Q. Difference between h/w and s/w break point

Hardware breakpoints are actually **comparators**, comparing the current PC with the address in the comparator (when enabled). Hardware breakpoints are the best solution when setting breakpoints. Typically set via the debug probe (using JTAG, SWD, ...). The downside of hardware breakpoints: They are **limited**. CPUs have only a limited number of hardware breakpoints (comparators). The number of available hardware breakpoints depends on the CPU. ARM 7/9 cores have 2, modern ARM devices (Cortex-M 0,3,4) between 2 and 6, x86 usually 4.

Software breakpoints are in fact set by replacing the instruction to be breakpointed with **a breakpoint instruction**. The breakpoint instruction is present in most CPUs, and usually as short as the shortest instruction, so only one byte on x86 (0xcc, INT 3). On Cortex-M CPUs, instructions are 2 or 4 bytes, so the breakpoint instruction is a 2 byte instruction.

Software breakpoints can easily be set if the program is located in RAM (such as on a PC). **A lot of embedded systems have the program located in flash memory. Here it is not so easy to exchange the instruction, as the flash needs to be reprogrammed, so hardware breakpoints are used primarily**. Most debug probes support only hardware breakpoints if the program is located in **flash memory**. However, some (such as SEGGER's J-Link) allow reprogramming the flash memory with breakpoint instruction and aso allow an unlimited number of (software) breakpoints even when debugging a program located in flash.

Q. how would you set a breakpoint within your code without a debugger eg. Gdb

-Additional Log with \_\_LINE\_\_ or \_\_FILE\_\_

Q. Find and fix the bugs in the following function that is supposed to remove the head element from a singly linked list.

void RemoveHead (node \* head) {

free(head);

head = head - > next;

}

A.

Void RemoveHead(node \*head) {

Node \*temp = head;

head = head->next;

delete temp;

}

Q. Difference between crash and exception in c++ -

Exceptions are defined situations which c++ can manage and handle.

Some undefined behaviors like null pointer dereferencing is not handled through exception, and so causes the program to crash.

When a program encounters a situation that was not expected, there are three different behaviors the program could exhibit:

-Sometimes programs crash. A crash is when a program quits unexpectedly.

-Sometimes programs return an exception. An exception is an unexpected error that does not result in the program quitting.

-Sometimes programs hang. A hang is when a program remains open and fails to respond to input by the user.

**Difference between macros and inline functions.**

-Inline replaces a call to a function with the body of the function, however, inline is just a request to the compiler that could be ignored (you could still pass some flags to the compiler to force inline or use \*always\_inline\* attribute with gcc).

-compilers can also inline expand some **recursive** functions; recursive macros are typically illegal.

-A macro on the other hand, is expanded by the **preprocessor** before compilation, so it's just like **text substitution**, also macros are **not type checked**, inline functions are. There's a comparison in the wiki.

-For the sake of completeness, you could still have some kind of type safety with macros, using gcc's \_\_typeof\_\_ for example, the following generate almost identical code and both cause warnings if used with the wrong types

**Early Binding vs Late Binding (or static library, dynamic library)**

Early Binding: Static Binding

When perform Early Binding, an object is assigned to a variable declared to be of a **specific object type**. Early binding objects are basically a **strong type objects** or **static type objects**. While Early Binding, methods, functions and properties which are detected and checked during **compile time** and perform other **optimizations** before an application executes. The biggest advantage of using early binding is for **performance and ease of development.**

Ex:

System.IO.FileStream FS ;

FS = new System.IO.FileStream("C:\\temp.txt", System.IO.FileMode.Open);

Above code, create a variable FS to hold a new object and then assign a new object to the variable. Here **type is known before the variable is exercised during run-time**, usually through declarative means. The FileStream is a specific object type, the instance assigned to FS is early bound. Early Binding is also called static binding or compile time binding.

While performing Early Binding the compiler can ensure at compile time that the function will exist and be callable at runtime. Moreover the compiler guarantees that the function takes the exact number of arguments and that they are of the right type and can checks that the return value is of the correct type.

Late binding: Dynamic Binding

By contrast, in Late binding functions, methods, variables and properties are detected and checked only **at the run-time**. It implies that the **compiler does not know what kind of object or actual type of an object or which methods or properties an object contains until run time**. The biggest advantages of Late binding is that **the Objects of this type can hold references to any object, but lack many of the advantages of early-bound objects**

Ex:

object FS = null;

FS = CreateObject("Scripting.FileSystemObject");

Above code does not require a reference to be set beforehand, the instance creation and type determination will just happen at runtime. It is important to note that the Late binding can only be used to access type members that are declared as Public. Accessing members declared as Friend or Protected Friend resulted in a run-time error.

While perform late binding there is a possibility of the target function may not exist. Aslo the target function may not accept the arguments passed to it, and may have a return value of the wrong type.

what are types of memory issues one faces:

Latency:

in modern processors a memory access can go through multiple levels of caches and main memory. Depending on where the accessed data is, the latency of the access can go from a few cycles to hundred or thousands of them. When possible, a programmer should be careful to access memory with high **locality**. For example, If an image is stored in contiguous memory, with consecutive pixels in a raw occupying consecutive words in memory, a filter scanning the image should have the inner loop on columns and the outer on rows.

Coherency:

Not all multi-core architectures guarantee cache coherency across cores. In such cases, the developer has to make sure data accessed by more than one core is properly made coherent in caches.

Trashing:

Excessive use of memory/page trashing: An application using an excessive amount of memory can either run out or, in systems with virtual memory, cause trashing of loaded pages, causing excessive page swapping.

Q. What is wrong with this program :

main()

{

char \*p,\*q;

p=(char\*)malloc(25);

q=(char\*)malloc(25);

strcpy(p,"amazon" );

strcpy(q,"hyd");

strcat(p,q);

printf("%s",p);

}

--------------

As such, nothing is wrong with the program, unless you want to implement coding best-practices:

1. You should check return of malloc - whether it has returned NULL or not.

2. Use strncpy, and strncat; Although this is not be an absolute requirement as long as malloc size and strings are hard-coded.

3. main should return int, and should also return an exit value.

4. p and q should be initialized to NULL.

Q. How would you debug kernel code.

1. printk

2. kernel code can be debugged various methods some of them

1) UML ( User mode linux) - Best method but lacks Device Driver support

2) **KGDB** ( Kernel gnu debugger.. via serial,network, also we can use vmware etc)... Best needs two systems

3) **kdb** ( Builtin kernel debugger)... Source code debugger is not possible

4) kdump tools ( for Kerenel core dump etc) similar like UML

3. Just for embedded system.

If there is crash dump, i prefer use "Crash" utility from RedHAT.

If there is runtime debug, i prefer use "**Trace32**".

What happens when you get a segmentation fault? how do you get it and how is it implemented?

1. Most like the segmentation fault is caused by **memory illegal access**. So probably a good place to check is the place where you dynamically allocated/used the memory. Also if it is possible, try to use assert in the places where it might go wrong.

2. when u try 2 access restricted memory area

int \*p;

p=null;

printf("%d",p);

3.

segv is different from bus error (only two error messages on core dump in unix)

**segv means address is valid in the system arch but access violated**.

**bus error is when access is not even valid in the system**.

segv is implemented by access protection checks in OS

bus error - hardware timeout

Q. How to write on a specific location / address on a device driver or register ?

Depends on Device:

1, Processor is **memory-mapped IO**. Map the device's register to conventional memory.

\* Used appropriate sized and signed type for register: e.g. one byte can be declared as char; two byte can be declared as unsinged short.

\*int32\_t, Uint32\_t declare in <stdint.h>

\*Some compilers provide language extensions that will let you position an object at a specified memory address. For example, using the TASKING C166/ST10 C Cross-Compiler's \_at attribute you can write a global declaration such as:

unsigned short count \_at(0xFF08);

\* Use pointer:dual\_timers \*const timers

= (dual\_timers \*)0x03FF6000;

2, Processor is **port-mapped IO**. Map the device's register to seperate memory space.

In this scenario, needing exta library extention, or even assembly language.

e.g. IN and OUT for Intel 80X86

----------------

it is very easy to write into any memory location

Uint32\_t \*addr;

addr = (Uint32\_t)0x12345678; //0x12345678 - is any address location

\*addr = 0x2341; //0x2341 is the value u want to write

or simply:

\*((int\*)0x12345678) = 0x2341

If you have a very large code base, how do you detect **memory leaks** in it with minimum changes made to the actual code?

You do not need to make any changes at all. The interviewer is expecting some smart answer, so be smart and tell this:

1. **No code change**

2. Use standard **memory profiling tools** to chek for memory leaks

and the interviewer will be hapy to know that answer.

-----

I forgot to mention actually he said there are no memory profiling tools :)

-----

In this case, we can write **our own version of new and delete operators and keep information about the memory allocated and deleted.** at the end of program, we can check if any of the memory is still in hold and not deleted.

since we do have profiling tools, this is a waste question.

-----

Make use of Kernel Hooks !

By using mtrace() and muntrace() functions. for this you need to set the MALLOC\_TRACE environment variable to log file path.

**-----> valgrind**

Q. What happens when the following piece of code is executed?

Char \*ptr;

While(1) ptr = malloc(1024\*1024);

-------------

Malloc would simply start failing when it can no longer allocate more memory. The program will continue to run...

stack overflow cant occur malloc uses heap :)

-----main (){

int \*p;

while(1){

p=(int \*)malloc(1024\*1024);

if(!p)

printf("no more\n");}

}

output:

aftr some time

no more

no more

no more

.....

You have a piece of code that randomly crashes. How will correct the error

Check the log

If you find a random crash, that can be due to some issues with locking (synchronization) or accessing an invalid/freed memory (can again be due to synchronization).

* Race condition in multi-threads
* Uninitialized values
* Optimization

Say the program you are running has stack overflow. You want to know when and where this happens, but you don't want to use debuggers, because they slow. How would you do this?

A. **write a pattern to end of stack.**

uint32\_t \*StackPtr = (uint32\_t \*) 0x7FFFFFFF; //0x7FFFFFFFis end of stack

\*StackPtr = 0xDEADBEEF;

// A memory dump or a simple check in your code like will suffice

if (\*StackPtr != 0xDEADBEEF)

{ //Stack overflow }

Align the stack memory at the end of a read only memory, this would cause an automatic err fatal