Computer Efficiency and Delta Queuing

Ramsey Alsheikh

Abstract

Delta queuing is an algorithm in which computer processes are set to sleep for an amount of time offset from the other's in order to maintain a resource-light process management system. The purpose of this experiment was to implement a delta queuing algorithm within Linux (specifically, Ubuntu) and observe its efficiency in terms of the computer's resources. The hypothesis was that the algorithm would be efficient in comparison to a control algorithm implementing no resource-saving methodologies. Two system calls were developed, one for delta queuing and another as a control group. These system calls were called from a user-space wrapper function that outputted many statistics about the system calls. The delta queuing algorithm was found to be superior in time and process management, while being inferior to the control group in terms of CPU usage and memory usage. Overall, the hypothesis was supported, as the control algorithm was unable to fulfill its ultimate purpose of sleeping for a set amount of time, due to the excessive load per millisecond inherent with not using a delta queuing system. However, delta queuing may not be suitable for all systems. This experiment has applications in the field of low level, operating systems design and software engineering, by illustrating the merits and failures of the delta queuing algorithm.

Introduction

Statement of Purpose

The purpose of this experiment would be to test the difference in system resource use between delta queuing and normal queuing. The experimental group and control group will be implemented as individual system calls in Linux. The dependent variable is which of the two system calls is being tested, while the independent variable is the resource use of that system call. If the delta queuing system call performs better than the control system call, then it would provide favorable evidence for implementing a delta queue within Linux and other operating systems. Constant variables include the version of Linux being used and the computer being used.

Background Research

In a computer, any program running, such as a small, basic system maintenance task or a large end-user application, is called a process (Process). The processes running on a system can be put to sleep - which means it no longer can access the CPU (Barbar, 2012). Processes are put to sleep when the resources they need to enter the Running or Runnable state are not available/being used. Either the kernel or the process itself can make the process go to sleep (Barbar, 2012). Processes that are asleep can optionally be set to wake up after a certain time (Barbar, 2012). These sleeping processes are stored in a queue, a type of data structure in which items are processed in the order they are added (Stacks and Queues). One way to manage a queue of timed, sleeping process is through a delta queue. A delta queue is a queue in which the value holding how much time left to wait is offset relative to all the values before it. With sleeping

processes, it is possible to use a delta queue where the sleeping value is equal to initial amount of time left for that process subtracted from the sum of all the values in all the processes before it.. Delta queuing has the advantage of only having to continuously manage one process's sleeping time in the queue to appropriately affect all the items after it. See Figure 11 for a diagram of this process. This could save considerable system resources, especially when the fact that sleep queue is managed thousands of times a second. This experiment is relevant in today's world because every major operating system puts processes to sleep, and needs a way of timing how long each process has been asleep for. Researching how effective delta queueing is would provide valuable information on whether it is the best way to manage sleep processes, and if a new structure should be adopted.

Hypothesis

If a delta queue is compared to a control algorithm, it will be more resource efficient. In theory, a delta queue should only have to perform as many timer operations as the longest waiting process requires, due to the effect of offsetting the wait times from each. In contrast, the control algorithm will have to wait for each process individually, so the number of timer operations will be the sum of all given. Equations for both scenarios respectively are $\Delta = \max(x_1...x_n)$ and $\Gamma = \sum(x_1...x_n)$. This is not to be confused with the amount of time necessary to wait for all the process however, which should be the same for both, as multiple timer operations can be performed in one time interval (if the time is greater than the time it takes to finish the last one, it may be indicative of too much work for the intervals in between). Rather, it refers to the workload of the computer, the amount of items to keep track of. However, despite this property,

the delta queue requires the overhead of sorting the list of processes and offsetting them, which may overtake the savings earned in the main run. This experiment will test this.

Method

Materials

- A computer capable of running Ubuntu 5.2.7, with administrator access
- Charging Cable
- Source code, located at https://github.com/DatOneRam/delta

Procedure

1. Verify your computer has x86 architecture capable of running Ubuntu kernel version 5.2.7, and install Ubuntu.
2. Download all code from *https://github.com/DatOneRam/delta*.
3. Download the source code of kernel version 5.2.7 and add the two system calls (*control.c* and *delta.c* from the GitHub repository) to the kernel as appropriate to your machine.
4. Make and install the new kernel.
5. Reboot your system.
6. Navigate to where you saved the GitHub repository and run the bash script *experiment.sh*.
7. In the same folder, run bash script *control.sh*.
8. Open the new file *experiment_results.txt* and analyze data.
9. Open the new file *control_results.txt* and analyze data

Results

	Peak Resident Set Size	Real CPU Time	System CPU Time
Trial One	748 kilobytes	9.86 seconds	9.74 seconds
Trial Two	736 kilobytes	9.86 seconds	9.79 seconds
Trial Three	812 kilobytes	9.86 seconds	9.77 seconds
Average	765 kilobytes	9.86 seconds	9.77 seconds

Table 1. Delta Group Trial Results

	Peak Resident Set Size	Real CPU Time	System CPU Time
Trial One	740 kilobytes	9.88 seconds	9.86 seconds
Trial Two	716 kilobytes	9.97 seconds	9.86 seconds
Trial Three	792 kilobytes	9.87 seconds	9.86 seconds
Average	748 kilobytes	9.91 seconds	9.86 seconds

Table 2. Control Group Trial Results

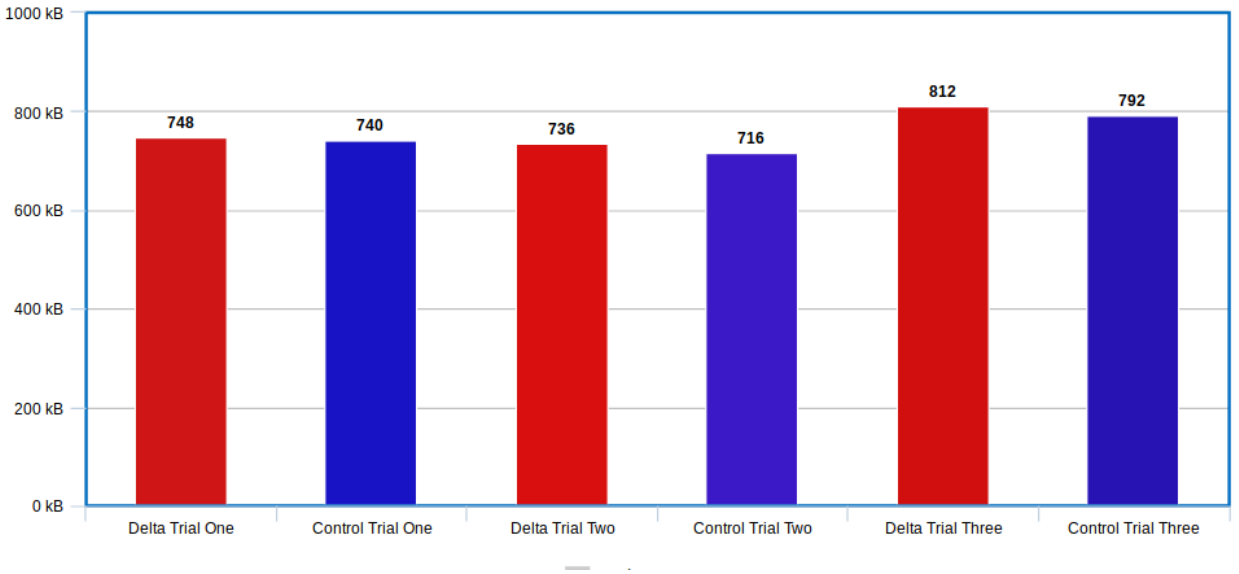


Figure 1. Memory Usage

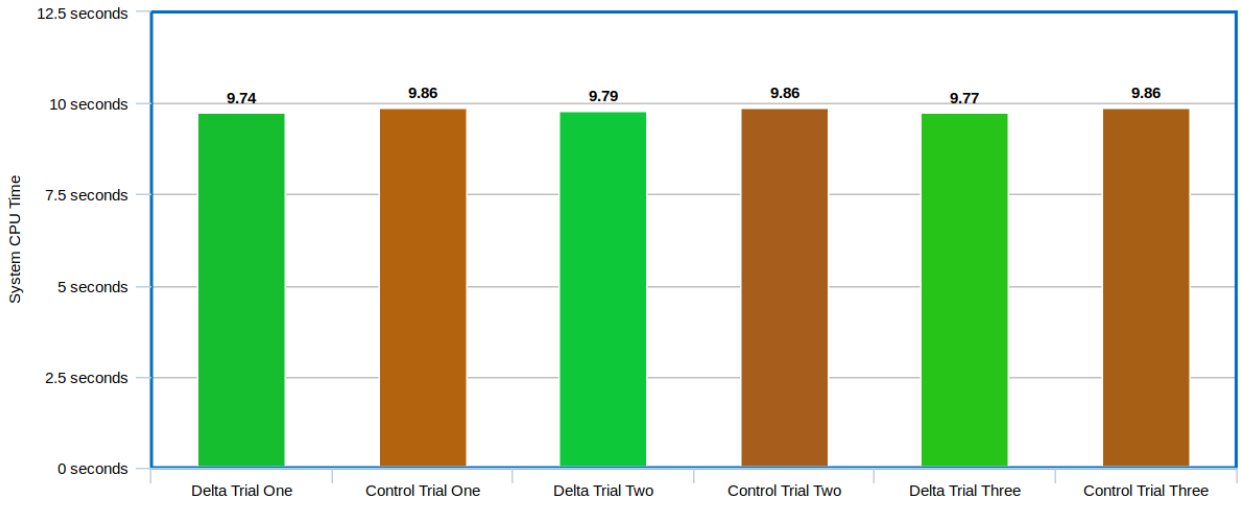


Figure 2. System CPU Time

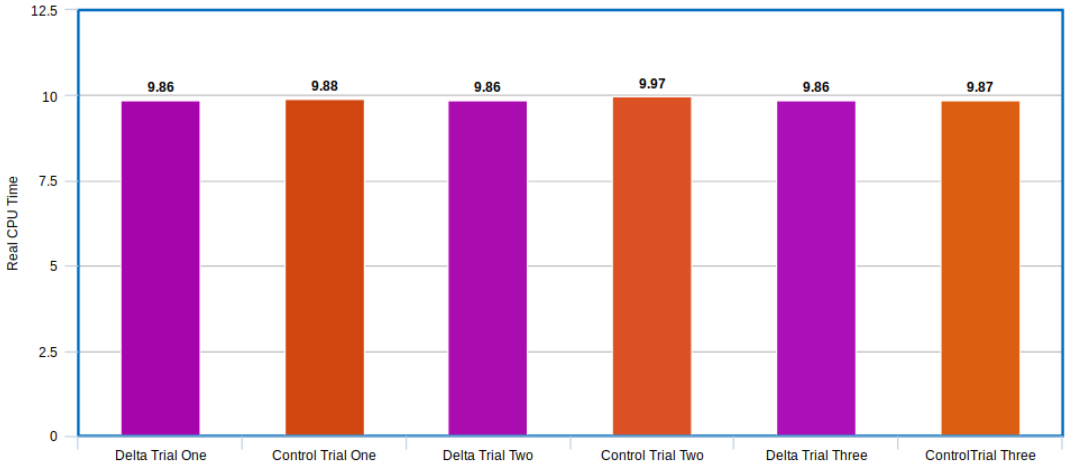


Figure 3. Real CPU Time

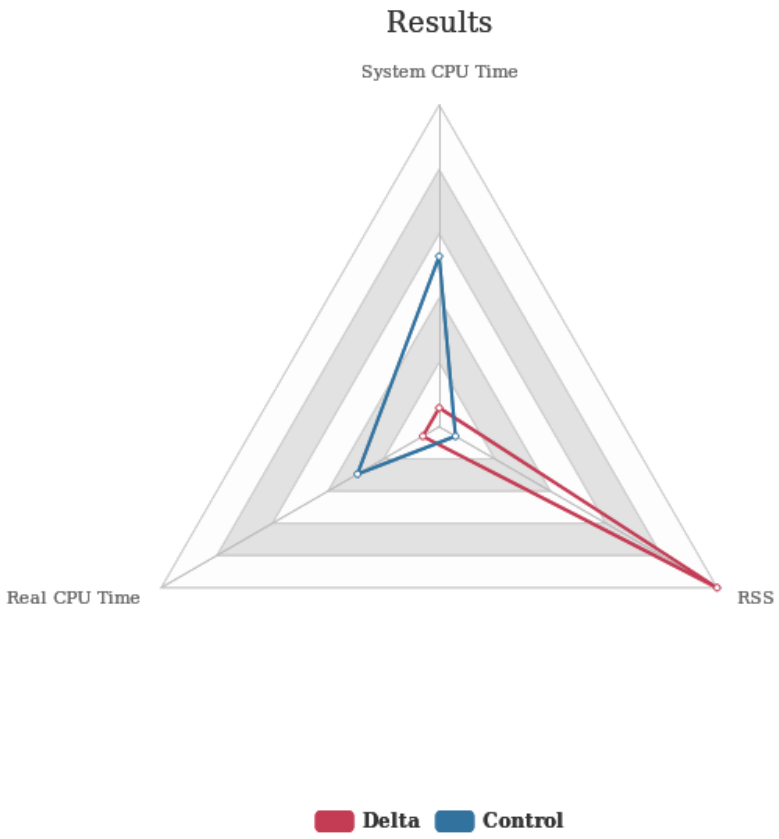


Figure 4. Comparison Radar Graph

```
typedef struct _delta_entry
{
        struct task_struct *task;
        struct list_head list;
        int delta_time;
} delta_entry;
```

Figure 5. Code Representing a Delta Process

```
typedef struct _control_entry
{
        struct list_head list;
        struct task_struct *task;
        int time;
} control_entry;
```

Figure 6. Code Representing a Control Process

```
//method to add a delta_entry into the delta queue
//if there are no entries in the delta queue as of yet, dont use this, use the list_add that comes w
void dq_add(delta_entry *new, int init, delta_entry* head)
{
        delta_entry* pos;
        int sum = 0;

        //add up the delta times before the time we want to add
        pos = list_entry(head->list.prev, delta_entry, list);
        action:
        sum += pos->delta_time;
        if (pos != head && init > sum)
        {
                pos = list_entry(pos->list.prev, delta_entry, list);
                //uncomment for debugging purposes
                //printk("dq_add: sum = %d, init = %d", sum, init);
                goto action;
        }

        //calcuate the delta_time for our new entry
        sum -= pos->delta_time;
        new->delta_time = init - sum;

       //add our new entry at the appropiate spot
       __list_add(&new->list, &pos->list, &list_entry(pos->list.next, delta_entry, list)->list);

       //update the entry behind it so that it maintains its total time
       if(new->list.prev != head)
       {
                list_entry(new->list.prev, delta_entry, list)->delta_time -= new->delta_time;
       }
```

Figure 7. Code to Maintain the Delta Queue

```
list_for_each_entry(pos, &head.list, list)
{
        pos->time -= 1;
        if (pos->time <= 0)
        {
                //printk("REMOVING");
                pos->list.prev->next = pos->list.next;
                pos->list.next->prev = pos->list.prev;
                wake_up_process(pos->task);
        }
}
```

Figure 8. Code to Maintain the Control Algorithm

```
//method to actually make and run the delta queue
asmlinkage long sys_delta(void)
{
        //an array of 100 randomly generatred numbers in the range of 1-1000. Used as the different times each threa
        int sleep_times[] = {6503, 4363, 9529, 5259, 3730, 1428, 9558, 7083, 4226, 9760, 3778, 2244,
5511, 5194, 3769, 6417, 745, 7777, 7072, 795, 4951, 558, 3834, 9643, 2668, 515, 8776, 4654, 2707, 570, 7729,
 4395, 6742, 6911, 9908, 5395, 1381, 9220, 9162, 1783, 82, 3484, 5781, 4605, 884, 6366, 1149, 2970, 5000,
1285, 7987, 9786, 5607, 8714, 2642, 9048, 8883, 4872, 1545, 5425, 7418, 8389, 2709, 2602, 2869, 5262, 2939,
 9808, 7402, 919, 334, 4308, 6390, 9354, 5306, 7809, 2240, 7090, 6001, 1844, 8200, 7613, 9668, 5191, 3381,
996, 4721, 3641, 4334, 1271, 2971, 2334, 9872, 8185, 3626, 7530, 1954, 8026, 845, 1693};
        //the head of the delta_queue
        delta_entry head = {NULL, LIST_HEAD_INIT(head.list), NULL};
        //an array of 100 delta entriek(s for our delta queue
        delta_entry entries[100];
        //counter variable
        int i;

        //make delta_entries and enqueue them
        for (i = 0; i < 100; i++)
        {
                delta_entry de = {make_thread(sleep_times[i]), LIST_HEAD_INIT(de.list), sleep_times[i]};
                entries[i] = de;
                dq_add(&entries[i], entries[i].delta_time, &head);
        }

        //main loop
        do
        {
                mdelay(1);
                dq_update(&head.list);
                //printk("just waiting again...")
        }
        while (dq_size(&head.list) > 0);

        return 0;

}
```

Figure 9. Main Loop of Delta Algorithm

```
asmlinkage long sys_control(void)
{
        int i;
        control_entry entries[100];
        control_entry *pos;
        control_entry head;
        INIT_LIST_HEAD(&head.list);
        head.time = NULL;

        int sleep_times[] = {169, 280, 560, 602, 714, 1018, 1290, 1451, 1506, 1532, 1584, 1690, 1706, 2229,
2309, 2432, 2476, 2499, 2554, 2559, 2567, 2659, 2755, 2822, 2899, 3129, 3147, 3176, 3333, 3365, 3427, 3485,
3618, 3620, 3703, 3845, 3970, 4020, 4431, 4472, 4505, 4676, 4677, 4784, 4789, 4944, 4975, 5082, 5108, 5154,
5208, 5237, 5359, 5527, 5700, 5842, 5860, 5924, 6067, 6169, 6173, 6250, 6274, 6332, 6353, 6566, 6597, 6674,
6790, 6847, 6883, 7148, 7214, 7317, 7400, 7435, 7547, 7640, 8031, 8089, 8102, 8152, 8383, 8448, 8500, 8515,
8735, 8778, 8799, 8881, 8887, 8934, 9046, 9075, 9254, 9266, 9462, 9568, 9927, 9930};

        //create control entries
        for (i = 0; i < 100; i++)
        {
                INIT_LIST_HEAD(&entries[i].list);
                entries[i].time = sleep_times[i];
                entries[i].task = kthread_run(napb, NULL, "control entry for inital %d", sleep_times[i]);
                list_add(&entries[i].list, &head.list);
        }

        //main loop
        do
        {
                mdelay(1);
                list_for_each_entry(pos, &head.list, list)
                {
                        pos->time -= 1;
                        if (pos->time <= 0)
                        {
                                //printk("REMOVING");
                                pos->list.prev->next = pos->list.next;
                                pos->list.next->prev = pos->list.prev;
                                wake_up_process(pos->task);
                        }
                }
        }
        while (size(&head.list) > 0);

        return 0;
}
```

Figure 10. Main Loop of Control Algorithm

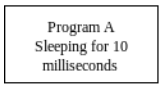
```
//make delta_entries and enqueue them
for (i = 0; i < 100; i++)
{
        delta_entry de = {make_thread(sleep_times[i]), LIST_HEAD_INIT(de.list), sleep_times[i]};
        entries[i] = de;
        dq_add(&entries[i], entries[i].delta_time, &head);
}
```

Figure 11. Delta Process Creation

```
//create control entries
for (i = 0; i < 100; i++)
{
        INIT_LIST_HEAD(&entries[i].list);
        entries[i].time = sleep_times[i];
        entries[i].task = kthread_run(napb, NULL, "control entry for inital %d", sleep_times[i]);
        list_add(&entries[i].list, &head.list);
}
```

Figure 12. Control Process Creation

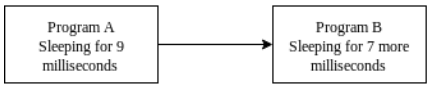
Initial Delta Queue



We add Process B, which needs to sleep for 17 milliseconds.

Since A is already sleeping for 10 seconds, we only need to sleep 7 seconds after A finishes to wait for B's 17 seconds.

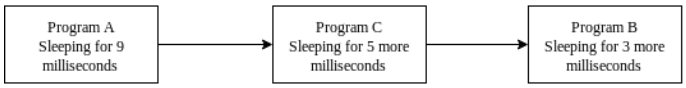
Every millisecond, we must subtract one from the program at the front to affect all subsequent programs, since they are offset from each other.



If we want to add program C, which sleeps for 14 seconds, it will be between A and B, so that it does not wait for too long or cause others to wait for too long.

It will wait for 5 milliseconds, since 14 - 9 = 5.

In addition, we must update B to 3 milliseconds, since the delta queue will be now be waiting for 14 milliseconds before C will wait. 17 - 14 = 3.



After 9 milliseconds, we can wake up A and C will be at the front of the delta queue.

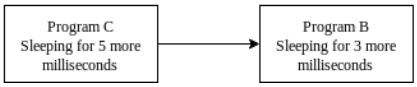


Figure 13. Sample Delta Queue Diagram

Calculations

Average = sum of the data / amount of the data

Percentage = part/whole * 100

The averages for each of the data fields was calculated by taking the summation of the data in the field and dividing by the number of data entries. A percentage is calculated by dividing a part of a whole by the whole and multiplying by 100. The individual data entries were obtained by outputting the /proc/$PPID/status file to a text file before it was deleted. It contains valuable information about process memory use. However, it does not cover cpu time, so the bash command “time” was used. This output of time was also printed onto the same output file mentioned above. The CPU of the client computer has a base frequency of 1.6 GHz and a maximum frequency of 3.4 GHz. The average is 2.5 GHz, which coincides with the average international CPU frequency (CPU Trends, 2018).

Discussion

Conclusion

In this experiment, a delta queue algorithm and control algorithm were run through two system calls, and their resource use was measured. While the delta queue algorithm took up on average 17 more kilobytes of memory, the control algorithm used around 0.1 more seconds of CPU time, and on average 0.05 more seconds of real time. The delta algorithm would have used an extra 0.000002 percent of memory on the average 8 gigabyte system (Memory Trends, 2018) miniscule overhead. Moreover, the delta queue saved about 1% of the average CPU’s frequency. This tenth of a second of CPU time is quite significant when one considers the speed of today’s

processors - on modern CPUs it can amount to billions of instructions. Thus, the hypothesis was supported.

Applications

This experiment is of use to computer scientists through demonstrating the effectiveness of the delta queue and proving its worth. It was able to provide evidence and reasons for the implementation of it in a real operating system. When the system calls were run, delta queuing was marginally less effective on memory, and made up for it in saved CPU time. Existing operating systems, such as Linux or Windows, have a reason to create and test a delta queue implementation of their own. Newer Operating systems currently in development might find it worthwhile to shift the system while it is still young to delta queuing. In addition, it has merits beyond this in in showing that there are other ways to manage processes besides the current, straightforward method. Overall, this experiment provided reasons for delta queuing's place as a legitimate, efficient algorithm in modern computer science.

Limitations

This experiment could be improved if a final, usable, official implementation was tested. This experiment was only a simulation, and could be more refined. Although an attempt was made to simulate a real implementation as much as possible, using lower level functions and procedures would have a high chance of reducing variables that could have affected efficiency. Testing the delta queuing algorithm under a variety of different architectures would make for a better experiment. A team of developers, rather than just the researcher, would foster a better program. Review by professional operating system developers would help ensure good code as well.

Error Analysis

The delta queue algorithm was developed by one person, capable of mistakes and failures. It is possible the implementation tested is not a representative delta queue algorithm. The code was run on the same computer every time, however, other factors relating to the computer, such as battery power left, location of the computer, temperature of environment, etc. were variable, and could possibly have affected the results. In the recording of data, numbers may have been recorded differently than what was on the screen due to human error. For example, the value of 6,435 could be recorded by mistake instead of the value of 6,453.

Future Analysis

It would be beneficial to the scientific community to have the code and setup peer reviewed by experienced programmers, to verify the software used to test the algorithms was installed correctly and contained logical code. It would be worth investigating the effectiveness of delta queueing on different operatings systems apart from Ubuntu Linux, such as Windows, other Linux distributions, etc. Expanding the resources measured over the course of this experiment would also help to give the community a broader understanding of the advantages and disadvantages of the delta queuing algorithm. In addition, a lower level and more practical implementation could be developed to verify the efficiency of the experiment provided evidence for.

References

Blocking Process. (2012, August 5). Retrieved September 6, 2019, from https://wiki.osdev.org/Blocking_Process

CPU Trends. (2018). Retrieved from https://techtalk.pcmatic.com/research-charts-cpu/

International Rules for Pre-College Science Research: Guidelines for Science and Engineering Fairs 2019–2020. (2020). In *International Rules for Pre-College Science Research: Guidelines for Science and Engineering Fairs 2019–2020*. Retrieved September 6, 2019, from https://drive.google.com/file/d/1w0CEtDtbJNAy64TwYHjnthlmNpVQcE26/view

Memory Trends. (2018). Retrieved from https://techtalk.pcmatic.com/research-charts-memory/

Palm Beach Regional Science & Engineering Fair 2019 -20 Handbook for Parents, Students, and Teachers. (n.d.). In *Palm Beach Regional Science & Engineering Fair 2019 -20 Handbook for Parents, Students, and Teachers*. Retrieved September 6, 2019, from https://docs.google.com/document/d/1c2w9JaqKgpjUHRYA0JkxT_ZF7-xKyUke3uuTXq1YVMk/edit

Process. (n.d.). Retrieved April 4, 2019, from techterms.com website: https://techterms.com/definition/process

Process States in Linux. (2017, June 28). Retrieved May 31, 2019, from kerneltalks.com website: https://kerneltalks.com/linux/process-states-in-linux/

Stacks and Queues. (n.d.). Retrieved April 4, 2019, from everythingcomputerscience.com website:

http://www.everythingcomputerscience.com/discrete_mathematics/Stacks_and_Queues.html

2019-2020 RULES SUPPLEMENT To the International Science and Engineering Fair Rules. (n.d.). In *2019-2020 RULES SUPPLEMENT To the International Science and Engineering Fair Rules*. Retrieved September 6, 2019, from https://drive.google.com/file/d/1UK3wtkUGoBMVhbsNMAyCPiTaqZOQuJCL/view

What is Linux? (n.d.). Retrieved September 16, 2019, from https://www.linux.com/what-is-linux/

What Is Linux Kernel? (2016, October 31). Retrieved May 31, 2019, from linuxandubuntu.com website: http://www.linuxandubuntu.com/home/what-is-linux-kernel