MARMARA ÜNİVERSİTESİ

TEKNOLOJİ FAKÜLTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

BLM3003 İŞLETİM SİSTEMLERİ

FİNAL PROJESİ RAPORU

ÖĞRENCİ NO: 171421012

AD SOYAD: Mertcan GELBAL

ÖĞRENCİ NO: 170421041

AD SOYAD: Ahmet Yasir KULAKSIZ

Bu rapor, N adet child process kullanarak belirli dosyalardaki k. en büyük sayıları bulma işlemini ele almakta ve bu süreci optimize etmek için message queue'ler ve thread'lerin nasıl kullanılabileceğini incelemektedir. Farklı N değerleri, dosya boyutları ve k değerleri üzerinde yapılan deneylerim, farklı yöntemlerin performansını değerlendirmek ve bu işlem için en etkili stratejiyi belirlemek amacıyla gerçekleştirilmiştir. Çalışmanın amacı, çoklu süreçlerin, mesaj kuyruklarının ve thread'lerin kullanımının, k. en büyük sayıları bulma görevindeki performans artışına nasıl katkıda bulunduğunu anlamak ve bu yöntemlerin avantajlarını araştırmaktır.

PART 1

Bu programın amacı, belirtilen sayıda child process oluşturarak N adet dosyadaki en büyük k. sayıları bulmak ve bunları sıralayarak bir çıkış dosyasına yazmaktır. İşlem adımları şu şekilde özetlenebilir:

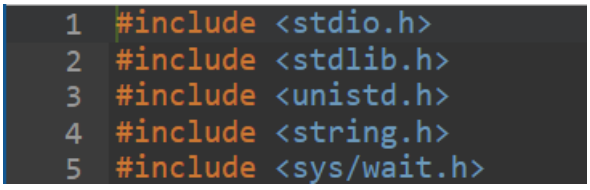
1. Komut Satırı Parametrelerini Alma:
- Program, komut satırından alınan parametreleri kullanarak çalışmaktadır. Bu parametreler arasında k (1 ile 1000 arasında), N (1 ile 5 arasında), giriş dosyaları ve çıkış dosyası bulunmalıdır.

2. Child Process Oluşturma:
- Program, belirtilen sayıda (N) child process oluşturmaktadır.

3. Her Child Process'in Görevi:
- Her child process, kendisine atanmış olan giriş dosyasını okuyarak en büyük k. sayıyı belirlemektedir. Bu sayıyı bulduktan sonra, bu değeri bir ara dosyaya yazmaktadır.

4. Ana Process'in Görevi:
- Ana process, tüm child processlerin bitmesini beklemektedir.
- Tüm ara dosyaları okuyarak içerdikleri en büyük k. sayıları birleştirmekte ve sıralamaktadır.
- Sonuçları çıkış dosyasına yazmaktadır.

5. Ara Dosyaları Silme:
- Tüm işlemler tamamlandığında, program ara dosyaları silmektedir.



Şekil 1

Şekil 1'de gerekli olan kütüphaneler eklenmektedir.

1. selectionsort

```c
int selectionSort(int arr[], int size,int k) {  // diziyi sıralayıp en büyük k. elemenı döndürmek için
    int i, j, minIndex;
    for (i = 0; i < size-1; i++) {
        minIndex = i;
        for (j = i+1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap işlemi
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
    return arr[size-k];
}
```

Şekil 2

Şekil 2'deki fonksiyon selection sort algoritmasıdır. Fakat bir dönüş değeri olarak bize dizideki en büyük k. elemanı döndürmektedir.

2. process_function

```c
void process_function(int process_id,int k, char *outfilename, char *readfilename) {
  int size=0;
  int value;

  FILE *fp = fopen(readfilename, "r");  // okunacak dosya
  if (!fp) {
    perror("Dosya açılamadı");
    exit(1);
  }
  while (fscanf(fp, "%d", &value) == 1) {   // Okunacak dosyanın boyutunu bulmak için
    size++;
  }
  fseek(fp,0,SEEK_SET); // Dosyayı tekrar okurken en başa gelmek için
  int *arr=(int*)malloc(size*sizeof(int)); // Bulunan boyuta göre bir dizi oluşturmak için
  for(int i=0;i<size;i++ ) {   // Sayıları bulup ve intermediate dosyaya ve diziye kaydetmek için
    fscanf(fp, "%d",&value);
    arr[i]=value;
  }
  int num=selectionSort(arr,size,k); // k. en büyük sayıyı bulup geri döndürmek için

  FILE *fp1 = fopen(outfilename, "w");  // Ara dosya
  if (!fp1) {
    perror("Dosya açılamadı");
    exit(1);
  }
  fprintf(fp1,"%d\n",num); // Bulunan değeri ara dosyaya yazmak için
  fclose(fp1);
  fclose(fp);
}
```

Şekil 3

Şekil 3'teki fonksiyonda okunan dosyadaki satır sayısı bulunarak uygun bir dizi oluşturulmaktadır. Bu dizi şekil 2'deki fonksiyona gönderilerek en büyük k. değer bulunmaktadır. Bulunan değer intermediate dosyaya yazılmaktadır.

3. main

```c
int main(int argc, char *argv[]) {  // argc değeri fonksiyonu çalıştırırken eklediğimiz parametrelerin sayısı argv parametrelerin string hali
  if (argc < 5) {
    printf("Kullanım: findtopk <k> <N> <infile1> ...<infileN> <outfile>\n");
    return 1;
  }
  int k = atoi(argv[1]); // k değerini kontrol etmek için
  if (k < 1 || k > 1000) {
    printf("K değeri 1 ile 1000 arasında olmalıdır.\n");
    return 1;
  }


  int N = atoi(argv[2]);  // N değerini kontrol etmek için
  if (N < 1 || N > 5) {
    printf("N değeri 1 ile 5 arasında olmalıdır.\n");
    return 1;
  }
```

Şekil 4

Şekil 4'teki kod bloğunda parametrelerin kontrolü yapılmaktadır.

```c
  char *intermediate_filenames[N]; // ara dosyaların adını tutmak için
  for (int i = 0; i < N; i++) {   // ara dosyaları oluşturmak için
    intermediate_filenames[i] = malloc(100 * sizeof(char));  // dosya ismi için bellekte yer ayırmak için
    sprintf(intermediate_filenames[i], "intermediate%d", i);  // dosya ismini belirlemek için
    FILE *fp = fopen(intermediate_filenames[i], "w"); // dosya yoksa oluşturması için
    fclose(fp);
  }
```

Şekil 5

Şekil 5'teki kod bloğunda ara dosyaların oluşturulması, bellekte yer ayrılması ve isimlendirme işlemleri yapılmaktadır.

```c
pid_t pids[N];   // Child processleri oluşturmak için
for (int i = 0; i < N; i++) {
  pids[i] = fork();  //parent processden süreci kopyalayıp child process olarak atamak için
  if (pids[i] == 0) { // pids[i]==0 olması bu processin child process olduğunu gösterir 0 dan küçükse hata mesajı 0dan büyükse parent process olduğunu gösterir
    process_function(i, k, intermediate_filenames[i], argv[i+3]); // Child process
    exit(0);
  }
}

for (int i = 0; i < N; i++) {   //Child processlerin bitmesini beklemek için
  wait(NULL);
}
```

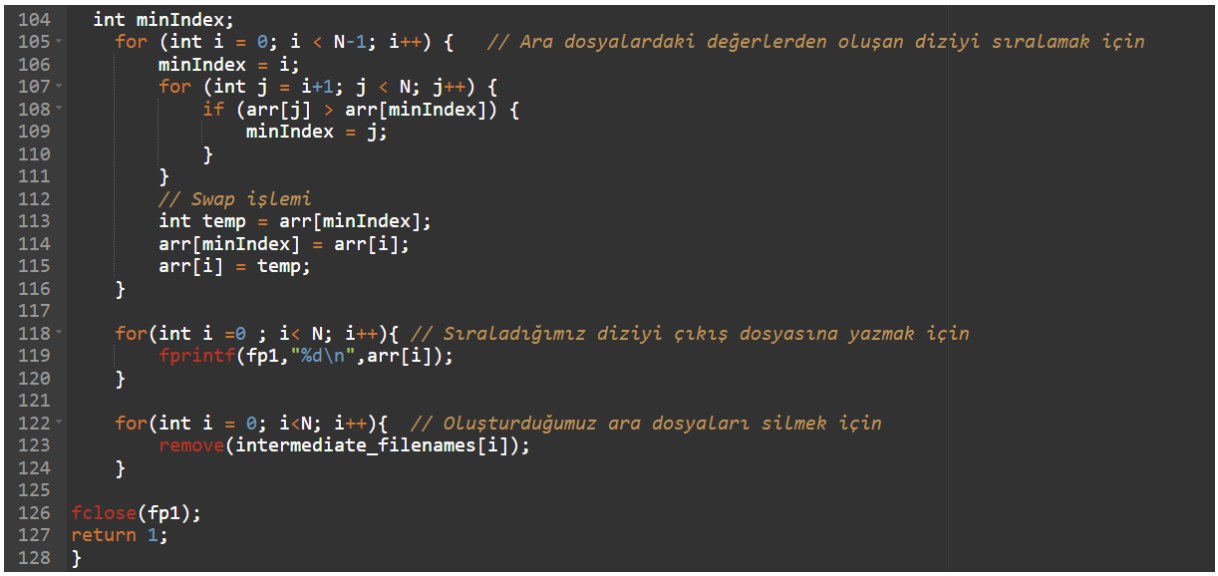
Şekil 6

Şekil 6'daki kod bloğu child process'leri oluşturup parent process'den sürecin kopyalanması sağlanmaktadır. Sonrasında ise child process'lerin işlemleri bitene kadar program beklemektedir.

```c
  int arr[N];
  char *outfile=argv[argc-1];
  FILE *fp1 = fopen(outfile, "w");
  for (int i = 0; i < N; i++) {   //oluşturulan tüm ara dosyaları okuyup K. en büyük sayıyı bulmak için
    FILE *fp = fopen(intermediate_filenames[i], "r");
    int value;
    fscanf(fp, "%d", &value);  // ara dosyalardaki k. en büyük değeri okuyup value olarak kaydetmek için
    arr[i]=value;
    fclose(fp);
  }
```

Şekil 7

Şekil 7'deki kod bloğu oluşturulan ara dosyalardaki k. en büyük değeri arr dizisine kaydetmektedir.



```
104   int minIndex;
105     for (int i = 0; i < N-1; i++) {   // Ara dosyalardaki değerlerden oluşan diziyi sıralamak için
106         minIndex = i;
107         for (int j = i+1; j < N; j++) {
108             if (arr[j] > arr[minIndex]) {
109                 minIndex = j;
110             }
111         }
112         // Swap işlemi
113         int temp = arr[minIndex];
114         arr[minIndex] = arr[i];
115         arr[i] = temp;
116     }
117 
118     for(int i =0 ; i< N; i++){ // Sıraladığımız diziyi çıkış dosyasına yazmak için
119         fprintf(fp1,"%d\n",arr[i]);
120     }
121 
122     for(int i = 0; i<N; i++){  // Oluşturduğumuz ara dosyaları silmek için
123         remove(intermediate_filenames[i]);
124     }
125 
126 fclose(fp1);
127 return 1;
128 }
```

Şekil 8

Şekil 8'deki kod bloğu elde ettiğimiz arr dizisini sıralayıp çıktı dosyasına yazmaktadır. Ayrıca oluşturulan ara dosyaları silme işlemi burada gerçekleşmektedir.

PART 2

Part 2 için işlem adımları şu şekilde özetlenebilir:

1. Komut Satırı Parametrelerini Alma:
- Program, komut satırından alınan parametreleri kullanarak çalışmaktadır. Bu parametreler arasında k (1 ile 1000 arasında), N (1 ile 5 arasında), giriş dosyaları ve çıkış dosyası bulunmaktadır.

2. Message Queue Oluşturma:
- Program, POSIX message queue'ları kullanarak parent ve child process'ler arasında bilgi paylaşımını sağlamaktadır. Message queue'ların oluşturulması için mq_open fonksiyonu kullanılmaktadır.

3. Child Process Oluşturma:
- Program, belirtilen sayıda (N) child process oluşturmaktadır. Her bir child process, kendi message queue'larına erişim sağlamaktadır.

4. Her Child Process'in Görevi:
- Her child process, kendisine atanmış olan giriş dosyasını okuyarak en büyük k. sayıyı belirlemektedir.
- Bu sayıyı parent process'e iletmek için message queue kullanmaktadır.

5. Parent Process'in Görevi:
- Ana process, tüm child process'lerin bitmesini beklemektedir.
- Tüm child process'lerden alınan en büyük k. sayıları birleştirmekte ve sıralamaktadır.
- Sonuçları çıkış dosyasına yazmaktadır.

Message Queue'ların Temizlenmesi:

- Tüm işlemler tamamlandığında, program oluşturulan message queue'ları kapatmalı ve silmektedir.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <mqueue.h>
```

Şekil 9

Şekil 9'da gerekli olan kütüphaneler eklenmektedir.

1. selectionsort

```c
int selectionSort(int arr[], int size,int k) {  // diziyi sıralayıp en büyük k. elemenı döndürmek için
    int i, j, minIndex;
    for (i = 0; i < size-1; i++) {
        minIndex = i;
        for (j = i+1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap işlemi
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
    return arr[size-k];
}
```

Şekil 10

Şekil 10'daki fonksiyon selection sort algoritmasıdır. Fakat bir dönüş değeri olarak bize dizideki en büyük k. elemanı döndürmektedir.

2. process_function

```c
void process_function(int process_id, int k, mqd_t mq, char *readfilename) {
    int size = 0;
    int value;

    FILE *fp = fopen(readfilename, "r"); // okunacak dosya
    if (!fp) {
        perror("Dosya açılamadı");
        exit(1);
    }
    while (fscanf(fp, "%d", &value) == 1) {  // Okunacak dosyanın boyutunu bulmak için
        size++;
    }
    fseek(fp,0,SEEK_SET); // okuma konumunu en başa almak için
    int *arr = (int *)malloc(size * sizeof(int));  // Bulunan boyuta göre bir dizi oluşturmak için
    for (int i = 0; i < size; i++) { // Sayıları bulup diziye kaydetmek için
        fscanf(fp, "%d", &value);
        arr[i] = value;
    }
    int num = selectionSort(arr, size, k);  // k. en büyük sayıyı bulup geri döndürmek için
    char message[20];   // Bulduğumuz değerin string halini saklamak için
    snprintf(message, 20, "%d", num); // int değeri stringe çevirmek için
    if(mq_send(mq, message, sizeof(message), 0)== -1){ // Bulduğumuz değeri parent processe göndermek için
    printf("hata");
    }
    fclose(fp);
}
```

Şekil 11

Şekil 11'deki fonksiyonda okunan dosyadaki satır sayısı bulunarak uygun bir dizi oluşturulmaktadır. Bu dizi şekil 10'daki fonksiyona gönderilerek en büyük k. değer bulunmaktadır. Bulunan değerler message queue ile parent process'e gönderilmektedir.

3. main

```
int main(int argc, char *argv[]) {
    if (argc < 5) {
        printf("Kullanım: findtopk_mqueue <k> <N> <infile1> ...<infileN> <outfile>\n");
        return 1;
    }

    int k = atoi(argv[1]);
    if (k < 1 || k > 1000) {
        printf("K değeri 1 ile 1000 arasında olmalıdır.\n");
        return 1;
    }

    int N = atoi(argv[2]);
    if (N < 1 || N > 5) {
        printf("N değeri 1 ile 5 arasında olmalıdır.\n");
        return 1;
    }
```

Şekil 12

Şekil 12'deki kod bloğunda parametrelerin kontrolü yapılmaktadır.

```
    struct mq_attr attr;       // Kuyruğun yapısal özellikleri
    attr.mq_flags = 0;         // 0 veya O_NONBLOCK
    attr.mq_maxmsg = 5;        // Kuyruktaki maksimum mesaj sayısı
    attr.mq_msgsize = 20;      // Her bir mesajın maksimum boyutu
    attr.mq_curmsgs = 0;       // Başlangıçta kuyrukta bulunan mesaj sayısı
    mqd_t mq = mq_open("/a_mq", O_CREAT | O_RDWR, 0666, &attr); // Kuyruğu oluşturmak için
    if (mq == (mqd_t)-1) {
        perror("Mesaj kuyruğu oluşturma başarısız oldu");
        exit(1);
    }
```

Şekil 13

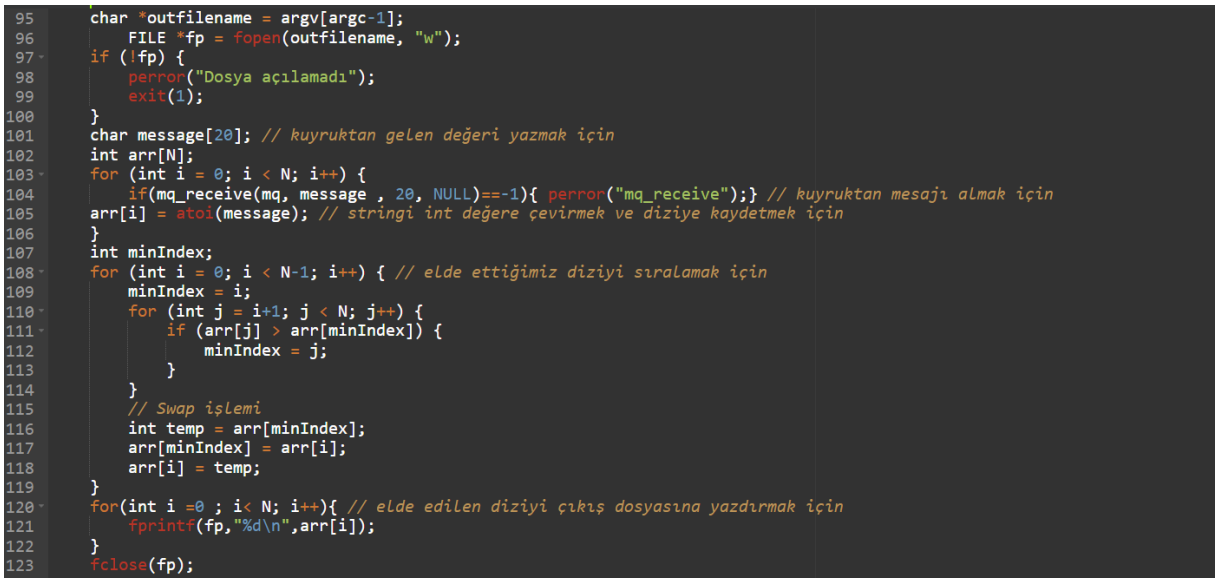
Şekil 13'teki kod bloğunda kuyruğun yapısal özellikleri belirlenmekte ve kuyruk oluşturulmaktadır.

```
pid_t pids[N];
for (int i = 0; i < N; i++) { // Child processleri oluşturmak için
    pids[i] = fork();   //parent processden süreci kopyalayıp child process olarak atamak için
    if (pids[i] == 0) {// pids[i]==0 olması bu processin child process olduğunu gösterir 0dan küçükse hata mesajı 0dan büyükse parent process olduğunu gösterir
        process_function(i, k, mq, argv[i + 3]); // Child process
        exit(0);
    }
}

for (int i = 0; i < N; i++) { //Child processlerin bitmesini beklemek için
    wait(NULL);
}
```

Şekil 14

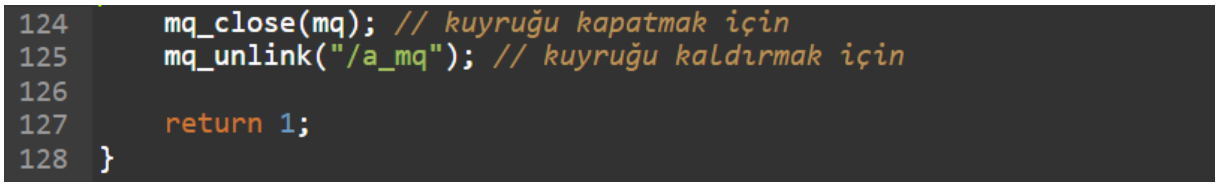
Şekil 14'teki kod bloğu child process'leri oluşturup parent process'den sürecin kopyalanması sağlanmaktadır. Sonrasında ise child process'lerin işlemleri bitene kadar program beklemektedir.



```
    char *outfilename = argv[argc-1];
        FILE *fp = fopen(outfilename, "w");
    if (!fp) {
        perror("Dosya açılamadı");
        exit(1);
    }
    char message[20]; // kuyruktan gelen değeri yazmak için
    int arr[N];
    for (int i = 0; i < N; i++) {
        if(mq_receive(mq, message , 20, NULL)==-1){ perror("mq_receive");} // kuyruktan mesajı almak için
    arr[i] = atoi(message); // stringi int değere çevirmek ve diziye kaydetmek için
    }
    int minIndex;
    for (int i = 0; i < N-1; i++) { // elde ettiğimiz diziyi sıralamak için
        minIndex = i;
        for (int j = i+1; j < N; j++) {
            if (arr[j] > arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap işlemi
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
    for(int i =0 ; i< N; i++){ // elde edilen diziyi çıkış dosyasına yazdırmak için
        fprintf(fp,"%d\n",arr[i]);
    }
    fclose(fp);
```

Şekil 15

Şekil 15'deki kod bloğu message queue' den k. en büyük değeri alıp arr dizisine kaydetmektedir. Daha sonra elde ettiğimiz arr dizisini sıralayıp çıktı dosyasına yazmaktadır.



```
    mq_close(mq); // kuyruğu kapatmak için
    mq_unlink("/a_mq"); // kuyruğu kaldırmak için

    return 1;
}
```

Şekil 16

Şekil 16'daki kod bloğu oluşturulan kuyrukları kapatıp silmektedir.

PART 3

Part 3'te, her bir girdi dosyası için ayrı bir thread oluşturarak paralel işlemler kullanarak en büyük k. sayıları bulma işlemi gerçekleştirmektedir. Global değişkenler veya thread özel veri yapıları aracılığıyla thread'ler arasında bilgi paylaşımı yapmaktadır. İşlem adımları şu şekilde özetlenebilir:

1. Komut Satırı Parametrelerini Alma:

- Program, komut satırından alınan parametreleri kullanarak çalışmalıdır. Bu parametreler arasında k (1 ile 1000 arasında), N (1 ile 5 arasında), giriş dosyaları ve çıkış dosyası bulunmaktadır.

2. Thread Oluşturma:

- Program, belirtilen sayıda (N) POSIX thread oluşturmalıdır. Her bir thread, kendi görevini yerine getirmek üzere tasarlanmıştır.

3. Her Thread'in Görevi:

- Her bir thread, kendisine atanmış olan giriş dosyasını okuyarak en büyük k. sayıyı belirlemektedir.
- En büyük k. sayıyı bir thread global değişkenine (veya bir thread özel veri yapısına) kaydetmektedir.

4. Thread Sonuçlarını Birleştirme:
- Thread'lerin çalışması bittiğinde, ana thread bu thread'lerden alınan en büyük k. sayıları birleştirmekte ve sıralamaktadır.

5. Sonuçları Çıkış Dosyasına Yazma:
- Elde edilen sıralı en büyük k. sayıları çıkış dosyasına yazmaktadır.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
```

Şekil 17

Şekil 17'de gerekli olan kütüphaneler eklenmektedir.

```c
// Global değişkenler
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int k;
int result[5];
int result_index = 0;

// Fonksiyon imzaları
int selectionSort(int arr[], int size, int k);
void *process_function(void *arg);
```

Şekil 18

Şekil 18'de global değişkenler ve fonksiyon imzaları eklenmiştir.

1. selectionsort

```c
int selectionSort(int arr[], int size,int k) {  // diziyi sıralayıp en büyük k. elemenı döndürmek için
    int i, j, minIndex;
    for (i = 0; i < size-1; i++) {
        minIndex = i;
        for (j = i+1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap işlemi
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
    return arr[size-k];
}
```

Şekil 20

Şekil 20'deki fonksiyon selection sort algoritmasıdır. Fakat bir dönüş değeri olarak bize dizideki en büyük k. elemanı döndürmektedir.

2. process_function

```c
void *process_function(void *arg) {
    char *readfilename = (char *)arg;

    int size = 0;
    int value;

    FILE *fp = fopen(readfilename, "r"); // okunacak dosya
    if (!fp) {
        perror("Dosya açılamadı");
        exit(1);
    }

    while (fscanf(fp, "%d", &value) == 1) {  // Okunacak dosyanın boyutunu bulmak için
        size++;
    }
    fseek(fp,0,SEEK_SET);  // okuma konumunu en başa almak için
    int *arr = (int *)malloc(size * sizeof(int));  // Bulunan boyuta göre bir dizi oluşturmak için
    for (int i = 0; i < size; i++) {  // Sayıları bulup diziye kaydetmek için
        fscanf(fp, "%d", &value);
        arr[i] = value;
    }

    fclose(fp);

    pthread_mutex_lock(&mutex);    // Kritik bölgeye giriş

    int num = selectionSort(arr, size, k);
    result[result_index++] = num;

    pthread_mutex_unlock(&mutex);    // Kritik bölgeden çıkış

    pthread_exit(NULL); // Thread'in işi bittiğinde sonlanması
}
```

Şekil 21

Şekil 21'deki fonksiyon, thread'lerin her birinde çalışan fonksiyondur. Fonksiyon, öncelikle okunacak dosyayı açar ve dosya boyutunu bulmaktadır. Ardından, bulunan boyuta göre bir dizi oluşturur ve dosyayı okuyarak diziyi doldurmaktadır. Son olarak, dizideki k. en büyük sayıyı bulur ve bunu global bir değişkene kaydetmektedir.

3. main

```c
int main(int argc, char *argv[]) {  // argc değeri fonksiyonu çalıştırırken eklediğimiz parametrelerin sayısı argv parametrelerin string hali
    if (argc < 5) {
        printf("Kullanım: findtopk_thread <k> <N> <infile1> ...<infileN> <outfile>\n");
        return 1;
    }

    k = atoi(argv[1]);  // k değerini kontrol etmek için
    if (k < 1 || k > 1000) {
        printf("K değeri 1 ile 1000 arasında olmalıdır.\n");
        return 1;
    }

    int N = atoi(argv[2]);  // N değerini kontrol etmek için
    if (N < 1 || N > 5) {
        printf("N değeri 1 ile 5 arasında olmalıdır.\n");
        return 1;
    }
```

Şekil 22

Şekil 22'deki kod bloğunda parametrelerin kontrolü yapılmaktadır.

```
    pthread_t threads[N];    // Thread dizisi

    // Giriş dosyaları için thread'leri oluşturmak için
    for (int i = 0; i < N; i++) {
        if (pthread_create(&threads[i], NULL, process_function, (void *)argv[i + 3]) != 0) {
            perror("Thread oluşturma hatası");
            exit(1);
        }
    }

    // Thread'leri bekleyerek programın bitmesini sağlamak için
    for (int i = 0; i < N; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("Thread beklenemedi");
            exit(1);
        }
    }
```

Şekil 23

Şekil 23'deki kod bloğu her bir dosya için thread oluşturarak process'i thread'lere göndermektedir. Sonrasında ise thread'leri bekleyerek programın bitmesini sağlamaktadır.

```
    // Sonuçları dosyaya yazmak için
    FILE *fp1 = fopen(argv[argc - 1], "w");
    if (!fp1) {
        perror("Dosya açılamadı");
        exit(1);
    }
    int minIndex;    // elde ettiğimiz diziyi sıralamak için
    for (int i = 0; i < N-1; i++) {
        minIndex = i;
        for (int j = i+1; j < N; j++) {
            if (result[j] > result[minIndex]) {
                minIndex = j;
            }
        }
        // Swap işlemi
        int temp = result[minIndex];
        result[minIndex] = result[i];
        result[i] = temp;
    }
    for (int i = 0; i < result_index; i++) { // diziyi çıkış dosyasına yazdırmak için
        fprintf(fp1, "%d\n", result[i]);
    }

    fclose(fp1);

    return 0;
}
```

Şekil 24

Şekil 24'teki kod bloğu elde edilen result dizisi sıralanarak çıkış dosyasına yazdırma işlemi yapılmaktadır.

PART 4

Part 4'te, findtopk, findtopk_mqueue ve findtopk_thread programlarının performansını, çeşitli k, N ve dosya boyutları için incelenmektedir.

Tablo 1: Programların N Değerine Göre Karşılaştırma Tablosu

Experiment	Algorithm	Time (seconds)	User (seconds)	Kernel (seconds)
N=1	findtopk	19.980	19.889	0.008
N=1	findtopk_mqueue	15.093	14.958	0.014
N=1	findtopk_thread	10.687	10.669	0.008
N=5	findtopk	57.732	153.405	0.084
N=5	findtopk_mqueue	56.898	153.036	0.059
N=5	findtopk_thread	137.373	136.579	0.136

Tablo 1'de dosya boyutu ve k değeri sabit tutularak N=1 ve N=5 değerleri için karşılaştırma yapılmaktadır. (k=500, dosya ise 100.000 sayıdan oluşmaktadır.)

Tablo 2: Programların Dosyanın Boyut Değerine Göre Karşılaştırma Tablosu

Experiment	Algorithm	Time (seconds)	User (seconds)	Kernel (seconds)
1k	findtopk	0.006	0.004	0.000
1k	findtopk_mqueue	0.018	0.006	0.001
1k	findtopk_thread	0.006	0.005	0.000
1m	findtopk	3415.471	3411.183	0.488
1m	findtopk_mqueue	3404.702	3359.097	0.440
1m	findtopk_thread	2850.652	2849.277	0.192

Tablo 2'de k ve N değerleri sabit tutularak dosya boyutu kıyaslanmaktadır. Dosya boyutları 1000 sayıdan ve 1000000 sayıdan oluşmaktadır. (k=500, N=1)

Tablo 3: Programların k Değerine Göre Karşılaştırma Tablosu

Experiment	Algorithm	Time (seconds)	User (seconds)	Kernel (seconds)
1	findtopk	0.20228	0.20126	0.00260
1	findtopk_mqueue	0.20082	0.20038	0.00080
1	findtopk_thread	0.13938	0.13913	0.00000
500	findtopk	19.980	19.889	0.008
500	findtopk_mqueue	15.093	14.958	0.014
500	findtopk_thread	10.687	10.669	0.008

Tablo 3'te N değeri ve dosya boyutu sabit tutularak k=1 ve k=500 değerleri için karşılaştırma yapılmaktadır. (N=1, dosya ise 100.000 sayıdan oluşmaktadır.)

Yapılan deneyler sonucunda, findtopk_thread programının, findtopk ve findtopk_mqueue programlarına göre daha hızlı olduğu görülmüştür. Bu durum, findtopk_thread programının, findtopk programının kullandığı intermediate dosyaları ve findtopk_mqueue programının kullandığı message queue'leri kullanmamasından kaynaklanmaktadır.

findtopk_thread programının çalışma süresi, N değerine bağlı olarak değişmektedir. N değeri arttıkça, findtopk_thread programının çalışma süresi de artmaktadır. Bu durum, findtopk_thread programının, N adet thread oluşturması ve bu thread'leri koordine etmesi ile ilgilidir.

findtopk_mqueue programının çalışma süresi, N değerine bağlı olarak daha az değişmektedir. Bu durum, findtopk_mqueue programının, N adet thread oluşturmasına gerek duymamasından kaynaklanmaktadır.

findtopk programının çalışma süresi, N değerine bağlı olarak önemli ölçüde artmaktadır. Bu durum, findtopk programının, N adet intermediate dosyayı okuması ve işlemesi ile ilgilidir.

K değerinin çalışma süresine etkisi, N değerine göre daha azdır. K değeri arttıkça, findtopk ve findtopk_mqueue programlarının çalışma süresi artmaktadır.

Sayıların boyutunun çalışma süresine etkisi, N ve K değerlerine göre daha fazladır. Sayıların boyutu arttıkça, findtopk, findtopk_thread ve findtopk_mqueue programlarının çalışma süresini arttırmaktadır. Bu durum, findtopk, findtopk_thread ve findtopk_mqueue programlarının, daha fazla sayıyı okumaları ve işlemeleri ile ilgilidir.

Bu sonuçlara göre, findtopk_thread programı, findtopk ve findtopk_mqueue programlarına göre daha hızlıdır. Bu durum, findtopk_thread programının, intermediate dosyaları ve message queue'leri kullanmamasından kaynaklanmaktadır.