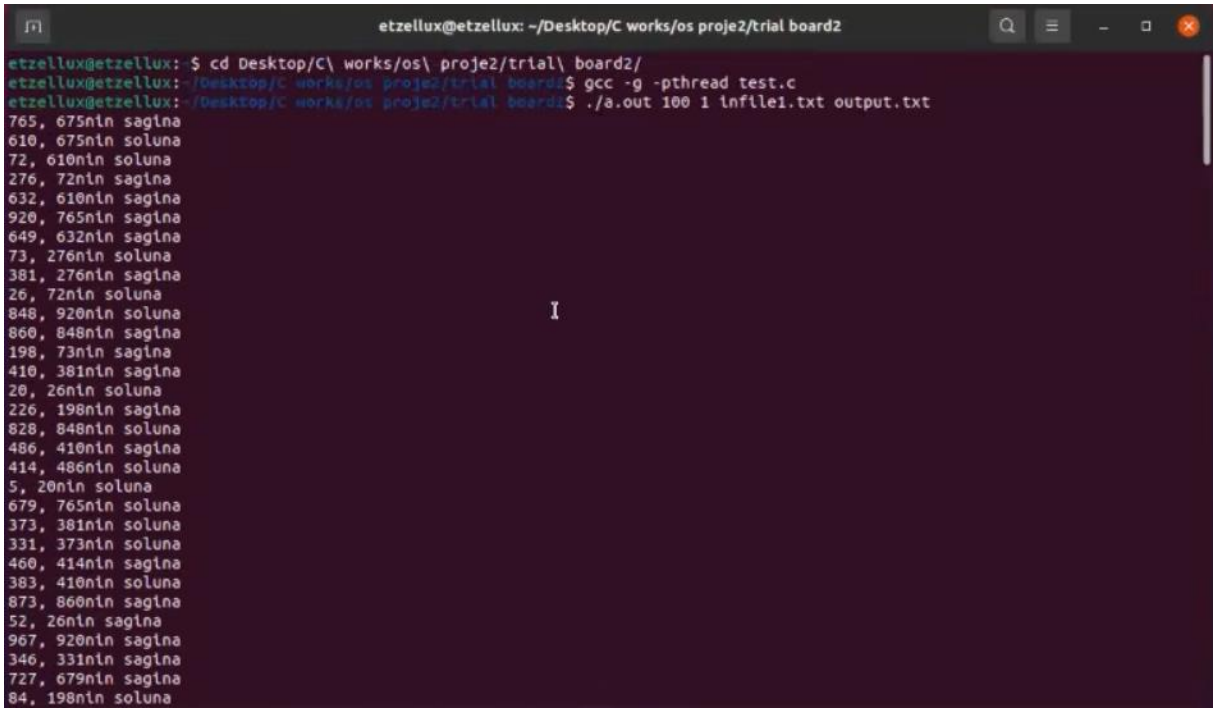
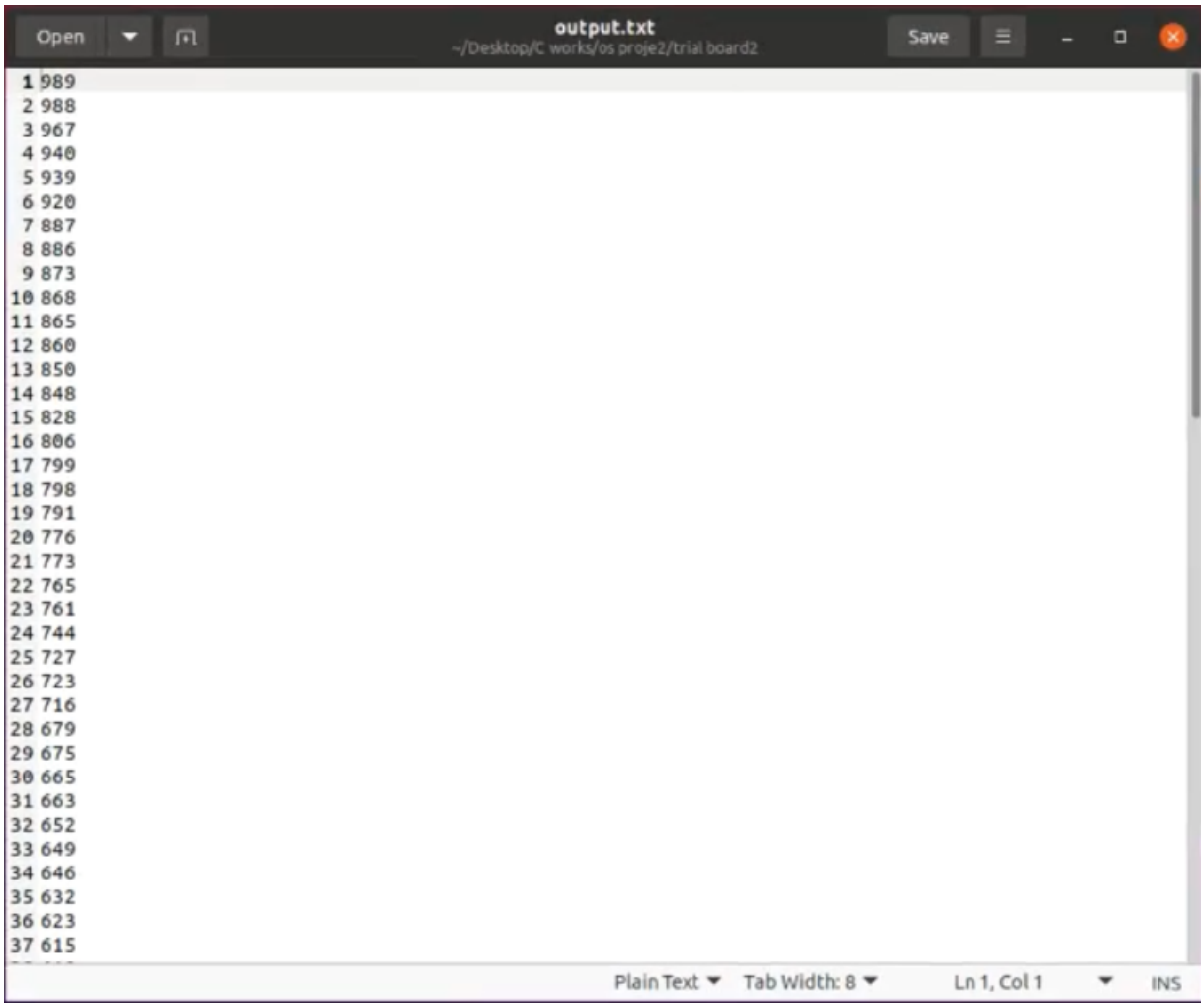
İşletim Sistemleri Proje 2 Rapor

PART – A

- Argümanlar terminalden input olarak alındı.
- Argümanların belirlenen değerler arasında olup olmadığı kontrol edildi.
- Thread senkronizasyonu için kullandığımız mutex yapısını başlattık.
- Bir for döngüsü ile istenilen sayıda thread oluşturuldu.
- Threadleri oluştururken readAndInsert fonksiyonu kullanıldı. Fonksiyon dosya ismini argüman olarak alıyor ve o dosyanın sonuna kadar okuma yapıyor. Eğer o thread ile fonksiyona aynı anda ulaşmaya çalışan bir başka thread yoksa insertData fonksiyonu çağırılıyor. Thread senkronizasyonu bu adımda kullanıldı.
- insertData fonksiyonunda ise gönderilen data Binary Search Tree' ye uygun şekilde yerleştirildi.
- Main fonksiyonunda pthread_join ile bütün threadlerin işlemlerini bitirmeleri bekleniyor.
- pthread_mutex_destroy ile mutex sonlandırılıyor.
- writeOutput fonksiyonu çalıştırılıyor argüman olarak Binary Search Tree'nin Rootunu, N'yi , K'yı ve Output dosyasını alıyor ve bunların içerisine istenen değerleri yazdırıyor.
- freeTree fonksiyonu da rekürsif bir şekilde Binary Search Tree oluştururken kullandığımız bellek alanını boşaltıyor.

Program Çıktısı




Open
output.txt
~/Desktop/C works/os proje2/trial board2
Save
1 989
2 988
3 967
4 940
5 939
6 920
7 887
8 886
9 873
10 868
11 865
12 860
13 850
14 848
15 828
16 806
17 799
18 798
19 791
20 776
21 773
22 765
23 761
24 744
25 727
26 723
27 716
28 679
29 675
30 665
31 663
32 652
33 649
34 646
35 632
36 623
37 615
Plain Text
Tab Width: 8
Ln 1, Col 1
INS

Ağaca Eleman Ekleme

```
void insertData(int data)
{
        struct node *newNode = (struct node*) malloc(sizeof(struct node));
        struct node *childNode;
        struct node *parentNode;

        newNode->data = data;
        newNode->leftNode = NULL;
        newNode->rightNode = NULL;

        if(root == NULL)
        {
                root = newNode;
        }
        else
        {
                childNode = root;
                parentNode = NULL;

                while(1)
                {
                        parentNode = childNode;

                        if(data < parentNode->data)
                        {
                                childNode = parentNode->leftNode;

                                if(childNode == NULL)
                                {
                                        parentNode->leftNode = newNode;
                                        printf("%d, %dnin soluna\n",newNode->data,parentNode->data);
                                        return;
                                }
                        }
                        else if(data > parentNode->data)
                        {
                                childNode = parentNode->rightNode;

                                if(childNode == NULL)
                                {
                                        parentNode->rightNode = newNode;
                                        printf("%d, %dnin sagina\n",newNode->data,parentNode->data);
                                        return;
                                }
                        }
                        else
                        {
                                return;
                        }
                }
        }
}

void* readAndInsert(void *arg)
{
        char *filename = (char*) arg;
        int buffer;
        unsigned int control;

        FILE* fp;
        fp = fopen(filename,"r");

        control = fscanf(fp,"%d\n",&buffer);

        while(control != EOF)
        {
                pthread_mutex_lock(&mutex);
                insertData(buffer);
                pthread_mutex_unlock(&mutex);
                control = fscanf(fp,"%d\n",&buffer);
        }
        fclose(fp);
}
```

Output Dosyası Oluşturma ve Ağacı Sonlandırma

```c
void writeOutput(struct node *Node,int count,char* filename)
{
        if(Node == NULL)
        {
                return;
        }

        writeOutput(Node->rightNode,count,filename);
        if(terminate == count)
        {
                return;
        }
        FILE* fp;
        fp = fopen(filename,"a");
        fprintf(fp,"%d\n",Node->data);
        printf("%d ",Node->data);
        fclose(fp);
        terminate++;
        writeOutput(Node->leftNode,count,filename);


}

void freeTree(struct node* Node)
{
        if(Node != NULL)
        {
                freeTree(Node->leftNode);
                freeTree(Node->rightNode);
                free(Node);
        }
}
```

PART – B

PartB kısmı tamamen çözülemedi.

- Argümanlar terminalden input olarak alındı.
- Child processler verilen input dosyalarından sayıları başarılı bir şekilde okudu.
- Child process ile parentprocess arasında ‘k’ boyutunda bir sharedmemory kullanıldı
- Bu alana veri girişi ve çıkışı sağlandı.
- Alan dolu olduğunda yapılan gerekli karşılaştırma işlemi ile dinamik olarak alanın içeriğinin değişmesi işlemi programa eklendiğinde çıkan sorunlar yüzünden tamamlanamadı (semaforlu - semaforsuz , fonksiyonel olarak alanda karşılaştırma – fonksiyonel olmadan karşılaştırma gibi birçok yol denendi).

Memory Dolu Olduğunda En Küçük Sayının Bulunması ve Gerekli ise Değiştirilmesi

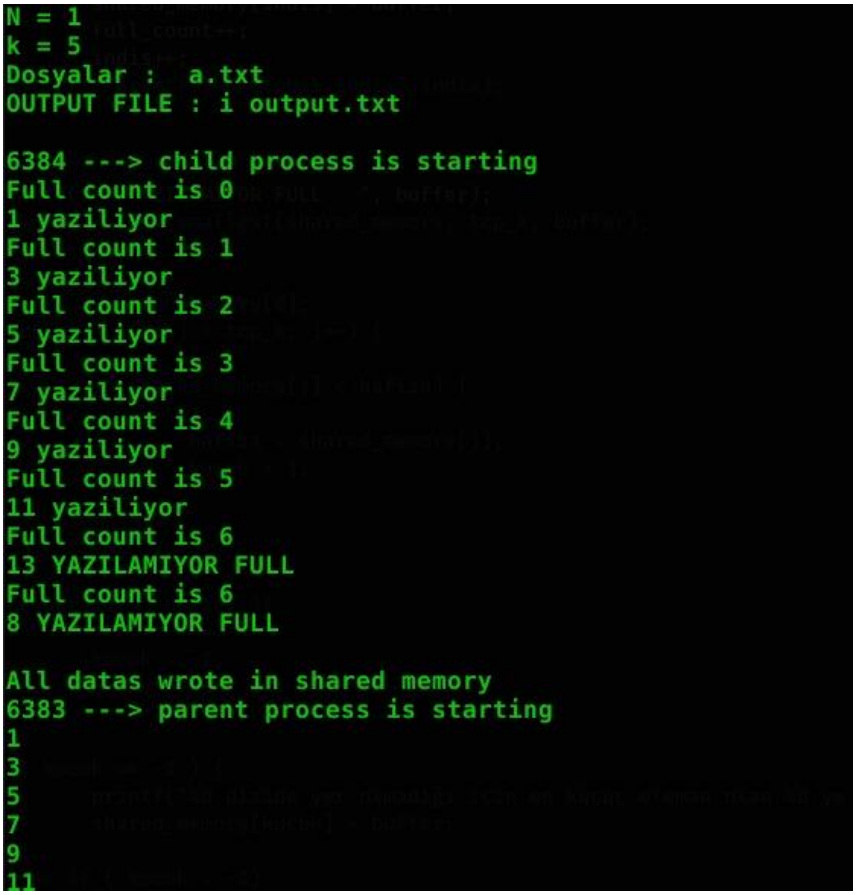
```
void find_smallest(int *shared_memory, int top_k, int buffer) {

	printf("find_smallest içerisinde");
	int mem;
	int kucuk;
	mem = shared_memory[0];
	for(int j = 1; j < top_k; j++) {

		if(shared_memory[j] < mem) {

			mem = shared_memory[j];
			kucuk = j;

		}

	}
	if( mem < buffer ){

		printf("%d dizide yer olmadığı için en küçük eleman olan %d ye yazılıyor\n",buffer,shared_memory[kucuk]);
		shared_memory[kucuk] = buffer;


	}else
		printf("%d dizideki en kücük elemandan da kucuk\n",buffer);

}
```

```
//**** FOR SHARED MEMORY *****

int segment_id;
int *shared_memory;

shared_memory = malloc(top_k * sizeof(int));

segment_id = shmget(IPC_PRIVATE, top_k * sizeof(int),S_IRUSR | S_IWUSR);
shared_memory = (int*) shmat(segment_id, NULL, 0);

//**** FOR SHARED MEMORY *****
```

Dosyadan Veri Okunması ve Belleğe Yazılması

```
while(fscanf(fp,"%d\n",&buffer) ≠ EOF) {

	//sem_wait(&sem2);
	printf("Full count is %d \n",full_count);

	if( full_count ⩽ top_k) {
			printf("%d yaziliyor \n", buffer);
			shared_memory[indis] = buffer;
			full_count++;
			indis++;
			//printf("%d global indis",indis);

	}else {

		printf("%d YAZILAMIYOR MEMORY FULL \n", buffer);
		//find_smallest(shared_memory, top_k, buffer);
```

Program Çıktısı

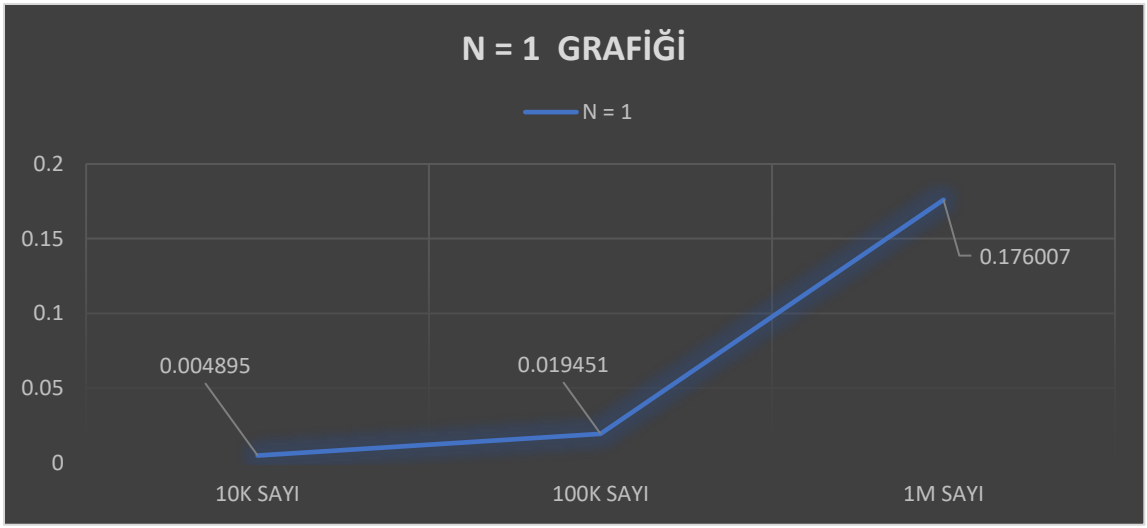


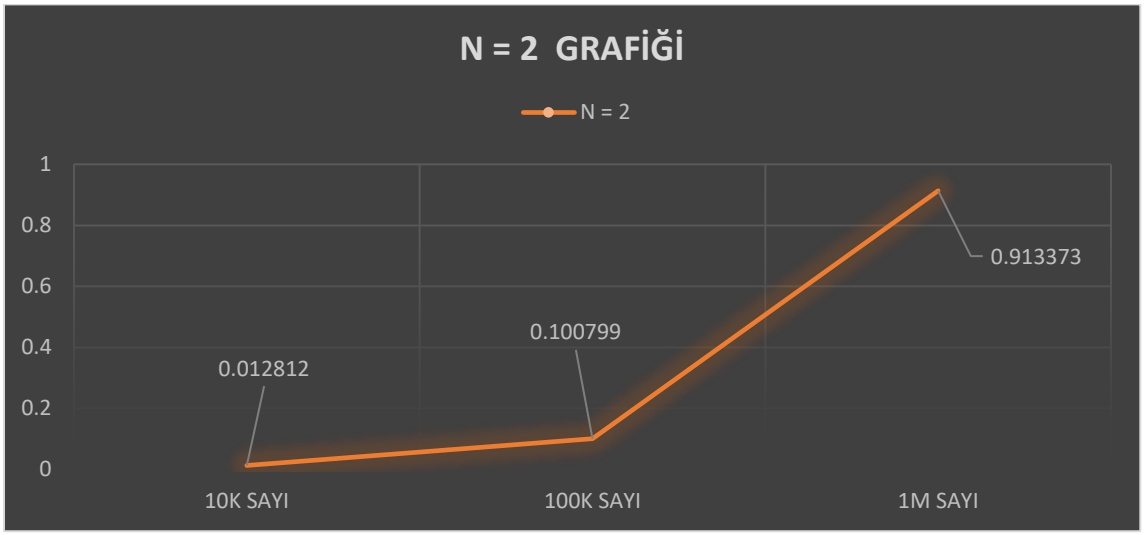
PART – C

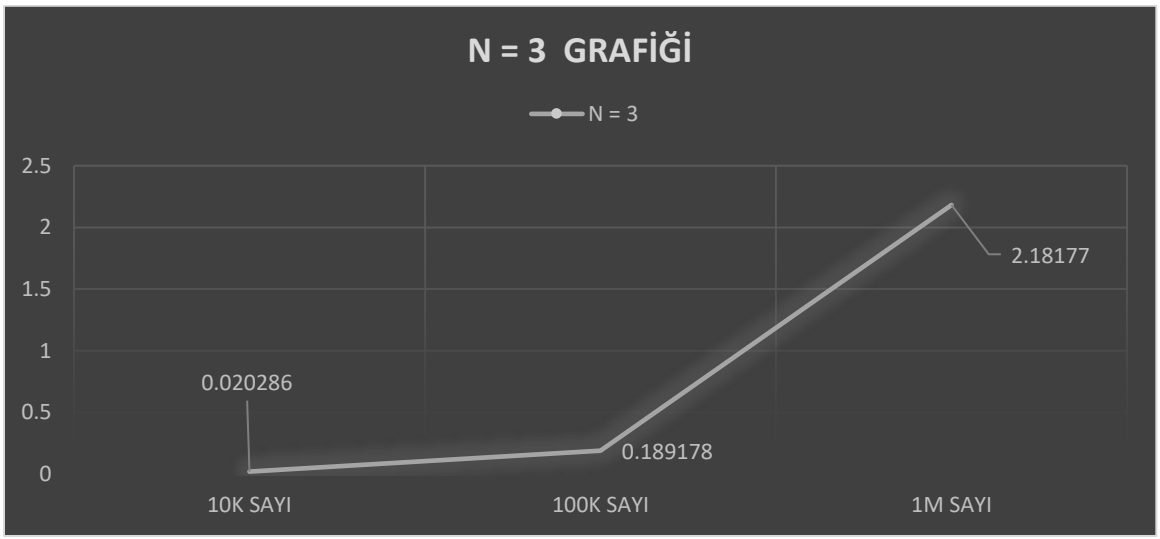
Threadler ve processler senkronize edilmez ise bir eş zamanlı çalışan threadler ya da processler aynı anda aynı yere erişmeye çalışırsa burada tutarsız durumlar ortaya çıkar.

Örneğin sharedmemory alanına bir değer eklenecek 1. Process X indisinde kararlıdır ve buraya yazmak için bekliyor. Aynı anda 2. Process de buraya 1. Processin değişiklik yapılmadan kontrol eder ve değişiklik yapıldıktan sonra kendi işlemini tamamlarsa burada elde edilen sonuçlar anlamsız olabilir.

N DEĞERİ SABİT GRAFİKLER (PART A)







K DEĞERİ SABİT GRAFİKLER (PART A)