**Лекция 3**

**Интересные приемы, которые можно делать с gdb**

**Watchpoints** - это функция в дебагере, которая позволяет отслеживать значения в программе.

Можно поставить watchpoint на изменение состояния объекта, и каждый раз, когда он меняет своё состояние, процесс выполнения программы останавливается. Это позволяет посмотреть значение и места, откуда и кем было изменено значение.

В лекции применение watchpoint показывалось на примере MergeSort.

Чтобы применить эту функцию, нужно:

1) поставить breakpoint (b) на нужной строке

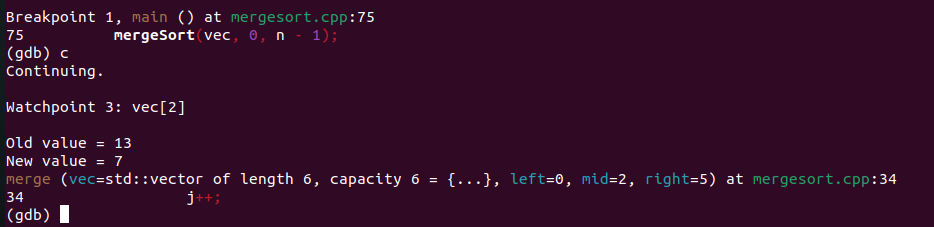
2) run

3) *watch arr[313]* - set a watchpoint for expression expr

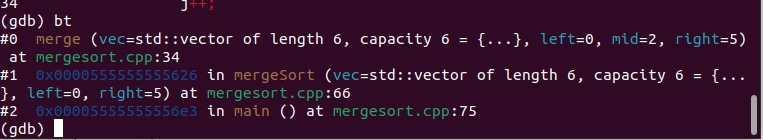
4) c

Таким образом наблюдаем за 313 ячейкой массива.

Примерно так это выглядит:



Можно написать *bt* и увидеть стек рекурсии:

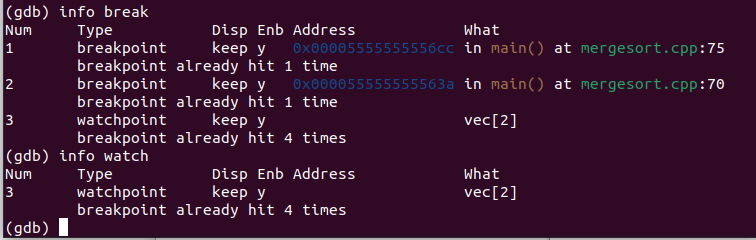


А команда *list* покажет строчки, вокруг которых мы на данный момент находимся.

Можно отобразить список всех активных breakpoints и watchpoints, и удалить их.

*Info break* и *info watch —* отображение

*Delete —* удаление breakpoint



**Сore dump**

Segmentation fault (core dumped) — операционная система не просто уронила программу, а записала состояние памяти (в момент падения программы) в специальный файл, который можно найти. Этот файл можно использовать для отладки программы и идентификации причины ошибки.

Этот файл хранится в такой директории: */var/lib/apport/coredump*

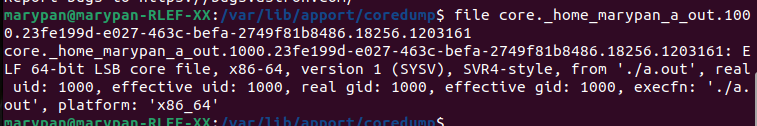
marypan@marypan-RLEF-XX:/var/lib/apport/coredump$ ls

*core.\_home\_marypan\_a\_out.1000.23fe199d-e027-463c-befa-2749f81b8486.18256.1203161*

Есть команда *ulimit* — она позволяет узнать, какой может быть максимальный разрешенный размер создаваемых системой core файлов.

*ulimit -c unlimited* — поменять размер на неограниченный

Утилита **file —** читает файл и определяет его тип.

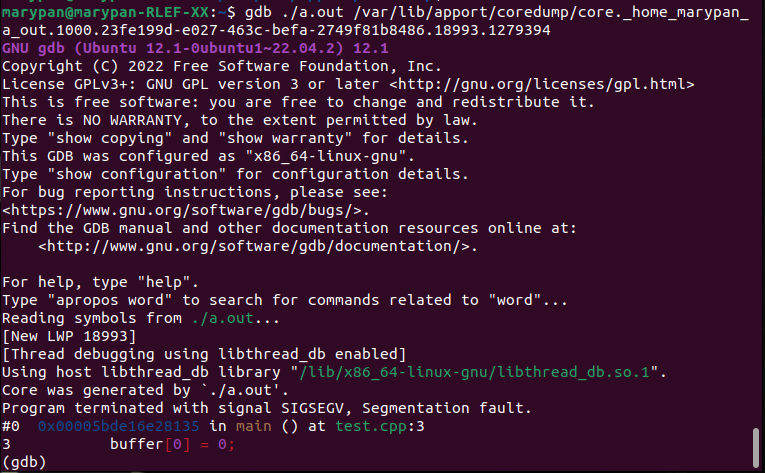


Core dump файл можно также читать с помощью **readelf**. Так мы обнаружим новый тип **CORE.**

Сделаем дебажную сборку нашего кода. Теперь можем запустить такое:

*gdb ./a.out /var/lib/apport/coredump/core.\_home\_marypan\_a\_out.1000.23fe199d-e027-463c-befa-2749f81b8486.18993.1279394*

Таким образом, имея DWARF файл с дебажной информацией и имея core dump файл, gdb способен восстановить информацию о том, в какой момент упала программа и что в этот момент происходило.

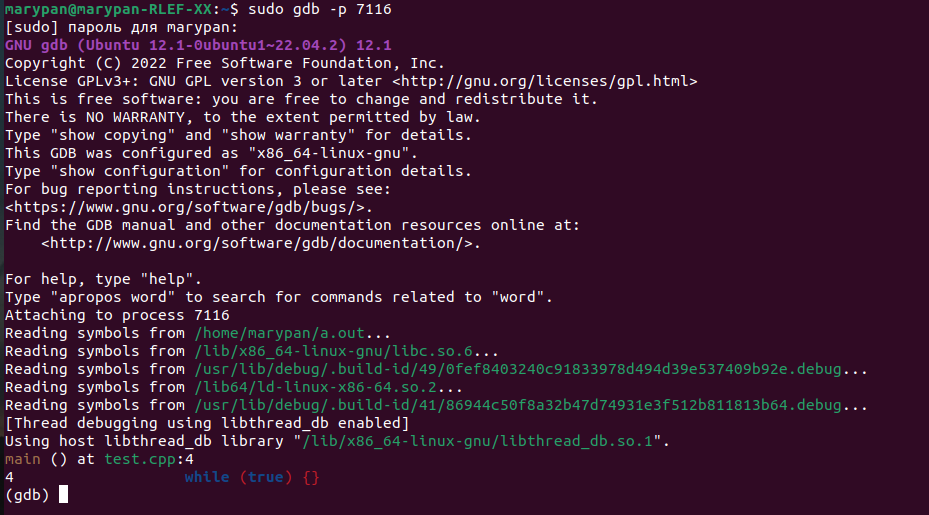


**Еще одно полезное умение.** Заставим программу зависнуть в какой-то момент (while true…).

Если знаем программу, которая зависла, и она собрана с дебажной информацией, с помощью gdb можно подцепиться к этой работающей программе и понять, в какой момент она зависла.

Для начала нужно понять, под каким id работает в системе наша программа. В этом может помочь утилита **htop**. Она позволяет видеть список работающих в системе процессов.

*sudo gdb -p num\_of\_process* — так можно посмотреть, что с процессом случилось не так. Gdb как бы просит у операционной системы память, которая была выделена этому процессу, забирает ее себе и разбирается с ней.



Теперь попробуем зайти через дебаггер в код std::cin.

Для этого нам нужна дебажная версия библиотеки. Это можно установить так:

*sudo apt install libstdc++6-11-dbg*

Теперь эта либа находится в папке */usr/lib/x86\_64-linux-gnu/debug/*

Теперь компилим код с этой либой:

*g++ -g -D\_GLIBCXX\_DEBUG test.cpp -Wl,-rpath,/usr/lib/x86\_64-linux-gnu/debug/*



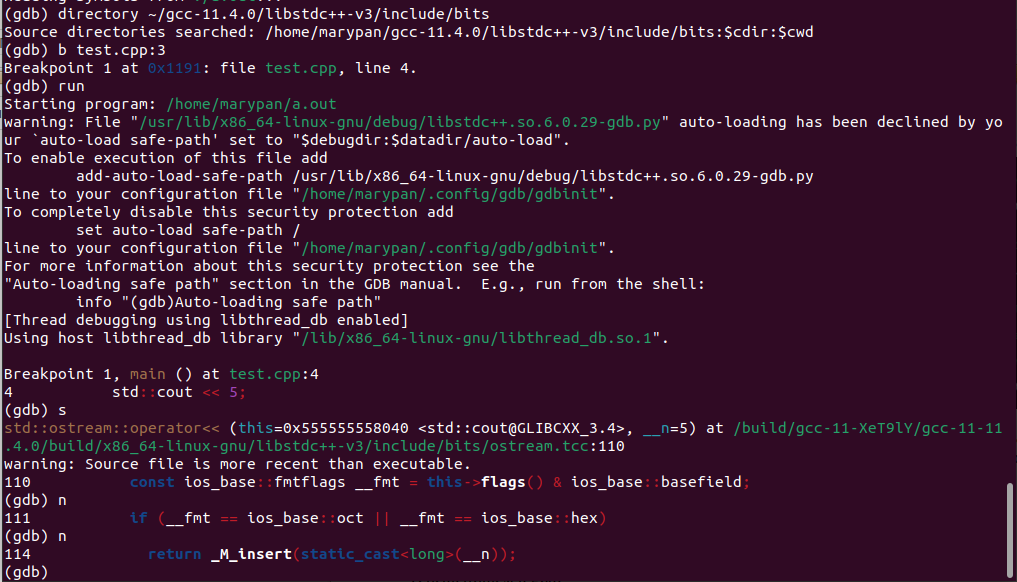
Можно указать gdb, где искать файл с исходным кодом, чтобы он мог по нему шагать. Это можно сделать с помощью команды *directory \*путь к исходному коду\*.* Этот код нужно предварительно скачать.

Исходники можно скачать отсюда: <https://mirror.linux-ia64.org/gnu/gcc/releases/gcc-11.4.0/>

В папке *~/gcc-11.4.0/libstdc++-v3/include/bits* как раз находятся те файлы, которые инклюдятся.

Теперь пишем в gdb:

*directory ~/gcc-11.4.0/libstdc++-v3/include/bits*

Так он будет искать исходники в том числе в этой папке.

Также gdb умеет дизассемблировать код. Команда *disassemble*

**Как запускать линковщик вручную**

(у меня так и не получилось это сделать………………… а эта часть из конспекта прошлого года)

Это делается с помощью программы **ld** — линковщик объектных файлов.

После запуска a.out запускается динамический линковщик. Затем запускается некая функция \_start, после неё - main. Для динамического линковщика в решении нужен параметр -I, а для того, чтобы линковщик не считал main начальной функцией \_start - параметр -e.

*ld -e main test.o /usr/lib/x86\_64-linux-gnu/libstdc++.so.6 /usr/lib/x86\_64-linux-gnu/libm.so.6 /usr/lib/x86\_64-linux-gnu/libgcc\_s.so.1 /usr/lib/x86\_64-linux-gnu/libc.so.6 /usr/lib64/ld-linux-x86-64.so.2 /usr/lib/gcc/x86\_64-linux-gnu/11/crtend.o -I /lib64/ld-linux-x86-64.so.2*

./a.out

> 5

seg. fault

Падение, потому что после исполнения инструкций программа пытается выйти из main, но у неё это не получается. На выходе из \_start такого нет, потому что у него прописана специальная инструкция exit, говорящая ОС завершить работу программы.