Design Overview

Team: W's Only

- **Kevin Wang** *(kwang43)* - design, case switching, splatter scheduling
- **Malcolm Neill** *(mneill)* - design, rng, splatter scheduling
- **Nicolette Miller** *(nmiller2)* - design, priority queue
- **Edmund Yu** *(eyu9)* - design, benchmark, benchmark analysis

Scheduling Cases

Case 1: ULE scheduler and FIFO queues.

Base implementation of the FreeBSD system. No changes needed. Note that the original code must be preserved and any changes will require if/else statements to check if a thread is user or kernel, as well as what scheduling case is active.

`schedcase` = 1

Case 2: ULE scheduler with priority queues.

ULE scheduling assigns the process threads to their respective run queues. The threads will be inserted into the FIFO queues by priority (simulating a priority queue).

`schedcase` = 2

Case 3: Splatter scheduler and FIFO queues.

User process threads will be assigned to a random FIFO run queue.

`schedcase` = 3

Case 4: Splatter scheduler and priority queues.

User process threads will be assigned to a random run queue. The threads will be inserted into the FIFO queues by priority (simulating a priority queue).

`schedcase` = 4

Design

Switching Cases

In order to be able to easily implement all four cases, we will use a global value `schedcase`. This static int value can be updated using the FreeBSD sysctl(9). This will allow us to switch between scheduling cases during runtime while the kernel is loaded.

Ignore Kernel Threads

Note that a kernel thread has priority value `td->td_priority` of 0 to 47 and 80 to 119 -- inclusive. We will use a boolean value `isKernel` which will say if a priority falls within the boundaries.

Random Number Generator

In order to generate a random priority, we will utilize random(9) to get a number between 0 and 255. Because we don't want to send non kernel threads to interrupt and kernel run queues, we will adjust them.

The adjustment won't be completely fair and certain priority values will be called more often than others. To combat this, we will seed the randomizer with the current system time in order to make it more random.

Assigning a Random Run Queue

If `schedcase` is equal to 3 or 4, we will use a random priority provided by the RNG function. When assigning a random run queue, FreeBSD typically uses `td_priority` however we can set the buffer value `pri` to a random one.

For kernel threads and other cases, the buffer value `pri` will still be equal to `td->td_priority`.

Assigning of queues will be the same for all cases, with varying results due to `pri`.

Priority Queues

If `schedcase` is equal to 2 or 4, we will add a thread to its assigned queue, ordered by priority. While the run queue will still use FreeBSD's FIFO, it will simulate a priority queue. Because we will be inserting items throughout the run queue, we will be using linked-lists. While we should use heaps -- O(lg n) -- and implement an extractMin for choosing a thread to run, linked-lists -- O(n) is probably easier with the library TAILQ procedures.

If a run queue is empty, we will insert the thread at the head. Otherwise, we will cycle through the run queue and compare our threads actual priority against the temporary selected run queue thread's priority.

When comparing, if our thread has a lower priority (bigger value), we will either continue to the next item or -- if at the end of the run queue -- insert it after. If our thread has a higher priority (smaller value), we will insert it before the temporarily selected thread.

Benchmark

The benchmark program used for testing our kernel will consist of three parts.

The first is a forkbomb that creates many child processes to consume CPU time. It recursively calls fork() and prints which iteration it is in. Each iteration runs a small calculating function to ensure that the compiler does not optimize it away.

The second function is a stress test that tests how the scheduler assigns a large calculation task. Trying to calculate large Fibonnaci numbers without the use of dynamic programming requires memory management and handling many smaller tasks.

The final part is a simple multithreading test that iterates though a loop and prints out the current threadID. `<thread>` is a C++ 11 header file, which required an update on both the program and Makefile. Using threads helps further test the scheduler.

We will also use the FreeBSD time command to capture run time.

- `real` is the total elapsed time.
- `user` is the amount of CPU time spent in user mode.
- `sys` is the amount of CPU time spent in kernel mode.

Kernel Modifications

Data

kern_switch.c

```
static int schedcase    // used to determine which kernel case is being used
```

Functions

kern_switch.c

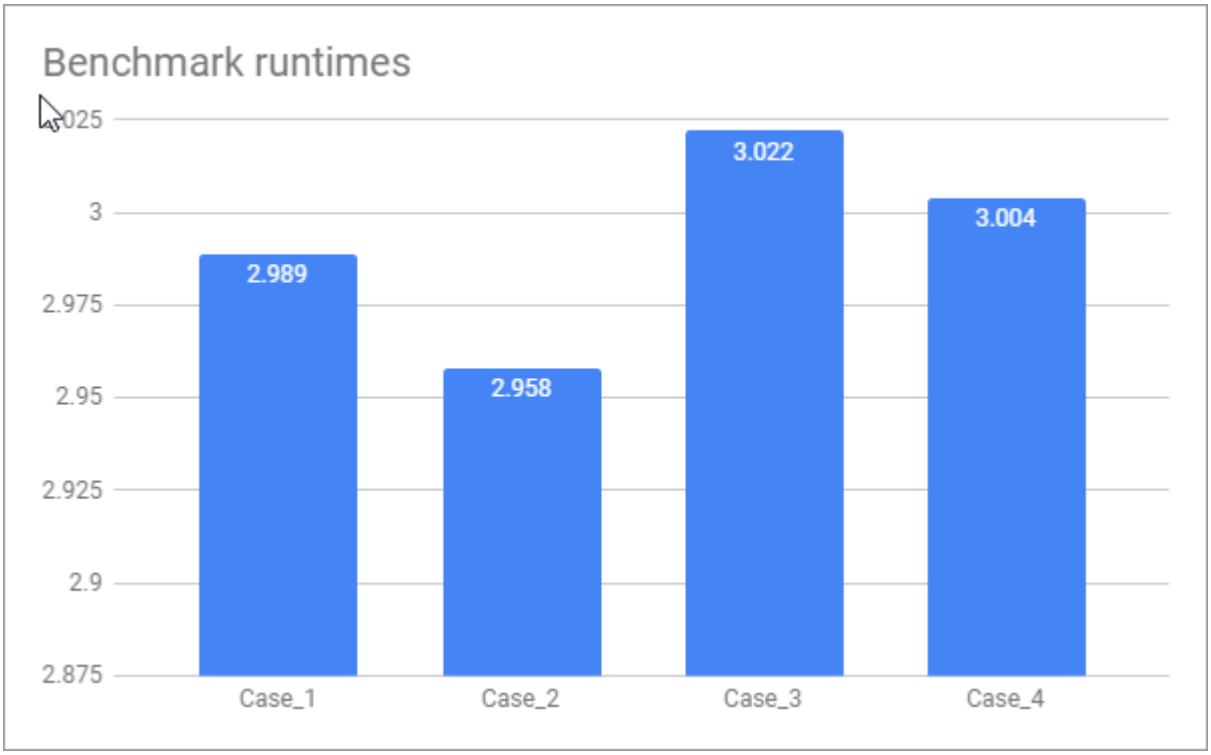
```
/* This function is used to generate a random priority value */

int
getRandom(void) {
  use system time to set generator seed
  get random value between 0 and 255
  adjust to avoid kernel and interrupt queues

  return random value
}
```

```
/* This function is used to insert threads into an ordered run queue */

void
runq_priority_queue(struct rqhead *rqh, struct thread *td, int flags)
{
  if (run queue is empty) {
    insert thread at head
  } else {
    for each temp thread in runqueue {
      if (temp thread priority <= td priority) {
        if (temp thread is last thread in queue)
          insert td after  temp thread
        else
          continue through queue
      }
      else
        insert the thread before temp thread
  }
}
```

```
/* Modified FreeBSD function for assigning threads to a run queue by priority */

void
runq_add(struct runq *rq, struct thread *td, int flags) {
  check if a kernel thread, set isKernel

  if (!isKernel and schedcase == 3 or 4)
    set pri to random priority
  else if (isKernel or schedcase == 1 or 2)
    set pri to actual thread priority

  use pri value to set run queue for all cases

  if (!isKernel and schedcase == 2 or 4)
    send to priority queue insertion procedure
  else if (isKernel or schedcase == 1 or 3)
    inserts into queue normally
}
```

```
/* Modified FreeBSD function for assigning threads to a run queue by priority */

void
runq_add_pri(struct runq *rq, struct thread *td, u_char pri, int flags) {
  check if a kernel thread, set isKernel

  if (!isKernel and schedcase == 3 or 4)
    set pri to random priority
  // actual thread priority is already set in arg[2]

  use pri value to set run queue for all cases

  if (!isKernel and schedcase == 2 or 4)
    send to priority queue insertion procedure
  else if (isKernel or schedcase == 1 or 3)
    inserts into queue normally
}
```

Analysis



The graph shows the average runtimes in seconds after running the benchmark program on each case. Each benchmark ran 20 times for each of the 4 cases. The results from the benchmark tests did not vary as much as expected, which makes sense if the program stays the same. The slight variations between the cases also confirm beliefs of splatter scheduling decreasing performance and priority queue increasing performance. The benchmark also shows no page faults or swaps happening in any of the tests. The tests could be improved by higher scaling to show bigger differences in the cases, or by adding more functions that respond more to the changes in the scheduler.

Sources

- "Design and Implementation of the FreeBSD Operating Systems"
- Piazza
- http://www.leidinger.net/FreeBSD/dox/kern/html/
- https://wiki.freebsd.org/AndriyGapon/AvgThreadPriorityRanges
- https://www.freebsd.org/doc/en/books/handbook/kernelconfig.html
- https://www.freebsd.org/doc/handbook/kernelconfig/building.html