НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Розрахунково-графічна робота
з дисципліни
“Системи Реального Часу”

Виконав:
студент групи ІП-83
Кухаренко Олександр

Київ 2021

Мета роботи

Змоделювати роботу планувальника задач у системі реального часу.

Завдання

Змоделювати планувальник роботи системи реального часу. Дві дисципліни планування: перша – RR, друга задається викладачем або обирається самостійно.

Основні теоретичні відомості

Планування виконання завдань

Протягом існування процесу виконання його потоків може бути багаторазово перерване і продовжене. Перехід від виконання одного потоку до іншого здійснюється в результаті планування і диспетчеризації. Робота по визначенню того, в який момент необхідно перервати виконання поточного активного потоку і якому потоку дати можливість виконуватися, називається плануванням.

Планування виконання завдань (англ. Scheduling) є однією з ключових концепцій в багатозадачності і багатопроцесорних систем, як в операційних системах загального призначення, так і в операційних системах реального часу. Планування полягає в призначенні пріоритетів процесам в черзі з пріоритетами.

Найважливішою метою планування завдань є якнайповніше завантаження доступних ресурсів. Для забезпечення загальної продуктивності системи планувальник має опиратися на:

• Використання процесора — дати завдання процесору, якщо це можливо.

• Пропускну здатність — кількість процесів, що виконуються за одиницю часу.

• Час на завдання — кількість часу, для повного виконання певного процесу.

• Очікування — кількість часу, який процес очікує в черзі готових.

• Час відповіді — час, який проходить від подання запиту до першої відповіді на запит.

• Справедливість — Рівність процесорного часу для кожної нитi

Системи масового обслуговування

Система масового обслуговування (СМО) - це система, яка обслуговує вимоги, що надходять до неї (заявки). Основними елементами системи є вхідний потік вимог, канали обслуговування, черга вимог та вихідний потік вимог.

Вимоги (заявки) на обслуговування надходять через дискретні (постійні або випадкові) інтервали часу. Важливо знати закон розподілу вхідного потоку. Обслуговування триває деякий час, постійний або випадковий. Випадковий характер потоку заявок та часу обслуговування призводить до того, що в деякі моменти часу на вході СМО може виникнути черга, в інші моменти – канали можуть бути недозавантаженими або взагалі простоювати.

Задача теорії масового обслуговування полягає в побудові моделей, які пов‘язують задані умови роботи СМО з показниками ефективності системи, що описують її спроможність впоратися з потоком вимог. Під ефективністю, що обслуговує системи розуміють характеристику рівня виконання цією системою функцій, для яких вона призначена.

В залежності від наявності можливості очікування вступниками вимогами початку обслуговування СМО поділяються на:

• системи з втратами, в яких вимоги, що не знайшли в момент надходження жодного вільного приладу, втрачаються;

• системи з очікуванням, в яких є накопичувач нескінченної місткості для буферизації надійшли вимог, при цьому очікують вимоги утворюють чергу;

• системи з накопичувачем кінцевої місткості (чеканням і обмеженнями), в яких довжина черги не може перевищувати місткості накопичувача; при цьому вимога, що надходить в переповнену СМО (відсутні вільні місця для очікування), втрачається.

Алгоритми планування процесів

Найчастіше зустрічаються такі дві групи алгоритмів планування: побудовані на принципі квантування або на принципі пріоритетів. В першому випадку зміна активного процесу відбувається, якщо:

• процес закінчився і покинув систему;

• процес перейшов в стан Очікування;

• закінчився квант процесорного часу, відведений даному процесові.

Процес, для якого закінчився його квант, переводиться в стан готовності і очікує, коли йому буде надано новий квант процесорного часу, а на виконання відповідно до певного правила вибирається новий процес з черги готових. Жодний процес не захоплює процесор надовго, тому квантування широко використовується в системах розподілу часу. По-різному може бути організована черга готових процесів:

• циклічно;

• FIFO (перший прийшов — перший обслуговується);

• LIFO (останній прийшов — перший обслуговується).

В другому випадку використовується поняття ”пріоритет”. Пріоритет — це число, яке характеризує ступінь привілейованості процесу при використанні ресурсів комп’ютеру, зокрема, процесорного часу. Чим вище пріоритет, тим вище привілеї, тим менше часу він буде проводити в чергах.

Пріоритетами можуть призначатись адміністратором системи в залежності від важливості роботи, або внесеної плати, або обчислюватись самою ОС за певними правилами. Він може залишатись фіксованим упродовж всього життя процесу або мінятись в часі відповідно до деякого закону. В останньому випадку пріоритети називають динамічними. Є алгоритми, які використовують:

• відносні пріоритети;

• абсолютні пріоритети.

Дисципліна RR

Round-robin - це алгоритм, що займається розподілом навантаження обчислювальної системи. Для цього він використовує упорядочення черги у вигляді кільця. Round Robin використовує інтервал часу(квант) для виконання необхідного процесу. Якщо процес не завершився в межах виділеного йому кванту часу, то його робота примусово переривається, і він переміщується в хвіст черги. Дуже важливу роль відіграє кількісна величина кванту часу, адже чим більший квант, тим більше ця стратегія схожа на FIFO. І навпаки, якщо квант дуже малий, то можна вважати, що кожен процес виконується власним процесором(потужність якого ділиться між всіма процесами).

Дисципліна MFQS (Multilevel Feedback Queue Scheduling)

MFQS - це алгоритм, що займається розподілом навантаження обчислювальної системи. Для цього він використовує N кількість черг. Кількість черг визначається в залежності від призначення алгоритму, від очікуваної інтенсивності вхідного потоку, від очікуваної ваги вхідних заявок. Спочатку усі заявки потрапляють у першу чергу(1), де кожньому процесу виділяється квант часу рівний 8 мс. Якщо процес не встиг виконатися, то він попадає у наступну чергу(2). Заявки з черги 2 починають виконуватися тільки коли черга 1 - пуста. У другій черзі процесам виділяється вдвічі більше часу(16 мс). Якщо заявка не встигла виконуватись - вона йде у наступну чергу(3) і так далі, поки заявка не дійде до N черги. Що відбудеться, якщо заявка не встигне виконатись у N черзі - залежить від конкретної реалізації. Можливі варіанті:

1) Відправити заявку знову у N чергу

2) Відправити заявку у чергу 1

Яку заявку з черги обрати для опрацювання процессор обирає за алгоритмом FIFO. Це у загальному випадку, проте можливі варіації реалізації з іншими стратегіями планування.

Параметри, які потрібно визначити для алгоритму MFQS:

- Кількість черг
- Алгоритм планування для черг(може бути відмінним від FIFO)
- Критерій для збільшення пріоритету заявки
- Критерій для зменшення пріоритету заявки(і чи потрібно це взагалі)

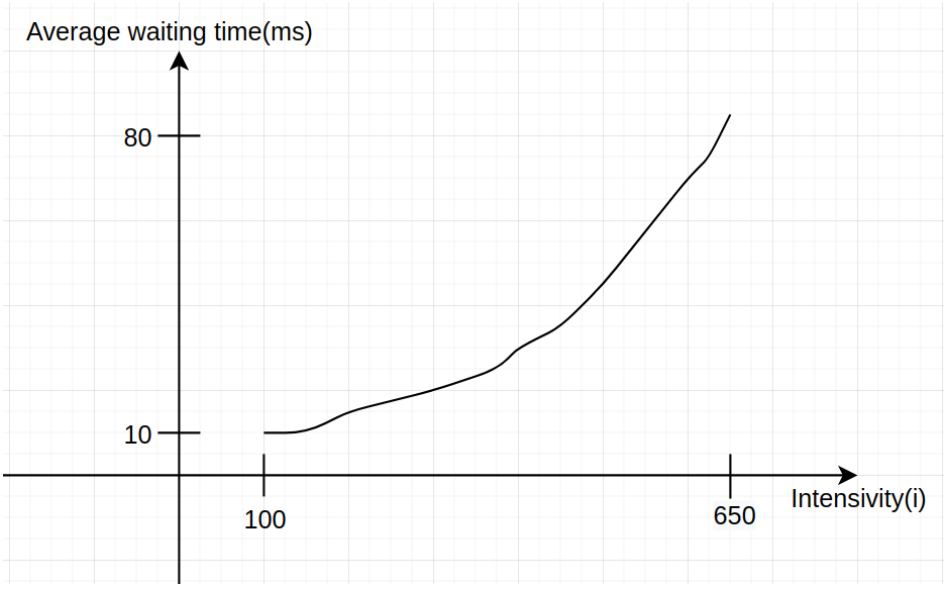
MFQS вважається найбільш загальною та універсальною, бо поєднує різні властивості з інших стратегій планування, проте й найбільш складною, через велику кількість параметрів описаних вище та складності реалізації. Ця дисципліна обслуговування є без пріоритетною та витісняючою.

Результати Роботи

Результат для RR:

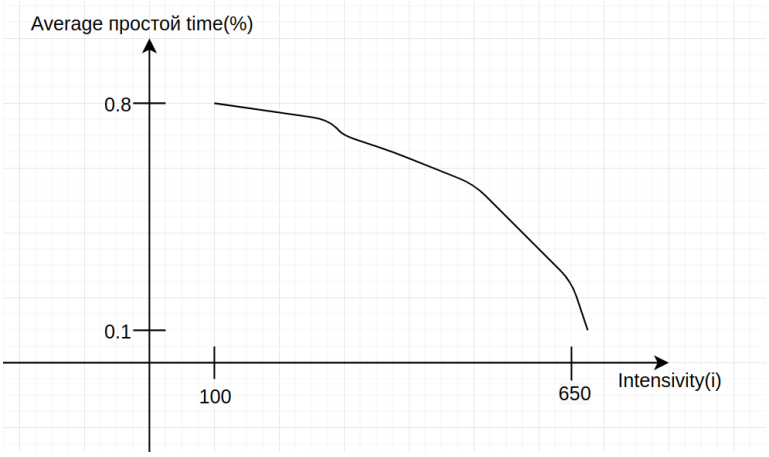
Примітка: величина інтенсивності - це коєфіцієнт який відображає величину вхідного потоку. Чим більше число - тим сильніший потік. Ця величина - усереднення, бо інтенсивність задавалася діапазоном.
Всі графіки не ідеальні, бо пріоритети для заявок генерувалися рандомно, а також, як сказано вище - величина інтенсивності усереднена.

1) **графік залежності середнього часу очікування від інтенсивності вхідного потоку заявок**



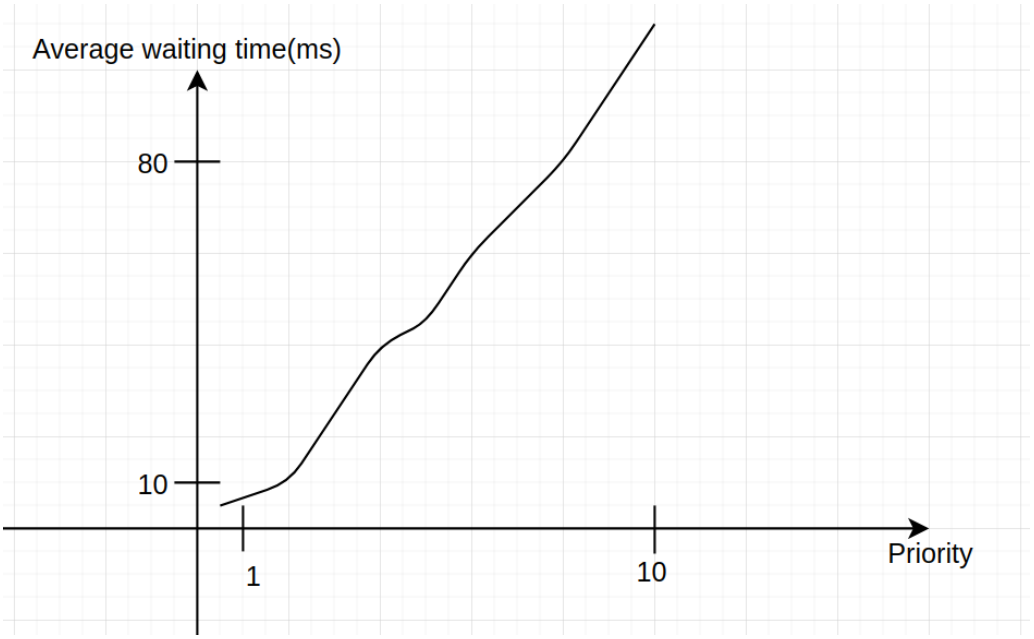
Пояснення: при збільшенні інтенсивності - у черці збільшується кількість заявок, що очікують на виконання. Отже і збільшується час, за який процессор проходить кільце заявок. Графік не лінійний, бо у нас алгоритм без витіснення, а це означає що процессор буде витрачати на кожну нову заявку не лінійну кількість часу.

2) **залежність проценту простою ресурсу від інтенсивності вхідного потоку заявок**



Пояснення: при збільшенні інтенсивності - у черці збільшується кількість заявок, що очікують на виконання. Отже процессору треба більше часу, щоб виконати всі заявки з кільця, і простою виникає менше, бо процессор не встигає опрацьовувати всі заявки.

3) **побудувати графік залежності середнього часу очікування від пріоритету при фіксованій інтенсивності вхідного потоку заявок**



(Заявки з нижчим значенням пріоритету - опрацьовуються першочергово)

Пояснення: більш пріоритетні задачі опрацьовуються першочергово, для них час очікування менший. При зменшенні пріоритету - час збільшується, бо процессор зайнятий більш пріоритетними заявками.

Опис переваг на недоліків Round Robin
Переваги:

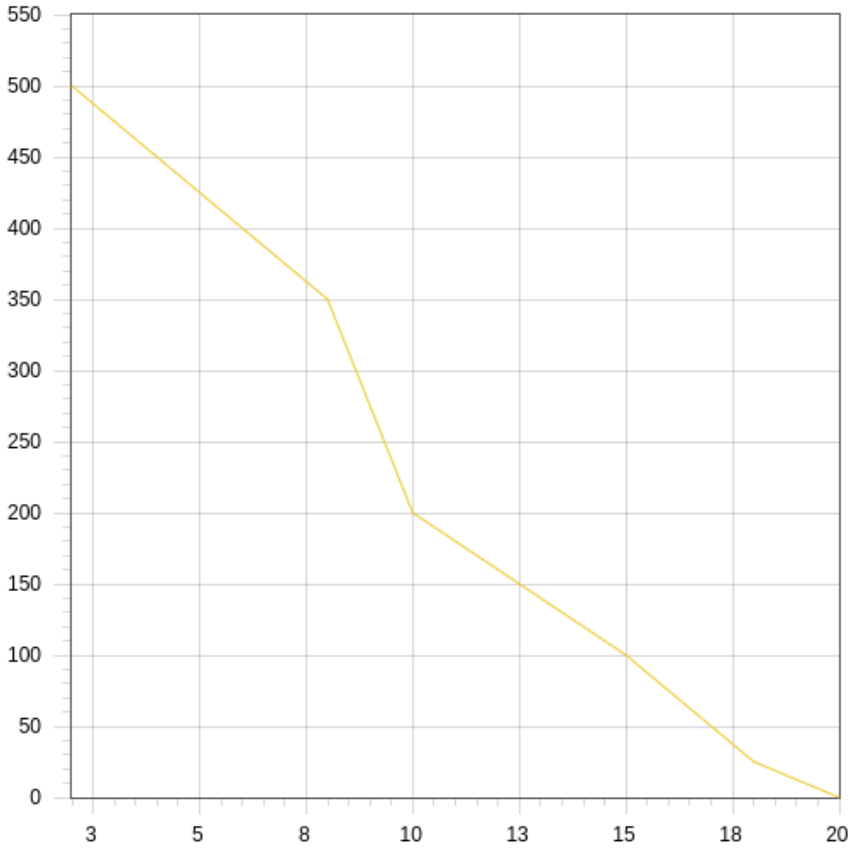
- Усі процеси отримують ресурси процесора
- Ефективність конфігурується квантом часу відповідно до умов
- Використовує базові структури даних

Недоліки:

- Чим вищий квант часу - тим вище час відповіді(час для повного виконання процесу)
- Чим нижчий квант часу - тим більше ресурсів процессора використовується на переключення між контекстами процесів

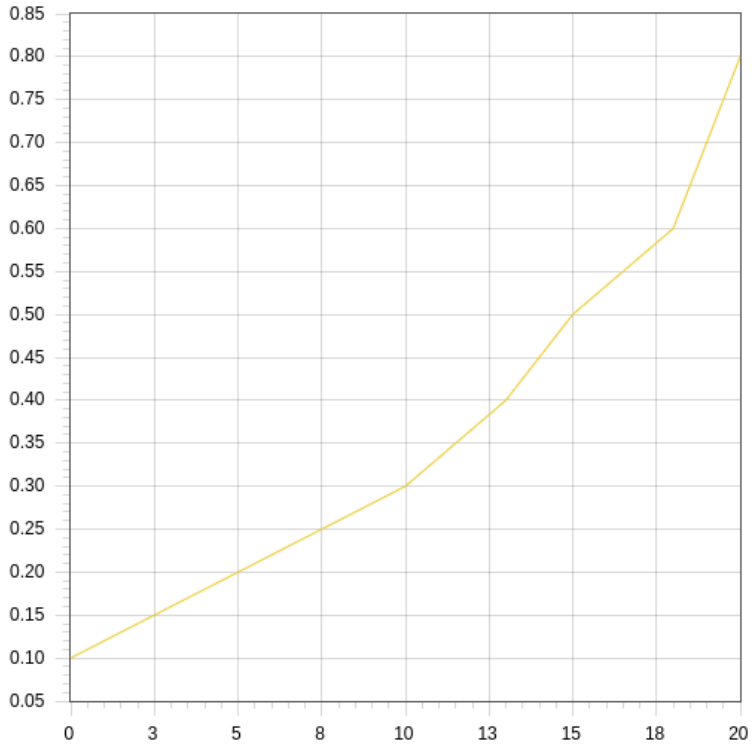
Результат для MFQS:

1) **графік залежності середнього часу очікування від інтенсивності вхідного потоку заявок**



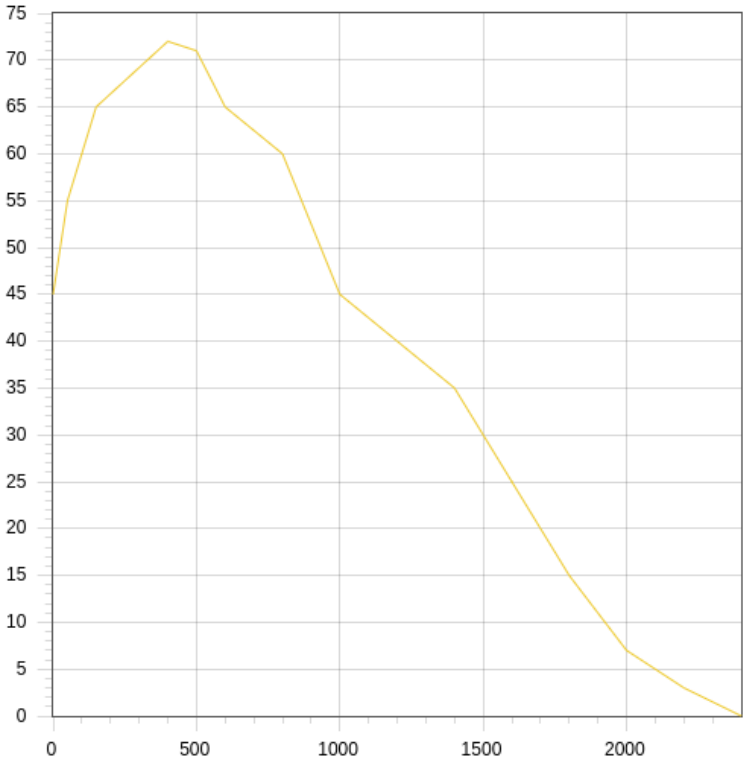
Пояснення: Ось x - інтервал між заявками; ось y - середній час очікування на виконання заявки. При збільшенні інтервалу - процессор частіше підходить до обробки задач з 2 до N черг, відповідно час очікування стрімко зменшується

2) **залежність проценту простою ресурсу від інтенсивності вхідного потоку заявок**



Пояснення: Ось x - інтервал між заявками; ось y - середній час очікування на виконання заявки. При зменшенні інтервалу - у чергах зменшується кількість заявок, що очікують на виконання. Отже процессору треба менше часу, щоб виконати всі заявки з кожного кільця черги, і простою виникає менше, заявки менше часу затримуються у чергах.

3) **графік залежності кількості заявок від часу очікування при фіксованій інтенсивності вхідного потоку заявок**



Пояснення: Ось x - час; ось y - середня кількість задач. На початку графіку ми бачимо зростання кількості задач, бо на цьому етапі процессор витрачає час на довгі заявки(які не виконаються за час першої черзі та перейдуть до наступних). Тому цей пік зумовлений довгими заявками, які чекають на обробку у другій та наступних чергах. Потім поступово кількість заявок знижується, коли процессор переходить до виконання заявок з другої та наступних черг.

Опис переваг на недоліків MFQS

Переваги:

- універсальність, поєднує властивості інших стратегій планування
- забезпечує швидку відповідь для коротких заявок
- не відбувається старіння(starvation), коли завки з високим пріоритетом заблоковані через заявки з нижчим пріоритетом.
- має багато варіантів для модифікацій реалізації

Недоліки:

- складність алгоритму, потребує визначення багатьох параметрів для забезпечення найкращої ефективності для конкретної задачі

- при дуже високому вхідному потоці - заявки, які потребують багато часу для обробки блокуються швидкими заявками. (Якщо немає механізму збільшення пріоритету для заявок що довго не виконуються)

- великі затрати CPU на планування

Висновки

Під час виконання розрахунково-графічної роботи я ознайомився з теорією планування виконання завдань та систем масового обслуговування. У результаті роботи було створено планувальник з 2 доступними дисциплінами: Round Robin та Multilevel Feedback Queue Scheduling.

Додаток (Код програми)

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

void findWaitingTime(int processes[], int n,
                int bt[], int wt[], int quantum)
{

     int rem_bt[n];
     for (int i = 0 ; i < n ; i++)
          rem_bt[i] = bt[i];

     int t = 0; // Current time

     while (1)
     {
          bool done = true;
          for (int i = 0 ; i < n; i++)
          {
               if (rem_bt[i] > 0)
               {
                    done = false;
                    if (rem_bt[i] > quantum)
                    {
                         t += quantum;

                         rem_bt[i] -= quantum;
                    }
                    else
                    {
                         t = t + rem_bt[i];

                         wt[i] = t - bt[i];

                         rem_bt[i] = 0;
```

```
                    }
                }
            }

            if (done == true)
            break;
        }
}

void findTurnAroundTime(int processes[], int n,
                                int bt[], int wt[], int tat[])
{
        for (int i = 0; i < n ; i++)
            tat[i] = bt[i] + wt[i];
}

void findavgTime(int processes[], int n, int bt[],
                                                int quantum)
{
        int wt[n], tat[n], total_wt = 0, total_tat = 0;

        findWaitingTime(processes, n, bt, wt, quantum);

        findTurnAroundTime(processes, n, bt, wt, tat);

        cout << "Processes "<< " Burst time "
            << " Waiting time " << " Turn around time\n";

        for (int i=0; i<n; i++)
        {
            total_wt = total_wt + wt[i];
            total_tat = total_tat + tat[i];
            cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
                << wt[i] <<"\t\t " << tat[i] <<endl;
        }

        cout << "Average waiting time = "
            << (float)total_wt / (float)n;
        cout << "\nAverage turn around time = "
            << (float)total_tat / (float)n;
}

int main()
{
```

```
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];

    int burst_time[] = {10, 5, 8};

    int quantum = 2;
    findavgTime(processes, n, burst_time, quantum);
    return 0;
}

#include<stdio.h>
#include<iostream>
#include<queue>

using namespace std;

#define MAX 1000
#define QUANT 4

int
flag[MAX],at[MAX],bt[MAX],pt[MAX],rt[MAX],ft[MAX],fe[MAX],fe_f
lag[MAX],pid[MAX],tms,qt[MAX];

queue<int> q;  //RR queue

void RR()
{
     if(!q.empty())
     {
         if(rt[q.front()]>0 && qt[q.front()]<4)
         {
                  qt[q.front()]++;
                  rt[q.front()]--;
                  if(rt[q.front()]==0)
                  {
              ft[q.front()]=tms+1;
              q.pop();
                  }
                  if(rt[q.front()]!=0 && qt[q.front()]==4)
                  {
                  qt[q.front()]=0;
                  q.push(q.front());
                  q.pop();
                  }
```

```
            }
        }
}

int main()
{
    int i=0,n=0,smallest=0,last_smallest=-1,min,sum=0,large=0;
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
          scanf("%d %d %d %d",&pid[i],&at[i],&bt[i],&pt[i]);
          if(at[i]>large)
               large=at[i];
            sum+=bt[i];
            rt[i]=bt[i];
    }
    min=MAX;
    for(tms=0;!q.empty() || tms<=sum+large ;tms++)
    {
      min=MAX;
      smallest=-1;
      for(i=0;i<n;i++)
      {
          if(at[i]<=tms && pt[i]<min && rt[i]>0 && !flag[i])
          {
              min=pt[i];
                   smallest=i;
          }
      }
      if(smallest==-1 && !q.empty())
      {
          if(last_smallest !=-1 && rt[last_smallest]==0)
          {
              ft[last_smallest]=tms;
                   flag[last_smallest]=1;
          }
          last_smallest=-1;
          RR();
          continue;
      }
      else if(smallest!=-1 && !q.empty() && last_smallest==-1)
      {
          if(qt[q.front()]<=4 && qt[q.front()]>0)
          {
```

```
            q.push(q.front());
            q.pop();
        }
    }
    if(smallest!=-1 && !fe_flag[smallest])
    {
        fe[smallest]=tms-at[smallest];
        fe_flag[smallest]=1;
    }
       if(  smallest!=last_smallest  &&  last_smallest!=-1  &&
!flag[last_smallest])
    {
        q.push(last_smallest);
        flag[last_smallest]=1;
    }
    if(smallest !=-1)
        rt[smallest]--;
```

```
               if((smallest   !=-1)   &&   ((rt[smallest]==0)
||(bt[smallest]-rt[smallest])==QUANTA))
    {
        if((bt[smallest]-rt[smallest])==QUANTA            &&
rt[smallest]!=0)
        {
            flag[smallest]=1;
            q.push(smallest);
        }
        else if(smallest!=-1)
        {
                ft[smallest]=tms+1;
                flag[smallest]=1;
        }
    }
    last_smallest=smallest;
  }
 for(int i=0;i<n;i++)
                cout<<pid[i]<<"    "<<fe[i]<<"    "<<ft[i]<<"
"<<ft[i]-bt[i]-at[i]<<endl;
  return 0;
}
```