**Dining philosophers problem using mutex**

**Problem Description**  
Develop a program to implement the solution of the dining philosopher’s  
problem using **threads**. The input to the program is the number of philosophers to be seated around the table. Output shows the various stages that each philosopher passes through within a certain time. A philosopher can be in anyone of the three stages at a time: thinking, eating or finished eating.

**Data Structures and Functions**

**The main data structures used here are**: Arrays

The arrays represent the philosophers and corresponding chopsticks for them.Each element in the philosopher’s array corresponds to a thread and each element in the chopstick’s array corresponds to a mutex variable.

The functions used here are:

1. pthread\_mutex\_init (&mutex, NULL) – initialization of mutex variable
2. pthread\_mutex\_lock (&mutex) – attempt to lock a mutex
3. pthread\_mutex\_unlock (&mutex) – unlock a mutex
4. pthread\_create (ptr to thread, NULL, (void\*) func, (void\*) )
5. pthread\_join (ptr to thread, &msg)-This function will make the main program wait until the called thread is finished executing it’s task.
6. pthread\_mutex\_destroy (ptr to thread)-
7. pthread\_exit(NULL)

Note: while compiling this program use the following:  
[root@Linux philo]# gcc –o dining dining.c -lpthread

**Algorithm**  
Algorithm for process:  
1. Start.  
2. Declare and initialize the thread variables (philosophers) as required.  
3. Declare and initialize the mutex variables (chopsticks) as required.  
4. Create the threads representing philosophers.  
5. Wait until the threads finish execution.  
6. Stop.

**Algorithm for thread (philosopher i) function:**

1. Start.
2. Philosopher i is thinking.
3. Lock the left fork spoon.
4. Lock the right fork spoon.
5. Philosopher i is eating.
6. sleep
7. Release the left fork spoon.
8. Release the right fork spoon.
9. Philosopher i Finished eating.
10. Stop.

**CODE USINGS THREADS ONLY**

1. #include<stdio.h>  
   #include<stdlib.h>  
   #include<pthread.h>  
   #include<semaphore.h>  
   void \*func(int n);  
   pthread\_t philosopher[5];  
   pthread\_mutex\_t chopstick[5];  
   int main()  
   {  
   int i,k;  
   void \*msg;  
   for(i=1;i<=5;i++)  
   {  
   k=pthread\_mutex\_init(&chopstick[i],NULL);  
   if(k==-1)  
   {  
   printf(“\n Mutex initialization failed”);  
   exit(1);  
   }  
   }  
   for(i=1;i<=5;i++)  
   {  
   k=pthread\_create(&philosopher[i],NULL,(void \*)func,(int \*)i);  
   if(k!=0)  
   {  
   printf(“\n Thread creation error \n”);  
   exit(1);  
   }  
   }  
   for(i=1;i<=5;i++)  
   {  
   k=pthread\_join(philosopher[i],&msg);  
   if(k!=0)  
   {  
   printf(“\n Thread join failed \n”);  
   exit(1);  
   }  
   }  
   for(i=1;i<=5;i++)  
   {  
   k=pthread\_mutex\_destroy(&chopstick[i]);  
   if(k!=0)  
   {  
   printf(“\n Mutex Destroyed \n”);  
   exit(1);  
   }  
   }  
   return 0;  
   }void \*func(int n)  
   {  
   printf(“\nPhilosopher %d is thinking “,n);  
   pthread\_mutex\_lock(&chopstick[n]);//when philosopher 5 is eating he takes fork 1 and fork 5  
   pthread\_mutex\_lock(&chopstick[(n+1)%5]);  
   printf(“\nPhilosopher %d is eating “,n);  
   sleep(3);  
   pthread\_mutex\_unlock(&chopstick[n]);  
   pthread\_mutex\_unlock(&chopstick[(n+1)%5]);  
   printf(“\nPhilosopher %d Finished eating “,n);

}

**CODE USINGS SEMAPHORES ONLY**

//Dining philosopher using multiprogramming using semaphores  
#include<stdio.h>  
#include<fcntl.h>  
#include<semaphore.h>  
#include<sys/wait.h>  
#include<pthread.h>  
#include<stdlib.h>  
sem\_t \*sem[20];  
int n;  
int main()  
{  
pid\_t cpid[5];  
char semname[5];  
int i,j=0;  
n = 5;  
for(i=0;i<n;i++)  
{  
sprintf(semname,”%d”,getpid()+i);  
sem[i]=sem\_open(semname,O\_CREAT|O\_EXCL,0666,1);  
if(sem[i]==SEM\_FAILED)  
perror(“Unable to create semaphore”);

}

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

cpid[i]=fork();  
if(cpid[i]==0)  
break;

}  
if(i==n)  
{  
int status;  
for(i=0;i<n;i++)  
waitpid(cpid[i],&status,WUNTRACED);

//waitpid is a function which waits for the child process to finish executing after that //control switches back to parent  
for(i=0;i<n;i++)  
{  
sem\_close(sem[i]);  
sprintf(semname,”%d”,getpid()+i);  
sem\_unlink(semname);  
}  
}  
else  
reader(i);

}  
int reader(int val)  
{  
printf(“%d Thinking\n”,val+1);  
while(1)  
{  
sem\_wait(sem[val%n]);  
if(!sem\_trywait(sem[(val+1)%n]))  
break;  
else  
sem\_post(sem[val%n]);  
}  
printf(“%d Eating\n”,val+1);  
sleep(2);  
sem\_post(sem[val%n]);  
sem\_post(sem[(val+1)%n]);  
printf(“%d Finished Eating\n”,val+1);  
}

**EXPECTED OUTPUT**

**REMEMBER :*use the command -pthread***

[root@localhost ~]$ gcc -o c dining.c -pthread  
[root@localhost ~]$ ./c

Philosopher 1 is thinking  
Philosopher 1 is eating  
Philosopher 2 is thinking  
Philosopher 3 is thinking  
Philosopher 3 is eating  
Philosopher 4 is thinking  
Philosopher 5 is thinking  
Philosopher 1 Finished eating  
Philosopher 3 Finished eating  
Philosopher 4 is eating  
Philosopher 5 is eating  
Philosopher 2 is eating  
Philosopher 4 Finished eating  
Philosopher 5 Finished eating  
Philosopher 2 Finished eating

/\* To implement the DINING PHILOSOPHER problem\*/

Author  Raymond Antony

#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <semaphore.h>

#define phil 5 // phil –> PHILOSOPHERS.

void \* thread\_func(int n);

pthread\_mutex\_t cs[phil]; // cs –> CHOPSTICKS.

int main()  
{  
pthread\_t a\_thread[phil];  
int i,k;  
void \* message; // Default message.

for(i=0;i<phil;i++)  
{  
k = pthread\_mutex\_init(&cs[i],NULL);  
if(k==-1)  
{  
printf(“\n\t MUTEX INITIALISATION FAILED \n”);  
exit(1);  
}  
}

for(i=0;i<phil;i++)  
{  
k = pthread\_create(&a\_thread[i],NULL,thread\_func,(int)i);  
if(k!=0)  
{  
printf(“\n\t THREAD CREATION FAILED \n”);  
exit(0);  
}  
}

for(i=0;i<phil;i++)  
{  
k = pthread\_join(a\_thread[i],message);  
if(k!=0)  
{  
printf(“\n\t THREAD JOINING FAILED \n”);  
exit(0);  
}  
}

exit(0);  
printf(“\n\t THREADS JOINED SUCCESSFULLY \n\t%s\n”,(char \*)message);

for(i=0;i<phil;i++)  
{  
k = pthread\_mutex\_destroy(&cs[i]);  
if(k==-1)  
{  
printf(“\n\t MUTEX DESTRUCTION FAILED \n”);  
exit(1);  
}  
}  
return 0;  
}

void \* thread\_func(int n) // n is the philosopher.  
{  
int i;  
for(i=0;i<5;i++)  
{  
sleep(1); // PHILOSOPHER is thinking \*/  
pthread\_mutex\_lock(&cs[n]);  
pthread\_mutex\_lock(&cs[(n+1)% phil ] );  
printf(“\n\t PHILOSOPHER %d is EATING \n”,n);  
sleep(2);

printf(“\n\t PHILOSOPHER %d has FINISHED EATING and is now THINKING \n”,n);  
pthread\_mutex\_unlock(&cs[n]);  
pthread\_mutex\_unlock(&cs[(n+1)%phil] );  
sleep(1);  
}  
pthread\_exit(“\n\n\t THANK YOU \n\n”);  
}