Two Fundamental Time Scales
- Processor: ~10^{-9} sec.
- External events: ~10^{-2} sec.
  - Keyboard input
  - Disk seek
  - Screen refresh

Implication
- Can execute many instructions while waiting for external event to occur
- Can alternate among processes without anyone noticing

Measurement Challenge
How Much Time Does Program X Require?
- CPU time
  - How many total seconds are used when executing X?
  - Measure used for most applications
  - Small dependence on other system activities
- Actual ("Wall") Time
  - How many seconds elapse between the start and the completion of X?
  - Depends on system load, I/O times, etc.

Confounding Factors
- How does time get measured?
- Many processes share computing resources
  - Transient effects when switching from one process to another
  - Suddenly, the effects of alternating among processes become noticeable

"Time" on a Computer System

real (wall clock) time

= user time (time executing instructions in the user process)

= system time (time executing instructions in kernel on behalf of user process)

= some other user’s time (time executing instructions in different user’s process)

We will use the word “time” to refer to user time.

cumulative user time
**Activity Periods: Light Load**

- Most of the time spent executing one process
- Periodic interrupts every 10ms
  - Interval timer
  - Keep system from executing one process to exclusion of others
- Other interrupts
  - Due to I/O activity
- Inactivity periods
  - System time spent processing interrupts
  - ~250,000 clock cycles

**Activity Periods: Heavy Load**

- Sharing processor with one other active process
- From perspective of this process, system appears to be “inactive” for ~50% of the time
  - Other process is executing

**Interval Counting**

**OS Measures Runtimes Using Interval Timer**

- Maintain 2 counts per process
  - User time
  - System time
- Each time get timer interrupt, increment counter for executing process
  - User time if running in user mode
  - System time if running in kernel mode

**Interval Counting Example**

**(a) Interval Timings**

```
A 110u + 40s
B 70u + 30s
```

**(b) Actual Times**

```
A 120.0u + 33.3s
B 73.3u + 23.3s
```
**Unix time Command**

```bash
time make osevent
gcc -O2 -Wall -g -march=i486 -c clock.c
gcc -O2 -Wall -g -march=i486 -c options.c
gcc -O2 -Wall -g -march=i486 -c load.c
gcc -O2 -Wall -g -march=i486 -o osevent osevent.c . . .
```

- 0.82 seconds user time
  - 82 timer intervals
- 0.30 seconds system time
  - 30 timer intervals
- 1.32 seconds wall time
- 84.8% of total was used running these processes
  - \((0.82+0.3)/1.32 = 0.848\)

**Accuracy of Interval Counting**

- Computed time = 70ms
- Min Actual = 60 + \(\varepsilon\)
- Max Actual = 80 – \(\varepsilon\)

**Worst Case Analysis**

- Timer Interval = \(\delta\)
- Single process segment measurement can be off by \(\pm\delta\)
- No bound on error for multiple segments
  - Could consistently underestimate, or consistently overestimate

**Average Case Analysis**

- Over/underestimates tend to balance out
- As long as total run time is sufficiently large
  - Min run time ~1 second
  - 100 timer intervals
- Consistently miss 4% overhead due to timer interrupts

**Cycle Counters**

- Most modern systems have built in registers that are incremented every clock cycle
  - Very fine grained
  - Maintained as part of process state
    - In Linux, counts elapsed global time
- Special assembly code instruction to access
- On (recent model) Intel machines:
  - 64 bit counter.
  - RDTSC instruction sets %edx to high order 32-bits, %eax to low order 32-bits
Cycle Counter Period

Wrap Around Times for 550 MHz machine
- Low order 32 bits wrap around every \(2^{32} / (550 \times 10^6) = 7.8\) seconds
- High order 64 bits wrap around every \(2^{64} / (550 \times 10^6) = 33539534679\) seconds
  - 1065 years

For 2 GHz machine
- Low order 32-bits every 2.1 seconds
- High order 64 bits every 293 years

Measuring with Cycle Counter

Idea
- Get current value of cycle counter
  - store as pair of unsigned's cyc_hi and cyc_lo
- Compute something
- Get new value of cycle counter
- Perform double precision subtraction to get elapsed cycles

```
void start_counter()
{
    /* Keep track of most recent reading of cycle counter */
    static unsigned cyc_hi = 0;
    static unsigned cyc_lo = 0;

    access_counter(&cyc_hi, &cyc_lo);
}
```

Accessing the Cycle Cntr.

- GCC allows inline assembly code with mechanism for matching registers with program variables
- Code only works on x86 machine compiling with GCC

```
void access_counter(unsigned *hi, unsigned *lo)
{
    /* Get cycle counter */
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
         :="r" (*hi),="r" (*lo)
         : /* No input */
         : "%edx", "%eax");
}
```

Closer Look at Extended ASM

**Instruction String**
- Series of assembly commands
  - Separated by ";" or "\n"
  - Use "%%" where normally would use "%"

```
asm("Instruction String"
     : Output List
     : Input List
     : Clobbers List);
```

```
void access_counter(unsigned *hi, unsigned *lo)
{
    /* Get cycle counter */
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
         :="r" (*hi),="r" (*lo)
         : /* No input */
         : "%edx", "%eax");
}
```
Closer Look at Extended ASM

### Output List
- Expressions indicating destinations for values \( %0, %1, \ldots, %j \)
  - Enclosed in parentheses
  - Must be *lvalue
    - Value that can appear on LHS of assignment
- Tag "=" indicates that symbolic value (\( %0, \ldots, %j \)), should be replaced by register

### Input List
- Series of expressions indicating sources for values \( %j+1, %j+2, \ldots \)
  - Enclosed in parentheses
  - Any expression returning value
- Tag "r" indicates that symbolic value (\( %0, \ldots, %j \)) will come from register

### Clobbers List
- List of register names that get altered by assembly instruction
- Compiler will make sure doesn’t store something in one of these registers that must be preserved across asm
  - Value set before & used after

---

void access_counter
(unsigned *hi, unsigned *lo)
{
   /* Get cycle counter */
   asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
        : "=r" (*hi), "=r" (*lo)
        : /* No input */
        : "%edx", "%eax");
}

---

Emitted Assembly Code

```
movl 8(%ebp),%esi # hi
movl 12(%ebp),%edi # lo
#APP
rdtsc; movl %edx,%ecx; movl %eax,%ebx
#NO_APP
movl %ecx,(%esi) # Store high bits at *hi
movl %ebx,(%edi) # Store low bits at *lo
```

- Used %ecx for *hi (replacing %0)
- Used %ebx for *lo (replacing %1)
- Does not use %eax or %edx for value that must be carried across inserted assembly code
Completing Measurement

- Get new value of cycle counter
- Perform double precision subtraction to get elapsed cycles
- Express as double to avoid overflow problems

double get_counter()
{
    unsigned ncyc_hi, ncyc_lo
    unsigned hi, lo, borrow;
    /* Get cycle counter */
    access_counter(&ncyc_hi, &ncyc_lo);
    /* Do double precision subtraction */
    lo = ncyc_lo - cyc_lo;
    borrow = lo > ncyc_lo;
    hi = ncyc_hi - cyc_hi - borrow;
    return (double) hi * (1 << 30) * 4 + lo;
}

Timing With Cycle Counter

Determine Clock Rate of Processor

- Count number of cycles required for some fixed number of seconds

double MHZ;
int sleep_time = 10;
start_counter();
sleep(sleep_time);
MHZ = get_counter()/(sleep_time * 1e6);

time function P

- First attempt: Simply count cycles for one execution of P

double tsecs;
start_counter();
P();
tsecs = get_counter() / (MHZ * 1e6);

Measurement Pitfalls

Overhead

- Calling get_counter() incurs small amount of overhead
- Want to measure long enough code sequence to compensate

Unexpected Cache Effects

- artificial hits or misses
- e.g., these measurements were taken with the Alpha cycle counter:
  
  ```
  foo1(array1, array2, array3); /* 68,829 cycles */
  foo2(array1, array2, array3); /* 23,337 cycles */
  vs.
  foo2(array1, array2, array3); /* 70,513 cycles */
  foo1(array1, array2, array3); /* 23,203 cycles */
  ```

Dealing with Overhead & Cache Effects

- Always execute function once to “warm up” cache
- Keep doubling number of times execute P() until reach some threshold
  - Used CMIN = 50000

```
int cnt = 1;
double cmeas = 0;
double cycles;
do {
    int c = cnt;
P(); /* Warm up cache */
    get_counter();
    while (c-- > 0)
        P();
    cmeas = get_counter();
cycles = cmeas / cnt;
cnt *= cnt;
} while (cmeas < CMIN); /* Make sure have enough */
return cycles / (1e6 * MHZ);
```
Multitasking Effects

Cycle Counter Measures Elapsed Time
- Keeps accumulating during periods of inactivity
  - System activity
  - Running other processes

Key Observation
- Cycle counter never underestimates program run time
- Possibly overestimates by large amount

K-Best Measurement Scheme
- Perform up to N (e.g., 20) measurements of function
- See if fastest K (e.g., 3) within some relative factor $\varepsilon$ (e.g., 0.001)

K-Best Validation

- Very good accuracy for < 8ms
  - Within one timer interval
  - Even when heavily loaded

- Less accurate of > 10ms
  - Light load: ~4% error
  - Interval clock interrupt handling
  - Heavy load: Very high error

Compensate For Timer Overhead

K = 3, $\varepsilon = 0.001$

Subtract Timer Overhead
- Estimate overhead of single interrupt by measuring periods of inactivity
- Call interval timer to determine number of interrupts that have occurred

Better Accuracy for > 10ms
- Light load: 0.2% error
- Heavy load: Still very high error

Acceptable accuracy for < 50ms
- Scheduler allows process to run multiple intervals

Less accurate of > 10ms
- Light load: 2% error
- Heavy load: Generally very high error


**Time of Day Clock**

- **Unix gettimeofday() function**
- **Return elapsed time since reference time (Jan 1, 1970)**
- **Implementation**
  - Uses interval counting on some machines
    - Coarse grained
  - Uses cycle counter on others
    - Fine grained, but significant overhead and only 1 microsecond resolution

```c
#include <sys/time.h>
#include <unistd.h>

struct timeval tstart, tfinish;
double tsecs;

gmtimeofday(&tstart, NULL);
P();
gmtimeofday(&tfinish, NULL);
tsecs = (tfinish.tv_sec - tstart.tv_sec) +
     1e6 * (tfinish.tv_usec - tstart.tv_usec);
```

**Measurement Summary**

Timing is highly case and system dependent

- What is overall duration being measured?
  - > 1 second: interval counting is OK
  - << 1 second: must use cycle counters
- On what hardware / OS / OS version?
  - Accessing counters
    - How gettimeofday is implemented
  - Timer interrupt overhead
  - Scheduling policy

**Devising a Measurement Method**

- Long durations: use Unix timing functions
- Short durations
  - If possible, use gettimeofday
  - Otherwise must work with cycle counters
  - K-best scheme most successful