

Advanced C and System Programming

Anandkumar



D-Bus

D-Bus

- ▶ Created in 2002
- ▶ Is part of the *freedesktop.org* project
- ▶ Maintained by RedHat and the community
- ▶ Is an Inter-process communication mechanism
- ▶ Initiated to standardize services of Linux desktop environments



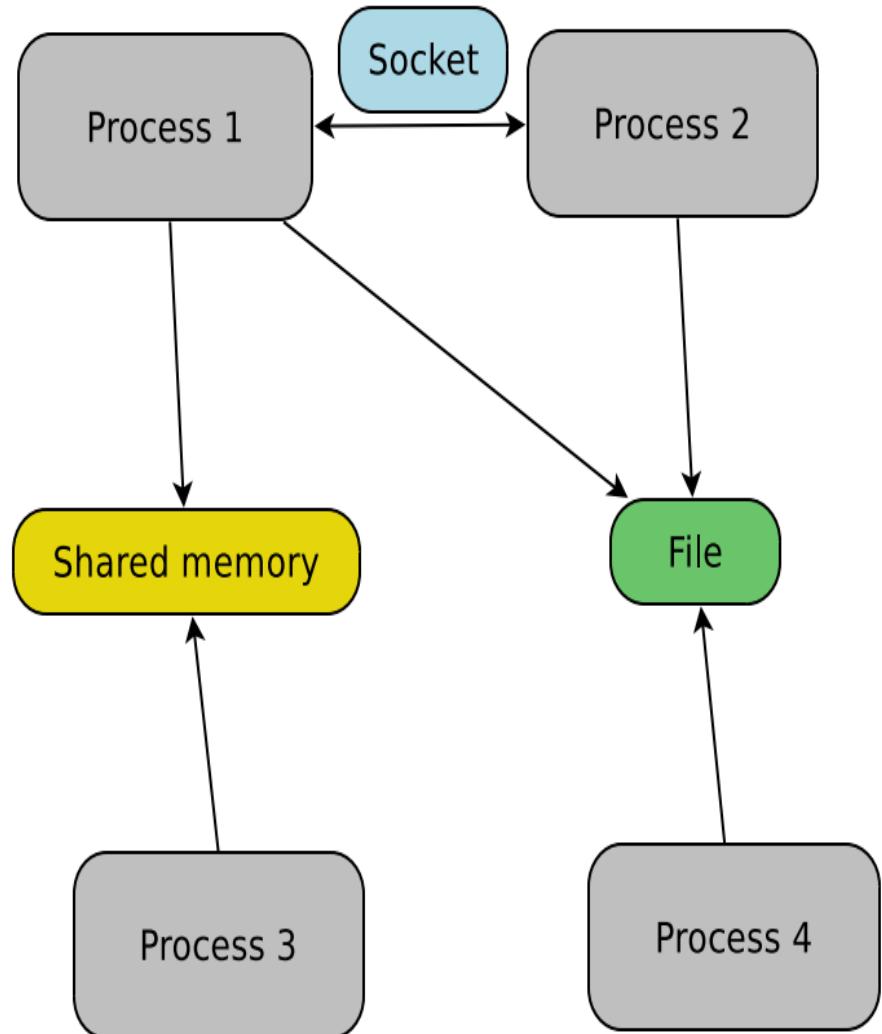
freedesktop.org



redhat

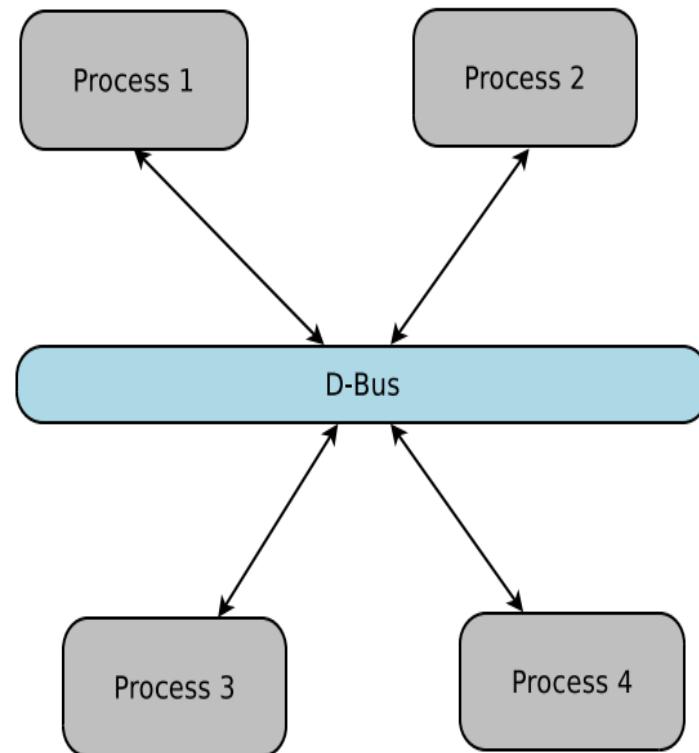
D-Bus

- ▶ Mechanisms allowing processes to communicate with each other
 - ▶ **Shared memory:** read/write into a defined memory location
 - ▶ **Memory-mapped file:** same as shared memory but uses a file
 - ▶ **Pipe:** two-way data stream (standard input / output)
 - ▶ **Named pipe:** same as pipe but uses a file (FIFO)
 - ▶ **Socket:** communication even on distant machines
 - ▶ and others



D-Bus

- ▶ Uses the socket mechanism
- ▶ Provides software bus abstraction
- ▶ Way simpler than most alternatives



D-Bus

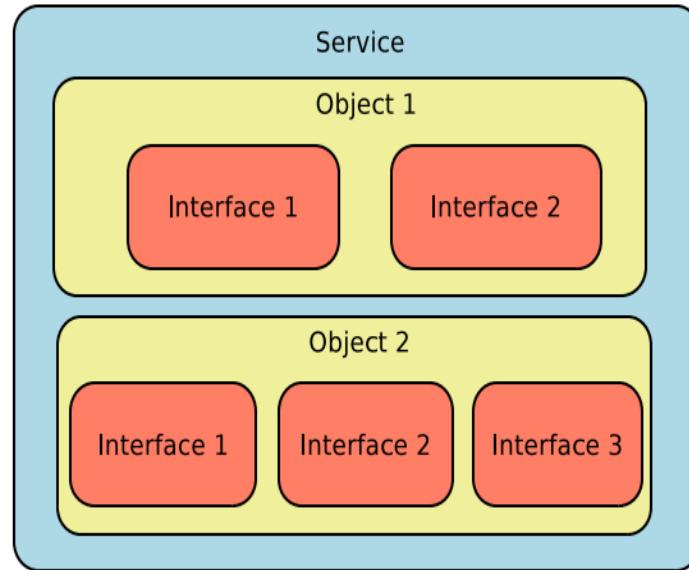
- ▶ D-Bus includes:
 - ▶ libdbus: a low-level library
 - ▶ dbus-daemon: a daemon based on libdbus. Handles and controls data transfers between DBus peers
 - ▶ two types of busses: a `system` and a `session` one. Each bus instance is managed by a `dbus-daemon`
 - ▶ a security mechanism using `policy` files

D-Bus

- ▶ System bus
 - ▶ On desktop, a single bus for all users
 - ▶ Dedicated to system services
 - ▶ Is about low-level events such as connection to a network, USB devices, etc
 - ▶ On embedded Linux systems, this bus is often the only D-Bus type
- ▶ Session bus
 - ▶ One instance per user session
 - ▶ Provides desktop services to user applications
 - ▶ Linked to the X session

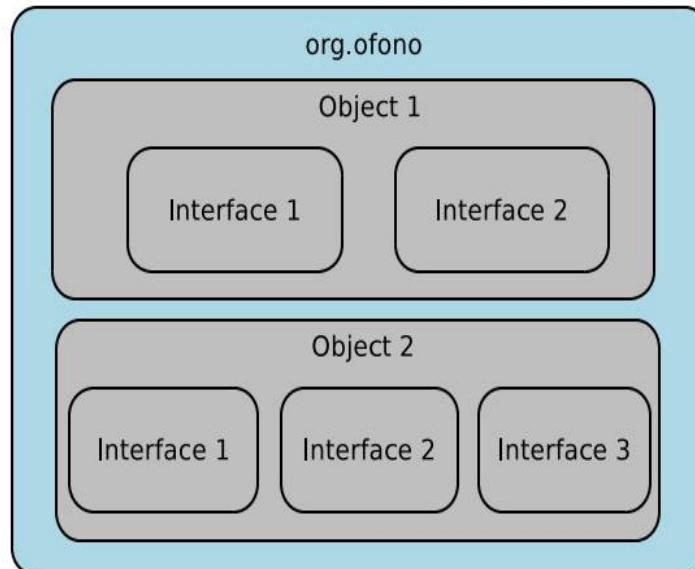
D-Bus

- ▶ D-Bus is working with different elements:
 - ▶ Services
 - ▶ Objects
 - ▶ Interfaces
 - ▶ Clients: applications using a D-Bus service
- ▶ One D-Bus *service* contains *object(s)* which implements *interface(s)*



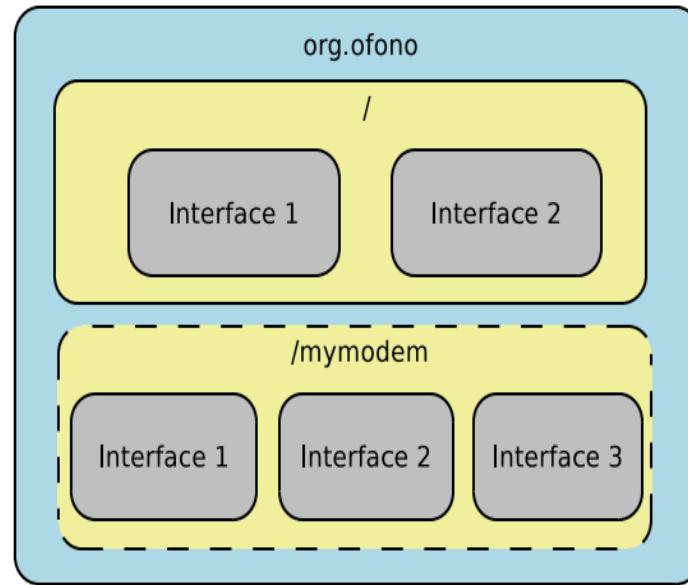
D-Bus Service

- ▶ An application can expose its services to all D-Bus users by registering to a bus instance
- ▶ A service is a collection of objects providing a specific set of features
- ▶ When an application opens a connection to a bus instance, it is assigned a unique name (ie :1.40)
- ▶ Can request a more human-readable service name: the **well-known name** (ie org.ofono) See the [freedesktop.org specification](#)



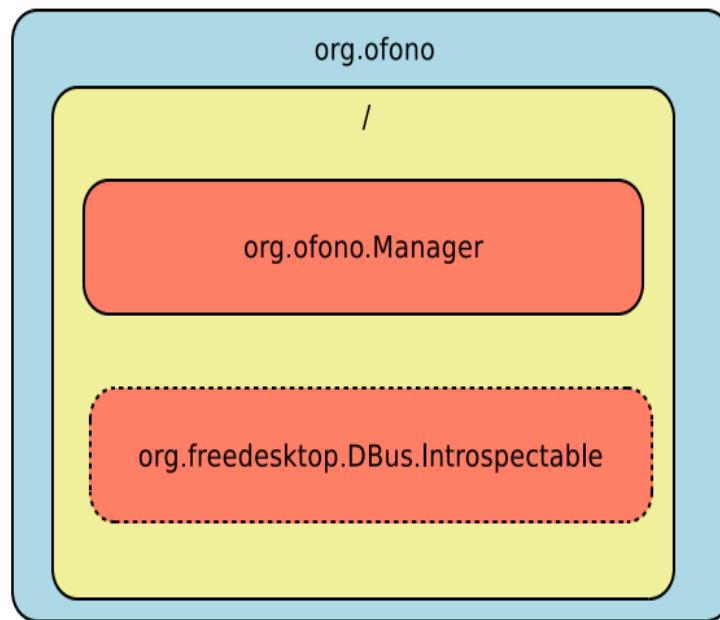
D-Bus Objects

- ▶ Are attached to one service
- ▶ Can be dynamically created or removed
- ▶ Are uniquely identified by an **object path** (ie / or /net/connman/technology/cellular)
- ▶ Implement one or several interfaces



D-Bus Interfaces

- ▶ Can be compared to a “namespace” in Java
- ▶ Has a unique name resembling Java interface names, using dots (ie `org.ofono.Manager`)
- ▶ Contains *members*: properties, methods and signals



D-Bus Interfaces

- ▶ D-Bus defines a few standard interfaces
- ▶ They all belong to the namespace “`org.freedesktop.DBus`” :
 - ▶ **`org.freedesktop.DBus.Introspectable`** : Provides an introspection mechanism.
Exposes information about the object (interfaces, methods and signals it implements)
 - ▶ **`org.freedesktop.DBus.Peer`** : Provides methods to know if a connection is alive
(ping)
 - ▶ **`org.freedesktop.DBus.Properties`** : Provides methods and signals to handle properties
 - ▶ **`org.freedesktop.DBus.ObjectManager`** : Provides an helpful API to handle sub-tree objects
- ▶ Interfaces expose properties, methods and signals

D-Bus Properties

- ▶ Directly accessible fields
- ▶ Can be read / written
- ▶ Can be of different types defined by the D-Bus specification :
 - ▶ basic types: bytes, boolean, integer, double, ...
 - ▶ string-like types : string, object path (must be valid) and signature
 - ▶ container-types: structure, array, variant (complex types) and dictionary entry (hash)
- ▶ Very convenient standard interface : `org.freedesktop.DBus.Properties`
- ▶ Types are represented by characters

byte	y	string	s	variant	v
boolean	b	object-path	o	array of int32	ai
int32	i	array	a	array of an array of int32	aai
uint32	u	struct	()	array of a struct with 2 int32 fields	a(ii)
double	d	dict	{}	dict of string and int32	{si}

D-Bus Methods

- ▶ allow remote procedure calls from one process to another
- ▶ Can be passed one or several parameters
- ▶ Can return values/objects
- ▶ Look like any method you could know from other languages

`org.freedesktop.DBus.Properties :`

```
Get (String interface_name, String property_name) => Variant value
GetAll (String interface_name) => Dict of {String, Variant} props
Set (String interface_name, String property_name, Variant value)
```

D-Bus Signals

- ▶ Messages / notifications
- ▶ Unidirectionnal
- ▶ Sent to every clients that are listening to it
- ▶ Can contain parameters
- ▶ A client will subscribe to signals to get notifications

`org.freedesktop.DBus.Properties :`

`PropertiesChanged (String, Dict of {String, Variant}, Array of String)`

D-Bus Policy

- ▶ Adds a security mechanism
- ▶ Represented by XML files
- ▶ Handled by each `dbus-daemon` (under `/etc/dbus-1/session.d` and `/etc/dbus-1/system.d`)
- ▶ Allows the administrator to control which user can talk to which interface, which user can send message to which interface, and so on
- ▶ If you are not able to talk with a D-Bus service or get an `org.freedesktop.DBus.Error.AccessDenied` error, check this file!
- ▶ `org.freedesktop.PolicyKit1` has been created to handle all security accesses

D-Bus

- ▶ In this example, "toto" can :
 - ▶ own the interface org.ofono
 - ▶ send messages to the owner of the given service
 - ▶ call GetContexts from interface org.ofono.ConnectionManager

```
<!DOCTYPE busconfig PUBLIC
`~-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN''
`~http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd'>
<busconfig>
  <policy user="toto">
    <allow own="org.ofono"/>
    <allow send_destination="org.ofono"/>
    <allow send_interface="org.ofono.ConnectionManager" send_member="GetContexts"/>
  </policy>
</busconfig>
```

- ▶ Can allow or deny

D-Bus

- ▶ Libdbus
 - ▶ This is the low-level library used by the dbus-daemon.
 - ▶ As the homepage of the project says: *"If you use this low-level API directly, you're signing up for some pain"*.
 - ▶ Recommended to use it only for small programs and you do not want to add many dependencies
- ▶ GDbus
 - ▶ Is part of GLib (GIO)
 - ▶ Provides a very comfortable API
- ▶ QtDBus
 - ▶ Is a Qt module
 - ▶ Is useful if you already have Qt on your system
 - ▶ Contains many classes to handle/interact such as `QDBusInterface`

D-Bus

- ▶ Bindings exist for other languages: dbus-python, dbus-java, ...
- ▶ All the bindings allow to:
 - ▶ Interact with existing D-Bus services
 - ▶ Create your own D-Bus services, objects, interfaces, and so on!
 - ▶ but... D-Bus is not a high performance IPC
 - ▶ Should be used only for **control** and not data
 - ▶ For example, you can use it to activate an audio pipeline but not to send the audio stream

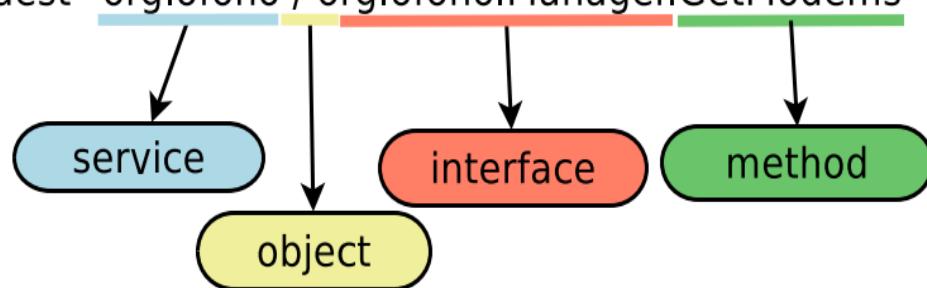
D-Bus

- ▶ Will present every tool with a demo
- ▶ dbus-send: Command-line interface (cli) to call method of interfaces (and get/set properties)
- ▶ dbus-monitor: Cli to subscribe and monitor signals
- ▶ gdbus: A GLib implementation of a more complete tool than dbus-send/monitor
- ▶ d-feet: A GUI application to handle all D-Bus services
- ▶ and others...

D-Bus

- ▶ Can chose the session or system bus (`--session` or `--system`)
- ▶ Here is an example:

```
dbus-send --system --print-reply --dest=org.ofono / org.ofono.Manager.GetModems
```



D-Bus

- ▶ Get properties:

```
dbus-send --system --print-reply --dest=net.connman / net.connman.Clock.GetProperties
```

- ▶ Set property:

```
dbus-send --system --print-reply --dest=net.connman \  
/ net.connman.Clock SetProperty \  
string:TimeUpdates variant:string:manual
```

- ▶ Using standard interfaces:

```
dbus-send --system --print-reply --dest=net.connman \  
/ org.freedesktop.DBus.Introspectable.Introspect
```

```
dbus-send --system --print-reply --dest=fi.w1.wpa_supplicant1 \  
/fi/w1/wpa_supplicant1 org.freedesktop.DBus.Properties.Get \  
string:fi.w1.wpa_supplicant1 string:Interfaces
```

D-Bus

- ▶ Can monitor all traffic (including methods and signals if enabled in policy):
`dbus-monitor`
- ▶ Or filter messages based on the interface:
`dbus-monitor --system type=signal interface=net.connman.Clock`

D-Bus

- ▶ Also provides a command line interface
- ▶ Is more featureful than `dbus-send` because it handles “dict entry”
- ▶ Has a different interface: must add a “command” such as “call” or “monitor”

```
gdbus call --system --dest net.connman \
            --object-path / --method net.connman.Clock.GetProperties
gdbus call --system --dest net.connman --object-path / \
            --method net.connman.Clock SetProperty 'TimeUpdates' "<'manual'>"
gdbus monitor --system --dest net.connman
```

- ▶ Can even emit signals

```
gdbus emit --session --object-path / --signal \\
            net.connman.Clock.PropertyChanged ``['TimeUpdates', ``\<'auto'\>'' ]''
```

D-Bus

- ▶ KDE: A desktop environment based on Qt
- ▶ Gnome: A desktop environment based on gtk
- ▶ Systemd: An init system
- ▶ Bluez: A project adding Bluetooth support under Linux
- ▶ Pidgin: An instant messaging client
- ▶ Network-manager: A daemon to manage network interfaces
- ▶ Modem-manager: A daemon to provide an API to dial with modems - works with Network-Manager
- ▶ Connman: Same as Network-Manager but works with Oftono for modem
- ▶ Oftono: A daemon that exposing features provided by telephony devices such as modem

Serial programming using Raspberry pi

Memory management

3.2 Variable Partitioning

With fixed partitions we have to deal with the problem of determining the number and sizes of partitions to minimize internal and external fragmentation.

If we use variable partitioning instead, then partition sizes may vary dynamically.

In the variable partitioning method, we keep a table (linked list) indicating used/free areas in memory.

3.2 Variable Partitioning

Initially, the whole memory is free and it is considered as one large block.

When a new process arrives, the OS searches for a block of free memory large enough for that process.

We keep the rest available (free) for the future processes.

If a block becomes free, then the OS tries to merge it with its neighbors if they are also free.

3.2 Variable Partitioning

There are three algorithms for searching the list of free blocks for a specific amount of memory.

First Fit

Best Fit

Worst Fit

first fit

- ❑ First Fit : Allocate the first free block that is large enough for the new process.
- This is a fast algorithm.

first fit

Initial memory mapping

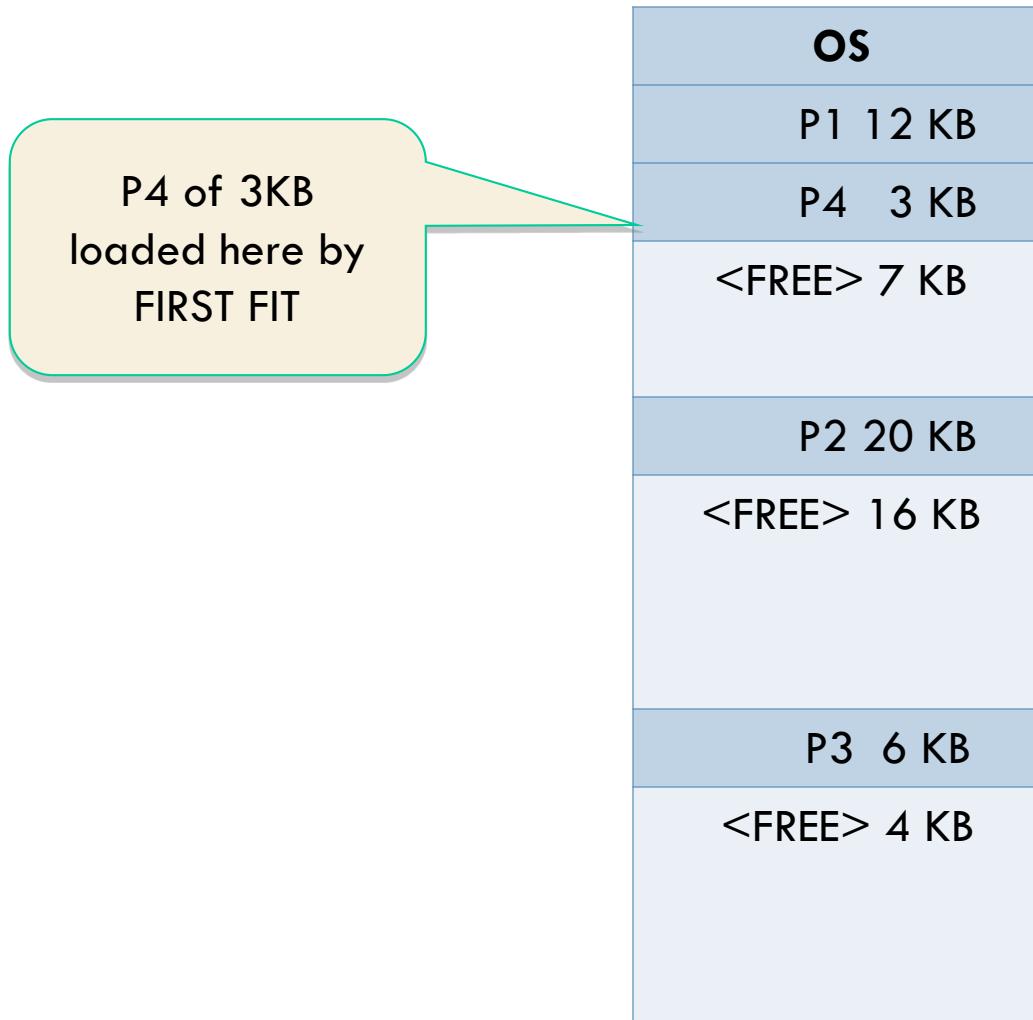
OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

first fit

P4 of 3KB
arrives

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

first fit



first fit

P5 of 15KB
arrives

OS
P1 12 KB
P4 3 KB
<FREE> 7 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

first fit

P5 of 15 KB
loaded here by
FIRST FIT

OS	
P1	12 KB
P4	3 KB
<FREE>	7 KB
P2	20 KB
P5	15 KB
<FREE>	1 KB
P3	6 KB
<FREE>	4 KB

Best fit

- Best Fit : Allocate the smallest block among those that are large enough for the new process.
- In this method, the OS has to search the entire list, or it can keep it sorted and stop when it hits an entry which has a size larger than the size of new process.
- This algorithm produces the smallest left over block.
- However, it requires more time for searching all the list or sorting it
- If sorting is used, merging the area released when a process terminates to neighboring free blocks, becomes complicated.

best fit

Initial memory
mapping

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

best fit

P4 of 3KB
arrives

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

best fit

P4 of 3KB
loaded here by
BEST FIT

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
P4 3 KB
<FREE> 1 KB

best fit

P5 of 15KB
arrives

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
P4 3 KB
<FREE> 1 KB

best fit

P5 of 15 KB
loaded here by
BEST FIT

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P5 15 KB
<FREE> 1 KB
P3 6 KB
P4 3 KB
<FREE> 1 KB

worst fit

- ❑ Worst Fit : Allocate the largest block among those that are large enough for the new process.
- Again a search of the entire list or sorting it is needed.
- This algorithm produces the largest over block.

worst fit

Initial memory mapping

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

worst fit

P4 of 3KB
arrives

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
<FREE> 16 KB
P3 6 KB
<FREE> 4 KB

worst fit

P4 of 3KB
Loaded here by
WORST FIT

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P4 3 KB
<FREE> 13 KB
P3 6 KB
<FREE> 4 KB

worst fit

No place to load
P5 of 15K

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P4 3 KB
<FREE> 13 KB
P3 6 KB
<FREE> 4 KB

worst fit

No place to load
P5 of 15K

Compaction is
needed !!

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P4 3 KB
<FREE> 13 KB
P3 6 KB
<FREE> 4 KB

compaction

Compaction is a method to overcome the external fragmentation problem.

All free blocks are brought together as one large block of free space.

Compaction requires dynamic relocation.

Certainly, compaction has a cost and selection of an optimal compaction strategy is difficult.

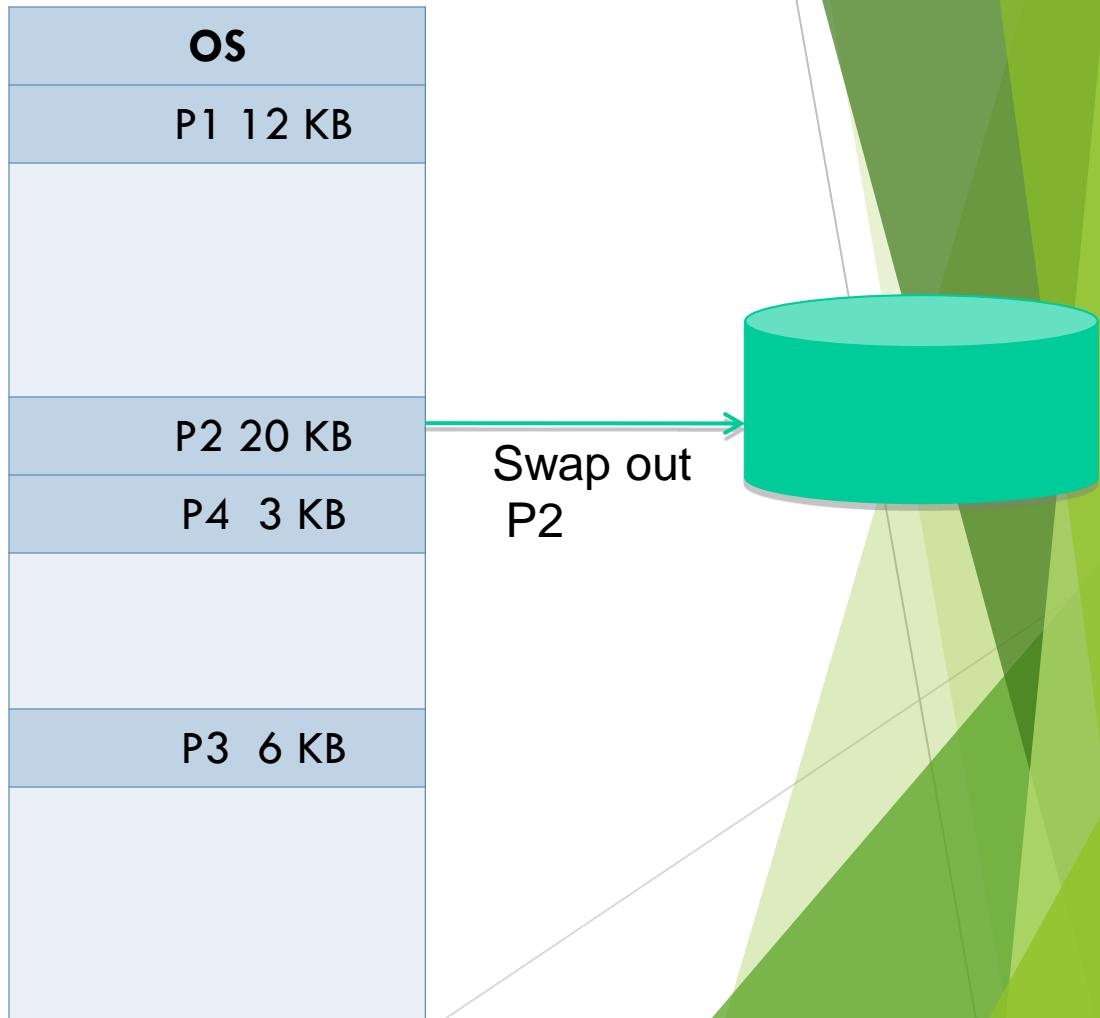
One method for compaction is swapping out those processes that are to be moved within the memory, and swapping them into different memory locations

compaction

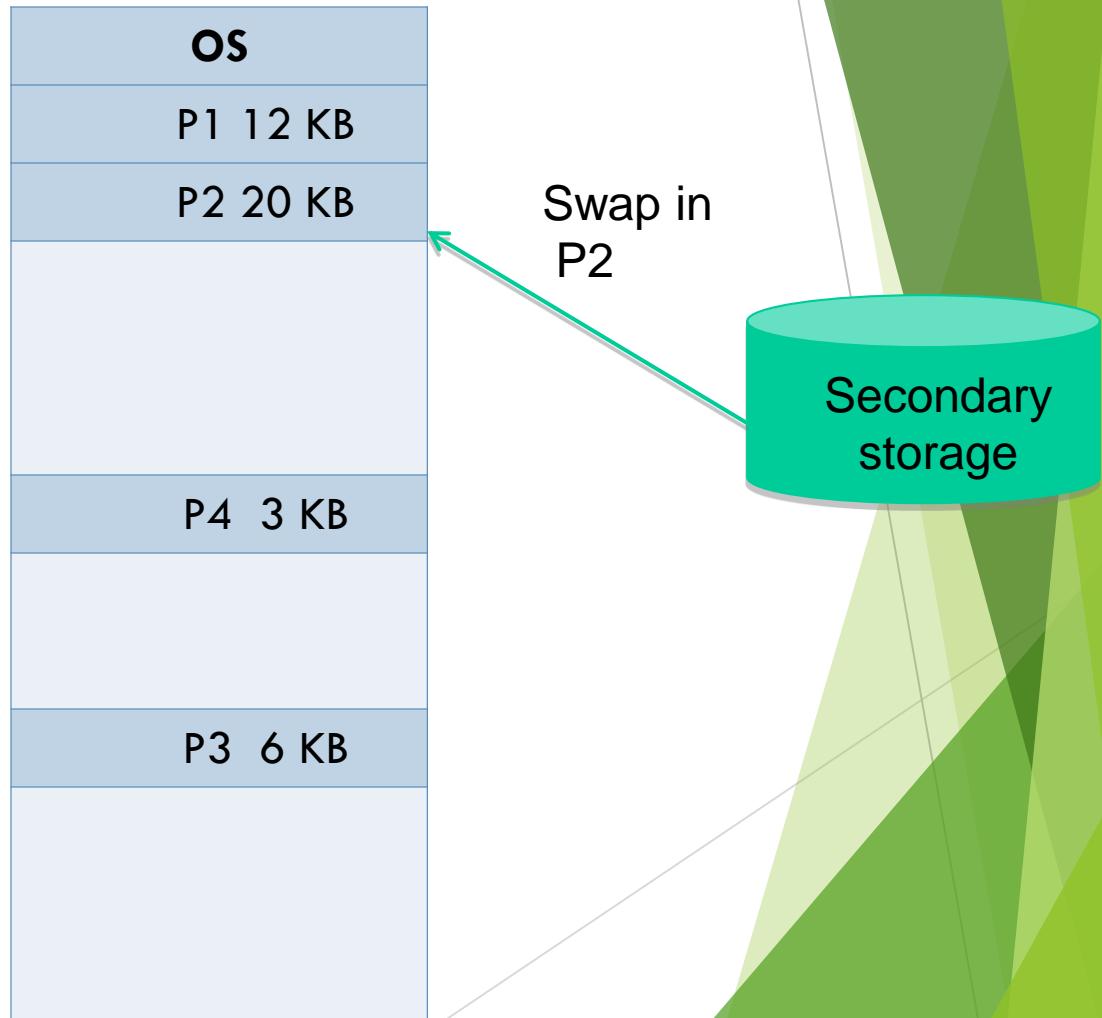
Memory mapping
before
compaction

OS
P1 12 KB
<FREE> 10 KB
P2 20 KB
P4 3 KB
<FREE> 13 KB
P3 6 KB
<FREE> 4 KB

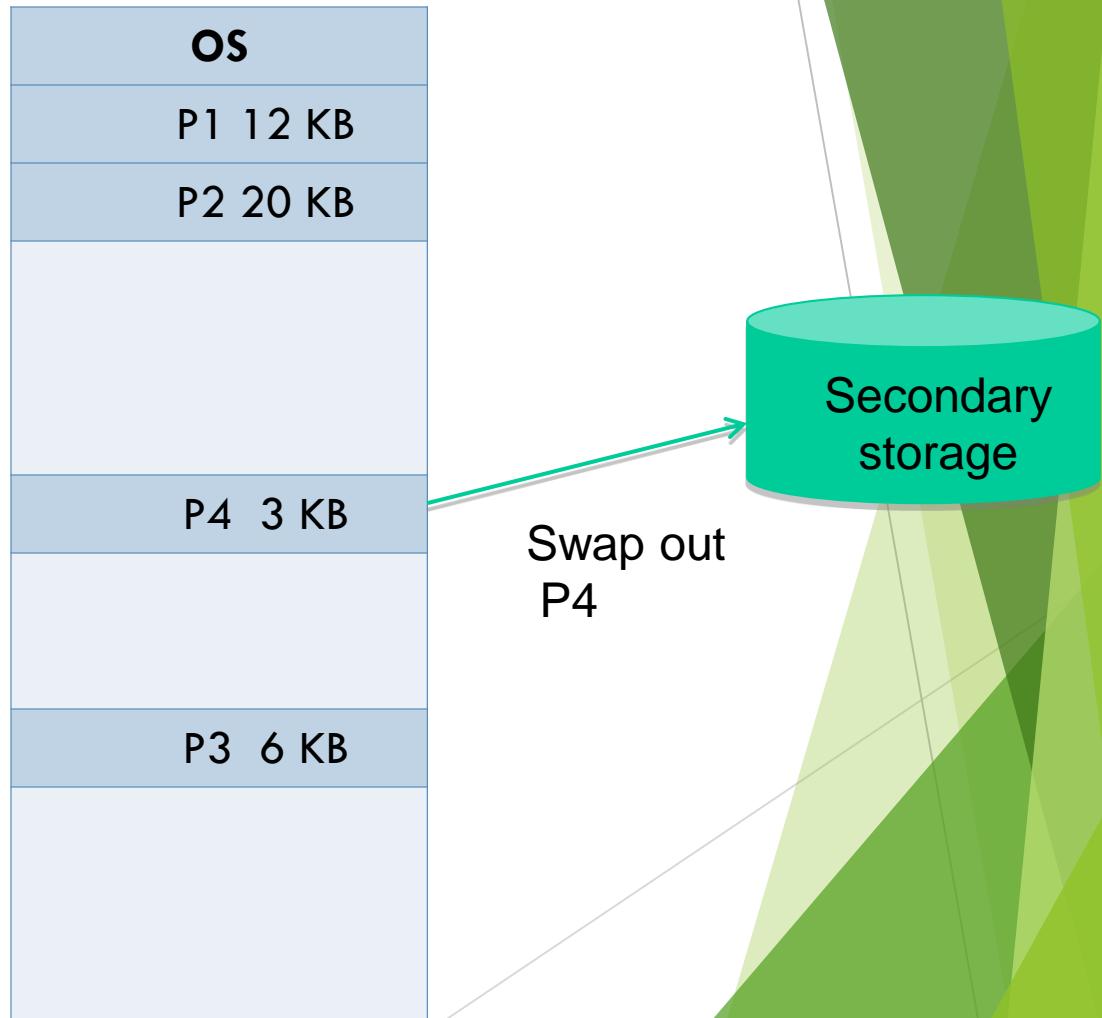
compaction



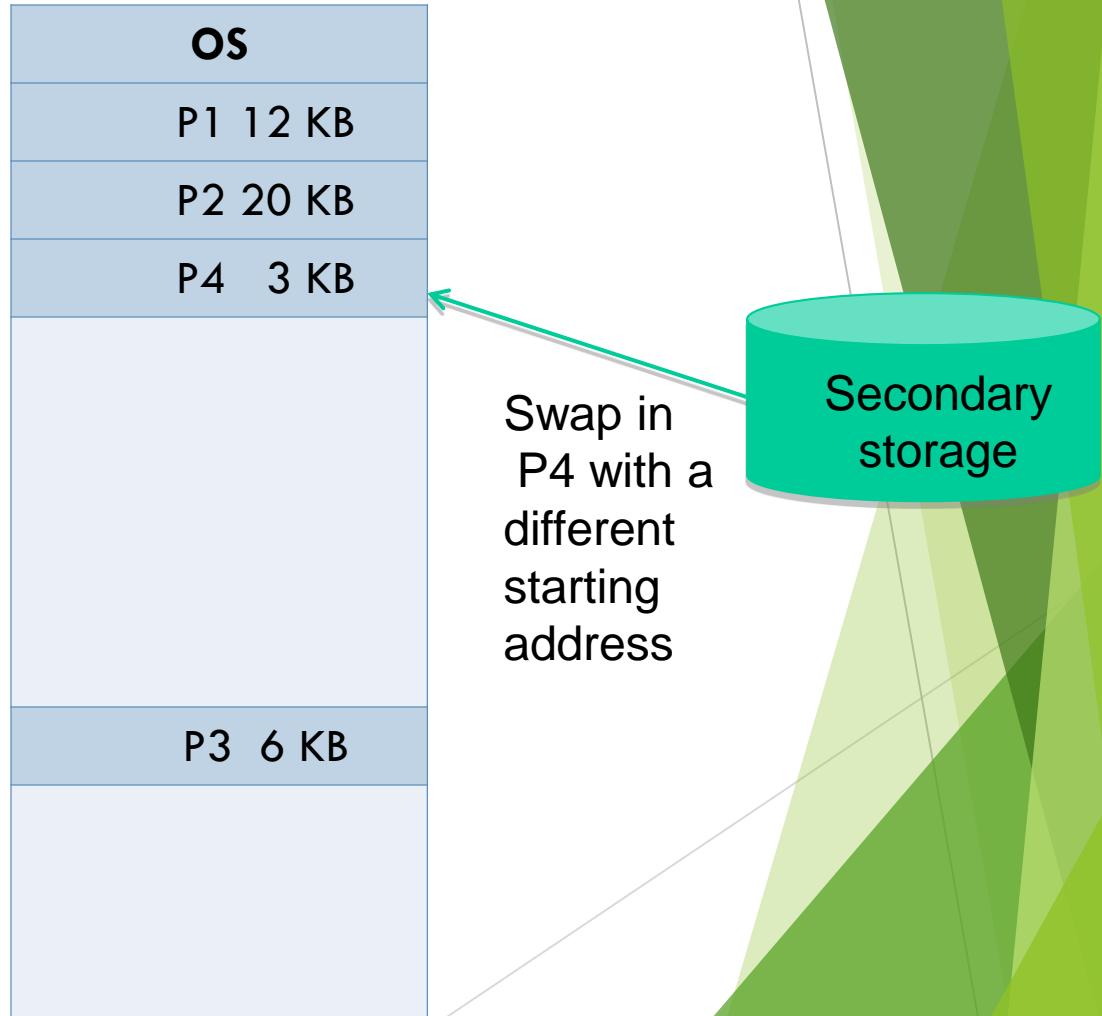
compaction



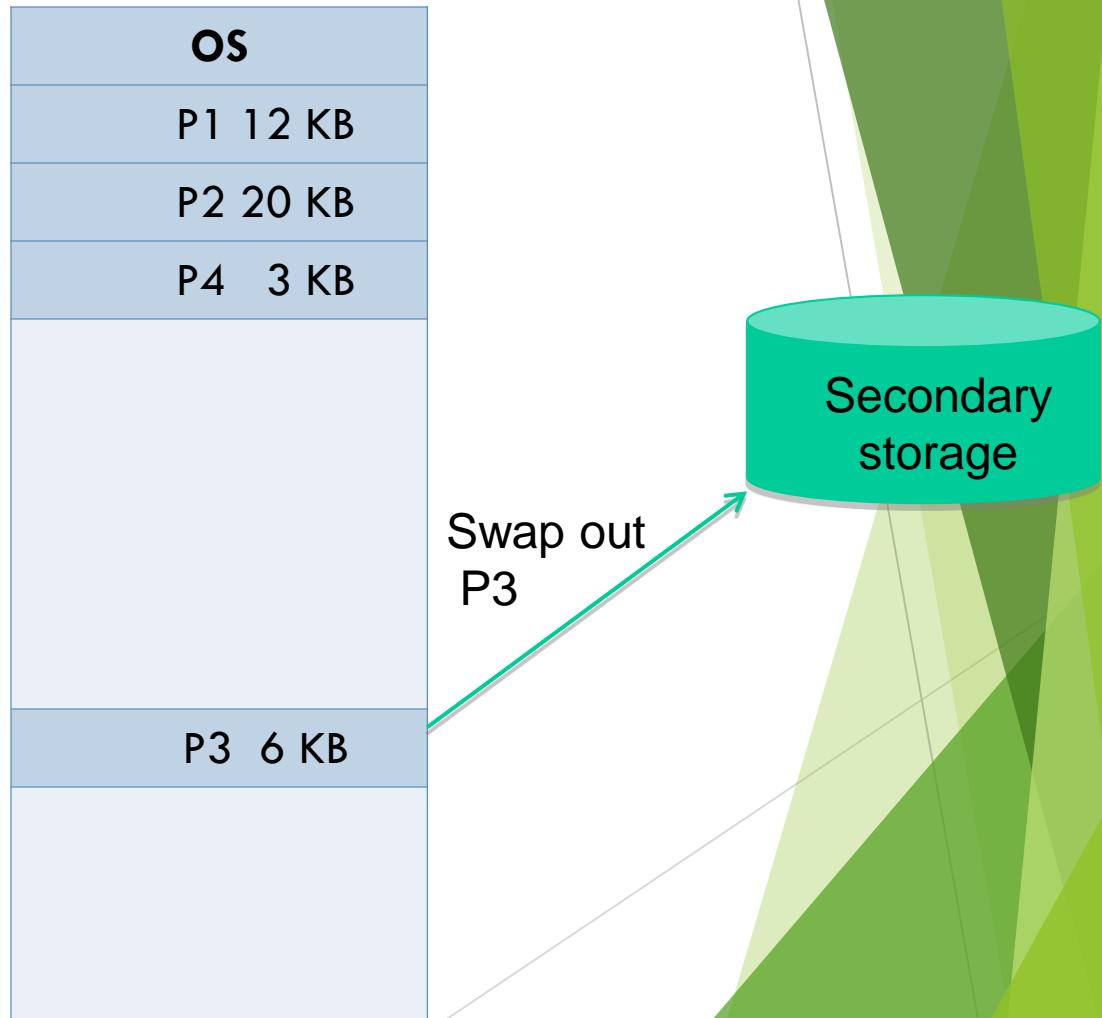
compaction



compaction



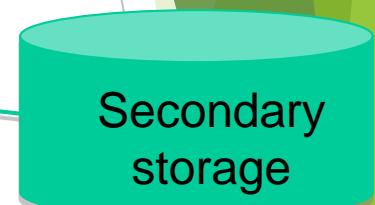
compaction



compaction

OS	
P1	12 KB
P2	20 KB
P4	3 KB
P3	6 KB

Swap in
P3



compaction

Memory mapping
after compaction

OS	
P1	12 KB
P2	20 KB
P4	3 KB
P3	6 KB
<FREE>	27 KB

Now P5 of 15KB
can be loaded
here

compaction

OS
P1 12 KB
P2 20 KB
P4 3 KB
P3 6 KB
P5 12 KB
<FREE> 12 KB

P5 of 15KB is loaded

Memory Allocation Functions

malloc

Allocates requested number of bytes and returns a pointer to the first byte of the allocated space

calloc

Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.

free

Frees previously allocated space.

realloc

Modifies the size of previously allocated space.

We will only do `malloc` and `free`

Allocating a Block of Memory

A block of memory can be allocated using the function **malloc**

Reserves a block of memory of specified size and returns a pointer of type **void**

The return pointer can be type-casted to any pointer type

General format:

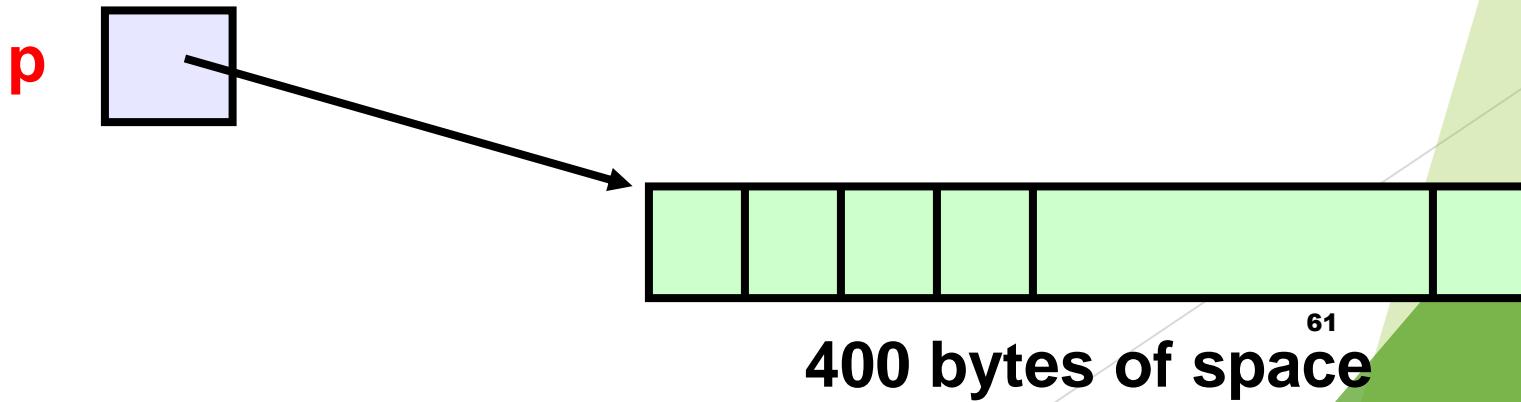
```
type *p;  
p = (type *) malloc (byte_size);
```

Example

```
p = (int *) malloc(100 * sizeof(int));
```

A memory space equivalent to **100 times the size of an int bytes** is reserved

The address of the first byte of the allocated memory is assigned to the pointer **p** of type **int**



Contd.

```
cptr = (char *) malloc (20);
```

Allocates 20 bytes of space for the pointer `cptr` of type `char`

```
sptr = (struct stud *) malloc(10*sizeof(struct  
stud));
```

Allocates space for a structure array of 10 elements. `sptr` points to a structure element of type `struct stud`

Always use sizeof operator to find number of bytes for a data type, as it can vary from machine to machine

Points to Note

malloc always allocates a block of contiguous bytes

The allocation can fail if sufficient contiguous memory space is not available

If it fails, **malloc** returns **NULL**

```
if ((p = (int *) malloc(100 * sizeof(int))) == NULL)  
{  
    printf ("\n Memory cannot be allocated");  
    exit();  
}
```

Releasing the Allocated Space: **free**

An allocated block can be returned to the system for future use by using the **free** function

General syntax:

```
free (ptr);
```

where **ptr** is a pointer to a memory block which has been previously created using **malloc**
Note that no size needs to be mentioned for the allocated block, the system remembers it for each pointer returned

Implementation of malloc and free

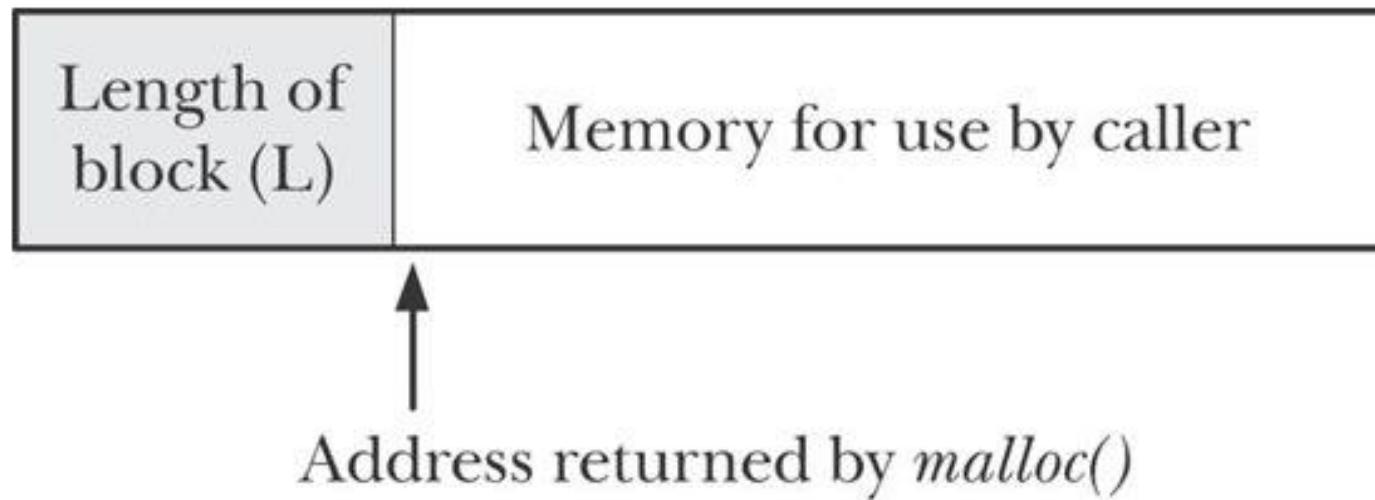


Figure 7-1. Memory block returned by malloc()

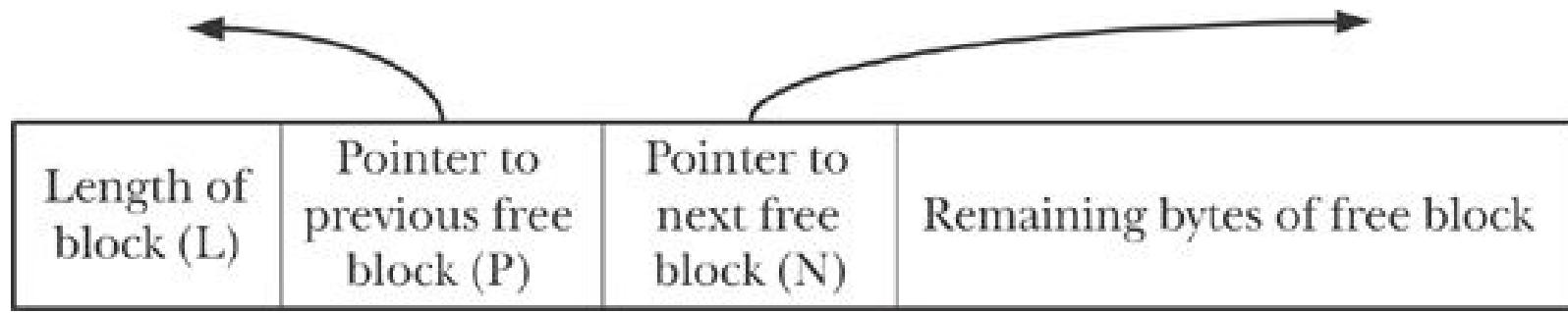


Figure 7-2. A block on the free list

Fig 7.3

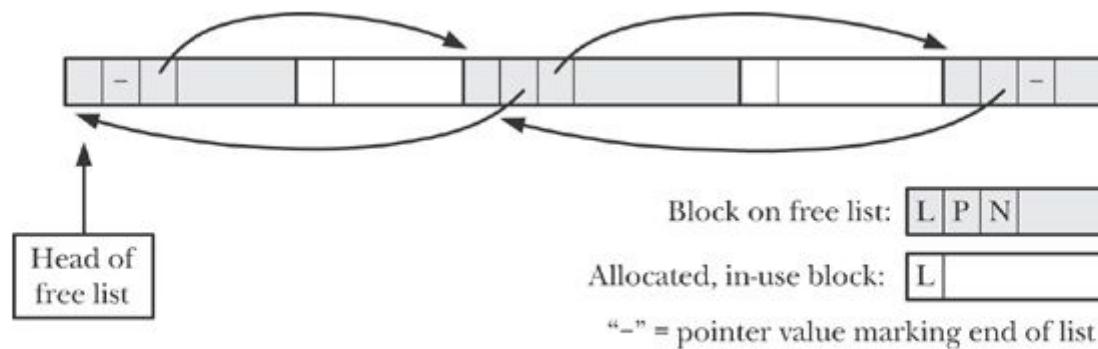


Figure 7-3. Heap containing allocated blocks and a free list

Checking for memory leaks?

NAME -- brk, sbrk - change data segment size

SYNOPSIS

```
#include <unistd.h>  
  
int brk(void *addr);  
  
void *sbrk(intptr_t increment);  
  
brk(), sbrk(): _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 500
```

DESCRIPTION - brk() and sbrk() change the location of the program break, which defines the end of the process's data segment. Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

brk() sets the end of the data segment to the value specified by addr, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see setrlimit(2)).

sbrk() increments the program's data space by increment bytes.

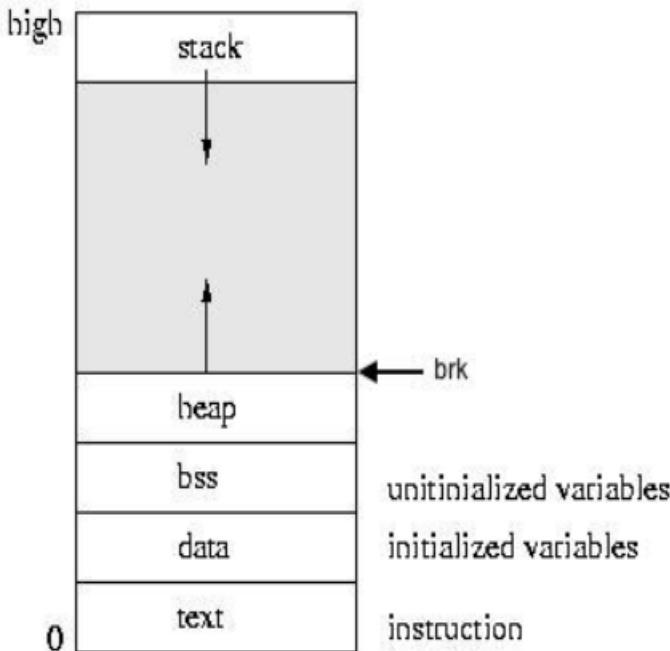
Text section: The part that contains the binary instructions to be executed by the processor.

Data section: Contains non-zero initialized static data.

BSS (Block Started by Symbol) : Contains zero-initialized static data. Static data uninitialized in program is initialized 0 and goes here.

Heap: Contains the dynamically allocated data.

Stack: Contains your automatic variables, function arguments, copy of base pointer etc.



As you can see in the image, the stack and the heap grow in the opposite directions.

Sometimes the data, bss and heap sections are collectively referred to as the "*data segment*", the end of which is demarcated by a pointer named program break or **brk**.

That is, *brk* points to the end of the heap.

Now if we want to allocate more memory in the heap, we need to request the system to increment *brk*. Similarly, to release memory we need to request the system to decrement *brk*.

`malloc_usable_size` - obtain size of block of memory allocated from heap

Synopsis

```
#include <malloc.h>
```

```
size_t malloc_usable_size (void *ptr);
```

Description

The `malloc_usable_size()` function returns the number of usable bytes in the block pointed to by *ptr*, a pointer to a block of memory allocated by [malloc\(3\)](#) or a related function.

Return Value

`malloc_usable_size()` returns the number of usable bytes in the block of allocated memory pointed to by *ptr*. If *ptr* is NULL, 0 is returned.

NAME[top](#)

malloc_stats - print memory allocation statistics

SYNOPSIS[top](#)

```
#include <malloc.h>

void malloc_stats(void);
```

DESCRIPTION[top](#)

The **malloc_stats()** function prints (on standard error) statistics about memory allocated by [malloc\(3\)](#) and related functions. For each arena (allocation area), this function prints the total amount of memory allocated and the total number of bytes consumed by in-use allocations. (These two values correspond to the *arena* and *uordblks* fields retrieved by [mallinfo\(3\)](#).) In addition, the function prints the sum of these two statistics for all arenas, and the maximum number of blocks and bytes that were ever simultaneously allocated using [mmap\(2\)](#).

`__malloc_hook`, `__malloc_initialize_hook`, `__memalign_hook`, `__free_hook`, `__realloc_hook`,
`__after_morecore_hook` - malloc debugging variables

Synopsis

```
#include <malloc.h>
void *(*__malloc_hook)(size_t size, const void *caller);
void *(*__realloc_hook)(void *ptr, size_t size", const void *" caller );

void *(*__memalign_hook)(size_t alignment, size_t size,
    const void *caller);
void (*__free_hook)(void *ptr, const void *caller);
void (*__malloc_initialize_hook)(void);
void (*__after_morecore_hook)(void);
```

Description

The GNU C library lets you modify the behavior of [`malloc\(3\)`](#), [`realloc\(3\)`](#), and [`free\(3\)`](#) by specifying appropriate hook functions. You can use these hooks to help you debug programs that use dynamic memory allocation, for example.

The variable `__malloc_initialize_hook` points at a function that is called once when the malloc implementation is initialized. This is a weak variable, so it can be overridden in the application with a definition like the following:

```
void (*__malloc_initialize_hook)(void) = my_init_hook;
```

Now the function `my_init_hook()` can do the initialization of all hooks.

`alloca` - allocate memory that is automatically freed

Synopsis

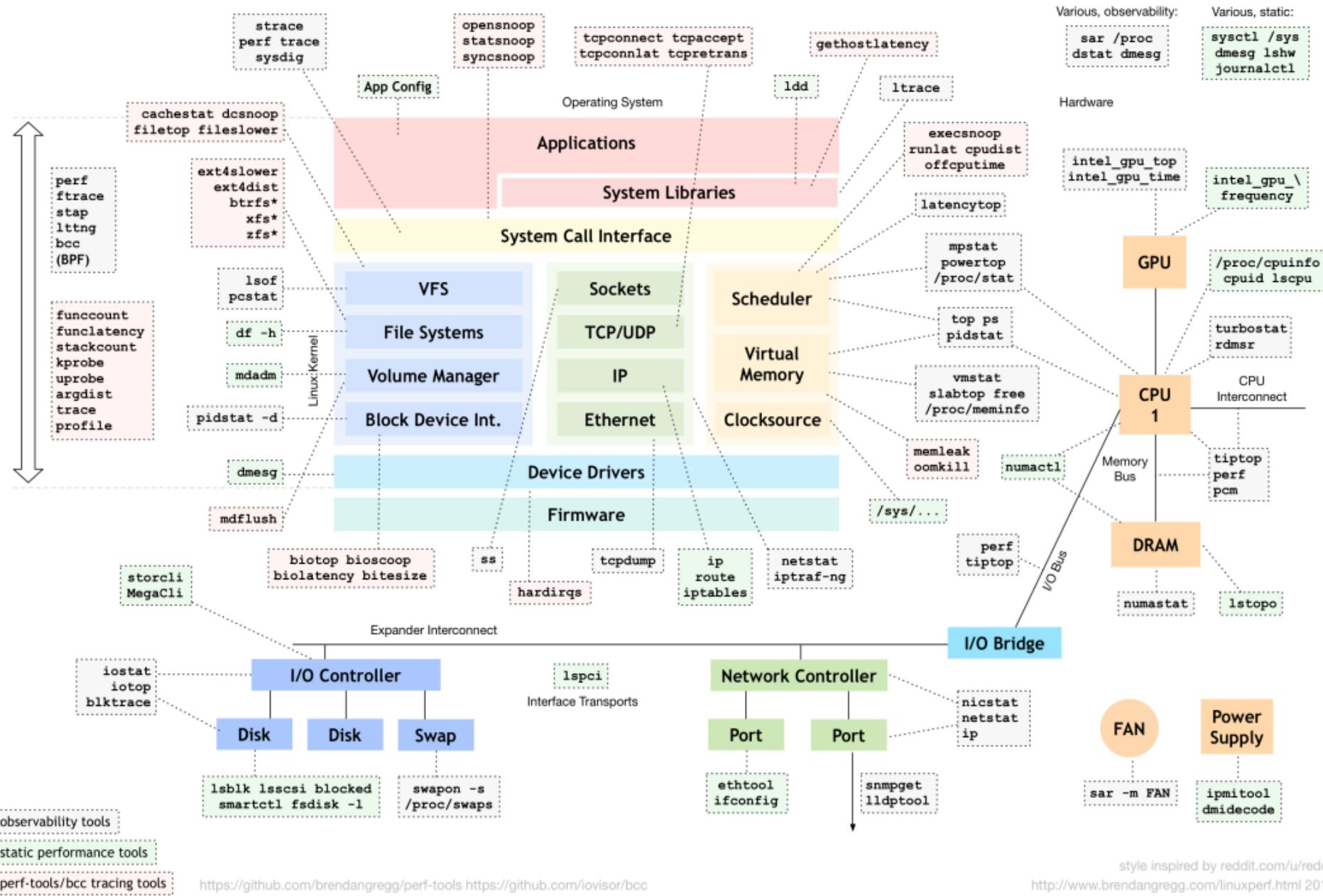
```
#include <alloca.h>
```

```
void *alloca(size_t size);
```

Description

The `alloca()` function allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed when the function that called `alloca()` returns to its caller.

Linux Application Debugging



Syslog

Syslog is a known standard for message logging. Most times, the system that does the logging and the software that gets to generate them tend to interfere during processes. But syslog helps separate the software generating the logs from the system that stores the logs, thereby making the process of logging less complicated and stressful.

- **Syslog Daemon:** It is a daemon that listens for logs and writes them to a specific location. The location(s) is defined in the configuration file for the daemon. rsyslog is the Syslog daemon shipped with most of the distros.

- **Syslog Message Format:** It refers to the syntax of Syslog messages. The syntax is usually defined by a standard (for eg RFC5424).

Syslog

- **Syslog Protocol:** It refers to the protocol used for remote logging. Modern Syslog daemons can use TCP and TLS in addition to UDP which is the legacy protocol for remote logging.

Syslog

Benefits of syslog

- Helps analyze the root cause for any trouble or problem caused
- Reduce overall downtime helping to troubleshoot issues faster with all the logs
- Improves incident management by active detection of issues
- Self-determination of incidents along with auto resolution
- Simplified architecture with different level of severity like error,info,warning etc

Syslog

Display syslogs with the ls command

Listing the contents of /var/log for an Ubuntu 20.04 machine using the ls command:

\$ sudo ls /var/log

```
test@test-VirtualBox:~$ sudo ls /var/log
[sudo] password for test:
alternatives.log      auth.log.3.gz   dmesg          journal          syslog.5.gz
alternatives.log.1    auth.log.4.gz   dmesg.0        kern.log         syslog.6.gz
alternatives.log.2.gz  boot.log       dmesg.1.gz     kern.log.1      syslog.7.gz
apport.log            boot.log.1     dmesg.2.gz     kern.log.2.gz   ubuntu-advantage.log
apport.log.1          boot.log.2     dmesg.3.gz     kern.log.3.gz   ubuntu-advantage.log.1
apport.log.2.gz        boot.log.3     dmesg.4.gz     kern.log.4.gz   ubuntu-advantage.log.2.gz
apport.log.3.gz        boot.log.4     dpkg.log       lastlog         unattended-upgrades
apport.log.4.gz        boot.log.5     dpkg.log.1    openvpn         vboxadd-install.log
apport.log.5.gz        boot.log.6     dpkg.log.2.gz  private         vboxadd-setup.log
apport.log.6.gz        boot.log.7     faillog        speech-dispatcher  vboxadd-setup.log.1
apport.log.7.gz        bootstrap.log  fontconfig.log syslog          vboxadd-setup.log.2
apt                  btmp           gdm3           syslog.1       vboxadd-setup.log.3
auth.log              btmp.1         gpu-manager.log syslog.2.gz   vboxadd-setup.log.4
auth.log.1            cups           hp             syslog.3.gz   vboxadd-uninstall.log
auth.log.2.gz          dist-upgrade  installer      syslog.4.gz   wtmp
```

test@test-VirtualBox:~\$

Syslog

View system logs in Linux using the tail command

Using the tail command you can view the last few logs. Adding the -f option lets you watch them in real time.

```
$ sudo tail -f /var/log/syslog
```

Similarly, the tail command can be used to view kernel logs (kern.log), boot logs (boot.log), etc .

Syslog

Syslog Configuration

The rules for which logs go where are defined in the Syslog daemon's configuration file.

```
$ sudo vi /etc/rsyslog.conf
```

```
$ sudo vi /etc/rsyslog.d/50-default.conf
```

```
# Default rules for rsyslog.
#
# For more information see rsyslog.conf(5) and /etc/rsyslog.conf

#
# First some standard log files. Log by facility.
#
auth,authpriv.*          /var/log/auth.log
*.*;auth,authpriv.none   -/var/log/syslog
#cron.*                  /var/log/cron.log
#daemon.*                -/var/log/daemon.log
kern.*                   -/var/log/kern.log
#lpr.*                   -/var/log/lpr.log
mail.*                   -/var/log/mail.log
#user.*                  -/var/log/user.log
local.*                  -/var/log/test41.log
```

Strace

strace is a powerful command line tool for debugging and trouble shooting programs in Unix-like operating systems such as Linux. It captures and records all system calls made by a process and the signals received by the process.

Trace Linux Command System Calls

You can simply run a command with strace like this, here we are tracing of all system calls made by the free command.

```
# strace free
```

Strace

Trace Linux Process PID

If a process is already running, you can trace it by simply passing its PID as follows; this will fill your screen with continues output that shows system calls being made by the process, to end it, press [Ctrl + C]

```
# sudo strace -p 3569
```

Get Summary of Linux Process

Using the -c flag, you can generate a report of total time, calls, and errors for each system call, as follows.

```
# strace -c free
```

Strace

Print Instruction Pointer During System Call

The **-i** option displays the instruction pointer at the time of each system call made by the program.

```
# sudo strace -i free
```

Show Time of Day For Each Trace Output Line

You can also print the time of day for each line in the trace output, by passing the **-t** flag.

```
# sudo strace -t free
```

Strace

Print Command Time Spent in System Calls

To shows the time difference between the starting and the end of each system call made by a program, use the -T option.

```
# sudo strace -T free
```

Trace Only Specific System Calls

In the command below, trace=write is known as a qualifying expression, where trace is a qualifier (others include signal, abbrev, verbose, raw, read, or write). Here, write is the value of the qualifier.

Strace

The following command actually shows the system calls to print df output on standard output.

```
# sudo strace -e trace=write free
```

```
# sudo strace -e trace=open,close free
```

```
# sudo strace -e trace=open,close,read,write free
```

```
# sudo strace -e trace=all free
```

Strace

Trace System Calls Based on a Certain Condition

Let's look at how to trace system calls relating to a given class of events. This command can be used to trace all system calls involving process management.

```
# sudo strace -e trace=process free
```

Strace

Trace System Calls Based on a Certain Condition

Let's look at how to trace system calls relating to a given class of events. This command can be used to trace all system calls involving process management.

```
# sudo strace -e trace=process free
```

Next, to trace all system calls that take a filename as an argument, run this command.

```
$ sudo strace -q -e trace=file free
```

Strace

To trace all system calls involving memory mapping, type.

```
$ sudo strace -q -e trace=memory free
```

You can trace all network and signals related system calls.

```
$ sudo strace -e trace=network free
```

```
$ sudo strace -e trace=signal free
```

Strace

Redirect Trace Output to File

To write the trace messages sent to standard error to a file, use the -o option. This means that only the command output is printed on the screen as shown below.

```
$ sudo strace -o df_debug.txt free
```

strace a program and threads

To trace both program and its threads using option -f

```
# strace -f ./program
```

Strace

strace a program and print strings

This will print the first 80 characters of every string.

```
# strace -s 80 -f ./program
```

ftrace

Ftrace is a tracing framework for the Linux kernel. It was added to the kernel back in 2008 and has evolved a lot since then.

Ftrace stands for function tracer and basically lets you watch and record the execution flow of kernel functions. It was created by Steven Rostedt, derived from two other tools called latency tracer from Ingo Molnar and Steven's logdev utility.

ftrace

With ftrace you can really see what the kernel is doing. You can trace function calls and learn a lot about how the kernel works. You can find out which kernel functions are called when you run a user space application. You can profile functions, measure execution time and find out bottlenecks and performance issues. You can identify hangs in kernel space. You can measure the time it takes to run a real-time task and find out latency issues. You can measure stack usage in kernel space and find out possible stack overflows. You can really do a lot of things to monitor and find bugs in the kernel!

ftrace

With ftrace you can really see what the kernel is doing. You can trace function calls and learn a lot about how the kernel works. You can find out which kernel functions are called when you run a user space application. You can profile functions, measure execution time and find out bottlenecks and performance issues. You can identify hangs in kernel space. You can measure the time it takes to run a real-time task and find out latency issues. You can measure stack usage in kernel space and find out possible stack overflows. You can really do a lot of things to monitor and find bugs in the kernel!

ftrace

And this is the filesystem interface provided by ftrace:

```
# ls /sys/kernel/tracing/
README                      set_ftrace_filter
available_events             set_ftrace_notrace
available_filter_functions   set_ftrace_pid
available_tracers            set_graph_function
buffer_size_kb               set_graph_notrace
buffer_total_size_kb         snapshot
current_tracer              stack_max_size
dyn_ftrace_total_info        stack_trace
enabled_functions            stack_trace_filter
events                       timestamp_mode
free_buffer                  trace
function_profile_enabled     trace_clock
hwlat_detector               trace_marker
instances                     trace_marker_raw
max_graph_depth              trace_options
options                      trace_pipe
per_cpu                       trace_stat
printk_formats               tracing_cpumask
saved_cmdlines                tracing_max_latency
saved_cmdlines_size           tracing_on
saved_tgids                   tracing_thresh
set_event                     uprobe_events
set_event_pid                 uprobe_profile
```

ftrace

We can print the list of available tracers:

```
# cat available_tracers
hwlat    blk      function_graph  wakeup_dl   wakeup_rt
wakeup   irqsoff  function        nop
```

There are function tracers (*function*, *function_graph*), latency tracers (*wakeup_dl*, *wakeup_rt*, *irqsoff*, *wakeup*, *hwlat*), I/O tracers (*blk*) and many more!

To enable a tracer, we just have to write its name to *current_tracer*:

```
# echo function > current_tracer
```

ftrace

And we can read the trace buffer with the `trace` or `trace_pipe` file:

```
# cat trace
# tracer: function
#
#          _-----> irqs-off
#          / _----> need-resched
#          | / _---> hardirq/softirq
#          || / _--> preempt-depth
#          ||| /     delay
#
#      TASK-PID  CPU#  |||  TIMESTAMP  FUNCTION
#      | |       |  |||  |       |
<idle>-0  [001] d...  23.695208: __raw_spin_lock_irqsave <-hrtimer_next_event_wi...
<idle>-0  [001] d...  23.695209: __hrtimer_next_event_base <-hrtimer_next_event...
<idle>-0  [001] d...  23.695210: __next_base <-__hrtimer_next_event_base
<idle>-0  [001] d...  23.695211: __hrtimer_next_event_base <-hrtimer_next_event...
<idle>-0  [001] d...  23.695212: __next_base <-__hrtimer_next_event_base
<idle>-0  [001] d...  23.695213: __next_base <-__hrtimer_next_event_base
<idle>-0  [001] d...  23.695214: __raw_spin_unlock_irqrestore <-hrtimer_next_eve...
<idle>-0  [001] d...  23.695215: get_iowait_load <-menu_select
<idle>-0  [001] d...  23.695217: tick_nohz_tick_stopped <-menu_select
<idle>-0  [001] d...  23.695218: tick_nohz_idle_stop_tick <-do_idle
<idle>-0  [001] d...  23.695219: rcu_idle_enter <-do_idle
<idle>-0  [001] d...  23.695220: call_cpuidle <-do_idle
<idle>-0  [001] d...  23.695221: cpuidle_enter <-call_cpuidle
```

ftrace

The [function graph tracer](#) is an alternative function tracer that traces not only the function entry but also the return of the function, allowing you to create a call graph of the function flow and output the trace data in a C-like style with information about the duration of each function.

```
# echo function_graph > current_tracer

# cat trace
# tracer: function_graph
#
# CPU  DURATION          FUNCTION CALLS
# |    |    |
#)  1.000 us   } /* idle_cpu */
#)           tick_nohz_irq_exit()
#)           ktime_get()
#)  1.000 us   clocksource_mmio_readl_up()
#)  7.666 us   }
#) + 14.000 us }
#)           rcu_irq_exit()
#)  1.334 us   rcu_nmi_exit()
#)  7.667 us   }
#) ! 556.000 us } /* irq_exit */
#) # 1187.667 us } /* __handle_domain_irq */
#) # 1194.333 us } /* gic_handle_irq */
#) <===== |
#) # 8197.333 us } /* cpuidle_enter_state */
#) # 8205.334 us } /* cpuidle_enter */
[...]
```

Trace-cmd

[trace-cmd](#) is a user-space command-line tool for ftrace created by Steven Rostedt.

It is a front-end to the tracefs filesystem and can be used to configure ftrace, read the trace buffer and save the data to a file (trace.dat by default) for further analysis.

```
# trace-cmd

trace-cmd version 2.6.1

usage:
trace-cmd [COMMAND] ...

commands:
record - record a trace into a trace.dat file
start - start tracing without recording into a file
extract - extract a trace from the kernel
stop - stop the kernel from recording trace data
restart - restart the kernel trace data recording
show - show the contents of the kernel tracing buffer
reset - disable all kernel tracing and clear the trace buffers
report - read out the trace stored in a trace.dat file
stream - Start tracing and read the output directly
profile - Start profiling and read the output directly
hist - show a histogram of the trace.dat information
stat - show the status of the running tracing (ftrace) system
split - parse a trace.dat file into smaller file(s)
options - list the plugin options available for trace-cmd report
listen - listen on a network socket for trace clients
list - list the available events, plugins or options
restore - restore a crashed record
snapshot - take snapshot of running trace
```

Trace-cmd

For example, the command below will start a function tracing:

```
# trace-cmd start -p function
```

And the command below will trace GPIO events:

```
# trace-cmd start -e gpio
```

We can also trace a specific process and record the tracing data to a file:

```
# trace-cmd record -p function -F ls /
  plugin 'function'
CPU0 data recorded at offset=0x30d000
  737280 bytes in size
CPU1 data recorded at offset=0x3c1000
  0 bytes in size

# ls trace.dat
trace.dat
```

Trace-cmd

We can show the tracing data with the *report* command:

```
# trace-cmd report
CPU 1 is empty
cpus=2
    ls-175 [000] 43.359618: function:
    ls-175 [000] 43.359624: function:
    ls-175 [000] 43.359625: function:
    ls-175 [000] 43.359627: function:
                                         mutex_unlock <-- rb_simple_write
                                         __fsnotify_parent <-- vfs_write
                                         fsnotify <-- vfs_write
                                         __sb_end_write <-- vfs_write
```

Or we can open it with **KernelShark**.

KernelShark

[KernelShark](#) is a graphical tool that works as a frontend to the *trace.dat* file generated by the *trace-cmd* tool.

Address Sanitizer

AddressSanitizer (ASan) is an instrumentation tool created by Google security researchers to identify memory access problems in C and C++ programs.

When the source code of a C/C++ application is compiled with AddressSanitizer enabled, the program will be instrumented at runtime to identify and report memory access errors.

Address Sanitizer

Memory access errors and AddressSanitizer

C and C++ are very insecure and error-prone languages. And one of the main sources of problems is memory access errors.

Different kind of bugs in the source code could trigger a memory access error, including:

- **Buffer overflow or buffer overrun** occurs when a program overruns a buffer's boundary and overwrites adjacent memory locations.
- **Stack overflow** is when a program crosses the boundary of function's stack.

Address Sanitizer

- **Heap overflow** is when a program overruns a buffer allocated in the heap.
- **Memory leak** is when a program allocates memory but does not deallocate.
- **Use after free (dangling pointer)** is when a program uses memory regions already deallocated.
- **Uninitialized variable** is when a program reads a memory location before it is initialized.

All these errors are due to programming bugs. They could prevent the application from executing, cause invalid results or expose a vulnerability that could be exploited by a malicious actor. They are usually very hard to reproduce, debug and fix.

Address Sanitizer

- **Heap overflow** is when a program overruns a buffer allocated in the heap.
- **Memory leak** is when a program allocates memory but does not deallocate.
- **Use after free (dangling pointer)** is when a program uses memory regions already deallocated.
- **Uninitialized variable** is when a program reads a memory location before it is initialized.

All these errors are due to programming bugs. They could prevent the application from executing, cause invalid results or expose a vulnerability that could be exploited by a malicious actor. They are usually very hard to reproduce, debug and fix.

Address Sanitizer

-fsanitize=address

Thread Sanitizer

ThreadSanitizer is a tool that detects data races. It consists of a compiler instrumentation module and a run-time library. Typical slowdown introduced by ThreadSanitizer is about 5x-15x. Typical memory overhead introduced by ThreadSanitizer is about 5x-10x.

-fsanitize=thread

Valgrind

Valgrind is a multipurpose code profiling and memory debugging tool for Linux when on the x86 and, as of version 3, AMD64, architectures. It allows you to run your program in Valgrind's own environment that monitors memory usage such as calls to malloc and free (or new and delete in C++). If you use uninitialized memory, write off the end of an array, or forget to free a pointer, Valgrind can detect it. Since these are particularly common problems, this tutorial will focus mainly on using Valgrind to find these types of simple memory problems, though Valgrind is a tool that can do a lot more.

Valgrind

```
# valgrind --tool=memcheck --leak-check=yes  
--show-reachable=yes --num-callers=20 ./test
```

Valgrind

Connecting GDB to a valgrind GDB Server

```
# valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --num-callers=20 --vgdb=yes --vgdb-error=0 ./val5
```

```
# gdb ./val5
```

```
(gdb) target remote | vgdb
```

Valgrind

Cachegrind

Cachegrind simulates how your program interacts with a machine's cache hierarchy and (optionally) branch predictor. It simulates a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). This exactly matches the configuration of many modern machines.

```
# valgrind --tool=cachegrind prog
```

Valgrind

Manually specifies the I1/D1/L2 cache configuration, where size and line_size are measured in bytes. The three items must be comma-separated, but with no spaces, eg:

```
# valgrind --tool=cachegrind --I1=65536,2,64
```

You can specify one, two or three of the I1/D1/L2 caches. Any level not manually specified will be simulated using the configuration found in the normal way (via the CPUID instruction for automagic cache configuration, or failing that, via defaults).

Valgrind

Cache-simulation specific options are:

--I1=<size>,<associativity>,<line size>

Specify the size, associativity and line size of the level 1 instruction cache.

--D1=<size>,<associativity>,<line size>

Specify the size, associativity and line size of the level 1 data cache.

--L2=<size>,<associativity>,<line size>

Specify the size, associativity and line size of the level 2 cache

Valgrind

Callgrind

valgrind Callgrind is a program that can profile your code and report on its resources usage. It is another tool provided by Valgrind, which also helps detect memory issues.

```
# valgrind --tool=callgrind program-to-run program-arguments
```

```
# callgrind_annotate --auto=yes callgrind.out.pid
```

Valgrind

helgrind

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives.

The main abstractions in POSIX pthreads are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

Valgrind

helgrind

Helgrind can detect three classes of errors, which are discussed in detail in the next three sections:

- Misuses of the POSIX pthreads API.
- Potential deadlocks arising from lock ordering problems.
- Data races -- accessing memory without adequate locking or synchronisation.

```
# valgrind --tool=helgrind program_name
```

Valgrind

massif

Massif is a heap profiler. It measures how much heap memory your program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program's stack(s), although it does not do so by default.

Valgrind

massif

Heap profiling can help you reduce the amount of memory your program uses. On modern machines with virtual memory, this provides the following benefits:

- It can speed up your program -- a smaller program will interact better with your machine's caches and avoid paging.
- If your program uses lots of memory, it will reduce the chance that it exhausts your machine's swap space.

```
# valgrind --tool=massif prog
```

```
# ms_print massif.out.pid
```

gdb (GNU Debugger)

Debuggers are programs which allow you to execute your program in a controlled manner, so you can look inside your program to find a bug.

gdb is a reasonably sophisticated text based debugger. It can let you:

- Start your program, specifying anything that might affect its behavior.

- Make your program stop on specified conditions.

- Examine what has happened, when your program has stopped.

- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

SYNOPSIS

gdb [prog] [core|procID]

gdb

GDB is invoked with the shell command `gdb`.

Once started, it reads commands from the terminal until you tell it to exit with the GDB command `quit`.

The most usual way to start GDB is with one argument or two, specifying an executable program as the argument:

gdb program

You can also start with both an executable program and a core file specified:

gdb program core

You can, instead, specify a process ID as a second argument, if you want to debug a running process:

gdb program 1234

would attach GDB to process 1234

Compiling with the -g Option

To use gdb best, compile your program with:

```
gcc -g -c my_math.c
```

```
gcc -g -c sample.c
```

```
gcc -o sample my_math.o sample.o
```

or:

```
gcc -o sample -g my_math.c sample.c
```

That is, you should make sure that -g option is used to generate the .o files.

This option tells the compiler to insert more information about data types, etc., so the debugger gets a better understanding of it.

Common Commands for gdb

Here are some of the most frequently needed GDB commands:

b(reak) [file:]function Set a breakpoint at function (in file).

r(un) [arglist] Start program (with arglist, if specified).

bt or where especially Backtrace: display the program stack; especially useful to find where your program crashed or dumped core.

print expr Display the value of an expression.

c Continue running your program (after stopping, e.g. at a breakpoint).

n(ext) Execute next program line (after stopping); step over any function calls in the line.

Common Commands for gdb

- s(tep) Execute next program line (after stopping); step into any function calls in the line.
- help [name] Show information about GDB command name, or general information about using GDB.
- q(uit) Exit from GDB.
- l(ist) print the source code

Gdb pass comma line arguments

```
# gdb --args executablename arg1 arg2 arg3
```

Or

```
$ gdb executablename  
(gdb) r arg1 arg2 arg3
```

Debugging fork

set follow-fork-mode mode

Set the debugger response to a program call of fork or vfork. A call to fork or vfork creates a new process. The mode argument can be:

parent

The original process is debugged after a fork. The child process runs unimpeded. This is the default.

child

The new process is debugged after a fork. The parent process runs unimpeded.

Debugging fork

set detach-on-fork mode

Tells gdb whether to detach one of the processes after a fork, or retain debugger control over them both.

on

The child process (or parent process, depending on the value of follow-fork-mode) will be detached and allowed to run independently. This is the default.

off

Both processes will be held under the control of GDB. One process (child or parent, depending on the value of follow-fork-mode) is debugged as usual, while the other is held suspended.

Debugging fork

(gdb) info inferior

(gdb) inferior

Debugging exec

set follow-exec-mode mode

Set debugger response to a program call of exec. An exec call replaces the program image of a process.

follow-exec-mode can be:

new

GDB creates a new inferior and rebinds the process to this new inferior. The program the process was running before the exec call can be restarted afterwards by restarting the original inferior.

Debugging exec

same

GDB keeps the process bound to the same inferior. The new executable image replaces the previous executable loaded in the inferior. Restarting the inferior after the exec call, with e.g., the run command, restarts the executable the process was running after the exec call. This is the default mode.

Thread debugging

(gdb) info thread

(gdb) thread

Core dump debugging

(gdb) info thread

(gdb) thread

Variable manipulation

(gdb) print variable

(gdb) print func::variable

print variable in different format

x

Regard the bits of the value as an integer, and print the integer in hexadecimal.

d

Print as integer in signed decimal.

u

Print as integer in unsigned decimal.

Variable manipulation

o

Print as integer in octal.

t

Print as integer in binary. The letter `t' stands for "two". (1)

a

Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

(gdb) p/a 0x54320

\$3 = 0x54320 <_initialize_vx+396>

Variable manipulation

c

Regard as an integer and print it as a character constant.

f

Regard the bits of the value as a floating point number and print using typical floating point syntax.

Variable manipulation

```
(gdb) set variable=90
```

```
(gdb) print &a
```

Type **info variables** to list "All global and static variable names".

Type **info locals** to list "Local variables of current stack frame" (names and values), including static variables in that function.

Type **info args** to list "Arguments of the current stack frame" (names and values).

Conditional break point

(gdb) condition <breakpoint> <condition>

Example:

(gdb) break 28

(gdb) condition 2 i>5

- **watch**: gdb will break when a *write* occurs
- **rwatch**: gdb will break when a *read* occurs
- **awatch**: gdb will break in *both cases*

Ptrace

ptrace is a system call found in Unix and several Unix-like operating systems. By using ptrace (the name is an abbreviation of "process trace") one process can control another, enabling the controller to inspect and manipulate the internal state of its target. ptrace is used by debuggers and other code-analysis tools, mostly as aids to software development.

Ptrace

ptrace is used by debuggers (such as gdb and dbx), by tracing tools like strace and ltrace, and by code coverage tools. ptrace is also used by specialized programs to patch running programs, to avoid unfixed bugs or to overcome security features. It can further be used as a sandbox and as a run-time environment simulator (like emulating root access for non-root software).

Ptrace

By attaching to another process using the ptrace call, a tool has extensive control over the operation of its target. This includes manipulation of its file descriptors, memory, and registers. It can single-step through the target's code, can observe and intercept system calls and their results, and can manipulate the target's signal handlers and both receive and send signals on its behalf. The ability to write into the target's memory allows not only its data store to be changed, but also the application's own code segment, allowing the controller to install breakpoints and patch the running code of the target.

Ptrace

By attaching to another process using the ptrace call, a tool has extensive control over the operation of its target. This includes manipulation of its file descriptors, memory, and registers. It can single-step through the target's code, can observe and intercept system calls and their results, and can manipulate the target's signal handlers and both receive and send signals on its behalf. The ability to write into the target's memory allows not only its data store to be changed, but also the application's own code segment, allowing the controller to install breakpoints and patch the running code of the target.

Introduction to Bash Shell

BASH Scripting

A beginners guide to Bourne Again SHell scripting

What is Shell?

The shell is a command interpreter.

It is the layer between the operating system kernel and the user.

BASH script header

- On the first line
 - `#!/bin/bash`
 - Tells the operating system that the following will be a script and not a regular text file
 - You add - to the end to prevent users from spoofing the interpreter
- Change the file to give it execute permissions
 - `chmod +x test.sh`

How scripts work

```
#!/bin/bash  
echo "Hello"  
ls -al
```

- /bin/bash echo "Hello"
- /bin/bash ls -al

Comments in Bash Script

'#' symbol is used to add single line comment in bash script.

```
#!/bin/bash  
  
# Add two numeric value  
((sum=25+35))  
  
#Print the result  
echo $sum
```

You can use multi line comment in bash in various ways. Use : ' to open and ' to close.

```
#!/bin/bash  
:  
The following script calculates  
the square value of the number, 5.  
:  
((area=5*5))  
echo $area
```

Printing in Bash Script - echo

You can use echo command with various options.

```
echo [ OPTION(S) ] [ STRING ]
```

Options:

Option	Description
-n	Omit newline from the output.
-e	Enable the function of backslash(/) character.
-E	Disable the function of backslash(/) character.
--version	Display the version information
--help	Display help messages related to the uses of this command

Printing in Bash Script - echo

When you use ‘echo’ command without any option then a newline is added by default.

‘-n’ option is used to print any text without new line and

‘-e’ option is used to remove backslash characters from the output.

```
#!/bin/bash
echo "Printing text with newline"
echo -n "Printing text without newline"
echo -e "\nRemoving \t backslash \t characters\n"
```

Printing in Bash Script - echo

‘echo’ command is used with ‘-E’ option in the following script. This option disables the function of backslash(/). The new line(\n) used in the text will not work for the following command.‘echo’ command is used with ‘-E’ option in the following script. This option disables the function of backslash(/). The new line(\n) used in the text will not work for the following command.

```
$ echo -E "I\nlike\nbash\programming"
```

Printing in Bash Script - printf

The “printf” command in bash scripting works the same way “printf ()” works in the C language.

```
printf <var> <formate> <arguments...>
```

<var> : It is optional to assign output to a variable.

<formate> : It is a string that may contain different format specifiers such as “%s”, “%b”, “%d”, “%f”, and backslash escape

<arguments...> : The arguments can be any value or variable

Specifier	Description
%%	Prints “%” symbol
%c	Takes arguments as a single character
%e and %E	Take argument in floating-point number and prints in exponential notation, %e for lower case letter and %E for capital letter
%g and %G	Take argument in floating-point number and prints in normal or exponential notation
%f	Takes argument as floating numbers
%d	Takes arguments as signed integers
%u	Takes argument as unsigned integers
%o	Takes argument as an unsigned octal number
%x and %X	Takes arguments as unsigned hexadecimal integers

Bash Read data from input

Use to get input (data from user) from keyboard and store (data) to variable.

Syntax:

read variable1, variable2,...variable

```
#  
#Script to read your name from key-board  
#  
echo "Your first name please:"  
read fname  
echo "Hello $fname, Lets be friend!"
```

Bash Read data from input

To catch a line of input NAMES and options are not required by read. When NAME is not specified, a variable named REPLY is used to store user input.

```
{  
echo -n "Type something and press enter: ";  
read;  
echo You typed ${REPLY}  
}
```

To catch a word of input, the -d option is required. In the case of a word we would set -d to a space, read ‘-d ‘. That is when the user presses the space bar read will load REPLY with the word.

```
{  
echo -n "Type something and hit space: ";  
read '-d '';  
echo "";  
echo "You typed ${REPLY}"  
}
```

Bash Read data from input

The -p option allows a message to be displayed before reading from standard input.

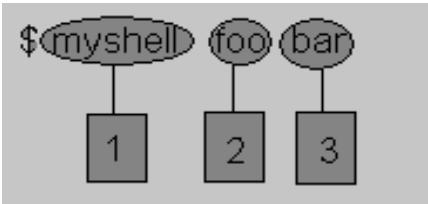
```
{  
read -p 'Type something and hit space: ' -d ';  
echo "";  
echo "You typed ${REPLY}"  
}
```

When catching user input without it showing up in the terminal, the -s option comes in handy. read -s -p allows you to catch and hide user input

```
{  
read -s -p 'Type something I promise to keep it a secret: '  
echo "";  
echo "Your secret is safe with me" ; unset REPLY ;  
echo "${REPLY}"  
}
```

Arguments - Specification

\$ myshell foo bar



[1] Shell Script name i.e. **myshell**

[2] First command line argument passed to **myshell** i.e. **foo**

[3] Second command line argument passed to **myshell** i.e. **bar**

In shell if we wish to refer this command line argument we refer above as follows

1	myshell	it	is	\$0
2	foo	it	is	\$1
3	bar	it is \$2		

- Here \${#} (built in shell variable) will be 2 (Since **foo** and **bar** only two Arguments),
- Please note at a time such 9 arguments can be used from \${1..\$9},
- You can also refer all of them by using \${*} (which expand to ` \${1..\$9}`).
- Note that \${1..\$9} i.e command line arguments to shell script is known as "*positional parameters*".

Arguments - Example

```
#!/bin/bash
#
# Script that demos, command line args
#
echo "Total number of command line argument are $#"
echo "$0 is script name"
echo "$1 is first argument"
echo "$2 is second argument"
echo "All of them are :- $* or $@"
```

Redirection of Input/Output

In Linux (and in other OSs also)

it's possible to send output to file

or to read input from a file

For e.g.

\$ *ls* command gives output to screen;

to send output to file of *ls* command give command

ls > filename

It means put output of *ls* command to filename.

redirection symbols: ‘>’

There are three main redirection symbols: >, >>, <

(1) > Redirector Symbol

Syntax:

Linux-command > filename

To output Linux-commands result to file.

Note that if the file already exist,

it will be overwritten

else a new file will be created.

For e.g.

To send output of **ls** command give

\$ ls > myfiles

if '**myfiles**' exist in your current directory

it will be overwritten without any warning.

redirection symbols: ‘>>’

(2) >> Redirector Symbol

Syntax:

Linux-command >> filename

To output Linux-commands result

to the END of the file.

if file exist:

it will be opened

new information/data will be written to the **END** of the file,

without losing previous information/data,

if file does not exist, a new file is created.

For e.g.

To send output of date command

to already exist file give command

\$ date >> myfiles

redirection symbols: ‘<’

(3) < Redirector Symbol

Syntax:

Linux-command < filename

To provide input to Linux-command

from the file instead of standard input (key-board).

For e.g. To take input for cat command give

\$ cat < myfiles

Pipes

A pipe is a way to connect the output of one program to the input of another program without any temporary file.

Definition

"A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands Multiple commands from same command line."

Syntax:

command1 | command2

Pipe - Examples

Command using Pipes	Meaning or Use of Pipes
\$ ls more	Output of ls command is given as input to the command more So output is printed one screen full page at a time.
\$ who sort	Output of who command is given as input to sort command So it will print sorted list of users
\$ who sort > user_list	Same as above except output of sort is send to (redirected) the file named user_list
\$ who wc -l	who command provides the input of wc command So it will count the users logged in.
\$ ls -l wc -l	ls command provides the input of wc command So it will count files in current directory.
\$ who grep raju	Output of who command is given as input to grep command So it will print if particular user is logged in. Otherwise nothing is printed

if condition

A basic if statement effectively says, **if** a particular test is true, then perform a given set of actions. If it is not true then don't perform those actions. It follows the format below:

```
if [ <some test> ]
then
  <commands>
fi
```

Anything between **then** and **fi** (if backwards) will be executed only if the test (between the square brackets) is true.

Mathematical Operators

Mathematical Operator in Shell Script	Meaning	Normal Arithmetical/Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	<code>5 == 6</code>	<code>if test 5 -eq 6</code>	<code>if [5 -eq 6]</code>
-ne	is not equal to	<code>5 != 6</code>	<code>if test 5 -ne 6</code>	<code>if [5 -ne 6]</code>
-lt	is less than	<code>5 < 6</code>	<code>if test 5 -lt 6</code>	<code>if [5 -lt 6]</code>
-le	is less than or equal to	<code>5 <= 6</code>	<code>if test 5 -le 6</code>	<code>if [5 -le 6]</code>
-gt	is greater than	<code>5 > 6</code>	<code>if test 5 -gt 6</code>	<code>if [5 -gt 6]</code>
-ge	is greater than or equal to	<code>5 >= 6</code>	<code>if test 5 -ge 6</code>	<code>if [5 -ge 6]</code>

String Operators

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

File and Directory Operators

Test	Meaning
-s file	Non empty file
-f file	File exists or is a normal file and not a directory
-d dir	Directory exists and not a file
-w file	file is a writeable file
-r file	file is a read-only file
-x file	file is executable

Logical Operators

Operator	Meaning
<code>! expression</code>	Logical NOT
<code>expression1 -a expression2</code>	Logical AND
<code>expression1 -o expression2</code>	Logical OR

if...else...fi

Sometimes we want to perform a certain set of actions if a statement is true, and another set of actions if it is false. We can accommodate this with the **else** mechanism.

```
if [ <some test> ]
then
    <commands>
else
    <other commands>
fi
```

if...elif...fi

Sometimes we may have a series of conditions that may lead to different paths.

```
if [ <some test> ]
then
    <commands>
elif [ <some test> ]
then
    <different commands>
else
    <other commands>
fi
```

case

Sometimes we may wish to take different paths based upon a variable matching a series of patterns. We could use a series of **if** and **elif** statements but that would soon grow to be unwieldy. Fortunately there is a **case** statement which can make things cleaner. It's a little hard to explain so here are some examples to illustrate:

```
case <variable> in
<pattern 1>
  <commands>
  ;;
<pattern 2>
  <other commands>
  ;;
esac
```

Loops in Shell Scripts

Bash supports:

for loop

while loop

Note that in each and every loop,

(a) First, the variable used in loop condition

- must be initialized,
- then execution of the loop begins.

(b) A test (condition) is made

at the beginning of each iteration.

(c) The body of loop ends

with a statement modifies

the value of the test (condition) variable.

for Loop

Syntax:

for { variable name } in { list }

do

execute one for each item in the list

until the list is not finished

(And repeat all statements between do and done)

done

while Loop

Syntax:

while [condition]

do

command1

command2

..

....

done

until Loop

The **until** loop is fairly similar to the while loop. The difference is that it will execute the commands within it until the test becomes true.

```
until [ <some test> ]
do
  <commands>
done
```

Break

The **break** statement tells Bash to leave the loop straight away. It may be that there is a normal situation that should cause the loop to end but there are also exceptional situations in which it should end as well. For instance, maybe we are copying files but if the free disk space get's below a certain level we should stop copying.

Continue

The **continue** statement tells Bash to stop running through this iteration of the loop and begin the next iteration. Sometimes there are circumstances that stop us from going any further. For instance, maybe we are using the loop to process a series of files but if we happen upon a file which we don't have the read permission for we should not try to process it.

Bash Function

A Bash function is essentially a set of commands that can be called numerous times. The purpose of a function is to help you make your bash scripts more readable and to avoid writing the same code repeatedly. Compared to most programming languages, Bash functions are somewhat limited.

The first format starts with the function name, followed by parentheses. This is the preferred and more used format.

```
function_name () {  
    commands  
}
```

Bash Function

The second format starts with the reserved word function, followed by the function name.

```
function function_name {  
    commands  
}
```

Function Variables

Global variables are variables that can be accessed from anywhere in the script regardless of the scope. In Bash, all variables by default are defined as global, even if declared inside the function.

Local variables can be declared within the function body with the local keyword and can be used only inside that function. You can have local variables with the same name in different functions.

Function Return value

Unlike functions in “real” programming languages, Bash functions don’t allow you to return a value when called. When a bash function completes, its return value is the status of the last statement executed in the function, 0 for success and non-zero decimal number in the 1 - 255 range for failure.

The return status can be specified by using the `return` keyword, and it is assigned to the variable `$?`. The `return` statement terminates the function. You can think of it as the function’s exit status .

Function Return value

To actually return an arbitrary value from a function, we need to use other methods. The simplest option is to assign the result of the function to a global variable.

Function Return value

Another, better option to return a value from a function is to send the value to stdout using echo or printf.

Function Arguments

To pass any number of arguments to the bash function simply put them right after the function's name, separated by a space. It is a good practice to double-quote the arguments to avoid the misparsing of an argument with spaces in it.

- The passed parameters are \$1, \$2, \$3 ... \$n, corresponding to the position of the parameter after the function's name.
- The \$0 variable is reserved for the function's name.
- The \$# variable holds the number of positional parameters/arguments passed to the function.
- The \$* and \$@ variables hold all positional parameters/arguments passed to the function.

SED

SED command in UNIX is stands for stream editor and it can perform lot's of function on file like, searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening it, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.

- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- SED command in unix supports regular expression which allows it perform complex pattern matching.

SED

SED command in UNIX is stands for stream editor and it can perform lot's of function on file like, searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening it, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.

- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- SED command in unix supports regular expression which allows it perform complex pattern matching.

Syntax:

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

SED

Replacing or substituting string : Sed command is mostly used to replace the text in a file. The below simple sed command replaces the word “unix” with “linux” in the file.

```
$ sed 's/unix/linux/' sample.txt
```

Here the “s” specifies the substitution operation. The “/” are delimiters. The “unix” is the search pattern and the “linux” is the replacement string.

By default, the sed command replaces the first occurrence of the pattern in each line and it won’t replace the second, third...occurrence in the line.

SED

Replacing the nth occurrence of a pattern in a line : Use the /1, /2 etc flags to replace the first, second occurrence of a pattern in a line. The below command replaces the second occurrence of the word “unix” with “linux” in a line.

```
$ sed 's/unix/linux/2' sample.txt
```

SED

Replacing all the occurrence of the pattern in a line : The substitute flag /g (global replacement) specifies the sed command to replace all the occurrences of the string in the line.

```
$ sed 's/unix/linux/g' sample.txt
```

SED

Replacing from nth occurrence to all occurrences in a line : Use the combination of /1, /2 etc and /g to replace all the patterns from the nth occurrence of a pattern in a line. The following sed command replaces the third, fourth, fifth... “unix” word with “linux” word in a line.

```
$ sed 's/unix/linux/3g' sample.txt
```

SED

Parenthesize first character of each word : This sed example prints the first character of every word in parenthesis.

```
$ echo "Welcome To The Geek Stuff" | sed 's/\b[A-Z]\b/\(\1\)/g'
```

SED

Replacing string on a specific line number : You can restrict the sed command to replace the string on a specific line number. An example is

```
$ sed '3 s/unix/linux/' sample.txt
```

SED

Deleting lines from a particular file : SED command can also be used for deleting lines from a particular file. SED command is used for performing deletion operation without even opening the file

Examples:

1. To Delete a particular line say n in this example

Syntax:

```
$ sed 'nd' filename.txt
```

Example:

```
$ sed '5d' filename.txt
```

SED

2. To Delete a last line

Syntax:

\$ sed '\$d' filename.txt

3. To Delete line from range x to y

Syntax:

\$ sed 'x,yd' filename.txt

Example:

\$ sed '3,6d' filename.txt

SED

5. To Delete from nth to last line

Syntax:

\$ sed 'nth,\$d' filename.txt

Example:

\$ sed '12,\$d' filename.txt

6. To Delete pattern matching line

Syntax:

\$ sed '/pattern/d' filename.txt

Example:

\$ sed '/abc/d' filename.txt

SED

Here, to delete three lines, we have specified three separate commands with -e option.

```
# sed -e '1d' -e '2d' -e '5d' books.txt
```

```
# sed -n '  
h;n;H;x  
s/\n/, /  
/Paulo/!b Print  
s/^/- /  
:Print  
p' books.txt
```

SED

The essential command: s for substitution

Sed has several commands, but most people only learn the substitute command: s. The substitute command changes all occurrences of the regular expression into a new value. A simple example is changing "day" in the "old" file to "night" in the "new" file:

```
# echo "day day day" | sed 's/day/night/'  
# echo "day day day" | sed 's/day/night/g'
```

SED

```
# echo Sunday | sed 's/day/night/'
```

Delete a file line starting with & ending with a pattern

To delete a line starting with a particular string & ending with another string, use

```
# sed -e 's/^danger.*stops$/g' testfile.txt
```

This will delete the line with ‘danger’ on start & ‘stops’ in the end & it can have any number of words in between , ‘.*’ defines that part.

SED

The slash as a delimiter

The character after the s is the delimiter. It is conventionally a slash, because this is what ed, more, and vi use. It can be anything you want, however. If you want to change a pathname that contains a slash - say /usr/local/bin to /common/bin - you could use the backslash to quote the slash:

```
# sed 's/>\usr\local\bin/>\common\bin/' <old>new
```

```
# sed 's/_/usr/local/bin/_/common/bin_-' <old>new  
# sed 's:/usr/local/bin:/common/bin:' <old>new  
# sed 's|\usr\local\bin||\common\bin|' <old>new
```

SED

Using & as the matched string

Sometimes you want to search for a pattern and add some characters, like parenthesis, around or near the pattern you found. It is easy to do this if you are looking for a particular string:

```
# sed 's/abc/(abc)/* <old>new
```

```
# sed 's/[a-z]*/(&)* <old>new
```

```
# echo "123 abc" | sed 's/[0-9]*/& &/'
```

```
# echo "123 abc" | sed 's/[0-9][0-9]*/& &/'
```

SED

Using \1 to keep part of the pattern

The escaped parentheses (that is, parentheses with backslashes before them) remember a substring of the characters matched by the regular expression. You can use this to exclude part of the characters matched by the regular expression. The "\1" is the first remembered pattern, and the "\2" is the second remembered pattern. Sed has up to nine remembered patterns.

If you wanted to keep the first word of a line, and delete the rest of the line, mark the important part with the parenthesis:

```
# sed 's/([a-z]*).*/\1/'
```

"[a-z]*" matches zero or more lower case letters, and tries to match as many characters as possible. The ".*" matches zero or more characters after the first match. Since the first one grabs all of the contiguous lower case letters, the second matches anything else.

```
# echo abcd123 | sed 's/([a-z]*).*/\1/'
```

SED

Using \1 to keep part of the pattern

The escaped parentheses (that is, parentheses with backslashes before them) remember a substring of the characters matched by the regular expression. You can use this to exclude part of the characters matched by the regular expression. The "\1" is the first remembered pattern, and the "\2" is the second remembered pattern. Sed has up to nine remembered patterns.

If you wanted to keep the first word of a line, and delete the rest of the line, mark the important part with the parenthesis:

```
# sed 's/([a-z]*).*/\1/'
```

"[a-z]*" matches zero or more lower case letters, and tries to match as many characters as possible. The ".*" matches zero or more characters after the first match. Since the first one grabs all of the contiguous lower case letters, the second matches anything else.

```
# echo abcd123 | sed 's/([a-z]*).*/\1/'
```

SED

```
# sed 's/[^\n]*(&)/' <old >new
```

```
# sed 's/[^:]*//2' </etc/passwd
```

SED

/p - print

By default, sed prints every line. If it makes a substitution, the new text is printed instead of the old one. If you use an optional argument to sed, "sed -n," it will not, by default, print any new lines. I'll cover this and other options later. When the "-n" option is used, the "p" flag will cause the modified line to be printed. Here is one way to duplicate the function of grep with sed:

```
# sed -n 's/pattern/&/p' <file
```

SED

The sed has two buffers, the pattern space and the hold space. Both are initially empty. The following commands manipulate the hold space:

command description

- h Replace the contents of the hold space with the contents of the pattern space.
- H Append a newline to the contents of the hold space, and then append the contents of the pattern space to that of the hold space.
- g Replace the contents of the pattern space with the contents of the hold space.
- G Append a newline to the contents of the pattern space, and then append the contents of the hold space to that of the pattern space.
- x Exchange the contents of the hold and pattern spaces.

SED

```
# printf '%s\n' one two | sed --debug 'H;1h;$x;$s/\n/_/g'
```

AWK

Awk is a scripting language used for manipulating data and generating reports. The awk command programming language requires no compiling, and allows the user to use variables, numeric functions, string functions, and logical operators.

Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line.

Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then performs the associated actions.

AWK

Awk is abbreviated from the names of the developers – Aho, Weinberger, and Kernighan.

- AWK Operations:
 - A. Scans a file line by line
 - B. Splits each input line into fields
 - C. Compares input line/fields to pattern
 - D. Performs action(s) on matched lines
- Useful For:
 - A. Transform data files
 - B. Produce formatted reports

AWK

- Programming Constructs:
 - A. Format output lines
 - B. Arithmetic and string operations
 - C. Conditionals and loops

```
$ awk '{print}' employee.txt
```

Print the lines which matches with the given pattern.

```
$ awk '/manager/ {print}' employee.txt
```

Splitting a Line Into Fields

```
$ awk '{print $1,$4}' employee.txt
```

AWK

Built In Variables In Awk

Awk's built-in variables include the field variables—\$1, \$2, \$3, and so on (\$0 is the entire line) — that break a line of text into individual words or pieces called fields.

NR: **NR** command keeps a current count of the number of input records. Remember that records are usually lines. Awk command performs the pattern/action statements once for each record in a file.

NF: **NF** command keeps a count of the number of fields within the current input record.

AWK

FS: FS command contains the field separator character which is used to divide fields on the input line. The default is “white space”, meaning space and tab characters. FS can be reassigned to another character (typically in BEGIN) to change the field separator.

RS: RS command stores the current record separator character. Since, by default, an input line is the input record, the default record separator character is a newline.

OFS: OFS command stores the output field separator, which separates the fields when Awk prints them. The default is a blank space. Whenever print has several parameters separated with commas, it will print the value of OFS in between each parameter.

AWK

ORS: **ORS** command stores the output record separator, which separates the output lines when Awk prints them. The default is a newline character. **print** automatically outputs the contents of **ORS** at the end of whatever it is given to print.

Use of **NR** built-in variables (Display Line Number)

```
$ awk '{print NR,$0}' employee.txt
```

Use of **NF** built-in variables (Display Last Field)

```
$ awk '{print $1,$NF}' employee.txt
```

AWK

```
$ awk 'BEGIN {print NR;} {print NR;} END {print NR;}'  
employee.txt
```

GNU Autotools

Kbuild