TU

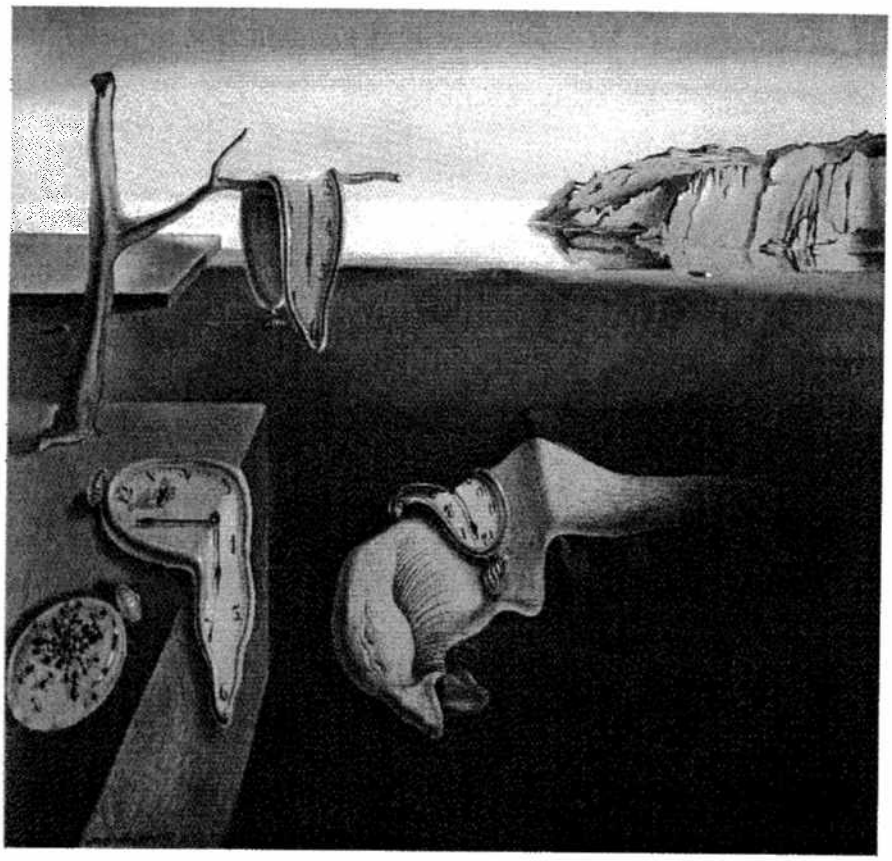
Institut für Automation
Abt. für Automatisierungssysteme

Technische
Universität
Wien

Projektbericht Nr. 183/1-109
October 2000

Mapping of a Real-Time Operating System to Unix

Bernd Burgstaller and Wolfgang Kastner



Salvador Dali, "Die Beständigkeit der Erinnerung"

Vienna University of Technology
Department of Computer-Aided Automation

Guiding Instruction
.....-.....-....-06-7659

CP Operating System
Feasibility Study
"Mapping of a Real-Time Operating System to UNIX"

Author: Bernd Burgstaller,
Wolfgang Kastner
Authorization: Univ.-Prof. Dr. Ing. Gerhard H. Schildt
Translator:
Notification:

In addition to the authors named on the cover page the following persons have collaborated on this document:

The document comprises 86 pages, all pages have issue no 06.

The document is based on template manuale.dot,

document number P30305-W5299-A001-02-7635.

This issue was created on 18.9.2000.

This document was edited with MS Word 97, drawings have been created with Visio Professional 5.0c

The path name of the main document file was study.doc.

0 GENERAL INFORMATION

0.1 Issue Control

The document comprises 86 pages, all pages have issue no 06.

0.2 History

Issue	Date	Reason for Changes
01	24.8.2000	First issue Skeleton, mapping of PROCESS primitiv.
02	30.8.2000	Mapping of basic VOCOS primitives.
03	31.8.2000	Version proposed for review.
03a	7.9.2000	Add-on for version 0.3.
04	18.9.2000	Merge between version 0.3 and 0.3a
05	18.9.2000	Incorporated review-remarks.
06	4.10.2000	Merge between version 05 of this document and version 02 of the effort estimation document which has been inserted into Section 15.3. Final version delivered by subcontractor.

Table 0-1: History

0.3 References

/Call2/ Meeting Minutes
Phone call with contractor regarding performance considerations
minutes-25082000.txt
25.8.2000

/CHILL/ Description
P30308-A2737-B000-**-7618
CHILL Language and Compiler

/Concurrency/ P30303-X0033-C699-03-7635
EWSD
User Manual
CP113 Operating System
Concurrency Control

/Control/ Controlling Software Projects
Management, Measurement & Estimation
Tom DeMarco
ISBN 0-13-171711-1 025

/Cost/ Software Engineering Economics
Barry Boehm
ISBN 0-13-822122-7

/ErrorH/ P30303-X0033-G099-**-0035
User Manual für CP-Sicherungssoftware
Kapitel G, Reg. 1
Behandlung von Softwarefehlern am CP113

/Except/ Design Specification
P30303-D2284-E100-07-76D8
CP113
Exception Handler for V11 V11C

/Heap/ P30303-X0033-C399-**-7635
User Manual for the CP Operating System
Chapter C, Section 3
Calls to Heap Management

/IPC/ P30303-X0033-C299-03-7635
CP OS User Manual
Chapter C2
Interprocess Communication (IPC)

/Mauro/ Inside Solaris
Threads Libraries in Solaris
Jim Mauro
http://www.sunworld.com

/Pflicht/ Pflichtenheft
Mapping of a Realtime-Operating System to SUN SOLARIS

/Posix1003.1b/ ANSI/IEEE Posix 1003.1b
POSIX Real-Time Extensions

/Posix1003.1c/ ANSI/IEEE Posix 1003.1c
POSIX Threads Standard

/PosixProg/ POSIX.4
Programming for the Real World
Bill O. Gallmeister
ISBN: 1-56592-074-0
O'Reilly & Associates

/PosixThread/ Pthreads Programming
Bradford Nichols, Dick Buttlar & Jacqueline Proulx Farrel
ISBN: 1-56592-115-1
O'Reilly & Associates

/Proz/ P30303-X0033-B099-05-0035
EWSD
User Manual
für das Betriebssystem im CP
Prozeßkonzept

/ProzSpec/ Entwurfsspezifikation
VISION O.N.E
CP113: Prozeßsteuerung,
Prozeßverwaltung
Prozeßtabellen
Subsystem ON, OS
Erweiterung in Vision O.N.E
P30303-B2283-E100-*-00D8

/Scheduler/ P30303-D2286-E100-03-00D8
Entwurfsspezifikation
CP113 Betriebssystem
Scheduler (Ablaufsteuerung)

/Solaris/ Better by Design-The Solaris Operating Environment
Whitepaper
Sun Microsystems
http://www.sun.com

/SolarisRT/ Scalable Real-Time Computing in the Solaris Operating Environment
Whitepaper
Sun Microsystems
http://www.sun.com

/Stall/ Operating Systems
Internals and Design Principles
Third Edition
William Stallings
ISBN 0-13-887407-7

/Startup/ P30303-X0033-C199-**-0035
User Manual für das Betriebssystem im CP
Kapitel C1
Aufrufe an die Start-/Endebehandlung

/UML/ Unified Modeling Language in a Nutshell
Sinan Si Alhir
First Edition
ISBN 1-56592-448-7

/Update/ P30303-X0033-H009-**-7635
User Manual for the Operating System in the CP
Chapter H
Interface for Updating, Main and Background Memories

/VxGuide/ VxWorks
Programmer's Guide
Edition 1, Version 5.4
Wind River Systems
http://www.windriver.com/products/html/vxwks54.html

/VxRef/ VxWorks
Reference Guide
Edition 1, Version 5.4
Wind River Systems
http://www.windriver.com/products/html/vxwks54.html

0.4 Glossary and Abbreviations

API	Application Programmers' Interface
	Interface available to application programs in general.
APM	ATM Processor (Bridge between SSNC and CP)
APS	Anlagenprogrammsystem
	The totality of all SW which is loaded into the processors that make up an EWSD or EWSX switch. The APS is made up of several part-APSes which contain all SW for a specific type of processor (e.g. CP, MP).
BAP	Base Processor of CP113
CAP	Call Processor of CP113
COCOMO	COnstructive COst MOdel
CP113	Coordination processor 113
	A Motorola 68020/68040 based multiprocessor used in EWSD which is mainly programmed in CHILL.
EWSD	Elektronisches Wählsystem Digital
	A Siemens Solution O.N.E product line for narrowband switching systems.
EWSD Broadband Node	
	An EWSD switch with additional broadband switching capability which will be available in future EWSD versions.
EWSD Powernode	
	An EWSD switch with an enhanced #7 signalling system which will be available from EWSD Version 13.
EWSX	Elektronisches Wählsystem Express
	A Siemens Solution O.N.E product line for broadband switching systems.
IDS	Interactive Debugging System
IOP:SCDP	IO-Processor for X.25
IPC	Interprocess Communication
LOC	Lines of Code
LTG	Line Trunk Group

	Peripheral processor in EWSD used as interface for customer and inter-office lines.
LWP	Lightweight Process
MM	Man Month
MP	Main Processor.
	An Intel based monoprocessor used in the SSNC of the EWSD Powernode, and also in the EWSD Broadband Node and in EWSX. The MP is mainly programmed in CHILL.
MY	Man Year
OSA	Operating System Abstractionlayer
PIF	Published Interface (see section 2.1)
PCP	Peripheral processor in EWSD used as interface for customer and inter-office lines.
Pthread	POSIX thread
RTOS	Real-Time Operating System
SPU	Service Provision Unit (see section 2.1)

Siemens Solution O.N.E

	Overall term indicating product palette of the Siemens Public Network division.
SP	Synchronization Point
SSNC	The part of the EWSD Powernode that performs #7 signalling.
SVC	Supervisor Call
	Feature of Siemens CHILL by which application programs invoke Operating System functions that execute in the supervisor mode of the processor.
SWET	Software Error Treatment
SWSG	Software Safe-Guarding
TBD	To be defined

VISION O.N.E SW structure

	SW structure defined for several Siemens Solution O.N.E product lines which involve a very large amount of SW, such as EWSD and EWSX.
VOCOS	VISION O.N.E CHILL Operating System
	Operating System supporting the VISION O.N.E SW structure for the CHILL processors (CP113 and MP).
VOCOS/CP	VOCOS variant for the CP113.
VOCOS/MP	VOCOS variant for the MP.
WBS	Work Breakdown Structure
WCP	Worst-Case Performance

0.5 Definitions

Priority-based Scheduling: Each process is assigned a priority, highest priority process is scheduled first.

Preemptive Scheduling: Type of scheduling where the scheduler interrupts and suspends the currently running process in order to start or continue running another process if that other process is of higher priority and (suddenly) becomes runnable.

Cooperative Scheduling: Contrasts preemptive scheduling as each task (or process) must include calls to allow it to be de-scheduled periodically.

Transient Data: Data defined at module level that is read- and write-able.

Semi-permanent Data: Data defined at module-level through the /CHILL/ `PROTECT` option. This kind of data is generally read-only to all processes. It can be only modified through the `UPDATE` function (cf. /Update/).

Permanent Data: Data defined through the /CHILL/ `READ` attribute with `PERMANENT` option. Only read-only access possible.

0.6 Notation

The overall approach throughout this study was to take a VOCOS/CP primitive, explain in brief what it does, explain the options provided by POSIX and present then how within these options this primitive can be resembled. Although most of the document contains plain English prose, we chose to use sequence diagrams to capture dynamic behavior (cf. /UML/).

0.7 Keyword/Descriptor

Operating System
VOCOS

0.8 List of Figures and Tables

1 Management Summary

This study examines the possibilities available to offer the VOCOS OS functionality on a commercial platform. It was considered as a main goal to abstract from proprietary solutions by using a more abstract base as provided by the POSIX (Portable Operating System Interface) standards.

As it is suggested by this study, it is possible to map the core VOCOS functionality (process management, scheduling, IPC, concurrency-related features, Recovery, Error-Handling, Startup, and Heap-Management) onto POSIX primitives. It must however be noted that this mapping is not always straight-forward, and that it is not always possible to achieve 100% compliance.

Moreover there exist certain aspects of VOCOS that are highly depending on the target hardware, and it is therefore still open, whether the new target hardware will provide this feature, and whether the target RTOS allows access to this feature. Finally it must be noted that due to time constraints it was necessary to focus this study on important aspects of VOCOS and to leave out other, presumably less difficult aspects. These aspects certainly impose a certain amount of risk on the project.

Several topics have been identified so far that allow more then one possible solution. A decision is needed in that areas. To facilitate the process of decision-finding we recommend to execute simulations that aid in judging the feasibility of each solution as well as in analyzing the impacts on the system.

Generally there can be no objection found that would prohibit the emulation of a given software system by another software system. It is therefore rather a question of whether the given requirements can be met under the imposed constraints. As requirements we denote the needs of the VOCOS application- as well as supervisor software, and by constraints we consider the need to come to a solution within reasonable time as most stringent. The required effort for the project must certainly be considered in terms of man-years rather than man-month, and it is highly dependent on the software process of the organization carrying out the project. Based on the facts identified in this study the respective organization has therefore to provide an estimate.

A go/no-go decision for the whole project at the present level of coverage must be considered too early, the study can however be considered promising enough to justify the effort needed to carry out the next steps such as solving the open points where this study currently offers several possible solutions and implementing a prototype of the most critical features of the system.

2 Scope and Approach

This document describes the outcomes of the feasibility study aimed at replacing the VOCOS operating system by a commercial RTOS. For details concerning the scope of the study cf. /Pflicht/.

2.1 Approach

In order to hide the underlying target RTOS from the application software, an additional layer, called the Operating System Abstraction Layer (OSA), is introduced. The purpose of this layer is to resemble the behavior of the VOCOS/CP OS in mapping its primitives onto primitives of the target OS. In order to retain flexibility regarding the choice of the target RTOS an abstraction based on POSIX, the Portable System Interface, is chosen to place OSA upon. Since it becomes more and more common for commercial RTOS vendors to support POSIX, there will become more and more platforms available that are capable of hosting the resulting software system. Among the more prominent vendors already supporting the POSIX standard are Windriver Systems with VxWorks and SUN with SOLARIS. As specified in /Pflicht/ the current version of this study aims at an implementation atop of the POSIX interface of SUN SOLARIS. As soon as the target RTOS and hardware have finally been chosen it has to be examined whether a division of the OSA into a part executing on user level and another kernel-level part (executing in supervisor mode through traps) is feasible. The current version of this document does not require an OSA kernel, although it can be useful (e.g. to avoid race conditions between processes of different priorities executing concurrently within OSA).

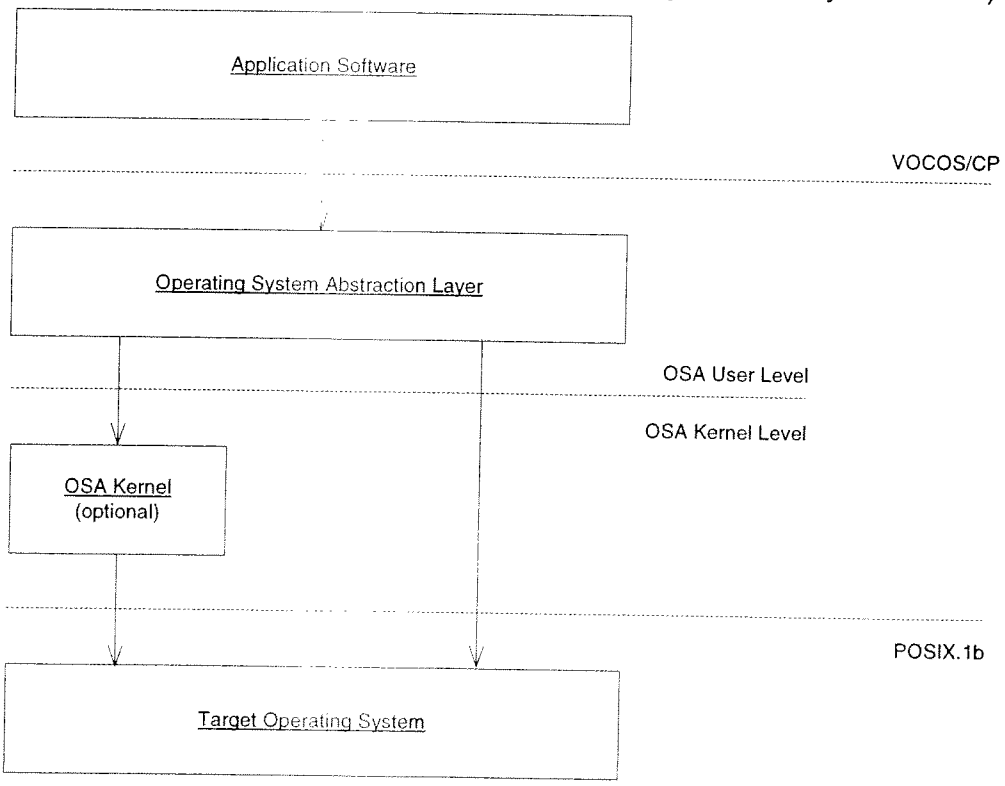


Figure 2-1: Software Layers

2.2 Structure

Section 3 introduces the reader to the POSIX standard, the following sections are dedicated to distinct features of VOCOS/CP and their mapping onto POSIX primitives. Section 9 addresses performance issues, Section 15 identifies several risks, gives a work breakdown structure of the next steps that have to be carried out, and lists some effort figures, Section 16 finally presents a summary and conclusion.

2.3 Coverage

Due to the sheer size of VOCOS/CP it was not possible to cover *every* aspect of the magnitude of functionality offered therein. Preferred have therefore been what are usually regarded the corner stones of an operating system: process management, scheduling, synchronization, concurrency, interprocess communication, and timer management.

2.4 General Assumptions

In addition to the requirements stated in /Pflicht/ the following assumptions have been made:

2.4.1 Hardware

Although VOCOS is a multiprocessor OS, the new target hardware will not have multiprocessor capabilities. The system comprises one BAP-M, how this will be integrated with a BAP-S is yet TBD.

2.4.2 Hardware Architecture

Intel hardware architecture favored due to simplified rework of CHILL compiler backend.

3 POSIX

3.1 Overview

POSIX, the Portable Operating System Interface, is an evolving set of standards being produced by the IEEE and standardized by ANSI and ISO. The goal of those standards is the portability of application software at the source-code level. POSIX standards addressing operating system issues are closely related to (or based on) UNIX. The ratified POSIX standards that generally pertain to real-time OS's consist of:

1003.1: Nucleus of common OS functions like process-, filesystem- and device API (basic UNIX system calls).

1003.2: Utilities (like the UNIX shell,...)

1003.1b: Real-time extensions of POSIX 1003.1 like shared memory, priority scheduling, semaphores, real-time signals, message queues, synchronized I/O, time management

1003.1c: Provides the ability to run multiple, concurrent threads of execution (pthreads) within a single POSIX process.

POSIX standards are explained in terms of interface descriptions and contain therefore not more than what can be covered by such descriptions. Not every aspect is tightly specified, there exist areas of implementation-defined behavior and even undefined behavior (e.g. specifications in the time domain). However, on the whole POSIX provides a sound base that facilitates porting of applications between operating systems, which is by far superior to reverting to proprietary solutions.

Regarding proprietary solutions it has to be noted that it is common practice within OS vendors to have the POSIX API co-exist with their proprietary API. In this way it is possible to a certain extend to mix (or interchangeably use) features of both APIs. This also provides a fallback-solution if a certain POSIX feature, after all, should turn out to be inadequate for the needs of the application.

Care must be taken with standard 1003.1b, since it is structured as a set of options (cf. Table 3-1), and it is sometimes considered good practice by OS vendors to claim POSIX compatibility while only standard 1003.1 is covered and the options regarding 1003.1b are 'switched off'. The vendor-specific POSIX conformance statement is therefore an important document.

3.1.1 Shorthand Notation for POSIX Standards

Since this study addresses only OS-specific features of the POSIX standards, we use the following shorthand notation for standards: **POSIX.<x>** where **POSIX** stands for POSIX 1003, and <x> is usually one of ('1', '.1b', .'1c').

3.2 POSIX Conformance of Commercial RTOSs

3.2.1 SOLARIS

According to /SolarisRT/ POSIX.1b is supported in SOLARIS Version 2.6 or later. For POSIX.1c the following table is given:

POSIX Feature	OS Version		
	2.6	7	8
_POSIX_THREADS	Yes	Yes	Yes
_POSIX_THREAD_ATTR_STACKSIZE	Yes	Yes	Yes
_POSIX_THREAD_ATTR_STACKADDR	No	Yes	Yes
_POSIX_THREAD_ATTR_PRIORITY_SCHEDULING	No	Yes	Yes
_POSIX_THREAD_ATTR_PRIO_INHERIT	No	No	Yes
_POSIX_THREAD_ATTR_PRIO_PROTECT	No	No	Yes
_POSIX_THREAD_ATTR_PROCESS_SHARED	Yes	Yes	Yes
_POSIX_THREAD_SAFE_FUNCTIONS	Yes	Yes	Yes

Table 3-1: Compliance of SOLARIS Threads to POSIX.1c

Therefore if POSIX threads are to be used SOLARIS Version 8 is recommended, otherwise version 2.6 should be sufficient.

3.2.2 VxWorks

Not considered at great detail yet, a few already known deviations are listed below:

- POSIX processes are not supported, instead a model comparable to threads, called tasks, is provided.
- Semaphores do not provide _POSIX_PRIORITY_SCHEDULING (i.e. they will not be unblocked according to the process priority).

4 Processes

Several approaches exist to map the /CHILL/ PROCESS primitive, each of which with its own advantages and disadvantages.

4.1 PROCESS as a Regular POSIX-forked Process

When mapping the PROCESS primitive defined in /CHILL/ it has to be noted that on a single legacy CP all processes run in the same address space. Creating a process as defined in /PosixProg/ via the `fork()` SVC is different in that respect since the new process runs in a separate virtual address space which is an exact copy of the parent's address space (the parent is duplicated). Although forking a process does not double the memory requirements imposed on the OS at once, at least as soon as the new process tries to access a portion of a code or data segment, the OS will generate a page fault and map the corresponding page into the address space of the newly created process (demand paging). Since this mechanism represents a serious indeterminism in the time-domain of the OS, it is usually circumvented by locking the pages of a process into main memory upon process creation time or before the process enters a critical section. /Posix1b/ foresees two ways of locking pages into memory:

`int mlockall(int flags)` attempts to lock the complete code, date, heap, and stack of a process

`mlock(void *address, size_t length)` allows to lock a given address range of the virtual address range

Although *`mlockall()`* is the more convenient way to lock the memory of a process, it has to be noted that in this way the amount of required memory is doubled, which can be a burden especially with big statically linked applications where each process accesses only a (small) part of the duplicated code/data thus creating huge areas of unnecessarily locked memory.

Figure 4-1 gives an example of such a statically linked application that contains code for two processes named *P1* and *P2* and a section of code shared between those processes. The leftmost image corresponds to the binary as it resides on the disk drive. Furthermore it is depicted how pages of virtual memory are mapped to physical memory for *P1*. After a new process is forked (*P2*), a new virtual address space containing the whole image is generated (rightmost shaded area). Note that despite the fact that the images for *P1* and *P2* are in fact the same the operating system does not treat them as such. Consequently the Common Data section is not common anymore (it would have to be declared as shared memory if this was desired). Furthermore it can be seen that in this example there are not enough pages of physical memory left to lock the new process into memory thus inducing additional run-time overhead due to page-faults.

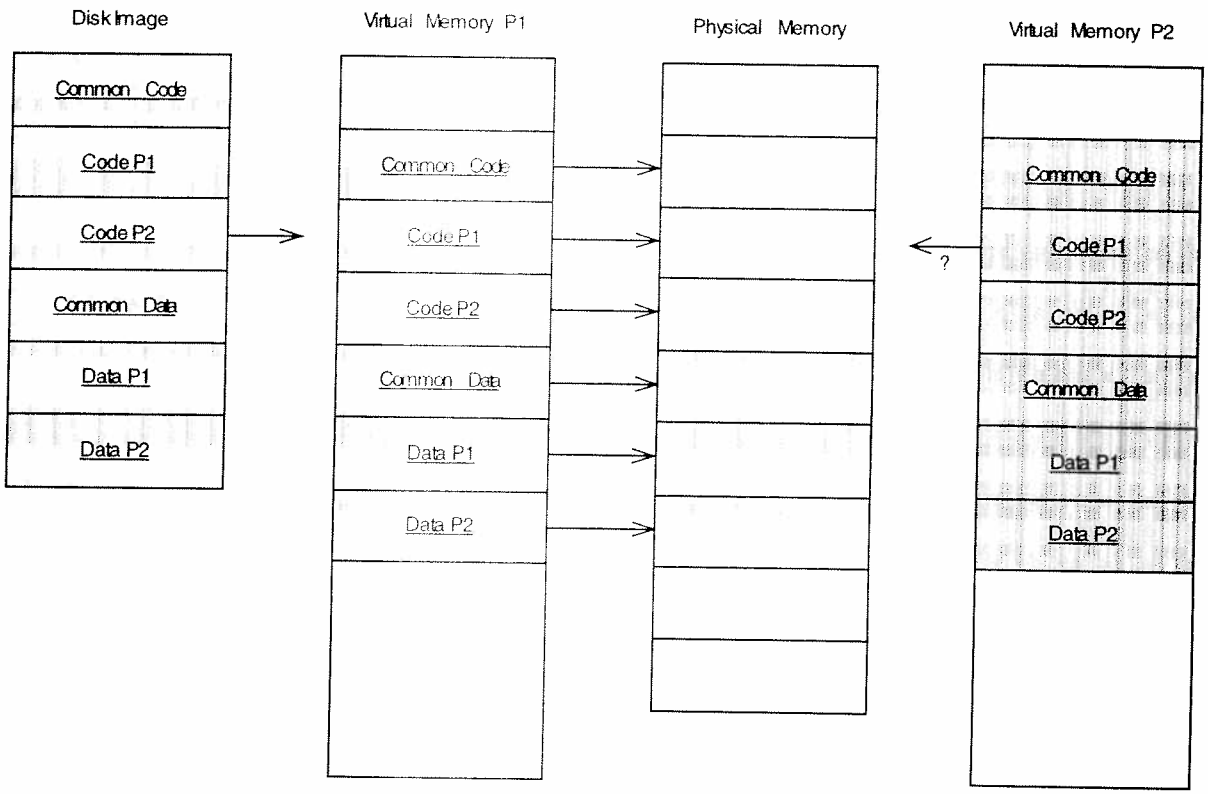


Figure 4-1: Memory utilization of process fork() from a statically linked application.

`mlock()` would provide us from unnecessarily locking memory, but it is a tedious task to determine a PROCESS' s particular pages and lock them separately.

4.1.1 POSIX-forked Processes Using *exec*()*

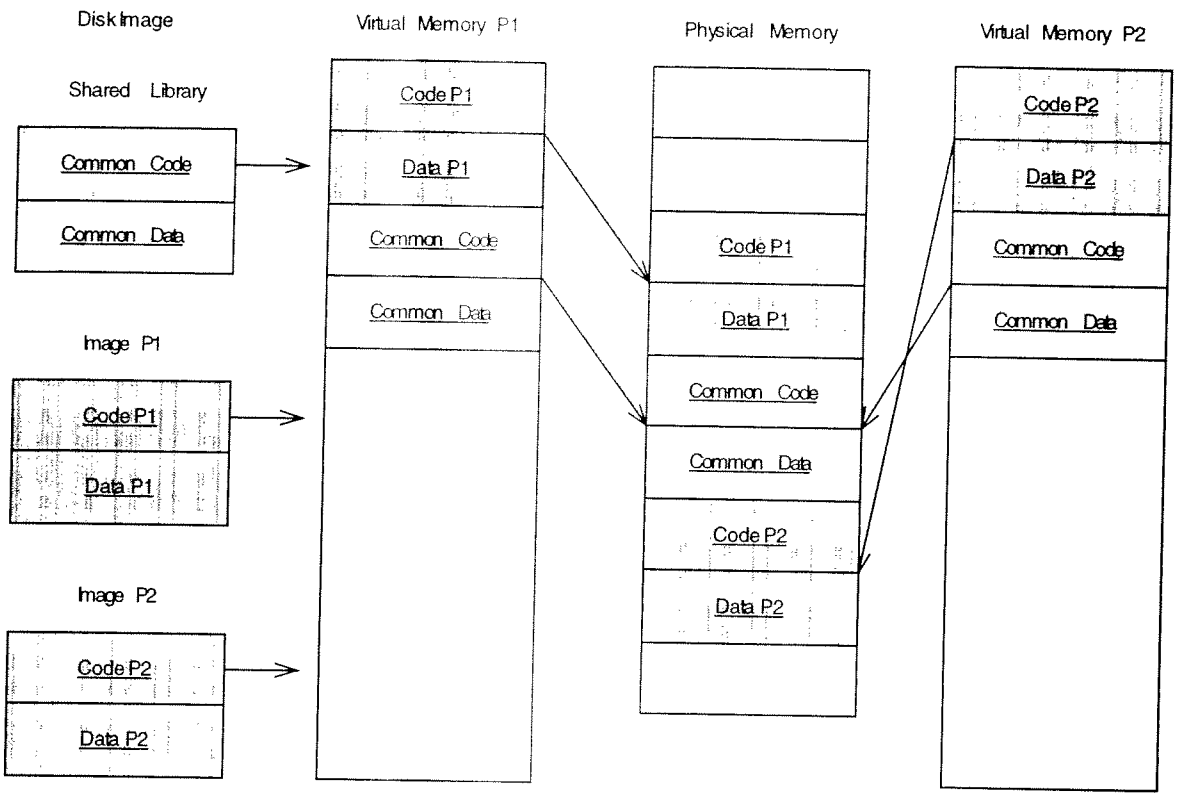


Figure 4-2: Memory utilization of process fork() with exec and shared library.

In order to overcome the problem of duplicating one huge binary image in memory once for every created process, the binary can be partitioned into separate processes started through the `exec*()` SVC. Unlike `fork()`, this SVC reinitializes the memory space of the newly created process with a new executable program which has to exist as a separate binary image on the file-system. Note that for this approach to work it is necessary that code common to several processes (e.g. library routines) has to be placed in shared libraries. Linking such code statically to the process would again result in waste of (locked) memory. Figure 4-2 shows this approach. The image is now split up into a shared library containing common code and data, and two process images *P1* and *P2*. Each code/data section resides now exactly once in physical memory, and the common code/data of the shared library is mapped into the virtual address space of *P1* and *P2*.

Figure 4-3 shows the corresponding sequence diagram that leads to the activation of *P1* and *P2*. It starts with the user typing "P1<enter>" on the system console[1]. The shell itself forks a new process that is going to be *P1*. On behalf of this SVC the OS kernel allocates the corresponding data structures for a new process and creates it. The PID of the new process is returned to the shell as a result of the `fork()` SVC. As soon as the new process is scheduled for the first time, it issues SVC `execve()` in order to overlay its virtual address space with the image of *P1*. The OS loads itself loads the shared library needed by *P1*[2]. Thereafter *P1* issues SVC `mlockall()` to lock all its memory pages into physical memory. Later *P1* itself forks a

[1] Assuming P1 is the root process of the application.
[2] In case of Solaris, this is determined by the environment variable LD_BIND_NOW. Other operating systems might follow different strategies in that respect (e.g. with Linux the process itself consults the dynamic linker to acquire all shared libraries before it starts to execute).

new process to start *P2*, which is done according to the same scheme, except that the shared library common to *P1* and *P2* is already loaded and has only to be mapped into the virtual address space of *P2*[3].

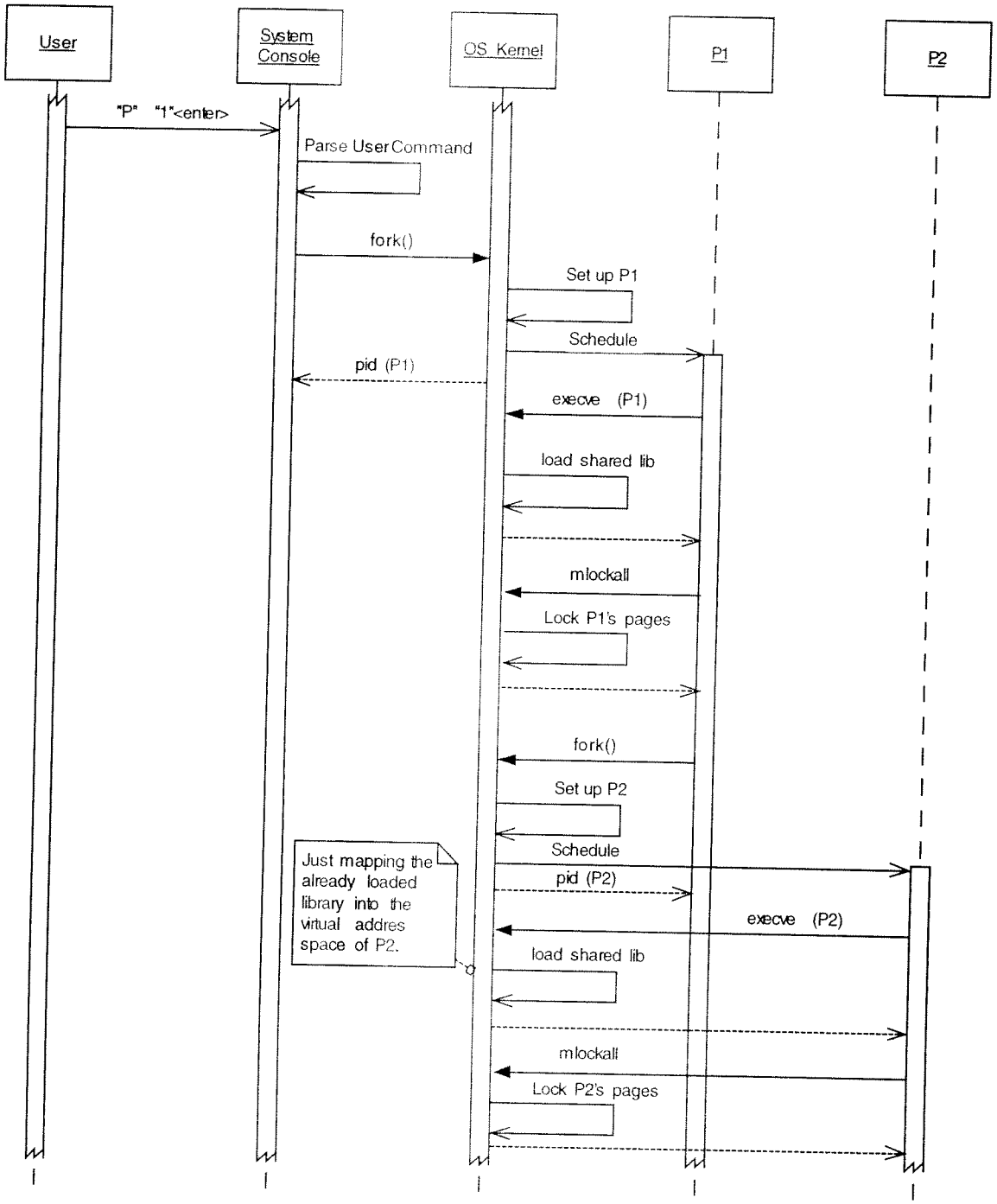


Figure 4-3: Dynamic aspects of a process fork()

[3] It is TBD how to treat the 'Initialisierungsprozedur' of a process in case of separated address spaces. One approach is to put data that ought to be initialized into a shared library.

4.2 PROCESS as a POSIX Thread

/Posix1003.1c/ describes the interface for light-weight threads within a regular process that share global data (global variables, files, etc.) but maintain their own stack, local variables and program counter. Since their context is smaller than the context of processes (e.g. no separate virtual address space), these threads are called 'light-weight'. There are two broad categories of thread implementation: user-level threads (ULTs) and kernel-level threads (KLTs). /Posix1003.1c/ leaves it open whether threads are implemented within the kernel or on user-level, even combined approaches are possible.

4.2.1 POSIX User-Level Threads

With a pure ULT implementation the complete thread management is done by the application and the OS kernel is not aware of the existence of threads. This leads to the following disadvantages:

1) A blocking system call of one thread blocks not only this single thread but all threads within that process.

2) A ULT implementation cannot take advantage of a multiprocessing environment, since the kernel can only assign the whole process and not threads of it to a processor.

4.2.2 POSIX Kernel-Level Threads

With KLTs all thread management is done by the kernel, even scheduling by the kernel is done on a thread basis. The kernel maintains context information for the whole process as well as for the threads within that process. In this way the two disadvantages of ULTs are overcome. The disadvantage of KLTs on the other hand is that each thread-related activity (such as transfer of control between threads) requires an SVC to the kernel, thus inducing increased runtime overhead. Although this overhead can generally be expected to be smaller than the overhead required for processes, it is not completely in line anymore with the original intention for threads as a means of *light-weight* processes.

4.2.3 POSIX Threads a'la SUN SOLARIS

Figure 3-4 which is taken from /SolarisRT/ shows the process- and thread concepts of the Solaris OS.

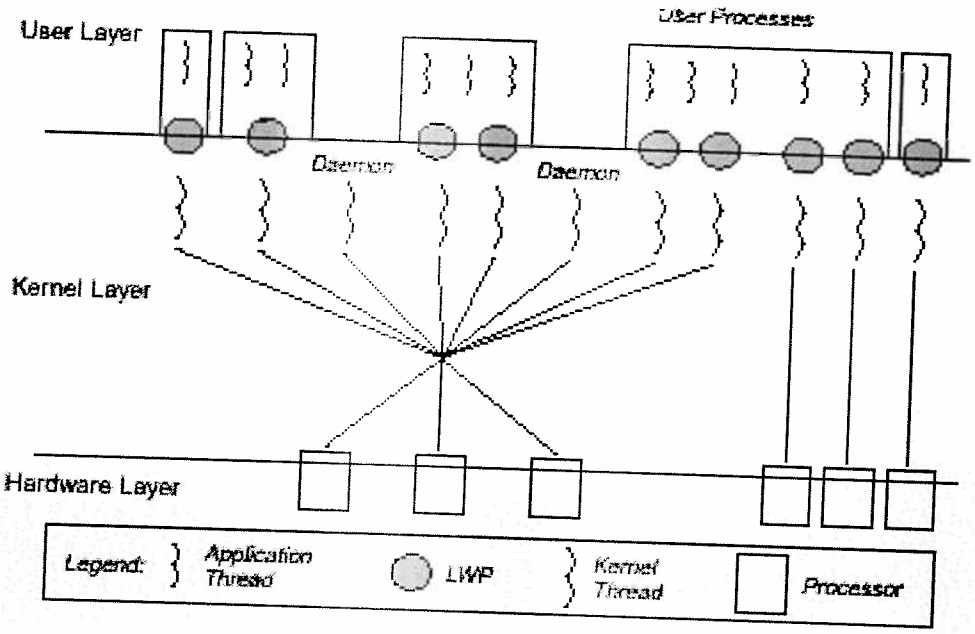


Figure 4-4: SUN SOLARIS Thread Model

Solaris makes use of four distinct thread-related topics:

Process: Equivalent of the UNIX process including address space, stack, and process control block.

User-level thread: Implemented through a threads library in the address space of a process and therefore invisible to the kernel.

Lightweight process: A lightweight process (LWP) is a mapping between ULTs and threads of the kernel. Each LWP supports one or more ULTs and maps to exactly one kernel thread. Each LWP is scheduled by the kernel independently, LWPs of one user-level process may even execute on different processors.

Kernel threads: Fundamental units for Solaris scheduling.

As can be seen in Figure 4-4 this concept offers great flexibility to the user: the leftmost user process contains one ULT that maps to one LWP. This corresponds to a single-threaded program or a POSIX process in other operating systems. The next user process to the right maps two ULTs to one LWP which exactly corresponds to the concept of ULTs introduced in Section 4.2.1. Further user processes show how to map one or more ULTs to one LWP. If each ULT is assigned one LWP then this corresponds to the concept of KLTs introduced in Section 4.2.2.

It is also worth noting that Solaris supports the concept of *processor sets* to which single LWPs or even whole (POSIX) processes can be bound to (in case of a whole process this simply means that all LWPs of the process are bound to the processor set). This is also illustrated in the above example where the three leftmost processors form a processor set to which the eight leftmost LWPs are bound to. A processor set can contain as little as one processor, and serve as little as one LWP. The SOLARIS system call `int pset_bind (psetid_t pset, idtype_t idtype, id_t id, ...)` binds the to-be-bound primitive `id` to the processor set specified by `pset`. `idtype` determines the type of `id` (LWP, process, ...).

Furthermore it is even possible to bind single LWPs or even whole (POSIX) processes to single processors (as illustrated by the rightmost user process in Figure 4-4) through the SOLARIS system call `int processor_bind(idtype_t idtype, id_t id, processorid_t processorid, ...)`, where `idtype` determines the type of the to-be-bound primitive `id` (LWP, process, ...), and `processorid` denotes the processor to which `id` should be bound to.

4.3 PROCESS as a Proprietary RTOS Primitive

Besides POSIX threads and processes RTOS manufactures have found ways to express concurrency within application programs. A common approach is to have several threads of control executing within **one** (virtual) address space. Those threads of control are often called *tasks*. Although they are similar to POSIX threads with respect to their memory address space, inter-task communication and synchronization is often provided by means of the corresponding POSIX inter-process communication and synchronization primitives (cf. e.g. /VxGuide/).

4.4 Summary and Decision-Aid

Disadvantages of POSIX Threads:

- POSIX leaves to implementation whether ULT or KLT -> less portable
- POSIX makes no assumption with respect to multi-processor hardware (also for processes, but 'process' as unit of scheduling more common and rigidly defined than 'thread').
- no inter-thread communication foreseen->has to be implemented 'by hand'[4].
- POSIX processes on the other hand are accessible on system level: they can be started, suspended, stopped, profiled from the system console. Depending on the implementation threads are more or less hidden within a process.

Disadvantages of POSIX Processes:

- Separated address space between processes induces a run-time overhead which is not required since current VOCOS implementation does not support it.
- Partitioning the whole application into shared libraries and process-images is a major software architecture change the feasibility of which has yet to be determined.

Disadvantage of the proprietary task-model

- Not POSIX-compliant

Overall the task-model seems to be the method of choice. The reduced effort to implement it might make it worth to sacrifice POSIX compliance in that respect. Note however, that e.g. Solaris does not provide a task model and would therefore become a non-option!

[4] Or one resorts to a proprietary solution.

5 Scheduling

5.1 Priorities

POSIX.1b requires an implementation to support at least 32 priority levels for SCHED_FIFO. They can be queried by the calls

- `sched_get_priority_min(SCHED_FIFO)`
- `sched_get_priority_max(SCHED_FIFO)`

In addition to the 16 VOCOS priority levels given in /Proz/ five new priority levels are introduced. Note that for priority levels that fall in between two adjacent VOCOS priority levels, decimal numbers are used. As a matter of fact the resulting range of priorities (integer and decimal numbers) have to be mapped in a second step (cf. below).

- 16: Bigger than every existing priority in VOCOS. Used to boost a process to a level uninterruptible for other processes.
- 17: Priority above all user-processes in the system, assigned to OSA.
- 18: Highest priority in the system, assigned to the OSA software watchdog.
- 12.5: Priority used for CALLP to support special scheduling mode during booting (cf. Section 5.2).
- 1.5: Needed by the PAUSE SVC. This priority level is chosen as a separation between priority-levels too small to make use of a PAUSE SVC (priorities 0 and 1), and those priorities that are high enough (>=2) to preempt a process offering to be paused by SVC PAUSE.
- -1: Smaller than every priority currently supported by VOCOS. Used to lock-out a process from the processor (in fact the system idle process locks out processes of this level).

The resulting 20 priority levels have to be mapped accordingly onto the levels provided by the underlying POSIX system. Note that POSIX leaves the actual values open, therefore this mapping is implementation-defined, throughout the remaining document only the priority-levels of VOCOS are therefore used.

5.2 Scheduling Algorithms

5.2.1 POSIX

With POSIX, processes run with a particular scheduling policy and associated scheduling attributes. Policy and attributes can be changed on a per-process base[5]. The following scheduling policies are supported:

SCHED_FIFO: Preemptive, priority-based scheduling. The only attribute available with SCHED_FIFO is the priority of a process.

[5] VxWorks is not compliant in that respect since the scheduling policy can only be set on a system-wide base.

SCHED_RR: preemptive, priority-based scheduling with time-quanta according to round-robin principle. Since with POSIX it is not foreseen to change time-quanta the only attribute available with SCHED_RR is again the priority of a process.

SCHED_OTHER: policy left to the implementation (e.g. time-sharing scheduling policy).

5.2.2 VOCOS

In principle the VOCOS scheduling policy can also be called *preemptive priority-based*, since processes are selected based on their priority, and are left executing until they give up the processor due to a blocking I/O-, semaphore-, etc. call or till they are preempted ('vertagt') by a process of higher priority (cf. Scheduler). In addition VOCOS priorities are assigned statically through the PROCESS_INFO attribute <PRIORITY> of the MDH (cf. /CHILL/).

This can be resembled by the POSIX policy SCHED_FIFO. However, /Scheduler/ Section 3.3.3 specifies 4 different variants of this policy, depending on the current mode oft the OS. In order to resemble those modes the only way of influencing a POSIX SCHED_FIFO compatible scheduler is by dynamically shifting process priorities through the POSIX sched_setparam() SVC.

Normal Scheduling: equal to SCHED_FIFO.

Special Scheduling During Boot on BAPM: During this mode the CALLP (level 7) is favored over processes within levels 8-12. Only every SPCDELAYth time is the process with the highest priority within levels 7-12 selected. Processes of levels > 12 are not affected by this exception. It is questionable whether this VOCOS-specific behavior can be exactly resembled on the target RTOS (one way would be to shift the priority of the CALLP temporarily from 7 to 12.5 and back), but it is on the other hand TBD whether scheduling during boot has to be resembled exactly and at any price.

Reduced Scheduling During NSTART0: In this phase the priorities in decreasing order are: CALLP, OSAIM, IDLE. All other processes are locked out, which can be resembled by assigning them a priority of –1. Processor utilization by OSAIM is enforced by the DROSSEL-mechanism.

Scheduling During STOPCALLP on BAPM: In order to prohibit nesting calls of SVC STOPCALLP on BAPM[6] a special scheduling mode is entered on the BAPM after the first STOPCALLP has been issued and the calling process has been blocked. In this mode only the CALLP and IDLE processes are allowed to gain the processor. According to /Scheduler/ Section 3.3.3 this is realized that after blocking the calling STOPCALLP process it is checked whether CALLP is the first process in its ready-queue. If this is the case, it is activated until it blocks at one of the SVCs IORECEIVE, OUT, or XXREGEN. If CALLP is not the first in the queue, it is assumed that CALLP is blocked in a consistent state.

It is questionable whether peeking in the ready queue of the target RTOS will be possible. TBD: what if we allow CALLP at any time to run until it blocks by shifting its priority temporarily to 16 as depicted in Figure 5-2? **NOTE**: This assumes that CALLP does not share any region or lock with another process that prevents it from reaching a consistent state at one of the IORECEIVE, OUT, or XXREGEN SVCs. If this was not the case, then the processes CALLP depends upon will also gain the processor. Or we allow the whole process ensemble to go on until we catch CALLP (without boosting CALLPs priority)!!!

[6] SVC STOPCALLP is only allowed on BAPM.

5.3 Drossel Mechanism

/Scheduler/ Section 3.3.5 explains how the CALLP on the BAPM is sometimes restricted from reading its IORECEIVE input list in order to free the processor in favor of O&M processes of lower priority. In case of enough CAPs coping with the whole call-processing work the CALLP on the BAPM is prevented from reading its input list at all. Since for the feasibility study no CAPs are planned, we have to model the case where time on the BAPM is divided into two time-slices namely NVT and VT. During NVT the Drossel bit is set which prevents CALLP from reading its input list. During the VT time-slice reading of the input list is possible. The algorithm that toggles the Drossel-bit can be implemented within OSA, `IORECEIVE` calls would be processed accordingly.

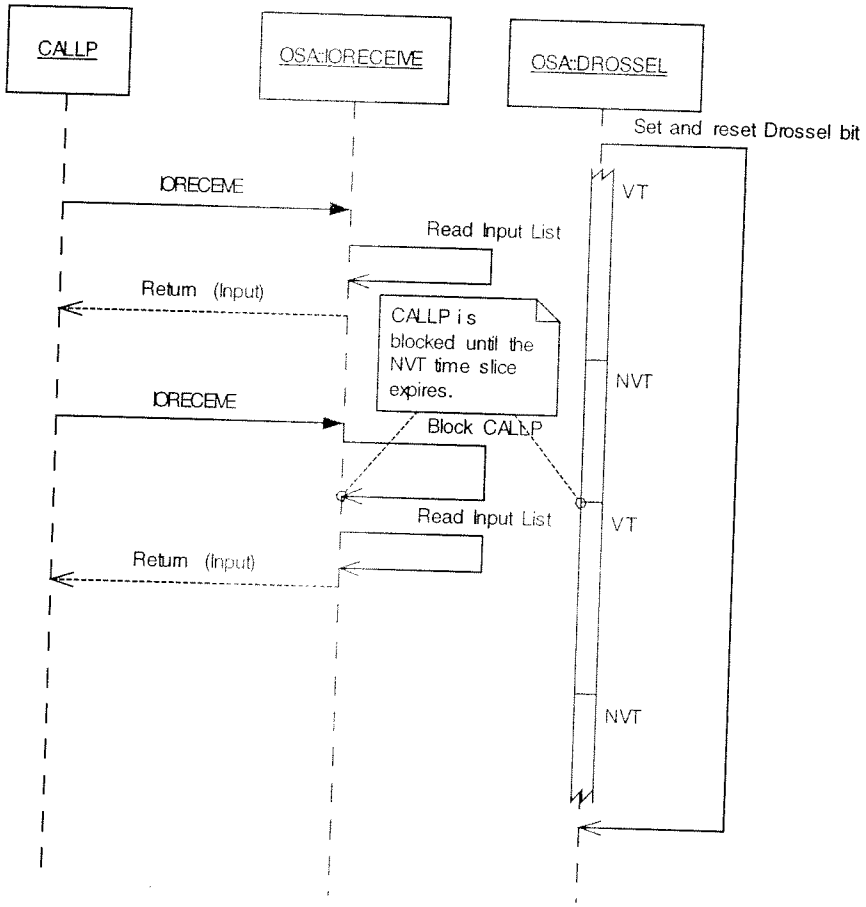


Figure 5-1: Drossel Mechanism

5.4 STOPCALLP, GOCALLP

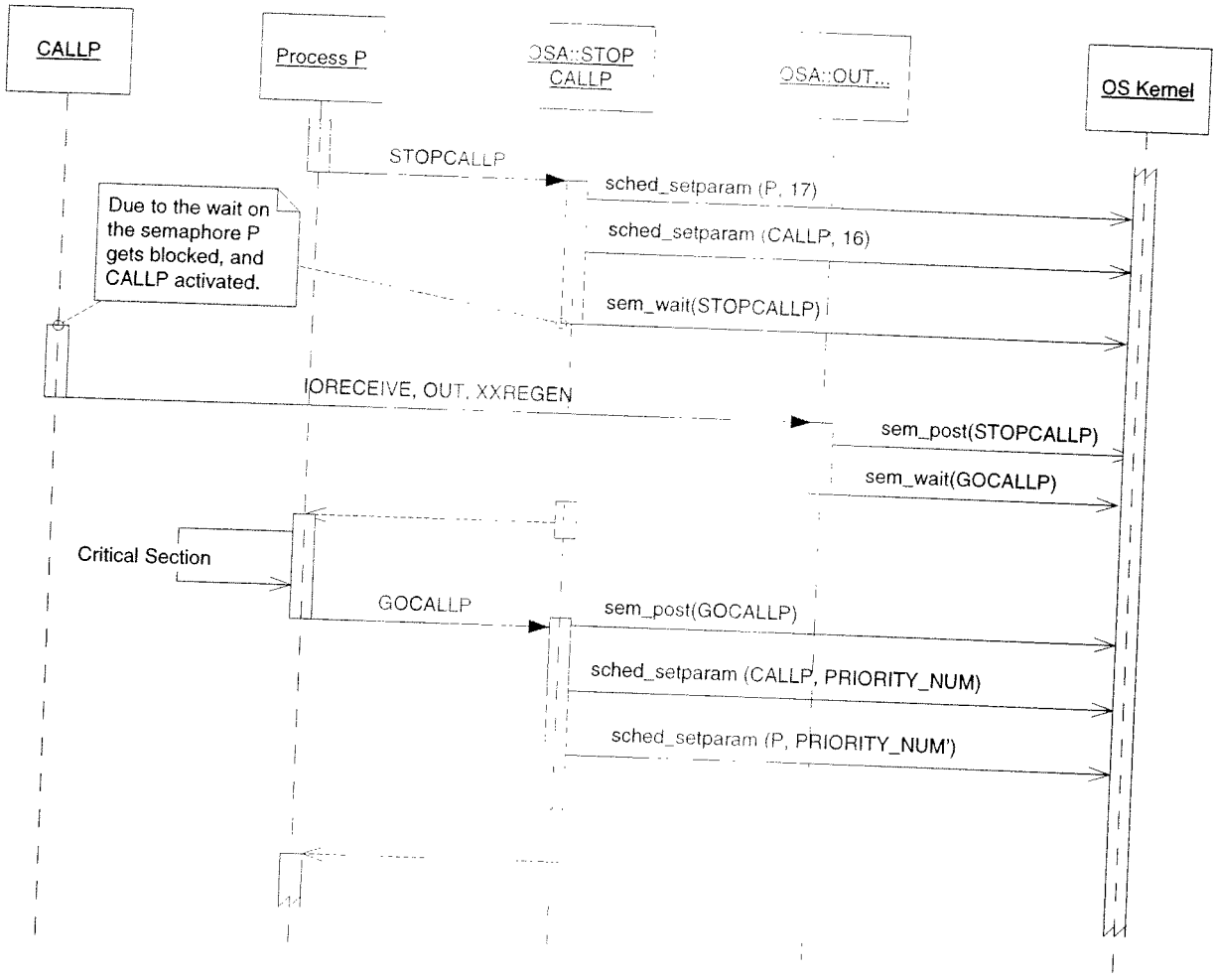


Figure 5-2: STOPCALLP/GOCALLP SVCs

5.5 SCHON/SCHOFF

SVC SCHOFF is used to switch the scheduler off. The calling process keeps the processor even if processes of higher priority become 'ablaufbereit'. SVC SCHON switches the scheduler on again. For target RTOSs where the scheduler cannot be switched off[7], the same effect is achieved by assigning the calling process the highest priority in the system. Thus there is no need for the scheduler to reclaim the processor from this process.

[7] With VxWorks the scheduler is switchable.

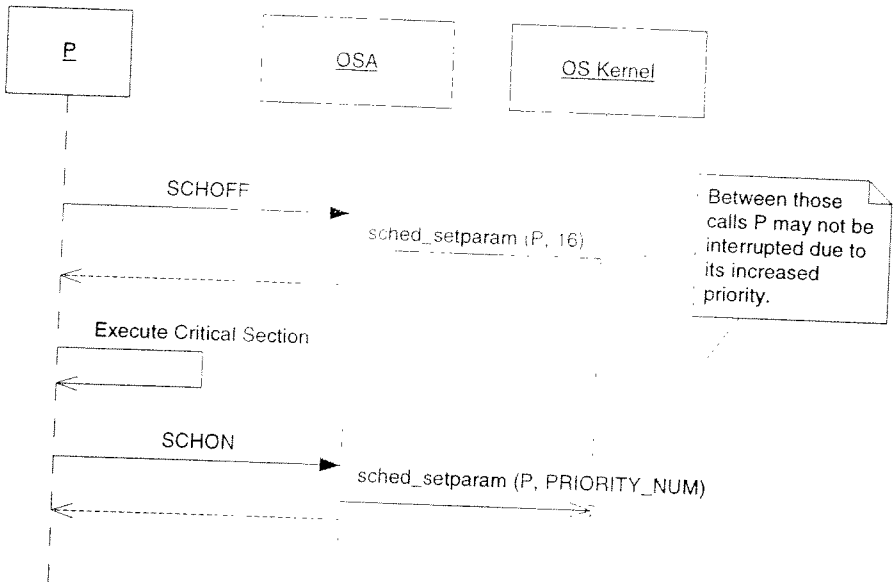


Figure 5-3: SCHOFF/SCHON SVCs

5.6 SVC Pause

As described in /Proz/ SVC `PAUSE` can be used to give up the processor in favor of other, even lower priority processes. If lower priority processes are in state 'ablaufbereit', VOCOS will delay the calling process for 50-100ms in favor of the lower priority processes.

The problem with this SVC is that for commercial RTOSs there is no possibility of peeking into the internal data structures of the scheduler to determine if lower-priority processes are pending. Therefore this behavior has to be resembled by other means.

POSIX `sched_yield()` cannot be used because it only moves a process to the end of the scheduler's queue for the given priority level of the process. Yielding a process with priority 10 would therefore not allow a process of priority 9 or less to gain the processor.

5.6.1 Simple Implementation

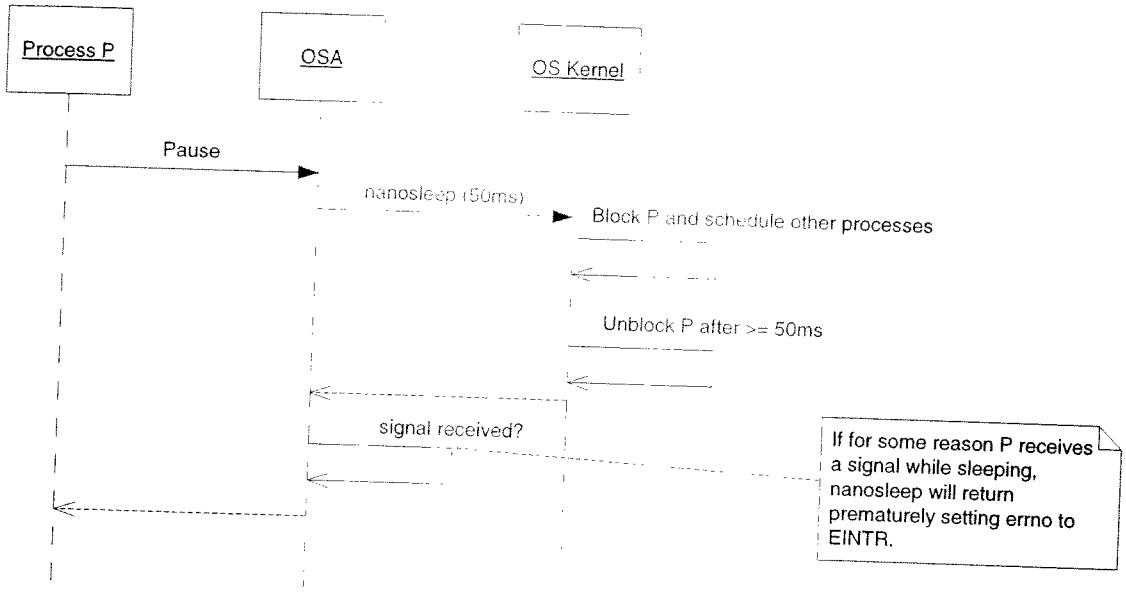


Figure 5-4: Simple Implementation of SVC PAUSE

Here a process issuing SVC `PAUSE` is delayed by the POSIX `nanosleep()` SVC for 50 ms. Within this time other lower priority processes can gain the processor. Upon return of the `nanosleep()` call it has to be checked whether the call returned because the requested sleep time elapsed or whether the paused process received a signal that has to be handled. Care has to be taken that the `PAUSE` SVC has to be reentrant since `PAUSE` might be issued while some processes are already paused.

Note that this simple implementation deviates from /Proz/ since it unconditionally pauses a process whereas /Proz/ states that if no other process (except audits and idle processes) is in state 'ablaufbereit', the calling process will be continued immediately.

5.6.2 Complex Implementation

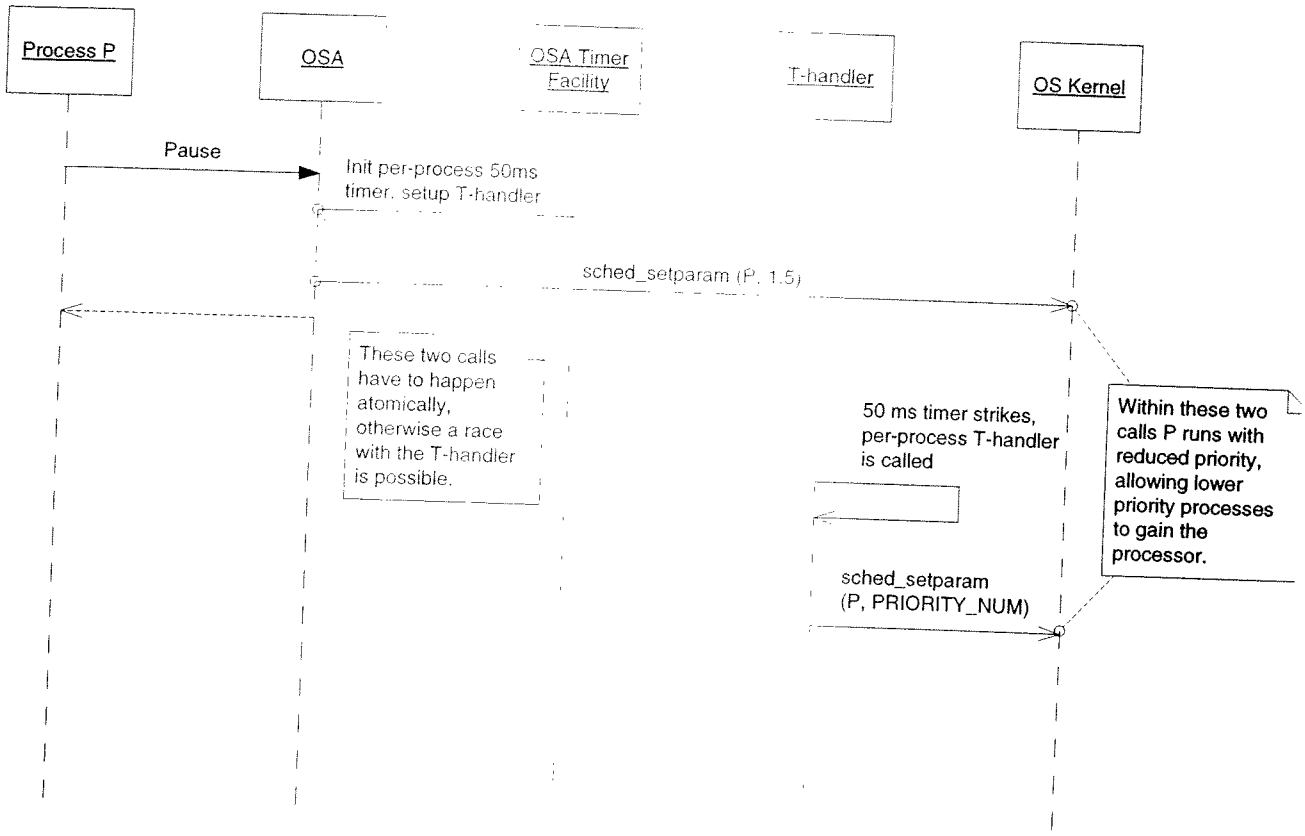


Figure 5-5 Complex Implementation of SVC PAUSE

With a more complex implementation the priority of the calling processes is decreased to a level that allows lower-level processes to gain the processor. In this way the calling process will continue to run if no other processes above level 1.5 are in state 'ablaufbereit'. After a delay of 50ms the priority of the calling process is set to its normal level as given in the PROCESS_INFO description. Note however, that while the calling process runs with decreased priority it can be interrupted at any time by other processes. This again deviates from the semantics of PAUSE stated in /Proz/.

6 Concurrency

6.1 Lock Sequences

As stated in /Concurrency/, a lock sequence is made up of all statements within a special DO-Block, where, at the head, the LOCK option together with a list of all to-be-locked data is specified.

Syntax:

```
DO <> LOCK X, Y <>; /* Begin of LOCK sequence */
   ...;       /* Code within the LOCK sequence */
OD;    /* End of LOCK sequence */
```

The purpose of a LOCK sequence is to protect data (*X* and *Y* in the above example) from concurrent access by several processes. In order to have processes acquire a lock for the shortest possible time and to avoid priority inversion, the priority of a VOCOS/CP process is raised to the OS level once it has acquired the lock. This ensures that the process cannot be interrupted anymore. In this context priority does not denote the priority associated with a process and used by the scheduler, but the priority of the interrupt of the underlying processor. Lifting a VOCOS/CP process from interrupt level 0 (application software) to interrupt level 2 means in fact that everything, including VOCOS itself, is switched off, to allow the process to execute without any interruption.

For the uni-processor case this protection mechanism is already sufficient, since it ensures that, once a process has acquired a LOCK (no matter what lock), it executes until it is out of the LOCK sequence again. Note that on a uni-processor system acquiring a lock will always be successful (otherwise the process asking for the lock would not be in state 'laufend'). Note also that on a uni-processor system the LOCK data is not relevant, since the question boils down to whether a process has monopolized the processor or not.

In case of multi-processor systems additional semaphores are required to keep processes on other processors from entering the LOCK sequence.

Mapping

As the new target hardware will be a uni-processor system, locking data is equivalent to raising the priority of the process acquiring the lock to the OSA level. Note that contrary to the current approach in VOCOS we denote here the priority used by the scheduler, and not about interrupt priorities. As long as OSA runs completely in user mode, this is sufficient (cf. also Section 1).

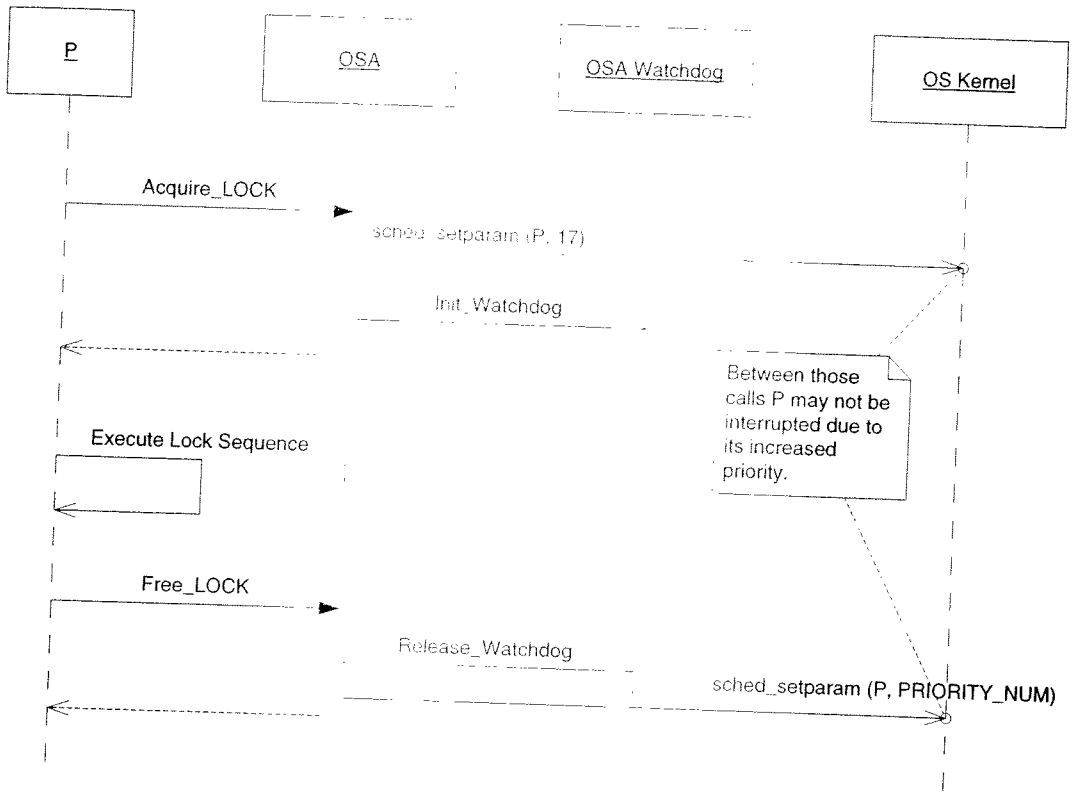


Figure 6-1: LOCK Sequences

`Acquire_LOCK` and `Free_LOCK` are calls implicitly generated by the backend. /Except/ contains already two SVCs called `LOK` and `UNLOK` that might be already used for this purpose. Within the `Acquire_LOCK` call the priority of the calling process is raised to OSA-level, thus preventing any other process (including OSA itself) from being scheduled. Once the process has executed the `LOCK` sequence it frees the `LOCK` by calling `Free_LOCK` which in turn lowers the priority of the process to the level specified as specified in `PROCESS_INFO`. Since it is checked by the compiler whether `LOCK` data is erroneously accessed outside a `LOCK` sequence, OSA does not have to provide any means to ensure this.

To avoid a hangup of the complete system the OSA watchdog is initialized. In the test phase a lock of more than 10ms would result in calling `RESTART`, at the customer `NSTART1` is called after 100ms. Those times will have to be adjusted due to different execution times on the new target hardware.

Note that a nesting of `Acquire_LOCK` within a `SCHON/SCHOFF` sequence cannot occur since the process acquiring the lock cannot be in state 'running' while another process is within `SCHON/SCHOFF` which also monopolizes the processor.

6.2 Regions

Assumptions: It is allowed to call a critical procedure within a critical procedure. It is assumed that the backend does not generate a `REGION_ENTRY` SVC with such nested calls of critical procedures.

Regions are very similar to the ADA95 language primitive of protected objects, where it is enforced that procedures belonging to a given `REGION`/protected object cannot be executed in parallel. Once a process is within the `REGION` all other processes attempting to enter the same region are blocked until the first process leaves it. By means of a `DELAY` statement a process

might block itself within a REGION. In such a case another process is allowed to enter the region. With CONTINUE a process within a region can unblock a process waiting at a DELAY statement. This takes only effect after the process issuing the CONTINUE SVC has left the region. Blocking of several processes at REGION entry or at a DELAY statement is done in FIFO order.

Mapping

The REGION concept is resembled solely by use of POSIX semaphores that are used to block/unblock processes

(1) on entry to the REGION.

(2) that DELAY themselves within the REGION.

For each REGION we use one so-called *entry semaphore* to guard against concurrent entry of processes. Entry semaphores are initialized with one upon startup. In the sequence diagrams given below this semaphore is named 'Rx'.

Furthermore we require one so-called *event semaphore* for each EVENT which is used to block processes delaying themselves on that particular EVENT. Event semaphores are set to zero upon startup. In the sequence diagrams given below one such semaphore named 'Ev' is used.

Since POSIX requires that processes blocking at a semaphore are unblocked according to their priorities, we cannot simply use a semaphore to have processes wait until they are allowed to enter the REGION or continue after a DELAY, because entrance to a REGION has to be granted in the order of arrival (FIFO). To overcome this problem the priority of a process is set to 16 before entry of a REGION. In this way all blocking processes have the same priority (the priorities are flattened) and are thus unblocked with the desired FIFO ordering. The first activity after returning from the `sem_wait()` operation is then to restore the original priority of the process.

Data structures are needed within OSA to maintain e.g. the correct ordering for the delivery of CONTINUE statements. Care has to be taken in the OSA design that access to this data structures is atomic (due to the fact that interleaving between different processes entering different regions is possible).

6.2.1 Simple REGION Entry and Exit

In order to map ongoing activities to the corresponding regions it is required that the backend generates the call REGION_ENTRY at the place a process attempts to enter a REGION. This call must also contain the particular REGION the process is competing for. OSA must at first obtain the pid as well as the current priority of the calling process. This is done by `getpid()` and *`sched_getparam()`*. The priority could also be obtained from the PROCESS_INFO description. However, by dynamically querying the priority we also catch the case where the PROCESS_INFO priority of the process has already been modified. The reason that OSA needs the priority of the calling process is that it has to modify this priority and needs to restore the original priority later on. To enforce FIFO-order unblocking OSA then sets the priority of the calling process to 16 and issues `sem_wait()` on the entry semaphore. Upon return of `sem_wait()` the REGION is dedicated to the calling process (*P*). OSA then restores the original priority and passes control back to the controlling process to have it execute the critical procedure.

Once the process reaches the end of the critical procedure, the backend-generated SVC REGION_EXIT is issued. Again we require that this call denotes the REGION that will be exited.

Since no CONTINUE has been issued in this example, OSA issues a sem_post() on the entry semaphore to free the REGION for other processes. Thereafter control is passed back to the calling process.

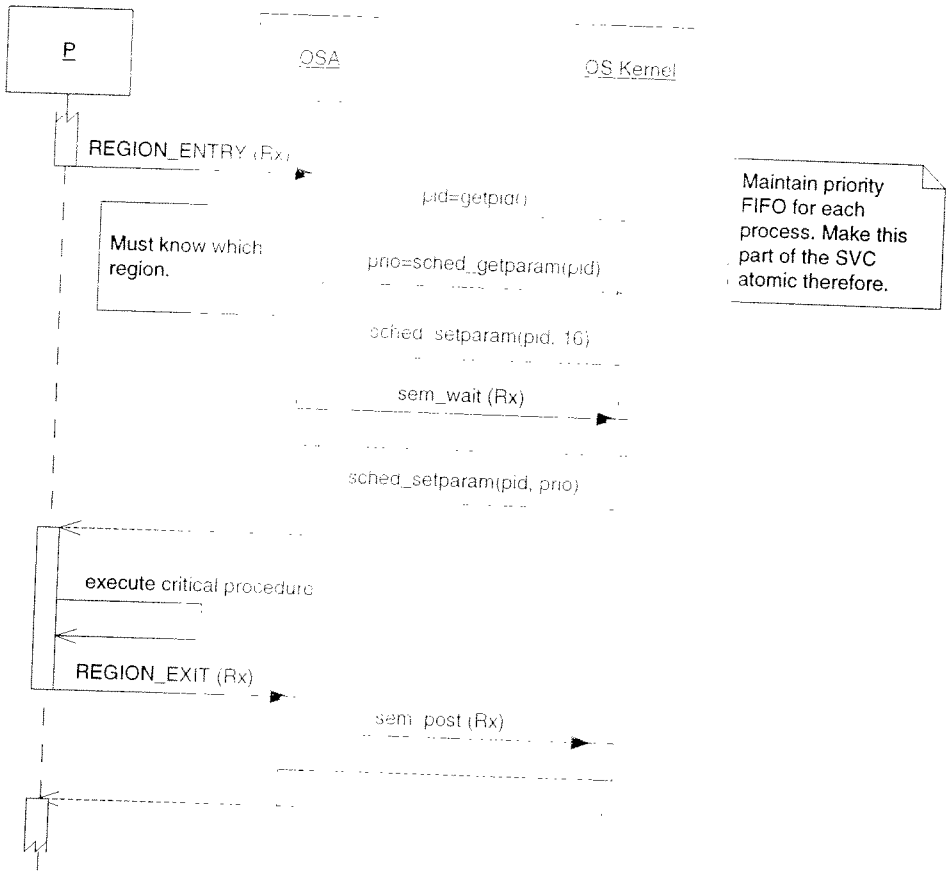


Figure 6-2: Free REGION

6.2.2 Contention at the REGION Entry

Multiple entry into the same REGION is prevented by the entry semaphore (*Rx*) as depicted in the following example.

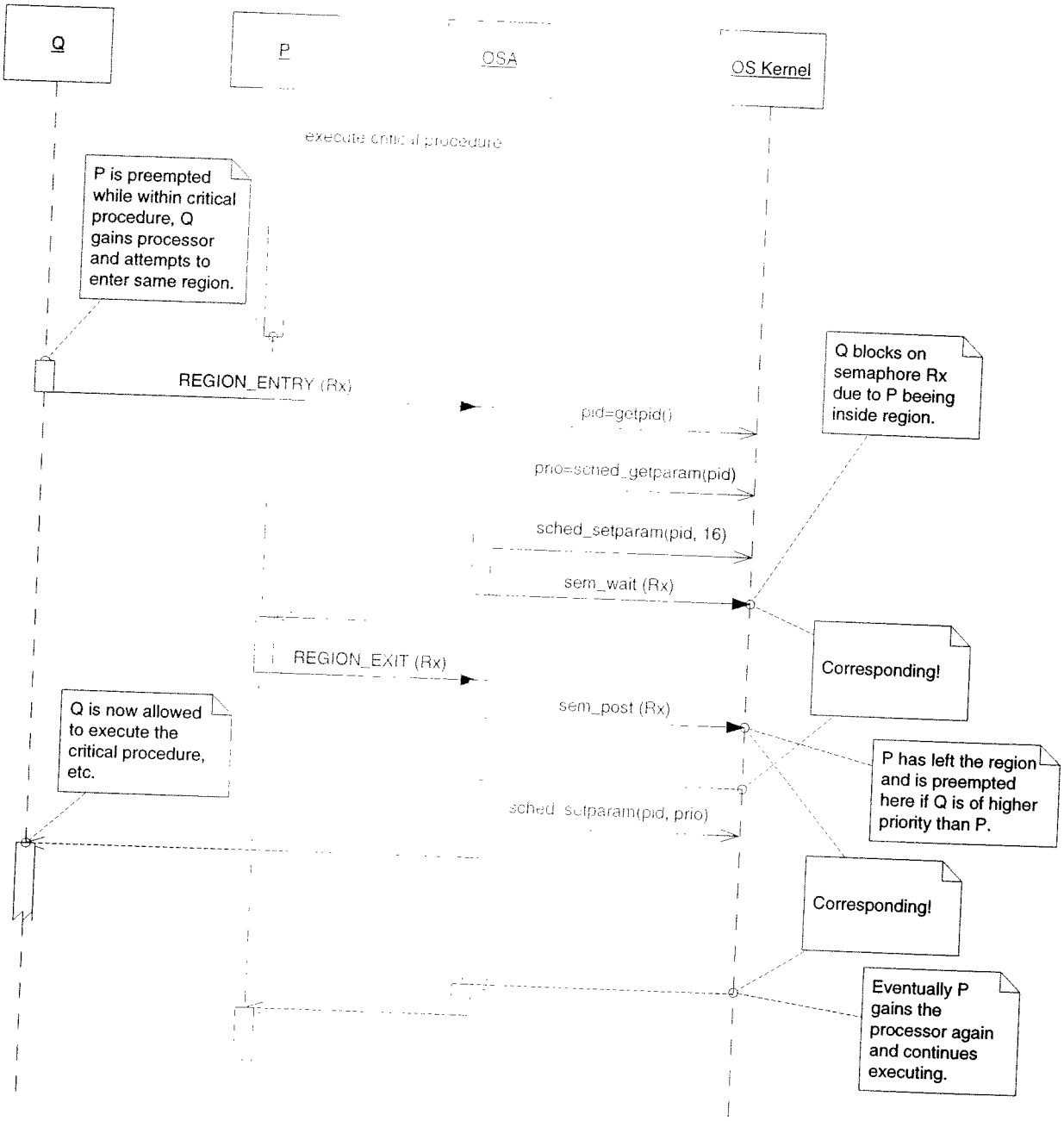


Figure 6-3: REGION with Contention.

6.2.3 Delay and Continue

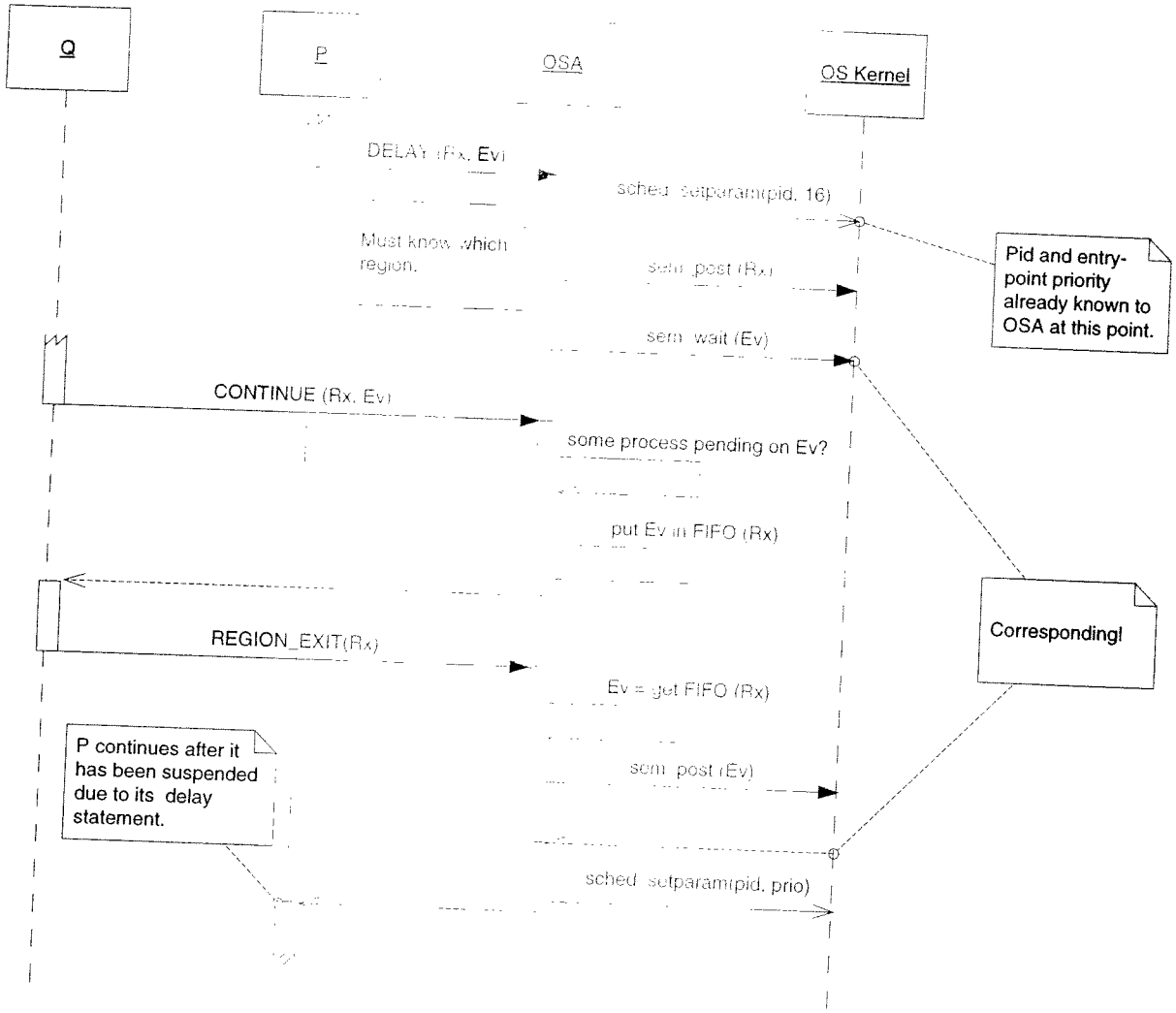


Figure 6-4: Delay and Continue.

Upon the beginning of this example process *P* is already within a `REGION` and issues SVC `DELAY` on event *Ev*. OSA sets the priority of *P* to 16 and calls `sem_post()` on the entry semaphore in order to free the `REGION` for another process and block *P* on the event semaphore corresponding to event *Ev*. Setting *P*'s priority to 16 has also the neat side effect that there is no schedule switch between `sem_post(Rx)` and `sem_wait(Ev)` which could otherwise lead to race conditions with the process doing the `CONTINUE`.

Eventually *Q* enters the `REGION` (left out in the drawing for space considerations) and issues SVC `CONTINUE` on event *Ev*. Since *Q* itself is still in the `REGION`, all that OSA has to do at that moment is to check whether a process is pending on *Ev*. If this is the case, it stores *Ev* in a FIFO queue that is used at the `REGION_EXIT` call to unblock delayed processes in the correct order. If no process was blocked on *Ev*, then OSA would simply discard this event (cf. /Concurrency/.

As soon as *Q* leaves the `REGION`, the corresponding `REGION_EXIT` call is issued. Due to the preceding `CONTINUE` OSA must now get the first entry of the FIFO event queue (there is only one in this example, namely *Ev*) and unblock a process on the corresponding event semaphore. Note that the entry semaphore is left unchanged. Thus the precedence of

DELAYed processes over processes at the REGION entry is enforced. Only after the last REGION_EXIT where all delayed processes have been continued and the FIFO event queue is empty will the entry semaphore be posted.

6.3 COMPARE_AND_SWAP, COMPARE_AND_SWAP2

These two commands are mapped on a hardware-feature of the Motorola processor MC68020/MC68040 that allows an atomic read-modify write operation.

Syntax:

```
COMPARE_AND_SWAP ( compare_value,
                   new_value,
                   destination,
                   success,
                   result_value);
```

Semantics:

```
IF compare_value = destination
THEN
        destination := new_value;
        success := TRUE;
        result_value := new_value;
ELSE
        success := FALSE;
        result_value := destination;
FI;
```

Without interruption, the following steps are executed:

Comparison of *compare_value* with the content of the memory content of *destination*.

If both are equal, destination is set to *new_value*, *success* is returned

If not, *false* is returned.

COMPARE_AND_SWAP2 works analogous for two *compare_values*, where the update only takes place if both *compare_values* are equal to their corresponding destination.

Since those primitives are based on a hardware-feature, they are very efficient and cause little blocking of other processes. It is therefore desirable to map them to a hardware feature on the target hardware, too.

Note however that to prevent race condition among different processes a hardware feature is not needed. If it turns out that a hardware-feature is not available, disabling interrupts or boosting the calling processes priority can also be used to resemble the behavior of those two features.

7 Interprocess Communication

VISION O.N.E. software architecture consists of autonomous entities called Service Provision Units (SPUs). A SPU is partitioned into a set of processes. Each process can be seen as a task of independent actions, that may execute concurrently with other tasks. Since co-operating tasks need to exchange data, to synchronize actions and to share resources, VOCOS provides facilities for interprocess communication, synchronization and mutual exclusion.

Communication is possible between processes independent of their location in the system. As stated in /IPC/, processes can be located

- on the same HW-platform running under the control of the same OS (intra-platform communication) or
- on different HW-platforms (inter-platform communication).

Since the new target hardware is not supposed to have multi-processor capabilities, this chapter deals only with intra-platform communication suitable for a desired uni-processor environment. We start with a short survey on interprocess communication facilities in VOCOS. Next, POSIX.1 and POSIX.1b interprocess mechanisms are discussed. Finally, we show *one* possible mapping from VOCOS supervisory calls (SVCs) to POSIX.1b function calls.

7.1 IPC in VOCOS

VOCOS supports asynchronous and synchronous communication. We use the term asynchronous communication when the sending process is not suspended until a receipt message arrives, whereas the term synchronous communication means, that the sender must wait, until it gets the result of the request.

7.1.1 Asynchronous Communication

Message buffers provide a flexible, general-purpose mechanism to implement asynchronous communication. Typically, a message buffer is associated with one particular process (i.e., a particular incarnation of the process, that starts receiving from that buffer). However, if a message buffer is declared with the special compiler directive `<>THREADED<>`, it is owned by a process type. Thus, messages of such buffers may be received by any process incarnation of the corresponding process type. Since process incarnations are out of scope of this feasibility study, we focus on normal buffers only.

A message buffer provides a uni-directional communication. If a bi-directional communication is necessary, the application has to setup two message buffers, one responsible for the communication between a sender and a receiver, the other one responsible for acknowledge messages back to the sender. The IPC related SVCs are listed below together with a brief description:

`CONNECT_SERVICE()` and
`GETUBI()` The setup of a communication between a sender and receiver is done by means of service management (i.e., the SVC `CONNECT_SERVICE`). Main result of the SVC is a reference to components of the given service instance including a so-called Unique Buffer Identifier (UBI), that has to be used for the subsequent communication.
In case that sender and receiver are located in the same SPU, the system call `GETUBI` may be used. As indicated in /IPC/, its envisaged use is to get the

UBI of a local buffer in order to include it in a subsequent message. Using that UBI, a receiver is able to return an acknowledge to the sender.

`SERVICE_CAST` and `CAST()` — The SVC `CAST` is used to transfer messages into message buffers. It is a non-blocking routine. Hence, if the corresponding message buffer is full, the sending process is not suspended, but is informed about the situation by a special error code. Messages sent by means of `CAST` must have a global header containing some control information provided by the operating system or the application. The application data part follows immediately after the header with length from 800 bytes up to 32 Kbytes.
The SVC `SERVICE_CAST` is a shortcut to connect to a service and to send a message. The sending functionality is the same as described above.

`SEND()` — Alternatively to the SVC `CAST`, the SVC `SEND` may be used to transfer a message into a message buffer located within the same SPU. In addition a priority may be defined determining the order, in which the corresponding message is read from the buffer by the receiver. Notice, that `SEND` is a blocking primitive! If a message buffer is already full, the message cannot be queued and the sending process is suspended.

`PMPOST()` — Similar to the SVC `SEND`, the SVC `PMPOST` transfers a message to the specified buffer within a SPU. Again priorities may be specified to alter the queuing of the message. However, this SVC is non-blocking and allows the transfer of 800 bytes user data.

`RECEIVE()` — Using the SVC `RECEIVE`, an application gets the oldest message with the highest priority contained in the specified buffer. `RECEIVE` is a blocking primitive. If the specified buffer is empty, the receiver is suspended, until a message eventually arrives.

`RECEIVE CASE` — By means of the SVC `RECEIVE CASE` the receiver may wait on several message buffers. If at least one buffer contains a message, the receiver gets the oldest message with the highest priority contained in this buffer. When all buffers are empty, the receiver is blocked, if no `ELSE`-part is specified. Otherwise the statements following the `ELSE`-part are immediately executed.

`PMTAKE()` — Unlike the SVC `RECEIVE`, the `PMTAKE` is a non-blocking SVC. Hence, if the buffer is empty the receiver continues and will not be suspended.

`GETDATA()` — VOCOS provides a facility to transfer data blocks of up to 32 Kbytes. Such a message consists of a trigger message (received by means of the SVC `RECEIVE`) and a supplementary user data block retrieved afterwards by means of the SVC `GETDATA`. The supplementary data are stored in the heap space of the application, which in turn has on one hand to ensure, that it has allocated enough heap space. On the other hand it is responsible to release the heap, after the user data has been processed.

7.1.2 Synchronous Communication

Synchronous communication allows processes to call procedures located on other machines. This mechanism is widely known under the synonym Remote Procedure Call (RPC). When a process on machine *A* calls a procedure on machine *B*, the calling process *A* is suspended and

execution of the called procedure takes place on *B*. Information is transported from the caller to the callee in the parameters and can return in the procedure result.

RPC achieves its transparency using so-called stub processes. When a procedure is called, that actually is a remote procedure, a client stub is invoked. Unlike a normal procedure, it does not put the parameters on a stack (or registers), but packs them into a message and sends them to a stub-process on the callee side. The server-stub unpacks the parameters from the message and then calls the server procedure in the usual way. The server-stub gets control back after the call has completed. It packs the result in a message and retransmits it to the client-stub, where the client-stub inspects it, unpacks the result and passes it to the original caller, which from now on may continue its work.

In VOCOS the definition of a Remote Procedure Call is done in the module description header in the `REMOTE_PROC_INFOS` section. Among other attributes the programmer can specify how many stub processes are generated by the compiler and started by VOCOS.

7.2 IPC in POSIX.1 and POSIX.1b

Generally spoken, POSIX.1 and POSIX.1b supplies message queues – like message buffers a highly flexible, general-purpose mechanism to implement asynchronous communication, discussed in more detail in the following sections. However, synchronous communication by means of RPC are neither defined by POSIX.1 nor by POSIX.1b.

7.2.1 Asynchronous Communication

POSIX.1 provides a possibility for synchronous communication by means of pipes and their named counter-parts FIFOs.

`pipe()` — A message buffer usually has two types of users. A source (client), that posts messages, and a sink (server), that consumes messages. Hence, a pipe opened using the system call `pipe()` returns two file descriptors. One is used as the writing end of the pipe, the other one is used as the reading end.

`write()` and `read()` — Communication is done like writing and reading of files, i.e., the standard UNIX system calls `write()` and `read()` have to be used to put messages into the pipe as well as to read messages out of the pipe. By default reading data out of a pipe will block (i.e. suspend the process), until data appears in the pipe. To avoid this blocking, the standard UNIX system call `fcntl()` may be used to set the a certain flag, called `O_NONBLOCK`, for the file descriptors.

Notice, that a pipe is just a uni-directional mean for communication. Hence, one process may put messages into the pipe, while the other one is reading out of pipe. If both processes would write messages into the pipe at the same time, messages get intermingled.

A pipe requires to be set up in a parent process, where a child process is started afterwards. For a lot of distributed applications this hierarchy is cumbersome. That's the reason, why POSIX.1 also defined FIFOs:

`mkfifo()` and `open()` — A FIFO is simply a pipe, that has a name in the file system. Once created by means of the system call `mkfifo()`, you may open it using the standard

UNIX system call `open()`, where you can specify its name and if you want to use it for writing (`O_WRONLY`) or reading (`O_RDONLY`). A FIFO cannot be opened for both writing and reading!

There are a couple of limitations, that apply to both pipes and FIFOS:

- Pipes and FIFOs are strictly first-in-first-out communication mechanisms. Nevertheless, it is not possible to post messages with higher priority in order to let it get in front of lower priorities ones, that have been already queued. Such mechanisms can only be built at the application level using separate pipes or FIFOs.
- There is no possibility to control or even know the amount of buffer space available for a given pipe or FIFO.
- Pipes and FIFOs transport nothing more than a stream of bytes. Hence, on the source side no structures can be put into a pipe or FIFO, unless they were converted to a simple byte-stream. On the sink side such a byte-stream has to be rebuilt to the structure the client wants to access.
- Since two file descriptors are necessary for each pipe (remember, one for each end) and one file descriptor is needed for each FIFO, some UNIX operating systems may get in trouble with applications with hundreds of pipes, FIFOs and normally files opened.

Unlike pipes and FIFOs, message queues in POSIX.1b are not strictly first-in-first-out oriented. Each message has a priority (from 0 to `MQ_PRIO_MAX`, usually 32 defined in the header file <`limits.h`>). Higher-priority messages get in front of the queue. At a single priority level, messages are still put in FIFO order.

Since `open()`, `write()` and `read()` are standard UNIX system calls (usually used for opening, writing or reading normal files), POSIX.1b decided to avoid the overhead associated, when calling them, and defined a set of similar ones (yet being more efficient).

`mq_open()` The `mq_open()` function call establishes a connection between a process and a message queue. The arguments include the name of the message queue, that has to be conform to the construction rules of regular path-names, and flags, where you can specify, if the message queue should be created, opened for writing, reading or both (i.e., for simultaneously sending and receiving of messages). In addition the parameter `mq_attr` can be used to define some attributes of the message queue, like

- `mq_maxmsg` to set the maximum number of messages in the queue,
- `mq_msgsize` to set the maximum size of a single message,
- `mq_flags` to modify the behavior of the message queue (blocking or non-blocking by means of `O_NONBLOCK`),
- `mq_curmsg` to get the number of messages currently in the queue.

`mq_send()` Sending a message looks like writing to a file, except for one parameter, that allows to set a message priority. If the specified message queue is not full, the message is inserted into the queue indicated by the priority parameter. If the message queue is full and `O_NONBLOCK` has not been set, the function call blocks until space becomes available and the calling process is suspended.

`mq_receive()` This call removes the message at the head of the corresponding message queue and places it in a buffer, which has to be provided by the application as function argument. The buffer to be passed has to be at least as large as the maximum message size for the message queue. If the specified message queue is empty and `O_NONBLOCK` has not been set, the function call blocks,

until a new message arrives. If more than one process is waiting at an empty queue, then the process with the highest priority, that has been waiting longest is selected to receive the message and to continue its work.

`mq_close()` and `mq_unlink()` By means of these function calls a message queue may be closed and unlinked. Latter means, that such a message queue becomes inaccessible for any other process. If messages still have not been picked up, they get lost.

`mq_getattr()` and `mq_setattr()` These functions should be used to read all attributes (i.e., get the current status of a message queue, for instance to check, whether it contains some "pending" messages) or to set the `mq_flag` attribute associated with an open message queue.

`mq_notify()` Last but not least, it should be mentioned, that the function `mq_notify()` provides a mechanism for sinks to receive notice, that messages arrived in a message queue. This nice feature implemented by means of a signal is a real a asynchronous mechanism compared to the synchronously waiting due to the function call `mq_receive()`!

While message queues in POSIX.1b are a better mean for inter-process communication (at least they give you the facility to have a little bit control over the internal structure), they still have the limitation, that only byte-streams may be transferred. When transferring bulks of data, this must of course be avoided (we will elaborate on that drawback at the end of the next section).

7.2.2 Asynchronous Communication

It has already been mentioned, that neither POSIX.1 nor POSIX.1b define mechanisms for RPC. Nevertheless, most UNIX systems (including SUN Solaris) support RPC by means of special library routines. These routines allow C language programs to make procedure calls on other machines across a network. RPC works as described above: first, the client sends a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service and then sends back a reply.

7.3 Mapping VOCOS-IPC to POSIX.1(b)-IPC

This section surveys a possible mapping of interprocess communication facilities of VOCOS to interprocess communication mechanisms defined by POSIX.1(b).

7.3.1 Mapping of asynchronous communication

VOCOS message buffers are bound to processes, where one process may own more than one buffer. This is no problem for POSIX.1b message queues, that are accessed via names, thus facilitating a common communication mechanism. Those processes, who know the name of the message queue, simply have access to it. Hence, the VOCOS process, that declares the message buffer, should be the one, who cares for the proper creation and initialization of the corresponding message queue. We suggest to set up a message queue with the attribute `mq_flags` set to `O_NONBLOCK` ensuring, that a subsequent sending function call will not block. The attributes `mq_maxmsg` and `mq_msgsize` should be set according to the declaration of the message mode and buffer mode.

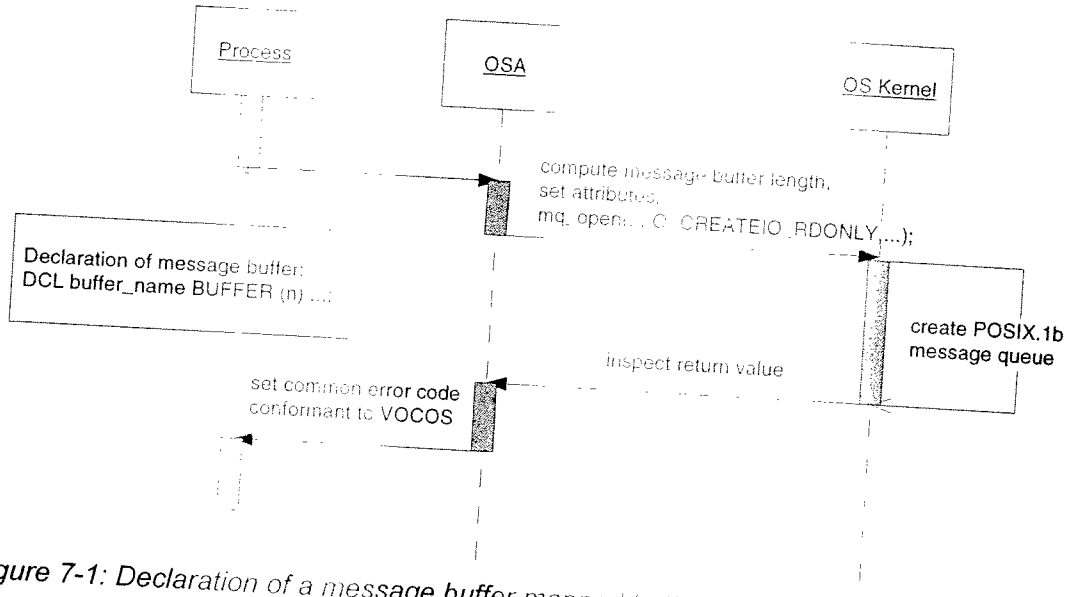


Figure 7-1: Declaration of a message buffer mapped to the creation of a message queue

Whenever the UBI of the message buffer is retrieved by means of the SVC SERVICE_CONNECT or GETUBI, the OSA has to map it to the name of the corresponding message queue. This happens transparently to the application. Notice, that the message queue now has to be opened without the O_CREATE flag.

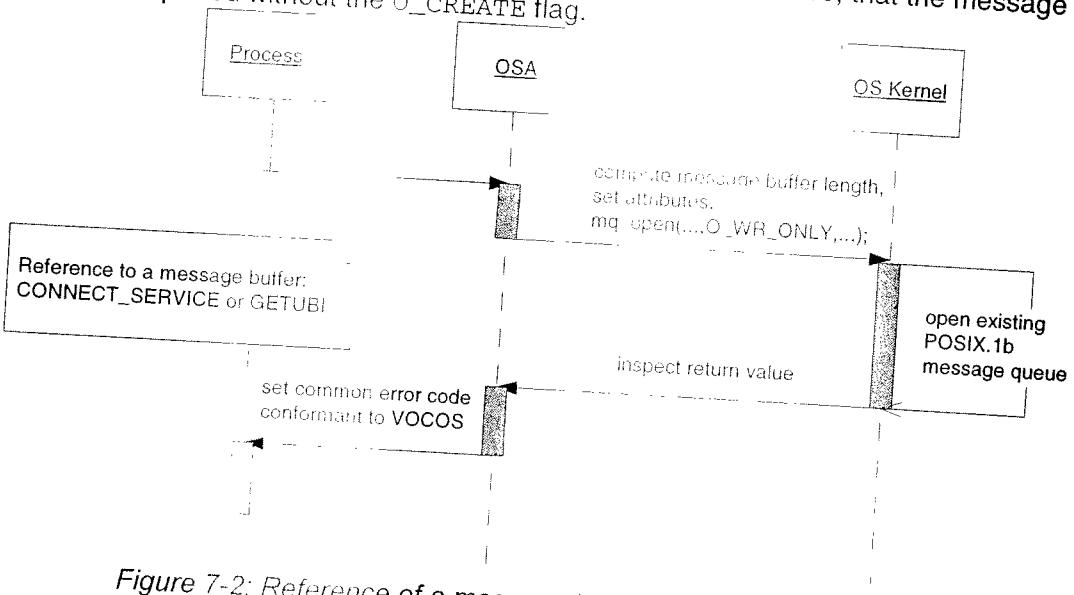


Figure 7-2: Reference of a message buffer mapped to mq_open()

We offer to map the SVCs CAST, SEND and PMPOST to the function call mq_send(). However, since mq_send() allows to transfer byte-streams only, it is necessary to generate wrapping functions, that convert the corresponding structures to streams and fill the message header, if necessary. In case of SEND and PMPOST a priority may be issued. Since the SVCs CAST and PMPOST are non-blocking SVCs, the message queue can be used as it was initialized (with O_NONBLOCK feature). Thus, the semantics is as follows: if there is room in the message queue for another message, the message will be added without problems. Otherwise mq_send() returns -1 and the external variable errno, which is set, whenever an error occurs, reads EAGAIN, which in turn indicates, that the specified message queue is already full.

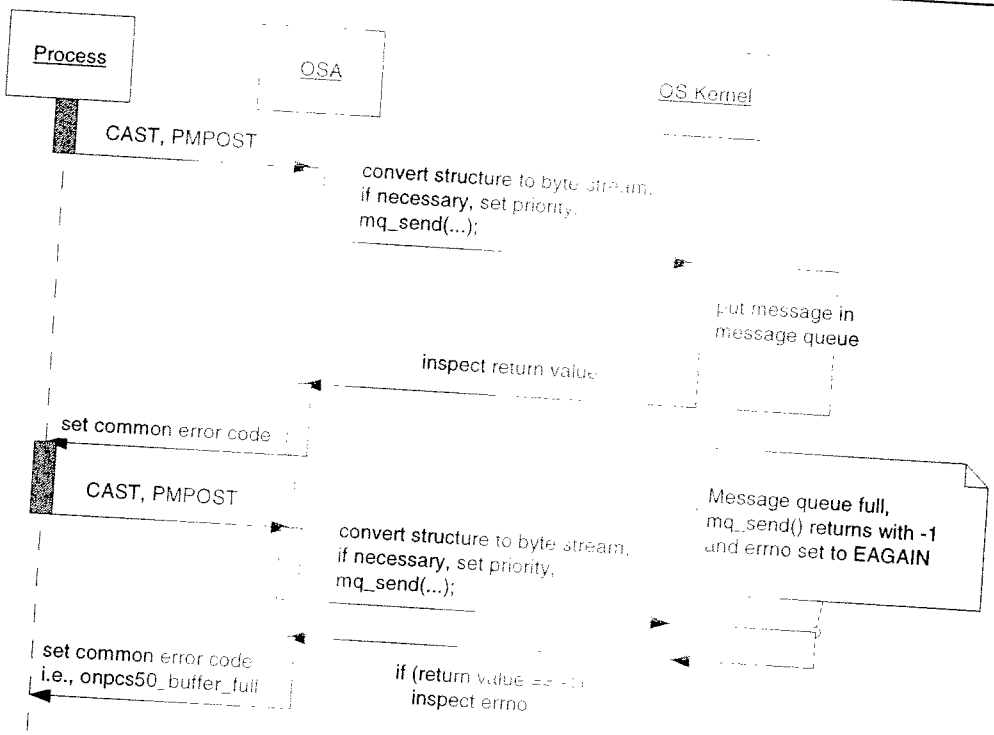


Figure 7-3: Mapping CAST and PMPOST to mq_send()

However, the situation is different for the SVC `SEND`, which should block, if the message queue is already full. Fortunately, POSIX.1b provides the function call `mq_setattr()` to alter the behavior of a specific message queue, thus enabling us to clear the `O_NONBLOCK` flag. Now the semantics changed as follows: if there is room in the message queue for another message, the message will be added without problems. Otherwise the calling process is suspended. If more than one process is waiting to send, when space becomes available in the message queue, the process of the highest priority, which has been waiting the longest, is unblocked to send its message.

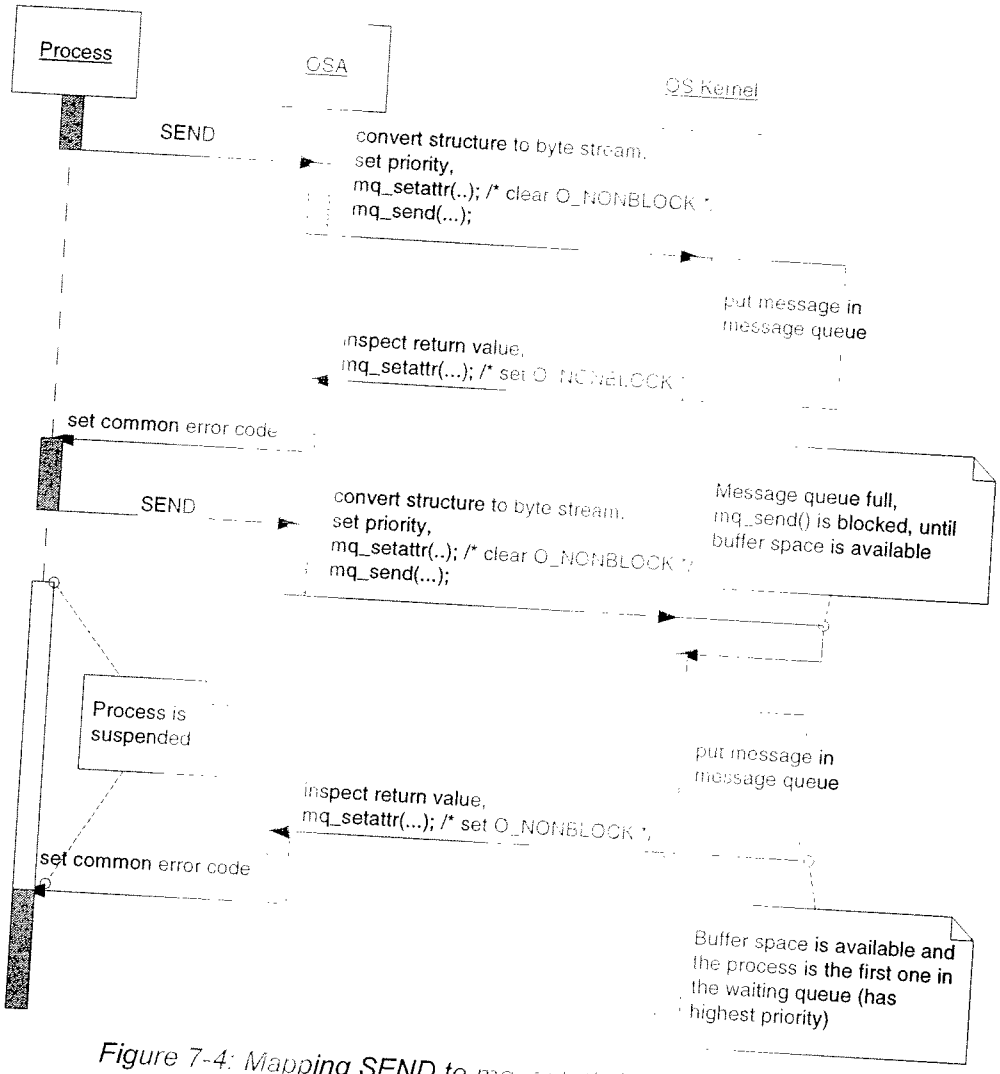


Figure 7-4: Mapping SEND to mq_setattr() and mq_send()

So far so good. Now let's have a glance at the SVCs to receive a message out of a message buffer. Since the corresponding SVC `RECEIVE` should block, the behavior of our message queue again has to be altered by means of `mq_setattr()` (alternatively, the message queue can be created without the `O_NONBLOCK` feature). The function `mq_receive()` gets the oldest of the highest priority message from the specified message queue. If the specified message queue is empty `mq_receive()` will block, waiting until a new message eventually is queued. If a message arrives at an empty queue and more than one process is waiting, then the process of highest priority, that has been waiting the longest, is selected to receive the message.

Once more, it has to be stated, that POSIX.1b message queues transfer byte-streams only. Hence, we need an extra wrapper function to reconstruct the desired structure.

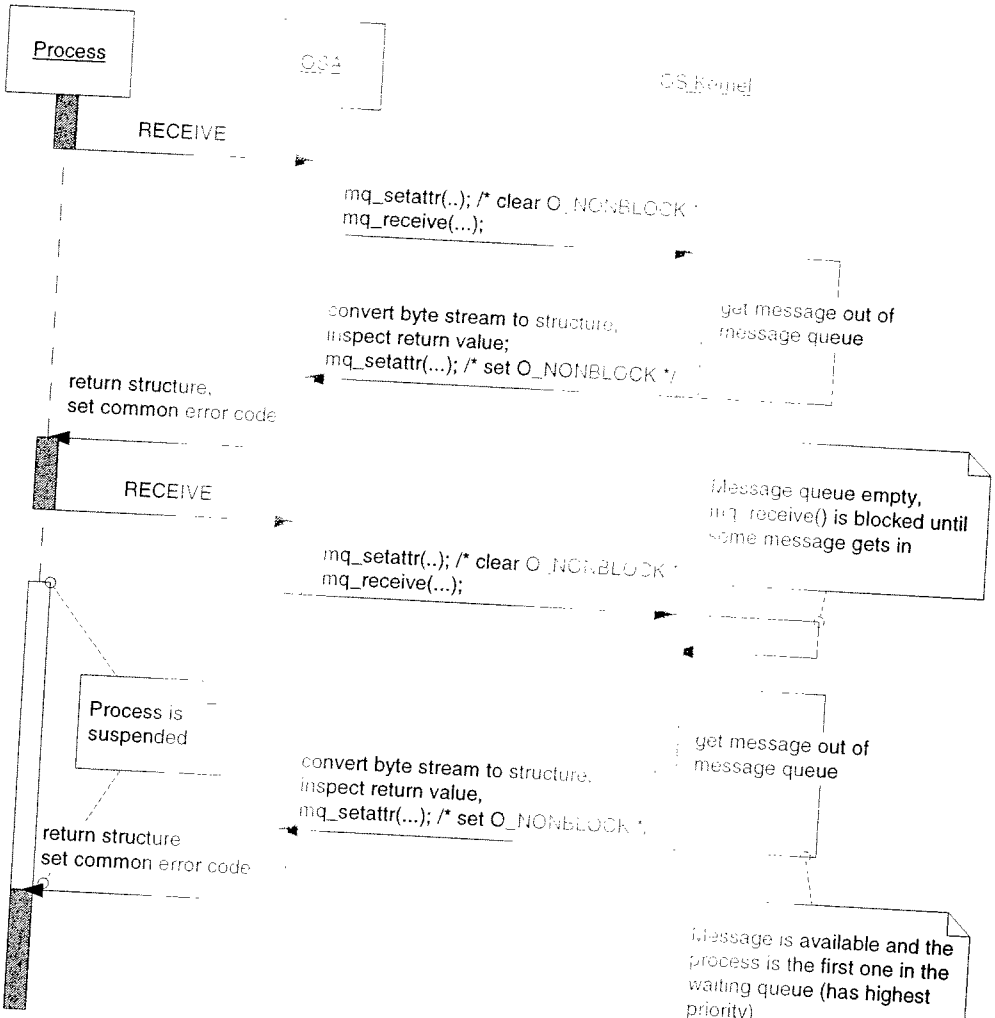


Figure 7-5: Mapping RECEIVE to mq_receive()

The SVC `RECEIVE CASE` demands to get a message from one message buffer out of a sequence of message buffers. Since `mq_receive()` permits to receive messages out of one buffer only, we need some workaround. What can be done, is to use `select()`, unfortunately, not part of POSIX.1 or POSIX.1b. In fact, it is a standard UNIX function!

By means of the function `select()` we may check, which of the specified file descriptors are ready for reading or have an error condition pending. If the specified condition is false for all of the specified file descriptors, `select()` normally blocks, up to a user-defined timeout interval, or until the specified condition becomes true for at least one of the specified file descriptors. If the timeout argument points to an object, whose members are zero, `select()` does not block (which is extremely useful for the SVC `RECEICE CASE` with an `WAIT`-part).

The `select()` function supports regular files, FIFOs and pipes and other character based devices. Unfortunately, the behavior of `select()` on file descriptors, that refer to other types of files is unspecified and thus has to be tested for POSIX.1b message queues.

Suppose select will work for message queues, the SVC `RECEIVE CASE` could be mapped as follows:

1. The file descriptor masks, a necessary argument, for the `select()` function have to be initialized with the macros `FD_CLR()` and `FD_SET()`.
2. Afterwards `select()` is called.
3. If some message buffers contain a message, the messages are read out of it (using a `switch()` statement list, for instance) and converted into the corresponding structures.
4. Finally, the structures have to be returned to the calling process.

If in addition the SVC `RECEIVE CASE` contains an `ELSE`-part, the timeout has to be set to zero ensuring, that the process may immediately continue even if the messages queues are empty. Notice, although this looks quite simple in theory, the appropriate automatic mapping done by a future compiler back-end seems to be hard to construct.

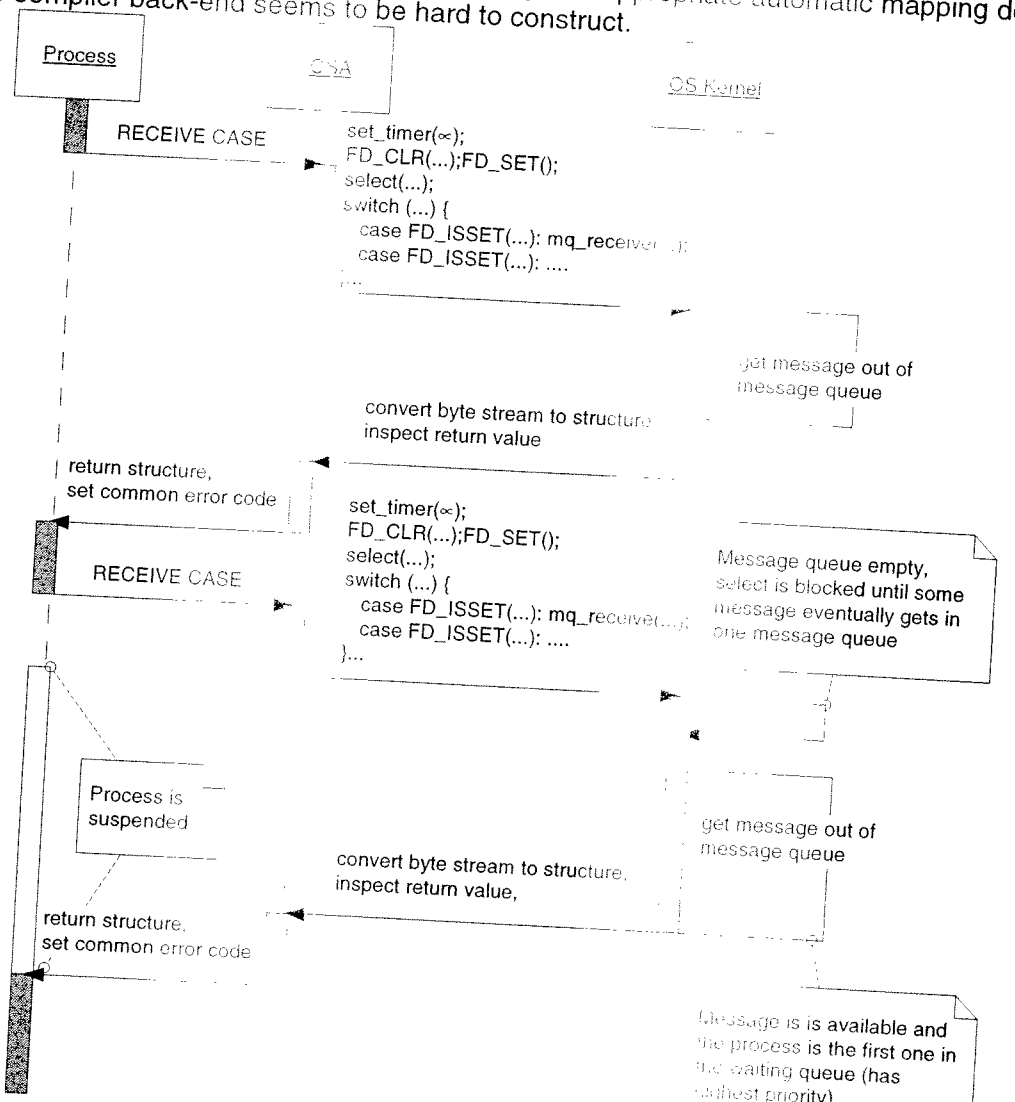


Figure 7-6: An approach to map SVC RECEIVE CASE to select() and mq_receive()

The SVC `PMTAKE` may be mapped to `mq_receive()`, too. Since this SVC is non-blocking, we can omit to clear the `O_NONBLOCK` flag. Hence, if there is a message stored in the message queue, it is picked up and converted to the defined data structure. Otherwise `mq_receive ()` returns -1 and the external variable `errno`, which is set, whenever an error occurs, reads `EAGAIN`, which indicates, that the specified message queue is empty.

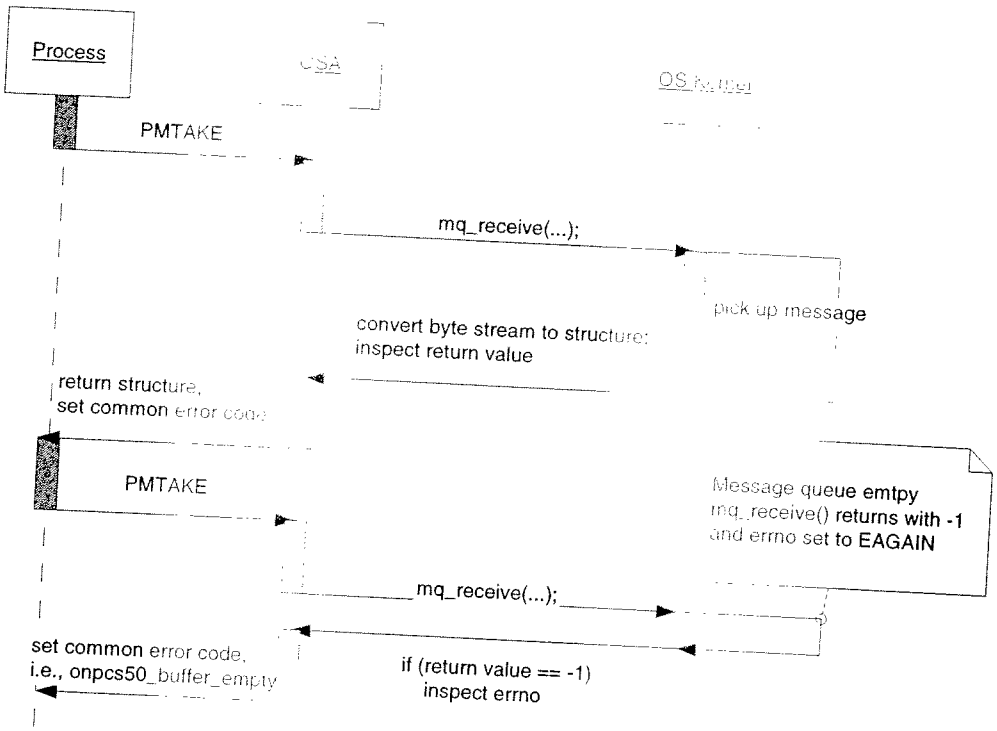


Figure 7-7: Mapping PMTAKE to mq_receive()

Finally, we have to discuss the SVC `GETDATA()` that is called, whenever a user receives a trigger message, that describes a supplementary data block. To map this functionality further actions at the sender as well as at the receiver side are necessary (which we frankly omitted above for the sake of simplicity).

Whenever a SVC `CAST` is called including a VOCOS category III or IV message (big and fat messages with lengths up to 32 Kbytes), the sender has to `mmap()` the supplementary data into shared memory area (opened as discussed earlier by means of `shm_open()`). At the receiver side the wrapper function called after the `mq_receive()`, has to examine the message header. Once it discovered a header of category III or IV messages, it has to `mmap()` the supplementary data block into user space. Afterwards the message structure is rebuilt and the receiver may call `GETDATA`. Again, what seems easy in theory, could be very hard to apply in practice!

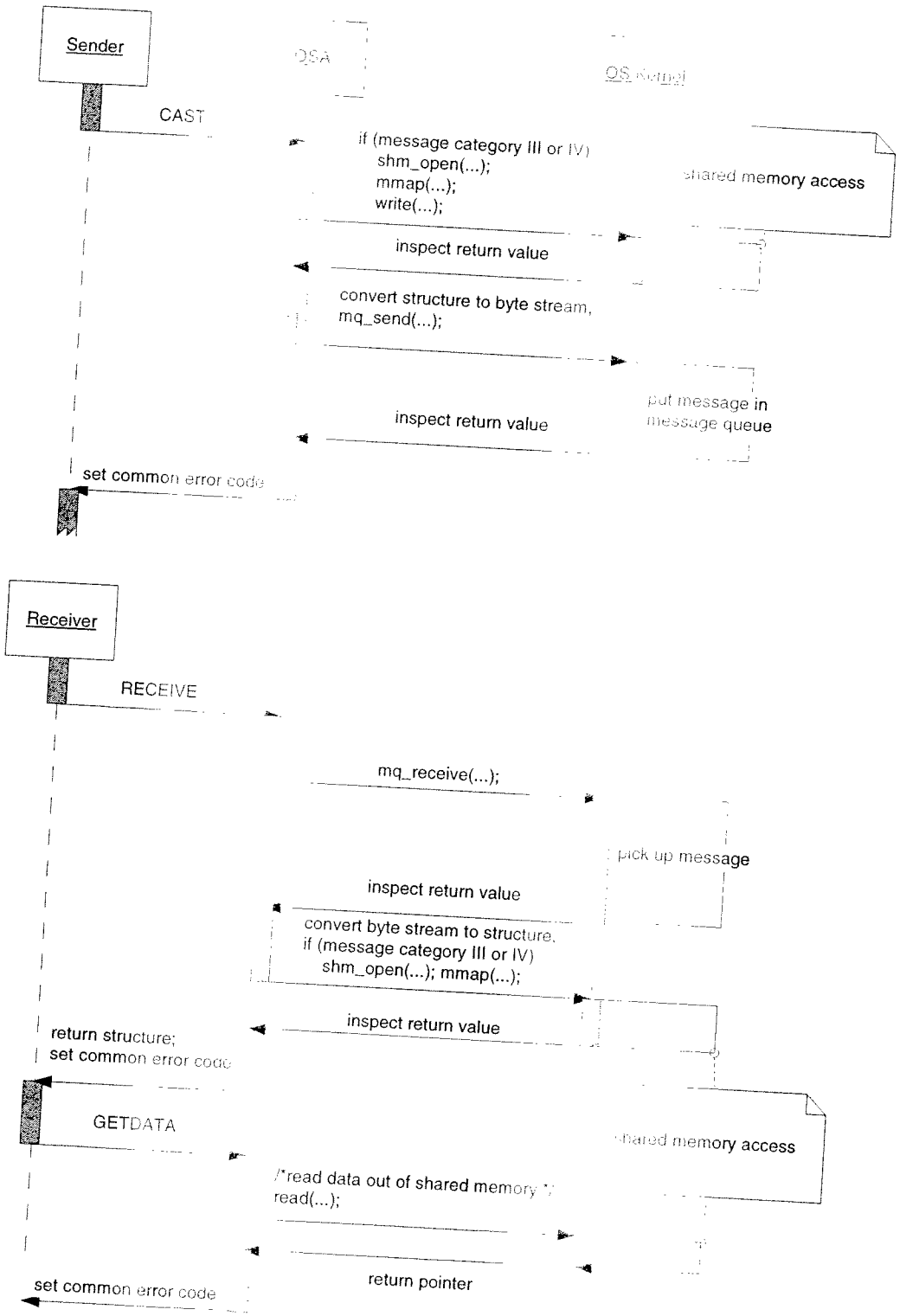


Figure 7-8: Mapping SVC GETDATA to mmap()

7.3.2 Mapping of synchronous communication

Since the future target system will be a uni-processor and hence all processes may have access to a common (virtual) address space, we propose to optimize the performance of RPCs by using normal procedures, that can be bundled into a shared library.

Notice, that the caller of the former RPC sees no difference. However, the procedure has to be re-entrant and must access data in a secure manner (i.e., with mutual exclusion). As already stated, in VOCOS the definition of a RPCs takes place in the module description header in the `REMOTE_PROC_INFOS` section. Among other attributes the programmer can specify how often a RPC may be invoked concurrently. Therefore, before entering the normal procedure, the OSA has to check, if the entry is allowed. This can simply be done using a semaphore and the associated POSIX.1b function calls `sem_wait()` and `sem_post()`.

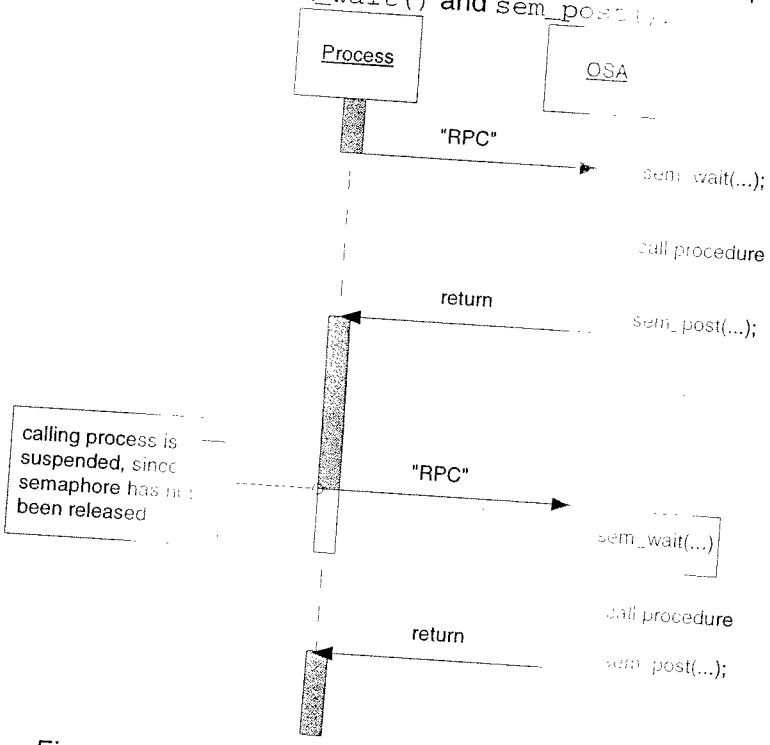


Figure 7-9: Mapping RPCs to normal procedure calls

8 Time Management

Due to lack of time, we could only briefly survey the different kinds of timer facilities defined in VOCOS, i.e.,

- relative timer jobs, which become due after a specified period of time,
- absolute timer jobs, that become due at a certain point in time and
- time discontinuity jobs, that keep user software informed about discontinuities in time of day.

Since time granularity in VOCOS is around 50ms, we dare to say, that the POSIX.1b timer management facilities will probably meet the desired functionality, for instance,

`timer_create()` The `timer_create()` function creates a timer using the specified clock, clock_id, as the timing base and returns, in a further referenced location a timer ID used to identify the timer in timer requests. This timer ID will be unique within the calling process until the timer is deleted. Another argument, if non-null, points to a `sigevent` structure. This structure, allocated by the application, defines the asynchronous notification that will occur when the timer expires.

`timer_settime()`, `timer_gettime`, `timer_delete`, ...

9 Recovery

/ErrorH/ specifies four so-called *action groups* which determine the actions that have to be carried out during different levels of system recovery ('Anlaufstufen'). They can be summarized as follows:

Phase	Action Group	Measurements
NSTART0	1_0	(Re)startof all PROCESSes that match the following process-properties (cf. /Proz/, Section 3.3): CYCLIC=Yes (means that this process belongs to a STARTGROUP) SWITCHING_GROUP = NO Remarks: restart of all NVT processes.
NSTART1	1_1	(Re)start of all processes with property CYCLIC=Yes, initialization of the operating system
NSTART2	1_2	load semi-permanent data & NSTART1
NSTART3	1_3	load code and transient data & NSTART2

Table 9-1: System Recovery Phases

To emulate the functionality given in Table 9-1 the following mechanisms have to be provided:

(1) (Re)start of a CHILL PROCESS

(2) OS Initialization

(3) Write protection of semi-permanent and permanent data

(4) Loading of transient and semi-permanent data

(5) Loading of code

A separate section is dedicated to the explanation of each of the mechanisms mentioned above.

9.1 (Re)start of a CHILL PROCESS

Depending on how we map the CHILL PROCESS primitive we have to restart a

- POSIX process
- POSIX thread
- proprietary thread of control (e.g. task)

Note, that although it is possible for POSIX processes and threads to establish means to cleanup before termination, undoing of acquired resources (e.g. REGIONs) can only be done by OSA itself, since the associated resources (e.g. entry- and event semaphores) are transparent to the user PROCESSes.

9.1.1 POSIX Process

Restarting of a POSIX process can be achieved by sending the process the unmaskable signal SIGKILL which inevitably terminates the process.

Afterwards a process of this type can be forked again. Note, that by that means there is also a loading of code/data of the process involved, which is not specified for e.g. NSTART0.

Sending a process an unmaskable signal means that it cannot accomplish any cleanup functions (e.g. freeing acquired resources like semaphores), but cf. note above!

By using a maskable signal, the process can achieve cleanup-functions in its signal handler. However, it must then be mandatory to call the exit function as the last instruction of the signal handler. Otherwise processes would be able to circumvent a terminating signal.

9.1.2 POSIX Threads

The following termination-related interface is provided for POSIX threads (cf. /PosixThread/):

- `pthread_cancel()`: Cancels thread execution.
- `pthread_setcancelst...()`: Sets the cancellation state of a thread (disabled: cancellation is kept pending until cancel-state is enabled again).
- `pthread_setcanceltyp...()`: Sets the cancellation type of a thread (deferred/asynchronous).
- `pthread_testcancel()`: Creates a cancellation point in the calling thread.
- `pthread_cleanup_push()`: Pushes a cleanup handler routine.
- `pthread_cleanup_pop...`: Pops a cleanup handler routine.

Threads can be cancelled synchronously (only when they reach a cancellation point, such a thread-related call), or asynchronously (cancelled immediately). Stacked cleanup handlers may be used to cleanup a thread's acquired resources at the moment of cancellation. After thread cancellation the thread may be started again using `pthread_cr...()`.

9.1.3 Proprietary Threads of Control

The following interface is supported by VxWorks to support deletion and start of tasks (note, that this interface is not POSIX-compliant):

- `taskDelete()`: delete a task
- `taskSave()`: (temporarily) protect a task against deletion (e.g. before entry to critical region)
- `taskUnsave()`: allow deletion of task
- `taskSpawn()`: creates a new task

9.2 OS Initialization

Contrary to VOCOS the meaning of OS-initialization is limited to the OSA layer itself, since commercial RTOSes do not allow the user to trigger internal OS initialization. A shutdown and restart of the target RTOS would on the other hand imply a reload of all process code, which violates the semantics of the NSTART1 phase (besides imposing a severe penalty time-wise).

9.3 Write Protection of Semi-Permanent and Permanent Data

In order to support semi-permanent and permanent data it is necessary to protect memory areas from write-access (cf. Section 0.5). In case of semi-permanent data only the UPTSPF function (cf. /Update/) is allowed to write the protected data, in case of permanent data no write access is allowed at all. In the following sections two possible approaches to memory protection are presented.

9.3.1 Protection According to POSIX

The POSIX.1b memory protection mechanism[8] distinguishes the following attributes for memory pages:

`PROT_READ`: page can be read

`PROT_WRITE`: page can be written

`PROT_EXEC`: page can be executed

`PROT_NONE`: page cannot be accessed

Attributes can be or-ed together as long as this is supported by the underlying hardware (e.g. some machines require PROT_READ in order to allow execution of code). Attributes can be set either through the POSIX.1b SVC `mmap()` (cf. Section 9.4.2) or through the POSIX.1b `mprotect()` SVC.

`mprotect(const void *addr, size_t length, int prot)`: Allows to set the memory protections `prot` of the mapped area starting at `addr` and proceeding for the given `length`.

The protection modification is done in units of PAGESIZE, to include the area specified. On some systems it may be required to pass in an `addr` that is a multiple of PAGESIZE.

Note that POSIX.1b provides protection of memory pages only if they have been previously mapped using mmap()! In this way memory protection is restricted to shared memory, and further devices where mapping makes sense (cf. /Posix1003.1b/), such as regular files.

`ftruncate (int fildes, off_T length)` can be used to (re)size the underlying representation of the mapped area specified via `fildes` to the specified size given in `length`. If `fildes` was previously smaller than `length`, zeroes are inserted from the EOF onwards. If `fildes` was previously longer, bytes past `length` will no longer be accessible. Note that this does not automatically resize the mapped area also. Note, that `mmap()` cannot be used to implicitly resize the underlying representation of the mapped area. Any reference to addresses beyond the end of the object will result in the delivery of a SIGBUS signal.Violation of a memory protection attribute results in POSIX signal SIGSEGV delivered to the causing process. Treatment of this signal may not be trivial as is depicted in Figure 9-1. Process `P` erroneously sets the pointer `ptr` to address `0x00000000`[9]. Within the next instruction it attempts to assign a value to the memory location pointed to by `ptr`. As `P` has installed a signal handler, the OS calls this signal handler after the trap due to the erroneous statement. Note, that this trap occurred prior to the execution of the erroneous statement! After attempting a counter-measure due to SIGSEGV in its signal handler and passing control back to the OS, `P` eventually gets scheduled again. The program counter of `P` is still at the offending instruction, and another attempt is made to execute the instruction, resulting in the same trap, and so on. It is yet to be determined how this problem will be treated, but a solution will propably be based on stack modifications (as done by `setjmp()` / `longjmp()`). If a POSIX process model is used, then it

[8] Not implemented in VxWorks.

[9] Or any other address it must not write to, such as an address within a write-protected area.

might be easier since the offending process can be terminated in which case the POSIX.1 SIGCHLD signal is received by the parent process (OSA, in our case). If threads can be terminated out of a signal handler is yet TBD. A way to treat protection errors within OSA itself must also be investigated.

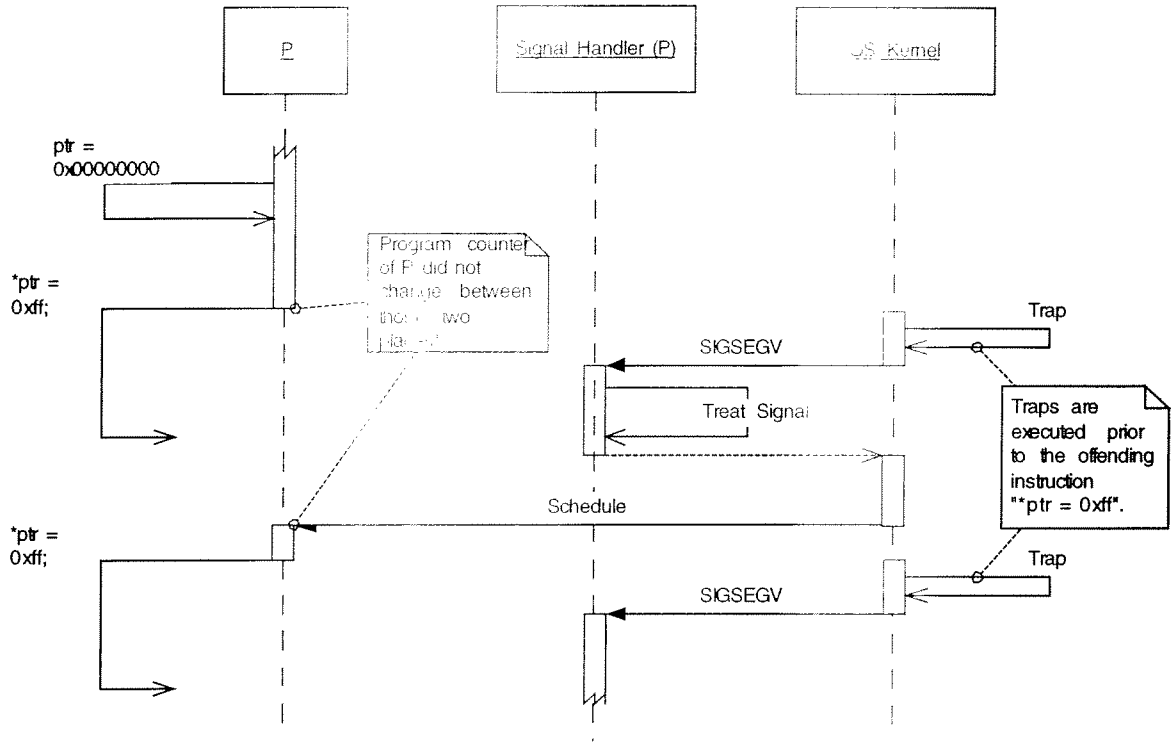


Figure 9-1: Treatment of SIGSEGV

9.3.1.1 Permanent Data

For permanent data the contents can be mapped at startup and protected `PROT_READ` from then on.

9.3.1.2 Semi-Permanent Data

For semi-permanent data the contents are also mapped at startup and protected `PROT_READ`. In case of the procedure `GPTSPF` the protection is temporarily changed to `PROT_READ | PROT_WRITE`.

9.3.2 Protection Through Run-Time Checks

Contrary to the approach presented in Section 9.3.1, where protection is achieved by OS mechanisms, it is also possible to achieve protection through run-time checks. Currently the CHILL compiler backend already inserts several run-time checks into the code (cf. /ErrorH/, Section 3.2), e.g. for checking array boundaries. For every statement that potentially accesses a protected area a run-time check has to be inserted, this can also be achieved through a dedicated OSA-Access Control component (cf. Figure 9-2). Since run-time checks impose a

performance overhead, it is desirable that the backend inserts such checks only at those places where it cannot verify a statement's inoffensiveness at compile time.

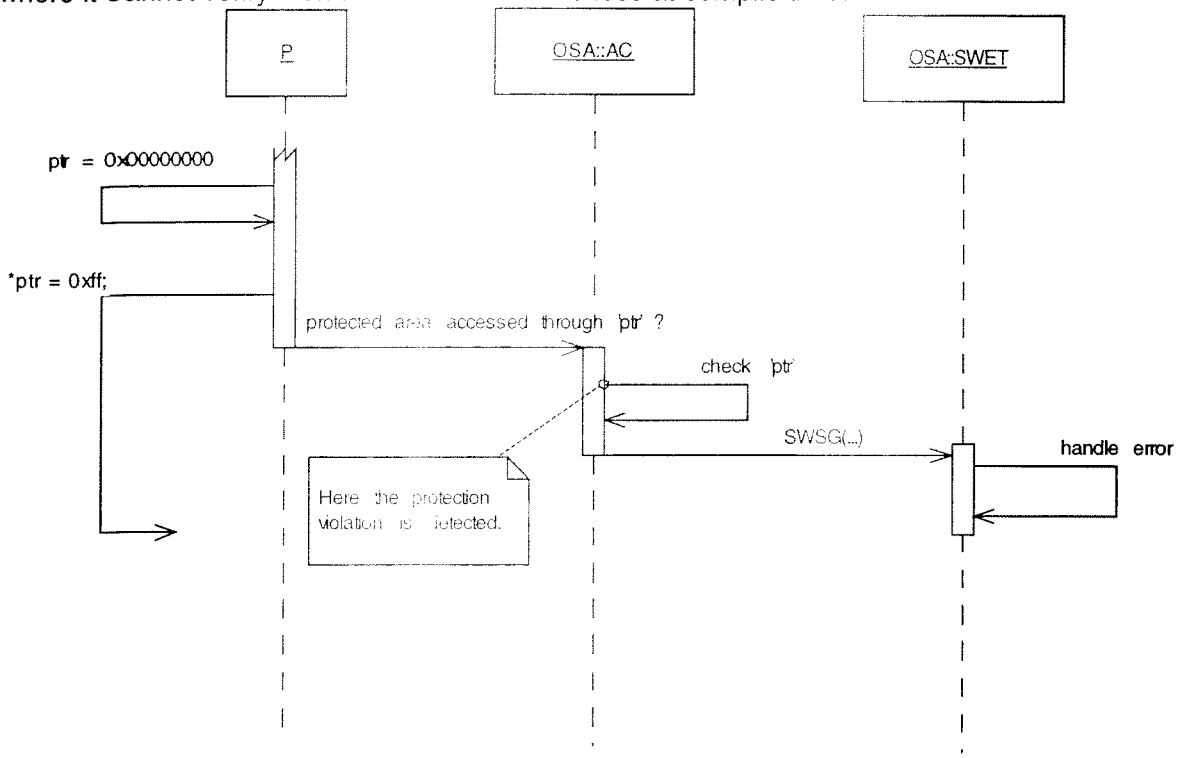


Figure 9-2: Memory Protection Through Run-Time Checks

9.4 Loading of Transient and Semi-Permanent Data

9.4.1 Low-level Load

Given a binary image on disc OSA can at any time open this image and write its content to dedicated memory locations through the standard POSIX `open()`, `read()`, and `close()` system calls. In case of POSIX processes OSA (as a process) can only access data in shared memory (e.g. libraries).

9.4.2 Loading by Mapping Files into Memory

`void * mmap(void * start, size_t length, int prot, int flags, int fd, off_t offset)` allows a POSIX process to map the contents of a file into its address space (cf. Figure 9-3).

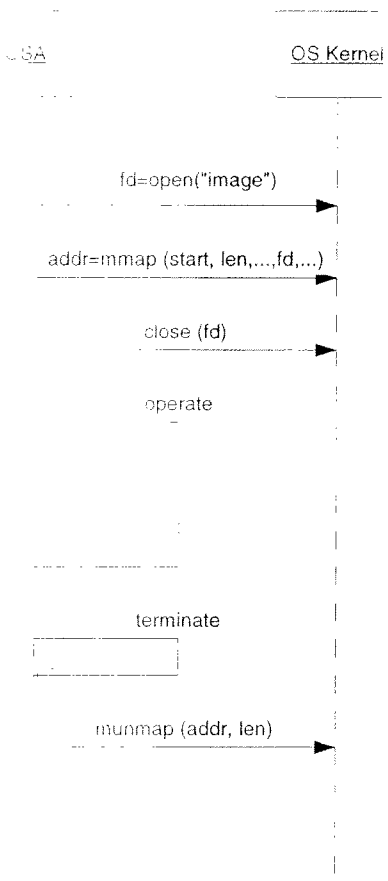


Figure 9-3: Mapping of Regular Files into the Address Space of a Process

The meaning of the parameters of `mmap()` is as follows:

- `start`: A pointer to the address where the mapped-in area should start. This usually cannot be chosen arbitrarily by the application, since it often must be aligned to the page-size of the OS.
- `length`: Length of the area that should be mapped.
- `prot`: Protection of the mapped memory area (cf. also Section 9.3.1).
- `flags`: either `MAP_SHARED` (this mapping is shared with other POSIX processes), `MAP_PRIVATE` (updates of the mapped area are only visible to this process). Or-ing `MAP_FIXED` to those flags tells the OS that it is required to map the memory definitely at the address specified through `start` (otherwise the OS is free to map the area at any address).
- `fd`: A file descriptor to an open file.
- `offset`: the offset into `fd` for the to-be-mapped data.

Care has to be taken that `start` contains a valid address, otherwise the OS may decide to chose another address or the call may fail at all (in case of `MAP_FIXED`). The application can determine the address where the OS chose to map through the return value of `mmap()`, which in fact is the address where the mapped in area starts.

If several POSIX processes map the same file it is not guaranteed that the file is mapped at the same address into the address space of the respective processes. If this is needed (e.g. for

trading of pointers), it has to be taken care of by the application itself. Note that mappings are inherited through the fork() system call. In this way it will be beneficial to establish all mappings in OSA before forking any POSIX child process.

In case of several POSIX processes, it has to be taken care that global data (accessible by all processes) is mapped by each process as MAP_SHARED so that updates become visible to other processes. Care must also be taken on the performance penalty induced due to this kind of file-based data. If it should turn out that the OS overhead to maintain file-based shared data is too high, then the approach depicted in Figure 9-4 has to be taken, where the to-be-mapped area is copied into shared memory by OSA. This approach has also to be taken if it is undesirable that updates of the mapped data are reflected in the underlying file (e.g. for transient data). Note, that the approach taken in Figure 9-4 is depending on the underlying object file format of the target OS (e.g. ELF; a.out,...) and has therefore be subject to further study.

As we already mentioned different portions of a file can be mapped to different memory areas. By that means, it is possible that separate link sections (e.g. UBFL, MILLOCON) that are placed in one binary image are mapped to distinct areas in the address space of the application.

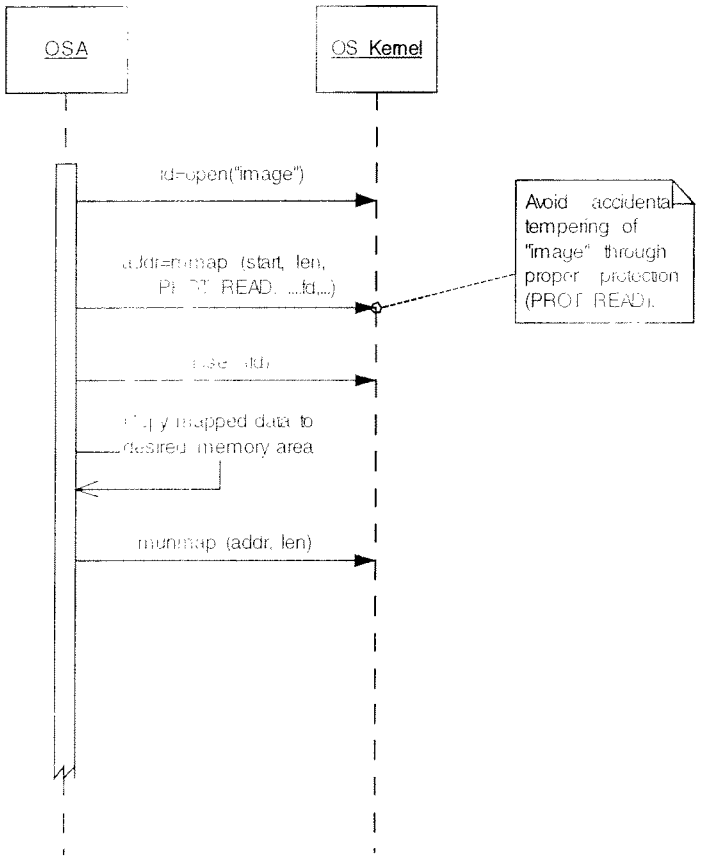


Figure 9-4: : Copying after Mapping of Regular Files into the Address Space of a Process

9.5 Loading of Code

Loading code of a process is a function typically located in the realm of the operating system, and not of the application software being executed by the operating system. We recommend a discussion whether this feature is really needed.

If it is needed, a possible solution would be to reduce the code of a process to just a wrapper that calls a function e.g. in a shared library that can be m-mapped (and un-mapped, closed, opened and mapped again) into the address space of the process.

10 Startup/Termination of Processes

10.1 START

START processname (buffer, start_no, <parameters>) allows starting of a CHILL process. The process with the name processname is started, on its stack it receives the optional arguments given in <parameters>. This parameter-list is of variable length. start_no is a number that is sent by the OS with the 'Start-' and 'Ende-Quittung', it must be kept by OSA for that purpose. buffer is the IPC primitive, to which the 'Start-' and 'Ende-Quittung' has to be sent. This is mapped as specified in Section 7 and must also be kept by OSA. Depending on the chosen model to map a CHILL PROCESS (POSIX process, POSIX thread, or task) OSA has to take appropriate measures as detailed in Section 9.1 to start the process. Note, that for POSIX threads only one argument can be passed. Therefore it is necessary in that case that the backend implicitly converts the variable-length argument list to one single argument that is passed to the thread upon SVC START. Measures to dissect this compound argument into its members as expected by the CHILL PROCESS code must also be provided.

10.2 PMSTARI

/Startup/ tells that this is a special kind of SVC only usable during system startup. A detailed description is deferred to /ProcSpez/. It can be considered as a 'Special Deal' between OS and application development.

10.3 OSSTP00

OSSTP00(process_name) starts a parameter-less resident process. Depending on the chosen model to map a CHILL PROCESS (POSIX process, POSIX thread, or task) OSA has to take appropriate measures as detailed in Section 9.1. Note, that due to process_name being a resident process, OSA must lock the pages of the process by the system call mlockall() in case of a POSIX process.

10.4 OSSTP02

OSSTP02(process_name, mmi_parameters) starts an MMI process. For the mode of mmi_parameters cf. /Startup/. This SVC is to be treated according to SVC OSSTP00.

10.5 OSSTP03

OSSTP03(process_name, mmi_parameters) starts a REGEN process.

10.6 PMINCAR

Returns the incarnation number of the calling process. If several incarnations of the same CHILL PROCESS are possible then OSA has to keep track of them.

10.7 PMINFP0_GET_OWN_INCAR_NO

Same meaning as PMINCAR, but available outside of the CP-SPU.

10.8 PMINFP1_GET_ALL_STARTED_INCARNATIONS

Returns all incarnations of the calling CHILL PROCESS. If several incarnations of the same CHILL PROCESS are possible then OSA has to keep track of them.

10.9 PMINFP2_GET_OWN_PID

Used to retrieve the so-called *process identifier* of a given process. /Startup/ only enumerates the member variables of such a process identifier, but defers the explanation of the meaning of those variables to /ProzSpec/, which was not accessible at the time of this study. However, it can be expected that the existing mechanism for maintaining the process identifier can be resembled by OSA.

10.10 PMPRNO

This SVC returns the *process-number* of the calling process. According to /Startup/ the process number corresponds to the PID of a process as maintained by a UNIX OS. Therefore this call is directly mapped to the standard UNIX `getpid()` system call.

10.11 PMGPRNA

This procedure returns the process-name for a given process-number. OSA needs to maintain adequate data structures to provide this mapping.

10.12 END

SVC END is used to terminate a process, freeing every resource associated with the PCB of the process. If it is detected during this SVC that resources are still held by the process (such as messages in a buffer or blocks of heap memory), adequate error handling methods are invoked. On demand a so-called 'Endequittung' has to be sent by OSA to the parent process. If the parent process has already terminated, this messages has to be suppressed. For a detailed format of the 'Endequittung' cf. /Startup/.

10.13 PMENDMS

A process within the CP-SPU can use this SVC to require transmission of an 'Endeqittung' to its parent. OSA has to maintain a flag for each such process telling whether an 'Endequittung' is requested. For non CP-SPUs this is required per default.

11 Error Handling

SWSG (Software Safeguarding) is a mechanism defined in /ErrorH/ that allows the handling of errors comprising determination and logging possible error causes, maintenance of error statistics, and initiation of proper recovery routines.

SVC `SWSG_x_LAB:SWSG (ErrorNumber, Indizienpaket, RecoveryLevel)` is the interface to this mechanism, the parameters have the following meaning:

`ErrorNumber`: Number in the range of 1-127, choose-able by the user to distinguish different kinds of errors.

`Indizienpaket`: A defined set of system-relevant data that is collected by the SVC and stored in the so-called 'Indiziensicherstellungsbereich', a memory area from where it is dumped to the file SG.SESYMP on the disc. 14 Indizienpakete ranging from 0 to 13 are defined, several of them are dedicated to certain parts of the OS- or user-software (which means that no-one except that certain OS or user software part may use it). Each distinct system-relevant piece of data contained in an Indizienpaket is called a *component*.

`RecoveryLevel`: System Recovery Level desired by the SWSG user. The different recovery levels are defined at the beginning of Section 9.

In addition to user-initiated SWSG calls several conditions exist that result in an SWSG initiated by the HW or by parts of the VOCOS OS. Treatment of those conditions is discussed in Section 11.2.

11.1 System-Relevant Data

/ErrorH/ enumerates all Indizienpaket-components. The following table lists those components as well as the source of information for OSA where they can be collected from. In some cases the component description given in /ErrorH/ is unclear, for such cases the description in the table is marked as 'TBD'. For some components it is not clear whether and how they will be represented in the new system, for those components the source-field is marked as 'TBD'.

Component	Description	Source
Internal Error	SWET-internal error number, if during error treatment an error occurs.	OSA::SWET, the OSA equivalent of the SWET component.
RecoveryLevel	System Recovery Level desired by the SWSG user.	SWSG SVC parameter
ErrorNumber	Number of the error picked by the SWSG user	SWSG SVC parameter
Modulname	Name of the module issuing SWSG.	Most effective would be to pass the name as a further (implicit) parameter to SWSG (to be checked with the compiler backend). Otherwise the code-address can be derived from the stack, a mapping from this address to the module must be established with help of linker map files.

Module number	Similar to module name.	Cf. Modulname
SVC-Modul-Name	Name of the module that issued the latest SVC to the OS.	OSA must remember the pid of the latest calling CHILL process. From that a mapping to the respective module can be established (with help of backend).
SVC-Modul-Version	Version of the module that issued the latest SVC to the OS.	Presumably the version number comes from the project library. The current mechanism on how this information is passed to the OS has to be examined and possibly rebuilt.
Fehler-unterbrechungs-adresse	Latest address, from where a call to SWSG has been issued.	OSA::SWSG stack
SVC-Unterbrechungs-Adresse	Latest address, from which an SVC has been issued.	OSA stack
Codeumgebung	The last 56 bytes of code before the SWSG/SVC call	OSA/OSA::SWSG stack->calling process's program code
R-Modul	Content of the semaphore-module of the interrupted code module.	TBD whether this is kept.
Gesperrter semaphor	Content of the semaphore that has been locked for too long.	OSA
Dringlichkeit	68k processor interrupt level at the time of the error detection (range:0-7).	Highly target hardware-dependent, to be customized to new hardware.
Registerstände	Content of the 68k processor registers.	Highly target hardware-dependent, to be customized to new hardware.
Prozessornummer	Number of the processor where an error was encountered.	Highly target hardware-dependent, to be customized to new hardware.
Prozeß der LOCK-Daten sperrt:	Process that keeps LOCK data for too long.	OSA
Deadlock-Prozeß	Process responsible for deadlock.	OSA
Aktueller PCB	Process control block of the currently executing prozess.	TBD how to get this from the new target RTOS.
IRH-Daten/Stack	Data/Stack of the interrupt hander.	TBD whether OSA will contain an interrupt handler.
Runtime-Stack	Stack of the currently executing process.	OSA::SWET stack

Master Stack	TBD	TBD
P-,T-Modul	TBD	TBD
relevanter T-Deadlockmodul	TBD	TBD
SYMO-Flags	System Monitoring Flags,TBD how system monitoring will take place.	TBD
SYMO-Trace-Bereich	'Indiziensicherstellungsbereich' of SYMO	TBD, OSA:SYMO?
Dump-Bereich	TBD	TBD
Vermittlungstechinische Daten	TBD	TBD
Interruptmaskierung	Interrupt mask before watchdog struck.	Highly target hardware-dependent, to be customized to new hardware.
SWET-Communication-area	Internal SWET data that gives an overview regarding currently running SWSGs on different processors	OSA, since we will have only one processor.
Dynamische AC-Slot-Tabelle	Memory area of the access control component	TBD whether OSA will have an AC component.
Access-Violation Register	TBD	Highly target hardware-dependent, TBD
Zyclus-Error-Register 1	Errors regarding CMY	Highly target hardware-dependent, TBD
Zyclus-Error-Register-2	Errors regarding LMY	Highly target hardware-dependent, TBD
PI-Alarm-Register	TBD	TBD
ECC-Error-Register	TBD	TBD
ERR-Adress-Register	TBD	TBD
Grund für Bufferüberlauf	Cause for a buffer-overrun and subsequent blocking of a process	OSA, provided that OSA maintains a shadow-state for the used POSIX.1b mqueue primitive.
Addresse des Buffers	Address of overrun buffer	OSA, address of the overrun POSIX.1b mqueue.
Modulname zum Buffer	Name of the modul that contains the overrun buffer	TBD, check current mechanism.
Prozess zum Buffer	Name and PCB of the blocked process that ought to read from the overrun buffer.	OSA, TBD to what extend PCB can be read out from the target RTOS (no POSIX primitives foreseen).
Belegte Ressource	Address of an occupied resource that lead to the blocking of a process. Resource may be a buffer, region, or	Buffers are mapped to queues, regions and events are mapped to semaphores.

	event.	This data is available in OSA.
Modulname zur Ressource	Name, version, and start-address of the module containing the occupied resource.	TBD, check current mechanism
Prozeß der Ressource	Processname and PCB of the process that occupies the resource (buffer, region).	OSA, TBD to what extend PCB can be read out from the target RTOS (no POSIX.1b primitives foreseen).
Bufferdaten	Additional data that may be specified at creation time of a buffer, TBD	TBD

Table 11-1: 'Indizienpaket'-Components

11.2 Modul-Independent SWSGs

Conditions that are specified unclearly in /ErrorH/ are marked as 'TBD'.

11.2.1 HW-Detected

Several HW-related SWSGs of this Section are mapped to POSIX.1 signals that have to be handled within the signal handler of the POSIX process.

SWSG-Condition	Mapping
Illegaler Adresszugriff (Zugriff nicht auf Halbwortgrenze)	target-dependent
Adr. von nicht ausgebautem Speicher	SIGSEGV (memory access exception)
SW-Fehler in beiden Speichern	TBD
falsche Einstellung in beiden B:CMY	TBD
beide Requestsperren sind eingelegt	TBD
AC Zugriffsverletzungen	very broad area, what exactly meant here??? ->TBD, but cf. also Section 9.3.2.
Function Code Fehler	TBD
WD-Ablauf bedingt durch SW-Fehler (Einzelprozessorausfall)	TBD
privilegierter Befehl in nicht privilegiertem Zustand	target-dependent
Unzuläßige Instruktion	SIGILL (illegal instruction exception)
Division durch 0	SIGFPE (floating point exception)
Undefinierter Interrupt	target-dependent

Table 11-2: Mapping of HW-Detected Error Conditions

Note: only processes can receive signals. In this way error treatment regarding the above mentioned SWSGs is distributed among all processes, as opposed to the current VOCOS

solution where a central facility handles them. This could however be circumvented by standard signal-handlers that call the central SWET facility on receipt of a signal. The problem does not exist with threads though.

Note2: proprietary RTOSs can have their own idea of how to treat exceptions, VxWorks for instance restarts the system if an illegal instruction is encountered in an interrupt service routine (cf. /VxGuide/, Section 2.5.4 ("Exceptions at Interrupt Level")).

11.2.2 SW-Detected

11.2.2.1 System Monitoring

SWSG-Condition	Mapping
Deadlock of CALLP or SM0MD	TBD
Timing-fault within certain priority level	TBD
Deadlock through buffer-overrun	Detectable through OSA::IPC, cf. Section 11.1.
Timing-fault within processor interrupt service routines	Target-dependent, dependent also on whether OSA will have a kernel-level part or not.

Table 11-3: SWSG-Conditions Detected through System Monitoring

11.2.2.2 Interrupt Handler

SWSG-Condition	Mapping
LOCK-Alarm after 5 or 100 ms	Handled by OSA, cf. Section 6.1.

Table 11-4: SWSG-Conditions Detected by the Interrupt Handler

11.2.2.3 Run-Time Checks

Those checks are inserted into the code by the compiler backend, cf. also Section 9.3.2.

11.3 OSA::SWET

11.3.1 OSA::SWET Startup

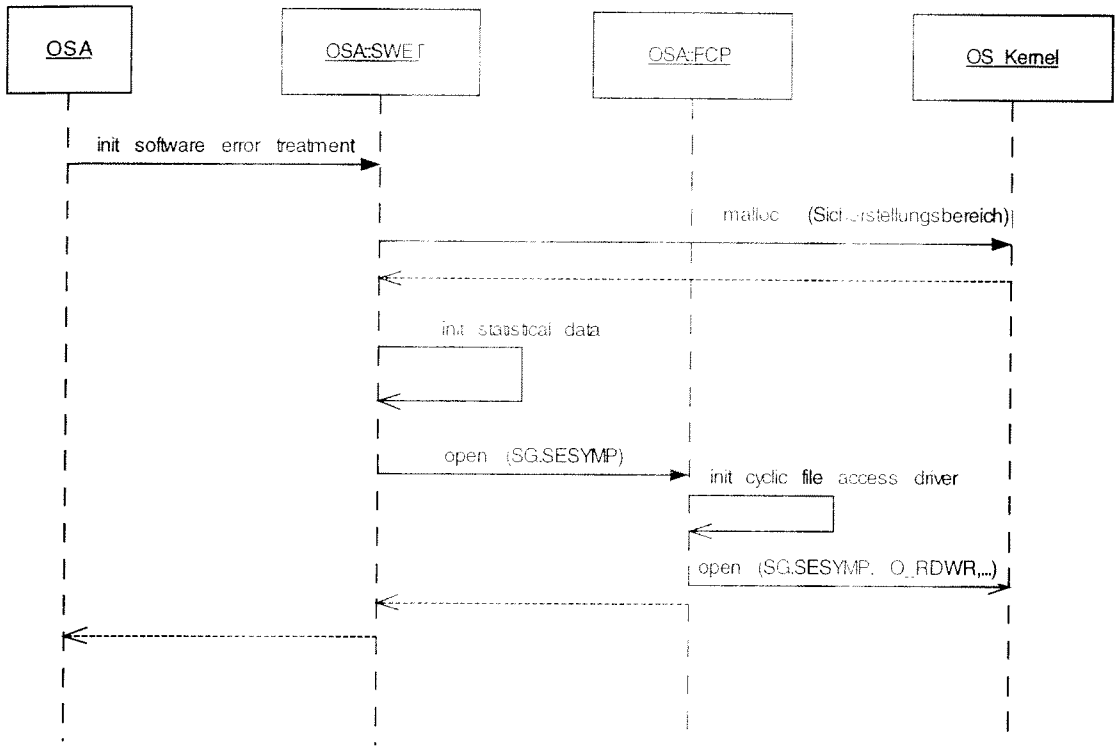


Figure 11-1: Initialization of OSA::SWET

Initialization of component OSA::SWET comprises the following tasks (cf. Figure 11-1):

(1) Allocating a block of memory big enough to host the 'Indiziensicherstellungsbereich'. Contrary to the allocation of the heap area (cf Section 12.3) this is done via malloc() instead of using shared memory since the 'Indiziensicherstellungsbereich' must only be accessible for OSA::SWET and not for all processes in the system.

(2) Initialize the statistical data OSA::SWET has to maintain in order to maintain a view on what SWSG calls were issued at which time in order to escalate to the proper recovery level according to the procedures described in /Update/, Section 'Eskalation'.

(3) Opening the cyclic file SG.SESYMP that is used to dump the 'Indiziensicherstellungsbereich' to. Since cyclic files are not provided by the standard UNIX file-systems, a separate component named OSA::FCP is needed to take care of that.

11.3.2 SASDAT

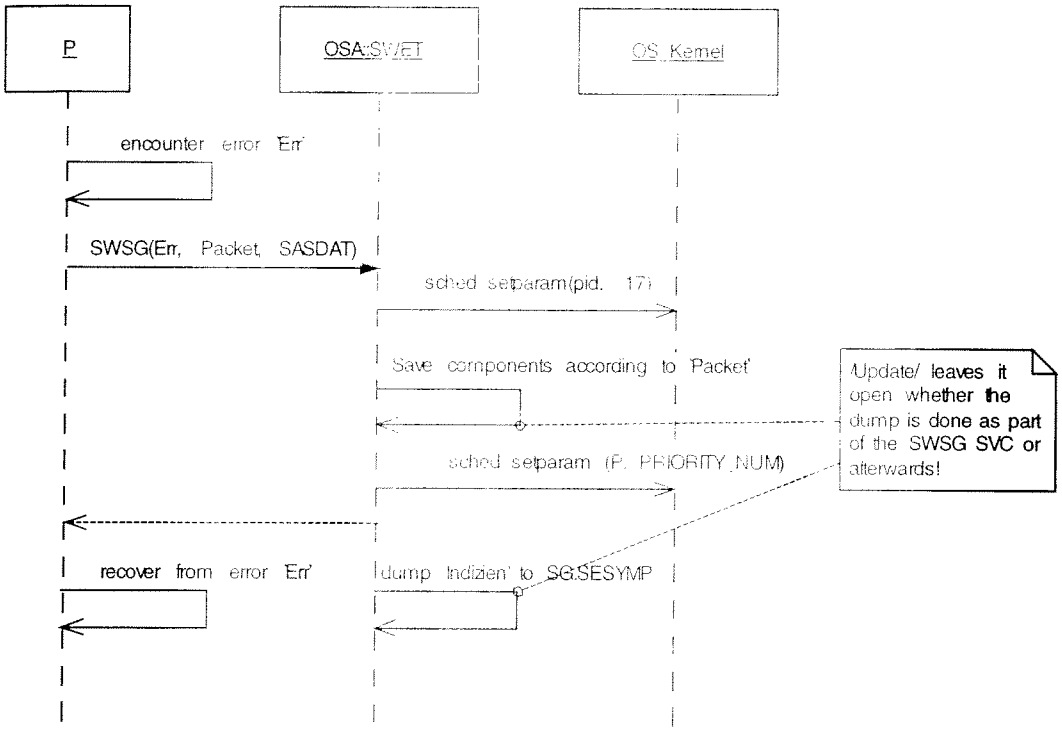


Figure 11-2: SASDAT Recovery Level

SASDAT is the only recovery level that does not escalate to an NSTARTx. Thus it is the duty of the calling process P to recover from the encountered error Err on its own. All SASDAT does is to save all components of the 'Indizienpaket' Packet in the 'Sicherstellungsbereich' which is then dumped to disc (cf. Figure 11-2). In order to make SWSG un-interruptible the priority of the calling process is temporarily raised to 17, the level of the OSA layer itself. /Update/ specifies that the system is temporarily stopped at SVC SWSG by raising the processor interrupt level of the SVC temporarily to 2 and later on even to 4. This behavior is highly target-dependent and it is yet TBD whether it can be resembled in this way on the new target RTOS.

11.3.3 SASDATS

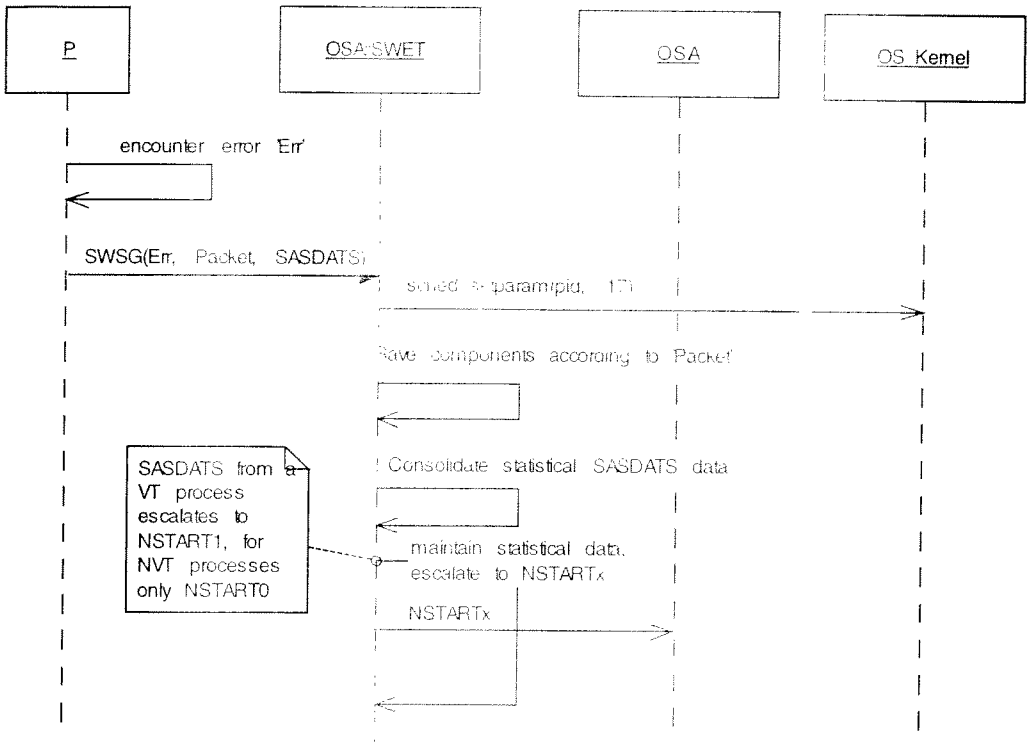


Figure 11-3: SASDATS Recovery Level

Contrary to SASDAT, SASDATS may escalate to NSTART1 in case of repeated SASDATS occurrences.

11.3.4 NSTARTx

The progression from SVC SWSG with an 'Indizienpaket' NTSTARTx is similar to the 'Indizienpaket' SASDATS (cf. Figure 11-3). The system is stopped (emulated by setting the priority of the calling process to 17), the components of the relevant 'Indizienpaket' are collected and stored in the respective memory area, maintaining statistical data and initiating the required system recovery level as specified in Section 9.

12 Heap-Management

Vision O.N.E. software applications may allocate and manipulate memory in the free store (the so-called heap). Space in the heap is allocated and released on demand. When the blocks in the heap are free to move, a heap-manager can often reorganize the heap to free space when necessary to fulfill a memory-allocation request. However, in some operating systems (e.g. VOCOS) blocks in the heap cannot move. In these cases, you need to pay close attention to memory allocation and management to avoid fragmenting your heap and running out of memory.

In VOCOS, requests for heap memory, like requests for inter-process communication, are made by special instructions, which trap to run-time system procedures. Therefore, we start with a short survey on heap-management facilities in VOCOS. Next, we discuss POSIX.1b mechanisms for shared memory, which we are going to use for *one* possible mapping from VOCOS heap-management (SVCs) to OSA.

12.1 Heap-Management in VOCOS

As indicated in /Heap/, the heap is a physically contiguous part of the general main memory. VOCOS defines the following SVCs to access and release free storage out of the heap.

`HM_GETHEAP_PAGED()`, `HMGETHP()` and
`GET_HEAP()` Heap-management provides these three different SVCs for requesting a free store. Whereas the two SVCs `HM_GETHEAP_PAGED` and `HMGETHP` need a parameter to identify the length of the requested heap, the required size is determined at compile time for the SVC `GET_HEAP`.

`REL._HEAP()` Heaps, that are not longer needed, must be released immediately via the SVC `RELEASE_HEAP`. Before the end of a process, it must be determined, that all heaps are released. If this is not the case, heap-management will force the release of these "open" heaps.

`HMTRAHP()` and
`HMSNDHP()` A process can transfer its allocated heaps to any process, that wants to receive those heaps. Notification of the receiver is dependent of the used SVCs: applying the SVC `HMTRAHP` leaves it open to the responsibility of the user to transfer the heap address to the receiving process (e.g. by the means of inter-process communication); issuing the SVC `HMSNDHP` the receiving process is informed by the heap-management (i.e., without explicitly using mechanisms of the inter-process communication at the application layer).

12.2 Shared Memory in POSIX.1b

In standard UNIX operating systems (and ANSI C as well), library functions `malloc()`, `calloc()` and `realloc()` are used to allocate free store and `free()` is used to release the allocated memory back to the free store. Since heap memory of VOCOS can be passed between different processes, we need a common memory area to accomplish such mechanisms. To meet this requirement we propose the use of shared memory, as explained below.

`shm_open()` The `shm_open()` function creates or establishes a connection between a shared memory object and a file descriptor. Flags, that have to be specified

with the shm_open() function define, whether the memory is used for reading (O_RDONLY) or both reading and writing (O_RDWR).

`mmap()` By means of the function mmap() the shared memory space may be mapped into the address space of a specific process.

`shm_close()` and
`shm_unlink()` These function calls allow to close or remove a shared memory object.

12.3 Mapping Heap-Management in VOCOS to OSA

This section provides a general description of how to manage blocks of memory in OSA. Be aware, that we focus on just one way to map the heap-management facilities of VOCOS to OSA. Hence, features like

- relocatable blocks,
- properties of relocatable blocks,
- heap purging and compaction,
- heap fragmentation,
- dangling pointers and
- low-memory conditions

are not discussed (since they are – as far as we know – not part of the VOCOS heap-manager, too).

During the initialization of OSA, the OSA Heap-Manager creates and opens a shared memory object for the globally accessible heap by means of the POSIX.1b function calls `shm_open()` and `mmap()`. Since processes are able to allocate memory concurrently, we need a kind of mutual exclusion mechanism, done via a special semaphore bound to the OSA Heap-Manager. This semaphore is created by the function call sem_open().

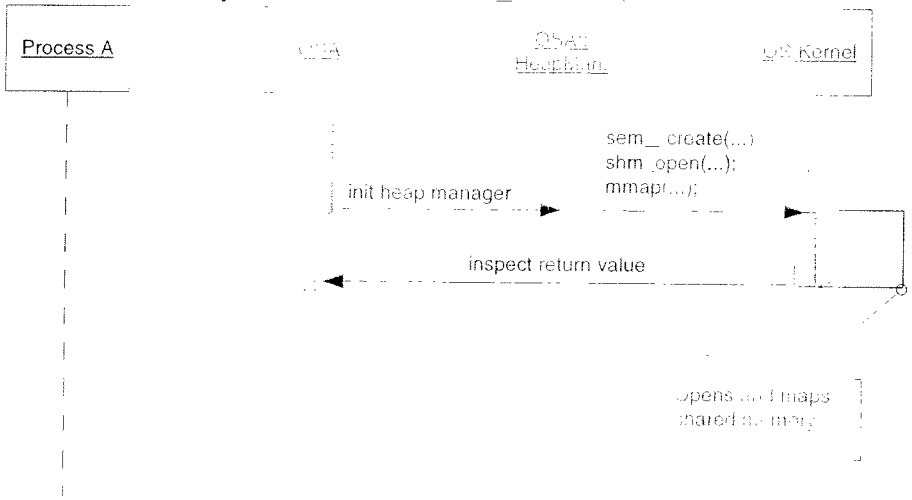


Figure 12-1: Initialization of the OSA Heap-Manager

Afterwards processes may allocate memory space out of the heap. Depending on the issued command the compiler back-end has to compute the required memory size. However, before requiring memory, the semaphore has to be checked (via sem_wait()). If no other process has entered the Heap-Manager, memory can be granted (if available).

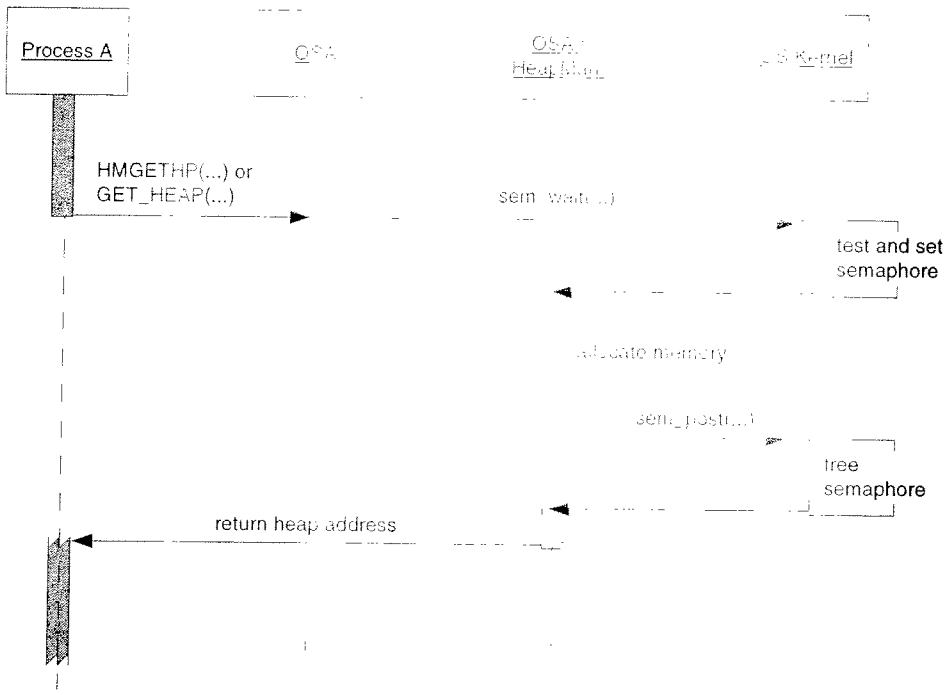


Figure 12-2: Allocating heap space

Releasing the memory is again task of the OSA-Heap-Manager. For internal use the Heap-Manager can maintain a list containing process descriptors and their heap-memory allocations. Dependent on the way the Heap-Manager works, this list can be used to implement further algorithms to combat memory leaks.

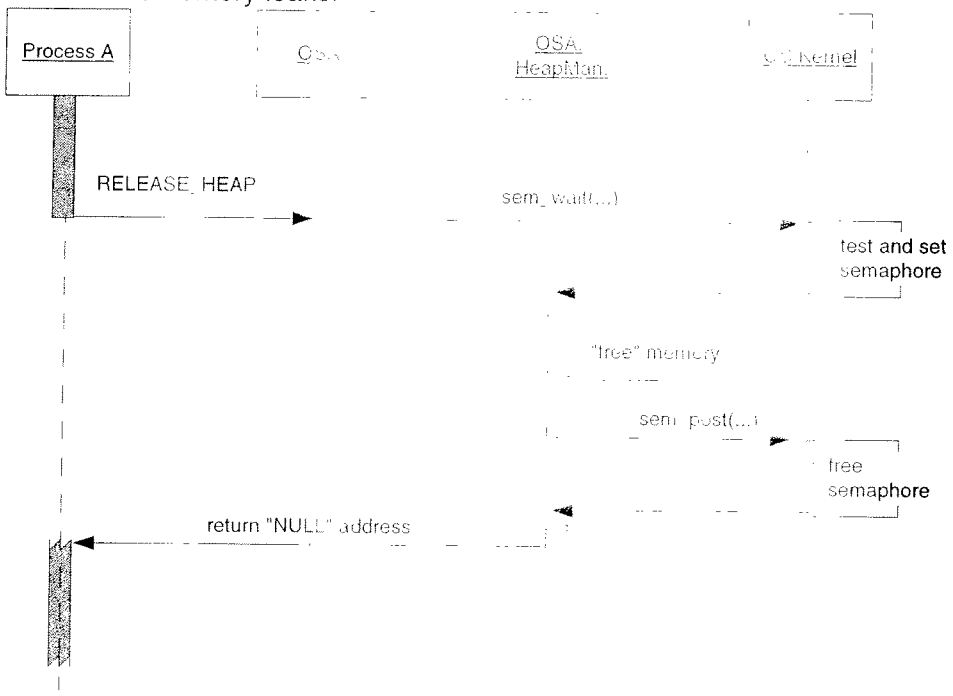


Figure 12-3: Releasing heap space

Transferring heap space between different processes is a more challenging task. However, strikingly simply – once it is discovered – by means of shared memory. In case of the SVC HMSNDHP, that allows transfer of heap with the notification of the receiver by the heap-management, the mapping could be done as follows. The OSA-Heap-Managers extracts out of

the parameters of the SVC the buffer address and translates it to the mapped message queue. Afterwards the internal data structures describing the relation between heap and process are changed to the new process (i.e. the receiver). Finally the heap address is sent to the receiver using the POSIX.1b function mq_s...

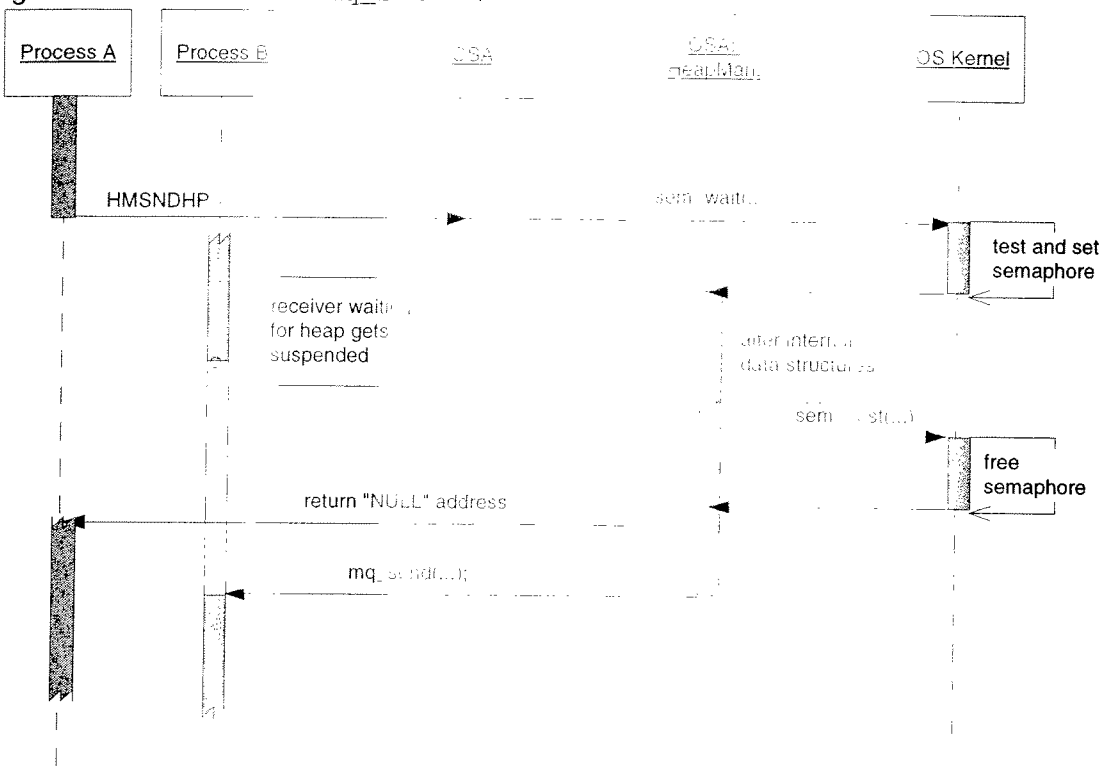


Figure 12-4: Transferring heap space

13 Update Management

According to /Update/ VOCOS SVCs provide mechanisms to modify semi-permanent data in the main memory. Means to write-protect semi-permanent data have already been discussed in Section 9.3. In order to provide the SVC `TRUSPV` on the new RTOS it is not necessary to rebuild the whole update functionality from scratch. Instead it is recommended to reuse the existing update component from VOCOS[10]. Taken this assumption for granted the `SEIZE`-interface of the existing update component to has to be determined (e.g. by checking the update design specification).

In addition VOCOS provides mechanisms to transfer semi-permanent data from main memory to external storage by using the `UPDATE` SVC. Again, we recommend reusing the existing code and adapt the `SEIZE`-interface according to the design specification.

[10] Keep in mind that transaction mechanisms are not part of the POSIX.1b specification.

14 Performance Issues

Part of /Pflicht/, namely worst case performance considerations regarding VOCOS and the new target RTOS, have not been considered so far. The reason for this is threefold:

(1) Timing specifications are not part of the VOCOS user documentation, although they contain some figures regarding measurements of specific SVCs (e.g. SEND and RECEIVE). During /Call2/ it was agreed that a complete instrumentation of VOCOS for measurement purposes will clearly exceed the scope of the feasibility study. According to /Call2/ the only stringent performance checks are in terms of overall system response and throughput to incoming requests from the net. Since this includes not only the OS kernel but the whole user software these measurements are probably on a far too global level to derive timing data for the VOCOS/CP kernel from it.

(2) At the moment timing data of target RTOSs is not available, too. In /SolarisRT/ performance figures for the SOLARIS API are promised, a request to SUN regarding this data did not produce any results. Availability of timing data regarding VxWorks is yet TBD.

(3) Estimations of the execution architecture of OSA can only be made as soon as the OSA design has reached a sufficiently mature level. This is due to the fact that in order to match the WCP of VOCOS/CP the following equation must hold[11]:

$$WCP(\text{target RTOS}) + WCP(\text{OSA}) <= WCP(\text{VOCOS})$$

Equation 14-1

[11] Note also that WCP need not equal measured performance data!

15 Outlook: Further Work, Risks, Effort Estimations, and Open Points

15.1 Work Breakdown Structure

15.1.1 Review and Solving of Open Points

At several places this study presents several possibilities of resembling VOCOS/CP behavior. Together with VOCOS specialists a decision has to be made for those cases.

15.1.2 Create OSA Design Document

15.1.3 Identify Interface to the CHILL Compiler Backend

15.1.4 Verify Feasibility of Chosen Approach

Based on the selected target RTOS and (a part) of the OSA design implement an OSA prototype in order to verify the feasibility of the chosen approach. Pick a few SPUs and have them execute on OSA.

15.1.5 OSA Implementation

Based on the OSA Design and the results of the OSA prototype implement the complete OSA layer.

15.1.6 Performance Considerations

As already pointed out in Section 9 worst case performance considerations regarding VOCOS and the new target RTOS have not been considered so far. To meet the WCP of VOCOS/CP this can be regarded an area of further activity. The following WBS applies:

(1) Acquire VOCOS/CP performance data

(2) Acquire performance data of target RTOS

(3) Estimate performance of OSA based on its design

(4) Verify that Equation 14-1 holds

15.2 Risks

(1) Most considerations of this study are based on VOCOS user-level documentation. When delving into the corresponding design specifications aspects not mentioned in the user-level documentation are often revealed. From this observation it can be assumed that the case is more complex than it might appear.

(2) Special Deals: VOCOS documentation sometimes mentions that apart from the presented possibilities special deals with the OS development department are possible. It can be

expected that not all those deals are known at present, and that those special deals will complicate the project.

(3) Implicit assumptions of the application software on VOCOS internals are likely to be broken if not explicitly incorporated into the OSA design.

(4) Shortcomings: This study has already pointed out that due to the in many cases very specialized nature of VOCOS/CP an emulation will at times have to accept shortcomings if it has to be done with an off-the-shelf commercial RTOS (by a shortcoming we denote a feature that cannot be resembled 100%). It can always turn out that such shortcomings are for various reasons in-acceptable from the point of view of the application software and that alternatives have to be provided.

(5) Parts of the problem domain (e.g. the OSA part resembling VOCOS scheduling behavior) are prone to race conditions (e.g. through preemption of processes executing in the OSA layer, due to the scheduler of the target RTOS) and even deadlocks (e.g. by terminating a process that has currently acquired an OSA resource such as a semaphore (cf. also Section 6.2)).

(6) POSIX-compliance of the selected target RTOS: compliance (or rather non-compliance) of the selected target RTOS can add additional effort to the project.

15.3 Effort Estimations

For the risks identified above and due to the fact that the software process of those carrying out the project is not known (the nature of the software process (light-weight, elaborate, not-at-all) will influence the effort that has to be estimated for each of the phases of the project life cycle), effort estimations can only be given on (very crude level).

Estimations are even more complicated due to the fact that no historical data of projects in that specific area are available.

Based on the findings of this study and the identified risks required effort has to be expected in the range of man-years rather than man-months. Several of the risks stated above add uncertainty (and therefore possible project delays) to the estimations.

15.3.1 Historical Data

All historical data used in the estimations is based on personal experience of the authors gained in various software projects. It is recommended to execute the same estimations with historical data gathered from Siemens-internal software projects, since this data will more accurately match what we have to expect in reality. Further figures for cost modeling can be found in /Cost/.

The following figures have been used:

Hours per Design Page: this figure determines the number of hours needed to produce one page of design (including review, rework, ...).

LOC per Design Page: this figure determines the number of source lines that are generated on average per design page.

Brutto Hours per MY: this is the total number of hours per year a SWE is paid for. This also includes holidays, illness, courses, activities that are not project related etc.

Netto Hours per MY: this is the number of hours per year a SWE can spend on a given project.

Software Life Cycle Phases: these figures determine the contribution in percent for each software life cycle phase to the overall cost of the complete software life cycle.

15.3.2 Selected Methods

For the following reasons we have refrained from estimating LOCs for the number of SVCs identified:

- Besides SVCs there exists also a prozedural API that has to be rebuilt. Estimating only SVCs would therefore miss part of the work that has to be carried out.
- Not all SVCs have been considered so far. /ProzSpec/ specifies a total of 55 SVCs, but the user-level documentation this study is based upon mentions only 25 of those 55 SVSs. Extrapolation would in theory be possible, but we consider this inaccurate since literally nothing is known regarding the remaining SVCS.
- LOC is a so-called *late metric*, which means that it can be measured only later on in the software life cycle. Regarding estimations this means that activities in the far future have to be estimated, the base of which is currently not known. We find it therefore more accurate to put the next phase of the software life cycle, namely the design phase, in the center of our estimations, and to extrapolate from that phase instead of the coding phase (cf. Section 15.3.3.4).

15.3.2.1 Extrapolation Based on Detailed Design Phase

We have divided the software life cycle into the following phases:

1) Functional Requirements: determining the functional requirements of the system. This includes also the software-software interface (SSI) between uSA and the compiler backend.
2) Top Level Design: achieving a top-level decomposition of the system into subsystems.
3) Detailed Design: decomposing the subsystems into a series of components for which a component design has to be made, which is the base for the coding phase.
4) Coding: self-evident.
5) Integration: putting together components into subsystems, and subsystems into the final system.
6) Testing: self-evident.

For the detailed design phase we have identified a list of design items (cf. Section 15.3.3.3) which have to be covered. With our approach we try to estimate the required effort for those design items. Given the effort-contribution of the design phase to the overall software life cycle, we can then extrapolate the overall effort from the estimated overall effort of the design phase.

15.3.2.2 Wideband Delphi

All estimations of the detailed design phase have been done using the Wideband Delphi technique, where each estimate is made independently by a group of individuals in a so-called *estimation round*. Thereafter the estimates are compared and results that are radically different are discussed. Estimation rounds are conducted until the results converge to a sufficient degree (10% in our case). This method is more reliable than estimations by single individuals since it requires a common consensus (cf. /Cost/).

15.3.2.3 COCOMO

Developed by TRW (cf. /Control/), this model is based on historical data of 63 projects, which were divided into three separate domains defined by product type and by certain characteristics of the project and its team members (e.g. embedded system projects were separated into their own domain). The effort for each domain is calculated as follows:

Mode	Effort
Organic	$MM=2.4*(KDSI)^{1.05}$
Semidetached	$MM=3.0*(KDSI)^{1.12}$
Embedded	$MM=3.6*(KDSI)^{1.20}$

Table 15-1: COCOMO Estimation Method

MM denotes the number of man months (one COCOMO man month equals 152 man hours), and KSD stands for thousands of LOC. Based on the LOC calculated in Section 15.3.2.1 we have calculated the COCOMO effort for the 'Embedded' class of projects. The reason to choose the most costly development class is that the criteria for class 'Embedded' are a complex environment (RTOS) and constraints that cannot be changed due to cost reasons (in our case the constraints imposed by the user software and the new target RTOS).
Note, that COCOMO does not incorporate the effort needed for the requirements phase, it only comprises design, code, integration and test phases. COCOMO requests another 8% of effort for the requirements phase.

15.3.3 Estimated Data

15.3.3.1 Accuracy

The following facts have to be considered when judging the accuracy of these estimations:

(1) Due to the fact that not all VOCOS SVCs are mentioned in the user level documentation a part of VOCOS has not been considered so far.

(2) The risks identified so far in the study (cf. corresponding Section in main document).

15.3.3.2 Historical Data

This section gives the figures for the historical data we have collected.

Historical Data	
Hours per Design Page:	10
LOC per Design Page:	50
Brutto Hours per Man Year:	1920
Netto Hours per ManYear:	1600
Software Life Cycle	
Functional Requirements (%):	8
Top Level Design (%):	10
Detailed Design (%):	17
Coding (%):	20
Integration (%):	5
Testing (%):	40
Sum (%):	100

Table 15-2: Collected Historical Data

15.3.3.3 Estimates for Design Items

Here we enumerate the items identified so far for which a design specification is needed. In the column entitled 'Pages' the estimator can enter his estimates. Based on historical data on hours per design page and LOC per design pages the effort and resulting LOC for the design item is calculated in column 'Effort for Pages' and 'LOC for Pages'.

For all estimates we assume that the estimated design document is on a ready-to-code level, which means that no further partitioning into sub-designs is necessary.

Item	Pages	Effort for Pages	LOC for Pages
Prozeßkonzept/Scheduling	110	1100	5500
Start/Endebehandlung	25	250	1250
IPC/Service Addressing	90	900	4500
Heap management	30	300	1500
Time Management	25	250	1250
Concurrency Control	40	400	2000
IOCP	110	1100	5500
Error Treatment	45	450	2250
Startup	25	250	1250
Recovery	60	600	3000
Sum:	560	5600	28000

Table 15-3: Estimates for Identified Design Items

15.3.3.4 Extrapolated Effort

Here the extrapolated effort based on the effort estimations of the detailed design phase is given.

Accumulated/Extrapolated Effort

Pages Design (h):	560
Effort Design (h):	5600
Extrapolated Effort (h):	32941.18
Extrapolated Effort (net. my):	17.16
Extrapolated Effort (my):	20.59

Table 15-4: Extrapolated Effort

15.3.3.5 COCOMO

Here we list the effort figures based on the COCOMO method. Note again, that this does excludes 8% effort for the requirements phase. Adding the effort for the requirements phase leads to the effort figure given in Table 15-4.

COCOMO

Effort (net. h):	29835.7
Effort (net my):	15.5
Effort (my):	18.6

Table 15-5: Effort Calculated by the COCOMO Method

15.4 Open Points

Several problem-areas have been identified in this study where more than one possible solution exist. For those cases we have attempted to enumerate all possibilities without bias in order to assist the reader in bringing about a decision. To provide an overview these open points and their respective Sections are listed here.
Moreover, in the course of the review of this document further points of interest have been identified, which are also listed here.

15.4.1 Mapping of the Chill PROCESS Primitive

Section 4 gives an overview of feasible approaches to map a CHILL PROCESS onto either POSIX processes, threads, or tasks. Which approach will be chosen is yet TBD. Throughout this document a process model is assumed, if not stated otherwise.

15.4.2 Number of CHILL PROCESSes

In the course of the review it has been reported that the current EWSD system consists of up to 1400 CHILL PROCESSes. Another remark mentioned 5000 such processes.

For SUN SOLARIS[12] /Mauro/ reports the following figures:

[12] Figures for other RTOSs have not been determined so far.

	Solaris 7		Solaris 8
	32 bit	64 bit	64 bit
Number of LWPs:	32768*	21845*	87381***
Number of Threads/Process:	2000**	2000**	>> 10000

*... Systems with at least 256MB physical memory

**...With default stack size for threads

***...1GB of physical memory

It is however recommended to verify the numbers mentioned above through a simulation of an EWSD system regarding aspects such as schedule-ability, communication, and synchronization.

15.4.3 VOCOS Memory Protection Mechanism

Section 9.3 lists two possible protection mechanisms for memory pages.

15.4.4 Traps

It is yet TBD whether it will be possible on the new system to realize VOCOS SVCs as traps (cf. Section 2.1).

16 Conclusion

In this study we have examined the possibilities available to offer the VOCOS OS functionality on a commercial platform. It was considered as a main goal to abstract from proprietary solutions by using only functionality supported by the POSIX standards.

Generally there can be no objection found that would prohibit the emulation of a given software system by another software system. It is therefore rather a question of whether the given requirements can be met under the imposed constraints. As requirements we denote the needs of the VOCOS application- as well as supervisor software, and by constraints we consider the need to come to a solution within reasonable time as most stringent. Due to this constraint we must consider the project as infeasible if the capabilities of the new target RTOS are too different from the requirements (e.g. it will probably take too much effort to implement a real-time scheduler from scratch on the target RTOS).

As it is suggested by this study, it is possible to map the core VOCOS functionality (process management, scheduling, IPC, and concurrency-related features onto POSIX primitives). It must however be noted that this mapping is not always straight-forward, and that it is not always possible to achieve 100% compliance. This imposes a certain amount of risk on the project (cf. also Section 15.2)! It must also be noted that due to time constraints it was necessary to focus on important aspects of VOCOS and to leave out other, presumably less difficult aspects. Based on the findings of this study, it is certainly necessary to develop a thorough and complete design and to have this design reviewed by experts in the VOCOS domain. As stated in Section 15.1, a prototype implementation of the most critical features is also recommended. A go/no-go decision for this whole project at the present level of coverage must be considered too early, the study has however been fruitful enough to carry out the next WBS steps.

We would like to mention that, no matter what platform will finally be chosen, if it is in the open-source domain, then there always exists the possibility to patch the RTOS kernel itself in case it should turn out that some VOCOS behavior cannot be resembled exactly in the application domain.As a final note we recommend further investigations finding out more pros and contras for OSA and a radically different approach, namely a pure hardware-emulation of the CP-system on a given target RTOS.