Linux Essential Concepts

system calls

- system calls (short: *syscalls*)
  - function invocations made from user spaceinto kernel, to request services/resources from the OS
  - exmples: `read()`, `write()`, `get_thread_area()` (~ 380 syscalls)

glibc

- implements standard C library
- provides wrappers for system calls (and threading support + basic application facilities)

API and ABI

- Application Programming Interface (*API*)
  - defines interfaces by which one piece of software communicates with another at the *source level*
- Application Binary Interface (*ABI*)
  - defines binary interface between pieces of software on a particular architecture (e.g. x86-64)
  - concerned with e.g.
    - calling conventions, byte ordering, register use, system call invocation, *(cont'd)*
    - linking, library behavior, and the binary object format
- enforced by the toolchain (compiler, linker)

standards

- *POSIX* - Portable Operating System Interface
- *SUS* - Single UNIX Specification
- POSIX and SUS document e.g. the C API for a Unix-like operating system interface
- SUS subsumes/includes POSIX
- *LSB* Linux Standard Base; LSB Wikipedia
  - > ... standardize the software system structure, including the Filesystem Hierarchy Standard used in the Linux kernel. The LSB is based on the POSIX specification, the Single UNIX Specification (SUS) and several other open standards, but extends them in certain areas. ...

files / filesystem

- *everything is a file* philosophy

- to be accessed, a file must be opened; can be open for reading or writing or both

- **file *descriptor*** - reference to open file, maps metadata of open file to specific file

  - of C type `int`
  - *fd* abbreviated
  - shared with user space
  - a file can be opened more than once, each open instance has their own unique fd
  - concurrent file access - must be coordinated by user-space programs themselves

- **regular files**

  - bytes of data, organized in linear array called *byte stream*

  - *file offset* or *file position* = location within the file

- *offset* max value = size of the C type used to store it (64 bits usually on modern systems)
- writing bytes to file positions beyond the end of file will cause intervening bytes padded with zero

  ```
  #include <fcntl.h> // for open
  #include <unistd.h> // for write
  int main() {
    int fd; // file descriptor
    // O_RDWR open read+write, O_CREAT create if not exist
    fd = open("file.bin", O_RDWR|O_CREAT);
    char onebyte = 1;
    write(fd, &onebyte, sizeof(char));
    // 'p'=position; last param is  off_t offset
    pwrite(fd, &onebyte, sizeof(char), 10);
    close(fd);
  }
  ```

  ```
  $ hexdump file.bin
  0000000 01 00 00 00 00 00 00 00 00 00 01
  ```

- **inodes**
  * a file is referenced by an *inode* (information node)
  * inode number (or *i-number* or *ino* = integer value unique to filesystem
  * *inode* stores file *metadata* e.g.
    · e.g. modification timestamp, owner, type, length, and the location
    · metadata does *not* include filename
  * both physical obj on disk; and data structure `struct inode` / in-meory representation in kernel
    · see kernel doc at The Inode Object
    ·
      > They live either on the disc (for block device filesystems) or in the memory (for pseudo filesystems)
  * `ls -i` lists inode numbers e.g. `ls -i /tmp/file.bin`, output: `2671722 /tmp/file.bin`
  * *directories* - mapping of human-readable names to inode numbers
    · like any normal file, with difference that it contains only mappings of names to inodes
  * *link* = (file-)name and inode pair(-mapping)
    ·
      > . . . The physical on-disk form of this mapping—for example, a simple table or a hash—is implemented and managed by the kernel code that supports a given filesystem. . .
  * *dentry* = directory entry
  * *directory or pathname resolution* - kernel's walk though dentries to find a specific inode
  * *fully qualified / absolute pathname* - starting at `root` "/" directory
  * unlike normal files, kernel does not allow directories to be opened/manipulated like regular files
    · can only be manipulated via specific syscalls
- **hard links** and **symbolic links**
  * *hard link* multiple links map different (file-)names to the same inode
    · deleting a file *unlinks* it from directory (removing name-inode pair from directory)
    · each inode contains a *link count*
  * *symlinks* - has its own inode and data chunk, containing the complete pathname of linked-to file
  * Example: *note string length "hardlink.txt" = 12*

    ```
    inode   Permissions   Links Size  Name
    ```

```
2674927 .rw-r--r--      2    0  hardlink.txt
2674927 .rw-r--r--      2    0  hardlink2.txt
2674944 lrwxr-xr-x      1   12  softlink.txt -> hardlink.txt
```

special files

- types of special files (four)
  - block device files
  - character device files
  - named pipes (FIFO)
  - Unix domain sockets
- **device files** - two groups 1. *character devices* and 2. *block devices*
  - character device - linear queue of bytes
  - block device - (accessed as) an array of bytes
- **names pipes** (or **FIFO**)
  - Named pipes are an interprocess communication (IPC)
  - communication channel over a fd (file descriptor), accessed via a special file
- **sockets** (Unix domain sockets)
  - advanced form of IPC (interprocess communication), multiple varieties
  - communication between two different processes; on same of different machines
  - *Unix domain socket* = form of socket used for communication within the local machine

filesystem and namespaces

- *unified namespace* in Linux/Unix (e.g. in Windows drives have a separate namespace such as `A:\`
- *filesystem* = collection of files and directories in a hierarchy
- *mount* / *unmount* - adding/removing a filesystem to global namespace
- filesystems…
  - *physical* - stored on disk
  - *virtual* - only exist in memory
  - *network* - exist on machines across network
- *sector* - smallest addressable unit on block device; pysical attribute of device
- *block* - smallest logically addressable unit on a filesystem
  - usually a power-of-two multiple of the sector size
  - generally larger than sector
- see Kernel doc 'Queue sysfs files'…
  - … and man (2) page for syscall 'stat, fstat, lstat, fstatat - get file status'
  - **hw_sector_size (RO)** - hardware sector size of the device, in bytes.

    ```
    $ cat /sys/block/dm-0/queue/hw_sector_size
    512
    # block size
    $ sudo blockdev --getbsz /dev/dm-0
    4096
    $ stat -f .
    Block size: 4096       Fundamental block size: 4096
    Blocks: Total: 3724026    Free: 2246906    Available: 2052807
    Inodes: Total: 950272     Free: 794298
    ```
- *per-process namespaces* by default
  - each process inherits the namespace of parent
  - a process may create its own namespace with own set of mount points and a unique root directory.

processes

- *processes* are object code in executing / active, running programs, consisting of
  - object code
  - data
  - resources
  - state
  - a virtualized computer
- **ELF** *Excecutable and Linkable Format*
  - machine-runnable code in executable format kernel understands, contains:
  - metadata
  - multiple *sections* of code and data
- ELF **sections** - linear chunks of obj code, all bytes in a section are treated same (e.g. permission)
  - *text section* - executable code and read-only data (e.g. constants); read-only
  - *data sections* - initialized data e.g. C variables with defined values; read-write
  - *bss section* - uninitialized global data to be initialized (optimization) by *zero page*
  - *absolute section* - nonrelocatable symbols
  - *undefined section* - (catchall)
- process **resources**
  - managed by kernel
  - resource manipulation through system calls
  - examples: timers, pending signals, open files, network connections, IPC, ..
  - **process descriptor** - inside kernel structure for process resources, data, statistics, ..
  - process = **virtualization abstraction**
    - kernel supports both *preemptive multitasking* and *virtual memory*
    - each process has a single linear address space, as if it were in control of all of system memory
  - **threads**
    - each process consists of one or more *threads* ('thread of execution')
    - *thread* = the unit of activity within a process
    - a *thread* consists of
      - a stack (stores local variables)
      - processor status
      - current location of object code (usually processor's *instruction pointer*)
    - shared with process:
      - address space; thread share same virtual memory abstraction
    - kernel internal - views thread as normal processes to share some resources
  - **process hierarchy**
    - each process is identified by unique positive integer *process ID*
    - processes form a strict hierarcy, the *process tree*
    - PID = 1 first process or *init process*
    - new processes created via the `fork()` syscall
      - creates a duplicate of the calling (=parent) process
      - original process = *parent*, new process = *child*
      - child processes inherit the uids of their parents.
      - *reparenting* - if parent terminates before child, kernel will *reparent* child to init process
    - a process is not immediately removed…
      - instead kernel keeps part in memory to allow parent to inquiry about staus WAIT(2):
      - > …In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state …"

users and groups

- *authorization* in Linux is provided by *users* and *groups*
- *user ID* (uid) - unique positive int associated with user
  - each process in turn associated with one uid; called process *real uid*
- uid 0 = *root*
- each user has a *primary* or *login group* (in `/etc/passwd`); *supplemental groups* are in `/etc/groups`

permissions

- each file is associated with
  - an owning user
  - an owning group
  - three sets of permission bits

signals

- *signals* - mechanism for one-way asynchronous notification
- typically alert a process about some event; about 30 signals implemented in kernel
- may be sent from:
  - kernel to process
  - process to another process
  - process to itself
- each signal is represented by a numeric constant and a textual name e.g.

Signal	Value	Comment
SIGHUP	1	Hangup detected on controlling terminal
SIGINT	2	Interrupt from keyboard
SIGKILL	9	Kill signal

- signals interrupt a process…
  - causing it to stop whatever it is doing + immediately perform a predetermined action
  - processes may control what happens when receiving a signal ('predetermined action')
  - exceptions `SIGKILL` (always terminates) and `SIGSTOP` (always stops) processes

interprocess communication

- IPC mechanisms supported by Linux include:
  - pipes
  - named pipes
  - semaphores
  - message queues
  - shared memory
  - futexes

Filesystem Hierarchy Standard (FHS)

FHS Specification Series

- based on two independent distinctions:
  - shareable vs. unshareable
  - variable vs. static
  - example:

    * *sharable*: user home dirs are shareable, device lock files not
    * *static*: binaries, libraries; files that do not change without system admin invtervention

root filesystem

dir	description
bin	Essential command binaries
boot	Static files of the boot loader
dev	Device files
etc	Host-specific system configuration
lib	Essential shared libraries and kernel modules (in subdir/link `moduels`)
media	Mount point for removable media
mnt	Mount point for mounting a filesystem temporarily
opt	Add-on application software packages
run	Data relevant to running processes
sbin	Essential (vital) system binaries and root-only commands
srv	Data for services provided by this system
tmp	Temporary files
usr	Secondary hierarchy
var	Variable data

- Optional:
  - `/home` user home dirs
  - `/root` home dir for root user
  - `/lib<qual>` alternate format essential libraries, e.g.

    ```
    $ ls /lib
    lib/   lib64/
    ```
- /**sys** *not in table above*
  - the location where information about devices, drivers, and some kernel features is exposed.

  sysfs is a ram-based filesystem [...]. It provides a means to export kernel data structures, their attributes, ...
- /**proc** *not in table above*
  - referred to as *process information pseudo-file system*
  - regarded as '*control and information center for the kernel*'
    * many sys utils just use the files in proc
    * `lsmod` is the same as `cat /proc/modules`
  - contains runtime system information e.g. mounted devices, hardware, configuration
  - read/change kernel parameters by using files in proc (`sysctl`)
  - *Note:* all files in '/proc' have a file size of '0' (with the exception of kcore, mtrr and self)
  - detail description in man page `man 5 proc`
  - see also
    * tldp: 1.14. /proc
    * proc man page

    * kernel.org doc on 'proc'

- **/run**

  The purposes of this directory were once served by `/var/run`. In general, programs may continue to use `/var/run` to fulfill the requirements set out for /run for the purposes of backwards compatibility. E.g. on both Ubuntu and ArchLinux

```
$ ls -ld /var/run
lrwxrwxrwx 6 root 19 Jan 02:32 /var/run -> ../run
```

- Examples entries in 'run':

```
/run/sshd.pid
$ ls /srv/
ftp  http
```

- **/tmp**

  Programs must not assume that any files or directories in `/tmp` are preserved between invocations of the program.

/usr/*

dir	description
bin	Most user commands
lib	Libraries
local	Local hierarchy (empty after main installation)
sbin	Non-vital system binaries
share	Architecture-independent data

- Optional:
  - `games` games and educational binaries
  - `include` header files included by C programs
  - `libexec` binaries run by other programs
  - `lib<qual>` alternate Format Libraries
  - `src` source code
- **/urs/bin**

  primary directory of executable commands on the system. E.g. python, perl, etc
  /bin contains *essential* user command binaries such as `mount`, `rm`, `ls` etc

```
[archlinux@archlinux ~]$ ls -ld /bin
lrwxrwxrwx 7 root 19 Jan 02:32 /bin -> usr/bin

ubuntu@ubuntu20:~$ ls -ld /bin
lrwxrwxrwx 1 root root 7 Feb  1 17:20 /bin -> usr/bin
```

- **/usr/local**

  The /usr/local hierarchy is for use by the system administrator when installing software locally. It needs to be safe from being overwritten when the system software is updated.

- Requires the following sub-dirs (exerpt): `bin`, `etc`, `include`, `share`, etc

- **/usr/sbin**

  > ...non-essential binaries used exclusively by the system administrator. Note:
  > System admin programs required for system repair, system recovery, mounting /usr, or other essential functions must be placed in /sbin instead. No subdirectories allowed.

```
[archlinux@archlinux ~]$ ls -ld /usr/sbin
lrwxrwxrwx 3 root 19 Jan 02:32 /usr/sbin -> bin
[archlinux@archlinux ~]$ ls -ld /sbin
lrwxrwxrwx 7 root 19 Jan 02:32 /sbin -> usr/bin

ubuntu@ubuntu20:/usr$ ls -ld /usr/sbin/
drwxr-xr-x 2 root root 16384 May 23 09:58 /usr/sbin/
ubuntu@ubuntu20:/usr$ ls -l /sbin
lrwxrwxrwx 1 root root 8 Feb  1 17:20 /sbin -> usr/sbin
```

- **/usr/share**

  all read-only architecture independent (i386, Alpha, etc) data files. E.g. the following directories (or symlinks) must be in `/usr/share`

dir	description
man	man pages
misc	Misc arch-independent data

/var/*

- Variable data files e.g.
  - spool directories
  - administrative data
  - logging data
  - temp and transient files
- contains both
  - shareable portions (e.g. `/var/mail`, `/var/cache/fonts`)
  - non-shareable portions (e.g. `/var/lock`, `/var/log`)

dir	description
cache	Application cache data
lib	Variable state information
local	Variable data for /usr/local
lock	Lock files
log	Log files and directories
opt	Variable data for /opt
run	Data relevant to running processes
spool	Application spool data
tmp	Temporary files preserved between system reboots

- **/var/lib**

  This hierarchy holds state information pertaining to an application or the system. State information is data that programs modify while they run, and that pertains to one specific host. Examples: `/var/lib/pacman`, `/var/lib/apt`, `/var/lib/man-db`

- **/var/opt**

  Variable data of the packages in /opt must be installed in /var/opt/<subdir>, where <subdir> is the name of the subtree in /opt where the static data

- **/var/spool**
- data which is awaiting some kind of later processing, e.g.
  - `lpd` - printer spool dir

  - `mqueue` - outgoing mail queue
- **/var/tmp**

  The /var/tmp directory is made available for programs that require temporary files or directories that are preserved between system reboots. Therefore, data stored in /var/tmp is more persistent than data in /tmp.

Links, references etc

- `/usr/local` vs `/opt`
  - Linux Journal: Point/Counterpoint - /opt vs. /usr/local
  - Stackexchange: What is the difference between /opt and /usr/local?

Process management

- Operating-System-Concepts
  - as a process executes, can have one of the following states
    * *new* being created
    * *running*
    * *wating* e.g. for an I/O completion of receiving a signal
    * *ready* waiting to be assigned
    * *terminated*
- **PCB** - **P**rocess **C**ontrol **B**lock
  - the kernel datastructure representing a process in an OS
  - containg information such as process state, list of open files, pointer to process' parent etc
  - in Linux represented by *task_struct* (doubly linked list), exposed via `/proc/<process-id>/...`
- process scheduling queues
  - *ready queue* - ready and waiting to be executed
  - *waits queue* - processes waiting for a certain event such as completion of I/O
- **IPC** interprocess communication (IPC), two fundamental models
  - (1) shared memory e.g. `shm_open` - create/open POSIX shared memory objects
  - (2) message passing
  - *pipes* - allow two processes to communicate in standard producer-consumer fashion
    * ordinary / anonymous pipes
    * names pipes or FIFOs - `mkfifo()` syscall
  - *sockets* - 'socket' = endpoint for communication
  - *RCP* - remote procedure calls
- Linux-Kernel-Development
  - processes provide two virtualizations
    * (1) a virtualized processor
    * (2) virtual memory
  - in current Linux, `fork()` is implemented via the `clone()` system call; clone man page:
    * *By contrast with fork(2), these system calls provide more precise control over*
    * *what pieces of execution context are shared between the calling process and the child process.*
  - process state: `state` field in `task_struct`:
    * `TASK_RUNNING`
    * `TASK_INTERRUPTIBLE` *waiting* - sleeping/blocked, receives and reacts to signals
    * `TASK_UNINTERRUPTIBLE` *waiting* - does not wake and become runnable (even when recv signal)
    * `__TASK_TRACED` - being traced (e.g. debugger, ptrace)
    * `__TASK_STOPPED`

note:

> On Linux, 0 typically represents success; a nonzero value, such as 1 or -1, corresponds to failure.

- in Unix, the creation of a new process and the act of loading a new binary is separated
  - `fork()`
  - `exec()`
- *idle process* (PID=0) process the kernel 'runs' when there's not other runnable process
- *init process* (PID=1) first process kernel executes after booting
  - can be specified with *init* kernel command-line parameter
  - kernel tries four executables:
    * (1) `/sbin/init`
    * (2) `/etc/init`
    * (3) `/bin/init`
    * (4) `/bin/sh`
- **process id allocation**
  - kernel: pid is of type `pid_t`, generally `int` e.g. *posix_types.h* `typedef int  __kernel_pid_t;`
  - maximum process id:
    ```
    $ cat /proc/sys/kernel/pid_max
    4194304
    ```
  - allocation is in a strictly linear fashion
  - kernel does not reuse process IDs until it wraps around from top
- **process hierarchy**
  - every process is spawned from another process (exept `init` with pid 1)
  - releationship is recorded in *parent process ID* (**ppid)
  - each process is part of a *process group*
    * e.g. `ls|less` processes belong to same process group
    ```c
    #include <unistd.h>
    #include <stdio.h>
    int main() {
    printf("pid  = %d\n", getpid());
    printf("ppid = %d\n", getppid());
    }
    ```
- **exec()**
  - there's no single exec function, instead a range of exec functions built on singel syscall
  - `execl()` replaces current process image with new one specified with `path`
    ```c
    int main() {
      execl("/bin/ls", "ls", NULL);
    }
    ```
  - Unix convention is to pass the program name as the program's first argument.
  - open files are inherited across an exec
    * often not the desired behavior
    * usual practice is to close files before the exec
    * can be done automatically via kernel with `fcntl()` (fcntl=file control)
- **fork()**
  - fork() creates a new process
  - child and the parent process are (almost) identical except for
    * pid / ppid
    * resource statistics (reset to zero in child)
    * any pending signals are cleared
    * file locks are not inherited by child
      ```c
      int main() {
        pid_t pid = fork ();
        //sleep(1);
        if (pid > 0)
          printf ("Parent of new child; child pid=%d\n", pid);
      ```

```
        else if (!pid)
          printf ("Child with pid=%d and ppid=%d\n", getpid(), getppid() );
      }
      # output without sleep -> parent terminates first and kernel reparents child
       Parent of new child; child pid=60525
       Child with pid=60525 and ppid=1
      # output with sleep
       Parent of new child; child pid=60874
       Child with pid=60874 and ppid=60873
```

    * `fork()` - returns 0 to the child process, and returns pid of child process to the parent process

```
      pid_t pid = fork();
      if (pid==0) printf("I am the child.\n");
      if (pid!=0) printf("I am the parent.\n");
      printf("..and I will be printed twice. pid=%d\n", getpid());
      # output:
       I am the parent.
       ..and I will be printed twice. pid=68782
       I am the child.
       ..and I will be printed twice. pid=68783
```

  - **copy-on-write**
    * modern Unix systems do not copy the parent's address space, but employ copy-on-write (COW) pages
    * COW is a lazy optimization strategy designed to mitigate the overhead of duplicating resources
    * modern machine architectures provide HW support for COW in their memory management units (MMUs)
- **exit() / _ exit()**
  - standard function for terminating the current process
  - `exit()` (Standard C Library function), performs following functions (in order):
    * (1) Call the functions registered with the atexit(3) function, in the reverse order of their registration.
    * (2) Flush all open output streams.
    * (3) Close all open streams.
    * (4) Unlink all files created with the tmpfile(3) function.
  - when done with above steps, `exit()` invokes the syscall `_ exit()`
  - the kernel handles the rest of the termination process

daemons

- process that runs in the background, not connected to a (controlling) terminal
- two requirements:
  - must run as child of *init*
  - must not be connected to terminal
- for a program to become a daemon:
  - (1) `fork()`
  - (2) in the parent call `exit()`; *reparents* process to pid 1 (=init)
  - (3) call `setsid()` - with process itself as leader
  - (4) change the working directory to the root directory "/" via `chdir()`
  - (5) close all file descriptors (`ulimit -a`: open files (-n) 1024)
  - (6) open file descriptors 0, 1, 2 and redirect them to `/dev/null`

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

```
#include <unistd.h>
int main (void) {
        pid_t pid = fork (); // (1)
        if (pid != 0)  // (2)
          exit (EXIT_SUCCESS);
        setsid (); // (3)
        chdir ("/"); // (4)
        for (int i = 0; i < 1024; i++) // (5)
                close (i);
        // (6) note: open() always allocates lowest available fd number
        int fd = open ("/dev/null", O_RDWR); /* 0 stdin */
        dup (0);                             /* 1 stdout */
        dup (0);                             /* 2 stderror */
        /* daemon code goes here */
        return 0;
}
```

Memory Management

Operating-System-Concepts

- **paging** - a memory-management scheme allowing process's physical address space to be non-contiguous.

Linux-System-Programming

- processes do not directly address physical memory
- each process is associates with a (unique) *virtual address space*
- address space is *linear* (start at zero) and *flat*
- **page** smallest addressable unit of memory by MMU (memory management unit)
  - page size is defined by the hardware
  - get pagesize via `/proc`, `getconf`, or syscall `getpagesize()`

    ```
    # cat /proc/1/smaps | grep -i pagesize | head -2
    KernelPageSize:         4 kB
    MMUPageSize:            4 kB
    # getconf PAGESIZE
    4096
    ```

    ```
    #include <unistd.h>
    #include <stdio.h>
    int main() {
       printf("%d\n", getpagesize());
    }
    ```

  - pages are either valid or invalid
    * *valid page* - associated with a page of data (in RAM or secondary storage/swap)
    * *invalid page* - not associated with anything (unused, unallocated address space)
  - **page fault**: - *valid* page but not currently in RAM and needs to be *paged in*
  - **segmentation fault** - caused by trying to access an *invalid* page

```c
#include <stdlib.h>
int main() {
  int *ip;
  ip = (int *) malloc( sizeof(int)*10 );
  ip[100000] = 99; // outside array's 10 elements, seg fault
}
```

- memory regtions (areas/mappings) in every process
  - *text segment* the actual program code, constants etc; ready only data
  - *data segment* or *heap* - dynamic memory (`malloc()`)
  - *bss segment* unitialzed global variables (zeroes)
  - *stack* grows/shrinks dynamically; e.g. local vars, function return data. Thread: one stack per thread

```
$ ulimit -Sa | grep -i stack
stack size                  (kbytes, -s) 8192
```

- **dynamic memory** - allocated at runtim
  - `void * malloc (size_t size);`
  - `calloc()` - zero's out bytes ('c'=clear)
  - `realloc()` > if unable to enlarge the existing chunk of memory by growing the chunk in situ, > the function may allocate a new region of memory size bytes in length, > copy the old region into the new one, and free the old region
- **anonymous memory mapping**
  - for large allocations, glibc doesn't use the heap; instead creates an anonymous memory mapping
  - `mmap()` creates a memory mapping and the system call `munmap()` destroys a mapping
- **advanced memory allocation**
  - `mallopt()` - sets the memory-management-related parameter e.g.
    * `M_MMAP_THRESHOLD` - malloc allocation request via an anonymous mapping instead of heap
    * default value (Linux) if not set for `M_MMAP_THRESHOLD` = 128k (128*2024)
- **stack-based allocations**
  - one can use the stack for dynamic memory allocations too (as long as it doesn't overflow)
  - `alloca()` system call for dynanimc memory allocation from stack > alloca() returns a pointer to size bytes of memory. > This memory lives on the stack and is automatically freed when the invoking function returns.

```c
#include <alloca.h>
#include <stdlib.h>
int main() {
  void *p;
  // int size = 8388608; /* will segfault as ulimit: stack size (kbytes, -s) 8192
  int size = 8388608 / 2;
  p = alloca(size);
  // free(p); /* will result in Aborted (core dumped) */
}
```

  - very fast compared to `malloc()` (just increases stack pointer)
- **locking memory** - two situations you might want to change default paging behaviour:

  - (1) *determinism* - page fault are costly b/c of I/O operations (disk)
  - (2) *security* - if secrets kept in memory are paged out to (unencrypted) disk
  - `mlock()` - to locks a specific memory area/allocation, unlock with `munlock()`
  - `mlockall()` - locks the whole process space, unlock with `munlockall()`

Multithreading / Synchronization

Threading

- Operating-System-Concepts
  - **Note:** *concurrency* vs *parallelism*
  - programming callenges for multicore/multi-threaded systems:
    * identifying tasks - what can be divided into separate tasks (ideally independent)
    * balance - tasks performing equal work of equal value; if not worth separate task/thread at all?
    * data splitting - data accessed and changes by tasks must be divided too
    * data dependency - synchronization in case of data dependencies of tasks
    * testing debugging
- Linux-Kernel-Development
  > To the Linux kernel, there is no concept of a thread.
  > Linux implements all threads as standard processes
  > The Linux kernel does not provide any special scheduling semantics or
  > data structures to represent threads. Instead, a thread is merely
  > a process that shares certain resources with other processes.
  - syscall `clone()` to create threads
  - *kernel threads* standard processes that exist solely in kernel-space.

Threading - creation and management of multiple units of execution within a single process.
Binary - dormant program resigin on storage
Process - OS abstraction representing the loaded binary, virtual. mem., kernel resources (fds etc)
Thread - unit of execution withing a process: a virtualized processor, a stack, and program state

- Linux-System-Programming
  - Modern OSs have two fundamental (virtualized) abstractions to user-space:
    * virtualized processsor
      · associated with with threads (and not the process)
    * virtual memory
      · associated with process; hence each process has a unique view of memory
      · however all threads in a process *share* this memory
  - Six primary benefits to multithreading:
    * (1) Programming abstraction - dividing work as assigning to units of execution (threads)
    * (2) Parallelism
    * (3) Improving responsiveness
    * (4) Blocking I/O
    * (5) Context switching - cheaper from thread-to-thread (*intraprocess switching*) than process-to-process
    * (6) Memory savings
  - User-level threading
    * *N:1 threading*, also called *user-level threading*
    * a process with N threads maps to a single kernel process (N:1)
    * scheduler etc is a all user-space code
    * not true prallelism

  - Hyper threading
    - *N:M* threading, also known as *hybrid threading*
    - the kernel provides a native thread concept, while user space also implements user threads
  - Threading patterns, the two core programming patterns are:
    - *thread-per-connection* - unit of work is assigned to one thread, which "runs until completion"
    - *event-driven*:
      - comes from observation that threads do a lot of waiting: reading files, waiting for dbs, etc
      - suggested to consider the *event-driven pattern* first i.e.
      - asynchronous I/O, callbacks, an event loop, a small thread pool with just a thread per processor.
  - *Race condition* situation in which unsynchronized access of shared resources lead to errors
    - shared resources examples: system's hw, a kernel resource, data in memory etc
    - *critical region* - region of code that should be synchronized
  - **Synchronization**
    - The fundamental source of races is that critical regions are a window during which correct program behavior requires that threads do not interleave execution. To prevent race conditions, then, the programmer needs to synchronize access to that window, ensuring mutually exclusive access to the critical region.
    - *atomic operation* - indivisible, unable to be interleaved with other operations
      - problem with critical regions: they are not atomic
  - **Locks / Mutexes**
    - *lock* mechanism for ensuring mutual exclusion within a cricial region, making it *atomic*
    - also knows as *mutexes* (e.g. in Pthreads)
    - the smaller the critical region, the better:
      - locks prevent concurreny thus negating threading benefits
  - ***Deadlock*** - situation where 2 threads are waiting for the other to finish, thus neither does

Synchronization

Operating-System-Concepts

- **race condition**
  - several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place
  - examples in *kernel code*
    - data structure maintaining a list of all open files
    - variable *next_available_pid*
- **mutex lock** - simplest 'tool' to protect critical sections and prevent race conditions
  - short for **mu**tual **ex**clusion
  - example, *note* calls to `acquire` and `release` must be performed atomacially e.g. using CAS (compare-and-swap)

```
acquire() {
  while (!available)
    ; /* busy wait */
    available = false;
}

release() {
  available = true;
}
```

  - mutex **lock** states:
    - *contended* thread blocks while trying to aquire lock

        - *uncontended* lock is available when thread attempts to require it
    - another name for this *type of mutex lock*: **spinlock**
    - can be held by at most one thread of execution
    - main **disadvantage**: ***busy waiting*** (spinlock)
        - process *spins* while waiting for lock (busy wait)
        - advantage: no context switch required:
            - *a context switch to move the thread to the waiting state and a*
            - *second context switch to restore the waiting thread once the lock becomes available*
- **semaphore** - an `int` accessed only through two standard atomic operations: `wait()` and `signal()`
    - from Linux-Kernel-Development:
    - in Linux, sepaphores are *sleeping locks*
        - *When a task attempts to acquire a semaphore that is unavailable, the semaphore places*
        - *the task onto a* ***wait queue*** *and puts the task to sleep.*
    - *pro* because task sleeps, semaphores are well suited to locks that are held for a long time
    - *con* because of overhead (two context switches) not optimal for locks held for short periods
    - often 'used' as sleeping *mutual exclusion lock* ("sleeping spin lock") with a *count* of one
    - *counting semaphore* (value between 0 and 1) - can be used instead of mutex locks
    - can have arbitrary number of simultaneous lock holders
        - spinlock as well as `mutex_lock` at most one task/thread
    - both for critical sections (mutual exclusion) and coordination (scheduling)
- **monitor**
    - provides mutual exclusion; programming language construct (not available in all languages)
    - *"The monitor construct ensures that only one process at a time is active within the monitor."*
    - A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.
    - Examples:
        - in Java part of language; `synchronized` keyword
        - in Rust not part of language but provided via `Struct shared_mutex::monitor::Monitor`
- **liveness**
    - refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle.
- **deadlock**
    - a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.
    - **prevention** - few simple rules:
        - Implement *lock ordering*; document lock ordering. Always aquire lock in same order (ABAB vs ABBA)
        - Prevent *starvation*. Will code always finish?(or dependent on certain event on potentially wit forever)
        - Do *not double aquire the same lock*
        - *Simplicity* - design for simplicity; complexity in locking schemes 'invites' deadlocks
- locking *approaches*
    - *optimistic* - first update variable and then use collision detection
    - *pessimistic* - assume another thread is concurrently updating the variable...
        - so you pessimistically acquire the lock before making any updates.

Semaphores vs Mutexes vs Spinlock

- Mutexes and Smaphores are simalar; both sleep wait processes
    - use new mutex type whenever possible over semaphore (b/c it's simpler)
- Spinlocks if short lock hold time or in interrupt handler (semaphores can't as they'd sleep)

Synchronization Examples

Note:

> Prior kernel 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run.

- **Kernel**
  - Linux-Kernel-Development *"mutex" is a generic name to refer to any sleeping lock that enforces mutual exclusion*
  - *Atomics*: Linux kernel atomic integer `atomic_t` with atomic operations `atomic_inc(&i)` or `atomic_read(&i)`
  - *Semaphores*:
    - Linux-Kernel-Development "Until recently, the only sleeping lock in the kernel was the semaphore"

    ```
    struct semaphore name;
    sema_init(&name, count);
    ```
  - *Mutex locks*: `mutex_lock()` / `mutex_unlock()`
    - "If the mutex lock is unavailable, a task calling `mutex lock()` is put into a sleep state and is awakened when the lock's owner invokes `mutex unlock()`"
    - Linux-Kernel-Development - behaves similar to a semaphore with a count of one, but it has a simpler interface
  - *Spinlock*

  ```
  <linux/spinlock.h>
  spin_lock(S);
  /* critical section */
  spin_unlock(S);
  ```
- POSIX **Pthreads**
  - mutex locks data type `pthread_mutex_t`

  ```
  #include <pthread.h>
  int main() {
      pthread_mutex_t mutex;
      pthread_mutex_init(&mutex,NULL); /* create + init the mutex lock */
      pthread_mutex_lock(&mutex); /* acquire lock */
      /* ******************************
       * critical section goes here
       * ******************************/
      pthread_mutex_unlock(&mutex); /* release lock */
  }
  ```
- Pthreads deadlock example:

```
$ ps uaxH  # process state: S
archlin+     902  0.0  0.0  18524   568 pts/0    Sl+  10:28   0:00 ./a.out
# PROCESS STATE CODES
# S    interruptible sleep (waiting for an event to complete)
# l    is multi-threaded
```

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

pthread_mutex_t mutex_A;
```

```
pthread_mutex_t mutex_B;

void *thread_function_01(void *param) {
    pthread_mutex_lock(&mutex_A);
    sleep(1); /* this will force deadlock as thread02 will lock mutex_B  */
    pthread_mutex_lock(&mutex_B);
        printf ("%s\n", "thread01"); /* do work */
    pthread_mutex_unlock(&mutex_B);
    pthread_mutex_unlock(&mutex_A);
    pthread_exit(0);
}
void *thread_function_02(void *param) {
    pthread_mutex_lock(&mutex_B);
    pthread_mutex_lock(&mutex_A);
        printf ("%s\n", "thread02"); /* do work */
    pthread_mutex_unlock(&mutex_A);
    pthread_mutex_unlock(&mutex_B);
    pthread_exit(0);
}
int main(void) {
    pthread_t thread_01, thread_02;
    pthread_mutex_init(&mutex_A, NULL);
    pthread_mutex_init(&mutex_B, NULL);
    pthread_create(&thread_01, NULL, thread_function_01, (void *) NULL);
    pthread_create(&thread_02, NULL, thread_function_02, (void *) NULL);
    pthread_join (thread_01, NULL);
    pthread_join (thread_02, NULL);
    return 0;
}
```

- **livelock**
  - a liveness failure
  - *livelock occurs when a thread continuously attempts an action that fails.*
  - *typically occurs when threads retry failing operations at the same time.*
  - *can generally be avoided by having each thread retry the failing operation at random times*
  - from Java documentation: > A thread often acts in response to the action of another thread. > If the other thread's action is also a response to the action of another thread, > then livelock may result.

Scheduling

Linux-Kernel-Development

Two flavours of multitasking OS:

- *cooperateive multitasking*
  - process does not stop until it itself voluntarily decides so
- *preemtive multitasking*
  - scheduler decides when a process ceases running and a new process begins running
  - the act of involuntarily suspending a running process is called preemption
  - time process runs before preempted is predetermined = *timeslice*
- Process classification (not mutually exclusive):
  - *I/O bound*
    - spends much of its time submitting and waiting on I/O

        - runs only shortly before it blocks waiting on I/O
    - *Processor bound*
        - majority of time spend executing code
- Process priority - Linux kernel has two separate priority ranges
    - *nice* -20 to +19, default 0 (lower = higher prio)
        - in Linux, a control over the *proportion of timeslice* (other Unixes: *absolute timeslice*)
        - `nice` syscall:

          ```
          #include <unistd.h>
          int nice(int inc);
          ```
    - *real-time priority* 0 to 99
- Timeslice / scheduler
    - Linux's CFS - Completely Fair Scheduler
        - does not directly assign timeslices to processes
        - assigns processes a proportion of the processor.
        - thus, amount of processor time a process receives is a function of the load of the system