μPandOS: Phase 1

Luca Bassi (luca.bassi14@studio.unibo.it)
Gabriele Genovese (gabriele.genovese2@studio.unibo.it)

November 6, 2023

μPandOS

- μPandOS is an educational *microkernel* operating system
- evolution of AMIKaya and PandOS; they are also the evolution of a long list of O.S. proposed for educational purposes (PandOS+, Kaya, HOCA, TINA, ICARO, etc).
- μPandOS may be realized on the μMPS3 architecture or on the μRISCV architecture (for the brave's ones)
- Architecture based on six levels of abstraction, on the model of the O.S. THE proposed by Dijkstra in one of his articles of 1968

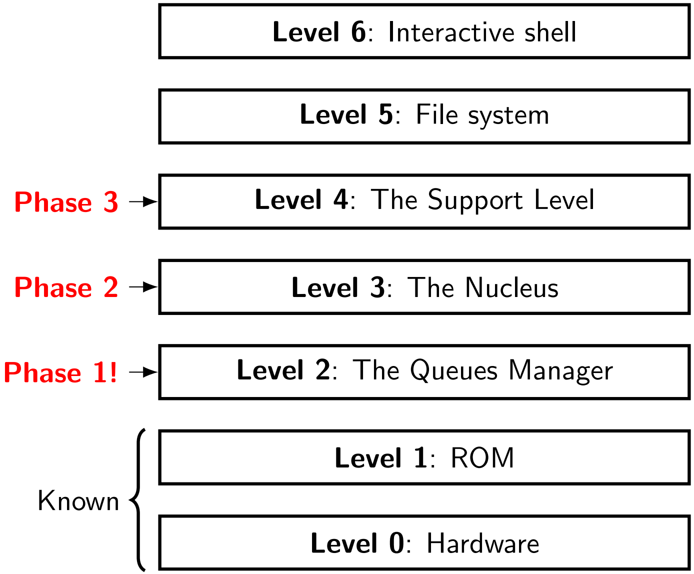
μMPS3

- Emulates a MIPS architecture (introduced in 1985)
- MIPS architecture is widely used in embedded systems
- Tool's version 3 released in 2020
- Pros: tool tested and working, easy to install
- Cons: limited debugger

μRISCV

- Emulates a RISC-V architecture (project started in 2010 and it's evolving)
- RISC-V architecture is fully open source
- Tool not yet fully tested and released
- Pros: debug with GDB
- Cons: no GUI (WIP), tool and toolchain need to be compiled from source

μPandOS: six levels of abstraction



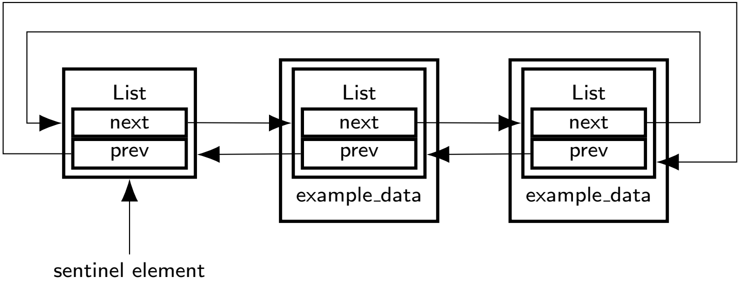
μPandOS Phase 1 requests

- For this first phase, you have to add the functions requested, and the test procedure must conclude correctly.
- Level 2 requirements are purely logical; it is not yet touched the specific hardware of the emulator.
- You are provided with a **.tar.gz** file, that contains useful files to begin with.

The listx module

- A provided module, with a subset of functions of the Linux's list.h module
- It implements generic and type-oblivious lists
- Take a closer look to its content!!

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



Level 2 specs: PCB

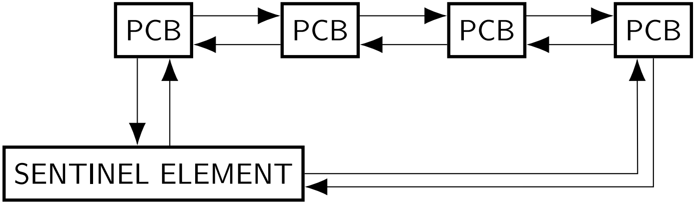
- μPandOS level 2 (The Queues Manager) provides the implementation of the data structures used from the level 3 (The Nucleus)
- The Process Control Block (PCB) represent a process in the system.
- The queue manager implements PCB-related features:
  - Allocation and Deallocation of PCBs.
  - PCB Queue Management.
  - PCB Tree Management.
  - Management of incoming messages.
- ASSUMPTION: no more than MAXPROC (40) concurrent processes in μPandOS

PCB data structure

```
/* 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 */

    /* First message in the message queue */
    struct list_head msg_inbox;

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

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

PCB's list

- PCBs can be organized into queues, called process queues (e.g. active process queue).
- Each list is managed via list_head fields.
- A list is identified by a sentinel element of type list_head.



PCB allocation

Main data structures:

- `pcbFree_h`: free or not used PCBs' list.
- `pcbTable[MAXPROC]`: array of PCBs with max size of MAXPROC.

PCB's functions to implement

Allocation and Deallocation of PCBs:

1. `void freePcb(pcb_t *p)`: insert the element pointed to by `p` onto the `pcbFree` list.
2. `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 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.
3. `void 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.

PCB's functions to implement

PCB Queue Management:

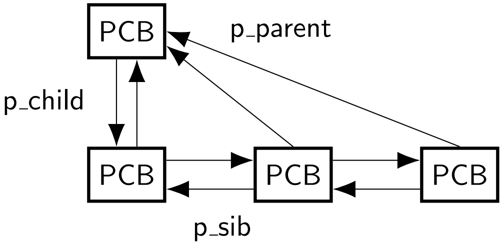
4. `void mkEmptyProcQ(struct list_head *head)`: this method is used to initialize a variable to be head pointer to a process queue.
5. `int emptyProcQ(struct list_head *head)`: return TRUE if the queue whose head is pointed to by `head` is empty. Return FALSE otherwise.
6. `void insertProcQ(struct list_head *head, pcb_t *p)`: insert the PCB pointed by `p` into the process queue whose head pointer is pointed to by `head`.

PCB's functions to implement

7. `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.
8. `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.
9. `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.

PCB's trees

- In addition to the possibility of participating in a process queue, PCBs can be organized into process trees.
- Each parent contains a pointer to the list of the children (`p_child`)
- Each child has a pointer to the parent (`p_parent`) and a pointer that connects the brothers to each other.



PCB's functions to implement

PCB Tree Management:

10. `int emptyChild(pcb_t *p)`: return TRUE if the PCB pointed to by `p` has no children. Return FALSE otherwise.
11. `void insertChild(pcb_t *prnt, pcb_t *p)`: make the PCB pointed to by `p` a child of the PCB pointed to by `prnt`.
12. `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.
13. `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).

Messages

- Access to resources occurs through message passing
- The Message Block (msg) represent a message in the system.
- The queue manager implements msg-related features:
  - Allocation and Deallocation of msgs.
  - Msg Queue Management.
- ASSUMPTION: no more than MAXMESSAGES (40) concurrent messages in μPandOS.

```
typedef struct msg_t
{
    /* message queue */
    struct list_head m_list;

    /* thread that sent this message */
    struct pcb_t *m_sender;

    /* the payload of the message */
        unsigned int m_payload;
} msg_t, *msg_PTR;
```

Messages allocation

Main data structures:

- `msgFree_h`: free or not used msgs' list.
- `msgTable[MAXMESSAGES]`: array of msgs with max size of `MAXMESSAGES`.

Msg's functions to implement

Allocation and Deallocation of msgs:

1. `void freeMsg(msg_t *m)`: insert the element pointed to by `m` onto the `msgFree` list.
2. `msg_t *allocMsg()`: return NULL if the `msgFree` list is empty. Otherwise, remove an element from the `msgFree` list, provide initial values for ALL of the messages fields and then return a pointer to the removed element. Messages get reused, so it is important that no previous value persist in a message when it gets reallocated.
3. `void initMsgs()`: initialize the `msgFree` list to contain all the elements of the static array of `MAXMESSAGES` messages. This method will be called only once during data structure initialization.

Msg's functions to implement

Msg Queue Management:

4. `void mkEmptyProcQ(struct list_head *head)`: used to initialize a variable to be head pointer to a message queue; returns a pointer to the head of an empty message queue.
5. `int emptyMessageQ(struct list_head *head)`: returns TRUE if the queue whose tail is pointed to by `head` is empty, FALSE otherwise.
6. `void insertMessage(struct list_head *head, msg_t *m)`: insert the message pointed to by `m` at the end (tail) of the queue whose head pointer is pointed to by `head`.

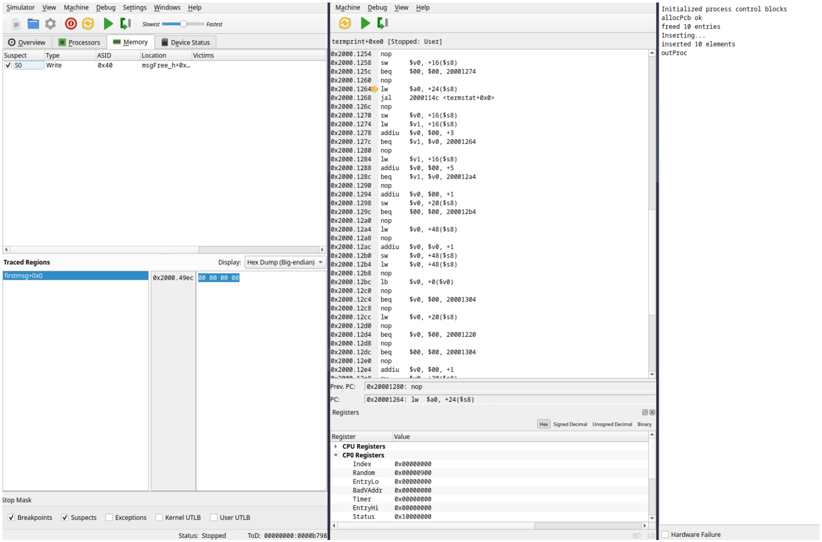
Msg's functions to implement

7. `void pushMessage(struct list_head *head, msg_t *m)`: insert the message pointed to by `m` at the head of the queue whose head pointer is pointed to by `head`.
8. `msg_t *popMessage(struct list_head *head, pcb_t *p_ptr)`: remove the first element (starting by the head) from the message queue accessed via `head` whose sender is `p_ptr`. If `p_ptr` is NULL, return the first message in the queue. Return NULL if the message queue was empty or if no message from `p_ptr` was found; otherwise return the pointer to the removed message.
9. `msg_t *headMessage(struct list_head *head)`: return a pointer to the first message from the queue whose head is pointed to by `head`. Do not remove the message from the queue. Return NULL if the queue is empty.

Suggestions

- There isn't a single way to implement Phase 1's functions.
- Use `static` methods and variables where possible.
- Lean to use a debugger: if you're using `uriscv`, use `gdb`; if you're using `umps3`, use the embedded one (it has some limitations).
- Get familiar with `umps3`

umps3 GUI



Suspect	Type	ASID	Location	Victims
✓ S0	Write	0x40	msgFree_h+0x…	

Traced Regions Display: Hex Dump (Big-endian)

```
firstmsg+0x0    0x2000.49ec    00 00 00 00
```

Stop Mask

✓ Breakpoints ✓ Suspects ☐ Exceptions ☐ Kernel UTLB ☐ User UTLB

Status: Stopped ToD: 00000000:0000b798

```
termprint+0xe0 [Stopped: User]
0x2000.1254  nop
0x2000.1258  sw     $v0, +16($s8)
0x2000.125c  beq    $00, $00, 20001274
0x2000.1260  nop
0x2000.1264  lw     $a0, +24($s8)
0x2000.1268  jal    2000114c <termstat+0x0>
0x2000.126c  nop
0x2000.1270  sw     $v0, +16($s8)
0x2000.1274  lw     $v1, +16($s8)
0x2000.1278  addiu  $v0, $00, +3
0x2000.127c  beq    $v1, $v0, 20001264
0x2000.1280  nop
0x2000.1284  lw     $v1, +16($s8)
0x2000.1288  addiu  $v0, $00, +5
0x2000.128c  beq    $v1, $v0, 200012a4
0x2000.1290  nop
0x2000.1294  addiu  $v0, $00, +1
0x2000.1298  sw     $v0, +20($s8)
0x2000.129c  beq    $00, $00, 200012b4
0x2000.12a0  nop
0x2000.12a4  lw     $v0, +48($s8)
0x2000.12a8  nop
0x2000.12ac  addiu  $v0, $v0, +1
0x2000.12b0  sw     $v0, +48($s8)
0x2000.12b4  lw     $v0, +48($s8)
0x2000.12b8  nop
0x2000.12bc  lb     $v0, +0($v0)
0x2000.12c0  nop
0x2000.12c4  beq    $v0, $00, 20001304
0x2000.12c8  nop
0x2000.12cc  lw     $v0, +20($s8)
0x2000.12d0  nop
0x2000.12d4  beq    $v0, $00, 20001220
0x2000.12d8  nop
0x2000.12dc  beq    $00, $00, 20001304
0x2000.12e0  nop
0x2000.12e4  addiu  $v0, $00, +1

Prev. PC:  0x20001280: nop
PC:        0x20001264: lw  $a0, +24($s8)
```

Registers

Register	Value
CPU Registers	
CP0 Registers	
Index	0x00000000
Random	0x00000900
EntryLo	0x00000000
BadVAddr	0x00000000
Timer	0x00000000
EntryHi	0x00000000
Status	0x10000000

```
Initialized process control blocks
allocPcb ok
freed 10 entries
Inserting...
inserted 10 elements
outProc
```

☐ Hardware Failure

Figure: Live Demo

Alternative tricks for debugging in umps3

- The provided file klog.c allows to “print” text and vars. On your code use the klog_print function, then on umps3 add the klog_buffer as a traced region.
- Create an empty function to simulate a break-point

```
#include "klog.c"

void bp() {}

void scheduler() {
    ...
    klog_print("Sono arrivato qua\n");
    bp();
    ...
}
```

Setup tools

You need to install μMPS3.

For example in Debian 12: `sudo apt install umps3`

This will also install the cross-compiler toolchain package: `gcc-mipsel-linux-gnu`

Provided files

These file are provided in a **.tar.gz**:

- `const.h`: constant (`MAXPROC`, **MAXMESSAGES**, etc...) definition file
- `type.h`: struct types (`pcb_t`, `msg_t`, etc...) definition file
- `listx.h`: functions and macros for list manipulation
- `pcb.h`: PCB header file
- `msg.h`: message header file
- `pcb.c`: **write your implementation here**
- `msg.c`: **write your implementation here**
- `Makefile`: file to compile your project
- `p1test.c`: test file

Submission

The deadline is set for **Sunday December 31, 2023 at 23:59**.

Upload a single `phase1.tar.gz` in the folder associated to your group with:

- All the source code with a Makefile
- Documentation
- README and AUTHOR files

Please comment your code!

We will send you an email with the hash of the archive, you should check that everything is correct.

You will receive another email with the score out of 10.