1. Which applications can be multithreaded?

Answer:

The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program that can run concurrently.

Threads are independent. If an exception occurs in one thread, it doesn't affect the others.

Some multithreaded applications would be:

1. **Web Browsers - A web browser can download any number of files and web pages (multiple tabs) at the same time and still lets you continue browsing. If a particular web page cannot be downloaded, that is not going to stop the web browser from downloading other web pages.**
2. **Web Servers - A threaded web server handles each request with a new thread. There is a thread pool and every time a new request comes in, it is assigned to a thread from the thread pool.**
3. **Computer Games - You have various objects like cars, humans, birds which are implemented as separate threads. Also playing the background music at the same time as playing the game is an example of multithreading.**
4. **Text Editors - When you are typing in an editor, spell-checking, formatting of text and saving the text are done concurrently by multiple threads. The same applies for Word processors also.**
5. **IDE - IDEs like Android Studio run multiple threads at the same time. You can open multiple programs at the same time. It also gives suggestions on the completion of a command which is a separate thread.**

2.Which applications cannot be multithreaded?

Answer:

On a single processor machine and a desktop application, you use multi threads so you don't freeze the app but for nothing else really.

1. **On a single processor server and a web based app, no need for multi threading because the web server handles most of it.**
2. **On a multi-processor machine and desktop app, you are suggested to use multi threads and parallel programming. Make as many threads as there are processors.**
3. **On a multi-processor server and a web based app, no need again for multi threads because the web server handles it.**

3.How to create or use POSIX threads?

Answer:

POSIX threads is also known as pthread. Pthread can be created by following the below steps.
POSIX provides pthread_create() API to create a thread
pthread_create() accepts 4 arguments i.e.

1. **Pointer of the Thread ID, it will update the value in it.**
2. **Attributes to set the properties of thread.**
3. **Function pointer to the function that thread will run in parallel on start. This function should accept a void * and return void * too.**
4. **Arguments to be passed to function.**

So, now let's call pthread_create() by passing function pointer and other arguments i.e.

```
pthread_t threadId;
int err = pthread_create(&threadId, NULL, &threadFunc, NULL);
Check if thread is created sucessful
```

pthread_create() returns the error code to specify the result of thread creation request. If thread is created successfully then it will return 0. Where as, if thread creation is failed it will return error code to specify the error. We can use strerror() to get the detail of error.

```
if (err)
std::cout << "Thread creation failed : " << strerror(err);
else
std::cout << "Thread Created with ID : " << threadId << std::endl;
```

Main function and other created threads runs in parallel. But when main function ends, complete process exits and all the other thread will also be terminated. Therefore, in main function before ending we should wait for other threads to exit.

5. **POSIX Library provides a function for it i.e. pthread_join() to wait for other thread to exit.**

```
err = pthread_join(threadId, NULL);
if (err)
std::cout << "Failed to join Thread : " << strerror(err) << std::endl;
```

It accepts a thread ID and pointer to store the return value from thread.

2. How to compile and execute program using POSIX threads API. Mention the statements to compile a file mythread.c and generate an executable "mythread".

   Answer:

   to compile we need following command

   ```
           g++ <cpp filename> -o -lpthread
   ```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
  void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

3. Is mutex same as semaphore?

   Answer:

   A Mutex is different than a semaphore as it is a locking mechanism while a semaphore is a signalling mechanism. A binary semaphore can be used as a Mutex but a Mutex can never be used as a semaphore. So mutex is not same as semaphore.

4. What is the difference between mutex and semaphore? Can we use mutex as a semaphore and vice versa?

   Answer:

   Semaphore is an integer variable. Mutex allows multiple program threads to access a single resource but not simultaneously. Semaphore allows multiple program threads to access a finite instance of resources. Mutex object lock is released only by the process that has acquired the lock on the mutex object.

   No we can't use mutex as a semaphore but A binary semaphore can be used as a Mutex but a Mutex can never be used as a semaphore

5. What are the 2 thread standards? Which one is commonly used?

   Answer:

   Two major Unified thread series are in use: UN and UNR. For the UN series, you specify C (coarse), F (fine), or EF (extra fine), as required. As any professional knows, it's important to understand and be able to identify every single type of thread. Given that the most common thread type is the UN/UNF thread type, you may become more familiar with this kind of thread type

6. What do you mean by starvation and deadlock? When will it occur? What are the solutions to handle that?

   Answer:

   Starvation occurs if a process is indefinitely postponed. This may happen if the process requires a resource for execution that it is never alloted or if the process is never provided the processor for some reason.

   deadlock occurs when two or more processes need some resource to complete their execution that is held by the other process.

Occurrence: Starvation occurs when one or more threads in your program are blocked from gaining access to a resource and, as a result, cannot make progress.

Deadlock, the ultimate form of starvation, occurs when two or more threads are waiting on a condition that cannot be satisfied.

Solution of starvation and deadlock:

Random selection of processes for resource allocation or processor allocation should be avoided as they encourage starvation. The priority scheme of resource allocation should include concepts such as aging, where the priority of a process is increased the longer it waits

Deadlock can be prevented by eliminating any of the four necessary conditions, which are mutual exclusion, hold and wait, no preemption, and circular wait. Mutual exclusion, hold and wait and no preemption cannot be violated practically. Circular wait can be feasibly eliminated by assigning a priority to each resource

7. What all are shared across threads?

Answer:

The items that are shared among threads within a process are:

- **Text segment**
- **Data segment**
- **BSS segment**
- **Open file descriptors.**
- **Signals.**
- **Current working directory.**
- **User and group IDs.**

8. What is the difference between thread, process and a program?

Answer:

The Process and Thread are the essentially associated. The process is an execution of a program whereas thread is an execution of a program driven by the environment of a process. Another major point which differentiates process and thread is that processes are isolated with each other whereas threads share memory or resources with each other.

Is thread id same as PID? How will you get thread id?

Answer:

You will get same Process ID as all threads are sharing your program data which is your process so when you call for Process ID you get the same.

yes, Thread and process have the same PID. Whenever a process spawns a thread or multiple threads, all of them (including process) have the same PID. The difference will be in their TGID (Thread group ID). Threads are identifying by their unique ID called TGID.

we call gettid() in Linux, to get thread ID(TID).

9. Name at least 5 thread attributes.

   Answer:

   1) **Priority.**
   2) **Stack size**
   3) **Name.**
   4) **Thread group.**
   5) **Detach state**
   6) **Scheduling policy.**
   7) **Inherit scheduling.**

10. Refer link below and answer the question.

    https://man7.org/linux/man-pages/man3/pthread_attr_setscope.3.html

    what are the 2 thread scopes, if there are 4 processes, each with 3 threads then comment on %allocation of CPU to every thread in the 2 thread scope cases.

    Answer:

    1) process local scheduling (known as Process Contention Scope, or Unbound Threads—the Many-to-Many model) and

    2) system global scheduling (known as System Contention Scope, or Bound Threads—the One-to-One model).

    if there are 4 processes, each with 3 threads then comment on %allocation of CPU to every thread in the 2 thread scope cases.

    % of allocation of CPU to every thread in the 2 thread scope cases will be 4.1% for each thread on the basis of pcs and scs scopes.

11. Which one is memory intensive, multithreading or multiprocessing?

    Answer:

    Multithreading is memory intensive.

12. Why are threads referred as LWP?

    Answer:

    Threads are sometimes called lightweight processes because they have their own stack but can access shared data. Because threads share the same address space as the process and other threads within the process, the operational cost of communication between the threads is low, which is an advantage.

13. Can threads of different process communicate with each other? If yes, what are the mechanisms available?

    Answer:

    No, threads of different process cannot communicate with each other.

14. What is the difference between concurrent and parallel processing?

    Answer:

    Concurrency is the task of running and managing the multiple computations at the same time. While parallelism is the task of running multiple computations simultaneously.