# Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

## Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

# Asynchronous System Calls

Genesis and Status

Zach Brown
*Oracle*
zach.brown@oracle.com

## Abstract

The Linux kernel provides a system call interface for asynchronous I/O (AIO) which has not been widely adopted. It supports few operations and provides asynchronous submission and completion of those operations in limited circumstances. It has succeeded in providing cache-avoiding reads and writes to regular files, used almost exclusively by database software. Maintaining this minimal functionality has proven to be disproportionately difficult which has in turn discouraged adding support for other operations.

Recently Ingo Molnar introduced a subsystem called *syslets* [3]. Syslets give user space a new system call interface for performing asynchronous operations. They support efficient asynchronous submission and completion of almost every existing system call interface in the kernel.

This paper documents the path that leads from the limits of the existing AIO implementation to syslets. Syslets have the potential to both reduce the cost and broaden the functionality of AIO support in the Linux kernel.

## 1 Background

Before analyzing the benefits of syslets we first review the motivation for AIO and explain the limitations of the kernel's current support for AIO.

### 1.1 Briefly, Why AIO?

AIO can lead to better system resource utilization by letting a single process do independent work in parallel.

Take the trivial example of calculating the cryptographic hash of a very large file. The file is read in pieces into memory and each piece is hashed. With synchronous `read()` calls, the CPU is idle while each piece is read from the file. Then as the CPU hashes the file in memory the disk is idle. If it takes the same amount of time to read a piece as it takes to calculate its hash (a bit of a stretch these days) then the system is working at half capacity.

If AIO is used, our example process fully utilizes both the CPU and the disk by letting it issue the read for the next piece without blocking the CPU. After issuing the read, the process is free to use the CPU to hash the current piece. Once this hashing is complete the process finds that the next read has completed and is available for hashing.

The general principles of this trivial example apply to more relevant modern software systems. Trade a streaming read from a large file for random reads from a block device and trade hashing for non-blocking network IO and you have an ISCSI target server.

### 1.2 KAIO Implementation Overview

The kernel provides AIO for file IO with a set of system calls. These system calls and their implementation in the kernel have come to be known as KAIO. Applications use KAIO by first packing the arguments for IO operations into `iocb` structs. IO operations are initiated by passing an array of `iocb`s, one element per IO operation, to `io_submit()`. Eventually the result of the operations are made available as an array of `io_event` structures—one for each completed IO operation.

In the kernel, the `io_submit()` system call handler translates each of the `iocb`s from user space into a `kiocb` structure—a kernel representation of the pending operation. To initiate the IO, the synchronous file IO code paths are called. The file IO paths have two choices at this point.

The first option is for a code path to block processing the IO and only return once the IO is complete. The process calling `io_submit()` blocks and when it eventually returns, the IO is immediately available for `io_getevents()`. This is what happens if a buffered file IO operation is initiated with KAIO.

The second option is for the file IO path to note that it is being called asynchronously and take action to avoid blocking the caller. The KAIO subsystem provides some infrastructure to support this. The file IO path can return a specific error, `-EIOCBQUEUED`, to indicate that the operation was initiated and will complete in the future. The file IO path promises to call `aio_complete()` on the `kiocb` when the IO is complete.

The O_DIRECT file IO path is the most significant mainline kernel path to implement the second option. It uses block IO completion handlers to call `aio_complete()` after returning `-EIOCBQUEUED` rather than waiting for the block IO to complete before returning.

### 1.3 KAIO Maintenance Burden

The KAIO infrastructure has gotten us pretty far but its implementation imposes a significant maintenance burden on code paths which support it.

- Continuation can't reference the submitting process. Kernel code has a fundamental construct for referencing the currently executing process. KAIO requires very careful attention to when this is done. Once the handler returns `-EIOCBQUEUED` then the submission system call can return and kernel code will no longer be executing in the context of the submitting process. This means that an operation can only avoid blocking once it has gotten everything it needs from the submitting process. This keeps O_DIRECT, for example, from avoiding blocking until it has pinned all of the pages from user space. It performs file system block offset lookups in the mean time which it must block on.

- Returning an error instead of blocking implies far-reaching changes to core subsystems. A large number of fundamental kernel interfaces block and currently can't return an error. `lock_page()` and `mutex_lock()` are only two of the most important. These interfaces have to be taught to return an error instead of blocking. Not only does this push changes out to core subsystems, it requires rewriting code paths to handle errors from these functions which might not have handled errors before.

- KAIO's special return codes must be returned from their source, which has promised to call `aio_complete()`, all the way back to `io_setup()`, which will call `aio_complete()` if it **does not** see the special error codes. Code that innocently used to overwrite an existing error code with its own, say returning `-EIO` when O_SYNC metadata writing fails, can lead to duplicate calls to `aio_complete()` . This invariant must be enforced through any mid-level helpers that might have no idea that they're being called in the path between `io_submit()` and the source of KAIO's special error codes.

- `iocb`s duplicate system call arguments. For any operation to be supported by KAIO it must have its arguments expressed in the `iocb` structure. This duplicates all the problems that the system call interface already solves. Should we use native types and compatibility translation between user space and kernel for different word sizes? Should we use fixed-width types and create subtle inconsistencies with the synchronous interfaces? If we could reuse the existing convention of passing arguments to the kernel, the system call ABI, we'd avoid this mess.

Far better would be a way to provide an asynchronous system call interface without having to modify, and risk breaking, existing system call handlers.

## 2 Fibrils

### 2.1 Blame Linus

In November of 2005, Linus Torvalds expressed frustration with the current KAIO implementation. He suggested an alternative he called *micro-threads*. It built on two fundamental, and seemingly obvious, observations:

1. The C stack already expresses the partial progress of an operation much more naturally than explicit progress storage in `kiocb` ever will.

2. `schedule()`, the core of the kernel task scheduler, already knows when an operation blocks for any reason. Handling blocked operations in the scheduler removes the need to handle blocking at every potential blocking call site.

The proposal was to use the call stack as the representation of a partially completed operation. As an operation is submitted its handler would be executed as normal, exactly as if it was executed synchronously. If it blocked, its stack would be saved away and the stack of the next runnable operation would be put into place.

The obvious way to use a scheduler to switch per-operation stacks is to use a full kernel thread for each operation. Originally it was feared that a full kernel thread per operation would be too expensive to manage and switch between because the existing scheduler interface would have required initiating each operation in its own thread from the start. This is wasted effort if it turns out that the operations do not need their own context because they do not block. Scheduling stacks only when an operation blocks defers the scheduler's work until it is explicitly needed.

After experience with KAIO's limitations, scheduling stacks offers tantalizing benefits:

- The submitting process never blocks
- Very little cost is required to issue non-blocking operations through the AIO interface

Most importantly, it requires no changes to system call handlers—they are almost all instantly supported.

### 2.2 Fibrils prototype

There were two primary obstacles to implementing this proposal.

First, the kernel associates a structure with a given task, called the `task_struct`. By convention, it's considered private to code that is executing in the context of that task. The notion of scheduling stacks changes this fundamental convention in the kernel. Scheduling stacks, even if they're not concurrently executing on multiple CPUs, implies that code which accesses `task_struct` must now be re-entrant. Involuntary preemption greatly increases the problem. Every member of `task_struct` would need to be audited to ensure that concurrent access would be safe.

Second, the kernel interfaces for putting a code path to sleep and waking it back up are also implemented in terms of these `task_structs`. If we now have multiple code paths being scheduled as stacks in the context of one task then we have to rework these interfaces to wake up the stacks instead of the `task_struct` that they all belong to. These sleeping interfaces are some of the most used in the kernel. Even if the changes are reasonably low-risk this is an incredible amount of code churn.

It took about a year to get around to seriously considering making these large changes. The result was a prototype that introduced a saved stack which could be scheduled under a task, called a *fibril* [1].

## 3 Syslets

The fibrils prototype succeeded in sparking debate of generic AIO system calls. In his response to fibrils [4], Ingo Molnar reasonably expressed dissatisfaction with the fibrils construct. First, the notion of a secondary simpler fibrils scheduler will not last over time as people ask it to take on more functionality. Eventually it will end up as complicated as the task scheduler it was trying to avoid. There were already signs of this in the lack of support for POSIX AIO's prioritized operations. Second, effort would be better spent adapting the main task scheduler to support asynchronous system calls instead of creating a secondary construct to avoid the perceived cost of the task scheduler.

Two weeks later, he announced [2] an interface and scheduler changes to support asynchronous system calls which he called *syslets*.[1]

The syslet implementation first makes the assertion that the modern task scheduler is efficient enough to be used for swapping blocked operations in and out. It uses full kernel tasks to express each blocked operation.

The syslet infrastructure modifies the scheduler so that each operation does not require execution in a dedicated task at the time of submission. If a task submits a system call with syslets and the operation blocks, then the

[1] The reader may be forgiven for giggling at the similarity to Chicklets, a brand of chewing gum.

scheduler performs an implicit `clone()`. The submission call then returns as the **child** of the submitting task. This is carefully managed so that the user space registers associated with the submitting task are migrated to the new child task that will be returning to user space. This requires a very small amount of support code in each architecture that wishes to support syslets.

Returning from a blocked syslet operation in a cloned child is critical to the simplicity of the syslets approach. Fibrils tried to return to user space in the same context that submitted the operation. This lead to extensive modifications to allow a context to be referenced by more than one operation at a time. The syslet infrastructure avoids these modifications by declaring that a blocked syslet operation will return to user space in a new task.

This is a viable approach because nearly all significant user space thread state is either shared between tasks or is inherited by a new child task from its parent. It will be very rare that user space will suffer ill effects of returning in a new child task. One consequence is that `gettid()` will return a new value after a syslet operation blocks. This could require some applications to more carefully manage per-thread state.

## 4 Implementing KAIO with syslets

So far we've considered asynchronous system calls, both fibrils and syslets, which are accessible through a set of new system calls. This gives user space a powerful new tool, but it does nothing to address the kernel maintenance problem of the KAIO interface. The KAIO interface can be provided by the kernel but implemented in terms of syslets instead of `kiocb`s.

As the `iocb` structures are copied from user space their arguments would be used to issue syslet operations. As the system call handler returns in the syslet thread it would take the result of the operation and insert it into the KAIO event completion ring.

Since the syslet interface performs an implicit clone we cannot call the syslet submission paths directly from the submitting user context. Current KAIO users are not prepared to have their thread context change under them. This requires worker threads which are very carefully managed so as not to unacceptably degrade performance.

Cancellation would need to be supported. Signals could be sent to the tasks which are executing syslets which could cause the handler to return. The same accounting which associated a user's `iocb` with a syslet could be annotated to indicate that the operation should complete as canceled instead of returning the result of the system call.

### 4.1 Benefits

Implementing KAIO with syslets offers to simplify existing paths which support KAIO. Those paths will also provide greater KAIO support by blocking less frequently.

System call handlers would no longer need to know about `kiocb`s. They could be removed from the file IO paths entirely. Synchronous file IO would no longer need to work with these `kiocbs` when they are not providing KAIO functionality. The specific error codes that needed to be carefully maintained could be discarded.

KAIO submission would not block as often as it does now. As explored in our trivial file hashing example, blocking in submission can lead to resource underutilization. Others have complained that it can make it difficult for an application to measure the latency of operations which are submitted concurrently. This has been observed in the field as O_DIRECT writes issued with KAIO block waiting for an available IO request structure.

### 4.2 Risks

Implementing KAIO with syslets runs the risk of adding significant memory and CPU cost to each operation. It must be very carefully managed to keep these costs under control.

Memory consumption will go up as each blocked operation is tracked with a kernel thread instead of a `kiocb`. This may be alleviated by limiting the number of KAIO operations which are issued as syslets at any given time. This measure would only be possible if KAIO continues to only support operations which are guaranteed to make forward progress.

Completion will require a path through the scheduler. O_DIRECT file IO demonstrates this problem. Previously it could call `aio_complete()` from block IO

completion handlers which would immediately queue the operation for collection by user space. With syslets the block IO completion handlers would wake the blocked syslet executing the IO. The syslet would wake up and run to completion and return at which point the operation would be queued for collection by user space.

### 4.3 Remaining Work

An initial rewrite of the KAIO subsystem has been done and put through light testing. At the time of this writing conclusive results are not yet available. There is much work yet to be done. Results may be found in the future on the web at `http://oss.oracle.com/~zab/kaio-syslets/`.

## 5 Conclusion

KAIO has long frustrated the kernel community with its limited functionality and high maintenance cost. Syslets offer a powerful interface for user space to move towards in the future. With luck, syslets may also ease the burden of supporting existing KAIO functionality while addressing significant limitations of the current implementation.

## 6 Acknowledgments

Thanks to Valerie Henson and Mark Fasheh for their valuable feedback on this paper.

## References

[1] Zach Brown. Generic aio by scheduling stacks. `http://lkml.org/lkml/2007/1/30/330`.

[2] Ingo Molnar. Announce: Syslets, generic asynchronous system call support. `http://lkml.org/lkml/2007/2/13/142`.

[3] Ingo Molnar. downloadable syslets patches. `http://people.redhat.com/mingo/syslet-patches/`.

[4] Ingo Molnar. in response to generic aio by scheduling stacks. `http://lkml.org/lkml/2007/1/31/34`.