Upravljanje procesima u operacijskom sustavu Linux

Debelec, Emma

Sveučilište u Rijeci – Odjel za informatiku

Preddiplomski jednopredmetni studij informatike

Emma Debelec

Upravljanje procesima u operacijskom sustavu Linux

Završni rad

Mentor: Doc. dr. sc. Vanja Slavuj

Rijeka, rujan 2020.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 17. siječnja 2019. godine

Zadatak za završni rad

Pristupnik: Emma Debelec

Naziv završnog rada:

Upravljanje procesima u operacijskom sustavu Linux

Naziv završnog rada na eng. jeziku:

Process management in Linux operating system

Sadržaj zadatka:

Opisati pojam procesa, vrste procesa te načine upravljanja procesima u operacijskom sustavu. Na primjeru operacijskog sustava Linux prikazati načine korisničkog upravljanja procesima u sustavu primjenom programa sučelja naredbenog retka (CLI) ili grafičkog sučelja (GUI).

Mentor

Doc. dr. sc. Vanja Slavuj

Voditelj za završne radove

Dr. sc. Miran Pobar

Zadatak preuzet: 15. rujna 2020.

(potpis pristupnika)

Sažetak

U ovome radu opisan je osnovni element operacijskog sustava – proces. Proces je aktivnost koja nastaje pri izvođenju nekog programa i za vrijeme svoga života može poprimiti tri stanja (stanje spremnosti, stanje izvođenja procesa i stanje blokiranosti). Sastoji se od jedne ili više procesnih dretvi koje međusobno dijele adresni prostor, datoteke i dodijeljene resurse. Sve svoje podatke zapisuju u tablicu procesa, koja se u Linux operacijskom sustavu implementira dvostruko povezanom listom. Procesi mogu međusobno komunicirati koristeći cijevi ili slanjem signala. Također, opisan je upravljač procesorom (engl. scheduler) – dio jezgre koji određuje koji proces je sljedeći na redu za obradu primjenom algoritma za dodjelu prioriteta birajući procese iz redova čekanja. Korisnik može upravljati procesima upisivanjem odgovarajućih naredbi u terminal, primjerice naredbom `ps` može dobiti informacije o trenutno aktivnim procesima, naredbom `kill` može slati signale procesima, naredbom `nice` pokreće proces s odgovarajućim prioritetom i naredbom `renice` mijenja prioritet izvođenja aktivnog procesa.

Ključne riječi: Linux, operacijski sustav, procesi, upravljanje procesima

Sadržaj

Uvod

Programi u operacijskom sustavu imaju zajedničke karakteristike. Sadrže programski kod, datoteke i podatke smještene na nekim memorijskim lokacijama. Ove, ali i druge karakteristike objedinjuje koncept procesa. Proces je aktivnost koja nastaje pri izvođenju nekog programa, što znači da operacijski sustav mora znati kada i koliko dugo će izvoditi određeni proces. To utječe na stabilnost i performanse sustava budući da operacijski sustav istovremeno obrađuje više stotina procesa.

Neizbježno je da će doći do greške u radu sustava uzrokovane izvršavanjem procesa, npr. preveliko opterećenje procesora, potrošnja dostupne memorije i slično. Za ispravan oporavak od ovakvih grešaka nužno je znati kako efikasno upravljati procesima, što uključuje kontrolu izvođenja procesa, sigurno završavanje procesa, kontrolu njihovih okruženja i slično.

Duboko razumijevanje koncepta procesa omogućuje nam da na vrijeme reagiramo prije nego se se sustav otme kontroli, što je izuzetno važno za administratore koji održavaju poslužitelje (npr. web stranice). Osim što je znanje korisno za sistemske administratore i napredne korisnike, korisno je i za programere koji prilagođavaju aplikaciju Linux okruženju. Programeri mogu koristiti usluge operacijskog sustava kako bi poboljšali rad aplikacije, primjerice praćenje signala radi pružanja otpornosti na veće greške, ali i izbjegavanje usluga koje su dostupne isključivo Linuxu.

Ovaj rad objašnjava implementaciju procesa u Linux sustavu i način na koji se njima upravlja. Završni rad podijeljen je u dva velika poglavlja. U prvom poglavlju opisani su osnovni elementi operacijskog sustava. Opisan je proces, njegov nastanak i opisana su stanja koja on može poprimiti za vrijeme svog životnog ciklusa. Opisan je upravljač procesorom, dio jezgre zadužen za upravljanje procesima i procesnim dretvama. Njegovo upravljanje određeno je algoritmom, pa su opisana dva algoritma, jedan koji se koristio u prošlosti i drugi koji ga je zamijenio. Drugo poglavlje opisuje kako upravljati elementima opisanim u prvom poglavlju rada. Opisano je na koji način korisnik upravlja procesima i poslovima upotrebom naredbi i korištenjem odgovarajućih parametra. Osim upravljanja, pokriveno je i nadgledanje procesa i poslova.

1. Elementi operacijskog sustava

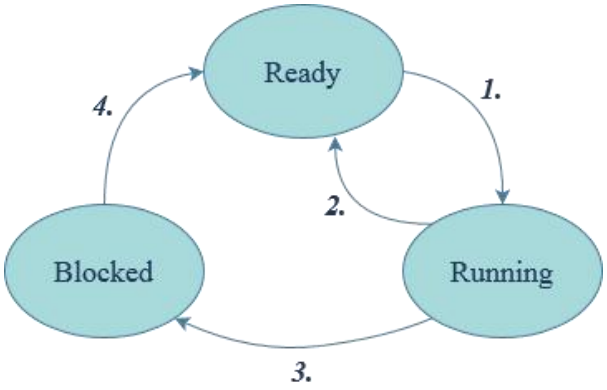
1.1. Proces

Proces je osnovni element operacijskog sustava, tj. aktivnost koja nastaje pri izvođenju nekog programa. Svaki proces je jedinstven, zauzima memorijski prostor u kojem sadrži svoj programski kod, podatke za obradu, otvorene datoteke i ostale potrebne resurse. Operacijski sustav u danom trenutku obrađuje više procesa, međutim računalo nije u mogućnosti izvršavati više procesa istovremeno (Tanenbaum & Bos, 2015).

Paralelno izvođenje više računalnih programa zahtjeva i paralelno izvođenje njihovih procesa. U slučaju jednojezgrenog procesora postupak multiprogramiranja izvodi se cikličkim ponavljanjem procesa u kratkotrajnim vremenskim intervalima sve dok program ne završi svoje izvođenje. Iako se prividno čini da su svi procesi radili istovremeno, zapravo se u jednom trenutku izvršavao jedan proces. Cijeli postupak nazivamo pseudoparalelizam. Stvarno paralelno izvođenje moguće je kod računala sa više procesorskih jedinica (engl. central processing unit, CPU), gdje se također procesi ponavljaju ciklički, ali na više jezgri istovremeno (Tanenbaum & Bos, 2015).

Kada neki proces izgubi pravo na procesor, pohranjuje svoje stanje u trenutku kada je bio prekinut i ponovno nastavlja svoje izvođenje pri novoj dodjeli procesora. Moguća stanja procesa su: ready (stanje spremnosti), running (stanje izvođenja procesa) i blocked (stanje blokiranosti procesa i čekanje ispunjenja uvjeta za daljnji rad) (Tanenbaum & Bos, 2015).

Slika 1 prikazuje spomenuta stanja procesa. Kada se novi proces stvori, nalazi se u ready stanju. U ready stanju proces čeka dodjelu procesora kako bi započeo svoje izvođenje ili nastavio ako je već bio pokrenut. Čim dobije procesor, proces prelazi iz ready u running stanje (1. prijelaz) te započinje svoju obradu. Proces je u running stanju sve do gubitka procesora kada prelazi natrag u ready stanje (2. prijelaz). Ako za vrijeme izvođenja procesa nastanu uvjeti koji onemogućuju daljnje izvođenje, proces prelazi u blocked stanje (3. prijelaz). U blocked stanju čeka ispunjenje uvjeta (npr. čitanje podataka s diska) kako bi nastavio svoj rad. Po ispunjenju uvjeta proces ponovno poprima stanje ready (4. prijelaz).



Slika 1 – *Stanja procesa.*

Više procesa mogu međusobno komunicirati i na taj način sinkronizirati svoje aktivnosti. Sinkronizacija se odvija zaustavljanjem tijeka rada jednog procesa i nastavljanjem izvođenja istog nakon nekog vremenskog intervala. Kako bi se proces nastavio od trenutka gdje je bio zaustavljen, svoje prethodno stanje pohranjuje u tablicu procesa (Tanenbaum & Bos, 2015).

1.1.1. Stvaranje procesa

Operacijski sustavi koji se baziraju na UNIX platformama koriste sistemski poziv `fork` za stvaranje novog procesa. Proces koji poziva `fork` (proces „roditelj“) stvara kopiju originalnog procesa koji se zove proces „dijete”. (Kerrisk, 2017a)

1.1.1.1. Sistemski pozivi

Kao što je već spomenuto, sistemski poziv `fork` stvara kopiju procesa koji ima odvojenu memoriju od procesa koji je pozvao `fork`. Ako proces roditelj naknadno mijenja neke varijable, te promjene neće biti vidljive procesu djetetu i obrnuto. Roditelj i dijete međusobno dijele datoteke, što znači da će promjene unutar datoteke biti vidljive jednom i drugom procesu, neovisno koji je proces napravio promjenu (Tanenbaum & Bos, 2015).

Budući da su to kopije istog procesa, kako procesi znaju koji je dijete, a koji roditelj. Sistemski poziv `fork` vraća vrijednost 0 procesu dijete, dok proces roditelj dobiva vrijednost različitu od nule. Prema vrijednosti koju su dobili raspoređuju svoje zadatke. Svaki proces ima identifikator procesa (PID, Process Identifier). PID je broj koji jednoznačno određuje proces u sustavu (Tanenbaum & Bos, 2015). Tablica 1 opisuje shemu izvođenja nakon sistemskog poziva `fork`.

Kada je novi proces stvoren, procesu roditelju je dodijeljen PID djeteta, dok proces dijete sistemskim pozivom *getpid* dobiva novi identifikacijski broj.

Tablica 1 – *Shema izvođenja nakon sistemskog poziva fork.*

`pid = fork();`
 	
`if(pid < 0){`
`  handle_error();`
`}else if(pid > 0){`
`  /*parent code*/`
`}else{`
`  /*child code*/`
`}`
	Poziva se sistemski poziv fork i izlazna vrijednost zapisuje u varijablu pid.
	Ako je pid manji od nule dogodila se neka greška.
	Ako je pid veći od nule znači da je to proces roditelj i izvršava svoj kod.
	Ako pid nije ni veći niti manji od nule, znači da je pid jednak nuli i to odgovara procesu djetetu koji ima svoj kod.

Nakon sistemskog poziva `fork`, ako se ne izvršava isti program, slijedi sistemski poziv `exec`[1]. Ovim pozivom zamijenjuje se slika sadašnjeg procesa s novom slikom. Slika je novi program koji će se izvršavati. U slučaju da proces roditelj mora čekati izvršenje procesa dijete, roditelj poziva `waitpid` kojim čeka terminaciju djeteta (čeka bilo koje dijete ako postoji više od jednog) (Tanenbaum & Bos, 2015).

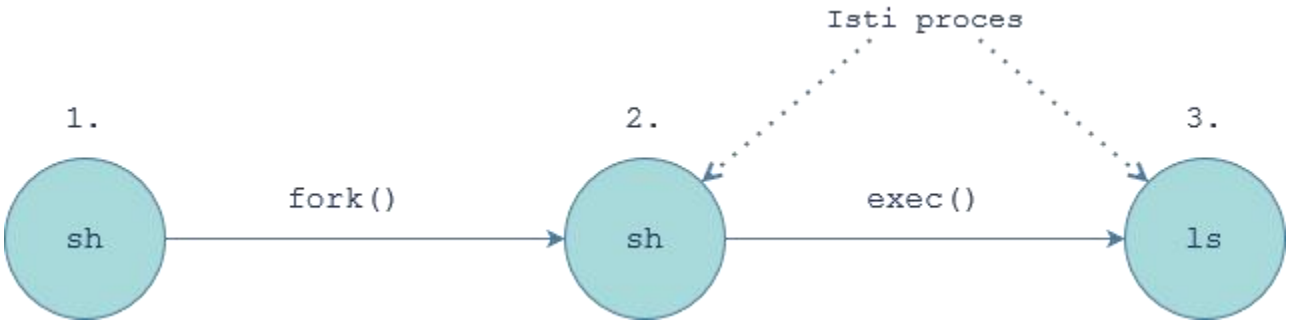
`Waitpid` može zaprimiti tri parametra. `Waitpid` može čekati specifično dijete postavljanjem prvog parametra na -1. Drugi parametar `waitpid` poziva pohranjuje izlazno stanje procesa dijete (je li dijete normalno ili abnormalno terminiralo). Trećim parametrom moguće je postaviti razne opcije, npr. da funkcija završi odmah ako niti jedno dijete nije postojalo (Kerrisk, 2020e).

Pogledajmo kako ljuska koristi `fork`. Kada se pokrene neka naredba, ljuska pokreće sistemski poziv `fork` kako bi stvorila novi proces. Novonastali proces dijete izvršava upisanu naredbu. Prikaz izvođenja sistemskog poziva od strane ljuske, dan je u Tablici 2.

[1] Zapravo se poziva jedna od sljedećih funkcija: `execl, execlp, execle, execv, execvp, execvpe.`

Tablica 2 – *Pojednostavljeni rad ljuske.*

`while(TRUE){`
`  type_prompt();`
`  read_command(command, parameters);`
`if(fork()!=0){`
`  /* parent code */`
`  waitpid(-1, &status, 0);`
`}else{`
`  /* child code*/`
`  execve(command, parameters, 0);`
`  }`
`}`
	Unos se stalno ponavlja.
	Prikazuje se poruka za unos naredbe.
	Ljuska čita naredbu s terminala.
	Ako proces nije dijete, izvršava se kod roditelja.
	Roditelj pokreće `waitpid` i čeka terminaciju djeteta.
	Inače, proces je dijete izvršava `exceve`.

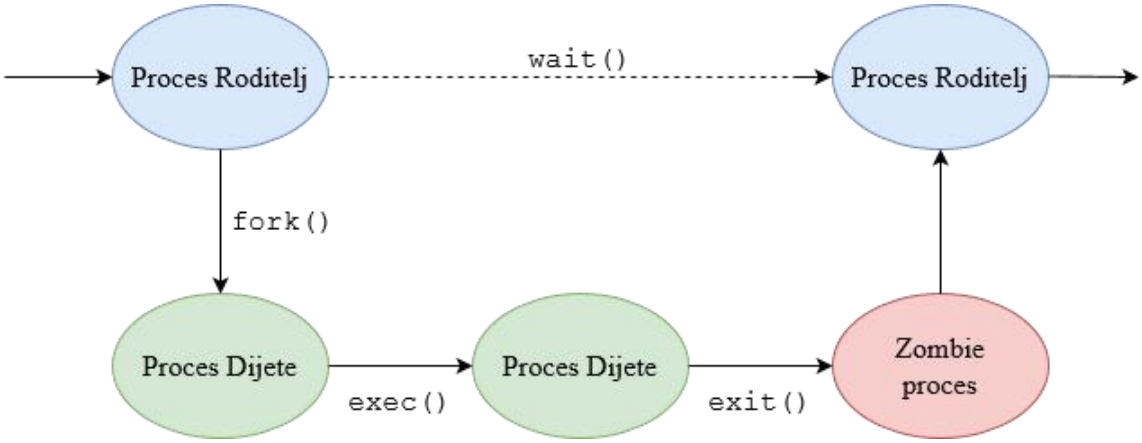


Slika 2 – *Koraci pri pokretanju naredbe ls u ljusci.*

Slika 2 Prikazuje korake koji se izvode kada se pokrene naredba `ls`. Ljuska (1. sh) prvo pokreće sistemski poziv `fork()` i stvara kopiju sebe (2. sh). Novonastala ljuska poziva `exec()` funkciju i zamjenjuje se sa slikom programa `ls` (3. ls). U memoriji se sadržaj zamijeni sa sadržajem datoteke naredbe `ls`. Nakon toga pokreće se i izvodi naredba `ls`.

1.1.1.2. Zombie stanje

Slika 3 prikazuje nastanak tzv. zombie procesa. Proces dijete ulazi u zombie stanje ako završi svoju obradu, a roditelj ga još nije dočekao tj. još ne pročitao njegovo izlazno stanje. Zombie stanje je stanje procesa kada je proces aktivan, izvršio je svoj zadatak, no ne može prekinuti svoje postojanje bez roditelja. Tek kada ga roditelj dočeka, proces prekida svoje postojanje i prestaje biti na listi procesa (Tanenbaum & Bos, 2015).



Slika 3 - *Prikaz zombie stanja procesa.*

Slika 4 prikazuje kod koji rezultira zombie procesom. Slika 5 Prikazuje da je proces sa PID vrijednosti 6033 u zombie stanju.

```
1  import os
2  import sys
3  import time
4
5  pid = os.fork()
6
7  if pid > 0:
8          print("Proces roditelj, PID:", pid)
9          time.sleep(5)
10 elif pid == 0:
11         print("Proces dijete, PID:", os.getpid(), ". Izlazim...")
12         sys.exit(0)
```

Slika 4 – *Python kod za dobivanje zombie procesa.*

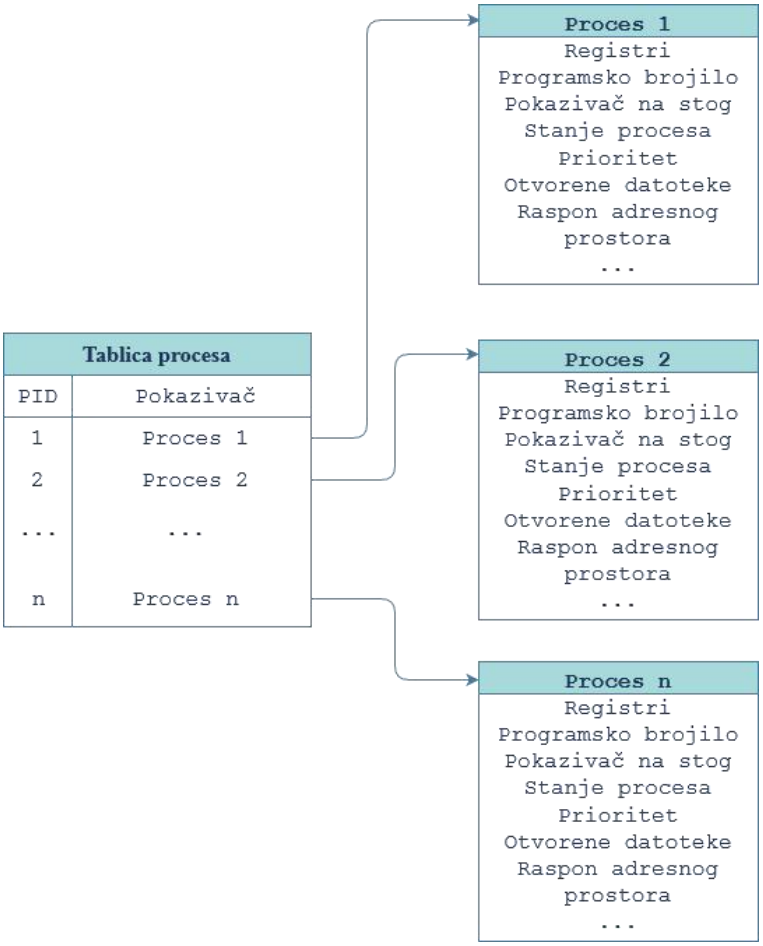
```
bolfijaner@bolfijaner: ~
bolfijaner@bolfijaner:~$ python3 zombie.py &
[3] 6032
[2]   Done                    python3 zombie.py
bolfijaner@bolfijaner:~$ Proces roditelj, PID: 6033
Proces dijete, PID: 6033 . Izlazim...

bolfijaner@bolfijaner:~$ ps aug | grep python
bolfija+  6032  0.7  0.4  22984  8704 pts/0    S    18:00   0:00 python3 zombie.py
bolfija+  6033  0.0  0.0      0     0 pts/0    Z    18:00   0:00 [python3] <defunct>
bolfija+  6035  0.0  0.0  14636   920 pts/0    S+   18:00   0:00 grep --color=auto python
bolfijaner@bolfijaner:~$
```

Slika 5 – *Popis Python procesa među kojima je induciran i zombie proces.*

1.1.2. Tablica procesa

Operacijski sustav koristi tablicu procesa za zapisivanje bitnih informacija o svakom procesu. Slika 6 shematski prikazuje koncept tablice procesa. Svaki proces upisan je u tablicu samo jednom. Svaki zapis sadrži informacije o stanju procesa, programsko brojilo, pokazivač na stog, raspon alocirane memorije, podatke o otvorenim datotekama i podatke bitne za upravljač procesima (engl. scheduler). Svi ovi podaci bitni su za proces kako bi se mogao probacivati iz stanja u stanje (spomenuta stanja ready, running i blocked). Svaki nastali proces dobije svoj zapis u tablici procesa i njegovi podaci se često ažuriraju dok god je proces aktivan. Nakon terminacije procesa, njegov zapis se briše iz tablice (Tanenbaum & Bos, 2015).

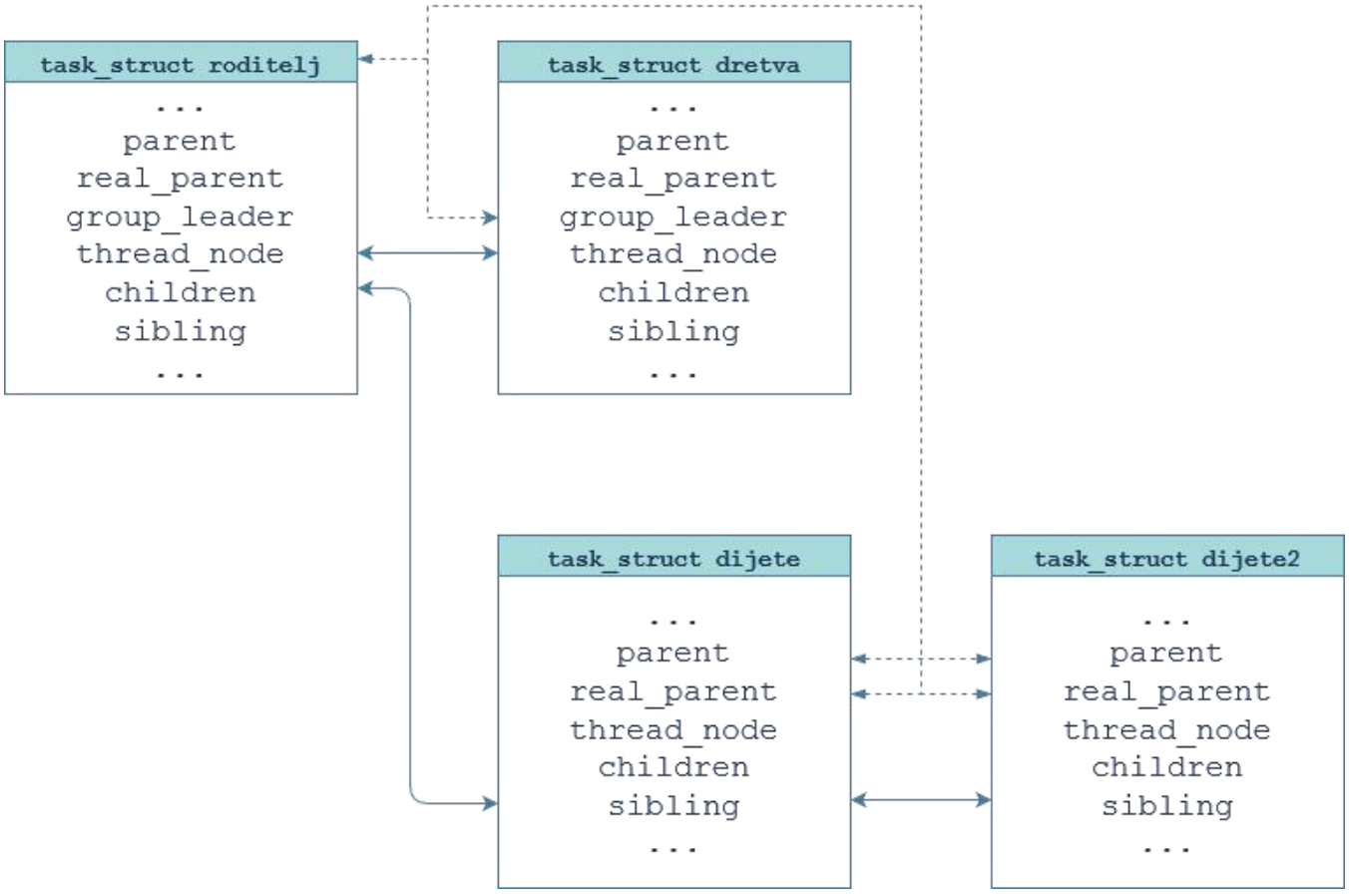


Slika 6 – *Koncept tablice procesa.*

U operacijskom sustavu Linux, tablica procesa zapravo je implementirana dvostruko povezanom listom u kojoj je svaki čvor `task_struct` struktura definirana u `include/linux/sched.h`. Glava tj. početak liste je *init* proces: to je prvi proces koji se pokreće u Linux operacijskom sustavu. Lista procesa djece pohranjena je u *children* članu te strukture. Na isti način pohranjena je i lista rođaka i roditelja procesa (Torvalds, 2020).

Linux ne razlikuje procese i procesne dretve kao dvije različite strukture te ih prikazuje na isti način. Razlika je u njihovim međusobnim odnosima i podacima zapisanim unutar tih struktura. Na primjer, `task_struct` procesa sadrži pokazivač `group_leader` na glavnu dretvu procesa, odnosno na drugu `task_struct` strukturu. Prvoj i svim ostalim `task_struct` dretvama moguće je pristupiti preko `thread_node` člana koji je lista koja sadrži sve dretve procesa (Torvalds, 2020). Slika 7 prikazuje odnose *task* struktura između procesa i procesne dretve te procesa i njegove djece.

Čitanje podataka ove liste moguće je koristeći `procfs`, o čemu će više biti rečeno kasnije.



Slika 7 – *Odnosi task struktura između procesa i dretve.*

1.1.3. Komunikacija među procesima

1.1.3.1. Cijevi

Procesi u Linuxu mogu međusobno komunicirati. Moguće je stvoriti kanal između dva procesa unutar kojeg će jedan proces zapisivati podatke drugomu procesu za čitanje. Takvi kanali se zovu cijevi (engl. pipes). Sinkronizacija čitanja i pisanja poruka odvija se na način da ako neki proces pokuša pročitati iz prazne cijevi, bit će blokiran dok podaci za čitanje ne postanu dostupni. Kada ljuska (engl. shell) vidi unos poput: (Tanenbaum & Bos, 2015)

```
sort <f | head
```

kreiraju se dva procesa, *sort* i *head*. Procesi komuniciraju tako da je izlazna vrijednost prvog procesa ulazna vrijednost drugom procesu. Na taj se način koriste podaci bez zapisivanja u datoteku.

1.1.3.2. Signali

Osim putem cijevi, procesi mogu međusobno komunicirati koristeći signale. Po primitku signala proces može reagirati na jedan od sljedećih zadanih načina koji zovemo dispozicije (engl. disposition): (Kerrisk, 2020d)

- proces se terminira,
- proces ignorira signal,
- proces terminira i zapisuje svoje sadržaje u memoriju na disk,
- proces se zaustavlja i
- proces nastavlja s radom ako je ranije bio zaustavljen.

Također je moguće da aplikacija sama određuje način na koji će obraditi signal. Aplikacija može koristiti sistemske pozive `signal()` ili `sigaction()` kako bi promijenio dispoziciju to jest može definirati vlastitu funkciju koja prima identifikator signala kao jedini parametar i postupati na drugačiji način od definiranog. Preporučeni poziv za promjenu dispozicija je `sigaction()` budući da ponašanje poziva `signal()` varira kroz razne inačice Linux jezgre i ima problem s prenosivosti (Kerrisk, 2017b, 2020c). Slika 8 prikazuje deklaraciju `sigaction` funkcije.

Poziv `sigaction()` prima tri parametra. Prvi je identifikator signala, drugi parametar je pokazivač na novu `sigaction` strukturu, a treći parametar je pokazivač na staru `sigaction`

strukturu. Prvi član `sigaction` stukture je pokazivač na funkciju aplikacija koja obrađuje primljeni signal. `Sigaction` vraća vrijednost 0 ako je uspješno završio, a inače vrijednost -1.

```
#include <signal.h>

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

Slika 8 – *Deklaracija `sigaction` funkcije.*

Mijenjanje dispozicije moguće je za sve signale izuzev signala SIGKILLL i SIGSTOP. Dakle, te signale nije moguće uhvatiti u aplikaciji (Kerrisk, 2020c).

Signali mogu biti sinkroni i asinkroni. Sinkroni signali pojavljuju se kao rezultat normalnog tijeka izvođenja programa, dok se asinkroni signali eksplicitno šalju procesu (eksterni) (Bar, 2000). Primjeri asinkronih signala su SIGKILL i SIGTERM, a sinkronih SIGSEGV.

Proces može slati signale samo članovima svoje grupe (engl. process group). Procesna grupa se sastoji od roditelja, rođaka i djece. Proces također može poslati signal svim članovima procesne grupe putem jednog sistemskog poziva (Kerrisk, 2014a).

Osim za komunikaciju između procesa, signali se koriste i u druge svrhe. Ako proces izvršava neku nedozvoljenu ili nedefiniranu matematičku operaciju (npr. dijeljenje s nulom) dobit će signal SIGFPE (engl. floating-point exception) (Tanenbaum & Bos, 2015). Tablica 3 prikazuje listu signala u Linux sustavu. Prvi stupac je identifikator signala, nakon njega slijedi naziv signala, njegov opis te naposlijetku rezultat izvođenja to jest zadana dispozicija (Kerrisk, 2020d).

Tablica 3 - *Prikaz signala Linux sustava.*

Identifikator	Naziv	Opis	Rezultat izvođenja
1	SIGHUP	Signal koji se šalje procesu kad je njegov upravljački terminal zatvoren.	Terminacija
2	SIGINT	Signal koji korisnik šalje procesu kada želi prekinuti proces. Npr. CTRL + C u Linux terminalu.	Terminacija
3	SIGQUIT	Korisnik je zatražio zapis rada procesa (core dump).	Terminacija i zapis memorije
4	SIGILL	Korisnik je pritisnuo DEL tipku i prekinuo proces.	Terminacija i zapis memorije
5	SIGABRT	Poslano za prekid procesa i prisilno izbacivanje sadržaja memorije na disk (core dump).	Terminacija i zapis memorije
5	SIGTRAP	Signal koji se šalje procesu kada dođe do iznimke ili dođe do ispunjenja uvjeta koji je zatražio debugger. Na primjer kada program dođe do određene linije koda.	Terminacija i zapis memorije
6	SIGIOT	Sinonim za SIGABRT.	Terminacija i zapis memorije
7	SIGBUS	Signal koji se šalje procesu prilikom pristupa fizičkoj memorijskoj lokaciji koja ne postoji.	Terminacija i zapis memorije
8	SIGFPE	Dogodila se greška prilikom rada s float varijablom. Na primjer dijeljenje s nulom.	Terminacija i zapis memorije
9	SIGKILL	Poslan da ubije proces. (Nije ga moguće uhvatiti niti ignorirati).	Terminacija
10	SIGUSR1	Dostupno u svrhe definirane aplikacijom.	Terminacija
11	SIGSEGV	Proces je upućen na neispravnu memorijsku adresu.	Terminacija i zapis memorije
12	SIGUSR2	Dostupno u svrhe definirane aplikacijom.	Terminacija i zapis memorije
13	SIGPIPE	Proces je napisao poruku cijevi bez procesa čitatelja.	Terminacija i zapis memorije
14	SIGALRM	Signal koji se pojavljuje kada odbrojavanje alarma završi.	Terminacija i zapis memorije
15	SIGTERM	Koristi se za zahtjevanje terminacije procesa.	Terminacija i zapis memorije
16	SIGSTKFLT	Šalje se prilikom pojave greške na stogu, na primjer čitanje s praznog stoga. Linux ne koristi ovaj signal.	Terminacija i zapis memorije

17	SIGCHLD	Signal koji se šalje kada se terminira proces dijete.	Ignorira se
18	SIGCONT	Koristi se za nastavak zaustavljenog procesa.	Nastavak zaustavljenog procesa
19	SIGSTOP	Koristi se za zaustavljanje procesa.	Stopiranje
20	SIGTSTP	Signal koji terminal šalje za stopiranje procesa, na primjer pritiskom tipki CTRL + Z.	Stopiranje
21	SIGTTIN	Signal koji se šalje kada pozadinski proces ne može čitati ulaz s terminala.	Stopiranje
22	SIGTTOU	Slično kao i SIGTTIN, ali proces ne može zapisivati na terminal.	Stopiranje
23	SIGURG	Signal koji se šalje prilikom pojave hitnog stanja na utičnici.	Ignorira se
24	SIGXCPU	Signal koji se šalje kada je proces prekoračio dozvoljeno vrijeme korištenja procesora koje je korisnik definirao.	Terminacija i zapis memorije
25	SIGXFSZ	Signal se šalje kada proces zapiše više od dozvoljene količine podataka.	Terminacija i zapis memorije
26	SIGVTALRM	Signal koji se šalje kada je proces iskoristio vrijeme korištenja procesora u korisničkom dijelu. Vrijeme definirano `setitimer()` pozivom.	Terminacija
27	SIGPROF	Signal koji se šalje kada je proces iskoristio vrijeme korištenja procesora uključujući i ono vrijeme provedeno u kernelu. Vrijeme je definirano `setitimer()` pozivom.	Terminacija
29	SIGIO	Signal koji obavještava da je operacija čitanja ili pisanja moguća na nekom objektu poput utičnice ili datoteke.	Terminacija
29	SIGPOLL	Sinonim za SIGIO.	Terminacija
30	SIGPWR	Signal koji se šalje ako je nestalo struje.	Terminacija
31	SIGSYS	Signal koji se šalje procesu kada je proslijedio krivi argument nekom sistemskom pozivu.	Terminacija i zapis memorije
31	SIGUNUSED	Sinonim za SIGSYS.	Terminacija i zapis memorije

1.2. Procesne dretve

Unutar svakog procesa izvršava se barem jedna procesna dretva, ali proces može imati i više procesnih dretvi koje međusobno dijele adresni prostor, datoteke i dodijeljene resurse. Konceptualno, to je dio operacijskog sustava koji izvršava programski kod. Procesne dretve lakše je stvoriti i uništiti za razliku od procesa, a u mnogim sustavima kreiranje procesne dretve može biti 10 do 100 puta brže nego kreiranje procesa (Tanenbaum & Bos, 2015).

Već je spomenuto da Linux krenel u unutrašnjosti prezentira procese kao zadatke (engl. tasks) tj. jednako kao procese pomoću strukture `task_struct`. Ostali operacijski sustavi razlikuju procese i procesne dretve, dok Linux koristi *task* strukturu za bilo koji kontekst izvršenja zadataka. Jednodretven proces bit će prezentiran kao jedna *task* struktura, a višedretveni procesi će imati jednu *task* strukturu za svaku od korisnikovih procesnih dretvi. Kernel je višedretven i ima procesne dretve koje nisu povezane korisnikovim procesnim dretvama. U literaturi bolje performanse uglavnom su usko vezane uz manjak potrebe za upisom podataka u tablicu procesa, što u Linuxu nije slučaj budući da se za procese i dretve jednako mora stvoriti `task_struct` struktura, odnosno lista (Tanenbaum & Bos, 2015; Torvalds, 2020).

Prilikom stvaranja novog procesa potrebno je pozvati sistemske pozive `fork()` i `exec()` dok to nije slučaj kod stvaranja nove procesne dretve, što rezultira boljim performansama. Budući da dretve dijele adresni prostor, njihova međusobna komunikacija je puno lakše izvediva u odnosu na komunikaciju dva procesa te su one bolji način za postizanje paralelizma.

Evo i primjer kako je paralelizmom poboljšan rad. Uzmimo primjer poslužitelja koji prima zahtjeve od klijenata i šalje im neke podatke. Ako imamo jednu dretvu, ona nije u mogućnosti istovremeno slati podatke spojenim klijentima i spajati nove klijente. Procesna dretva prvo primi klijentov zahtjev za spajanje, a nakon toga spoji tog klijenta i primi novi zahtjev za spajanje. Prije nego što obradi taj novi zahtjev za spajanje, ona mora prvom klijentu poslati podatke. Tek nakon što je poslala podatke može obrađivati nove zahtjeve za spajanjem. Jedno od rješenja tog problema je da glavna procesna dretva bude zadužena samo za obrađivanje zahtjeva za spajanje novih klijenata, dok za potrebe slanja podataka klijentima stvori novu procesnu dretvu. Glavna procesna dretva stvara jednu procesnu dretvu za svakog klijenta kojemu je potrebno slanje podataka.

S obzirom da dretve dijele adresni prostor, a s time i memoriju, nije potrebno uvoditi mehanizme za međuprocesnu komunikaciju. Ako dretva mora pristupiti nekom objektu unutar procesa, dovoljno je samo da dobije referencu na njega.

1.3. Scheduler - upravljač procesima u Linuxu

Sada ćemo pogledati u Linuxov scheduling algoritam. Scheduler ili upravljač procesorom je dio jezgre koji određuje koji proces ili procesna dretva je sljedeći na redu za obradu. Scheduler donosi odluku primjenom algoritma za dodjelu prioriteta birajući procese odnosno dretve iz redova čekanja (Tanenbaum & Bos, 2015).

Za početak, Linuxove dretve su kernel dretve, stoga je scheduling baziran na procesnim dretvama, a ne na procesima. Linux razlikuje tri razine dretvi za svrhe schedulinga: (Tanenbaum & Bos, 2015)

- Stvarno vrijeme FIFO (First in - first out)
- Stvarno vrijeme Real robin
- Timesharing

FIFO dretve u stvarnom vremenu su najvećeg prioriteta. Jedino novije dretve FIFO mogu imati veći prioritet. Dretve Round robin jednake su kao i FIFO dretve, osim što imaju određeno vrijeme - kvantum povezan s njima, te su zato vremenski određene. Ako je više Round Robin dretvi spremno, svaka je pokrenuta u svom vremenu, poslije čega ide na kraj liste na kojoj su Round robin dretve u stvarnom vremenu. Nijedan od ovih razreda procesnih dretvi nije zapravo određen u stvarnom vremenu ni u jednom smislu.

Ove klase procesnih dretvi zapravo su višeg prioriteta nego standardne dretve timesharing klase. Procesne dretve u stvarnom vremenu su u unutrašnjosti prezentirane po razinama prioriteta kojima je raspon od 0 do 99, gdje je 0 najviša razina prioriteta, a 99 najniža razina (Tanenbaum & Bos, 2015).

Procesne dretve koje nisu određene stvarnim vremenom formiraju svoju posebnu klasu i one se ne natječu s dretvama u svarnom vremenu. U unutrašnjosti takve dretve imaju razine prioriteta od 100 do 139. Ukupno u unutrašnjosti razlikujemo 140 razina prioriteta (Tanenbaum & Bos, 2015).

Osim razine prioriteta, svaka procesna dretva ima i svoju *nice* vrijednost. *Niceness* određuje koliko često neki proces ili procesna dretva dolazi na red za izvođenje. Zadana vrijednost je 0, no ta vrijednost

može se izmijeniti pozivanjem sistemskog poziva `nice()` drugom vrijednošću. Raspon `nice` vrijednosti je od -20 do +19. Vrijednost kreće od -20 što znači da taj proces češće dobiva procesor. Analogno, procesi s `nice` vrijednošću +19 rjeđe dolaze na red (Tanenbaum & Bos, 2015).

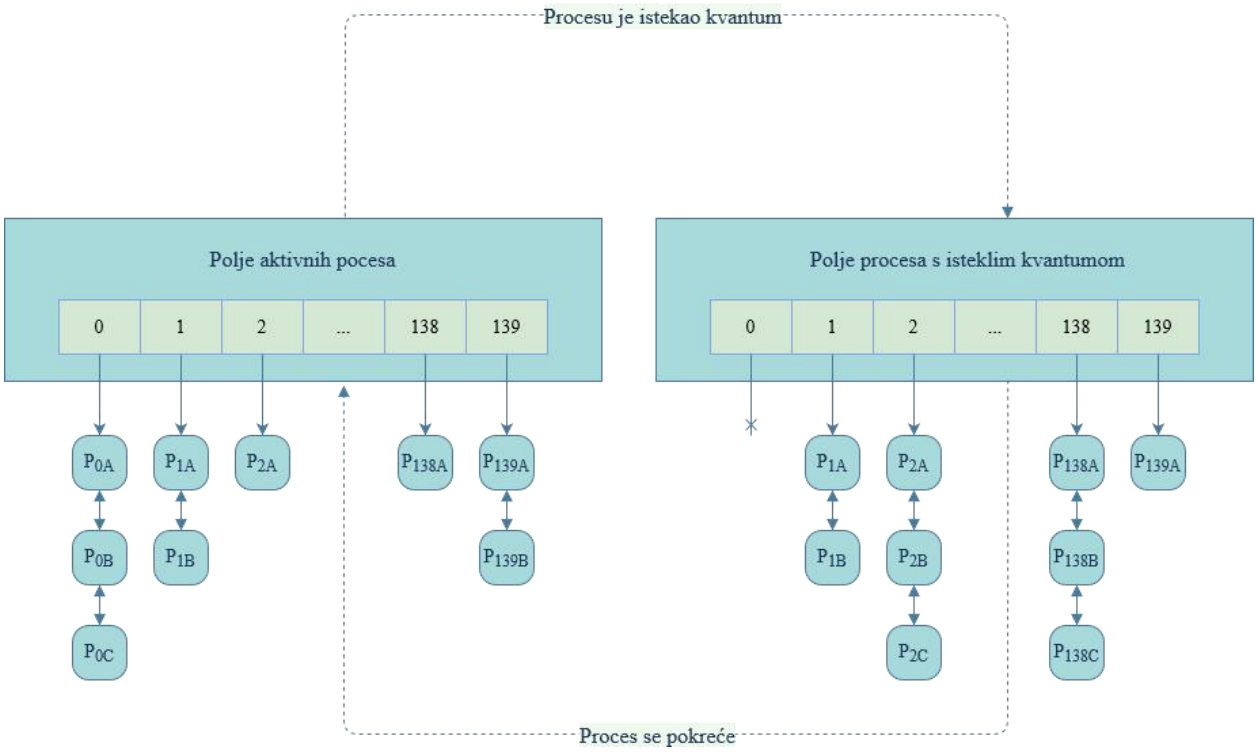
1.3.1. O(1) scheduler

O(1) *scheduler* implementira red čekanja kao dva polja, polje koje sadrži aktivne procese i polje u kojima su procesi koji su izgubili pravo na procesor. Svaki element polja je lista `task_struct` struktura koje imaju određeni prioritet. Polje sadrži ukupno 140 elemenata za prioritete od 0 do 139. Ova dva polja postoje za svaku jezgru procesora. *Scheduler* odabire zadatak s liste najvišeg prioriteta u polju aktivnih procesa (Tanenbaum & Bos, 2015).

Ako neka dretva s višim prioritetom dođe u stanje *ready*, trenutna dretva koja je bila u *running* stanju i ima niži prioritet, završava svoju obradu i ponovno se vraća na listu čekanja. Ako kvantum neke procesne dretve istekne, ona se premješta na popis dretvi s isteklim rokom te dobiva drugačiji prioritet. Ako procesna dretva završi u *blocked* stanju jer npr. čeka korisnikov unos, prije nego što će joj isteći kvantum ona se premješta natrag na listu aktivnih procensih dretvi, no smanjuje joj se vrijeme za korištenje procesora. Kada se njezin kvantum potpuno iscrpi, tj. kada potroši vrijeme koje joj je dodijeljeno, ide na listu procesa s isteklim rokom (Tanenbaum & Bos, 2015). Slika 9 shematski prikazuje polja aktivnih procesa i procesa kojima je istekao kvantum.

Kada više nema procesa sa zadacima u aktivnoj listi, slijedi zamjena lista sa aktivnim procesima i onima kojima je istekao rok. Ova metoda osigurava da zadaci niskog prioriteta također dobiju dio procesora. Na primjer, nekom procesu koji ima prioritet 100 dodijeljeno je 800 milisekundi kvantuma, dok će neki drugi proces s prioritetom 19 dobiti samo 5 milisekundi za korištenje procesora (Tanenbaum & Bos, 2015).

Ideja ove sheme je brzo obavljanje procesa. *Scheduler* O(1) je imao značajne nedostatke. Najveći nedostatak pokazivao je u radu s interaktivnim zadacima i zato je Ingo Molnar predložio novi *scheduler* pod nazivom potpuno pošten *scheduler* (engl. Completely Fair Scheduler / CFS) (Tanenbaum & Bos, 2015).



Slika 9 – *Polja aktivnih procesa i procesa kojima je istekao kvantum.*

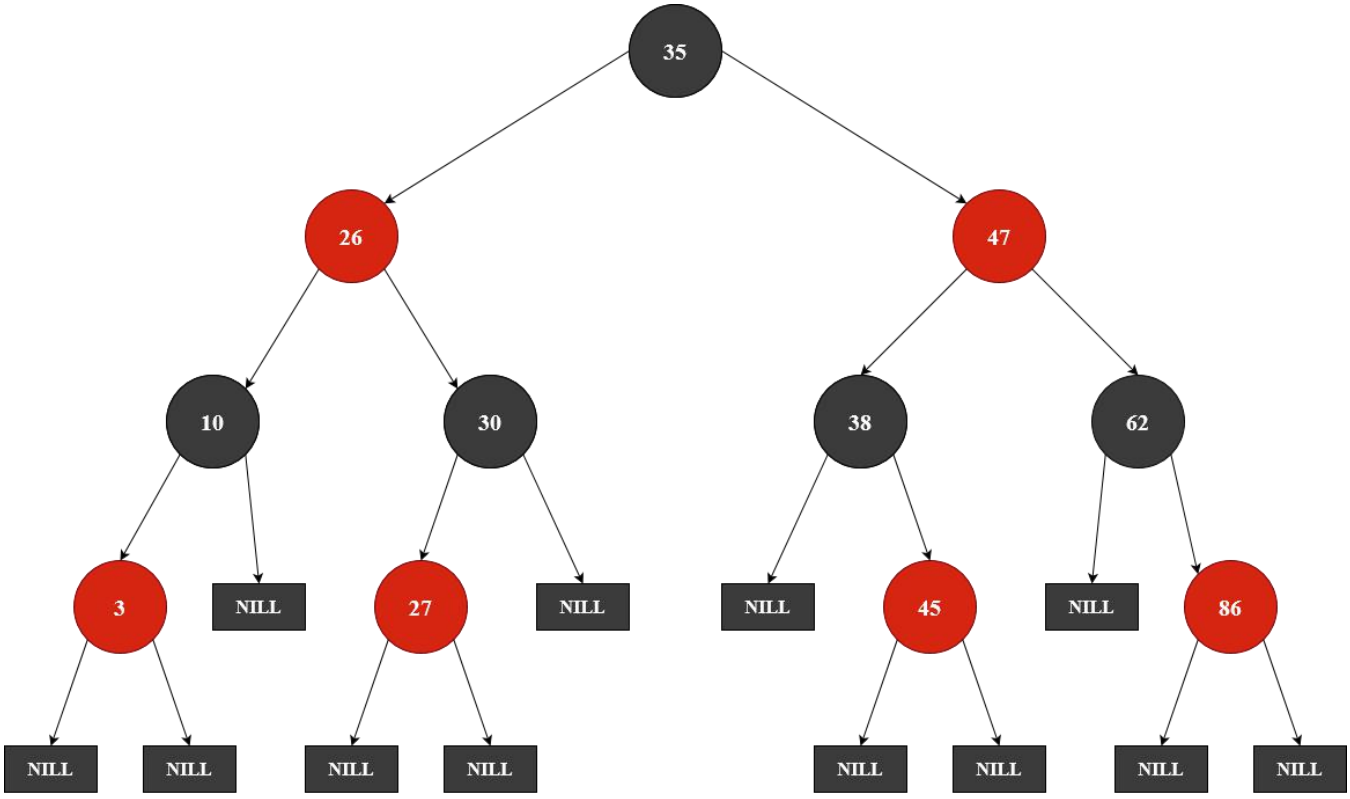
1.3.2. CFS - Potpuno pošten scheduler

Potpuno pošten *scheduler* rješava problem interaktivnih procesa koji je imao O(1) *scheduler*. Glavna ideja iza potpuno poštenog *schedulera* je korištenje crveno - crnog stabla kao strukture podataka u listi čekanja na obradu (Slika 10). Zadaci su na stablu poredani na temelju vremena koje provode koristeći CPU. To vrijeme nazivamo *vruntime.* Potpuno pošten *scheduler* obračunava vrijeme izvođenja zadataka u nanosekundama. Slika 10 prikazuje stablo i svaki čvor stabla odgovara jednom zadatku. Djeca s lijeve strane stabla su zadaci koji zahtijevaju manje vremena za obradu i ti zadaci će biti prije pozvani za obradu. Djeca s desne strane crno - crvenog stabla su oni zadaci koji su do sada potrošili više procesorskog vremena (Tanenbaum & Bos, 2015).

Potpuno pošten *scheduler* uvijek uzima čvor u stablu koji se nalazi na krajnje lijevoj poziciji, tj. onaj čvor koji je do sad najmanje vremena imao procesor. Povremeno, *scheduler* povećava vrijednost izvođenja zadataka *vruntime* na temelju vremena koje je do sad proveo u izvođenju. Tu vrijednost uspoređuje s trenutnim krajnjim lijevim čvorom u stablu. Ako zadatak koji je trenutno u obradi ima manju *vruntime* vrijednost, nastaviti će svoju obradu. Inače, bit će umetnut na odgovarajuće mjesto u

crveno - crnom stablu i procesor će biti dodijeljen onom zadatku koji odgovara novom krajnje lijevom čvoru.

Kako bi se uzele u obzir razlike u prioritetima zadataka i njihove *nice* vrijednosti, potpuno pošten *scheduler* mijenja brzinu kojom virtualno vrijeme zadataka prolazi kada je zadatak na procesoru. Za zadatke nižeg prioriteta vrijeme brže prolazi, njihova *vruntime* vrijednost se brže povećava (Tanenbaum & Bos, 2015). Ovisno o ostalim zadacima, zadaci nižeg prioriteta izgubit će procesor i vratiti se u stablo prije nego što bi se vratili zadaci višeg prioriteta. Na taj način, *scheduler* izbjegava korištenje zasebnih struktura za zadatke različitog prioriteta (Tanenbaum & Bos, 2015).



Slika 10 - *Crveno - crno stablo.*

2. Upravljanje i konfiguracija

2.1. Upravljanje procesima

2.1.1. Stvaranje procesa

Korisnik ne može puno utjecati na samo stvaranje procesa, već može utjecati na način da mijenja PATH varijablu pri sistemskom pozivu `exec()`. Poziv `exec()` prvim parametrom zahtjeva od programa putanju do programa kojim će se zamijeniti trenutna slika memorije. Ako nije proslijeđena apsolutna putanja i ako nije postavljena PATH varijabla okruženja, implementacija `exec()` funkcije će pretraživati memoriju na lokacijama `/bin` i `/usr/bin`. Ako je varijabla PATH postavljena, onda će pretraživati po tim putanjama (Kerrisk, 2019a).

Na primjer, ako promijenimo PATH varijablu da sadrži samo putanju trenutno aktivnog direktorija i nakon toga pokrenemo naredbu `ls`, `ls` neće biti obrađen jer se neće moći pronaći program `ls`.

```
PATH=$(pwd) ls
```

Nakon te pokrenute naredbe, dobit ćemo izlaznu poruku od ljuske da program `ls` nije pronađen.

```
bash: ls: command not found
```

2.1.2. Dateotečni sustav `proc`

Datotečni sustav `proc` je prividni datotečni sustav koji omogućuje korisniku pogled na jezgrine strukture vezane uz procese. Većina datoteka u `proc` sustavu je samo otvorena za čitanje, iako postoje neke koje omogućuju korisniku da zapisuje i mijenja varijable. Direktorij `/proc` sadrži direktorije za svaki proces u sustavu koji je u *running* stanju (Slika 11). Nazivi direktorija su PID vrijednosti procesa i unutar direktorija postoje datoteke koje sadrže informacije o procesu (Slika 12). Direktorij `/proc/self` simbolična je poveznica koja vodi do trenutnog procesa. Na primjer, pokrenemo li naredbu: (Kerrisk, 2020b)

```
ls -l /proc/self/exe
```

dobit ćemo kao izlaz proces `ls` koji smo pokrenuli: `/proc/self/exe -> /usr/bin/ls`

```
bolfijaner@bolfijaner:~$ ls /proc
1     1246  13    1338  1402  154  254   4     482   5669  619   6443  929        diskstats    keys         mtrr         sys
10    125   130   134   1408  16   26    4047  486   570   6194  6444  987        dma          key-users    net          sysrq-trigger
1010  1251  1304  1345  1410  17   27    454   493   5732  6195  657   988        driver       kmsg         pagetypeinfo sysvipc
1013  1258  1309  1357  1419  173  28    455   499   5972  632   662   acpi       execdomains  kpagecgroup  partitions   thread-self
1016  126   131   1359  1423  18   29    456   500   600   635   666   asound     fb           kpagecount   pressure     timer_list
1030  1266  1313  136   143   2    3     463   517   6040  636   676   buddyinfo  filesystems  kpageflags   sched_debug  tty
11    1268  1317  1364  1430  21   30    464   518   6067  6364  682   bus        fs           loadavg      schedstat    uptime
1160  127   1321  1383  1438  214  301   465   536   6072  6374  688   cgroups    interrupts   locks        scsi         version
12    1275  1328  139   1461  215  302   466   5374  6102  6385  805   cmdline    iomem        mdstat       self         version_signature
123   128   1331  1392  1473  22   354   467   546   6138  639   808   consoles   ioports      meminfo      slabinfo     vmallocinfo
1232  129   1333  1396  1486  23   371   472   550   6182  6390  816   cpuinfo    irq          misc         softirqs     vmstat
1237  1294  1335  14    15    24   3743  474   5659  6184  6391  859   crypto     kallsyms     modules      stat         zoneinfo
124   1299  1336  140   1531  25   3761  477   5664  6185  6403  9     devices    kcore        mounts       swaps
```

Slika 11 - *Datoteke /proc direktorija i direktoriji aktivnih procesa.*

```
bolfijaner@bolfijaner:/proc$ ls -l 6364
total 0
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 arch_status
dr-xr-xr-x 2 bolfijaner bolfijaner 0 Sep 22 18:02 attr
-rw-r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 autogroup
-r-------- 1 bolfijaner bolfijaner 0 Sep 22 18:02 auxv
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 cgroup
--w------- 1 bolfijaner bolfijaner 0 Sep 22 18:02 clear_refs
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 17:58 cmdline
-rw-r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 comm
-rw-r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 coredump_filter
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 cpuset
lrwxrwxrwx 1 bolfijaner bolfijaner 0 Sep 22 18:02 cwd -> /home/bolfijaner
-r-------- 1 bolfijaner bolfijaner 0 Sep 22 18:02 environ
lrwxrwxrwx 1 bolfijaner bolfijaner 0 Sep 22 17:58 exe -> /usr/bin/mate-terminal
dr-x------ 2 bolfijaner bolfijaner 0 Sep 22 17:58 fd
dr-x------ 2 bolfijaner bolfijaner 0 Sep 22 18:02 fdinfo
-rw-r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 gid_map
-r-------- 1 bolfijaner bolfijaner 0 Sep 22 18:02 io
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 limits
-rw-r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 loginuid
dr-x------ 2 bolfijaner bolfijaner 0 Sep 22 18:02 map_files
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 maps
-rw------- 1 bolfijaner bolfijaner 0 Sep 22 18:02 mem
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 mountinfo
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 mounts
-r-------- 1 bolfijaner bolfijaner 0 Sep 22 18:02 mountstats
dr-xr-xr-x 5 bolfijaner bolfijaner 0 Sep 22 18:02 net
dr-x--x--x 2 bolfijaner bolfijaner 0 Sep 22 18:02 ns
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 numa_maps
-rw-r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 oom_adj
-r--r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 oom_score
-rw-r--r-- 1 bolfijaner bolfijaner 0 Sep 22 18:02 oom_score_adj
-r-------- 1 bolfijaner bolfijaner 0 Sep 22 18:02 pagemap
-r-------- 1 bolfijaner bolfijaner 0 Sep 22 18:02 patch_state
-r-------- 1 bolfijaner bolfijaner 0 Sep 22 18:02 personality
```

Slika 12 - *Datoteke procesa s PID-om 6364, tj. direktorija /proc/6364.*

2.1.3. Naredba `ps`

Naredba `ps` prikazuje informacije o aktivnim procesima. Za brzo listanje aktivnih procesa, pokrećemo naredbu `ps` u terminalu. Naredba `ps` omogućuje korisniku kontroliranje ispisa željenih procesa i efektivno pretraživanje ciljanih procesa u sustavu (svejedno radi li se o vlastitim procesima ili procesima drugog korisnika) (Kerrisk, 2020f). Budući da naredba `ps` ima mnogo parametara koje je također moguće i kombinirati, u nastavku su prikazani najčešće korišteni slučajevi.

2.1.3.1. Ispis bez argumenata

Ako pokrenemo naredbu `ps` bez dodatnih argumenata, dobijemo ispis svih procesa koji su pokrenuti s trenutno korištenog terminala (Kerrisk, 2020f).

```
$ ps

PID   TTY    TIME      CMD
7348  pts/0  00:00:00  bash
7354  pts/0  00:00:00  ps
```

Značenje polja:

- PID - Identifikacija procesa.
- TTY - Terminalni uređaj gdje je proces aktivan.
- TIME - Količina CPU vremena u minutama i sekundama koje je proces iskoristio do sad. Drugim riječima, ukupna količina vremena kojeg je proces iskoristio za pisanje uputa na procesoru.
- CMD – Naziv pokrenutog programa.

2.1.3.2. Odabir procesa

Koristeći parametar `-p` ili `-pid` moguće je odabrati specifične procese za prikaz (Kerrisk, 2020f).

```
$ ps -p 808,1010

PID   TTY    TIME      CMD
808   ?      00:00:00  bluetoothd
1010  ?      00:21:03  pulseaudio
```

U ovom slučaju ispisani su programi koji nisu pokrenuti preko terminala pa je vrijednost TTY stupca označena znakom upitnika (?).

Koristeći parametar `-C` možemo odabrati procese za ispis po njihovim imenima odnosno po imenima njihovih programa (Kerrisk, 2020f).

```
$ ps –C NetworkManager,bash,nepoznati

PID    TTY    TIME         CMD
499    ?      00:00:00     NetworkManager
7348   pts/0  00:21:03     bash
```

Ispisali su se samo procesi s nazivima `NetworkManager` i `bash`, a nepoznati se nije ispisao jer program s tim imenom nije pokrenut.

Pokretanjem naredbe ps sa parametrom `–u` ili `--user` prikazuje popis svih procesa navedenog korisnika sustava. Npr. ispis svih procesa koje je pokrenuo korisnik root dobiva se na sljedeći način (Kerrisk, 2020f):

```
$ ps –u root

PID    TTY    TIME         CMD
1      ?      00:00:25     systemd
2      ?      00:00:00     kthreadd
...
7400   ?      00:00:00     kworker/1:1H
```

Kao i kod ranije spomenutih parametara, može se navesti više korisnika.

```
$ps –u root,bolfijaner
```

Uz procese korisnika `root`, ispisat će i procesi korisnika bolfijaner.

Parametar `–u` prikazuje procese korisnika sa navedenim imenom ili efektivnim identifikatorom korisnika, dok parametar `–U(--User)` prikazuje procesa korisnika sa pravim identifikatorom korisnika. Recimo, prilikom pokretanja naredbe `sudo`, efektivni identifikator korisnika privremeno postaje 0 (`root`), a pravi identifikator korisnika ostaje isti (Kerrisk, 2020f).

Po istom principu funkcioniraju i sljedeći parametri. Parametar `–g` ispisuje procese koju pripadaju grupi korisnika koju navedemo u parametru, dok parametar `–G` ispisuje procese od grupe korisnika s pravim identifikatorom (RGID – real group ID). Parametrom `–s(--sid)` odabiremo one procese koji su pokrenuti u određenoj sesiji. Sesija je vrijeme povedeno od početka rada s ljuskom do kraja rada. Dakle, ako više puta počinjemo rad s ljuskom, imat ćemo više različitih sesija (Kerrisk, 2020f).

Moguće je ispisati samo procese određenog terminala koristeći parametar `-t (--tty)` (Kerrisk, 2020f).

2.1.3.3. Upravljanje ispisom

Korisniku je omogućena prilagodba ispisa dodavanjem raznih parametara naredbi `ps`. On može, koristeći parametar `-o` sam definirati koje stupce želi da mu budu prikazani. Svaki stupac koji želi prikazati piše iza parametra `-o` odmaknut razmakom i nabraja ih. Nabrojeni stupci odvojeni su zarezom (Kerrisk, 2020f). Recimo, ako želimo prikazati sve procese, tj. njihov efektivni identifikator korisnika, pravi identifikator korisnika, identifikator procesa, naredbu i njezine argumente, možemo pokrenuti sljedeći primjer:

```
$ ps -e -o euid,uid,pid,args

EUID   UID    PID    COMMAND
...
0      0      666    [irq/122-iwifi]
199    199    676    /usr/bin/whoopsie -f
115    115    688    /usr/sbin/kerneloops
0      0      808    /usr/lib/bluetooth/bluetoothd
...
```

Postoje također i unaprijed definirani oblici ispisa poput parametra `-F` koji je ekvivalentan parametru `-o uid,pid,ppid,c,sz,rss,psr,start_time,tty,time,cmd` (Kerrisk, 2020f).

```
$ ps -e -F

UID    PIDPPID   C   SZ     RSS    PSR    STIME     TTY    TIME      CMD
root   1   0      0   41685  7272   0      Sep18     ?      00:00:27  /sbin/init-
root   2   0      0   0      0      0      Sep18     ?      00:00:00  [kthreadd]
...
```

Značenje polja (Kerrisk, 2020f):

- UID – Prikazuje efektivni korisnikov identifikator.
- PPID – Prikazuje identifikator procesa roditelja.

- C – Prikazuje CPU vrijeme podijeljeno s vremenom koliko je proces bio u running stanju izraženo u postotku.
- SZ – Prikazuje broj stranica fizičke memorije koju proces zauzima. U tu vrijednost su uključeni programski kod, podaci i prostor na stogu.
- RSS – Prikazuje fizičku memoriju koju je proces koristio bez zamjene (u kilobajtima).
- PSR – Prikazuje kojem procesoru ili jezgri je taj proces trenutno dodijeljen.
- STIME – Prikazuje vrijeme ili datum početka rada procesa. Godina će biti prikazana samo u slučaju da proces nije započeo rad iste godine kao i naredba `ps`.

Korisno je znati u kojem je stanju proces. Na primjer:

```
$ ps –e –o user,pid,cmd,state

USER         PID   CMD                          S
root         1     /sbin/init splash            S
bolfijaner   1357  mate-maximus                 S
bolfijaner   7930  ps –e –o user,pid,cmd,state  R
...
```

Proces može biti u sljedećim stanjima (Kerrisk, 2020f):

- D - Blokirano stanje koje nije moguće prekinuti.
- I - Stanje mirovanja.
- R - Radno stanje procesa.
- S - Blokirano stanje koje je moguće prekinuti.
- T - Stanje procesa prekinuto signalom SIGTSTP ili SIGSTOP.
- t - Proces je prekinuo debugger.
- W - U procesu straničenja.
- X - Mrtvo stanje (ne bi se smio prikazati).
- Z - Zombie proces.

2.1.3.4. Modifikatori ispisa

Kada korisnik zna ispisati željene procese, taj ispis može dodatno urediti koristeći modifikatore ispisa. To su također parametri naredbe `ps`. Recimo, korisnik može ispisati hijerarhiju procesa, tj. stablasti prikaz procesa. Dodamo li na naredbu `ps` parametar `-f` (`--forest`) dobijemo sljedeći izlaz (Kerrisk, 2020f):

```
$ ps -e -forest
PID   TTY    TIME      CMD
...
7339  ?      00:00:21  \_mate-terminal
7348  pts/0  00:00:00  |   \_bash
8155  pts/0  00:00:00  |      \_ps
8020  ?      00:00:00  \_mate-notificati
...
```

Također, jedan od korisnih modifikatora procesa je parametar `--sort` naredbe `ps`. Parametar `--sort` omogućuje sortiranje procesa po nazivima stupaca koje odredi korisnik. Moguće je proslijediti listu stupaca uz znak + ili – ako želimo da ta lista bude sortirana ulzlazno (+) ili silazno (-) (Kerrisk, 2020f).

```
$ ps -user bolfijaner -o pid,cmd -sort +pid

PID   CMD
987   /lib/systemd -user
988   (sd-pam)
1010  /usr/bin/gnome-keyring-deamon -deamonize -login
1016  mate-session
...
```

Ponašanje i ispis procesa također je moguće kontrolirati koristeći određene varijable okruženja, no to se ne preporuča osim postavljanja vrijednosti varijable CMD_ENV i PS_PERSONALITY. Korištenjem tih varijabli moguće je kontrolirati koji standard prati naredbu `ps`, što je obično Linux (Kerrisk, 2020f).

2.1.4. Naredba `top`

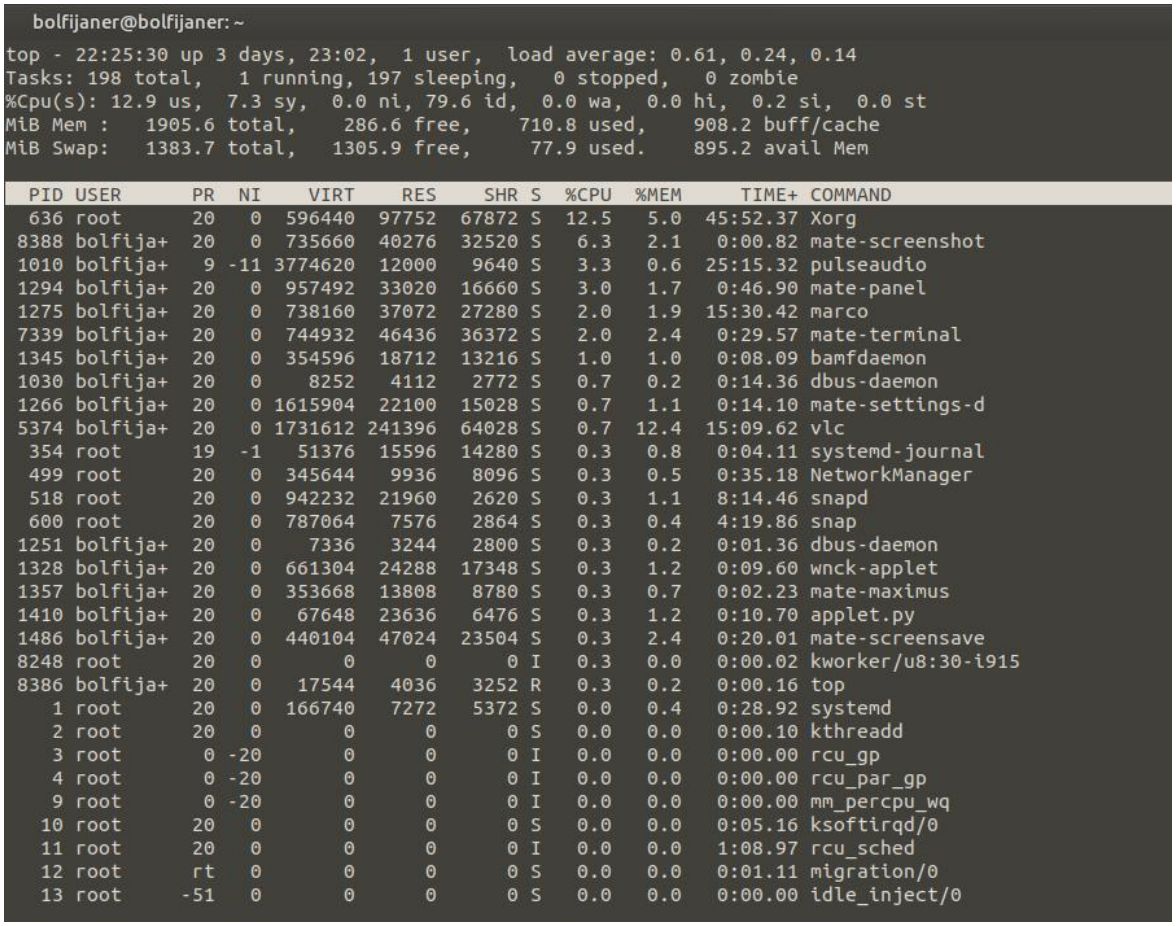
Naredba `top` je dinamički oblik naredbe `ps`. Naredba `ps` prikazuje procese u trenutku izvršenja naredbe, dok naredba `top` ažurira informacije o procesima u stvarnom vremenu. Slika 13 prikazuje ispis naredbe `top`. U zaglavlju ispisa naredbe `top` prikazani su razni podaci (Kerrisk, 2019b).

U prvom redu zapisano je trenutno vrijeme (22:25:30), nakon toga zapis koliko dugo je prošlo od zadnjeg pokretanja sustava (3 dana, 23:02 sati). Iza vremena zapisana je informacija o broju prijavljenih korisnika (jedan korisnik) te na kraju prvog reda piše prosječno opterećenje procesora kroz zadnju minutu (0.61), zadnjih pet minuta (0.24) i zadnjih 15 minuta (0.14).

Drugi red zaglavlja prikazuje ukupan broj procesa (198) i kategorizira ih po stanjima running (1), sleeping (197), stopped (0) i zombie (0).

Treći red prikazuje udio vremena koji je procesor proveo u korisnikovom okruženju jezgre (12.9 us), udio vremena u okruženju jezgre (7.3 sy), udio vremena koji je proveo izvršavajući procese kojima je promijenjen niceness (0.0 ni), udio vremena proveden u ready stanju (79.6 id), koliko je čekao na I/O operacije (0.0 wa), koliko vremena je proveo obrađujući hardverske prekide (0.0 hi), isto za softverske prekide (0.2 si) i na kraju reda koliki je udio vremena dobiven od druge virtualne mašine (0.0 st) uz uvjet da se naredba `top` izvodi na nekog virtualnoj mašini.

Četvrti i peti red prikazuju ukupnu količinu radne memorije u megabajtima. U četvrtom redu prvo je prikazana ukupna radna memorija (1905.6), nakon radne memorije prikazana je slobodna memorija (286.6), iskorištena memorija (710.8) i *buff/cache* memorija - memorija koji zauzimaju predmemorirane stranice i međuspremnici u jezgri (988.2). Peti redak na analogni način poput četvrtog predstavlja virtualnu memoriju. Razlika *available* memorije i *free* memorije je u tome što *available* memorija uključuje i stranice koje se mogu zamijeniti ako ponestane stvarne memorije.



Slika 13 – *Prikaz top sučelja.*

2.1.4.1. Pokretanje naredbe `top`

Prilikom upisivanja naredbe `top` u terminal, korisnik može dodati jedan ili više sljedećih parametara kako bi prilagodio ispis svojim potrebama. Ovo su neki od najkorisnijih parametra pri pokretanju `top`-a (Smith, 2006).

- `-d` (delay) – Ovim parametrom korisnik određuje koliko često će top ažurirati podatke o procesima. Obično je postavljeno na pet sekundi.
- `-b` (batch-mode) – Određuje režim paketne obrade. Korisno je za slanje izlaznih vrijednosti iz `top` programa nekom programu u obliku datoteke.
- `-p` (pid) – Ovaj parametar služi da korisnik nadgleda samo one procese čiju je PID vrijednost upisao nakon parametra `-p`. Može upisati do 20 PID vrijednosti, koristeći ovu opciju više puta.

- `-u` ili `-U` (user) – Ovim parametrom moguće je prikazati procese određenog korisnika. Parametrom `-u` prikazujemo procese čiji je efektivni korisnički identifikator jednak onome zadanom pri upisu naredbe, dok `-U` prikazuje sve procese pod tim korisnikom koji je naveden (svejedno radi li se o efektivnom ili pravom korisničkom identifikatoru).
- `-H` – Ovim parametrom naredba `top` prikazuje individualne dretve.
- `-n` – Ovim parametrom određujemo koliko će se puta `top` ažurirati prije završetka rada.

2.1.4.2. Upravljanje sučeljem

Osim upisivanja raznih parametara pri pokretanju `top` naredbe, moguće je za vrijeme rada `top` programa definirati ispise. Slijedi popis tipki i njihovih akcija (Kerrisk, 2019b; Smith, 2006):

- `Enter` ili `space` tipkama možemo odmah ažurirati informacije o procesima.
- Tipke `?` ili `h` prikazuju pomoć pri korištenju naredbe `top`.
- Tipkom `q` izlazimo iz programa.
- Tipkama `d` ili `s` možemo mijenjati vrijeme ažuriranja, poput parametra `-d` koji se upiuje kod pokretanja naredbe. Nakon pritiska tipki prikazuje se *prompt* za upis željene vrijednosti i potvrđuje se tipkom Enter. Nije dopušteno upisati negativnu vrijednost. Unos 0 zahtjeva od programa kontinuirano ažuriranje što onemogućuje sučelju da tako brzo prikazuje sve promjene.
- Tipkom `r` moguće je promijeniti *niceness* vrijednost određenog procesa. Ako se ne napiše PID procesa kojemu se mijenja *niceness* ili ako se upiše negativna vrijednosti, `top` će uzeti u obzir prvi prikazani proces. Ako upišemo PID vrijednost 0, misli se na proces `top`.
- Tipkom `k` moguće je poslati signal procesu. Program će tražiti unos PID vrijednosti onog procesa kojem želimo poslati određeni signal.
- Tipkama `shift + m` prikaz procesa će se razvrstati po tome koliko memorije troše.

2.1.5. Naredba `kill`

Naredba `kill` omogućuje korisniku slanje signala procesima. Sintaksa naredbe izgleda ovako: (Shotts, 2009)

```
kill [-signal] PID.
```

Ako ne odredimo koji signal se šalje, po zadanoj vrijednosti šalje se signal SIGTERM. Popis svih signala koji korisnik može poslati nalazi se u Tablici 1. Korisnik mora biti vlasnik procesa kojeg želi terminirati. Ostalim procesima signal može poslati samo korisnik `root`. U sljedećem primjeru prikazat ćemo proces s imenom vlc, prekinuti ga u izvođenju slanjem signala pod rednim brojem 9 (SIGKILL) i ponovno izlistati procese kako bi se uvjerili da je stvarno uklonjen (Kerrisk, 2014a).

```
$ ps –C vlc

PID    TTY    TIME      CMD
5374   ?      00:15:39  vlc

$ kill -9 5374

$ ps –C vlc
PID    TTY    TIME      CMD
```

Sljedeći primjer pokazat će kako možemo zaustaviti neki proces te ga kasnije ponovno pokrenuti. Prvo prikažemo signale naziva caja. Kada smo pročitali identifikator procesa za taj proces, šaljemo mu signal SIGSTOP naredbom `kill`. Prikazujemo stanje procesa caja kako bi utvrdili da je on stvarno u stanju T – zaustavljen signalom. Kada želimo da proces nastavi svoje dosadašnje izvođnje, šaljemo mu signal SIGCONT. Nakon signala, proces s PID-om 1321 iz zaustavljenog se stanja vraća u svoje dosadašnje stanje.

```
$ ps -C caja
PID   TTY    TIME      CMD
1321  ?      00:01:38  caja

$ kill -SIGSTOP 1321

$ ps -C caja -o pid,cmd,s
PID   CMD    S
1321  caja   T

$ kill -SIGCONT 1321

$ ps -C caja -o pid,cmd,s
PID   CMD    S
1321  caja   S
```

2.1.6. Naredba `killall`

Naredba `kill` šalje signal jednom procesu, a naredba `killall` šalje signale odabranoj grupi procesa. Kao i kod naredbe `kill` procese možemo odabrati imenom ili PID-om (Shotts, 2009). Slijedi primjer kako dodatkom parametra `-u` signal šaljemo svim procesima korisnika bolfijaner.

```
$ killall -u bolfijaner
```

Dodatkom parametra `-I`, možemo poručiti programu `killall` da ne pravi razliku između velikih i malih slova prilikom čitanja imena procesa (Kerrisk, 2018).

```
$ killall bAsH
bAsH: no process found
$ killall bAsH -I
```

Još jedan od korisnih parametra je `-s` koji omogućuje korisniku slanje određenog signala umjesto zadanog SIGTERM. Signal se specificira jednako kao kod naredbe `kill`.

Dodatkom parametra `-r` ljuska će interpretirati ime procesa kao regularni izraz. Na primjer, želimo ubiti grupu procesa kojima naziv počinje slovima gvfs, koristimo naredbu (Kerrisk, 2018):

```
$ killall -r gvfs.*
```

2.2. Kontrola poslova

Posao je koncept vezan uz ljusku odnosno to je proces koji je pokrenut iz ljuske tipično u interaktivnoj sesiji. Kontrola poslova omogućava interakciju sa poslovima u pozadini, obustavu procesa u prvom planu i slično. Na Posix sustavima, poslovi se implementiraju kao procesne skupine (engl. process groups), s tim da je jedan proces vođa grupe. Svaki *tty* (terminal) ima jednu grupu procesa u prvom planu (engl. foreground), u kojem je dopuštena interakcija s terminalom. Korisnik može interaktivno sudjelovati u kontroli poslova koji su u prvom planu tj. korisnik ima prikaz ispisa programa, može slati signale koristeći tipkovnicu i ima mogućnost unosa podataka u program. Sve ostale procesne grupe s istim kontrolnom *tty* vrijednošću smatraju se pozadinskim poslovima (engl. background jobs) i mogu biti aktivni ili obustavljeni. Kada se neki proces izvodi u pozadini, korisnik nije u mogućnosti interaktivno ga koristiti što znači da će posao biti obustavljen signalom SIGTSTP. Korisnik slanjem SIGCONT signala može staviti posao u prvi plan izvođenja (Fox & Ramey, 2018).
Na primjer, naredba `cat &` bude odmah suspendirana, umjesto da radi u pozadini, budući da zahtjeva korisnikov unos.

Pritiskom određenih tipki na terminalu uzrokuje slanje signala svim procesima u grupi procesa koje rade u prvom planu. Kombinacije tipki mogu se konfigurirati naredbom `stty`, a zadane vrijednosti su (Fox & Ramey, 2018):

- Ctrl + Z šalje signal `SIGTSTP` poslu u prvom planu (suspendira posao)
- Ctrl + C šalje signal `SIGINT` poslu u prvom planu (terminira posao)
- Ctrl + \ šalje signal `SIGQUIT` poslu u prvom planu (prekida posao)

Posao u prvom planu može se obustaviti pritiskom tipki Ctrl + Z. Ne postoji način da se referiramo na posao u prvom planu u `bash`-u. Ako postoji posao u prvom planu koji nije `bash`, tada `bash` čeka na završetak tog posla pa se stoga ne može izvršiti niti jedan kod. Sljedeće naredbe rade samo na pozadinskim poslovima.

Kontrola poslova uključuje sljedeće naredbe (Fox & Ramey, 2018):

- `fg:` Pozadinski posao stavlja u prvi plan
- `bg:` Pokreće suspendirani posao u pozadini
- `suspend:` Suspendira ljusku

Ostale naredbe za komunikaciju s poslovima uključuju:

- `jobs` [*options*] [*jobspec* ...]: lista obustavljenih i pozadinskih poslova. Opcije uključuju:
  - `-p` (lista samo identifikacije procesa)
  - `-s` (lista samo obustavljene poslove)
  - `-r` (lista samo pozadinske poslove koji su aktivni). Ako je jedan ili više *jobspecs* specificiran, svi ostali poslovi su ignorirani.
- `kill:` slanje signala

Kontrola poslova dozvoljava da može postojati više aktivnih stvari u jednoj sesiji terminala. To je bilo vrlo bitno ranije kada se koristio samo jedan terminal na zaslonu i nije bilo načina za stvoriti više virtualnih terminala. Na suvremenim sustavima i hardveru postoji više izbora. Možemo, na primjer, pokrenuti `screen` ili `tmux` unutar terminala da dobijemo više virtualnih terminala, ili se sa X sesijom može se otvoriti više `xterm` ili sličnih terminalnih emulatora.

No, ponekad jednostavna kontrola poslova `suspend` i `bg` može dobro poslužiti. Moguće je da se pokrene *backup* i traje duže od očekivanog. Može se obustaviti sa kombinacijom tipki Ctrl + Z, te zatim staviti u pozadinu s naredbom `bg`. Tada imamo slobodnu komandnu liniju za upisivanje naredbi, tako da se u terminalu može raditi nešto drugo dok se ujedno radi i *backup* (Ward, 2015.).

2.1.1. Specifikacije poslova

Specifikacija poslova ili *jobspec* je način komunikacije procesom koji izvodi posao. *Jobspec* može biti (Fox & Ramey, 2018):

- `%n` se odnosi na posao broja *n*
- `%str` se odnosi na posao koji je započet sa zapovijedi koja počinje sa `str`. Greška je ako postoji više od jednog takvog posla.
- `%?str` se odnosi na posao koji je započet sa zapovijedi koja sadrži `str`. Greška je ako postoji više od jednog takvog posla.
- `%%` ili `%+` se odnosi na sadašnji posao: Najnedavnije pokrenut u pozadini ili obustavljen iz prvog plana. Naredbe `fg` i `bg` će raditi na ovom poslu ako specifikacija posla nije određena.
- `%-` za prethodne poslove (posao koji je bio `%%` prije sadašnjeg)

Moguće je pokrenuti proizvoljnu naredbu sa specifikacijom poslova, koristeći

```
jobs -x ''cmd args...''.
```

To zamjenjuje argumente, koji izgledaju kao jobspec, s PID vrijednostima odgovarajućih procesa koji su vođe grupa procesa, te se zatim pokreće naredba. Na primjer, naredba `jobs -x strace -p %%` će spojiti `strace` sa sadašnjim poslom (korisniji ako se pokreće u pozadini nego kao obustavljen).

Naposljetku, sama specifikacija poslova može se koristiti kao naredba:

Naredba `%1` je ekvivalent naredbi `fg %1`

Naredba `%1 &` je ekvivalentna naredbi `bg %1`

2.3. Niceness

2.3.1. Naredba `nice`

Kao što je već ranije spomenuto, *niceness* određuje koliko često neki proces ili procesna dretva dolazi na red za izvođenje. Naredbu `nice` koristimo kada želimo pokrenuti program s određenim prioritetom, a naredbu `renice` koristimo kako bi izmijenili prioritet već aktivnog programa. Na primjer, ako na računalu postoji neki posao koji je važniji od drugih poslova i korisnik želi povećati prioritet, može upotrebom *niceness*-a (Kerrisk, 2020a).

Naredbom `nice` možemo dodijeliti prioritet na tri načina. Jedan od načina je unošenjem crtice ispred vrijednosti, drugi način je određivanje prioriteta iza parametra `-n` i treći način je određivanje prioriteta parametra `--adjustment=`. Naredbena linija izgleda ovako (Kerrisk, 2020a):

```
nice [argument] [command [command-arguments]].
```

Ako izostavimo vrijednost prioriteta, `nice` koristi 10 kao zadanu vrijednost. Opseg mogućih prioriteta je između -20 i +19. Negativne vrijednosti imaju veći prioritet. Samo `root` korisnik može pokrenuti program s negativnim prioritetom, no svaki korisnik može pokrenuti `nice` s nižim prioritetom. Ako se pokrene program bez naredbe nice, imat će prioritet 0.

Npr. `nice -10 number_count data.txt` će pokrenuti program number_count s prioritetom 10 i prenositi u datoteku `data.txt`.

2.3.2. Naredba `renice`

Naredba `renice` služi za promjenu prioriteta aktivnog programa bez njegovog prekida rada. Sintaksa za naredbu `renice` izgleda ovako (Kerrisk, 2014b):

```
renice priority [[-p] pids] [[-g] pgrps]  [[-u] users].
```

Nakon same naredbe `renice` potrebno je odrediti prioritet, nakon čega se upisuje jedan ili više identifikatora procesa za koje korisnik želi mijenjati prioritet izvođenja. Nakon PID vrijednosti, korisnik upisuje jedan ili više grupnih ID brojeva (pgrps), te naposljetku dodaje ili jedno ili više korisničkih imena (engl. users).

Npr. `renice 5 14637 -u student` će postaviti prioritet 5 za proces koji ima PID 14637 koji je vlasništvo korisnika student.

Zaključak

A process in Linux is like an iceberg: you only see the part above the water, but there is also an important part underneath. (Tanenbaum & Bos, 2015)

Linux kao operacijski sustav možda može izgledati zastrašujuće ili zbunjujuće u početku za neiskusnog korisnika, no kako bi se u potpunosti iskoristio potencijal OS-a potrebno se upoznati s njegovim konceptima. Kao što je i Tanenbaum napisao, Linux je poput sante leda: iako vidimo samo dio sustava „iznad vode“ kroz korisničko sučelje, ovim seminarom pokazano je kako se najbitniji dio zapravo izvršava u unutrašnjosti.

U radu je objašnjeno kako korisnik može upravljati stvaranjem procesa kroz mijenjanje varijable PATH i naredbama nice i renice, također upotrebom naredbi ps, jobs, top, kill i killall prikazano je kako nadgledati procese, slati im signale i manipulirati procesima neovisno radi li se o interaktivnim poslovima ili ostalim procesima.

Prikaz objašnjenih naredbi i njihov opis je zapisan u standardnom formatu (sa stajališta Linux sustava), iako se mogu koristiti i drugi formati prilagođeni korisniku (poput BSD koji se često koristi u Macintosh računalima).

Postoje i grafički prikazi programa za upravljanje procesima operacijskog sustava kao na primjer *„System Monitor“*, no nisu navedene jer već opisane naredbe (`ps`, `top`, `kill` i sl.) služe istoj svrsi, a prednost im je što se nalaze na gotovo svim Linux sustavima, čak i onima koji ne koriste grafičko sučelje (npr. superračunalo Bura).

Opisani koncepti i naredbe u ovom radu su složeni, ali njihovo razumijevanje i aktivno korištenje ključno je za efikasno korištenje i održavanje sustava.

Bibliografija

1. Brian Fox, & Chet Ramey. (2018). Bash(1). In *General Commands Manual*. Kernel.org. https://www.man7.org/linux/man-pages/man1/bash.1.html
2. Michael Kerrisk. (2014a). Kill(1). In *User Commands*. Kernel.org. https://man7.org/linux/man-pages/man1/kill.1.html
3. Michael Kerrisk. (2014b). Renice(1). In *User Commands*. https://man7.org/linux/man-pages/man1/renice.1.html
4. Michael Kerrisk. (2017a). Fork(2). In *Linux Programmer's Manual*. Kernel.org. https://man7.org/linux/man-pages/man7/man-pages.7.html
5. Michael Kerrisk. (2017b). Signal(2). In *Linux Programmer's Manual*. Kernel.org. https://man7.org/linux/man-pages/man2/signal.2.html
6. Michael Kerrisk. (2018). Killall(1). In *User Commands*. https://man7.org/linux/man-pages/man1/killall.1.html
7. Michael Kerrisk. (2019a). Exec(3). In *Linux Programmer's Manual*. https://man7.org/linux/man-pages/man3/exec.3.html
8. Michael Kerrisk. (2019b). Top(1). In *User Commands*. https://man7.org/linux/man-pages/man1/top.1.html
9. Michael Kerrisk. (2020a). Nice(1). In *User Commands*. https://man7.org/linux/man-pages/man1/nice.1.html
10. Michael Kerrisk. (2020b). Proc(5). In *Linux Programmer's Manual*. https://man7.org/linux/man-pages/man5/proc.5.html
11. Michael Kerrisk. (2020c). Sigaction(2). In *Linux Programmer's Manual*. https://man7.org/linux/man-pages/man2/sigaction.2.html
12. Michael Kerrisk. (2020d). Signal(7). In *Linux Programmer's Manual*. Kernel.org. https://man7.org/linux/man-pages/man7/signal.7.html
13. Michael Kerrisk. (2020e). Wait(2). In *Linux manual page*. Kernel.org. https://man7.org/linux/man-pages/man3/waitpid.3p.html
14. Michael Kerrisk. (2020f). Ps(1). In *User Commands*. Kernel.org. https://man7.org/linux/man-pages/man1/ps.1.html

15. Moshe Bar. (2000, May 1). *The Linux Signals Handling Model | Linux Journal*. https://www.linuxjournal.com/article/3985

16. Smith, R. W. (2006). Linux +. In *Linux +* (str. 289–292).

17. Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (Fourth edition). Pearson.

18. Torvalds. (2020). *Linux Kernel* (5.8.10.) [Computer software].

19. Ward, B. *How Linux Works, 2nd Edition*. publisher not identified. http://proquest.safaribooksonline.com/9781457185519

20. William E. Shotts Jr. (2009). The Linux command line. In *The Linux command line* (str. 121–126).

Popis slika

Popis tablica