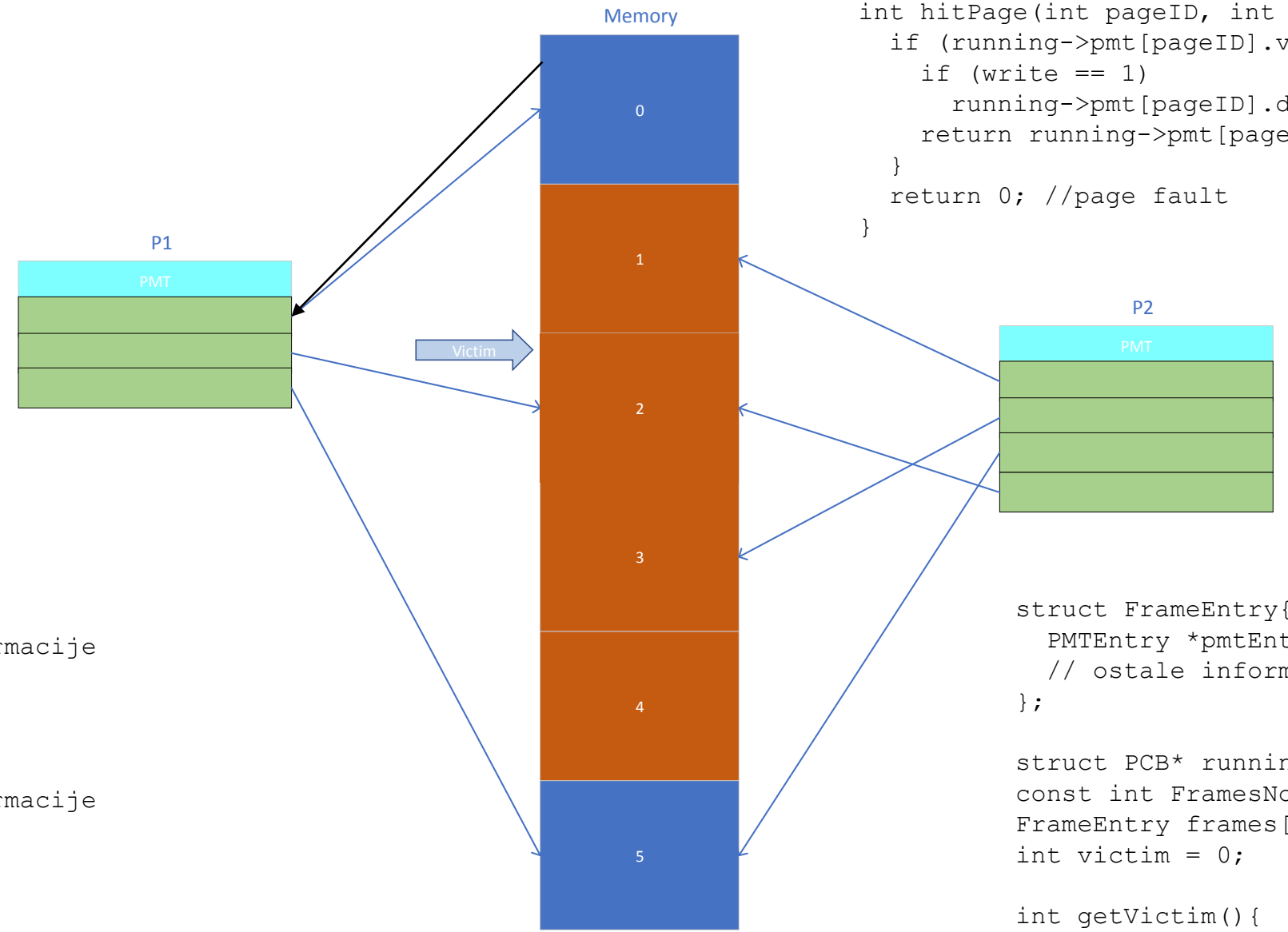
Operativni sistemi 2

Vežbe 4
Upravljanje memorijom

Lista predmeta:
IR smer - 13e113os2@lists.etf.rs
SI Smer - 13s113os2@lists.etf.rs

Zadatak

Implementirati FIFO algoritam zamene stranica kod virtuelne memorije.




```
//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
}
```

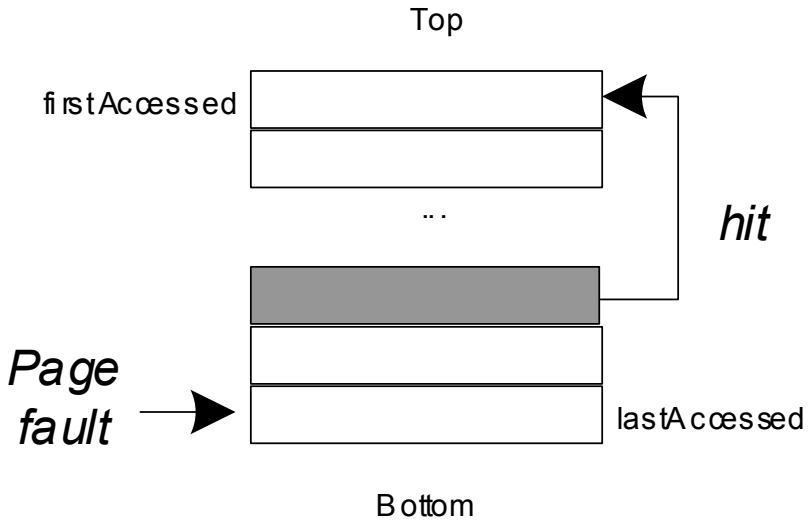
```
struct PMTEntry{
  int valid;
  int dirty;
  int frame;
  // ostale informacije
};

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;
}
```

```
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

Implementirati LRU algoritam zamene stranica kod virtuelne memorije.



```
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;
}
```

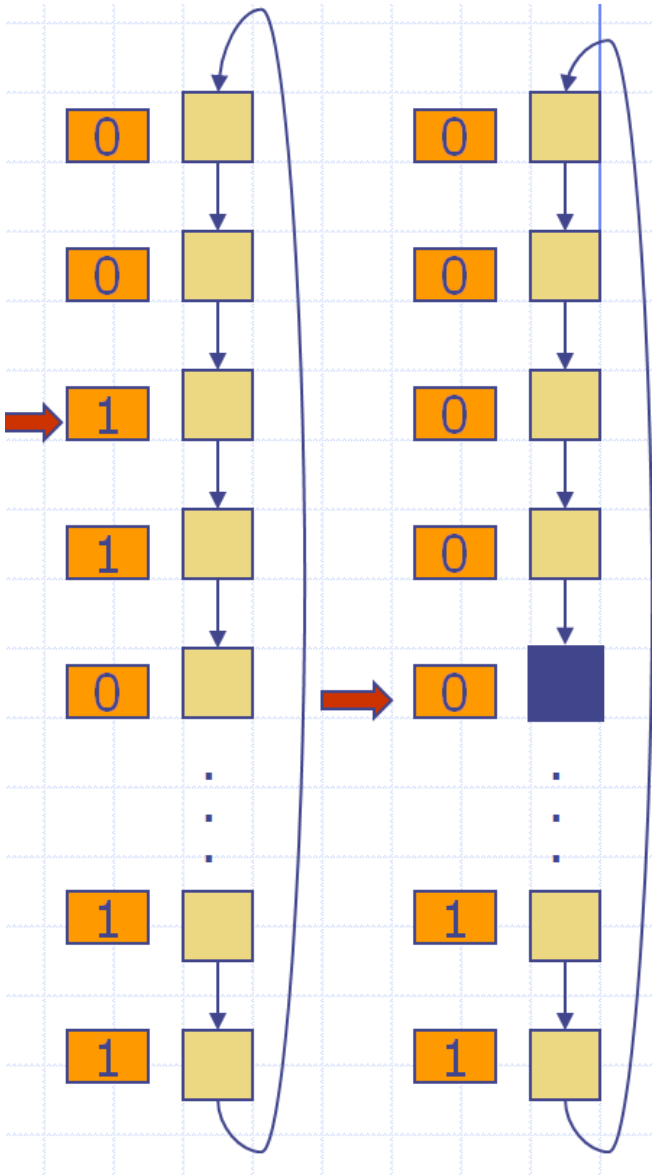
```
//hardware
int hitPage(int pageID, int write){
  if (running->pmt[pageID].valid == 1){
    if (write == 1)
      running->pmt[pageID].dirty = 1;
    // ako nije na vrhu
    if (firstAccessed != running->pmt+pageID){
      // Da li je na dnu?
      if (running->pmt[pageID].next)
        running->pmt[pageID].next->prev =
                              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 – 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.


0
0
1
1
0
1
1
0
0
0
0
1
1

```
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;
}
```

Zadata – 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 3	
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	

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

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

	Situacija 1		Situacija 2		Situacija 3	
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	

Zadatak – Jun 2006.

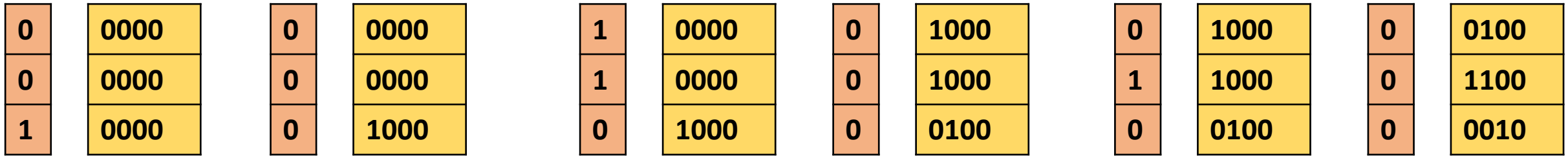
Za izbor stranice za zamenu u nekom sistemu koristi se aproksimacija LRU algoritma sa dodatnim bitima istorije referenciranja. Struktura koja čuva bite referenciranja implementirana je kao niz reči, pri čemu svakoj stranici odgovara jedan bit, a biti su poređani od najnižeg ka najvišem: stranici 0 odgovara 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.



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
  }
}
```

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 – 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.

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

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

Zadatak – Decembar 2017.

Neki sistem koristi modifikovani algoritam časovnika/nove šanse (engl. *clock/second chance*), tako što se, umesto bita referenciranja, u obzir uzima brojač pridružen svakoj stranici. Prilikom obilaska kazaljke, za žrtvu se izabira stranica čiji je taj brojač nula; ako je brojač veći od nule, smanjuje se za jedan i stranici daje nova šansa. Operativni sistem povremeno, na periodične prekide, inkrementira brojač pridružen stranici na osnovu njenog bita referenciranja, s tim da ne dozvoljava prekoračenje (ako je brojač stigao do maksimalne vrednosti, takav i ostaje).
Sistem primenjuje straničenje u dva nivoa, virtuelna adresa je 64 bita, adresibilna jedinica je bajt, a smeštanje višebajtnih reči u memoriju je po šemi niži bajt-niža adresa (*little endian*). Svaki ulaz u PMT drugog nivoa, odnosno deskriptor stranice, sadrži dve 64-bitne reči. Nižu reč koristi hardver prilikom preslikavanja, dok je viša reč potpuno slobodna za korišćenje od strane operativnog sistema. Tu reč kernel koristi tako što u najniža 24 bita čuva nižih 24 bita virtuelne adrese iste ovakve (više) reči deskriptora sledeće stranice u kružnoj listi stranica uvezanih po FIFO principu za algoritam zamene stranica, u naredna 24 bita isti takav pokazivač na prethodnu u toj listi, a u najviših 16 bita smešta pomenuti brojač korišćenja stranice. Kernel preslikava svoj deo fizičke memorije u najviših 16 MB virtuelnog adresnog prostora svakog procesa, a prilikom izvršavanja kernel koda preslikavanje odgovara tekućem procesu (aktivan je memorijski kontekst tog procesa).

Primenjuje se globalna politika zamene stranica, a globalna promenljiva `clockHand` tipa `uint64*` predstavlja kazaljku za algoritam časovnika i ukazuje na drugu (višu) reč deskriptora stranice.

```
typedef unsigned long uint64; // 64 bits
typedef unsigned int uint32; // 32 bits
typedef unsigned short uint16; // 16 bits
typedef unsigned char uint8; // 8 bits
uint64* getVictim ();
```

Implementirati funkciju `getVictim` koja treba da vrati pokazivač na prvu (nižu) reč deskriptora stranice izabrane za zamenu.

```
uint64* getVictim () {
  while (clockHand) {
    uint16* pCnt = ((uint16*)clockHand) + 3;
    if (*pCnt == 0) // Found the victim!
      return clockHand-1;
    (*pCnt)--; // Decrement the counter
    // Move the clock hand to the next:
    uint64 next = *clockHand;
    next |= ~(uint64)0xffffffUL;
    clockHand = (uint64*)next;
  }
  return 0; // No pages or another exception
}
```

Zadatak – 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.

Zadatak – Decembar 2016.

Neki sistem koristi rezervoar (engl. pool) slobodnih okvira radi ubrzanja obrade straničnih grešaka. Pri tom se za svaki oslobođeni okvir koji se smešta u rezervoar čuva informacija o tome kom procesu i kojoj njegovoj stranici je taj okvir pripadao. Prilikom alokacije okvira za traženu stranicu, sistem najpre pokušava da nađe okvir u kome je upravo tražena stranica tog procesa bila smeštena, i ako se takav okvir u rezervoaru pronađe, on se alocira. U suprotnom, alocira se bilo koji drugi slobodan okvir iz rezervoara.
Evidencija slobodnih okcira u rezervoaru vodi se u nizu `freeFramePool` veličine `FreeFramePoolSize` (to je maksimalna broj okvira u rezervoaru) deskriptora `FreeFrameDescr`. Element ovog niza može biti „prazan“ (`isSlotFree==true`) ili „zauzet“, u kom slučaju informacije o njemu govore kom procesu i kojoj njegovoj stranici je pripadao dati okvir referenciran tim deskriptorom.

```
typedef uint unsigned int;
struct FreeFrameDescr {
  ushort isSlotFree; // Is this slot in the pool's array
free?
  uint frame; // The frame referenced by this descriptor
  uint pid; // PID of the process that owned (released) this
frame
  uint page; // The page of the process that owned this frame
};
FreeFrameDescr freeFramePool[FreeFramePoolSize]; // The pool
int getFreeFrame (uint pid, uint page, uint* frame);
```

Implementirati funkciju `getFreeFrame` koja treba da pronađe slobodan okvir za tražen stranicu `page` datog procesa `pid` i njegov broj upiše u lokaciju na koju ukazuje argument `frame`. Ako nađe baš okvir koji je pripadao toj stranici, ova funkcija treba da vrati 1, ako nađe neki drugi slobodan okvir, treba da vrati 0, a ako slobodnih okvira u rezervoaru nema, treba da vrati -1.

```
int getFreeFrame (uint pid, uint page, uint* frame) {
  int cur = 0;
  for (cur=0; cur<FreeFramePoolSize; cur++) {
    if (!freeFramePool[cur].isSlotFree &&
       freeFramePool[cur].pid==pid && freeFramePool[cur].page==page) {
      // The frame of the same page found, return it and declare as free:
      *frame = freeFramePool[cur].frame;
      freeFramePool[cur].isSlotFree = 1;
      return 1;
    }
  }

  for (cur=0; cur<FreeFramePoolSize; cur++) {
    if   (!freeFramePool[cur].isSlotFree) {
      // Another free frame found, return it and declare as free:
      *frame = freeFramePool[cur].frame;
      freeFramePool[cur].isSlotFree = 1;
      return 0;
    }
  }

  // No free frame in the pool!
  return -1;
}
```

Zadatak - 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.

```
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 – Septembar 2015.

Radi sprečavanja pojave zvane trashing, neki sistem prati učestanost strničnih grešaka za svaki proces na sledeći način. Za svaki proces broje se stranične greške u svakoj periodi (intervalu vremena) i pamte se ti brojevi za ukupno `PFLTCOUNTERS` poslednjih perioda.
U PCB svakog procesa postoji niz `pageFaultCounters` sa `PFLTCOUNTERS` elemenata tipa `unsigned` koji predstavlja kružni bafer brojača straničnih grešaka po periodama. Na brojač koji odgovara tekućoj periodi ukazuje polje `pageFaultCursor` tipa `int` u opsegu od `0` do `PFLTCOUNTERS-1`. Kada istekne data perioda, sistem pomera kurzor na sledeću poziciju u kružnom baferu, tako da elementi ovog bafera uvek čuvaju brojeve straničnih prekida poslednjih `PFLTCOUNTERS` perioda.
Implementirati sledeće operacije:

- `void incPageFaultCounter(PCB* pcb)`: za proces sa datim PCB-om inkrementira brojač straničnih grešaka za tekuću periodu; ova operacija poziva se prilikom svake stranične greške datog procesa;
- `void shiftPageFaultCounters(PCB* pcb)`: za proces sa datim PCB-om pomera kurzor u kružnom baferu; ova operacija poziva se periodično, na isteku svake periode, za svaki proces;

• `unsigned getNumberOfPageFaults(PCB* pcb)`: za proces sa datim PCB-om vraća ukupan broj straničnih grešaka u poslednjih `PFLTCOUNTERS` perioda; ova operacija poziva se prilikom provere pojave trashing za dati proces.

```
void incPageFaultCounter (PCB* pcb) {
  if (pcb==0) return; // Exception!
  pcb->pageFaultCounters[pcb->pageFaultCursor]++;
}
void shiftPageFaultCounters (PCB* pcb) {
  if (pcb==0) return; // Exception!
  pcb->pageFaultCursor++;
  if (pcb->pageFaultCursor>=PFLTCOUNTERS)
    pcb->pageFaultCursor = 0;
  pcb->pageFaultCounters[pcb->pageFaultCursor] = 0;
}
```

• `unsigned getNumberOfPageFaults(PCB* pcb)`: za proces sa datim PCB-om vraća ukupan broj straničnih grešaka u poslednjih `PFLTCOUNTERS` perioda; ova operacija poziva se prilikom provere pojave trashing za dati proces.

```
void incPageFaultCounter (PCB* pcb) {
  if (pcb==0) return; // Exception!
  pcb->pageFaultCounters[pcb->pageFaultCursor]++;
}
void shiftPageFaultCounters (PCB* pcb) {
  if (pcb==0) return; // Exception!
  pcb->pageFaultCursor++;
  if (pcb->pageFaultCursor>=PFLTCOUNTERS)
    pcb->pageFaultCursor = 0;
  pcb->pageFaultCounters[pcb->pageFaultCursor] = 0;
}
```

```
unsigned getNumberOfPageFaults (PCB* pcb) {
  if (pcb==0) return; // Exception!
  unsigned sum = 0;
  for (int i=0; i<PFLTCOUNTERS; i++)
    sum += pcb->pageFaultCounters[i];
  return sum;
}
```

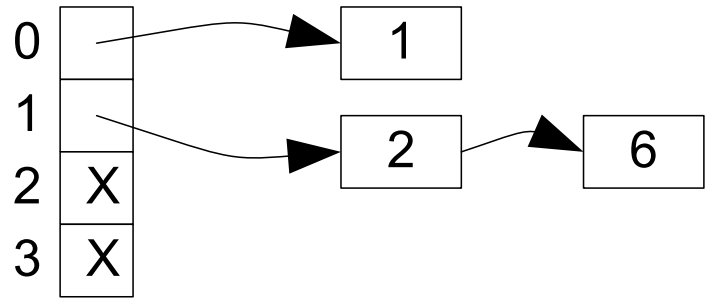
Zadatak – Septembar 2009.

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.

Rešenje:

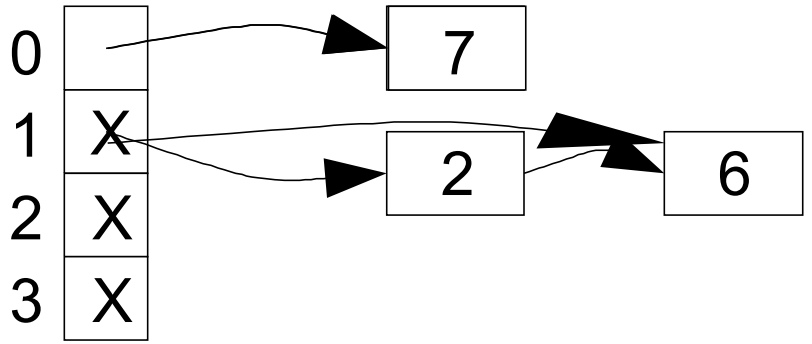
Zadatak – Septembar 2010.

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.

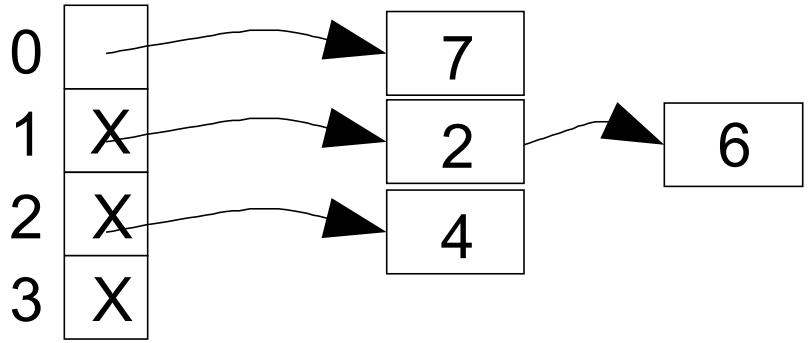


Nacrtati izgled ove strukture nakon što je, za prikazano početno stanje:

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


0
1
2
3
X
X
X
7
2
6

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.



Zadatak – Decembar 2017.

Neki sistem koristi sistem parnjaka (engl. *buddy*) za alokaciju memorije. Adresibilna jedinica je bajt, a sistem parnjaka upravlja delom operativne memorije počev od adrese A00000h i veličine 16 stranica, sa stranicom od 4 KB kao najmanjom jedinicom alokacije.

Trenutno su slobodni sledeći blokovi memorije (data je adresa i veličina bloka u stranicama; sve vrednosti su heksadecimalne):
A06000/2, A08000/2, A0C000/4.

a)(2) Popuniti drugu kolonu sledeće tabele navodeći početne adrese slobodnih delova memorije odgovarajuće veličine koji su uvezani u listu u datom ulazu tabele, u skladu sa implementacijom strukture za potrebe ovog algoritma.

A06000/2, A08000/2, A0C000/4.

0	
1	A06000/2, A08000/2
2	A0C000/4
3	
4	

b)(4) U stanju iz tačke a) izvršava se operacija alokacije bloka memorije veličine 4 KB. Popuniti tabelu za stanje strukture nakon izvršavanja te operacije alokacije.

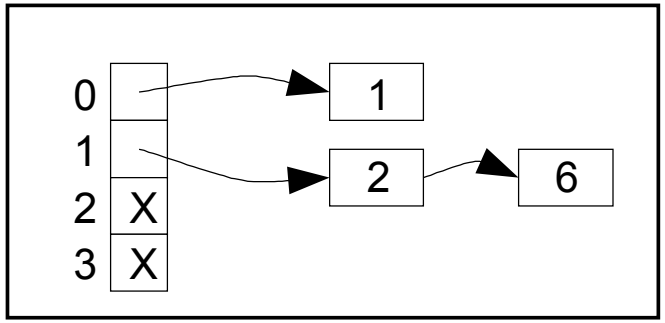
0	A07000/1
1	A06000/2, A08000/2
2	A0C000/4
3	
4	

c)(4) U stanju iz tačke b) izvršava se operacija dealokacije dela memorije veličine 8 KB počev od adrese A0A000h. Popuniti tabelu za stanje strukture nakon izvršavanja te operacije dealokacije.

0	A07000/1
1	A08000/2, A0A000/2
2	A08000/4, A0C000/4
3	A08000/8
4	

Zadatak – 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);
```

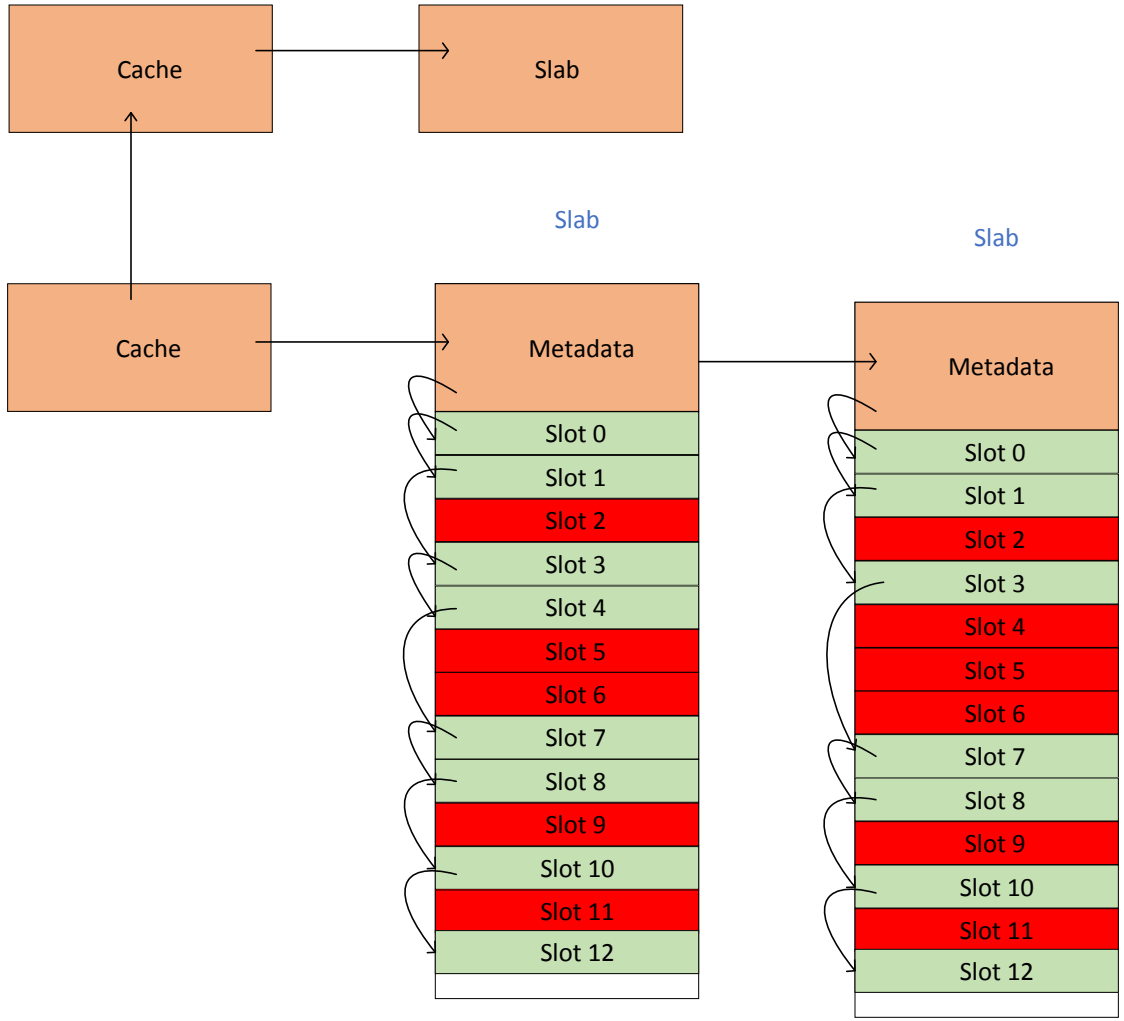
Realizovati funkciju:

```
void buddy_init ();
```

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

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

Slab alokator



Zadatak – 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.

```
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 – 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.
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.

```
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;
}
```

```
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;
}
```