Assignment 2: Splatter Scheduling
Authors: Robert Sato, Keerthi Krishnan, Pranav Nampoothiri
Cruz ID: rssato, kvkrishn, pnampoot
Due Date: 05/05/19

Contributions:

Robert: As the majority of this assignment was focused on understanding and modifying existing kernel code, we all worked together to understand and plan our implementation of the assignment.

Contributions: I implemented the FIFO to priority runq portion of the assignment. I created the benchmark code (not including spookMark) and all of the Makefiles. These Makefiles handle file transfer, kernel building, kernel compiling, benchmarking and output redirection. I also compiled, ran and benchmarked all of the FreeBSD code we implemented.

Ratings: Our entire group did a great job meeting up consistently and working proactively throughout the time period we had to work on this assignment. Unfortunately, with the combinations of schedules not matching up and difficulties with FreeBSD and git, I was the only one with a working VirtualBox throughout this assignment. Therefore, I had to handle the testing and implementations of our assignment making it difficult for my teammates to compile and test their own changes. Despite this, my teammates did a great job helping plan and implement our changes.

Keerthi: I contributed by helping with some benchmarks, figuring out the pseudocode for both priority queues and splatter scheduling, as well as creating the graphs for the benchmark data and the design document. With the group, I worked on writing out the pseudocode for both priority queues and splatter scheduling. I also helped in trying to print out runqueue statistics for the splatter scheduling and attempting to implement splatter scheduling. Otherwise, I worked on creating some benchmarks, particularly with page faults and helped reformat and debug some benchmarks when needed. I worked primarily on using the data and analyzing as well as putting it in graphs with the 4 cases. I also helped with the design document, making sure everything was conveyed clearly and adding thoughts when needed.

Ratings: Our group did a great job of meeting up proactively and making sure that we were on top of our assignment. Unfortunately, for some of our group, our FreeBSD and git was not cooperating, therefore, we would commit to git and one person would have to implement and test it on their FreeBSD, which was frustrating, but we all worked together to make sure that we planned and implemented our changes correctly. Overall, as a group, we worked as a team and collaborated with each other well to get this assignment successfully done.

Pranav: My contributions involved working on some benchmarks, particularly assisting in evaluating what performance statistics we would make benchmarks over, and how we would use the getrusage system call to write our functions. I assisted in structuring the gathered data we got from the benchmarks by calculating benchmark averages, so we would be able to make the visual representations/bar graphs of our data analysis. I wrote a significant portion of the Design Document, with assistance from my teammates.
With the coding portions of the assignment, I assisted in figuring out the logic and flow of our priority queue implementation as well as splatter scheduling, as well as providing assistance in writing out pseudocode.
I also formalized our descriptions and analysis of our benchmarks, by writing thorough captions of our visuals, all of which we laid out in our Design Document.

Ratings: Considering our decently conflicting schedules over the past 2 weeks, we all did a great job getting together, being proactive in making progress with the assignment, and helping each other understand the concepts, readings, and the assignment. Our team captain, Robert, did a great job leading the way, initiating the discussions, and handling the significant implementations of this project, including the Makefile. Me and our other group mate, Keerthi, had rather persistent setup issues, so in our efforts to not take too much time away from the actual project work, our captain had to take care of all the testing on FreeBSD as a result. Keerthi did a great job creating our bar graphs, assisting with the Design Document, writing and analyzing our benchmarks, and writing pseudocode for our implementations. This was an overall successful effort, and a good team experience.

Data Structure Selection Idea:

Our initial idea for choosing the data structure to implement priority queues, was using heaps, because inserting into a list and Heapifying to get a thread priority in the right order within the priority queue, would take O(logn). But, building the heap would take O(n), and both are necessary in order for the implementation to work. Thus priority queue implementation with heaps, would take O(nlogn).

We decided to use linked lists instead because for priority queues, inserting into linked lists to have the threads in order of decreasing priority number allows us to insert and delete nodes in worst-case O(n); this is more time efficient than the heap implementation, since linked lists don't require extra time to be constructed.

Priority Queues Low-Level Scheduler:

We decided to implement priority queues in functions, runq_add and runq_add_pri, from the source file, kern_switch.c, because priority queue implementation is the alternative to the FIFO queue implementation, both of which are variants to the FreeBSD low-level scheduler. Adding our priority queues here make sense, because these are the core functions where threads are assigned to queues, the data structure that manages scheduling.

To implement the priority queues, we account for the ranges of user-level threads, between the ranges 48-79 and 224-255 for real-time and idle thread priorities respectively. Threads with priorities between the range of 120-223 are set for timesharing threads. If the runq is empty, we add a thread at the head of the TailQ. Otherwise, we traverse through the list by setting the current element in the list, marked by a "temp" variable, to the next element of the list (this is defined in the loop structure definition, before the code actually inside of the loop).

Within the loop, a next variable, which was assigned to the variable of the current element in the list that we are looking at (this assignment was done in the loop structure definition), is initialized with a queue FreeBSD library function, TAILQ_NEXT. Then, if the priority value of the thread at the current element of the list, is greater than that of the thread that we
are trying to insert (in other words, the thread that we are trying to insert has more priority than the thread at the current element of the list), we insert our thread before the thread at the current element of the list, using the queue FreeBSD library function, TAILQ_INSERT_BEFORE. Then break out of the loop to fetch the next thread from scheduler.

If the next element of the list is null, then we use the queue FreeBSD library function, TAILQ_INSERT_TAIL, to insert our thread at the end of the tail queue. Then break out of the loop to fetch the next thread from scheduler.

Otherwise, if the thread in the next element of the list has a priority value greater than that of the thread that we are trying to insert in the list (again, in other words, the thread that we are trying to insert has more priority than the thread in the next element of the list), we insert our thread after the thread in the CURRENT element of the list, using the queue FreeBSD library function, TAILQ_INSERT_AFTER. Once again, we break out of the loop to fetch the next thread from scheduler.

Splatter Scheduling High-Level Scheduler:

Since splatter scheduling involves RANDOMLY assigning a process/thread to a run queue, we decided to use the random number generator defined in sched_ule.c, since this would randomize priorities, as we were hoping to achieve. This is because, we want to use the random numbers that we get from this generator, to represent priority values of threads; in our attempts to establish this representation, we would be able to avoid being bounded by a specific ordering of priority for threads; thus we would be able to assign our threads to a run queue randomly.

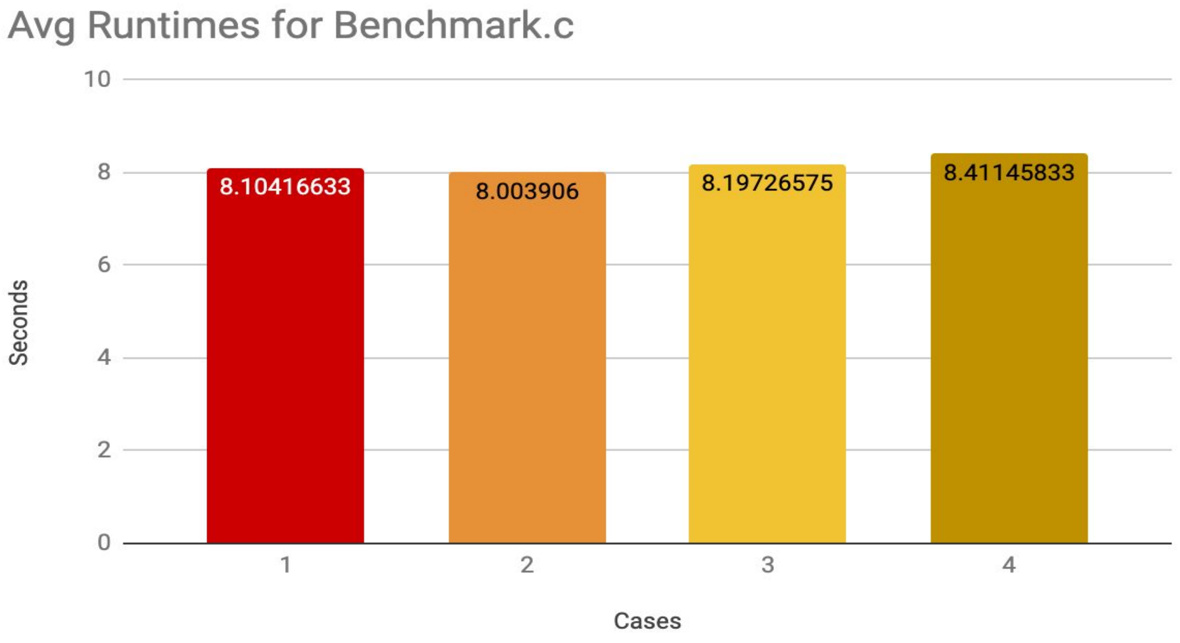
We implemented splatter scheduling within the sched_ule.c function, tdq_runq_add. We decided to do this because tdq_runq_add adds a thread to the runq based on the priority value of the thread; essentially, we realized tdq_runq_add was the function where priority scheduling (high-level scheduler) was implemented. Therefore, to implement splatter scheduling, we called the random number generator in this function to randomly assign priority to threads. We used the unsigned character variable pri, which represents the priority value of the thread; here we assigned it the random number generator, sched_random, that was defined in sched_ule.c source file. Since the return value of the random number generator conflicted with the pri variable type, we casted it an unsigned character value.

We also decided to mod the value given by the random number generator by the variable RQ_NQS, which holds the number of queues in the runqueue. This was done in order to maintain the pri value within the priority range given. The pri variable will now give a random priority value to the thread, causing it to be assigned to a runqueue randomly, thus achieving splatter scheduling.

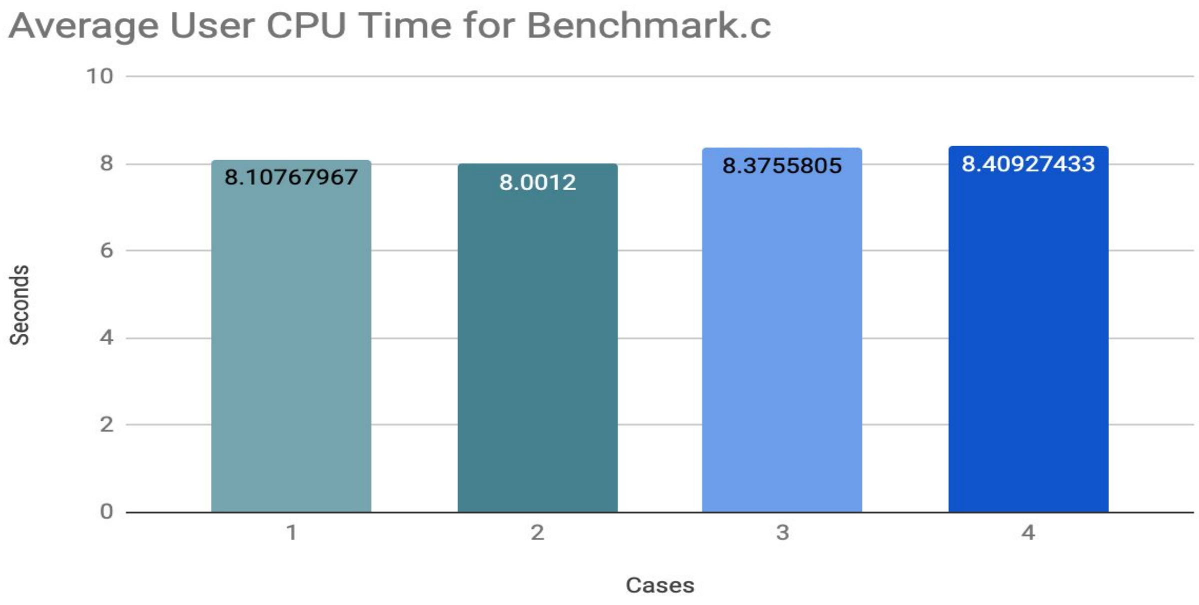
Benchmarks:

We created a directory, called benchmarks in order to test our codes and analyze the behavior. Within the directory, contains two files, benchmark.c and spookMark.c which we used for testing our 4 cases of the FreeBSD scheduler. The benchmark.c file contained CPU-intensive functions, while spookMark.c contained functions highly reliant on I/O. We gathered data and averages of the data, and created the bar graphs shown below:

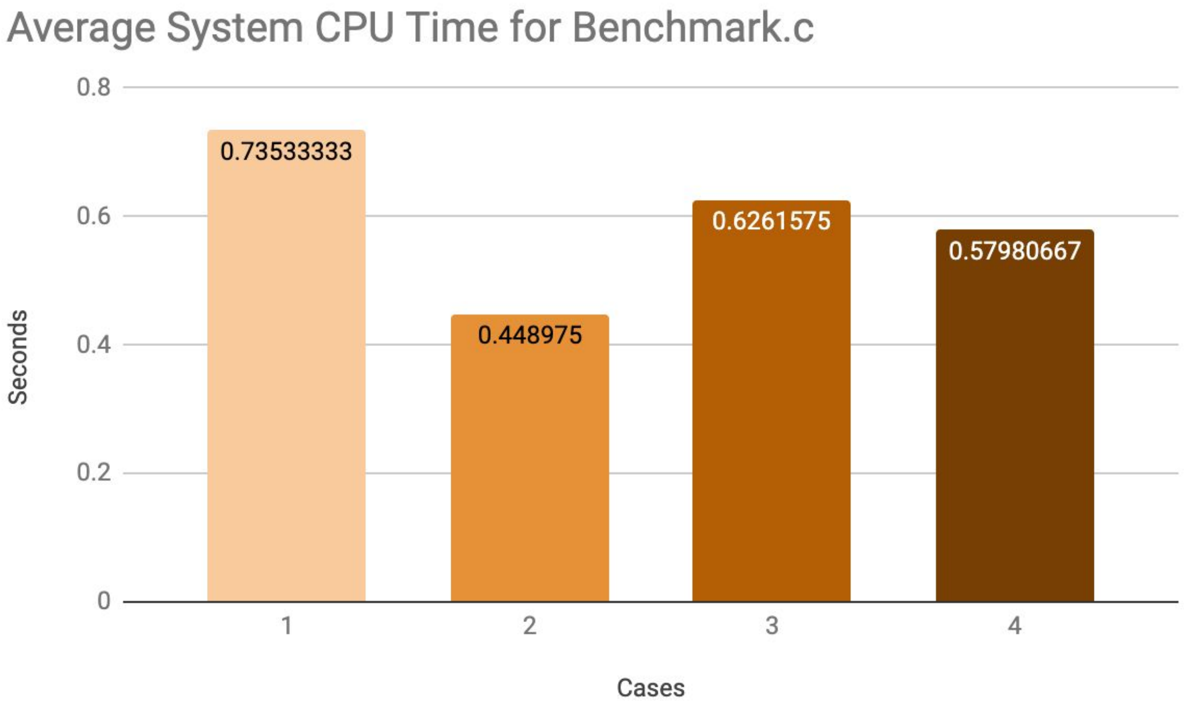
Benchmark Data and Statistics:



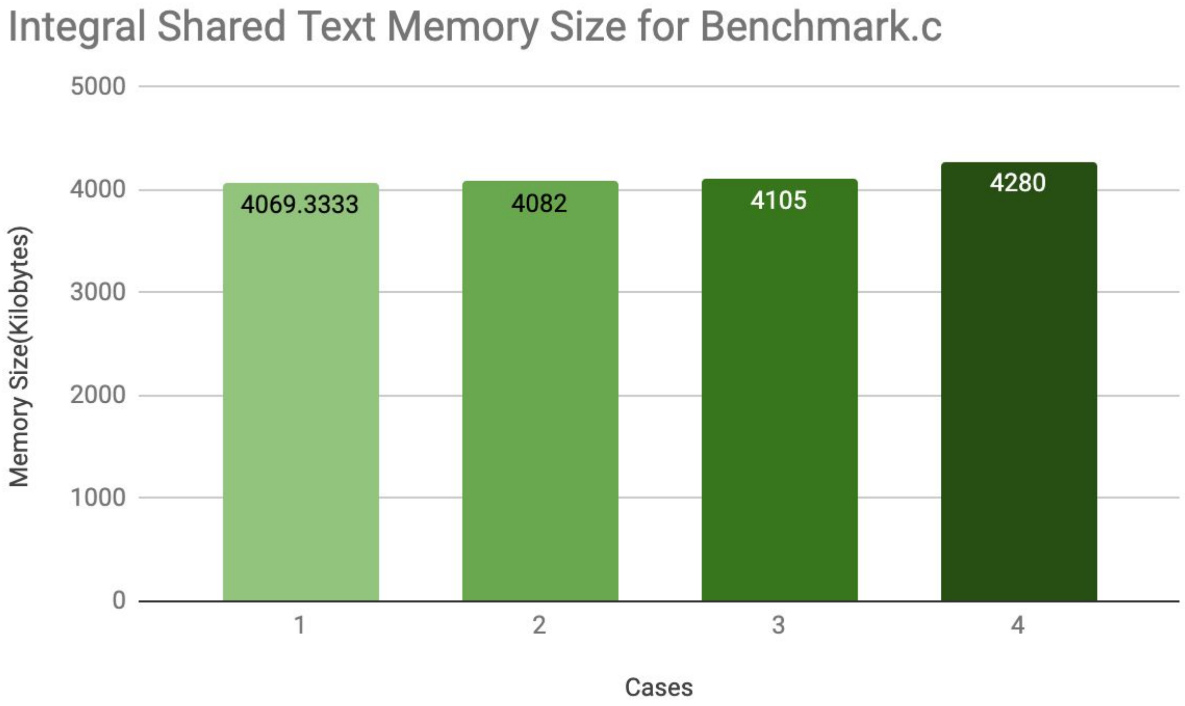
Description: This shows the average runtimes for the benchmark, benchmark.c, and its functions. As you can see, the times are very close to each other apart from the slight decrease in runtime for priority queues compared to FIFO queues for priority scheduling; however Case 3 and Case 4, Splatter scheduling with FIFO and Splatter scheduling with priority, took longer than Case 1 and Case 2. This is because of the scheduling change with randomization, rather than priority-assigning to threads.



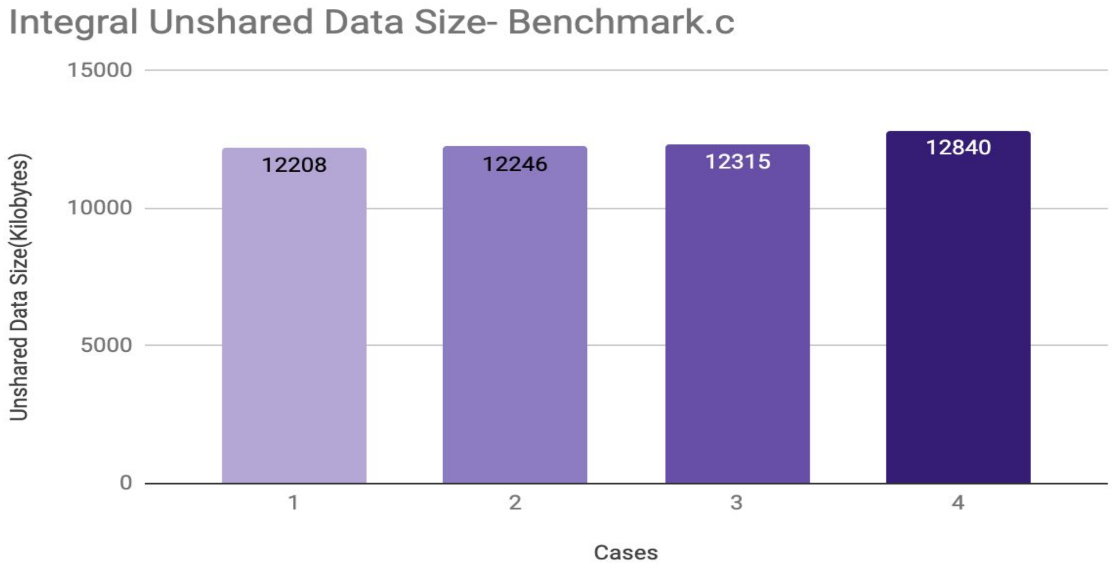
Description: This graph shows average User CPU Time for Benchmark.c for all the cases. As we see again, the randomized scheduler for Case 3 and Case 4 takes longer than the priority scheduler in Case 1 and 2. This is again because of the scheduling change with randomization, rather than priority-assigning to the threads. Compared to the System CPU Time, the User CPU Time takes longer, as it does not have that close of an affinity with the CPU.

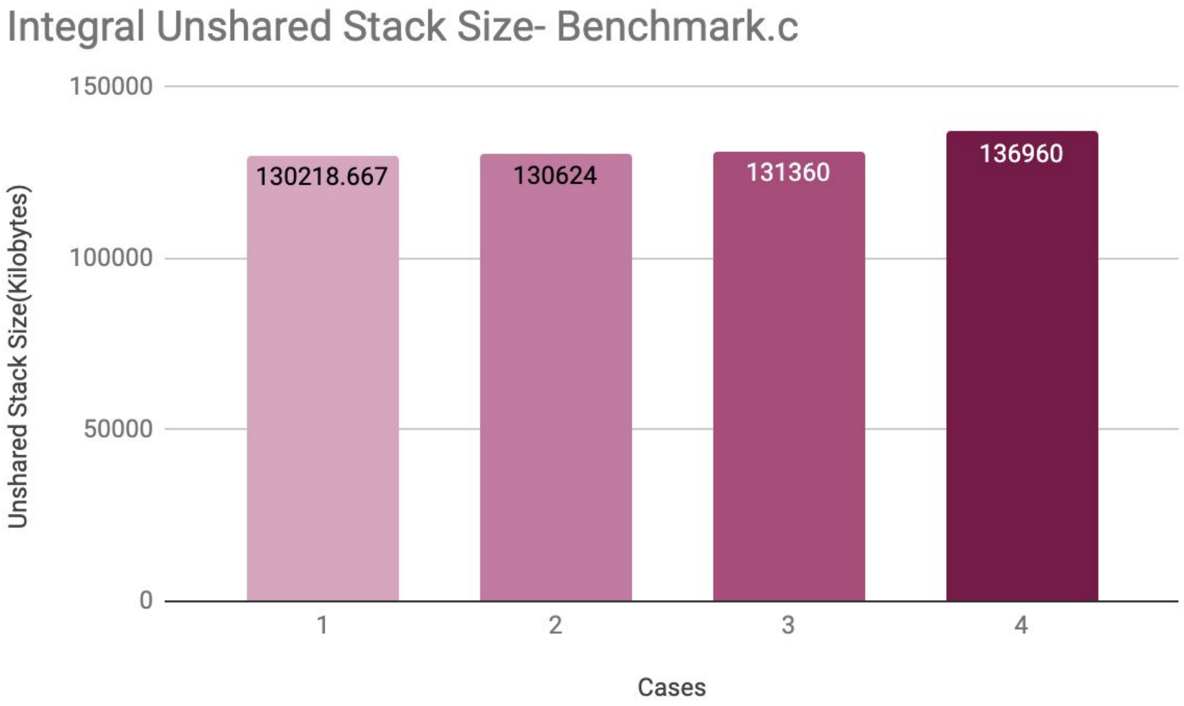


Description: The benchmark file, benchmark.c, is CPU-intensive. This can be seen from the results of the bar graph for average system CPU time for each of the cases. The low-level scheduler shows the most drastic of the differences between the scheduling implementations, while, though still within the system, the high-level scheduler doesn’t have as drastic of a difference between the priority scheduling and splatter scheduling. Ultimately, these differences between the system CPU times of the different cases are explained by the greater CPU affinity that the system level has, than the user level.

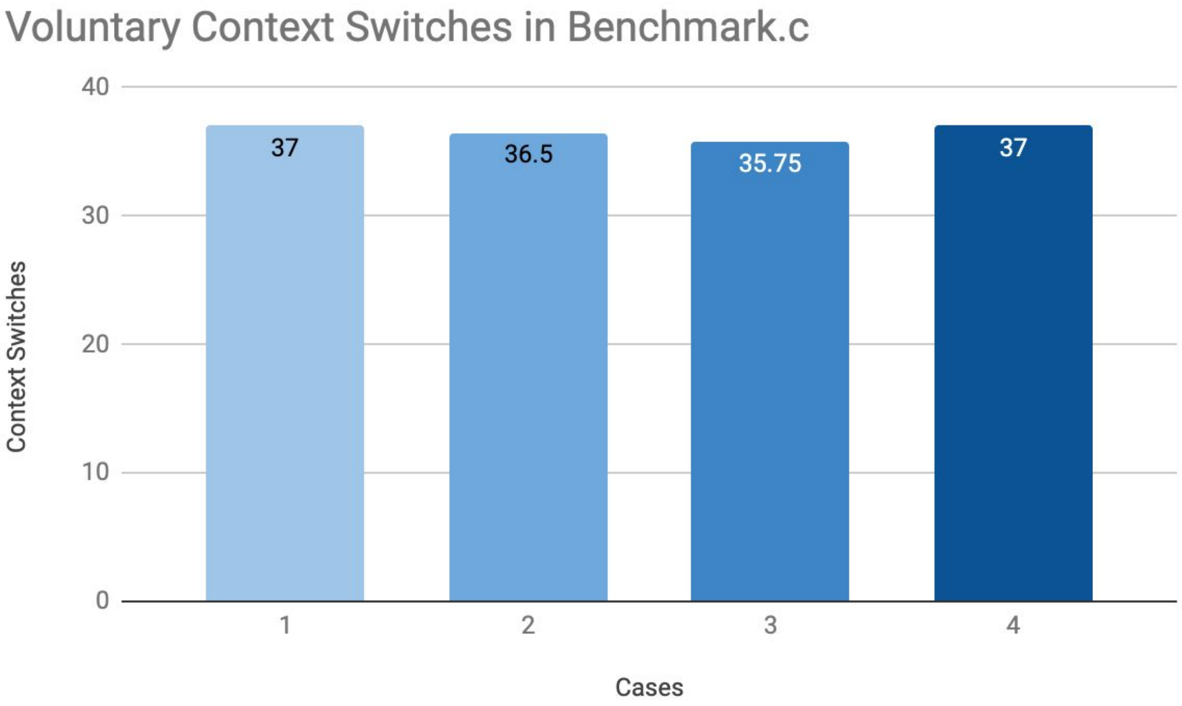


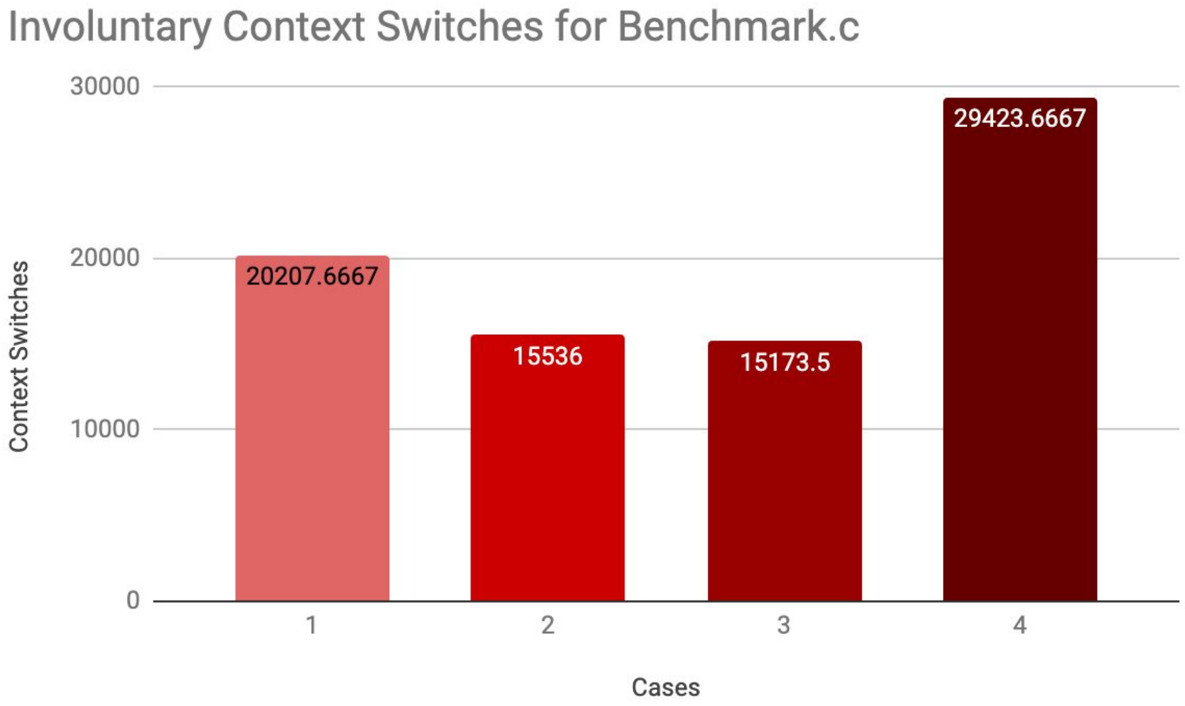
Description: This graph portrays the Integral Shared Text Memory Size for Benchmark.c. It basically portrays the amount of memory used by text that is shared with other processes. As shown, Cases 3 and 4 have a higher shared text memory size than Cases 1 and 2.





Description: The graphs above portrays the Integral Unshared Data Size and the graph below portrays the Unshared Stack Size for Benchmark.c. It basically portrays the amount of data and space of stack, independent between the processes for each case.

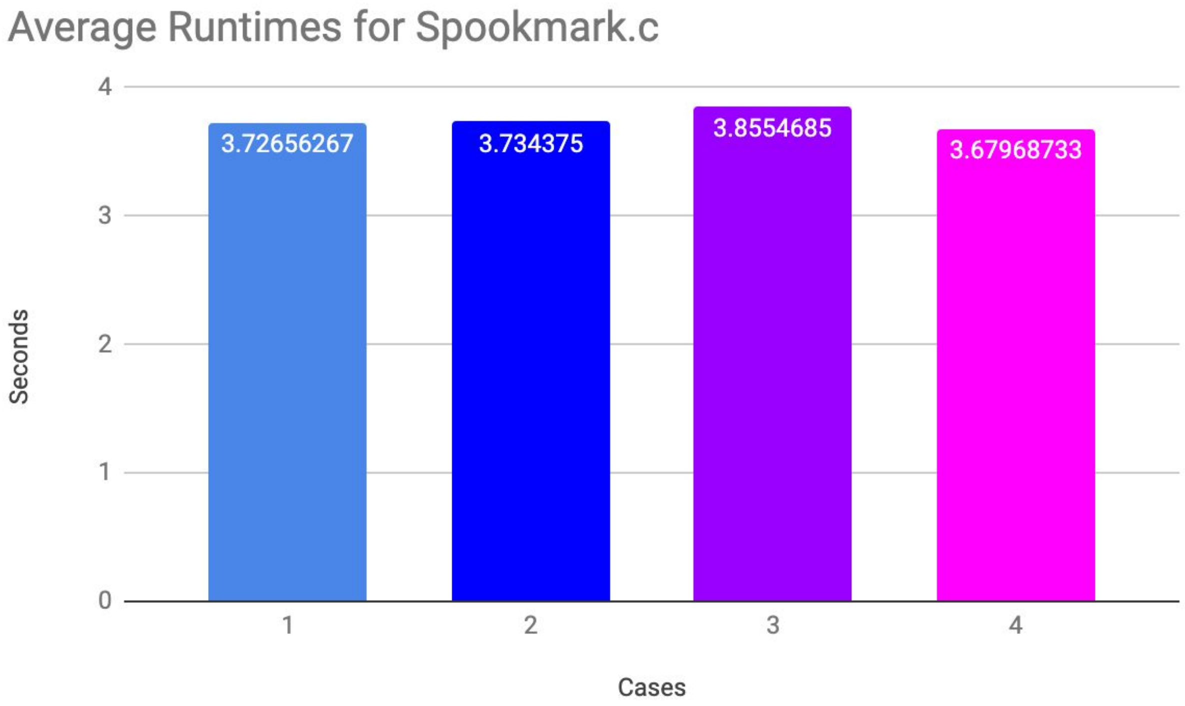




Description: The graphs above portray both voluntary and involuntary context switches between all cases for benchmark.c. The differences between the cases for voluntary context switches is not much, since the resources that become unavailable don't change much; in other words, the access to the data structure resources is not changed much since, both FIFO and priority queues have two openings for thread data to come in and come out.

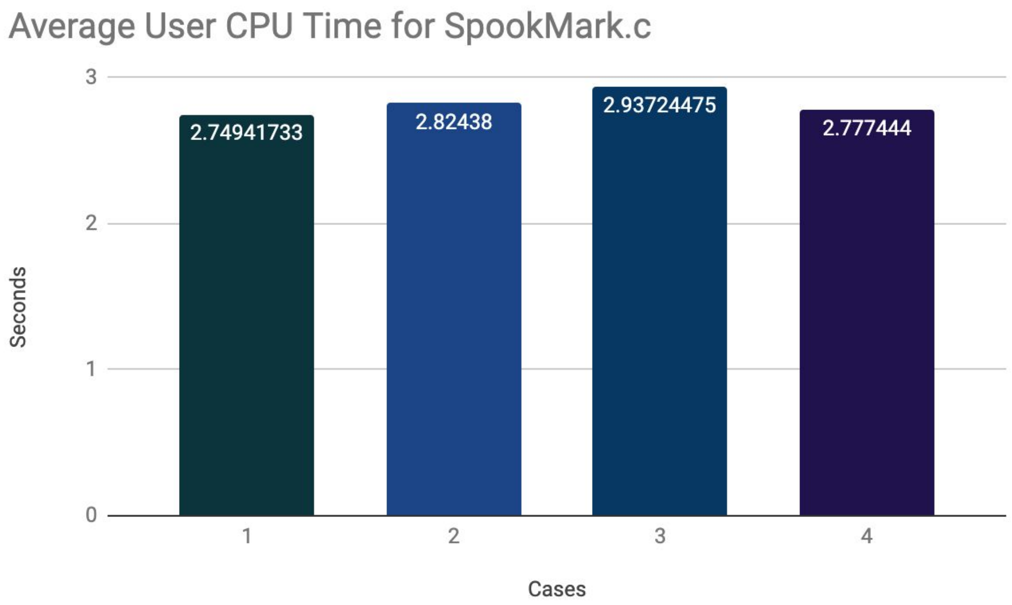
On the other hand with involuntary context switches, priority queues make a difference compared to FIFO queues with simple priority scheduling as the high-level scheduler (case 2 vs. case 1), since priority queues put the priorities in order; it prevents an increase in identifying higher level priority threads to run since they are in order, and unexpected changes in priorities are not encountered. FIFO queues usually have a more random order since threads are added to them, just on the basis of the high-level scheduler, and thus, more unexpected changes in priorities of threads are in fact encountered. This is why case 2 has less involuntary context switches compared to case 1.

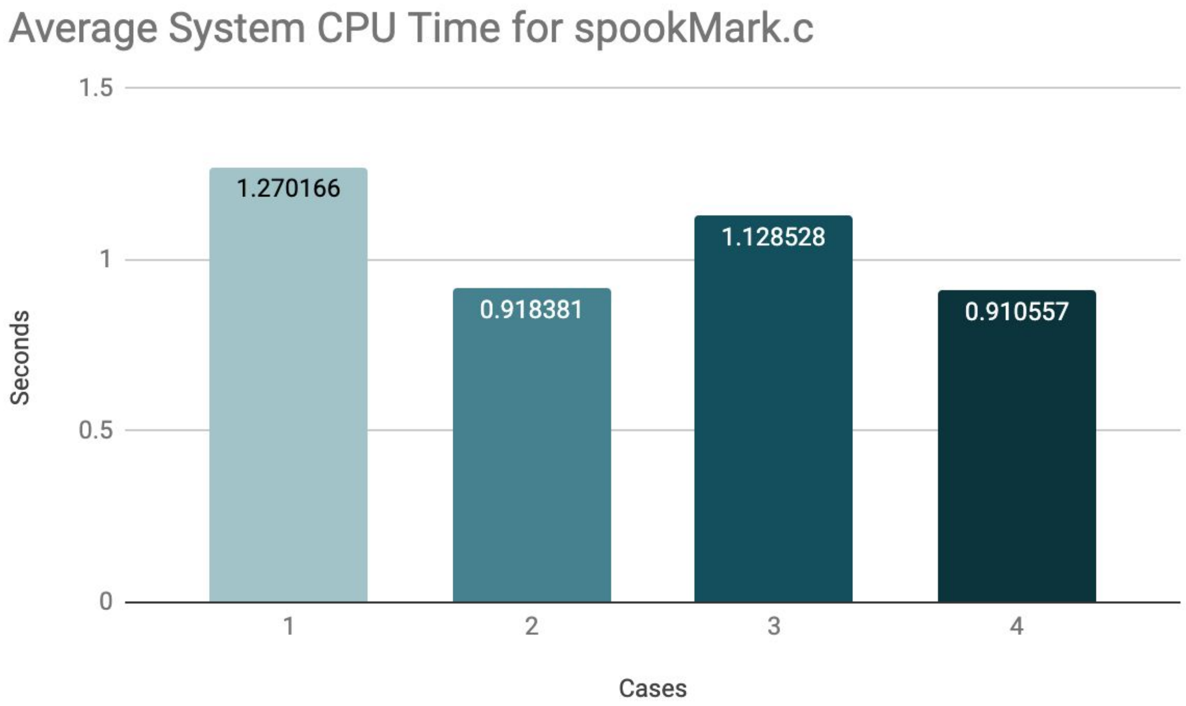
Splatter scheduling is a randomized scheduling method, and thus maintains the run of threads through their time slices in a much less orderly fashion; so a drastic increase in involuntary context switches is seen in the results of the data average for case 4.



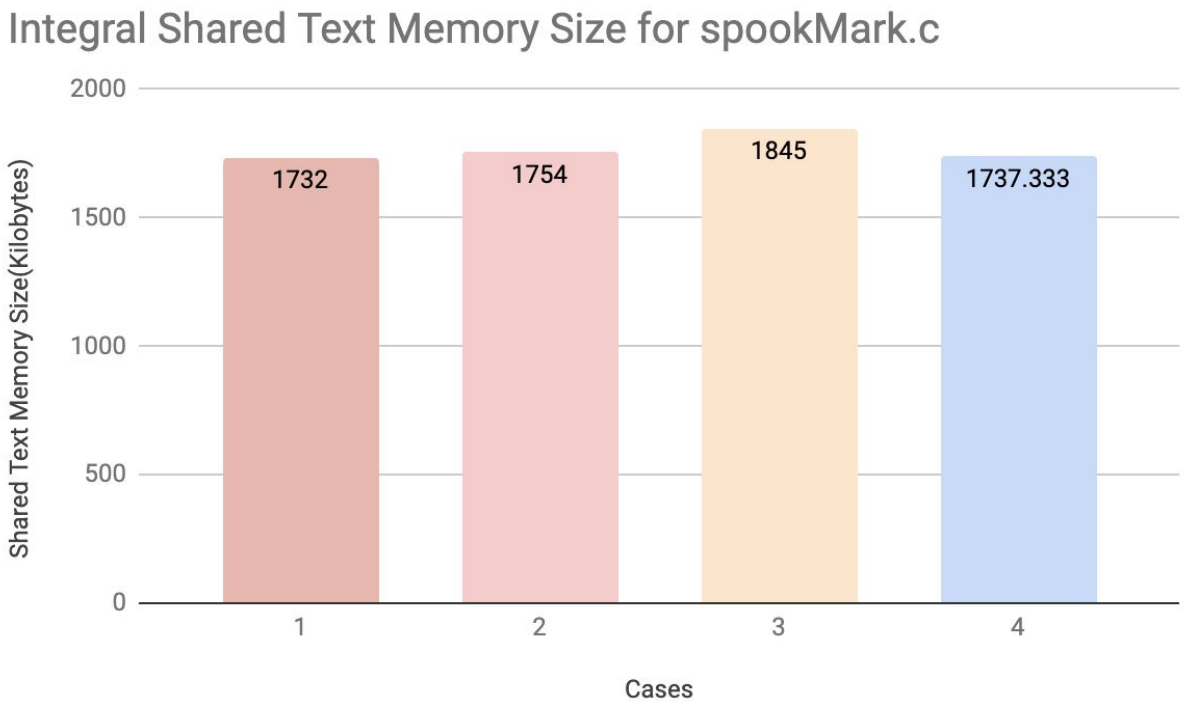
Description: This graph shows the average runtimes for the test file spookMark.c, which is more I/O intensive. We ran this benchmark for all the cases and as we can see, the runtime for Case 3 was greater than the runtime for Case 1,2, and 4. This could be due to the fact that Case 3 was implementing splatter scheduling with FIFO queues, as it is randomized as well as implemented with FIFO queues, therefore it could take longer for the program to run.

Description: This graph below contains the User CPU Time for all 4 cases, and again, it does take a lot longer than System CPU Time, as the User CPU Time has less affinity with the CPU. The spookMark.c file also has functions that are more I/O intensive, and as such, priority queue scheduling along with splatter scheduling increases the user CPU time, since I/O is a user-reliant activity.

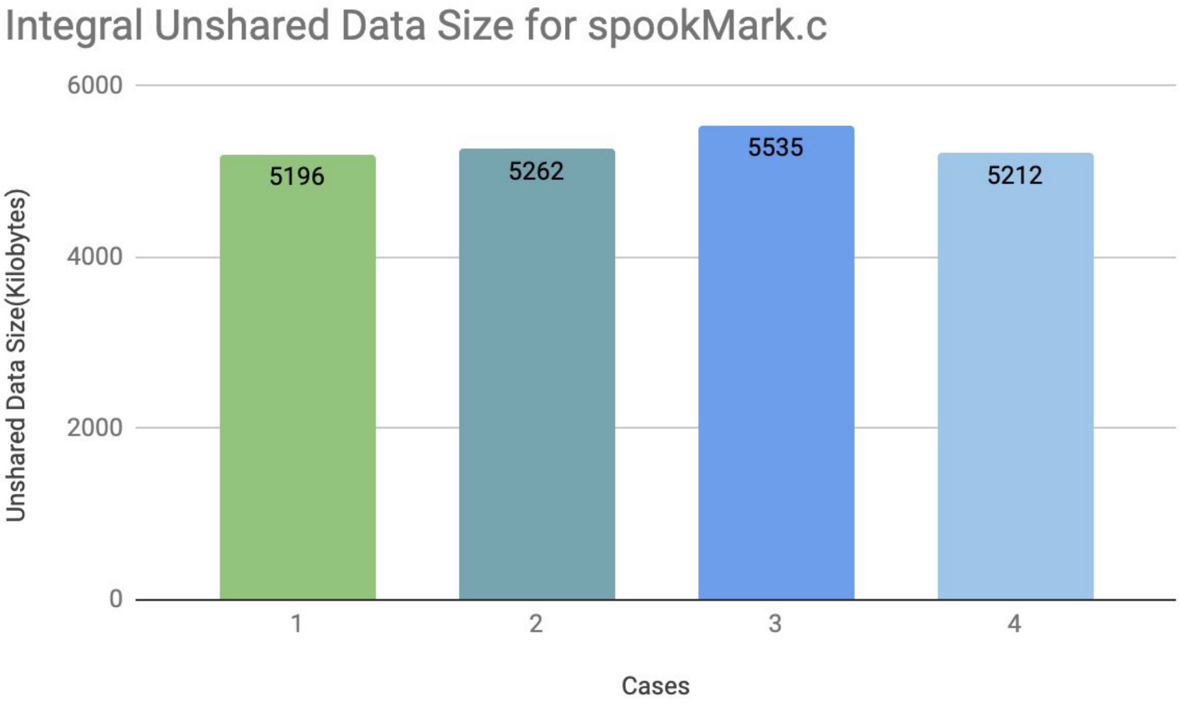

Average User CPU Time for SpookMark.c

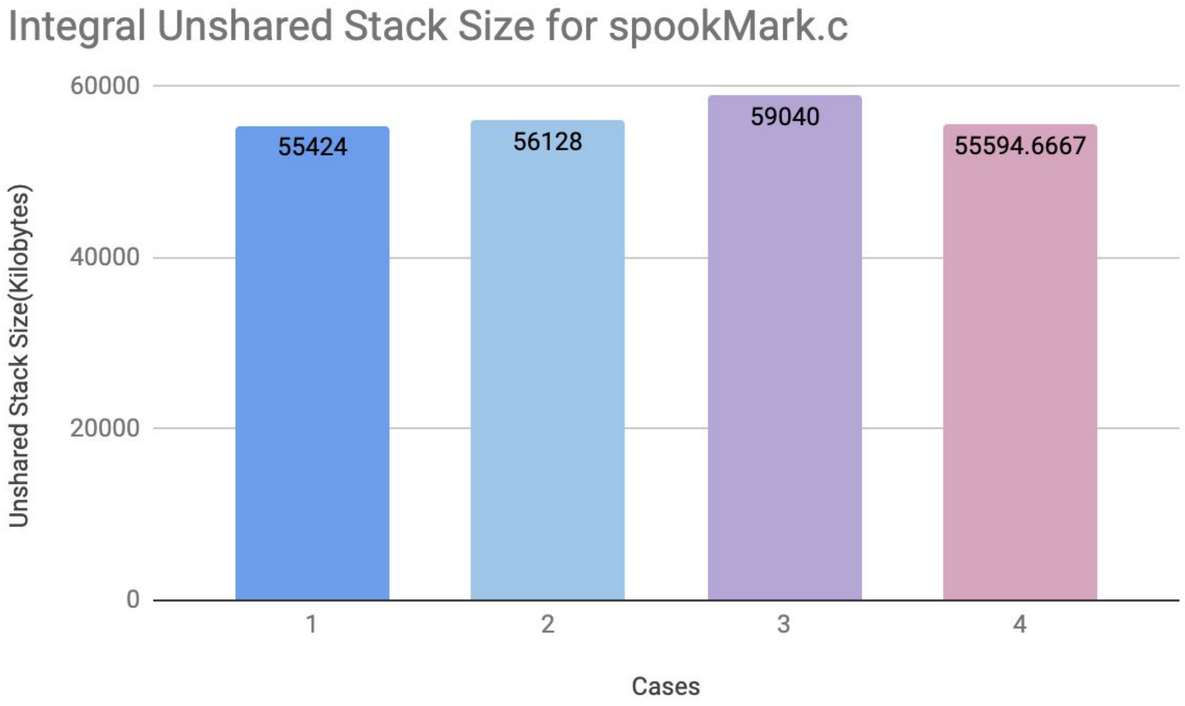

Average System CPU Time for spookMark.c

Description: Shown above is the Average System CPU Time it takes for the spookMark.c on all 4 cases. This value is significantly decreased compared to the User CPU Time, as it has more affinity with the CPU comparatively. On top of that, since the spookMark.c file is more I/O intensive, it is more tightly-knit with the user space than with the kernel space. So as can be seen, the system CPU time between the four cases generally decreases. In particular, the low-level scheduler change from FIFO queues to priority queues has a greater decrease in system CPU time, than the high-level scheduler change, since the low-level scheduler is even deeper in the kernel space, and thus, further from I/O-prominent activity that happens in spookMark.c

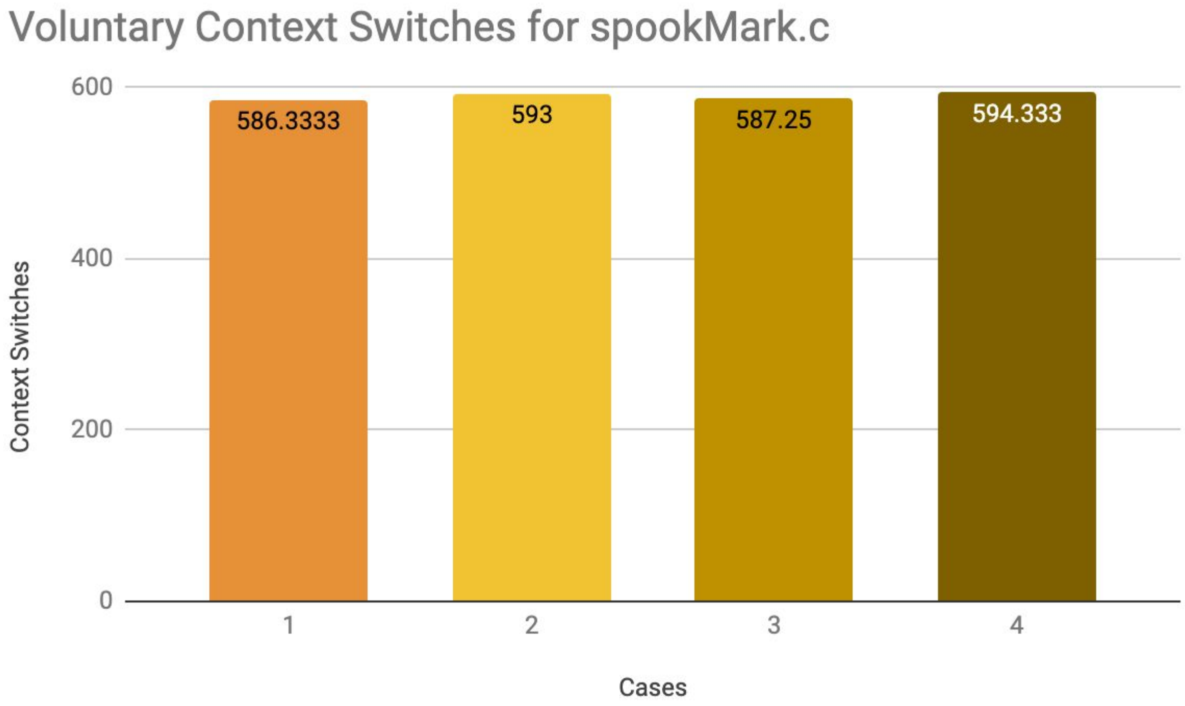


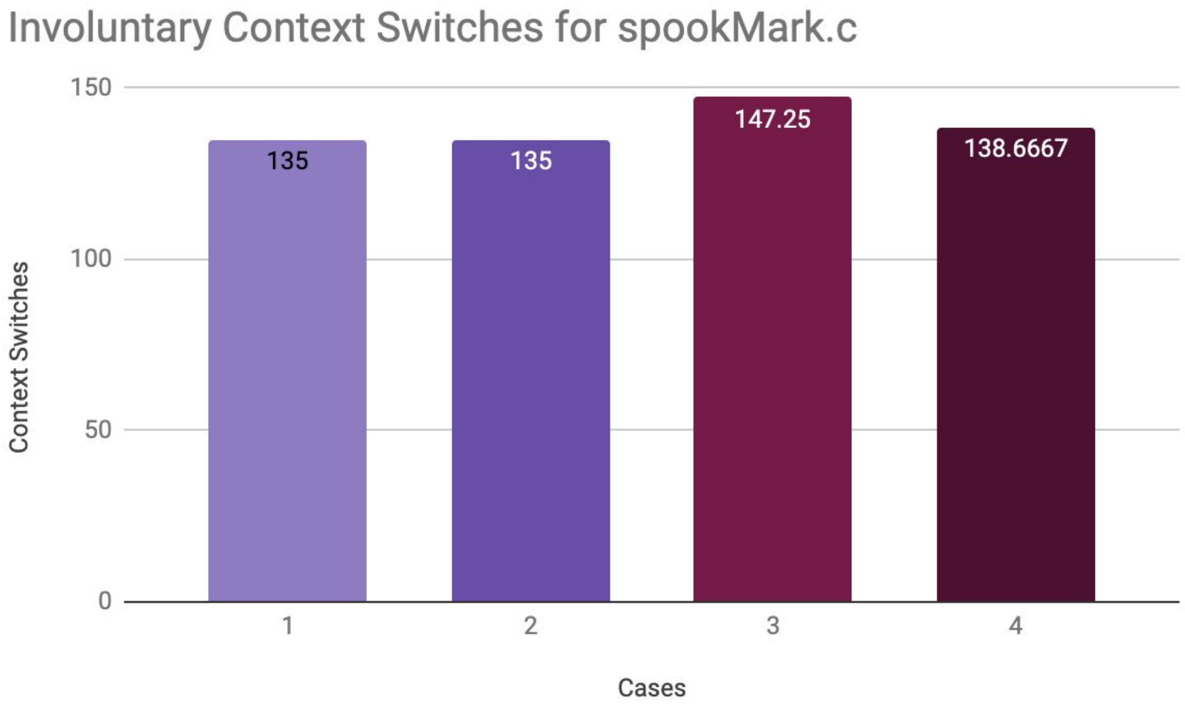
Description: Shown above is the Integral Shared Text Memory Size for spookMark.c. It basically portrays the amount of memory used by text that is shared with other processes. As shown again, Case 3 has the highest text memory size compared to the rest of the cases, which can be due to the fact of the splatter scheduling and implementation of the FIFO queues.


Integral Unshared Data Size for spookMark.c
6000
4000
2000
0
Unshared Data Size(Kilobytes)
5196
5262
5535
5212
1
2
3
4
Cases



Description: The graphs above show the Unshared Data and Stack size in the program spookMark.c. This basically shows the amount of data and space of stack, independent between processes.





Description: The two graphs above show voluntary and involuntary context switches for spookMark.c on all cases. As we can see, Case 3 has a higher involuntary context switch, whereas Case 4 has the highest voluntary context switch. Since spookMark.c is I/O reliant, the functions in this file run primarily in user space. So there are a much larger number of voluntary context switches for all the 4 cases for spookMark.c, since the threads require data structure resources more frequently. This is the opposite for involuntary context switches.