Achieving Multiprocessing

Objective

- Students will be able to get and set the processor's affinity mask through UNIX commands and programmatically.
- Students will be able to schedule their process on a dedicated CPU.

To exploit multiprocessing, a scheduler can be forced to schedule a process on a particular CPU. Nowadays, CPUs contain multiple cores with hyper threading. Each core and hyper threading capability is deemed as a separate CPU to the operating system.

Processor affinity or CPU pinning enables the binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU.

Scheduling-algorithm implementations vary in adherence to processor affinity. Under certain circumstances, some implementations will allow a task to change to another processor if it results in higher efficiency. For example, when two processor-intensive tasks (A and B) have affinity to one processor while another processor remains unused, many schedulers will shift task B to the second processor in order to maximize processor use. Task B will then acquire affinity with the second processor, while task A will continue to have an affinity with the original processor.

On Linux, the CPU affinity of a process can be altered with the taskset command and the sched_setaffinity(2) system call.

Taskset

SYNOPSIS

```
taskset [options] mask command [argument...]

taskset [options] -p [mask] pid
```

DESCRIPTION

taskset is used to set or retrieve the CPU affinity of a running process given its PID, or to launch a new command with a given CPU affinity. CPU affinity is a scheduler property that "bonds" a process to a given set of CPUs on the system. The Linux scheduler will honor the given CPU affinity and the process will not run on any other CPUs. Note that the Linux scheduler also supports natural CPU affinity: the scheduler attempts to keep processes on the same CPU as long as practical for performance reasons. Therefore, forcing a specific CPU affinity is useful only in certain applications.

The CPU affinity is represented as a bitmask, with the lowest order bit corresponding to the first logical CPU and the highest order bit corresponding to the last logical CPU. Not all CPUs may exist on a given system but a mask may specify more CPUs than are present. A retrieved mask will reflect only the bits that correspond to CPUs physically on the system. If an invalid mask is given (i.e., one that corresponds to no valid CPUs on the current system) an error is returned. The masks may be specified in hexadecimal (with or without a leading "0x"), or as a CPU list with the --CPU-list option. For example,

0x00000001 is processor #0,

0x00000003 is processors #0 and #1,

0xFFFFFFFF is processors #0 through #31,

How to display the affinity mask for a process by its PID

```
adnan@AdnanPc:/mnt/c/code$ ps
  PID TTY          TIME CMD
   10 tty1     00:00:00 bash
  141 tty1     00:00:00 ps
adnan@AdnanPc:/mnt/c/code$ taskset -p 10
pid 10's current affinity mask: f
```

The current affinity mask is f which is in binary (1111). Each bit in a mask represent a CPU. Least significant bit represent the CPU#0 and most significant bit represents CPU#3. '1' indicates that the CPU is set and '0' indicates not set.

Interpret mask as numerical list of processors instead of a bitmask.

```
adnan@AdnanPc:/mnt/c/code$ ps
  PID TTY          TIME CMD
   10 tty1     00:00:00 bash
  166 tty1     00:00:00 ps
adnan@AdnanPc:/mnt/c/code$ taskset -pc 10
pid 10's current affinity list: 0-3
```

Numbers are separated by commas and may include ranges. For example: 0,5,8-11.

Setting new affinity mask for a given PID.

```
adnan@AdnanPc:/mnt/c/code$ taskset -p 1 10
pid 10's current affinity mask: f
pid 10's new affinity mask: 1
```

Running a program with new affinity mask

```
adnan@AdnanPc:/mnt/c/code$ ps
  PID TTY          TIME CMD
   10 tty1     00:00:00 bash
  200 tty1     00:00:00 ps
adnan@AdnanPc:/mnt/c/code$ taskset 1 ls / -l
total 580
drwxr-xr-x  1 root root    512 Oct 26 14:34 bin
drwxr-xr-x  1 root root    512 Jul 25  2018 boot
drwxr-xr-x  1 root root    512 Nov 19 10:12 dev
drwxr-xr-x  1 root root    512 Nov 19 10:12 etc
drwxr-xr-x  1 root root    512 Nov 24  2018 home
-rwxr-xr-x  1 root root 591344 Jan  1  1970 init
drwxr-xr-x  1 root root    512 Nov 24  2018 lib
drwxr-xr-x  1 root root    512 Jul 25  2018 lib64
drwxr-xr-x  1 root root    512 Jul 25  2018 media
drwxr-xr-x  1 root root    512 Nov 24  2018 mnt
drwxr-xr-x  1 root root    512 Jul 25  2018 opt
dr-xr-xr-x  9 root root      0 Nov 19 10:12 proc
drwx------  1 root root    512 Jul 25  2018 root
drwxr-xr-x  1 root root    512 Nov 19 10:12 run
drwxr-xr-x  1 root root    512 Oct 26 14:34 sbin
drwxr-xr-x  1 root root    512 Jul 19  2018 snap
drwxr-xr-x  1 root root    512 Jul 25  2018 srv
dr-xr-xr-x 12 root root      0 Nov 19 10:12 sys
drwxrwxrwt  1 root root    512 Nov 19 12:48 tmp
drwxr-xr-x  1 root root    512 Jul 25  2018 usr
drwxr-xr-x  1 root root    512 Jul 25  2018 var
adnan@AdnanPc:/mnt/c/code$
```

Display information about the CPU architecture

```
adnan@AdnanPc:/bin$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  4
On-line CPU(s) list:     0-3
Thread(s) per core:      2
Core(s) per socket:      2
Socket(s):               1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   142
Model name:              Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Stepping:                9
CPU MHz:                 2701.000
CPU max MHz:             2701.0000
BogoMIPS:                5402.00
Hypervisor vendor:       Windows Subsystem for Linux
Virtualization type:     container
```

Setting Affinity Programmatically

```
#include <sys/sysinfo.h>

int get_nprocs(void);
int get_nprocs_conf(void);
```

The function **get_nprocs_conf**() returns the number of processors configured by the operating system.

The function **get_nprocs**() returns the number of processors currently available in the system. This may be less than the number returned by **get_nprocs_conf**() because processors may be offline (e.g., on hot pluggable systems).

Displaying the number of processors available in a system

```
#include <sys/sysinfo.h>
#include <stdio.h>

int main(void) {
printf("This system has %d processors configured and "
                    "%d processors available.\n",
                    get_nprocs_conf(), get_nprocs());

return 0;
}
```

```
#include <sched.h>

int sched_getcpu(void);
```

sched_getcpu() returns the number of the CPU on which the calling process is currently executing.

Displaying the number of CPU the current process is running

```
#include <sched.h>
#include <stdio.h>
int main(void) {
printf("This process is currently executing on processor# =
%d ", sched_getcpu());
}
```

```
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t cpusetsize,
                      const cpu_set_t *mask);
```

sched_setaffinity() sets the CPU affinity mask of the process whose ID is *pid* to the value specified by *mask*. If *pid* is zero, then the calling process is used. The argument *cpusetsize* is the length (in bytes) of the data pointed to by *mask*. Normally this argument would be specified as *sizeof(cpu_set_t)*.

If the process specified by *pid* is not currently running on one of the CPUs specified in *mask*, then that process is migrated to one of the CPUs specified in *mask*.

Following macros are used to manipulate the **mask** in **set_affinity().**

void CPU_ZERO(cpu_set_t *set);
Clears *set*, so that it contains no CPUs.

void CPU_SET(int *cpu*, **cpu_set_t *set);**
Add CPU *cpu* to *set*.

void CPU_CLR(int *cpu*, **cpu_set_t *set);**
Remove CPU *cpu* from *set*.

int CPU_ISSET(int *cpu*, **cpu_set_t *set);**
Test to see if CPU *cpu* is a member of *set*.

int CPU_COUNT(cpu_set_t *set);
Return the number of CPUs in *set*.

```
int sched_getaffinity(pid_t pid, size_t cpusetsize,
                      cpu_set_t *mask);
```

sched_getaffinity() writes the affinity mask of the process whose ID is *pid* into the *cpu_set_t* structure pointed to by *mask*. The *cpusetsize* argument specifies the size (in bytes) of *mask*. If *pid* is zero, then the mask of the calling process is returned.

Getting affinity mask of current process and display it.

```
#define _GNU_SOURCE
#include<sched.h>
#include<stdio.h>
#include <unistd.h>
#include <sys/sysinfo.h>
int main()
{
cpu_set_t my_set;        /* Define your cpu_set bit mask. */
CPU_ZERO(&my_set);       /* Initialize it all to 0, i.e. no
CPUs selected. */
sched_getaffinity(0, sizeof(cpu_set_t), &my_set);
/* Get affinity of this process to
/* the defined mask */

int nproc = get_nprocs_conf();
   for (int i = 0; i < nproc; i++)
        printf("Processor# %d = %d \n", i,CPU_ISSET(i,
&my_set)); /* check one by one if the affinity of processor
is in the mask*/
    return 0;
}
```

Setting affinity of current process to CPU#1 & then display

```
#define _GNU_SOURCE
#include<sched.h>
#include<stdio.h>
#include <unistd.h>
#include <sys/sysinfo.h>
int main()
{

cpu_set_t my_set;        /* Define your cpu_set bit mask. */
CPU_ZERO(&my_set);       /* Initialize it all to 0, i.e. no
CPUs selected. */
CPU_SET(1,&myset);  /*setting the affinity to CPU1*/
sched_setaffinity(0, sizeof(cpu_set_t), &my_set);
/* Set affinity of this process to
/* the defined mask, i.e. only 1. */

int nproc = get_nprocs_conf();
   for (int i = 0; i < nproc; i++) /*displaying affinity*/
        printf("Processor# %d = %d \n", i,CPU_ISSET(i,
&my_set)); /* check one by one if the affinity of processor
is in the mask*/
```

```
return 0;
}
```

Putting it altogether!

- **Displaying the process# on which the process is running on before setting the new affinity**
- **Getting & displaying the current affinity of running process**
- **Setting & displaying the new affinity of running process**
- **Displaying the process# on which the process is running on after setting the new affinity**

```
#define _GNU_SOURCE
#include<sched.h>
#include<stdio.h>
#include <unistd.h>
#include <sys/sysinfo.h>

int main(void) {
printf("This process is currently running on processor#
%d\n",sched_getcpu());
cpu_set_t mask;
int nproc = get_nprocs_conf();
sched_getaffinity(0, sizeof(cpu_set_t), &mask);
printf("Current Affinity is\n");
for (int i = 0; i < nproc; i++)
       printf("Processor %d = %d \n",i,CPU_ISSET(i, &mask));


CPU_ZERO(&mask);
CPU_SET(3, &mask);
sched_setaffinity(0, sizeof(cpu_set_t), &mask);

printf("\n\nNew Affinity is\n");
sched_getaffinity(0, sizeof(cpu_set_t), &mask);
for (int i = 0; i < nproc; i++)
       printf("Processor %d = %d \n",i,CPU_ISSET(i, &mask));
printf("\nThis process is currently running on processor#
%d\n",sched_getcpu());

    return EXIT_SUCCESS;
}
```

Graded Tasks

Task 1(to be submitted in the class) Marks 10

Write a C program that displays the number of available CPUs and lets the user select one. After choosing the CPU, whatever command user issues must be run on that CPU. (Hint: Your program would provide an interface between user and shell.)

Task 2 (to be submitted in the class) Marks 20

Write a program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three child processes. One process will calculate and display the average of the numbers, the second will show the maximum and maximum values, and the third will display the list in sorted order. Each process should run on a separate CPU.

Input:

$./a.out 90 81 78 95 79 72 85

Output: (in any order)

The average value is 82

The minimum and maximum values are 72 and 95.

Sorted list: 72 78 79 81 85 90 95

Task 3(Home task to be submitted the next day before 11:55PM) Marks 20

The formula for computing π is given below.

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

With the above formula we can compute the value of π via numerical integration over a large number of points. For example we can choose to use 10 million points. The serial code is shown below.

Serial code of computing π

```
#include <stdio.h>
int main()
{
long int nsteps = 10000000; double x;
double dx = 1.0 / nsteps;
double pi = 0.0;
for(long int i=0;i<nsteps;i++)
    {
     x = (i + 0.5) * dx;
    pi += 4.0 / (1.0 + x * x);
     }
pi *= dx;
printf("%lf",pi);
     return 0;
}
```

To parallelize the serial code for computing π, we need to divide the "for" loop into sub-tasks and distribute them to the worker processes. In other words, we need to evenly distribute the task of evaluating the integrand at 10 million points. This can be conveniently done by providing the start, stop and step arguments.

Write a program that divides the above code in instances equal to the number of CPUs available in a system to compute the value of π. Each instance should run on a separate CPU.