**CSE 221 – Graduate Operating Systems**

**Homework 2**

Name: Andreas Prodromou

SID: A53049230

**Question 1:**

**1.a )** Mesa uses WAIT and NOTIFY respectively for WAIT and SIGNAL in Hoare's paper.

In Mesa, the invariance ( I ) can be assumed at the start of an entry procedure and just after a WAIT. That means that the monitor invariant must be established just before the a return from an entry procedure or just before a WAIT.

Essentially, Mesa monitors require: I --> WAIT --> I. The WAIT call is also enclosed in a while (NOT B) loop.

In the case of NOTIFY we can't assume anything on the invariant, because no context switch to another process will occur without testing for B first. Technically, it looks like B --> NOTIFY --> B, but B's on the left and right hand sides are not necessary. No assumptions about our program can be made.

**1.b )**

|  |  |  |
| --- | --- | --- |
|  | Invariance | Condition |
| [1] | Unknown | Unknown |
| [2] | Unknown | !hold |
| [3] | hold | !hold |
| [4] | hold | Unknown |
| [5] | Unknown | Unknown |

**Question 2:**

**2.1 )** In GMS, memory is managed entirely by the operating system. Time is divided into epochs and at the beginning of each epoch a node is chosen to be the "leader". The leader then creates and broadcasts an ordering of the page information from every other node. When a node needs to send a page, it locally "rolls a (weighted) dice" and chooses the idle node to send its page. The weights come from the ordering the leader node broadcasted. Applications cannot know where was the memory they are accessing. The authors of GMS followed Lampson's first principle.

Exokernel reveals information and applications can use it to implement their own management mechanisms. (They followed Lampson's second principle)

**2.2 )** GMS chose to hide the actual location of memory pages. The memory pages accessed by an application could be on the local machine's memory, in some other machine or in the disk. However, the memory management system was transparent, so the application could not know where the page came from. The way this was implemented is described in my answer in 2.1.

Exokernel on the other hand chose to expose a large amount of information about the system: Memory, CPU, TLB, interrupts exceptions, cross-domain calls and resource allocation and revocation. To achieve this, Exokernel uses secure bindings to allow applications to securely bind to machine resources, visible resource revocation for applications to participate in resource revocation protocols, and abort protocol to allow exokernel to break secure bindings of uncooperative applications by force.

**2.3 )** In GMS, when a page fault occurs, the replacement algorithm will find the node that holds this memory page and seamlessly replace the page in the node that accessed it.

In Exokernel, applications are allowed to manipulate the TLB, in order to make locally optimized decisions (according to the application's knowledge) of which entries to be added or evicted.

**2.4 )** GMS algorithm discards the oldest page in the cluster whenever a faulted page is read from the disk. this implies that every node holds full global information about the nodes and their pages. This was impossible to maintain and the authors decided to use the algorithm described in my answer (2.1) in order to have approximate information on every node. Throughout an epoch, this aging information might not be very accurate, but the beginning of the next epoch will correct it.

One of Exokernel's issues was the need for the applications to be able to revoke secure bindings from applications and so they included a revocation protocol, which demands all the participants to agree. However, in case some application was not responsive the revocation protocol would fail. To solve this, they included an abort protocol which allows the operating system to break secure bindings on the detection of a non-cooperative application.

**2.5 )** GMS authors wanted to create a fast and scalable memory management system. Minimizing the number of assumptions that applications can make, helps in the system's scalability. Following the first principle, the authors managed to create a system that would offer significant speedup -- without any code modification -- and it was easy to use. The applications -- without any knowledge of the memory management part -- could be kept simple.

Exokernel tried to gain performance, by exposing the system's information and allowing the applications to make administrative decisions based on their local knowledge and future needs. I believe that the goal of Exokernel was technically achieved, but I also believe that the system is too complicated to be used and maintained. It's technically achieved, because if some applications needs such fine-grained control of the system it can have it with Exokernel. However, the complexity that is added to the code in order to take care of managing the system's resources would prevent people from using it.

**Question 3:**

**3.1 )** In Exokernel, the kernel domain is privileged and the web server process is unprivileged.

In L4, the L4 Server and the Web Server application domain is unprivileged, Micro-kernel is privileged.

In Xen, Xen-kernel is privileged, while DOM0, Virtual Machine and the Web Server process domains are unprivileged.

**3.2 )**

Exokernel:

When the kernel receives a packet it will identify the destination application (Web server) from the bound predicates that applications register on the packet. It will transfer the packet through an IPC.

L4:

When a packet is received, the NIC will interrupt the kernel. The kernel will then emulate an interrupt on the L4 server via an IPC. When the application (Web server) requests the packet, the L4 sever will copy the http request in the Web Server's memory.

Xen:

Xen kernel will first send the packet to Dom0. Dom0 will then determine which virtual OS is the recipient. Xen will put the packet in an unused page frame in the receive I/O ring of the recipient OS. When the packet is written, the guest OS will be notified with an interrupt and it will handle the packet and forward it to the Web server. The guest OS's drivers have been modified to support this mechanism.

**3.3 )**

In my opinion, Exokernel will be beneficial for the Web server application, since every aspect of the system is exposed. The application might be more difficult to write, however when it's written for Exokernel, it should outperform both L4 and Xen, because it will be highly optimized for the underlying system.

**3.4 )** Exokernel will first check that the virtual address is not in a special segment. In the case of our web server, this is the case, thus the kernel will let the application handle the page fault.

In L4, a page fault will be handled by the kernel transmitting it via IPC to a user level page associated with the thread that caused the page fault. The L4 server will handle the page fault, by bringing the memory page from its pool which has been allocated during boot.

In the case of a page fault in Xen, it will start handling the page fault and give the information of CR2 register's content to the guest OD so it can deal the page fault from now on. The memory page will come from the guest OS's physical memory.