Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 07
Lecture – 33
Threads (Light Weight Processes) Part 1

Consider this particular problem. If it takes one man, 10 days to do a job then how many days will it take 10 men to do the same job? So, we have all done such problems during our school days, and the answer is quite simple to compute. Essentially it would take 10 men 1 day to do the job.

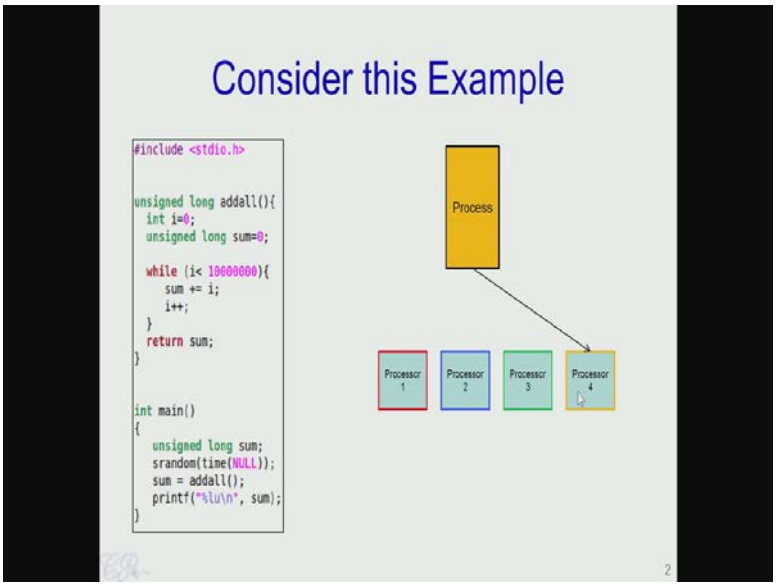
Now the job is done much more quickly because the 10 men are working in parallel. So, each man just needs to do one-tenth of the entire job and the entire job is just completed within a single day. So, this very same concept of parallelization is applied extensively in computer systems. Essentially, parallelization in a computer system is used quite often to improve performance of applications. So, what is done is that if there is a large job to do, we will parallelize it in the system. So, that different computing entity, these are computing entities which are very small and do a small portion of the larger job and these computing entities execute in parallel and all together the job will complete much more quickly.

So, these days parallelization is extensively supported by several computer hardware. In fact, there is some hardware that is present which are called GP use or graphing processing units which are capable of performing of tasks in parallel. So, each of these tasks are a very small part of a larger job, but the fact that all these tasks are done in parallel will achieve a much more lesser time to complete the larger job.

Now one important construct when doing parallelization is threading or threads, or threads are essentially execution entity very much like processes which we are studied in the previous videos; however, threads are extremely light weight processes, and they are essentially used to make parallelization much easier.

In this video we will look about threads and essentially provide an introduction to threads. So, the video will cover how threads are created and destroyed how it differs from processes, and how different operating systems support threads in different ways.

(Refer Slide Time: 03:12)



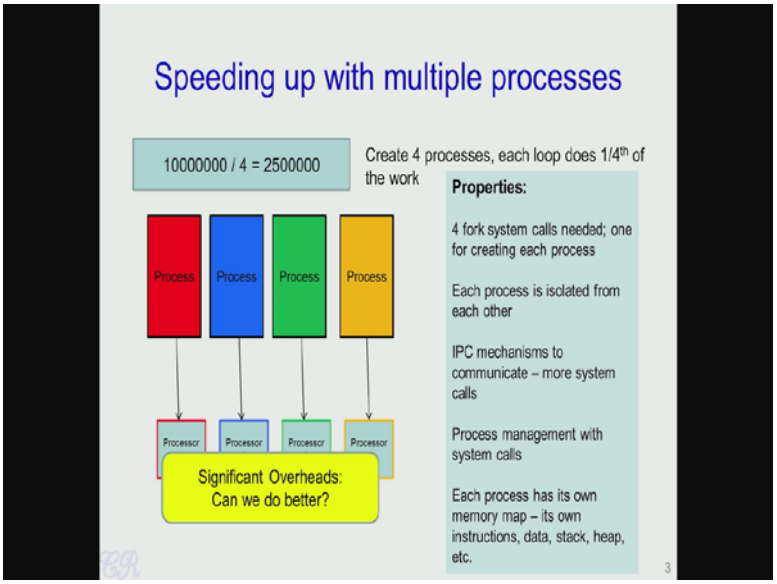
Let us start this particular video with an example. Let us consider this particular program essentially we have a function here called add all which is invoked from main and returns unsigned long. So, this function add all, just sums up or adds up the first 10 million possible integers and the sum is returned over here, and printed in the main function. So obviously, this particular program could be written in a much more different and much easier way, without having to iterate through every number from 0 to 10 million; however, for stay sake of understanding threads, let us go with this particular program.

So, what we have seen in the earlier videos is that when we after we write this program, we use a compiler such as GCC which creates an executable and then that executable is run to create a process. So, we will eventually have a process which has its own memory space and part of it may be in the RAM and now let us see what would happen if this process is executed on a system with say 4 different processes. So, what happens is that the scheduler in the operating system is going to pick up this process and assign it to one of these processes. So, always once assigned this process will be executed in one of these processes, and a really this process may migrate to 2 other processes, but let us not get concerned with that.

Now, what we observe here is that even though we have 4 processors present in the system. These entire programs which we take consider the long to execute, on a system.

Just runs on a single processor, leaving the other 3 processors idol or doing some other tasks. Thus the execution time of this entire program will be quite large because this some is found sequentially by iterating through all numbers from 1 to 10 million in a sequential manner as done in this particular while loop. So, given that we have hardware like this with 4 processors this is not a very good way or efficient way to write this particular program. So, what can we do better to make this program parallelizable and make it to execute on all these 4 processors.

(Refer Slide Time: 06:07)



So, what we will do is that we will take 10 million numbers and divide it by 4 to get 4 quarters of 2.5 million each. So, then we would create 4 processes and each process will loop through a quarter of these numbers; that means, the first process would loop through numbers from 0 to 2.5 million and find the sum of all these numbers, process 2 will find the sum of the numbers from 2.5 million to 5 million, a process 3 would find the sum of numbers from 5 million to 7.5 million and process 4 will find the sum of numbers from 7.5 million to 10 million.

If we are able to create 4 processes in such a way, then when we execute these 4 processes, it will be something like this. So, each of these 4 processes execute on a different processor in the system and an each of these processors just do a smaller job of finding the sum of 2.5 million numbers instead of finding the sum of the entire range of 10 million numbers.

Since, each process scheduled in its own processor therefore, we get parallelization. Thus what one would expect is that the entire program of finding the sum of the 10 million numbers can be completed 4 times faster. So, this seems to solve our problem of parallelization and it will more effectively use the processors, when more effectively use the processors present in the system. And some of the properties that we have of this particular approach are as follows. So, in order to create these particular 4 processes we would require 4 fork system calls. So, each of the fork system calls that we invoke would create one of these processes.

Further what we notice another property of this particular model of speeding up or achieving parallelization, is that each of these processes are isolated from each other. That is each process has its own set of instructions, data heap and the stack and these segments in this process are isolated from the other processes among these 4.

Next due to this isolation we requite inter process communication techniques, as we have seen in a previous video in order to communicate from with one process with another. So, we have seen that IPCs can be achieved in several ways. So, one way is by having a send and receives system calls and the operating system will help in creating a channel through which IPCs are achievable. The other ways are by shared memories, in which case shared memory is created in one of these processes again by use of system calls and this shared memory is it can then be used for IPCs between the processes.

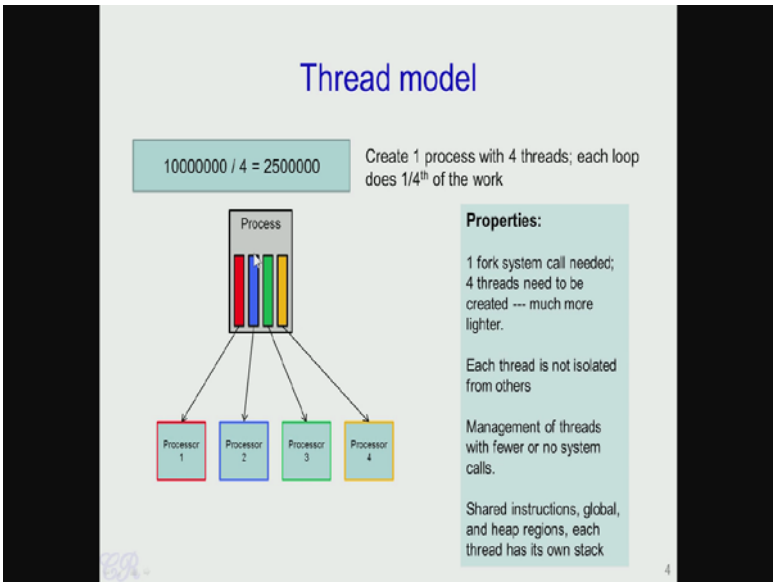
Further, since there are various stages in the process, such as the creation destroying or file open operation and so on. So, all these process management activities are done through system calls and also the last point as we mentioned, each process has its own isolated memory map comprising of instructions, data stack and heap. Now, what we notice among all these from these properties is that there are considerable amount of system calls involved in solving this problem in this particular way that is by having multiple processes to perform parallelization.

And as we have seen in an earlier video these system calls have considerable overheads. So, every time one of these processes invokes a system call it result in the operating system being executed being triggered due to a software trap and the operating system then determines what system call has invoked and the corresponding system call handler will be executed. So, as we have seen in an earlier video this entire process could be

quite considerable.

And other overhead comes from the fact that a large portion of these 4 processes are similar. For example, all these 4 processes would have the same set of instructions which they operate upon and also there could be the same array which they are accessing same global data which they are accessing and therefore, what we see is that there are a lot of duplication of instructions and data which is happening due to having multiple processes which execute a job in parallel. So, is there a way we can do better than this? Is there a way we can have a solution, where the amount of overheads present in parallelization can be reduced? So, in order to achieve this, what is used is the concept of threading.

(Refer Slide Time: 12:02)



So, with threading like before, we divide the entire space of 10 million into 4 parts, of 2.5 million each, but instead of creating 4 processes as we have done earlier, we will only create one process and within this process we create execution entity which we call a thread. So, each of these threads quite like the previous idea where of using parallelized processes, would have a loop which does quarter of the work that is each of these threads would find the sum of 2.5 million numbers.

So, pictorially this is what it is going to look like. So, we have 1 process. So, this 1 process would mean that it has one set of instructions a global data heap another segments, and within this process we have 4 executing context. So, these 4 executing context are the 4 threads and they execute within the particular process. And each of

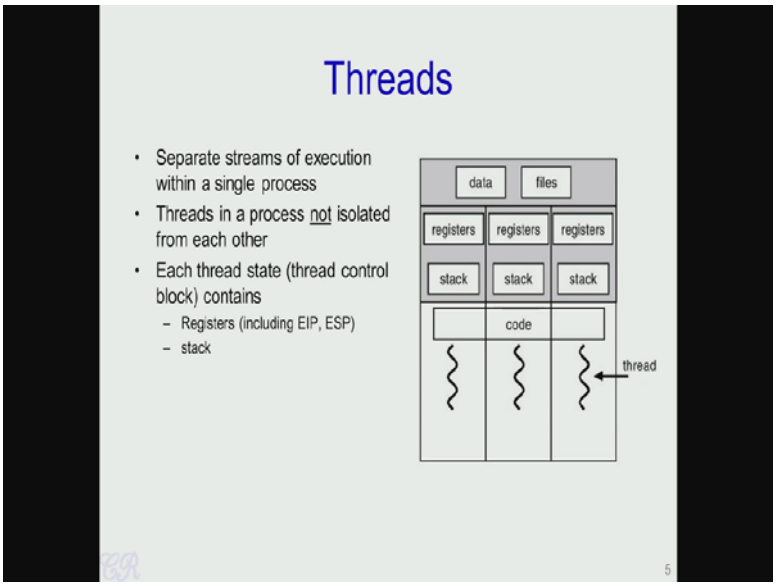
these threads are executed in a different processor or typically executed in a different processor for example, the red thread are goes into processor 1, and its executing here the blue threads goes into processor 2 green and yellow go into processor 3 and 4 respectively.

Thus we are able to achieve parallelization by using threads. So, this parallelization is quite similar with respect to the process parallelization what we seen in the previous slide. However, it has lower overheads. So, the property of this particular thread model of a parallelization is as follows; since we have created only one process over here. So, what we need require is just one fork system call to create this process. Now within this particular process in order create the 4 threads, we may use a library call such as P thread create, which may or may not require system calls.

Further, a major difference with respect to the process parallelization from the previous slide is that these threads are not isolated from each other. So, what this means is that since all this 4 threads execute in the same process space. So, these threads could access the processes segment such as the heap segment and the global data. Also it is possible that the instructions or the text segment in this process is also shared among the various threads. Now what is different or what distinguishes the different threads is that each of these threads will have its own stack and having its own stack will allow it to have own execution context. So, each of these threads will be able to use the stack to create local variables as well as to do the, to store information about function calls and so on.

Further management of these threads that is for example, waiting for another thread or exiting a thread, may not require to have system calls. So, what we will see now is these threads in more detail

(Refer Slide Time: 15:53)



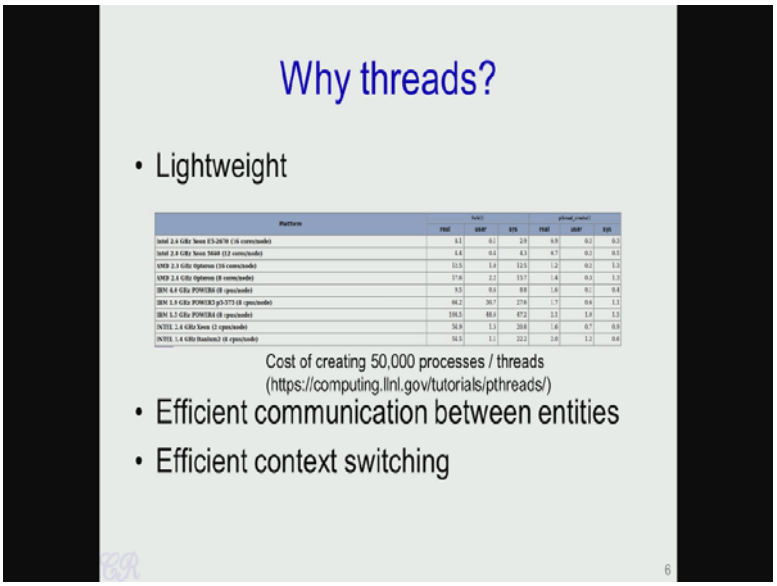
Let us look at it h with this particular figure. So, we have this single process over here and within this particular process we have multiple threads. So, these are the execution context of this process. So, these execution contexts all share the same code, and have access to the global data and heap of the process, and also we will have access to the same files which are opened by the process. So, what distinguishes 1 thread from the other is that each thread has its own stack which it uses for local variables and for function calls.

These threads are as mentioned execution context and each of these threads could be schedule to run on one of the processors present in the system. Or it could be also possible that these threads are scheduled on to the same processor I will beat at different time instance. Now each thread is executing almost independently of each other therefore, this registers over here signifies the execution context of a thread. So, as we have seen, each thread has its own execution context.

The execution context of this thread comprises of set registers like the EIP and the ESP register. Similarly the execution context of this thread comprises of the stack and registers values. So, these register values would help in the stopping and restarting of threads after or during the context switch.

Now, what is the major advantage of having threads over processes?

(Refer Slide Time: 17:52)



This is actually seen from this particular slide; essentially the most important difference comes on the fact that threads are extremely light weight compare to processes. To take an example let us have a look at this particular table, which shows the time required to create 50,000 processes using the fork system call and it compares this with creating 50,000 threads on exactly the same machines. So, let us take any of these machines as an example let us say the first one, and what we see and if we look in this particular column, which shows the time taken is 8.1 in order to create 50,000 processes using the fork system call on the other hand, creating 50,000 threads on exactly the same machine, only requires a time of 0.9.

o, we see that creating threads are much, much more efficient than creating processes in a similar way if we actually analyze other operations as well, we would understand that creating and managing threads are much lighter than managing processes. Another important advantage of threads is that they are they allow efficient communication between the execution entities. For example, we have seen in processes since each process is isolated from each other therefore, the only way that processes can communicate is by IPCs, and as we have seen IPCs involve quite a bit of system calls. So, for example, in the message passing IPC, where we use send and receive each send and receive invocation is a system call and significant overheads.

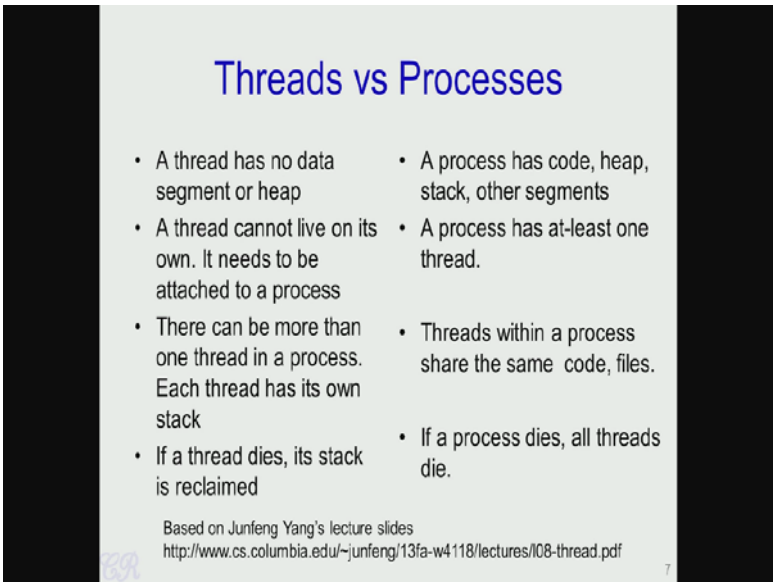
Similarly, for shared memory IPC s creating and managing the shared memories is again

done by system calls, again have overheads. On the other hand communication between two threads in a single process can be done extremely easily, the reason for this is that the global data of the process it shared among the various threads and this means that each thread has access to the global data in the process. Therefore, communication between threads in a single process can be easily achieved via the shared global data or similarly via the shared heap of the process.

Another big advantage of threads is that it results in efficient context switching. So, when we switch from one process to another, context switching time is considerable because the process data is quite large also we require the TLB we flushed and the page tables for the new process to become active and this process this new page table would then loaded into the TLB and so on. On the other hand, switching from one thread to another first of all requires a smaller saving of the context, because the context of the thread is much smaller than that of a process. Therefore, lesser amount of context needs to be saved when switching between threads therefore; threads context switching would be faster.

And other reason why context switching is faster in threads is that we are not changing page tables essentially since the threads execute from the same process, and each process is associated with a set of page directly and page tables. When switching from one thread to another, this page tables and page directories will remain the same. Therefore, there is there are no overheads of making another page table the active and therefore, the locality is maintained and cached entries in the TLB will still be valid.

(Refer Slide Time: 22:27)



This particular slide re iterates a lot of differences between processes and threads, and we will quickly go through this. So, essentially a process is an isolated execution entity which has its own code heap stack and other segments on the other hand, a thread does not have its own data segment and heap and essentially it adopts or it uses the processes data segment and heap. So, all threads within a particular process would use the same data segment and heap another difference is that a process as itself is a complete entity.

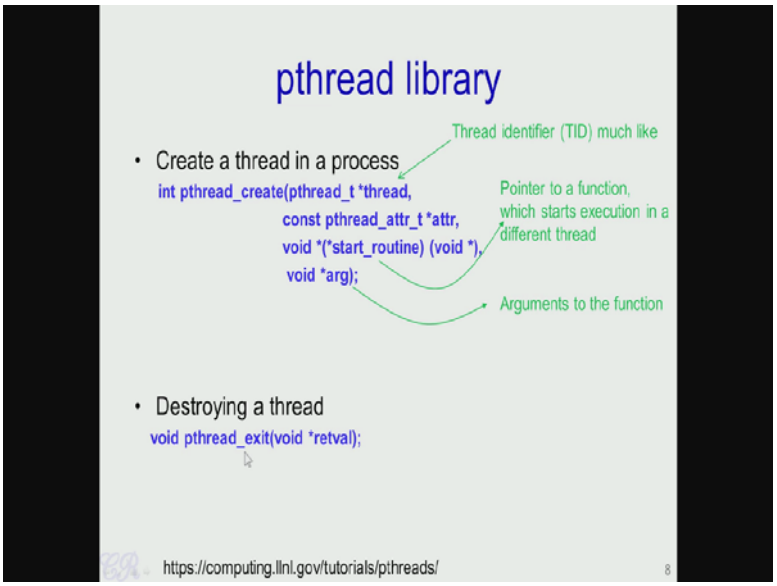
A process you could consider it as having single threads that is the single execution context and could survive on its own on the other hand; a thread cannot live on its own. So, it needs to be attached to a process. So, when a process terminates then, all the threads that are present in that process will also terminated. So, in other words without a process a thread cannot execute on the other hand, a process would just have a single thread and that is sufficient for the process to execute.

So, why do we say a single thread? This means any execution context of a process forms the single thread. So, threads within the process share the same codes and file as we seen in the earlier slide, but each thread has its own stack and this and this stack is reclaimed, when the thread dies. So, it is possible that when a thread terminates the remaining of the process will continue to execute. For example, if a process has a 10 threads and one of the threads terminates then the remaining nine threads will continue to execute.

On the other hand if a process terminates, then all threads within that process will also

die or will also terminate.

(Refer Slide Time: 24:50)



So, how do we create and manage threads? Now there are several libraries to do so, but what we will look in this particular video is a very popular library known as the P thread library. So, we will see some of the very critical or very important functions in the P thread library which are used to create and manage threads. So, let us see how we could create a thread in a process. So, in the P thread library a thread is created by using this function P thread create, which takes several arguments. So, it takes like 4 arguments and these 4 arguments are as shown over here.

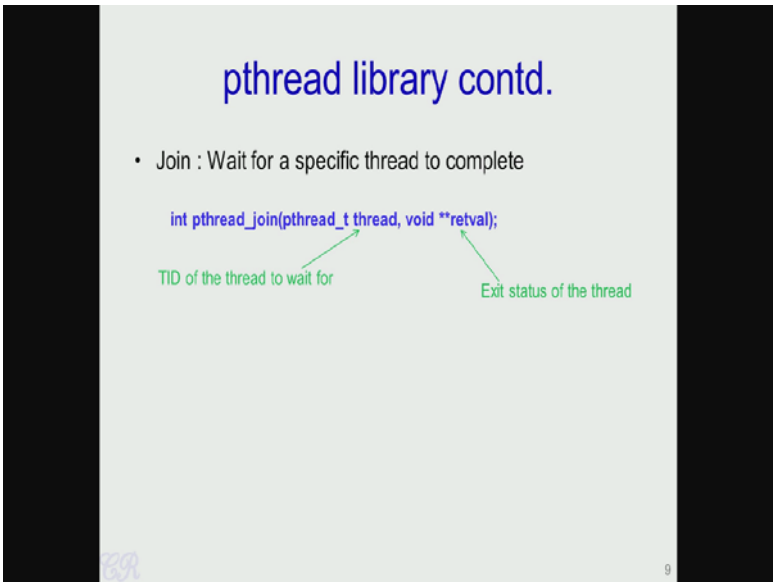
The first argument which is of defined a form which is the pointer to P thread underscore t is the thread identifier or TID. So, this is very similar to the process identifier or PID which we have studied in the process video. The second argument to P thread create is the attribute.

Through this attribute, you could specify several properties of the thread that you have been creating. So, what P thread would underscore create function would do is that when it is invoked, it is going to create a thread context or rather a new thread context in the process and it is going to start executing that new context, from this function specified over here. This third parameter specified as start underscore routine is a pointer to a function, which begins to execute in the new thread context.

This function will start to execute in a different thread the last argument for P thread create arg or a r g which is a pointer to the argument to this particular start routine function. So, in the next couple of slides, we will see examples of how to use P thread create. And also examples of the other functions in P thread.

Now, that we have created a P thread and let us say it has done its job. The next thing is to actually destroy a thread and this is done by this function called P thread underscore exit. So, P thread underscore exit will also pass the pointer to the return value in order to pass the return status of the thread. So, this is in many ways similar to the exit system call that we have seen in processes.

(Refer Slide Time: 27:28)

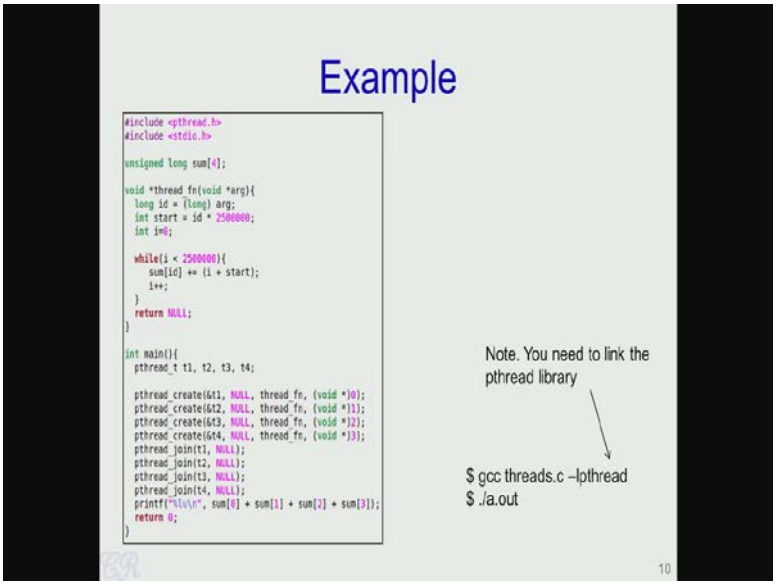


Now, another important P thread library function, is this particular function call P thread underscore join. So, through this a process or a thread could wait for a specific thread to complete. So, this is a much like the wait system call that we have seen with respect to processes. So, in P thread underscore join, what is specified is the TID of the thread to wait for that is the thread I d of the thread and what is obtained over here is the exit status of the thread.

This P thread underscore join will block the calling thread, until the thread specified by this TID passed over here exits and when the thread exits then P thread underscore join will wake up and it will able to read the exit status of the exited thread specified by the TID. So, given these particular three functions P thread underscore create P thread

underscore exit, and P thread underscore join. Let us see how we could use these P threads in order to parallelize our program of finding the sum of the first 10 million positive integers.

(Refer Slide Time: 28:48)



So, this is the code to do that. So, what this particular code does is that it creates 4 threads and each thread finds the sum of a different 2.5 million set of numbers, and these 4 threads execute in parallel and therefore, very quickly find the different 2.5 million numbers and then we add the results to get the sum of the first 10 million numbers.

Let us see this example in more detail. So, first of all we defined 4 variables of type P thread underscore t. So, these are the t 1, t 2, t 3 and t 4. So, these are going to be used to create the 4 threads and these specific would contain the thread ID or the TIDs now as we have seen in the earlier slide, creating the thread is invoke is done by this function call P thread underscore create. So, each of these P thread underscore create invocations would pass the pointer to one of this that is, t 1, t 2, t 3 and t 4. This then gets filled with the thread identifier, or the TID in of the newly created thread.

The second parameter that is the attributes of the thread is specified as null while the third parameter is function pointer which specifies the function where the thread should execute. So, over here the function pointer is a pointer to this function thread underscore f n, which is defined over here and it takes an argument void star that is specified over here arg. So, the last parameter which is passed to P thread underscore create, is the

argument which would be passed over here in the arg of this particular function.

So, in this way when P thread underscore create is invoked, it would create a thread context in the process and this thread context or this execution context will begin to execute from this particular function. Now the argument passed in this particular thread would be as specified in the last argument of P thread create in this way we have invoked 4, P thread underscore create and thus creating 4 threads, within the single process all these 4 threads have the same starting function that is thread underscore f n. And the thread IDs will be filled into this first parameter that is t 1, t 2, t 3, and t 4 respectively.

Now each of these 4 threads which will begin executing from this thread function, which is like if you think of like the main of the thread would see a different value of arg for example, thread 1 would have arg value of 0, thread 2 would have a arg value of 1, thread 3 would have an arg value of 2 and thread 4 will would have an arg value of 3. So, using this arg value which is then type casted to ID and then it is actually used to determine the start.

So, start is then use to determine which of the 2.5 million numbers that particular thread should add. For instance if the arg passed, arg in a thread is 0, then the value of ID after typecasting is 0 and the stacked value would also be 0. So, this thread with the ID 0 would add the first 2.5million numbers that is the number from 0 to 2.5million.

Now, in addition to this, what is defined is a global array called sum. So, this is unsigned long array of 4 elements. Now this sum is used to accumulate the sum for each thread. So, sum zero would contain the sum of the numbers from 0 to 2.5million which is filled by the thread 0 or thread 1 that is the t 1 thread.

Similarly, sum 1 will have the sum of the numbers from 2.5 million to 5 million and this some of one is filled by thread t 2 in similar way sum 2 and sum 3 would have the corresponding 2.5million sum and it is filled by t 3 and t 4. So, while this process is going on in the 4 threads, that is this while loop which takes considerably long time, is being executed by the 4 loops by the 4 threads independently.

What the main thread does is after creating the 4 threads it will invoke join. So, it invokes 4 joins corresponding to the 4 threads and what this join call would do is that it is going to block until the corresponding thread specified by the TID over here will exit.

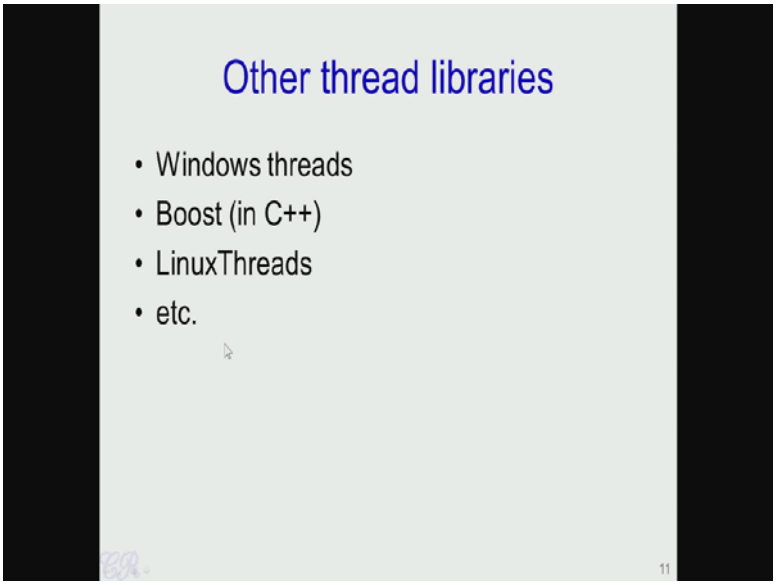
Thus when thread t 1 exits from this particular function this P thread underscore join of t 1 will wake up and it will complete executing. So, in this week all 4 join functions on completing will indicate that all the 4 threads have completed their execution.

Then after the 4 threads have exited, the elements of sum which contain the partial sums of the 2.5million numbers are taken and added to get the final result which is printed on to the using the printf. So, this final sum printed is the sum of the first 10 million numbers and therefore, we have seen that the large job of adding a numbers from 0 to 10 million is broken down into 4 a parts and each part is then worked is then computed by us, a thread to get a partial result which is then added on.

Now in order to execute and run this program we use something like gcc threads dot c which is the compilation and we also specify a, a minus l P thread which is the P thread library, which also needs to be linked into this particular executable. So, gcc would result in an - a dot out executable which is then going to be executed.

So, you can actually try out this particular program on a LINUX system and you could try to determine the amount of speed up that you would obtain by having different threads, to do the job compared to different processes and also a sequential program where you have only one process executing this job.

(Refer Slide Time: 36:04)



A besides the P threads there are several other libraries which you could possibly use, to

create multi threaded programs for example, the windows threads library which is a library used for Microsoft windows based applications, boost is a library for C++ then there is also another library known as Linux threads and so on. So, if we actually Google for thread libraries you will find a large list of such libraries which are present.

Thank you.