MultiPandOS: Phase 1

Luca Bassi (luca.bassi14@studio.unibo.it)
Luca Orlandello (luca.orlandello@studio.unibo.it)

November 10, 2024

Introduction

The MultiPandOS operating system described below is inspired to the previous experience of PandOSplus and pKaya OS.

1 Phase 1 - Level 2: The Queue Managers

Level 2 of MultiPandOS 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* (PCBs).

```c
/* process table entry type */
typedef struct pcb_t {
    /* process queue  */
    struct list_head p_list;

    /* process tree fields */
    struct pcb_t    *p_parent; /* ptr to parent */
    struct list_head p_child;  /* children list */
    struct list_head p_sib;    /* sibling list  */

    /* process status information */
    state_t p_s;    /* processor state */
    cpu_t   p_time; /* cpu time used by proc */

    /* Pointer to the semaphore the process is currently blocked on */
    int *p_semAdd;

    /* Pointer to the support struct */
    support_t *p_supportStruct;

    /* process id */
    int p_pid;
} pcb_t, *pcb_PTR;
```

The Process Queue Manager will implement three PCB related sets of functions:

- The allocation and deallocation of PCBs.
- The maintenance of queues of PCBs.
- The maintenance of trees of PCBs.

2 Processes

2.1 The Allocation and Deallocation of PCBs

One may assume that MultiPandOS supports no more that `MAXPROC` concurrent processes; where `MAXPROC` should be set to 20 (in the `const.h` file). Thus this level needs a "pool" of `MAXPROC` PCBs to allocate from and deallocate to. Assuming that there is a set of `MAXPROC` PCBs, the free or unused ones can be kept on a double, circularly linked list (using the `p_list` field), called the `pcbFree` list, whose head is pointed to by the variable `pcbFree_h`.

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

- PCBs which are no longer in use can be returned to the `pcbFree_h` list by using the method:

  `void freePcb(pcb_t *p)`
  Insert the element pointed to by `p` onto the `pcbFree` list.

- PCBs 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 PCBs fields (i.e. NULL and/or 0) and then return a pointer to the removed element. PCBs 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` PCBs and gets these `MAXPROC` PCBs 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` PCBs will be allocated as static storage. A `static pcb_t pcbTable[MAXPROC]` will be declared in Process Queue Manager module [Section 4]. Furthermore, this method will insert each of the `MAXPROC` PCBs 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` PCBs. 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 PCBs to be manipulated, which are called process queues, are all double, circularly linked lists, via the `p_list` field.

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

`void mkEmptyProcQ(struct list_head *head)`
This method is used to initialize a variable to be head pointer to a process queue.

`int emptyProcQ(struct list_head *head)`
Return TRUE if the queue whose head is pointed to by `head` is empty. Return FALSE otherwise.

`void insertProcQ(struct list_head *head, pcb_t *p)`
Insert the PCB pointed by `p` into the process queue whose head pointer is pointed to by `head`.

`pcb_t *headProcQ(struct list_head *head)`
: Return a pointer to the first PCB from the process queue whose head is pointed to by `head`. Do not remove this PCB from the process queue. Return NULL if the process queue is empty.

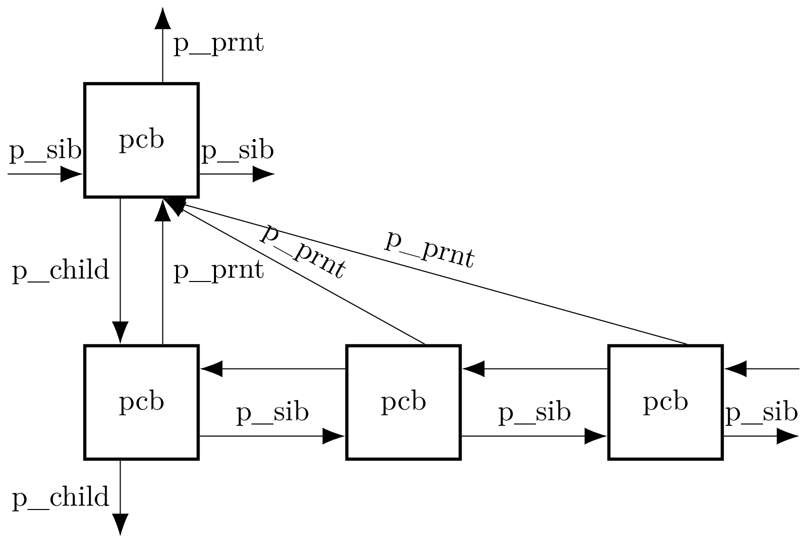
`pcb_t *removeProcQ(struct list_head *head)`
: Remove the first (i.e. head) element from the process queue whose head pointer is pointed to by `head`. Return NULL if the process queue was initially empty; otherwise return the pointer to the removed element.

`pcb_t *outProcQ(struct list_head *head, pcb_t *p)`
: Remove the PCB pointed to by `p` from the process queue whose head pointer is pointed to by `head`. 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.

2.3 Process Tree Maintenance

In addition to possibly participating in a process queue, PCBs are also organised into trees of PCBs, called *process trees*. `p_parent` (abbreviated as `p_prnt`), `p_child`, and `p_sib` are used for this purpose.

The process trees should be implemented as follows. A parent PCB contains a list of children PCBs (`p_child`), each child process `p_sib` can be used to add the child to this list. Each child process has a pointer to its parent PCB (`p_prnt`).



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.

`void 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` could be in an arbitrary position (i.e. not be the first child of its parent).

3 Semaphores

3.1 The Active Semaphore List (ASL)

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 MultiPandOS'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:

- A pointer to physical address in memory where the integer is stored (`s_key` field of `sem_d`).
- A process queue (`s_procq` field of `sem_d`).

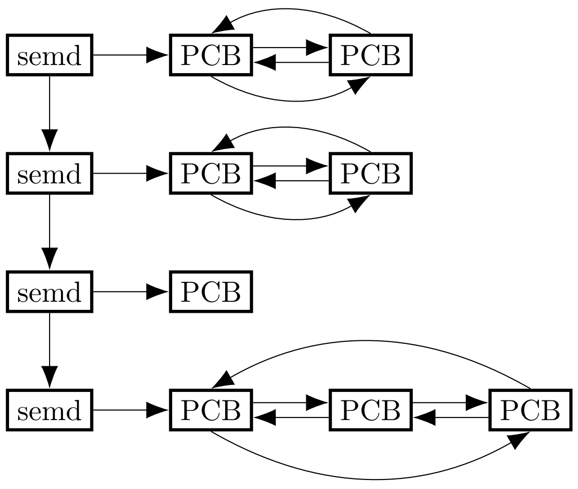
A semaphore is *active* if there is at least one PCB on the process queue associated with it (i.e. The process queue is not empty: `emptyProcQ(s_procq)` is FALSE). The following implementation is suggested: Maintain a sorted NULL-terminated single, linearly linked list (using the `s_link` field) of semaphore descriptors whose head is pointed to by the variable `semd_h`. The list `semd_h` points to will represent the *Active Semaphore List* (ASL).

```c
/* semaphore descriptor (SEMD) data structure */
typedef struct semd_t {
    /* Semaphore key */
    int *s_key;
    /* Queue of PCBs blocked on the semaphore */
    struct list_head s_procq;

    /* Semaphore list */
    struct list_head s_link;
} semd_t, *semd_PTR;
```

Maintain a second 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 double, circularly linked list (using the `s_link` field).

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



To support the ASL 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 key is `semAdd` and set the semaphore address of `p` to semaphore with `semAdd`. If the semaphore is currently not active (i.e. there is no descriptor for it in the ASL), allocate a new descriptor from the semdFree list, insert it in the ASL (at the appropriate position), initialize all of the fields (i.e. set `s_key` to `semAdd`, 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 ASL 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 ASL 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 ASL. 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 ASL or if the process queue associated with `semAdd` is empty.

`initASL()`

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 ASL code so that there is one internal/helper function that traverses the ASL and is used by `insertBlocked`, `removeBlocked`, `outBlocked` and `headBlocked`.

4 Nuts and Bolts

There is no one right way to implement the functionality of this level. The recommended approach is to create two modules (i.e. files): the first module, `asl.c`, for the ASL and the sencond module, `pcb.c`, for PCB initialisation/allocation/deallocation, process queue maintenance, and process tree maintenance. The second module, in addition to the public and private (`HIDDEN`) helper functions, will also contain the declaration for the `pcbTable` and for the private global variable that points to the head of the `pcbFree` list.

```
static pcb_t pcbTable[MAXPROC];
LIST_HEAD(pcbFree_h);
static int next_pid = 1;
```

The ASL module, `asl.c`, in addition to the public and private (`HIDDEN`) helper functions, will also contain the declarations for the `semd_table` and for the private global variables that point to the head of the `semdFree` list and `semd` list.

```
static semd_t semd_table[MAXPROC];
static struct list_head semdFree_h;
static struct list_head semd_h;
```

The declarations for `pcb_t` and `semd_t` would then be placed in the `types.h` file. This is because many other modules will need to access these definitions.

5 Testing

There is a provided test file, `p1test.c`, that will check the behaviour of your code.

As with any non-trivial system, you are strongly encouraged to use the `cmake` program to maintain your code. A sample `CMakeLists.txt` has been supplied for you to use.

Once your source files have been correctly compiled, linked together (with appropriate linker script, `crtso.o`, and `liburiscv.o`), and post-processed with `uriscv-elf2uriscv` (all performed by the sample `CMakeLists.txt`), your code can be tested by launching the μRISCV emulator.

At a terminal prompt, enter: `uriscv`. Create a new machine configuration in the project folder.

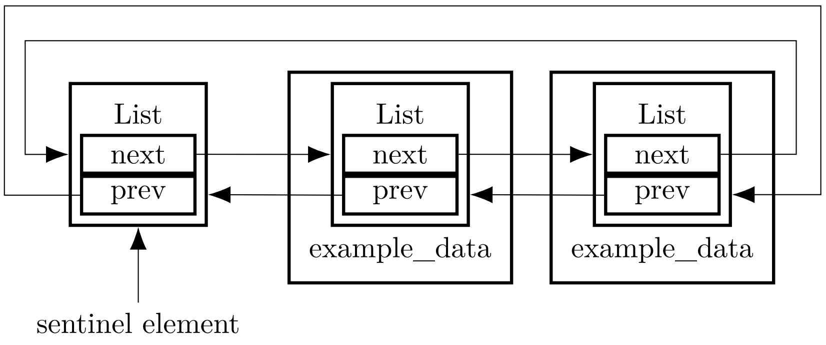
The test program reports on its progress by writing messages to Windows > Terminal 0. These messages are also added to one of two memory buffers; `errbuf` for error messages and `okbuf` for all other messages. At the conclusion of the test program, either successful or unsuccessful, μRISCV will display 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 (you could have to click the play button a second time if Exceptions is enable in the emulator Stop Mask).

Linux kernel lists

We will use the generic and type-oblivious lists of the Linux kernel. If it's needed to create a list for an existing data type, just add a `list_head` field in the struct (this means that the `list_head` struct is type-oblivious). These are double linked lists, so every element of the list is pointed to the next and previous element in the list. There is a sentinel element that isn't an element of the list, but it's used to link the first and last elements of the list. This data type comes with a useful range of macros and functions, located in the `listx.h` file (it's suggested to take a look at the file and read the code to understand its functionality).

```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
}

struct example_data {
    int value;
    // ...
    struct list_head e_list;
}
```



`#define LIST_HEAD_INIT(name)`
Macro to create an empty list.

`#define LIST_HEAD(name)`
Macro to create an empty list (declare also the variable).

`void INIT_LIST_HEAD(struct list_head *list)`
: Inline function to create an empty list.

`int list_empty(const struct list_head *head))`
: Return 1 if the list pointed by `head` is empty.

`void list_add(struct list_head *new, struct list_head *head)`
: Add element point by `new` to the head of list pointed by `head`.

`void list_add_tail(struct list_head *new, struct list_head *head)`
: Add element point by `new` to the tail of list pointed by `head`.

`void __list_add(struct list_head *new, struct list_head *prev, struct list_head *next)`
: Add element point by `new` in an arbitrary position of list pointed by `head`.

`void list_del(struct list_head *entry)`
: Remove element point by `entry` from the list in which it is contained.

`#define container_of(ptr, type, member)`
: Macro to get the struct containing the `list_head` pointed by `ptr`, `type` is the struct name that contain the `list_head`, `member` is the name of the `list_head` inside the struct.

`#define list_for_each(pos, head)`
: Scroll through the list pointed by `head`, the current position will be pointed by `pos`.

`#define list_for_each_entry(pos, head, member)`
: Scroll through the content of the list pointed by `head`, the current element will be pointed by `pos`, `member` is the name of the `list_head` inside the struct.