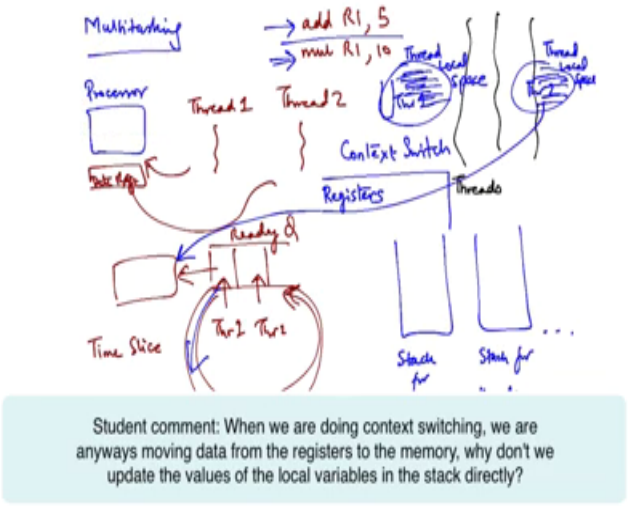
Introduction to Parallel Programming in OpenMp
Dr. Yogish Sabharwal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 14
Context Switching

(Refer Slide Time: 00:04)



Student comment: When we are doing context switching, we are anyways moving data from the registers to the memory, why don't we update the values of the local variables in the stack directly?

So, what is the problem when you throw out one thread and bring an another thread, what do you need to do? So, where is the dynamic state of the thread being maintained? There are hardware registers, where it has fetched data kept it in the registers it is doing some computation on it right, for data which is fetched from the memory the register contains copy of the data in the memory right?

At the end of the day this data is supposed to go back to the memory location right if some update is performed. So, now, each processor has some data registers and it is doing some computation let us say it was supposed to do some addition and then it was supposed to do some multiplication, it did the addition suddenly got thrown out. So, what are the actual assembly instructions? So, the assembly instructions will look something like this, add to register 1 the value 5 multiply register one with 10 something of this sort right, this maybe put the code looks like. Now suppose the thread was executing this instruction, add R 1 comma 5, right, it was adding 5 to r 1. So, it 5 to R 1 and then the operating system decided I need to schedule this out it is time slice got over.

So, another thread comes in, maybe that thread is using the register one for some other purpose. So, it is going to replace the contents of register one, and now when this particular thread comes back it comes back and starts executing multiply R 1 10, but now it has some other data, right. So obviously, we do not want this to happen.

So, whenever a thread is scheduled out right that is called a context switch, whenever a context switch takes place a copy is created for all the registers, and kept in some space which is thread local space right. So, there will be a thread local space for thread 1, there will be a thread local space for a thread 2 and so on. So, every trade will have some local space. So, now, what happens is that when this thread 1 is getting scheduled out, at that time all the registers will be copied into the local space of the thread. This is in the memory; you are copying the value from the registers to some memory location.

So, these is copied into this particular memory and then let us say thread 2 is getting scheduled in, it is going to pick up all the registers for thread 2 and put them into the actual hardware registers, and then it is going to start the execution of thread 2. So, that way is what happens is that a thread always maintains it is state right. So, whenever the context which happened this state is saved, and whenever it is scheduled back the state is copied back into the hardware registers.

Student: We maintain different stacks (Refer Time: 03:01).

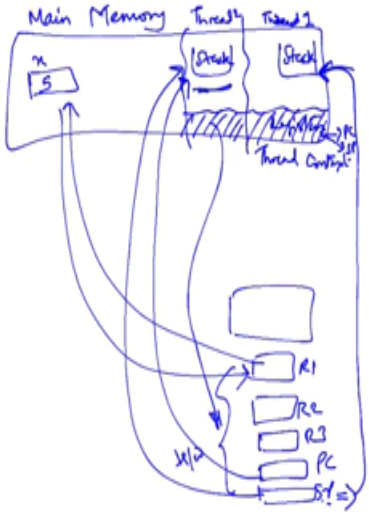
Yeah. So, you could possibly maintain the stack also in the thread local space right you could.

Student: So, we need to copy the contents of the stack also.

No, no, no, no, no, this is your main memory.

Student: (Refer Time: 03:15).

(Refer Slide Time: 03:13)



I have created some space for thread 1 right I am maintaining the stack for thread 1, and I maintain other things as well I will talk about what these things are. I maintain some space for thread 2, and I maintain the stack for thread 2. This is the memory right clear now there is the processor and the processor has registers, register R 1 register R 2, register R 3 program counter, stack pointer and so on right it has lots of resistors, limited set of registers right you cannot have arbitrary number of resistors. So, it is quite costly, it is very close to the processor right the arithmetic and logic unit directly works on registers when it over it has to do addition multiplication and so on.

But you have an integer x with value 5 in the main memory, and you want to let us say increment it you want to add one to it, you cannot go and add one to it in the main memory what do you have to do? You have to fetch it into a register let us say R 1 add one to it, and then put it back. You only have a limited set of registers maybe just 32 registers that first of all you have. Main memory is in gigabytes, when you switch the context from one thread to another, the main memory is still there right it is huge it has enough space to store data for thread 1 as well as thread 2 and you know 10 threads, and global variables and code and everything right it has enough space what is the problem is the register's.

The processor only works on the data that is residing in the registers; whenever it said get scheduled in it needs to know which stack it is working on. So, that is done by a

register called the stack pointer. The stack pointer just points to this particular stack, stack 1 that is all. All the contents of the stack are not in the registers, you just have a pointer to the stack which is kept in the register. So, now, when the thread gets scheduled out what do I need to save what state space do I need to save. So, there is no problem with the memory, where is the problem the problem is with the registers. I need to save the value of the resistors because one thread may require as many as 30 registers right, but I do not have 30 times 10 registers for even 30 for each thread, I do not have so, many resistors. So, that is why I need to save the state of all the registers somewhere in the memory, remember I had this space for each thread I will go and save it over here right. So, this is the thread context.

So, I will store the registers over here all right and now once I store the resistors over there this is a free for use right anybody can come and use them, because I have saved my state whenever I get scheduled next, I will copy it back I will copy this data back. So, that my registers are up to date right my state is the same, and then I will start executing the instructions.

So, what does that mean even the program counter goes here, even the stack pointer goes here in which instruction was getting executed that is also saved in the thread context, which stack is corresponds to it that is also stored in the thread context right. Now when thread 2 comes in, it is data from it is thread context gets moved into the registers; which means that now the stack pointer will be pointing to it is own stack and the program counter will be pointing to the instruction that it was executing last when it was scheduled out. Main memory I do not need to bother I do not need to save state of main memory I need to save the state of register, all right. So, the number of registers are small, but a context switch is still costly right because all these registers need to be backed up into the memory and the registers of the new thread need to be copied back into the registers.

So, this is a time consuming operation, it takes time to do this. So, that is why typically that the time slice is not so small right does it thread off in the time slice. I want to execute many in parallel; I want to give them short time slices. So, that you know I get the feeling that there are more tasks running in parallel right, but at the same time I cannot make it too small because otherwise the context switching is going to start taking turn yeah.

Student: When we are doing context switching (Refer Time: 07:57) they have been change in the (Refer Time: 08:00) copied into the memory.

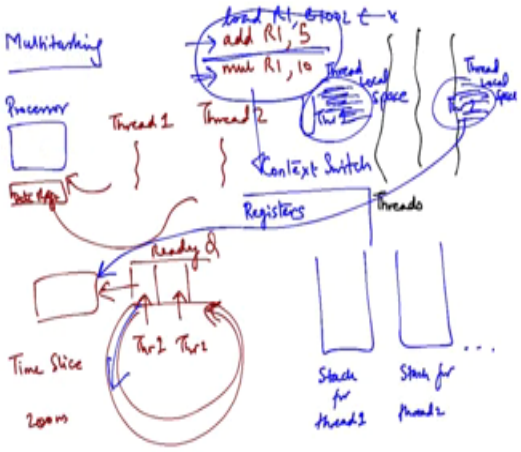
Student: (Refer Time: 08:01).

So, there are several things over here, but the most important point is that these are not directly related, I mean I do not know what is there in a register right the compiler knows that.

Student: (Refer Time: 08:33).

Look this is the work of the compiler, the compiler figures out that what am I supposed to keep in what register and it generates code of this sort add R 1 5 mul R 1 10. So, it knows whatever I kept in R 1, it has kept track when I said multiply x by 10 or add 5 to x, it kept track before this it will have an instruction load R 1 with some at the rate 1 0 0 2 which is actually the location for variable x right.

(Refer Slide Time: 08:51)

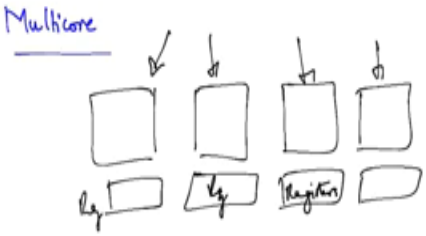


So, this is all being done by the compiler. At runtime I have no way of figuring this out when the thread is getting executed for it to figure out that you know what is stored in R 1 oh it is the variable x, it is not possible it has no clue whatsoever. The compiler has the complete picture and it has generated the code for that, but the point is at this point of time when I have scheduled this thread out I have no way of knowing up ahead whether a store instruction is coming or not right, and for me could try to figure that out is

challenging, there are lots of issues and doing that. So, to keep things simple all you do is that whenever thread goes out save all the registers, whenever it comes back and bring back all the registers and start executing exactly where you left off and that cook that causes no problem to the compiled code whatever just store, it was working on it find exactly the same value in the resister.

The only issue over here is yes it takes time to do the context switching to save all those resistor values and wing number. So, this is how multiple threads execute on a single processor, there is context switching that happens either based on time slices or the threads making blocking calls where they do not need the cp resources for some period of time and therefore, they are moved out. So, you get the impression that multiple threads are executing concurrently at the same thing, right.

(Refer Slide Time: 10:27)



Because there is just a single processor, which is educating them all let us come to multi core architectures. So, what happens in multi core architectures?

Student: (Refer Time: 10:31).

So, in multi cores you just have multiple processors each with it is own register set.

Student: (Refer Time: 10:34).

Let us say you have four cores and you are executing up to 4 threads, you will see that the operating system will just schedule them on separate course and there is no context switching that happens all of this is with a caveat there are some demon processes which wake up from time to time and do some stuff. So, they may schedule out your threads and you know do some work, but barring that you will find that you will get good utilization of all the cores with different threads being allocated different processes. So, if you have 4 cores and you are running up to four threads you will see very good performance.

Sometimes to avoid these demon threads and all we try to run off fewer threads than the number of cores. So, next time when you are trying to execute some parallel code, you should understand what the underlying architecture is now only then you will be able to figure out that what kind of performance gains you are going to get. Right is it a single core and if you are doing something competition heavy you are not going to see much gains with multiple threads, but if you have multi core you will start seeing the gains.