Some Performance Patterns

Author Prof. Peter Sommerlad
HSR Rapperswil, 8640 Rapperswil, +41-79-432 23 32
peter.sommerlad@hsr.ch

Abstract This collection of patterns revives some old wisdom of experienced programmers with Sidestep System Calls and explains tactics to improve the performance and predictability of multi-threaded applications with Locking Categories and Thread-local Memory-Pool. They are mined and applied in server applications that provide services for a large number of users and for users that are sensible to fast response times.

Overview Today, many developers and software architects are shielded from the low-level consequences of their doings that they no longer can be aware of the performance issues. Several factors have created such a situation:

- Low-level programming is "uncool", because often the abstractions provided by an operating system are too cumbersome to use efficiently.
- Moore's law makes hardware faster and faster.
- Popular high-level languages, tools and libraries provide a lot of useful functionality without giving awareness of performance implications. And some tool architectures really are slow.

In the early days of mass-market internet, users connected via relatively slow modems and the bandwidth of the connection was the limiting factor of an application server's performance. Today, broadband access is becoming more an more popular and slow response times and high latency not only annoy users but also limit effeciency in web-based work places.

Bad performance is particularly a problem for developers creating server applications that need to handle either a large number of concurrent users or users that expect immediate responses.

Not only performance, but also predictability of behavior is becoming important in a 24x7 operating condition. First, crashes and recognizable down-times become less acceptable. Second, multiple services on a single machine call for predictable behavior in terms of resource utilization, so that one service running under load doesn't block another on the same hardware.

The patterns presented here cover some old folk wisdom with Sidestep System Calls and its specialization Locking Categories. In addition it introduces a new one for multi-threaded server architectures with Thread-local Memory Pool that in turn helps to implement Locking Categories more efficient for transient, but dynamically allocated objects.

References

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[POSA96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-oriented Software Architecture–A System of Patterns*, J. Wiley & Sons, 1996

[POSA2000] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann: *Pattern-oriented Software Architecture Volume 2 –Patterns for Concurrent and Networked Objects*, J. Wiley & Sons, 2000

[PLOPD3] *Pattern Languages of Program Design 3*, Addison-Wesley, 1997

[EPLOP1998] P.Sommerlad, M.Rüedi: *Do-it yourself Relfection*, EuroPLoP 1998

[KircherJain] M. Kircher, P. Jain*: Patterns for Resource Management,* Workshopped at EuroPLoP 2002

Credits Thanks to my shepherds for EuroPLoP 2002 Michael Kircher and Ali Arsanjani and the workshoppers in Irsee for their suggestions and feedback.

Sidestep System Calls

In applications or systems where performance matters, decrease the number of system calls made, because they are an order of magnitude more expensive than regular function calls within a program. Especially on modern hardware with large register sets, caches and multiple CPUs, the penalty payed per system call is relatively high. Understand how your program behaves with respect to system calls and learn how you can reduce the number of system calls required. For example, employ mechanisms like buffering, cacheing, or avoid unnecessary system calls by improved architectural structure.

Also Known As Buffering, presented as Avoid System Calls at EuroPLoP 2002.

Example Consider you write a simple program counting the occurences of the letter 'A' in a file. When you just rely on using Unix system calls `open()` and `read()` you might end with a solution like this:

```
/* program 1 */
 #include <fcntl.h>
 int main(int argc,char **argv)
 {
     char c; int counter=0;
     int fd = open(argv[1],O_RDONLY);
     if (fd < 0) return;
     while (read(fd,&c,1) == 1){ if ('A'==c) counter++; }
     printf("number of A's:%d",counter);
 }
```

when you run this program on a large input it is significantly slower than the following program using the stdio library’s `fopen()` and `fgetc()`:

```
/* program 2 */
 #include <stdio.h>
 int main(int argc,char **argv)
 {
     int c; int counter=0;
     FILE *f =fopen(argv[1],"r");
     if (!f) return;
    while ((c=fgetc(f)) != EOF ){ if ('A'==c) counter++; }
     printf("number of A's:%d",counter);
 }
```

On my system timing the programs with `/usr/share/dict/words` results in

```
program 2                    program 1
 real    0m0.038s             real    0m0.437s
 user    0m0.040s             user    0m0.130s
 sys     0m0.000s             sys     0m0.290s
```

As you can see the second program using stdio's buffering mechanisms is more than 10 times faster than the first one.

Context You are developing an application or a system that expects either very high load or stresses the limits of its hardware and operating system.

Problem Omnipotent libraries and naive programmers tend to neglect the performance requirements of some applications. On the other hand, libraries or programmers can sometimes provide optimizations that are useful.

Program code runs fastest on modern system if it fits into the processor's cache memory and employs no calls or data accesses outside the cache, thrashing the cache's content. A major reason for calling outside of the current code context is a call to the operating system. In addition to thrashing the cache memory, the processor has to change its internal mode of operation and raise the privilege level. The operating system needs to save the process' execution context and establishes the kernel context to perform the OS' call (and vice versa on return). Thus this is a relatively costly operation compared to a regular in-program function call.

On the other hand, program code that does something useful needs to have an external (lasting) effect. This can only be obtained by calling the operating system's features.

With many operating systems (especially UNIX) such calls are an order of magnitude more expensive than a regular function call within the program. In addition often data needs to be passed from the process' user address space to or from the OS' address space to obtain a desired result (i.e. write the contents of a file).

Whenever the operating system needs to perform input or output (i.e. disk, network), the communication aspect with the peripheral units adds another order of magnitude overhead. Reducing the number and frequency of such expensive system calls can give a program a boost in performance and a drop in the usage of system resources.

How can you minimize the number of operating system calls in your program to improve its performance?

In particular you want to address the following *forces*:

- calling the operating system is expensive, but
- system cannot be avoided completely.
- a process or thread of your system will be interrupted within a system call, if a resource is not available, thus giving other threads or processes the chance to continue. The overhead for switching to and from a process or thread increases the overall performanc impact.
- a long running thread or process will be interrupted when its time slice is exceeded, even when it is not issuing system calls.
- some libraries provide mechanisms to reduce the number of system calls, i.e. by buffering (stdio, iostreams).
- some libraries hide or encapsulate system calls so deeply, that a programmer is unaware of the implications using them. [1]

Solution The solution is two-fold: There is the technical aspect of using mechanisms like buffering and caching to reduce the need to call the OS too often. The other is the human aspect of educating programmers to understand the implications of their code or of the libraries or programming languages they are using.

For the technical aspect of the solution exist several popular approaches:

- *Buffering* input/output operations and issue a system call only when a buffer is full or empty (as in stdio or iostream).

- *Cacheing* of results (see Proxy pattern [GOF][POSA96]). Keep the result of an expensive operation in a proxy object and reuse this result later on instead of re-issuing the operation on the original

1. For example, many middleware infrastructures strive to give the programmer the illusion of a local call, when a remote procedure call happens. Unfortunately, the relative cost of such an external call increased with each generation of middleware from RPC, Corba, Java-RMI, J2EE Beans, SOAP. I wonder who is re-inventing this overhead over and over.

object. An example of such cacheing is the implementation BIND named of the Internet domain name service (DNS).

- *Architectural approaches* as demonstrated, for example, in the Thread-local Memory Pool pattern.

The human aspect is approached by writing or reading this pattern. In addition you can use tools provided by your development platform to learn about the system call behavior of your programs.

For a better understanding of implications imposed by system calls to your program use existing tools to monitor your program (for example, on UNIX use ps, top, and time), trace the system calls issued (e.g., truss or strace), or read the library source code and documentation to understand what is really going on under the blanket.

Example Resolved Tracing the number of system calls made for our two programs results in about 120 system calls for the efficient program 2 and more than 400'000 system calls for program 1. No wonder, the naive program is an order of magnitude slower.

```
strace ./program2 /usr/share/dict/words 2>&1 | wc -l
129
strace ./program1 /usr/share/dict/words 2>&1 | wc -l
408889
```

However, fortunately we humans never need to stop to learn. Look at the following program, using the more sophisticated `mmap()` system call for counting the 'A's. This program runs about 4 times faster than program 2 and uses only about 25 system calls (the majority used by the dynamic library loader on my system).

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
int main(int argc,char **argv)
{
    char c; int counter=0;
    long l=0; const char *s;
    int fd = open(argv[1],O_RDONLY);
    if (fd < 0) return;
    l=lseek(fd,0,SEEK_END);
    s=mmap(0,l,PROT_READ,MAP_PRIVATE,fd,0);
    if (s == MAP_FAILED) return;
    while (l--){ if ('A'==*s++) counter++; }
    printf("number of A's:%d",counter);
}
```

On my system I get the following timing:

```
real    0m0.009s
user    0m0.010s
sys     0m0.000s
```

Known Uses

The technical aspect of this pattern is omnipresent in many libraries dealing with input or output, among them are stdio, iostream or Java's buffered streams that can improve performance of a solution dramatically as shown in the example.

However, as stated above, the opposite is also true, for libraries or middleware hiding from the programmers the huge performance impacts of using distributed components.

The human aspect today falls short, but this pattern is a step in educating you the reader to take care.

Consequences

The pattern implies the following **benefits**:

- reducing the effective number of system calls made can improve the performance of your program, if it needs so.
- libraries or your own Wrapper Facades [POSA2000] to system calls can implement buffering or cacheing and provide a more convenient API than bare-bone operating system calls. In addition higher-level error handling can be easier to deal with than the operating system's abstractions.
- learning the implications of your deeds can make you a better programmer

However, the Sidestep System Calls pattern also has its **liabilities**:

- Complex mechanisms for buffering or caching are error prone if implemented yourself and can be hard to maintain. Libraries take time time to mature in that respect.
- Premature optimization or optimization without need can lead to obscure and hard to maintain programs. Often the speed of a program is a neglectable today.
- It can be hard to retro-fit an existing system to employ significant fewer system calls.

- Error handling can be more fine grained and deal with individual situations, if you program on a system call level. Thus your program can behave better that just saying “sorry” to the user. On the other hand, doing good and failure resistent error handling on the level of system calls can increase complexity of your code heavily.
- Sometimes avoiding system calls can make an application more fragile or hard to debug. One example that happened to me was writing a log file with a large in-program buffer. The contents of the log file itself never showed where the application really crashed, because in case of the crash the relevant log data has not been written to disk.

See Also Locking Categories: locks are implemented by system calls, therefore minimizing the number of locks needed is a special case of Sidestep System Calls.

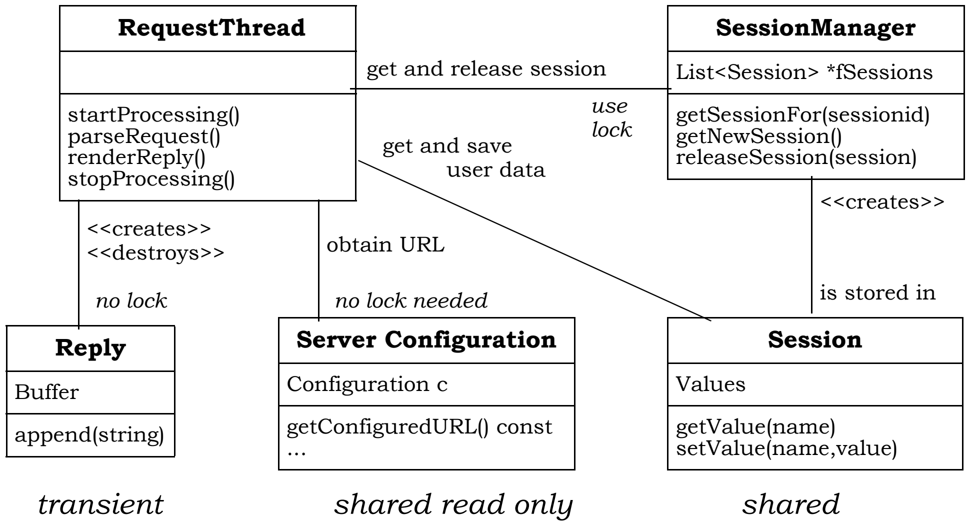
Thread-local Memory Pool minimizes the use of locks required for allocating and deallocating objects.

Locking Categories

When programming multi-threaded systems, thread-safety is omnipresent. However, locking any object or method that might be shared somewhen carries not only the danger of deadlocks but also even if deadlock-free, the dread of poor performance. Good locking strategies as the one suggested by this pattern are therefore needed. This pattern gives you categories of objects to look for, so that you can carefully minimize the need for locking.

Also Known As This pattern was named Minimize Locking when presented at EuroPLop 2002.

Example Consider an application server, that keeps user sessions and serves request within individual threads. Each shared resource, that such a server uses, needs to be secured by locks. Otherwise, the threads running in parallel might access such a shared resource in an inconsistent state.



One such shared resource is the session manager object, that keeps a list of active user sessions. Each request handling thread will obtain the session for a request from the session manager. Such a session can be newly instantiated by the session manager. In turn the manager modifies its list of sessions. A request of a returning user will get its already existing session object from the session list by the session manager. Multiple requests occur in parallel, therefore the session manager must use a lock to serialize access by request threads.

Another shared resource our server provides is its current configuration in a server configuration object. This object holds all configuration data of the server like the URL to process. It is initialized on server start-up and never changes later on. Threads processing requests need to access the server configuration object to obtain the URL of the server, because it must be rendered to the reply. Caution will tell you also to use locks for the server configuration object, because it is accessed simultaneously.

Each request processing thread builds up its output in a reply object that resembles an in memory buffer, before the result is sent over the network back to the client. Such a reply buffer will only be accessed by an single request processing thread and never used by another thread.

Context You are developing a multi-threaded (server) application that requires access to shared resources.

Problem One common solution to the situation where several threads access a shared resource (i.e. an object) is to provide a lock, that must be acquired for every access to this object (for example, via java's `synchronized` keyword).

Without locking you cannot guarantee that your system runs correctly, because a thread might read inconsistent information from the shared resource while another one is updating this information. This can result in your system crashing or misbehaving.

However, using locks for each access to a potentially shared object can be very expensive: instead of a memory access, you need a system call to acquire a lock before you access the object and release the lock by another system call after you are done.[2] As we have learned from Sidestep System Calls this can result in bad performance.

2. On a multi-processor hardware, lock acquisition needs to synchronize all processors and caches, making it especially expensive.

Many operating systems provide read-write locks, that allow multiple simultaneous threads read an object preventing all others to write at the same time. Read-write locks promise increase in parallelism. This pays off, when you only rarely update a shared object.

How can you implement a multi-threaded system, that does not pay the performance penalty of too many locks?

In particular you want to address the following *forces*:

- You must protect a shared mutable resource by a lock, so that all threads have a consistent image of it. Neglecting locks will lead to systems that crash.
- Excessive use of locks not only is expensive, but can also lead to race conditions or deadlocks, when locks are applied without a clear and working strategy (see also Thread-Safe Interface pattern [POSA2000]).
- Using read-write-locks allows you better sharing of a resource read mostly, but still requires a system call per acquire and release.
- Lock acquisition and release are expensive operations, especially on multi-processor hardware and can be hard to implement correctly (see Double-Checked Locking pattern [POSA2000]).
- You have decided that a simpler single-threaded design like shown in the Reactor pattern [POSA2000] cannot meet the goals of performance and system utilization.
- Read-only objects can be accessed by multiple threads in parallel without requiring locks.
- Objects allocated on the heap, that are only used by a single thread still require obtaining a lock by the memory management library, because the heap memory itself is a shared global resource. Only stack-allocated objects are really local to the thread and require no locks at all.

Solution There is no boilerplate solution to build a system just using the minimal number of lock acquisitions and releases. However, this pattern represents an engineering strategy to create a system, that uses a less locks than a naive design might come up with.

In a server application you can classify your objects into three categories:

- **transient** objects that will ever only be used by a single thread, i.e. those that are only useful during the processing of a single request (see the Reply Buffer object in our example). Objects allocated on the stack are examples of such objects. The Thread-local Memory Pool pattern allows for efficient allocation of transient heap objects without locking.
- **shared** objects that are used by several threads, potentially in parallel and thus require locks. The session manager of our example definitely is a such a shared object.
- **read-only shared** objects that are created by a single-threaded or contention-free initialization phase (i.e. according to Eager Acquisition pattern [Kircher Jain]) and do not change their statelater on, like the server configuration object. Thus, they can be employed by several threads in parallel without the need for locks, not even read-write locks.

To minimize the overall locking overhead, you can strive to optimize the number of objects in the shared category. Too many shared objects might result in too many locks to acquire and release, even when good parallelism is won. Too few shared objects (e.g. one) might limit throughput, because of contention for the lock of these objects. Your goal must be have no objects unnecessarily in the shared category.

Access to shared objects should be implemented using the Thread-Safe Interface pattern and if you are using C++ by the Scoped Locking idiom [POSA2000].

Configuration objects, that are initialized on system startup but later on not changed, fall in the category read-only shared. They can provide flexible, data-driven operation of your system, without the need to sychronize their access during normal system operation. Having such objects pre-allocated and initilized is shown in Eager Acquisition as opposite to what Lazy Acquisition pattern [Kircher Jain] and Double Checked Locking Optimization pattern [POSA2000] talk about.

Structure You will have the following types of components in your resulting system:

Initializer running only at system startup in a single thread. An Initializer sets up all *read-only shared* objects. It might also initialize shared objects.[3]

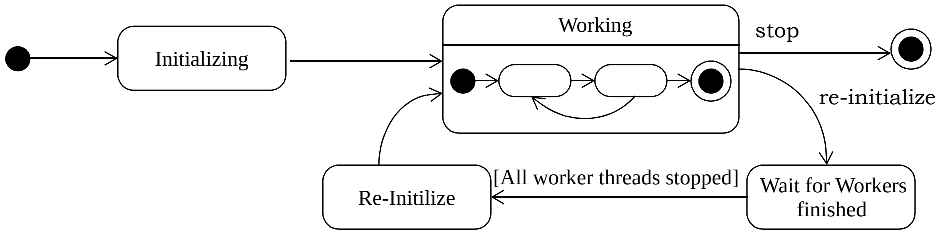
Worker Threads are started after the Initializer is done. You will have multiple, for example in a thread pool as shown in Leader/Followers [POSA2000].

Shared Objects represent shared resources with an associated lock (perhaps a read-write lock) used by Worker Threads.

Read-only Shared Objects represent shared resources that do not change after the Initializer is done.

Transient Object corresponds to objects created (and destroyed!) by a thread that is never passed across the thread's boundary while it is in use.

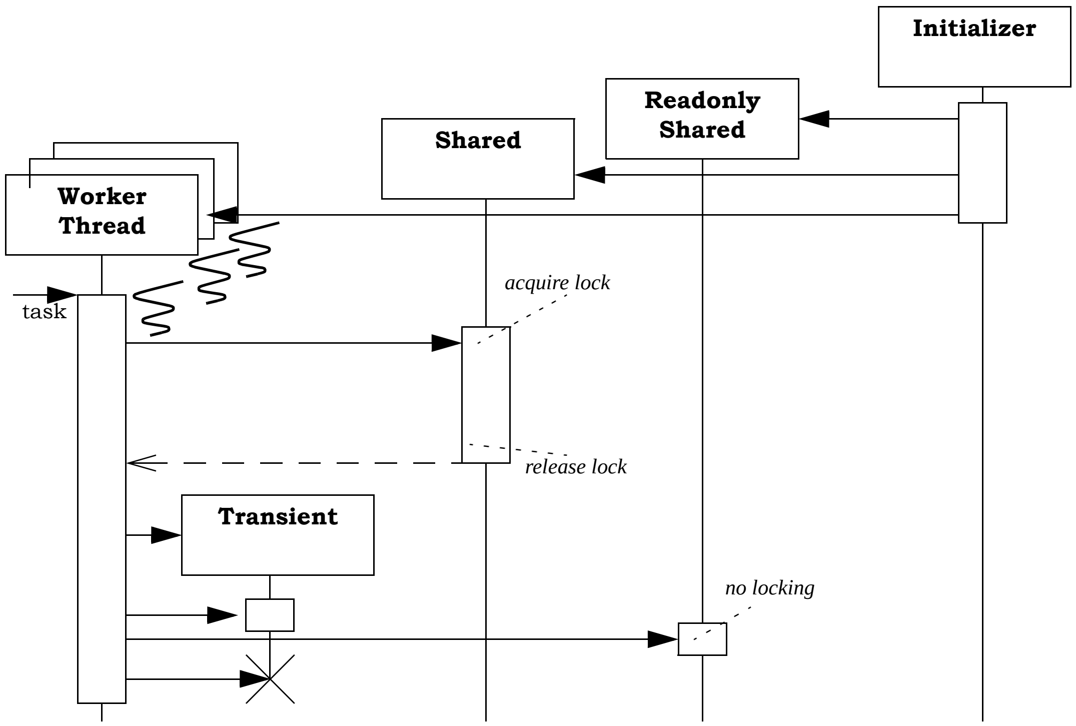
Dynamics To allow for read-only shared objects your system needs to be working accordig to the following phases:



- Initializer started and initializing (1 active thread):
  - initialize read-only shared objects
  - initialize shared objects
  - initialize Thread-Local Memory Pool (optional, see next pattern)
  - initialzie Worker Threads in a Thread Pool

3. The Initializer is not shown in the diagram of the example.

- start normal operation:
  - start Worker Threads in the Thread Pool.
  - within each thread: create, use and destroy transient objects and access read-only shared objects without locks.
- stop normal operation for re-initilization:
  - stop threads, each thread will release its transient objects.
  - wait until threads are finished and only one active performing the re-initialization remains.
  - re-create/re-initialize shared and shared read-only objects
- stop the entire system.



Providing a means of re-initialization without completely stopping the system can be a daunting task. You need to provide a means to stop all threads in the thread pool and enter a mode where only one active thread remains that is used for re-initialization. However, it might much easier and sufficient to just simply kill and restart the system. For example, stopping the worker threads might take a long time, when some threads are waiting for I/O to complete and cannot be interrupted in this state. Killing the process by brute force, the operating system guarantees that all threads and I/O is stopped. However, active users of your system might have some inconvenience.

An alternative solution for re-initialization, outside the scope of this pattern, is to take read-only shared object as almost read-only and provide them with a read-write lock. So re-initialization needs to obtain a write lock on these objects before changing. Such a solution significantly increases the number of locks and lays the burden on the worker thread’s code to obtain and release the read locks often. Otherwise the re-initialization cannot take place when worker threads are running.

Implementation To implement the Locking Categories several activities are needed:

1 First, you have to keep an eye on what locking behavior your system will require or actually has. A careful analysis might come up with objects in each locking category.

2 Implement the Initialzer, that allocates and initializes all shared and read-only shared objects. Often an application server provides such an initializer in a generic form, where configuration data actually tells what objects to create and how to set up their state.

3 Implement your shared objects. Add a lock to each such object and use the Thread-Safe Interface pattern [POSA2000] to clearly distinguish external operations, that can be called from different threads and that acquire the lock from the internal ones that assume the lock is already acquired. Nevertheless, be careful in your design and try to minimize the chance that a Worker Thread needs to keep several locks of different shared objects. This will increase the chance for deadlocks. In such cases, a helpful strategy is to always acquire such locks in the same order and to release all held locks if you cannot obtain the next one needed. The trylock operation usually is used in these cases.

4 You might be unfamiliar with the category of read-only shared objects, that are pre-allocated during initialization (see Eager Acquisistion pattern [Kircher Jain]). For example, the Server Configuration object is allocated and preset by the Initializer and later

on only accessed in a non-mutating way. However, you must clearly document such behavior and if you are providing a framework using such read-only shared objects, ensure that framework users do not introduce mutating operations when deriving new subclasses instantiated by the Initializer as read-only shared objects. One solution to avoid application of mutating operations on read-only shared objects is to provide read-only adaptors to the pre-allocated objects. After initialization, the read-only shared objects are only accessed via these adapters ensuring the non-mutability.

5 Transient objects are only created and used by one Worker Thread. If they are allocated on the stack (as it is possible in C++ compared to Java) there is no locking overhead at all. However, heap allocation of transient objects requires locks for allocation and release of the memory (see Thread-local Memory Pool to sidestep these locks).

Known Uses SYNLOGIC's server application framework WebDisplay employs the strategy presented by Locking Categories. WebDisplay uses read-only shared objects heavily to avoid locking. It uses locks for its session list manager and individual session objects. Each request context object, most strings, Anythings [EPLOP1998] and iostream objects are transient objects thus not requiring locks.

TAO distinguishes between stack, heap and synchronized objects, which kind of falls in the three categories.

Apache Tomcat does something similar with request/session/ application objects.

Variants For the sake of simplicity this pattern talks about multi-threaded systems with process-internal shared resources, but almost everything also applies to multi-process systems with external shared resources (e.g. shared memory or files). The Apache httpd server (up to 1.3.x) initializes itself on start-up creating read-only shared objects representing its configuration and then spawns further child processes for request processing. The child processes inherit the read-only shared objects from their parent and if they never change data, the operating system effectively keeps only one copy of the according memory pages.

See Also A special case as well as an implementation option for making heap objects of the category transient is the Thread-local Memory Pool.

Eager Acquisition [KircherJain] shows consequences of up-front initialization.

Thread-Safe Interface [POSA2000] is a good strategy to implement objects in category shared.

Consequences The pattern implies the following **benefits**:

- you cannot avoid the need for all locks in a multi-threaded system, but following the guidelines of the pattern you might be able to reduce the number of locks during request processing close to the minimum.
- the object categories give you a model to work with while designing your system. Thus you can strive to minimize the number of locks by reducing the number of shared objects.
- With read-only shared and transient objects, that do not need locks you can effectively reduce the number of locks required.

However, the Locking Categories pattern also has its **liabilities**:

- Even with the clear distinction of the object categories, multi-threaded programming is hard. Without care it is easy to implement chances for deadlocks or race conditions.
- Implementing an on-the-fly re-configuration in a safe way can be beyond the abilities of your threading architecture and you might need to introduce locks you tried to get rid of by using read-only shared objects.
- Your application might not easily fit with the three categories and you end up with too many objects in the category shared. Then this pattern might be of little help. Too many shared objects, might be a sign of a poor design strategy, that might lead to deadlocks and race conditions, when a thread needs to lock several objects to perform an operation.
- On the other hand, using only a few large grained shared objects employed by many worker threads can serialization of worker threads, thus eliminating performance gains by multi-threading.

Thread-local Memory Pool

If you are implementing Locking Categories you might be confronted with transient objects that require dynamic memory management, such as strings. However, heap memory is a global shared resource, so heap management requires locks to protect its data structures. A universal memory manager can not be aware that memory will only be used and freed by the thread requesting it. Thread-local Memory Pool shows you a way out of this dilemma, by implementing your own allocators for each thread and using them for transient objects. Access to allocators is implemented using the Thread-Specific Storage pattern [POSA2000], so that classes and programmers usually do not need to deal with the issue of having different allocators.

Example Your multi-threaded web application server uses a thread per request architecture with thread pool. During request processing your code creates more and more of the web page content. To be able to give a definitive size of the result you first collect the page content in a string buffer (reply object). However, this string grows and might require frequent re-allocation resulting in locking overhead, even though it can be categorized as transient, that wouldn't need any locks by itself.

Context You are applying the Locking Categories with transient objects that require dynamic memory management in a multi-threaded system. Your implementation language allows you to implement your own memory management features, like C++.

Problem Even when you try to minimize locking you face the challenge that object-oriented programming often requires you to allocate objects on the heap, i.e. using `std::operator new()`. Unfortunately the programs heap is a resource shared by all threads and thus dynamic memory management requires locks, even when needed by transient objects. Especially data objects like strings suffer from frequent allocation or reallocating of buffer space on the heap, even if the object itself is placed on the stack in C++.

How can you alleviate this situation with transient objects needing heap allocation without paying the penalty of aquiring and releasing the memory management lock?

In particular you want to address the following *forces*:

- heap memory is a global shared resource. allocating and releasing memory each requires locks on its management data structures.
- you do not want to pay the price of locking overhead for objects used only by a single thread.
- objects that are passed up the call chain need to be allocated dynamically and cannot be allocated just on the stack.

Solution Implement your own memory manager that provides a memory pool for transient objects in each thread. Access the individual memory manager by using Thread-Specific Storage [POSA2] so that programmers automatically access the right one. These thread-local memory managers do not require locks for their management data structures.

If you have classes that will be instantiated in transient thread-local fashion as well as shared objects you also need to keep track which allocator was used for a specific instance. You wrap the global heap manager with the same interface as your thread-local allocators.

Dynamics The dynamics are similar to Locking categories. Just the Initializer needs to be extended to set up the thread-local memory pools and their references in thread-local storage when setting up the worker threads.

Implementation Implement thread-local memory pools following these guidelines

1 Define the API for allocators. For example:

```
class Allocator  {
public:
    Allocator(long allocatorid);
    virtual ~Allocator();
    void *Calloc(int n, size_t size)
    void *Malloc(size_t size) { return Alloc(size);}
    virtual void  Free(void *vp) = 0;
    static Allocator *Current();
    static Allocator *Global();
protected:
    //!hook for allocation of memory
    virtual void *Alloc(u_long allocSize) = 0;
};
```

2 We implement the auxiliary methods like Calloc, based on the hook method Alloc. Note this code is simplified neglegting error conditions.

```
void *Allocator::Calloc(int n, size_t size)
{
    void * ret = Alloc(n*size);
    if (ret && n*size>0) memset(ret,0,n*size);
    return ret;
}
```

3 Implement the interface using the regular global (malloc) allocator, so that at least one is available as a default global allocator to use.

```
class GlobalAllocator: public Allocator {
public:
GlobalAllocator();
virtual ~GlobalAllocator() {}
virtual void  Free(void *vp) { ::free(vp);}
protected:
static Allocator *getInstance(); //Singleton
virtual void *Alloc(u_long allocSize)
{ return ::malloc(allocSize);}
};
```

Access the global allocator in `Allocator::Global()` using the GlobalAllocator's Singleton [GHJV95] implementation:

```
Allocator* Allocator::Global()
{ return GlobalAllocator::getInstance();}
```

4 Implement an allocator with a allocation strategy suited to your application that is largely independent of the global allocator. One example is using a pool of pre-allocated memory. The GNU malloc library implements a strategy using the mmap system call with anonymous files for getting memory chunks (so-called arenas) outside the process heap-space. For brevity, we leave out the messy details of pooled memory management and just give the class declaration:

```
class PoollAllocator: public Allocator {
public:
PoolAllocator();
virtual ~PoolAllocator() {}
virtual void  Free(void *vp) ;
protected:
virtual void *Alloc(u_long allocSize) ;
};
```

5 Extend your system's initializer with code to set up a memory pool for each thread. Each pool will allocate its initial pool space from global heap memory using Eager Acquisition [KircherJain]. If the worker threads are also managed in a fixed sized pool, you can get a stable,

predictable memory footprint of your system, if you choose the memory pool sizes are big enough for normal operation.

6 Use Thread-specific Storage for keeping the pointers to the thread-local memory pools you initialize. Access these allocators in `Allocator::Current()` like the following:

```
Allocator::Current()
{
    Allocator *current =
        (Allocator *)pthread_getspecific(ALLOCATOR_KEY);
    if (current) return current;
    return Allocator::Global(); //fallback
}
```

7 For classes that are only used to create transient objects implement `operator::new()` and `operator::delete()` to use allocator `Storage::Current()`. Classes that have both use for shared objects as well as for transient objects and that are instantiated often enough to have recognizable locking overhead in the transient case, you need to keep track which allocator was used, so that a deletion of an instance asks the correct allocator to reclaim its memory.

8 Classes that manage their own buffers internally, may need to be explicitely given the allocator to use. For example, our `string` class is used both for transient strings and also for strings in shared objects. Thus, provide class `String` with an additional `Allocator*` parameter and use that allocator for managing the Strings buffer.

```
class String {
public:
String(Allocator *a=Storage::Current()):fAllocator(a){}
// all the uses of the Allocator are left for the reader
to imagine
private:
Allocator *fAllocator;
}
```

Note that the default allocator used is Storage::Current(). This way a developer does not need to care about allocators and does not need to specify the extra parameter.

Now, all transient string objects can use the thread-local pool allocator without incurring the overhead of a lock required by the default operator::new()/malloc() implementation.

Known Uses

SYNLOGIC's WebDisplay implement Thread-local Memory Pools and greatly benefited from the performance boost by reducint the number of locks used during multi-threaded processing.

Apache's http server version 1.3.x implements this pattern in a multi-process manner. Such a multi-process single-threaded server is an extreme variant of this pattern where the thread-local memory pool is trivially provided by the generic memory manager, but sharing objects is the complex case using shared memory system calls.

GNU's glibc's malloc implementation, goes in the direction minimizing locks for memory allocator access by using a clever schema of arenas (corresponds to a pool) and creating new arenas for each thread on the fly. However, because it cannot assign an arena strictly to a single thread the design is complicated and thus each arena requires its own lock. A thread uses thread local storage to keep its last used arena and a trylock call to that arena's lock. In addition the new arenas are obtained from the system by the use of mmap, so the orignal process' heap (the main arena) is still a single shared resource always using a lock. So with multi-threading and without mmap GNU malloc will acquire and release a lock for every call (malloc or free) to the allocator.

RogueWave ATS (application tuning system) reduces locking overhead, but its memory manager is not backed by application design. It might be better than the default memory manager, but still needs some locking when freeing memory, because it might have been allocated by a different thread. From an outside view, its implementation strategy looks similar to the modern glibc's malloc.

Consequences

The pattern implies the following **benefits**:

- You save the cost of acquiring and releasing the lock for each object allocation and deallocation for transient objects. This is especially true for strings or similar transient data objects (e.g. Anything [EPLOP1998]).
- Pre-allocating memory pools allows a stable memory footprint of your server, if pool size is adjusted accordingly. Such a stable memory footprint gives better predictability and monitoring capabiltiy.

However, the Thread-local Memory Pool pattern also has its **liabilities**:

- Implementing your own memory management is an advanced programming topic and might lead to headaches if you make errors.
- Thread-local Memory Pools require discipline. For example, if an object allocated thread-locally is passed to another thread, that deletes it, you easily get a crash. Such errors are very hard to detect.
- Using non-standard memory management can be error prone or at least bloat interfaces for classes requiring it. Use it only for classes where you know, that this optimization is worth the complexity.
- Tuning parameters of your memory pools (like the initial size, the increment when it overflows) might be too complicated.
- Pre-allocating memory pools wastes resources, if the pool size is too generous or most of your threads do not fill the pool's memory.

See Also Thread-Local Memory Pools use Eager Acquisition [KircherJain] for the pool memory.

Locking Categories is a prerequisite organization schema of your system, so that you can decide what objects are candidates that might profit from a thread-local memory pool.