Processos em Linguagem C

Projetos de Sistemas Operacionais

Prof. Dr. Denis M. L. Martins

Engenharia de Computação: 5º Semestre

Processos em C

```c
//getpid.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    pid_t myPid = getpid();
    printf("My process ID is %d\n", myPid);
    return 0;
}
```

```
$ ./getpid
My process ID is 18814
$ ./getpid
My process ID is 18829
```

Processos em C

```c
//getpid.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    pid_t myPid = getpid();
    printf("My process ID is %d\n", myPid);
    return 0;
}
```

```
$ ./getpid
My process ID is 18814
$ ./getpid
My process ID is 18829
```

Função `fork()`

Criando um processo filho usando `fork()`. O novo processo é uma cópia do processo pai.

```c
//myprogram.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

Função `fork()`

Criando um processo filho usando `fork()`. O novo processo é uma cópia do processo pai.

Processo A

```c
//parentchild.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

Processo B

```c
//parentchild.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

Pergunta: Qual seria a saída do comando abaixo?

```
$ ./parentchild
Hello, world!
Goodbye!
Goodbye!
```

Função `fork()`

Criando um processo filho usando `fork()`. O novo processo é uma cópia do processo pai.

Processo A

```c
//parentchild.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

Processo B

```c
//parentchild.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

Pergunta: Qual seria a saída do comando abaixo?

```
$ ./parentchild
Hello, world!
Goodbye!
Goodbye!
```

Função `fork()`

```c
//pidouzero.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pid = fork();
    printf("fork retornou %d\n", pid);
    return 0;
}
```

```
$ ./pidouzero
Hello, world!
fork retornou 1111
fork retornou 0
```

- Processo-pai (original) cria um novo processo-filho.
- O processo-filho executa a próxima instrução do programa.
- O processo-pai continua executando a próxima instrução do seu programa.
- `fork()` é chamada uma vez, mas retorna duas vezes:
  - No processo-pai: retorna o **pid** do processo-filho.
  - No processo-filho: retorna **zero**.

Função `fork()`

```c
//pidouzero.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pid = fork();
    printf("fork retornou %d\n", pid);
    return 0;
}
```

```
$ ./pidouzero
Hello, world!
fork retornou 1111
fork retornou 0
```

- Processo-pai (original) cria um novo processo-filho.
- O processo-filho executa a próxima instrução do programa.
- O processo-pai continua executando a próxima instrução do seu programa.
- `fork()` é chamada uma vez, mas retorna duas vezes:
  - No processo-pai: retorna o **pid** do processo-filho.
  - No processo-filho: retorna **zero**.

Função `fork()`

Não podemos assumir a order de execução dos processos. O SO decide a ordem de execução com base em seu algoritmo de escalonamento (aula futura).

```c
//pidouzero.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pid = fork();
    printf("fork retornou %d\n", pid);
    return 0;
}
```

Primeira execução

```
$ ./pidouzero
Hello, world!
fork retornou 1111
fork retornou 0
```

Segunda execução

```
$ ./pidouzero
Hello, world!
fork retornou 0
fork retornou 1111
```

Função `fork()`

Não podemos assumir a order de execução dos processos. O SO decide a ordem de execução com base em seu algoritmo de escalonamento (aula futura).

```c
//pidouzero.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pid = fork();
    printf("fork retornou %d\n", pid);
    return 0;
}
```

Primeira execução

```
$ ./pidouzero
Hello, world!
fork retornou 1111
fork retornou 0
```

Segunda execução

```
$ ./pidouzero
Hello, world!
fork retornou 0
fork retornou 1111
```

Função `fork()`

Não podemos assumir a order de execução dos processos. O SO decide a ordem de execução com base em seu algoritmo de escalonamento (aula futura).

```c
//pidouzero.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pid = fork();
    printf("fork retornou %d\n", pid);
    return 0;
}
```

Primeira execução

```
$ ./pidouzero
Hello, world!
fork retornou 1111
fork retornou 0
```

Segunda execução

```
$ ./pidouzero
Hello, world!
fork retornou 0
fork retornou 1111
```

Quantos processos são criados, incluindo o processo-pai?

```c
//myprogram.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    fork();
    fork();
    fork();
    return 0;
}
```

Encerramento de um Processo

Um processo é encerrado quando termina a execução de seu último comando e solicita ao sistema operacional que o exclua, usando a chamada de sistema `exit()`.

- O processo pode retornar um valor de status (normalmente, um inteiro) para seu processo-pai. Exemplo: `exit(1)`;
- Todos os recursos do processo são desalocados pelo sistema operacional: memória, arquivos, etc.
- No encerramento normal, `exit()` pode ser chamada diretamente (como mostrado acima) ou indiretamente por um comando `return` em `main()`.

Encerramento de um Processo (cont.)

Um processo-pai pode esperar o encerramento de um processo-filho usando a chamada de sistema `wait()`.

- `wait()` recebe um parâmetro que permite que o pai obtenha o status de saída do filho.
- Exemplo: `int status; pid = wait(&status);`

Quando um processo termina, seus recursos são desalocados pelo sistema operacional. No entanto, **sua entrada na tabela de processos** deve permanecer **até que o pai chame** `wait()`, porque a tabela de processos contém o status de saída do processo.

- Um processo que foi encerrado, mas cujo pai ainda não chamou `wait()`, é conhecido como processo **zumbi**.
- Quando o pai chama `wait()`, o identificador do processo zumbi e sua entrada na tabela de processos são liberados.

Funções `wait()` e `exit()`

```c
//myprogram.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    pid_t pid = fork(); //Cria processo-filho
    if (pid < 0){
        printf("Erro ao criar processo.\n")
    }
    else if (pid == 0){
        printf("Eu sou o filho.\n");
        exit(1); //Encerra o processo
    }
    else {
        printf("Eu sou o pai.\n");
        wait(NULL); //Espera o processo-filho encerrar
    }
    return 0;
}
```

Processo **zumbi**

```c
//myprogram.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    pid_t pid = fork(); //Cria processo-filho
    if (pid < 0){
        printf("Erro ao criar processo.\n")
    }
    else if (pid == 0){
        printf("Eu sou o filho.\n");
        exit(1); //Encerra o processo
    }
    else {
        printf("Eu sou o pai.\n");
        sleep(120); //Enquanto dorme 2min, o processo-filho é zumbi
        wait(NULL); //Espera o processo-filho encerrar
    }
    return 0;
}
```

Dúvidas e Discussão

Prof. Dr. Denis M. L. Martins

denis.mayr@puc-campinas.edu.br