Software Development based on Intel® Multi-core Architecture
Software Development Tools for Multi-core Computer System

Content
- Intel® performance tuning and monitoring tools
- IPP and MKL
- Intel® Threading Building Blocks
- Runtime Analysis
- Open source tools such as GCC, GDB
Software Development Tools for Multi-core Computer System

- Part1 Intel® performance tuning and monitoring tools
VTune™ Performance Analyzer

What Vtune can do

- Collecting performance data from the system running your application.
- Organizing and displaying the data in a variety of interactive views, from system-wide down to source code or processor instruction perspective.
- Identifying potential performance issues and suggesting improvements.
Supported Environments

- Intel® IA-32 Processors
  - Microsoft Windows* operating systems
  - Red Hat Linux*
  - SuSE Linux

- Itanium® Family Processors
  - Microsoft Windows operating systems
  - Red Hat Linux
  - SuSE Linux
Feature of the Vtune

- Sampling
- Call graph
Hotspots

- Where in an application or system there is a significant amount of activity
  - Where = address in memory => OS process => OS thread => executable file or module => user function (requires symbols) => line of source code (requires symbols with line numbers) or processor (assembly) instruction
  - Significant = activity that occurs infrequently probably does not have much impact on system performance
  - Activity = time spent or other internal processor event
    - Examples of other events: Cache misses, branch mispredictions, floating-point instructions retired, partial register stalls, and so on
Sampling

- Sampling is the Statistical Method of Finding Hotspots

- A sampling collector periodically interrupts the processor to collect the execution context
Sampling collector

- Periodically interrupts the processor
  - Time-based sampling (TBS) is triggered by
    - Operating system timer services
    - Every n processor clockticks
  - Event-based sampling (EBS) is triggered by
    - These events are processor-specific, like L2 cache misses, branch mispredictions, floating-point instructions retired, and so on
Sampling collector

- Collects the execution context
  - Execution address in memory (CS:IP)
  - Operating system process and thread ID
  - Executable module loaded at that address
    - If you have symbols for the module, post-processing can identify the function or method at the memory address.
    - Line numbers from the symbol file can direct you to the relevant line of source code.
How Event-based Sampling (EBS) Works

Conceptual Diagram

Select Event Signal → Count Down → “Sample After” Number

Underflow to Zero → Interrupt CPU to Take Sample

Internal Interrupt Controller

How do you choose a “Sample After” number?
How Many Samples Are Enough

- About 1,000 samples per second
- It is chosen by experience
- It is a good balance between significance and overhead
Three Key Benefits of Sampling

- **You do not have to modify your code.**
  - But DO compile/link with symbols and line numbers.
  - But DO make release builds with optimizations.

- **Sampling is system-wide.**
  - Not just YOUR application.
  - You can see activity in operating system code, including drivers.

- **Sampling overhead is very low.**
  - Validity is highest when perturbation is low.
  - Overhead can be reduced further by turning off progress meters in the user interface.
Call Graph Profiling

- Win32 applications
- Stand-alone Win32* DLLs
- Stand-alone COM+ DLLs
- Java applications
- .NET* applications
- ASP.NET applications
- Linux32* applications
Call Graph Profiling

- Tracks the function entry and exit points of your code at run time
- Uses binary instrumentation
- Uses this data to determine program flow, critical functions and call sequences
- Not system-wide: Only profiles code in applications call path in Ring 3
# Call Graph Metrics

<table>
<thead>
<tr>
<th>Performance Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self Time</td>
<td>Total time in a function, excluding time spent in its children (includes wait time)</td>
</tr>
<tr>
<td>Total Time</td>
<td>Time measured from a function entry to exit point</td>
</tr>
<tr>
<td>Total Wait Time</td>
<td>Time spent in a function and its children when the thread is blocked</td>
</tr>
<tr>
<td>Wait Time</td>
<td>Time spent in a function when the thread is blocked (excludes blocked time in its children)</td>
</tr>
<tr>
<td>Calls</td>
<td>Number of times the function is called</td>
</tr>
</tbody>
</table>
## Compare between the sampling and call graph

<table>
<thead>
<tr>
<th>Sampling</th>
<th>Call graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low overhead</td>
<td>Higher overhead</td>
</tr>
<tr>
<td>System-wide</td>
<td>Ring 3 only on your application call tree</td>
</tr>
<tr>
<td>System-wide address histogram</td>
<td>Show function level hierarchy with call counts, times, and the critical path</td>
</tr>
<tr>
<td>For function level drill-down, must have debug information</td>
<td>Must re-link with /fixed:no, automatically instruments</td>
</tr>
<tr>
<td>Can sample based on time and other processor events</td>
<td>Results are based on time</td>
</tr>
</tbody>
</table>
Software Development Tools for Multi-core Computer System

- Part2 IPP and MKL
MKL (Math Kernel Library)

- **Purpose**
  - Improve the performance
  - Intel®’s engineering, scientific, and financial math library

- **Not used on**
  - Geometric Transformation
  - “small” counts
  - call vector math functions on small n
MKL (Math Kernel Library)

Addresses:
- Solvers (BLAS, LAPACK)
- Eigenvector/eigenvalue solvers (BLAS, LAPACK)
- Some quantum chemistry needs (dgemm)
- PDEs, signal processing, seismic, solid-state physics (FFTs)
- General scientific, financial [vector transcendental functions (VML) and vector random number generators (VSL)]
BLAS (Basic Linear Algebra Subroutines)
- Level 1 BLAS – vector-vector operations
  - 15 function types
  - 48 functions
- Level 2 BLAS – matrix-vector operations
  - 26 function types
  - 66 functions
- Level 3 BLAS – matrix-matrix operations
  - 9 function types
  - 30 functions
- Extended BLAS – level 1 BLAS for sparse vectors
  - 8 function types
  - 24 functions
Contents

- LAPACK (linear algebra package)
  - Solvers and eigensolvers. Many hundreds of routines total!
  - There are more than 1000 total user callable and support routines
- DFTs (Discrete Fourier transforms)
  - Mixed radix, multi-dimensional transforms
  - Multithreaded
- VML (Vector Math Library)
  - Set of vectorized transcendental functions
  - Most of libm functions, but faster
- VSL (Vector Statistical Library)
  - Set of vectorized random number generators
LAPACK (linear algebra package)

- Most important LAPACK optimizations:
  - Threading – effectively uses multiple CPUs
  - Recursive factorization
    - Reduces scalar time (Amdahl’s law: $t = t_{scalar} + \frac{t_{parallel}}{p}$)
    - Extends blocking further into the code
Vector Statistical Library (VSL)

- Numerous non-uniform distributions
- VML used extensively for transformations
- Parallel computation support – some functions
- User can supply own BRNG or transformations
- Five basic RNGs (BRNGs) – bits, integer, FP
  - MCG31, R250, MRG32, MCG59, WH
Using VSL

- Basically a 3-step Process
  - Create a stream pointer.
    ```c
    VSLStreamStatePtr stream;
    ```
  - Create a stream.
    ```c
    vslNewStream(&stream,VSL_BRNG_MC_G31, seed );
    ```
  - Generate a set of RNGs.
    ```c
    vsRngUniform( 0, &stream, size, out, start, end );
    ```
  - Delete a stream (optional).
    ```c
    vslDeleteStream(&stream);
    ```
IPP (Integrated Performance Primitives)

- IPP is a software library which provides a range of library functions for multimedia, audio codecs, video codecs, image compression, image processing, signal processing, speech compression plus computer vision as well as math support routines for such processing capabilities.

- Intel® IPP is divided into three large data domains that contain primitives operating on:
  - signals (with linear array or vector data)
  - images (with 2D arrays for typical color spaces)
  - matrices (with \( N \times M \) arrays for matrix operations)
fits between the application layer and the hardware layer
IPP (Integrated Performance Primitives)

- Support for Multi-Core Processors
  - Multi-core enable the true parallel execution of multi-threaded software applications

- Multi-Platform Compatibility
  - Supported Windows*, Linux* and Mac OS*
  - easily used and integrated with popular development tools and environments, such as Microsoft Visual Studio*, Xcode*, Eclipse*, GCC, and the Intel® C++ Compiler

- Processors
  - Intel® Core™ Duo and Intel® Pentium® D multi-core processors
  - Intel® Core™ Solo processors
  - Intel® Xeon® processors
  - Processors with Intel® EM64T, including 64-Bit Intel® Xeon processors, Intel® Pentium® D processors, Intel® Pentium® processor Extreme Edition
  - Intel® Pentium® 4 and Intel® Pentium® M processors
  - Processors based on Intel® XScale® technology, including Intel® IXP4xx processors and Intel® PXA27x application processors with Intel® Wireless MMX™ Technology support.
  - Intel® Itanium® 2 processors

- Performance-Otimized Functions
Part3 Intel ® Threading Building Blocks
Intel® Threading Building Blocks 1.0 is a C++ runtime library that simplifies threading for performance. It provides parallel algorithms and concurrent data structures that eliminate tedious threading implementation work. It’s a tested and performance-tuned parallel substrate for your application.
TBB Features

Ready to use parallel algorithms:
Select from a library of highly-efficient parallel algorithm templates, and rapidly obtain the advantages of multi-core Intel processors.

- Quickly employ commonly needed algorithms designed for parallel performance and scalability.
- Generic templates let you easily tailor these algorithms to your needs.
- Supports easy plug-in deployment into applications to deliver scalable software speed-up, optimizing for both available cores and cache locality.
- Reduce the work required to produce threaded software in many cases, by means of pre-built parallel constructs.
TBB Features

Cross platform support:

Write applications once and deploy on multiple OS's.

- Provides a single solution for Windows*, Linux*, and Mac OS* on 32-bit and 64-bit platforms using Intel®, Microsoft, and GNU compilers.
- Supports industry-leading compilers from Intel, Microsoft and GNU.
- Speeds deployment of applications on multiple multi-core platforms.
TBB Features

Task based parallelism:

- Specify threading functionality in terms of logical tasks instead of physical threads.
- Lets developer focus on higher level of scalable task patterns instead of low-level thread mechanics.
- Uses proven data-decomposition abstractions that efficiently use multiple cores.
- Enables automatic load balancing.
- Efficiently supports nested parallelism, allowing parallel components to be built from other parallel components.
**TBB Features**

**Library based solution:**

- Get highly optimized parallel functionality now with minimal effort.
- Your C++ application simply calls the Threading Building Blocks library
- Standard C++ - no need to rewrite code in a new language
- Compatible with other threading packages
- Allows unlimited distribution of the runtime libraries with your software
- Seamlessly integrates into existing development environments
Highly concurrent containers:

- Optimize the processor's ability to perform simultaneous tasks.
- Simplify multithreaded application development with interfaces designed for thread-safety and high concurrency.
- Improve application quality by employing pre-tested data structures.
- Improve application performance by enabling multiple execution cores or processors to work together more efficiently.
TBB Library Components

- Generic Parallel Algorithms
- Thread-Safe Containers
- Synchronization Primitives
- Task Scheduler
- Memory Allocation
- Timing
Software Development Tools for Multi-core Computer System

- Part4 Runtime Analysis
Runtime Analysis

- What is runtime analysis
  - Analysis the applications in the runtime

- Tools of the runtime analysis
  - VTune™ Performance Analyzer
  - Intel® Thread Checker
  - Intel® Thread Profiler
Intel® Thread Checker

- **Why it useful**
  - New class of problems are caused by the interaction between concurrent threads

- **What it can do**
  - Find data races or storage conflicts
    - More than one thread accesses memory without synchronization
  - Find deadlocks
    - Thread waits for an event that will never happen
Intel® Thread Checker

- Debugging tool for threaded software
- Finds threading bugs in Win32*, POSIX*, and OpenMP* threaded software
- Locates bugs quickly that can take days to find using traditional methods and tools
  - Isolates problems, not the symptoms
  - Bug does not have to occur to find it!
- Plug-in to VTune™ Performance Analyzer
  - Same look, feel, and interface as VTune™ environment
Feature

- Supports several different compilers
  - Intel®C++ and Fortran Compilers, v7 and higher
  - Microsoft* Visual* C++, v6
    - Integrated into Microsoft Visual Studio .NET* IDE
- View (drill-down to) source code for Diagnostics
- One-click help for diagnostics
  - Possible causes and solution suggestions
- API for user-defined synchronization primitives
Race Conditions

- More than one thread access the same variable at the same time
- Most common in multithreaded program
- May not be apparent at all times
Solutions

- **Solution 1**: Scope variables to be local to threads
  - **When to use**
    - Value computed is not used outside parallel region
    - Temporary or “work” variables
  - **How to implement**
    - OpenMP scoping clauses (*private, shared*)
    - Declare variables within threaded functions
    - Allocate variables on thread stack
    - TLS (Thread Local Storage) API
Solution

Solution 2: Control shared access with critical regions

When to use
- Value computed is used outside parallel region
- Shared value is required by each thread

How to implement
- Mutual exclusion and synchronization
- Lock, semaphore, event, critical section, atomic...
- Rule of thumb: Use one lock per data element
Deadlock

- Caused by thread waiting on some event that will never happen

DWORD WINAPI threadA(LPVOID arg)
{
    EnterCriticalSection(&L1);
    EnterCriticalSection(&L2);
    processA(data1, data2);
    LeaveCriticalSection(&L2);
    LeaveCriticalSection(&L1);
    return(0);
}

DWORD WINAPI threadB(LPVOID arg)
{
    EnterCriticalSection(&L2);
    EnterCriticalSection(&L1);
    processB(data1, data2);
    LeaveCriticalSection(&L1);
    LeaveCriticalSection(&L2);
    return(0);
}

ThreadA: L1, then L2
ThreadB: L2, then L1
Deadlock

Add lock per element, not whole Q array

typedef struct {
    // some data things
    SomeLockType mutex;
} shape_t;

shape_t Q[1024];

void swap (shape_t A, shape_t B) {
    lock(a.mutex);
    lock(b.mutex);
    // Swap data between A & B
    unlock(b.mutex);
    unlock(a.mutex);
}

Thread 1
swap(Q[34], Q[986]);

Grabs mutex 34

Grabs mutex 986

Thread 4
swap(Q[986], Q[34]);
Two Ways to Ensure Thread Safety

- Routines can be written to be reentrant
  - Any variables changed by the routine must be local to each invocation
    - Don’t modify globally shared variables
- Routines can use mutual exclusion to avoid conflicts with other threads
  - If accessing shared variables cannot be avoided

It is better to make a routine reentrant than to add synchronization
Avoids potential overhead

- What if third-party libraries are not thread safe?
  - Will likely need to control threads access to library
Thread Safety Example

- Check for safety issues between
  - Multiple instances of `routine1()`
  - Instances of `routine1()` and `routine2()`
- Set up sections to test all permutations
- Still need to provide data sets that exercise relevant portions of code

```c
#pragma omp parallel sections
{
  #pragma omp section
  routine1(&data1);
  #pragma omp section
  routine1(&data2);
  #pragma omp section
  routine2(&data3);
}
```
Intel® Thread Profiler

- Plugs in to the VTune™ performance environment
- Motivation
  - Developing efficient multithreaded applications is hard
  - New performance problems are caused by the interaction between concurrent threads
    - Load imbalance
    - Contention on synchronization objects
    - Threading overhead
Features

- Supports several different compilers
  - Intel® C++ and Fortran Compilers, v7 and higher
  - Microsoft® Visual® C++, v6
    - Integrated into Microsoft Visual Studio .NET® IDE
- Binary instrumentation of applications
- Different views and filters available to assist and organize analysis
- Uses critical path analysis
Critical Path

What is the Flow

- A new flow is created when a thread is created or resumes
- Flow ends when a thread terminates or blocks on a synchronization primitive
Critical Path

- Critical Path is the longest execution flow
Critical Path Analysis

- Purpose: If the **critical path** is shortened, the application will run in less time

- System Utilization
  - Relative to the system executing the application
    - **Idle**: no threads
    - **Serial**: a single thread
    - **Under-subscribed**: more than one thread, less than cores
    - **Parallel**: # threads == # cores
    - **Oversubscribed**: # threads > # cores

- Thread interaction categories
  - **Cruise**: threads running without interference
  - **Overhead**: thread operation overhead
  - **Blocking**: thread waiting on external event
  - **Impact**: thread preventing some other thread from executing
System Utilization

- Examines processor utilization to determine concurrency level of the application
- Concurrency is the number of *active* threads

![Concurrency Graph]

- **Thread 1**
  - Acquire Lock L
  - Wait for Threads 2 & 3
  - Release L
  - Acquire L
  - Release L
  - Acquire L
  - Threads 2 & 3 Done

- **Thread 2**
  - Acquire Lock L
  - Release L
  - Wait for L
  - Acquire L
  - Threads 2 & 3 Done

- **Thread 3**
  - Acquire Lock L
  - Wait for Threads 2 & 3
  - Release L
  - Acquire L
  - Threads 2 & 3 Done

Graph Legend:
- **Idle**
- **Serial**
- **Under-subscribed**
- **Parallel**
- **Over-subscribed**
Execution Time Categories

- Analyze thread interaction and behavior along critical path
- Record objects that cause CP transitions

Cruise time  
Overhead  
Blocking time  
Impact time

Thread Interaction

- Threads 2 & 3 Done
- Threads 2 & 3 Wait
- Acquire L
- Release L
- Wait for L
- Acquire lock L
- Release L
- Wait for Threads 2 & 3
- Threads 1 Complete
- Threads 2 & 3 Complete

Thread 3
Thread 2
Thread 1

Time

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T10</td>
<td>T11</td>
<td>T12</td>
<td>T13</td>
<td>T14</td>
<td>T15</td>
</tr>
</tbody>
</table>
Merging Concurrency and Behavior

Start with System utilization
Further categorize by behavior

<table>
<thead>
<tr>
<th>Concurrency Level</th>
<th>Critical Path</th>
<th>Thread Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Legend:
- No thread, blocking time
- Serial cruise time
- Serial blocking time
- Serial impact time
- Undersubscribed cruise time
- Undersubscribed blocking time
- Undersubscribed impact time
- Fully parallel cruise time
- Fully parallel blocking time
- Fully parallel impact time
- Oversubscribed cruise time
- Oversubscribed blocking time
- Oversubscribed impact time
- Overhead time
Thread Profiler Views

- Critical Path View
  - Shows breakdown of the critical path

- Profile View
  - Shows the breakdown of selected critical paths
  - User can select other views of the selected profile
  - Concurrency level, threads, objects

- Timeline View
  - Shows thread activity and critical path transitions for the entire application

- Source View
  - Transition source view, creation source view
Critical Path View

- Critical path 1: Length 7.16E-06, Associated with thread 1, Contended transition rate = 9110.65 per second.

Double Click
Profile View – Concurrency Level

Let’s look at the threads view. Threads ran simultaneously ~35% of the time.
Timeline View
Source view
Common Performance Issues

- Load balance
  - Improper distribution of parallel work

- Synchronization
  - Excessive use of global data, contention for the same synchronization object

- Parallel Overhead
  - Due to thread creation, scheduling..

- Granularity
  - No sufficient parallel work
Load Imbalance

- Unequal work loads lead to idle threads and wasted time
Redistribute Work to Threads

- **Static assignment**
  - Are the same number of tasks assigned to each thread?
  - Do tasks take different processing time?
    - Do tasks change in a predictable pattern?
      - Rearrange (static) order of assignment to threads
    - Use dynamic assignment of tasks

- **Dynamic assignment**
  - Is there one big task being assigned?
    - Break up large task to smaller parts
  - Are small computations agglomerated into larger task?
    - Adjust number of computations in a task
    - More small computations into single task?
    - Fewer small computations into single task?
    - Bin packing heuristics
Synchronization

- By definition, synchronization serializes execution.
- Lock contention means more idle time for threads.
Synchronization Fixes

- Eliminate synchronization
  - Expensive but necessary “evil”
  - Use storage local to threads
    - Use local variable for partial results, update global after local computations
    - Allocate space on thread stack (alloca)
    - Use thread-local storage API (TlsAlloc)
  - Use atomic updates whenever possible
    - Some global data updates can use atomic operations (Interlocked API family)
Synchronization Fixes

- Reduce size of critical regions protected by synchronization object
  - Larger critical regions tie up sync objects longer; other threads sit idle longer waiting to acquire objects
  - Only accesses to shared variables need to be protected
Synchronization Fixes

- Use best synchronization object for job
  - Critical Section
    - Local object
    - Available to threads within the same process
    - Lower overhead (~8X faster than mutex)
  - Mutex
    - Kernel object
    - Accessible to threads within different processes
    - Deadlock safety (can only be released by owner)

- Other objects are available
General Optimizations

- Serial Optimizations
  - Serial optimizations along the critical path should affect execution time

- Parallel Optimizations
  - Reduce synchronization object contention
  - Balance workload
  - Functional parallelism

- Analyze benefit of increasing number of processors
- Analyze the effect of increasing the number of threads on scaling performance
Software Development Tools for Multi-core Computer System

- Part5 Open source tools : GCC GDB
GNU C Programming Steps

Edit the source code with vi or emacs

Edit Makefile

Use “make” command to compile the project

Debug and modify the source code
How to Produce a Executable

link/load map

.o object

Shared libs

Static libs

command-line arguments

linker control files

dump symbol file

executable file

linker control files

dump symbol file

link/load map
 GCC

- GNU C compiler gcc
  - gcc [ option | filename ]
- gcc has nearly 100 options
- If you do not declare any library file directories or header file directories, gcc just uses default directory.
  - Default libc related header files searching directory /usr/include /usr/local/include /usr/lib/gcc-lib/i386-redhat-linux/3.2.2/include
  - Default library searching directory: /lib, /usr/lib
- The main functions of gcc is:
  - Precompile source code
  - Produce object file from source code
  - Use GNU ld to finish link work, then to produce executable file
GCC

Gcc options
- Output the object file without link : -c
  [root@localhost root]#gcc -c hello.c
- Compile and link, output the execute file : -o
  [root@localhost root]#gcc hello.c -o hello
- Find the head file : -l
  [root@localhost root]#gcc hello.c -o hello -l/usr/include
- See the version of the gcc
  [root@localhost root]#gcc -v
- Print the warning messages
  [root@localhost root]#gcc -Wall hello.c -o hello
- Find the lib file : -L
  [root@localhost root]# gcc hello.c -o hello –L/usr/local/lib
GCC

gcc optimize option : -O -O1 -O2 -O3 -O0 -Os

- O -O1
  Just reduce the program’s size, improve the executing speed. Not do the optimization need a lot of time and space

- O2
  Ordinary options. Almost do all the optimization that gcc can do. Compiling is slow. It is the tradeoff of the space and the speed of the code

- O3
  The highest optimization, include all the optimize methods in the -O2 option

- O0
  Without any optimization

- Os
  Optimize the length of the code only
GDB options

- Run: Run the program
- break NUM: Set the break point at the selected line
- bt: view all the stack which are used
- Clear: delete all the break points in the specifically source code
- continue: Continue the program
- display EXPR: when the program stopped, display the value of the express which is composed by the variables
- file FILE: load the executed file to debug
- help NAME: display the help information of the command
- info break: display the list of the break points, include the times get to the break point
- info files: display the detailed information of the file debugged
- info func: display the name of all the functions
- info local: display the information of the local variable in the function
- info prog: display the state of the program debugged
- info var: display the name of all the static variable
- kill: stop the program debugged
- list: show the source code
- make: run the make tools without exit the gdb
- next: execute the next line of the source code
- print EXPR: print the value of the express of the EXPR
- quit: quit the gdb
References

Thank You