UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

Dennis Ritchie

2

Phase 1 - Level 2: The Queues Manager

Level 2 of Pandos instantiates the key operating system concept that active entities at one layer are just data structures at lower layers. In this case, the active entities at a higher level are processes (i.e. programs in execution) and the data structure(s) that represent them at this level are *process control blocks* (*pcb*s).

```
/* process control block type */
typedef struct pcb_t {
    /* process queue fields */
    struct pcb_t        *p_next,     /* pointer to next entry */
                        *p_prev,     /* pointer to prev entry */

    /* process tree fields */
                        *p_prnt,     /* pointer to parent     */
                        *p_child,    /* pointer to 1st child  */
                        *p_sib;      /* pointer to sibling    */

    /* process status information */
    state_t             p_s;         /* processor state */
    cpu_t               p_time;      /* cpu time used by proc */
    int                 *p_semAdd;   /* pointer to sema4 on   */
                                     /* which process blocked */

    /* support layer information */
    support_t           *p_supportStruct;
                                     /* ptr to support struct */

    /* namespace layer information */
    nsd_t               *namespaces[NS_TYPE_MAX];
                        /* Namespace pointer per type */

} pcb_t;
```

The queue manager will implement four *pcb* related sets of functions:

- The allocation and deallocation of *pcb*s.
- The maintenance of queues of *pcb*s.
- The maintenance of trees of *pcb*s.
- The maintenance of an hash table of *active semaphore descriptors*, each of which supports a queue of *pcb*s: The ASH.

- The maintenance of a lis of *namespace descriptors* per type.

2.1 The Allocation and Deallocation of *pcb*s

One may assume that Pandos supports no more that *MAXPROC* concurrent processes; where MAXPROC should be set to 20 (in the const.h) file.[1] Thus this level needs a "pool" of MAXPROC *pcb*s to allocate from and deallocate to. Assuming that there is a set of MAXPROC *pcb*s, the free or unused ones can be kept on a NULL-terminated single, linearly linked list (using the `p_next` field), called the *pcbFree* List, whose head is pointed to by the variable `pcbFree_h`.

To support the allocation and deallocation of *pcb*s there should be the following three externally visible functions:

- *pcb*s which are no longer in use can be returned to the pcbFree list by using the method:
  `void freePcb(pcb_t *p)`

  > /* Insert the element pointed to by `p` onto the pcbFree list. */

- *pcb*s should be allocated by using:
  `pcb_t *allocPcb()`

  > /* Return NULL if the pcbFree list is empty. Otherwise, remove an element from the pcbFree list, provide initial values for ALL of the *pcb*s fields (i.e. NULL and/or 0) and then return a pointer to the removed element. *pcb*s get reused, so it is important that no previous value persist in a *pcb* when it gets reallocated. */

There is still the question of how one acquires storage for MAXPROC *pcb*s and gets these MAXPROC *pcb*s initially onto the pcbFree list. Unfortunately, there is no `malloc()` feature to acquire dynamic (i.e. non-automatic) storage that will persist for the lifetime of the OS and not just the lifetime of the function they are declared in. Instead, the storage for the MAXPROC *pcb*s will be allocated as *static* storage. A static array of MAXPROC *pcb*s will

[1] A supplied "starter" version of const.h can be found in/usr/include/umps3/umps

be declared in `initPcbs()`. Furthermore, this method will insert each of the MAXPROC *pcb*s onto the pcbFree list.

- To initialize the pcbFree List:
  `initPcbs()`

  /* Initialize the pcbFree list to contain all the elements of the static array of MAXPROC *pcb*s. This method will be called only once during data structure initialization. */

2.2 Process Queue Maintenance

The methods below do not manipulate a particular queue or set of queues. Instead they are generic queue manipulation methods; one of the parameters is a pointer to the queue upon which the indicated operation is to be performed.

The queues of *pcb*s to be manipulated, which are called *process queues*, are all double, circularly linked lists, via the `p_next` and `p_prev` pointer fields. Instead of a head pointer, each queue will be pointed at by a tail pointer.

To support process queues there should be the following externally visible functions:

`pcb_t *mkEmptyProcQ()`

/* This method is used to initialize a variable to be tail pointer to a process queue.
Return a pointer to the tail of an empty process queue; i.e. NULL.
*/

`int emptyProcQ(pcb_t *tp)`

/* Return TRUE if the queue whose tail is pointed to by `tp` is empty. Return FALSE otherwise. */

`insertProcQ(pcb_t **tp, pcb_t *p)`

/* Insert the *pcb* pointed to by `p` into the process queue whose tail-pointer is pointed to by `tp`. Note the double indirection through `tp` to allow for the possible updating of the tail pointer as well.
*/

`pcb_t *removeProcQ(pcb_t **tp)`

> /* Remove the first (i.e. head) element from the process queue whose tail-pointer is pointed to by `tp`. Return NULL if the process queue was initially empty; otherwise return the pointer to the removed element. Update the process queue's tail pointer if necessary. */

`pcb_t *outProcQ(pcb_t **tp, pcb_t *p)`

> /* Remove the *pcb* pointed to by `p` from the process queue whose tail-pointer is pointed to by `tp`. Update the process queue's tail pointer if necessary. If the desired entry is not in the indicated queue (an error condition), return NULL; otherwise, return `p`. Note that `p` can point to any element of the process queue. */

`pcb_t *headProcQ(pcb_t *tp)`

> /* Return a pointer to the first *pcb* from the process queue whose tail is pointed to by `tp`. Do not remove this *pcb*from the process queue. Return NULL if the process queue is empty. */

2.3 Process Tree Maintenance

In addition to possibly participating in a process queue, *pcb*s are also organized into trees of *pcb*s, called *process trees.* The `p_prnt`, `p_child`, and `p_sib` pointers are used for this purpose.

The process trees should be implemented as follows. A parent *pcb* contains a pointer (`p_child`) to a NULL-terminated single, linearly linked list of its child *pcb*s. Each child process has a pointer to its parent *pcb* (`p_prnt`) and possibly the next child *pcb* of its parent (`p_sib`). For greater efficiency you may want to make the linked list of child *pcb*s a NULL-terminated double, linearly linked list.

To support process trees there should be the following externally visible functions:
`int emptyChild(pcb_t *p)`

> /* Return TRUE if the *pcb* pointed to by `p` has no children. Return FALSE otherwise. */

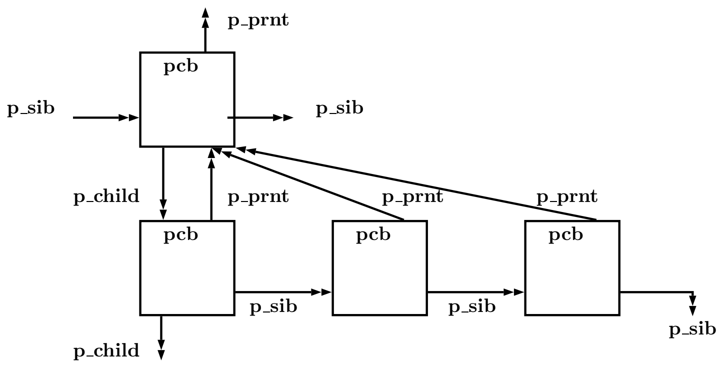

Figure 2.1: Process Tree

`insertChild(pcb_t *prnt, pcb_t *p)`

> /* Make the *pcb* pointed to by `p` a child of the *pcb* pointed to by `prnt`. */

`pcb_t *removeChild(pcb_t *p)`

> /* Make the first child of the *pcb* pointed to by `p` no longer a child of `p`. Return NULL if initially there were no children of `p`. Otherwise, return a pointer to this removed first child *pcb*. */

`pcb_t *outChild(pcb_t *p)`

> /* Make the *pcb* pointed to by `p` no longer the child of its parent. If the *pcb* pointed to by `p` has no parent, return NULL; otherwise, return `p`. Note that the element pointed to by `p` need not be the first child of its parent. */

2.4 The Active Semaphore Hash (ASH)

A *semaphore* is an important operating system concept. While understanding semaphores is not yet needed, this level nevertheless implements an important data structure/abstraction which supports Pandos's implementation of semaphores.

For the purpose of this level it is sufficient to think of a semaphore as an integer. Associated with this integer is:

- An address; semaphores, like all integers, have a physical address in memory.
- A process queue.

A semaphore is *active* if there is at least one *pcb* on the process queue associated with it. (i.e. The process queue is not empty: `emptyProcQ(s_procq)` is FALSE.)

The following implementation is suggested: Maintain an hash table of semaphore descriptors which key is represented by its pointer. The hash is represented by `semd_h`. The hash `semd_h` points to will represent the *Active Semaphore Hash* (ASH).

```
/* semaphore descriptor type */
typedef struct semd_t {
    struct hash_table_entry s_link;      /* ASH reference */
    struct semd_t           *s_freelink; /* next element on the */
                                         /* free semaphore list */
    int                     *s_semAdd;   /* semaphore pointer   */
    pcb_t                   *s_procQ;    /* tail pointer to a   */
                                         /* process queue       */
} semd _t;
```

Maintain a list of semaphore descriptors, the *semdFree* list, to hold the unused semaphore descriptors. This list, whose head is pointed to by the variable `semdFree_h`, is kept, like the pcbFree list, as a NULL-terminated single, linearly linked list (using the `s_freelink` field).

The semaphore descriptors themselves should be declared, like the *pcb*s, as a static array of size MAXPROC of type `semd_t`.

To support the ASH there should be the following externally visible functions:

`int insertBlocked(int *semAdd, pcb_t *p)`

> /* Insert the *pcb* pointed to by `p` at the tail of the process queue associated with the semaphore whose physical address is `semAdd` and set the semaphore address of `p` to semAdd. If the semaphore is currently not active (i.e. there is no descriptor for it in the ASH), allocate a new descriptor from the semdFree list, insert it in the ASH, initialize all of the fields (i.e. set `s_semAdd` to `semAdd`,

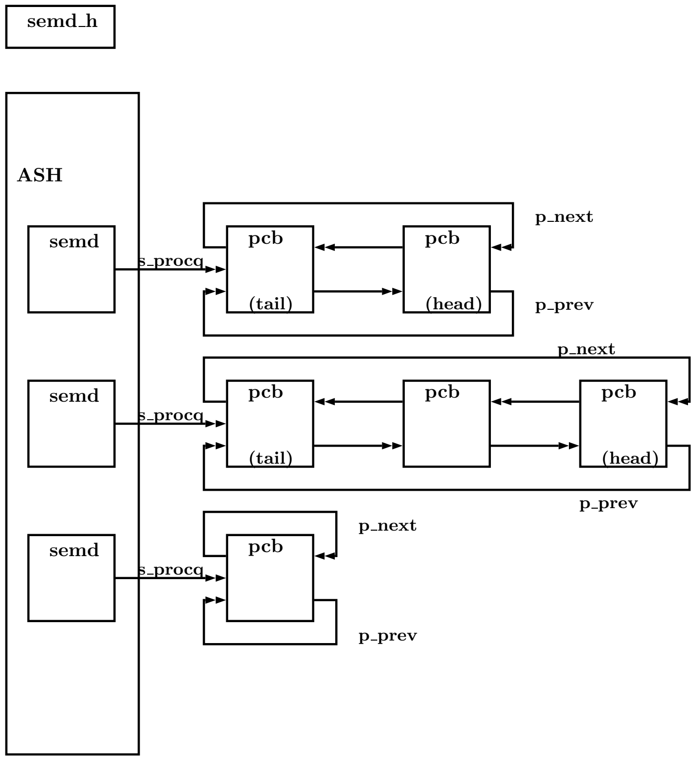

Figure 2.2: Active Semaphore Hash

and `s_procq` to `mkEmptyProcQ()`), and proceed as above. If a new semaphore descriptor needs to be allocated and the semdFree list is empty, return TRUE. In all other cases return FALSE. */

`pcb_t *removeBlocked(int *semAdd)`

/* Search the ASH for a descriptor of this semaphore. If none is found, return NULL; otherwise, remove the first (i.e. head) *pcb* from the process queue of the found semaphore descriptor and return a pointer to it. If the process queue for this semaphore becomes empty (`emptyProcQ(s_procq)` is TRUE), remove the semaphore descriptor from the ASH and return it to the semdFree list. */

`pcb_t *outBlocked(pcb_t *p)`

/* Remove the *pcb* pointed to by `p` from the process queue associated with `p`'s semaphore (`p`→ `p_semAdd`) on the ASH. If *pcb* pointed to by `p` does not appear in the process queue associated with `p`'s semaphore, which is an error condition, return NULL; otherwise, return `p`. */

`pcb_t *headBlocked(int *semAdd)`

/* Return a pointer to the *pcb* that is at the head of the process queue associated with the semaphore `semAdd`. Return NULL if `semAdd` is not found on the ASH or if the process queue associated with `semAdd` is empty. */

`initASH()`

/* Initialize the semdFree list to contain all the elements of the array
static `semd_t semdTable[MAXPROC]`
This method will be only called once during data structure initialization. */

Technical Point: Strive to structure the ASH code so that there is one internal/helper function that traverses the ASH and is used by `insertBlocked`, `removeBlocked`, `outBlocked`, and `headBlocked`.

2.5 Namespace management

A *namespace* is an operating system-level virtualization concept. Every process or thread can have its view on the Pandos's system.

Every namespace is then represented by a type implemented as an integer.

A NULL value in the namespace key represent the *base* namespace. i.e. the namespace in which, by default, every process resides.

You can use the following structure to describe a namespace.

```
/* namespace descriptor type */
typedef struct nsd_t {
    struct nsd_t *n_next;     /* next element on the free list */
    int          n_type;      /* pointer to the semaphore      */
} nsd_t;
```

Maintain a list of free namespace descriptors per each type, e.g. *pid_nsFree* indexed by their pointer to hold unused namespace descriptors. These lists, whose are represented by the variables `type_nsFree`, are kept, like the ASH free list, in various global variables.

The namespace descriptors themselves should be declared, like the *pcb*s, as one array of size MAXPROC (per namespace type) of type `nsd_t`. For the sake of phase1, only a single namespace type (PID_NS) should be declared.

To support the Namespaces there should be the following externally visible functions:

`nsd_t *getNamespace(pcb_t *p, int type)`

> /* Return the pointer to the namespace descriptor of type `type` associated with the pcb p. */

`nsd_t *allocNamespace(int type)`

> /* Allocate a namespace from the `type_nsFree` list and return to the user, this value can be used for the next calls to refer to this namespace. */

`void freeNamespace(nsd_t *descriptor)`

> /* Free a namespace, adding its list pointer (`n_link`) to the correct `type_nsFree` list. */

`int addNamespace( *ns, pcb_t *p)`

> /* Insert the namespace `ns` as the namespace for the correct type of `p` to ns. If the namespace is currently the base (i.e. there are no descriptor for the namespace), allocate a new descriptor from the type_nsFree list, insert it in the correct list, initialize all of the fields, and proceed as above. If a new namespace descriptor needs to be allocated and the `type_nsFree` list is empty, return TRUE. In all other cases return FALSE. */

`void initNamespaces()`

> /* Initialize the type_nsFree list to contain all the elements of the arrays
> **static** `ns_t type_nsTable[MAXPROC]`
> This method will be only called once during data structure initialization. */

2.6 Nuts and Bolts

There is no one right way to implement the functionality of this level. The recommended approach is to create three modules (i.e. files): one for the ASH, one for the Namespace management, and one for *pcb* initialization, allocation, deallocation, process queue maintenance, and process tree maintenance.

The second module, pcb.c, in addition to the public and HIDDEN/private helper functions, will also contain the declaration for the private global variable that points to the head of the pcbFree list.

```
HIDDEN pcb_t *pcbFree_h;
```

The ASL module, asl.c, in addition to the public and HIDDEN/private helper functions, will also contain the declarations for `semd_h` and `semdFree_h`

```
HIDDEN semd_t *semd_h, *semdFree_h;
```

Since the ASL module will make calls to the process queue module to manipulate the process queue associated with each active semaphore, this module should

```
#include "pcb.h"
```

This will insure that the ASH can only use the externally visible functions from pcb.c for maintaining its process queues.

Furthermore, the declaration for `pcb_t` would then be placed in the types.h file.[2] This is because many other modules will need to access this definition. The declaration for `semd_t` can be placed in either asl.c (because no other module will ever need to access this definition), or types.h.

2.7 Testing

There is a provided test file, p1test.c that will "exercise" your code. [Appendix **??**]

[2]A supplied "starter" version of types.h can be found in /usr/include/umps3/umps

As with any non-trivial system, you are strongly encouraged to use the *make* program to maintain your code. A sample *Makefile* has been supplied for you to use. See Chapter **??** in the POPS reference for more compilation details.

Once your (three?) source files have been correctly compiled, linked together (with appropriate linker script, `crtso.o`, and `libumps.o`), and post-processed with umps3-elf2umps (all performed by the sample *Makefile*), your code can be tested by launching the μMPS3 emulator. At a terminal prompt, enter:

```
umps3
```

The test program reports on its progress by writing messages to TERMINAL0. These messages are also added to one of two memory buffers; `errbuf` for error messages and `okbuf` for all other messages. At the conclusion of the test program, either successful or unsuccessful, μMPS3 will display a final message and then enter an infinite loop. The final message will either be System Halted for successful termination, or Kernel Panic for unsuccessful termination.