REPORT

SYSTEM CALLS

TRACE

Added the system call trace and a user program strace.

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

execution and print the following details regarding system call in the following format-:

The process id, The name of the system call, (The decimal value of the arguments), The return value of the syscall.

IMPLEMENTATION-:

- Included new variable tracemask in struct proc in proc.h.
- Setting it for any new process entering, to 0 in allocproc() and making sure that child process inherits the mask value from parent process.
- Added a user function strace.c in user and added entry(”trace”) in usys.pl.
- Declared the system call in syscall.h and made necessary changes in syscall.c to trace the syscall.
- Implemented sys_trace to take the value of tracemask as given by user.

SIGALARM & SIGRETURN

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-:

- Included new variables ticks, now_ticks, trapframe_copy ... in struct proc in proc.h.
- Implemented a function sys_sigreturn() and sys_sigalarm() in kernel/sysproc.c.
- Changing values of ticks and now_ticks in usertrap() in 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.

SCHEDULING ALGORITHMS

RR

It is by default implemented in xv6 with a time slice of 1 ticks.

FCFS

NON - PREEMPTIVE

It selects the process which has the least creation time.(i.e in order which they enter the ready queue.

IMPLEMENTATION-:

- Included new variable createtime in struct proc in proc.h.
- Setting createtime for any new process entering, to ticks (global clock) in allocproc().

```
// FCFS
p->createtime = ticks;
```

- To stop preemption, we added a condition on type of algo in usertrap and kerneltrap where the process is getting yield.
- implemented in scheduler by just finding process with minimum createtime.

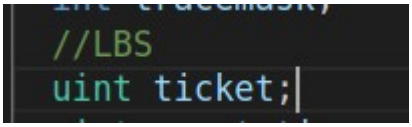
LBS

PREEMPTIVE

The probability of the process to run in each time slice is directly proportional to the number of tickets the process has.

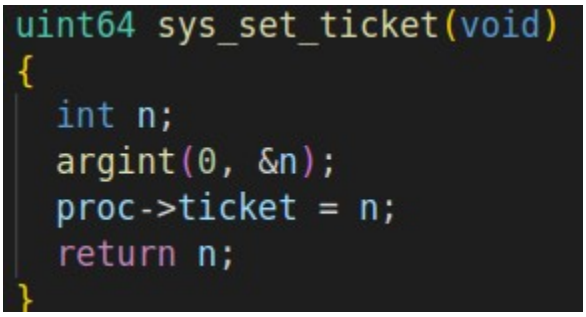
IMPLEMENTATION-:

- Included new variable ticket in struct proc in proc.h.



Setting it for any new process entering, to a default value(1) In allocproc().

- Implementing a syscall ‘sys_set_tickets’ to assign the input ticket given by user to the process. By default, each process should get 1 ticket, calling this routine changes the number of tickets. Also, the child inherits the number of tickets from the parent process.



- Declared variable totaltickets in scheduler function and initialized it to 0 and adding tickets of process and finding min value of sum greater than the random number generated.

PBS

NON - PREEMPTIVE

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)

IMPLEMENTATION-:

- Included new variable priority,runtime,waittime in struct proc in proc.h.
- Setting priority for any new process entering, to a default value(60) and rest to 0 in allocproc().
- Implementing a syscall ‘sys_set_priority’ which reschedules the processes if priority of processes increases.

```
int set_priority(int new, int pid)
{
  // parameters: process id and its new priority
  int old = -1;
  struct proc *p;
  for (p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);
    if (p->pid == pid)
    {
      // set new priority to the process
      // but save old priority
      old = p->priority;
      p->priority = new;
      release(&p->lock);

      // if new priority is lowwer than original, yield
      if (new < old)
        yield();

      break;
    }
    release(&p->lock);
  }

  return old;
}
```

```
uint64
sys_set_priority(void)
{
  // similar
  int priority, pid_;
  int oldpriority = 101;
  argint(0, &priority);
  argint(1, &pid_);
  struct proc *p;

  for (p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);

    if (p->pid == pid_ && priority >= 0 && priority <= 100)
    {
      p->sleepTime = 0;
      p->runTime = 0;
      oldpriority = p->priority;
      p->priority = priority;
    }

    release(&p->lock);
  }
  if (oldpriority > priority)
    yield();

  return oldpriority;
}
```

- Added a new user program setpriority.c in user.

```c
int main(int argc, char *argv[])
{
    // #ifdef PBS
    if( argc < 3 )
    {
        fprintf(2,"insufficient arguments!\n");
        exit(1);
    }

    int given_p = atoi(argv[1]);
    int given_pid = atoi(argv[2]);

    if( given_p<0 || (given_pid > 101) || (given_pid<0))
    {
        fprintf(2,"error: check arguments!\n");
        exit(1);
    }                          int given_p
    int sret = set_priority(given_p,given_pid);

    if( sret == 101 )
    {
        printf("process not found!\n");
        exit(1);
    }
    // #endif

    exit(0);
}
```

- To stop preemption, we added a condition on type of algo in usertrap and kerneltrap where the process is getting yield.
- In the scheduler function, you select the process according to their priority.
- To calculate dynamic priority.

- To calculate the niceness:
  - Record for how many ticks the process was sleeping and running from the last time it was scheduled by the kernel. (see sleep() & wakeup() in kernel/proc.c)
  - New processes start with niceness equal to 5. After scheduling the process, compute the niceness as follows:

$$\text{niceness} = \text{Int}\left(\frac{\text{Ticks spent in (sleeping) state}}{\text{Tick spent in (running + sleeping) state}} * 10\right)$$

- Use Dynamic Priority to schedule processes which is given as:

$$DP = \max(0, \min(SP - \text{niceness} + 5, 100))$$

MLFQ

PREEMPTIVE

It allows processes to move between different priority queues based on their behavior 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 Created a file `queue.c` to implement queue-related functions such as `push()`, `pop()`, `remove()`, `empty()`. `push()` pushes a process onto the specified queue and changes it's state to `QUEUED`, `pop()` returns the next process in queue and pops it from the queue as well as changing the state to `NOTQUEUED`. `remove()` removes the process specified from the queue specified, and makes it `NOTQUEUED`. `empty()` returns whether the specified queue is empty or not.

2. Created an array of queues of size `5` to store the queued processes in MLFQ.

3. Create a function `queuetableinit()` to initialize all structs of the `queuetable` when starting the xv6.

4. The function goes through all the processes in the process array and inserts them into the respective queues if they are `RUNNABLE`, but `NOTQUEUED`.

5. Goes through the `queuetable` to find the process with the lowest priority value, and the first one among them. It picks this process as the next process to run.

6. In `kernel/trap.c`, we only want to preempt the running process if the running time has exceeded `p->queuelevel`. If we need to preempt it, we decrease the priority (by increasing queue level value)

7. To prevent aging, the queueentertime of every process every time mlfqsched is called, to check if any process has been waiting for more than `AGE` ticks (which has been set to 20). if so, the priority of that process is increased (decreasing queue level).

8. To preempt processes when a higher priority process is added to the queue, a function `getpreempted()` checks if the levels above the current processes queuelevel are empty. If any one of them isn't, then we yield in the `usertrap` and `kerneltrap` functions on a timer interrupt.

Possible exploitation of implemented scheduling algorithm

As per the instructions, if a process voluntarily relinquishes control of the CPU before it's time slice is over, on returning the process enters the same queue instead of downgrading in priority.

While this might be logical, and should work for a great percentage of time, there may be people who could be aware of this policy's design, and attempt to exploit it so that their processes get more priority than others, hence getting an unfair share of processing power.

A person who knows how long the time slice is in each priority level, can design a process such that just before the time slice is about to expire, it can ask for a very small I/O request, hence relinquishing itself from the CPU, and rejoining the same priority level very quickly. Hence, it ends up avoiding the downgrade in priority level. Therefore the process can continue to be in the same priority level, taking most of the processor time by this.

For example, say the first priority level has a time slice of 4 ticks. A process can be designed to run for 3 ticks, and then make an I/O request, such as requesting 1 byte from the disk. Hence, the I/O request will be satisfied quickly and it'll be put into the same queue at the end, avoiding the downgrade, and maintaining it's high priority.

One way to counterract this is to not reset the time slice, hence if the process has used 3 ticks, after the I/O request it'll only have one tick left in the priority queue before priority downgrade.

Comparison Between different scheduling mechanism

| Scheduler | Avg. Running time | Avg. Waiting time |

| Round Robin (default) | 13 | 152 |

| FCFS | 25 | 112 |

| LBS | 13 | 150 |

| PBS | 13 | 125 |

| MLFQ | 13 | 148 |

COPY ON WRITE FORK()

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, and the kernel makes a copy of the page that contains the faulted address with both read and write permissions, for both parent and child. 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.

Implementation:

1) When fork() is called, it initially maintains the same pagetable for both the parent and the child process. When uvmcopy() is called in fork(), it sets a COW flag (PTE_C) to the pagetable and removes flag PTE_W, that is the pagetable is in read only mode.

If any of the 2 processes attempts to write to the pagetable, page fault is generated. The error is recognized in usertap() using r_scause() and cow_trap_handler() function is called.

2) In cow_trap_handler(), a new page is created using kalloc() with both read and write permissions, and the data from original pageable is copied into it for both child and parent process.

3) We maintain a cow_reference_count[] array for all processes to record the number of user page tables that refer to that page. This count is incremented when the page is shared by a child due to fork() and decremented when new page is created.

4) We call kfree() to free a page when its page count becomes zero.4

5) Modify copyout() to use the same procedure as usertrap when it encounters a COW page.