CS3500 : Operating Systems

O(1) Scheduler for xv6

Team Sooners

Viswanath Tadi
CS18B047
G V S Praveen
CS18B017

kernel / proc.h / 106

uint64 priority is added to store the static priority of the process.

Static Priority initialized to **20**

```c
96
97      // these are private to the process, so p->lock need not be
98      uint64 kstack;                 // Bottom of kernel stack for t
99      uint64 sz;                     // Size of process memory (byte
100     pagetable_t pagetable;         // Page table
101     struct trapframe *tf;          // data page for trampoline.S
102     struct context context;        // swtch() here to run process
103     struct file *ofile[NOFILE];    // Open files
104     struct inode *cwd;             // Current directory
105     char name[16];                 // Process name (debugging)
106     uint64 priority;                                   // static priority
107     uint64 dynamic_priority;            // dynamic priority of the
108     struct proc* next;                  // next process pointer in
109     int last_queue;                                    // queue from which
110     uint64 running_since;
111     uint64 sleeping_since;
112     uint64 total_runtime;
113     uint64 total_sleeptime;
```

uint64 priority is added to store the static priority of the process.

Static Priority initialized to **20**

kernel / proc.h / 107

```
96
97    // these are private to the process, so p->lock need not be
98    uint64 kstack;               // Bottom of kernel stack for t
99    uint64 sz;                   // Size of process memory (byte
100   pagetable_t pagetable;       // Page table
101   struct trapframe *tf;        // data page for trampoline.S
102   struct context context;      // swtch() here to run process
103   struct file *ofile[NOFILE];  // Open files
104   struct inode *cwd;           // Current directory
105   char name[16];               // Process name (debugging)
106   uint64 priority;                         // static priority
107   uint64 dynamic_priority;        // dynamic priority of the
108   struct proc* next;              // next process pointer in
109   int last_queue;                          // queue from which
110   uint64 running_since;
111   uint64 sleeping_since;
112   uint64 total_runtime;
113   uint64 total_sleeptime;
```

uint64 dynamic_priority is added to store the dynamic priority of the process

kernel / sysproc.c

System call sys_setpriority() allows a process to change its own static priority.

Meant for use in testing programs

System call sys_getpriority() allows a process to reads its own dynamic priority.

Meant for use in testing programs

```c

uint64
sys_setpriority(void)
{
        int n;
  if(argint(0, &n) < 0)
    return -1;
  struct proc* p = myproc();
  if(n>39 || n<0)return -1;
  p->priority=n;
  return 0;
}

uint64
sys_getpriority(void)
{
        struct proc* p = myproc();
        int n;
        n = p->dynamic_priority;
        return n;
}

```

kernel / exec.c

Added support for **nice** command from shell.

$ my_prog -nice 10

- Check if second last argument is **"-nice"**
- Convert last argument to integer
- Update priority of process
- Remove the last two arguments from argv before passing on to the process.

```c
    if(argcval > 2 && strlen(argv[argcval-2])==5 && argv[argcval-2][0]=='-' && argv[argcval-2][1]=='n'
     && argv[argcval-2][2]=='i' && argv[argcval-2][3]=='c' && argv[argcval-2][4]=='e')
    {
        if(strlen(argv[argcval-1])<=2)
        {
            if(strlen(argv[argcval-1])==2 && argv[argcval-1][0]>='0' && argv[argcval-1][0]<='9' &&
            argv[argcval-1][1]>='0' && argv[argcval-1][1]<='9')
            {
                    int val=0;
                    val+=(argv[argcval-1][0]-'0') * 10;
                    val+=(argv[argcval-1][1]-'0');
                    if(val<=39 && val>=0)
                    p->priority=val;
            }
            else if(strlen(argv[argcval-1])==1 && argv[argcval-1][0]>='0' && argv[argcval-1][0]<='9')
            {
                    int val=0;
                    val+=(argv[argcval-1][0]-'0');
                    if(val<=39 && val>=0)
                    p->priority=val;
            }
        }
        argcval-=2;
        argv[argcval] = 0;
    }
```

kernel / sched.c

struct proc* q contains two arrays of 40 pointers each. Two arrays act as active and passive queues interchangeably. 40 entries denote the heads of 40 priority queues.

struct proc* qlast contains two arrays of 40 pointers each. 40 entries denote the tails of 40 priority queues.

Denotes which of q[0] and q[1] is active

Necessary spinlocks for the above structures. Help synchronize access across cores.

```c
//Linked list implementation

struct sched_queue
{
        struct proc* q[2][40];
        struct proc* qlast[2][40];
        int sched_active;
        struct spinlock lock[2][40];
        struct spinlock lock_active;
};
```

Initialize active and passive queues to empty.

Initialize q[0] as the active queue by setting sched_active to 0

```c
void
sched_init()
{
        int i;
        for(i=0;i<40;i++)
        {
                queue.q[0][i]=queue.q[1][i]=queue.qlast[0][i]=queue.qlast[1][i]=0;
        }
        queue.sched_active = 0;
}
```

kernel / main.c / 31

sched_init() called from main.c before calling scheduler.

sched_init() initializes the active and passive queues to all empty queues.

```c
    binit();         // buffer cache
    iinit();         // inode cache
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    sched_init();        // initializing scheduler structures
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
  } else {
    while(started == 0)
      ;
    __sync_synchronize();
    printf("hart %d starting\n", cpuid());
```

kernel / sched.c / sched_insert()

Dynamic priority of a process is calculated just before it is inserted into a queue.

Bonus = (sleeptime*11)/(sleeptime + runtime)

Dynamic priority = Static_priority - bonus + 5

Dynamic priority in the struct proc is updated just before insertion.

```c
void
sched_insert(struct proc* curp,int active)
{
        if(active!=0 && active!=1)panic("active value out of bounds");
        curp->next = 0;
        int bonus = 5;
        if(!(curp->total_sleeptime==0 && curp->total_runtime==0)){
            bonus = ((curp->total_sleeptime*11)/(curp->total_sleeptime+curp->total_runtime));
        }
        int dprio = curp->priority - bonus + 5;
        if(dprio <= 0) dprio = 0;
        if(dprio >= 39) dprio = 39;
        curp->dynamic_priority = dprio;
        int p = curp->dynamic_priority;
```

⋮

kernel / sched.c / sched_insert()

If active is 1, process is inserted into currently active queue. If active is 0, process is inserted into the queue which was passive when it was allocated to the CPU. Index denotes the queue into which insertion takes place.

Here, p is the dynamic priority.
Process curp is inserted into queue denoted by index and priority queue denoted by p.
Insertion is done at the end of the queue in O(1) time bounds.

Necessary locking is present in place to ensure synchronization.

```c
        int index;

        acquire(&queue.lock_active);
        if(active == 1)
        {
                index = queue.sched_active;
        }
        else if (active == 0)
        {
                index = 1 - curp->last_queue;
        }
        acquire(&queue.lock[index][p]);
        if(queue.q[index][p]==0)
        {
                queue.q[index][p]=curp;
                queue.qlast[index][p]=curp;
        }
        else
        {
                queue.qlast[index][p]->next=curp;
                queue.qlast[index][p]=curp;
        }
        release(&queue.lock_active);
        release(&queue.lock[index][p]);
}
```

kernel / sched.c / sched_get()

Iterate through all the active queues in decreasing order of priority to find the first non-empty queue.

If found,remove the first process from queue and return it.

If not found,make the other queue as active and vice versa by changing sched_active.

```c
struct proc*
sched_get()
{
        int i;
        struct proc* p;
        sched_L:
        acquire(&queue.lock_active);
        for(i=0;i<40;i++)
        {
                acquire(&queue.lock[queue.sched_active][i]);
                if(queue.q[queue.sched_active][i]!=0)
                {
                        p = queue.q[queue.sched_active][i];
                        p->last_queue = queue.sched_active;
                        queue.q[queue.sched_active][i] = queue.q[queue.sched_active][i]->next;
                        if(p->next==0)
                        {
                                queue.qlast[queue.sched_active][i]=0;
                        }
                        release(&queue.lock[queue.sched_active][i]);
                        release(&queue.lock_active);
                        return p;
                }
                release(&queue.lock[queue.sched_active][i]);
        }
        queue.sched_active = 1 - queue.sched_active;
```

⋮

kernel / sched.c / sched_get()

Iterate through all the new active queues in decreasing order of priority to find the first non-empty queue.

If found,remove the first process from queue and return it.

If not found,run the whole function again with the other queue as active.

```c
	for(i=0;i<40;i++)
	{
		acquire(&queue.lock[queue.sched_active][i]);
		if(queue.q[queue.sched_active][i]!=0)
		{
			p = queue.q[queue.sched_active][i];
			p->last_queue = queue.sched_active;
			queue.q[queue.sched_active][i] = queue.q[queue.sched_active][i]->next;
			if(p->next==0)
			{
				queue.qlast[queue.sched_active][i]=0;
			}
			release(&queue.lock[queue.sched_active][i]);
			release(&queue.lock_active);
			return p;
		}
		release(&queue.lock[queue.sched_active][i]);
	}

	queue.sched_active = 1 - queue.sched_active;
	release(&queue.lock_active);
	goto sched_L;
}
```

kernel / proc.c / allocproc()

When the process is created, following values are initialized

- static priority to 20.
- total_runtime to 0.
- total_sleeptime to 0.

```c
  p->pagetable = proc_pagetable(p);

  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof p->context);
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  // Make static priority = 5
  p->priority = 20;
  p->total_runtime = 0;
  p->total_sleeptime = 0;

  return p;
}
```

kernel / proc.c / userinit()

When the first user process becomes RUNNABLE, insert it into the active queue.

```c
  p->tf->sp = PGSIZE;  // user stack pointer

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;

  sched_insert(p,1);

  release(&p->lock);
}

// Grow or shrink user memory by n bytes.
```

kernel / proc.c / fork()

When the child process becomes RUNNABLE, insert it into the active queue.

```c
  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  np->state = RUNNABLE;

  sched_insert(np,1);

  release(&np->lock);

  return pid;
}

// Pass p's abandoned children to init.
```

kernel / proc.c / scheduler()

Find the next process to schedule using sched_get().

Calculate time-slice using the dynamic priority of the process.

Configure timer interrupt by updating CLINT_MTIMECMP to CLINT_MTIME + timeslice

Update p->running_since to CLINT_MTIME since process is set to RUNNING state.

Switch to process's context.

```c
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

        // get next process
        p = sched_get();

        // timeslice calculation
        int timeslice;
        if(p->dynamic_priority < 20)
                timeslice = (40-p->dynamic_priority)*500000;
        else
                timeslice = (40-p->dynamic_priority)*250000;

        *(uint64*)CLINT_MTIMECMP(cpuid()) = *(uint64*)CLINT_MTIME + timeslice;

        acquire(&p->lock);

        // Switch to chosen process.  It is the process's job
    // to release its lock and then reacquire it
    // before jumping back to us.
    // printf("%s,%d\n",p->name,p->dynamic_priority);
    p->running_since = *(uint64*)CLINT_MTIME;
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->scheduler, &p->context);
```

kernel / proc.c / yield()

Calculate run time using current MTIME and p->running_since, and add it to p->total_runtime.

Insert it into the passive queue since it’s state is now RUNNABLE.

```c
// Give up the CPU for one scheduling round.
void
yield(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  p->total_runtime += *(uint64*)CLINT_MTIME - p->running_since;
  //printf("%s,%d,%d\n",p->name,*(uint64*)CLINT_MTIME,p->running_since);
  p->state = RUNNABLE;
  sched_insert(p,0);
  sched();
  release(&p->lock);
}
```

kernel / proc.c / exit()

Calculate run time using current MTIME and p->running_since, and add it to p->total_runtime.

```c
  // Parent might be sleeping in wait().
  wakeup1(original_parent);

  p->total_runtime += *(uint64*)CLINT_MTIME - p->running_since;
  p->xstate = status;
  p->state = ZOMBIE;

  release(&original_parent->lock);

  // Jump into the scheduler, never to return.
  sched();
  panic("zombie exit");
}

```

kernel / proc.c / sleep()

Set p->sleeping_since to current MTIME as state is changed to SLEEPING.

Calculate run time using current MTIME and p->running_since, and add it to p->total_runtime.

```c
    if(lk != &p->lock){  //DOC: sleeplock0
      acquire(&p->lock);  //DOC: sleeplock1
      release(lk);
    }

    // Go to sleep.
    p->total_runtime += *(uint64*)CLINT_MTIME - p->running_since;
    p->sleeping_since = *(uint64*)CLINT_MTIME;
    p->chan = chan;
    p->state = SLEEPING;

    sched();

    // Tidy up
```

kernel / proc.c / wakeup()

Calculate sleep time using current MTIME and p->sleeping_since, and add it to p->total_sleeptime.

Since the process becomes RUNNABLE, insert it into the active queue.

Similar changes made in wakeup1.

```c
void
wakeup(void *chan)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == SLEEPING && p->chan == chan) {
      p->total_sleeptime += *(uint64*)CLINT_MTIME - p->sleeping_since;
      p->state = RUNNABLE;
      sched_insert(p,1);
    }
    release(&p->lock);
  }
}

```

kernel / proc.c / kill()

Calculate sleep time using current MTIME and p->sleeping_since, and add it to p->total_sleeptime.

Since the process becomes RUNNABLE, insert it into the active queue.

```
      acquire(&p->lock);
      if(p->pid == pid){
        p->killed = 1;
        if(p->state == SLEEPING){
          // Wake process from sleep().
          p->total_sleeptime += *(uint64*)CLINT_MTIME - p->sleeping_since;
          p->state = RUNNABLE;
          sched_insert(p,1);
        }
        release(&p->lock);
        return 0;
      }
      release(&p->lock);
    }
    return -1;
```