Relatório Trabalho 1

SSC-0640 Sistemas Operacionais I

Aluno: João Pedro Doimo Torrezan 9806933

Parte 1

Para a parte 1 foram feitos dois programas, um deles é que possui interação com o usuário, contendo um Menu e opções para serem escolhidas, como Gerenciamento de Memória, Gerenciamento de Arquivos e Gerenciamento de Processos. Essa interação é feita por interface textual. O outro programa não possui interação com o usuário e ao ser compilado e executado já roda todos os comandos de Gerenciamento.

O programa é dividido em três partes, uma que contém as syscalls referentes ao gerenciamento de Arquivos, outra as syscalls referentes ao gerenciamento de Processos e a última referente ao gerenciamento de Memória.

Gerenciamento de Arquivos

A parte do programa referente ao gerenciamento de arquivos tem como função, abrir um arquivo que continha a sentença “Hello World!”, copiar o conteúdo para um buffer e em seguida fechar o arquivo.

As syscalls de Gerenciamento de Arquivos foram:

- Open: [`int open(const char *pathname, int flags, mode_t mode)`]: essa syscall abre (ou cria, caso haja a flag O_CREAT) algum arquivo no caminho especificado por *pathname*, caso haja a flag de criação o argumento *mode* deve ser especificado, caso contrário, será ignorado. Porém em alguns sistemas operacionais, ao chamar essa syscall em código, na compilação ela pode ser trocada pela *openat* a qual é usada quando se é passado um *pathname* relativo.
- Read: [`ssize_t read(int fd, void *buf, size_t count)]:` A syscall read lê a quantidade de bytes indicada no argumento *count*, salvando no buffer *buf* o conteúdo lido do arquivo indicado por *fd.*
- Close [`int close(int fd)`]: Essa syscall é bem simples, ela é responsável pelo fechamento do arquivo aberto apontado por *fd.*

Gerenciamento de Processos

O funcionamento da porção de código referente as syscalls de gerenciamento de processos se dá da seguinte maneira: a função obtém o pid (ID do processo), em seguida o processo gera um filho e aguarda o encerramento de sua execução.
As syscalls utilizadas foram:

- getpid: [`pid_t getpid(void)`]: Função responsável por retornar o ID do processo que a chamou.
- Fork: [`pid_t fork(void)`]: Essa system call cria um processo filho idêntico ao pai, retornando para o pai o pid do filho e para o filho o número 0, com isso é possível definir, em código, o que cada uma das partes do programa fará.
- Waitpid: [`pid_t waitpid(pid_t pid, int *wstatus, int options)`]: System call que faz com que o processo que a chamar, geralmente o pai, aguarde um sinal vindo de um, ou de todos, os seus processos filhos.

  Ao executar esse código notamos que o *fork*, ao ser executado pelo sistema operacional, é trocado pela system call *clone*, a qual possui um funcionamento bem semelhante.

Gerenciamento de Memória

O funcionamento da função de gerenciamento de memória foi feito usando duas system calls, porém uma delas foi usada de dois modos de diferentes, para assim totalizar 3 usos. Entretanto, ao se chamar as system calls em C, somente uma delas é executada nos 3 casos.

As syscalls são:

- Sbrk: [`void *sbrk(intptr_t increment)`]: Incrementa o espaço de armazenamento do programa no valor de *increment*.
  - Caso o parâmetro *increment* for 0 (zero) o valor retornado pela função retorna o valor final do espaço de armazenamento;
  - Caso o parâmetro *increment* for diferente de zero esse valor é acrescentado no final do espaço de armazenamento.

- Brk: [`int brk(void *addr)`]: Essa syscall seta o valor do process break, que é referente ao final do espaço de armazenamento, para posição apontada por **addr*.

Entretanto ao se rodar esse programa foi constatado que somente a syscall brk é chamada nos três casos.

Resultados Obtidos

Ao se executar o código sem interação com usuário no Cluster a resposta seguinte foi obtida:

% time	seconds	usecs/call	calls	errors	syscall
58.77	0.000191	191	1		wait4
18.77	0.000061	61	1		clone
14.46	0.000047	16	3		open
4.31	0.000014	2	6		brk
1.85	0.000006	3	2		read
1.54	0.000005	2	3		close
0.31	0.000001	1	1		getpid
100.00	0.000325		17		total

Como pode ser visto, as syscalls waitpid, fork e sbrk não foram utilizadas, entretanto foram substituídas por outras que executam as mesmas funções. Waitpid foi substituída por wait4, a qual consumiu mais tempo na execução, já que esta aguarda por sinais vindo de outros processos. Além disso outras syscalls foram chamadas mais vezes do que o previsto, isso por que para a correta execução do programa o processo depende das interações com o kernel, como por exemplo para realizar um printf.

Parte 2

CPU e IO Bound

Para a segunda parte foram criados dois códigos, um CPU bound e outro IO bound. O código CPU bound realiza a contagem de quantas vezes um determinado dígito (determinado pelo usuário) aparece na contagem de 0 a 1 Bilhão. Já o código IO bound é um gerador de números aleatórios, que os salva em um arquivo. No segundo código são gerados 10 Milhões de números.

Para a execução de ambos pasta serem compilados e executados. O CPU bound depende de uma entrada do usuário com o dígito desejado.

Os resultados gerados para os códigos serão representados abaixo.

	IO Bound	**CPU Bound**
Tempo decorrido em segundos	14.98	22.48
Tempo decorrido em modo Kernel	0.66	0.01
Tempo decorrido em modo usuario	14.13	22.14
Número de mudanças de contexto involuntárias	1858	1985
Número de mudanças de contexto voluntárias	22	2
Número de arquivos de entrada do programa	48	0
Número de arquivos de saida do programa	341464	0

Com os dados da tabela podemos concluir que realmente os programas são IO e CPU Bound como esperado, já que no IO Bound obtivemos 341464 arquivos de saída e 48 de entrada. Já o processo CPU Bound não possui arquivos de entrada e saída, além disso possui um tempo mínimo em modo Kernel, provavelmente utilizado quando é solicitado ao usuário o dígito a ser analisado.