Memory Management in K42

Jonathan Appavoo
Marc Auslander
Dilma DaSilva
David Edelsohn
Orran Krieger
Michal Ostrowski
Bryan Rosenburg
Robert W. Wisniewski
Jimi Xenidis

This document describes the kernel mechanisms that support virtual memory and the protocols used to communicate with clients that suffer page faults and with file servers that provide paging I/O.

1. Overview and Motivation

The K42 kernel memory management component manages page frames and the mapping of virtual addresses to these page frames. Paging I/O is offloaded to a file system or a swap system. K42's overall goals of high performance, multiprocessor scalability, and application specific customizability, as well as specific design decisions of other components, all constrain the memory manager design. We indicate some of these design goals and constraints here:

- Locality: Avoid global data structures and locks. Manage each processor separately.
- Uniform Buffer Cache: Have a single mechanism for caching file and computational data in page frames. This mechanism must be partitioned to maintain locality.
- Support user-mode scheduling: Avoid blocking for page I/O in the kernel. Reflect page faults and page-fault completions to the faulting process.
- Bounded Kernel Resources: Avoid blocking threads in the kernel. Rather, use a continuation style in which work is queued and then resumed when an interrupt or an interprocess message indicates that it can be continued.
- External File Servers: Provide interfaces so that processes other than the kernel can manage file system meta data and file system I/O initiation.
- Pageable kernel: Provide for paging much of the kernel, and particularly the paging system data needed to represent non-kernel processes.
- Unix Semantics: Provide efficient implementations of fork and copy-on-write mappings.
- Customizability: Allow for the possibility of introducing customized paging implementations for specific applications.
- Processor-Specific-Memory: Provide virtual memory locations in an address space that are backed by different page frames on each processor.

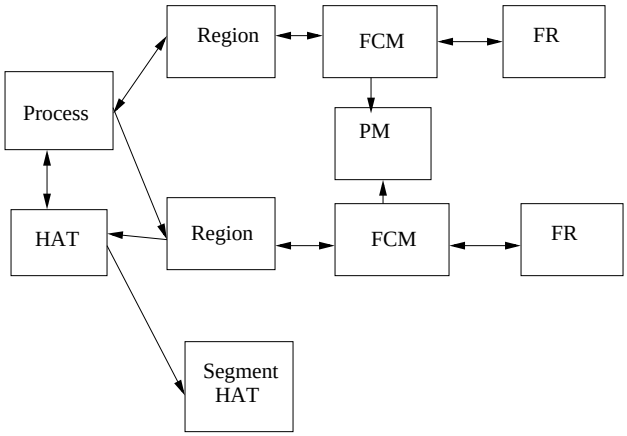

Figure 1. Memory Management Objects

2. Overall Structure

Each K42 Process contains a single address space. The address space is made up of Regions, each of which spans a range of virtual addresses in the Process's address space. A Region maps its range of addresses onto a range of offsets in a "file". We use the term file even for computational storage, although a special version of the "file" implements this case. A Process also contains the hardware specific information needed to map virtual addresses to page frames.

We represent this structure using K42 Clustered Objects connected by K42 object references.

- Process - root of the object tree representing a Process in the kernel. The Process maintains a list of the Regions that exist in its address space, and refers to the root of the hardware specific information.
- Region - represents the mapping from a contiguous, page-aligned, range of virtual addresses to a contiguous, page-aligned range of file offsets.
- FR - The File Representative represents the kernel realization of a file. It communicates with the external (e.g. file system) implementation of the file to do I/O and for other file system purposes. The FR also represents the kernel realization of the file to client implementations of an open file, and is used, for example, to memory map the file. See Section 4.1.
- FCM - The File Cache Manager controls the page frames currently assigned to contain file content in memory. It implements the local paging policy for the file and supports Region requests to make file offsets addressable in virtual memory.
- PM - the Page Manager controls the allocation of page frames to FCM's, and thus implements the more global aspects of the paging policy.
- HAT - the Hardware Address Translator manages the hardware representation of an address space.
- SegmentHAT - manages the hardware representation of a hardware segment. Segments are of hardware dependent size and alignment, for example 256 megabytes in PowerPC. The hard-

ware data they manage might be a segment's page table, or a PowerPC VSID. SegmentHAT's can be private or shared among multiple address spaces.

This particular decomposition was chosen because it was judged to separate mechanisms and policy that could be independently customized and composed. For example, a normal Region or one specialized to implement processor-specific-memory by including the processor number in its mapping of virtual address to file offset can both be connected to different kinds of files.

The next section follows a page fault through the objects introduced above. Following sections then discuss various design decisions and strategies.

3. History of a Page Fault

A page fault in a user Process causes a kernel thread to be allocated to run the page-fault handler. While that thread runs, the faulting "dispatcher" [1] cannot run, since its current state is represented in the page fault handling thread. A page fault in the kernel pushes the faulting state onto that faulting thread's stack. In either case, the handle fault method of the Process object for the faulting process is called. Locking during this processing is local to the objects involved, and follows the K42 strategy of releasing locks early. See Section 9.4. The steps to resolve the fault are:

- The Process finds the Region that covers the faulting address (or raises a bad-address fault in the dispatcher). It asks that Region to process the fault.
- The Region checks for permission violations (e.g. this is a write access to a read-only Region) and then asks the FCM to map the corresponding file offset to the virtual address with the correct permissions.
- The FCM determines if the requested file offset resides in a page frame. If so, or if a zero-filled page should be allocated, the FCM asks the HAT to update the address space mappings to map the faulting virtual address to that page frame. In this case, a page-fault complete indication is passed all the way back to the low-level code that called the Process object, and the faulting instruction is retried.
- If the FCM cannot provide a page frame immediately, it allocates a page frame and descriptor and calls its FR to schedule I/O to fill the page frame. In the normal case, the fault may be processed asynchronously. The FCM queues a notification request object on the page-frame descriptor and returns an indication that the page fault has not been resolved. The low-level code responds to this result by entering the faulting dispatcher at its page-fault entry point. The state of the dispatcher when the page fault occurred is passed back in this call. See the Scheduling white paper for more details.
- If the FCM discovers that the requested page has already been scheduled for I/O it proceeds as described in the previous paragraph, but without scheduling any new I/O.
- Sometimes, a page fault cannot be handled asynchronously. This is the case if the faulting dispatcher cannot accept a page-fault entry, because it is disabled or because it has exceeded

1. Dispatcher is a technical term in K42. It refers to an execution context scheduled by the kernel. Each Process has one or more dispatchers on each processor. See the Scheduling white paper for a complete discussion of dispatchers and page-fault processing on kernel threads.

the fixed limit of outstanding faults a dispatcher is allowed. In addition, kernel page faults are never handled asynchronously. In these cases the FCM blocks the (kernel) thread that is processing the fault until I/O is complete.

Kernel threads are a limited resource. Blocking for page fault I/O in these special cases cannot lead to exhausting the kernel threads. The blocked thread is either the reserved thread of the faulting dispatcher or the faulting kernel thread. The kernel design guarantees that a reserved thread is available for each dispatcher. When a kernel thread faults, the fault is always handled in the kernel without using other threads that could fault. Notice that in this case the faulting dispatcher or kernel thread is eventually resumed without ever being "told" about the fault.

- The FR that starts the I/O for its FCM normally sends an asynchronous IPC to its file server. This up-call is safe because the kernel does not block waiting for a response. Once the up-call is made, the kernel thread processing the fault normally terminates. When the I/O is complete the file system makes a PPC [2] to the kernel indicating IOComplete. That PPC is to the FR, which then calls the FCM. The FCM marks the corresponding page frame available and processes the queued notifications. Each notification either schedules an asynchronous "page-fault-complete" interrupt to a dispatcher or unblocks a blocked kernel thread. (The processing done by the PPC service thread in the kernel is an example of a "continuation").

3.1. Mapping Faults

The kernel can not tolerate kernel page faults except when the kernel is running enabled on a thread (See the Scheduling white paper for a discussion of enabled threads). And it does not tolerate a page fault while processing a kernel page fault. This is possible because all page frames accessed in those cases are pinned (wired) into memory. On some hardware architectures, it is impossible or impractical to prevent page faults on pinned pages. This usually occurs in architectures with software TLB miss handling or with inverted page tables.

To prevent these "mapping faults" from reaching the kernel, the low-level page-fault code checks for faults in well known ranges of addresses and handles them directly. The kernel maintains the tables used by this mechanism. All mapping fault addresses are put in an address range above all other kernel addresses, so the low-level code can distinguish this case with a single check.

Most such addresses are so called v-maps-r addresses. The whole physical address space of the machine is mapped into the kernel virtual address space at a fixed offset. These virtual addresses can thus be mapped to the corresponding page frames using that offset. By using these v-maps-r addresses to reference pinned memory, the kernel guarantees that it will not suffer real page faults.

---

2. Protected Procedure Call (PPC) and asynchronous IPC are two fundamental inter-process message services in K42. A separate white paper will describe them in detail. For our purposes, PPC has remote procedure call semantics, including implicit blocking until the target Process completes the operation and returns results. Asynchronous IPC causes a thread in the target Process to process the message, but the caller continues and receives no information about the progress of the operation. Asynchronous IPC is often used for up-calls, where the caller cannot risk any dependency on the response of the called Process.

In addition, certain tables, such as "exception local" (See the Scheduling white paper), used by interrupt code are placed in well known kernel virtual addresses and mapped by the low level code

4. Regions

Regions map contiguous ranges of an address space. Most Regions map to a file or computational FR/FCM. A Region usually maps its virtual range to a contiguous file offset range, but any mapping is allowed.

4.1. Creating Regions

A Region is created by calling a Region create method and passing the FR to be mapped and the Process whose address space the Region is to serve. The Process maintains a map of address ranges to Regions. Only one Region can map a given address. A Region can be created with a fixed address and size, in which case the address range must be available. Alternately, the request may ask the Process to provide an address for the Region.

Because Region creation involves access to both an FR and a Process, each request must be authenticated. The rule is that the create call must pass an FR object handle owned by the requesting Process, and a Process object handle owned by the requesting Process with attach rights. This prevents accidental or malicious manipulation of another Process's address space or of an arbitrary FR.

A Region is created with a single access type, for example read/write or read-only. There is no support for changing the protection of pages within a Region. Rather, protection operations like mprotect are implemented by remapping the address range with multiple Regions.

The mapping of virtual addresses to Regions in the Process must be efficient when many Regions exist, particularly because of the protection strategy of splitting Regions. It must allow searching for any address within a Region, rather than just the Region origin. And it must be able to support allocation of available address ranges for new Regions.

The actual protocol for attaching a Region to a file requires that the Region invoke the attachRegion method of the FR. The FR guarantees that an FCM exists, asks the FCM to record its attachment to the Region, and then returns the FCM reference to the Region. Thus, the protocol flow differs from the object reference connections seen in Figure 1.

4.2. Destroying Regions

A Region maps a range of a virtual address space onto a set of file offsets. The Region is connected directly to the FCM. Destroying a Region does not directly destroy the FCM, although once the Region is detached from the FCM that FCM may have no other users. In that case, the FCM informs its FR that it has no users, which may cause the FR to destroy the FCM.

Before a Region can be destroyed, it must be unmapped. There is a race between page fault requests that are in progress and Region destruction. We must not map a Region virtual address to a page frame after unmapping and detaching that Region from its FCM. We use a non-blocking request counter in the Region that can block new mapping requests and wait until all existing requests complete. We are considering replacing this mechanism with one based on the thread generation-count mechanism (See the Clustered Object white paper). The advantage of

this change would be reduced page fault path length, at the cost of increased Region destruction latency.

The hardware specific part of the paging system can be optimized for Region unmapping. In most memory management architectures, it is possible to unmap a hardware segment without processing every page address in the hardware segment. On machines with tree-structured page tables, this is done by disconnecting a lower level page table. On machines with inverted page tables this is done by removing a segment ID from the segment mapping and not reusing that ID until it is certain all page table entries have been overwritten by normal processing.

Regions do not always conform exactly to hardware segment boundaries. There is no harm in unmapping the segments that cover a Region because other mappings will be recovered as needed. This optimization does depend on having sensibly sized segments, so that not too many other mappings are thrown away along with those to be cleared.

5. Page Replacement

A paging system must write modified page frames to backing store, unmap page frames from their current use, and make them available to back new requests. The choice of which page frames to free, or "replace", is usually based on a combination of two strategies. The first is Least Recently Used (LRU). Under LRU page frames not used for a long time are selected for replacement. The second strategy, working set, is based on the idea that a specific user of page frames (e.g. a process or a file) should be given a quota and manage replacement internally. This internal management is often done using local LRU.

K42 page replacement does not use global LRU or the "clock" algorithm used to implement it. We avoid global LRU because the global data structures and locks needed to implement it are points of contention hindering multiprocessor scaling. Rather, we implement a hierarchical working set mechanism.

The page manager, or PM object, implements hierarchical working set management. Each PM manages a set of child PM's and a set of child FCM's. The top of the tree is a root PM. For example, a PM for each Process manages all the computational FCM's for that Process.

A PM maintains estimates of the number of page frames, and the number of modified or "dirty" page frames in each child. Children make page-frame allocate and deallocate requests to their parent PM's. This allows a distribution of all the page-frame allocation mechanisms by moving them down in this tree.

Page replacement is done by walking the PM tree, asking FCM's to free page frames. Small FCM's are asked to free all of their page frames. Larger FCM's are asked to free a fraction of their page frames. The assumption is that large FCM's will maintain LRU information and estimates on the value of page frames, so that the system can balance the page frames allocated to each FCM to optimize performance. Currently however, a simple random approach is used.

PM's will also be responsible for implementing policies such as limiting the resources consumed by a process or user, and for allocation page frames in a way that supports good NUMA behavior.

5.1. Guaranteed Forward Progress

An important property of a paging system is that it does not reach a state where no frames can be freed to satisfy a page fault. This could happen, for example, if the memory manager data structures needed to find a frame to be replaced were all paged out, and no free frames were available to page them in.

K42 guarantees forward progress by segregating storage into kernel pinned memory, kernel paged memory, and the rest. Kernel pinned memory contains all programs and data necessary to do paging I/O, and contains the memory manager objects which manage the kernel paged memory. The kernel paged memory is backed by frames managed by a special kernel computational FCM. This FCM uses pinned memory to represent all of its state. It does all page replacement internally by reusing a pool of frames it has been given. This kernel computational FCM can always make forward progress as long as it has even a single frame in its pool.

Of course, the Memory Manager must adjust the actual number of frames held by the kernel computational FCM. But this is always done be giving frames to that FCM or taking them from it. The kernel computational FCM never calls the page frame allocator, and thus can never be blocked while that allocator attempts to find free frames.

6. File System Interactions

K42 implements file systems in file server processes outside of the kernel. But file I/O is done using the memory management system in the kernel. This leads to a uniform buffer cache, because both file data and computation data are cached in page frames managed by the kernel. It provides directly for memory mapped access to files (e.g. mmap). The FR is the file system's representative of a file in the kernel.

6.1. Authentication of FR access

The file system controls client access to files. But file system clients need controlled access to the FR in order to memory-map files, and possibly for other operations. The protocol to provide and control this access is:

- The FR methods intended for file system use are declared, in the IPC interface definition, to require "filesystem" access rights. (See the interprocess communications white paper for a discussion of authentication).
- The file system makes a kernel request to create an FR for a file. The newly created FR creates an object handle that carries "filesystem" access rights for the file-system Process.
- When a client opens a file by calling the file system, the file system asks that file's FR to create an object handle for use by that client. This object handle does not have "filesystem" rights and may be further restricted to read-only use. The file system returns that object handle to the client. The client can use that object handle for direct access to the FR. (Note that in K42, it is not "possession" of the object handle that confers rights. Rather, the object handle was created to confer specific rights to a specific Process).

Once a client has an FR object handle, that handle can be used to create Regions that map parts of the "file" into the client's address space. See Section 4.1 for details on creating Regions.

6.2. Lazy Close and File Destruction

A file system FR/FCM manages the page-frame cache of a file system file. We do not attempt to free this cache as soon as the file is closed (has no active users). Rather, we wait until the page replacement mechanism causes all its page frames to be freed. This "lazy close" optimizes the rapid sequential reuse of files.

The file system manages FR and thus FCM destruction. The destruction lock hierarchy is file system->FR->FCM. We us protocols in which lower elements (e.g. FCM) notify higher elements (e.g. FR) whenever a transition to a not-in-use state occurs. This notification is made with no locks held, and thus depends on the K42 deferred object deletion strategy for correctness.

Once the file system has been notified that the FR is not-in-use, it can choose to ask the FR to destroy itself. This call is conditional, since the FR may have come back into use in the interim. The call either succeeds, or returns an indication that the FR is in use. In the later case, the file system will get another not-in-use up-call when the FR is again not-in-use.

We thus count on file systems to manage FR resources in the kernel. If a file system misbehaves, the ultimate solution is to terminate the file system server. This will trigger the destruction of all the FR's and FCM's created by that file system.

When a file is actually removed, meaning that it has no links in its file system and is not open in any Process, the file system tells the FR to be more aggressive about signaling not-in-use. Specifically, the FR no longer waits for all frames held by the FCM to be released by normal paging. Instead, as soon as there are no processes using the FR/FCM, the file system is notified and destroys them.

6.3. Scheduling Page-Frame I/O

There are two important cases of page-frame I/O. Some file systems, such as local disk file systems, schedule I/O to or from the actual page frames. However, file systems like NFS need to process data in a virtual page in the file system. This data must be transferred to or from the real page frame. Although one can imagine techniques for manipulating virtual mappings to avoid copying data in this case, it turns out to be difficult to do this with performance better than that of the copy.

In K42, we deal with the need for two or more strategies by customizing the FR. Since the file system chooses the specific implementation of its FR's, the choice is up to the file system. In fact, other strategies could be tried with no effect on any existing implementations by providing a new FR customization for a new file system.

We expect to short circuit file system physical I/O to avoid multiple passes through the file system by providing a "smarter" FR for file systems that can support it.

The paging I/O system must control I/O pacing. I/O requests originate in the kernel. But the kernel cannot use an interface that makes direct I/O requests on the file system, since the file system cannot always satisfy, or even accept, an I/O request from the kernel. A protocol that can defer I/O requests without blocking kernel threads or allocating additional kernel resources is needed to guarantee forward progress.

The pacing strategy is to inform the file system that I/O requests are available, but have the file system make calls on the kernel to fetch additional requests. The is done separately for each FR.

Requests are queued in the kernel in the FCM by making a list of page descriptors needing I/O. Finally, a list of FR's that need I/O and have not been able to communicate with the file system is kept for each file system.

The pacing mechanism also allows an FCM to batch I/O requests, which can improve I/O efficiency.

7. Computational Storage

Computational FR/FCM's implement the storage that backs computation, such as data, stacks, and heaps. It is paged to paging space, but never survives a restart of the system.

7.1. Computational Segments

The FR/FCM pairs that implement computational storage are specialized. Naively, one could try to back each computational Region with a temporary file. The difficulties with the naive approach are:

- Another paging I/O mechanism would then be needed to page both the kernel and the file system that provided the paging files.
- Efficient implementation of fork-copy requires the efficient passing of the disk blocks backing paged-out computational page frames from one FCM to another. Normal file systems do not provide such a mechanism.
- In computational storage, all page frames start as "zero-fill" page frames. Unless the file meta data were reproduced in the kernel, a file system up-call would we needed on each initial page fault to discover that the page was not backed on disk and should be zero-filled.

Thus, computational FR's use a separate swap mechanism to do paging I/O. These FR's maintain sufficient meta data to determine which pages without page frames are in fact backed on disk. Others are zero-filled immediately. The swap I/O system will be implemented completely in the kernel and device sub-system, using code and data that is not paged.

Computational FR/FCM's are destroyed aggressively because they cannot be reused.

7.2. Fork-Copy

Fork-copy is the mechanism that implements the Unix fork semantics as it applies to computational memory. At the instant of fork, it must appear as if a copy of the computational memory had been made, with one copy given to the parent and the other to the child. K42 follows the standard practice of using lazy copy to implement this efficiently. It builds a tree of two empty child FCM's sharing a parent that is given all the page frames and disk blocks from the original. All the page frames are unmapped. Parent FCM's never have Regions attached to them. They only provide page frames or copies of page frames to their children.

Before doing a fork-copy, the request count mechanism is used to stop the Region attached to the FCM and to unmap it completely. For this reason, fork-copy is in fact a method on the Region that maps the FCM, rather than on the FR. Fork-copy computational FCM's allow only a single Region to be attached when a fork-copy request is made. With this restriction, the implementation is significantly simplified while allowing Unix fork to be implemented.

In response to a page fault, the fork-copy mechanism searches the fork-tree for an existing page frame and/or disk block and copies or gives it to the fork-child. As parent nodes in the tree become empty of page frames and disk blocks, they must be removed to avoid unbounded growth of the fork-tree. We call this process collapsing the fork-tree. The details of fork-copy and collapse are complex and arcane and are discussed in the detailed K42 design documentation.

8. Copy-On-Write

Copy-on-write is a common optimization in memory management systems. It is used in the implementation of fork-copy, debugging support for program text, and in support of mmap operations with copy semantics. In all these cases, the goal is to share a page frame until an attempt to modify it is made.

Since a copy-on-write copy is, semantically, a copy, a separate FR/FCM pair is needed to represent it. This FR and FCM are specialized implementations, called the FRCRW and the FCMCRW. The FRCRW is created by providing a reference to the FR that is to be "copied". The FRCRW in fact attaches to the base FR using an adapter Region in the FRCRW. Once attached, the FCMCRW can copy pages from the base FCM or ask the base FCM to map page frames for the Regions attached to the FCMCRW.

Currently, K42 only implements copy-on-reference, simplifying the interaction between the copy and base FCM's.

9. Portability and Hardware Exploitation

9.1. Page Tables

In the description of the overall structure of Memory Management (Section 2) we say that K42 separates the logical and hardware implementations of Memory Management. The fact that hardware page tables are managed as a cache allows K42 to adapt to the three major MMU designs: hierarchical page tables, inverted page tables, and software TLB miss handling.

Systems designed to exploit a hierarchical page table architecture (e.g. X86) often merge logical and physical information. They use the page table entry as the page-frame descriptor for mapped addresses, and also to hold the disk block numbers of pages that have been swapped to disk. This approach can provide space and time advantages, but it leads to compromises. And implementations on other architectures wind up using the original hardware tables as a logical structure, continuing the compromises into these other architectures.

9.2. Change and Reference Bits

Many MMU's have hardware support for tracking page references and modifications. Reference information is useful in estimating LRU, and change information is used to determine when page frames are "dirty" and need to be written before being reassigned.

The alternative to this hardware is a logical approach that uses first faults to track references, and uses write protection to detect modifications. The cost of this technique is extra in-core page faults. The advantage is that reference and change information is known earlier. Of course, this approach is intrinsically portable.

K42 currently uses the logical approach. Exploitation of hardware will probably be needed, but has been deferred. Such exploitation is more difficult in K42 because of the separation of logical and physical implementation, and because tracking reference and change information can lead to global data structures and locks, both of which are being avoided.

9.3. Shared Segments

A useful optimization for many MMU architectures is sharing the hardware mappings of a segment across address spaces. In hardware terms, this is done by using the same page-table subtree in several high-level page tables (hierarchical page tables) or using the same virtual segment id in several address spaces (inverted page tables). In inverted page-table systems like PowerPC, even the TLB entries are then shared across address spaces.

K42 includes mechanisms in the logical memory management system to make this sharing possible. The essential idea is to have some FCM's manage shared segment HAT's that represent a mapping of the FCM's content. Protocols between the Region, FCM, and HAT allow this shared segment to be installed into a HAT when a Region is attached to the FCM. These protocols are designed so that the Region has a chance to choose a virtual address range for the Region that is consistent with the hardware requirements for shared mappings. This is always an optimization, and the Region falls back to a private mapping if the shared mapping is not appropriate. The virtual address allocation mechanism mentioned in Section 4.1, which allows the kernel to suggest the origin address of a Region, was chosen to make this possible.

The most common uses for shared segments will be the text of shared code libraries and mapped files. One important advantage of a 64-bit address space is that files can be freely mapped into the virtual address space without exhausting the address range. Sharing the mapping, and maintaining the mapping even when the file is not open (see lazy close in Section 6.2) make this particularly attractive.

9.4. Locking

Memory management follows the K42 locking strategy. Each object instance has its own lock. Locks are (mostly) not held when one object calls another. Instead, the general clustered object mechanisms guarantee the safety of these calls. For example:

- The Process holds a lock while looking up a faulting address to find the Region to service it. But the Process releases that lock before handing the fault to the Region. If the Region "disappears" the call will return a deleted-object error, which will be converted to a segment violation trap to the faulting Process.
- The FCM holds a lock while finding the page descriptor for the needed page frame. Once it finds or creates a descriptor, it locks the descriptor. It can then release its lock (not completely implemented yet). Specifically, the FCM does not hold its lock when it requests allocation of a new page frame to satisfy a fault. This avoids deadlock if the allocator needs to ask FCM's to free page frames.

9.5. Multiprocessor Support

K42 memory management is designed to work well on multiprocessors. The design avoids global data structures and locks, and uses fine grained, short duration locks. In addition, research is in progress on distributed implementations and the use of Clustered Object technology to support those implementations.

The fundamental technical problem of distributed memory management is that certain operations should be done locally on each processor involved. The prime example is unmapping. Although MMU hardware sometimes offers support for global page tables and TLB manipulation, these mechanisms do not scale well. K42 uses local tables on each processor, and tracks Region and segment residency on each processor. When a Region or segment needs to be unmapped, only the processors that have mapped it are involved. Cross processor messages and barrier synchronization are used to do the actual operations locally on each processor.

The goal of distributed implementations is to further reduce lock contention and storage interference. This is done by building per-processor caches of information such as page descriptors so that many faults can be processed without any cross processor operations.

The implementation also uses K42 storage allocation features to avoid false sharing across processors by making sure that independent data is stored in independent cache lines.

This area is one of the fundamental research subjects of K42, and we expect more results to follow as we enter our full-scale experimental phase.

9.5.1. Processor-Specific Memory

Processor-Specific Memory is a form of a Region that maps the same virtual address to a different page frame on each processor. Processor-Specific Memory is used in the implementation of the clustered object mechanism (See the Clustered Object white paper) and in other ways. The logical implementation is simple. A specialized Region includes the processor number in its conversion of virtual addresses to file offsets, thus placing the same virtual address on different file offsets for each processor. It is easy to make a fork-copy of a Processor-Specific Region, since the FR and FCM are not specialized, only the Region is different.

For this to work, however, it must be the case that asking the HAT to map the same virtual address to different page frames on different processors will "work". Thus, hardware paging must uses different mappings of an address space for each processor. On machines with hierarchical page tables, this is implemented by using a different top-level page table for each processor. On machines with inverted page tables, this can be accomplished either by using different segment descriptors for each processor, or different inverted page tables for each processor. We choose the later to prevent lock contention on page table manipulations.

The fact that K42 uses different mappings for each processor allows the kernel to track the processor residency of page frames, Regions, and FCM's, since a fault must occur on each processor that uses these elements. Thus, many hardware manipulations related to unmapping page frames can be done only on the relevant processors. This is necessary if small programs are to work well on large systems.

9.6. Concluding Remarks

Memory management is a rich subject for research, particularly given the goals and technologies of K42. This paper indicates only the broadest outlines of the problems and their solutions. We look forward to presenting the results of our research as they evolve.