Module Parameters

- Parameters/Command Line Arguments may be passed to the modules during module insertion
- The macro ***"module_param (name, type, perm)"*** is used to initialize the parameter at runtime.
- It takes 3 arguments which are the parameter name, parameter type and the permissions associated with the parameter
- This macro may be defined anywhere in the module

Parameter Passing

This Macro is used to mention that these variables can be initialised from command line

charp = character pointer
bool = boolean
invbool = inverted Boolean
Rest are same data types

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>

//MODULE_LICENSE("GPL");  /* <<-- tells that module bears free license          */
MODULE_AUTHOR("i am");  /* <<-- name of the author                             */

/* Variables are declared as static to keep their scope local to this module..  *
 * and avoid namespace poolution                                                */

static char* charvar = "module";
static int intvar   = 10;

/* using the following macro, variables are enabled to be modified from command-*
 * line                                                                         */
/* module_param takes three arguments: var name, type of variable, permission   */

module_param(charvar, charp, S_IRUGO);
module_param(intvar, int, S_IRUGO);


static int __init param_init(void)
{
        printk(KERN_ALERT "\n We are in init function\n");
        printk(KERN_ALERT "\n The value of charvar is %s\n", charvar);
        return 0;
}

static void __exit param_exit(void)
{
        printk(KERN_ALERT "\n GoodBye\n");
}

module_init(param_init);
module_exit(param_exit);
~
```

Parameter Passing ...

```
root@KERNEL:~/kern_mod_parm# insmod kern_parm.ko
root@KERNEL:~/kern_mod_parm# dmesg
[  998.301970]
[  998.301972]  We are in init function
[  998.301974]
[  998.301975]  The value of charvar is module
root@KERNEL:~/kern_mod_parm# lsmod | grep kern_parm
kern_parm              1596  0
root@KERNEL:~/kern_mod_parm#
```

No parameter

Printed local value

Passing command line parameters

```
root@KERNEL:~/kern_mod_parm# insmod kern_parm.ko charvar="command"
root@KERNEL:~/kern_mod_parm# dmesg
[ 1131.653534]
[ 1131.653536]  We are in init function
[ 1131.653539]
[ 1131.653539]  The value of charvar is command
root@KERNEL:~/kern_mod_parm#
```

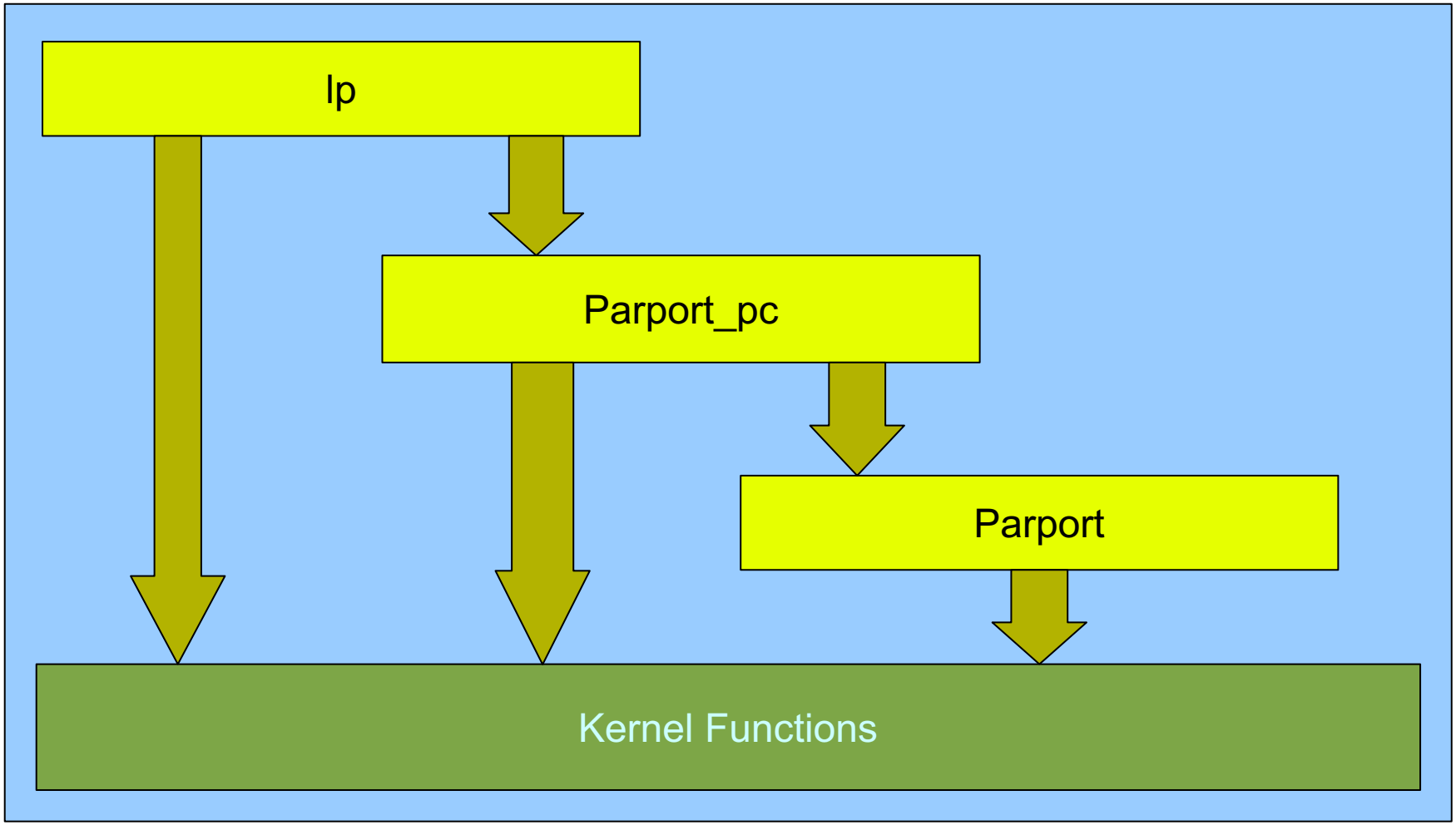
charvar is a charp variable we defined in module

Printed command line value

Library Support

- Standard C library functions cannot be used in module programming
- Functions/Variables shared between modules have to be exported as kernel symbols into a global space
  - To see the symbols exported into the kernel “cat /proc/kallsyms”
  - printk() is one such example
- External symbols used in a module (printk) should be resolved at runtime
- If symbol is not found, module will not be loaded to kernel and return back with an error

The Kernel Symbol Table



The Kernel Symbol Table

- Global Kernel Space
- Modules can export services to other modules through the use of kernel macros ***"EXPORT_SYMBOL(name)"***
  - It takes the name of the parameter or function as argument
- This exported function may be used by other modules that require similar functionality
- For example
  - Module 1 defines a function "**int Add(int a, int b)**" which takes two arguments and returns an integer output
  - This function is exported from Module1 to the Kernel Symbol Table using the macro **"EXPORT_SYMBOL(Add)"**
  - When this Module1 is inserted in the kernel, the exported symbol Add becomes available for other modules to use
  - Now, Module 2 can call the function Add() in its execution, provided Module1 is already inserted
- Please Note
  - The order of insertion and deletion of modules should always be maintained for success

Exporting symbol – Module1

Program name
kern_sym.c

This is a simple add function declearation that we are going to export

Command to export symbol

Header file containing declaration of the add function

```
/***** header file*****/
int my_add(int, int);
```

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>

MODULE_LICENSE("GPL");  /* <<-- tells that module bears free license    */
MODULE_AUTHOR("i am");  /* <<-- name of the author                      */

/* This is addition function that we are going to export as symbol      */
static int my_add(int a, int b)
{
        return (a + b);
}

/* Command to export symbol into kernel symbol table                    */
EXPORT_SYMBOL(my_add);


/* To initialise this module and load it into kernel symbol table       */

static int __init hello_init(void)
{
/* printk behaves similar to printf but it works without use of C library */
/* KERN_ALERT is the priority message; decides the seriousness of message */
        printk(KERN_ALERT "\nHELLO TO ALL\n\n");
        return 0;
}

/* This removes module from kernel symbol table                         */

static void __exit hello_exit(void)
{
        printk(KERN_ALERT "\nBYE TO ALL\n\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Exporting Symbol: Module2

```c
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>
#include"kern_add.h"

MODULE_LICENSE("GPL");  /* <<-- tells that module bears free license    */
MODULE_AUTHOR("i am");  /* <<-- name of the author                      */

static int one = 1;
static int two = 2;


static int __init add_init(void)
{
        printk(KERN_ALERT "\nwe are going to add\n");
        printk(KERN_ALERT "\nadd result is: %d\n", my_add(one,two));
        return 0;
}

static void __exit add_exit(void)
{
        printk(KERN_ALERT "\n we are leaving \n");
}

module_init(add_init);
module_exit(add_exit);
```

Header containing the declaration of symbol

We are using the symbol in this module

Resolving Multiple Dependencies

- What if, you had a module, that is dependent on 10 other modules?
  - Using insmod, you have to manually load each module in specific order, before you can load your module
  - Unloading also requires you to follow the same process
- Is there no other mechanism to automate this??
  - Use modprobe
  - modprobe works in similar way as that of insmod, but it also loads any other modules that are required by the module you want to load.

Logically Thinking….

- Questions/Doubts
  - How does the kernel know where these modules are?
  - How does the kernel know what are the symbols that are exported by each module?
- Solutions
  - Provide a standard location from where the kernel will find the module
  - Provide a dependency file which resolves the modules dependencies on other modules that provide those functionalities

How modprobe works?

- The Linux Kernel maintains a module dependency file which stores the dependencies of one module on the other.
- Modprobe checks this file to resolve the dependency of modules and loads those corresponding modules
- The file is called modules.dep and is located in **/lib/modules/`uname -r`/**
- To know the dependency, modprobe looks into modules.dep file

How to update the modules.dep file?

- This file is updated by the command ***'depmod –a'***
- depmod command searches standard locations where the kernel stores its required modules.
  - It opens the modules present in the standard location in the filesystem,
  - It identifies the symbols that are not resolved,
  - It searches for modules that export these symbols
  - It updates the modules.dep file to convey this information.
- Alternately, to make depmod search into a user defined location on the filesystem, the absolute path of the module file should be provided to depmod
  - e.g depmod -a /home/user/test/kern_sym.ko /home/user/test/kern_add.ko

The mechanics of modprobe

- modprobe can now search for the modules that export the unresolved symbols by looking into the modules.dep file and loading the corresponding modules
- modprobe searches for modules in standard directories
  - /lib/modules/(kernel version)/
- To put your modules in standard directories, use the label to execute the modules_install statement in your Makefile.

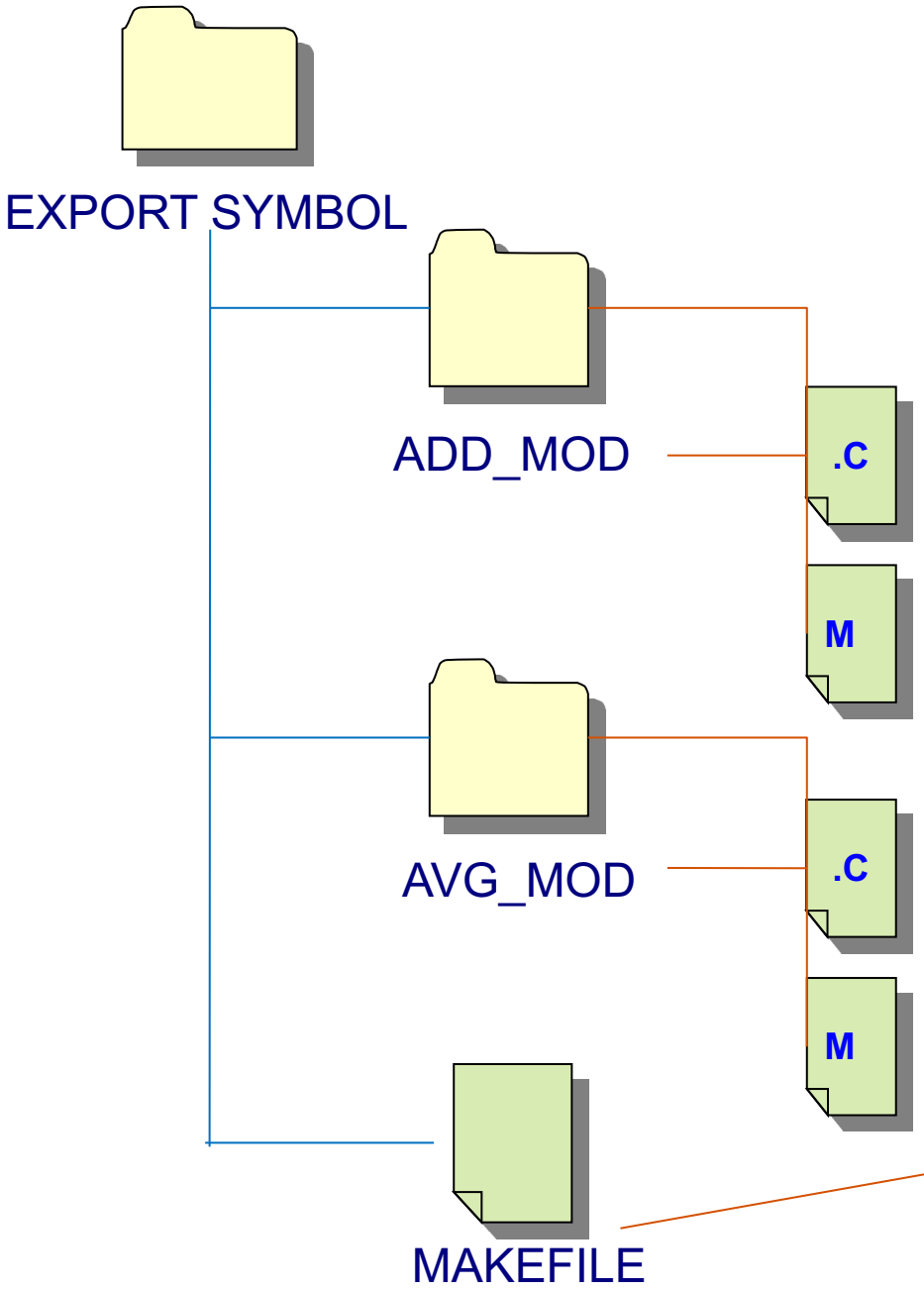
What should we do to use modprobe?

- Modify the Makefile
- Add these lines to Makefile:
  - install:

    <next line><tab>$(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install

    depmod -a
- Doing “make install” will create a folder named “**extra**” in /lib/modules/`uname -r`/ folder and copies all .ko files in the current directory into that folder
- When we do modprobe, it takes .ko files from there and loads into the kernel

Exporting Symbol – A few additions...




```
obj-y := AVG_MOD/ ADD_MOD/

KERNELDIR = /lib/modules/$(shell uname -
            r)/build
PWD := $(shell pwd)

default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD)
modules

clean:
	$(MAKE) -C $(KERNELDIR) M=$(PWD)
clean

install:
	$(MAKE) -C $(KERNELDIR) M=$(PWD)
modules_install
	depmod -a
```