Assignment 3 - Report

Group Members
Surajprakash Narwariya [2024JCS2044]
Anand Sharma [2024JCS2049]

Course - Operating Systems [COL633]

April 30, 2025

Contents

1 Introduction

In this assignment we extend xv6 with two major capabilities:

1. **Memory Printer:** Prints the number of user-space pages resident in RAM for each active process (PID ≥ 1) in the `SLEEPING`, `RUNNABLE`, or `RUNNING` state. Invoked via `Ctrl+I`.
2. **Page Swapping:** When physical memory usage exceeds a threshold, pages are evicted to a dedicated swap partition on disk, using an adaptive page-replacement policy parameterized by α and β.

2 Memory Printer

The Memory Printer feature allows a user to inspect, at runtime, how many user-space pages each active process has resident in physical memory. It is invoked by pressing `Ctrl+I` at the console prompt.

2.1 Handler Registration

In the kernel's console input loop (in `console.c`), we add a case for the ASCII code corresponding to `Ctrl+I`. When that key is detected, we call the `memprint()` function. This registration ensures that the normal keyboard input dispatch mechanism triggers our memory-printing routine without altering any other console behavior.

2.2 memprint() Logic

The `memprint()` function (implemented in `proc.c`) performs the following steps:

1. It begins by printing a header line to the console indicating that `Ctrl+I` was detected.
2. It then prints column titles ("PID" and "NUM_PAGES") to label the output.
3. It iterates over the global process table array (`ptable.proc`), examining each `proc` structure.
4. For each entry, it checks two conditions:
    - The process ID (`pid`) is at least 1, ensuring we only include user-level processes.
    - The process state is one of `SLEEPING`, `RUNNABLE`, or `RUNNING`, so that only active processes are reported.

5. For each qualifying process, it computes the number of pages by rounding up the process's address space size (`sz`) to the nearest page boundary and dividing by the page size (`PGSIZE`).
6. It prints one line per process, showing the process's PID and its resident page count.
7. After printing all entries, control returns to the interrupted console command, allowing the user to continue as if nothing had changed except for the printed report.

This design provides a lightweight, non-invasive method to monitor per-process memory usage on xv6 without requiring external tools or halting the system.

3 Swapping Implementation in xv6

To add swapping to xv6, we modify the on-disk layout, introduce in-kernel data structures to track swap slots, implement swap-out and swap-in paths, and integrate with the physical memory allocator.

3.1 Disk Layout Modifications

To accommodate swapping, we extend the file-system layout in `mkfs.c` by carving out a dedicated swap region immediately after the superblock. We reserve

$$\texttt{NSWAPBLOCKS} = \texttt{NSWAPSLOTS} \times \texttt{BLOCKS\_PER\_PAGE}$$

disk blocks for swap. The code iteratively computes the total number of metadata blocks until convergence:

$$\begin{aligned}
\text{ninodeblocks} &= \lceil NINODES/IPB \rceil, \\
n\_log &= LOGSIZE, \\
nmeta &= 2 + n\_log + \text{ninodeblocks} + nbitmap, \\
nblocks &= FSSIZE - nmeta - NSWAPBLOCKS, \\
nbitmap &= \left\lceil \frac{nblocks}{BSIZE \times 8} \right\rceil.
\end{aligned}$$

Once `nmeta` stabilizes, we set the superblock fields as follows:

- `sb.nblocks` is assigned the usable data blocks (`nblocks`), excluding swap.
- `sb.logstart` begins at block #2 plus the swap region, so the write-ahead log does not overlap swap.

- `sb.inodestart` and `sb.bmapstart` follow sequentially after the log and inode regions.
- `freeblock` is initialized to `nmeta + NSWAPBLOCKS`, pointing at the first free data block after swap.

This layout guarantees that:

1. Swap blocks occupy disk sectors #2 through #(2+`NSWAPBLOCKS`-1).
2. The write-ahead log, inode blocks, and free-block bitmap are correctly positioned immediately afterward.
3. The kernel's `freeblock` pointer skips over the swap area when allocating new files.

3.2 In-Kernel Swap Data Structures

Inside the kernel, we represent the swap space with an array of `NSWAPSLOTS` entries, each capable of storing one virtual page (composed of `BLOCKS_PER_PAGE` contiguous disk blocks). We introduce:

- A new PTE flag `PTE_SWAP` $= 0x200$, marking page-table entries whose page is resident on disk.
- A `struct swap_slot` containing:
  - `page_perm`, which records the original PTE permission bits (`PTE_U` and `PTE_W`) before eviction.
  - `is_free`, a Boolean indicating whether the slot is available.
- The global array `swap_slots[NSWAPSLOTS]` to manage all slots.

At system boot (in `main.c`), `swap_init()` is invoked to mark every slot as free, ensuring a clean starting state. Slot allocation via `swap_alloc(perm)` scans this array for the first free entry, sets its `page_perm` to `perm`, marks it non-free, and returns its index. Conversely, `swap_free(idx)` resets the slot's `is_free` flag, making it available again.

To transfer data between memory and disk, we provide:

- `swap_write(idx,page)`, which writes the entire page to disk sectors starting at #$(2 +$ `BLOCKS_PER_PAGE` $\times\, idx)$, bypassing the log layer to avoid write-ahead overhead.
- `swap_read(idx,page)`, which reverses the operation, restoring the page contents back into kernel memory.

All operations on the `swap_slots` array are protected by a global spinlock (declared in `pageswap.c`), ensuring atomic slot allocation and freeing in the presence of concurrent page-fault handlers and allocator threads.

3.3 Swap-Out Procedure

The `swap_out()` function reclaims physical memory by evicting pages to disk. It executes under a global spinlock to ensure atomicity:

1. **Victim Process Selection.** Iterate over every process in the process table (`ptable.proc`). For each process that is not `UNUSED`, compare its `rss` (resident set size) against the current maximum. The process with the largest `rss` is chosen as the victim; ties are broken in favor of the lower PID. This heuristic targets the heaviest memory user for eviction.

2. **Victim Page Selection.** Within the victim process's address space, scan all user-space virtual addresses from 0 up to `KERNBASE` in page-sized strides. For each address, obtain its page-table entry (PTE).

   - If the page is present (`PTE_P`=1) but has not been recently accessed (`PTE_A`=0), select it immediately as the eviction candidate.
   - Otherwise, clear its accessed bit (`PTE_A` $\leftarrow$ 0) and remember the first qualifying page as a fallback.

   This approximates an LRU policy by preferring pages that have not been used recently.

3. **Swap Slot Allocation.** Invoke `swap_alloc(perm)` passing the victim page's original user/write permission bits. This returns a free slot index in the swap region. If no slot is available, the system panics, as swapping cannot proceed.

4. **Writing the Page to Disk.** Extract the physical frame address from the PTE via `PTE_ADDR`. Convert it to a kernel-virtual pointer (`P2V(pa)`) and call `swap_write(slot, mem)`. This writes all constituent blocks of the page to the reserved swap region on disk, bypassing the file-system log to avoid extra write-ahead overhead.

5. **Updating the Page Table and TLB.** Replace the victim PTE with a new entry encoding the swap-slot index (shifted left 12 bits) OR'd with `PTE_SWAP`. Then flush the corresponding TLB entry by invoking `invlpg` on the page's virtual address, ensuring no stale mapping remains.

6. **Freeing the Physical Frame and Adjusting RSS.** Call `kfree(mem)` to return the physical page to the free-list. Decrement the victim process's `rss` counter to reflect one fewer resident page.

3.4 Swap-In (Page Fault) Handling

When a process accesses a swapped-out page, the hardware raises a page-fault (`T_PGFLT`). The kernel's trap handler checks if the faulting PTE has `PTE_SWAP` set; if so, it calls `handle_pgfault(pgdir, va)`:

1. **Allocating a New Frame.** Invoke `kalloc()`. If no free frame is available, call `swap_out()` to evict another page, then retry `kalloc()`. Panic if allocation still fails.
2. **Reading from the Swap Slot.** Extract the slot index from the PTE (upper bits of its address field). Call `swap_read(slot, mem)` to copy the page contents from disk back into the newly allocated kernel buffer.
3. **Restoring the Page Table Entry.** Construct a new PTE by combining the physical-frame address (`V2P(mem)`), the present bit (`PTE_P`), and the original permission bits saved in `swap_slots[slot].page_perm`. Update the PTE and flush the TLB via `invlpg(va)`.
4. **Freeing the Swap Slot and Updating RSS.** Invoke `swap_free(slot)` to mark the disk slot free again. Increment the current process's `rss` counter to reflect the newly resident page.

3.5 Integration with the Allocator

In `kalloc()` (`kalloc.c`):

- If `freelist` is empty, release locks and call `swap_out()`, then retry.
- Ensures that xv6 reclaims memory under pressure instead of panicking.

4 Adaptive Page Replacement Strategy

We embed a feedback-driven eviction mechanism that dynamically adjusts both the free-page threshold (Th) and the number of pages evicted (N_{pg}) based on runtime memory pressure.

4.1 Parameters and Initialization

- $Th_0 = 100$: initial free-page threshold.
- $N_{\text{pg},0} = 2$: initial pages to evict on trigger.
- $\alpha = 25,\ \beta = 10$: tuning constants (via `-DALPHA=25 -DBETA=10`).
- `kmem.free_pages` tracks the current count of free frames, initialized in `kinit2()`.

4.2 Eviction Recurrences

After each successful `kalloc()` decrements `kmem.free_pages` below the current threshold Th_i, the following steps execute:

$$\textbf{(1) Evict:}\quad \underbrace{\texttt{swap\_out()}}_{N_{\text{pg},i}\text{ times}}$$

$$\textbf{(2) Update threshold:}\quad Th_{i+1} = \left\lfloor Th_i \times \left(1 - \tfrac{\beta}{100}\right)\right\rfloor$$

$$\textbf{(3) Update eviction count:}\quad N_{\text{pg},\,i+1} = \min\Big(LIMIT,\ \lfloor N_{\text{pg},\,i} \times (1 + \tfrac{\alpha}{100})\rfloor\Big)$$

$$\textbf{(4) Log status:}\quad \texttt{Current\_free=kmem.free\_pages},\quad Th = Th_{i+1},\quad N_{\text{pg}} = N_{\text{pg},\,i+1}.$$

4.3 Impact of α and β

- α (growth factor for N_{pg}):
  - *Low* α (e.g. 5–10%): slow increase in eviction batch size, smoothing I/O but potentially reacting too slowly under sudden memory pressure.
  - *High* α (e.g. 50%+): rapid growth in N_{pg}, quickly reclaiming memory but risking high disk throughput and contention.
- β (decay factor for Th):
  - *Low* β (e.g. 5%): threshold remains relatively high after eviction, delaying subsequent swaps—this can reduce swap frequency but may lead to thrashing if active pages are evicted too aggressively.
  - *High* β (e.g. 20%+): threshold drops sharply, triggering more frequent evictions—this maintains a low memory footprint but increases CPU stalls and I/O overhead.

Balancing α and β is critical to optimizing across three dimensions:

1. *Swap I/O throughput*: total disk bandwidth consumed by swap.
2. *CPU stall time*: delays incurred on page faults and during eviction.
3. *Working-set preservation*: minimizing eviction of actively used pages.