Assignment-4 : Enhancing XV-6

```
Gowlapalli Rohit        Losetti Mourya
   2021101113             2021101121
```

> - Possible Exploitation of MLFQ Policy by a process

If a process voluntarily relinquishes control of the CPU(eg. For doing I/O), it leaves the queuing network

When the process becomes ready again after the I/O, it is inserted at the tail of the same queue, from which it is relinquished earlier

This can be exploited by a programmer (who knows the Time-slices assigned to each queue) by adding short I/O burst of frequency < Time slice of the corresponding queue (remaining / Just before its Time-slice is going to be finished) which abdicates itself from the CPU and rejoins the same queue promptly thereby avoiding degradation of priority

This causes the process to be free from preemption(kind of) and can complete its CPU/IO cycles in the Same/Higher priortiy Queue until its executed completely

---

> - waitx command is referred from (to implement schedulertest.c) https://github.com/AbhijnanVegi/xv6-tutorial/commit/48b6022d67b0eb54b43f3fe5deb4831f0a01e9bb
> - schedulertest.c is referred from https://courses.iiit.ac.in/pluginfile.php/142146/mod_forum/attachment/71430/schedulertest.c?forcedownload=1 and is modified as well
> - alarmtest.c is referred from (To test sigalarm and sigreturn) https://gist.github.com/AbhijnanVegi/0df4ec53cea38618b36ceb28c0ae573b
> - It is assumed that schedulertest.c is used to check validity of Specification-2 (not usertests.c) as instructed by Group-8 TA

- > Command to be executed

  $ `make qemu`

Specification 1: System Calls

- > System Call 1 : trace

  $ `strace mask command [args]`

```
+ xv6-riscv/kernel/syscall.c

  static uint64 (*syscalls[])(void) = {
      [SYS_fork] sys_fork,
      [SYS_exit] sys_exit,
      [SYS_wait] sys_wait,
      [SYS_pipe] sys_pipe,
      [SYS_read] sys_read,
      [SYS_kill] sys_kill,
      [SYS_exec] sys_exec,
      [SYS_fstat] sys_fstat,
      [SYS_chdir] sys_chdir,
      [SYS_dup] sys_dup,
      [SYS_getpid] sys_getpid,
      [SYS_sbrk] sys_sbrk,
      [SYS_sleep] sys_sleep,
      [SYS_uptime] sys_uptime,
      [SYS_open] sys_open,
      [SYS_write] sys_write,
      [SYS_mknod] sys_mknod,
      [SYS_unlink] sys_unlink,
      [SYS_link] sys_link,
      [SYS_mkdir] sys_mkdir,
      [SYS_close] sys_close,
      [SYS_strace] sys_strace,
      [SYS_sigalarm] sys_sigalarm,
      [SYS_sigreturn] sys_sigreturn,
      [SYS_set_priority] sys_set_priority,
      [SYS_waitx]   sys_waitx,
      [SYS_settickets] sys_settickets,
  };

  char *syscallnames[] = {
      "",
      [SYS_fork] "fork",
      [SYS_exit] "exit",
      [SYS_wait] "wait",
      [SYS_pipe] "pipe",
      [SYS_read] "read",
      [SYS_kill] "kill",
      [SYS_exec] "exec",
      [SYS_fstat] "fstat",
      [SYS_chdir] "chdir",
      [SYS_dup] "dup",
      [SYS_getpid] "getpid",
      [SYS_sbrk] "sbrk",
      [SYS_sleep] "sleep",
      [SYS_uptime] "uptime",
      [SYS_open] "open",
      [SYS_write] "write",
      [SYS_mknod] "mknod",
      [SYS_unlink] "unlink",
      [SYS_link] "link",
      [SYS_mkdir] "mkdir",
      [SYS_close] "close",
      [SYS_strace] "trace",
      [SYS_sigalarm] "sigalarm",
      [SYS_sigreturn] "sigreturn",
      [SYS_set_priority] "set_priority",
      [SYS_waitx]   "waitx",
      [SYS_settickets] "settickets",
  };

  int syscallarguments[] = {
      0,
      [SYS_fork] 0,
      [SYS_exit] 1,
      [SYS_wait] 1,
      [SYS_pipe] 1,
      [SYS_read] 3,
      [SYS_kill] 1,
      [SYS_exec] 2,
      [SYS_fstat] 2,
      [SYS_chdir] 1,
      [SYS_dup] 1,
      [SYS_getpid] 0,
      [SYS_sbrk] 1,
      [SYS_sleep] 1,
      [SYS_uptime] 0,
      [SYS_open] 2,
      [SYS_write] 3,
      [SYS_mknod] 3,
      [SYS_unlink] 1,
      [SYS_link] 2,
      [SYS_mkdir] 1,
      [SYS_close] 1,
      [SYS_strace] 1,
      [SYS_sigalarm] 0,
      [SYS_sigreturn] 0,
      [SYS_set_priority] 2,
      [SYS_waitx]   3,
      [SYS_settickets] 1,
  };

  void syscall(void)
  {
    int num;
    struct proc *p = myproc();
    num = p->trapframe->a7;
    int azero = p->trapframe->a0;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
      p->trapframe->a0 = syscalls[num]();
      int rightshift = (p->integermask) >> num;
      int and = rightshift & 1;

      if (and)
      {
        printf("%d: ", p->pid);
        printf("syscall ");
        printf("%s ", syscallnames[num]);
        printf("(");
        if (syscallarguments[num] > 1)
        {
          printf("%d", azero);
          for (int i = 0; i < syscallarguments[num] - 1; i++)
          {
            printf(" %d", argraw(i + 1));
          }
          printf(")");
          printf(" -> ");
          printf("%d\n", p->trapframe->a0);
        }
        else
        {
          printf("%d", azero);
          printf(")");
          printf(" -> ");
          printf("%d\n", p->trapframe->a0);
        }
      }
    }
    else
    {
      printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
      p->trapframe->a0 = -1;
    }
  }
```

strace runs the specified command until it exits.

It intercepts and records the system calls which are called by a process during its execution.

It takes one argument, an integer mask, whose bits specify which system calls to trace.

Following properties of each system call are displayed through strace as follows:

> 1. The process id
> 2. The name of the system call
> 3. The decimal value of the arguments(xv6 passes arguments via registers).
> 4. The return value of the syscall.

The trace system call enables tracing for the process that calls it and any children that it subsequently forks, but doesn't affect other processes.

```
Output

$ strace 32 grep hello README
5: syscall read (3 2768 1023) -> 1023
5: syscall read (3 2823 968) -> 968
5: syscall read (3 2773 1018) -> 235
5: syscall read (3 2768 1023) -> 0

$ strace 2147483647 grep hello README
6: syscall trace (2147483647) -> 0
6: syscall exec (12240 12208) -> 3
6: syscall open (12240 0) -> 3
6: syscall read (3 2768 1023) -> 1023
6: syscall read (3 2823 968) -> 968
6: syscall read (3 2773 1018) -> 235
6: syscall read (3 2768 1023) -> 0
6: syscall close (3) -> 0
```

- > System Call 2 : sigalarm and sigreturn

  $ `alarmtest`

```
+ xv6-riscv/kernel/proc.h ::
  int is_sigalarm;
  int ticks;
  int ticks_passed;
  uint64 handler;
  struct trapframe *trapframe_copy;

+ xv6-riscv/kernel/sysproc.c ::
  void restore()
  {
    struct proc *p = myproc();
    p->trapframe_copy->kernel_satp = p->trapframe->kernel_satp;
    p->trapframe_copy->kernel_sp = p->trapframe->kernel_sp;
    p->trapframe_copy->kernel_hartid = p->trapframe->kernel_hartid;
    p->trapframe_copy->kernel_trap = p->trapframe->kernel_trap;
    (*p->trapframe) = (*p->trapframe_copy);
  }

  uint64 sys_sigreturn(void)
  {
    restore();
    struct proc *p = myproc();
    p->is_sigalarm = 0;
    return p->trapframe->a0;
  }

+ xv6-riscv/kernel/proc.c ::
```

```
void allocproc():
  if ((p->trapframe_copy = (struct trapframe *)kalloc()) == 0)
    {
      release(&p->lock);
      return 0;
    }
  p->is_sigalarm = 0;
  p->handler = 0;

static void freeproc():
  if (p->trapframe_copy)
    {
      kfree((void *)p->trapframe_copy);
    }

+ xv6-riscv/kernel/trap.c ::

void usertrap(void):
    p->ticks_passed++;
    int diff = p->ticks_passed - p->ticks;
    int check = (p->ticks >= 1) && (diff >= 0) && (!p->is_sigalarm);
    if (check)
    {
      p->ticks_passed = 0;
      p->is_sigalarm = 1;
      (*p->trapframe_copy) = *(p->trapframe);
      p->trapframe->epc = p->handler;
    }
```

A mechanism(primitive form of user-level interrupt/fault handlers) is implemented that periodically alerts a process as it uses CPU time.

This might be useful for compute-bound processes that want to limit how much CPU time they chew up, or for processes that want to compute but also want

A system call sigalarm(interval, handler) system call is implemented

If an application calls alarm(n, fn) , then after every n "ticks" of CPU time that the program consumes, the kernel will cause application function fn to be called. When fn returns, the application will resume where it left off.

Another system call sigreturn() is implemented

sigreturn() resets the process state to before the handler was called. This system call is made at the end of the handler so the process can resume where it left off.

```
Output
```

```
$ alarmtest
test0 start
.......................alarm!
test0 passed
test1 start
....alarm!
...alarm!
...alarm!
....alarm!
...alarm!
...alarm!
...alarm!
...alarm!
...alarm!
....alarm!
test1 passed
test2 start
.........................................alarm!
test2 passed
test3 start
test3 passed
```

Specification 2: Scheduling

Performance Comparision

The below results may vary from machine to machine

> - RR (Round-Robin)

```
Output

$ schedulertest
Process 7 finished
Process 8 finished
Process 6 finished
Process 5 finished
Process 9 finished
PPrPrroocceesssso  c12e  sfsf iinniisshh0e efdd
i
niPsPrheorcdeoscs
 e3s fsin is4he d
finished
Average wtime 114,  rtime 15
```

> - FCFS (First Come First Serve)

For FCFS, because identical processes are being created and FCFS does not allow pre-emption,the extra overhead due to context switches is absent, and therefore, the waiting time is the least.

```
Output

$ schedulertest
Process 2 finished
Process 0 finished
Process 1 finished
Process 3 finished
Process 4 finished
Process 5 finished
Process 7 finished
Process 6 finished
Process 8 finished
Process 9 finished
Average wtime 45,  rtime 37
```

> - LBS (Lottery Based Scheduler)

```
Output

$ schedulertest
Process 9 finished
Process 5 finished
Process 7 finished
Process 6 finished
Process 8 finished
PPPrroocceerossss c1 es3  ffs ii0nn ifisshehdi
enidshP
roecd
essP 4r foincisehesd
s 2 finished
Average wtime 119,  rtime 10
```

> - PBS (Priority Based Scheduler)

The default priority is set to 60, and priorities are not changed in between. Round Robin is applied for processes having equal priorities, which explains the output.

```
Output

$ schedulertest
Process 6 finished
Process 7 finished
Process 5 finished
Process 9 finished
Process 8 finished
PPPrrroocococecesesss ss 2  1f0i n fifsihneindi
sihesd
hePrdoc
ePsrs o3 cfiesnissh ed4
 finished
Average wtime 107,  rtime 18
```

> - MLFQ (Multi Level Feedback Queue)

```


Output

$ schedulertest
Process 5 finished
Process 8 finished
Process 9 finished
Process 6 finished
Process 7 finished
Process 0 finished
Process 1 finished
Process 2 finished
Process 3 finished
Process 4 finished
Average wtime 170,  rtime 18
```

The waiting time for FCFS is the lowest as it doesn't utilise time for choosing the process and directly takes the process having the lowest creation time.

The waiting time of FCFS is lesser than the waiting time of PBS due to pre-emption of PBS which causes CPU to lose valuable time by choosing process again and again.

The waiting time of RR is high due to continuous process change which makes CPU less efficient.

MLFQ has some CPU overhead as it needs to switch between the queues.

In FCFS the latter processes suffer from a higher waiting time, leaving them with a high response time (Convoy effect)

In RR, each process is given a time slice to execute in CPU. The waiting time for each process is the lowest.

In MLFQ, aging is implemented to eradicate starvation, making the waiting time for the process more remarkable than the RR.

In MLFQ, CPU spends a considerable amount of time deciding which process to schedule next, managing queues, upgrading and degrading processes in respective queues, etc

The priority-based mechanism in PBS has reduced waiting time considerably while taking minimum hit in runtime.

Specification 3: Copy-on-write fork

```
+ xv6-riscv/kernel/vm.c ::
  int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
  {
    pte_t *pte;
    uint64 pa, i;
    uint flags;

    for (i = 0; i < sz; i += PGSIZE)
    {
      if ((pte = walk(old, i, 0)) == 0)
        panic("uvmcopy: pte should exist");
      if ((*pte & PTE_V) == 0)
        panic("uvmcopy: page not present");
      pa = PTE2PA(*pte);
      *pte &= ~PTE_W;
      flags = PTE_FLAGS(*pte);
      increse(pa);
      if (mappages(new, i, PGSIZE, (uint64)pa, flags) != 0)
      {
        goto err;
      }
    }
    return 0;

  err:
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;

  int refcnt[PHYSTOP / PGSIZE];
  void freerange(void *pa_start, void *pa_end)
  {
    char *p;
    p = (char *)PGROUNDUP((uint64)pa_start);
    for (; p + PGSIZE <= (char *)pa_end; p += PGSIZE)
    {
      refcnt[(uint64)p / PGSIZE] = 1;
      kfree(p);
    }
  }
  int cowfault(pagetable_t pagetable, uint64 va)
  {
    if (va >= MAXVA)
      return -1;
    pte_t *pte = walk(pagetable, va, 0);
    if (pte == 0)
      return -1;
    if ((*pte & PTE_U) == 0 || (*pte & PTE_V) == 0)
      return -1;
    uint64 pa1 = PTE2PA(*pte);
    uint64 pa2 = (uint64)kalloc();
    if (pa2 == 0){
      return -1;
    }
    memmove((void *)pa2, (void *)pa1, PGSIZE);
    *pte = PA2PTE(pa2) | PTE_U | PTE_V | PTE_W | PTE_X|PTE_R;
    kfree((void *)pa1);
    return 0;
  }

+ xv6-riscv/kernel/trap.c ::

void usertrap(void):
  if (r_scause() == 15)
  {
    if ((cowfault(p->pagetable, r_stval()) )< 0)
    {
      p->killed = 1;
    }
  }

+ xv6-riscv/kernel/kalloc.c

  void * kalloc(void)
  {
    struct run *r;
    acquire(&kmem.lock);
    r = kmem.freelist;
    if (r)
    {
      int pn = (uint64)r / PGSIZE;
      if(refcnt[pn]!=0){
        panic("refcnt kalloc");
      }
      refcnt[pn] = 1;
      kmem.freelist = r->next;
    }
    release(&kmem.lock);
    if (r)
      memset((char *)r, 5, PGSIZE); // fill with junk
    return (void *)r;
  }

  void increse(uint64 pa)
  {
    acquire(&kmem.lock);
    int pn = pa / PGSIZE;
    if(pa>PHYSTOP || refcnt[pn]<1){
      panic("increase ref cnt");
    }
    refcnt[pn]++;
    release(&kmem.lock);
  }

  void kfree(void *pa)
  {
    struct run *r;
    r = (struct run *)pa;
    if (((uint64)pa % PGSIZE) != 0 || (char *)pa < end || (uint64)pa >= PHYSTOP)
      panic("kfree");
    acquire(&kmem.lock);
    int pn = (uint64)r / PGSIZE;
    if (refcnt[pn] < 1)
      panic("kfree panic");
    refcnt[pn] -= 1;
    int tmp = refcnt[pn];
    release(&kmem.lock);
    if (tmp >0)
      return;
    memset(pa, 1, PGSIZE);
    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
  }
```

The idea behind a copy-on-write is that when a parent process creates a child process then both of these processes initially will share the same pages in memory and these shared pages will be marked as copy-on-write which means that if any of these processes will try to modify the shared pages then only a copy of these pages will be created and the modifications will be done on the copy of pages by that process and thus not affecting the other process.

The basic plan in COW fork is for the parent and child to initially share all physical pages, but to map them read-only.

Thus, when the child or parent executes a store instruction, the RISC-V CPU raises a page-fault exception. In response to this exception, the kernel makes a copy of the page that contains the faulted address.

It maps one copy read/write in the child's address space and the other copy read/write in the parent's address space. #### After updating the page tables, the kernel resumes the faulting process at the instruction that caused the fault.

Because the kernel has updated the relevant PTE to allow writes, the faulting instruction will now execute without a fault.

In kalloc.c , we maintain the refcnt for every physical page .In the initialization , the refcnt will be written to 1,because in the freerange ,we call kfree which decreases the refcnt for every pa

with uvmcopy we will not allocate new pages , we increase the refcnt for the pa.

increase refcnt and kfree is a combination , which increases the refcnt of the pa , the other is decrease the refcnt of the pa .In the case when the refcnt of the pa down to zero , we really free the pa

cowfault() handles the invalid va (when va is more than MAXVA , va is not in the page table ,not set user bit or valid bit)

cowfault() also handles allocation a new pa , copying the original content to the new pa , unmapping and mapping for the pte entry