# Sistemas de Operação 2016/2017



# Trabalho I: Mini-Shell

## Objetivo

O objetivo principal deste trabalho é fornecer aos alunos um primeiro contacto com programação avançada em Linux tendo como base a linguagem C. Em particular, pretende-se que os alunos sejam capazes de lidar com argumentos passados através da linha de comando e com chamadas ao sistema para manipulação de processos, tais como: `fork()`, `pipe()`, `dup2()`, `exec()`, `wait()`, `exit()`, entre outras.

## Trabalho

O trabalho é composto por um conjunto de pequenas etapas que no conjunto levarão à implementação de uma mini-shell. Em nenhuma etapa deve fazer uso das chamadas ao sistema `system()` ou `popen()`. Como alternativa, deve recorrer sempre às chamadas ao sistema `fork()`, `dup2()` e família de funções `exec()`. A implementação de cada etapa deve ser o mais robusta possível, i.e., se uma chamada a uma função do sistema falhar, o programa deve apresentar uma mensagem de erro e terminar com um código de *exit* apropriado. Para tal, procure tirar partido das funções `perror()` ou `strerror()` para gerar mensagens de erro apropriadas.

Comece por considerar o exercício da segunda aula prática para implementar uma linha de comandos, adaptado tal como no ficheiro my_prompt.c, em que se incluíram as seguintes alterações:

1. Um novo campo `next` na estrutura `COMMAND` para fazer o encadeamento de comandos.
2. Uma variável global `inputfile` para guardar o nome de ficheiro em caso de redireccionamento da entrada padrão.
3. Uma variável global `outputfile` para guardar nome de ficheiro em caso de redireccionamento da saída padrão.
4. Uma variável global `background_exec` para indicação de execução concorrente com a mini-shell.
5. A implementação completa da função `parse()` disponível no ficheiro parser.c. Esta função (i) cria uma lista ligada de comandos (nós do tipo `COMMAND`), obtida pelo parsing da string `linha`; (ii) preenche convenientemente as variáveis globais `inputfile`, `outputfile` e `background_exec`; e (iii) retorna um apontador para o primeiro elemento da lista.
6. A implementação completa da função `print_parse()` que permite visualizar o resultado da função `parse()`.
7. A adição das funções `execute_commands()` e `free_commlist()`, as quais deverá completar de acordo com as etapas que se seguem.

## Etapas do Trabalho

### Etapa 1: Parsing

Comece por compilar o programa e experimente uma série de comandos com/sem pipes (|), redirecionamentos (< e >) e execução em background (&) de modo a compreender o funcionamento das novas variáveis globais e o encadeamento de comandos (estruturas `COMMAND`) gerado pela função `parse()`. Implemente a função `free_commlist()` que deverá libertar a memória alocada para a lista ligada de comandos.

### Etapa 2: Comandos Simples

Estenda a mini-shell de forma a executar comandos do utilizador. Comece por considerar apenas comandos sem argumentos e depois admita argumentos para os comandos. Assuma que o número máximo de argumentos para um comando é 100 (`#define MAXARGS 100`). Exemplos:

- `ls`
- `date`
- `more nomeficheiro`
- `ls -l /tmp`
- `gcc -s -O -g -o pois pois.c`

### Etapa 3: Redirecionamento de Input

Estenda a mini-shell para lidar com redirecionamento simples de input. O input para o comando (parte antes de `<`) deve ser lido do ficheiro em vez do standard input. Se o ficheiro não puder ser lido, deve ser gerado um erro e o comando não deve ser executado. Exemplos:

- `more < ficheiro_in`
- `more<ficheiro_in`
- `wc < ficheiro_in`
- `wc < ficheiro_in -l`

### Etapa 4: Redirecionamento de Output

Estenda a mini-shell para lidar com redirecionamento simples de output. O output que o comando (parte antes de `>`) gera deve ser enviado para o ficheiro em vez de para o standard output. Se o ficheiro já existir, deve ser re-escrito. Se o ficheiro não puder ser criado, deve ser gerado um erro e o comando não deve ser executado. Exemplos:

- `ls > ficheiro_out`
- `ls>ficheiro_out`
- `ls > ficheiro_out -l`
- `ps -aux > ficheiro_out`

### Etapa 5: Encadeamento de Comandos

Estenda a mini-shell de forma a suportar o encadeamento de comandos (uso de pipes). O output que o comando inicial (parte antes de `|`) gera deve ser enviado para uma pipe em vez de para o standard output. Por sua vez, o segundo comando (parte depois de `|`) deve tomar o seu input a partir da pipe utilizada pelo primeiro comando para enviar o seu output. Numa segunda fase estenda a mini-shell de forma a que possa encadear mais do que uma pipe na linha de comandos. Exemplos:

- `ls -l | more`
- `ls -l|more`
- `ls -l | wc -l`
- `ps -ef | grep bash | wc`

### Etapa 6: Concorrência

Estenda a mini-shell para lidar com concorrência de processos. Se o caracter `&` for o último caracter da linha de comandos (variável global `background_exec == 1`), a shell deverá executar o comando (parte antes do `&`) concorrentemente com a própria shell, ou seja, a shell não deverá esperar que o comando em causa termine para apresentar a prompt e desse modo aceitar novos comandos. Exemplos:

- `gcc -o pois pois.c &`
- `emacs &`
- `emacs&`

### Etapa 7: Junção Final

Caso tenha resolvido separadamente as funcionalidades das etapas anteriores, junte agora tudo num só programa de forma a que a mini-shell aceite a combinação dos vários comandos. Exemplos:

- `ls -l /tmp > ficheiro_in &`
- `grep ola < ficheiro_in > ficheiro_out`
- `ps -aux | grep root`
- `grep ola < ficheiro_in | wc > ficheiro_out &`