DRAFT: Requirements for a XymbiOS kernel to meet industry standards

Wojciech A. Koszek

05.08.2008

Abstract

XymbiOS is the code name of the operating system kernel which needs to be designed for the ParalleX execution environment. The novelty of the ParalleX design and the timeline for which the project is targeted (year 2020) makes the whole design process complicated. It requires the set of final requirements to be specified very tightly, so that the top-down design methodology could be used within the implementation process. On of the goals for XymbiOS is not only to manage highly parallel computer architecture in order to achieve extreme performance gains, but also to create enough abstraction to make it usable in computation-intensive production environments.

In this short draft paper I present things which industry standards require from the OS functionality nowadays. Since the hardware support is strictly related to the functionality that OS can or can not provide, in the first section I propose changes which I believe will have to be brought to the ParalleX architecture design in order to make OS implementation easier. Second section is both the introduction to the system requirements and introduction to my view on UNIX, it's success and a reason why it made research a bit stagnant. Later, I propose two approaches of distributed operating system, with all their advantages and disadvantages. The rest of this document is a slide between topics known to cause problems in OS development both with proposal of eventual solutions in ParalleX context. At the end, I present several projects which I think are worth looking at and which could help a ParalleX group to establish official XymbiOS document related to it's requirements and proposed design directions.

Each section containing questions has been discussed with ParalleX group and a lot of useful responses have been provided. However, I decided to leave them here for a matter of documenting them, just not to get them lost. Some of questions repeat between sections, since some of them seem to touch more than one subsystem. I try to answer each question separately within a context of discussion.

Proposed hardware changes

I would like to propose several improvements which can make OS implementation possible and more reliable.

Hardware parcel handling and software parcel handling.

By software parcel handling I mean a mechanism that requires software assistance in processing. Receiving specific types of parcels should involve immediate processing in exceptional manner – just like we do in conventional processors, when important event takes place, and a stream of processing needs to be interrupted. System calls and traps can be an example of such an exception, which needs to be handled soon after happening.

Hardware parcel handling should be designated to the work that can easily be dispatched on the hardware side. Let me just make an example:

Two parcels are being sent to the X node. First parcel creates a thread that starts actual work. Second parcels contains “Migrating isn’t allowed” request. Second parcel is immediately dispatched, even when still lying in a parcel queue. NO-MIGRATE “flag” within a processor is set and until processing finishes, if the other parcel with a request of migration is received, automatic response is generated informing the other node, that the thread can’t be migrated from X.

The reasoning between distinguishing between two types of parcels handing methodologies are as follows:

- Context saving in case of a need to handle an exception is costly
- Messages send through parcel mechanism can happen very frequently and involving software in every parcel handling isn’t needed
- Software won’t help in generation of a response to some of the messages. Let say following have been sent to the node: “Are you free”, “Can you take a parcel from me”, “Checkpoint”, “Suspend”, “Resume”, “Can migrate”, “IO to node XZY”. All of those questions that might be present in CPU protocol can be answered directly by bitwise analysis of CPU state, and parcels can be generated without a need to invoke a software.

Hardware performance counters

Just to make sure ParalleX is profiled in every development stage, I believe there should be an assistance from the hardware side of some sort in terms of code profiling. The way in which we could distinguish which kernel thread state belongs to which user-level thread is to be discussed.

My proposal is based on the fact that a lot of improvement in current systems code from the possibility to see what actually happens on the CPU. Not only it can

act as a performance measurement sandbox, but can also be used in real-time, to help the system to figure out, how to affect system functionality to gain maximum performance (e.g. scheduling).

Introduction: the UNIX problem

UNIX is operating system deployed widely within the industry. Great design let engineers to come up with number of implementations, which are used till know. Opinions expressed in this article are based on my experience related to FreeBSD, Solaris and Linux kernel, which I also view as “UNIX”.

Since the time of it’s introduction, UNIX has became some sort of a problem in the operating system research.

Software systems research is irrelevant

There were number of experiments within the OS community that targeted to repeat this success, but none of them really survived in untouched form to these times.

The UNIX environment thanks to the POSIX started to define levels of usefulness. Some of the ABI calls should be changed to match real-world needs, but in my opinion that it’s pretty amazing that UNIX API become pretty much the same, when industry needs tend to change. Later, I note some facts which I think are the reason that let the API to as mature as It is right now. I think several concepts from the UNIX functionality and architecture should act as important hints for the future of XymbiOS and ParalleX in general.

The UNIX way

I believe that a lot of the UNIX success comes from the unification and simplification of the system. The concept of having everything represented by a file of different types has shown to be useful and right. This is why I think representing every request by a parcel of different type, which is going to be dispatched appropriately is right. Just like UNIX has it’s own “protocol” of communication between user and kernel space, XymbiOS could have it’s own unified way to communicate between node’s boundaries.

Having the same API to work for a textual file, stream or device just works. We do have the same identifiers both for file descriptors and sockets . This is why it’s possible to tell:

READ 100 BYTES FROM THIS DESCRIPTOR

And don’t really care about what is really represented by particular descriptor.

The concept could change to even further unify the OS structure. Right now calling socket-specific calls on a descriptor that represents typical, ASCII file will return an error. Going to the direction in which Plan9 went could be interesting. Having everything represented by a typical path name would let us to stay away from distinguishing the networking operations from a typical, file system related ones. Since every device in ParalleX is covered by PGAS and reachable via parcel communication mechanism, it is believed that access to disk and memory mapped device will look the same. Some ideas from UNIX resulted in my proposal of unified exception handling presented later.

System architecture

Running operating systems on huge number of processing elements lets us to pick two approaches. One is that each CPU has it’s own operating system, and whole kernel-side specific processing is done on that particular CPU from the point of view of the operating system.

Another approach exercises the possibility of having one operating system that manages all the processing elements by scheduling thread all over the available cores.

Each of there approaches has advantages and disadvantages.

One core == One XymbiOS instance

With each CPU having it’s own microkernel instance, we could possibly simplify the OS code to the minimum: kernel level, machine-dependent parts would be responsible for handling communication (via PARCELS for example) and relay system-call requests to upper layer software. Debugging would be minimized, since the only thing which would have to be figured out in case of a problem is a state of particular core and a state of a thread running on it. Since the thread level processing within the ParalleX is going to be minimized to the minimum, if only thread level processing granularity (piece of the work that is to be done) could let us to do so, this model could be very effective from the programmers stand point. Such a model brings a lot of questions on system consistency: how to manage VM layout, so that each XymbiOS instance sees “the same” object all over the PGAS.

Q: How such a system could keep information about particular object lying in the DGAS consistent?

Hardware parcel handling could be used as a way to keep the information between VM views (view of the VM from each XymbiOS instance).

Q: How to preserve the protection checks within the DGAS? For example, since DGAS can contain both the general purpose processors as well as high-speed cryptography hardware, how one can prevent from accessing to the unwanted resource, when each XymbiOS has private view of the system. Is some sort of global container where such a global attributes like security privileges needed?

SUGGESTION: Could security model look like this: no matter what the LOCAL core does in it's LOCAL memory, it is seen as being "privileged" and access is granted. If the LOCAL core want to access the REMOTE range of addresses, it has to ask for a permission. It's up to the REMOTE core to receive a request (parcel?) from the LOCAL node and perform privilege check, and if the check succeeds, response parcel is generated, and access to the REMOTE memory is granted.

Thousand of cores == one XymbiOS instance

Another approach would be to treat XymbiOS as a central manager for processing across all available processors. One XymbiOS would have to take care of the management of security, VM subsystem, scheduling and I/O. Number of advantages come from this design as well – having one central part of the system make VM management easier.

Q: Probably VM consistency could be kept away from the system?

Once hardware parcel handling will be developed, keeping consistency across the system should be a problem any more.

Q: How would the system software interact with VM subsystem in this model? Right now RISC processors designate some set of handler related to TLB, cache hit/miss processing and it's up to the programmer to choose the right way of management.

Some events like an access to privileged DGAS address must have a way to signalize an error to the software. Access to the memory range within a locality can be resolved by a controlling node itself. In case of a problem, I believe it should create a parcel directed to itself. This is what I call UNIFIED EXCEPTION HANDLING, and what probably deserves a better name. The key concept is that no matter if the access is local or remote, exception handling could be implemented entirely by parcel dispatching.

Q: Is there any known concept of "sending" a parcel from the node to itself?

Low-Level startup

Typical operating system designates one CPU to lead the startup procedure based on a machine-specific list of steps. Later on in the booting process, once the "master CPU" has already bootstrapped itself, the other CPU elements are woken up. The initialization procedure is iterative, which means that the time increase is linear to the growth of the number of cores. It's not a problem for 4 or 8-core

machine, but it might appear as a problem for large number of processing elements within the ParalleX.

I believe some startup in “lazy” manner need to be exercised, in which only those cores that are needed for computation are woken up.

Q: What happens if the XymbiOS has an error and “machine” reboot is required.

Since ParalleX is not targeted to the environments where frequent, typical reboots are required, or needed at (all) I define “reboot” as the last sort of operation, which can be useful in case of machine/programmer error, where bogus data/computation has been spread out all over available nodes. Since specific group of nodes can be “polluted”, I believe each node running under XymbiOS control should have a way to “restart” itself once “REBOOT NEEDED” request is received and should be able to redirect it to another nodes as well. Such an event (parcel) should be handed through hardware parcel handling mechanism, since software might be itself a reason of failure.

Q: How the suspend/resume (also known as check pointing) works for the ParalleX computer?

Software trap could be used to jump from typical mode of processing to privileged mode, as we do on conventional architectures. The reason for jump into “privileged” mode is that actual state for suspending must contain resources unavailable in typical, unprivileged mode of execution. Once such state could be packed as a data parcel, it could be directed to the I/O device for check pointing.

Mechanism known from Xen virtual machine monitor could be taken upon expertise as well. Node to be suspended could loan the memory access to another node placed nearby. Since hardware parcel handling could transparently arrange memory mappings between nodes, it could also let the another node to access local memory without invoking permission exceptions. Later, such a privileged node could simply take remote node’s memory and pack it so that it could be easily send to the mass storage media.

Q: How should such a mechanism encode an address, to which parcel with this suspended data could be directed?

If “Copy the whole node’s state through another node” approach is chosen, directing data isn’t a problem, since it would be up to the “copying node” to pick the destination place of the suspended data. Otherwise, node would have to know, where to sent it’s saved context.

Small-percent of cores running XymbiOS; the rest for user-level jobs

Light-weight, short-term threads requiring assistance from the kernel-level code could benefit from being run on the same core to improve ratio of cache hit in order to increase code execution performance. My major concern related with short-term threads Is that cache-related issues might become an bottleneck, since we can't actually say at this stage, what the execution characteristics will look like (how many remote memory accesses will particular thread need, how often it will have to communicated, what's the cost of interrupt/exception handling). Separating operating system activities to only small subset of general number of core MAY have some positive effect:

- Kernel running on separate cores wouldn't have to run in privileged mode, since all communication channels would be related with parcels, which can easily be controlled in the kernel space
- Cache hit ratio on both the "kernel cores" and "user-level threads" would be improved
- Designating specific cores to specific task would let some kernel subsystems to perform aggressive optimizations, since they'd have more uninterrupted execution time

Main problem in this approach is that I'am not aware of any previous work which would let me to study eventual advantages and disadvantages.

Q: How would user-level cores reliably signalize the "kernel cores" that some important events needing urgent actions had just happened and how such a remote "kernel core" could react to it?

Debug ability

Ability to debug the OS kernel is very important factor , especially in the implementation procedure. No matter if the architecture is concentrated around microkernel concept or the monolithic one, the mistakes do happen, and having way to track them saves a lot of time in case of a problem.

For fairly complicated environment to which we can count ParalleX in, I believe the presence of some sort of kernel-level debugging support is mandatory.

The above comment comes from the fact, that the major advantage in writing operating system code for moderately decent UNIX systems is the availability of a kernel debugger. The major advantages of having debugger include:

- Seeing the processor state

- Seeing the system state from the period of time just before the crash happened
- Being able to list executing context of either the processes or threads

I think that Linux development model really suffers from not having a debugger, and it was actually strongly criticized in the past. Historically, there were number of complaints from the Linux kernel developers for not having an official kernel debugger. This situation hasn't changed so far, even that two unofficial debuggers exist.

Mach microkernel had kernel-level debugger from the beginning, and the code present in the FreeBSD operating system is derived from their sources. It simplifies bug tracking a lot.

VM subsystem

ParalleX makes use of the notion of Partitioned Global Addressable Space. Interesting part of XymbioOS design will be designing the VM subsystem, since the overall architecture of the ParalleX is targeted to the environments, where amount of memory will be extremely big.

Q: Whether each XymbiOS has it's own VM subsystem and each XymbiOS instance has to pay attention on consistency, or rather we have some sort of central VM view?

Q: What happens, when one instance of XymbiOS gets a request like:

> Allocate 100000….. bytes for me

and such a request can't be fulfilled with local DRAM memory. Do we send a parcel to another node about such a request, or do we migrate execution context to the other node?

Both such requests should be possible, and it would be up to the allocation policy to choose which is better.

Q: Why parcels couldn't be used to handle consistency model across the whole DGAS space as well?

Since each CPU is going to have information about a memory which it's connected to, I believe XymbiOS instances could share information about memory consistency just like people share information about their memory with each other. Having hardware parcel handling should make such operation possible.

Q: Will this be optimal operation and whether there will be urgent need to inform local core on what actually happens with remote (other node's) memory?

Protection

Protection in the UNIX environment consists of several parts:

- Process address space separation
- Process owner separation
- Kernel space/user space distinction, with the later being less priviledged than the first

Q: Will the protection model include switching between process and kernel mode?

Since the prototype is being implemented on conventional architectures, there is going to be this split between both privileged and unprivileged execution modes. Some sort of protection will have to be implemented in native ParalleX architecture as well, since one buggy process cannot disturb the another one.

Q: What the cost of the switch might be? Right now, in conventional processors, the cost is becoming a problem as the more and more processors is having a huge number of resources (registers) that need to be save before the execution context can be switched.

Q: How to handle security checks within the DGAS?

Scheduler

Q: Will the XymbiOS kernel run on particular number of processing elements and make use of the others, or rather to run on each particular core.

Typical cases of thread preemption in 1:1 mapping scenario (one kernel thread mapped on one user-level thread) requires full context switch. Cost of operation like that is significant. Such model can be useful for real-time execution, which falls beyond topic of this document.

However, in my opinion possibility of performing an execution in such a way should be reconsidered, since some computational tasks making heavy use of caching could benefit from it. Simple but frequent actions like copying/filling a memory could be implemented that way. Such a core could simply act as “offload” processor for small, specific set of tasks.

ParalleX execution environment uses 1:M mapping policy (one kernel thread serving as a window to M user-level thread). Problem with 1:M lies in lack of user-level scheduler, which could work in conjunction (or completely separately) with kernel-level scheduler on picking the most appropriate user-level threads for the execution. Each user-level thread should be able to forcibly preempt other thread running on other node in such a way, that come back to the previous execution

thread is possible. The problem in typical preemption lies in a need to save a full processor context, which requires significant amount of time, especially in terms of user/kernel level privilege switching. Doing it in conventional way makes the whole concept useless. In order to make light-weight thread scheduling worthwhile, several functions must be implemented:

- User-level preemption without kernel assistance and without mandatory privilege switching; similar work in this area has been done in Exokernel project cited later
- User-thread synchronization without kernel assistance

Proposal

On conventional architectures like x86-related processor switching between 0^{th} and 3^{rd} ring is the most expensive. It would be worth to experiment with possibility to run privileged code somewhere in between, so that switch cost wouldn't be so big.

Q: Why couldn't OS kernel be implemented as a layered software structure consuming two protection levels at the same time; kernel protection would still require full context switch, but "intermediate user privilege level" could work with only necessary state saved; it could even have It's own exception handlers not needing kernel-level interaction.

Device architecture

Since the DGAS is going to cover all hardware peripherial that is to be a part of ParalleX, devices will have to appear as other "active" elements of the system. Such an idea is motivated by the fact, that other approaches has already been exercised (e.g. x86 I/O ports) and it appeared to work much worse than memory-mapped devices in RISC-alike System-on-Chip fashion.

The main reasoning of placing "typical" devices in GAS is that any other core in ParalleX on which the thread may be executed must have a way to reference to particular devices, since work can be migrated between the execution units in case of resource shortage.

Q: How doing something really simple (reading a keystroke from a keyboard) should look like?

Other approach would be based on hardwiring devices to strictly-specific processing elements but this way could not work in reliable way: placing high-density storage close to only one core and relying on it's permanent availability sounds wrong, . It dramatically decreases fault tolerance, which is unacceptable.

Since the ParalleX execution environment is supposed to contain heterogeneous processors, I believe devices put in GAS will behave (more or less)

like other I/O devices: sending a parcel to a device is just like sending a parcel to a processor. The only major difference is that device-specific parcel should be created with device driver that is aware of requirement of particular device. The fact of reception of a parcel would be guaranteed by a method used to connect a device to system bus. It would be up to the device to dispatch a parcel and involve necessary processing.

Having a mechanism such as this described above would let to easily migrate from the prototype working on conventional processors to fully featured ParalleX working on it's dedicated hardware platform. Within a prototype working on x86 processors, sending a parcel to a device is nothing more like sending a parcel structure to a region, in which the device memory is mapped. By doing so, parcel dispatching code on FPGA board could be exercised. Later, in native architecture, sending a parcel and dispatching it on a device could become entirely hardware-related operation.

Problem exists in handling high-priority devices, like realtime-clock. One of the solutions proposed by Maciej Brodowicz is based on a fact that most types of computation performed in ParalleX won't actually require hardwired clock signal to perform useful action, and that only those nodes that require clock interrupt to operate correctly such have an access to it.

Another problem is related with a routing of parcels within ParalleX architecture: keystrokes typed on a keyboard, storage I/O or data to be put on screen/terminal each have different priority. It's believed that "NOT ABLE TO RECEIVE" or "RELAY TO NODE X" parcels should have a higher priority than "I/O READY" parcels. The fast routing of these messages is left unexplained here and is going to be a part of research.

Interrupts, exceptions and system calls

I think parcels can be easily used to implement all necessary event handling methods. I define "interrupt" as an event of high priority that has to be delivered immediately to the node. It can happen in asynchronous way. Exception is an event caused by a user interaction and has to be delivered in synchronous manner. System call is an event generated through the application and is a request of some type. Priorities of those events won't be discussed here. However, being able to redirect parcels to other nodes in the ParaleX would let the XymbiOS instances to perform a kind of load balancing, so priorities would let the architecture to pick the closest node in neighborhood.

Being able to receive all of those events as parcels would let the OS designer to unify interrupt, exception and system call handling and treat it like "just parcel dispatching".

Proposal of OS requirements

Several POSIX-compliant systems share some functionality within particular subsystems as well as in the system itself. I try to outline those, with simple description of advantages coming from their implementation.

General architecture

Type-aware communication between user/kernel domain

The major problem in communication between user space code and kernel-level privileged code in UNIX is that they lack proper type checking. There is no proof, that data packed as “unsigned long” in the user space application is received as “unsigned long” in the kernel space.

The proper checking of data types lowers a probability of software bugs. Since the values passed from the user space are often crucial to the security, proper type checking increases security as well, since there is strict enforcement of what is to be fetched on the kernel side. For example, in abstract API written in ANSI C language, sending “unsigned long” variable named “string_length” to the kernel could look like this:

```
PxValSend(PXVAL_UNSIGNED_LONG, “string_length”, &str_len);
```

On the kernel side, receiving the same variable could look like this:

r = Px ValRecv(PXVAL_UNSIGNED_LONG, “string_length”, &str_len)

With r being equal to -1 if there were no “string_length” of “unsigned long” type. Such a consistent communication mechanism makes it impossible to misinterpret the variable types.

Process/job/thread identifiers

Without going into much detail I propose implementing some kind of unique identifiers, which could help to easily reference to particular execution object.

Fetch information from the remote end

Almost all I/O within UNIX is implemented on “file” descriptors – internal structures unifying system’s architecture. In order to abstract low-level view from the media-specific implementation details, no medium-specific details are carried up. However, there are situations in which such a knowledge of remote/local side parameters is crucial.

In UNIX, when descriptor references a file placed of a physical media, querying can be done by “fstat” system call. Information returned include data size,

file modification/creation/last access dates, and other data relevant from file-systems standpoint. When I/O object is a socket, querying can be done in two ways. One is querying for current state of a local side of the connection. It is done by “getsockname()” system calls. As a result, programmer is able to fetch both an IP address and TCP port’s number, which gives him valuable data for further processing. The second call – “getpeername()” is able to return the very same information, but for the remote side of connection. Thus, communication channel can be identified with two calls.

ParalleX could also implement a possibility of identifying a communication channel, since a knowledge of local and remote sides of the connection in such a dynamic system will be needed pretty often. Having a way to reference the remote node in the middle of communication will be important. At a debugging stage local node might simply want to ask about remote node’s status. At computation time, information about remote node can be useful to perform conditional actions.

It’s an implementation detail whether “optimized neighbor discovery” scheme should be done in software of hardware.

Memory allocation and memory management

Some scheme for memory allocation must be implemented. In latest years, typical trend was to migrate to slab-based allocators, where we allocate/destroy and garbage collect data structures of the same characteristics. Garbage collecting within the group of slabs of the same size and destination place can be easily simplified, since management of small, unallocated number of slabs is much easier than managing huge number of slabs of varying sizes. It helps to predict cache behavior as well, since probability of cache misses when slabs are lying nearby is lowered. Requests to the memory allocator can be easily analyzed and dispatched to specific processing code, which is “slab-specific”.

Copy-on-write

Copy-on-write mechanism is used nowadays in fork() system call implementation. Fork() within UNIX environment is used to replicate a current process which is being executed, so that further execution of the code can follow concurrently. The copy of an original process (so called a “child” process) is nearly identical to it’s parent, with the minor exception of process ID and other system-specific fields that have to be changed in order to identify those processes in unique way within the system.

I speculate that such mechanism could be useful in the ParalleX environment as well. Copy-on-write could improve backtracking algorithms execution significantly, where some part of the computation can be continued with changed parameters on another nodes. Copy-on-write could mean redirecting a “work” request to other nodes with proper parameter preprocessing both with beginning to execute the work on the node.

Memory-mapped I/O

If the idea of referencing to every I/O device through the parcel mechanism will be deployed, this will mean that memory-mapped I/O is already implemented. Otherwise, way will need careful reconsideration, since memory-mapped I/O is widely used in nowadays' applications with big success. Because it's possible to tie the underlying object (piece of dynamic memory, file placed on the permanent storage, memory mapped to the device) things like event logging or data gathering can be implemented in universal way. So no matter if a service for statistics gathering wants to aggregate 1TB of data and processes it later, or do it in real-time or use some special hardware device for offloading, it can do it with only one API. The functionality is based on how the service has been configured.

Some sort of protection should be implemented as well as a part of the VM subsystem and it's interface, so that mapping resources in read-only way could be possible. This is achievable in the POSIX through madvise() system calls nowadays.

Multiplexed I/O

Several mechanisms for multiplexing I/O exist in the UNIX programming environment. Lack of standardization procedure on time caused several versions which appeared in various operating systems, but poll() exists in each version. The most important function's parameter is array or structures, each of which contains a descriptor referring to the object which we want to perform I/O on, and a descriptor of an action we want to execute (read/write). If any descriptor is ready for the I/O, poll() returns and lets the programmer to start the I/O process. This mechanism is useful, when we want to wait for user interaction, fetch data from the network and log an activity at once.

Having this mechanism in XymbiOS should be taken upon consideration.

Proposed parcel messages

Below, I propose several (pretty obvious) parcel messages. Those can be either exception/interrupt/system call notifications. I assume we can handle parcels by software and hardware.

CREATE A THREAD

Since threads are going to live on CPU for a small portion of time, I believe destroying a thread can be occasional operation. I propose non-standard way of removing a thread if it appears to be absolutely necessary: creating a thread with No-Operation instructions and overwriting previous code should do the job.

DESTROY THREAD

See "CREATE THREAD" section.

MEMORY/PAGE FAULT

SYSTEM CALL

HARDWARE-specific MESSAGE

Handled entirely in the hardware.

Proposals from the open-source community (a bit humorous)

FreeBSD project has one internal, developer-only mailing list, where messages with information and personal content can be sent. Before coming to Baton Rouge I let FreeBSD people know that I'll have a chance to visit Louisiana due to the internship in Thomas Sterling's group with brief description of what ParalleX and XymbiOS will be in the future. The overall suggestion which can help a project of that size was to choose good software license, so that both academic and industry people could profit from valuable research in HPC field.

Proposal needing consideration (interesting and helpful projects)

LLVM

Both FreeBSD and Apple Inc. developers are taking a close look at project called LLVM:

http://llvm.org

LLVM is Low Level Virtual Machine Compiler Infrastructure project. Right now it comes with C/C++ compiler, which can actually be used to compile FreeBSD kernel, and is used by commercial companies (Apple). It seems to have great infrastructure in terms of frontend/backend construction. Together with huge number of code optimizers and universal Intermediate Representation it makes this project look very interesting in terms of programming-language playground.

OSKit

OSKit was a research project from the University of Utah. OSKit is a set of libraries, that makes operating system development really easy. It was possible to write a simple micro-kernel with just a few function calls. Before running on the bare hardware, It was possible to compile your kernel as a typical UNIX-alike application, so that debugging cost could be lowered to the minimum. Implementing simple "OS" running in privileged mode on x86 that could have a control over interrupts

both in single and multiprocessor system wasn’t a big deal, as it is when implementing everything by hand.

Unfortunately, OSKIT is proved to not work nowadays with modern tool-chain (mostly due to GCC changes).

EXOKernel

Exokernel is the old MIT research project:

http://pdos.csail.mit.edu/exo.html

Was mainly targeted to loan as much of the OS functionality control to the user-space applications as possible. It also included resource allocation management and resource control. Some research has been made on lightweight exception handling as well. It’s worth revisiting before making final design decisions.