Programação Sistema com Processos

Processos

- Actualmente os sistemas operativos permitem a execução de vários programas em simultâneo. Por exemplo o utilizador pode estar a trabalhar no Ms Word enquanto o Outlook Express descarrega as mensagens de e-mail. Ambos os programas se encontram a correr.
- Um programa a correr é normalmente designado de processo.
- Um processo está sempre associado a um espaço de memória. Neste espaço encontra-se o programa e os dados necessários ao seu funcionamento.
- Em computadores com apenas 1 processador o sistema operativo vai disponibilizando tempo de processador a cada um dos processos que se encontram a correr (time slice ou quantum).
- Em computadores com mais de 1 processador o sistema operativo pode distribuir os vários processos pelos vários processadores, acelerando assim a execução dos vários programas.
- Quando o processo que está a ocupar o processador inicia uma operação de entrada/saída (por exemplo a leitura de um ficheiro) então o sistema operativo fornece o processador a outro processo pois o primeiro não pode avançar no programa enquanto a operação de entrada/saída não terminar.
- Os processos podem estar essencialmente em três estados distintos: Correr (Running), Preparado (Ready), Bloqueado (Blocked).
- Num sistema com 1 processador apenas existe 1 processo no estado Correr. O sistema operativo só escolhe para Correr processos que estejam no estado Preparado. Os processos que estejam à espera da conclusão de operações de entrada/saída encontram-se no estado Bloqueado.
- No UNIX, um processo pode criar outro processo para que este desempenhe outras funções que lhe sejam atribuidas.
- A criação de um novo processo é realizada através da chamada ao sistema fork.
- O novo processo (processo filho) é uma cópia do processo que o criou. Fica com o mesmo programa do processo que o criou.
- Como normalmente se pretende que o processo filho realize algo diferente do processo pai, o programa deve conter instruções específicas para o processo filho, e instruções específicas para o processo pai.
- Esta diferenciação, entre o processo pai e o processo filho, é normalmente realizada logo após a função fork através de um if.
- Se for o pai que está a correr o programa executa um dos ramos do if, se for o filho executa o outro ramo do if.

Criação de processos

- Neste exemplo é criado um processo filho. Ambos os processos correm o mesmo programa, ou seja escrevem ambos os número de 1 até 10.

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

int main(){
  int i;

  fork();

  for( i = 0; i <= 10; i++ )
    printf("%d\n", i );

  return 0;
}
```

- Programa semelhante ao anterior mas agora utiliza-se a função getpid para se saber em cada momento qual o processo que escreve no ecrã.

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

int main(){
  int i;

  printf("Sou o pai. Tenho o pid  %d.\n", getpid() );

  fork();

  for( i = 0; i <= 10; i++ )
    printf("Sou o %d: %d\n", getpid(), i );

  return 0;
```

```
}
```

- Este programa consegue distinguir quem o está a correr, o processo pai, ou o processo filho. Para isso o programa observa o valor devolvido pelo fork. Quando o fork devolve zero isso quer dizer que é o filho que está a correr o programa. Quando o fork devolve um valor diferente de zero (na verdade o pid do filho) isso quer dizer que é o processo pai que está a correr o programa. Desta forma com um if a seguir ao fork é possível o processo pai realizar uma operação (escrever os números de 0 a 10), e o processo filho realizar outra operação (escrever os números de 10 a 0).

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

int main(){
  int i, pid;

  printf("Sou o pai. Tenho o pid  %d.\n", getpid() );

  pid=fork();
  if( pid == 0 ){

    for( i = 10; i >= 0; i-- )
      printf("Sou o %d: %d\n", getpid(), i );

  }
  else{

    for( i = 0; i <= 10; i++ )
      printf("Sou o %d: %d\n", getpid(), i );

  }

  return 0;
}
```

- Este programa é semelhante ao anterior, mas neste caso o processo pai vai esperar que o processo filho escreva os seus números, e termine, para depois escrever os seus números. O processo pai espera que o filho termine através da função wait. O parâmetro desta função disponibiliza informação sobre a execução do processo filho.

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

int main(){
  int i, pid, childStatus;

  printf("Sou o pai. Tenho o pid  %d.\n", getpid() );

  pid=fork();
  if( pid == 0 ){

    for( i = 10; i >= 0; i-- )
      printf("Sou o %d: %d\n", getpid(), i );

  }
  else{

    wait(&childStatus);

    for( i = 0; i <= 10; i++ )
      printf("Sou o %d: %d\n", getpid(), i );
  }

  return 0;
}
```

- No programa seguinte são lançados tantos processos filhos quanto o número de argumentos passados ao programa. Com base nesta estrutura podem ser desenvolvidos programas que colocam um processo filho diferente a tratar de cada argumento recebido.

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

```
#include <stdlib.h>

int main(int argc, char *argv[]){
  int i, pid, childPid, childStatus, nChilds;

  printf("Sou o pai. Vou criar tantos filhos quanto os argumentos passados.\n");

  nChilds = argc -1;

  for( i = 0; i < nChilds; i++ ){

    pid=fork();
    if( pid == 0 ){

      printf("Sou o filho %d e tenho o pid %d. Vou tratar do argumento %s\n", i, getpid(), argv[ i + 1 ] );

      exit(0); //o filho termina aqui, se continuasse iria seguir para o ciclo...
    }
    else{
      //o pai não faz nada, simplesmente continua no ciclo para criar outro processo
    }

  }

  //Depois de criar os vários filhos o pai espera por cada um deles
  for( i = 0; i < nChilds; i++ ){
    childPid = wait(&childStatus);
    printf("Terminou o filho com o pid %d\n", childPid);
  }

  return 0;
}
```

- Como exemplo de aplicação do programa anterior, mostra-se uma primeira versão de um programa que lança vários filhos para copiar cada um dos fcheiros fornecidos na linha de comando para uma diretoria também fornecida na linha de comando. A diretoria destino é o último argumento da linha de comando. Repare que este programa ainda não realiza a cópia dos ficheiros, apenas mostra no ecrã o caminho para o ficheiro original e o caminho para nova cópia deste ficheiro.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>
#include <libgen.h>

#define MAX_FILENAME 500

int main(int argc, char *argv[]){
  int i, pid, childPid, childStatus, nChilds;

  printf("Sou o pai. Vou criar um filho diferente para copiar cada ficheiro fornecido na linha de comando. O último a

  nChilds = argc - 2;

  for( i = 0; i < nChilds; i++ ){

    pid=fork();
    if( pid == 0 ){

      printf("Sou o filho %d e tenho o pid %d. Vou tratar do argumento %s\n", i, getpid(), argv[ i + 1 ] );

      char sourceFileName[MAX_FILENAME];
      char baseFileName[MAX_FILENAME];
      char newFileName[MAX_FILENAME];

      //O filho 0 copia o ficheiro no argv[ 1 ], o filho 1 copia o ficheiro no argv[ 2 ], ...
      strcpy( sourceFileName, argv[ i + 1 ] );

      //Constroi o nome completo do novo ficheiro: diretoria de destino + "/" + nome simples do ficheiro
      strcpy( newFileName, argv[ argc - 1 ] );
      strcat( newFileName, "/" );
      strcat( newFileName, basename(sourceFileName) );

      printf("A copiar: %s >>> %s\n", sourceFileName, newFileName );

      exit(0); //o filho termina
```

```
    }
    else{
      //o pai continua no ciclo para criar outro processo
    }

  }

  //Depois de criar os vários filhos o pai espera por cada um deles
  for( i = 0; i < nChilds; i++ ){
    childPid = wait(&childStatus);
    printf("Terminou o filho com o pid %d\n", childPid);
  }

  return 0;
}
```

Execução de um programa

- A familia de funções exec permite a um processo executar um programa como se este fosse chamado pela linha de comando. Uma destas funções é execlp, à qual deve ser fornecido o nome do programa e a lista de argumentos. Esta lista deve terminar com (char *) NULL. Quando um processo chama uma das funções exec ele troca de programa (não é uma chamada de função!) e nunca mais retorna ao programa original. No exemplo seguinte é executado o comando ls -l. Como se pode verificar a instrução printf nunca será executada.

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

int main(){

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

  printf("Esta instrução nunca será executada!\n");

  return 0;
}
```

- Se o programa for executado por um processo filho, o processo pai pode continuar a realizar outras operações.

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

int main(){
  int i,pid, childPid;

  pid=fork();
  if( pid == 0 ){

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

  }
  else{

    wait(&childPid);

    printf("Programa executado! Posso continuar...\n");
  }

  return 0;
}
```

- A função execvp permite que a lista de argumentos seja fornecida através de um vector. Desta forma a quantidade de argumentos pode ser variável.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
```

```
#include <string.h>

#define MAX_ARGS 100

int main(){
  int i,pid, childPid;
  char *argumentsVector[MAX_ARGS];

  //Guarda o primeiro argumento no vetor
  argumentsVector[0] = (char *) malloc(3);
  strcpy(argumentsVector[0], "ls");

  //Guarda o segundo argumento no vetor
  argumentsVector[1] = (char *) malloc(3);
  strcpy(argumentsVector[1], "-l");

  //Termina a lista de argumentos
  argumentsVector[2] = (char *) NULL;

  pid=fork();
  if( pid == 0 ){

    execvp( argumentsVector[0], argumentsVector);

  }
  else{

    wait(&childPid);

    printf("Programa executado! Posso continuar...\n");

  }

  return 0;
}
```

Redireção e Pipes

- O programa seguinte redireciona o standard output para um ficheiro e executa o comando ls -l. A listagem produzida por este comando será armazenada no ficheiro.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(){
  int fd;

  fd = open("output_ls.txt", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

  dup2(fd, STDOUT_FILENO );

  close( fd );

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

  printf("Esta instrução nunca será executada!\n");

  return 0;
}
```

- O programa seguinte realiza a mesma operação que o anterior mas através de um processo filho.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(){
  int i, pid, childPid, fd;
```

```
  pid=fork();
  if( pid == 0 ){

    fd = open("output_ls.txt",  O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

    dup2(fd, STDOUT_FILENO );

    close( fd );

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

    printf("Esta instrução nunca será executada!\n");

  }
  else{

    wait(&childPid);

    printf("Programa executado! Posso continuar...\n");
  }

  return 0;
}
```

- O programa seguinte executa o comando ls -l | more. Para isso é criado um pipe e redirecionado o standard output do processo filho, que executa o programa ls -l, e o standard input do processo pai, que executa o programa more.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(){
  int i, pid, childPid, fd[2];

  pipe(fd);

  pid=fork();
  if( pid == 0 ){

    dup2(fd[1], STDOUT_FILENO);

    close( fd[0] );
    close( fd[1] );

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

  }
  else{

    wait(&childPid);

    dup2(fd[0], STDIN_FILENO);

    close( fd[0] );
    close( fd[1] );

    execlp( "more", "more", (char *) NULL);

  }

  return 0;
}
```

- No programa seguinte o processo pai não faz nada. Os comandos são executados por processos filhos.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int main(){
  int i, pid, childPid, fd[2];

  pipe(fd);

  pid=fork();
  if( pid == 0 ){

    dup2(fd[1], STDOUT_FILENO);

    close( fd[0] );
    close( fd[1] );

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

  }
  else{

    pid=fork();
    if( pid == 0 ){

      dup2(fd[0], STDIN_FILENO);

      close( fd[0] );
      close( fd[1] );

      execlp( "more", "more", (char *) NULL);

    }
    else{

      close( fd[0] );
      close( fd[1] );

      wait(&childPid);
      wait(&childPid);
    }

  }

  return 0;
}
```

Última revisão efectuada em 23/01/19