Monitors

- Monitor: A tool used to observe the activities on a system.
- Usage:
  - A system programmer may use a monitor to improve software performance. Find frequently used segments of the software.
  - A systems manager may use a monitor to measure resource utilizations and to find the performance bottleneck.
  - A systems manager may also use a monitor to tune the system.
  - A systems analyst may use a monitor to characterize the workload.
  - A systems analyst may use a monitor to find model parameters, to validate models, and to develop inputs for models.

Monitor Terminology

- Event: A change in the system state is called an event.
  Examples: process context switching, beginning of seek on a disk, and arrival of a packet.
- Trace: A log of events
- Overhead: Artifact
- Domain: The set of activities observable by the monitor.
- Input Rate: The maximum frequency of events that a monitor can correctly observe.
  - Burst-mode rate: the rate at which an event can occur for a short duration.
  - Sustained rate: Can tolerate for long durations.
- Resolution: The coarseness of the information observed is called the

resolution.
For example, record time only in units of 16 milliseconds.

- Input Width: The number of bits of information recorded on a event is called the input width.

Monitor Classification

- Implementation:
  - Software monitor
  - Hardware monitor
  - Firmware monitor
  - Hybrid monitor
- Trigger Mechanism:
  - Event-driven: Good for rare events.
  - Timer-driven (sampling monitor): Good for frequent events.
- By result display ability:
  - On-line monitors: display the system state continuously
  - Batch monitors: collect the data that can be analyzed later.
- Example: A particular monitor may be classified as a hybrid-sampling-batch

monitor.

Software Monitors

- Used for operating systems and higher level software such as networks and databases.
- Suitable only if the input rate is low.
- Also used if overhead is not an issue. For example, in an instruction-tracing monitor, every single-user instruction executed may be interrupted.
- Compared to H/W monitor:
  - Lower input rates
  - Lower resolutions
  - Higher overhead
  - Higher input widths
  - Higher recording capacities
  - Easier to develop
  - Easier to modify

Issues in Software-Monitor Design

1. Activation Mechanism:
   (a) Trap instruction
   (b) Trace mode
   (c) Timer interrupt
2. Buffer Size:
   - Large $\Rightarrow$ Too much time per write
   - Small $\Rightarrow$ Too many write operations
   - Optimum = function of input rate, input width, and the emptying rate.
3. Number of Buffers:
   - Minimum two.
   - More to allow varying
4. Buffer Overflow:
   - Overwrite: old info is lost
   - Block: new info is lost

In any case, record buffer overflows. Similarly, counters can be stuck-at or zeroed on overflow.

5. Data Compression or Analysis:
   - Process the data as it is observed
     - Reduces the space required.
     - Increases the overhead.
6. On/Off Switch: Use conditional (IF ... THEN ...)
   - Reduces overhead
   - Helps during development and debugging.
7. Language: Use the same language as the system
8. Priority:
   - Should not affect system operations $\Rightarrow$ Low priority for the monitor

- Timely recording $\Rightarrow$ High priority

9. Abnormal-Events Monitoring: Should observe normal as well as abnormal events. For example:
    - System initialization
    - Device failures
    - Program failures.
10. Users prefer to monitor abnormal events:
    - Abnormal events occur at a lower rate and impose less monitoring overhead
    - Abnormal events help the user take preventive action

Hardware Monitors

- Separate equipment attached to the system being monitored via probes.
- No system resources consumed in monitoring
- Higher input rate
- Less chances of introducing bugs into the system operation

Components:

1. Probes
2. Counters
3. Logic Elements: AND, OR, and other logic gates.
4. Comparators: To compare counters or signal values with preset values.
5. Mapping Hardware: Allows histograms

6. Timer: For time-stamping or sampling
7. Tape/Disk: To store the data

Probe-point libraries: A list of points on the system where the probes can be attached.

Software vs. Hardware Monitors

Criterion	Hardware Monitor	Software Monitor
Domain	Difficult to monitor operating system events.	Difficult to monitor hardware events unless recognizable by an instruction.
Input Rate	Sampling rates of 10^5 per second possible.	Sampling rate limited by the processor MIPS and overhead required.
Time Resolution	10 ns is possible.	Generally 10 to 16 ms.
Expertise	Requires intimate knowledge of hardware.	Requires intimate knowledge of software.
Recording Capacity	Limited by memory and secondary storage. Not a problem currently.	Limited by overhead desired.
Input Width	Can record several simultaneous events.	Can't record several simultaneous events unless there are multiple processors.
Monitor Overhead	None	Overhead depends upon the input rate and input width. Less than 5% adequate and more than 100% possible.
Portability	Generally portable.	Specific to an operating system.
Availability	Monitoring continues even during system malfunction or failure.	Can't monitor during system crash.
Errors	Possible to connect the probes to wrong points.	Once debugged, errors are rare.
Cost	High	Medium

Firmware Monitors

- Implemented by modifying the processor microcode.
- For applications that fall in between the software and hardware monitoring boundaries.
- Similar to software monitors.
- Tighter timing limitations
- Very limited data reduction, if any.

Examples:

- Network interfaces microprogrammed to monitor all traffic on the network
- Address profiles of microcode (micro-PC histograms)

Hybrid Monitors

- Combination of software, hardware, or firmware
- Hardware data-gathering + software data-reduction

Example: Dimond monitor by Hughes (1980)

- Hardware part can observe all traffic on the system bus.
- Software part can read instruction addresses, processor modes, and system and user identifications.
- Two parts can communicate through device status and control registers.

Program-Execution Monitors

- Software monitors designed to observe application software.
- Purpose:
  - Tracing: To find the execution path of a program.
  - Timing: To find the time spent in various modules of the program.
  - Tuning: To find the most frequent or most time-consuming sections of the code.
  - Assertion Checking: To verify the relationships assumed among the variables of a program.
  - Coverage Analysis: To determine the adequacy of a test run.
- Criteria for Program Selection:
  - Time critical: to find out where the time

is being spent.

- Freqeuncy of use: high frequency programs should be optimized first.
- Percentage of resources: to optimize
- Most expensive resource should be optimized first.

Steps in Program-Execution Monitoring

Figure ??

Issues in Designing a Program-Execution Monitor

- All issues in s/w monitor design apply.
- Program Execution Specific:

1. Measurement Unit: modules, subroutines, high-level language statements, or machine instruction.
2. Measurement Technique: tracing (traps) or sampling
3. Instrumentation Mechanism: Instrumentation added before/during/after compilation or during run time.

   By augmenting the source code, the compiler-generated object code, the run-time environment, the operating system, or the hardware.
4. Profile Report:

– Show a frequency and time histogram.
– Summaries by modules, procedurs, statements.
– Show self-time and inherited time
– Allow limiting or expanding the detail (zoom-in or zoom-out)

Techniques for Improving Program Performance

- Code optimization
- I/O optimization,
  - Blocking I/Os
  - Changing the file-access method
  - Caching or prefetching data.
- Paging optimization.

Box 7.1 Techniques for Improving Program Performance

1. Optimize the common case. The most frequently used path should also be the most efficient. A procedure should handle all cases correctly and common cases efficiently.
2. Arrange a series of IF statements so that the most likely value is tested first.
3. Arrange a series of AND conditions so that the condition most likely to fail is tested first.
4. Arrange entries in a table so that the most frequently sought values are the first to be compared.
5. Structure the main line of the program so that it contains the most frequent path of the program. Errors and exceptions should be handled separately.
6. Question the necessity of each instruction in the main path (time-critical or most frequent) path of the code.
7. Trade memory space for processor time wherever possible. If a function is computed more than once, compute it once and store the result. Some functions with parameters can be replaced by a table of precomputed values.
8. Use hints. Keeping a fast but possibly inaccurate hint along with a slow but robust algorithm may save time in most cases.
9. Cache the data that is accessed often. However, one must ensure that there is sufficient locality before using caches.
10. Unroll short loops. Save the cost of modifying and testing loop indexes.
11. Replace searches by direct indexing, wherever possible. In some cases, this may require more space than minimum.
12. Use the same size for data fields that need to be compared or added together.
13. Use the full word widths of the computer to evaluate expressions. For example, use 32 bits on a 32-bit computer even if you need only 31.
14. Align data fields on word boundaries, wherever possible.
15. Evaluate items only when needed, particularly if it is likely that it will not be needed.

Continued on the next page...

Box 7.1 Techniques for Improving Program Performance (Continued)

16. Initialize data areas during run time only when used. Wherever possible, the values should be initialized at the compile time.
17. Use algebraic identities to simplify conditional expressions.
18. Replace threshold tests on monotone (continuously nondecreasing or continuously nonincreasing) functions by tests on their parameters, thereby, avoiding the evaluation of the function.
19. Evaluate variables not changing in a loop before entering the loop.
20. Combine two nearby loops if they use the same set of conditions.
21. Use shifts in place of multiplication and division by powers of two.
22. Keep the code simple. Simpler programs are more efficient.
23. Block I/O requests together to reduce the number of I/O operations.
24. Preload small disk files into tables in memory.
25. Use multiple buffers for files to allow prefetching.
26. Link the most frequently used procedures together to maximize the locality of reference.
27. Reference data in the order stored. Arrays stored by columns should be referenced by columns.
28. Store data elements, that are used concurrently together.
29. Store subroutines in sequence so that calling and called subroutines will be loaded together.
30. Align I/O buffers to page boundaries.
31. Open files, which are used together, in sequence so that buffers will be located together.
32. Pass simple subroutine arguments by value rather than by reference, wherever possible.
33. Pass large arrays and other data structures to subroutines by reference rather than value.
34. Separate read-only code areas from read-write data areas to minimize the number of page-writes.

Exercises

7.1 For each of the following measurements list the type of the monitor that can and cannot be used. Which type of monitor would you prefer and why.

a. Interrupt response time

b. Instruction opcode frequency

c. Program reference pattern

d. Virtual memory reference pattern in a multiprogramming system

e. CPU time required to send one packet on a network

f. Response time for a database query

7.2 For each of the following environments, describe how you would implement a monitor to produce a program counter histogram:

a. Using a H/W monitor

b. Using a S/W monitor on an IBM PC with CPU having a trace-bit.
c. Using a S/W monitor on a TRS-80$^{\text{TM}}$ with CPU not having a trace-bit.