ASSIGNMENT 4 - REPORT

Strace:

1. It intercepts and records the system calls which are called by a process during its execution.
2. The trace system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.
3. We define a new system call in user/user.h
4. A New Entry is defined in user/usys.pl
5. We define a new system call number in syscall.h

Sigint and Sigalarm:

1. Similar to the strace functionality.
2. We add variables number of ticks, alarm trapframe, is alarm working, current ticks to struct proc in proc.h
3. We initialize the variables in proc.c
4. We add new system call numbers for the given syscalls, in syscall.c

First-Come-First-Serve:

1. Firstly, we modify the make file, by changing the SCHEDULER definition. This is common for all the scheduler modifications
2. The Basic Principle is the selection of the process with the least creation time (i.e, the Process that came first)

3. We basically do this by introducing a new variable that denotes time in the proc.h/c files, and initialize it to ticks.
4. We also define a new variable n_proc to the struct proc.
5. In the scheduler function, we find the process with the least creation time and assign the CPU to that process.
6. We change the processor to switch from other processes to this and increase the n_proc count.
7. We also disable the preemption of processes after the clock interrupts in trap.c
8. We do all of this by basically using the #ifdef and #endif declarations, whenever we write something specifically for our FCFS we write the proprietary code under the given declarations.

<u>Priority Based:</u>

1. The Basic Principle behind the PBS is the selection of the processes with the highest priority
2. When two processes have the same priority, the number of times the process has been scheduled is used to determine the priority.
3. In case of a tie, we give the priority to the process which was started first.
4. We add a sys_set_priority system call in all the appropriate files, similar to the first spec.
5. We add a variable stime (how long the process was sleeping for), nrun (number of times the process was picked by the scheduler), niceness, stp, etc.

6. We Modify the scheduler function, in which we initialize a new pointer to the process. We then assign this to the process with the highest priority.
7. Add new functions, as required in the scheduler function.
8. Make changes to the clockintr() function of proc.c to track runtimes and wait times.

<u>MultiLevel Feedback Queue:</u>

1. MLFQ is a preemptive scheduling policy that allows processes to move between various priority queues. If a process uses too much CPU time, it is pushed to a lower priority queue. Basically it leaves input/output heavy processes and interactive processes in the higher priority queues.
2. To Prevent Starvation, aging is implemented
3. Make appropriate changes
4. Aging has been implemented, just before scheduling, via a simple for loop that iterates through the runnable processes, and promotes them to a higher-priority queue if (ticks -) > 16 ticks.
5. Demotion of processes after their time slice has been completed is done in kernel/trap.c, whenever a timer interrupt occurs.
6. If the time spent in the current queue is greater than 2(current_queue_number), then it is demoted (current_queue is incremented).
7. The position of the process in the queue is determined by its struct proc::entry_time, which stores the entry time in the current queue. It is reset to the current time

whenever it is scheduled, making the wait time in the queue 0. If it is relinquished by the CPU, its entry time is again reset to the current number of ticks.

Lottery Based:

1. We implement a preemptive scheduler that assigns a time slice to the process randomly in proportion to the tickets it owns.
2. We implement a system call that sets the number of tickets of the calling process.
3. Each child inherits the same number of tickets as its parents.
4. All the other changes have been taken care of in the xv6 modification.

Copy on write fork:

1. The Main Idea behind the copy on write is that when a parent process creates a child process, initially they share all the same pages in memory.
2. If either process wants to modify the shared pages then only a copy of these pages will be created as opposed to a regular fork, which always creates copies.
3. Therefore this is basically a way to optimize the fork system call.
4. Modify the uvmcopy() to map the parent's physical pages into the child, instead of allocating new pages. To do this we clear the PTE_W in the PTEs of both child and parent.
5. Modify usertrap() to recognize page faults. When a page-fault occurs on a COW page, allocate a new page

with kalloc(), copy the old page to the new page, and install the new page in the PTE with PTE_W set.

6. We just allocate a page in the cow page and map the new page to the pagetable. We need to find kfree() to free the previous page which is a cow page if no process owns it.
7. Modify the copyout() to use the treat cow pages as page faults.
8. Make the additional changes to make cow, work.