VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEM (CO2017)

Assignment

"Simple Operating System"

Instructor(s): Hoàng Lê Hải Thanh

Students: Nguyễn Thái Học - 2311100 *(Group TN01 - Team 02, **Leader**)*
Lê Nguyễn Kim Khôi - 2311671 *(Group TN01 - Team 02)*
Hồ Anh Dũng - 2310543 *(Group TN01 - Team 02)*
Nguyễn Thiện Minh - 2312097 *(Group TN01 - Team 02)*
Huỳnh Đức Nhân - 2312420 *(Group TN01 - Team 02)*

HO CHI MINH CITY, APRIL 2025

Contents

List of Figures

List of Tables

Member list & Workload

No.	Fullname	Student ID	Problems	% done
1	Nguyễn Thái Học	2311100	- Implement memory section - Answering memory questions - Put it all together	100%
2	Lê Nguyễn Kim Khôi	2311671	- Implement memory section - Explaining the memory implementation - Put it all together	100%
3	Huỳnh Đức Nhân	2312420	- Implement sheduling, syscall section - Explaining schedule, syscall implementation - Answering syscall questions	100%
4	Hồ Anh Dũng	2310543	- Implement sheduling and syscall section - Shedule, syscall testcase - Answering schedule, syscall questions	100%
5	Nguyễn Thiện Minh	2312097	- Implement memory section - Create, explain Memory Testcase - Put it all together	100%

Table 1: Member list & workload

1 Queue Implementation Strategy

In our project, the queue is implemented as a fixed-size array (`MAX_QUEUE_SIZE = 10`), with two main operations: `enqueue`, which adds a process to the queue, and `dequeue`, which removes the process with the highest priority (i.e., the lowest `prio` value).
There are three general strategies to implement this queue:

1. **Simple Enqueue + Min-Priority Dequeue:** Insert the new process at the end of the queue. During `dequeue`, scan the entire array to find the process with the minimum priority, remove it, and shift the remaining elements to fill the gap.

2. **Sorted Enqueue + Fast Dequeue:** Insert the new process while maintaining the array sorted in descending priority order. This allows `dequeue` to simply remove the last element.

3. **Hybrid Enqueue + Dequeue:** Depending on the scheduler type (`#ifdef MLQ_SCHED` for MLQ or `#else` for a single shared queue), different strategies are used. In MLQ mode, each queue contains processes of the same priority, so we can simply append the new process during `enqueue` and take the first process during `dequeue`. In single-queue mode, strategy (2) offers optimal performance.

Although strategies (2) and (3) offer better asymptotic performance for `dequeue`, they introduce additional implementation complexity, such as maintaining a sorted structure or handling dynamic memory. Given that our queue is small (size = 10) and fixed, the overhead of scanning and shifting elements in strategy (1) is negligible in practice.
Therefore, I chose to implement the queue using strategy (1). This decision is motivated by the simplicity and clarity of the approach, as well as its consistency across both scheduling modes: `#ifdef MLQ` (multi-level queues with same-priority processes in a queue) and `#else` (a single queue shared across all priorities). Using a unified queue logic also improves modularity and reusability of the queue structure, which may benefit future extensions beyond process scheduling.

```c
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    if (q == NULL || proc == NULL || q->size >= MAX_QUEUE_SIZE) {
        printf("Queue is full or NULL\n");
        return;
    }
    q->proc[q->size] = proc;
    q->size++;
}
struct pcb_t * dequeue(struct queue_t * q) {
    if (q == NULL || q->size == 0) return NULL;
    int MAX_INDEX = 0;
    #ifdef MLQ_SCHED
        uint32_t MAX = q->proc[0]->prio;
        for (int i = 1; i < q->size; i++) {
            if (q->proc[i]->prio < MAX) {
                MAX = q->proc[i]->prio;
                MAX_INDEX = i;
            }
        }
    #else
        uint32_t MAX = q->proc[0]->priority;
        for (int i = 1; i < q->size; i++) {
```

```c
            if (q->proc[i]->priority < MAX) {
                MAX = q->proc[i]->priority;
                MAX_INDEX = i;
            }
        }
    #endif
    struct pcb_t * ret_proc = q->proc[MAX_INDEX];
    for (int i = MAX_INDEX; i < q->size - 1; i++) {
        q->proc[i] = q->proc[i + 1];
    }
    q->size--;
    return ret_proc;
}
```

We also implement an extra function, `remove_proc`, which removes a specified process from the queue.

2 Scheduling Logic and Design Decisions

The core of the scheduler implementation consists of several key functions that manage the selection and movement of processes for execution. While some functions are relatively straightforward, the primary complexity lies in the logic of `get_proc`, particularly under the multi-level queue (MLQ) model.

2.1 Process Management Functions

Basic functions such as `put_proc` and `add_proc` are implemented with minimal logic:

- `add_proc`: This function is called by `ld_routine`, meaning it is used when a process is loaded. The process is simply enqueued into either `mlq_ready_queue` or `ready_queue`, and the process's pointer is set accordingly.
- `put_proc`: After `cpu_routine` retrieves a process from `get_proc`, it calls `put_proc` to reinsert the previously running process into the appropriate queue. This function removes the process from `running_list` (using `remove_proc`) and enqueues it into the ready queue.

```c
void put_proc(struct pcb_t * proc) {
    proc->ready_queue = &ready_queue;
    proc->mlq_ready_queue = mlq_ready_queue;

    pthread_mutex_lock(&queue_lock);
    enqueue(&mlq_ready_queue[proc->prio], proc);
    remove_proc(&running_list, proc);
    proc->running_list = NULL;
    pthread_mutex_unlock(&queue_lock);

    return;
}

void add_proc(struct pcb_t * proc) {
    proc->ready_queue = &ready_queue;
    proc->mlq_ready_queue = mlq_ready_queue;
```

```

    return add_mlq_proc(proc);
}
```

Additionally, the `running_list` is used to track processes that are currently running. So if a process is in ready queue, running_list pointer must be NULL.
Depending on the scheduling mode (controlled via `#ifdef MLQ`), `get_proc` returns a process from the appropriate queue. In non-MLQ mode, this is done by simply dequeuing the highest-priority process from the `ready_queue`. In MLQ mode, the logic is more involved:

- The scheduler maintains an array of queues, one for each priority level up to `MAX_QUEUE = 140`.
- Each queue has an associated `time_slot`, calculated as `MAX_QUEUE - prio`, meaning that higher-priority queues are allowed more execution slots.
- The `get_proc` function iterates through the queues in order of priority and attempts to retrieve a process from the first queue that has remaining time slots.

A critical design decision was determining when to reset the time slots. Several options were considered:

- Reset only when all time slots are exhausted (which may lead to starvation if processes only exist in a few priority levels).
- Reset when there are no runnable processes available (i.e., all queues are empty or blocked).

To balance safety and fairness, our implementation resets time slots under either of the following conditions:

1. All time slots have reached zero.
2. No process is runnable in the currently selected queue (i.e., `proc == NULL`).

```
struct pcb_t * get_mlq_proc(void) {
    static int reset_slot = 0;
    struct pcb_t * proc = NULL;

    pthread_mutex_lock(&queue_lock);
    for (int i = 0; i < MAX_PRIO; i++) {
        if (slot[i] == 0) continue;
        proc = dequeue(&mlq_ready_queue[i]);
        if (proc != NULL) {
            --slot[i];
            reset_slot = 1;
            break;
        }
    }

    if (proc != NULL) {
        enqueue(&running_list, proc);
        proc->running_list = &running_list;
    }
```

```
    else if (reset_slot == 1) {
        for (int i = 0; i < MAX_PRIO; i++)
            slot[i] = MAX_PRIO - i;
    }
    pthread_mutex_unlock(&queue_lock);

    return proc;
}
```

Before returning a process from `get_proc`, it is added to the `running_list`, and its pointer is updated accordingly too. To avoid premature or unnecessary resets, a flag `reset_slot` is introduced to detect changes in the slot states. This ensures that time slot resets occur only after actual usage begins, and not immediately after initialization when all time slots are still untouched.

2.2 Process Termination Functions

Since the initial codebase lacks functionality for terminating a process, we implement such functionality for use in `syscall killall`. The main function we introduce is `remove_pcb`, which removes a process from all relevant queues and deallocates its memory (delegated to the helper function `delete_pcb`).
Because our operating system does not support forcibly terminating a running process, this function is implemented to terminate only ready or finished process. If we want to terminate a process, we need to make that process finishes its job first (we will dicuss about this case in killall implementation). Function `remove_pcb` simply dequeued from all relevant queues and passed to `delete_pcb` for memory deallocation.

```
void delete_pcb(struct pcb_t *proc)
{
    if (proc->page_table != NULL) free(proc->page_table);
    if (proc->code != NULL) {
        if (proc->code->text != NULL) free(proc->code->text);
        free(proc->code);
    }
    #ifdef MM_PAGING
        if (proc->mm != NULL) {
            if (proc->mm->pgd != NULL) free(proc->mm->pgd);
            struct vm_rg_struct* head = proc->mm->mmap->vm_freerg_list;
            while (head != NULL) {
                struct vm_rg_struct* tmp = head;
                head = head->rg_next;
                free(tmp);
            }
            if (proc->mm->mmap != NULL) free(proc->mm->mmap);
            free(proc->mm);
        }
    #endif
}

void remove_pcb(struct pcb_t *proc) {
    pthread_mutex_lock(&queue_lock);

    if (proc->running_list != NULL) {
```

```
        remove_proc(proc->running_list, proc);
    }
    else {
        #ifdef MLQ_SCHED
            remove_proc(&mlq_ready_queue[proc->prio], proc);
        #else
            remove_proc(&ready_queue, proc);
        #endif
    }
    delete_pcb(proc);
    free(proc);

    pthread_mutex_unlock(&queue_lock);
}
```

3 Memory Implementation

In the Memory Management design of this simple OS implementation, the design is divided into smaller modules as follows: Libmem acts as an interface that allows users to interact with the OS; the mm-vm layer is responsible for managing virtual memory; the mm layer directly handles the interaction between the virtual memory layer and the physical memory layer; and finally, the mm-memphy layer manages the physical memory regions within the OS.

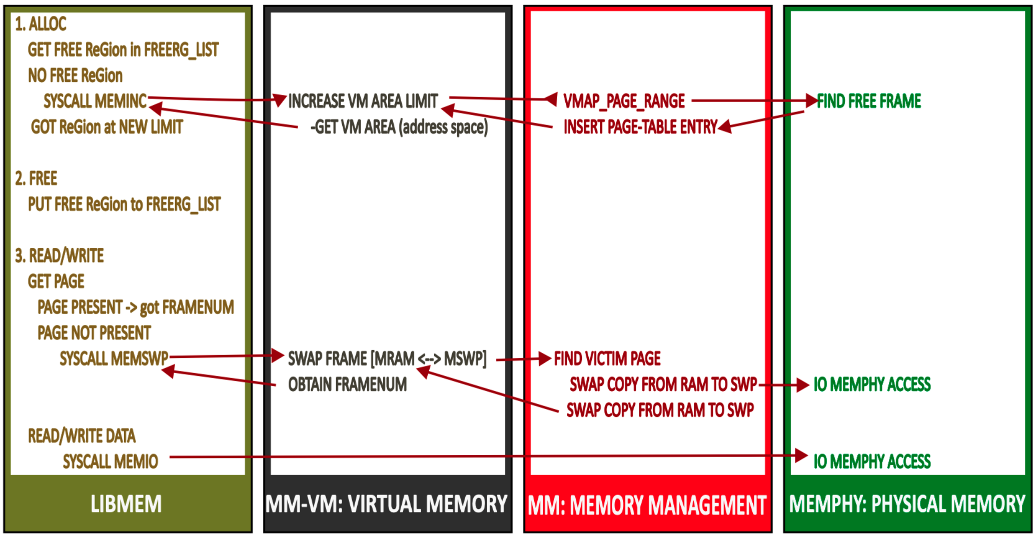


Figure 1: Memory System Modules

In order to gather the requirements from the design, it is necessary to reiterate the design assumptions that were mentioned in the problem statement for designing a basic operating system as follows.

- ALLOC: user call the library functions in libmem and in most cases, it fits into data segment area. If there is no suitable space, we need to expand the memory space by lift up the barrier set by sbrk. Since it have never been touched, it may needs to leverage some MMU systemcalls to obtains physical frames and then map them using Page Table Entry.

- FREE: user call the library functions in libmem to revoke the storage space associated with the given region id. Since we cannot reclaim the taken physical frame which might cause memory holes, we just keep the collected storage space in a free list for future alloc request, all are embedded in libmem library.
- READ/WRITE: requires to get the page to be presented in the main memory. The most resource consuming step is the page swapping. If the page is in the MEMSWAP device, it needs to be brought back to MEMRAM device (swapping in) and if there is a lack of space, we need to give back some pages to MEMSWAP device (swapping out) to make more rooms.

3.1 Alloc Implementation

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *
    alloc_addr)
{
  if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
    return -1;
  /*Allocate at the toproof */
  pthread_mutex_lock(&mmvm_lock);
  struct vm_rg_struct rgnode;

  /* TODO: commit the vmaid */
  // rgnode.vmaid

  if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
  {
    caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
    caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;

    *alloc_addr = rgnode.rg_start;

    pthread_mutex_unlock(&mmvm_lock);
    return 0;
  }

  /* TODO get_free_vmrg_area FAILED handle the region management (Fig.6)
      */

  /* TODO retrieve current vma if needed, current comment out due to
      compiler redundant warning*/
  /*Attempt to increate limit to get space */
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
  if(cur_vma == NULL){
    pthread_mutex_unlock(&mmvm_lock);
    return -1;
  }

  /* TODO INCREASE THE LIMIT as invoking systemcall
   * sys_memap with SYSMEM_INC_OP
```

```
   */
  struct sc_regs regs;
  regs.a1 = SYSMEM_INC_OP;
  regs.a2 = vmaid;
  regs.a3 = size;
  /* SYSCALL 17 sys_memmap */
  syscall(caller, 17, &regs);

  /* TODO: commit the limit increment */
  if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0){
    caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
    caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;

    *alloc_addr = rgnode.rg_start;

    pthread_mutex_unlock(&mmvm_lock);
    return 0;
  }
  /* TODO: commit the allocation address */
  pthread_mutex_unlock(&mmvm_lock);
  return -1;
}
```

Listing 1: Function `__alloc`

The `__alloc` function acts as a middleware interface between user processes and the virtual memory management system, allowing a process to request a memory region within a specified segment (e.g., heap or data). It receives parameters such as the calling process (PCB), virtual memory area ID (`vmaid`), region ID (`rgid`), requested size, and a pointer to store the allocated address. Initially, it validates input parameters—especially ensuring `rgid` is within bounds. A mutex is used to lock the memory resource, preventing race conditions. The system first attempts to find an existing free region; if found, it updates the allocation info in the `symrgtbl` and returns the base address. If no suitable space is available, it invokes the system call `sys_memmap` with `SYSMEM_INC_OP` to expand memory limits. The allocation attempt is then retried. If it still fails, the function returns `-1`. This two-step strategy (search–expand–retry) improves efficiency by avoiding unnecessary memory expansion and ensures proper synchronization and memory reuse, thereby enhancing system performance.

```
int inc_vma_limit(struct pcb_t *caller, int vmaid, int inc_sz)
{
  struct vm_rg_struct * newrg = malloc(sizeof(struct vm_rg_struct));
  int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
  int incnumpage =  inc_amt / PAGING_PAGESZ;
  struct vm_rg_struct *area = get_vm_area_node_at_brk(caller, vmaid,
      inc_sz, inc_amt);
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

  int old_end = cur_vma->vm_end;

  if (area == NULL) return -1;

  /*Validate overlap of obtained region */
  if (validate_overlap_vm_area(caller, vmaid, area->rg_start, area->
      rg_end) < 0){
```

```c
    free(area);
    return -1; /*Overlap and failed allocation */
  }

  /* TODO: Obtain the new vm area based on vmaid */
  // cur_vma->vm_end += inc_amt;
  // cur_vma->sbrk += inc_amt;
  // // inc_limit_ret...

  if (vm_map_ram(caller, area->rg_start, area->rg_end,
                  old_end, incnumpage , newrg) < 0){
    free(area);
    return -1; /* Map the memory to MEMRAM */
  }

  cur_vma->vm_end += inc_amt;
  cur_vma->sbrk += inc_amt;
  enlist_vm_rg_node(&caller->mm->mmap->vm_freerg_list, newrg);
  free(area);
  return 0;
}
```

Listing 2: Function `inc_vma_limit`

The `inc_vma_limit` function is responsible for safely expanding the size limit of a virtual memory area (VMA) within a process, typically used when regions like the heap or data segment run out of allocatable space. It takes the calling process, the target VMA ID, and the requested increment size in bytes. The size is first aligned to the system's page size for compatibility with paging. The function then seeks a free memory region adjacent to the current VMA end to maintain continuity. It verifies that the proposed extension does not overlap with any existing memory areas. If the region is valid, the function maps it into physical RAM. If the mapping fails, the expansion is aborted. Upon success, control structures are updated: `vm_end` is moved to the new boundary, and if applicable, `sbrk` is adjusted. Any surplus memory is added to the free region list for reuse. Overall, `inc_vma_limit` bridges logical allocation needs and physical memory mapping, ensuring expansion is safe, non-overlapping, and well-integrated into memory management.

```c
int get_free_vmrg_area(struct pcb_t *caller, int vmaid, int size, struct
    vm_rg_struct *newrg)
{
  struct vm_area_struct * vma = get_vma_by_num(caller->mm, vmaid);
  if (vma == NULL) return -1;
  for (struct vm_rg_struct ** p_rgit = &vma->vm_freerg_list; *p_rgit;
      p_rgit = &((*p_rgit)->rg_next)) {
    struct vm_rg_struct * rgit = *p_rgit;
    if (rgit->rg_start + size == rgit->rg_end) {
      newrg->rg_start = rgit->rg_start;
      newrg->rg_end = rgit->rg_end;
      *p_rgit = rgit->rg_next;
      free(rgit);
      return 0;
    }
    if (rgit->rg_start + size < rgit->rg_end) {
      newrg->rg_start = rgit->rg_start;
```

```c
      newrg->rg_end = rgit->rg_start += size;
      return 0;
    }
  }
  return -1;
}
```

Listing 3: Function `get_free_vmrg_area`

The `get_free_vmrg_area` function is a key component in managing a process's virtual memory regions (VMRs), responsible for locating a free memory region within a specified virtual memory area (VMA). It receives the requesting process, the VMA ID, the aligned size to allocate, and a reference to `newrg` which will hold the resulting region if successful. First, it retrieves the corresponding `vm_area_struct`. If not found, the function returns `-1`. It then iterates over the `vm_freerg_list`, checking whether any free region is large enough for the requested size. If a region matches exactly, it is removed from the list and returned. If larger, the region is split—`newrg` receives the front portion, and the leftover remains in the free list. If no suitable region exists, the function returns `-1`, signaling the need to expand the VMA using `inc_vma_limit`. This strategy optimizes memory reuse, avoids unnecessary expansions, and enhances performance by leveraging existing free space within the virtual memory area.

```c
int vmap_page_range(struct pcb_t *caller,           // process call
                    int addr,                       // start address
                        which is aligned to pagesz
                    int pgnum,                      // num of mapping
                        page
                    struct framephy_struct *frames, // list of the mapped
                         frames
                    struct vm_rg_struct *ret_rg)    // return mapped
                        region, the real mapped fp
{                                                   // no guarantee all
    given pages are mapped
  // struct framephy_struct *fpit = malloc(sizeof(struct framephy_struct)
      );
  struct framephy_struct *fpit;
  int pgit = 0;
  int pgn = PAGING_PGN(addr);

  /* TODO: update the rg_end and rg_start of ret_rg */
  ret_rg->rg_start = addr;
  ret_rg->rg_end = addr + pgnum * PAGING_PAGESZ;
  // ret_rg->vmaid = ;

  /* TODO map range of frame to address space
   *      [addr to addr + pgnum*PAGING_PAGESZ
   *      in page table caller->mm->pgd[]
   */

  if(frames == NULL)  return -1;
  fpit = frames;
  for (; pgit < pgnum; ++pgit)
  {
    pte_set_fpn(&caller->mm->pgd[pgn + pgit], fpit->fpn);
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
```

```

    fpit = fpit->fp_next;
    free(frames);
    frames = fpit;
  }

  return 0;
}
```

Listing 4: Function `vmap_page_range`

The `vmap_page_range` function plays a crucial role in mapping a contiguous range of virtual memory pages to corresponding physical frames and updating the process's page table. It takes the calling process, the starting address (aligned to page size), the number of pages to map, the list of allocated physical frames, and a pointer `ret_rg` to store the mapped region. The function calculates the end address of the mapping, validates the input, and checks if the frame list is empty. It then iterates over the requested pages, updating the page table entry for each page, mapping the virtual address to the physical frame, and adding the virtual page number to the FIFO queue for future page replacement policies. If all mappings succeed, the function returns 0. This process is integral for managing virtual memory and physical memory mappings, ensuring the system's memory management operates efficiently and correctly.

3.2 Free Implementation

```
int __free(struct pcb_t *caller, int vmaid, int rgid)
{
  struct vm_rg_struct *rgnode;

  // Dummy initialization for avoiding compiler dummy warning
  // In incomplete TODO code rgnode will be overwritten through
      implementing
  // the manipulation of rgid later

  if(rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
    return -1;
  pthread_mutex_lock(&mmvm_lock);

  rgnode = &caller->mm->symrgtbl[rgid];
  if(rgnode->rg_start >= rgnode->rg_end){
    pthread_mutex_unlock(&mmvm_lock);
    return -1;
  }

  /* TODO: Manage the collect freed region to freerg_list */

  /* Enlist the obsoleted memory region */
  enlist_vm_freerg_list(caller->mm, rgnode);
  pthread_mutex_unlock(&mmvm_lock);

  return 0;
}
```

Listing 5: Function `__free`

The `__free` function is responsible for freeing previously allocated memory. It removes a memory region, identified by `rgid`, from the process's memory symbol table and adds it to the free memory list (`freerg_list`) for future reuse. The function checks the validity of `rgid`, and if invalid, returns `-1`. It then locks the memory manager mutex (`mmvm_lock`) to ensure thread safety. After retrieving the corresponding memory region from the symbol table, it checks whether the region's start address is valid. If valid, it adds the region to the free memory list and unlocks the mutex before returning `0`, signaling successful memory deallocation. This function helps optimize memory reuse and ensures safe concurrent access by using mutex locking.

3.3 Read and Write Implementation

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *
    caller)
{
  uint32_t pte = mm->pgd[pgn];

  if (!PAGING_PAGE_PRESENT(pte))
  { /* Page is not online, make it actively living */
    int vicpgn, swpfpn;
    int vicfpn;
    uint32_t vicpte;

    int tgtfpn = PAGING_PTE_SWP(pte); // the target frame storing our
        variable

    /* TODO: Play with your paging theory here */
    /* Find victim page */
    if(find_victim_page(caller->mm, &vicpgn) != 0)  return -1;

    /* Get free frame in MEMSWP */
    vicfpn = PAGING_PTE_FPN(caller->mm->pgd[vicpgn]);
    MEMPHY_get_freefp(caller->active_mswp, &swpfpn);

    /* TODO: Implement swap frame from MEMRAM to MEMSWP and vice versa */
    struct sc_regs regs;
    regs.a1 = SYSMEM_SWP_OP;
    regs.a2 = vicfpn;
    regs.a3 = swpfpn;

    /* SYSCALL 17 sys_memmap */
    syscall(caller, 17, &regs);

    /* Swap target frame from SWAP to MEMRAM */
    __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);

    /* Update page table */
    pte_set_swap(&mm->pgd[vicpgn], caller->active_mswp_id, swpfpn);
    pte_set_fpn(&mm->pgd[pgn], vicfpn);

    enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
  }

```

```
  *fpn = PAGING_FPN(mm->pgd[pgn]);

  return 0;
}
```

Listing 6: Function `pg_getpage`

The `pg_getpage` function is responsible for retrieving a specific page from RAM, identified by the Page Number (`pgn`). If the page is not present in RAM, it performs actions to swap the page from swap memory into RAM. The function first checks the page's status in the page table (`pgd[pgn]`) and if the page is not present, it initiates a page replacement process. A victim page is selected and swapped out to swap memory using syscall 17 (`sys_memmap`) with swap operation. The required page is then swapped into RAM. The page table is updated to reflect these changes, including pointing the victim page to swap memory and the new page to the corresponding physical frame. Finally, the page is added to the FIFO queue of accessed pages and the function returns the physical frame number (FPN) of the requested page.

```
int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct pcb_t *
    caller)
{
  int pgn = PAGING_PGN(addr);
  int off = PAGING_OFFST(addr);
  int fpn;
  pthread_mutex_lock(&mmvm_lock);

  /* Get the page to MEMRAM, swap from MEMSWAP if needed */
  if (pg_getpage(mm, pgn, &fpn, caller) != 0)
    return -1; /* invalid page access */

  int phyaddr = (fpn << PAGING_ADDR_FPN_LOBIT) + off;

  /* SYSCALL 17 sys_memmap with SYSMEM_IO_READ */
  struct sc_regs regs;
  regs.a1 = SYSMEM_IO_READ;
  regs.a2 = phyaddr;

  syscall(caller, 17, &regs);

  /* Update data */
  *data = (BYTE)regs.a3;
  pthread_mutex_unlock(&mmvm_lock);
  return 0;
}
```

Listing 7: Function `pg_getval`

The `pg_getval` function is responsible for reading a value from a virtual address in a process's memory space and returning it. If the page for that address is not present in RAM, the function performs a swap operation to load the page from swap memory into RAM. The function first extracts the page number (PGN) and the offset from the virtual address using `PAGING_PGN(addr)` and `PAGING_OFFST(addr)`. It then locks the memory mutex (`mmvm_lock`) to prevent race conditions. The page is fetched from RAM, and if not found, it is swapped in from swap memory using the `pg_getpage` function. Once the physical page frame number (FPN) is obtained, the physical address is calculated and the value is read from memory using syscall 17 (SYSMEM_IO_READ).

The value read is returned via the `data` pointer. Finally, the mutex is unlocked and the function returns successfully.

```
int pg_setval(struct mm_struct *mm, int addr, BYTE value, struct pcb_t *
    caller)
{
  pthread_mutex_lock(&mmvm_lock);
  int pgn = PAGING_PGN(addr);
  int off = PAGING_OFFST(addr);
  int fpn;

  /* Get the page to MEMRAM, swap from MEMSWAP if needed */
  if (pg_getpage(mm, pgn, &fpn, caller) != 0)
    return -1; /* invalid page access */

  int phyaddr = (fpn << PAGING_ADDR_FPN_LOBIT) + off;

  /* SYSCALL 17 sys_memmap with SYSMEM_IO_WRITE */
  struct sc_regs regs;
  regs.a1 = SYSMEM_IO_WRITE;
  regs.a2 = phyaddr;
  regs.a3 = value;

  syscall(caller, 17, &regs);

  pthread_mutex_unlock(&mmvm_lock);
  return 0;
}
```

Listing 8: Function `pg_setval`

The `pg_setval` function is responsible for writing a value to a specific virtual address in a process's memory. If the page for that address is not present in RAM, the function performs a swap operation to load the page from swap memory into RAM. First, it extracts the page number (PGN) and offset from the virtual address using `PAGING_PGN(addr)` and `PAGING_OFFST(addr)`. It then locks the memory mutex (`mmvm_lock`) to ensure safe access and prevent race conditions. The function uses `pg_getpage` to fetch the physical memory frame (FPN) from the page table. If the page is not found, the function returns an error. Once the physical frame is obtained, the physical address is calculated and the value is written to memory using syscall 17 (SYSMEM_IO_WRITE). After the operation, the mutex is unlocked and the function returns success.

4 System Call `killall` Design

To support bulk process termination by name, we implement a new system call named `killall`. Given the simplicity of our operating system's architecture, this syscall is designed to terminate all processes whose names match a specified target string.
The name of the target process to be terminated is passed via a predefined memory region identified by `REGIONID`, and is retrieved using the utility function `libread`, which has already been implemented in the memory module.
Each process in our system stores its executable name within the `path[]` field of its `pcb_t` structure. Using this field, we compare against the target name to identify matching processes.

To ensure that no matching processes are missed, we scan all relevant queues:

- The `running_list`, which contains processes currently executing.
- Either `mlq_ready_queue[]` (in MLQ mode) or the single `ready_queue` (in non-MLQ mode).

Since the logic for scanning a queue and terminating matching processes is common across these queues, we define a helper function, `kill_in_queue()`, to encapsulate this functionality. It iterates over a given queue, compares each process's name with the target, and invokes `remove_pcb()` for each match in ready queue. If process is running, we only make its pc go to the end of code to notice the CPU that the process is finished and it's time to kill the process. The total number of terminated processes is accumulated and returned by the syscall.

```c
int kill_in_queue(struct queue_t *proc_queue, char *proc_name)
{
    int count = 0, i = 0;

    while (i < proc_queue->size) {
        struct pcb_t *proc = proc_queue->proc[i];
        if (proc == NULL) {
            ++i;
            continue;
        }

        int same_name = check_name(proc_name, proc->path);
        if (same_name == 1) {
            printf("Process %s with PID: %2d is killed\n", proc_name,
                proc->pid);
            if (proc->running_list != NULL) {
                remove_proc(proc->running_list, proc);
                proc->running_list = NULL;
                proc->pc = proc->code->size;
            }
            else {
                remove_pcb(proc);
            }
            ++count;
        }
        else ++i;
    }

    return count;
}
```

An important design decision was to place the `remove_pcb()` function within `sched.c`. This is due to the potential concurrency issues that may arise when removing a process that is currently running or being accessed by other components (e.g., via `get_proc()`).
To mitigate such risks, `remove_pcb()` includes appropriate mutex locking mechanisms to ensure thread-safe access to all queues, including `running_list`, `mlq_ready_queue[]`, and `ready_queue`. Centralizing process removal logic in the scheduler improves synchronization and helps maintain the integrity of the scheduling subsystem.
The steps performed by the `killall` syscall are as follows:

1. Read the target process name using `libread()` from `REGIONID`.
2. Call `kill_in_queue()` for each relevant queue (including `running_list` and the ready queue).
3. Return the total number of processes successfully terminated.

```
int __sys_killall(struct pcb_t *caller, struct sc_regs* regs)
{
    char proc_name[100];
    uint32_t data;
    uint32_t memrg = regs->a1;

    int i = 0;
    data = 0;
    while(data != -1){
        libread(caller, memrg, i, &data);
        proc_name[i]= data;
        if(data == -1) proc_name[i]='\0';
        i++;
    }
    printf("The procname retrieved from memregionid %d is \"%s\"\n",
        memrg, proc_name);

    int killed_count = 0; // Expected output

    struct queue_t *running_list = caller->running_list;
    if (running_list != NULL) {
        killed_count += kill_in_queue(running_list, proc_name);
    }

    #ifdef MLQ_SCHED
        struct queue_t *mlq_ready_queue = caller->mlq_ready_queue;
        if (mlq_ready_queue != NULL) {
            for (int i = 0; i < MAX_PRIO; ++i) {
                struct queue_t *queue = &(mlq_ready_queue[i]);
                if (queue != NULL) {
                    killed_count += kill_in_queue(queue, proc_name);
                }
            }
        }
    #else
        struct queue_t *ready_queue = caller->ready_queue;
        if (ready_queue != NULL) {
            killed_count += kill_in_queue(ready_queue, proc_name);
        }
    #endif

    return killed_count; // Expected output
}
```

This modular approach ensures clarity, code reuse, and synchronization safety within the scheduling system.

5 Testcase

5.1 Scheduler Testcases

This section details the execution and results of the scheduler test cases, demonstrating the Multi-Level Queue (MLQ) scheduling policy implemented in the Simple Operating System.

5.1.1 Testcase: `sched`

Configuration: The simulation is configured using the `input/sched` file:

```
4 2 3
1048576 16777216 0 0 0
0 p1s 1
1 p2s 0
2 p3s 0
```

Listing 9: Content of input/sched

This specifies:

- Time Quantum: 4 time units
- CPUs: 2
- Processes: 3 (details loaded below)
- Memory: RAM 1MB, SWAP 16MB (default active)
- Process `p1s` (PID 1): Load at time 0, Priority 1
- Process `p2s` (PID 2): Load at time 1, Priority 0
- Process `p3s` (PID 3): Load at time 2, Priority 0

Output Log:

```
Time slot   0
ld_routine
  Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
  CPU 1: Dispatched process  1
Time slot   1
  Loaded a process at input/proc/p2s, PID: 2 PRIO: 0
Time slot   2
  CPU 0: Dispatched process  2
  Loaded a process at input/proc/p3s, PID: 3 PRIO: 0
Time slot   3
Time slot   4
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  3
Time slot   5
Time slot   6
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot   7
.........................................
```

```
........................................
Time slot  14
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  1
Time slot  15
  CPU 1: Processed  3 has finished
Time slot  16
  CPU 1 stopped
Time slot  17
Time slot  18
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  19
Time slot  20
  CPU 0: Processed  1 has finished
  CPU 0 stopped
```

Listing 10: Output log: sched test

Execution Analysis: The simulation runs with two CPUs and the MLQ scheduler. Higher priority processes (lower `prio` number) are favored.

- **Time 0-1:** Process 1 (`p1s`, `prio` 1) loads and is dispatched to CPU 1.
- **Time 1-2:** Process 2 (`p2s`, `prio` 0) loads.
- **Time 2-3:** Process 3 (`p3s`, `prio` 0) loads. CPU 0 dispatches Process 2 (higher priority 0).
- **Time 4:** CPU 1 finishes its time slice (4 units) for Process 1. It enqueues Process 1. CPU 1 then dispatches Process 3 (highest available priority 0).
- **Time 6:** CPU 0 finishes its time slice for Process 2. Enqueues Process 2 (`prio` 0) and immediately dispatches it again (Round-Robin among priority 0 processes).
- **Time 8:** CPU 1 finishes its time slice for Process 3. Enqueues Process 3 (`prio` 0) and dispatches it again (next in RR for prio 0).
- **Execution Continues:** CPUs execute processes 2 and 3 (both `prio` 0) using RR until one finishes. Process 2 completes first (Time 14).
- **Time 14:** Process 2 finishes on CPU 0. CPU 0 dispatches Process 1 (only remaining process, `prio` 1).
- **Time 15:** Process 3 finishes on CPU 1. CPU 1 stops.
- **Time 20:** Process 1 finishes on CPU 0. CPU 0 stops.

The scheduler correctly prioritizes the priority 0 processes over priority 1 and applies Round-Robin scheduling within the priority 0 level.
Gantt Chart:

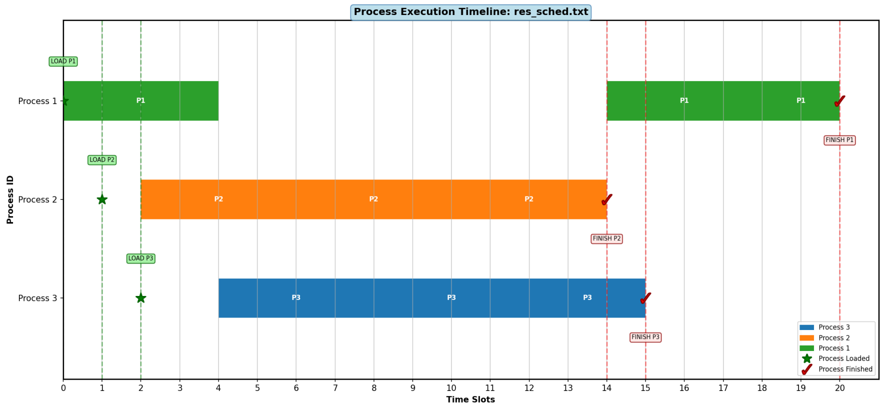


Figure 2: Gantt chart for the `sched` test case.

5.1.2 Testcase: `sched_0`

Configuration:

```
2 1 2
1048576 16777216 0 0 0
0 s0 4
4 s1 0
```

Listing 11: Content of input/sched_0

This specifies:

- Time Quantum: 2 time units
- CPUs: 1
- Processes: 2
- Memory: RAM 1MB, SWAP 16MB
- Process `s0` (PID 1): Load at time 0, Priority 4
- Process `s1` (PID 2): Load at time 4, Priority 0

Output Log:

```
Time slot   0
ld_routine
  Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
  CPU 0: Dispatched process  1
Time slot   2
Time slot   3
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
  Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   4
Time slot   5
```

```
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  2
Time slot   6
........................................
........................................
Time slot  12
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  1
Time slot  13
........................................
........................................
Time slot  22
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  23
  CPU 0: Processed  1 has finished
  CPU 0 stopped
```

Listing 12: Output log: sched_0 test

Execution Analysis: This test demonstrates priority preemption on a single CPU.

- **Time 0-1:** Process 1 (`s0`, `prio` 4) loads and runs.
- **Time 3:** CPU 0 finishes the time slice (2 units) for Process 1. It re-dispatches Process 1.
- **Time 4:** Process 2 (`s1`, `prio` 0) loads.
- **Time 5:** CPU 0 finishes the time slice for Process 1. It enqueues Process 1. The scheduler selects Process 2 (higher priority 0) for dispatch, preempting Process 1.
- **Time 5-11:** Process 2 (`prio` 0) runs uninterrupted.
- **Time 12:** Process 2 finishes. CPU 0 dispatches the waiting Process 1 (`prio` 4).
- **Time 12-23:** Process 1 runs until completion.

The arrival of the higher-priority Process 2 correctly preempts the lower-priority Process 1.
Gantt Chart:

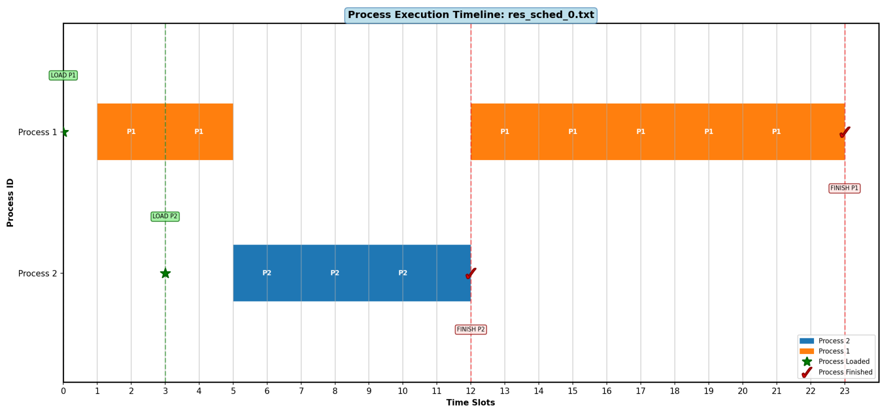


Figure 3: Gantt chart for the `sched_0` test case.

5.1.3 Testcase: sched_1

Configuration:

```
2 1 4
1048576 16777216 0 0 0
0 s0 4
4 s1 0
6 s2 0
7 s3 0
```

Listing 13: Content of input/sched_1

This specifies:

- Time Quantum: 2 time units
- CPUs: 1
- Processes: 4
- Memory: RAM 1MB, SWAP 16MB
- Process s0 (PID 1): Load at time 0, Priority 4
- Process s1 (PID 2): Load at time 4, Priority 0
- Process s2 (PID 3): Load at time 6, Priority 0
- Process s3 (PID 4): Load at time 7, Priority 0

Output Log:

```
Time slot   0
ld_routine
  Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
  CPU 0: Dispatched process  1
Time slot   2
Time slot   3
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
  Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   4
Time slot   5
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  2
  Loaded a process at input/proc/s2, PID: 3 PRIO: 0
Time slot   6
Time slot   7
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
  Loaded a process at input/proc/s3, PID: 4 PRIO: 0
Time slot   8
Time slot   9
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
```

```
Time slot  10
Time slot  11
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  4
Time slot  12
Time slot  13
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  3
Time slot  14
.........................................
.........................................
Time slot  34
  CPU 0: Processed  3 has finished
  CPU 0: Dispatched process  4
Time slot  35
  CPU 0: Processed  4 has finished
  CPU 0: Dispatched process  1
Time slot 36
.........................................
.........................................
Time slot  45
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  46
  CPU 0: Processed  1 has finished
  CPU 0 stopped
```

Listing 14: Output log: sched_1 test

Execution Analysis: This test demonstrates Round-Robin scheduling among multiple high-priority processes with staggered arrivals.

- **Time 0-4:** Process 1 (`s0`, `prio` 4) runs.
- **Time 4:** Process 2 (`s1`, `prio` 0) loads.
- **Time 5:** Process 1 is preempted; Process 2 (`prio` 0) runs.
- **Time 6:** Process 3 (`s2`, `prio` 0) loads.
- **Time 7:** Process 2's slice ends → enqueued. Process 3 (`prio` 0) runs. Process 4 (`s3`, `prio` 0) loads.
- **Time 9:** Process 3's slice ends → enqueued. Process 2 runs (first ready `prio` 0).
- **Time 11:** Process 2's slice ends → enqueued. Process 4 runs.
- **Time 13:** Process 4's slice ends → enqueued. Process 3 runs.
- **Execution Continues:** Processes 2, 3, and 4 (`prio` 0) execute in Round-Robin order (effectively 2 → 4 → 3 → 2 → ...) until completion. Process 2 finishes first (Time 22), then Process 3 (Time 34), then Process 4 (Time 35).
- **Time 35:** All priority 0 processes are finished. CPU 0 dispatches the waiting Process 1 (`prio` 4).

- **Time 35-46:** Process 1 runs until completion.

This correctly shows preemption and RR scheduling within the highest priority level.
Gantt Chart:

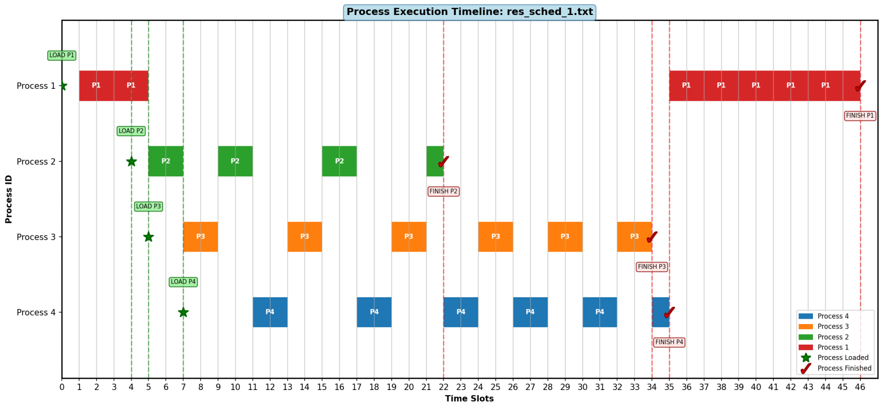


Figure 4: Gantt chart for the `sched_1` test case.

5.2 Memory Management

5.2.1 Testcase 1: Basic Process Loading and Page Table Management

Input Files

- `mem_test_1`

```
10 1 1
1048576 0 0 0 0
0 mem_proc_1 1
```

- `mem_proc_1`

```
1 3
alloc 300 0
alloc 200 1
alloc 200 2
```

Output Content

```
Time slot   0
ld_routine
        Loaded a process at input/proc/mem_proc_1, PID: 1 PRIO: 1
        CPU 0: Dispatched process  1
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=0 - Address=0 - Size=300 byte
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Page Number: 0 -> Frame Number: 1
```

```
Page Number: 1 -> Frame Number: 0
================================================================
Time slot   1
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=300 - Size=200 byte
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
================================================================
Time slot   2
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=2 - Address=512 - Size=200 byte
print_pgtbl: 0 - 768
00000000: 80000001
00000004: 80000000
00000008: 80000002
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 0
Page Number: 2 -> Frame Number: 2
================================================================
Time slot   3
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Explanation

This testcase demonstrates the loading and memory allocation process for a single user process using paging-based virtual memory management.
The file `mem_proc_1` contains information for one process:

- The first line indicates there is only 1 process.
- The second line shows its priority is 1.
- The next three lines correspond to the sizes of its three memory regions: 300, 200, and 200 bytes.

In time slot 0, the loader routine starts by creating the process (PID = 1). It assigns 300 bytes for Region 0 starting from address 0. Since each page is 256 bytes, Region 0 spans across two virtual pages. The page table maps:

- Page 0 to Frame 1
- Page 1 to Frame 0

In time slot 1, Region 1 (200 bytes) is allocated from address 300. This region is still within the bounds of the first two pages, so no new frame is needed, and the page table remains unchanged.
In time slot 2, Region 2 (200 bytes) begins at address 512, which lies in a new page. Therefore, a new mapping is created:

- Page 2 to Frame 2

The page table now covers addresses up to 768 bytes, reflecting the additional mapping.
By time slot 3, the process finishes execution, and CPU 0 becomes idle.
This testcase is useful for verifying:

- The loader correctly splits memory across pages.
- Proper frame allocation based on virtual address range.
- The incremental update of the page table as more regions are allocated.
- Clean termination and memory release behavior (implied after final time slot).

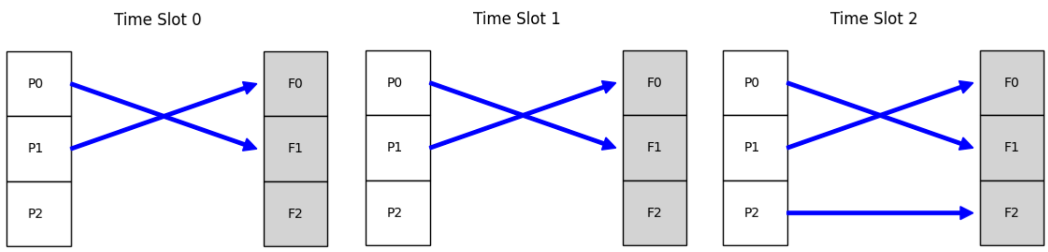


Figure 5: Virtual memory mapping across time slots.

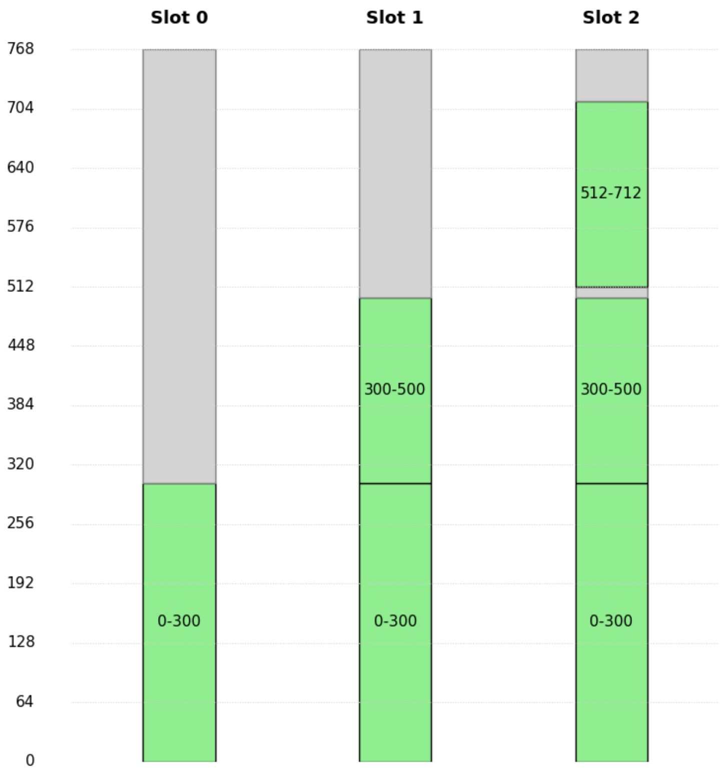


Figure 6: Physical memory state over time.

5.2.2 Testcase 2: Dynamic Allocation and Deallocation with Page Table Reuse

Input Files

- mem_test_2

```
10 1 1
1048576 0 0 0 0
0 mem_proc_2 1
```

- mem_proc_2

```
1 6
alloc 100 0
alloc 100 1
free 0
alloc 50 0
alloc 25 2
alloc 50 3
```

Output Content

```
Time slot   0
ld_routine
        Loaded a process at input/proc/mem_proc_2, PID: 1 PRIO: 1
        CPU 0: Dispatched process  1
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=0 - Address=0 - Size=100 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
Time slot   1
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=100 - Size=100 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
Time slot   2
===== PHYSICAL MEMORY AFTER DEALLOCATION =====
PID=1 - Region=0
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
Time slot   3
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=0 - Address=0 - Size=50 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
Time slot   4
```

```
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=2 - Address=50 - Size=25 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
Time slot   5
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=3 - Address=200 - Size=50 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
Time slot   6
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Explanation
This testcase demonstrates the dynamic behavior of memory allocation and deallocation using paging. A process is loaded, memory is allocated to different regions, and one region is explicitly deallocated and reused.
The file `mem_proc_2` describes a process that performs six memory-related operations:

- It allocates 100 bytes in region 0.
- It then allocates another 100 bytes in region 1.
- Region 0 is freed, releasing the associated memory.
- It reallocates 50 bytes in region 0 (smaller size).
- It allocates 25 bytes in region 2.
- It allocates 50 bytes in region 3.

In time slot 0, the process is loaded, and 100 bytes are allocated for region 0 starting from address 0. Since one page is 256 bytes, all data fits within one page, and a single page-to-frame mapping is made.
Time slot 1 continues with another 100-byte allocation at address 100 for region 1. The data still fits within the first page, so no additional frame is needed. The mapping remains unchanged.
Time slot 2 frees region 0. Although the virtual address space is unaffected, the associated memory block becomes available for reuse.
In time slot 3, the program reuses the freed space to allocate a smaller region of 50 bytes at address 0. The system reuses the same frame (frame 0), showing efficiency in page management.
Time slot 4 allocates a 25-byte region at address 50, still within the same page and frame.
Finally, in time slot 5, 50 bytes are allocated at address 200 for region 3, which again fits within the same page. The entire process's memory footprint lies within the 256-byte range of one page, thus requiring only one frame.
By time slot 6, the process finishes execution, and the CPU stops.
This testcase is useful for verifying:

- Memory deallocation and reallocation on the same page.
- Efficient reuse of freed memory frames.

- Handling of multiple small allocations within a single page.
- Correctness of page table consistency across changes.

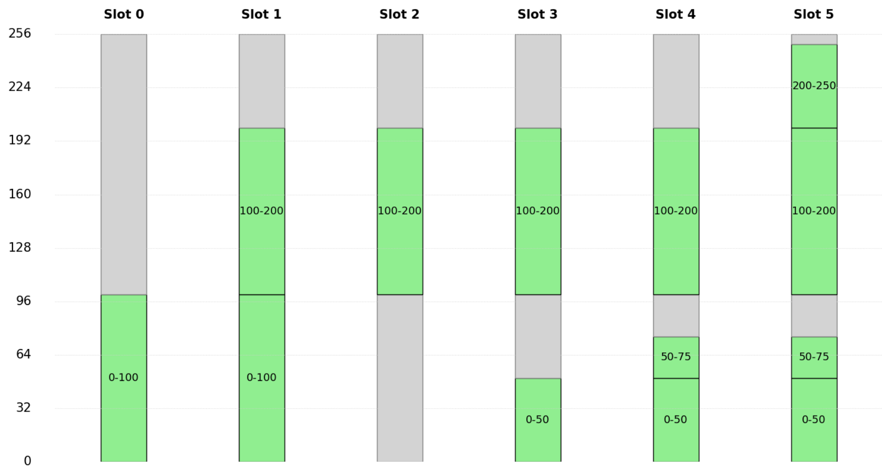


Figure 7: Physical memory state over time.

5.2.3 Testcase 3: Memory Access with Paging - Read/Write Across Pages

Input Files

- mem_test_3

```
10 1 1
1048576 0 0 0 0
0 mem_proc_3 1
```

- mem_proc_3

```
1 8
alloc 200 0
alloc 500 1
write 12 1 200
read 1 200 10
write 34 1 400
read 1 400 10
write 56 1 400
read 1 400 10
```

Output Content

```
Time slot   0
ld_routine
        Loaded a process at input/proc/mem_proc_3, PID: 1 PRIO: 1
Time slot   1
        CPU 0: Dispatched process  1
```

```
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=0 - Address=0 - Size=200 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
Time slot   2
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=256 - Size=500 byte
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000002
00000008: 80000001
Page Number: 0 -> Frame Number: 0
Page Number: 1 -> Frame Number: 2
Page Number: 2 -> Frame Number: 1
================================================================
Time slot   3
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=200 value=12
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000002
00000008: 80000001
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 000002c8: 12
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   4
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=200 value=12
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000002
00000008: 80000001
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 000002c8: 12
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   5
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=400 value=34
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000002
00000008: 80000001
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000190: 34
BYTE 000002c8: 12
===== PHYSICAL MEMORY END-DUMP =====
```

```
================================================================
Time slot   6
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=400 value=34
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000002
00000008: 80000001
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000190: 34
BYTE 000002c8: 12
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   7
===== PHYSICAL MEMORY AFTER WRITING =====
write region=1 offset=400 value=56
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000002
00000008: 80000001
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000190: 56
BYTE 000002c8: 12
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   8
===== PHYSICAL MEMORY AFTER READING =====
read region=1 offset=400 value=56
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000002
00000008: 80000001
================================================================
===== PHYSICAL MEMORY DUMP =====
BYTE 00000190: 56
BYTE 000002c8: 12
===== PHYSICAL MEMORY END-DUMP =====
================================================================
Time slot   9
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Explanation

This testcase verifies the correctness of virtual memory management using paging. It demonstrates:

- Proper page-to-frame mappings during multiple region allocations.
- Accurate translation of virtual addresses to physical addresses during memory operations.
- Consistency of page table entries through successive memory accesses.
- Correctness of read/write operations with updated data values.

The memory operations in `mem_proc_3` involve:

- **Region 0** is allocated `200 bytes`, starting at virtual address `0`. It fits entirely within page 0.
- **Region 1** is allocated `500 bytes`, starting at virtual address `256`. This region spans across **three pages**, because `256 + 500 = 756`, which exceeds one and two page boundaries (each page is 256 bytes).

The virtual-to-physical mapping after allocations is:

- Page 0 → Frame 0
- Page 1 → Frame 2
- Page 2 → Frame 1

Specific memory operations:

- A **write** operation at `offset 200` in region 1.
  - Virtual address = `base address (256) + offset (200)` = `456`.
  - Virtual address 456 belongs to page 1 (`page number = 456 / 256 = 1`).
  - Page 1 is mapped to frame 2.
  - Physical address = `frame 2 * 256 + (456 mod 256)` = `2*256 + 200` = `712` (000002C8).
- A **read** operation at the same offset confirms the value `12` is correctly retrieved from physical address 712.
- A **write** at `offset 400` in region 1.
  - Virtual address = `256 + 400` = `656`.
  - Virtual address 656 belongs to page 2 (`page number = 656 / 256 = 2`).
  - Page 2 is mapped to frame 1.
  - Physical address = `frame 1 * 256 + (656 mod 256)` = `1*256 + 144` = `400` (00000190).
- A **read** at the same offset confirms the written value (`34` initially, later updated to `56`) is correctly stored and retrieved from physical address 400.

The page table remains consistent throughout all time slots, and memory dumps validate that the data is properly written, updated, and read according to the virtual-to-physical mappings.

5.2.4 Testcase 4: Multi-Process Memory Allocation and Paging Consistency

Input Files

- `mem_test_4`

```
10 3 3
1048576 0 0 0 0
0 mem_proc_4_1 1
1 mem_proc_4_2 1
2 mem_proc_4_3 1
```

- mem_proc_4_1

```
1 6
alloc 200 0
calc
calc
alloc 200 1
calc
calc
```

- mem_proc_4_2

```
1 4
alloc 200 0
calc
calc
alloc 200 1
```

- mem_proc_4_3

```
1 5
alloc 200 0
calc
calc
alloc 200 1
calc
```

Output Content

```
Time slot   0
ld_routine
        Loaded a process at input/proc/mem_proc_4_1, PID: 1 PRIO: 1
        CPU 2: Dispatched process  1
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=0 - Address=0 - Size=200 byte
print_pgtbl: 0 - 256
00000000: 80000000
Page Number: 0 -> Frame Number: 0
================================================================
Time slot   1
        Loaded a process at input/proc/mem_proc_4_2, PID: 2 PRIO: 1
        CPU 1: Dispatched process  2
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=2 - Region=0 - Address=0 - Size=200 byte
print_pgtbl: 0 - 256
00000000: 80000001
Page Number: 0 -> Frame Number: 1
================================================================
Time slot   2
        Loaded a process at input/proc/mem_proc_4_3, PID: 3 PRIO: 1
Time slot   3
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=1 - Region=1 - Address=256 - Size=200 byte
```

```
print_pgtbl: 0 - 512
00000000: 80000000
00000004: 80000002
Page Number: 0 -> Frame Number: 0
Page Number: 1 -> Frame Number: 2
================================================================
        CPU 0: Dispatched process  3
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=0 - Address=0 - Size=200 byte
print_pgtbl: 0 - 256
00000000: 80000003
Page Number: 0 -> Frame Number: 3
================================================================
Time slot   4
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=2 - Region=1 - Address=256 - Size=200 byte
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000004
Page Number: 0 -> Frame Number: 1
Page Number: 1 -> Frame Number: 4
================================================================
Time slot   5
Time slot   6
        CPU 2: Processed  1 has finished
        CPU 2 stopped
===== PHYSICAL MEMORY AFTER ALLOCATION =====
PID=3 - Region=1 - Address=256 - Size=200 byte
        CPU 1: Processed  2 has finished
        CPU 1 stopped
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000005
Page Number: 0 -> Frame Number: 3
Page Number: 1 -> Frame Number: 5
================================================================
Time slot   7
        CPU 0: Processed  3 has finished
        CPU 0 stopped
```

Explanation

This testcase demonstrates consistent memory allocation and virtual-to-physical mapping in a multi-core and multi-process paging system. It ensures the following:

- Each process independently allocates memory in two separate regions.
- The virtual pages are correctly mapped to distinct physical frames.
- Frame reuse is avoided across processes to preserve isolation and correctness.
- Page tables reflect accurate mappings for each PID, confirming region-wise paging.

Detailed process activity includes:

- Process 1 (PID=1) is dispatched on CPU 2 and allocates 200 bytes in Region 0, then another 200 bytes in Region 1 (total 2 frames used).
- Process 2 (PID=2) runs on CPU 1 and similarly allocates 200 bytes in Region 0, then another 200 bytes in Region 1 (2 frames).
- Process 3 (PID=3) starts on CPU 0 slightly later and performs identical allocations, with unique frame mappings.

By Time slot 7, all three processes have terminated, and the memory state shows six separate frame allocations corresponding to three processes, each with two memory regions. The output validates that the memory manager correctly maintains separation and mapping for all processes in a concurrent environment.

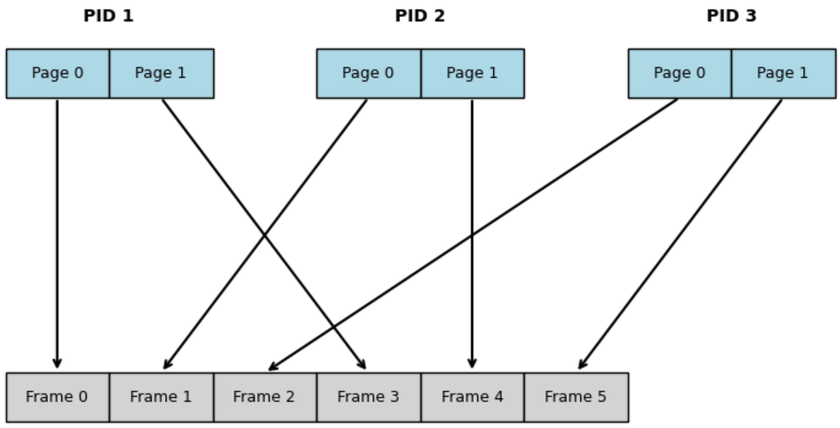


Figure 8: Virtual memory mapping at time slot 6.

5.3 System Call Testcases

This section verifies the implementation and interaction of system calls within the Simple Operating System, considering the build process defined in the `Makefile` and the interface function in `libstd.c`.

5.3.1 Testcase: `os_syscall_list`

Configuration:

```
2 1 1
2048 16777216 0 0 0
9 sc1  15
```

Listing 15: Content of input/os_syscall_list

```
20 1
syscall 0
```

Listing 16: Content of input/proc/sc1

System Call Tested: Syscall number 0 (`sys_listsyscall`)
Output Log:

```
1 Time slot   0
2 ld_routine
3 Time slot   1
4 Time slot   2
5 Time slot   3
6 Time slot   4
7 Time slot   5
8 Time slot   6
9 Time slot   7
10 Time slot   8
11 Time slot   9
12         Loaded a process at input/proc/sc1, PID: 1 PRIO: 15
13 Time slot  10
14         CPU 0: Dispatched process  1
15 0-sys_listsyscall
16 17-sys_memmap
17 101-sys_killall
18 440-sys_xxxhandler
19 Time slot  11
20         CPU 0: Processed  1 has finished
21         CPU 0 stopped-
```

Listing 17: Output log: os_syscall_list test

Execution Analysis:

- Process `sc1` (PID 1) is loaded and dispatched.
- It executes `syscall 0`. The CPU calls `libsyscall(proc, 0, 0, 0, 0)`.
- `libsyscall` creates the `sc_regs` struct and calls the kernel function `syscall(proc, 0, sc_regs)`.
- The kernel's `syscall` dispatcher finds `case 0:` and invokes `__sys_listsyscall(caller, sv_regs)`.
- The `__sys_listsyscall` handler executes, printing the registered system calls (lines 12-14).
- The handler returns 0, and the process finishes.

This confirms syscall 0 is correctly mapped and functional. Interaction Flow: User Process → CPU → `libsyscall` → Kernel `syscall` Dispatcher → `__sys_listsyscall` Handler → Standard Output.

5.3.2 Testcase: `os_syscall`

Configuration:

```
2 1 1
2048 16777216 0 0 0
9 sc2  15
```

Listing 18: Content of input/os_syscall

```
20 5
alloc 100 1
write 80 1 0
write 48 1 1
write -1 1 2
syscall 101 1
```

Listing 19: Content of input/proc/sc2

System Call Tested: Syscall number 101 (`sys_killall`)
Output Log:

```
Time slot   0
ld_routine
Time slot   1
Time slot   2
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
Time slot   9
        Loaded a process at input/proc/sc2, PID: 1 PRIO: 15
Time slot  10
        CPU 0: Dispatched process  1
Time slot  11
Time slot  12
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  13
Time slot  14
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
The procname retrieved from memregionid 1 is "P0"
Time slot  15
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Listing 20: Output log: os_syscall test

Execution Analysis: This analysis details the execution flow for the `os_syscall` test case, considering a time slice of 2, 1 CPU, and 1 process.

- **Time 0-8:** The system initializes.
- **Time 9:** The loader loads process `sc2` (PID 1, Priority 15).
- **Time 10:** CPU 0 dispatches PID 1. The process begins executing its code segment. The first instruction, `alloc 100 1`, is executed, allocating memory region 1.
- **Time 11:** PID 1 continues execution on CPU 0. The second instruction, `write 80 1 0` (writing 'P'), executes. Since the time slice is 2, the process yields at the end of this time slot.

- **Time 12:** CPU 0 puts PID 1 back into the ready queue. As it's the only process, it's immediately re-dispatched. PID 1 executes its next instruction, `write 48 1 1` (writing '0').
- **Time 13:** PID 1 continues on CPU 0 and executes `write -1 1 2` (writing the null terminator), completing the string "P0" in memory region 1. The process yields again at the end of the time slot.
- **Time 14:** CPU 0 puts PID 1 back and re-dispatches it. PID 1 executes its final instruction: `syscall 101 1`.
  - The CPU triggers the system call mechanism via `libsyscall(proc, 101, 1, 0, 0)`.
  - `libsyscall` sets `regs.a1 = 1` and invokes the kernel `syscall` function.
  - The kernel dispatcher identifies syscall 101 and calls `__sys_killall(caller, sv_regs)`.
  - The `__sys_killall` handler interprets $regs \to a1$ (value 1) as the memory region ID containing the target program name.
  - The handler uses `libread` to fetch the string "P0" from region 1 of PID 1's memory.
  - It prints the confirmation: "The procname retrieved from memregionid 1 is "P0"".
  - The handler search for processes named "P0" to terminate. In this specific test case, there are no other processes running, so the search yields no targets.
  - The handler returns 0, indicating completion.
- The `syscall 101 1` instruction finishes execution within time slot 14. The process's program counter (`pc`) advances past this final instruction.
- **Time 15:** At the start of the time slot, the OS checks PID 1. The condition $proc->pc == proc->code->size$ is now true. The OS marks the process as finished and prints "Processed 1 has finished". Since there are no other ready processes, CPU 0 stops.

This test successfully demonstrates the `killall` syscall invocation. It confirms that the REGIONID (`1`) is correctly passed via `regs->a1` and used by the kernel to retrieve the target program name ("P0") from the user process's memory. Although no other processes were present to be killed in this specific scenario, the retrieval mechanism is verified. Interaction Flow: User Process (`alloc`, `write`) → Memory Management → CPU → `libsyscall` → Kernel `syscall` Dispatcher → `__sys_killall` Handler → Kernel Memory Read (`libread`) → Standard Output → (Process List Scan - no matches) → Return 0 → Process Completion.

Gantt Chart

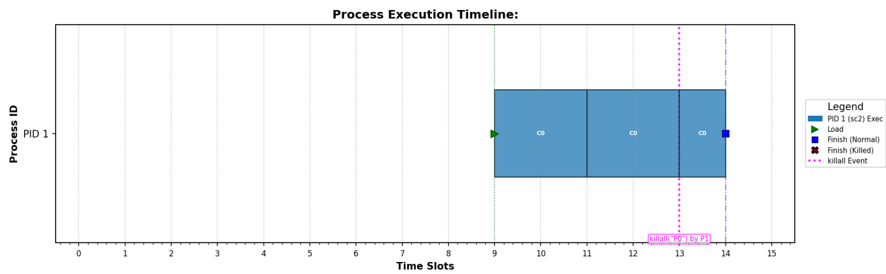


Figure 9: Gantt chart for the os_syscall testcase

5.3.3 Testcase: `os_sc`

Configuration:

```
2 1 1
2048 16777216 0 0 0
9 sc3  15
```

Listing 21: Content of input/os_sc

```
20 1
syscall 440 1
```

Listing 22: Content of input/proc/sc3

System Call Tested: Syscall number 440 (Implemented as `sys_xxxhandler`)
Output Log:

```
1  Time slot   0
2  ld_routine
3  Time slot   1
4  Time slot   2
5  Time slot   3
6  Time slot   4
7  Time slot   5
8  Time slot   6
9  Time slot   7
10 Time slot   8
11   Loaded a process at input/proc/sc3, PID: 1 PRIO: 15
12 Time slot   9
13   CPU 0: Dispatched process  1
14 The first system call paramter 1
15 Time slot  10
16   CPU 0: Processed  1 has finished
17   CPU 0 stopped
```

Listing 23: Output log: os_sc test

Execution Analysis

- Process `sc3` (PID 1) is loaded at time 8 and dispatched by CPU 0 at time 9.
- The process executes its only instruction: `syscall 440 1`.
- The CPU invokes `libsyscall(proc, 440, 1, 0, 0)`.
- `libsyscall` (in `libstd.c`) populates the `struct sc_regs` with `regs.a1 = 1` and calls the kernel function `syscall(proc, 440, s)`.
- The kernel's `syscall` function in `syscall.c` evaluates its `switch(nr)` statement. Based on the modified `src/syscall.tbl` (which now includes `440 xxx xxxhandler`) and the updated `Makefile` (linking `sys_xxxhandler.o`), the dispatcher finds `case 440:` and invokes the handler function `__sys_xxxhandler(caller, s)`.
- The `__sys_xxxhandler` function (defined in `sys_xxxhandler.c`) executes. It accesses the argument passed from user space via `regs->a1` (which holds the value 1) and prints the message: "The first system call paramter 1" (as seen on line 13 of the output log).

- The handler function returns 0, indicating successful execution to the kernel.
- This return value propagates back to the CPU execution loop. The `syscall 440 1` instruction is considered complete.
- The process's program counter (`pc`) advances past this only instruction.
- At the beginning of the next time slot (Time 10), the CPU detects that the process has completed (`proc->pc == proc->code->size`) and prints the "Processed 1 has finished" message, subsequently stopping.

This test case successfully demonstrates the implementation and invocation of a custom system call (number 440). The output confirms that the system call was correctly registered in `syscall.tbl`, compiled and linked via the `Makefile`, dispatched by the kernel, and executed the handler code (`__sys_xxxhandler`), including correctly receiving and using the argument passed from user space via `regs->a1`. Interaction Flow: User Process → CPU → `libsyscall` → Kernel `syscall` Dispatcher → `__sys_xxxhandler` → Standard Output → Return 0 → Process Completion.

5.3.4 Testcase: `os_killall_complex`

Configuration

```
2 2 8
2048 16777216 0 0 0
0 s0 5
3 s1 10
5 s1 15
8 s1 20
12 s2 8
15 s3 12
20 killer 1
25 s1 5
```

Listing 24: Content of input/os_killall_complex

This specifies:

- Time Quantum: 2 time units
- CPUs: 2
- Processes: 8
- Memory: Standard Configuration
- Processes (Load Time, Name, Priority): (0, s0, 5), (3, s1, 10), (5, s1, 15), (8, s1, 20), (12, s2, 8), (15, s3, 12), (20, killer, 1), (25, s1, 5)

```
20 5
alloc 100 1
write 115 1 0 (ASCII 's' )
write 49 1 1  (ASCII '1')
write -1 1 2   (Null terminator)
syscall 101 1
```

Listing 25: Content of input/proc/killer

System Call Tested: Syscall number 101 (`sys_killall`), invoked by the `killer` process to target processes named "s1".

Output Log

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 5
        CPU 1: Dispatched process  1
Time slot   1
Time slot   2
        Loaded a process at input/proc/s1, PID: 2 PRIO: 10
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot   3
Time slot   4
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/s1, PID: 3 PRIO: 15
Time slot   5
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
..........................................
..........................................
Time slot  19
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  5
        Loaded a process at input/proc/killer, PID: 7 PRIO: 1
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  7
Time slot  20
Time slot  21
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  5
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  22
Time slot  23
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  5
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
The procname retrieved from memregionid 1 is "s1"
Time slot  24
        Loaded a process at input/proc/s1, PID: 8 PRIO: 5
        CPU 0: Processed  5 has finished
        CPU 0: Dispatched process  6
        CPU 1: Processed  7 has finished
        CPU 1: Dispatched process  8
Time slot  25
Time slot  26
        CPU 1: Put process  8 to run queue
        CPU 1: Dispatched process  8
Time slot  27
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
```

```
Time slot  28
        CPU 1: Put process  8 to run queue
        CPU 1: Dispatched process  8
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
Time slot  29
Time slot  30
        CPU 1: Put process  8 to run queue
        CPU 1: Dispatched process  8
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
Time slot  31
        CPU 1: Processed  8 has finished
        CPU 1 stopped
        CPU 0: Processed  6 has finished
        CPU 0 stopped
Time slot  32
```

Listing 26: Output log: os_killall_complex test

Execution Analysis

This test case simulates a more complex scenario with multiple processes of varying priorities running concurrently, culminating in the execution of a dedicated `killer` process (PID 7) that invokes the `killall` system call.

- **Time 0-19:** Various processes (PIDs 1-6) with priorities 5, 10, 15, 20, 8, 12 are loaded and scheduled across the 2 CPUs according to the MLQ policy (higher priority first, RR within priorities). Process 2 (Prio 10) finishes at Time 10. Process 1 (Prio 5) finishes at Time 15. Processes 3 (Prio 15), 4 (Prio 20) are loaded but likely spend time waiting due to higher priority processes 5 (Prio 8) and 6 (Prio 12).
- **Time 20:** The `killer` process (PID 7, Prio 1 - highest) is loaded.
- **Time 19 (Dispatch):** Although loaded at T=20, due to scheduling checks, CPU 1 (which was running PID 6, Prio 12) puts PID 6 back and immediately dispatches the new highest priority process, PID 7 (`killer`), at the end of time slot 19.
- **Time 20-23:** PID 7 (`killer`) executes on CPU 1. It performs `alloc 100 1`, then writes 's', '1', '\0' into region 1. Concurrently, PID 5 (Prio 8) runs on CPU 0.
- **Time 23 (Syscall):** PID 7 executes `syscall 101 1` during this time slot.
- The `__sys_killall` handler is invoked. It retrieves the region ID 1 from $regs- > a1$.
- The handler reads the target program name "s1" from region 1 of PID 7's memory. This is confirmed by the output "The procname retrieved from memregionid 1 is "s1"" printed between time slots 23 and 24.
- **Kill Logic Execution:** Based on the instructor's clarification, the handler now iterates through the existing processes in the system's queues (`running_list`, `mlq_ready_queue`).
  - It checks PID 3 (process created from `input/proc/s1`). The base name "s1" matches the target. PID 3 is terminated and removed.
  - It checks PID 4 (process created from `input/proc/s1`). The base name "s1" matches the target. PID 4 is terminated and removed.

  - Other processes (PID 1, 5, 6, 7) have different base names ("s0", "s2", "s3", "killer") and are skipped.

- The `killall` syscall completes.
- **Time 24 (Post-Syscall):**
- A new process (PID 8, from `input/proc/s1`, Prio 5) is loaded.
- CPU 0 finishes executing PID 5 (`s2`). Since PIDs 3 and 4 (`s1`) were killed, the next available process is PID 6 (`s3`). CPU 0 dispatches PID 6.
- CPU 1 finishes executing PID 7 (`killer`). Since PIDs 3 and 4 were killed, the next highest priority process available is the newly loaded PID 8 (`s1`). CPU 1 dispatches PID 8.
- **Time 25-31:** The remaining processes, PID 6 (`s3`) and PID 8 (`s1`), run concurrently on CPUs 0 and 1 respectively until they finish.
- **Time 31:** Both PID 8 and PID 6 finish, and both CPUs stop.

This complex test demonstrates the MLQ scheduler handling multiple processes and priorities. Crucially, it shows the `killall` system call (PID 7) correctly identifying and targeting processes based on the program name ("s1") retrieved from the specified memory region ID. The absence of PIDs 3 and 4 in the scheduling decisions after Time 23 confirms they were successfully terminated by the system call. Interaction Flow: Multi-Process Scheduling → Killer Process Execution (`alloc`, `write`) → Syscall Interface (`syscall 101 1`) → Kernel `__sys_killall` Handler → Kernel Memory Read (`libread`) → Process List Traversal & Comparison → Process Termination (Implicit) → Resumption of Scheduling for remaining processes.

Gantt Chart

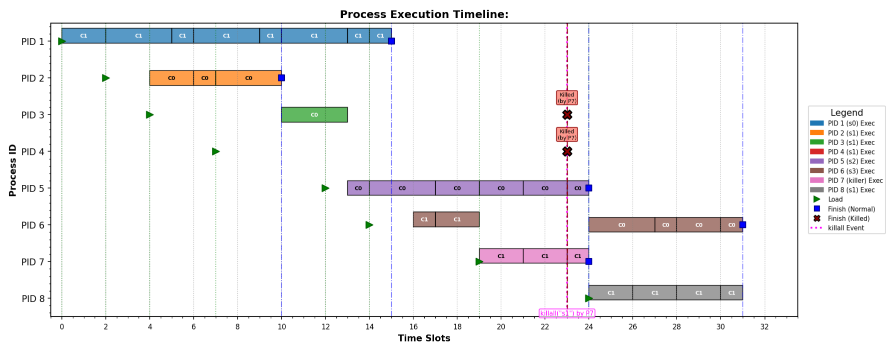


Figure 10: Gantt chart for the `os_killall_complex` test case.

6 Answering questions

Question1: What is the mechanism to pass a complex argument to a system call using the limited registers?

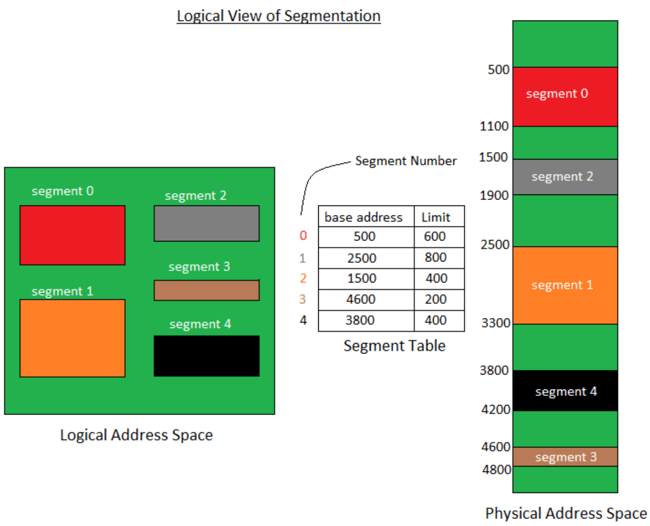
System calls bridge user processes and the kernel. While simple arguments fit directly into registers (like `a1`, `a2`, `a3` in `struct sc_regs`), complex data like strings requires an indirect mechanism due to the limited number of registers.

1. Common OS Approach: Passing Virtual Address Pointers
   Most general-purpose operating systems (e.g., Linux, Windows) handle complex arguments by passing **virtual memory pointers**.
   - **User Setup:** The process allocates memory in its virtual address space and stores the complex data (e.g., a string for a filename, a buffer for reading data) there.
   - **Syscall Invocation:** The process places the starting **virtual address** of this data buffer into one of the argument registers when invoking the system call.
   - **Kernel Operation:** The kernel retrieves the virtual address from the register. It performs essential **validation** (checking if the address range is valid and accessible for the process) and **translation** (using page tables to find the physical address) before accessing the user's data buffer. This is typical for calls like `read()` or `write()`.
2. Assignment's `killall` Approach: Passing Region ID for Program Name
   The `killall` system call (syscall 101) in this assignment uses a specific variation tailored to its function: it passes a **Memory Region Identifier** (`REGIONID`) that references the location of the **target program name string**.
   - **User Setup:** A user process (like `sc2`) uses instructions like `alloc 100 1` to associate memory region ID `1` with a buffer, and then uses `write` to store the target program name (e.g., "P0") into that region.
   - **Syscall Invocation:** The process calls `syscall 101 1`, passing the `REGIONID` (`1`) as the argument.
   - **Kernel Argument Handling:** `libsyscall` packages this ID into `regs.a1 = 1`.
   - **Kernel Name Retrieval:** The `__sys_killall` handler receives the ID via `regs->a1`. It knows this ID refers to a region containing a program name. It uses the ID (`memrg = regs->a1`) with kernel memory functions (`libread(caller, memrg, ...)`) to read the actual program name string (e.g., "P0") from the calling process's specified memory region.
   - **Purpose & Subsequent Action:** The purpose of retrieving this target name ("P0") is for the kernel handler to then iterate through the system's process lists (like `running_list` and `mlq_ready_queue`). For each process found, the kernel extracts the base program name from its `proc->path` field and compares it to the retrieved target name ("P0"). If the names match, the kernel terminates that process (as described by the instructor's clarification, although the termination code itself might be simplified or partially implemented in the provided skeleton). If the names don't match, the kernel bypasses that process and continues the search.

   This method uses the system call argument as a specific identifier for the program name's location, leveraging the simulation's memory region management (`libread` handling the lookup based on ID) rather than passing a raw, general-purpose pointer. This allows `killall` to target processes based on the name stored in a designated region.

Question 2: In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments?

Segmentation is the mechanism where a process is divided into variable-sized segments according to their purposes , such as text, data, stack, heap, etc. These segments are managed and mapped to the physical memory though Segment Table.



This design provides security and protection among segments. When a segment is attached, it is isolated to other, avoiding unauthorized access from other segments. Additionally, segmentation enhances Inter-process communication, allowing processes sharing segments without worrying about security. when other process acquire the memory in a certain segment, we can give access to that exact segment without splitting or duplicating data.

Also, segmentation offer flexibility when the segments is varying in size. We can allocate the segments in different size and contiguously.This improves CPU utilization, because we don't have to jump between pages for a segment, unlike in paging technique.

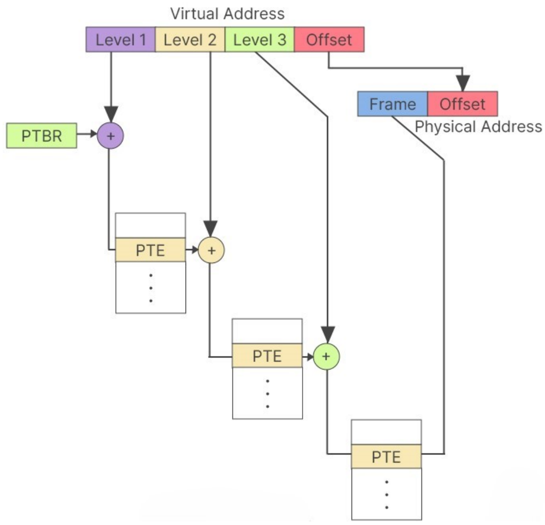
Beside that, the Segment Table consume less memory compared to Page Table. We can briefly search for a certain segment also.

Question 3: What will happen if we divide the address to more than 2 levels in the paging memory management system?

Multilevel-Paging is the memory management mechanism in which it breaks down the virtual address space into smaller pieces so that easier to manage through Page Table. Each entry of the page table represent the index of the frame mapped or the index of the entry in lower level. Then we combining the physical address of the frame with the offset in virtual address to get the real address of the instruction wanted.

Here is the example of splitting the 22-bits virtual address space into multilevel bits:

1. Three level: 5 bits for level 1 + 5 bits for level 2 + 4 bits for level 3 + 8 bits offset
2. Four level: 4 bits for level 1 + 4 bits for level 2 + 3 bits for level 3 + 3 bits for level 4 + 8 bits offset



Advantages:

1. Lookup faster:We do not need to traverse n entries to access the n-th page.
2. Less memory overhead for each Page Table: Less entry to hold for each table.
3. Suitable for large address space
4. Only a portion of entries in the system is created to manage the process.

Disadvantages:

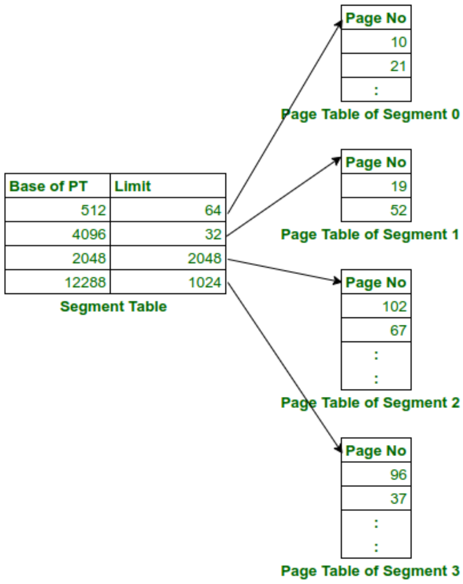
1. Increase overhead for page table lookup: We have to go through multiple step/level the get the physical address
2. More space needed to create high-level page tables, even when we need less space for each page table
3. Harder to create and debug

Question 4: What are the advantages and disadvantages of segmentation with paging?

As i explained about paging and segmentation above, we have a few knowledge about these technique. Each of them have pros and cons that are opposite to the opposite one. So, there is a mixed mechanism to lessen the disadvantages of them, Segmentation with Paging. The process is firstly divided into segments according to their purposes. Then these segments is divided into fixed-size pages and mapped to frames in physical devices.

Advantage

- A large segment can be divided into multiple pages, so that it can fit in ram without necessarily having a large sequential segment within.
- Decrease the effect of external fragmentation.
- Optimize memory usage and more logical due to grouping data according to their purposes (stack,heap, data, ...)



- Easier to share pages and segments between processes.
- Decrease page table size, because it only need to represent the data according to that segment.
- More secure between segments since it has its own regions, and limits other segment's fault access

Disadvantages:

- Still have internal fragmentation since the size of a frame is alligned.
- Need to handle multiple pages and segments, increase overhead for lookup
- Storing page directory for processes is expensive for system that has limited size.

Question 5: What are the advantages of using the scheduling algorithm described in this assignment compared to other scheduling algorithms?

The scheduling algorithm implemented in this assignment is Multi-Level Queue (MLQ) scheduling with the following characteristics:

- **Fixed Priority Levels:** Processes are assigned priorities (0-139) when loaded, based on their type or configuration
- **Multiple Ready Queues:** One queue exists for each priority level
- **Priority-Based Inter-Queue Scheduling:** The scheduler always selects processes from the highest-priority non-empty queue
- **Round-Robin Intra-Queue Scheduling:** Within a single priority queue, processes are scheduled using Round-Robin with fixed time slices (implemented by the slot mechanism)
- **Preemption:** Higher-priority process arrivals preempt currently running lower-priority processes

Comparative advantages of MLQ against other common scheduling algorithms:

- MLQ vs. First-Come, First-Served (FCFS): MLQ provides significantly better responsiveness for high-priority processes. It eliminates the "convoy effect" where short or important tasks get blocked behind long-running, low-priority tasks that arrived earlier. MLQ ensures critical processes gain CPU access promptly based on their assigned priority.
- MLQ vs. Shortest Job First (SJF)/Shortest Remaining Time First (SRTF): MLQ does not require predicting future CPU burst times, which is a major practical limitation of SJF/SRTF. Instead, MLQ relies on fixed priorities that can be determined in advance, making it more feasible to implement in real systems where future behavior is unknown. It also generally prevents the starvation problem inherent in pure SJF/SRTF where long jobs might never execute during continuous streams of short jobs.
- MLQ vs. Simple Priority Scheduling: By incorporating Round-Robin scheduling within each priority level, MLQ prevents CPU monopolization by a single high-priority process. This introduces fairness among processes of equal priority, ensuring that all processes at a given priority level receive CPU time. Simple priority scheduling might allow one high-priority process to run indefinitely until completion, even when other equally important processes are waiting.
- MLQ vs. Simple Round Robin (RR): MLQ allows for differentiation between process types and importance levels. Critical or interactive processes can receive higher priorities to ensure prompt CPU allocation, resulting in better overall system responsiveness. Simple RR treats all processes equally regardless of their importance or requirements, which fails to optimize system performance across diverse workloads.

Question 6: What happens if the syscall job implementation takes too long execution time?

In the context of system call (`syscall`) implementation, one important consideration is the duration of execution. While most syscalls are expected to complete quickly, a long-running syscall can negatively impact the responsiveness and stability of the operating system, such as:

- **Kernel Blocking:** In non-preemptive kernel architectures (as simplified OS designs), a syscall holds control over the entire kernel until it finishes. This means no other syscall or scheduling operation can proceed, effectively stalling the whole system.
- **Responsiveness Issues:** Since all other processes are blocked during the syscall execution, user interaction and background tasks are delayed, leading to a poor user experience. In extreme cases, this can result in system "freezes".
- **Scheduling Delays and Starvation:** A lengthy syscall can delay process switching and cause starvation for other runnable processes. This is especially problematic in real-time systems where timing guarantees are critical.
- **Concurrency Hazards:** If the syscall acquires locks (e.g., mutexes on queues or lists), holding them for too long increases the risk of deadlocks and race conditions. This is particularly dangerous if other kernel components rely on the same resources.

To address the risks of long-running syscalls, several mitigation strategies can be employed:

- **Task Chunking:** Break the operation into smaller pieces, allowing multiple calls or yield points between chunks.
- **Asynchronous Processing:** Defer the heavy work to a background task and return from the syscall quickly.
- **Timeout or Bounded Execution:** Limit execution time or steps per syscall invocation to avoid monopolizing CPU time.

In our simplified OS, due to the limited number of processes and small-scale scheduling queues, we assume all syscalls complete in a short, bounded time. However, we remain cautious and avoid long-running loops within syscall implementations to prevent system hangs.

Question 7: What happens if the synchronization is not handled in your Simple OS? Illustrate the problem of your simple OS (assignment outputs) by example if you have any.

Synchronization is an attempt to handle the order of process's execution to avoid common errors like data race, race condition, etc.

Race condition occurs when multiple process access to the shared memory in an undesirable order, resulting in the fault result after a sequence of operations.

Otherwise, data race happens if multiple processes did not wait other processes complete their execution, they apply several operation on the same space at the same time, leading to the unexpected result.

To solve these problems, many methods like mutex lock, semaphores, monitors, have been applied to ensure the synchronization of the program.

If the program has not applied the synchronization into it, it can cause the unexpected value in memory regions.

References