"APPLICATIONS OF THREADS"

PROJECT REPORT

Submitted for the course: CSE2005 OPERATING SYSTEMS

By:

P.G.S.KOUSHIK (19BCE0062)
N.ANAND SUBBA RAJU(19BCE0264)
N.L.NARASIMHA RAJU(19BCE2247)

Slot: F1

Name of faculty:

NAGARAJA RAO A

(SCOPE)

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

TABLE OF CONTENT:

1.ABSTRACT

Our main objective is to carefully understand the various applications of threads, specifically that of multithreading. To study how these are used and to be able to use these methods ourselves. We have given five problems in which threads are used to understand. And then, to solve a new problem using threads.

These five problems give us a good idea on how to use threads and how these problems are approached. We chose to discuss the Reader/Writer, Dinning Philosopher, Producer and Consumer, Doctor-Patient problem and Cigarette-Smokers problem.

The problem that we chose can be solved using the concept of the Cigarette smoker's problem and Semaphores. We have used C language to solve the given problem.

2. INTRODUCTION

Programmers use multiple threads of control for a variety of reasons: to build responsive servers that interact with multiple clients, to run computations in parallel on a multiprocessor for performance, and as a structuring mechanism for implementing rich user interfaces. In general, threads are useful whenever the software needs to manage a set of tasks with varying interaction latencies, exploit multiple physical resources, or execute largely independent tasks in response to multiple external events.

We believe the only way to significantly improve performance is to enhance the processor's computational capabilities. In general, this means increasing parallelism— in all its available forms. At present only certain forms of parallelism are being exploited. Current superscalars, for example, can execute four or more instructions per cycle; in practice, however, they achieve only one or two, because current applications have low instruction-level parallelism. Placing multiple superscalar processors on a chip is also not an effective solution, because, in addition to the low instruction-level parallelism, performance suffers when there is little thread-level parallelism. A better solution is to design a processor that can exploit all types of parallelism well.

3.OBJECTIVES:

Manufacturing of a car requires 3 important parts:

1. Engine
2. Chassis
3. Electrical supply system

The headquarters of the company can accommodate only 2 of the 3 resources at a time.

The 2 resources are supplied to the headquarters for testing and then the selected resources of these 2 are transported to the country where the 3rd resource is available.

After the resources are received at the place of the 3rd resource, the car's body is assembled and tells the headquarters that the car is ready to be sold in the market.

4.EXISTING ALGORITHS:

SIMULATANEOUS MULTITHREADING:

Simultaneous multithreading is a processor design that meets this goal, because it consumes both thread-level and instruction-level parallelism. In SMT processors, thread-level parallelism can come from either multithreaded, parallel programs or individual, independent programs in a multiprogramming workload. Instruction-level parallelism comes from each single program or thread. Because it successfully (and simultaneously) exploits both types of parallelism, SMT processors use resources more efficiently, and both instruction throughput and speedups are greater.

Simultaneous multithreading combines hardware features of wide-issue superscalars and multithreaded processors. From superscalars, it inherits the ability to issue multiple instructions each cycle; and like multithreaded processors it contains hardware state for several programs (or threads). The result is a processor that can issue multiple instructions from multiple threads each cycle, achieving better performance for a variety of workloads. For a mix of independent programs (multiprogramming), the overall throughput of the machine is improved. Similarly, programs that are parallelizable, either by a compiler or a programmer, reap the same throughput benefits, resulting in program speedup. Finally, a single-threaded program that must execute alone will have all machine resources available to it and will maintain roughly the same level of performance as when executing on a single-threaded, wide issue processor.

Equal in importance to its performance benefits is the simplicity of SMT’s design. Simultaneous multithreading adds minimal hardware complexity to, and, in fact, is a straightforward extension of, conventional dynamically scheduled superscalars. Hardware designers can focus on building a fast, single threaded superscalar, and add SMT’s multithread capability on top.

THREAD PRIORITY PROBLEM IN SMT:

Threads are meant to be parallel, but notices that threads interference by memory access can serialize the schedule. Their solution is to have a parallelism-aware batch scheduler that preserves memory-level parallelism across scheduled threads. Furthermore, the scheduler prevents thread starvation, and promotes fairness by priority.

The solution of for the thread priority problem in SMT processing systems is to design a new processor and scheduler in hardware which understand thread priority. Their real-time processor is called Responsive MultiThreaded (RMT) and it is also designed to control the instructions per cycle of each thread, which speeds up or slows down the thread throughput. This method of observing and manipulating the IPC (instructions per cycle, not to be confused with interprocessor communication) to achieve the throughput corresponding to the thread priority is called IPC control.

The response time of the OS was studied and they found that static scheduling assumptions can be very different from the real behavior due to unresponsiveness of the OS. The schedule can be stretched out in quite undesirable ways when a task on the critical path of the schedule is waiting for the OS to respond. The problem is pronounced when many cores are depending on a centralized OS. The jitter in OS response time can have a variety of sources including ”user space processes, kernel threads, interrupts, SMT interference and hypervisor activity”.

Co-scheduling, the idea of scheduling tasks simultaneously across the cores is one way to reduce the observed OS jitter. Another possibility is the use of microkernels at the various cores. Many complex sources of jitter are not easily avoided. For example a direct mapped cache shared between cores will cause interference that just cannot be programmed away. This interference can also happen between threads that are running together in a superscalar processor, and during process migration. By using the extra thread bandwidth in the system, and also the unused cores, proposes that reducing jitter is possible. The reduction of the number of active kernel threads, intelligent interrupt handling, and thread priority tuning are also used to attack the problem of OS jitter. Their implementation was specific to the Power Architecture and the Linux OS.

Performance isolation is the opposite of thread interference, where co-runners disrupt the schedule in unanticipated ways due to hardware resource conflicts. A novel scheduling algorithm for the OS of a multiprocessor is described in. Interestingly, this paper explains how co-runners can interfere in terms of memory accesses such as cache reservation. The described approach preserves priority enforcement by the OS, and provides QoS capability. The solution is software based and it is aware or the cache allocation policy. The cache allocation is not guaranteed to be fair, instead the application runs just as quickly as it would if there had been fair cache allocation.

SMT vs. MULTIPROCESSORS:

SMT obtained better speedups than the multiprocessors (MP2 and MP4), not only when simulating the machines at their maximum thread capability (eight for SMT, four for MP4, and two for MP2), but also for a given number of threads. At maximum thread capability, SMT's throughput reached 6.1 instructions per cycle, compared with 4.3 for MP2 and 4.2 for MP4.

Speedups on the multiprocessors were hindered by the fixed partitioning of their hardware resources across processors, which prevents them from responding well to changes in instructionand thread-level parallelism. Processors were idle when threadlevel parallelism was insufficient; and the multiprocessor's narrower processors had trouble exploiting large amounts of instruction-level parallelism in the unrolled loops of individual threads. An SMT processor, on the other hand, dynamically partitions its resources among threads, and therefore can respond well to variations in both types of parallelism, exploiting them interchangeably. When only one thread is executing, (almost) all machine resources can be dedicated to it; and additional threads (more thread-level parallelism) can compensate for a lack of instruction-level parallelism in any single thread.

To understand how fixed partitioning can hurt multiprocessor performance, we measured the number of cycles in which one processor needed an additional hardware resource and the resource was idle in another processor. (In SMT, the idle resource would have been used.) Fixed partitioning of the integer units, for both arithmetic and memory operations, was responsible for most of MP2's and MP4's inefficient resource use. The floating-point units were also a bottleneck for MP4 on this largely floatingpoint-intensive workload. Selectively increasing the hardware resources of MP2 and MP4 to match those of SMT eliminated a particular bottleneck. However, it did not improve speedups, because the bottleneck simply shifted to a different resource. Only when we gave each processor within MP2 and MP4 all the hardware resources of SMT did the multiprocessors obtain greater speedups. However, this occurred only when the architecture executed the same number of threads; at maximum thread capability, SMT still did better.

The speedup results also affect the implementation of these machines. Because of their narrower issue width, the multiprocessors could very well be built with a shorter cycle time. The speedups indicate that a multiprocessor's cycle time must be less than 70% of SMT's before its performance is comparable.

5.EXISTING PROBLEMS

There are five major examples we have described:

1. Reader / Writer
2. Dining Philosopher
3. Producer and Consumer
4. Doctor-patient problem
5. Cigarette – smokers problem

Each are classic problems for synchronization.

- **Reader/Writer problem**

When one writer is writing into the data area, another writer or reader can’t access the data area for its purpose.

When one reader is reading from the data area, other readers can also read but writer can’t write.

- **Dining Philosopher Problem:**

There are 5 philosophers, who share a round table, which has one chair for each person. There are 5 chopsticks placed in between them and a bowl of rice in middle of the table. When a philosopher gets hungry, he grabs the nearest two chopsticks and starts eating, if the person beside him is already eating, then he has to wait until he gets the chopstick from that person.

- **Producer Consumer Problem:**

A producer produces items which are consumed by the consumers. When the producer has not yet produced items, the consumer can’t buy it.

When the consumer has not yet consumed all the products in the stack, then producer has to wait until a place gets empty in the stack.

- **DoctorPatient problem:**

In clinics, patients are given chairs where they can wait until they go meet the doctor.

If the number of patients are more, then the latest patients have to stand and wait until a patient comes out of doctor’s room and a patient enters the doctor’s room.

Later, when a chair empties, the waiting patient can take the seat.

- **Cigarette smoker’s problem:**

There are three requirements to make the cigarette, and each smoker will carry any one ingredient of his choice.

Later, two random requirements out of three are kept on the table and the lucky person who has the other ingredient can make the cigarette and smoke. Within that time, another two ingredients can be kept on the table and the process continues on.

All the above discussed problems are being solved by using multi-threading concept and using locks.

6.PROPOSED SOLUTION (PSEUDOCODE)

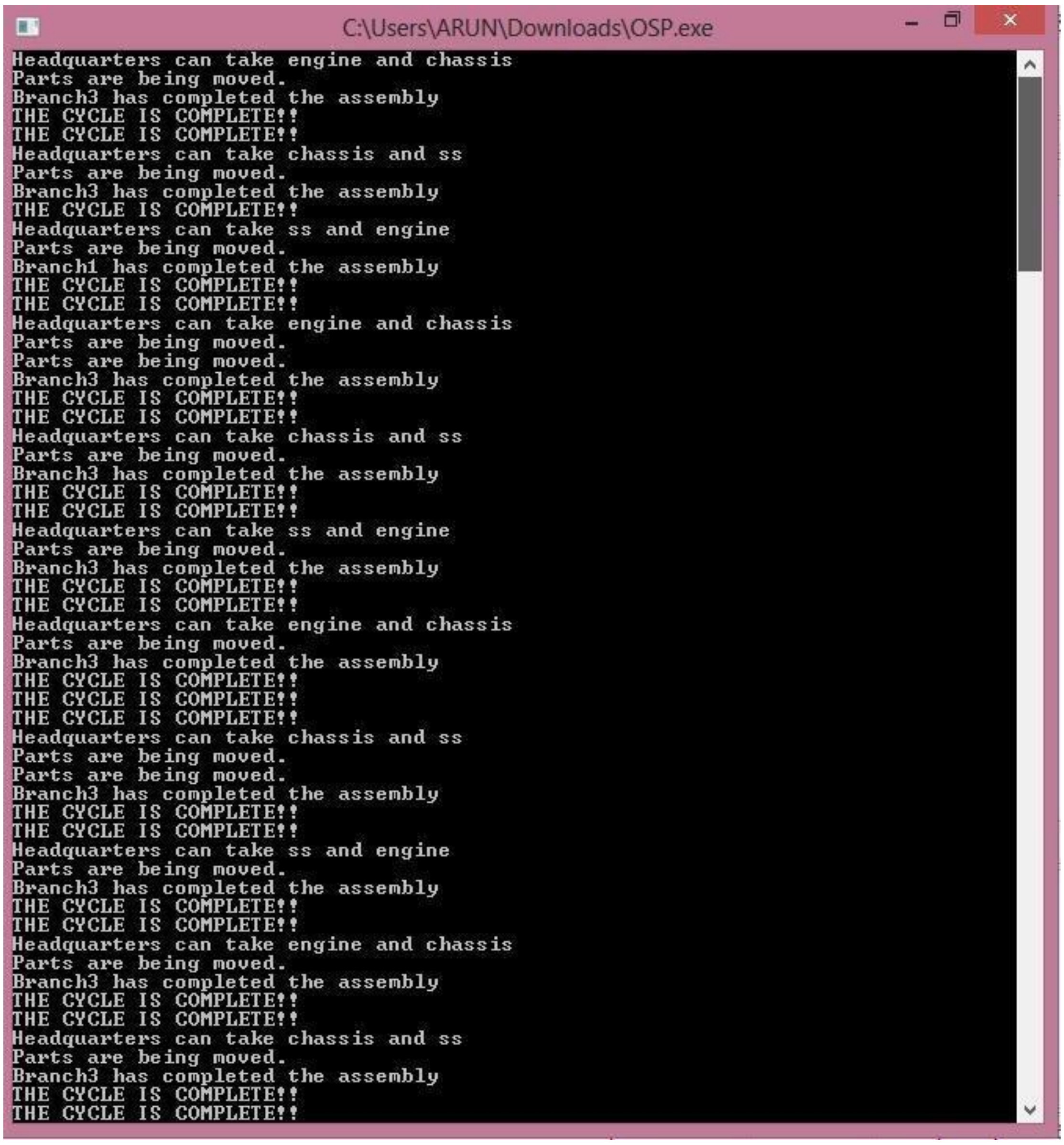
- Start.
- Initialize the clear_cycle=1 and part[2],generated=0.
- Declare the parts of the car as ‘engine’, ‘chassis’, ‘ss’.
- Initialize the semaphore sem_t ready.
- Assign the ids from 0 to 3.
- Declare the function void * gen branch(void*arg) which generates the assembly branch.
- In the above function declare i,j,k=0 .
- Repeat the step 7 until while(1).
- Declare sleep(1) so that when the loop runs the next time it puts threads to sleep for 1 sec.
- Declare sem_wait( ) which will decrement the value pointed by the semaphore when greater than zero.
- So, sem_wait(&ready) and if (clear_cycle= =1) then
- i=k, increment the value of i by 1 and assign it to j.
- Assign k=j.
- Declare sem_post( ) which increments the value pointed by the semaphore.
- Print “ Parts are being moved”.
- Declare the function void*assembly(void*arg) which does the assembly in assembly branch.
- In the above function declare flag=1.
- Repeat the step 16 until while(1).
- Check if the generated branch is not the any two other branches.

- Print “ Branch has completed the assembly”.
- Check if clear_cycle==1 and flag= = 0 then print “ THE CYCLE IS COMPLETE”.
- Declare sem_init(&ready,0,1) which will initialize the unnamed semaphore.
- Return 0 and stop.

7.PROPOSED CODE

```
#include<stdio.h>

#include<sys/types.h>

#include<semaphore.h>

#include<pthread.h>

int clear_cycle=1;

int part[2],generated=0;

char       *parts[]={"engine","chassis","ss"};

sem_t ready;

int    place    id[5]={0,1,2,3};

void *gen_branch(void *arg)

{

int i,j,k=0;

while(1)

{

sleep(1);

sem_wait(&ready);

if(clear_cycle==1)
{

i=k;

j=i+1;

if(j==3)

j=0;

k=j;

part[0]=i;

part[1]=j;

printf(”Headquarters can take %s and %s\n”,parts[i],parts[j]);

generated=1;//cuz the assembly branch is generated

clear_cycle=0;//this will be an old cycle

}

sem_post(&ready);

printf(”Parts are being moved.\n”);

}

}

void *assembly(void *arg)

{

int p_id=*((int*)arg);
int flag=1;

while(1)

{

sleep(1);

sem_wait(&ready);

if(clear_cycle==0)

{

if(generated && part[0]!=p_id &&part[1]!=p_id)

{

printf(”Branch %d has completed the assembly\n”,p_id);

clear_cycle=1;

generated=0;

flag=0;

}

}

sem_post(&ready);

if(clear_cycle==1 && flag==0)

printf(”THE CYCLE IS COMPLETE!!\n”);
}

}
int main()

{

pthread_t branch1,branch2,branch3,headquarters;

sem_init(&ready,0,1);

pthread_create(&headquarters,NULL,gen_branch,&place_id[0]);

pthread_   create(&branch1,NULL,assembly,&place   id[1]);

pthread_   create(&branch2,NULL,assembly,&place   id[2]);

pthread_   create(&branch3,NULL,assembly,&place   id[3]);

while(1);

return 0;

}
```

8.RESULTS AND DISCUSSION


C:\Users\ARUN\Downloads\OSP.exe

```
Headquarters can take engine and chassis
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
Headquarters can take chassis and ss
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
Headquarters can take ss and engine
Parts are being moved.
Branch1 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
Headquarters can take engine and chassis
Parts are being moved.
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
Headquarters can take chassis and ss
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
Headquarters can take ss and engine
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
Headquarters can take engine and chassis
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
Headquarters can take chassis and ss
Parts are being moved.
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
Headquarters can take ss and engine
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
Headquarters can take engine and chassis
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
Headquarters can take chassis and ss
Parts are being moved.
Branch3 has completed the assembly
THE CYCLE IS COMPLETE!!
THE CYCLE IS COMPLETE!!
```

Figure 7.1:
The results generated for all cases in a loop represented above. Showcases how both threads come together .

Table 7.1:

Input			Output
Engine	**Chassis**	**Electrical Supply**	**Manufacturing**
1	1	0	1
1	0	0	0
0	1	1	1
1	0	1	1
0	1	0	0
0	0	1	0

The program initially runs a loop where through randomization we determine which two of the three parts needed to manufacture the car is sent to the headquarters and their place ids are stored into variables and made a note of which are then called back at the assembly function which takes up the place id of the branch or part leftover and transfers the other parts to that location and through verification are assemble all together to form the vehicle.

All the different cases, in this program 3 cases, are displayed in the output box give above.

9.CONCLUSION

We learnt a lot about threads through this task. We came across many problems which are solved by using threads, we shortlisted these problems to 5. These were the very basic problems which will also help us in out in the upcoming exams since we tried to stay close to the syllabus, since the whole point of our project was to learn how to apply threads. So we have given an example that we solved. And hence we have completed our task.

10.REFERENCES

- Martin Rinard, MIT CSAIL,Parallel Synchronizaton,Pg no 1 to 3.
- Andrew D Birrell,An introduction to programming with threads,Pg no2.
- Joshua A. Redstone, Susan J. Eggers and Henry M. Levy University of Washington, An Analysis of Operating System Behaviour on a Simultaneous Multithreaded Architecture.
- http://www.reliasoft.com/BlockSim/multithread.htm
- Paul Eggert, Operating Systems Principles
- http://www.nakov.com/inetjava/lectures/part-1-sockets/InetJava-1.3-Multithreading.html.
- Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch and Mark Weiser, Using Threads in Interactive Systems.
- https://no-shoveling.com/wp-content/uploads/2015/01/threads.pdf
- Comprehensive Approach to Learning Java-Kishori Sharan/Wikepedia
- https://users.cs.cf.ac.uk/Dave.Marshall/C/node32.html
- http://www.cs.umd.edu/~hollings/cs412/s96/synch/smokers.html

RESEARCH PAPERS REFERENCES

- Characteristics of multithreading models for high-performance IO driven network applications

(https://arxiv.org/ftp/arxiv/papers/0909/0909.4934.pdf)

- EFFECT OF THREAD LEVEL PARALLELISM ON THE PERFORMANCE OF OPTIMUM ARCHITECTURE FOR EMBEDDED APPLICATIONS

(https://arxiv.org/ftp/arxiv/papers/1204/1204.2772.pdf)

- Discovering Threads in Research Papers (https://pdfs.semanticscholar.org/e4d4/0cc49d57cedaeb7bc157ce00d55d76bb017a.pdf)

- Visualizing massively multithreaded applications (http://ccl.cse.nd.edu/research/papers/threadscope-ccpe.pdf)

- Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors (https://ieeexplore.ieee.org/document/6338254/)

- Research of multi-thread applications for real-time control systems on humanoid robot embedded platforms (https://ieeexplore.ieee.org/document/5602743/)

- Analysis of Multithreaded Programs (https://people.csail.mit.edu/rinard/paper/sas01.pdf)

- The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications (https://www.nas.nasa.gov/assets/pdf/papers/saini_s_impact_hyper_threading_2011.pdf)

- SIMULTANEOUS MULTITHREADING: A Platform for Next-Generation Processors (https://dada.cs.washington.edu/smt/papers/ieee_micro.pdf)