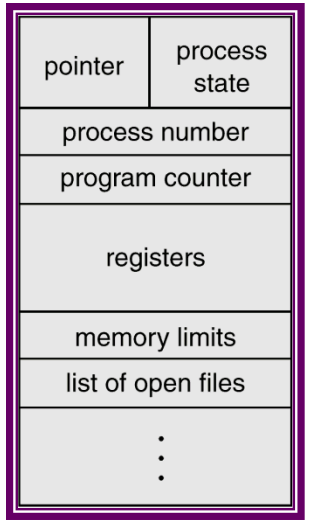
Structure of processes

- Kernel sadrži 2 strukture koje opisuju proces
- **1. PT (process table)** = globalna tabela sa jednim ulazom za svaki proces
  - **PT ulaz opisuje stanje svakog aktivnog procesa** u sistemu,
- **2. u area** za svaki proces pojedinacno
  - u-area **sadrži dodatne informacije** koje kontrolišu operaciju procesa.
- **Ulaz u PT** i **u-area** su deo **konteksta procesa**.
- **procesi** se **najviše razlikuju** po **njihovim adresnim prostorima**.



- U ovoj lekciji obradićemo:
  - stanja procesa i tranzicije
  - principe memorijskog upravljanja za procese i kernel i kako se upravlja virtuelnom memorijom
  - kontekst procesa i low-level algoritme koji manipulišu sa kontekstom procesa
  - način na koji kernel čuva kontekst procesa za vreme prekida, SC i kako nastavlja prekinuti proces
  - algoritme za manipulaciju procesnim adresnim prostorom
  - algoritme za sleep i wakeup

Process states and transitions

- U lekciji 2 je data **osnovna šema** stanja procesa i tranzicija, a ovde će biti data **potpuna šema**.
- **Sledeća lista** sadrži **kompletno stanje procesa**:
  - 1. **User Running:**
  - 2. **Kernel Running:**
  - 3. **Ready to run in memory:**
  - 4. **Asleep in memory:**
  - 5. **Ready to run, Swapped:**
  - 6. **Sleep, Swapped:**
  - 7. **Preempted:**
  - 8. **Created:**
  - 9. **Zoombie:**

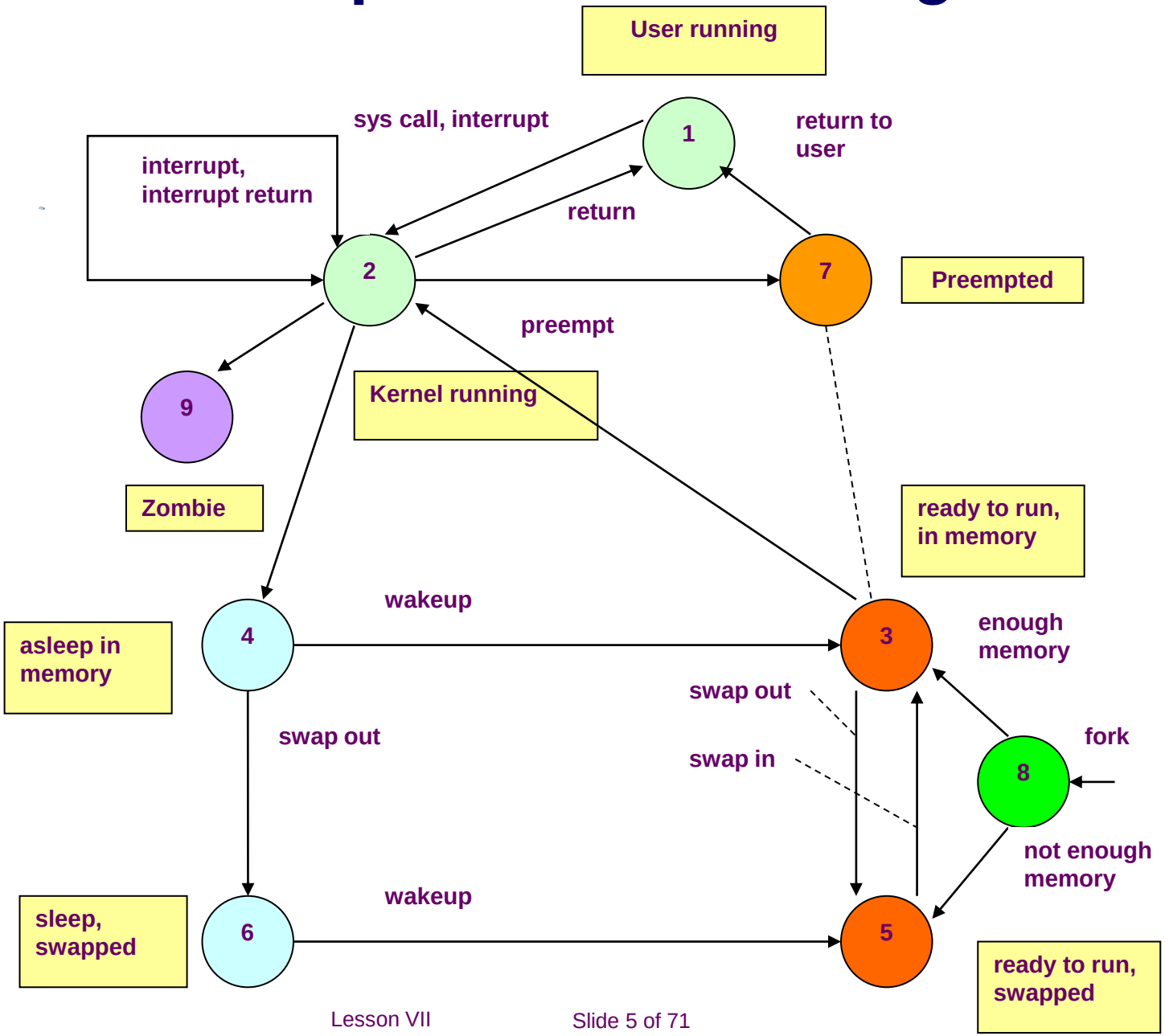
Process states and transitions

- 1. **User Running**:
  - **Proces se izvršava** u korisničkom modu
- 2. **Kernel Running**:
  - **Proces se izvršava** u kernelskom modu
- 3. **Ready to run in memory**:
  - Proces **se ne izvršava** ali je **spreman potpuno** čim ga kernel prozove
- 4. **Asleep in memory**:
  - Proces je **uspavan u memoriji**
- 5. **Ready to run, Swapped**:
  - **Proces je spreman za rad**, ali je na **swapu** pa proces **swapper** mora da ga **prebaci u memoriju pre** nego što ga **kernel prozove**

Process states and transitions

- 6. **Sleep, Swapped:**
  - **Proces je uspavan** na **swapu, prvo uspavan pa prebačen** na **swap**
- 7. **Preempted:**
  - **proces se vraća** iz **kernelskog moda** u **korisnički mod**,
  - **ali kernel** ga **preempt-uje**
  - i **obavlja kontekst switch** da bi **aktivirao drugi proces**.
  - **vrlo brzo** će preći u **stanje 3 (ready to run)**.
- 8. **Created:**
  - Proces je **novo kreiran** i **nalazi se u tranzicionom stanju**,
  - praktično, **proces postoji** ali **nije ni spreman za rad,** a **nije ni uspavan**,
  - ovo je praktično stanje svih procesa izuzev za proces 0.
- 9. **Zoombie:**
  - Proces izvršava **exit SC** i nalazi se u **zombi stanju**.
  - **Proces više ne postoji**, ali ostavlja **exit_code**
  - i **neke vremenske statističke podatke**
  - **za procesa roditelja** koji ih sakuplja.
  - **zombie** stanje je **finalno stanje** procesa.

full process-state diagram



Proces-kontrolabilne tranzicije

- Proces **može sam kontrolisati neke tranzicije** na **user-nivou.**

- **1. Prvi slucaj - fork,**
  - **proces** može da kreira novi proces (fork SC),
  - a stanje novog procesa će biti ili 3 (**Ready to run in memory**) ili 5 (**Ready to run, swapped**)
  - zavisno od kernela i to ne zavisi od samog procesa.

- **2. Drugi slucaj – SC (any)**
  - proces može uvek sebe prebaciti sa 1 u 2 tako što pozove SC,
  - ali za obrnutu transformaciju iz 2 u 1 proces nema kontrolu, to je pod ingerencijom kernela,
  - a može da se to ne dogodi nikad (2->1),
  - ako se pojavi **signal Termination** i da uleti u zombie stanje.

- **3. slucaj –SC exit** ,
  - proces sam izvršava **exit SC**,
  - koji ga prebacuje u **zombie** stanje.

Kernel-kontrolabilne tranzicije

- **Sve druge tranzicije** su **pod kontrolom kernela**,
  - **zavise** od **spoljnih događaja**
  - **zavise** od **količine memorije**,

- **neka pravila** su **jasna**,
- **ni jedan proces** se **ne može preempt-ovati** dok je u **kernel modu**,
- **preemption**, samo kada se obavlja tranzicija **2->1**.

- **Sleep – Wakeup**

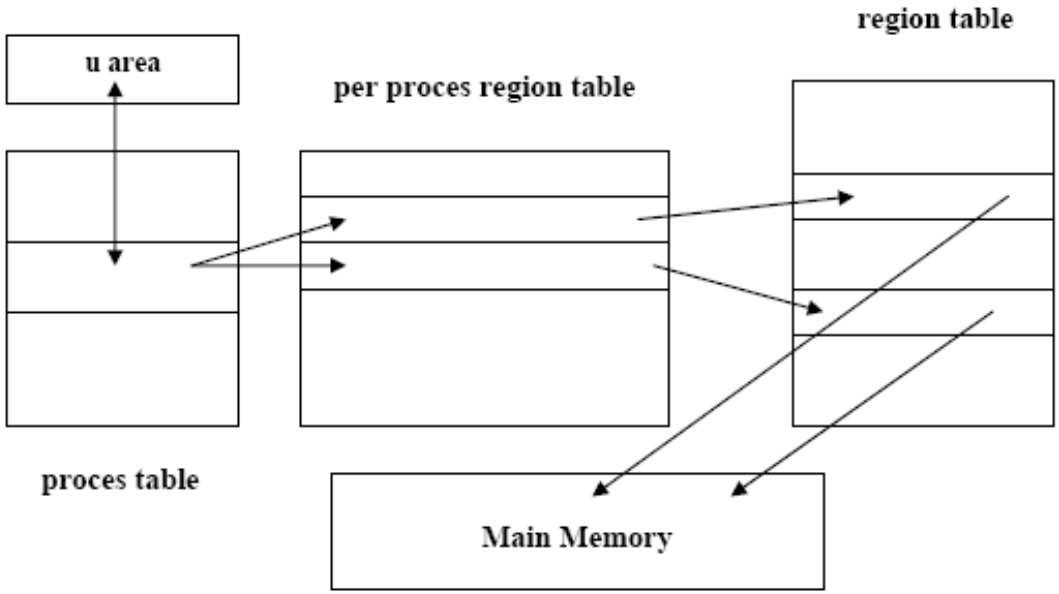
- **Swap-out, Swap in**
- **Ukoliko** je **preveliki broj procesa**,
  - **swapper** može **neki od procesa prebaciti iz**
    - (**3-Ready to run in memory**)
    - u
    - (**5-Ready to run, swapped**).

PT entry + u area

- **Dve strukture podataka** opisuju **stanje procesa**:
- **PT** ulaz: koji **sadrži polja** koja su uvek **raspoloživa kernelu**

- **u-area sadrži polja** koja su **potrebna jedino procesu koji se izvršava**

- kernel alocira prostor u **u-area** svaki put kada se **kreira novi proces**,
- tako da samo popunjeni ulazi u PT imaju svoje **u-area alokacije**.



PT entry

- Polja u PT ulazu su:
  - **1. state field**: identifikuje **stanje procesa**
  - **2. u-area pointers**:
  - omogućaju **kernelu** da **lociraju u-area** u memoriji ili eventualno na disku.
  - **Kernel koristi ove informacije za obavljanje Context Switch**,
    - kad se proces prebacuje iz stanja 3 (**ready to run in memory)** u 2 (**kernel running**) ili
    - iz stanja 7 (**preemted**) u stanje 1 (**user running**).
  - Takođe, **kernel koristi** ove **infomacije kada radi swaping** ili **paging** procesa između **2 memorijska stanja** i **2 odgovarajuća swap stanja**..
  - **3. proces size:** polje ukazuje kolika je **veličina procesa**, kako bi kernel mogao da alocira dovoljno memorije za proces

PT entry

- **4. user IDs or UIDs**:
  - **nekoliko user identifikatora** koji određuju **privilegije za proces**.
  - Na primer UID polja određuju skup procesa koji mogu međusobno da **razmenjuju signale** između sebe.
- **5. process IDs or PIDs**:
  - nekoliko **proces identifikatora** koji **specificiraju relacione odnose između** samih **procesa i**
  - **ta polja** se **postavljaju prilikom fork SC**, kada proces **ulazi** u **stanje created**.
- **6. event descriptor: polje značajno** kad je proces **uspavan**
- **7. scheduling parameters:**
  - omogućavaju kernelu da **odrede poredak** u kome se procesi pomeraju u stanja 2 (kernel running) i stanja 1 (user running)
- **8. signal field:**
  - **označava signale** koji su **poslati procesu** a **nisu** još **obrađeni**
- **9. timer parameters:**
  - označavaju **vreme izvršavanja** i **korišćenje kernelskih resursa** koji
  - **omogućavaju accountig procesa** i
  - **određivanje prioritet jednog procesa**. .

u area

- **u-area sadrži sledeća polja** koja **dopunski karakterišu stanja jednog procesa**:
  - **1. ukazivač na PT**
    - koji **identifukuje ulaz** koji odgovara **u-area**
  - **2. realni i efektivni user ID**
    - koji **opisuju prava-privilegije** koje **ima proces**
  - **3. tajmersko polje:**
    - **zapisuje vreme procesa** i **njegove dece koji** su **proveli izvršavajući** se u **user i kernel modu**
  - **4. signal array:**
    - **pokazuje kako** će **proces reagovati** na **signale**
  - **5. control terminal field:**
    - identifikuje **login terminala** koji **pridružen procesu** ako to postoji
  - **6. error polje:**
    - **zapisuje greške** koji se **dogodile** u **toku SC**
  - **7. return value polje:**
    - **sadrži rezultat SC**

u area

- **6. IO parameters:**
  - opisuju **količinu podataka za transfer**, **adresu source** i **destination data array** u korisničkom prostoru, **file ofset** itd
- **7. tekući direktorijum i tekući root:**
  - koji opisuju **FS okolinu** za procese
- **8. UFDT:**
  - zapisuje **sve datoteke** koje je **proces otvorio**
- **9. limit polja:**
  - određuju **max veličinu procesa** i **datoteka** koje **proces može napraviti**
- **10. permission modes polje:**
  - **definišu inicijalna prava pristupa** za **novu datoteku**

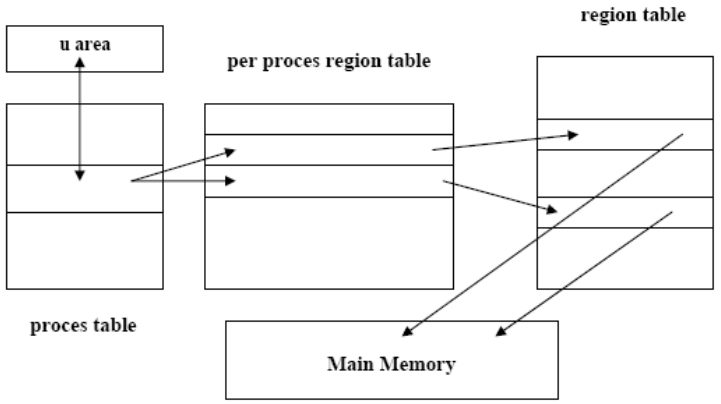
- **Svaka tranzicija ima poseban tretman** sa **fizičkom** i **virtulenom memorijom**

Layout of System Memory

- Pretpostavimo da je **fizička memorija adresibilna** sa početnom adresom **0** i **tako sve** do **ukupne količine memorije**.
- **proces** na UNIX-u se sastoji od **3 logičke sekcije**: **text, data i stack**.
  - **text sekcija**
    - **sastoji** se od **seta instrukcija** u kojima nalaze programske adrese (skokovi i pozivi procedura),
  - **data sekcija**:
    - sadrzi adrese za pristup **globalnim promenljivim**
  - **stack sekcija**:
    - adrese za pristup **lokalnim podacima u procedurama**.
- **Umesto** da se bave **fizičkim adresama**, **prevodioci** koriste princip **virtuelnih adresa**, i oni ne treba da znaju gde će kernel njihove programe napuniti za **vreme izvršavanja**.
- Takođe više **varijanti istog programa** mogu postojati u **memoriji**, svi imaju **iste virtualne** ali **različite fizičke adrese**.
- **Deo kernela i hardvera** koji obavlja **translaciju virtulenih** u **fizičke adrese** naziva se **MMU** (memory management unit-subsystem)

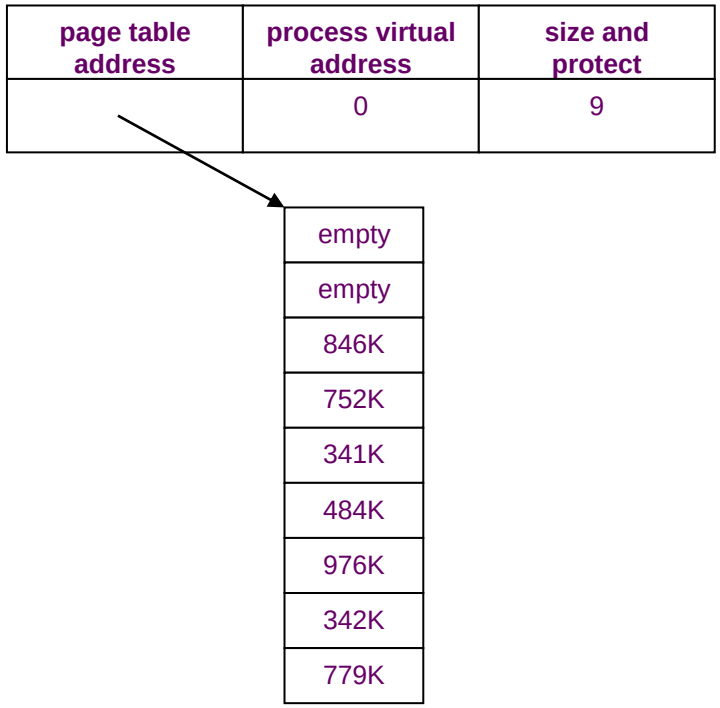
Regions

- Kernel kod UNIX System V deli **virtuelni adresni** prostor procesa u **logičke regione**.
- **Region** je **kontinualni područje virtulenih adresa** procesa koje se tretiraju kao poseban objekat koji se može **deliti** ili biti **zaštićen/private**.
- To znači da su **text, data** i **stack posebni regioni procesa**.
- **Više procesa mogu** da **dele jedan region**,
  - na primer **više procesa koji izvršavaju isti program, deliće text region,**
  - a **moguće** je da **više procesa** imaju **zajedničke podatke** pa će imati **deljivi data region**.
- **Kernel sadrži**
- **1. RT - region tabelu** (jednu i globalnu)
  - za **svaki region postoji jedan ulaz**,
  - koji treba da **definiše** njegovu **virtulenu i fizičku adresu**.
- 2. **PPRT** - **per proces region table** or **pregion**
  - za **svaki proces** postoji **posebna tabela PPRT**
  - zvaćemo je **pregion**.



pregion entry

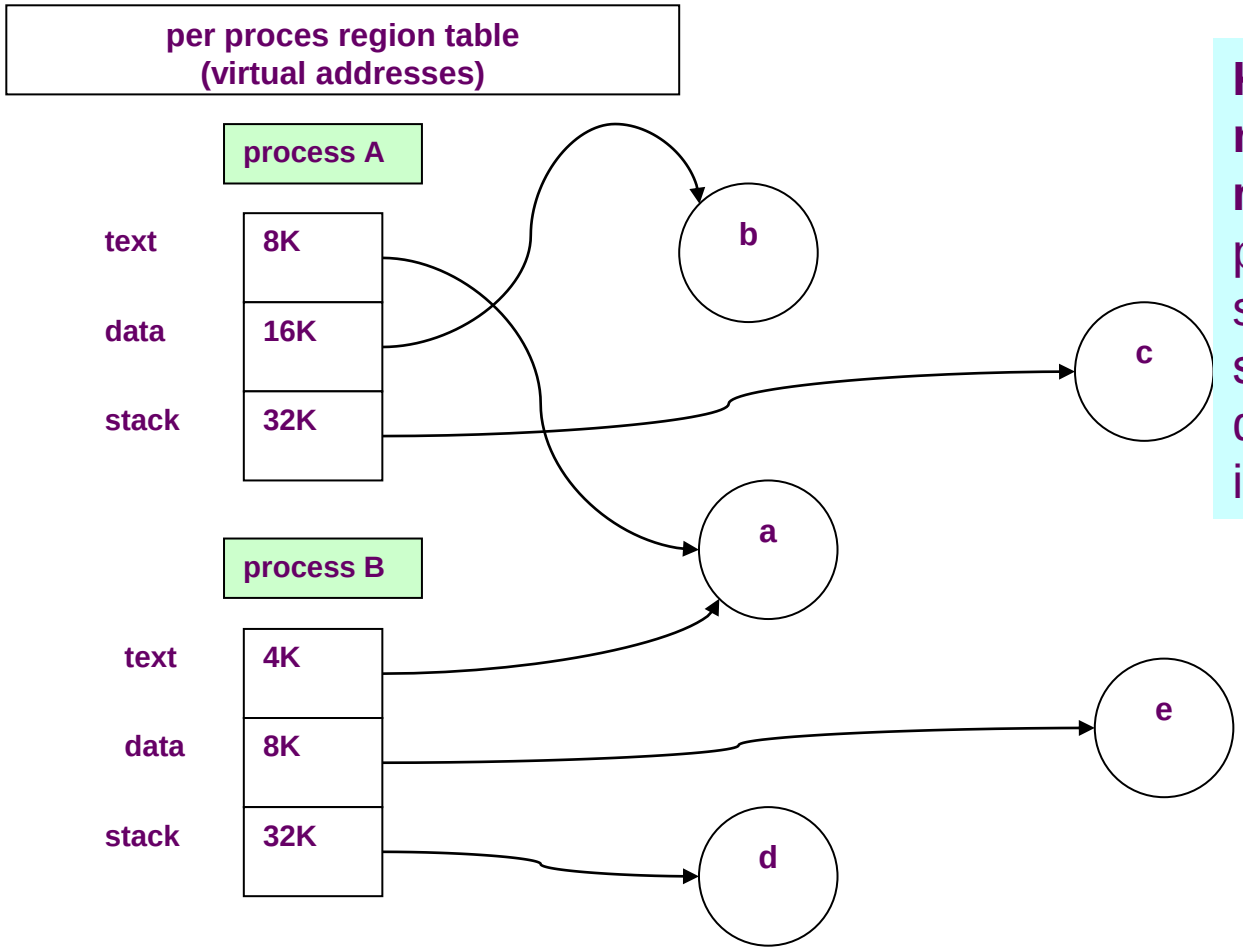
- **pregion entry** se **mogu nalaziti** u:
  - **PT**
  - **u-area**
  - **u posebnom delu memorije,**
- **zavisno** od **implementacije UNIX-a**.
- **pregion entry sadrži:**
  - **1. pointer** na **ulaz RT**
  - **2. početnu-virtuelnu adresu regiona.**
  - **3. zaštitne atribute**, kao što su
    - **read-only,**
    - **read-write,**
    - **read-execute**.



- **Deljivi regioni** mogu imati **različite virtulene adrese** za **različite procese**.
- **pregion i region** liče na **FT i inode strukturu**,
  - **više procesa mogu** da **dele regione**,
  - kao što **mogu** da **više procesa pristupaju datoteci preko FT i inode tabele**.

regions, pregion, primer

- Na sledećoj slici su data **2 procesa A i B**, sa njihovim regionima, pregionima i virtulenim adresama gde su regioni konektovani.
- **Oba procesa dele text region a**, koji za jedan proces počinje na **virtuelnoj adresi 8K** i na **virtulenoj adresi 4K**, dok su im **data** i **stack regioni privatni.**



Koncept regiona je **nezavistan** od **memorijske šeme** koju primenjuje operativni sistem, da li je paging ili swapping system, a takođe da li se primenjuje paging ili segmentacija

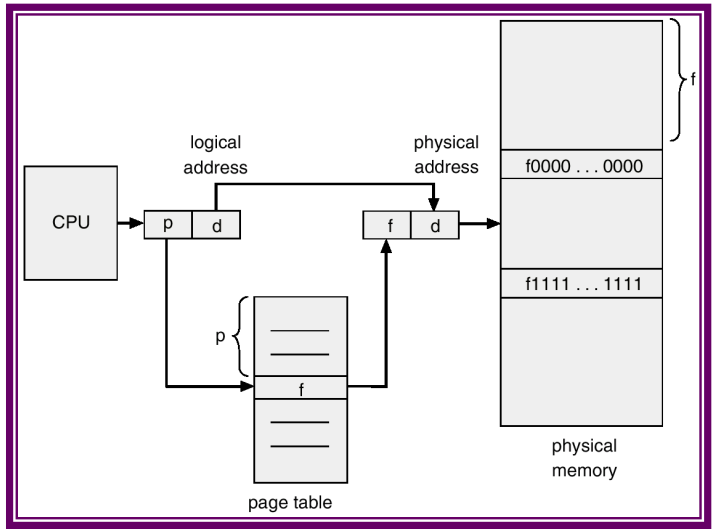
Pages and page tables

- **Kernel mapira virtulene** u **fizičke adrese** preko **tabele stranica**,
- tako što **mapira logičke stranice regiona** u **fizičke stranice**.

- Pošto je **region kontinulani opseg virtuelnih adresa**,
  - **logička stranica je index** u **tabelu stranica**
  - koja **pored mapiranja** može sadržavati i **neke zaštitne bite**,
  - a svaki ulaz u **RegionTable** sadrži **ukazivač** na **PageTable**.

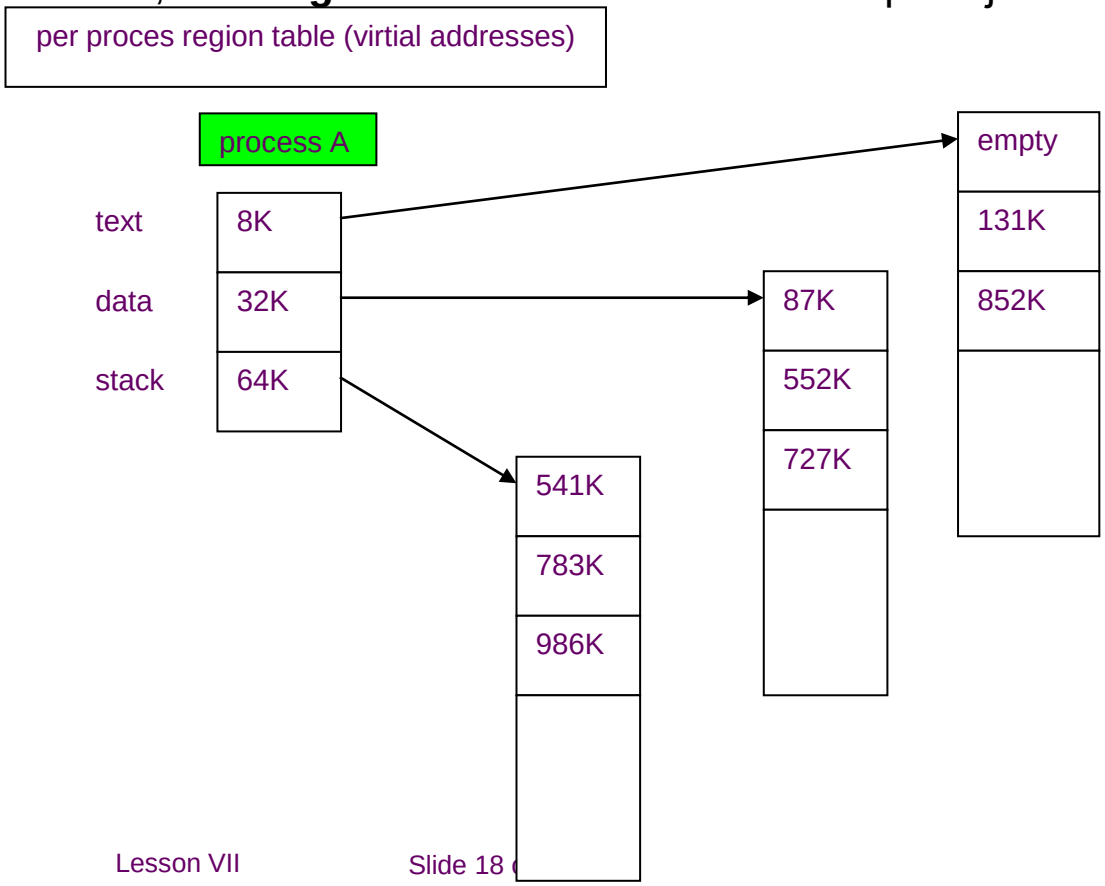
- **PageTable** je jedna od **tipičnih kernelskih data struktura**.

Logical Page Number	**Physical Page Number**
0	177
1	54
2	209
3	17



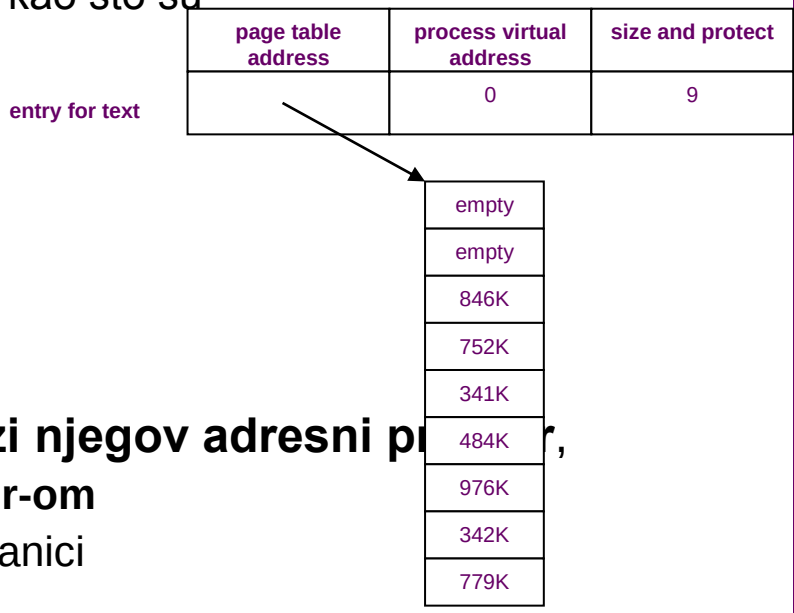
paging example

- **Pretpostavimo** da je **page size=1K**, i pretpostavimo da proces pristupa virtuelnoj adresi 68.432.
  - **Pregion ulazi** ukazuju da je to **virtulena adresa** u **stack regionu** koja počine na 64K (65536) i pretpostavimo da stack raste na gore.
  - Kada **oduzmemo adresu** 68432 -65536 dobijamo **offset 2896** unutar **regiona**.
  - Pošto imamo **1K pages**, ta adresa je **page 2** (stranice se broje od 0) sa offsetom 848 unutar stranice, a iz **PageTable** se čita da ta stranica počinje na **986K.**



register triplet

- Moderne mašine koriste različite **hardverske registre** i keš da **ubrzaju straničenje.**
- **Pretpostavimo** sledeći **memorijski model**:
- memorija je organizovana u **1K pages**
- sistem sadrži skup **tripleta memorijskih registara**, od kojih:
  - **1. prvi registar** sadrži **adresu PageTable** u **memoriji**,
  - **2. drugi registar** sadrži **početnu virtuelnu adresu** mapiranu preko ovog tripleta
  - **3. treći registar** sadrži **kontrolne informacije**, kao što su
    - **broj stranica** u **PageTable**,
    - **prava pristupa** za **stranicu**.



- Ovaj model odgovara **regionima**,
  - kada **kernel priprema proces** za **izvršavanje**
  - on puni **ove triplete** na **bazi pregion ulaza**.
- Ako **proces generiše adresu koja prevazilazi njegov adresni pr r**,
  - **hardver** mora da reaguje sa **exception handler-om**
  - ako **proces** pokušava da piše **po read-only** stranici
  - **sve** se **završava** sa **exit-om** za taj **proces**

layout of kernel

- **Kada** se **kernel izvršava** u **kontekstu procesa**, **virtulena memorija** dodeljena **kernelu** je **nezavisna** od **svih procesa**.
- **code i data** za **kernel** ostaju u **sistemu permanentno**
  - **svi procesi** je **dele**,
  - kada se sistem podiže, **kernelski kod** se puni u memoriju,
  - **postavljaju se potrebne tabele i registri** da mapiraju virtulene adrese u fizičke adrese.
- **Kernelska PageTabela** je **analogna PageTabeli** dodeljena procesima i mehanizam za adresnu translaciju je sličan kao kod **korisničkih adresa**.
- Kod mnogih mašina, **virtuleni prostor se deli na više klasa**, uključujući **sistemsku** i **korisničku klasu**, a svaka ima sopstvenu **PageTable**.
- Kada se **proces prebaci** u **kernel mod**,
  - dozvoljava mu se pristup **kernelskim adresama**,
  - ako je u korisničkom modu blokira mu se pristup kernelskim adresama,
  - što se postiže tako što **OS sarađuje sa hardverom** ili se **puni specijalni registri u CPU**
  - (podsetimo, proces prelazi u kernelski mod samo **preko SC** ili ako se dogodi **prekid**)

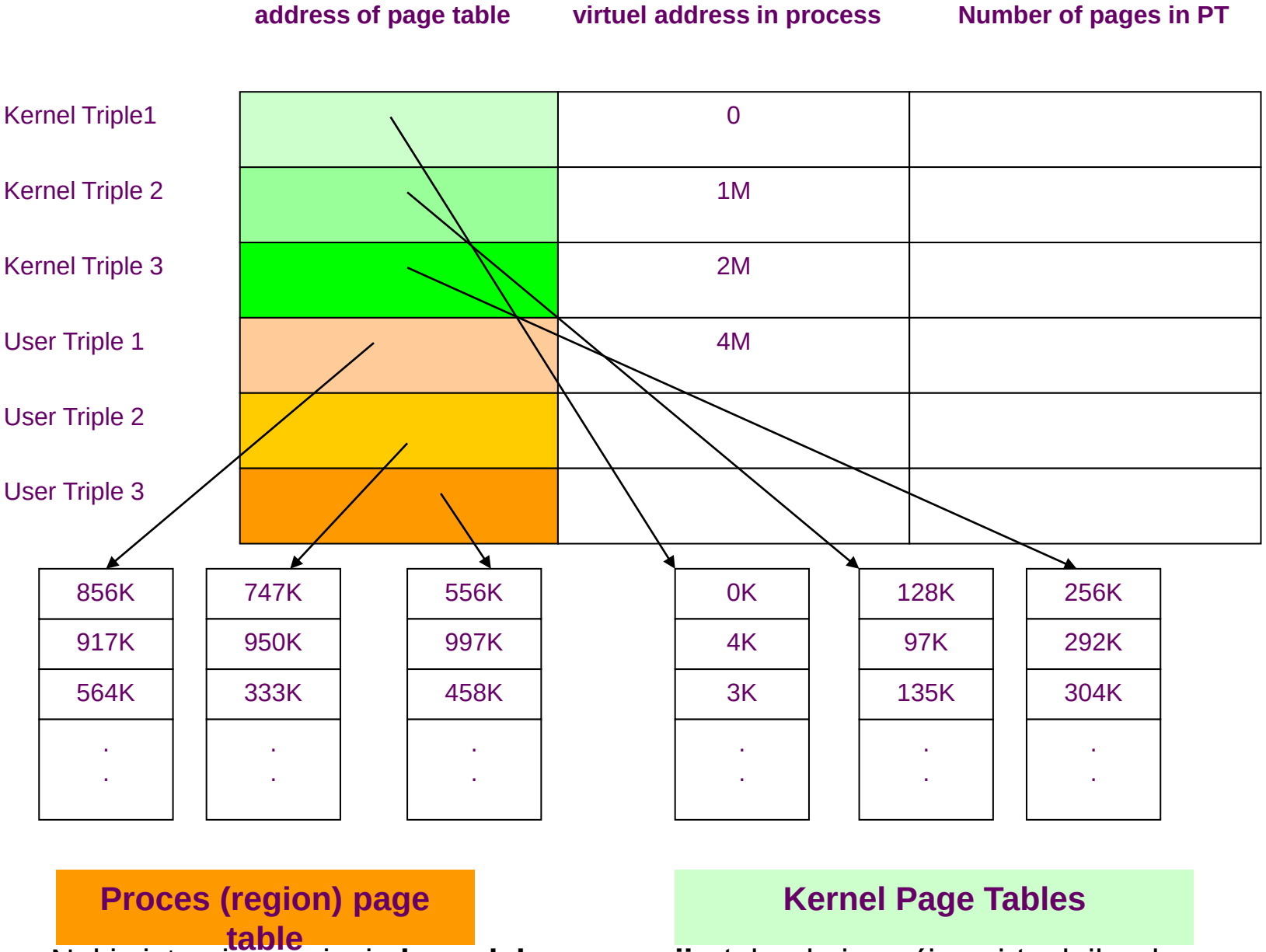
Layout of kernel

- Evo **jednog primera na slici**,
- gde je **virtuleni prostor kernela od 0-4M-1**
- a **korisnički prostor preko 4M**.

- Postoje **2 skupa registarskih tripleta**,
- od koji **prvi skup definiše kernelske adrese**, a **drugi korisničke triplete**.

- **Svaki triplet ukazuje** na **PageTable** koja definiše **vezu između virtulenih** i **fizičkih adresa.**

- Naravno **proces** će **moći da pristupi kernelskim adresama** i **kernelskim tripletima samo u kernelskom modu.**



- Neki sistemi organizuju **kernelsku memoriju** tako da je većina virtuelnih adresa identična sa svojim **fizičkim**, ali u-area zahteva virutuelno mapiranje i za kernel.

u area

- **Svaki proces** ima privatnu **u-area**,
  - ali kernel joj pristupa samo ako ima **jedna jedina u-area** u sistemu,
  - i **to od procesa koji se izvršava**.
- Kernel menja svoju mapu za adresnu translaciju u **saglasnosti sa procesom** koji se **izvršava pristupajući u-area tog procesa**.
- Kada se prevodi OS, punilac-loader dodeljuje **promeljivu u**, na **fiksnu virtulenu adresu.**
- **Vrednost ove promenljive** i njena adresa su poznati ostalim delovima kernela, a posebno je to značajno za modul koji obavlja **kontext switch**.
- Adresa **u-area** je različita **za svaki proces**, ali **kernel** joj pristupa **preko iste virtuelne adrese**.
- **Svi procesi** pristupaju svojoj **u-arei** kada su **kernelskom modu**,
- nikako kada su u user modu.
- **Kernel** preko **svoje virtulene adrese** može pristupati **samo jednoj u-area** i to **onoj** od **aktivnog procesa.**

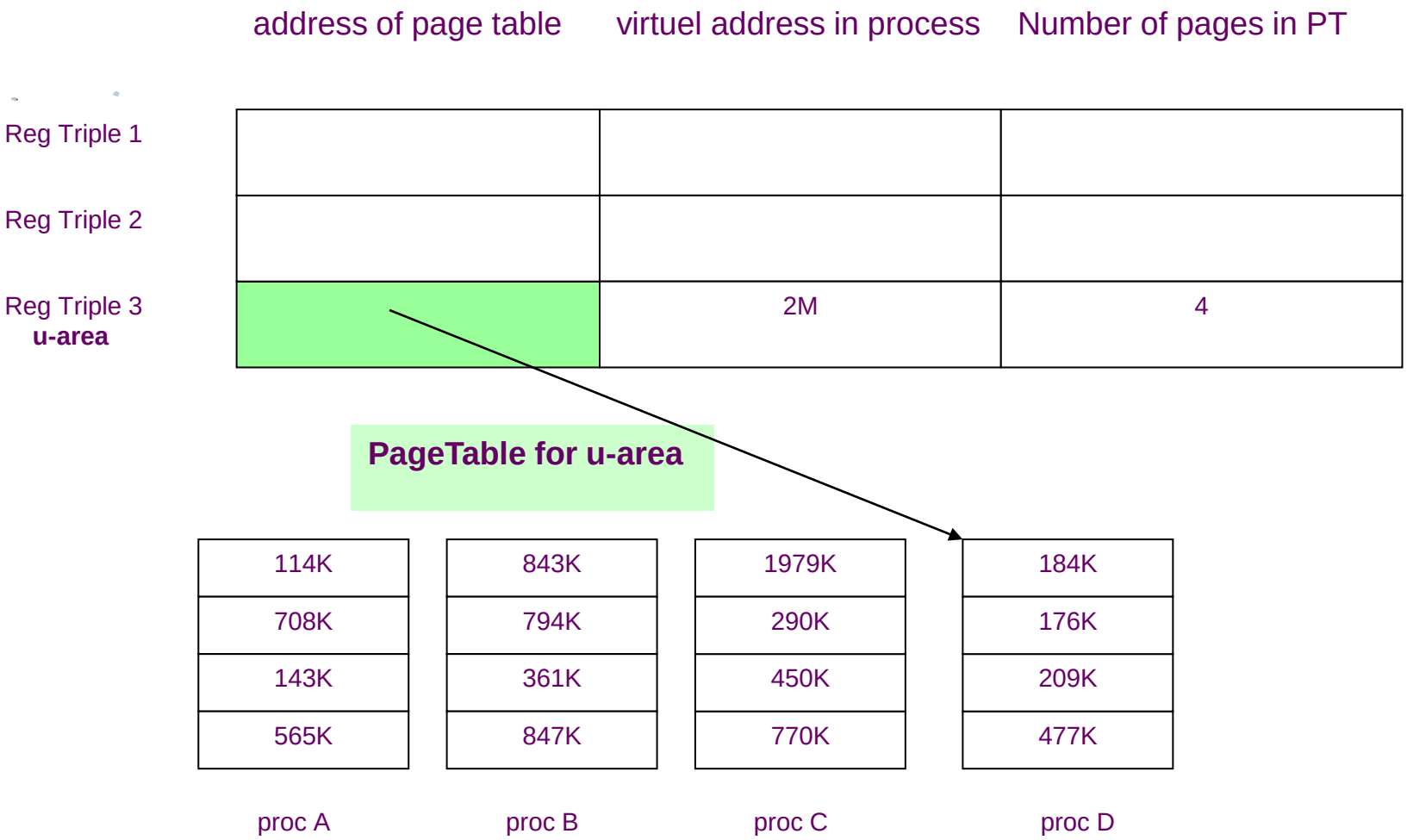
u area - example

- Na primer prepostavimo da je
- **u-area 4K veličine** i
- nalazi se na **kernelskoj virtulenoj memoriji 2M**, kao na slici.

- Za kernel su prikazana **3 tripleta**,
  - **prvi triplet** je **kod kernel-a**,
  - **2-gi triplet** je **data kernel-a**, a
  - **treći** je **u-area pointer**, i trenutno ukazuje na **u-area procesa D**.

- Ako bi kernel želeo da pristupa **u-area procesa A,** on kopira odgovarjuću informaciju o **PageTable** za u-area procesa A u svoj treći registarski triplet.
- Naravno, uvek **treći triplet** mora da ukazuje na **u-area aktivnog procesa.**
- **Prilikom CSw ne menjaju se prva 2 tripleta kernela** zato sto svi procesi dele **kernelski code i data.**

u area - example



Context of a process

- **Kontekst procesa sastoji** od
  - **sadržine** njegovog **adresnog prostora**
  - **sadržine** hardverskih **registara CPU**
  - **kernelskih data struktura** koje **pripadaju procesu**.
- Formalno, **kontekst procesa** je **unija** njegovih:
  - **user-level konteksta – memory context**
  - **registarskog konteksta**
  - **system-level konteksta**.

user-level context – memory context

- **User-level kontekst** sastoji se:
  - **text**
  - **data**
  - **stack**
  - **shared memorije** koja **okupira virtuleni adresni prostor procesa**

- **Delovi virtuelnog** adresnog prostora **procesa mogu biti**
  - delimično u memoriji
  - delimično na swapu

register-level context

- **Registarski kontekst sastoji** se od **sledećih komponenti**:

- **PC (program counter)** specificira **adresu sledeće instrukcje** koju će CPU izvršavati (adresa je **virtulena** u kernelskom ili korisničkom prostoru)

- **PSW (processor status register)** sadrži **status hardvera**.
  - Po pravilu delovi registra sadrže informacije vezane za **zadnju izvršenu CPU instrukciju**, tipa da li je rezultat **pozitivan, ili negativan**.
  - Drugi delovi registra mogu ukazivati na **prekoračenje** (**carry flag**).
  - Veoma bitna infomacija je **u kom režimu** se izvršva proces (**kernel mode, user mode**), što govori da li proces može izvršavati privilegovane instrukcije i da li može pristupati **kernelovim data stukturama**.

- **SP (stack pointer)** sadrži tekuću adresu **sledećeg ulaza** u **korisničkom** ili **kernelskom stack-u**. Naravno, CPU arhitektura diktira da li SP ukazuje na prvu prvu free lokaciju na stacku ili na zadnju zauzetu, kao i smer u kome **stack raste na gore i na dole.**

- **GPRs (general-purpose registers)** sadrži podatke koji se generišu u procesu za vreme izvršavanja

System-level context-static part

- **System-level context** procesa ima:
  - statički deo
  - dinamički deo
  - pri čemu proces ima **jedan statički deo** za vreme svog izvršavanja,
  - dok može imati **više dinamičkih delova**.
- **Dinamički deo system-level** konteksta je
  - **stack of contex leyers** (stek kontekst lejera)
  - koje **kernel gura** i **skida sa steka**
  - na bazi različitih događaja.
- **Static part** System-level kontekst **sastoji** od **sledećih komponenti**:
  - **PT entry** procesa **definiše stanje procesa** (sekcija 6.1) i sadrži kontrolne informacije koji kernel uvek može pristupiti
  - **u-area procesa** sadrži **kontrolne informacije koje su jedino potrebne** u **kontekstu procesa**. **Generalne informacije** kao što su **prioritet procesa** se čuvaju u **proces tabeli,** pošto se njima pristupa izvan konteksta procesa
  - **Pregion ulazi, region tabele i tabele stranica**,
    - definišu **mapiranje** između **virtuelnih i fizičkih adresa** i zato definišu **text, data i stack regione procesa**.
    - **Ako više procesa dele iste regione**, ti regioni su sastavni delovi konteksta **svakog procesa**, zato što svaki proces **manipuliše regionom nezavisno**.
    - Takođe, deo memorijskog upravljanja predstavlja task **koji označava koji deo adresnog prostora nije u memoriji, odnosno nalazi** i na **swap.**

System-level context-dynamic part

- **Kernelski stack** sadrži **stack-frames (okvire) kernelskih procedura** kada **se proces izvršava** u **kernelskom modu**.
- Mada svi procesi izvršavaju identičan **kernelski kod**, svi imaju **privatne kopije kernelskog stack**, koji opisuje **specifično pozivanje kernelskih funkcija**.
  - Na primer, **jedan proces** može pozvati **creat SC** i otići na spavanje dok kernel ne dodeli novi inode za njega, a **drugi proces** može pozvati **read SC** i otići na spavanje dok se obavi transfer podataka.
  - **Oba procesa izvršavaju kernelske funkcije** ali imaju **posebne kernel stacks**.
- **Kernel mora da bude sposoban da obnovi sadržaj kernelskog** steka i poziciju **SP registra** da bi nastavio izvršavanje procesa u **kernelskom modu**.
- Mnogi sistemi plasiraju **kernelski stek u u-area**, mada može da postoji i kao **nezavisna celina** u **memoriji**.
- **Kernelski stek** se prazni kada se proces vrati u **user mod**.
- **Layers**: **Dinamički deo system-level konteksta procesa** se **sastoji** od **skupa layera**, koji rade na **principu LIFO stacka**.
- Svaki **system-level kontekst layer sadrži informacije** da **rekonstruiše prethodni sloj**, uključujući **registerski kontekst prethodnog sloja**.

System-level context-dynamic part

- **push:** kernel **gura** kontekst layer na **stack**
  - kada se **dogodi prekid**
  - kada proces **obavi SC**
  - kada se obavi **kontekst switch**

- **pop:** Kernel **skida** sa steka konteks layer
  - kada se obavlja **povratak iz obrade prekida**
  - kada se **završi SC**
  - kada se obavlja **kontekst switch**.

- **CSW:Kontekts switch** obuhvata **push i pop system-level kontekst layera**:
  - **kernel gura** na **stek layer starog procesa**
  - **skida** sa **steka layer novog procesa**.

- **PT ulaz memoriše informacije neophodne** da se **rekonstruiše tekući kontekst layer**.

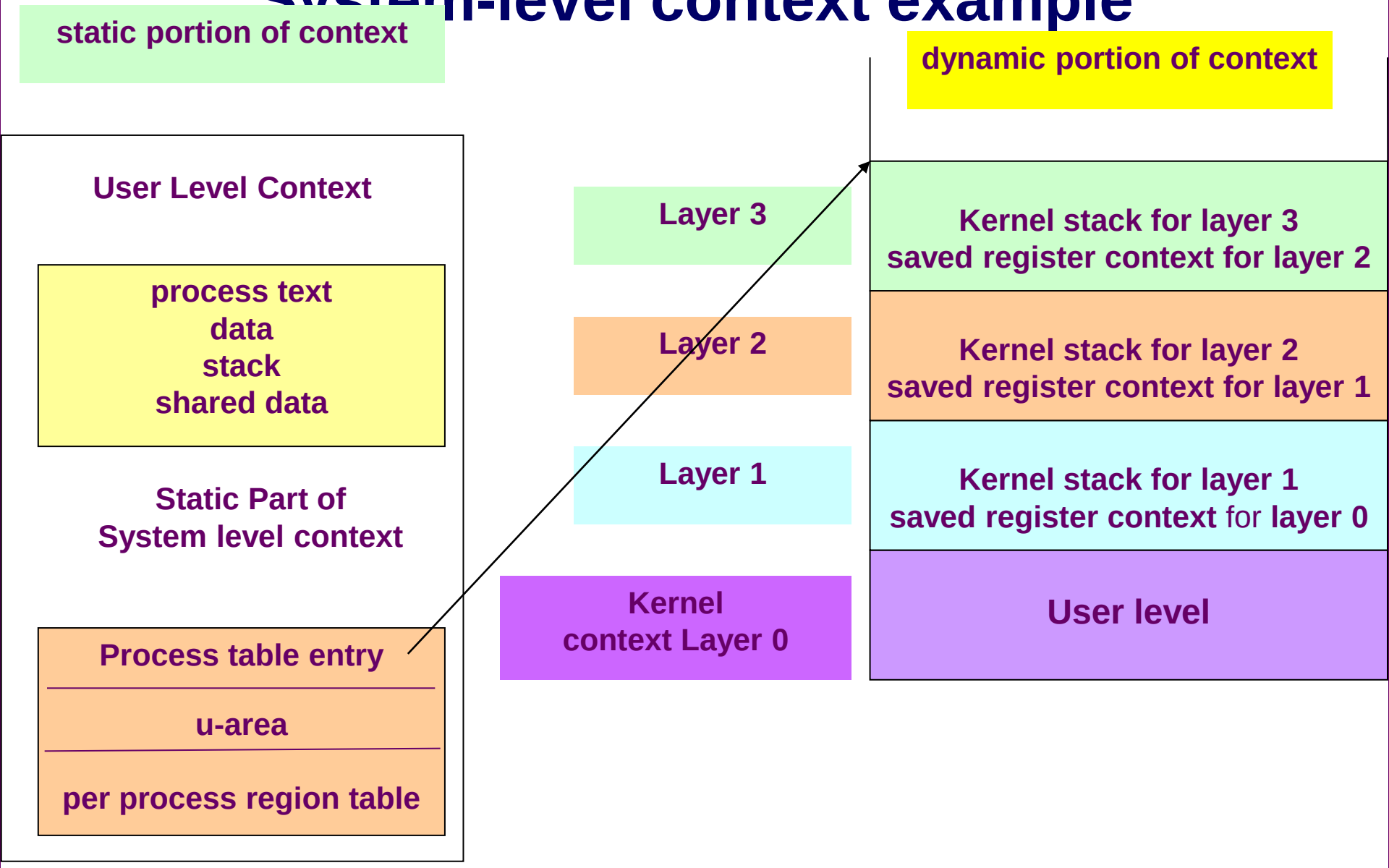
System-level context example

- Na sledećoj slici su **prikazane komponente konteksta procesa**.

- **Leva strana slike sadrži statičku porciju** konteksta, koja se sastoji od
  - **user-level konteksta** (text, data, stack, shared memory),
  - **statičkog nivoa systemskog konteksta** (PT, u-area, pregion ulazi).

- **Na desnoj strani** nalazi se **dinamički deo konteksta**,
  - koja se sastoji od više **stack okvira**,
  - gde svaki okvir sadrži **sačuvani registarski kontekst** prethodnog nivoa
  - to je **kernelski stack**.
  - **Nivo 0 predstavlja user-level kontekst**,
  - stack je ovde u korisničkom prostoru, kernelski stack je ovde null.

System-level context example

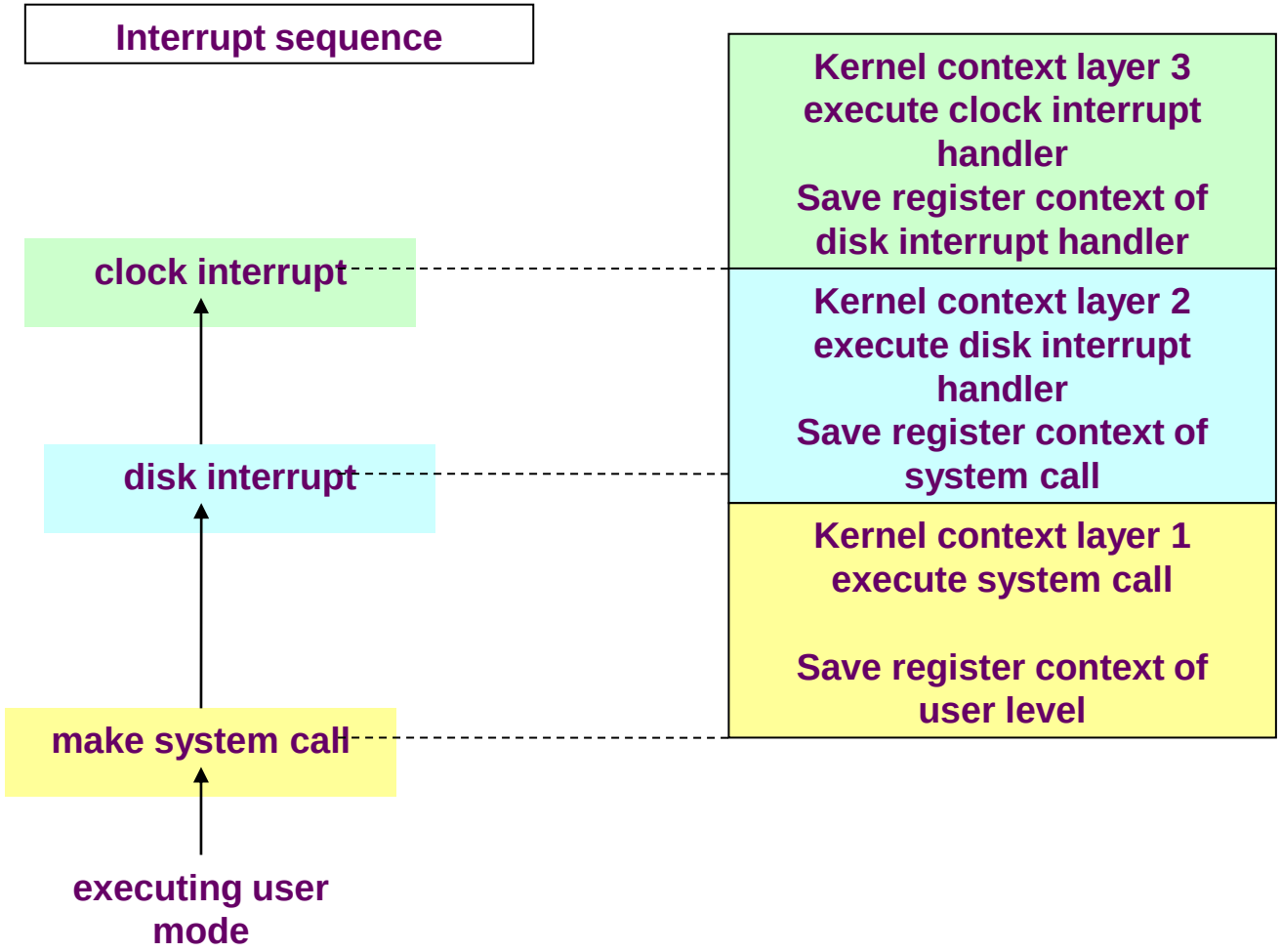


Saving the context of a process

- **kernel** čuva **kontekst procesa**
  - **by push** of novi system context layer,
  - a to se **dešava uvek**
    - kada se **dogodi prekid**
    - kada proces **izvršava SC** ili
    - kada kernel obavlja **context switch.**

Interrupt - example

- Na sledećoj slici **prikazaćemo situaciju** kada **proces obavi SC**, a dogodi se **disk prekid**, a tada se dogodi **clock prekidni signal**. Svaki put kada sistem **primi prekidni signal** ili **obavi SC**, kreira se novi **kontext layer** i čuva se registarski kontekst **prethodnog layera**.



Context switch

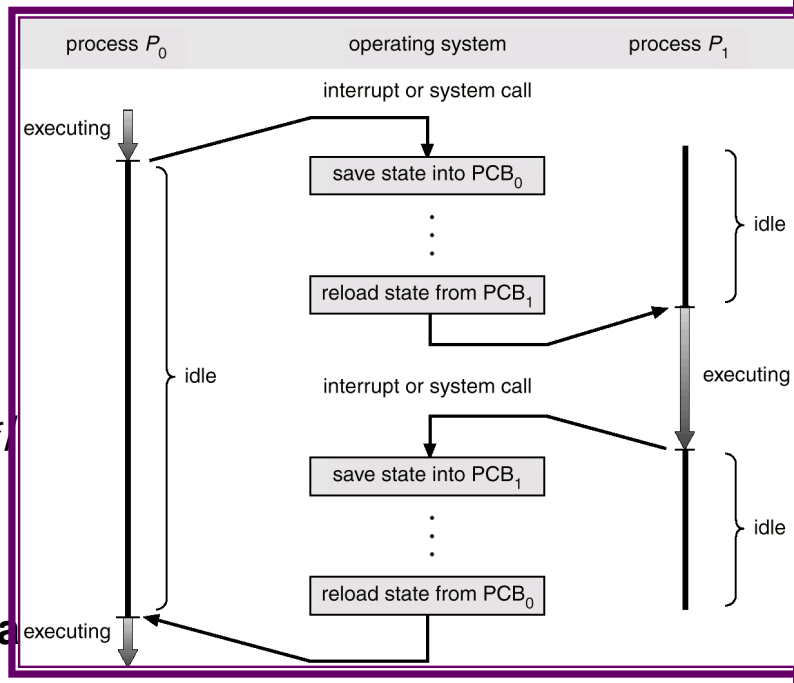
- **kernel omogućava context switch** pod **4 uslova**:
  - **1. kada proces** samog **sebe uspava**
  - **2. kada proces završava aktivnosti**, obavlja **exit**
  - **3. kada se vrati iz SC(kernel mode**) u user mode, ali nije najpogodniji proces koji bi nastavio rad (ima prioritetnijih)
  - **4. kada se vrati iz prekida u user mod**, ali nije najpogodniji proces koji bi nastavio rad (ima prioritetnijih)

- Kernel čuva **integtritet** svojih struktura
- **sprečavajući proizvoljan CSw**
- **a mora obezbediti** da **stanje podataka bude konzistetno pre nego što odobri CSw**:
  - na primer da li su podaci **ažurirani,**
  - da li su redovi čekanja povezani **korektno**,
  - da li su **lock-ovi postavljeni** ili **skinuti korektno**.

- **Procedura** za **CSw** je slična kao **interrupt handling** ili **SC**,
  - sa **izuzetkom** što **kernel obnavlja kontekst layer drugog procesa**
  - umesto sa obnavlja **prethodni kontekst** istog procesa.

scenario (pseudo-code) za CSw

- if (**save_context**()) /*save context of executing process*/
- {
- /* **pick another process to run**/
- ….
- ….
- **resume_context** (**new_process**);
- /* never gets here! */
- }
- /* **ressuming process executes from here**/



- Funkcija **save_context**
  - **obavlja čuvanje konteksta tekućeg procesa**
  - ako uspe vraća vrednost 1.
  - Između brojnih vrednosti **kernel čuva vrednost PC** na **lokaciji 0** i
  - to se koristi kao **povratna vrednost** iz funkcije **save_context()**.

scenario (pseudo-code) za CSw

- **Primer CSW(A)->B**

- **Kernel**
  - nastavlja da izvršava **kontext** starog procesa A,
  - tako što bira **novi proces B** i
  - poziva **resume_context**
  - koja će **obnoviti kontekst novog procesa B.**

- Kada se **obnovi njegov kontekst**, **CPU izvršava proces B**,
  - dok **proces A** ostaje u svom **sačuvanom kontekstu**.
  - njega će aktivirati **neki drugi proces**,
  - koji će **obaviti CSw** i
  - **izabrati njega**, a **on će nastaviti** u **/*ressuming ....*/**

Saving context for abortive returns

- **Postoje situacije** kada **kernel mora prekinuti tekuću sekvencu** i neposredno **izvršiti prethodno sačuvani kontekst**.
- U sekcijama ove glave obradićemo situacije kada se **sleeping** ili **signali** nateraju proces da **iznenada promeni svoj kontekst**.

- **Algoritam koji čuva kontekst** je **setjmp**,
- dok **algoritam koji obnavlja kontekst** je **longjmp**.

- **Metodi** su **identični** kao u funkciji **save_context**,
  - **izuzev** što **save_context** gura novi **context layer**,
  - dok **setjmp memoriše sačuvani kontekst** u **u-aera** i
  - i nastavlja da izvršava **stari kontekst layer.**

- Kada kernel obnavlja kontekst koga je sačuvao **setjmp**,
- on obavi **longjmp** koji obnavlja kontekst iz **u-area.**

Manipulation of the process address space

- Do sada nismo uključivali **priču** o **virtulenoj memoriji i regionima u okviru SC** i **CSw**, međutim to je jako bitno **jer** se **memorija procesa može dinamički menjati.**

- **RT ulaz** sadrži **sledeće informacije** koje **opisuju region**:
  - **1. ukazivač** na **inode datoteke** koja je **napunjena** u **region**
  - **2. tip regiona** (text, shared memory, private data, stack)
  - **3. veličina regiona**
  - **4. lokacija regiona** u fizičkoj memoriji
  - **5. status regiona** koji može biti kombinacija:
    - **locked**: zaključan
    - **in demand**: traži se
    - **in process** of **being loaded into memory**
    - **valid** (loaded into memory)
  - **6. RC**, koj predstavlja **broj procesa** koji **imaju referencu na taj region**

Manipulation of the process address space

- **Operacije** koje deluju na **region** su:
  - **lock**
  - **unlock**
  - **allocate**
  - **attach** **regiona u virtuleni memorijski prostor procesa**
  - **change size**
  - **load a region from file**
  - **free** **oslobađanje fizičke memorije regiona**
  - **detach** **regiona iz virtuelnog memorijskog prostora procesa**
  - **duplikacija sadržine regiona**

- Na primer kada se izvršava **exec SC**,
  - koji prepisuje korisnički adresni prostor sa sadržajem egzekutabilne datoteke,
  - **detach-uju** se stari regioni,
  - **oslobađa** se adresni prostor procesa osim ako nisu deljivi,
  - **alociraju se novi regioni**
  - attach-uju se regioni i pune se sa sadržajem datoteke.

- Opisaćemo sve region operacije u detalje.

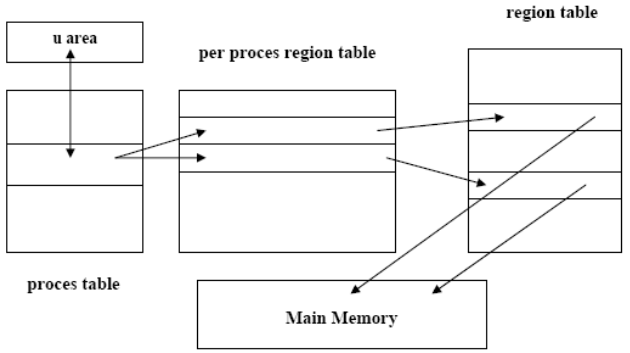
locking and unlocking region

- Kernel ima mogućnost
  - da obavi **lock i unlock** regiona
  - a **da ga ne oslobađa** (kao kod inoda iget i iput).

- Kernel **lockuje region** da bi **sprečio druge procese da manipulišu** a njim,
- a potom ga oslobodja sa unlock

Allocating a Region (RT entry)

- Kernel alocira novi region (alogoritam **allocreg**) za vreme SC
  - **fork**
  - **exec**
  - **shmget** (shared memory).
- Kernel sadrži RT čiji ulazi se nalaze (formiraju) u
  - linkovanoj slobodnoj listi ili
  - u aktivnoj linkovanoj listi.

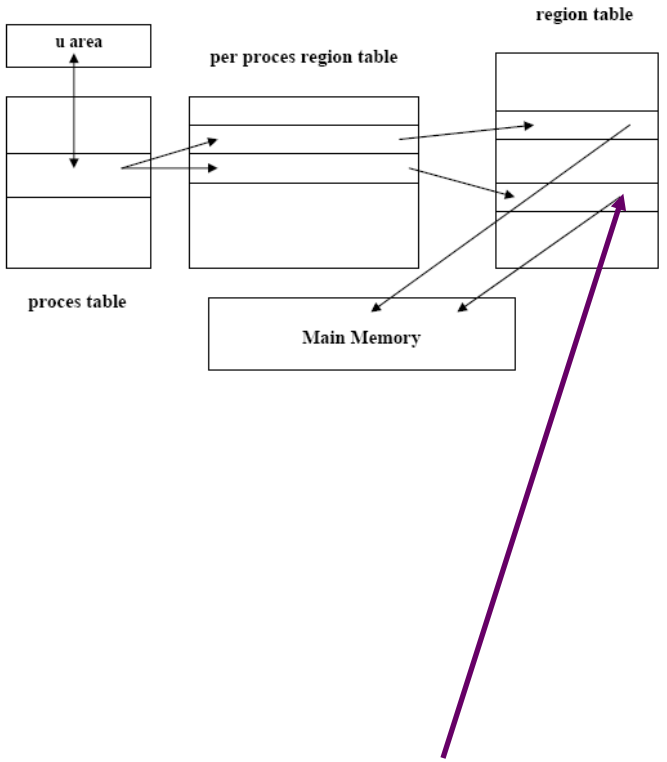


- Kada se alocira RT ulaz,
  - kernel **uklanja prvi raspoloživi ulaz** iz **slobodne liste**,
  - stavlja ga u **aktivnu listu,**
  - **lock-uje taj region** i
  - **markira njegov tip** (shared, private).

- Uz par izuzetaka, **svaki proces** se **udružuje sa egzekutabilnom datotekom** kao rezultat **exec SC** koji poziva **allocreg** koji setuje **inode** polje u **RT ulazu** da **ukazuje** na **napunjenu datoteku**.
- **Inode identifukuje region u kernelu**, tako **drugi procesi mogu** da **dele** taj **region**. **Za svaki proces** koji **deli region**, **kernel će inkrementirati RC** i **sprečiti** da se **region** oslobodi **sve dok ga procesi koriste**.
- **Allocreg** vraća **locked alocirani region**.

algortithm allocreg

- **algortithm allocreg /* allocate a region data structure*/**
- **input:**
  - (1) inode pointer
  - (2) region type
- **output:**
  - **locked region**
- {
- **remove** region from **linked list** of **free regions**;
- **assign** region **type**;
-
- **assign** region **inode pointer**;
- if (inode pointer **not null**) **increment inode RC**;
- **place** region on linked list of **active region**;
- **return(locked region)**;
- }

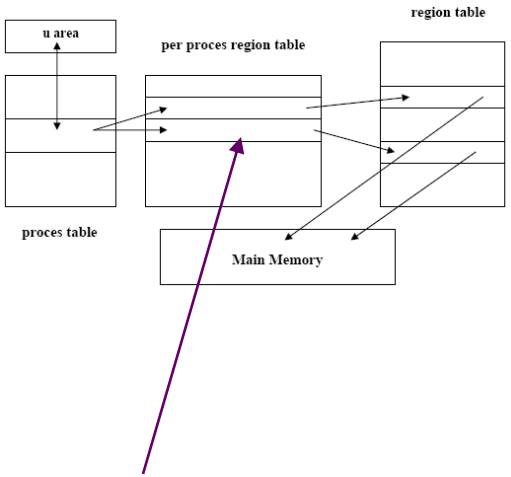


attaching a region to a process (pregion entry)

- Kernel **attach**-uje region (algoritam attachreg) za vreme SC
  - **fork**
  - **exec**
  - **shmget** (**shared memory**) SC,
- pri čemu se **region** konektuje na **adresni prostor procesa**.
- Region može biti **novo alocirani region** ili **neki od postojećih regiona** koga proces deli sa drugim procesima.
  - **Kernel alocira** slobodan **pregion ulaz**,
  - **setuje type polje** na **odgovarajuću vrednost** (text, data, shared memory, stack) i
  - upisuje **virtuelnu adresu** na poziciji gde će se region locirati u adresnom prostoru procesa.
- Proces **ne sme da prevazidje limite** za **najvišu virtulenu adresu** i
  - virtuelne adrese novog regiona ne smeju da se preklapaju sa postojećim regionima procesa.
  - Na primer, ako proces ima najvišu virtulenu adresu od 8MB, ne može mu se attach-ovati region od 1M na virtuelnu adresu od 7.5MB
- Ako je **sve legalno prilikom attaching**-a regiona,
  - kernel **inkrementira veličinu procesa u PT ulazu** saglasno sa **new-attached regionom**
  - **povećava RC za region**.

algortithm attachreg

- **algortithm attachreg /* allocate a region data structure*/**
  - **input:** (1) **pointer to (locked) region** is being attached
  - (2) **process** to which region is **being attached**
  - (3) **virtuel address** in process where region will be attached
  - (4) **region type**
  - **output:** per process region table entry = **pregion entry**
- {
- **allocate** per process region table **entry** for process;
- **initialize** per process region table **entry**;
- **set pointer** to **region** to be **attached**;
- **set type field**;
- **set virtual address field**;
- **check legality** of **virtual address**, **region size**;
- **increment region Reference Count**;
- **increment process** size according to **attached region**;
- **initialize new hardware register triple** for **process**;
- **return**(per process region table **entry**)
- }



algortithm attachreg

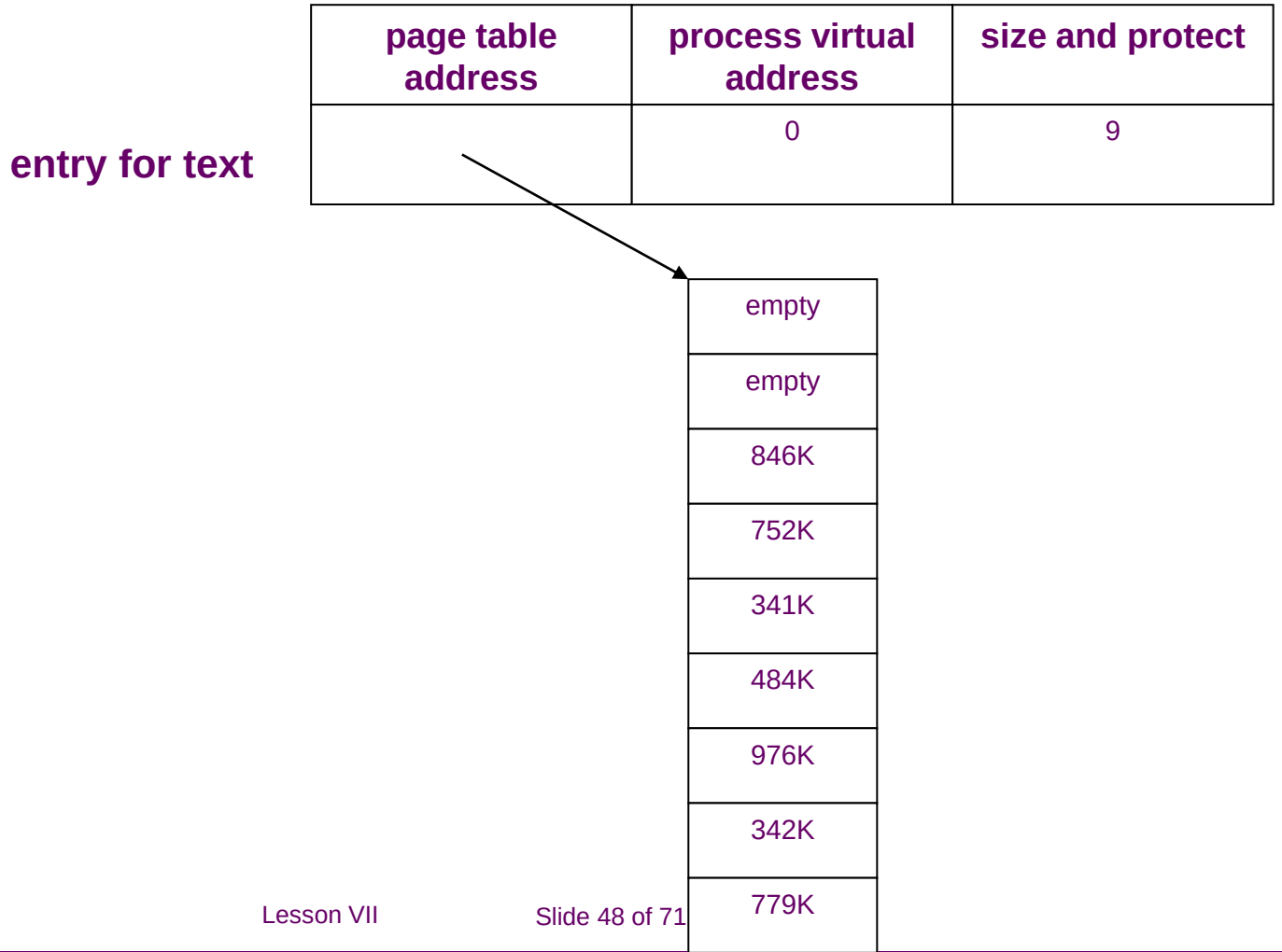
- **attachreg** inicijalizuje **novi** skup **memory management register triplet** za proces:
- **Postoje 2 situacije za triplet**
- **1. region is non-attached**
- **ako region nije** već **attached** za **drugi proces**,
  - **kernel alocira new page tabele** za **region**
  - **koje će se kasnije popuniti** u **growreg SC**.

- **2. region is attached, yet**
- ako je region već **attached koriste** se **postojeće page tabele.**

- Na kraju, **attachereg** vraća **pointer** na **pregion entry** za **new-attached region**.

algortithm attachreg

- **Na primer** pretpostavimo da **kernel želi da attach-uje postojeći shared text** region **od 7K na virtuelnu adresu 0: kernel alocira novi triplet i inicijalizuje triplet** sa **adresom page tabela za region**, **procesovom virtuelnom adresom 0** i **veličinom PT=9 ulaza**, kao na slici



changing a size of region

- **Proces**
- može **proširiti ili smanjiti svoj virtuleni adresni** prostor
- sa **sbrk SC**.

- Slično, **stack procesa se automaski proširuje** ako se **poveća dubina** ugnježdenih poziva procedura.

- **Interno**, kernel poziva algoritam **growreg** koji će promeniti veličinu regiona.

- **Prilikom ekspanzije** regiona kernel mora da obezbedi
  - da se virtuelne adrese proširenog regiona **ne preklapaju sa drugim regionima**
  - da se ne prokorači maksimalna dozvoljena **virtuelna adresa procesa**.

- **Kernel nikada neće** menjati **veličinu shared regiona** koga deli više procesa.

- Algoritam **growreg** se koristi u **2 slučaja**
  - **sbrk** na **data region**
  - **automatsko povećanje** user **stack** regiona

algorithm growreg

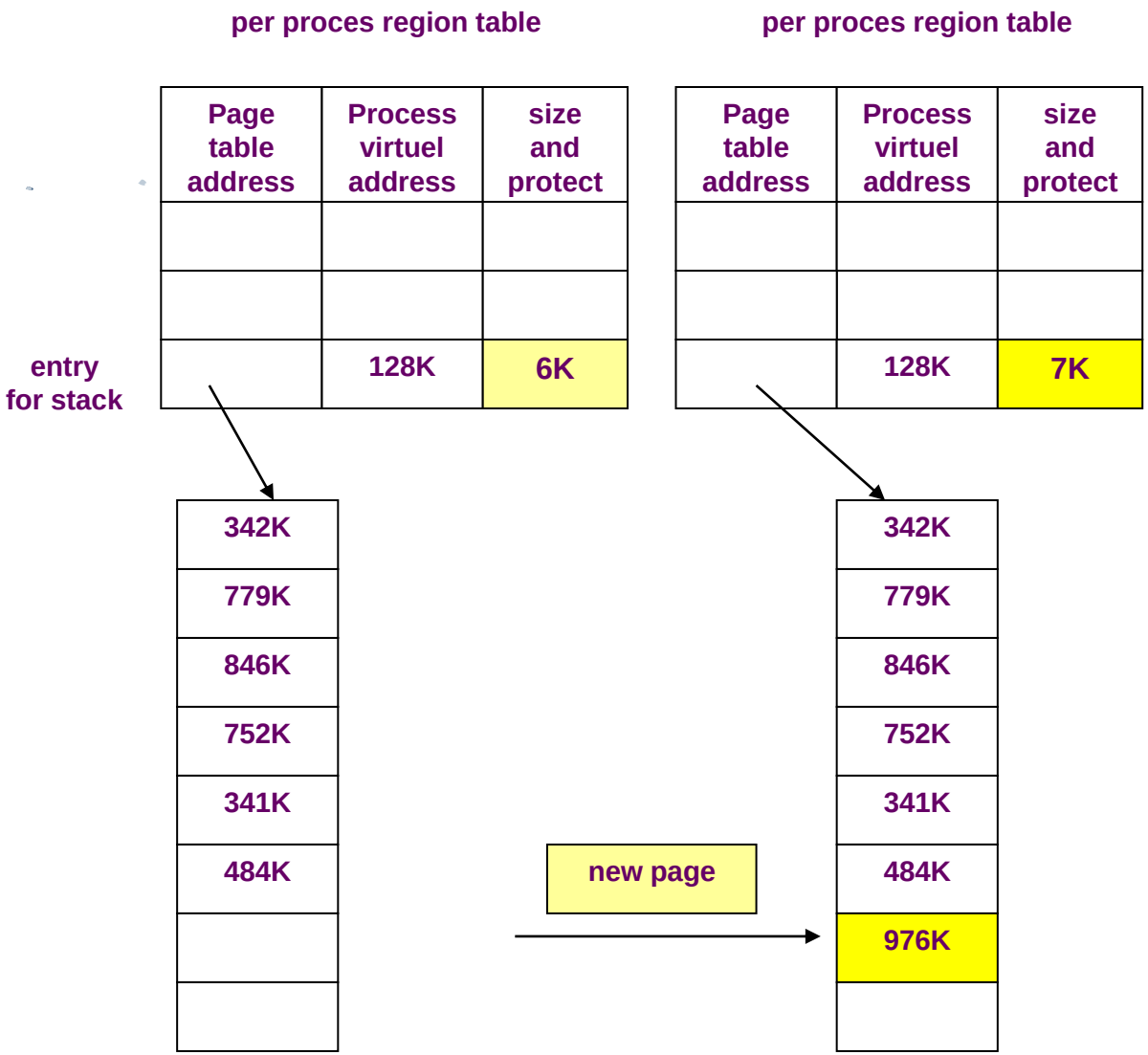
- **algortithm growreg /* change size of a region */**
- input:
  - (1) **pointer** to per proces region table **entry**
  - (2) **change in size** of region (+ or -)
- output: none
- {
- if(**region** size **increasing**)
- {
- **check legality** of **new region size**;
- **allocate auxilliary page tables**;
-
- **if(not system** supporting **demand paging**)
- {
- **allocate physical memory**;
- **initialize auxiliary tables**, as necessary;
- }
- }

algorithm growreg

- else /* region size **decreasing***/
- {
- **free physical memory**;
- **free auxiliary tables**;
- }

- **do** (other) **initialize auxiliary tables**, as necessary;
- **set size field** of **process table**;
- }

Example for algorithm growreg



loading a region

- U sistemu **koji podržavaju DP**, kernel može **mapirati datoteku u procesov adresni prostor** za vreme **exec** SC, organizujući pristup fizičkim stranicama **kasnije** na **DP bazi**.
- Ako sistem **ne podržava DP**, **mora se kopirati executable-file** u **fizičku memoriju**.
- **Može se** attach-ovati region na različitim virtulenim adresama u odnosu na one gde je napunjena exe datoteka, **praveći rupe u page tabeli**. Ova osobina se koristi da se napravi memory faults kada korisnički program pristupa adresi 0 ilegalno??. **Gap se uvek ostavlja**.
- **Da bi** se **napunila datoteka** u **region** korisiti se **loadreg**, koji obračunava **gap** između **virtulene adrese regiona** i **početne virtuelne adrese** podataka regiona i onda sledi podešavanje ??'. Tada se region postavlja u stanje „**being loaded into memory**" i puni region iz datoteke preko **read** algoritma.
- Ako kernel koristi **punjenje text regiona koji će deliti više procesa**,
  - **moguće** je da **drugi proces pokuša** pristup regionu **koji nije još napunjen**,
  - zato što **prvo proces spava dok se čita datoteka**,.
  - **Ovde ne može da se primeni lock zbog sintakse exec SC**.
  - **Kernel ovo razrešava** tako **što proveri** da li region **napunjen** i
  - **ako nije proces koji ga traži ide na spavanje**,
  - a **probudiće** ga **prekidni signal**.

algortithm loadreg

- **algortithm loadreg /* load** a **portion** of **file into** a **region** */
  - input:
    - (1) **pointer to per process** region table **entry**
    - (2) **virtuel address** to load region
    - (3) **inode pointer** of file for loading region
    - (4) **byte offset** in file for start of region
    - (5) **byte count** for amount of data loaded
  - output: none
- {
- **increase region size according** to **eventual size of region** (algorithm **growreg**);
- **mark** region **state**: **being loaded into memory**;
- **unlock region**;
- setup u -area parameters for reading file;
- target virtual address where data is **read to**
- start offset value for reading file
- count of byte to read from file
- **read file into region** (algorithm **read**)
- **lock region**;
- **mark** region **state: completely loaded into memory**;
- **awaken all process** waiting for **region to be loaded**;
- }

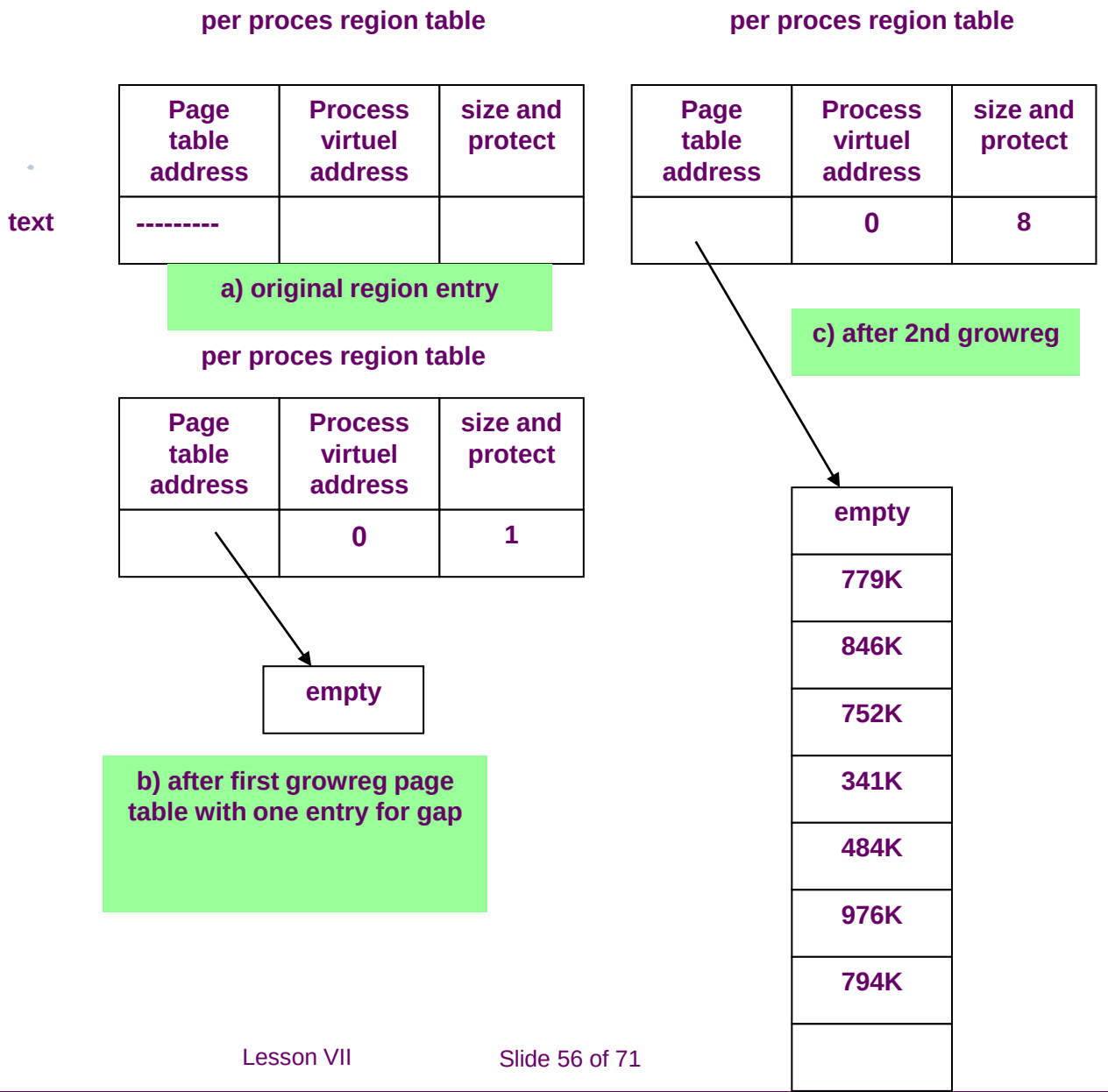
example for loadreg

- **Na primer** pretpostavimo da **kernel puni**
  - **text od 7K** u **region** koji je **attached na VA=0**,
  - **ali želi se gap** od **1K na početku regiona**, kao na slici.

- Kernel
- **prvo mora** da **alocira RT ulaz preko allocreg** i
- potom obavlja **attach** preko **attachreg na VA=0**.

- Sada će se pozvati **loadreg**, koji će obaviti **growreg 2 puta**,
  - **prvi put** za **1K gap** sa **praznim ulazom** u **PageTable**,
  - a **drugi put** za **text od 7K** sa **popunjenim ulazima u PageTable,**
  - **kernel zatim puni datoteku u region na VA=1, kao na slici**

example for loadreg



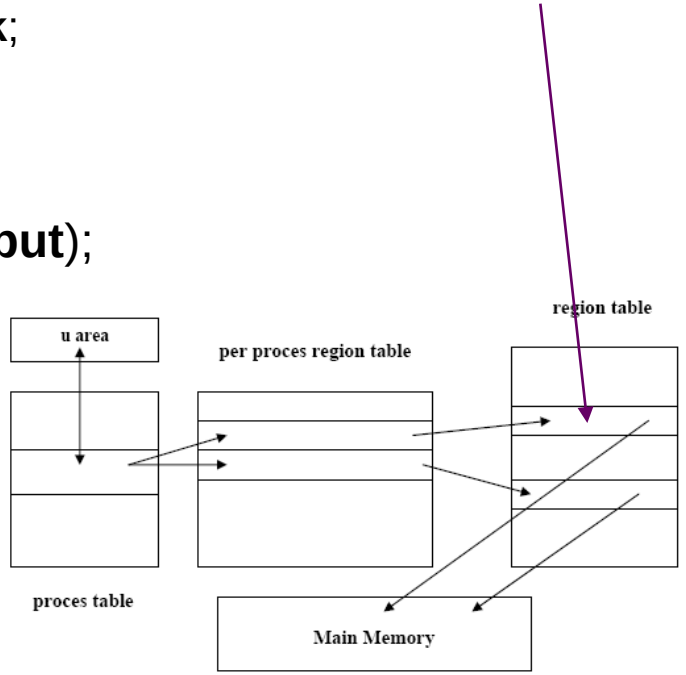
freeing a region

- Kada region više **nije attached ni jednom procesu**,
  - **kernel oslobađa region i**
  - **vraća ga u listu slobodnih regiona.**
- Ako je **regionu pridružen inode**, **kernel otpušta inode** preko algoritma **iput**.
- **Kernel optušta fizičke resurse vezane** za **region**, kao što su:
  - **page tabele**
  - **pages**

freeing a region

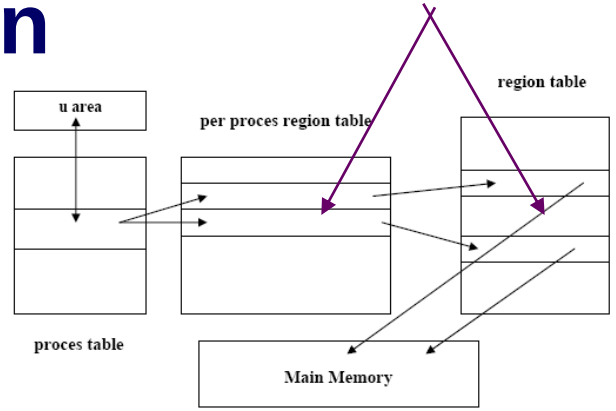
- **algortithm freereg /* free an alocated region */**
  - input: **pointer to** a **locked region**
  - output: none
- {
- if (region **RC non zero**) /* **some process still using region***/
  - {
  - **release region lock**;
  - if (region **has associated inode**) release **inode lock**;
  - **return**;
  - }
- **/*RC=0*/**
- if (**region** has **associated inode**) release **inode** (**iput**);

- **free physical memory** still associated with region;
- **free aux tables** associated with region;
- **clear region fields**;
- **place** region on the **free list**;
- **unlock** region;
- }



- **Na primer** ako kernel želi da oslobodi **stack region** sa slike i ako je RC za **region=0**, tada će se osloboditi **7 pages fizičke memorije** i tabele stranica

detaching a region



- Kernel **detach**-uje regione u SC
  - **exec**
  - **exit**
  - **shmdt** (detach shared memory) SC.
- Kernel **ažurira pregion** ulaz i više konekcija fizičke memorije, tako što **poništi triplet**.
- Kernel **dekrementira**
  - **RC** i
  - **size polje u PT ulazu** za proces **saglasno veličini** regiona.
- Ako **RC=0**, **kernel će tada osloboditi region** preko **freereg**,
  - a **ako nije**, **otpustiće se lock za region i inode,**
  - ali će region ostati **alociran za druge procese** koji ga još uvek koriste.

Detaching a Region

- **algortithm detachreg /* detach a region from a process */**
- input: **pointer to per process region table entry**
- output: none
- {
- get aux memory management tables for process, release as aproupriate;
- **decrement process size**;
- **decrement region RC**;
-
- if (**region RC=0**) **free region** (algorrithm **freereg**)
- else
- {
- **free inode lock** (if inode associated with region)
- **free region lock**;
- }
- }

duplicating a region

- **Fork SC zahteva** da kernel **duplicira regione procesa**.

- Ako je **region shared**,
  - kernel **nema potreba da fizički kopira region**,
  - već se samo **inkrementira RC**,
  - dozvoljavajući roditelju i detetu da dele region.

- Ako **region nije deljiv**,
  - kernel **mora** u **forku da kopira region**,
  - dodeljujući novi ulaz u region tabeli,
  - novu tabelu stranica i fizičku memoriju za region.

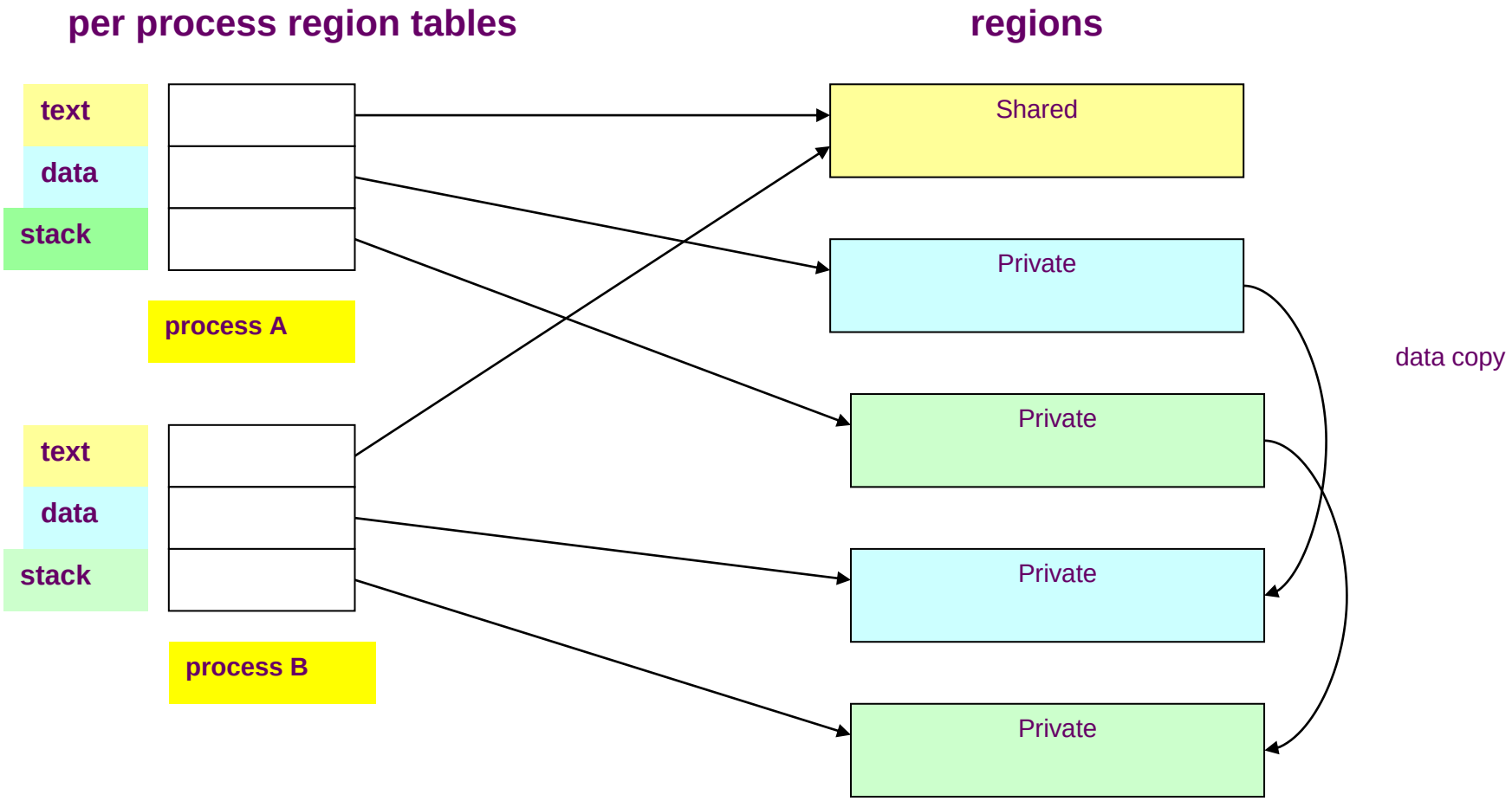
duplicating a region

- **algortithm dupreg** /* **duplicate** an **existing region** */
  - input: **pointer** to region table **entry**
  - output: **pointer** to **region** that **looks identical** to **input region**
  - **{**
- **if(region type** is **shared) return(input region pointer)**
- **/*caller will increment RC*/**

- **/*not shared*/**
  - **allocate** new **region** (algorithm **allocreg**);
  - **setup aux memory** structures as exist in **input region;**
  - **allocate** physical memory for region contents;
  - **copy** region **contents** from **input region** to **newly allocated region**;
  - **return**(**pointer** to **allocated region**)
  - **}**

duplicating a region

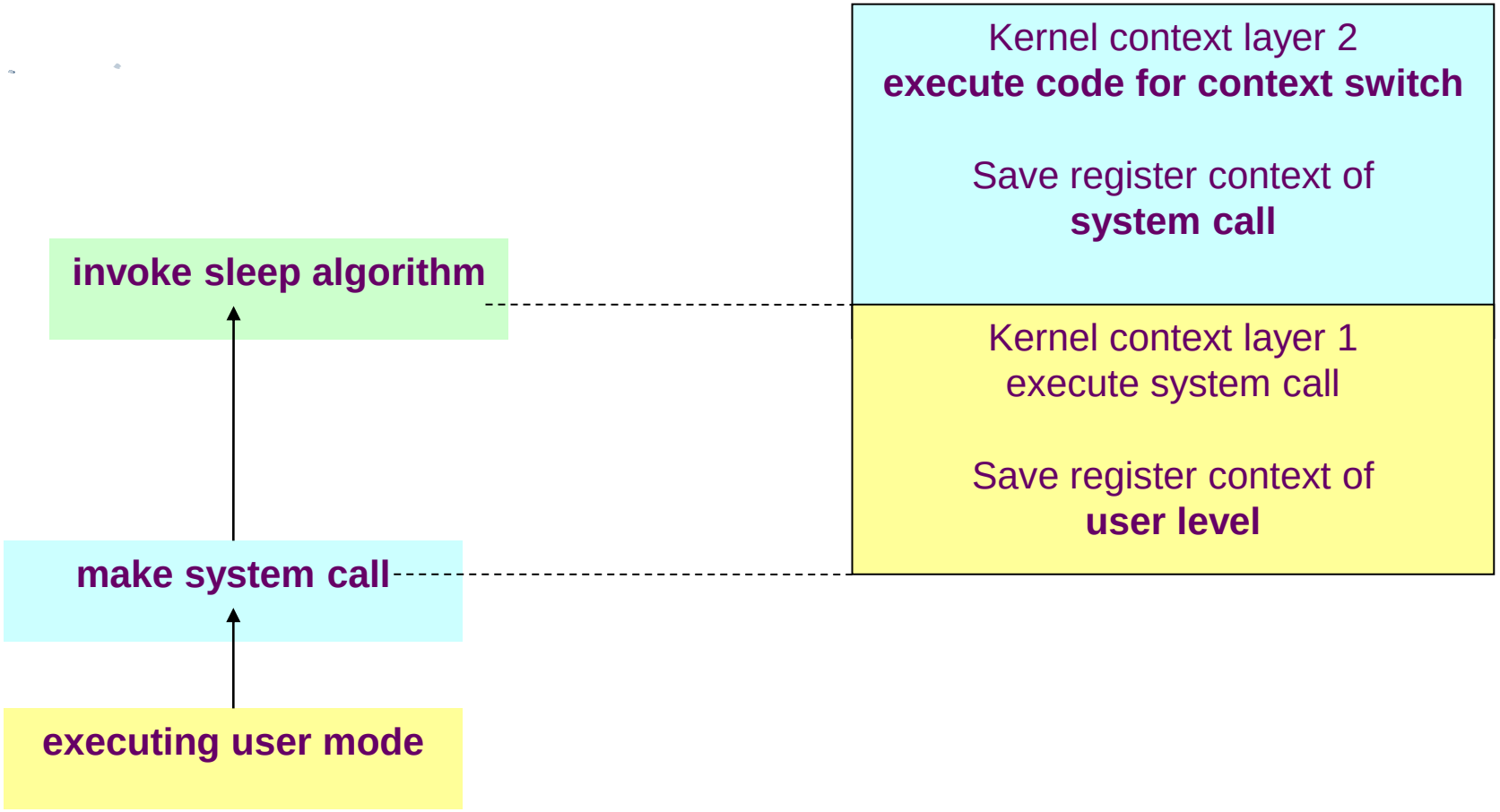
- Na slici proces **A fork-uje proces B** i **duplicira svoje regione**, pri čemu se **text region** deli, ali **data** i **stack su privatni**. Tako se kreiraju **novi regioni** koji su **kopije regiona** procesa A, mada ne mora da se **pravi fizička kopija uvek**.



sleep

- Do sada smo obradili **low-level funkcije** koje se izvršavaju prilikom tranzicija u i iz stanja 2 (kernel running) osim funkcija za uspavljivanje i buđenje procesa.
- **Objasnićemo algoritme**
  - **sleep**
    - koji menja proces iz stanja 2 (**kernel running**) u stanje (**asleep in memory**)
  - **wakeup**
    - koji menja proces iz stanja (**asleep in memory**) u stanje (**ready to run**) u memoriji ili u swapu.
- Kada **proces odlazi na spavanje**, to se po pravilu **dešava preko SC**:
  - **proces ulazi** u **kernel** (context layer 1) kada **izvršava OS trap**
  - i **odlazi** na **spavanje čekajući** na **neki događaj**.
- Kada proces ide na **spavanje**,
  - **on obavlja CSw**, gurajući **svoj tekući kontekst** na **stack**
  - i izvršavajući kernel context layer 2, kao na slici.
- **Proces takođe ide na spavanje** kada mu se **dogodi PF**, ako rezultat virtulene adrese koja nije u fizičkoj memoriji i spava dok kernel ne pročita sadržinu stranice

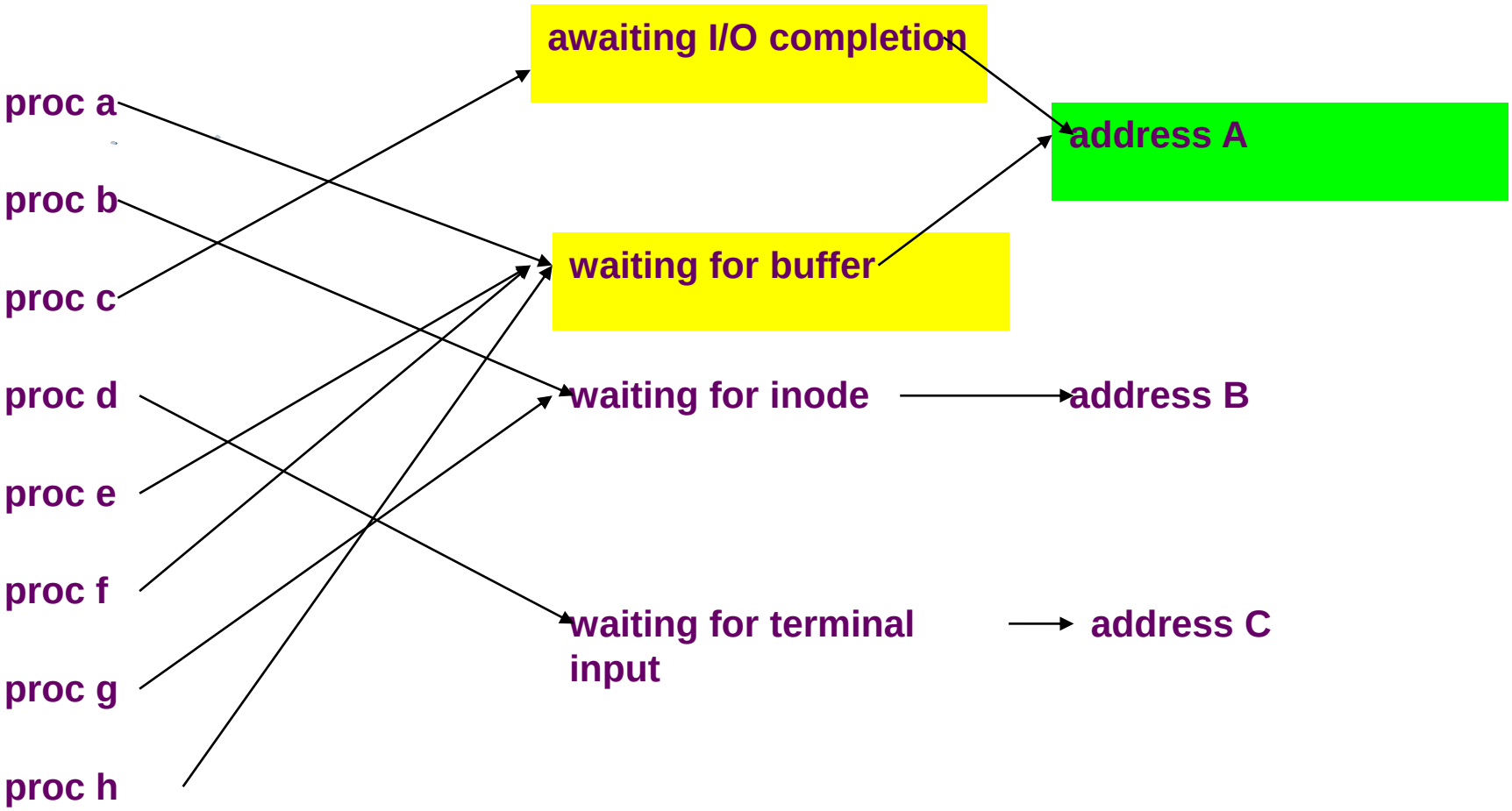
algorithm sleep



sleep events and adresses

- Podsetimo da se kaže da proces ide na spavanje na **događaj**
  - što znači da **spava dok se događaj ne desi**,
  - a **kada se događaj desi**,
  - **proces se budi** i **ulazi** u stanje **ready to run** koje **može da bude u memoriji** ili na **swap-u.**
- Mada sistem koristi **apstrakciju spavanja na događaj**, implementacija **mapira skup događaja** na **skup kernelovih virtulenih adresa**.
- Adrese koje reprezentuje događaje koji se koduju u kernel, i njihovo jedino značenje je to da kernel očekuje događaje da ga mapira na partikularnu adresu.
- Apstrakcija događaja **ne razlikuju koliko procesa čekaju na događaj**, a kao rezultat **2 anomalije** mogu proisteći.
  - **Prva** je što kernel **budi odjedanput sve procese** koji čekaju na taj događaj i oni prelaze stanje „ready to run“. Kernel ih ne budi jedan po jedan.
  - **Druga** anomalija je što **više događaja mogu da se mapiraju** na **jednu istu adresu**. U primeru sa slike, događaji kao "**waiting for the buffer**" i "**awaiting I/O completion**" mapiraju se na adresu bafera ("**addr A**"). Kada se I/O za bafer kompletira, kernel budi sve procese koji čekaju na obe vrste događaja, zato što **proces koji je inicirao I/O i čeka ga**, on je **lock-ovao bafer**, dok drugi procesi čekaju da se **bafer oslobodi**, tako da praktično **2 vrste događaja čekaju na istu stvar**.

sleep events and adresses



algortithm for sleep and wakeup

- **algorithm sleep /* asleep a process*/**
- input: **(1) sleep address**
- **(2) priority**
- output: **1 if** process awakened as a **result of signal** that process **catches**,
- **longjmp** algorithm if process **awakened** as a **result** of **signal** that is **not catch**
- **0 otherwise**
- {
- **raise processor execution level to block all interrupts;**
- **set** process state "**sleep**"
- **put process** on **sleep hash queue based** on **sleep address;**
- **save sleep address** in **process table slot**;
- **set process priority** level to **input priority**;
- if (**process sleep** is **NOT interruptible**)
- {
- **do context switch**;
- /* **process resumes execuiton here when if wake up***/
- **reset processor execution level to allow inerrupts as when process went to sleep;**
- return(0);
- }

algortithm for sleep

- **/* here process sleep is interruptible by signal*/**
- **if(no signal pending against process) /*no signal*/**
- **{**
- **do context switch**;
- /* process **resumes execution** here when **if wake up***/
  - if(no signal pending against process)
  - {
  - reset processor execution level to allow interrupts as when process went to sleep;
  - return(0);
  - }
- }
- /there is signal in sleep and after wakeup*/
- **remove process from sleep hash queue**, if still there;
- **reset processor execution level** to **allow interrupts** as when process went to sleep;

- if(**process sleep priority set to catch signals**) **return(1)**
- do **longjmp** algorithm; /* **wakeup if no catch***/
- }

algorithm for sleep

- Algoritam za **sleep** je prikazan na **sledećoj slici.**
- Kernel podiže **CPU ExLevel** da **blokira sve prekide**, tako da nema **race condition,** kada se manipuliše sa sleep redovima čekanja, a potom se sačuva informacija o starom CPU Exlevel, da bi se kasnije rekonstruisalo.
- **Stanje procesa** se **označi** kao **uspavan (asleep)**,
  - sačuva se **sleep adresa** i **prioritet u PT**, i
  - proces se gurne u **hash queue uspavanih procesa**.
- **U prostom slučaju** (sleep alg **se ne prekida**),
- proces **obavlja CSw** i **bezbedno je uspavan**,
- Kada se proces probudi, kernel ga kasnije može izabrati za izvršavanje.
- **Proces se restauira** iz **njegovog CSw** u **sleep algoritmu**,
  - obnavlja se CPU ExLevel na vrednost koju je proces imao kada je ulazio u uspavljivanje i
  - **obavlja povratak**.

algorithm wakeup

- **algorithm wakeup /* wake up a sleeping process*/**
  - **input: sleep address**
  - output: none
- {**raise processor execution level to block all interrupts**;
- **find sleep hash queue** for **sleep address**;
- **for** (**every process asleep on sleep address**)
- {
- remove process from hash queue;
- mark process state "**ready to run**"
- **put process** on **scheduler** list of process ready to run;
- **clear field** in **PT entry for sleep address**;

  - **if(process not loaded in memory**) **wakeup swapper process (0);**
  - {
  - else
    - if (**awakened process is more eligible** to **run than currently running process**) **set scheduler flag**;
  - } /*for*/
- restore processor execution level to original level;
- }