wait_event_interruptible(writeQ,(pos<10))`. Kako bi se proces probudio, potrebno je pročitati vrednost i osloboditi mesto u lifo baferu. Na kraju lifo_read funkcije se bude svi procesi koji čekaju upis: `wake_up_interruptible(&writeQ);`

#cat /dev/lifo

```
[  +4.813951] Succesfully opened lifo
[  +0.002155] Succesfully read
[  +0.001567] Succesfully wrote value 3
[  +0.000430] Succesfully closed lifo
[  +0.000719] Succesfully closed lifo
[2]+  Done                    echo '3' > /dev/lifo &
```

Možemo primetiti da čim se oslobodi mesto , prethodni proces nastavlja izvršavanje, te na slobodno mesto upisuje vrednost ‘3’.

3.3. Primer hazarda pri blokiranju u lifo drajveru

Potrebno je primetiti šta bi se desilo ako bi u redu za upis *writeQ* čekala dva procesa. Pri pozivu funkcije za čitanje *lifo_read*, bi se oslobodilo jedno mesto dekrementiranjem promenljive *pos*, a zatim bi se probudili procesi koji čekaju u redu za upis *writeQ*. Moglo bi se predpostaviti da bi jedan od dva procesa inkrementirao promenljivu *pos* te da bi drugi proces pri buđenju video da je pos=10 i nastavio da spava. Na sistemu sa jednim procesorom ovo bi skoro pa zasigurno bio slučaj. Bitno je razumeti zašto ne možemo računati na ovo ponašanje. Poziv *wake_up_interruptable (&writeQ)* će probuditi oba procesa koji čekaju na resurs. Sasvim je moguće da oba procesa provere uslov pos<10 pre nego što jedan od njih uspe da inkrementira vrednost promenljive pos. Dakle, oba procesa će izvršiti upis iako je ostalo slobodno samo jedno mesto u *lifo* baferu. Ove situacije se ne mogu zanemariti na procesorima sa dva ili više jezgara i preemptive operativnim sistemima, te će sledeće poglavlje razmatrati rešavanje ovih hazarda.

Za demonstraciju u prethodno napunjen lifo jedinicama, upisujemo dve dodatne vrednosti ‘2’ i ‘3’. Oba procesa bivaju blokirana u redu za upis *writeQ*:

```
#echo '2' > /dev/lifo &
[2] 8917
[ +12.087523] Succesfully opened lifo
#echo '3' > /dev/lifo &
[3] 8918
[  +6.159904] Succesfully opened lifo
```

Zatim oslobađamo jedno mesto pozivom komande cat /dev/lifo. Može se primetiti da su se oba procesa probudila i upisala vrednosti na istu lokaciju *lifo*[9]. Budući da je drugi proces upisao vrednost 3 nakon što je prvi upisao vrednost 2, upis prvog procesa će biti prebrisan drugim. Doduše, veći problem je što su oba procesa inkrementirala globalnu promenljivu *pos*, koja sada ima vrednost 11. Ukoliko se sada pokuša pročitati vrednost, čitaće se sa adrese lifo[10] koja je van opsega statičkog niza *lifo*, te pristup toj lokaciji može rezultovati u ozbiljnim memorijskim problemima koji kompromituju stanje kompletnog sistema. Ukoliko drajver ne bi predvideo ovu situaciju, upis na lokaciju koja je van opsega niza je još veći problem.

```
#cat /dev/lifo
[  +9.984259] Succesfully opened lifo
[  +0.002153] Succesfully read
[  +0.001553] Succesfully wrote value 2
[  +0.000005] Succesfully wrote value 3
[  +0.001099] Succesfully closed lifo
1 [3]+  Done                    echo '3' > /dev/lifo
[  +0.007180] Succesfully closed lifo
[  +0.003838] Succesfully closed lifo
[2]+  Done                    echo '2' > /dev/lifo
```

4. Konkurentno izvršavanje operacija pod Linux-om

Upravljanje konkurentnošću je jedan od glavnih problema u programiranju operatnivnih sistema. Greške koje su vezane sa konkurentnošću se veoma lako prave, a teško se otkrivaju. U ranijim Linux kernelima imali smo malo izvora konkurentnosti. Simetrični multiprocesorski (SMP) sistemi nisu bili podržani od strane kernela, i jedini uzrok konkurentnog izvršavanja je bilo servisiranje hardverskih prekida. Ovaj pristup nudio je jednostavnost, ali se ne koristi u svetu gde se cene performanse sistema sa više procesora, i gde sistemski odzivi na određene događaje moraju biti što brži. Da bi odgovorio zahtevima modernih aplikacija i hardvera Linux kernel se razvio do tačke gde se mnoge stvari izvršavaju gotovo istovremeno. Međutim, to je znatno zakomplikovalo programiranje kernela. Programeri drajvera, moraju uzeti obzir konkurentnost i moraju dobro da razumeju sve alate koje nudi kernel za upravljanje konkurentnošću.

Više korisniških procesa se smenjuju na procesoru te mogu pristupiti kodu našeg drajvera na iznenađujuće mnogo načina. U SMP sistemima se na različitim procesorima može izvršavati isti deo kernela. Pošto je Linux kernel *preemptive* tipa, drajver može izgubiti kontrolu nad procesorom u svakom trenutku, a proces koji ga je zamenio može takođe koristiti funkcionalnost koju pruža taj isti drajver. Prekidi koje generišu neki uređaji su asinhroni događaji koji takođe mogu zahtevati izvršenje koda u okviru drajvera. Takođe, u današnjem svetu gde su popularni uređaji koji se sami inicijalizuju prilikom uključenja u sistem (hot-pluggable), može se desiti da uređaj nestane u toku rada drajvera koji ga kontroliše.

Dakle, u modernim Linux sistemima imamo mnošto izvora konkurentnosti, te postoji velika mogućnost pojave problema prilikom pristupa deljenim resursima. Ovi problemi se nazivaju *race conditions*. Ako nema deljenih resursa medju procesorskim nitima, neće ni biti ovakvih problema. Stoga, potrebno je potruditi se da kernelski kod ima što manje deljenih resursa. Najočigledniji primer ovoga je izbegavanje korišćenja globalnih promenljivih.

U realnosti nije moguće u potpunosti eliminisati deljene resurse. Hardverski resursi su po svojoj prirodi deljeni, dok softverski često moraju biti vidljivi u većem broju niti izvršavanja ili procesa.

U primeru uređaja iz prethodnog poglavlja, globalno su deklarisani niz **lifo** i promenljiva **pos**, koje dele svi procesi. Rukovanje ovim promenljivim je kritična sekcija u tom drajveru, jer može rezultovati u hazardima ukoliko nema adekvatne zaštite. Pogledajmo jednostavan primer sa dva procesa A i B, gde oba žele da upišu vrednost na poslednje mesto u lifo baferu. Zamislimo da se oni izvršavaju u sistemu sa *preemptive* kernelom i jednim procesorom, te da je proces A u stanju RUN a B u stanju READY. A se prvi izvršava, te proveri vrednost promenljive *pos* koja je jednaka 9 što znači da ima slobodno mesto u lifo baferu. Pre nego što stigne da upiše vrednost i da inkrementira *pos* desi se neki spoljašnji događaj (npr. prekid,

eng. *interrupt*), što prisili kernel da proces A prebaci u stanje READY. Nakon što kernel opsluži spoljašnji događaj, on treba da dodeli procesor nekom od procesa u listi čekanja. Ukoliko se planer opredeli za proces B, desiće se hazard. Naime, proces B sada takođe proveri vrednost promenljive *pos*, koja je još uvek jednaka 9. Proces B izvršava upis, te uveća vrednost promenljive *pos* na 10. Proces se završava, te kernel prebacuje proces A u stanje RUN. Budući da je proces A proverio stanje promenljive *pos* pre nego što se desio prekid, sada on nastavlja izvršavanje tako što upisuje vrednost na poziciju lifo[10] - van opsega niza. Ovaj problem je teško rekreirati za potrebe vežbi, ali se u praksi može desiti te se ne sme zanemariti.

Mnogo jednostavniji primer za razumeti i rekreirati je demonstriran u prethodnom poglavlju, gde se procesi A i B izvršavaju konkurentno na dva procesorska jezgra.

4.1. Kritične sekcije

Ključni problem je sledeći: procesi su prekinuti u tački gde ne bi trebalo da budu prekidani ako želimo da pravilno rade. Očigledno, potencijalno rešenje je da označimo sekcije koda tako da one ne mogu biti prekinute od strane planera. Zato uvodimo pojam **kričnih sekcija**.

Cilj ovoga je, da jednom delu koda koji smo proglasili kritičnom sekcijom, može pristupiti samo jedan proces u datom trenutku. Na primer, deo koda za upravljanje nizom *lifo* i promenljivom *pos* naveden ranije ćemo proglasiti kritičnom sekcijom. Procesi koji su ušli u kritičnu sekciju mogu biti prekinuti jedino od procesa koji nemaju nameru pristupa kritičnoj sekciji.

Nisu sve kritične sekcije iste, pa usled toga kontrola pristupa različitim kritičnim sekcijama može biti regulisana različitim mehanizmima zaključavanja koje nam obezbeđuje Linux kernel.

4.2. Semafori i mutexi

Semafori predstavljaju jednostavan mehanizam zaštite, a sastoje se od promenljivih koje mogu da imaju pozitivnu i negativnu celobrojnu vrednost.

Dve standardne operacije su definisane za manipulisanje semaforima - **UP** and **DOWN**. One se koriste da kontrolišu ulazak i izlazak iz kritičnih sekcija.
Kada proces pristupa kritičnoj sekciji u kodu, on poziva funkciju DOWN. Ovo umanjuje vrednost semafora za 1 i ukoliko je vrednost semafora veća od 0 izvršava se kritični kod. Kada obavi sve u kritičnoj sekciji, poziva se UP funkcija koja povećava vrednost semafora za 1. Očigledno je da se na ovaj način može dozvoliti proizvoljnom broju procesa da istovremeno pristupe kritičnoj sekciji, tj. deljenom

resursu. Ukoliko je inicijalno vrednost semafora postavljena na 1, tada će samo jedan proces moći da pristupi deljenom resursu u datom trenutku.

Semafori funkcionišu na sledeći način:

1. Pretpostavimo da je proces A uzeo kontrolu nad semaforom (čija je vrednost inicijalno postavljena na ‘1’ i ušao u kritičnu sekciju koda. Kada proces B pokuša da prisupi sekciji kritičnog koda, on prvo mora da obavi DOWN operaciju nad semaforom. Zbog toga što je prvi proces već unutar kritične sekcije, vrednost semafora je 0. Ovo uzrokuje da proces B bude suspendovan dok se ne oslobodi semafor. Drugim rečima, on čeka dok proces A ne izađe iz kritične sekcije. Od velike važnosti za implementiranje DOWN operacije je to da se ona obavlja kao elementarni korak (atomička operacija) gledano sa stanovišta ostalh procesa. Ono ne može biti prekinuto od strane planera što dalje znači da se ne može pojaviti *race condition* (hazard). Iz kernelovog ugla gledanja očitavanje varijable i promena njene vrednosti su dve različite akcije koje korisnik vidi kao jednu (atomičku) operaciju. Kada proces spava čekajući semafor, kernel ga je prebacio u blokirano stanje i takođe ga smestio na listu čekanja sa ostalim procesima koji čekaju na semafor.
2. Kada proces izlazi iz kritične sekcije obavlja operaciju UP. Ona ne samo što povećava vrednost semafora za 1, već i inicira odabir sledećeg procesa od onih koji se nalaze u redu čekanja na semafor. Izabrani proces biva aktiviran i nakon kompletiranja DOWN operacije može bezbedno da izvršava kod iz kritične sekcije.

4.2.1. Implementacija semafora u Linux-u

Da bi smo mogli koristiti mehanizam semafora kernel kod mora da uljuči <linux/semaphore.h>. Rad sa semaforima omogućen je preko strukture **semaphore**:

```
struct semaphore {
        atomic_t count;
        int sleepers;
        wait_queue_head_t wait;
};
```

Semafor može biti deklarisan i inicijalizovan na dva načina. Jedan od načina je da kreiramo semafor direktno, korišćenjem funkcije *sema_int,* gde je *val* inicijalna vrednost semafora:

```
void sema_init (struct semaphore *sem, int val);
```

Ako je potrebno semafor inicijalizovati u mutex modu (val = 1) u toku izvršavanja (*runtime*) programa, koristimo dinamičku inicijalizaciju:

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(stuct semaphore *sem);
```

Za dekrementiranje vrednosti semafora u Linux-u koristimo tri verzije down funkcije:

- **void** down(**struct semaphore** *sem) - dekrementuje vrednost semafora, a ako je semafor već rezervisan, tada proces ide u *sleep* mod i nalazi se u TASK_UNINTERRUPIBLE stanju. U ovom stanju proces ne može da primi nijedan spoljašnji signal. Tako proces čeka na oslobađanje semafora, koliko god vremena da je potrebno. Korišćenje ove opcije je pogodno kad hoćemo da napravimo procese koje ne možemo da terminiramo spoljašnjim signalom i, u principu, ovakav pristup se retko koristi.
- **int** down_interruptible(**struct semaphore** *sem); - dekrementuje vrednost semafora. Ako je semafor rezervisan, u tom slučaju proces ide u *sleep* mod i nalazi se u TASK_INTERRUPTIBLE stanju. To znači da se proces može biti prekinuti i u sleep modu od strane korisnika, i na taj način odustati od čekanja na resurs. (na primer, pomoću CTRL+C). Ako je proces prekinut u sleep stanju funkcija vraća vrednost različitu od nule, dok u suprotnom vraća nulu. Korišćenje ove funkcije podrazumeva pažljivo rukovanje jer treba pratiti koje vrednosti vraća funkcija.
- **int** down_trylock(**struct semaphore** *sem); - dekrementuje vrednost semafora. Ako je semafor zauzet proces ne ide u *sleep* mod da čeka kad će se osloboditi semafor, već nastavlja sa izvršavanjem. Funkcija vraća vrednost 0 ako je proces dobio na raspolaganje semafor, dok u suprotnom vraća vrednost različitu od 0.

Kad proces obavi sve operacije u kritičnoj sekciji, mora da pozove funkciju koja inkrementira vrednost semafora i tako oslobodi semafor omogućavajući drugim procesima da pristupe kritičnoj sekciji.

Funkcija koja ikrementira vrednost semafora u Linux-u je:

- **void** up(**struct semaphore** *sem);

Posebna pažnja se često zahteva u slučaju pojave grešaka u samoj kritičnoj sekciji, jer se i tada mora obezbediti da se semafor oslobodi.

4.2.2. Primer upotrebe semafora u lifo drajveru

Na početku ovog poglavlja su bile opisane situacije u kojima može doći do *race condition* hazarda pri radu sa lifo drajverom. U prethodnom poglavlju je demonstrirana situacija kada dva procesa istovremeno upisuju u poslednje mesto u lifo baferu, te prouzrokuju memorijski problem. Ukoliko želimo da lifo drajver radi u modernim operativnim sistemima i na platformama sa više procesora, pristup

globalnim promenljivim kao što su niz *lifo* i promenljiva *pos*, moraju biti zaštićeni semaforom.

Prvo, mora biti deklarisana struktura semafora kao globalna promenljiva:

struct semaphore sem;

Takođe mora biti inicijalizovana u module_init funkciji pozivom:

sema_init(&sem,1);

Početno stanje semafora je postavljeno na ‘1’, što znači da će u svakom trenutku kritičnoj sekciji moči da pristupi samo jedan proces. Prethodni poziv je mogao biti zamenjen sa **init_MUTEX**(&sem) što je ekvivalentan poziv.

Modifikovane *lifo_read* i *lifo_write* funkcije su date u nastavku:

```
ssize_t lifo_read(struct file *pfile, char __user *buffer, size_t
length, loff_t *offset)
{
	int ret;
	char buff[20];
	long int len = 0;
	if (endRead){
		endRead = 0;
		return 0;
	}

	if(down_interruptible(&sem))
		return -ERESTARTSYS;
	while(pos == 0)
	{
		up(&sem);
		if(wait_event_interruptible(readQ,(pos>0)))
			return -ERESTARTSYS;
		if(down_interruptible(&sem))
			return -ERESTARTSYS;
	}

	if(pos > 0)
	{
		pos --;
		len = scnprintf(buff, strlen(buff), "%d ", lifo[pos]);
		ret = copy_to_user(buffer, buff, len);
		if(ret)
			return -EFAULT;
		printk(KERN_INFO "Succesfully read\n");
	}
	else
	{
```

```
                printk(KERN_WARNING "Lifo is empty\n");
        }

        up(&sem);
        wake_up_interruptible(&writeQ);
        endRead = 1;
        return len;
}

ssize_t lifo_write(struct file *pfile, const char __user *buffer, size_t
length, loff_t *offset)
{
        char buff[20];
        int value;
        int ret;

        ret = copy_from_user(buff, buffer, length);
        if(ret)
                return -EFAULT;
        buff[length-1] = '\0';

        if(down_interruptible(&sem))
                return -ERESTARTSYS;
        while(pos == 10)
        {
                up(&sem);
                if(wait_event_interruptible(writeQ,(pos<10)))
                        return -ERESTARTSYS;
                if(down_interruptible(&sem))
                        return -ERESTARTSYS;
        }

        if(pos<10)
        {
                ret = sscanf(buff,"%d",&value);
                if(ret==1)//one parameter parsed in sscanf
                {
                        printk(KERN_INFO "Succesfully wrote value %d", value);
                        lifo[pos] = value;
                        pos=pos+1;
                }
                else
                {
                        printk(KERN_WARNING "Wrong command format\n");
                }
```

```
        }
        else
        {
               printk(KERN_WARNING "Lifo is full\n");
        }

        up(&sem);
        wake_up_interruptible(&readQ);

        return length;
}
```

Modifikovani kod koji štiti kritičnu sekciju je označen plavom bojom. Pogledajmo slučaj koji je pre pravio hazard - kada se više procesa blokira pokušavajući da upišu u pun lifo baffer, nakon čega se čitanjem oslobodi mesto. Dakle, budući da je bafer pun (sve vrednosti '1'), *pos* = 10. Proces A pokušava da upiše vrednost '2' u lifo:

 # echo '2' > /dev/lifo &
[2] 2189
[+14.351779] Succesfully opened lifo

On prvo zauzima semafor pozivom `down_interruptible (&sem).` Zatim ulazi u while petlju zato što je vrednost pos =10. On oslobađa semafor `up(&sem);` a zatim se blokira u redu za upis pozivom `wait_event_interruptible (writeQ,(pos<10)).` Bitno je osloboditi semafor pre blokiranja kako bi neki drugi proces mogao pročitati vrednost iz lifo bafera i probuditi blokirani proces. Proces A čeka na uslov pos<10, dakle da bude barem jedno prazno mesto u lifo baferu.

Proces B takođe pokušava da upiše u lifo i biva blokiran na isti način. Sada se u redu za upis writeQ nalaze procesi A i B.

 # echo '3' > /dev/lifo &
[3] 2190
[Nov27 12:39] Succesfully opened lifo

Proces C čita vrednost '1' iz lifo bafera, nakon čega budi procese iz reda za čitanje pozivom `wake_up_interruptible(&writeQ)` u lifo_read funkciji. Procesi A i B se probude zato što je sada pos<10 (ima jedno slobodno mesto), ali samo jedan od ova dva procesa uspeva da zauzme semafor u sledećoj liniji `down_interruptible (&sem)`, dok drugi biva blokiran u istoj liniji i čeka da se semafor oslobodi. Onaj proces (u našem slučaju B) koji je zauzeo semafor, ispada iz *while* petlje, te pristupa kritičnoj sekciji i upisuje vrednost '3' u lifo. Kada on završi upis, oslobađa semafor i budi proces A koji čeka na isti. Proces A zauzima semafor,

ali ne izlazi iz while petlje jer je pos=10 (proces B je upisao podatak na slobodno mesto). U *while* petlji proces A oslobađa semafor i blokira se u redu za upis *writeQ*.

```
# cat /dev/lifo
1
[ +12.608858] Succesfully opened lifo
[  +0.002158] Succesfully read
[  +0.001565] Succesfully wrote value 3
[  +0.000042] Succesfully closed lifo
[  +0.001030] Succesfully closed lifo
[3]+  Done                    echo '3' > /dev/lifo
```

Možemo ponovno da pročitamo vrednost iz lifo bafera, te na taj način oslobodimo jedno mesto i probudimo proces A koji čeka u redu za upis *writeQ.* Proces A će uspeti da zauzme semafor, izaći će iz *while* petlje i upisati vrednost ‘2’ na poslednje mesto u lifo baferu:

```
# cat /dev/lifo
3
[Nov27 12:45] Succesfully opened lifo
[  +0.002152] Succesfully read
[  +0.001559] Succesfully wrote value 2
[  +0.000449] Succesfully closed lifo
[  +0.000662] Succesfully closed lifo
[2]+  Done                    echo '2' > /dev/lifo
```

Zahvaljujući semaforu, samo jedan proces može u istom trenutku da pristupi kritičnoj sekciji čime se rešavaju *race condition* hazardi.

4.3. Spinlock-ovi

Spinlokovi se koriste za zaštitu kratkih sekvenci koda koje sadrže samo nekoliko C naredbi pa je njihovo izvršavanje i izlazak brz. Veći deo kernel strukture podataka ima svoje spinlokove. Stoga možemo reći da se većina zaključavanja u Linux-u obavlja kroz mehanizam spinlokova. Za razliku od semafora, spinlokovi se mogu koristi u delu koda koji nema mogućnost odlaska u sleep mod, kao što je na primer upravljanje prekidima. Ukoliko se pravilno koriste, spinlokovi nude, generalno, bolje performanse nego semafori. Spinlokovi predstavljaju jednostavan koncept. Oni su zapravo ništa drugo nego međusobno isključiv mehanizam koji ima samo dve vrednosti ”zaključan” ili ”otključan”. Ta funkcionalnost je najčešće implementirana kao jedan bit u celobrojnoj vrednosti. Ako je lock bit “1” tada proces može da pristupi kritičnoj sekciji. Ako je lock bit “0”, tada znamo da se mehanizam koristi od strane drugog procesa i proces odlazi u petlju u kojoj stalno proverava da li je spinlok i dalje

zaključan, sve dok on ne postane slobodan.

Prava implementacija spinloka je ipak malo kompleksnija u odnosu na ono kako je gore objašnjeno. Operacije “test i set” moraju se obavljati kao atomične operacije, što obezbeđuje da se u toku izvršavanja tih operacija ne može desiti da nešto prekine njihovo izvršavanje. Spinlokovi se po svojoj prirodi namenjeni za multiprocesorske sisteme, kao i za jednoprocesorske radne stanice čiji kernel kod je *preeptive* tipa.

4.3.1. Implementacija spinlock-ova u Linux-u

Da bi mogao da koristi spinlok primitive, kod mora da uključuje <linux/spinlock.h>. Struktura spinlock_t implementira kompletnu funkcionalnost spinlokova. Spinlokovi moraju biti inicijalizovani. Inicijalizaciju možemo izvršiti prilikom kompajliranja na sledeći način:

spinlock_t my_lock = **SPIN_LOCK_UNLOCKED**;

ili u toku izvršavanja programa pomoću:

void spin_lock_init(spinlock_t *lock);

Pre ulaska u kritičnu sekciju kod koji implementira spinlok mora imati funkciju za zaključavanje sekcije:

void **spin_lock**(spinlock_t *lock);

Većina spinlokova su u *UNINTERRUPTIBLE* stanju. Jednom kad pozovemo spin_lock, proces će biti u petlji sve dok spinlok ne postane slobodan. Za oslobađanje spinlok mehanizma moramo koristiti:

void **spin_unlock**(spinlock_t *lock);

Pored ovih osnovnih spinlok funkcija, imamo na raspolaganju mnogo veći set funkcija:

- void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
- void spin_lock_irq(spinlock_t *lock);
- void spin_lock_bh(spinlock_t *lock)

- void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
- void spin_unlock_irq(spinlock_t *lock);
- void spin_unlock_bh(spinlock_t *lock);

- int spin_trylock(spinlock_t *lock);
- int spin_trylock_bh(spinlock_t *lock);

4.3.2. Spinlock-ovi i atomički kontekst

Zamislimo da naš drajver dobije na raspolaganje spinlok mehanizam i krene sa radom u kritičnoj sekciji. Pretpostavimo da se negde u sred rada drajvera u kritičnoj sekciji desi to da drajver izgubi procesor (pozivanjem na primer funkcije *copy_from_user*) koja suspenduje proces. Moguć je naravno i scenario u kome kernel prekine naš proces procesom višeg prioriteta. Naš kod sada drži spinlok koji neće biti oslobođen određeno vreme (trajanje tog vremena je nepredvidivo). Ako pri tom neki drugi proces pokuša da zauzme isti spinlok, on će u najboljem slučaju čekati dugo vremena. U najgorem slučaju sistem može ući u mrtvu petlju, ako proces koji čeka na spinlok zapravo treba da aktivira suspendovani proces koji drži spinlok.

Ovakav scenario valja izbeći. Zbog toga je glavno pravilo primene spinlokova da svaki kod koji zadržava spinlok mora da bude atomički.

U slučaju takve realizacije proces ne može da ide na spavanje i ne može da izgubi procesor iz bilo kog razloga osim obrade prekida.

Sam *preemtion* mehanizam je upravljan od strane koda za spinlok. To znači da je, uvek kada kernel kod drži zaključan spinlok, preemption mod zabranjen na procesoru na kome se izvršava kod.

Koliko god to delovalo jednostavno, izbegavanje poziva funkcija koje mogu poslati proces u sleep mod (suspendovati ga) nije nimalo jednostavan zadatak. Razlog tome leži u činjenici da veliki deo kernel koda može da pošalje proces u sleep mod. Dodatan problem predstavlja činjenica takvi delovi koda često nisu najbolje dokumentovani. Zbog svega toga korisnik mora biti izuzetno oprezan prilikom pozivanja svake funkcije dok drži kontrolu nad spinlokom.

Evo još jednog problematičnog scenarija:

Drajver se izvršava i on je upravo zauzeo spinlok koji kontroliše pristup uređaju. Dok je spinlok zauzet, uređaj generiše prekid koji uzrokuje da se pokrene prekidna rutina. Prekidna rutina, pre nego što pristupi uređaju, mora prvo da dobije spinlok na raspolaganje. Imajući u vidu da je kod koji je prvobitno zauzeo spinlok suspendovan i poslat u sleep mod bez prethodnog oslobađanja spinlok mehanizma, prekidna rutina stalno proverava u petlji da li je spinlok slobodan, i procesor ostaje u mrtvoj petlji zauvek.

Ovakav scenario se dešava u slučaju kad se i proces i prekidna rutina izvršavaju na istom procesoru.

Da bi se ovakav scenario sprečio neophodno je onemogućiti prekide na lokalnom procesoru, dokle god je spinlok zauzet.

Još jedno veoma važno pravilo u vezi korišćenja spinlokova je da spinlokovi moraju uvek biti zauzeti što je moguće manje vremena. Ukoliko duže vremena zadržavamo spinlok utoliko će duže vremena ostali procesi čekati na procesor, sve dok proces koji je zauzeo spinlok ne oslobodi i procesor.

5. Asinhrono obaveštavanje

5.1. Opis aplikacije koja podržava asinhrono obaveštavanje

Iako na prvi pogled izgleda da je kontrola sistema putem blokiranja sasvim dovoljna, postoje ipak situacije koje nisu dovoljno dobro podržane tehnikama koje su do sada prikazane.

Zamislimo proces koji izvršava dugačku petlju koja nešto računa sa niskim prioritetom, ali takođe mora da procesira ulazne podatke što je pre moguće. Ako ovaj proces odgovara na pojavu podataka na nekoj periferiji, želeo bi odmah da zna kada je novi podatak dostupan. Ova aplikacija bi mogla biti napisana tako da povremeno proverava da li ima podataka na raspolaganju (*polling* metoda), no postoji i bolji način. Upotrebom asinhronog obaveštavanja, ova aplikacija bi mogla da prima signal od strane kernela svaki put kada podaci postanu dostupni i ne mora trošiti dragoceno vreme na periodičnu proveru dostupnosti podataka.

Sledeći kodni listing ilustruje aplikaciju koja koristi asinhrono obaveštavanje. Modul koji obaveštava aplikaciju jeste prethodno opisani Lifo, sa malim modifikacijama kako bi podržao asinhrono obaveštavanje (Modifikacije će biti opisane uskoro).

```
// asynctest.c: use async notification to read stdin

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>

int gotsignal=0;
int datacnt=0;
int end = 0;
void sighandler(int signo)
{
    if (signo==SIGIO)
    {
      gotsignal=1;
      datacnt++;
    }
    return;
}
```

```
char buffer[4096];
int main(int argc, char **argv)
{
    int count;
    struct sigaction action;
    int fd=open("/dev/lifo",O_RDONLY|O_NONBLOCK);
    if (!fd)
    {
      exit(1);
      printf("Error opening file\n");
    }
    memset(&action, 0, sizeof(action));
    action.sa_handler = sighandler;
    sigaction(SIGIO, &action, NULL);

    fcntl(fd, F_SETOWN, getpid());
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | FASYNC);

    while(1)
    {
      sleep(8600);
      if (!gotsignal)
          continue;

      fflush(stdout);
      gotsignal=0;
      count=read(fd, buffer, 4096);
      buffer[count]='\0';
      printf("Read: %s\n",buffer);
      count=read(fd, buffer, 4096);
    }
    close(fd);
}
```

Prvo što je neophodno uraditi, kada program krene da se izvršava, jeste otvaranje modula sa kojim se komunicira.

```
int fd=open("/dev/lifo",O_RDONLY|O_NONBLOCK);
```

U primeru to je *lifo* modul, koji se otvara samo za čitanje i to u neblokirajućem modu. Nakon toga potrebno je specificirati da će proces biti osetljiv na spoljašnji signal i koja funkcija se poziva kada se taj signal desi. U prethodnom kodu to je učinjeno pomoću:

```
memset(&action, 0, sizeof(action));
action.sa_handler = sighandler;
```

```
sigaction(SIGIO, &action, NULL);
```

Prva linija koda jeste *void *memset(void *s, int c, size_t n)* koja popunjava prvih “n” bajtova memorijskog prostora na koji pokazuje “s”, sa konstantom “c”. U našem primeru ona je zadužena da postavi sva polja *action* strukture na 0. Sledeća linija inicijalizuje *sa_handler* polje *action* strukture i to je jedino polje koje je od interesa. Ono specificira koja funkcija se poziva prilikom asinhronog obaveštavanja i u našem slučaju to je funkcija *sighandler().* Ona, ukoliko se desio SIGIO signal, treba da postavi *gotsignal* promenljivu na 1 i inkrementira *datacnt.* Kada su polja *action* strukture podešena, ono se prosleđuje kao jedan od parametara *sigaction()* sistemskog poziva (treća linija). Procesu koji pozove *sigaction()*, kernel omogućava asinhrono obaveštavanje.
Prototip ovog sistemskog poziva je sledeći:

int sigaction(**int** *signum*, **const struct sigaction** **act*, **struct sigaction** **oldact*);

- ***Signum**,* specificira signal na koji je proces osetljiv. U našem slučaju to je SIGIO, koji se šalje kada je fajl deskriptor spreman za upis ili čitanje. Samo specifični fajlovi mogu da generišu SIGIO signal (terminali, soketi ili u našem lifo drajver), dok ga ostali, “regularni” fajlovi, nikada ne generišu.
- ***Act**,* ukoliko nije NULL pokazivač, pokazuje na strukturu u kojoj je specificirana akcija koja treba da se poveže sa specificiranim signalom.
- ***Oact**,* ukoliko nije NULL pokazivač, pokazuje na strukturu u kojoj je specificirana akcija koja je prethodno bila povezana sa specificiranim signalom.

Detaljnije o ovom sistemskom pozivu može se naći na sledećem linku:
http://man7.org/linux/man-pages/man3/sigaction.3p.html

Nakon što je definisano koja akcija se pokreće kada fajl pošalje signal, neophodno je izvršiti još dva koraka da bi asinhrono obaveštavanje funkcionisalo:

```
fcntl(fd, F_SETOWN, getpid());
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | FASYNC);
```

Vidimo da se u oba koraka poziva *fnctl()* sistemski poziv. On vrši jednu od operacija nad otvorenim fajl deskriptorom u zavisnosti od prosleđenih parametara (preko fajl deskriptora menja polja fajl strukture opisane na prethodnim vežbama). Njegov prototip je sledeći:

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

- *Fd* parametar predstavlja fajl deskriptor nad kojim se vrši određena operacija

- *Cmd* parametar predstavlja komandu koja se vrši nad fajl deskriptorom
- *Arg* je opcioni parametar. Da li je on potreban ili ne zavisi od *cmd* parametra.

Prvi korak, odnosno prvi poziv *fnctl()* funkcije, postavlja proces koji se izvršava za vlasnika („owner"-a) fajla koji predstavlja periferni uređaj. Kada proces pozove F_SETOWN komandu, korišćenjem fcntl sistemskog poziva, identifikacioni broj procesa (process ID, "fd" u našem slučaju) je sačuvan u polju filp->f_owner za kasniju upotrebu(ta informacije je potrebna *Lifo* modulu). Da bi asinhrono obaveštavanje zaista bilo moguće, potrebno je izvršiti drugi korak, odnosno korisnički programi moraju setovati FASYNC fleg u uređaju pomoću F_SETFL fcntl komande. F_SETFL komanda, menja *f_flags* polje *file* strukture. Nova vrednost tog polja jeste treći parametar *fnctl* sistemskog poziva, i ako se pogleda prethodni kod može se videti da je treći parametar `fcntl(fd, F_GETFL) | FASYNC`. Pozivom `fcntl(fd, F_GETFL)` dobija se trenutna vrednost *f_flag* polja, a kada se uradi " | FASYNC", vrši se proširenje *f_flags* polja.

Nakon što su ova dva poziva izvršena proces postaje osetljiv na SIGIO signal koji šalje *Lifo* modul. Ostatak koda vrši čitanje sadržaja *Lifo* modula samo kada on pošalje signal da je u njega nešto upisano.

```
while(1)
    {
      sleep(8600);
      if (!gotsignal)
          continue;

      fflush(stdout);
      gotsignal=0;
      count=read(fd, buffer, 4096);
      buffer[count]='\0';
      printf("Read: %s\n",buffer);
      count=read(fd, buffer, 4096);
    }
close(fd);
```

Iz koda se može videti da proces uglavnom spava jer je pozvan sistemski poziv sleep(8600). Spavanje se može prekinuti na dva načina, prvi ukoliko istekne 8600s i drugi ukoliko *Lifo* modul pošalje signal. Kada istekne 8600s *if* uslov pita da li je *gotsignal=0*, ako jeste, vrati se na spavanje jer još nije poslat SIGIO signal da nešto može da se pročita. Kada *Lifo* pošalje SIGIO, poziva se *sighandler()* funkcija, koja postavlja *gotsignal* promenljivu na 1. Nakon što se *sighandler()* funkcija izvši, *main* kod nastavlja da se izvršava i opet se dolazi do *if* uslova. Sada, pošto je *gotsignal=1,*

if uslov nije ispunjen (signal poslat) i moćiće da se izvrši ostatak koda, koji samo čita jedan karakter. Nakon što je karakter pročitan, proces se vraća na spavanje.

5.2. Opis drajvera koji podržava asinhrono obaveštavanje

Do sada smo analizirali kako se koristi asinhrono obaveštavanje sa stanovišta aplikacije, tj. šta treba da se uradi u korisničkom programu da bi mogao da prima poruke od kernela o događajima koji su se desili. Sada ćemo da vidimo kako se implementira slanje poruka od strane drajvera, tj. kako kernel šalje poruke onim procesima koji su se prijavili za prijem asinhronih poruka. Pozivom prethodno pomenutih *fnctl* funkcija proces menja polja fajl strukture i te promene drajver registruje na sledeći način

- Kada se pozove fcntl sa komandom F_SETOWN ništa se posebno ne dešava osim što se postavlja filp->f_owner.
- Kada se izvrši F_SETFL komanda kako bi se uključio FASYNC, poziva se fasync funkcija unutar drajvera. Ona se poziva svaki put kada se menja vrednost FASYNC u filp->f_flags kako bi drajver bio obavešten o promeni i kako bi mogao da odreaguje na odgovarajući način. Kada se fajl otvara ovaj fleg je resetovan i o tome treba voditi računa.

Iz prethodno opisanog sledi da bi *Lifo* drajver trebalo da implementira funkciju koja se poziva svaki put kada proces setuje FASYNC u *f_flags* polju fajl strukture. Da bi se to omogućilo, u prethodno opisanom *Lifo* drajveru, treba prvo proširiti *file_operation* strukturu:

```
struct file_operations my_fops =
{
    .owner = THIS_MODULE,
    .open = lifo_open,
    .read = lifo_read,
    .write = lifo_write,
    .release = lifo_close,
    .fasync = lifo_fasync,
           ...
};
```

Vidimo da je u njoj definisan još jedan pokazivač *.fasync* koji pokazuje na implementaciju naše funkcije u drajveru. Na ovaj način smo naznačili koja funkcija treba da se pozove kada se setuje FASYNC. U nastavku je data njena implementacija:

```
static int lifo_fasync(int fd,struct file *file, int mode)
```

```
{
    return fasync_helper(fd, file, mode, &async_queue);
}
```

Lifo async funkcija unutar sebe poziva *fasync_helper* funkciju kako bi se dodao ili uklonio element sa liste procesa kojima treba slati asinhrone signale kada se menja stanje FASYNC flega otvorenog fajla.

```
int fasync_helper(int fd, struct file *filp, int mode, struct
                  fasync_struct **fa);
```

Prva tri argumenta *fasync_helper* funkcije i argumenti *lifo_fasync* su u potpunosti isti i iz prethodnog primera može se videti da su oni samo prosleđeni *fasync_helper* funkciji. 4. argument, *fa,* predstavlja red u koji se smeštaju procesi koji zahtevaju asinhrono obaveštavanje i taj red je u drajveru neophodno definisati kao globalnu promenljivu:

```
struct fasync_struct *async_queue;
```

Kada se pozove *lifo_fasync* funkcija, proces je registrovan za asinhrono obaveštavanje, ali u drajveru još uvek nije naglašeno kada da se pošalje signal. Da bi se to uradilo, neophodno je pravovremeno pozvati sledeću funkciju:

```
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

Prvi argument, *fa,* je red u kome se nalaze procesi koji su registrovani za asinhrono obaveštavanje, drugi argument je tip signala koji se šalje (u našem slučaju SIGIO) i treći argument može da bude POLL_IN ili POLL_OUT što naznačava da proces koji čeka signal može da pročita ili upiše podatak.

U našem sličaju drajver treba da obavesti procese koji čekaju svaki put kada se upiše novi podatak, stoga je *kill_fasync* funkcija pozvana u *write* funkciji drajvera. Sledeći kodni listing prikazuje modifikaciju ‘*write*’ funkcije *Lifo* drajvera kako bi se podržala asinhrona komunikacija:

```
...
if(pos<10)
      {
            ret = sscanf(buff,"%d",&value);
            if(ret==1)//one parameter parsed in sscanf
            {
                  printk(KERN_INFO "Successfully wrote value %d",
value);
```

```
                lifo[pos] = value;
                pos=pos+1;
                kill_fasync(&async_queue, SIGIO, POLL_IN);
            }
            else
            {
                printk(KERN_WARNING "Wrong command format\n");
            }
        }
        else
        {
            printk(KERN_WARNING "Lifo is full\n");
        }
...
```