An Efficient Live Process Migration Approach for High Performance Cluster Computing Systems

Ehsan Mousavi Khaneghah, Najmeh Osouli Nezhad, Seyedeh Leili Mirtaheri,
Mohsen Sharifi, and Ashakan Shirpour
School of Computer Engineering
Iran University of Science and Technology
Tehran, Iran
{msharifi, mirtaheri, emousavi}@iust.ac.ir
{n_osuli, a_shirpour}@comp.iust.ac.ir

Abstract. High performance cluster computing systems have used process migration to balance the workload on their constituent computers and thus improve their overall throughput and performance. They however fail to migrate processes lively in the sense that moving processes are blocked (frozen) and are non-responsive to any requests sent to them while they are moving to their new destinations and have not reached and resumed their work on their new destinations. Previous efforts to prevent losing requests during process migration have been inefficient. We present a more efficient approach that keeps migrating processes live and responsive to requests during their journey to their new destinations. To achieve this, we have added a new state called the exile state to the traditional state model of processes in operating systems. A migratory process changes its status to the exile state before starting to migrate. All requests to the migratory process are executed locally on the old location of the process until the process reaches its destination computer and resumes its work anew. We show that our approach improves the performance of clusters supporting process migration by decreasing freeze time.

Keywords: High Performance Computing Clusters, Process Migration, Live Migration, Process Status, Distributed Systems,

1 Background

Process migration is the act of moving processes from one machine to other machines in distributed systems for load balancing, locality of resource usage, access to more processing power, and fault resilience. The context of a process (process state) to be migrated includes heap data, stack content, processor registers, address space, process running state, and process communication state (like open files or message channels) that all of them are essential for a process to continue its execution elsewhere [1],[2],[3]. The process running state is a part of process state that represents the running state of a process (Section 2).

Part of process state that has the most overhead in process migration mechanisms is the process address space that might have hundreds of megabyte of data [2],[3], so the transmission of address space in recent implementations has been mentioned more than other process state parts.

In the following subsections, we will review process migration algorithms demonstrating how to transfer the process address space and how important are challenges of these algorithms.
In Section 2, we review the evolution of process models and explain the features of each state. In Section 3, we present our new efficient live process migration approach for high performance cluster computing systems. Section 4 reports an evaluation of the approach and Section 5 concludes the paper.

1.1 Total-Copy Algorithm

Total-Copy is the first and the most popular process migration algorithm [2] that is sometimes called eager (all) too [1]. Demos/MP [4], Amoeba [5], and Charlotte [6] are examples that use some versions of this algorithm in user level or kernel level. In this algorithm (Fig. 1) when a process is stopped executing in a source node upon migration, all of its state is transferred to the destination node and then resumes in the destination [4],[5],[6]. A modified version of this algorithm is eager (dirty) that is used in Mosix. This algorithm can be used only if the system has remote paging facilities. In this algorithm, just the modified pages are transferred at the migration start and then other pages are transferred on demand. The first cost of eager (dirty) is lower than Total-Copy [1] ,[7] ,[8],[9].

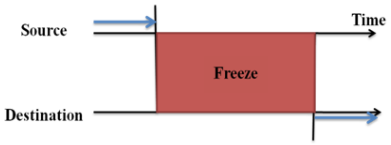

Fig. 1. Total-Copy algorithm

1.2 Pre-Copy Algorithm

The main goal of this algorithm when presented in System V [10] was to prevent failed communications in total copy algorithm [1]. In this algorithm, address space is transferred from source node to destination node while the process is executing on the source node (Fig. 2). When the number of modified migratory process's pages in source node becomes under some threshold, the process is frozen (blocked). Then all of the remaining process state is transferred and the process resumes in destination node [10]. The freeze time is lower than that in Total-Copy algorithm.

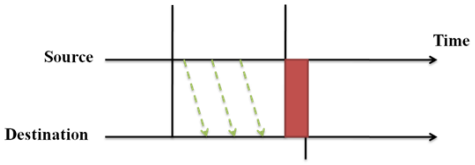

Fig. 2. Pre-Copy algorithm

1.3 Demand Paging Algorithm

This algorithm was first implemented in the Accent operating system by Zayas [2]. In this algorithm, the migratory process is frozen, the execution, the control data, and the address space metadata are transferred to destination node (Fig.3), but the address space remains in the source node. The destination node requests pages if it needs them. This algorithm is called Copy-On-Reference [11]. To improve this algorithm, some operating systems transfer one page of heap, stack, and code at migration time too [2].

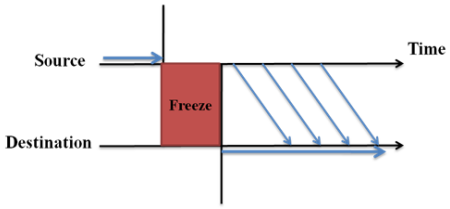

Fig. 3. Demand paging

1.4 File Server Algorithm

This algorithm was first developed by Douglis that uses an added machine as a file server in the Sprite operating system [12]. In the system the process is blocked, the modified pages and the modified file blocks are flushed to file server, and then the migratory process is resumed in the destination node and requests the pages from the file server if it needs them (Fig. 4). This algorithm is called Flushing [1].

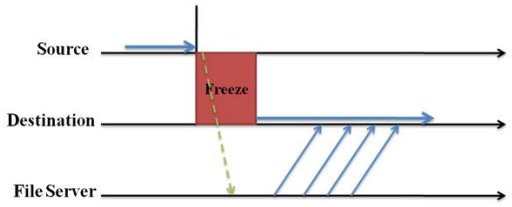

Fig. 4. File server

2 Evolution of Process State Models

In the early operating systems that were not multi-task, when a process execution was started, it used the processor until it was finished and so those systems did not need any process running state. In the recent operating systems that are multi-task and processes can execute interleaved, any process in its lifetime could have different running states (that are exclusive) [13]. The kernel should maintain the information of process running state in order to improve dispatcher's performance. For example, LINUX operating system holds this information in *state* field in *task_struct* structure [14]. In the following subsections, we review process running state evolution.

2.1 Two-State Model

In this model, that is the primary model in operating systems, a process is in the RUNNING state or NOT RUNNING state (Fig. 5). The disadvantage of this model is that processes can be in the NOT RUNNING queue for two reasons: have finished their processor quantum or waiting for I/O. The dispatcher should search the entire list of processes to select a suitable process and spend unnecessary time resulting in lower performance [13].

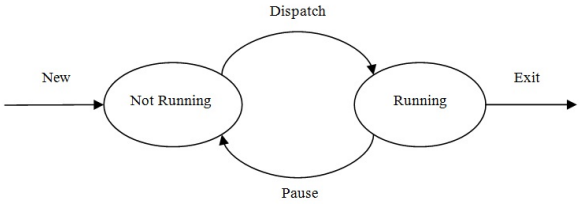

Fig. 5. Two-state transition diagram

2.2 Three-State Model

MINIX uses a three-state process model (Fig. 6) [15].

- RUNNING: In this state, the process is executing its instructions.
- READY: the only resource that the process does not have in this state is the processor. The state of a process changes from RUNNING to READY when it finishes its processor quantum, or from BLOCKED to READY when its desired request for I/O has been satisfied.
- BLOCKED: When a process executes a system call for I/O and the I/O is not available in memory, the process is removed from the dispatcher queue, its register values are saved in the process table, and it is blocked until its I/O is brought to memory. A blocked process cannot continue executing even if the processor is idle.

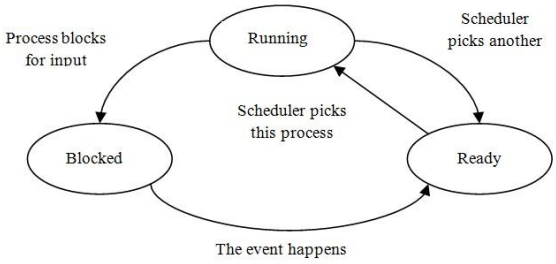

Fig. 6. Three-state transition diagram

2.3 Five-State Model

Another process state model contains five states (Fig. 7) [17]:

- **NEW:** This state denotes the case where a process is created in response to *fork()* or *exec()* system calls. Process structures and process identifier are allocated to the process, but the process is still not loaded into the memory possibly because of a restriction on the number of processes in the memory.
- **RUNNING:** This state is the same as in the three-state model.
- **READY:** In this state, the only resource that a process does not have is the processor. While the system is ready to execute a new process, one process can move from the NEW state to this state. On the other hand, a process can move from the RUNNING state to this state because of finishing the processor quantum or process pre-emption (because the event that a process with higher priority in BLOCKED state was waiting for has occurred).
- **WAITING:** The process is waiting for completion of I/O or the occurrence of a special event.
- **TERMINATED (exit):** In this state, the memory allocated to a process is released. In this state, the operating system releases the process. Only the accounting programs calculate process usages for billing purposes. A process may move to this state from the READY state because of termination of its parent process or from the BLOCK state because a parent process has killed his children.

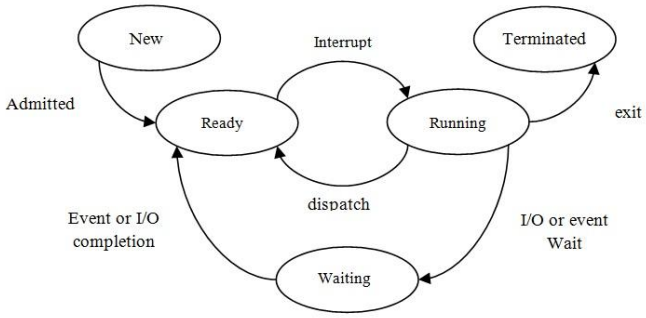

Fig.7. Five-state transition diagram

2.3 Seven-State Model

Because the processor's speed is more than I/O speed, all processes in the memory may be in the WAITING state. In order to prevent busy waiting the processor time, a new state named SUSPEND has been introduced (Fig. 8) [17].

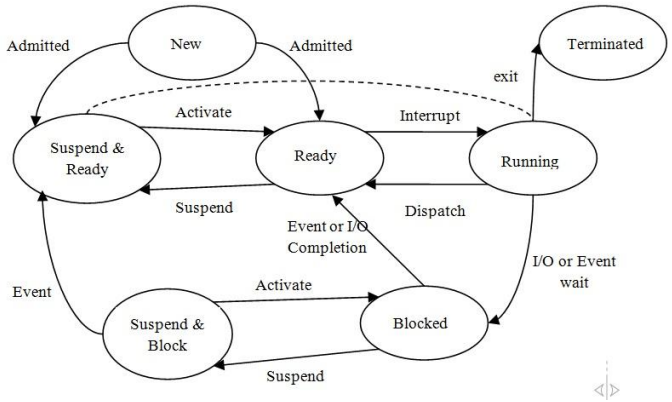

Fig.8. Seven-state transition diagram

The NEW, TERMINATED, READY, RUNNING, and BLOCKED states are exactly similar to their counterparts in the five-state model. The states SUSPENDED & READY and SUSPENDED & BLOCKED are different. If all the processes in the main memory are blocked and the processor is idle, some blocked processes can be moved to secondary storage and theie state changed to BLOCK&SUSPEND. If the event that the process in the secondary storage was waiting for occurs, the process changes its state to this state and remains in secondary storage. If the event that the process in the secondary storage was waiting for occurs, the process changes its state to the READY&SUSPEND state and remains in the secondary storage.

2.4 Nine-State Model

In UNIX operating system [16], [17], the process state diagram has nine states (Fig. 9). In this model, the RUNNING state is split into USER RUNNING and KERNEL RUNNING, and the TERMINATED state is called ZOMBIE. The PRE-EMPTED state queue is the same as the READY IN MEMORY; their only difference is how the process changes to this state. When the process is running in kernel mode and completes its execution in this mode, the kernel may decide to pre-empt the current process because of a ready process with higher priority. Therefore, the current process goes to this state. Notice that the process only could pre-empt when it is switching from kernel mode to user mode.

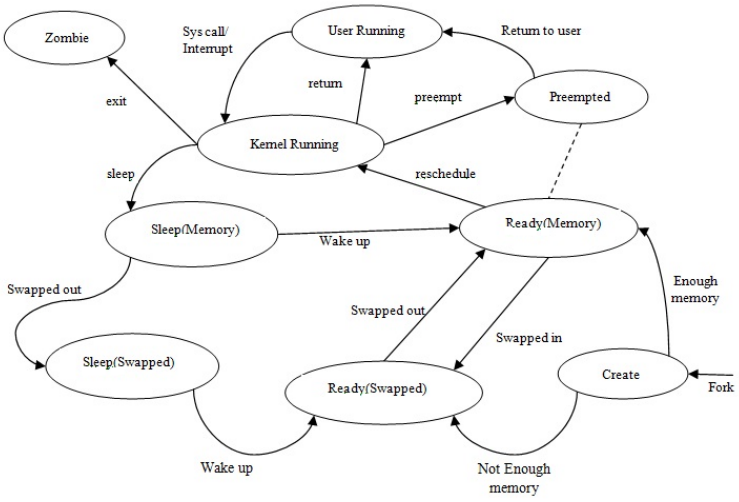

Fig.9. Nine-state transition diagram

3 Our Approach

As we stated in previous sections, in the current implementations of process migration, there is a freeze time for the migratory process wherein the process cannot respond to any requests. For example, in the Total-Copy algorithm, the speed of address space transfer is too low even for processes with small address space sometimes taking several minutes. In the Pre-Copy algorithm, although we can have the small freeze time, but this time is dependent on the modified pages in the last step and may be more than the Total-Copy algorithm. In the Demand Paging algorithm, the freeze time is too small compared to other algorithms but its main disadvantage is that the source node should keep the address space until the completion of the process execution. This data dependency decreases the fault resilient.

The freeze time has overhead and decreases the performance, but it is important for the migratory process requiring to communicate with other processes. If the freeze time is too long, the migratory process is assumed to have failed and communications are terminated. This is important especially for HPC clusters because the number of critical requests with low waiting time is more than other distributed systems.

We can consider the following steps in process migration:

1) **Negotiation**: After negotiation between the source and destination nodes, if the destination accepts the migration request, an instance of a process is created in the destination node.
2) **Primary migration**: Some parts - based on the algorithm – of the process state are transferred to the destination node and imported to the new instance.

3) **Freezing process**: The migratory process is blocked in the source node and communications are temporarily suspended. The migration is completed by transferring the remaining parts of the process.
4) **Some means of forwarding references**: For ease of communication after migration we have 3 solutions:
    - Keeping the address of the destination node on the source node (Sprite)
    - Searching the migrated process with Multicasting (System V)
    - Notifying the communicating processes (Charlotte)
5) **Resume**: The migratory process is resumed in the destination node.

One challenge in process migration algorithm is to decrease the freeze time in step 3. Therefore, the number of the failed communications will be lower than the current implementations. As we noted in the previous section, there is no state that a process can migrate and respond to its requests. Therefore, if we define a new process running state that all the machines in the cluster could see, and move the process to this state while migrating, and enable the process to respond to the requests while it resides in this state, then the availability of the process is improved.

The new state (Exile) has been added to the seven-state process model (Fig.10). We describe this model in the following paragraphs.

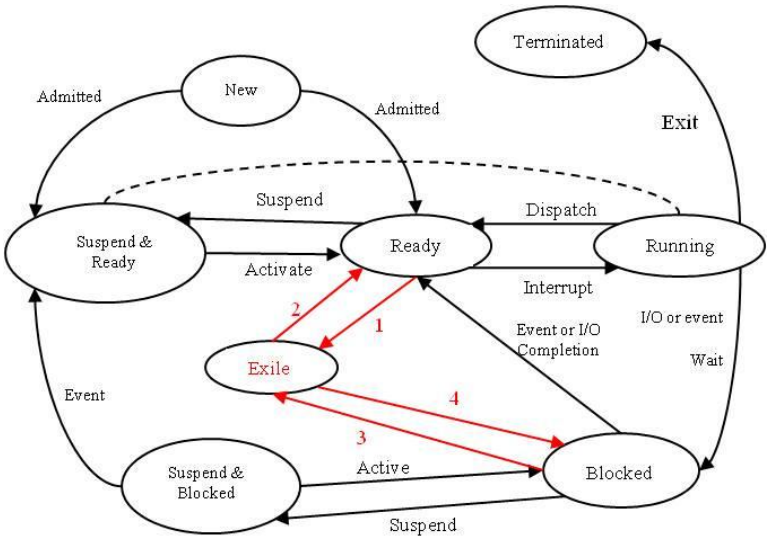

Fig. 10. Exile state in diagram.

If a process is in the NEW state and it is selected for migration, this migration means remote execution. In this paper, the migration does not mean remote execution. It means the process can migrate after starting its execution, but in the NEW state, only the kernel data structures have been allocated. Therefore, there is no transition between the NEW state and the Exile state.

If a process is in the RUNNING state and the load balancer selects it for migration, it moves the process to the READY state before migrating it.

If a suspended process (ready or blocked) is selected for migration, it is first moved to memory, its state is changed to READY or BLOCKED, and then to Exile state, and then it is migrated.

When the process is in READY queue and has been selected for migration, it will be better to change its state to Exile, because in the READY state there may be requests. On the other hand, after completing migration, the migratory process should move from Exile to READY state at destination node because the process state should be similar before and after migration.

In addition, if the process is waiting for an event in the BLOCKED state, it will be better to change its state to Exile because there may be signals from his parent and they should not be lost. Therefore, the new approach is like this:

1- **Negotiation**: After negotiation between the source and destination nodes, if the destination accepts it, an instance of a process will be created in the destination node.
2- **Primary migration**: Some parts - base on the algorithm – of the process state are transferred to the destination node and imported to the new instance.
3- **Go to Exile state**: The migratory process goes to this state and can responds to some requests that could not in BLOCKED state. The migration is completed by transferring the remaining parts of the process.
4- **Go to state that it was in it before migration.**
5- **Some means of forwarding references**: for easing communication after migration we have 3 solutions:
6- **Resume**: The migratory process is resumed in the destination node.

4 Evaluation

To evaluate our approach, we categorize the requests based on the "maximum time that the sender could wait for receiving respond" in three groups and carried out three experiments:

a) Critical processes with low waiting time. There are many such processes in HPC clusters.
b) Processes with medium waiting time.
c) Processes with high waiting time.

We have assumed the following waiting times in our experiments: a) 10 quantum, b) 30 quantum, and c) 50 quantum. In experiment 1 (Fig.11), we assumed the following numbers of processes a=20, b=30, c=40. That means 20 processes sent requests to a migratory process before the migratory process froze. Each one waited 10 quantum on average to receive response from the migratory process, 30 processes sent requests to the migratory process before the migratory process froze and each one waited 30 quantum on average to receive response, and 40 processes sent to the migratory process and each one waited 10 quantum on average to receive response. In

experiment 2 (Fig.12), the numbers were (a=40, b=20, c=30), and in experiment 3 (Fig.13) are (a=30, b=40,c=20).

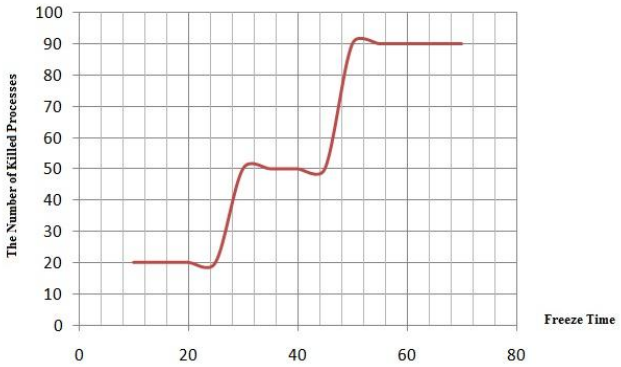

Fig.11. Minimum number of killed processes in experiment 1: a = 20, b=30, c=40

As Fig.11 shows, the minimum numbers of killed processes directly depend on the freeze time. When the freeze time decreases the number of killed processes decreases too. So in our approach because of the existence of a new state and because the migratory process can respond to requests in this state, the availability of the migratory process is increased. Fig.11 represents a distributed system with small number of critical requests.

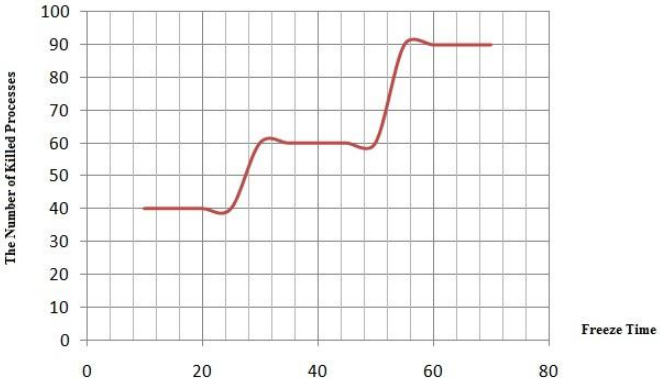

Fig.12 Minimum number of killed processes in experiment 2: a=40, b=20, c=30

In Fig.12, the number of processes in category (b) is more than in two other categories. This experiment is for HPC clusters, because in these systems the response time is very important in inter process communications. When the freeze time decreases, the migratory process can respond to more requests compared to current process migration mechanisms, and the performance is increased.

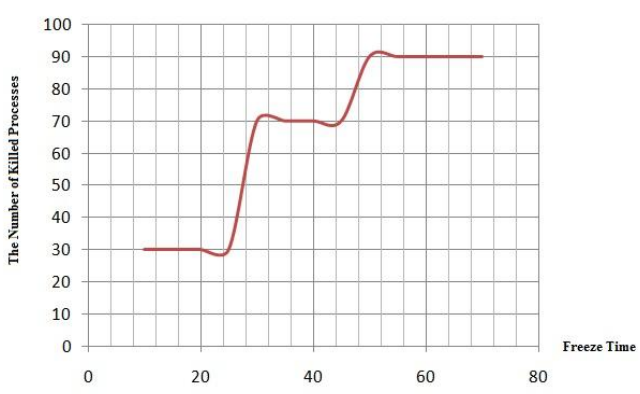

Fig.13. Minimum number of killed processes in experiment 3: a=30, b=40, c=20

As Fig. 13 shows, when the freeze time of the migratory process is decreased the number of killed processes is decreased too. Therefore, in HPC clusters we need a process migration approach with small freeze time to improve the performance.

5 Conclusion

One of the important challenges of processes in HPC clusters is the freeze time that the migratory process cannot respond to any requests and the sender processes may be killed because of not receiving any responses. In this paper, we propose to define a new state, named exile, over the cluster that the migratory processes stay in that while migrating. This state lets the process to be available and responsive to critical requests. Therefore, the communications does not break and the performance is increased. We evaluated this approach with the communications parameter. We considered three types of processes: critical processes, medium waiting time processes, and high waiting time processes. The results showed that the proposed approach improves the response time, and that it is especially essential for decreasing the killed rate of critical processes. One of the future works can be evaluating this approach with computing parameter. The relationship between the quanta that we used in our experiments with three categories of requests can be nearer to real ones in HPC clusters.

6 References

1. Milojicic, F., Douglis, F., Paindaveine, Y.,Wheeler, R., Zhou, S.: Process Migration. ACM Computing Surveys (CSUR). 32, 241--299 (2000).
2. Roush, E, T., Campbell, R, H.: Fast Dynamic Process Migration. In: Distributed Computing Systems, pp.637, (1996).

3. Ho, R, S,C., Wang, C., Lau, F, C.: Lightweight Process Migration and Memory Prefetching in openMosix. In: Parallel and Distributed Processing, pp.1—12, Miami (2008).
4. B. P. M. Michael L.Powel: Process Migration in Demos/MP. ACM SIGOPS Operating Systems Review. 17, 110--119 (1983).
5. Steketee, C., Socko, P.,†Kiepuszewski, B.: Experiences with the Implementation of a Process Migration Mechanism for Amoeba. In: Computer Science, pp.140—148, Melbourne (1996).
6. Artsy, Y., Finkel, R.: Designing a process migration facility:The Charlotte experience. In: IEEE Computer, pp.47-56, (1989).
7. Barak, A., La'adan, O., Shiloh, A.: Scalable Cluster Computing with MOSIX for LINUX. (1999).
8. O. L. Amnon Barak: The Mosix Multicomputer Operating System for High Performance Cluster Computing, Future Generation Computer Systems.13, 361—372 (1998).
9. Barak, A., Shiloh, A.: The MOSIX Management System for Linux Clusters, Multi-Clusters and Clouds. (2009).
10. Theimer, M., Lants, K., Cheriton, D.: Preemptable Remote Execution Facilities for the V System. In: Operating systems principles, pp.2-12, Washington (1985).
11. Paoli, D., Goscinski, A.: Copy on Reference Process Migration in Rhodos. In: Algorithms and Architectures for Parallel Processing, pp.100—107, (1997).
12. Douglis, F.: Transparent Process Migration: Design Alternatives and The Sprite Implementation. Software—Practice & Experience.21, 757—785 (1991).
13. Tanenbaum, A.: Modern Oprating Systems. United States of America(2009).
14. Kumar, A.: TASK_KILLABLE: New process state in Linux. IBM (2008).
15. Tanenbaum, A.: Operating Systems Design and Implementation. Prentice Hall(2006).
16. Bach, M, J.: The design of the UNIX operating system. Prentice Hall (1986).
17. Stallings, W.: Operating Systems: Internals and Design Principles. Prentice Hall(2007).