Trustworthy kernel separation through monads

Peter White: wpd@qwest.net

Jim Hook: hook@cse.ogi.edu

Outline

- Spook and Programatica
- Architecture for high assurance of separation
- End game: Running COTS applications on COTS Hardware
- Results

Spook and Programatica

What is Spook?

- A high assurance POSIX compliant kernel
  - POSIX chosen to support COTS applications
  - We think a real OS with high assurance is within reach
- Adds domain concept to POSIX
  - Strict separation between domains
    - Enforcement of a communication policy between domains
  - Special separation domain providing strict separation between processes
    - Enforcement of a communication policy on user processes
- High assurance of strict separation based mostly on types, and partly on proof of properties

Spook and programatica

- Programatica adds properties to programs
  - Properties are specified along with the program, in the same text file.
- Programatica is providing a formally specified Haskell (syntax, semantics, and logic)
- Spook is a large program having a property (Separation) as its main objective
- Spook properties take advantage of some of the unique features of Haskell
  - Laziness and infinite lists
  - Potentially undefined computations
  - Monads
- Conclusion: Spook is a good test case for Programatica

Spook theme

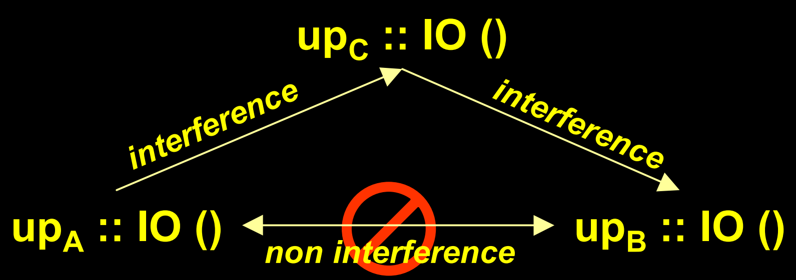
(This presentation in one slide)

- Separation based on the ST monad
- Concurrency based on the IO monad, and laziness
- Avoid proof, use types!
- Bottom line: high assurance on a larger OS

Architecture for Assurance

Final Goal: Strictly separated POSIX user processes

- Definition of Goguen and Meseguer [2]:
  - "To establish that information does not flow from object A to object B, it is sufficient to establish that A's behaviour has no effect on what B can observe."
  - "B's view of the system is independent of A's behaviour"
  - The definition is formalized in terms of potentially infinite lists of inputs and outputs for A and B.
- Historical note: This definition superseded by Rushby [3]
- Fundamental goal of Spook: provide high grade separation between Spook processes and / or domains



By policy, some processes are permitted to communicate, others are not.

up = "User Process"

Architectural theme: Avoid proof, use type inference instead

A simple example

- Assurance by types alone [4]
  - **m :: (a -> b) -> m a -> m b**
  - **∴ m f = map f . r where r is a rearrangement**

- Assurance by proof
  - **m :: (a -> b) -> ma -> m b**
  - **~~∴~~ as < m f as**

- This conclusion cannot be reached based on types alone, requires proof based on the properties of particular f:
  - **∴ as < m (+1) as**

Architectural theme: Separation arising from types

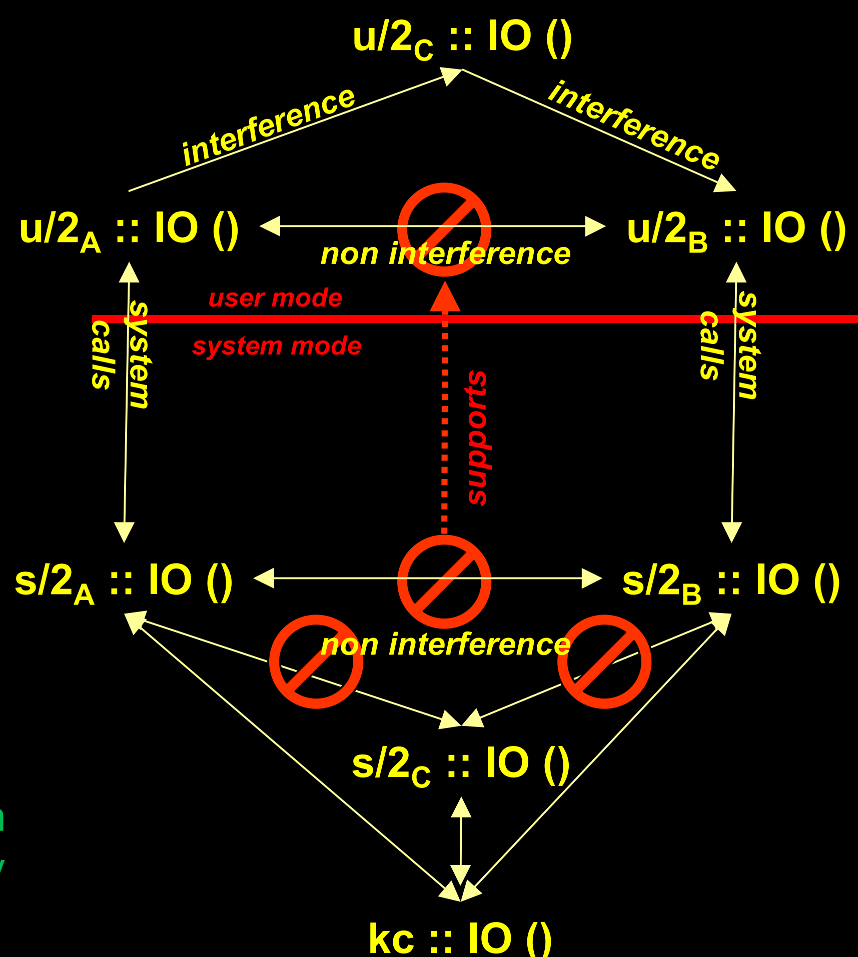
- Assurance by types alone [4]
  - **st_1 :: [m] -> ST s [m]**
  - **st_2 :: [m] -> ST s [m]**
- st_1 and st_2 are independent when encapsulated by runST
  - **∴ following code fragment will return True no matter how interleave1 and interleave2 do their interleaving**

```
let out1 = runST (st1 in1)
    out2 = runST (st2 in2)
    mixed1 = interleave1 out1 out2
    mixed2 = interleave2 out1 out2
    out1a = filter fromst1 mixed1
    out1b = filter fromst1 mixed2
in take n out1a == take n out1b
```

- Unfortunately, a kernel must do IO activities, so [m] -> ST s [m] is not the correct type for the kernel.
- Type of the kernel must be IO ().
- Basic Haskell rule: something of type *IO ()* can call something of type *[m] -> ST s [m],* but something of type *[m] -> ST s [m]* cannot (safely) call something of type *IO ().*
- Architectural imperative of Spook: Put as much as possible into the type *[m] -> ST s [m],* and as little as possible into the type *IO ().*

First step: A portion of the kernel for each user process

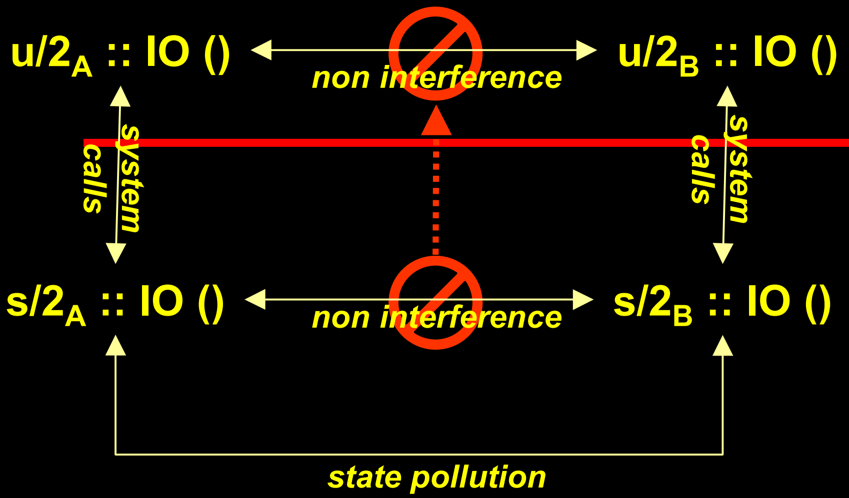
- Each process has
  - User half (u/2): portion of the process running the user's application.
  - System half (s/2): portion of the kernel dedicated to the process.
  - Posix interface between u/2 and s/2.
- The goal of non interference between the user halves should be supported by non interference between the system halves
- Interprocess communication goes through the kernel core (kc), which enforces the communication policy



u/2 = "User Half"
s/2 = "System Half"
kc = "Kernel core"

Major problem: Getting the non interference between $s/2_A$ and $s/2_B$

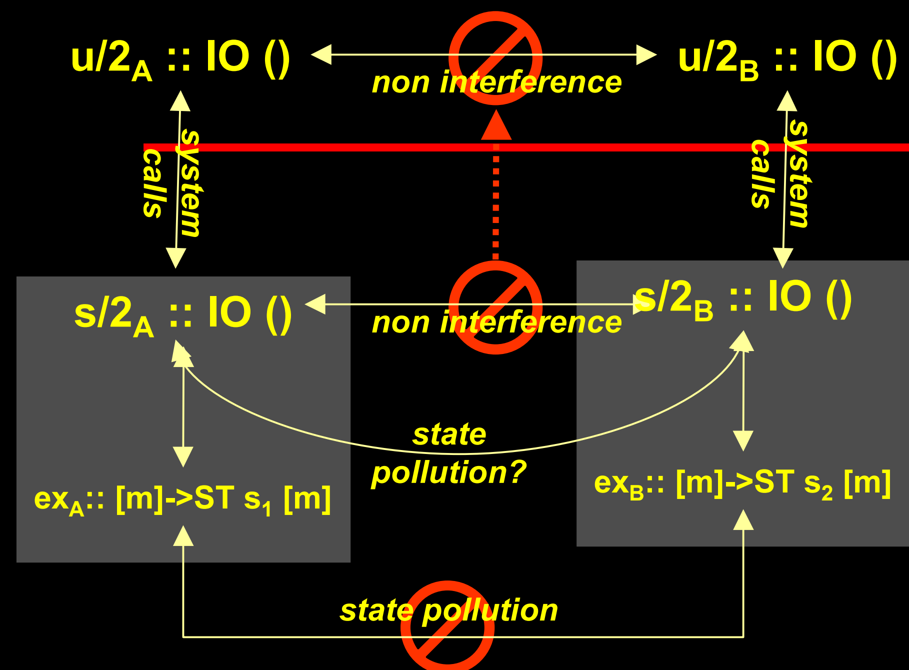
- Because they support POSIX, each system half will be complex and state intensive
  - In typical programming languages (e.g. C, C++), it is difficult (or impossible) to guarantee that there are no coding errors whereby state manipulations in one system half affect the state in another system half.



u/2 = “User Half”
s/2 = “System Half”

Solution: Encapsulation of a state transformer by runST

- Theorem of Launchbury [1]:
  - When the state transformer is encapsulated by *runST*, Values within one state transformer cannot depend upon references within another state transformer
  - When encapsulated by *runST*, the behaviour of the state transformer is independent of the layout of data in memory
  - *runST* :: ***forall*** *a.(**forall** s.ST s a)-> a* is a pure function
- The IO shell part of the system half is not covered by this theorem:
  - The IO shell should be thin
  - The executive can be thick

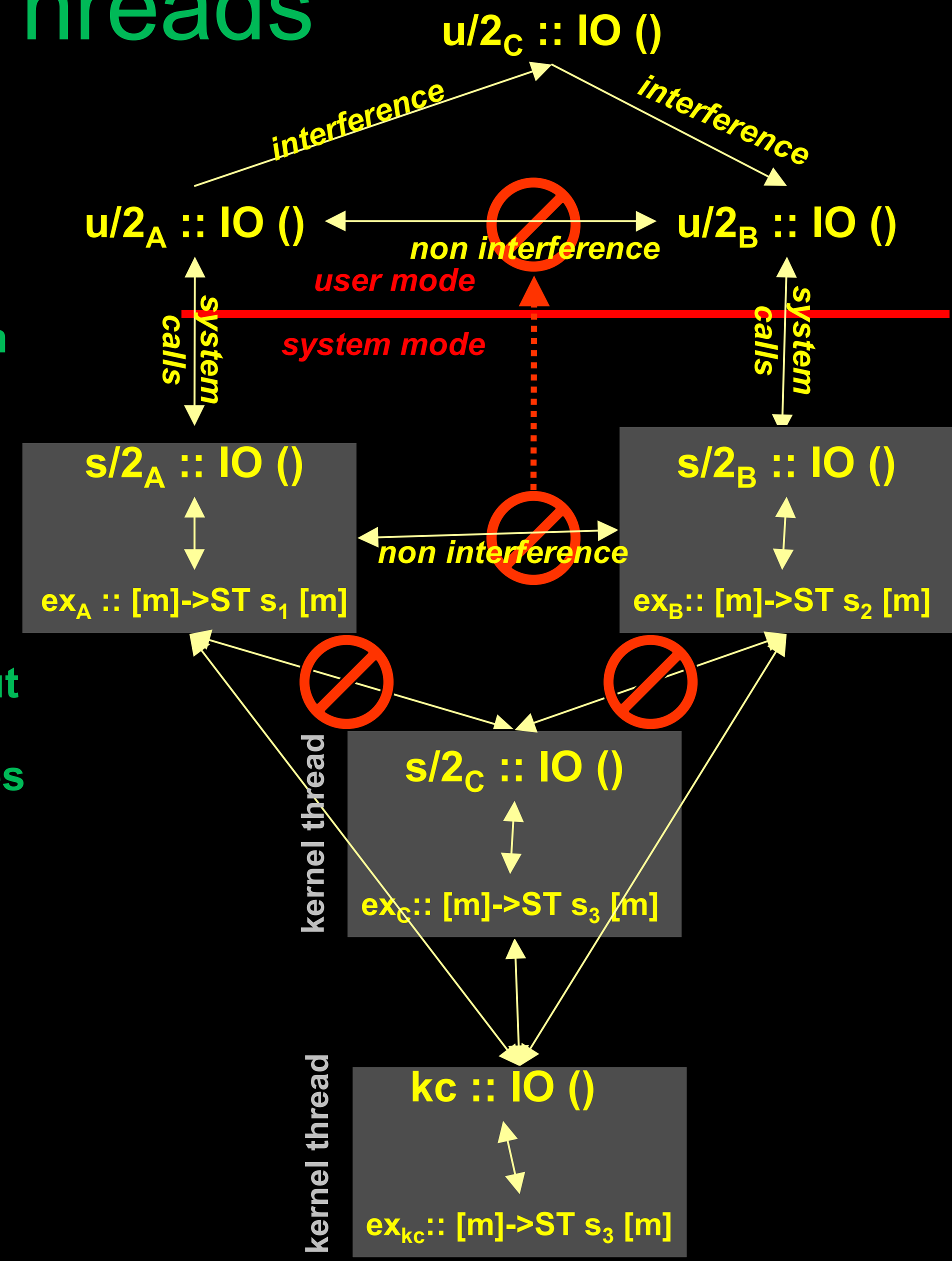


u/2 = “User Half”
s/2 = “System Half”
exec = “Executive”
grey box = Kernel thread

- *The statements of state transformer independence, and non interference are tantalizingly close.*

Message Passing and Kernel Threads

- Each system half is contained in a kernel thread
- The kernel core is contained in a kernel thread
- The kernel threads communicate with messages
- The executive is a state transformer (encapsulated by runST) that transforms a (potentially infinite) list of input messages into a (potentially infinite) list of output messages
- The executive is polymorphic (with class constraint) in the message type, meaning that most details of the message type do not affect reasoning about the executive



u/2 = “User Half”
s/2 = “System Half”
kc = “Kernel core”

for Spook: Model = Program

- Because the state transformer type *[m] -> ST s [m]* can handle infinite lists, the statements of separation and non interference apply directly to an object in the program.
- Because Programatica provides a logic of Haskell programs, the separation statement applies directly to the program.
- Because Programatica places the properties in the same text as the program, for Spook, the model and the program are one and the same.

Mediation of message passing

- By design, the state transformers only communicate with the kernel core. Direct interference of state transformers is therefore mediated by the kernel core.
- State transformers cannot interfere with each other by means of internal state manipulations.
  - Enforced by type inference
- State transformers do not directly interfere with each other by means of messages,
  - Property of design
  - Will try to raise “do not” to “cannot” by better use of message types.

Overlapping system calls

- Activities on behalf of a process must be overlapped (IO activity with non-IO system call)
- The asynchronad is a monad developed for Spook, it permits a system call program to be implemented in steps, where steps from different programs can be interleaved.
- The executive is responsible for making the interleaving work.

Monad 101

- Bind:
  - **>>= :: M a -> (a -> M b) -> M b**
- Unit:
  - **return :: a -> M a**
- The monad sequences monad actions.
- Operation a_1 returns a value, which is plugged into x_2, etc.
- Normally, when the sequence of operations begins, the computation continues until the return without an explicit break.

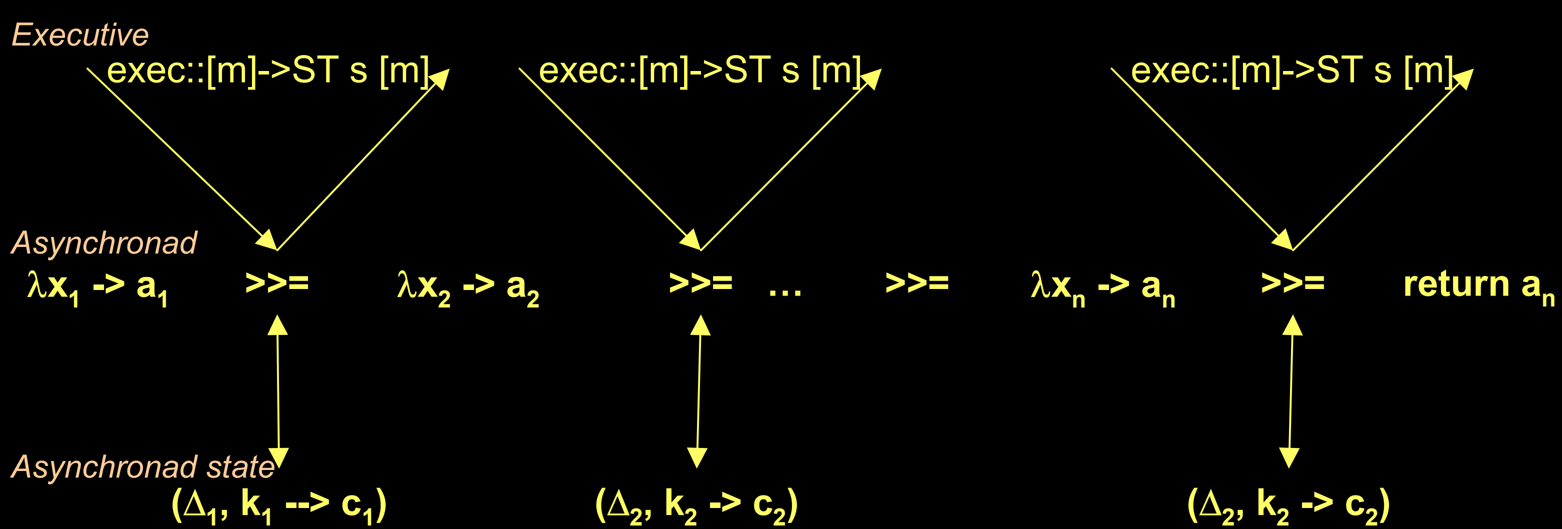
$$\lambda x_1 \text{ -> } a_1 \text{ >>= } \lambda x_2 \text{ -> } a_2 \text{ >>= } \ldots \text{ >>= } \lambda x_n \text{ -> } a_n \text{ >>= return } a_n$$

The first parameter (x_1) is a parameter to the entire sequence of monad actions

Asynchronad

Broken operation with partitioned state

- At each bind (>>=) operation, the asynchronad can:
  - Take in an input message
  - Continue or break
    - On break, can produce one or more “uniqified” output messages
    - On continue, can accumulate one or more output messages
  - Transform the partitioned state (k_n -> c_n)
  - Reduce the partitioned state guard (Δ_n)
- The executive is now a state transformer layer to coordinate the asynchronad actions



Partitioned State

- The state is broken into state components
- Each system call is assigned a guard:
  - *observe* set of state components that it is allowed to access
  - *alter* set of state components it is allowed to modify
- The local (system half) and global access / modify sets are used to control the interleaving of system calls on behalf of a single process
- The global (kernel core) observe / alter sets can be used for covert channel analysis and elimination

Overall kernel thread structure

- Three layers
  - IO shell (IO monad) :: *IO ()*
  - Executive State transformer (ST monad): *[m] -> ST s [m]*
  - Actions (Asynchronad) :: *Asynchronad k c m p a*

End Game

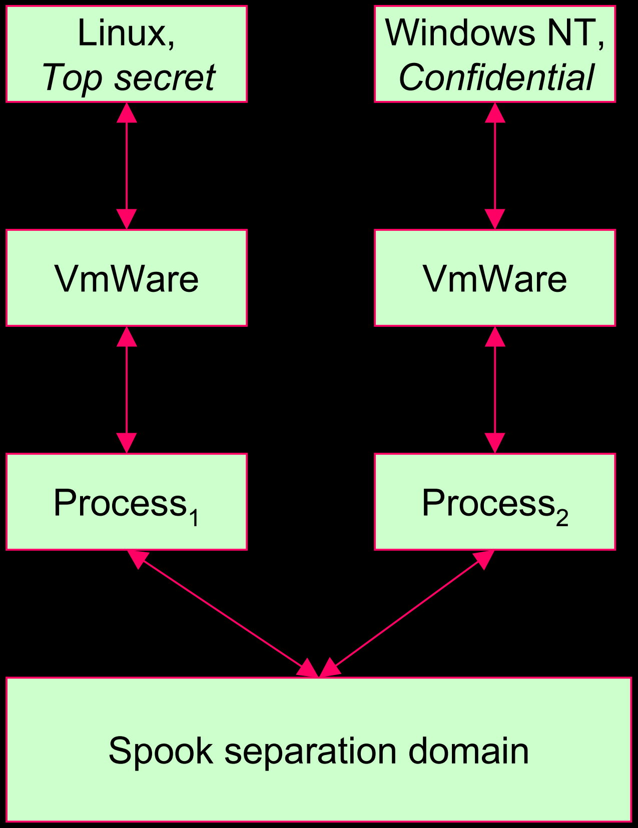
Running COTS applications

Running real COTS applications

- COTS applications are rarely strictly POSIX compliant, they use other features of Linux or Windows, and interfere with each other.
- Running them in non interfering processes will break them.

Running real COTS applications: one approach

- Run VmWare, encapsulated in a process in a separation domain.
- Requires some Linux extensions to POSIX
- This would support a NetTop style architecture



Another approach: separated domains *(process / domain model)*

- Divide spook into separated domains
  - Standard domain: Standard POSIX, with enough Linux hooks added to support common COTS applications.
  - Separated domain: Standard POSIX with limitations, strictly separated processes
- Provide socket inter-domain communication, mediated by Spook

Results

ST s [m] encapsulation

(as of 3/5/2002)

- System half encapsulated: 6766 HLOC
- System half IO shell: 166 HLOC (2.4%)
- Kernel core encapsulated: 1859 HLOC
- Kernel core IO shell: 166 HLOC (8.2%)
- Kernel core other (e.g. init): 3715 HLOC
- Total Spook: 14959 HLOC

System calls implemented

(22 so far)

- fork*
  - fork interacts with many features of POSIX. As more features are introduced, fork must be revisited.
- exit
- getpid
- getppid
- getpgrp
- getpgid
- setpgid
- setpgrp
- setsid
- alarm
- pause
- sigaction
- sigprocmask
- sigpending
- kill
- sigsuspend
- sleep
- sigemptyset
- sigfillset
- sigaddset
- sigdelset
- sigismember

System calls coming soon

- getlogin
- getuid
- geteuid
- getgid
- getegid
- getgroups

- mq_open
- mq_close
- mq_send
- mq_receive
- mq_notify
- mq_getattr
- mq_setattr
- mq_unlink

References

- John Launchbury and Simon Peyton Jones*, Lazy Functional State Threads.* In PLDI'94: Programming Language Design and Implementation, Orlando, Florida, pages 24-35, June 1994, ACM Press.
- J. A. Goguen and J. Meseguer, *Security Policies and Security Models*, In IEEE Symposium on Security and Privacy, 1982.
- J. Rushby, *Noninterference, Transitivity, and Channel-Control Security Policies,* 1992.
- P. Wadler, *Theorems for free*, Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, FPCA 1989, London, UK.

Backup

Spook and Haskell

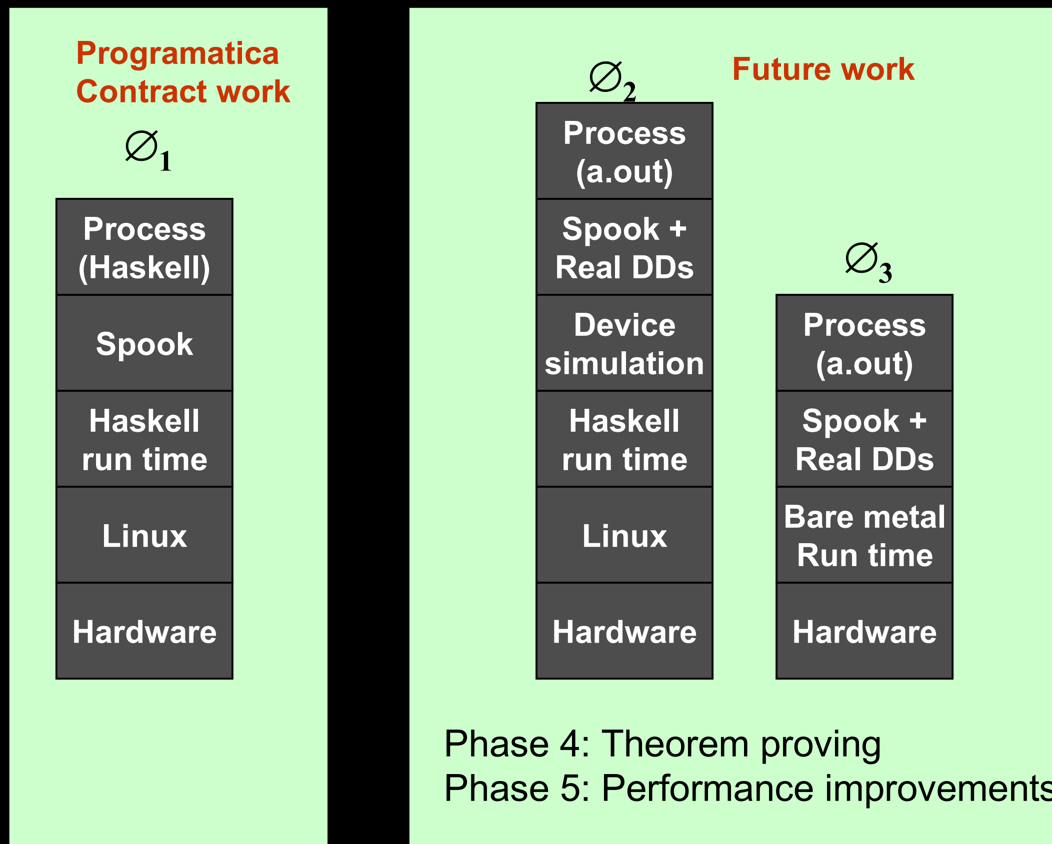
- Why would you write an operating system in Haskell?
  - Type safety for assurance
  - ST monad provides a good basis for separation
  - Haskell has excellent concurrency primitives, and the full power of a functional language for combining and composing concurrency operations.
  - Heap allocation, which is a good basis for some resource allocation problems.
- In other words, many parts of the problem are already solved!

Further work

Device drivers

- So far, there is only a timer device driver
  - Uses only resources allocated by the system halves (no resource covert channels)
  - However, lack of covert channels still depends on correctness of timer device driver
- File system device driver will be hard work, this may be part of the kernel core

Haskell on bare metal



Programatica will provide many POSIX.1 and some POSIX.4 interfaces

Covert channel elimination

- There is a potential covert channel when system calls by different user processes affect the same component of the kernel core partitioned state
- Techniques to eliminate the covert channels:
  - Partition resources according to the process / domain model
  - Special case techniques, such as random generation of process Ids

POSIX.1 compliance

(98 calls)

- Access
- Alarm
- Cfgetispeed
- Cfgetospeed
- Cfsetispeed
- Cfsetospeed
- Chdir
- Chmod
- Chown
- Close
- Closedir
- Creat
- Ctermid
- Cuserid
- Dup
- Dup2
- Execl
- Execle
- Execlp
- Execv
- Execve
- Execvp
- _exit
- Fcntl
- Fdopen
- Fileno
- Fork
- Fpathconf
- Fstat
- Getcwd
- Getegid
- Getenv
- Geteuid
- Getgid
- Getgrgid
- Getgrnam
- Getgroups
- Getlogin
- Getpgid
- Getpgrp
- Getpid
- Getppid
- Getpwnam
- Getpwuid
- Getuid
- Kill
- Link
- Lseek
- Mkdir
- Mkfifo
- Open
- Opendir
- Pathconf
- Pause
- Pipe
- Read
- Readdir
- Rename
- Rewinddir
- Rmdir
- Setgid
- Setpgid
- Setpgrp
- Setsid
- Setuid
- Sigaction
- Sigaddset
- Sigdelset
- Sigemptyset
- Sigfillset
- Sigismember
- Siglongjmp
- Sigpending
- Sigprocmask
- Sigsetjmp
- Sigsuspend
- Sleep
- Stat
- Sysconf
- Tcdrain
- Tcflow
- Tcflush
- Tcgetattr
- Tcgetpgroup
- Tcsendbreak
- Tcsetattr
- Tcsetpgrp
- Time
- Times
- Ttyname
- Tzset
- Umask
- Uname
- Unlink
- Utime
- Wait
- Waitpid
- Write

Pink = Coming soon
Lavender = Completed

POSIX.4 compliance
(58 calls)

Sigwaitinfo
Sigtimedwait
Sigqueue
Sched_setparam
Sched_getparam
Sched_setscheduler
Sched_getscheduler
Sched_yield
Sched_get_priority_max
Sched_get_priority_min
Sched_rr_get_interval
Clock_settime
Clock_gettime
Clock_getres
Timer_create
Timer_delete

Timer_settime
Timer_gettime
Timer_getoverrun
Nanosleep
Aio_read
Aio_write
Lio_listio
Aio_suspend
Aio_cancel
Aio_error
Aio_return
Aio_fsync
Fdatasync
Msync
Aio_fsync
Fsync

Mmap
Munmap
Ftruncate
Msync
Mlockall
Munlockall
Mlock
Munlock
Mprotect
Mq_open
Mq_close
Mq_unlink
Mq_send
Ma_receive
Mq_notify
Mq_setattr

Mq_getattr
Sem_init
Sem_destroy
Sem_open
Sem_close
Sem_unlink
Sem_wait
Sem_trywait
Sem_post
Sem_getvalue