Assignment 4 : Enhancing XV-6

Operating Systems and Networks, Monsoon 2022

xv6 is a simplified operating system developed at MIT. Its main purpose is to explain the main concepts of the operating system by studying an example kernel. xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern RISC-V multiprocessor using ANSI C.

Team:

1. **Nipun Tulsian (2021101055)**
2. **Vyom Goyal (2021101099)**

System calls and their Implementation:

System call 1 : trace

Added the system call trace and an accompanying user program strace . The command will be executed as follows :

```
strace mask command [args]
```

strace runs the specified command until it exits.

It intercepts and records the system calls which are called by a process during its execution and print the following details regarding system call:

1. The process id
2. The name of the system call
3. The decimal value of the arguments
4. The return value of the syscall.

Implementation:

1. Added a new variable trace_mask in struct proc in kernel/proc.h and initialised it to 0 in allocproc() in kernel/proc.c and making sure that child process inherits the mask value from parent process.
2. Added a user function strace.c in user and added entry(”trace”) in user/usys.pl

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

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        fprintf(2, "strace : strace mask command [args]\n");
        exit(1);
    }
    int mask = atoi(argv[1]);
    trace(mask);
    if (exec(argv[2], &argv[2]) == -1)
    {
        fprintf(2, "exec %s failed\n", argv[2]);
    }
    exit(0);
}
```

3. Made necessary changes kernel/syscall.c, kernel/syscall.h and changed certain part of syscall() so that system call which we want to trace is traced :

```
if (num > 0 && num < NELEM(syscalls) && syscalls[num])
{
  // Use num to lookup the system call function for num, call it,
  // and store its return value in p->trapframe->a0
  uint64 arg1 = p->trapframe->a0;
  p->trapframe->a0 = syscalls[num]();
  if (p->trace_mask & 1 << num)
  {
    printf("%d: syscall %s ", p->pid, syscall_name[num]);
    int num_arg = syscall_num_para[num];
    if (num_arg == 0)
    {
      printf("-> ");
    }
    else if (num_arg == 1)
    {
      printf("( %d ) -> ", arg1);
    }
    else if (num_arg == 2)
    {
      printf("( %d %d ) -> ", arg1, p->trapframe->a1);
    }
    else if (num_arg == 3)
    {
      printf("( %d %d %d ) ->", arg1, p->trapframe->a1, p->trapframe->a2);
    }
    printf("%d", p->trapframe->a0);
    printf("\n");
  }
```

4. Implemented a function sys_trace() in kernel/sysproc.c to set trace_mask value given by user

```
uint64
sys_trace(void)
{
  int mask;
  argint(0, &mask);

  myproc()->trace_mask = mask;
  return 0;
}
```

System Call 2 : sigalarm and sigreturn :

A feature that periodically alerts a process as it uses CPU time

Added a new sigalarm(interval, handler) system call. 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.

Added another system call sigreturn() , to reset the process state to before the handler was called.

Implementation:

1. Added new variables in ticks, ticks_after, alarm in struct proc in kernel/proc.h and initialised them to 0 in allocproc() in kernel/proc.c.
2. Implemented a function sys_sigreturn() and sys_sigalarm() in kernel/sysproc.c

```
uint64
sys_sigalarm(void)
{
  int arg1;
  uint64 arg2;
  argint(0, &arg1), argaddr(1, &arg2);
  myproc()->alarm = 1, myproc()->ticks = arg1, myproc()->handler = arg2;
  return 0;
}

uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  p->alarm = 1;
  p->copy->kernel_satp = p->trapframe->kernel_satp;
  p->copy->kernel_hartid = p->trapframe->kernel_hartid;
  p->copy->kernel_sp = p->trapframe->kernel_sp;
  p->copy->kernel_trap = p->trapframe->kernel_trap;

  *(p->trapframe) = *(p->copy);
  return p->trapframe->a0;
}
```

3. Changing values of ticks and ticks_after in usertrap() in kernel/trap.c and changing alarm variable in proc to 0 and reset it to 1 in sigreturn because incase of handler function runs for more time than ticks after which handler should be ran:

```
myproc()->ticks_after++;
if (myproc()->alarm && myproc()->ticks == myproc()->ticks_after)
{
  *(myproc()->copy) = *(myproc()->trapframe);
  myproc()->trapframe->epc = myproc()->handler;
  myproc()->ticks_after = 0;
  myproc()->alarm = 0;
}
```

Scheduling Types and implementation

(By default **Round Robin** is implemented in xv6 with time slice of 1 tick)

1. FCFS (First Come First Service) :

A policy that selects the process with the lowest creation time (creation time refers to the tick number when the process was created). The process will run until it no longer needs CPU time.

1. Added a variable intime in struct proc in kernel/proc.h
2. Initialised intime variable to ticks in allocproc() function in kernel/proc.c
3. Implemented scheduling functionality in scheduler() function in kernel/proc.c, where the runnable process of lowest intime is selected from all processes.
4. Used pre-processor directives to declare the alternate scheduling policy in scheduler() in kernel/proc.c.
5. yield() function in kerneltrap() and usertrap() functions in kernel/trap.c is disabled to disable timer interrupts thus disabling preemption.

```
#ifdef FCFS
    uint64 mini = 446744073709551615;
    struct proc *proc_2 = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      acquire(&p->lock);
      if (p->state == RUNNABLE && p->intime < mini)
      {
        if (proc_2 != 0)
          release(&proc_2->lock);
        mini = p->intime;
        proc_2 = p;
      }
      else
      {
        release(&p->lock);
      }
    }
    if (proc_2 != 0)
    {
      proc_2->state = RUNNING;
      c->proc = proc_2;
      swtch(&c->context, &proc_2->context);
      c->proc = 0;
      release(&proc_2->lock);
    }
#endif
```

2. Lottery Based Scheduler (LBS) :

A preemptive schedule that assigns a time slice to the process randomly in proportion to the number of tickets it owns. That is the probability that the process runs in a given time slice is proportional to the number of tickets owned by it.

1. Implemented a new system call settickets , which sets the number of tickets of calling process. By default each process should get 1 ticket , calling this routine changes the number of tickets . Also the child process inherits the number of tickets from parent process.
2. Added a new variables tickets in struct proc in kernel/proc.h
3. Initialised tickets to 1 in allocproc() in kernel/proc.c and made a function sys_settickets() in kernel/sysproc.c
4. Declared a global variable totaltickets in kernel/proc.c and initialised it to 0 in initproc() in kernel/proc.c and adding tickets of process when it’s state is changing is to **RUNNABLE** and subtracting it when it’s state is changing to **RUNNING**

5. Implemented scheduling functionality in scheduler() function in kernel/proc.c . We call a rand function which selects a golden ticket between 1 and total tickets of runnable process and from this ticket we choose which process range ticket this ticket belong to.

```
#ifdef LBS
    int rand_tickets = (rand() % totaltickets) + 1;
    int tickets_temp = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      acquire(&p->lock);
      // printf("%d %d %d\n",totaltickets,tickets_temp,p->tickets);
      if (p->state == RUNNABLE)
      {
        tickets_temp += p->tickets;
        if (tickets_temp >= rand_tickets)
        {
          // Switch to chosen process.  It is the process's job
          // to release its lock and then reacquire it
          // before jumping back to us.
          p->state = RUNNING;
          c->proc = p;
          totaltickets -= p->tickets;
          swtch(&c->context, &p->context);

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

3. Priority Based Scheduler (PBS):

A non-preemptive priority-based schedules that selects the process with the highest priority for execution. In case two or more processes have the same priority, we use the number of times the process has been scheduled to break the tie. If the tie remains, use the start-time of the process to break the tie(processes with lower start times should be scheduled further)

1. Made a new system call set_priority() and added sys_set_priority() function in kernel/sysproc.c which reschedules the processes if priority of processes increases.
2. Added a new user program setpriority.c in user

3. Added variables priority, runtime , waittime, nsched in struct proc in kernel/proc.h
4. Initialised priority to 60 and runtime, waittime, nsched to 0 in allocproc() in kernel/proc.c
5. yield() function in kerneltrap() and usertrap() functions in kernel/trap.c is disabled to disable timer interrupts thus disabling preemption.
6. Implemented scheduling functionality in scheduler() function in kernel/proc.c, where the runnable process according to algorithm is chose.
7. Used pre-processor directives to declare the alternate scheduling policy in scheduler() in kernel/proc.c.

```
#ifdef PBS
    struct proc *chosen = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      acquire(&p->lock);
      if (p->state == RUNNABLE)
      {
        if (chosen == 0)
        {
          chosen = p;
          continue;
        }
        if (priority_dp(chosen) > priority_dp(p))
        {
          release(&chosen->lock);
          chosen = p;
          continue;
        }
        else if (priority_dp(chosen) == priority_dp(p))
        {
          if (chosen->nsched > p->nsched)
          {
            release(&chosen->lock);
            chosen = p;
            continue;
          }
          else if (chosen->nsched == p->nsched)
          {
            if (chosen->intime > p->intime)
            {
              release(&chosen->lock);
              chosen = p;
              continue;
            }
          }
        }
      }
      release(&p->lock);
    }
    if (chosen != 0)
    {
      chosen->state = RUNNING;
      chosen->nsched++;
      c->proc = chosen;
      chosen->runtime = 0;
      chosen->waittime = 0;
      swtch(&c->context, &chosen->context);
      c->proc = 0;
      release(&chosen->lock);
    }
```

8. To calculate dynamic priority of process in kernel/proc.c

```c
int priority_dp(struct proc *p)
{
  int nice = 5, dp;
  if (p->runtime != 0 || p->waittime != 0)
  {
    nice = ((p->waittime) / (p->waittime + p->runtime)) * 10;
  }

  dp = p->priority - nice + 5 < 100 ? p->priority - nice + 5 : 100;
  dp = dp > 0 ? dp : 0;

  return dp;
}
```

---

4. Multilevel Feedback Queue (MLFQ):

A simplified preemptive MLFQ scheduler that allows processes to move between different priority queues based on their behaviour and CPU bursts.

- If a process uses too much CPU time, it is pushed to a lower priority queue, leaving I/O bound and interactive processes in the higher priority queues.
- To prevent starvation, aging is implemented.

Implementation:

1. In MLFQ implementation we store new arguments in the struct proc that are queue_level, tick_ctr and last_exec.
   - Queue_level argument stores the priority queue number
   - tick_ctr is used for storing runtime of the process to be used in preemption
   - last_exec is used to calculate wait time used in implementing ageing.
2. Also the argument in_time is used in calculating the position in the queue, that is the process with lower in_time is scheduled first and whenever we upgrade or degrade the queue we again set it to the present value of variable ticks.
3. In the scheduler function we select the process that has min queue level and min in time among all processes with that same queue level.

4. In the timer trap we increment the tick_ctr for the currently running process and check if its time slice has been completed. If so it is preempted by calling the yield function.
5. Also if any process reaches the defined wait_time it is upgraded in the queue and if it has a queue_level less than the running process we preempt.
6. Now to implement the point wherein the process voluntarily relinquishes control of the CPU, we update its in_time in the queue when the process wakes up using the wakeup function.
7. Below is the snippet of code that we implemented in scheduler() in kernel/proc.c

```
#ifdef MLFQ
    struct proc *proc_2 = 0;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      acquire(&p->lock);
      if(proc_2==0 && p->state==RUNNABLE) proc_2=p;
      else if (p->state == RUNNABLE && ((p->queue_level == proc_2->queue_level && p->in_time < proc_2->in_time)||p->queue_level<proc_2->queue_level))
      {
        release(&proc_2->lock);
        proc_2 = p;
      }
      else
      {
        release(&p->lock);
      }
    }
    if (proc_2 != 0)
    {
      proc_2->state = RUNNING;
      proc_2->last_exec=ticks;
      c->proc = proc_2;
      swtch(&c->context, &proc_2->context);
      c->proc = 0;
      release(&proc_2->lock);
    }
#endif
```

8. Below is snippet of code that we implemented in usertrap() and kerneltrap() in kernel/trap.c to handle timer interrupt.

```
#ifdef MLFQ
    int flg = 0;
    p->tick_ctr++;
    struct proc *p_sched;
    for (p = proc; p < &proc[NPROC]; p++)
    {
      acquire(&p->lock);
      if (p->state ==RUNNABLE && ticks - p->last_exec > WAIT_TIME)
      {
        if (p->queue_level > 0)
        {
          p->last_exec = ticks;
          p->in_time = ticks;
          p->queue_level--;
        }
      }
      if (p->state == RUNNABLE && p->queue_level < myproc()->queue_level)
      {
        flg = 1;
      }
      release(&p->lock);
    }
    p_sched = myproc();
    int x = p_sched->queue_level;
    int y = 1;
    while (x--)
      y *= 2;
    if (p_sched->tick_ctr == y)
    {
      p_sched->tick_ctr = 0;
      p_sched->in_time = ticks;
      if (p_sched->queue_level < 4)
        p_sched->queue_level++;
      yield();
    }
    else if (flg == 1)
    {
      p_sched->tick_ctr = 0;
      p_sched->in_time = ticks;
      yield();
    }
#endif
```

Copy - On - Write fork :

In this version of xv6 is using RISCV-39 CPU's. In these CPU's the virtual address of the memory location is of 39 bits and rest 25 bits are waste bits. Whenever memory allocation is requested, a memory block of 4096 bytes is allocated to the process and this unit is called a page. This page has a virtual address which is stored in process pagetable in the form of a page table entry. Their are certain flags for each page which denote the permissions that have been granted for the access of that page.

For eg. PTE_V tells us if a page is valid or not, PTE_W tells about write permission etc.

In this specification , when fork is called, it calls the function uvmcopy() of kernel/proc.c to generate the exact copy of parent memory block for child. We change this function so that a copy is not generated and the pages are the pages used by parent are shared. Now to ensure concurrency and prevent faults, we disable writing to this page.

Whenever a write operation is performed on such a page, we detect the trap in kernel/trap.c by using r_scause()==15 which indicates store page fault and create a copy of the page where this process can write by enabling write in new page. The other page remains read only and is deallocated if the number of processes using it become 0. This happens in write_trap function of file kernel/trap.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);
    flags = PTE_FLAGS(*pte);
    if(flags&PTE_W){
      flags &= (~PTE_W);
      flags|=PTE_C;
      *pte = PA2PTE(pa)|flags;
    }
    if(mappages(new, i, PGSIZE, pa, flags) != 0){
       goto err;
     }
    add((void*)pa);
  }
  return 0;

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

```c
int write_trap(void *va, pagetable_t pagetable)
{

  struct proc *p = myproc();
  if ((uint64)va < MAXVA && ((uint64)va < PGROUNDDOWN(p->trapframe->sp) - PGSIZE || (uint64)va > PGROUNDDOWN(p->trapframe->sp)))
  {
    pte_t *pte;
    uint64 pa;
    uint flags;
    va = (void *)PGROUNDDOWN((uint64)va);
    pte = walk(pagetable, (uint64)va, 0);
    if (pte)
    {
      pa = PTE2PA(*pte);
      if(!pa) return -1;
    }
    else
    {
      return -1;
    }
    flags = PTE_FLAGS(*pte);
    if (flags & PTE_C)
    {
      flags = (flags | PTE_W) ;
      flags &= (~PTE_C);
      char *mem;
      mem = kalloc();
      if (!mem)
        return -1;
      memmove(mem, (void *)pa, PGSIZE);
      *pte = PA2PTE(mem) | flags;
      kfree((void *)pa);
      return 0;
    }
    return 0;
  }
  else
  {
    return -1;
  }
}
```

Comparison Between different scheduling mechanism

Scheduler	Avg. Running time	Avg. Waiting time
Round Robin (default)	13	152
FCFS	25	112
LBS	13	145
PBS	13	125
MLFQ	12	150

The above running time and scheduling time are calculated by running user/schedulertest.c on 1 CPU.

MLFQ Scheduling Analysis

Timeline graphs for processes that are being managed by MLFQ Scheduler with ageing time of 30 ticks.