What is Dining Philosophers problem in C?

Suppose there are N philosophers meeting around a table, eating spaghetti and talking about philosophy. Now let us discuss the problem. There are only N forks available such that only one fork between each philosopher. Since there are only 5 philosophers and each one requires 2 forks to eat, we need to formulate an algorithm which ensures that utmost number of philosophers can eat spaghetti at once.

The next question is why we are detailing problems in this manner? Sometimes when it comes to computers, some of the analogous situations often demands solutions in a creative fashion. This is somewhat like an abstract problem in a novel dimension.

In this problem, the condition is each philosopher has to think and eat alternately. Assume that there is an infinite supply of spaghetti and eating is by no means limited by the quantity of food left. When available, each philosopher can pick up the adjacent fork. But he can eat only if the right and left forks are available.

**Dining Philosophers Problem using Semaphore**

**PROBLEM DEFINITION**

To implement Dining Philosophers Problem using Threads and Semaphores

**ALGORITHM**

1. Define the number of philosophers
2. Declare one thread per philosopher
3. Declare one semaphore (represent chopsticks) per philosopher
4. When a philosopher is hungry
   1. See if chopsticks on both sides are free
   2. Acquire both chopsticks or
   3. eat
   4. restore the chopsticks
   5. If chopsticks aren’t free
5. Wait till they are available

**PROGRAM DEVELOPMENT**

# include<stdio.h>

# include<pthread.h>

# include<stdlib.h>

# include<unistd.h>

# include<ctype.h>

# include<sys/types.h>

# include<sys/wait.h>

# include<semaphore.h>

# include<sys/sem.h>

sem\_t chopstick[100];

int n;

void \*thread\_func(int no)

{

int i,iteration=5;

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

{

sem\_wait(&chopstick[no]);

sem\_wait(&chopstick[(no+1)%n]);

printf(“\nPhilosopher %d –> Begin eating”,no);

sleep(1);

printf(“\nPhilosopher %d –> Finish eating\n”,no);

sem\_post(&chopstick[no]);

sem\_post(&chopstick[(no+1)%n]);

}

pthread\_exit(NULL);

}

int main()

{

int i,res;

pthread\_t a\_thread[100];

printf(“\nEnter the number of philosopher :”);

scanf(“%d”,&n);

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

{

res=sem\_init(&chopstick[i],0,0);

if(res==-1)

{

perror(“semaphore initialization failed”);

exit(1);

}

}

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

{

res=pthread\_create(&a\_thread[i],NULL,thread\_func,(int\*) i);

if(res!=0)

{

perror(“semaphore creation failed”);

exit(1);

}

sem\_post(&chopstick[i]);

}

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

{

res=pthread\_join(a\_thread[i],NULL);

if(res!=0)

{

perror(“semaphore join failed”);

exit(1);

}

}

printf(“\n \n thread join succesfull\n”);

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

{

res=sem\_destroy(&chopstick[i]);

if(res==-1)

{

perror(“semaphore destruction failed”);

exit(1);

}

}

exit(0);

}

**OUTPUT**

$ gcc dining\_sem.c -o dining\_sem\_op -lpthread

$ ./dining\_sem\_op

Next let us check out another method to solve the problem.

**Dining Philosophers Problem using MUTEX**

**PROBLEM DEFINITION**

To implement Dining Philosophers Problem using Threads and mutex

**ALGORITHM**

1. Define the number of philosophers
2. Declare one thread per philosopher
3. Declare one mutex(represent chopsticks) per philosopher
4. When a philosopher is hungry
   1. See if chopsticks on both sides are free
   2. acquire chopsticks
   3. eat
   4. restore the chopsticks
   5. If chopsticks aren’t free
5. wait till they’re available

**PROGRAM DEVELOPMENT**

# include<stdio.h>

# include<pthread.h>

# include<stdlib.h>

# include<unistd.h>

# include<ctype.h>

# include<sys/types.h>

# include<sys/wait.h>

# include<semaphore.h>

# define philosopher 5

pthread\_mutex\_t chopstick[philosopher]

int main()

{

int i,res;

pthread\_t a\_thread[philosopher];

void \*thread\_func(int n)

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

{

res=pthread\_mutex\_init(&chopstick[i],NULL);

if(res==-1)

{

perror(“mutex initialization failed”);

exit(1);

}

}

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

{

res=pthread\_create(&a\_thread[i],NULL,thread\_func,(int)i);

if(res!=0)

{

perror(“mutex creation failed”)

exit(1);

}

}

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

{

res=pthread\_join(a\_thread[i],NULL);

if(res!=0)

{

perror(“mutex join failed”);

exit(1);

}

}

printf(“thread join successful\n”);

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

{

res=pthread\_mutex\_destroy(&chopstick[i]);

if(res==-1)

{

perror(“mutex destruction failed”)

exit(1);

}

}

exit(0);

}

void \*thread\_func(int n)

{

int i,iteration=5;

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

{

sleep(1);

pthread\_mutex\_lock(&chopstick[n]);

pthread\_mutex\_lock(&chopstick[n+1)%philosopher]);

printf(“\nBegin eating :%d”,N);

sleep(1);

printf(“\nFinish eating”%d”,n);

pthread\_mutex\_unlock(&chopstick[n]);

pthread\_mutex\_unlock(&chopstick[n+1)%philosopher]);

}

pthread\_exit(NULL);

}

**OUTPUT**

$ gcc dining\_sem.c -o dining\_sem\_op -lpthread

$ ./dining\_sem\_op

What is difference between mutex and semaphore?

A mutex is owned by a thread/process. So once a thread locks it, then other threads/processes will either spin or block on the mutex. Whereas, semaphore allows one or more threads/processes to share the resource.

1. A semaphore can be a Mutex but a Mutex can never be semaphore. This simply means that a binary semaphore can be used as Mutex, but a Mutex can never exhibit the functionality of semaphore.
2. Both semaphores and Mutex (at least the on latest kernel) are non-recursive in nature.
3. No one owns semaphores, whereas Mutex are owned and the owner is held responsible for them. This is an important distinction from a debugging perspective.
4. In case the of Mutex, the thread that owns the Mutex is responsible for freeing it. However, in the case of semaphores, this condition is not required. Any other thread can signal to free the semaphore by using the sem\_post()function.
5. A Mutex, by definition, is used to serialize access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A semaphore, by definition, restricts the number of simultaneous users of a shared resource up to a maximum number
6. Another difference that would matter to developers is that semaphores are system-wide and remain in the form of files on the filesystem, unless otherwise cleaned up. Mutex are process-wide and get cleaned up automatically when a process exits.
7. The nature of semaphores makes it possible to use them in synchronizing related and unrelated process, as well as between threads. Mutex can be used only in synchronizing between threads and at most between related processes (the pthread implementation of the latest kernel comes with a feature that allows Mutex to be used between related process).
8. According to the kernel documentation, Mutex are lighter when compared to semaphores. What this means is that a program with semaphore usage has a higher memory footprint when compared to a program having Mutex.
9. From a usage perspective, Mutex has simpler semantics when compared to semaphores.

What is a command interpreter and why is it separate from the kernel?  
  
A command interpreter is an interface of the operating system with the user. The user gives commands with are executed by operating system (usually by turning them into system calls). The main function of a command interpreter is to get and execute the next user specified command. Command-Interpreter is usually not part of the kernel, since multiple command interpreters (shell, in UNIX terminology) may be support by an operating system, and they do not really need to run in kernel mode.

There are two main advantages to separating the command interpreter from the kernel.  
1. If we want to change the way the command interpreter looks, i.e., I want to change the interface of command interpreter, I am able to do that if the command interpreter is separate from the kernel. I cannot change the code of the kernel so I cannot modify the interface.  
2. If the command interpreter is a part of the kernel it is possible for a malicious process to gain access to certain part of the kernel that it showed not have to avoid this ugly scenario it is advantageous to have the command interpreter separate from kernel

Fork : The fork call basically makes a duplicate of the current process, identical in almost every way (not everything is copied over, for example, resource limits in some implementations but the idea is to create as close a copy as possible).

The new process (child) gets a different process ID (PID) and has the the PID of the old process (parent) as its parent PID (PPID). Because the two processes are now running exactly the same code, they can tell which is which by the return code of fork - the child gets 0, the parent gets the PID of the child. This is all, of course, assuming the fork call works - if not, no child is created and the parent gets an error code.

Vfork : The basic difference between vfork and fork is that when a new process is created with vfork(), the parent process is temporarily suspended, and the child process might borrow the parent's address space. This strange state of affairs continues until the child process either exits, or calls execve(), at which point the parent process continues.

This means that the child process of a vfork() must be careful to avoid unexpectedly modifying variables of the parent process. In particular, the child process must not return from the function containing the vfork() call, and it must not call exit() (if it needs to exit, it should use \_exit(); actually, this is also true for the child of a normal fork()).

Exec : The exec call is a way to basically replace the entire current process with a new program. It loads the program into the current process space and runs it from the entry point.exec() replaces the current process with a the executablepointed by the function. Control never returns to the original program unless there is an exec() error.

Clone : Clone, as fork, creates a new process. Unlike fork, these calls allow the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.

When the child process is created with clone, it executes the function application fn(arg). (This differs from for, where execution continues in the child from the point of the fork call.) The fn argument is a pointer to a function that is called by the child process at the beginning of its execution. The arg argument is passed to the fn function.

When the fn(arg) function application returns, the child process terminates. The integer returned by fn is the exit code for the child process. The child process may also terminate explicitly by calling exit(2) or after receiving a fatal signal.