UNIVERSIDADE ABERTA
www.uab.pt

Código 21111

Sistemas Operativos

(ano letivo 2017-18)

Laboratório 2

Esta actividade está escrita na forma de tutorial e consiste na interpretação, compilação e execução de pequenos programas que efectuam chamadas a funções do sistema operativo UNIX/Linux para criação de processos. A informação referida neste enunciado foi necessáriamente simplificada de modo a não sobrecarregar as ideias e conceitos principais que se pretendem transmitir.

Para realizar esta actividade é necessário ter instalada uma distribuição GNU/Linux no seu computador e de preferência ter lido as secções 1.5, 1.6 e 2.1 do livro [MOS3e] ou os capítulos (secções recomendadas) 3 e 4 do livro [SO2e], nomeadamente os conteúdos relacionados com o conceito de processo. Como leitura complementar aos conteúdos de programação aqui introduzidos, poderá ler o capítulo 3 do livro [ALP], com possível excepção da secção 3.3. Os programas mencionados nesta actividade são fornecidos no ficheiro `lab2-progs.zip`.

I - Introdução

Para observar os processos que estão a correr no sistema utiliza-se o comando "`ps`". Abra uma janela com um terminal para ter acesso a uma linha de comandos. Execute o comando,

```
>> ps
```

onde ">>" representa a prompt do interpretador de comandos (shell). Em princípio deverão ser listados apenas dois processos, que são os que estão relacionados com o terminal em que se encontra: o interpretador de comandos (normalmente a `bash` shell) e o próprio comando `ps`. Para obter uma lista de todos os processos a correr no sistema, execute o comando,

```
>> ps -e
```

Cada processo é identificado no sistema operativo por um número, o PID (de **P**rocess **Id**entifier). O primeiro da lista é o processo `init`, que tem sempre o PID=1 e é vulgarmente designado por "pai de todos os processos", pois é a partir dele, ou dos seus "processos filhos", que são criados todos os processos do sistema. O PID do processo pai é identificado como PPID (de **P**arent **PID**).

Para se obter informação simplificada sobre um determinado comando (GNU), pode-se invocar o mesmo com a opção "`--help`", como por exemplo,

```
>> ps --help
```

Para informação mais detalhada, deve-se consultar a respectiva página da secção/manual online (documentação residente no disco). Estas páginas são vulgarmente designadas por "man pages". A quantidade de informação apresentada pode ser muito grande pelo que se deve ler apenas as partes que interessam em particular. Este comando (UNIX) segue o formato "`man [secção] nome`", onde `secção` entre `[]` significa que a sua especificação é opcional. Caso não seja especificada a secção, é dada informação sobre a primeira secção que contenha uma referência ao nome em questão. Regra geral, a secção 1 contém informação sobre comandos, a secção 2 sobre funções de sistema (system calls) e a secção 3 sobre funções de biblioteca (library functions).

Para obter informação sobre o comando `ps`, execute o comando,

```
>> man 1 ps
```

ou, nos casos em que não existem ambiguidades, tais como comandos e funções com o mesmo nome, pode simplificar-se para,

```
>> man ps
```

A informação é apresentada através de um utilitário de paginação. Para se avançar uma página prime-se a tecla espaço, para uma linha a tecla return, para recuar uma página a tecla b e para sair a tecla q. Para se obter informação sobre o próprio comando `man`, fazer,

```
>> man man
```

Felizmente o comando `ps` permite a partir da opção "`-o`" especificar as características que se pretendem visualizar para os processos. Para não se estar constantemente a especificar as características pretendidas cada vez que se invoca o comando, pode-se definir um "alias" no interpretador de comandos (shell) através de

```
>> alias pso='ps -o pid,ppid,user,state,time,comm'
```

Defina o alias, execute o comando,

```
>> pso
```

e verifique que permite obter o resultado desejado. Os termos "`state`", "`time`" e "`comm`" representam respectivamente o estado do processo, o tempo de execução e o nome do programa/comando associado ao processo. Relativamente aos estados de um processo, nesta actividade apenas interessam os estados definidos pelas letras:

`R` - Running or Runnable (Em execução ou pronto para ser executado. De facto este estado engloba dois dos estados definidos no livro recomendado [MOS3e])

`S` - Sleeping (Bloqueado)

`Z` - Zombie or Defuncted (Processo que terminou mas cuja informação ainda consta na Tabela de Processos do Sistema)

Nota: para ter o alias `pso` automaticamente definido sempre que faz login na sua área, insira uma linha com a definição do alias no ficheiro de configuração `.bashrc` existente na sua directoria por defeito.

II - Criação simples de um novo processo com `system()`

A função de biblioteca `system()` permite a partir de um programa executar um comando/programa como se este fosse dado directamente à shell. O seu protótipo é dado por,

```
#include <stdlib.h>

int system(char *string);
```

Esta função executa o comando/programa indicado no argumento `string` e espera que este termine antes de regressar. Retorna -1 em caso de erro ou o estado terminal (estado de retorno) do comando executado. Para mais pormenores sobre a utilização desta função de biblioteca, consulte a respectiva página do manual através do comando,
`>> man 3 system`

(i) Criar, observar e relacionar o PID e PPID de um processo

Analize o programa exemplificativo `cmd01.c`. Para simplificar não são verificados possíveis erros de retorno das funções. Compile o programa `cmd01.c` com o comando,
`>> gcc –Wall -o cmd01 cmd01.c`
onde o nome associado à opção "`-o`" indica o nome do ficheiro executável final, neste caso `cmd01` (sem qualquer extensão). Execute o programa dando o comando,
`>> ./cmd01`
onde "`./`" é necessário para indicar que se pretende executar o programa `cmd01` que se encontra na directoria corrente.

L2.1 Interprete os dados de saída do programa, nomeadamente relacione os campos PID, PPID e COMMAND.

L2.2 Faça experiências com outros comandos, por exemplo altere o programa para que seja executado o comando que se encontra comentado no programa.

III - Duplicar um processo com `fork()`

A função de sistema `fork()` permite criar um novo processo (filho) que é uma réplica do processo (pai) que invoca a função. O seu protótipo é dado por,

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Relativamente a esta função pode dizer-se que "retorna duas vezes". A primeira no processo pai, retornando o PID do processo filho criado. A segunda no processo filho, retornando zero. O percurso de execução de ambos os processos "continua" com o

retorno da função. Assim, ambos os processos, pai e filho, podem testar o valor de retorno para saber quem é quem e actuar em conformidade.
No caso de ocorrência de erro, não é criado o processo filho e é retornado -1 no processo que invocou a função. Para mais pormenores sobre a utilização desta função de sistema, consulte a respectiva página do manual através do comando,

```
>> man 2 fork
```

(i) Duplicar e obter o PID e o PPID de um processo

Analize o programa exemplificativo `fork01.c`. Este programa utiliza uma chamada à função de sistema `fork()` para criar um novo processo filho e recorre às funções de sistema `getpid()` e `getppid()`, que permitem respectivamente obter o PID e o PPID do processo que as invoca. Estas funções têm os seguintes protótipos,

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);
pid_t getppid(void);
```

Compile e execute o programa `fork01.c`.

L2.3 Interprete os dados de saída, nomeadamente relacione os identificadore PID e PPID. Note que ambos os processos pai e filho imprimem a mesma informação, mas que esta é obtida de maneira diferente.

Nota: Pode acontecer que o processo pai termine a sua execução antes do processo filho. Neste caso o sistema operativo pode atribuir como processo pai outro processo, normalmente o processo 1, pelo que nesta situação o processo filho pode reportar PPID=1.

(ii) Criar muitos processos

Analize agora o programa exemplificativo `fork02.c`. Este programa é semelhante ao anterior `fork01.c` mas contém o corpo principal do programa no interior de um ciclo `for`, sendo o número de iterações controlado pela constante `NITER`, que por defeito é 2. Os segmentos de código do processo pai e filho simplesmente imprimem o PID e o PPID do processo correspondente.

L2.4 Sabendo que em cada iteração do ciclo, a função `fork()` duplica cada processo (que está a executar essa iteração do ciclo), determine a partir de considerações teóricas o número de processos que estão a executar uma determinada iteração `i` do ciclo a seguir à função `fork()`.

Compile e execute o programa `fork02.c`.

L2.5 Observe e compare os resultados de saída com as suas previsões. Note que as mensagens vêm identificadas com o valor da iteração `i` do ciclo.

L2.6 Relacione o "parentesco" de uns processos com os outros. Tente desenhar graficamente numa folha uma árvore invertida de processos, em que no topo está o PID do processo inicial e em cada "nível" abaixo o PID dos processos em execução em cada iteração `i=1,2,...,` e assim sucessivamente, em que os ramos unem cada processo com os resultantes da bifurcação por aplicação da função `fork()`. Note que por cada execução da função `fork()` não são criados dois processos novos mas sim mantido o inicial (pai, continua a existir com o mesmo PID) e criado um novo (filho, com um novo PID).

L2.7 Altere o número de iterações do ciclo para `NITER=3`. Repita a análise e confirme novamente os resultados.

L2.8 Altere o número de iterações do ciclo para `NITER=4` e `5`. Confirme novamente os resultados, mas apenas em termos do número de processos criados.

Atenção: não aumente o valor de `NITER` mais do que algumas unidades, sob risco de paralizar o seu sistema.

Nota: podem eventualmente surgir algumas "anomalias" na saída de dados deste programa. Uma é que poderá surgir PPID=1 para algum processo. Tal facto será explicado melhor mais à frente nesta atividade. Outra anomalia é a prompt da linha de comandos aparecer "misturada" com os dados de saída dos programas/comandos. Isto é normal. Trata-se de um problema de sincronização entre processos em que vários processos estão a escrever simultaneamente na saída padrão. Para a shell é altura de imprimir uma nova prompt quando o processo que executou terminou. No entanto, se foram criados processos filho, estes podem continuar a gerar dados de saída que vão aparecer misturados com a prompt. Se após dar o primeiro comando não vir a prompt mas a shell estiver activa, simplesmente prima return para forçar a shell a reimprimir a prompt. De seguida dê um novo comando, se desejado.

(iii) Processo pai termina antes do processo filho

Analize agora o programa exemplificativo `fork03.c`. Este programa contém na parte do código do processo filho uma chamada à função de sistema `sleep()`. O seu protótipo é dado por,

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Esta função suspende o processo que a invoca pelo número de segundos indicado na variável `seconds`. Esta função ao ser introduzida na parte do código correspondente ao processo filho, vai provocar um atraso na sua execução, fazendo com que o processo pai termine antes do processo filho, deixando este último "orfão". Quando esta situação acontece o processo filho ganha sempre e automaticamente um novo processo pai, normalmente o processo `init` que tem PID=1.

Compile o programa `fork03.c`. Pretende-se observar o processo filho após o processo pai terminar. Para tal é necessário dar dois comandos, um a seguir ao outro, com menos

de 3 segundos de intervalo. O primeiro para executar o programa `fork03` e o segundo com o comando `pso` (definido como um alias anteriormente) para observar o processo filho ainda bloqueado (adormecido). Dois ou mais comandos podem ser dados de uma vez à shell, sendo executados sequencialmente um a seguir ao término do outro, se separados por ";". Execute o comando,

```
>> fork03; pso
```

e aguarde uns segundos que o processo filho termine.

L2.9 Observe e interprete os dados de saída gerados, nomeadamente os valores de PPID do processo filho.

(iv) Processo filho termina antes do processo pai

Analize o programa exemplificativo `fork04.c`. Este programa contém uma chamada à função de sistema `sleep()` na parte do código do processo pai. Agora é o processo filho que vai terminar antes do processo pai, ficando no estado designado de Zombie, até que o pai termine também, ou "de algum modo", recolha o código de retorno do filho. O código de retorno de um processo é o valor associado a `return` no final da função `main()` ou indicado na função `exit()` em qualquer ponto do programa.

Compile o programa `fork04.c`. Pretende-se observar o processo pai após o processo filho terminar. Relembra-se que a shell só apresenta de novo a prompt ou aceita um novo comando para executar quando o comando anterior já terminou. Assim, se for dada a sequência de comandos,

```
>> fork04; pso
```

o comando `pso` só é executado após o processo `fork04` (pai) terminar, não chegando a ser observado. Este problema é resolvido com o conceito de execução em plano de fundo (background), em que um programa é executado mas deixando o terminal livre para que possa aceitar outros comandos. Assim, execute os seguintes comandos em dois passos, um imediatamente a seguir ao outro,

```
>> fork04 &
>> pso
```

onde o caracter "`&`" no final da linha de comando indica à shell que o referido programa é para ser executado em plano de fundo. Se após dar o primeiro comando não vir a prompt, simplesmente prima return para forçar a shell a reimprimir a prompt e de seguida dê o outro comando.

Aguarde uns segundos que o processo pai termine.

L2.10 Observe e interprete os dados de saída gerados, nomeadamente o estado do processo pai e filho.

(v) Processo pai espera que o processo filho termine

Analize o programa exemplificativo `fork05.c`. Este programa contém na parte do código do processo filho uma chamada à função `sleep()` e na parte do código do processo pai uma chamada à função de sistema `wait()`. O seu protótipo é dado por,

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

Esta função bloqueia o processo que a invoca até que um qualquer dos seus processos filhos termine, retornando o PID desse processo filho. Note que só se consideram processos filho os processos criados directamente, não os processos criados pelos filhos. Esta função coloca na posição de memória apontada por `status` o estado de retorno do processo filho. Deste modo, o processo pai nunca termina antes do processo filho, garantindo que o processo filho nunca passa ao estado de Zombie. Note-se que estando o processo pai bloqueado, não está a desperdiçar tempo de CPU.

Nota: também existe a função `waitpid()` que permite esperar por um processo filho específico através da especificação do seu pid num dos argumentos.

Nota: o estado de retorno do processo é diferente do código de retorno do processo proveniente da execução de `return` ou `exit()`. O estado contém diversa informação codificada, entre a qual se encontra o código de retorno. Para aceder aos vários "ítens" de informação, existem macros pré-definidas que extraiem essa informação do estado retornado. Consulte a página do manual de `wait()` para mais informação.

Compile e execute o programa `fork05.c` tal e qual como fez com `fork04` em dois passos.

L2.11 Observe e interprete os dados de saída gerados, nomeadamente o estado dos processos e códigos PID e PPID.

IV - Criar um novo processo com `fork()` + `exec()`

A função `fork()` permite ao processo que a invoca criar um novo processo que é uma réplica de si próprio. No entanto podemos estar interessados em criar um novo processo que seja diferente, que execute código não presente no processo pai. A criação de um processo diferente é obtida em dois passos:

1) Criar um novo processo igual invocando a função `fork()`;
2) Sustituir a imagem do novo processo invocando uma das funções da família `exec()`.

Por novo processo entende-se o processo filho e por imagem do processo entende-se o seu espaço de endereçamento, que compreende o código propriamente dito do programa (também designado por texto), a zona de memória correspondente aos dados e a zona de memória correspondente à pilha. Quando a imagem do processo é substituída, algumas características do processo mantêm-se como é o caso do PID e dos apontadores `FILE* stdin, stdout e stderr`.

Os protótipos das funções da família `exec()` são dados por,

```
#include <unistd.h>

extern char **environ;

int execl(char *path, char *arg0, char *arg1, ...);
int execlp(char *file, char *arg0, char *arg1, ...);
int execle(char *path, char *arg0, ..., char *envp[]);

int execv(char *path, char *argv[]);
int execvp(char *file, char *argv[]);
int execve(char *path, char *argv[], char *envp[]);
```

Os diferentes argumentos destas funções são apenas modos diferentes de indicar o ficheiro executável do novo programa e de representar os argumentos passados à sua função `main()`. Relembrando, na sua forma mais completa, a função `main()` de um programa tem o seguinte protótipo,

```
int main(int argc, char *argv[], char *envp[]);
```

Ambos os vectores de apontadores `argv` e `envp` usados nas funções `exec()` devem ser terminados pelo apontador `NULL`. O vector `envp` contém definições de variáveis de ambiente, como por exemplo "`PATH=/bin:/usr/bin`", que neste caso particular define uma lista de diretorias onde procurar comandos/programas. O vector `argv`, de dimensão `argc`, contém no seu primeiro elemento o nome do programa e nos seguintes os seus argumentos. Nas funções `execl?` estes argumentos são dados individualmente, com `NULL` no último, em vez de através de um vector. Nas funções com argumento `path` espera-se a pathname absoluta (ou seja, iniciada por "`/`") ou a pathname relativa do ficheiro executável. Se este estiver na directoria corrente pode simplesmente ser dado só o seu nome. Nas funções com argumento `file`, se file não contém o carater "`/`" (ou seja, não é uma pathname, absoluta ou relativa), o ficheiro executável é procurado nas diretorias indicadas por `PATH`. Nas funções que não têm `envp` como argumento, as variáveis de ambiente são as definidas na variável global `environ`.

Nenhuma destas funções retorna, a não ser que ocorra um erro. Em termos teóricos, a transferencia de controlo do CPU para o código do novo programa faz-se como como se fosse um salto (jump) e não uma chamada a uma função com retorno posterior. Só no caso de ocorrência de erro as funções da família `exec()` retornam -1 e é atribuido à variável global `errno` o código do respectivo erro.

Analize o programa exemplificativo `exec01.c`. Este programa substitui a imagem do processo filho pela imagem do comando "`ls`" que se encontra na directoria "`/bin`", utilizando a função `execl()`. Repare como é dado à função a pathname absoluta do comando, o nome do comando, os argumentos do comando ("`-l`" para listagem detalhada e "`-a`" para listar todos os ficheiros, que em conjunto podem ser abreviados para o argumento único "`-al`"), e o apontador `NULL` a indicar o fim da lista de argumentos.

L2.12 Compile e execute o programa `exec01.c`. Relacione o código do programa com os dados de saída do mesmo. Note que nem a mensagem de erro nem a de comando `ls` executado são imprimidas pelo processo filho.
Nota: se executar o programa várias vezes, é possível que a sequência de dados de saída do programa varie de execução para execução. Isto é normal quando vários processos estão a ser executados em paralelo (ou em pseudo-paralelismo).

L2.13 Altere o programa anterior para `exec02.c` de modo a executar o comando "`ps -o pid,ppid,user,state,time,comm`" utilizando outras funções da família `exec()`, por exemplo `execvp()`.

FIM