Design and Implementation of Simple Operating System

Ben Lichtman and Michael Yoo
<firstname.lastname@unsw.edu.au>

Abstract

Our Simple Operating System (SOS) is an operating system personality running on top of the seL4 microkernel. It provides basic UNIX-like facilities such as device drivers, file system abstractions, and process management to userspace processes. It also provides abstractions over memory management including memory regions, heap management, and demand paging.

This document aims to explain our design and implementation, their benefits and limitations, and our justification for the choices made.

1 Background

seL4 is a capability based, formally verified, L4 derived microkernel developed at Trustworthy Systems. Our project involved building a functioning operating system on seL4 given minimal framework code.

2 Execution Model

The execution model of an operating system involves how it is designed to handle multiple operations concurrently in order to multiplex the hardware for different operations. For example, the operating system needs to handle multiple user processes, long running syscalls, and device driver interrupts without compromising the liveness of the system.

Our SOS uses the event loop model to handle concurrency in the operating system. We chose this model because it does not require sharing state between threads and does not require complex arrangement of services.

The event loop is implemented with a simple while-switch construct that routes different events to their corresponding handlers.

```
while (1):
  sysno, args := message
  switch (sysno):
    case sys1:
      dispatch_sys1(args)
      break;
    case sys2:
      dispatch_sys2(args)
      break;
    default:
      break;
```

Figure 1: Description of the core event loop and dispatch mechanism.

Many of our operating system functions are performed asynchronously due to requiring I/O in one way or another. In particular, we cannot busy-wait for these operations to complete. For example, an operating system functionality that reads from a file must return control back to the event loop so that the device driver can process incoming packets.

Therefore, for every asynchronous functionality, we divide the implementation into multiple distinct parts that can be called at a later time. When we invoke an asynchronous operation, we pass in a function pointer as an argument, together with a heap allocated structure holding the state needed to resume the operation. These two things combined are called continuations. The asynchronous operation will store the given continuation in it's own continuation to be resumed at a later time, and upon it's resumption, unwrap the continuation from its own and call it to resume the outer operation.

Continuations ensure that our asynchronous functions are executed in sequence only when they have the required data to resume, thus preserving the order of operations as if they ran continuously. Functions that require the completion of an asynchronous operation act as preemption points for the operating system to wait for interrupts or other syscalls (or to execute user processes in the absence of such.)

The operating system maintains consistency across asynchronous operations by ensuring that all asynchronous functions are reentrant-safe. In other words, each iteration of the event loop leaves the operating system in a consistent state such that it can begin handling other events without any violation of global invariants.

When an interrupt, syscall, or a fault occurs, SOS takes control as it runs at a higher priority than the user processes.

There are some limitations with this approach. In particular, having to save the state necessary for resumption explicitly requires us to break down simple loops into a state machine if an asynchronous operation is involved. It also hides control flow and makes refactoring tedious.

3 Timer Device

Our timer device driver uses a tickless priority queue for ordering timer interrupts.

For this, we use a heap data structure implemented in an array for `O(log n)` insertion and removal. To add a timer, we use the absolute time that the interrupt should occur, and add that to the heap.

Whenever the head of the heap changes, we re-calibrate the resolution of the hardware timer to bring the next interrupt to as close to the required time as possible without surpassing it, then when the timer interrupt comes the resolution is set to an even finer grain until it reaches the finest resolution and the interrupt comes just as the timer is requested, the head is popped off the heap and the hardware timer is set to the next target time.

This comes at the overhead cost of having to reset the hardware timer several times for each requested time, however in exchange we get good accuracy.

The heap structure was chosen for its scalability - it would work in a timely fashion even when hundreds of timers are queued, however this does come at the cost of performing heap operations instead of a simple linked list, where insertion and removal from the head can be quick, but sorting and locating timers may be very slow.

4 Syscall Interface

Our operating system services are exposed to userspace through a syscall interface designed to use seL4's IPC facility. seL4 provides fast IPC between threads with the possibility to send payloads with these calls. Basic arguments, such as syscall numbers and integer arguments are passed through this facility. However, because we do not want to unnecessarily copy data between threads and because seL4's IPC facility has a size limit, we pass user space pointers for large amounts of data and let the kernel use it's process page table to scatter-gather user space memory for I/O operations.

Message Register	Purpose
MR 0	Syscall ID
MR 1	Error code or 0
MR 2	Arguments
...	

Figure 2: Our syscall interface convention. The seL4 message registers are the size of a machine word.

Because replying to an IPC is represented through invoking a capability, asynchronous handling of syscalls is trivial by saving the reply capability as part of a continuation, and invoking it at a later time.

5 File System / Mountpoints

Our Simple Operating System provides an abstraction over file systems to userspace. Similar to a UNIX-like system, it provides user processes with open, read, write, and other operations on arbitrary path names over arbitrary backing stores.

Currently, SOS supports a serial file system by using libserial and Network File System by using libnfs. Because of our event loop execution model, both of these libraries must be used in asynchronous mode.

Mountpoints

In order to provide a principled approach for resolving which path names are routed to which file system, SOS provides a mountpoint interface. This interface sits alongside the file system related syscall handlers, and is responsible for resolving a path name to a device type and corresponding device private data.

The device type is used to route to a specific file system, and the device private data usually contains data structures related to the libraries we used to implement the file systems.

Virtual File System

Because our execution model is continuation based, it is sufficient to expose function signatures that file systems must call to resume syscall handling.

File systems that are implemented must take in a continuation with a commonly defined function signature, and call this function at the end of their operation. These function signatures commonly take status codes and bytes processed as their arguments.

File Descriptor Allocation

Each process has a separate copy of the file descriptor table. In order to quickly allocate file descriptors, a bitmap allocation strategy is used (e.g. each bit in the bitmap array represents an allocation.) This has the advantage of being simple to implement, whilst maintaining lowest-first allocation semantics.

6 Virtual Memory

Virtual memory is an abstraction of system memory provided to user space processes such that the process's view of memory addresses and layout does not correspond to the physical layout of memory. Virtual memory is necessary for providing many UNIX-like functionalities such as process isolation, arbitrary ELF file execution, among other functionalities.

Because SOS runs on the seL4 microkernel, many features required for virtual memory are already provided to us. This includes address space management and attaching them to threads, flushing of appropriate architectural state between context switches (e.g. translation lookaside buffer,) and capabilities that can be invoked for maintenance of hardware-resolvable page tables. The management of frames is also provided to us via the project skeleton.

However, virtual memory presents some challenges for kernel-userspace I/O. In order to read from a virtual address provided to us by userspace, we must translate it to a physical memory address. To be more specific, we must resolve the virtual address to a reference on the frame table and to the corresponding address on the physical-mapping address region of SOS.

SOS does not map in userspace pages to its own address space for the reason that this would require implementing virtual memory for SOS itself, which we considered too prohibitive.

Shadow Page Table

To solve the problem of translating user space addresses into physical frames, we implement a 4-level shadow page table, similar in structure to the hardware-resolvable page table maintained by seL4 for the AArch64 architecture, but containing frame references and capabilities instead of physical address translations.

The shadow page table subsystem exposes functionalities to map an arbitrary frame to a user process at a given virtual address location. However, after manipulating the hardware-resolvable page table, it also maintains an internal shadow page table to mirror the changes made. In this way, we can obtain visibility into the process's address space and corresponding frames at a future point in time in the absence of functionality to inspect the hardware-resolvable page table maintained by seL4.

Regions / Mapping On-Demand

For each process, SOS keeps a linked list of valid address regions that a process is allowed to access (e.g. from their ELF file definition) and their permissions.

When a process attempts to access a virtual address not already mapped into its page table, it triggers a virtual memory fault which traps into SOS. From there we check the virtual address against the process's regions to make sure that the permissions of the access match the permissions specified in the ELF file or memory map. If the permission check fails, a segmentation fault occurs and the process is terminated. Otherwise, a correct mapping is inserted to the process's page table at runtime and execution resumes.

Kernel-Userspace I/O Vectors

In order to perform I/O operations on userspace buffers which may be spread out over different frames, SOS provides helper functions that perform scatter-gather operations between a contiguous buffer and a vector of frames. It also supports page pinning and virtual frames which will be explained below.

7 Demand Paging

Our Simple Operating System allows user processes to operate under conditions where there is not enough physical memory to keep all user pages resident in memory at the same time. The operating system uses a swapfile, usually on disk, to evict non-recently used pages to free up physical memory (frames) to be used for other pages. The process of evicting, loading, and managing user pages under constrained memory as demand changes is called demand paging.

Virtual Frame Table

To facilitate demand paging, our SOS uses a data structure called a Virtual Frame Table (VFT). A virtual frame is an abstraction over memory that represents an allocation over a committed amount of system memory. It has a notion of backing, such that a virtual frame can be backed by a physical frame or a swapfile reference.

Once the system encounters a reference to user space memory either through a virtual memory fault or kernel-userspace I/O, it must first ensure that this virtual frame is or becomes backed by physical memory. This operation may involve swapping the page back in to residency.

Pinning

When performing asynchronous kernel I/O, which occurs when swapping in a page or performing filesystem I/O, it must "pin" the virtual frame before using it. Pinning takes a virtual frame, and ensures that it can be dereferenced into a physical frame with the guarantee that it will be backed by physical memory until it is no longer marked as pinned. This invokes the virtual memory subsystem to perform whatever operations necessary to fulfil this pinning, including making use of any free frames, or potentially evicting a page to the swapfile to free up a frame.

Pinning is necessary for asynchronous I/O because we do not want the physical frame currently backing this virtual frame to suddenly represent a different virtual frame, or be swapped out to disk. This would corrupt the content of the virtual frame.

Eviction Strategy

Upon physical frame contention, the second chance page replacement algorithm is used to pick a page to evict. Because AArch64 does not have hardware-maintained reference bits, we implement a pseudo-reference bit kept in each virtual frame. We intentionally unmap pages when the reference bit is cleared so access to them produce a virtual memory fault. Upon SOS trap, we set the reference bit and look up the proper permissions for the fault address, and remap the page with proper permissions.

Swapfile Allocation

Our system uses a bitmap to perform allocation and deallocation for the swapfile, similar to how file descriptors are allocated.

8 Process Management

Our Simple Operating System also provides a basic process management facility, which comprises of syscalls to spawn a new process, list processes, wait for a process, and kill a process. We do not implement hierarchical process trees or parent-child process relationships, however a process that is killed while waiting for another to complete will stay alive until that point.

Maintaining System Consistency

A difficult problem arises when implementing process deletion, whereby the consistency of the operating system must be maintained across asynchronous operations even though the process may already be marked as dead and killed. Replying to such a process, or even interacting with it in any meaningful way leaves the kernel exposed to manipulating garbage data.

To overcome this problem, we implement an uninterruptible state to processes. If a process is under the running state, meaning it is executing its own user code, no special action is required and the process is killed immediately. However, if the process was executing a syscall and SOS is in-between asynchronous operations to fulfil that request, the killing of the process is delayed and the process enters a dying state until we are ready to reply to the process and internal and external states are consistent again (e.g. file handlers flushed, swapfile written.) At this point, the process is terminated and all waiters are notified of the process's deletion.

Process ID Allocation

The allocation of process IDs are done using a bitmap, similar to file descriptor allocation, However, a circular bitmap is used to ensure that the allocation of process IDs wraps around and that we do not immediately reuse process IDs.

9 Conclusion

In this document we have outlined the design of our monolithic OS implemented on top of the seL4 microkernel. This Simple Operating System provides basic UNIX-like functionalities that are sufficient to operate simple programs including a shell program.

Appendix A: Better Designs

This section lists the "Better Designs" criteria that we have satisfied or not satisfied.

Milestone 1: A timer driver

Satisfied:
- Tickless timer

Unsatisfied:
- None

Milestone 2: System call interface

Satisfied:
- Clear framework for handling blocking

Unsatisfied:
- None

Milestone 3: A virtual memory manager

Satisfied:
- As much heap and stack as the address space can provide.
- Designs that probe page table efficiently - minimal control flow checks in the critical path, and minimal levels traversed.
- Designs that have a clear SOS-internal abstraction for tracking ranges of virtual memory for applications.
- Solutions that minimise the size of page table entries (e.g. only contain the equivalent of a PTE and a cptr).
- Enforcing read-only permissions as specified in the ELF file or API calls.
- Designs that defer the actual mapping of pages until they are used (mapping on the fault rather than always mapping on the initial request).

Unsatisfied:
- Designs that don't use SOS's malloc to allocate SOS's page tables (to avoid hitting the fixed size of the memory pool).

Milestone 4: The filesystem

Satisfied:
- Perform no virtual memory management (mapping) in the system call path (for better performance).
- Solutions that support multiple concurrent outstanding requests to the NFS server, i.e. attempts to

overlap I/O to hide latency and increase throughput.
- Better solutions only pin in main memory the pages associated with currently active I/O. Not necessarily every page in a large buffer.
- Avoid double buffering.

Unsatisfied:
- None

Milestone 5: Demand paging
Satisfied:
- Solutions that do not increase the page table entry size when implementing demand paging.
- In theory, support large page files (e.g. 2-4 GB)

Unsatisfied:
- Avoid paging out read-only pages that are already in the page file.
- Solutions that avoid keeping the entire free list of free page-file space in memory.

Milestone 6: Process management
Satisfied:
- A sound strategy for handling waiting on a process that exited quickly (before the call to wait).

Unsatisfied:
- Lazy load the executable or data from the file.
- Mapping read-only ELF segments read-only, and not paging them, but re-reading them from the ELF file