1. Introduction

In the past several years, the research of operating systems has shifted from the concept of isolated machines to the notion of a distributed operating system [7][9][10]. At the same time, single-machine operating systems started to use microkernel architecture [1]. This paper describes the T4 project - an attempt to design and to implement a microkernel, suitable for both single-machine and distributed operating systems, and a distributed operating system using this microkernel.

Contemporary implementations of distributed operating systems impose high requirements on hardware [3][10]. Since such hardware is not widely available there was a great emphasis on reduction of hardware requirements in this research project. Another reason for this approach is an intent to use the microkernel as a basis for single-machine operating environment.

Among other important design issues, excluding any special compatibility features from the microkernel allows implementation of arbitrary system interface to user programs. Thus, for example, compatibility with UNIX is obtainable with minimum effort while the microkernel itself is not loaded with any embedded compatibility [5]. Of course this approach probably causes certain loss of performance compared to systems designed with the compatibility in mind, but it was not the performance of UNIX programs that inspired this project.

The authors tried to focus mainly on the following topics:

- kernel speed
- quick message passing
- transparent remote communication
- unified process environment
- transparent object and process migration
2. System Architecture

The architecture of the T4 system consists of the following layers: The hardware, the microkernel, the by-kernel, system and user processes.

2.1 Hardware

The authors tried to design the T4 system with minimal dependence on hardware. All hardware-specific parts are strictly separated from the rest of T4 so that porting to any currently used platform would be easy and straightforward. The design allows both single- and multiprocessor machines with minimal restrictions to processor type.

2.2 Microkernel

The T4 microkernel deals with these tasks: Process and thread creation and deletion, processor time sharing, local message passing, thread fault handling, physical and virtual memory management, low level hardware operations and hardware interrupt handling.

The level of these services is significantly lower than their analogues in usual operating systems (including many microkernels). For instance, the time sharing algorithm in the T4 microkernel does not recognize any priorities, the virtual memory support does not understand the notion of swapping in and out; it only performs requested page table operations, notifies page faults, and performs a very simple page-discarding algorithm.

This layout enables building different systems based on this microkernel - single machine as well as distributed operating systems with good IPC support. T4 itself is one example of such a system.

2.3 By-kernel

Above the microkernel, there is a layer of modules providing basic functionality of operating system. This layer is called by-kernel. The difference between the kernel and the by-kernel is in the fact that by-kernel modules run with user privilege level and have only indirect access to kernel entities.

Compared to the user and system modules, by-kernel processes run in a restricted computing environment, because their task is to provide services to form a complete computing environment for the processes. The microkernel provides certain services (e.g. memory allocation) for them in restricted and simplified manner, while the complete range of the same services for the system and user processes is provided by the by-kernel layer. The most important by-kernel modules are: Process manager, virtual memory managers, timer, primary file system, net manager.
2.4 System processes

System processes are loaded by by-kernel after its initialization and their use may be categorized into four groups:

- enhancement of by-kernel services (richer access to more media etc.)
- extending interprocess communication to remote machines
- public services (e.g. remote file servers)
- management of user protection domains

The most important system processes are: Remote communicator, secondary file systems, name server, server manager, process loader, user interface managers.

3. Interprocess Communication

In this chapter we will describe basic types of communication between processes. By interprocess communication we understand not only communication between two or more separate processes, but communication and synchronisation between any threads, even communication within one thread.

It is commonly believed that communication is one of the most important design issues of microkernel based and distributed operating systems with considerable impact on overall system performance [2][3]. Therefore great attention to communication was paid in T4 system design. The main goals of the T4 communication scheme are:

- As fast as possible "short" message passing
- Effective passing of large amount of data - mapping instead of copying whenever possible
- No unpredictable events like exhausting some resources due to the function or malfunction of other processes
- Unifying the expressions of rights to send a message and to use something
- Trusted local objects and services invocation without need of any encryption mechanism or sending long identification bitstreams
- Safe communication with untrusted or unknown counterpart process
- Possibility of transparent extension to remote communication
- Straightforward process and object migration without significant performance degradation

In the T4, a message is not only a piece of data passed between two threads or processes, but an abstract concept which may consist of three parts:

- synchronisation, i.e. the fact one thread has sent the message and so another one is able to receive it
- rights and relations being handed over to another process
- information - the data being sent

The T4 message passing system (as incorporated in the kernel) provides local interprocess communication, but offers no remote communication facilities. On the other hand, the definition of the message passing principles allows transparent extension of the message passing to remote computers by means of non-kernel processes.
In the simplest case, it means that there may be a couple of proxy processes inserted to the communication channel between two processes where each process communicates (through kernel facilities) with the proxy on its side while the proxy processes are connected on their own, e.g. through sending packets over a network. The key issue is that the communication through proxy processes is transparent for the communicating processes, i.e. that the message passing scheme is strong enough to allow emulation and weak enough to be emulated.

3.1 Communication Principles

At the bottom of every message-based system, there is the send/receive paradigm. If there are only point-to-point messages (i.e. if there are no broadcasts) and if there is no message loss the send/receive model becomes a duality - every message sent is to be received just once and every message received was sent just once.

With client-server model, there comes another duality - request/reply dichotomy of messages. In an ideally functioning system, every request message must be sooner or later answered with a reply message and vice versa.

The T4 system introduces the third case of duality between running a thread and passing a message. This rule is inspired by the observation that real programs usually combine sending a message with waiting for another one - a client program sends a request and waits for an answer, a server sends a reply and waits for another request. In other words, sending a message causes loss of right to run for the sending thread and receiving a message revives the receiving thread. This observation leads to the notion of logical thread: A computation that is continuously transferred among communicating processes, switching between the running state - executing within a thread - and the passing state - waiting for a thread as a message. In the T4, logical threads (and not the physical ones) form the base of the synchronisation and time-sharing system.

The message passing system combines all the three dualities mentioned above:

- The basic message passing primitives (i.e. the kernel calls) always combine sending a message with receiving of another one.
- The implementation of the logical thread relies on the term stamp. Stamp is basically a "right to run"; any thread may run only in the case it owns one stamp and any message may be passed only if it is coupled with a stamp. Physically, the stamp is one of the basic kernel entities; the next chapters describe them and their handling from the point of view of a process.
- The message sending operations reflect the request/reply protocol: The most important modes are push and pop for sending a request and a reply, respectively. The names push/pop come from the fact that the execution of a logical thread in a client-server environment resembles stack operations. The push operation sends a message via a one-way communication channel, typically from a client to a server. The pop operation returns the reply back to the sender of the corresponding push without a need to establish a channel. The message passing system maintains a stack of successive push operations (called envelope
stack) associated with a logical thread and allows replying to a request without specifying any information about the target.

Although both the send/receive and the request/reply dualities are present in the communication primitives, it does not necessarily mean that the message received immediately after sending a request must be the answer to it. This feature is frequently used by servers while simple client processes usually conform to the suspending mode of service calls. These mechanisms allow advanced logically multithreaded servers to be implemented using only one thread.

3.2 Vertices

The most basic and most important entity used for interprocess communication is a vertex. Vertices help threads to run, send and receive messages, synchronize processes etc. They are maintained by the microkernel and they are parked in slots which are organised in arrays called racks. The racks are attached to processes; a process may access only the vertices parked in its racks. The fact that the vertices are addressed through its position in the process’s racks allows transparent process migration.

There are several basic types of vertices:

*bag* - Bags are used to merge input from different channels: All messages coming through receivers into one bag are queued here and later passed to a thread asking the bag for a message. In the complementary case, threads may be queued in the bag while waiting for messages. The number of messages or threads in a bag is not limited. There may be any number of receivers connected to a bag; on the other hand, every receiver is connected to exactly one bag. To be able to distinguish messages coming through different receivers, the owner process of the bag and the receivers should assign them magic numbers, which are attached to the messages passing through and later returned to the receiving thread.

*transmitter* - This vertex is used for transmitting messages: Every transmitter is connected to exactly one receiver, a receiver may be connected to an unlimited number of transmitters. Transmitters may be created either by attaching to a receiver or by copying existing transmitters (without the owner of the receiver noticing) and they may be easily passed via communication channels. The messages passed through different transmitters to a certain receiver are undistinguishable. Transmitters have attributes which may be used to restrict the set of messages that can be passed through them. With these attributes, it is possible to transfer partial rights to access objects and also to use the transmitters not to pass messages but to prove rights to something.

*stamp* - Stamp is a "right to run". The stamp may be either active, passive or bound.

The active stamp represent the top of an envelope stack and it is attached to a thread. When the active stamp is detached from the thread and waiting for execution, it becomes passive. Bound stamps form the middle of the envelope stack. Stamps are of two kinds - pure stamps (forming the bottoms of execution stacks) and stamp ends of envelopes. The former is used only to represent the right to run, the latter is also used for message passing.
Envelope - Envelope is used to represent a message. In its life, the envelope may be used to pass a message repeatedly; there is no need to allocate resources for every message. Some of these envelope states are called stamped: Stamped envelopes have two ends, one of which acts as a stamp. A message is sent when it is put into an envelope, the envelope is then stamped and sent through a transmitter.

Receiver - Receiver is able to receive envelopes, more exactly to pass envelopes to the bag attached to the receiver.

The following figure shows the pictograms used to denote various forms of vertices and entities mentioned in this document:

![Fig. 1. Basic kernel entities](image)

### 3.3 Basic Communication Scheme

The T4 communication system is based on channels formed by trusted kernel entities, like transmitters, receivers, bags, etc.

Fig. 2 shows a communication channel with some messages already transmitted. With such a channel, every transmitter holder (i.e. sender) can send a message through to the receiver holder (i.e. addressee). Messages coming to the receiver are passed to the corresponding bag where they form a queue. Every thread asking the bag for a message gets then the first available message in the queue.

![Fig. 2. Basic communication scheme](image)

When a thread sends a message, it must use an envelope, fill in the appropriate message data and then ask the system for posting. Since all transmitted envelopes must be stamped, the thread's active stamp is used (upon receiving, it becomes the addressee's active stamp). The sender loses its stamp and to be able to run again, it must obtain another one. Incoming messages have stamps glued to them, so
obtaining a new stamp is easy - the thread must receive a message. Therefore the sender must specify a bag from where it expects an answer.

When returning the envelope back to its previous sender, the destination is known as the message envelope points to its former sender. In the other case, the thread must specify also the message destination (by specifying the transmitter through which the message should be sent).

3.4 Message Types

In the T4, there are several message types. The basic message type is the short message containing up to 4 words of 32 bits (one machine word) each. To send a longer message, one must use a vanilla page or simply vanilla (the term was - although slightly modified - borrowed from the SPRING system). It is also possible to add an attachment - a set of transmitters. For convenience, the authors have designed message passing so that there is no difference between sending messages of different types.

For purposes where 4 words are not enough, a vanilla page is used. Vanilla messages may store up to 4 KB in this page. The mechanism is the same for both vanilla and non-vanilla messages - there is one virtual page associated with every envelope (non-vanilla messages do not use it). The vanilla is mapped into the process's address space for the process to be able to access it. When sending the message which uses a vanilla, the vanilla page is mapped out of the sender's address space and mapped into the addressee's address space so that the destination process may use it without the data being affected by the sender.

Both short and vanilla messages may contain an attachment representing some rights represented by transmitters, eg. a right to share a memory region, a right to access some object or service, etc. There is no special kernel service for sending messages with attachments; their existence is evident from the envelope being sent. All vertices which are about to be sent out as an attachment, are detached from the sender's rack and "packed" in the message. While sending, no special actions are necessary, the attachment is transferred automatically.

4. Message Passing

For safe and easy use of the communication primitives, the kernel maintains the envelope stack of request calls for every virtual thread, so it can be very easily returned back and back to the original request sender either as voluntary response or in case of process crash. There are three different modes of sending a message; they can be distinguished by what logically happens to the envelope stack:

- push - the message is sent further to a server and a new envelope is added to the envelope stack;
- pop - the message is returned to the sender of the last push into a bag specified by "return link";
- forward - the message is forwarded to another destination without any changes in the envelope stack.
As mentioned above, the sending of a message is always coupled with receiving of another one, so the sender thread must always specify a bag from which it wants to receive a new message (and then a new stamp to run on). Note that message reception does not depend on the mode of message sending, for instance the message received after pushing one might be not only the response to the sent one but also another response or message pushed by another thread as well.

In the following sections, the reception phase of message sending primitives is not mentioned and the need to specify the bag for reception is not explicitly stated.

4.1 Push

This variant of sending can be viewed as sending a request from a client to a server.

The sending thread must specify what (the envelope to be sent) and where (the transmitter through which the message should be sent). The system then performs the following actions (Fig. 3a shows relevant information about the situation before the action):

- A stack link is created for the specified envelope to point back to the stamp the sender was running with. Since the sender's stamp is already linked to another stamp using its stack link, those links form the envelope stack mentioned above. Every stack begins with an active or passive stamped envelope at top and leads through some bound stamped envelopes to a pure stamp forming its bottom
- The envelope is changed to a stamped one, the stamp will serve as the receiving thread's right to run in the future
- The envelope is put into the bag denoted by the communication channel

After completing the sending phase, the reception phase of the push primitive is performed as mentioned in the previous section.

![Fig. 3. Situation before and after push](image)

So far, we have described the "push" mode of message sending. Now let us concern on receiving a message that has been pushed.

The receiver's thread may ask the destination bag for a message only when it has posted something else out and so it doesn't have any active stamp to run with. When such situation occurs, the first message in the queue is taken out and passed to the
receiving thread. The received envelope's stamp end is parked into the receiver's rack (note that the other end of the stamped envelope is still parked in the sender's rack). This stamp becomes the thread's active stamp and the receiver may thus continue running and process the received message. For convenient usage, the thread also gets identification (magic number) of the receiver the message came through. The situation after receiving a pushed message is shown at Fig. 3b.

4.2 Pop

This variant is used mainly as an answer from a server back to the client - currently active envelope (the stamp end of which allows the server to run) is sent back to its original sender (i.e. the client).

The thread doesn't need to specify the destination as it is determined by the other end of the active stamp. Neither does it need to specify the envelope to be used to store the message, therefore the pop operation has no parameters other than the message contents (and the bag to receive next message as mentioned earlier).

When the client - the original sender of the push - withdraws the popped message from the bag's queue, it continues to run on the stamp it was running before the push while the message envelope is changed to unstamped one. After receiving the popped message, the situation is exactly the same as before the original push.

5. Process Environment

In this section, we will give the reader an idea about the environment a process lives in. We will describe mainly the situation just after new process is born as it shows all the relevant relations between processes and between process and the system.

Every process has at least one rack. Any vertex must be linked from the rack for the process to be able to access it. It is possible a vertex comes as a part of a message; in that case, the process must explicitly ask the system for unpacking it and linking it to the rack (the system does it as an atomic action).

After being born, a process has the following vertices parked in his rack: one active stamp, one free envelope, one bag and one transmitter; the envelope is tied to the bag and the transmitter is connected through a via to a receiver somewhere else.

The stamp is the active stamp ("right to run") of the main thread. It is this thread the process begins to run with. For the environment to be the same for all new processes, the only way to ask for anything is to send a message using the transmitter; that is also why there is one free envelope created with every new process. For the process to receive anything, it needs also one bag where messages come. The initial transmitter is connected somewhere and it doesn't matter where. The only thing the process must know is that anything it requires must be asked for through that channel (e.g. establishing new connection to any kind of server, other process etc.).
6. Current State and Future Work

Currently, the T4 is an operating prototype. The microkernel (both mono- and multiprocessor versions) is completed by now. Our future work will concentrate on the improvement of system services such as support for process migration, support for distributed memory sharing, etc. The most important (and the most interesting) tasks of future T4 development are:

- An implementation of a UNIX environment emulator.
- Fully transparent process migration.
- Graphical user interface - based on the XWindow system.
- Ports to other platforms.

Although the basic layout of the T4 is uncommon and may seem strange, our project showed that the basic paradigms used in the system design conformed to most of our expectations. Thus, our approach represents an acceptable tradeoff between the complexity of basic primitives and requirements imposed by intended applicability in the area of distributed systems.

7. References