PSET 7

Jonathan Lam
Prof. Hakner
Computer OS
12/15/19

PROBLEM 1 - Some assembly required

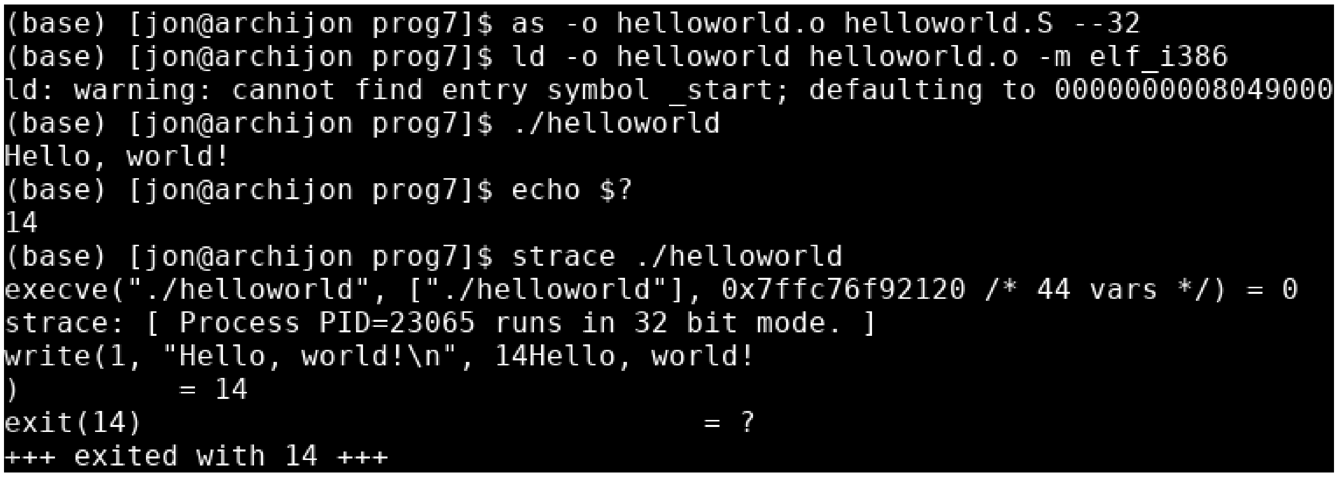
helloworld.S

```
.data
  str:  .string "Hello, world\!\n"

.text
  # write(1, "Hello, world!\n", 14)
  movl  $0x4,%eax   # write syscall# = 4
  movl  $0x1,%ebx   # fd = 1
  movl  $str,%ecx   # addr of string "Hello, world!\n"
  movl  $14,%edx    # len of string "Hello, world!\n"
  int   $0x80       # invoke syscall

  # exit(0)
  movl  %eax,%ebx   # exit code = retval of write
  movl  $0x1,%eax   # exit syscall# = 1
  int   $0x80       # invoke syscall
```

Screenshot of terminal output for compilation/linking/running/strace-ing:



```
(base) [jon@archijon prog7]$ as -o helloworld.o helloworld.S --32
(base) [jon@archijon prog7]$ ld -o helloworld helloworld.o -m elf_i386
ld: warning: cannot find entry symbol _start; defaulting to 0000000008049000
(base) [jon@archijon prog7]$ ./helloworld
Hello, world!
(base) [jon@archijon prog7]$ echo $?
14
(base) [jon@archijon prog7]$ strace ./helloworld
execve("./helloworld", ["./helloworld"], 0x7ffc76f92120 /* 44 vars */) = 0
strace: [ Process PID=23065 runs in 32 bit mode. ]
write(1, "Hello, world!\n", 14Hello, world!
)         = 14
exit(14)                                = ?
+++ exited with 14 +++
```

PROBLEM 2 – Scheduling

The following program can be called in the form given in the assignment (./nicetest [num_proc] [nice_val] [test_time]) or without arguments to generate a CSV file with results from testing a

nicetest.c

```c
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <unistd.h>

#define PROG "nicetest"

// these are for generating the CSV; time is constant and num_proc is varied
// from 2 to PROC_MAX
#define TEST_TIME 5
#define PROC_MAX 16

// somewhat relaxed error checking, as this program is exploratory and not
// meant to be rigorously user-tested (i.e., user-abused)
#define ERR_FAT(cmd, ctx, msg) {\
  /*check errno b/c some syscalls (e.g., nice) may have a non-err neg retval*/\
  if(cmd<0 && errno) {\
    dprintf(2, "%s: %s: %s\n", PROG, ctx, msg);\
    kill(0, SIGKILL); /*kill other children if called from child*/\
    exit(EXIT_FAILURE);\
  }\
}
#define ERRNO_FAT(cmd, ctx) ERR_FAT(cmd, ctx, strerror(errno));
#define ERRMSG_FAT(cond, msg)\
  if(cond) {\
    dprintf(2, "%s: %s\n", PROG, msg);\
    exit(EXIT_FAILURE);\
  }

double run_nice_test(int num_proc, int nice_val, int test_time) {
  int i, wstatus, pid, ndn_pid; // non-default nice pid
  long tot_usec, ndn_usec, tmp_cum_usec;
```

```
static long cum_usec;         // cumulative time (since rusage is per-proc)
double tim_pcnt;
struct rusage ndn_ru, tot_ru;

for(i=0; i<num_proc; i++) {
  switch(pid=fork()) {
  case -1:
    dprintf(2, "err fork\n");
    exit(EXIT_FAILURE);
    break;
  case 0:
    signal(SIGTERM, SIG_DFL);
    if(!i) {
      errno = 0;
      ERRNO_FAT(nice(nice_val), "setting nice value");
    }
    while(1);
  default:
    if(!i)
      ndn_pid = pid;
  }
}

// since this is called after forks began, total time may be slightly larger
// than expected total time (i.e., (# processors)*(expected time))
ERRNO_FAT(-sleep(test_time), "sleep-ing for test_time seconds");

// ignore sigterm in parent
signal(SIGTERM, SIG_IGN);

// get ndn pid; unfortunately, may cause this to exit slightly earlier than
// others; forgot about existence of wait4 and should have used that in
// hindsight
ERRNO_FAT(kill(ndn_pid, SIGTERM), "kill ndn child with SIGTERM");
ERRNO_FAT(waitpid(ndn_pid, &wstatus, 0), "waitpid for ndn child");
ERRNO_FAT(getrusage(RUSAGE_CHILDREN, &ndn_ru), "getrusage for ndn child");

// kill rest of children and get rusage
// kill 0 sends signal to everything in process group
ERRNO_FAT(kill(0, SIGTERM), "kill non-ndn child");
for(i=1; i<num_proc; i++)
  ERRNO_FAT(wait(&wstatus), "wait for non-ndn child");
ERRNO_FAT(getrusage(RUSAGE_CHILDREN, &tot_ru), "getrusage for non-ndn child");

tmp_cum_usec = cum_usec;
```

```c
  ndn_usec = (ndn_ru.ru_utime.tv_sec+ndn_ru.ru_stime.tv_sec)*1000000
             +ndn_ru.ru_utime.tv_usec+ndn_ru.ru_stime.tv_usec-cum_usec;
  tot_usec = (tot_ru.ru_utime.tv_sec+ndn_ru.ru_stime.tv_sec)*1000000
             +tot_ru.ru_utime.tv_usec+ndn_ru.ru_stime.tv_usec-cum_usec;
  cum_usec = tmp_cum_usec+tot_usec;

  tim_pcnt = ((double)ndn_usec)/tot_usec*100;
  dprintf(2, "=====\n"
            "Num processes:\t%d\n"
            "Task0 nice val:\t%d\n"
            "Test time:\t\t%d\n"
            "Total CPU time:\t%ldus\n"
            "Task0 CPU time:\t%ldus\n"
            "Task0 CPU %:\t%lf%\n",
          num_proc, nice_val, test_time,
          tot_usec, ndn_usec, tim_pcnt);
  return tim_pcnt;
}

int nice_test_all() {
  int ofd, num_proc, nice_val, test_time;
  char buf[20];

  dprintf(2, "=====\nWriting output to niceout.csv\n");

  ERRNO_FAT((ofd=open("niceout.csv", O_WRONLY|O_TRUNC|O_CREAT, 0644)),
           "Opening outfile");

  strncpy(buf, "num_proc,", 10);
  // no need to check partial write cond
  ERRNO_FAT(write(ofd, buf, strlen(buf)), "Writing to outfile");
  for(nice_val=-20; nice_val<20; nice_val++) {
    sprintf(buf, "%d,", nice_val);
    ERRNO_FAT(write(ofd, buf, strlen(buf)), "Writing to outfile");
  }
  strncpy(buf, "\n", 2);
  ERRNO_FAT(write(ofd, buf, strlen(buf)), "Writing to outfile");

  for(num_proc=2; num_proc<=PROC_MAX; num_proc++) {
    sprintf(buf, "%d,", num_proc);
    ERRNO_FAT(write(ofd, buf, strlen(buf)), "Writing to outfile");

    for(nice_val=-20; nice_val<20; nice_val++) {
      sprintf(buf, "%.5f,", run_nice_test(num_proc, nice_val, TEST_TIME));
      ERRNO_FAT(write(ofd, buf, strlen(buf)), "Writing to outfile");
    }

    strncpy(buf, "\n", 2);
    ERRNO_FAT(write(ofd, buf, strlen(buf)), "Writing to outfile");
  }

  ERRNO_FAT(close(ofd), "Closing outfile");
  exit(EXIT_SUCCESS);
}

int main(int argc, char **argv) {
  // time in seconds, nice values from -20 to 19
  // nice values will be validated; time (in seconds) will only checked to be
  // positive but should be a value >5 to see discernible results
  int num_proc, nice_val, test_time;

  dprintf(2, "=====\nnicetest.c\n");

  // if called with no args, run series of tests and generate csv
  if(argc==1)
    return nice_test_all();

  // else just run single test with given params
  ERRMSG_FAT(argc<4, "Usage: nicetest [num_proc] [nice_val] [test_time]");
  ERRMSG_FAT((num_proc=atoi(argv[1]))<1,
             "num_proc must be a positive integer");
  ERRMSG_FAT((nice_val=atoi(argv[2]))<-20||nice_val>19,
             "nice number must be in range [-20,19]");
  ERRMSG_FAT((test_time=atoi(argv[3]))<1,
             "test_time must be a positive integer");

  run_nice_test(num_proc, nice_val, test_time);
}
```

Test run outputs

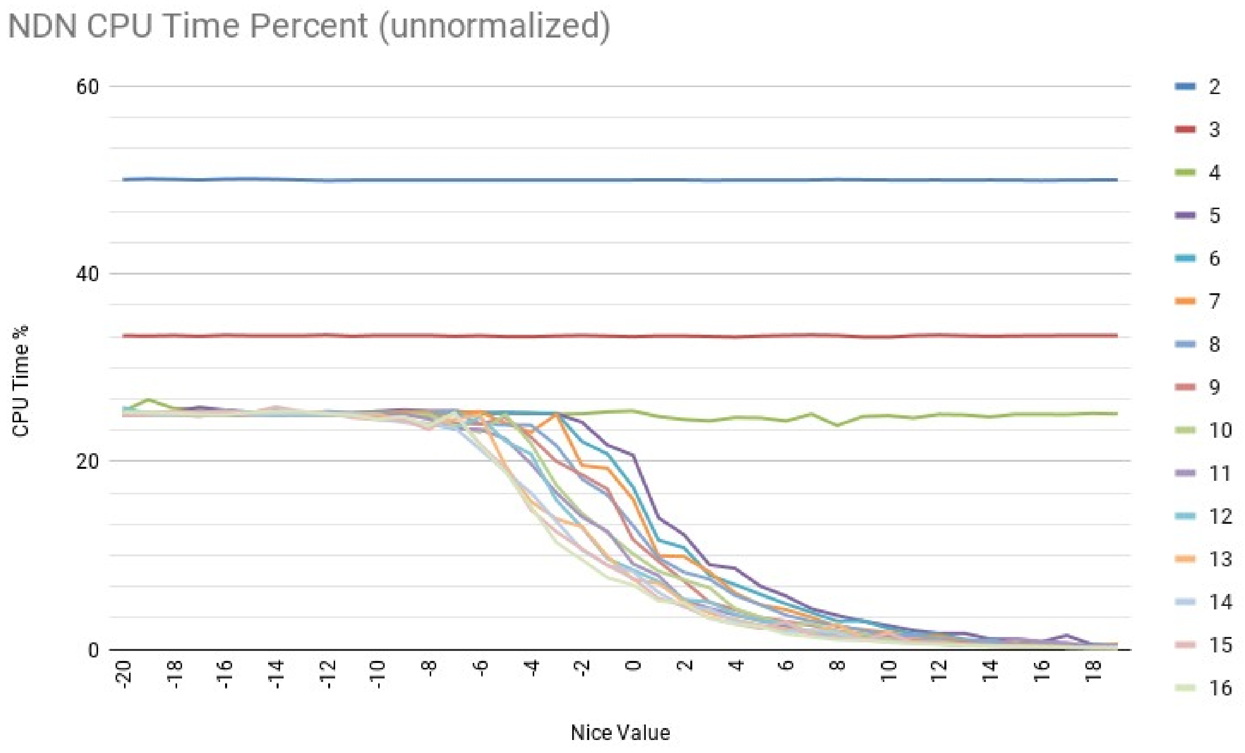
Note: Test CPU: Intel i7 7500U (4 logical cores)
(usage: ./nicetest [num_proc] [nice_val] [test_time])

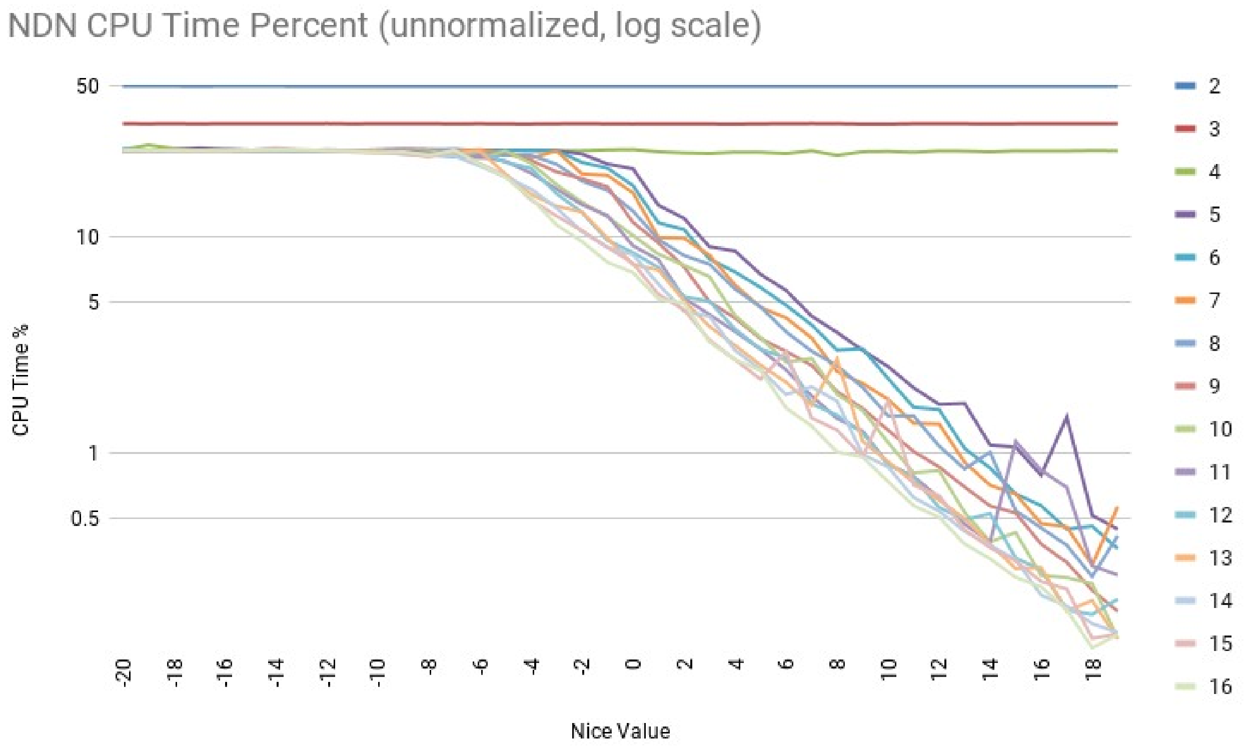
```
(base) [jon@archijon prog7]$ ./nicetest 16 0 5
=====
nicetest.c
=====
Num processes:    16
Task0 nice val:   0
Test time:        5
Total CPU time:   19250036us
Task0 CPU time:   1193019us
Task0 CPU %:      6.197490%
(base) [jon@archijon prog7]$ ./nicetest 16 10 5
=====
nicetest.c
=====
Num processes:    16
Task0 nice val:   10
Test time:        5
Total CPU time:   18173635us
Task0 CPU time:   137645us
Task0 CPU %:      0.757388%
```

These are close to the demo’s values shown of 6.25549% and 0.69258%, respectively.

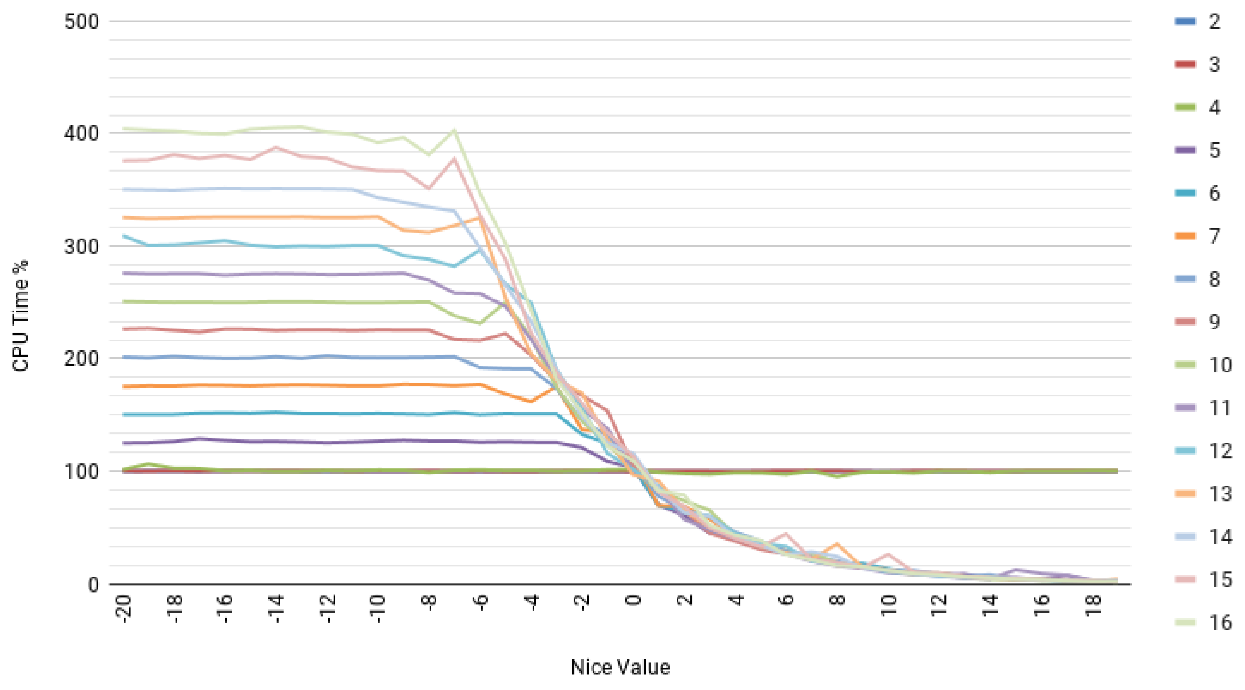
Test run outputs

```
(usage: ./nicetest)
```

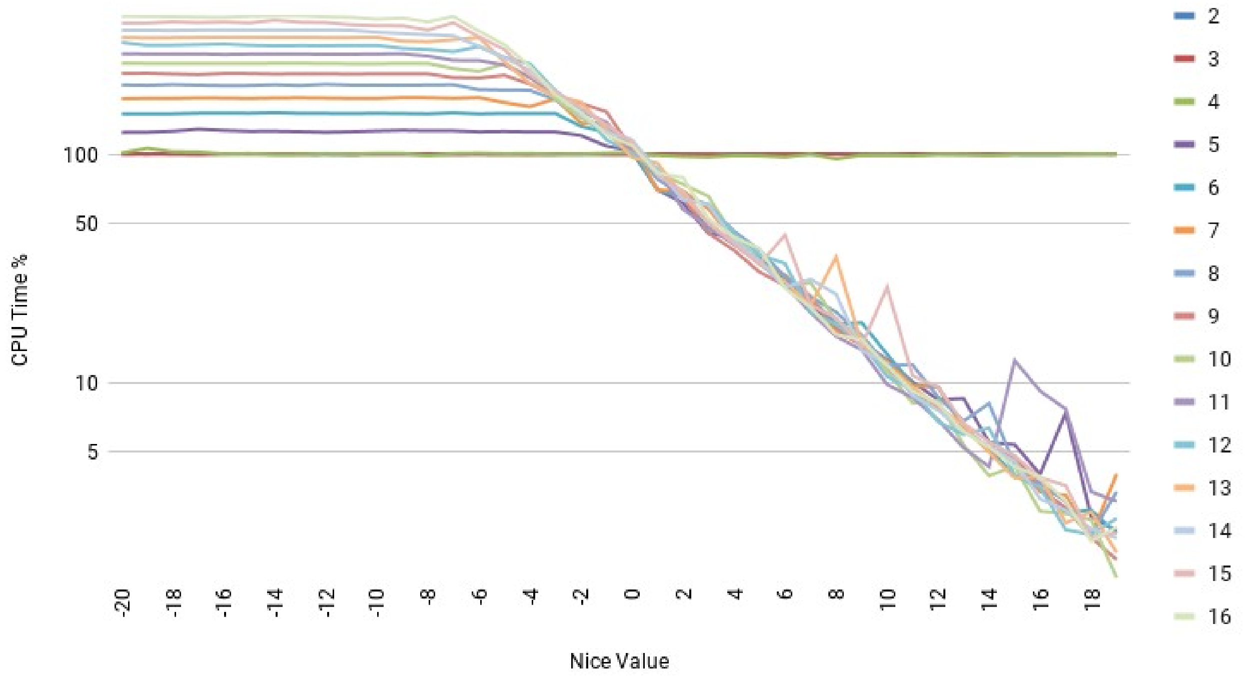




NDN CPU Time Percent (normalized)



NDN CPU Time Percent (normalized, log scale)



Explanation of the charts:

The graphs are plots of CPU time share (%) (CTS) vs. nice value of the non-default nice (NDN) process. Each line is a given value for number of processors (NP) used in the test. These are generated directly from Google Sheets by loading the generated niceout.csv file generated when calling niceout.c without any arguments.

The first two plots are non-normalized (i.e., the plotted CTS are exactly the CTS calculated as (NDN proc time)/(total CPU time for all processes)). In the latter two, they are normalized to the number of processors using spreadsheet magic (i.e., the plotted values are (NDN proc time)/(total CPU time for all processes)*(NP)). Thus, the “100%” value in the normalized graphs indicates the CTS for one of N processes if all had the same nice value; anything higher means the process had more than average CPU share, and anything lower means the process had less. We can call this the “expected” CTS for one of N processes.

One noticeable aspect of the graphs is that for N∈[2,4], there is virtually no variation from the expected CTS value. This can be expected on a system with four logical cores (which the test environment has) that doesn’t have much else running (the test environment was relatively “quiet” other than the test) since little scheduling happens. Not much is really competing for CPU time other than these processes, so all get roughly the same amount of CPU time.

However, for any NP>5, there is a clear correlation between CTS and nice value. There appears to be a roughly-constant region until some negative cutoff nice value (we can denote this NCO, “nice cutoff” value). This cutoff value may be explained by the CPU allowing a minimum scheduling time for the other processes – after a certain point, it doesn’t allow the high-priority NDN process to monopolize the CPU. This can better be seen with the unnormalized plots: for NP>4 (when the nice values start affecting the CTS), we see that the maximum unnormalized CTS is capped at roughly 25%. (Given that this is a quad-logical core CPU, this makes sense – no process can be using more than one CPU simultaneously).

It appears that the drop is inverse exponential when nice value>NCO, as expected, asymptoting towards 0. The reason for this asymptote is obvious.