Load Balancing by Process Migration in Distributed Operating System

Vatsal Shah, Viral Kapadia

***Abstract* -- Distributed operating system is nothing but the more than one cpu are connected with each other, but user can feel it as virtual uniprocessor. Now as more than one cpu are connected with each other its obvious that load will be increase. To compete with this load it is necessary to balance it. So in this paper I have focus on process migration technique for load balancing. For that I have describe two algorithms. 1) sender-initiated algorithm. 2) receiver-initiated algorithm.**

I. INTRODUCTION

A. Load Balancing

To understand Load balancing, it is necessary to understand load. Load may be define as number of tasks are running in queue, CPU utilization, load average, I/O utilization, amount of free CPU time/memory, etc., or any combination of the above indicators. Load balancing can be done among interconnected workstations in a network or among individual processors in a parallel machine. Load balancing is nothing but the allocation of tasks or jobs to processors to increase overall processor utilization and throughput.

Actually load balancing is done by process migration. But to balance the load it is necessary to measure the load of individual node in network or in a distributed environment. For calculating node above mentioned factor in a definition of load are calculated. After calculating the node of individual, mark the underloaded/free and overloaded/busy node.

Now to balancing load transfer the process from overloaded node to underloaded node. In this way load can be balance in network of work station or in a distributed environment.

A. Process Migration

Process migration is the transfer of process from one node to another node in network of workstations or nodes. It is very useful mechanism for balancing the load on distributed system. Load balancing in a distributed system can be done through transferring a process form heavily loaded node to lightly loaded node.

There are two types of process migration. (1) Preemptive Process Migration (2) Non-preemptive Process migration. Preemptive process transfers [3] involve the transfer of a process that is partially executed. This transfer is an expensive operation as the collection of a process's state (which can be quite large and complex) can be difficult. Typically, a process state consists of a virtual memory image, a process control block, unread I/O buffers and messages, file pointers, timers that have been set, etc. Non-preemptive process transfers [3], on the other hand, involve the transfer of processes that have not begun execution and hence do not require the transfer of the process's state. In both types of transfers, information about the environment in which the process will execute must be transferred to the receiving node.

II. MECHANISM

Process migration mechanism consists following four steps [3]:

1) Freezing the process on its source node and restarting it on its destination node.
2) Transferring the process's address space from its source node to its destination node.
3) Forwarding messages meant for the migrant process.
4) Handing communication between cooperating processes that have been separated (placed on different nodes) as a result of process migration.

Process migration can be done in homogeneous as well as heterogeneous environment. First of all we will focus on homogeneous environment because it is efficient.

For implementing preemptive as well as non-preemptive process migration there are two type of algorithm [1]:

1) Sender-initiated algorithm
2) Receiver-initiated algorithm

In Sender-initiated algorithm, I have implemented non-preemptive migration because sender means overloaded node wants to send(transfer) its process to another node. So it will directly transfer the newly arrived process rather than executing process.

While in receiver-initiated algorithm underloaded node wants to receive a process from another node which is overloaded. So whenever its requesting to node for process then overloaded directly transfer its executing process. Thus I have implemented non-preemptive migration for sender-initiated algorithm while preemptive migration for receiver-initiated algorithm. In the following section I have describe

Vatsal Shah, Post Graduate Student of Computer Engineering, Birla Vishwakarma Mahavidyalaya, V.V.Nagar, Anand, INDIA, Mobile No.: +91 94291 59259, vatsalshah95@yahoo.com .

Mr. Viral Kapadia, Asst. Prof at Computer Engineering department, Birla Vishwakarma Mahavidyalaya, V.V.Nagar, Anand, INDIA, Mobile No.: +91 94281 65939, vvkapadia@bvmengineering.ac.in .

my implementation details for load balancing by process migration.

III. IMPLEMENTATION

A. *Calculating CPU Load*

I have measured the load at user level. At user level we can measure system load by reading the information residing in the /proc file system or using sys-info system call (/usr/include/sysinfo.h) [1]. I have used both approaches for measuring the load.
We can use sysinfo system call (include/sysinfo.h) to get the current load of the CPU.

int sysinfo(struct sysinfo *s).
sysinfo system call gathers the current CPU statistics in sysinfo structure.

```
struct sysinfo
{
unsigned long uptime; /*time in jiffies since boot*/
unsigned long totalram; /*total memory*/
unsigned long freeram; /*free memory*/
unsigned long totalswap; /*size of swap area*/
unsigned long freeswap; /*size of free swap area*/
unsigned short procs; /*no of running process*/
unsigned long buffers; /*buffer memory*/
unsigned long cached; /*cached memory*/
………….
…………
 };
```

Another way to gather the CPU statistics is to read the /proc/stat file and measure the following factors.

CPU – Total jiffies (1/100ths of a second) that the processor spent on user, nice, and system processes. Obtain a percentage of total I/O – Blocks read/written to disk per second.

Memory – Page operations per second. Example: a virtual memory page fault requiring a page to be loaded into memory from disk.

Interrupts – System interrupts per second. Example: incoming Ethernet packet.

Context Switches – How many times the processor switched between processes per second.

Determine thresholds for each of these parameter using Micro Benchmark programs. The function GetCurrentCPULoad () determines current CPU load.
Determine whether node is busy/free
The function GetCurrentCPULoad () determines current CPU load.
For each load factor such as number of processes (CPU queue length), memory usage, swap area usage, etc., determine two thresholds: low watermark and high watermark. Mark the node free if most of the factors are below low water marks. Mark the node busy if most of the factors are above the high watermarks. The function IsCPUBusy () marks the computer as busy/free.

B. *Process Migration Policy*

Process migration incurs considerable amount of performance overhead and network traffic. Moreover, there exist possibilities that some of the message interactions with migrating process initiated by a migration underway application may experience a timeout and thus pose on adverse effect on the semantic transparency of such an application. Therefore, we need some policies to guide and justify process migration decision.

1. When to migrate a process? [2]
We could base this decision on the processor load of the host machine, and the presence of idle processors in the distributed system.

2. Which process to select for migration? [2]
We could decide this by estimating the overhead involved in migrating the process, like the size of the address space to migrate, the number of connections to break and setup, the dependence of the process on the host machine for services or IPC with other host machine processes, etc. Preemptive migration involves checkpoint/restart of process image.

3. When to allow remote processes to execute on a system? [2]
When should a processor send back the remote processes if the load on that machine increases? Should it just be based on a threshold or should we migrate back to the host machine once the remote processor becomes non-idle due to its own processes.

4. How to migrate a process? [2]
Should we do a total-copy (transfer the whole address space), or should we do demand fetching of pages in the address space.

C. Migration Algorithm for Sender-Initiated Algorithm

To achieve this I have created my own shell because we are not authorized to make changes in in-built shells. I have assumed homogeneous environment in the distributed system. For the node communication in the distributed system, I have used socket programming.
The modified shell is working as follow:

- Shell parses command line argument: To execute any executable file, when we enter the file name at the shell prompt, the shell parses the command line argument.
- Shell calls IsCPUBusy (): After parsing the command line argument, the shell will call IsCPUBusy () to check whether the host node is underloaded/free or overloaded/busy.
- Host node is underloaded /free: In this case, the shell will fork the process and call exec () system call to execute the process locally.
- Host node is overloaded /busy: If the host node is busy, the shell will poll other nodes randomly in the distributed system to determine underloaded/free remote node.

- Free node is found: If any free node is found in the distributed system, the newly arrived process is migrated via socket on this free node. The remote free node will fork the process and execute the migrated process.
- Finally, this remote node will send the results back to the original node.

D. Migration Algorithm for Receiver-Initiated Algorithm

1. Checkpointing the process

After process has been selected for migration we need to save the process states. The process checkpoint is the process of saving process state to the file at arbitrary point of execution.

For process migration we need to save following states
Process's *task_struct*
Process's *mm_struct*
Process address space (stack, data, code, heap etc)
Open files (regular files, pipes, sockets etc.)
Signals
Current working directory and current root.

a) Process's *task_struct*

Each process in Linux has *task_struct* associated with it. The information in the task structure are user id, group id, process priority, process state, process id etc. Save all these information in checkpoint file.

b) Process's *mm_struct*

Each process has memory map structure known as *mm_struct* whose address is stored in process's *task_struct* structure. It contains pointer to the page directory, start and end address of virtual memory areas like heap, stack, data and code regions. It also contains address of argument vector and environment vector passed to process. Save all these information to checkpoint file to reconstruct the memory map on destination node.

c) Process address space (Process's virtual memory areas)

On Linux, each process runs in its own address space which ranges from 0 to 4GB, but only small part of it is actually used. Each process has an associated memory map structure (*mm_struct)* which contains pointer to the first virtual memory area (*vm_area_struct*) in the list of *vm_area_structs*. We need to traverse this list to save process address space.

During the checkpoint we need to access the target process' address space from the kernel. Common C functions like strcpy () wouldn't work because pointers in kernel and user space have completely different meanings. Instead, the Linux provides a set of helper functions (copy_from_user (), copy_to_user (), etc) that allow data transfer between kernel and the current process. But now we want to access a process's address space that is not the current one, so we can't use these functions either.

Each process has its own page directory and page tables that are initialized in a fork and switched to in a context switch. All the processes have an address space from 0 to 4GB, but mapped to different physical memory regions due to their different page structures.

The kernel also has its own page directory and page tables, but it is special because it just maps the physical address to itself. For instance, a pointer of 0x00004000 in a process may actually points to a physical address of 0x01234000, but in the kernel a virtual address is exactly its physical address. Save the list of *vm_area_struct* page by page so that it is easy to recover on destination.

d) Open files

Linux supports several types of files like regular files, directories, symbolic links, pipes, sockets, block special files, character special files etc. But I have considered only regular files. Process refers files by file descriptors. Kernel refers each file by struct *file*. I have assumed NFS file system for my implementation. In NFS, each machine in the cluster mounts its file system on the NFS server. So we do not have to explicitly transfer the file contents along with the checkpoint file. This will reduce the network overhead.

2. Checkpointing Regular File

Since we are using NFS file system. We do not have to save the contents of the file but we only have to save the information in the struct *files_struct* and struct *file*. The *files_struct* contains information like number of open files, bit map for the open file, files to be closed on exec, array of pointers to the file objects. For every with an entry in the fd array, the array index is the file descriptor. A process can not use more than NR_OPEN file descriptors. The file object contains information like file object usage counter, process access mode, current file offset etc. so for each open file we need to save these information. The function DumpRegularFile () saves the information like file descriptor, file open mode, file offset, flags etc. to the checkpoint file.

3. Restarting the process

Process is reconstructed from the checkpoint file. Ideally, restarting should be invoked as soon as checkpoint file reaches to the receiver node. Restart module does the following things.
Create a new empty process.
Recover *task_struct* and *mm_struct*
Recover register set
Recover address space
Recover open files (regular files, pipes and sockets).

a) Recover *task_struct* and *mm_struct*

After new process has been forked on the receiver node, the restart module reads the task structure information like uid, gid, euid, egid, state etc. to the fields of *task_struct* of newly created process. It also reads the information about memory map from the checkpoint file. RestoreTaskStructure () function will restore the information about *task_struct* from the checkpoint file. Same way RestoreMMStructure () function will restore the information about memory map (*mm_struct*).

b) Recover register set

The register set in the checkpoint file is copied in to the register set of the newly created process. RestoreRegisters () function will restore the register set from the checkpoint file.

c) Recover process address space

In checkpoint file, we have saved the process's virtual memory areas. So restart module will call mmap () function to map the contents of virtual memory areas in checkpoint file to the process address space. We will recover all the regions of process like code, data, heap, stack etc. RestoreVMAreas () function will restore the virtual memory areas of process from the checkpoint file.

d) Recover open(Regular) files

Since we are using NFS file system, we do not save the contents of the file at the time of migration. The restart module reopens the file with same file descriptors. For each open file, it will restore the file permission mode, flags, file offset etc. So the process can continue with file operations after restart on new node. RestoreOpenFiles () function will restore the information about open files.

E. Algorithm

- Shell parses command line argument: To execute any executable file, when we enter the file name at the shell prompt, the shell parses the command line argument.
- Shell calls IsCPUBusy (): After parsing the command line argument, the shell will call IsCPUBusy () to check whether the host node is underloaded/free or overloaded/busy.
- Host node is overloaded /busy: In this case, the shell will fork the process and call exec () system call to execute the process locally.
- Host node is underloaded /free: If the host node is free, the shell will poll other nodes randomly in the distributed system to determine overloaded/busy remote node.
- Busy node is found: If any busy node is found in the distributed system, a process is selected on that node for the migration. Then checkpoint process is used to save process state to the file at arbitrary point of execution. The file is migrated via socket on the receiver node. Process is reconstructed from the checkpoint file and executed on the receiver node.
- Finally, the receiver node will send the results back to the original node.

IV. CONCLUSION

So the final conclusion is that using process migration one can easily balance the load of distributed operating system efficiently and quickly.

REFERENCES

[1] M Beck, H Bohme, M Dziadzka, U Kunitz, R Magnus, and D Verworner, "Linux Lernel Internals." Second Edition, Pearson Education Asia, Addison-Wesley.

[2] Partha Dasgupta and Ravikanth Nasika. "Transparent migration of distributed communicating processes", Arizona state university.

[3] Pradeep K. Sinha, "Distributed Operating Systems: Concepts and Design", 1997 by IEEE, Prentce-Hall of India.

[4] Narayan Joshi, Dr. D. B. Choksi, "Checkpointing Process Virtual Memory Area for Process Migration"; International journal of Emerging Technologies and Applications in Engineering Technology and Sciences; June-2010; pp-42-44