1. Processos

- Processo é um programa em execução no sistema operativo
- A execução consiste em executar instruções no CPU de forma sequencial
- Um SO moderno é capaz de gerir vários processos ao mesmo tempo
 - Em cada núcleo/CPU, executa um processo de cada vez
 - O scheduler do SO está constantemente a alternar entre processos, ou seja, a cada processo é dada uma oportunidade de executar algumas intruções durante algum tempo, até que é dada oportunidade a outro processo
 - Perceção de responsividade para o utilizador
 - Maximizar a utilização de CPU
 - E.g., um processo pode estar pendente de aceder a memória RAM ou a um dispositivo I/O
 - Durante esse período o processo está bloqueado, sem fazer nada de útil
 - Logo, podemos dar oportunidade a outro processo num estado de "Ready"
 - A ordem de execução dos processos depende de um algoritmo do scheduler
 - Varia de implementação p/ implementação
 - Depende de vários fatores...
 - Logo, não devemos assumir nada sobre a ordem de execução dos processos
- Um processo é caraterizado por
 - Estado (a executar? bloqueado à espera de algo? pronto a executar?)
 - ID (process identitifer)
 - Número > 0
 - Alguns PIDs s\u00e3o reservados, e.g., o primeiro processo a arrancar (init em alguns sistemas) tem PID 1 (invocado automaticamente pelo kernel no fim da sua iniciliza\u00e7\u00e3o)
 - Se consultarem a lista de processos, talvez o processo com PID 1 não se chame init , mas sim systemd
 - De qualquer forma, deve existir um executável com o nome init
 - \$ ls -l /sbin/init
 - lrwxrwxrwx. 1 root root 22 jun 16 18:47 /sbin/init ->
 ../lib/systemd/systemd
 - Os PIDs são reutilizáveis
 - Memória/dados
 - Descritores de ficheiros abertos
 - o ...
- Os processos são organizados hierarquicamente; cada processo tem informação sobre:
 - O seu parent, o processo que criou este (apenas 1)
 - Os processos criados por este, ou seja, os seus childs (0 ou mais)

2. Layout de memória de um processo

O layout exato e ordem podem variar, mas de forma geral é o seguinte:

- 1. Argumentos (lembrem-se do argv , parametro da função main) e variáveis de ambiente
- 2. Stack, que é dividida em stack frames; para cada invocação de uma função, é criada uma nova stack frame que preserva o estado de todas as variáveis locais, endereço de retorno, outras informações relativas ao hardware (i.e., registos)
- 3. Heap, onde é feita a gestão manual de memória dinâmica (malloc, realloc, ...)
- 4. Segmento dados não inicializados (BSS); variáveis globais não inicializadas no código
- 5. Segmento dados inicializados; variáveis globais inicializadas
- 6. Segmento de texto, contém as instruções máquina a serem executadas no CPU

3. Criação de processos

- Em Unix, system call fork()
- O processo que invoca o fork() é designado parent
- O processo criado é designado child
- O child é uma cópia do processo parent (sort of...)
 - Nova entrada na tabela de processos, logo child tem um novo PID
 - O child tem uma cópia do espaço de memória do parent no instante do fork()
 - Espaço de memória inclui a stack, heap, segmento de texto para instruções,
 - É uma cópia!! Logo, parent e child não partilham estas regiões de memória
 - \triangle O child herda todos os file descriptors abertos pelo parent antes do fork()
 - Implica que, se tanto o child como o parent usarem o mesmo descritor para ler um ficheiro, o conteúdo que obtêm é intercalado
 - Notem que parte da informação relativa a um ficheiro é o offset (posição atual no ficheiro)
 - Se o parent faz um read(), o offset do ficheiro avança N bytes. Esse avanço reflete-se no child e vice versa
 - Para demonstrar este comportamento, vejam demos/main.c e demos/random.txt

```
$ cd f5/demos/
$ make fd
$ ./fd
```

- Quando o fork() termina, passamos a ter dois processos, child e parent, que continuam a sua execução de forma independente
 - NOTA: não podem assumir nada sobre a ordem de execução...
 - Já que child é uma cópia de parent, significa que no instante imediatamente após o fork() os processos têm carateristicas comuns
 - O segmento de texto, ou o conjunto de instruções do processo, são iguais
 - O Program Counter, ou a instrução a ser executada, é a mesma
 - Do ponto de vista do código, têm que imaginar que tanto o parent como o child vão fazer a ação imediatamente a seguir à invocação do fork()
 - E.g., guardar o valor de retorno numa dada variável
- Tanto o child como o parent executam o mesmo código... então, como consigo diferenciar entre os dois?

- Não parece muito útil clonar um processo para fazerem exatamente a mesma computação sobre os mesmos dados
- Geralmente queremos que o parent e o child façam tarefas distintas
- A forma de distinguir é através do retorno do fork()
 - parent: o valor de retorno é o PID do seu processo filho
 - child: o valor de retorno é zero
 - NOTA: o fork(), pode falhar, e nesse caso retornará -1

```
int pid = fork();

if (pid == -1) {
    // ERROR, child processo was not creater
} else if (pid == 0) {
    // I am the child process
} else {
    // I am the parent process
}
```

3.1. parent é uma cópia de child

Esta afirmação não é 100% correta! Isso implicaria que toda a informação que carateriza um processo seria simplesmente clonado, e já vimos que isso não é o caso. Por exemplo:

- o child tem um PID diferente e único
- o parent PID é naturalmente diferente nos processos parent e child
- ...

Existem outros detalhes que não são propriamente herdados ou copiados no processo filho. Por curiosidade, podes ver o man 2 fork .

3.2 O espaço de memória do *parent* é duplicado para o processo *child*

Do nosso ponto de vista, enquanto programadores, podemos dizer que sim! Em termos práticos, isto seria super ineficiente e por vezes inútil. Por exemplo, o *parent* e *child* executam a mesma sequência de instruções. Até que ponto faz sentido duplicar o segmento de texto? Não seria mais eficiente o *child* e *parent* partilharem esse segmento de memória?

Claro! Faz todo o sentido e os kernels mais modernos à partida fazem isso mesmo :)

Também existe uma técnica chamada <u>Copy-on-write (COW)</u>, que é uma abordagem *lazy* de duplicar os segmentos de memória do processo *parent*. Em vez de fazer logo uma cópia, os dois processos partilham os segmentos de memória. A cópia é atrasada até que um dos processos faça uma operação de escrita. Nesse caso, é mesmo necessário fazer a cópia, já que o estado desse pedaço de memória diverge nos dois processos.

3.3. Quando é que faz sentido lançar um novo processo?

Dependendo da aplicação que estamos a desenvolver isto pode ser dar jeito em duas vertentes:

- 1. O vosso programa processa tarefas que são independentes. Por um exemplo, um servidor.
 - (A) Ser capaz de receber os pedidos Web atráves de um socket, o que implica estar à escuta de novos pedidos em qualquer instante.
 - (B) Processar o pedido Web (atualizar valores na base de dados, carregar uma imagem ou página HTML, etc).
 - As tarefas (A) e (B) são independentes e como tal faz todo ter:
 - um processo principal a receber os pedidos
 - vários outros processos a tratar dos pedidos que vão chegando
 - Uma das vantagens desta abordagem é que todos estes processos executam de forma concorrente nos vários CPUs, aumentando assim a capacidade de processamento da aplicação
- Executar outro programa. Por exemplo, uma shell (terminal), é um programa que interpreta os vossos comandos e grande parte das vezes irá lançar processos para responder ao vosso comando
 - Compilar ficheiros com o gcc
 - Executar os programas que desenvolveram
 - Listar ficheiros com ls
 - o ...

4. Exercicios

01

Ao todo temos 8 processos, incluindo o principal/main

- 1. O processo principal, main, faz o primeiro fork. Passam a existir dois processos: main e child1
- 2. O main e o child1 avançam para a próxima instrução, o segundo fork() . Assim, cada processo, main e child1, criam um processo cada: child2 e child3
- 3. Por fim, os 4 processos em execução vão todos fazer mais um fork(), criando mais 4 processos: child4, child5, ...

Alguns comentários adicionais

Imaginem que adicionam printf("after 1st fork, pid: %d. My parent is, pid: %d\n",
getpid(), getppid()); após cada fork();

- O printf irá mostrar o identificador do processo que chama o printf
- e também o identificador do seu processo parent

```
after 1st fork, pid: 356614. My parent is, pid: 341477
after 1st fork, pid: 356615. My parent is, pid: 356614
after 2nd fork, pid: 356614. My parent is, pid: 341477
after 2nd fork, pid: 356615. My parent is, pid: 356614
after 3rd fork, pid: 356614. My parent is, pid: 341477
after 3rd fork, pid: 356615. My parent is, pid: 341477
after 3rd fork, pid: 356615. My parent is, pid: 356614
after 3rd fork, pid: 356618. My parent is, pid: 4270
after 2nd fork, pid: 356617. My parent is, pid: 4270
after 2nd fork, pid: 356616. My parent is, pid: 4270
after 3rd fork, pid: 356616. My parent is, pid: 4270
after 3rd fork, pid: 356616. My parent is, pid: 4270
after 3rd fork, pid: 356616. My parent is, pid: 4270
```

```
after 3rd fork, pid: 356621. My parent is, pid: 4270
after 3rd fork, pid: 356619. My parent is, pid: 4270
after 3rd fork, pid: 356620. My parent is, pid: 4270
```

Algumas observações:

- 1. A ordem dos prints é algo aleatória. Por exemplo, a certa altura há processos a reportar que já fizeram o seu terceiro fork, enquanto outros ainda estão a fazer o seu segundo fork
 - Isto tem haver com o escalonamento/scheduling de processos do sistema operativo
 - Não podem assumir que um certo processo corre primeiro que outro processo
 - Cuidado com as race conditions...

2. Os identificadores dos processos são incrementais e de ordem ascedente?

- **Não devem assumir que será sempre assim**. Por outras palavras, não tentem adivinhar o PID do processo criado após um fork()
- Depende da configuração do kernel, de quantos processos estão ativos, de outros processos que possam ser criados entre as chamadas fork() no vosso programa, ...
- Se precisam de saber o identificador do processo filho, devem guardar o retorno do fork()
- A única coisa que podem assumir é que os PID são números maiores que zero

3. Porque é que alguns processos têm como parent um processo cujo PID é 4270?

- Notem que a forma como os forks s\u00e3o feitos implica que os processos n\u00e3o esperam pelos seus child
- Então pode acontecer o processo inicial, que neste caso é o 356614 , terminar mas os seus filhos ainda estarem a executar
- Quando isto acontece, os filhos designam-se de orfãos e ficam associados a um processo especial de boot/inicialização, que é o primeiro processo a arrancar (+/-)
 - Geralmente designa-se por init , e tem por defeito o PID 1
 - https://en.wikipedia.org/wiki/Init
 - Mas existem outras implementações, como systemd
 - https://en.wikipedia.org/wiki/Systemd
- Como podem ver no bloco em baixo, o PID 4270 corresponde ao
 /usr/lib/systemd/systemd , que foi lançado pelo init com PID=1. Portanto isto
 pode variar um bocado de sistema para sistema, mas efetivamente todos os
 processos orfãos ficam associados a um processo do sistema. Só não tem que ser
 necessariamente o init como poderão ler na literatura
 - Quote do POSIX: https://stackoverflow.com/a/40424702
- Já agora, notem que o processo inicial, PID=356614, tem como parent PID=341477.
 Neste caso, corresponde à shell/terminal onde o programa foi executado

Q2

Parecido com o anterior, mas desta vez temos o fork() dentro de um loop. São criados 16 processos.

- Apenas têm que perceber que quando um processo faz o fork(), o processo child herda o valor do contador i do parent
 - No entanto, cada processo tem a sua própria cópia da variável i ,
- A instrução seguinte ao fork() será i++ , logo o child faz menos uma iteração no total que o parent

llustração em árvore da criação dos processos, e com indicação do valor inicial de i em cada novo processo *child*

- · main, faz 4 forks
 - child(i=0) faz 3 forks
 - child(i=1) faz 2 forks
 - child(i=2) faz 1 fork
 - child(i=3) faz 0 fork
 - child(i=3) faz 0 fork
 - child(i=2) faz 1 fork
 - child(i=3) faz 0 forks
 - child(i=3) faz 0 forks
 - child(i=1) faz 2 forks
 - child(i=2) faz 1 fork
 - child(i=3) faz 0 forks
 - child(i=3) faz 0 forks
 - child(i=2) faz 1 fork
 - child(i=3) faz 0 forks
 - child(i=3) faz 0 forks

Q3

Overview

O exemplo mostra duas coisas:

- Uma possível abordagem para que o processo child e parent executem regiões de código distintas
- Como é que o parent pode esperar que o processo child termine

Sobre o primeiro ponto, isso é possível com base no valor de retorno do <code>fork()</code> . Após a chamada da system call é criada uma cópia do processo parent, o child. Embora os dois processos tenham a mesma sequência de instruções, os valores de retorno do <code>fork()</code> irão ser diferentes para cada processo:

- O processo child recebe o valor 0
- O processo parent recebe um valor > 0, que corresponde ao PID do child

Então, uma forma de diferenciar entre o *parent* e o *child* no código é verificar o valor de retorno de fork().

```
if ((pid = fork()) == -1) {
    perror("fork");
    return EXIT_FAILURE;
}
```

```
else if (pid == 0) {
    /* child process */
    /* ... */
}
else {
    /* parent process */
    /* ... */
}
```

No exemplo acima, retorno do fork() é guardado na variável pid . Quando a função retorna, tanto o *parent* como o *child* executam esta sequência:

- pid == -1 ? Se sim, imprime a mensagem de erro
 - Nota que se a system call tivesse falhado, o child não existiria. Portanto esta condição só executada no parent
- pid == 0 ?
 - No caso do processo child a condição será verdade, e vai executar o corpo do else
 if (pid == 0).
 - No parent não é, e portanto executa apenas o corpo do else.

Explicação do output

Relativamente ao output, irão obter algo deste género:

```
CHILD: value = 1, addr = 0x7fff78223258

PARENT: value = 0, addr = 0x7fff78223258
```

Valor de value

- Embora o processo child seja uma cópia do processo parent, o que inclui o espaço de memória, os dois processos são independentes. Após o fork(), os estados de cada processo e valores na memória são independentes, i.e., cada processo tem o seu espaço de memória
 - Quando se diz que child começa com uma cópia da memória do pai isso inclui todo o segmento de memória: Stack, heap, segmentos de dados/texto, ...
- No instante do fork(), o valor de value é 0 no parent, então também o child vai ter value a 0
- Depois o child executa a instrução value=1. Esta atribuição não afeta a variável value em parent, pois os processos são independentes e têm espaços de memória separados! O value de child estará numa posição de memória física diferente do value de parent
- Logo, child imprime 1, parent imprime 0

Endereços iguais???

- Algo que pode ser confuso (e interessante), é que os endereços de memória da variável value são iguais nos dois processos
 - Estes endereços são endereços de memória virtuais!
 - O mapeamento para endereços de memória físicos serão endereços de memória distintos em cada processo
- É bastante conveniente que os endereços virtuais sejam preservados no processo child

```
int value = 0;
int ptr = &value;
/* ... */
int pid = fork();
if (pid == 0) {
    /* child process */
    *ptr = 10;
}
```

- Considera o exemplo acima. Antes do fork(), o parent guarda o endereço da variável value em ptr. Supõe que é 0x123.
- Considera a hipótese em que após o fork() o endereço virtual da variável value seria diferente no processo *child*, passando a ser 0x444.
- No entanto, o valor da variável ptr no child continuaria a ser 0x123. Ou seja, o conteúdo das variáveis foi preservado, mas foram atribúidos novos endereços virtuais a cada variável
- Quando o child tentasse desreferenciar ptr, estaria a aceder a um espaço de memória que não corresponderia à variável value
 - E provavelmente resultaria em Segmentation Fault.
- Assim, é bastante útil que os **endereços virtuais** sejam preservados após o fork
 - Embora, apesar de serem iguais, mapeiam para endereços físicos distintos em ambos os processos *child* e *parent*

Q4

Este é um exemplo em que se faz fork() para lançar um novo processo, mas é executado outro programa que esteja disponível no sistema como executável

- · Um comando da shell
- · Um script
- ...

Para tal é usada uma system call exec

- Não existe exec , mas existe execlp , execv , ... São 6 ou 7 :)
- · Para facilitar, vou designar todas como exec

Quando vocês fazem o exec , toda a memória do processo é escrita por cima com as instruções do programa que lançaram. Portanto, o código que vocês têm após a *system call* exec nunca será executado, exceto caso haja um erro!

Notem que o processo é mesmo. Simplesmente já não vai executar o código que vocês têm no vosso .c , mas sim o comando que indicaram, e.g, ls . Inclusivé, o PID é preservado!

```
/* ... */
else if (pid == 0) {
    /* child process */
    if (execlp(argv[1],argv[1],NULL) == -1) {
        /* apenas executado se falhar, caso contrário o processo _child_ criado com
    o fork é totalmente sobrescrito, mas o processo é o mesmo, e o PID é preservado */
        perror("execlp");
        return EXIT_FAILURE;
```

```
}
}
/* ... */
```

Se a função execlp executa com sucesso, como é que o processo filho sinaliza o seu término ao processo pai?

A resposta a esta pergunta é que o *parent* usa a função waitpid , que faz com que o *parent* bloqueie e não execute nada até o *child* terminar.

- É possível fazer com que o waitpid não bloqueie
- Há outras circustâncias em que o waitpid pode retornar sem que o filho tenha terminado
- · Leiam o man waitpid com atenção!

Q5

Porque é que não é possível executar comandos com argumentos, e.g., ls -l ou uname -n?

Neste caso o programa irá falhar, pois o valor da variável command é literalmente "ls -l", e não existe nenhum comando deste género.

```
./myshell: couldn't exec ls -l: No such file or directory
```

Para que funcionasse, a system call teria que ser:

```
/* child */
execlp("ls", "ls", "-l", (char *)NULL);
```

- O primeiro argumento, "ls", é o nome do executável
- Todos os argumentos seguintes correspondem aos argumentos a ser passados ao programa.
 - É basicamente especificar o array argsv que vocês recebem na função main dos vossos programas
 - Notem que o primeiro argumento é também "ls"
 - Por convenção, o primeiro argumento --- correspondente ao argv[0] --- é o nome do executável. Devem respeitar essa convenção, embora pudessem passar outro valor qualquer...
 - ♠ Para indicar o fim da lista de argumentos para o programa, devem usar NULL , mas é fundamental que façam o cast para char * . Vejam man execlp
 - Caso tenham interesse, <u>Variable Argument Lists in C using va_list</u>

06

O truque para este exercicio é usar as funções da familia exec que suportem um array de strings para especificar o número de argumentos, já que utilizando a sintaxe anterior teria que ser hard-coded...

Se consultarem o \max , irão ver que as funções da familia \exp com o sufixo v todas aceitam um array de strings. O equivalente a \exp seria então \exp .

```
int execl(const char *pathname, const char *arg, ... /* (char *) NULL */);
int execv(const char *pathname, char *const argv[]);
```

De resto, é trivial :) É só usar o strtok para preparar o array de strings com os argumentos todos. Não esquecer que o argv[0] , por convenção, deve ser o nome do executável. E não se esqueçam também que o array deve ter um NULL para indicar o fim da lista de argumentos.

```
v - execv(), execvp(), execvpe()
```

The char *const argv[] argument is an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers must be terminated by a null pointer.

Por isso têm que criar um array estático ou dinâmico de char * , e depois do último argumento, garantir que o valor é NULL .

Pitfalls

Algumas notas extra para alguns cuidados a ter no contexto deste exercicio, baseando no que fui vendo nas aulas.

Array de strings

Uma das formas para declararem um array de strings, cujas strings têm tamanhos fixos, seria uma matriz.

Em baixo têm duas formas equivalentes onde é declarado um array de strings com tamanho máximo de 25 carateres. Os primeiros valores são inicializados com "ls", -l e o resto do array, como não é especificado, é automaticamente inicializado com zeros, o que equivale a string vazias, i.e., o primeiro char é \0.

```
char args[10][25] = {
    "ls",
    "-l",
};
```

```
char args[10][25] = {
          {'l', 's', '\0'},
          {'-', 'l', '\0'},
};
```

Este tipo de abordagem é limitada, pois a matriz é fixa. É um bloco contiguo de memória de 10 * 25 bytes. Eu posso escrever nas células da matriz e interpretar cada linha da matriz como se fossem strings. Mas não posso "romper" o bloco de memória. E.g., meter args[2] a apontar para NULL ou qualquer outro endereço. A região de memória associada a args[2] é fixa, apenas posso alterar o conteúdo dessa região. Esta limitação impede que vocês usem este tipo de declaração com o execvp já que requer que a última string seja NULL . Notem que "" não é a mesma coisa que NULL !

Então a alternativa teria que ser declarar um array de char * .

```
char *args[10] = {
    "ls",
```

```
"-l"
};
```

Assim temos a flexbilidade de poder alterar endereços contidos nas posições de args , e as strings deixam de estar limitadas a 25 carateres. Notem que com este tipo de inicialização, todas as posições a partir de args[2] estão inicializadas a zero, o que é equivalente a NULL .

```
// inicializa todas as posições do array a zero
char *args[10] = {};

// primeiros elementos são "ls", "-l" e o resto é inicializado a zero
char *args[10] = {
    "ls",
    "-l"
};

// a memória vai ter "lixo"!!
char *args[10];
```

Função strtok

Nas aulas eu fui alertando para o facto do strtok não alocar memória, e por isso vocês teriam que copiar os tokens para memória gerida por vocês (fazer malloc, strcpy, ...). Na verdade, não é assim tão linear. Por isso vou tentar clarificar o funcionamento do strtok, e cabe a vocês avaliarem se podem ou não usar os retornos do strtok para construir o array de argumentos a ser usado no execvp, porque efetivamente depende 😂

O primeiro problema do strtok é que vai modificar a string original. Sempre que é encontrado um delimitador, ele vai escrever um \0 nessa posição e retorna o endereço para o inicio da substring/token, endereço esse que faz parte da string passada por parametro. Esse endereço corresponde:

- à posição 0 da string, na primeira chamada
- à posição imediatamente a seguir ao último delimitador encontrado

```
char str[] = "isto, é, um exemplo!"

strtok(str, ","); // retorna endereço para o inicio da string "isto", mete um '\0'
na 1ª vírgula
strtok(NULL, ","); // retorna endereço para o inicio da string " é", mete um '\0' na
2ª vírgula
strtok(NULL, ","); // retorna endereço para o inicio da string " um exemplo!",
chegou ao fim da string
```

Portanto a string original ia ficar assim: "isto\0 é\0 um exemplo!".

Dependendo do contexto, vocês podem ou não usar diretamente os retornos do strtok para construir o vosso array de strings. Só têm que se lembrar que se a string que vocês passam por parâmetro ao strtok deixar de existir, os endereços retornados pelo strtok serão inválidos eventualmente!

Para demonstrar isto, vejam o exemplo demos/strtok.c.

```
$ make strtok
$ ./strtok

String original: 0x9d22a0
Primeiro token: 0x9d22a0
arg 0: 'ls'
arg 1: '--all'
arg 2: '-l'
arg 3: '--human-readable'
...
```

Como podem ver, o endereço da string e do primeiro token são iguais.

Agora experimentem descomentar o free(str) na linha 30. Como a string str, passada ao strtok, vai ser apagada, os endereços guardados no array args deixam de apontar para posições de memória válidas.

```
$ make strtok
$ ./strtok

String original: 0xe022a0
Primeiro token: 0xe022a0
arg 0: ''
arg 1: ''
arg 2: ' *'
arg 3: ''
```

Em suma, quando usam o strktok :

- 1. Devem assumir que a string passada ao strtok para extratir tokens vai ser modificada! Devem copiar a string caso necessitem de preservar a original
- 2. Devem ter cuidado ao usar diretamente as strings retornadas pelo strtok , já que este retorna endereços da string passada por parâmetro. Se esta string deixar de existir, os tokens também deixam de existir.

Na minha implementação, q6/q6.c, não é preocupante se a variável command é alterada. Além disso, sei que essa variável estará definida até invocar o execvp, pelo que optei por não alocar espaço para os tokens.