Page replacement FIFO

```
#include <stdio.h>
#include <stdbool.h>
int main() {
  int num_frames, num_pages, frames[20],
pages[50];
  int page_faults = 0, page_hits = 0;
  printf("--- FIFO Page Replacement ---\n");
  printf("Enter frames: ");
  scanf("%d", &num_frames);
  printf("Enter sequence length: ");
  scanf("%d", &num_pages);
  printf("Enter page sequence (space separated):
");
  for (int i = 0; i < num_pages; i++) {
    scanf("%d", &pages[i]);
  }
  for (int i = 0; i < num_frames; i++) frames[i] = -1;
  int victim_index = 0;
  printf("\nRef | Frames | Status\n");
  for (int i = 0; i < num_pages; i++) {
    int page = pages[i];
    bool hit = false;
    for (int j = 0; j < num_frames; j++) {
      if (frames[j] == page) {
        hit = true;
        page_hits++;
        break;
      }}
    printf("%3d |", page);
    if (!hit) {
      frames[victim_index] = page;
      victim_index = (victim_index + 1) %
num_frames;
      page_faults++;
      for (int j = 0; j < num_frames; j++) {
        (frames[j] != -1) ? printf("%3d ", frames[j])
: printf("  - ");
      }
      printf(" | FAULT\n");
    } else {
      for (int j = 0; j < num_frames; j++) {
        (frames[j] != -1) ? printf("%3d ", frames[j])
: printf("  - ");
      }
      printf(" | HIT\n");}}
  printf("\nTotal Hits: %d, Total Faults: %d\n",
page_hits, page_faults);
  return 0;
}
```

Page replacement LRU

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
int main() {
  int num_frames, num_pages, frames[20], pages[50],
last_used_time[20];
  int page_faults = 0, page_hits = 0;
  int time_counter = 0;
  printf("--- LRU Page Replacement ---\n");
  printf("Enter frames: ");
  scanf("%d", &num_frames);
  printf("Enter sequence length: ");
  scanf("%d", &num_pages);
  printf("Enter page sequence (space separated): ");
  for (int i = 0; i < num_pages; i++) {
    scanf("%d", &pages[i]);   }
  for (int i = 0; i < num_frames; i++) {
    frames[i] = -1;
    last_used_time[i] = 0;   }
  printf("\nRef | Frames | Status\n");
  for (int i = 0; i < num_pages; i++) {
    int page = pages[i];
    bool hit = false;
    int frame_index = -1;
    for (int j = 0; j < num_frames; j++) {
      if (frames[j] == page) {
        hit = true;
        page_hits++;
        frame_index = j;
        break;}}
    time_counter++;
    printf("%3d |", page);
    if (hit) {
      last_used_time[frame_index] = time_counter; //
Update last used time
      for (int j = 0; j < num_frames; j++) {
        (frames[j] != -1) ? printf("%3d ", frames[j]) :
printf("  - ");     }
      printf(" | HIT\n");
    } else {
      page_faults++;
      int lru_index = -1, min_time = INT_MAX;
      int empty_index = -1;
      for (int j = 0; j < num_frames; j++) {
        if (frames[j] == -1) {
          empty_index = j;
          break;   }
        if (last_used_time[j] < min_time) {
          min_time = last_used_time[j];
          lru_index = j;}}
      int replace_index = (empty_index != -1) ?
empty_index : lru_index;
      frames[replace_index] = page;
      last_used_time[replace_index] = time_counter;
      for (int j = 0; j < num_frames; j++) {
        (frames[j] != -1) ? printf("%3d ", frames[j]) :
printf("  - ");     }
      printf(" | FAULT\n");}   }
  printf("\nTotal Hits: %d, Total Faults: %d\n", page_hits,
page_faults);
  return 0;}
```

Page replace OPTIMAL

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
int find_opt_victim(int cur, int F[], int P[], int f_sz,
int p_sz) {
   int max_dist = -1, victim = -1;
   for (int i = 0; i < f_sz; i++) {
      int next_use = p_sz;
      for (int j = cur + 1; j < p_sz; j++) {
         if (F[i] == P[j]) {
            next_use = j;
            break;}}
      if (next_use == p_sz) return i;
      if (next_use > max_dist) {
         max_dist = next_use;
         victim = i;}}
   return victim;}
int main() {
   int F_SZ, P_SZ, F[20], P[50];
   int faults = 0, hits = 0;
   printf("Frames (F_SZ): "); scanf("%d", &F_SZ);
   printf("Sequence Length (P_SZ): "); scanf("%d",
&P_SZ);
   printf("Sequence: ");
   for (int i = 0; i < P_SZ; i++) scanf("%d", &P[i])
   for (int i = 0; i < F_SZ; i++) F[i] = -1;
   printf("\nRef | Frames | Status\n");
   for (int i = 0; i < P_SZ; i++) {
      int page = P[i];
      bool hit = false;
      for (int j = 0; j < F_SZ; j++) {
         if (F[j] == page) {
            hit = true;
            hits++;
            break;}}
      printf("%3d |", page);
      if (!hit) {
         faults++;
         int empty_idx = -1;
         for (int j = 0; j < F_SZ; j++) {
            if (F[j] == -1) { empty_idx = j; break; }
         int replace_idx = (empty_idx != -1) ?
empty_idx : find_opt_victim(i, F, P, F_SZ, P_SZ);
         F[replace_idx] = page;
         for (int j = 0; j < F_SZ; j++) (F[j] != -1) ?
printf("%3d ", F[j]) : printf("  - ");
         printf(" | FAULT\n");
      } else {
         for (int j = 0; j < F_SZ; j++) (F[j] != -1) ?
printf("%3d ", F[j]) : printf("  - ");
         printf(" | HIT\n");}}
   printf("\nTotal Hits: %d, Total Faults: %d\n", hits,
faults);
   return 0;
}
```

Disk schedule(FCFS)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_REQUESTS 50
int main() {
 int req[MAX_REQUESTS], n, head, mov = 0;
 float avg_mov = 0.0;
 printf("Enter the number of requests (max
%d): ", MAX_REQUESTS);
 if (scanf("%d", &n) != 1 || n <= 0 || n >
MAX_REQUESTS) {
 printf("Invalid number of requests.\n");
 return 1;
 }
 printf("Enter the initial head position: ");
 if (scanf("%d", &head) != 1 || head < 0) {
 printf("Invalid head position.\n");
 return 1;
 }
 printf("Enter the request sequence:\n");
 for (int i = 0; i < n; i++) {
 printf("Request %d: ", i + 1);
 if (scanf("%d", &req[i]) != 1 || req[i] < 0) {
 printf("Invalid request.\n");
 return 1;
 }
 }
 printf("\n--- FCFS Disk Scheduling Results ---
\n");
 int current_pos = head;
 printf("Seek Sequence: %d", current_pos);
 for (int i = 0; i < n; i++) {
 mov += abs(req[i] - current_pos);
 current_pos = req[i];

 printf(" -> %d", current_pos);
 }
 avg_mov = (float)mov / n;
 printf("\nTotal Head Movement (Seek
Time): %d\n", mov);
 printf("Average Head Movement (Seek
Time): %.2f\n", avg_mov);
 return 0;
}
```

BANKERS algoritm

```
#include <stdio.h>
int main() {
 int n, m, i, j, k;
 printf("Enter number of processes: ");
 scanf("%d", &n);
 printf("Enter number of resources: ");
 scanf("%d", &m);
 int alloc[n][m], max[n][m], avail[m];
 int need[n][m], finish[n], safeSeq[n];
 printf("\nEnter allocation matrix:\n");
 for (i = 0; i < n; i++)
 for (j = 0; j < m; j++)
 scanf("%d", &alloc[i][j]);
 printf("\nEnter maximum matrix:\n");
 for (i = 0; i < n; i++)
 for (j = 0; j < m; j++)
 scanf("%d", &max[i][j]);
 printf("\nEnter available resources:\n");
 for (j = 0; j < m; j++)
 scanf("%d", &avail[j]);
 for (i = 0; i < n; i++)
 for (j = 0; j < m; j++)
 need[i][j] = max[i][j] - alloc[i][j];
 for (i = 0; i < n; i++)
 finish[i] = 0;
 int count = 0;
 while (count < n) {
 int found = 0;
 for (i = 0; i < n; i++) {
 if (finish[i] == 0) {
 int flag = 0;
 for (j = 0; j < m; j++) {
 if (need[i][j] > avail[j]) {
 flag = 1;
 break; } }
 if (flag == 0) {
 for (k = 0; k < m; k++)
 avail[k] += alloc[i][k];
 safeSeq[count++] = i;
 finish[i] = 1;
 found = 1; }} }
 if (found == 0) {
 printf("\nSystem is in UNSAFE state!\n");
 return 0; }}
 printf("\nSystem is in SAFE state.\nSafe sequence
is: ");
 for (i = 0; i < n; i++)
 printf("P%d ", safeSeq[i]);
 printf("\n");
 return 0; }
```

Producer consumer:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int in = 0, out = 0;
sem_t empty, full;
pthread_mutex_t mutex;
void* producer(void* arg) {
 int item;
 for (int i = 0; i < 10; i++) {
 item = rand() % 100;
 sem_wait(&empty);
 pthread_mutex_lock(&mutex);
Caption
 buffer[in] = item;
 printf("Producer produced: %d\n", item);
 in = (in + 1) % BUFFER_SIZE;
 pthread_mutex_unlock(&mutex);
 sem_post(&full);
 }
 pthread_exit(NULL);
}
void* consumer(void* arg) {
 int item;
 for (int i = 0; i < 10; i++) {
 sem_wait(&full);
 pthread_mutex_lock(&mutex);
 item = buffer[out];
 printf("Consumer consumed: %d\n", item);
 out = (out + 1) % BUFFER_SIZE;
 pthread_mutex_unlock(&mutex);
 sem_post(&empty);  }
 pthread_exit(NULL); }
int main() {
 pthread_t prod, cons;
 sem_init(&empty, 0, BUFFER_SIZE);
 sem_init(&full, 0, 0);
 pthread_mutex_init(&mutex, NULL);
 pthread_create(&prod, NULL, producer, NULL);
 pthread_create(&cons, NULL, consumer, NULL);
 pthread_join(prod, NULL);
 pthread_join(cons, NULL);
 sem_destroy(&empty);
 sem_destroy(&full);
 pthread_mutex_destroy(&mutex);
 return 0; }
```

Dining philosopher

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2

#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = {0, 1, 2, 3, 4};
sem_t mutex;
sem_t S[N];
void test(int phnum) {
   if (state[phnum] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {

      state[phnum] = EATING;

      sem_post(&S[phnum]);

      printf("Philosopher %d takes fork %d and
%d\n", phnum + 1, LEFT + 1, phnum + 1);
      printf("Philosopher %d is EATING \n",
phnum + 1);
   }
}

void take_fork(int phnum) {
   sem_wait(&mutex);
   state[phnum] = HUNGRY;
   printf("Philosopher %d is HUNGRY \n",
phnum + 1);
   test(phnum);

   sem_post(&mutex);
   sem_wait(&S[phnum]);
    sleep(1);
}
void put_fork(int phnum) {
   sem_wait(&mutex);

   state[phnum] = THINKING;
   printf("Philosopher %d putting fork %d and
%d down\n", phnum + 1, LEFT + 1, phnum +
1);
   printf("Philosopher %d is THINKING \n",
phnum + 1);

   test(LEFT);
   test(RIGHT);
   sem_post(&mutex);
}
void* philosopher(void* num) {
   int phnum = *(int*)num;
   while (1) {
      sleep(1 + (rand() % 3));
      take_fork(phnum);
      sleep(1 + (rand() % 3));
       put_fork(phnum);
   }
}
int main() {
   int i;
   pthread_t thread_id[N];
   sem_init(&mutex, 0, 1);
   for (i = 0; i < N; i++) {
      sem_init(&S[i], 0, 0);
   }
   for (i = 0; i < N; i++) {
      pthread_create(&thread_id[i], NULL,
philosopher, &phil[i]);
   }
   for (i = 0; i < N; i++) {
      pthread_join(thread_id[i], NULL);
   }
    sem_destroy(&mutex);
   for (i = 0; i < N; i++) {
      sem_destroy(&S[i]);
   }

   return 0;
}
```

READER WRITER

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <time.h>
sem_t wrt;
pthread_mutex_t mutex;
int cnt = 1;
int numreader = 0;
#define MAX_THREADS 20
int thread_ids[MAX_THREADS];
void* writer(void* wno) {
   int writer_id = *(int*)wno;
   sleep(rand() % 2);
   sem_wait(&wrt);
   cnt *= 2;
   printf("Writer %d modified count to %d\n",
writer_id, cnt);
   sleep(1);
   sem_post(&wrt);
   return NULL;
}
void* reader(void* rno) {
   int reader_id = *(int*)rno;
   sleep(rand() % 3);
   pthread_mutex_lock(&mutex);
   numreader++;
   if (numreader == 1) {
      sem_wait(&wrt);
   }
   pthread_mutex_unlock(&mutex);
   printf("Reader %d reading count as %d\n",
reader_id, cnt);
   sleep(1);
   pthread_mutex_lock(&mutex);
   numreader--;
   if (numreader == 0) {
      sem_post(&wrt);
   }
   pthread_mutex_unlock(&mutex);
   return NULL;
}
int main() {
   int num_readers, num_writers;
   srand(time(NULL));
   printf("Enter the number of Reader threads
(Max %d): ", MAX_THREADS);
   if (scanf("%d", &num_readers) != 1 ||
num_readers <= 0 || num_readers >
MAX_THREADS) {
      printf("Invalid input.\n");
      return 1;}
   printf("Enter the number of Writer threads
(Max %d): ", MAX_THREADS);
   if (scanf("%d", &num_writers) != 1 ||
num_writers <= 0 || num_writers >
MAX_THREADS) {
      printf("Invalid input.\n");
      return 1;}
   pthread_t *read_threads =
(pthread_t*)malloc(num_readers *
sizeof(pthread_t));
   pthread_t *write_threads =
(pthread_t*)malloc(num_writers *
sizeof(pthread_t));
   if (!read_threads || !write_threads) {
      perror("Memory allocation failed");
      return 1;}
   for (int i = 0; i < MAX_THREADS; i++) {
      thread_ids[i] = i + 1;
   }
   pthread_mutex_init(&mutex, NULL);
   sem_init(&wrt, 0, 1);

   for (int i = 0; i < num_writers; i++) {
      if (pthread_create(&write_threads[i],
NULL, writer, &thread_ids[i]) != 0) {
         perror("Error creating writer thread");
         return 1;
      }
   }
   for (int i = 0; i < num_readers; i++) {
      if (pthread_create(&read_threads[i],
NULL, reader, &thread_ids[i + num_writers])
!= 0) {
         perror("Error creating reader thread");
         return 1; }  }
   for (int i = 0; i < num_writers; i++) {
      pthread_join(write_threads[i], NULL); }
   for (int i = 0; i < num_readers; i++) {
      pthread_join(read_threads[i], NULL);  }
   pthread_mutex_destroy(&mutex);
   sem_destroy(&wrt);
   free(read_threads);
   free(write_threads);
   printf("\nFinal count value: %d\n", cnt);
   return 0;
}
```

ROUND ROBIN

```
#include <stdio.h>

struct Process {
   int id, at, bt, rt, ct, tat, wt, rtm, first_run;
};

int main() {
   int n, tq, time = 0, done = 0,
executed_in_cycle, execution_time;
   float total_tat = 0, total_wt = 0, total_rtm =
0;

   printf("Enter number of processes: ");
   if (scanf("%d", &n) != 1 || n <= 0) return 1;

   struct Process p[n];

   printf("Enter Arrival Time and Burst Time for
each process:\n");
   for(int i=0; i<n; i++) {
      p[i].id = i + 1;
      printf("P%d (AT BT): ", i+1);
      if (scanf("%d %d", &p[i].at, &p[i].bt) != 2
|| p[i].bt <= 0) return 1;
      p[i].rt = p[i].bt;
      p[i].first_run = 0;
   }

   printf("Enter Time Quantum: ");
   if (scanf("%d", &tq) != 1 || tq <= 0) return 1;

   while(done < n) {
      executed_in_cycle = 0;

      for(int i=0; i<n; i++) {
         if(p[i].at <= time && p[i].rt > 0) {
            executed_in_cycle = 1;

            if (p[i].first_run == 0) {
               p[i].first_run = time;
               p[i].rtm = p[i].first_run - p[i].at;
            }

            if(p[i].rt > tq) {
               execution_time = tq;
            } else {
               execution_time = p[i].rt;
            }

            time += execution_time;
            p[i].rt -= execution_time;

            if(p[i].rt == 0) {
               p[i].ct = time;
               p[i].tat = p[i].ct - p[i].at;
               p[i].wt = p[i].tat - p[i].bt;
               done++;

               total_tat += p[i].tat;
               total_wt += p[i].wt;
               total_rtm += p[i].rtm;
            }
         }
      }

      if(!executed_in_cycle) {
         time++;
      }
   }


printf("\nPID\tAT\tBT\tCT\tTAT\tWT\tRT\n");
   for(int i=0; i<n; i++) {

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
            p[i].id, p[i].at, p[i].bt, p[i].ct,
            p[i].tat, p[i].wt, p[i].rtm);
   }

   printf("\nAverage Turnaround Time:
%.2f\n", total_tat / n);
   printf("Average Waiting Time: %.2f\n",
total_wt / n);
   printf("Average Response Time: %.2f\n",
total_rtm / n);

   return 0;
}
```

FCFS

```
#include <stdio.h>
struct Process {
 int id, at, bt, wt, tat, ct;
};
int main() {
 int n;
 printf("Enter number of processes: ");
 scanf("%d", &n);
 struct Process p[n];
 printf("Enter Arrival Time and Burst
Time:\n");
 for (int i = 0; i < n; i++) {
 p[i].id = i + 1;
 printf("P%d (AT BT): ", i + 1);
 scanf("%d %d", &p[i].at, &p[i].bt);
 }
 for (int i = 0; i < n - 1; i++) {
 for (int j = 0; j < n - i - 1; j++) {
 if (p[j].at > p[j + 1].at) {
 struct Process temp = p[j];
 p[j] = p[j + 1];
 p[j + 1] = temp;
 }
 }
 }
 int time = 0;
 float avgWT = 0, avgTAT = 0;
 for (int i = 0; i < n; i++) {
 if (time < p[i].at)
 time = p[i].at;
 time += p[i].bt;
 p[i].ct = time;
 p[i].tat = p[i].ct - p[i].at;
 p[i].wt = p[i].tat - p[i].bt;
 avgWT += p[i].wt;
 avgTAT += p[i].tat;
 }
 printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
 for (int i = 0; i < n; i++) {
 printf("P%d\t%d\t%d\t%d\t%d\t%d\n",
p[i].id, p[i].at, p[i].bt,
 p[i].ct, p[i].tat, p[i].wt);
 }
 printf("\nAverage Waiting Time = %.2f",
avgWT / n);
 printf("\nAverage Turnaround Time = %.2f",
avgTAT / n);
 return 0;
}
```

First FIT

```
#include <stdio.h>
void firstFit(int blockSize[], int m, int
processSize[], int n) {
   int allocation[n];
   for (int i = 0; i < n; i++) allocation[i] = -1;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         if (blockSize[j] >= processSize[i]) {
            allocation[i] = j + 1;  // Store block
number starting from 1
            blockSize[j] -= processSize[i];
            break;
         }
      }
   }
   printf("\nFirst
Fit:\nProcess\tSize\tBlock\n");
   for (int i = 0; i < n; i++)
      printf("%d\t%d\t%d\n", i + 1,
processSize[i], allocation[i] == -1 ? -1 :
allocation[i]);
}
int main() {
   int m, n;
   printf("Enter number of memory blocks: ");
   scanf("%d", &m);
   int blockSizeOriginal[m];

   printf("Enter sizes of %d memory blocks:\n",
m);
   for (int i = 0; i < m; i++) {
      printf("Block %d: ", i + 1);
      scanf("%d", &blockSizeOriginal[i]);
   }
   printf("\nEnter number of processes: ");
   scanf("%d", &n);
   int processSize[n];
   printf("Enter sizes of %d processes:\n", n);
   for (int i = 0; i < n; i++) {
      printf("Process %d: ", i + 1);
      scanf("%d", &processSize[i]);
   }
   int blocks[m];
   for (int i = 0; i < m; i++) blocks[i] =
blockSizeOriginal[i];
   firstFit(blocks, m, processSize, n);
   return 0;
}
```

BEST fit

```
#include <stdio.h>
void bestFit(int blockSize[], int m, int
processSize[], int n) {
  int allocation[n];
  for (int i = 0; i < n; i++) allocation[i] = -1;
  for (int i = 0; i < n; i++) {
    int bestIdx = -1;
    for (int j = 0; j < m; j++) {
      if (blockSize[j] >= processSize[i]) {
        if (bestIdx == -1 || blockSize[j] <
blockSize[bestIdx])
          bestIdx = j;}   }
    if (bestIdx != -1) {
      allocation[i] = bestIdx + 1;
      blockSize[bestIdx] -= processSize[i];
    }
  }
  printf("\nBest
Fit:\nProcess\tSize\tBlock\n");
  for (int i = 0; i < n; i++)
    printf("%d\t%d\t%d\n", i + 1,
processSize[i], allocation[i] == -1 ? -1 :
allocation[i]);}
int main() {
  int m, n;
  printf("Enter number of memory blocks: ");
  scanf("%d", &m);
  int blockSizeOriginal[m];
  printf("Enter sizes of %d memory blocks:\n",
m);
  for (int i = 0; i < m; i++) {
    printf("Block %d: ", i + 1);
    scanf("%d", &blockSizeOriginal[i]);
  }
  printf("\nEnter number of processes: ");
  scanf("%d", &n);
  int processSize[n];

  printf("Enter sizes of %d processes:\n", n);
  for (int i = 0; i < n; i++) {
    printf("Process %d: ", i + 1);
    scanf("%d", &processSize[i]);
  }
  int blocks[m];
      // Best Fit
  for (int i = 0; i < m; i++) blocks[i] =
blockSizeOriginal[i];
  bestFit(blocks, m, processSize, n);
      return 0;
```

Worst fit

```
#include <stdio.h>
// Worst Fit
void worstFit(int blockSize[], int m, int
processSize[], int n) {
  int allocation[n];
  for (int i = 0; i < n; i++) allocation[i] = -1;
  for (int i = 0; i < n; i++) {
    int worstIdx = -1;
    for (int j = 0; j < m; j++) {
      if (blockSize[j] >= processSize[i]) {
        if (worstIdx == -1 || blockSize[j] >
blockSize[worstIdx])
          worstIdx = j;
      }
    }
    if (worstIdx != -1) {
      allocation[i] = worstIdx + 1;
      blockSize[worstIdx] -= processSize[i];
    }
  }
  printf("\nWorst
Fit:\nProcess\tSize\tBlock\n");
  for (int i = 0; i < n; i++)
    printf("%d\t%d\t%d\n", i + 1,
processSize[i], allocation[i] == -1 ? -1 :
allocation[i]);
} int main() {
  int m, n;
  printf("Enter number of memory blocks: ");
  scanf("%d", &m);
  int blockSizeOriginal[m];
  pritf("Entersizesof%dmemoryblocks:\n", m)
  for (int i = 0; i < m; i++) {
    printf("Block %d: ", i + 1);
    scanf("%d", &blockSizeOriginal[i]);
  }
  printf("\nEnter number of processes: ");
  scanf("%d", &n);
  int processSize[n];
  printf("Enter sizes of %d processes:\n", n);
  for (int i = 0; i < n; i++) {
    printf("Process %d: ", i + 1);
    scanf("%d", &processSize[i]);  }
  int blocks[m];
  for (int i = 0; i < m; i++) blocks[i] =
blockSizeOriginal[i];
  worstFit(blocks, m, processSize, n);
  return 0;
}
```

Priority(Non preemptive)

```
#include <stdio.h>
struct Process {
   int id, at, bt, pr, ct, tat, wt;
   int done;
};
int main() {
   int n, time = 0, done = 0;
   float total_tat = 0.0, total_wt = 0.0;
   printf("Enter number of processes: ");
   if (scanf("%d", &n) != 1 || n <= 0) {
      printf("Invalid number of processes.\n");
      return 1;}
   struct Process p[n];
   printf("Enter Arrival Time, Burst Time,
Priority (Lower number = Higher Priority):\n");
   for(int i = 0; i < n; i++) {
      p[i].id = i + 1;
      printf("P%d (AT BT PR): ", i + 1);
      if (scanf("%d %d %d", &p[i].at, &p[i].bt,
&p[i].pr) != 3 || p[i].bt <= 0) {
         printf("Invalid input.\n");
         return 1;    }
      p[i].done = 0;    }
   while(done < n) {
      int idx = -1;
      int minPr = 9999;
            for(int i = 0; i < n; i++) {
         if(!p[i].done && p[i].at <= time &&
p[i].pr < minPr) {
            minPr = p[i].pr;
            idx = i;}  }
      if(idx != -1) {
         time += p[idx].bt;
         p[idx].ct = time;
         p[idx].tat = p[idx].ct - p[idx].at;
         p[idx].wt = p[idx].tat - p[idx].bt;
         total_tat += p[idx].tat;
         total_wt += p[idx].wt;
         p[idx].done = 1;
         done++;
      } else { time++;}}
printf("\nPID\tAT\tBT\tPR\tCT\tTAT\tWT\n");
   for(int i = 0; i < n; i++) {
     printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           p[i].id, p[i].at, p[i].bt, p[i].pr,
           p[i].ct, p[i].tat, p[i].wt);  }
   printf("Average Turnaround Time (ATAT):
%.2f\n", total_tat / n);
   printf("Average Waiting Time (AWT):
%.2f\n", total_wt / n);  return 0;}
```

Priority(Preemptive)

```
#include <stdio.h>
struct Process {
   int id, at, bt, pr, rt, ct, tat, wt; };
int main() {
   int n, time = 0, done = 0;
   float total_tat = 0.0, total_wt = 0.0;
   printf("Enter number of processes: ");
   if (scanf("%d", &n) != 1 || n <= 0) {
      printf("Invalid number of processes.\n");
      return 1;}
   struct Process p[n];
   printf("Enter Arrival Time, Burst Time,
Priority (Lower number = Higher Priority):\n");
   for(int i = 0; i < n; i++) {
      p[i].id = i + 1;
      printf("P%d (AT BT PR): ", i + 1);
      if (scanf("%d %d %d", &p[i].at, &p[i].bt,
&p[i].pr) != 3 || p[i].bt <= 0) {
         printf("Invalid input.\n");
         return 1;       }
      p[i].rt = p[i].bt; }
   while(done < n) {
      int idx = -1;
      int minPr = 9999;
      for(int i = 0; i < n; i++) {
         if(p[i].at <= time && p[i].rt > 0 &&
p[i].pr < minPr) {
            minPr = p[i].pr;
            idx = i;       }   }
      if(idx != -1) {
         p[idx].rt--;
         time++;
         if(p[idx].rt == 0) {
            p[idx].ct = time;
            p[idx].tat = p[idx].ct - p[idx].at;
            p[idx].wt = p[idx].tat - p[idx].bt;
            total_tat += p[idx].tat;
            total_wt += p[idx].wt;
            done++;       }
      } else {
         // CPU is idle, advance time
         time++;}}
   printf("\n--- Preemptive Priority Scheduling
Results ---\n");
 printf("\nPID\tAT\tBT\tPR\tCT\tTAT\tWT\n");
   for(int i = 0; i < n; i++) {
     printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           p[i].id, p[i].at, p[i].bt, p[i].pr,
           p[i].ct, p[i].tat, p[i].wt); } return 0;  }
```

SJF(Preemptive) OR SRTF

```
#include <stdio.h>
#include <limits.h>
struct Process {
 int id, at, bt, rt, wt, tat, ct;
};
int main() {
 int n;
 printf("Enter number of processes: ");
 scanf("%d", &n);
 struct Process p[n];
 for (int i = 0; i < n; i++) {
 p[i].id = i + 1;
 printf("P%d (AT BT): ", i + 1);
 scanf("%d %d", &p[i].at, &p[i].bt);
 p[i].rt = p[i].bt; // remaining time
 }
 int completed = 0, time = 0, idx = -1, minRT;
 float avgWT = 0, avgTAT = 0;
 while (completed < n) {
 idx = -1;
 minRT = INT_MAX;
 for (int i = 0; i < n; i++) {
 if (p[i].at <= time && p[i].rt > 0 && p[i].rt <
minRT) {
 minRT = p[i].rt;
 idx = i; } }
 if (idx != -1) {
 p[idx].rt--;
 if (p[idx].rt == 0) {
 completed++;
 p[idx].ct = time + 1;
 p[idx].tat = p[idx].ct - p[idx].at;
 p[idx].wt = p[idx].tat - p[idx].bt;
 avgWT += p[idx].wt;
 avgTAT += p[idx].tat;}}
 time++; }
 printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
 for (int i = 0; i < n; i++) {
 printf("P%d\t%d\t%d\t%d\t%d\t%d\n",
p[i].id, p[i].at, p[i].bt,
 p[i].ct, p[i].tat, p[i].wt); }
 printf("\nAverage Waiting Time = %.2f",
avgWT / n);
 printf("\nAverage Turnaround Time = %.2f",
avgTAT / n);
 return 0;}
```

SJF(Non preemptive)

```
#include <stdio.h>
#include <limits.h>
struct Process {
 int id, at, bt, wt, tat, ct, completed;
};
int main() {
 int n;
 printf("Enter number of processes: ");
 scanf("%d", &n);
 struct Process p[n];
 for (int i = 0; i < n; i++) {
 p[i].id = i + 1;
 printf("P%d (AT BT): ", i + 1);
 scanf("%d %d", &p[i].at, &p[i].bt);
 p[i].completed = 0;
 }
 int time = 0, completed = 0;
 float avgWT = 0, avgTAT = 0;
 while (completed < n) {
 int idx = -1, minBT = INT_MAX;
 for (int i = 0; i < n; i++) {
 if (p[i].at <= time && p[i].completed == 0 &&
p[i].bt < minBT) {
 minBT = p[i].bt;
 idx = i;
 }
 }
 if (idx != -1) {
 time += p[idx].bt;
 p[idx].ct = time;
 p[idx].tat = p[idx].ct - p[idx].at;
 p[idx].wt = p[idx].tat - p[idx].bt;
 p[idx].completed = 1;
 avgWT += p[idx].wt;
 avgTAT += p[idx].tat;
 completed++;
 } else {
 time++;
 }
 }
 printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
 for (int i = 0; i < n; i++) {
 printf("P%d\t%d\t%d\t%d\t%d\t%d\n",
p[i].id, p[i].at, p[i].bt,
 p[i].ct, p[i].tat, p[i].wt); }
 printf("\nAverage Waiting Time = %.2f",
avgWT / n);
 printf("\nAverage Turnaround Time = %.2f",
avgTAT / n);
 return 0;}
```

SSTF disk:

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

int main() {
   int n, i, j;
   printf("Enter the number of disk requests:
");
   scanf("%d", &n);

   int requests[n], visited[n];
   printf("Enter the disk requests: ");
   for(i = 0; i < n; i++) {
      scanf("%d", &requests[i]);
      visited[i] = 0;
   }

   int head;
   printf("Enter initial head position: ");
   scanf("%d", &head);

   int totalSeek = 0;
   printf("Seek sequence: %d", head);

   for(i = 0; i < n; i++) {
      int minDist = INT_MAX, idx = -1;
      for(j = 0; j < n; j++) {
         if(!visited[j] && abs(requests[j]-head) <
minDist) {
            minDist = abs(requests[j]-head);
            idx = j;
         }
      }
      visited[idx] = 1;
      totalSeek += minDist;
      head = requests[idx];
      printf(" -> %d", head);
   }

   printf("\nTotal seek time: %d\n", totalSeek);
   return 0;
}
```

Scan disk

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int cmp(const void *a, const void *b) {
   return (*(int *)a - *(int *)b);
}

int main() {
   int n, head, diskSize, i;
printf("Enter number of requests, initial head,
and disk size (e.g., 8 50 200):\n");
   if (scanf("%d %d %d", &n, &head, &diskSize)
!= 3) return 1;
   int requests[n];
   printf("Enter the %d disk requests:\n", n);
   for(i = 0; i < n; i++) scanf("%d",
&requests[i]);

   qsort(requests, n, sizeof(int), cmp);

   int totalSeek = 0, idx = n, currentHead =
head;
   for(i = 0; i < n; i++) if(requests[i] >= head) {
idx = i; break; }

   printf("\nSeek sequence: %d", head);
   for(i = idx; i < n; i++) {
      totalSeek += abs(requests[i] -
currentHead);
      currentHead = requests[i];
      printf(" -> %d", currentHead);
   }
   int maxCylinder = diskSize - 1;
   totalSeek += abs(maxCylinder -
currentHead);
   currentHead = maxCylinder;
   printf(" -> %d", currentHead);
   totalSeek += abs(0 - currentHead);
   currentHead = 0;
   printf(" -> %d", currentHead);

   for(i = 0; i < idx; i++) {
      totalSeek += abs(requests[i] -
currentHead);
      currentHead = requests[i];
      printf(" -> %d", currentHead);
   }
   printf("\nTotal seek time: %d\n", totalSeek);
   return 0;
}
```