

# **MODUL PRAKTIKUM #7**

## **1031202/1041202 - Sistem Operasi**

### **MANAJEMEN MEMORI**

<b>Minggu</b>	:	13/2&3
<b>Setoran</b>	:	Laporan praktikum dan kode program
<b>Batas Waktu Setoran</b>	:	-
<b>Tujuan</b>	:	Mahasiswa mampu mampu menulis program untuk mengimplementasikan pengelolaan memori pada Linux

#### **Petunjuk Praktikum:**

1. Anda mengerjakan tugas ini secara mandiri.
2. Mencontoh pekerjaan dari orang lain akan dianggap plagiarisme dan anda akan ditindak sesuai dengan sanksi akademik yang berlaku di IT Del atau sesuai dengan kebijakan saya dengan memberikan nilai 0.
3. Gunakan Sistem Operasi Linux boleh menggunakan Distro apapun namun disarankan untuk mempermudah praktikum gunakan Ubuntu.

#### **Referensi:**

1. L. Robert, Linux System Programming, 2<sup>nd</sup> edition, Chapter 9, O'Reilly Media, Inc., 2013.
2. Tanenbaum S. Andrew, Bos Herbert, Modern Operating System, 4<sup>th</sup> edition, Pearson, 2015.

# **PEMROGRAMAN**

## **A. MEMORY REGIONS**

Setiap proses pada sistem operasi Linux mempunyai ruang alamat secara logik yang terdiri dari tiga segmen yaitu text, data dan stack.

- Segmen text terdiri atas instruksi mesin yang dihasilkan oleh compiler dan assembler sebagai contoh translasi dari Bahasa pemrograman C, C++ atau Bahasa pemrograman lainnya menjadi kode mesin.
- Segment data terdiri atas variabel global yang ada di kode program dengan dua bagian yaitu variabel yang telah diinisialisasi atau belum diinisialisasi.
- Segmen stack digunakan untuk menyimpan lokal variabel yang pengalamatan memorinya telah dilakukan pada saat kompilasi.

## **Allocating Dynamic Memory**

Memori dapat dialokasikan saat run-time sebagai contoh, ketika data dibutuhkan dari sebuah file atau masukan pengguna melalui keyboard maka ukuran file ataupun *keystroke* yang diketikkan oleh pengguna tidak dapat diprediksi. Dengan demikian dibutuhkan buffer yang dapat menampung data sebanyak mungkin untuk memungkinkan data dari file ataupun dari keyboard dapat dibaca. Alokasi memori secara dinamis dapat dilakukan dengan menggunakan malloc() atau *memory allocation* yang akan meminta ruang memori dari sistem operasi sesuai dengan yang dibutuhkan. Malloc() adalah *interface* yang disediakan oleh C untuk mendukung alokasi memori secara dinamis.

```
#include <stdlib.h>
void * malloc (size_t size);
```

Malloc() yang berhasil dipanggil akan mengalokasikan memori dalam ukuran bytes dan mengembalikan sebuah pointer ke ruang memori yang dialokasikan. Jika Malloc() tidak berhasil maka akan mengembalikan NULL pointer. Oleh karena itu, sangat penting untuk selalu menangani kondisi error dan menghentikan program (terminasi) apabila malloc() mengembalikan NULL. Contoh penggunaan malloc() dapat dilihat pada kode program dibawah. Pada contoh ini ukuran memori yang diminta untuk dialokasikan adalah sebesar 2KB. Ketikkanlah program tersebut dengan nama program malloc\_1.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     char *p;
6
7     /* meminta memori dialokasikan sebesar 2KB */
8     p = malloc (2048);
9
10    if(p = NULL){
11        printf("Memori tidak dialokasikan\n");
12        exit (0);
13    }else{
14        printf("Memori dapat dialokasikan\n");
15    }
16
17    return 0;
18 }
19

```

Buatlah contoh program malloc() di bawah dengan nama program malloc\_2. Berilah penjelasan Anda terhadap tujuan kode program tersebut dan tampilkan hasil eksekusi program.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int *x, num_element=4;
6
7     x = malloc (num_element * sizeof(int));
8
9     if(!x){
10        perror("Memori tidak dapat dialokasikan menggunakan malloc");
11        return -1;
12    }else{
13        printf("Memori dapat dialokasikan menggunakan malloc\n");
14
15        for(int i=0; i<num_element; i++){
16            x[i] = i + 1;
17        }
18
19        printf("Element: ");
20
21        for(int i=0; i<num_element; i++){
22            printf("%d ", x[i]);
23        }
24
25    }
26
27    return 0;
28 }

```

### **Allocating Array**

Selain malloc(), terdapat calloc() atau contiguous allocation yang digunakan untuk mengalokasikan memori dengan tipe yang telah ditentukan. Setiap blok memori akan diinisialisasi dengan nilai 0.

```
#include <stdlib.h>
void * malloc (size_t nr, size_t size);
```

Dengan demikian, perhatikanlah kode di bawah ini, berikanlah penjelasan Anda terhadap perbedaan antara malloc() dan calloc(), berilah nama program calloc\_1. Kemudian, modifikasilah kode program malloc\_2 dengan menggunakan calloc() dengan nama program calloc\_2.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int *x, *y;
6
7     x = malloc (50 * sizeof(int));
8
9     if(!x){
10         perror("Memori tidak dapat dialokasikan menggunakan malloc");
11         return -1;
12     }else{
13         printf("Memori dapat dialokasikan menggunakan malloc\n");
14     }
15
16
17     y = calloc (50, sizeof(int));
18
19     if(!y){
20         perror("Memori tidak dapat dialokasikan menggunakan calloc");
21         return -1;
22     }else{
23         printf("Memori dapat dialokasikan menggunakan calloc\n");
24     }
25
26     return 0;
27 }
28
```

### **Resizing Allocations**

Fungsi realloc() digunakan untuk mengubah ukuran dari blok memori yang dialokasikan secara dinamis sebelumnya. Dengan demikian, realloc() akan menyesuaikan ukuran dari blok memori terhadap ukuran yang baru.

```
#include <stdlib.h>
void * realloc (void *p, size_t size);
```

dimana p adalah pointer menuju blok memori sekarang ini, sedangkan size adalah ukuran blok memori yang baru. Fungsi ini akan mengembalikan pointer kepada blok memori yang ukurannya telah diubah, atau mengembalikan NULL jika permintaan perubahan blok memori gagal. Dengan menggunakan realloc() tidak akan mengubah nilai isi dari blok memori yang telah dialokasikan sebelumnya, tetapi akan menambah jumlah blok memori yang berisi nilai yang belum diinisialisasi. Untuk memahami realloc(), ketikkanlah kode program di bawah dan beri penjelasan terhadap hasil pengamatan dari luaran kode program tersebut.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(){
6     char *s;
7     char *p = "manajemen";
8
9     s = (char*) malloc((strlen(p)+1));
10
11     //printf("%s", p);
12
13     if(s == NULL){
14         printf("Memori tidak dapat dialokasi menggunakan malloc");
15     }else{
16         strcpy(s, p);
17         printf("String dari p adalah %s, alamat memori: %p\n", s, &s);
18
19         s = (char*) realloc(s, ((strlen(p)+10)));
20         strcat(s, " memori");
21         printf("String dari reallocation adalah %s, alamat memori: %p\n", s, &s);
22     }
23
24     free(s);
25
26     return(0);
27
28
29 }

```

### **Freeing Dynamic Memory**

Blok memori yang telah dialokasikan secara dinamis harus dibebaskan secara manual. Seorang programmer harus bertanggungjawab untuk mengembalikan blok memori yang dialokasikan secara dinamis kepada sistem. Alokasi memori dengan menggunakan fungsi malloc(), calloc() atau realloc() harus dikembalikan kepada sistem dengan menggunakan fungsi free().

```

#include <stdlib.h>
void * free (void *ptr);

```

ketikkanlah kode progam di bawah dan beri penjelasan terhadap hasil pengamatan dari luaran kode program tersebut.

```

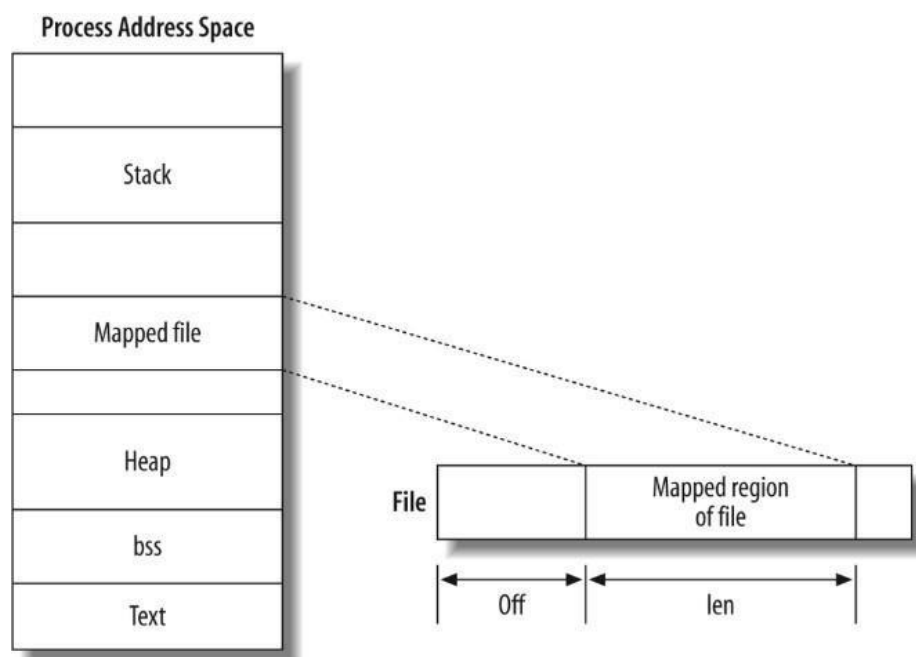
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(){
6     char *s;
7     char *p = "malloc";
8
9     s = (char*) malloc((strlen(p)+1));
10
11     //printf("%s", p);
12
13     if(s == NULL){
14         printf("Memori tidak dapat dialokasi menggunakan malloc");
15     }else{
16         strcpy(s, p);
17         printf("String dari p adalah %s\n", s);
18     }
19
20     free(s);
21
22 }

```

## B. MEMORY-MAPPED FILE

**Memory-Mapped file** pada Linux berfungsi untuk memetakan blok dari disk atau berkas ke halaman memori virtual dari setiap proses, seperti pada Gambar 1. Dengan demikian, berkas akan dapat diakses secara langsung ke memori dibandingkan harus melakukan akses **direct I/O** melalui *system call* untuk mengakses berkas di harddisk. Dengan adanya MMF, maka **shared-memory IPC (Inter-Process Communication)** dapat dilakukan yang memungkinkan dua atau lebih proses dapat membaca (**read**) atau menulis (**write**) ke dalam halaman memori yang sama. Untuk mendukung MMF, Linux mengimplementasikan sebuah *system call* yaitu **mmap()** yang akan memetakan berkas ke halaman memori. Fungsi **mmap()** akan menciptakan sebuah pointer ke halaman memori yang terkait dengan isi berkas yang diakses melalui **file descriptor** dari berkas yang dibuka.

```
#include <sys/man.h>
void *mmap(void *addr, size_t len, int prot, int flags, int
fildes, off_t off);
```



Gambar 1 Memory-Mapped File

Fungsi **mmap()** beroperasi pada **Page** yang merupakan unit memori terkecil. Dengan demikian, parameter **off** dan **len** harus selaras dengan ukuran dari **Page**. Ukuran kedua parameter tersebut harus kelipatan dari ukuran **Page**. Saat ini, Anda akan menuliskan program sederhana dengan mengaplikasikan **mmap()** pada proses enkripsi dengan menggunakan Rail Fence Cipher. Kode program dan penjelesannya dapat dilihat pada halaman selanjutnya.

## Rail Fence Chiper

Rail Fence Chiper adalah algoritma kriptografi klasik yang termasuk ke dalam kategori model algoritma transposisi. Algoritma ini mengenkripsi pesan asli (*plaintext*) menggunakan teknik perubahan posisi dari setiap huruf pada pesan asli, tanpa mengubah huruf sebenarnya. Teknik ini salah satu dari teknik kriptografi yang memerlukan kunci khusus atau yang disebut “*key*” untuk melakukan enkripsi pesan asli. Kemudian, hasil enkripsi dari pesan asli ini akan disebut dengan istilah *chipertext*. Langkah-langkah yang dilakukan untuk mengenkripsi pesan asli tersebut adalah:

1. Membentuk matriks sebagai wadah untuk mengatur ulang urutan huruf dari pesan asli dengan ketentuan jumlah kolom = panjang pesan asli, sedangkan jumlah baris = *key*.
2. Menuliskan huruf dari pesan asli secara diagonal baris-per-baris.
3. Membaca susunan huruf yang baru yang telah dienkripsi (*chipertext*) secara baris-per-baris.

Misalnya diketahui sebuah *plaintext* DEFENDTHEEASTWALL, dengan *key* = 2 dan panjang pesan asli = 17. Berdasarkan langkah-langkah di atas (dari 1 sd. 2), maka proses enkripsi pesan asli DEFENDTHEEASTWALL dapat digambarkan sebagai berikut.

D		F		N		T		E		A		T		A		L
	E		E		D		H		E		S		W		L	

Sehingga, sesuai dengan langkah ke-3 maka *chipertext* yang didapatkan adalah DFNTEATALEEDHESWL.

## Prosedur

1. Buat dua file yang berekstensi.txt dengan nama **plaintext.txt** dan **chipertext.txt**. File **plaintext.txt** akan berisi data yang akan dienkripsi, sebagai contoh tuliskan **Hello World**. Selanjutnya, data yang telah dienkripsi disimpan di dalam file **chipertext.txt**.
2. Buat file kode dengan nama **railFenceChiper.c**. Tambahkan header file yang dibutuhkan (baris 1 sd. 5) dan deklarasi fungsi yang diberi nama **encryptPlainText()** (baris 7), seperti pada kode berikut.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <sys/mman.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6
7 void encryptPlainText(char *text, int key);
```

3. Pada file yang sama tambahkan kode pada fungsi main yang diberikan pada halaman selanjutnya.



```

9 int main(){
10     int fd, pagesize, key=3;
11     char *ptr;
12
13     fd = open("plaintext.txt", O_RDWR | O_CREAT, S_IRWXU);
14     pagesize = getpagesize();
15     ptr = (char*) mmap(NULL, pagesize, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
16
17     if(ptr == MAP_FAILED){
18         perror("Error: ");
19         exit(1);
20     }
21
22     close(fd);
23
24     printf("Mapped at %p\n", ptr);
25     printf("Plaintext: %s \n", ptr);
26     encryptPlainText(ptr, key);
27
28     munmap(ptr, pagesize);
29
30     return (0);
31 }

```

4. Baris 10 sd. 11 mendeklarasikan variabel yang diperlukan.
5. Pada baris ke-13 *system call* **open ()** menerima 3 parameter yaitu file plaintext.txt yang berisi data yang akan dienkripsi, *flag* **O\_RDWR** dengan model **read only** atau *flag* **O\_CREAT** yang akan membuat file baru apabila file plaintext.txt belum tercipta. Sedangkan, parameter ke-tiga adalah mode **S\_IRWXU** dengan akses **00700** yang memberi akses baca, tulis, dan eksekusi (**read**, **write**, **execute**) kepada pemilik file.
6. Baris 15 adalah fungsi **mmap ()** yang dipanggil untuk memetakan file yang telah dibuka ke memori. Pada fungsi **mmap ()** terdapat 6 argumen sebagai berikut:
  - a. Argumen pertama pada fungsi ini diberi **NULL** pointer yang menyatakan bahwa alokasi alamat memori yang akan dipetakan akan diserahkan kepada sistem operasi.
  - b. Argumen kedua adalah panjang data yang akan dipetakan ke memori. Panjang data didapatkan dari baris 14 dengan menggunakan fungsi **getpagesize ()** yang mengembalikan ukuran dari *page virtual memory*.
  - c. Argumen ketiga adalah "*protection*" yang menyatakan akses terhadap area memori yang dipetakan. Pada kode mode *protection* yang digunakan adalah **PROT\_READ** untuk akses baca (*read*) atau **PROT\_WRITE** untuk akses tulis (*write*).
  - d. Argumen keempat adalah *flag* yang terdiri atas dua pilihan yaitu **MAP\_PRIVATE** atau **MAP\_SHARED**. Jika perubahan pada file akan dibagi dengan proses yang lain maka *flag* yang digunakan adalah **MAP\_SHARED**.
  - e. Argumen kelima adalah *file descriptor* yang telah dibuka sebelumnya pada baris ke-13.
  - f. Argumen keenam adalah *offset* dari file yang akan dipetakan.
7. Pada baris 17 sd. 20 adalah penanganan jika fungsi **mmap ()** gagal. Kemudian pada baris ke-21 *file descriptor* ditutup dengan menggunakan *system call* **close ()**.



8. Pada baris ke-26 fungsi untuk melakukan enkripsi dipanggil melalui fungsi **encryptPlainText(ptr, key)** dengan dua parameter yaitu ptr (data yang telah dipetakan) dan key yang digunakan untuk mengenkripsi plaintext.
9. Setelah fungsi **mmap()** selesai digunakan maka harus di-unmap dengan menggunakan fungsi **munmap()**, dengan parameter data yang telah dipetakan dan panjang data.
10. Berikutnya kode pada fungsi **encryptPlainText(char \*text, int key)** yang mengimplementasikan Rail Fence Cipher.

```

33 void encryptPlainText(char *text, int key){
34     FILE *fp;
35     int i, j, k = -1, col = 0, row = 0;
36     int textLength = strlen(text);
37     char rail[key][textLength];
38
39     for(i=0;i<key;++i){
40         for(j=0;j<textLength;++j){
41             rail[i][j] = '\n';
42         }
43     }
44
45     for(i=0;i<textLength;++i){
46         rail[row][++col] = text[i];
47
48         if(row == 0 || row == key-1){
49             k = k * (-1);
50         }
51         row = row + k;
52     }
53
54     fp = fopen("chipertext.txt", "w+");
55
56     if(fp = NULL){
57         perror("Error: ");
58         exit(1);
59     }
60

```

11. Dimulai dari baris 34 sd. 37 adalah deklarasi variabel yang diperlukan. Selanjutnya, pada baris 39 sd. 52 adalah kode untuk melakukan enkripsi.
12. Hasil enkripsi atau *ciphertext* akan ditulis ke dalam file dengan nama **chipertext.txt** pada baris ke-54.
13. Data yang telah dienkripsi akan ditampilkan pada layar menggunakan kode pada baris ke-65, dan akan dituliskan ke dalam file pada baris ke-66. Setelah itu file chipertext.txt ditutup dengan memanggil fungsi **fclose()** pada baris ke-71. Kode program di bawah masih merupakan lanjutan dari fungsi encryptPlainText().

```

61 printf("Encrypted Plaintext: ");
62 for(i=0;i<key;++i){
63     for(j=0;j<textLength;++j){
64         if(rail[i][j] != '\n'){
65             printf("%c", rail[i][j]);
66             fprintf(fp, "%c", rail[i][j]);
67         }
68     }
69 }
70
71 fclose(fp);
72 printf("\n");
73 }

```

14. Hasil enkripsi dari **Hello World** adalah **Horel ollWd** seperti pada tampilan gambar di bawah.

```

./mmap_railFenceChiper
Mapped at 0x7fc433486000
Plaintext: Hello World
Encrypted Plaintext: Horel ollWd

```

15. Tampilkan hasil enkripsi dengan menggunakan perintah **cat nama\_file**.

```

cat chipertext.txt
Horel ollWd

```

### C. SHARED-MEMORY IPC BY MMF

Cara komunikasi antar proses yang paling efisien adalah melalui akses memori secara langsung. Hal ini dimungkinkan dengan menggunakan salah satu mekanisme dari *Inter-Process Communication* yaitu *Shared-Memory*. Dengan mekanisme ini, memori dapat diakses secara bersamaan oleh beberapa proses. Kinerja *shared-memory* ditingkatkan oleh fungsi **mmap()** atau *map pages of memory* yang dapat memetakan blok dari disk atau berkas ke halaman memori virtual dari setiap proses. Dengan demikian, kernel tidak dilibatkan lagi untuk melewatkan data antar proses melalui akses I/O secara langsung.

Untuk dapat menggunakan *shared-memory* sebuah proses harus terlebih dahulu menciptakan objek dari *shared-memory* melalui *system call* **shm\_open()**. Terdapat tiga parameter pada *system call* **shm\_open()**, yaitu parameter **name** yang menetapkan nama objek *shared-memory*. Proses akan menggunakan nama objek ini untuk mengakses *shared-memory* yang dituju. Parameter kedua adalah **flags** yang terdiri atas **O\_CREAT** atau **O\_RDWR**. *Flag* **O\_CREAT** akan menciptakan object *shared-memory* jika objek belum tercipta, sedangkan *flag* **O\_RDWR** menyatakan bahwa objek terbuka untuk ditulis dan dibaca. Parameter ketiga adalah *permission* atau izin terhadap hak akses dari objek. Jika *system-call* **shm\_open()** berhasil maka nilai yang akan dikembalikan adalah integer file descriptor dari objek *shared-memory*.

Selanjutnya, setelah objek terbentuk fungsi **ftruncate()** digunakan untuk mengkonfigurasi ukuran objek dalam bytes. Terdapat dua parameter pada fungsi

**ftruncate()** yaitu file descriptor dari shared-memory, sedangkan parameter kedua adalah ukuran objek. Fungsi berikutnya adalah fungsi **mmap()** yang akan membentuk Memory-Mapped File (MMF) dengan salah satu parameter dari fungsi ini adalah file descriptor dari shared-memory.

Selanjutnya, Anda akan mengimplementasikan *shared-memory* IPC dan **mmap()**. Pada kode program berikutnya terdapat dua proses yaitu proses *Sender* yang membuat pemetaan file, sedangkan proses kedua adalah *Receiver* yang membuka pemetaan file. Kedua proses akan saling berkomunikasi dengan mengakses pemetaan file tersebut dengan cara mengirimkan pesan.

### Prosedur untuk program Sender

1. Buat kode program baru dengan nama berkas sender.c, kemudian tambahkan kode berikut.

```
1 #include<stdio.h>
2 #include<string.h>
3 #include<sys/types.h>
4 #include<sys/mman.h>
5 #include<sys/wait.h>
6 #include<unistd.h>
7 #include<fcntl.h>
8 |
9 #define STORAGE_ID "/shm-test"
10 #define STORAGE_SIZE 64
11 #define DATA "Hello receiver...I am sender with PID %d"
12
13 int main(){
14     int shmFd, shmSize, pagesize, fd, length;
15     char *ptr, data[STORAGE_SIZE];
16     FILE *fp;
17     pid_t pid;
18
19     pid = getpid();
20     sprintf(data, DATA, pid);
21
22     pagesize = getpagesize();
23     shmFd = shm_open(STORAGE_ID, O_CREAT | O_EXCL | O_RDWR, 0600);
24     if(shmFd == -1){
25         perror("Error shm: ");
26     }
27
28     shmSize = ftruncate(shmFd, STORAGE_SIZE);
29     if(shmSize == -1){
30         perror("Error ftruncate: ");
31     }
```

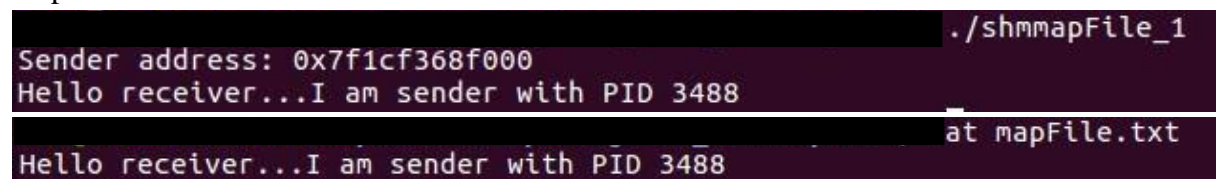
2. Tambahkan header file yang dibutuhkan (baris 1 sd. 7), kemudian tentukan nama dari shared-memory object seperti pada baris 9, storage\_size pada baris 10, dan data yang akan dikirimkan dari sender ke receiver pada baris ke-11.
3. Pada baris ke-14 sampai dengan 17 adalah deklarasi variabel yang diperlukan, sedangkan baris ke-19 akan mengembalikan ID proses.
4. Data yang telah ditentukan pada baris ke-11 akan disimpan ke dalam buffer data.
5. Dimulai dari baris 23 sd. 26 Sender memanggil fungsi **shm\_open()** untuk membentuk objek shared-memory. Pada sistem operasi Linux objek ini dapat

diakses di `/dev/shm`. Selanjutnya, dari baris ke-28 sd. 31 Sender memanggil fungsi `ftruncate()` untuk menetapkan ukuran dari objek shared-memory.

6. Tambahkan kode pada halaman selanjutnya ke program Anda.
7. Selanjutnya Sender memanggil fungsi `mmap()` dimulai dari baris ke-33 sd. 36. Perhatikan bahwa file descriptor yang diberikan pada parameter `mmap()` adalah file descriptor dari shared-memory bukan file descriptor berkas.
8. Berikutnya sender akan mengirimkan pesannya ke proses Receiver dimulai dari baris ke-39 sd. 48. Sender menggunakan fungsi `memcpy()` untuk menyalin sebuah blok memori dari satu lokasi ke lokasi lain. Data yang disalin kemudian dituliskan ke dalam file yang bernama `mapFile.txt`. Setelah, file selesai dituliskan kemudian akan ditutup kembali.

```
32
33 ptr = mmap(0, pagesize, PROT_READ | PROT_WRITE, MAP_SHARED, shmFd, 0);
34 if(ptr == MAP_FAILED){
35     perror("Error MAP: ");
36 }
37 printf("Sender address: %p\n", ptr);
38
39 length = strlen(data) + 1;
40 memcpy(ptr, data, length);
41
42 fp = fopen("mapFile.txt", "w+");
43 if(fp == NULL){
44     perror("Error File: ");
45 }
46
47 fprintf(fp, "%s\n", ptr);
48 fclose(fp);
49
50 printf("%s\n", ptr);
51 munmap(ptr, STORAGE_SIZE);
52 close(shmFd);
53
54 return(0);
55 }
```

9. Proses sender dan receiver akan saling mengirimkan pesan melalui file `mapFile.txt` yang sudah dipetakan ke memori. Pada baris ke-50 pesan yang dikirimkan oleh sender kepada receiver dapat dicetak ke layar atau dapat dilihat dengan membuka file `mapFile.txt`.



```
./shmmapFile_1
Sender address: 0x7f1cf368f000
Hello receiver...I am sender with PID 3488
at mapFile.txt
Hello receiver...I am sender with PID 3488
```

10. Pada baris ke-51 memory-mapped file harus di-unmap, dilanjutkan pada baris ke-52 objek shared-memory harus ditutup.

## Prosedur untuk program Receiver

Setelah membuat kode program sender, berikutnya Anda akan menulis kode program receiver yang akan membuka pemetaan file. Ikutilah prosedur berikut ini:

1. Buat kode program baru dengan nama berkas `receiver.c`, kemudian tambahkan kode program di bawah.

```
1 #include<stdio.h>
2 #include<string.h>
3 #include<sys/types.h>
4 #include<sys/mman.h>
5 #include<sys/wait.h>
6 #include<unistd.h>
7 #include<fcntl.h>
8
9 #define STORAGE_ID "/shm-test"
10 #define STORAGE_SIZE 64
11 #define DATA "Hello Sender...I am receiver with PID %d"
12
13 int main(){
14     int shmFd, pagesize, length;
15     char *ptr, data[STORAGE_SIZE];
16     pid_t pid;
17     FILE *fp;
18     pid = getpid();
19     sprintf(data, DATA, pid);
20
21     pagesize = getpagesize();
22     shmFd = shm_open(STORAGE_ID, O_RDWR, 0777);
23     if(shmFd == -1){
24         perror("Error shm: ");
25     }
26
27     ptr = mmap(0, STORAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shmFd, 0);
28     if(ptr == MAP_FAILED){
29         perror("Error mmap: ");
30     }
31     printf("Receiver address: %p\n", ptr);
32 }
```

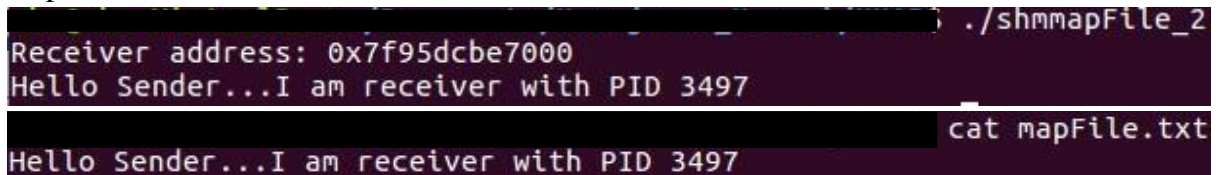
2. Perhatikan bahwa nama objek shared-memory harus sama dengan yang telah ditetapkan oleh sender yaitu `/shm-test`.
3. Kode program antara sender dan receiver memiliki kemiripan seperti menggunakan fungsi `shm_open()`, `ftruncate()`. Tetapi, Anda perlu memperhatikan bahwa parameter dari setiap fungsi berbeda antara *sender* dengan *receiver*, seperti pada baris 22 sd. 30.
4. Tambahkan kode program yang ada pada halaman selanjutnya.
5. Pada kode program receiver juga menggunakan fungsi `memcpy()` yang akan menyalin blok memori dari data yang akan dikirimkan kepada sender dengan cara menuliskannya pada file `mapFile.txt`.
6. Pada baris ke-45 `memoy-mapped` file harus di-unmap, dilanjut pada baris ke-46 objek shared-memory harus ditutup, kemudian pada baris ke-47 objek shared-memory dihapus (Anda dapat memeriksa ketersediaan objek di `/dev/shm`).

```

32
33 length = strlen(data) + 1;
34 memcpy(ptr, data, length);
35
36 fp = fopen("mapFile.txt", "w+");
37 if(fp == NULL){
38     perror("Error File: ");
39 }
40
41 fprintf(fp, "%s\n", ptr);
42 fclose(fp);
43
44 printf("%s\n", ptr);
45 munmap(ptr, STORAGE_SIZE);
46 close(shmFd);
47 shm_unlink(STORAGE_ID);
48 return(0);
49 }

```

7. Hasil dari data yang dikirimkan oleh receiver dapat dilihat pada layar atau file mapFile.txt.



The screenshot shows a terminal window with the following output:

```

./shmmapFile_2
Receiver address: 0x7f95dcbe7000
Hello Sender...I am receiver with PID 3497
cat mapFile.txt
Hello Sender...I am receiver with PID 3497

```

#### D. TUGAS PEMROGRAMAN

1. Buatlah sebuah program menggunakan alokasi memori secara dinamis menggunakan malloc() dan free() untuk menjumlahkan nilai yang diberikan oleh pengguna. Jumlah angka berada direntang 1 sd. 10, nilai dari setiap angka (dengan tipe data integer) diterima berdasarkan masukan dari pengguna kemudian dijumlahkan.
2. Buatlah sebuah program menggunakan alokasi memori secara dinamis menggunakan calloc() dan free() untuk menemukan nilai element yang paling besar dari array. Jumlah total element array sesuai dengan masukan pengguna dengan rentang dari 1 sd. 20 element.