Operativni Sistemi 2

4. Upravljanje memorijom

Zadatak 1. FIFO algoritam
Implementirati FIFO algoritam zamene.

```
struct PMTEntry{
  int valid;
  int dirty;
  int frame;
  // ostale informacije
};

struct PCB{
  PMTEntry* pmt;
  // ostale informacije
}

struct FrameEntry{
  PMTEntry *pmtEntry;
  // ostale informacije
};

struct PCB* running;
const int FramesNo = ...;
FrameEntry frames[FramesNo];
int victim = 0;

int getVictim(){
  return victim;
}

//hardware
int hitPage(int pageID, int write){
  if (running->pmt[pageID].valid == 1){
    if (write == 1)
      running->pmt[pageID].dirty = 1;
    return running->pmt[pageID].frame;
  }
  return 0; //page fault
}
```

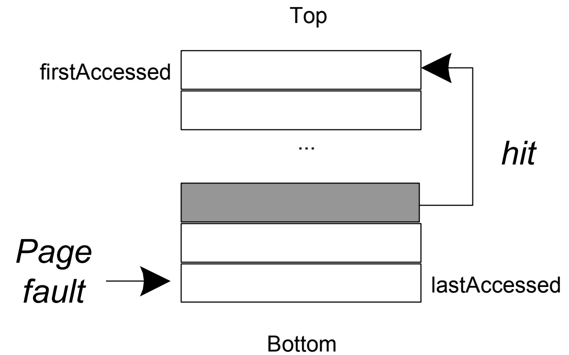
Modul OSa (pager) koji obrađuje page fault:

1. proverava da li je pristup traženoj stranici uopšte dozvoljen procesu; ako nije, proces se može ugasiti
2. pronalazi slobodan okvir u memoriji
   ako takvog nema:
   i. izbacuje taj proces i učitava drugi (swapping) ili
   ii. bira stranicu-”žrtvu” (victim) istog ili drugog procesa koju će izbaciti iz OM po algoritmu zamene (page replacement algorithm); ako je potrebno, pokreće operaciju snimanja stranice koju izbacuje na disk
3. pokreće se operacija sa diskom za učitavanje tražene stranice u odabrani okvir
4. kada se operacija završi, ažurira se PMT procesa

```
int loadPage(int pageID){
  if (frames[victim].pmtEntry){
    if (frames[victim].pmtEntry->valid){
      frames[victim].pmtEntry->valid = 0;
      if (frames[victim].pmtEntry->dirty == 1){

            // vraca stranu na hdd.
      }
    }
    frames[victim].pmtEntry = 0;
  }

  // ucitava stranu

  running->pmt[pageID].dirty = 0;
  running->pmt[pageID].frame = victim;
  running->pmt[pageID].valid = 1;
  frames[victim].pmtEntry = running->pmt + pageID;
  int newFrame = victim;

  victim = (victim+1) % FramesNo;
  return newFrame;
}
```

Zadatak 2. LRU algoritam

Implementirati LRU algoritam zamene stranica.




```
struct PMTEntry{
  int valid;
  int dirty;
  int frame;
  struct PMTEntry *prev, *next;
  // ostale informacije
};
struct PCB{
  struct PMTEntry *pmt;
  // ostale informacije
};
struct PCB* running;
struct PMTEntry *firstAccessed=0, *lastAccessed=0;
int victim = 0; // victim < FramesNo - ima slobodnih
const int FramesNo = ...;

int getVictim(){ //ako ima slobodnih dodeljujemo redom
  if (victim < FramesNo) return victim;
    else return lastAccessed->frame;
}

int loadPage(int pageID){
  int freeFrame = 0;
  if (victim < FramesNo) freeFrame = victim++;
  else{
    lastAccessed->valid = 0;
    freeFrame = lastAccessed->frame;
    if (lastAccessed->dirty) {
      //vrati stranu na disk
      lastAccessed->dirty = 0;
    }
    lastAccessed = lastAccessed->prev;
    lastAccessed->next = 0;
  }

  //dovuci trazenu stranu sa diska u okvir freeFrame

  //azuriraj PMT
  running->pmt[pageID].next = firstAccessed;
  firstAccessed->prev = running->pmt+pageID;
  firstAccessed = firstAccessed->prev;
  firstAccessed->prev = 0;
  firstAccessed->frame = freeFrame;
  firstAccessed->dirty = 0;
  firstAccessed->valid = 1;
```

```
  return freeFrame;
}

//hardware
int hitPage(int pageID, int write){
  if (running->pmt[pageID].valid == 1){
    if (write == 1)
      running->pmt[pageID].dirty = 1;
     // ako nije na vrhu
    if (firstAccessed != running->pmt+pageID){
      // Da li je na dnu?
      if (running->pmt[pageID].next)
        running->pmt[pageID].next->prev =
                             running->pmt[pageID].prev;
         //u slučaju da je na dnu
        else lastAccessed = lastAccessed->prev;
      running->pmt[pageID].prev->next =
                             running->pmt[pageID].next;
      firstAccessed->prev = running->pmt + pageID;
      running->pmt[pageID].next = firstAccessed;
      firstAccessed = firstAccessed->prev;
      firstAccessed->prev = 0;
    }
   return running->pmt[pageID].frame;
  }
  return 0;
}
```

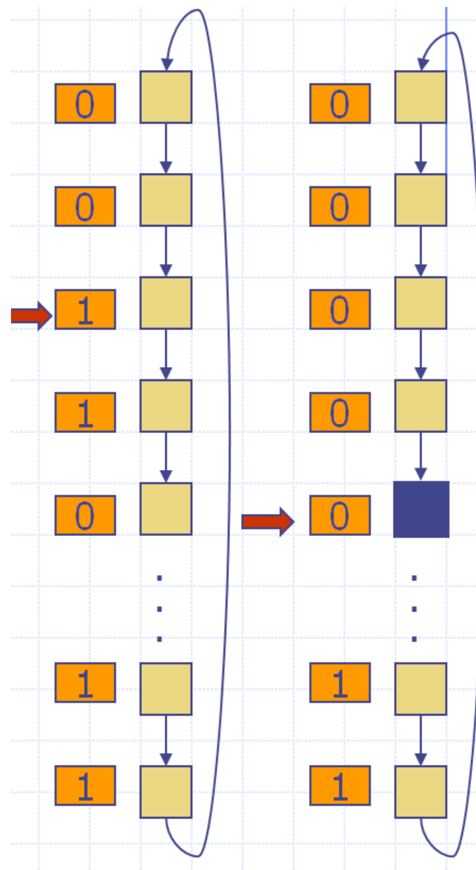
Zadatak 3. Aproksimacija LRU algoritma - Bit pristupa

- Januar 2006.

Za izbor stranice za zamenu u nekom sistemu koristi se algoritam „davanja nove šanse“ ili „časovnika“ (*second-chance, clock algorithm*). Struktura koja čuva bite referenciranja implementirana je kao prosti niz (konstanta `FrameNum` predstavlja broj okvira):

```
const unsigned long int FrameNum = ...;
int refereceBits[FrameNum]; // 0 - not referenced, !=0 -
referenced
unsigned long int clockHand;
```

Implementirati funkciju `getVictimFrame()` koja vraća broj okvira čiju stranicu treba zameniti po ovom algoritmu zamene.



Rešenje:

```
unsigned long int getVictimFrame () {
  while (referenceBits[clockHand]) {
    referenceBits[clockHand] = 0; // Give a second chance
    clockHand=(clockHand+1)%FrameNum;
  }
  int victim = clockHand;
  clockHand=(clockHand+1)%FrameNum;
  return victim;
}
```

- Prošireni algoritam nove šanse - uređeni par (*reference bit*, *modify bit*)
    - (0,0): najbolji kandidat za izbacivanje, jer stranica nije ni korišćena, niti modifikovana, pa ne mora ni da se snima na disk
    - (0,1): sledeći kandidat za izbacivanje, jer stranica nije korišćena, ali je modifikovana, pa mora da se snima na disk
    - (1,0): sledeći kandidat za izbacivanje, jer stranica jeste korišćena, ali nije modifikovana, pa ne mora da se snima na disk
    - (1,1): najlošiji kandidat za izbacivanje, jer je stranica i korišćena i modifikovana

Zadatak 4. Januar 2007.

Neki sistem primenjuje algoritam časovnika (davanja nove šanse, *clock*, *second-chance*) za izbor stranice za izbacivanje. U donjoj tabeli date su različite situacije u kojima treba odabrati stranicu za zamenu. Date su vrednosti bita referenciranja za sve stranice koje učestvuju u izboru i pozicija „kazaljke“. Kazaljka se pomera u smeru prema višim brojevima stranica. Za svaku od datih situacija navesti koja stranica će biti zamenjena i novo stanje posle izbora stranice za izbacivanje.

	Situacija 1		Situacija 2		Situacija 2	
Stranica	Bit ref.	Kazaljka	Bit ref.	Kazaljka	Bit ref.	Kazaljka
0	1		1		1	
1	1		0		1	
2	0		1		1	
3	0	X	1		1	X
4	1		0		1	
5	0		1	X	1	
6	1		1		1	
7	1		1		1	

Odgovor:

	Situacija 1		Situacija 2		Situacija 2	
Stranica	Bit ref.	Kazaljka	Bit ref.	Kazaljka	Bit ref.	Kazaljka
0	1		0		0	
1	1		0		0	
2	0		1	X	0	
3	0		1		0	
4	1	X	0		0	X
5	0		0		0	
6	1		0		0	
7	1		0		0	

Zamenjena stranica:

Situacija 1: 3 Situacija 2: 1 Situacija 3: 3

Zadatak 5. Jun 2006.

Za izbor stranice za zamenu u nekom sistemu koristi se aproksimacija LRU algoritma sa dodatnim bitima istorije referenciranja. Struktura koja čuva bite referenciranja implementirana je kao niz reči, pri čemu svakoj stranici odgovara jedan bit, a biti su poređani od najnižeg ka najvišem: stranici 0 odgovara bit 0 u reči 0, stranici 1 odgovara bit 1 u reči 0, itd. Veličina jedne reči (tip `int`, u bitima) je definisan konstantom `bitsInWord`. Registri istorije referenciranja implementirani su nizom `referenceHistory`, pri čemu svakoj stranici odgovara jedan element ovog niza veličine jedne reči.

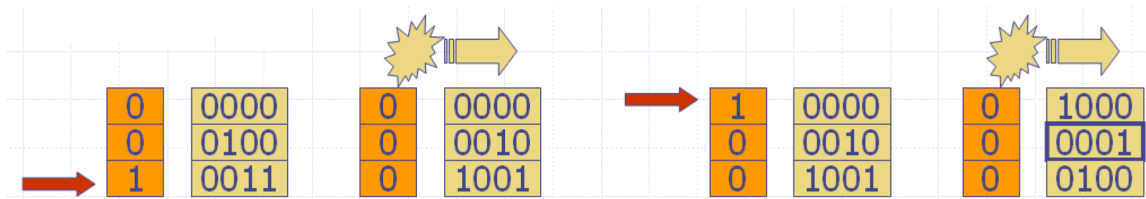
```
const unsigned int bitsInWord = ...;
const unsigned int NumOfPages = ...;  // Number of pages in address
space
unsigned int* referenceBits; // 0 - not referenced, 1 - referenced
unsigned int referenceHistory[NumOfPages];
```

Implementirati:

a)(5) funkciju `updateReferenceHistory()` koja se poziva periodično i koja treba da ažurira registre istorije referenciranja bitima referenciranja;

b)(5) funkciju `getVictimPage()` koja vraća redni broj stranice koja je izabrana za zamenu.

Rešenje:



a)(5)

```
void updateReferenceHistory () {
  for (int pg=0; pg<NumOfPages; pg++) {
    unsigned int refBitsWordNo = pg/bitsInWord; //br. refencirane reci
    unsigned int refBitsBitNo = pg%bitsInWord; //pozicija bita u reci
    unsigned int refBitWord = referenceBits[refBitsWordNo];
    unsigned int refBit = refBitWord>> bitsInWord-refBitsBitNo-1;
    refBit<<=(bitsInWord-1);
    referenceHistory[pg]>>=1;  //Shift right reference history register
    referenceHistory[pg]|=refBit;  // and set its MSB to reference bit
  }
}
```

Napomena: dato rešenje je verovatno jedno od razumljivijih, ali ne i najefikasnije (u smislu manjeg broja pristupa memoriji). Npr.

```
unsigned int refBit = (refBitWord<<refBitsBitNo)& ~(~0U>>1)
```

b)(5)

```
unsigned int getVictimPage () {
  unsigned int result = 0;
  unsigned int minRefHist = referenceHistory[0];
  for (int pg=1; pg<NumOfPages; pg++)
    if (referenceHistory[pg]<minRefHist) {
      result = pg;
      minRefHist = referenceHistory[pg];
    }
  return result;
}
```

Zadatak 6. April 2006.

Za izbor stranice za zamenu u nekom sistemu koristi se aproksimacija LRU algoritma sa dodatnim bitima referenciranja (*additional-reference-bit algorithm*). Registar istorije bita referenciranja ima 4 bita. Posmatra se proces čije su četiri stranice označene sa 0..3 trenutno u operativnoj memoriji. Proces generiše sledeću sekvencu obraćanja stranicama; u ovoj sekvenci, oznaka X predstavlja trenutak kada stiže periodični prekid na koji operativni sistem pomera udesno registre istorije i upisuje u njih bite referenciranja:

`0,1,3,X,2,3,0,X,0,2,1,0,X,1,0,X,2,3,X,0,1,2,X,3,0,X,1,0,2,X,3,0,X,1,0,X`

Prikazati sadržaj registara istorije posle ove sekvence i navesti koja stranica bi bila izabrana za izbacivanje ukoliko se posle ove sekvence traži zamena stranice.

Rešenje:

0	1	0	0	0
1	1	0	0	0
2	0	0	0	0
3	1	0	0	0

0	1	1	0	0
1	0	1	0	0
2	1	0	0	0
3	1	1	0	0

0	1	1	1	0
1	1	0	1	0
2	1	1	0	0
3	0	1	1	0

...

Kraće rešenje – gledati samo poslednje 4 sekvence:

0	1	1	1	1
1	1	0	1	0
2	0	0	1	0
3	0	1	0	1

Bila bi izbačena stranica 2

Napomena: primetiti da je ovo razlog zašto ovaj algoritam predstavlja aproksimaciju LRU algoritma

Zadatak 7. Septembar 2006.

U nekom sistemu sa virtuelnom memorijom stranice se označavaju kao „zaprljane“ (*dirty*) ukoliko je izvršena neka operacija upisa u tu stranicu od trenutka njenog učitavanja u memoriju. Kada se ta stranica izabere za zamenu, ukoliko je označena kao „zaprljana“, potrebno je pokrenuti operaciju njenog snimanja na uređaj (disk) koji služi za zamenu stranica.

a)(5) Navesti tehniku koja povećava verovatnoću da stranica izabrana za zamenu nije označena kao „zaprljana“ i time poboljšava performanse sistema jer smanjuje broj operacija snimanja na disk prilikom zamene stranica, odnosno skraćuje prosečno vreme zamene stranica.

Odgovor:
OS u pozadini, tokom rada drugih procesa, obilazi „zaprljane“ stranice i pokreće operaciju njihovog snimanja na disk kad god je uređaj koji za to služi slobodan. Time se povećava verovatnoća da stranica koja je izabrana za zamenu nije označena kao „zaprljana“.

b)(5) Navesti tehniku koja odlaže operaciju upisa na disk, a omogućava da je prilikom stranične greške (*page fault*) najčešće već na raspolaganju slobodan okvir i time poboljšava performanse sistema skraćujući vreme zamene, jer ne postoji potreba za izbacivanjem i snimanjem stranice. Za kada se onda odlaže snimanje zaprljane stranice?

Odgovor:
Vođenje „bazena“ slobodnih stranica (*page pooling*): kada se traži slobodan okvir, uglavnom se pronalazi takav u bazenu i proces koji je tražio stranicu može odmah da nastavi izvršavanje (nema operacije snimanja zaprljane stranice koja se izbacuje). Da bi se bazen dopunio, OS u pozadini, tokom rada drugih procesa, bira neku stranicu za izbacivanje da bi okvir koji ona zauzima oslobodio i dodao u bazen. U tom trenutku, kada se neki okvir označava slobodnim i smešta u bazen, pokreće se snimanje njegovog sadržaja na disk.

Napomena: Naravno, prilikom stranične greške treba proveriti i da li se kojim slučajem ta stranica ne nalazi u bazenu slobodnih stranica.

Zadatak 8. Oktobar 2006.

U nekom sistemu sa straničnom organizacijom virtuelne memorije primenjuje se tehnika sprečavanja pojave *thrashing* pomoću praćenja radnog skupa stranica. Informacija o radnom skupu, zapravo o njegovoj aproksimaciji, dobija se tako što operativni sistem periodično prepisuje bite referenciranja stranica u PCB strukture procesa i zatim ih briše. PCB strukture su prealocirane u statički dimenzionisani niz `processes`. Date su sledeće deklaracije:

```
const unsigned NumOfVMPages = ...;// Num of pages in virtual address
space
unsigned NumOfFrames = ...;// Num of physical frames available for
paging
const unsigned MaxNumOfProcs = ...; // Max num of processes

struct PCB {
  int isActive; // Is this process active (1) or is swapped out (0)
  int reference[NumOfVMPages]; // Reference info for all pages
  ...
};

PCB processes[MaxNumOfProcs];
int isThrashing();
```

Implementirati funkciju `isThrashing()` koja treba da vrati 1 ako je po kriterijumu ukupne veličine radnih skupova nastala pojava *thrashing*, a 0 ako nije.

Rešenje:

```
int isThrashing() {
  unsigned long totalWSSize = 0;
  for (unsigned iProc=0; i<MaxNumOfProcs; iProc++)
    if (processes[iProc].isActive)
      for (unsigned iPage=0; iPage<NumOfVMPages; iPage++)
        totalWSSize += processes[iProc].reference[iPage];
  return totalWSSize>NumOfFrames;
}
```

Zadatak 9. Jun 2009.

Neki sistem koristi tehniku ploča (*slab*) za alokaciju memorije za potrebe jezgra. U datom kešu (*cache*) za objekte jednog tipa, evidencija slobodnih pregradaka (*slot*) vodi se kao jednostruko ulančana lista, pri čemu se pokazivač za ulančavanje upisuju u sam slobodan pregradak koji je zauzimao neki objekat. Date su sledeće deklaracije:

```
struct FreeSlot {
  FreeSlot* next; // Next FreeSlot in the list of free slots of a cache
};

struct Cache* {
  ...
  FreeSlot* freeSlots; // The list of free slots;
};

void free (Cache* cache, void* object);
```

Implementirati operaciju `free()` koja pregradak koji je zauzimao dati objekat u datom kešu proglašava slobodnim.
Rešenje:

```
void free (Cache* cache, void* object) {
  if (cache==0 || object==0) return; // Error
  FreeSlot* fs = (FreeSlot*)object;
  fs->next = cache->freeSlots;
  cache->freeSlots = fs;
}
```

Zadatak 10. Septembar 2009.

Za alokaciju memorije u jezgru nekog operativnog sistema primenjuje se sistem parnjaka (*buddy*). U nekom trenutku stanje zauzetosti prikazano je na donjoj slici (prikazani su blokovi i njihove veličine izražene u broju stranica; osenčeni blokovi su zauzeti, beli su slobodni). Na isti način prikazati stanje nakon izvršavanja zahteva za alokacijom memorije veličine 1 stranice.

4	2	2	4	4

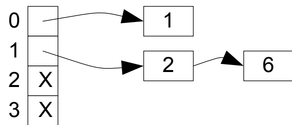
Rešenje:

4	1	1	2	4	4

Napomena: Koja je prednost tehnike ploča u odnosu na tehniku parnjaka? Ali da li se ona uvek može upotrebiti (npr. bafer)?

Zadatak 11. Septembar 2010.

Posmatra se neki alokator memorije za potrebe jezgra na principu „parnjaka“ (*buddy*). Najmanja jedinica alokacije je blok, a ukupna raspoloživa memorija kojom upravlja alokator ima 8 blokova koji su označeni brojevima 0..7. Za potrebe evidencije slobodnih komada memorije, alokator vodi strukturu čije je stanje u nekom trenutku prikazano na slici. Svaki ulaz *i* (*i*=0..3) ovog niza sadrži glavu liste slobodnih komada memorije veličine 2^i susednih blokova. U svakom elementu liste je broj prvog bloka u slobodnom komadu.

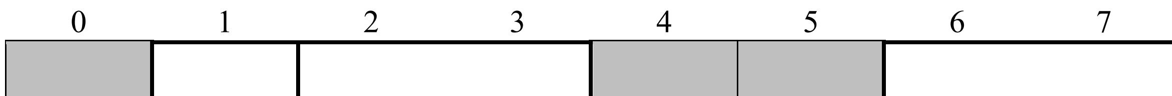


Nacrtati izgled ove strukture nakon što je, za prikazano početno stanje:

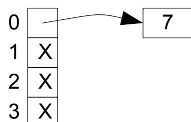
a)(5) Izvršena sukcesivno alokacija tri komada memorije veličine 2, 1 i 1 blok, tim redom.

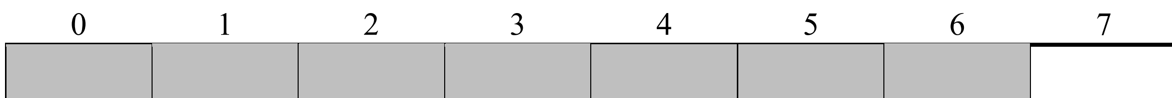
Rešenje:

Početno stanje:



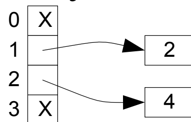
Nakon alokacije:

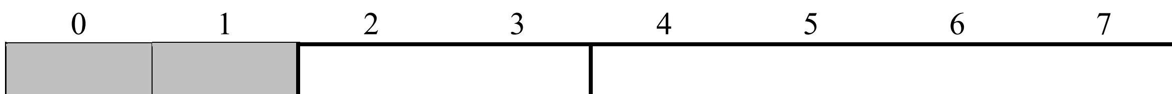




b)(5) Nakon stanja posle alokacije iz tačke a), izvršena sukcesivno dealokacija tri komada memorije koji počinju blokovima broj 2 veličine 2, 6 veličine 1 i 4 veličine 2, tim redom.

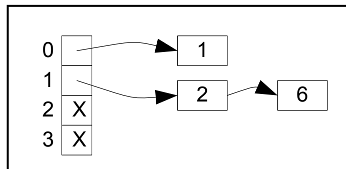
Rešenje:





Zadatak 12. Oktobar 2010

Posmatra se neki alokator memorije za potrebe jezgra na principu „parnjaka“ (*buddy*). Najmanja jedinica alokacije je blok, a ukupna raspoloživa memorija kojom upravlja alokator ima 2^{N-1} blokova koji su označeni brojevima $0..2^{N-1}-1$. Za potrebe evidencije slobodnih komada memorije, alokator vodi strukturu čije je stanje u nekom trenutku prikazano na slici, za primer $N = 4$. Svaki ulaz i (i=0..N-1) prikazanog niza `buddy` sadrži glavu liste slobodnih komada memorije veličine 2^i susednih blokova. Glava liste sadrži broj prvog bloka u slobodnom komadu, a broj narednog bloka u listi je upisan na početku svakog slobodnog bloka u listi (-1 za kraj liste). Deklaracije potrebnih struktura date su dole. Funkcija block() vraća adresu početka bloka broj n.




```
const int N = ...; // N>=1
int buddy[N];
void* block(int n);
```

Realizovati funkciju:

```
void buddy_init ();
```

koja inicijalizuje prikazanu strukturu za početno stanje alokatora u kome je ceo prostor od 2^{N-1} susednih blokova slobodan.

Rešenje:

```
void buddy_init () {
  for (int i=0; i<N-1; i++) buddy[i]=-1;
  buddy[N-1]=0;
  *((int*)block(0))=-1;
}
```

Zadatak 13. Jun 2007.

Neki operativni sistem podržava straničnu organizaciju operativne memorije sa učitavanjem stranica na zahtev (*demand paging*, DP), kao i memorijski preslikane fajlove (*memory-mapped files*, MMF). Sistem obezbeđuje uslugu kojom se proizvoljni fajl sa datim imenom preslikava u virtuelni adresni prostor pozivajućeg procesa počev od zadate adrese. Ukratko, ali precizno objasniti u čemu se sastoji najjednostavniji postupak kreiranja novog procesa nad programom u zadatom .exe fajlu.
Odgovor:
Kreiranje novog procesa svodi se praktično samo na kreiranje potrebnih internih struktura koje opisuju proces i njegov aresni prostor (PCB), pri čemu su sve stranice inicijalno označene kao neučitane. Kontekst treba kreirati nad istim, generičkim kodom koji obavlja sledeće radnje:

- poziv sistemske usluge kojom se dati .exe fajl sa programom preslikava u virtuelni adresni prostor procesa, počev od neke fiksne lokacije u virtuelnom prostoru;
- skok na neku fiksnu lokaciju u virtuelnom adresnom prostoru na kojoj se očekuje početak (prva instrukcija) programa, ukoliko je to propisano tako, ili indirektni skok preko neke takve lokacije u kojoj se očekuje adresa početka programa.

Sve ostalo će obaviti mehanizmi DP i MMF, jer su stranice programa preslikane u fajl sa programskim kodom i statički alociranim podacima, a inicijalno su neučitane, pa će izvršavanje već prve instrukcije generisati *page fault* i učitavanje stranice iz programskog fajla. Dinamičku alokaciju prostora, npr. za podatke ili čak i stek, obaviće sam program tokom svog izvršavanja, pozivajući odgovarajuće sistemske pozive.

Zadatak 14. Oktobar 2009.
Predložiti sistemski poziv kojim se uspostavlja preslikavanje dela virtuelne memorije sa straničnom organizacijom u fajl (*memory-mapped file*). Navesti potpis ovog sistemskog poziva u jeziku C i precizno objasniti značenje tog poziva.

Rešenje:

```
int mem_map_file (int pgNum, int size, char* fileName);
```

Ovaj poziv alocira deo virtuelnog memorijskog prostora počev od zadate stranice `pgNum` zadate veličine `size` stranica (proglašava taj deo memorije korišćenim od strane pozivajućeg procesa) i uspostavlja preslikavanje sadržaja tog dela memorije u fajl sa zadatim punim imenom `fileName`, organizovanjem odgovarajućih struktura podataka. U slučaju uspeha vraća 1, u slučaju greške vraća kod greške (različit od 0).

Zadatak 15. Kolokvijum 2. Novembar 2011.

U jezgru nekog operativnog sistema primenjuje se sistem ploča (*slab allocator*) za alokaciju struktura za potrebe jezgra. Za alokaciju nove ploče kada u kešu više nema slobodnih slotova koristi se niži sloj koji implementira *buddy* alokator. U nastavku su date delimične definicije klasa `Cache` i `Slab` i implementacije nekih njihovih operacija. Slotovi za alokaciju su tipa `X`. Struktura `Cache` predstavlja keš i čuva pokazivač `headSlab` na prvu ploču u kešu; ploče u kešu su ulančane u jednostruku listu. Strukura `Slab` predstavlja ploču. U njoj su pokazivač na sledeću ploču istog keša `nextSlab`, kao i pokazivač `freeSlot` na prvi slobodan slot u nizu `slots` svih slotova u ploči. Slobodni slotovi su ulančani u jednostruku listu preko pokazivača koji se smeštaju u sam slot, na njegov početak. Inicijalizaciju jedne ploče vrši dati konstruktor.

```
const int numOfSlotsInSlab = ...;

class Cache {
public:
  ...
  X* alloc();
private:
  friend class Slab;
  Slab* headSlab;
}

class Slab {
public:
  Slab (Cache* ownerCache);
  static void* operator new (size_t s) { return buddy_alloc(s); }
  static void  operator delete (void* p) { buddy_free(p,sizeof(Slab));
}
private:
  friend class Cache;
  Slab* nextSlab;
  X* freeSlot;
  X slots[numOfSlotsInSlab];
};

Slab::Slab(Cache* c) {
    this->nextSlab=c->headSlab;
    c->headSlab=this;
    this->freeSlot=&this->slots;
    for (int i=0; i<numOfSlotsInSlab-1; i++)
      *(X**)(&slots[i])=&slots[i+1];
    *(X**)(&slots[numOfSlotsInSlab-1])=0;
}
```

Implementirati operaciju `Cache::alloc()` koja treba da alocira jedan slobodan slot `X`. Nije potrebno optimizovati alokaciju tako da se slot traži najpre u delimično popunjenim pločama, pa tek u praznoj, već se može prosto vratiti prvi slobodan slot na koga se naiđe.

Rešenje:

```
X* Cache::alloc() {
  Slab* s=this->headSlab;
  for (; s!=0; s=s->nextSlab)  // Find a slab with a free slot
    if (s->freeSlot) break;
  if (s==0) // No free slot. Allocate a new slab:
    s = new Slab(this);
  if (s==0 || s->freeSlot==0) return 0; // Exception: no free memory
  X* ret = s->freeSlot;
  s->freeSlot=*(X**)s->freeSlot;
  return ret;
}
```