Advanced Topics in Module Design: Threadsafety and Portability

Aaron Bannert
aaron@apache.org / aaron@codemass.com

http://www.codemass.com/~aaron/presentations/apachecon2005/ac2005advancedmodules.ppt

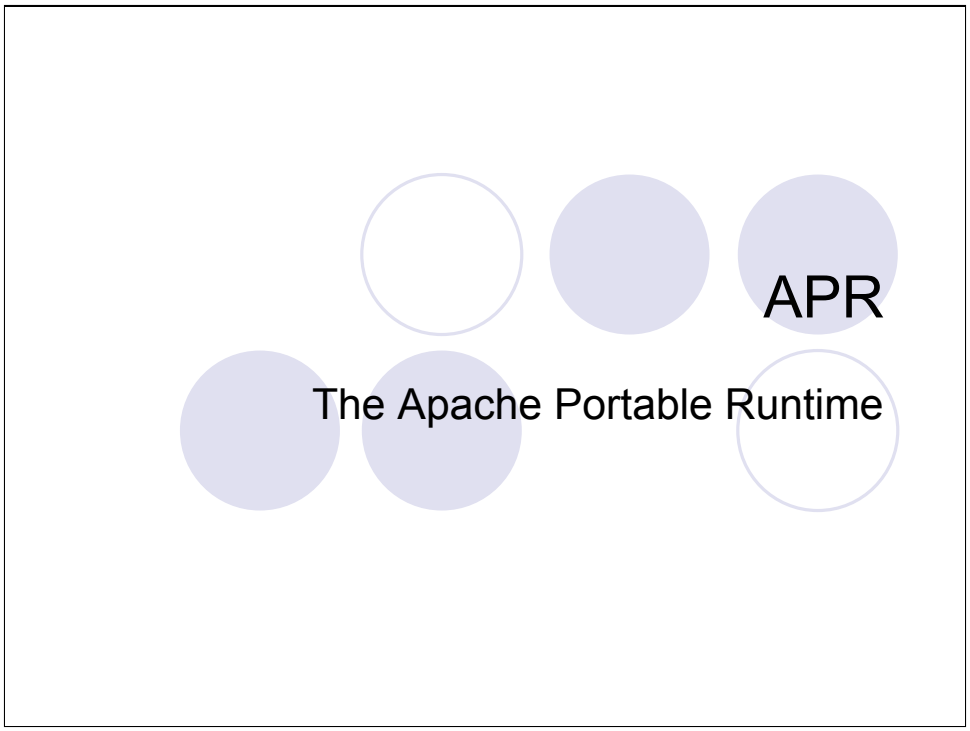
Thread-safe

From The Free On-line Dictionary of Computing (09 FEB 02) [foldoc]:

thread-safe

A description of code which is either *re-entrant* or protected from multiple simultaneous execution by some form of *mutual exclusion*.

(1997-01-30)


APR
The Apache Portable Runtime

The APR Libraries

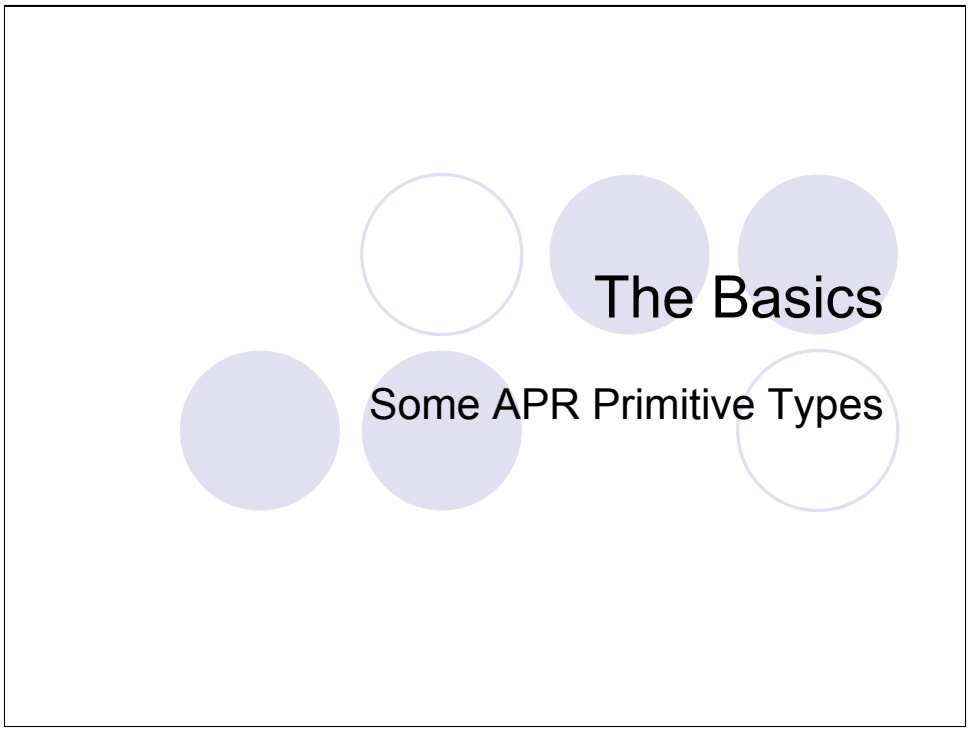
- APR
  - System-level “glue”
- APR-UTIL
  - Portable routines built upon APR
- APR-ICONV
  - Portable international character support

Glue Code vs. Portability Layer

- "Glue Code"
  - Common functional interface
    - Multiple Implementations
    - eg. db2, db3, db4, gdbm, …
      Sockets, File I/O, …
- "Portability Layer"
  - Routines that embody portability
    - eg. Bucket Brigades, URI routines, …

What Uses APR?

- Apache HTTPD
- Apache Modules
- Subversion
- Flood
- JXTA-C
- Various ASF Internal Projects
- ...


The Basics
Some APR Primitive Types

A Who's Who of Mutexes

- **apr_thread_mutex_t**
- **apr_proc_mutex_t**
- **apr_global_mutex_t**

- apr_xxxx_mutex_lock()
  - Grab the lock, or block until available
- apr_xxxx_mutex_unlock()
  - Release the current lock

The apr_thread_mutex_t type is used to synchronize execution between threads in the same process. The apr_proc_mutex_t type is used to synchronize different processes, but not necessarily multiple threads within those same processes. Finally, the apr_global_mutex_t type ensures that multiple threads in multiple processes are all properly synchronized.

One typically uses apr_thread_mutex_t to protect access to a resource in a single process, while apr_global_mutex_t can be used for resources that are shared between multiple multithreaded processes (which is typical for an MPM-agnostic module). One case where an apr_proc_mutex_t may be optimal would be when each process can guarantee that only one thread at a time will access some central resource. An example of this can be seen with the socket accept() code inside Apache 2.0.

Other than creating and destroying mutexes, the only operations that can be performed on them are the _lock() and _unlock() calls.

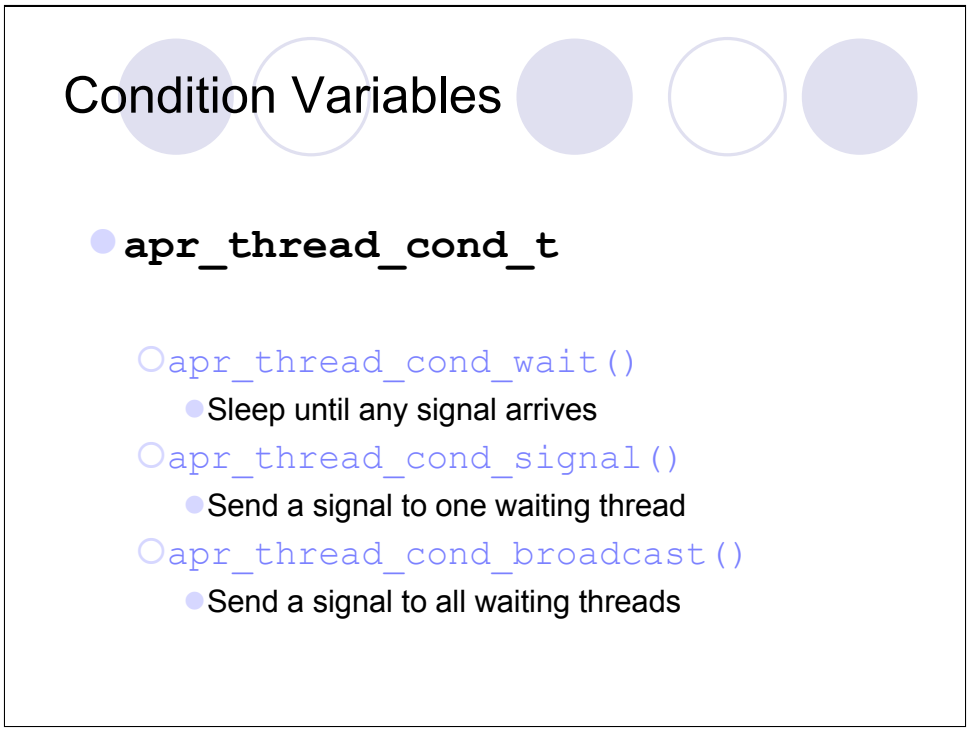
Normal vs. Nested Mutexes

- Normal Mutexes (aka Non-nested)
  - Deadlocks when same thread locks twice
- Nested Mutexes
  - Allows multiple locks with same thread
    (still have to unroll though)

Reader/Writer Locks

- **`apr_thread_rwlock_t`**
  - `apr_thread_rwlock_rdlock()`
    - Grab the shared read lock, blocks for any writers
  - `apr_thread_rwlock_wrlock()`
    - Grab the exclusive write lock, blocking new readers
  - `apr_thread_rwlock_unlock()`
    - Release the current lock

Reader/Writer locks are created and destroyed in much the same way mutexes are, but behave quite differently. Instead of having a simple locked/unlocked state, a reader/writer lock allows the caller to signify whether they desire a shared read state, or an exclusive write state of the lock. This allows finer-grain control over the synchronization between threads, which can lead to better parallelism. The way this works is you can have multiple readers at the same time, but as soon as a writer requests the lock, no new readers are allowed to acquire until that writer has released, and that writer does not acquire the lock until all the current readers have released.



Condition variables are always used in conjunction with a mutex. This mutex protects the signaling mechanism from race condition problems.

One common example that illustrates a use of condition variables is the producer/consumer problem. Imagine a single producer that can produce 5 widgets per second. There are also 10 consumers, each of which can consume 1 widget per second. Given only mutexes, there is no guarantee that the work will be shared between all of the consumers. With condition variables, each consumer waits for the signal to begin work. As the producer finishes readying each widget, he signals one consumer, who then wakes up and performs a consumptive task. Upon completing the consumption, the worker then returns to the queue.

Threads

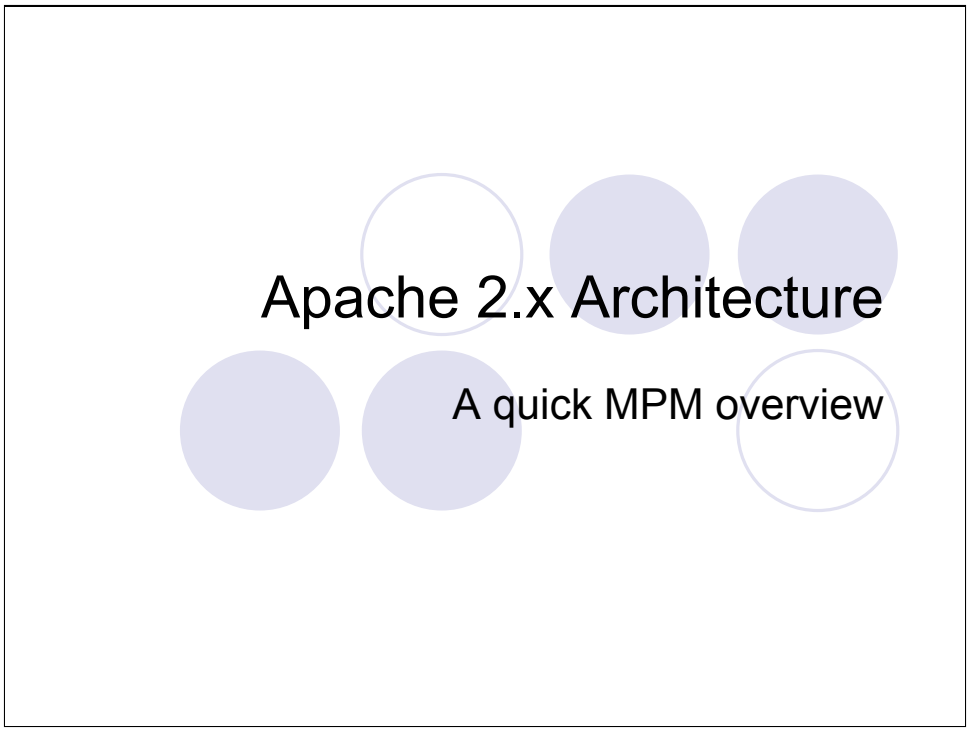
- **`apr_thread_t`**
  - `apr_thread_create()`
    - Create a new thread (with specialized attributes)
  - `apr_thread_exit()`
    - Exit from the current thread (with a return value)
  - `apr_thread_join()`
    - Wait until another thread exits.

Thread creation is fairly straightforward. Each thread is created from a *pool*, from which a thread-specific sub-pool is created and passed to the start function. A start function is specified at creation time, and as soon as the thread is created by the system, it begins executing that start function. APR provides a data passing system from the creating thread to the created thread, via the start function's prototype.

One-time Calls

- **`apr_thread_once_t`**
  - `apr_thread_once_init()`
    - Initialize an `apr_thread_once_t` variable
  - `apr_thread_once()`
    - Execute the given function once

One-time calls allow initialization of process-wide resources in a way that avoids race conditions. The call to `apr_thread_once()` guarantees that a specified call will only be called once, no matter how many times `apr_thread_once()` is called.


Apache 2.x Architecture
A quick MPM overview

What's new in Apache 2.x?

- Filters
- MPMs
  - Multithreaded Server
  - Native OS Optimizations
- SSL Encryption
- Improved Proxy and Cache
- lots more…

The main thing that we are concerned with in terms of writing threadsafe and multithread-aware modules is the MPMs.

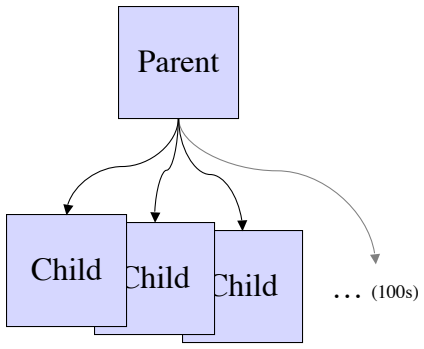
What is an MPM?

- "Multi-processing Module"
  - Different HTTP server process models
  - Each give us
    - Platform-specific features
    - Admin may chose suitable:
      - Reliability
      - Performance
      - Features

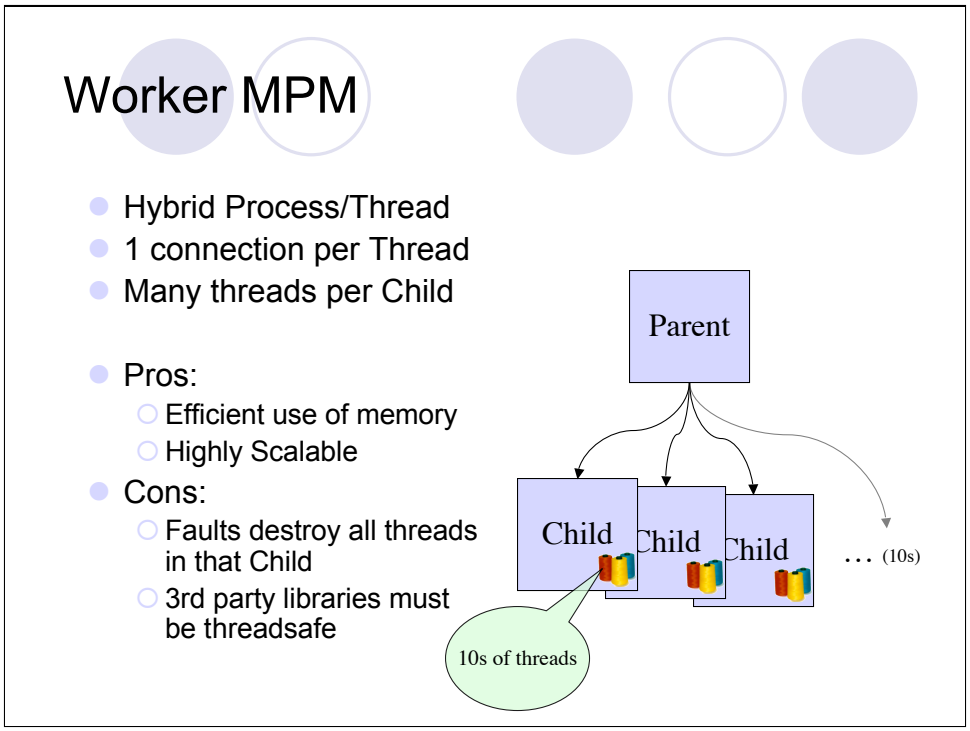
An MPM allows platform-specific optimizations in how Apache deals with incoming requests and parallelism.

Prefork MPM

- Classic Apache 1.3 model
- 1 connection per Child
- Pros:
  - Isolates faults
  - Performs well
- Cons:
  - Scales poorly (high memory reqts.)



Available on Unix. This MPM matches the model used since NCSA and through Apache up to Apache 1.3. In general it tends to perform well on uniprocessor machines, and is highly robust. The prefork MPM is not well suited for high-traffic, highly concurrent systems. Although the prefork MPM processes many hundreds of simultaneous connections without problem, systems are usually limited by the amount of memory consumed.

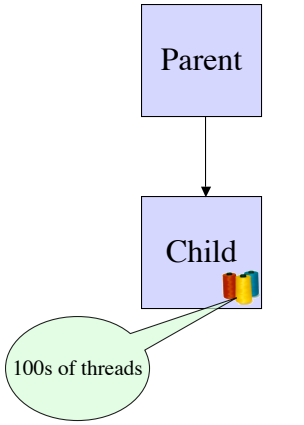


The worker MPM is the new hybrid multitprocess/multithreaded model specifically targeted by Apache 2.0. This MPM trades a small amount of fault isolation for massive scalability (especially in terms of memory consumption). It performs on par or slightly under par with a lightly loaded prefork MPM server, but is able to handle many more concurrent connections without significant memory consumption.

WinNT MPM

- Single Parent/Single Child
- 1 connection per Thread
- Many threads per Child

- Pros:
  - Efficient use of memory
  - Highly Scalable
- Cons:
  - Faults destroy all threads



The WinNT MPM is specifically customized for the Windows platform. It has one parent and one child, with the single child having many hundreds of concurrent threads. Each thread handles a separate request connection.

The MPM Breakdown

MPM	Multi-process	Multithreaded
Prefork	Yes	No
Worker	Yes	Yes
WinNT	No*	Yes

* The WinNT MPM has a single parent and a single child.

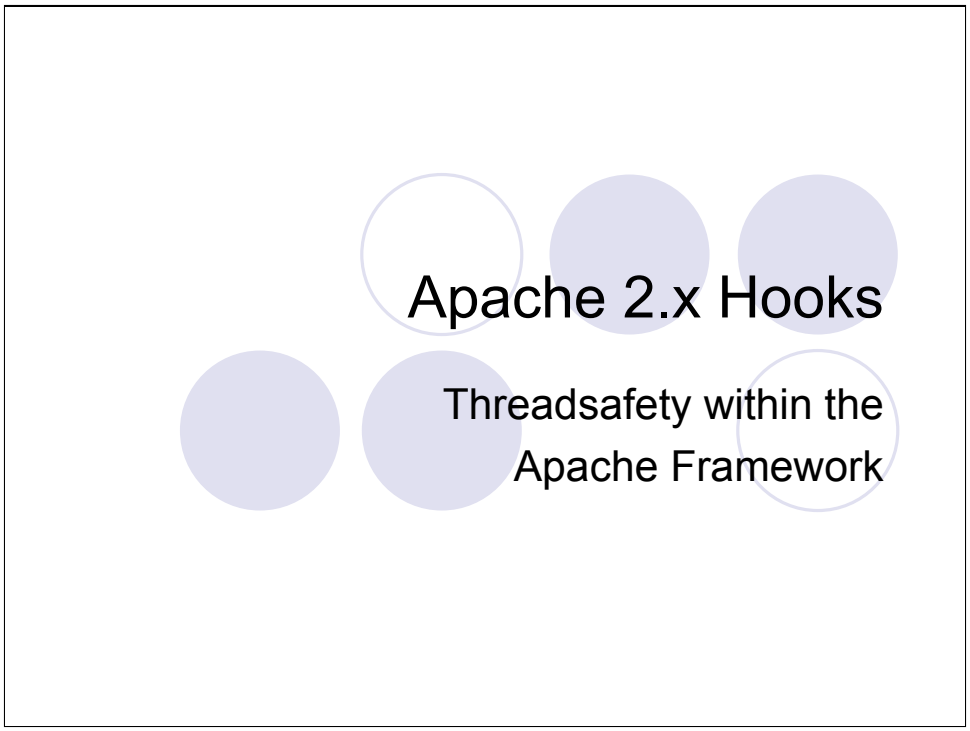
There is some talk of adding multi-child support for the WinNT MPM. This would help isolate faults in 3rd party modules, so that a single failure would not cause new requests to have to wait for a new child to be started

(which can mean downtime on the order of seconds, rather than not at all).

Also note that this list is by no means exhaustive. There are a few experimental MPMs that attempt improvements over the worker MPM, as well as a perchild MPM which is targeted at mass multi-hosting facilities.

Other MPMs

- BeOS
- Netware
- Threadpool
  - Similar to Worker, experimental
- Leader-Follower
  - Similar to Worker, also experimental


Apache 2.x Hooks
Threadsafety within the
Apache Framework

Useful APR Primitives

- mutexes
- reader/writer locks
- condition variables
- shared memory
- ...

The way we create, initialize, use, and destroy each of these types is pretty much the same across the board. This means that you can write the code once and expect it to work on both Unix and Windows, BeOS and Netware. APR tries to hide much of the trouble of dealing with inconsistencies between the platforms.

Global Mutex Creation

1. Create it in the Parent:
   - Usually in `post_config` hook
2. Attach to it in the Child:
   - This is the `child_init` hook

By creating in the parent process, we allow systems like Unix to inherit these types in a child process.

The post_config hook is ideal In most cases for module authors, since it is the first hook in the parent where all the configuration information is available. For example, a module may define a directive where a user can specify a filename where a global mutex is to be stored.

Example: Create a Global Mutex

```
static int shm_counter_post_config(apr_pool_t *pconf,
                                   apr_pool_t *plog,
                                   apr_pool_t *ptemp,
                                   server_rec *s) {
 int rv;
 shm_counter_scfg_t *scfg;

 /* Get the module configuration */
 scfg = ap_get_module_config(s->module_config,
                             &shm_counter_module);

 /* Create a global mutex from the config directive */
 rv = apr_global_mutex_create(&scfg->mutex,
                              scfg->shmcounterlockfile,
                              APR_LOCK_DEFAULT, pconf);
```

This example shows how we can pull the module configuration from the `server_rec` and use those directives as parameters to the global mutex creation routine.

Passing `APR_LOCK_DEFAULT` here signifies that we wish to have a non-recursive mutex. A recursive mutex is a mutex that can be repeatedly acquired by the same thread without immediately deadlocking (as long as the same number of releases occur). I do not believe that there are many cases that require a recursive lock, and tend to avoid them. In addition, they tend to perform poorly on some Unix systems, and also tend to mask programmer bugs more often than non-recursive mutexes.

Example: Attach Global Mutex

```
static void shm_counter_child_init(apr_pool_t *p,
                                   server_rec *s)
{
    apr_status_t rv;
    shm_counter_scfg_t *scfg
        = ap_get_module_config(s->module_config,
                               &shm_counter_module);

    /* Now that we are in a child process, we have to
     * reconnect to the global mutex. */
    rv = apr_global_mutex_child_init(&scfg->mutex,
                         scfg->shmcounterlockfile, p);
```

In the `child_init` hook we must attach to the mutex. Under Unix, this happens automatically, since mutexes are inherited by child processes via the `fork()` mechanism.

The reason we need to pass the filename of the mutex back into the `apr_global_mutex_child_init()` call is so that under Windows we are able to connect to the same global mutex from all child processes.

At this point, any child process may lock or unload the global mutex in order to synchronize access to some cross-process shared resource.

Common Pitfall

- The double DSO-load problem
  - Apache loads each module twice:
    - First time to see if it fails at startup
    - Second time to actually load it
    - Also reloaded after each *restart*.

Apache wants to be sure that all modules can be successfully loaded into the server before the server decides it is ok to start up. To do this it goes through a full load-unload-load cycle at startup. The problem with this is that it causes unexpected problems when we try to initialize our mutex twice.

Avoiding the Double DSO-load

- Solution:
  - Don't create mutexes during the first load

1. First time in post_config we set a *userdata* flag
2. Next time through we look for that *userdata* flag
   - if it is set, we create the mutex

One way to prevent this from happening is to set a flag on the first call to the post_config hook. On each subsequent call to post_config we check if that flag has been set, and if so we proceed with our initialization.

What is *Userdata?*

- Just a hash table
- Associated with each pool
- Same lifetime as its pool
- Key/Value entries

Example: Double DSO-load

```
static int shm_counter_post_config(apr_pool_t *pconf, apr_pool_t *plog,
                                   apr_pool_t *ptemp, server_rec *s)
{
    apr_status_t rv;
    void *data = NULL;
    const char *userdata_key = "shm_counter_post_config";

    apr_pool_userdata_get(&data, userdata_key, s->process->pool);
    if (data == NULL) {
        /* WARNING: This must *not* be apr_pool_userdata_setn(). */
        apr_pool_userdata_set((const void *)1, userdata_key,
                              apr_pool_cleanup_null, s->process->pool);
        return OK; /* This would be the first time through */
    }

    /* Proceed with normal mutex and shared memory creation . . . */
```

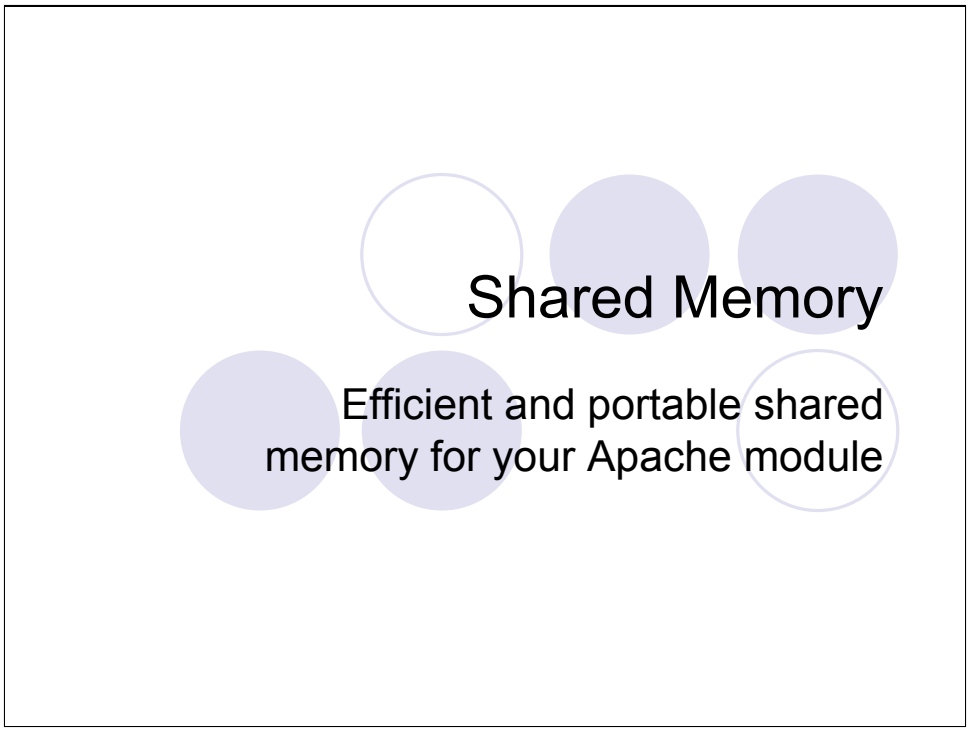
Bonus Question: Why is it so important that we don't use `apr_pool_userdata_setn()` to set the userdata flag?

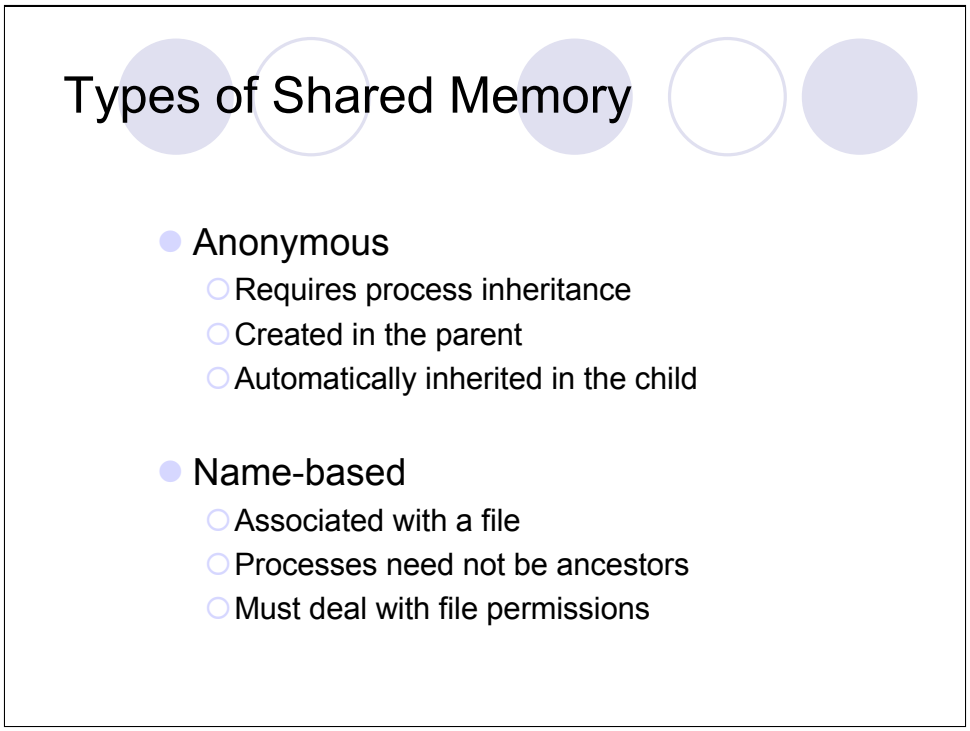
The reason for this is because the static symbol section of the DSO may not be at the same address offset when it is reloaded. Since setn() does not make a copy and only compares addresses, the get() will be unable to find the original userdata.

Summary

1. Create in the Parent (`post_config`)
2. Attach in the Child (`child_init`)

- This works for these types:
  - mutexes
  - condition variables
  - reader/writer locks
  - shared memory
  - etc…

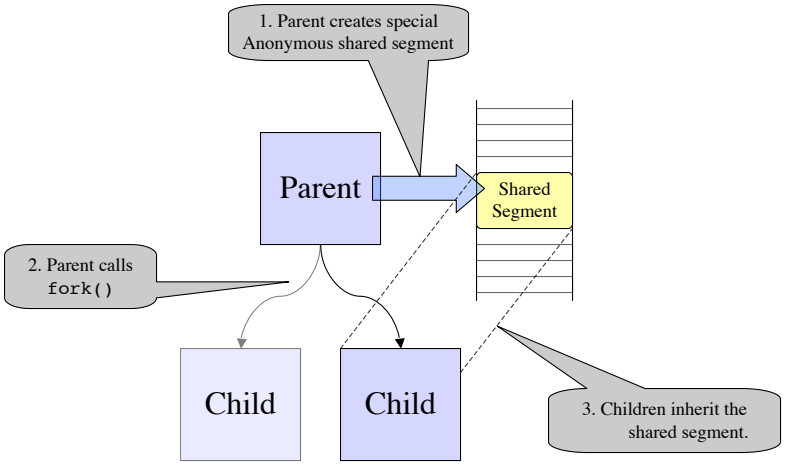

Shared Memory
Efficient and portable shared memory for your Apache module



The main difference between the two has to do with process inheritance. Anonymous shared memory is typcially easier to deal with, but is not as well supported as name-based shared memory. Windows does not support anonymous shared memory (with some exceptions).

Often a particular implementation of APR's anonymous shared memory will have size limitations, while that same platform may not have similar restrictions on a name-based implementation.

Anonymous Shared Memory



Example: Anonymous Shmem

```
static int shm_counter_post_config(apr_pool_t *pconf,
                                   apr_pool_t *plog,
                                   apr_pool_t *ptemp,
                                   server_rec *s)
{
 int rv;
 ...

 /* Create an anonymous shared memory segment by passing
  * a NULL as the shared memory filename */
 rv = apr_shm_create(&scfg->counters_shm,
                     sizeof(*scfg->counters),
                     NULL, pconf);
```

As with the mutex example, this example shows how we can pull the module configuration from the `server_rec` and use that information to create our segment.

Note that the double DSO-load preventing code is omitted for brevity, as is the mutex creation code.

By passing the requested size to the creation function we are guaranteed by APR that it will create at least that size segment or the create routine will fail. There is a small amount of overhead in some cases, but it is handled in a transparent manner.

Accessing the Segment

Segment is mapped as soon as it is created
- It has a start address
- You can query that start address

Reminder: The segment may not be mapped to the same address in all processes.

```
scfg->counters = apr_shm_baseaddr_get(scfg->counters_shm);
```

Accessing the segment is as simple as asking APR for the address of the start of the segment. Note that this address is only valid for the current process, not across processes. What this means is that simple pointers can not be stored in the segment, but must be relative to some global addressing scheme. A pointer in one process will not necessarily work in another process where the segment may have been mapped at a totally different location.

Windows Portability

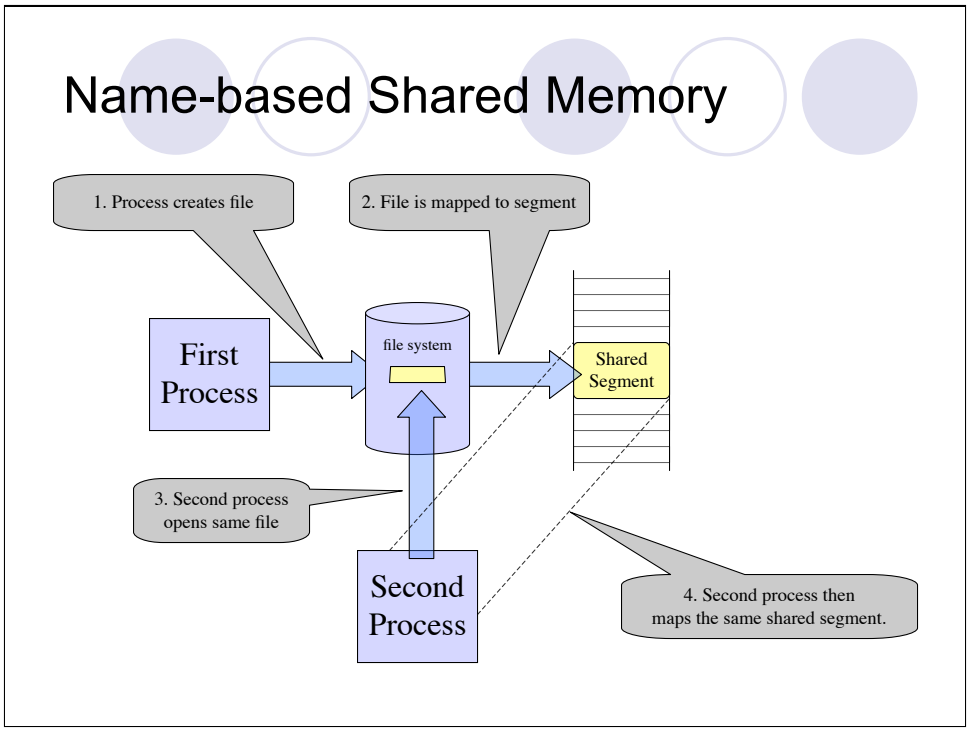
- Windows can't inherit shared memory
  - it has no `fork()` call!
- Solution:
  - Just like we did with mutexes:
    - The "child" process *attaches*

      (hint: to be portable to Windows, we can only use name-based shared memory.)

This isn't entirely true, there are ways around this, and there are types of "anonymous shared memory" available on Windows, but in the end there still needs to be a way for the Apache "parent" and "child" to rendezvous so that they can both attach to the same memory segment.

The best way to avoid all of this is to just use name-based shared memory. Unfortunately this means that you have to start worrying about things like where to put the file, and what permissions can it have.

The Apache 2.0 scoreboard is an excellent example of a hybrid solution. If the underlying system support anonymous shared memory, the scoreboard uses it, otherwise it falls back to name-based shared memory, using the name of a file from a configuration directive, or defaulting to a standard filename.



Bonus Question: Why do we have to worry about race conditions?

The second process cannot access the segment until after the first process has completely created it.

Sharing with external apps

- Must use name-based shm
- Associate it with a file
- The other programs can *attach* to that file

- Beware of race conditions
  - Order of file creation and *attaching*.
- Beware of weak file permissions

  (note previous security problem in Apache scoreboard)

Although APR's name-based shared memory requires that the segment be associated with a file, whether or not it uses that file for the actual contents of the segment is implementation (and platform) specific. By associating the segment with a file, APR allows fully autonomous external programs to all share the same segment. By this property of name-based shared memory, it is possible to write an external application that can monitor the status of your module via its file-based shared memory segment.

Recently there was a security problem in Apache (both 1.3 and 2.0) where it was possible for someone who had access to the user under which apache was running to send certain signals to any process on the system as though the signals were coming from root. The reason for this was that the apache user was able to open the shared memory segment and alter the entries holding the PIDs of the children, causing the parent process (running as root) to unwittingly send signals to whatever PID it read from the segment. Ouch!

Example: Name-based Shmem

```
static int shm_counter_post_config(apr_pool_t *pconf,
                                   apr_pool_t *plog,
                                   apr_pool_t *ptemp,
                                   server_rec *s) {
 int rv;
 shm_counter_scfg_t *scfg;
 ...

 /* Get the module configuration */
 scfg = ap_get_module_config(s->module_config,
                             &shm_counter_module);

 /* Create a name-based shared memory segment using the filename
  * out of our config directive */
 rv = apr_shm_create(&scfg->counters_shm, sizeof(*scfg->counters),
                     scfg->shmcounterfile, pconf);
```

Note again that the double DSO-load preventing code is omitted for brevity, as is the mutex creation code.

The only difference here from how we created the anonymous shared memory segment is that now we are passing an actual filename instead of just NULL.

Example: Name-based Shmem (cont)

```
static void shm_counter_child_init(apr_pool_t *p,
                                   server_rec *s)
{
 apr_status_t rv;
 shm_counter_scfg_t *scfg
     = ap_get_module_config(s->module_config,
                            &shm_counter_module);

 rv = apr_shm_attach(&scfg->counters_shm,
                     scfg->shmcounterfile, p);

 scfg->counters = apr_shm_baseaddr_get(scfg->counters_shm);
```

Now that we have one process which has created and (automatically) attached to the segment, we can have other programs use that same filename to attach to the same segment.

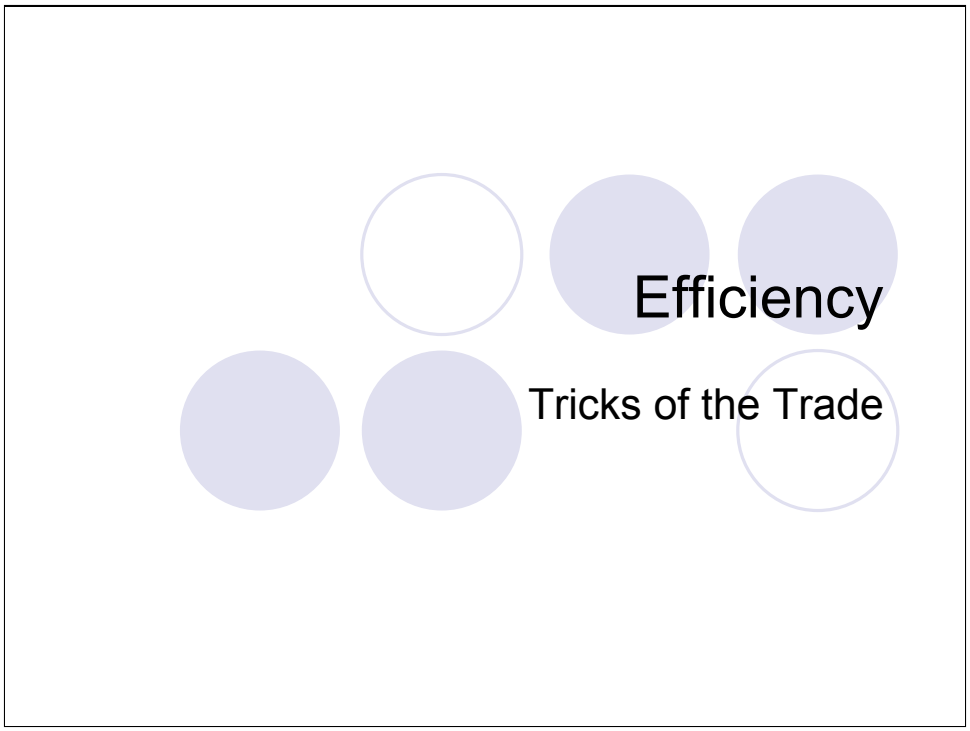
Once we have attached to the segment, we can grab that base address to the segment and start plugging away.

RMM (Relocatable Memory Manager)

- Provides `malloc()` and `free()`
- Works with any block of memory
- Estimates overhead
- Thread-safe
- Usable on shared memory segments

The RMM was created specifically for memory management within any contiguous block of memory, and is particularly suited for memory management within shared memory space. It is threadsafe in the sense that it can be accessed in parallel by multiple threads and processes without memory corruption.

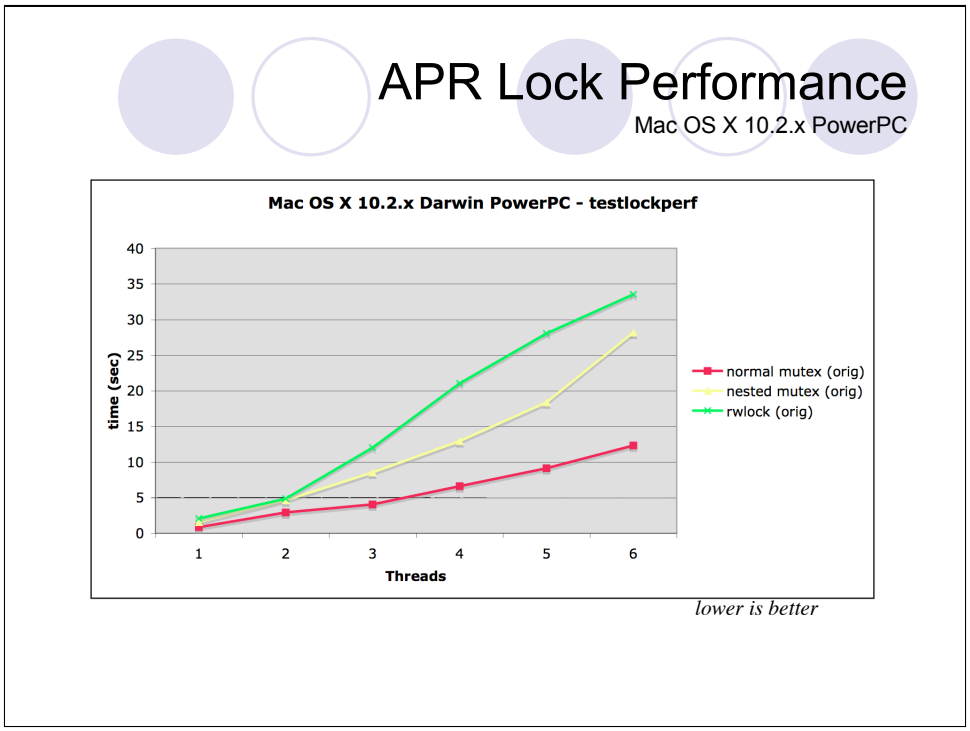
There is a significant amount of overhead associated with maintain a free list within the shared-memory segment, but that cost can be estimated by the RMM code itself.


Efficiency
Tricks of the Trade

Questions to ask yourself:

- Uniprocessor or Multiprocessor?
- What Operating System(s)?
- How can we minimize or eliminate our critical code sections?
- Exclusive access or read/write access?

Often times a clever design can avoid requiring a mutex altogether. One good example of this is the scoreboard in Apache. Although the scoreboard is read and written by many processes at the same time, the program was written in such a way to prevent simultaneous use of the same section of the scoreboard.

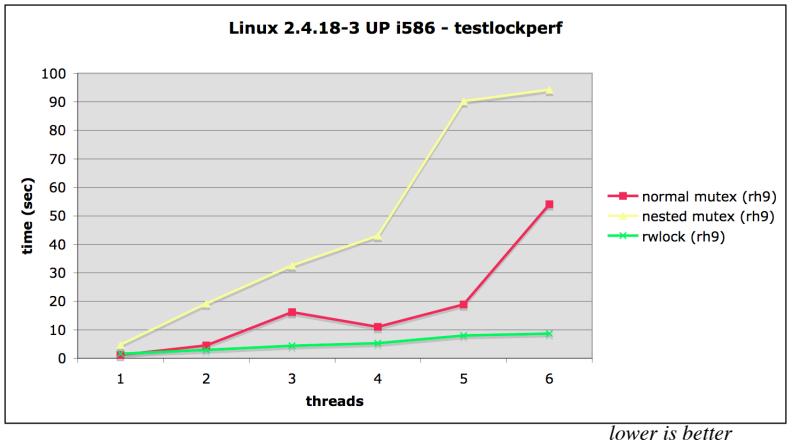


In this simple test (which is just a graph of the output from a modified version of the testlockperf program in APR's test directory), we are comparing the nested and non-nested mutexes and read/write locks. At each stage, the test creates a number of threads, and each thread contends for a mutex (the rwlock just grabs the exclusive write lock) increments a counter, and then releases the lock. Once the counter reaches 1 million, the threads are killed and the time is measured.

As is evident in the graph, a rwlock is far more expensive than either mutex type, at least while there are fewer threads contending for the mutex. Also, the nested mutex appears to be on average twice as expensive as the non-nested mutex.

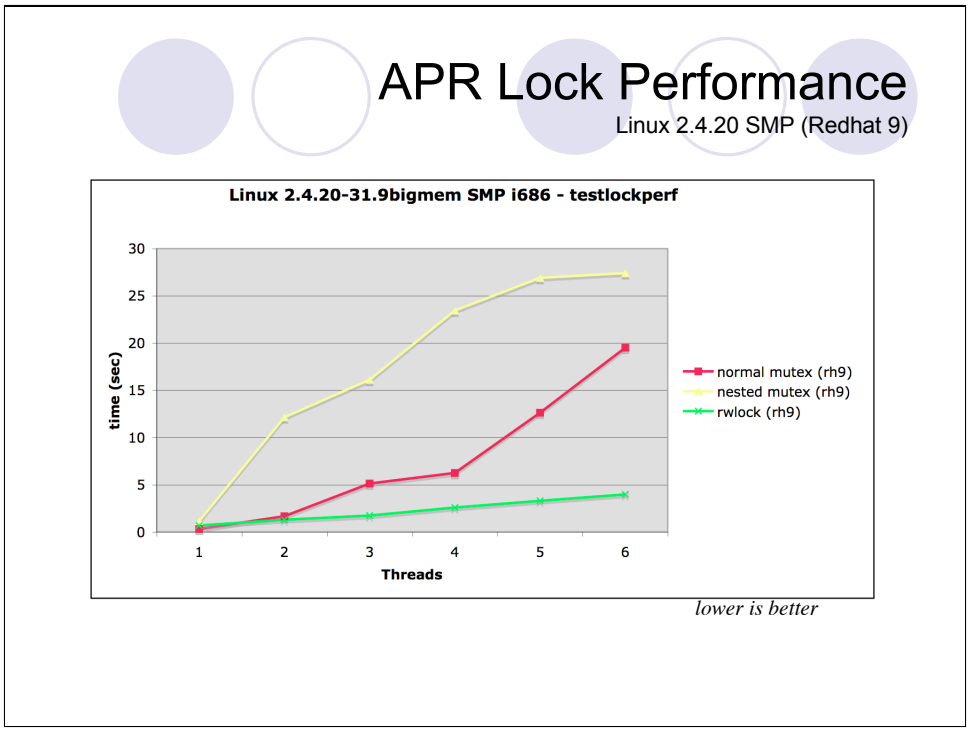
APR Lock Performance

Linux 2.4.18 (Redhat 7.3)

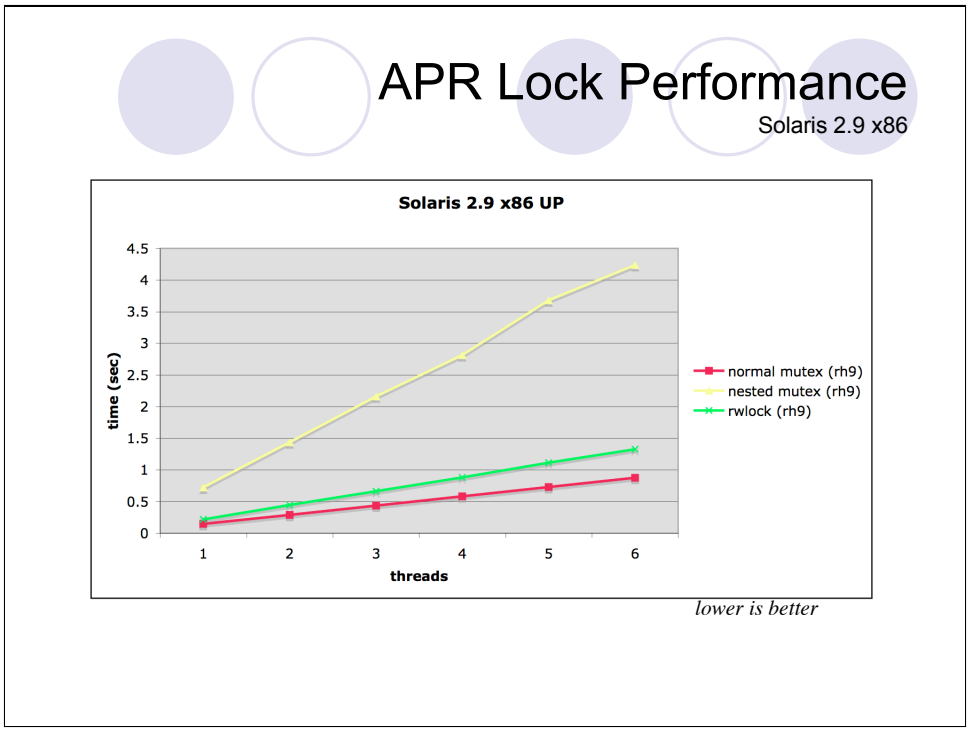


lower is better

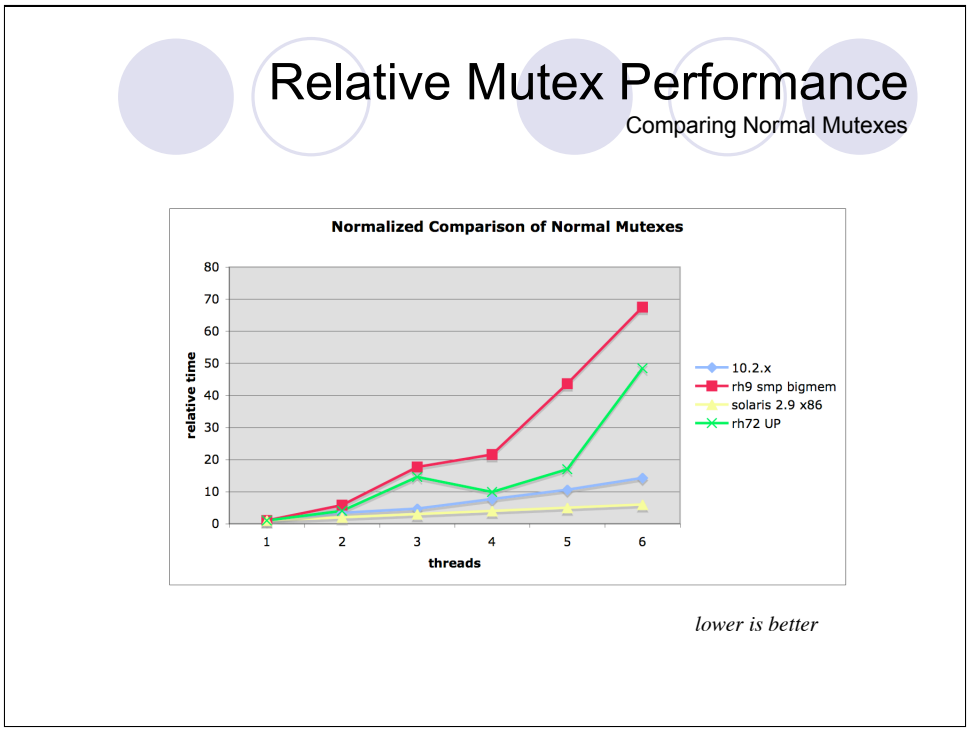
It becomes evident from this graph that the overhead of the nested mutex as currently implemented in APR is mostly CPU-bound. The main contrast between this graph and the previous graph is that the rwlock performance is dramatically better. Also, the performance of the mutexes does not appear to scale linearly. Although both machines were idle while performing the test, the performance of the linux implementation varies widely as more threads are added to the mix.



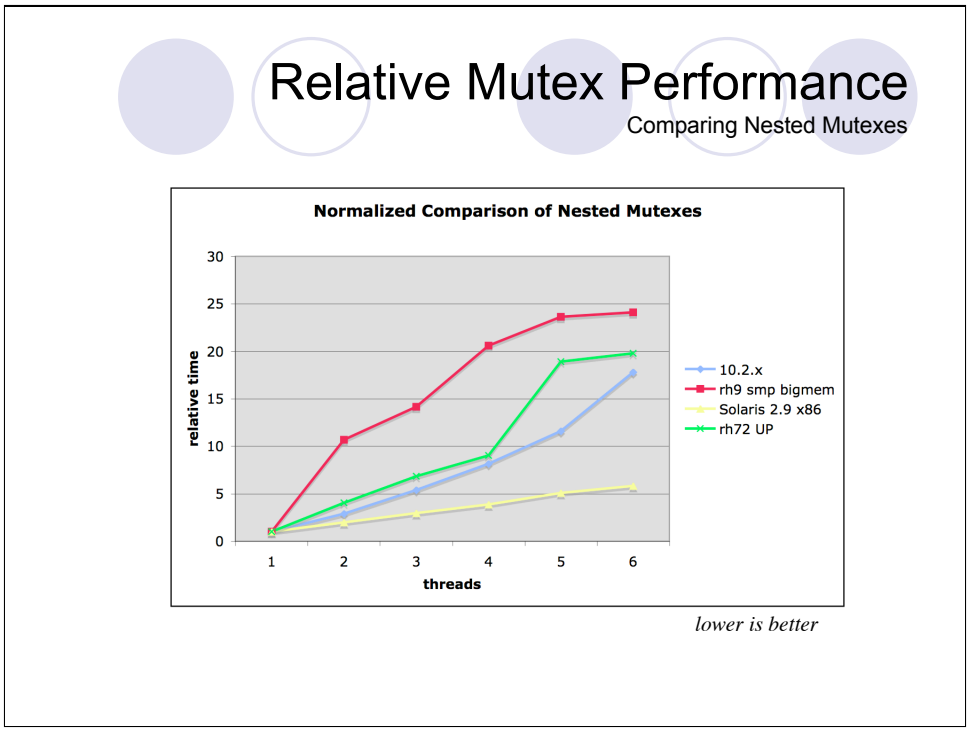
This graph demonstrates two major platform differences compared to the previous Redhat 7.3-based test: 1) This test demonstrates the new NPTL code in Redhat 9, and 2) This machine demonstrates the performance of Linux's locking primitives in a multi-processor environment. It appears that the scaling characteristics are not greatly affected by the addition of a second processor, which is to be expected since this test attempts to measure the overhead of the primitives, not the ability to perform parallel tasks. Similar to the previous linux results, the rwlocks are again the better performers, while both APR mutex implementations suffer from an apparent exponential scalability problem as more threads contend for the lock.



Solaris is impressively stable as the number of threads increase during the test. Since APR's normal mutexes and rwlocks are implemented on Solaris as a very thin layer above the operating system, this test demonstrates Solaris' impressive ability to scale with both rwlocks and mutexes. The severe penalty for using APR's nested mutexes is very evident in this graph.



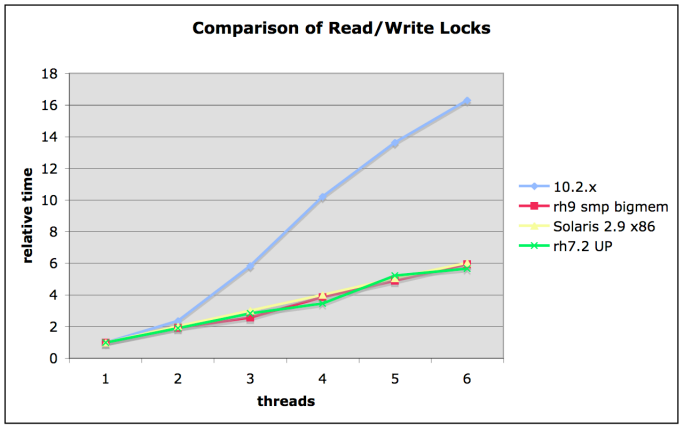
Linux is the obvious underachiever this this comparison. The likely reason that RedHat 9 suffers in this test might be due to the multi-processor aspect of the system. If Linux is running each thread on different processors, then those processors are having to contend for inter-processor resources, which will incur a severe overhead. RedHat 7.3 does not have the newly improved NPTL library, and therefore is suffering under the old LinuxThreads library. Both Jaguar and Solaris are able to perform quite well in this area, and both appear to have a linear or near-linear scalability.



APR's nested lock mutex adds an overhead that becomes very noticable in this graph. Since Solaris has such minimal overhead, it appears to be performing far better than the other platforms in this graph. One likely cause of the apparent "smoothing" of the curve for Redhat 9 might be due to the fact that the relative high-speed of the CPUs are able to overcome the overhead introduced by APR, and are getting closer to the inherent scalability factors of the OS.

Relative R/W Lock Performance

Comparing Read/Write Locks



lower is better

This graph reveals the most surprising results so far. It appears that both the new and old Linux implementations and Solaris all have linearly-scaling rwlock implementations. This is encouraging for those of you who are deploying your applications on these platforms. Jaguar appears to need some attention in its rwlock code, so it would be interesting to compare against the (now over a year old) Panther.

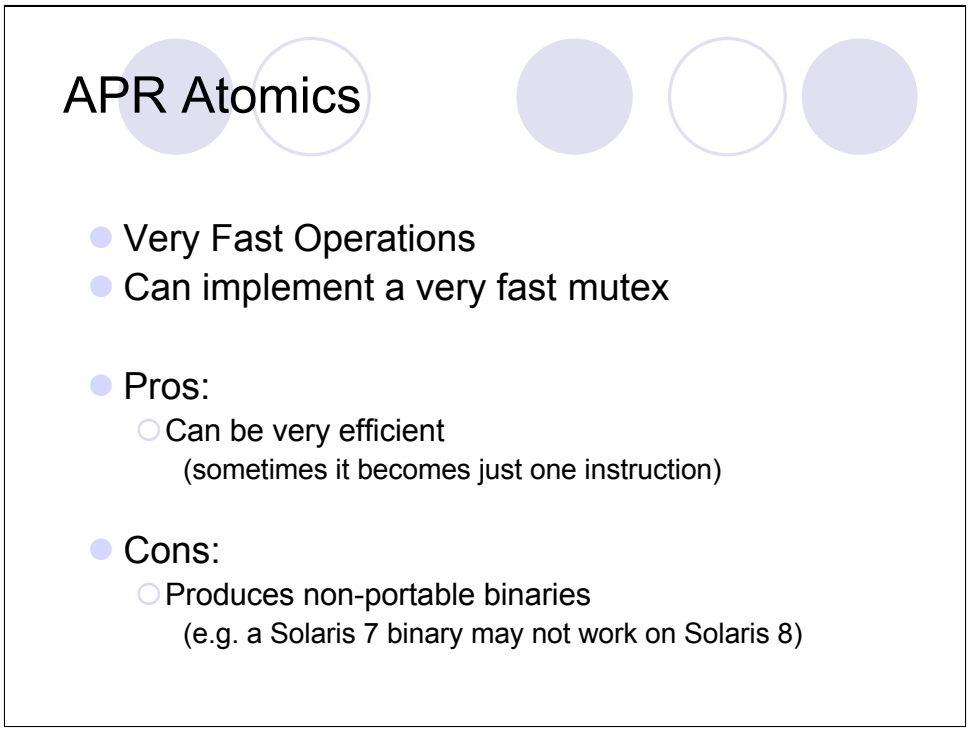
R/W Locks vs. Mutexes

- Reader/Writer locks allow parallel reads
- APR's nested mutexes are slow
- Reader/Writer locks tend to scale much better
- SMP hurts lock-heavy tasks

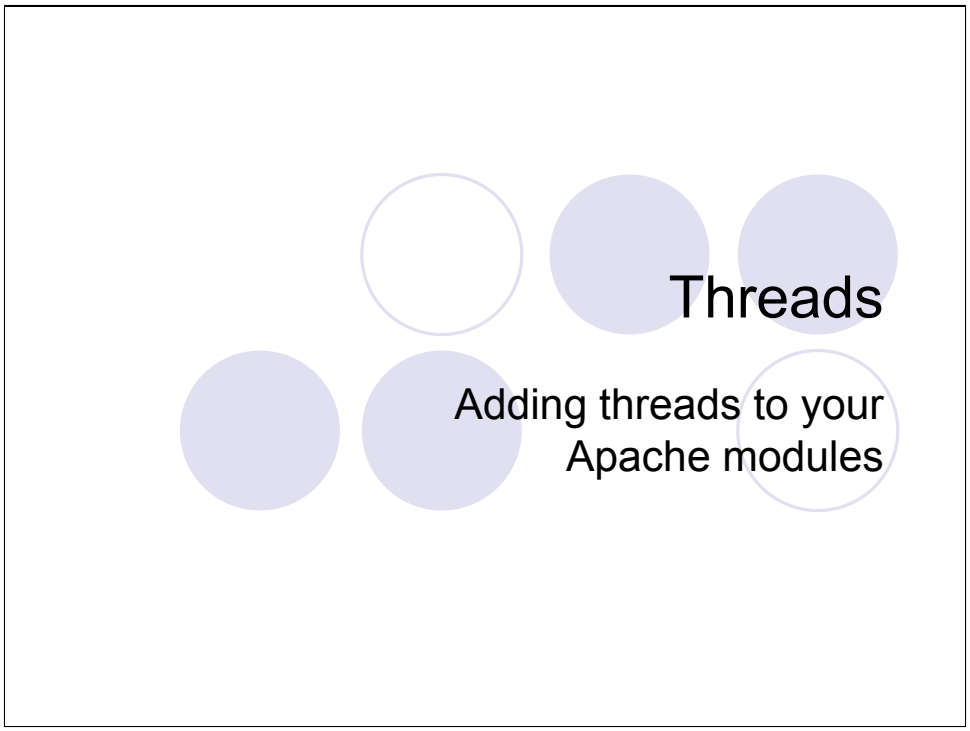
Behind the scenes, many rwlock implementations are actually using a mutex and a condition variable or two and a queue of readers and writers in order to resolve contention between the different locking classes. Manipulating those lists can be quite expensive when compared to a simcple mutex. Unfortunately APR can not guarantee the same type of performance response across all supported platforms, but it does its best to get out of the way of the OS.

OS Observations

- Solaris has very fast and stable locks
- Linux struggling but getting faster
  - NTPL shows improvement in overall thread performance, but not in lock overhead.
- MacOS (Jaguar) is stable and moderately fast
  - rwlocks could be improved



Recently an atomic-lock interface was added to APR. With it came some benefits and drawbacks. Although atomics can provide a very efficient locking mechanism, implementations require intimate knowledge of the underlying system, and often result in non-portable executables.


Threads
Adding threads to your
Apache modules

Why use threads in Apache?

- background processing
- asynchronous event handling
- pseudo-event-driven models
- high concurrency services
- low latency services

There are some cases where adding a thread to your module may prove beneficial. Even outside of the realm of Apache modules, you may wish to write a standalone application that takes advantage of APR's thread creation code.

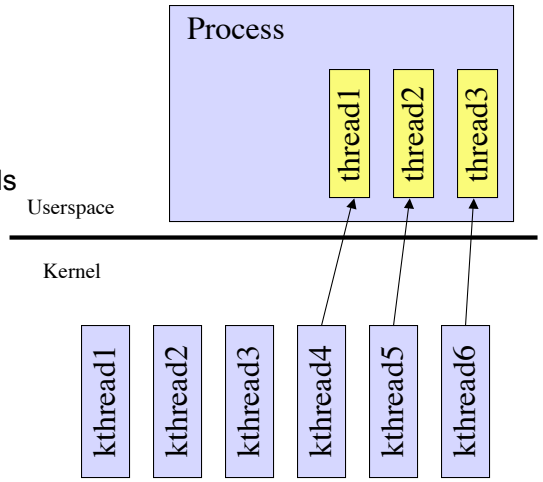
Often times by sticking a thread on an IO-bound operation, one can effectively simulate a fully event-driven model without having to explicitly deal with an event loop. Many thread libraries are very efficient at managing multiple IO-bound threads.

Thread Libraries

- Three major types
  - 1:1
    - one kthread = one userspace thread
  - 1:N
    - one kthread = many userspace threads
  - N:M
    - many kthreads ~= many userspace threads

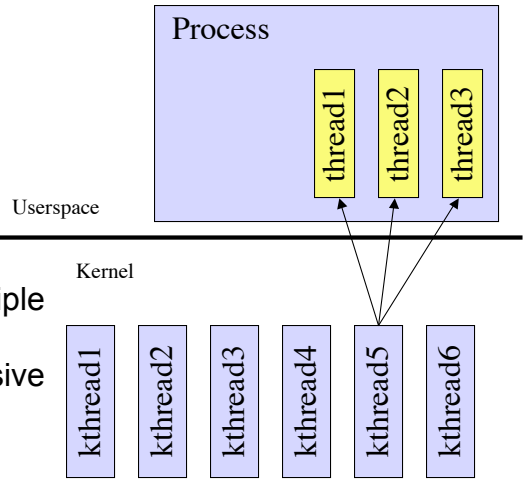
1:1 Thread Libraries

- E.g.
  - Linuxthreads
  - NPTL (linux 2.6?)
  - Solaris 9+'s threads
  - etc...
- Good with an O(1) scheduler
- Can span multiple CPUs
- Resource intensive



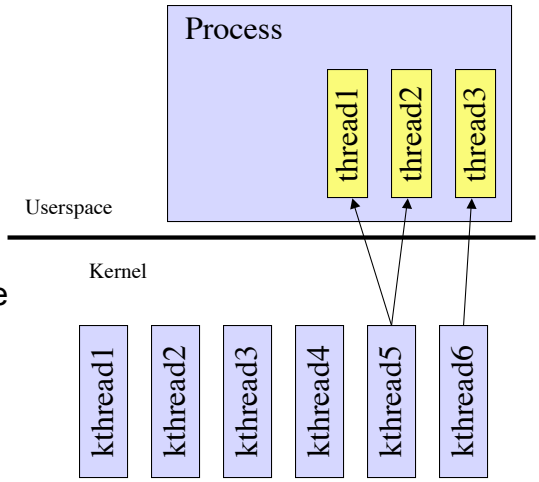
1:N Thread Libraries

- E.g.
  - GnuPth
  - FreeBSD <4.6?
  - etc...
- Shares one kthread
- Can NOT span multiple CPUs
- Not Resource Intensive
- Poor with compute-bound problems



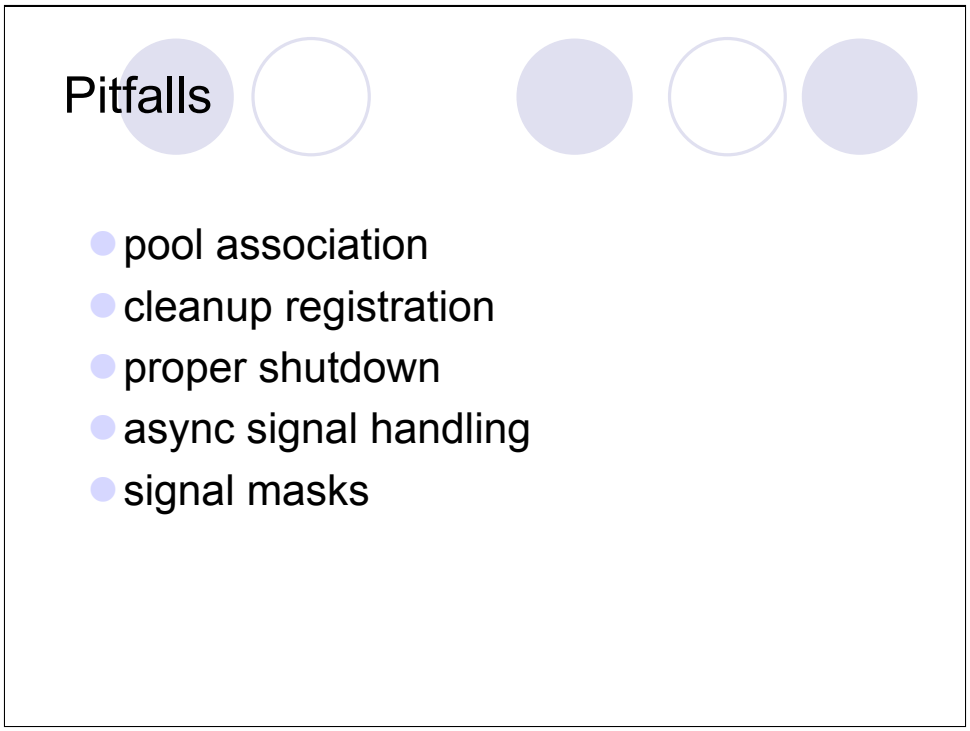
M:N Thread Libraries

- E.g.
  - NPTL (from IBM)
  - Solaris 6, 7, 8
  - AIX
  - etc...
- Shares one or more kthreads
- Can span multiple CPUs
- Complicated Impl.
- Good with crappy schedulers



See here for a good writeup comparing NPTL to NGPT:

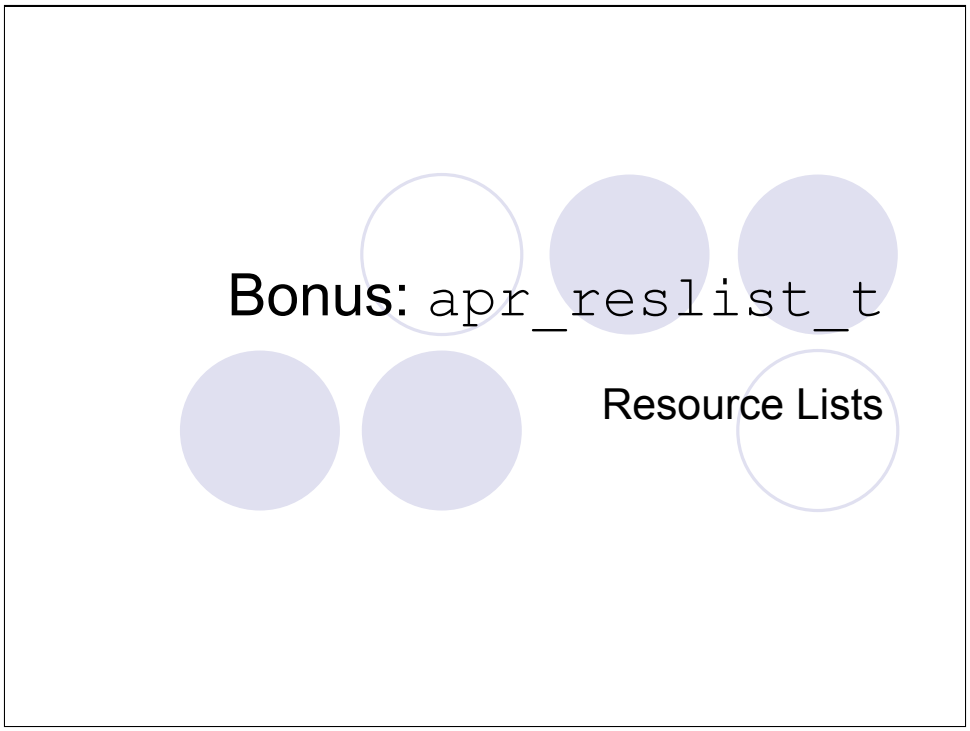
http://www.onlamp.com/pub/a/onlamp/2002/11/07/linux_threads.html



Since threads are always associated with a pool, it can be troublesome to synchronize the termination of that pool with the termination of the thread in that pool. APR does **not** terminate threads when their containing pool is cleared or destroyed. This has caused many elusive problems, and is an unfortunate side-effect of tying APR threads to pools.

Often, to maintain a small amount of synchronicity between threads and different events like the shutdown event, it is necessary to register a cleanup routine with the containing pool that somehow signals to the thread that it should shut down. APR does not provide for a way to terminate threads asynchronously, other than terminating the process itself.

There are some platforms that tend to have buggy or incomplete thread libraries. In general, there are very few platforms that behave in the same way, in terms of signal handling, signal masks, and preemption. APR is only able to provide for the lowest-common denominator of abilities, so that it can achieve a common set of portable routines.


Bonus: apr_reslist_t
Resource Lists

Resource Pooling

- List of Resources
- Created/Destroyed as needed
- Useful for
  - persistent database connections
  - request servicing threads
  - ...

Reslist Parameters

- min
  - min allowed available resources
- smax
  - soft max allowed available resources
- hmax
  - hard max on total resources
- ttl
  - max time an available resource may idle

Constructor/Destructor

- Registered Callbacks
- Create called for new resource
- Destroy called when expunging old
- Implementer must ensure threadsafety

Using a Reslist

1. Set up constructor/destructor
2. Set operating parameters
3. Main Loop
    1. Retrieve Resource
    2. Use
    3. Release Resource
4. Destroy reslist


Thank You
The End