Operating Systems
Assignment 3 – Easy

Nikhil Gupta (2024JCS2611)
Abhishek Gupta (2024JCS2047)

April 29, 2025

1 Memory Printer

The memory printer functionality is triggered using `Ctrl+I` and prints the number of user-space pages currently in RAM for each active process. This was implemented by modifying `proc.c`, `vm.c`, and `console.c`.

Process Traversal and Page Counting

A new function `memprinter()` was added in `proc.c`:

```
void memprinter(void) {
  acquire(&ptable.lock);
  cprintf("PID NUM_PAGES\n");
  for(struct proc *p = ptable.proc; p < &ptable.proc[NPROC];
      p++) {
    if(p->pid >= 1 && (p->state == RUNNABLE ||
                       p->state == SLEEPING ||
                       p->state == RUNNING)) {
      cprintf("%d   %d\n", p->pid, countprocpages(p->pgdir,
          p->sz));
    }
  }
  release(&ptable.lock);
}
```

This function locks the process table, filters processes based on PID and state, and calls `countprocpages()` to compute resident page counts.

Counting Pages in RAM

The helper function `countprocpages()` was added in `vm.c`:

It walks the page table from virtual address 0 up to the process size (`sz`), which represents the user-space virtual memory layout of the process. This includes memory allocated for code, data, heap, and stack. Kernel memory

```
int countprocpages(pde_t *pgdir, uint sz) {
  int num_pages = 0;
  for(uint va = 0; va < sz; va += PGSIZE) {
    pte_t *pte = walkpgdir(pgdir, (char*)va, 0);
    if(pte && (*pte & PTE_P))
      num_pages++;
  }
  return num_pages;
}
```

starts at `KERNBASE`, so scanning from 0 to `sz` ensures only user-space pages are considered.

The function checks if each page is present in RAM by testing for the `PTE_P` (Present) flag in the page table entry.

Keyboard Interrupt Handling

In `console.c`, the following case was added in the keyboard interrupt handler:

```
case C('I'):  // Ctrl-I
  printkey('I');
  release(&cons.lock);
  memprinter();
  acquire(&cons.lock);
  break;
```

And the message printing was handled with a helper:

```
void printkey(char key) {
  char *ctrl_msg = "Ctrl+";
  for(char *s = ctrl_msg; *s; s++)
    consputc(*s);
  consputc(key);
  char *rem_msg = " is detected by xv6\n";
  for(char *s = rem_msg; *s; s++)
    consputc(*s);
}
```

This setup ensures that the detection message is printed using `consputc()` (which is safe in interrupt context), and the memory printer runs without disrupting the current process state.

2 Swapping Mechanism in xv6

To support virtual memory under constrained physical memory, a swapping mechanism is implemented in `xv6`. This allows inactive pages to be evicted to

disk and later retrieved on demand. The design involves modifying the disk layout, identifying victim processes and pages, managing page eviction and restoration, and controlling swapping adaptively.

Relevant Macros and Structures

In `param.h`, the swap block size and total file system size are defined as:

```
#define NSWAPBLOCKS 6400  // Total number of 512-byte disk
    blocks reserved for swapping
#define FSSIZE      7400  // Total number of disk blocks in
    the file system
```

Each memory page is 4096 bytes, and each disk block is 512 bytes, so storing one page requires 8 blocks. Therefore, the total swap space of 6400 blocks can store $6400 \div 8 = 800$ pages.

The number of usable swap slots is defined in `pageswap.h`:

```
#define NSWAPSLOTS 800  // Number of 4KB swap slots
```

The metadata for each swap slot is stored in the `swaptable[]` array:

```
struct swapslot {
    int is_free;       // 1 if the slot is available, 0 if
        used
    int page_perm;     // Lower 12 bits of the original page
        's PTE (permission flags)
};
```

The field `page_perm` is used to save the original access flags of a page when it is evicted (e.g., read/write, user/kernel). When the page is loaded back, this field is restored into the PTE to recover the exact same permission bits.

The number of disk blocks used per swap slot is computed as:

$$\texttt{nblocks\_in\_swap\_slot} = \frac{\texttt{NSWAPBLOCKS}}{\texttt{NSWAPSLOTS}} = 8$$

Each slot is mapped to disk block offsets starting at block 2 (i.e., after the boot and superblock), with slot s using blocks $2 + s \times 8$ through $2 + s \times 8 + 7$.

Modified Disk Layout

A new swap block region is inserted between the superblock and the log. Each 4KB page is stored in 8 contiguous 512-byte blocks. A total of 6400 blocks are reserved, forming 800 swap slots. These blocks are not tracked by the file system and are managed entirely by the swap subsystem.

The updated layout is as follows:

```
[ boot | superblock | swap blocks | log | inode blocks | bitmap |
                                data blocks ]
```

The disk layout logic is modified to reduce usable data blocks and offset log, inode, and bitmap regions to make space for swap:

```
int nswapblocks = NSWAPBLOCKS;
nmeta = 2 + nlog + ninodeblocks + nbitmap;
nblocks = FSSIZE - nmeta - nswapblocks;

sb.size = xint(FSSIZE);
sb.nblocks = xint(nblocks);
sb.ninodes = xint(NINODES);
sb.nlog = xint(nlog);
sb.logstart = xint(2 + nswapblocks);
sb.inodestart = xint(2 + nswapblocks + nlog);
sb.bmapstart = xint(2 + nswapblocks + nlog + ninodeblocks);

freeblock = nswapblocks + nmeta;
nfsblocks = nlog + ninodeblocks + nbitmap;
balloc(nfsblocks);
```

Swap Slot Management

The swap system manages a fixed array of 800 swap slots. Each slot can store one memory page (4KB) using 8 consecutive disk blocks. The structure ‘swaptable[]‘ holds the state of each slot, including whether it is currently used and the saved page permissions.

To initialize this table, the following function is called at boot time. It marks all swap slots as free using a spinlock for synchronization.

```
void swapinit(void) {
  initlock(&swaplock, "swaplock");
  acquire(&swaplock);
  for (int i = 0; i < NSWAPSLOTS; i++)
    swaptable[i].is_free = 1;
  release(&swaplock);
}
```

When a page is being swapped out, a free slot must be allocated. The function ‘allocswap()‘ searches the swap table linearly for a free slot and marks it as used. If no slot is available, it returns -1.

```
int allocswap(void) {
  acquire(&swaplock);
  for (int i = 0; i < NSWAPSLOTS; i++) {
    if (swaptable[i].is_free) {
      swaptable[i].is_free = 0;
      release(&swaplock);
      return i;
    }
  }
  release(&swaplock);
  return -1;
}
```

After a swapped-out page is loaded back into memory, its slot can be reused. The ‘freeswap()‘ function marks a given slot as free again.

```
void freeswap(int slot) {
  acquire(&swaplock);
  if (slot >= 0 && slot < NSWAPSLOTS)
    swaptable[slot].is_free = 1;
  release(&swaplock);
}
```

The actual data transfer to disk is done using ‘writeswap()‘. It divides a 4096-byte page into 8 blocks of 512 bytes, then writes each block to disk at a calculated offset based on the slot number. The starting block is at index ‘2 + slot * 8‘.

```
void writeswap(char *page, int slot) {
  for (int i = 0; i < nblocks_in_swap_slot; i++) {
    struct buf *b = bread(ROOTDEV, 2 + slot *
        nblocks_in_swap_slot + i);
    memmove(b->data, page + i * 512, 512);
    bwrite(b);
    brelse(b);
  }
}
```

To load a page back from disk, the kernel calls ‘readswap()‘. It reads 8 consecutive blocks and reconstructs the original page in memory.

```
void readswap(char *page, int slot) {
  for (int i = 0; i < nblocks_in_swap_slot; i++) {
    struct buf *b = bread(ROOTDEV, 2 + slot *
        nblocks_in_swap_slot + i);
    memmove(page + i * 512, b->data, 512);
    brelse(b);
  }
}
```

These five functions form the core of the swap infrastructure. They manage slot allocation, free tracking, and raw I/O of pages between memory and disk independently of the file system. All functions are protected using a spinlock to avoid concurrent access issues during swapping.

Victim Process and Page Selection

The victim selection policy first identifies a process with the highest number of resident user pages (rss). Ties are broken by choosing the process with the lower PID. Only RUNNABLE, RUNNING, and SLEEPING processes are considered.

Once a victim process is selected, its virtual address space is scanned for a page that: - Is present in memory (`PTE_P` set) - Is user-accessible (`PTE_U` set) - Has not been recently accessed (`PTE_A` clear)

```
struct proc* getvictimproc(void) {
  struct proc *victim = 0;
  acquire(&ptable.lock);
  for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC
      ]; p++) {
    if (p->state == RUNNABLE || p->state == SLEEPING || p->
        state == RUNNING) {
      if (victim == 0 || p->rss > victim->rss ||
          (p->rss == victim->rss && p->pid < victim->pid))
        victim = p;
    }
  }
  release(&ptable.lock);
  return victim;
}
```

Swap-out Logic

The kernel initiates page eviction through the function `swapoutpage()`, which first tries to swap out a page using `tryswapoutpage()`.

`tryswapoutpage()` performs the actual swap:

- It scans the virtual address space of the process.
- It selects a page with `PTE_P`, `PTE_U`, and no `PTE_A`.
- It allocates a swap slot using `allocswap()`.
- It copies the physical page to disk using `writeswap()`.
- It stores the original page permissions in `swaptable[slot].page_perm`.
- It updates the page table entry to encode the swap slot index and set `PTE_S`.
- It frees the page and decrements `rss`.

```
int tryswapoutpage(struct proc *p) {
  for (uint va = 0; va < p->sz; va += PGSIZE) {
    pte_t *pte = walkpgdir(p->pgdir, (char *)va, 0);
    if (pte && (*pte & PTE_P) && (*pte & PTE_U) && !(*pte &
        PTE_A)) {
      char *pa = P2V(PTE_ADDR(*pte));
      int slot = allocswap();
      if (slot < 0)
        return -1;

      writeswap(pa, slot);
      swaptable[slot].page_perm = *pte & 0xFFF;
      *pte = (slot << 12) | PTE_S | PTE_U;

      kfree(pa);
      p->rss--;
      lcr3(V2P(p->pgdir));
      return 0;
    }
  }
  return -1;
}
```

If no eligible page is found, it clears the accessed bits (`PTE_A`) of up to 10% of user pages and retries.

```
int swapoutpage(struct proc *p) {
  int r = tryswapoutpage(p);
  if (r == 0)
    return 0;

  int maxcleared = p->sz / (PGSIZE * 10) + 1;
  int cleared = 0;
  for (uint va = 0; va < p->sz && cleared < maxcleared; va
      += PGSIZE) {
    pte_t *pte = walkpgdir(p->pgdir, (char*)va, 0);
    if (pte && (*pte & PTE_P) && (*pte & PTE_U)) {
      *pte &= ~PTE_A;
      cleared++;
    }
  }
  lcr3(V2P(p->pgdir));
  return tryswapoutpage(p);
}
```

Page Fault Handling

When a swapped-out page is accessed, a page fault occurs. The handler checks if PTE_S is set. If so, it retrieves the page from the swap slot, restores the permission bits, updates the page table, and reloads the CR3 register:

```
int slot = PTE_ADDR(*pte) >> 12;
char *mem = kalloc();
readswap(mem, slot);
*pte = V2P(mem) | swaptable[slot].page_perm | PTE_P | PTE_A;
freeswap(slot);
p->rss++;
lcr3(V2P(p->pgdir));
```

Custom PTE Flags

Defined in mmu.h:

```
#define PTE_A 0x020  // Accessed
#define PTE_S 0x100  // Swapped
```

PTE_A tracks if a page has been accessed recently. PTE_S marks the page as swapped and invalid in memory.

Trap Handling

Page faults are captured and handled in trap.c, which processes all exceptions and interrupts. When a page fault occurs (interrupt number T_PGFLT), the kernel checks if it originated in user mode and whether it can be resolved through swapping.

The relevant trap handling logic is as follows:

```
case T_PGFLT:
  if(myproc() != 0 && (tf->cs & 3) == DPL_USER)
    if(handlepagefault(myproc(), rcr2()) == 0)
      break;
```

The flow is:

- T_PGFLT corresponds to a page fault interrupt.
- tf->cs & 3 == DPL_USER checks if the fault occurred in user space.
- rcr2() retrieves the faulting virtual address.
- handlepagefault() attempts to load the missing page from swap if it is marked with PTE_S.
- If handlepagefault() succeeds (returns 0), the fault is considered handled, and normal execution resumes.

- If it fails, standard xv6 trap code continues and will likely terminate the process.

This ensures that swapped-out pages are restored transparently when accessed, and page faults caused by missing or invalid pages are handled safely.

Adaptive Replacement Strategy

The system swaps pages only when the number of free pages drops below a threshold `Th`. Initially, `Th = 100` and `Npg = 4`. After each swap trigger: - `Th` is reduced by $\lfloor Th \times \frac{\beta}{100} \rfloor$ - `Npg` is increased by $\lfloor Npg \times \frac{\alpha}{100} \rfloor$, capped at `LIMIT`

```
if(countfreepages() < Th){
  cprintf("Current Threshold = %d, Swapping %d pages\n", Th,
       Npg);
  for(int i = 0; i < Npg; i++){
    if (swapoutpage(getvictimproc()) < 0)
      break;
  }

  Th -= (Th * BETA) / 100;
  Npg += (Npg * ALPHA) / 100;
  if(Npg > LIMIT)
    Npg = LIMIT;

  return kalloc();
}
```

`ALPHA` and `BETA` are defined in the Makefile:

```
CFLAGS += -DALPHA=25 -DBETA=10
```

Effect of α and β on System Efficiency

β controls how aggressively the threshold `Th` decreases. A higher β causes the system to initiate swapping more often after the first trigger.

α controls how many pages are swapped per event. A larger α increases swap batch size, which helps when memory is very tight but can evict too many pages unnecessarily if too high.

Together, α and β balance swap frequency and volume. Choosing small values leads to slower, gradual response; higher values yield more aggressive recovery but may cause higher page fault rates. The system adapts based on current conditions using this feedback mechanism.