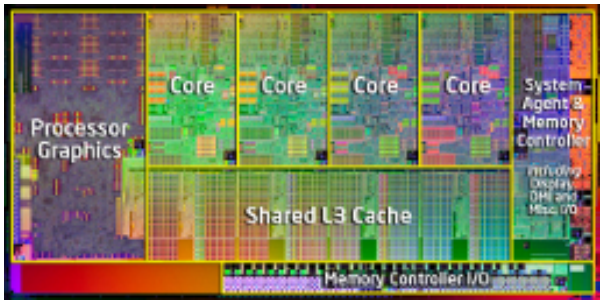
Virtual Machine Support for Parallel Language Runtimes EECS 441 Class Project Final Presentation

Jedidiah McClurg Kaicheng Zhang Yixi Zhang

Northwestern University

May 30, 2012

Introduction



1

2

- Parallelism is an imporant aspect of modern computing
- Machines offer anywhere from 4 to 200,000+ cores
- Programs should exploit this parallelism when possible
- *Programming languages* are beginning to offer built-in constructs which allow programmers to do this
- **What exactly is needed at the OS and (virtual) machine level to support the runtime for such a language?**

[1] http://www.pcmag.com/
[2] http://www.nccs.gov/

Project Goals

- We seek to obtain concrete answers to this question by doing the following:
  1. Identify a minimalist environment which could conceivably support a parallel language runtime
  2. Port the Racket language runtime to this minimal OS
  3. Add necessary OS-level functionality to support Racket's parallelism constructs
  4. Test the setup running on a virtual machine
- This will provide us with an understanding of the features that are unique to parallel language runtimes, and point us towards ways of improving the cooperation between the OS/hardware and the runtime.

Kitten Operating System

- We target the Kitten OS [3] for running Racket
- Kitten is a lightweight kernel (LWK) based on Linux
  - Designed for running on supercomputer nodes
  - Small, manageable code base
  - No support for physical disks (only virtual file system)
  - Provides support for Pthreads
  - Using Palacios VMM, can function as a host OS
  - Runs as a guest in VirtualBox, QEMU
- Our first goal is to get Racket running on this OS

[3] https://software.sandia.gov/~ktpedre/kitten_overview.pdf

Racket Runtime

- Racket is a functional programming language in the LISP/Scheme family
- Programs can be compiled to bytecode (to run on the Racket VM)
- Racket supports JIT compilation
- Racket also includes an interpreter (we focus on this)
- The `racket` command accepts either console input via the interactive read-eval-print loop, or input from source files
- The language now supports two parallelism constructs
  - Futures [1], which allow independent computations to be run in parallel within the (single-threaded) Racket VM
  - Places [2], which allow message-passing, and basically operate in new instances of the Racket VM

Project Overview

This is a list of things we had to do to accomplish our goals:

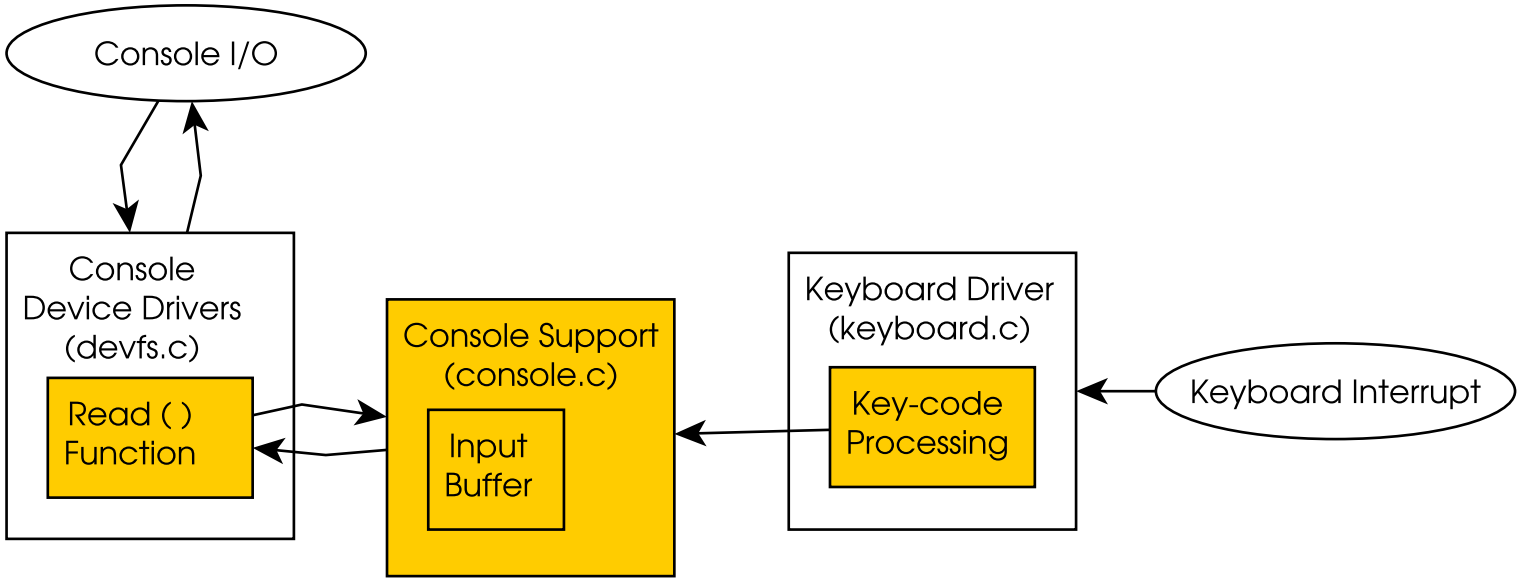
- Compile Racket as a static binary and connect it with the Kitten ISO image
- Add support for console input in Kitten
- Add support for instantiating a directory tree in the Kitten VFS (Racket collections)
- Implement needed system calls in Kitten
- Debug Racket memory issues
- Implement pipes in Kitten
- Test Racket running in a Kitten guest on top of Palacios

Static Racket Binary

- Kitten does not have straightforward support for shared libraries
- Thus, we compiled Racket from source and statically linked with libc, etc.
- The Kitten build generates an ISO image which can be run in a VM
- Kitten packages a given `init_task` program into the ISO image, so we pointed the build to our static Racket binary
- The Kitten build configuration allows a set of command line options to be passed to the `init_task` on startup, so we can use this to specify a Racket source file, etc.

Kitten Console Read

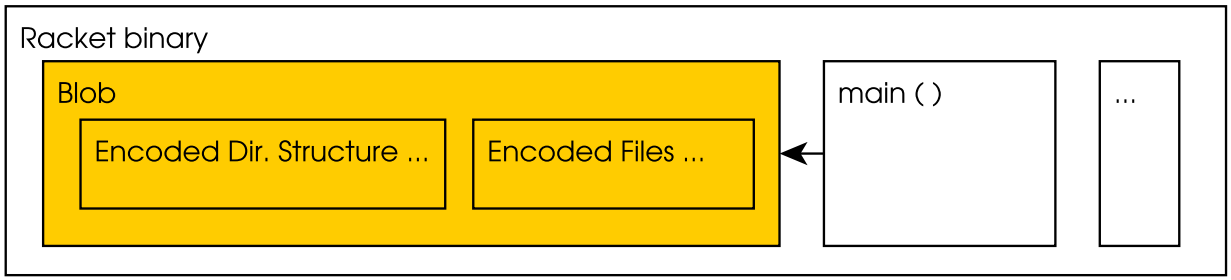
- Kitten previously had console *write*, but no *read*
- We want to use Racket interactively, so we added this support



- Keyboard interrupts are intercepted by the keyboard driver
- Key-codes are processed to determine characters, and characters are sent to console support
- Console support adds characters to the input buffer (spinlocks used for thread safety)
- Console reads are satisfied from the input buffer

Racket Collections and the Kitten VFS

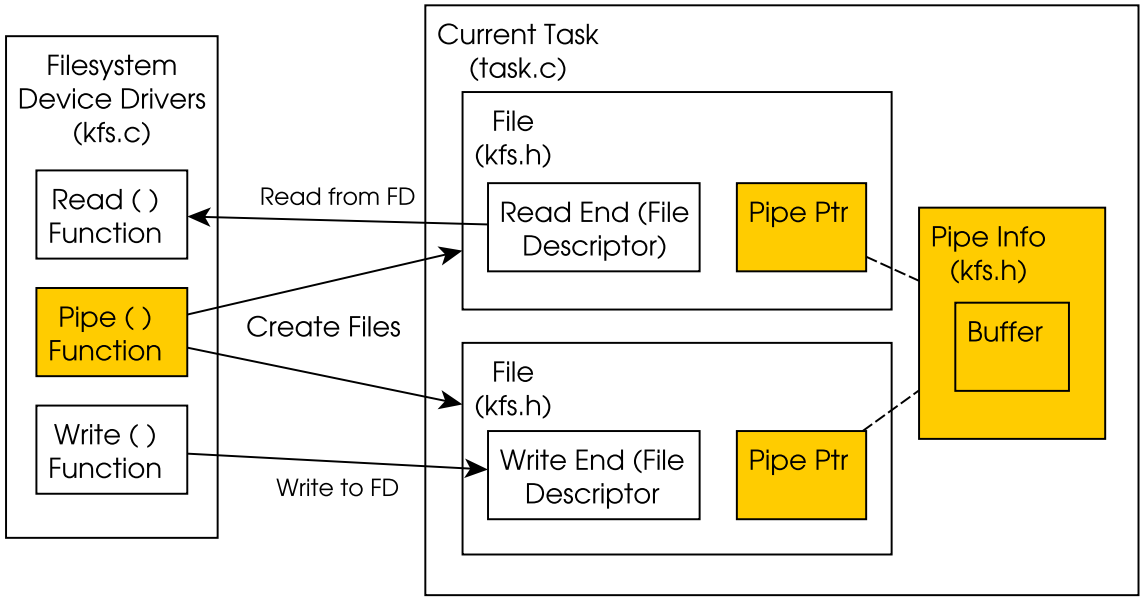
- Kitten does not support physical disk drives, meaning all filesystem access is via the memory-resident virtual file system (VFS)
- Racket requires a set of pre-compiled Racket libraries to start up
- Running it with "`racket --collects /tmp/racket`" loads these libraries from `/tmp/racket`
- How can we store these (1000+) files in `/tmp/racket` within Kitten?
- Embed them into the binary, and extract upon startup:



- Running above `racket` command with "`-expandblob`" now does this extraction

Kitten Pipes

- Kitten previously had no support for pipes, which are needed by Racket's parallelism constructs



- Each of a task's open files now includes an optional pointer to a pipe structure
- The `pipe` system call creates a pipe structure, and two regular files in the VFS which point to it
- For FDs which correspond to pipes, reads/writes from/to it are satisfied from the pipe buffer

Racket Memory Issues

- Even with all the preceding functionality in place, Racket was segfaulting (and failing to start) on Kitten
- We verified that the segfault occured on our host OS as well (using `gdb`), but it did not prevent Racket from starting on that machine
- We noticed that the previous Kitten page fault handler generated an unrecoverable error
- In an attempt to fix the problem, we modified the Kitten page fault handler to instead send the SIGSEGV signal to the offending task (Racket has a SIGSEGV handler)
- This allowed Racket to initialize further, but a subsequent segfault caused the task to hang
- Finally, we switched from the Racket 3M garbage collector to the conservative garbage collector (CGC), and this solved the problem

Results / Demo

- We are able to run Racket interactively on top of a Kitten VirtualBox (or Palacios) guest
- Futures and places operate as expected
- We can successfully run the following simple Racket demo programs
- (let ([x 12345])
  (begin (print x) (begin (printf "\nHello from Racket\n")
  "This is the return value\n")))
- #lang racket
  (let ([f (future (lambda () (+ 1 2)))]) (print (list (+ 3 4) (touch f))))
  (let ([pls (for/list ([i (in-range 2)]) (dynamic-place "tmp/racket/place-worker.rkt" 'place-main))]) (for ([i (in-range 2)] [p pls]) (place-channel-put p i) (printf ">>>Place message: a\n" (place-channel-get p))) (map place-wait pls))

Conclusion

- We have demonstrated that it is possible to port a parallel language runtime to a minimal OS
- More generally, we have demonstrated that parallel language runtimes can be supported with only a minimal set of OS functionalities, namely threads and pipes, etc.
- We have also shown that a parallel language runtime can run successfully in a virtual machine environment like Palacios
- The details we have presented are a useful (and necessary) first step in investigating the interaction between such a language runtime and the (virtual) machine

Bibliography

J. Swaine, K. Tew, P. Dinda, R.B. Findler, and M. Flatt.
Back to the futures: incremental parallelization of existing sequential runtime systems.
In *ACM Sigplan Notices*, volume 45, pages 583–597. ACM, 2010.

K. Tew, J. Swaine, M. Flatt, R.B. Findler, and P. Dinda.
Places: adding message-passing parallelism to racket.
In *Proceedings of the 7th symposium on Dynamic languages*, pages 85–96. ACM, 2011.