Process Control

- U prethodnoj glavi smo **definisali kontekst procesa** i **algortime** za **manipulaciju** sa **kontekstom procesa**,

- ova glava opisuje korišćenje i implementaciju SC koji kontrolišu kontekts procesa.
  - **fork** SC kreira novi proces
  - **exit** SC završava izvršavanje programa
  - **wait** SC omogućava roditelju da sinhroniše svoje izvršavanje sa **exitom** procesa deteta
  - **signals** informišu procese o asinhronim događajuma i zato što **kernel sinhroniše izvršavanje wait i exit SC** preko **signala,** oni će biti obrađeni prvi
  - **exec** SC dozvoljava procesu da pozove novi program prepisujući svoj adresni prostor sa executabilnim image datoteke
  - **brk** SC dozvoljava procesu da alocira memoriju dinamički, kao što OS dozvoljava da user stack raste dinamički, koristeći sličan mehanizam kao brk.

Na kraju će biti objašnjene **glavne petlje** u **shell** i **init** procesima.

SC for processes

- Na slici su dati odnosi između SC opisanih u ovom poglavlju i memorijskih algoritama opisanih u zadnjem poglavlju. Svi SC uglavnom koriste i **sleep** i **wakeup**, dok **exec**(SC) koristi SC calls za datoteke.

System calls dealing with **memory management**				System calls dealing with **syncronization**			Miscellanous	
fork	exec	brk	exit	wait	signal	kill	setpgrp	setuid
dupreg attachreg	detachreg allocreg attachreg growreg loadreg mapreg	growreg	detachreg					

Process creation

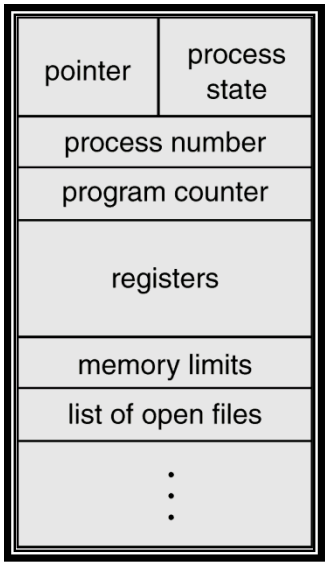
- **Jedini način** koji korisnik ima za kreiranje novog procesa na UNIX operativnom sistemu je preko **fork SC**. Proces koji poziva fork SC se naziva **roditeljski proces**, dok se novo kreirani proces **naziva dete proces**.
- **Sintaksa za fork SC je:**
- **pid=fork();**
- **Na povratku** iz **fork SC**, **2** procesa imaju **potpuno identične user-level kontekste izuzev za pid koja je povratna vrednost fork SC-a**. Za **roditeljski** proces **pid= child PID**, dok je za **dete proces pid=0**.
- **Proces 0**, koji **interno kreiran od kernela** je **jedini proces koji se ne kreira preko fork SC-a.**
- **Kernel** obavlja **sledeću sekvencu operacija** za **fork SC**
  - **Alocira slot u PT** za novi proces
  - Dodeljuje novi jednistveni ID za proces dete
  - **Pravi logičku kopiju konteksta roditeljskog procesa**. Pošto izvesni delovi procesa, kao što su text region, mogu da se dele između procesa, kernel ponekad **inkrementira RC** za region umesto da kopira kopira ceo region na novu fizičku lokaciju u memoriju
  - **Inkrementira RC za datoteke** u **inode tabeli** za datoteke dodeljene procesu
  - **Vraća ID broj procesa** deteta roditeljskom procesu i 0-tu vrednost za proces dete

Process creation

- **Implementacija fork SC nije trivijalna**,
- zato što se proces dete pojavljuje na početku izvršavanja,
- Sama realizacija zavisi do UNIX-a
  - da li primenuje **DP**
  - ili
  - swaping tehniku
- No pretpostavimo, da sistem ili **dovoljno memorije za ceo proces dete**.

algorithm fork

- **algorithm fork**
  - input: none
  - output: to parent process child PID number, to child process 0
- {
- **check** for **available kernel resources**;
- **get free proc table slot**, **unique PID number**;
- **check** that user not running **too many processes**;
- **mark child** state "**being created**";
- copy data from parent ProcessTable slot to **new child slot**;
- **increment counts** on **current directory inode** and **changed root** (if applicable);
- increment **open file counts** in **FileTable**;
- **make copy** of **parrent context** (**u area, text, stack**) in memory;
- **push dummy system level context layer** onto **child system level context**;
- **dummy context** contains data allowing **child process to recognize itself**,
- and **start running from** here when **scheduled**



algorithm fork

- **if(executing process is parent process)**
- {
- **change child state** to "**ready to run**";
- **return(child ID);**
- }
- else /* **executing process is child process***/
- {
- initialize **u area** timing fields;
- **return(0)**; /*to user*/
- }
- }

fork description-check first

- Kernel **prvo mora da proveri raspoloživost resursa** da bi se fork uspešno obavio.
  - Na **swapping sistemima**, potrebno je imati **dovoljno prostora ili u memoriji** ili na **swapu** (disku) za proces-dete.
  - Na **paging sistemima potrebno je samo alocirati memoriju** za **pomoćne tabele** kao što su **tabele stranica.**
  - Naravno ukoliko su resursi neraspoloživi **fork SC** će otkazati.
- **Kernel prvo nalazi slot u PT** da bi otpočeo **konstrukciju konteksta** za proces **dete**, ali mora proveriti da nije već suviše procesa već u izvršavanju.
  - Zatim se nalazi **jedinstveni ID** broj za novi proces, **za jedan veći od poslednjeg PID-a** .
  - Ako drugi proces već ima taj PID, kernel ide na sledeći veći broj, a ako se dostigne neki maksimalni broj, počinje se od 0.
  - Mnogi procesi će trajati kratko u memoriji, pa će obaviti exit, i fork će u svakom slučaju **naći slobodan PID**.
- UNIX **mora udariti limit korisniku** za **maksimalni broj procesa** koje user može da izvršava simultano.
  - Takođe ako su svi ulazi u PT zauzeti, proces mora da čeka.
  - Na drugoj strani superuser nema limite, **jedino broj ulaza u PT**,
  - a takođe superuser može naterati svaki proces da prekine aktivnost, odnosno da nasilno obavi exit SC.

fork description-pt entry creation

- **Kernel potom inicijalizuje PT slot za proces dete**,
  - **kopirajući razna polja iz roditeljskog slota**.
  - Na primer, **dete nasleđuje od roditelja**
    - **realni i efektivni user ID,**
    - **roditeljsku procesnu grupu**,
    - **roditeljsku nice vrednost** koja se koristi za izračunavanje prioriteta procesa.
- Kernel dodeljuje roditeljski PID u slot deteta da zna ko mu je roditelj.
  - Zatim se proces-dete ubacuje u stablo procesa i
  - **inicijalizuju mu se razni schedulling parametri**, kao što je
  - inicijalni prioritet,
  - inicijalno CPU korišćenje i drugi timing parametri,
-
- čime se završava kreiranje stanja procesa "**being created**".

fork description-open files

- Kernel zatim **prilagođava RC za datoteke** koje se detetu automatski dodeljuju.
  - **Prvo**, <**current directory**>
    - **dete će stanovati u tekućem direktorijumu roditeljskog procesa**.
    - Broj procesa koji koriste **tekući direktorijum se povećava za** 1, što se mora upisati u inode RC (in-core inode).
  - **Drugo**, <**root directory**>
    - **ako je proces roditelj ili** bilo koji od njegovih predaka **ikada pomenio svoj root direktorijum (chroot SC),**
    - **dete će naslediti taj promenjeni root** i za njegov inode se mora inkrementirati RC.

- Na kraju, kernel pretražuje **UFDT** za otvorene datoteke roditelja i
  - inkrementira globalni FT **RC za svaku otvorenu datoteku**.
  - Ne samo da dete nasleđuje **prava pristupa za datoteku**, već takođe i **svi pristupi** i **file pointeri** se **takodje dele** zato što oba procesa imaju **identične FT ulaze**.
  - **Efekat forka je sličan kao dup za otvorene datoteke**:
    - novi ulaz u UFDT ukazuje na ulaz u glavnoj FT za otvorenu datoteku,
    - samo što su kod dup-a ulazi u UFDT svi vezani za jedan proces,
    - a kod forka oni su potpuno isti ali **pripadaju različitim procesima.**

fork description-static part

- Kernel je sada **spreman da kreira user-level kontekst procesa deteta**,
  - **alocira memoriju** za **u-area proces-deteta**, **regione**, **pomoćne tabele stranica**,
  - **duplicira svaki region** iz **roditeljskog procesa preko dupreg**
  - **attach-uje** svaki region u proces dete preko **attachreg**.
- **Na swapping sistemima**, kopira se sadržina regiona koji nisu deljivi u novo područje memorije.
- Podsetimo da **u-area** sadrži **ukazivač na ulaz u PT**
  - **to jedina razlika** u kojoj **se inicijalno razlikuju u-area** za **roditelja i dete**,
  - **ali**
  - **posle fork SC** će svako još više razlikovati u **lifetime** procesa deteta.
- Ovo što je opisano do sada predstavlja kreiranje **statičkog dela** konteksta za **proces dete**.

fork description-dynamic part

- Sledi kreiranje **dinamičkog dela konteksta**.
- Kernel kopira **roditeljski kontest layer 1**, koji sadrži **sačuvani registarski kontekst** za **user mod i kernel stack okvir za fork SC**.
  - **Ako se kernel stack kreira u u-area**, kernel automatski kreira child-kernel-stack u **dete's u-area** i to baš onda kada **kreira u-area**.
  - U suprotnom, roditeljski proces **mora kreirati svoju kopiju kernelskog stacka** u privatnu memoriju procesa deteta.
- U oba slučaja **kernelski stack za roditelja** i **dete se identični**.
- Kernel zatim kreira **dummy kontekts layer(2) za proces dete**, koje sadrži sačuvani registarski **kontekst za layer (1).**
  - Postavljaju se **PC i drugi registri u sačuvanom registarskom kontekstu**, tako da se konteksti za dete mogu obnoviti **iako se praktično nikada nisu dogodili**.
  - **Kada kernel kod testira vrednost registra 0**, on će proceniti da li proces roditelj ili dete, i on će upisati tu vrednost u **detetov registarski kontekts** u **layer-u 1**.

fork description

- Kada je **detetov kontekst gotov**, roditelj **završava svoj deo fork SC**,
  - tako što promeni detetovo stanje u "**ready to run**"
  - vrati **detetov PID korisniku**.
- dete kasnije kao i svi drugi procesi **čeka da ga scheduler prozove** i
  - tada dete **prvo kompleira svoj deo fork SC**,
  - tako što se obnovi **kontekst iz layera (2)**
  - **vrati se vrednost 0 kao rezultat fork SC**.
- **Kontekst deteta** je **setovao roditelj**,
- a za kernel,
  - **dete izgleda kao proces**
  - koji se **probudio** nakon **čekanja resursa**.

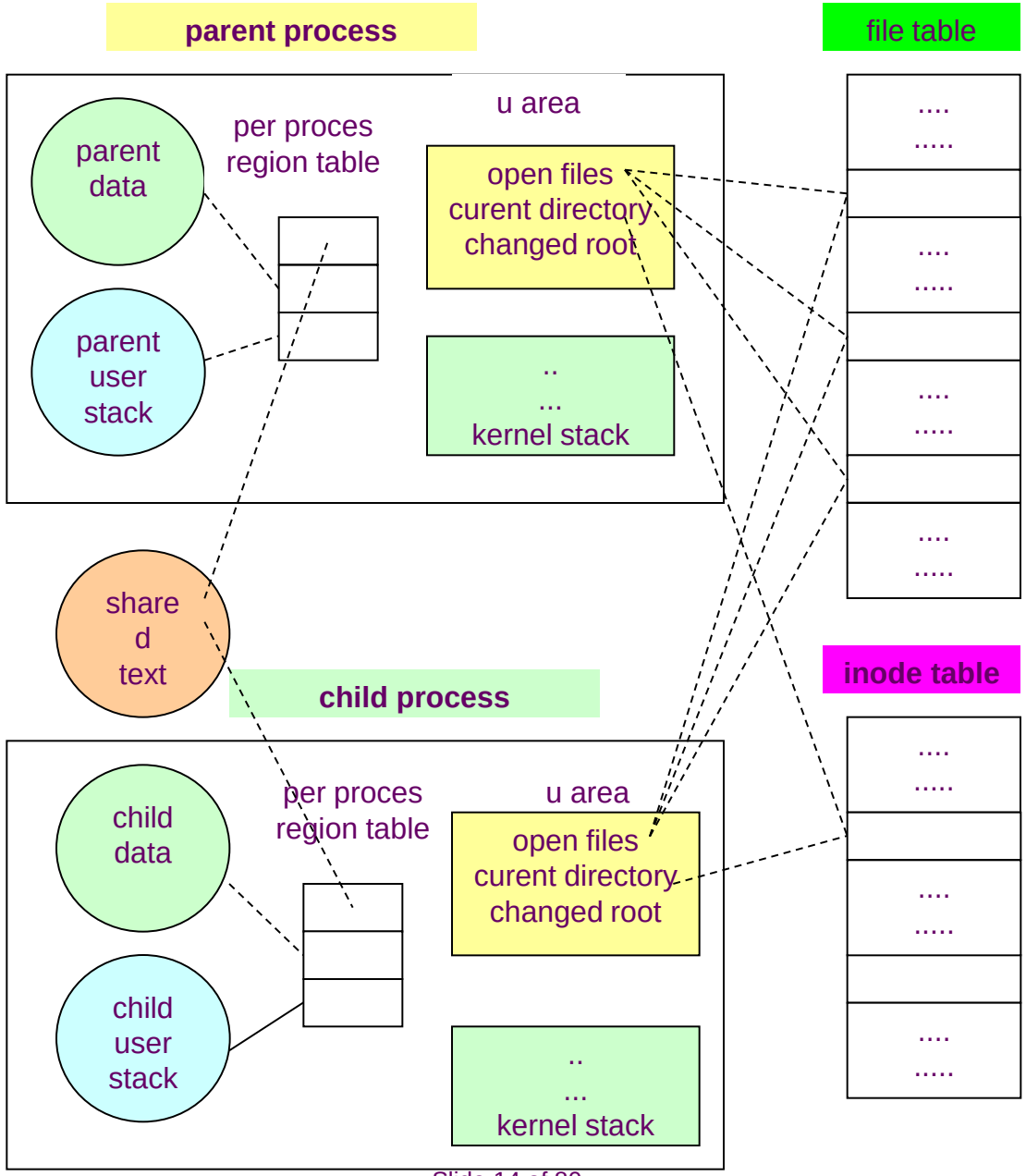
fork description

- Na **slici je dat logički izgled procesa roditelja i deteta** i njihovih odnosa sa drugim kernelskim data strukturama **neposredno pošto se okonča fork SC**.

- Ukratko, **oba procesa dele datoteke koje je roditelj otvorio do vremena pre fork SC**, i RC za sve otvorene datoteke se **uvećava za** 1.

- I roditelj i dete imaju isti **tekući direktorijum** i **promenjeni root**,
- a RC za te direktorijume se uvećava za 1.

- Procesi imaju identični **kopiju text, data i user-stack regiona**,
- a tip regiona ukazuje da li će ih proces deliti ili će praviti fizičke kopije


parent process
file table
u area
per proces
region table
parent
data
parent
user
stack
open files
curent directory
changed root
..
...
kernel stack
....
.....
share
d
text
child process
inode table
child
data
child
user
stack
per proces
region table
u area
open files
curent directory
changed root
..
...
kernel stack

fork example 1

```
#include <fcntl.h>
    int fdrd, fdwt;
    char c;
main (argc, argv)
int argc;
char *argv[]
{
if(argc != 3)  exit(1);
if((fdrd = open(argv[1], O_RDONLY)) == -1)    exit(1);
if((fdwt = creat(argv[2], 0666)) == -1)              exit(1);
fork();
/*both processes execute same code*/
rdwrt();
exit(0);
}
--------------------------------------------------------------------------------
rdwrt()
{
        for(;;)
{
if(read(fdrd, &c, 1) != 1) return;
   write(fdwt, &c, 1);
}
}
```

fork example 1

- Analizirajmo program koji je primer deljenja datoteka preko **fork SC**.
- Korisnik će pozvati program sa **2 parametra**:
  - ime postojeće datoteke i
  - nove datoteke koja će biti kreirana.
- Proces otvara postojeću datoteku, kreira novu datoteku i ako nema grešaka, preko **fork SC** kreira proces dete.
- Interno, kernel pravi kopije roditeljskog konteksta za proces dete, a potom se roditeljski **proces izvršava u jednom adresnom prostoru a dete u drugom**.
- Svaki proces pristupa **privatnim kopijama globalnih varijabli fdrd, fdwt** i **c** i privatnim kopijama stack promenljivih argc i argv, ali samo svojim kopijama, nema mešanja.
- Međutim, kernel kopira **u-area** originalnog procesa za vreme forka, pa dete nasleđuje pristup roditeljskim datotekama

fork example 1

- I roditelj i dete zovu funkciju **rdwrt** nezavisno, koja izvršava petlju, čita jedan bajt izvorne datoteke i upisuje u target datoteku.
- Funkcija se završava kada read SC dostigne kraj datoteke.

- U ovom slučaju svaki proces ima sopstvene FileDecriptors i to **fdrd** i **fdwr,**
  - ali oni ukazuju na iste ulaze u FT,
  - pa svaki proces modifikuju offset na svaki read i wite i
  - oba procesa nikada neće imati isti offset,
  - **nema prepisivanja.**

- Efekat je duplo kopiranje iste datoteke u istu target datoteku,
- ali stanje target datoteke zavisi od redosleda izvršavanja procesa.

- Na primer 2 bajta "**ab**" u originalu
  - mogu završiti u kopiji ili kao "ab", ako proces obavi **read(a)/write(a)/read(b)/write(b)**
  - ali i kao **"ba"** ako bude **read(a), read(b), write(b), write(a)**.

fork example 2

- #include <string.h>
- char string [] = "**hello world**"
- main()
- {
- int count,i;
- int **to_par[2], to_chil[2]**
- char buf[256];

- **pipe(to_par);**

- **pipe(to_chil);**

fork example 2

```
if(fork() == 0)
{
/*child process executes here*/ /*write to parent file, read from child file*/
close(0);                      /*close old standard input*/
dup(to_chil[0]);               /*dup pipe read to standard input*/
close(1);                      /*close old standard output*/
dup(to_par[1]);                /*dup pipe write to standard output*/

close(to_par[1]);              /*close unnecessary pipe descriptors*/
close(to_chil[0]);
close(to_par[0]);
close(to_chil[1]);
for (;;)
{
   if((count =read(0, buf, sizeof(buf)) == 0) exit();
   write(1, buf, count);
}
}
```

fork example 2

- /***parent process executes here***/
- {
- **close(1);** /*rearrange standard in, out */
- /*write to child file, read from own-parent file*/
- dup**(to_chil[1]);**
- **close(0);**
- dup(**to_par[0]**);

- close(to_chil[1]);
- close(to_par[0]);
- close(to_chil[0);
- close(to_par[1]);

- for (i = 0; i<15, i++)
- {
- **write(1, string, strlen(string));**
- **read(0, buf, sizeof(buf));**
- }
- }

fork example 2

- Analizirajmo sledeći program u kome dete nasleđuje file-descriptors 0 i 1 (standard input i output) od svog roditelja.
- Izvršavanje **svakog pipe SC** alocira **2** file-deskriptora u polju
  - **to_par**
  - **to_chil**
- **Proces obavlja fork SC** i time kopira svoj kontekst: svaki proces pristupa svojim privatnim podacima.
- **Proces roditelj**
  - **zatvara svoj standardni output** (file-descriptor 1) i
  - **duplicira** (sa dup) **write descriptor** to_chil[1]
  - koga je dobio od pipe(to_chil).
  - Zato što se prvi slobodan slot u roditeljskoj UFDT briše preko close,
  - kernel kopira pipe write descriptor u slot 1 u UFDT, i fd=1 pipe write descriptor za **to_chil**.

- Roditeljski proces obavlja sličnu operaciju
- da bi napravio svoj **standardni fd=0** za read pipe descriptor **to_par**[0].

fork example 2

- Slično, proces dete
  - zatvara svoj standarni ulaz (**fd=0**) i
  - duplicira pipe read descriptor za **to_chil[0]**.
  - Zato što se prvi slobodan slot u roditeljskoj UFDT briše preko close, kernel kopira pipe write descriptor u slot 1 u **UFDT**, i
  - **fd=1** pipe write descriptor za to_par[1].
- Proces dete obavlja sličan set operacija da napravi svoj standardni izlaz pipe write descriptor za to_par.
- Oba procesa zatvaraju file-deskriptore koje im je vratio pipe, a to je dobra programerska praksa.
- Kao rezultat toga, **kada roditelj upisuje** nešto na svoj **standardni izlaz**, to se upisuje na pipe **to_chil**, odnosno podaci se šalju procesu detetu, koji čita pipe kao svoj standardni ulaz.
- Kada **dete upisuje** nešto na svoj standardni izlaz, to se upisuje na pipe **to_par**, odnosno podaci se šalju procesu roditelju, koji čita pipe kao svoj standardni ulaz.
- **Procesi tako izmenjuju poruke preko ova 2 pipe.**
- **Rezultat ovog programa je invarijantnost**, **efekat je isti** bez obzira kojim se **redosledom roditelj i dete izvršavaju** iza fork SC. Na primer ako dete obavi svoj read SC pre nego što roditelj napravi write SC, dete će se uspavati sve dok roditelj ne pošalje nešto na pipe

fork example 2

- Ako roditelj obavi **write SC**, a dete to ne pročita, on će da čeka na **write**. Ovde je **poredak izvršenja fiksan**: svaki proces kompletira read i write SC i ne može da kompletira svoj **sledeći read dok drugi proces ne kompletira read i write SC.**

- **Roditeljski proces izlazi posle 15 iteracija**, a **dete zatim čita EOF** zato što **pipe nema više proces writer i završava (exit**). Ako je dete upisalo nešto na pipe, nakon što je roditelj izašao, primiće signal da je upisao na pipe na kome **nema reader procesa**.

- **Dobra je programerska praksa zatvarati suvišne file descriptore iz 3 razloga**.
  - **Prvi** je smanjuje opasnost od sistemskog limita za broj **fd**-a.
  - **Drugi je**, ako proces dete obavi exec, fd ostaju dodeljeni novom kontekstu. Zatvaranje nepotrebnih **fd** omogućava programima da se izvršavaju u čistom okruženju sa jedino **fd=0,1 i 2** otvorenim.
  - Na kraju, čitanje **pipe** vraća **EOF**, ako nema više procesa koji pišu u **pipe**. Ako proces **reader** drži pipe **write-descriptor** otvorenim, nikada se neće znati da li je writer proces zatvorio njegov kraj pipe-a.
- Prethodni primer ne radi dobro ako dete ne zatvori svoj write pipe descriptor pre ulaska u svoju petlju.

Signals

- **Signali informišu procese** da su se **dogodili asinhroni događaji.**
- Procesi mogu **jedan drugome** slati signale **sa kill SC**, ili im **kernel** može slati signale **interno**.
- U UNIX System V postoje 19 signala koji mogu da se klasifikuju:
  - **signali koji prekidaju procese**, šalju se kada **proces obavlja exit** ili kada proces pozove SC sa "**death of child**" parametrom
  - **signali vezani za exception uslove** (adresa izvan virtuelnog opsega, write u read only područje..)
  - **signali vezani za nepopravljive uslove za vreme SC**, kao na primer **nedostatak sistemskih resursa**
  - **signali izazvani neočekivanom greškom za vreme SC**, kao što je pogrešan ulazni parametar za SC, upis u pipe koji nema reader procesa
  - **signali generisani od procesa u user modu**, kao proces želi da primi **alarm** signal posle izvesnog perioda vremena
  - **signali vezani** za **terminalsku komunikaciju**
  - **signali** za **praćenje izvršavanja procesa**

Signals

- **Obrada signala** ima **više aspekta**,
  - od načina **kako se signali šalju**
  - **kako proces upravlja signalom**
  - **kako proces kontroliše svoje reakcije na signal**.

- Da bi **kernel poslao signal procesu**,
  - **kernel setuje bit** u **signal polju** u **ulazu proces tabele**,
  - **svakom tipu signala** odgovara po je **jedan bit**..
  - **ako je proces uspavan, kernel će ga probuditi prilikom slanja signala.**
  - **proces** može **zapamtiti više tipova signala ali nema informaciju** koliko **puta je primio isti signal**.
  - Na primer, ako proces **primi hangup signal** i **kill signal**, postaviće oba **bita u signal polju svog PT ulaza**, ali **nema tragova koliko je puta** primio **te signale**.

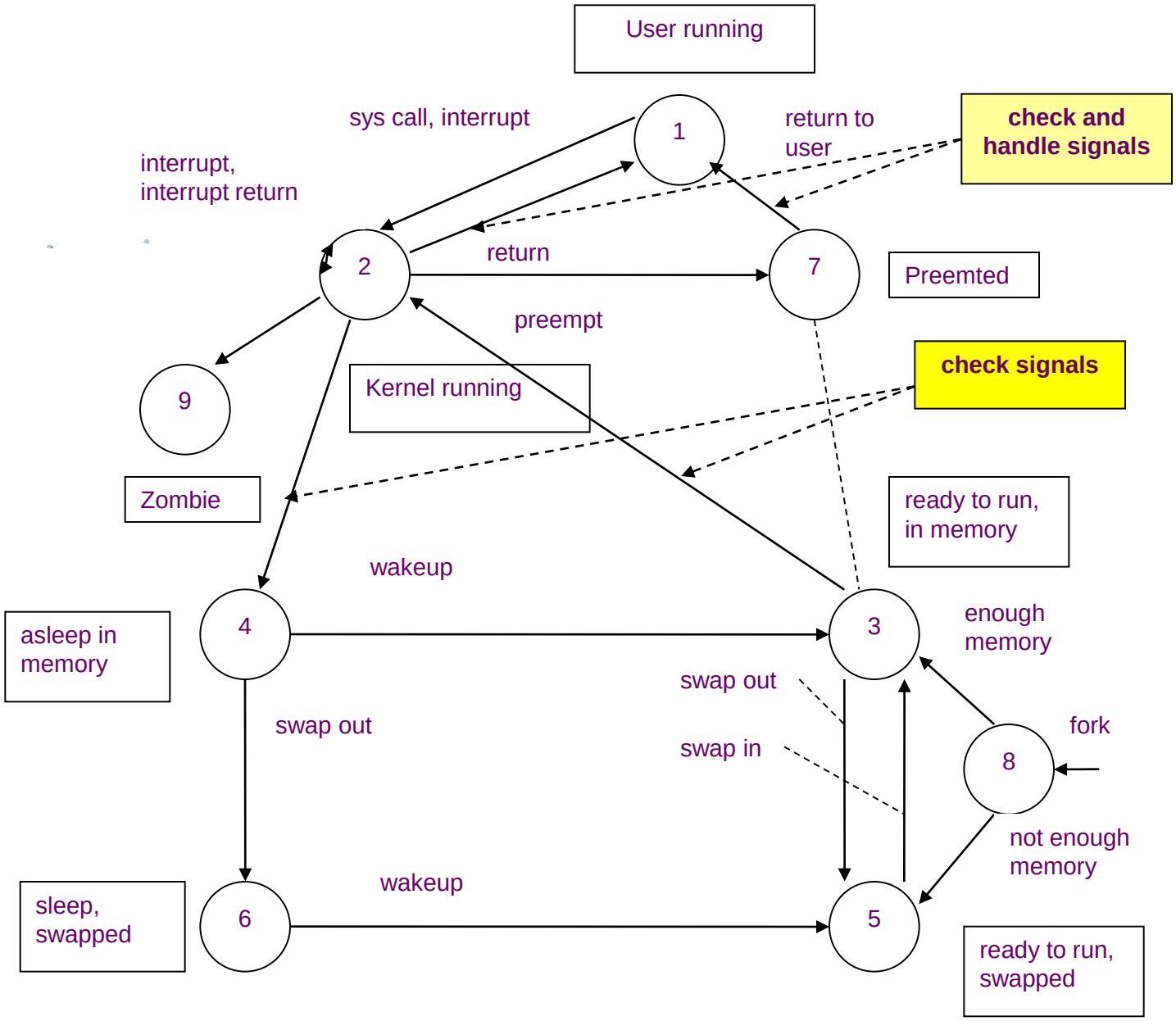
Signals

- **Kernel proverava** da li je **primljen signal**
  - **kada** je **proces spreman** da se **vrati iz 2 u 1**
  - ili **kada se proces budi**.

- **Proces upravlja signalima**
- **jedino** kada se proces vraća **iz stanja 2 u 1**,
- tako da signal nema efekta u **kernelskom modu**.

- **Ako je proces** u **user modu** i
  - **pošalje mu se signal**,
  - **slanje signala će napraviti prekid**,
  - a **signal će se obraditi po povratku iz prekida** kada je proces **spreman da napusti kernel mod**.

- **obrada signala** je **isključivo u kernelskom režimu**


User running
1
sys call, interrupt
interrupt,
interrupt return
return to
user
check and
handle signals
2
return
7
Preemted
preempt
9
Kernel running
check signals
Zombie
ready to run,
in memory
wakeup
4
asleep in
memory
3
enough
memory
swap out
swap out
swap in
fork
8
not enough
memory
wakeup
6
sleep,
swapped
5
ready to run,
swapped

issig

- Demonstriraćemo **kernelski algoritam issig**,
- **koji se određuje** da li je **proces primio signal**
- (za sada ne obrađujemo signal "**death of child**“)
- **naglasimo** da **proces može izabrati da ignoriše signale**
  - **koristeći** signal SC,
    - u kome kernel **prosto isključuje indikaciju onih signala** koje proces **želi da ignoriše,**
    - ali **prisustvo signala kill se ne ignoriše**.

issig

- **algorithm issig /* test for receipt of signal*/**
  - input: none
  - output:
    - **true**, if process received signals that it does not ignore
    - **false** otherwise

- **while(received signal field** in **PT entry not 0**)
- {
- **find** a **signal number sent to process**;
- if(signal is **death of child**)
- {
- **if(ignoring death of child signal**) **free PT entries of zombie children**;
- else if (**catching death of child signals**) return(**true**);
- }
- else if (**not ignoring signal**) return(**true**);
- **turn off signal bit** in received signal field in PT entry;
- }
- return(**false**);
- }

Handling Signals

- **Kernel upravlja signalima** u **kontekstu procesa koji ih prima** tako da proces **mora da obavlja upravljanje signalima**.

- Ima **3 slučaja za upravljanje signalima**:
  - **proces se prekida po prijemu signala**
  - **proces ignoriše signal**
  - **proces izvršava posebnu (user) funkciju po prijemu signala**

- Podrazumevana akcija je je prelazak u kernel mod, ali proces može specificirati specijalnu akciju koja se preuzima po prijemu signala, a to se postiže preko **signal SC**.

signal SC

- **Sintaksa za signal SC je:**

- **oldfunction = signal(signum, function);**

- pri čemu je
  - **signum** signal za koji se specificira akcija,
  - **function** je adresa user funkcije koja se preuzima po prijemu signala,
  - **oldfunction** povratna vrednost koju vraća funkcija koja se izvršava.

- **Proces može postaviti 0 ili 1 umesto funkcije**, pri čemu:
  - **1 ignoriše signal**
  - **0 ga prihvata** i **prebacuje proces u kernel mod**.

- **u-area** sadrži vektor (**array**) **signal-handler polja**, po **jedan handler za svaki signal koji postoji u UNIX-u**.

- Kernel čuva adresu korisničke funkcije u polju koje odgovara signalu, svaki handler ili funkcija za jedan tip signala je **nezavistan od drugih**.

- **Handlovanje signala je prikazano** u **psig** algotirmu.

algorithm psig

- **algorithm psig /* handle signals after recognizing their existance */**
  - input: none
  - output: none
- {
- **get signal number, set** in **PT entry**;
- **clear signal number, set** in **PT entry**;
- if(user had called **signal SC** to **ignore this signal**) **return;** /* done */
- **if(user specified function** to **handle the signal**) **……..signal catcher**
- {
  - **get user virtual address** of **signal catcher** stored in **u-area**;
  - /* the next statement has undesirable side-effects*/
  - **clear u area entry** that **stored address in signal catcher**;
  - **modify user level contex**t;
    - **artificially create user stack frame** to **mimic call to signal catcher function**;
  - **call** to **signal catcher function**;
    - **modify system level context**;
  - **write address** of **signal catcher** into **program counter** field of **user saved register context**;
  - **return**;
- }

algorithm psig

- **if(signal is type** that system should **dump core** image of **process**)
- {
- **create file named** "core" in **current directory**;
- **write contents** of **user level context** to file "**core**";
- }
- **invoke exit** algorithm immediately;
- }

algorithm psig

- Kada **handluje signal**, kernel određuje tip signala i ukida odgovarajući signal bit u PT ulazu koji je **setovan kada je proces primljen**.

- **#case 0: signal acceptance**
  - Ako je **signal-handling funkcija** postavljena na **0 (default),** kernel će sačuvati memorijski image proces "**core dump**" za izvesne tipove signala pre **exiting** procesa.
  - Taj **dump je pogodan za programere** i omogućava im da ispravljaju greške u svom programu.
  - Kernel **dump-uje core** za **one signale koji su poslati** kad nešto nije u redu sa procesom, na primer kad izvršava ilegalnu instrukciju ili ako prekorači svoju virtuelnu adresu.
  - Kernel **ne dumpuje one signale koje ne uključuju neku programsku grešku**. Na primer, kada korisnik pošalje **break** taster sa terminala, to znači da user želi da prekine proces i prekidni signal znači da da je sa procesom sve u redu, pa se ne pravi **core dump**.
  - Na drugoj strani, **quit signal** će uzrokovati da se generiše **core_dump** tekućeg procesa, što se postiže pritiskom control-vertical-bar na tastaturi, a jako je pogodan za procese koji su se zaglavili, pa se prekinu sa core dumpom.

- **#case 1: signal ignorance**
  - Kada proces primi signal koga je prethodno rešio da **ignoriše**,
  - **proces nastavlja egzekuciju** kao da se **signal nikada nije desio**,
  - **kernel ne resetuje polje** u **u-area** koje **definiše ignorisanje signala**,
  - **proces** će **ignorisati svaku novu pojavu signala.**

signal catching

- Ako **proces primi signal** koga je **prethodno odlučio** da hvata (**catch**), izvršiće se **user funkcija** neposredno **po povratku u user mod**, stim što će kernel pre toga da obavi **sledeće akcije**:

- kernel pristupa sačuvanom user registerskom kontekstu, nalazi PC i SP koji su sačuvani za povratak u user mode.

- **briše signal handler polje** u **u-area**, postavljajući ga u **default stanje**

- **kernel kreira novi stack frame** na **user stack-u**,
  - u koga upisuje PC i SP iz user konteksta i
  - to će se korisititi ako se zbog signala pozove user funkcija (**signal catcher function**)
- **kernel menja stanje** sačuvanog **user registerskog konteksta**:
  - PC se postavlja na **adresu signal catcher funkcije**, a
  - SP se podešava da povratak bude u pravi user kontekst

- Po **povratku u user mode**, **proces** će prvo izvršiti **signal catcher funkciju**, a potom se vraća u svoj **originalni kontekst**

signal catching example

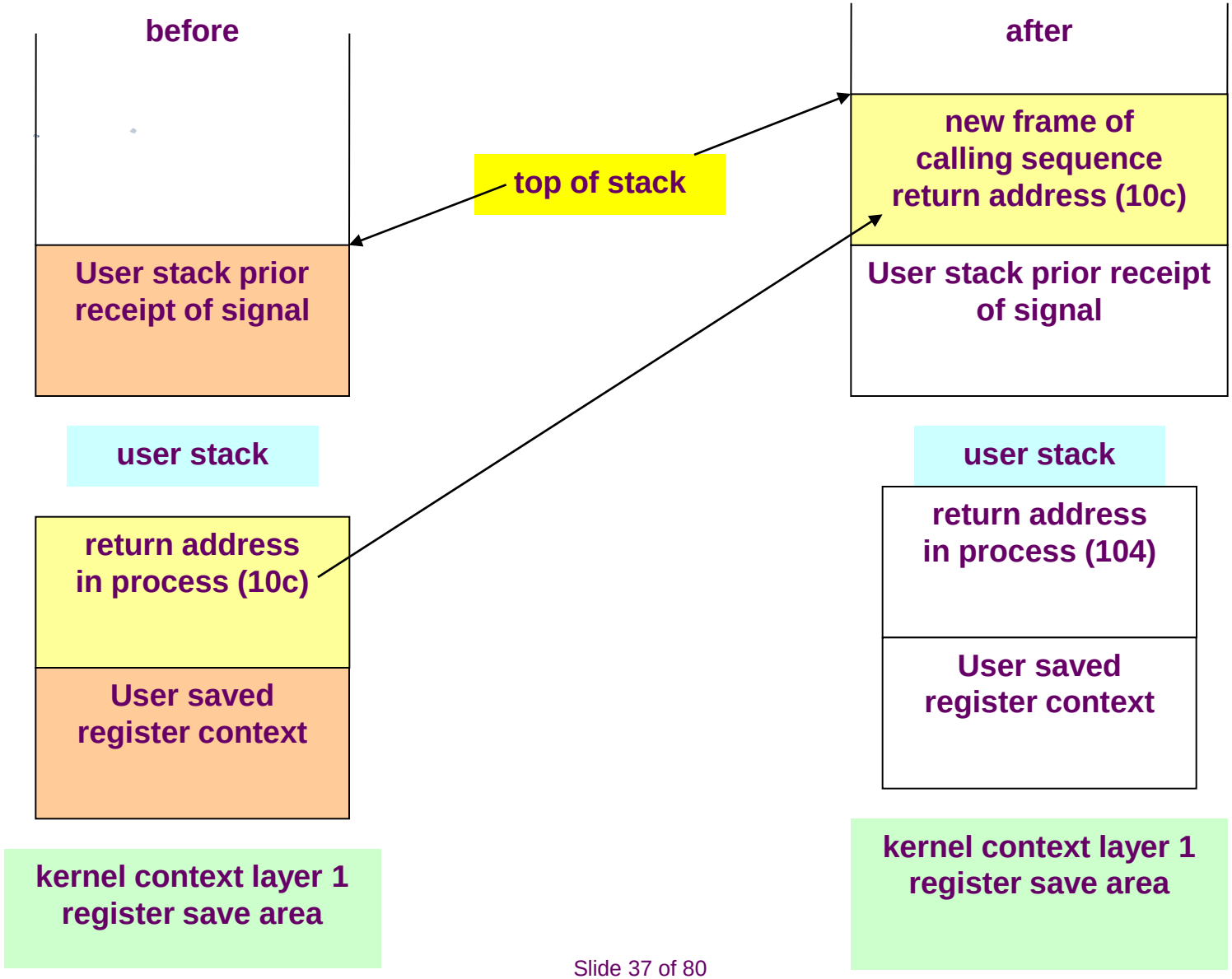
- Na primer, program koji hvata prekidni signal (**SIGINT**) i šalje sebi prekidni signal.
- #include <signal.h>
- main() {
- extern catcher();
- **signal(SIGINT, catcher);**
- **kill(0, SIGINT)**
- }

- **catcher() { }**

- a externa rutina je data u VAX assembleru
- **_catcher()**
- 104:
- 106: ret
- 107: halt

- dok je assembleslerska rutina za **kill**
- **_kill()**
- 108:
- # next line traps into kernel
- 10a: chmk $0x25 #**change to kernel mode**
- 10c: bgequ 0x6 <0x114>
- 10e: jmp 0x14(pc)
- 114: clrl r0
- 116: ret

Izgled stacka pre i posle prijema signala (SIGINT)



race condition in the catching

- **Više anomalija** postoje u opisanim algoritmima za tretiranje signala.

- Prva i **najznačajnija** je kada proces **obrađuje signal**,
- što kernel pre povratka u user mod **briše polje u u-area** koje sadrži **adresu** za **signal handler funkciju**,
- tako da za **nove signale** proces **mora ponovo pozvati signal SC**,
- a to nije zgodno jer može doći do **race condition**,
- zato što druga **instanca signala** može doći pre nego što proces pozove **signal SC.**

- Sledeći program ilustruje **race condition**.

race condition in the catching

```
#include <signal.h>
sigcatcher()
{
printf("PID %d caught one", getpid()); /* print process id*/
signal(SIGINT, sigcatcher); /*again*/
}

main()
{
int ppid;
signal(SIGINT, sigcatcher)

if(fork() ==0)
{
/* child*/ /* give enough time for both process to set up*/
sleep(5)                    /*lib function to delay 5 secs*/
ppid = getppid();           /* get parent id*/
for(;;)   if (kill(ppid, SIGINT) == -1)  exit();
}

/*parent*/ /* lower priority, greater chance of exhibiting race*/
nice(10);
for(;;) ;
}
```

race condition in the catching

- Proces zove signal SC da hvata prekidni signal, i izvrši funkciju sigcatcher()..
- Proces kreira dete proces, poziva **nice SC** da smanji svoj prioritet, i odlazi u beskonačnu petlju (for(;;)).
- Proces **dete spava 5 secundi**, dajući šansu roditelju da obavi **nice**, a potom ide u petlju u kojoj roditelju šalje prekidni signal za vreme svake iteracije.
- Ako se kill SC završi sa greškom, na primer zato što roditelj više ne postoji, proces dete obavlja **exit**.
- Ideja je da roditeljski proces treba da pozove **signal catcher** svaki put kad dobije **prekidni signal**, koji će štampati poruku na ekranu i sa signal SC ponovo postaviti signal catcher i tako bi roditelj po toj zamisli nastavio da izvršava beskonačnu petlju.

race condition in the catching

- Moguća je sledeća sekvenca događaja:
  - 1. proces-dete pošalje prekidni signal roditelju
  - 2. proces-roditelj hvata signal i poziva **signal catcher**, ali kernel preemptuje roditelja i obavi CSw pre nego što roditelj obavi **signal SC** ponovo
  - 3. proces dete dobija CPU i pošalje novi prekidni signal
  - 4. proces roditelj nema više postavljen signal catcher, pa kad dobije CPU obaviće exit

- Ovo će se veoma verovatno dogoditi, jer je proces roditelj sa nice smanjio svoj prioritet i proces dete će se birati mnogo češće (podsetimo da oba izvršavaju beskonačnu petlju).

- Ukupan efekat je da će oba procesa da završe, ali ne može se odrediti kada će se to dogoditi.

death of child

- Na kraju, **kernel ne tretira** "**death of child**" signale **kao sve druge signale**.
- Kada proces **prepozna "death of child"**,
  - on postavlja notifikaciju signala u signal polju PT ulaza u **default stanje**,
  - i **ponaša** se da **ništa od signala nije primio**.
  - **Efekat ovog signala je buđenje uspavanog procesa**.
- Ako **proces hvata ovaj signal**, on *poziva signal handler kao i za sve druge signale.*
- **Operacija** koju kernel radi **ako proces ignoriše ovaj signal** diskutovaćemo u sekciji 7.4.

Process Groups

- Mada se procesi na UNIX sistemu identifikuju po jedinstvenom PID, poželjno je da sistem **može da grupiše procese** tako da se **identifikuju po grupi**.
-
- Na primer proces **shell je roditelj svim user procesima** sa **tog terminala** i oni se mogu grupisati da **primaju istovremeno signale**. Kernel koristi grupni ID da identifikuju grupe povezanih procesa koji treba da dobiju **zajednički signal**. Grupni ID se čuva u PT.
- SC koji služi da **inicijalizuje PGID** i **setuje** na **istu vrednost kao** i **PID**.
- Sintaksa za **setgrp SC** je:
- **grp = setpgrp();**
- gde je **grp** novi **PGID**.
- **Proces dete nasleđuje** od **svog roditelja PGID** preko **fork SC.**

Slanje signala iz procesa

- Procesi koriste kill SC da bi slali signale.
- **Sintaksa za kill je**
- **kill(pid, signum)**
- gde je
  - **pid** skup procesa koji primaju signal, a
  - **signum** je broj signala koji se šalje.
- Sledeća lista prikazuje korespodenciju između vrednosti pid-a i skupa procesa
  - ako je **pid pozitivan broj**, **kernel šalje signal procesu** sa **PID=pid**
  - ako je **pid=0,** kernel šalje **signal svim procesima** koji su sa **procesom koji šalje signal u istoj grupi**
  - ako je **pid=-1**, kernel šalje signal svim procesima čiji je **realni UID** jednak **efektivnom UID procesa koji šalje signal**.
    - Ako proces koji šalje ima **efektivni UID=root**, kernel šalje **signal svim procesima osim za PID=0 i PID=1**
  - ako je pid **negativan broj koji nije -1**, kernel **šalje signale svim procesima** u grupi čiji je **PGID jednak apsolutnoj vrednosti pid-a**.
- U svim slučajevima, ako proces koji šalje signal **nema efektivnu vrednost UID=root**, ako **pošalje signal procesima čija realna ili efektivna vrednost ne odgovara primajućim procesima**, **kill će otkazati**

primer za setgrp

```
#include <signal.h>
main()
{
    register int i;
setpgrp();
for(i=0; i<10; i++)
{
if(fork() == 0) {

/* child */
if (i & 1) setpgrp();
printf("pid %d pgrp =%d \n", getpid(), getgrp()); /* print proc id*/
pause();                                          /* SC to suspend execution*/
}/*if*/
}/*for*/

/*parent*/
kill(0, SIGINT);
}
```

primer za setgrp

- U ovom programu
- proces **setuje svoj PGID** i **kreira 10 dece**,
- od kojih svako dete ima ima isti **PGID** kao roditelj,
- ali procesi za vreme **neparnih iteracija**, setuju svoje sopstvene **PGID**.

- **SC getpid** i **getgrp** vraćaju **PID i PGID** tekućeg procesa,
- dok **pause SC suspenduje izvršavanje procesa dok proces ne primi signal**.

- Na kraju, roditelj izvršava **kill** i **šalje SIGINT svim procesima** iz **svoje grupe**,
- tako
  - **5 parnih procesa** primiti taj signal, i obaviti exit
  - **5 neparnih će ostati u paused stanju**

Process Termination

- **Procesi** na **UNIX-u završavaju svoje aktivnosti** preko **exit SC**. Proces koji završava aktivnosti ulazi u **zombie stanje**, otpuštajući sve svoje resurse.

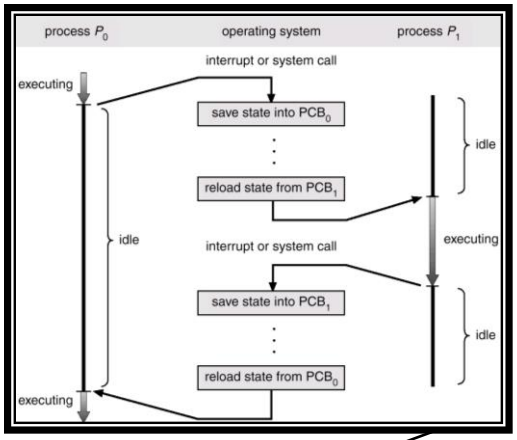
- Sintaksa je
- **exit(status)**
- gde je **status vrednost** koja se vraća **procesu roditelju**.

- Procesi mogu pozvati **exit explicitno ili implicitno** na kraju programa, rutina koja je startovala program će na **kraju po povratku iz main programa pozvati exit SC**.
- Kernel može pozvati **exit SC** explicitno
  - ako proces primi signal **koji ne hvata**
  - i tada će **status programa da bude broj signala.**

- OS ne odlučuje **koliko će proces trajati** (process **swaper** i **init** postoje **sve dok je aktivan UNIX**, a na drugoj strani neki procesi budu kratko pa isčeznu kao **getty**).

algorithm exit

- **algorithm exit /* */**
- input: **return code for parrent process**
- ouput: none
- {
- **ignore all signals**;
- **if(process group leader** with associated control terminal)
- { send hangup signal to all member of process group;
- reset process group for all members to 0; }
- **close all open files** (algorithm close(internal version))
- **release current directory** (algorithm iput);
- **release current (changed) root**, if exist (algorithm **iput**);
- **free regions**, memory associated with process (algorithm **freereg**);
- write accounting record;
- **make process state zombie**;
- assign **parent PID** of all child processes **to be init process**;
- if any children **were zombie**, send **death of child signal to init**;
- **send death of child signal** to **parent process**;
- **context switch**;
- }



algorithm exit

- **Kernel prvo blokira obradu signala** za proces koji završava, zato što za njega više neće imati smisla. Ako **proces vođa grupe procesa** na pridruženom terminalu, tada će se prekinuti svi procesi u grupi. Na primer ako neko shell procesu prosledi **<ctrl-d>** karakter, **shell je vođa i svim procesi** sa njim u grupi će se okončati.
- Kernel **takođe PGID svih procesa iz te grupe na 0**, da ne bi došlo **do haosa** ako **neki novi proces dobije PID** od vođe koji je upravo uradio **exit**, pa on postaje **novi vođa** za one koji se **nisu završili**.
- Kernel zatim analizira **otvorene file descriptore** koje zatvara pomoću internog close i oslobađa inode za tekući direktorijum i za promenjeni root preko iput algoritma.
- Zatim, **kernel oslobađa svu korisničku memoriju**, oslobađajući regione preko **detachreg** i menja stanje procesa u **stanje zombie**. On čuva exit status code i akomulira i upisuje statističke informacije o vremenima u user modu, kernelskom modu itd, CPU usage, memory usage.
- Na kraju, kernel izbacuje proces iz stabla aktivnih procesa, a **proces init će da usvoji svu procesovu živu decu-procese**. Ako je bilo koje dete zombie, **proces koji završava** će **poslati procesu ini**t "**death of child**" signal tako init izbaci sve zombie iz proces tabele.
- **Proces koji završava takođe** šalje **svom roditelju** "**death of child**" signal.

exit SC

- Po tipičnom UNIX scenariju, roditeljski proces izvršava **wait SC** za sinhronizaciju sa decom koja završavaju rad. Proces koji završava (**now-zombie**) radi još **CSw**, tako da kernel može da **izabere drugi proces za rad**, jer kernel nikada neće izabrati proces u zombie stanju.
- U sledećem primeru,
  - proces kreira proces dete, koje štampa svoj PID i obavlja pause SC, **blokirajući sebe dok ne primi neki signal**.
  - Roditelj proces štampa dečiji PID i završava vraćajući detetov PID kao **exit status**. Ako exit call nije prisutan, startup rutina će pozvati exit kada se proces vrati za main.
  - Proces-dete će živeti sve dok ne primi signal, bez obzira da li je **proces-roditelj živ ili ne.**
- **main()** {
- int child;
- **if ((child = fork()) = = 0)**
- { **/* child process*/**
- printf("child PID %d \n", getpid());
- pause(); }

- **/*parrent process*/**
- printf("child PID %d \n", child); **exit(child);**
- }

awaiting process termination

- **Proces može sinhronizovati svoje izvršavanje** sa **završetkom procesa-deteta** preko **wait SC**.
- Sintaksa za SC je:
- **pid = wait(stat_addr)**
- gde je
- **pid** proces **ID deteta** koje **treba da postane zombie**,
- **stat_addr** je **integer adresa** u **korisničkom prostoru** koja sadrži **izlazni status deteta..**
- Na slici je prikazan algoritam za **wait SC**.

awaiting process termination

- **algorithm wait**
  - **input: address** of **variable** to **store status** of **exiting process**
  - **output: child ID, child exit code**
- {
- **if(waiting process has no child processes) return(error);**
- for (;;)
- {
- **if(waiting process** has **zombie child**)
- {
  - **pick arbitrary zombie child**;
  - **add child CPU usage** to **parent**;
  - **free child PT entry;**
  - **return(child ID, child exit code);**
- }
- **if(process** has **no child processes**) **return error;**
- **sleep** at **interruptible priority** (**event child process exits**);
- **}/*for*/**
- }

wait description

- Kernel traži **zombie decu** i ako **nema nijednog deteta**, vraća se **greška**.
- Ako se nađe **zombie dete**, ekstrakuju se **PID takvog deteta** i **parametri smešteni** u završetak **exit SC** takvog deteta, a to mogu da budu **različite vrednosti** koje opisuju šta se **dogodilo** sa **detetom**.
- Kernel uzima **akomulisano vreme** koje je dete radilo u user i kernelskom modu i dodaje u u-area procesa roditelja, a na kraju otpušta slot u PT koji je zauzimalo dete.
- Ako proces obavlja **wait SC** i pri tome ima decu procese ali nisu u zombie stanju (nijedno dete), proces će se uspavati sve dok se ne pojavi signal. Kernel nema poseban SC za buđenje procesa koji **spava u wait SC**, takav **proces se jedino budi na prijem signala**.
- Za bilo koji signal osim "**death of child**" proces **će reagovati** na **sledeći način**:
- u podrazumevanom slučaju, proces će se probuditi u **svom spavanju u wait-u** i **sleep** će tada **pozvati algoritam isig** da **proveri ima li signala**.
- Ako **isig** detektuje signal "**death of child**" tada vraća "**false**". Kernel ne obavlja **longjmp** iz **sleep-a** ali se vraća u **wait SC**, gde će pronaći barem jedno **zombie dete** i vratiti se iz **wait SC**.
  - ako proces **hvata signal** "**death of child**", kernel poziva **user-handler rutinu** kao i za druge signale
  - **ako proces ignoriše** signal "**death of child**", kernel ulazi u **wait petlju**, oslobađa slotove za zombie decu i traži odnosno **čeka novu zombie decu**.

sigcld demo

- Uzmimo primer kada se sledeći program pozove sa ili bez argumenata.
- #include <signal.h>
- main(argc, argv)
  - int argc;
  - char *argv[];
- {
- int i, ret_val, ret_code;
- **if(argc > 1) signal(SIGCLD, SIG_IGN); /* ignore death of children*/**
- for(i=0; i<15; i++)
- **if(fork() == 0) /* child processs*/**
- {
- printf("child proc %x \n", getpid()); /* print proc id*/
- **exit(i);** /* SC to suspend execution*/
- }
- **ret_val = wait (&ret_code);**
- printf("wait ret_val %x ret_code %x\n", **ret_val**, **ret_code**);
- }

sigcld demo

- Uzmimo slučaj da user pozove program bez argumenata,
- (argc = 1, program name, only).

- **Proces roditelj kreira 15 procesa dece**
- koja eventualno završavaju sa exit kodom(i) a to je vrednost brojača kad je proces kreiran.

- Kernel **izvršava wait SC** za proces roditelj,
  - pronalazi zombi-dete proces koje je završilo aktivnost,
  - uzima njegov PID i exit status,
  - ali se ne zna koje će to dete da bude.

sigcld demo

- Ako se program pozove sa argumentom (**argc >1**)
- roditeljski proces **signalizira da ignoriše signal** "**death of child**".

- Pretpostavimo da roditelj spava, a dete koje obavlja exit SC, koji postavlja signal **"death of child"** koga **roditelj ignoriše**, on praktično uklanja ulaz u PT za to zombie dete, ali nastavlja wait kao da se signal nije dogodio.
- Svaki put se obavi ova procedura za svako dete koje završi aktivnosti, i onda se dođe do procesa roditelja koji nema više dece, kada se **wait SC** završava sa **exit kodom =-1.**

- Vidimo da
- u **prvom slučaju roditeljski** proces **čeka bilo koje dete da završi**, a
- u **drugom slučaju** roditelj **čeka svu decu da završe**.

- **Stare verzije** implementiraju **exit i wait SC bez signala** "death of child",
- a umesto tog signala exit SC budi roditeljski proces.
  - ako je roditeljski proces zaspao **u wait SC**, sve je u redu,
  - **exit SC** deteta će ga probuditi,
  - ali ako roditelj nije uspavan u wait SC buđenje nema efekat,
  - ali će roditelj sledeći put kad uđe u wait SC da nađe zombie decu.

Invoking other programs-exec

- **SC exec poziva drugi program**, **prepisujući** memorijski prostor procesa sa **kopijom izvršne datoteke**. Sadržina **user-level konteksta** koji je postojao pre exec SC **nije više dostupna**, osim **exec parametara** koje kernel kopira iz starog adresnog prostora u novi adresni prostor.
- Sintaksa za exec SC je:
- **execve(filename, argv, envp)**
- gde je
  - **filename** ime **izvršne datoteke** koja se poziva,
  - **argv** je ukazivač na polje ukazivača koji su **parametri za izvršnu datoteku**,
  - **envp** je **ukazivač na okruženje** (enviroment) izvršnog programa.
- Ima više funkcija biblioteke koje pozivaju exec SC kao što se **execl, execv, execle** itd.
- Kada program koristi komandne linijske parametre kao
- **main(argc, argv)**
- polje argv je kopija argv parametera exec SC. Što se okoline tiče ona se sastoji od karakter nizova tipa "name=value" i koristi korisne informacije za program kao što je user HOME ili PATH. Procesi mogu pristupati njihovoj okolini preko globalne varijable **environ**, koju inicijalizuje C startrup rutina.

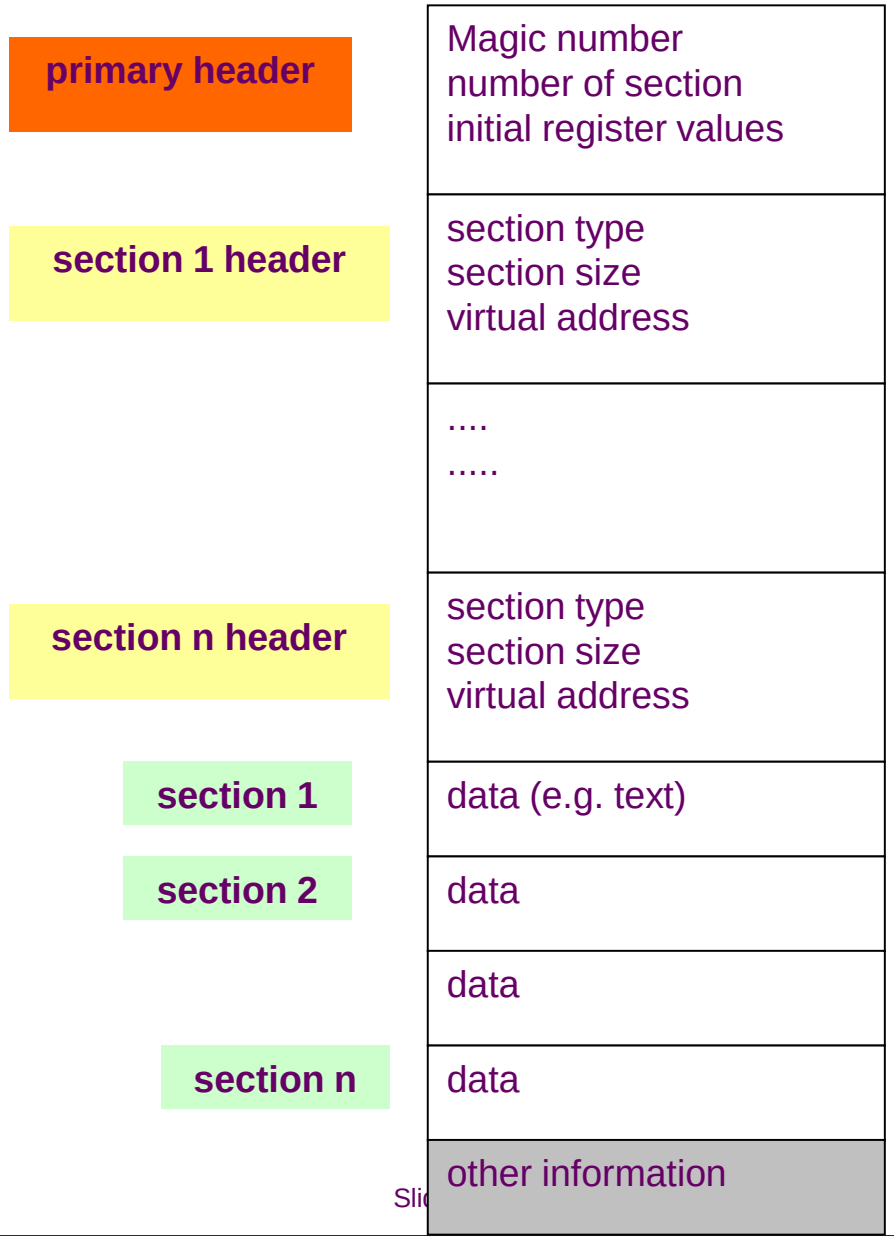
algorithm exec

- **algorithm exec**
  - input: (1) file name (2) parameter list (3) enviroment variables list
  - output: none
- {
- **get file inode** (algorithm **namei**)
- verify file executable, user has permission to execute;
- read file headers, check that it is a load module;
- copy exec parameters from old address space to system space;
- for (**every region attached to process**) **detach all old region (algorithm detach);**
- for (**every region specified in load module**)
- {
- **allocate** new regions (algorithm **allocreg**);
- **attach** the region (algorithm **attachreg**);
- **load** region into memory if appropriate (algorithm **loadreg**);
- }
- **copy exec** parameters into **new user stack region**;
- **special processing** for **setuid programs**, **tracing**;
- **initialize user register save area** for **return** to **user mode**;
- **release** inode of file (algorithm **iput**);
- }

format of executable file

- SC exec prvo pristupa datoteci, tako što joj nađe inode preko algoritma **namei** i proverava da li je to **obična datoteka sa x pravima** i da li user koji je vlasnik procesa **ima pravo** da **izvršava program.**
- Kernel zatim **čita header** datoteke da bi odredio **layout izvršne datoteke**.
- Na slici je prikazan **logički format izvršne datoteke**
  - **koju generiše assembler** ili loader i
  - koja se **sastoji od 4 dela**:
- **1. primarni header** opisuje
  - koliko ima sekcija u datoteci,
  - početnu adresu za izvršenje procesa i
  - magični broj koji daje tip izvršne datoteke
- **2. headeri sekcija** opisuju svaku sekciju u datoteci, opisujući section size, virtuelnu adresu sekcije koju sekcija treba da zaizme u vreme izvršavanja i druge informacije
- **3. text sekcije:** koje sadrže podatke kao što je **text** koje se inicijalno pune u adresni prostor procesa
- **4. data sekcije:** razne sekcije sa **tipičnim data osobinama**: simboličke **tabele i druge vrste podataka**

format of executable file



exec description

- Mnogi formati su se menjali, ali sve izvršne datoteke sadrže **primarni header sa magičnim brojevima**, pri čemu **magični brojevi** mogu da definišu neke **osobine izvršne datoteke vezane za procesor**, a magični brojevi su veoma važni u **paging sistemu**.

- Dakle, **kernel prvo pristupa inodu izvršne datoteke** i proveri da li može da je izvršava. Sada će se prepisati **user-level kontekst procesa**, prepisujući datoteku, pa prvo što treba uraditi je kopirati **parametre novog programa** koji su starom kontekstu. Kernel ih prvo kopira u privremeni bafer sve dok ne attach-uje regione za user kontekst.

- **Parametri** za **exec SC** su **adrese** za **parametre**, pa kernel **prvo kopira adrese** parametara a **potom i same parametre**. Kernel može izabrati **više lokacija** za privremeni **smeštaj parametara**, kao što je **kernel stack**, nealocirane stranice, swap.

- **Naprostiji** način kopiranja parametara je novi **user-level kontekst** u **kernelskom stacku**, ali parametri mogu **biti dugački**, a **kernel stack ograničen**. Zato se prelazi na druge metode, kao što su **alociranje stranica u memoriji**, a **swap se izbegava jer je spor**.

exec description

- Posle kopiranja exec parametara, kernel detach-uje stare regione procesa preko **detachreg** algoritma.
  - **Text regioni** imaju **poseban tretman**. U ovom trenutku proces nema user-level kontekst, tako da ako mu se dogodi greška on se prekida.
  - **Kernel alocira i attachuje regione za text i data**, a onda se **oni pune iz sekcija izvršne datoteke** (allocreg, attachreg, loadreg).
  - **Data region procesa se deli na 2 dela**, data koji se **inicijalizuju u vreme prevođenja** i data koji **se ne inicijalizju u vreme prevođenja** (bss).
  - Prvo se **alociraju** i **attach-uju inicijalizovani podaci**, a kernel potom obavi uvećanje data regiona sa **growreg za bss**, koga **inicijalizuje sa 0**.
  - Na kraju se **alocira stack region**, attachu-uje se procesu i **alocira se memorija za exec parametre koji se kopiraju u user stack region.**

- Kernel briše adrese korsiničkih **signal catcher-a** iz u-area, zato što ono više nemaju značenje za **novi user kontekst**.
  - Potom **setuje registre** za novi user-level kontekst od kojih su najznačajniji **PC i SP.** PC se puni iz headera datoteke.
  - Kernel uzima specijalni akciju za **setuid programe** i za proces tracing, a to ćemo obraditi kasnije.
  - **Na kraju** se **otpušta inode izvršne datoteke algoritam iput**.
  - Praktično **exec SC u odnosu na datoteku**, obavi sve **što i open SC**, sem što nema ulaza u FT.
  - Kada se vrati iz exec SC, proces počinje da izvršava **kod novog programa**.
  - Mada je proces **potpuno promenjen po kodu**, njegov **PID i mesto u proces stablu ostaje isto**, **jedino mu se promenio user-level kontekst**.

exec example

- Na primer, sledeći program kreira proces dete koje obavi exec SC.
- **main()**
- **{**
- **int status;**
- **if(fork() == 0) execl("/bin/date", "date", 0);**
- **wait(&status);**
- **}**
- Kada se obavi fork i roditelj i dete izvršavaju isti kod.
- Pre nego što dete obavi exec SC, njegov text se sastoji od gornjeg programa, data region sadrži nizove "/bin/date", i "date", a stack sadrži stack okvire za realizaciju exec SC.
- Kernel pronalazi /bin/date uverava se da je to izvršna datoteka koju user može da obavi.
- Kernel **kopira parametre exec SC** ("/bin/date", "date") **negde privremeno**, a potom **oslobodi sve regione** koji su pripadali procesu, **a potom alocira nove**, u novi text region kopira text sekciju, a kopira data sekcije u data region. U alocirani stack region kopira ulazne parametre.
- Posle exec SC, proces **dete ne izvšava više gornji program**, dete se više nikada neće vratiti na **wait**, njegov kod je kod programa **/bin/date**. Kada dete obavi svoje, roditelj dobija exit status i **završava wait.**

xalloc

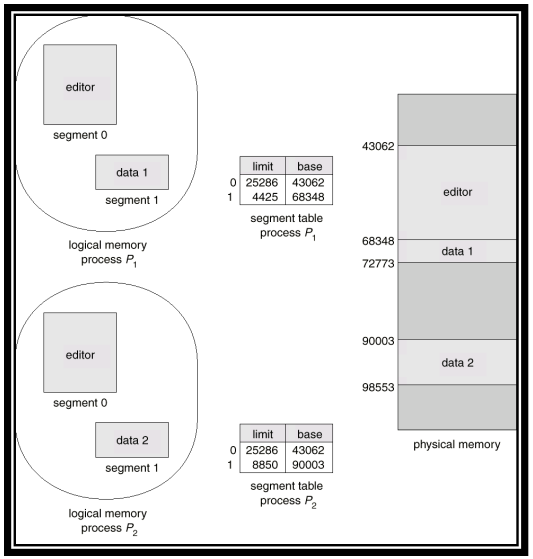
- **Text i data su po pravilu razdvojeni u izvršnim datoteka**
- što im donosi **2 velike prednosti**:
  - **zaštitu**
  - **deljenje**

- Na primer **ako su razdvojeni,**
- **text** je **nepromenljiv** i
- može se **deliti između više procesa**.

- Kako je **UNIX uveo strogo razdvajanje text i data sekcija**,
- **uveden je novi algoritam xalloc** za **dodelu text regiona,**
- pri čemu **kernel mora da proveri u magičnim brojevima**
  - da li je **text sekcija deljiva i ako jeste**,
  - da li je **već u memoriji od strane nekog drugog procesa**

xalloc

- **algorithm xalloc /* allocate and initialize text region*/**
  - input: **inode of executable file**
  - output: none {
- if(executable file **does not have separate text region**) **return**;
- **if(text region associated** with **text of inode**)
- {
- **/* text region exist .....attach to it*/**
- lock region;
- **while(contents of region not ready yet**)
- { /*manipulation of refernce count prevents total removal of region*/
- **increment region RC**;
- unlock region;
- sleep(event contents of region ready)
- lock region;
- **decrement region RC; }/*while*/**
- **attach region to process (algorithm attachreg);**
- unlock region;
- return; }/*if*/



xalloc

- **/* no such text region---create one*/**
- **allocate text region** (algorithm **allocreg**); /* region locked*/
- if(inode mode **has stickey bi**t) turn on **region stickey flag**;
- **attach region** to **virtual address indicated** by **inode file header**
  - (algorithm **attachreg);**
- **if(file specially formated** for **paging system**) /* lesson 9*/
- **else /*not formated for paging system*/**
  - **read file text** into **region** (algorithm **loadreg);**
  - **change region protection** in per process region table **to read-only**;
  - **unlock region**;
- }

xalloc description

- **Kernel u xalloc algoritmu** pretražuje **aktivnu region** listu za **text region** izvršne datoteke, tražeći ima li neku **region** sa **odgovarajućim inodom**, istim kao za traženu datoteku. **Ako ne postoji**, kernel mora alocirati novi region (**allocreg**), attach-uje ga procesu (**attachreg**), napuni ga u memoriju (**loadreg**) i promeni ga na read-only.
- **Ako ga nađe u listi aktivnih regiona imamo 2 situacije**:
  - mora se **uveriti** da je **region u memoriji**
  - ili će da **spava ako je počelo punjenje**.
- Kernel unlock-uje region kada završi **xalloc**, i decrementira RC kasnije, tek kada obavi detach za vreme exit-a.
- Podsetimo, da prilikom alokacije regiona, inkrementira se RC(region) i za RC(inode), tako da je RC=1 minimum, **niko ne može da obriše datoteku** bez obzira na **unlink SC**, datoteka će ostati tu.
- Prepostavimo da **/bin/date** ima **razdvojene text i data sekcije**. **Prvi put** kada proces izvrši /bin/date, kernel napravi RT ulaz za text i postavi inode RC (in core) na 1, kada se exec kompletira. Kada /bin/date završi, detachreg i freereg dekrementiraju inode RC na 0. **Ako se inode** RC ne bi inkrementirao kada se prvi put izvršava exec, exec drugog programa bi mogao da namesti drugi program u **in-core inode tabeli**, pa bi exec mogao da izvrši pogrešan program. **Zato je inode RC** za **shared text minumum 1**.

sticky bit

- Mogućnost da se dele text regioni može da se **ubrza** preko **stickey** bita, koji **se setuje** preko **chmod** komande.
- **Stickey bit izaziva da kernel ne otpušta memoriju lociranu** za **stickey bit text region**, čak i ako **RC za region padne na nulu**.
- Kernel će ostaviti t-text region u memoriji sa inode **RC=1**, čak i kad nema više aktivnih procesa za njega. Zato će svako **novi exec za taj stickey-text** da ima **mali startup time**, jer je **već u memoriji**, pa čak i da je **swapovan** taj **text region**, brže će se napuniti iz **swap-a** nego iz **datoteke**.

- Kernel **će ukloniti ulaze za stickey-bit text regione** u sledećim slučajevima:
  - ako proces otvori **datoteku za write**, čime se menja text
  - ako proces **promeni permision mode (skine stickey bit)**
  - **ako proces obriše file (unlink), a nema ni jednog procesa** koji radi sa **datotekom**
  - **ako proces umount-uje FS**
  - **ako nestane mesta na swapu**

- U **prva 2 slučaja menja se sadržaj datoteke**, odnosno ukida se **stickey bit**. U ostalim slučajevima, **ne menja se datoteka**, ali **nema ni jednog procesa koji je koristi**, a više i **nema uslova da je neko koristi**.

Changing the size of process

- **Proces može povećati ili smanjiti veličinu svog data regiona**
- korišćenjem **brk SC**,
- čija je sintaksa:
- **brk(endds)**
- gde je
- **endds** vrednost **najviše virtuelne adrese data regiona** procesa (break value).

- Postoji još **jedna alternativa**
- **oldends=sbrk(increment)**
- gde
  - **increment** vrednost povećava tekuću **break vrednost** za **zadati broj bajtova**,
  - **oldendds** je **break vrednost prethodnog brk SC**.

Changing the size of process

- SC sbrk je C library rutina koja poziva brk.
- Ako se **region uvećava**, **novi virtuelni prostor je takođe kontinulan** sa **starim prostorom**, ondosno **stari adresni prostor se proširuje**.
- **Kernel mora da proveri** da li **zahtev legalan**, da li to može da se realizuje i da se ne preklapa sa drugim regionima.
- **Ako sve provere prođu**, kernel poziva **growreg** algoritam da **alocira aux** memoriju ako što su **page tabele** za novi region i uvećava polje za veličinu procesa u PT.
- Na swapping sistemima, takođe se pokušava da se **alocira novi prostor** u memoriji i **nova memorija se puni nulama**. Ako nema dovoljno mesta u memoriji neki id procesa mora na swap (swap out).
- **Ako se brk** koristi za **smanjenje regiona**, kernel će **osloboditi memoriju** i **ažurirati PT** i **page tabele**.

algortithm brk

- **algorithm brk**
- input: new break value
- output: old break value
- {
- **lock process data region**;
- **if(region size increasing**)
- **if(new region** size is **illegal**)
- {
- unlock data region;
- return(**error**);
- }
- **change region size** (algorithm **growreg**);
- **zero out** in **new space**;
- **unlock process data region**;
- }
-

algortithm brk example

- Slika prikazuje program koji koristi **brk** SC.
- #include <signal.h>
- char *cp
- int callno;
- main()
- {
- char *sbrk();
- **extern catcher();**
- **signal(SIGSEGV, catcher);**
- **cp = sbrk((0));**
- **print("original brk value %u",cp)**
- for(;;)
- ***cp++=1**
- }

algortithm brk example

- **catcher(signo)**
- int signo;
- {
- callno++;
- **printf (caught sig %d %dth call at addr %u", signo, callno, cp)**
- **sbrk(256)**
- **signal(SIGSEGV, catcher**);
- }

algortithm brk example

- Pošto podesi **catch za segmentation violation signale**,
- proces poziva **sbrk(0)**, pa prikazuje početnu **break vrednost**.

- Tada se obavlja petlja inrementira se **karakter pointer cp i upiše se vrednosu** njega, sve dok ne **udari u kraj data regiona**,
- kada se generiše signal **SIGSEGV**, a njegov **catcher handler** pozove **sbrk(256)** i **poveća region za 256 bajtova**.
- Petlja se nastavlja dalje.

- **Kernel automatski proširuje user stack region** kada se **dogodi overflow**, **koristeći sličan algoritam** kao **brk**.
- **Proces originalno sadrži dovoljan user stack** da čuva **exec parametre**, ali u **toku egzekucije može doći do prekoračenja**, što izaziva **memory fault** a to je prekid, u kome kernel vidi da je **Memory Fault napravio stack overflow**, **komrarira se faulted SP** i **tekuća veličina stack regiona** pa se **poveća preko brk SC**

The shell

- **Shell je kompleksniji** nego što je ovde opisano, ali dosta dobro je opisana **glavna shell petlja** i u njoj je demonstrirana **asinhrono izvršavanje**, **redirekcija izlaza i pipe**. Glavna **shell petlja** je prikazana na sledećoj slici.
- **Shell čita komandnu liniju** sa **standardnog ulaza** koju zatim interpretira po fiksnim pravilima. **Standardni ulazni i izlazni file-deskriptor** za **shell** je obično **terminal** na kome se **user loguje**.
- Ako shell prepozna ulazni string kao **neku unutrašnju komandu** (cd, for, while..), **shell izvršava komandu interno bez kreiranja novog procesa**, u protivnom **ulazni niz je ime egzekuabilne datoteke**.
- Naprostije komandne linije sadrže ime programa i neke parametre kao na primer:
- **who**
- **grep –n include *.c**
- **ls –l**
- **Shell forkuje i kreira proces dete** koje će izvršiti **program zadat** u **komandnoj liniji**.
- **Roditeljski proces, shell, čeka da dete završi, a onda se ponovo vrti u petlji čekajući novu komandu.**

The shell

- **Da bi se proces stratovao asinhrono** (in background) potrebno je upotrebiti **&** kao u primeru
- **nroff –mm bigdocument &**
- **pri čemu shell setuje svoju internu varijablu amper** koja se postavlja **nakon analize komandne linije**, a **ako nađe &,** shell ne izvršava **wait SC** nego se opet vraća na **loop za sledeću komandu**.

- Kao što vidimo sa slike, proces dete u shell-u ima kopiju komandne linije posle fork SC-a.

- **Redirekcija standarne izlazne datoteke** se obavlja kao u primeru
- **nroff –mm bigdocument > output**
- kada proces **dete kreira izlaznu datoteku** specificiranu u komandnoj liniji, pri čemu **ako creat otkaže** (zbog nedostatka prava pristupa), proces **dete će realizovati exit neposredno**.
- Ali ako **creat uspe, proces-dete zatvara svoj starndarni izlaz**, **duplicira fd za novu output datoteku**, čime standarni izlaz **postaje nova datoteka**. Proces dete zatvara **fd** nove datoteke jer više nije potreban.

- Redirekcija standardnog ulaza i standardne greške je slična.

shell

```
/*read commnad line until "end of file*/
while(read(stdin, buffer, numchars))
{
/* parse command line/*
if(/* command line contain & /* )
 amper=1;
else
amper=0;

/* for commands not part of shell command language*/
if(fork()==0)
{
/* redirection of I/O? /*
if(/* redirection output */)
{
        fd=creat(newfile, mask);
        close(stout);
        dup(fd);
        close(fd);
        /* stdout is now redirected*/
}
```

shell

- **if(/* piping */)** {
- pipe(fildes);
- if(fork()==0)
- { **/* first component od command line*/**
- **close(stout);**
- **dup(fildes[1]);**
- **close(fildes[1]);**
- **close(fildes[0]);**
- /*stdout now goes to pipe*/
- /*child process does, command*/
- execlp(command1, command1,0)
- } **/*2nd command line component of command line*/**
- **close(stdin);**
- **dup(fildes[0]);**
- **close(fildes[0]);**
- **close(fildes[1]);**
- /* standard input now comes to pipe*/
- }/* end of pipe*/
- execve(command2, command2, 0);
- }/* end of child*/
- **/*parent continue here…., wait for child to exit if required */**
- if(amper == 0) retid = wait(&status);
- }

shell piping

- Posmatrajmo kod za pipe situaciju u kojoj imamo single pipe:
- **ls –l | wc –l**
- Pošto roditelj forkuje i kreira **dete koje upravlja drugim delom komande linije**,
- **dete kreira pipe**, a **zatim dete forkuje svoje dete (unuče)** koje će upravljati **prvim delom komande linije**.
- Unuče, koje je nastalo kao **drugi fork izvršavače prvu komandu komponentu** komande linije (ls). Unuče upisuje u pipe, zatvara svoj standarni izlaz, duplicira pipe write deskriptor i zatvara pipe write descriptor, a zatim se izvršava prvi deo komandne linije.
- Roditelj unučeta- (ls) je proces koji će **obaviti wc**, je **dete shell procesa**, kao na slici. Ovaj proces zatvara svoj **standardni ulaz**, duplicira pipe read deskriptor, a zatim zatvara pipe read descriptor i izvršava drugi deo komandne linije.
- **Oba procesa** se **izvršavaju asinhrono**, ali se **sinhronišu preko pipe datoteke**, tako što je **izlaz jedne komande ulaz za drugu**.

shell piping

- **Glavni roditelj čeka svoje dete da završi (wc),**
- a to **dete čeka svoje dete ls –l**
- i
- **cela linija se završava kada wc obavi exit**, a prvo se završi **ls.**