The Mungi Kernel API, Release 1.0

Gernot Heiser Jerry Vochteloo Kevin Elphinstone Stephen Russell

E-mail: {gernot,jerry,kevine,smr}@cse.unsw.edu.au
WWW: http://www.cse.unsw.edu.au/ ˜disy

UNSW-CSE-TR-9701 — April 1997

Abstract

This document describes release 1.0 of the application programming interface to the kernel of the *Mungi* single-address-space operating system. This interface will, in general, only be used by low-level software, most applications are expected to use a higher-level interface implemented as system libraries. Such libraries will be described in separate documents.

Department of Computer Systems
School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Contents

---

1 Introduction

Mungi [RSE+92, HERH93, VRH93, HERV94, ERHL96, VERH96] is a single-address-space operating system (SASOS) [CLFL94, WMR+97] developed by the Distributed Operating Systems Group at the University of New South Wales. It is conceptually similar to Angel [WSO+92, WM96] and Opal [CLBHL92, CLFL94], although quite different in many important aspects. Most notably, Mungi presents the single-address-space model in a very pure form, as it provides no inter-process-communication mechanisms other than shared memory. Many of the basic ideas in Mungi go back to the IBM System/38 [Ber80, HSH81] and its successor, the AS/400 [Sol96].

This document presents the application programming interface (API) of the Mungi kernel. While Mungi does not pretend to be a microkernel (in fact, the prototype is implemented on top of the L4 microkernel [Lie95]), it nevertheless presents a minimal and low-level interface to the programmer. Policies are, as far as possible, left to be implemented by higher software layers. Similar to microkernels, Mungi allows the implementation of device drivers and page fault handlers at user level.

As a consequence, application programs will not normally interface directly to the Mungi kernel, but are expected to call a higher level library interface. A UNIX-like interface has partially been developed, however this will be described in a separate document.

The following sections list and explain the Mungi system calls. These are presented in (hopefully) intuitive pseudocode. The actual C language bindings are presented in the Appendix.

2 Objects and Capabilities

Objects are Mungi's storage abstraction, they are the unit of virtual memory allocation and protection. Objects are page aligned and consist of an integral number of pages. Newly allocated objects are zero filled. From the system's point of view, an object is simply a contiguous, aligned region of virtual address space; the system imposes no structure on objects (but higher software levels are free to do so). Access to objects is controlled by *password capabilities* [APW86].

2.1 Password capabilities

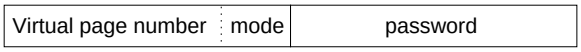

Figure 1: Format of a capability

Figure 1 shows the format of a capability. The virtual page number refers to the base address of an object. The mode indicates the rights conferred by the capability. This field is only a hint for the user, it is ignored by the system when a capability is presented. The system maintains a directory, the *object table* (OT), of object attributes, including the set of valid passwords and their corresponding modes.

There are five different rights capabilities may grant over an object: read (R), write (W), execute (X), delete (D), and *protection domain extension* (PDX); the latter will be explained in Sect. 3.2. Each valid capability grants the holder a combination of these rights to an object[1]. A capability granting RWXD

[1]Note that, as we are relying on the hardware to enforce protection, on many architectures we cannot guarantee that a user cannot read an object to which they only hold an X capability.

rights is, by definition, an owner capability. A capability containing a zero password is considered a valid capability granting no access rights.

Capabilities are user objects and can be stored and passed around freely. They are protected from forgery by sparsity, hence careful choice of passwords is important. The system provides a library routine to create random passwords.

As capabilities are user objects, it is not possible to determine who has access to a particular object. It is also normally impossible to prevent a particular user, who has been given a capability for an object, from handing this capability to other users. However, the *ObjDelPasswd* call allows revocation of a capability. Furthermore there exist a mechanism for confining clients of one's objects, see Sect 3.3.

type *ObjInfo* $\equiv$ {
  {*password*, *mode*}[*n_caps*],
  {*password*, *entry_pt*[*n_entry*]}[*n_pdx*],
  clist_cap,
  bank_acct_cap,
  pager_cap,
  extent,
  create_time, *modify_time*, *access_time*, *accounting_time*,
  user_info, *accounting_info*, ...
  is_acct, *sharable*, *persistent*, ...,
}

Figure 2: Object descriptor (*info*) data structure.

2.2 Object descriptors

Figure 2 shows the format of an object descriptor which is used to describe object attributes. That descriptor format is used when inquiring or setting object attributes in the OT. The meaning of the fields is as follows:

password, mode: list of non-PDX passwords plus access rights conferred by each password.

password, entry_pt[]: list of passwords allowing PDX call, plus the set of entrypoints allowed for each password.

clist_cap: Clist capability for extending the APD when the object is invoked via the *PdxCall* system call (c.f. Sect. 3.2).

bank_acct_cap: R capability to the *bank account* to which storage costs for the object is charged. Bank accounts are the basis if Mungi's management of memory resources. The kernel only provides the mechanism by ensuring that a bank account is associated with every object, and by protecting bank accounts from forgery. The operation of the accounting system will be documented elsewhere.

pager_cap: PDX capability to the page fault handler for the object. Null if handled by the default pager.

extent: unit of backing store allocation. The system rounds this to an integer multiple of the page size (which may be the nearest integer multiple, or the nearest power-of-two multiple). The system may ignore changes to the original setting.

is_acct: flag indicating that the object can be used as a bank account. Bank account creation requires special privilege.

sharable: flag indicating that the object can be shared with other tasks. If set, the object may be accessed by any thread whose protection domain contains a valid capability to the object. If not set, the object may be unaccessible by others even if they hold a valid capability. Unshared objects may be cheaper to create and destroy. The system may use the initial setting of this flag as a hint to direct the allocation strategy. The flag can be turned on at any time. The system may ignore attempts to turn it off.

persistent: flag indicating that the object may outlive its creator. An object created with this flag off will be cleaned up by the system when the creator task exits. The flag can be turned on by an owner, however, this may have no effect if called from a task other than the creator. The flag can be turned off by an owner, in which case the object is marked for cleanup when the caller's task exits.

create_time, *modify_time*, *access_time*: timestamp of object creation, last modification, last access. The modification and access time stamps are automatically updated by the system on certain events.

accounting_time: timestamp used by the accounting system.

accounting_info: reserved for usage by accounting system.

user_info a user-defined field which may be used, e.g., for implementing a type system on top of the Mungi kernel. The system does not use this field.

2.3 System calls

System calls dealing with objects are summarised in Table 1 and explained below.

ObjCreate: allocates a new object of a specified *size*. The call returns an *owner capability*, the password part of which is supplied by the caller. The caller supplies an *ObjInfo* data structure which is used to initialise the object descriptor in the OT. The fields used are *bank_acct_cap*, *extent*, *sharable*, *persistent*, *user_info*, and *is_account*, other entries are ignored. Setting the *is_account* flag requires R capability to the OT.

The system may use some of the data in *info* as a hint to optimise its allocation strategy.

Initially, the page faults on the object are handled by the default pager, which initialises pages to zero.

ObjDelete: deallocates an object. The caller must supply a delete capability. Virtual memory previously allocated to that object may be reused for objects created in the future, hence it is important that passwords cannot be guessed.

ObjResize: expands or shrinks the object to *new_size*. An object may not be able to grow if the following address space has already been allocated to a different object. In this case, the *may_move* parameter determines whether the system copies the object to a different location or returns a *cannot_grow* error. A successful operation returns a new owner capability. This will be equal to the capability supplied in the call, unless the object was moved. In the latter case, the new capability will be constructed from the new base address and the old owner capability's password.

System calls:		
ObjCreate	(*size, password, info*)	→ (owner_cap)
	raises *zero_size, no_memory*	
ObjDelete	(*cap*)	
	raises *invalid_cap*	
ObjResize	(*owner_cap, new_size, may_move*)	→ (owner_cap)
	raises *invalid_cap, zero_size, cannot_grow*	
ObjNewPager	(*owner_cap, pager_cap*)	
	raises *invalid_cap, apd_locked*	
ObjCrePasswd	(*owner_cap, password, mode*)	→ (cap)
	raises *invalid_cap, overflow, apd_locked*	
ObjCrePdx	(*owner_cap, password, clist_cap, n_entries, entry, ...*)	→ (cap)
	raises *invalid_cap, overflow, apd_locked, no_pdx*	
ObjDelPasswd	(*owner_cap, password*)	
	raises *invalid_cap*	
ObjInfo	(*obj_cap, OT_entry, update_flags*)	→ (*OT_entry*)
	raises *invalid_cap*	

Table 1: System calls dealing with objects and capabilities.

ObjNewPager: registers a new page fault handler for the object. A null *pager_cap* reverts to the system's default pager. Executing this system call implies calling *PageUnmap(base,length,zero)* on the object. See Sect. 5 for details.

ObjCrePasswd: registers a new capability for an object. The *mode* parameter specifies the strength of that capability. If *mode* is RWXD, the new capability is an owner capability, conferring the same rights as the original owner capability. The PDX bit, if set in *mode*, is ignored.

ObjCrePdx: registers a new PDX capability for an object, together with the protection domain extension (one Clist) and a list of valid entry points for which that password can be used. If a PDX password for this object has been registered before, *clist_cap* will overwrite any protection domain extension previously registered, except if null, in which case the protection domain extension remains unchanged. The call fails if no protection domain extension is registered at all, i.e. if an attempt is made to register the first PDX password with *clist_cap* being null.

PDX procedures are described further in Sect. 3.2.

ObjDelPasswd: de-registers (revokes) a password. The password will then no longer grant any rights over the object. Due to caching of validation data, revocation may not have immediate effect, however there is a maximum delay after which a revocation is guaranteed to be effective.

ObjInfo: updates the object descriptor from the OT. The system ignores all passwords as well as fields *is_acct* and *pager* and may ignore *extent*. Returns the previous value of the object descriptor. Null values on input mean no change.

Only an owner may use the call to change the bank account for the object (the kernel will validate the bank account capability).

Time stamps may be updated by setting an appropriate bit in *update flags*. The table below gives the minimum amount of privilege required for the various operations possible. Here *touch* means setting a time stamp to the present time and date, while *set* means setting to an arbitrary time and date. The *R-OT* privilege means that a R capability for the OT is required (note that this implies RWXD capability on any object). As usual, *RWXD* stand for an owner capability, *W* is a capability which allows write access, while *R/X* means a capability allowing read *or* execute access. The creation timestamp cannot be changed.

Time stamp	*min. privilege for*	
	touch	*set*
modification	W	RWXD
access	R/X	RWXD
accounting	R-OT	R-OT

The calls *ObjNewPager*, *ObjCrePasswd* and *ObjCrePdx* are not allowed when the task's *APD* is *locked* (c.f. Sect 3.3).

3 Protection Domains

A task's *protection domain* is the task's set of access rights to objects. In a capability system, this is equivalent to a set of capabilities.

Mungi's protection system has been designed to be unintrusive, hiding its operation from applications as far as possible. For this reason, Mungi does not expect capabilities to be presented explicitly when an object is accessed. Instead, users store their capabilities in a datastructure which is searched by the kernel when an access validation needs to be performed. This datastructure is called a task's *active protection domain* (APD).

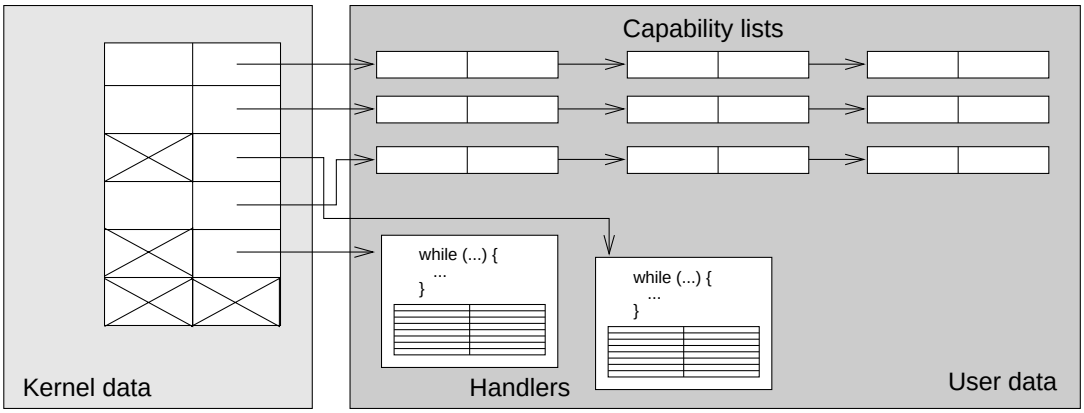

Figure 3: Active protection domain

3.1 Active protection domains

The APD is represented as a list (kept in kernel space) containing capabilities to *capability lists* (Clists) and pointers to *capability handlers*, as shown in Fig. 3.

Clists are user-maintained datastructures in a standard format. Users can add or remove capabilities in their Clists at any time without system intervention. Capability handlers are user functions which are up-called by the kernel. Clist capabilities and capability handler pointers can be added or removed via system calls.

Validation of an access is normally performed by the system in response to a protection fault, i.e. an object was accessed for which the kernel does not have information on the validity of the access, or that information is inconsistent with the mode of the access. In order to perform the validation, the system searches the OT with the faulting address. If no matching object is found, a segmentation fault is signalled to the user thread. Otherwise, the APD is searched for a capability matching one of those in the OT with appropriate mode. If found, a mapping for the object is established and the validation information is cached by the kernel to avoid having to validate each page of a large object individually. The basic algorithm is shown in Fig. 4.

```
funct validate(fault_adr, mode) ≡
  if ¬OT_lookup(fault_adr) → (base, limit, caps) then
    raise segmentation_fault;
  fi;
  entry := APD.head;
  while entry ≠ null do
      if entry.is_handler ∧ ¬APD_locked then
        cap := upcall(entry.adr, base, mode);
        if cap ∈ caps ∧ mode ⊆ cap.mode then
          return (base, limit, cap.mode);
        fi;
      elsif entry.is_clist then
        for c ∈ entry do
          if cap ∈ caps ∧ mode ⊆ cap.mode then
            return (base, limit, cap.mode);
          fi;
        od;
      fi;
      entry := entry.next;
  od;
  raise prot_fault.
```

Figure 4: Access validation algorithm.

As the validation algorithm shows, the search is terminated when the first capability of sufficient strength is found. Users can use this to arrange their capabilities such as to avoid double validation faults.

Presently there are two Clist formats, an unsorted one and one where capabilities are sorted by object base address. The kernel uses binary search on sorted Clists.

Any invalid capabilities encountered while searching the APD are ignored. This avoids race conditions with newly created capabilities, and synchronisation problems problems if validation occurs concurrently with Clist updates. Searching a (due to updates) inconsistent “sorted” Clist may fail to locate an existing capability. Users are therefore responsible for setting the format indicator to “unsorted”

before adding or removing capabilities in a Clist.[2]

Caching of validations could delay the effect of revocations of passwords indefinitely. To prevent this, the system guarantees that no validations are cached for more than a certain amount of time. Clist capabilities in the APD are also periodically revalidated, if such a capability is found to have become invalid it is silently removed from the APD.

3.2 Protected procedure calls

Mungi provides a protected procedure call mechanism similar to the *profile adoption* mechanism of the IBM System/38 [Ber80]. Mungi's mechanism, called *protection domain extension* (PDX) allows the caller of a PDX procedure to extend its protection domain, for the duration of the call, by the protection domain of the callee.

More specific, a PDX procedure has, in the object table, registered a set of valid entry points and a capability for a Clist (c.f. Sect. 2.2). When a *PdxCall* system call is executed, the system first verifies that the caller possesses a valid PDX capability and tries to access a valid entry point, then extends the caller's APD by the PDX's Clist, and finally transfers control to the PDX code. When the PDX procedure returns, the PDX Clist (and all cached validation information relating to that Clist) is removed from the caller's APD. Note that during the duration of the PDX call, the calling thread executes in a protection domain different from other threads of the same task, i.e. other threads have no access to the called object (unless they also perform a PDX call to the same object).

Instead of having the PDX procedure execute in a superset of the caller's protection domain, the caller has the option of explicitly supplying a protection domain when calling the PDX procedure. In this case, the call executes in a protection domain which is the union of the supplied one with that registered for the PDX object. This gives the caller maximum control over which objects the PDX procedure can access. In particular, an empty protection domain may be passed to the PDX procedure, the latter then has no access to any of the caller's data (other than by-value parameters).

3.3 Confinement

Note that, as the APD data structure contains actual capabilities for Clists, there is no need for Clists capabilities to be contained in any of a task's Clists. As Clist capabilities are immediately validated when added to the APD, the system can rely on all Clists referenced by the APD to be in the user's protection domain. A task holding no read capability to any of its Clists can use the Clists to access objects, but cannot look at the capabilities contained in the Clists.

Applications can make use of this to *confine* untrusted libraries. Mungi provides the possibility of *locking* a task's APD, which means that no Clist capabilities can be added. When performing an access validation in a locked APD, all capability handlers will be ignored and only Clists searched. A locked APD also prevents execution of PDX calls. A task with a locked APD, and with no write capabilities to objects readable by others, is unable to betray any of the data it has access to.

The system libraries contain a procedure which, using one-way functions, generates reduced-strength capabilities from a given capability. While usage of this method is not enforced by the system, using it makes it easy to construct Clists containing only read-only (or execute-only) capabilities

[2] Note that this scheme cannot completely prevent such failures. However, we expect that sorted Clists are modified very infrequently. The extremely rare chance of a transient failure to locate a capability does not seem to justify the expense of proper concurrency control. Paranoid users can, of course, replace the whole Clist, which can be done without ever leaving an inconsistent Clist in the APD.

to globally used objects, without requiring the owners to distribute a whole set of capabilities for each object.

3.4 System calls

APD operations are summarised in Table 2. These calls work as follows:

System calls:		
ApdInsert	(*pos, handler_ptr_or_clist_cap, is_handler*)	
	raises *invalid_cap, inv_pos, overflow, apd_locked*	
ApdDelete	(*pos*)	
	raises *inv_pos*	
ApdGet	()	→ ({*ptr,is_handler*}[n_pd], locked)
ApdFlush	()	
ApdLookup	(*address, mode*)	→ (*cap*)
ApdLock	()	
PdxCall	(*adr, n_pd, . . . , n_args, . . .*)	→ (*any*)
	raises *invalid_cap, invalid_entry, apd_locked*	

Capability handler signature:		
cap_handler	(*obj_adr, mode*)	→ (*obj_cap*)
	raises *upcall_failed*	

Table 2: System calls dealing with protection domains.

ApdInsert: Insert a new Clist capability or capability handler pointer into the APD at index *pos*. If *is_handler* is TRUE, a capability handler pointer is inserted, otherwise a Clist capability. Clist capabilities are immediately validated. It is the user's responsibility to ensure that when a capability handler is installed, capabilities to the handler's code and all objects accessed by the handler are contained in Clists or handlers "above" the new handler. Otherwise a protection fault can lead to an infinite loop of handler invocations, from which no recovery is possible other than killing the faulting thread or task.

The previous entry at position *pos*, as well as all further ones, are shifted down one position. This call will fail on a locked APD.

ApdDelete: Remove a Clist capability or capability handler pointer from the APD. Following entries will be shifted upwards. Due to validation caching, this will not immediately make the Clist's objects inaccessible. However, cached validations are guaranteed to be invalidated after a specific time interval.

ApdFlush: Flush the task's validation cache immediately and re-validate all Clist capabilities in the APD.

ApdGet: Returns the list of Clist and capability handler pointers in the APD, as well as the APD locking status. For Clist capabilities, only the address part (i.e. the pointer to the Clist object) is returned, passwords are not returned.

ApdLookup: Performs an explicit validation of an access of type *mode* to *address*. If successful, returns the first capability granting (at least) the requested rights to the specified address, and caches the validation, otherwise returns NULL. If the APD is locked, the password part of the returned capability is zeroed.

ApdLock: Lock the APD. The task will no longer be able to perform *ApdInsert*, *ObjNewPager*, *ObjCrePasswd*, *ObjCrePdx*, or *PdxCall* operations. No capability handlers will be invoked during an access validation. A locked APD cannot be unlocked.

PdxCall: Call a PDX object via entrypoint *adr*, passing the specified arguments. The call will execute in an APD which is the union of the domain passed via the *n_pd, ...* parameters, and the domain registered for the PDX object. The parameters have the format of capabilities, if the password part is zero, they are assumed to be addresses of capability handlers.

If *n_pd* is zero, the caller's APD is extended by the PDX domain to form the APD of the call. If *n_pd* is negative, an empty protection domain is passed to the call, and the call executes just in the protection domain registered for the PDX object. The call will fail if the caller's APD is locked.

The PDX procedure returns via a normal function return. It may return a value, which is the value returned by *PdxCall*. While the set of permitted return values is implementation dependent, it is always possible to return a numeric value or a capability.

If the PDX procedure does not return, but instead exits (by an explicit *ThreadDelete(myself,...)* or a fault), the calling thread is killed. If the PDX procedure creates new threads or tasks, and does not kill them prior to returning, these may or may not survive the PDX call. If they survive the return of the PDX call, they may be killed by the system at any time later on.

4 Tasks and Threads

4.1 Process model

Threads are the basic execution abstraction, they are kernel scheduled. The set of threads sharing a protection domain form a *task*. Each task (or protection domain) therefore has at least one thread, tasks or protection domains cannot exist independently of threads.

Creating a new thread (within the same task) is a very lightweight operation. The new thread's stack is supplied by the caller, to reduce the cost of creating threads which need no, or only a very small stack.

Creating a new task is significantly more heavyweight than creating a thread. A new APD, stack, and environment must be set up. The APD may be either explicitly supplied by the creator or the new task inherits a copy of the creator's APD.

The task's first thread's stack object is allocated by the system as part of task creation. This object is created with the *sharable* and *persistent* flags both off, i.e. it is, by default, not accessible by other tasks and will be cleaned up when the task exits. (This can be changed using the *ObjSetInfo* call.) A thread can use *ApdLookup* to obtain the capability to its own stack.

The system also allocates an *environment*, which may be in a separate object or in the in the first thread's stack object. The first word of the environment must contain its length, to allow the kernel to copy the environment. The address of the environment is passed to the task's first thread as a parameter. Otherwise, the environment is completely under user control, the system does not care about its contents. It may be used to store things like bank accounts, address of a directory service, etc., which are typically inherited from the parent.

No heap (data) object is set up for a task by the kernel. A heap object for a task can be allocated by the run-time library on demand, and the heap's address can be stored in the environment or the first Clist (see the description of *TaskCreate*).

There is a hierarchy of tasks: A task T_1 created by another task T_0 cannot survive T_0, unless it is adopted by a task higher up in the hierarchy. Hence, killing a task kills the whole hierarchy of tasks created by it, unless children are adopted. Only tasks in the caller's hierarchy (i.e. children or more remote offspring) can be killed.

4.2 System calls

The system interface dealing with threads and tasks is given in Table 3. The system calls work as follows:

System calls:		
ThreadCreate	(*start_adr, stack*)	→ (*thread_id*)
	raises *prot_fault, no_threads*	
ThreadDelete	(*thread_id, status*)	
	raises *invalid_thread*	
ThreadSleep	(*thread_id, time*)	
	raises *invalid_thread*	
ThreadResume	(*thread_id*)	
	raises *invalid_thread*	
ThreadWait	(*thread_id*)	→ (*thread_id, status*)
	raises *invalid_thread, wait_deadlock*	
ThreadMyId	()	→ (*thread_id*)
TreadInfo	(*thread_id, prio, cpu_lim, mem_lim*)	→ (*thread_info*)
TaskCreate	(*start_adr, stack_size, n_pd, . . . , env, n_args, . . .*)	→ (*task_id*)
	raises *no_tasks, apd_locked*	
TaskDelete	(*task_id, status, adopt*)	
	raises *invalid_task*	
TaskWait	(*task_id*)	→ (*task_id, status*)
	raises *invalid_task*	
TaskMyId	()	→ (*task_id*)
TaskInfo	(*task_id, prio, cpu_lim, mem_lim*)	→ (*task_info*)

Table 3: System calls dealing with tasks and threads.

ThreadCreate: Creates a new thread which starts to execute at *start_adr* and uses the supplied stack. It is the caller's responsibility to set up a stack frame for the new thread, and to ensure that the stack is big enough. Parameters to the thread can be passed directly on that stack. A null pointer may be passed as *stack* if the thread needs no stack.

ThreadDelete: Kills the specified thread, which must belong to the same task as the caller. The *status* value will be returned to a thread waiting for the target thread to exit. A *thread_id* of *myself* kills the caller.

ThreadSleep: Stops a thread, which must belong to the same task as the caller, for a specified interval of real time, or until explicitly awaken by *ThreadResume*. A *thread_id* of *myself* stops the caller.

ThreadResume: Resume a stopped thread, which must belong to the same task as the caller.

ThreadWait: Wait until the specified thread, belonging to the caller's task, is killed. If *thread_id* is *any*, wait for any thread to be killed. Returns the *thread_id* of the thread that was killed, plus the status supplied when killing it. The call will return immediately, returning a *thread_id* of *any* and an undefined *status*, if there is nothing to wait for (i.e. waiting for a non-existent thread is not considered an error).

ThreadMyId: Return the ID of the calling thread.

TreadInfo: Sets attributes of the specified thread, which must belong to the same task as the caller, *myself* means the calling thread. Returns values of thread attributes before the call. Zero values for *prio*, *cpu_lim*, *mem_lim*, *thread_info* mean no limit. Limits can only be restricted, not relaxed by this call. Returns pre-call values of thread attributes via the *thread_info* parameter, whose format is shown in Fig. 5.

TaskCreate: Create a new task with the specified APD. If *n_pd* is zero, the child will receive a copy of the caller's APD, otherwise its APD is set up from the *n_pd* Clist capabilities or capability handler pointers specified in the call. These parameters have the format of capabilities, if the password part is zero, they are assumed to be addresses of capability handlers.

The new task will contain a single thread, which starts executing at the specified address. The system supplies the thread with a stack, which is at least as big as specified by the *stack size* parameter (a reasonable default size will be chosen if that parameter is zero). Any parameters are pushed on that stack. It is the caller's responsibility to ensure that the child receives at least an execute capability to the object containing the start address.

The remaining parameters are passed as arguments to the new task's first thread. The first one, *env*, is the *address* of a by-value parameter, the length of which is given in its first word. Further *n_args* parameters are passed by-value.

The system sets up a Clist holding owner capabilities for the system-supplied objects (stack, environment and Clist for these, all of which may actually be allocated in the same object). The capability for that Clist is entered in the first slot of the kernel's APD datastructure (c.f. Sect. 3.1) for the task. When a new task inherits its parent's APD, this first Clist is **not** passed on to the child. Hence, if the child is to have access to the parent's stack object, a capability for that object must be explicitly inserted in one of the other Clists. A task may insert further capabilities into the first Clist (e.g. the run-time system may put the capability to heap objects there). Like the system-supplied objects, these will not be automatically inherited by a child task.

If the caller's APD is locked, the child can only inherit the caller's APD (including its locking state), i.e. *n_pd* must be zero in that case.

TaskDelete: Kill the specified task, which must be one of the descendents of the caller's task. The *status* value will be returned to a thread waiting for the task to exit. A *task_id* of *myself* kills the caller. If *adopt* is true, any immediate children of the task to be killed become children of the caller's task. Otherwise, all descendants will be killed as well.

The target task is not killed immediately, instead it (and its descendants, if they are not adopted) receives a *task_shutdown* exception giving it a chance to clean up. If the task has registered a handler for this exception, this is called after setting the task's CPU time limit to a small amount. If no *task_shutdown* handler was registered, the task is killed immediately.

TaskWait: Wait until the specified task is killed (either by an explicit *TaskDelete*, or by all its threads being deleted). If *task_id* is *any*, wait for any task to be killed. Returns the *task_id* of the task that was killed, plus the status supplied when killing it. If the task finished because its last thread was killed, the status value of the task's last thread is returned.

The call will return immediately, returning a *task_id* of *any* and an undefined *status*, if there is nothing to wait for (i.e. waiting for a non-existent task is not considered an error).

TaskMyId: Return the ID of the calling thread's task.

TaskInfo: Sets attributes of the specified task, which must be one of the descendents of the caller's task, the caller's task if *task_id* is *myself*. Returns values of task attributes before the call. Zero values for *prio*, *cpu_lim*, *mem_lim*, *task_info* mean no limit. Limits can only be restricted, not relaxed by this call. Returns pre-call values of task attributes via the *task_info* parameter, whose format is similar to *thread_info*.

type *ThreadInfo* $\equiv$ {
thread_id, *task_id*,
mem_limit, *time_limit*,
start_time, *sys_time*, *user_time*,
priority
}

Figure 5: Thread (and task) descriptor data structure.

5 Page Fault Handlers and Virtual Memory Mappings

5.1 User-level page fault handlers

When an object is created, no backing store or physical memory is allocated initially. Hence, an access to any of the object's pages will lead to a page fault. Initially, such page faults are handled by the system's *default pager*, which will allocate a disk block for the accessed page, allocate a zero-filled physical frame, and enter the appropriate information into the faulting task's page table. Page replacement and residency faults are handled in the usual fashion.

Alternatively, tasks can register their own page fault handlers for particular objects, using the *ObjNewPager* call (see Table 1). This system call is passed a PDX capability for a pager object; a null capability re-instates the default pager.

An object's pager is called by the kernel whenever a page fault happens on the object. It is invoked by a *PdxCall* passing an empty protection domain (c.f. Sect. 3.4). Hence the pager executes in just the protection domain defined for it in its OT entry.

There are three kinds of page faults:

residency fault: an access failed because the page was not resident, the pager should establish a mapping for the faulting page (or indicate failure);

write fault: a write access was attempted on a read-only (R/O) page, the pager should establish a R/W mapping to the page (or indicate failure);

flush: a *PageFlush* operation was requested for a page; the pager should ensure that the page is clean (usually by forwarding the flush operation to the object the faulting object is mapped to).

5.2 Virtual memory mapping operations

The Mungi kernel interface does not provide for any explicit I/O operations. Instead, all devices are mapped into particular locations of virtual memory. The default pager performs I/O between physical memory and paging disk, and establishes mappings between virtual pages and physical frames.

User page fault handers, since they are normal user code, have no access to physical devices. Instead they use virtual memory operations to map one virtual memory (VM) region (the object whose page faults are handled by the pager) to another, which is handled by another pager. Eventually, the mapping chain must end at an object handled by the default pager.

Virtual memory mappings introduce *aliasing* — the same data (physical frame) is potentially visible at different virtual addresses. However, using aliases for accessing data is *strongly discouraged*: Mungi makes **absolutely no guarantee** about any consistency between data accessed via aliases. All accesses to data should *always* use the same virtual memory address.

5.3 System calls

Table 4 lists system calls dealing with virtual memory mappings. The meaning of the calls is as follows:

System calls:		
PageCopy	(*from_page, to_page, n_pages*)	
	raises *prot_fault, range_fault*	
PageMap	(*from_page, to_page, n_pages, mode, fault_in*)	
	raises *prot_fault, range_fault*	
PageUnmap	(*page, n_pages, disp*)	
	raises *prot_fault, range_fault*	
PageUnalias	(*page, n_pages*)	
	raises *prot_fault, range_fault*	
PageFlush	(*page, n_pages*)	
	raises *prot_fault, range_fault*	
Library calls:		
PageSize	(*obj_adr*)	$\rightarrow$ (*size*)
Pager signature:		
Pager	(*page, n_pages, fault_type*)	$\rightarrow$ (*success* $\|\|$ *fail*)

Table 4: System calls dealing with virtual memory mappings and call interface for user-level pagers.

PageCopy: Copy a range of pages, using copy-on-write (where possible). The two ranges must not overlap. Requires R capability on the source and W capability on the destination.

PageMap: Alias two page ranges. If *mode* is *read_only*, a subsequent write attempt on the range starting with *to_page* will result in a write fault. The two ranges must not overlap, unless they are identical and *mode* is *read_only*, in which case the call serves to turn on write protection on the range of pages. Unless the ranges are identical, this operation implies *PageUnmap(to_page,n_pages,zero)*.

The parameter *fault_in* determines what happens if some source pages are not resident. If *fault_in* is TRUE, such pages are forced to become resident (by invoking their pager). Otherwise, *PageMap* is a no-op where the source pages are not resident.

The operation requires RW capability on the destination. On the source, R capability is required if *mode* is *read_only*, otherwise RW capability is required.

The alias so established between two virtual pages vanishes as soon as either the source or destination page becomes non-resident for whatever reason (VM page replacement or explicit unmap). Therefore this operation is only of use for page fault handlers.

PageUnmap: Invalidate page mappings. For objects handled by a user-level pager, their mappings simply vanish, while for objects handled by the default pager, the semantics depend on the *disp* parameter. If *disp* is *zero*, the pages loose all association with physical memory or backing store, reverting them to the state they were in when the object was originally created by an *ObjCreate* call. If *disp* is *replace*, the virtual pages' association with any physical frames is lost, but any association with backing store is retained. However, dirty pages are **not** flushed to backing store, an explicit *PageFlush* call needs to be performed first if this is desired. If *disp* is *keep*, the pages' mappings are not lost at all, but are marked as invalid, forcing pager invocation on the next access. This can be used to force updating of time stamps on the next access.

RW capability is required, except for R/O mapped pages handled by a user-level pager, where R capability is sufficient.

PageUnalias: Remove all mappings **from** this page, i.e. unmap all destination pages which used the specified pages as the source of their mappings. This is a no-op for pages which have no aliases (including non-resident pages). Requires RW capability.

PageFlush: For pages handled by the default pager, ensures that the pages are cleaned (i.e. flushed to disk, if necessary), **and** that their association with backing store is recorded in stable storage. For pages handled by a user-level pager, this is simply translated into a pager invocation with a *fault_type* of *flush*. The operation is a no-op on clean or non-resident pages. Requires RW capability.

PageSize: A library call returning the page size for the object containing *page*. Different page sizes may be in use for different objects, if this is supported by the underlying hardware. The system may select a page size based on heuristics or hints, such as the object's *extent* value. Requires no access rights to the object.

Pager: The calling convention of a user-level pager.

In all these cases, where a range of pages is specified (i.e. $n\_pages > 1$), the whole range must belong to the same object.

6 Miscellaneous System Calls

6.1 Exceptions

Exceptions may be generated as a result of a program fault (e.g. division by zero, segmentation fault, protection violation). An exception handler can be registered to handle an event. If an exception occurs for which a handler had been registered, that handler is called as an un-programmed function call of the thread which caused the exception. To handle an exception which is no associated with a particular thread, such as *task_shutdown*, the system selects any thread.

6.2 Semaphores

Semaphores are used for synchronisation. A semaphore is identified by a byte in an object. The association of a semaphore with a particular address serves to name semaphores and to integrate them with Mungi's protection system. The contents of the byte addressed by the semaphore's name has nothing to do with the state of the semaphore, and it can be used independently of the semaphore.

6.3 Others

6.3.1 Timers

No system calls are necessary to access timers, instead they are mapped (R/O) into VM locations. There is a timer for date and time (with at least a second resolution) and, if supported by the hardware, a high-resolution timer (which may wrap around).

6.3.2 Profiling

There will eventually be system calls supporting profiling of code, but these have not yet been defined.

6.4 System calls

The system calls are listed in Table 5 and explained below.

ExcptReg: Register a handler for *exception*. Returns the address of the previously registered handler. NULL indicates no handler, i.e. the system will take a default action if the exception arises (usually the default action is to terminate the faulting thread).

ExcptRet: Return from an exception, continue at the point where the exception happened. In cases of exceptions caused by programming faults it may be better to recover using the *setjmp*, *longjmp* facility.

handler: Calling convention for an exception handler. A handler executes on the faulting thread's stack. Handlers can return via a library call *ExcptRet* to the point where the exception happened, or can use the *setjmp*, *longjmp* facility. If the handler exits, this will exit the faulting thread.

SemCreate: Create a semaphore named by the specified *address* and initialised to *value*. Requires R/W capability on the object containing *address*.

SemDelete: Delete a semaphore. Any threads waiting on the semaphore will be faulted. Requires R/W capability on the object containing *address*.

Asynchronous exceptions:
ExcptReg (*exception, handler_adr*) → (*old_handler*) **raises** *inv_exception*

Library calls:
ExcptRet ()

Exception handler signature:
handler (*exception, address*)

Semaphores:
SemCreate (*address, value*) **raises** *prot_fault, in_use* *SemDelete* (*address*) **raises** *prot_fault, inv_semaphore* *SemWait* (*address*) **raises** *prot_fault, inv_semaphore* *SemSignal* (*address*) **raises** *prot_fault, inv_semaphore*

Table 5: Miscellaneous system calls.

SemWait: Perform a *wait* operation on the specified semaphore, i.e. do atomically:

$$\textbf{while}\ sem \leq 0\ \textbf{do}\ \ \textbf{od};$$
$$sem := sem - 1;$$

The implementation does not use a busy wait. Requires R capability on the object containing *address*.

SemSignal: Perform a *signal* operation on the specified semaphore, i.e. do atomically:

$$sem := sem + 1;$$

Requires R capability on the object containing *address*.

References

[APW86] M. Anderson, Ronald Pose, and Chris S. Wallace. A password-capability system. *The Computer Journal*, 29:1–8, 1986.

[Ber80] Viktors Berstis. Security and protection in the IBM System/38. In *Proceedings of the 7th Symposium on Computer Architecture*, pages 245–250. ACM/IEEE, May 1980.

[CLBHL92] Jeff S. Chase, Hank M. Levy, Miche Baker-Harvey, and Edward D. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pages 80–85, Key Biscayne, FL, USA, 1992. IEEE.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12:271–307, November 1994.

[ERHL96] Kevin Elphinstone, Stephen Russell, Gernot Heiser, and Jochen Liedtke. Supporting persistent object systems in a single address space. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, NJ, USA, May 1996. Morgan Kaufmann.

[HERH93] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Graham R. Hellestrand. A distributed single address space system supporting persistence. School of Computer Science and Engineering Report 9302, University of NSW, University of NSW, Sydney 2052, Australia, March 1993.

[HERV94] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating system. In *Proceedings of the 17th Australasian Computer Science Conference*, pages 271–80, Christchurch, New Zealand, January 1994.

[HSH81] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Symposium on Computer Architecture*, pages 341–348. ACM/IEEE, May 1981.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[RSE+92] Stephen Russell, Alan Skea, Kevin Elphinstone, Gernot Heiser, Keith Burston, Ian Gorton, and Graham Hellestrand. Distribution + persistence = global virtual memory. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 96–99, Dourdan, France, September 1992. IEEE.

[Sol96] Frank G. Soltis. *Inside the AS/400*. Duke Press, Loveland, CO, USA, 1996.

[VERH96] Jerry Vochteloo, Kevin Elphinstone, Stephen Russell, and Gernot Heiser. Protection domain extensions in Mungi. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, pages 161–165, Seattle, WA, USA, October 1996. IEEE.

[VRH93] Jerry Vochteloo, Stephen Russell, and Gernot Heiser. Capability-based protection in the Mungi operating system. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, pages 108–15, Asheville, NC, USA, December 1993. IEEE.

[WM96] Tim Wilkinson and Kevin Murray. Evaluation of a distributed single address space operating system. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 494–501, Hong Kong, May 1996. IEEE.

[WMR+97] Tim Wilkinson, Kevin Murray, Stephen Russell, Gernot Heiser, and Jochen Liedtke. Single address space operating systems. In Nayeem Islam and Roy Campbell, editors, *Modern Operating System Research*. IEEE Computer Society Press, 1997.

[WSO+92] Tim Wilkinson, Tom Stiemerling, Peter E. Osmon, Ashley Saulsbury, and Paul Kelly. Angel: A proposed multiprocessor operating system kernel. In *European Workshop on Parallel Computing*, pages 316–319, Barcelona, Spain, 1992.

A C Language Bindings

A.1 mungi/types.h

```
/*************************************************************************\
.                                                                         .
.         mungi/types.h: Mungi kernel type definitions                    .
.         (c) Copyright 1994,95,96,97 by University of New South Wales    .
.                                                                         .
.         Author: Jerry Vochteloo and Gernot Heiser                       .
.         Distributed Systems Research Group                              .
.         Department of Computer Systems                                  .
.         School of Computer Science and Engineering                      .
.         The University of New South Wales                               .
.         Sydney 2052                                                     .
.         Australia                                                       .
.                                                                         .
.         For details please check http://www.cse.unsw.edu.au/~disy       .
.         or contact <disy@cse.unsw.edu.au>.                              .
.                                                                         .
.         Permission to make verbatim digital/hard copy of this           .
.         file for personal or classroom use is granted without fee       .
.         provided that copies are not made or distributed for profit     .
.         or commercial advantage. To copy otherwise requires prior       .
.         specific permission.                                            .
.                                                                         .
\*************************************************************************/

#ifndef _MUNGI_TYPES_H_
#define _MUNGI_TYPES_H_

/***********\
 * General *
\***********/

typedef enum { FALSE, TRUE }  bool;
typedef char                  int8;
typedef unsigned char         uint8;
typedef short                 int16;
typedef unsigned short        uint16;
typedef unsigned int          uint;
typedef int                   int32;
typedef unsigned int          uint32;
typedef long long             int64;
typedef unsigned long long    uint64;


typedef struct {
  uint64  passwd;
  void    *address;
} Cap_t;
```

```
/***********\
 * Objects *
\***********/

typedef int32 Access_t;
#define M_READ       ((Access_t) 1<<1)
#define M_WRITE      ((Access_t) 1<<2)
#define M_EXECUTE    ((Access_t) 1<<3)
#define M_PDX        ((Access_t) 1<<4)
#define M_DESTROY    ((Access_t) 1<<5)

#define O_MAX_CAPS      10
#define O_MAX_PDX       10
#define O_MAX_ENTPT     10

typedef struct {
  Access_t  rights;
  uint64    passwd;
} CapList_t;

typedef struct {
  uint64     passwd;
  int        n_entry;
  uint64     entry[O_MAX_ENTPT];
} PdxList_t;

typedef uint32 Date_t;
typedef uint32 Time_t;

typedef uint ObjFlags_t;
#define O_PERS         ((ObjFlags_t)0x001)    /* is persistent */
#define O_SHARE        ((ObjFlags_t)0x002)    /* is sharable */
#define O_ACCT         ((ObjFlags_t)0x004)    /* is a bank account */

typedef uint DateFlags_t;
#define O_SET_NONE      ((DateFlags_t)0x00)  /* don't change */
#define O_SET_MODIFY    ((DateFlags_t)0x01)  /* set modify time */
#define O_SET_ACCESS    ((DateFlags_t)0x02)  /* set access time */
#define O_SET_ACCNT     ((DateFlags_t)0x04)  /* set accounting time */
#define O_TCH_MODIFY    ((DateFlags_t)0x11)  /* touch modify time */
#define O_TCH_ACCESS    ((DateFlags_t)0x12)  /* touch access time */
#define O_TCH_ACCNT     ((DateFlags_t)0x14)  /* touch accounting time */

typedef struct {
  uint       n_caps;
  uint       n_pdx;
  CapList_t  capList[O_MAX_CAPS];
  PdxList_t  pdxList[O_MAX_PDX];
  Cap_t      clist;  /* for PDX */
  Cap_t      account;
  Cap_t      pager;
```

```
  uint       extent;
  Date_t     creation;
  Date_t     modification;
  Date_t     access;
  Date_t     accounting;
  uint64     userinfo;
  uint64     acctinfo;
  ObjFlags_t flags;
} ObjInfo_t;


/*******\
 * APD *
\*******/

#define A_MAX_APD      0x10

typedef struct {
  bool    locked;
  uint8   n_apd;
  uint16  is_handler;          /* boolean array */
  uint64  ptr[A_MAX_APD];      /* address of Clist or handler */
} ApdDesc_t;


typedef uint8 ClistFormat_t;
#define CL_UNSRT_0      ((ClistFormat_t)0x1)    /* unsorted format */
#define CL_SRT_0        ((ClistFormat_t)0x2)    /* sorted format */

typedef struct {
  char          type;          /* magic number 'c' */
  uint8         rel_major;     /* Presently 1 */
  uint8         rel_minor;     /* Presently 0 */
  ClistFormat_t format;
  uint32        n_caps;        /* size of Clist */
  uint32        max_caps;      /* max size of Clist */
  uint32        reserved;
  Cap_t         caps[];
} Clist_t;

typedef Cap_t (*CapHndlr_t)(void *, Access_t);  /* Capability handler */

typedef Cap_t (*Pdx_t)();                       /* PDX procedure */


/*********************\
 * Tasks and Threads *
\*********************/

typedef void (*Thread_t)();
typedef void (*Task_t)(void *env, int argc, char **argv);
```

```
typedef uint64 ThreadId_t;
typedef uint64 TaskId_t;
#define THREAD_MYSELF  ((ThreadId_t)-1)
#define THREAD_ANY     ((ThreadId_t)-1)
#define TASK_MYSELF    ((Task_t)-1)
#define TASK_ANY       ((Task_t)-1)


typedef struct {
  ThreadId_t self;
  TaskId_t   task;
  uint64     mem_limit;
  Time_t     time_limit;
  Time_t     start_time;
  Time_t     sys_time;
  Time_t     user_time;
  uint       prio;
} ThreadInfo_t;

typedef struct {
  TaskId_t   self;
  TaskId_t   parent;
  uint64     mem_limit;
  Time_t     time_limit;
  Time_t     start_time;
  Time_t     sys_time;
  Time_t     user_time;
  uint       prio;
} TaskInfo_t;


/* Kernel process table */

#define T_INFO_PAGE_ADR 0xa0080    /* fixed location of kernel proc info */
#define T_DEF_NAMER_ADR (T_INFO_PAGE_ADR+sizeof(Cap_t))

#define ProcTableCap \
        (* ((Cap_t *) T_INFO_PAGE_ADR))
/* Read capability to kernel's global table of tasks and threads.
 * Its format is not yet stable and will be defined in the future.
 */

#define DefaultNamerCap \
        (* ((Cap_t *) T_DEF_NAMER_ADR))
/* Execute capability to the system's default naming service.
 * This is essentially a directory pointing to individual user's namers.
 */


/*****************\
 * Page Mappings *
\*****************/
```

```
typedef int PageFault_t;
#define PF_RESID        ((PageFault_t)1)        /* Residency fault */
#define PF_WRITE        ((PageFault_t)2)        /* Write fault */
#define PF_FLUSH        ((PageFault_t)3)        /* Flush event */

typedef int PageDisp_t;
#define P_DSP_ZERO      ((PageDisp_t)1)
#define P_DSP_REPLACE   ((PageDisp_t)2)
#define P_DSP_KEEP      ((PageDisp_t)3)


/**************\
 * Exceptions *
\**************/

typedef int Excpt_t;
#define E_KILL  ((Excpt_t)1)    /* thread killed */
#define E_SHUT  ((Excpt_t)2)    /* task shutdown */
#define E_PROT  ((Excpt_t)3)    /* protection violation */
#define E_NOOB  ((Excpt_t)4)    /* touching a non existent object */
#define E_ARITH ((Excpt_t)5)    /* arithmetic exception */
#define E_UPCL  ((Excpt_t)6)    /* upcall failed */
#define E_ILL   ((Excpt_t)7)    /* illegal instruction */

typedef void (*ExcptHndlr_t)(Excpt_t,void*);

#endif /* _MUNGI_TYPES_H_ */
```

A.2 mungi/status.h

```
/*************************************************************************\
.                                                                         .
.         mungi/status.h: Mungi kernel error status values                .
.         Release 1.0 of 1997-04-30                                       .
.         (c) Copyright 1994,95,96,97 by University of New South Wales    .
.                                                                         .
.         Author: Jerry Vochteloo and Gernot Heiser                       .
.         Distributed Systems Research Group                              .
.         Department of Computer Systems                                  .
.         School of Computer Science and Engineering                      .
.         The University of New South Wales                               .
.         Sydney 2052                                                     .
.         Australia                                                       .
.                                                                         .
.         For details please check http://www.cse.unsw.edu.au/~disy       .
.         or contact <disy@cse.unsw.edu.au>.                              .
.                                                                         .
.         Permission to make verbatim digital/hard copy of this           .
.         file for personal or classroom use is granted without fee       .
.         provided that copies are not made or distributed for profit     .
.         or commercial advantage. To copy otherwise requires prior       .
.         specific permission.                                            .
.                                                                         .
\*************************************************************************/


#ifndef _MUNGI_STATUS_H_
#define _MUNGI_STATUS_H_

#define ST_NOMEM        0x01    /* out of memory */
#define ST_SIZ          0x02    /* invalid size */
#define ST_NULL         0x03    /* invalid null value */
#define ST_POS          0x04    /* invalid position */
#define ST_CAP          0x05    /* invalid capability */
#define ST_CLIST        0x06    /* invalid Clist */
#define ST_PWD          0x07    /* invalid password */
#define ST_LOCK         0x11    /* APD locked */
#define ST_NOGROW       0x12    /* cannot grow */
#define ST_OVFL         0x13    /* table overflow */
#define ST_THR          0x14    /* invalid thread ID */
#define ST_TSK          0x15    /* invalid task ID */
#define ST_PROT         0x16    /* protection violation (non-offspring) */
#define ST_RNG          0x17    /* invalid range */
#define ST_EXCPT        0x18    /* invalid exception */
#define ST_USE          0x19    /* semaphore in use */
#define ST_SEMA         0x1a    /* invalid semaphore */

#endif /* _MUNGI_STATUS_H_ */
```

A.3 mungi/syscalls.h

```
/*****************************************************************************\
.                                                                             .
.         mungi/syscalls.h: Mungi kernel API                                  .
.         Release 1.0 of 1997-04-30                                           .
.         (c) Copyright 1994,95,96,97 by University of New South Wales        .
.                                                                             .
.         Author: Jerry Vochteloo and Gernot Heiser                           .
.         Distributed Systems Research Group                                  .
.         Department of Computer Systems                                      .
.         School of Computer Science and Engineering                          .
.         The University of New South Wales                                   .
.         Sydney 2052                                                         .
.         Australia                                                           .
.                                                                             .
.         For details please check http://www.cse.unsw.edu.au/~disy           .
.         or contact <disy@cse.unsw.edu.au>.                                  .
.                                                                             .
.         Permission to make verbatim digital/hard copy of this               .
.         file for personal or classroom use is granted without fee           .
.         provided that copies are not made or distributed for profit         .
.         or commercial advantage. To copy otherwise requires prior           .
.         specific permission.                                                .
.                                                                             .
\*****************************************************************************/


#ifndef _MUNGI_SYSCALLS_H_
#define _MUNGI_SYSCALLS_H_

#include <mungi/types.h>



/***********\
 * Objects *
\***********/


Cap_t
ObjCreate(uint64 size, uint64 passwd, const ObjInfo_t *info);
/* Allocates object of "size" bytes with owner password "passwd" and object
 * info "*info".
 * Returns owner capability or NULL (ST_NOMEM, ST_SIZ).
 */

uint
ObjDelete(Cap_t obj);
/* Deallocates object "obj".
 * Returns 0 if successful, !=0 otherwise (ST_CAP).
 */
```

```
Cap_t
ObjResize(Cap_t obj, uint64 new_size, bool may_move);
/* Resizes object "obj" to new size "new_size". If "may_move" object will
 * move if it cannot be extended in situ.
 * Returns (new) object capability or NULL (ST_CAP, ST_SIZ, ST_NOGROW).
 */

uint
ObjNewPager(Cap_t obj, Cap_t pager);
/* Registers PDX cap "pager" as the page fault handler for "obj".  Returns
 * zero if successful, otherwise !=0 (ST_CAP, ST_LOCK).  The pager
 * signature is
 *    bool pager(const void *adr, uint n_pages, PageFault_t fault).
 */


Cap_t
ObjCrePasswd(Cap_t obj, uint64 passwd, Access_t mode);
/* Registers a new capability for "obj", containing the password "passwd"
 * and conferring rights "mode".
 * Returns the new capability if sucessful, NULL otherwise (ST_CAP,
 * ST_OVFL, ST_LOCK).
 */

Cap_t
ObjCrePdx(Cap_t obj, uint64 passwd, Cap_t clist,
          uint n_entrypt, Pdx_t entry_0, ...);
/* Registers a new PDX capability for "obj", containing the password
 * "passwd". When called, the PDX procedure will extend the callers
 * protection domain by the specified "clist". PDX calls via the new
 * capability are valid to one of the "n_entrypt" specified entry points
 * "entry_0, ...".
 * Returns the new capability if sucessful, NULL otherwise (ST_CAP,
 * ST_OVFL, ST_LOCK, ST_CLIST, ST_NULL).
 */

uint
ObjDelPasswd(Cap_t obj, uint64 passwd);
/* Revoke the specified password for "obj".
 * Returns zero if successful, != 0 otherwise (ST_CAP, ST_PWD).
 */

uint
ObjInfo(Cap_t obj, DateFlags_t flags, ObjInfo_t *info);
/* Update the object table entry for "obj". The parameter "flags" specifies
 * which time stamps are set (to the value specified in "info") or touched
 * (i.e. set to the present time).
 * Returns, through "info", the pre-call attribute settings.
 * Returns zero if successful, != 0 otherwise (ST_CAP).
 */
```

```
/*******\
 * APD *
\*******/


uint
ApdInsert(uint8 pos, Cap_t hndlr_clist, bool is_handler);
/* Insert at position "pos" in the kernel's APD data structure
 * "hndlr_clist", shifting up any further entries. If "is_handler" this is
 * a capability handler pointer, otherwise the cap of a Clist.
 * Returns zero if successful, != 0 otherwise (ST_CAP, ST_POS, ST_OVFL,
 * ST_LOCK).  */

uint
ApdDelete(uint8 slot);
/* Remove the Clist cap or capability handler pointer at position "pos" in
 * the kernel's APD data structure (shifting down any further entries).
 * Returns zero if successful, != 0 otherwise (ST_POS).
 */


uint
ApdGet(ApdDesc_t *apd);
/* Return a copy of the kernel's APD data structure through "apd". Only the
 * address part of Clist caps are returned (no passwords).
 * Returns zero if successful, != 0 otherwise (???).
 */

uint
ApdFlush(void);
/* Flush the validation cache, forcing revalidation to occur for all future
 * object accesses. Also perform re-validation of Clist caps in the APD.
 * Returns zero if successful, != 0 otherwise (???).
 */


Cap_t
ApdLookup(const void *address, Access_t mode);
/* Validate access to "address" of type "mode".  Returns capability
 * granting the requested rights, NULL if this access is not possible. If
 * the APD is locked, the password part of the returned cap is zeroed. If
 * the access is allowed, the validation is cached as a side effect.
 */


void
ApdLock(void);
/* Lock the caller's APD, imposing restrictions on a number of system
 * calls. A locked APD cannot be unlocked.
 */
```

```
Cap_t
PdxCall(Pdx_t proc, int8 n_pd, uint n_args, ...);
/* Call "proc" as a PDX procedure. The parameter "n_pd" specifies the
 * number of protection domain parameters passed in the variable part of
 * the parameter list. If "n_pd" is zero, the PDX procedure executes in a
 * protection domain which is the union of the caller's APD with the Clist
 * registered for the PDX. If "n_pd" is negative, no protection domain is
 * passed, and the PDX executes in an APD consisting solely of its
 * registered Clist. Otherwise, the extended APD is constructed from the
 * registered Clist and the protection domain explicitly passed via
 * parameters. These parameters are in the form of Clist capabilities, if
 * their password part is zero, they are assumed to be capability handler
 * addresses.  The parameter "n_args" specifies the number of arguments
 * passed to the PDX, these are specified in the argument list after the
 * protection domain parameters.
 * Returns the PDX function's return value.
 */



/*********************\
 * Tasks and Threads *
\*********************/


ThreadId_t
ThreadCreate(Thread_t ip, void *sp);
/* Start a new thread in the caller's task, starting execution at address
 * "ip", using the stack pointer "sp" (possibly NULL).
 * Returns ID of new thread.
 */

TaskId_t
TaskCreate(Task_t p, uint64 n_stack, const void *env,
           uint8 n_pd, uint n_args, ...);
/* Start a new task, running "p" and requiring a stack of size
 * "n_stack". The new task gets a copy of "env" and an APD made up of the
 * "n_pd" Clist capabilities following the declared parameters. (A
 * capability with a zero password is considered to be a pointer to a
 * capability handler.) If "n_pd" is zero, pass caller's APD. Further
 * "n_args" parameters are passed to "p".
 * Returns ID of new task.
 */

uint
ThreadDelete(ThreadId_t thread, int status);
/* Terminates "thread" with an exit value of "status", THREAD_MYSELF
 * terminates the caller.
 * Returns 0 on success, !=0 otherwise (ST_THR).
 */
```

```
uint
TaskDelete(TaskId_t task, int status, bool adopt);
/* Terminates "task" (and all its threads) with an exit value of "status",
 * TASK_MYSELF terminates the caller. If "adopt" is false, all children are
 * terminated too, otherwise they become children of the caller's task.
 * Returns 0 on success, !=0 otherwise (ST_TSK, ST_PROT).
 */

uint
ThreadSleep(ThreadId_t thread, uint msecs);
/* Suspends "thread" for a minimum of "msec" milliseconds, or until
 * explicitly resumed by a ThreadResume system call (whichever comes
 * first). The thread is suspended indefinitely if "msecs" is
 * zero. THREAD_MYSELF suspends the caller.
 * Returns 0 on success, !=0 otherwise (ST_THR).
 */

uint
ThreadResume(ThreadId_t thread);
/* Resumes stopped "thread".
 * Returns 0 on success, !=0 otherwise (ST_THR).
 */

ThreadId_t
ThreadWait(ThreadId_t thread, int *status);
/* Waits for "thread" to exit, THREAD_ANY means any thread.
 * Returns the ID of the exited thread, and, through "status",
 * the exit status, as specified in the ThreadDelete system call.
 */

TaskId_t
TaskWait(TaskId_t task, int *status);
/* Waits for "task" to exit, TASK_ANY means any task.
 * Returns the ID of the exited task, and, through "status", the exit
 * status. That status is the one specified in the "TaskDelete" call, if
 * the task was thus terminated, otherwise the exit status of the task's
 * last exiting thread.
 */

ThreadId_t
ThreadMyID(void);
/* Returns ID of calling thread. */

TaskId_t
TaskMyID(void);
/* Returns ID of task of calling thread. */

uint
ThreadInfo(ThreadId_t thread, int32 prio, uint64 cpulim,
           uint64 memlim, ThreadInfo_t *info);
/* Set/get info on "thread". "prio", "cpulim", "memlim" pass values to set
 * the scheduling priority, CPU time and memory limits, zero means no
```

```
 * limit. Limits can only be decreased, not increased (and attempts to
 * increase them are ignored).
 * Returns, via "info", the pre-call values.
 * Returns 0 if successful, otherwise !=0 (ST_THR).
 */

uint
TaskInfo(TaskId_t task, int32 prio, uint64 cpulim,
           uint64 memlim, TaskInfo_t *info);
/* Set/get info on "task". "prio", "cpulim", "memlim" pass values to set
 * the scheduling priority, CPU time and memory limits, zero means no
 * limit. Limits can only be decreased, not increased (and attempts to
 * increase them are ignored).
 * Returns, via "info", the pre-call values.
 * Returns 0 if successful, otherwise !=0 (ST_TSK, ST_PROT).
 */


/*****************\
 * Page Mappings *
\*****************/


uint
PageCopy(const void *from, const void *to, uint n_pages);
/* Copy memory starting at address "from" to the destination "to", both of
 * which must be page aligned. A total of "n_pages" pages are
 * copied. Copy-on-write is used where possible.  The "from" and "to"
 * ranges must be disjoint. Each range must be fully contained in a single
 * object.
 * Returns 0 if successful, otherwise !=0 (ST_RNG, ST_NULL).
 */

uint
PageMap(const void *from, const void *to, uint n_pages,
        Access_t mode, bool fault_in);
/* Make "n_pages" of virtual address space starting at "to" an alias for
 * the virtual address space starting at "from". The address range starting
 * at "to" will be accessible according to "mode". If "fault_in" is TRUE,
 * non-resident pages in the "from" range are faulted in, otherwise the
 * operation is a no-op on such pages.  The two address ranges must either
 * be disjoint or identical. Each range must be fully contained in a single
 * object.
 * Returns 0 if successful, otherwise !=0 (ST_RNG, ST_NULL).
 */

uint
PageUnMap(const void *page, uint n_pages, PageDisp_t disp);
/* Invalidate virtual memory mappings for "n_pages" pages starting at
 * "page". If the pages belong to an object handled by a user-level pager,
 * their mappings simply vanish, otherwise "disp" determines the semantics
 * of the operation. If disp is P_DSP_ZERO, the pages are zeroed (by
```

```
 * disassociating them from any backing store). If disp is P_DSP_REPLACE,
 * the physical frames are freed (without first flushing dirty pages) but
 * the association with backing store is kept. If disp is P_DSP_KEEP, the
 * mappings are retained but marked as invalid, forcing a page fault on the
 * next access. All pages of the range must belong to the same object.
 * Returns 0 if successful, otherwise !=0 (ST_TSK, ST_NULL).
 */

uint
PageUnalias(const void *page, uint n_pages);
/* Remove all mappings established _from_ the specified page range (which
 * must be part of a single object). This is a no-op on unaliased pages.
 * Returns 0 if successful, otherwise !=0 (ST_RNG, ST_NULL).
 */

uint
PageFlush(const void *page, uint n_pages);
/* Ensure that the designated range of pages is clean, i.e. all changes are
 * flushed to disk. Also ensures that association with backing store of
 * pages handled by the default pager is recorded on stable storage. The
 * whole page range must belong to a single object.
 * Returns 0 if successful, otherwise !=0 (ST_RNG, ST_NULL).
 */

#define PageSize(adr) 0x1000
/* Returns the page size of the object containing "adr".
 */



/**************\
 * Exceptions *
\**************/


ExcptHndlr_t
ExcptReg(Excpt_t exception, ExcptHndlr_t handler);
/* Registers "handler" for "exception".
 * Returns previous handler or NULL if none was registered.
 */



/**************\
 * Semaphores *
\**************/


uint
SemCreate(const void *address, int64 value);
/* Creates a semaphore identified by "address" and initialised to "value".
 * Returns zero if successful, != 0 otherwise (ST_USE).
```

```
 */

uint
SemDelete(const void *address);
/* Destroys the semaphore identified by "address".
 * Returns zero if successful, != 0 otherwise (ST_SEMA).
 */

uint
SemWait(const void *address);
/* Performs a wait operation on the semaphore identified by "address".
 * Returns zero if successful, != 0 otherwise (ST_SEMA).
 */

uint
SemSignal(const void *address);
/* Performs a signal operation on the semaphore identified by "address".
 * Returns zero if successful, != 0 otherwise (ST_SEMA).
 */


#endif /* _MUNGI_SYSCALLS_H_ */
```