CS 446: Project 3, Scheduling

Due on March 22, 2024

Sara Davis Section 1001

Christopher Howe

Instructions

This project should be ran at least once on a machine running linux on bare metal in order to get accurate results (no VM)

Question 1

Instructions

Run the program with the same number of threads as there are CPUs on the machine. What do you observe?

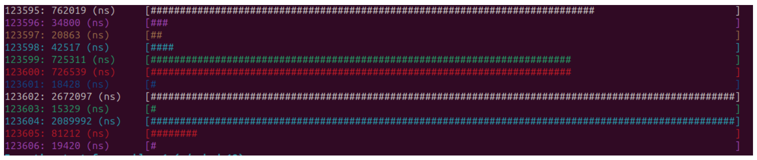
Answer

output of ‘nproc -all‘’

Based on the output of ‘nproc –all‘, my CPU has 12 cores. However, it is worth noting this includes 6 virtual cores for each real core as well.

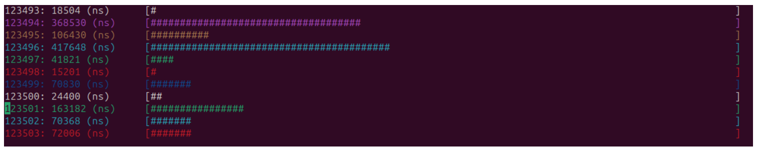
Observations

I noticed that when I ran the program with the same number of threads as there are CPUs on the machine (12), the maximum latency jumped in a huge way. When running the program with 11 cores, I saw an average max latency of around 500,000 ns (.5ms). 300,000 ns (0.3ms).



```
123595: 762019 (ns)     [############################################################################                        ]
123596: 34800 (ns)      [###                                                                                                 ]
123597: 20863 (ns)      [##                                                                                                  ]
123598: 42517 (ns)      [####                                                                                                ]
123599: 725311 (ns)     [########################################################################                            ]
123600: 726539 (ns)     [########################################################################                            ]
123601: 18428 (ns)      [#                                                                                                   ]
123602: 2672097 (ns)    [####################################################################################################]
123603: 15329 (ns)      [#                                                                                                   ]
123604: 2089992 (ns)    [####################################################################################################]
123605: 81212 (ns)      [########                                                                                            ]
123606: 19420 (ns)      [#                                                                                                   ]
```

Figure 1: Output progress bars when sched executable is ran on 12 cores



```
123493: 18504 (ns)      [#                                                                                                   ]
123494: 368530 (ns)     [####################################                                                                ]
123495: 106430 (ns)     [##########                                                                                          ]
123496: 417648 (ns)     [#########################################                                                           ]
123497: 41821 (ns)      [####                                                                                                ]
123498: 15201 (ns)      [#                                                                                                   ]
123499: 70830 (ns)      [#######                                                                                             ]
123500: 24400 (ns)      [##                                                                                                  ]
123501: 163182 (ns)     [################                                                                                    ]
123502: 70368 (ns)      [#######                                                                                             ]
123503: 72006 (ns)      [#######                                                                                             ]
```

Figure 2: Output progress bars when sched executable is ran on 11 cores

Question 2

Instructions

Run the program again with the same number of threads. At the same time run ‘watch -n .5 grep ctxt /proc/<pid>/status‘. This command outputs the number of voluntary and involuntary context switches that a process has undergone, updating every 0.5 seconds. PID of sched.c should be printed by the sched executable by the ‘$print_progress$‘ function. Preform this procedure with both the real time and normal scheduling policies Preform this procedure with a few different priority levels. Report the commands used to find find different results. Report any observations.

Answer

Bash Script used to test question 2

```bash
#!/bin/bash
iterate_tests(){
    realtime_RR(){
        local prio=$1
        local pid=$2
        sudo chrt -p -r $prio $pid
        chrt -p $pid
    }
    realtime_FIFO(){
        local prio=$1
        local pid=$2
        sudo chrt -p -f $prio $pid
        chrt -p $pid
    }
    normal_scheduling(){
        pid=$1
        sudo chrt -p -o 0 $pid
        chrt -p $pid
    }
    sleep 1
    PID=$(pgrep sched)
    echo Running tests for sched with PID=$PID
    #Get the SPIDs (Thread IDs) associated with the process ID 172216
    spids=($(ps -T -p $PID | awk 'NR > 1 {print $2}'))
    # Print the array elements (SPIDs)
    echo sched SPIDs \(TIDs\)
    echo "${spids[@]}"
    TID=${spids[1]}
    echo Watching Thread 0 \($TID\) and adjusting prios
    sleep 1 # wait for executable to start
    gnome-terminal -- ./get-average-ctx-switches.sh $TID
    LOW_PRIO="1"
    MED_PRIO="50"
    HIGH_PRIO="99"
    read -p "Press enter to start real time with round robin and low priority"
    realtime_RR $LOW_PRIO $TID
    read -p "Press enter to start real time with round robin and medium priority"
    realtime_RR $MED_PRIO $TID
    read -p "Press enter to start real time with round robin and high priority"
    realtime_RR $HIGH_PRIO $TID
    read -p "Press enter to start real time with FIFO and low priority"
    realtime_FIFO $LOW_PRIO $TID
    read -p "Press enter to start real time with FIFO and medium priority"
    realtime_FIFO $MED_PRIO $TID
    read -p "Press enter to start real time with FIFO and high priority"
    realtime_FIFO $HIGH_PRIO $TID
    read -p "Press enter to start normal scheduling"
    normal_scheduling $TID
    read -p "Press Enter to exit..."
}
main(){
    export -f iterate_tests
    info "Executing test for problem 2."
    THIS=$BASH_SOURCE
    gnome-terminal --window -- bash -c 'iterate_tests; bash'
}
main $1
```

Observations

While running the test defined in the bash script I noticed that the number of context switches in the FIFO implementation was the lowest. In real time FIFO scheduling the number of context switches is minimal since the only process that can preempt a FIFO running process is a process with a higher priority. The algorithm does not periodically preempt them. In running the RR scheduling I noticed that the number of context switches was higher than the FIFO implementation. This is most likely due to the fact that every time quanta, round robin causes a time switch to the next process in the same queue. However, RR did have lower latency values that FIFO since every process got the CPU faster.

Question 3

Instructions

Run the program again with the same number of Threads. Create cpuset named system with all CPUs except 1. This is described in the background section containing the Linux Multi-Processor Control subsection that contains Method b. Move all Tasks (all User-Level and Kernel-Level ones that are possible to move) into that set. Create a cpuset named dedicated with only the CPU that is excluded from the system cpuset. Move one of the Threads of sched.c to the dedicated cpuset. What sequence of commands did you use to answer this question? What did you observe?

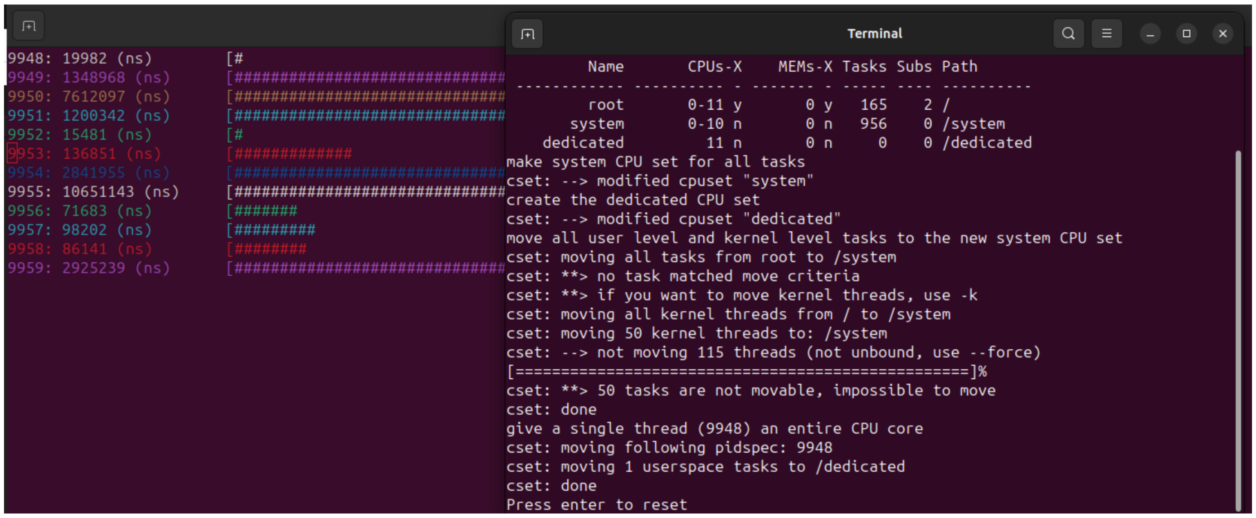
Answer

Bash Script to answer question 3

```
#!/bin/bash
set -e
run_test(){
    sleep 1
    PID=$(pgrep sched)
    echo Running tests for sched with PID=$PID
    #Get the SPIDs (Thread IDs) associated with the process ID
    spids=($(ps -T -p $PID | awk 'NR > 1 {print $2}'))
    # Print the array elements (SPIDs)
    echo sched SPIDs \(TIDs\)
    echo "${spids[@]}"
    echo active cpu sets
    sudo cset set -l
    echo make system CPU set for all tasks
    sudo cset set -c 0-10 system
    echo create the dedicated CPU set
    sudo cset set -c 11 dedicated
    echo move all user level and kernel level tasks to the new system CPU set
    sudo cset proc -m -f root -t system
    sudo cset proc -k -f root -t system
    echo give a single thread \(${spids[1]}\) an entire CPU core
    sudo cset proc -m -p ${spids[1]} -t dedicated
    read -p "Press enter to reset"
    cset set -d dedicated
    cset set -d system
}
main(){
    export -f run_test
    gnome-terminal --window -- bash -c 'run_test; bash'
}
main $1
```

Observations

I was able to successfully allocate an entire core to one thread of the CPU. When this operation was preformed, the max latency of the associated thread never rose too high. This makes sense since if one thread has an entire CPU there are no other tasks besides some few kernel tasks also using the same CPU.



```
9948: 19982 (ns)        [#
9949: 1348968 (ns)      [##############################
9950: 7612097 (ns)      [##############################
9951: 1200342 (ns)      [##############################
9952: 15481 (ns)        [#
9953: 136851 (ns)       [#############
9954: 2841955 (ns)      [##############################
9955: 10651143 (ns)     [##############################
9956: 71683 (ns)        [#######
9957: 98202 (ns)        [#########
9958: 86141 (ns)        [########
9959: 2925239 (ns)      [##############################
```

```
        Name       CPUs-X    MEMs-X Tasks Subs Path
------------ ---------- - ------- - ----- ---- ----------
        root       0-11 y       0 y   165    2 /
      system       0-10 n       0 n   956    0 /system
   dedicated         11 n       0 n     0    0 /dedicated
make system CPU set for all tasks
cset: --> modified cpuset "system"
create the dedicated CPU set
cset: --> modified cpuset "dedicated"
move all user level and kernel level tasks to the new system CPU set
cset: moving all tasks from root to /system
cset: **> no task matched move criteria
cset: **> if you want to move kernel threads, use -k
cset: moving all kernel threads from / to /system
cset: moving 50 kernel threads to: /system
cset: --> not moving 115 threads (not unbound, use --force)
[==================================================]%
cset: **> 50 tasks are not movable, impossible to move
cset: done
give a single thread (9948) an entire CPU core
cset: moving following pidspec: 9948
cset: moving 1 userspace tasks to /dedicated
cset: done
Press enter to reset
```

Figure 3: Output from Question 3 tests

Question 4

Instructions

Run the same procedure as question 3 but overserve the context switches using 'watch -n .5 grep ctxt /proc/<pid>/status'. Preform this procedure with both the real time and normal scheduling policies Preform this procedure with a few different priority levels. Report the commands used to find find different results. Report any observations.

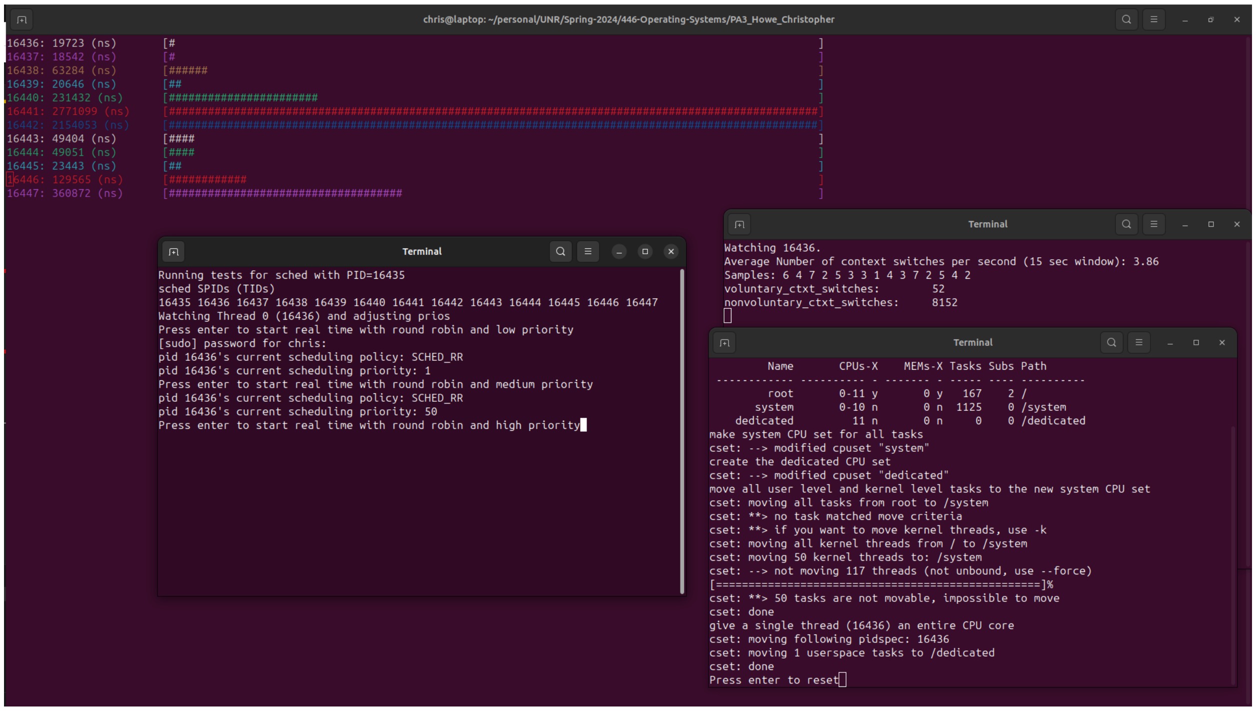
Answer

Commands Ran

In order to preform these tasks, I combined the scripts I wrote for parts 2 and part 3.

Observations

The results were similar to those obtained in part 2 except the first thread (with the dedicated CPU) did not undergo any of the issues either with FIFO or RR. Ie the first thread did not see increased context switches while using RR or a increase in latency while using FIFO. I did also notice at this stage that sometimes the entire sched executable would freeze for about half a second. I think this may be due to the fact that there is a decrease in the number of available cores and the 5 other real cores were not able to keep up with everything else running



```
chris@laptop: ~/personal/UNR/Spring-2024/446-Operating-Systems/PA3_Howe_Christopher
16436: 19723 (ns)       [#                                                                                                  ]
16437: 18542 (ns)       [#                                                                                                  ]
16438: 63284 (ns)       [######                                                                                             ]
16439: 20646 (ns)       [##                                                                                                 ]
16440: 231432 (ns)      [#######################                                                                            ]
16441: 2771099 (ns)     [###################################################################################################]
16442: 2154053 (ns)     [###################################################################################################]
16443: 49404 (ns)       [####                                                                                               ]
16444: 49051 (ns)       [####                                                                                               ]
16445: 23443 (ns)       [##                                                                                                 ]
16446: 129565 (ns)      [############                                                                                       ]
16447: 360872 (ns)      [####################################                                                               ]
```

```
Terminal
Running tests for sched with PID=16435
sched SPIDs (TIDs)
16435 16436 16437 16438 16439 16440 16441 16442 16443 16444 16445 16446 16447
Watching Thread 0 (16436) and adjusting prios
Press enter to start real time with round robin and low priority
[sudo] password for chris:
pid 16436's current scheduling policy: SCHED_RR
pid 16436's current scheduling priority: 1
Press enter to start real time with round robin and medium priority
pid 16436's current scheduling policy: SCHED_RR
pid 16436's current scheduling priority: 50
Press enter to start real time with round robin and high priority
```

```
Terminal
Watching 16436.
Average Number of context switches per second (15 sec window): 3.86
Samples: 6 4 7 2 5 3 3 1 4 3 7 2 5 4 2
voluntary_ctxt_switches:        52
nonvoluntary_ctxt_switches:     8152
```

```
Terminal
         Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         root       0-11 y       0 y   167    2 /
       system       0-10 n       0 n  1125    0 /system
    dedicated         11 n       0 n     0    0 /dedicated
make system CPU set for all tasks
cset: --> modified cpuset "system"
create the dedicated CPU set
cset: --> modified cpuset "dedicated"
move all user level and kernel level tasks to the new system CPU set
cset: moving all tasks from root to /system
cset: **> no task matched move criteria
cset: **> if you want to move kernel threads, use -k
cset: moving all kernel threads from / to /system
cset: moving 50 kernel threads to: /system
cset: --> not moving 117 threads (not unbound, use --force)
[==================================================]%
cset: **> 50 tasks are not movable, impossible to move
cset: done
give a single thread (16436) an entire CPU core
cset: moving following pidspec: 16436
cset: moving 1 userspace tasks to /dedicated
cset: done
Press enter to reset
```

Figure 4: Output demonstrating combined efforts of part 2 and part 3 (Different scheduling policies selected with an isolated CPU)

Suplemental scripts

get-average-ctx-switches.sh

```bash
#!/bin/bash
pid=$1
get_total_ctxt_switches() {
    local pid=$1
    local voluntary=$(grep "^voluntary_ctxt_switches" /proc/"$pid"/status | awk '{print $2}')
    local nonvoluntary=$(grep "nonvoluntary_ctxt_switches" /proc/"$pid"/status | awk '{print $2}')
    echo $((voluntary + nonvoluntary))
}
samples=()
num_samples=0
lastTickVal=$(get_total_ctxt_switches $pid)
while true; do
    currentTotalCTXSwitches=($(get_total_ctxt_switches $pid))
    difference=$((currentTotalCTXSwitches- lastTickVal))
    lastTickVal=$currentTotalCTXSwitches
    samples+=($difference)
    if [ $num_samples -eq "15" ]; then
        samples=("${samples[@]:1}")
    else
        num_samples=$((num_samples + 1))
    fi
    sum=0
    for val in "${samples[@]}"; do
        sum=$((sum + val))
    done
    clear
    tput cup 0 0
    echo Watching $pid.
    echo Average Number of context switches per second \(15 sec window\): $(echo "scale=2; $sum / $num_
    echo Samples: ${samples[@]}
    grep ctxt /proc/"$pid"/status
    sleep 1
done
```

test.sh

```
#!/bin/bash
set -e
EXECUTABLE_NAME=sched
cd $PWD
make
info(){
    teal_color='\e[1;36m'
    white_color='\e[0m'
    echo -e "${teal_color}$@${white_color}"
}
NUM_CORES=$(nproc --all)
echo Your CPU has $NUM_CORES Cores.
if [ -z "$1" ]; then
    echo please specify a problem demonstration to run.
    echo Problem 1: Run sched exec with same number of cores as CPU \($NUM_CORES\).
    echo Problem 2: Opens up three gnome terminal windows and runs the executable with some different s
    echo First window shows the running process latencies
    echo Second window shows the context switch watch output
    echo Third window shows which test is currently running.
    echo Tests ran: Realtime Round Robin Scheduling Real time FIFO Scheduling, Normal Scheduling, all wi
    echo Problem 3: Run with one thread having its own CPU
    echo Problem 4: Run with combined steps of part 2 and 3
    exit
fi
if [ "$1" -eq 1 ]; then
    info Executing test for problem 1 \(./sched $NUM_CORES\)
    ./sched 12
    exit
fi
if [ "$1" -eq 2 ]; then
    ./swap-scheduler-and-watch.sh
    ./sched 12
    exit
fi
if [ "$1" -eq 3 ]; then
    ./cpuset.sh
    ./sched 12
    exit
fi
if [ "$1" -eq 4 ]; then
    ./swap-scheduler-and-watch.sh
    ./cpuset.sh
    ./sched 12
    exit
fi
```