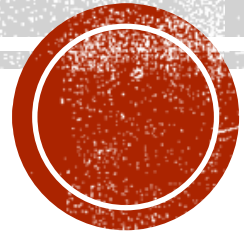
UPRAVLJANJE MEMORIJOM U C-U

Arhitektura računala 2



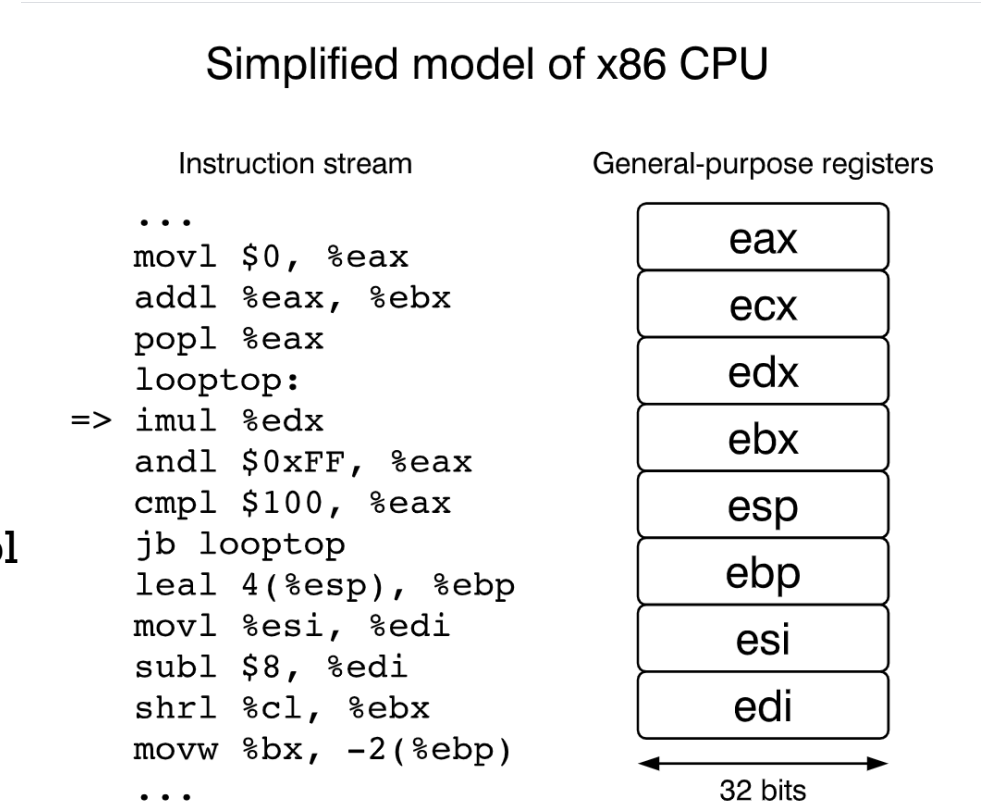
Andi Škrgat

SADRŽAJ

- Aplikacijsko binarno sučelje Windows x64 ABI
- Pohranjivanje podataka na stogu i gomili
- Rukovanje gomilom
- Mapiranje datoteka u memoriju računala
- Korištenje binarnog zapisivanja

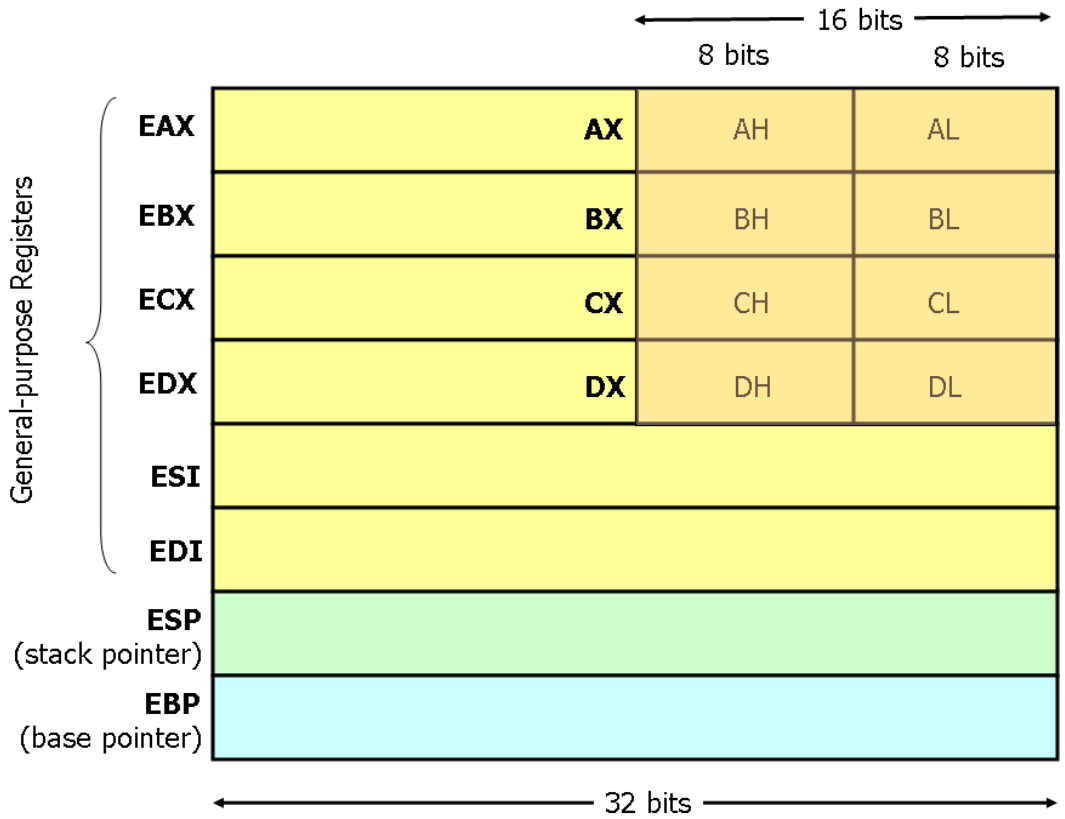
X86 ARHITEKTURA

- Kompatibilna ekstenzija x86 arhitekture – podržava i 32-bitan način rada i 64
- AMD64 i Intel64
  - Instrukcijski set im je jako sličan
- Kompatibilna unazad
- Puno načina adresiranja
  - reg + reg
  - reg
  - disp16 (a 16bit displacement)
  - reg + reg + disp8/16 (an 8 or 16bit displ acement)
  - reg + disp8/16



X86 ARHITEKTURA

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b



X86 ABI

- Konvencije pozivanja
  - Caller clean-up
    - cdecl
      - Parametri na stog
      - EAX, ECX, EDX očuvani od strane pozivatelja
      - Integeri i adrese se vraćaju u EAX registru
      - Realni brojevi u ST0 x87 registru
    - syscall
      - EAX, ECX, EDX nisu očuvani
    - optlink
      - Parametri preko registara EAX, EDX, ECX, ST0-ST3
      - Povratna vrijednost EAX i ST0

X86 ABI

- Callee clean-up
  - pascal
    - slijeva na desno argumenti na stog
    - Povratne vrijednosti u AL(8-bit), AX(16-bit), EAX(32-bit), DX:AX(32-bit on 16-bit systems)
  - stdcall
    - S desna nalijevo argumenti na stog
    - Win32 API
  - Microsoft fastcall
    - Prva 2 argumenta u EDX i ECX ostali na stog
    - S desna nalijevo
  - Microsoft vectorcall, Borland registers, Watcom register, TopSpeed, Clarion, JPI, safecall

MICROSOFT X64 KONVENCIJA POZIVANJA

- Prva 4 cjelobrojna parametra(i pointeri) –> RCX, RDX, R8 i R9 registri
- Prva 4 realna parametra -> XMM0 – XMM3 – SSE registri
- Ostatak na stog
- Pointer ili cjelobrojna povratna vrijednost -> RAX registar
- Realna povratna vrijednost -> XMM0
- RAX, RCX, RDX, R8-R11 promjenjivi
- RBX, RBP, RDI, RSI, R12-R15 nepromjenjivi
- RSP je pokazivač na vrh stoga

IZVORNI KOD PREVOĐENJE

```
#include<iostream>

int proc(int a, int b){
        int c[2];
        c[0]=a-b;
        c[1]=2*c[0];
        return c[0]+c[1];
}

int main(){
        std::cout << proc(3,1) << std::endl;
}
```

NEOPTIMIZIRANO PREVOĐENJE

```
        .file   "arh.c"
        .intel_syntax noprefix
        .text
        .globl  proc
        .def    proc;   .scl    2;      .type   32;     .endef
        .seh_proc       proc
proc:
        push    rbp //rbp na stog
        .seh_pushreg    rbp
        mov     rbp, rsp //premjesti pokazivač na vrh stoga u rbp
        .seh_setframe   rbp, 0
        and     rsp, -16
        sub     rsp, 16
        .seh_stackalloc 16
        .seh_endprologue
        mov     DWORD PTR 16[rbp], ecx
        mov     DWORD PTR 24[rbp], edx
        mov     eax, DWORD PTR 16[rbp]
        sub     eax, DWORD PTR 24[rbp] //c[0] = a - b
        mov     DWORD PTR -8[rbp], eax
        mov     eax, DWORD PTR -8[rbp]
        add     eax, eax //c[1] = 2*c[0]
        mov     DWORD PTR -4[rbp], eax
        mov     edx, DWORD PTR -8[rbp]
        mov     eax, DWORD PTR -4[rbp]
        add     eax, edx  // c[0] + c[1]
        mov     rsp, rbp
        pop     rbp
        ret
        .seh_endproc
        .def    __main; .scl    2;      .type   32;     .endef
        .globl  main
        .def    main;   .scl    2;      .type   32;     .endef
        .seh_proc       main
main:
        push    rbp
        .seh_pushreg    rbp
        mov     rbp, rsp
        .seh_setframe   rbp, 0
        and     rsp, -16
        sub     rsp, 32
        .seh_stackalloc 32
        .seh_endprologue
        call    __main
        mov     edx, 1
        mov     ecx, 3
```

OPTIMIZIRANO PREVOĐENJE

```
        .file   "main.c"
        .intel_syntax noprefix
        .text
        .def    __main; .scl    2;      .type   32;     .endef
        .section .rdata,"dr"
.LC0:
        .ascii "%d\12\0"
        .section        .text.startup,"x"
        .p2align 4,,15
        .globl  main
        .def    main;   .scl    2;      .type   32;     .endef
        .seh_proc       main
main:
        sub     rsp, 40
        .seh_stackalloc 40
        .seh_endprologue
        call    __main
        mov     edx, 1
        mov     ecx, 3
        call    proc
        mov     edx, eax
        lea     rcx, .LC0[rip]
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret
        .seh_endproc
        .ident  "GCC: (x86_64-win32-seh-rev0, Built by MinGW-W64 project) 8.1.0"
        .def    proc;   .scl    2;      .type   32;     .endef
        .def    printf; .scl    2;      .type   32;     .endef
```

```
        .file   "arh22.c"
        .intel_syntax noprefix
        .text
        .p2align 4,,15
        .globl  proc
        .def    proc;   .scl    2;      .type   32;     .endef
        .seh_proc       proc
roc:
        .seh_endprologue
        sub     ecx, edx
        lea     eax, [rcx+rcx*2]
        ret
        .seh_endproc
        .ident  "GCC: (x86_64-win32-seh-rev0, Built by MinGW-W64 project) 8.1.0"
```

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

ALOKACIJA NA STOGU

```cpp
#include<iostream>

int main(void) {

    int x = 1;
    int y = 2;
    int z = 3;

    std::cout << "Adresa x: " << &x << std::endl;
    std::cout << "Adresa y: " << &y << std::endl;
    std::cout << "Adresa z: " << &z << std::endl;

}
```

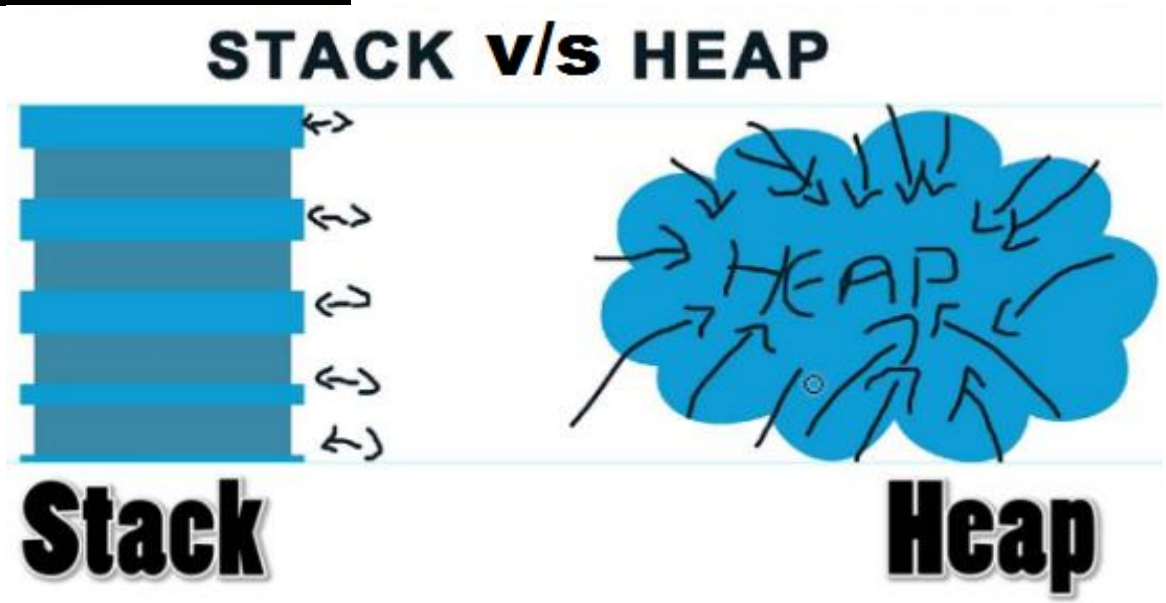
```
C:\Users\Korisnik\Desktop\C++Arh\Stack>StackAdresses
Adresa x: 0x61fe1c
Adresa y: 0x61fe18
Adresa z: 0x61fe14

C:\Users\Korisnik\Desktop\C++Arh\Stack>
```

JOŠ O STOGU...

- I ako imamo više tipova objekata, rezultat će biti isti(ako debugiramo u VSC onda će se dodati bajtovi između objekata kao „safety guards", zbog overflowa i slično, ali ovdje su kontinurano)
- Ispis na sljedećem slajdu

```cpp
#include <iostream>

class VectorInt {
    public:
        int x, y, z;

        VectorInt() : x(1), y(2), z(3) {}

};


class VectorFloat  {
    public:
        float x, y, z;

        VectorFloat() : x(4), y(4), z(4) {}

};



int main(void) {

    VectorInt intArr[2];

    VectorFloat floatArr[2];

    std::cout << "Adresa int1: " << &intArr[0] << std::endl;
    std::cout << "Adresa int2: " << &intArr[1] << std::endl;
    std::cout << "Adresa float1: " << &floatArr[0] << std::endl;
    std::cout << "Adresa float2: " << &floatArr[1] << std::endl;

}
```

```
C:\Users\Korisnik\Desktop\C++Arh\Stack>g++ StackDifferentObjects.cpp

C:\Users\Korisnik\Desktop\C++Arh\Stack>a
Adresa int1: 0x61fdf0
Adresa int2: 0x61fdfc
Adresa float1: 0x61fdd0
Adresa float2: 0x61fddc

C:\Users\Korisnik\Desktop\C++Arh\Stack>
```

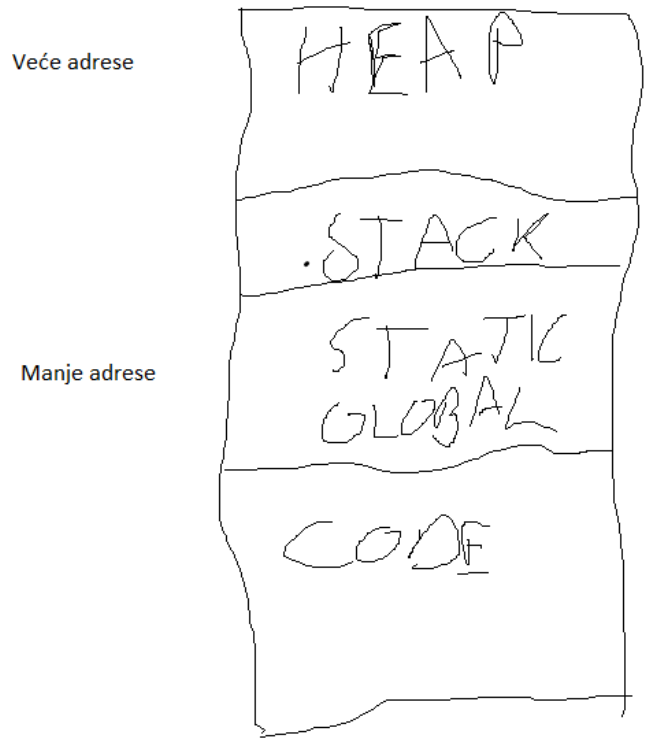


ALOKACIJA NA HEAPU

```cpp
#include <iostream>


int main(void) {

    int* x = new int;
    int* y = new int;
    int* z = new int;

    std::cout << "Adresa x: " << x << std::endl;
    std::cout << "Adresa y: " << y << std::endl;
    std::cout << "Adresa z: " << z << std::endl;


    delete x;
    delete y;
    delete z;


}
```

```
C:\Users\Korisnik\Desktop\C++Arh\Stack>g++ HeapAdresses.cpp

C:\Users\Korisnik\Desktop\C++Arh\Stack>a
Adresa x: 0x7024f0
Adresa y: 0x702510
Adresa z: 0x702530

C:\Users\Korisnik\Desktop\C++Arh\Stack>
```

GOMILA VS STOG



Windows OS, na Linuxu su adrese stoga veće od adresa koje pripadaju gomili

- Stog koristiti kad god je to moguće
- Rad sa stogom znači doslovno pomicanje pokazivača na vrh stoga -> brzina -> praktički 1CPU instrukcija
- Objekti su na stogu samo u onom bloku u kojem su inicijalizirani -> zato im ne možemo pristupiti kada izađemo iz bloka
- Gomilu koristiti samo u situacijama kada nam treba puno memorije <- veličina stoga ~ 1-2MB
- Gomila se također koristi u situacijama kada nam je potreban duži životni vijek objekta
- Gomila može rasti kako se naš program mijenja
- Lokacija u RAM-u !!!
- Gomila se naziva dinamičkom memorijom zato što veličinu objekta zadajemo tek tijekom izvođenja

DINAMIČKA ALOKACIJA NA STOGU

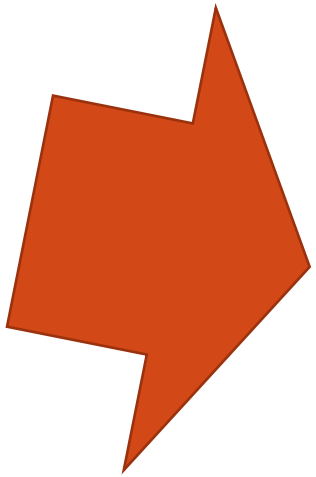
- Ideja: alocirati memoriju na gomili je skupa operacija
  - Zašto ne na stacku :)
- Memorija se oslobađa nakon izlaska iz pozivajuće funkcije
  - Nema poziva funkcija alokatora npr. delete, free ili new
- Na Windowsima najviše 1MB, dok je na Linuxu moguće rezervirati do 8MB
  - To ovisi o funkcijama koje su se pozivale prije, moguće je da imamo I manje naravno
- Koriste se funkcije _alloca I _malloca
  - Bolje je koristiti _malloca jer će, ako nema dovoljno mjesta na stogu, memorija biti rezervirana na gomili

```
1  void allocStackDynamic(size_t size)
2  {
3      assert(size > 0);
4      char *mem = static_cast<char *>(_malloca(size));
5      memset(mem, 0, size);
6      _freea(mem);
7  }
```

KLJUČNA RIJEČ NEW

New → malloc → OS

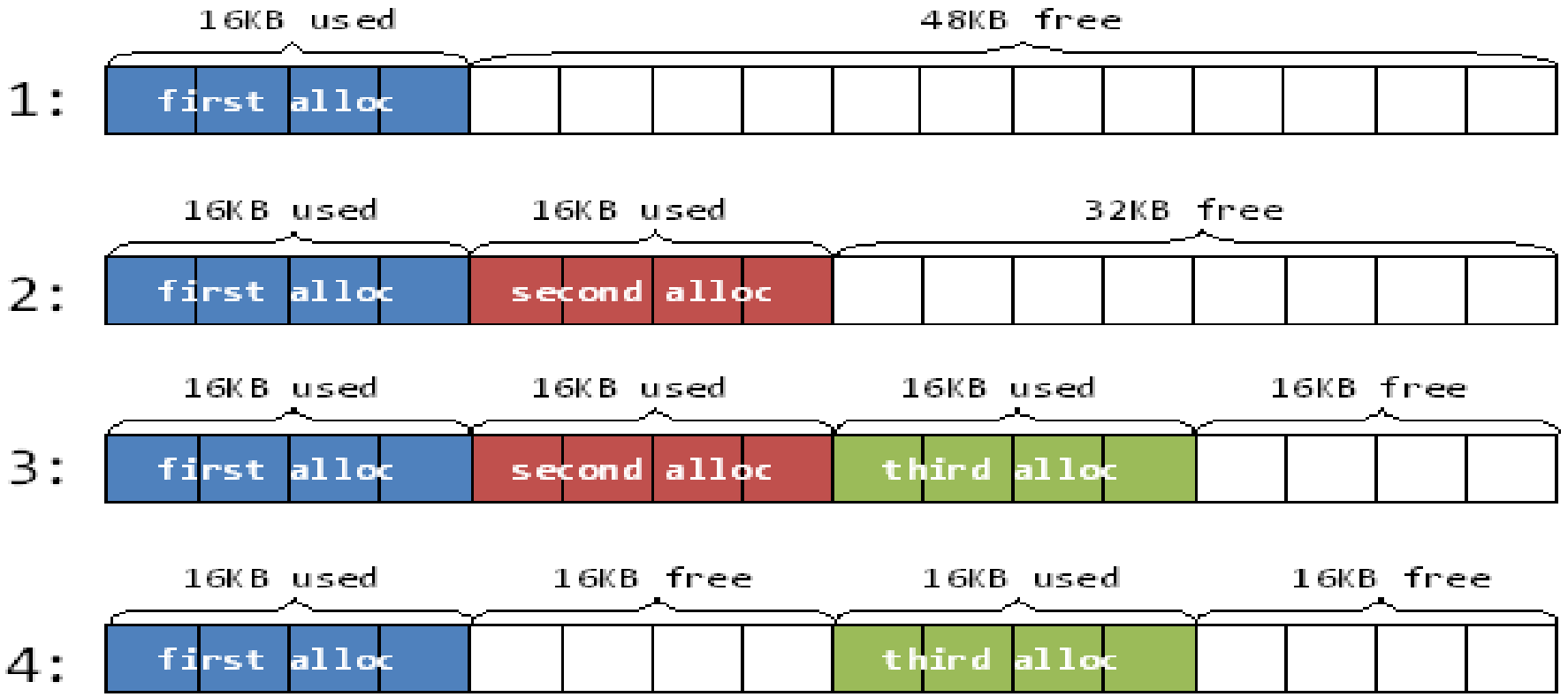
- Interno se održava free lista -> koji su sve komadi memorije slobodni, gdje su, koliko zauzimaju i sl. -> kako?



OS nam dodijeli određeni dio memorije prilikom započinjanja programa

New najčešće u pozadini koristi malloc, ali to ovisi o konkretnoj implementaciji STL-a(ne kompajler ili OS)

Operacija koja je pogotovo skupa: treba mi još memorije!!!


16KB used
48KB free
1:
first alloc
16KB used
16KB used
32KB free
2:
first alloc
second alloc
16KB used
16KB used
16KB used
16KB free
3:
first alloc
second alloc
third alloc
16KB used
16KB free
16KB used
16KB free
4:
first alloc
third alloc

MALLOC

- Zahtjevi dobrog memorijskog alokatora(malloc, realloc, free):
  - **Kompatibilnost** s drugima, poštuje ANSI/POSIX konvencije
  - **Portabilnost** -> što je manje moguće sistemski-ovisnih poziva
  - Alokator bi trebao zauzimati memoriju na način da **minimizira fragmentaciju !!!**
  - **Vrijeme izvođenja**
  - Neke **opcije izvođenja** se mogu mijenjati - statički pomoću #define, dinamički pomoću kontrolnih naredbi kao što je npr. mallopt
  - Maksimiziranje prepoznavanja greške
  - **Neovisnost** o vrsti programa koji se izvodi(GUI, kompajleri, web programi, procesiranje stringova i sl.)

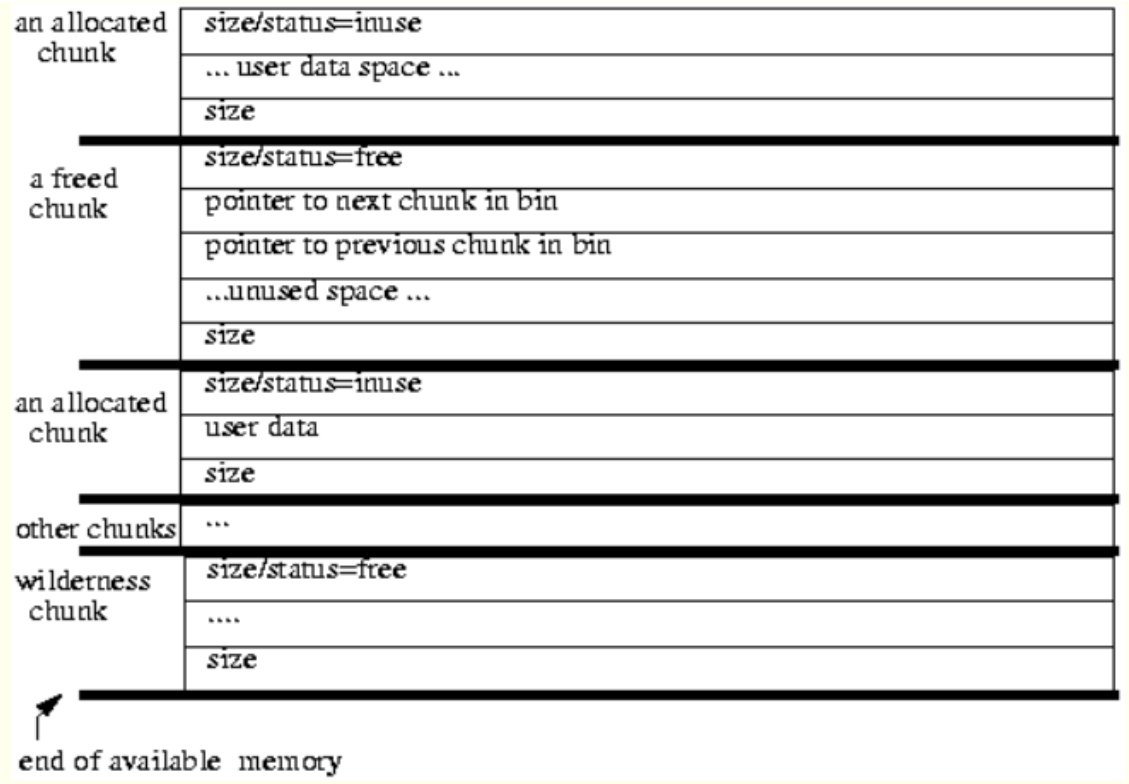
U teoriji vremenski najbrža inačica je ona koja alocira uvijek susjedni dio memorije, a implementacija za free je no-op. Na taj način imamo jako malo promašaja kod straničenja i keširanja, ali brzo dolazimo do situacije u kojoj više nemamo dostupne memorije.

ALGORITMI MALLOCA

1. Granične oznake

- Svaki komadić memorije(chunk) sadrži u sebi informacije o **veličini prijašnjeg i sljedećeg chunka** -> time je olakšano spajanje više njih u jedan veći chunk
- Omogućena je pretraga **naprijed i nazad**

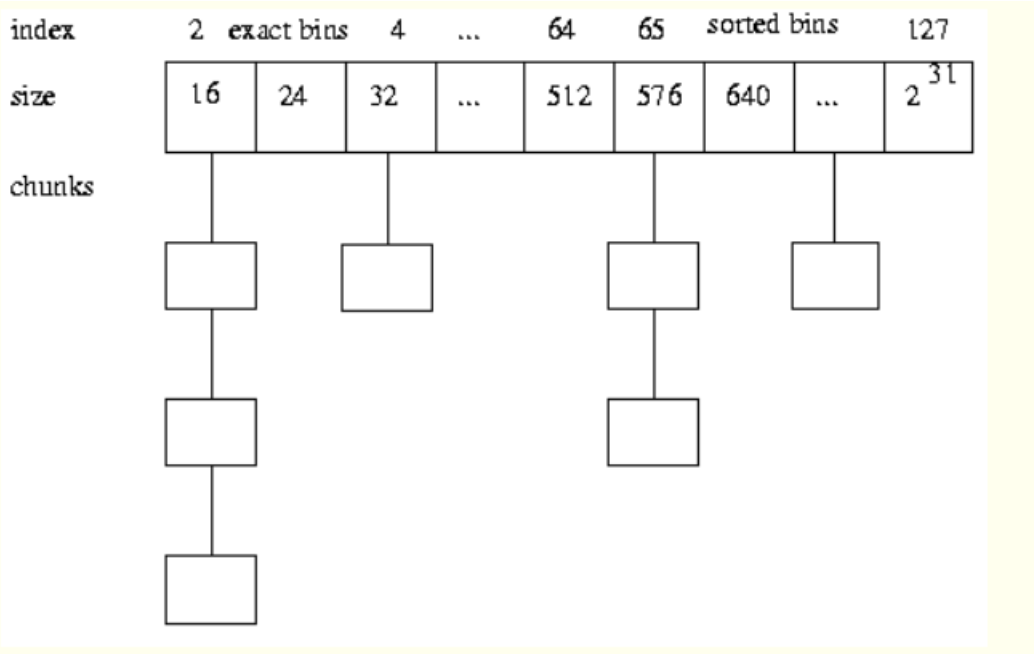
- Prvotne verzije su doslovno to koristile
- Poboljšanje: chunkovi koji su aktivni ne nose te informacije jer ih i ovako neće moći koristiti = otežana detekcija grešaka, ali povećana efikasnost



ALGORITMI MALLOCA

2. Binning

- Dostupni chunkovi su organizirani u kontejnere(grupirani po veličini)
- Najmanji mogući chunk koji nam je dovoljan se traži -> best-fit search -> najmanja fragmentacija



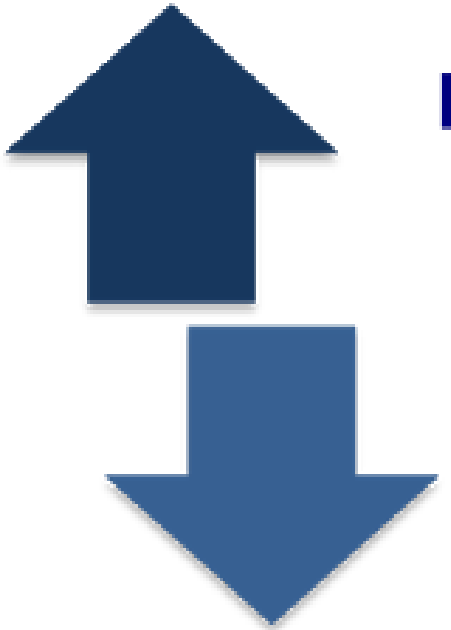
Optimizacija na način da se chunkovi unutar kontejnera sortiraju.

Zaključak:

Najbolji prvi sa spajanjem, oslobođeni se spajaju s a susjedima i drže u pripadnom sortiranom kontejneru

HEURISTIČKA POBOLJŠANJA ALOKATORA

1. Očuvanje lokalnosti
2. „Wilderness preservation”
3. Mapiranje memorije
4. Keširanje
5. „Lookasides”



Heuristic techniques
are ideal as a quick fix

but

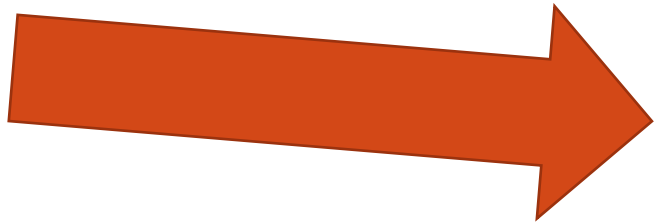
can introduce error, bias &

may constrain creativity

Heuristika omogućava efikasnije korištenje alokatora

OČUVANJE LOKALNOSTI

- Temelji se na pretpostavci da chunkovi memorije koji su alocirani u približno slično vrijeme imaju slična svojstva(slične reference, doseg…)
- Takva heuristika pokušava gotovo uvijek koristiti nearest-fit strategiju, upravo zbog pretpostavke lokalnosti



Fragmentacija

- Koristi se u situacijama kada stvara najmanje konflikata sa ostalim zahjevima alokatora

WILDERNESS PRESERVATION

- Predstavlja prostor koji graniči s najvišom adresom rezerviranom od sustava
- Jedini chunk memorije koji se može proizvoljno proširiti, naravno pod pretpostavkom da ima dostupne memorije

Koristimo na 2 načina

Tretiramo kao i ostale chunkove
Brzina, jednostavnost, ali moguće loše memorijske karakteristike

Tretiramo chunk kao veći, nego što je
jer zapravo on takav može postati -> koristi
se samo onda kada drugi chunkovi ne postoje

LOOKASIDES

- U starijim verzijama minimalna veličina chunka je bila 16B pa je u situacijama u kojim je npr. neka struktura podataka koristila samo 8B(za jedan čvor) došlo do nepotrebnog gubljenja memorije
- New alokator bi mogao provjeravati adrese da li se koriste, kojem prostoru pripadaju i sl., ali to bi bilo neefikasno
- U tim situacijama programer bi mogao napisati svoj prilagođeni alokator
  - Novije verzije new alokatora te stvari imaju rješene, a implementiranje svojeg alokatora je u većini situacija nepotrebno

CACHING

- Najosnovnija verzija algoritma spaja susjedne chunkove, a odvaja ih samo kad je to nužno(skupe operacije!!!)

- 2 caching strategije:

1. Nemoj spajati susjede, nego se nadaj da će ubrzo doći sljedeći zahtjev u kojem ćeš iskoristiti chunk te veličine(odgođeno spajanje)
2. Odvoji chunkove unaprijed i to više njih, a ne jedan po jedan(prealociranje)

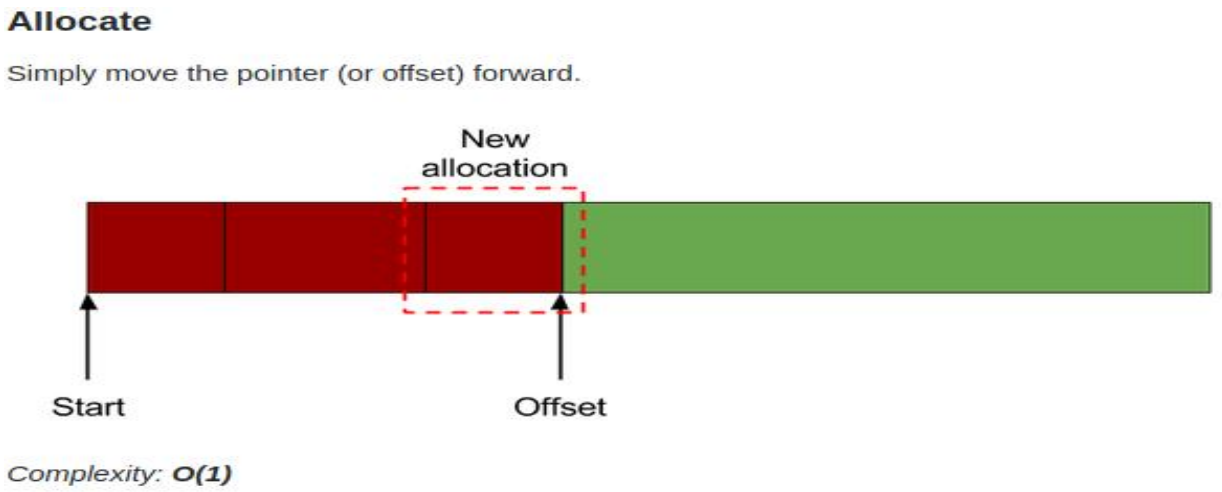
Koristi se u situacijama kada se često stvaraju i otpuštaju chunkovi iste veličine, npr. struktura podataka. Koristi se samo za keširanje chunkova malih veličina. Moderni programi koriste klase različitih veličine pa teško profitiramo od ovoga.

PRILAGOĐENI ALOKATORI U C++

- Alokator je programski alat iz STL-a kojeg koristimo za upravljanje memorijom
- Alexander Stepanov želio proširiti STL, ciljevi:
  - Apstrahiranje pristupa različitim memorijama(shared memory, garbage collected memory) – danas strože, nego što je on to izvorno zamislio
  - Povećanje performanse programa
- Alokator new općenit alokator što znači da mora prekrivati sve slučajeve
  - Bolje radi za rijeđi pristup većim objektima kao što su vektori
  - Dobar primjer gdje se može postići povećanje performansi pisanjem prilagođenog alokatora je u programu koji često stvara i otpušta objekte male veličine
- Gotovo svim custom alokatorima je zajedničko da unaprijed rezerviraju određeni blok memorije
  - Konkretna implementacija onda određuje I mijenja pristup

LINEARAN ALOKATOR

- Iznimno jednostavan, jedino što nam treba je pointer koji se pomiče kako rezerviramo memoriju
  - Free operacija skupa jer se odjednom oslobađa cijeli blok memorije, ne razlikujemo pojedini blok memorije od drugog

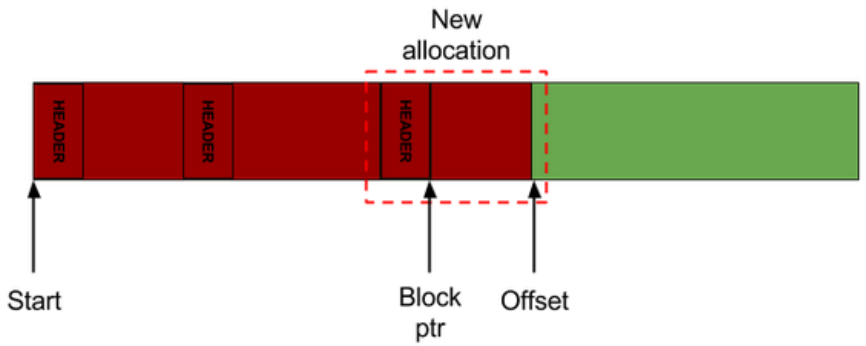


STOG ALOKATOR

- Unaprijeđenje linearnog alokatora
- Pri alociranju pointer se pomiče prema naprijed, ali za razliku od linearnog alokatora sada ne moramo dealocirati cijeli blok memorije odjedanput, nego pomičemo pointer unazad

Allocate

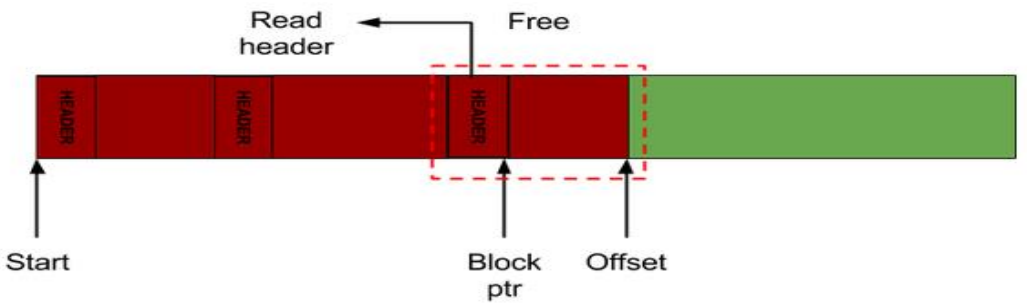
Simply move the pointer (or offset) forward and place a header right before the memory block indicating its size.



Complexity: ***O(1)***

Free

Simply read the block size from the header and move the pointer backwards given that size.

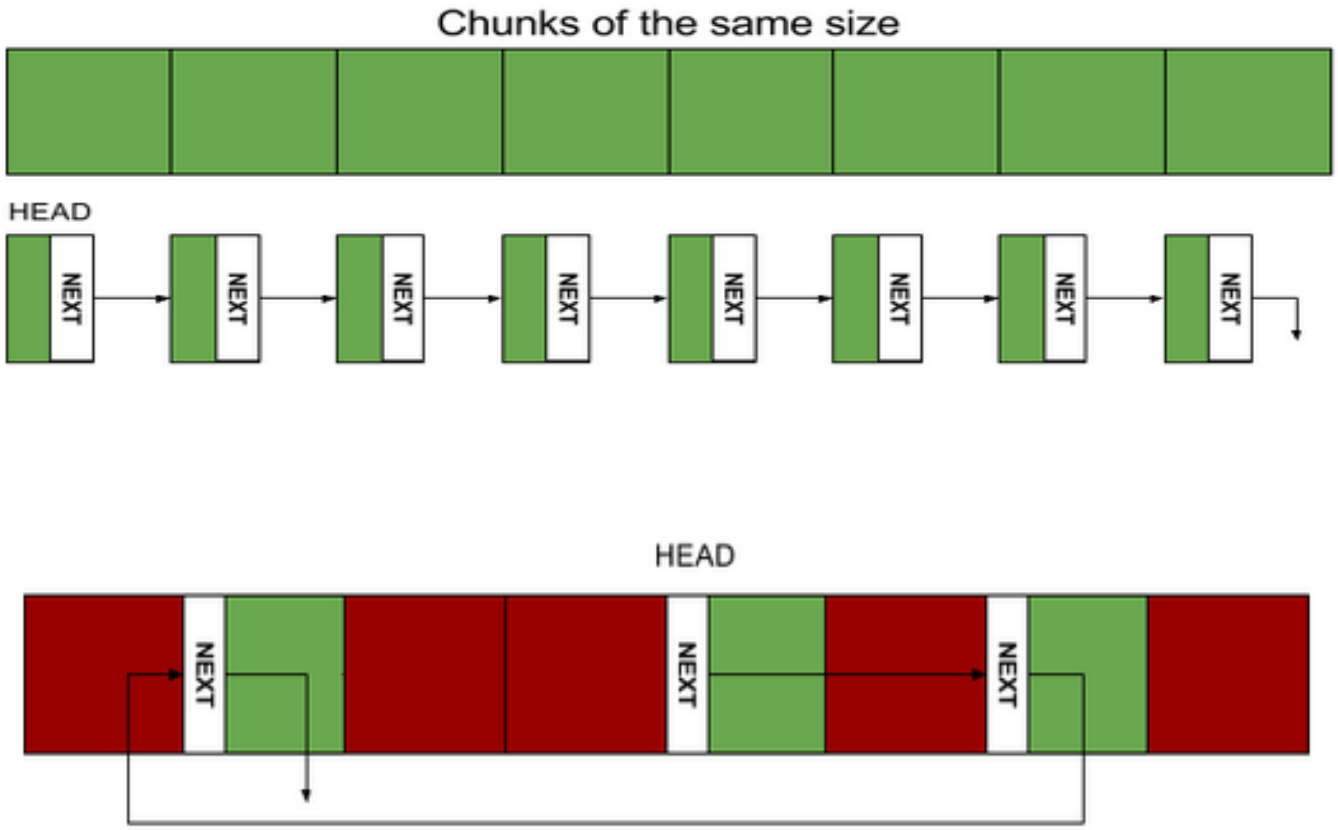


Complexity: ***O(1)***

POOL ALOKATOR

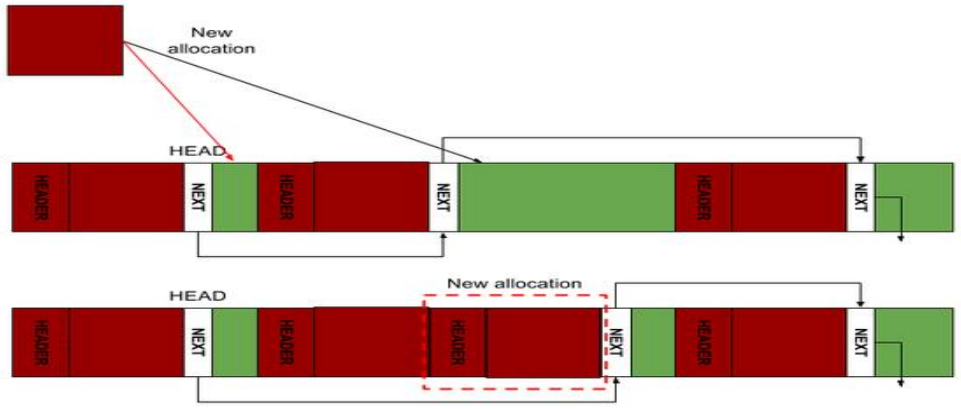
- Rezervira unaprijed veći blok memorije te ga podijeli u manje dijelove iste veličine
- U svakom chunku je zapisan pokazivač na sljedeći slobodni blok memorije
  - Pazi: Ograničenje je onda da svaki chunk mora biti minimalno veličine čvora povezane liste u kojoj se pamte adrese na sljedeći slobodni blok memorije
- Alociranje: Uzmi prvi blok iz povezane liste
- Dealociranje: Oslobođeni blok stavi na prvo mjesto u povezanu listu
- Alociranje I dealociranje imaju konstantnu vremensku složenost, ali korištenje pool alokatora je u konačnosti linearno u vremenu zbog inicijalizacije jednostruko povezane liste
- Pogodno kod korištenja objekata koji su uvijek iste veličine

POOL ALOKATOR

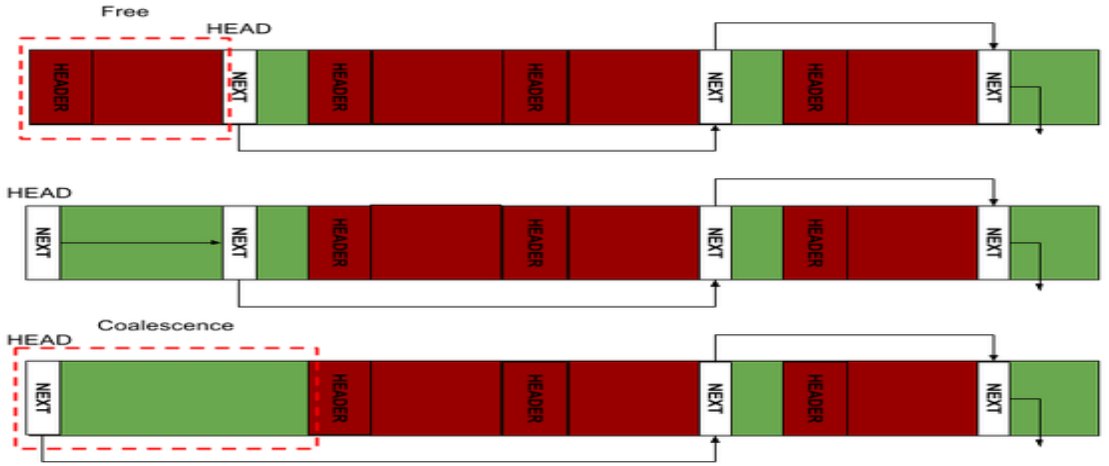


ALOKATOR – LINEARNA LISTA

- Svi prethodni alokatori stavljaju ograničenja na pristup podatcima -
  - Slijedno – sa stalnim oslobađanjem ili bez – linearan I stogovni alokator
  - Svi objekti su iste veličine - pool alokator
- Alokator koji u pozadini koji koristi linearnu listu daje puno više slobode pri korištenju memorije
  - Pri tome imamo smanjenje performanse zbog toga jer obje operacije(alociranje I dealociranje) imaju linearnu vremensku složenost



Complexity: ***O(N)*** where N is the number of free blocks

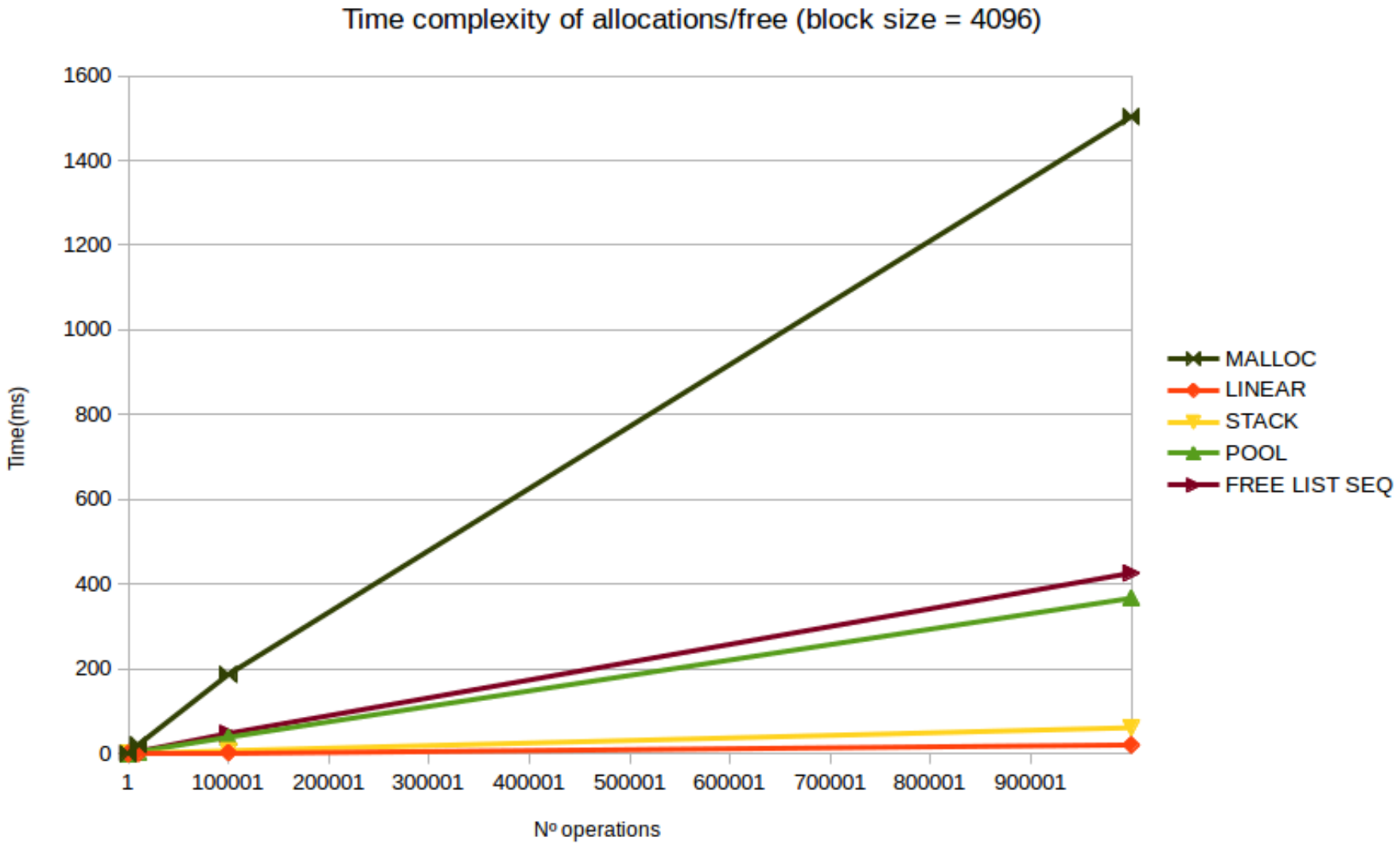


Complexity: ***O(N)*** where N is the number of free blocks

USPOREDBE ALOKATORA

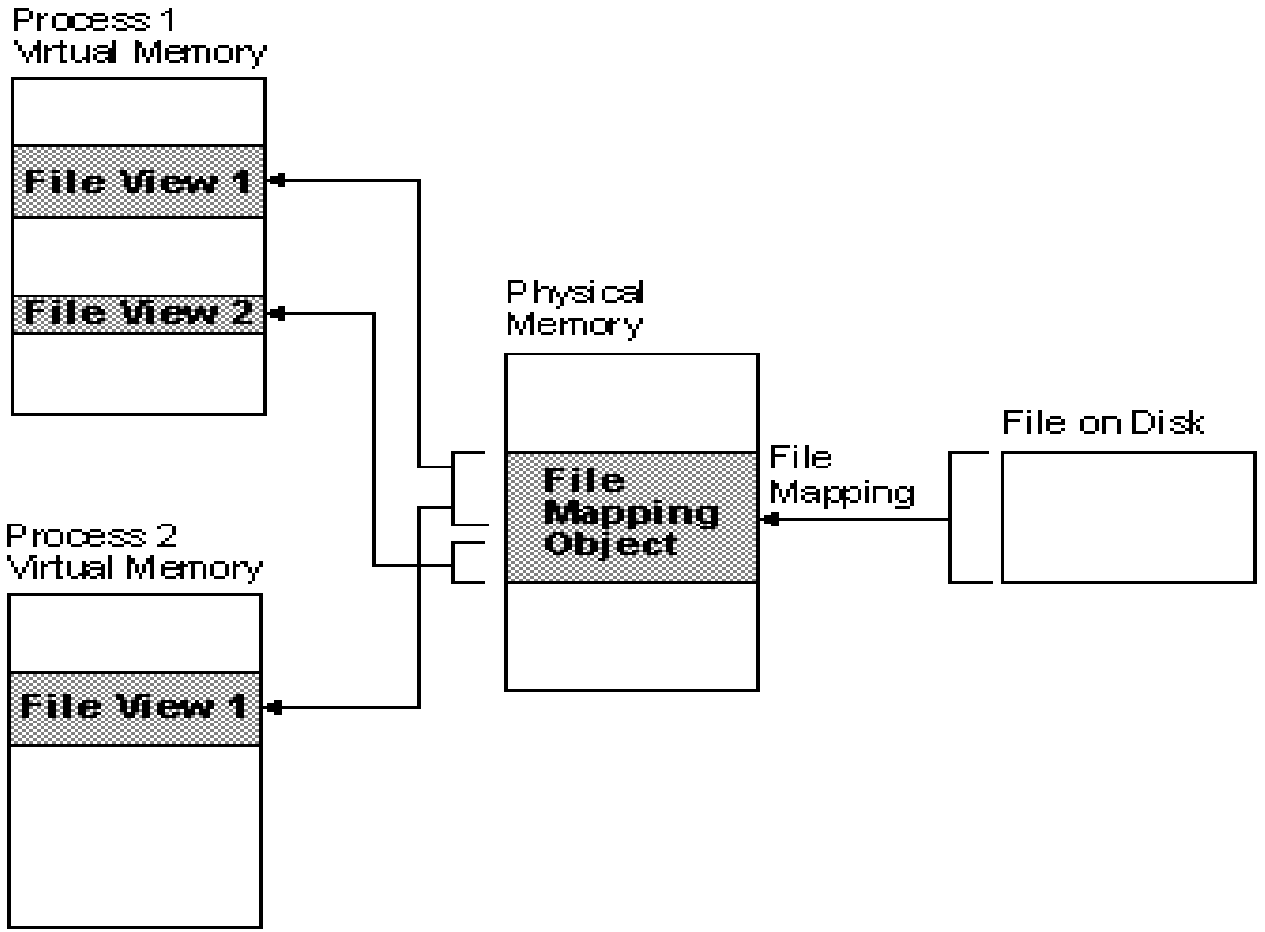
- Malloc je najsporiji zbog svoje općenitosti
- Pool alokator, linearan I stogovni alokator su puno brži međutim zahtijevaju korištenje memorije u posebnom obliku
  - Linearni - ne dopušta oslobađanje malih dijelova memorije
  - Stogovni – sve operacije imaju konstantnu vremensku složenost, ali se memoriji mora pristupat na način da se koristi LIFO(Last-In-First-Out)
  - Pool alokator – zahtijeva da su svi dijelovi memorije koje koristimo iste veličine

- Alokator koji koristi u pozadini listu je najbolji izbor
  - Njega se može ubrzati koristeći stablo kao strukturu podataka pri čemu bismo imali sortirane dijelove memorije logaritamske vremenske složenosti


Time complexity of allocations/free (block size = 4096)
Time(ms)
1600
1400
1200
1000
800
600
400
200
0
1
100001
200001
300001
400001
500001
600001
700001
800001
900001
Nº operations
MALLOC
LINEAR
STACK
POOL
FREE LIST SEQ

MEMORY MAPPING

- Virtualna memorija se brine za alociranje odvojenih prostore memorije programu
  - Najčešće mapiramo datoteku veće veličine
- Smanjuje fragmentaciju – ne stvaraju se rupe u memoriji
- Nakon mapiranja datoteci se pristupa kao memoriji što znači da se mogu koristiti mnogobrojni postupci za ubrzavanje pristupa sadržaju
  - Npr. compiler auto-vectorization, SIMD intrinsics, prefetching, optimized in-memory parsing routines, OpenMP
- Vrijedno koristiti samo u nekim situacijama:
  1. Više procesa koji dijele istu memoriju, mmap omogućava da svi ti procesi dijele istu memoriju
  2. Optimiziranje operacija straničenja
     - ako npr. A i B koriste istu memoriju, A preko buffera, i B preko mmap, ako se promijeni sadržaj memorije B će moć svejedno koristit tu memoriju jer će znat koje stranice su promijenile

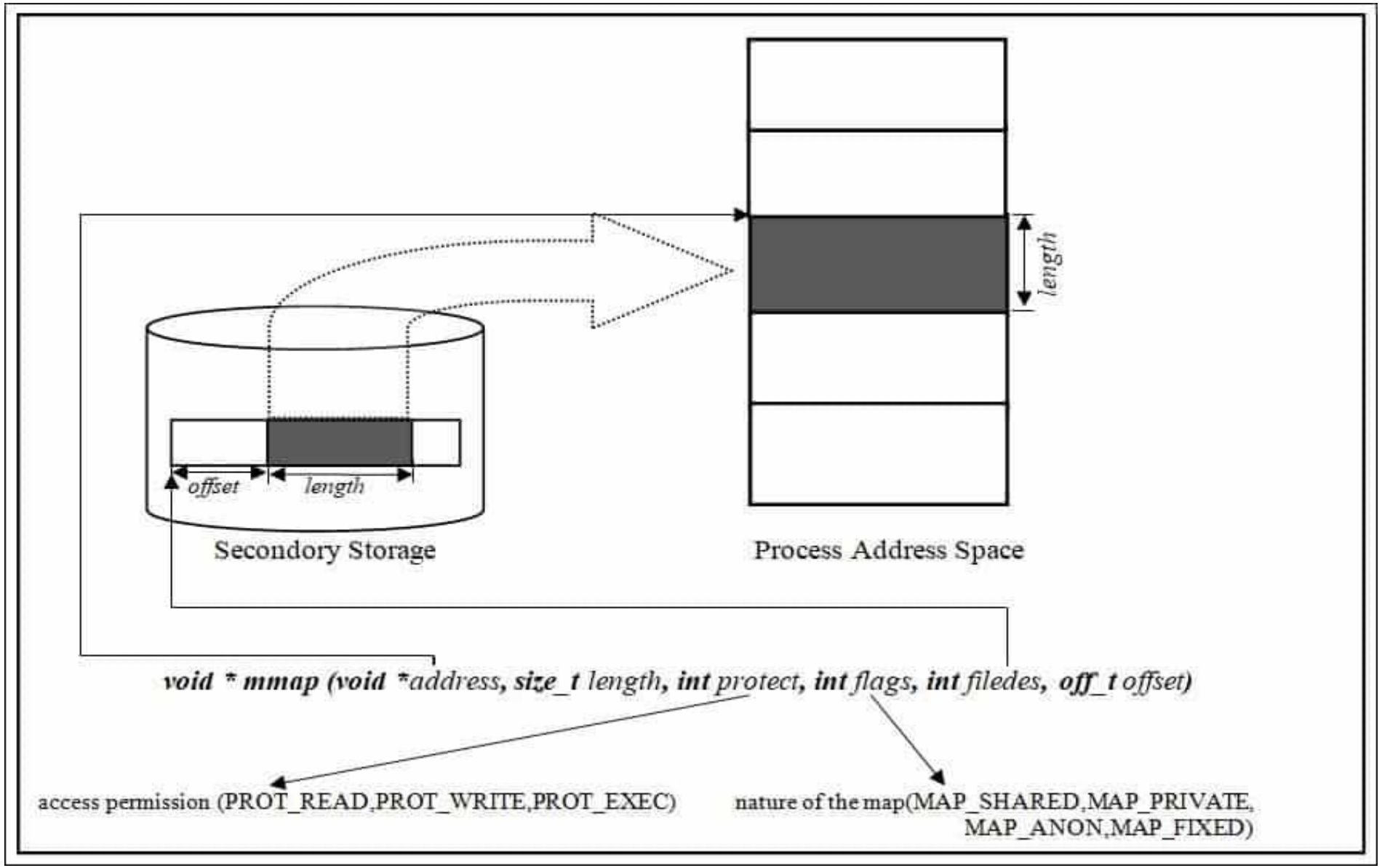

Process 1
Virtual Memory
File View 1
File View 2
Process 2
Virtual Memory
File View 1
Physical
Memory
File
Mapping
Object
File
Mapping
File on Disk

POZITIVNE STRANE MAPIRANJA

- Alternativa običnom čitanju
- Nasumičan pristup datoteci
- Koristimo istu datoteku više vremena(ne koristimo je samo jednom pri čitanju)
- Omogućava korištenje stranica iz keša I nakon zatvaranja I ponovnog otvaranja datoteke, s običnim čitanjem datoteka je već vjerojatno izbačena iz keša, zbog poziva flush()
- Dijeljenje memorije između procesa
  - Procesi koriste iste fizičke stranice
- Potreban je samo 1 sistemski poziv za mapirat cijelu datoteku
- Nakon toga, nema nikakvog više kopiranja iz jezgre prema korisniku

USPOREDBA S OBIČNIM ČITANJEM

- Da bi se npr. datoteka od 50GB mapirala, uz veličinu stranice od 4K, potrebno je 12.5 milijuna promašaja prilikom straničenja -> uz best-case scenario gdje je svaki promašaj ~100ns dolazimo do brojke od 1.25s
- Najveći je nedostatak običnog čitanja to što svaki poziv metode read() zahtijeva KOPIRANJE N bajtova iz jezgre prema korisniku
- Prema tome osnovna stvar koju se ovdje pitamo je:
  - Da li je veći trošak kopiranje bajtova iz jezgre ili posao koji moramo učiniti pri svakom promašaju kod straničenja
  - Odgovor ovisi o puno faktor:
    - Copy_to_user faktor kopiranja, hardverska potpora mapiranju, veličina TLB-a itd.

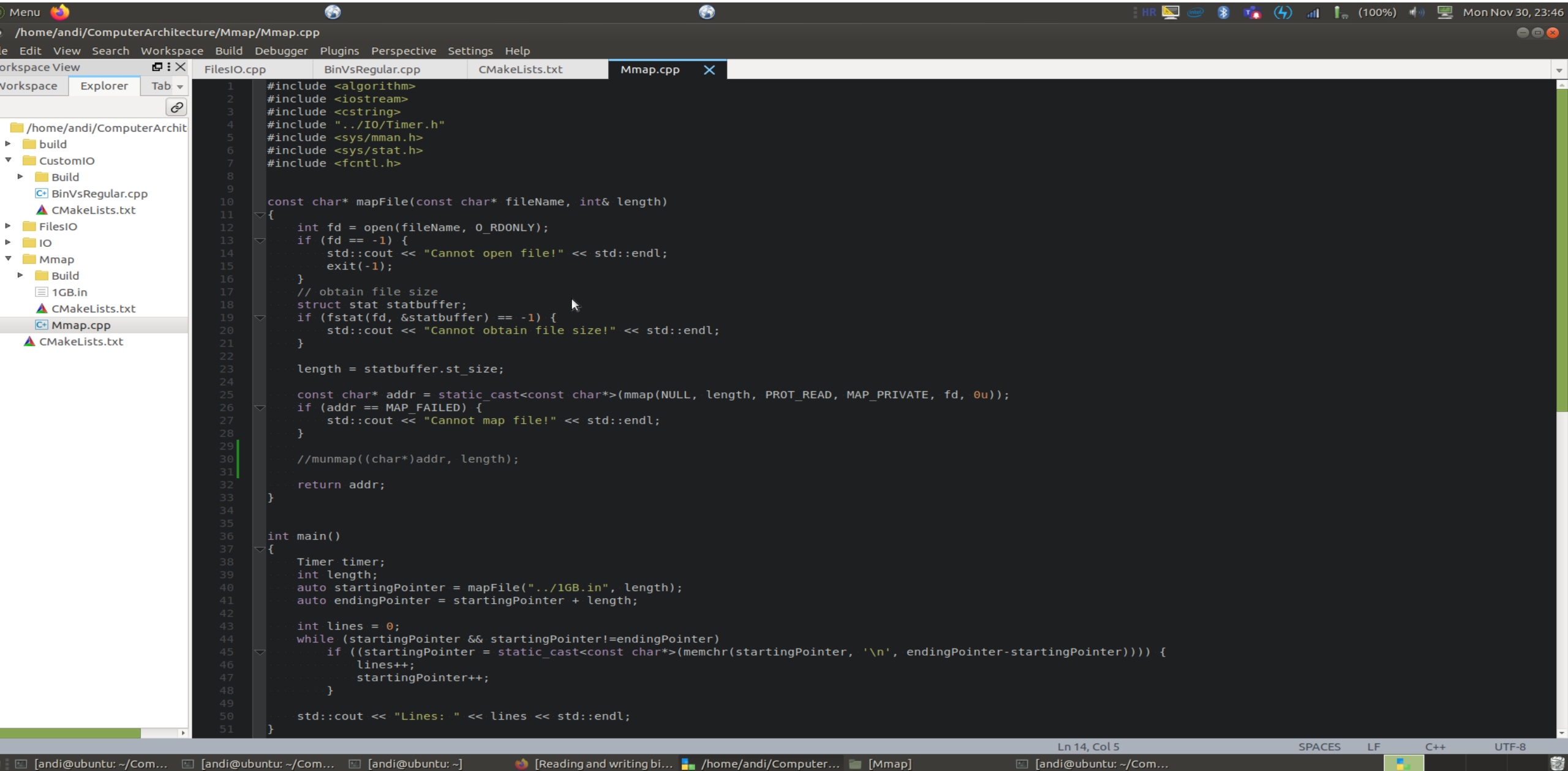

length
offset
length
Secondory Storage
Process Address Space
void * mmap (void *address, size_t length, int protect, int flags, int filedes, off_t offset)
access permission (PROT_READ,PROT_WRITE,PROT_EXEC)
nature of the map(MAP_SHARED,MAP_PRIVATE,
MAP_ANON,MAP_FIXED)

MEMORIJSKO MAPIRANJE U C++, LINUX

- 1. argument je željena adresa, jezgra bira najbližu moguću adresu, ako mu damo null onda jezgra sama nasumično odabere početnu adresu
- 2. broj bajtova koje trebamo rezervirati
- 3. razina prava – pisanje, čitanje, izvršavanje
- 4. argument - zastavice
  - Map shared – dijeljenje memorije s ostalim procesima, promjene se upisuju u datoteku
  - Map private – promjene neće biti upisane u datoteku
  - Map anonymous – anonimno mapiranje, prostor nije mapiran u datoteku, primitivni način za povećanje gomile
  - Map fixed – zahtijeva se eksplicitno korištenje prostora od dane adrese

Više informacija o memorijskom mapiranju na Windowsima
https://docs.microsoft.com/en-us/windows/win32/memory/file-mapping

MMAP IN C++


Menu
Mon Nov 30, 23:46
/home/andi/ComputerArchitecture/Mmap/Mmap.cpp
Edit View Search Workspace Build Debugger Plugins Perspective Settings Help
Workspace View
Workspace Explorer Tab
/home/andi/ComputerArchit
build
CustomIO
Build
BinVsRegular.cpp
CMakeLists.txt
FilesIO
IO
Mmap
Build
1GB.in
CMakeLists.txt
Mmap.cpp
CMakeLists.txt
FilesIO.cpp
BinVsRegular.cpp
CMakeLists.txt
Mmap.cpp
#include <algorithm>
#include <iostream>
#include <cstring>
#include "../IO/Timer.h"
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
const char* mapFile(const char* fileName, int& length)
{
int fd = open(fileName, O_RDONLY);
if (fd == -1) {
std::cout << "Cannot open file!" << std::endl;
exit(-1);
}
// obtain file size
struct stat statbuffer;
if (fstat(fd, &statbuffer) == -1) {
std::cout << "Cannot obtain file size!" << std::endl;
}
length = statbuffer.st_size;
const char* addr = static_cast<const char*>(mmap(NULL, length, PROT_READ, MAP_PRIVATE, fd, 0u));
if (addr == MAP_FAILED) {
std::cout << "Cannot map file!" << std::endl;
}
//munmap((char*)addr, length);
return addr;
}
int main()
{
Timer timer;
int length;
auto startingPointer = mapFile("../1GB.in", length);
auto endingPointer = startingPointer + length;
int lines = 0;
while (startingPointer && startingPointer!=endingPointer)
if ((startingPointer = static_cast<const char*>(memchr(startingPointer, '\n', endingPointer-startingPointer)))) {
lines++;
startingPointer++;
}
std::cout << "Lines: " << lines << std::endl;
}
Ln 14, Col 5
SPACES
LF
C++
UTF-8
[andi@ubuntu: ~/Com...
[andi@ubuntu: ~/Com...
[andi@ubuntu: ~]
[Reading and writing bi...
/home/andi/Computer...
[Mmap]
[andi@ubuntu: ~/Com...

andi@ubuntu: ~/ComputerArchitecture/Mmap/Build

File Edit View Search Terminal Help

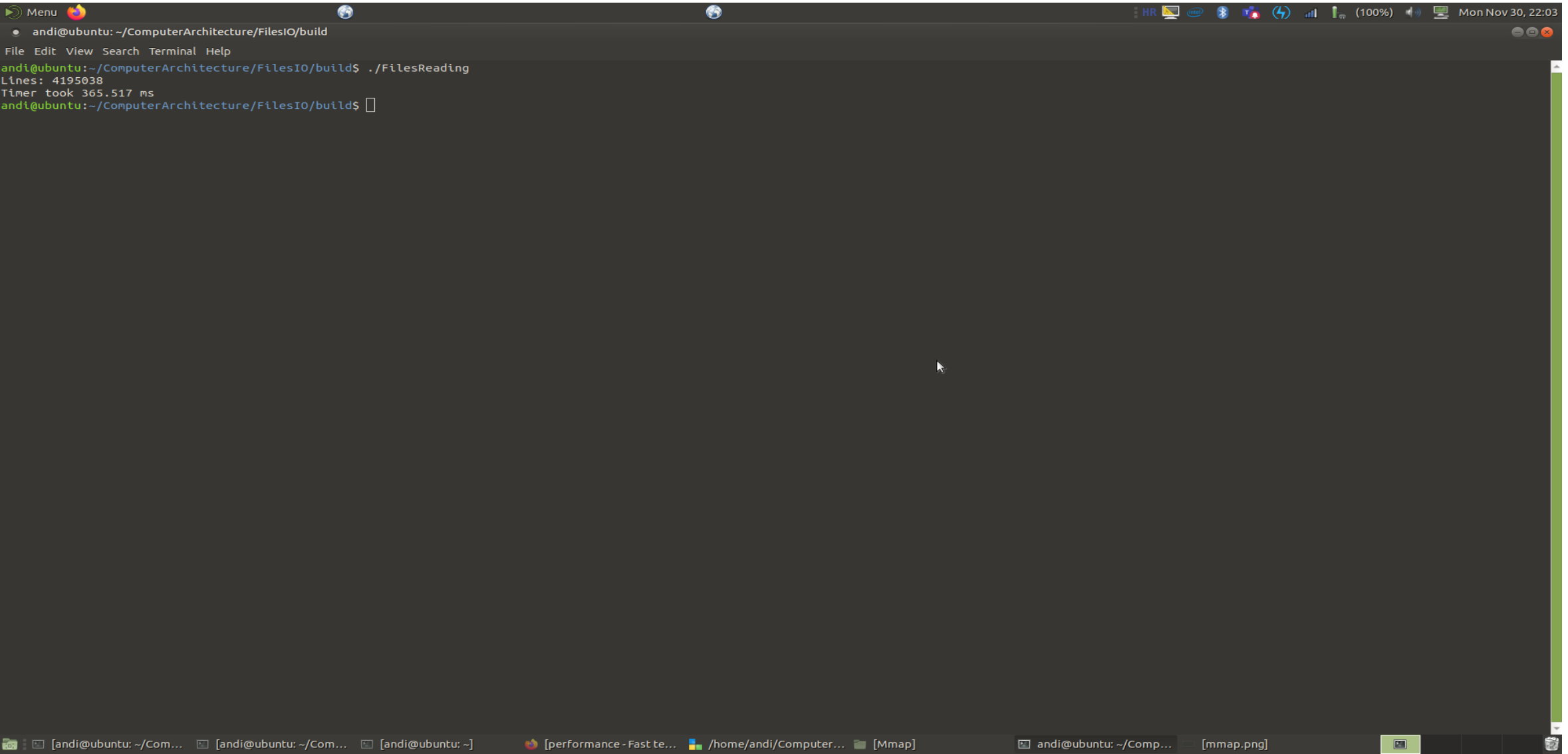
```
andi@ubuntu:~/ComputerArchitecture/Mmap/Build$ ./Mmap
Lines: 4195037
Timer took 158.489 ms
andi@ubuntu:~/ComputerArchitecture/Mmap/Build$
```

NORMALNO ČITANJE IZ DATOTEKE

```cpp
#include <iostream>
#include <fstream>
#include "../IO/Timer.h"

//reads one 1GB file with ifstream
// to create 1GB file run dd if=/dev/urandom of=file.txt bs=1048576 count=1024
void readFile() {
    std::ifstream input;
    input.open("../1GB.in");
    if(!input) {
        std::cout << "Cannot open file" << std::endl;
        return;
    }
    std::string line;
    int lines = 0;
    while(getline(input, line)) {
        lines++;
    }
    std::cout << "Lines: " << lines << std::endl;
}


int main() {
    Timer timer;
    readFile();

}
```

USPOREDBA

- Duplo brže se izvodi program koji koristi memorijsko mapiranje datoteke veličine 1GB



BINARNI UI

- Razmislimo o jednom čestom načinu korištenja dva programa:
  1. Prvi program stvara strukturu podataka i zapisuje u datoteku
  2. Drugi program čita podatke iz datoteke i koristi ih

2 načina implementacije, prvo pomoću zapisivanja teksta u odredišnu datoteku, a drugi način je zapisivanje u binarnu datoteku.

Za procjenu rezultata koristili smo datoteku veličine 81MB koju smo zapisivali i čitali najprije u txt datoteku, a onda u .dat datoteku.

Pogledajmo rezultate ☺

```cpp
#include <unordered_map>
#include <iostream>
#include <string>
#include "../IO/Timer.h"
#include <fstream>


#define NUM_ACTIONS 10000000

//Simulations of typical use, we create some object in file and then need to pass it to other file when this process ends. That's the reason why we
//are allocating space on heap two times when we could actually only once. WE SIMULATE STORING IN THIS FILE AND READING FROM ANOTHER FILE.
//enum needs bytes accordingly to information that stores. For example size of ActionType is ~~4bits~~ bytes

enum ActionType {PAST, PRESENT, FUTURE};

std::unordered_map<std::string, enum ActionType> conversion_table;



//we only care about performance not about data
class Action {
    public:
        std::string actionName;
        ActionType actionType;
        int actionIndex;

        Action() {
            actionName = "RandomName";
            actionType = PRESENT;
            actionIndex = 1;
        }

        void printAction() {
            std::cout << actionName << " " << actionType << " " << actionIndex << std::endl;
        }
```

```
        ~Action() {

        }
};


Action* createActions() {
    Action* actions= new Action[NUM_ACTIONS];
    return actions;
}


//neglect reserving on heap and deleting objects from heap
void storeActionsAsText() {
    Action* actions = createActions();
    std::ofstream stream("Actions.txt");
    for(int i = 0; i < NUM_ACTIONS; i++) {
        stream << actions[i].actionName << " " << actions[i].actionType << " " << actions[i].actionIndex << std::endl;
    }
    delete[] actions;
}

//neglect reserving on heap and deleting objects from heap
void readActionsFromText() {
    std::ifstream stream("../Action.txt");
    Action* actions = createActions();
    for(int i = 0; i< NUM_ACTIONS; i++) {
        std::string enumName;
        stream >> actions[i].actionName;
        stream >> enumName;
        actions[i].actionType = conversion_table[enumName];
        stream >> actions[i].actionIndex;
    }
    delete[] actions;
}
```

```
void simulateText() {
    Timer timer;
    storeActionsAsText();
    readActionsFromText();
}

void storeActionsAsBinary() {
    Action* actions = createActions();
    std::ofstream wb("../actions.dat", std::ios::out | std::ios::binary);
    if(!wb) {
        std::cout << "Cannot open file for writing" << std::endl;
        return;
    }
    for(int i = 0; i < NUM_ACTIONS; i++) {
        wb.write((char*) &actions[i], sizeof(Action));
    }
    wb.close();
    if(!wb.good()) {
        std::cout << "Error occurred while writing to file" << std::endl;
        return;
    }
    delete[] actions;
}

void readActionsAsBinary() {
    Action* actions = createActions();
    std::ifstream rf("../actions.dat", std::ios::out | std::ios::binary);
    if(!rf) {
        std::cout << "Cannot open file for reading!" << std::endl;
        return;
    }
    for(int i = 0; i < NUM_ACTIONS; i++) {
        rf.read((char*) &actions[i], sizeof(Action));
    }
    rf.close();
```

```
    if(!rf.good()) {
        std::cout << "Error occurred at reading time!" << std::endl;
        return;
    }
    delete[] actions;
}


void simulateBinary() {
    Timer timer;
    storeActionsAsBinary();
    readActionsAsBinary();
}




int main() {
    conversion_table["PAST"] = ActionType::PAST;
    conversion_table["PRESENT"] = ActionType::PRESENT;
    conversion_table["FUTURE"] = ActionType::FUTURE;
    std::cout << "Text simulation: " << std::endl;
    simulateText();
    std::cout << std::endl << "Binary simulation: " << std::endl;
    simulateBinary();
}
```

```
Timer took 17426.5 ms

Binary simulation:
Timer took 1321.14 ms
andi@ubuntu:~/ComputerArchitecture/CustomIO/Build$ ./io
Text simulation:
Timer took 17120.9 ms

Binary simulation:
Timer took 1273.58 ms
andi@ubuntu:~/ComputerArchitecture/CustomIO/Build$ ./io
Text simulation:
Timer took 17060.3 ms

Binary simulation:
Timer took 1313.96 ms
andi@ubuntu:~/ComputerArchitecture/CustomIO/Build$ ./io
Text simulation:
Timer took 17165.6 ms

Binary simulation:
Timer took 1300.05 ms
andi@ubuntu:~/ComputerArchitecture/CustomIO/Build$ ./io
Text simulation:
Timer took 17390.7 ms

Binary simulation:
Timer took 1235.92 ms
andi@ubuntu:~/ComputerArchitecture/CustomIO/Build$ ./io
Text simulation:
Timer took 16980.2 ms

Binary simulation:
Timer took 1249.22 ms
andi@ubuntu:~/ComputerArchitecture/CustomIO/Build$ ./io
Text simulation:
Timer took 17020 ms

Binary simulation:
Timer took 1277.06 ms
andi@ubuntu:~/ComputerArchitecture/CustomIO/Build$ ./io
Text simulation:
Timer took 17230.6 ms

Binary simulation:
Timer took 1244.97 ms
andi@ubuntu:~/ComputerArchitecture/CustomIO/Build$ ./io
Text simulation:
Timer took 17507.7 ms

Binary simulation:
Timer took 1337.59 ms
andi@ubuntu:~/ComputerArchitecture/CustomIO/Build$
```

BINARNI UI: ZAKLJUČAK

- Brži način zapisivanja podataka u bajtovima
  - Npr. Txt datoteka će za broj 12345 koristiti 5 bajtova, a binarna samo 2
  - Konačna binarna datoteka je puno manja, nego tekstualna pa je zato I takav način puno efikasniji

- Txt datoteka je čitljiva ljudima, a binarne datoteke ne
- Binarni UI je teže implementirati I koristiti, za datoteke relativno male veličine nemamo toliku dobit
- Biblioteke kao što su Boost nam mogu pomoći u tome
- Kod deserijalizacije treba paziti na "endianess"

SERIJALIZACIJA I PROTOCOL BUFFER

- Google ih je razvio
  - Amazon -> Ion https://amzn.github.io/ion-docs/
  - Facebook - >Apache Thrift https://thrift.apache.org/
- Metoda serijaliziranja struktura podataka
  - Ideja: jednostavno, sigurno, brzo I lako mrežno komuniciranje
  - Machine readable
  - ASCII format serijalizacije
  - Može se koristiti I za spremanje podataka
    - Ideja: strukture podataka = poruke su opisane u .proto datoteci
    - Ta se datoteka prevađa sa protoc komapjlerom
    - Prevedeni kod onda koriste hipotetski pošiljatelj I primatelj

PROTOCOL BUFFER

```
//polyline.proto
syntax = "proto2";

message Point {
  required int32 x = 1;
  required int32 y = 2;
  optional string label = 3;
}

message Line {
  required Point start = 1;
  required Point end = 2;
  optional string label = 3;
}

message Polyline {
  repeated Point point = 1;
  optional string label = 2;
}
```

```
// polyline.cpp
#include "polyline.pb.h"  // generated by calling "protoc polyline.proto"

Line* createNewLine(const std::string& name) {
  // create a line from (10, 20) to (30, 40)
  Line* line = new Line;
  line->mutable_start()->set_x(10);
  line->mutable_start()->set_y(20);
  line->mutable_end()->set_x(30);
  line->mutable_end()->set_y(40);
  line->set_label(name);
  return line;
}

Polyline* createNewPolyline() {
  // create a polyline with points at (10,10) and (20,20)
  Polyline* polyline = new Polyline;
  Point* point1 = polyline->add_point();
  point1->set_x(10);
  point1->set_y(10);
  Point* point2 = polyline->add_point();
  point2->set_x(20);
  point2->set_y(20);
  return polyline;
}
```

ZAVRŠNO O IO OPERACIJAMA

- Napomenimo za kraj kako korištenje '\n' u odnosu na std::endl također ubrzava izvođenje
  - Prvi svakom pozivu std::endl napravi se i flush stream-a

- Scanf i printf su brži od cin i cout, ali možemo jednako brzo čitati sa cin i cout uz pomoć ove dvije naredbe u main funkciji

- `ios_base::sync_with_stdio(false); cin.tie(NULL);`
  - uz using namespace std ;☺

LITERATURA

- https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture#:~:text=The%20x64%20architecture%20is%20a,sets%20are%20close%20to%20identical
- http://gee.cs.oswego.edu/dl/html/malloc.html
- https://www.youtube.com/watch?v=wJ1L2nSIV1s&list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb&index=54
- https://linuxhint.com/using_mmap_function_linux/
- https://en.wikipedia.org/wiki/Protocol_Buffers
- http://www.tutorialdost.com/C-Programming-Tutorial/27-C-File-IO-Handling.aspx
- https://en.wikipedia.org/wiki/Allocator_(C%2B%2B)#Custom_allocators
- https://geidav.wordpress.com/2013/03/21/anatomy-of-dynamic-stack-allocations/
- https://www.cs.swarthmore.edu/~newhall/cs31/resources/addrleal.php
- https://sourceware.org/binutils/docs/as/P2align.html#P2align
- https://www.techspot.com/article/2166-mmx-sse-avx-explained/
- https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- https://www.rapidtables.com/code/linux/gcc/gcc-o.html

KRAJ

- https://github.com/as51340/Memory-management-in-C

Andi Škrgat

Mentor: profesor Siniša Šegvić