

```
/******  
** Notes on Program Execution Time Measurements on Linux/Unix/Solaris  
**  
** Intended audience: Those who would like to learn more about  
** measuring program execution time in modern computer systems.  
**  
** Used: CPE 631 Advanced Computer Systems and Architectures  
**       CPE 619 Modeling and Analysis of Computer and Communication Systems  
**  
** ver 0.1, Spring 2007 notes on performance measurements (Solaris)  
** ver 0.2, Spring 2011 updated notes on program execution time measurements (Linux)  
** ver 0.3, Spring 2012 updates for execution time measurements in Linux  
** ver 0.4, Spring 2018 updates for execution time measurements on blackhawk (runs CentOS 6.9)  
** ver 0.5, Fall 2020 updates for execution time measurements on blackhawk (runs Red Hat Enterprise  
Linux Server release 7.8 (Maipo)  
  
** @Aleksandar Milenkovic, milenkovic@computer.org  
*****/
```

Program Execution Time Measurements

Let us start with a simple C program that sums up elements of an integer array. The program initializes elements randomly and then sums the elements and prints the result. We would like to measure program execution time.

* Copy this directory into your current directory.

```
<<~~~~~  
-bash-4.1$ cp -r /apps/arch/arch.tut/exetimemeasure .  
-bash-4.2$ cd exetimemeasure/  
~~~~~>>
```

1 Using the date command

* To learn more about the date command type in man date.

```
<<~~~~~  
-bash-4.1$ man date  
~~~~~>>
```

* Inspect the code in file arrsum.c.

* Compile the code using gcc with O2 optimization level.

```
<<~~~~~
```

```
-bash-4.1$ gcc -O2 arrsum.c -o arrsum.exe
~~~~~>>
```

* Measure time using date command as follows.

```
<<~~~~~
-bash-4.2$ date && ./arrsum.exe 1024 && date
Wed Aug 26 13:31:05 CDT 2020
array sum is 1114356211233
Wed Aug 26 13:31:05 CDT 2020
~~~~~>>
```

In this example we get the date&time before the execution of the program (13:31:05) and after execution of the program (13:31:05). Time resolution for the date utility is 1 second, and the arrsum program takes far less time than 1 second to complete. Actually, this utility would not be useful even if we use much larger arrays. For example, if we use 1 million elements instead of 1024, we still cannot see the difference. However, if your program takes more time than 1 second, you will be able to observe differences in time and to determine program execution time.

```
<<~~~~~
-bash-4.2$ date && ./arrsum.exe 1000000 && date
Wed Aug 26 13:32:42 CDT 2020
array sum is 1073756018481283
Wed Aug 26 13:32:42 CDT 2020
~~~~~>>
```

Note: An alternative is that you invoke multiple instances of your program (e.g., 100,000 times) and measure time it takes to execute these instances as a standalone program. For example, you can have a loop that repeats your task as many times as you want. By dividing the elapsed time with the number of repetitions of your task, you can get an estimated execution time for your task.

2 Using the time command

* To learn more the time utility type in man time.

```
<<~~~~~
-bash-4.1$ man time
~~~~~>>
```

* A better alternative for measuring time is using time utility. The time utility prints a message with time statistics about the program run, including

- (i) the elapsed real-time between the program invocation and termination
- (ii) the user CPU time, and
- (iii) the system CPU time.

* Examples of using time are given below:

```
<<~~~~~
```

```
-bash-4.1$ time arrsum.exe 1024
array sum is 1114360406973
```

```
real    0m0.001s
user    0m0.000s
sys     0m0.000s
```

```
-bash-4.1$ time arrsum.exe 2048
array sum is 2205251413056
```

```
real    0m0.001s
user    0m0.000s
sys     0m0.000s
```

```
~~~~~>>
```

* By using the maximum number of elements we can see a change in execution time.

```
<<~~~~~
```

```
-bash-4.1$ time arrsum.exe 1048576
array sum is 1125894357517602
```

```
real    0m0.011s
user    0m0.010s
sys     0m0.000s
```

```
~~~~~>>
```

3 Using the clock() function

We often want to measure time execution time for a certain program section. The clock() function allows us to do so.

* To learn more about the clock() function type in man clock.

```
<<~~~~~
```

```
-bash-4.1$ $ man clock
```

```
~~~~~>>
```

* Below is a typical program template for using the clock() function.

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#include <ctime.h>
```

```
....
```

```
int main(void) {
```

```
clock_t start_time, finish_time;
```

```
...
```

```
// determine overhead
```

```
start_time = clock();
```

```
finish_time = clock();
```

```
double delay_time = (double) (finish_time - start_time);
```



```

processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 85
model name    : Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
stepping     : 4
microcode    : 0x2000065
cpu MHz      : 2991.809
cache size   : 19712 KB
physical id  : 0
siblings     : 24
core id      : 0
cpu cores    : 12
apicid       : 0
initial apicid : 0
fpu          : yes
fpu_exception : yes
cpuid level  : 22
wp           : yes
flags        : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf
eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr
pdc_m pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c
rdrand lahf_lm abm 3dnowprefetch ept cat_l3 cdp_l3 invpcid_single intel_ppin intel_pt
ssbd mba ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust
bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx
smap clflushopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 cqm_llc
cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts pku ospke md_clear
spec_ctrl intel_stibp flush_lld
bogomips     : 5200.00
clflush size : 64
cache_alignment : 64
address sizes : 46 bits physical, 48 bits virtual
power management:

processor      : 1
vendor_id     : GenuineIntel
cpu family    : 6
model         : 85
model name    : Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
stepping     : 4
microcode    : 0x2000065
cpu MHz      : 3558.813
...
~~~~~>>

```

How many processor cores does blackhawk have?

What is the size of main memory? Hint: use `$ cat /proc/meminfo`.

4 Using the `gettimeofday` function

* To learn more about this function type in `man gettimeofday`.

```

<<~~~~~>>

-bash-4.1$ $ man gettimeofday

~~~~~>>

```

The function `gettimeofday` returns two integers.

The first one indicates the number of seconds from January 1, 1970

and the second returns the number of microseconds since the most recent second boundary.

* Below is a sample program that uses `gettimeofday()`.

```

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
#include <stdio.h>
#include <sys/time.h>

struct timeval start, finish ;
int msec;

int main ()
{
    gettimeofday (&start, NULL);

    sleep (200); /* wait ~ 100 seconds */

    gettimeofday (&finish, NULL);

    msec = finish.tv_sec * 1000 + finish.tv_usec / 1000;
    msec -= start.tv_sec * 1000 + start.tv_usec / 1000;

    printf("Time: %d milliseconds\n", msec);
}
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

```

* This utility is useful for measuring execution times of tasks that take a lot of time to complete.

The `clock()` function overflows depending on the length of `clock_t`. For example, if `CLOCKS_PER_SEC` is 1,000,000 and `clock_t` is 32-bit, the overflow will occur every $2^{32}/1,000,000$ seconds, which is approximately every 72 hours. Also, be careful when using `clock()` in parallel programs.

5 Using PAPI

See the LaCASA Getting Started with PAPI page at

<http://lacasa.uah.edu/portal/index.php/tutorials/34-getting-started-with-papi> and PAPI documentation.

6 Using perf

See the LaCASA PerfTool tutorial at <http://lacasa.uah.edu/portal/index.php/tutorials/28-perf-tool> and perf documentation.