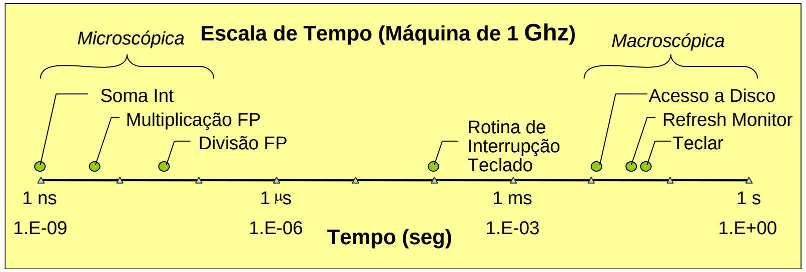
Estrutura do tema Avaliação de Desempenho (IA32)

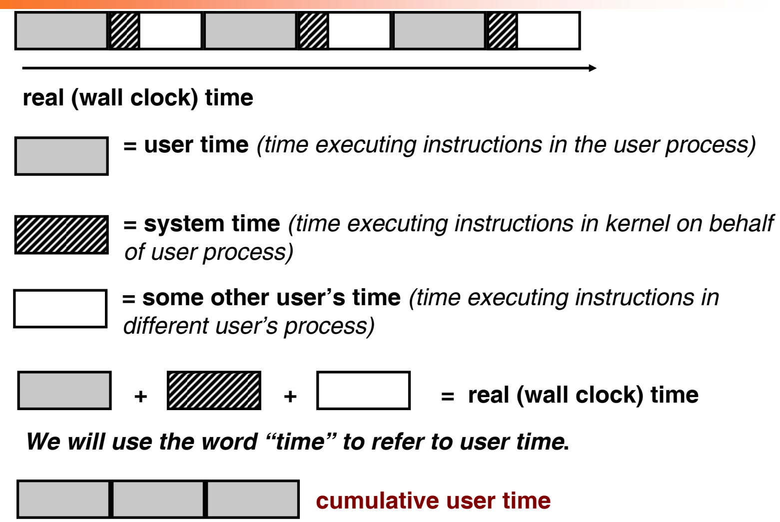
1. A avaliação de sistemas de computação
2. Técnicas de optimização de código (IM)
3. Técnicas de optimização de *hardware*
4. Técnicas de optimização de código (DM)
5. Outras técnicas de optimização
6. Medição de tempos

Os próximos slides foram adaptados da aula do Prof. Bryant em 2002

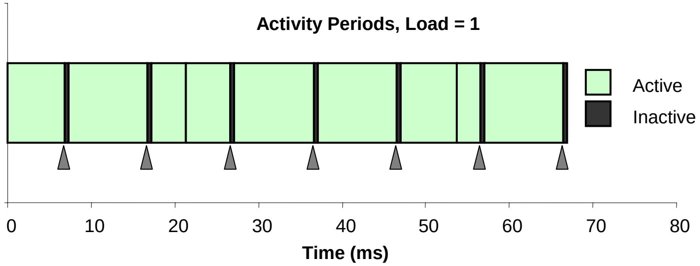


- Escalas fundamentais de tempo:
  - Processador: ~10^{-9} seg.
  - Eventos externos: ~10^{-2} seg.
    - *Keyboard input*
    - *Disk seek*
    - *Screen refresh*
- Implicações
  - pode executar várias instr enquanta espera que ocorram eventos externos
  - pode alternar execução entre código de vários proc sem ser notado

- 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

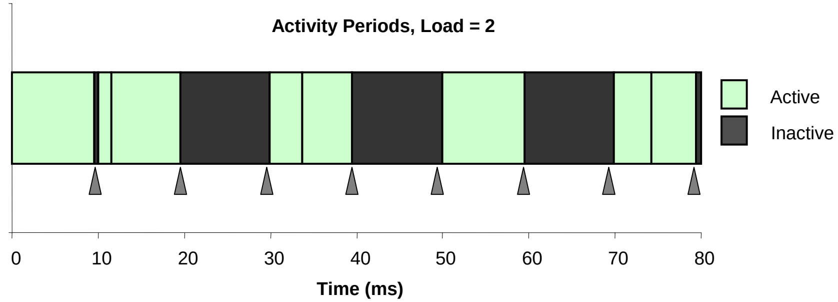


Activity Periods: Light Load



- 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

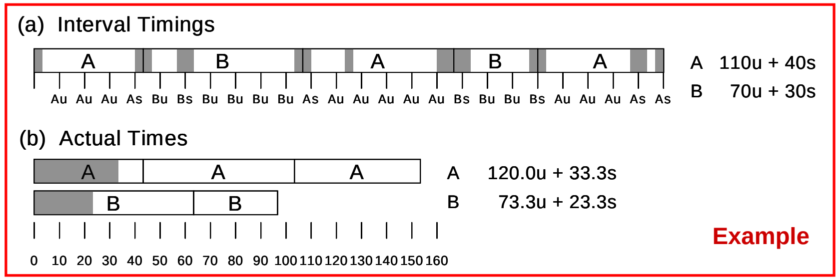
Activity Periods: Heavy Load



- 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: (i) get timer interrupt, (ii) increment counter for executing process
    - User time if running in user mode
    - System time if running in kernel mode

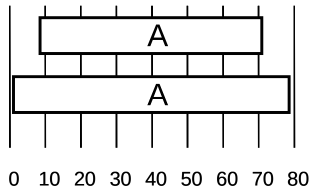


Unix `time` Command

```
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 . . .
0.820u 0.300s 0:01.32 84.8%     0+0k 0+0io
4049pf+0w
```

- 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
  - (.82+0.3)/1.32 = .848

Accuracy of Interval Counting (1)

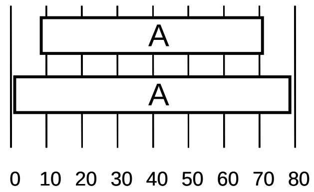


Minimum
Maximum

- **Computed time = 70ms**
- **Min Actual = 60 + ε**
- **Max Actual = 80 – ε**

- Worst Case Analysis
  - Timer Interval = δ
  - Single process segment measurement can be off by ±δ
  - No bound on error for multiple segments
    - Could consistently underestimate, or consistently overestimate

Accuracy of Int. Counting (2)



Minimum
Maximum

- **Computed time = 70ms**
- **Min Actual = 60 + ε**
- **Max Actual = 80 – ε**

- 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
    $2^{32} / (550 * 10^6) = 7.8$ seconds
  - High order 64 bits wrap around every
    $2^{64} / (550 * 10^6) = 33539534679$ seconds
    - 1065 years
- For **2 GHz** machine
  - Low order 32-bits every 2.1 seconds
  - High order 64 bits every 293 years

- 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

```
/* Keep track of most recent reading of cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

void start_counter()
{
  /* Get current value of cycle counter */
  access_counter(&cyc_hi, &cyc_lo);
}
```


- – 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");
}
```

–Emit assembly with `rdtsc` and two `movl` instructions


```
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");
}
```

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");
}
```

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

```
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");
}
```

Input List

- Series of expressions indicating sources for values `%j+1`, `%j+2`, …
  - Enclosed in parentheses
  - Any expression returning value
- Tag `"r"` indicates that symbolic value (`%0`, etc.) will come from register


```
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");
}
```

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

- 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
```

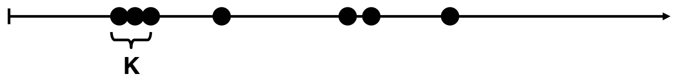
- Used `%ecx` for `*hi` (replacing `%0`)
- Used `%ebx` for `*lo` (replacing `%1`)
- Does not use `%eax` or `%edx` for value that must be carried across inserted assembly code

- 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;
}
```

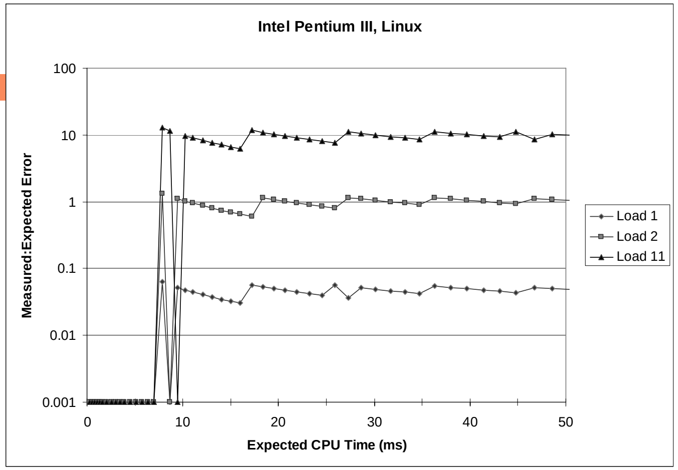
- Measurement Pitfalls: Overhead
  - calling `get_counter()` incurs small amount of overhead
  - want to measure long enough code sequence to compensate
- Measurement Pitfalls: Unexpected Cache Effects
  - artificial hits or misses
- Dealing with Cache Effects:
  - always execute function <u>once</u> to "warm up" cache

- 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 ε (e.g., 0.001)



K-Best Validation

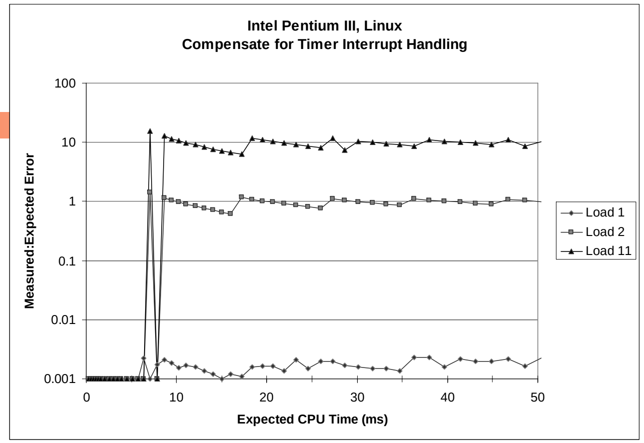
K = 3, ε = 0.001



- Very good accuracy for < 8ms
  - Within one timer interval
  - Even when heavily loaded
- Less accurate of > 10ms
  - Light load: ~4% error
    - Interval clock int handling
  - Heavy load: Very high error

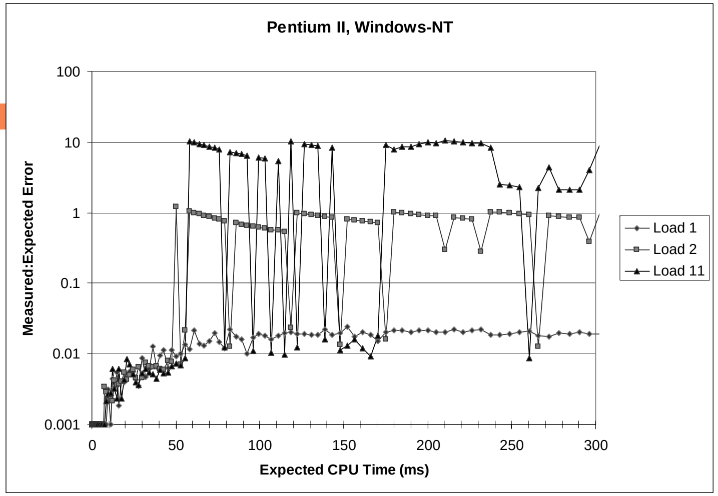
Compensate For Timer Overhead

K = 3, ε = 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

K-Best on NT



K = 3, ε = 0.001

- 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

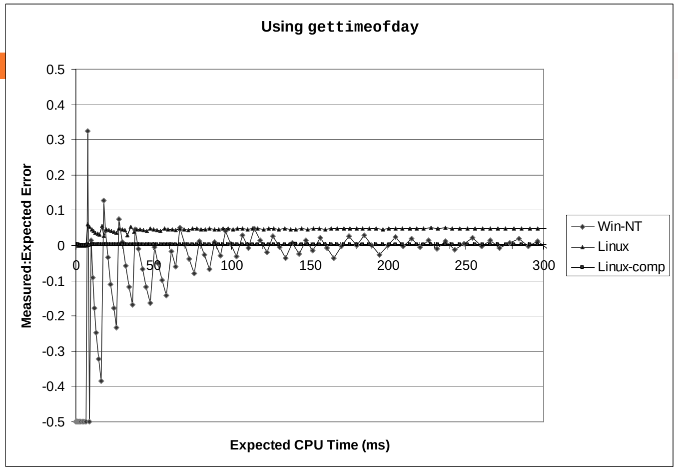
Time of Day Clock

- 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 μsec resolution

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

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


K-Best Using `gettimeofday`



- Linux
  - As good as using cycle counter
  - For times > 10 microseconds
- Windows
  - Implemented by interval counting
  - Too coarse-grained

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