Chapter 1

Introduction

1.1 Background

Today, almost any enterprise application operates on data. A part of these data often should be *persistent*, that is, should outlive the process that created it. In this definition *a process* is not an operating system process, but instead some abstract operation that creates data.

The division between persistent and non-persistent data is not dichotomic. Several degrees of persistence exist – from temporary values during expression evaluations to long-stored records in a database. For example, when a customer uses a bank card to purchase goods, the process is the withdrawal of money from their bank account. This process creates a record about the withdrawal that can be seen later in a bank app. In the case of expression evaluation, the process is an execution of a CPU instruction.

This range of data persistence closely resembles the memory hierarchy of modern computers. Such similarity is for a reason: more persistent data are stored in slower and energy-independent upper levels of memory. The presence of the memory hierarchy brings the need for moving portions of data to lower memory levels when a program needs to access the data. On the other hand, the limited size of low memory levels forces programmers to move data that is no longer needed to the higher levels to make space for new data.

Modern operating systems manage moving data between the main memory and registers automatically, without the intervention from developers of applications for those systems. However, when an application needs to access data from the secondary storage, it should read and load data to the main memory by itself. Writing a program code for loading and saving data from and to secondary storage is a burden for developers of the applications.

To remedy this management of moving data between memory layers, Atkinson and Morrison [1] propose to use persistent support systems. These systems are a software for automatic management of physical memory layers. The systems provide its users with a sandbox to run programs where all data virtually have the same persistence.

1.2 Phantom OS

The idea of persistent support systems was developed later with changing focus to operating systems [2], [3]. The most modern of them up to date is Phantom OS. This system consists of a stateless kernel and a Phantom Virtual Machine (PVM). PVM hosts processes, which execution states persist across restarts of the host machine. Persistence is achieved with periodic snapshotting. In the context of Phantom OS, a snapshot is a memory dump of the PVM memory space. The system uses the latest snapshot as a recovery point on booting.

Even though Phantom OS is a working persistent support system, there is still a room for improvement. Currently, Phantom OS does not restore the state of the Transmission Control Protocol (TCP) stack. The operating system only handles sockets in the LISTEN state. If a socket was in this state when the snapshot was taken, the system will reopen it on boot and bind to the same address. Sockets in other states will become invalid. Any attempt to use these sockets will result in an error. The application that tries to use such socket will receive this error and will be responsible for its handling. Usually, that handling implies reestablishing a connection with a remote peer and restarting data transmission from the very beginning.

Because of the presence of these errors, existence in a persistent environment is not completely transparent for Phantom OS applications. Moreover, the common way of handling these errors, which was described above, results in inefficient network use, especially when power is off for a short period of time.

1.3 Problem statement

Recently, Phantom OS was ported to the Genode OS framework. The PVM was ported as a userspace Genode component, and all PVM syscalls are implemented as functions provided by other Genode parts. In particular, the network stack was implemented in the Genode as a Virtual File System (VFS) plugin. Due to this fact and the flexibility of the Genode VFS, changing implementation of the networking stack became easier in comparison to Phantom OS before this port. My hypothesis is that it is possible to develop an enhancement to the Phantom networking stack entirely in the Genode part of the port.

The original document describing TCP [4] suggests using a Finite State

Machine (FSM) to implement the protocol. Most implementations of the protocol, including the one used in Phantom-over-Genode (PoG), use this FSM technique. The theoretical challenge behind implementation of the enhancement described above is the integration of an external state machine into the Phantom persistence model. In this context, “external” means that the code of the state machine is not running as managed code inside PVM. The first aim of this paper is to develop a methodology for integrating such state machines into Phantom OS.

The second aim of this paper is to enhance PoG TCP stack to achieve two goals. The first is to reduce the number of errors that should be handled by applications supervised by the PVM. This means that as many errors as possible should be either prevented or handled at the Genode layer of the OS. The second goal is to make the network utilization more efficient. This means avoiding extra TCP transmissions, if they are not necessary.

Bibliography cited

[1] M. Atkinson and R. Morrison, "Orthogonally persistent object systems," *The VLDB Journal*, vol. 4, no. 3, pp. 319–401, 1995.

[2] C. R. Landau, "The checkpoint mechanism in keykos," in *[1992] Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, IEEE, 1992, pp. 86–91.

[3] A. Dearle, R. Di Bona, J. Farrow, *et al.*, "Grasshopper: An orthogonally persistent operating system," *Computing Systems*, vol. 7, no. 3, pp. 289–312, 1994.

[4] P. John, "Transmission control protocol," *RFC 793*, 1981.