XV6: A COW Fork and Scheduling Experiment

Team Name: system_haters
Team Members: Gaurav Kumar, Kakadiya Omikumar Anilbhai
github link:


12
1
2
3
4
5
6
7
8
9
10
11

Domain of Project

Part 1: Copy-On-Write (COW) in xv6

- **Domain**: Operating systems, memory management.
- **Why this area?**
  - COW is a practical optimization to reduce memory overhead during process creation, making it essential for systems handling high workloads.
  - Implementing and evaluating COW in xv6 offers insights into how modern OS optimize resource utilization efficiently.

Part 2: Scheduling Experiments in xv6

- **Domain**: Operating systems, CPU scheduling.
- **Why this area?**
  - CPU scheduling determines process execution order, impacting system performance.
  - The focus is on comparing different scheduling algorithms based on their runtime in xv6, providing a clear understanding of their efficiency under various workloads.

Problem Description

Problem Description

The project focuses on optimizing and analyzing critical aspects of an operating system:

1. **Copy-On-Write (COW)**: Efficient memory usage during process creation.
2. **Scheduling Algorithms**: Understanding and comparing the efficiency of different CPU scheduling strategies.

Problem Statement

- In modern operating systems, memory and CPU are key resources that demand efficient utilization.
- This project implements COW in xv6 to demonstrate memory savings during process forking and compares scheduling algorithms in xv6 based on their runtime performance to evaluate their efficiency.

Scope, Goals, and Deliverables

Scope

- **COW Implementation**: Analyze how COW reduces memory usage by deferring page duplication during process creation.
- **Scheduling Comparisons**: Focus on evaluating different scheduling algorithms based on their runtime under various workloads.

Goals

- Implement COW in xv6 and measure memory savings effectively.
- Compare scheduling algorithms (FCFS, Round Robin, Priority Scheduling, MLFQ) in xv6 by benchmarking their runtime efficiency.

Deliverables

1. A working COW implementation with measurable memory savings.
2. Comparative results and analysis of scheduling algorithms based on their runtime.

Component of Project Part I: COW fork

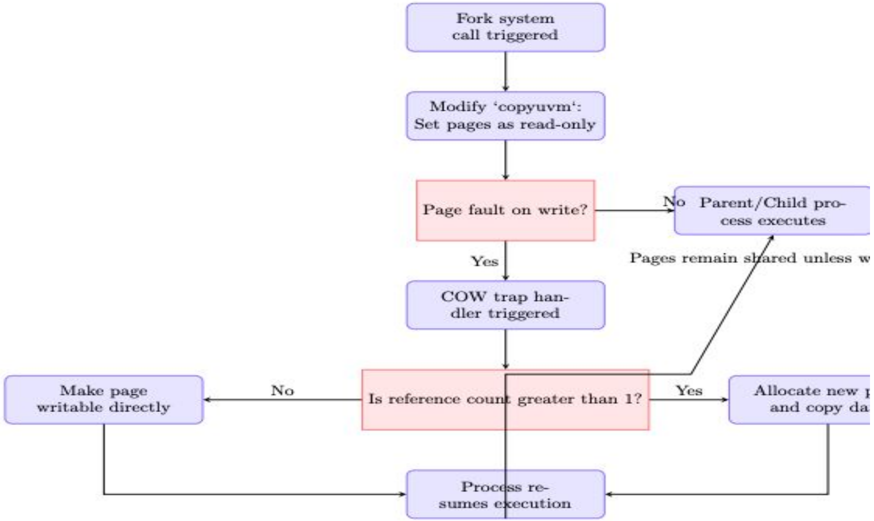
The implementation of Copy-On-Write (COW) in xv6 involved the following key modifications:

1. **Modification to copyuvm Function**
   - The copyuvm function, responsible for duplicating the address space during fork(), was modified to make all shared pages **read-only**, by marking the pages as non-writable, any write attempt by either the parent or child triggers a **page fault**, allowing the kernel to defer page duplication until it is actually needed.
2. **Trap Handler for Page Faults**
   - A new trap handler was implemented to specifically handle **COW_Fault.** This handler identifies when a write attempt on a shared read-only page occurs and ensures the process requesting the write is allocated a separate, writable copy of the page.
3. **Reference Count Management**
   - A dedicated mechanism was created to track the **reference count** of each physical page in memory.
   - The reference count is incremented when a page is shared between processes and decremented when a process releases a page.
   - The COW trap handler checks the reference count:
     - If the reference count is greater than one, a new page is allocated, and the data from the shared page is copied into it.
     - If the reference count is one, the page is unshared and made writable directly.

Design Part I: COW fork

Components: copyuvm, Page tables, COW handler, Reference counter

Flow: Fork → Pages set as read-only in copyuvm→ Write causes page fault → COW handler → Reference count check → Allocate new page or make writable → Resume execution



Implementation Details Part I: COW fork

- **Abstract Idea**: Save memory during fork() by sharing pages until a write occurs.
- **Implementation**:
    - Modified copyuvm() to set pages as read-only.
    - Created a COW trap handler to handle page faults and allocate new pages.
    - Added reference counting to manage shared pages.
- **Challenges**:
    - Figuring out the flow of how memory is allocated during fork().
- **Testing**:
    - Ran benchmarks to measure memory savings during fork().

- Modified copyuvm() to set pages as read-only.

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");

    //make the page as non writeable so when written, triggers COW_FAULT
    *pte=(~PTE_W) & *pte;
    flags=PTE_FLAGS(*pte);
    pa = PTE_ADDR(*pte);

    if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) {
      freevm(d);
      lcr3(V2P(pgdir));
      return 0;
    }
    i_rCount(pa); //increase the refernce count to track number of reference count to this page.
  }
  lcr3(V2P(pgdir));//flush tlb to update the cache with new flag permission.
  return d;

bad:
  freevm(d);
  return 0;
}
```

- COW Fault handling

```c
void COW_FAULT(uint err_code)
{
  uint va = rcr2();       //get virtual address of COW_FAULT
  pte_t *pte;
  pte = walkpgdir(myproc()->pgdir, (void*)va, 0);
  uint pa = PTE_ADDR(*pte);//get physical address
  uint ref_count = get_rCount(pa);// get ref. count of curr. page
  if(ref_count < 1){
    panic("Error in COW_FAULT: Incorrect Reference Count");
  }
  else if(ref_count == 1){
    *pte = PTE_W | *pte;   // writable now since alone
    lcr3(V2P(myproc()->pgdir));// update TLB as PTE changed
    return;
  }
  else{
    char* mem = kalloc();
    if(mem != 0){  //page is available
      memmove(mem, (char*)P2V(pa), PGSIZE); //now map the pages to the child identical to parent.
      *pte =  PTE_U | PTE_W | PTE_P | V2P(mem);//update the permission to write permissions.
      d_rCount(pa);
      lcr3(V2P(myproc()->pgdir));// update the TLB as PTE changed.
      return;
    }
    myproc()->killed = 1;//if kalloc failed this happens!
    cprintf("Error in COW_FAULT: Memory out of bounds, killing process
    %s with pid %d\n", myproc()->name, myproc()->pid);
    return;
  }
  lcr3(V2P(myproc()->pgdir));//flush TLB as PTE changed
}
```

- Reference count functions

```
void
i_rCount(uint pa)
{
  if(pa >= PHYSTOP || pa < (uint)V2P(end))
    panic("i_rCount");
  acquire(&kmem.lock);
  kmem.page_rCount[pa >> PGSHIFT] = kmem.page_rCount[pa >> PGSHIFT] + 1;
  release(&kmem.lock);
}


void
d_rCount(uint pa)
{
  if(pa >= PHYSTOP || pa < (uint)V2P(end))
    panic("d_rCount");
  acquire(&kmem.lock);
  kmem.page_rCount[pa >> PGSHIFT] = kmem.page_rCount[pa >> PGSHIFT] - 1;
  release(&kmem.lock);
}

uint
get_rCount(uint pa)
{
  if (pa >= PHYSTOP || pa < (uint)V2P(end)) {
    cprintf("get_rCount: Invalid pa=%p (PHYSTOP=%p, V2P(end)=%p)\n", pa, PHYSTOP, V2P(end));
    panic("getReferenceCount");
  }
  uint count;
  acquire(&kmem.lock);
  count = kmem.page_rCount[pa >> PGSHIFT];
  release(&kmem.lock);
  return count;
}
```

Evaluation I: COW fork

1. Evaluation Objectives

- Does the implementation reduce memory usage by sharing pages until modified?
- How many pages are saved by COW compared to traditional fork()?
- Does the memory reclaim correctly after the child processes are terminated?

Test Description:

- The parent process initializes a large shared array (shared_array) to ensure substantial page usage.
- A child process is forked and modify the array to trigger Copy-On-Write (COW).

Test Functions:

- getfreepages(): Used to measure free pages in the system before and after writes.
- fork(): Creates child processes to test COW behavior.

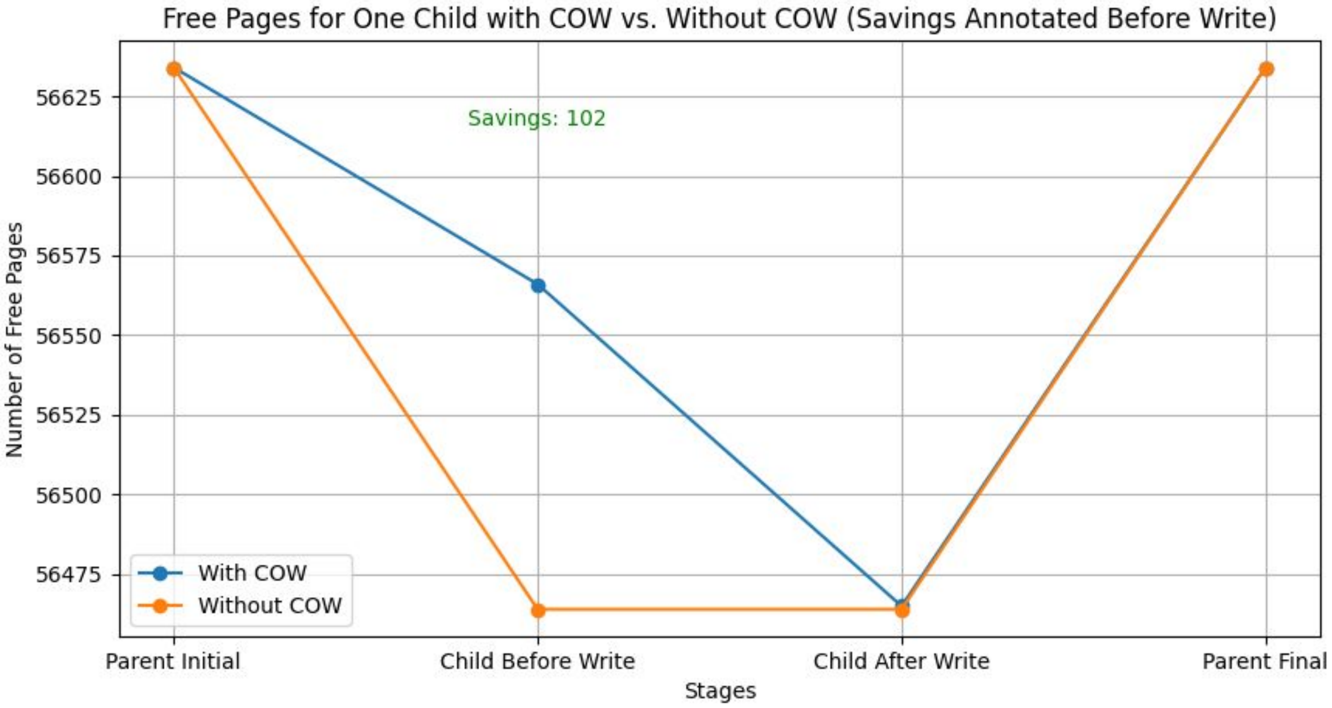
2. Workloads

- **Shared Data Size**: Large array size (102400 integers) to induce approximately 100 page faults.
- **Child Actions**: Modify the shared array to force COW and measure the memory allocation.
- **Parent Actions**: Track memory usage before and after child process complete.

3. Results Summary

- **Pages Saved by COW**:
  - The number of pages saved before any modification (shared pages).
  - Pages allocated post-modification by each child due to COW.
- **Parent's Memory**: The memory usage reverts to the initial state after reaping the child processes.

Pages saved by COW fork



Component of Project Part II: Scheduling Experiment

The major changes in the project were made to the `proc` structure and the `scheduler` function to support enhanced scheduling features and performance tracking.

Key Changes to `proc` Structure

1. **Process Timings**: Track process creation, runtime, end time, and I/O time for precise performance metrics.
2. **Priority Management**: Store and manage process priorities for **Priority-Based Scheduling (PBS)**.
3. **Queue Management**: Enable multi-level scheduling by tracking the queue and time spent in each queue.
4. **Execution Metrics**: Count the number of process runs, ticks used, and last execution timestamp to assist in fair scheduling decisions.

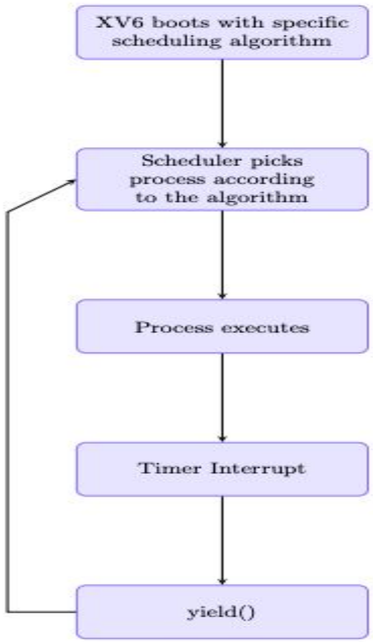
Key Changes to `scheduler` Function

1. **Enhanced Scheduling Logic**: Incorporate the new priority and queue-based scheduling mechanisms.
2. **Dynamic Queue Adjustments**: Allow processes to move between queues based on execution metrics.
3. **Fine-Grained Metrics Tracking**: Update process timing and execution details during context switches for more accurate comparisons.

Design Part II: Scheduling Experiment

Components: proc structure (with added fields), Scheduler function, Queues.

Flow: XV6 boots with specific scheduling algorithm → Scheduler picks process according to the algorithm → → Process executes → Timer Interrupt → yield() → Scheduler picks next RUNNABLE process according to algorithm



Implementation Details Part II: Scheduling experiment

- **Abstract Idea**: Compare scheduling algorithms based on runtime.
- **Implementation**:
  - Added fields to struct proc for tracking runtime, priority, queues, etc.
  - Modified scheduler() function to implement different algorithms (FCFS, RR, PBS, MLFQ).
- **Challenges**:
  - Tuning parameters for fairness in MLFQ.
- **Testing**:
  - Collected runtime data for each algorithm using a set of predefined workloads.
  - Generated comparative plots.

- Added fields to struct proc for tracking runtime, priority, queues, etc.

```c
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 ctime;                   //! for waitx - creation time
  int rtime;                   //! for waitx - totalcputime
  int etime;                   //! for waitx - endtime
  int iotime;                  //! for waitx - iotime
  int priority;                //! priority of process for PBS
  int queue;                   //! queue where the process belongs
  int q_ticks[5];              //! ticks at each q
  int curr_q_ticks;            //! ticks in current queue
  int n_run;                   //! number of runs
  int lastWorkingticks;        //! last tick the process worked at


};
```

- FCFS Implementation

```c
struct proc *nextone = NULL; //this is the next process to run
acquire(&ptable.lock);       //lock table
//loop through process table to find the process with min creation time
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->state != RUNNABLE)
    continue;
  if (nextone == NULL)
  {
    nextone = p;
  }
  else if (p->ctime < nextone->ctime)
  {
    nextone = p; //this has lower creation time, change nextone
  }
}
if (nextone != NULL)
{
  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  p = nextone; //p is the next one to run, so change it to nextone
  //oldcode from default
  c->proc = p;
  switchuvm(p);
  p->n_run++;
  p->state = RUNNING;
  p->lastWorkingticks = ticks;
  cprintf("Process ID: %d, creation_time: %d, ticks: %d\n", p->pid, p->ctime, ticks);
  swtch(&(c->scheduler), p->context);
  switchkvm();
  p->lastWorkingticks = ticks;

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

- RR Implementation

```c
acquire(&ptable.lock);
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->state != RUNNABLE)
    continue;

  // 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->n_run++;
  p->state = RUNNING;
  p->lastWorkingticks = ticks;
  cprintf("ticks: %d\n", ticks);
  swtch(&(c->scheduler), p->context);
  switchkvm();
  p->lastWorkingticks = ticks;

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

- PBS Implementation

```c
struct proc *minone = NULL; //the current minimum
acquire(&ptable.lock);      //lock table
//loop through process table to find the process with min priority
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->state != RUNNABLE)
    continue;
  if (minone == NULL)
  {
    minone = p;
  }
  else if (p->priority < minone->priority)
  {
    minone = p; //this has lower priority , change minone
  }
}
```

```c
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->state != RUNNABLE)
    continue;
  if (p->state == RUNNABLE && p->priority == minone->priority)
  { //if p is runnable and it has the minimum priority run in RR style
    // 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->n_run++; //update
    p->state = RUNNING;
    p->lastWorkingticks = ticks;
    if(ticks%200)
    cprintf("Running process: %d creation time: %d with priority: %d, ticks: %d\n", p->pid, p->ctime, p->priority, ticks);
    swtch(&(c->scheduler), p->context);
    switchkvm();
    p->lastWorkingticks = ticks;

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
    // here the process got yielded - 1 tick over
    int flag = 0; //flag to check if better process is there.. if it is there break from RR
    for (struct proc *nextone = ptable.proc; nextone < &ptable.proc[NPROC]; nextone++)
    {
      if (nextone->state != RUNNABLE)
      {
        continue;
      }
      if (nextone->state == RUNNABLE && nextone->priority < minone->priority)
      {
        flag = 1; //better is one is here... stop RR
        break;
      }
    }
    if (flag)
    {
      break; //hey break RR go back to normal mode
    }
  }
}
release(&ptable.lock); //release lock and go for another round
```

- MLFQ Implementation

- Code for handling starvation

```
struct proc *nextone = NULL; //this is the nextone to execute
acquire(&ptable.lock);
struct proc *queues[5] = {0}; //this will hold process chosen from each queue
//next we age all processess to avoid starvation
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->state != RUNNABLE)
    continue;

  if (p->queue > 0 && (ticks - (p->lastWorkingticks)) > q_max_time[p->queue])
  {
    p->queue--;
    if (!PLOT)
    {
      // cprintf("%d promoted to queue %d from %d due to aging\n", p->pid, p->queue, p->queue+1);
      // cprintf("%d,%d,%d\n", ticks, p->pid, p->queue);
    }
    p->lastWorkingticks = ticks;
  }
}
```

- Runnable process for each queue

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

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

  if (p->queue != 4)
  {
    if (queues[p->queue] == 0)
    {
      queues[p->queue] = p;
    }
    else if (p->ctime < queues[p->queue]->ctime && queues[p->queue]->state == RUNNABLE)
    {
      queues[p->queue] = p;
    }
  }

  else
  {
    if (queues[p->queue] == 0)
    {
      queues[p->queue] = p;
    }

    else if (queues[p->queue]->lastWorkingticks > p->lastWorkingticks)
    {
      queues[p->queue] = p;
    }
  }
}
```

- prioritising the queue

```
for (int i = 0; i < 5; i++)
{
  if (queues[i] != 0)
  {
    queues[i]->curr_q_ticks = 0;
    queues[i]->n_run++;
    while (queues[i]->state == RUNNABLE)
    {
      nextone = queues[i];
      c->proc = nextone;
      nextone->lastWorkingticks = ticks;
      switchuvm(nextone);
      nextone->state = RUNNING;
      cprintf("%d,%d,%d\n", ticks, queues[i]->pid, queues[i]->queue);
      swtch(&(c->scheduler), nextone->context);
      switchkvm();
      c->proc = 0;

      if (nextone->curr_q_ticks >= max_tick_of_q[i]) //if it finished i
        break;
    }

    if (queues[i]->curr_q_ticks >= max_tick_of_q[i] && i != 4) //it use
    {
      queues[i]->queue++;
      if (!PLOT)
      {
        // cprintf("%d used up its time slice and demoted from queue %d
        // cprintf("%d,%d,%d\n", ticks, queues[i]->pid, queues[i]->qu
      }
    }
    queues[i]->lastWorkingticks = ticks;
    break;
  }
}
release(&ptable.lock);
```

- Time slices used for benchmarking

```
int max_tick_of_q[5] = {1, 2, 4, 8, 16}; //
// int q_max_time[5] = {1, 2, 3, 4, 5};
int q_max_time[5] = {1, 5, 10, 15, 20}; //m
// int q_max_time[5] = {1, 20, 30, 40, 120}
```

Evaluation II: Scheduling

1. Evaluation Objectives (Questions)

- How does the choice of wait times in different MLFQ queues impact its performance?
- How do CPU-bound and I/O-bound processes behave under the MLFQ scheduling policy?
- How do FCFS, RR, PBS, and MLFQ compare in terms of runtime efficiency?

2. Setup

- **Test Description**:
  - Configured MLFQ with varying wait times across queues to observe its effect on process performance.
  - Created workloads with CPU-bound and I/O-bound processes to study queue switching dynamics.
  - Compared the runtime performance of four scheduling algorithms: FCFS, Round Robin (RR), Priority-Based Scheduling (PBS), and Multilevel Feedback Queue (MLFQ).

Evaluation II: Scheduling

3. Workloads

- **CPU-bound Processes**: Long-running compute-intensive processes.
- **I/O-bound Processes**: Processes with frequent I/O wait states.

4. Parameters/Configurations (Independent Variables)

- **Wait Times in MLFQ Queues**: Configured to prioritize different types of processes at various levels.
- **Scheduling Algorithms**: FCFS, RR, PBS, and MLFQ.
- **Workload Type**: CPU-bound, I/O-bound

5. Metrics (Dependent Variables)

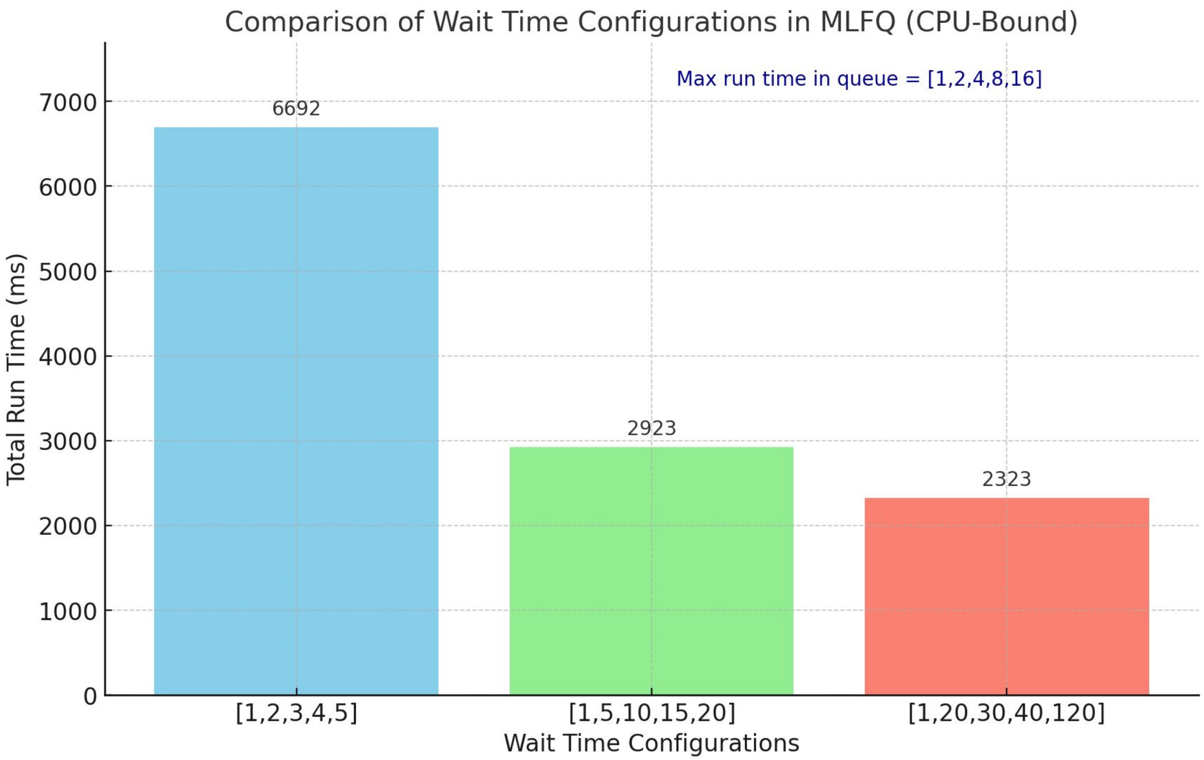
- **Runtime**: Total time taken by each algorithm to complete all processes in a workload.
- **Queue Switching Dynamics**: Frequency and pattern of processes moving between queues in MLFQ.
- **Process Completion Efficiency**: Time taken for individual CPU-bound and I/O-bound processes to complete.

Evaluation II: Scheduling

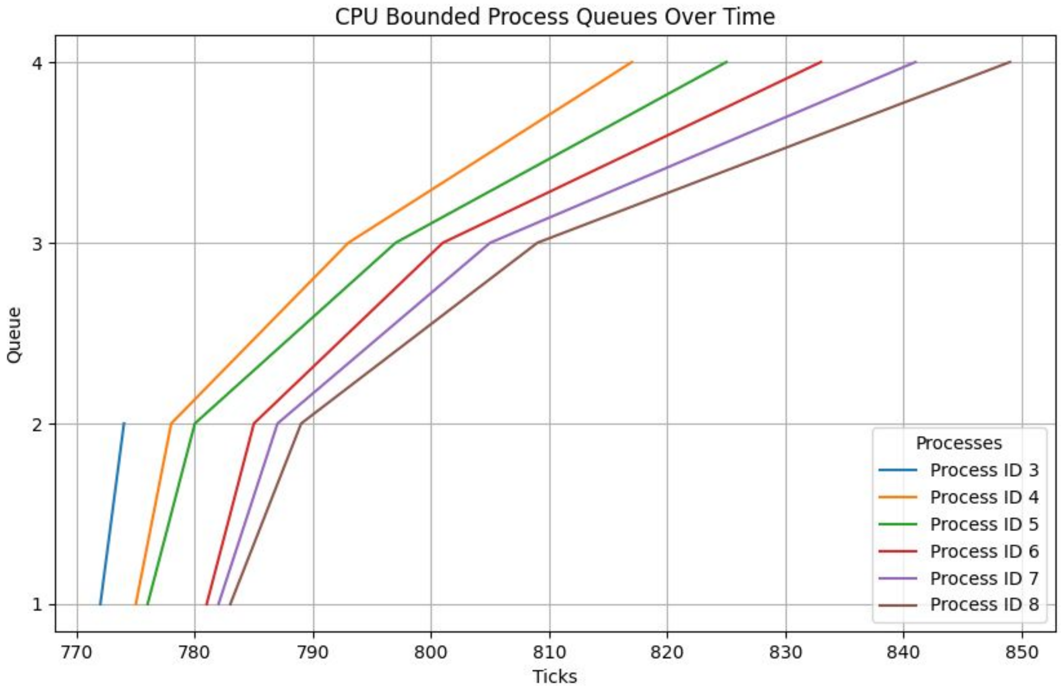
6. Results Summary

- **MLFQ Performance**: Wait times in queues significantly influence process prioritization and efficiency.
- **CPU-Bound vs. I/O-Bound**: MLFQ adapts dynamically, with CPU-bound processes gradually moving to lower-priority queues, while I/O-bound processes stay at higher priorities.
- **Algorithm Comparison**:
  - **FCFS**: Simple but inefficient for mixed workloads.
  - **RR**: Fair but leads to higher context switching overhead.
  - **PBS**: Good prioritization but less adaptive to dynamic workloads.
  - **MLFQ**: Best balance of fairness and efficiency for varied workloads.

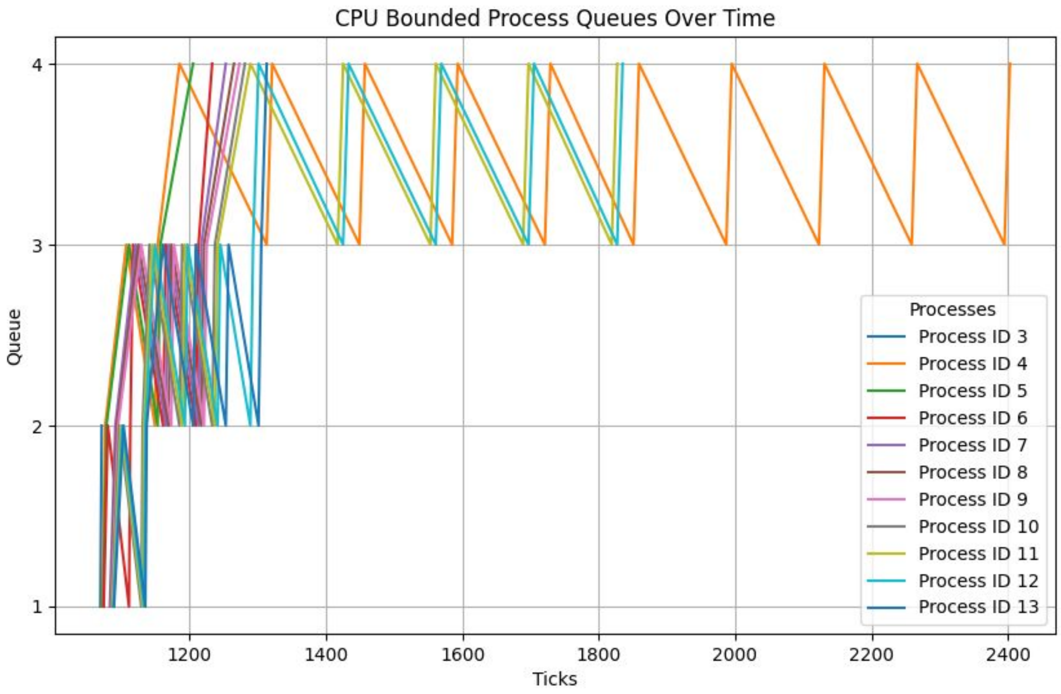
MLFQ Configuration w.r.t queue wait and run time



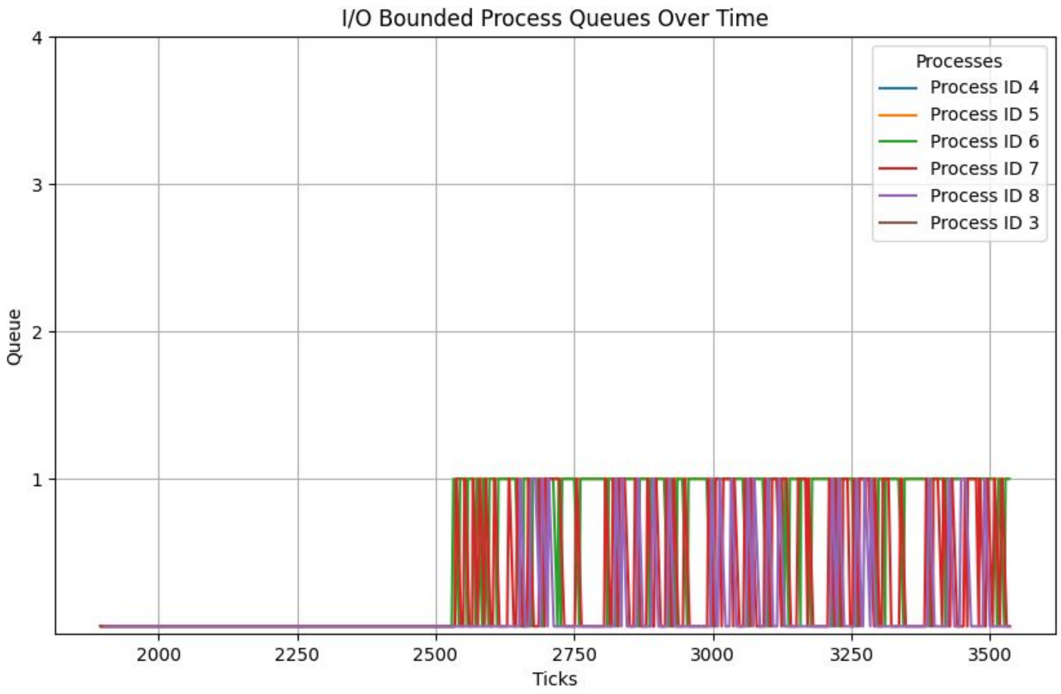
CPU bound Process runs in MLFQ (for 5 processes)



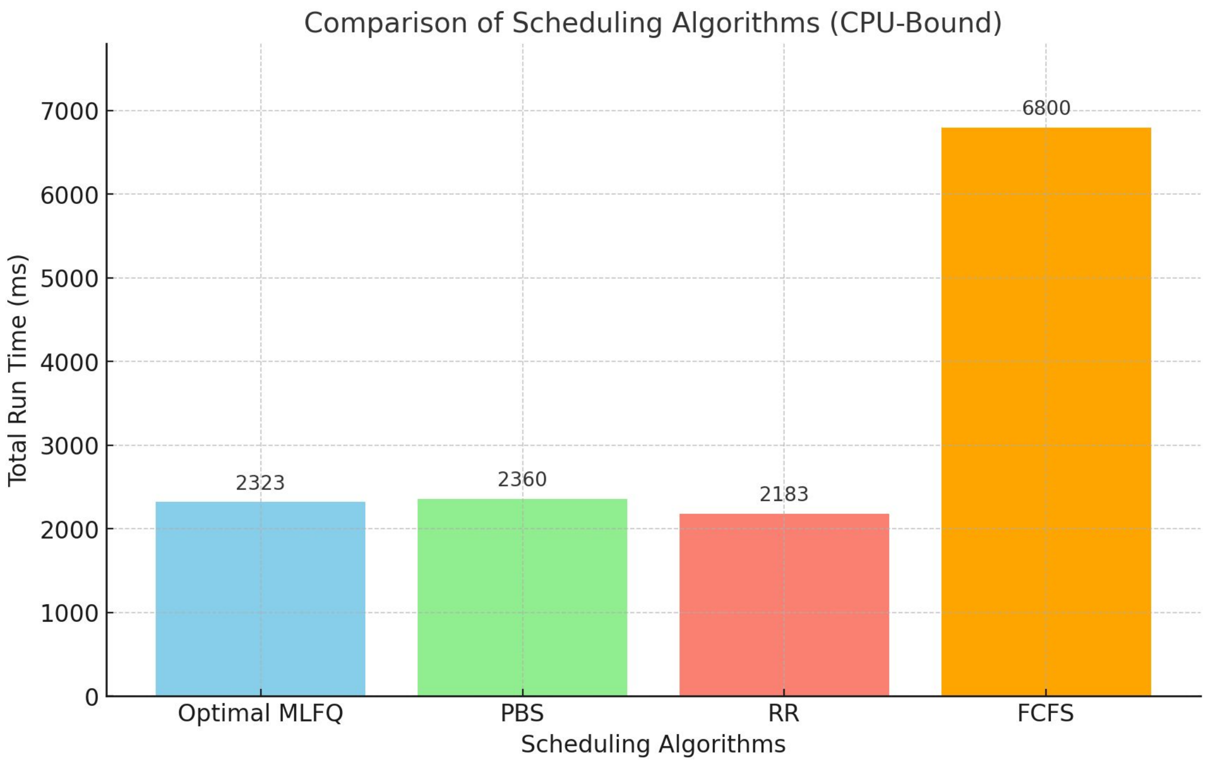
CPU bound processes run in MLFQ (for 10 processes)



I/O bound process run in MLFQ (5 processes)



Comparison of different scheduling algorithms (w.r.t 10 processes with same workload.)



Unfinished Scope

Copy-On-Write (COW)

1. **Advanced Performance Metrics**:
   - Current implementation focuses on memory savings, but detailed performance analysis (e.g., page fault handling overhead, execution time impact) has not been conducted.
2. **Stress Testing**:
   - The COW implementation is tested on simple workloads; testing under heavy workloads or multi-level fork scenarios remains incomplete.

Scheduling

1. **Lottery Scheduling Comparison**:
   - Implementing and comparing **Lottery Scheduling**, a probabilistic scheduling algorithm, with **FCFS**, **RR**, **PBS**, and **MLFQ** can introduce the concept of fairness and unpredictability in scheduling.
   - This comparison could highlight how systems handle stochastic workloads or systems requiring randomness.

Challenges

Handling Complex Page Faults in COW:

- Ensuring that the page fault handler correctly identifies when a page needs to be copied, and managing the memory mapping during the page fault process, was challenging.
- Ensuring the child processes did not modify shared pages unnecessarily, while also maintaining efficient memory usage, required careful design.

Implementing Efficient Process Scheduling Algorithms:

- Comparing the performance of different scheduling algorithms such as **FCFS**, **RR**, **PBS**, and **MLFQ** required careful implementation to measure and assess runtime fairly.
- Handling the complexity of scheduling decisions, such as context switching and accounting for CPU-bound vs. I/O-bound processes, added layers of complexity.

Reflection

Understanding Memory Optimization with COW:

- Implementing **Copy-On-Write (COW)** provided a deep understanding of how memory can be efficiently shared and optimized between parent and child processes.
- The real-time effect of memory-saving techniques and how COW helps reduce memory usage during process creation and modification was a key learning experience. Tracking the number of free pages before and after forking allowed for clear insights into the efficiency of the technique.

Exploring Multiple Scheduling Algorithms:

- The exploration of various scheduling algorithms, such as **FCFS**, **Round Robin (RR)**, **PBS**, and **MLFQ**, was intriguing, especially when analyzing their different performances under varying workloads. Understanding how each algorithm prioritizes tasks and impacts system responsiveness was a valuable aspect of the project.
- The challenge of comparing **CPU-bound** and **I/O-bound** processes, and understanding how they are handled differently by scheduling algorithms, was also quite engaging. It was particularly interesting to see the **MLFQ** algorithm adapt to these changing workloads.

Conclusion

This project provided valuable hands-on experience with memory optimization through Copy-On-Write (COW) and in-depth analysis of scheduling algorithms like FCFS, RR, PBS, and MLFQ. By comparing their performance in various scenarios, I gained a deeper understanding of how different algorithms impact system efficiency and responsiveness. The exploration of memory management and process scheduling highlighted the trade-offs between performance, fairness, and resource utilization, providing a comprehensive view of system-level optimizations and their practical implications.

References

- ***XV6 source code*** *– used for implementing and testing Copy-On-Write and scheduling algorithms.*
- ***"Operating Systems: Three Easy Pieces" (OSTEP)*** *by Arpaci-Dusseau and Arpaci-Dusseau – provided theoretical insights into scheduling and memory management concepts.*