**Student Name:** Arshad Ali

**Student ID :** 11705816

**Email Address:** arshadali1515@gmail.com

**GitHub Link:** https://github.com/arshad1515/OS-project

# Question 1:

A number of cats and mice inhabit a house. The cats and mice have worked out a deal where the mice can steal pieces of the cats’ food, so long as the cats never see the mice actually doing so. If the cats see the mice, then the cats must eat the mice (or else lose face with all of their cat friends). There are NumBowls cat food dishes, NumCats cats, and NumMice mice. Your job is to synchronize the cats and mice so that the following requirements are satisfied: No mouse should ever get eaten. You should assume that if a cat is eating at a food dish, any mouse attempting to eat from that dish or any other food dish will be seen and eaten. When cats aren’t eating, they will not see mice eating. In other words, this requirement states that if a cat is eating from any bowl, then no mouse should be eating from any bowl. Only one mouse or one cat may eat from a given dish at any one time. Neither cats nor mice should starve. A cat or mouse that wants to eat should eventually be able to eat. For example, a synchronization solution that permanently prevents all mice from eating would be unacceptable. When we actually test your solution, each simulated cat and mouse will only eat a finite number of times; however, even if the simulation were allowed to run forever, neither cats nor mice should starve.

**1. Code:**

#include <stdio.h>

#include <types.h>

#include <lib.h>

#include <test.h>

#include <thread.h>

#include <synch.h>

extern int initialize\_bowls(unsigned int bowlcount);

extern void cat\_eat(unsigned int bowlnumber);

extern void mouse\_eat(unsigned int bowlnumber);

extern void cat\_sleep(void);

extern void mouse\_sleep(void);

int NumBowls;

int Number\_of\_Cats;

int Number\_of\_Mice;

int Number\_of\_Loops;

struct semaphore \*CatMouseWait;

static void cat\_simulation(void \* unusedpointer,unsigned long catNumber\_of\_ber)

{

int i;

unsigned int bowl;

(void) unusedpointer;

(void) catNumber\_of\_ber;

for(i=0;i<Number\_of\_Loops;i++)

{

cat\_sleep();

bowl = ((unsigned int)random() % Number\_of\_Bowls) + 1;

cat\_eat(bowl);

}

V(CatMouseWait);

}

static void mouse\_simulation(void \* unusedpointer,unsigned long mouseNumber\_of\_ber)

{

int i;

unsigned int bowl;

(void) unusedpointer;

(void) mouseNumber\_of\_ber;

for(i=0;i<Number\_of\_Loops;i++)

{

mouse\_sleep();

bowl = ((unsigned int)random() % Number\_of\_Bowls) + 1;

mouse\_eat(bowl);

}

V(CatMouseWait);

}

int catmouse(int nargs,char \*\* args)

{

int index, error;

int i;

if (nargs != 5)

{

printf("Usage: Number\_of\_\_BOWLS Number\_of\_\_CATS Number\_of\_\_MICE Number\_of\_\_LOOPS\n");

return 1;

}

Number\_of\_Bowls = atoi(args[1]);

if (Number\_of\_Bowls <= 0)

{

printf("\t\nCat Mouse: invalid Number\_of Bowls:\*\* %d\n",Number\_of\_Bowls);

return 1;

}

Number\_of\_Cats = atoi(args[2]);

if (Number\_of\_Cats < 0)

{

printf("\t\nCat Mouse: invalid Number\_ of Cats: %d\n",Number\_of\_Cats);

return 1;

}

Number\_of\_Mice = atoi(args[3]);

if (Number\_of\_Mice < 0)

{

printf("\tCat Mouse: invalid Number\_ of Mice: %d\n",Number\_of\_Mice);

return 1;

}

Number\_of\_Loops = atoi(args[4]);

if (Number\_of\_Loops <= 0)

{

printf("catmouse: invalid Number\_of Loops: %d\n",Number\_of\_Loops);

return 1;

}

printf("Using %d bowls, %d cats, and %d mice. Looping %d times.\n",Number\_of\_Bowls,Number\_of\_Cats,Number\_of\_Mice,Number\_of\_Loops);

CatMouseWait = sem\_create("CatMouseWait",0);

if (CatMouseWait == NULL)

{

printf("catmouse: could not create semaphore\n");

}

if (initialize\_bowls(Number\_of\_Bowls))

{

printf("catmouse: error initializing bowls.\n");

}

for (index = 0; index < Number\_of\_Cats; index++)

{

error = thread\_fork("cat\_simulation thread",NULL,index,cat\_simulation,NULL);

if (error)

{

printf("cat\_simulation: thread\_fork failed: %s\n", strerror(error));

}

}

for (index = 0; index < Number\_of\_Mice; index++)

{

error = thread\_fork("mouse\_simulation thread", NULL ,index ,mouse\_simulation ,NULL);

if (error)

{

printf("mouse\_simulation: thread\_fork failed: %s\n",strerror(error));

}

}

for(i=0;i<(Number\_of\_Cats+Number\_of\_Mice);i++)

{

P(CatMouseWait);

}

sem\_destroy(CatMouseWait);

return 0;

}

**2. Description:**

In this assignment, you will implement synchronization primitives for and learn how to use them to solve several synchronization problems. Once you have completed the written and programming exercises you should have a fairly solid grasp of the pitfalls of concurrent programming and, more importantly, how to avoid those pitfalls in the code you will write later this term.

To complete this assignment you will need to be familiar with the threading code. The thread system provides interrupts, control functions, and semaphores. You will implement locks and condition variables

**3. Solve a Synchronization Problem:**

In this section of the assignment, you are asked to design and implement a solution to a synchronization problem using the synchronization primitives .The synchronization problem is called the “cats and mice” problem.

**The Cats and Mice Problem**

A number of cats and mice inhabit a house. The cats and mice have worked out a deal where the mice can steal pieces of the cats’ food, so long as the cats never see the mice actually doing so. If the cats see the mice, then the cats must eat the mice .There are Number\_of\_Bowls catfood dishes, Number\_of\_Cats cats, and Number\_of\_Mice mice. Your job is to synchronize the cats and mice so that the following requirements are satisfied:

**No mouse should ever get eaten.**

You should assume that if a cat is eating at a food dish, any mouse attempting to eat from that dish or any other food dish will be seen and eaten. When cats aren’t eating, they will not see mice eating. In other words, this requirement states that if a cat is eating from any bowl, then no mouse should be eating from any bowl.

**Neither cats nor mice should starve.**

A cat or mouse that wants to eat should eventually be able to eat. For example, a synchronization solution that permanently prevents all mice from eating would be unacceptable. When we actually test your solution, each simulated cat and mouse will only eat a finite number of times; however, even if the simulation were allowed to run forever, neither cats nor mice should starve. Your solution must not rely on knowledge of the numbers of cats and mice in the system. In particular, you should not make direct or indirect use of the variables Number\_of\_Cats and Number\_of\_Mice in your solution. (Those parameters should be used only by the catmouse() function to create the correct numbers of cats and mice to run a particular test.) It is OK to make use of the Number\_of\_Bowls parameter in your solution. There are many ways to synchronize the cats and mice that will satisfy the requirements above. From among the possible solutions that satisfy the requirements, we prefer solutions that are (a) efficient and (b) fair.

# Question 2:

Implement the multi-level feedback queue scheduling algorithm by considering the following diagram: You can use the code of others to implement Roud-Robin, and FCFS but implement aging by your own self.

**1. Code:**

#include<stdio.h>

#include<sys/type.h>

#include<sys/stat.h>

static int n;

int i,j,total\_time=0;

float avgtat=0,avgwt=0;

int queue[];

int time=0,front=0,rear=0,q\_count=0,m=0,s=0;

struct data

{

int at,st,ct,tat,wt;

char pname[20];

float ntat;

}temp;

int main()

{

printf("\t\nenter the Number of process\n");

scanf("%d",&n);

struct data aa[n];

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

{

printf("enter the name of process %d,at,st\n",i);

scanf("%s",&aa[i].pname);

scanf("%d%d",&aa[i].at,&aa[i].st);

}

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

{

for(j=i;j>=1;j--)

{

if(aa[j].at<aa[j-1].at)

{

temp=aa[j-1];

aa[j-1]=aa[j];

aa[j]=temp;

}

else if(aa[j].at==aa[j-1].at)

{

if(aa[j].st<aa[j-1].st)

{

temp=aa[j-1];

aa[j-1]=aa[j];

aa[j]=temp;

}

}

}

}

total\_time+=aa[0].at+aa[0].st;

for(i=1;i<n;i++)

{

if(aa[i].at>total\_time)

total\_time=aa[i].at;

total\_time+=aa[i].st;

}

finding(aa);

}

void finding(struct data a[])

{

int temp\_st[n],flag=0,count=0,p\_process;

j=0;

int tq;

printf("enter the time quantum\n");

scanf("%d",&tq);

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

{

temp\_st[i]=a[i].st;

}

time=a[0].at;

q\_count=1;

push(j++);

while(time<=total\_time)

{

if(flag==1||q\_count!=0)

{

if(flag==0&&count==0)

{

p\_process=pop();

count=0;

flag=1;

}

temp\_st[p\_process]--;

if(temp\_st[p\_process]==0)

{

time++;

count=0;

a[p\_process].ct=time;

flag=0;//a[i].st+=1000;

q\_count--;

check(a);

continue;

}

count++;

if(count==tq)

{

count=0;

time++;

check(a);

push(p\_process);

flag=0;

}

else

{

time++;

check(a);

}

}

else

{

time++;

check(a);

}

}

display(a);

}

void push(int q)

{

queue[rear++]=q;

m++;

}

int pop()

{

s++;

int x;

x=queue[front++];

return x;

}

void check(struct data a[])

{

while(a[j].at<=time&&j<n)

{

q\_count++;

push(j++);

}

}

void display(struct data a[])

{

printf("n=%d \n",n);

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

{

a[i].tat=a[i].ct-a[i].at;

a[i].wt=a[i].tat-a[i].st;

a[i].ntat=(float)a[i].tat/a[i].st;

}

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

{

avgtat+=a[i].tat;

avgwt+=a[i].wt;

}

avgtat=avgtat/n;

avgwt=avgwt/n;

printf("pname\tat\tst\tct\ttat\twt\tntat\n");

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

{

printf("%s\t%d\t%d\t%d\t%d\t%d\t%f\n",a[i].pname,a[i].at,a[i].st,a[i].ct,a[i].tat,a[i].wt,a[i].ntat);

}

printf("avgtat=%f\navgwt=%f\n",avgtat,avgwt);

}

**2. Description:**

a. FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.

b. Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time

c. FCFS can also block the system in a busy dynamic system in another way, known as the ***convoy effect***. When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle. When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.

**Multilevel Feedback-Queue Scheduling**

Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:

If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.

Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.

Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters. Some of the parameters which define one of these systems include:

The number of queues.

The scheduling algorithm for each queue.

The methods used to upgrade or demote processes from one queue to another. ( Which may be different. )

The method used to determine which queue a process enters initially.