UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Excercise Sheet 6
Solution

Lecture Distributed Systems
Summer Term 2020

Exercise 1: Java Security Manager

a) The security manager allows a Java application to implement security policies. I.e., the application can determine whether certain (potentially "dangerous") operations should be performed or not. These include, for example, accessing the file system, communicating over the network, terminating the virtual machine, or creating windows. This is always useful if the application uses classes (i.e. foreign code) whose source may not be trustworthy. An example are Java applets whose code is loaded from an arbitrary web server and executed in the web browser. To prevent the applet code from causing any damage, applets always run under the control of a security manager.

The security manager protects the resources of the local computer from damage, which could be caused by the execution of foreign or unknown code.

Technically, the security manager is implemented by the class `SecurityManager` (or a subclass thereof). The methods of this class (based on a security policy, e.g. in the form of a policy file) perform a preliminary check as to whether a certain method/operation may be executed. If yes, they return, if no, they throw the exception `java.lang.SecurityException` (see Java documentation for `java.lang.SecurityManager`).

b) When a security manager is defined, all critical operations are prohibited unless explicitly allowed (e.g. by a policy file). If the client is started without specifying the policy file (property `java.security.policy`), file access to `/tmp/test` is not allowed.

You will find the necessary policy file in the archive `l06eFiles.zip`[1] on the lecture's web page.

c) You will find the solution to this task in the archive `l06eFiles.zip`[2] on the lecture's web page. The method `checkWrite()` of the security manager checks whether the file name ends with `.java` and throws a `SecurityException` in this case. Otherwise, the `checkWrite()` method of the `SecurityManager` superclass must be called to execute the standard access controls.

Exercise 2: Load balancing strategies

a) Static load distribution: a schedule is created before execution; disadvantage: all processes and their execution times must be known a priori, dynamics of processes and external load cannot be taken into account.

Dynamic load distribution: processes are (only) placed on a node when they are created and remain there; disadvantage: later change of the load situation (due to dynamics of the processes or external load) is not taken into account.

Preemptive dynamic load distribution: processes can switch nodes during their runtime (even several times); disadvantages: migration problems, thrashing possible; advantage: very adaptive, especially with external load.

b) Not all ready processes cause the same amount of load (e.g. processes that frequently wait for I/O). A better load measure could be the CPU load (i.e. how much % of the time the CPU is actually calculating), which however is not always easy to determine and can also fluctuate strongly. Memory requirements, page error rates, I/O rates, communication rates, etc. could also be included in load metrics. However, the (frequent) determination of these metrics means additional overhead, which is not always amortized by the "better" metric. Experiments have shown

[1] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l06eFiles.zip`
[2] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l06eFiles.zip`

that (due to this overhead) usually no better load balancing is achieved by such metrics (i.e. the runtime of a program system is not reduced).

Exercise 3: Graph partitioning and list scheduling

Load distribution by graph partitioning:
Given is a number of concurrent, communicating processes and the communication load between each two processes. This is shown as a graph. The nodes represent the processes and the edges show the communication load. Objective: to distribute the resulting load as evenly as possible in order to prevent overload on individual nodes. The graph is partitioned in such a way that the weight sum of the cut edges is minimal, i.e. the communication between nodes is minimized. All load balancing strategies base their decisions on some *load metric*.

List-Scheduling:
The program consists of several dependent tasks, which may work on output data of their predecessor tasks, but do not communicate *during* processing. This is represented as a DAG. The nodes contain tasks with execution times and the edges show the necessary communication along with the transmission time. Objective: To schedule the tasks so that the processing time (*makespan*) is minimized.

Similarities:
Graphs with tasks as nodes and communication as edges, edge weights, task structure is fixed, number of CPUs is given, schedule (assignment of processes to CPUs) is calculated before program start, goal is usually to minimize the runtime, ...

Differences:
Program structure (with graph partitioning: concurrent processes that communicate during execution; with list scheduling: tasks that may require output data from other tasks but do not communicate *during* execution), meaning of node weights, undirected vs. directed graph, ...

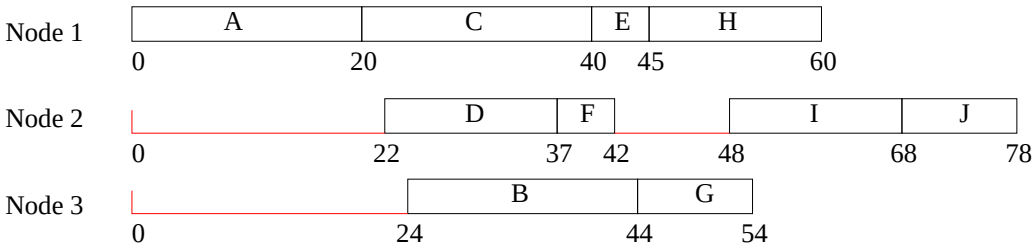
Exercise 4: Loda Distribution via Graph Partitioning

In this distribution node 1 has the processes A, E and G, node 2 has the processes B, C and F, and node 3 has the processes D, H and I. The cut between node 1 and 2 now contains the edges AB and EB with a communication load of 5. The cut between node 2 and 3 now contains the edges CD, CI, FI and FH with a communication load of 14. The cut between node 1 and 3 now contains the edges EH and GH with a communication load of 8. The sum is 27.

Exercise 5: List Scheduling

Earliest Task First (ETF):

Step	Selected node / Static level	Start time on Processor p1	p2	p3	Selected processor
1	A / 70	0	0	0	p1
2	C / 50	20	28	28	p1
3	D / 45	40	22	22	p2
4	B / 40	40	37	24	p3
5	F / 35	40	37	44	p2
6	E / 30	40	42	44	p1
7	G / 20	52	52	44	p3
8	H / 25	45	53	53	p1
9	I / 30	60	48	54	p2
10	J / 10	76	68	76	p2



High Level First with Estimated Time (HLFET):

Step	Selected node / Static level	Start time on Processor p1	p2	p3	Selected processor
1	A / 70	0	0	0	p1
2	C / 50	20	28	28	p1
3	D / 45	40	22	22	p2
4	B / 40	40	37	24	p3
5	F / 35	40	37	44	p2
6	E / 30	40	42	44	p1
7	I / 30	50	48	50	p2
8	H / 25	45	68	53	p1
9	G / 20	60	68	44	p3
10	J / 10	76	68	76	p2

In this example, HLFET results in exectly the same schedule as ETF.

Exercise 6: Process Migration

There are several state components of a process that are stored in the OS, e.g.

- the table of open files (the process could not access them after migration)
- the signal mask (the process might not handle certain signals correctly anymore)
- alarm timer set (the process would not receive the alarm signal anymore)
- etc.

So at least this information of the OS kernel would have to be transferred. Alternatively, you can send system calls back to the source node (and forward signals etc. from it to the new node).

A migration in the described way is of course only possible between computers that use exactly the same CPU architecture and OS version.

Exercise 7: Policies of Dynamic Load Balancing Systems

a) The **Transfer Policy** describes how a computer decides whether (or when) to move workload. Typically, this is determined by load thresholds. If a computer has a load that is greater than this threshold, it will surrender tasks (processes or jobs). Conversely, a node whose load is less than a threshold can request tasks from other nodes.

Once a node is set as a sender, it must choose which task to send to another node (**Selection Policy**). It must be ensured that the overhead caused by moving the task does not nullify the benefit of the move. One method would be to send new tasks that have just entered the queue. This ensures that a task that needs to be sent has not already been executed, i.e. has no state.

The **Location Policy** describes how to determine a receiver node for the tasks to be moved (or a sender node if the receiver initiates load balancing). A very simple way is to take a random node. It would be better to ask several nodes about their load and select the least loaded node as the receiver.

The **Information Policy** describes how/when the load information is determined and distributed. For example, the load can be determined regularly, or event-driven (if the load situation changes, e.g. when tasks are created or terminated). The load information can be distributed globally to all nodes or only locally to the nearest neighbors.

b) Receiver-initiated methods can result in a slowdown of the system (with low load). The performance of the system decreases.

If the system never has a high load, then usually all nodes by the transfer policy choose to be a receiver. If we use receiver-initiated procedures, each node will constantly send requests for tasks, and there are no senders to fulfill them. These task requests fill the network with unnecessary traffic and can slow down actual task transmissions, thereby reducing performance.

The latency between sending a request and a task arriving can also cause idle times on nodes, although there may be enough tasks in total to keep all nodes busy.