Arena ProcessBase Implementation and Compliance

Ken Mayes

CNC/2001/001
Centre for Novel Computing
Department of Computer Science
University of Manchester.

October, 2001

1 Introduction

The pbase language (Morrison et al., 1999) has been developed to support Process Modelling applications. Programs written in pbase are executed by the pbase interpreter. The pbase language interpreter provides threads, semaphores and persistent objects to pbase programs. This document deals with the implementation of these entities based on the Arena run-time operating system (Mayes and Bridgeland, 1997). Comparisons are made with implementations on the Linux monolithic general-purpose operating system. One of the principal aims is to investigate mechanisms for implementing a compliant system architecture (Morrison et al., 2000). In monolithic operating systems such as Linux, the resource management policy is fixed within the operating system kernel. There are two basic problems with using monolithic kernels. Firstly, where resource management policies do not meet the requirements of the application, in this case the pbase interpreter, the policies must be reimplemented in the application. There are thus redundant components of the operating system, whose code remains unused. Secondly, where an operating system-provided abstraction (such as the Unix file) can be used to access a resource (disk blocks), but where the general-purpose nature of this abstraction may affect the efficiency of the implementation.

Arena is an attempt to solve these problems. The idea behind the Arena system is to separate resource management policy from the low-level mechanisms required to implement the policy. These low-level mechanisms are provided by a thin layer of software loaded directly onto the processor (the Arena Hardware

Object HWO)). The policy for resource management is provided by Arena resource manager libraries (implemented as C++ objects). The pbase interpreter is provided with interpreter-specific resource manager libraries with which it executes. Arena is thus a library or run-time operating system. This approach provides the possibility of implementing pbase as a compliant system. That is where there is

1. no reimplementation of resource management policy at levels in the system.
2. the ability to communicate application-specific information between levels in the system, both up and down[1]
3. reduced reliance on general-purpose abstractions.

The Arena implementations of the pbase interpreter show examples of these three compliant system features:

1. Arena threads and semaphores are used to directly implement pbase threads and semaphores.
2. Upcalls and downcalls between the interpreter and Arena are used to support persistence of threads and semaphores.
3. In Arena it is possible to directly read and write specific disk blocks. Even within the context of Arena, more general-purpose calls can be replaced by shared variables to increase efficiency at the cost of increased coupling.

There are two other aspects to compliance:

1. each level should comply with the functional requirements of the level above it.
2. each level should comply with the efficiency or performance requirements of the level above it.

2 Compliant Process Management in Arena

Interaction between the interpreter and Arena are via *downcalls* to Arena from the interpreter, and *upcalls* from Arena to the interpreter. The Process Manager (PM) Arena resource manager library is a recipient of downcalls, and an originator of upcalls. There are two sets of entry points into the PM corresponding to downcalls.

[1] Unix signals propagate information to the application and allow application-specific actions to be carried out. New calls can be added to the Unix system call, though modifications to monolithic systems tend to be non-trivial and may result in growth in size and complexity of the operating system code.

1. PM routines called from within pbase application programs, via an Arena pbase threads library. That is, the pbase program is linked with the Arena pbase threads library and can then make downcalls, via the pbase interpreter, into the underlying PM.

2. PM routines called from within the interpreter itself, so that the PM will perform some activity on behalf of the interpreter. An upcall may be made from the PM to the interpreter in response to a downcall, or in response to some occurrence.

2.1 support for maintenance of interpreter-level thread state

Interpreter-level threads are based on Arena threads, such that one interpreter thread corresponds to one Arena thread. Arena threads maintain PM-level state, such as the register context and C stack, necessary for the dispatching and scheduling of threads. The interpreter maintains additional interpreter-level thread state, relevant to the interpreter function, such as the pbase execution stacks, in a Thread Control Block (TCB) for each thread.

As threads execute, their status may change. The pbase thread interface library routine *getStatus()* returns the status of the interpreter-level thread. However, since the thread status may change as a result of activity within the Arena PM, there is a requirement that the state of the Arena-level thread is reflected in the status field of the TCB for that thread. Upcalls from the Arena-level help to maintain this state as the PM responds to downcalls from pbase via the interpreter. Similarly, Semaphore Control Blocks (SCBs) are used to maintain interpreter-level state of Arena-level semaphores (see section 2.2.1).

The effects on interpreter TCB status for Arena pbase interface library downcalls *yield()*, *suspend()*, *resume()*, *sleep()*, *sema_wait()* and *sema_post()* are:

- *yield()*: no thread status change from RUNNABLE

- *suspend()*: change status to SUSPENDED before making a downcall to the PM.

- *resume()*: change status to RUNNABLE before making a downcall to the PM. However, if the target thread has not been previously suspended, the PM immediately returns the value E_TH_NOT_SUSP. This information, that the target thread has a pending resume, is stored in the TCB of the target thread for use in restore, for example after reboot (see section 2.2.2).

- *sleep()*: change status from RUNNABLE to SLEEPING before making a downcall to the PM. The duration of the timeout period is recorded in the TCB corresponding to the thread.

- *interp_threadAwake()*: is an upcall into the interpreter from the timer-handling code of the PM when a sleep period ends. Changes TCB status from SLEEPING to RUNNABLE.

- *sema_wait()* retain status as RUNNABLE. However there is the possibility of an upcall if the calling thread becomes blocked on the semaphore.

  - *interp_blockOnSem()*: upcall in context of the sema_wait() downcall when the current thread will be blocked and a new thread will be dispatched. Changes TCB status from RUNNABLE to BLOCKEDONSEM, and updates the SCB copy of the semaphore count (for persistence, see 2.2).

- *sema_post()* retain status as RUNNABLE. However this call may result in an upcall from the PM if some other thread becomes unblocked as a result of this call.

  - *interp_unblockOnSem()*: upcall in context of sema_post() downcall when some thread, previously blocked on a semaphore, is unblocked. Changes status from BLOCKEDONSEM to RUNNABLE and updates updates the SCB copy of the semaphore count (for persistence, see 2.2).

2.2 support for persistence

The basic mechanism for implementing persistence lies in the persistent object model of the pbase interpreter. All persistent objects are reachable from a root object. At a checkpoint/stabilise operation, all reachable objects are written to disk in such a way that inter-object references on-disk can be resolved. Given that the most recently-written root object is always in a well-known disk location, all other checkpointed objects can be read into the interpreter heap, lazily or eagerly, on restoration. That is, references in some parent object enable the system to read in its children from disk, and so on. In addition to the objects created by a pbase application, the interpreter itself maintains objects, reachable from the root, which represent the state of entities in the interpreter, such as the TCB objects which maintain the state associated with threads at the interpreter-level. These interpreter- state objects are stabilised along with the application objects, and are read in on restore by the normal object restoration mechanism using some mechanism for resolving inter-object on-disk references.

2.2.1 restoring semaphores

The eventual solution to the problem of supporting persistent semaphores on Linux was to implement semaphores entirely within the interpreter. That is, as interpreter semaphore objects which, as all objects, are implicitly persistent.

However, this is not a compliant solution – any semaphore-supporting code within the operating system is present in the system but is unused, being re-implemented in the interpreter.

The main problem with basing interpreter-level semaphores on operating system-level semaphores lies in the restoration of semaphore state after a restore. The internal count of the semaphore has to be restored, and threads previously blocked on a semaphore must be placed back in the restored semaphore. The solution is to use Semaphore Context Block (SCB) objects which maintain information as to the current OS-level semaphore count and blocked threads, at interpreter-level, so that this semaphore state can be restored. The behaviour of the underlying Arena semaphores, in blocking and unblocking threads, is transmitted by upcalls, *interp_blockOnSem()* and *interp_unblockOnSem()*, from the PM to the interpreter level. This means that at each stabilise, the SCBs, each reflecting the current state of the underlying semaphore, are written to disk. On restore, each recovered SCB is used to create a new Arena semaphore, initialised with the stabilised semaphore count. Then, blocked thread information from the SCBs allows the appropriate Arena threads to be blocked on the correct semaphores by using the downcall *PM_blockThreadOnSem()*.

2.2.2 restoring suspended threads

The marking of the TCB status as SUSPENDED (see section 1) indicates that the corresponding Arena thread is suspended in the PM. On a restore after a reboot, the downcall *PM_blockOnSuspThs()* adds the Arena thread to the set of suspended threads within the PM. It is also necessary to restore state relating to threads which had received a *resume()* without a previous *suspend()*. The TCBs of such threads have a modified RUNNABLE state. This is restored in the PM using the downcall *PM_addPendingResume()*.

2.2.3 restoring sleeping threads

The marking of the TCB status as SLEEPING (see section 2.1) indicates that the corresponding OS-level thread is on a timeout queue in the Arena PM. On restore, the downcall *PM_addTimeoutThread()* adds the Arena thread to the set of sleeping threads within the PM. However, it is difficult to *efficiently* keep track, at interpreter-level, of the time remaining for a sleeping thread. Instead, the original sleep period is stored in the TCB along with the SLEEPING status (see section 1), and the restored thread begins its original timeout again on reboot. This is a compromise in complaince between efficiency and functionality. Here efficiency has been taken to be of greatest importance.

2.3 support for time-slicing of threads

The default behaviour of the interpreter on Arena is to rely on applications to use thread library calls (*yield()*, *suspend()*, *resume()*) to cooperatively schedule threads, and to alter thread priority (*getPriority()*, *setPriority()*), on the basis that application-specific information can best determine when threads should run. Scheduling is also affected by blocking and unblocking of threads on semaphores. However, where such cooperative scheduling mechanisms are inappropriate, either because of code complexity or performance insensitivity, two pbase thread library calls, *startSlicing()* and *stopSlicing()*, are provided to commence and to discontinue time slicing between Arena threads. More pragmatically, it was necessary to implement time-slicing in Arena for the performance comparisons with Linux store implementations (Section 15), since the Linux interpreter only provides timeslicing threads. With Arena, the application can determine, by the duration argument to *startSlicing()*, the period between preemptive thread switches. However, time-slicing is viewed as general-purpose mechanism where the system takes thread dispatching decisions away from the application. It should be noted that this Arena time-slicing is not intended to be a sophisticated implementation which gives equal slices to all threads. It is simply intended to cause a thread switch at application-specified intervals so that all runnable threads (whose priority is the highest of those on the run queue) will be dispatched in a round-robin fashion. A single timeslice may be shared between cooperatively-scheduled threads. That is, the timeslice is a property of the system, not of the individual thread. More sophisticated timeslicing policies could be implemented. However, as stated above, the default behaviour is that the pbase application should determine its own scheduling policy where possible, for example by introducing a *yield()* call when a thread has reached some safe-point in the application code.

2.4 atomicity of interpreter activity

2.4.1 atomic instructions

An atomic instruction is an interpreter instruction which is executed without a thread-switch occurring during its execution. However, there are instructions which themselves cause a thread switch, e.g. *yield()*. Here the current thread *ct* is descheduled and another thread is dispatched. When *ct* eventually runs again, it must safely return from the *yield()* instruction. That is, blocking instructions must be implemented so that they are reentrant. In the default behaviour of the Arena implementation, cooperative scheduling of threads is by means of library calls which are themselves executed as synchronous interpreter "instructions". Thus other instructions (providing they cause no I/O events) are necessarily atomic, in that no rescheduling of threads can occur while they execute.

There are two atomicity problems associated with low-level events.

timeslicing When thread timeslicing is turned on by the application by a call to *startSlicing()*, instructions can no longer be assumed to be atomic. A timer event, which would result in the preemption of the current thread, can occur at any point during execution. For this reason thread switching at timer events must be postponed if the timer event occurs within an instruction.

There exists a downcall to the Arena PM, *deferEvents()*, which specifies that the full handling of timer events is to be postponed. This has been generalised so that other events, such as those arising from disk I/O and network activity, could also be specified in this downcall. A subsequent downcall to *allowEvents()* checks for pending timer events and arranges for them to be handled safely. These calls were placed in the main execution loop of the interpreter and successfully assured interpreter instruction atomicity. It was only subsequently, when performance was considered, that it became obvious that these general-purpose calls were non-compliant with the efficiency requirements of the interpreter. Useful as they appear to be, from the perspective of separating mechanism and policy, in allowing applications to determine when to safely deal with specific low-level events, they were far too heavyweight for the main execute loop of the interpreter[2]. Mechanisms used in such a tight loop must be implemented efficiently. Thus, the *deferEvents()* and *allowEvents()* calls to the PM were replaced by a bitmask variable shared between interpreter and PM. With this method, when the timer was in use, the switching of threads on timer events was always deferred by default. The handling of timer events proceeded only to the extent of simply registering the fact that an event had occurred. The actual thread switch only took place when it was safe – that is, between instructions. This meant merely checking the shared bitmask, rather than making the two PM calls. Compliance with the efficiency requirements of the interpreter here has required tight coupling between the interpreter and the Arena PM.

rescheduling on I/O During a restore, for example after a reboot, threads accessing objects may cause the reading of disk blocks. There is a danger that instructions causing disk reads might be reentered by the blocking of the reading thread. Thus, say thread *th1* needs to restore object *o* – *th1* is blocked awaiting the result of the read, and thread *th2* is dispatched. But it is possible that *th2* also needs object *o* and so tries to read object *o* into a different address in the interpreter store, with resulting inconsistency. The introduction of some mechanism whereby *th2* is explicitly delayed from reading until *o* had been read by *th1*, would be possible. However, the cost of updating some state required to record all objects currently being read, plus the selective scheduling only of threads whose reads are safe, is prohibitive. It is simpler (at least in the case where objects reside on local disks) to simply make reads (and writes) atomic by blocking threads doing disk I/O. This is achieved within the interpreter-compliant

[2] This instruction execution loop was found to execute about 5.5 million pbase instructions per second.

POSM and PM which disables thread switches during disk reads and writes. This issue is dealt with further in the next section.

2.4.2 stabilising the object store

In a general system, for efficiency, the default behaviour might be to deschedule any thread waiting on a disk read or write, so that some other thread can run (though this obviously depends on the relative latencies of disk activity and thread switching). As noted in the previous section, threads, involved in an instruction involving disk I/O must not be descheduled. In the case of stabilising the store, the entire set of disk writes during the stabilise must be performed atomically. No other thread must run, as no updates must occur whilst the stabilise "snapshot" is being taken. In the default cooperative scheduling case, this is not an issue, as the checkpointing thread is guaranteed to run to completion of the stabilise. However with timeslicing, the preemptive timer events could occur within the stabilise execution. To ensure that stabilise is atomic when timeslicing is turned on, timer events are postponed using the mechanism described above.

2.4.3 garbage collection

With activities such as garbage collection, where no thread switch should occur, it is possible to simply raise the priority of the collecting thread to some priority which exceeds the maximum priority able to be set by an Arena threads library call *setPriority()*. Even if the system is time-slicing, and timer events are not deferred, the scheduler will always schedule the highest priority, collecting, thread. If I/O events occur during the garbage collection, these must be blocking so that no other thread can run.

2.5 pbase process management in Unix

Pbase threads were originally intended to be based on Solaris threads and, subsequently, on Linux pthreads. However, lack of compliance in this thread provision resulted in moving the thread implementation from the operating system into the interpreter itself. A single operating system thread of control is multiplexed by a scheduler in the interpreter to provide the interpreter-level threads. Although this allows an efficient implementation (in that the scheduler is tightly-coupled with the interpreter), the system *as a whole* cannot be said to be compliant – the operating system scheduler is redundant.

Pbase semaphores similarly were ultimately implemented within the interpreter because operating system-provided semaphores lacked compliance, particularly with regard to restoration after a reboot. Again, any provision for semaphores within the operating system is redundant.

3 Persistent store implementations

The bulk of the work described in this document consisted of implementation of persistent stores on Arena. Two existing Unix implementations from St Andrews University (Munro 1994) (an after-image store using disk page tables and a before-image store using a store buffer) were ported to Arena. A third implementation consisted of the before-image store modified to omit the store buffer. Two further implementations used a shadow-store or, more correctly, a mirror-store approach. These implementations are described below.

There are thus five stable store implementations on Arena:

1. After_image with disk page table (**aipt**)
2. Before_image
   (a) Before_image with store buffer (**bisb**)
   (b) Before_image with no store buffer (**binb**)
3. Shadow store
   (a) Shadow store with disk I/O at object_granularity (**shst_obgr**)
   (b) Shadow store with disk I/O at page_granularity (**shst_pggr**)

As noted above, the after_image with disk page table and before_image with store buffer stores are also implemented on Unix. Their performance on Linux was measured so as to have some comparison with the Arena ports.

3.1 Operating system support for the stable stores

The thread and semaphore implementation has already been described in section 2. The major difference in the stable store implementation is that the Unix implementations rely on the Unix file system, whereas Arena allows the interpreter to write directly to disk blocks. The Unix file system provides general-purpose abstractions for accessing disk blocks:

1. seeking to specific offsets within a file and then writing some number of bytes to, or reading some number of bytes from, that offset.
2. mapping and unmapping portions of the file into the virtual address space of the Unix process, so that portions of the file can be faulted into virtual address pages.

For efficiency Unix systems maintain some sort of file buffer so that all writes and reads do not have to go to disk. However for cases where writes *must* go through to disk, applications can specify that buffers are flushed to disk.

In contrast, the Arena HWO provides two operations *hwo_write_blocks()* and *hwo_read_blocks()* which access specific disk blocks directly. An application requiring some sort of buffering (such as the **bisb** stable store) would implement it on top of these HWO operations. However, applications not requiring a disk buffer do not implement one.

4 Levels in the persistent system

This section describes the overall structure of the system. Objects in the system can have an existence at three levels (Fig 1). This basic structure is as given in Munro (1994). When initially created by the interpreter, objects exist only in the *interpreter local heap*. In order to stabilise this local heap, each local heap object, reachable from the root, must have a representation in the underlying *stable heap*. This is achieved by explicitly creating an identically-sized object in stable heap, and copying between local heap and stable heap representations. Finally, objects in the stable heap are written to the lowest level, the *stable store* disk blocks.

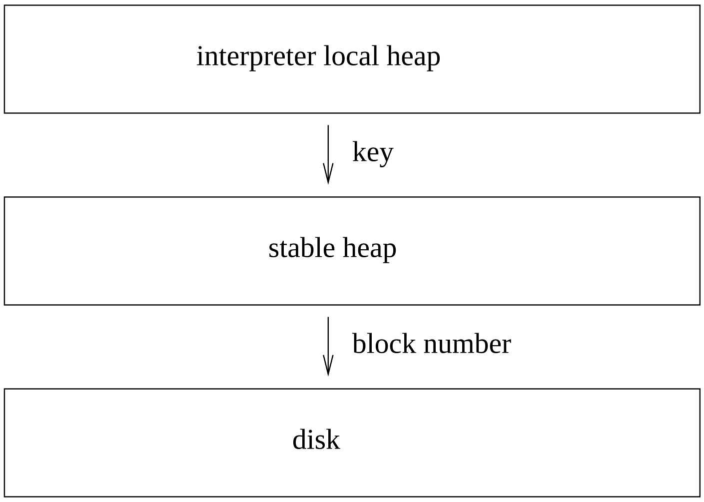

Figure 1: The three levels in the system at which objects can exist.

The following sections are intended partly as an implementation guide, and therefore name and describe specific functions in the interpreter and supporting code.

5 Interpreter local heap

This section describes the nature of the St Andrews local heap. Objects created by the interpreter are of two kinds.

1. pbase objects created as part of a pbase program.

2. internal objects of the interpreter, representing the state of the interpreter, such as thread control blocks.

The persistent system makes no distinction between these two kinds of objects, so that the state of the interpreter, as well as the state of the pbase program, is persistent.

The fact that an object can have three representations at different levels in the system means that there must a mappings between the levels.

The mapping between local heap and stable heap is by a unique *key* associated with each object. This key is allocated by the stable heap, when an object is created in stable heap. The local heap maintains a *KeyToRAM* table which maps object keys to virtual addresses in the local heap. In fact, the interpreter KeyToRAM table contains the virtual address of each object in the local heap. The key of an object resides in a prefix word at the start of the object. This basic local heap implementation is common to all stable store implementations.

5.1 Interpreter heap checkpoint

Inter-object references in the local heap can be via local heap virtual addresses or object keys. At checkpoint, all inter-object references are translated into keys. This may require the creation of new stable heap objects if necessary. Having inter-object references as stable heap keys ensures that object references will be meaningful to the stable heap level.

Checkpointing[3] the interpreter local heap is done via a call to *Nstabilise()*. This routine calls:

- *mark()* – go through the local heap *KeyToRam* table, mark all objects reachable from this table.
- *alloc_keys()* – go through the local heap (starting at the heap base) and, for each marked object, if the object has no key[4], then call *SH_create_object()*.
- *trans_keys()* – go through the local heap, and replace any inter-object references which are local heap virtual addresses with object keys[5].
- *write_out()* – called for each marked object in the local heap, this results in a call to *SH_write_words()* to copy the local heap object into its corresponding stable heap object.
- *SH_stablise()* – cause the stable heap objects to be written to disk.

[3] the terms “checkpoint” and “stabilise” are used as synonyms in this document.
[4] meaning that it has no representative object in the stable heap
[5] so the inter-object references will be meaningful in the stable heap

6 The stable heap interface

This interface is used by the local heap. It is the interface of the St Andrews heap. This interface is mainly of **object granularity**.

- *psint SH_first_object();*
  return the key of the root object.

- *void SH_close ();*
  close the entire stable heap.

- *psint SH_open(char *, openArgFunc,openArgFunc, openArgFunc, openArgFunc);*
  open the entire stable heap.

- *psint SH_read_word(psint key, psint index);*
  read a word at offest *index* in the stable heap object referenced by *key*.

- *void SH_read_words (psint key, psint index, psint *buffer, psint nwords);*
  read *nwords* at offset *index* in the stable heap object referenced by *key* into local heap *buffer*.

- *void SH_write_word (psint key, psint index, psint word);*
  write the word *word* at offest *index* in the stable heap object referenced by *key*.

- *void SH_write_words (psint key, psint index, psint* buffer, psint nwords);*
  write *nwords* at offset *index* in the stable heap object referenced by *key* from local heap *buffer*.

- *psint SH_create_object (psint size);*
  create a new object in the stable heap.

- *void SH_destroy_object (psint key);*
  destroy the specified stable heap object

- *void SH_stabilise (void);*
  write stable heap objects, modified since the previous call to *SH_stabilise()*, to disk.

- *void SH_garbage_collect(void);*
  garbage collect the stable heap.

- *psint SH_can_modify(psint key);*
  ensure that object referenced by *key* is shadowed in the stable heap before being modified.

- *void SH_stabilise(void);*
  cause the state of the stable store to be written to disk such that it can be restored on reboot.

In the Arena implementations, this interface is provided by a Persistent Object Store Manager C++ class.

7 Stable heap internal routines

Underlying the stable heap interface are stable heap internal routines used to implement the required functionality. In the St Andrews Unix implementation, the SH layer calls a set of SS primitives. In Arena, this functionality is provided by a DiskManager C++ class. These internal routines are briefly described here for two reasons:

1. to document the mechanism by which the stores function in reading from and writing to disk.
2. to provide a background for the performance comparisons in Section 15.

These routines are **page granularity**. Some implementations do not use all these routines. A page here means a small region of stable heap space. The process of shadowing means allocating a disk block to hold a copy of a page.

- *saveVM(u_int addr1, u_int addr2);*
  depending on the state of pages between *addr1* and *addr2*, take steps to ensure that the pages will be saved to disk at the next stabilise.

- *shadowCopy (u_int page);*
  allocate a disk block to shadow the stable store page *page*, update the page-to-block mapping, and write the page to the block.

- *mapPage (u_int page, u_int flags);*
  allocate a disk block to shadow the stable store page *page* and update the page-to-block mapping and page state with *flags*.

- *stabilise();*
  cause all designated pages to be written to disk.

- *readWords(u_int key, int index, int *buffer, u_int nwords);*
  read *nwords* from offset *index* in the stable heap object referenced by *key* to local heap *buffer*.

- *writeWords(u_int key, int index, int *buffer, u_int nwords);*
  write *nwords* at offset *index* in the stable heap object referenced by *key* from local heap *buffer*.

- *keyToAddress(u_int key);*
  return the virtual address, in the stable heap, of the object referenced by *key*. This address may then be used to write to or read from the object directly. The use of this routine thus assumes that the stable heap pages are contiguous. It cannot be used in all implementations.

- *writeToKey(u_int key, u_int obj);*
  write the value *obj* at offset *key* in the stable heap.

8 How disk activity occurs

It is not always immediately clear what causes disk I/O to occur. The purpose of this section is to identify exactly how disk activity occurs. As will be discussed in the results section (Section 15), in the Linux implementations, issuing a write or read does not necessarily cause actual disk device activity because of internal buffering in the file buffer cache.

Underlying the behaviour of the stable stores are the mechanisms, by means of which, pages in the stable heap are written to, and read from, the blocks of the disk. Whilst a call to *SH_stabilise()* explicitly causes objects to be written to disk, other *SH* primitives may implicitly cause reads and writes. These are *SH_can_modify()*, *SH_read_words()* and *SH_write_words()*.

SH_can_modify() is called when the interpreter is about to modify some object. It ensures that, if the store occupied by the object needs to be shadowed in order to guarantee its persistence, then that store will be shadowed. The need for writing to disk is determined by the state associated with the stable store page on which the object resides. This page state is described in subsequent sections for each store. *SH_read_words()* and *SH_write_words()* can also lead to reading a page from disk in the case where the stable store page is not resident. A page might not be resident because it was previously flushed to disk from a store buffer (in the case of **bisb**) or because a reboot and restore has occurred at some previous checkpoint. Figures 2 and 3 show how these stable heap interface routines can cause disk accesses, on Unix and on Arena respectively.

9 Stable heap layouts

At the stable heap level, an object key is used to locate the object in the stable heap.

There are two basic stable heap implementations. In the first, memory for objects is allocated by a "moving pointer" into the heap which is simply incremented by object size each time an object is created. In the second, stable heap pages are maintained in a free list, and, when used, each page is divided into equal_sized slots, chained into a free list, for object allocation.

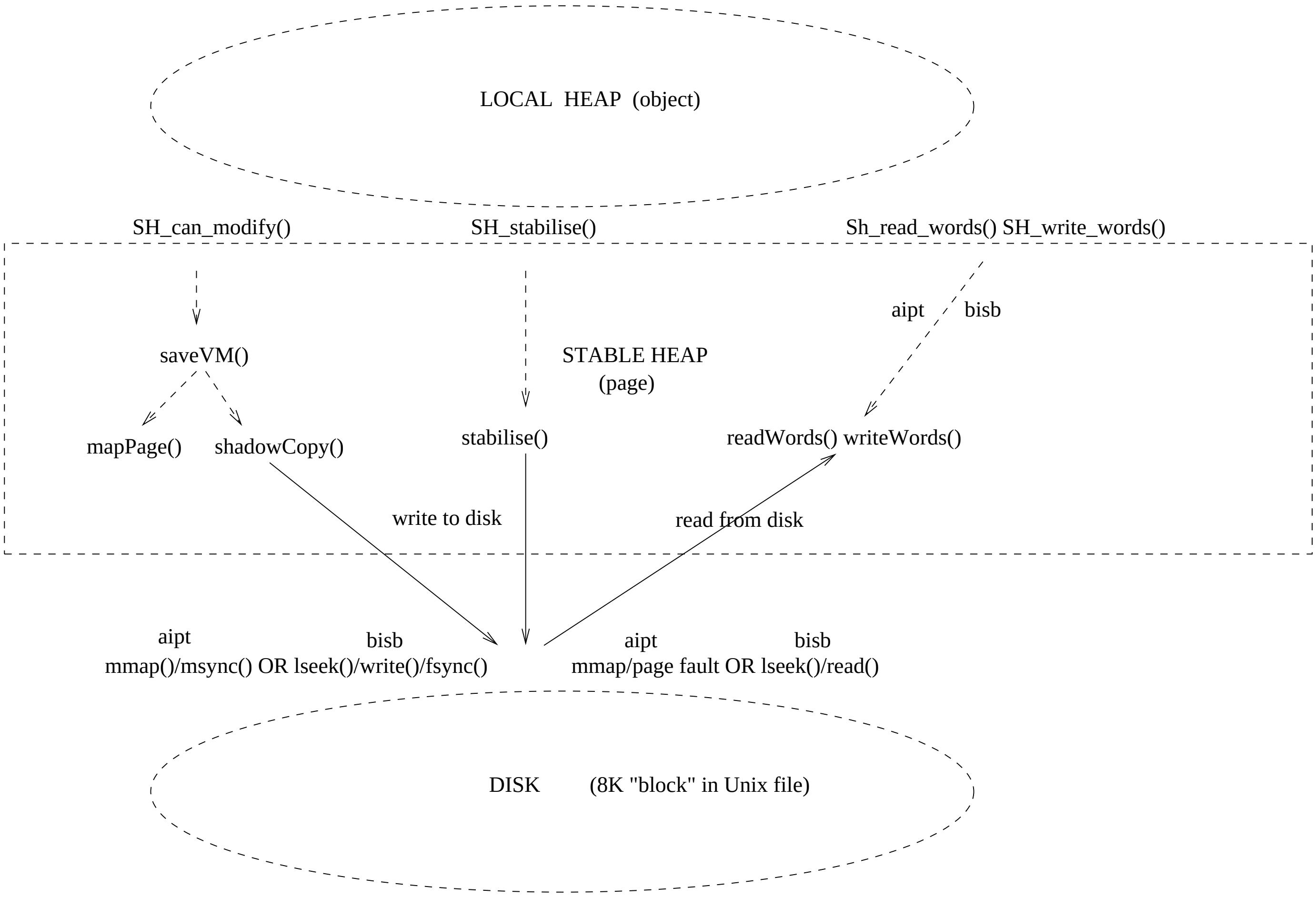

Figure 2: Levels and interfaces in Unix implementations. The after-image with disk page table (**aipt**) implementation is based on *mmap()* and *msync()*, whereas the before-image with store buffer (**bisb**) implementation is based on *lseek()*, *read()*, *write() fsync()*. Both use a conventional Unix file as the stable store.

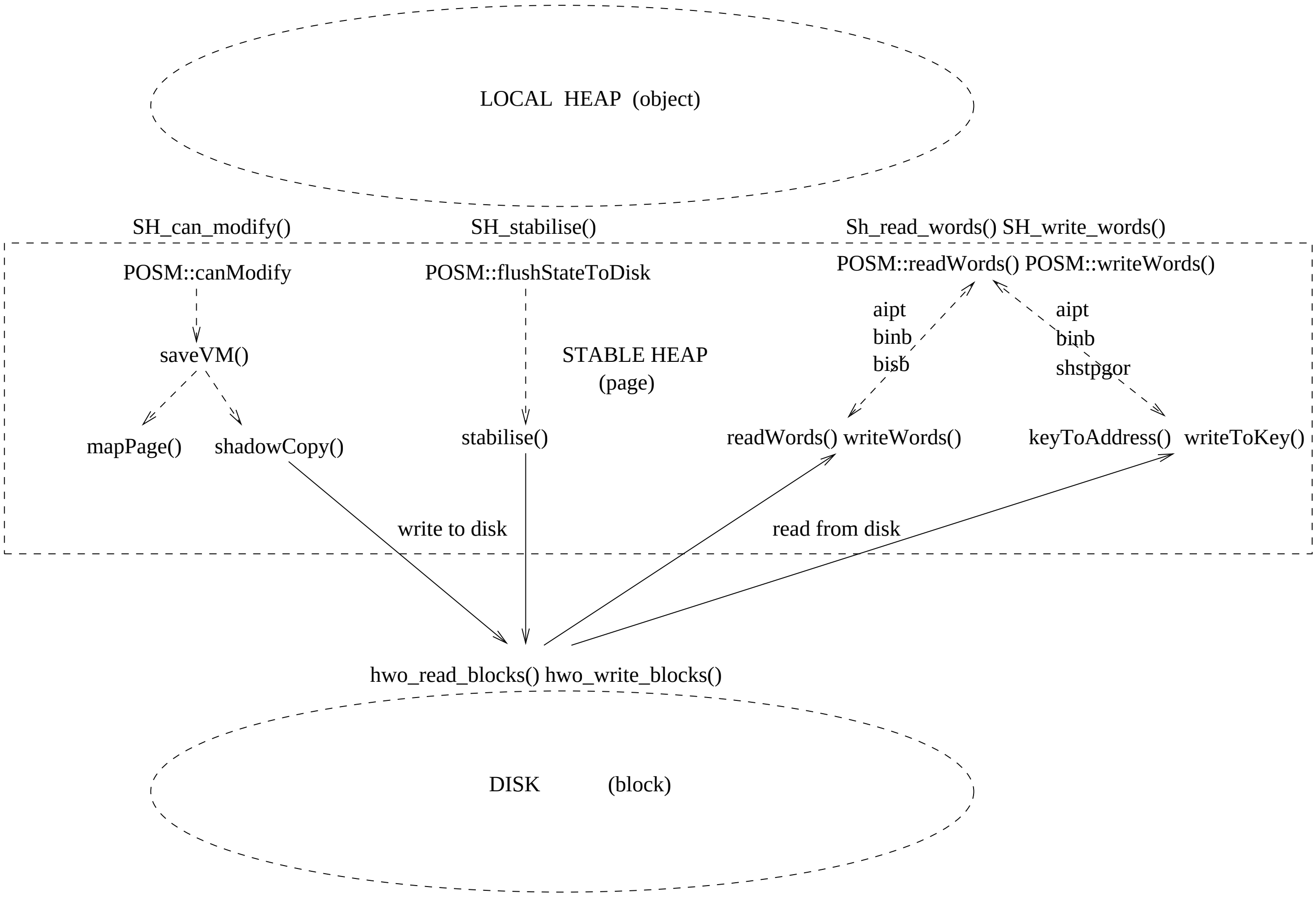

Figure 3: Levels and interfaces in Arena implementations. POSM-level *SH_readWords()* and *SH_writeWords()* can be implemented using either a lower-level *readWords()* and *writeWords()*, or using *keyToAddress()*. In Arena, disk sectors are read and written directly.

9.1 Moving pointer stable heap

This is the original St Andrews implementation (Munro, 1994). The heap is divided logically into two regions (Fig 4). Keys are allocated from the upper region, and objects from the lower. The system maintains a pointer into each of these heap regions, and these pointers are used to allocate keys and objects respectively. In this scheme, the key is simply an offset to the key region of the stable heap. At this offset is stored the offset to the associated object within the stable heap. So the key is an indirection to the object. All values used in this heap are *offsets*, not actual virtual addresses.
Features:

- allocation is easy
- random deallocation is difficult
- Because the key space and the object space are both in the stable heap, keys and objects can be located on disk via a single mapping mechanism.
- Objects may be allocated across page boundaries.
- no store is wasted
- large and small objects use the same allocation mechanism.

This layout is used in the **aipt**, **bisb** and **binb** stable heaps.

9.2 Object slot stable heap

The heap is explicitly divided into pages of MMU-page size (4Kb). These pages are linked together into a page free list. Pages are allocated from this free list and join a used pages list. Each page in the used pages list is divided into equal-sized slots for object allocation (Fig 5). Slot sizes are power-of-2 bytes, up to MMU-page size. Objects greater than MMU-page size require the allocation of contiguous pages in clusters. An object key is generated from a counter, and is mapped to the slot virtual address via an Indirection Table. The Indirection Table entry also gives the disk block address of the object (section 13).

Features:

- a further data structure is required to map stable heap keys to the virtual address of an object – the Indirection Table. It is also used to store the disk address of the object[6].
- store is wasted because object size will usually be less than slot size.

[6] An Indirection Tabel entry is $<$ OID, stable heap address, disk address $>$

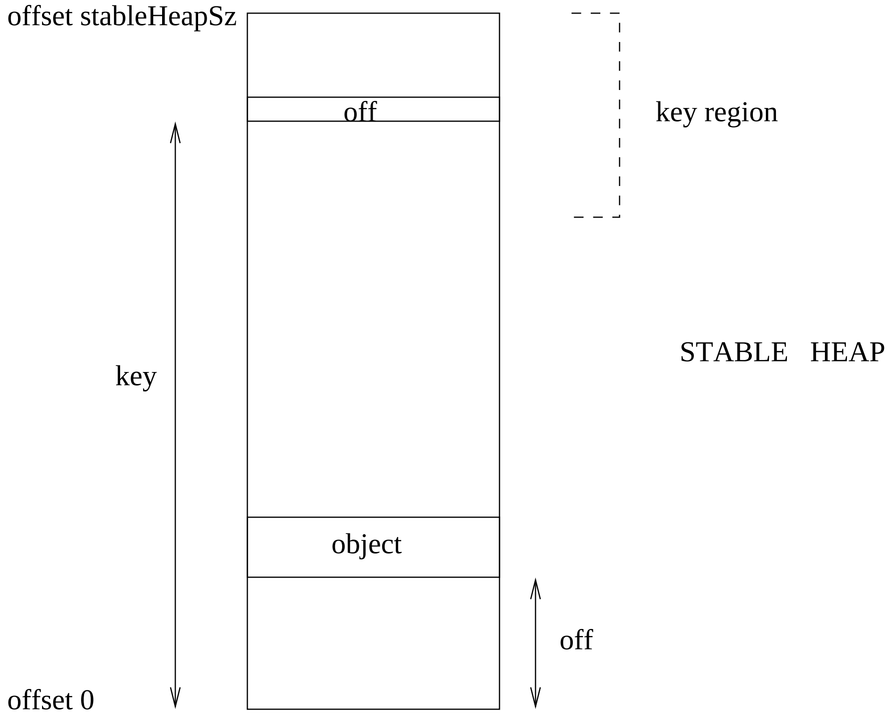

Figure 4: The moving pointer stable heap. Objects, and keys, are allocated from the stable heap by means of pointers into the heap. Keys are allocated downwards, objects upwards. The key associated with an object is the offset to a location, in the key region, which holds the offset to the object itself. Offsets are from zero, but can be transformed into virtual addresses. See section 10

- random deallocation is very easy (see Section 14).
- fragmentation may occur
- object larger than a page requires a further mechanism – a cluster of pages.

10 Mapping between stable heap and disk

In order to locate the objects on disk, there must be some mapping between the stable heap and the disk. However, this mapping depends on the stable heap layout. With the 'moving pointer' stable heap, the heap is divided into 8Kb 'pages' and these are mapped individually to 8Kb disk 'blocks' (see sections 10.1, 10.2). With the 'object slot' stable heap, the heap to disk mapping can be at the object level of granularity (section 10.3) or at the MMU-page level of granularity (section 10.4).

Note that the nature of the mapping between stable heap page and disk block used depends largely on the stable store persistence mechanism. That is, if the

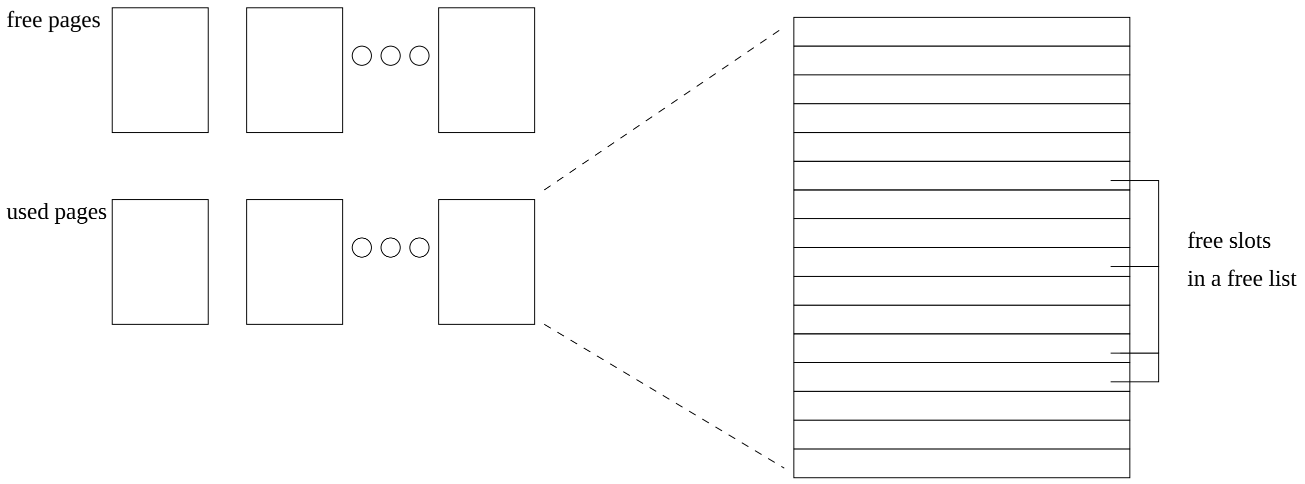

Figure 5: The object slot stable heap. Stable heap pages are used to provide fixed-size slots for object allocation and deallocation.

disk "block" associated with a stable heap "page" changes with the shadowing process, then the mapping must be able to reflect that change.

10.1 Disk Page Table mapping

This is used with the moving pointer stable heap layout. A disk page table (DPT), indexed by stable heap page number, can map stable heap pages to disk blocks. Each DPT entry contains the disk block number and various flags. This mapping mechanism allows the association between a stable heap page and a disk block to be changed over time. It can thus be used in the after-image implementation (Section 11). It is described in Munro (1994).

10.2 One-to-one mapping

This is used with the moving pointer stable heap layout. Here the stable heap page maps to the numerically-corresponding disk block, so that page n is always written to block n at stabilise. The before-image stores implemented here, which do not alter this page/block relationship, can use this simple mapping (Section 12).

10.3 Object granularity mapping

This is used with the object slot stable heap layout. The precise disk address of an object within a disk *sector* is given by an entry in an Indirection Table, specifying the disk sector number and offset within that sector. This disk address is used to enable the system to read and write individual objects (Section 13).

10.4 Page granularity mapping

This is used with the object slot stable heap layout. The precise disk address of an object within a disk 'block' of MMU page size (i.e. 4Kb – a set of eight 512b disk sectors) is given by an Indirection Table entry. The offset of the object within this 4K space is also kept, but this offset is only used to locate the object after the entire 4K 'block' has been read into the stable heap. That is, this approach is page-oriented in order to meet the efficiency requirements of the stable store (Section 13).

11 After_image stable store description

With the after-image mechanism, a modified page is never written back to disk to same block from which it was read. Instead, when about to be first modified, the page is associated with a new unused disk block, and the stable heap to disk block mapping is updated. This is a no_undo/no_redo system. The description of this mechanism is found in Munro, 1994. Because the mapping between stable store page and disk block changes, a 1:1 mapping cannot be used. In this implementation, a disk page table is used to record this mapping.

There are two ports, the original St Andrews Unix version ported to Linux[7], and a port of it to Arena. However, much of the implementation is common, and largely involved putting the original C code into C++ classes.

11.1 Common Linux and Arena implementation

The stable heap uses the 'moving pointer' mechanism for object and key allocation (section 9.1). It uses the Disk Page Table stable heap to disk mapping (see section 10.1). There are always two DPTs:

1. a DPT safely stored on disk, allowing a previous stabilised system state to be restored on reboot,

2. a DPT currently in RAM, being used to map between stable store pages and new shadow disk blocks at the next stabilise.

That is, DPT on disk represents the previous stable state. The transient DPT in virtual memory represents the current state. This transient DPT is written to disk (along with the modified pages) at the next checkpoint. It then becomes the stable page table. The transient page table copy remaining in virtual memory is subsequently updated with new mappings, and so on.

The objects in the stable heap at stabilise refer to each other by means of stable heap keys. A key is a stable heap offset (see section 7.1 and Fig 4), and

[7] This Unix implementation had to be modified slightly to run on Linux because of assumptions made by the interpreter about the way Unix chooses regions of virtual address space for *mmap()* and *malloc()*

this offset page has an entry in the DPT. In this DPT entry is the disk block to which the stable heap page, containing the key offset, has been written, and which can be read into the stable heap page. The contents of the key address is the offset to the object, whose disk block can again be found from the DPT. In this way on-disk inter-object references can be resolved to stable heap addresses, if necessary, on restoring a stable heap on reboot.

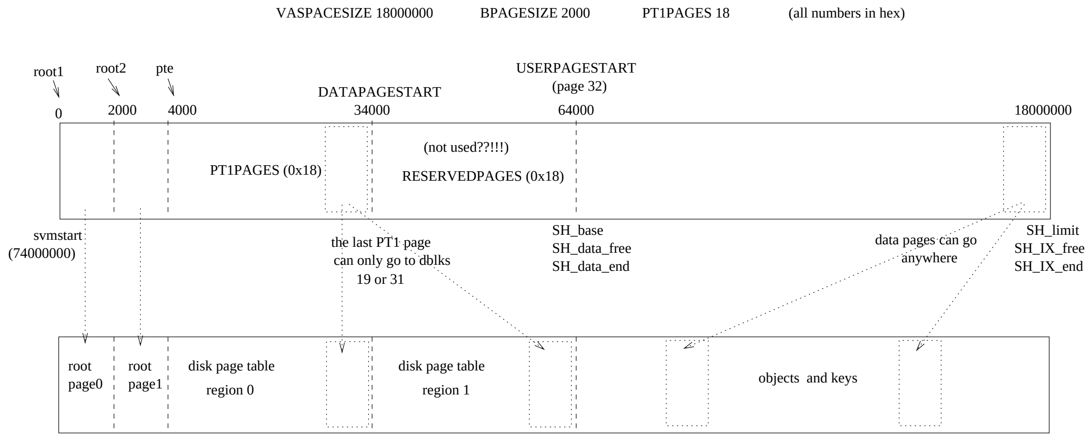

Figure 6: The basic layout of the after-image stable heap in RAM and on disk. The PT1PAGES region contains the pages occupied by the in-RAM disk page tables. This is written to disk to one of two sets of disk blocks alternately. In the figure, the last data page (which contains key offsets) is mapped to some disk block by its disk page table entry. This entry will be in the page table in the last page of PT1PAGES. This page table page will be written to a disk block (either 19 or 31) by the last entry in the secondary page table of the current root page. Each root page contains a disk page table for the PT1PAGES.

There are two root pages, stored in the first two blocks of the disk. There are two sets of disk blocks which are used to hold the disk page table pages. These disk page table pages themselves are able to be located in the disk by a secondary page table in each root page (Fig 6).

11.1.1 After-image stable store page state

In order to understand how the St Andrews after-image mechanism functions, it is necessary to understand page state. The writing of pages to disk at stabilise is based on the state of the stable heap pages. The DPT contains flags which give the state of the page with respect to making a shadow copy and writing to disk at stabilise. The state of each stable heap page is given in its DPT entry.

- DONTNEED means that no block is associated with this stable heap page.
- PROTECT means that the page has been written to disk on last stabilise, and so must now be shadowed to a new disk block before being modified.

- SAVE means that the page already has a shadow disk block associated with it, and that it will be written to disk at next stabilise. SAVE is defined as (PROTECT|WRITTEN).
- WRITTEN means that there is a shadow disk block already associated with the page, and that the page can therefore be modified.
- BLOCKLISTED means that the page has been placed on the list of pages to be written to disk at the next stabilise (presumably this blockList is used for efficiency during stabilise so that all DPT entries do not need to be searched for state SAVE).

During execution, *SH_can_modify()* in the interpreter calls the stable heap routine *saveVM()* for all stable heap pages occupied by the object (see sections 6 and 7). If the state of a page is PROTECT, a new disk block is allocated and entered in the DPT entry for that page with state SAVE and BLOCKLISTED. This means that all writes on that page will now go to the new, shadow, disk block. The page is entered in the list of pages to be written to disk at next stabilise. The page is written to the new block via *shadowCopy()* [8]. A page with state DONTNEED, which is being dealt with by *saveVM()*, is a stable store page not previously checkpointed. That is, it is a new stable store page. It is mapped to a shadow disk block in the DPT entry and its state is set to SAVE, but there is nothing to copy immediately to the shadow block.

At *SH_stabilise*, pages with state SAVE are written to disk, and their state is set to PROTECT. For efficiency, only pages in the blockList, with state BLOCKLISTED, are examined.

SH_read_words() and *SH_write_words()* have no effect on the state of stable store pages in this St Andrews after-image implementation. These routines however may cause the page to be read in from disk. The mechanism for reading the page differs in the Unix and Arena implementations.

The actions of *SH_can_modify*, *SH_stabilise*, *SH_read_words()* and *SH_write_words()* in the after-image stable store are summarised in Figure 7.

11.2 Unix-specific after-image implementation

The local heap is obtained via *malloc()*. The disk level is implemented using a unix file. The stable heap is implemented using *mmap()* – that is, *individual* 8K pages of the stable heap are *mmap()*ed to blocks of the file. A disk block is an 8Kb chunk of the file. A call to *msync()* is required to ensure that the stable heap pages are written to disk in *SH_stabilise()*. Reading pages from disk, which may occur when doing *SH_read_words()* or *SH_write_words()*, occurs on a page fault on *mmap()*ed pages. Writing a page to a new shadow disk block in *shadowCopy()*

[8] Whether this write actually goes through to the disk is discussed in section 15.

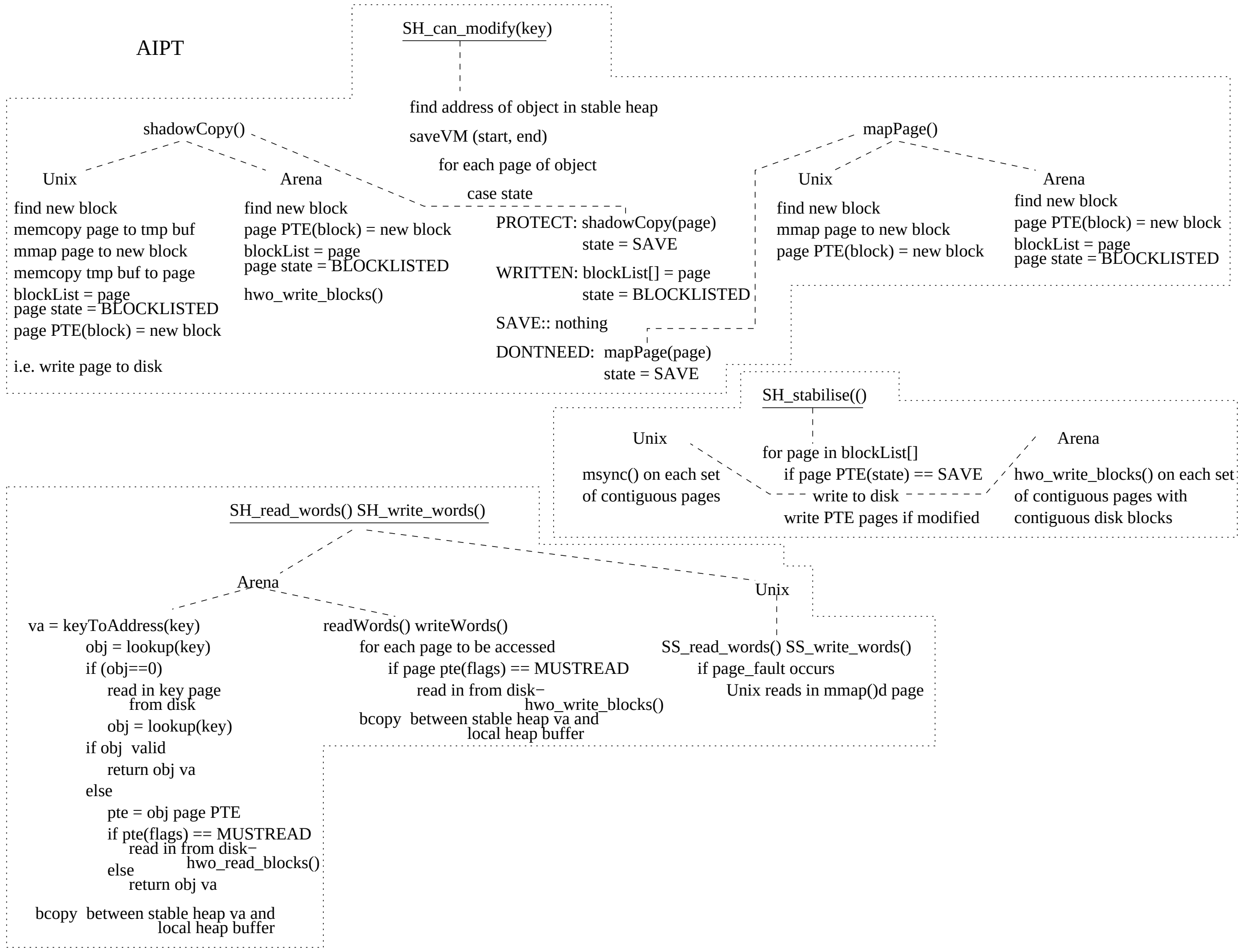

Figure 7: After-image disk page table implementation basics on Arena and Unix. Note that where an object occupies more than one page, *all* pages occupied by the object will be saved at stabilise, not just those written by SH_write_words(). In fact, SH_write_words() has no effect on page state maintained in the disk page table. Pages written at stabilise() are those on the blockList[] and having state SAVE. This occurs only through the SH_can_modify() interface. For the Arena implementation, an additional page state value, *MUST_READ* has been added. Accessing a *MUST_READ* page causes it to be read in from disk. In contrast, the Unix after-image implementation relies on page-faulting on an mmap()d page to cause a page to be read in from disk. There are two possible implementations of SH_read_words() and SH_write_words() in Arena, one based on an internal keyToAddress() routine, and one based on internal readWords() and writeWords() routines.

(section 7 and Section 11.1.1) occurs by copying the page to a temporary buffer, *mmap()*ing the page to the new 'block' in the Unix file representing the disk, and then copying the buffer back into the page[9].

11.3 Arena-specific after-image implemenation

The local heap is obtained via Arena *malloc()*. The stable heap is a range of valid virtual addresses obtained by a call to the HWO operation *hwo_validate_varange()*. Various properties of the virtual address range can be specified. In this case, physical pages will be allocated, by the HWO, to virtual pages on first access (i.e. on page fault), invisibly to the application. The disk block level is implemented by actual disk blocks, so that the DPTs contain actual disk sector numbers. These blocks are accessed directly via *hwo_posm_write_blocks()* and *hwo_posm_read_blocks()*. Whereas *mmap()* works by page faulting and reading the store file into the faulting *mmap()*ed address, the Arena implementation reads stable store pages from disk directly, without the need for a page fault. This is achieved by adding an additional stable heap page state to the DPT entry – MUST_READ. With Arena writes and reads of sectors via *hwo_posm_write_blocks()* and *hwo_posm_read_blocks()* occur immediately to disk. There is not file buffer cache, and no need for some *sync()*-like operation.

The Arena after-image port can use either of two implementations of *SH_read_words()* and *SH_write_words()*. One implementation is based on *keyToAddress()*, the other on *readWords()* and *writeWords()*[10]. The *keyToAddress()* implementation can be used here because objects are stored in contiguous stable heap pages.

aipt summary Key is an address – an offset into the stable heap.

- *keyToAddress()* and *writeToKey()* – job of reading in pages from disk if state == *MUST_READ*.
- *readWords()* and *writeWords()* – job of reading in pages from disk if state == *MUST_READ*.
- *saveVM()* – put page on *blockList[]* and set state to *SAVE*.
- *stabilise()* – write *blockLIst[]* pages with state == *SAVE* to disk.

12 Before_image stable store description

With a before_image implementation, when a page is about to be first modified, a copy of the page is written to a new unused disk block, and modifications

[9] This write may not go through to the disk.

[10] Source code note – at present the implementation used, in a given build of the after-image POSM, is based on a conditional compilation flag 'KTA' in POSM_shadowPage.C.

to the page take place on the original disk block. That is, the shadow copy contains the *old* data - a before-modification image of the page. The location of the shadow pages (disk blocks) is recorded in a *beforeMap* data struct which basically consists of <page number, shadow block> pairs. The beforeMap must be written to disk before modifying the original disk block (so the old data can be found if necessary). On restore, the data struct with the <page number, shadow block> pairs is obtained from disk, and the old data copied to each modified page from the shadow block in the data struct. That is, any updates since the last checkpoint are undone. That is, this method takes a copy of each original page/disk block before the page/disk block is modified. The record of the copies and their page location is thus a record of the previous state of the data that has changed since the last checkpoint. So to get back to the last checkpoint state, the original copies are used to overwrite the changes since the last checkpoint. This is an undo/no_redo system. It is described briefly in Munro (1994).
Because the relation between a stable store page and the disk block to which it is written does not change, the mapping can be 1:1 (section 10.2). However, the additional data structure, the beforeMap, is required to record the before_image shadows.

There are two before_image stable store implemenations. Both use a moving pointer stable heap layout. As was stated in section 9.1, all values used in this heap layout are offsets within the heap. Actual virtual addresses of the objects must be somehow determined. The mapping from stable heap offset to virtual address is different in the two before_image implementations:

1. Here, the virtual address of objects in the stable heap are found by adding the start virtual address of the stable heap. This is called here the **before_image no buffer stable store (binb)**. An object key is an offset in the stable heap. As with the after_image implemenation, the virtual address of this offset is found by adding the stable heap start address, and contains the offset to the object associated with the key.

2. Here, objects reside in the pages of a store buffer. The stable heap offsets map to the virtual addresses of these store buffer pages. This is called here the **before_image with store buffer (bisb)** stable store. An object key is an offset within the stable heap. However, stable heap offsets are not translated to virtual addresses by adding a start address. Objects and keys actually reside in the store buffer pages, so offsets in the stable heap must be resolved into store buffer addresses. That is, stable heap "addresses" are purely logical names used to

    (a) locate the objects in the store buffer.

    (b) map stable store pages to disk blocks (using a 1:1 mapping, Section 10.2).

To locate objects in the store buffer, it is necessary to search the store buffer for the store buffer page currently associated with a stable heap "address". Both keys and objects are located in this way.

The store buffer is a vector of pg_buffer structs, which contains an 8K page copy and the logical page number in the stable heap currently associated with the page copy. Each pg_buffer also has an lru field used when a buffer has to be paged out due to overflow (there are 32 pg_buffer structs used in the Linux and Arena implementations). A pg_modified flag is maintained for each page in the store buffer.

There are two ports of the **bisb** stable store, the original on Linux and a port of it to Arena. The **binb** stable store is only ported to Arena.

12.1 before_image common implementation

The stable heap uses the 'moving pointer' mechanism for object and key allocation (section 7.1).

It uses a 1:1 mapping between stable heap page and disk block, such that stable heap page n will be written to disk block n (section 8.2).

The layout of the stable store is shown in Figure 8. The stable heap and disk are divided into an *active space* and a *shadow space*. The shadow space blocks are used to keep "before update" copies of the pages in the active space. At stabilise the active space stable heap page is written to its corresponding active space block.
The root page consists largely of the beforeMap vector (see previous section). There are two root pages, and two root blocks.

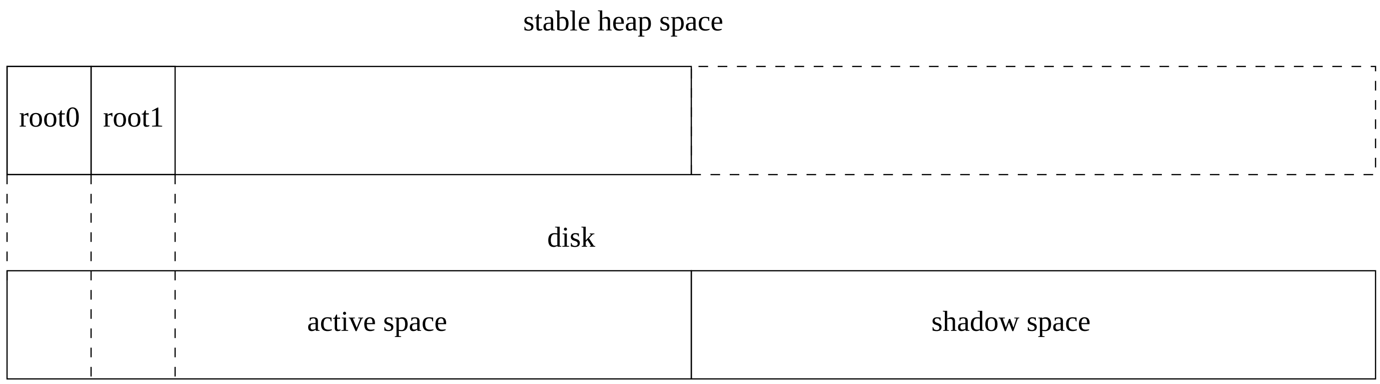

Figure 8: before image *active* and *shadow* spaces. In the Linux implementation, the disk is actually a Unix file.

On stabilise, modified pages are simply written to the original disk blocks, the beforeMap data structure is thrown away and the shadow blocks discarded. When using the beforeMap vector, the vector element index gives the shadow disk block and each vector element contains the page address shadowed to that block.

12.1.1 Before-image stable store page state

The state of logical pages of the stable heap is maintained in two bitmasks (the equivalent of the DPT flags in after_image). These bitmaps are protectedBM, writtenBM. A further flag, pg_modified, necessary to the implementation, is implemented differently in **bisb** and **binb** stable stores. The actions of *SH_can_modify()*, *SH_stabilise()*, *SH_read_words()* and *SH_write_words()* in the **bisb** stable store are summarised in Figure 9.

- PROTECTED means that the page must be shadowed to a disk block before being modified.
- WRITTEN means that the page has been shadowed.
- PG_MODIFIED means that the page has been modified.

At startup, and after stabilise, all pages are set as being *protected*, not *written* and not *pg_modified*. The interpreter calls *SH_can_modify()* to cause a stable heap page to be shadowed via the *saveVM()* routine, and must be called *before* pages are modified. The protected and written flags act to indicate whether a page should be shadowed in the call to *saveVM()*. After shadowing a page its written flag is set to true.

SH_readWords() and *SH_writeWords()*, as with the after-image store, may cause a stable heap page to be read in from disk. The pg_modified state is set to true only as a result of *SH_write_words()*, called by the interpreter *write_out()* routine which writes valid local heap objects to the stable heap (see Section 5.1).

In *SH_stabilise()*, if a stable heap page has state pg_modified and written, then the page is written to its active space disk block. That is, only modified (==pg_modified) and shadowed (==written) pages are written to disk at stabilise.

12.1.2 Unix implementation (bisb)

Only the **bisb** stable store is implemented on Linux.

bisb The local heap is obtained via *malloc()*. The stable heap is simply a logical space that is not directly associated with virtual memory. The store buffer is a static vector of *pg_buffer* structs[11]. The disk blocks are implemented as a Unix file. Unix system calls *lseek()*, *read()*, *write()* and *fsync()* are used to access these "blocks". The protected and written page flags are maintained as bitmasks. The *pg_modifed* flag resides in the state associated with each page of the store buffer.

[11] An 8K page plus associated state, including the *pg_modified* flag.

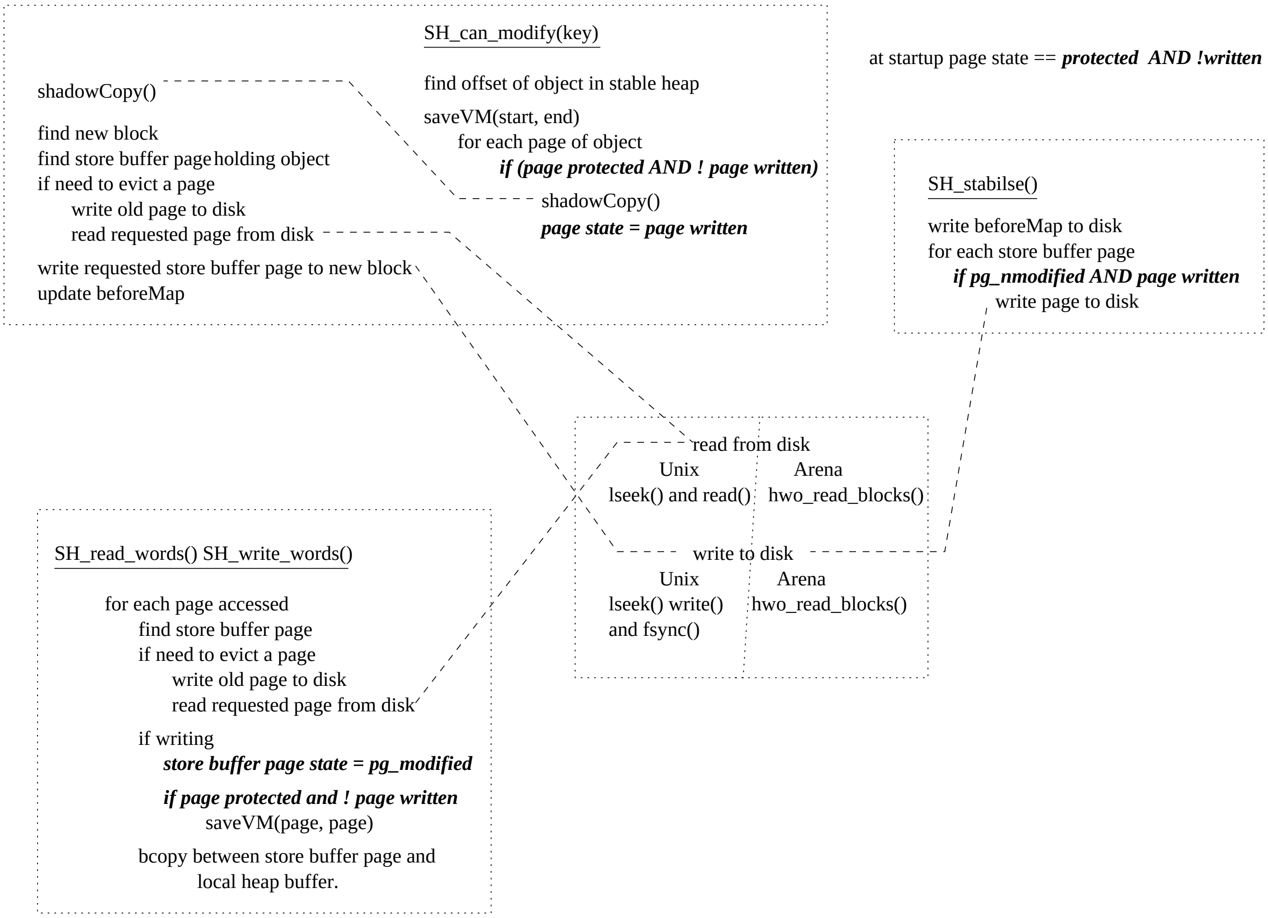

Figure 9: Before-image with store buffer implementation. Arena and Unix implemenations are very similar, differing only in how pages are read from and written to disk. *SH_write_words()* modifies page state such that only modified pages are written to disk at the next stabilise. Calling *SH_can_modify()* on a page is not sufficient to cause it to be saved at stabilise, as it only sets the page state to be *written*. Only *SH_write_words()* sets the *pg_modified* flag of the store buffer page.

12.1.3 Arena implementation (bisb and binb) (bisb and binb) (bisb and binb) (bisb and binb) (bisb and binb) (bisb and binb) (bisb and binb) (bisb and binb) (bisb and binb)

Both **bisb** and **binb** stable stores are implemented on Arena.

bisb The local heap is obtained via *malloc()*. As in the Unix implementation, the stable heap is a purely logical space that is mapped to the store buffer. The store buffer is obtained by a call to *hwo_validate_va_range()* to validate a range of virtual addresses to which physical pages will be allocated on page fault. The use of a store buffer means that consecutive stable heap pages may not be represented by consecutive store buffer pages. This means that the POSM interface *keyToAddress()* which assumes that objects crossing page boundaries are on contiguous pages, cannot be used to implement *SH_readWords()*.

bisb summary Key is an address – an offset into the stable heap.

- *readWords()* and *writeWords()* – job of reading in pages from disk if not in store buffer. Sets *pg_modifed* == TRUE on write.
- *saveVM()* – if page *protected* shadowCopy() and set state *written*. *shadowCopy()* may evict a page to disk and read in from disk if requested page is not in store buffer.
- *stabilise()* – go through store buffer pages. If a store buffer page is *pg_modified* and *written* then write to disk.

binb Uses a similar technique to the after_image implementation to avoid reading stable store on page faults in the stable heap – but whereas the after_image uses the DPT to store the mustRead flag, this implementation uses a *mustRead* bitmask. The implementation retains the *protected* and *written* bitmasks of **bisb**. The *pg_modifed* flag is implemented as a bitmask. Whereas the **bisb** implementation writes to disk only one stable heap page at a time (one 8K "page"), the **binb** can write several pages, because pages may be contiguous. Because stable store pages are guaranteed to be contiguous, there are two possible implementations of *SH_read_words()* and *SH_write_words()*. As with the **aipt**, either *keyToAddress()* or *readWords()/writeWords()* may be used [12].

binb summary Key is an address – an offset into the stable heap.

- *keyToAddress()* and *writeToKey()* – job of reading in pages from disk if state == *must_read* in mustReadBitMask. Sets state to *modified* in modifiedBitMask if write.

[12] Source code note: At present the implementation used, in a given build of the **binb** POSM, is based on a conditional compilation flag 'KTA' in POSM_**binb**_shadowPage.C

- *readWords()* and *writeWords()* – job of reading in pages from disk if state == *must_read* in mustReadBitMask. Sets state to *modified* in modifiedBitMask if write.

- *saveVM()* – if page *protected* in protectedBitMask call *shadowCopy()*. Set state to *written* in the writtenBitMask.

- *stabilise()* – go through store buffer pages. If a store buffer page is *modified* and *written* then write page to disk.

13 shadow store

In this POSM implementation, the stable heap is viewed as being a cache of objects. Information about each object is maintained in an Indirection Table (see section 9.2). Each entry is a <key, virtual address, disk address> tuple representing an object. In normal execution, the inter-object references are either keys or virtual addresses in the stable heap. At stabilise, these inter-object references are translated, or "swizzled" into disk addresses. In contrast, the **aipt**, **bisb** and **binb** POSMs, inter-object references on disk are the keys used at stable heap level. That objects refer to each other by disk addresses when on-disk means that there is no need to maintain disk page tables to locate referenced objects on disk. A different approach to stabilising the store is required.

The basic idea underlying the implementation is that there are two shadow (or more correctly *mirror*) stores. Between stabilises, reads only take place from the current store. Writes to disk occur at stabilise – so at the start of stabilise, a switch is made to the non-current shadow store. Then the stable heap pages are written to that shadow. This then becomes the current store, and reads occur from that latest current store. On next stabilise, the store is switched again to what becomes the new shadow or mirror. In the case of a crash, the non-current store can be restored.
This shadow or mirror store has been ported to Arena only. There are two implementations of it: one with an object-granularity (Marvie, 1998), and one with a page-granularity, for writing to disk at stabilise. Naturally, the latter is the more efficient of the two[13].

13.1 Object key to object in stable heap mapping

The stable heap uses the 'object slot' mechanism for object allocation (see section 9.2). The object key is an integer such that the first object has key 1, the second has key 2, etc.. The Indirection Table maps this key to the stable store virtual address of the object (i.e. not to an offset into the stable heap). If an object has

[13] The object-granularity stable store is non-compliant with the persistent store requirements. The nature of the interpreter-level Nstabilise() routine (section 5.1) assumes that individual objects are written to the stable store, but not to disk.

not yet been read into the stable heap, then its Indirection Table entry has a null stable heap address.

13.2 Stable heap to disk block mapping

In the case of the object-granularity implementation, the disk address maintained for each object in the Indirection table is intended to enable an individual object to be read in from, or written to, disk without causing the overwriting of other objects on the same stable heap page or disk sector. That is, the Indirection Table contains entries giving the precise location of the object on disk. In the page-granularity implemenation, the start sector of a 4K disk block (actually a set of eight contiguous 512b disk sectors) is kept in each Indirection Table entry, and the entire 4Kb stable heap page is read or written at the same time. The offset within the 4Kb page, which is also kept in each Indirection Table entry, can then be used to locate the required object within the stable heap page.
The point here is that the object-granularity stable heap is *not compliant* with the needs of the persistence mechanism of pbase. It allows individual objects to be read and written between stable heap and disk. This is finer granularity than is required – the interpreter checkpoint only requires that entire pages are read and written between stable heap and disk.

13.2.1 Logical and actual disk blocks

The block number in the Indirection Table is treated as a logical disk block and is the same regardless of which of the two shadow, or mirror, stores is actually being written to. At startup, the Disk Manager of the POSM is given the actual *physical* disk start sector and number of disk sectors. The Disk Manager can then determine where to place the two shadow, stores *in terms of actual sectors.* The Disk Manager "remembers" these sectors of the physical disk, so it can switch between shadow stores. It does this by loading the start_sector into the disk device driver and so determining what start_sector value is added to the logical block value sent down in read/writes. This loading is done by a modified call to *hwo_posm_diskinfo().* The rest of the POSM still uses logical blocks.

13.3 Shadow store stable store page state

13.3.1 object-granularity

In this implementation, objects, rather than pages, are read and written between stable store and disk (Marvie, 1998). Thus, no page state, as previously described in sections 11.1.1 and 12.1.1, is maintained by the system. Pages are simply considered either as being free or as being used for object allocation (see section 9.2). At stabilise, all objects in the Indirection Table are *individually* flushed to disk.

13.3.2 page-granularity

Here, in addition to being either used or free, each page has two modified flags. One modified flag is not enough for the shadow store implementation. This is because each shadow store must be complete in itself, so that there is a need, in a sense, to write the entire stable store to each shadow store. However, more strictly, it is only necessary to write pages, to store A, that have been modified since the last write to store A. Whatever has happened with respect to store B is irrelevant. Two modified flags associated with each stable store page are used to implement this. Both flags are set when a page is modified. When the page is written to a shadow store, only the flag corresponding to that shadow is unset.

Stable store pages are contiguous, so the *keyToAddress()*-based implementation of *SH_read_words()* and *SH_write_words()* is applicable.

The key in this stable heap is an integer object identifier stored in the Indirection Table, rather than an offset into the stable heap (see sections 9.1 and 9.2). Since *readWords()* and *writeWords()* simply read and write stable heap locations, they are inappropriate where a key does not directly identify a location. Thus in this stable store, these routines are not used to implement *SH_read_words()* and *SH_write_words()*.

keyToAddress() also has the function of causing stable heap pages to be read in from disk when necessary. When an object is read in from disk, the entire stable store page is read from the required disk block, and the disk block becomes associated with the stable store page. However, only the required object is processed[14] An object which has not yet been read in from disk has a zero heap address in its Indirection Table entry, and there exists no used page associated with the disk address of the object. Where such an object is to be read, its entire page is then read in from disk.

On stabilise, the used list of pages is traversed, and those which have been modified, with respect to the current stable store, are written to disk (see Section 13.2.1 for the translation of logical to actual blocks).

SH_can_modify() and *saveVM()* are not used in this stable store, since it is assumed that there is an existing shadow of any page, and that the system has sufficient disk blocks to shadow any page.

shst_pggr summary Key is not an address, it is an object identifier.

- *keyToAddress()* – job of reading in pages from disk if va in Indirection Table == 0, and corresponding disk block not in used_pages. Set page to be *modified*.

- *stabilise()* – if a used_page is *modified* then write it to disk.

[14] An object read in from disk contains references to child objects in the form of disk addresses. These are replaced by newly- allocated object identifiers, and a new Indirection Table entry made for each child mapping its object identifier to its disk address, and having a zero stable heap address. Nothing further happens to the child objects. Even if a child resides on the page just read in, it remains unknown beyond its initial Indirection Table entry.

14 Garbage collection of the stable heap

Garbage collection occurs in the local heap on a failed heap allocation request. A global GC flag is set and the thread that issued the request takes responsibility for collecting the interpreter local heap. No further execution of interpreter instructions by any thread occurs until garbage collection is completed.

With the moving pointer stable heap (section 9.1), garbage collection appears to occur in a similar manner – a call is made to *SH_garbage_collect()* [15] when the stable heap runs out of space. It has not yet been necessary, for the purposes of performance comparisons using the test program, to implement *SH_garbage_collect()* in Arena **aipt**, **bisb** or **binb**.

However, the object slot stable heap (section 7.2) allows garbage collection of the stable heap to be carried out at the same time as garbage collection of the local interpreter heap. This is possible because object slots can be easily moved to and from the slot free list. When an object in the local heap is being garbaged, a call, *SH_destroy_object()*, is made to the stable heap. This simply puts the stable heap space occupied by the object back on to the appropriate free list, preventing garbage accumulating in the stable heap. A free page is put back on the page free-list. The execution cost of this was found to be negligible. The disadvantages of this are that

1. no compaction of the store occurs, so that objects might become sparsely distributed over the stable heap, and so the stabilise operation might be expected to take longer.

2. it is conceivable that small, long-lived objects will occupy object slots and cause wastage of store which cannot be garbage collected. However, some more space-efficient object-slot mechanism could be found.

The **shst_pggr** store used in the performance measurements does this simultaneous garbage collection of both local and stable heaps.

15 Performance measurements

A pbase test program was used which created four threads. These threads each created numbers of large vector objects such that garbage collection of the local heap occurred regularly. The application periodically called for a checkpoint and any vector objects existing in the local heap were written to disk. Performance comparisons were made between Arena and Linux for **aipt** and **bisb** stores.

Although it was intended that Arena pbase programs would use cooperative scheduling, the Linux interpreter implementation only supports time-slicing

[15] Source code note: From the heap.c routine Pdiskgc().

of threads. In order to allow performance comparisons, time-slicing was implemented in the Arena pbase-compliant PM.

The source code for both the Linux and Arena interpreter and persistent stores was compiled with (different versions of) gcc using level two optimisation. The Arena HWO was compiled with level one optimisation. Linux makefiles appear to use level two optimisation other than for architecture-specific files, which are level one. The Linux stable store source is written entirely in C, whereas the Arena stable store code is written in C++.

On Linux, timings used the *times()* call, which returns the number of clock ticks that have elapsed since the system booted. The number of such clock ticks per second is given by either calling *sysconf*(_SC_CLK_TCK) or by the CLK_TCK macro. Both gave the number of ticks per second as 100, so each clock tick represented 10 milliseconds.

On Arena there are two clocks available: from calls to *hwo_gethighrestime()* and *hwo_gettime()*. *hwo_gethighrestime()* returns the number of high-resolution ticks and has a resolution of 838 nanoseconds, but is only suitable for measuring short duration events. *hwo_gettime()* returns the number of timer interrupt ticks, and has a resolution of 54926 microseconds.

The total number of pbase instructions executed during the run of the program, by both Arena and Linux, was within the same range (number of instructions executed varied bewteen 275008134 and 275008184).

15.1 Runtime and stabilise duration

The main results are given in Figure 10. The most obvious finding is that overall runtime is determined largely by the time spent doing stabilise operations. If this stabilise time is discounted, the run durations for *all* stores, on both Arena and Linux, is very similar at between 48.17 and 48.52 seconds. On the whole, this is to be expected, as the applications are mainly sitting in the interpreter execute-loop performing CPU-bound activity. However, garbage collection does cause disk I/O and appears to introduce variation into the run times (see below).

The main interest, from the point of view of compliance, is to compare the performance of Arena **aipt** and **bisb** ports with those on Linux. As mentioned in the previous paragraph, differences in runtime are largely determined by stabilise times. And so, although Arena is faster overall than Linux, the differences reside in the stabilise durations. For **aipt**, Linux is slower than Arena by a factor of 1.8. For **bisb**, Linux is 5.5 times slower than Arena.

15.2 Disk I/0 in Arena and Linux

Where does the difference in stabilise duration reside? The most obvious factor is the efficiency of accessing a disk via the Unix file system as compared to accessing a disk directly as in Arena. However, there is a great problem in comparing the

store impl	total runtime (secs)	total stabilise time (secs)	runtime minus stabilises (secs)	time for 1 stabilise (ms)	sectors written		sectors read		I/O during stabilises sectors/sec
					during stabilises	outside stabilises	during stabilises	outside stabilises	
Arena									
aipt	53.39	4.87	48.52	304	33012	4592	0	0	6787
shst_pggr	55.80	7.63	48.17	477	47326	11	17	0	6208
binb	55.99	7.55	48.44	472	49936	3640	0	0	6613
bisb	70.48	22.08	48.44	1380	67068	3654	47985	2681	5212
shst_obgr	187.37	139.14	48.23	8696	87186	4	7125	1	678
Linux									
aipt	57.33	8.97	48.36	560	40064	4160*	0?	0?	4467
bisb	168.79	120.60	48.18	7540	69456	3088*	42880	2800	943

Figure 10: Results for the pbase application running on Arena and Linux using a variety of stable store implementations. Figures are mean values of several runs. In each run there were always 16 stabilise operations. Linux stores write in units of 8K, but these are multipled by 16 to give 512-byte disk sector equivalents. Disk reads for the Linux **aipt** store cannot be easily detected as they occur implicitly as a result of faulting on an mmap()ed region, but it is inferred from the Arena result that disk reads on Linux are zero. Sectors written outside stabilise on Linux, indicated by *, do not represent actual disk I/O (see text).

disk I/O performance in Arena and Linux. In Arena, a call to *hwo_write_blocks()* immediately results in a write to the disk. In contrast, Linux disk I/O is buffered and writes do not necessarily reach the disk until the file buffer is explicitly flushed. Thus, in Figure 10, most of the sectors "written to disk" outside the stabilise operations for the Linux **aipt** and **bisb** stores do not result in immediate disk activity. Only during the stabilise operations does disk I/O definitely occur in Linux – because here the **aipt** store issues *msync()* and the bstore issues *fsync()* calls to ensure the operating system flushes its buffers to disk.

During stabilise, when disk I/O does occur, Arena **aipt** writes 1.5 times more sectors per second than Linux **aipt**. Similarly, with **bisb**, during stabilise Arena writes 5.5 times more sectors per second than Linux – the reason for this large difference is discussed below (in section 15.5).

15.3 Effects of garbage collection

The bulk of the writes which occur outside the stabilise operations is found to be due to the effects of garbage collection. During garbage collection the TCBs and interpreter thread stacks (section 2.1) may need to be shadowed. *SH_can_modify()* (section 6) is called, and this may, depending on the state of the store pages (sections 11.1.1 and 12.1.1), result in a call to *shadow_copy()* (section 7), eventually causing a block to be "written to disk". However, on Linux, this write does not go through to the disk. The number of store pages needing to be shadowed at garbage collection depends on the sequence of stabilise operations and garbage collections. Stabilise causes pages to have state PROTECT and so the *SH_can_modify()* call in the garbage collection code causes shadowing of these pages. Subsequent garbage collections before the next stabilise will not cause shadowing of pages, and will result in no blocks being written.

The time taken during garbage collection is determined largely by the number of blocks actually written to disk. This is demonstrated in Figure 11. An average garbage collection on Linux **aipt** and **bisb** stores takes 32.24 ms and 30.58 ms respectively, but here no disk I/O actually occurs. If an *msync()* call (for **aipt**) and an *fsync()* call (for **bisb**) is added to cause writes to go through to disk, the average garbage collection time obviously increases, but especially so for the **bisb** store.

Figure 11 also shows the garbage collection performance on Arena. Here disk activity does occur as a result of the *shadow_copy()* calls. However, if *shadow_copy()* is altered (with **aipt**) so as to do cause no writes to disk, the garbage collection duration is very similar to that on Linux. However, these average values for garbage collection duration mask the fact that garbage collection times generally have one of two values when disk I/O is able to occur. The high resolution timer on Arena allows a closer breakdown of garbage collection performance. The pass through the heap to mark objects during garbage collecting takes a consistent 18 ms. Other activities vary between 8 and 12 ms.

store impl	total garbage collection time (ms)	total sectors written	number of collections	time per collection (ms)	total bytes reclaimed (10^6)
Arena					
aipt (no disk I/O)	829	23	27	30.71	95.1
aipt (disk I/O)	1319	3301	26.5	49.78	95.2
bisb (disk I/O)	1676	3229	26	64.89	95.5
Linux					
aipt (no disk I/O)	910	0?	28	32.24	95.2
aipt (disk I/O)	1400	2916	26	50.18	94.4
bisb (no disk I/O)	795	0?	26	30.58	94.4
bisb (disk I/O)	6630	2832	26	255.00	95.6

Figure 11: Garbage collection times and disk writes for Arena and Linux **aipt** and **bisb** stores. Garbage collection duration is largely determined by disk I/O. With Linux, there is no disk I/O unless calls are made to msync() or fsync(). In Arena disk I/O always occurs.

However the garbage collection component (a routine named *save_data()*) which may cause disk I/O shows two distinct durations. When no disk I/O occurs, *save_data()* lasts an insignificant 5 microseconds and garbage collection takes 26 ms for **bisb** and 27 ms for **aipt**. However, when *save_data()* causes writing to disk, *save_data()* lasts for 58 ms of a garbage collection time of 88 ms for **bisb**, and 33 ms out of 63 ms garbage collection duration for **aipt**.

15.3.1 Sector write times

It is of interest to examine the costs of writing to disk during *shadow_copy()* in Linux and Arena **aipt** stores. Linux writes to disk by "re-*mmap()*ing" a store page to some new "block" in the store file, and *memcopy()*ing the contents of the page so it is written to the file via the *mmap()* mechanism. Adding an *msync()* call ensures that such writes actually go through to the disk. Timing such writes[16] shows that the time for a 512 byte chunk (deemed equivalent to a 512 byte disk sector[17] write in *shadow_copy()* is 223 microseconds. On Arena the figure is 156 microseconds[18]. Adding an *fsync()* call for pages written by Linux **bisb** gives a per sector write time of 2778 microsconds.

15.4 Shadow copy and disk writes with aipt

It was noted that in Figure 10 the Linux **aipt** and **bisb** stores do not actually write through to disk between stabilise operations, whereas Arena stores do write to disk. In fact, this write to disk in *shadow_copy()*, at least in the **aipt** store, is not necessary. The page being shadowed already exists, in its pre-modified state, on the original disk block. It will be written, as modified, to the new block shadow at the next stabilise.

Figure 12 gives a comparison of results of running **aipt** with and without disk writes from *shadow_copy()* on both Linux and Arena. As expected, the overall execution time appears to decrease slightly.

15.5 Why Linux bisb is so slow

It has been mentioned that the stabilise times for the Linux **bisb** implementation seem excessively large, compared both to the Arena **bisb** implementation and to the Linux **aipt** store. The reason for these disparities appears to be related to the number of *fsync()* calls made by the Linux **bisb** implementation. This is particularly associated with the need in a before image implementation to write the root page (containing the beforeMap (see section 12) before a store page can

[16] The resolution of the timer does not allow individual writes to be timed, so 100 writes were timed in a loop

[17] Arena writes actual 512 byte sectors to disk, Linux writes 8 Kb "blocks" to the Unix file representing the disk

[18] During stabilise the **aipt** stores attempt to write blocks in contiguous groups. On Arena the time per sector write decreased when more sectors were written in a single write – a single call of *hwo_write_blocks()* for 160 sectors gave a per sector write time of 128 microseconds. This reduction from 156 microseconds to 128 microseconds may be due to having to take only one trap for the entire write.

store impl	total runtime (secs)	total stabilise time (secs)	runtime minus stabilises (secs)	time for 1 stabilise (ms)	sectors written		sectors read		I/O during stabilises sectors/sec
					during stabilises	outside stabilises	during stabilises	outside stabilises	
Arena									
shadow_copy									
no disk I/O	52.88	4.91	47.97	307	33406	704	0	0	6804
disk I/O	53.39	4.87	48.52	304	33012	4592	0	0	6787
Linux									
shadow_copy									
no disk I/O	57.33	8.97	48.36	560	40064	4160*	0?	0?	4467
disk I/O	58.16	9.30	48.86	581	39984	4164	0?	0?	4299

Figure 12: Performance with **aipt** on Arena and Linux when *shadow_copy()* causes, or does not cause, disk I/O. In Arena the default behaviour is to cause disk I/O. Linux, by default causes no disk I/O so *msync()* calls are added to ensure disk I/O occurs. The "sectors written" value marked with * do not all actually occur.

be modified). In a typical run, 3778 *fsync()* calls are made for an overall total of 4867 8K pages written (that is, 0.776 *fsync()* calls per 8K page). The duration of a single *fsync()* calls is 35 ms (measured by timing 100 *fsync()* calls). From Figure 10, the Linux **bisb** writes (69456/16) 4341 8K pages which, at a rate of 0.776 *fsync()* calls per page, means 3369 *fsync()* calls. At 35 ms per call, this gives a total estimated *fsync()* call time of 118 ms. This compares well with the measured 120.60 ms for the average stabilise time, and indicates the large overhead of the *fsync()* call. Thus, the Arena **bisb** store implementation, where there is no such *fsync()* overhead, is faster than the Linux by a factor of 5.5.

However, even the Arena **bisb** store is slow compared to the other Arena stores (excepting the non-compliant **shst_obgr**) (Fig 10). That this is probably due to the presence of the store buffer, rather than anything intrinsic in the before-image approach, is shown by the **binb** store implementation. Here the same before-image mechanism is used, but there is no store buffer. There is thus no time wasted searching for the actual virtual address of store page addresses, and no need to handle store buffer overflow (see section 12). Both Linux and Arena **bisb** implementations have disk reads occurring outside the stabilise operations. However, these are most likely to be associated with disk writes – due to overflow of the store buffer and subsequent reading back in from disk of previously overflowed buffer pages. It is likely that these overflow inefficiencies could be reduced by increasing the size of the store buffer.

15.6 object granularity disk I/O is non-compliant

The durations of typical stabilise operations with the mirror or shadow stores on Arena are given in Figure 10. The **shst_pggr** store, which has page-level granularity for disk I/O at the stable store level, has performance that is comparable with the Arena **aipt** and **binb** stores. Moreover, these figures include time for garbage collecting the stable store simultaneously with the local heap[19] (section 14).

The performance of the **shst_obgr** store, which has object-level granularity for disk I/O at the stable store level, has very poor performance (Fig 10). Its stabilise time is even worse than that of the Linux **bisb** store. This **shst_obgr** reads and writes to disk individual objects. This means that it must buffer all disk I/O so as not to over-write objects sharing the same disk sector. Even so, the number of reads carried out by the **shst_obgr** store is much less than the **bisb** stores. It is possible that the amount of copying between buffers required by the **shst_obgr** store is responsible for the poor performance of this store.

The flexibility of this **shst_obgr**, in allowing writing and reading of individual objects to and from disk, is not required by the interpreter. It does not meet the performance requirements of the interpreter, and is not compliant with it.

16 Conclusions

- The provision of compliant disk I/O by Arena means that:
  - the direct disk block I/O is more efficient than that provided by the general purpose file abstraction of Linux.
  - In that Arena disk I/O definitely occurs at the point that the interpreter makes the I/O request, the behaviour of the system is more comprehensible.
- The provision of thread scheduling by Arena means that:
  - the interpreter can be less complex, in that it does not have to concern itself with scheduling.
  - there is only one scheduler in the system – scheduling functionality is not replicated at levels in the system.
  - the thread scheduler can nevertheless be compliant with the scheduling requirements of the interpreter.

[19] This is only garbage collecting in the strict sense of freeing garbage such that it is able to be re-allocated. However the nature of the slot object allocation mechanism (section 9.2) means that store may be wasted within each slot.

- The provision of semaphores by Arena means that:
  - there is only one semaphore implementation in the system – functionality is not replicated at levels in the system.
- The ability to make upcalls as well as downcalls mean that:
  - the state of the interpreter can reflect the state of the Arena level beneath it.
  - special downcalls can be provided for restoration of Arena state in order to meet the persistency requirements of the interpreter.

17 References

Mayes, K.R. and Bridgland, J. (1997) Arena - A run-time operating system for parallel applications. Proceedings of Euromicro '97, 253-258.

Marvie, R.(1998) A distributed persistent object store manager for the Arena system. IUP GM13 report.

Morrison, R, Balasubramaniam, D, Greenwood, RM, Kirby, GNC, Mayes, K, Munro, DS, Warboys, BC (1999) ProcessBase Reference Manual. Technical Report, School of Mathematical and Computational Sciences, University of St Andrews.

Morrison, R, Balasubramaniam, D, Greenwood, RM, Kirby, GNC, Mayes, K, Munro, DS, Warboys, BC (2000) A Compliant Persistent Architecture. Software - Practice and Experience, Special Issue on Persistent Object Systems 30(4), 363-386.

Munro, D.(1994) On the Integration of Concurrency, Distribution and Persistence. University of St Andrews, Department of Mathematical and Computational Sciences Research report CS/94/1.