|  |  |
| --- | --- |
| **Name**: Shivam Tiwari | **Roll No**: 5117060 |

**Aim:** Implementation of multi-threading application.

**Theory:**

In computer architecture, multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system. This approach differs from multiprocessing. In a multithreaded application, the threads share the resources of a single or multiple core, which include the computing units, the CPU caches, and the translation lookaside buffer (TLB).

Where multiprocessing systems include multiple complete processing units in one or more cores, multithreading aims to increase utilization of a single core by using thread-level parallelism, as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and with CPUs with multiple multithreading cores.

The multithreading paradigm has become more popular as efforts to further exploit instruction-level parallelism have stalled since the late 1990s. This allowed the concept of throughput computing to re-emerge from the more specialized field of transaction processing. Even though it is very difficult to further speed up a single thread or single program, most computer systems are multitasking among multiple threads or programs. Thus, techniques that improve the throughput of all tasks result in overall performance gains.

**Advantages:**

If a thread gets a lot of cache misses, the other threads can continue taking advantage of the unused computing resources, which may lead to faster overall execution, as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle.

**Disadvantages:**

Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved and can be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.

Overall efficiency varies; Intel claims up to 30% improvement with its Hyper-Threading Technology, while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100% speed improvement when run in parallel. On the other hand, hand-tuned assembly language programs using MMX or AltiVec extensions and performing data prefetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading and can indeed see degraded performance due to contention for shared resources.

From the software standpoint, hardware support for multithreading is more visible to software, requiring more changes to both application programs and operating systems than multiprocessing. Hardware techniques used to support multithreading often parallel the software techniques used for computer multitasking. Thread scheduling is also a major problem in multithreading.

**Types of multithreading:**

Coarse-grained multithreading

The simplest type of multithreading occurs when one thread runs until it is blocked by an event that normally would create a long-latency stall. Such a stall might be a cache miss that has to access off-chip memory, which might take hundreds of CPU cycles for the data to return. Instead of waiting for the stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when the data for the previous thread had arrived, would the previous thread be placed back on the list of ready-to-run threads.

For example:

Cycle i: instruction j from thread A is issued.

Cycle i + 1: instruction j + 1 from thread A is issued.

Cycle i + 2: instruction j + 2 from thread A is issued, which is a load instruction that misses in all caches.

Cycle i + 3: thread scheduler invoked, switches to thread B.

Cycle i + 4: instruction k from thread B is issued.

Cycle i + 5: instruction k + 1 from thread B is issued.

Conceptually, it is similar to cooperative multi-tasking used in real-time operating systems, in which tasks voluntarily give up execution time when they need to wait upon some type of the event. This type of multithreading is known as block, cooperative or coarse-grained multithreading.

The goal of multithreading hardware support is to allow quick switching between a blocked thread and another thread ready to run. To achieve this goal, the hardware cost is to replicate the program visible registers, as well as some processor control registers (such as the program counter). Switching from one thread to another thread means the hardware switches from using one register set to another; to switch efficiently between active threads, each active thread needs to have its own register set. For example, to quickly switch between two threads, the register hardware needs to be instantiated twice.

Additional hardware support for multithreading allows thread switching to be done in one CPU cycle, bringing performance improvements. Also, additional hardware allows each thread to behave as if it were executing alone and not sharing any hardware resources with other threads, minimizing the amount of software changes needed within the application and the operating system to support multithreading.

Many families of microcontrollers and embedded processors have multiple register banks to allow quick context switching for interrupts. Such schemes can be considered a type of block multithreading among the user program thread and the interrupt threads.

Interleaved multithreading

The purpose of interleaved multithreading is to remove all data dependency stalls from the execution pipeline. Since one thread is relatively independent from other threads, there is less chance of one instruction in one pipelining stage needing an output from an older instruction in the pipeline. Conceptually, it is similar to preemptive multitasking used in operating systems; an analogy would be that the time slice given to each active thread is one CPU cycle.

For example

Cycle i + 1: an instruction from thread B is issued.

Cycle i + 2: an instruction from thread C is issued.

This type of multithreading was first called barrel processing, in which the staves of a barrel represent the pipeline stages and their executing threads. Interleaved, preemptive, fine-grained or time-sliced multithreading are more modern terminology.

In addition to the hardware costs discussed in the block type of multithreading, interleaved multithreading has an additional cost of each pipeline stage tracking the thread ID of the instruction it is processing. Also, since there are more threads being executed concurrently in the pipeline, shared resources such as caches and TLBs need to be larger to avoid thrashing between the different threads.

**Simultaneous multithreading:**

The most advanced type of multithreading applies to superscalar processors. Whereas a normal superscalar processor issues multiple instruction from a single thread every CPU cycle, in simultaneous multithreading (SMT) a superscalar processor can issue instructions from multiple threads every CPU cycle. Recognizing that any single thread has a limited amount of instruction-level parallelism, this type of multithreading tries to exploit parallelism available across multiple threads to decrease the waste associated with unused issue slots.

For example:

Cycle i: instructions j and j + 1 from thread A and instruction k from thread B are simultaneously issued.

Cycle i + 1: instruction j + 2 from thread A, instruction k + 1 from thread B, and instruction m from thread C are all simultaneously issued.

Cycle i + 2: instruction j + 3 from thread A and instructions m + 1 and m + 2 from thread C are all simultaneously issued.

To distinguish the other types of multithreading from SMT, the term "temporal multithreading" is used to denote when instructions from only one thread can be issued at a time.

In addition to the hardware costs discussed for interleaved multithreading, SMT has the additional cost of each pipeline stage tracking the thread ID of each instruction being processed. Again, shared resources such as caches and TLBs have to be sized for the large number of active threads being processed.

Implementations include DEC (later Compaq) EV8 (not completed), Intel Hyper-Threading Technology, IBM POWER5, Sun Microsystems UltraSPARC T2, Cray XMT, and AMD Bulldozer and Zen microarchitectures.

**Implementation specifics:**

A major area of research is the thread scheduler that must quickly choose from among the list of ready-to-run threads to execute next, as well as maintain the ready-to-run and stalled thread lists. An important subtopic is the different thread priority schemes that can be used by the scheduler. The thread scheduler might be implemented totally in software, totally in hardware, or as a hardware/software combination.

Another area of research is what type of events should cause a thread switch: cache misses, inter-thread communication, DMA completion, etc.

If the multithreading scheme replicates all of the software-visible state, including privileged control registers and TLBs, then it enables virtual machines to be created for each thread. This allows each thread to run its own operating system on the same processor. On the other hand, if only user-mode state is saved, then less hardware is required, which would allow more threads to be active at one time for the same die area or cost.

**Program:**

import threading

import time

# dummy function

def do\_something(id, sleep):

print(f"From thread {id}, I will sleep for {sleep} seconds")

time.sleep(sleep)

print(f"Thread {id} now awake and exiting...")

# pool/buffer of threads

threads = []

# initialize all the threads and add them to the buffer

for i in range(10):

threads.append(threading.Thread(target=do\_something, args=(i + 1, 1)))

# start the clock

start = time.perf\_counter()

# start running the threads

for thread in threads:

thread.start()

# join the threads

for thread in threads:

thread.join()

# stop the clock

end = time.perf\_counter()

# print the time taken

print(f"Time taken to execute the code: {round(end - start, 4)} secs")

**Output:**

E:\Desktop\College\College Exps\Sem 8\DC\Experiment 3>python exp3.py

From thread 1, I will sleep for 1 seconds

From thread 2, I will sleep for 1 seconds

From thread 3, I will sleep for 1 seconds

From thread 4, I will sleep for 1 seconds

From thread 5, I will sleep for 1 seconds

From thread 6, I will sleep for 1 seconds

From thread 7, I will sleep for 1 seconds

From thread 8, I will sleep for 1 seconds

From thread 9, I will sleep for 1 seconds

From thread 10, I will sleep for 1 seconds

Thread 1 now awake and exiting...

Thread 3 now awake and exiting...

Thread 4 now awake and exiting...

Thread 2 now awake and exiting...

Thread 7 now awake and exiting...

Thread 8 now awake and exiting...

Thread 6 now awake and exiting...

Thread 9 now awake and exiting...

Thread 10 now awake and exiting...

Thread 5 now awake and exiting...

Time taken to execute the code: 1.0126 secs