6/8/2019

EXPERIMENT - 1
CPU SCHEDULING

1) AIM :-

Simulate the following non-preemptive CPU scheduling algorithms to find the turn-around time and waiting time.

a) FCFS b) SJF c) RR (preemptive)

d) Priority

2) ALGORITHM :-

a) FCFS :-

Step 1 :- Input the processes along with their burst times (bt)

Step 2 :- Find the waiting time for all processes (wt)

Step 3 :- First process that comes need not wait so waiting time for the first process is zero. (wt[0] = 0)

Step 4 :- Find the waiting time for all other processes

For process i,

$$wt[i] = bt[i-1] + wt[i-1]$$

Step 5 :- Find turnaround time

turnaround = waiting + burst

OUTPUT:-

FCFS:-

Enter the number of processes : 5

Enter the burst and arrival time : 5 0

16 0

9 0

4 0

3 0

SI	Bst	Arr	Turn	Wait
1	5	0	5ms	0ms
2	16	0	21ms	5ms
3	9	0	30ms	21ms
4	4	0	34ms	30ms
5	3	0	37ms	34ms

SJF:-

Enter the number of processes : 4

Enter the burst and arrival time: 4 0

12 0

5 0

2 0

SI	Bst	Arr	Turn	Wait
4	2	0	2ms	0ms
1	4	0	6ms	2ms
3	5	0	11ms	6ms
2	12	0	23ms	11ms

Step 6:- Print the turnaround time, waiting time for each process in the order of arrival.

b) SJF:-

Step 1:- Input the process along with their burst times

Step 2:- Sort the processes in the increasing order of burst time.

Step 3:- Find the waiting time for all the process

Step 3.1:- First process will have waiting time 0 ie wt[0]=0

Step 3.2:- For all other process i,

wt[i] = bt[i-1] + wt[i-1]

Step 4:- Find turnaround time

turnaround = waiting + burst

Step 5:- Print the turnaround time, waiting time for each process in increasing order of burst time.

c) Round Robin:-

Step 1:- Input the processes along with their burst times.

RR:

Enter the number of processes: 4

Enter burst time and arrival time: 4 0

12 0

5 0

2 0

Enter time slice: 2

SI	Bst	Arr	Wait	Turn
1	4	0	6ms	10ms
2	12	0	11ms	23ms
3	5	0	12ms	17ms
4	2	0	6ms	8ms

PRIORITY:

Enter number of processes: 4

Enter burst and arrival time: 12 0

3 0

5 0

9 0

Enter priority: 1 5 3 4

SI	Bst	Arr	Prio	Turn	Wait
2	12	0	5	12ms	0ms
4	3	0	4	15ms	12ms
3	5	0	3	20ms	15ms
1	9	0	1	29ms	20ms

Step 2 :- Create an array rem_bt[] to track the remaining burst time which is a copy of the values of burst time.

Step 3 :- Create an array wt[] to store the waiting times of processes with initial value of 0.

Step 4 :- Initialize t=0

Step 5 :- Traverse all the processes until they are finished ie

```
If rem_bt[i] > quantum
    t = t + quantum
    bt_rem[i] = quantum
Else
    t = t + bt_rem[i]
    wt[i] = t - bt[i]
    bt_rem[i] = 0
```

d) PRIORITY :-

Step 1 :- Input the process with burst times and priorities

Step 2 :- Sort the process (with burst times) according to the priority.

Step 3 :- Find the waiting times of all processes

Step 3.1 :- First process have waiting time 0. ie wt[0]=0

Step 3.2 :- For all other process. say i
wt[i] = bt[i-1] + wt[i-1]

Step 4 :- Find the turnaround time
turnaround = waiting + burst

Step 5 :- Print the waiting time and turnaround time of all process according to priority.

a) SINGLE-LEVEL

Enter name of directory - D1
1. Create File 2. Delete File 3. Search File
4. Display File 5. Exit
Enter choice - 1
Enter name - F1
Enter choice - 1
Enter name - F2
Enter choice - 4
The Files are - F1 F2
Enter choice - 2
Enter name - F1
File F1 is deleted
Enter choice - 4
The Files are - F2
Enter choice - 3
Enter name - F1
File F1 not Found
Enter choice - 5

EXPERIMENT-2
FILE ORGANIZATION

1) AIM:-

Simulate the following file organization techniques

a) Single-level directory b) Two level directory c) Hierarchical

2) ALGORITHM:-

a) Single-level Directory

It is the simplest of all directory structures. There is only one directory containing all the files.

Algorithm:-

Step1:- Input the number of files, say N

Step2:- while (N!=0)

Step 2.1:- Input filename.

Step 2.2:- Decrement N

Step3:- Display the file structure.

b) Two-level Directory

It contains a root directory, under which each user has a private directory.

6) 1) Create Directory 2) Create File 3) Delete File
4) Search File 5) Display 6) Exit

Enter choice - 1
Enter name of Directory - D1
Directory created

Enter choice - 2
Enter name of Directory - D2
Directory created.

Enter choice - 5
Directory Files
D1
D2

Enter choice - 2
Enter directory name - D1
Enter name of file - F1
File created
Enter choice - 5
Directory Files
D1 F1
D2

Enter choice - 4
Enter name of directory - D1
Enter name of file - F1
File F1 is found.

Step 1:- Initialize the root directory
Step 2:- Input the number of users, and the number of files they have.
Step 3:- Initialize the user directory and add it to the tree structure
Step 4:- Add files of each user to each user directory.

c) Hierarchical.

It allows subdirectories to be created within user directories and itself.

Step 1:- Initialize the root directory
Step 2:- Input the number of users, The number of subdirectories/files they have.
Step 3:- Initialize each user directory and their subdirectories and files
Step 4:- Display as a tree structure.

Enter choice - 3
Enter name of directory - D1
Enter name of file - F1
File F1 is deleted
Enter choice - 5
Directory Files
D1
D2
Enter choice - 6

c) HIERARCHICAL

You are in root
1) List everything 2) change directory 3) go to parent directory 4) add new file 5) delete file 6) create new directory 7) delete directory 8) exit.

Enter choice : 6
Enter name : D1

Enter choice : 4
Enter name : F1

Enter choice : 2
Enter name : D1
Directory changed

Enter choice : 3
Directory changed.

Enter choice : 1

DI F1

Enter choice : 5

Enter name of directory or file to delete : F8

Successfully deleted

Enter choice : 1

Empty directory

Enter choice : 3

Enter choice : 7

Enter name of file or directory to delete : DI

Successfully deleted.

Enter choice : 1

F1

Enter choice : 8

Exit

EXPERIMENT-3
BANKER'S ALGORITHM

Aim:-

Implement the bankers algorithm for deadlock.

ALGORITHM:-

a) Saftey Algorithm:-

Step 1:- Let work and finish be arrays of length m and n, respectively
Initialize:
work = Available
Finish[i] = false for i = 0, 1, ..., n-1

Step 2:- Find i such that both
Finish[i] = false
Need[i] ≤ work
if no such i exists, go to step 4

Step 3:- work = work + Allocation[i]
Finish[i] = true
Go to step 2

Step 4:- If Finish[i] == true for all i, then the system is in safe state

OUTPUT:-

```
Enter the number of processes and resources: 5
Enter the resource allocation table :
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the number of resources available of
Type 1 : 10
Enter the number of resources available of
Tye 2 : 5
Enter the number of resources available of
Type 3 : 7
Enter the maximum resource table
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

System is in safe state
Order is : P2 - P4 - P5 - P1 - P3
```

b) Resource Request Algorithm

Step 1:- If Request[i] ≤ Need[i], go to Step 2
Else raise an error

Step 2:- If Request[i] ≤ Available, go to step 3
Otherwise P[i] must wait

Step 3: Available[i] = Available - Request[i]
Allocation[i] = Allocation[i] + Request[i]
Need[i] = Need[i] - Request[i];

OUTPUT:-

```
Enter buffer size : 1
Enter your choice :- 1
1. Produce  2. Consume  3. Exit
Your choice : 1
Produced Item..
Produced : 1  Consumed : 0
Your Choice : 1
Buffer is full!!!
Produced : 1  Consumed : 0
Your choice : 2
Consumed Item
Produced : 1  Consumed : 1
Your choice : 2
Buffer empty!!!
Produced : 1  Consumed : 1
Your Choice : 3
```

EXPERIMENT-4
PRODUCER - CONSUMER PROBLEM

1) AIM:-

To implement the producer-consumer problem using semaphores.

2) ALGORITHM:-

The algorithm uses three semaphores, two counting semaphores full and empty and a binary semaphore mutex.

Step 1:- Enter the size of buffer say n.
Step 2:- Initialize full = 0 and empty = n, mutex = 1
Step 3:- Input operation to be performed i.e produce or consume or exit
Step 4:- If operation is produce
  Step 4.1:- If mutex = 1 and buffer not full
  begin
    Step 4.1.1:- mutex = mutex - 1
    Step 4.1.2:- full = full + 1
    Step 4.1.3:- empty = empty - 1
    Step 4.1.4:- print "Item produced"
    Step 4.1.5:- mutex = mutex + 1
  end
  Step 4.2:- else print "Buffer Full"

Step 5:- If operation is consume.
  Step 5.1:- If mutex = 1 and buffer not empty
  begin
    Step 5.1.1:- mutex = mutex - 1
    Step 5.1.2:- ~~signal = signal + 1~~

Step 5-1-2 :- full = full-1
Step 5-1-3 :- empty = empty+1
Step 5-1-4 :- print "Consumed Item"
Step 5-1-5 :- mutex = mutex+1
end
Step 5-2 :- else print "Buffer Empty"
Step 6 :- Repeat steps 3, 4, 5 until exit

OUTPUT :

```
Enter philosopher number (1-5) : 1
Enter operation
1. start Eating  2: Stop Eating  3. Exit
1
Philosopher1 is eating
Enter philosopher number (1-5) : 2
Enter operation
1
Philosopher 2 cannot eat
Enter philosopher number (1-5) : 4
Enter operation
1
Philosopher 4 is eating
Enter philosopher number (1-5) : 1
Enter operation
2
Philosopher 1 is thinking
Enter philosopher number (1-5) : 2
Enter operation
1
Philosopher 2 is eating
```

EXPERIMENT - 5
DINING PHILOSOPHER'S PROBLEM

1) AIM :-

To simulate the working of dining philosopher's problem.

2) ALGORITHM :-

Step 1 :- Declare set of states THINKING, HUNGRY, EATING for philosophers and flag values and initialize each philosopher to think.

Step 2 :- Input philosopher's id/number.

Step 3 :- Enter the operation to be performed ie Eating, Stop Eating, Exit

Step 4 :- If operation is Eating

Step 4.1 :- Set state as HUNGRY.

Step 4.2 :- If the philosphers to left and to right are not EATING

Step 4.2.1 :- Set state as EATING

Step 4.2.2 :- Set flag as 1.

Step 4.3 :- If flag is set to 1

Step 4.3.1 :- Output philosphes is eating

Step 4.3.2 :- else output cannot eat.

Step 5 :- If operation is Stop eating

Step 5.1 :- Set philosophes state as THINKING

Step 5-2 :- Test for states of philosophers to the left and right

Step 5-3 :- print philsophier is THINKING

Step 6 :- Repeat 3, 4, 5 until exit

OUTPUT:

```
Enter the range: 200
Enter the number of disk requests: 3
Enter the head position (<200): 60
Enter the requests: 50 70 80
Enter your choice
1. FCFS  2. SCAN  3. C-SCAN  4. EXIT
1
No    Disk Req
0     60
1     50
2     70
3     80
Seek time : 40ms
Enter your choice
1. FCFS  2. SCAN  3. C-SCAN  4. EXIT
2
No    Disk Req
0     60
1     50
2     0
3     70
4     80
Seek Time : 140ms
Enter your choice
1. FCFS  2. SCAN  3. C-SCAN  4. EXIT
```

EXPERIMENT - 6
DISK SCHEDULING

1) AIM:-

To simulate disk scheduling algorithms
i) FCFS ii) SCAN iii) C-SCAN

2) ALGORITHM:-

i) FCFS:-

1) Start
2) input no. of requests, say n
3) for i = 2 to n+2
   3.1) input each request, say t[i]
4) for i = 1 to n+1
   4.1) totim[i] = t[i+1] - t[i]
   4.2) if (totim<0)
      4.2.1) totim[i] = totim[j] * -1;
5) for i=1 to n+1
   5.1) tott = totim[i]
6) avtim = (float)[tott/n]
7) for i=0 to n
   7.1) print t[i], totim[i], avtim
8) Stop

3

No	Disk Req
0	60
1	50
2	0
3	199
4	70

Seek time : 270ms

Enter choice

1. FCFS 2. SCAN 3. C-SCAN 4. EXIT

4

Thus

SCAN:-

1) Start
2) Enter no of inputs, n
3) Enter head position, h
4) initialize t[0]=0 & t[1]=h
5) for i=2 to n+2
   5.1) input each request, t[i]
6) for i=0 to n+2
   6.1) for j=0 to (n+2)-i-1
      6.1.1) if (t[j]>t[j+1])
         6.1.1.1) swap values, t[j], t[j+1]
7) for i=0 to n+2
   7.1) if t[i]==h
      j=i, k=i
8) for p=k+1 to 2
   8.1) atr[p]=t[k+1]
9) for j=0 to n
   9.1) if (atr[j]>atr[j+1])
      9.2) d[j]=atr[j]-atr[j+1]
      else
      9.3) d[j]=atr[j+1]-atr[j]
10) print average movement (float) sum/n.

C-SCAN:

1) Start
2) No. of requests, n
3) Head position say h
4) Head tracks, tot
5) for i = 3 to n+2
    5.1) input disk req, t[i]
    5.2) for j = 0 to n+2-i-1
        5.2.1) if (t[i] > t[j+1])
            5.2.1.1) swap values, t[j], t[j+1]
6) for i = 0 to n+2
    6.1) if (t[i] = h)
        6.1.1) j = i, break.
7) p = 0
8) while (t[j] != pt-1)
    8.1) atr[p] = t[j]
    8.2) j++, p++
9) while (p != (n+3) && t[i] != t[h])
    9.1) atr[p] = t[i]
10) for j = 0 to n+2
    10.1) if (atr[j] > atr[j+1])
        10.1.1) d[j] = atr[j] - atr[j+1]
        else
        10.1.2) d[j] = atr[j+1] - atr[j]

(0.1.3) sum += d[j]

11) printf(sum, sum/n.

INPUT:

COPY	START	1000
-	LDA	ALPHA
-	ADD	ONE
-	SUB	TWO
-	STA	BETA
ALPHA	BYTE	C'KLNCE
ONE	RESB	2
TWO	WORD	5
BETA	RESW	1
-	END	-

OUTPUT:

	COPY	START	1000
1000	-	LDA	ALPHA
1003	-	ADD	ONE
1006	-	SUB	TWO
1009	-	STA	BETA
1012	ALPHA	BYTE	C'KLNCE
1017	ONE	RESB	2
1019	TWO	WORD	5
1022	BETA	RESW	1
1025	-	END	-

Program Length = 25

EXPERIMENT-7
TWO PASS ASSEMBLER

1) AIM:-

To implement pass 1 and pass 2 of a 2-pass assembler by creating a symbol table with suitable hashing.

2) ALGORITHM:-

Pass 1:

```
begin
  read first input line
  if OPCODE = 'START' then
  begin
     save #[OPERAND] as starting address
     initialize LOCCTR to starting address
     write to intermediate file
     read next input line
  end (if START)
else
  initialize LOCCTR to 0
while OPCODE ≠ 'END' do
  begin
     if this is not a comment line then
        begin
        if there is a symbol in LABEL field then
           begin
              search SYMTAB for LABEL
              if found then
                 set error flag (duplicate symbol)
              else
                 insert {LABEL, LOCCTR} into SYMTAB
           end (if symbol)
```

Pass 2:

H^COPY^1000^25
T^001000^001012^0[illegible]1017^051019^2310221
7576786769^00005^

```
search OPTAB for OPCODE
if found then
    add 3 to LOCCTR
else if OPCODE = 'RESW' then
  add 3 * #{OPERAND} to LOCCTR
else if OPCODE = 'WORD' then
    add 3 to LOCCTR
else if OPCODE = 'RESB' then
    add #{OPERAND} to LOCCTR
else if OPCODE = 'BYTE' then
begin
    find length of constant in bytes
    add length to LOCCTR
end (if BYTE)
else
    set error flag (invalid opcode)
end (if not a comment)
write line to intermediate file
read next input line
end (while not END)
write last line to intermediate file
save {LOCCTR - starting address} as program length
end (pass 1)
```

```
Pass 2:
 begin
    read first input line (from intermediate file)
    if OPCODE = 'START' then
        begin
            write listing line
            read next input line
        end (if START)
    write Header record to object program
    initialize first Text Record
    while OPCODE ≠ 'END' do
        begin
            if this is not a comment line then
                begin
                    search OPTAB for OPCODE
                        if found then
                            begin
                                if there is a symbol
                            in operand field
                                then
                                begin
                                    search SYMTAB for OPERAND
                                     if found then
                                         store symbol value
                                         as operand address
                                    else
                                      begin
                                         store 0 as operand
                                          address
                                         set error flag
                                      end.
                                end (if symbol)
```

```
        else
            store 0 as operand address
            assemble the object code instruction
        end (if opcode found)
        else if OPCODE = 'BYTE' or 'WORD' then
            convert constant to object code
        if object code will not fit in current
TEXT Record then
        begin
            write TEXT record to the object program
            initialize new Text Record
        end
        add object code to Text Record
    end (if not comment)
    write listing line
    read next input line
    end (while not END)
write last Text Record to object program
write END record to object program
write last listing line
end (Pass 2)
```

4/11/19

INPUT:

```
H COPY  001000  00107A
T 001000 1E 141033 482039 001036 281030
301015 482061 3C1003 00102A 0C1039 001
T 00101E 15 0C1036 482061 081033 4C0000
454F46 000003 000000
        :
E 001000
```

OUTPUT

MEMORY ADDRESS	CONTENTS			
1000	141033 48	20390010	36281030	301015
1010	20613C10	0300102A	0C103900	102D0
1040	XXXXXXXX	XXXXXX04	10300010	30E0
1090	20792C10	36		

EXPERIMENT-8

ABSOLUTE LOADER

1) Aim:-

To implement an absolute loader

2) ALGORITHM:-

```
begin
    read Header Record
    verify program name and length
    read & list Text Record.
    while record type ≠ 'E' do
    begin
        {if object code is in character form, convert
         into internal representation}
        move object code to specified location in
        memory
        read next object program record.
    end
    jump to address specified in End Record
end.
```

4/11/17

INPUT :-

```
COPY   START  1000
-      LDA    ALPHA
-      STA    BETA
ALPHA  RESW   1
BETA   RESW   1
-      END    -
```

OUTPUT

```
H^COPY^1000^0C
T^001000^0C^000000^23000
T^1001^02^1006
T^1004^02^1009
E^001000
```

EXPERIMENT-9
SINGLE PASS ASSEMBLER

1) Aim:-

To implement a single pass assembler

2) ALGORITHM:-

```
begin
 if starting address is given then
     LOCCTR = starting address
 else
     LOCCTR=0
 while OPCODE!=END do
 begin
    read a line from the code
    if there is a label then
    begin
      if this label is in SYMTAB then
        error
      else
        insert (label, LOCCTR) into SYMTAB
    end
   search OPTAB for the OPCODE
    if found then
        LOCCTR+=N
    else if this is an assembly directive then
        update LOCCTR as directed
    else
        error
    write line to intermediate file
 end
 program size=LOCCTR- starting address
end.
```

4/11/19

INPUT:

```
EX1  MACRO &A,&B
-    LDA   &A
-    STA   &B
-    MEND  -
SAMPLE START 1000
-    EX1  N1,N2
N1   RESW  1
N2   RESW  1
-    END   -
```

OUTPUT

```
SAMPLE  START  1000
  -       EX1    N1,N2
  -       LDA    N1
  -       STA    N2
 N1       RESW   1
 N2       RESW   1
  -        END   -
```

EXPERIMENT - 10
MACRO PROCESSOR

1) Aim:-

To implement a two pass macro processor

2) ALGORITHM:-

```
Pass 1:
begin
    read first input line
    while OPCODE != 'MEND' do
    begin
        if OPCODE = 'MACRO' then
        begin
            insert label & macro name into
            NAMTAB
            insert label, operand into DEFTAB
        end
        else
            insert opcode, operand into DEFTAB
        read next input line
    end
    insert opcode into DEFTAB
end

Pass 2:
begin
    read first input line
    while OPCODE != 'END' do
    begin
        if OPCODE = 'MACRO' then
        begin
            read next input line
            while OPCODE != 'MEND' do
            begin
```

```
        read next input line
      end
    end
    else
      begin
        search NAMTAB for OPCODE
        if found then
        begin
          insert operand into ARGTAB
        read next row of DEFTAB
        while OPCODE != 'MEND' do
        begin
          if OPERAND[0] = '&' then
            search ARGTAB for argument
          end
        read next row of DEFTAB
        end
        end
      end
    read next input line
  end
end.
```

Symbol Table menu.
1. Create a Symbol Table.
2. Search in the Symbol Table.
3. Exit
Enter your choice: 1
Enter address: 567
Enter label: abc
Do you want to continue? (Y/N)? y
Enter address: 678
Enter label: ghi
Do you want to continue? (y/n): y
Enter address: 123
Enter label: hij
Do you want to continue? (y/n) n

Symbol Table

Hash	Values	Label
0	0	
1	0	
2	123	hij
3	0	
4	0	
5	0	
6	567	abc
7	678	ghi
8	0	
9	0	
10	0	

EXPERIMENT - 11

SYMBOL TABLE WITH HASHING

1) AIM:-

To implement a symbol table with suitable hashing.

ALGORITHM:-

```
Set MAX = 11
create structure symb with add and label[]
declare structure variable sy[]
void search()
{ open file symbol.txt onto fp1 with mode
  read la.
  for(i=0; i<MAX; i++)
  { Read j and sy[i].add from fp1
    if (sy[i].add != 0)
    Read sy[i].label from fp1
  }
for(i=0; i<MAX; i++)
{ if(sy[i].add != 0)
  { if( strcmp[sy[i].label, la) == 0)
    { set = 1;
      s = sy[i].add;
    }
  }
}
if(set == 1)
    Display label is present
else
    Display label is not present
}
void display(int a[MAX])
```

Enter your choice : 2
Enter label : bij
The label is not present in the Symbol Table

```
{ Open file symbol.txt into fp with wb mode
  Display the symbol table.
  for(i=0; i<MAX; i++)
      write i, sy[i].add, sy[i].label onto
      fp
 }
int create(int num)
{ int key;
    key = num % 11;
   return key;
}
void hprob(int a[MAX], int key, int num)
{ Declare flag=0, count=0
   if (a[key] == 0)
   { a[key] = num;
      sy[key].add = num;
      strcpy(sy[key].label, l);
   }
else
 { i=0;
   while(i<MAX)
   { if (a[i] != 0
         count++;
         i++;
   }
 if(count == MAX)
 { Display Hash table is Full
   display(a)
 }
 exit();
```

```
for (i=key+1; i<MAX; i++){
  if(a[i]==0)
  {  a[i]=num;
     flag=1;
     sy[key].add=num;
     strcpy( sy[key].label,l);
     break;
  }
for(i=0;i<key && flag==0;i++){
   if(a[i]==0)
   {  a[i]= num;
      flag=1;
      sy[key].add=num;
      strcpy(sy[key].label,l);
      break;
   }
  }
}
void main()
{  Declare a[MAX], num, key, ch, ans='y';
   for(i=0;i<MAX;i++)
        a[i]=0;
   do
   {  Read ch
      switch(ch)
      {  case 1: while(ans=='y')
                 { Read num
                   key = create(num);
                   Read l
                   lprob(a, key, num);
```

```
Read ans
  }
display (a)
break;
case 2: search()
        break
case 3: exit(0)
}
} while (ch<=3);
}
```