SSC0640 - Sistemas Operacionais I

Trabalho 1
Processos de Chamadas de sistema, CPU-bound e I-O bound

Grupo: gso03
Higor Tessari - nºUSP: 10345251
Lucas Tavares dos Santos - nºUSP: 10295180
Renata Oliveira Brito - nºUSP: 10373663

Link dos códigos no gitHub

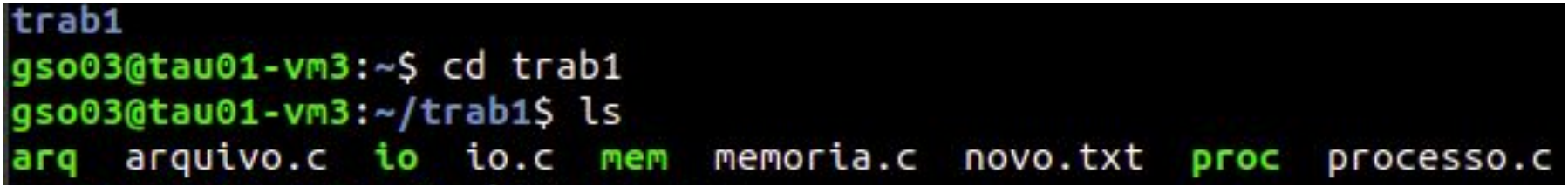
https://github.com/lucast98/Sistemas-Operacionais-gso3

Link da explicação em vídeo

https://youtu.be/OijL9ROmKZY

Parte 1: Chamadas ao Sistema

Dentro da máquina virtual foram criados 4 arquivos de códigos que correspondem aos processos de chamadas



arquivo.c, io.c, memoria.c e processo.c

Parte 1: Chamadas ao Sistema

Foram utilizadas as seguintes bibliotecas em nossa solução:

<stdio.h> que contém várias funções de entrada e saída, como exemplo as funções de printf do nosso programa.

<string.h> contém uma série de funções para manipular strings, como por exemplo a strcpy.

<unistd.h> é a biblioteca que fornece acesso à api do sistema operacional, com essa podemos fazer o uso da função write e close por exemplo.

<termios.h> essa biblioteca contém as definições usadas pelas interfaces de E/S do terminal, onde tem por exemplo o tempo da taxa de transmissão de entrada e saída que são armazenadas na estrutura termios.

<sys/stat.h> essa biblioteca foi necessária pois utilizamos os modos de operação do arquivo do tipo mode_t.

<fcntl.h> essa biblioteca foi utilizada pois usamos modos de controle do arquivo, ela define os argumentos utilizados na função open.

<sys/mman.h> essa biblioteca contém declarações de gerenciamento de memória.

<stdlib.h> essa biblioteca contém funções de alocação de memória e controle de processos.

<sys/wait.h> define as constantes simbólicas para uso com waitpid.

Parte 1: arquivo.c

Para a solução do arquivo.c, criaremos um arquivo, e faremos três operações, abertura, escrita, e por fim fechamos o arquivo.
Foram escolhidas para isso as 4 chamadas ao sistema:

- **Creat**
- **Open**
- **Write**
- **Close**

```
gso03@tau01-vm3:~/trab1$ cat arquivo.c
#include<stdio.h>
#include<unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

/** System calls usadas: creat(nome, modo), open(arquivo, como, ...),
write(fd, buffer, nbytes) e close(fd) */
int main() {
    int fd, s; // fd = diretorio do arquivo, s = retorno caso de erro
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH; // modos de o
peração do arquivo
    char *filename = "novo.txt"; // nome do arquivo a ser criado
    fd = creat(filename, mode); //cria novo arquivo, passando como par
amentro o mode, caso bem sucedido retorna >= 0
    if(fd<0){
        return -1;
    }
    fd = open(filename, O_WRONLY | O_APPEND); //abre o arquivo
    if(fd<0){
        return -1;
    }
    write(fd, "texto", 5); //escreve "texto" no arquivo

    s = close(fd); // fecha o arquivo
    if(s == -1) //erro
        printf("Arquivo nao foi fechado corretamente!\n");
    else
        printf("Arquivo fechado corretamente!\n");

    return 0;
}
```

Parte 1: arquivo.c

Creat:

Uma maneira de criar um novo arquivo a partir de:

fd = creat(“abc”, mode)

Onde abc é o nome do arquivo que será criado e o mode são os bits determinam quais usuários podem acessar o arquivo e como, na solução foram utilizados os seguintes modos:

S_IRUSR/S_IWUSR: lê/escreve o bit de permissão do arquivo para o usuário dono do arquivo.
S_IRGRP: lê o bit de permissão do arquivo para o grupo dono do arquivo.
S_IROTH: lê o bit de permissão do arquivo para outros usuários.

Para que as próximas chamadas possam acessar o arquivo, basta utilizar o descritor de arquivo (“fd”).

```
gso03@tau01-vm3:~/trab1$ cat arquivo.c
#include<stdio.h>
#include<unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

/** System calls usadas: creat(nome, modo), open(arquivo, como, ...),
write(fd, buffer, nbytes) e close(fd) */
int main() {
    int fd, s; // fd = diretorio do arquivo, s = retorno caso de erro
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH; // modos de o
peração do arquivo
    char *filename = "novo.txt"; // nome do arquivo a ser criado
    fd = creat(filename, mode); //cria novo arquivo, passando como par
amentro o mode, caso bem sucedido retorna >= 0
    if(fd<0){
        return -1;
    }
    fd = open(filename, O_WRONLY | O_APPEND); //abre o arquivo
    if(fd<0){
        return -1;
    }
    write(fd, "texto", 5); //escreve "texto" no arquivo

    s = close(fd); // fecha o arquivo
    if(s == -1) //erro
        printf("Arquivo nao foi fechado corretamente!\n");
    else
        printf("Arquivo fechado corretamente!\n");

    return 0;
}
```

Parte 1: arquivo.c

Open:

Abre um arquivo para leitura, escrita ou ambos a partir de:

fd = open (nome do caminho, flags)

Onde abre o arquivo especificado pelo nome do caminho, e seu valor de retorno é um descritor de arquivos, se o arquivo nao existe, basta enviar uma determinada flag e o arquivo é criado e aberto. Na solução foram utilizadas as seguintes flags:

O_WRONLY: Aberto apenas para gravação.
O_APPEND: Defina o modo de acréscimo, escreve apenas no final do arquivo.

```
gso03@tau01-vm3:~/trab1$ cat arquivo.c
#include<stdio.h>
#include<unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

/** System calls usadas: creat(nome, modo), open(arquivo, como, ...),
write(fd, buffer, nbytes) e close(fd) */
int main() {
    int fd, s; // fd = diretorio do arquivo, s = retorno caso de erro
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH; // modos de o
peração do arquivo
    char *filename = "novo.txt"; // nome do arquivo a ser criado
    fd = creat(filename, mode); //cria novo arquivo, passando como par
amentro o mode, caso bem sucedido retorna >= 0
    if(fd<0){
        return -1;
    }
    fd = open(filename, O_WRONLY | O_APPEND); //abre o arquivo
    if(fd<0){
        return -1;
    }
    write(fd, "texto", 5); //escreve "texto" no arquivo

    s = close(fd); // fecha o arquivo
    if(s == -1) //erro
        printf("Arquivo nao foi fechado corretamente!\n");
    else
        printf("Arquivo fechado corretamente!\n");

    return 0;
}
```

Parte 1: arquivo.c

Write:

Escreve dados de um buffer para um arquivo a partir de:

write(fd, buffer, nbytes)

Onde fd é o descritor de arquivo dizendo qual arquivo aberto para escrita, o buffer, um endereço de buffer dizendo onde vai tirar os dados para escrita e uma contagem de bytes, nbytes dizendo quantos bytes vão ser escritos.

```
gso03@tau01-vm3:~/trab1$ cat arquivo.c
#include<stdio.h>
#include<unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

/** System calls usadas: creat(nome, modo), open(arquivo, como, ...),
write(fd, buffer, nbytes) e close(fd) */
int main() {
    int fd, s; // fd = diretorio do arquivo, s = retorno caso de erro
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH; // modos de o
peração do arquivo
    char *filename = "novo.txt"; // nome do arquivo a ser criado
    fd = creat(filename, mode); //cria novo arquivo, passando como par
amentro o mode, caso bem sucedido retorna >= 0
    if(fd<0){
        return -1;
    }
    fd = open(filename, O_WRONLY | O_APPEND); //abre o arquivo
    if(fd<0){
        return -1;
    }
    write(fd, "texto", 5); //escreve "texto" no arquivo

    s = close(fd); // fecha o arquivo
    if(s == -1) //erro
        printf("Arquivo nao foi fechado corretamente!\n");
    else
        printf("Arquivo fechado corretamente!\n");

    return 0;
}
```

Parte 1: arquivo.c

Close:

Uma maneira de fechar um arquivo é a partir de:

n = close(fd)

Onde fd é o descritor de arquivo, e então ele é passado como parâmetro para que ele não se refira mais a nenhum arquivo e possa ser reutilizado. Retorna zero em caso de sucesso. Em caso de erro, -1.

```
gso03@tau01-vm3:~/trab1$ cat arquivo.c
#include<stdio.h>
#include<unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

/** System calls usadas: creat(nome, modo), open(arquivo, como, ...),
write(fd, buffer, nbytes) e close(fd) */
int main() {
    int fd, s; // fd = diretorio do arquivo, s = retorno caso de erro
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH; // modos de o
peração do arquivo
    char *filename = "novo.txt"; // nome do arquivo a ser criado
    fd = creat(filename, mode); //cria novo arquivo, passando como par
amentro o mode, caso bem sucedido retorna >= 0
    if(fd<0){
        return -1;
    }
    fd = open(filename, O_WRONLY | O_APPEND); //abre o arquivo
    if(fd<0){
        return -1;
    }
    write(fd, "texto", 5); //escreve "texto" no arquivo

    s = close(fd); // fecha o arquivo
    if(s == -1) //erro
        printf("Arquivo nao foi fechado corretamente!\n");
    else
        printf("Arquivo fechado corretamente!\n");

    return 0;
}
```

Parte 1: io.c

Nessa solução, alteramos a taxa de transferência de entrada/saída para B9600.
Utilizamos 4 chamadas ao sistema:

- **Tcgetattr**
- **Cfgetospeed**
- **Cfsetospeed**
- **Tcsetattr**

Os seguintes valores de see_speed, são valores válidos para objetos do tipo speed_t, são definidos dentro da estrutura do termios, sendo a taxa de transmissão de entrada e saída, sempre vai entrar em algum desses casos, mas nem todas as taxas de transmissão precisam ser suportadas pelo hardware subjacente.

```
gso03@tau01-vm3:~/trab1$ cat io.c
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<termios.h>

/** System calls usadas: cfsetospeed(&
e de saida,
                        cfgetospeed(&
e de saida,
                        tcsetattr(fd,
tos da velocidade,
                        tcgetattr(fd,

/* Função que auxilia o print da veloc
ntecer algum dos valores abaixo */
char *see_speed(speed_t speed) {
    static char   SPEED[20];
    switch (speed) {
        case B0:
            strcpy(SPEED, "B0");
            break;
        case B50:
            strcpy(SPEED, "B50");
            break;
        case B75:
            strcpy(SPEED, "B75");
            break;
        case B110:
            strcpy(SPEED, "B110");
            break;
        case B134:
            strcpy(SPEED, "B134");
            break;i
```

```
            strcpy(SPEED, "B75");
            break;
        case B110:
            strcpy(SPEED, "B110");
            break;
        case B134:
            strcpy(SPEED, "B134");
            break;i
        case B150:
            strcpy( SPEED, "B150");
            break;
        case B200:
            strcpy(SPEED, "B200");
            break;
        case B300:
            strcpy(SPEED, "B300");
            break;
        case B600:
            strcpy(SPEED, "B600");
            break;
        case B1200:
            strcpy(SPEED, "B1200");
            break;
        case B1800:
            strcpy(SPEED, "B1800");
            break;
        case B2400:
            strcpy(SPEED, "B2400");
            break;
        case B4800:
            strcpy(SPEED, "B4800");
            break;
        case B9600:
            strcpy(SPEED, "B9600");
            break;
        case B19200:
```

Parte 1: io.c

Tcgetattr:

Uma maneira de obter os atributos de saída é a partir de:

tcgetattr (int fd, struct termios * termios_p)

Onde fd é o descritor de dispositivo dentro do sistema (em nosso exemplo, usamos o STDOUT_FILENO, onde é apenas uma entrada padrão) e o endereço de uma estrutura de dados que permite associar atributos de transferências de I/O, que é do tipo termios (biblioteca que possui definições usadas pelo terminal em interfaces I/O),

```
        case B19200:
            strcpy(SPEED, "B19200");
            break;
        case B38400:
            strcpy(SPEED, "B38400");
            break;
        default:
            sprintf(SPEED, "unknown (%d)", (int)speed);
    }
    return SPEED;
}

int main() {
    speed_t speed; // taxa de velocidade de transferencia
    struct termios termAttr; // conjunto de instruções para interface
de io do terminal

    tcgetattr(STDOUT_FILENO, &termAttr); /** obtém os atributos de sai
da*/
    speed = cfgetospeed(&termAttr); /** obtem a velocidade de saida do
 que foi feito em termAttr */
    printf("Velocidade de saida: %s\n", see_speed(speed));
    if (speed != B9600){
        cfsetospeed(&termAttr, B9600); /** ajusta a velocidade de said
a para B9600, pode ser qualquer um estamos apenas utilizando a funcao,
  se ja nao for */
        tcsetattr(STDOUT_FILENO, TCSADRAIN, &termAttr); /** ajusta os
atributos de saida */
        speed = cfgetospeed(&termAttr); /** obtem a nova velocidade de
 saida do que foi feito em termAttr */
        printf("Nova velocidade de saida: %s\n", see_speed(speed));
    }

    return 0;
}
```

Parte 1: io.c

Cfgetospeed:

Para obter a taxa de transmissão de saída utilizamos:

s = cfgetospeed(&termio)

s então é a taxa de transmissão de saída armazenada na estrutura termios apontada por termio.

```
        case B19200:
            strcpy(SPEED, "B19200");
            break;
        case B38400:
            strcpy(SPEED, "B38400");
            break;
        default:
            sprintf(SPEED, "unknown (%d)", (int)speed);
    }
    return SPEED;
}

int main() {
    speed_t speed; // taxa de velocidade de transferencia
    struct termios termAttr; // conjunto de instruções para interface
de io do terminal

    tcgetattr(STDOUT_FILENO, &termAttr); /** obtém os atributos de sai
da*/
    speed = cfgetospeed(&termAttr); /** obtem a velocidade de saida do
 que foi feito em termAttr */
    printf("Velocidade de saida: %s\n", see_speed(speed));
    if (speed != B9600){
        cfsetospeed(&termAttr, B9600); /** ajusta a velocidade de said
a para B9600, pode ser qualquer um estamos apenas utilizando a funcao,
  se ja nao for */
        tcsetattr(STDOUT_FILENO, TCSADRAIN, &termAttr); /** ajusta os
atributos de saida */
        speed = cfgetospeed(&termAttr); /** obtem a nova velocidade de
 saida do que foi feito em termAttr */
        printf("Nova velocidade de saida: %s\n", see_speed(speed));
    }

    return 0;
}
```

Parte 1: io.c

Cfsetospeed:

Alteramos a taxa de transmissão de saída armazenada na estrutura termios apontada por termios para speed (B9600), uma das constantes declaradas anteriormente em see_speed a partir de :

cfsetospeed(&termios, speed)

```
        case B19200:
            strcpy(SPEED, "B19200");
            break;
        case B38400:
            strcpy(SPEED, "B38400");
            break;
        default:
            sprintf(SPEED, "unknown (%d)", (int)speed);
    }
    return SPEED;
}

int main() {
    speed_t speed; // taxa de velocidade de transferencia
    struct termios termAttr; // conjunto de instruções para interface
de io do terminal

    tcgetattr(STDOUT_FILENO, &termAttr); /** obtém os atributos de sai
da*/
    speed = cfgetospeed(&termAttr); /** obtem a velocidade de saida do
 que foi feito em termAttr */
    printf("Velocidade de saida: %s\n", see_speed(speed));
    if (speed != B9600){
        cfsetospeed(&termAttr, B9600); /** ajusta a velocidade de said
a para B9600, pode ser qualquer um estamos apenas utilizando a funcao,
  se ja nao for */
        tcsetattr(STDOUT_FILENO, TCSADRAIN, &termAttr); /** ajusta os
atributos de saida */
        speed = cfgetospeed(&termAttr); /** obtem a nova velocidade de
 saida do que foi feito em termAttr */
        printf("Nova velocidade de saida: %s\n", see_speed(speed));
    }

    return 0;
}
```

Parte 1: io.c

Tcsetattr:

Ajustamos os atributos de saída a partir de:

tcsetattr(fd, opt, &termios)

Onde fd é o descritor de dispositivo dentro do sistema (STDOUT_FILENO), uma ação opcional (TCSADRAIN, que faz com altere os atributos quando a saída estiver esgotada.) e o endereço de uma struct termios.

Por fim utilizamos a cfgetospeed para mostrar a nova taxa alterada.

```
        case B19200:
            strcpy(SPEED, "B19200");
            break;
        case B38400:
            strcpy(SPEED, "B38400");
            break;
        default:
            sprintf(SPEED, "unknown (%d)", (int)speed);
    }
    return SPEED;
}

int main() {
    speed_t speed; // taxa de velocidade de transferencia
    struct termios termAttr; // conjunto de instruções para interface
de io do terminal

    tcgetattr(STDOUT_FILENO, &termAttr); /** obtém os atributos de sai
da*/
    speed = cfgetospeed(&termAttr); /** obtem a velocidade de saida do
 que foi feito em termAttr */
    printf("Velocidade de saida: %s\n", see_speed(speed));
    if (speed != B9600){
        cfsetospeed(&termAttr, B9600); /** ajusta a velocidade de said
a para B9600, pode ser qualquer um estamos apenas utilizando a funcao,
  se ja nao for */
        tcsetattr(STDOUT_FILENO, TCSADRAIN, &termAttr); /** ajusta os
atributos de saida */
        speed = cfgetospeed(&termAttr); /** obtem a nova velocidade de
 saida do que foi feito em termAttr */
        printf("Nova velocidade de saida: %s\n", see_speed(speed));
    }

    return 0;
}
```

Parte 1: memoria.c

Nessa solução, primeiramente mapeamos um arquivo na memória e, depois, se possível, removemos esse mapeamento.
Utilizamos 3 chamadas ao sistema:

- **Brk**
- **Mmap**
- **Munmap**

```
gso03@tau01-vm3:~/trab1$ cat memoria.c
#include<stdio.h>
#include<unistd.h>
#include<sys/mman.h>

/** System calls usadas: brk(addr), muda o tamanho do segmento de dado
s
                        mmap(addr, len, prot, flags, fd, offset) mape
ia o arquivo na memoria
                        munmap(addr, len) remove o mapeamento */
int main() {
    size_t size = getpagesize(); /** tamanho da pagina */
    int s = brk(0); /** especifica o tamanho do segmento de dados, ini
cialmente com 0 */
    char *map = mmap(0, size, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANON
|MAP_PRIVATE, 0, 0); /** mapeia um arquivo na memoria */
    if(map == MAP_FAILED)
        printf("Nao foi possivel mapear o arquivo na memoria\n");
    else{
        printf("Arquivo mapeado com sucesso!\n");
        printf("Agora tentaremos remover esse mapeamento...\n");
        int unmap = munmap(0, size); /** tenta remover o mapeamento do
 arquivo */
        if(unmap == -1)
            printf("O mapemento do arquivo nao foi removido\n");
        else
            printf("O mapemento do arquivo foi removido\n");

    }
    return 0;
}
```

Parte 1: memoria.c

Brk:

Para conseguir especificar o tamanho do segmento de dados, usaremos:

s = brk(addr)

Passamos um valor especificado por end_data_segment (no exemplo usamos 0, para pegarmos o início do segmento), o endereço do primeiro byte além dele.

```
gso03@tau01-vm3:~/trab1$ cat memoria.c
#include<stdio.h>
#include<unistd.h>
#include<sys/mman.h>

/** System calls usadas: brk(addr), muda o tamanho do segmento de dado
s
                        mmap(addr, len, prot, flags, fd, offset) mape
ia o arquivo na memoria
                        munmap(addr, len) remove o mapeamento */
int main() {
    size_t size = getpagesize(); /** tamanho da pagina */
    int s = brk(0); /** especifica o tamanho do segmento de dados, ini
cialmente com 0 */
    char *map = mmap(0, size, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANON
|MAP_PRIVATE, 0, 0); /** mapeia um arquivo na memoria */
    if(map == MAP_FAILED)
        printf("Nao foi possivel mapear o arquivo na memoria\n");
    else{
        printf("Arquivo mapeado com sucesso!\n");
        printf("Agora tentaremos remover esse mapeamento...\n");
        int unmap = munmap(0, size); /** tenta remover o mapeamento do
 arquivo */
        if(unmap == -1)
            printf("O mapemento do arquivo nao foi removido\n");
        else
            printf("O mapemento do arquivo foi removido\n");

    }
    return 0;
}
```

Parte 1: memoria.c

Mmap:

E então mapeamos o arquivo na memória a partir de:

a = mmap(addr,len,prot,flags,fd,offset)

É necessário passar diversos parâmetros: o primeiro, addr, determina o endereço no qual o arquivo está mapeado (utilizamos o 0); o segundo, len, diz quantos bytes mapear (utilizamos o tamanho que pegamos a partir do getpagesize); o terceiro, prot, determina a proteção do arquivo mapeado utilizamos:

PROT_READ: página pode ser lida.
PROT_WRITE: página pode ser escrita.
PROT_EXEC: página pode ser executada.

```
gso03@tau01-vm3:~/trab1$ cat memoria.c
#include<stdio.h>
#include<unistd.h>
#include<sys/mman.h>

/** System calls usadas: brk(addr), muda o tamanho do segmento de dado
s
                        mmap(addr, len, prot, flags, fd, offset) mape
ia o arquivo na memoria
                        munmap(addr, len) remove o mapeamento */
int main() {
    size_t size = getpagesize(); /** tamanho da pagina */
    int s = brk(0); /** especifica o tamanho do segmento de dados, ini
cialmente com 0 */
    char *map = mmap(0, size, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANON
|MAP_PRIVATE, 0, 0); /** mapeia um arquivo na memoria */
    if(map == MAP_FAILED)
        printf("Nao foi possivel mapear o arquivo na memoria\n");
    else{
        printf("Arquivo mapeado com sucesso!\n");
        printf("Agora tentaremos remover esse mapeamento...\n");
        int unmap = munmap(0, size); /** tenta remover o mapeamento do
 arquivo */
        if(unmap == -1)
            printf("O mapemento do arquivo nao foi removido\n");
        else
            printf("O mapemento do arquivo foi removido\n");

    }
    return 0;
}
```

Parte 1: memoria.c

Mmap:

O quarto, flags, controla se um arquivo é privado ou anonimo, utilizamos:

MAP_ANON: anonimo
MAP_PRIVATE: privado

E se addr é uma exigência ou não; o quinto, fd, é o descritor de arquivo para o arquivo a ser mapeado (apenas arquivos abertos podem ser mapeados, utilizamos o 0, pois queremos o inicio do segmento); o sexto, offset, diz onde no arquivo deve começar o mapeamento (como queremos começar do início, utilizamos o 0).

```
gso03@tau01-vm3:~/trab1$ cat memoria.c
#include<stdio.h>
#include<unistd.h>
#include<sys/mman.h>

/** System calls usadas: brk(addr), muda o tamanho do segmento de dado
s
                        mmap(addr, len, prot, flags, fd, offset) mape
ia o arquivo na memoria
                        munmap(addr, len) remove o mapeamento */
int main() {
    size_t size = getpagesize(); /** tamanho da pagina */
    int s = brk(0); /** especifica o tamanho do segmento de dados, ini
cialmente com 0 */
    char *map = mmap(0, size, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANON
|MAP_PRIVATE, 0, 0); /** mapeia um arquivo na memoria */
    if(map == MAP_FAILED)
        printf("Nao foi possivel mapear o arquivo na memoria\n");
    else{
        printf("Arquivo mapeado com sucesso!\n");
        printf("Agora tentaremos remover esse mapeamento...\n");
        int unmap = munmap(0, size); /** tenta remover o mapeamento do
 arquivo */
        if(unmap == -1)
            printf("O mapemento do arquivo nao foi removido\n");
        else
            printf("O mapemento do arquivo foi removido\n");

    }
    return 0;
}
```

Parte 1: memoria.c

Munmap:

Remove o mapeamento do arquivo o mapeamento a partir de:

s = unmap(addr, len)

Passamos o addr (utilizamos o 0), para determinar o endereço no qual o arquivo está mapeado e len (utilizamos o tamanho da página), que diz quantos bytes desmapear, sendo que, assim, conseguimos remover um arquivo mapeado.

```
gso03@tau01-vm3:~/trab1$ cat memoria.c
#include<stdio.h>
#include<unistd.h>
#include<sys/mman.h>

/** System calls usadas: brk(addr), muda o tamanho do segmento de dado
s
                        mmap(addr, len, prot, flags, fd, offset) mape
ia o arquivo na memoria
                        munmap(addr, len) remove o mapeamento */
int main() {
    size_t size = getpagesize(); /** tamanho da pagina */
    int s = brk(0); /** especifica o tamanho do segmento de dados, ini
cialmente com 0 */
    char *map = mmap(0, size, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANON
|MAP_PRIVATE, 0, 0); /** mapeia um arquivo na memoria */
    if(map == MAP_FAILED)
        printf("Nao foi possivel mapear o arquivo na memoria\n");
    else{
        printf("Arquivo mapeado com sucesso!\n");
        printf("Agora tentaremos remover esse mapeamento...\n");
        int unmap = munmap(0, size); /** tenta remover o mapeamento do
 arquivo */
        if(unmap == -1)
            printf("O mapemento do arquivo nao foi removido\n");
        else
            printf("O mapemento do arquivo foi removido\n");

    }
    return 0;
}
```

Parte 1: processo.c

Nessa solução, criaremos um processo filho a partir do seu pai, o pai espera o processo filho terminar de agir e o filho executa uma nova funcionalidade imposta a ele.
Utilizamos 3 chamadas de sistemas

- **Fork**
- **Waitpid**
- **Execve**

```
gso03@tau01-vm3:~/trab1$ cat processo.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

/** System calls usadas: fork(), cria o processo que é identico ao atu
al
                        waitpid(pid, &statloc, opts) faz o processo p
ai esperar o que processo filho esta processando
                        execve(name, argv, envp) antes o filho é igua
l ao pai, com isso ele substitui a funcionalidade antiga */
int main() {
    int pid, status; // pid = identificador de processos
    char *args[] = {"/bin/ls", NULL};

    pid = fork(); /** cria um processo filho */
    if(pid < 0){ /** condicao de erro */
        printf("Erro no fork");
        return 0;
    }
    if(pid != 0){
        printf("Criado o processo %d\n", pid);
        waitpid(-1, &status, 0); /** processo pai espera o filho */
        printf("Processo pai espera o filho (%d) agir\n", pid);
    }else{
        if(execve("/bin/ls", args, NULL) == -1){ /** filho executa sua
 nova funcionalidade, retorna um valor e printa os arquivos do diretor
io atual, comando ls */
            printf("Erro");
        }
    }
    return 0;
}
```

Parte 1: processo.c

Fork:

Para criamos um novo processo, chamado de processo filho, basta fazer:

pid = fork()

Não é necessário passar parâmetros, pois ele apenas cria uma cópia de um processo qualquer, retornando uma identificação do pai na variável pid.

```
gso03@tau01-vm3:~/trab1$ cat processo.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

/** System calls usadas: fork(), cria o processo que é identico ao atu
al
                        waitpid(pid, &statloc, opts) faz o processo p
ai esperar o que processo filho esta processando
                        execve(name, argv, envp) antes o filho é igua
l ao pai, com isso ele substitui a funcionalidade antiga */
int main() {
    int pid, status; // pid = identificador de processos
    char *args[] = {"/bin/ls", NULL};

    pid = fork(); /** cria um processo filho */
    if(pid < 0){ /** condicao de erro */
        printf("Erro no fork");
        return 0;
    }
    if(pid != 0){
        printf("Criado o processo %d\n", pid);
        waitpid(-1, &status, 0); /** processo pai espera o filho */
        printf("Processo pai espera o filho (%d) agir\n", pid);
    }else{
        if(execve("/bin/ls", args, NULL) == -1){ /** filho executa sua
 nova funcionalidade, retorna um valor e printa os arquivos do diretor
io atual, comando ls */
            printf("Erro");
        }
    }
    return 0;
}
```

Parte 1: processo.c

Waitpid:

E então fazemos o processo pai esperar até o processo filho terminar a partir de:

waitpid(pid, &statloc, opts)

É necessário passar como parâmetros um chamador para esperar por um filho específico (-1, qualquer filho serve), o segundo é o endereço de uma variável que receberá o estado de saída do filho (utilizamos a variável com nome status), permitindo que o pai saiba o destino do seu filho, o terceiro determina se o chamador bloqueia ou retorna se nenhum filho já tiver sido terminado.

```
gso03@tau01-vm3:~/trab1$ cat processo.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

/** System calls usadas: fork(), cria o processo que é identico ao atu
al
                        waitpid(pid, &statloc, opts) faz o processo p
ai esperar o que processo filho esta processando
                        execve(name, argv, envp) antes o filho é igua
l ao pai, com isso ele substitui a funcionalidade antiga */
int main() {
    int pid, status; // pid = identificador de processos
    char *args[] = {"/bin/ls", NULL};

    pid = fork(); /** cria um processo filho */
    if(pid < 0){ /** condicao de erro */
        printf("Erro no fork");
        return 0;
    }
    if(pid != 0){
        printf("Criado o processo %d\n", pid);
        waitpid(-1, &status, 0); /** processo pai espera o filho */
        printf("Processo pai espera o filho (%d) agir\n", pid);
    }else{
        if(execve("/bin/ls", args, NULL) == -1){ /** filho executa sua
 nova funcionalidade, retorna um valor e printa os arquivos do diretor
io atual, comando ls */
            printf("Erro");
        }
    }
    return 0;
}
```

Parte 1: processo.c

Execve:

Para substituir a imagem da memória de um processo, fazemos a partir de:

execve(name, args, envp)

É necessário passar como parâmetros o name, que é uma contagem do número de itens na linha de comando, incluindo o nome do programa (utilizamos "/bin/ls"). O segundo parâmetro, args, é um ponteiro para um conjunto. O elemento i do conjunto é um ponteiro para a i-ésima cadeia na linha de comando. O terceiro parâmetro, envp, é um ponteiro para o ambiente, um conjunto de cadeias contendo designações da forma nome = valor usadas para passar informações como tipo de terminal e nome do diretório--raiz para um programa.

Fazemos com isso, que o processo filho execute o comando “ls” no terminal.

```
gso03@tau01-vm3:~/trab1$ cat processo.c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

/** System calls usadas: fork(), cria o processo que é identico ao atu
al
                        waitpid(pid, &statloc, opts) faz o processo p
ai esperar o que processo filho esta processando
                        execve(name, argv, envp) antes o filho é igua
l ao pai, com isso ele substitui a funcionalidade antiga */
int main() {
    int pid, status; // pid = identificador de processos
    char *args[] = {"/bin/ls", NULL};

    pid = fork(); /** cria um processo filho */
    if(pid < 0){ /** condicao de erro */
        printf("Erro no fork");
        return 0;
    }
    if(pid != 0){
        printf("Criado o processo %d\n", pid);
        waitpid(-1, &status, 0); /** processo pai espera o filho */
        printf("Processo pai espera o filho (%d) agir\n", pid);
    }else{
        if(execve("/bin/ls", args, NULL) == -1){ /** filho executa sua
 nova funcionalidade, retorna um valor e printa os arquivos do diretor
io atual, comando ls */
            printf("Erro");
        }
    }
    return 0;
}
```

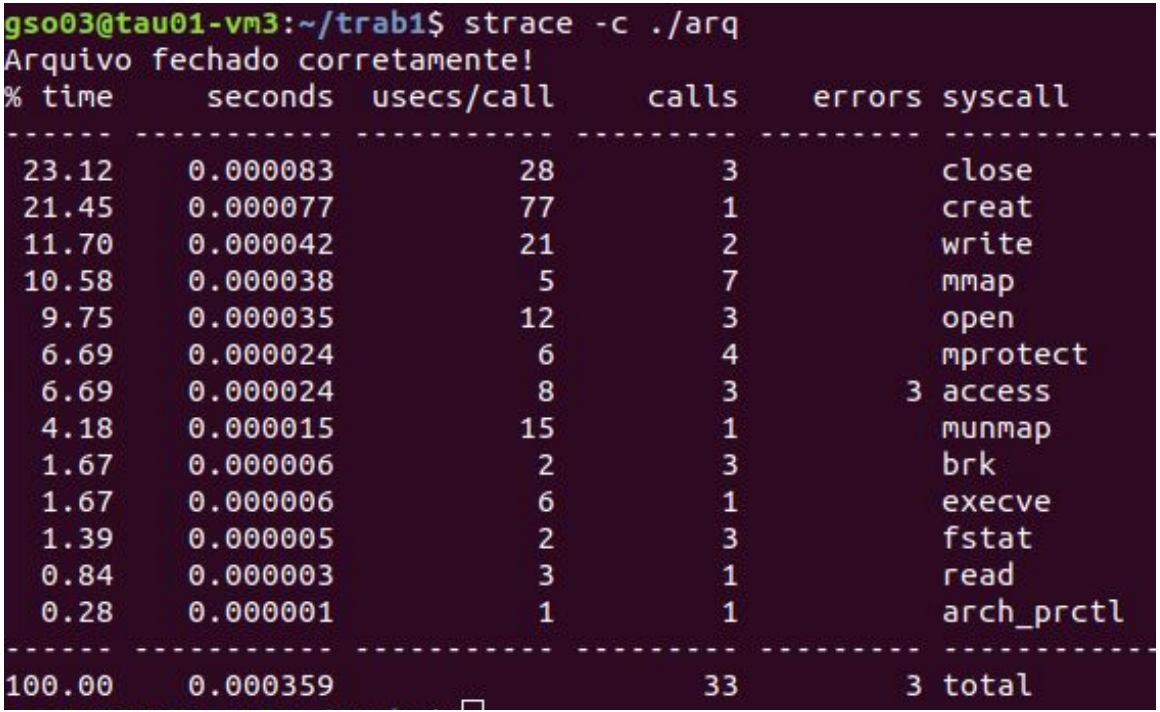
Parte 1: Interação

Após a realização dos programas dentro e do diretório onde se encontra os mesmos, segue os seguintes passos para a execução:

- Criar os binários com “gcc -o nome_exec nome_arq.c”
- Executá-los com “strace -c ./nome_exec”, pois com essa ferramenta obtivemos as estatísticas relacionadas ao uso de chamadas ao sistema em cada um dos programas desenvolvidos.

Parte 1: Relação das system calls do arquivo.c

No programa de arquivos, todas as chamadas utilizadas estão listadas. A chamada close é a que demanda mais tempo, pois é necessário que ela faça um writeback para garantir confirmar que todos os erros sejam reportados. Da mesma forma, creat também leva um bom tempo de execução. Write é usado para inserir texto no arquivo, configurando uma processo de E/S, sendo mais lento. Open é utilizado para abrir o arquivo e confirmar se está tudo certo, sendo também uma das syscalls mais utilizadas.



```
gso03@tau01-vm3:~/trab1$ strace -c ./arq
Arquivo fechado corretamente!
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ------------
 23.12    0.000083          28         3           close
 21.45    0.000077          77         1           creat
 11.70    0.000042          21         2           write
 10.58    0.000038           5         7           mmap
  9.75    0.000035          12         3           open
  6.69    0.000024           6         4           mprotect
  6.69    0.000024           8         3         3 access
  4.18    0.000015          15         1           munmap
  1.67    0.000006           2         3           brk
  1.67    0.000006           6         1           execve
  1.39    0.000005           2         3           fstat
  0.84    0.000003           3         1           read
  0.28    0.000001           1         1           arch_prctl
------ ----------- ----------- --------- --------- ------------
100.00    0.000359                    33         3 total
```

Parte 1: Relação das system calls do io.c

No programa de I/O, as chamadas de sistemas utilizadas compartilham uma única chamada de sistema, ioctl, que realiza um grande número de ações específicas dos dispositivos em arquivos especiais.

```
gso03@tau01-vm3:~/trab1$ strace -c ./io
Velocidade de saida: B38400
Nova velocidade de saida: B9600
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ------------
 22.06    0.000015           2         7           mmap
 16.18    0.000011           6         2           write
 14.71    0.000010           3         4           mprotect
 11.76    0.000008           3         3         3 access
 10.29    0.000007           4         2           open
  7.35    0.000005           5         1           munmap
  5.88    0.000004           1         4           ioctl
  4.41    0.000003           1         3           brk
  2.94    0.000002           2         1           execve
  1.47    0.000001           1         2           close
  1.47    0.000001           0         3           fstat
  1.47    0.000001           1         1           arch_prctl
  0.00    0.000000           0         1           read
------ ----------- ----------- --------- --------- ------------
100.00    0.000068                    34         3 total
```

Parte 1: Relação das system calls do memoria.c

No exemplo do programa de memória, vemos que uma das chamadas mais utilizadas é a write, que é uma chamada de sistema de E/S, responsável por auxiliar no mapeamento o arquivo para um espaço de endereçamento (faz sentido ela usar tanto tempo de execução, mesmo com só 3 chamadas).
O Mmap é o responsável por controlar o processo de mapeamento do arquivo, o que também demanda grande parte do tempo de execução (pois utiliza 8 chamadas!!).
O munmap é mais rápido que mmap, pois é menos complexo e é chamado menos vezes (2 vezes e tb dá pra ver com strace -e trace=memory ./mem).
O brk apenas especifica o tamanho do segmento de dados e, por isso, não é tão utilizado quanto os outros (apenas 3 vezes).
As outras syscalls são padrões de todos os programas, e estão lá para que funcione independente das chamadas utilizadas explicitamente.

```
gso03@tau01-vm3:~/trab1$ strace -c ./mem
Arquivo mapeado com sucesso!
Agora tentaremos remover esse mapeamento...
O mapemento do arquivo foi removido
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
 27.08    0.000052          17         3           write
 19.79    0.000038           5         8           mmap
 13.02    0.000025           6         4           mprotect
 11.46    0.000022           7         3         3 access
  9.90    0.000019          10         2           open
  8.85    0.000017           9         2           munmap
  2.60    0.000005           2         3           fstat
  2.60    0.000005           2         3           brk
  1.56    0.000003           3         1           read
  1.56    0.000003           3         1           execve
  1.04    0.000002           1         2           close
  0.52    0.000001           1         1           arch_prctl
------ ----------- ----------- --------- --------- ----------
100.00    0.000192                    33         3 total
```

Parte 1: Relação das system calls do processo.c

É possível perceber que as chamadas de sistema fork e waitpid não estão listadas. Isso ocorre pois, em C, a chamada de sistema não é realmente invocada, mas sim uma função de biblioteca padrão do C, que representa um empacotamento da chamada. No caso do fork, por exemplo, ao chamar fork(), a implementação invoca clone(2) com flags que dão o mesmo efeito da chamada original. O waitpid funciona da mesma forma, mas invocando wait4(2).

A syscall wait4 é a que mais demora, pois depende de eventos externos para que se encerre.

Já o execve está listado e altera o funcionamento do processo filho, como esperado.

```
gso03@tau01-vm3:~/trab1$ strace -c ./proc
Criado o processo 15399
arq  arq2  arquivo.c  arquivo2.c  io  io.c  mem  memoria.c  no
Processo pai espera o filho (15399) agir
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
 80.92    0.001107        1107         1           wait4
  5.41    0.000074          74         1           clone
  2.85    0.000039           6         7           mmap
  1.90    0.000026          13         2           write
  1.90    0.000026           7         4           mprotect
  1.68    0.000023           8         3         3 access
  1.54    0.000021          11         2           open
  1.17    0.000016          16         1           munmap
  1.02    0.000014          14         1           execve
  0.66    0.000009           3         3           brk
  0.44    0.000006           2         3           fstat
  0.22    0.000003           3         1           read
  0.22    0.000003           2         2           close
  0.07    0.000001           1         1           arch_prctl
------ ----------- ----------- --------- --------- ----------
100.00    0.001368                    32         3 total
```

Parte 1: Descrição de algumas outras syscalls

Quando utilizamos a ferramenta strace, obtivemos as relações de algumas outras syscall além das que nós propusemos para a solução, abaixo segue resumidamente suas descrições:

wait4: Aguarda o processo mudar de estado, pode ser usado para selecionar um filho específico, ou filhos, no qual aguardar.

clone: Cria um processo filho.

ioctl: É uma chamada de núcleo de propósito geral para enviar comandos customizados para drivers e módulos de núcleo).

mprotect: Habilita proteção de acesso em uma região da memória.

access: Confere permissão do usuário por um arquivo.

fstat: Obtém o status do arquivo.

read: Lê uma sequência de bytes do descritor de arquivo.

arch_prctl: Habilita o estado de thread (encadeamento) específico da arquitetura.

Parte 2: Processos CPU-bound e I-O bound

Introdução

Um programa carregado em um sistema ocioso pode, em qualquer momento, estar em um dos 3 estados a seguir:

- Executando um código diretamente, sendo que esse tempo é denotado como user time (u) (ou seja, no contexto do usuário)
- Executado pelo kernel, sendo que esse tempo é denotado como system time (s), ou kernel time.
- Esperando uma operação externa se completar, normalmente de periféricos. Esse tempo é denominado como idle time.

O tempo total que um programa gasta nos três estados é denominado real time (r). A soma dos tempos de programa no modo user e system é também referido como CPU time.

Parte 2: Processos CPU-bound e I-O bound

Como podemos ver da tabela ao lado, programas cujo tempo real r é muito maior que o seu tempo de CPU u+s são chamados de I/O-bound. Tais programas gastam a maior parte do seu tempo ocioso esperando por periféricos mais lentos responderem.

Programas cujo tempo de usuario u são semelhantes ao tempo real r são chamados de CPU-bound. Tais programas são limitados pela velocidade do processador, como cálculos, por exemplo.

Timing Profile	$r \gg u + s$	$s > u$	$u \simeq r$
Characterization	I/O-bound	Kernel-bound	CPU-bound
Diagnostic tools	Disk, network, and virtual memory statistics; network packet dumps; system call tracing	System call tracing	Function profiling; basic block counting
Resolution options	Caching; efficient network protocols and disk data structures; faster I/O interfaces or peripherals	Caching; a faster CPU	Efficient algorithms and data structures; other code improvements; a faster CPU or memory system

Parte 2: Interação

Com os programas feitos e dentro e do diretório onde se encontra os mesmos, segue os seguintes passos para a execução:

- Criar os binários com <gcc -o nome_exec nome_arq.c>
- Executá-los com </usr/bin/time -f "%e geral\n%U usr\n%S sys\n%c inv\n%w vol\n%P CPU\n%I entrada\n%O saida" ./nome_exec>, com isso podemos obter as estatísticas de uso de recursos em cada um dos programas desenvolvidos, como os seguintes parâmetros:

  %e: Tempo real decorrido (em segundos).

  %U: Número total de segundos de CPU que o processo passou no modo de usuário.

  %S: Número total de segundos de CPU que o processo passou no modo kernel.

  %c: Número de vezes que o processo foi alternado involuntariamente (porque o intervalo de tempo expirou).

  %w: Número de esperas: vezes que o programa foi alternado de contexto voluntariamente, por exemplo, enquanto aguarda a conclusão de uma operação de E / S.

  %P: Porcentagem da CPU que esse trabalho obteve, calculada como (% U +% S) /% E.

  %I: Número de entradas do sistema de arquivos pelo processo.

  %O: Número de saídas do sistema de arquivos pelo processo.

Parte 2: Processos CPU-bound

```c
int main (){
    for (volatile unsigned long long i = 0; i < 1000000000ULL; ++i);

    return 0;
}
```

Para nossa solução decidimos utilizar um programa que faz muitas iterações, podendo assim analisarmos a saída com a ferramenta time, e concluirmos que o programa se trata de CPU-bound, pois o tempo de usuario (usr) são semelhantes ao tempo real (geral), e vemos também que o número de entradas e saída do sistema de arquivos pelo processo é 0, com isso não se trata de um programa de I/O-Bound.

```
gso03@tau01-vm3:~/trab1$ /usr/bin/time
cpu
2.44 geral
2.44 usr
0.00 sys
21 inv
1 vol
99% CPU
0 entrada
0 saida
```

Parte 2: Processos I/O-bound

Para nossa solução decidimos utilizar um programa que abre um arquivo (lorem.txt) e cópia cada caracter 10000 vezes para um novo arquivo de texto (novo.txt), podendo assim analisarmos a saída com a ferramenta time, e concluirmos que o programa se trata de I/O-bound, pois o tempo real (geral) é maior que o seu tempo de CPU (usr + sys), e vemos também que o número de saída do sistema de arquivos pelo processo é muito alto, ajudando concluir que se trata de um processo de I/O-bound.

```
gso03@tau01-vm3:~/trab1$ /usr/bin/time
io2
File opened for copy...
17.56 geral
15.27 usr
1.72 sys
388 inv
18 vol
96% CPU
0 entrada
2754008 saida
```

```
gso03@tau01-vm3:~/trab1$ cat io2.c
#include <stdio.h>

void main(){
    FILE *fp1, *fp2;
    char ch;
    int pos;

    if ((fp1 = fopen("lorem.txt","r")) == NULL){
        printf("\nFile cannot be opened");
        return;
    }
    else{
        printf("File opened for copy...\n");
    }
    fp2 = fopen("novo.txt", "w");
    fseek(fp1, 0L, SEEK_END); // file pointer at end of file
    pos = ftell(fp1);
    fseek(fp1, 0L, SEEK_SET); // file pointer set at start
    while (pos--)
    {
        ch = fgetc(fp1);  // copying file character by character
        for(int i=0; i < 10000; i++)
            fputc(ch, fp2);
    }
    fclose(fp1);
    fclose(fp2);
}
```

Bibliografia

- https://syscalls.kernelgrok.com/ - Acesso em 27/03 as 11:00 horário de brasília
- https://linux.die.net/man/1/time - Acesso em 27/03 as 11:00 horário de brasília
- http://man7.org/linux/man-pages/man1/strace.1.html - Acesso em 27/03 as 11:00 horário de brasília
- S.TANENBAUM Andrew; BOS, Herbert. Daniel Vieira e Jorge Ritter. Sistemas operacionais modernos. 4 ed. São Paulo: Pearson Education do Brasil Ltda, 2016.
- SPINELLIS Diomidis. Code Quality: The Open Source Perspective. 1 ed. U.S,Scott meyers Consulting Editor, 2006.