Artificial Intelligence Operating System

Article · October 2017

CITATIONS
0

READS
9,975

1 author:

Sergey Korneev
BaltRobotics Sp.z.o.o. (http://www.baltrobotics.com)
35 PUBLICATIONS **6** CITATIONS

SEE PROFILE

Operating systems to base AI-applications: the overview and general technical requirements (in English)

Article · August 2017

CITATIONS
0

READS
9

1 author:

Sergey Korneev
BaltRobotics Sp.z.o.o. (http://www.baltrobotics.com)
18 PUBLICATIONS **0** CITATIONS

SEE PROFILE

Sergii Kornieiev

Operating systems to base AI-applications: the overview and general technical requirements

Abstract
This article analyzes the prospects of creating operating systems for AI-applications to be based, and the general requirements for such systems that would better meet the expectations of users on reliability and security, including malicious software and threats of a various nature.

In what areas are currently concentrated applications of *artificial intelligence* (AI)?
First of all, this is:

- "intelligent" Internet applications (analysis of consumer and electoral preferences, organization of companies to manipulate public opinion);
- autonomous vehicles, drones, marine mobile vehicles (UGV - "unmanned ground vehicles", UAV - "unmanned aerial vehicles", USV - marine "unmanned surface vehicles", AUV/UUV - "autonomous underwater vehicles"/"unmanned underwater vehicles"-);
- Medical Intelligent Diagnostic Systems;
- Legal Intellectual Systems
- social and medical care robots for the sick and the elderly people.

This article is more concerned with the problem of creating control platforms for autonomous mobile vehicles. The fact is that with existing operating systems and computer platforms, the mass creation of autonomous mobile devices, primarily ground and airborne basing, is a socially dangerous phenomenon, both because of the unreliability of existing operating systems, and because of the vulnerability of existing operating systems from a variety of "malicious" software: "viruses", "worms", “trojans”, etc.

The specific problem of autonomous vehicles is that in the period of their productive functioning they are beyond of the possibility of intervention of qualified personnel in the case of "loss of control." Given that these vehicles (UAV or UGV, for example) are gradually becoming heavier and heavier, the danger of incorrect operation of these devices is increasing.

It is also possible to imagine only in a "nightmare", as some terrorist / blackmailer or just not quite a normal person is injecting a virus program into an unmanned vehicle that will travel "fifteen meters to the right of the road" in a certain area of geographical coordinates… - on the pavement ... Prior to reaching a certain territory, the virus, for example, may not manifest itself ... Further, you can supplement the "horror" by presenting a typical spread of the virus…

With the development of operating systems, history is actually quite "muddy" ... Well, judge for yourself: in fact, the technology of "modern" operating systems is the approaches to software design of the 60s-70s ... And nothing significant in terms of operating system technology since then has not changed!

At present, software applications work primarily on the platforms that were created more than 50 years ago and this age is beginning to be clearly felt ... These platforms represent an extensive collection of software operating systems, programming languages, compilers, libraries, run-time systems, middleware, etc., as well as equipment that allows these programs to run.

On the one hand, these platforms have been a huge success both in financial and in practice. In the financial field, this was resulted in the creation of the software industry, which by now only in the section of "packaged software" has demonstrated revenue of about $ 200 billion dollars. The

practical result was revolutionary innovations in the form of the Internet, mobile communications, etc.

On the other hand, these platforms leave much to be desired in terms of expectations and requirements of the majority of users for reliability, stability and security. From the financial point of view, it is also obvious that nothing more or less adequate to the revenue received in the development of operating systems was invested! All of the projects listed below were mainly carried out by university teams and rarely spent more than $ 2-3 million per project. The total costs for all two dozen projects are unlikely to exceed $ 50 million in 30 years! And this is with total income of two hundred billion ...

Yes, there were projects of “giants” - Microsoft Research and IBM, - but they can hardly be seen serious industrial investments, although they, in aggregate, can double the above financial estimates. In general, based on the following and proper consideration of this issue, the reader can form his own assessment of the problems and prospects of the industry in this direction.

The issue

The existing problem is that our current platforms did not go far beyond computer architectures, operating systems and programming languages of the 1960s and 1970s. The computer environment of that period was very different from today's: (1) computers were extremely limited in speed and memory capacity; (2) is used only by a relatively small group of technically literate and non-malicious users; 3) rarely connect to the network or to the controlled objects.

Today none of these conditions is true!

But modern computer architectures, operating systems and programming languages did not evolve sufficiently to provide fundamental changes in computers and their use. Criticalness of the situation also is increasing by the increasing of number of applications of artificial intelligence implemented and the expectation of the mass emergence of mobile autonomous vehicles in all kind of environment: land, sea and air. Against the background of their appearance, the task of computer security is directly transformed into a ordinary security problem: safety of life, health and free activity of people.

“Mass” operating systems today (Windows, UNIX / LINUX (and mobile operating systems based on LINUX), MacOS) are several million lines of code for a monolithic kernel, in which, according to statistics, several dozen errors (from 2 to 75 according to statistics) for every 1000 lines of source code in (depending on the size of the software module) [Basili V, ... "Software Errors and Complexity: an Empirical Investigation", Commun. Of the ACM, vol.27, Jan. 1984, pp. 42-52; Ostrand T., ... "The distribution of faults in a large industrial software system", Proc. Int'l Symp. On Software Testing and Analysis, ACM, 2002, pp. 55-64]. Using even a very optimistic forecast, the number of errors in LINUX code can be estimated at 15,000, in Windows-xx - at least twice as much!

In addition to errors, the factor of viruses and other "malicious" software becomes more and more critical!

Alan Turing with his "endless punched tape" seems to have done a colossal diversion “purposely or unwittingly”! In the sense that "commands are interleaved with data" ... that directly had been reflected in the command system of CISC-processors.

Well, let's not offend the “master” - let's just say: "he relied heavily on the predominantly positive thinking of computer users…" which at that time really belonged to the intellectual elite.

As a result, commands in operating systems are "mixed" with data, which makes it so easy to "inject" computer viruses into the software “en masse” through the global network! Willy-nilly you will think that "if computer viruses exist, it means that someone needs it"…

What does the author say? At least, that if it were originally intended and projected that the commands and data should be located in different memory address spaces (which makes the task technically not so complicated), then the introduction of a computer virus "through the network" or "from a floppy disk" would be absolutely impossible! If this simple technical solution was originally adopted, then the computer world would become different ...

The number of "windowed" viruses is measured by "tens / hundreds" of thousands, in the unix / linux environment there are about of one or two thousand. The reason for this quantitative imbalance in the threats between the main operating systems lies not only in the mass distribution. However, a detailed consideration of this issue is too special and will lead us away from the topic of the article, which is more of an overview and problematic nature.

It is the expected massive appearance on the roads of unmanned vehicles that will catalyze a "new wave" in the creation of next-generation operating systems.

Focusing on the problems of control of autonomous vehicles, it is necessary to emphasize that the requirements for computer platforms that serve mass computers and control systems of autonomous vehicles are very different! It will lead to sufficiently independent development and significant differentiation of these systems in perspective as it was in aviation for example.

Autonomous vehicle, in contrast to a typical "computer user":

- do not "hang out" in the global network;
- multitasking is assumed, but only in some priority and hierarchy;
- text editors, spreadsheets, e-mail, any GUIs - all this beyond the task of autonomous control of mobile vehicles. In particular, this leads to the uselessness of the huge number of drivers that now serve I/O tasks and included in the operating system kernels.

As for the drivers, they were the cause of the "crash", for example, Windows XP - in 85% of cases! In 2003, Windows XP contained more than 35,000 different drivers in more than 120,000 versions – most of them are nothing for autonomous vehicles.

Research Projects

The "Nooks" project, implemented at the University of Washington in 2002-2004, was dedicated specifically to the problem of "driver errors". As a result, the same name subsystem was designed and implemented to improve the reliability of the operating system by isolating the core of the OS from the consequences of driver errors.

To achieve this, the drivers "were isolated" inside the kernel address space in the form of some domains. It was used the software and hardware to prevent the possibility of changing the executable code of the kernel in case of an error in the driver, monitored the use of kernel resources by the drivers and restarted them in the event of errors, using the original, i.e. "right", unaffected, version of the executive code. Testing with the Linux 2.4.18. showed that in 99% of the simulated situations of fatal driver errors, which in the standard operating system were guaranteed to crash, Nooks restored correct functioning; in the case of "non-fatal" errors - the result was more modest - 55% of cases were detected and restored. This recovery became possible due to the introduction of "shadow drivers" - in the form of "agents in the kernel", which 1) hid driver errors from "clients", including the operating system and applications; 2) transparently restored drivers back to working condition.

In 1995-1997 in the project DROPS project, the research group of the University of Karlsruhe (Germany) developed a core called "L4" ("predecessor" - "L3"), which was intended for managing Linux operating systems, which were launched within the framework of "virtual machines". Each such operating system within its virtual machine could perform a variety of functions in a safe mode, without jeopardizing the processes managed by other virtual machines, including the encapsulation of drivers in this way. It is this project that still "lives" on the Internet

[http://os.inf.tu-dresden.de/L4/LinuxOnL4/]. Performance measurements showed that the "fee" for a significant increase in security was quite low - 3-8%.

The approaches described above concerned the expansion of the functionality of existing operating systems. The following projects focused on the essence of the problem: it is impossible to have reliable code and reliable functioning in a huge monolithic program containing millions of lines of code than are the main operating systems at the moment!

The approach to developing of operating systems based on the “microkernel approach” is that the *kernel* contains only a few thousand lines of source code, and all *user processes*, like most even *system processes*, are launched in a completely isolated address environment running their own *software servers*. An essential condition is that all the *kernel data structures* are static.

From the standard functions of the core of the "normal" operating system, which include: scheduling, file system, network protocols, device drivers, memory management, etc. - in the microkernel typically focuses: memory addresses management, inter-processes communication and basic queue management.

Actually, the idea of "microkernels" appeared in the 80's. A number of academic projects were implemented, mostly related, for obvious reasons, to the UNIX operating system, these are the projects: Amoeba, Choices, Ra, V. Some projects even had a "commercial" continuation, such as "Chorus", L3, Mach. The Mach project can be identified as the most revealing and important for the organization of further work.

The Mach project used "paging". The application tasks were encapsulated in the address spaces of the “pages” in the virtual memory. If an *application user program error* was detected on the page, the processing the error was assigned to a special program that also belonged to the user level and did not belong to the kernel. This task restored some correct image of the erroneous page and passed it to the kernel for placement in memory.

In the project Exokernel, the kernel of the operating system in its common configuration was moved beyond privileged operations at all.

The project Spin worked out the principles of using a safe subset of the algorithmic language with the appropriate compiler for designing the kernel of the operating system.

At the University of Amsterdam, in 1985-87, 2004-2006, under the guidance of Professor Andrew S. Tanenbaum, the "micronuclear" operating systems MINIX-MINIX-3 were developed. The project in the status of *open source* "lives" to the present (!), and can be downloaded and installed, - the author was convinced of this himself (www.minix3.org). There are articles and speeches at conferences dated 2015. The result of the projects was the "microkernel" operating system MINIX-3, which is POSIX-compatible with other operating systems of the UNIX family.

The dependability of the MINIX-3 OS derives from several sources. First, it's only 4000 lines of kernel source code (in C mostly). Based on the justified “standard” of about 6 errors per 1000 lines of code, you can expect to have only 24 errors in the kernel (compared to about 15,000 errors in LINUX and even more in Windows).

The small size of the source code allows both manual and instrumental deep checking. All drivers except the *timer* work outside the kernel, which significantly reduces the number of associated "crashes" of the system (up to 85% for monolithic cores). All kernel data structures are static. All drivers and user servers work in their own address spaces, which they cannot exit from (with the exception of requests to the kernel, which the kernel checks for validity).

The kernel uses the principle of dividing memory into commands and data, which excludes the possibility of external implementation of the "malicious code." The presence of the "Reincarnation Server" and the mechanism for the safe removal of erroneous processes and their restart provides additional opportunities for the "survival" of the system when errors occur.

We have to pay for everything ... For many years in the 80s-90s, the approach to design operating systems based on microkernels was criticized for significant performance losses. In the

case of MINIX-3, the performance loss of drivers outside the kernel in the "user" status was 10%. The drivers are restored in 4 seconds. (Using the Athlon processor, 2.2, Ghz). The MINIX-3 OS supports up to 400 standard UNIX applications, including the X Window System, two C compilers, a series of editors, a full TCP / IP stack with "BSD sockets" support, as well as standard "shell", a file system and other UNIX utilities.

The reason for such attention to the OS of MINIX-3 is that the project is open, accessible and working!

If in the period of 2004-2012 there were several such projects in open access, including the projects of the "world grandees" of Microsoft and IBM, which are discussed below. After 2011-2012, all such projects disappeared from *open source*…., including the Microsoft project *Singularity* and the IBM project - "K42", which were also "open" on the "start"… Now references to these projects are following by "insufficient access rights" ... It can be assumed that the most serious projects of the world leaders in the industry are actually "live", but were moved under firewalls of corporate portals.

In 2005 Microsoft started a project called "Singularity" [“An Overview of the Singularity Project”, Microsoft Research Technical Report MSR-TR-2005-135. - http://research.microsoft.com/os/singularity].

"Singularity" is a research project of *Microsoft Research* that started with the question: what would the software platform look like if it were designed "from scratch" with the main purpose of dependability, rather than with a more general goal of performance?

A key aspect of Singularity is an extension model based on software-isolated processes (SIPs) that encapsulate parts of an application or system, provide information hiding, isolation of errors to create fault tolerance, and reliable interfaces. SIP is used throughout the operating system and application software. Engineers at *Microsoft Research* suggest that building a system based on this abstraction will lead to the creation of more reliable software.

SIPs are the OS processes on Singularity. All code outside the kernel executes in a SIP. SIPs differ from conventional operating system processes in a number of ways:

- SIPs are closed object spaces, not address spaces; two Singularity processes cannot simultaneously access an object; communications between processes transfers exclusive ownership of data;
- SIPs are closed code spaces; a process cannot dynamically load or generate code;
- SIPs do not rely on memory management hardware for isolation. Multiple SIPs can reside in a physical or virtual address space;
- Communications between SIPs is through bidirectional, strongly typed, higher-order channels. A channel specifies its communications protocol as well as the values transferred, and both aspects are verified;
- SIPs are inexpensive to create and communication between SIPs incurs low overhead; low cost makes it practical to use SIPs as a fine-grain isolation and extension mechanism;
- SIPs are created and terminated by the operating system, so that on termination, a SIP’s resources can be efficiently reclaimed;
- SIPs executed independently, even to the extent of having different data layouts, run-time systems, and garbage collectors.

Singularity was written primarily on the basis of "safe language" Sing # (based on C#). On the website of *Microsoft Research*, Singularity was the last mentioned in 2007 [“An Overview of the Singularity Project” by Galen Hunt, and others, Microsoft Research Technical Report MSR-TR-2005-135; https://www.microsoft.com/en-us/research/wp-content/uploads/2005/10/tr-2005-135.pdf].

Also worth noting is the Erlang / OTP project (http://erlang.org), originally developed by the *Ericsson Computer Science Laboratory* to create high-availability "soft realtime" systems for telecommunications routers. The project itself includes: *Erlang language interpreter*, compiler, the

communication protocol between servers, the CORBA query broker, the distributed database server, analytical tools and libraries, i.e. It, like the Singularity project, has basically a "linguistic approach". The project also developed a methodology for reliable design.

The main idea of the project was to assume that the programs making up the system will necessarily contain errors, the consequences of which must be overcome by restarting processes, by standardization of asynchronous communication messages between processes, by automatic garbage collection in each process, etc.

An important part of ERLANG is the support for fault recovery. Fault tolerance is provided by organizing the processes of ERLANG-applications in the form of *tree structures*. In these structures, "parental processes" track failures of their "children" and are responsible for their restart. Libraries support the creation of such structures during the initialization of the system.

The system was designed initially as multithreaded with support for simultaneous execution of tens of thousands of processes, which is especially important for telecommunication systems in the execution of the "process-user". Erlang / OTP was tested on a series of Ericsson products, for example, AXD301 (ATM-switch). The project is currently supported on the *open source* by a dedicated division of Ericsson.

IBM Research developed the project of the operating system "K42" from 1996 to 2006 in cooperation with a number of universities. The fundamental solution was to use object-oriented programming and C++ language when designing all the components of the system. The second basic solution was the abandonment of centralized code paths and data structures. The operating system was focused on supporting processor clusters with 64-bit addressing. Otherwise, the system inherited the general characteristics of the "microkernel" design approach: moving the kernel functionality to "user servers" and placing these servers in their own address spaces.

The K42 kernel code is available (https://github.com/jimix/k42). But the author did not attempt to launch it. Some later "echoes" of the project can be found, but it seems that they are more indirect.

Thus, the main basis for the development of *reliable computer operating systems* is concentrated in the "first wave" projects, these are: Chorus, Nooks, SYNTHESIS, MACH3, Sprite, Synthesis, Peace, Amoeba, Clouds, Spring, Apertos, Choices, Opal, VINO, Plan9, Exokernel , SPIN, Rialto, Paramecium, Nemesis, Scout, Tornado, Eros.

The "second wave" is: K42, Flux, Singularity, Asbestos, CuriOS, MINIX-3. "The second wave" did not give a proved productive result, only the research experience gave. "The second wave" was completed by the beginning of the "crisis" of 2008. The "third" one has not yet begun ...

Requirements for "Operating System of Artificial Intelligence"

The further presentation is the position of the author and naturally does not pretend to be complete and true in such a new and absolutely non-standardized sphere as "artificial intelligence".

At the same time, the author is less interested in "internet-based smart applications". They will be based on the technology and those operating systems that will evolve in the traditional way.

The situation is different with autonomous vehicles. In fact, their "intellectualization" turns them into some if not "individuals", then "entities" that are able to make their own decisions.

Proceeding from this immediately the question arises: "Can such objects have a "Reset" button? - A person whom we recognized as "normal" has no "reinstallation"…

The author believes that the intellectual apparatus should not have any "reset"! As fundamental principle! Maybe it sounds unusual? The author had to deal with such a situation with complex missile control system in his practice earlier in 80-s.

«Crash-Only Software»

Operating systems that do not have a "reset" are referred to as "Crash-Only Software".

Assuming that for the *intellectual entity* the *continuity of conscious functioning* is also very important, as for living beings, the author tends to include requirements for "crash-only software" in the composition of the requirements for AI-OS.

Crash-only software means:

- strong modularity with relatively impermeable component boundaries;
- timeout-based communication;
- lease-based resource allocation;
- self-describing requests that carry a time-to-live and information on whether they are idempotent;
- transactional principle also should be used when adoptable.

Dependability & security

Dependability and Security are the main requirements for AI-OS. It is clear that these concepts are very closely interrelated. The both concepts closely tied with:

- availability;
- reliability;
- safety;
- confidentiality;
- integrity;
- maintainability.

From security reason all the above requirements mean "authorized actions". We need to remember: faults, error, failures - and process their consequences. These activities relate to all phases of the life cycle of the system: development, production and productive functioning.

To the above, we can add that in the case of autonomous vehicles, it is also necessary to take into account "real time", at least in a certain time range, assuming that AI-OS will hierarchically interact with the system of "hard real time" (in the case of UAV, and some UGV) or "soft real time" (in the case of terrestrial and marine systems).

Instead of conclusion

In the modern world, there are not so many technical areas where specialists could manifest themselves to the same extent as in the case of the development of applications of artificial intelligence and especially with the operating systems on which these applications should be based!

This short overview shows that at present there are practically no prerequisites for "monopolization" of this sphere by even "industry leaders" in the form of: Microsoft, IBM, GOOGLE and others. Let’s go.

Some authors articles are represented in https://www.researchgate.net/profile/Sergey_Korneev2