Memory Printer and Adaptive Page Swapping in xv6

Umesh Kumar (2022CS11115)
Aditya Sahu (2022CS11113)

Operating Systems (COL331), Spring 2025

Abstract

We present two major enhancements to the xv6 teaching kernel:

1. A *Memory Printer* facility that, upon user request, reports each active process's resident set size (RSS) in pages.
2. An *Adaptive Page Swapping* subsystem that automatically moves pages to and from disk under a fixed 4 MiB physical memory constraint, using a feedback-driven eviction policy parameterized by α and β.

Detailed design, implementation snippets, and parameter analysis are provided.

1 Introduction

In modern operating systems, understanding per-process memory usage and gracefully handling memory overcommit are core responsibilities. We augment xv6 with:

- **Memory Printer:** On pressing `Ctrl+I`, the kernel prints "PID NUM_PAGES" for each user process in RAM.
- **Adaptive Swapping:** When free pages fall below a threshold T_h, the kernel swaps out N_{pg} pages, then reduces T_h by factor $(1 - \beta/100)$ and grows N_{pg} by $(1 + \alpha/100)$, all controlled via Makefile macros.

Our modifications touch the console driver, trap handler, page allocator, VM system, fs/mkfs, and a new `pageswap.c` module.

2 Memory Printer

2.1 Console Interrupt Hook

In `console.c`, we detect ASCII 9 (`Ctrl+I`) in the input loop:

```
// console.c
...
case C('I'):
  memory_printer();
  break;
...
```

2.2 Memory Printer Routine

Defined in `proc.c`, this function locks the process table, iterates active procs, and counts PTE_P bits:

```
// proc.c

// Count resident pages for a process
static int
count_resident_pages(struct proc *p) {
  int cnt = 0;
  for (uint va = 0; va < p->sz; va += PGSIZE) {
    pte_t *pte = walkpgdir(p->pgdir, (void*)va, 0);
    if (pte && (*pte & PTE_P))
      cnt++;
  }
  return cnt;
}

// Print header and per-PID counts
void
memory_printer(void) {
  cprintf("Ctrl+I is detected by xv6\nPID NUM_PAGES\n");
  acquire(&ptable.lock);
  for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->pid >= 1 &&
        (p->state==SLEEPING || p->state==RUNNABLE || p->state==RUNNING)) {
      cprintf("%d %d\n", p->pid, count_resident_pages(p));
    }
  }
  release(&ptable.lock);
}
```

3 Adaptive Page Swapping

3.1 Disk Partitioning in mkfs.c

We reserve 800 slots of 8 sectors each immediately after the superblock:

```
// mkfs.c
// superblock layout adjustments:
sb.swapstart = xint(2);              // block 2
sb.swapsize  = xint(800 * 8);        // 800 slots * 8 blocks
sb.logstart  = xint(sb.swapstart + sb.swapsize);
```

We then write the updated superblock to sector 1.

3.2 Swap Slot Data Structure

In pageswap.h:

```
// pageswap.h
#define NUM_SLOTS 800
#define BLOCKS_PER_SLOT 8
#define PTE_S 0x200  // custom swapped-out flag

struct swapslot {
  int page_perm;  // original PTE flags
  int is_free;    // 1 if slot is unused
};
```

3.3 Initialization at Boot

In fs.c, called from main():

```
// fs.c
void
init_swap(void) {
  swapstart = 2; // as per superblock
  for (int i = 0; i < NUM_SLOTS; i++) {
    swap_slots[i].is_free = 1;
    swap_slots[i].page_perm = 0;
  }
}
```

3.4 Page Fault Handler

In trap.c, under case T_PGFLT:

```
// trap.c
case T_PGFLT: {
```

```
  uint va = rcr2();
  struct proc *p = myproc();
  pte_t *pte = walkpgdir(p->pgdir, (void*)va, 0);
  if (pte && (*pte & PTE_S)) {
    uint slot = *pte >> 12;
    swap_in(p, va, slot);
    return;
  }
  // otherwise: kill as usual
  p->killed = 1;
} break;
```

3.5 Swapping Logic in pageswap.c

```
// pageswap.c

// Write 8 blocks of the page to disk
static void
write_to_swap(char *mem, int slot) {
  for (int j = 0; j < BLOCKS_PER_SLOT; j++) {
    struct buf *b = bread(ROOTDEV, swapstart + slot*8 + j);
    memmove(b->data, mem + j*BSIZE, BSIZE);
    bwrite(b);
    brelse(b);
  }
}

// Find and reserve a free slot
int
find_free_slot(void) {
  for (int i = 0; i < NUM_SLOTS; i++)
    if (swap_slots[i].is_free) {
      swap_slots[i].is_free = 0;
      return i;
    }
  panic("No free swap slots");
}

// Free a slot when page is either reloaded or process exits
void
free_swap_slot(uint slot) {
  swap_slots[slot].is_free = 1;
  swap_slots[slot].page_perm = 0;
}

// Swap out a single page
```

```
void
swap_out_one_page(void) {
  // 1. Find victim process by max rss
  struct proc *victim = 0; int maxrss = -1;
  acquire(&ptable.lock);
  for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if ((p->state==RUNNING||p->state==RUNNABLE||p->state==SLEEPING)
        && p->rss>maxrss) {
      victim = p; maxrss = p->rss;
    }
  }
  if (!victim) panic("swap_out: no victim");
  // recompute rss
  int cnt=0;
  for (uint a = 0; a < victim->sz; a += PGSIZE)
    if ((walkpgdir2(victim->pgdir,(void*)a,0)) &&
        (*walkpgdir2(victim->pgdir,(void*)a,0)&PTE_P))
      cnt++;
  victim->rss = cnt;
  release(&ptable.lock);

  // 2. Select a page: first with PTE_P=1,PTE_A=0
  uint va; pte_t *pte;
  for (va=0; va<victim->sz; va+=PGSIZE) {
    pte = walkpgdir2(victim->pgdir,(void*)va,0);
    if (pte && (*pte & PTE_P) && !(*pte & PTE_A)) break;
    if (pte) *pte &= ~PTE_A; // second chance
  }
  if (va>=victim->sz) panic("swap_out: no page");

  // 3. Write out
  int slot = find_free_slot();
  char *mem = (char*)P2V(*pte & ~0xFFF);
  swap_slots[slot].page_perm = *pte & 0xFFF;
  write_to_swap(mem, slot);

  // 4. Update PTE
  *pte = (slot<<12) | PTE_S;
  victim->rss--;
  kfree(mem);
}

// Called by kalloc() when free pages <= Th
void
swap_out_if_needed(void) {
  if (get_free_pages() <= Th) {
    cprintf("Current Threshold = %d, Swapping %d pages\n", Th, Npg);
```

```
    for (int i=0; i<Npg; i++) swap_out_one_page();
    Th   = (Th*(100-BETA))/100;
    Npg  = min(LIMIT, (Npg*(100+ALPHA))/100);
  }
}

// Reload a swappedout page on fault
void
swap_in(struct proc *p, uint va, uint slot) {
  swap_out_if_needed();
  char *mem = kalloc();
  read_from_swap(mem, slot);
  pte_t *pte = walkpgdir2(p->pgdir,(void*)va,0);
  *pte = V2P(mem) | swap_slots[slot].page_perm | PTE_P;
  p->rss++;
  swap_slots[slot].is_free = 1;
}
```

4 Parameter Analysis

The adaptive controller is specified by:

$$\alpha \in [0, 100], \quad \beta \in [0, 100], \quad T_h(0) = 100, \quad N_{pg}(0) = 2, \quad \text{LIMIT} = 100.$$

Whenever the free-page count $F \leq T_h$, the kernel:

1. Logs "Current Threshold = T_h, Swapping N_{pg} pages."
2. Calls `swap_out_one_page()` exactly N_{pg} times.
3. Updates

$$T_h \leftarrow \left\lfloor T_h \left(1 - \tfrac{\beta}{100}\right) \right\rfloor, \quad N_{pg} \leftarrow \min\bigl(\text{LIMIT}, \left\lfloor N_{pg} \left(1 + \tfrac{\alpha}{100}\right) \right\rfloor\bigr).$$

4.1 Effect of α (Growth Factor)

- **Low α (e.g. 0–10):**
  - Batch size N_{pg} grows slowly (or not at all).
  - *Pros:* Minimizes I/O burst sizes, preserves working set longer.
  - *Cons:* Under-eviction on sustained high pressure → repeated page faults.
- **Moderate α (20–40):**
  - Balanced growth: each eviction round frees more pages than the last, adapting to severity.

– Typically yields good throughput in mixed workloads.

- **High α (50–100):**
  – Aggressive growth (doubling or more).
  – *Pros:* Quickly drains memory under pathological pressure.
  – *Cons:* Large I/O spikes, potential to evict pages that will soon be reused ("cache thrash").

4.2 Effect of β (Decay Factor)

- **Low β (0–10):**
  – Threshold T_h remains near its initial value.
  – Evictions occur rarely but in large batches (once triggered).
  – Risks *longer stalls* before any swapping kicks in.
- **Moderate β (10–30):**
  – Gradual lowering of T_h, leading to more frequent but smaller swap bursts.
  – Smooths out memory pressure handling.
- **High β (40–100):**
  – Rapid decay of T_h—subsequent rounds trigger almost immediately, potentially over-evicting.
  – May cause repeated small I/O writes, lowering effective throughput.

4.3 Tuning Trade-Offs

This controller behaves like a proportional feedback loop:

$$\Delta N_{pg} \propto \alpha, \quad \Delta T_h \propto -\beta.$$

One tunes (α, β) to balance:

- *Fault Latency:* Time a process waits on a page-in (smaller N_{pg} and lower β help).
- *I/O Efficiency:* Amortization of each disk write (larger N_{pg} and higher β help).
- *Working-Set Preservation:* Risk of evicting soon-to-be-used pages (lower α, β).

Empirical tuning in xv6 with $\alpha = 25, \beta = 10$ provides smooth behavior under typical academic workloads.