Sveučilište J. J. Strossmayera u Osijeku

Fakultet elektrotehnike, računarstva i informacijskih tehnologija

Kneza Trpimira 2b

HR-31000 Osijek

www.ferit.unios.hr

Laboratorijska vježba 6:

Upravljanje memorijom unutar operacijskog sustava

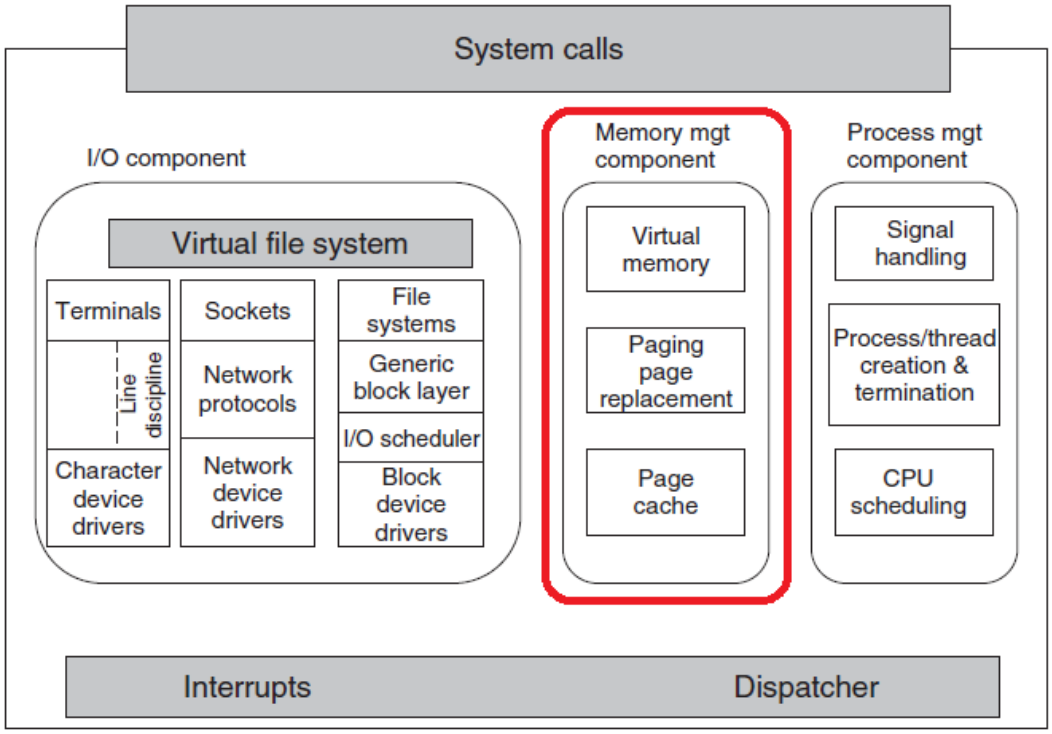
Sadržaj

I. UVOD

U operacijskom sustavu koji podržava samo jedan proces (engl. *uniprogramming system*) u memoriji, glavna memorija računala u nekom trenutku sadrži samo operacijski sustav i korisnički dio programa koji se izvršava. Operacijski sustavi koji podržavaju više procesa (engl. *multiprogramming systems*) u memoriji, moraju dodatno podijeliti glavnu memoriju za smještanje više procesa. Zadatak raspodjele glavne memorije obavlja operacijski sustav, tj. njegova komponenta zadužena za **upravljanje memorijom**.

Efektivno upravljanje memorijom ključno je za svaki *multiprogramming* operacijski sustav. Ako je malo procesa u memoriji, procesor će većinu vremena čekati na ulazno-izlazne jedinice. Stoga je poželjno smjestiti što veći broj procesa u glavnu memoriju, kako bi se izbjeglo čekanje na I/O. No, u današnjim operacijskim sustavima svi procesi se ne mogu istovremeno postaviti u glavnu memoriju, pa dolazi do preopterećenja memorije (engl. *memory overload*). Tijekom godina razvijene su dvije opće strategije za rješavanje preopterećenja memorije. Jednostavnija strategija se naziva **izmjena** (engl. *swapping*), a sastoji se od premještanja cjelokupnih procesa između glavne memorije i tvrdog diska. Procesi koji su blokirani ili čekaju se najčešće premještaju na disk. Druga strategija se naziva **virtualna memorija** (engl. *virtual memory*) i dozvoljava izvršavanje programa ako su samo djelomično u glavnoj memoriji.

U ovoj laboratorijskoj vježbi ćemo izložiti teorijsku podlogu upravljanja memorijom u današnjim operacijskim sustavima. Detaljnije ćemo proučiti algoritme za dodjeljivanje memorije i zamjenu stranica te prikazati njihov rad na zadanim skupovima procesa i memorijskih blokova. U konačnici, implementirati ćemo jedan od algoritama za dodjeljivanje memorije u programskom jeziku C. U zadatku za samostalni rad, potrebno je izraditi implementaciju preostalih algoritama na sličan način.



Slika 1. Struktura Linux jezgre

1.1. Particioniranje memorije

Temeljni faktor o kojem ovisi tijek izmjene procesa je veličina particije u koju je moguće smjestiti proces u glavnoj memoriji. Tehnika kojom se glavna memorija dijeli na manje dijelove se naziva **particioniranje** (engl. *partitioning*) memorije. Postoje dva glavna oblika particioniranja - fiksno i dinamičko, a oba mogu rezultirati određenim problemima.

1.1.1. Fiksno particioniranje memorije

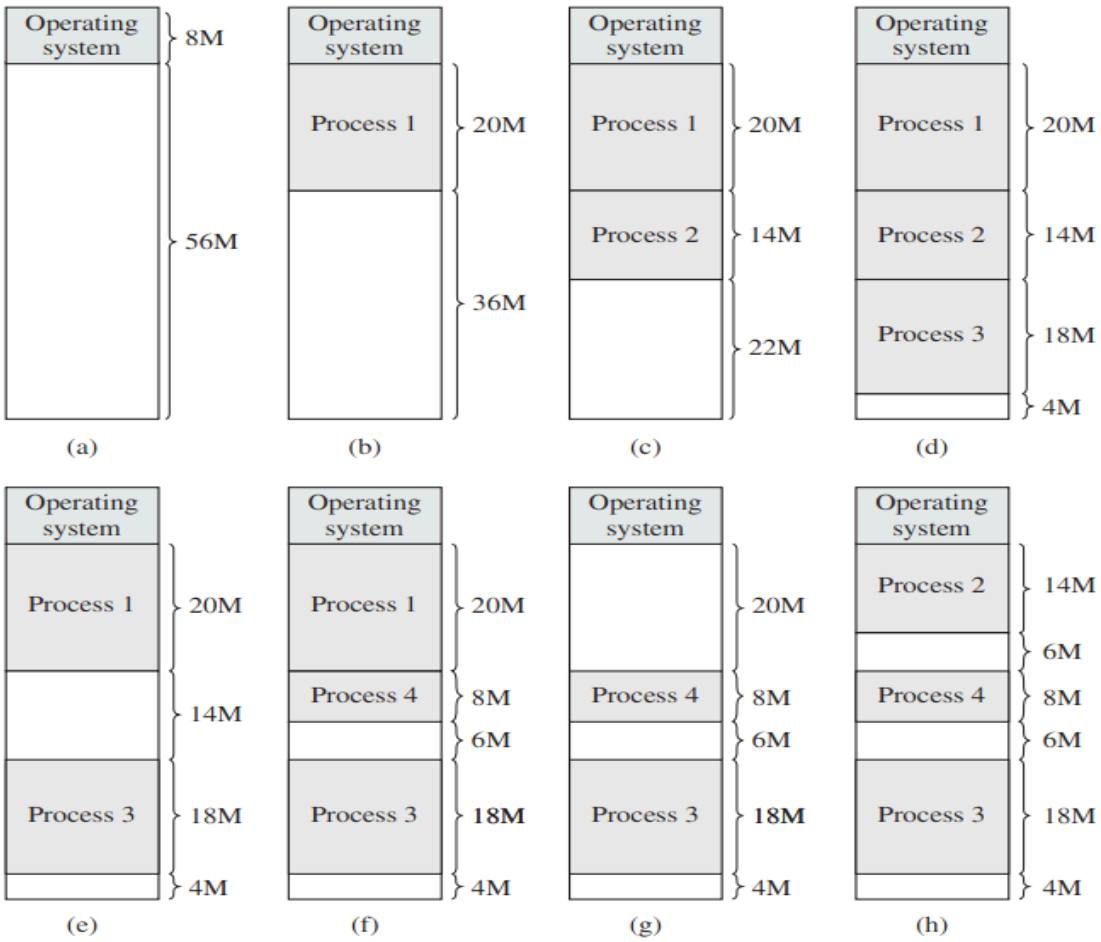
U većini shema upravljanja memorijom, možemo pretpostaviti da operacijski sustav zauzima neki fiksni dio glavne memorije, a ostatak je dostupan za procese. Najjednostavnija tehnika kojom se to postiže je particioniranje memorije u regije (particije) s fiksnim granicama. Jedna mogućnost je stvaranje regija jednakih veličina. U tom slučaju, svaki proces čija je veličina manja ili jednaka veličini particije smješta se u bilo koju particiju. Nedostaci ovog pristupa su:

- Program je prevelik za particiju - u tom slučaju, program mora biti implementiran tako da radi ako mu je samo jedan dio u glavnoj memoriji.
- Iskorištenje glavne memorije je neefikasno - bilo koji program okupira particiju, ako u nju stane. Primjerice, program od 2MB okupira cijelu particiju od 8MB, a 6MB ostaje neiskorišteno. Ovaj fenomen, u kojem dio memorije ostaje neiskorišten i blokiran za korištenje se nazva unutarnjom fragmentacijom (engl. *internal fragmentation*).

Upotreba particija nejednakih veličina može ublažiti spomenute probleme, ali i ne ih riješiti do kraja.

1.1.2. Dinamičko particioniranje

Kako bi se izbjegle određene poteškoće prilikom fiksnog particioniranja, razvijen je postupak dinamičkog particioniranja, u kojem su particije varijabilne duljine i broja. Kad proces dolazi u glavnu memoriju, iskorištava točnu količinu memorije koliko mu je potrebno i ne više od toga. U ovom načinu particioniranja, operacijski sustav neprestano izmjenjuje procese između particija, kako bi se stvorila particija dovoljne veličine za novi proces. S vremenom, u memoriji nastaje sve više "rupa", a efikasnost iskorištenja pada. Ovaj fenomen se naziva **eksternom fragmentacijom** (engl. *external fragmentation*), a prikazan je na slici 2.



Slika 2. Efekt dinamičkog particioniranja

Tehnika za savladavanje eksterne fragmentacije je zbijanje (engl. *compaction*), kojom operacijski sustav pomiče procese u memoriji tako da postaju slijedni, a slobodna memorija čini jedan veći blok.

1.2. Dodjeljivanje memorije

S obzirom da je zbijanje memorije vremenski zahtjevno, operacijski sustav na pametan način mora dodijeliti procese memoriji. Jedan od načina strukturiranja memorije je održavanje povezanog popisa dodijeljenih i slobodnih memorijskih segmenata. Dodijeljeni memorijski segmenti sadrže procese, dok prazni segmenti predstavljaju rupu između dva procesa. Unutar povezanog popisa, segmenti su sortirani prema adresama. Upravitelj memorije ima informaciju koliko memorije treba alocirati za svaki novi proces. Prilikom učitavanja novog procesa u memoriju, postoji više slobodnih memorijskih blokova u koje može biti smješten. Operacijski sustav koristi algoritme za dodjeljivanje memorije kako bi utvrdio najprikladniji slobodan blok memorije za učitavanje novog procesa. Najpoznatiji algoritmi za dodjeljivanje memorije su: ***First-fit***, ***Best-fit*** i ***Worst-fit***.

1.2.1. *First-fit* algoritam

Najjednostavniji algoritam dodjeljivanja memorije je *First-fit* algoritam. Upravitelj memorije skenira listu memorijskih segmenata sve dok ne pronađe prazni segment koji je dovoljno velik da se u njega smjesti novi proces. Tada se odabrani memorijski segment dijeli na dva dijela -

jedan za proces i drugi za neiskorištenu memoriju, osim u slučaju da veličina procesa točno odgovara veličini odabranog memorijskog segmenta. Algoritam *First-fit* je brz algoritam, jer najkraće pretražuje povezani popis.

Primjer 1: Dodjela memorije po First-fit algoritmu

Operacijski sustav analizirao je memoriju računala i dao na raspolaganje 5 praznih memorijskih blokova sustavu za dodjeljivanje memorije procesima (SDMP). Ukupno 10 procesa zahtijeva memorijski prostor, dok SDMP vrši dodjelu prema „First-fit" principu. U tablicama su zadane veličine memorijskih blokova te zahtjevi procesa. Skicirajte način dodjeljivanja zadanih procesa u dostupne memorijske blokove.

Proces	Tražena memorija	Trajanje
Proces 1	**100k**	**3**
Proces 2	**10k**	**1**
Proces 3	**35k**	**2**
Proces 4	**15k**	**1**
Proces 5	**23k**	**2**
Proces 6	**6k**	**1**
Proces 7	**25k**	**1**
Proces 8	**55k**	**2**
Proces 9	**88k**	**3**
Proces 10	**100k**	**3**

Blok memorije	Veličina
Blok 1	50k
Blok 2	200k
Blok 3	70k
Blok 4	115k
Blok 5	15k

Rješenje:

Blok memorije	Veličina	1.	2.	3.	4.	5.	6.
Blok 1	50k	P2	P5	P5			
Blok 2	200k	P1	P1	P1	P10	P10	P10
Blok 3	70k	P3	P3	P8	P8		
Blok 4	115k	P4	P7	P9	P9	P9	
Blok 5	15k	P6					

1.2.2. *Best-fit* algoritam

Jedan od najpoznatijih algoritama za dodjeljivanje memorije je *Best-fit* algoritam, koji dodjeljuje proces najmanjem dostupnom bloku memorije. Prema tom algoritmu, upravitelj memorije pretražuje cijeli povezani popis memorijskih segmenata i odabire najmanji memorijski segment, dovoljno velik da se u njega smjesti novi proces. U odnosu na *First-fit* algoritam, *Best-fit* nastoji smanjiti rupe koje nastaju pri popunjavanju segmenata, tako da se smanji udio neiskorištene memorije. Međutim, *Best-fit* algoritam je sporiji jer zahtijeva pretraživanje cijele liste memorijskih segmenata za svaki novi proces. Osim toga, u praksi se pokazalo da *Best-fit* algoritam u konačnici ostavlja više rupa u memoriji, jer su neiskorišteni dijelovi memorijskih segmenata nedovoljne veličine za smještanje novih procesa.

Primjer 2: Dodjela memorije po *Best-fit* algoritmu

Operacijski sustav analizirao je memoriju računala i dao na raspolaganje 5 praznih memorijskih blokova sustavu za dodjeljivanje memorije procesima (SDMP). Ukupno 10 procesa zahtijeva memorijski prostor, dok SDMP vrši dodjelu prema „*Best-fit*" principu. U tablicama su zadane veličine memorijskih blokova te zahtjevi procesa. Skicirajte način dodjeljivanja zadanih procesa u dostupne memorijske blokove.

Proces	Tražena memorija	Trajanje
Proces 1	**100k**	**3**
Proces 2	**10k**	**1**
Proces 3	**35k**	**2**
Proces 4	**15k**	**1**
Proces 5	**23k**	**2**
Proces 6	**6k**	**1**
Proces 7	**25k**	**1**
Proces 8	**55k**	**2**
Proces 9	**88k**	**3**
Proces 10	**100k**	**3**

Blok memorije	Veličina
Blok 1	50k
Blok 2	200k
Blok 3	70k
Blok 4	115k
Blok 5	15k

Rješenje:

Blok memorije	Veličina	1.	2.	3.	4.	5.	6.
Blok 1	50k	P3	P3				
Blok 2	200k	P5	P5	P9	P9	P9	
Blok 3	70k	P4	P7	P8	P8		
Blok 4	115k	P1	P1	P1	P10	P10	P10
Blok 5	15k	P2	P6				

1.2.3. *Worst-fit* algoritam

U pokušaju rješavanja problema nastajanja premalih memorijskih segmenata, uzrokovanog poglavito zbog dodjeljivanja procesa u segmente najprikladnije veličine, kao ideja je nastao *Worst-fit* algoritam. Prema tom algoritmu, novi proces se smješta u najveći prazni memorijski segment u cijelom povezanom popisu. Na taj način, veća je vjerojatnost da će neiskorišteni dio tog memorijskog segmenta biti dovoljno velik za smještanje novog procesa u idućim koracima.

Primjer 3: Dodjela memorije po *Worst-fit* algoritmu
Operacijski sustav analizirao je memoriju računala i dao na raspolaganje 5 praznih memorijskih blokova sustavu za dodjeljivanje memorije procesima (SDMP). Ukupno 10 procesa zahtijeva memorijski prostor, dok SDMP vrši dodjelu prema „*Worst-fit*" principu. U tablicama su zadane veličine memorijskih blokova te zahtjevi procesa. Skicirajte način dodjeljivanja zadanih procesa u dostupne memorijske blokove.

Proces	Tražena memorija	Trajanje
Proces 1	**100k**	**3**
Proces 2	**10k**	**1**
Proces 3	**35k**	**2**
Proces 4	**15k**	**1**
Proces 5	**23k**	**2**
Proces 6	**6k**	**1**
Proces 7	**25k**	**1**
Proces 8	**55k**	**2**
Proces 9	**88k**	**3**
Proces 10	**100k**	**3**

Blok memorije	Veličina
Blok 1	50k
Blok 2	200k
Blok 3	70k
Blok 4	115k
Blok 5	15k

Rješenje:

Blok memorije	Veličina	1.	2.	3.	4.	5.	6.
Blok 1	50k	P4	P7				
Blok 2	200k	P1	P1	P1	P9	P9	P9
Blok 3	70k	P3	P3	P8	P8		
Blok 4	115k	P2	P5	P5	P10	P10	P10
Blok 5	15k	P6					

1.3. Programska implementacija algoritama za dodjeljivanje memorije

U ovom poglavlju implementirati ćemo jedan od prikazanih algoritama za dodjeljivanje memorije u programskom jeziku C. Odabrani algoritam je *Best-fit*. Iako se svi algoritmi razlikuju u logici, većina koristi jednake strukture podataka te ispisuje jednaku listu memorijskih particija.

1.3.1. Strukture podataka

Algoritmi za dodjeljivanje memorije prvo primaju podatke o procesima za izvođenje i o dostupnim memorijskim blokovima. Od ulaznih podataka, potrebno je pohraniti sljedeće:

- Polje veličina memorijskih blokova (engl. *blockSize*)
- Polje veličina procesa (engl. *processSize*)

Izlaz svakog algoritma je u obliku povezanog popisa koji za svaki proces ispisuje broj memorijskog bloka kojemu je taj proces dodijeljen. Ti podaci se spremaju u polje alokacija (engl. *allocation*).

Primjer 4: Strukture podataka za algoritam *Best-fit*

```
#include <stdio.h>
#include <stdlib.h>

int main() {
 int n,m,i,j;
 printf("Unesite broj procesa\n");
 scanf("%d", &n);
 printf("Unesite broj memorijskih blokova\n");
 scanf("%d", &m);

 int *processSize = (int*) calloc(n, sizeof(int));
 int *blockSize = (int*) calloc(m, sizeof(int));

 int *allocation = (int*) calloc(n, sizeof(int));
 for(i=0; i<n; i++){
  allocation[i] = -1;
 }
 for (i=0; i<n; i++)
  scanf("%d", &processSize[i]);

 for (i=0; i<m; i++)
  scanf("%d", &blockSize[i]);
```

1.3.2. Izlaz programa

Izlaz programa prikazan u primjeru 5, nije posve nalik rješenjima u prethodnim primjerima, jer ova programska implementacija ne uzima u obzir trajanja procesa. Za svaki proces se ispisuju njegov pid, veličina i broj dodijeljenog memorijskog bloka. Slika 3 prikazuje izgled izlaza programa.

```
Process PID        Process size          Block number
 1                       100                    4
 2                       10                     4
 3                       35                     1
 4                       15                     1
 5                       23                     3
 6                       6                      5
 7                       25                     3
 8                       55                     2
 9                       88                     2
 10                      100                    1
```

Slika 3. Izlaz programske implementacije algoritma *Best-fit*

Primjer 5: Izlaz programa za algoritam *Best-fit*

```
 printf("\nProcess PID \t Process size \t Block number\n");
 for (i=0; i<n; i++){
  printf(" %d \t\t\t %d \t\t\t", i+1, processSize[i]);
  if (allocation[i] != -1)
   printf("%d\n", allocation[i] +1);
  else
   printf("Not allocated\n");
 }
```

1.3.3. Logika algoritma

Izvođenje algoritma *Best-fit* prikazano je u primjeru 2. Algoritam odabire memorijski blok najmanje dovoljne veličine za smještanje nadolazećeg procesa. U implementaciji prikazanoj u primjeru 6, stvorena je *for* petlja koja prolazi kroz listu svih veličina procesa. Za svaki proces odabire se indeks memorijskog bloka u koji će se on smjestiti. Na početku se indeks postavlja na vrijednost -1, jer nije sigurno da postoji memorijski blok dovoljno velik za smještanje procesa. Nadalje se novom *for* petljom prolazi kroz veličine svih memorijskih blokova i za svaki ispituje je li veći ili jednak veličini trenutnog procesa. Ako je nađen takav memorijski blok, njegov indeks se sprema u zasebnu varijablu. Za svaki idući memorijski blok se također provjerava isti uvjet te je li njegova veličina manja od spremljenog memorijskog bloka. Prolaskom kroz sve memorijske blokove dobiva se onaj s najmanjom dovoljnom veličinom za smještanje procesa i njegov indeks je spremljen u varijabli *bestBlockIndex*.

Primjer 6: Logika algoritma *Best-fit*

```
 for (i=0; i<n; i++){
  int bestBlockIndex = -1; //Ne moze biti 0, jer nismo sigurni da postoji dovoljno velik
memorijski blok
  for (j=0; j<m; j++){
   if (blockSize[j] >= processSize[i]) {
    if (bestBlockIndex == -1)
     bestBlockIndex = j;
    else if (blockSize[j] < blockSize[bestBlockIndex])
     bestBlockIndex = j;
   }
  }

  if (bestBlockIndex != -1){ //Nađen je odgovarajući memorijski blok
   allocation[i] = bestBlockIndex;
   blockSize[bestBlockIndex] -= processSize[i];
  }
 }
```

2. VIRTUALNA MEMORIJA

Temeljna zadaća upravljanja memorijom je smještanje procesa u glavnu memoriju računala gdje ih čita procesor prilikom izvršavanja. U gotovo svim modernim operacijskim sustavima ova zadaća se provodi pomoću **virtualne memorije** (engl. *virtual memory*). Ona označava sposobnost operacijskog sustava da koristi hardver i softver u svrhu kompenziranja nedostataka fizičke memorije, privremeno premještajući podatke iz glavne memorije na disk. Za razliku od izmjene procesa, gdje je potrebno premjestiti čitavi program iz glavne memorije na disk i obratno, virtualna memorija je zasnovana na mogućnosti izvršavanja programa ako je samo njegov jedan dio smješten u glavnu memoriju. Virtualna memorija se temelji na dvjema tehnikama upravljanja memorijom: segmentaciji (engl. *segmentation*) i straničenju (engl. *paging*).

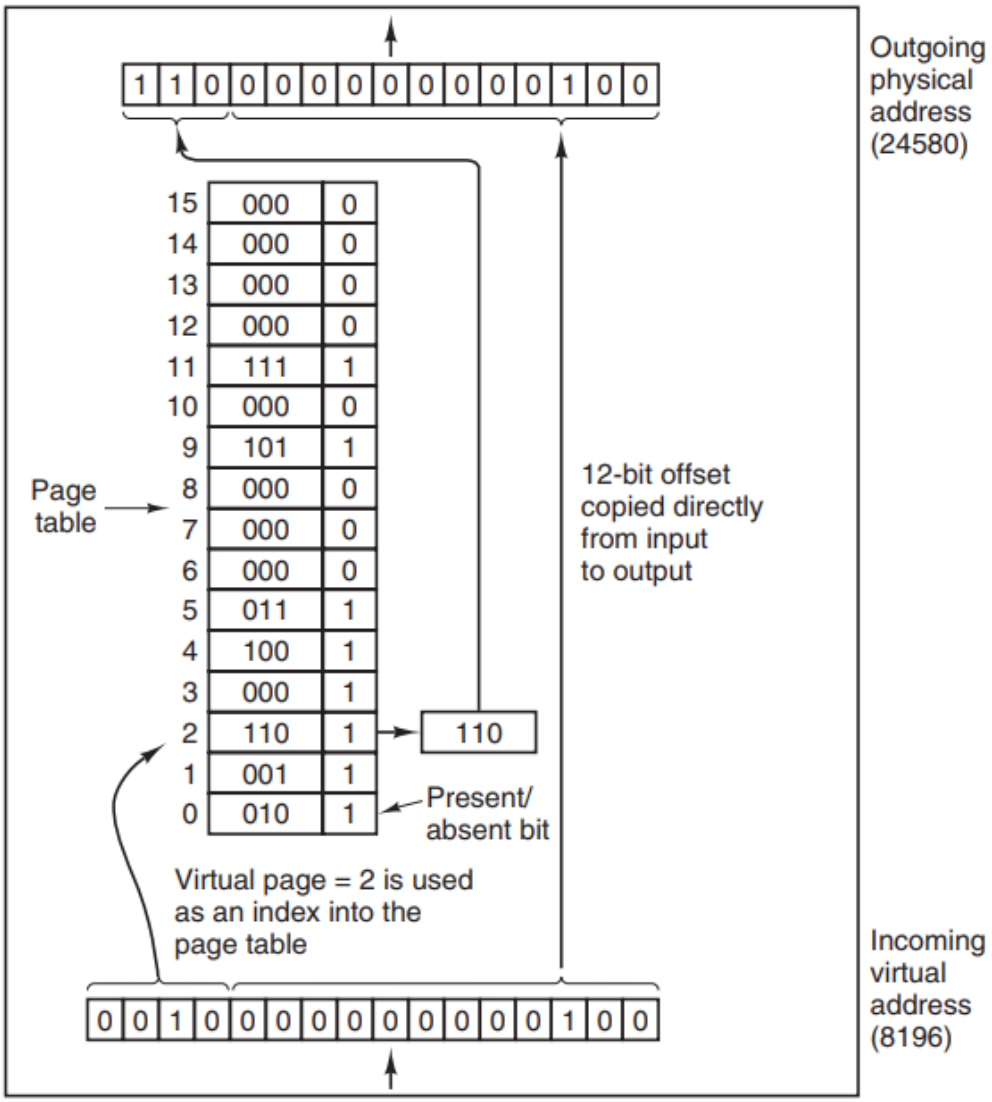
2.1. Straničenje

Fiksno i dinamičko particioniranje memorije su neefikasni u iskorištenju memorije, jer rezultiraju unutarnjom, tj. eksternom fragmentacijom. Stoga je razvijen postupak particioniranja memorije koji dijeli memoriju na relativno male dijelove fiksne veličine, a svaki proces se također dijeli u manje dijelove iste veličine. Tako podijeljeni dijelovi procesa se nazivaju **stranice** (engl. *pages*), a mogu se dodijeliti u dostupne dijelove memorije zvane **okviri** (engl. *frames*). U ovom načinu ne dolazi do eksterne fragmentacije, a stupanj unutarnje fragmentacije je znatno smanjen.

S obzirom da stranice procesa mogu biti smještene na različita mjesta u memoriji, operacijski sustav održava tzv. **tablicu stranica** (engl. *page table*) za svaki proces. Svaka tablica stranica pokazuje lokaciju memorijskog okvira za svaku stranicu procesa. Unutar programa, svaka logička adresa se sastoji od broja stranice i pomaka (engl. *offset*) unutar stranice. Zadaća procesora je translatirati logičku adresu u fizičku adresu tako da pročita zapis iz tablice stranica. Kada program referencira dio adresnog prostora koji je u fizičkoj memoriji, procesor jednostavno mapira adresu s fizičkom lokacijom. Ukoliko program referencira adresu koja ne postoji u fizičkoj memoriji, operacijski sustav vrši mapiranje na temelju tablice stranica.

2.1.1. Tablice stranica

Pri mapiranju virtualne adrese u fizičku, ona se dijeli na virtualni broj stranice i pomak. Primjerice, za 16-bitnu adresu i stranicu veličine 4KB, gornja 4 bita predstavljaju jednu od 16 virtualnih stranica, a donjih 12 bita specificira pomak u bajtovima (0 do 4095) unutar odabrane stranice. Broj virtualne stranice se koristi kao indeks u tablici stranica za lakši pronalazak te stranice. Unutar retka u tablici stranica sadržan je viši dio pomaka, koji zajedno s onih 12 bita čini fizičku adresu u memoriji.



Slika 4. Postupak čitanja tablice stranica

Točna struktura retka unutar tablice stranice ovisi o računalnom sklopovlju, no sadržane informacije su gotovo jednake u svim implementacijama. Veličina retka također ovisi o računalu, a 32 bita je uobičajena veličina. Uobičajena polja koja sadrži jedan redak u tablici stranica su:

- *Page frame number* - broj okvira u koji je smještana stranica
- *Present/absent bit* - oznaka je li stranica prisutna u memoriji (1) ili nije (0)
- *Protection* - definira tip pristupa (0 - *read/write*, 1 - *read*)
- *Modified* - vrijeme zadnjeg ažuriranja stranice
- *Referenced* - vrijeme zadnjeg referenciranja stranice
- *Caching disabled* - oznaka je li moguće smjestiti stranicu u pričuvnu memoriju ili ne

2.2. Algoritmi zamjene stranica

S obzirom da virtualna memorija dozvoljava izvršavanje procesa, iako nije cijeli sadržan u memoriji, može doći do problema greške stranice (engl. *page fault*), tj. da tražena stranica nije u memoriji. Tada se ta stranica mora alocirati u fizičku memoriju, ukoliko u njoj ima dovoljno

prostora. **Zamjena stranice** (engl. *page replacement*) se događa kada tražena stranica nije u memoriji, a nema slobodnih mjesta za njeno alociranje. U tom slučaju, odabire se neka od postojećih stranica. Pri tome, cilj je odabrati stranicu za koju je manje vjerojatno da će se ubrzo koristiti, kako bi se izbjeglo ponovno vraćanje stranice u fizičku memoriju. Mnogo teorijskog i eksperimentalnog istraživanja je provedeno u svrhu dizajniranja optimalnog algoritma zamjene stranica. Najbolji mogući algoritam je *Optimal page replacement* algoritam, koji nije izvediv u praksi. Osim njega, najpoznatiji algoritmi još su: *First-In, First-Out (FIFO) page replacement* i *Least Recently Used (LRU) page replacement*.

2.2.1. *Optimal page replacement* algoritam

Najbolji mogući algoritam zamjene stranica je jednostavan, ali nemoguć za implementaciju. Ako se pojavi potreba za zamjenom stranica, upravitelj memorije uklanja iz fizičke memorije onu stranicu koja će se zadnja koristiti. Primjerice, ako je za svaku stranicu u tablici stranica poznat broj instrukcija nakon kojih će se ta stranica koristiti, uklanja se ona stranica s najvećim takvim brojem. Međutim, operacijski sustav ne može unaprijed znati za sve stranice u tablici kada će se ponovno koristiti. Ovaj algoritam se često koristi uz simulator procesa, koji pokušava simulirati izvođenje i na temelju kojeg se generiraju vjerojatnosti korištenja u stvarnosti.

Primjer 7: Zamjena stranica u tablici stranica prema *Optimal page replacement* algoritmu

Zadan je referentni string koji treba pohraniti u memoriju koristeći OPR algoritam: a, b, c, d, c, a, d, b, e, b, a, b, c, d. Veličina okvira (broj stranica) je 4. Napišite postupak upisivanja referentno stringa u memoriju i označite svaki "Page fault".

Rješenje:

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
a	b	c	d	c	a	d	b	e	b	a	b	c	d
a	a	a	a	a	a	a	a	a	a	a	a	a	d
	b	b	b	b	b	b	b	b	b	b	b	b	b
		c	c	c	c	c	c	c	c	c	c	c	c
			d	d	d	d	d	e	e	e	e	e	e
X	X	X	X					X					X

Ukupno je nastalo 6 *Page fault* događaja.

2.2.2. *First-In, First-Out (FIFO) page replacement* algoritam

Jednostavan algoritam za zamjenu stranica, koji je izvediv u praksi je *First-In, First-Out (FIFO) page replacement* algoritam. Operacijski sustav održava listu svih stranica u memoriji, tako da je najnovija stranica na kraju liste, a najstarija na čelu liste. U slučaju greške stranice, uklanja se stranica na čelu, a nova stranica se dodaje na kraj liste. Drugim riječima, u slučaju potrebe za zamjenom stranica, iz memorije se uklanja ona stranica koja je i prva postavljena u memoriju. Problem ovog algoritma je što ne provjerava je li ta najstarija stranica još uvijek potrebna nekom drugom procesu. S obzirom da je najduže u memoriji, moguće je da je često korištena i da je još uvijek potrebna.

Primjer 8: Zamjena stranica u tablici stranica prema *The First-In, First-Out (FIFO) page replacement* algoritmu

Zadan je referentni string koji treba pohraniti u memoriju koristeći FIFO algoritam: a, b, c, d, c, a, d, b, e, b, a, b, c, d. Veličina okvira (broj stranica) je 4. Napišite postupak upisivanja referentno stringa u memoriju i označite svaki "Page fault".

Rješenje:

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
a	b	c	d	c	a	d	b	e	b	a	b	c	d
a	a	a	a	a	a	a	a	e	e	e	e	e	d
	b	b	b	b	b	b	b	b	b	a	a	a	a
		c	c	c	c	c	c	c	c	c	b	b	b
			d	d	d	d	d	d	d	d	d	c	c
X	X	X	X					X		X	X	X	X

Ukupno je nastalo 9 *Page fault* događaja.

2.2.3. *Least Recently Used page replacement* algoritam

Kao što je prikazano u strukturi tablice stranica, svaki redak sadrži bitove koji označavaju vrijeme zadnje izmjene i referenciranja. Kada proces započinje s izvršavanjem, niti jedna stranica procesa nije u memoriji. Ubrzo dolazi do referenciranja jedne njegove stranice, događa se greška stranice (*page fault*) i ona se smješta u memoriju te joj se postavlja bit za referenciranje. Ukoliko se pri izvođenju ta ista stranica ažurira, također se događa greška stranice i postavlja se bit za ažuriranje. Ti bitovi se koriste pri dizajnu *Least Recently Used page replacement* algoritma, koji se još naziva i *Not Recently Used (NRU)* algoritam. Prema tom algoritmu, prilikom zamjene stranica, uklanja se ona stranica iz memorije koja je najranije ažurirana ili referencirana. Algoritam je zasnovan na pretpostavci da će se stranice koje su se zadnje koristile vjerojatno nastaviti koristiti, dok je za stranice koje su najduže neiskorištene korištenje manje vjerojatno. *Least Recently Used page replacement* algoritam daje adekvatne performanse, a relativno je jednostavan za implementaciju.

Primjer 9: Zamjena stranica u tablici stranica prema *Least Recently Used page replacement* algoritmu

Zadan je referentni string koji treba pohraniti u memoriju koristeći LRU algoritam: a, b, c, d, c, a, d, b, e, b, a, b, c, d. Veličina okvira (broj stranica) je 4. Napišite postupak upisivanja referentno stringa u memoriju i označite svaki "Page fault".

Rješenje:

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
a	b	c	d	c	a	d	b	e	b	a	b	c	d
a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b	b	b
		c	c	c	c	c	c	e	e	e	e	e	d
			d	d	d	d	d	d	a	d	d	c	c
X	X	X	X					X				X	X

Ukupno je nastalo 7 *Page fault* događaja.

3. ZADACI ZA SAMOSTALNI RAD

Pristupiti poslužitelju linux.etfos.hr pomoću Putty SSH klijenta. Sljedeće zadatke riješite koristeći uređivač teksta nano, sintaksu programskog jezika C i Bash ljusku operacijskog sustava Linux. Za kompajliranje programa napisanih u C jeziku koristite GCC kompajler koji je već instaliran na poslužitelju. Za svaki zadatak kopirajte programski kod, naredbe ljuske i dobiveni izlaz u Zadaću. Ako je potrebno, napišite dodatno pojašnjenje Vašeg rješenja.

1. Napišite program koji simulira *First-fit* algoritam za dodjeljivanje memorije.
2. Napišite program koji simulira *Worst-fit* algoritam za dodjeljivanje memorije.
3. Zadan je referentni string koji treba pohraniti u memoriju koristeći FIFO algoritam: a, c, d, b, b, a, e, d, a, b, c, e, a. Veličina okvira (broj stranica) je 4. Napišite postupak upisivanja referentnog stringa u memoriju i označite svaki "Page fault". Zadatak riješite ručno, a korake rješenja skicirajte i opišite u zadaći na Loomenu.
4. Zadan je referentni string koji treba pohraniti u memoriju koristeći LRU algoritam: a, c, d, b, b, a, e, d, a, b, c, e, a. Veličina okvira (broj stranica) je 4. Napišite postupak upisivanja referentnog stringa u memoriju i označite svaki "Page fault". Zadatak riješite ručno, a korake rješenja skicirajte i opišite u zadaći na Loomenu.

3.1. Zadaci za vježbu

Ove zadatke nije potrebno riješiti tijekom laboratorijske vježbe. Oni služe isključivo za vježbanje za pismeni ispit, a svoja rješenja možete provjeriti na konzultacijama.

1. Napišite program koji simulira algoritam zamjene stranica *FIFO*.
2. Napišite program koji simulira algoritam zamjene stranica *LRU.*
3. Za primjere 7-9 pretpostavite da sabirnički ciklus (duljina izvođenje jedne instrukcije, tj. stranice) traje T_B = 10 ns, te da dohvat stranice s diska traje T_D = 10 ms. Izračunajte koliko će se program usporiti (u postocima) u svakom od primjera 7-9, zbog *Page Fault* događaja. Prema dobivenim rezultatima usporedite performanse OPR, FIFO i LRU algoritama.
4. Pretpostavite da operacijski sustav održava tablicu stranica čiji je sadržaj jednak kao u primjeru 4. Na temelju takve tablice stranica, prevedite sljedeće logičke adrese u fizičke adrese: 4098, BABA, DEDA (adrese su iskazane u heksadekadskom brojevnom sustavu).

4. LITERATURA

[1] Stallings, W., 2012. Operating systems: internals and design principles. Boston: Prentice Hall,.

[2] Tanenbaum, A.S. and Bos, H., 2015. Modern operating systems. Pearson.

[3] Bovet, D.P. and Cesati, M., 2005. Understanding the Linux Kernel: from I/O ports to process management. " O'Reilly Media, Inc.".

[4] Love, R., 2010. Linux kernel development. Pearson Education.

[5] Love, R., 2013. Linux system programming: talking directly to the kernel and C library. " O'Reilly Media, Inc.".

[6] Jelenković L., 2019. Operacijski sustavi: Interni material za predavanja iz predmeta. Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva (http://www.zemris.fer.hr/~leonardo/os/fer/_OS-skripta.pdf)