Sveučilište J. J. Strossmayera u Osijeku

Fakultet elektrotehnike, računarstva i informacijskih tehnologija

Kneza Trpimira 2b

HR-31000 Osijek

www.ferit.unios.hr

Laboratorijska vježba 4:

Upravljanje procesima unutar operacijskog sustava Linux

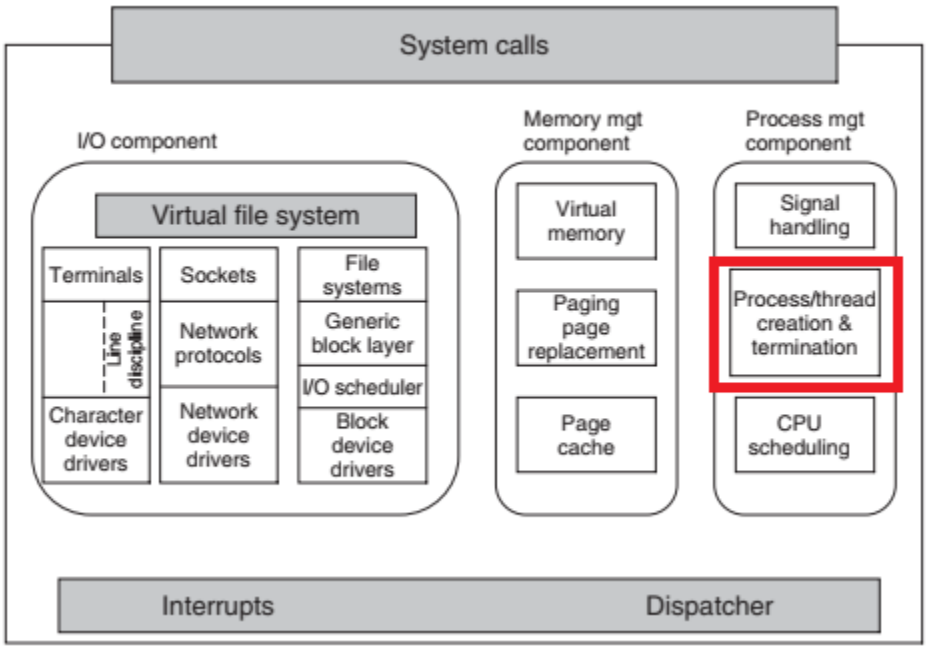
Sadržaj

I. UVOD

Koncept procesa je fundamentalan za strukturu modernog operacijskog sustava. Svi operacijski sustavi koji podržavaju više programa (engl. *multiprogramming systems*) su izgrađeni na temelju koncepta procesa. Većina zahtjeva koje operacijski sustav mora ispuniti mogu se izraziti u odnosu na proces:

- Operacijski sustav mora rasporediti izvršavanje više procesa u svrhu maksimiziranja iskorištenja procesora.
- Operacijski sustav mora alocirati resurse upravo za procese sukladno specifičnim pravilima, a izbjegavajući zastoje (engl. *deadlock*).
- Operacijski sustav mora podržati međuprocesnu komunikaciju i kreiranje procesa, unutar korisničkih aplikacija.

U ovoj laboratorijskoj vježbi ćemo dati pregled načina na koji operacijski sustavi reprezentiraju i kontroliraju procese. Zatim ćemo prodiskutirati stanja ponašanja procesa i podatkovne strukture kojima se oni implementiraju. U trećem poglavlju ćemo prikazati osnovne strukture podataka, funkcije, sustavske pozive i biblioteke pomoću kojih Linux jezgra upravlja procesima. Na kraju ćemo prikazati osnovne naredbe ljuske pomoću kojih korisnik operacijskih sustava ima uvid u izvršavanje procesa.



Slika 1. Struktura Linux jezgre

I.1. Što je proces?

Proces je instanca programa koja se izvršava na računalu i karakterizirana je slijedom instrukcija, stanjem i pridruženim skupom računalnih resursa. Za neki proces u izvršavanju, operacijski sustav stvara strukturu podataka zvanu **kontrolni blok procesa** (engl. *process control block*) koja se pohranjuje u memoriju računala. Elementi strukture su sljedeći:

- Identifikator – jedinstveni identifikator procesa kojim se razlikuje od ostalih procesa
- Stanje
- Prioritet

- Programski brojač – adresa instrukcije koja se iduća izvršava
- Memorijski pokazivači – pokazivači na dio programskog koda ili podatke asocirane s procesom
- Kontekstualni podaci – podaci u registrima procesora koji su potrebni za izvršavanje procesa
- I/O statusna informacije – uključuje I/O zahtjeve i uređaje pridružene procesu, listu datoteka koje proces koristi i sl.
- Statističke informacije – količina procesorskog vremena, vremenska ograničenja,…

1.2. Stvaranje i završetak procesa

Životni ciklus procesa omeđen je njegovim stvaranjem (engl. *creation*) i završetkom (engl. *termination*). Prilikom stvaranja novog procesa, operacijski sustav gradi strukture podataka kojima upravlja tim procesom te alocira adresni prostor u glavnoj memoriji za taj proces. Uobičajeno, četiri događaja uzrokuju stvaranje procesa:

- Novi slijedni posao (engl. *batch job*) – operacijski sustav sekvencijalno čita slijed naredbi s nekog toka podataka (magnetska traka ili disk)
- Interaktivna prijava – korisnik terminala se prijavi u sustav
- Kreiranje od stane operacijskog sustava za pružanje usluge – operacijski sustav može kreirati proces koji odrađuje posao u ime korisničkog programa, kako bi izbjegao čekanje korisnika
- Kreiranje od strane drugog procesa (engl. *process spawning*) – u tom slučaju proces koji je inicirao kreiranje se zove roditeljski proces (engl. *parent process*), a proces koji se kreira se naziva proces dijete (engl. *child process*).

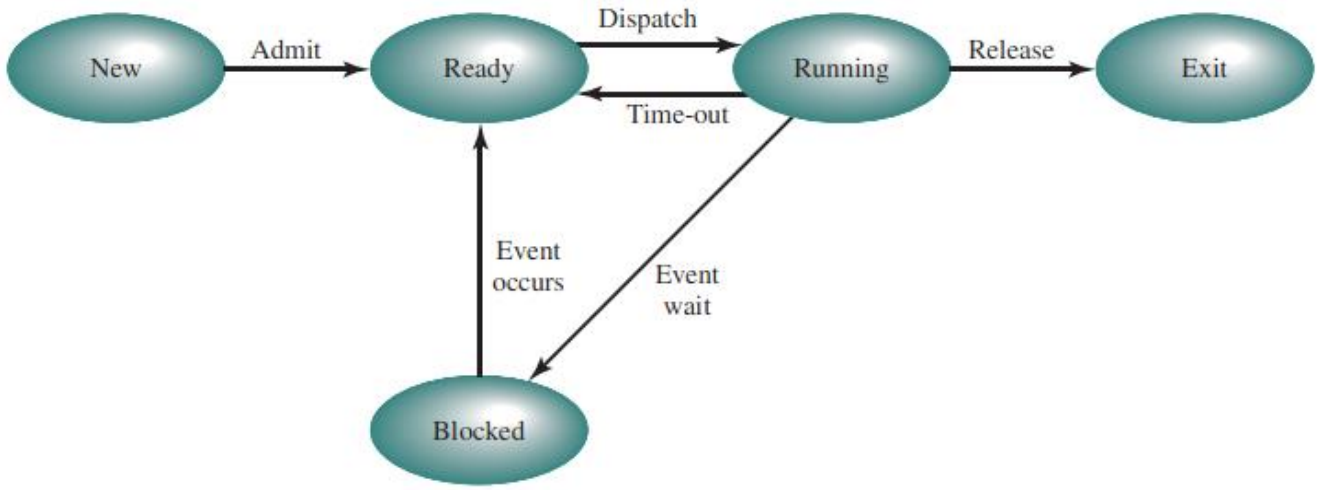
S druge strane, uobičajeni događaji za završetak procesa su:

- Normalni završetak
- Prekoračenje vremenskog ograničenja
- Nedostupnost memorije
- Narušavanje memorijskih granica – pokušaj pristupa nedozvoljenoj datoteci ili dijelu memorije
- Aritmetička greška
- Prekoračenje vremena za čekanje
- Greška prilikom ulaza ili izlaza
- Nepravilna instrukcija
- Privilegirana instrukcija
- Podatkovna greška
- Intervencija operatora ili operacijskog sustava
- Završetak roditeljskog procesa
- Zahtjev roditeljskog procesa za završetkom djeteta

1.3. Stanja procesa

Glavna odgovornost operacijskog sustava je kontroliranje izvršavanja procesa, što podrazumijeva određivanje njihovog rasporeda i alociranje potrebnih resursa. Procesi koji se ne izvršavaju su zadržani u redu čekanja. Tako možemo evidentno razlikovati 2 osnovna stanja procesa: izvršavanje i čekanje. Ukoliko neki od procesa u redu čekanja nisu spremni za izvršavanje ili su blokirani (čekaju na I/O operaciju), raspoređivač (tj. operacijski sustav) bi trebao pomno skenirati cijeli red čekanja kako bi pravedno odredio koji je proces sljedeći

na redu (npr. onaj koji nije blokiran i čeka najduže). U svrhu efikasnijeg raspoređivanja, procesi u redu čekanja se dodatno dijele u dva stanja: spremni i blokirani. U konačnici, Slika 2 prikazuje model s 5 stanja, najpoznatiji model stanja procesa unutar operacijskog sustava.



Slika 2. Model s 5 stanja procesa

Prema tom modelu, svaki proces može biti u nekom od sljedećih stanja:

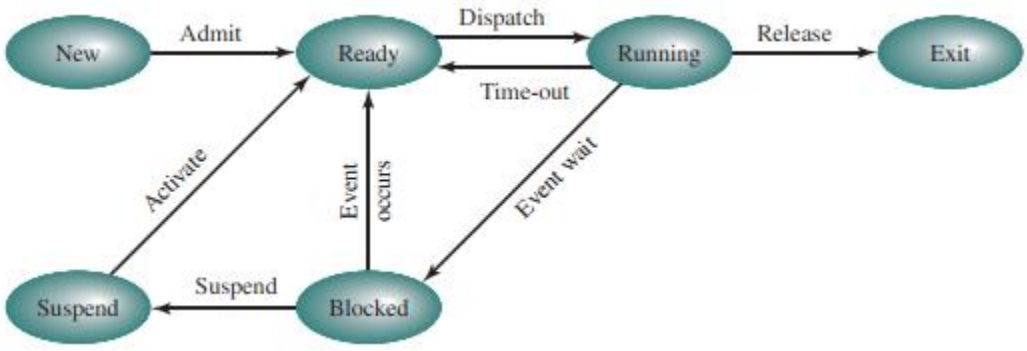
- Proces se izvršava (engl. *running*)
- Proces je spreman za izvršavanje (engl. *ready*)
- Proces je blokiran (engl. *blocked/waiting*) – proces se ne može izvršiti sve do pojave određenog događaja, npr. završavanje I/O operacije
- Proces je tek kreiran (engl. *new*) – novi proces još nije stavljen u glavnu memoriju, tj. u red čekanja, ali mu je kreiran kontrolni blok procesa
- Proces je završio (engl. *exit*) – proces je završio s radom ili je nasilno prekinut

Slika 1 također prikazuje vrste događaja koji vode ka tranziciji procesa između stanja:

- *New* → *Ready* – operacijski sustav će postaviti novi proces u red čekanja kad bude spreman ovisno o broju procesa koji već čekaju te o količini virtualne memorije pridijeljene svakom od tih procesa.
- *Ready* → *Running* – kada je operacijski sustav spreman preuzeti novi proces na izvršavanje, izabire jednog iz reda čekanja. Pri tome koristi podsustav operacijskog sustava zvani raspoređivač (engl. *scheduler/dispatcher*) koji izabire proces prema određenim pravilima raspoređivanja.
- *Running* → *Exit* – proces koji se trenutno izvršava može završiti ili biti nasilno prekinut od strane operacijskog sustava.
- *Running* → *Ready* – najčešći razlog za ovu tranziciju je da je proces u izvršavanju dostigao maksimalno vrijeme izvršavanje ili je na red došao proces većeg prioriteta, pa kažemo da je operacijski sustav istisnuo (engl. *preemted*) proces u izvršavanju.
- *Running* → *Blocked* – proces je postavljen u blokirano stanje ako zahtijeva nešto za što mora čekati.
- *Blocked* → *Ready* – proces se vraća iz blokirano stanje u stanje čekanja kada se dogodio događaj na kojeg je čekao.
- *Ready* → *Exit* – u nekim slučajevima proces može biti prekinut prije nego što se izvršio, najčešće kada se prekida roditeljski proces pa time i sva njegova djeca.
- *Blocked* → *Exit* – na sličan način kao i kod prethodne tranzicije, proces može biti prekinut dok je blokiranom stanju.

1.4. Potreba za razmjenom procesa

Staromodni operacijski sustavi su zadržavali procese u izvođenju cijelim dijelom u glavnoj memoriji. No, kada je neki proces blokiran on nepotrebno zauzima glavnu memoriju. Tada procesor ostaje neiskorišten čekajući da se taj proces odblokira. Na prvi pogled, ovaj problem je moguće riješiti proširenjem glavne memorije, no to povlači dodatan trošak. Osim toga, povećavanjem glavne memorije ne garantiramo da će u nju stati više procesa, nego može stati isti broj, ali većih procesa. Drugo poznato rješenje se naziva **razmjena procesa** (engl. *swapping*), koje uključuje premještanje dijela ili cijelog procesa iz glavne memorije na disk. Kada nijedan od procesa u glavnoj memoriji nije u stanju spremnosti (*Ready*), operacijski sustav premješta jedan od tih procesa na disk, a u glavnu memoriju učitava neki od spremnih procesa iz reda čekanja ili dozvoljava kreiranje novog procesa. Iako je razmjena procesa I/O operacija, koja je znatno sporija od rada procesora, disk je generalno najbrža ulazno-izlazna jedinica u sustavu, pa se tako razmjenom poboljšavaju performanse. Implementiranjem razmjene, potrebno je dodati još jedno stanje u model stanja: suspendirano stanje (engl. *Suspend*). Slika 3 prikazuje model stanja procesa proširen sa stanjem suspendiranosti. Proces je u tom stanju kada biva premješten na disk, a iz njega se može vratiti samo u stanje čekanja.

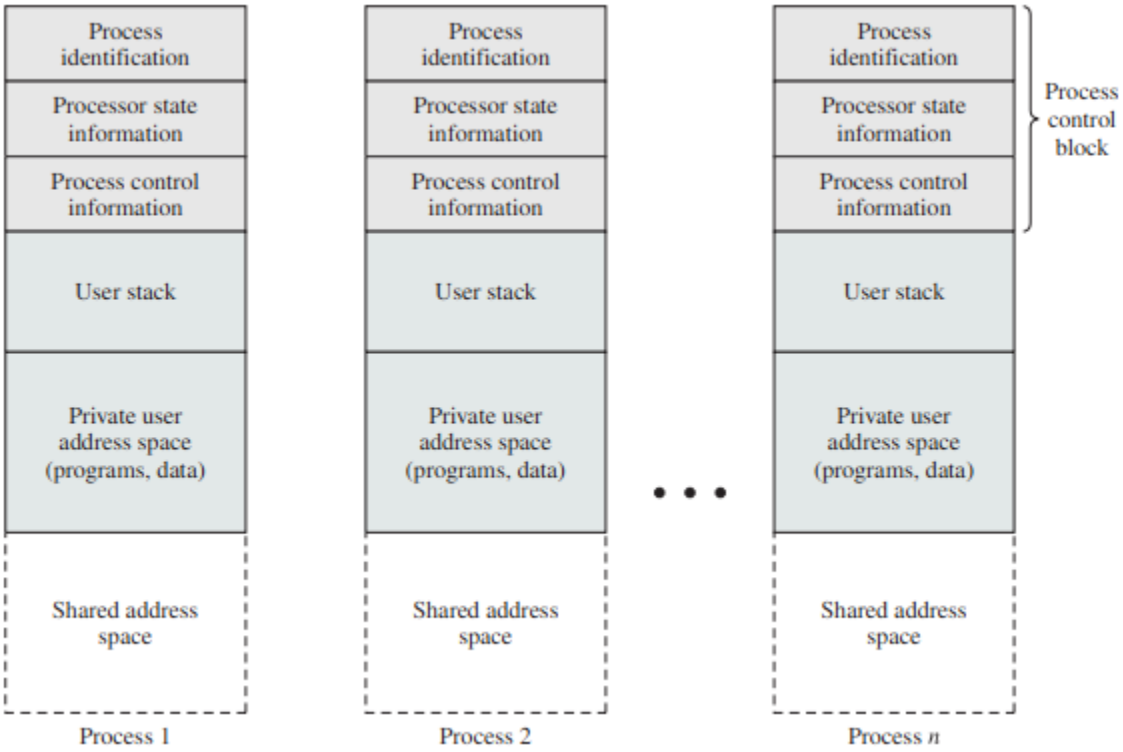


Slika 3. Model stanja procesa proširen sa stanjem suspendiranosti

1.5. Fizička reprezentacija procesa

Kako bi operacijski sustav upravljao procesima i njihovim resursima, mora imati informaciju o njihovom trenutnom statusu. Univerzalni pristup osiguravanja ove informacije je konstruiranje i održavanje procesnih tablica koje sadrže informacije o svim procesima u sustavu. Procesne tablice sadrže onoliko redaka koliko ima procesa, a svaki redak predstavlja **sliku procesa** (engl. *process image*). Slika procesa predstavlja njegovu fizičku manifestaciju, tj. konkretno definira strukture i tipove podataka koje predstavljaju proces u memoriji:

- Korisnički podaci – promjenjivi dio korisničkog prostora, mogu uključivati ulazne podatke u program i sl.
- Korisnički program – program koji se treba izvršiti
- Stog – stog pohranjuje parametre i adrese procedura i sistemskih poziva
- Kontrolni blok procesa – struktura podataka koju stvara operacijski sustav za identifikaciju procesa.



Slika 4. Struktura slike procesa

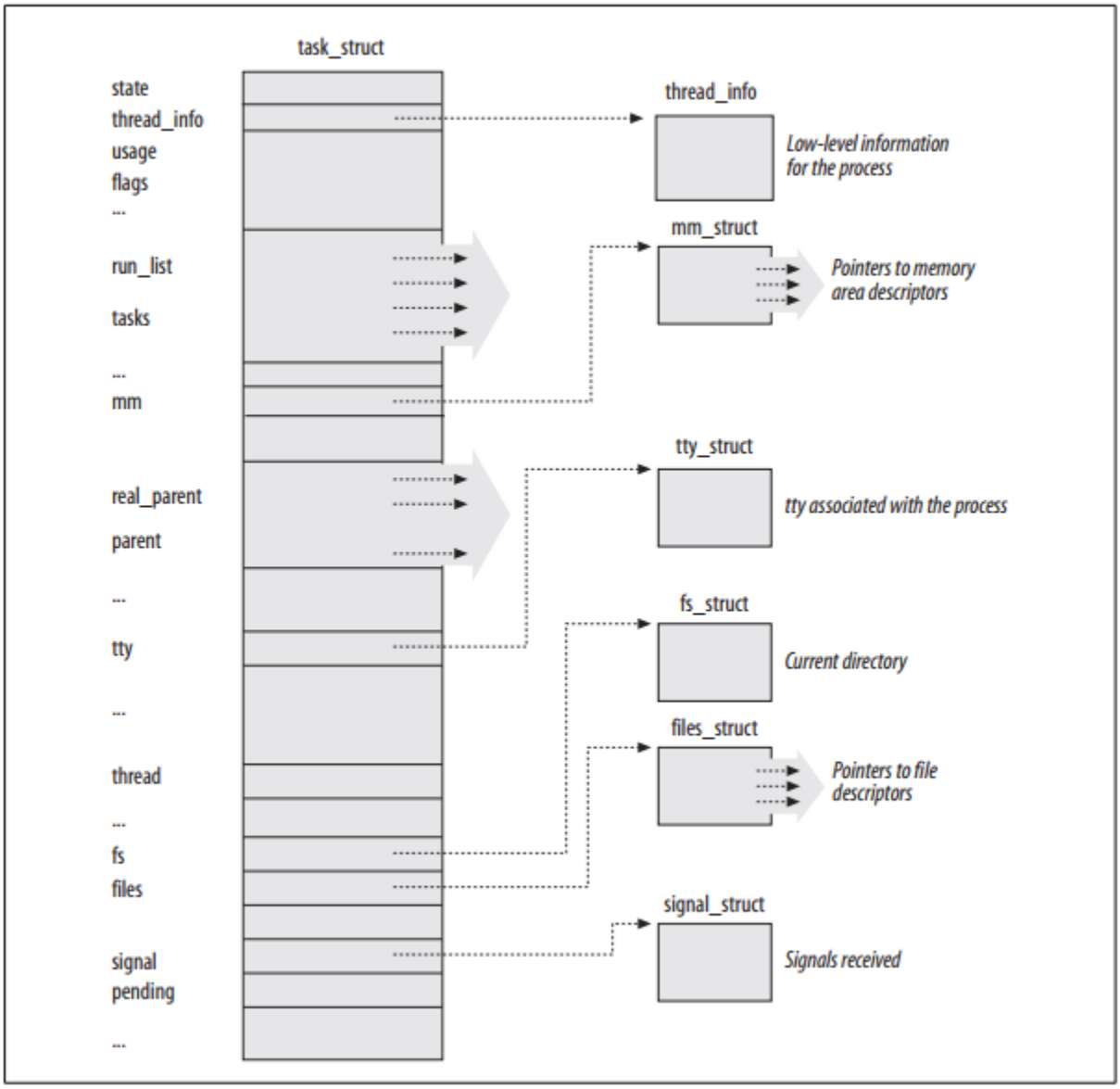
Svaki operacijski sustav na svoj način organizira strukturu i lokacije slike procesa, a kontrolni blok procesa je najpodložniji promjeni s obzirom da ga direktno koristi operacijski sustav, a ne proces. Tri su osnovne kategorije informacija koje su sadržane u kontrolnom bloku procesa: identifikatori procesa, informacije o stanju procesa te kontrolne informacije.

2. MEHANIZMI LINUX JEZGRE ZA UPRAVLJANJE PROCESIMA I NITIMA

Koncept procesa je fundamentalan za svaki operacijski sustav koji podržava istovremeni rad više programa (engl. *multiprogramming system*). U ovom poglavlju ćemo prikazati osnovne strukture podataka, funkcije, sustavske pozive i biblioteke pomoću kojih Linux jezgra upravlja procesima. Cijeli izvorni kod jezgre je javan, a većina koda odgovornog za upravljanje procesima i nitima je dio biblioteke *sched.h* (https://github.com/torvalds/linux/blob/master/include/linux/sched.h). Prije upuštanja u praktične primjere, prikazat ćemo načine na koji jezgra prezentira procese i odgovarajuće funkcije u programskom jeziku C. U praktičnim primjerima i zadacima, zadržat ćemo se samo u **korisničkom načinu rada**, preko **sustavskih poziva** i **naredbi ljuske**. Linux pruža dobro definiran i širok API za korištenje sustavskih poziva. Programski jezik C ne razumije koncept sustavskih poziva, no oni se uključuju u programski kod putem biblioteke ***linux/unistd.h*** Popis svih sustavskih poziva za upravljanje procesima se može pronaći na: http://linasm.sourceforge.net/docs/syscalls/process.php, a pozivi su podijeljeni na nekoliko kategorija upravljanja, npr. kreiranje procesa, identifikatori procesa, raspoređivanje procesa, niti, virtualna memorija i sl. Za ozbiljniji rad s Linux jezgrom (primjerice razvoj modula jezgre), potrebno je pristupiti jezgrenom načinu rada koji omogućuje pristup hardveru i mnogim zaštićenim operacijama.

2.1. Implementacija procesa

Kako bi mogla upravljati procesima, jezgra mora imati jasnu sliku svakog procesa. U tu svrhu, Linux svakom procesu definira njegov deskriptor, tj. strukturu podataka *task_struct* u programskom jeziku C. Slika 5 prikazuje sažeti sadržaj strukture *task_struct*, a cijela struktura se može pronaći na: https://github.com/torvalds/linux/blob/master/include/linux/sched.h. S obzirom da sadržava puno informacija te pokazivača na još nekoliko drugih struktura, struktura *task_struct* je iznimno kompleksna. Šest podatkovnih struktura izdvojenih na desnoj strani slike se odnose na specifične resurse koje proces posjeduje.



Slika 5. Sažeti prikaz strukture *task_struct*

2.1.1. Stanje procesa

Podatkovno polje *state* unutar strukture podataka *task_struct* opisuje trenutno stanje procesa. Tip podataka polja *state* je *volatile long*, a moguće vrijednosti su sljedeće:

- *TASK_RUNNING* – proces se trenutno izvršava ili čeka na izvršavanje
- *TASK_INTERRUPTIBLE* – proces je suspendiran sve dok se ne ispune odgovarajući uvjeti
- *TASK_UNINTERRUPTIBLE* – kao i prethodna vrijednost, no ne može biti prekinut u čekanju
- *TASK_STOPPED* – izvršavanje procesa je završilo
- *TASK_TRACED* – izvršavanje procesa je zaustavio debugger
- *EXIT_ZOMBIE* – izvršavanje procesa je prekinuto, no roditeljski proces još nije zatražio informaciju o prekinutom procesu
- *EXIT_DEAD* – konačno stanje; proces je završen i izbačen iz sustava

Vrijednost *state* polja se može dohvatiti preko pokazivača na strukturu *task_struct*. Linux jezgra također koristi *set_task_state* i *set_current_state* makroe, koji postavljaju stanje određenog procesa ili procesa koji se trenutno izvršava. Ovi makroi osiguravaju operaciju pridruživanja stanja procesa.

2.1.2. Identifikacija procesa

Generalno pravilo raspoređivanja procesa je da svaki kontekst izvršavanja (proces), mora imati svoj deskriptor. Striktan jedan-na-jedan odnos između procesa i njegovog deskriptora definira 32-bitna adresa strukture *task_struct*, po kojoj jezgra razlikuje procese. S druge strane, kako bi korisnik mogao razlikovati procese, svakom procesu je dodijeljen jedinstveni identifikator **PID** (engl. *Process ID*), koji je pohranjen u polju *pid* (tip podataka *pid_t*). Identifikatori procesa su određeni sekvencijalno, po rasporedu kreiranja. Gornja granica PID vrijednosti (u skladu s ograničenjem tipa podataka *int*) je 32,767 (PID_MAX_DEFAULT-1) u 32-bitnim arhitekturama ili 4,194,303 u 64-bitnim arhitekturama, a administrator sustava može smanjiti to ograničenje u datoteci */proc/sys/kernel/pid_max*. Prilikom recikliranja PID brojeva, jezgra održava polje podataka *pidmap_array* koje označava koji PID-ovi su trenutno zauzeti, a koji su oslobođeni. Linux jezgra asocira različiti PID za svaki laki proces u sustavu, dok UNIX svim nitima unutar grupe dodjeljuje isti PID. Prema POSIX 1003.1c standardu, sve niti u višenitnoj aplikaciji moraju imati isti ID. U svrhu udovoljenja standardu, Linux stvara grupe niti pri čemu sve niti u grupi dijeli PID voditeljske niti (engl. *thread group leader*) koji se sprema u *tgid* polje.

Proces ***idle***, kojeg jezgra pokreće kad nema nijednog procesa koji se izvršava ima PID 0. Prvi proces kojeg jezgra pokreće je ***init***, s PID-om 1. Programski, ID procesa je reprezentiran tipom podataka *pid_t*, koji je definiran u biblioteci <*sys/types.h*>. U programskom jeziku C, *pid_t* je zapravo *typedef* na tip podataka *int*. Dohvaćanje ID-a nekog procesa ili ID-a njegovog roditeljskog procesa izvodi se pomoću funkcija *getpid()* i *getppid*(), kako je pokazano u Primjeru 1. Povratna vrijednost funkcija se *cast*-a u *intmax_t* tip podataka, što je C/C++ tip koji garantira pohranjivanje najveće moguće vrijednosti cijelog broja u tim jezicima.

Primjer 1: Dohvaćanje PID-a i PPID-a

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);

printf("Moj PID=%jd\n", (intmax_t) getpid());
printf("PID roditelja=%jd\n", (intmax_t) getppid ());
```

2.1.3. Lista procesa

Linux jezgra prati postojeće procese upotrjebljujući poznatu strukturu podataka – dvostruko povezani popis (engl. *doubly linked lists*). Takav povezani popis čiji su elementi strukture deskriptora procesa se naziva lista procesa (engl. *process list*). Struktura *task_struct* sadrži polje *tasks* tipa *list_head*, koji se sastoji od pokazivača *prev* i *next* na strukturu *task_struct*. Glava procesne liste se naziva *init_task*, a predstavlja deskriptor procesa 0. Polje *tasks->prev* pokazuje na zadnji proces u listi.

2.1.4. Odnosi između procesa

Procesi kreirani programom imaju roditelj-dijete odnos. Kada proces kreira više procesa djece, oni imaju odnos kao braća ili sestre (engl. *siblings*). Nekoliko polja u deskriptoru procesa opisuju ove odnose, a prikazani su u tablici 1.

Tablica 1. Polja deskriptora procesa za definiranje odnosa roditelj-dijete

Podatkovno polje	Tip podataka	Opis
real_parent	*task_struct**	Pokazuje na deskriptor onog procesa koji ga je kreirao ili na proces *init*, ukoliko roditelj više ne postoji u sustavu
parent	*task_struct**	Pokazuje na trenutnog roditelja, tj. proces koji ga je pozvao. Većinom je isti kao *real_parent*, no može biti drugačije (npr. prilikom debugiranja)
children	*list_head*	Glava liste u kojoj se nalaze svi procesi djeca od procesa čiji član se gleda
sibling	*list_head*	Pokazivači na sljedeće i prijašnje elemente u listi bratskih procesa, tj. onih koji imaju istog roditelja

Nadalje, postoje dodatni odnosi između procesa koji nastaju uslijed grupiranja procesa. Primjerice, proces može biti voditelj grupe procesa, sesije za prijavu, voditelj grupe niti ili može pratiti trag (engl. *trace*) izvršavanja drugih procesa.

2.2. Kreiranje procesa

Linux/UNIX operacijski sustavi uvelike se oslanjaju na kreiranje novih procesa radi zadovoljavanja korisničkih zahtjeva. Tradicionalni UNIX sustavi tretiraju sve procese na isti način: resursi roditeljskog procesa se dupliciraju u proces dijete. Takav pristup čini kreiranje procesa vrlo sporim i neefikasnim, jer zahtijeva kopiranje cijelog adresnog prostora roditelja, a proces dijete zapravo rijetko treba sve resurse roditelja. Moderne UNIX jezgre rješavaju ovaj problem pomoću tri mehanizma:

- Tehnika *Copy On Write* – omogućuje i roditelju i djetetu čitanje istih fizičkih stranica. Kad jedan od njih poželi pisati na fizičku stranicu, jezgre kopira njen sadržaj u novu stranicu koja se dodjeljuje procesu pisanja.
- Laki procesi (engl. *lightweight*) – omogućuju i roditelju i djetetu dijeljenje podatkovnih struktura jezgre, kao što su tablice straničenja, otvorene datoteke, signale i sl.
- Sustavski poziv *vfork()* – kreira proces koji dijeli adresni prostor roditelja. Za sprječavanje roditelja da prepiše podatke potrebne djetetu, izvršavanje roditelja se blokira sve dok postoji dijete.

2.2.1. Sustavski poziv *fork()*

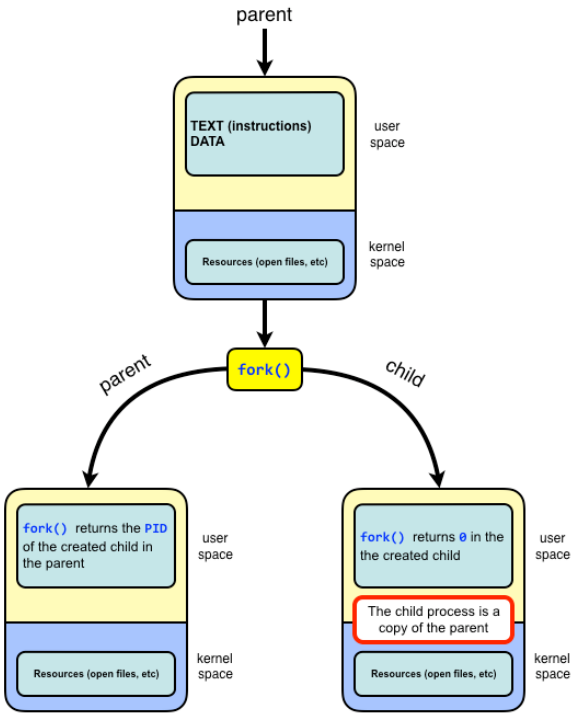
U Linux jezgri, sustavski poziv *fork()* je najpoznatiji sustavski poziv za kreiranje procesa, a implementiran je kroz korištenje *Copy-on-write* stranica. Umjesto dupliciranja adresnog prostora, roditeljski proces i proces dijete dijele pokazivače na adrese u memoriji. Međutim, ako jedan od njih odluči izvesti operaciju pisanja, stvara se duplikat memorijskog bloka na kojeg želi pisati i svakom procesu se dodjeljuje jedinstvena kopija.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Tradicionalni sustavski poziv *fork()* je implementiran u Linuxu kao sustavski poziv *clone()*, sa specifičnim zastavicama. No, zahvaljujući *Copy-on-write* mehanizmu, roditelj i dijete dobivaju različiti korisnički stog. Nakon što je proces dijete kreiran, oba procesa će izvršiti iduću

instrukciju nakon sustavskog poziva *fork()*. Sustavski poziv *fork()* ne prima parametre, a vraća tri kategorije cjelobrojnih vrijednosti:

- Negativna vrijednost - kreiranje djeteta je neuspješno
- 0 – izvođenje se vraća novokreiranom procesu dijete
- Pozitivna vrijednost – izvođenje se vraća roditelju; povratna vrijednost je PID novokreiranog procesa dijete



Slika 6. Dijagram sustavskog poziva *fork()*

Jednostavan primjer uporabe sustavskog poziva *fork()* je prikazan u Primjeru 2.

Primjer 2: Jednostavna upotreba sustavskog poziva fork()

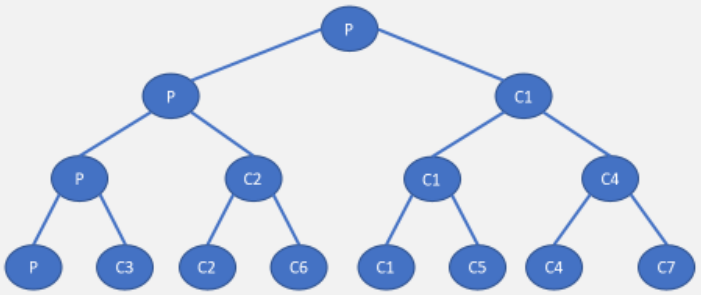
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
  // Kreiramo 2 procesa koji izvršavaju istu naredbu nakon poziva fork()
  fork();
  printf("Hello world!\n");
  return 0;
}

Output:
Hello world!
Hello world!
```

Ukupan broj procesa koji se kreira na ovaj način je 2^n, gdje je n broj sustavskih poziva *fork()*. U Primjeru 2 to je $2^1 = 2$. Primjer 3 prikazuje **stablo procesa** za slučaj tri sustavska poziva *fork()*. U tom slučaju ukupno se kreira $2^3 = 8$ procesa djece, jer svaki proces dijete izvršava iduću naredbu nakon sustavskog poziva kojim je on kreiran. Ako je ta sljedeća naredba

također poziv *fork()*, onda i taj proces stvara djecu. U primjeru su prikazana 2 stabla procesa, pri čemu prvo predstavlja broj novih procesa po liniji, a drugo stvarnu hijerarhiju procesa.

Primjer 3: Dijagram sustavskog poziva fork()

```
fork ();  // Linija 1
fork ();  // Linija 2
fork ();  // Linija 3
```

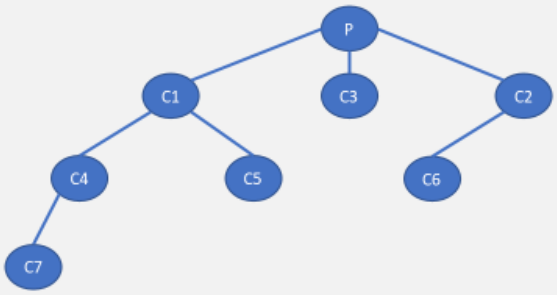


// 1 proces roditelj = funkcija main

// 1 proces dijete u liniji 1

// 2 dječja procesa u liniji 2

// 4 dječja procesa u liniji 3



// Glavni proces (roditelj)

// Procesi kreirani u 1. forku: P1

// Procesi kreirani u drugom forku: C2, C4

// Procesi kreirani u trećem forku: C3, C5, C6, C7

Proces dijete i proces roditelj se izvršavaju **istovremeno (engl. *concurrently*)** na procesu računala. To podrazumijeva da se procesi istovremeno natječu za resurse računala (procesor, memorija, disk, …). Operacijski sustav odlučuje o tome kojem će procesu prvome pridijeliti kontrolu. Kako bi programski utvrdili koji proces je trenutno u izvršavanju potrebno je provjeriti povratnu vrijednost sustavskog poziva *fork()*. Primjer 4 prikazuje na koji način možemo pridijeliti različiti programski kod procesima roditelj i dijete na izvršavanje.

Primjer 4: Programski kod specifičan za procese roditelj i dijete

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()  {
  int x = 1;
  if (fork() == 0)
    printf("Proces dijete x = %d\n", ++x);
  else
    printf("Proces roditelj x=%d\n", --x);
}
```

Output:

```
  Proces roditelj x=0
  Proces dijete x=2
    ili
  Proces dijete x=2
  Proces roditelj x=0
```

2.2.2. *Fork* bomba

Fork bomba je program koji šteti sustavu tako što ga ostavlja bez memorije. Program u beskonačnoj petlji poziva sustavski poziv *fork()* kako bi ispunio memoriju. *Fork* bomba je oblik napada uskraćivanja usluge (engl. *Denial-of-Service (DOS)*) protiv sustava utemeljenog na Linuxu. Jednom kad se uspješna *Fork* bomba aktivira, možda neće biti moguće nastaviti normalan rad bez ponovnog pokretanja sustava, jer je jedino rješenje uništiti sve instance kreiranih procesa. Primjer 5 pokazuje jednostavan programski kod za stvaranje takvog oblika napada. **Ovaj primjer ne morate pokretati na računalu.**

Primjer 5: Jednostavna Fork bomba

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
  while(1)
    fork();
  return 0;
}
```

2.2.3. Sustavski poziv *vfork()*

Sustavski poziv *vfork()* obavlja isti posao kao i *fork()*, osim što se unosi iz tablice straničenja roditeljskog procesa ne kopiraju. Umjesto toga, proces dijete izvršava vlastitu nit u adresnom prostoru roditelja, a roditelj biva blokiran sve dok dijete ne pozove *exec()* ili završi. Proces dijete ne smije pisati po adresnom prostoru roditelja. Sustavski poziv *vfork()* je implementiran kao sustavski poziv *clone()* s posebnim skupom zastavica.

```
#include <sys/types.h>
#include <unistd.h>

pid_t vfork(void);
```

2.3. Izvršavanje procesa

U operacijskom sustavu Linux kreiranje procesa se razlikuje od učitavanja procesa u memoriju te njegova izvršavanja. Jedan sustavski poziv učitava binarnu sliku programa u memoriju, zamjenjuje prijašnji sadržaj na tom adresnom prostoru te započinje izvršavanje novog programa. Takva cjelokupna funkcionalnost se naziva **izvršavanje** (engl. *executing*) programa, a omogućena je putem *exec()* obitelji sustavskih poziva. Postoji više članova te obitelji, pri čemu je najjednostavniji poziv *execl()*.

```
#include <unistd.h>

int execl (const char *path, const char *arg, …);
```

Pozivanje *execl()* poziva zamjenjuje sliku trenutnog procesa s novom slikom koja se učitava u memoriju na lokaciju *path*. Parametar *arg* je prvi argument novog programa. Primjer 6 pokazuje upotrebu sustavskog poziva *execl()*. Prvi parametar je putanja programa koji se želi pokrenuti, a drugi naziv tog programa. Ovaj program pokreće uređivač teksta vi ili ispisuje da je došlo do greške pri pokretanju sustavskog poziva *execl()*.

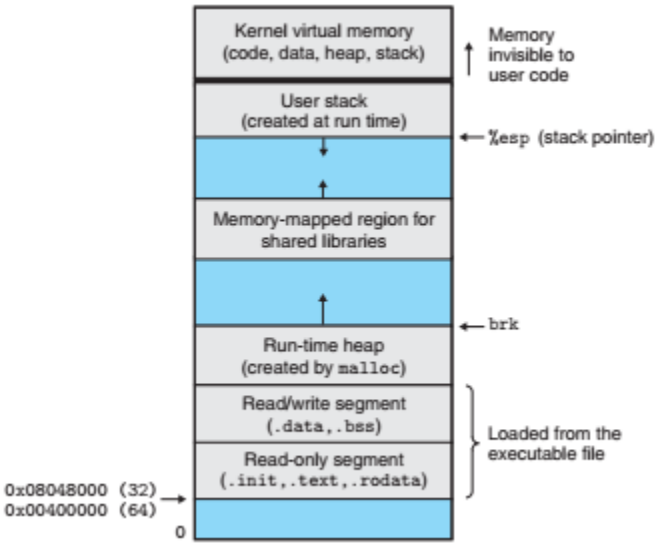
Primjer 6: Primjer upotrebe sustavskog poziva execl

```
#include <unistd.h>
int ret;
ret = execl ("/bin/vi", "vi", NULL);
if (ret == -1)
  perror ("execl")
```

Ostali sustavski pozivi u *exec()* obitelji su *execlp*, *execv*, *execvp*, *execve*.

2.3.1. Učitavanje procesa u radnu memoriju računala

Operacijski sustav svakom procesu u memoriji pruža iluziju da ima ekskluzivno pravo na korištenje cijelog adresnog prostora sustava. Međutim, svaki proces ima svoj privatni adresni prostor koji je po strukturi jednak adresnom prostoru radne memorije. Koncept operacijskog sustava zvan **virtualna memorija** vodi računa o mapiranju više privatnih adresnih prostora procesa u adresni prostor računala (*Ovaj koncept će se proučiti u više detalja u sljedećim laboratorijskim vježbama*). Slika 7 prikazuje privatni adresni prostor procesa kreiranog unutar operacijskog sustava x86-64 Linux.



Slika 7. Privatni adresni prostor procesa unutar operacijskog sustava Linux

2.4. Završavanje procesa

Većina procesa "umire" na način da završe izvršavanje svojeg koda. Kada se to dogodi, jezgra mora biti obaviještena kako bi mogla otpustiti resurse procesa. Standardni način završetka procesa u Linuxu je putem funkcije *exit()*, koji otpušta resurse alocirane iz C biblioteka, izvršava funkcije koje su predviđene za izvršavanje po završetku prekinutog procesa te poziva sustavski poziv koji uklanja proces iz sustava. Funkcija *exit()* može biti implicitno ubačena od strane programera, no C kompajler uvijek postavlja *exit()* funkciju na kraju svake *main()* funkcije.

U Linuxu postoje dva sustavska poziva za završetak aplikacije koja se pokreće u korisničkom načinu rada:

1. Sustavski poziv *exit_group()* – prekida cijelu grupu niti, tj. cijelu višenitnu aplikaciju. Osnovna funkcija jezgre koja implementira ovaj sustavski poziv je *do_group_exit()*.
2. Sustavski poziv *exit()* – prekida jedan proces, neovisno o drugim procesima u grupi. Osnovna funkcija jezgre koja implementira ovaj sustavski poziv je *do_exit()*.

```
#include <stdlib.h>

void exit (int status);
```

Sustavski poziv *exit()* je dio biblioteke *stdlib.h*, nema povratnu vrijednost, a prima jedan parametar - *status*, koji označava izlazni status procesa. Ostali programi provjeravaju tu vrijednost. Najčešći statusi su zastavice *EXIT_SUCCESS* I *EXIT_FAILURE* koje označavaju uspješno, tj. neuspješno uništavanje procesa. U operacijskom sustavu Linux, 0 tipično predstavlja uspjeh, a vrijednost različita od 0 neuspjeh pri završetku procesa.

2.5. Čekanje na završetak procesa

Kada proces završi, jezgra šalje signal *SIGCHLD* roditelju. Kao što su sustavski pozivi način komunikacije korisničkih aplikacija s jezgrom, tako su signali u operacijskom sustavu način komunikacije jezgre s korisničkim aplikacijama. Uobičajeno, signal *SIGCHLD* se ignorira te se

ne poduzima nikakva akcija. Signal *SIGCHLD* se može generirati i poslati u bilo koje vrijeme, jer je završetak procesa djeteta asinkron u odnosu na roditelja. Međutim, vrlo često roditelj želi znati više o završetku djeteta ili čak eksplicitno čekati na njegov završetak. Ako se dogodi da proces dijete završi prije nego njegov roditelj, operacijski sustav Linux ga postavlja u posebno stanje - ***zombie***. Proces u ovom stanju čeka da njegov roditelj zatraži njegov status. Tek nakon što roditelj dobije informacije o završetku djeteta ono formalno završava i briše se iz sustava. Čekanje na završetak djeteta je moguće postići korištenjem sustavskog poziva *wait()*, koji vraća PID završenog djeteta ili -1 za grešku. Sve dok proces dijete ne završi, roditelj biva blokiran. Parametar *status* sadrži dodatne informacije o djetetu.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *status);
```

Često proces roditelj ima više djece, a želi čekati završetak specifičnog djeteta. U tu svrhu se koristi sustavski poziv *waitpid()* koji dodatno prima PID procesa na kojeg je potrebno čekati. Moguće vrijednosti *pid* parametra su:

- manje od -1 - čekanje na bilo koje dijete čiji je ID grupe jednak apsolutnoj vrijednosti *pid* parametra
- -1 - čekanje na bilo koji proces dijete (kao *wait*)
- 0 - čekanje na bilo koji proces dijete koji pripada istoj grupi kao i pozivajući proces
- veće od 0 - čekanje na proces dijete čiji je PID jednak predanoj vrijednosti *pid* parametra

Moguća je također i situacija da roditelj završi prije nego proces dijete. U tom slučaju, jezgra operacijskog sustava postavlja proces *init* za roditelja tog djeteta. Proces dijete čiji je roditelj završio prije njega se naziva proces ***orphan.***

2.6. „Uspavljivanje“ procesa

Sustavski poziv *sleep* suspendira proces na specifični vremenski period. Funkcija *sleep* vraća 0 kada je proteklo specificirano vrijeme, a u suprotnom vraća preostali broj sekundi za spavanje. Potonji slučaj je moguć ako je *sleep* funkcija završila preuranjeno ili je prekinuta signalom.

```
#include <unistd.h>

unsigned int sleep (unsigned int secs);
```

2.7. Primjene "*multiprogramming*" koncepta

U modernom operacijskom sustavu, uobičajeno je da istovremeno postoji nekoliko konkurentnih procesa koji se izvršavaju. Za operacijski sustav koji omogućava takvo izvršavanje kažemo da slijedi *multiprogramming* koncept. Odgovornost operacijskog sustava je da upravlja tim procesima efikasno i efektivno. Pokazano je da u operacijskom sustavu Linux, sustavski poziv *fork()* služi za kreiranje novih procesa koji se istovremeno natječu za računalne resurse. Drugim riječima, taj sustavski poziv je temelj za provođenje *multiprogramming* koncepta. Najčešći primjeri programa iz prakse koji se u svom radu oslanjanju na taj pristup su:

- Ljuska operacijskog sustava - koristi *fork()* za pokretanje programa na temelju ulaza iz komandnog sučelja

- Web poslužitelji - koriste *fork()* za kreiranje višestrukih poslužiteljskih procesa, pri čemu svaki rukuje zahtjevima u vlastitom adresnom prostoru
- Web preglednici - koriste *fork()* za rukovanje svakom stranicom kao odvojenim procesom
- Paralelno programiranje - sustavski poziv *fork()* se koristi za pokretanje procesa u paralelnim bibliotekama (npr. MPI)
- Skriptni jezici - koriste *fork()* za indirektno pokretanje procesa djece.

Primjer 6 prikazuje program koji implementira pojednostavljenu ljusku operacijskog sustava. Ljuska čeka na podatke sa standardnog ulaza (konzole), koje zatim čita te prosljeđuje procesu djetetu.

Primjer 6: Primjer pojednostavljene ljuske

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <string.h>

#define BUFFER 1024
int main(void) {
  char line[BUFFER];
  while(1) {
    printf("$ ");
    if(!fgets(line, BUFFER, stdin)) break;   // Dohvaćanje podataka s konzole
    char *p = strchr(line, '\n');
    if (p) *p = 0;         // Uklanjanje znaka za prelazak u novi red
    if(strcmp(line, "exit")==0) break;     // Izvođenje naredbe exit
    char *args[] = {line, (char*) 0};
    int pid = fork();                  // Kreiranje procesa koji će izvesti novu naredbu
    if(pid==0){
      execvp(line, args);    // Izvršavanje naredbe
      perror("exec");       // Ispis u slučaju greške u izvršavanju
      exit(1);
    } else {
      wait(NULL);
    }
}
  return 0;
}
```

3. MEHANIZMI LINUX LJUSKE ZA UPRAVLJANJE PROCESIMA I NITIMA

Već je spomenuto kako su procesi programi u izvršavanju i da njima upravlja ljuska operacijskog sustava. Za njihovu identifikaciju, dodjeljuje im se jedinstveni identifikator procesa, PID, po poretku njihova nastajanja. Upravljanje procesima i nitima u operacijskim sustavima UNIX/Linux je iznimno kompleksan posao kojeg u cijelosti obavlja jezgra upotrjebljujući mnoštvo struktura podataka, funkcija i biblioteka. Ipak, kako bi „običan" korisnik tih operacijskih sustava mogao imati djelomičan uvid u procese koje se izvršavaju na njegovom računalu, korisničko sučelje, tj. ljuska operacijskog sustava pruža odgovarajuće naredbe koje to omogućuju. Takav skup naredbi se može najbliže usporediti s upraviteljem resursa (engl. *task manager*) operacijskog sustava Windows.

Prilikom pokretanja sustava prvi proces koji se pokreće je *init* (engl. *Initialization*) s PID-om 1, koji inicijalizira ostale procese. Procesi se dijele po nekoliko kriterija:

- *daemon*: proces koji postoji zbog specifične uloge (npr apache deamon za http servis), slično kao NT servisi, pokreću se u pozadini i neaktivni su dok ih se ne pozove
- proces roditelj (engl. *Parent*): proces koji kreira druge procese, svaki proces osim procesa init ima roditeljski proces
- proces dijete (engl. *Child*): proces pokrenut od drugog, roditeljskog procesa koji ima oznaku PPID (engl. *Parent PID*)
- *orhpan*: aktivni proces čiji je roditeljski proces završen, takav proces preuzima init proces koji mu postaje roditeljski proces
- *zombie* (engl. *Defunct*): proces dijete koji se sa svojim izlaznim podacima ne vraća roditeljskom procesu i ostaje „izgubljen" u sustavu, može se izbrisati iz tablice procesa jedino ponovnim pokretanjem OS-a

Lista pokrenutih procesa u ljusci operacijskog sustava Linux, može se dobiti pozivanjem naredbe **ps**.

ps
(*report a snapshot of the current process* – prikazuje listu procesa)

Koristi se za dobivanje brzog uvida u popis trenutnih procesa. Sintaksa naredbe: **ps** [-option]. Dobiveni izlaz je u obliku tablice s informacijama o:

- PID-u procesa
- TTY – kontrolni terminal asociran procesu
- STAT – statusni kod procesa
- TIME – ukupno vrijeme korištenja procesa
- CMD – naziv naredbe ljuske

Mnoge su opcije koje se mogu pozvati s naredbom **ps**, a lista opcija se može pronaći u stranicama priručnika. Neke od često korištenih opcija su: -a (prikaz svih procesa u izvršavanju od svih korisnika), -u (detaljnije informacije o procesima) te –x (lista svih procesa koji nemaju pridružen TTY).

top
(*display Linux processes* – prikazuje procese sustava)

Naredbom *top* mogu se u stvarnom vremenu, s mogućnošću podešavanja intervala osvježenja, prikazivati podaci o procesima i ostalim parametrima rada sustava. Sintaksa naredbe: **top** [-option]. Pregled se može posložiti po vrsti podatka za koji želimo prikazati listu procesa.

kill
(*send a signal to a process* – šalje signal procesu)

Koristi se za slanje odgovarajućeg signala procesu, najčešće signala za 'ubijanje' procesa. Sintaksa naredbe: **kill** [-option] **[signal]**. Ako se za vrijednost signala postavi cijeli broj, on se promatra kao PID procesa kojeg je potrebno „ubiti“, npr. *kill 1234*. Prema uobičajenim postavkama, šalje se signal *SIGTERM* procesu kojeg se želi prekinuti i dozvoljava mu se oslobađanje njegovih resursa i spremanje stanja. S druge strane, moguće je specificirati i signal uz naredbu **kill**, npr. *kill -9 1234*, pri čemu -9 označava signal za *sure kill*, tj. sigurno ubijanje procesa. Druge moguće vrijednosti signala su:

- SIGHUP – šalje se procesu kada se zatvara terminal u kojem je on pokrenut
- SIGINT – prekidni signal za „lagano ubijanje“ procesa, npr. kombinacijom tipaka Ctrl+C
- SIGTERM – ubijanje procesa uz dozvoljeno čišćenje i spremanje stanja
- SIGKILL – ubijanje procesa, bez mogućnosti za čišćenjem i spremanjem
- SIGSTOP – stopiranje/suspendiranje procesa

Ubijanjem roditeljskog procesa ubijaju se i procesi koje je taj proces kreirao. Izlistanje stabla procesa može se dobiti korištenjem naredbe **pstree**.

Informacije o procesima su pohranjene u datotečnom sustavu Linux operacijskog sustava, točnije u */proc* direktoriju. Za svaki PID postoji poddirektorij, a unutar poddirektorija se nalaze datoteke s podacima o procesu. Primjerice, status procesa 1234 se može pogledati naredbom: *cat /proc/1234/status*.

Prema uobičajenim postavkama, procesi koji se pokreću upisivanjem naredbe u ljusku operacijskog sustava su blokirajući procesi, tj. korisnik ne može činiti ništa dok oni ne završe. Srećom, to je moguće kontrolirati kroz samu ljusku, tako da se proces postavlja u jedno od dva stanja: **prednje** (engl. *foreground*) i **pozadinsko** (engl. *background*). Proces se pokreće u pozadini tako da se stavlja znak & na kraj naredbe koja ga pokreće, npr. *sleep 1000* &. Naredba *jobs* ispisuje sve poslove koji se obavljaju u pozadini a koji su pokrenuti sa tog terminala/prozora, te njihov status.

jobs
(*display status of jobs in the current session* – prikazuje status poslova u trenutnoj ljusci)

Ako je posao pokrenut u pozadini (stavljanjem znaka & na kraju naredbe) terminal/prozor je slobodan za novi upis naredbe, a posao se izvršava u pozadini. Može se pokrenuti više pozadinskih poslova. Svaki pokrenuti posao dobiva svoj jedinstveni broj koji se ispisuje u uglatim zagradama prilikom pokretanja posla u pozadini. Osim broja posla ispisuje se i identifikacijski broj procesa (PID). Nadalje, oznaka + označava da je taj proces posljednji pokrenut u pozadini, a – drugi najnoviji. Za premještanje poslova iz pozadine u blokirajuće stanje ili obratno, moguće je koristiti razne kombinacije tipaka ili naredbi:

- Zaustavljanje posla
  - *^Z* - posao se može zaustaviti s Ctrl+Z. Na taj način se posao zaustavlja i može se ponovno pokrenuti u pozadini naredbom **bg** ili u prvom planu s naredbom **fg**
  - ***stop*** *%job* - posao koji se odvija u pozadini može se zaustaviti naredbom **stop** i brojem posla ispred kojeg se piše znak %
- Ponovno pokretanje posla
  - *fg %job* - naredba **fg** koristi se za ponovno pokretanje u prvom planu zaustavljenog posla ili posla koji se odvija u pozadini
  - *bg %job* - naredba **bg** se koristi za ponovno pokretanje zaustavljenog posla u pozadini
- Prekid posla

- ^*C (Ctrl-C)* - posao koji se trenutno odvija u prvom planu može se trajno prekinuti s Ctrl+C
- *kill %job* - poslovi koji se odvijaju u pozadini mogu se zaustaviti naredbom **kill**

4. ZADACI ZA SAMOSTALNI RAD

Pristupiti poslužitelju linux.etfos.hr pomoću Putty SSH klijenta. Sljedeće zadatke riješite koristeći uređivač teksta nano, sintaksu programskog jezika C i Bash ljusku operacijskog sustava Linux. Za kompajliranje programa napisanih u C jeziku koristite GCC kompajler koji je već instaliran na poslužitelju. Za svaki zadatak kopirajte programski kod, naredbe ljuske i dobiveni izlaz u Zadaću. Ako je potrebno, napišite dodatno pojašnjenje Vašeg rješenja.

1. Koristeći sustavski poziv fork() kreirajte program kako biste stvorili dječji proces i omogućili roditeljskom procesu da ispisuje „roditelj“, a djetetu da ispisuje „dijete“ na ekranu. Dodatno, oba procesa moraju ispisati svoj PID i PPID.

2. Napišite program u kojem proces roditelj kvadrira parne brojeve, a proces dijete neparne brojeve nekog niza cijelih brojeva. Oba procesa ispisuju novi niz.

3. Napišite program koji simulira procese Zombie i Orphan koristeći sustavske pozive fork(), sleep() i exit(). Zombie proces je proces dijete koji završava s radom prije svog roditelja. Orphan proces je proces dijete čiji je roditelj završio s radom prije njega.

4. Koristeći sustavske pozive fork(), execv() i wait() napišite program koji pokreće naredbu Linux ljuske ls (putanja do izvršne datoteke je "/bin/ls") .

5. U ljusci pokrenite naredbu sleep 1000 u pozadini. Pokrenite naredbu za izlistavanje svih podataka o pokrenutim procesima. Koji je PID a koji PPID procesa sleep 1000 koji ste upravo pokrenuli? Na koji ćete način odrediti koji je proces roditelj navedenog procesa (sleep 1000)?

4.1. Zadaci za vježbu

Ove zadatke nije potrebno riješiti tijekom laboratorijske vježbe. Oni služe isključivo za vježbanje za pismeni ispit, a svoja rješenja možete provjeriti na konzultacijama.

1. U programskom kodu iz zadatka 3. za samostalni rad zamijenite sustavski poziv *fork()* sa sustavskim pozivom *vfork()*. Kako bi *vfork()* ispravno radio, potrebno je završiti proces dijete. To možete napraviti s naredbom *exit(EXIT_SUCCESS*). Usporedite ispis u zadatku 4 s ispisom programa u zadatku 3 te komentirajte razliku.
2. Koliko puta će se ispisati riječ „OS“ izvršavanjem sljedećeg programskog koda?

```
#include <stdio.h>
#include <unistd.h>

int main() {
        for (int i=0; i < 3; i++){
                fork();
                printf(“OS“);
        }
        return 0;
}
```

3. Nacrtajte stablo procesa za sljedeći programski kod i za svaki proces u stablu naznačite kojim je po redu sustavskim pozivom *fork()* stvoren. Funkciju *main()* smatrajte glavnim

roditeljskim procesom, tj. korijenskim čvorom stabla. Napomena: roditeljski proces se prenosi kao čvor u stablu na iduću razinu nakon izvođenja *fork()* poziva.

```
#include <stdio.h>
#include <unistd.h>

int main() {
        if (fork() || fork())
                fork();
        return 0;
}
```

4. Nacrtajte stablo procesa za sljedeći programski kod i za svaki proces u stablu naznačite kojim je po redu sustavskim pozivom *fork()* stvoren. Navedite izlaz programa. Funkciju *main()* smatrajte glavnim roditeljskim procesom, tj. korijenskim čvorom stabla. Napomena: roditeljski proces se prenosi kao čvor u stablu na iduću razinu nakon izvođenja *fork()* poziva.

```
#include <stdio.h>
#include <unistd.h>

int main() {
        for (int i=0; i < 3; i++){
                if (i%2 == fork())
                        fork();
                printf("%d ", i);
                fflush(stdout);
        }
        return 0;
}
```

5. Koji je izlaz sljedećeg programskog koda? Komentirajte zašto.

```
#include <stdio.h>
#include <unistd.h>

int counter = 1;

int main() {
        if (fork () == 0) {
                counter ++;
                exit(0);
        }
        else {
                wait (NULL);
                counter ++;
                printf("counter = %d\n", counter);
        }
        return 0;
}
```

6. Koje su sve moguće kombinacije izlaza sljedećeg programa? Komentirajte zašto.

```
#include <stdio.h>
#include <unistd.h>

int counter = 1;

int main() {
        printf("p");
        fflush(stdout);
        if (fork() != 0) {
                printf("q");
                fflush(stdout);
                return 0;
        } else {
                printf("r");
                fflush(stdout);
                waitpid(-1, NULL, 0);
        }
        return 0;
}
```

7. Koristeći sustavske pozive opisane u vježbi (*fork*, *exit*) te naredbe *umask*, *setsid* napišite programski kod koji izvršava proces *daemon*. Proces *daemon* se mora pokretati u pozadini, vrjeti u beskonačnoj petlji i izvoditi proizvoljan zadatak (primjerice: zapisivati nekakve informacije o sustavu u proizvoljnu *log* datoteku). Pomoć: http://www.netzmafia.de/skripten/unix/linux-daemon-howto.html.
8. Koristeći sustavske pozive opisane u vježbi napišite program koji pokreće naredbu Linux ljuske *ls* (putanja do izvršne datoteke je "*/bin/ls*"), pri čemu se kao argument u komandnom sučelju predaje naziv direktorija čiji sadržaj treba ispisati te potencijalne opcije naredbe (npr. *-l*, *-a*,…).

5. LITERATURA

[1] Stallings, W., 2012. Operating systems: internals and design principles. Boston: Prentice Hall,.

[2] Tanenbaum, A.S. and Bos, H., 2015. Modern operating systems. Pearson.

[3] Bovet, D.P. and Cesati, M., 2005. Understanding the Linux Kernel: from I/O ports to process management. " O'Reilly Media, Inc.".

[4] Love, R., 2010. Linux kernel development. Pearson Education.

[5] Love, R., 2013. Linux system programming: talking directly to the kernel and C library. " O'Reilly Media, Inc.".

[6] https://linuxjourney.com/

[7] https://www.geeksforgeeks.org/operating-systems/

[8] https://www.sourcecodesolutions.in/2010/09/cs1254-operating-system-lab.html

[9] http://linasm.sourceforge.net/docs/syscalls/process.php

[10] Bryant, R. and O'Hallaron, D., 2016. Computer Systems: A Programmer's Perspective. Carnegie Mellon University

[11] Jelenković L., 2019. Operacijski sustavi: Interni material za predavanja iz predmeta. Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva (http://www.zemris.fer.hr/~leonardo/os/fer/_OS-skripta.pdf)