README.md

ASSIGNMENT 5 OS

Run

To run xv6 type :

1. RR

```
$ make clean
$ make qemu SCHEDULER=RR

OR

$ make clean
$ make qemu
```

2. FCFS

```
$ make clean
$ make qemu SCHEDULER=FCFS
```

3. PBS

```
$ make clean
$ make qemu SCHEDULER=PBS
```

4. MLFQ

```
$ make clean
$ make qemu SCHEDULER=MLFQ
```

IMPLEMENTATION

waitx

```
int waitx(int *wtime, int *rtime)
{
       struct proc *p;
       int havekids, pid;
       struct proc *curproc = myproc();

       acquire(&ptable.lock);
       for (;;)
       {
              // Scan through table looking for exited children.
              havekids = 0;
              for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
              {
```

```
                    if (p->parent != curproc)
                          continue;
                    havekids = 1;
                    if (p->state == ZOMBIE)
                    {
                          // Found one.
                          *rtime = p->rtime;
                          *wtime = p->wtime;
                          pid = p->pid;
                          kfree(p->kstack);
                          p->kstack = 0;
                          freevm(p->pgdir);
#ifdef MLFQ
                          rem(p, p->queue);
#endif
                          p->pid = 0;
                          p->parent = 0;
                          p->name[0] = 0;
                          p->killed = 0;
                          p->state = UNUSED;
                          release(&ptable.lock);
                          return pid;
                    }
              }

              // No point waiting if we don't have any children.
              if (!havekids || curproc->killed)
              {
                    release(&ptable.lock);
                    return -1;
              }

              // Wait for children to exit.  (See wakeup1 call in proc_exit.)
              sleep(curproc, &ptable.lock); //DOC: wait-sleep
        }
}
```

- the value of rtime and wtime were set to be equal to the run time and wait time of the process .
- in MLFQ we remove the process from the queue in which it is present (similarly in wait system call)

NOTE :

- rtime of a running process is updated on each tick
- wtime of a runnable process is updated on each tick

ps

It prints the information about all processes on the terminal . It calls system call **print_pinfo** for this . ps and print_pinfo codes respectively :

```
int main()
{
   int a = fork();

   if (a == 0) //child
   {
       print_pinfo();
   }
   else //parent
   {
       wait();
   }

   exit();
}
```

```
int print_pinfo()
{
	struct proc *p;
	int ret = -1;
	char *states[] = {
		[UNUSED] "unused",
		[EMBRYO] "embryo",
		[SLEEPING] "sleeping",
		[RUNNABLE] "runnable",
		[RUNNING] "running\t",
		[ZOMBIE] "zombie"};
	cprintf("PID\tPriority\tState\t\tr_time\tw_time\tn_run\tcur_q\tq0\tq1\tq2\tq3\tq4\n\n");

	for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
	{
		if (p->pid != 0)
		{
			cprintf("%d%s%d%s\t%s%s%d%s%d%s%d%s%d%s%d%s%d%s%d%s%d%s\n", p->pid, a, p->priority, a,
states[p->state], a, p->rtime, a, p->w_time, a, p->num_run, a, p->queue, a, p->ticks[0], a, p->ticks[1], a,
p->ticks[2], a, p->ticks[3], a, p->ticks[4]);
		}
	}
	ret = 1;
	return ret;
}
```

- Note ps may not output immediately in FCFS and PBS (if priority is low in comparison to other processes)

SCHEDULING ALGORITHMS

1. FCFS :

```
for (;;)
	{
		struct proc *alottedP = 0;

		// Enable interrupts on this processor.
		sti();
		// Loop over process table looking for process to run.
		acquire(&ptable.lock);
		struct proc *minctimeProc = 0;
		for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
		{
			if (p->state == RUNNABLE)
			{
				if (minctimeProc)
				{
					if (p->ctime < minctimeProc->ctime)
						minctimeProc = p;
				}
				else
				{
					minctimeProc = p;
				}
			}
			else
			{
				continue;
			}
		}
		if (minctimeProc != 0 && minctimeProc->state == RUNNABLE)
		{
			alottedP = minctimeProc;
			c->proc = alottedP;
			switchuvm(alottedP);
```

```
                alottedP->num_run++;
                alottedP->state = RUNNING;
                alottedP->w_time = 0;

                swtch(&(c->scheduler), alottedP->context);
                switchkvm();

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

- In FCFS process with smaller `ctime` is run first .
- Preemption is not allowed.
- In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed unless it *sleeps* .
- Therefore just checked the ptable for the process with smallest ctime and run it .

2. PBS :

```
for (;;)
    {
        struct proc *alottedP = 0;

        // Enable interrupts on this processor.
        sti();
        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        int minPrio = 101;
        for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        {
            if (p->state == RUNNABLE && p->priority <= (minPrio - 1))
            {
                minPrio = p->priority;
            }
            else
            {
                continue;
            }
        }
        struct proc *p, *p2;
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        {
            if (minPrio == 101)
            {
                break;
            }
            if (p->state != RUNNABLE)
            {
                continue;
            }
            else if (p->state == RUNNABLE)
            {
                if (p->priority == minPrio)
                {
                    alottedP = p;
                    // Switch to chosen process.  It is the process's job
                    // to release ptable.lock and then reacquire it
                    // before jumping back to us.
                    c->proc = alottedP;

                    switchuvm(alottedP);
                    alottedP->num_run++;
                    alottedP->state = RUNNING;
                    p->w_time = 0;
```

```
                                // cprintf("[PBSCHEDULER] pid %d on cpu %d (prio %d)\n",
                                //         alottedP->pid, c->apicid, alottedP->priority);
                                swtch(&(c->scheduler), alottedP->context);

                                switchkvm();
                                // Processis done running for now.
                                // It should have changed its p->state before coming back.
                                // else it has been interrupted and so we will exit this for loop as scheduler has
called
                                // it again and the next process will be scheduled accordingly
                                c->proc = 0;

                                int minPrio2 = 101;
                                for (p2 = ptable.proc; p2 < &ptable.proc[NPROC]; p2++)
                                {
                                       if (p2->state == RUNNABLE && p2->priority <= (minPrio2 - 1))
                                       {
                                              minPrio2 = p2->priority;
                                       }
                                       else
                                       {
                                              continue;
                                       }
                                }

                                if (minPrio2 <= (minPrio - 1))
                                {
                                       break;
                                }
                         }
                  }
            }
            release(&ptable.lock);
      }
```

- PBS is a preemptive.
- Each process is assigned a priority. Process with highest priority (numerically least) is run first and so on.
- Processes with same priority are executed in a round robin fashion.
- If a process of higher priority (numerically less) arrives while a lower priority process is being executed the lower priority process is preempted.
- Initially on process allocation default priority is `60`
- Therefore we search for process with minimum priority in the ptable and then run it .

3. MLFQ

```
for (;;)
      {
            // struct proc *alottedP = 0;

            // Enable interrupts on this processor.
            sti();
            // Loop over process table looking for process to run.
            acquire(&ptable.lock);
            struct proc *p = 0;

            // int oof = 0;
            for (int i = 0; i <= 4; i++)
            {
                  if (queue_end[i] > -1)
                  {
                        p = queue[i][0];
                        rem(p, i);
                        break;
                  }
                  else
                  {
                        continue;
                  }
```

```
            }

            if (p != 0 && p->state == RUNNABLE)
            {
                  p->num_run++;
                  c->proc = p;

                  switchuvm(p);
                  p->state = RUNNING;
                  p->w_time = 0;
                  swtch(&c->scheduler, p->context);
                  switchkvm();
                  c->proc = 0;

                  if (p != 0 && p->state == RUNNABLE)
                  {
                        if (p->change_queue != 0)
                        {
                              p->change_queue = 0;
                              if (p->queue <= 3)
                              {
                                    p->w_time = 0;
                                    p->queue++;
                              }
                              p->curr_ticks = 0;
                        }
                        else if (p->change_queue == 0)
                        {
                              p->curr_ticks = 0;
                        }
                        push(p, p->queue);
                  }
            }
            release(&ptable.lock);
      }
```

- In MLFQ processes are initially assigned the 0th queue when they are allocated.
- It allows a process to move between queues. If a process uses too much CPU time, it will be moved to a lower-priority queue. Similarly if a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- Aging prevents starvation of a process .
- This algorithm makes system highly responsive
- `AGE` (max age) is set to 80 , after this a process moves to higher priority queue .
- w_time (wait time which gets reset) is set to 0 every time queue changes or the process goes `RUNNING` state
- rem and push are functions to delete and insert into queue respectively

`set_priority` system call

```
int set_priority(int pid, int priority)
{
      struct proc *p;
      int intrpt = 0;
      int prevp = 0;
      if (priority > 100)
            return 1;
      else
      {
            for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
            {
                  if (p->pid == pid)
                  {
                        intrpt = 0;
                        acquire(&ptable.lock);
                        prevp = p->priority;
                        p->priority = priority;
```

```
                        if (prevp > p->priority)
                        {
                              intrpt = 1;
                        }
                        else
                        {
                              intrpt = 0;
                        }
                        release(&ptable.lock);
                        break;
                  }
            }
            if (intrpt == 0)
            {
                  return prevp;
            }
            else if (intrpt == 1)
            {
                  yield();
                  return prevp;
            }
      }
      return prevp;
}
```

- setPriority is user process that can be typed on terminal , it calls set_priority to set the priority of a process
- if new priority is > 100 then we return else we change the priority and follow next 2 points
- if new priority is > than old priority (numerically) then we return the old priority
- if new priority is < than old priority (numerically) then we yield and return the old priority

INTERRUPTS

```
      if (myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0 + IRQ_TIMER)
      {
#ifdef MLFQ
            if (myproc()->curr_ticks > (timeslice[myproc()->queue] - 1))
            {
                  change_queue_flag(myproc()); //timeslice finished
                  myproc()->w_time = 0;
                  yield();
            }
#endif

#if defined(DEFAULT) || defined(RR) || defined(PBS)
            yield();
#endif


if (cpuid() == 0)
            {
                  acquire(&tickslock);
                  ticks++;
                  func();
                  calc_wait();
                  wakeup(&ticks);
                  release(&tickslock);
            }
```

- if curr_ticks (number of ticks in the present queue) is equal to the timeslice of the queue then we then we call `change_queue_flag(myproc())` to change flag to imply queue has to be changed now which is then handled in MLFQ scheduling as shown in code above and then we yield where we call the scheduler .
- if scheduling algo is RR or PBS then we yield on timer
- `func` and `calc_wait` are functions to update rtime , wtime , w_time and ticks in each queue for the process .