WPI-CS-TR-99-11 January1999

PThreads Performance

by

Bhupesh Kothari and Mark Claypool

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

PThreads Performance

Bhupesh Kothari
Mark Claypool
{bhupesh,claypool}@cs.wpi.edu
Computer Science Department
Worcester Polytechnic Institute

Abstract

Threads are very small compared with processes, thread creation is relatively cheap in terms of CPU costs. Threads are a powerful tool for designing multi-tasking applications. Multithreading (MT) is a modern programming paradigm which enables applications to move from strictly serial processing as in traditional heavy weight servers to concurrent processing. Especially, network-based information servers are often candidates for multi-threading where the individual operations are short, most operations are I/O bound,and a large number of logical copies of the server are operational at once; Database servers are a good candidates for multithreading for example. Typical operations involving threads in such an application environment include: thread creation, context switch between threads and synchronization, via mutexes and/or signals, between threads. This paper discusses the performance characteristics of Pthreads and gives Client Server Model as an example of application that can benefit from its use.

1 Introduction

1.1 Motivation

Just as multitasking operating systems can do more than one thing concurrently by running more than a single process, a process can do the same by running more than a single thread. Each thread is a different stream of control that can execute its instructions independently, allowing a multithreaded process to perform numerous tasks concurrently. One thread can run the GUI, while a second thread does some I/O, while a third one performs calculations. Threads are often called lightweight processes. Multiple threads share the same address space and its resources and communication can be achieved through shared data. Processes are heavy weight and do not share resources. Any communication between processes must be achieved through some kind of interprocess communication . Switching between processes is also expensive when compared to threads switching. Threads can increase performance in applications which performs operations that are likely to block or cause delays, such file or socket I/O.

1.2 Approach

As mentioned earlier, typical operations involving threads in such an application environment include: thread creation, context switch between threads and synchronization, via mutexes and/or signals, between threads. I have developed a number of micro-benchmarks involving threads to make timing estimates of the above basic operations. A Client Server Model is also developed to give a more realistic application to depict the benefits of using threads. In this model, one thread listens for new clients to attach, then creates a new thread to handle each client. The thread is dedicated to its client, doing work only for that client.

I provide a brief description of the POSIX thread features.

- Thread Creation: A single threaded program has a single thread running main(). In a MT program one of the libraries linked is the threads library. At creation time the operating system associates a single thread, the initial thread, to a process. The initial thread can create one or more new threads by calling the pthread_create() system service; the creator thread tells the operating system the starting address of the new thread and how much stack it will require. Threads create other threads, like UNIX processes create other processes via fork and exec. A simple call to create POSIX threads is pthread_create(&tid, NULL, start_fun, arg);
- Thread Exit: A thread is exited by calling the appropriate thread exit function or simply returning from the initial function. A simple call to exit POSIX threads is pthread_exit(status);
- POSIX Mutexes: A mutex is a system tool providing mutual exclusion: only one thread may hold a mutex at a time. To enter the mutex, a thread calls pthread_mutex_lock, to exit a mutex pthread_mutex_unlock is used. A thread trying to acquire a mutex held by another thread will be blocked until the mutex is unlocked. A typical use for mutexes is for protecting critical sections.
- Thread Yielding: To make an explicit call, sched_yield is used which causes a thread to yield its execution in favor of another thread with the same priority.

2 Related Work

There is a small number of high-level design strategies which have been discussed in several books. Few of them are

- Master/Slave: One thread does the main work of the program, creating other threads to help on some portion of the work.

- Client/Server (Thread per Request): One thread listens for requests, then creates a new thread to handle each request.
- Client/Server (Thread per Client): One thread listens for new clients to attach, then creates a new thread to handle each client. The thread is dedicated to its client, doing work only for that client.
- Client/Server (Producer/Consumer): Some threads create work requests and put them on the queue. Other threads take the work requests off the queue and execute them.

Bil Lewis and Daniel J. Berg have given the implementation of Client/Server(Threads per Request) model in their book *Multithreading Programming With PThreads*. This book also has the Producer/Consumer Version for Socket Server. While the Client/Server(Threads per Request) design has some positive aspects to it e.g., simplicity and directness, it also admits to some drawbacks. The cost of thread creation is not going to be significant unless the task itself is very short ($<$ 10 ms). Of more significance is that the programmer has no simple control over the number of threads running at any one time. Should there be a sudden increase in the incoming number of requests, there will be an equal spike in the number of threads, causing performance degradation due to excess number of threads competing for the same CPUs, memory, locks and other resources.

My implementation of Client/Server (Thread per Client) have a thread devoted to each client. The advantage of having a thread devoted to an individual client is that the thread can maintain state for that client implicitly by what's on the stack and in thread specific data.

Micro-benchmarks involving threads to make timing estimates of some of the basic operations are at the site *http://atddoc.cern.ch/Atlas/Notes/006/Note006-18.html*. This is the only site I could locate which says about the timing measurements of few operations like thread creation, mutex lock and context switch.All the measurements were performed on the RTPC/604 board with LynxOS 2.3.1. It gives the performance evaluation of PowerPC VME Boards Running a Real-Time UNIX System. A more comprehensive timing numbers and the best I could find is in the book *Multithreading Programming With PThreads* by Bil Lewis and Daniel J. Berg. My timing measurements differed very slightly from the numbers given in the book.

3 Implementation

Pthreads is the POSIX 1003.1c thread standard put out by the IEEE standards committee. This standard passed international Standards Organization (ISO) Committee Document (CD) balloting in February 1995 and got the IEEE Standards Board approval in June 1995.

A standard traditional server listens on a socket port and, when a message arrives, forks a process to service the request. Since a fork() system call is used in a non-threaded program, any communication between the parent and the child must be done through some sort of interprocess communication, something my program avoids.All measurements were done on

SunOS 5.6. Generic sun4u sparc SUNW,Ultra-1. The libraries which I used are lpthread, lthread, lposix4, lsocket.

3.1 Thread/Process Creation Time

The program for Thread creation takes the number of threads to be created as an argument from the command line. The program repeatedly creates the threads in a loop. The pthread_create(&tid, NULL, start_func, arg) creates a thread with a function to run and an argument to the function to run on. The thread goes in that function which does nothing but exits by executing the pthread_exit(). After each thread creation I do a join on the thread by using pthread_join(tid, NULL) so as to wait till the present thread exits before creating another one.

The program for Process creation works almost the same way as the thread creation program. The number of times the loop is to be run to create a child process is passed as an argument to the program. The system call fork() creates a child process. The child process runs a function which does nothing but exit by executing the exit() system call. The parent process waits for the child process by executing the wait() system call to exit before going again in the loop to create a child process.

For measuring the thread and process creation time I used the gettimeofday(&p,NULL) system call. The gettimeofday() function gets the system's notion of the current time. The current time is expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. The resolution of the system clock is hardware dependent; the time may be updated continuously or in clock ticks. The p argument points to a timeval structure, which includes the following members:
long tv_sec; /* seconds since Jan. 1, 1970 */
long tv_usec; /* and microseconds */

3.2 Thread/Process Destruction Time

The program for thread destruction takes the number of threads to be killed as an argument. In the thread function I calculate the time before it executes pthread_exit(). After the thread exits I again calculate the time. Since this is in a loop I add up the difference between these two times in each iteration.

The program for process destruction involved a lot more to do for measuring time since the processes do not share the same memory. I created a shared memory segment to store the time before the child process executes exit() system call. After measuring the time when it returns I calculate the difference between these two times and add this in each iterative loop. The call for creating a shared memory is:
(char *) shmcreate(key, 6*sizeof(char)))
which returns a pointer to this memory. To avoid mutual exclusion problem I used semaphores.

The function semcreate(SEMKEY, 1)) creates a semaphore and returns a integer. The calls semwait(sem); and semsignal(sem); are used to guard the critical region.

3.3 Locking

A single program executes N times the sequence pthread_mutex_lock and pthread_mutex_unlock : the total elapsed time T is recorded.
To calculate the semaphore post wait time, sem_post(&sem) and sem_wait(&sem) was executed N times and the total elapsed time T is recorded.

3.4 Context Switching

There are two ways of scheduling threads: process *local* scheduling also known as as *Process Contention Scope, or Unbound Threads* and system *global* scheduling known as *Global Contention Scope, or Bound Threads*. Both are defined only in POSIX. Two threads are created and call repeatedly sched_yield which forces the currently running thread to relinquish the CPU. The scheduler looks to see if there is another thread of the same priority. By default the threads are created of the same priority. So every time , the other thread gets scheduled. The time measured includes the execution of the yield routine and the time to switch the context.
To set the contention scope of the threads the following two functions are used:
pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS); /* unbound thread */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); /* bound thread */

A direct measurement of the context switch for processes was performed by using the sched_yield system routine. This system call allows the calling process to relinquish the CPU. Both the child process and parent process call this yield routine repeatedly for N times.

3.5 Shared Memory

Anything that can be done with threads, one can also do with processes sharing memory. Only thing is that it won't be as easy or run as fast. Programs that use two or more processes to access common data through are effectively applying more than one thread of control. However such process must maintain a complete process structure. The cost of creating and maintaining this large amount of state makes each process much more expensive, in both time and space, than a thread.

To calculate the time to access a shared global memory between threads is quite straight forward. A global buffer is created. The main thread writes data in that buffer and the creates a thread. The thread created reads the data from the buffer. The timing measurements are done before and after the read in the child thread. The measurement is done in a loop and the average value is calculated.
Calculation of the time to access the shared memory among processes required a lot more work. A shared memory is created and then a child process is created. The child process reads the

data from the shared memory written by the parent process. As before, the measurement is done in a loop and the average value is calculated.

4 Performance Analysis

The benefits and cost need to be weighed before proposing the class of applications that can benefit from multi-threading. Benefits include performance gains increased throughput, responsiveness, efficient use of system resources just to name a few. I looked at the general thread performance issues.

4.1 Creation Time

The thread creation time was measured iteratively from 1 to 100,000 times. The result are as shown in the Figure 1

From 1000 point onwards in the graph, the creation time is almost constant. I got the same output line for other threads timings measurements. Thus the time numbers I give is measured form 1000 times onwards iteratively and then taking the average of it. The thread creation time was measured at 67.9 μs. The process creation time was measured to be around 11000 μs. which is almost 150 times the creation time of a thread

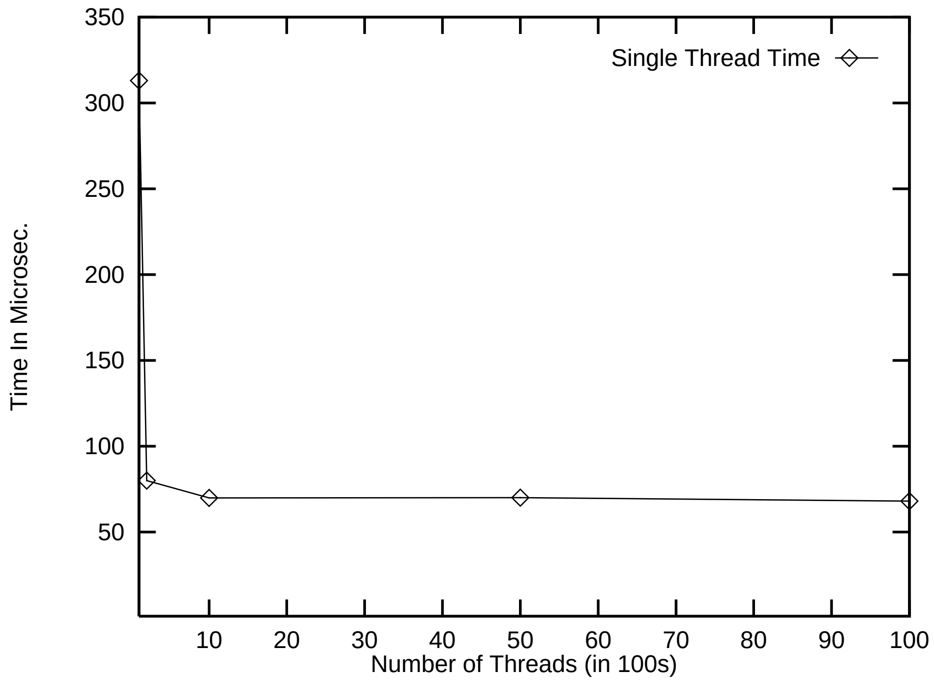

Figure 1: Microbenchmark Results-Average Creation Time of a Thread.

Test	num runs	time
Thread Creation	10	67.8961 μs.
Process Creation	10	11604.4 μs.

Table 1: Creation Time of Thread and Process.

4.2 Destruction Time

The destruction time microbenchmark for thread, creates 1000 to 100,000 threads and measures the average time for killing a thread. The time was measured at 49.779068 μs. The destruction time for processes required a lot more to do. The shared memory is updated before killing the process in each iterative loop.The loop was run 100 times. The time was measured at 5229.938030 μs.

Test	num runs	time
Thread	10	49.779 μs.
Process	10	5229.938 μs.

Table 2: Micro-benchmark Results-Destruction Time of Thread and Process.

4.3 Locking

Synchronization performance is critical in Multithreaded systems. The locking micro-benchmark repeatedly acquires and releases a single lock. The loop is run 10,000 to 100,000 times. The time required for each lock/unlock pair using pthread_mutex_lock/pthread_mutex_unlock was measured at 0.699 μs. Semaphore performance measurement was measured in the same way. The lock was acquired and released repeatedly. The loop was run 10,00 to 100,100 times. The time required for each lock/unlock pair using sem_post/sem_wait was measured at 2.898 μs. The result obtained are summarized in Table 3.

Test	num runs	time
Mutex Lock	10	0.699 μs.
Semaphore lock	10	2.898 μs.

Table 3: Microbenchmark Results-Lock/Unlock.

4.4 Context Switching

A thread context switch is very light weight. The yielding thread saves its register state and then the new thread loads its register status and continues. Process context switch is much more expensive, requiring all the current registers to be stored in the process structure for the yielding process and then storing all the register values from the process structure for the new process to be loaded into CPU's registers.

The thread context switch microbenchmark creates two threads which call repeatedly sched_yield which forces the currently running thread to relinquish the CPU. The time measured for unbound threads was measured at 23.954 μs. The time measured for bound threads was measured to be 17.925 μs. which is a little less than the unbound threads.
The process context switch microbenchmark gave the time for context switching between two processes to be 21.624613 μs.

Test	num runs	time
Bound threads	10	17.925 μs.
Unbound threads	10	23.954 μs.
Process	10	21.62 μs.

Table 4: Microbenchmark Results-Yield.

4.5 Shared Memory

The shared memory microbenchmark for threads creates a child thread which reads the data written by parent thread from the global buffer. The access time for different data sizes was measured.
The shared memory microbenchmark for process calculates the access time for different data sizes in the same way. The Figure 2 shows the result obtained. To access the global buffer between processes was found to be more than that of between threads. In addition, the inherent separation between processes requires a much greater effort in implementing to communicate among the different processes. By using threads for communication instead of processes, the program will be easier to debug and can run faster.

5 Client/Server Model

A server needs to handle numerous overlapping requests simultaneously. For e.g., DBMS servers require large number of requests that require the server to do some I/O, then process the results and return answers. Completing one request at a time would be very slow.
The traditional server which forks a process for each client request or connection can handle more than one client request simultaneously but the cost and benefit of this needs to be evaluated.
The Client/Server(Thread per Client) model listens for new clients to attach, then creates a new thread to handle each client such as in Figure 3. The thread is dedicated to its client, doing work only for that client. Where the traditional server would need to communicate among its

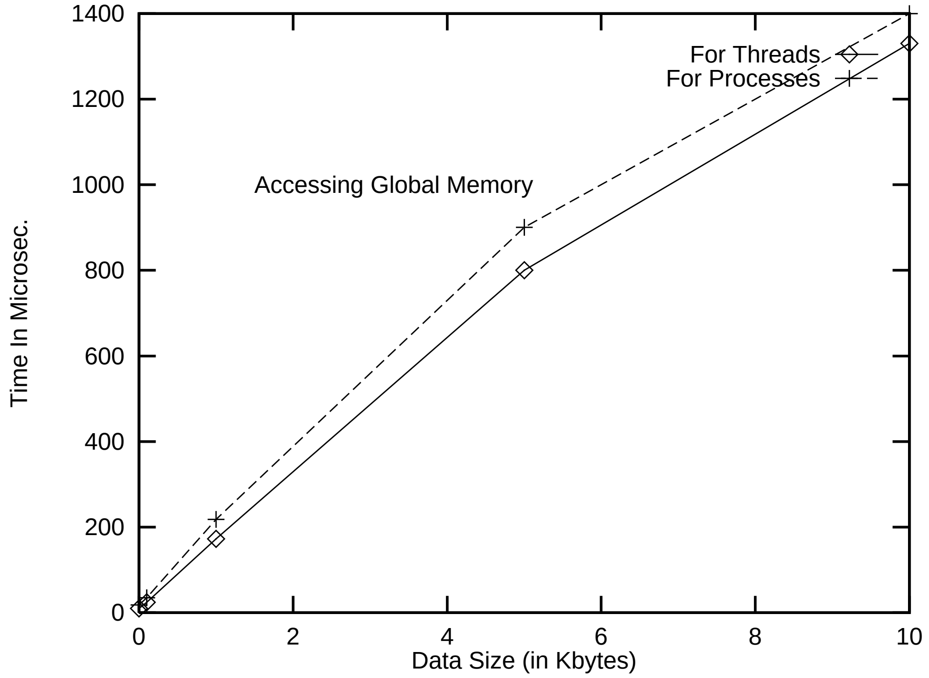

Figure 2: Access Time for shared memory between threads and processes.

processes through traditional interprocess communications facilities(e.g., pipes or sockets), the threaded application can communicate via the inherently shared memory of the process. The threads in the MT process can maintain separate connections while sharing data in the same address space.

A standard socket server listens on a socket port and, when a message arrives, forks a process to service the connection. The Multi-threaded server first sets up all the needed socket information. The server then enters a loop, waiting to service a socket port. When a message is sent to the socket port, the server creates a new thread to handle the connection on this file descriptor. That thread remains dedicated to the client as long as the client wants to have.

The newly created thread listener thread receives requests on this file descriptor until the string "Cntrl-C" come across. For each connection request, the listener thread creates a new thread to handle it. The worker thread process the request in the function server(), and then sends a reply back across the file descriptor.

The overall scenario works as follows. The client opens a connection to the server. The client reads from standard input and writes the data to the server. The server in turn reads the data from the connection, adds some transaction information and writes the resulting data back to the client. The client reads the resulting data (from the server) and writes it onto standard output. The same client program interacts with the multi-threaded server and the traditional heavy weight server. No changes were needed to make to the client program to make it interact with the two servers.

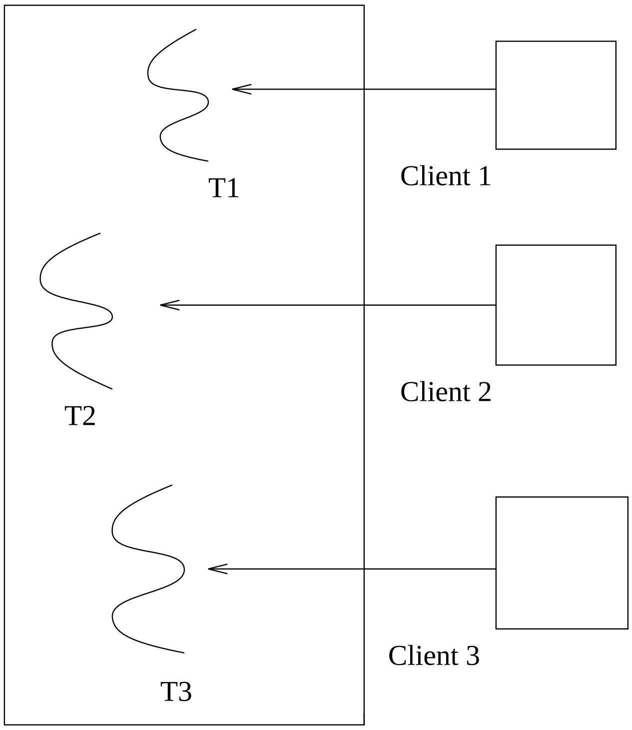

Figure 3: Clients being handled by different Threads.

The client side program sends data to the server for each data size and file descriptor I request on the command line. It waits for each reply from the server. The client code can also be run from different machines by multiple users.

I measured the time it takes to send a data from the client side and to receive it back from the server. My request on the command line is of the type: client 1000 6565 crane , where 1000 is the data size of 1000 bytes, 6565 is the socket number and 'crane' is the host-name where the server program is running. The client could be running on a different machine or on the same machine. I ran the client and server on different machines in the experiment.

It is quite possible that client makes a connection with the server for a very short duration. In cases like this , the time would be a lot different than if the connection had been for a long time. I measured the time for both short and long connections.The results when the client sends the data to Multi-threaded server and Traditional Heavy Weight server and makes a short connection, are shown in Figure 4.

It is clear from the Figure 4 that it took more time when the client send the data to Heavy Weight server. The results for long term connections are as shown in Figure 5. The results shows that for long term connections the low level measurements doesn't matter. In-fact the line in the Figure 5 for Heavy Weight server drops down a little below the line for the MT server. For long term connections the client sends the data to the server 50 times and then the average value is calculated. The time drops down very fast in the beginning but then it

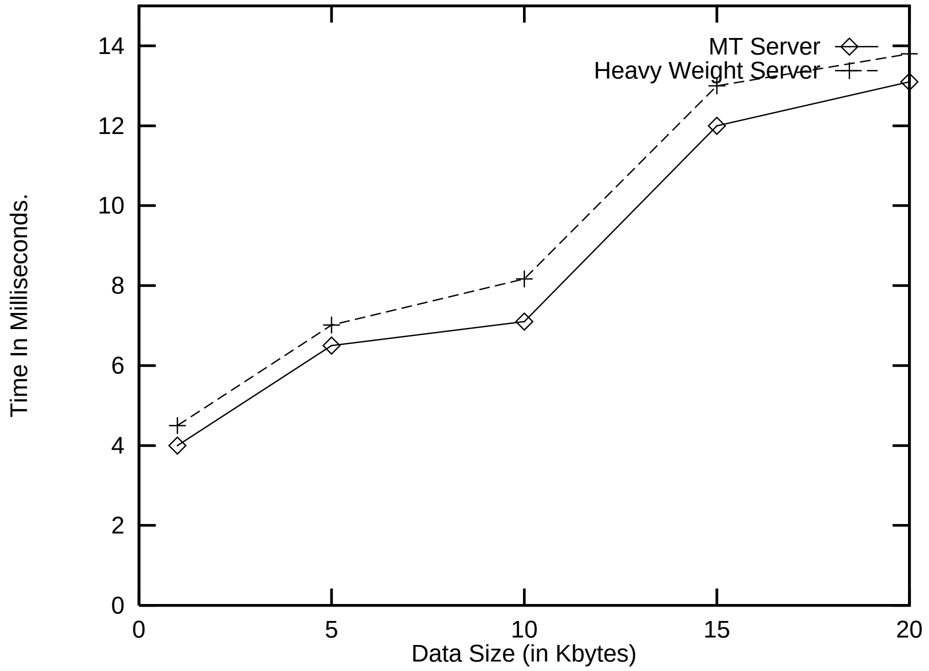

Figure 4: Response Time for Short-Term Connections.

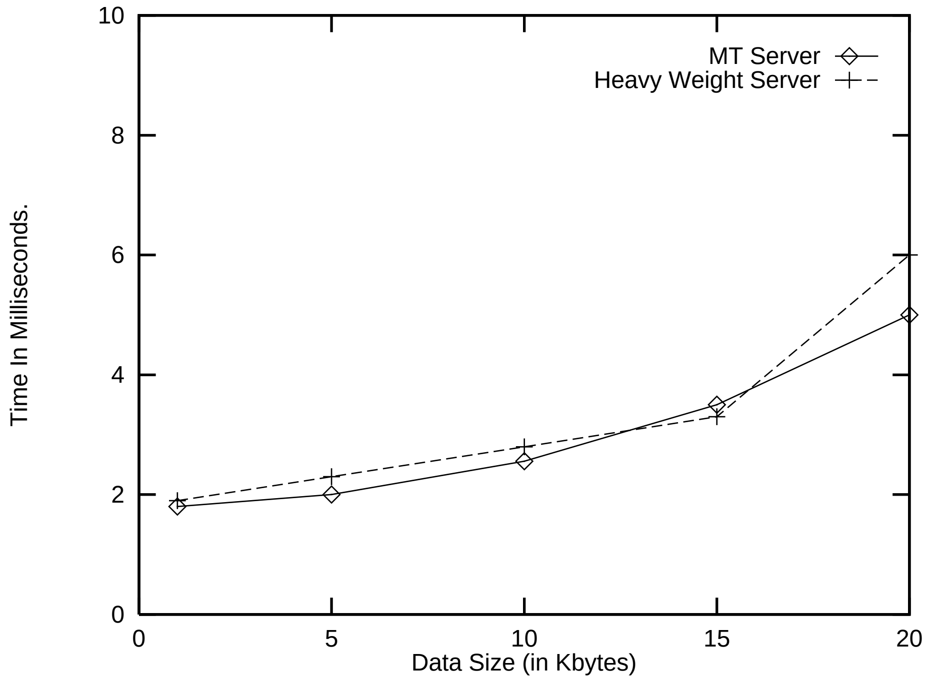

Figure 5: Response Time for Long-Term Connections.

becomes almost constant after a while. For e.g., time taken to send a 10K data from the same machine(server and client both running on same machine)took 1.469 μs first time, but then after a while it became almost stable around 0.45 μs. Thus for response time issue, MT server has the benefit only if the clients makes connections for very short duration. From Figure 4, the time to send and receive a 1K bytes to a Heavy Weight Server takes around 4.7 milliseconds whereas to a Multi-threaded server takes around 4.0 milliseconds. The difference is not of the same magnitude as the difference between thread and process creation time but still the time in case of Heavy Weight server is more. In addition this difference was almost constant for the whole experiment as is clear from the Figure 4. For long term connections the difference was almost negligible. Thus it looks that for long term connections the low-level measurements does not matter, only network overhead matters.

6 Conclusions

The performance characteristics of Multi-threaded server (Thread per Client) are well suited for Client/Server applications where the client makes short connections most of the times.WEB servers could be such application where this model could benefit. In addition, the major conclusions I drew from my low-level measurements are:

- Process creation time is of the order of 150 times magnitude more than the Thread creation time.
- Process destruction time is of the order of 100 times magnitude more than the Thread destruction time.
- Mutexes are faster than Semaphore
- Thread context switch is faster than Process context switch.
- To access the shared memory between processes required a lot more work then that of threads. In addition it took more time to access the shared memory between processes then among threads

7 Future Work

A lot more cost and benefit issues need to be evaluated for both Multi-threaded and Traditional servers so that the class of applications that can benefit for their use can be categorized. Irrespective of the connection, whether short or long, the cost of creating and maintaining the large amount of space, as each process must maintain a complete process structure, will be a lot expensive in terms of space , than a Multi-threaded server. If a lot of clients are connected to the server for a long time, the machine could become slow over the time and could degrade the performance. Issues like this needs to be evaluated. In addition the Client/Server model could be extended to the applications like WEB servers or a Rating server ,which is used in Collaborating Filtering Technique,where it just sends back the requested data and then closes the connection.

References

[1] M. Seltzer, Scheduler Activations on BSD: Sharing Thread Management Between Kernel and Application, *Technical Report TR-31-95, 1995*

[2] B.D. Marsh, M.L. Scott, T.J. LeBlanc, and E.P. Markatos: First-Class User-Level Threads, 13th ACM Symposium on Operating Systems Principles, *October 1991*

[3] David Keppel, Tools and Techniques for Building Fast Portable Threads Packages, University of Washington, *Technical Report UW-CSE-93-05-06.*

[4] Comparison of POSIX pthreads and Solaris threads, SunSoft, *http://www.sun.com/workshop/threads/posix.html*

[5] Tom Wagner, Don Towsley, Getting Started With POSIX Threads, University of Massachusetts at Amherst, *http://128.119.41.247/~wagner/threads_html/tutorial.html*

[6] A.D. Birrell, An Introduction to Programming with Threads, *SRC Research Report 35, DEC,January 1989*

[7] Bil Lewis, Daniel J. Berg, Multithreading Programming with Pthreads, Sun Microsystem Press, A Printice Hall Title