Dawit Woldegiorgis
COP 4600
Section: 0467
September 10, 2015
Project: Part #3

Honesty Pledge

I affirm that this project submission is solely the product of my own efforts and that I determined the answers from the information I found in the OpenBSD 3.5 source code. Excepting my lecture notes and the textbook, I neither sought nor obtained information about schedulers from any other source.

SIGN: 

DATE: November 8, 2015

Dawit Woldegiorgis
COP 4600
Section: 0467
September 10, 2015
Project: Part #3

Learning experience summary

General stuff:

- The one thing I was happy about learning, although I stomped my feet and crossed my arms at the thought of learning it initially, was using the **grep** command, along with **find**. It was only then that made the statement "grep is your friend", which I had seem countless number of times before, seem meaningful.
- Another thing I learnt was that comments in code are extremely important for other users. No matter how nifty the variable names are, it doesn't come close to good comments when it comes to trying to understand what's happening in a segment of code. Although I was aware of this before, I did not consider it too much when writing code, because I always assume that if I can understand what I'm writing, so can anyone else reading my code – which is silly.
- Keeping documentation updated is important. Why have a documentation if it's going to be misleading?
- Operating systems are not pure magic. It's a compilation of several nifty codes working together to produce something. I "knew" this before, of course, but it didn't really register until I looked at the source code. Oh I don't know, I still feel like its magic even after looking at the source code…

Topic specific:

- Learn the basics of how the scheduler in OpenBSD works, and how it interacts with other components to produce an efficient mechanism.
- Figure out, with more depth, the interaction between a parent and child process.
- Determine the algorithms and tricks used by the scheduler – like roundrobin, decay, reducing context switch overhead, etc.

Dawit Woldegiorgis
COP 4600
Section: 0467
September 10, 2015
Project: Part #3

OpenBSD 3.5 scheduler analysis

Introduction:

The operating system is responsible for managing the process that are running on the system. This manager is called the scheduler. The schedulers task is, then, to determine which process get's to run at a certain time, for how long this process is to run, and with what frequency is this process going to get selected. As a result, the scheduler has to deal with the complex process of managing processes by implementing fairness, while still maintaining a hierarchy of "social status" among the many processes that exist.

In OpenBSD, the hierarchy of process status is determined by what is known as the **process priority**. This process priority, as is evident form the name, determines how frequently the process is to be picked by the scheduler. The higher the priority, the more likely it is to be picked by the scheduler. But, the implementation of the priority is actually the reverse. A process with low number priority is deemed to be more likely to be picked by the scheduler than is a process with high priority number. This is akin to a ranking system in the Olympics, for example, where a lower number indicates higher achievement and thus associated with higher status. But, what is more important than process priority is the run queue in which the process currently resides. A good example is the basic social classes that were used in historical (and some current) societies – the royal family, the military, the merchants, the peasants, etc. These different classes contain a population with different status. The royal family will have a king, which has a higher status than a prince would have, yet both are in the royal class, which has a higher priority compared to the peasants – which might also consist of a father farmer and a son farmer, with the father having more priority over the son, but both having the least priority in the society. Similarly, the processes in a run queue might have different priorities (within 4 priority levels), and any process with a priority level greater than 4 is surely to be located in a different run queue that might be at a higher or lower level.

Dawit Woldegiorgis
COP 4600
Section: 0467
September 10, 2015
Project: Part #3

In this table, for example, a darker shaded cell indicates a higher priority than the corresponding lighter colored cell. The numbers also correspond to the priority levels, in addition to color.

Run queue 1:			
Priority 0	Priority 1	Priority 2	Priority 3
Run queue 2:			
Priority 4	Priority 5	Priority 6	Priority 7
Run queue 3:			
Priority 8	Priority 9	Priority 10	Priority 11
…			
Run queue 32:			
Priority 124	Priority 125	Priority 126	Priority 127

The scheduler uses the priority to determine on which run queue to work on at a time, ignoring all run queues with lower priorities, until all processes in said queue are completed.

One can quickly see that this might lead to starvation of the lower priority queues if processes keep running in the high priority queues. To circumvent such situation, a decaying mechanism is employed where the priority of a process deteriorates the more CPU time it uses. Therefore, if a process A, for example, has a priority of 20, and uses too much CPU time, it might get "demoted" to priority 40 (remember, higher priority level = lower status / less likely to be rescheduled). This way, high priority processes that hog up the CPU will get demoted, and lower priority processes will be more likely to be scheduled in the future.

Dawit Woldegiorgis
COP 4600
Section: 0467
September 10, 2015
Project: Part #3

Questions & Answers:

1) In theory, how many priority levels does OpenBSD support—i.e., what does the documentation say? In actuality, how many are supported—i.e., what does the source code say? How is this accomplished?

   OpenBSD supports 41 priority levels, according to the definition, as the integer range [-20, 20] defines the total possible priorities available. Therefore, each digit should theoretically correspond to a single priority levels.
   But, in the source code – **parm.h** and **sched.h**– declare/imply that the priorities range from 0-127 (total 128 possible priorities). Furthermore, these priorities don't dictate the levels, but rather the NQS (defined in **proc.h**) suggests that there are 32 run queues, and each run queue has 4 priorities (PPQ = 128/32). Therefore, the source code suggest there are 32 priority levels, with each 4 priority levels. As indicated in the comments in **param.h** "differences less than 4" of priority levels are insignificant.

   man GETPRIORITY(2)

   ```
   prio is a value in the range -20 to 20.  The default
   priority is 0; lower priorities cause more favorable
   scheduling.
   ```

   man NICE(1)

   ```
   The priority can be adjusted over a range of -20 (the
   highest) to 20 (the lowest).
   ```

   resource.h

```
/*
 * Process priority specifications to get/setpriority.
 */
#define PRIO_MIN        -20
#define PRIO_MAX        20
```

param.h

```
/*
 * Priorities.  Note that with 32 run queues, differences
less than 4 are
 * insignificant.
 */
...

#define MAXPRI  127     /* Priorities range from 0 through
MAXPRI. */
```

sched.h

```
#define PPQ     (128 / NQS)             /* priorities per
queue */
```

proc.h

```
#define NQS     32                  /* 32 run queues. */
```

Dawit Woldegiorgis
COP 4600
Section: 0467
September 10, 2015
Project: Part #3

2) What algorithm does OpenBSD use to choose between runnable processes with equal priorities?

OpenBSD uses round robin as the algorithm of choice, to switch between processes with equal priorities. This is evident in the source code:

kern_synch.c

```
/*
 * Force switch among equal priority processes every
100ms.
 */
/* ARGSUSED */
void
roundrobin(arg)
        void *arg;
{
        struct timeout *to = (struct timeout *)arg;
        struct proc *p = curproc;
        int s;

        if (p != NULL) {
                s = splstatclock();
                if (p->p_schedflags & PSCHED_SEENRR) {
                        /*
                         * The process has already been
through a roundrobin
                         * without switching and may be
hogging the CPU.
```

```
                        * Indicate that the process
should yield.
                        */
                       p->p_schedflags |=
PSCHED_SHOULDYIELD;
                } else {
                        p->p_schedflags |= PSCHED_SEENRR;
                }
                splx(s);
        }
        need_resched();
        timeout_add(to, hz / 10);
}
```

3) Does OpenBSD's scheduler prevent starvation? If so, how?

The OpenBSD scheduler tries to prevent starvation by "recomput[ing] process priorities, every hz ticks" and thus determining which process is to be favored in future runs. The method employed to compute the priorities involve decaying of priority as it runs, favoring the process that has not yet run recently.

kern_synch.c

```
/*
 * Recompute process priorities, every hz ticks.
 */
/* ARGSUSED */
void
```

Dawit Woldegiorgis
COP 4600
Section: 0467
September 10, 2015
Project: Part #3

```
schedcpu(arg)
        void *arg;
{
   ...
   p->p_cpticks = 0;
   newcpu = (u_int) decay_cpu(loadfac, p->p_estcpu);
   p->p_estcpu = newcpu;
   resetpriority(p);
   ...
}

...

/*
 * Compute the priority of a process when running in user
mode.
 * Arrange to reschedule if the resulting priority is
better
 * than that of the current process.
 */
void
resetpriority(p)
        register struct proc *p;
{
   ...
}

/*
 * We adjust the priority of the current process.  The
 * priority of a process gets worse as it accumulates CPU
```

```
 * time.  The cpu usage estimator (p_estcpu) is increased
 * here.  The formula for computing priorities (in
 * kern_synch.c) will compute a different value each time
 * p_estcpu increases. This can cause a switch, but unless
the
 * priority crosses a PPQ boundary the actual queue will
not
 * change.  The cpu usage estimator ramps up quite quickly
 * when the process is running (linearly), and decays away
 * exponentially, at a rate which is proportionally slower
 * when the system is busy.  The basic principle is that
the
 * system will 90% forget that the process used a lot of
CPU
 * time in 5 * loadav seconds.  This causes the system to
 * favor processes which haven't run much recently, and to
 * round-robin among other processes.
 */
```

4) How long is the OpenBSD quantum?

The round robin algorithm forces the processes with equal priorities to switch every 100ms, which is the quantum given to a process.

kern_synch.c

```
/*
 * Force switch among equal priority processes every
100ms.
```

```
 */
/* ARGSUSED */
void
roundrobin(arg)
   ...
```

5) Why would the procedure `remrunqueue` ever be used?

The definitions for `remrunqueue` procedure indicates that it is useful to remove a process from the running queue. In the schedcpu() function, the `remrunqueue` procedure is used when the process is already in the running queue but its priority needs to be changed. To change the priority, the process is first removed from the running queue, its priority updated, and put back on the running queue as seen in the code below:

kern_synch.c

```
/*
 * Recompute process priorities, every hz ticks.
 */
/* ARGSUSED */
void
schedcpu(arg)
        void *arg;
{
   ...

     if (p->p_priority >= PUSER) {
```

```
          if ((p != curproc) && p->p_stat == SRUN &&
          (p->p_flag & P_INMEM) &&
               (p->p_priority / PPQ) != (p->p_usrpri /
          PPQ))
               {
                    remrunqueue(p);
                    p->p_priority = p->p_usrpri;
                    setrunqueue(p);
               } else
                    p->p_priority = p->p_usrpri;
               }
   ...
}
```

i386/locore.s

```
/*
 * setrunqueue(struct proc *p);
 * Insert a process on the appropriate queue.  Should be
called at splclock().
 */
NENTRY(setrunqueue)
          ...


/*
 * remrunqueue(struct proc *p);
 * Remove a process from its queue.  Should be called at
splclock().
```

```
 */
NENTRY(remrunqueue)
            ...
```

6) Consider a context switch occurring when the only runnable process is the same as the one that was just running. Is there any difference in how that case is handled vis-a-vis when there is a different runnable processes available? Explain.

   Since the context switch handler (scheduler) doesn't know ahead of time if the process is the only one that's currently running, it tries to attempt a context switch. During this process, if it realizes, when comparing the "new" process with the current process, that the current process is to be rescheduled, it stops the context save, and jumps to the instruction that restores the process on the stack.

   Call hierarchy*: kern_synch.c* [preempt() → mi_switch() → cpu_switch()] → *locore.s* [switch_search: → switch_return:]

kern_synch.c

```
    /*
    * General preemption call.  Puts the current process
    back
    * on its run queue and performs an involuntary
    context
    * switch.  If a process is supplied, we switch to
    that
    * process.  Otherwise, we use the normal process
    selection
    * criteria.
```

```
*/
void
preempt(newp)
        struct proc *newp;
{
     ...
     mi_switch();
     ...
}
/*
 * Must be called at splstatclock() or higher.
 */
void
mi_switch()
{
     ...
     cpu_switch(p);
     ...
}
```

i386/locore.s

```
/*
 * cpu_switch(void);
 * Find a runnable process and switch to it.  Wait if
necessary.  If the new
 * process is the same as the old one, we short-
circuit the context save and
 * restore.
```

```
 */
ENTRY(cpu_switch)
...
switch_search:
        /*
         * First phase: find new process.

          ...

          /* Skip context switch if same process. */
          cmpl    %edi,%esi
          je      switch_return
...

switch_return:
   /*
    * Restore old cpl from stack.  Note that this is
    * always an increase, due to the spl0() on entry.
    */
```

man CTXSW(9)

```
The mi_switch() function implements the machine-
independent prelude to a process context switch.

...
```

```
cpu_switch() will make a choice amongst the processes
which are ready to run from a priority queue data-
structure.  The priority queue consists of array qs[NQS]
of queue header structures each of which identifies a list
of runnable processes of equal priority (see
<sys/proc.h>).  A single word whichqs containing a bit
mask identifying non-empty queues assists in selecting a
process quickly.  cpu_switch() must remove the first
process from the list on the queue with the highest
priority (lower indices in qs indicate higher priority),
and assign the address of its process structure to the
global variable curproc.  If no processes are available on
the run queues, cpu_switch() shall go into an ``idle''
loop. The idle loop must allow interrupts to be taken that
will eventually cause processes to appear again on the run
queues.  The variable curproc should be NULL while
cpu_switch() waits for this to happen.
```

7) When a parent forks off a child, what is the relationship of the child's priority to the parents?

The child will have the same priority as the parent upon forking.

kern_fork.c

```
/*
 * set priority of child to be that of parent
```

```
 * XXX should move p_estcpu into the region of struct proc
 * which gets copied.
 */
   scheduler_fork_hook(p1, p2);
```

8) Is a parent process's "schedulability" affected by how much CPU time its children have used? If so, how? [Note: the answer is *not* that the CPU time must now be allocated among more processes, therefore the parent will be scheduled less frequently.]

The schedulabiliy of the parent is not affected by how much time the children have used, because the parent process and the child processes have separate estimated cpu usage variables that keep track of how much CPU time they used when running. During forking, the child inherits the schedule history of the parent, but beyond that each should be treated as independent processes when it comes to schedulabiity. When parent is waiting on the child, the parent's *estcpu* is changed to the sum of the parent and the children *estcpu*, although this happens when the process reaper is in action.

sched.h

```
/* Inherit the parent's scheduler history */

static __inline void
scheduler_fork_hook(parent, child)
        struct proc *parent, *child;
{
        child->p_estcpu = parent->p_estcpu;
}
```

```
/* Chargeback parents for the sins of their children.  */

static __inline void
scheduler_wait_hook(parent, child)
        struct proc *parent, *child;
{
        /* XXX just return if parent == init?? */

        parent->p_estcpu = ESTCPULIM(parent->p_estcpu +
child->p_estcpu);
}
```

9) Most of the scheduling code is written in C. What's a plausible reason for why some of it might be written in assembly?

The code in the machine dependent section of the code (arch/) folder contains all the assembly language. The reason why this might be the case is likely to take into account the differences in architecture, and therefore difference in how instructions are carried out when interacting with the hardware. Therefore, to make sure that the operations described in the C code are possible in these different hardware architectures, it would make sense to write the instructions with a language that is “native” to that specific hardware structure, ultimately ensuring compatibility.