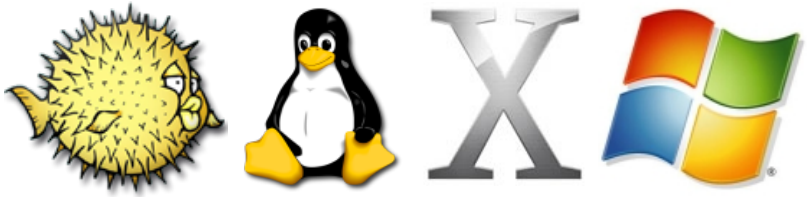
KMux: Kernel Extension at the Hardware Interface

Tareque Hossain
The George Washington University
Department of Computer Science
April 27, 2011

Directed by: Dr. Gabriel Parmer

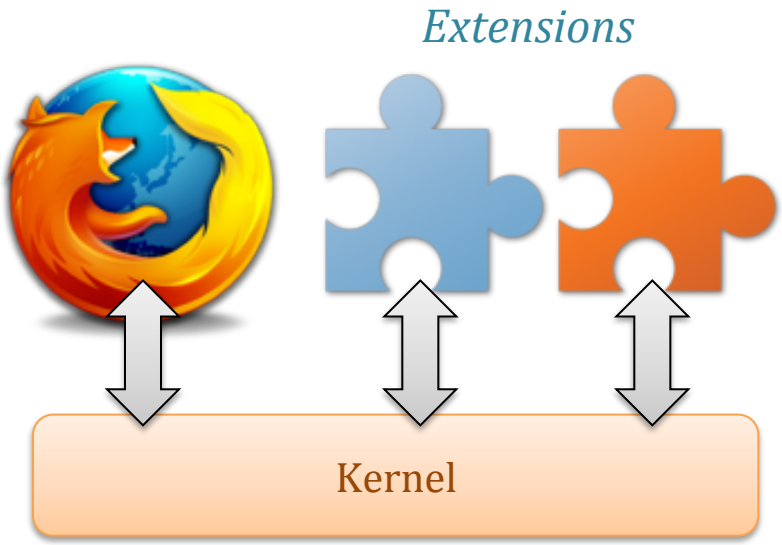
Motivation & Background

- Most commercial operating systems are:
  - Monolithic
  - General purpose



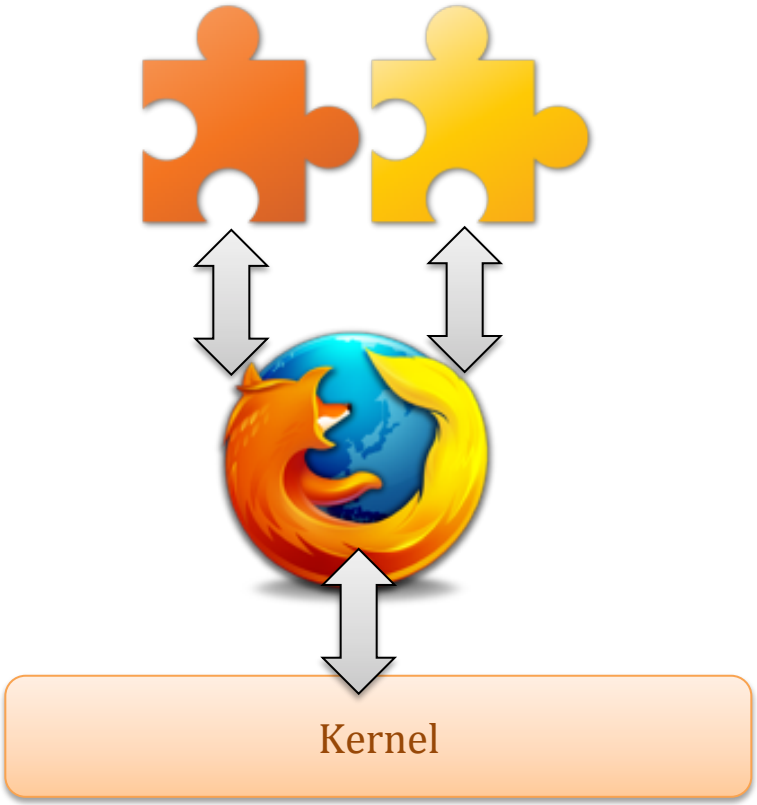
- Kernel interfaces:
  - Tightly coupled into the kernel code
  - Lack customizability/extensibility
  - Generalized to accommodate wide array of services
  - Inefficient for specific needs
    - a system that only runs a web server
- Kernel interfaces should be:
  - Decoupled & easily configurable
  - Easily enhanced, extended or replaced

Motivation



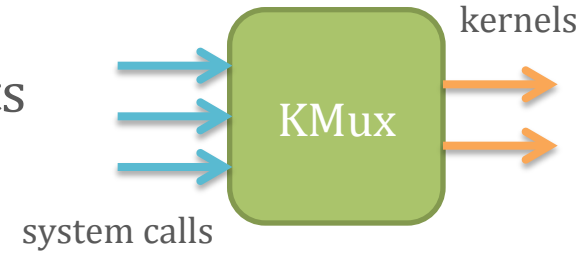
- Present state
  - Browser and plugins independently talk to kernel

- Desired state
  - Browser is in control



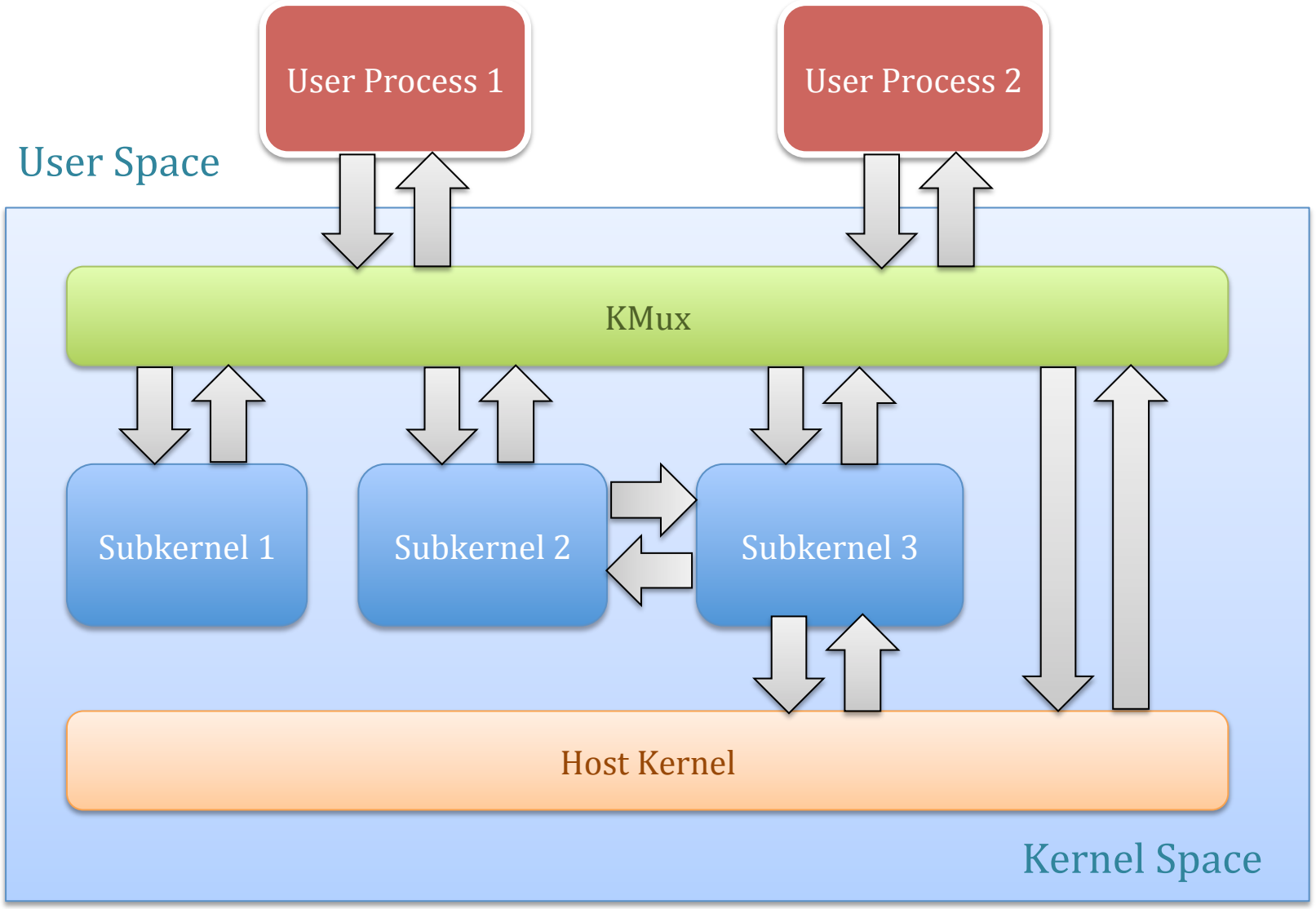
Introducing KMux

- KMux = Kernel Multiplexer
  - Controls user & kernel space communication
  - Input:
    - User-level system call service requests
  - Output path:
    - Lightweight & efficient *kernels* (yes!)
    - Implement, extend, enhance, or restrict kernel interfaces
    - Subkernels



- Challenges: (details after diagram)
  - Control user – kernel communication w/o introducing significant overhead
  - Implement configurable sequence of subkernels to process user requests
  - KMux & subkernel configuration from user space

KMux Overview



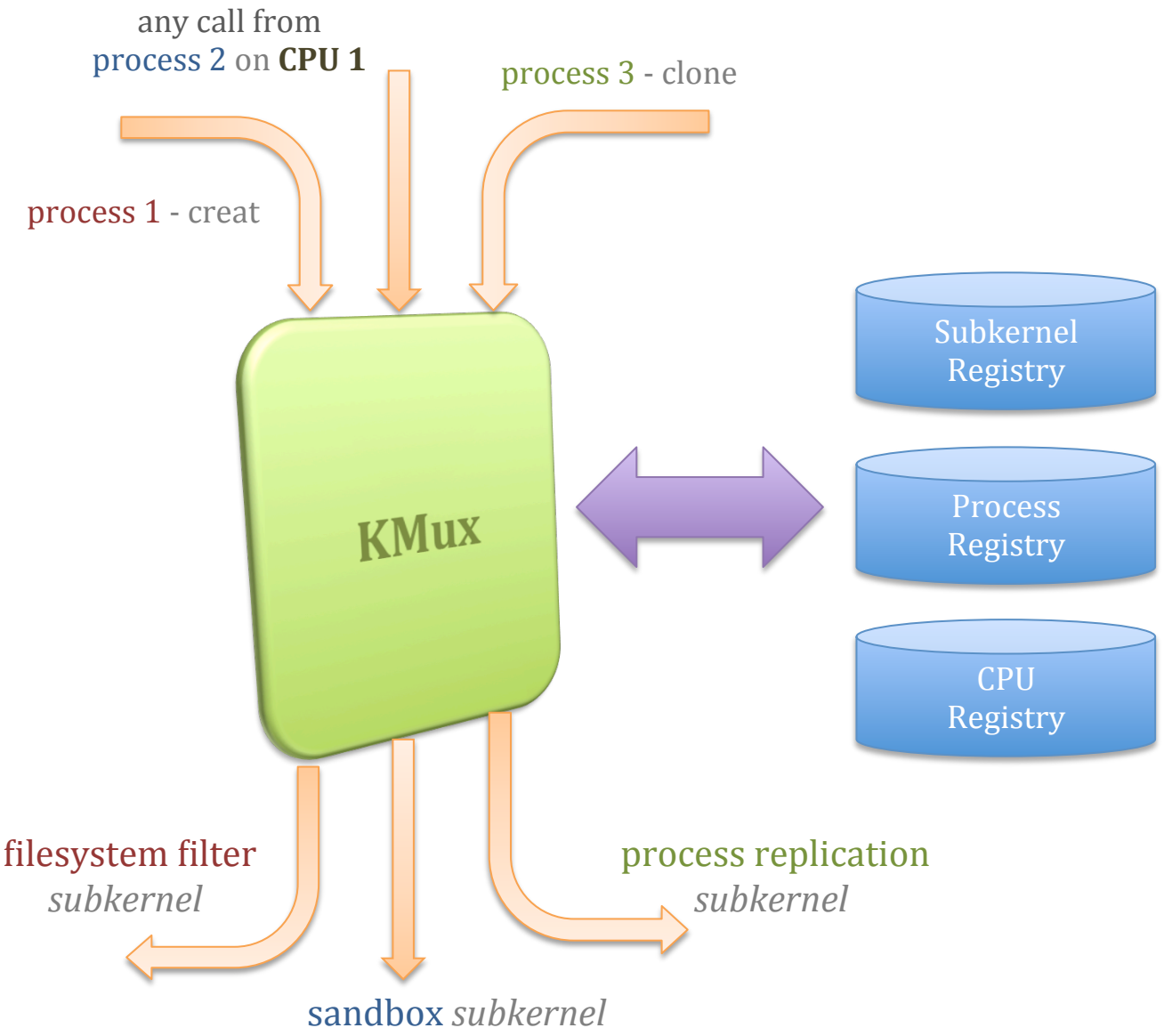
Implementation: Extension at H/W Interface

- How to intercept system calls? Options:
  - POSIX ptrace
    - User space
    - Unacceptable overhead
    - Non-portable across kernel versions (latest standard in 2008)
  - Kernel Modification
    - Non-portable (kernel interfaces in flux)
    - Complicated development
    - Acceptance into “mainline” - large barrier
  - New: Override at the hardware interface
    - Change hardware-defined kernel entry points!
      - Change the default system call hander
        - » Defined in special-purpose registers
    - Extremely fast
    - x86 interfaces are stable

Implementation: Multiplexing

- Kernel entry point overridden to point to KMux routine
  - Invoked every time a system call is executed
  - General purpose registers saved
  - KMux system call handler invoked
- Different system calls come from different processes
  - With different contexts
    - Is this request from a browser extension?
    - Or the browser itself
    - Or a third party application
    - *Or on a specific CPU?!*
- KMux handler analyzes the context
- Delegates control to appropriate subkernel
- *Specific subkernels handle requests for specific apps!*

Multiplexing Overview

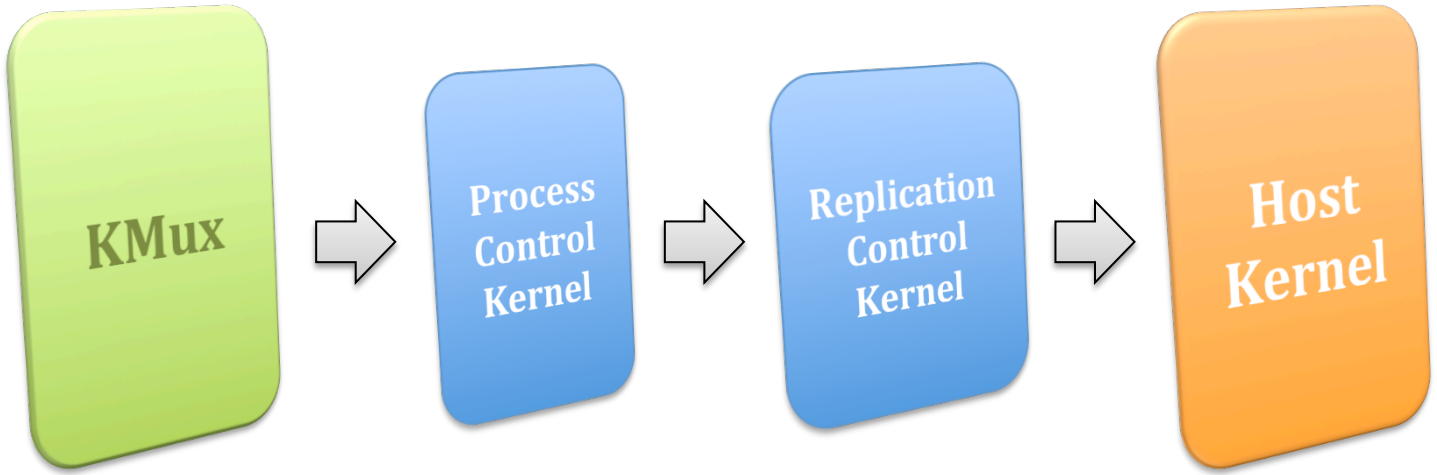


Implementation: Subkernels

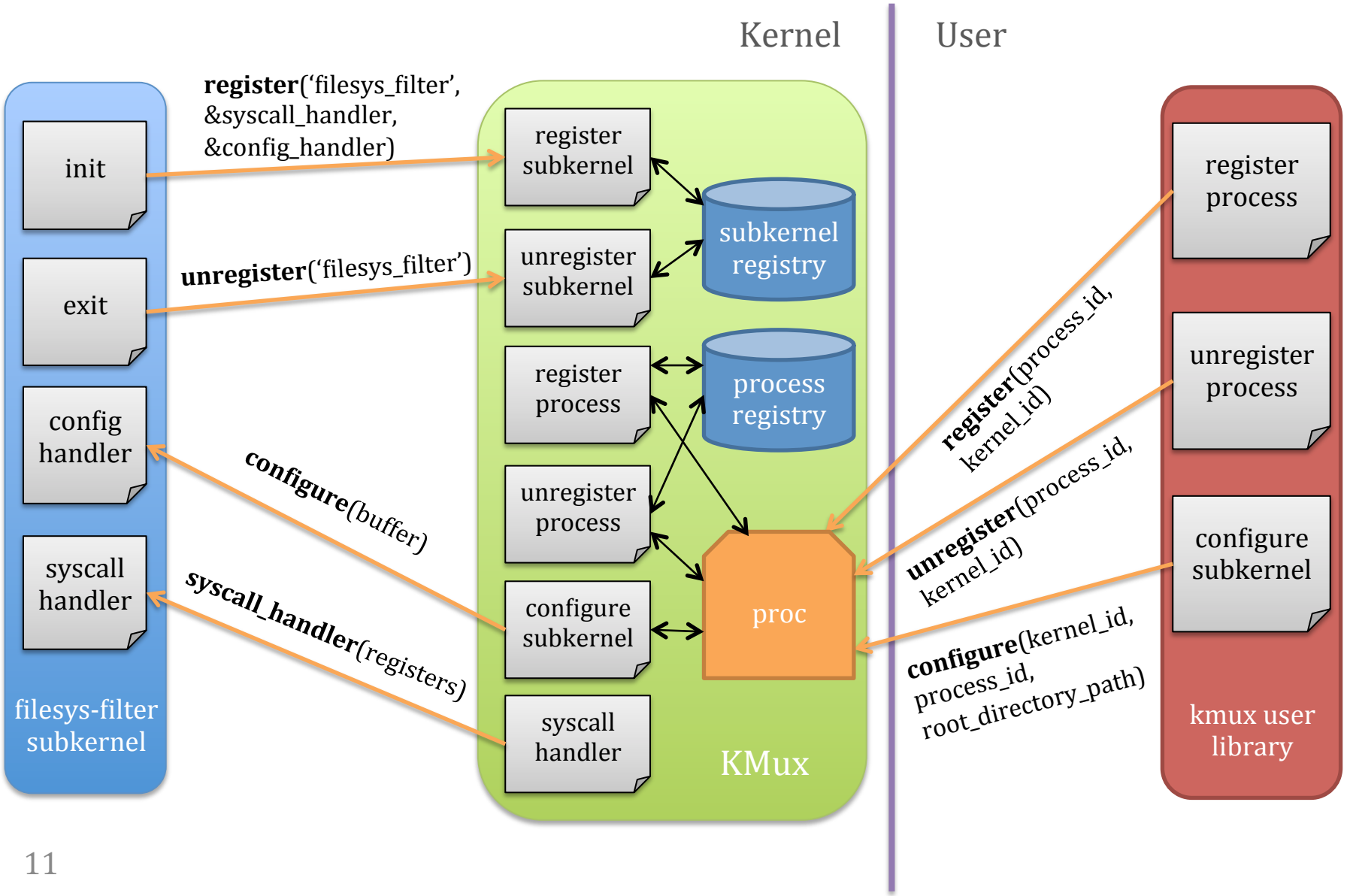
- Host kernel from which KMux assumes control becomes the first subkernel
  - Boot into Linux, *then* activate KMux & other subkernels
- Ranges from simple system call filters to self-sufficient kernels
  - *System can run many kernels concurrently*
- Registers themselves with KMux on initialization
- Examples: Process creation control kernel
  - Monitors process creation and termination
  - Simply monitor these calls, or…
  - Prevent processes from creating new process

Subkernel Chaining

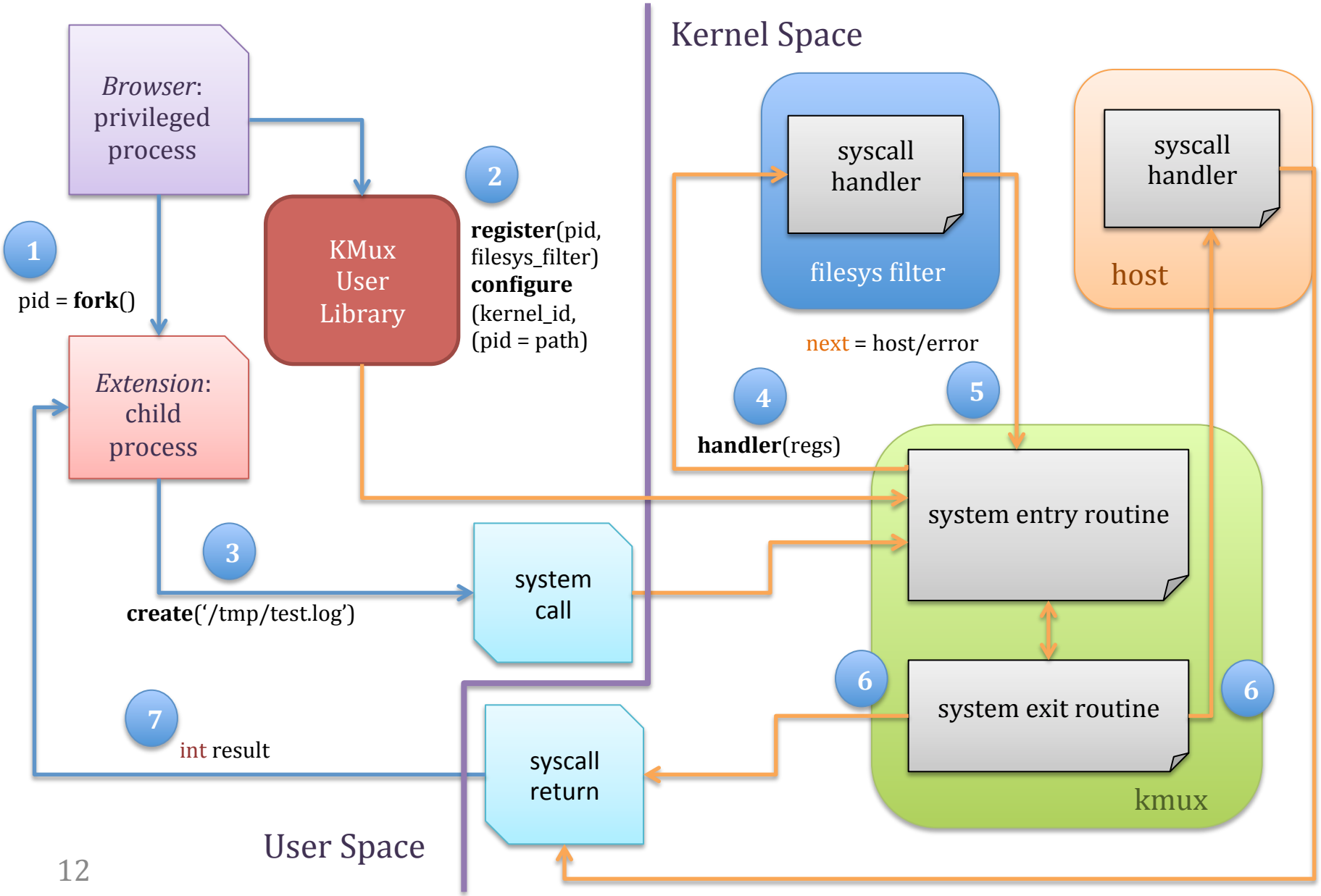
```
ls –al | grep kmux | less
```



Creating Subkernel/App Mapping



Accepting/Rejecting a System Call



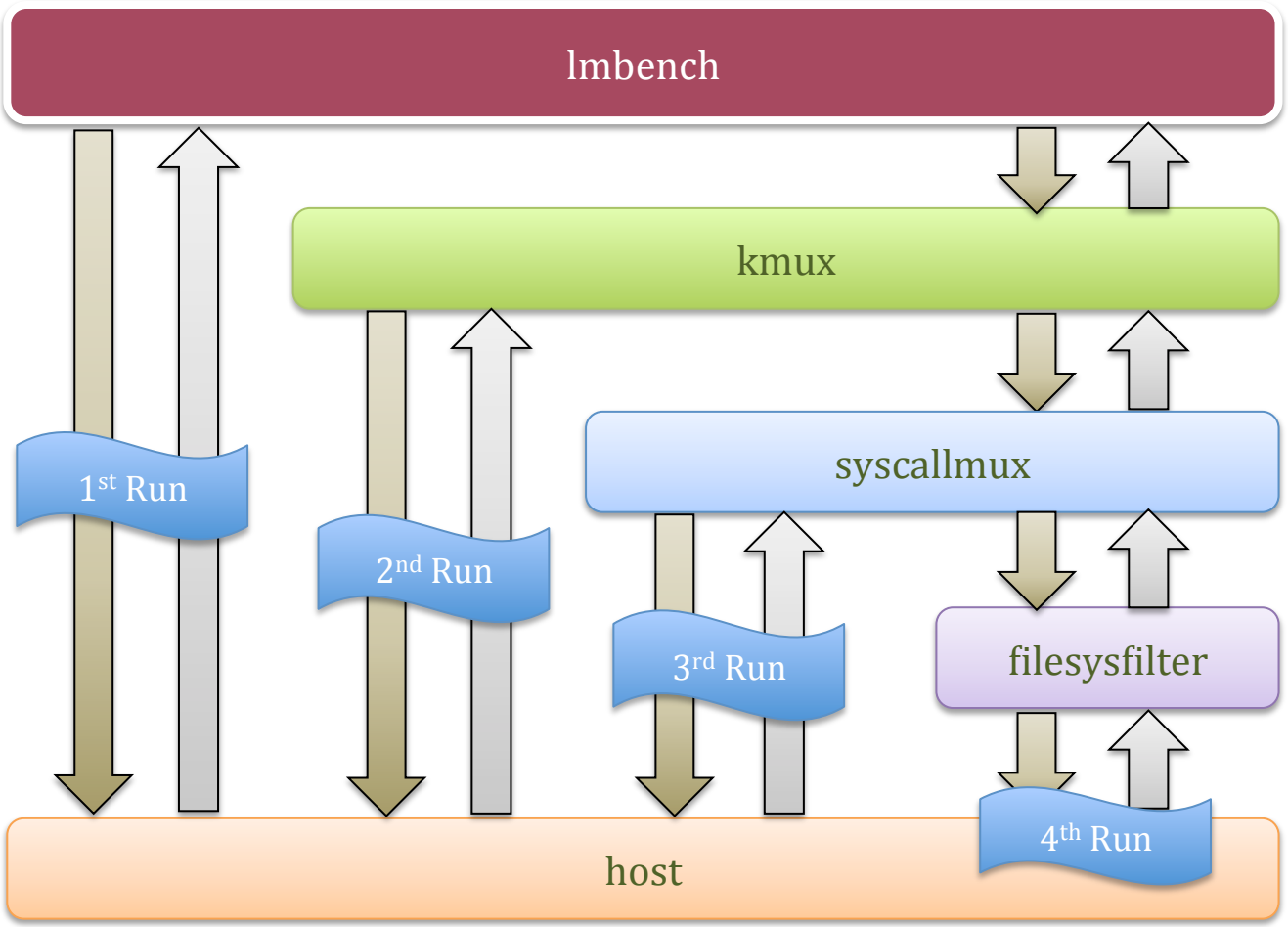
Benchmarks

- KMux overhead
- Subkernel overhead
- Multiple subkernel overhead

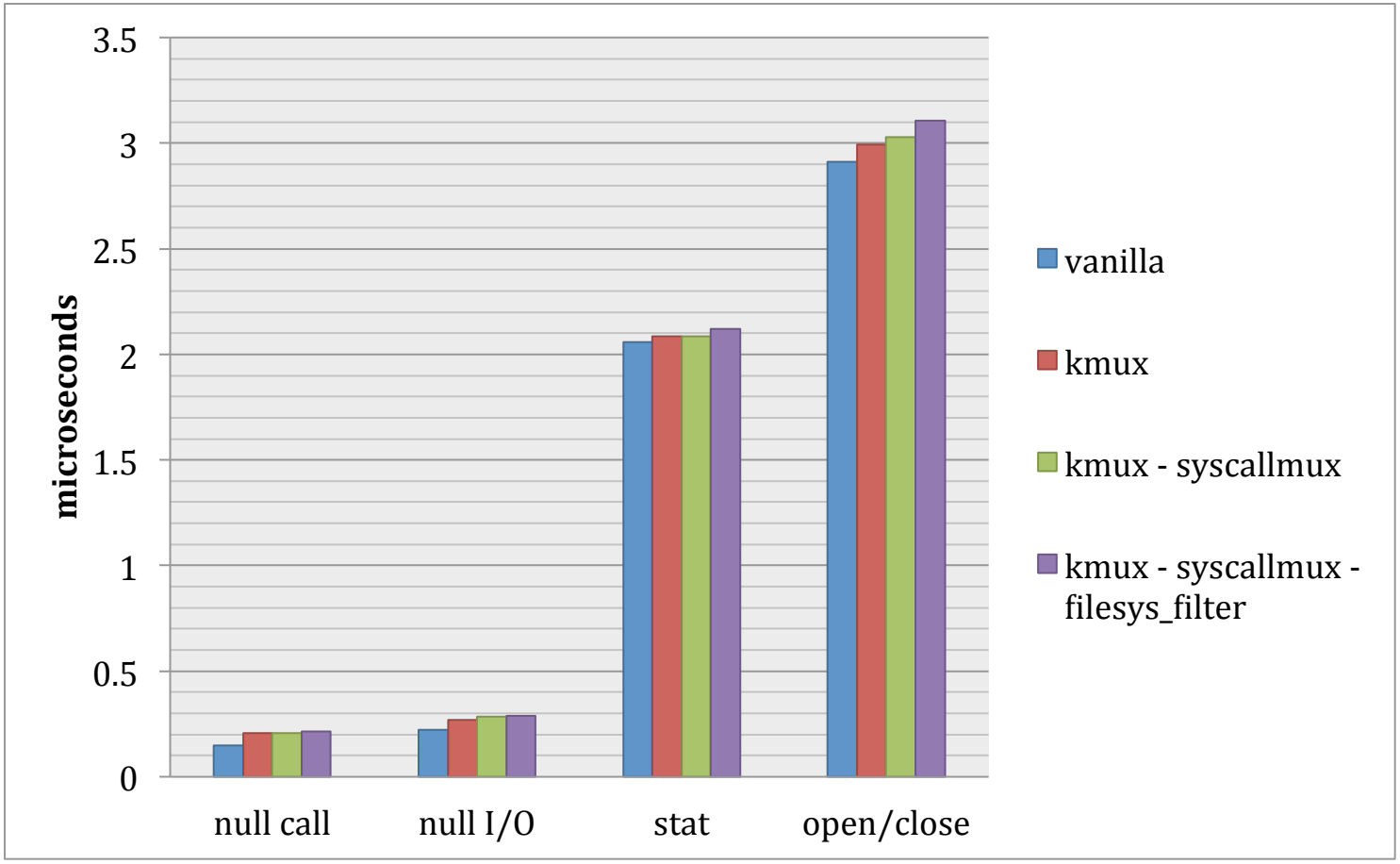
- Subkernels prepared:
  - *Null*: does not perform any action and returns to KMux immediately
  - *Syscall Multiplexer*: keeps track of which system call should be handled by which subkernel
  - *File System Filter*: confines file creation and deletion system calls from a registered process to a registered directory root
  - *Sandbox*: allows a preconfigured list of system calls to pass through for a given process or group of processes

Microbenchmarks

- LMBench OS only, on Core 2 Duo 1.8 GHz, 876 MB RAM

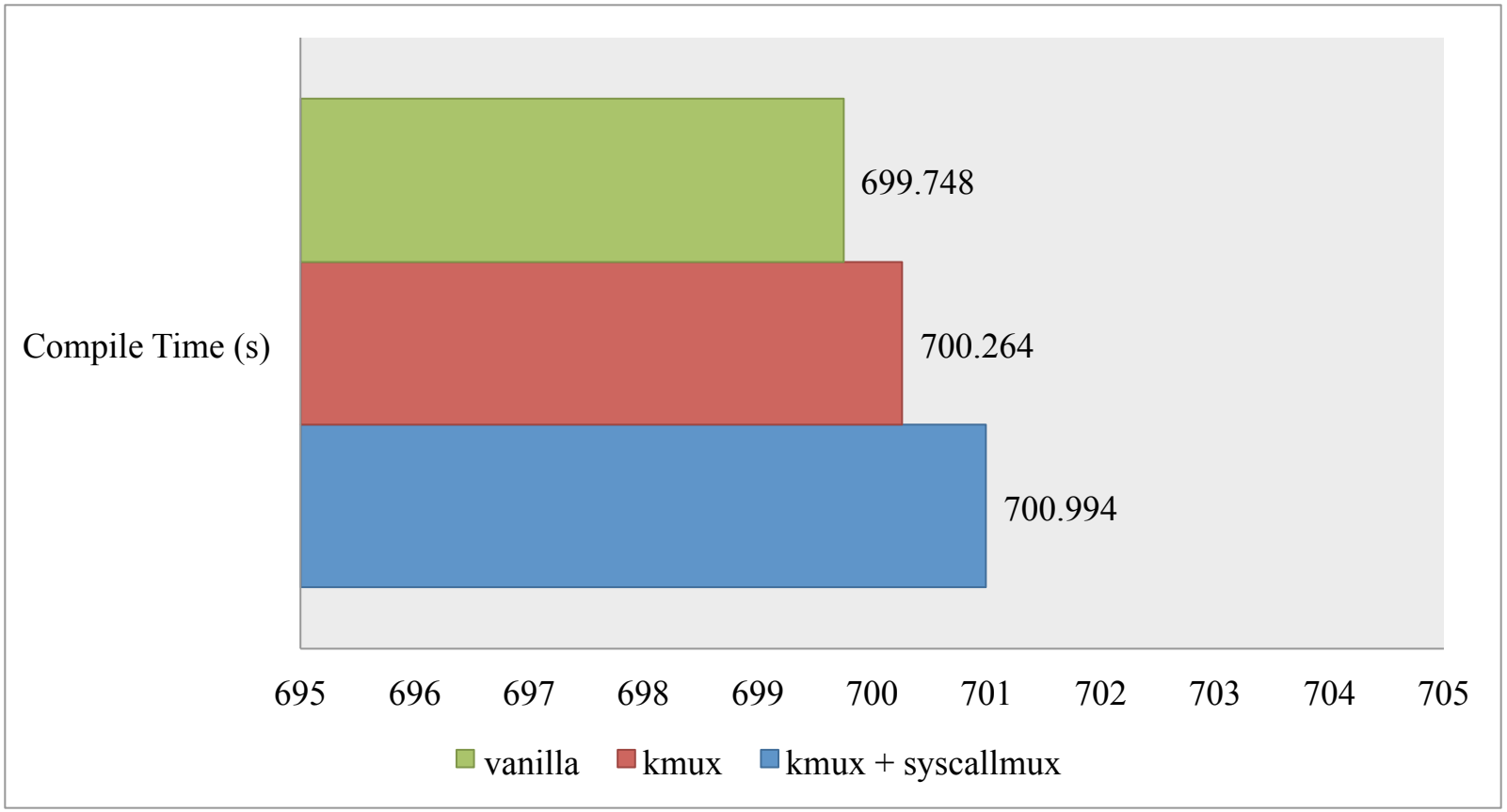


Basic OS Calls



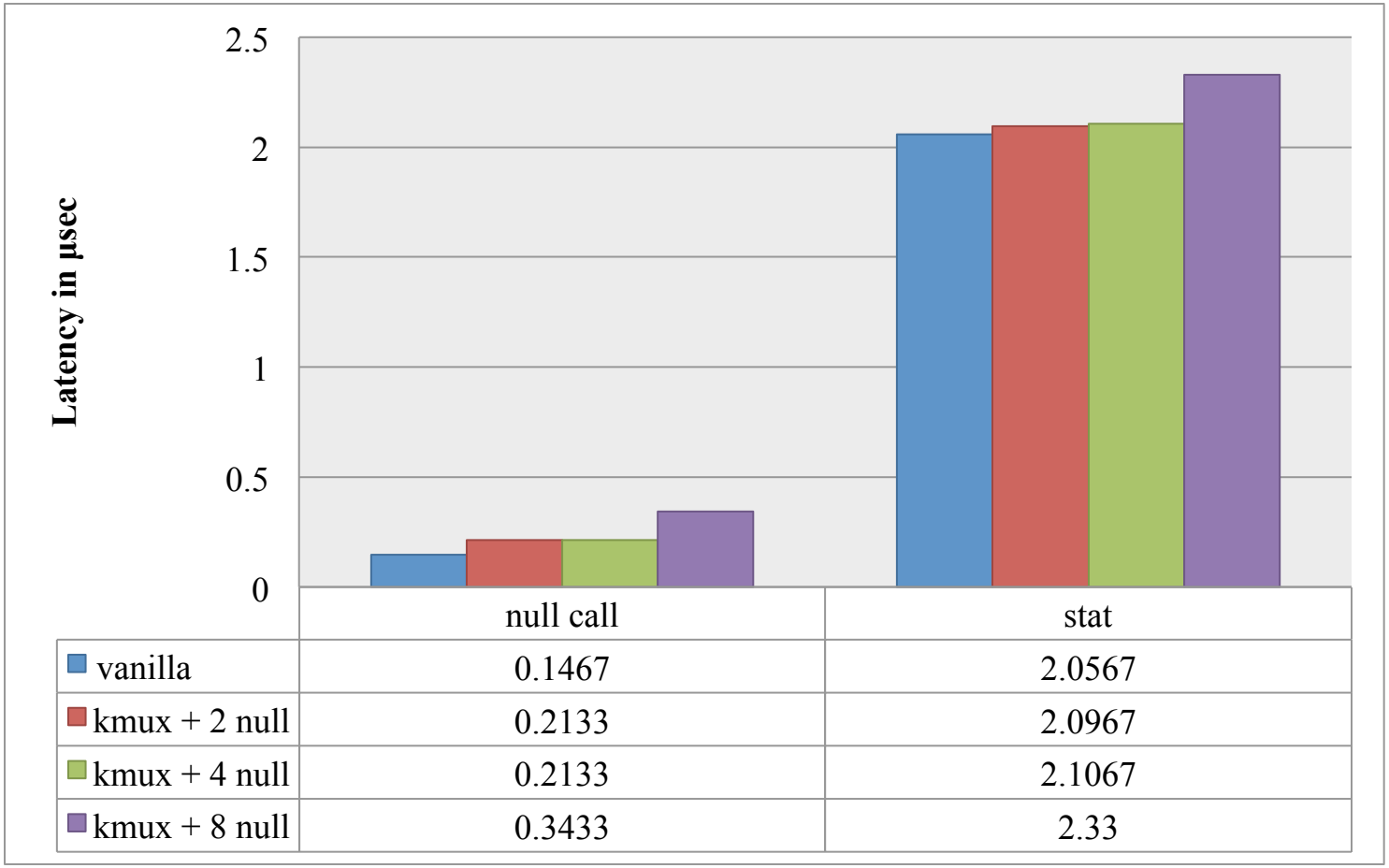
call	null call	null I/O	stat	open/close
Max overhead	40.9%	23.9%	3%	6.7%

Macrobenchmark



- Linux Kernel v2.6.33
  - Same configuration for all runs
  - Maximum overhead 0.17%

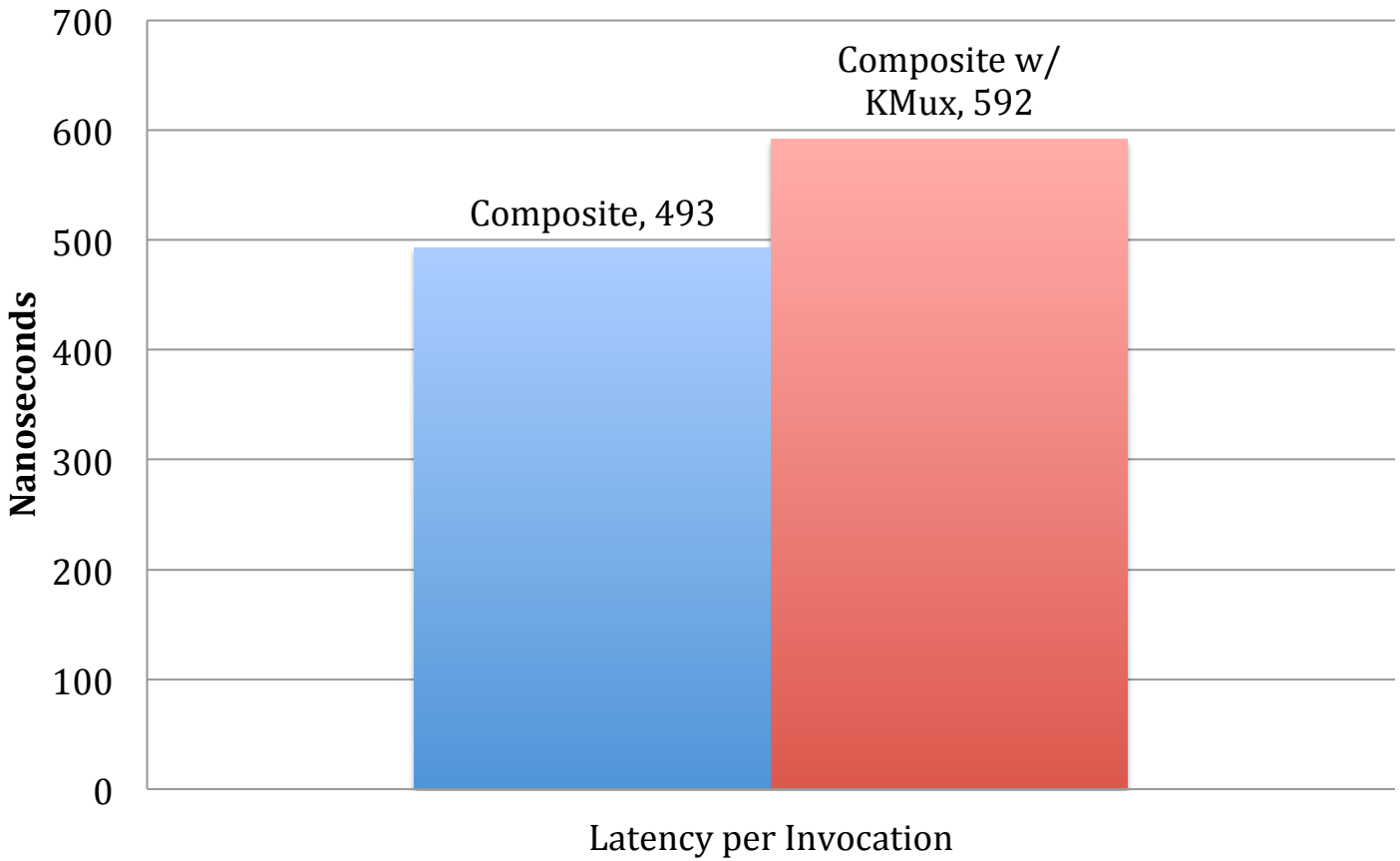
Pure Subkernel Chain Overhead



	null call	stat
vanilla	0.1467	2.0567
kmux + 2 null	0.2133	2.0967
kmux + 4 null	0.2133	2.1067
kmux + 8 null	0.3433	2.33

call	null call	stat
Max overhead	134%	13.2%

Self-sufficient Kernel - Composite



- Only 15 lines of code added to Composite code, mostly kernel and process registration/ deregistration
- Average overhead per invocation ~20%

Things KMux doesn't do

- KMux does not provide all functions necessary for subkernels,
  - Subkernels currently rely on host kernel for resources
- Isolation of subkernels not strictly enforced
  - One subkernel can trivially crash the system
  - Currently requires "trusted" subkernels

Related Work

- Hijack method, used by Composite component-based system
- System-call table overriding
  - SLIC, Systrace, virus checking software
- SPIN, Interposition agents
- SFI, XFI, Vx32, Native Client sandboxing

Conclusion

- KMux:
  - Practically efficient
  - Allows multiple kernels in the system!
  - Highly configurable
  - Works with commercial systems out of the box
  - Allows extension, enhancement, monitoring and even complete replacement of system calls
  - Perfect for sandboxing untrusted applications!

Questions

- Source available at:
  - github.com/tarequeh/kmux

Future

- Get multiple standard Linux kernels to work with KMux
- Modify Firefox/ Chrome to register extension processes with KMux
  - Activate extension sandboxing
- Implement KMux for Windows/ Mac OSX
- Provide interfaces necessary for subkernels and isolate them

Subkernels Chaining

- Upon receiving control for a system call, a subkernel may:
  - Perform necessary action and delegate to another subkernel, adding to the chain
  - Return control to host kernel or return to user space, ending the chain
- Chaining allows:
  - Decoupling of functionality
  - Subkernels to utilize each others capabilities
  - Analysis from multiple perspective before a system call is approved
  - Optimal use of the subkernel network

CPU Multiplexing

- Imagine 600 processors in a system
- Multiple kernels should be able to function
  - Each claiming a number of CPUs
- Create CPU – subkernel mapping
  - KMux provides this feature

- Efficient utilization of CPU
- Easier accommodation of multiple kernels
- Isolation domains

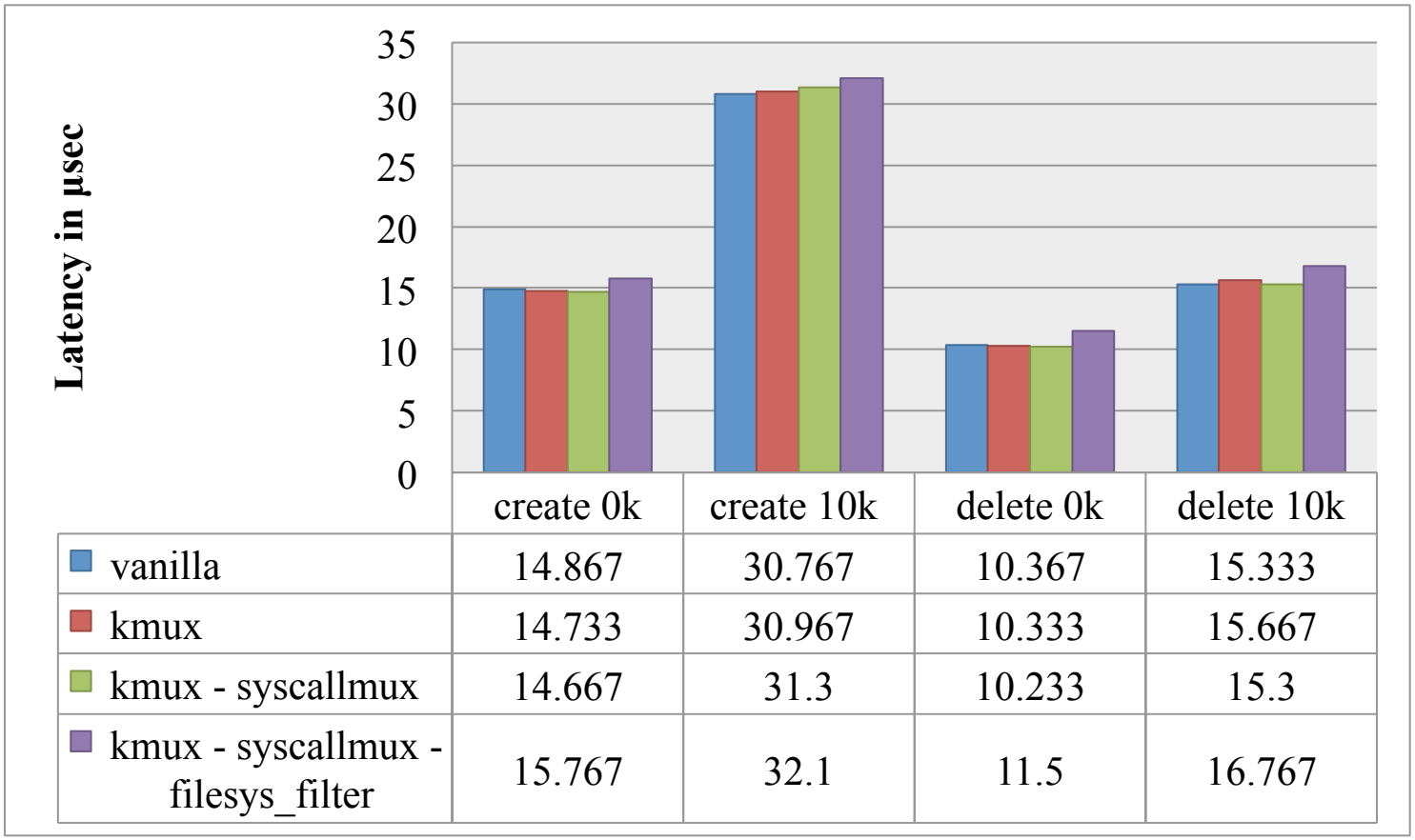
Using KMux in Kernel Space

- KMux provides interface at kernel level
- Subkernel registration
  - Subkernels can register themselves with KMux upon initialization
  - Subkernels can unregister themselves, and KMux provides a graceful exit path
- Subkernel configuration
  - Subkernels may choose to accept configuration parameters
  - Upon initialization they register a configuration handler with KMux
  - Accepts simple character string containing configuration information

Using KMux in User Space

- KMux user library provides access from user level
- Process Binding
  - KMux will by default delegate control for all system calls to the host kernel
  - Privileged processes or users can bind a process to a particular subkernel
  - All system calls from a bound process will be delegated to chosen subkernel
- Subkernel configuration
  - Privileged processes or users can configure subkernels to suit their needs
  - Example configurations are: specifying next subkernel in chain or set of system calls to analyze

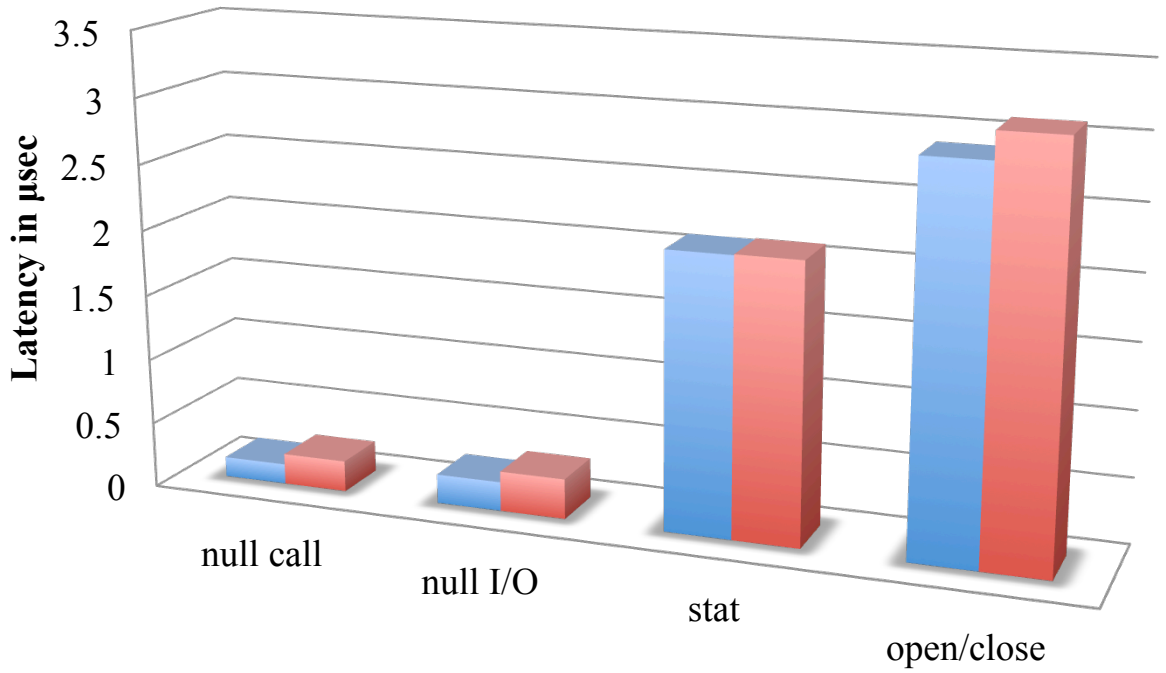
File System Calls



	create 0k	create 10k	delete 0k	delete 10k
vanilla	14.867	30.767	10.367	15.333
kmux	14.733	30.967	10.333	15.667
kmux - syscallmux	14.667	31.3	10.233	15.3
kmux - syscallmux - filesys_filter	15.767	32.1	11.5	16.767

call	create 0k	create 10k	delete 0k	delete 10k
Max overhead	6%	4.3%	10.9%	9.3%

Sandbox Performance



	null call	null I/O	stat	open/close
Vanilla	0.1533	0.2233	2.1033	2.8966
KMux + Sandbox	0.2366	0.3066	2.11	3.1

call	null call	null I/O	stat	open/close
Max overhead	54%	37%	1%	7%