

Shell

A shell is actually how you are going to be interacting with the system. Before user-friendly operating systems, when a computer started up all you had access to was a shell. This meant that all of your commands and editing had to be done this way. Nowadays, our computers boot up in desktop mode, but one can still access a shell using a terminal.

```
(Stuff) $
```

It is ready for your next command! You can type in a lot of Unix utilities like ls, echo Hello and the shell will execute them and give you the result. Some of these are what are known as shell-builtins meaning that the code is in the shell program itself. Some of these are compiled programs that you run. The shell only looks through a special variable called path which contains a list of colon separated paths to search for an executable with your name, here is an example path.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:
/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

So when the shell executes ls, it looks through all of those directories, finds /bin/ls and executes that.

```
$ ls
...
$ /bin/ls
```

You can always call through the full path. That is always why in past classes if you want to run something on

the terminal you've had to do `./exe` because typically the directory that you are working in is not in the `PATH` variable. The `.` expands to your current directory and your shell executes `<current_dir>/exe` which is a valid command.

Shell tricks and tips

- The up arrow will get you your most recent command
- `ctrl-r` will search commands that you previously ran
- `ctrl-c` will interrupt your shell's process
- `!!` will execute the last command
- `!<num>` goes back that many commands and runs that
- `!<prefix>` runs the last command that has that prefix
- `!$` is the last arg of the previous command
- `!*` is all args of the previous command
- `patsub` takes the last command and substitutes the pattern `pat` for the substitution `sub`
- `cd -` goes to the previous directory
- `pushd <dir>` pushes the current directory on a stack and cds
- `popd` cds to the directory at the top of the stack

What's a terminal?

A terminal is an application that displays the output from the shell. You can have your default terminal, a quake based terminal, terminator, the options are endless!

Common Utilities

1. `cat` concatenate multiple files. It is regularly used to print out the contents of a file to the terminal but the original use was concatenation.

```
$ cat file.txt
...
$ cat shakespeare.txt shakespeare.txt > two_shakes.txt
```

2. `diff` tells you the difference between the two files. If nothing is printed, then zero is returned meaning the files are the same byte for byte. Otherwise, the longest common subsequence difference is printed

```
$ cat prog.txt
hello
world
$ cat adele.txt
hello
it's me
$ diff prog.txt prog.txt
$ diff shakespeare.txt shakespeare.txt
2c2
< world
---
> it's me
```

3. grep tells you which lines in a file or standard input match a POSIX pattern.

```
$ grep it adele.txt
it's me
```

4. ls tells you which files are in the current directory.
5. cd this is a shell builtin but it changes to a relative or absolute directory

```
$ cd /usr
$ cd lib/
$ cd -
$ pwd
/usr/
```

6. man every system programmers favorite command tells you more about all your favorite functions!
7. make executes programs according to a makefile.

Syntactic

Shells have many useful utilities like saving some output to a file using redirection \geq . This overwrites the file from the beginning. If you only meant to append to the file, you can use \gg . Unix also allows file descriptor swapping. This means that you can take the output going to one file descriptor and make it seem like it's coming out of another. The most common one is $2>1$ which means take the stderr and make it seem like it is coming out of standard out. This is important because when you use \geq and \gg they only write the standard output of the file. There are some examples below.

```
$ ./program > output.txt # To overwrite
$ ./program >> output.txt # To append
$ ./program 2>&1 > output_all.txt # stderr & stdout
$ ./program 2>&1 > /dev/null # don't care about any output
```

The pipe operator has a fascinating history. The UNIX philosophy is writing small programs and chaining them together to do new and interesting things. Back in the early days, hard disk space was limited and write times were slow. Brian Kernighan wanted to maintain the philosophy while omitting intermediate files that take up hard drive space. So, the UNIX pipe was born. A pipe takes the stdout of the program on its left and feeds it to the stdin of the program on its right. Consider the command tee. It can be used as a replacement for the redirection operators because tee will both write to a file and output to standard out. It also has the added benefit that it doesn't need to be the last command in the list. Meaning, that you can write an intermediate result and continue your piping.

```
$ ./program | tee output.txt # Overwrite
$ ./program | tee -a output.txt # Append
$ head output.txt | wc | head -n 1 # Multi pipes
$ ((head output.txt) | wc) | head -n 1 # Same as above
$ ./program | tee intermediate.txt | wc
```

The `and` `||` operator are operators that execute a command sequentially. `only` executes a command if the previous command succeeds, and `||` always executes the next command.

```
$ false && echo "Hello!"
$ true && echo "Hello!"
$ false || echo "Hello!"
```

What are environment variables?

Each process gets its own dictionary of environment variables that are copied over to the child. Meaning, if the parent changes their environment variables it won't be transferred to the child and vice versa. This is important in the fork-exec-wait trilogy if you want to exec a program with different environment variables than your parent (or any other process).

For example, you can write a C program that loops through all of the time zones and executes the date command to print out the date and time in all locals. Environment variables are used for all sorts of programs so modifying them is important.

Stack Smashing

Each thread uses a stack memory. The stack ‘grows downwards’ - if a function calls another function, then the stack is extended to smaller memory addresses. Stack memory includes non-static automatic (temporary) variables, parameter values, and the return address. If a buffer is too small some data (e.g. input values from the user), then there is a real possibility that other stack variables and even the return address will be overwritten. The precise layout of the stack’s contents and order of the automatic variables is architecture and compiler dependent. With a little investigative work, we can learn how to deliberately smash the stack for a particular architecture.

The example below demonstrates how the return address is stored on the stack. For a particular 32 bit architecture Live Linux Machine, we determine that the return address is stored at an address two pointers (8 bytes) above the address of the automatic variable. The code deliberately changes the stack value so that when the input function returns, rather than continuing on inside the main method, it jumps to the exploit function instead.

```
// Overwrites the return address on the following machine:
// http://cs-education.github.io/sys/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void breakout() {
    puts("Welcome. Have a shell...");
    system("/bin/sh");
}

void input() {
    void *p;
    printf("Address of stack variable: %p\n", &p);
    printf("Something that looks like a return address on stack:
           %p\n", *((&p)+2));
    // Let's change it to point to the start of our sneaky function.
    *((&p)+2) = breakout;
}

int main() {
    printf("main() code starts at %p\n",main);

    input();
    while (1) {
        puts("Hello");
        sleep(1);
    }

    return 0;
}
```

There are a lot of ways that computers tend to get around this.

Assorted Man Pages

Malloc

Copyright (c) 1993 by Thomas Koenig (ig25@rz.uni-karlsruhe.de)

%%%LICENSE_START(VERBATIM)

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a. permission notice identical to this one.

Since the Linux kernel and libraries are constantly changing, this manual page may be incorrect or out-of-date. The author(s) assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. The author(s) may not have taken the same level of care in the production of this manual, which is licensed free of charge, as they might when working professionally.

Formatted or processed versions of this manual, if unaccompanied by the source, must acknowledge the copyright and authors of this work.

%%%LICENSE_END

MALLOC(3) Linux Programmer's Manual MALLOC(3)

NAME

malloc, free, calloc, realloc - allocate and free dynamic memory

SYNOPSIS

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

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
reallocarray():
    _GNU_SOURCE
```

DESCRIPTION

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().

The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(),

`calloc()`, or `realloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is `NULL`, no operation is performed.

The `calloc()` function allocates memory for an array of `nmemb` elements of size `bytes` each and returns a pointer to the allocated memory. The memory is set to zero. If `nmemb` or `size` is 0, then `calloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized. If `ptr` is `NULL`, then the call is equivalent to `malloc(size)`, for all values of `size`; if `size` is equal to zero, and `ptr` is not `NULL`, then the call is equivalent to `free(ptr)`. Unless `ptr` is `NULL`, it must have been returned by an earlier call to `malloc()`, `calloc()`, or `realloc()`. If the area pointed to was moved, a `free(ptr)` is done.

The `reallocarray()` function changes the size of the memory block pointed to by `ptr` to be large enough for an array of `nmemb` elements, each of which is `size` bytes. It is equivalent to the call

```
realloc(ptr, nmemb * size);
```

However, unlike that `realloc()` call, `reallocarray()` fails safely in the case where the multiplication would overflow. If such an overflow occurs, `reallocarray()` returns `NULL`, sets `errno` to `ENOMEM`, and leaves the original block of memory unchanged.

RETURN VALUE

The `malloc()` and `calloc()` functions return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return `NULL`. `NULL` may also be returned by a successful call to `malloc()` with a size of zero, or by a successful call to `calloc()` with `nmemb` or `size` equal to zero.

The `free()` function returns no value.

The `realloc()` function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from `ptr`, or `NULL` if the request fails. If `size` was equal to 0, either `NULL` or a pointer suitable to be passed to `free()` is returned. If `realloc()` fails, the original block is left untouched; it is not freed or moved.

On success, the `reallocarray()` function returns a pointer to the newly allocated memory. On failure, it returns `NULL` and the original block of memory is left untouched.

ERRORS

`calloc()`, `malloc()`, `realloc()`, and `reallocarray()` can fail with

the following error:

```
ENOMEM Out of memory. Possibly, the application hit the
RLIMIT_AS or RLIMIT_DATA limit described in getrlimit(2).
```

ATTRIBUTES

For an explanation of the terms used in this section, see `attributes(7)`.

Interface	Attribute	Value
<code>malloc()</code> , <code>free()</code> , <code>calloc()</code> , <code>realloc()</code>	Thread safety	MT-Safe

CONFORMING TO

`malloc()`, `free()`, `calloc()`, `realloc()`: POSIX.1-2001, POSIX.1-2008, C89, C99.

`reallocarray()` is a nonstandard extension that first appeared in OpenBSD 5.6 and FreeBSD 11.0.

NOTES

By default, Linux follows an optimistic memory allocation strategy. This means that when `malloc()` returns non-NULL there is no guarantee that the memory is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer. For more information, see the description of `/proc/sys/vm/overcommit_memory` and `/proc/sys/vm/oom_adj` in `proc(5)`, and the Linux kernel source file `Documentation/vm/overcommit-accounting`.

Normally, `malloc()` allocates memory from the heap, and adjusts the size of the heap as required, using `sbrk(2)`. When allocating blocks of memory larger than `MMAP_THRESHOLD` bytes, the glibc `malloc()` implementation allocates the memory as a private anonymous mapping using `mmap(2)`. `MMAP_THRESHOLD` is 128 kB by default, but is adjustable using `mallopt(3)`. Prior to Linux 4.7 allocations performed using `mmap(2)` were unaffected by the `RLIMIT_DATA` resource limit; since Linux 4.7, this limit is also enforced for allocations performed using `mmap(2)`.

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-management data structures employed by these functions. In a multithreaded application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional memory allocation arenas if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using `brk(2)` or `mmap(2)`), and managed with its own mutexes.

SUSv2 requires `malloc()`, `calloc()`, and `realloc()` to set `errno` to `ENOMEM` upon failure. Glibc assumes that this is done (and the glibc versions of these routines do this); if you use a private

malloc implementation that does not set `errno`, then certain library routines may fail without having a reason in `errno`.

Crashes in `malloc()`, `calloc()`, `realloc()`, or `free()` are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

The `malloc()` implementation is tunable via environment variables; see `mallopt(3)` for details.

SEE ALSO

`valgrind(1)`, `brk(2)`, `mmap(2)`, `alloca(3)`, `malloc_get_state(3)`, `malloc_info(3)`, `malloc_trim(3)`, `malloc_usable_size(3)`, `mallopt(3)`, `mcheck(3)`, `mtrace(3)`, `posix_memalign(3)`

System Programming Jokes

0x43 0x61 0x74 0xe0 0xf9 0xbf 0x5f 0xff 0x7f 0x00

Warning: Authors are not responsible for any neuro-apoptosis caused by these “jokes.” - Groaners are allowed.

Light bulb jokes

- Q. How many system programmers does it take to change a lightbulb?
- A. Just one but they keep changing it until it returns zero.
 - A. None they prefer an empty socket.
 - A. Well you start with one but actually it waits for a child to do all of the work.

Groaners

Why did the baby system programmer like their new colorful blankie? It was multithreaded.
 Why are your programs so fine and soft? I only use 400-thread-count or higher programs.
 Where do bad student shell processes go when they die? Forking Hell.
 Why are C programmers so messy? They store everything in one big heap.

System Programmer (Definition)

A system programmer is...

Someone who knows sleepsort is a bad idea but still dreams of an excuse to use it.

Someone who never lets their code deadlock... but when it does, it causes more problems than everyone else combined.

Someone who believes zombies are real.

Someone who doesn't trust their process to run correctly without testing with the same data, kernel, compiler, RAM, filesystem size, file system format, disk brand, core count, CPU load, weather, magnetic flux, orientation, pixie dust, horoscope sign, wall color, wall gloss and reflectance, motherboard, vibration, illumination, backup battery, time of day, temperature, humidity, lunar position, sun-moon, co-position...

A system program ...

Evolves until it can send email.

Evolves until it has the potential to create, connect and kill other programs and consume all possible CPU, memory, network, ... resources on all possible devices but chooses not to. Today.