Process Scheduling and time

- U **time sharing sistemima**, **kernel alocira CPU procesu** za **period vremena** koji se zove **time slice** ili **time quantum**, proces se **preempt-uje** i raspoređuje se drugi proces. UNIX koristi relativno vreme izvršavanja kao parametar da se odredi koji se proces raspoređuje sledeći.
- Svaki proces ima **scheduling prioritet**, a kernel kada obavlja **CSw** bira proces sa **najvećim prioritetom**. Kernel **ponovo izračunava prioritet** kada se proces vrati iz kernel moda u user mod i periodično podešava prioritet za ready-to-run procese.

- Neki procesi imaju potrebu da znaju **količinu vremena**, pa na primer **time** komanda meri vreme izvršavanja, a **date komanda** prikazuje datum i vreme.
- Različiti SC omogućavaju procesima da setuju ili dobijaju **kernelske vremenske podatke**.
- Sistem održava vreme preko **sistemskog časovnika**, koji šalje prekidni signal u regularnim trenucima i to je **time tick**
- Obradićemo process **scheduling** i **time related** SC.

Process scheduling

- **UNIX je poznat** kao **RR with multilevel feedback**,
- što znači da **kernel:**
  - **alocira CPU** procesu za **jedan time quantum**
  - **preemptuje** proces kad mu **istekne TimeQuantum**
  - **vraća ga** u neki od **prioritetnih queue**.
- Proces bi mogao
- da **napravi više itracija** kroz **feedback petlju**
- pre nego što **završi**

- Kada kernel obavi **CSw**
- on obnavlja **kontekst procesa**
- tako da proces uvek **nastavi** tamo gde je **stao**, bio **suspendovan**

Process scheduling

- **U okviru svakog CSw:**
  - kernel mora obaviti **Scheduling algoritam**
  - koji će izabrati proces iz **ready queue**
  - i to proces sa **najvećim prioritetom**.

- **To važi samo** za **procese koji su memoriji**, ne može se selektovati swap-ovani proces.

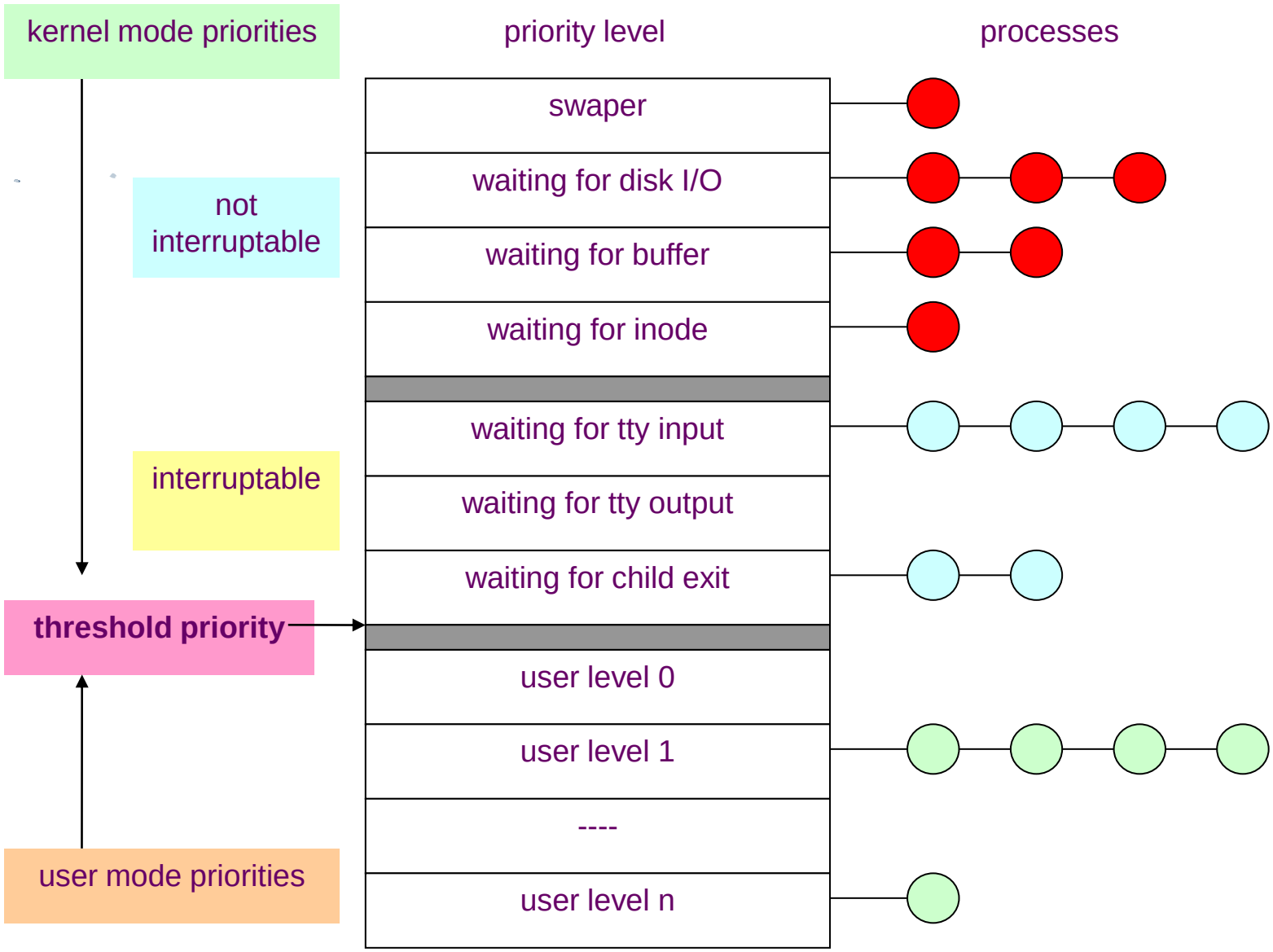
- Ako **više procesa ima isti prioritet**, bira se onaj koji je **najduže čekao**, poštujući **RR**.

- **Ako nema procesa** u **ready queue**, **CPU je idle**,
  - čeka **prvi timer tick**,
  - a **onda ponovo pokušava scheduling**

algorithm schedule_process

- **algorithm schedule_process**
- input: none
- output: none
- {
- **while(no process picked to execute**)
  - {
- for (every process on run queue)
- **pick highest priority process** that is loaded in memory;
- **if(no process eligible to execute**) idle the machine;
- /*interrupt takes machine out of idle state*/
  - }
- **remove chosen process** from **ready queue**;
- **switch context** to **that** of **chosen process**, **resume its execution**;
- }

Scheduling Parameters



Scheduling Parameters

- **Svaki ulaz u PT ima polje za prioritet**, pri čemu je prioritet procesa u user modu je funkcija **njegovog nedavnog CPU korišćenja** procesa.
- **Opseg procesovih prioriteta** može da se podeli u **2 klase** kao na slici: **korisnički prioriteti** i **krenelski prioriteti**. Svaka klasa sadrži više prioritetnih vrednosti i svaki prioritet ima queue za procese koji se **logički dodeljuju** u **taj queue.**
- **Proces sa user-level prioritetima** se preemptuju na njihovom **povratku** iz **kernelskog** u **user mod**, a procesi sa **kernel-level prioritetitima** dobijaju ih u **sleep algoritmu**.
- User-level prioriteti su **ispod threshold vrednosti**, kernel-level prioriteti su iznad **threshold** vrednosti.
- Kernel-level prioriteti se dalje dele: **procesi sa nižim prioritetima bude se po prijemu signala**, dok procesi sa **višim prioritetom nastavljaju da spavaju.**
- Slika prikazuje **threshold prioritet** između **user prioriteta i kernel prioriteta** kao **dupla linija** između 2 prioriteta "waiting for child exit" i "user level 0".
- Prioriteti "swapper", "**waiting for disk I/O**", "waiting for buffer" "waiting for inode" su visoki i **neprekidljivi sistemski prioriteti**, sa 1, 3, 2 i 1-nim procesom u redu čekanja.
- Kernelski prioriteti "**waiting for tty input**", "**waiting for tty output**"i "**waiting for child exit**" su niži prioriteti koji su prekidljivi, sa 4, 0, i 2 procesa u redu čekanja.
- Slika razdvaja user prioritete koji se ovde nazivaju "user level 0", "user level 1".. "user level n", sa 0, 4 i 1 procesom u queue

Scheduling Parameters

- **Kernel kalkuliše** prioritet procesa u **specifičnom stanju procesa**.
- Kernel dodeljuje prioritet procesu **koji treba da ode na spavanje**, pri čemu je **prioritet vezan** za **uzrok uspavljivanja procesa**. Prioritet ne zavisi od run-time karakteristika procesa (I/O bound, CPU bound).
- **Procesi** koji **spavaju u nižim algortimima** teži da izazove mnogo više uskih grla u sistemu ako su neaktivni, pa im se daje veći prioritet.
- **Na primer** proces koji spava i **čeka na disk I/O** ima **veći prioritet od procesa** koji **čeka** na **bafer** **iz više razloga**:
- **prvo: proces** koji čeka na **disk I/O** već **ima bafer** i kad se probudi ima šanse da **obavi procesiranje** i **potom oslobodi bafer** i **druge resurse**, što je dobro za sistem.
- **drugo:** proces koji čeka da **oslobodi bafer**, možda čeka bafer koji drži proces koji čeka na disk I/O i kad se disk I/O završi oba procesa se **uskoro bude**, jer **spavaju** na istoj **adresi-baferu**. **U protivnom** to je **deadlock**, ovaj **drži bafer** a čeka ga proces **većeg prioriteta** koji čeka njegov **bafer**.

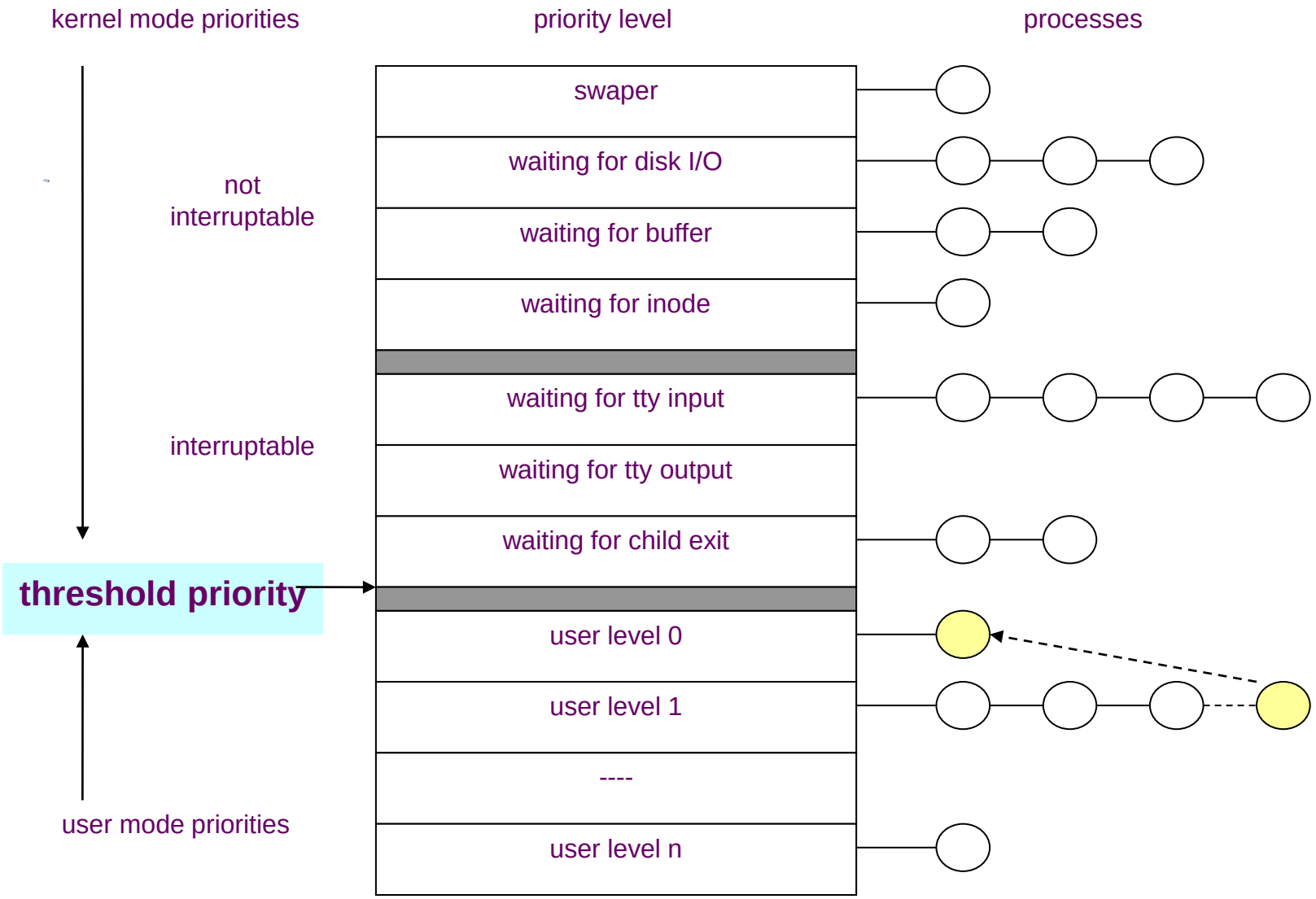
Scheduling Parameters

- **Kernel podešava prioritet procesa kada se**
  - **proces vraća**
  - **iz kernelskog u user mod**
- **Proces može prethodno** da uđe u **uspavano stanje**
  - kada mu se menja prioritet iz user-skog u **kernelski mod**
  - onda mu se **prioritet** opet **menja** kada se vrati **u user mod**
- **Kernel penalizuje proces koji se izvršava** i nastoji da bude fer prema svima procesima
- **Clock handler** prilagođava **prioritet svih procesa u user modu**,
  - na svaku 1 secundu (RR)
  - izaziva da kernel obavi **scheduling algoritam**
  - da bi sprečio da neki proces monopolizuje CPU
- **Clock** može **prekinuti** **proces više puta**
  - **za vreme** njegovog **time quantuma**
  - a na svaki timer tick clock handler inkrementira **polje u PT**
  - koji memoriše informaciju o **CPU-usage**.

Schedulling Parameters

- Na svaku sekundu, clock handler podešava isto polje na bazi decay funkcije:
- **decay(CPU) = CPU /2 = recent CPU usage (UNIX system V)**
- a zatim se ponovno izračunava prioritet
- za svaki proces i stanju "**preempteed but ready to run**" po sledećoj formuli:
- **priority = ("recent CPU usage"/2) + (base level user priority)**
- gde je "base level user priority" threshold prioritet između kernelskog i userskog moda. Što je broj manji to je prioritet procesa efektivno veći. Analizirajući formule za decay i prioritet, što više proces radi to ima veći broj (usage) a manji P, a takođe što duže čeka to ima manji broj (usage) a veći prioritet
- **Efekat one-second** rekalkulacije čini da se user procesi **pomeraju između user-queue**, što je ilustrovano na slici, gde proces menja svoj prioritetne queue sa 1 na 0. Na realnom sistemu, svi user procesi menjaju svoje queue, ali se samo jedan proces selektuje za rad. Kernel ne menja prioritete procesima u **kernelskom modu**, niti dozvoljava user procesu da pređe u kernelski prioritet zbog druge formule. (**jedino ako obavi SC**)

Scheduling Parameters



Scheduling Parameters

- Kernel pokušava da ponovno izračuna prioritet svih user procesa na svaku **T=1sec**, ali taj interval može da varira.
- Na primer ako kernel izvršava **svoje kritične sekcije**, timer tick se neće dogoditi, pa zato kernel pamti da je trebao da obavi rekalkulaciju.
- Kernel podešava **inicijani prioritet** za neke procese kao što su **editori teksa** (high **level of idle_time_to_CPU_usage** ratio) na **visoke brojeve** odnosno na **niski prioritet**.
- Postoje implementacije UNIX gde **TQ varira**
  - **dinamički između 0 i 1 sekunde**,
  - **zavisno od opterećenja sistema**.
- Takvi sistemi daju **brži odziv** jer ne moraju da **čekaju na 1secundu**, ali kernel ima više **kontekst switch-eva**.

Examples of process Scheduling

- Kernel kalkuliše decay of CPU usage preko formule:

- **decay(CPU) = CPU /2 = recent CPU usage** **(UNIX system V)**
  - a zatim se ponovno izračunava prioritet za svaki proces i stanju
  - "**preempteed but ready to run**" po sledećoj formuli:

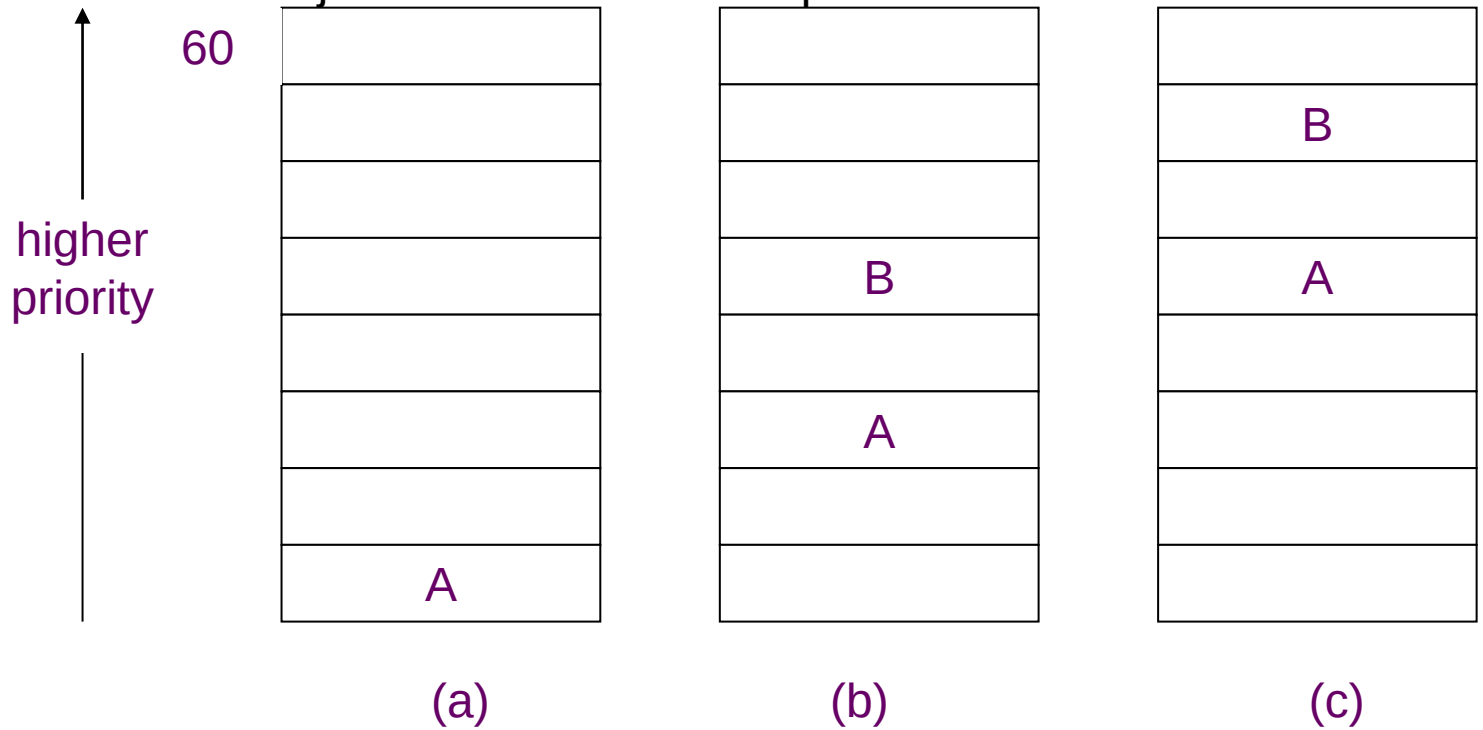
- **priority = ("recent CPU usage"/2) + 60**

- Pretpostavimo da je A prvi proces za izvršavanje i staruje da radi na TQ=1sec.
- Za vreme od jednog **TQ**, 60 puta se dogodi time tick, tako da se CPU usage polje uveća na 60 proces A na 60.
- Potom istekne TQ, kernel obavi CSw i rekalkuliše prioritete za sve procese (A=75, B=60, C=60) i bira proces B za izvršavanje.
- Opet time **tick 60 puta** okida, a pa se opet dogodi **TQ** i rekalkulacija proriteta

primer1						
time	proc A		proc B		proc C	
base	60		60		60	
nice	0		0		0	
seconds	priority	cpu count	priority	cpu count	priority	cpu count
0	60	0	60	0	60	0
		1				
		2				
		..				
		60				
1	75	30	60	0	60	0
				1		
				2		
				..		
		30		60		
2	67	15	75	30	60	0
						1
						2
						..
		15		30		60
3	63	7	67	15	75	30
		8				
		9				
		..				
		67		15		30
4	76	33	63	7	67	15
				8		
				9		
				..		
		33		67		15
5	68	16	76	33	63	7

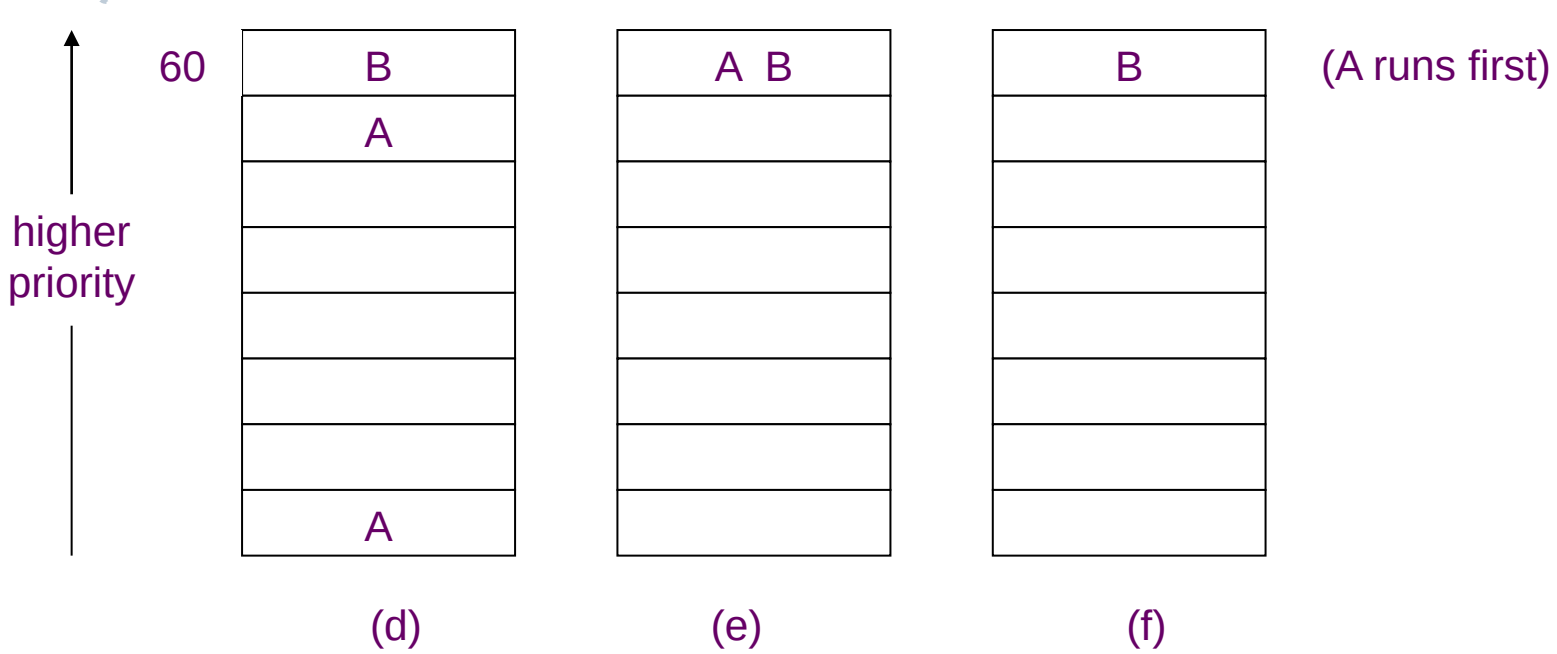
Examples of process Scheduling

- Posmatrajmo slučaj na slici sa prioritetima i pretpostavimo da su to jedini procesi u sistemu. Kernel može preemtovati proces A i vratiti ga u stanje ready-to-run, a kao posledica više TQ koji su se dogodili, njegov prioritet pada (**slika a**).
- Zatim u sistem uđe proces B čiji inicijalni prioritet može biti veći od A (slika b).
- Ako postoje i drugi procesi, A i B mogu ostati neko vreme u ready-queue, tako da im se prioriteti približe ili izjednače (slika c i d), a može se dogoditi da A bude selektovan jer duže čeka u istom queue.



Examples of process Scheduling

- Ako postoje i drugi procesi, A i B mogu ostati neko vreme u ready-queue, tako da im se prioriteti približe ili izjednače (slika c i d), a može se dogoditi da A bude selektovan jer duže čeka u istom queue.



Examples of process Scheduling

- Podsetimo da kernel bira proces na završetku CSw:

- CSW:proces mora da **obavi CSw**
  - 1. kada ide na spavanje
  - 2. kad obavi exit
  - **3. preemption** kada se vraća iz kernelskog moda u user mod (preemption) i

- **Preemption** se dešava ako u **ready-queue** postoji proces **većeg prioriteta.**
- To se dešava:
- **1. prvi slučaj** ako **kernel probudi proces visokog prioriteta**,
  - na primer svaki **probuđeni kernelski proces** ima veći prioritet
- **2. drugi slučaj** je kad **clock handler obavi reviziju prioriteta** za sve u **ready-queue**, pa tekućem **procesu padne prioritet** a **drugome procesu se poveća..**

Controlling Process priorities

- Procesi mogu sami da utiču na svoj prioritet preko **nice SC**:
- **nice(value);**
- gde je value vrednost koja se dodaje na formulu
- **priority =**
- **("recent CPU usage"/constant)**
- **+ (base level user priority)**
- **+(nice value)**
- Nice SC može **inkrementirati ili dekrementirati nice** polje u PT za vrednost datu u nice SC, mada **samo root može povećavati prioritet procesa**. Običan user može **smanjiti svoj prioritet .**
- Procesi nasleđuju **nice vrednost** od **svog roditelja**.
- **Nice SC** radi samo na procesu koji se izvršava
  - (**proces samom sebi menja nice i nijednom drugom procesu**).
- **To znači da administrator ne može da smanji prioritet procesima koji zauzimaju dugo CPU, osim da ih ubije sa kill SC.**

Fair Share Scheduler

- Do sad opisan **scheduler algoritam ne razlikuje klase usera.**
- No pojavila se potreba da se uvede mehanizam za favorizovanje grupe usera, kako bi im se obezbedio bolji odziv, šema se naziva **Fair Share Scheduler.**
- **Princip FSS** je **podeliti sve usere** na **fair share groups**,
  - tako da članovi grupe imaju običan **sheduling** za svoju grupu,
  - a **sistem alocira CPU proporcionalno svakoj grupi**
  - **bez obzira na broj procesa u grupi**
- Na primer pretpostavimo da ima
  - **4 fair grupe** u sistemu i
  - da svaka grupa **dobije 25% CPU** i
  - da **grupe sadrže 1, 2, 3 i 4 CPU bound** procesa koji nikada ne završavaju (na primer svaki je endless loop).
  - Imamo 10 procesa i ako bi se koristio običan RR, svaki bi proces imao 10% CPU usage (ima 10 procesa). Ali ako se koristi FSS, proces u grupi 1 imaće 2 puta više CPU, nego u grupi 2, 3 puta više nego u grupi 3 i 4 puta više nego u grupi 4, **dok za istu grupu** procesi imaju ravnomerno dodeljen CPU.

Fair Share Scheduler

- Impementacija ove šeme je jednostavna, a može joj se **dodati**
- **"fair share group priority".**

- Svaki proces ima **novo polje u svoji u-area**,
- koje ukazuje na fair share CPU usage polje **koje je zajedničko za sve procese iz grupe.**

- Svaki timer tick inkrementira polje FSG CPU, kao što inkrementira CPU usage polje za tekući proces, **na jednu sekundu se izračunava**
  - **decay vrednost** za sve procese
  - **decay** za **celu grupu.**

- Kada se izračunava prioritet procesa,
- nova komponenta koja se dodaje je **grupni CPU usage**,
- normalizovan za CPU vreme dodeljeno grupi.

Fair Share Scheduler -example

- Na primer, uzmimo opet ona 3 procesa,
  - ali da je A u jednoj grupi
  - a B i C u drugoj grupi.

- **Neka se prvo izvršava** A, što će inkrementirati CPU i group polje u toku 1-og TQ=1sec.
- Kada se ponovno **izračunava prioritet**, procesi B i C su većeg prioriteta, i neka kernel **izabere B.**
- B se izvršava a ažurira se **njegovo usage polje** i **grupno usage polje zajedničko za B i za C**.

- Kada se u sekundi 2 izračuna prioritet,
- za proces C će da bude 75,
- a za **A=74**,
- pa je sekvenca procesa **A, B, A, C, A** itd.

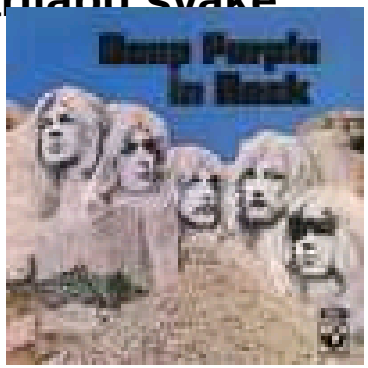
Time	proces A			proces B			proces C		
	priority	CPU	Group	priority	CPU	Group	priority	CPU	Group
	60			60			60		
0	60	0	0	60	0		60	0	
		1	1						
		2	2						
		..	..						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					..	..			..
		30	30		60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		75	75		30	30			30
3	96	37	37	74	15	15	67	0	15
						16		1	16
								2	
		37	37		15	75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20							
		78	78		7	37		30	37
5	98	39	39	70	3	18	76	15	18

Real time processing

- Real time procesiranje implicira mogućnost da obezbedi **trenutni odgovor na eksterne događaje**, tako što se **izabere odgovarajući proces** kao **odziv na događaj**, ali u **ograničenom intervalu vremena.**
- Na primer, **life-suport sistem** u bolnici mora odreagovati na svaku promenu stanja bolesnika. Na drugoj strani editori teksta nisu real-time, user očekuje brzi odgovor, ali ako sačeka par sekundi, to nije strašno.
- **Prethodni scheduler algoritmi nisu pogodni za real-time**, zato što **ne mogu da garantuju da će startovati neki proces u okviru fiksnog vremenskog limita.**
- **Druga nepovoljnost** za **real-time kod UNIX-a** je što je kernelski **mod non-preemptive**, kernel ne može izabrati real-time proces koji je user modu ako već izvršava proces koji je u **kernel modu**, osim ako se UNIX osetno ne promeni.
- U principu, programeri bi mogli ubaciti **real time** procese **u kernel** da bi **postigli real time odziv**.
- Pravo rešenje je dozvoliti real time procesima da **postoje dinamički** (ne u kernelu) **ali da imaju mehanizam** da **obaveste kernel** o njihovim **real-time ograničenjima** .

System call for time

- Ima više vremenski orijentisanih SC,
- **stime**, **time**, **times** i **alarm**
  - od kojih se (**stime** i **time**) bave **globalnim sistemskim vremenom**,
  - a **times** i **alarm** se bave **vremenom za individualne procese**.
- Poziv **stime** dozvoljava superuseru da **setuje** **globalnu kernelsku varijablu** koja sadrži vrednost za **tekuće sistemsko vreme**:
- **stime(pvalue);**
- gde je **pvalue** **long integer** koji daje vreme mereno **u sekundama** u odnosu na **Januar 1, 1970 GMT**. **Clock prekid inkrementira ovu varijablu svake sekunde.**



- Poziv **time** uzima sistemsko vreme kao
- **time(tloc);**
- gde je **tloc** user lokacija u koju poziv upisuje vreme.
- Komande tipa **date** koriste **SC time** da bi dobili tekuće vreme**.**

System call for time

- Poziv **SC times** dobija **kumulativno vreme** koje je prozvani proces proveo izvršavajući se u korisničkom i **kernelskom modu** i **kumulativna vremena** koja su **sva njegova zombie deca provela izvršavajući** se u **korisničkom i kernelskom modu:**
- **sintaksa** za **times** je:
- **times(tbuffer)**
- **struct tms *tbuffer;**
- gde je struktura **tms definisana** na **sledeći način**:
- struct **tms**
- {
- **/* time_t is data structure for time*/**
- **time_t tms_utime**; /* user time of process*/
- **time_t tms_stime;** /* kernel time of process*/
- **time_t tms_cutime;** /* user time of children*/
- **time_t tms_cstime**; /* kernel time of children*/
- }
-
- **Times** vraća proteklo vreme iz **proizvoljne tačke iz prošlosti**, obično od **vremena podizanja sistema.**

System call for time-example

- Na slici je prikazan program koji kreira 10 dece, i
- svako dete obavi petlju 10000 puta.

- Proces roditelj pozove **times SC**,
  - **prvi put**, pre kreiranja dece i
  - **drugi put, kada sva deca završe** (exit),
- a **proces dete** zove **times** pre i posle svoje petlje.

- Neko će naivno pomisliti
  - **da su roditeljska vremena** (**user time of children** i **kernel time of children**)
  - **jednaka zbiru vremena user i kernelskih parametara dece**,
  - ali **deca uopšte ne računaju fork SC** kao i svoj **exit SC**.

- Takođe sva vremena se **menjaju zbog prekida** i **kontekst CSw**.

System call for time-example

```
#include <sys/types.h>
#include <sys/times.h>
extern long times();
main()
{
        int i;
        struct tms pb1, pb2;
        long pt1, pt2;
pt1 = times(&pb1);                  /*start time*/
for (i=0; i<10; i++) if (fork()==0) child(i);

for (i=0; i<10; i++) wait((int *) 0 );
pt2 = times(&pb2);                  /*finished time*/
printf("parent real %u user %u sys %u cuser %u csys %u", pt2-pt1,
pb2.tms_utime-pb1.tms_utime, pb2.tms_stime-pb1.tms_stime,
pb2.tms_cutime-pb1.tms_cutime, pb2.tms_cstime-pb1.tms_cstime);
}
```

System call for time-example

- **child(n)**
- int n;
- {
- int i;
- struct tms cb1, cb2;
- long t1, t2;

- **t1 = times(&cb1);** /*start time*/

- for (i=0; i<10.000; i++) ;

- **t2 = times(&cb2);** /*finished time*/

- printf("**child** %d **real** %u **user** %u **sys** %u", n, t2-t1, cb2.tms_utime-cb1.tms_utime, cb2.tms_stime-cb1.tms_stime);

- **exit**();
- }

alarm SC

- User procesi mogu da prozovu **alarm signale** preko **alarm SC**.
- Na primer, progam na slici proverava **vreme pristupa datoteci** svakog **minuta** i prikazuje poruku da je **datoteka imala pristup**.

- Da bi se to uradilo, generiše se beskonačna petlja:
  - za vreme svake iteracije pozove se **stat SC**
  - koji obaveštava o vremenu pristupa datoteci i
  - ako je u toku **zadnjeg minuta** došlo do pristupa,
  - štampa se poruka.

- Proces zatim postavlja **signal SC**,
  - catching alarm signal,
  - poziva alarm SC da emituje alarm signal za 60 sekundi, i
  - zove pause SC da supenduje svoju aktivnost dok ne primi signal.
  - Posle 60sec, pojavi se alarm signal,
  - kernel setuje user stack za proces da pozove signal catcher
  - (wakeup u ovom slučaju),
  - koji vrati proces na stanje iza pause SC, a to je petlja.

alarm SC - example

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>

main(argc, argv)
int argc;
char *argv[]
{
        extern unsigned alarm();
        extern wakeup();
        struct stat statbuf;
        time_t axtime;
if(argc != 2)
{
                printf("only 1 arg");
                exit();
}
axtime = (time_t) ->0;
```

alarm SC - example

```
for(;;)
{
    /* find out file access time*/
if(stat(argv[1], &statbuf ) == -1)
    {
    printf("file %s not there", argv[1]);  exit();
    }
if(axtime != statbuf.st_atime )
    {
                printf("file %s accessed", argv[1]);
                axtime  = statbuf.st_atime;
    }
signal(SIGALRM, wakeup)
alarm(60); /* wakeup after 60 sec) /*slanje alarm, signala procesu na zahtev*/
pause(); /* go to sleep*/
}/*for*/
}/*main*/
wakeup() {}
```

Clock

- **Funkcije** za **clock interrupt handler** su:
  - **restart časovnika**
  - **scheduling pozivanja internih kernel funkcija** baziranih na unutrašnjim tajmerima
  - obezbeđivanje za **profiling** za kernelske i user procese
  - **skupljanje account statistike** za sistem i proces
  - čuvanje zapisa o vremenu
  - **slanje alarm signala,** procesu na zahtev
  - **periodično buđenje swapper procesa**
  - **kontrola process-CPU schedulinga**
- Neke operacije se **rade na svaki clock prekid**, dok se neki izvršavaju na više tick-ova.
- Clock handler se **izvršava** na **visokom CPU ExLevel**, spečavajući mnoge prekide da se obrađuju dok handler radi.
- **Clock handler mora zato biti brz**,
  - jer to **kritično vreme** kad su prekidi blokirani
  - mora da bude **veoma kratko**.

algorithm clock

- algorithm clock
- input: none
- output: none
- {
- **restart clock**; /* so that it will interrupt again*/
- **if(callout table** not empty)
- {
- **adjust callout time**;
- **schedule callout** function **if time elapsed**;
- }
- **if(kernel profiling on)** note program counter at time of interrupt;
- **if(user profiling on)** note program counter at time of interrupt;
- **gather system statistics;**
- **gather statistics per process;**
- **adjust measure of process CPU utilization;**

algorithm clock

- **if(1 second or more since last here** and **interrupt not in critical region** of **code**)
- {
- for (**all process in the system**)
- {
- **adjust alarm time** if **active**;
- **adjust measure** of **process CPU utilization**;
- if(process to execute in **user mode**) adjust **process priority**;
- }
- **wakeup swapper** process **if is necessary**;
- }/*if*/
- }

Restarting the clock

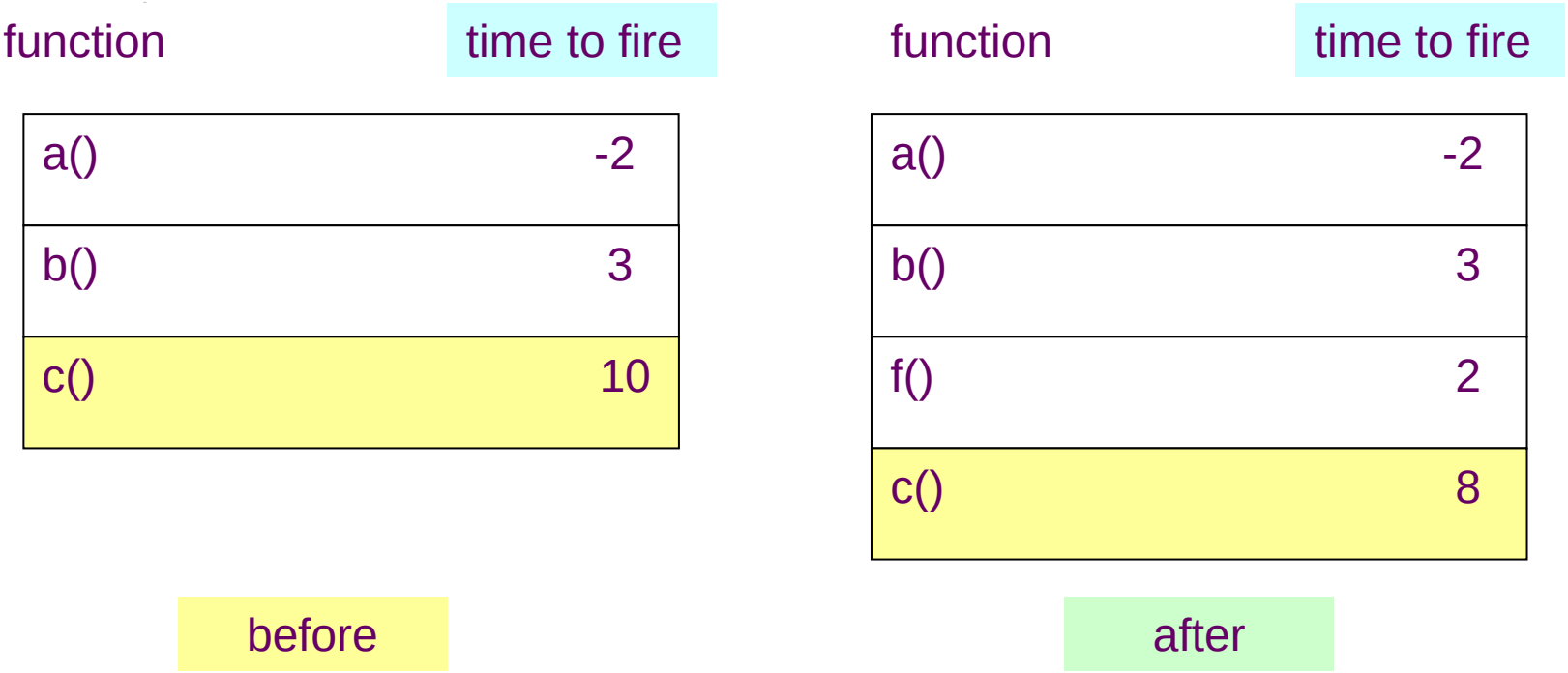
- Kada se dogodi clock prekid,
- mnoge mašine zahtevaju,
- da se omogući **clock prekid ponovo**,
- a takve instrukcije zavise od hardvera i nećemo ih ovde diskutovati.

Internal system timeouts

- Neke kernel operacije, posebno device drajveri i mrežni protokoli, zahtevaju pozivanje kernel funkcija na real-time osnovi.
- Na primer, **proces može postaviti terminal** u **raw mod** tako da kernel **zadovoljava read zahteve** u **fiksnim vremenskim intervalima**, umesto da čeka user-a da otkuca **CR karakter**.
- Kernel čuva sve potrebne informacije u **callout tabeli** (viditi algoritam za clock), **koja se sastoji od**
  - **funkcije** koju treba izvršiti kada istekne vreme,
  - **parametar za funkciju**
  - **broj timer tikova** koji treba da se dogodi pre nego što se funkcija pozove.
- Korisnik nema direktnu kontrolu nad ulazima **callout tabele**, ali različiti kernel algoritmi mogu da ih **kreiraju po želi usera**.
- Kernel sortira ulaze **callout table** prema parametru "**time to fire**", nezavisno od redosleda po kojim su pristigli u tabelu.
  - zbog vremenskog poretka polje za "time to fire" se **namešta** u odnosu na p**rethodni** ulaz (previous time to fire).
  - **totalni time to fire** je **zbir svih elementa ispred ulaza i sam ulaz**.

callout example

- Slika prikazuje jednu **callout** tabelu i dodavanje novog ulaza za funkciju f koju treba izvršiti posle 5 clock tika. Zato se ona ubacuje između b i c, a za c se menja realtni broj u 8.



callout example

- Kernel može koristiti linkovane liste za svaki ulaz tabele ili može podesiti sve ulaze, što je bolja varijanta kada se callout tabela ne koristi mnogo.
- **Na svaki clock prekid**, clock handler proverava da li ima ulaza u callout tabeli **i ako ih ima dekrementira se time polje prvog ulaza**.
- To praktično **znači** da su i **svi ostali ulazi dekremntirani**, jer kernel sabira adrese svi prethodnih.
- Ako je vremensko polje prvog **ulaza manje ili jednako 0**, tada se poziva odgovarajuća funkcija.
- Clock handler **ne poziva funkciju direktno**, jer bi **mogao da se blokira** (na **primer timer irq** je **blokiran** a ako pozove funkciju koja traje duže od 1 timer tick-a, taj tick je izgubljen.
- Umesto toga, **clock handler** pozove funkciju u vidu **softveskog prekida**, a on je na **CPU ExLevel** koji dozvoljava **hardverske prekide**.
- **Dok se izvršava** funkcija (**software interrupt**)
  - **mnogi se timer tick mogu dogoditi**,
  - pa prvo polje može imati **negativne vrednosti**.
  - Kada se softverski prekid na kraju završi,
  - **ukljanjaju se svi ulazi iz callout tabele koji su istekli**
  - a **njihove funkcije se pozovu**.

callout example

function	time to fire
a()	-2
b()	3
a()	10

before

function	time to fire
a()	-2
b()	2
c()	10

after

- Kada je polje **prvog ulaza negativno**, **clock handler** mora naći prvi pozitivan ulaz i dekrementirati ga. Na slici, vredsnot prvog ulaza je -2, što znači da je prošlo 2 timer ticka od kad je a trebao da se izvrši, pa kernel preskače a i dekremenira polje za b.

Profiling

- **Kernel profiling obavlja merenje vremena,**
  - **koliko je sistem radio u kernelskom modu**
  - koliko u **user modu**
  - koliko su se **pojedine rutine izvršavale u kernelu**.
- **Kernel profile driver**
  - **monitoriše relativne performanse kernelskih modula**,
  - odabirajući sistemske aktivnosti,
  - kada se dogoditi tajmerski prekid.
- **Profile drajver** ima **listu kernelskih adresa za sampling**,
  - to su obično adrese kernelskih funkcija.
  - **Proces je prethodno downloadovao te** adrese upisom u **profile drajver**.
- Ako je **kernelski profiling pušten**,
  - clock handler poziva interrupt handler za profile drajver,
  - koji prvo određuje koji je mod (user/kernel) bio pre timer tick-a.
  - Ako je bio user mod,
  - profiler inkrementira interni brojač koji odgovara PC registru.
- Korisnički procesi mogu **čitati profile drajver** da **dobiju kernelske brojače** i **obave statistička merenja.**

example

- Na primer, sledeća slika prikazuje adresu **kernelskih rutina**:

Algorithm	Address	count
bread	**100**	**5**
breada	**150**	**0**
bwrite	**200**	**0**
brelse	**300**	**2**
getblk	**400**	**1**
user	**-**	**2**

- Ako je sekvenca **PC vrednosti samplovanih u 10 timer ticks**,
  - **110, 330, 145, 125, 440, 130, 320, 104**, onda je to gornja slika.
  - Na prvi pogled, sistem je proveo 20% u user modu, a 50% u algortitmu bread.
  - 20% u brelse i 10%u getblk
- Ako se profiling obavlja za duži period vremena slika korišćenja sistema je realnija.
- Ali **profiler ne uključuje** koliko vremena sistem proveo u **clock handleru** i **kritičnim sekcijama koda**,
  - zato što se profile ne poziva tada,
  - a kritični delovi koda zauzimaju značajni deo profila

profiling in the user mode

- Korisnik može obaviti profiling za procese u **user modu** pozivajući **profil SC**:
- **profil(buff, bufsize, offset, scale);**
- gde je:
  - **buff** adresa polja (array) u user prostoru,
  - **bufsize** je veličina polja,
  - **offset** je **virtulena adresa korisničkog potprograma**, i
  - **scale** je faktor koji mapira korisničke vituelne adrese u polje (array).

- Kernel tretira **scale** kao **binarnu frakciju** (fiksne dužine).

- Kada **clock prekine program u user** modu, clock handler određuje PC u vreme prekida, komparira offset, inkrementira lokaciju u buf na bazi **bufsize** i **scale**.

example

- Na primer posmatrajmo **program za profiling**,
- koji poziva **2 funkcije u beskonačnoj petlji**.

- Proces prvo pozove signalSC da ubaci funkciju **theend** kao **catcher za SIGINT**,
- zatim kalkuliše **opseg text adresa koji želi da profiluje**,
- proširujući adresu funkcije main sa adresom funkcije **theend** i
- na kraju zove **profil SC** da informiše kernel da želi **profil** svoje **exekucije**.

- **Na prvi prekid sa tastature** <Ctrl-C> se **generiše buf display**.

example

- #include <signal.h>
- **int buffer[4096]**
- main()
- {
- int offset, endof, scale, eff, gee, text;
- **extern theend(), f(), g();**
- **signal(SIGINT, theend)**
- **endof** = (int) **theend**; **offset** = (int) **main**;
- /* **calculate number of words in program text**/
- **text** = (endof-offset+sizeoff(int)-1)/sizeoff(int);
- **scale=0xffff;**
- **printf("offset %d endof %d text %d ", offset, endof, text)**
- eff = (int) f; gee = (int) g;
- printf("f %d g %d fdiff %d gdiff %d", eff, gee, eff-offset, gee-offset)
- **profil(buffer, sizeoff(int)*text, offset, scale);**

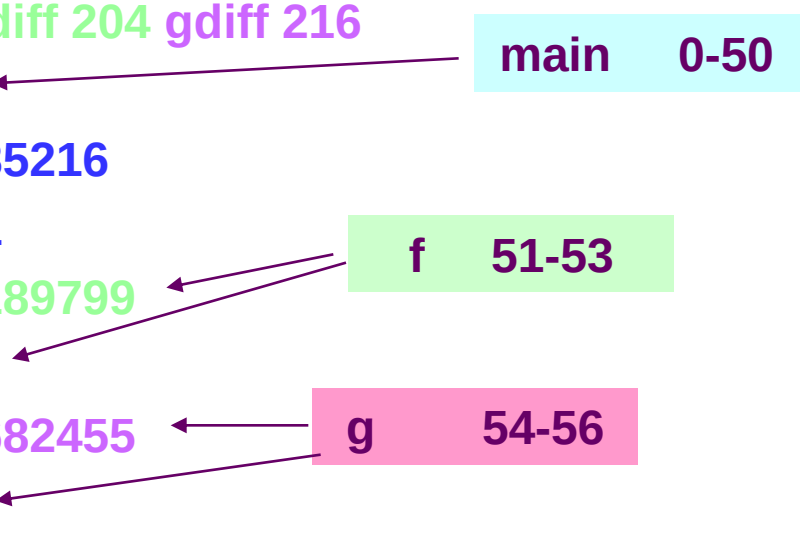
example

```
for(;;)
 {
        f();
       g();
 }
}

theend()
 {
int i;
for(i=0, i<4096; i++)
if (buffer[i]) printf("buf[%d] = %d", i, buffer[i]);
exit();
 }
```

results

- Izlaz bi mogao d bude sledeći.
- **offset 212 endof 440 text 57** **ceo program 57 ulaza {0-56}**
- **f 416 g 428** **fdiff 204 gdiff 216**
- **buf[46] = 40**
- **buf[48] = 8585216**
- **buf[49] = 151**
- **buf[51] = 12189799**
- **buf[53] = 65**
- **buf[54] = 10682455**
- **buf[56] = 67**



- Adresa f je 204 veća od 0th profiling adrese, i
- **f** se **mapira na bafer ulaze 51, 52, 53.**
- **g se mapira na 54, 55 i 56.**
- Adrese 46, 48 i 49 su adrese glavne petlje u main funkciji

accounting and statistics

- **Kada clock prekine** sistem, sistem može imati izvršavanje u **kernelskom modu** u **user modu** ili može da bude **besposlen.**
- Ako je **idle, svi procesi su uspavani**.
- Kernel čuva **interne brojače za sva stanja procesora** i podešava ih uvek za **vreme clock prekida**.
- User procesi mogu kasnije da izvuku te infomacije iz kernela i naprave statistiku.
- Svaki proces ima **2 polja u svojoj u-area** u kojoj kernel loguje **memory usage**. Kada se dogodi clock prekid, kernel izračunava **totalnu memoriju za privatne** regione procesa i proporcionalno **korišćenje shared regiona**.
- **Na primer**, ako proces
  - deli text region od 50K sa 4 druga procesa, a
  - koristi svoj privatni data i
  - stack region veličine 25 i 40K,
  - kernel tereti proces sa 75K = 50/5 + 25 +40).
- Za **paging sistem**, kalkuliše se ukupan broj **validnih stranica** u svakom regionu, tako što sabira **sve validne stranice** u **privatnim regionima** + **proprocionalan broj shared starnica**.
- Kernel upisuje ove informacije u **accounting record** kada proces obavi **exit**, i to se može koristiti **za naplaćivanje usluga korisnicima**. .

Keeping time

- Kernel inkrementira **timer promenljivu** na **svaki timer tick**,
- vreme se cuva kao **broj ticks od butiranja sistema**.

- Kernel **koristi ovu varijablu** da **vrati vreme** na **time SC** i
- izračunava **ukupno vreme (real) izvršavanja procesa**.

- **Kernel čuva procesov start time** u **njegovoj u-area** procesa **još u fork SC**, i oduzme vreme iz **timer promenljive kada** proces **izađe (exit**) i **tako dobije realno vreme izvršavanja procesa.**

- **Druga vremenska varijabla** se **ažurira svake sekunde**, i
  - **služi za stime SC,**
  - a opisuje **kalendarsko vreme.**