Operating Systems Research

kaneton people

June 3, 2011

Outline

Outline

About

- A course on practical operating systems issues
- Pointers on ongoing research, recent projects
- What you may want to work on, advices on how to do it

Plan

- computing landscape
  - hardware issues
  - software issues
- kernel and OS mechanisms addressing those issues
  - Singularity
  - Helios
  - Barrelfish
  - coping with existing software
- approach when doing practical OS related research
  - summary
  - perspectives
- references
  - research groups
  - projects
  - conferences
  - jobs, internships

Outline

Singularity - 0

Kernel of a full operating system research framework from Microsoft.

- answers the following question: "what would a software plateform look like if it was designed from scratch, with the primary goal of improved depenability and trustwothiness"
- ie. without the legacy of the 70's
- aims to rethink the software stack to avoid unexpected interactions between applications
- unexpected interactions lead to software vulnerabilities and lack of robustness

Singularity - 1

3 key architectureal features

- support for execution
  - software isolated processes (SIPs)
- support for communication
  - contract based channels
- support for system verification
  - manifest based programs

Singularity - 2

SIPs

- single AS, multithreaded software isolated processes
- all communication done via message passing
  - ie. no CreateRemoteXxx()
- lightweight creation costs, management
  - part of the work is done statically
  - no TLB/cache flush as context is switched
  - costs: ~0.5 fork() on Linux
- no code modification
  - plugins run in a separated process with its own security descriptor
  - thus there is no need for a conccurent security model (ie. think about active x...)
  - memory accesses are checked statically

Singularity - 3

Contract based channels

- every communication between SIPs occur through channels
- 2 endpoints typed message queue
  - // todo: channel example
- declarative, implemented in Sing#
  - declarative means less prone to programmer's error, thus improved security
  - implied documentation
- described as a state automata

Singularity - 4

Manifest based programs

- a manifest describes a program and its interactions with the system
  - which resources it needs, which restrictions it has over the system
  - drives the SIPs code placement // todo: show the code
  - channels interconnections
- it describes an expected behavior
- the kernel itself has a manifest // todo: show the code

Singularity - 5

Singularity kernel

- single address space microkernel, SIPs execute outside the kernel
  - but 192 ABI functions
- 90% implemented in Sing#
  - garbage collection, introspection...
- rigorous ABI, no ioctl like calls
- follow the least priviledge principle
  - initially, a process can only manipulate its own state, and create child SIPs
  - according to its manifest, it can be created with existing endpoints
  - it can receive endpoints via a channel
    - this last point allows for statically checked use of a resource
    - for instance, notepad.exe using a Winsock socket...
- the ABI is versionned, allowing to handle backward compatibility
  - no need for awful versionned structures, thus improved security

Singularity - 6

The exchange heap

- the only area where data can be exchanged between SIPs
- ownership of a block is transfered via a channel, then released to the destination SIP
- a heap block is accessible by a single thread at a time
- questions
  - how to implement broadcast like message, data duplication? (since only one dest)
  - what if 2 thread want to communicate data to 2 different endpoints
  - for instance network and fs: the paper says there is only one heap block per SIP at a time

Singularity - 7

Protection domains

- even if SIPs are isolated in software, the Singularity framework allows for hardware protection, ie. address spaces
- called augmented SIPs
- "in case software based verification fails" :)
- protection domains can be selected at will, allowing for different favors of separation schemes to be used (ie. microkernel, monolithic...)

Singularity - 8

Singularity framework / Research Development Kit

- the whole framework is open source and documented
- http://singularity.codeplex.com/
- asplos2008_singularity_rdk_tutorial.pdf

Singularity - 9

Conclusion

Singularity - 10

References

- http://research.microsoft.com/en-us/projects/singularity/

Helios - 0

Issue addressed by Helios

- more and more tasks offloaded to heterogeneous processing units
- programmable hardware (FPGAs), DPSs, GPUs...
- Helios places an application on the unit it is the more affine with

Helios - 1

Sattellite kernels

- a sattellite kernel is instanciated for each processor: cpu, programmable device...
- implemented as a process, each with its own scheduler.
- requierments: timer, interrupt controller, exception handling.
- each kernel contains: scheduler, memory manager.
- the coordinator contains the namespace manager (queryable via remote message passing)

Helios - 2

Affinity model

- processes can specify processes they are bound to
- for instance, network stack bound to network interface
- affinity is used to choose a kernel the process will run on

Helios - 3

Implementation

- modification of the singularity os (singularity rdk). written in Sing#
- Application first compiled into the CIL bytecode, then into the ISA of the particular processor
- if vendor refuses to ship the cil bytecode, it can provide fat binaries

Helios - 4

Discussion

- no automated way is proposed to decide about the affinity
  - 2 processors, 1 with a FPU
  - 2 applications, 1 using the FPU
  - how to choose the more affine unit
- requirements to be considered as a processing unit are too strong
  - GPUs have no programmable timers
  - Cryptographic coprocessors are blackboxes
- application must be redesigned in term of processes
  - break the multithread programming model

Helios - 5

References

- www.sigops.org/sosp/sosp09/papers/nightingale-sosp09.pdf
- research.microsoft.com/en-us/groups/os/singularity/

References

- http://www.barrelfish.org/barrelfish_sosp09.pdf
- http://www.barrelfish.org/
- http://research.microsoft.com/en-us/groups/camsys/default.aspx
- http://www.systems.ethz.ch/

Outline

Coping with existing software

Sometimes not possible to start from scratch to resolve an issue and have to cope with existing software. In this section we see research projects dealing with such issues.

shadowdrivers - 0

- addresses the application state recovery problem
  - if an error occurs, restore the application last valid state
- shadowdrivers is a way to improve operating system stability by hiding to the userland application a driver has crashed
- it adds a transparent local redudancy
- similar to fault tolerance schemes used in distributed systems, but assumes non malicious drivers
- however, the shadow driver is not a full replica, implements only the needed services

shadowdrivers - 1

Discussion

- assume it is possible to detect driver crashes
- either not possible, or too late (ie. kernel has already crashed)
- a well written application has to handle errors. the problem should not be solved at the driver level
- anoter approach: the operating system provides a snapshot/recovery API for the userland applications to snapshot and recover their states in case of crash
  - but not transparent, and this is the goal of such a project
- this approach would be well suited for a stackable model like the NT one
- works even between drivers, since drivers can be seen as clients of one another

nooks

nooks is another project aiming at improving device driver reliability

- defines an interposition layer between drivers (ie. kernel extension) and kernel (linux)
- the framework implements the following layers:
  - isolation
    - memory management is modified so that the kernel is mapped read only in the extensions
    - extension procedure calls (XPC) to transfer control flow from extensions to kernel
  - interposition
    - kernel and extension wrappers (ie. indirect calls as modules are loaded)
    - some wrapper create a shadow copy of the params, which is then synchronized as the XPC returns
  - object tracking
    - instrument kernel functions allocating objects to keep track of all the objects
    - different scheme used to stay performant
    - object garbage collection
  - recovery
    - detects potential bugs: invalid parameters detection, deadlocks, invalid memory deallocation...
    - on bug, reload the extension without the application knowing it

References

- http://www.cs.washington.edu/homes/levy/shadowdrivers.pdf
- http://www.cs.washington.edu/homes/levy/, for more work about OS reliability
- http://nooks.cs.washington.edu/

Outline

Approach - 0

Operating systems research is very large, difficult to have a clear view

- many heterogeneous topics
- lot of research groups, rarely working together
- even in the same lab, it is possible teams do not collaborate (esp. true if not in same country)
- lot of unused (useless?) and redundant work
- emphasis on producing papers
- but there still exists labs working on real things
  - http://qnx.com
  - http://www.ok-labs.com
  - http://ertos.nicta.com.au
  - http://www.xtreemos.eu
  - *check their opportunities page if interested*

Approach - 1

From this course point of view, research must be practical. There should at least be a prototype. Some advices:

- generally, targeting an existing software is better than reimplementing from scratch
- do something useful for NT, it is likely someone form MS will get in touch with you
- for instance: http://dokan-dev.net/en/
- main reason: it is nearly impossible to get ride of backward compatibility(existing user base) and very difficult to cope with legacy (badly designed APIs...)
- the real challenge: implementing new ideas that are backward compatible with previous software (esp. challenging in proprietary software)
- common point between Qemu, VMWare, NaCL: they preserve what already exists
- not to say that working on something new is a bad idea

Approach - 2

To illustrate, here is a system project example

- automatically (ie. not by hand) monitors PCI bus requests on a NT system
  - you have which what register is hit, the access mode... at a given time T
- automatically monitor application behavior
  - you have the function called by an application at T
- correlate the events together
  - you have an automated way to tell what a PCI register is used for
- usage
  - automated PCI driver specifications (couple it with a DSL: http://code.google.com/p/rathaxes/)
  - propietary driver reverse engineering
  - virtualisation: add missing functions, correct bugs...

Outline

Conferences

- http://www.eecg.toronto.edu/ livio/os/, up to date conf refs
- http://usenix.org/, references every system related conf
- http://usenix.org/events/osdi08/, Operating Systems Design and Implementation
- http://www.sigops.org/sosp/sosp09/, Operating Systems Principles
- http://www.usenix.org/events/hotos09/, Hot Operating Systems Topics
- http://eurosys2010.sigops-france.fr/

Research Groups

- http://research.microsoft.com/en-us/groups/camsys/default.aspx
- http://www.systems.ethz.ch/
- http://research.google.com/pubs/Systems.html
- http://www.inf.tu-dresden.de/index.php?node_id=1421&ln=en
- http://groups.csail.mit.edu/carbon/?page_id=39

Companies

- http://www.ok-labs.com/
- http://ertos.nicta.com.au/research