In God we trust

Os lab EX5

Mani Hosseini 810102552
Shayan Maleki 810102515
Hamid Mahmoudi 810102549

Question 1 :

Virtual memory area:

The buffer cache in a real-world operating system such as linux is significantly more complex than xv6's, but it serves the same two purposes: caching and synchronizing access to the disk. Xv6's buffer cache, like V6's, uses a simple least recently used (LRU) eviction policy; there are many more complex policies that can be implemented, each good for some workloads and not as good for others. A more efficient LRU cache would elimi nate the linked list, instead using a hash table for lookups and a heap for LRU evic tions. Modern buffer caches are typically integrated with the virtual memory system to support memory-mapped files.

Guard page:

The stack is a single page, and is shown with the initial contents as created by ex ec. Strings containing the command-line arguments, as well as an array of pointers to them, are at the very top of the stack. Just under that are values that allow a program to start at main as if the function call main(argc, argv) had just started. To guard a stack growing off the stack page, xv6 places a guard page right below the stack. The guard page is not mapped and so if the stack runs off the stack page, the hardware will generate an exception because it cannot translate the faulting address. A real world operating system might allocate more space for the stack so that it can grow be yond one page.

Question 2:

A hierarchical (two-level) paging structure reduces memory consumption by efficiently handling sparse address spaces.

Single-level problem: In a single-level page table for a 32-bit architecture with 4KB pages, the table would require 2^20 entries. Since the table must be contiguous, this would consume 4 MB of RAM for every single process, regardless of how small the process is.

Hierarchical solution: With a two-level structure (Page Directory and Page Table), the system only needs to allocate the top-level Page Directory (4KB) initially. Secondary

Page Tables are allocated only for the regions of virtual memory that the process actually uses. Since most processes use only a fraction of the 4GB address space (leaving large gaps between the stack and code/heap), the operating system avoids allocating memory for the page tables corresponding to these unused "holes"

Question 3 :

Each 32-bit entry in the Page Directory and Page Table controls address translation and access permissions. The bits are defined as follows:

- Bit 0 (P - Present): Indicates if the page is currently in physical memory.
- Bit 1 (W - Writable): Determines if the page can be written to.
- Bit 2 (U - User): Controls user-mode access.
- Bit 3 (WT - Write-Through): Controls the caching policy.
- Bit 4 (CD - Cache Disabled): Disables caching for this page.
- Bit 6 (D - Dirty): Set by hardware when the page is written to.
- Bits 7-11 (AVL): Available for system software use.
- Bits 31-12 (PPN): The Physical Page Number. This points to the base address of the next structure (the Page Table or the physical Page Frame).

Difference between Page Directory and Page Table entries: The document states that the entries are identical in structure, except for the Dirty (D) bit.

- In a Page Table Entry (PTE), the D bit indicates that the page has been modified (written to).
- In a Page Directory Entry (PDE), the D bit is typically 0 (unused for 4KB pages).

Question 4 :

kalloc allocates Physical Memory.

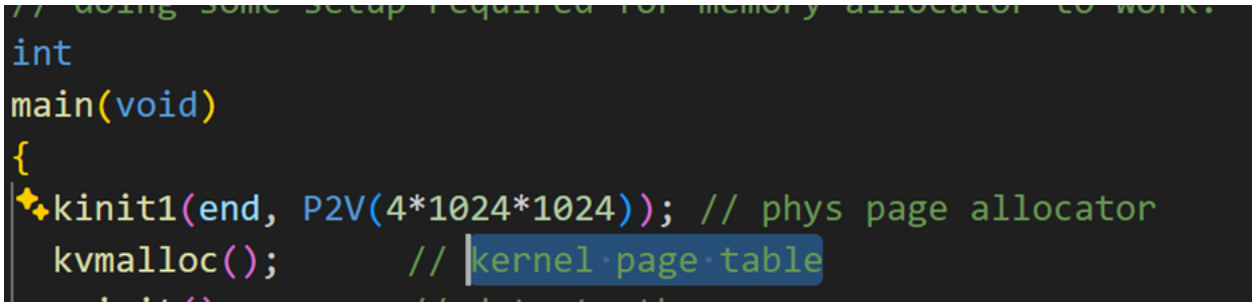
kalloc is responsible for managing the system's RAM. It retrieves a free 4KB physical page (frame) from the kernel's free list.

While the address returned by kalloc is a kernel virtual address (so the kernel can access it), the actual resource being reserved is a page of physical memory. This physical page is then typically mapped to a specific virtual address using functions like

mappages. This is distinct from functions like allocuvm, which handle the setup of virtual address spaces.

Question 5: What is the purpose of the mappages function?

When we look at the main.c we can see it calls the kvmalloc to setup the kernel page table



Then inside it , it calls the setup-kvm() which is the part where its using the mappages function.

```
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

It first allocates a page of memory to hold the page directory. Then it calls mappages to install the translations that the kernel needs, which are described in the kmap array.

mappages maps a range of virtual addresses to a range of physical addresses, with specific permissions (e.g., read/write, user/kernel access). It does this by updating Page Table Entries (PTEs) in the page directory.

Question 6: Are all parts of virtual memory from address zero to proc->sz usable by the user?

No, not all parts are usable.

While the user process has virtual memory from address 0 to proc->sz, not all of it is accessible. The key reason is the presence of a guard page.

proc->sz represents the current size of the process's virtual address space.

The stack is a single page at the top of the user space.

Right below the stack is a guard page (an unmapped page).

This guard page is not mapped in the page table (i.e., no PTE is valid).

If the stack grows beyond its page (e.g., due to a stack overflow), it will try to access the guard page.

Since the guard page is not mapped, the hardware generates a page fault, which the OS handles as a stack overflow.

The guard page acts as a safety mechanism to detect stack overflows.

Even though the guard page is within the 0 to proc->sz range, it is not usable by the user program.

This prevents accidental or malicious memory corruption.

Question 7: Explain the walkpgdir function. What hardware operation does this function simulate?

It's responsible for navigating the two-level page table hierarchy in software. Given a page directory and a virtual address,

For each virtual address to be mapped, mappages calls walkpgdir to find the address of the PTE for that address. It then initializes the PTE to hold the relevant physical page number, the desired permissions (PTE_W and/or PTE_U), and PTE_P to mark the PTE as valid

walkpgdir performs the following steps:

1. **Page Directory Traversal:** It examines the page directory to find the corresponding page table.

2. **Page Table Entry (PTE) Lookup:** Within the found page table, it searches for the PTE associated with the given virtual address.

3. **Page Table Allocation (if needed):** If the required page table doesn't exist, walkpgdir can allocate a new one.

walkpgdir simulates the hardware page translation process, also known as the Page Table Walk, which is handled by the CPU's Memory Management Unit (MMU).

When a program accesses a virtual address, the hardware automatically performs a traversal of the page tables (Page Directory -> Page Table -> Physical Address).

walkpgdir is the software equivalent of this hardware operation, enabling the kernel to manually inspect, create, or modify memory mappings. It allows the kernel to perform the same table traversal as the MMU would.

Question 8 : exaplain the mappages and allocuvm functions:

We explained the mappages function a little bit earlier its task was to map a range of virtual addresses to a range of physical addresses.
mappages installs mappings into a page table for a range of virtual addresses to a corresponding range of physical addresses. It does this separately for each virtual address in the range, at page intervals. For each virtual address to be mapped, mappages calls walkpgdir to find the address of the PTE for that address. It then initializes the PTE to hold the relevant physical page number, the desired permissions (PTE_W and/or PTE_U), and PTE_P to mark the PTE as valid .

As shown in the setupkvm function the mappages has these following args:

pgdir: The page directory (top-level page table).

virt: Starting virtual address.

size: Number of bytes to map.

phys_start: Starting physical address.

perm: Permissions (e.g., PTE_W for write, PTE_U for user access).

The exact procedure it performs is explained below:

It Iterates over each page in the given virtual address range.

For each page:

Calls walkpgdir to find or create the **PTE** (Page Table Entry) for that virtual address.

If the page table is not allocated, walkpgdir allocates it.

Writes the **physical address** and **permissions** into the PTE.

Marks the PTE as valid (PTE_P).

Allocuvm:

The allocuvm function is responsible for **allocating physical memory** and **mapping it into the user's virtual address space**. Essentially, it's how the kernel provides a process with more memory.

It allocates new **physical pages** (usually using kalloc).

It then **updates the process's page table** to map these newly allocated pages to the process's virtual address space. This increases the process's "valid" memory region.

Usage:

allocuvm is heavily used during the exec system call. When a new program is started, allocuvm is called to allocate the necessary physical memory for the new program's code, data, and stack.

It ensures that the virtual addresses used by the program are backed by actual physical memory, enabling the program to run.

In short, allocuvm bridges the gap between physical memory and the process's view of memory, allowing processes to have their own private address spaces backed by real hardware.

Question 9: Describe the process of loading a program into memory by the exec system call.

As we know The exec system call replaces the calling process's memory with a new memory image loaded from a file stored in the file system. The file must have a particular format, which specifies which part of the file holds instructions, which part is data, at which instruction to start, etc. xv6 uses the ELF format, which Chapter 2 discusses in more detail. When exec succeeds, it does not return to the calling program; instead, the instructions loaded from the file start executing at the entry point declared in the ELF header. Exec takes two arguments: the name of the file containing the executable and an array of string arguments.

The journey of program execution begins with a system transition, a process known as exec. This involves loading the new program's code and data into the system's memory. A key step in this process is the establishment of a virtual address space for the new program, which is managed through page tables.

```
if((pgdir = setupkvm()) == 0) goto bad;
```

As we can see above This code initializes the page directory (pgdir), a component for mapping virtual addresses to physical memory. The setupkvm() function is responsible for creating this page directory.

Next, the program's code and data are loaded from the file into the allocated memory regions, which are then mapped into the new process's virtual address space.

```
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)) {
    readi(ip, (char*)&ph, off, sizeof(ph));
    if(ph.type != ELF_PROG_LOAD) continue;
    if(ph.memsz < ph.filesz) goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr) goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0) goto bad;
    if(ph.vaddr % PGSIZE != 0) goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0) goto bad;
}
```

This loop iterates through the program headers (ph) to identify loadable segments, allocating memory for each segment using allocuvm() and loading the program's code and data into it using loaduvm().

Following memory allocation, the system prepares the stack, a region of memory used for function calls and local variables. A guard page is allocated immediately before the stack to help prevent stack overflows.

```
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0) goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

This code allocates memory for the stack and the guard page, clears the page table entry for the guard page, and sets the stack pointer (sp) to the top of the stack.

The command-line arguments are then placed on the stack in a specific order, preparing the environment for the new program.

```
for(argc = 0; argv[argc]; argc++) {
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1);
    ustack[3+argc] = sp;
}
ustack[3+argc] = 0;
ustack[0] = 0xffffffff; // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4; // argv pointer
sp -= (3+argc+1)*4;
copyout(pgdir, sp, ustack, (3+argc+1)*4);
```

This loop iterates through the command-line arguments, calculates the appropriate offset on the stack, copies the argument string to the stack using copyout(), and updates the stack pointer. The arguments are stored in a specific order, and the return address is also set on the stack.

Finally, the system switches to the new process's address space, effectively starting the new program's execution.

```
oldpgdir = curproc->pgdir;
curproc->pgdir = pgdir;
curproc->sz = sz;
curproc->tf->eip = elf.entry;  // entry point
curproc->tf->esp = sp;
switchuvm(curproc);
freevm(oldpgdir);
return 0;
```

Here, the page directory of the current process is replaced with the new process's page directory. The instruction pointer (eip) of the new process's thread control block (tf) is set to the program's entry point (elf.entry), and the stack pointer (esp) is set to the top of the stack. The switchuvm() function performs the actual switch to the new address space. The old page directory is then freed.

Question 10 :

1. Linux (Two-List LRU):

Linux uses a Two-List LRU (Least Recently Used) algorithm to approximate standard LRU efficiently.

It maintains two lists:

Active List: Pages frequently accessed.

Inactive List: Pages rarely accessed, candidates for eviction.

When a page is accessed, it moves to the Active List.

When memory is low, pages are evicted from the tail of the Inactive List.

Advantage: Prevents thrashing from one-time large file reads, as such pages are quickly moved to Inactive and evicted without affecting active pages.

2. Windows (Working Set + Clock Algorithm):

Windows uses a Working Set model combined with a Clock Algorithm (a variation of Second-Chance).

Each process has a Working Set (minimum and maximum physical memory).

The Balance Set Manager trims working sets when memory is low.

The Clock Algorithm scans pages in the working set:

If the Accessed bit is set → clear it and give a second chance.

If the bit is clear → page is eligible for eviction.

Advantage: Idle processes lose memory to active ones, improving overall system responsiveness.

Question 11 :

Yes, the algorithms work as expected for example in the first program all algorithms other than least frequently used get around 50% because of reads getting hits and writes getting missed, or similarly second program is getting many missed reads because the access is circular and five elements causing lfu to be the only good algorithm.(The original more detailed report for this question is in Google docs but because of the Internet I couldn't get it. If necessary I can send that to you (with a modified date before 8 Jan to prove legitimacy))

Implementation

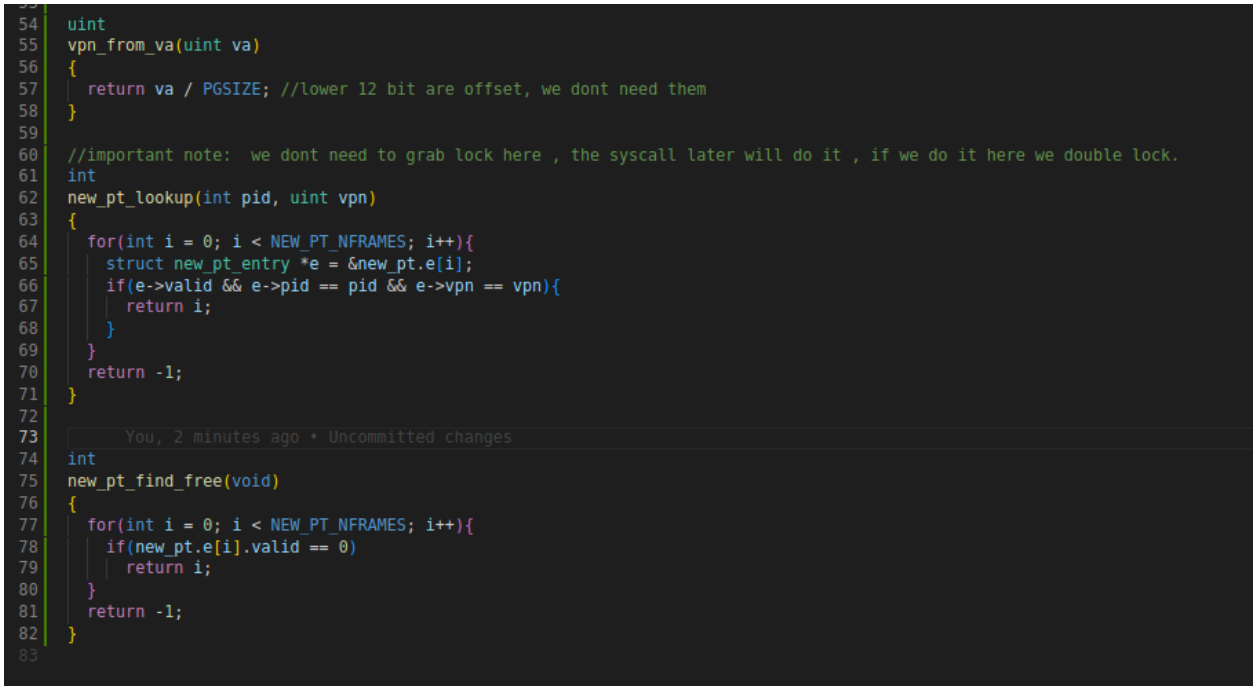
Step1:

- Created a global page-table cache with 4 entries
- Added a spinlock to protect it (shared kernel structure)
- Allocated 4 physical pages at boot using kalloc
- Each entry starts invalid (valid = 0) even though memory is allocated
- Initialized basic metadata (pid, vpn, timestamps) for later steps

```c
#ifndef NEW_PTABLE_H
#define NEW_PTABLE_H

#include "types.h"
#include "spinlock.h"

#define NEW_PT_NFRAMES 4

// One cache slot (one "frame" in our new_ptable)
struct new_pt_entry {
  char *frame_kva;   // kernel virtual address of the allocated  frame
  uint  vpn;         // virtual page number (va / PGSIZE)
  int   pid;
  int   valid;       // valid bit

  // Replacement related stuff
  uint  load_time;      // FIFO
  uint  last_used_time; // LRU
  uint  use_count;      // LFU
  uint  refbit;         // Clock
};

struct new_ptable {
  struct spinlock lock;
  struct new_pt_entry e[NEW_PT_NFRAMES];


  uint time;     // keep track of time for replacement
  int  hand;     // clock hand index [0..3]
};

extern struct new_ptable new_pt;

void new_ptable_init(void);
void new_ptable_dump(void);

#endif
```

Step2:

- Added lookup helpers for new_ptable (no real page tables yet)
- Convert virtual address → VPN
- Implemented HIT check using
- Implemented free-slot search (valid == 0)
- No eviction yet, just detect hit, miss, or free
- Locking rule: caller holds the spinlock, helpers don’t lock



```
54  uint
55  vpn_from_va(uint va)
56  {
57    return va / PGSIZE; //lower 12 bit are offset, we dont need them
58  }
59
60  //important note:  we dont need to grab lock here , the syscall later will do it , if we do it here we double lock.
61  int
62  new_pt_lookup(int pid, uint vpn)
63  {
64    for(int i = 0; i < NEW_PT_NFRAMES; i++){
65      struct new_pt_entry *e = &new_pt.e[i];
66      if(e->valid && e->pid == pid && e->vpn == vpn){
67        return i;
68      }
69    }
70    return -1;
71  }
72
73        You, 2 minutes ago • Uncommitted changes
74  int
75  new_pt_find_free(void)
76  {
77    for(int i = 0; i < NEW_PT_NFRAMES; i++){
78      if(new_pt.e[i].valid == 0)
79        return i;
80    }
81    return -1;
82  }
83
```

Step3:

- Implemented “make sure page is in cache” function
- Validates user VA, computes VPN + page-aligned va_page
- HIT: return slot index, update metadata (time / last_used / use_count / refbit)
- MISS + free slot: find real user page, then memmove() 4KB into cached frame,upate metadata, return slot.
- MISS + full table: return -2 (signals “eviction needed later in Step 4”)

```c
int
new_pt_check_page(struct proc *p, uint va)
{
  // must be user address
  if(va >= KERNBASE)
    return -1;

  // this line like rips off the offset leaving the base of page so we dont start copy from middle of the page.
  uint va_page = PGROUNDDOWN(va);

  uint vpn = vpn_from_va(va);


  // check1:  do we have the page already?
  int idx = new_pt_lookup(p->pid, vpn);
  if(idx >= 0){
    new_pt.time++;
    new_pt.e[idx].last_used_time = new_pt.time;
    new_pt.e[idx].use_count++;
    new_pt.e[idx].refbit = 1;
    return idx;
  }


  // check2: if we dont have the page , are there free space or we need eviction
  int freei = new_pt_find_free();
  if(freei < 0){
    // Table full(need eviction)
    return -2;
  }

  // Copy the real user page into our cached frame
  char *src = userva_page_to_kva(p, va_page);
  if(src == 0){
    // user VA not mapped/present
    return -1;
  }

  char *dst = new_pt.e[freei].frame_kva;
  memmove(dst, src, PGSIZE);

  new_pt.time++;
  new_pt.e[freei].pid = p->pid;
  new_pt.e[freei].vpn = vpn;
  new_pt.e[freei].valid = 1;
  new_pt.e[freei].load_time = new_pt.time;
  new_pt.e[freei].last_used_time = new_pt.time;
  new_pt.e[freei].use_count = 1;
  new_pt.e[freei].refbit = 1;

  return freei;
}
```

Step4:

- Added replacement policy selection (FIFO / LRU / LFU / CLOCK) via enum + global new_pt_policy
- Implemented new_pt_pick_victim()
- On MISS + table full:
  - pick a victim slot
  - write back victim frame to the victim process’s real user page
  - reuse that slot as the “free” slot
- Then load requested page into that slot, and overwrite all metadata

```
int new_pt_pick_victim(void)
{
  int victim = -1;

  if (new_pt_policy == NEWPT_FIFO)
  {
    uint min = 0xFFFFFFFF;
    for (int i = 0; i < NEW_PT_NFRAMES; i++)
    {
      if (new_pt.e[i].load_time < min)
      {       You, 3 minutes ago • Uncommitted changes
        min = new_pt.e[i].load_time;
        victim = i;
      }
    }
  }

  else if (new_pt_policy == NEWPT_LRU)
  {
    uint min = 0xFFFFFFFF;
    for (int i = 0; i < NEW_PT_NFRAMES; i++)
    {
      if (new_pt.e[i].last_used_time < min)
      {
        min = new_pt.e[i].last_used_time;
        victim = i;
      }
    }
  }

  else if (new_pt_policy == NEWPT_LFU)
  {
    uint min = 0xFFFFFFFF;
    for (int i = 0; i < NEW_PT_NFRAMES; i++)
    {
      if (new_pt.e[i].use_count < min)
      {
        min = new_pt.e[i].use_count;
        victim = i;
      }
    }
  }

  else if (new_pt_policy == NEWPT_CLOCK)
  {
    while (1)
    {
      int i = new_pt.hand;
      if (new_pt.e[i].refbit == 0)
      {
        victim = i;
        new_pt.hand = (i + 1) % NEW_PT_NFRAMES;
        break;
      }
      new_pt.e[i].refbit = 0;
      new_pt.hand = (i + 1) % NEW_PT_NFRAMES;
    }
```

Step5:

- Added 2 syscalls: newpt_write(va, value) and newpt_read(va)

```
int
sys_newpt_write(void)
{
  int va;
  int value;
  struct proc *p = myproc();

  if(argint(0, &va) < 0) return -1;
  if(argint(1, &value) < 0) return -1;

  acquire(&new_pt.lock);
  int slot = new_pt_check_page(p, (uint)va);
  if(slot < 0){
    release(&new_pt.lock);
    return -1;
  }

  uint offset = ((uint)va) % PGSIZE;
  if(offset + sizeof(int) > PGSIZE){
    release(&new_pt.lock);
    return -1;
  }

  *(int*)(new_pt.e[slot].frame_kva + offset) = value;

  release(&new_pt.lock);
  return 0;
}

int
sys_newpt_read(void)
{
  int va;
  struct proc *p = myproc();

  if(argint(0, &va) < 0) return -1;

  acquire(&new_pt.lock);
  int slot = new_pt_check_page(p, (uint)va);
  if(slot < 0){
    release(&new_pt.lock);      You, 21 minutes ago • Uncommitted changes
    return -1;
  }

  uint offset = ((uint)va) % PGSIZE;
  if(offset + sizeof(int) > PGSIZE){
    release(&new_pt.lock);
    return -1;
  }

  int value = *(int*)(new_pt.e[slot].frame_kva + offset);

  release(&new_pt.lock);
  return value;
}
```

Step6:

- Added metrics syscall for hit count, hit ratio and runtime (ticks)
- added newpt_setpolicy(policy) syscall to:
    - switch FIFO / LRU / LFU / CLOCK
    - reset stats + metadata between runs

```
38  int
39  sys_newpt_report(void)
40  {
41    int start;
42    if(argint(0, &start) < 0)
43      return -1;
44
45    acquire(&new_pt.lock);
46    uint hits = new_pt_hits;
47    uint misses = new_pt_misses;
48    release(&new_pt.lock);
49
50    uint accesses = hits + misses;
51    uint ratio = 0;
52    if(accesses > 0)
53      ratio = (hits * 100) / accesses;
54
55    int end = uptime();
56    int runtime = end - start;
57    if(runtime < 0) runtime = 0;
58
59    cprintf("---new_pt metrics---\n");
60    cprintf("count hit: %d\n", hits);
61    cprintf("ratio hit: %d%%\n", ratio);
62    cprintf("runtime: %d ticks\n", runtime);
63
64    return 0;
65  }
66
67
68  int
69  sys_newpt_setpolicy(void)
70  {
71    // this function is made for testing later
72    int pol;
73    if(argint(0, &pol) < 0)      You, 8 minutes ago • Uncommitted changes
74      return -1;
75
76    if(pol < NEWPT_FIFO || pol > NEWPT_CLOCK)
77      return -1;
78
79    acquire(&new_pt.lock);
80    new_pt_policy = pol;
81
82    // reset stats per run
83    new_pt_hits = 0;
84    new_pt_misses = 0;
85    new_pt.time = 0;
86    new_pt.hand = 0;
87
88    // clear refbits/counters so each run is fair
89    for(int i = 0; i < NEW_PT_NFRAMES; i++){
90      new_pt.e[i].load_time = 0;
91      new_pt.e[i].last_used_time = 0;
92      new_pt.e[i].use_count = 0;
93      new_pt.e[i].refbit = 0;
94    }
```

Step7:

- On process exit, scan new_ptable and:
    - invalidate every entry with matching pid
    - clear metadata (pid, vpn, counters, refbit)
- Do NOT free frames because frames belong to new_ptable
- Hooked invalidate call into exit() before process becomes ZOMBIE
- Prevents stale cached pages, PID reuse bugs, and incorrect HITs

```
void
new_pt_invalidate_pid(int pid)
{
  acquire(&new_pt.lock);

  for(int i = 0; i < NEW_PT_NFRAMES; i++){
    if(new_pt.e[i].valid && new_pt.e[i].pid == pid){
      new_pt.e[i].valid = 0;

      // just in case we clear other fields
      new_pt.e[i].pid = -1;      You, 11 minutes ago • Uncom
      new_pt.e[i].vpn = 0;
      new_pt.e[i].load_time = 0;
      new_pt.e[i].last_used_time = 0;
      new_pt.e[i].use_count = 0;
      new_pt.e[i].refbit = 0;
    }
  }

  release(&new_pt.lock);
}
```

Step8:

In Program 1 (sequential scan of ≥10 pages), FIFO, LRU, and CLOCK show similar hit ratios (~50%) because the only locality comes from reading immediately after writing the same page.
LFU behaves differently: in a single-CPU, single-process setting it achieves a higher hit ratio (~62%) because pages accessed more frequently early in the scan accumulate higher frequency and remain resident.
However, under multiprocess execution, LFU performance degrades due to frequency tie-breaking and interleaving, illustrating LFU's sensitivity to concurrency.

Program 2 demonstrates that when the access pattern cycles over one more page than the cache can hold, FIFO/LRU/CLOCK can thrash severely, while LFU may sometimes

retain higher-frequency pages and achieve a better hit ratio. This behavior matches expected theoretical properties of the algorithms.
Program 3 when multiprocessing: generates strong frequency locality by accessing three “hot” pages 15,000 times per process and five “cold” pages only 250 times per process. With a single global cache of four frames shared across multiple processes, six hot pages (3 per process) compete for four frames, causing unavoidable contention and evictions. In our results, LFU achieves the highest hit ratio (~49%) because it preferentially retains frequently accessed pages, while FIFO/LRU/CLOCK show significantly lower hit ratios under multiprocess interleaving.
When not multiprocessing: Program 3 creates strong frequency locality by accessing 3 hot pages (15,000 accesses) and 5 cold pages (250 accesses). With a 4-frame cache, the three hot pages remain resident most of the time, while the remaining frame is used for cold pages. As a result, all algorithms achieve a high hit ratio (~97–98%), with LFU slightly higher because it prioritizes frequently accessed pages.
Program 4 repeatedly accesses 5 pages in the pattern (0,1,2,3,0,1,4). Pages 0 and 1 are accessed twice per iteration, making them “hot”, while page 4 is accessed once and acts as a disruptor in a 4-frame cache. In the single-process run, LFU achieves the best hit ratio (~71%) by retaining the most frequently accessed pages (0 and 1). LRU performs next (~57%) by keeping recently accessed hot pages and evicting less recent pages (2/3). FIFO and CLOCK perform worse (~28%) because they do not adapt well to the repeated disturbance caused by page 4.In the multi-process run, cache contention becomes dominant because the global 4-frame table must serve 10 pages (5 per process). FIFO/LRU/CLOCK experience severe thrashing and their hit ratios drop to ~2–3%. LFU remains significantly better (~36%) because it preferentially retains the globally most frequent pages and is more resistant to interleaving effects.

Q11:

Yes. I compared the programs using the required metrics (hit count, hit ratio, and runtime) under all four policies, and I also tested multi-process execution as requested.

The behaviors were mostly predictable from the access patterns:

- Program 1 (sequential scan of many pages): hit ratio stayed around ~50% for FIFO/LRU/CLOCK because each page is typically a miss on first touch and an immediate hit on the next access; replacement policy cannot help much on a scan.
- Program 2 (cyclic reads over 5 pages with only 4 frames): FIFO/LRU/CLOCK showed near 0% hit due to classic thrashing when working set = cache size + 1;

this is predictable. LFU sometimes did better because it can keep higher-frequency pages.

- Program 3 (3 very hot pages + 5 cold pages): with one process, hit ratio became very high (~97–98%) because the 3 hot pages fit in 4 frames; this is predictable. With multiple processes, hit ratios dropped because hot pages from different processes compete for the same 4 frames (global cache contention), which is also predictable.
- Program 4 (pattern 0,1,2,3,0,1,4): with one process, LFU > LRU > FIFO/CLOCK as expected because 0 and 1 are most frequent. With multiple processes, FIFO/LRU/CLOCK collapsed due to contention, while LFU remained best; this is predictable.

So overall, the comparisons were done using the requested criteria, and the observed behaviors are explainable and mostly predictable based on locality (recency/frequency) and on contention in the multi-process case.