ASSIGNMENT 3

Operating System Lab

Sujit Haloi 220123065

PART A: Lazy Memory Allocation

In xv6, whenever a process requires more memory than what has already been allocated, it uses the `sbrk()` system call to request additional memory from the operating system. This function plays a key role in dynamic memory management, allowing a process to grow its heap space as needed during runtime.

The `sbrk()` system call internally utilizes the `growproc()` function, which is defined in `proc.c`. The primary role of `growproc()` is to adjust the process's size by increasing or decreasing the amount of memory assigned to it. This is done by either adding or freeing memory pages, depending on the request.

A deeper inspection of `growproc()` reveals that it, in turn, calls the `allocuvm()` function. `allocuvm()` is responsible for the actual allocation of memory pages. It does this by finding free physical memory and assigning it to the process. Additionally, `allocuvm()` handles the crucial task of mapping the new virtual addresses requested by the process to the corresponding physical addresses. This mapping is done within the process's page tables, which are responsible for translating virtual memory addresses used by the process into the actual physical memory locations managed by the OS.

The process of memory allocation involves several key steps:

1. **Page Allocation**: `allocuvm()` allocates one or more pages of physical memory. A page is the basic unit of memory management, and in xv6, each page is 4096 bytes.
2. **Page Table Mapping**: After allocating the pages, `allocuvm()` updates the process's page table to create a mapping between the newly allocated physical memory pages and the corresponding virtual addresses in the process's address space. This mapping ensures that when the process tries to access memory through

a virtual address, the system knows where to find the corresponding data in physical memory.

In this assignment, the key objective is to implement **Lazy Memory Allocation**, a technique where memory is not allocated at the time of the request, but only when it is actually accessed by the process. This method helps in optimizing memory usage by deferring memory allocation until it is absolutely necessary, thereby saving resources when requested memory is not immediately needed.

To implement Lazy Memory Allocation in xv6, we begin by modifying the behavior of the `sbrk()` system call, which traditionally allocates memory as soon as a process requests it. In the original implementation, `sbrk()` would invoke `growproc()`, which in turn calls `allocuvm()` to allocate memory pages and map them to the process's virtual address space. However, in Lazy Memory Allocation, this immediate allocation step is bypassed.

Modifying `sbrk()`

Instead of allocating memory right away, we modify `sbrk()` by commenting out the call to `growproc()`. Instead of triggering memory allocation, we simply update the size variable (`proc->sz`) of the current process to reflect the new size, giving the process the illusion that the requested memory has been allocated. In reality, no physical memory is allocated at this point, and the process believes it has access to the memory based on the updated size variable.

```
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  myproc()->sz += n;

  //  if(growproc(n) < 0)
  //    return -1;
  return addr;
}
```

Triggering a Page Fault

When the process later attempts to access this "allocated" memory, which it believes is available, it encounters a **Page Fault**. Since the memory was not actually allocated, the hardware detects this invalid memory access and raises a page fault, generating a **T_PGFLT (Trap 14)** trap. This trap is handled by the kernel, and the page fault is processed inside the `trap()` function in `trap.c`.

```
switch(tf->trapno){
case T_PGFLT:
  if (lazyAllocate()<0) {
    cprintf("Lazy allocation failed! \n");
    panic("Trap");
  }
  break;
```

Handling the Page Fault

In the `trap()` function, when a **T_PGFLT** trap occurs, we call a custom function, `lazyAllocate()`, to allocate memory lazily. This function is responsible for responding to the page fault by allocating the necessary memory and setting up the correct mappings.

1. **Identifying the Faulting Address**: To determine which memory address caused the page fault, we use the `rcr2()` function. `rcr2()` retrieves the **faulting virtual address**—the address that the process tried to access but was not mapped to any physical memory.
2. **Rounding the Address**: The next step is to align this faulting address to the start of a memory page using the `PGROUNDDOWN()` macro. This gives us the `rounded_addr`, which points to the beginning of the memory page that contains the faulting virtual address.
3. **Allocating a Physical Page**: Once we know the page that needs to be allocated, we call `kalloc()`. The `kalloc()` function is responsible for allocating a free physical page from the system's pool of available memory. This pool is managed using a linked list of free pages, called `freelist`, inside the `kmem` structure. After the call to `kalloc()`, we now have a physical memory page available for use.
4. **Mapping the Physical Page to the Virtual Address**: After obtaining the physical page, we need to map it to the process's virtual address space. This is done using the `mappages()` function, which updates the page tables of the current process. `mappages()` sets up the mapping between the `rounded_addr` (the virtual page address) and the newly allocated physical page, ensuring that future accesses to this memory will not cause further page faults.

By following these steps, the page fault is resolved, and the process can continue executing as if the memory had been allocated all along. Lazy allocation ensures that memory is only allocated when needed, optimizing the overall memory usage of the system.

```c
int lazyAllocate() {
    struct proc *p = myproc();  // Get the current process
    char *mem;
    uint faulting_addr, aligned_addr;

    // Retrieve the faulting address from CR2 register
    faulting_addr = rcr2();

    // Align the faulting address to the nearest page boundary
    aligned_addr = PGROUNDDOWN(faulting_addr);

    // Allocate a new physical page
    mem = kalloc();
    if (mem == 0) {
        // If memory allocation fails, return an error code
        cprintf("kalloc failed for addr: %x\n", aligned_addr);
        return -1;
    }

    // Clear the allocated memory (set all bytes to 0)
    memset(mem, 0, PGSIZE);

    // Map the new page into the process's address space
    if (mappages(p->pgdir, (void *)aligned_addr, PGSIZE, V2P(mem), PTE_W | PTE_U) < 0) {
        kfree(mem);  // Free the allocated memory if mapping fails
        return -1;
    }

    return 0;  // Return success
}
```

To enable the use of lazyAllocate() function in `trap.c`, the following declaration is added in `trap.c`.

```c
int lazyAllocate();  // Declaration for the lazy allocation function in vm.c
```

Understanding `mappages()`

The `mappages()` function is responsible for mapping a virtual address to a physical address within a process's page table. It takes the following parameters:

- The **page directory** of the current process (which holds the mappings for its virtual address space).
- The **virtual address** of the starting point of the memory to be mapped.
- The **size of the data** to be mapped (in bytes).
- The **physical address** of the memory page, which we derive by using the V2P macro. This macro converts a kernel virtual address to a physical address by subtracting KERNBASE from it.
- The **permissions** associated with the page table entry, such as read/write and user/supervisor permissions.

Breaking Down `mappages()`

The `mappages()` function processes the memory pages between a starting address and an ending address. Let's take a closer look at how it works:

1. **Calculating the Page Range**: The function first calculates the first page (`a`) and the last page (`last`) that need to be mapped. These pages correspond to the virtual address of the starting point and the ending point of the memory region that needs to be mapped. `PGROUNDDOWN()` is typically used to align the virtual address to a page boundary.
2. **Loop to Map Pages**: The function then enters a loop, running from the first page (`a`) to the last page. In each iteration, it maps one page of virtual memory to a physical memory page.
3. **Using `walkpgdir()`**: To perform the mapping, `mappages()` uses `walkpgdir()`, a function that helps navigate the process's two-level page table structure. The page table in xv6 is split into two levels: the **page directory** and the **page table**. `walkpgdir()` takes a page directory and a virtual address as input and returns the corresponding **page table entry (PTE)** for that virtual address.
   Here's how `walkpgdir()` works:
   - It extracts the first 10 bits of the virtual address using the `PDX()` macro. This gives the **page directory entry (PDE)**, which points to the appropriate page table.
   - It then extracts the next 10 bits using the `PTX()` macro to identify the **page table entry (PTE)** within the page table.
   - Finally, `walkpgdir()` returns the PTE, which will be updated with the physical address and relevant permissions during the mapping.
4. **Mapping the Page**: For each page, the physical address is written into the page table entry. The virtual address is then mapped to the physical memory page, and permissions (such as read, write, or execute) are set. This ensures that the virtual address space of the process points to valid physical memory.

Key Steps:

- **Virtual to Physical Address Conversion**: The physical address that we pass to `mappages()` is obtained using the `V2P()` macro. This macro converts a kernel virtual address to its corresponding physical address by subtracting `KERNBASE`, which is the base of the kernel's virtual address space.
- **Two-Level Page Table Navigation**:
  - The **PDX()** macro extracts the upper 10 bits of the virtual address to index into the page directory.
  - The **PTX()** macro extracts the next 10 bits to index into the page table.
- **Walkpgdir()**: This function returns the **Page Table Entry (PTE)** for a given virtual address, enabling the mapping of virtual addresses to physical addresses.

By performing this process for each page between the start and end addresses, `mappages()` ensures that the virtual addresses requested by the process are properly

mapped to physical memory pages, allowing the process to access memory without causing further page faults.

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

In this process, when the **page table** corresponding to a **page directory entry** is already present in memory, we retrieve its starting address. To do this, we use the `PTE_ADDR()` macro. This macro clears the last 12 bits of the address (which represent the offset) and gives us the address aligned to the start of the page table. The pointer to this first entry in the page table is then stored in a variable, usually named `pgtab`.

If the page table isn't already in memory, meaning there is no valid entry in the page directory, we allocate a new page table and load it. After loading the page table, we update the permissions and mark the corresponding page directory entry to indicate that the page table is now present. This is done by setting appropriate permission bits in the **page directory entry (PDE)**.

Once the page table is loaded or found in memory, the function proceeds to return a pointer to the **Page Table Entry (PTE)** that corresponds to the virtual address being handled. This entry will be used to map the virtual address to the corresponding physical address.

Handling Virtual Address Mapping in `mappages()`

At this point, `mappages()` has the relevant PTE for the virtual address. The next step is to check whether this PTE is already in use:

1. **Checking the PRESENT Bit**:
    - If the **PRESENT** bit of the page table entry is set, it means that this virtual address is already mapped to a physical page. In such cases, `mappages()` generates an error, signaling that a "remap" has occurred. A remap error indicates that the system is trying to map a virtual address to a physical page that has already been mapped, which is not allowed.
2. **Mapping the Virtual Address**:
    - If the PRESENT bit is not set, meaning the page table entry is free, `mappages()` proceeds to associate this entry with the physical page. The physical address is written into the page table entry, and the permission bits (such as read/write and user/kernel access) are set according to the page's intended use.
    - After mapping the physical address to the virtual address, the PRESENT bit is set, indicating that this page table entry now corresponds to a valid virtual-to-physical mapping.

By setting the PRESENT bit, we ensure that future accesses to the virtual address will find the corresponding physical page in memory, and no further page faults will occur for this page (unless the page is swapped out or the process terminates). This entire process allows the system to manage memory efficiently, ensuring that physical pages are allocated and mapped only when required.

Summary of Key Steps:

- **PTE_ADDR Macro**: Used to obtain the starting address of a page table by clearing the last 12 bits of the address.

- **Page Table Allocation**: If the page table is not already present, a new page table is allocated, and the permissions are updated in the page directory entry.
- **Checking the PRESENT Bit**: If the virtual address is already mapped, an error is generated (remap error). If not, the virtual address is mapped to a physical page.
- **Setting the PRESENT Bit**: Once the virtual address is mapped, the PRESENT bit in the PTE is set, indicating that this entry is valid and mapped.

This process ensures that every virtual address requested by a process is properly mapped to a corresponding physical page, and the system can handle any page faults effectively by allocating memory only when needed (lazy memory allocation).

ANSWERS TO PART-B QUESTIONS

The memory management in xv6 involves tracking physical pages and limiting the number of processes through several key mechanisms. Here's an explanation of the points raised:

1. How does the kernel know which physical pages are used and unused?

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

The xv6 kernel maintains a **linked list** of free pages in the file `kalloc.c`. This list is managed within a structure called `kmem`, which keeps track of available pages.

- **Linked List of Free Pages**: The linked list is used to track free physical memory pages. Every free page is represented as a node in this list.

2. What data structures are used to track free memory?

struct run: Each node in the linked list is a structure defined in `kalloc.c` called `struct run`. Free physical pages are typecast to `struct run` so that they can be inserted into the linked list when they are freed.

freelist: This linked list of free pages is called `freelist`. It is declared inside the `kmem` structure and is responsible for tracking all the free memory pages in the system.

3. Where does this data structure reside?

The `freelist` linked list is declared inside the `kalloc.c` file within the `kmem` structure. Each node of this linked list is of type `struct run`, and free pages are represented as elements of this linked list.

When a page is freed, it is inserted into this `freelist` using the function `kfree()`:

4. Does the memory management mechanism limit the number of user processes?

Yes, the memory management in xv6 imposes a limit on the number of user processes, due to the maximum size of the **process table (`ptable`)**. The `ptable` contains a fixed number of entries, each representing a process.

- **NPROC**: The maximum number of user processes is limited by the `NPROC` constant, which is defined in `param.h`. By default, `NPROC` is set to 64, meaning that xv6 can handle a maximum of 64 user processes at any given time.

5. What is the lowest number of processes xv6 can have at the same time, assuming the kernel requires no memory?

From a memory management perspective, the lowest number of user processes that xv6 can have is **1**, as explained below:

- **initproc**: When the xv6 operating system boots, there is only one process called `initproc`. This process is created during boot and is responsible for forking the first user shell process (`sh`). All other user processes are eventually forked from this shell process.

Therefore, the lowest number of processes after boot is **1** (the `initproc`). This is because a user shell process is required to interact with the system, and it cannot function without at least this one process. In practice, there cannot be zero processes since user interaction requires at least one active user process.

Additionally, since the virtual address space is limited to **4 GB** (`KERNBASE`), and the maximum physical memory available is assumed to be **128 MB** (`PHYSTOP`), one user process could theoretically occupy all the available physical memory. However, this is a theoretical limit, and the actual number of processes will depend on memory allocation and management policies.

PART B(All Tasks)

Task 1: Kernel Process Creation (`create_kernel_process()`)

1. **Kernel-Only Process**:

- The function `create_kernel_process()` is implemented in `proc.c` to create a kernel-only process that will remain in kernel mode at all times. Unlike user processes, it does not need a trapframe since it doesn't switch between user mode and kernel mode.
- **Trapframe Omission**: Since the process stays in kernel mode, it doesn't need to store user register values, so the initialization of the trapframe can be skipped.

2. **Setting the `eip` Register**:
   - The `eip` (instruction pointer) in the process context is set to the entry point of the function that this kernel process is supposed to execute. This ensures that the process begins execution at the specified function.

3. **Allocating Process Table Entry**:
   - The `allocproc()` function is responsible for assigning a process slot in the `ptable` (process table) for the kernel process, similar to how it would for a regular process.
4. **Setting Up the Kernel Page Table**:
   - Since this is a kernel-only process, the virtual address mapping only needs to consider addresses above KERNBASE, and the page table is set up to map virtual addresses to physical addresses between `0` and PHYSTOP (which is the upper limit of physical memory).

```
void create_kernel_process(const char *name, void (*entrypoint)()){

  struct proc *p = allocproc(); // Allocate a new process structure.

  if(p == 0)
    panic("create_kernel_process failed");  // Panic if process allocation fails.

  // Initialize the kernel page table for the process.
  // `setupkvm` creates a new page directory with only kernel mappings.
  if((p->pgdir = setupkvm()) == 0)
    panic("setupkvm failed");

  // This is a kernel-only process, so no trap frame (tf) is needed.
  // Since it doesn't interact with user space, we don't need to assign any memory for user space.

  // Set the entry point of the process.
  // `eip` (Extended Instruction Pointer) holds the address of the first instruction to execute.
  p->context->eip = (uint)entrypoint;

  // Safely copy the process name into the process structure.
  safestrcpy(p->name, name, sizeof(p->name));

  // Make the process runnable by setting its state to RUNNABLE.
  acquire(&ptable.lock);  // Lock the process table to ensure consistency.
  p->state = RUNNABLE;    // Mark the process as ready to run.
  release(&ptable.lock);  // Release the process table lock.

}
```

Task 2:

Task 2 focuses on creating a circular process queue (`rq`) that tracks processes denied additional memory due to the lack of free pages. This queue manages processes with swap-out requests.

We implemented two key functions:

1. **`rp_sh()`**: Adds a process to the queue when it can't get more memory.
2. **`rpop()`**: Removes a process from the queue when memory becomes available.

The queue is synchronized using a lock, initialized in `pinit()`, to ensure thread safety. The start (s) and end (e) pointers of the queue are initialized in `serinit()`.

To allow access to these queue functions across the codebase, their prototypes are declared in `defs.h`.

In proc.c:

```
struct rq{
  struct spinlock lock;
  struct proc* queue[NPROC];
  int s;
  int e;
};

//for swapping out requests, we use this circular request queue
struct rq rqueue;

struct proc* rpop(){

  acquire(&rqueue.lock);
  if(rqueue.s==rqueue.e){
    release(&rqueue.lock);
    return 0;
  }
  struct proc *p=rqueue.queue[rqueue.s];
  (rqueue.s)++;
  (rqueue.s)%=NPROC;
  release(&rqueue.lock);

  return p;
}

int rpush(struct proc *p){

  acquire(&rqueue.lock);
  if((rqueue.e+1)%NPROC==rqueue.s){
    release(&rqueue.lock);
    return 0;
  }
  rqueue.queue[rqueue.e]=p;
  rqueue.e++;
  (rqueue.e)%=NPROC;
  release(&rqueue.lock);

  return 1;
}
```

```c
void pinit(void){
  initlock(&ptable.lock, "ptable");
  initlock(&rqueue.lock, "rqueue");
  initlock(&sleeping_channel_lock, "sleeping_channel");
  initlock(&rqueue2.lock, "rqueue2");
}
```

```c
void userinit(void){
  acquire(&rqueue.lock);
  rqueue.s=0;
  rqueue.e=0;
  release(&rqueue.lock);

  acquire(&rqueue2.lock);
  rqueue2.s=0;
  rqueue2.e=0;
  release(&rqueue2.lock);

  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

  release(&ptable.lock);

}
```

```
struct stat;
struct superblock;
struct rq;
```

```
extern int swap_out_process_exists;
extern int swap_in_process_exists;
extern struct rq rqueue;
extern struct rq rqueue2;
int rpush(struct proc *p);
struct proc* rpop();
struct proc* rpop2();
int rpush2(struct proc* p);
```

Whenever `kalloc()` fails to allocate memory to a process, it returns zero, signaling `allocuvm()` that memory wasn't allocated (`mem=0`). At this point, the process state is changed to "sleeping."

Note: The process sleeps on a special channel, `sleeping_channel`, which is secured by `sleeping_channel_lock`. Another channel, `sleeping_channel_cont`, handles specific cases during system boot.

After changing the process state, we add the current process to the swap-out request queue, `rq`.

Global declarations for the queue are also added to `defs.h` as external variables.

```
struct spinlock sleeping_channel_lock;
int sleeping_channel_count=0;
char * sleeping_channel;
```

```
int allocuvm(pde_t *pgdir, uint oldsz, uint newsz){
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      // cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);

      // sleep part
      myproc()->state=SLEEPING;
      acquire(&sleeping_channel_lock);
      myproc()->chan=sleeping_channel;
      sleeping_channel_count++;
      release(&sleeping_channel_lock);

       rpush(myproc());
      if(!swap_out_process_exists){
        swap_out_process_exists=1;
        create_kernel_process("swap_out_process", &swap_out_process_function);
      }

      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
  return newsz;
}
```

Note: The `create_kernel_process` function is used to create a swap-out kernel process, which allocates memory for the current process if it couldn't get a page. If the swap-out process doesn't already exist, it's created, and the variable `swap_out_process_exists` (defined in `defs.h` and initialized to 0 in `proc.c`) is set to 1. This prevents multiple swap-out processes from being created. Once the swap-out process finishes, `swap_out_process_exists` is reset to 0. The details of the swap-out process are explained later.

Next, we set up a system where, whenever free pages become available, all processes that were waiting for memory (sleeping on `sleeping_channel`) are woken up. To do this, we modify `kfree()` in `kalloc.c` to call the `wakeup()` system call, which wakes up all processes sleeping on `sleeping_channel` due to a lack of memory.

```
void
kfree(char *v)
{

  struct run *r;
  // struct proc *p=myproc();


  // Validate the address being freed:
  // - Must be page-aligned.
  // - Must not point to memory before the kernel's end address.
  // - Must not exceed physical memory limits.
  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP){
    panic("kfree");
  }

  // Overwrite the memory with 1s to detect any dangling references.
  // memset(v, 1, PGSIZE);
  for(int i=0;i<PGSIZE;i++){
    v[i]=1;
  }

  // Lock the memory system if required and add the page to the free list.
  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);

  // If any processes are sleeping on the channel, wake them up.
  if(kmem.use_lock)
    acquire(&sleeping_channel_lock);
  if(sleeping_channel_count){
    wakeup(sleeping_channel);
    sleeping_channel_count=0;
  }
  if(kmem.use_lock)
    release(&sleeping_channel_lock);

}
```

Now, let us see the swap-out process. The entry point for this process is the `swap_out_process_function`.

```
void swap_out_process_function(){

  acquire(&rqueue.lock);
  while(rqueue.s!=rqueue.e){
    struct proc *p=rpop();

    pde_t* pd = p->pgdir;
    for(int i=0;i<NPDENTRIES;i++){

      //skip page table if accessed
      if(pd[i]&PTE_A)
        continue;
      pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pd[i]));
      for(int j=0;j<NPTENTRIES;j++){

        if((pgtab[j]&PTE_A) || !(pgtab[j]&PTE_P))
          continue;
        pte_t *pte=(pte_t*)P2V(PTE_ADDR(pgtab[j]));

        int pid=p->pid;
        int virt = ((1<<22)*i)+((1<<12)*j);

        char c[50];
        int_to_string(pid,c);
        int x=strlen(c);
        c[x]='_';
        int_to_string(virt,c+x+1);
        safestrcpy(c+strlen(c),".swp",5);

        int fd=proc_open(c, O_CREATE | O_RDWR);
        if(fd<0){
          cprintf("error creating or opening file: %s\n", c);
          panic("swap_out_process");
        }

        if(proc_write(fd,(char *)pte, PGSIZE) != PGSIZE){
          cprintf("error writing to file: %s\n", c);
          panic("swap_out_process");
        }
        proc_close(fd);

        kfree((char*)pte);
        memset(&pgtab[j],0,sizeof(pgtab[j]));

        pgtab[j]=((pgtab[j])^(0x080));

        break;
      }
    }
  }

  release(&rqueue.lock);

  struct proc *p;
  if((p=myproc())==0)
    panic("swap out process");

  swap_out_process_exists=0;
  p->parent = 0;
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED;
  sched();
}
```

The process keeps running in a loop until the swap-out request queue (`rq_queue1`) is empty. The queue stores processes that need to free up memory. When the queue becomes empty, the swap-out process terminates.

Inside the Loop:
The loop begins by taking the first process from `rq_queue1`. It then uses the **Least Recently Used (LRU)** policy to find a "victim page" (a page that hasn't been used recently) in the process's page table.

For each process in the queue, we go through its page table (`pgdir`). The function checks every entry in the secondary page tables (these hold actual mappings for the process's memory) to find pages that haven't been used recently. We extract the physical address of each secondary page table and examine all its entries.

Checking the Accessed Bit (A Bit):
For each page table entry, we look at the **Accessed Bit (A bit)**, which is the sixth bit from the right. This bit indicates whether a page has been accessed recently. To check if the bit is set, we use a bitwise AND operation on the entry and `PTE_A` (which we defined as 32 in `mmu.c`).

Why the Accessed Flag Matters:
It's important to note that whenever a process is switched in and out by the scheduler, all the accessed bits in its page table are reset to 0. Therefore, when the `swap_out_process_function` examines the accessed bit, it tells us whether the page was used during the process's last execution. If the accessed bit is still set, it means the page was accessed, and it might not be a good candidate for swapping out.

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

      //If the swap out process has stopped running, free its stack and name.
      if(p->state==UNUSED && p->name[0]=='*'){

        kfree(p->kstack);
        p->kstack=0;
        p->name[0]=0;
        p->pid=0;
      }

      if(p->state != RUNNABLE)
        continue;

      for(int i=0;i<NPDENTRIES;i++){
        //If PDE was accessed

        if(((p->pgdir)[i])&PTE_P && ((p->pgdir)[i])&PTE_A){

          pte_t* pgtab = (pte_t*)P2V(PTE_ADDR((p->pgdir)[i]));

          for(int j=0;j<NPTENTRIES;j++){
            if(pgtab[j]&PTE_A){
              pgtab[j]^=PTE_A;
            }
          }

          ((p->pgdir)[i])^=PTE_A;
        }
      }

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

In the **scheduler**, there is a section of code that clears (unsets) the **Accessed Bit (A bit)** for every page in a process's page table and secondary page tables. This is necessary so that when the `swap_out_process_function` later runs, it can accurately identify which pages haven't been accessed recently.

Back to the `swap_out_process_function`:
Once the function finds a page table entry where the accessed bit is unset, it selects the corresponding physical page as the **victim page** (i.e., the page to be swapped out). The function determines the **physical page number** using certain macros (previously explained in Part A of the report). This page is then swapped out, meaning its contents are saved to the hard drive.

File Naming for Swapped Pages:
The page's data needs to be saved into a file. To create a unique filename for each swapped-out page, the process's **pid (process ID)** and the **virtual address** of the page are used. These two pieces of information are combined to form a filename, which looks like `<pid>_<virt>.swap`.

To help generate the filename, we created a function called `int_to_string` (declared in `proc.c`). This function converts integers like the **pid** and **virtual address (virt)** into strings, which are then used to form the filename.

```
void int_to_string(int x, char *c){
  if(x==0)
  {
    c[0]='0';
    c[1]='\0';
    return;
  }
  int i=0;
  while(x>0){
    c[i]=x%10+'0';
    i++;
    x/=10;
  }
  c[i]='\0';

  for(int j=0;j<i/2;j++){
    char a=c[j];
    c[j]=c[i-j-1];
    c[i-j-1]=a;
  }

}
```

Problem with File Operations in `proc.c`:
Normally, file system operations (like opening, writing, reading, and closing files) are not allowed in `proc.c`. The standard file system functions reside in `sysfile.c`, and they take arguments in a way that's not compatible with `proc.c`.

To overcome this, we copied the file system functions (like `open`, `write`, `read`, and `close`) from `sysfile.c` to `proc.c`, made necessary adjustments to how they handle arguments, and renamed them (e.g., to `proc_open`, `proc_write`, `proc_read`, and `proc_close`). This way, we can use file operations within `proc.c` for the swap-out process.

```
int
proc_close(int fd)
{
  struct file *f;

  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;

  myproc()->ofile[fd] = 0;
  fileclose(f);
  return 0;
}

int
proc_write(int fd, char *p, int n)
{
  struct file *f;
  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;
  return filewrite(f, p, n);
}
```

Writing Pages to Storage:

To save the selected page to disk, we use the **`proc_open`** function to create or open a file with **O_CREATE** and **O_RDWR** permissions, which we defined in **`fcntl.h`**. These permissions allow the file to be created if it doesn't already exist and allow both reading and writing. The file descriptor is stored in a variable called **`fd`**. After that, we use **`proc_write`** to write the contents of the selected page to this file.

Once the page is successfully written to disk, we add the page back to the free page queue using **`kfree`**, so it becomes available for future allocations. Additionally, when a page is added back to the free queue, all processes that were sleeping due to a lack of memory (waiting on `sleeping_channel`) are woken up.

We then **clear the page table entry** for the swapped-out page using `memset`.

Marking Pages as Swapped Out:

To keep track of whether a page was swapped out or not (for Task 3), we use the 8th bit in the page table entry. We mark this by setting the **7th bit (2^7)** in the corresponding secondary page table entry, using the **OR operator** to set this bit.

Suspending the Kernel Process:

Once the swap-out request queue (rq) is empty, the swap-out process suspends. This involves two steps:

1. **Inside the Process:**
   We cannot clear the kernel stack (`kstack`) from within the process itself, as the process would lose track of what to do next. Instead, we preempt the process, allowing the scheduler to handle its termination.

```
struct proc *p;
if((p=myproc())==0)
  panic("swap out process");

swap_out_process_exists=0;
p->parent = 0;
p->name[0] = '*';
p->killed = 0;
p->state = UNUSED;
sched();
}
```

2. **From the Scheduler:**
   When the scheduler finds a kernel process in the **UNUSED** state, it clears the process's `kstack` and resets its name. This is done by changing the first character of the process name to **'*'** when the process ends. The scheduler uses this flag to identify and clean up unused kernel processes.

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

  //free the stack and name if the process that has been swapped out has stopped running
  if(p->state==UNUSED && p->name[0]=='*'){

    kfree(p->kstack);
    p->kstack=0;
    p->name[0]=0;
    p->pid=0;
  }
```

Key Assignments Requirements Achieved:

- **Request Queue for Swap-out:** We implemented a queue for swap-out requests (as seen in earlier parts of the report).
- **Process Suspension:** The swap-out process suspends itself when no requests are pending.

- **Waking Up Suspended Processes:** When free memory becomes available, all processes that were sleeping due to lack of memory are woken up using `kfree` and `sleeping_channel`.
- **Swapping Only User Space Memory:** Only user-space memory can be swapped out, and since we iterate through page tables top-down (from virtual addresses below **KERNBASE**), we only swap out user-space pages that weren't accessed in the last iteration.

Task 3:

In this task, we set up a new swap request queue, referred to as `rq2`, in `proc.c`, similar to the previous queue established in Task 2. We also declare an external prototype for `rq2` in `defs.h`.

Key Steps:

1. **Queue Creation:**
   We created the swap request queue `rq2` along with the necessary functions for managing it, specifically `rpop2()` and `rpush2()`, in `proc.c`.
2. **Prototype Declaration:**
   Prototypes for these new functions were added to `defs.h` to ensure they can be accessed from other files.
3. **Initialization:**
   The lock for the new queue was initialized in the `pinit` function, and any state variables related to the queue were set up in `userinit`.
4. **Struct Modification:**
   We added an additional field, `addr` (of type `int`), to the `proc` structure defined in `proc.h`. This new field will be used to store the virtual address where a page fault occurred, which is critical for managing the swapping process.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int addr;                    // ADDED: Virtual address of pagefault

};
```

Handling Page Faults in Task 3

In this section, we focus on managing page faults (T_PGFLT traps) in `trap.c` through the `handlePageFault()` function.

```
void handlePageFault(){
  int addr=rcr2();
  struct proc *p=myproc();
  acquire(&swap_in_lock);
  sleep(p,&swap_in_lock);
  pde_t *pde = &(p->pgdir)[PDX(addr)];
  pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));

  if((pgtab[PTX(addr)])&0x080){
    //virtual address pf page that has been swapped out
    p->addr = addr;
    rpush2(p);
    if(!swap_in_process_exists){
      swap_in_process_exists=1;
      create_kernel_process("swap_in_process", &swap_in_process_function);
    }
  } else {
    exit();
  }
}
```

```
case T_PGFLT:
  handlePageFault();
break;
```

Key Steps:

1. **Identifying the Faulting Address:**
   - Inside `handlePageFault()`, we utilize `rcr2()` to retrieve the virtual address where the page fault occurred.
2. **Process Suspension:**
   - The current process is put to sleep using a newly introduced lock called `swap_in_lock`, which is initialized in `trap.c` and declared as an extern variable in `defs.h`.
3. **Page Table Entry Retrieval:**
   - We obtain the page table entry corresponding to the faulting address. This logic mirrors that of the `walkpgdir()` function.
4. **Check for Swapped Out Page:**
   - To determine if the page was swapped out, we check the 7th bit (2^7) of the page table entry using the bitwise AND operation with `0x80`. This bit indicates whether the page has been swapped out (as established in Task 2).
5. **Initiating Swap-in Process:**
   - If the 7th bit is set, we check if the swap-in process (`swap_in_process`) already exists by referencing the variable `swap_in_process_exists`. If it doesn't exist, we initiate it.
   - If the page is not swapped out, we safely suspend the process using `exit()` as required by the assignment.
6. **Swap-in Process Function:**
   - The entry point for the swap-in process is defined in the `swap_in_process_function`, declared in `proc.c`. The details of this function will be discussed in the next section.
7. **File Management:**
   - The function operates a loop that continues until the `rq2` swap request queue is empty. In each iteration:
     - We pop a process from the queue and extract its PID and the address value.
     - We create a filename string c using the `int_to_string()` function described in Task 2.
     - We open the corresponding file in read-only mode (`O_RDONLY`) using `proc_open`.
8. **Memory Allocation and Reading:**
   - We allocate a free memory frame for the process using `kalloc()`.
   - The file is read into this allocated frame with `proc_read` using the file descriptor.
9. **Mapping the Page:**
   - We then utilize `mappages()` to map the newly allocated physical page to the corresponding virtual address, completing the swap-in process.
10. **Waking the Process:**
    - After successfully allocating and mapping the page, we wake the previously sleeping process using `wakeup()`.

11. **Termination of the Swap-in Process:**
    - Once the loop concludes and all requests have been processed, we execute the termination instructions for the kernel process, as detailed earlier in the report.

```
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```

```
void swap_in_process_function(){

  acquire(&rqueue2.lock);
  while(rqueue2.s!=rqueue2.e){
    struct proc *p=rpop2();

    int pid=p->pid;
    int virt=PTE_ADDR(p->addr);

    char c[50];
      int_to_string(pid,c);
      int x=strlen(c);
      c[x]='_';
      int_to_string(virt,c+x+1);
      safestrcpy(c+strlen(c),".swp",5);

      int fd=proc_open(c,O_RDONLY);
      if(fd<0){
        release(&rqueue2.lock);
        cprintf("could not find page file in memory: %s\n", c);
        panic("swap_in_process");
      }
      char *mem=kalloc();
      proc_read(fd,PGSIZE,mem);

      if(mappages(p->pgdir, (void *)virt, PGSIZE, V2P(mem), PTE_W|PTE_U)<0){
        release(&rqueue2.lock);
        panic("mappages");
      }
      wakeup(p);
  }

    release(&rqueue2.lock);
    struct proc *p;
  if((p=myproc())==0)
    panic("swap_in_process");

  swap_in_process_exists=0;
  p->parent = 0;
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED;
  sched();

}
```

In this task, all specified objectives were successfully achieved, ensuring efficient handling of page faults and memory management.

Task 4: Sanity Test

In this section, our goal is to create a testing method to verify the functionalities we developed in earlier tasks. We will implement a user-space program called `memtest` for this purpose. The implementation of `memtest` is outlined below.

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int math_func(int num) { return num * num - 4 * num + 1; }

int main(int argc, char *argv[])
{
    for (int i = 0; i < 20; i++)
    {
        if (!fork())
        {
            printf(1, "Child %d\n", i + 1);
            printf(1, "iteration matching differences\n\n");
            for (int j = 0; j < 10; j++)
            {
                int *arr = malloc(4096);
                for (int k = 0; k < 1024; k++)
                    arr[k] = math_func(k);
                int matched = 0;
                for (int k = 0; k < 1024; k++)
                    if (arr[k] == math_func(k))
                        matched += 4;

                if (j < 9)
                    printf(1, "    %d      %dB      %dB\n", j + 1, matched, 4096 - matched);
                else
                    printf(1, "   %d      %dB      %dB\n", j + 1, matched, 4096 - matched);
            }
            printf(1, "\n");
            exit();
        }
    }
    while (wait() != -1)
        ;
    exit();
}
```

Implementation Overview:

1. **Includes Necessary Headers:**
    - The program includes standard headers like `types.h` and `user.h` to access required functions and types.
2. **Defines a Mathematical Function:**

    - A function called `math_func(int num)` is defined. It returns a value calculated using the formula `num*num - 4*num + 1`.
3. **Main Function Logic:**
    - Inside the `main()` function, we create two child processes using the `fork()` system call.
    - Each child process enters a loop that runs 10 times.
4. **Memory Allocation:**
    - In each iteration, the program allocates 4096 bytes (or 4KB) of memory using the `malloc()` function.
5. **Calculating Values:**
    - The value stored at each index of the allocated memory array is determined by the mathematical expression `i*i - 4*i + 1`, which is computed by calling `math_func()`.
6. **Tracking Correct Values:**
    - A counter named `matched` is maintained to track the amount of memory that contains the correct values.
    - This is done by checking the value stored at each index against the value returned by the `math_func()` for that index.
7. **Output:**
    - At the end of the execution, the program prints whether the expected values matched or not.

This program serves to test the memory management and process handling functionalities developed in the previous tasks.

Running `memtest`

To run the `memtest` program, we need to include it in the Makefile under the sections labeled `UPROGS` and `EXTRA`. This makes it accessible to the xv6 user.

Output from Running `memtest`:

Upon executing `memtest`, we see the following output:

```
$ memtest
Child 1
iteration matching differences

    1       4096B       0B
    2       4096B       0B
    3       4096B       0B
    4       4096B       0B
    5       4096B       0B
    6       4096B       0B
    7       4096B       0B
    8       4096B       0B
    9       4096B       0B
   10       4096B       0B

Child 2
iteration matching differences

    1       4096B       0B
    2       4096B       0B
    3       4096B       0B
    4       4096B       0B
    5       4096B       0B
    6       4096B       0B
    7       4096B       0B
    8       4096B       0B
    9       4096B       0B
   10       4096B       0B

Child 3
iteration matching differences

    1       4096B       0B
    2       4096B       0B
    3       4096B       0B
    4       4096B       0B
    5       4096B       0B
    6       4096B       0B
    7       4096B       0B
    8       4096B       0B
    9       4096B       0B
   10       4096B       0B

Child 4
iteration matching differences

    1       4096B       0B
    2       4096B       0B
    3       4096B       0B
    4       4096B       0B
    5       4096B       0B
    6       4096B       0B
    7       4096B       0B
    8       4096B       0B
    9       4096B       0B
   10       4096B       0B
```

Implementation Result:

From the output, we can observe that our implementation passes the sanity test since all the indices hold the correct values.

Further Testing:

To further evaluate our implementation, we perform tests with different values for PHYSTOP, which is defined in `memlayout.h`. The default value for PHYSTOP is `0xE0000000` (224MB). We changed this value to `0x400000` (4MB). We selected 4MB because it is the minimum memory required by xv6 to execute the `kinit` function.

After running `memtest` with this new value, we find that the output is identical to the previous results, indicating that the implementation is correct.