International Journal of Computer Science & Applications
Vol. 4, No. 1, pp. 33 - 38

Experimental dependability evaluation of memory manager in the real-time operating system

Pawel Pisarczyk
Institute of Computer Science
Warsaw University of Technology
15/19 Nowowiejska
Warsaw, Poland
P.Pisarczyk@ii.pw.edu.pl

ABSTRACT
The paper presents results of experimental dependability evaluation of the Phoenix-RTOS operating system. Experiments are conducted using a self-developed testing environment and a kernel fault injector. Dependability evaluation is the last stage of a system development process. Results will be used in the future research to propose enhanced error detection techniques embedded into the operating system kernel.

Categories and Subject Descriptors
D.4.5 [**Operating Systems**]: Reliability—*dependability evaluation*

General Terms
Reliability

Keywords
operating system dependability evaluation

1. INTRODUCTION
Dependability evaluation of operating systems is the focus of many research jobs. Developed methodologies are based mainly on software-fault injection technique. In papers [2], [8] new methods of dependability evaluation for microkernel-based operating systems are proposed. In paper [5] experimental environment for operating system with monolithic kernel and results of experimental dependability evaluation for Linux kernel are presented. The paper [6] presents model of error propagation between device drivers and operating system services. New measures characterizing error propagation (exposure and diffusion) have been introduced. Results of experimental estimation of introduced measures have been presented. In parallel to performing the presented works focused on dependability evaluation of popular operating systems, new architectures of mission critical operating systems have been proposed. The main goal of these architectures is to maximize efficiency of error detection and to eliminate fail silence violation. In paper [3], [1] the time-triggered architecture and results of experimental dependability evaluation for the architecture implementation have been presented. The main drawback of presented architecture is the requirement of specific application decomposition. It significantly decreases number of possible system implementations and practically disables application porting.

Presented works, related to dependability evaluation of popular operating systems, are focused only on selected modules and system services provided for applications. Proposed operating system models do not include functional dependencies between subsystems and error propagation is evaluated using simple metrics. Presented results have no influence on development of new methods and algorithms for error detection and propagation barriers embedded into operating systems. New algorithms for kernel-level error detection and fault tolerance have been only proposed for MARS operating system.

Phoenix-RTOS operating system developed by the author is an open-source real-time operating system for widely used event-driven applications. System provides all services (like resource protection, threads and processes, IPC, IP networking and graphical interface) required by industrial applications. Development process of every subsystem has been extended with dependability evaluation experiments. Results of these experiments allow to propose new error detection and fault tolerance algorithms integrated with the operating system kernel. Detailed error propagation analysis will be used to propose propagation barriers.

This paper presents results of experimental dependability evaluation of the memory management subsystem of Phoenix-RTOS real-time kernel Section 2 presents briefly Phoenix-RTOS operating system and memory management. Section 3 shows the used error model and introduced measures characterizing the system behavior after the fault injection. Section 4 describes experimental setup and developed fault injector embedded into the operating system kernel. Section 5 presents the analysis of results of conducted experiments. Section 6 presents conclusions.

2. PHOENIX-RTOS
Phoenix-RTOS operating system has been developed in the Institute of Computer Science in Warsaw University of Tech-

Table 1: **Virtual memory subsystem functions**

Function	Byte size	Instructions	Part
`pmap_enter`	610	185	V_{hal}
`vm_pageAlloc`	953	291	V_{pg}
`vm_pageFree`	249	85	V_{pg}
`vm_kmap`	521	164	V_{kmap}
`vm_kunmap`	809	241	V_{kmap}
`vm_kmalloc`	908	285	$V_{kmalloc}$
`vm_kfree`	655	216	$V_{kmalloc}$
`vm_map`	-	-	V_{map}
`vm_unmap`	-	-	V_{map}

nology in years 2005-2006 as the successor of the prototype operating system Phoenix [7]. System has been developed as a result of work which goal was to prepare the prototype for real industrial applications. Phoenix-RTOS is the time sharing real-time operating system intended for embedded systems. System executes threads located in separated address spaces associated with processes. The system has been implemented for IA32 based PC computers used in embedded applications. Ports for ARM7 and PowerPC architectures are available. Phoenix-RTOS has been developed for real industrial needs which is the health monitoring appliance of one of a Polish medical equipment vendor. This paper is not focused on Phoenix-RTOS architecture therefore only the memory subsystem will be briefly presented.

Virtual memory management subsystem consists of five parts: hardware abstraction layer V_{hal}, page allocator V_{pg}, kernel virtual space allocator V_{kmap}, kernel heap allocator V_{kmal} and process virtual space allocator V_{map}. Memory manager functions are presented in the table 1.

Hardware abstraction layer V_{hal} implements functions managing the virtual memory mappings and operating on MMU data structures.

Page allocator manages available physical memory using a page map structure. The map describes the accessible memory regions dividing it into segments consisting of page groups. Possible segment size values correspond to succeeding powers of 2 starting from 0. Each entry of the map points to a list of segments with a given size. The first entry points to the list of segments consisting of one page only. The Nth entry points to the list of segments of 2^{N-1} pages. After the system initialization the memory is divided into segments with maximum sizes. During the allocation process, allocating function (`vm_pageAlloc`) looks for the first segment with a size greater or equal than requested one. When the segment size is greater than the requested size it is divided into two smaller segments. One of them is returned to the map. This process repeats until the obtained segment size value is closest the requested value. When segment is released (using function `vm_pageFree`) the kernel merges it with a succeeded or proceeded segment on the list. Described memory allocator (named buddy allocator) is typical for the most operating systems and has been presented in [9].

The kernel virtual space allocator V_{kmap} manages kernel virtual space V_K. An allocated segment is not accessible for the kernel and it should be mapped into the kernel address space if the kernel intends to use it. Mapping function `vm_kmap` looks for the first virtual space region at which a segment can be mapped. After mapping, the kernel marks this region as used. The last stage of mapping is the invocation of a hardware dependent `pmap_enter` function which fills structures used by MMU. Opposite function `vm_kunmap` removes segment from kernel virtual space, marks the region as free and inaccessible using `pmap_enter` function. Kernel space allocation is not a common mechanism in operating systems because many of them assume that the virtual address space is much greater than the physical memory size and the kernel space can correspond to the physical memory.

The kernel heap allocator is used to manage the allocation of memory chunks which size is less than the size of the page (minimal size of the segment). The main data structure of this allocator is the `size[]` array. Array entries point to segments divided into smaller chunks with specified size. The relation between the entry index (i) and the chunk size is given by the function 2^{4+i}. When the table entry is empty (no chunks are available) the new segment is allocated, mapped into the kernel address spaces and divided into chunks. Heap allocator consists of allocating function `vm_kmalloc` and releasing function `vm_kfree`. Each of theses function depends on the segment allocation and mapping functions.

3. ERROR MODEL AND MEASURES

The error model assumed in this paper is an error having impact on correct execution of an instruction by processor. This model is similar to the model presented in [5]. Single-bit errors are injected to have impact on instructions of functions belonging to the memory manager. Each injected error can be characterized by following attributes:

- *trigger* - An error is injected when instruction is fetched by processor.
- *location* - Errors are injected to code of selected functions at random locations. Errors are injected to any byte of an instruction.
- *type* - One single-bit (bit-flip) error per byte of an instruction is injected.
- *duration* - Two types of errors are considered - transient errors which are removed after the erroneous instruction execution and persistent errors which exist during the experiment execution time. Type of error is selected randomly, but the probability of selecting persistent error is 25%.

Outcomes from fault injection experiments are classified according to observable kernel states presented in table 2.

To characterize what impact errors injected into the function f have on the operating system the special measure, called operating system resistance R, was introduced. Resistance R for function f is the vector of percentage shares of every observable state during the injection campaign for this function f (1).

Table 2: Operating system states observable after the fault injection

State	Description
S_{err}	Injected fault has been detected using software mechanisms embedded into the kernel.
S_{exc}	Injected fault has been detected using CPU exception.
S_{hang}	Kernel hang ups and goes to a non-operational state. Restart is required.
S_{fs}	The corrupted instruction has been executed and no visible impact on the system has been observed either by the workload application and operating system kernel.
S_{fsv}	The corrupted instruction has been executed and requests generated by workload application are not properly serviced (i.e. two succeeded memory allocations returns the same heap chunk). This is the fail silence violation state.
S_{tm}	Injected error is not executed in the time assumed for the experiment.

$$R(f) = [p_{S_{err}}, p_{S_{exc}}, p_{S_{hang}}, p_{S_{fs}}, p_{S_{fsv}}, p_{S_{tm}}] \quad (1)$$

Other metrics characterizing error detection timing were introduced.

- average error activation time T_{act} - the time elapsed from the beginning of the experiment to the moment of execution of the corrupted instruction;
- standard deviation of the average error activation time σT_{act};
- average error detection time T_{det} - the time elapsed from the beginning of the experiment to the moment of error detection;
- standard deviation of average error detection time σT_{det}.

Error propagation is characterized for each function using a propagation ratio measure *Prop*(f) (2).

$$Prop(f) = \frac{n_{prop}}{n_{act}} \quad (2)$$

Propagation ratio *Prop* for the function f is the fraction of the number of errors detected outside the function (n_{prop}) to the number of all errors activated in this function (n_{act}). Value 0 of propagation ratio for function f indicates that all faults are detected during the function execution. Value 1 indicates that all faults are detected after passing control to other functions or to non-code segments.

Experiments has been performed for workload generated by thread calling routines used for validation of implementation of a memory management subsystem. These routines activate all functions of the memory manager.

4. EXPERIMENTAL SETUP

Dependability evaluation of the operating system requires different methodology than the dependability evaluation of applications running under the control of an operating system. Application running in presence of faults injected into its address space has no impact on operating system stability and all faulty conditions can be properly handled using debugging subsystem routines. Methodologies and tools developed for the dependability evaluation of user applications have been presented in [4].

Dependability evaluation of an operating system requires the existence of separate computer which controls fault execution and gathers outcomes of each experiment. This computer is called an experiment controller. Errors are injected by the fault injector embedded into the operating system kernel which communicates with the controller. Each fault injection corresponds to one experiment. Many experiments associated with a selected function constitute a campaign. The experiment process is presented below:

- **Select the location of the fault and other fault attributes**
- **Set the instruction breakpoint address**

  Before setting the breakpoint instruction stream is disassembled to establish the location of the instruction containing the error location. Disassembled code is sent to the experiment controller. This step of the injection experiment is called the 'inject' phase.
- **Wait for the instruction execution**

  When a selected instruction is fetched by the processor injector modifies the selected memory location and enables the step mode of instruction execution. This step is called the 'preinject' phase. Disassembled instruction stream and CPU context are sent to the controller.
- **Execute the corrupted instruction**

  After the execution of the corrupted instruction the experiment enters into the 'postinject' phase. In this phase the injector sends CPU context and disassembled instruction stream starting from address of the program counter and restores the original value of the memory byte if the transient error is simulated.
- **Wait for the fault manifestation**

  An experiment enters into the last stage which is the fault manifestation.
- **Restart the system using watchdog**

Outcomes from experiments are stored as files consisting data coming from each experiment state. The example file is presented below.

```
<inject time=12000 addr=c0060c90 bit=1 fault=t>
    <code>
    c0060c8e: movl 10(%ebx),%ebp
    c0060c91: movl %ecx,fffffffc(%esi)
    c0060c94: movl $fe8,%eax
    </code>
</inject>

<preinject time=125000>
    <cpu>
    eax=ffe2de3b ebx=c018e000 ecx=c018e0cc
    edx=c018e0a8 esi=c018e0ac edi=c006f12c
    ebp=0000b13c esp=00000001 eip=c0060c8e
     cs=00000008  ds=00000008  es=00000010
     fs=00000010  gs=00000010  ss=00000010
    err=00000000 eflags=00000186
    </cpu>

    <code>
    c0060c8e: movl 12(%ebx),%ebp
    c0060c91: movl %ecx,fffffffc(%esi)
    c0060c94: movl $fe8,%eax
    c0060c99: movl c(%ebx),%esi
    </code>
</preinject>

<postinject time=125000>
    <cpu>
    eax=ffe2de3b ebx=c018e000 ecx=c018e0cc
    edx=c018e0a8 esi=c018e0ac edi=c006f12c
    ebp=14b40000 esp=00000001 eip=c0060c91
     cs=00000008  ds=00000008  es=00000010
     fs=00000010  gs=00000010  ss=00000010
    err=00000000 eflags=00000086
    </cpu>

    <code>
    c0060c91: movl %ecx,fffffffc(%esi)
    c0060c94: movl $fe8,%eax
    c0060c99: movl c(%ebx),%esi
    c0060c9c: movl %edx,8(%ebx)
    </code>
</postinject>

<timeout time=60460800> </timeout>
```

This file describes the fault injected 12 ms after the system initialization. This injection modifies the second bit of byte belonging to the `movl` instruction at address `0xc0060c8e`. The corrupted instruction has been executed 125 ms after the system initialization. Fault modifies offset of the operand moved to the EBP register. After the instruction execution the EBP register stored incorrect value but this error had no impact on workload and the operating system (fail silence state S_{fs}). After the timeout system was restarted and the next experiment was performed.

5. EXPERIMENTAL RESULTS

During the memory subsystem tests, 1503 experiments have been performed. Outcomes from experiments are divided into groups corresponding persistent and transient errors. For each function of the memory subsystem introduced measures are calculated. Table 3 presents operating system resistances R for persistent errors injected into the memory management functions. Table 4 presents resistances for transient errors.

Table 3: Operating system resistance for persistent error injected into the memory manager

Function	N	$R(f)$
`pmap_enter`	124	[0%, 1%, **46%**, 10%, 0%, 43%]
`vm_pageAlloc`	46	[0%, 0%, 13%, 30%, 0%, **57%**]
`vm_pageFree`	41	[0%, 5%, 27%, **58%**, 0%, 10%]
`vm_kmap`	27	[0%, 4%, **55%**, 4%, 0%, 37%]
`vm_kunmap`	36	[0%, 3%, 33%, 6%, 0%, **58%**]
`vm_kmalloc`	97	[0%, 13%, 3%, 21%, 0%, **63%**]
`vm_kfree`	39	[0%, 10%, 8%, 38%, 0%, **44%**]

Table 4: Operating system resistance for transient error injected into the memory manager

Function	N	$R(f)$
`pmap_enter`	367	[0%, 4%, **50%**, 9%, 0%, 37%]
`vm_pageAlloc`	145	[0%, 2%, 14%, 25%, 0%, **59%**]
`vm_pageFree`	101	[0%, 3%, 31%, **52%**, 0%, 13%]
`vm_kmap`	87	[0%, 2%, 40%, 11%, 0%, **47%**]
`vm_kunmap`	120	[0%, 3%, 22%, 5%, 0%, **70%**]
`vm_kmalloc`	108	[0%, 13%, 1%, 22%, 0%, **63%**]
`vm_kfree`	162	[0%, 7%, 7%, 28%, 0%, **58%**]

The column named N presents the number of experiments conducted during the campaign. The number of experiments is relatively low in comparison to the number of function instructions but the goal of experiments was not to evaluate dependability of the memory manager in details. The goal of the experiments was to present the methodology and framework for a quick dependability estimation of implemented kernel functions using introduced measures. Such estimation allows for selecting functions requiring introduction of propagation barriers or enhancing error detection mechanisms. From development process perspective it makes no sense to evaluate in details dependability of every function because implementation is changing during the system life-cycle.

Results show that many of injected faults are not executed. Most faults are ignored in functions `vm_kmalloc` and `vm_kunmap`. This is a consequence of workload nature. Typical execution of `vm_kmalloc` requires no page allocation and the branch instruction at the function begin omits execution of a large part of code. Similar scenario takes place in `vm_kunmap`. When a kernel address space region is released, the kernel tries to concatenate it with other free regions on the list. If no candidate for concatenation is found, branch instruction moves execution to the end of the function. Least faults are omitted in function `vm_pageFree`. The function code contains many branches but they omit only short parts of the code.

The other important fact should be noted. Results show that software error detection techniques like assertions verifying correctness of input arguments passed to a function are insufficient. No error has been detected using this mechanism. Paper [6] is focused only on injecting faults into input arguments of functions and his author concluded that wrapping is a good technique for prevention of fail silence

Table 5: Detection times for persistent errors injected into the memory manager

Function	T_{act} [μs]	σT_{act} [μs]	T_{det} [μs]	σT_{det} [μs]
`pmap_enter`	114027	3616	111000	0
`vm_pageAlloc`	137379	6869	-	-
`vm_pageFree`	**146455**	5595	**1862500**	2413355
`vm_kmap`	109600	1131	108800	0
`vm_kunmap`	114000	6245	109000	0
`vm_kmalloc`	116829	5537	116692	5212
`vm_kfree`	115200	3211	115200	4460

Table 6: Detection times for transient errors injected into the memory manager

Function	T_{act} [μs]	σT_{act} [μs]	T_{det} [μs]	σT_{det} [μs]
`pmap_enter`	113100	3799	111787	2901
`vm_pageAlloc`	135396	5429	129933	2003
`vm_pageFree`	**145813**	4919	538950	787508
`vm_kmap`	111691	4461	110200	3959
`vm_kunmap`	113556	4086	113800	4275
`vm_kmalloc`	114958	4261	**814100**	2611190
`vm_kfree`	116035	3739	115615	4768

violations and increasing the number of detected errors. Authors assumed that measures introduced by them could be helpful in finding vulnerable operating system parts which potentially demand enhancing argument validation. Presented results show that such methodology is insufficient. The percentage of errors detected using assertion and exceptions is extremely low when faults are injected into the base system component which is crucial for overall system stability.

In presented results there are no outcomes associated with fail silence violation state S_{fsv}. This is very difficult to verify outputs of memory allocation and mapping functions. If the mapping function fails and validating function tries to reference the mapped segment system hangs or processor raises the exception. This is the possible explanation of such situation. This explanation is convinced by the relatively high number of system hangups. Most hangups were observed when faults were injected into mapping function which operate directly on MMU.

Tables 5, 6, 7 present detection times and propagation ratio for each tested function. Injected errors have been detected practically after activation. In some cases only the latency between activation and detection was about 1 s. Highest error propagation was observed for `vm_kmalloc` and `vm_kunmap` functions. This is the consequence of function complexities and dependencies with other memory manager functions.

Table 7: Propagation ratios for persistent and transient errors injected into the memory manager

Function	$Prop_{pers}$	$Prop_{trans}$
`pmap_enter`	0.00000	0.00435
`vm_pageAlloc`	0.00000	0.00000
`vm_pageFree`	0.00000	0.02273
`vm_kmap`	0.00000	0.02174
`vm_kunmap`	0.00000	**0.05714**
`vm_kmalloc`	**0.05556**	0.05128
`vm_kfree`	0.00000	0.01471

Presented results for persistent and transient errors are similar. When fault is injected it is practically manifested after the first execution of corrupted instruction. This conclusion is convinced by the measured short times between error activation end error detection.

6. CONCLUSIONS

The paper presents methodology for dependability evaluation of Phoenix-RTOS real-time operating system developed by author for real application needs. Dependability evaluation is the last stage of implementation of every Phoenix-RTOS function. Outcomes from fault injection experiments allow to determine the kernel resistance (efficiency of error detection mechanisms) for errors introduced into the selected function and the function ability to propagate errors. In opposite to popular real-time operating system, which authors concentrate only on system stability, this is the novel approach which allows to propose new kernel-level error detection mechanisms and propagations barriers. Obtained results related to the memory manager (presented in section 5) allowed to select functions demanding the enhancement of error detection mechanism and introduction of propagation barriers. According to proposed methodology the goal of the introduction of error detection enhancements and error propagation barriers is to obtain (for every VM function) specific error resistance vectors with dominant $p_{S_{exc}}$ and $p_{S_{err}}$ components.

Furthermore obtained results show that techniques based on wrapper, proposed in the paper [6]), are insufficient when errors are injected into the functions of crucial operating system parts.

Obtained results will be used in next stage of Phoenix-RTOS development to propose the enhanced error detection techniques embedded into the operating system kernel. The last stage of the development process will be the dependability evaluation of the health monitoring system based on Phoenix-RTOS.

7. ACKNOWLEDGMENTS

This work was supported by KBN grant 4T11C049 25.

8. REFERENCES

[1] A. Ademaj. A methodology for dependability evaluation of the time-triggered architecture using software implemented fault injection. In *Fourth European Dependabe Computing Conference (EDCC-4) Proceedings*, 2002.

[2] J. Arlat, J. Fabre, M. Rodriguez, and F. Salles. Dependability of cots microkernel-based systems. *IEEE Transaction on Computers*, 51(2), 2002.

[3] E. Fuchs. An evaluation of the error detection mechanisms in mars using software-implemented fault injection. In *2nd European Dependable Computing Conference (EDCC-2) Proceedings*, 1996.

[4] P. Gawkowski. *Analysing an enhancing fault immunity of programs in systems with COTS elments*. Ph.D. thesis, Warsaw University of Technology, 2005.

[5] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *IEEE DSN Proceedings*, 2003.

[6] A. Johansson and N. Suri. Error propagation profiling of operating systems. In *IEEE DSN Proceedings*, 2005.

[7] P. Pisarczyk. Phoenix - realtime kernel for embedded applications. In *Real-Time Systems Conference Proceedings*, 2002.

[8] M. Rodriguez, F. Salles, J. C. Fabre, and J. Arlat. Mafalda: Microkernel assessment by fault injection and design aid. In *3rd European Dependable Computing Conference (EDDC-3) Proceedings*, 1999.

[9] A. S. Tanenbaum. *Modern Operating Systems 2nd Edition*. Prentice-Hall, 2001.