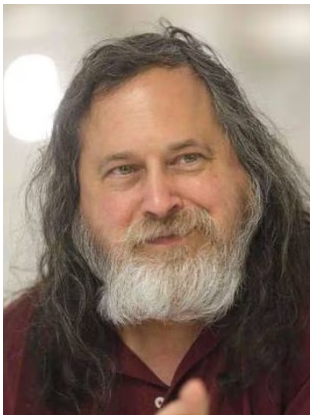
O ambiente de desenvolvimento GNU


Richard Stallman (1953)

O projeto "GNU" – que é a abreviação recursiva de "*Gnu is Not Unix*" – começou na década de 80 com o trabalho de Richard Stallman: seu objetivo era a criação de um sistema operacional completo, totalmente gratuito e sem nenhuma restrição de uso (*free-software* e *open-source*), baseado no Unix. Já que o próprio nome "Unix" era disputado como marca registrada, o sistema operacional foi chamado sarcasticamente de "*not* Unix".

Um dos primeiros impasses para o novo sistema viria a ser o seu próprio ambiente de desenvolvimento: não fazia sentido usar montadores, compiladores e vinculadores (*linkers*) comerciais – cuja licença era, portanto, *fechada* – para o desenvolvimento de um software aberto; qualquer usuário que pretendesse reconstruir o sistema a partir de seu código-fonte precisaria adquirir legalmente uma licença desses softwares, o que violaria diretamente a filosofia original.

Sendo assim, Stallman e sua equipe começaram a desenvolver *suas próprias ferramentas*, entre as quais um montador (`as`), vinculador (`ld`), arquivador (`ar`), compilador `C` e suas respectivas bibliotecas (`gcc`), depurador (`gdb`) e até um editor de texto (`emacs`). Hoje em dia essas ferramentas já foram adaptadas para praticamente todas as arquiteturas e sistemas operacionais, permitem compilar diversas linguagens de alto nível (de fato, "gcc" passou a ser a abreviatura de "*gnu compiler collection*") e foram utilizadas para o desenvolvimento de vários sistemas operacionais, entre eles o BSD, Darwin (MacOS), Linux além, é claro, do próprio Hurd (que é o núcleo do sistema operacional Gnu).

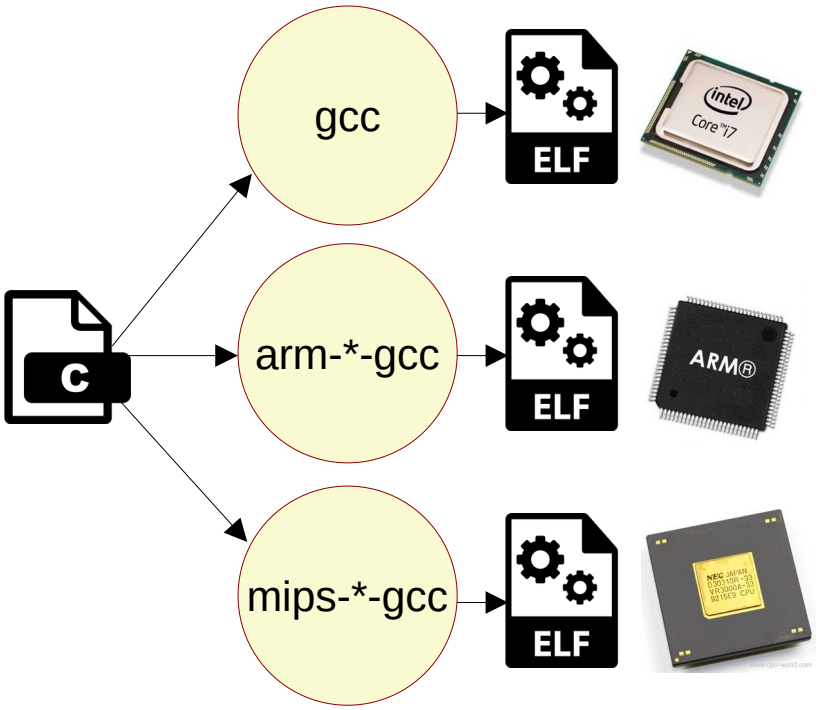
No Laboratório de Microprocessadores vamos utilizar esses utilitários no sistema Linux:

- **`gcc`** – programa principal do compilador C, que automaticamente chama diversos outros programas para a produção de arquivos contendo instruções de máquina, conforme a necessidade (**`as`** para arquivos em *assembly*, **`ld`** para gerar um arquivo carregável pelo sistema operacional, etc.).
- **`objdump`** e **`readelf`** – programas auxiliares para analisar o conteúdo de arquivos binários (geralmente produzidos pelo **`gcc`**), por exemplo, para visualizar o programa no formato de sequência de instruções ("desmontagem" ou *disassembly*).
- **`gdb`** – programa depurador que, com a ajuda do sistema operacional, permite monitorar um outro programa durante sua execução, podendo interrompê-lo, restaurá-lo, visualizar o conteúdo de memória, etc.

De um modo geral, esses softwares produzem e analisam código para a *mesma* arquitetura que os estão executando: em um computador AMD-64 vão gerar e analisar instruções características do conjunto de instruções dessa arquitetura. No entanto, a linguagem de máquina de destino é completamente arbitrária e poderia, eventualmente, corresponder a uma arquitetura *diferente* da arquitetura da máquina que executa esses programas: vamos chamar a máquina que executa o ambiente de desenvolvimento de *hospedeira* ("*host*") e a máquina que executará as instruções produzidas de *alvo* ("*target*"). No caso em que as

arquiteturas hospedeira e alvo são a mesma, esse processo é chamado de *desenvolvimento nativo*; quando são arquiteturas diferentes, chamaremos de *desenvolvimento cruzado*.

No caso particular do **`gcc`** e programas associados, desenvolvimentos cruzados são sempre tornados explícitos por uma nomenclatura especial dos aplicativos: o seu *prefixo*. Assim, o programa chamado "**`as`**" sempre produz código para a mesma arquitetura da máquina hospedeira (*assembly* nativo); um programa (diferente), cujo nome será algo como "**`mips-linux-gnueabi-as`**", vai produzir código para ser executado por uma arquitetura MIPS. A parte "mips-linux-gnueabi-" é chamada de *prefixo*[1] e sempre vai identificar a produção de código para uma arquitetura diferente da arquitetura da máquina hospedeira.



No Laboratório de Microprocessadores estudaremos a arquitetura ARM (*Advanced RISC Machine*), mas trabalharemos com os computadores do laboratório, que são máquinas Intel. Já sabemos como podemos gerar código para uma arquitetura diferente, porém como faremos para *executar* as instruções do processador ARM?

1 Preste atenção que o *prefixo* traz outras informações, além da arquitetura da máquina alvo, a saber: o sistema operacional alvo e o formato de interface binária (ABI), que informa, entre outras coisas, como diferentes tipos de dados são representados na memória e como parâmetros são passados para as funções nessa arquitetura.

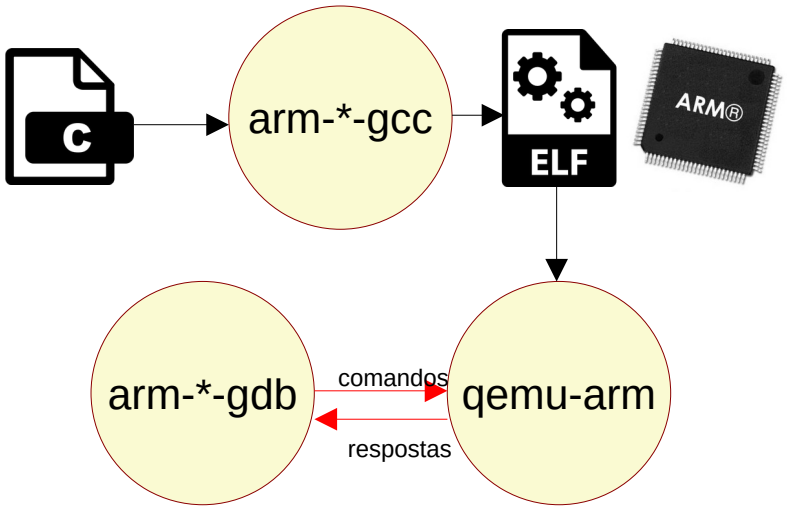
O Emulador de Arquiteturas (QEMU)

Fabrice Bellard (1972)

O software *Quick Emulator* ou QEMU foi desenvolvido como software livre por Fabrice Bellard[2]. Sua principal função é imitar ou emular a execução de instruções de diferentes processadores, independentemente da máquina que o executa (denominada *máquina hospedeira* ou *host*), através de *tradução binária*: processo no qual uma ou mais instruções de máquina do computador hospedeiro são usadas para produzir um efeito *equivalente* ao da execução de uma instrução de uma arquitetura diferente (ou eventualmente da *mesma* arquitetura). O *Quick Emulator* também é utilizado para a criação de *máquinas virtuais*, emulando o funcionamento de um *hardware* eventualmente diferente do equipamento real.

De forma equivalente ao objetivo dos *prefixos* do Gnu, são os *sufixos* que identificam a arquitetura emulada por QEMU. Por exemplo, o programa de nome "**`qemu-arm`**" vai emular a execução de código que deve conter instruções do processador ARM, mesmo que a máquina local não tenha um processador compatível. Entre as arquiteturas que podem ser emuladas pelo `qemu` estão Intel x86, AMD-64, ARM, ARM-64, MIPS, MIPS-64, PowerPC, Sparc, e várias outras.

O modelo geral que utilizaremos no Laboratório envolve compilar ou montar programas contendo instruções do processador ARM usando o `gcc` cruzado (**`arm-linux-gnueabi-gcc`**) e executá-los nos computadores de arquitetura Intel com o emulador `qemu` (**`qemu-arm`**); vamos observar o comportamento da execução emulada através do depurador Gnu, **`gdb`** (**`arm-linux-gnueabi-gdb`** ou **`gdb-multiarch`**). Para isso, os processos do emulador `qemu` e do depurador vão se conectar por um *socket*: comandos podem ser enviados do depurador para o emulador, que serão respondidos, eventualmente interrompendo a execução do programa emulado para a análise dos valores dos registradores, memória, etc.



2 O QEMU é o principal software de virtualização no Linux, com suporte ao *driver* KVM (*kernel-based virtual machine*). Você provavelmente já conhece vários outros programas que foram desenvolvidos por Bellard, tais como o FFmpeg, QuickJS, TinyC *compiler* e vários softwares para compactação de dados.

O Depurador do Gnu

O depurador do sistema Gnu (Gnu *Debugger* ou `gdb`) é um software com muitos recursos para analisar, depurar e alterar outro programa, na maior parte das vezes *enquanto esse programa é executado*. Dessa forma, é possível observar a ocorrência de eventos (frequentemente erros) em tempo real e analisar as condições que favoreceram tais eventos. Além disso, com a ajuda do sistema operacional, o `gdb` é capaz de conectar-se a um programa que já começou sua execução (sem o depurador) ou a um programa que não esteja mais em execução: lendo uma cópia em disco de suas áreas de memória (o que é chamado de "*core dump*"). Alguns dos recursos oferecidos pelo `gdb` são:

- Sincronização entre o ponto de execução atual e o código-fonte do programa, caso disponível;
- Interrupção da execução do programa a qualquer momento (*break*);
- Interrupção do programa depurado em pontos de parada (*breakpoints*) definidos pelo usuário em endereços de memória arbitrários;
- Monitoramento do estado de posições de memória (variáveis, símbolos em geral), podendo provocar interrupção do programa depurado, conforme seu novo valor (*watchpoints*), de forma semelhante aos *breakpoints*;
- Execução passo a passo, seja por linhas do código-fonte (em C ou outra linguagem), seja por instruções de máquina individuais (*assembly*);
- Visualização da memória em diversos formatos: mapa de memória, caracteres, *strings*, números inteiros (hexadecimal, octal, decimal ou binário), números em ponto flutuante, instruções de máquina (*disassembly*), etc.;
- Visualização do *contexto* do processador através dos registradores internos;
- Visualização do estado do programa a partir da pilha de chamadas de sub-rotinas;
- Alteração (edição) do ponto de execução do programa: execução de saltos, reinícios (*reset*), encerramento do programa (*exit*), etc.;
- Alteração (edição) de qualquer posição de memória (desde que a escrita nessa posição seja permitida) e dos registradores do processador.

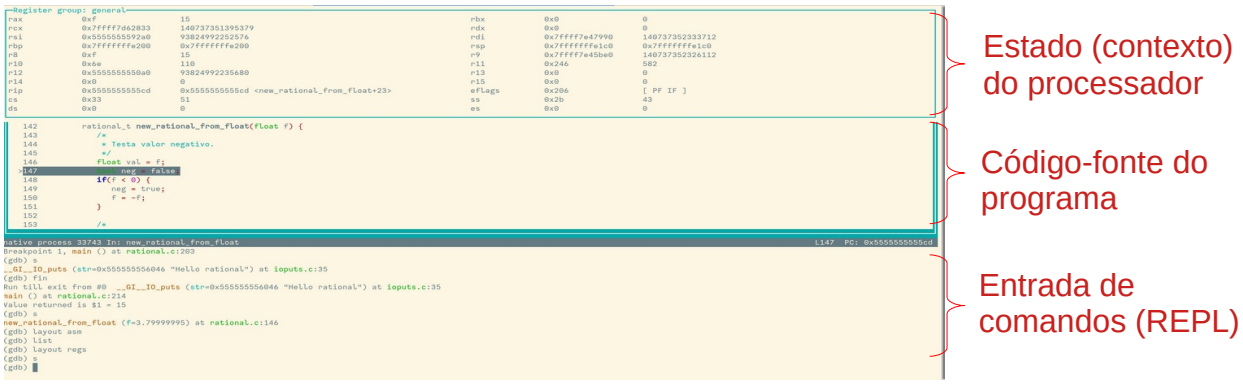
O depurador pode ser um programa único, executado pela mesma máquina e sistema operacional que executa o programa depurado ou pode ser dividido em duas partes que se comunicam (por exemplo, através de uma rede de computadores, uma porta USB ou um canal serial). Nesse último caso, o programa depurado pode estar em *outra máquina*, eventualmente em uma arquitetura diferente e ser executado por um sistema operacional diferente (ou mesmo não possuir qualquer sistema operacional envolvido). A parte do `gdb` que é executada na máquina de teste, juntamente com o programa depurado, é chamado de *stub*. Essa é uma situação comum quando depuramos um programa embarcado (em uma placa com um microcontrolador ou um aplicativo em um telefone celular, por exemplo) ou quando depuramos o próprio *kernel* do Linux executando em uma outra máquina. No caso do sistema emulado com o `qemu`, o *stub* está incluído no próprio programa do emulador.

O `gdb` ou o seu *stub* podem ser controlados por qualquer programa diretamente, através de troca de mensagens, mas também existem *bibliotecas* de sistema (tais como `libgdb.so`) que oferecem uma interface de programação (API) de nível mais alto, que facilita a integração. Mas

o uso mais comum é através de um aplicativo de linha de comando ("gdb"): esse é um aplicativo tipo *read-eval-print-loop* ou REPL: aguarda um comando do usuário via terminal, executa o comando, mostra de volta o resultado do comando e repete o processo indefinidamente.

Alguns dos comandos (tais como "*run*" e "*continue*") transferem o controle ao programa em depuração e somente vão retornar com algum resultado quando esse programa for encerrado ou interrompido por uma exceção ou por um *breakpoint*. Outros comandos são interativos, respondendo imediatamente. Em qualquer momento, a interrupção do programa em execução pode ser forçada pelo usuário do gdb com a combinação de teclas *break* (geralmente Control + C): neste caso, o *loop* de interpretação de comandos retorna.

É importante notar que enquanto o depurador aguarda um comando do usuário, o processo em teste não está sendo executado e permanece em seu último estado e contexto.



Os comandos mais importantes do gdb em linha de comando serão descritos nos próximos parágrafos.

Comandos de execução e controle do processo

- "*run*" ou "r" – executa o programa a partir do início, somente retornando quando terminar, ocorrer uma exceção, encontrar um *breakpoint* ou receber um comando de interrupção (Control + C);
- "*continue*" ou "c" – executa o programa a partir da posição atual, somente retornando quando terminar, ocorrer uma exceção, encontrar um *breakpoint* ou receber um comando de interrupção (Control + C);
- "*step*" ou "s" – executa a próxima linha do código-fonte ou a próxima instrução de máquina (se estiver no modo *assembler*), retornando em seguida;
- "*next*" ou "n"– executa a próxima linha do código-fonte ou a próxima instrução de máquina (se estiver no modo *assembler*), retornando em seguida. Caso seja uma *chamada de sub-rotina* (função, procedimento, método, etc.), executa a sub-rotina inteira antes de retornar;

  Os comandos "*continue*", "*step*" ou "*next*" podem incluir um *número de repetições*. Nesse caso, o comando será repetido esse número de vezes antes de retornar ao *loop* de comandos.

Exemplos:

`step 3`	Executa as próximas três linhas (ou três instruções)
`next 5`	Executa as próximas cinco linhas/instruções, pulando sub-rotinas
`c 10`	Somente pára após dez interrupções do programa (*breakpoints*)

- "*finish*" ou "fin" – executa a sub-rotina atual até o final e retorna;
- "*kill*" – encerra o programa;
- "*backtrace*", "ba" ou "*where*" – mostra o ponto de execução atual, incluindo todas as chamadas de sub-rotinas na pilha do sistema.
- Se for enviado um comando *vazio* (ou seja, pressionar a tecla <enter> sem digitar nenhum comando), o `gdb` vai **repetir** o comando anterior, o que é muito útil para executar sucessivas linhas com "*step*", "*continue*" ou "*next*", ou ainda para continuar a visualização da memória com os comandos "*list*" ou "x".

Breakpoints

Comando "break"

- O comando "*break* <local>" ou "b <local>" introduz um novo ponto de parada (*breakpoint*) no local especificado.

  O parâmetro <local> pode ser um *endereço* em memória, o *nome* de uma função, um *rótulo*, um *número de linha* referente ao arquivo-fonte atual ou de outro arquivo, no formato "nome do arquivo:número de linha". Se <local> for omitido, assume-se a posição atual.

Exemplos:

`b`	*Breakpoint* na posição atual
`break +3`	*Breakpoint* daqui a três linhas (ou três instruções)
`break main`	*Breakpoint* na entrada da função main()
`b main+2`	*Breakpoint* a segunda linha da função main()
`b *0x5555555555e4`	*Breakpoint* no endereço virtual
`break teste.c:54`	*Breakpoint* na linha 54 do arquivo teste.c
`b 63`	*Breakpoint* na linha 63 do arquivo atual

É importante frisar que, para que o depurador possa encontrar os símbolos definidos no código-fonte (por exemplo, "*main*") é necessário que *a tabela de símbolos esteja presente no arquivo executável*: algo que normalmente é evitado, para economizar espaço. O programa depurado deve ter sido compilado ou montado com as opções `–g` ou `–ggdb`, que obrigam o compilador (na verdade, o vinculador ou *linker*) a copiar explicitamente a tabela de símbolos no arquivo de saída. Além disso, é conveniente reduzir – ou mesmo desabilitar – a realização de *otimizações* pelo compilador, que podem eventualmente alterar a disposição das instruções no código-objeto, dificultando o entendimento da saída do depurador. Para isso, pode-se utilizar a opção `–O` do compilador.

Comando "break" condicional

- O comando "*break* <local> *if* <condição>" permite definir uma **condição** para que a parada no *breakpoint* aconteça (*breakpoint* ***condicional***).

  O parâmetro <local> pode ser um *endereço* em memória, o *nome* de uma função, um *rótulo*, um *número de linha* referente ao arquivo-fonte atual ou de outro arquivo, no formato "nome do arquivo:número de linha". Se <local> for omitido, assume-se a posição atual.

  O parâmetro <condição> pode ser qualquer expressão da linguagem `C`, envolvendo quaisquer símbolos definidos (no escopo global ou local, em relação à posição do *breakpoint*), ponteiros (endereços) e constantes. Caso a expressão seja avaliada com um valor *diferente de zero*, o programa será interrompido no ponto de parada definido; do contrário, o ponto de parada é ignorado.

Exemplos:

`break if ok==0`	*Breakpoint* condicional na posição atual
`b loop+5 if data[4]>=8`	*Breakpoint* condicional em um rótulo
`b teste.c:32 if (a>0) && (i>8)`	*Breakpoint* condicional na linha 32 do arquivo

Edição de breakpoints

- "*info breakpoints*" ou "i b" mostra todos os *breakpoints* definidos, ativos ou não.

A informação mais importante da lista de *breakpoints* é o **índice** de cada *breakpoint*. É a partir desse número que os comandos a seguir podem identificar um *breakpoint* específico para alterá-lo, desabilitá-lo ou removê-lo completamente:

- "*condition* <índice> <condição>" – muda ou acrescenta uma condição ao *breakpoint* cujo índice é especificado no comando;
- "*ignore* <índice> <número>" – ignora o "<número>" de ocorrências do *breakpoint* **antes** de interromper o programa e retornar ao *loop* de comandos. Isso permite que um *breakpoint* somente seja acionado após uma quantidade de ocorrências;
- "*disable* <índices>" – desabilita o(s) *breakpoint*(s) cujo(s) índice(s) é(são) especificado(s) no comando. O(s) *breakpoint*(s) poderá(ão) ser reabilitado(s) no futuro;
- "*enable* <índices>" – habilita o(s) *breakpoint*(s) cujo(s) índice(s) é(são) especificado(s) no comando;
- "*delete* <índices>" – remove permanentemente o(s) *breakpoint*(s).

Caso o valor de <índices> seja omitido nos comandos "*disable*", "*enable*" ou "*delete*", a operação afetará **todos** os *breakpoints* existentes.

Exemplos:

`ignore 1 10`	Somente aciona o *breakpoint* 1 após dez ocorrências
`ignore 2 2`	Aciona o *breakpoint* 2 uma vez sim, uma vez não
`disable 3 4 5`	Desabilita os *breakpoints* de índices 3, 4 e 5
`enable 4`	Reabilita o *breakpoint* 4
`delete 3 5`	Remove os *breakpoints* 3 e 5

Obtendo informações sobre os arquivos-fonte e funções

- O comando "info sources <expressão>" mostra informações sobre todos os arquivos-fonte que correspondam à expressão regular <expressão>;
- O comando "info functions <expressão>" mostra todos os nomes de função que correspondam à expressão regular expressão. Combinado com a opção -t, permite selecionar também o tipo desejado de funções.

Exemplos:

`info functions test`	Lista funções cujos nomes contém com "test"
`info functions –t void .`	Lista todas as funções do tipo "void"
`info functions ^fput.$`	Lista funções com a expressão regular (fputs, fputc, etc)

Variáveis e conteúdo da memória

Comandos "print" e "display"

- "*print* <expressão>" ou "p <expressão>" – avalia e mostra o valor da expressão, a qual pode incluir endereços, constantes e o nome de variáveis que sejam visíveis no escopo corrente;
- "*display* <expressão>" ou "d <expressão>" – o mesmo que "*print*", porém **memoriza** a expressão, recalcula e mostra o seu valor atualizado a cada passo da execução na linha de comando do depurador;
- Os comandos "*print*" e "*display*" podem especificar o formato no qual a expressão será exibida, utilizando a notação "*print*/<formato> <expressão>". Os valores permitidos para <formato> são:
  - 'a' = ponteiro, 'c' = caractere, 'd' = inteiro com sinal, 'u' = inteiro sem sinal;
  - 'o' = inteiro em octal, 't' = inteiro em binário, 'x' = inteiro em hexadecimal;
  - 'f' = ponto flutuante;
  - 's' = *string*.

Exemplos:

`print i`	Mostra o valor atual da variável "i"
`print i/x`	Mostra o valor atual da variável "i" em hexadecimal
`display i`	Mostra o valor da variável "i" a cada iteração do depurador
`p x+y`	Avalia e mostra a expressão "x+y"

- A lista de expressões incluídas pelo comando "*display*" pode ser visualizada e alterada a partir do comando "*info display*" (ou "i *display*"), de forma semelhante ao apresentado anteriormente para os *breakpoints*.
  - "*disable display* <índice>" – interrompe a exibição da expressão identificada por <índice>;
  - "*enable display* <índice>" – ativa novamente a exibição, anteriormente desabilitada por "*disable display ...*";
  - "*undisplay* <índice>" – remove permanentemente a expressão identificada por <índice> da lista de expressões.

Watchpoints

- "*watch* <dado>" ou "w <dado>" cria um ponto de observação (*watchpoint*) relacionado a <dado>, que pode ser um símbolo do programa ou um endereço de memória. Sempre que o **valor** de <dado> for modificado pelo programa em depuração, o processo será interrompido, de forma equivalente ao que acontece um com um *breakpoint*;
- Os pontos de observação ativos podem ser visualizados e alterados a partir do comando "*info watch*" (ou "i *watch*"). Os *watchpoints* também aparecem na lista de *breakpoints* (com "info break").
  - "*disable* <índices>" – desabilita o(s) *watchpoint*(s)/*breakpoint*(s) cujo(s) índice(s) é(são) especificado(s) no comando. O(s) *watchpoint*(s)/*breakpoint*(s) poderá(ão) ser reabilitado(s) no futuro;
  - "*enable* <índices>" – habilita o(s) *watchpoint*(s)/*breakpoint*(s) cujo(s) índice(s) é(são) especificado(s) no comando;
  - "*delete* <índices>" – remove permanentemente o(s) *watchpoint*(s)/*breakpoint*(s).

Caso o valor de <índices> seja omitido nos comandos "*disable*", "*enable*" ou "*delete*", a operação afetará **todos** os *watchpoints*/*breakpoints* existentes.

Exemplos:

`watch res`	Interrompe o processo se a variável res for alterada
`info watch`	Lista de watchpoints
`disable 1`	Desabilita o primeiro *breakpoint*/*watchpoint*

Obtendo informação sobre símbolos e variáveis

- Use o comando "*info scope* <local>" ou "i *scope* <local>" para saber quais são os símbolos visíveis em um determinado escopo: <local> pode ser o nome de uma função ou um endereço de uma instrução;
- O comando "*whatis* <símbolo>" permite identificar o tipo que foi declarado para um símbolo (inteiro, ponteiro, ponto flutuante, etc.);

Visualizar o conteúdo da memória

- O comando "x/<quantidade><formato><tamanho> <endereço>" permite visualizar o conteúdo de uma ou várias posições de memória, em diversos formatos diferentes.
  - <quantidade> especifica o número de **registros** na memória a ser exibidos pelo comando;
  - <formato> é um caractere que segue <quantidade>, especificando o tipo ou o formato desejado de exibição de cada registro:
    - 'd' = decimal, 'u' = decimal sem sinal, 'x' = hexadecimal, 'o' = octal, 't' = binário;
    - 'a' = endereço, 'i' = instrução (*disassembly*)
    - 'c' = caractere, 's' = *string*, 'f' = ponto flutuante.
  - <tamanho> é um caractere (opcional) que segue <formato> e especifica a quantidade de endereços consecutivos que forma um único registro:
    - 'b' = *byte* – um registro por endereço;
    - 'h' = *half-word* – um registro a cada dois endereços (16 bits);
    - 'w' = *word* – um registro a cada quatro endereços (32 bits);
    - 'g' = *giant* – um registro a cada oito endereços (64 bits).

Exemplos:

`x/16xb 0x7fffffff0000`	Mostra 16 *bytes* em hexadecimal a partir do endereço
`x/16xb &vetor`	Mostra 16 *bytes* em hexadecimal da memória alocada para a variável "vetor"
`x/1fw &a`	Mostra o valor de "a" como ponto flutuante de 32 bits (*float*)
`x/1fg &a`	Mostra o valor de "a" como ponto flutuante de 62 bits (*double*)
`x/20i &calcula`	Mostra as primeiras 20 instruções da função "calcula"
`x/20c str`	Mostra os primeiros 20 caracteres do *string* "str"
`x/s str`	Mostra o *string* "str" completo (terminado em zero)

- O comando "x" – sem a especificação do endereço – repete o comando "x" anterior, com o mesmo formato, a partir do **último** endereço mostrado.

Controle da interface do usuário

Layout

O programa gdb pode exibir várias informações simultaneamente, dependendo do suporte do terminal utilizado pelo usuário. Caso o terminal permita, informações do depurador podem ser exibidas em diferentes "janelas" no mesmo terminal.

- O comando "*layout*" permite definir o modelo de exibição do estado do depurador ao usuário:
  - "*layout src*" – mostra o código-fonte juntamente com a linha de comando do depurador;
  - "*layout asm*" – mostra o programa executável em assembler juntamente com a linha de comando do depurador;
  - "*layout regs*" – mostra o contexto do processador (registradores).