Shell scripting

Palindrome

```
echo "enter the number"
read n
function pal
{
number=$n
reverse=0
while [ $n -gt 0 ]
do
a=` expr $n % 10 `
n=` expr $n / 10 `
reverse=` expr $reverse \* 10 + $a`
done
echo $reverse
if [ $number -eq $reverse ]
then
        echo "number is palindrome"
else
        echo "number is not palindrome"
fi
}
r=` pal $n`
echo "$r"
```

Factorial

```
1 echo "Enter number : "
2 read n
3 m=$n
4 fact=1
5 while [ $n -gt 0 ]
6 do
7 fact=`expr $fact \* $n`
8 n=`expr $n - 1`
9 done
10 echo "Factorial of $m is $fact"
```

Occurrence of letter

```
read s
read c
len=${#s}
ans=0
for((i=0;i<len;i++))
do
        if[[${s:$i:1} == ${c:0:1} ]]
        then
                ans=$(($ans+1))
        fi
done
echo "$c appeared in string $s $ans times"
```

Codes

Banker's Algorithm:

```
#include <stdio.h>
int main()
{
   int n, m, i, j, k;
   n = 5;
   m = 3;
   int alloc[5][3] = {{1, 1, 0}, {2, 0, 1}, {3, 1, 2}, {0, 1, 3}, {0, 0, 1}};
   int max[5][3] = {{2, 1, 3}, {2, 1, 1}, {3, 0, 4}, {1, 1, 1}, {0, 2, 3}};
   int avail[3] = {3, 3, 2};
   int f[n], ans[n], ind = 0;
   for (k=0; k<n; k++)
   {
      f[k] = 0;
   }
   int need[n][m];
   for(i=0; i<n; i++)
   {
      for(j=0; j<m; j++)
      {
         need[i][j] = max[i][j] - alloc[i][j];
      }
   }
   int y = 0;
   for(k=0; k<5; k++)
   {
      for(i=0; i<n; i++)
```

```
    {
      if(f[i]==0)
      {
        int flag = 0;
        for(j=0; j<m; j++)
        {
          if(need[i][j] > avail[j])
          {
            flag = 1;
            break;
          }
        }
        if(flag==0)
        {
          ans[ind++] = i;
          for(y=0; y<m; y++)
          {
            avail[y] += alloc[i][y];
          }
          f[i] = 1;
        }
      }
    }
  }
  int flag = 1;
  for(int i=0; i<n; i++)
  {
    if(f[i]==0)
    {
      flag = 0;
      printf("The following system is not safe");
      break;
    }
  }
  if(flag==1)
  {
    printf("The following is the safe Sequence\n");
    for(i=0; i<n-1; i++)
    {
      printf("P%d ->", ans[i]);
    }
    printf("P%d ->", ans[n-i]);
  }
  return (0);
}
```

FCFS:

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

void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
   wt[0] = 0;
   for (int  i = 1; i < n ; i++ )
   {
      wt[i] =  bt[i-1] + wt[i-1];
   }
}

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 wt[n], tat[n], total_wt = 0, total_tat = 0;
   findWaitingTime(processes, n, bt, wt);
   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<<endl;
   cout<<"\nAverage turn around time = "<<(float)total_tat / (float)n<<endl;
}

int main()
{
   int processes[] = { 1, 2, 3};
   int n = sizeof processes / sizeof processes[0];
   int  burst_time[] = {10, 5, 8};
   findavgTime(processes, n,  burst_time);
   return 0;
}
```

Round Robin:

```
#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;
   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 << "PN\t "<< " \tBT " << "  WT " << " \tTAT\n"<<endl;
   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<<endl;
   cout<<"\nAverage turn around time = "<<(float)total_tat / (float)n<<endl;
}

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;
}
```

SJF:

```
#include <bits/stdc++.h>
using namespace std;
//structure for every process
struct Process {
  int pid; // Process ID
  int bt; // Burst Time
  int art; // Arrival Time
};
void findTurnAroundTime(Process proc[], int n, int wt[], int tat[]) {
  for (int i = 0; i < n; i++)
  tat[i] = proc[i].bt + wt[i];
}
//waiting time of all process
void findWaitingTime(Process proc[], int n, int wt[]) {
  int rt[n];
  for (int i = 0; i < n; i++)
  rt[i] = proc[i].bt;
  int complete = 0, t = 0, minm = INT_MAX;
  int shortest = 0, finish_time;
```

```
  bool check = false;
  while (complete != n) {
    for (int j = 0; j < n; j++) {
      if ((proc[j].art <= t) && (rt[j] < minm) && rt[j] > 0) {
        minm = rt[j];
        shortest = j;
        check = true;
      }
    }
    if (check == false) {
      t++;
      continue;
    }
    // decrementing the remaining time
    rt[shortest]--;
    minm = rt[shortest];
    if (minm == 0)
      minm = INT_MAX;
      // If a process gets completely
      // executed
      if (rt[shortest] == 0) {
        complete++;
        check = false;
        finish_time = t + 1;
        // Calculate waiting time
        wt[shortest] = finish_time -
        proc[shortest].bt -
        proc[shortest].art;
        if (wt[shortest] < 0)
          wt[shortest] = 0;
      }
      // Increment time
      t++;
  }
}

void findavgTime(Process proc[], int n) {
  int wt[n], tat[n], total_wt = 0,
  total_tat = 0;
  // Function to find waiting time of all
  // processes
  findWaitingTime(proc, n, wt);
  // Function to find turn around time for
  // all processes
  findTurnAroundTime(proc, n, wt, tat);
  cout << "Processes " << " Burst time " << " Waiting time " << " Turn around time
";
  for (int i = 0; i < n; i++) {
```

```
    total_wt = total_wt + wt[i];
    total_tat = total_tat + tat[i];
    cout << " " << proc[i].pid << "\t\t" << proc[i].bt << "\t\t " << wt[i] << "\t\t " << tat[i] << endl;
  }
  cout << "
Average waiting time = " << (float)total_wt / (float)n; cout << "
Average turn around time = " << (float)total_tat / (float)n;
}
// main function
int main() {
  Process proc[] = { { 1, 5, 1 }, { 2, 3, 1 }, { 3, 6, 2 }, { 4, 5, 3 } };
  int n = sizeof(proc) / sizeof(proc[0]);
  findavgTime(proc, n);
  return 0;
}
```

Dining-philosophers:

```
#include<iostream>
#define n 4
using namespace std;

int compltedPhilo = 0,i;
struct fork{
int taken;
}ForkAvil[n];

struct philosp{
int left;
int right;
}Philostatus[n];

void goForDinner(int philID){ //same like threads concept here cases implemented
if(Philostatus[philID].left==10 && Philostatus[philID].right==10)
      cout<<"Philosopher "<<philID+1<<" completed his dinner\n";
//if already completed dinner
else if(Philostatus[philID].left==1 && Philostatus[philID].right==1){
        //if just taken two forks
        cout<<"Philosopher "<<philID+1<<" completed his dinner\n";

        Philostatus[philID].left = Philostatus[philID].right = 10; //remembering that he completed dinner
by assigning value 10
        int otherFork = philID-1;

        if(otherFork== -1)
          otherFork=(n-1);
```

```
        ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0; //releasing forks
        cout<<"Philosopher "<<philID+1<<" released fork "<<philID+1<<" and fork "<<otherFork+1<<"\n";
        compltedPhilo++;
      }
      else if(Philostatus[philID].left==1 && Philostatus[philID].right==0){ //left already taken, trying for
right fork
            if(philID==(n-1)){
               if(ForkAvil[philID].taken==0){ //KEY POINT OF THIS PROBLEM, THAT LAST PHILOSOPHER
TRYING IN reverse DIRECTION
                  ForkAvil[philID].taken = Philostatus[philID].right = 1;
                  cout<<"Fork "<<philID+1<<" taken by philosopher "<<philID+1<<"\n";
               }else{
                  cout<<"Philosopher "<<philID+1<<" is waiting for fork "<<philID+1<<"\n";
               }
            }else{ //except last philosopher case
               int dupphilID = philID;
               philID-=1;

               if(philID== -1)
                  philID=(n-1);

               if(ForkAvil[philID].taken == 0){
                  ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;
                  cout<<"Fork "<<philID+1<<" taken by Philosopher "<<dupphilID+1<<"\n";
               }else{
                  cout<<"Philosopher "<<dupphilID+1<<" is waiting for Fork "<<philID+1<<"\n";
               }
            }
         }
         else if(Philostatus[philID].left==0){ //nothing taken yet
               if(philID==(n-1)){
                  if(ForkAvil[philID-1].taken==0){ //KEY POINT OF THIS PROBLEM, THAT LAST PHILOSOPHER
TRYING IN reverse DIRECTION
                     ForkAvil[philID-1].taken = Philostatus[philID].left = 1;
                     cout<<"Fork "<<philID<<" taken by philosopher "<<philID+1<<"\n";
                  }else{
                     cout<<"Philosopher "<<philID+1<<" is waiting for fork "<<philID<<"\n";
                  }
               }else{ //except last philosopher case
                  if(ForkAvil[philID].taken == 0){
                     ForkAvil[philID].taken = Philostatus[philID].left = 1;
                     cout<<"Fork "<<philID+1<<" taken by Philosopher "<<philID+1<<"\n";
                  }else{
                     cout<<"Philosopher "<<philID+1<<" is waiting for Fork "<<philID+1<<"\n";
                  }
               }
      }else{}
```

```
}

int main(){
for(i=0;i<n;i++)
      ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n){
/* Observe here carefully, while loop will run until all philosophers complete dinner
Actually problem of deadlock occur only thy try to take at same time
This for loop will say that they are trying at same time. And remaining status will print by go for dinner
function
*/
for(i=0;i<n;i++)
         goForDinner(i);
cout<<"\nTill now num of philosophers completed dinner are "<<compltedPhilo<<"\n\n";
}

return 0;
}
```

FIFO:

```
// C++ implementation of FIFO page replacement
// in Operating Systems.
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using FIFO
int pageFaults(int pages[], int n, int capacity)
{
        unordered_set<int> s;

        // To store the pages in FIFO manner
        queue<int> indexes;

        // Start from initial page
        int page_faults = 0;
        for (int i=0; i<n; i++)
        {
                // Check if the set can hold more pages
                if (s.size() < capacity)
                {
                        // Insert it into set if not present
                        // already which represents page fault
                        if (s.find(pages[i])==s.end())
                        {
```

```
                        // Insert the current page into the set
                        s.insert(pages[i]);

                        // increment page fault
                        page_faults++;

                        // Push the current page into the queue
                        indexes.push(pages[i]);
                    }
                }

                // If the set is full then need to perform FIFO
                else
                {
                    // Check if current page is not already
                    // present in the set
                    if (s.find(pages[i]) == s.end())
                    {
                        // Store the first page in the
                        // queue to be used to find and
                        // erase the page from the set
                        int val = indexes.front();

                        // Pop the first page from the queue
                        indexes.pop();

                        // Remove the indexes page from the set
                        s.erase(val);

                        // insert the current page in the set
                        s.insert(pages[i]);

                        // push the current page into
                        // the queue
                        indexes.push(pages[i]);

                        // Increment page faults
                        page_faults++;
                    }
                }
            }
            return page_faults;
}

// Driver code
int main()
{
            int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
```

```
	int n = sizeof(pages)/sizeof(pages[0]);
	int capacity = 4;
	cout << pageFaults(pages, n, capacity);
	return 0;
}
```

Producer Consumer:

```
#include<bits/stdc++.h>
#include<pthread.h>
#include<semaphore.h>
#include <unistd.h>
using namespace std;

// Declaration
int r1,total_produced=0,total_consume=0;

// Semaphore declaration
sem_t notEmpty;

// Producer Section
void* produce(void *arg){
	while(1){
	cout<<"Producer produces item."<<endl;
	cout<<"Total produced = "<<++total_produced<<
		" Total consume = "<<total_consume*-1<<endl;
	sem_post(¬Empty);
	sleep(rand()%100*0.01);
	}
}

// Consumer Section
void* consume(void *arg){
	while(1){
	sem_wait(¬Empty);
	cout<<"Consumer consumes item."<<endl;
	cout<<"Total produced = "<<total_produced<<
		" Total consume = "<<(--total_consume)*-1<<endl;
	sleep(rand()%100*0.01);
	}
}

int main(int argv,char *argc[]){

	// thread declaration
	pthread_t producer,consumer;
```

```
	// Declaration of attribute.....
	pthread_attr_t attr;

	// semaphore initialization
	sem_init(¬Empty,0,0);

	// pthread_attr_t initialization
	pthread_attr_init(&attr);
	pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);

	// Creation of process
	r1=pthread_create(&producer,&attr,produce,NULL);
	if(r1){
	cout<<"Error in creating thread"<<endl;
	exit(-1);
	}

	r1=pthread_create(&consumer,&attr,consume,NULL);
	if(r1){
	cout<<"Error in creating thread"<<endl;
	exit(-1);
	}

	// destroying the pthread_attr
	pthread_attr_destroy(&attr);

	// Joining the thread
	r1=pthread_join(producer,NULL);
	if(r1){
	cout<<"Error in joining thread"<<endl;
	exit(-1);
	}

	r1=pthread_join(consumer,NULL);
	if(r1){
	cout<<"Error in joining thread"<<endl;
	exit(-1);
	}

	// Exiting thread
	pthread_exit(NULL);

	return 0;
}
```

Reader Writer:

```
#include<semaphore.h>
#include<stdio.h>
#include<pthread.h>
# include<bits/stdc++.h>
using namespace std;

void *reader(void *);
void *writer(void *);

int readcount=0,writecount=0,sh_var=5,bsize[5];
sem_t x,y,z,rsem,wsem;
pthread_t r[3],w[2];

void *reader(void *i)
{
	cout << "\n-------------------------";
	cout << "\n\n reader-" << i << " is reading";

	sem_wait(&z);
	sem_wait(&rsem);
	sem_wait(&x);
	readcount++;
	if(readcount==1)
	   sem_wait(&wsem);
	sem_post(&x);
	sem_post(&rsem);
	sem_post(&z);
	cout << "\nupdated value :" << sh_var;
	sem_wait(&x);
	readcount--;
	if(readcount==0)
	   sem_post(&wsem);
	sem_post(&x);
}

void *writer(void *i)
{
	cout << "\n\n writer-" << i << "is writing";
	sem_wait(&y);
	writecount++;
	if(writecount==1)
	sem_wait(&rsem);
	sem_post(&y);
	sem_wait(&wsem);

	sh_var=sh_var+5;
```

```
        sem_post(&wsem);
        sem_wait(&y);
        writecount--;
        if(writecount==0)
        sem_post(&rsem);
        sem_post(&y);
}

int main()
{
        sem_init(&x,0,1);
        sem_init(&wsem,0,1);
        sem_init(&y,0,1);
        sem_init(&z,0,1);
        sem_init(&rsem,0,1);

        pthread_create(&r[0],NULL,(void *)reader,(void *)0);
        pthread_create(&w[0],NULL,(void *)writer,(void *)0);
        pthread_create(&r[1],NULL,(void *)reader,(void *)1);
        pthread_create(&r[2],NULL,(void *)reader,(void *)2);
        pthread_create(&r[3],NULL,(void *)reader,(void *)3);
        pthread_create(&w[1],NULL,(void *)writer,(void *)3);
        pthread_create(&r[4],NULL,(void *)reader,(void *)4);

        pthread_join(r[0],NULL);
        pthread_join(w[0],NULL);
        pthread_join(r[1],NULL);
        pthread_join(r[2],NULL);
        pthread_join(r[3],NULL);
        pthread_join(w[1],NULL);
        pthread_join(r[4],NULL);

        return(0);
}
```

Least Recently used LRU:

```
// C++ program for page replacement algorithms
#include <iostream>
#include<bits/stdc++.h>
using namespace std;

int main()
{
int capacity = 4;
int arr[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
```

```
deque<int> q(capacity);
int count=0;
int page_faults=0;
deque<int>::iterator itr;
q.clear();
for(int i:arr)
{

        // Insert it into set if not present
        // already which represents page fault
        itr = find(q.begin(),q.end(),i);
        if(!(itr != q.end()))
        {

        ++page_faults;

        // Check if the set can hold equal pages
        if(q.size() == capacity)
        {
                q.erase(q.begin());
                q.push_back(i);
        }
        else{
                q.push_back(i);

        }
        }
        else
        {
        // Remove the indexes page
        q.erase(itr);

        // insert the current page
        q.push_back(i);
        }

}
cout<<page_faults;
}
```

Optimal Page Replacement:

```
// CPP program to demonstrate optimal page
// replacement algorithm.
#include <bits/stdc++.h>
using namespace std;
```

```
// Function to check whether a page exists
// in a frame or not
bool search(int key, vector<int>& fr)
{
	for (int i = 0; i < fr.size(); i++)
		if (fr[i] == key)
			return true;
	return false;
}

// Function to find the frame that will not be used
// recently in future after given index in pg[0..pn-1]
int predict(int pg[], vector<int>& fr, int pn, int index)
{
	// Store the index of pages which are going
	// to be used recently in future
	int res = -1, farthest = index;
	for (int i = 0; i < fr.size(); i++) {
		int j;
		for (j = index; j < pn; j++) {
			if (fr[i] == pg[j]) {
				if (j > farthest) {
					farthest = j;
					res = i;
				}
				break;
			}
		}

		// If a page is never referenced in future,
		// return it.
		if (j == pn)
			return i;
	}

	// If all of the frames were not in future,
	// return any of them, we return 0. Otherwise
	// we return res.
	return (res == -1) ? 0 : res;
}

void optimalPage(int pg[], int pn, int fn)
{
	// Create an array for given number of
	// frames and initialize it as empty.
	vector<int> fr;
```

```
	// Traverse through page reference array
	// and check for miss and hit.
	int hit = 0;
	for (int i = 0; i < pn; i++) {

		// Page found in a frame : HIT
		if (search(pg[i], fr)) {
			hit++;
			continue;
		}

		// Page not found in a frame : MISS

		// If there is space available in frames.
		if (fr.size() < fn)
			fr.push_back(pg[i]);

		// Find the page to be replaced.
		else {
			int j = predict(pg, fr, pn, i + 1);
			fr[j] = pg[i];
		}
	}
	cout << "No. of hits = " << hit << endl;
	cout << "No. of misses = " << pn - hit << endl;
}

// Driver Function
int main()
{
	int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };
	int pn = sizeof(pg) / sizeof(pg[0]);
	int fn = 4;
	optimalPage(pg, pn, fn);
	return 0;
}
```