OSN xv6 Report

System Calls

strace()

Added `smask` to `struct proc` it is initialized to 0 in `allocproc`. Two additional arrays were needed, one to store the name of the syscall and one to store the number of arguments the syscall takes, these were called `syscall_names` and `syscall_nargs`.

Code Breakdown

For the child process to trace the same syscalls as its parent, fork() is modified.

```
int fork(void) {
  ...
  // child process should get the same mask as the parent process
  // Specification 1
  np->smask = p->smask;
  ...
}
```

To store the arguments of the syscall, an array was used before the syscall was run.

```
int nargs = syscall_nargs[num];
int args[6];
for (int i = 0; i < nargs; i++) {
  argint(i, &args[i]);
}
```

For a syscall to be traced, the corresponding bit should be set in `smask`, if this bit is set, the syscall trace info is printed after the syscall is executed with the following lines.

```
// if trace
int trace_call = p->smask & (1 << num);
if (trace_call) {
  printf("%d: syscall %s (", p->pid, syscall_names[num]);
  for (int i = 0; i < nargs; i++) {
    printf("%d", args[i]);
    if (i != nargs - 1)
      printf(" ");
  }
  printf(") ");
}

// if traced, print return value
if (trace_call) {
  printf("-> %d\n", p->trapframe->a0);
}
```

sigalarm() and sigreturn()

Added syscalls sigalarm and sigreturn that periodically alert a process as it uses CPU time and is used to invoke some action periodically.

The syscall `sigalarm(interval,handler)` calls the handler function periodically after a specific number of CPU ticks specified by the interval argument.

4 additional attributes were added to the `struct proc` - `handler`, `alarmOn`, `interval`, `alarmContext` and `nticks`

Code Breakdown

```
uint64 sys_sigalarm(void)
{
  struct proc *p;
  p = myproc();

  int interval;
  uint64 handler;

  argint(0, &interval);
  argaddr(1, &handler);

  p->interval = interval;
  p->handler = handler;

  return 0;
}
```

The above code was written in kernel/sysproc.c and sets the current processes interval and handler attributes the given argument values

```
if (which_dev == 2 && p->alarmOn == 0)
{
  // this is to check whether the interrupt was from the timer(look at the devintr() funct
ion on line 181)
  p->nticks += 1;

  if (p->nticks == p->interval)
  {
    struct trapframe *context = kalloc();
    memmove(context, p->trapframe, PGSIZE);
    p->alarmContext = context;
    p->alarmOn = 1; // done to prevent reentrance (test 2)
    p->trapframe->epc = p->handler;
  }
}
```

The above code was written in kernel/trap.c and at each timer interrupt which happens at each tick we increase the number of ticks that a current process has run and if that number equal that of the interval we save the register values of the process in the alarmContext attribute and change the program counter to point to the address of the handler function.

The syscall `sigreturn()` resets the process state to that before handler was invoked and this system call must be made before the end of the handler function.

```
uint64 sys_sigreturn(void)
{
  struct proc *p;
  p = myproc();

  memmove(p->trapframe, p->alarmContext, PGSIZE);
  int a0 = p->alarmContext->a0;
  kfree(p->alarmContext);
  p->alarmOn = 0;
  p->nticks = 0;
  p->alarmContext = 0;
  // this is done to restore the original value of the a0 register
  // as sys_sigreturn is also a systemcall its return value will be stored in the a0 regis
ter
  return a0;
}
```

The above code was written in the kernel/sysproc.c file and restores the register values of the process to that before the handler function was called.
The value stored in the a0 register was returned as to ensure the value in a0 register would stay the same as it contains the return value of the previous syscall.

Scheduling

```
First Come First Server - FCFS
```

Non-preemptive, schedules the task with the lowest creation time.

Code Breakdown

To disable preemption, we remove the `yield()` calls in `kerneltrap()` and `usertrap()` in trap.c We add `ctime` to `struct proc` which is set to `ticks` a global variable on birth.

```
static struct proc* allocproc(void) {
  ...
  // Specification 2.
  p->ctime = ticks; // 'ticks' is a global variable in trap.c
  ...
}
```

To select the process with the lowest creation time, the code in `scheduler()` was changed to,

```
struct proc *fp;
for(fp = proc; fp < &proc[NPROC]; fp++) {
  acquire(&fp->lock);
  if (fp->state != RUNNABLE) {
    release(&fp->lock);
    continue;
  }
  if (p == 0) {
    p = fp;
  } else {
    if (p->ctime > fp->ctime)
      p = fp;
  }
  release(&fp->lock);
}
```

To run the selected process, we have the following code, which acquires the process lock, switches context and changes the state of the process to running, this code is common to all schedulers.

```
void scheduler(void) {
  ...
  if (p) {
    acquire(&p->lock);
    if (p->state == RUNNABLE) {
      p->state = RUNNING;
      c->proc = p;
      swtch(&c->context, &p->context);
      c->proc = 0;
    }
    release(&p->lock);
  }
  ...
}
```

```
```

Lottery - LOTTERY

A probabilistic scheduler that assigns a time slice probabilistic-ally weighted to the number of tickets a process owns.

```
#define RAND_MAX 32767
int rand(uint64 seed) {
  seed = seed * 1103515245 + 12345;
  return (unsigned int)(seed/65536) % 32768;
}
```

A random value `[0,RAND_MAX]` is generated and the probability that a particular process is run is represented as an interval `[SV, SV+p]` where `SV` is the start value and `p` is the probability(scaled discretely to `[0,RAND_MAX]` instead of over the interval `[0,1]`). The process that gets selected is the process whose interval captures the random value generated.

Code Breakdown

First we calculate the total tickets in the system to calculate the weighted probability.

```
int tot_tk = 0;
struct proc *fp;
for(fp = proc; fp < &proc[NPROC]; fp++) {
  acquire(&fp->lock);
  if (fp->state == RUNNABLE)
    tot_tk += fp->tickets;
  release(&fp->lock);
}
```

```
int rv = rand((uint64)ticks);
int sv = 0;
for(fp = proc; fp < &proc[NPROC]; fp++) {
  acquire(&fp->lock);
  if (fp->state == RUNNABLE) {
    int prob = (fp->tickets * RAND_MAX) / tot_tk;
    if (sv <= rv && rv < sv + prob) {
      p = fp;
      release(&fp->lock);
      break;
    }
    sv += prob;
  }
  release(&fp->lock);
}
```

```
uint64 sys_settickets(void) {
  struct proc *p;
  int tk;

  argint(0, &tk);
  p = myproc();

  p->tickets += tk;
  return 0;
}
```

Priority Based Scheduling - PBS

Non-preemptive, Schedules task with lowest dynamic priority value, which is a function of Static Priority and niceness, the former's default value upon process creation is 60 but can be changed by using the syscall `set_priority(int priority, int pid)`

```
uint64 sys_set_priority(void){
  struct proc *p;
  int sp,pid;

  argint(0,&sp);
  argint(1,&pid);
  for(p=proc;p<&proc[NPROC];p++){
    acquire(&p->lock);
    if(p->pid==pid){
      int old_sp = p->staticP;
      p->staticP = sp;
      p->niceness = 5;
      p->rtime =0;
      p->stime =0;
      p->wtime =0;
      release(&p->lock);
      if(old_sp > sp){
        yield();
      }
      return old_sp;
    }
    release(&p->lock);
  }
  return -1;
}
```

6 attributes have been added to `struct proc` - `rtime`, `stime`, `wtime`, `niceness`, `staticP`, and `sch_no`

Here we must calculate running time and the sleeping time in order to calculate niceness which is calculated by using the formula

$$Int((sleepingtime)/(sleepingtime + runningtime) * 10)$$

To calculate running time, we add the following code to kernel/trap.c in the usertrap function

```
if (which_dev == 2) {
      struct proc *fp;
      for(fp = proc; fp < &proc[NPROC]; fp++) {
        acquire(&fp->lock);

        if (fp->state == RUNNING) fp->rtime +=1;

        release(&fp->lock);
      }
    }
```

In order to calculate we canot do something similar to that of the running time calculation as multiple CPUs are to be used.

Instead we use the `wakeup()` and `sleep()` functions to set the sleep_time and wake_time and thus calculate sleeping time by subtracting sleep_time from the wake_time.

The following code was added to `scheduler` function in the kernal/proc.c file. We iterate through every process in the proc array and we calculate the niceness and the dynamic priority using the following equation

$$DP = max(0, min(SP - niceness + 5, 100))$$

```
struct proc *fp;
    int min_dp = 103;
    for(fp = proc; fp < &proc[NPROC]; fp++) {
      acquire(&fp->lock);
      if (fp->state == RUNNABLE){
        int sleep = fp->wtime - fp->stime;
        if(!sleep && !fp->rtime){
          fp->niceness = 5;
        }
        else {
          fp->niceness = (int)((sleep/(sleep + fp->rtime)) * 10);
        }
        int dp = max(0,min(fp->staticP - fp->niceness + 5 , 100));
        if(dp < min_dp){
          p = fp;
          min_dp = dp;
        }
        else if(dp == min_dp){
          if(fp->sch_no < p->sch_no){
            p = fp;
          }
          else if(fp->sch_no == p->sch_no){
            if(fp->ctime < p->ctime){
              p = fp;
            }
          }
        }
      }
      release(&fp->lock);
    }
```

Multilevel Feedback Queue - MLFQ

A preemptive queue based scheduler. It assigns priority to a queue and changes priority based on whether a process expires its time slice or if it waits for too long in a particular queue. The data structure for the queues - `struct proc *queue[NPR][NPROC]`.

`rticks` and `wticks` was `struct proc` to measure the number of ticks running and waiting in queue respectively. The current priority of the process is also added through `enum procpr pr`.

Code Breakdown

To enqueue a process, we loop over the `proc` table and enqueue all `RUNNABLE` processes using `queue_proc()` which finds adds the new process at the tail of the queue.

```
void queue_proc(struct proc *p) {
  int i;

  for(i = 0; i < NPROC; i++) {
    struct proc *fp = queue[p->pr][i];
    if (fp && fp->pid == p->pid)
      return;

    if (!fp) {
      queue[p->pr][i] = p;
      break;
    }
  }
}
```

To dequeue a process that is selected to run or isn't `RUNNABLE` we use `dequeue_proc()`. Which moves all processes past the removed process one space back to replace the removed process.

```
void dequeue_proc(struct proc *p) {
  int i, j;

  for(i = 0; i < NPROC; i++) {
    struct proc *fp = queue[p->pr][i];
    if (fp && fp->pid == p->pid) {
      for(j = i + 1; j < NPROC; j++)
        queue[p->pr][j - 1] = queue[p->pr][j];
      break;
    }
  }
}
```

To punish a process that expires its time slice, we reduce its priority but assign it more time ticks to execute.

```
if(which_dev == 2) {
  if (p->rticks == (1 << p->pr)) {
    p->rticks = 0;
    if (p->pr != P4)
      p->pr++;
    yield();
  }
  ...
}
```

To age a process, we first update its waiting time by simply looping over the queue, we do not over count the number of ticks because we assume MLFQ to run with only one CPU.

```
for(i = 0; i < NPR; i++) {
  for(j = 0; j < NPROC; j++) {
    struct proc *fp = queue[i][j];
    if (fp && fp->state == RUNNABLE) {
      fp->wticks++;
    }
  }
}
```

To implement aging, we dequeue an aged process and enqueue it in a higher queue and reset its waiting ticks.

```
for(i = 1; i < NPR; i++) {
  for(j = 0; j < NPROC; j++) {
    fp = queue[i][j];
    if (fp && fp->wticks >= 10) {
      dequeue_proc(fp);
      fp->pr--;
      fp->wticks = 0;
      queue_proc(fp);
    }
  }
}
```

If a higher priority process exists, then the current process is preempted even if it did not finish its time slice, it is however not penalized in this case and is added back to the same queue it was preempted from.

```
if(which_dev == 2) {
  ...
  // check if higher priority process exists
  for(i = 0; i < p->pr; i++) {
    for(j = 0; j < NPROC; j++)
      if (!queue[i][j])
        break;

    if (j) {
      // higher priority process available (don't reset rticks)
      yield();
    }
  }
}
```

We then choose the highest possible priority process available, this implements the round robin implicitly at the last level.

```
p = 0;
for (i = 0; i < NPR; i++) {
  for (j = 0; j < NPROC; j++) {
    fp = queue[i][j];
    if (fp) {
      if (fp->state == RUNNABLE) {
        dequeue_proc(fp);
        p = fp;
        p->wticks = 0;
        goto schedule;
      } else
        dequeue_proc(fp);
    } else
      break;
  }
}
```

The selected process `p` is then run by switching context and changing the state of `p`.

Copy on Write

To implement copy on write, we modify the functions fork uses to prevent the copy of memory and instead map both the processes pagetables to point to the same memory.

To identify when either process tries to write to the memory, we remove write permissions from the mapped memory, then when a process tries to write to it a page fault is raised and then we can detect this and allocate the page properly with write permissions.

This makes knowing when to call `kfree` on a memory block much harder as many processes can refer to the same process, the way we dealt with this is by only freeing the block when no process refers to the block. To do this, we modify `kalloc` and `kfree` and store the number of processes referring to a particular page.

To keep track of references, we index an array. To identify the size and indexing function we use the fact that xv6 assumes only 128mb of memory and that the kernel starts at `KERNBASE` (0x80000000) and ends at `PHYSTOP` (0x88000000). Thus, the index of any page is given by the following function `fn`.

```
#define fn(X) ((uint64)X - KERNBASE) / PGSIZE
```

Code Breakdown

The following struct was added to kalloc.c to keep track of the number of processes referring to a particular page. A spinlock is used to handle concurrency, if two processes call `kfree` on the same page, it is possible to result in a double free and therefore the lock is used.

```
struct {
  struct spinlock lock;
  int fr[fn(PHYSTOP) + 1];
  char lock_kalloc;
} memref;
```

`char lock_kalloc` is added to determine whether `kalloc` should acquire the lock or not to edit `memref`, this is used solely for implementation and dealing with locks and is not related to the workings of cow.

We modify `kfree` to decrement a reference and free the page if the number of references is 0.

```
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
    panic("kfree");

  memref_lock();
  if (memref.fr[fn(pa)] > 1) {
    memref.fr[fn(pa)]--;
    memref_unlock();
    return;
  }

  memref.fr[fn(pa)] = 0;
  memref_unlock();
  ...
}
```

Similarly, `kalloc` was modified to set the number of references to 1.

```
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r) {
    if (memref.lock_kalloc) {
      memref_lock();
      memref.fr[fn(r)] = 1;
      memref_unlock();
    } else {
      memref.fr[fn(r)] = 1;
    }

    memset((char*)r, 5, PGSIZE); // fill with junk
  }
  return (void*)r;
}
```

Auxiliary functions were added to use `memref` in trap.c

```
void            memref_lock_kalloc();
void            memref_unlock_kalloc();
void            memref_lock();
void            memref_unlock();
int             memref_get(void*);
void            memref_set(void*, int);
```

Now to detect a page fault, we check if `r_scause` is `0x0000f` in `usertrap()`. Store faults set `r_scause` to `0x0000f` and the address at which the fault occurred is stored in `r_stval()`. The pagetable and virtual address is sent a function to handle the fault if it is due to a COW page.

```
void usertrap(void) {
  else if (r_scause() == 15) {
    // Specification 3(COW)
    // we now allocate this page to the child process
    // and we then set the PTE_W bit
    uint64 va = r_stval();
    if (handleCOW(p->pagetable, va)) {
      setkilled(p);
    }
  }
}
```

Before we explain what `handleCOW()` does, we will first talk about how a parent and child share the same memory, fork uses `uvmcopy()` to copy its memory to the child, this function is changed to,

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  // char *mem;

  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((mem = kalloc()) == 0)
    //   goto err;
    // memmove(mem, (char*)pa, PGSIZE);
    *pte = (*pte & (~PTE_W)) | PTE_COW;
    flags = (flags & (~PTE_W)) | PTE_COW;
    if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
      // kfree(mem);
      goto err;
    }
    memref_lock();
    int fq = memref_get((void*)pa);
    memref_set((void*)pa, fq + 1);
    memref_unlock();
  }
  return 0;

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

We remove the write bit and set a cow bit in bit 9. Bits 8 and 9 are software reserved bits provided by RISC.

```
*pte = (*pte & (~PTE_W)) | PTE_COW;
flags = (flags & (~PTE_W)) | PTE_COW;
```

We then map the child processes memory to `pa`, the parents page.

```
if(mappages(new, i, PGSIZE, (uint64)pa, flags) != 0){
  // kfree(mem);
  goto err;
}
```

Now, we can explain `handleCOW()`.

```
int handleCOW(pagetable_t pagetable, uint64 va) {
  pte_t *pte;
  uint64 pa;
  uint flags;
  char *mem;

  if (va >= MAXVA)
    return 1;

  va = PGROUNDDOWN(va);
  pte = walk(pagetable, va, 0);
  pa = PTE2PA(*pte);
  flags = PTE_FLAGS(*pte);

  if (va == 0 || (flags & PTE_COW) == 0 || pte == 0 ||
     (flags & PTE_V) == 0 || (flags & PTE_U) == 0)
  {
    return 1;
  }

  memref_lock();
  memref_unlock_kalloc();
  flags = (flags & (~PTE_COW)) | PTE_W;
  int fq = memref_get((void*)pa);
  // printf("Ref: %d\n", fq);
  if (fq == 1) {
    *pte = (*pte & (~PTE_COW)) | PTE_W;
  } else {
    if ((mem = kalloc()) == 0) {
      memref_unlock();
      return 1;
    }

    memmove((void*)mem, (void*)pa, PGSIZE);
    uvmunmap(pagetable, va, 1, 0);
    if (mappages(pagetable, va, PGSIZE, (uint64)mem, flags) != 0) {
      kfree(mem);
      memref_unlock();
      return 1;
    }
    memref_set((void*)pa, fq - 1);
  }
  memref_lock_kalloc();
  memref_unlock();

  return 0;
}
```

We first obtain the PTE associated with the faulting address. We then error check and return 1 if any errors are found or if the mapping wasn't a COW mapping.

We then freeze `memref` and allocate the page newly. If the number of references to the page is 1, then we can simply change the flags of this page.

```
*pte = (*pte & (~PTE_COW)) | PTE_W;
```

Otherwise, we need to allocate it memory and map the old pages to this new memory but with write permissions and removing the COW bit.

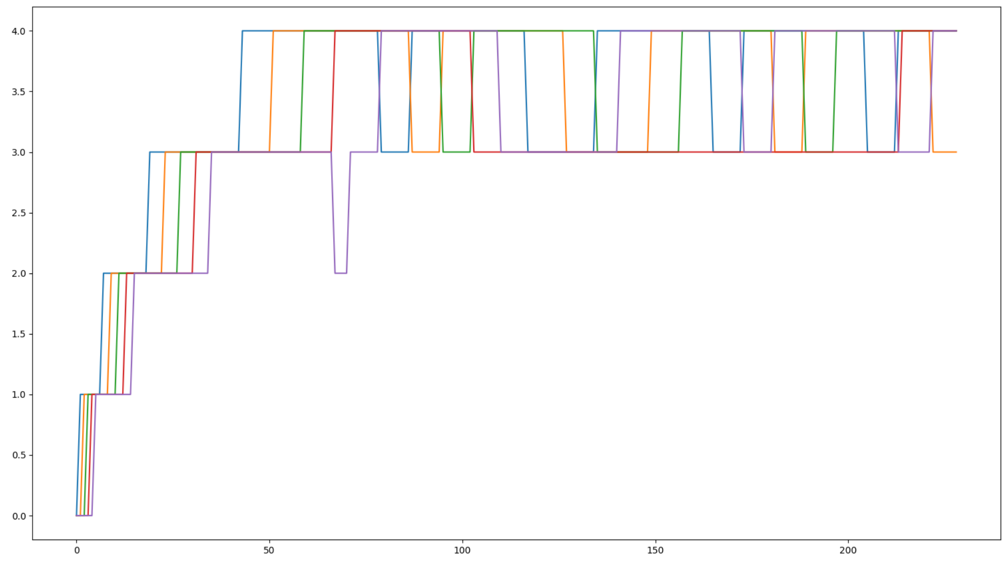
```
flags = (flags & (~PTE_COW)) | PTE_W;
...
{
  if ((mem = kalloc()) == 0) {
    memref_unlock();
    return 1;
  }

  memmove((void*)mem, (void*)pa, PGSIZE);
  uvmunmap(pagetable, va, 1, 0);
  if (mappages(pagetable, va, PGSIZE, (uint64)mem, flags) != 0) {
    kfree(mem);
    memref_unlock();
    return 1;
  }
  ...
}
```

If the number of references wasn’t one, we then decrement the reference count to the old page.

```
memref_set((void*)pa, fq - 1);
```

MLFQ Analysis



Priority vs Ticks