Programming Assignment 4:

Extending a Virtual Memory Implementation with Paging

Mehmet Altunören

22539

1. What Is This?

A minimal, educational "OS" layered on top of our LC-3 VM. It supports:

- **Multiple processes** (up to whatever fits in memory)
- **Simple paging** (32 virtual pages per process, 32 physical frames)
- **Cooperative scheduling** (yield and halt)
- **Basic heap management** via trap (tbrk)

All of it shoe-horned into a standard LC-3 simulator so you can actually see how an OS might wrangle memory and processes.

---

2. Design Goals

1. **Simplicity over cleverness.** If it works and is understandable, it wins.
2. **Explicit over implicit.** No magic. Every bitfield, every shift, every mask is spelled out.
3. **Fail-fast.** Misbehave and we stop immediately with a useful error. No silent corruption.

---

3. Memory Map & Bookkeeping

Region	Addresses	Purpose
mem[0] – mem[2]	Words 0–2	OS globals: current PID, count, status flags
mem[3], mem[4]	Words 3–4	Free-frame bitmaps (16 bits each)

mem[5] – mem[11]	Reserved	(Unused / future expansion)
mem[12] onwards	PCBs	3 words each: PID, PC, PTBR
Physical frames start at	Word 4096	32-word page tables + data pages

- **Free-frame bitmaps** use a 1 to indicate "free." We search for the highest 1-bit, clear it, and that gives us our frame.

PCBs are a flat array:

```
struct PCB { uint16_t pid, pc, ptbr; };
// Located at mem[12 + pid*3]
```

---

4. Paging Mechanics

- **Virtual → Physical Translation:**
    1. Virtual address: 16-bit word
        1. VPN = bits 15–11
        2. Offset = bits 10–0

PTE at mem[PTBR + VPN]:

```
[ Valid (bit 0) | Read (bit 1) | Write (bit 2) | PFN (bits 15–11) ]
```

2. On any load/store, we:
    1. Check vpn ≤ 5 (OS region) → segfault.
    2. Read PTE → if valid==0 → segfault.
    3. Check read/write permissions → fault if violated.
    4. Build physical address = (PFN << 11) | offset.
- **allocMem(ptbr, vpn, R, W)**
    1. Find a free PFN in mem[3..4].
    2. Clear its bit.
    3. Build PTE = (PFN<<11) | (R?0x2:0) | (W?0x4:0) | 0x1.
    4. Write it to mem[ptbr + vpn].

- **freeMem(vpn, ptbr)**
    1. Check PTE.valid.
    2. Extract PFN, set its bitmap bit back to 1.
    3. Clear valid bit in PTE.

---

5. Process Lifecycle

5.1 Creation

createProc(codeFile, heapFile) does:

1. Check OS status (full PCB list?).
2. pid = mem[1]++; pcb = 12 + pid*3; ptbr = 4096 + pid*32;
3. Initialize PCB: PID, PC=0x3000, PTBR.
4. Allocate two code pages (VPNs 6,7; read-only) and two heap pages (VPNs 8,9; read/write).
5. ld_img() into those frames, using get_file_size() to know how many words to read.

5.2 Running & Context Switch

The main loop is unchanged:

```
while (running) {
  instr = mr(PC++);
  op_ex[OPC(instr)](instr);
}
```

- **Yield (tyld trap)**:
    1. Save reg[RPC] into PCB.
    2. Scan PCBs for next non-terminated PID.
    3. Load its PC and PTBR.
    4. Print a switch message if pid changed.
- **Halt (thalt trap)**:
    1. Mark PCB invalid.
    2. Free *all* its pages via freeMem().
    3. Find next PID or set running=false.

---

6. Trap Interface & Heap Adjustment

- **tbrk()** inspects bits in R0:
  - Bit 0 = 1 → allocate new heap page at that VPN.
  - Bit 0 = 0 → free the page.
  - Prints status messages for clarity.
- Other LC-3 traps (getchar, putchar, etc.) piggy-back unchanged.

---

7. Error Handling

Never trust the process to behave:

- Segfaults (OS region, invalid pages) immediately kill running.
- Permission faults print a descriptive error.
- Out-of-frames or PCB overflow stops process creation with a message.

---

8. Testing

- **Edge Cases:** creating more than 32 pages total, double free of a page, invalid page access.
- **Concurrent I/O:** Obviously non-preemptive—if one process sits in an infinite loop without a yield, others starve.
- **Max Processes:** Limited by how many PCBs you reserved; you'll hit OSStatus when PCB area fills.

9. Summary

This is not production code—it's an **educational sandbox** showing you:

1. **How paging tables live in memory.**
2. **How simple bitmaps manage free frames.**
3. **How context-save/restore** enables multiple "processes" on a single core.
4. **How trap routines** can implement system calls (hello, tbrk).

Simplicity and clarity win over "clever hacks" any day.