Calypso: A Portable Translation Layer

Cristan Szmajda
cls@cse.unsw.edu.au

9 September 2001

Calypso is a virtual memory subsystem for L4, featuring fast software page table lookup, arbitrary page size mixtures, shared page table subtrees, and domain-based MMU protection.

While Calypso currently only runs in L4/MIPS, it should be reasonably portable to other machines and L4 implementations. Even L4 implementations on machines with hardware page table formats could benefit from a port of Calypso if only to provide an efficient and flexible mapping database implementation.

This paper outlines the API extensions made by Calypso to support newer hardware features, and documents some of its internals and implementation techniques.

Motivation

The main advantage of micro-kernel based systems is that system components can be protected from each other with hardware memory protection.

However, if the cost of address spaces is too high, system designers will be forced either to accept a performance penalty over monolithic systems, or to avoid memory protection altogether. Thus Mach and Windows NT have migrated many components back in to the kernel, losing all the flexibility and reliability advantages of a component-based design. Even Linux has demonstrated performance improvements in benchmarks by migrating services such as NFS and even Apache back into the kernel, lending memory protection a reputation as being expensive and unnecessary.

L4 has addressed the problem of poor IPC performance by providing highly optimized IPC. However, the cost switching address spaces remains high: each TLB miss costs over 30 cycles on modern microprocessors, and ever deeper pipelining increases this penalty. Each cross-address space IPC incurs many of these misses. Furthermore, services decomposed into multiple address spaces touch many more pages, and require corresponsingly higher TLB coverage, than the equivalent services in a monolithic kernel.

Calypso attempts to address this problem by optimizing page table lookup on machines with software TLB handling, allowing mixtures of large and small page sizes to increase TLB coverage and reduce misses, and taking advantage of domain-tagged TLBs to share TLB entries between address spaces.

Domains

The ARM *domain*, HP Precision Architecture *region ID*, and IA-64 *protection key* are generalized tags which allow TLB entries to be shared between address spaces, possibly with different protection attributes.

Domains on the StrongARM can be used in fast address space switching (Wiggins 1999). However, this is not the only (or even the primary) use of domains. Domains are intended for shared libraries. Sharing library TLB entries between address spaces has the potential to reduce TLB misses on cross-address space IPC.

Using domains for shared libraries can also be combined with the use of domains for small address space switching. Use of shared libraries can also help small address spaces from exceeding 32M on the StrongARM.

Domains are also ideal for single address space systems (SASOS's) such as Mungi. In order to achieve the simplifications and performance improvements possible in a SASOS, kernel support for domains is essential. Otherwise, Mungi on L4 will be forced to emulate a SASOS expensively using many separate address spaces.

Page Sizes

Many microprocessor TLBs support multiple page sizes. Making use of this capability can reduce the frequency of both compulsory and capacity TLB misses, which can be a significant overhead if address space switches are frequent or if the hardware TLB has low capacity.

machine	ITLB	DTLB	page sizes
Intel StrongARM	32	32	4k, 64k, 1M
Intel Pentium III	32	64	4k, 4M
Intel Itanium	64	96	4k, 8k, 16k, 64k, 256k, 1M, 4M, 16M, 64M, 256M, 4G
Alpha 21264	128	128	8k, 64, 512k, 4M
UltraSPARC	64		8k, 64k, 512k, 4M
MIPS R4000	96		4k, 16k, 64k, 256k, 1M, 4M, 16M
PowerPC 601	256		4k

Table 1: TLB and page sizes of some CPUs

Most operating systems only use larger page sizes for kernel virtual memory, or special-purpose mappings such as frame buffers. Calypso allows any mixture of page sizes to be used, provided the user specifies large fpage sizes. This gives the user freedom to mix page sizes in any desired policy.

Portablility

Another potential advantage of micro-kernel based systems is portability. As far as possible, Calypso exports a uniform interface to the user regardless of the capabilities of the underlying hardware. Use of super-pages is completely transparent; users can map or unmap fpages of any size and in any combination. (But for performance reasons, a minimum page size is configurable at compile-time.) Use of TLB domain tags is enabled by an API which can also be implemented on hardware with no domains, and in fact can be implemented efficiently by sharing branches of a multi-level page table.

Concurrency

Another challenge is concurrency. Protecting the VM subsystem with a single global lock (as in Linux) would be a significant barrier to multiprocessor scalability. This coarse-grained locking would also block high-priority threads from accessing the VM subsystem while a low-priority thread is in the middle of updating it. The latency of many L4 VM system calls is unbounded.

Calypso addresses this by protecting page table entries and memory manager data structures with individual spin-locks, and by preventing a thread from being pre-empted while holding on to a lock. The 'timeslice donation' solution to the priority inversion problem is unnecessary, because the length of time to wait for any lock should be bounded above by a limit in the microsecond range.

Furthermore, with locks on each PTE and separate arenas for each address space, no locks should be encountered unless two threads are updating mappings in the same address space at a time. High-priority threads in a separate address space or threads in address spaces on different CPUs should suffer no penalty from fine-grained locking, as the locks should never be contested.

1. An L4 API for domains

Why the current API is inadequate

In L4, a page cannot be mapped into multiple address spaces simultaneously. Therefore, L4 cannot tag a shared library page with a domain until it is mapped, one by one, into all members of that domain. If the pager must also wait for a page fault to be received from all such members, the map operations do not even occur together in time, because map is bound to IPC.

Also, L4 has to keep track of the set of address spaces sharing each page to assign domains to each such set. This information is present in the mapping database, but the amount of bookkeeping required to assign domains automatically from such sets is considerable. Transparent assignment of domains by the kernel is even less attractive when it is realized that the ARM has only 16 of them — the decision of how to assign domains should be up to the user.

Exactly the same problem occurs on architectures which do not have domains but do wish to take advantage of sharing by sharing page tables. Sharing page tables mostly saves space but can also save time when paging in and out shared library pages (a situation in which Linux performs a quadratic brute-force search of all page tables (Dillon 2000)).

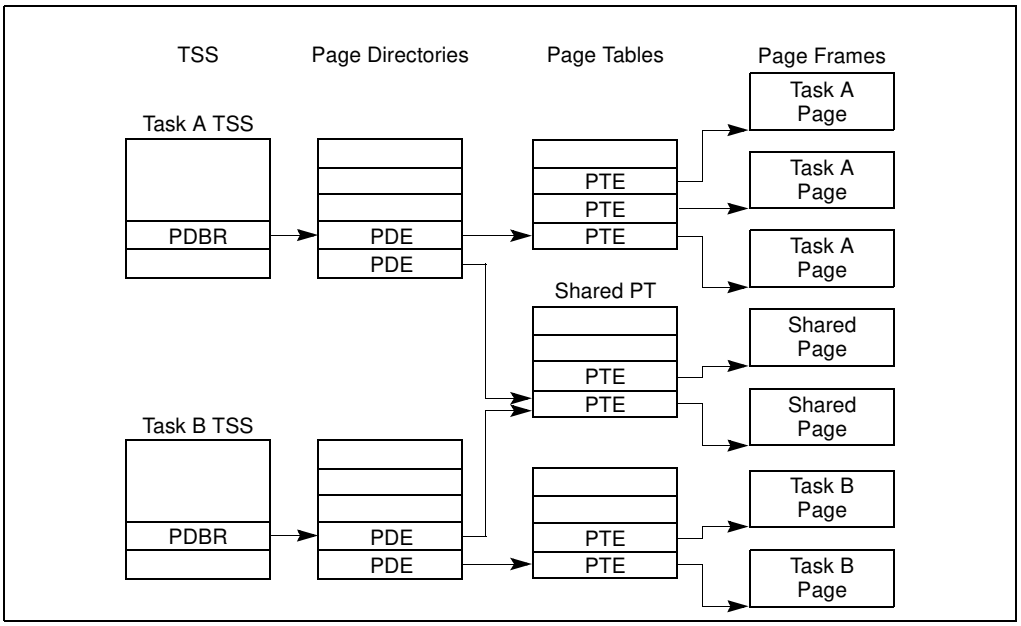

Figure 1: Sharing page tables on the Intel 80386 (Intel 1999)

These problems would be neatly solved if there was a primitive which could map a page into all members of a domain at once rather than into one address space at a time.

Link

It would seem that to take advantage of domains, primitives would need to be added to L4 for allocating domains, adding and removing domain membership, and mapping and unmapping pages in a domain. However, this would be a rather large change to the L4 API, and would require significant code changes to L4 applications. Furthermore, since domains are not available on all machines, portable L4 applications would have to refrain from using such primitives if not present, and include code for both APIs. Finally, unlike map which is bound to IPC and protected by the usual L4 confinement mechanisms, new primitives would require a new privilege model to ensure that domains could not be abused by untrustworthy applications.

Therefore, instead of inventing new primitives, Calypso extends the existing model with (yet anoth-

er) type of fpage, a **link** fpage. Link is like map except that instead of just copying a snapshot of the fpage, future updates to the mappings in the pager's address space are also reflected in the destination address space. This relationship may be cancelled with an unmap.

L4 primitive	Unix analogy
unmap	`rm`
map	`cp`
grant	`mv`
link	`ln -s`

Table 2: Unix analogy explaining link

The first time an fpage is linked, it is allocated a domain. Calypso even maintains domain identifiers for linkages on hardware with no domain support, to remember shared page table references. A domain table internally keeps track of address spaces which are members of each domain and its virtual address range.

Link may specify different protection bits in different address spaces. For machines without domains, this is implemented by storing a protection mask in a linked internal nodes. If the fpage is completely unmapped in all address spaces, the domain is deallocated.

Link also allows pages to be mapped asynchronously (that is, without IPC), after the fpage is linked. Thereafter, pages mapped into the pager's address space also appear in the destination address space without further IPC. This provides the benefit of asynchronous map without bypassing L4's IPC confinement.

Limitations

Calypso only links pages with identical virtual addresses in all address spaces. This restriction is a requirement for hardware domains, and simplifies the data structures for software page table format substantially. The check that enforces this restriction also prevents normal mappings that would cause inconsistency in virtually-indexed caches such as those on the MIPS R4000 and UltraSPARC.

A linked fpage may only be modified by the pager; attempts by other threads to accept mappings in a linked fpage fail. Similarly linked pages are only mappable to other address spaces by the pager. This restriction simplifies the mapping database substantially.

Domains are allocated first-come-first-served, and so may be abused by untrustworthy applications. While this problem also occurs with traditional ASID recycling, the small number (16) of domains on the ARM gives the problem greater urgency.

An alternative approach would be to generalize address space identifiers to include domains, and allow a thread to be a member of multiple domains. However, the link primitive described here fits in well with the existing IPC and address space model, and so requires very little change to application code and to the rest of the kernel.

2. Globally tagged pages

Not all machines support domains in hardware. Pentium and PowerPC do not even support ASIDs, and require the TLB to be invalidated on every address space switch.

However, almost all machines support a global bit in the TLB entry indicating that it should not be invalidated on context switch (Pentium), or that the entry is always valid regardless of the current ASID (Alpha, MIPS, SPARC). Machines with hardware domains can easily emulate a global bit by reserving domain 0 as a global domain in which every address space is a member.

In most systems, the global bit is only used to tag kernel pages. However, this feature could also be usefully applied to widely shared pages such as Unix `libc` or Mungi active protection domain 0.

In order to analyse the performance impact of using globally tagged pages, I propose to add an L4 interface to allow applications to use this feature. Address space 0 is reserved as the global address space. Pages in address space 0 are visible in all address spaces. A kernel thread in address space 0 accepts mappings in a window of its address space. The whole of address space 0 is not available for global mappings, because it is implicitly linked in all page tables, and because the potential for overlap between private and global mappings is too risky.

Permission to create global mappings is protected by the usual L4 IPC confinement mechanisms, and is initially available only to the root servers.

Again, a cleaner interface to this feature could be provided if L4 supported domains directly.

3. Kernel Memory Protocol

At boot time, σ_0 reserves a fraction of memory for the kernel to allocate TCBs, page tables, mapping database nodes, and other data structures. The rest of memory is available to initial servers on a first-come-first-served basis.

This fixed partition wastes memory if the reserved fraction is too high, or causes system calls to return errors (or worse) if the fraction is too low.

Some L4 implementations define a σ_1 task and a protocol allowing initial servers to provide the kernel with memory when it starts to run short. This solves the problem of fixed partition, but still allows malicious applications to consume unlimited kernel memory by creating many threads and mappings.

Calypso uses a similar protocol which both solves both problems. A separate free list is maintained for each address space. Whenever the kernel needs to allocate memory and this free list is exhausted, it suspends the thread and generates a page fault IPC to its pager. This IPC is exactly like the normal page fault IPC, except that the EPC (or faulting instruction pointer) is a magic kernel address not possible for a normal page fault. The pager then grants a page or pages which are added by the kernel to the thread's free list, and it is restarted. The pager must be the sole owner of any granted pages. (Calypso already enforces this restriction on grant, to avoid problems with directed unmap.)

Initial servers are provided with kernel memory by σ_0, which reserves a small amount of memory for this purpose.

No protocol is currently defined for getting pages back out of the kernel. It is not yet clear how to do this without the kernel taking over some of the function of σ_0, or putting σ_0 (back) into the kernel.

4. Internals

Page Table

Most 32-bit page tables do not perform well in 64-bit address spaces. For example, the common multi-level page table for 64-bit addresses would either require megabyte tables, or over five levels.

The guarded page table (Liedtke 1994) is a generalization of multi-level tables which allow redundant levels to be bypassed in the common case where the enormous 2^{64}-byte address space is populated only in a few sparse regions. However, the data structure is somewhat complex to search and maintain, and has only been implemented with a fixed level size that fails to adapt to sparse and dense regions of address space. In practice, the guarded page table is not fast enough to be used directly for TLB refill, and is usually accelerated by a software TLB.

The level- and path-compressed trie (Andersson & Nilsson 1993) is similar to the guarded page table, but is simpler and more general. Levels can be any power-of-two size, controlled by shift amounts in the internal node. No guards are explicitly stored in the nodes, and hit detection is deferred until the leaves.

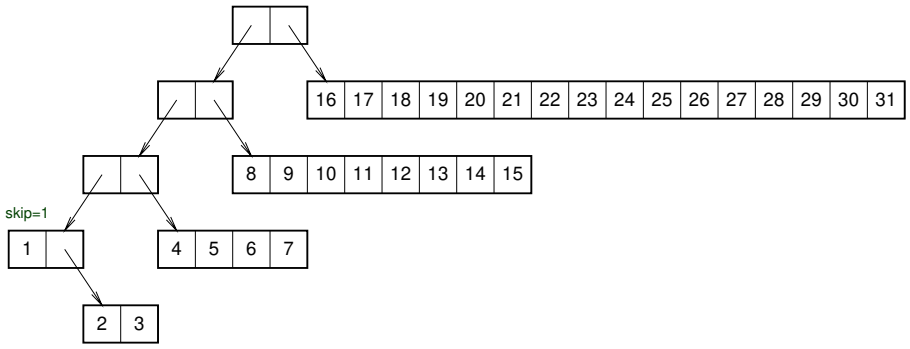

Figure 2: a level- and path-compressed trie

The size of each level can be any power of two, and is a tradeoff. Smaller levels save memory but larger levels provide faster lookup. In the extreme case of σ_0 and the initial servers, with address spaces one-to-one with physical memory, a single level page table can be allocated, resulting in a simple array index for page table lookup. This approaches the performance of translation using a software TLB.

For more complex address spaces, one or more levels of internal nodes are inserted. The lookup algorithm only tests necessary bits of the address. All other bits are skipped without checking, and are only verified at the PTE where a single comparison instruction suffices to check for page fault. This is similar to checking for a hit after looking up a bucket in a hash table, except that collisions do not occur. In fact, the trie can be considered as a form of collision-free hashing.

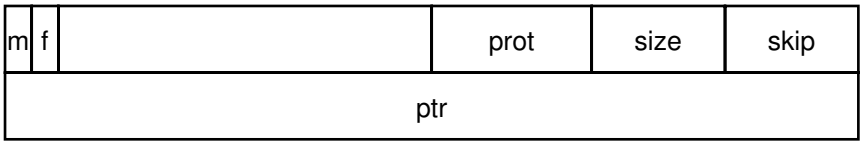

Figure 3: internal branching node

The allocation strategy is left to a memory manager, which attempts to allocate levels as large as possible given the available memory. If large levels remain unused for a long time, the memory manager reserves the right to reduce their size, reclaiming memory for other purposes. When separate arenas for each address space are used, this exports the policy of resolving the time–space tradeoff to the user. A system with plentiful memory can grant many pages for each address space's kernel memory. One with a tighter memory budget can allocate fewer pages to some or all address spaces, reducing performance of certain tasks to save memory.

The trie data structure also incorporates mixed page sizes naturally, unlike hash tables or software TLBs which require either multiple hash tables or duplicated entries.

Calypso prefers only a small number of page sizes to be present in a trie at a time. Too many different sizes would cause the trie levels to be fragmented and be expensive in lookup and memory management time. Therefore, page sizes will only be used if they can be accommodated in trie levels of a reasonable size (as allocated by the memory manager). In practice, this limits most address spaces to two or three different page sizes. (If this simple solution proves insufficient, the next possibility would be to replicate PTEs across levels.)

Large pages are especially beneficial for σ_0 and the initial servers, whose address spaces are one-to-one with physical memory. These address spaces often require only one PTE (and therefore only one TLB entry), vastly reducing address space switching costs.

Mapping Database

The VM data structures of most L4 implementations consist of a page table (either hardware-defined or GPT) per address space, a global frame table, and a mapping database with one entry for each PTE. The mapping database logs every map operation so that it can be undone with the unmap primitive.

The mapping database nodes are often implemented with pointers to lists of children and sibling nodes, and a back-pointer to the parent. Because one of these is allocated for each PTE, the mapping database is usually the single biggest consumer of kernel memory. Because the frame table and mapping database are shared data structures, they must also be protected by locks in a pre-emptable or multiprocessor kernel. Finally, the mapping database implementation becomes more complicated if the nodes can represent different page sizes. Recall that the Itanium supports a page size of 4G. Incorporating support for this feature is clearly desirable but requires careful implementation if the mapping database is not to consume large blocks of memory or require expensive linear searches.

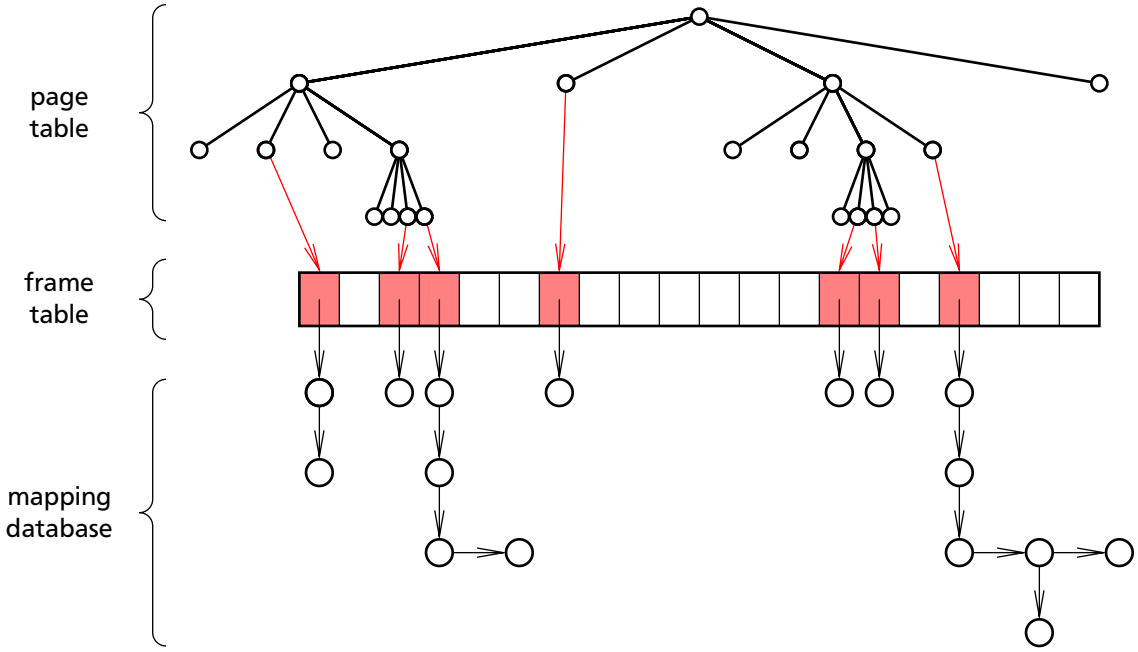

Figure 3: traditional L4 data structures

The first simplification to the mapping database is to represent the complex linked structure as a singly-linked list, by topologically sorting the mapping graph. The depth in the mapping graph is recorded as a small integer in each node: the *generation* number.

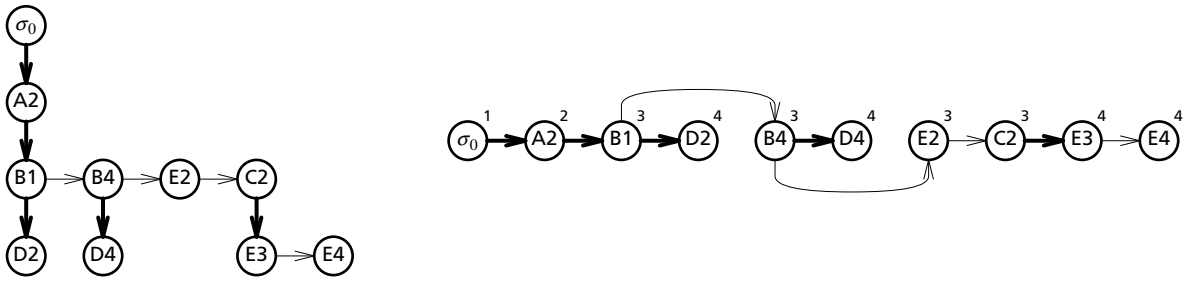

Figure 4: example mapping graph and topological sort order

In the software page table format used in Calypso, much of the information in the PTE and mapping database nodes was duplicated. Calypso therefore took the next simplifying step and merges the PTE and mapping database node into a single structure. This vastly accelerates unmapping, which is a simple linked list traversal. Merging the information in this way also eliminates the need for a frame table.

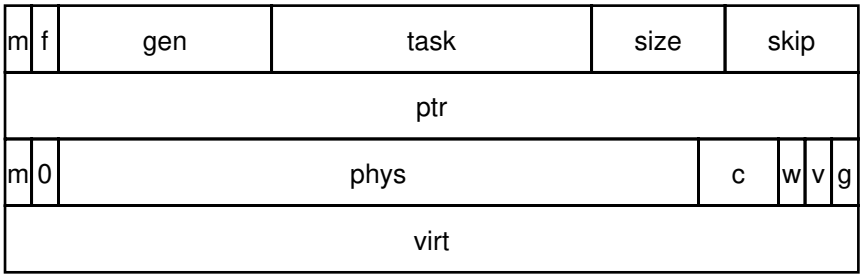

Figure 5: `struct pte`: combined page table entry and mapping database node

Each quadword is tagged with a magic bit, distinguishing internal and external nodes. This is required for the fast lookup loop, and is also assumed by the memory manager. The memory manager also requires an additional 'fencepost' bit to distinguish the head of a kernel memory block. Recall that the user can grant fpages to the kernel anywhere in the physical address space.

Updates to each PTE (map, unmap, permission changes, etc.) are protected by locking loads and stores. On RISC machines which support locking via *load locked* and *store conditional* instructions, such an update can fail due to contention. In this case, the update simply backs out and tries again. Lengthy spinning in the kernel (possibly with high priority) is not expected, because updates are short and pre-emptable.

The simple linked list representation gets complicated when page sizes are mixed. One PTE, which may represent a page of many megabytes in size, can be mapped piecewise thousands of times. If these child mappings were inserted into the list in random order, updates would require expensive linear searches. The desire to support sets of page sizes as large as the Itanium (with eleven sizes up to 4G) also ruled out the use of a simple multi-level array to cope with this problem. Instead a level and path compressed trie is used, almost identical to the page table data structure. The same routines which manipulate page tables are also used, with some modification, in the mapping database.

The internal page table node and mapping database internal node use a similar structure.

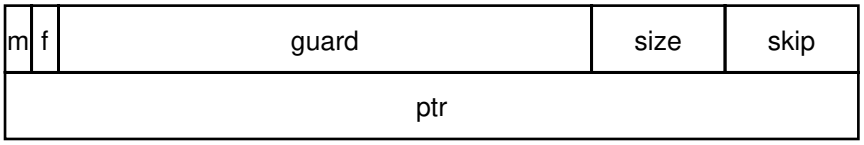

Figure 6: `struct dir`: internal page table and mapping database node

In the common case of mapping pages all of the same size, `ptr` from the PTE points directly to the next PTE, as before. (This is a singleton trie.) Otherwise, it points to an array of `struct dir`s, which is allocated by the same memory manager as the page table. A guard must be included because the hit comparison cannot be deferred until the leaves in the mapping database — the virtual address of a PTE gives no information about the virtual address it was mapped from in the pager.

This allows even very large pages to be mapped sparsely or in any page size combination without using too much memory.

Updates to each `struct dir`, whether in the page table or mapping database, are *not* protected by locks. Rather, if the array referenced by `ptr` is resized, or a new level inserted, it is first allocated and initialized, and the entire `struct dir` is written atomically with the new information. Provided

memory barriers are added between these operations for those machines that require them, this should keep trees consistent at all times, without the need for the fast page table lookup path to inspect locks.

Memory Manager

Calypso's memory manager is tailored to the demands of the page table. The buddy system is used to manage power-of-two sized regions and ensure that the largest possible free blocks are available.

Memory is allocated in two distinct phases. First, a block is allocated with `buddy_borrow`. The VM subsystem severely overestimates the amount of memory it needs to borrow, expecting the memory manager to return a smaller size. The memory manager applies a worst-fit algorithm, returning the largest block it has available, but typically much less than what was asked for.

At any time, the memory manager may call back some or all of the borrowed memory, unless the VM subsystem has called `buddy_lock` to declare that the memory is in use. `buddy_unlock` reverses this, making the memory available to the memory manager again. `buddy_lock` and `buddy_unlock` immediately split and join adjacent buddy blocks, performing at most $\lg(N)$ iterations, where N is the size of the arena in 16-byte units. However, when amortized over all units in the arena, the average number of iterations is $\lg(\lg(n))$, where n is the average page table size. The arena must be locked for the duration of these routines, preventing any other update of the same address space.

The buddy blocks themselves are simple doubly linked lists.

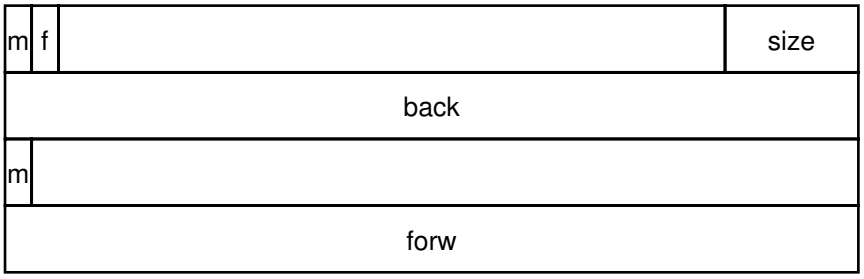

Figure 7: free block header

The page tables aggressively allocate as much memory as possible using worst-fit to build levels as large as possible when constructing new page table branches. Other memory (for TCBs etc.) is allocated using `buddy_alloc_best`, which allocates power-of-two sized memory regions from the same free list using a best-fit policy.

5. Status

Calypso is currently tailored for best performance in L4/MIPS. Many data structures are optimized for the MIPS PTE format and even MIPS instructions. Also, it relies on several L4/MIPS data structures, especially the TCB. Work is proceeding to remove this dependence, both for portability and because when debugging many L4 data structures are a crowded and dangerous place for Calypso to store data.

All MIPS-specific parameters and routines are in separate source files, so that Calypso should be easily retargetable to other similar machines. The data structures are even portable to 32-bit machines, where for debugging purposes they can be simulated on an Intel PC.

Work is still required to complete the locking support, and to test all features (especially link) exhaustively. Work is also progressing on measurement of the performance impact of each of the new features described here.

References

Andersson, A., S. Nilsson, 1993, 'Improved Behavior of Tries by Adaptive Branching', *Information Processing Letters*, 46:295–300. `http://www.nada.kth.se/~snilsson/public/papers/impr/`

Dillon, M., 2000, 'Design Elements of the FreeBSD VM System', *Daemon News*, January 2000. `http://www.daemonnews.org/200001/freebsd_vm.html`

Liedtke, J., 1994, 'Page Table Structures for Fine-Grain Virtual Memory', *IEEE Technical Committee on Computer Architecture Newsletter*.

Wiggins, A., Heiser, G., 1999, *Fast Address Space Switching on the StrongARM SA-1100 Processor*, UNSW-CSE-TR-9906. `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/9906.ps.Z`