Results and Concluding Remarks

7.1 Introduction
7.2 Summary of the Proposed Research Contributions
7.3 The Output
7.4 Results of the Checkpointing the Virtual Memory
7.5 Further Scope of Research
CHAPTER – 7

Results and Concluding Remarks

7.1 Introduction

This chapter includes the accounting of this thesis. The proposed work in the earlier chapters of this thesis is summarized in section 7.2. The section 7.3 shows and discusses some of the output generated by the proposed implementation mechanisms of the suggested algorithms for process migration and load balancing in this thesis. During the timespan of implementation of the proposed algorithms, we had experimented our proposed implementation work; we had collected certain observations for our work and an existing work; the results and analysis is described in section 7.4. Furthermore, the chapter describes further scope of research in the area of process migration and load balancing in the section 7.5.

7.2 Summary of the Proposed Research Contributions

As illustrated in the previous chapters, we have modeled mechanisms to carry out reliable and optimized process migration and load balancing. The process migration technique is used to achieve efficient resource-usage, load-balancing and fault-tolerance in the network of workstations.

In this thesis, we have proposed algorithms and their implementations to carry out optimized process migration to handle the problems such as - checkpointing the memory regions occupied by the process, resuming the migrated process with certain process-identifier value, resuming the migrated process in certain system call which the process was executing earlier before its migration took place on the source workstation.
Furthermore, in this thesis, we have presented algorithms and their implementations to synthesize an efficient load balancing technique which can operate under two different strategies: the server-managed load balancing and the client-initiated load balancing. The work is aimed at providing optimized process migration and efficient load balancing functionalities.

7.2.1 Checkpointing and resuming system calls

At the time of process migration, chances are there that the process could be executing some system call. In such a situation, the migrated process must resume the same system call execution on the destination workstation as well.

The solution suggested in the third chapter of this thesis helps the process migration mechanism to determine whether the process is running in some system call at the time of process migration; and if so, then the proposed solution makes suitable arrangements such that after migrating to the destination workstation, the process resumes the same system call (which it was running before migration took place on the source workstation).

7.2.2 Checkpointing the Memory Regions Occupied by the Process

In order to carry out a valid migration of a partially executed process, the process migration mechanism must perform checkpointing of the memory regions which are occupied by the process on the source workstation. However, it should function efficiently to make the overall migration optimized in the terms of speed.

However, the task of checkpointing the memory regions which are occupied by the process in the optimized way is not possible to be
carried out without difficulties. There may be variations in the type of the virtual memory areas which are occupied by the process. In order to achieve optimization in terms of speed, all virtual memory areas of a process should not be checkpointed through some common technique. Therefore, the process migration mechanism which is striving for its optimization, should deal differently with different types of memory areas; e.g. a memory region which is associated with the data segment should be treated differently from the memory area which is associated with the stack segment. In this thesis, we have incorporated this idea in chapter 4 and we have proposed algorithms and their implementations that help the process migration mechanism to efficiently checkpoint different types of memory regions that are occupied by the process to be migrated. Results (section 7.4) indicate that the proposed in chapter 4 operates efficiently to accomplish our aim of optimized process migration.

7.2.3 The Process-identifier

The process-identifier i.e. pid is an important credential for any process in the system, which is uniquely assigned by the system to all processes. Care should be taken such that after migrating to the destination workstation, the migrated process resumes with the same pid which it initially possessed before migration took place on the source workstation.

However, the task of resuming of the migrated process with a certain pid value on the destination workstation is not without difficulties. The pid-value and PCB of a process is not an ordinary data structure and cannot be modified as and when thought about to do so; and still if it does like this, some side-effects could arise such as either the process may crash or the process may disappear from the list of processes or it could lead the system towards failure. On the contrary, we want it to be safely carried out. We aim it to be not only safe but
also it should happen in an optimized way. Furthermore, this operation should be transparent to the system i.e. the kernel.

This problem can be solved by injecting certain pid value into kernel for our migrated process before its resumption on the destination workstation. In this dissertation, we have proposed a novel solution in the form of a new system call setforkpid() which is implemented by us. The approach is described in the fifth chapter.

### 7.2.4 Load Balancing

Load balancing can be of two types: server-managed load balancing and client-initiated load balancing. In our proposed load balancing algorithms, we extend the basic-idea of load transfer on the basis of the round-robin scheduling algorithm. The proposed algorithms in this thesis provide efficient load balancing by protecting the lightly loaded workstations from instantaneous overburdening.

### 7.3 The Output

In this section we present the output generated by our implementations of the proposed algorithms in the prior chapters. The section 7.3.1 describes working of the process migration mechanism in form of images. Furthermore, the section 7.3.2 describes working of the load balancing mechanism in form of images.

#### 7.3.1 Process migration

This section describes the output of the process migration work. The following sub-section 7.3.1.1 describes the span of process execution which occurs on the source workstation before the process migration takes place. After, partial execution on the source workstation, the process is migrated to the destination workstation. The section 7.3.1.2
describes the span of process resumption which happens after the migration of the process on the destination workstation.

### 7.3.1.1 Before Process Migration

In the figures 7.1 and 7.2, execution of a matrix multiplication program is shown. The program performs multiplication of two dynamic matrices. It first reads the dimensions (i.e. number of rows and columns) of two matrices M1 and M2 from user, and creates appropriate matrices; then it reads both the matrices’ elements’ values from user. The program calculates and prints the product of these two matrices as an output.

As shown in the figure-7.1, the program is running with the process-id 3308 on the source workstation [IP: 169.254.254.52]. The figure shows that the program first reads the dimensions for the matrix M1 from keyboard, creates the matrix M1 with appropriate number of rows and columns and then it reads M1’s elements from the keyboard. The same steps for the matrix M2 are shown in the figures 7.1 and 7.2.

Now, to show migration of the matrix-multiplication process in figure, we perform its migration to the destination workstation [IP: 169.254.254.51] which is shown in figure-7.3. The figure shows checkpointing of the memory regions which are occupied by the matrix multiplication process 3308. The figure also shows that at the time of checkpointing, the process was running in the system call number 162 (i.e. `sleep()`). Later, the checkpointed process image is migrated to the destination workstation for resumption.
Figure 7.1 Process execution before migration takes place – I
Figure 7.2 Process execution before migration takes place – II
Chapter 7: Results and Concluding Remarks

Optimized Process Migration for Load Balancing in Distributed Systems

Figure 7.3 Process Migration from the source workstation
### 7.3.1.2 Post-migration Resumption of a Process

The post migration resumption of process on the destination workstation [IP: 169.254.254.51] is described in the figure-7.4. Here, on the destination workstation, using our `setforkpid()` system call (chapter 5), the migrated process resumes with the same pid 3308, which it had on the source workstation before migration took place. The resumed process computes product of two dynamic matrices and shows output.
Figure 7.4 Resumption of the migrated process on destination system
7.3.2 Load Balancing

This section describes the output of the load balancing work proposed by us. The following sub-section 7.3.2.1 describes the behavior of the server-managed load balancing strategy. The section 7.3.2.2 describes the behavior of the client-initiated load balancing strategy.

7.3.2.1 Server-managed Load Balancing

1. Before load balancing

   In the top part of the figure-7.5, it is seen that initially the client_51 [IP: 169.254.254.51] is ACTIVE (i.e. 1) with the workload of 1.910, and the client_49 [IP: 169.254.254.49] is ACTIVE (i.e. 1) with the workload of 0.160.

   Now, the LoadBalancer module (i.e. the proposed function load_balancer_thread_fn()) determines the client_51 as highly loaded and the client_49 as lightly loaded clients. Therefore, it instructs the highly loaded client_51 to share its workload with the lightly loaded client_49 i.e. [IP:169.254.254.51] -> [IP:169.254.254.49].

   After the sending the migration instruction to the highly loaded client, the status of client_49 i.e. [IP: 169.254.254.49] is immediately toggled into PASSIVE (i.e. 2) in the load table (see bottom part of figure 7.5). This indicates that the LoadBalancer has chosen client_49 for accepting the shared workload, so the same client_49 should not be chosen for sharing of more workload till client_49’s status remains PASSIVE.

2. After load balancing

   In the figure-7.6, it is seen that after migrating certain processes from the client_51 to the client_49, workload of the
client_51 reduces to 0.370 and workload of the client_49 increases to 0.770. Once again, the status of client_49 is toggled back to ACTIVE (i.e. 1) status; which was earlier turned off into PASSVIE (i.e. 2) status.

Thus, some of the output of the server-managed load balancing work is shown further.
Figure 7.5 The lightly-loaded client [IP: 169.254.254.49] is now PASSIVE (see bottom of the shell)
Figure 7.6 The lightly-loaded client [IP:169.254.254.49] is now ACTIVE
7.3.2.2 Client-initiated Load Balancing

As described in the chapter 6, the client-initiated load balancing mechanism has been implemented in two modules: the request listener and the request sender. The workstation wishing to transfer its workload sends a request through its request sender module to the request listener module which is running on peer workstation. The request is sent on the round robin basis to the peer workstations. Upon receiving the request, the peer listener workstation decides whether to grant (i.e. YES) or deny (i.e. NO) the request. And the request listener replies the decision back to the requester workstation. In case of denial of request (i.e. NO), the requester sends request to the next workstation whose turn comes according to the round robin policy. Otherwise, if the listener grants (i.e. YES) the request for load transfer, it pauses listening more requests for some time interval.

1. **Listener is highly loaded, denies the request for load transfer**

   In the figure 7.7, the request listener for the workstation_50 [IP: 169.254.254.50] is shown; it is listening for the load transfer requests coming from the peer workstations. At the time of request arrival, the listener workstation itself is highly loaded (with current load 1.440 which is far more than the lower load threshold value of 0.500), and so the listener decides to not grant the request and replies back ‘NO’ to the requester.

2. **Listener is lightly loaded, grants the request for load transfer**

   In the figure 7.8, the request listener for the workstation_51 (IP: 169.254.254.51) is shown; it is listening for the load transfer requests coming from the peer workstations. At the time of request arrival, the listener workstation is not highly loaded (with current load 0.330 which is less than the lower load
threshold value of 0.500), and so the listener decides to grant the request and replies back ‘YES’ to the requester.

3. **Sender sends request to listeners**

In the figure 7.9, the request sender for the workstation_49 [IP: 169.254.254.49] is shown. The workstation_49 wants to share its workload with some peer workstation in network. So, in order to send request, as per the round robin policy, it sends request to the workstation_50 which is already highly overloaded as shown in the above case 1). It gets back the ‘NO’ reply. Therefore, it sends request to the next in turn workstation_51 (which is free as shown in the above case 2). It gets back the ‘YES’ reply. And it further proceeds for the workload transfer process migration to the workstation_51.

Thus, some of the output of the client-initiated load balancing work is shown next.
Figure: 7.7 Highly loaded workstation_50 says ‘NO’ to load sharing request
Chapter 7: Results and Concluding Remarks

Optimized Process Migration for Load Balancing in Distributed Systems

Figure 7.8 Lightly loaded workstation_51 says ‘YES’ to load sharing request
Figure 7.9 The workstation_49 requests for load sharing to the peer workstations
7.4 Results of the Checkpointing of Virtual Memory Areas

In the chapter 4, we described two implementation mechanisms for checkpointing the contents of memory regions of a process: the 4.7.4.1-ckpt_heap_stack_anonymous_segments and the 4.7.4.2-ckpt_data_segments. The following table 7.1 shows time taken for checkpointing sample sizes of virtual memory areas by our implementations and the existing technique [H. Zhong, 01] experimented by us. The chart 7.1 shows comparison of these performance results [N.A. Joshi, 11].

<table>
<thead>
<tr>
<th>Sample VMA Size (in Bytes)</th>
<th>Time Taken (in Jiffies)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Implementation 4.7.4.1</td>
</tr>
<tr>
<td>4096</td>
<td>14064</td>
</tr>
<tr>
<td>8192</td>
<td>29025</td>
</tr>
<tr>
<td>12288</td>
<td>47510</td>
</tr>
<tr>
<td>86016</td>
<td>679440</td>
</tr>
<tr>
<td>135168</td>
<td>1005880</td>
</tr>
</tbody>
</table>

Table 7.1 Results of the checkpointing the Virtual Memory Areas

Figure 7.10 Comparison of the implementations for checkpointing VMAs
The chart-7.1 shows better performance for our implementation 4.2.1 as the technique consumes least time to checkpoint the memory regions.

7.5 FURTHER SCOPE OF RESEARCH

7.5.1 Threads
The process can start multiple threads of execution in addition to the main thread of execution. In such a case, the process migration mechanism can consider the migration and resumption of these additional threads on the destination workstation.

As far as a thread is concerned, it uses the same memory regions which are occupied by the main process. However, a separate data-structure is maintained in kernel to preserve the per-thread specific CPU-register set. Moreover, the threads also do have unique id which differs from the main process id.

Models and their implementations can be developed for migration of a thread to the destination workstation and resumption of the thread within the associated migrated process.

7.5.2 Pipes
The pipe is a facility provided by operating system to redirect output of one process as an input to some other process. A set of processes may communicate with each other by means of pipes. So, further expansion to the process migration can be made in order to incorporate migration of the pipes across the workstations.

7.5.3 Sockets
Further extension to the process migration technique can be made in migration of sockets. A process may open socket connection with some remote process to communicate with it. Therefore, as an
extension to process migration of such a process, migration of the sockets and ports to the destination workstation can be experimented. Moreover, the peer workstation i.e. the other-end of the socket should be informed about migration of the socket to the destination address. Experiments can be carried out to migrate and restore the socket connections. Examination of this problem may provide a valuable insight into the migration of the communicating processes.

**7.5.4 Server-managed load balancing**

More efficient algorithms for the server-managed load balancing can be modeled for the purpose of enhancement of the scalability of the central server so that it may support large base of workstations for load balancing.
Figure A: Structure of the OptiMigrator (OM)

- LS - LoadServer Module
- LB - LoadBalancer Module
- LP - LoadPortal Module
- RS - RequestSender Module
- RL - RequestListener Module
- om - ProcessMigrator Module
- om_d - ProcessMigrator Module
- CSM - ClientStatusMaintainer Module