Assignment No. 5

Date:

TITLE: Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.

OBJECTIVE:

- Study how to use POSIX threads in Linux
- Implement multithreading in Linux using C.
- Implement POSIX thread functions for thread create, join, exit.
- Study of deadlock of avoidance.

SOFTWARE REQUIREMENTS:

1. Ubuntu 16.04
2. GNU C Compiler

THEORY:

POSIX thread (pthread) libraries

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.) Parallel programming technologies such as MPI and PVM are used in a distributed computing environment while threads are limited to a single computer system. All threads within a process share the same address space. A thread is spawned by defining a function and it's arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

Thread Basics:

Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.

- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.

Each thread has a unique:

Thread ID

set of registers, stack pointer

stack for local variables, return addresses
signal mask

priority

Return value: errno

pthread functions return "0" if OK.

pthread_create

int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine) (void *), void *arg);

Arguments:

thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)

attr - Set to NULL if default thread attributes are used.

*void * (*start_routine)* - pointer to the function to be threaded. Function has a single argument: pointer to void.

**arg* - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

pthread_exit

void pthread_exit(void *retval);

Arguments:

retval - Return value of thread.

This routine kills the thread. The pthread_exit function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join.

Thread Synchronization:

The threads library provides three synchronization mechanisms:

1. *mutexes* - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.

2. *joins* - Make a thread wait till others are complete (terminated).
3. *condition variables* - data type pthread_cond_t

Mutexes:

Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed. Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. One can apply a mutex to protect a segment of memory ("critical region") from other threads. Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

Joins:

A join is performed when one wants to wait for a thread to finish. A thread calling routine may launch multiple threads then wait for them to finish to get the results. One wait for the completion of the threads with a join.

```
int pthread_join(pthread_t thread, void **retval);
```

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

Bankers Algorithm for DeadLock Avoidance

Concept: Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether them allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Data structures

n-Number of process, m-number of resource types.
Available: Available[j]=k, k – instance of resource type Rj is available.
Max: If max[i, j]=k, Pi may request at most k instances resource Rj.
Allocation: If Allocation [i, j]=k, Pi allocated to k instances of resource Rj

Need: If Need[I, j]=k, Pi may need k more instances of resource type Rj, Need[I, j]=Max[I, j]-Allocation[I, j];

Safety Algorithm

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
2. Find an i such that both Finish[i] =False Need<=Work If no such I exists go to step 4.
3. work= work + Allocation, Finish[i] =True;
4. if Finish[1]=True for all I, then the system is in safe state. Resource request algorithm

Let Request i be request vector for the process Pi, If request i=[j]=k, then process Pi wants k instances of resource type Rj.

1. if Request<=Need I go to step 2. Otherwise raise an error condition.
2. if Request<=Available go to step 3. Otherwise Pi must since the resources are available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows;

Available=Available-Request I;

Allocation I =Allocation +Request I;

Need i=Need i-Request I;

If the resulting resource allocation state is safe, the transaction is completed and process Pi is allocated its resources. However if the state is unsafe, the Pi must wait for Request i and the old resource-allocation state is restored.

ALGORITHM:

1. Start .
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. or not if we allow the request.
10. stop .

Conclusion: Thus we have studied multithreading and implemented Deadlock Avoidance Bankers Algorithm.