Aluno: Matheus Laidler Universidade: UFRJ

Professor: Daniel Bastos Programação: Tarefa 10

System Calls X Library Functions

Nas tarefas deste módulo fizemos testes em laboratório para vermos o funcionamento de funções que fazem chamadas ao sistema e outras que apenas dependem unicamente da própria linguagem e suas respectivas bibliotecas, sendo melhor portável entre sistemas.

A execução e suas diferenças - (Intro + Códigos)

O teste pedido foi para que verificássemos a diferença da função de abrir um arquivo "pela linguagem" e "pelo sistema" enquanto adicionamos alguma coisa dentro do mesmo, para então sim vir a função de fechar o arquivo e finalizar o programa. Porém, antes disso foi adicionado uma função de "sleep" com uns 60 segundos para podermos verificar os arquivos enquanto a execução está em andamento. Portanto o sleep fica antes do close/fclose.
Veremos o código final dos programas e a análise de suas funções para verificarmos a diferença entre ambos. O arquivo que usamos foi o "*database.txt*" da tarefa anterior.

Programas criados: "***./fopen***" e "***./open***";

.Bibliotecas que podemos utilizar:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

Código final de fopen:

```
const char *str = "A melhor introdução à programação C/C + + por Daniel Bastos! - (UFRJ)\n";

int main(void) {
   const char* arquivo = "database.txt"; FILE* cta;
   cta = fopen(arquivo, "a+");
   if (!cta) {
      perror("fopen");
      return 1;
   }
   fprintf(cta, "%s", str);
   printf("Escrita feita com sucesso!\n");
   /*fclose(cta);*/
   sleep(60);
   fclose(cta);
   return 0;
}
```

$ > gcc fopenlab.c -o fopen

$ > ./fopen

> Escrita feita com sucesso!

$>

Código final de open;

```
const char *str = "A melhor introdução à programação C/C + + por Daniel Bastos! - (UFRJ)\n";

int main(void) {
    const char* arquivo = "database.txt"; int cta;
    cta = open(arquivo, O_CREAT | O_WRONLY | O_APPEND, S_IRWXU);
    if (cta == -1) {
        perror("open");
        return 1;
    }
    write(cta, str, strlen(str));
    printf("Escrita feita com sucesso!\n");
    sleep(60);
    close(cta);
    return 0;
}
```

$ > gcc openlab.c -o open

$ > ./open

> Escrita feita com sucesso!

$>

Analisando a ação de cada arquivo e vendo suas diferenças...

Execução de fopen:

Enquanto o programa está no ***sleep***, nada ainda foi escrito realmente, apenas após a finalização.

É interessante comentar também que da forma que fizemos pela primeira vez, com '*w+*', ele apagava tudo que estava no arquivo antes e escreve o que pedimos, então alterei para 'a+' para ter mais haver com a saída que queremos, que é apenas adicionar uma linha com a *string*.

Então, **durante** a execução do *sleep*, o *"database"*, da aula passada, estava **sem alteração**. Ocorrendo apenas **após** o término do programa rodar por inteiro, isto é, se terminarmos o programa com ***'ctrl-c'*** a escrita não é finalizada também, até porque **interrompemos o 'funcionamento' pausado** do programa para finalizá-lo, pois do jeito que fizemos **o programa só conclui a ação após os 60 segundos**.Quando o programa diz que quer escrever no arquivo, isto só acaba acontecendo de fato quando se depara com o fclose, porém ele é botado em "pausa" antes da finalização acontecer, e enquanto isso ocorre ele é finalizado a força, interrompendo o processo, logo a ação de escrever em "database" não chega a acontecer, pois necessita que o código termine de rodar. Se colocarmos o "***fclose"*** antes do "***sleep"***, o resultado muda e a escrita ocorre de '*imediato*'.

-O teste foi feito com fwrite e fprintf, sem mudanças no resultado final-

*(Colocando "**a+**" no **fopen** ficamos com o arquivo intacto, porém com a adição da string ao final. Já com o "**w+**" ele apaga o conteúdo caso o arquivo exista)*

Execução de open:

Podemos visualizar que enquanto o programa está em ação, isto é, ainda no *timing* do ***sleep(60)***, a ação de escrever a string no arquivo "database" já foi efetuada com sucesso!

A chamada do sistema para abertura, escrita e fechamento do arquivo mostram que a função foi delegada ao sistema e não para a linguagem em si, ou seja, se o sistema foi colocado para fazer algo, ela não precisa "esperar" pelo código para que a ação possa ser feita, bastando ser chamada. Como se a linguagem dissesse ao sistema: "*faz isso enquanto vou fazendo aquilo*".

(Ao menos, é um modo de como podemos interpretar as coisas)

-Quando uso o *open(arquivo, O_CREAT | O_WRONLY | O_APPEND, S_IRWXU)* quero, também, que seja adicionado no final do arquivo a string e crie caso não exista o arquivo. Eu posso usar um mais simples, como apenas *O_RDWR | O_CREAT* que apenas add a string no início do arquivo (que também acabou dando uma "*bugadinha"* na *database*, pois pegava o final da primeira linha e a colocava logo a baixo, deixando a linha antiga cortada)-

Analisando Diferenças;

CONCLUSÃO DA DIFERENÇA ENTRE AS DUAS FUNCIONALIDADES

*Analisando a diferença da execução de **open** e **fopen***

A diferença do "***fopen"*** pro "***open"*** que montei foi, essencialmente, apenas o momento em que a edição do arquivo fora feita. Com isso, uma delas pode ter a **ação** interrompida no *sleep* enquanto que a outra não. Em outras palavras, enquanto o ***"open"*** fez a manipulação do arquivo em *tempo real*, sem precisar esperar pelo *close*, o "***fopen"*** precisou esperar os *60 segundos* do ***sleep*** até chegar em ***fclose*** para termos o resultado da ação do programa em si.

Se o ***sleep(60)*** viesse depois, o resultado seria o mesmo de ***"open"***;

Até porque, se mandamos botar o programa para *"dormir"* para não ser um processo em atividade pela cpu, no **meio de sua execução**, ele **irá terminar apenas quando voltar as atividades**, então se ela precisa voltar às atividades para editar o arquivo e interrompemos esto, então a ação não pode acontecer mesmo, diferentemente de quando fazemos uma **chamada pro sistema** e pedimos a ele que **escreva** e depois fazemos ele "***dormir***". Nesse caso o programa estará apenas deixando para **fechar** e **finalizar** o programa depois, mas a função da **escrita** já foi feita pelo sistema no momento da execução, assim sendo, ***sem precisar esperar o código rodar até o "close"***. O "*fopen"* e o "*fclose"* iniciam e finalizam "juntas", se o *fclose* não chegar a ser lido as ações não se salvam.

(Ao menos esse foi meu entendimento diante dos testes, posso estar completamente enganado por estar aprendendo)

FORK: Como pais e filhos;

Uma breve introdução ao fork e processos

Fazer um “fork” nada mais é do que fazer um clone de, por exemplo, um processo. Em outras palavras, é apenas uma função que duplica um processo do sistema. É chamado de **processo pai** aquele que uma vez chamou a função fork. Portanto, o novo processo criado pela função é o chamado **processo filho**. Com isso, como uma boa clonagem, as áreas do processo são duplicadas dentro do sistema operacional. O filho herda informações do pai, como variáveis de ambiente, prioridade, diretório etc

Entretanto, o processo filho possui algumas informações diferentes, como um PID único dentro do sistema, isto é, uma identificação única (Process IDentification). Um PPID (Parent Process IDentification) do processo filho é o mesmo que o PID do processo pai. Verificaremos isso nas análises abaixo.

Fica fácil de perceber tais informações ao fazer os testes por sí só.

Criei um programa para testar todas o fork e depois que consegui fazer, decidi colocar um loop para manter os processos ativos enquanto estou a verificar e analisar novas informações. Legal verificar a utilização da CPU com os processos ativos junto. O programa basicamente chama um fork e nos indica a identificação tanto do processo pai quanto do filho e mantém eles funcionando até eu finalizar o programa.

Veja abaixo o código do programa:

O código do programa:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void){
	pid_t a;
	a = fork();
	if (a == -1){
		printf("Essa não! Um erro foi encontrado!\n\n");
		printf("O programa está sendo encerrando...\n");
		sleep(10);
		printf("Tente novamente.\n");
		return 1;
	}
	if (a > 0) {
		printf("\tProcess[0] - ID do Processo: %d\n",getpid());
		while(1);
	} else {
		printf("\tProcess[1] - ID do Processo: %d\n",getpid());
		while(1);
	}
	return 0;
}
```

Análises e Testes do FORK

Após a criação e compilação do programa, ao executar o mesmo, vemos o resultado dos ID's sendo fornecido pelo programa.

$> gcc fork.c -o fork

$> ./fork

Process[0] - ID do Processo: 305

Process[1] - ID do Processo: 306

_

Enquanto o programa continua em execução, vejo que o uso do processador aumenta e podemos encontrar estes dois rodando. Vamos então dar uma olhada a mais neles com o comando *'ps';*

$ ps

```
  PID TTY          TIME CMD
  205 pts/1    00:00:00 bash
  315 pts/1    00:00:00 ps
```

$> ps -a

```
  PID TTY          TIME CMD
  305 pts/0    00:07:02 fork
  306 pts/0    00:07:02 fork
  314 pts/1    00:00:00 ps
```

.Podemos verificar, aqui, os nossos dois processos abertos.

Se colocarmos para aparecer apenas quem tiver o PPID 305, que é o ID do pai, então encontraremos o filho:

```
$ ps --ppid 305
  PID TTY          TIME CMD
  306 pts/0    00:08:49 fork
```

(podemos usar o “ps --help list” para encontrarmos uma lista resumida em lista do que podemos utilizar nessa ocasião, ou “ps --help all” para saber ainda mais)

Para ficar mais claro, usei o -F para trazer mais informações de tabela e o A para nos trazer tudo (como uso o WSL não verá muita coisa). Também pensei em trazer o H para mostrar a hierarquia, assim veremos melhor sobre o parentesco;

```
$> ps -faH
UID        PID  PPID  C    STIME TTY      TIME    CMD
vidal      312   205     0     02:24 pts/1       00:00:00  ps -faH
vidal     *305*   11     99    02:19 pts/0       00:04:37  ./fork
vidal      306  *305*  99    02:19 pts/0       00:04:37     ./fork
```

Agora fica fácil de percebermos a relação do *PID* e *PPID* do processo pai e filho (Inclusive, marquei na tabela para facilitar).

Agora , para finalizar, veremos uma aplicação de fork e tentaremos fazer o processo pai não morrer antes do filho, e ainda guardar informação.

O programa da atividade

O grande problema que encontrei para fazer esta tarefa foi que o programa, de fato, não funcionava comigo. Sempre ocorria um erro diferente ao compilar, até consegui arrumar alguns erros, mas não deu para editar e testar o programa, pois realmente não compilava. Mesmo eu colocando o '}' quando precisasse, retirando os comentários que eram apenas '//' e compilando com a indentação correta, o erro permaneceu. Não sei se eu estou perdendo algo, mas comigo deu 'zebra' ao tentar rodá-lo. Porém, como não fazer não é uma opção, eu decidi continuar mesmo assim e estudar a função **wait** para fazer meu parágrafo. Como vou assumir que o problema está comigo, vou então darei aqui a minha suposição do que podemos fazer para o código rodar e não matar o pai antes: Podemos utilizar ***waitpid*** ou algo da família **wait** para podermos fazer o processo pai terminar apenas após o término do processo filho.
por exemplo: waitpid(r,NULL,0) -> o r é a variável do pid no código.Assim poderemos fazer um mecanismo de sincronização, fazendo o programa entrar em modo de espera até que o procedimento do pid passado no parâmetro se encerre.
Vou falar um pouco sobre o que acabei estudando sobre o waitpid e depois mostrarei o que eu acho que podemos adicionar no programa para fazer a funcionalidade requerida. Espero que eu esteja indo no caminho certo da ideia por trás da atividade, pois o aprendizado foi super interessante. A minha ideia ainda precisa ser adaptada, mas já deve dar para fazer a funcionalidade com poucos ajustes. Realmente é difícil fazer sem poder pôr em prática no código, ainda mais sendo algo que recém 'aprendi'.

Sobre **waitpid**

O waitpid requer três argumentos, o primeiro é o número de identificação do processo que estamos chamando de ‘pid’. Vamos falar dos valores dele como sendo apenas -1 e > 0. Sendo o -1 podendo ser usado para monitorar qualquer filho que mude seu estado, que é usado para implementar a funcionalidade wait . Nesse caso vamos considerar este como a funcionalidade que não queremos utilizar, portanto será a aba da tela de ‘erro’ do código. Já o ‘> 0’ será sobre o ID do processo real que foi retornado de fork, que é usado para monitorar apenas um filho específico (por isso usamos o ‘== 0’ para especificar a criança que acabamos de criar no fork. Enquanto o P do pai fica sendo seu PID (> 0), a da criança criada fica sendo 0),

O segundo argumento é um ponteiro inteiro, o programa vai armazenar as informações de status do filho nela, assim podemos usar as macros. O último argumento é do tipo int e é usado para especificar determinados eventos de processo filho a serem monitorados.

Aqui vai uma sugestão do que pode ser adicionado ao código. Infelizmente não consegui testar para saber se estou no caminho certo, então… poderíamos adaptar e seguir a lógica disto:

```
int status;
  if (waitpid(r, &status, WUNTRACED | WCONTINUED) == -1) {
        perror("wpid");
        exit(EXIT_FAILURE);
    }
  if (WIFEXITED(status)) {
        printf("exit status; %d\n", WEXITSTATUS(status));
    }
```