GDB on Windows
status & plans

Pedro Alves
Agenda

• Windows debug API particularities
• Non-stop mode, and how we're planning on implementing it on Windows
• Ctrl-C handling and what is different on Windows
• The GDB testsuite, why nobody is running it on native Windows, and what can we do about it.
• PDB (Portable Database), Microsoft's debug info format
• More
Windows debug API core

BOOL WaitForDebugEvent(
    [out] DEBUG_EVENT *lpDebugEvent,
    [in] DWORD dwMilliseconds
);

BOOL ContinueDebugEvent(
    [in] DWORD dwProcessId,
    [in] DWORD dwThreadId,
    [in](DWORD dwContinueStatus
);

DWORD SuspendThread(
    [in] HANDLE hThread
);

DWORD ResumeThread(
    [in] HANDLE hThread
);
Windows debug API, WaitForDebugEvent

```c
BOOL WaitForDebugEvent(
    [out] DEBUG_EVENT **lpDebugEvent,
    [in] DWORD dwMilliseconds  // 0 - return immediately; INFINITE - wait forever
);
```

- **Note:** not asynchronous.

- To avoid blocking must either:
  - periodically poll, or,
  - call from separate thread. <<< what GDB does.

- Must be called from the thread that attached or spawned the inferior.
  - Must make most debug API calls from that separate thread. << what GDB does.
BOOL ContinueDebugEvent(
    [in] DWORD dwProcessId,
    [in] DWORD dwThreadId,
    [in] DWORD dwContinueStatus
);

Where dwContinueStatus can be:

- **DBG_CONTINUE**
  - If the thread previously reported EXCEPTION_DEBUG_EVENT, stop all exception processing, the exception is marked as handled.

- **DBG_EXCEPTION_NOT_HANDLED**
  - If the thread previously reported EXCEPTION_DEBUG_EVENT, continue exception processing. If this is a first-chance exception event, the search and dispatch logic of the structured exception handler is used; otherwise, the process is terminated.
Async mode

- GDB's event loop reacts to multiple event sources at the same time
- target events + user input
- Background execution commands:
  
  ```
  ...
  (gdb) c&
  Continuing.
  (gdb)
  ```

- Most importantly
  - => let GUIs/IDEs communicate with GDB while inferior is running
    (read memory, set breakpoints, symbol queries, etc., etc.)
All-stop mode

1. [R] [R] [R] [R] [R] <<< all threads running free, T3 about to hit exception
2. [k] [k] [E] [k] [k] <<< T3 hit exception, kernel pauses whole process
3. ... <<< user inspects T3, backtrace, prints variables, etc.
4. [k] [k] [E] [k] [k] <<< user resumes, GDB issues ContinueDebugEvent(T3, DBG_CONTINUE or DBG_EXCEPTION_NOT_HANDLED)
5. [R] [R] [R] [R] [R] <<< all threads running free again

[R] - runnable (suspend count == 0)
[k] - suspended by kernel
[E] - exception event, suspended by kernel
[S] - suspended by GDB (suspend count == 1)
All-stop mode + "set scheduler-locking on"

1. [R] [R] [R] [R] [R] <<< all threads running free, T3 about to hit exception
2. [k] [k] [E] [k] [k] <<< T3 hit exception, kernel pauses whole process

3. ... <<< user inspects T3, backtrace, prints variables, etc.

4. [k] [S] [S] [S] [S] <<< user decides to resume only thread T1, suppress exception, GDB uses SuspendThread to freeze threads T2-T5, and is about to issue ContinueDebugEvent(T3, DBG_CONTINUE)

5. [R] [S] [S] [S] [S] <<< T1 running free again, others suspended

ciągły - runnable (suspend count == 0)  
wyjątek - exception event, suspended by kernel  
suspended by kernel  
suspended by GDB (suspend count == 1)
Non-stop mode

- only the thread that hits breakpoint/event reports stop to user
- other threads keep running
- only supported on GNU/Linux, and remote targets (some embedded systems)

```
... Thread 6 "pthreads" hit Breakpoint 3, thread2 (arg=0xdeadbeef)
at gdb.threads/pthreads.c:90
90    k += i;
(gdb) info threads
 Id  Target Id          Frame
  *1 Thread 4980.0x17b8 "pthreads" (running)
  2 Thread 4980.0x664 (running)
  3 Thread 4980.0xa50 (running)
  4 Thread 4980.0x154 "sig" (running)
  5 Thread 4980.0x91c "pthreads" (running)
  6 Thread 4980.0xad8 "pthreads"  thread2 (arg=0xdeadbeef)
at gdb.threads/pthreads.c:90
(gdb)
```
Non-stop mode plans

Non-stop mode **problem** on Windows:

WaitForDebugEvent returns an event => **kernel suspends the whole process** (all its threads)

Conflict: **non-stop wants to leave all other threads running**!

Easy, immediately SuspendThread the event thread (not others), and call ContinueForDebugEvent, right?

... not so fast!

The user hasn't decided yet whether to pass the exception to the inferior or not!
Non-stop mode plans, Win10 to the rescue

BOOL ContinueDebugEvent(
    [in] DWORD dwProcessId,
    [in] DWORD dwThreadId,
    [in] DWORD dwContinueStatus
);

Where dwContinueStatus can now also be:

- **DBG_REPLY_LATER**
  - "Supported in Windows 10, version 1507 or above, this flag causes `dwThreadId` to replay the existing breaking event after the target continues. By calling the `SuspendThread` API against `dwThreadId`, a debugger can resume other threads in the process and later return to the breaking."
Non-stop, sequence of events

1. [R] [R] [R] [R] [R] [R] <<< all threads running free, T3 about to raise exception
2. [k] [k] [E] [k] [k] [k] <<< T3 raises exception, kernel pauses whole process
3. [k] [k] [S] [k] [k] [k] <<< GDB suspends T3 (SuspendThread => suspend count == 1)
4. [R] [R] [S] [R] [R] [R] <<< GDB issues ContinueDebugEvent(T3, DBG_REPLY_LATER), remembers event will be repeated
5. ... <<< user inspects T3, backtrace, prints variables, etc.
6. [R] [R] [R] [R] [R] [R] <<< user resumes T3, GDB unsuspends T3 (ResumeThread => suspend count == 0)
7. [k] [k] [E] [k] [k] [k] <<< T3 immediately re-reports exception, kernel pauses whole process
8. [R] [R] [R] [R] [R] [R] <<< GDB issues ContinueDebugEvent(T3, DBG_CONTINUE or DBG_EXCEPTION_NOT_HANDLED)

R – runnable (suspend count == 0)  E – exception event, suspended by kernel
k – suspended by kernel  S – suspended by GDB (suspend count == 1)
Non-stop, multiple events works too

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. [R] [R] [R] [R] [R] [R]  <<< all threads running free, T3 about to raise exception
2. [k] [k] [E] [k] [k] [k]  <<< T3 raises exception, kernel pauses whole process
3. [k] [k] [S] [k] [k] [k]  <<< GDB suspends T3 (SuspendThread => suspend count == 1)
4. [R] [R] [S] [R] [R] [R]  <<< GDB issues ContinueDebugEvent(T3, DBG_REPLY_LATER), remembers event will be repeated

6. [E] [k] [S] [k] [k] [k]  <<< T1 raises exception, kernel pauses whole process
7. [S] [k] [S] [k] [k] [k]  <<< GDB suspends T1 (SuspendThread => suspend count == 1)
8. [S] [R] [S] [R] [R] [R]  <<< GDB issues ContinueDebugEvent(T1, DBG_REPLY_LATER), remembers event will be repeated

9. [S] [R] [R] [R] [R] [R]  <<< user resumes T3, GDB unsuspends T3 (ResumeThread => suspend count == 0)
A. [S] [k] [E] [k] [k] [k]  <<< T3 immediately re-reports exception, kernel pauses whole process
B. [S] [R] [R] [R] [R] [R]  <<< GDB issues ContinueDebugEvent(T3, DBG_CONTINUE or DBG_EXCEPTION_NOT_HANDLED)
Non-stop mode plans, there's more to it

There's more to it, but no time to go through it all today.

- Cygwin signal handling details
- Watchpoints support details
- SuspendThread accounting messy details
- Passing signal to right thread details
- $_siginfo per thread

Also, we have a few downstream Cygwin GDB patches, some of which we need to upstream:

- Unwind cygwin _sigbe and sigdelayed frames
- Drop special way of getting inferior context after a Cygwin signal
- Use cygwin pgid if inferior is a cygwin process
- Others...
GDB on Windows, two ports

Cygwin
- Cygwin is: "a DLL (cygwin1.dll) which provides substantial POSIX API functionality."
- You rebuild your application from source.
- Application aware of UNIX® functionality like signals, ptys, etc.
- C runtime / headers based on newlib.

MinGW [1]
- Port of GCC compiler to Windows systems, and other tools (binutils, .def and .idl files, etc.)
- Windows API Headers, C runtime headers, everything needed for linking and running code on Windows
- C runtime / headers based on MSVCRT.

The Cygwin GDB port uses posix signals, ptys, select/poll event loop, etc.
The MinGW GDB port uses WaitForMultipleObject event loop, etc.

Both ports share the backend code that talks to the Windows debug API (gdb/windows-nat.c) [1] - there are two MinGW projects, but we can ignore that fact here
GDB testsuite

• Built on DejaGnu => Built on expect => Built on TCL
• DejaGnu assumes Unix-like environment:
  o Posix shell and utilities, "kill", "cp", "mv", etc.
  o There is no Windows native expect port
• Testing a Cygwin GDB on a Cygwin environment works
  o Slow & not super stable, but works
  o But, not the same as native MinGW GDB
• MinGW GDB under Cygwin/Msys2
  o Windows GDB running under Cygwin expect sees input/output connected to a pipe, not an interactive pty
    => GDB disables interactive/readline mode
  o Terminal mode handling => CodeSourcery's cygwin-wrapper tool could help here?
  o Path mapping issues (what GDB sees != what testcases see)
• GDB's multi-threading tests use pthreads
  o Native Windows doesn't have that => MinGW-w64 has them w/ winpthreads, though
• Ideas?
  o Run DejaGnu on Cygwin / Msys2, spawn MinGW GDB? => need GDB hackery?
  o Run DejaGnu on GNU/Linux, spawn MinGW GDB on remote host? => where GNU/Linux could be WSL
  o Other?
GDB testsuite

- BTW, compiling GDB on Cygwin is ... sloooooooooooooooooooooooow
- Solution – cross compile from GNU/Linux
  o On Fedora, just install the cygwin cross compiler packages found in yselkowitz's Fedora copr:
    • [https://copr.fedorainfracloud.org/coprs/yselkowitz/cygwin/](https://copr.fedorainfracloud.org/coprs/yselkowitz/cygwin/)
  o Elsewhere, you can use my cygwin-cross wrapper – a docker container that pulls in yselkowitz's packages:
    • [https://github.com/palves/cygwin-cross](https://github.com/palves/cygwin-cross)

- Cross compile from GNU/Linux
- SMB-mount GNU/Linux build dir on Windows
- Run testsuite in Cygwin, inside Windows
- Configure just the testsuite (not the whole of gdb), and then run make check:

```
$ /path/to/src/gdb/testsuite/configure
$ make check-parallel -j8 RUNTESTFLAGS="
    GDB=/cygdrive/x/gdb/build-cygwin-cross/gdb/gdb 
    GDB_DATA_DIRECTORY=/cygdrive/x/gdb/build-cygwin-cross/gdb/data-directory"
```
GDB testsuite

- Testsuite on Cygwin, a struggle

- Sloooooooow

- Flaky

- Infinite hangs
  - Needs hand holding – kill gdb processes to unblock rest of run
  - Mitigated by skipping tests we know can't work, like fork tests
  - Remaining hangs odd => GDB hangs forever on exit, after DejaGnu closed stdout

- Lots of tests fail because regexps assume single-threaded
  - But all Cygwin programs are multi-threaded => adjust tests, busy work
PDB (Program Database)

- Microsoft's native debug info format
- It's not DWARF
- Proprietary, undocumented for many years
- Windows native dlls to read it
  - DIA SDK, dbghlp.dll
- MSFT provided a code dump of a reader on github a few years back
- LLVM since developed library to read PDB
- Other libraries appeared
- GCC patches to make GCC emit PDB
- No GDB patches
More IWBN features

• Microsoft C++ ABI
  • Structure layout
  • Name mangling (decoration) Scheme
    Most ABIs use the Itanium C++ ABI, and its mangling scheme
    ```
    $ echo _ZNSt6vectorIPKcSaIS1_E9push_backEOS1_ | c++filt
    std::vector<char const*, std::allocator<char const*> >::push_back(char const*&&)
    ```
    Microsoft has its own scheme

• Calling convention(s)
  • Calling functions in inferior
  • `finish/return` commands

• Exception handling
  • catch catch
  • catch throw
  • Intercept exceptions when stepping
The End
Non-stop mode plans

• GDB 13 made it possible to handle input and inferior events at the same time by moving this:

```c
BOOL WaitForDebugEvent(
    [out] LPDEBUG_EVENT lpDebugEvent,
    [in] DWORD dwMilliseconds
);
```

... to a separate thread.
Windows debug API particularities

To detach from an inferior:

```c
BOOL DebugActiveProcessStop(
    [in] DWORD dwProcessId
);
```

Must be called from the thread that started debugging the process..

..but if that thread is blocked waiting for events with "WaitForDebugEvent(INFINITE)"?

=> Can't detach!
Windows debug API particularities

Solution: force inferior process to report an event

// raise breakpoint trap
BOOL DebugBreakProcess(
    [in] HANDLE Process
);

// raise ctrl-c
BOOL WINAPI GenerateConsoleCtrlEvent(
    _In_ DWORD dwCtrlEvent,
    _In_ DWORD dwProcessGroupId
);

Awkward as forces the inferior to spawn a new thread.
Windows debug API particularities

Awkward as they force the inferior to spawn a new thread.

Would prefer if debug events were reported via standard `WaitForMultipleObjects` instead of `WaitForDebugEvent`.

Could then wait for both, simultaneously:

- debug events
- a Windows event (`SetEvent`) used to unblock the thread

But that's not how it works...
Non-stop mode plans, Win10 to the rescue

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>[R]</td>
<td>[R]</td>
<td>[R]</td>
<td>[R]</td>
</tr>
<tr>
<td>2.</td>
<td>[k]</td>
<td>[k]</td>
<td>[E]</td>
<td>[k]</td>
</tr>
<tr>
<td>3.</td>
<td>[k]</td>
<td>[k]</td>
<td>[S]</td>
<td>[k]</td>
</tr>
<tr>
<td>4.</td>
<td>[R]</td>
<td>[R]</td>
<td>[S]</td>
<td>[R]</td>
</tr>
</tbody>
</table>

5. ... <<< user inspects T3, backtrace, prints variables, etc. |

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6.</td>
<td>[R]</td>
<td>[R]</td>
<td>[R]</td>
<td>[R]</td>
</tr>
<tr>
<td>7.</td>
<td>[k]</td>
<td>[k]</td>
<td>[E]</td>
<td>[k]</td>
</tr>
<tr>
<td>8.</td>
<td>[R]</td>
<td>[R]</td>
<td>[R]</td>
<td>[R]</td>
</tr>
</tbody>
</table>

- **R** - runnable (suspend count == 0)
- **E** - exception event, suspended by kernel
- **k** - suspended by kernel
- **S** - suspended by GDB (suspend count == 1)
Non-stop mode plans, Win10 to the rescue

1. WaitForDebugEvent reports event for thread T // kernel suspends all the threads
2. SuspendThread thread T // we want T to be remain suspended after ContinueDebugEvent
3. ContinueDebugEvent DBG_REPLY_LATER // sets all other threads running free again
4. Record that we're expecting a repeated DBG_REPLY_LATER kernel event
5. Report event for T to GDB core

Later:
1. User resumes thread T again, decides to pass or not exception down
2. We record in T's data structure whether to pass exception down or not
3. ResumeThread thread T
4. Due to earlier DBG_REPLY_LATER, kernel reports same event for T again
5. GDB knows it is expecting the repeated event for T, and calls ContinueDebugEvent immediately:
   o with either DBG_CONTINUE or DBG_EXCEPTION_NOT_HANDLED appropriately