//chibi-dris

Banker

```
#include<iostream>
using namespace std;

int main(){
    int res[3]={8,6,8};
    int allocation[5][3]={{0,0,1},{3,0,0},{1,0,0},{2,3,2},{0,0,3}};
    int maxneed[5][3]={{7,6,3},{3,2,2},{8,0,2},{2,1,2},{5,2,3}};
    int avail[3];
    int rem_need[5][3];
    int sum1;
    for(int i=0;i<3;i++){
      sum1=0;
      for(int j=0;j<5;j++){
      sum1=sum1+allocation[j][i];
      }
      avail[i]=res[i]-sum1;
    }
    for(int i=0;i<3;i++){
     cout<<avail[i]<<"\t";
    }

    cout<<endl;

    for(int i=0;i<5;i++){
      for(int j=0;j<3;j++){
      rem_need[i][j]=maxneed[i][j]-allocation[i][j];
      if(rem_need[i][j]<0){
         rem_need[i][j]=0;
      }
      }
    }
    for(int i=0;i<5;i++){
      for(int j=0;j<3;j++){
         cout<<rem_need[i][j]<<"\t";
      }
      cout<<endl;
    }
    int f[5],ans[5],ind=0;

    for(int k=0;k<5;k++){
        f[k]=0;
    }

    for(int k=0;k<5;k++){
        for(int i=0;i<5;i++){
            if(f[i]==0){
                int flag=0;
                for(int j=0;j<3;j++){
                    if(rem_need[i][j]>avail[j]){
                        flag=1;
                        break;
                    }
                }
                if(flag==0){
                    ans[ind++]=i;
                    for(int y=0;y<3;y++){
                        avail[y]+=allocation[i][y];
                    }
                    f[i]=1;
                }
            }
        }
    }

    for(int i=0;i<5;i++){
        cout<<"p"<<ans[i]<<"->";
    }
    return 0;
}
```

Security and resource request code:

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

int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n, r;

void input();

void show();
void cal();

void request();

int main() {
    int i, j;
    printf("********** Banker's Algo ************\n");
    input();
    cal();
    show();
    request();
    //getch();
    return 0;
}

void input() {
    int i, j;
    printf("Enter the no of Processes\t");
    scanf("%d", &n);
    printf("Enter the no of resources instances\t");
    scanf("%d", &r);
    printf("Enter the Max Matrix\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < r; j++) {
            scanf("%d", &max[i][j]);
        }
    }
    printf("Enter the Allocation Matrix\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < r; j++) {
            scanf("%d", &alloc[i][j]);
        }

    }
    printf("Enter the available Resources\n");
    for (j = 0; j < r; j++) {
        scanf("%d", &avail[j]);
    }
}

void show() {
    int i, j;
    printf("Process\t Allocation\t Need\t Max\t Available");
    for (i = 0; i < n; i++) {
        printf("\nP%d\t ", i + 1);
        for (j = 0; j < r; j++) {
            printf("%d ", alloc[i][j]);
        }
        printf("\t");
        for (j = 0; j < r; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\t");
        for (j = 0; j < r; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\t");
        if (i == 0) {
            for (j = 0; j < r; j++)
                printf("%d ", avail[j]);
        }
    }
}

void cal() {
    int finish[100], flag = 1, k, c1 = 0;
    int i, j;
    for (i = 0; i < n; i++) {
        finish[i] = 0;
    }

    //find need matrix
    for (i = 0; i < n; i++) {
        for (j = 0; j < r; j++) {
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }
    printf("\n");
    while (flag) {
        flag = 0;
        for (i = 0; i < n; i++) {
            int c = 0;
            for (j = 0; j < r; j++) {
                if ((finish[i] == 0) && (need[i][j] <= avail[j])) {
                    c++;
                    if (c == r) {
                        for (k = 0; k < r; k++) {
                            avail[k] += alloc[i][j];
                            finish[i] = 1;
                            flag = 1;
                        }
                        printf("P%d->", i);
                        if (finish[i] == 1) {
                            i = n;
                        }
                    }
                }
            }
        }
    }
    for (i = 0; i < n; i++) {
        if (finish[i] == 1) {
            c1++;
        } else {
            printf("P%d->", i);
        }

    }
    if (c1 == n) {
        printf("\n The system is in safe state\n");
    } else {
        printf("\n Process are in dead lock\n");
        printf("\n System is in unsafe state\n");
    }
}

void request() {
    int c, pid, request[100][100], i;
    printf("\n Do you want make an additional request for any of the process ? (1=Yes|0=No)");
    scanf("%d", &c);
    if (c == 1) {
        printf("\n Enter process number : ");
        scanf("%d", &pid);
        printf("\n Enter additional request : \n");
        for (i = 0; i < r; i++) {
            printf(" Request for resource %d : ", i + 1);
            scanf("%d", &request[0][i]);
        }
        for (i = 0; i < r; i++) {
            if (request[0][i] > need[pid][i]) {
                printf("\n ******Error encountered******\n");
                exit(0);
            }
        }
        for (i = 0; i < r; i++) {
            avail[i] -= request[0][i];
            alloc[pid][i] += request[0][i];
            need[pid][i] -= request[0][i];
        }
        cal();
        //getch();
    } else {
        exit(0);
    }
}
```

Round Robin

```
#include<iostream>
using namespace std;

int main(){
     struct process{
        int id;
        int at;
        int bt;
        int tat;
        int wt;
        int rem_time;
        int ct;
     };
     int n,qt;
     cout<<"enter the number of process"<<endl;
     cin>>n;
     process p[n];

     for(int i=0;i<n;i++){
        cout<<"enter arrival time of process "<<i+1<<endl;
        cin>>p[i].at;
        cout<<"enter burst time of process "<<i+1<<endl;
        cin>>p[i].bt;
        p[i].rem_time=p[i].bt;
        p[i].id=i+1;
     }
     cout<<"enter time quantum"<<endl;
     cin>>qt;

    int currtime=0;
    bool alldone=false;

     while(!alldone){
        alldone=true;
        for(int i=0;i<n;i++){
            if(p[i].rem_time>0){
               alldone=false;
            if(p[i].rem_time>qt){
                p[i].rem_time=p[i].rem_time-qt;
                currtime=currtime+qt;
            }
            else{
                currtime=currtime+p[i].rem_time;
                p[i].ct=currtime;
                p[i].rem_time=0;
            }

        }

     }
     }
      for(int i=0;i<n;i++){
        p[i].tat=p[i].ct-p[i].at;
        p[i].wt=p[i].tat-p[i].bt;
      }

     cout<<"at \t"<<"bt \t"<<"ct \t"<<"tat \t"<<"wt \t"<<endl;
     for(int i=0;i<n;i++){
        cout<<p[i].at<<"\t"<<p[i].bt<<"\t"<<p[i].ct<<"\t"<<p[i].tat<<"\t"<<p[i].wt<<"\t"<<endl;
     }
     return 0;
}
```

Write a program using C/C++/Java to simulate the first fit, best fit and worst fit memory allocation strategy. Assume memory chunk and initial requirement for memory block from your side.

First fit

```
#include <iostream>
using namespace std;
int main()
{
    int n = 4; // Number of processes
    int m = 5; // Number of memory Blocks
    int Process_size[] = {212, 417, 112, 426};
    int Memory_block_size[] = {100, 500, 200, 300, 600};
    int allocation[n]; // To store the alloted memory block for a particular process

    for (int i = 0; i < n; i++)
    {
        // flag is used to keep a check if non of the available resource can accomodate the process
        int flag = 0;
        for (int j = 0; j < 5; j++)
        {
            if (Process_size[i] <= Memory_block_size[j])
            {
                allocation[i] = j + 1; //+1 for 1 as j stores only the index

                Memory_block_size[j] = Memory_block_size[j] - Process_size[i];

                flag = 1;
                break;
            }
        }
        // flag=0 implies no memory block can accomodate the process
        if (flag == 0)
        {
            allocation[i] = -1;
        }
    }

    // Printing the final result
    cout << "Process\t"
         << "Process Size\t"
         << "Block No." << endl;
    for (int i = 0; i < n; i++)
    {
        cout << (i + 1) << "\t" << Process_size[i] << "\t";

        // Check for the not allotted case
        if (allocation[i] == -1)
            cout << "Not Allotted" << endl;
        else
            cout << allocation[i] << endl;
    }

    return 0;
}
```

Best fit

```
#include <iostream>
using namespace std;
int main()
{
    int n = 4; // No of processes
    int m = 5; // No of memory blocks

    int Process_size[] = {212, 417, 112, 426};
    int Memory_block_size[] = {100, 500, 200, 300, 600};

    int allocation[n]; // To store the alloted memory block for a particular process

    for (int i = 0; i < n; i++)
    {
        // Calculating the minimum available memory block that can accomodate the process
        int min = 1000000;
        int min_index;
        for (int j = 0; j < m; j++)
        {
            if (Process_size[i] < Memory_block_size[j] && Memory_block_size[j] < min)
            {
                min = Memory_block_size[j];
                min_index = j;
            }
        }
        // min value is unchanged implies that the process cannot be accomodated in any available
memory block
        if (min == 1000000)
            allocation[i] = -1;
        else
        {
            Memory_block_size[min_index] = Memory_block_size[min_index] - Process_size[i];
            allocation[i] = min_index + 1; //+1 to convert index to process number
        }
    }

    // Printing the final result
    cout << "Process\t"
         << "Process Size\t"
         << "Block No." << endl;
    for (int i = 0; i < n; i++)
    {
        cout << (i + 1) << "\t" << Process_size[i] << "\t";

        // Check for the not allotted case
        if (allocation[i] == -1)
            cout << "Not Allotted" << endl;
        else
            cout << allocation[i] << endl;
    }
    return 0;
}
```

Worst fit

```
#include <iostream>
using namespace std;
int main()
{
    int n = 4; // No of processes
    int m = 5; // No of memory blocks

    int Process_size[] = {212, 417, 112, 426};
    int Memory_block_size[] = {100, 500, 200, 300, 600};

    int allocation[n]; // To store the alloted memory block for a particular process

    for (int i = 0; i < n; i++)
    {
        // Calculating the maximum available memory block that can accomodate the process
        int max = -1;
        int max_index;
        for (int j = 0; j < m; j++)
        {
            if (Process_size[i] < Memory_block_size[j] && Memory_block_size[j] > max)
            {
                max = Memory_block_size[j];
                max_index = j;
            }
        }
        // max value is unchanged implies that the process cannot be accomodated in any
available memory block
        if (max == -1)
            allocation[i] = -1;
        else
        {
            Memory_block_size[max_index] = Memory_block_size[max_index] - Process_size[i];
            allocation[i] = max_index + 1; //+1 to convert index to process number
        }
    }

    // Printing the final result
    cout << "Process\t"
         << "Process Size\t"
         << "Block No." << endl;
    for (int i = 0; i < n; i++)
    {
        cout << (i + 1) << "\t" << Process_size[i] << "\t";

        // Check for the not allotted case
        if (allocation[i] == -1)
            cout << "Not Allotted" << endl;
        else
            cout << allocation[i] << endl;
    }
    return 0;
}
```

SJF

```
#include <iostream>
using namespace std;

struct process
{
    int id;
    int burstTime;
    int arrivalTime;
    int completionTime;
    int waitingTime;
    int turnAroundTime;
};

int main()
{
    int num, smallest;
    cout << "Enter number of Processes" << endl;
    cin >> num;
    process p[num];
    int time = 0;
    int count = 0;
    for (int i = 0; i < num; i++)
    {
        cout << "Enter burst time for process " << i + 1 << endl;
        cin >> p[i].burstTime;
        cout << "Enter arrival time for process " << i + 1 << endl;
        cin >> p[i].arrivalTime;
        p[i].id = i + 1;
    }
    for (time = 0; count !=num;time++){
        smallest = 9;
        for(int i=0; i < num; i++){
            if(p[i].arrivalTime<=time && p[i].burstTime<p[smallest].burstTime && p[i].burstTime>0 )
                smallest=i;
        }
        p[smallest].burstTime--;

        if (p[smallest].burstTime == 0)
        {
            count++;
            p[smallest].completionTime = time + 1;
            p[smallest].turnAroundTime = p[smallest].completionTime - p[smallest].arrivalTime;
            p[smallest].waitingTime = p[smallest].turnAroundTime - p[smallest].burstTime;
            cout << "Process " << p[smallest].id << " completed at time " <<
p[smallest].completionTime << endl
                 << "Waiting time: " << p[smallest].waitingTime << endl
                 << "Turn around time: " << p[smallest].turnAroundTime << endl;
        }
    }
    return 0;
}
```

SRTF (SJF with Pre-emption)

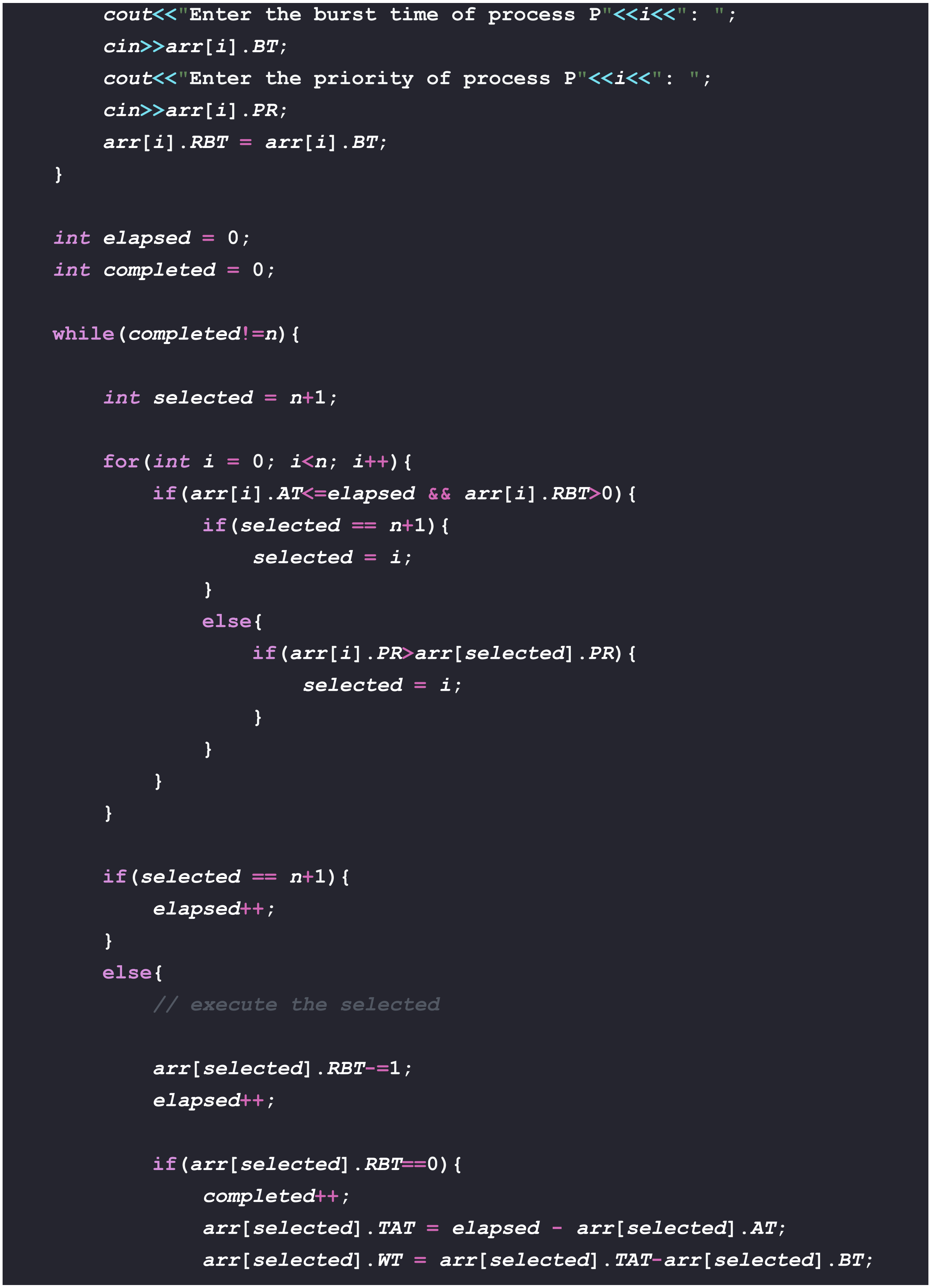
```
#include <iostream>
using namespace std;

int main() {
    int at[10], bt[10], rt[10], completionTime, i, smallest = 0;
    int remain = 0, n, t, sum_wait = 0, sum_turnaround = 0;

    cout << "Enter no of Processes: ";
    cin >> n;

    for(i = 0; i < n; i++) {
        cout << "Enter arrival time for Process P" << i + 1 << ": ";
        cin >> at[i];
        cout << "Enter burst time for Process P" << i + 1 << ": ";
        cin >> bt[i];
        rt[i] = bt[i];
    }

    cout << "\n\nProcess\t| Turnaround Time | Waiting Time\n\n";

    for(t = 0; remain != n; t++) {
        smallest = -1;
        for(i = 0; i < n; i++) {
            if(at[i] <= t && rt[i] && (smallest == -1 || rt[i] <
rt[smallest])) {
                smallest = i;
            }
        }
        if(smallest == -1) {
            continue;
        }
        rt[smallest]--;
        if(rt[smallest] == 0) {
            remain++;
            completionTime = t + 1;
            cout << "P[" << smallest + 1 << "]\t| " << completionTime -
at[smallest] << "\t\t | " << completionTime - bt[smallest] - at[smallest]
<< "\n";
            sum_wait += completionTime - bt[smallest] - at[smallest];
            sum_turnaround += completionTime - at[smallest];
        }
    }

    cout << "\n\nAverage waiting time = " << static_cast<double>(sum_wait)
/ n << endl;
    return 0;
}
```

FCFS

```
#include <iostream>
using namespace std;

struct process
{
    int id;
    int burstTime;
    int arrivalTime;
    int completionTime;
    int waitingTime;
    int turnAroundTime;
};

int main()
{
    int num;
    cout << "Enter number of Processes" << endl;
    cin >> num;

    process p[num];
    for (int i = 0; i < num; i++)
    {
        cout << "Enter burst time for process " << i + 1 << endl;
        cin >> p[i].burstTime;
        cout << "Enter arrival time for process " << i + 1 << endl;
        cin >> p[i].arrivalTime;
        p[i].id = i + 1;
    }
    // first come first serve cpu scheduling
    int time = 0;
    for (int i = 0; i < num; i++)
    {
        if (p[i].arrivalTime >= time)
        {
            time = p[i].arrivalTime;
        }
        time += p[i].burstTime;
        p[i].completionTime = time;
        p[i].turnAroundTime = p[i].completionTime - p[i].arrivalTime;
        p[i].waitingTime = p[i].turnAroundTime - p[i].burstTime;
        cout << "Process " << p[i].id << " completed at time " << p[i].completionTime << endl
             << "Waiting time: " << p[i].waitingTime << endl
             << "Turn around time: " << p[i].turnAroundTime << endl;
    }
    return 0;
}
```

Priority-Preemptive

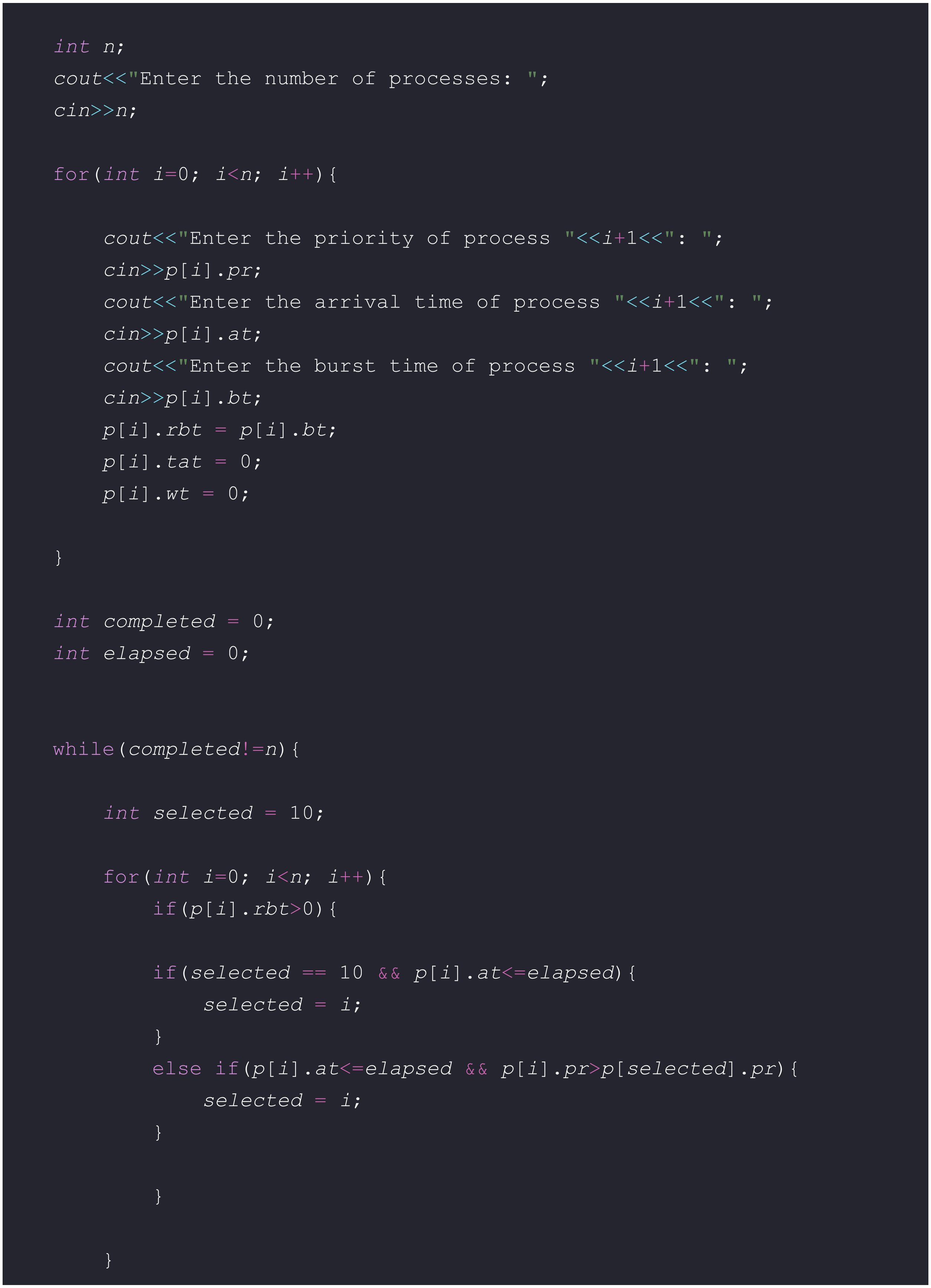
```
// priority preempetive

#include<iostream>

using namespace std;

struct process{
    int AT, BT, RBT, PR, TAT, WT;
};

int main(){

    process arr[10];

    int n;

    cout<<"Enter the number of processes: ";
    cin>>n;

    for(int i=0; i<n; i++){
        cout<<"Enter the arrival time of process P"<<i<<": ";
        cin>>arr[i].AT;
```



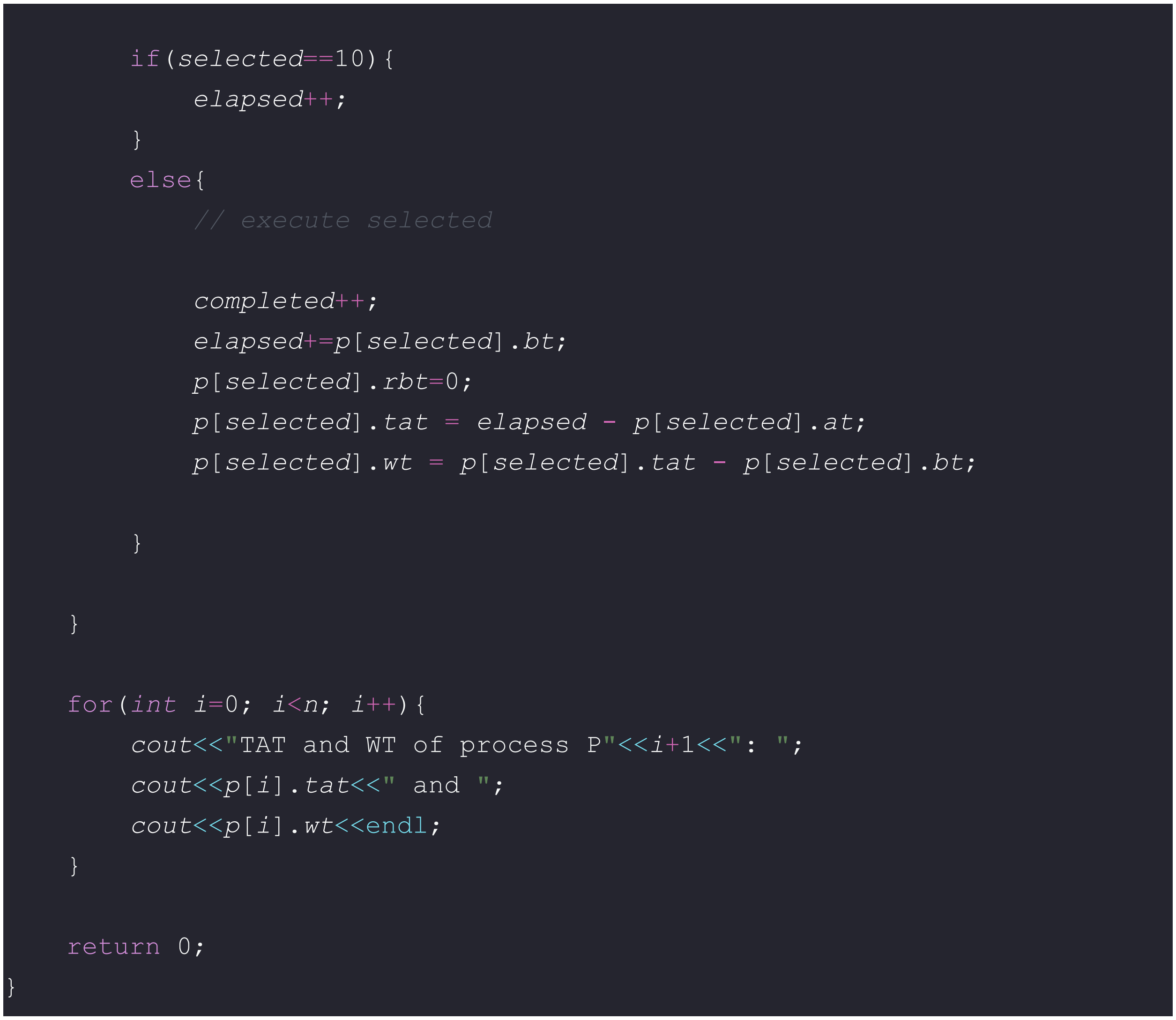
```
        cout<<"Enter the burst time of process P"<<i<<": ";
        cin>>arr[i].BT;
        cout<<"Enter the priority of process P"<<i<<": ";
        cin>>arr[i].PR;
        arr[i].RBT = arr[i].BT;
    }

    int elapsed = 0;
    int completed = 0;

    while(completed!=n){

        int selected = n+1;

        for(int i = 0; i<n; i++){
            if(arr[i].AT<=elapsed && arr[i].RBT>0){
                if(selected == n+1){
                    selected = i;
                }
                else{
                    if(arr[i].PR>arr[selected].PR){
                        selected = i;
                    }
                }
            }
        }

        if(selected == n+1){
            elapsed++;
        }
        else{
            // execute the selected

            arr[selected].RBT-=1;
            elapsed++;

            if(arr[selected].RBT==0){
                completed++;
                arr[selected].TAT = elapsed - arr[selected].AT;
                arr[selected].WT = arr[selected].TAT-arr[selected].BT;
            }


        }

    }

    float totalWT = 0;

    for(int i = 0; i<n; i++){
        cout<<"Waiting Time of process P"<<i<<" is "<<arr[i].WT<<endl;
        totalWT+=arr[i].WT;
    }

    cout<<endl<<"The Average Waiting Time is: "<<totalWT/n<<endl;

    return 0;
}
```

Priority non-preemptive

```
#include<iostream>

using namespace std;

struct process{
    int pr;
    int at;
    int bt;
    int rbt;
    int tat;
    int wt;
};

// higher number means higher priority

int main(){

    process p[10];
```



```
int n;
cout<<"Enter the number of processes: ";
cin>>n;

for(int i=0; i<n; i++){

    cout<<"Enter the priority of process "<<i+1<<": ";
    cin>>p[i].pr;
    cout<<"Enter the arrival time of process "<<i+1<<": ";
    cin>>p[i].at;
    cout<<"Enter the burst time of process "<<i+1<<": ";
    cin>>p[i].bt;
    p[i].rbt = p[i].bt;
    p[i].tat = 0;
    p[i].wt = 0;

}

int completed = 0;
int elapsed = 0;


while(completed!=n){

    int selected = 10;

    for(int i=0; i<n; i++){
        if(p[i].rbt>0){

        if(selected == 10 && p[i].at<=elapsed){
            selected = i;
        }
        else if(p[i].at<=elapsed && p[i].pr>p[selected].pr){
            selected = i;
        }

        }

    }
```



```
        if(selected==10){
            elapsed++;
        }
        else{
            // execute selected

            completed++;
            elapsed+=p[selected].bt;
            p[selected].rbt=0;
            p[selected].tat = elapsed - p[selected].at;
            p[selected].wt = p[selected].tat - p[selected].bt;

        }

    }

    for(int i=0; i<n; i++){
        cout<<"TAT and WT of process P"<<i+1<<": ";
        cout<<p[i].tat<<" and ";
        cout<<p[i].wt<<endl;
    }

    return 0;
}
```

Semaphore

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
// Define a global variable to be shared by two threads
int common_variable = 10;
// Define a semaphore
sem_t semaphore;
void *Incre(void *arg) {
        int thread_id = *((int *)arg);
        // Wait on the semaphore (Increment it)
```

```
    sem_wait(&semaphore);
    int i= common_variable;
    // Critical section: update the common variable
    i += 1;
    sleep(2);
    common_variable =i;
    printf("Thread %d updated common_variable to %d\n", thread_id, common_variable);
    // Signal that we're done with the critical section (increment the semaphore)
    sem_post(&semaphore);
    pthread_exit(NULL);
}
void *Decr(void *arg) {
    int thread_id = *((int *)arg);
    // Wait on the semaphore (decrement it)
    sem_wait(&semaphore);
    // Critical section: update the common variable
    int i= common_variable;
    // Critical section: update the common variable
    i -= 2;
    common_variable =i;
    printf("Thread %d decremented common_variable to %d\n", thread_id,
common_variable);
    // Signal that we're done with the critical section (increment the semaphore)
    sem_post(&semaphore);
    pthread_exit(NULL);
}

int main() {
    // Initialize the semaphore with a value of 1
    sem_init(&semaphore, 0, 1);
    pthread_t thread1, thread2;
    int thread_id1 = 1;
    int thread_id2 = 2;
    // Create two threads
    pthread_create(&thread1, NULL, Incre, &thread_id1);
    pthread_create(&thread2, NULL, Decr, &thread_id2);
    // Wait for the threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    // Destroy the semaphore
    sem_destroy(&semaphore);
    printf("Final common_variable value: %d\n", common_variable);
    return 0;
}
```

Multithreading

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

int data[10] = {5, 2, 9, 1, 5, 6, 3, 7, 10, 4};
int sum = 0;
int max = 0;
int min = 0;
int size = 10;

void *Sum(void *arg) {
    for(int i = 0; i<size; i++) {
        sum += data[i];
    }
    pthread_exit(NULL);
}

void *Max(void *arg) {
    max = data[0];
    for(int i = 1; i < size; i++) {
        if(data[i] > max) {
            max = data[i];
        }
    }
    pthread_exit(NULL);
}

void *Min(void *arg) {
    min = data[0];
    for(int i = 1; i < size; i++) {
        if(data[i] < min) {
            min = data[i];
        }
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t tid1, tid2, tid3;
    pthread_create(&tid1, NULL, Sum, NULL);
    pthread_create(&tid2, NULL, Max, NULL);
    pthread_create(&tid3, NULL, Min, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    printf("Sum: %d\n", sum);
    printf("Maximum: %d\n", max);
    printf("Minimum: %d\n", min);

    return 0;
}
```

IPC - POSIX

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main()
{
const int size = 4096;

const char* mess_0 = "Shaurya ";
const char* mess_1 = "Verma \n";

int shm_fd = shm_open("OS", O_CREAT|O_RDWR, 0666);
ftruncate(shm_fd, size);

void* ptr = mmap(0, size, PROT_WRITE, MAP_SHARED, shm_fd, 0);

sprintf(ptr,"%s", mess_0);
ptr += strlen(mess_0);


sprintf(ptr,"%s", mess_1);
ptr += strlen(mess_1);
}
```

Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
const int size = 4096;

int shm_fd = shm_open("OS", O_RDONLY, 0666);

void* ptr = mmap(0, size, PROT_READ, MAP_SHARED, shm_fd, 0);

printf("%s", (char*) ptr);

shm_unlink("OS");
}
```

Pipes

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
int fd[2], n;
char buffer[100];
pid_t p;

pipe(fd);
p = fork();

if(p>0)
{
write(fd[1],"Hello\n",6);
}else{
```

```
int m = read(fd[0],buffer,100);

printf("%s", buffer);

//write(1,buffer,m); "1" prints to terminal
}
}
```

Message Queues

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10

struct mesg_buffer {
        long mesg_type;
        char mesg_text[100];
} message;

int main()
{
        key_t key;
        int msgid;

        key = ftok("Shaurya", 65);


        msgid = msgget(key, 0666 | IPC_CREAT);
        message.mesg_type = 1;

        printf("Write Data : ");
        fgets(message.mesg_text,MAX,stdin);


        msgsnd(msgid, &message, sizeof(message), 0);

        printf("Data send is : %s \n", message.mesg_text);

        return 0;
}
```

Receive

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>


struct mesg_buffer {
	long mesg_type;
	char mesg_text[100];
} message;

int main()
{
	key_t key;
	int msgid;

	key = ftok("Shaurya", 65);


	msgid = msgget(key, 0666 | IPC_CREAT);


	msgrcv(msgid, &message, sizeof(message), 1, 0);

	printf("Data Received is : %s \n", message.mesg_text);

	msgctl(msgid, IPC_RMID, NULL);

	return 0;
}
```