COL331 Operating Systems Assignment 3

Abhinav Rajesh Shripad Entry Number: 2022CS11596
Jahnabi Roy Entry Number: 2022CS11094

April 30, 2025

Memory Printer in xv6

0.1 Ctrl + I Implementation

- For this implementation, like in the last assignment changes were made to the `console.c` file for recognising the keyboard interrupt.
- Changes were made to the `proc.c`. The ctrl_memoryprint function prints the detected keys and prints PID and the NUM_PAGES.

The main relevant code sections are as follows:
In `console.c`:

```c
void consoleintr(int (*getc)(void))
{
int to_memoryprint = 0;
(...)
case C('I'):
to_memoryprint = 1;
break;
(...)
if (to_memoryprint) {
    ctrl_memoryprint();
}
(...)
}
```

In `proc.c`:

```c
void ctrl_memoryprint(void)
{
  struct proc *p;
  int num_pages;
  acquire(&ptable.lock);
  cprintf("\nCtrl+I is detected by xv6\n");
  cprintf("PID NUM_PAGES\n");
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid > 0)
    {
      num_pages = PGROUNDUP(p->sz) / PGSIZE;
      cprintf("%d %d\n", p->pid, num_pages);
    }
  }
  release(&ptable.lock);
}
```

Page Swapping in xv6

0.2 Implementation Details

- System calls, `getrss` and `getNumFreePages` were implemented as according to implementation done for assignment 1.

- `find_victim_process` defined in `proc.c`, selects a victim process — i.e., one from which a page can be evicted — based on highest resident set size (RSS). `find_victim_page` defined in `proc.c` implements a Second-Chance (Clock)-style page replacement policy: it tries to find a page that hasn't been accessed recently. If all pages are accessed, give some of them a "second chance" by clearing PTE_A, then try again.
- Changes made in `kalloc.c`, `exec.c`, `mkfs.c`, `fs.c` and `trap.c` to implement the copy-on-write mechanism.
- This implementation adds dynamic swapping to xv6 by monitoring free physical memory and triggering page swaps when it falls below a threshold. The `kalloc()` function checks the number of free pages and, if low, calls `swapOut()` to free up space by evicting a page. `swapOut()` selects a victim process with the highest memory usage ('rss') using `find_victim_process()`, and within that process, it finds a suitable page to swap out using `find_victim_page()`, preferring cold (unaccessed) pages. The selected page is written to disk, its metadata is updated, and its physical memory is freed. This implements an adaptive, LRU-like swapping mechanism to maintain system responsiveness under memory pressure.
- `pageswap.c` was created and the code for the same is shown below.

The main relevant code is as follows:
In `sysproc.c`:

```c
(...)
int sys_getNumFreePages(void)
{
  return num_of_FreePages();
}

int sys_getrss()
{
  print_rss();
  return 0;
}
(...)
```

In `proc.c`:

```c
struct proc *find_victim_process(void)
{
  struct proc *p;
  struct proc *victim_p = NULL;
  int pid = 100000;
  uint highest_rss = 0;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid < 2)
      continue;

    if (p->rss >= highest_rss)
    {
      if (p->pid < pid)
      {
        pid = p->pid;
      }

      victim_p = p;
      highest_rss = p->rss;
    }
  }
  release(&ptable.lock);

  return victim_p;
}

uint find_victim_page(struct proc *p)
{
  uint i;
```

```
  uint sz = p->sz;
  pte_t *pte;
  uint count = 0;

  for (i = 0; i < sz; i += PGSIZE)
  {
    pte = walkpgdir(p->pgdir, (void *)i, 0);
    if ((*pte & PTE_P) && !(*pte & PTE_A) && (*pte & PTE_U))
    {
      p->rss -= PGSIZE;
      P2V(PTE_ADDR(*pte)));
      return i;
    }
    if (*pte & PTE_P)
      count++;
  }

  count = (count / 10) + 1;

  for (i = 0; i < sz; i += PGSIZE)
  {
    pte = walkpgdir(p->pgdir, (void *)i, 0);
    if ((*pte & PTE_P) && (*pte & PTE_A) && (*pte & PTE_U))
    {
      *pte &= ~PTE_A;
      count--;
    }
    if (!count)
      break;
  }
  return find_victim_page(p);
}
```

In exec.c:

```
(...)
curproc->rss = curproc->sz;
(...)
```

In kalloc.c

```
(...)

struct
{
    struct spinlock lock;
    int use_lock;
    uint num_free_pages;
    struct run *freelist;
} kmem;

struct
{
    int use_lock;
    struct spinlock lock;
    int ref[PHYSTOP / PGSIZE];
} rmap;

int getRmapRef(uint pa)
{
    if (rmap.use_lock)
        acquire(&rmap.lock);
    int num = rmap.ref[pa / PGSIZE];
    if (rmap.use_lock)
        release(&rmap.lock);
    return num;
}

void setRmapRef(uint pa, int val)
{
    if (rmap.use_lock)
        acquire(&rmap.lock);
    rmap.ref[pa / PGSIZE] = val;
    if (rmap.use_lock)
```

```c
            release(&rmap.lock);
    }

    void incRmapRef(uint pa)
    {
        if (rmap.use_lock)
            acquire(&rmap.lock);
        ++rmap.ref[pa / PGSIZE];
        if (rmap.use_lock)
            release(&rmap.lock);
    }

    void decRmapRef(uint pa)
    {
        if (rmap.use_lock)
            acquire(&rmap.lock);
        --rmap.ref[pa / PGSIZE];
        if (rmap.use_lock)
            release(&rmap.lock);
    }

    void kinit1(void *vstart, void *vend)
    {
        (..)
        initlock(&rmap.lock, "rmap");
        (..)
        rmap.use_lock = 0;
        (..)
    }

    void kinit2(void *vstart, void *vend)
    {
        (..)
        rmap.use_lock = 1;
        (..)
    }

    (...)

    void kfree(char *v)
    {
        (...)
        decRmapRef(V2P(v));
        if (getRmapRef(V2P(v)) > 0)
        {
            return;
        }

        (..)
        kmem.num_free_pages += 1;
        (..)
    }

    int Th = THRESHOLD;
    int Npg = NPG;

    char *
    kalloc(void)
    {
        struct run *r;
        int free = kmem.num_free_pages;

        if (free <= Th)
        {
            cprintf("Current Threshold = %d, Swapping %d pages\n", Th, Npg);
            for (int i = 0; i < Npg; i++)
            {
                swapOut();
            }
            // Th = Th * (100 - BETA) / 100;
            // Npg = Npg * (100 + ALPHA) / 100;
            Th -= (Th * BETA) / 100;
            Npg += (Npg * ALPHA) / 100;
```

```c
            if (Npg > THRESHOLD)
                Npg = THRESHOLD;
        }

        if (kmem.use_lock)
            acquire(&kmem.lock);
        r = kmem.freelist;
        if (r)
        {
            kmem.freelist = r->next;
            kmem.num_free_pages -= 1;
            setRmapRef(V2P(r), 1);
        }

        if (kmem.use_lock && 1)
            release(&kmem.lock);

        if (SWAPON)
        {
            if (!r)
            {
                swapOut();
                return kalloc();
            }
        }

        return (char *)r;
    }

    uint num_of_FreePages(void)
    {
        acquire(&kmem.lock);
        uint num_free_pages = kmem.num_free_pages;
        release(&kmem.lock);
        return num_free_pages;
    }
```

In mkfs.c

```c
    (...)
    nmeta = 2 + SWAPBLOCKS + nlog + ninodeblocks + nbitmap;
    nblocks = FSSIZE - nmeta;
    sb.size = xint(FSSIZE);
    sb.nblocks = xint(nblocks);
    sb.ninodes = xint(NINODES);
    sb.nlog = xint(nlog);
    // sb.logstart = xint(2);
    sb.logstart = xint(SWAPBLOCKS + 2);
    // sb.inodestart = xint(2+nlog);
    sb.inodestart = xint(2 + SWAPBLOCKS + nlog);
    // sb.bmapstart = xint(2+nlog+ninodeblocks);
    sb.bmapstart = xint(2 + SWAPBLOCKS + nlog + ninodeblocks);
    sb.swapblocks = xint(SWAPBLOCKS);
    sb.swapstart = xint(2);

    printf("nmeta %d (boot, super, swap blocks %u log blocks %u inode blocks %u, bitmap blocks %u) blocks %d total %d\n",
         nmeta, SWAPBLOCKS, nlog, ninodeblocks, nbitmap, nblocks, FSSIZE);

    (..)

    void balloc(int used)
    {
      uchar buf[BSIZE];
      int i, j;

      printf("balloc: first %d blocks have been allocated\n", used);

      for (j = 0;; j++)
      {
        int start = j * BSIZE * 8;
        if (start >= used)
          break;
        int end = start + BSIZE * 8;
```

```
        int chunk = end - start;
        if (chunk > used - start)
          chunk = used - start;

        // Read the j-th bitmap block (sector 2 + j)
        rsect(2 + j, buf);
        bzero(buf, BSIZE); // Initialize all bits to 0

        for (i = 0; i < chunk; i++)
        {
          buf[i / 8] |= 0x1 << (i % 8);
        }

        wsect(2 + j, buf);
      }
    }
```

In fs.c

```
    (...)
    cprintf("sb: size %d swapblocks %d nblocks %d ninodes %d nlog %d logstart %d\
inodestart %d bmap start %d\n", sb.size, SWAPBLOCKS, sb.nblocks,
          sb.ninodes, sb.nlog, sb.logstart, sb.inodestart,
          sb.bmapstart);
    (...)
```

In trap.c

```
    (...)
    case T_PGFLT:
        page_fault_handler();
        break;
    (...)
```

In pageswap.c:

```
    (..)
    #define SWAPPAGES (SWAPBLOCKS / 8)

    struct swapinfo
    {
        int pid;
        int page_perm;
        int is_free;
    };

    struct swapinfo swp[SWAPPAGES];

    // buffer

    void swapInit(void)
    {
        for (int i = 0; i < SWAPPAGES; i++)
        {
            swp[i].page_perm = 0;
            swp[i].is_free = 1;
        }
    }

    void swapOut()
    {
        struct superblock sb;
        readsb(ROOTDEV, &sb);
        // find victim page
        struct proc *p = find_victim_process();
        uint victim_page_VA = find_victim_page(p);

        pte_t *victim_pte = walkpgdir(p->pgdir, (void *)victim_page_VA, 0);

        int i;
        // find free swap slot
        for (i = 0; i < SWAPPAGES; ++i)
        {
            if (swp[i].is_free)
```

```
            {
                break;
            }
        }

        swp[i].page_perm = 0;
        swp[i].is_free = 0;
        swp[i].pid = p->pid;

        // update swap slot permissions to match victim page permissions
        swp[i].page_perm = PTE_FLAGS(*victim_pte);
        char *pa = (char *)P2V(PTE_ADDR(*victim_pte));
        uint addressOffset;
        for (int j = 0; j < 8; ++j)
        {
            addressOffset = PTE_ADDR(*victim_pte) + (j * BSIZE);
            struct buf *b = bread(ROOTDEV, sb.swapstart + (8 * i) + j);
            memmove(b->data, (void *)P2V(addressOffset), BSIZE);
            b->blockno = (sb.swapstart + (8 * i) + j);
            b->flags |= B_DIRTY;
            b->dev = ROOTDEV;
            bwrite(b);
            brelse(b);
        }
        (*victim_pte) = ((sb.swapstart + (8 * i)) << 12) & (~0xFFF);
        *victim_pte &= ~PTE_P;
        lcr3(V2P(p->pgdir));
        kfree(pa);
    }

    void swapIn(char *memory)
    {
        struct proc *p = myproc();
        uint addr = rcr2();
        pde_t *pd = p->pgdir;
        pte_t *pg = walkpgdir(pd, (void *)(addr), 0);

        uint swapSlot = (*pg >> 12); // physical address of swap block
        int swapIdx = (swapSlot - 2) / 8;
        *pg = (V2P((uint)memory) & (~0xFFF)) | swp[swapIdx].page_perm;
        // int swapSlot = swap;          // swap block number (convert it to integer)
        for (int i = 0; i < 8; i++) // writes page into physical memory
        {
            struct buf *b = bread(ROOTDEV, swapSlot + i);
            cprintf("");
            memmove(memory, b->data, BSIZE);
            brelse(b);
            memory += BSIZE;
        }
        swp[swapIdx].is_free = 1;
        swp[swapIdx].pid = 0;
    }

    void freeSwapSlot(int pid)
    {
        struct superblock sb;
        readsb(ROOTDEV, &sb);
        int i;
        for (i = 0; i < SWAPPAGES; i++)
        {
            if (swp[i].pid == pid)
                break;
        }
        for (int j = 0; j < 8; j++)
        {
            struct buf *b = bread(ROOTDEV, sb.swapstart + (8 * i) + j);
            memset(b->data, 0, BSIZE);
            brelse(b);
        }
        swp[i].is_free = 0;
        swp[i].pid = 0;
    }
```

0.3 Role of α and β in efficiency

This analysis examines how the parameters α (swap-out growth rate) and β (threshold reduction rate) impact the efficiency of the adaptive page replacement strategy. Using experimental data (as available in `swap_efficiency.csv`), we evaluate three key metrics:

- Swap Operations: Number of swap-out cycles
- Total Pages Swapped: Cumulative pages moved to disk
- Threshold Adaptation: Pattern of threshold (Th) reduction

0.3.1 Role of β (Threshold Reduction Rate)

$$Th_{new} = \left\lfloor Th_{current} \cdot \left(1 - \frac{\beta}{100}\right)\right\rfloor \tag{1}$$

Table 1: Impact of β Values ($\alpha = 10$)

β	Swap Ops	Total Pages	Final Th	Behavior
10	31	124	9	Gradual decay, frequent swaps
50	17	68	1	Rapid decay, fewer swaps
100	16	64	0	Threshold collapses immediately

Key observations:

- High β (50-100): Reduces swap frequency but risks premature threshold collapse
- Low β (10-30): Maintains memory pressure awareness at cost of higher I/O

0.3.2 Role of α (Swap-Out Growth Rate)

$$Npg_{new} = \min\left(LIMIT, \left\lfloor Npg_{current} \cdot \left(1 + \frac{\alpha}{100}\right)\right\rfloor\right) \tag{2}$$

Table 2: Impact of α Values ($\beta = 30$)

α	Swap Ops	Total Pages	Final Th	Behavior
10	17	68	3	Conservative growth, stable
50	9	147	7	Aggressive growth, bursty I/O
100	5	124	0	Reaches LIMIT immediately

Key observations:

- High α (50-100): Clears memory quickly but causes I/O spikes
- Moderate α (20-40): Balances swap size and frequency

0.3.3 Combined Impact Analysis

Table 3: Interaction Effects of α and β

α	β	Swap Ops	Total Pages	Final Th	Profile
10	10	31	124	9	Conservative
50	50	6	79	1	Aggressive
30	30	8	74	10	Balanced

0.3.4 Optimization Recommendations

- **Stable Workloads**: $\alpha = 20\text{-}40$, $\beta = 30\text{-}50$
- **Bursty Workloads**: $\alpha = 50\text{-}70$, $\beta = 10\text{-}20$
- **Avoid**: $\alpha > 80$ (I/O spikes) or $\beta > 80$ (threshold collapse)

0.3.5 Conclusion

The optimal efficiency is achieved with:

- Moderate α (30-50) to balance swap size and frequency
- Moderate β (20-40) to maintain memory pressure awareness
- Combined tuning using the relationship:

$$\text{Efficiency} \propto \frac{1}{\text{Swap Ops} \times \text{Total Pages}} \tag{3}$$