Operating Systems
Assignment 3 - Easy

Vaibhav Katendra (2024MCS2459)
Adithya R Narayan (2024MCS2445)

April 30, 2025

1 Introduction

This report documents the implementation of two key features in xv6: a Memory Printer and a Page Swapping mechanism. The Memory Printer displays the number of pages allocated to different processes upon pressing Ctrl+I. The Page Swapping implementation enables xv6 to move pages between RAM and disk when memory becomes scarce, using an adaptive page replacement policy. This report details our approach, implementation challenges, and the performance characteristics of our solution.

2 Memory Printer Implementation

We implemented a keyboard handler that responds to Ctrl+I by displaying information about the current memory usage of all processes. This involved modifying the keyboard handling code in `console.c` to detect the specific key combination.

```
// In console.c
if(c == 'I' - '@') {  // Ctrl+I
  cprintf("Ctrl+I is detected by xv6\n");
  cprintf("PID NUM_PAGES\n");
  print_memory_usage();
  break;
}
```

The `print_memory_usage()` function iterates through the process table and computes the number of pages allocated to each process with a state of SLEEPING, RUNNING, or RUNNABLE.

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

To track memory usage accurately, we modified the `struct proc` in `proc.h` to include an `rss` field (resident set size) that counts the number of pages in RAM:

```
struct proc {
  // Existing fields...
  int rss;         // Number of pages in RAM
  // Other fields...
};
```

We update the `rss` counter in several key functions: - During process creation in `exec()` - When memory is allocated via `sbrk()` system call - When memory is freed in `exit()`

2.1 Testing

```
Ctrl+I is detected by xv6
PID NUM_PAGES
1 3
2 4
CTRL + I gives correct output for PID 1
```

3 Swapping Implementation

3.1 Modified Disk Layout

We modified xv6's disk layout to include dedicated swap space between the superblock and log. This required changes to `mkfs.c` to allocate the space and initialize the swap slots:

```

#define SWAPSIZE 800       // Number of swap slots
#define BLOCKSPERSLOT 8    // Blocks needed for one page


struct swap_slot {
  int page_perm;
  int is_free;
};

// Initialize swap area
```

```
struct swap_slot swap_slots[SWAPSIZE];
for(i = 0; i < SWAPSIZE; i++) {
  swap_slots[i].page_perm = 0;
  swap_slots[i].is_free = 1;
}

// Write swap slots to disk
wsect(2, (void*)swap_slots);
```

The swap area consists of 800 slots, each capable of storing one 4KB page (8 disk blocks of 512 bytes). We created a separate module `pageswap.c` to manage these slots.

3.2 Memory Organization

We limited physical memory to 4MB by modifying `memlayout.h`:

```
// In memlayout.h
#define PHYSTOP         0x400000
```

The page swapping functions were implemented in a new file, `pageswap.c`, with the following key functions:

```
// Find a free swap slot
int findFreeSlot(void);

// Swap a page out to disk
int swapAndPageOut (struct proc *p, pde_t *pgdir, uint va);

// Swap a page in from disk
int swapPageIn (struct proc *p, pde_t *pgdir, uint va);

// Find a victim process for swapping
struct proc* findVictimProc ((void);

// Find a victim page in a process
uint findVictimPage ((struct proc *p, pde_t *pgdir);
```

3.3 Page Replacement Policy

Our victim selection algorithm follows two steps:

1. Process Selection: We select the process with the highest number of pages in RAM (highest `rss`), and in case of a tie, we choose the one with the lower PID.

```
struct proc*
findVictimProc(void)
{
  struct proc *victim = 0;
  int maxrss = 0;
  extern struct {
    struct spinlock lock;
    struct proc proc[NPROC];
  } ptable;
  acquire(&ptable.lock);
  struct proc *p = ptable.proc;
  while(p < &ptable.proc[NPROC]) {
    if(p->state != UNUSED && p->pid >= 1) {
      if(p->rss > maxrss || (p->rss == maxrss && victim && p->pid
          < victim->pid)) {
        maxrss = p->rss;
        victim = p;
      }
    }
    p++;
  }
  release(&ptable.lock);
  return victim;
}
```

2. Page Selection: We choose a page that is present in memory (PTE_P set) but has not been recently accessed (PTE_A not set).

```
uint
findVictimPage(pde_t *pgdir, uint *va_out)
{
  uint addr = 0;
  while(addr < KERNBASE) {
    pte_t *pte = walkpgdir(pgdir, (void*)addr, 0);
    if(pte && (*pte & PTE_P) && (*pte & PTE_U) && !(*pte & PTE_A)
       ) {
      *va_out = addr;
      return PTE_ADDR(*pte);
    }
    addr += PGSIZE;
  }
  addr = 0;
  while(addr < KERNBASE) {
    pte_t *pte = walkpgdir(pgdir, (void*)addr, 0);
    if(pte && (*pte & PTE_P) && (*pte & PTE_U))
      *pte &= ~PTE_A;
    addr += PGSIZE;
  }
  lcr3(V2P(pgdir));
  addr = 0;
  while(addr < KERNBASE) {
    pte_t *pte = walkpgdir(pgdir, (void*)addr, 0);
    if(pte && (*pte & PTE_P) && (*pte & PTE_U)) {
      *va_out = addr;
      return PTE_ADDR(*pte);
    }
    addr += PGSIZE;
  }
  return 0;
}
```

3.4 Swapping-out Procedure

The swapping-out procedure involves identifying a victim page, finding a free swap slot, and transferring the page contents to disk:

```
int
swapAndPageOut(pde_t *pgdir, uint va, uint pa)
{
  int slot_index = findFreeSlot();
  if(slot_index < 0)
    return -1;

  uint blockno = SWAP_BLOCK_START + slot_index *
     SWAP_BLOCKS_PER_PAGE;
  pte_t *pte = walkpgdir(pgdir, (void*)va, 0);
  if(!pte)
    return -1;
  if(!(*pte & PTE_P))
    return -1;

  acquire(&swap_area.lock);
  swap_area.slots[slot_index].page_perm = *pte & 0xFFF;
  release(&swap_area.lock);

  for(int i = 0; i < SWAP_BLOCKS_PER_PAGE; i++) {
    struct buf *b = bread(0, blockno + i);
    char *src = (char*)(P2V(pa)) + i*BSIZE;
    memmove(b->data, src, BSIZE);
    bwrite(b);
    brelse(b);
  }
  *pte = (slot_index << 12) | ((*pte) & ~PTE_P & 0xFFF);
  lcr3(V2P(pgdir));
  return 0;
}
```

3.5 Swapping-in Procedure

When a process tries to access a swapped-out page, a page fault occurs. We handle this in the `trap.c` file:

```
// In trap.c
case T_PGFLT:
  addr = rcr2();  // Get faulting address from CR2 register
  if(myproc() && (tf->cs & 3) == 3) {
    // User process caused page fault
    if(swap_in_page(myproc(), myproc()->pgdir, PGROUNDDOWN(addr))
        < 0) {
      // Failed to swap in page, kill process
      cprintf("pid %d %s: memory fault\n", myproc()->pid, myproc
         ()->name);
      myproc()->killed = 1;
    }
    break;
  }
  // Fall through if kernel fault
```

The swap_in_page() function finds the swap slot containing the page, allocates a physical page, and loads the contents from disk:

```
int
swapPageIn(pde_t *pgdir, void *va)
{
  uint page_addr = PGROUNDDOWN((uint)va);
  pte_t *pte = walkpgdir(pgdir, (void*)page_addr, 0);
  if(!pte)
    return -1;
  if(*pte & PTE_P)
    return 0;

  int slot_index = PTE_ADDR(*pte) >> 12;
  if(slot_index < 0 || slot_index >= SWAP_SLOT_TOTAL)
    return -1;
  if(swap_area.slots[slot_index].is_free)
    return -1;

  char *mem = kalloc();
  if(mem == 0) {
    checkAndSwap();
    mem = kalloc();
    if(mem == 0)
      return -1;
  }

  uint blockno = SWAP_BLOCK_START + slot_index *
     SWAP_BLOCKS_PER_PAGE;
  int i = 0;
  while(i < SWAP_BLOCKS_PER_PAGE) {
    struct buf *b = bread(0, blockno + i);
    memmove(mem + i*BSIZE, b->data, BSIZE);
    brelse(b);
    i++;
  }

  uint perm;
  acquire(&swap_area.lock);
  perm = swap_area.slots[slot_index].page_perm;
  release(&swap_area.lock);

  perm |= PTE_P;
  int mapping = mapPages(pgdir, (void*)page_addr, PGSIZE, V2P(mem
     ), perm);
  if(mapping < 0) {
    kfree(mem);
    return -1;
  }
  freeSlot(slot_index);

  struct proc *p = myproc();
  if(p)
    p->rss++;

  return 0;
}
```

3.6 Adaptive Page Replacement Strategy

We implemented the adaptive page replacement strategy as specified in the assignment. This strategy adjusts both the threshold for triggering swapping and the number of pages to swap based on memory pressure:

```
// In pageswap.c
#define limit 100
int threshold = 100;    // Initial threshold
int npages_to_swap = 2;     // Initial number of pages to swap

void
checkAndSwap(void)
{
  int free_pages = countFreePages();
  if(free_pages > threshold)
    return;

  cprintf("Current Threshold = %d, Swapping %d pages\n",
     threshold, npages_to_swap);

  swapOutPages();

  threshold -= (threshold * beta) / 100;
  if(threshold < 1)
    threshold = 1;

  npages_to_swap += (npages_to_swap * alpha) / 100;
  if(npages_to_swap > limit)
    npages_to_swap = limit;
}
```

The values of ALPHA and BETA significantly affect system performance:

- ALPHA (25): Controls how quickly the number of pages to swap increases. A higher value makes the system more aggressive in freeing memory but may lead to thrashing if set too high. Our value of 25 provides a balanced approach, gradually increasing swapping when needed.

- BETA (10): Controls how quickly the threshold decreases. A higher value makes the system more conservative about triggering swapping. Our value of 10 ensures that we don't trigger swapping too frequently but still respond to memory pressure.

This adaptive approach allows the system to become more aggressive in swapping as memory pressure increases, which helps prevent system crashes due to memory exhaustion while minimizing unnecessary swapping during normal operation.

4 Testing

We tested our implementation using the provided `memtest.c` program, which allocates memory until it exhausts the available RAM. With our swapping mechanism enabled, the program successfully completes instead of failing with "Memtest Failed."

The output shows the adaptive behavior of our swapping mechanism:

```
memtest
Current Threshold = 100, Swapping 4 pages
Current Threshold = 90, Swapping 5 pages
Current Threshold = 81, Swapping 6 pages
Current Threshold = 73, Swapping 7 pages
Current Threshold = 66, Swapping 8 pages
Current Threshold = 60, Swapping 10 pages
Current Threshold = 54, Swapping 12 pages
Current Threshold = 49, Swapping 15 pages
Current Threshold = 45, Swapping 18 pages
Current Threshold = 41, Swapping 22 pages
Current Threshold = 37, Swapping 27 pages
Current Threshold = 34, Swapping 33 pages
Current Threshold = 31, Swapping 41 pages
Current Threshold = 28, Swapping 51 pages
Current Threshold = 26, Swapping 63 pages
Current Threshold = 24, Swapping 78 pages
Current Threshold = 22, Swapping 97 pages
Current Threshold = 20, Swapping 100 pages
Current Threshold = 18, Swapping 100 pages
Current Threshold = 17, Swapping 100 pages
Memtest Passed
```

As memory pressure increases, the threshold decreases and the number of pages swapped increases, demonstrating our adaptive policy in action.

5 Impact of α and β on System Efficiency

The parameters α and β play a crucial role in determining the overall efficiency of the page swapping mechanism. Here's an in-depth analysis of their impact:

5.1 Impact of α (ALPHA)

The α parameter controls how aggressively the system increases the number of pages to swap out when memory pressure is detected. With our current value of $\alpha = 25$, the number of pages to swap increases by 25% each time the free memory falls below the threshold.

A higher value of α leads to:

- More aggressive memory reclamation
- Faster response to memory pressure
- Potential for thrashing if set too high

A lower value of α results in:

- More gradual increase in swapping activity
- Less disruption to running processes
- Potentially slower response to severe memory pressure

The optimal value of α depends on the workload characteristics. For workloads with sudden spikes in memory usage, a higher α might be beneficial. For more stable workloads, a lower α could provide better performance by reducing unnecessary swapping.

5.2 Impact of β (BETA)

The β parameter controls how quickly the threshold for triggering swapping decreases. With our current value of $\beta = 10$, the threshold decreases by 10% each time swapping is triggered.

A higher value of β leads to:

- More rapid decrease in the threshold
- More frequent swapping operations
- Potential for maintaining higher free memory

A lower value of β results in:

- More gradual decrease in the threshold
- Less frequent swapping operations
- System operates closer to memory limits

The β parameter effectively controls how conservative the system is about memory management. A higher β makes the system more proactive in maintaining free memory, while a lower β allows processes to utilize more memory before triggering swapping.

5.3 Interaction Between α and β

The interaction between α and β determines the overall behavior of the adaptive swapping mechanism:

- High α, High β: Very aggressive swapping strategy, potentially leading to thrashing but maintaining high free memory
- High α, Low β: Aggressive bursts of swapping but less frequent triggers
- Low α, High β: Frequent but moderate swapping activity

- Low α, Low β: Minimal swapping, allowing processes to use more memory but risking memory exhaustion

Our choice of $\alpha = 25$ and $\beta = 10$ represents a balanced approach that increases swapping aggressiveness moderately as memory pressure grows while not triggering swapping too frequently. This balance aims to maintain system stability while minimizing the performance impact of excessive swapping.

6 Conclusion

Our implementation successfully extends xv6 with memory monitoring and paging capabilities. The Memory Printer provides visibility into process memory usage, while the Page Swapping mechanism allows the system to handle memory pressure gracefully through adaptive page replacement.

The key strength of our implementation is its adaptive nature, which balances the need to free memory with the overhead of swapping. By adjusting both the threshold and the number of pages to swap based on memory pressure, the system can respond appropriately to different workloads.

The values of α and β significantly impact system performance, and finding the optimal values depends on the specific workload characteristics. Our implementation provides a foundation that can be tuned for different usage scenarios by adjusting these parameters through the Makefile.