Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 79-96

PARALLEL PRINCIPLES USED IN THE OpenVMS OPERATING SYSTEM

P. Czabala (Budapest, Hungary)

Abstract. This paper gives a brief overview of the techniques and methods of handling parallelism in the OpenVMS operating system. Our work is based on manuals provided with the operating system, many experiments and command procedures written for testing purposes and simple application programs using selected system services. We will highlight what system services and facilities are available for the end user and for the programmer.

1. Introduction

This paper is about parallelism in the OpenVMS [1] operating system. It gives an overview of the features which provide methods for the user or the programmer for using and handling parallel architectures.

We will talk about processes and threads, shared resources, mailboxes, and locks, and how processes and threads can communicate and synchronize with each other [2, 3]. We briefly explain these services and show some simple examples on how they can be used.

This work gives a brief discussion of the subject, while highlights features we consider interesting in theoretical sense.

1.1. General description of OpenVMS

The Virtual Memory System (VMS) operating system has a constantly increasing popularity since the early 70's, when Digital Equipment Corporation

Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T017800

released the VAX/VMS 1.0 operating system. It provides many forms of computing capabilities such as interactive, batch and real time processing; different computing styles including timesharing, multiprocessing and distributed client-server processing.

2. Parallel and distributed hardware

2.1. Multiprocessing

First of all, let us say some words about the underlying hardware. All the following methods are common in that the support for them is *higly integrated* into the operating system.

By OpenVMS terms the parallel computing environments can be divided into three categories:

- *Symmetric multiprocessing*, when all the processors run the same copy of the operating system, use common memory, share the file system and other resources. The system acts as a single computing node. The processes can be run in parallel on different processors and the operating system syncronizes and manages the communication between them. The distribution of processes on the processors is automatical, it is mostly transparent to the users and to the programs.

- A *VAXCluster* system consists of CPUs connected together, each CPU may have one or more processors. CPUs share computing capacity, mass storage such as file systems and other resources. All the CPUs run their own local copy of the operating system, they can be managed together or independently, the whole VMSCluster can be treated as a single unit by outside systems. Several methods exists for inter-process communication in a clustered environment even if the processes are on different computers. In that case the communication between processes is also mostly transparent for the processes (and for the programmer), it is efficient, low level while network-error-prone.

  The symmetric multiprocessing model and the VAXCluster system are very strong parts of the OpenVMS operating system. They are integrated to the system deeply and in a consistent, coherent way. These are ones from what OpenVMS is famous for.

- *Networked* configurations are built from independent computing nodes, however they can use many of each other's resources using common

network communication techniques with protocols such as DECNET or TCP/IP. With DECNET networks, the users and the application programs can use files over the network just like they were local files. The Logical Name Service described later can be used to simplify that task.

2.2. Processor models

The VAX family of processors are Digital's CISC architecture models. They all runs the OpenVMS VAX operating system, which is functionally equivalent with the OpenVMS AXP operating system. Existing code can be easily migrated from the VAX family to the AXP family. The OpenVMS AXP is the port of the OpenVMS operating system to the Alpha processor, Digital's industry leader performance RISC chip.

2.2.1. VAX vector processors

Digital has extended the traditional scalar architecture to have vector processing capabilities in contrast to the scalar architecture's memory-to-memory model. This feature has been announced for from the 6000 series of VAX processors to the 9000 series. For most of the operating system's code and most of the application programs, the handling of vector processors is managed automatically by the operating system. However, there are several system services for direct use of the vector processors.

Example 2.1. *The VAX 9000 series computer is one of the largest member of Digital's VAX family [5]. It is a shared memory multiprocessor based on a 2 GByte/s crossbar switch. With vector processing capabilities a four processor VAX 9000 has a peak performance of 500 MFlops.*

3. Terminology about processes

3.1. Processes and threads

Now turn to the software.

The OpenVMS operating system is based on core components, services, utilities and the handling of processes. The operating system itself consists of several processes, and the users interact with the system by creating and using processes.

The concept of process is well known in other operating systems also. The difference between them is that the OpenVMS system itself is based on some

processes, Unix systems rely mainly on a *kernel*, while the VM/ESA operating system uses the *virtual machine* concept as a guideline [6, 7].

3.1.1. Detached processes and subprocesses

By OpenVMS terms, processes can be grouped into *jobs*. A job consists of a detached process and zero or more subprocesses. A detached process has no parents, it exists independently from other processes. A detached process is created when a user logs in from a terminal, when some network action occurs from another hosts (in a VMSCluster environment, for example), or it can be created on demand by operating system utilities or application programs. As an example, the TCP/IP network handler software creates some detached processes for listening the network device activity. These kinds of detached processes are called daemons in Unix terminology. The difference between Unix and OpenVMS processes can be summarized as that the Unix processes form a tree with an *init* process in the root while OpenVMS processes form a forest with the detached processes in each tree's root.

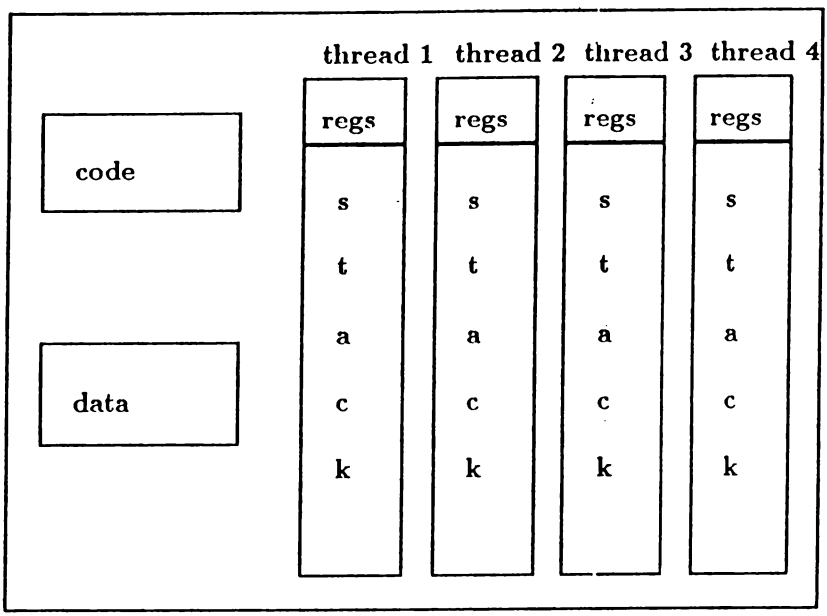

Fig.1. Threads in OpenVMS

Any process can create any number of subprocesses within the limitations exposed by the system user authorization file. Subprocesses must terminate before the parent process terminates. Subprocesses share most of the parent

process' environment (such as logical name tables), more precisely, a subprocess has a copy of the parent's environment and it has some additional information stored within it. Subprocesses are often used to accomplish a particular task such as setting some logical names and compiling a program.

The subprocesses are independent processes of their own apart from that they got a copy of the parent's environment, and that they must terminate before the parent. As independent processes, they all have their own memory area independent from the parent's.

3.1.2. Threads

A thread is a single, sequential flow of control in a program. Within a thread, there is only one point of execution. With the aim of *multithreaded programming*, it is possible to use interleaving parallelism within a process. Every process has its own set of registers and stack, but the code and program data exists only once in the memory. In every hardware model only processes can be run parallel on real processors, threads provide only apparent concurrency.

OpenVMS provides three kinds of services for writing multithreaded programs, these differ in their features and portability. Two of them conforms to the POSIX standard and the third one is Digital's own version for OpenVMS. In functionality all three variants are similar.

4. System services

In the following we will explain the available techniques and system services of the OpenVMS operating system with the interest of parallel or concurrent programming.

4.1. Event flags

Event flags can be used for posting status information between processes. With system services, one can set or test event flags, or can be held in a wait state while some particular event occurs. *Local event flags* can be used only within a single process, while *Common event flag clusters* can be shared by cooperating processes in the same group. Common event flag clusters can be used by processes to notify and synchronize each other, so this is a simple method of communication.

4.2. Asynchronous system traps

Asynchronous System Traps (ASTs) are used when a user program wants the system to interrupt asynchronously its execution when an event occurs such as completing an I/O operation, passing a scheduled timer interval or requesting the update of a section file on a disk. Delivering an AST is simply means calling a pre-specifed routine within the user program. Most process states[2] are interruptable by an AST.

For example it is a common technique to set a process in a ready-run state, place it in a *hibernated* state and let an AST wake it with very little delay when an event occurs. This method is used often in network handlers where the process has to answer very quickly to the network events.

4.3. Logical name services

OpenVMS name service includes logical name service and distributed name service.

Logical names provide a technique manipulating string data by placing information in logical name tables. There are several name tables, some are local to processes, some can be shared.

Logical names can be used to communicate information within processes by having one process defining a value in a shared logical name table and the others translate (read) it, or simply test the existence and/or value of a logical name for synchronization.

The Digital Distributed Naming Service provides similar features accessible in a network environment, such as a DECNET network.

4.4. I/O services, mailboxes

It is out of the scope of this paper to describe the huge variety of the different input-output services with respect to parallel processing, however some highlights are included:

- *Quotas, privileges and protection* of many kind provided limiting the access to the I/O devices by one process or a job of processes, to ensure availability of resources by other processes and the integrity of the OpenVMS operating system.

[2] E.g. LEF, Local Event Flag wait - for example a process waits for an IO operation, when the IO completes it sets the appropiate Event Flag.

- *Queuing I/O requests and synchronizing service completion.* When an application program requests an I/O operation from the system[3], the request is queued, and the process can continue execution immediately. It can choose a method for testing that the request is completed: an event flag can be specified to be set when the operation finished, an AST routine can be declared to call upon completion, the routine can specify a memory location as an address of the I/O status block to be written to after completion, or it can even directly wait the completion of the I/O. In the last case, a different system service, which does not return to the caller until the operation, has been finished[4].

- *Mailboxes* are virtual devices that can be used for communication between processes. One or more processes can write data into it concurrently while others can read from it. All accesses to the mailbox are synchronized. Processes can be set to a wait state while a message is read or written by some other process. Channels[5] assigned to mailboxes can be unidirectional or bidirectional, mailboxes can be temporary or permanent (that is it is not destroyed when no processes use it anymore). The operating system itself uses mailboxes to communicate with processes; mailboxes can even be set to receive a message by the parent, when a subprocess terminates. We can call mailboxes as *limited distributed queues.* Unix sockets are similar to the VMS mailboxes.

4.5. Timer and condition-handling services

Program activities can be scheduled based on system clock time. In particular, one can use the timer services for scheduling and AST for a given process, scheduling a wakeup request for a hibernated process, or cancel a wakeup request not yet processed. Process wakeups can be scheduled to occur only once or at a regular interval.

Special procedures can be declared to give control to when exception occurs. An exception occurs for example when a process tries to access an illegal memory location, when some arithmetical operation fails, when the process stack becomes invalid, an overflow occurs, etc.

---

[3] It is always achieved by using the $QIO (queue I/O request) system service.

[4] $QIOW (QIO and wait) can be used.

[5] These channels are managed by simple file-handling operations, such as Open, Read, Write, Close. The Open statement or system service can specify whether the channel is read-only, write-only or both.

4.6. Lock management services

A resource is an entity of the system that a process can read, write or execute. OpenVMS lock management services synchronize processes' access to shared resources. These services are effective only if all processes accessing a particular resource use it. Processes can stay in a queue while the required resource becomes available.

When a process issues a lock request, it must specify the name of the lock, the lock mode defining how the process wants to share the resource and the address of a lock status block. The lock request must be queued and after completition of the request the result is written to the given address.

If no other processes placed a lock on that resource, or the already placed locks are compatible with the one requested, then the lock will be accepted, otherwise it will be placed in a queue. A process also can change the mode of the lock, this is called a *lock conversion*.

Resources often have smaller parts, which can also be treated as resources, and these parts also can be locked or even having subparts, which are themselves resources, etc. For example it is possible to lock an item in a record, or lock the record in a file or lock the file, or open the volume containing the file. This is *granularity*, for example locking an item is a *fine granularity*, locking a file may be considered as *coarse granularity*. As an other example one can lock a file concurrent read, while other processes may have locked some records in it for concurrent write access.

There are six lock modes:

- *Null mode* means no lock at all, but it may express a process' interest on the resource or it may be later converted to other modes.
- *Concurrent read* lock grands read access to a resource while having any other mode of locks to be placed on the resource. For example a mailbox or an indexed file can be opened in concurrent read mode to read information from it, lets other processes write to it.
- *Concurrent write* gives write access to the process in a manner similar to the previous one. It makes possible to write "uncontrolled" to the resource.
- *Protected read* is the traditional share lock: other processes can read it but no one may place a write lock in it.
- *Protected write*: it is the traditional write lock. Concurrent read, but no other access is allowed.
- *Exclusive*: no other locks allowed on the resource. This is the traditional exclusive lock.

With each resource there are three queues associated: the GRANTED queue for already granted requests, WAITING for locks waiting to be granted and CONVERSION for the lock requests whose conversion mode tries to change a higher level.

If any group of locks are waiting for each other in a circular fashion, there is a *deadlock*. The OpenVMS operating system detects these conditions and breaks them selecting a *victim* process, whose request is denied, and notifying it about the deadlock condition in its lock status block. Granted locks, however are never revoked, only waiting locks can be selected as a victim.

For the notification of a process of completion or failure of a queued lock request, similar techniques to the case of I/O requests are available.

Note that regardless of locks every I/O operation is atomic in the sense that the state of the resource is always consistent.

4.7. Distributed transaction management

Distributed Transaction Management (DECdtm) services provide a complete and consistent way for using transactions in distributed system. Transaction can be started, ended or aborted. Each rquest can be invoked by awaiting its completion or by not awaiting it.

Under control of these services the application program, which determines the atomic transactions to be performed, and the *resource manager* and *transaction manager* processes work together.

The transaction manager serves the application programs' requests by sending instructions to the resource manager, which manages shared access to a set of recoverable resources.

4.8. Process control

Processes can be created, deleted, *suspended* or *hibernated*, they can be awakened by explicit call or by an AST or timer routine. They can exit or forced to exit and it is possible to set up special routines being called when the process terminates.

The cooperation and interaction of processes are controlled by process privileges. Each process can access other processes with the same UIC (User Indentification Code). Different privileges can be granted to processes to handle other processes in the same user GROUP or in the WORLD (for example a process with WORLD privilege can issue process control services for any process in the system).

4.8.1. Process information

Information about processes can be obtained under control of privileges similarly to process control. The processes to be examined can be given by their unique process indentification code (PID) or by their name within the same group.

4.9. Memory management

OpenVMS memory management routines control the relation between the processes virtual address space and physical memory. These operations generally are transparent to the user process, so there are rare occasions when a user program must directly control these operations. These include:

- Defining, extending or deleting virtual address space for a program.
- Locking or unlocking memory pages onto/from memory.
- Defining, updating or deleting global section or section files. These services let user processes to have some of their memory mapped into a physical file or the system paging file. Using this method, many processes can access concurrently and rapidly the same information. In addition, if processes use global section files, they can modify quickly the contents of the file, having the system paging mechanism write back it to the file, at the same time syncronising the data access of the cooperating processes.

Sharable code can also mapped into memory using image sections. A very fast method of process communication is using *page frame sections*, when physical memory is mapped into the processes' virtual address space.

There are also available *private sections* for the same purpose, but they are used exclusively by only one process.

5. Example programs in Ada

In this section we will show some examples on how to use the different system services. The examples are written in Ada83 language.

5.1. Ada and the OpenVMS

The DEC Ada compiler is a validated implementation of the Ada 83 object-based language. The object-oriented Ada 95 standard is not supported by the

DEC Ada compiler, however one can use the GNAT Ada 95 compiler which also uses threads for implementing tasks.

The DEC Ada compiler implements the Ada *tasks* using threads. This means that every Ada program executes in one single process. If one wants to execute the Ada tasks on different processors in different processes or even in different computers, the Ada95 language's Distributed Annex could be used. The following examples not only shows the way how system services can be used, but they give a basic idea on how it would be possible to implement the Ada95 distributed partition model to distribute the tasks on many processors.

In other languages like the DEC C++ special libraries are provided for multithreaded programming and other system services.

5.2. Event flags

This procedure is for setting an event flag or wait for one. It can be used by several processes to notify each other when a particular event occurs. It is similar to a binary semaphore known from the literature.

This program itself is quite simple, it shows the simple and uniform way of calling system services.

```
with STARLET, CONDITION_HANDLING;
use STARLET, CONDITION_HANDLING;

procedure CEF(set_operation :  in BOOLEAN) is

    status:  COND_VALUE_TYPE;
    err:  exception;

begin

AscEfc(status,64,"CLUSTER"); -- assign common event flag cluster
-- we give a name and an id to the cluster, and receive the
-- return status of the system call

if not Success(status) then -- this is the standard method for
raise err;                  -- checking that the system call was
end if;                      succesful

if set_operation then
    SetEf(status,64);            -- set event flag
                                (maybe somebody listening to it)

else
```

```
    WaitFr(status,64);          -- wait for event flag
end if;

if not Success(status) then
    raise err;
end if;

exception
when err ⇒ Signal(status); -- print descriptive error message
                           - in VMS style format
end CEF;
```

In this program we used the STARLET predefined library package. This packege contains direct Ada interface to most system services, while the LIB package can be used instead for mode comfortable, more portable routines. We used this only procedure for the set and wait operations because the code for them differs in only one line.

5.3 Mailboxes and logical name service

Here follows an example on how to use mailboxes and logical name service. In the program given below we create a mailbox, retrieve its name from the job level logical name table and place it in the group name table, so the same user or an other user in the same group can read it in different jobs. Note that special privilege (GRPNAM) is needed to write to the group logical name table. At the end of the program we write some text into the mailbox.

A similar program can be used to read the text from the mailbox.

```
with STARLET, TEXT_IO, CONDITION_HANDLING, SYSTEM;
use STARLET, TEXT_IO, CONDITION_HANDLING, SYSTEM;

procedure MbxWrite is
    test_mode:  constant BOOLEAN := true;

    status:  COND_VALUE_TYPE;
    chan:  CHANNEL_TYPE; -- channel to communicate with the
                         -- mailbox

    iosb:IO_STATUS_BLOCK_TYPE;
    -- after the IO completed, here we can see whether it was
    -- succesful.
```

```
    itmlst:  ITEM_LIST_TYPE(0..1);
    -- this is a null item terminated list of items.
    -- these items are used for logical names

    buf:  STRING(1..250); -- for the logical name

    ret:  UNSIGNED_WORD; -- length

    err:  exception;

    mbxname:constant LOGICAL_NAME_TYPE:="MBX_CZABY"; -- this will
                        -- be the logical name of the mailbox
    -- we will use instead of the real name
    tabname:constant LOGICAL_NAME_TYPE:="LNM$JOB".
grptabname:costant LOGICAL_NAME_TYPE:="LNM$GROUP"; -- the name of
                                        - the logical name tables
    -- to which we will insert
    -- the logical name
    -- these names are standard
    -- vms name tables for
    -- every job and group

    s:  STRING(1..1024):=(others ⇒ ' '); -- string to transmit
    l:  NATURAL;                        -- length of the string

-- we use two different error check procedure for word and double
-- word arguments.

procedure ErrChk(status:  in COND_VALUE_TYPE) is -- error check
begin
    if not Success(status) then
    Signal(status);
    raise err;
elsif test_mode then
    Signal(status);
    end if;
end ErrChk;

procedure ErrChk(status:  in WORD_COND_VALUE_TYPE) is
begin
    ErrChk(COND_VALUE_TYPE(status));
```

```
end ErrChk;

begin

    -- Create mailbox

    Crembx(STATUS ⇒ status,
                CHAN ⇒ chan,
                LOGNAM ⇒ mbxname);
    ErrChk(status);

    -- The logical name MBX_CZABY must be moved from LNM$JOB to
    -- LNM$GROUP to let processes in other jobs read it

    -- Read the present value

    itmlst(0):=(BUF_LEN ⇒ buf'length,
                ITEM_CODE ⇒ LNM_STRING,
                BUF_ADDRESS ⇒ buf'address,
                RET_ADDRESS ⇒ ret'address);
    itmlst(1):=(BUF_LEN ⇒ 0,
                ITEM_CODE ⇒ 0,
                BUF_ADDRESS ⇒ ADDRESS_ZERO,
                RET_ADDRESS ⇒ ADDRESS_ZERO);

    TRNLNM(STATUS ⇒ status, -- transfer logical name:  it gives
                TABNAM ⇒ tabname,      -- the name associated with
                LOGNAM ⇒ mbxname,      -- the logical name
                ITMLST ⇒ itmlst);
    ErrChk(status);

    Put_Line("Buf="&buf(1..NATURAL(ret))); -- see the name on the

    -- screen, create new name in lnm$group

    itmlst(0).BUF_LEN:=ret; -- this is the real length of the
                            -- buffer

    CRELNM(STATUS ⇒ status, -- create logical name
           TABNAM ⇒ grptabname,
           LOGNAM ⇒ mbxname,
           ITMLST ⇒ itmlst);
    ErrChk(status);
```

```
    -- write to channel

    s:=(other => ASCIL.NUL);
    s(1..5):="Hello";

    QIOW(STATUS => status,  -- Queue the IO request, and Wait for
                            -- completion.
        CHAN => chan,
        FUNC => IO_WRITEVBLK, -- we want to write a virtual block
        IOSB => iosb,
        P1 => TO_UNSIGNED_LONGWORD(s'address),
        P2 => 5; -- UNSIGNED_LONGWORD(250));
    ErrChk(status);          -- was the queueing succesful?
    ErrChk(iosb.status);      -- was the IO itself succesful?

exception
    when err => PUT_LINE("Something wrong!");
end MbxWrite;
```

6. Interprocess communication

In the previous chapters we have described the basic facilities and the most important system services related to parallelism. From an another point of view we summarize the techniques and methods. The following techniques can be used for interprocess communication:

- *Files* can share arbitrary amount of information, while this is the most time consuming method due to physical access to the disk. However, the simultaneous or shared access to a file can be precisely controlled by quotas, privileges and protection as mentioned above.
- *Common event flag clusters* for processes executing in the same group. They can signal each other for a particular event.
- *Logical name tables* can be used for defining and translating logical names with equivalence names (strings). A wider range of processes can access a logical name table, stronger privileges are needed to write to the table.

- *Mailboxes* can pass information, message or data between processes. Mailboxes also can be set up to receive information on how a particular process has been finished.
- *Global Sections* can be either disk files or page file sections containing code or data. These sections can be mapped into one or more process' virtual memory area. System paging occurs when one process writes into a common global section, resulting that data is written back to the disk file or the system page.
- *Lock management system services* can be used to control simultaneous access to resources. Some information can be passed to the other processes using the same resource with lock value blocks. Processes can be notified by blocking AST's when other processes are waiting for a resource. In addition, automatic deadlock detection is performed by the OpenVMS operating system.

7. Summary and conclusion

In this paper we have shown that the OpenVMS operating system has well designed, robust facilities in the area of parallelism. We have shown some popular parallel hardware devices available from Digital. Using the explained system services the application programmer can choose from several methods when writing programs for multiprocessor or distributed environments. With the help of some simple example programs we described and illustrated the use of these system services.

We can conclude that the OpenVMS operating system provides a good framework for writing parallel applications, it has most of the traditional properities and capabilities needed for an operating system. However it is lacking some of the new features known from modern Unix-like operating systems, and has quite significant (and sometimes annoying) overhead in the system services and system calls.

The two main area where OpenVMS systems are used are database management and general access-to-the-network support for many users with medium or high security requirements. For systems focusing on database management, it provides comfortable and secure, reliable solution. At systems, where many users use it simultaneously to access different network services, it has the advantage of high security and reliability for the system administrator and a consistent and easy to use interface to the end user and the systems or applications programmer. It may not be the best choice for systems where

speed is much more important than security or the high quality of system services.

This previous statement refers mostly to the VAX architecture and the operating system's OpenVMS VAX version. In the near future the classical points of view about OpenVMS systems will certainly change. Digital's new marketing strategy concentrates on the Alpha processor based systems. The new OpenVMS 7.0 and later versions contain many of those new features from Unixes and other modern systems regarding to for example the efficient and state of the art network access facilities. The description of the new features imposed, and prediction about the future operating systems may be interesting, but are out of the scope of this paper.

References

[1] *OpenVMS Software Overview, Version 6.1,* Digital Equipment Corporation, Maynard, Massachusetts, 1994.

[2] *Introduction to VMS System Services, Version 5.5,* Digital Equipment Corporation, Maynard, Massachusetts, 1991.

[3] *OpenVMS System Services Reference Manual, Version 6.1,* Digital Equipment Corporation, Maynard, Massachusetts, 1994.

[4] *OpenVMS Programming Concepts Manual, Version 6.1,* Digital Equipment Corporation, Maynard, Massachusetts, 1994.

[5] **Trew A. and Wilson G:.,** *Past, Present, Parallel,* Springer, 1991.

[6] **Csizmazia B.,** A UNIX operációs rendszer, draft paper, ELTE TTK, not published.

[7] **Welsh M.,** *Linux Installation and Getting Started V2.2.2,* mdwsunsite.unc.edu, Linux Documentation Project, 12 February 1995.

[8] Many INTERNET on-line resources such as the *Lycos search engine* available at <URL:http://lycos.cs.cmu.edu/>

[9] **Csombók J.,** *Párhuzamos programok írását támogató nyelvi elemek,* MSc Thesis, ELTE TTK, Budapest, 1992.

[10] **Czabala P. and Koszik T.,** *Üzenettovábbítás transzputer hálózatokban (Message Serving in Transputer Networks,)* BSc Thesis, OTDK Thesis, ELTE TTK, Budapest, 1992.

[11] **Horovitz E.,** *Magasszintű programnyelvek,* Műszaki Könyvkiadó, Budapest, 1987.

[12] **Kozics S.**, *Az ALGOL 60, a FORTRAN, a COBOL és a PL/I programozási nyelvek*, ELTE TTK, Budapest, 1992.

[13] **Kozics S.**, *Az Ada programozási nyelv*, ELTE TTK, Budapest, 1992.

[14] **Milenkovic**, *Operating Systems: Concepts and Design*, McGraw-Hill Book Company, 1987.

[15] **Barron D.**, *Computer Operating Systems for micros, minis and mainframes*, Chapman and Hall Computing, 1984.

[16] **McCrum W.A.**, Open System Interconnection and the Integrated Services Digital Network, *New Advances in Distributed Computer Systems*, ed. K.G. Beauchamp, D.Reidel Publishing Company, 1982, 87-96.

[17] **Bustard D., Elder J. and Welsh J.**, *Concurrent Program Structures*, Prentice Hall International, 1988.

[18] **Fortier P.J.**, *Design of Distributed Operating Systems: Concepts and Technology*, Intertext Publications Inc., McGraw-Hill Inc., 1986.

[19] **Horn C.**, Is Object Orientation a Good Thing for Distributed Systems? *Progress in Distributed Operating Systems and Distributed Systems Management, Proc.of European Workshop, Berlin, FRG, April 1989*, eds. W. Schröder-Preikschat and W. Zimmer, Springer, 1990.

P. Czabala
Department of General Computer Science
Eötvös Loránd University
VIII. Múzeum krt. 6-8.
H-1088 Budapest, Hungary
czaby@dtalk.inf.elte.hu