Internal Structure of XOK/ExOS

Parallel and Distributed Operating Systems Group

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02174

exopc-request@amsterdam.lcs.mit.edu
http://www.pdos.lcs.mit.edu

June 4, 2017

1 Introduction

This guides attempts to present a high-level view of the XOK/ExOS source code. The idea is that this guide should make understanding the existing source code easier. However, this guide does not attempt to replace the source code as the prime source of information as to how the system works.

Each of the following sections describes one sub-system such as memory management or disk-multiplexing. Sections about a hardware resource, such as memory management, contain a section on how XOK abstracts this resource and then how ExOS manages it. Other sections are purely about ExOS's emulation of some Unix feature such as processes. This document is still a very rough draft and so significant pieces of what we would like to ultimately be here are not here yet.

1.1 Implementation Notes

This section contains misc notes about the system as a whole. First, don't even try to use anything other than GCC. While almost all of the code is in C, there is still a good amount of inline asm sprinkled throughout as well as some other GCC-isms such as __attribute__ etc.

Second, many of the kernel data structures have been mapped read-only into each process. This avoids having to decide what information to expose to applications and which to hide. This is done by allocating such kernel data structures in a set of pages that are mapped by a single page directory entry (PDE). This PDE can then be inserted into each application's page directory and marked read-only.

Third, each system call usually takes an explicit argument of which environment to operate on and a capability(s) proving access to the resources being manipulated. To save programmer effort and speed system calls, each system call usually has an abbreviated form that defaults to operating on the caller's environment. Thus, there are typically two variants of each system call: `sys_foo` and `sys_self_foo`.

1.2 Source Tree Organization

The source tree for the exokernel has grown large and somewhat disorganized over time. It includes the xok kernel, the libraries (ExOS and others), and most ported applications.

The main subtrees are:

- **bin/** : most of the applications that have been ported to the exokernel appear here. Each gets its own subdirectory (which is listed in bin/GNUmakefile). Adding a new application is fairly straightforward, but does require understanding the basic aspects of the makefiles used in the exokernel source tree. The best approach is to look at an example or two (good ones are: diff and csh). Some high bits include: (1) bin/GNUmakefile should be kept in alphabetical order, except that "shlib" must remain first since the others depend on it, (2) only one real target (e.g., an application program) can be made in a directory, so extra subdirectories should be added to bin/newapp/ to support extras.

- **test/** : some other applications, usually short-term test programs, are located here. The structure matches that of bin/.

- **benchmarks/** : some benchmark programs that have been used for various papers and regression tests are located here. The structure macthes that of bin/.

- **doc/** : documentation on a number of specific topics related to working with the exokernel source tree and system. Under doc/tex/, there are latex sources for two useful get-started documents: this one (internals.tex) on the internal/source structure and one (guide.tex) that discusses configuring, making and booting.

- **ebin/** : applications that can be linked into the kernel as the first program run at boot time appear here. The guide.tex documentation discusses the options further.

- **include/** : most of the system-wide include files appear here. Most of them are taken from (and match) the corresponding OpenBSD files. The ExOS-specific header files are in include/exos/, and the xok-specific header files are in sys/xok/.

- **lib/** : the libraries (ExOS and otherwise) appear here. Most of them are taken directly from OpenBSD. The main exceptions are: alfs, xio, and libexos. alfs is an old version of an application-level file system, which is only used by the Cheetah web server at this point. The main file system, C-FFS, appears at lib/libexos/fd/cffs/. xio is where the TCP/IP code (up to TCP sockets) lives. It is organized in such a way that specializing it to application-specific purposes is strightforward. libexos is where the bulk of the ExOS code lives. libwafer is an example of the minimal things a library OS must do.

- **sys/** : the xok kernel lives here. It is broken into the following main subdirectories: conf/ (autoconfiguration of syscall, interrupt and exception dispatch tables), dev/ (device drivers), dpf/ (the dynamic packet filter; see networking section), kern/ (core kernel stuff), machine/ (xok-specific, xok-internal, machine-specific header files), ubb/ (the XN code; see disk section), vcode/ (dynamic code generation support needed for dpf, xn and wk predicates), xok/ (header files that are both system-wide and xok-specific), xok_include/ (header files that are xok-specific and internal), xoklibc/ (libc-style functions, like string manipulation, for inside the xok kernel).

- **tools/** : a variety of tools related to the exokernel system.

- **tree/** : base files and directory structure that gets installed into the NFS-mounted root partition during a "gmake install". Prime examples include /etc/* files and /home directores (and their contents).

2 General xok Kernel Stuff

This section describes a number of aspects of the xok kernel that apply to many of its subsystems.

2.1 Adding System Calls

Adding system calls to the xok kernel is very easy. To do so, one simply writes the code for the system call and adds one line to `sys/conf/syscall.conf`. Scripts invoked by the makefiles will re-generate the appropriate header files and stubs in order to export the new system call.

The exact format of the lines in `sys/conf/syscall.conf` is as follows (illustrated by example):

```
0x78 loopback_xmit int, struct ae_recv *
```

where 0x78 is the system call number, `loopback_xmit` is the name of the system call, "int" is the return type of the system call, and "struct ae_recv *" is the argument list for the call. Some notes:

- the system call number must be unique (among those specified in `sys/conf/syscall.conf`), which is why we keep it sorted by convention.
- the actual system call name (both as defined in the kernel and as defined at user level) will be augmented with "sys_". So, the example above specifies `sys_loopback_xmit`.
- the system call number (as an "unsigned int") will be prepended to the argument list for the system call. The auto-generated stubs will provide this extra argument (so application writers do not have to know about it), but kernel code writers must accept it in their system call implementations.
- the argument list for the system call can be augmented by adding more comma-separated types.

2.2 Device Drivers

2.3 Capabilities

2.4 Environments

XOK's idea of a process is called an environment[1]. An environment is fairly minimal: it is simply a key-ring of capabilities, a list of up-call entry points where control should be diverted when interesting events occur, a page table, and a collection of accounting info about the process. Further, each library operating system is allowed to store whatever data it finds convenient into the environment's "u-area".

The primary calls for manipulating environments are `sys_env_alloc` and `sys_env_free`.

An environment's page tables must be filled in explicitly and CPU time must be allocated before it will be run. See the sections on memory and scheduling for information on how to do this.

XOK's idea of a process is called an environment[2]. An environment is fairly minimal: it is simply a key-ring of capabilities, a list of up-call entry points where control should be diverted when interesting events occur, a page table, and a collection of accounting info about the process. Further, each library operating system is allowed to store whatever data it finds convenient into the environment's "u-area".

The primary calls for manipulating environments are `sys_env_alloc` and `sys_env_free`.

An environment's page tables must be filled in explicitly and CPU time must be allocated before it will be run. See the sections on memory and scheduling for information on how to do this.

[1] see sys/kern/env.c and sys/xok/env.h for details

[2] see sys/kern/env.c and sys/xok/env.h for details

3 CPU and Scheduling

The kernel multiplexes the processor by dividing it into time slices of a fixed number of ticks. An environment may then allocate time slices in order to be scheduled on the processor. By default, the kernel runs each time slice for the specified number of ticks and then switches to the next time-slice, in order. A time slice may also be marked as sleeping in which case the kernel will skip it and not deduct any ticks from that slice. When the kernel has cycled through all slices in the system it replenishes the ticks of each slice up to a maximum.

A process may donate the rest of its time slice to either a specified environment or to any environment via the `yield` ExOS call[3].

A process may also put itself to sleep by setting it's `u_status` to `U_SLEEP` and *another* process can wake that process up by setting the process's u_status to `U_RUN`. Note the problem that once a process goes to sleep it has to rely on some other process to wake it up. To get around this problem, wakeup predicates can be used.

3.1 Wakeup Predicates

Wakeup predicates are predicates on the system state that the kernel periodically evaluates. When the predicates are true the environment associated with them has its `u_status` set to `U_RUN`. Rather than use the raw kernel interface for predicates you should use ExOS's wrappers.[4]

Wakeup predicates are specified using a simple language with only an unsigned integer type. The predicate is a sum-of-terms in which each term is a product of "value op value" triples in postfix. Each value can either be a literal or the contents of a memory location. The op can be equal, not equal, less then, etc.

Additionally, tags can be associated with each product so that when a predicate evaluates to true and a process is moved from sleeping to runnable the process can tell which product triggered the wakeup. A tag is simply an integer specified with the `WK_TAG` instruction.

4 ExOS Process Management

When an application starts its library operating system must go through what is very similar to a full OS startup in a conventional system. This starts by calling `ProcessStartup` which then does per-subsystem initialization.[5] Further, the very first process to start does special one-time-per-boot initialization for the entire system.

Sometimes different ExOS subsystems want to be able to do their own processing when a process forks or execs (for example, the fd code wants to close close-on-exec fd's and up the ref counts on the other fd's every time a process forks or execs). We handle this with call-backs. You register a function with `OnExec` or `OnFork` or `atexit` or `ExosExitHandler`. The function will then be called back with some useful args when the specified event occurrs.[6]

[3] see include/exos/process.h

[4] see lib/libexos/os/uwk.c, sys/kern/wk.c, and sys/xok/wk.h

[5] see lib/libexeos/os/process.c

[6] see lib/libc/os/oncalls.c

4.1 Epilogue and Prologue code

Each process is responsible for saving and restoring its registers when it loses and gains the processor. The kernel notifies a process that it is about to gain or lose the processor by calling the processes' prologue and epilogue code. The locations of this code is set in the processes u-area.

Currently, this code does things like save and restore registers, check for timer time-outs, and check for signals.[7]

4.2 procd

Hector.

4.3 Process Creation

New ExOS processes are created through the standard POSIX `fork` and `exec` calls. These are slightly complicated by the fact that a process must duplicate itself or replace itself rather than having a nice kernel context to do this from.

4.4 exec

Exec just creates a new environment, loads the new process into it, and then kills itself.[8] The shared library is also mapped into the upper part of the user address space.

Shared libraries are implemented pretty poorly right now. Currently, they are just a program that has every libc function linked in. A stub .S file is then generated that points to the location of each symbol and a program wishing to link against the shared library just links against this stub file. `exec` then makes sure that each symbol is really where this stub file says it is by mapping in the shared library at the right place.

Exec must also run through all the ExOS subsystems and notify them that an exec has occurred so that they can update their reference counts and other data.

4.5 fork

Fork uses copy-on-write to speed forking. It creates a new environment, duplicates the page tables of the parent, and marks every non-shared entry in both environments as copy-on-write and read-only. Then, the page-fault handler in each process handles allocating new pages and duplicating them when a write occurs.[9]

Fork must also run through all the ExOS subsystems and notify them that a fork has occurred so that they can update their reference counts and other data.

5 Memory

This section describes how the kernel multiplexes physical memory. At a later time page revocation and how ExOS implements paging will be added.

[7] see lib/libexos/os/entry.S

[8] see lib/libexeos/os/exec or lib/libexos/os/shexec–I'm not sure which these days

[9] see lib/libexeos/os/cow.c

5.1 Xok and physical pages

Each physical page is described by a `Ppage` structure.[10] This structure tracks the following key information about the page:

- whether the page has been allocated by the kernel, a user level process, or if it's free.
- whether the page is pinned for pending DMA
- whether the page contains a disk buffer
- what capabilities are associated with the page via an ACL

Pages are reference counted if they are allocated to a user-level process. The kernel can hold references to such pages, but the only way a user-level process can hold a reference is by mapping the page. When the last mapping to a page is removed the page is considered free-able.

Process call `sys_insert` to modify the x86 page table associated with the environment. To unmap a page, insert a pte without the PG_P (present bit) set. To map a page, insert a pte that points to a physical page that is not allocated (to allocate it) or to a physical page that is allocated and which the caller presents an appropriate capability for.

5.2 Shared data structures in ExOS

In order for libos's to communicate with each other and share state (such as fd's across forks and the mount table), shared memory is used. Currently, sysv shared-mem interfaces are used to provide this.[11] Due to limitations in our sysv implementation the first process booting must allocate the segments that are going to be used and then other processes may attach them as needed.

Shared memory is being used less and less since it does not give the fault isolation that is typically desired. However, it is interesting to note that while developing ExOS we never ran into an occasion of a buggy libos crashing the system by scribbling bogus data into a shared memory region. The problem we did run into, though, was version control. Every time we changed the structure of the shared memory region every application had to be updated to understand the new layout.

5.3 Memory layout

Processes load at 800000 and their entry point is 800020 (to skip the initial a.out header). Their code, data, and heap all follow. Then staticly reserved blocks of memory begin.[12] After this, optionally, is a read-only mapping of the shared library followed by the private write-able data and heap for the shared library.

Growin down from the high-end of the address space is the kernel which maps all of physical memory high in the address space, then the kernel, kernel data, etc.[13] Then comes the kernel readonly data structures and finally an unmapped page to delineate between kernel addresses and user-addresses. At the upper end of the user part of the address space is the user stack.

[10] See sys/kern/pmap.c and sys/xok/pmap.h for details
[11] see lib/libexos/os/shared.c
[12] see lib/include/exos/vm-layout.h for details
[13] see sys/xok/mmu.h

5.4 Page faults

When the kernel detects a page fault of any kind (writing to RO page, access page not present, etc) it propagates the fault up to a user level fault handler that.[14] ExOS uses this facility to demand-page itself in and to dynamically grow the stack down as needed as well as to handle copy-on-write faults.

6 Disk Storage

6.1 The BC (Kernel-level buffer cache)

The kernel provides a simple buffer cache that applications control. Each physical page of memory in the system is in one of several states depending on whether it is mapped into any env's or not and whether it contains a disk block or not. The possible states are all possible combinations of those two variables: empty/allocated, empty/free, full/allocated, and full/free.

All the kernel buffer cache does is to remember what pages contain disk blocks (whether they are free or not) and expose this mapping to applications. Apps can then probe the buffer cache to find which ppn holds a given disk block and use the normal page mapping calls to then map this page. When a page that contains a block is freed it is still kept in the buffer cache, if the page is not dirty (if an app free's a dirty page that means the app doesn't want its changes hanging around so we dump the page).

Blocks can be inserted into the buffer cache in one of two ways. Either an application can insert arbitrary pages or the app can request the disk to read a set of blocks into the cache.

Protection is done via proxy exonodes. Before an application can insert an arbitrary page into the cache the app must present an exonode that says that the app has write access to the file. Only read access is required to have the disk read a set of blocks into the cache.

6.2 Direct I/O

describe sys_disk_request

6.3 XN

6.4 C-FFS

C-FFS is ExOS's default disk-resident file system. In addition to the fact that it is implemented as a library, ExOS's C-FFS differs from standard UNIX file systems in that it employs cute techniques (described in [**?**]) for improving small file performance. C-FFS has gone thru a number of iterations, and it has departed from the version described in the USENIX paper in order to improve performance and reduce complexity. In particular, embedded inodes are never moved (allowing them to have inode numbers that never change and directly specify their on-disk lcoations), which makes several aspects simpler.

In keeping with the spirit of a library file system, C-FFS tries to minimize its use of shared state amongst processes. By taking advantage of the bc registry, it can piggyback file system-level sharing on the sharing of underlying disk blocks. For example, there is no "in-core" inode table as exists in most UNIX systems. Instead, processes map the bc pages that cache copies of the disk inodes, maintain private tables for finding the right disk inode structures in memory, and work with the disk inodes directly (either via protected methods or via direct write access). Similarly, each process uses its own private logical index (i.e., <dev,fileno,offset>

[14] see lib/libexeos/os/fault.c

to memory location) and relies on the bc registry for a shared physical index (i.e., <dev,blkno> to memory page). Finally, C-FFS makes heavy use of ExOS's software (application-level) critical section mechanism to avoid the use of locks. (This last point remains an issue of debate.)

The C-FFS code base is quite a mess at this point, because it is simultaneously supporting a number of different design options. There are flags for (1) using XN or using the raw disk, (2) using the kernel-resident file system protected methods (see sys/xok/fsprot.c) or using direct write access to update metadata structures, (3) using explicit grouping or not, (4) using embedded inodes or not, and (5) pretending to use soft updates (i.e., simply using delayed writes for everything) or not. In addition, insertion of support for XN (which, itself, is only partially in place) was rushed and is therefore messy. Finally, we have intended to reorganize C-FFS to be more asynchronous and flexible, and the code base reflects a partial transition of this type. Sorry.

All of that said, ExOS's C-FFS supports most of the POSIX file system operations and provides a fairly solid base on which to work. Most of the crucial administrative utilities (e.g., newfs, mount, syncer, fsck) work as one would expect. The three main short-comings are that (1) there is no support for partitions, (2) only one C-FFS file system can be mounted at a time, and (3) XN does not support fsck (and, therefore, C-FFS over XN also does not support fsck). ExOS's C-FFS over a raw disk does support fsck.

6.5 File Descriptors

ExOS uses a VFS like interface between high-level file operations like read, write, close, etc and the underlying filesystem object that a fd represents.[15] Each underlying object specifies an array of function pointers that implement the specified operation allowing new filesystem objects to be transparently used.

Several of these file object methods deserve particular attention because they do not correspond to typical VFS-like operations:

- **select_pred**: This returns a wakeup predicate that will evaluate to true when the object is ready for an operation. The top level fd code collects wakeup predicates from all fd objects that a select is being done on and concatenates them together to form one large predicate that the process blocks on.
- **lookup**: This implements a partial namei. It maps a single pathname component from one file object to another. The source and destination file objects may be different types of file objects if the pathname component corresponds to crossing a mount point.
- **bmap**: Translates a logical block number to a physical block number. Both block numbers are only meaningful to the underlying fd object. For example, an NFS fd object might implement this function as the identity function since NFS has no concept of physical block. This function is used when programs wish to bypass the normal filesystem operations and want to read/write an object's blocks directly.
- **mount**: Associates a mapping from one file object to another. Subsequent lookup operations at this point will return the new file object.
- **close0**: Close operation called on the underlying object every time close is called on the fd. The close method is only called when the last reference to an fd is closed.

[15] see lib/libexos/fd/proc.c

7 Networking

Most of the networking code resides at user-level. The user-level code does two key things: managing routes/interfaces and implementing tcp/ip. The tcp/ip implementation is beyond the scope of this document. Useful examples are in bin/webserver/httpd, bin/videod, and test/xio-demo. The base tcp/ip stack itself can be found in lib/xio. Note that the prime goal of this library is to facilitate building specialized apps. This stack is structued as a series of primitives that each perform a usefull function rather than a monolithic stack.

bin/icmpd/icmpd.c is a fairly simple stand-along example of how to write network programs. It uses DPF, packet rings, routes, interfaces, and the packet sending interfaces.

7.1 Kernel Interface

The kernel implements two sets of interfaces: one for sending packets and the other for receiving packets. On the receiving side, packets are first demultiplexed to the proper owner and then buffered in user-supplied memory in packet rings.

7.1.1 DPF

Each process specifies what packets it's interested in receiving by downloading a packet filter. Ideally, the most specific filter gets a packet if filters overlap. Currently, however, this doesn't quite work. It's not quite clear what happens with overlapping filters at the moment.

Filters are specified by building up a collection of ¡offset, value¿ pairs. The offset corresponds to byte offsets into an incoming packet and value corresponds to the value that must be found. Here's an example that matches IP packets coming in over ethernet:

```
dpf_begin (&ir);
dpf_eq16 (&ir, eth_offset(proto), htons(EP_IP)); /* eth proto = ip */
dpf_eq32 (&ir, eth_sz + ip_offset(destination), ip);  /* ip dst = us */

demux_id = sys_self_dpf_insert (CAP_ROOT, &ir, ringid);
```

`demux_id` is a handle to the filter that can be used in future calls to manipulate the filter. `ringid` specifies the packet ring that matching packets should be copied into. See the following section for more details about packet rings.

7.1.2 Packet Rings

Packet rings are a ring of pointers to user-supplied buffer space. Incoming packets for an application are copied to the next empty buffer. Each buffer in the ring is owned by either the kernel or the application, depending on the value of a flag field in the buffer struct. If this flag is zero the buffer is empty and can be filled by the kernel. Non-zero values are the length of the packet that were copied into the buffer. When a process is done with a packet it should reset the flag to zero.

See bin/icmpd/icmpd.c for a complete example.

7.1.3 Sending Packets

`ae_eth_send` is a simple interface to both the loopback, slow ethernet, and fast ethernet drivers. It does not support scatter/gather at the moment so for demanding apps this isn't what you want. `sys_ed_xmit`, `sys_de_xmit`, `sys_loopback_xmit` are system calls for sending to each interface and which support the full capabilities of each card. The xio tcp stack is a good example of how to use `sys_de_xmit`.[16]

7.2 Routes, Interfaces, DNS, ARP, etc...

This section describes the misc. stuff involved in networking. XOK supports multiple ethernet interfaces and exopc comes with the typical ifconfig, route etc tools along with an arp demon and DNS resolver.

- **Interfaces**: During boot up, the kernel initializes the network cards, and the sysinfo structure. The sysinfo structure contains the number of slow and fast ethernet cards that the kernel recognized.

  The IP address of the machine will be determined by the first process. This process then proceeds to initialize a global interface table that contains basic "ifconfig" information, like physical interface number, ip address, ethernet address (cached from sysinfo), netmask and flags. Programs use this information to know if an IP address is on the local LAN or remote; if it is on a remote location, then they look in the route table for the route.

- **ARP**: The first process spawns an arp daemon. There is a global arp_table mapped into each process. Only the arp daemon can write to this table. Processes check the arp table to find a translation IP-¿ether. If not found, they ipc into the arp daemon to add the entry, and block until the status of the entry changes. The arp daemon will retransmit the arp request until it gets a reply or times out. The arp daemon also responds to arp requests from other machines in the network.[17] There are basically two functions user-level programs need to know about:[18]

  - `arp_resolve_ip(ipaddress,ethaddress for result, interface)`
  - `arp_remove_ip(ipaddress)`

- **Routes**: The route procedures are used to find out which is the proper interface for the ipaddress. All process have a route table that is in shared memory. Each process can read and write into this table. The first process sets the routes by default.[19] The "route" program can also be used for this purpose. It is important to set the routes early on, as most libos code uses this to locate which interface to use. More information as well as interface definitions can be found in the source.[20]

- **DNS**:

  ExOS uses the same routines used in OpenBSD. You just have to make sure that all the networking code is ready by the time you need to resolve your first address.

[16] See lib/xio/exos_net_wrap.c
[17] see bin/arpd/arpd.c and libexos/net/arp_table.c
[18] see include/exos/net/arp.h
[19] see ebin/rconsoled/setup_net.c
[20] see include/exos/net/route.h and libexos/net/routenif.c

References

[Ganger97] G. Ganger, M.F. Kaashoek, "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files", *USENIX Technical Conference*, January 1987, pp. 1–17.