Osker: A Kernel Written in Haskell
CS475 HW8
Branson Jones
5/3/24

When you think of an operating system, low-level imperative programming likely comes to mind - C, perhaps even assembly language. What if you could just abstract away all of the low level details? This paper discusses Osker, an operating system written in Haskell by researchers at Oregon Health and Science University and Portland State University to provide this question with an answer. Impressively, Osker's implementation "[amounts] to about 1200 lines of Haskell, supported by about 250 lines of C code", although 500 additional lines of C and 150 lines of assembly were added to support booting Osker on a bare machine and to set up paging infrastructure (2005, p. 126). Hallgren et al wanted to apply functional paradigms to avoid many common low-level systems programming pitfalls, such as not having strong typing guardrails or explicit memory safety measures such as bounds checking (2005, 116). They postulated that Haskell would be a strong candidate for this undertaking due to it being type and memory safe, eliminating many classes of bugs. This would allow them to focus on system design enabled by Haskell's expressive type system. In this analysis, we discuss how Osker's implementation turns traditional systems programming ideas on their head.

Process and Thread Management

Osker organizes systems as a collection of domains, which have unique properties. At runtime, the kernel and hardware track a list of all current running domains. Each domain has a distinct user-mode address space that is shared by multiple threads (Hallgren et al., 2005, p. 125). The haskell type definition for a system and domain is displayed in Figure 1 below:

```
type System = [Domain]

data Domain
  = Domain { uproc    :: UProc,
             runnable :: [Thread Runnable],
             blocked  :: [Thread Blocked] }
```

Figure 1: System and Domain type definitions (Hallgren et al., 2005, p. 125).

Each domain has three fields: *uproc*, which holds information about the domains' address space and user code; *runnable*, a priority queue of threads waiting to execute; and *blocked*, a list of threads waiting on I/O operations. Below, figure 2 displays the type definition for an Osker thread.

```
data Thread s = Thread { threadId :: ThreadId,
                         priority :: Int,
                         state    :: s }
```

Figure 2: Thread type definitions (Hallgren et al., 2005, p. 125).

As expected, we find *threadId* and *priority* fields. However, the *state* parameter is particularly noteworthy. Since the information the OS needs about a thread varies depending on its state, they define a custom state type that can hold one of the following types, shown in Figure 3 below:

```
data Running  = Running
data Runnable = Runnable { ctxt :: Context }
data Blocked  = Sending   ThreadId Context
              | Receiving ThreadId Context
```

Figure 3: State type definitions (Hallgren et al., 2005, p. 125).

Each of the options for state stores different thread information. *Runnable* and *Blocked* both store what the thread was doing at the time it was context switched. *Blocked* also notes which thread this thread is sending/receiving information to/from. *Running* doesn't have to store information at all!

This custom state parameter provides many unique benefits for Osker. Firstly, Haskell's type system won't allow the OS to accidentally schedule a *Blocked* thread or exile a *Runnable* thread to the blocked list of its domain. It also makes it straightforward for Osker to understand what to do with each thread. The parameter is also cleverly implemented such that it isn't expected to store information unless it is relevant to the current state of each thread.

CPU Scheduling

Osker schedules both domains and threads. Domains are scheduled from a typical FIFO queue using the round-robin scheduling policy (depicted in figure 4 below), and threads within each domain are scheduled using the aforementioned single-level priority queue of *runnable* threads.

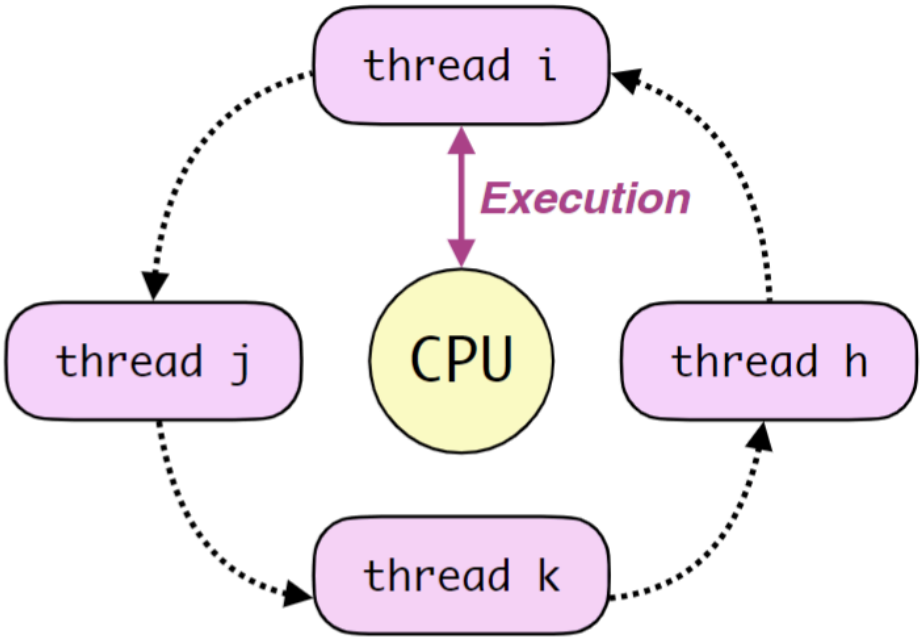

Figure 4: Round Robin Scheduling (David Chiu, 2024, CS475 scheduling slides, slide 20)

Imagine each pink bubble in Figure 4 as a domain, and threads h-k as the thread at the head of each respective domains' *runnable* priority queue. Round-robin scheduling gives all domains a chance at the CPU, preventing domain starvation, but it is unclear how the quantum size is determined. Figure 5 below depicts Osker's thread scheduler, *tScheduler*.

```
tScheduler :: (HMonad m, StateMonad System m)
                => Domain -> m ()
tScheduler dom
 = case runnable dom of
    []     -> return ()
    (t:ts) -> do dom' <-
                  execThread (uproc dom) t
                   `runStateTs` dom{runnable=ts}
                 update (insertDomain dom')
```

Figure 5: tScheduler type definitions (Hallgren et al., 2005, p. 126).

First, the domain scheduler selects a domain from the list of active domains using the round-robin scheduling policy. This domain is passed to the thread scheduler depicted in Figure 5 above, which is responsible for selecting threads to be run. If the selected domain's queue is empty, the domain is not rescheduled since it must be done working or deadlocked. Once the thread at the front of the queue is done executing, the state of the current domain is updated and the next active domain is swapped in.

Physical Memory Layout and Paging

Figure 6, displayed below, is a diagram of an expected physical memory layout using Osker. For simplicity, the system is presumed to run on 32-bit architecture and have 64MB of installed memory.

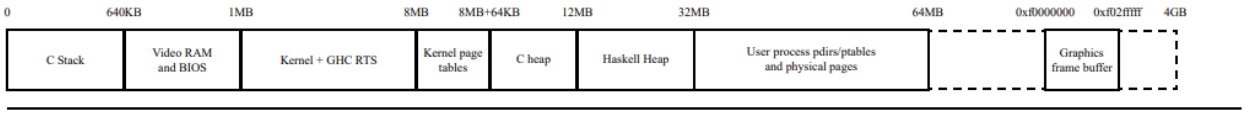

Figure 6. Physical Memory Layout (not to scale!)

(Hallgren et al., 2005, p. 127)

The C stack and heap are used by the startup code and any foreign function calls. The kernel executable contains pre-compiled Haskell code, code for the Glasgow Haskell Compiler (GHC) used to support bare machine booting and paging, and any C and assembly language extensions used by the OS. The Haskell heap stores Haskell data structures that are managed by the GHC's garbage collector.

Osker uses a 2-level paging scheme that is supported by IA32 architecture. Memory protection is enforced by hiding the first 64MB of virtual addresses from user processes and mapping them directly to their identical physical addresses. Only user addresses starting at 256MB are available to user processes. This scheme allows for the kernel to easily access all physical memory and preventing switching to and from a special page map when switching from user to kernel mode (Hallgren et al., 2005, p. 127). An interesting functionality of this paging scheme is that instead of freeing page maps, page maps no longer referenced by the kernel are automatically returned to a pool of available page maps.

I/O implementation

Since most I/O interactions Osker is capable of are fairly simple, the I/O implementation is fairly simple as well. Functions used to read and write to I/O ports are implemented as single IA32 instructions (Hallgren et al., 2005, p. 127). Regions of memory are allocated to I/O as part of Osker's startup procedure using information from the BIOS.

Conclusion

Overall, Osker provides compelling insight into the underexplored realm of functionally programmed operating systems. Clever usage of Haskell's dynamic typing system to implement systems programming processes in a high-level manner serves this implementation well. This implementation also forces us to think about systems programming from a high level perspective - seeing the forest for the trees, so to speak.

Bibliography (APA)

Hallgren, T., Jones, M. P., Leslie, R., & Tolmach, A. (2005). A principled approach to operating system construction in Haskell. *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, 116–128. https://doi.org/10.1145/1086365.1086380