Operating Systems Cheat Sheet

Command line

- `Ctrl-s` - Lock the console
- `Ctrl-q` - Unlock the console
- `Ctrl-w` - Delete last word
- `Ctrl-a` - Jump to beginning
- `Ctrl-e` - Jump to end
- `Ctrl-d` - End of file
- `Ctrl-` - cut text from current position until end of line

Vi shortcuts

- `u` - undo
- `Ctrl-r` - redo
- `v` - turn on/off visual editing
- `d` - cut
- `y` - copy
- `P` - paste before the cursor
- `p` - paste after cursor
- `>`, `<` - indent/unindent selected lines
- `:q!` - quit without saving

Linux manual

- **man shortcuts**
  - `/` - search
  - `n` - find next
  - `N` - find previous
- **finding a manual page**
  - `apropos`
  - `whatis`

Extended regular expressions

- `.` - any single character
- `\` - escape
- `[abc]` - any char in list
- `[a-z]` - range of chars
- `[^0-9]` - any character not in range
- `^` - beginning of line

- `$` - end of line
- `\<` - beginning of word
- `\>` - end of word
- `( )` - group
- `*` - zero or more times
- `+` - one or more times
- `?` - zero or one times
- `{m,n}` - at least `m`, at most `n` times
- `|` - logical `or`

Grep

- `-E` - use extended regular expression
- `-v` - show lines that don't match
- `-i` - ignore upper/lower case
- `-q` - don't display anything, exit with 0 if found, or 1 if not found
- `-o` - print only the matched text, in separate lines:

Sed

- `-E` - use extended regular expressions
- `-i` - edit files in-place
- ## search and replace
  - `sed -E "s/regex/replacement/flags" a.txt`
  - `s` - the search/replace command
  - `/` - separator, can use any character after `s`
  - the replacement can contain references to the grouped expressions in the regex as `\1`, `\2`, etc
  - `flags`:
    - `g` - replace globally, without it only first match is replaced
    - `i` - perform case-insensitive search
- transliterate:
  - `sed -E "y/aeiou/AEIOU/" a.txt`
- delete lines matching a regular expression:
  - `sed -E "/regex/d" a.txt`

Awk

- Given a separator character (by default it is space), treats the input text as a table, with each line being a row, and the
  fields of each row the tokens of the line, as determined by the separator.
- Processes the input based on a program written in a simple C-like language
- A program is a sequence of instruction blocks, prefixed by an optional selector

- Each block in the program is applied to every line of input matching its selector. If the block does not have a selector, it
  is applied to every line of input
- A selector is any valid conditional expression, or one of the following two special selectors
  - `BEGIN` - the block associated with this selector is executed before any input has been processed
  - `END` - the block associated with this selector is executed after all input has been processed
- Special variables
  - `NR` - number of the current line of input
  - `NF` - the number of fields on the current line
  - `$0` - the entire input line
  - `$1`, `$2`, ... - the fields of the current line
- The AWK program can be written in a file, or provided directly on the command line between apostrophes
- `awk -F:` - use `:` as separator
- `var ~ /regex/` - check if var matches regex
- `awk -F: '$NF ~ /nologin$/ {print $1}' /etc/passwd` - show username of all users having the last field end with `nologin`
- `awk -F: -f prog.awk input` - execute program provided in `prog.awk`

Other useful commands:

- **cut**
  - `cut -d: -f1,3-5` - use delimiter `:` and show fields 1, 3, 4 and 5
- **sort**
  - sorts lines
- **uniq**
  - merges adjacent matching lines
  - `-c` - prefix each line by number of occurrences
  - `-u` - print only unique lines
- **head**
  - output first part of files
- **find**
  - search for files in a directory
- **less**
  - pager for long outputs
- **ls**
  - list directories
- **ps**
  - show current processes
- **test**

- **wc**
  - `wc -l` - count number of lines
- **chmod**
  - `chmod 700 a.sh` - give a file execution permission
- **tr**
  - `tr '[A-Z]' '[a-z]'` - convert all uppercase characters to lowercase
- **file**
  - determine file type: `file a.txt | grep -q "text"`

I/O redirection

- `< input.txt`
- `> output.txt`
- `>> output.txt` - append to `output.txt`
- `2> output.err` redirect std error to `output.err`
- `rm some-file-that-does-not-exist.c > output.all 2>&1` redirect std output to output.all, redirect std error to the same place as std out

Shell variables

- `A="Tom"`
- `$A, ${A}`

Shell scripts

- `#!/bin/bash`
- **special variables**
  - `$0` - the name of the command
  - `$1` - `$9` - command line arguments
  - `$*` or `$@` - all the arguments
  - `$#` - number of arguments
  - `$?` - exit code of previous command
- `shift k` - shift out the first k arguments
- **for loop**

```
#!/bin/bash
for A in a b c d; do
echo Here is $A
done

for A in a b c d
do
echo Here is $A
done
```

- **expr**
  - `S=expr $S + $N`
- **if/elif/else/fi**

```
for A in $@; do
if [ -f $A ]; then
echo $A is a file
elif [ -d $A ]
then
echo $A is a dir
elif echo $A | grep -E -q "^[0-9]+$"; then
echo $A is a number
else
echo We do not know what $A is
fi
done
```

  - `[ -f $A ]` - equal to `test -f $A`
- **while**

```
#!/bin/bash
D=$1
S=$2
find $D -type f | while read F; do
N=`ls -l $F | awk '{print $5}'`
if test $N -gt $S; then
echo $F
fi
done
```

Processes

fork

- `fork()` - returns 0 in the child process, and the pid of the child in the parent process
- `getpid()`
- `getppid()`
- a child process becomes a zombie if it ends before the parent calls `wait`
- a zombie process stops when the parent calls `wait` or `waitpid`
- you need to call `wait` for each child process created

signals

```
#include <stdio.h>
#include <signal.h>
void f(int sgn) {
```

```
        printf("I refuse to stop!\n");
}

int main(int argc, char** argv) {
        signal(SIGINT, f);
        while(1);
        return 0;
}
```

- `kill(pid, SIGKILL)` - sends the `SIGKILL` signal to process `pid`

exec

```
// p: search for the program in PATH
char* a[] = {"ls", "-l", NULL};
execvp("ls", a);
execlp("ls", "ls", "-l", NULL);

a[0] = "/bin/ls"
execv("/bin/ls", a);
execl("/bin/ls", "/bin/ls", "-l", NULL);
```

pipe

```
int p[2];
pipe(p);

// in process 1:
close(p[0]); // close read end
write(p[1], &a, sizeof(int));
close(p[1]);

// in process 2:
close(p[1]); // close write end
read(p[0], &a, sizeof(int));
close(p[0]);
```

fifo

`mkfifo myfifo`
`mkfifo("myfifo", 0600);`

`rm myfifo`
`unlink("myfifo");`

`int fifo_d = open("myfifo", O_RDONLY);` waits until another process opens the fifo for the complementary operation

popen/pclose

```c
FILE* fp = popen("less", "w");
fprintf(fp, "This is sent to less\n");
pclose(fp);
```

```c
FILE* mypopen(char* cmd, char* type) {
        // works only for type = "w"
        int p[2];
        pipe(p);
        if (fork() == 0) {
                close(p[1]);
                dup2(p[0], 0);
                execlp("bash", "bash", "-c", cmd, NULL);
        }
        close(p[0]);
        return fdopen(p[1], type);
}
```

pthreads

```c
pthread_t t;
pthread_create(&t, NULL, f, &arg);
pthread_join(t, NULL);
```

mutexes

```c
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
pthread_mutex_lock(&m);
pthread_mutex_unlock(&m);
pthread_mutex_destroy(&m);
```

read-write locks

```c
pthread_rwlock_t rwl;
pthread_rwlock_init(&rwl, NULL);
pthread_rwlock_wrlock(&rwl);
pthread_rwlock_unlock(&rwl);
pthread_rwlock_rdlock(&rwl);
pthread_rwlock_unlock(&rwl);
pthread_rwlock_destroy(&rwl);
```

conditional variables

```c
pthread_mutex_t m;
pthread_cond_t c;
pthread_mutex_lock(&m);
```

```
// modify
pthread_mutex_unlock(&m);
pthread_cond_signal(&c);
pthread_cond_broadcast(&c);

pthread_mutex_lock(&m);
while (fuel < 40) {
        pthread_cond_wait(&c, &m);
        // Equivalent to:
        // pthread_mutex_unlock(&m);
        // wait for signal on c
        // pthread_mutex_lock(&m);
}
pthread_mutex_unlock(&m);
pthread_mutex_init(&m, NULL);
pthread_cond_init(&c, NULL);
pthread_mutex_destroy(&m);
pthread_cond_destroy(&c);
```

semaphores

- `P` - wait
- `V` - signal

```
sem_t s;
sem_init(&s, 0, 10);
sem_wait(&s);
sem_post(&s);
sem_destroy(&s);
```

barriers

```
pthread_barrier_t b;
pthread_barrier_init(&b, NULL, 10);
pthread_barrier_wait(&b);
pthread_barrier_destroy(&b);
```

Methods for preventing deadlocks

1. use a proper resource allocation strategy: employ a resource allocation strategy that ensures that resources are allocated in a way that avoids circular wait conditions, one of the necessary conditions for deadlocks occurrence. this can be achieved by implementing resource allocation algorithms such as the Banker's algorithm or using techniques like resource ordering.
2. avoid holding multiple resources simultaneously: encourage processes to request and acquire only one resource at a time, reducing the likelihood of circular dependencies. this approach helps prevent situations where a process holds one resource while waiting for another resource held by another process.

3. use a deadlock detection and recovery algorithm: implement a deadlock detection algorithm that periodically checks for the existence of deadlocks within the system. if a deadlock is detected, the algorithm can initiate recovery mechanisms such as process termination or resource preemption to resolve the deadlock and release the involved resources.
4. employ resource scheduling and allocation protocols: utilize scheduling and allocation protocols that aim to prevent or minimize the occurrence of deadlocks