telephonic .. MOSCHIP

1. char moschip= 'a';

char prajna = "a";

how much memory is allocacted for both of these?

how does compiler get to know that in char prajna ="a", it should allocate 2 bytes i.e, a followed by '\o'

2. main()

{

for(i=0;i<10;i++)

moschip();

}

moschip()

{

int k=10;

static int z=20;

printf("vlue of k and z");

}

2) why here static is used in the function?

and what are the values printed?

Static functions in C

In C, functions are global by default. The "*static*" keyword before a function name makes it static. For example, below function *fun()* is static.

```
static int fun(void)

{

  printf("I am a static function ");

}
```

Unlike global functions in C, access to static functions is restricted to the file where they are declared. Therefore, when we want to restrict access to functions, we make them static. Another reason for making functions static can be reuse of the same function name in other files.

For example, if we store following program in one file *file1.c*

```
/* Inside file1.c */

static void fun1(void)

{

  puts("fun1 called");

}
```

And store following program in another file *file2.c*

```
/* Iinside file2.c  */

int main(void)

{

  fun1();

  getchar();

  return 0;

}
```

Now, if we compile the above code with command "*gcc file2.c file1.c*", we get the error *"undefined reference to `fun1'"* . This is because *fun1()* is declared *static* in *file1.c* and cannot be used in *file2.c*.

Please write comments if you find anything incorrect in the above article, or want to share more information about static functions in C.

3.differnce between normal pointer and function pointer?

where did u use function pointers?

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

```
#include <stdio.h>

// A normal function with an int parameter

// and void return type

void fun(int a)

{

printf("Value of a is %d\n", a);

}

int main()

{

// fun_ptr is a pointer to function fun()

void (*fun_ptr)(int) = &fun;

/* The above line is equivalent of following two

void (*fun_ptr)(int);

fun_ptr = &fun;

*/

// Invoking fun() using fun_ptr

(*fun_ptr)(10);

return 0;

}
```

Output:

Value of a is 10

Why do we need an extra bracket around function pointers like fun_ptr in above example?
If we remove bracket, then the expression “void (*fun_ptr)(int)” becomes “void *fun_ptr(int)” which is declaration of a function that returns void pointer. See following post for details.
How to declare a pointer to a function?

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
3) A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing *, the program still works.

4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.
5) Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks

Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

4. what is the default return value of malloc()

The malloc() function allocates *size* bytes and returns a pointer to

the allocated memory. *The memory is not initialized*. If *size* is 0,

then **malloc**() returns either NULL, or a unique pointer value that can

later be successfully passed to **free**().

5. difference between structure and union?

struct

{

int a;

char b;

}

generally int allocates 4 bytes. but in our program if only 2bytes are used .then how do u tell to the compiler that only 2 bytes are used?

The difference between structure and union is,
1. ***The amount of memory required to store a structure variable is the sum of the size of all the members.***
On the other hand, in case of unions, the amount of memory required is always equal to that required by its largest member.

2. In case of structure, each member have their own memory space but In union, one block is used by all the member of the union.

Detailed Example:

```
1 struct stu
2 {
3     char c;
4     int l;
5     float p;
6 };
```

```
1 union emp
2 {
3     char c;
4     int l;
5     float p;
6 };
```

In the above example size of the structure stu is 7 and size of union emp is 4.

6. What is an inline function?

7. can we declare macro with an inline?

Many C and C++ programming beginners tend to confuse between the concept of macros and Inline functions.

Often the difference between the two is also asked in C interviews.

In this tutorial we intend to cover the basics of these two concepts along with working code samples.

1. The Concept of C Macros

Macros are generally used to define constant values that are being used repeatedly in program. Macros can even accept arguments and such macros are known as function-like macros. It can be useful if tokens are concatenated into code to simplify some complex declarations. **Macros provide text replacement functionality at pre-processing time.**

Here is an example of a simple macro :

```
 #define MAX_SIZE 10
```

The above macro (MAX_SIZE) has a value of 10.

Now let's see an example through which we will confirm that macros are replaced by their values at pre-processing time. Here is a C program :

```
#include<stdio.h>


#define MAX_SIZE 10


int main(void)

{

   int size = 0;
```

```
  size = size + MAX_SIZE;


  printf("\n The value of size is [%d]\n",size);


  return 0;
}
```

Now lets compile it with the flag *-save-temps* so that pre-processing output (a file with extension .i) is produced along with final executable :

```
$ gcc -Wall -save-temps macro.c -o macro
```

The command above will produce all the intermediate files in the gcc compilation process. One of these files will be macro.i. This is the file of our interest. If you open this file and get to the bottom of this file :

```
...
...
...
int main(void)
{
  int size = 0;
  size = size + 10;


  printf("\n The value of size is [%d]\n",size);


  return 0;
}
```

So you see that the macro MAX_SIZE was replaced with it's value (10) in preprocessing stage of the compilation process.

Macros are handled by the pre-compiler, and are thus guaranteed to be inlined. Macros are used for short operations and it avoids function call overhead. It can be used if any short operation is being done in program repeatedly. Function-

like macros are very beneficial when the same block of code needs to be executed multiple times.

Here are some examples that define macros for swapping numbers, square of numbers, logging function, etc.

```
#define SWAP(a,b)({a ^= b; b ^= a; a ^= b;})

#define SQUARE(x) (x*x)

#define TRACE_LOG(msg) write_log(TRACE_LEVEL, msg)
```

Now, we will understand the below program which uses macro to define logging function. It allows variable arguments list and displays arguments on standard output as per format specified.

```
#include <stdio.h>

#define TRACE_LOG(fmt, args...) fprintf(stdout, fmt, ##args);


int main() {

int i=1;

TRACE_LOG("%s", "Sample macro\n");

TRACE_LOG("%d %s", i, "Sample macro\n");

return 0;

}
```

Here is the output:

```
$ ./macro2

Sample macro

1 Sample macro
```

Here, TRACE_LOG is the macro defined. First, character string is logged by TRACE_LOG macro, then multiple arguments of different types are also logged as shown in second call of TRACE_LOG macro. Variable arguments are supported with the use of “…” in input argument of macro and ##args in input argument of macro value.

2. C Conditional Macros

Conditional macros are very useful to apply conditions. Code snippets are guarded with a condition checking if a certain macro is defined or not. They are very helpful in large project having code segregated as per releases of project. If some part of code needs to be executed for release 1 of project and some other part of code needs to be executed for release 2, then it can be easily achieved through conditional macros.

Here is the syntax :

```
#ifdef PRJ_REL_01

..

.. code of REL 01 ..

..

#else

..

.. code of REL 02 ..

..

#endif
```

To comment multiples lines of code, macro is used commonly in way given below :

```
#if 0

..

.. code to be commented ..

..

#endif
```

Here, we will understand above features of macro through working program that is given below.

```
#include <stdio.h>


int main() {


#if 0
```

```
printf("commented code 1");

printf("commented code 2");

#endif


#define TEST1 1


#ifdef TEST1

printf("MACRO TEST1 is defined\n");

#endif


#ifdef TEST3

printf("MACRO TEST3 is defined\n");

#else

printf("MACRO TEST3 is NOT defined\n");

#endif


return 0;

}
```

Output:

```
$ ./macro

MACRO TEST1 is defined

MACRO TEST3 is NOT defined
```

Here, we can see that “commented code 1”, “commented code 2” are not printed because these lines of code are commented under #if 0 macro. And, TEST1 macro is defined so, string “MACRO TEST1 is defined” is printed and since macro TEST3 is not defined, so “MACRO TEST3 is defined” is not printed.

2. The Concept of C Inline Functions

Inline functions are those functions whose definition is small and can be substituted at the place where its function call is made. Basically they are inlined with its function call.

Even there is no guarantee that the function will actually be inlined. Compiler interprets the inline keyword as a mere hint or request to substitute the code of function into its function call. Usually people say that having an inline function increases performance by saving time of function call overhead (i.e. passing arguments variables, return address, return value, stack mantle and its dismantle, etc.) but whether an inline function serves your purpose in a positive or in a negative way depends purely on your code design and is largely debatable.

Compiler does inlining for performing optimizations. If compiler optimization has been disabled, then inline functions would not serve their purpose and their function call would not be replaced by their function definition.

To have GCC inline your function regardless of optimization level, declare the function with the “always_inline” attribute:

void func_test() __attribute__((always_inline));

Inline functions provides following advantages over macros.

- Since they are functions so type of arguments is checked by the compiler whether they are correct or not.
- There is no risk if called multiple times. But there is risk in macros which can be dangerous when the argument is an expression.
- They can include multiple lines of code without trailing backlashes.
- Inline functions have their own scope for variables and they can return a value.
- Debugging code is easy in case of Inline functions as compared to macros.

It is a common misconception that inlining always equals faster code. If there are many lines in inline function or there are more function calls, then inlining can cause wastage of space.

Now, we will understand how inline functions are defined. It is very simple. Only, we need to specify “inline” keyword in its definition. Once you specify “inline” keyword in its definition, it request compiler to do optimizations for this function to save time by avoiding function call overhead. Whenever calling to inline function is made, function call would be replaced by definition of inline function.

#include <stdio.h>

```
void inline test_inline_func1(int a, int b) {
    printf ("a=%d and b=%d\n", a, b);
}

int inline test_inline_func2(int x) {
    return x*x;
}

int main() {

    int tmp;

    test_inline_func1(2,4);
    tmp = test_inline_func2(5);

    printf("square val=%d\n", tmp);

    return 0;
}
```

Output:

```
$ ./inline
a=2 and b=4
square val=25
```

8. what is dangling pointer?

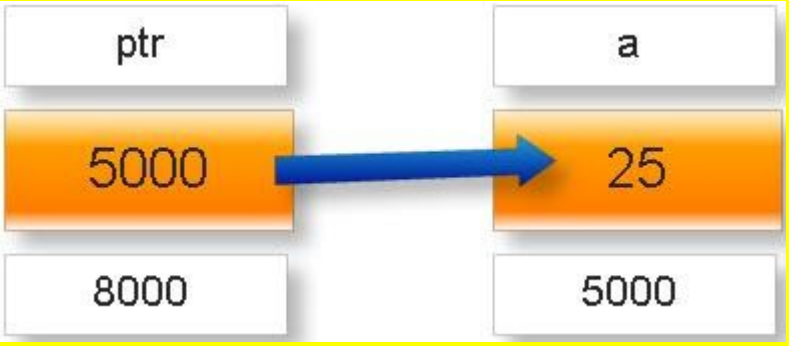
how do u avoid it?

1. Dangling pointer:

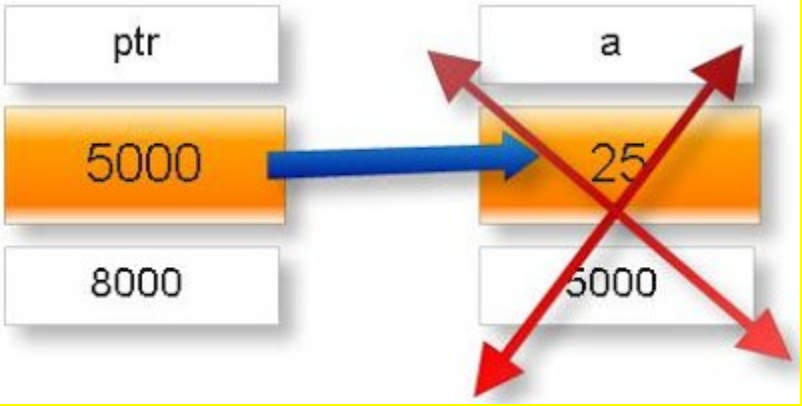
If any pointer is pointing the memory address of any variable but after some variable has deleted from that memory location while pointer is still pointing such memory location. Such pointer is known as dangling pointer and this problem is known as dangling pointer problem,

to avoid this by free after compte use,or use static as before that.

Initially:



Later:



For example:

(q)What will be output of following c program?

```
#include<stdio.h>
int *call();
void main(){

int *ptr;
ptr=call();

fflush(stdin);
printf("%d",*ptr);

}
int * call(){
```

```
int x=25;
++x;


return &x;
}
```

Output: Garbage value

Note: In some compiler you may get warning message **returning address of local variable or temporary**

Explanation: variable x is local variable. Its scope and lifetime is within the function call hence after returning address of x variable x became dead and pointer is still pointing ptr is still pointing to that location.

Solution of this problem: Make the variable x is as static variable.

In other word we can say a pointer whose pointing object has been deleted is called dangling pointer.

```
#include<stdio.h>
int *call();
void main(){


int *ptr;
```

```
ptr=call();


fflush(stdin);
printf("%d",*ptr);


}
int * call(){


static int x=25;
++x;


return &x;
}
```

Output: 26

9. what is wide pointer and far pointer?

Wild pointer:

A pointer in c which has not been initialized is known as wild pointer.

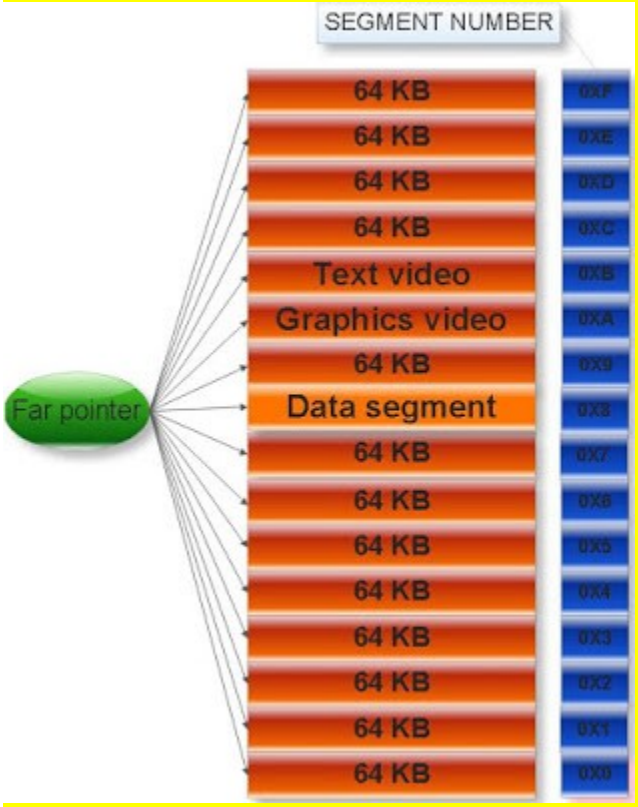
Example:

```
# include<stdio.h>
int main(){
int *ptr;
printf("%u\n",ptr);
printf("%d",*ptr);
return 0;
}
```

Output:
Any address
Garbage value

Far pointer

The pointer which can point or access whole the residence memory of RAM i.e. which can access all 16 segments is known as far pointer.



Far pointer:

(If you don’t know what is segment the click here)

Size of far pointer is 4 byte or 32 bit.

Examples:

(1) What will be output of following c program?

```
#include<stdio.h>
int main(){
int x=10;
int far *ptr;
ptr=&x;
printf("%d",sizeof ptr);

return 0;
}
```

Output: 4

10. main()

{

fork();

thread();

}

in this which has access to the resources of main()

11.how do u create a thread?

how the attr argument of pthread_create is used.

pthread_create - create a new thread

Synopsis

#include <pthread.h>

int pthread_create(pthread_t ***threadid(addres of id starting)*****, const pthread_attr_t** ***attr*****,**

void *(**functn name*** **) (void *), void** ****arg*****);**

Compile and link with *-pthread*.

Description

The **pthread_create**() function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*().

The new thread terminates in one of the following ways:

*

12.difference between semaphore and mutex?

Mutex vs Semaphore

What are the differences between Mutex vs Semaphore? When to use mutex and when to use semaphore?

Concrete understanding of Operating System concepts is required to design/develop smart applications. Our objective is to educate the reader on these concepts and learn from other expert geeks.

As per operating system terminology, mutex and semaphore are kernel resources that provide synchronization services (also called as synchronization primitives). Why do we need such synchronization primitives? Won't be only one sufficient? To answer these questions, we need to understand few keywords. Please read the posts on atomicity and critical section. We will illustrate with examples to understand these concepts well, rather than following usual OS textual description.

. Assume, we have a buffer of 4096 byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. Objective is, both the threads should not run at the same time.

Using Mutex:

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.

Using Semaphore:

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Misconception:

There is an ambiguity between *binary semaphore* and *mutex*. We might have come across that a mutex is binary semaphore. *But they are not*! The purpose of mutex and semaphore are different. May be, due to similarity in their implementation a mutex would be referred as binary semaphore.

Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

13. if there are two proces p1 and p2.they have to access memory which mechanism is best?(mutex or semaphor?)

and these processes are created at same time and have same priority. then if p1 has to access the memory how do u say that to compiler?

Check in ppt

14.if there are three thread p1,p2,p3, then how do u join them?

what does this joining mean?

Include <pthread.h>

int pthread_join(pthread_t thread id, void **value_ptr);

–The pthread_join() function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

–On return from a successful pthread_join() call with a non-NULL value_ptr argument, the value passed to pthread_exit() by the terminating thread is made available in the location referenced by value_ptr.

–If successful, the pthread_join() function returns zero.

15.we use gfp_kernel... for what is that used?

GFP_KERNEL This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep.

I've googled around and found most people advocating the use of `kmalloc`, as you're guaranteed to get contiguous physical blocks of memory. However, it also seems as though `kmalloc` can fail if a contiguous **physical** block that you want can't be found.
What are the advantages of having a contiguous block of memory? Specifically, why would I need to have a contiguous **physical** block of memory in a *system call*? Is there any reason I couldn't just use `vmalloc`?
Finally, if I were to allocate memory during the handling of a system call, should I specify `GFP_ATOMIC`? Is a system call executed in an atomic context?

GFP_ATOMIC
The allocation is high-priority and does not sleep. This is the flag to use in interrupt handlers, bottom halves and other situations where you cannot sleep.

16. what about ioperm?

NAME

ioperm - set port input/output permissions

SYNOPSIS

#include <sys/io.h> /* for glibc */

int ioperm(unsigned long from, unsigned long num, int turn_on);

DESCRIPTION

ioperm() sets the port access permission bits for the calling thread for num bits starting from port address from. If *turn_on* is nonzero, then permission for the specified bits is enabled; otherwise it is disabled. If *turn_on* is nonzero, the calling thread must be privileged (**CAP_SYS_RAWIO**).

Before Linux 2.6.8, only the first 0x3ff I/O ports could be specified in this manner. For more ports, the iopl(2) system call had to be used (with a *level* argument of 3). Since Linux 2.6.8, 65,536 I/O ports can be specified.

Permissions are not inherited by the child created by fork(2); following a fork(2) the child must turn on those permissions that it needs. Permissions are preserved across execve(2); this is useful for giving port access permissions to unprivileged programs.

This call is mostly for the i386 architecture. On many other architectures it does not exist or will always return an error.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

17.how do u find the number of nodes in a circular single linked list?

Answers:

```
struct node

{

int data;

struct node *next;

};

int count(struct node *pp)

{ struct node *start;

int count=0;

start=pp->next;

while(start->next!=pp)

{ start=start->next;

count++; }

return count;

}
```

18.differnce between kernel image,zimage and uimage?

A uImage file is a kernel with a modified header for u-boot. A tool called mkimage is used to convert a zImage (regular kernel compressed image) to a uImage file. And No, zImage files, as they are, are not compatible with U-Boot. You must convert them

make zImage - Creates a gzip'd kernel image that must be installed manually.
make bzImage - Creates a bzip2'd kernel image that must be installed manual

bz image is compressed image when compared to zimage

Image: the generic Linux kernel binary image file.

zImage: a compressed version of the Linux kernel image that is self-extracting.

uImage: an image file that has a U-Boot wrapper (installed by the **mkimage** utility) that includes the OS type and loader information.
A very common practice (e.g. the typical Linux kernel Makefile) is to use a zImage file. Since a zImage file is self-extracting (i.e. needs no external decompressors), the -

wrapper would indicate that this kernel is "not compressed" even though it actually is.

Note that the author/maintainer of U-Boot considers the (widespread) use of using a zImage inside a uImage questionable:

Actually it's pretty stupid to use a zImage inside an uImage. It is much better to use normal (uncompressed) kernel image, compress it using just gzip, and use this as poayload for mkimage. This way U-Boot does the uncompresiong instead of including yet another uncompressor with each kernel image.

19. examples of linux libraries

what is the extension of dynamic libraries and static libraries?

.so and .a these extensions belong to which libraries? (dynamic or static)?

Linux Library Types:

There are two Linux C/C++ library types which can be created:

1. **Static libraries (.a):** Library of object code which is linked with, and becomes part of the application.
2. **Dynamically linked shared object libraries (.so):** There is only one form of this library but it can be used in two ways.

1. Dynamically linked at run time but statically aware. The libraries must be available during compile/link phase. The shared objects are not included into the executable component but are tied to the execution.
2. Dynamically loaded/unloaded and linked during execution (i.e. browser plug-in) using the dynamic linking loader system functions.

Library naming conventions:

Libraries are typically names with the prefix "lib". This is true for all the C standard libraries. When linking, the command line reference to the library will not contain the library prefix or suffix.

Thus the following link command: **`gcc src-file.c -lm -lpthread`**
The libraries referenced in this example for inclusion during linking are the math library and the thread library. They are found in `/usr/lib/lib`**`m`**`.a` and `/usr/lib/lib`**`pthread`**`.a`.

Note: The GNU compiler now has the command line option "-pthread" while older versions of the compiler specify the pthread library explicitly with "-lpthread". Thus now you are more likely to see `gcc src-file.c -lm -pthread`

Static Libraries: (.a)

How to generate a library (object code archive file):

- Compile: `cc -Wall -c ctest1.c ctest2.c`
  Compiler options:
  - -Wall: include warnings. See man page for warnings specified.
- Create library "libctest.a": `ar -cvq libctest.a ctest1.o ctest2.o`
- List files in library: `ar -t libctest.a`
- Linking with the library:
  - `cc -o` *`executable-name`* `prog.c libctest.a`
  - `cc -o` *`executable-name`* `prog.c -L/path/to/library-directory -lctest`
- Example files:
  - ctest1.c

```
void ctest1(int *i)

{

  *i=5;
```

```
}
```

- ctest2.c

```
void ctest2(int *i)
{
  *i=100;
}
```

- prog.c

```
#include <stdio.h>
void ctest1(int *);
void ctest2(int *);
int main()
{
  int x;
  ctest1(&x);
  printf("Valx=%d\n",x);
  return 0;
}
```

Historical note: After creating the library it was once necessary to run the command: `ranlib ctest.a`. This created a symbol table within the archive. Ranlib is now embedded into the "ar" command.

Note for MS/Windows developers: The Linux/Unix ".a" library is conceptually the same as the Visual C++ static ".lib" libraries.

Dynamically Linked "Shared Object" Libraries: (.so)

How to generate a shared object: (Dynamically linked object library file.) Note that this is a two step process.

1. Create object code
2. Create library
3. Optional: create default version using a symbolic link.

Library creation example:

```
gcc -Wall -fPIC -c *.c

gcc -shared -Wl,-soname,libctest.so.1 -o libctest.so.1.0   *.o

mv libctest.so.1.0 /opt/lib

ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1

ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so
```

This creates the library `libctest.so.1.0` and symbolic links to it.

It is also valid to cascade the linkage:

```
ln -sf /opt/lib/libctest.so.1.0 /opt/lib/libctest.so.1

ln -sf /opt/lib/libctest.so.1   /opt/lib/libctest.so
```

If you look at the libraries in `/lib/` and `/usr/lib/` you will find both methodologies present. Linux developers are not consistent. What is important is that the symbolic links eventually point to an actual library.

Compiler options:

- `-Wall`: include warnings. See man page for warnings specified.
- `-fPIC`: Compiler directive to output position independent code, a characteristic required by shared libraries. Also see "-fpic".
- `-shared`: Produce a shared object which can then be linked with other objects to form an executable.
- `-Wl,`*`options`*: Pass options to linker.
  In this example the options to be passed on to the linker are: "`-soname libctest.so.1`". The name passed with the "-o" option is passed to `gcc`.
- Option `-o`: Output of operation. In this case the name of the shared object to be output will be "`libctest.so.1.0`"

Library Links:

- The link to `/opt/lib/libctest.so` allows the naming convention for the compile flag `-lctest` to work.
- The link to `/opt/lib/libctest.so.1` allows the run time binding to work. See dependency below.

Compile main program and link with shared object library:

Compiling for runtime linking with a dynamically linked `libctest.so.1.0`:

gcc -Wall -I*/path/to/include-files* -L*/path/to/libraries* prog.c -lctest -o prog

Use:

gcc -Wall -L/opt/lib prog.c -lctest -o prog

Where the name of the library is `libctest.so`. (This is why you must create the symbolic links or you will get the error "/usr/bin/ld: cannot find -lctest".)
The libraries will NOT be included in the executable but will be dynamically linked during runtime execution.

List Dependencies:

The shared library dependencies of the executable can be listed with the command:
`ldd` *`name-of-executable`*

Example: `ldd prog`

libctest.so.1 => /opt/lib/libctest.so.1 (0x00002aaaaaaac000)

libc.so.6 => /lib64/tls/libc.so.6 (0x0000003aa4e00000)

/lib64/ld-linux-x86-64.so.2 (0x0000003aa4c00000)

[Potential Pitfall]: Unresolved errors within a shared library may cause an error when the library is loaded. Example:

Error message at runtime:

ERROR: unable to load lib*name-of-lib*.so

ERROR: unable to get function address

Investigate error:

[prompt]$ ldd lib*name-of-lib*.so

libglut.so.3 => /usr/lib64/libglut.so.3 (0x00007fb582b74000)

libGL.so.1 => /usr/lib64/libGL.so.1 (0x00007fb582857000)

libX11.so.6 => /usr/lib64/libX11.so.6 (0x00007fb582518000)

libIL.so.1 (0x00007fa0f2c0f000)

libcudart.so.4 => not found

The first three libraries show that there is a path resolution. The last two are problematic.

The fix is to resolve dependencies of the last two libraries when linking the library `lib`*`name-of-lib`*`.so`:

- Add the unresolved library path in `/etc/ld.so.conf.d/`*`name-of-lib`*`-x86_64.conf` and/or `/etc/ld.so.conf.d/`*`name-of-lib`*`-i686.conf`
  Reload the library cache (`/etc/ld.so.cache`) with the command: `sudo ldconfig`
  or
- Add library and path explicitly to the compiler/linker command: `-l`*`name-of-lib`* `-L/path/to/lib`
  or
- Add the library path to the environment variable to fix runtime dependency: `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/lib`

Run Program:

- Set path: `export LD_LIBRARY_PATH=/opt/lib:$LD_LIBRARY_PATH`
- Run: `prog`

Man Pages:

- gcc - GNU C compiler
- ld - The GNU Linker
- ldd - List library dependencies
- ldconfig - configure dynamic linker run time bindings (update cache `/etc/ld.so.cache`)

Links:

- LDP: Shared libraries

20. operating modes of arm?

Abort

The privileged exception mode for handling Data Aborts and Prefetch Aborts.

System

The privileged user mode for operating system functions.

Undefined

The privileged exception mode for handling Undefined instructions.

Modes other than User mode are collectively known as privileged modes. Privileged modes are used to service exceptions or to access protected resources.

Processor operating modes

There are seven processor modes of operation:

User

The nonprivileged mode for normal program execution.

Fast interrupt (FIQ)

The privileged exception mode for handling fast interrupts.

Interrupt (IRQ)

The privileged exception mode for handling regular interrupts.

Supervisor

The privileged mode for operating system functions.

21. make menu config?

make config - Plain text interface (most commonly used choice)

make menuconfig - Text-based with colored menus and radiolists. This options

allows developers to save their progress. ncurses must be installed (sudo apt-get install libncurses5-dev).

Is used to configure the kernel and load as module or plot form driver we have to set

22. if there are two computers and there are files on one computer.how do u copy it to other computer?

You can use ssh, to transfer files. So you can transfer a file like this:

```
scp /home/user/examplefile otheruser@192.168.1.101:/home/otheruser/
```

```
scp -r remoteusername@192.168.1.56:/home/vrc/Desktop/www
/home/ourusername/Desktop
```

23. what is hashing technique?

Hashing is the process of mapping large amount of data item to a smaller table with the help of a hashing function. The essence of hashing is to facilitate the next level searching method when compared with the linear or binary search. The advantage of this searching method is its efficiency to hand vast amount of data items in a given collection (i.e. collection size).

Due to this hashing process, the result is a **Hash data structure** that can store or retrieve data items in an average time disregard to the collection size.

Hash Table is the result of storing the hash data structure in a smaller table which incorporates the hash function within itself. The **Hash Function** primarily is responsible to map between the original data item and the smaller table itself. Here the mapping takes place with the help of **an output integer in a consistent range** produced when a given data item (any data type) is provided for storage and this **output integer range** determines the location in the smaller table for the data item. In terms of implementation, the hash table is constructed with the help of an array and the indices of this array are associated to the output integer range.

Wipro Questions:

1.which ipc mechanism u used in your driver?

2types of semaphores--- explain

3A.bout projects.. PCI n UART.

4which linked list u used in projct

difference between processs and thread.

Threads

Will by default share memory
Will share file descriptors
Will share filesystem context
Will share signal handling

Processes

Will by default not share memory
Most file descriptors not shared
Don't share filesystem context
Don't share signal handling

how memory is allocated for process and thread.

locking mechanism in threads.

CPU scheduling, can round robin scheduling be pre emptive?

If an interrupt occurs.. does in run on pre emptive or non preemptive

What is pre-emptive and non-preemptive scheduling?

Answer
Tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called preemptive scheduling.

Eg: Round robin
In non-preemptive scheduling, a running task is executed till completion. It cannot be interrupted.
Eg First In First Out

What is pre-emptive and non-preemptive scheduling?

Preemptive scheduling: The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.

Non-Preemptive scheduling: When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.

to set bits from nth position to mth position without changing remaining bits..

addition of twonumbers using bit wise

(x & y) << 1 and add it to x ^ y to get the required result.

```
#include<stdio.h>

int Add(int x, int y)
{
    // Iterate till there is no carry
    while (y != 0)
    {
        // carry now contains common set bits of x and y
        int carry = x & y;

        // Sum of bits of x and y where at least one of the bits is
not set
        x = x ^ y;

        // Carry is shifted by one so that adding it to x gives the
required sum
        y = carry << 1;
    }
    return x;
}

int main()
{
    printf("%d", Add(15, 32));
    return 0;
}
```

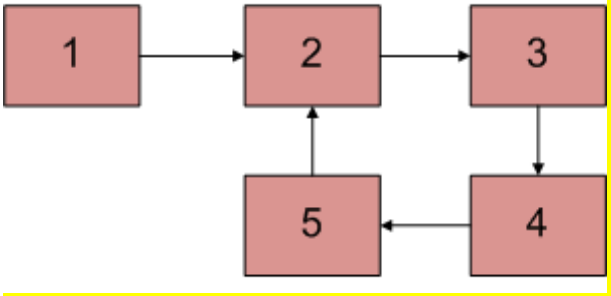
Following is recursive implementation for the same approach.

```
int Add(int x, int y)
{
    if (y == 0)
        return x;
    else
        return Add( x ^ y, (x & y) << 1);
}
```

linked list 1->2->3->4->5->6->2 here there Is a loop in list. how do u find that loop?

Write a C function to detect loop in a linked list

Below diagram shows a linked list with a loop



Following are different ways of doing this

Use Hashing:

Traverse the list one by one and keep putting the node addresses in a Hash Table. At any point, if NULL is reached then return false and if next of current node points to any of the previously stored nodes in Hash then return true.

Mark Visited Nodes:

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the linked list and keep marking visited nodes. If you see a visited node again then there is a loop. This solution works in O(n) but requires additional information with each node.

A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Just store the addresses of visited nodes in a hash and if you see an address that already exists in hash then there is a loop.

Floyd's Cycle-Finding Algorithm:

This is the fastest method. Traverse linked list using two pointers. Move one pointer by one and other pointer by two. If these pointers meet at some node then there is a loop. If pointers do not meet then linked list doesn't have loop.

Implementation of Floyd's Cycle-Finding Algorithm:

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
  int data;
  struct node* next;
};
```

```
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
  (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

int detectloop(struct node *list)
{
  struct node  *slow_p = list, *fast_p = list;

  while(slow_p && fast_p &&
          fast_p->next )
  {
    slow_p = slow_p->next;
    fast_p  = fast_p->next->next;
    if (slow_p == fast_p)
    {
       printf("Found Loop");
       return 1;
    }
  }
  return 0;
}

/* Drier program to test above function*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  push(&head, 20);
  push(&head, 4);
  push(&head, 15);
  push(&head, 10);
```

```
  /* Create a loop for testing */
  head->next->next->next->next = head;
  detectloop(head);

  getchar();
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Questions smart play

PCI n arm porting Explain

--write a function for sizeof operator

Table of content [hide]

C Programming sizeof operator

1. sizeof operator is used to calcualte the size of data type or variables.
2. sizeof operator can be nested.
3. sizeof operator will return the size in integer format.
4. sizeof operator syntax looks more like a function but it is considered as an operator in c programming

Example of sizeof() operator

Size of Variables

```
#include<stdio.h>


int main() {


  int ivar = 100;
```

```
  char cvar = 'a';
  float fvar = 10.10;

  printf("%d", sizeof(ivar));
  printf("%d", sizeof(cvar));
  printf("%d", sizeof(fvar));
  return 0;
}
```

Output :

214

In the above example we have passed a variable to sizeof operator. It will print the value of variable using sizeof() operator.

Size of Data Type

```
#include<stdio.h>

int main() {

  printf("%d", sizeof(int));
  printf("%d", sizeof(char));
  printf("%d", sizeof(float));

  return 0;
}
```

We will again get same output as that of above program. In this case we have directly passed an data type to an sizeof.

Size of constant

```
#include<stdio.h>
```

```c
int main() {


   printf("%d", sizeof(10));

   printf("%d", sizeof('A'));

   printf("%d", sizeof(10.10));


   return 0;

}
```

In this example we have passed the constant value to a sizeof operator. In this case sizeof will print the size required by variable used to store the passed value.

Nested sizeof operator

```c
#include<stdio.h>


int main() {

   int num = 10;

   printf("%d", sizeof(num));

   return 0;

}
```

We can use nested sizeof in c programming. Inner sizeof will be executed in normal fashion and the result of inner sizeof will be passed as input to outer sizeof operator.

Innermost Sizeof operator will evaluate size of Variable “num” i.e 2 bytes Outer Sizeof will evaluate **Size of constant** “2” .i.e 2 bytes

--function pointer, definition and syntax

--structure padding

Exploring memory alignment and the use of **padding** in **structures** in C/C++ Many of today's processors can address memory one 8-bit byte at a time. They can also

access memory as larger objects such as 2- or 4-byte integers, 4-byte pointers, or 8-byte floating-point numbers.

--bit field def n syntax

--array a={10,20}

p=&a

output of a[0],a[1] and *p for ++*p and *p++

--macro to reset a bit in a number

```
#define SETBIT(num,bitpos) (num|(1<<bitpos))
#define CLRBIT(num,bitpos) (num&(~(1<<bitpos)))

int a = 0x03; // a = 00000011b = 0x03(3)
SETBIT(a, 3); // a = 00001011b [3rd bit set] = 0x0b(11)

int b = 0x25; // b = 00100101b = 0x25(37)
CLRBIT(b, 2); // b = 00100001b [2nd bit clear] = 0x21(33)
```

Write a C program to set, reset, check, clear and toggle a bit.

How to set a bit?
value = data | (1 << n)). Where data is the input value and n be bit to be set.
Note: Bits starts from 0 to 7 (0000 0000). Least Significant Bit is the 0th bit.

Example: Input data is 8
Bit needs to be set is 2
value = (0000 1000) | (0000 0001 << 2) => 0000 1100 => 12

How to check a bit is set or unset?
value = data & (1 << n)

Example: Input data is 8
Bit needs to be checked is 3

value = (0000 1000) | (0000 0001 << 3) => 1
Third bit set in the given input value.

How to clear a bit?
value = data & ~(1 << n)

Example: Input data is 8
Bit needs to cleared is 3
value = (0000 1000) & (~(0000 0001 << 3))
= (0000 1000) & (~(0000 1000))
= (0000 1000) & (1111 0111) = 0

How to toggle a bit?
value = data ^ (1 << n)

Example: Input data is 8
Bit needs to be toggled is 2.
value = (0000 1000) ^ (0000 0001 << 2)
= (0000 1000) ^ (0000 0100) => (0000 1100) = 12

Interrupt Context

When executing an **interrupt** handler or bottom half, the kernel is in **interrupt context**. Recall that process **context** is the mode of operation the kernel is in while it is executing on behalf of a processfor example, executing a system call or running a kernel thread.

bottom half techniques

difference between tasklets and workques?

Tasklets:

- are old (around 2.3 I believe)
- have a straightforward, simple API
- are designed for low latency
- cannot sleep (run atomically in soft IRQ context and are guaranteed to never run on more than one CPU of a given processor, for a given tasklet)

Work queues:

- are more recent (introduced in 2.5)
- have a flexible API (more options/flags supported)
- are designed for higher latency

- can sleep

Bottom line is: use tasklets for high priority, low latency atomic tasks that must still execute outside the hard IRQ context.

You can control some level of priority with tasklets using `tasklet_hi_enable`/`tasklet_hi_schedule` (instead of their respective no-`_hi` versions). From

Can a same tasklet run on two different processsor

yes

spinlock.. usage

Semaphores are a useful tool for mutual exclusion, but they are not the only such tool provided by the kernel. Instead, most locking is implemented with a mechanism called a *spinlock*. Unlike semaphores, spinlocks may be used in code that cannot sleep, such as interrupt handlers. When properly used, spinlocks offer higher performance than semaphores in general. They do, however, bring a different set of constraints on their use.

interrupt context uses which of the following semaphore,mutex,spinlock.

mutex n semaphore diff

if a thread A has locked the mutex can thread B unlock it?

no

if in thread A has decremented a semaphore can thread B increment it?