Ficha 4

Para esta ficha devem utilizar sempre que possível *system calls*, especificas para POSIX.
No entanto para imprimir mensagens no STDOUT (consola/terminal), podem usar o `printf`, standard de C (para simplificar).

- Estas funções são uma interface para interagir com o kernel do sistema operativo
  - Ao contrário das funções standard de C, as as bibliotecas POSIX não são portáveis.
- O Windows terá outra API especifica para trabalhar com o kernel do Windows.
- Embora as bibliotecas standard de C (libc) sejam portáveis, *under the hood* as bibliotecas têm que ser implementadas para os diferentes kernels. No caso de Unix, as implementações usam as mesmas funções com que irão trabalhar nesta ficha!

Q1

Exemplo da ficha

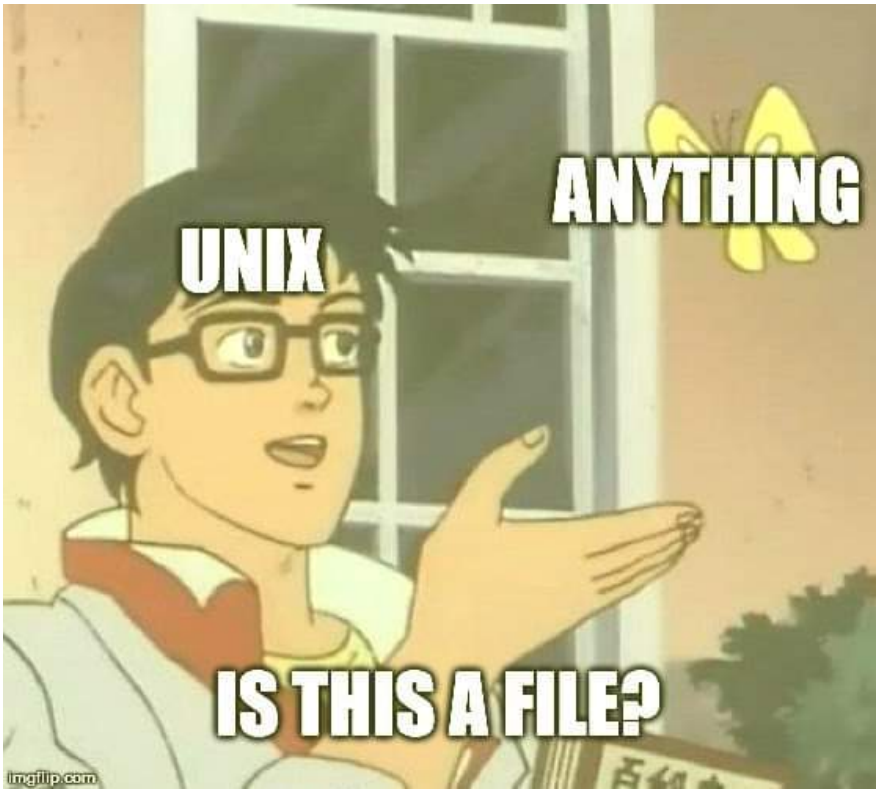
O exemplo é semelhante ao `cat` da ficha anterior, usando *system calls*. Se repararem, os passos são muito parecidos com que fazemos num programa standard de C (usando libc).

1. Abrir o ficheiro
   - Usa-se `open` em vez de `fopen`
   - O kernel distingue os ficheiros acedidos por um processo através de um *file descriptor*
     - Um identificador não negativo
     - O retorno do `open`, se bem sucedido, é o *file descriptor*
   - Vejam `man 2 open`
   - Enquanto que o modo de abertura no `fopen` é com strings (e.g. `"r"`, `"rw"`, ...), no `open` são valores númericos. Devem usar as flags/macros disponíveis.
     - As flags não são nada mais que valores númericos que permitem ativar certos bits.
     - O que significa que, para ativarem vários modos, têm que usar operações aritméticas bitwise. Como cada flag ativa um ou mais bits especificos, à partida irão querer sempre usar um OR através do operador `|`
     - Exemplo:
       - Modo de apenas leitura: `O_RDONLY` (0)
       - Modo de apenas escrita: `O_WRONLY` (1)
       - Modo de escrita + leitura: `O_RDWR` (2)
         - Notem que não é equivalente a `O_RDONLY | O_WRONLY`, pois isso dá 1, que é equivalente a `O_WRONLY`
       - Criar ficheiro se não existe, modo de escrita, e faz *append* caso o ficheiro já exista para não apagar o conteúdo atual: `O_CREAT | O_WRONLY | O_APPEND`
2. Leitura é feito com `read`
3. Escrita para a consola com `write`
   - Existem 3 ficheiros especiais em Unix para representar as streams de **input**, **output** e **erro**
     - Quando usam `printf`, este escreve as strings para a stream de *output*
     - Quando usam `scanf`, este lê da *input* stream.

    - Os *file descriptors* para cada uma destas streams têm valores reservados que são:
        - stdin: 0 (`STDIN_FILENO`)
        - stdout: 1 (`STDOUT_FILENO`)
        - stderr: 2 (`STDERR_FILENO`)
        - **No código, usem sempre as macros para ser legível!!**

- Uma nota sobre o `\0`
    - Quando usamos o `printf` e temos um placeholder para strings, `%s` , temos que passar uma string válida de C, i.e., tem que ter o `\0` para indicar a terminação da string
    - Ao ler o ficheiro, dependendo da função utilizada, esse `\0` não está incluido. Por isso, na ficha anterior, tinhamos que explicitamente inserir o `\0` após o `fread` , para que o buffer fosse uma string válida
    - As funções `read` e `write` tratam os dados de forma *raw*. Não interesse se é uma string, um número, uma imagem JPEG, ...
        - Ao fazer `write` dizemos para imprimir `N` bytes, e ele faz isso tal e qual.
        - Obviamente isto é também uma desvantagem. Se quiserem imprimir um número, não podem simplesmente enviar o binário do número para o STDOUT, pois o vosso terminal vai interpretar a informação como se fosse texto com um certo encoding (ASCII, utf-8, etc.)
        - Vejam o exemplo mais abaixo

- Curiosidade💡: em Unix é bastante comum tratar tudo como ficheiros para reduzir a complexidade das interfaces/APIs do kernel e ser possível utilisar as mesmas ferramentas em diversos contextos: https://en.wikipedia.org/wiki/Everything_is_a_file

4. Para fechar o ficheiro, usa-se `close` em vez de `fclose`

```c
int num = 0x0A61;
write(STDOUT_FILENO, &num, sizeof(int));

// no terminal vai aparecer um 'a' seguido de um linefeed. O número, na sua
representação hexadecimal, o '0A' é o código ASCII do linefeed (\n) e '61' é o
código ASCII do carater 'a'
```

Para terminar a explicação do exemplo: este lê o ficheiro por blocos, que é a maneira mais eficiente. A abordagem apresentada consiste em determinar o tamanho do ficheiro usando o `stat`. Sempre que é lido um novo bloco, `count` é decrementado e indica quantos bytes faltam ler. A função `next_block_size` calcula o tamanho do próximo bloco.

- Notem que não era preciso. Vocês podem sempre pedir ao `read` para ler `BUFFER_SIZE`. Se o ficheiro tiver menos de `BUFFER_SIZE` bytes para ler, não é causado nenhum erro. O `read` lê o que resta e retorna o número de bytes que leu.

Solução alternativa

No ficheiro `q1_alt.c` apresento uma abordagem alternativa onde não é preciso determinar o tamanho do ficheiro. Simplesmente uso o `read` e `write`, e meadiante o valor de retorno do `read` é possível determinar quando se chega ao fim do ficheiro e também apanhar erros que ocorram.

Gestão de erros em C

Por falar em erros, C tem uma abordagem muito simplista para gestão de erros nas *system calls*. Regra geral, as funções retornam `-1` para indicar que houve um erro e atribuem um código de erro a uma variável global `errno` (consultem `man errno`). Os códigos de erro estão associados a um tipo de erro especifico. Por exemplo, quando trabalham com I/O e tentam abrir um ficheiro que não existe, o valor de `errno` é `2`. Existem macros para cada código, e.g. `ENOENT` é equivalente ao valor `2`, e significa `No such file or directory`.

Uma forma de reportarem erros é utilisar as funções `perror` e `strerror`.

- `perror` imprime uma descrição do erro na stream de erro, i.e., `strerr`
- `strerror` retorna uma string com a descrição de erro
- A descrição de erro para um dado `errno` será a mesma com as duas funções. A vantagem de `strerror` é que permite customisar melhor o output, usando por exemplo o `fprintf`
  - `fprintf(stderr, "Cannot open file %s! Error: %s", argv[1], strerror(errno));`
- Notem que o `errno` pode mudar por cada chamada a função que fazem, quer *system calls*, quer *library calls*. Dependendo do contexto, poderá ser necessário guardar o valor

Q2

Permissões: overview

Os Sistemas Operativos modernos são multi-users e têm diversos programas, o que levanta questões de segurança. A informação co-existe no mesmo hardware (i.e. discos, memória RAM, ...), e por isso é

necessário um mecanismo para gerir quem pode aceder ao quê. Em Unix cada ficheiro (ficheiros regulares, diretórios, sockets, ...) tem informação acerca das suas permissões.

> *File permissions are defined for users, groups and others. User would be the username that you are logging in as. Furthermore, users can be organized into groups for better administration and control. Each user will belong to at least one default group. Others includes anyone the above categories exclude.*

Ou seja, cada ficheiro está associado a um utilizador e grupo, e as permissões são organizadas em três níveis:

- `user` : dizem respeito ao *owner* do ficheiro
- `group` : aplica-se a todos os utilizadores que pertencem ao grupo associado ao ficheiro
- `others` : qualquer outro utilizador que exista no sistema

Para cada nível existem três modos: read (r), write (w) e execute (x). As permissões são geridas de forma numérica, sendo que cada modo ativa um certo bit. Considerando os três modos, existem 8 combinações possíveis, sendo suficiente usar 3 bits.

- O formato dos bits é `rwx`
- Ou seja, `110` significa que tem permissões de leitura e escrita, mas não tem de execução
- **É conveniente pensar nas permissões usando a base octal!**

Considerando os três níveis, ao todo precisamos de 9 bits para representar estas permissões

- Existem mais uns bits na verdade, mas não são relevantes para já
- Assim, as permissões de um ficheiro são representadas como `rwx rwx rwx`, sendo a ordem dos níveis `user`, `group`, `other`.
- E.g., se considerarmos as seguintes permissões de um ficheiro `hello.c`: `rw- r-- ---`
  - O utilizador *owner* pode ler e escrever
  - Utilizadores associados ao grupo do ficheiro apenas podem ler
  - Qualquer outro user (others) não pode aceder ao ficheiro!

Quando utilizam `ls -l` ou o comando `stat` irão observar a utilização desta notação para indicar as permissões de cada ficheiro

- Notem que o primeiro carater é para reportar o tipo de ficheiro
  - O `-` indica que é um ficheiro regular
    - E.g. `- rw-r--r--. 1 fgaspar fgaspar 67050 mar 28 16:38 f4.pdf`
  - O `d` significa que é um diretório
    - `d rwxr-xr-x. 2 fgaspar fgaspar 4096 mar 31 19:45 q1`
    - 💡Food for thought: Então `q1` é um diretório e tem permissões de execução???
      - https://superuser.com/a/169418
      - `man chmod`
- Depois do tipo de ficheiro, têm então as permissões na notação indicada (rwx) na ordem `user`, `group`, `other`

Exercicio

Tendo em conta esta breve introdução, o objetivo do exercicio é vocês construirem o número que representa as permissões. Como disse anteriormente, as permissões são representadas por uma sequência de bits, e cada bit ativa um modo (`read`, `write`, `execute`) para cada nível (`user`,

`group`, `others`). Estes bits podem ser interpretados na base octal. Por exemplo, dizer que o `user` tem permissão 6, o binário equivalente é `110`, logo tem permissões de `read` e `write` (usando a outra notação, `rw-`).

As primeiras linhas do exemplo fazem parse do input, e extraiem o valor numérico para cada nível, que deve estar no intervalo [0,7].

- `operms`: others
- `gperms`: group
- `uperms`: user

Depois dependendo dos valores, são ativados certos bits. Por exemplo, se o valor de `uperms` é 5, o binário equivale a `101`, ou seja, `r-x`. Então fazemos uma operação bitwise na variável de permissões, `newperms`, para ativar os respetivos bits: `newperms |= S_IRUSR | S_IXUSR`

- reparem no sufixo USR => user

O mesmo se aplica para o `group` e `others`. Notem que o valor `101` para `group` teria o mesmo significado de `r-x`, no entanto os bits a ativar serão outros e como tal têm que usar as macros com o sufixo GRP: `newperms |= S_IRGRP | S_IXGRP`.

Isto é uma abordagem muito manual, mas é para perceberem como são geridas e representadas as permissões!

Solução 'lazy'

Visto que as permissões são representadas por um número, há soluções muito mais simples para este problema.

```
// imprimir a variável `newperms` após ter sido definida manualmente como no exemplo
printf("%o\n", newperms);
// criar o valor de permissões recorrendo a operações aritméticas bitwise
printf("%o\n", uperms << 6 | gperms << 3 | operms);
// cuidado ao utilizar o "atoi" para parsing, já que interpreta os números na base decimal
printf("%o\n", atoi(argv[1]));
// alternative seria o strtol, que interpreta na base que vocês quiserem, neste caso octal (8)
printf("%o\n", strtol(argv[1], NULL, 8));
```

Usando como referência o exemplo e as permissões `755`, o output dos printfs seria (`%o` interpreta os números na base octal):

```
755
755
1363
755
```

Como podem ver, extraindo os digitos das permissões para `uperms`, `gperms` e `operms` (e assumindo que estão no intervalo `[0,7]`) então basta fazer shifts e ORs aritméticos para construtir o valor de permissões. Se pensarem novamente na estrutura `rwx rwx rwx`, o shift de 6 mete os bits do *user* mais à esquerda, correspondendo ao primeiro grupo de `rwx`. O shift de 3 é para definir os bits do meio, i.e. do *group*. E os últimos bits, são para o *other* e não precisam de shift. **Sendo que**

estamos a usar números de 32 bits, mas apenas queremos definir os 9 bits menos significativos, é boa prática garantir que os restantes bits mais significativos estão a zero. Isso foi alcançado incializado a variável a zero.

Uma solução ainda mais lazy é utilizando o `strtol`, que converte a string para uma base que quisermos, neste caso, 8 (octal). Notem que o `atoi` interpreta o número na base decimal, consequentemente o binário resultante seria diferente do pretendido.

Q4

Existem pelo menos 3 opções para resolver este problema, que pode ser visto em dois sub-problemas:

1. Determinar se um ficheiro existe ou não
    - Se não existir, é só criar com o `open`
2. Alterar os metadados de um ficheiro que já existe

Alterar tempos acesso/modificação

Começando pelo segundo problema, basta utilizarem a função `utimes`. Notem que se o segundo parametro for `NULL`, o tempo atual é utilizado para as datas de acesso e modificação, poupando a necessidade de usar outras *system calls* para obter esse tempo!

```
utimes(fname, NULL);
```

Determinar se um ficheiro existe

Solução 1: `stat()`

Uma eventual abordagem é utilizar o `stat`, que irá falhar caso o ficheiro não exista. Nesse caso, irá retornar `-1` e irá escrever na variável `errno` o código de erro. Notem que existem vários motivos para o `stat` falhar, pelo que devem verificar o código de erro é `ENOENT` para assegurar que o erro deve-se ao pathname não existir.

```
struct stat s;
if (stat(fname, &s) == -1)
{
    // stat failed, ensure the error is due pathname not existing
    if (errno != ENOENT)
    {
        fprintf(stderr, "Cannot stat file '%s'. Error: %s\n", fname,
strerror(errno));
        return EXIT_FAILURE;
    }

    // stat failed because file does not exist, so create it
    // ...
} else {
    // stat was successful (pathname exists)
    // ensure it is a regular file
```

```
    if (!S_ISREG(s.st_mode))
    {
        fprintf(stderr, "The pathname '%s' is not a regular file\n", fname);
        return EXIT_FAILURE;
    }

    // change the access/modification times with 'utimes'
}
```

Outro cuidado a ter é que o `stat` pode ser usado em qualquer tipo de ficheiro, sejam regulares, diretórios, etc. Neste caso não seria critico que alterassem a data de modificação de um diretório. No entanto, o programa `mytouch` apenas deve manipular ficheiros regulares, e pode haver outros casos em que tenham que ter este cuidado. Como fazer essa verificação? A informação contida na `struct stat` tem um atributo, `st_mode`, que contém bits para indicar o tipo de ficheiro. Vejam `man 2 stat` e `man inode`. Segue um exemplo ilustrativo, que usa macros definidas para confirmar que se trata de um ficheiro regular, diretório, etc.

```
struct stat s;
stat(fname, &s);

if (S_ISREG(s.st_mode)) {
    // ficheiro regular
} else if (S_ISDIR(s.st_mode)) {
    // diretório
} else if (...) {
    ...
} ...
```

Solução 2: `access()`

> *access() checks whether the calling process can access the file pathname. If pathname is a symbolic link, it is dereferenced.*
>
> *The mode specifies the accessibility check(s) to be performed, and is either the value F_OK, or a mask consisting [...]. F_OK tests for the existence of the file.*

Esta abordagem herda o mesmo problema do `stat`. Verifica a existência de um ficheiro, mas não é necessariamente um ficheiro regular (ou seja, se for um diretórito, retornaria OK)! No entanto, com o `stat()` conseguimos fazer essa confirmação, usando a informação retornada na struct. Com o `access`, isso não é possível diretamente. Teriamos que chamar o `stat()`.

Assumindo que não interessa o tipo de ficheiro, o `access()` tem a vantagem de ter menos overhead. Segue um exemplo:

```
if (access(fname, F_OK) == -1) {
    // access failed, ensure the error is due pathname not existing
    if (errno != ENOENT) {
        fprintf(stderr, "Cannot stat file '%s'. Error: %s\n", fname,
strerror(errno));
        return EXIT_FAILURE;
    }
```

```
    // create the file with 'open'...
} else {
    // file exists, but is not necessarially a regular file!
    // change the times with 'utimes'...
}
```

Solução 3: Handling race conditions ⚠

O problema das abordagens com `stat` ou `access` é que a verificação de existência do ficheiro e a criação do mesmo são feitas em *system calls* distintas. Entre a chamada do `stat` / `access` e do `open`, o estado do sistema pode mudar. Como tal, quando o vosso programa invoca o `open`, achando que o ficheiro não existe, há a possibilidade de o ficheiro ter sido criado entretanto por outro processo.

- Explorar estas questões temporais é bastante comum
- https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use

Uma possível solução será usar o `open` para criar o ficheiro, mas forçar que dê erro caso o ficheiro já exista

- Isto é alcançado acrescentando a flag `O_EXCL`
- `open(fname, O_CREAT | O_EXCL ..., 0644)`

> *O_EXCL: Ensure that this call creates the file: if this flag is specified in conjunction with O_CREAT, and pathname already exists, then open() fails with the error EEXIST.*

Portanto, caso o retorno de `open()` seja `-1` e o valor de `errno` corresponde a `EEXIST`, então significa que o ficheiro existe e bastará alterar o tempo de acesso/modificação. Por outro lado, se o retorno for `0`, então o ficheiro foi criado com sucesso. Qualquer outro cenário representa um outro erro durante a *system call*.

Extras

O `man` é vosso melhor amigo!

Recomendo vivamente que se habituem a usar o `man`. Relembro que em contexto de avaliação, não há internet, mas há o nosso *lord and savior* `man`!

Algumas dicas:

- O man está organizado em secções. Isto é muito importante porque um comando/função como `stat` pode existem em várias secções
  - Secção 1: Comandos da shell. Quando usam comandos como `ls`, `grep` e outros, é nesta secção que encontram a respetiva documentação
  - Secção 2: *System calls*, a secção fundamental para esta ficha!
  - Secção 3: Funções disponíveis nas bibliotecas, é aqui que encontram informação da *libc*, por exemplo, `fopen`, `printf`. Mas também há documentação para outras bibliotecas, como *threads*, que eventualmente iremos usar!
  - Imaginem que querem documentação para a *system call* `stat`. Ora `stat` não só é uma *system call*, como também um comando da shell. Por isso fazer `man stat` vai-vos mostrar documentação para o comando, pois é o primeiro *match*.

        - Reparem na primeira linha. Vão ver algo como `STAT(1)`, o que significa que estão a ver a secção 1
        - Para verem secções que documentam o `stat`, usem a opção `-f` (aka find)
            - `man -f stat`. E vão ver que existe um match nas secções 1, 2 e talvez 3
        - Para abrirem uma secção especifica, usem `man <numero de secção> <nome>`.
            - Como queremos docs. para a *system call*, então fazemos `man 2 stat`.
    - Também podem pesquisar no `man`, embora não seja a melhor experiência de sempre porque funciona por *word matching*.
        - Usar opção `-k`
        - O padrão é regex, e por isso é melhor meter entre "", para que alguns carateres especiais como `*` não sejam interpretados pela shell
- Bonus: Se têm memória de peixe como eu, façam `man man`