Universidade Federal do Pará

Instituto de Tecnologia

Faculdade de Computação e Telecomunicações

Disciplina: Sistemas Operacionais

Docente: Diego Lisboa Cardoso

Discentes: Adryele Costa de Oliveira - 202206840035

Amanda Gabrielly Prestes Lopes - 202207040043

Felipe Daniel Setubal Alves - 202306840011

Filipe Corrêa da Silva - 202006840020

Lívia Stefane Hipólito Pires - 202206840043

Relatório de Sistemas Operacionais

Belém - Pará

2023

1. Chamadas de Sistemas

Objetivo: O objetivo deste trabalho é demonstrar como os programas interagem com o sistema operacional e como as tarefas comuns, como ler da entrada padrão e escrever na saída padrão, são realizadas usando chamadas de sistemas, que são solicitações feitas ao sistema operacional para realizar várias tarefas.

1.1. Código em Assembly

O código a seguir é um programa simples em Assembly que utiliza chamadas de sistema para interagir com o usuário e o sistema operacional. No Linux, as *syscall* são feitas usando a interrupção de software *int 0x80*, com o número da chamada de sistema especificado no registrador *eax*.

O programa inicia escrevendo uma mensagem para o usuário, solicitando a inserção de um número inteiro. Isso é realizado através da chamada de sistema *write* (representada pelo número 4), que escreve a string armazenada na variável prompt para a saída padrão (*stdout*, representada pelo número 1). Em seguida, é lido um caractere do teclado usando a chamada de sistema *read* (representada pelo número 3). Ele verifica se o caractere é uma quebra de linha, converte o caractere ASCII para número e atualiza a variável *num*. Se o caractere lido não for uma quebra de linha, o código subtrai '0' dele (*sub byte [char_in], '0'*). Isso é feito porque os caracteres numéricos em ASCII são representados pelos valores 48 ('0') a 57 ('9'). Subtraindo '0' (ou seja, 48) do valor ASCII de um caractere numérico dá o valor numérico real. O código então verifica se o número é par ou ímpar. Isso é feito verificando o bit menos significativo do número (se o bit menos significativo for 0, o número é par; se for 1, o número é ímpar).

Finalmente, o código escreve uma mensagem para a saída padrão indicando se o número é par ou ímpar. Isso é feito usando novamente a chamada de sistema *write*. O programa termina usando a chamada de sistema *exit*. Esta chamada de sistema termina o processo atual e retorna o controle ao sistema operacional.

```
section .data
    prompt db "Digite um número inteiro: ", 0
    even_msg db "é par.", 10, 0
    odd_msg db "é ímpar.", 10, 0

section .bss
    num resd 1
    char_in resb 1

section .text
    global _start

_start:
    ; Escreve a mensagem de prompt na tela
    mov eax, 4 ; sys_write
    mov ebx, 1 ; stdout
    mov ecx, prompt ; endereço da string
    mov edx, 27 ; tamanho da string
    int 0x80 ; chama a syscall
read_num:
    ; Lê um caractere do teclado
    mov eax, 3; sys_read
    mov ebx, 0; stdin
    mov ecx, char_in
    mov edx, 1
    int 0x80; chama a syscall

    ; Verifica se o caractere é uma quebra de linha
    cmp byte [char_in], 10
    je paridade; pula para paridade

    ; Converte o caractere ASCII para número
    sub byte [char_in], '0'

    ; Atualiza num (num = num*10 + char_in)
    mov eax, [num]
    imul eax, eax, 10
    add eax, [char_in]
    mov [num], eax

    jmp read_num
paridade:
    ; Verifica se o número é par ou ímpar e escreve o resultado na tela
```

```
    mov eax, [num]
    and eax, 1 ; verifica o bit menos significativo (1 para ímpar, 0
para par)
    jz even ; se for zero (par), pula para a label "even"

odd:
    ; Escreve a mensagem "ímpar" na tela
    mov eax, 4 ; sys_write
    mov ebx, 1 ; stdout
    mov ecx, odd_msg ; endereço da string "ímpar"
    mov edx, 12 ; tamanho da string "ímpar"
    int 0x80 ; chama a syscall

exit:
    ; Sai do programa
    mov eax, 1 ; sys_exit
    xor ebx, ebx ; código de saída (zero indica sucesso)
    int 0x80 ; chama a syscall

even:
    ; Escreve a mensagem "par" na tela
    mov eax, 4 ; sys_write
    mov ebx, 1 ; stdout
    mov ecx, even_msg; endereço da string "par"
    mov edx, 9; tamanho da string "par"
    int 0x80 ; chama a syscall
    jmp exit
```

Compilação e execução do código em Assembly

```
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ nasm -f elf64  Assembly.asm
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ ld Assembly.o -o assembly
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ ./assembly
Digite um número inteiro: 155
é ímpar.
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ ./assembly
Digite um número inteiro: 48
é par.
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ ./assembly
Digite um número inteiro: 53
é ímpar.
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ ./assembly
Digite um número inteiro: 66
é par.
```

1.2. Código em Linguagem C

O código a seguir é um programa simples em C que utiliza chamadas de sistema do Linux para fazer uma interação entre o usuário e o terminal do sistema operacional, assim como mostrado primeiramente em Assembly.

O programa começa escrevendo uma mensagem para o usuário, pedindo que seja inserido um número inteiro. Isso é feito usando a função *write*, que escreve a string armazenada na variável *prompt* para a saída padrão (*STDOUT_FILENO*). Em seguida, é lido o número que o usuário digitou, usando a função *read*. A entrada é lida da entrada padrão (*STDIN_FILENO*) e armazenada na variável *buf*. O código, então, converte a string lida em um número inteiro usando a função *sscanf*. Após, é verificado se o número é par ou ímpar. Isso é feito usando o *operador módulo* (%). Se o número for divisível por 2 (ou seja, num % 2 == 0), então o número é par; caso contrário, é ímpar.

Finalmente, o código escreve uma mensagem para a saída padrão que indica se o número é par ou ímpar. Isso é feito usando novamente a função *write*. O programa termina usando a chamada de sistema *_exit*. Esta chamada de sistema termina o processo atual e retorna o controle ao sistema operacional.

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define BUF_SIZE 1024

int main(void) {
    int num;
    char buf[BUF_SIZE];
    char prompt[] = "Digite um número inteiro: ";
    char even[] = " é par.\n";
    char odd[] = " é ímpar.\n";

    // Escreve a mensagem de prompt na tela
    write(STDOUT_FILENO, prompt, strlen(prompt));

    // Lê o número digitado pelo usuário
    read(STDIN_FILENO, buf, BUF_SIZE);
```

```
    sscanf(buf, "%d", &num);

    // Converte o número para uma string
    int len = sprintf(buf, "%d", num);

    // Verifica se o número é par ou ímpar e escreve o resultado na tela
    if (num % 2 == 0) {
        write(STDOUT_FILENO, buf, len);
        write(STDOUT_FILENO, even, strlen(even));
    } else {
        write(STDOUT_FILENO, buf, len);
        write(STDOUT_FILENO, odd, strlen(odd));
    }

    // Sai do programa usando a chamada de sistema exit
    _exit(0);

    return 0;
}
```

Compilação e execução do código em Linguagem C

```
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ gcc Impar_par.c -o impar_par
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ ./impar_par
Digite um número inteiro: 20
20 é par.
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ ./impar_par
Digite um número inteiro: 63
63 é ímpar.
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ ./impar_par
Digite um número inteiro: 98
98 é par.
filipecorrea@filipecorrea-VirtualBox:~/Downloads$ ./impar_par
Digite um número inteiro: 135
135 é ímpar.
```

2. Threads e Processos

2.1. - 1 Avô, 1 Pai e 1 Filho

Objetivo: Escreva um programa com 1 Avô, 1 Pai e 1 Filho.

Usando a função *Fork* duplicamos o pid1 (que será considerado o avô) que originará o pai. Logo, o pid2 (processo pai) se duplicará usando a função *fork* originando o processo filho. Todos os processos usam o *printf* para que o número do seu pid apareça na tela. Nota-se que a sucessão dos pids ocorre em progressão aritmética no resultado do emulador.

Código:

```c
//etapa 1
//Escreva um programa com 1 Avo, 1 Pai e 1 Filho

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

int main() {
    pid_t pid1, pid2;
    pid1 = fork();
    if (pid1 == 0) {
        // duplicacao de pid1
        pid2 = fork();
        if (pid2 == 0) {
            // processo filho
            printf("Eu sou o processo filho com PID %d\n", getpid());
        } else {
            // processo pai
            printf("Eu sou o processo pai com PID %d\n", getpid());
        }
    } else {
        // processo avo
        printf("Eu sou o processo avo com PID %d\n", getpid());
    }
    return 0;
}
```

Resultado:

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/8$ ./oito
Eu sou o processo avo com PID 159221
Eu sou o processo pai com PID 159222
Eu sou o processo filho com PID 159223
```

2.2. - 1 pai e dois filhos

Objetivo: Escreva um programa com 1 Pai e 2 Filhos.

Usando a função *Fork* há a criação do primeiro processo filho (pid1); se não houver erro o código irá printar na tela o seu pid e o pid do seu pai, ambos usarão, respectivamente, a função "*getpid*() e *getppid*()" que são duas funções que retornam identificadores de processos. O segundo filho (pid2) seguirá o mesmo código de criação de seu irmão o pid1. No final, irá printar na tela o pid do processo pai e o pid dos seus dois filhos. No resultado observa-se que ambos os filhos obtêm o mesmo identificador de número pid de seu processo pai que é 19793, confirmando que os dois foram gerados pelo mesmo processo.

Código:

```c
int main() {
    int pid1, pid2;
    //pid1 e pid2 sao identificadores de variaveis
    pid1 = fork();
    //duplicacao de processo
    if (pid1 == 0)
    //verificando se o processo atual é o processo filho
    {
        printf("Filho 1: Meu PID e %d. O PID do meu pai e %d.\n", getpid(), getppid());
        exit(0);
    //getpid retorna o identificador do processo atual
    //getppid retorna o identificador do pai do processo atual
    }

    pid2 = fork();
    //duplicacao de processo
    if (pid2 == 0) {
        printf("Filho 2: Meu PID e %d. O PID do meu pai e %d.\n", getpid(), getppid());
        exit(0);
    //o 2 filho retorna seu pid e o pid do seu pai
    }

    printf("Pai: Meu PID e %d. Os PIDs dos meus filhos sao %d e %d.\n", getpid(), pid1, pid2);
    //pid pai ira printar na tela seu pid e o pid de seus dois filhos
    return 0;
}
```

Resultado:

```
livia@john-Inspiron-3470:~/Documentos/SO/trabalho2/2$ gcc atividade2.c -o 2
livia@john-Inspiron-3470:~/Documentos/SO/trabalho2/2$ ./2
Pai: Meu PID é 19793. Os PIDs dos meus filhos são 19794 e 19795.
Filho 1: Meu PID é 19794. O PID do meu pai é 19793.
Filho 2: Meu PID é 19795. O PID do meu pai é 19793.
livia@john-Inspiron-3470:~/Documentos/SO/trabalho2/2$
```

2.3. Eliminação de um processo

Objetivo: Escreva um programa com 1 avô, 1 pai e 1 filho, "elimine" o processo pai e veja quem será o novo pai do processo filho.

O processo avô criará o processo pai através da função *fork*, logo o processo pai criará o processo filho utilizando a mesma função. O filho irá mostrar seu identificador PID e o de seu pai. O processo pai irá ser eliminado através da função *kill* da biblioteca de sistema signal.h. Após o pai ser eliminado ocorrerá uma espera de 10s. Na tela irá imprimir que o novo pai do processo filho será o processo avô.

Código:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main() {
    pid_t pid, pid2;

    pid = fork();
    //pid pai criando o filho

    if (pid == 0) {
        printf("Pai: PID=%d\n", getpid());
        exit(0);
        // se o pid for igual a 0, entao ele e o processo filho e mostrara seu pid
    } else if (pid > 0) {
        // se o pid for maior que 0, entao e o processo pai
        pid2 = fork();
        //processo pai se tornando avo
        if (pid2 == 0)
          {
            printf("Filho: PID=%d\n", getpid());
            exit(0);
```

```c
        } else if (pid2 > 0) {
            printf("Avo: PID=%d\n ", getpid());
            sleep(10);
            //apos a duplicacao, encontra-se o processo pai e avo
            //ha uma espera para a finalizacao do processo

            printf("Eliminando processo Pai...\n");
            kill(pid2, SIGKILL);
            sleep(10);
            //processo pai  eliminado pela funcao kill
            //ha uma espera de 10s para que o pai seja morto e entao encontra-se o novo pai do processo filho
            printf("Novo pai do processo Filho e o processo Avo.\n");
            //Quando um processo morre, seus filhos sao adotados pelo processo init, que tem PID = 1

        } else {
            perror("fork");
            exit(1);
        }
    } else {
        perror("fork");
        exit(1);
    }

    return 0;
}
```

Resultado:

```
livia@john-Inspiron-3470:~/Documentos/SO/trabalho2/3$ gcc atividade3.c -o 3
livia@john-Inspiron-3470:~/Documentos/SO/trabalho2/3$ ./3
Avo: PID=19976
Pai: PID=19977
Filho: PID=19978
 Eliminando processo Pai...
Novo pai do processo Filho e o processo Avo.
livia@john-Inspiron-3470:~/Documentos/SO/trabalho2/3$
```

2.4. Threads “ABC”

Objetivo: Crie um programa que cria 3 *threads*. A primeira escreve na tela “A”, a segunda “B” e a terceira “C”. Faça que seja sempre escrito na tela “ABC”

O programa deve criar 3 *threads*, cada *thread* possui uma função imprimir na tela um carácter específico. Cada uma delas é executada em um loop infinito que verifica a variável ***turn*** para determinar se é a sua vez de imprimir na tela. A variável *turn* é inicializada com o valor 0 e é atualizada assim que a *thread* terminar

de ser executada, para que a próxima *thread* possa realizar sua tarefa. Isso garante que as *threads* imprimam na ordem correta.

Código:

```
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 3
volatile int turn = 0;

void* imprime_A(void* threadid) { // uma thread vai rodar essa função
    while(1) { //loop infinito
        while(turn == 0){ //loop q continua quando turn for igual a 0.
Faz a thread esperar sua vez para printar A
        printf("A"); // imprime A na tela
        turn = 1; // muda o valor, passa para proxima thread
        }

    }
}

void* imprime_B(void* threadid) { // uma thread vai rodar essa função
    while(1) { //loop infinito
        while(turn == 1){ //loop q continua quando turn for igual a 1.
Faz a thread esperar sua vez para printar B
        printf("B"); // imprime B na tela
        turn = 2; // muda o valor, passa para proxima thread
        }

    }
}

void* imprime_C(void* threadid) { // uma thread vai rodar essa função
    while(1) { //loop infinito
        while(turn == 2){ //loop q continua quando turn for igual a 2.
Faz a thread esperar sua vez para printar C
        printf("C\n"); // imprime C na tela
        turn = 0; // muda o valor, passa para proxima thread
        }

    }
}

int main() {
    pthread_t thread[NUM_THREADS]; //cria array, cada elemento é um
identificador da thread
```

```
    pthread_create (&thread[0], NULL, imprime_A, NULL); //cria thread 1
e define sua função "impreme_A"
    pthread_create (&thread[1], NULL, imprime_B, NULL); //cria thread 2
e define sua função "impreme_B"
    pthread_create (&thread[2], NULL, imprime_C, NULL); //cria thread 3
e define sua função "impreme_C"

    for(int i=0; i< NUM_THREADS; i++){
        pthread_join(thread[i], NULL); //não permite que a função
principal termine antes de todas as threads forem executadas
    }

    return 0;
}
```

Resultado:

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/4$ ./quatro
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
ABC
```

2.5. Filho altera variável que o Pai imprime

Objetivo: Para o processo Pai e filho, declare uma variável visível ao pai e ao filho, no pai inicialize a variável com 1 e imprima seu valor antes do *fork*(). No filho, altere o valor da variável para 5 e imprima o seu valor entes do *exit*(). Agora, no pai, imprima novamente o valor da variável após o filho ter alterado a variável - após a *waitpid*(). Justifique os resultados obtidos.

O programa declara uma variável de valor 1, que pertence a um processo Pai. O processo Pai é clonado dando origem ao processo Filho que altera a variável 1 do Pai para o valor 5. Em seguida o Pai imprime sua variável que ainda possui o

valor 1, isso porque, o processo Filho é uma cópia perfeita do Pai, sendo assim, ele possui sua própria variável com o valor 1. Quando o Filho altera a variável para o valor 5 isso não altera a variável do Pai, pois, mesmo sendo a cópia do processo Pai, o Filho é um processo isolado e não tem relação com o processo Pai.

Código:

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

int main() {

    //criando a variavel com o pid do pai
    int pid = 1;

    //pid do pai antes de executar o comando fork
    printf("PID Pai: %d\n", pid);

    //comando fork sendo executado (criacao de um filho)
    int ret = fork();

    //se o numero de fork for igual a 0, é o processo filho
    if (ret == 0) {

        //alterando o valor do pid
        pid = 5;

        //valor do pid do filho antes do comando exit
        printf("PID do Filho: %d\n", pid);

        //comando exit ()
        _exit(0);


    } else if (ret > 0) { //se o numero do fork for maior que 0, é o
processo pai
```

```
        // Processo pai espera o processo filho finalizar
        waitpid(NULL);

        //valor final do pid do pai
        printf("Valor final do Pai: %d\n", pid);
    }


    //O valor inicial do PAI é 1 e o final continua o mesmo valor
independentemente da alteração feita pelo Filho
    //Isso acontece pois, o comando fork cria uma copia exata do
processo Pai, porém apos criada a copia (Filho)
    //o processo Pai age isolado do processo filho
    //Isso é, o Filho tem sua propria copia do valor e qualquer
alteracao feita nele não afetará o valor do Pai.

    return 0;
}
```

Resultado:

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/5$ ./cinco
PID Pai: 1
PID do Filho: 5
Valor final do Pai: 1
```

2.6. Filho ordena o vetor e Pai imprime

Objetivo: Ordenar um vetor de 100 posições em processos e threads. O filho ordena o vetor e o pai exibe os dados do vetor antes da criação do filho (*fork*) e depois da espera pelo filho (*waitpid* para processos). Eles usarão o mesmo vetor na memória? Justifique.

O programa deve criar um vetor de 100 posições e aleatoriamente preencher o vetor. O pai imprime o vetor “embaralhado” e logo em seguida o processo Filho é criado. O Filho utiliza a função ***sort*** (função criada para o processo

Filho organizar de forma crescente o vetor) e ordena o vetor e imprime o vetor ordenado. Logo depois, o Pai imprime seu vetor novamente.

Podemos notar que o vetor do processo Pai continua desordenado, pois ao gerar um novo processo o processo Pai é duplicado, mas o processo Filho atua como um processo independente, ou seja, ele possui seu próprio vetor. O processo Filho organiza sua própria duplicata do vetor, enquanto, o vetor do processo Pai continua alheio à função ***sort***.

Código:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void sort(int* vet, int n) { //codigo de ordenacao por ordem crescente
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (vet[j] > vet[j+1]) {
                int temp = vet[j];
                vet[j] = vet[j+1];
                vet[j+1] = temp;
            }
        }
    }
}

int main() {

    //cria um vetor de 100 posicoes
    int vet[100];

    //adiciona valor a cada posicao do vetor
    for (int i = 0; i < 100; i++) {
        vet[i] = rand() % 100;
    }

    //exibe o vetor 100 antes da criacao do Filho
    printf("Valor do Pai antes do Filho ordenar: \n");
    for (int i = 0; i < 100; i++) {
```

```
        printf("%d ", vet[i]);
    }
    printf("\n\n");

    //chama comando fork para copiar processo Pai
    int pid = fork();

    if (pid == 0) {

        //Processo Filho ordena o vetor 100
        sort(vet, 100);

        printf("Valor do vetor ordenado pelo Filho depois do fork: \n");
        for (int i = 0; i < 100; i++) {
            printf("%d ", vet[i]);
        }
        printf("\n\n");

        //finalizacao do processo Filho
        _exit(0);
    } else if (pid > 0) {

        //processo Pai espera a finalizacao do processo Filho
        waitpid(NULL);
        //processo Pai imprime na tela o vetor 100
        printf("Valor do vetor do Pai depois do Filho ordenar: \n");
        for (int i = 0; i < 100; i++) {
            printf("%d ", vet[i]);
        }
        printf("\n");
    }
    //os processos Pai e Filho terao vetores diferentes, pois apos a criacao
do Filho
    //ele age de forma independente, possuindo seu proprio espaço de memoria.
    //ou seja, qualquer alteracao realizada no vetor do filho nao
influenciara no processo Pai
    return 0;
}
```

Resultado:

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/6$ ./seis
Valor do Pai antes do Filho ordenar:
83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 2
9 73 21 19 84 37 98 24 15 70 13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64 43 50 87
8 76 78 88 84 3 51 54 99 32 60 76 68 39 12 26 86 94 39

Valor do vetor ordenado pelo Filho depois do fork:
2 3 5 5 8 11 11 12 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 26 27 27 29 29 29 29 30 32 34 35 35 36 36 37 39 39 40
 42 43 45 46 49 50 51 54 56 56 57 58 59 60 62 62 62 63 64 67 67 67 67 68 68 69 70 70 72 73 73 76 76 77 78 80 81 82 82 83
 84 84 84 86 86 86 87 88 90 91 92 93 93 94 95 96 98 99

Valor do vetor do Pai depois do Filho ordenar:
83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 2
9 73 21 19 84 37 98 24 15 70 13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64 43 50 87
8 76 78 88 84 3 51 54 99 32 60 76 68 39 12 26 86 94 39
```

2.7. Threads X Processos

Objetivo: Threads versus Processos. Os limites máximos. Em ambiente linux os limites superiores da quantidade de processos e threads que podem ser disparados são configuráveis, testar e identificar.

O Programa deve criar processos e threads até que um deles dê erro, ou seja, atinja o limite. No código foi definido os limites de threads e processos como 1000000 e em um ***for*** os processos e threads são criados repetidamente até atingir seu limite. Logo depois, é imprimido na tela a quantidade de processos e threads criados antes de dar erro.

Código:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define LIMITE_PROCESSO 100000000
#define LIMITE_THREAD 100000000

void* funcao_thread(void* arg){
    pthread_exit(NULL);
}
```

```c
int main()
{

    //Processos

    printf("Teste do limite de criacao dos Processos: \n");

    int i, pid;

    for(i=1; i <= LIMITE_PROCESSO; i++){
        pid = fork();
        if (pid < 0){
            printf("Limte de Processos excedido: %d\n", i - 1);
            break;
        }
        if(pid == 0){
            exit(0);
        }
    }

    if(pid > 0){
        for(int j=0; j < i; j++){
            wait(NULL);
        }
        printf("Teste processos concluido\n");
    }

    //Threads

    printf("\nTeste do limite de criacao de Threads: \n");

    pthread_t thread;

    for(int i = 1; i <= LIMITE_THREAD; i++){
        if(pthread_create(&thread, NULL, funcao_thread, NULL) != 0){
            printf("Limite de Threads excedido: %d\n", i+1);
            break;
        }
    }
```

```
    if(i == LIMITE_THREAD + 1){
        printf("Teste threads concluido\n");
    }
    return 0;
}
```

Resultado:

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO/trabalho2/7$ ./sete
Teste do limite de criacao dos Processos:
Limte de Processos excedido: 15281

Teste do limite de criacao de Threads:
Limite de Threads excedido: 2
```

2.8. Desempenho de fork e threads

Objetivo: Crie um programa (ou altere um dos anteriores) de modo a lhe permitir comparar o desempenho para a realização da(s) tarefa(s) do programa através do uso de primitivas fork e através de threads (calcule a média de tempo de 20 simulações).

Para se comparar o tempo necessário para criar processos e threads utiliza-se a função clock( ), essa função retorna o tempo de processamento (que é dividido pelo valor da constante CLOCKS_PER_SEC para obter o tempo em segundos) usado pelo programa desde o início da execução. Foi definido o número de simulações igual a 20, então o programa irá entrar em um loop por meio da estrutura de repetição **for** até obter 20 simulações. Após ter obtido as 20 simulações do código, tanto de processos quanto de threads, será calculado e impresso o tempo médio por segundos das 20 simulações.

Código:

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

#define NUM_SIMULATIONS 20

void* funcaot(void* threadid) {
// uma thread vai rodar essa funcao
    wait(NULL);
}
```

```c
int main() {
    pid_t pid;
    int status;
    pthread_t thread;
    clock_t start_time, end_time;
    double total_time_fork = 0.0, total_time_thread = 0.0;

    // Simulacao dos Forks
    for (int i = 0; i < NUM_SIMULATIONS; i++) {
        start_time = clock();
        pid = fork();
        if (pid == 0) {
            // Processo filho
            exit(0);
        } else if (pid > 0) {
            // Processo pai
            wait(&status);
            end_time = clock();
            total_time_fork += ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
        } else {
            printf("Erro ao criar o processo filho\n");
            exit(1);
        }
    }
```

```
43
44      // Simulacao das Threads
45
46      for (int i = 0; i < NUM_SIMULATIONS; i++) {
47          start_time = clock();
48          pthread_create(&thread, NULL, funcaot, NULL);
49          pthread_join(thread, NULL);
50          end_time = clock();
51          total_time_thread += ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
52  }
53
54      printf("Tempo medio por Processos: %f por segundos\n", total_time_fork / NUM_SIMULATIONS);
55      printf("Tempo medio por Threads: %f por segundos\n", total_time_thread / NUM_SIMULATIONS);
56
57      return 0;
58  }
59
```

Resultado:

```
amand@AmanLopes:~/amand/ufpa/3semestre/SO$ ./t2
Tempo médio por Processos: 0.000412 por segundos
Tempo médio por Threads: 0.000118 por segundos
```

3. Escalonadores

Objetivo: Aplicar os escalonadores Shortest Job First (SJF), Shortest Remaining Time First (SRTF), Prioridade e Round Robin (RR) no código fornecido para a atividade. Aumentar o número de processos do código original para 30 e comparar os escalonadores.

3.1 Aumentando o número de processos

Para aumentar a quantidade de processos, utilizamos a biblioteca *time.h*, para gerar números pseudoaleatórios com base na hora atual do sistema. O objetivo é gerar 30 processos, com tempo de duração de 5, 8 ou 12. Isso foi implementado usando o procedimento *gera_processos()*, mostrado a seguir:

```
void gera_processos(int proc[], int burst_time[], int n) {
    // Inicializa o gerador de números aleatórios com o tempo atual
    srand(time(NULL));
   int times[] = {5, 8, 12};
```

```
    for (int i = 0; i < n; i++) {
        proc[i] = i + 1;
        burst_time[i] = times[rand() % 3];  // Burst time 5,8 ou 12
        arrival_time[i] = i*3;    //Gera o arrival time de cada processo
}
}
```

Além das arrays que já estavam definidas no código, também foi acrescentado a informação do tempo de chegada(arrival time) de cada processo a CPU, para também ser considerado no escalonamento.

As arrays passadas para o procedimento são declaradas dentro da função *main()*, além disso, para o escalonador de prioridades foi adicionado também a geração da prioridade de cada processo dentro do laço *for()*.

```
priority[i] = rand() % 5 + 1;    // Prioridade entre 1 e 5
```

Resultado:

```
File  Edit  View  Search  Terminal  Help
felps@felps-VirtualBox:~/Trabalho4$ ./sched
Processes  Burst  Arrival Waiting Turn around
 1        12             0      0       12
 2        5              3      9       14
 3        12             6      11      23
 4        12             9      20      32
 5        5              12     29      34
 6        8              15     31      39
 7        8              18     36      44
 8        12             21     41      53
 9        12             24     50      62
 10       5              27     59      64
 11       8              30     61      69
 12       12             33     66      78
 13       5              36     75      80
 14       12             39     77      89
 15       5              42     86      91
 16       12             45     88      100
 17       8              48     97      105
 18       8              51     102     110
 19       8              54     107     115
 20       8              57     112     120
 21       5              60     117     122
 22       12             63     119     131
 23       8              66     128     136
 24       5              69     133     138
 25       5              72     135     140
 26       8              75     137     145
 27       8              78     142     150
 28       12             81     147     159
 29       5              84     156     161
 30       8              87     158     166
Average waiting time = 84.300003
Average turn around time = 92.733330
felps@felps-VirtualBox:~/Trabalho4$
```

3.2 SJF

Neste escalonador, a prioridade é dada aos processos de menor duração, o que significa que os processos com o tempo de execução mais curto são tratados com preferência. Para implementar essa lógica no código, utilizamos como base algoritmo de Selection Sort dentro da função *avgtime()* para organizar os processos com base em sua duração, classificando os processos mais curtos antes dos mais longos.

A verificação *arrival_time* <= *current_time* foi adicionada ao ordenador para determinar se o processo em questão já está pronto para a execução, caso contrário ele não deve ser escalonado. Para realizar essa verificação, a array *arrival_time* foi incorporada à função principal (*main*) para fornecer informações sobre o tempo de chegada de cada processo. Essa adição permite que o escalonador leve em consideração não apenas a duração dos processos, mas também seu tempo de chegada, tornando o escalonamento mais preciso e adequado para cenários nos quais os processos chegam em momentos diferentes.

A inicialização de *minId* e *min_burst* como -1 e INT_MAX respectivamente serve para garantir que o processo escalonado esteja realmente pronto, caso o algoritmo não encontre nenhum processo pronto ele incrementa a variável *current_time*, para avançar no tempo até encontrar algum processo pronto, e decrementa a variável *i* para continuar a busca sempre iniciando pelo mesmo processo, evitando que esse processo seja ignorado.

A função *avgtime()* recebe os processos, seu tempo de execução e o tempo de chegada, faz o escalonamento e, por fim, calcula o tempo médio de resposta e de espera.

Código:

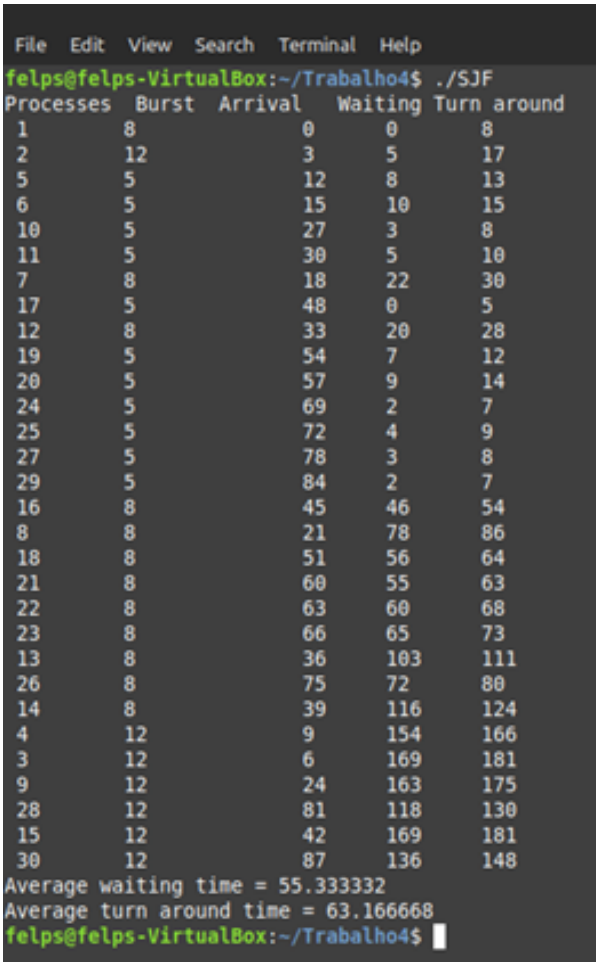
```
#include <stdio.h>
#include <time.h>

void gera_processos(int proc[], int burst_time[], int
arrival_time[], int n) {
    // Inicializa o gerador de números aleatórios com o tempo
atual
```

```
    srand(time(NULL));
   int times[] = {5, 8, 12};
    for (int i = 0; i < n; i++) {
        proc[i] = i + 1;
          burst_time[i] = times[rand() % 3];  // Burst time 5,8 ou
12
          arrival_time[i] = i*3;    //gera o arrrival time de cada
processo como multiplo de 3
    }
}
// Function to find the waiting time for all processes
int waitingtime(int proc[], int n,
int burst_time[], int wait_time[], int arrival_time[]) {
   // waiting time for first process is 0
   wait_time[0] = 0;
   // calculating waiting time
   for (int i = 1; i < n ; i++)
      wait_time[i] = burst_time[i-1] + wait_time[i-1];
    for (int i = 1; i < n ; i++)//ajustando o wait time conforme o
arrival
      wait_time[i] = wait_time[i] - arrival_time[i];
   return 0;
}
// Function to calculate turn around time
int turnaroundtime( int proc[], int n,
int burst_time[], int wait_time[], int tat[]) {
   // calculating turnaround time by adding
   // burst_time[i] + wait_time[i]
   int i;
   for ( i = 0; i < n ; i++)
   tat[i] = burst_time[i] + wait_time[i];
   return 0;
}
//Function to calculate average time
int   avgtime(   int   proc[],   int   n,   int   burst_time[],   int
arrival_time[]) {
   int wait_time[n], tat[n], total_wt = 0, total_tat = 0;
   int i,j,minId, min_burst;
   int current_time = 0;
   //ordenando os processos de acordo com o burstTime
   for(i = 0; i < n; i++) {
      minId = -1;
      min_burst = __INT_MAX__;
```

```
      for(j = i; j < n; j++) {
             if(burst_time[j] < min_burst && arrival_time[j] <=
current_time){
            minId = j;
       min_burst = burst_time[j];
         }
      }
   //troca os burstTime
   if(minId == -1){
   current_time++;
   i--;
   }else{if(i != minId){
       int temp = burst_time[i];
       burst_time[i] = burst_time[minId];
       burst_time[minId] = temp;
       // troca os proc
       temp = proc[i];
       proc[i] = proc[minId];
       proc[minId] = temp;
       // arrivaltime
       temp = arrival_time[i];
       arrival_time[i] = arrival_time[minId];
       arrival_time[minId] = temp;
        }
      current_time += burst_time[i];
   }
}
   //Function to find waiting time of all processes
   waitingtime(proc, n, burst_time, wait_time, arrival_time);
   //Function to find turn around time for all processes
   turnaroundtime(proc, n, burst_time, wait_time, tat);
   //Display processes along with all details
   printf("Processes  Burst  Arrival   Waiting Turn around \n");
   // Calculate total waiting time and total turn
   // around time
   for ( i=0; i<n; i++) {
      total_wt = total_wt + wait_time[i];
      total_tat = total_tat + tat[i];
             printf("  %d\t    %d\t\t  %d\t%d  \t%d\n", proc[i],
burst_time[i],arrival_time[i] ,wait_time[i], tat[i]);
   }
      printf("Average  waiting  time  =  %f\n",  (float)total_wt  /
(float)n);
```

```
    printf("Average turn around time = %f\n", (float)total_tat /
(float)n);

}
// main function
int main() {
   int proc[30];
   int n = sizeof(proc) / sizeof(proc[0]);
   int burst_time[30];
   int arrival_time[30];
   gera_processos(proc, burst_time, arrival_time, n);
   avgtime(proc, n, burst_time, arrival_time);
   return 0;
}
```

Resultado:



```
File  Edit  View  Search  Terminal  Help
felps@felps-VirtualBox:~/Trabalho4$ ./SJF
Processes  Burst  Arrival   Waiting Turn around
 1         8              0       0       8
 2         12             3       5       17
 5         5              12      8       13
 6         5              15      10      15
 10        5              27      3       8
 11        5              30      5       10
 7         8              18      22      30
 17        5              48      0       5
 12        8              33      20      28
 19        5              54      7       12
 20        5              57      9       14
 24        5              69      2       7
 25        5              72      4       9
 27        5              78      3       8
 29        5              84      2       7
 16        8              45      46      54
 8         8              21      78      86
 18        8              51      56      64
 21        8              60      55      63
 22        8              63      60      68
 23        8              66      65      73
 13        8              36      103     111
 26        8              75      72      80
 14        8              39      116     124
 4         12             9       154     166
 3         12             6       169     181
 9         12             24      163     175
 28        12             81      118     130
 15        12             42      169     181
 30        12             87      136     148
Average waiting time = 55.333332
Average turn around time = 63.166668
felps@felps-VirtualBox:~/Trabalho4$
```

3.3 SRFT

Neste escalonador, a prioridade é dada aos processos com o menor tempo restante de execução. A implementação dessa lógica foi realizada por meio da função *findMenorProc()* e do procedimento *SRTF()*.

A função *findMenorProc()* desempenha a função de localizar o processo disponível com o menor tempo restante de execução, também considerando o *arrival_time* para não selecionar um processo que não esteja pronto para a execução.

O procedimento *SRTF()* é responsável por implementar o escalonador, através um loop, em cada iteração decide qual processo deve ser executado no momento. Ele aproveita a função *findMenorProc()* para determinar a ordem de escalonamento dos processos, garantindo que aqueles com menor tempo restante de execução sejam executados primeiro. Caso um processo com menor tempo chegue para ser executado, o processo atual será parado para a execução do novo processo.

Código:

```
#include <stdio.h>
#include <time.h>

void gera_processos(int proc[], int burst_time[], int arrival_time[], int n)
{
    //inicia o gerador de numeros psedoaleatorios
    srand(time(NULL));
    int times[] = {5, 8, 12};
    for (int i = 0; i < n; i++) {
        proc[i] = i + 1;
        burst_time[i] = times[rand() % 3];  // Burst time 5,8 ou 12
         arrival_time[i] = i*3;    //gera o arrival time de cada processo com
multiplos de 3
    }
}
// Function to find the process with the shortest remaining time
int  findMenorProc(int  arrival_time[],  int  n,  int  remaining_time[],  int
current_time) {
    int min_time = __INT_MAX__;
```

```
    int menorProc = -1;
    for (int i = 0; i < n; i++) {
            if (remaining_time[i] > 0 && remaining_time[i] < min_time &&
arrival_time[i] <= current_time) {
            min_time = remaining_time[i];
            menorProc = i;
        }
    }
    return menorProc;
}
// Função do escalonador
int SRTF(int proc[], int n, int burst_time[], int arrival_time[]) {
    int wait_time[n], turnaround_time[n], remaining_time[n];
    int total_wait_time = 0, total_turnaround_time = 0;
    int current_time = 0;
    int completed = 0;
    // Inicia remainning_time array
    for (int i = 0; i < n; i++) {
        remaining_time[i] = burst_time[i];
    }
   int menorProc;
    while (completed < n) {
             menorProc = findMenorProc(arrival_time, n, remaining_time,
current_time);
        if (menorProc == -1) {//se nao encontrar nenhum processo
            current_time++;
        } else { // se tiver algum processo para execuçao
           remaining_time[menorProc]--;
               if (remaining_time[menorProc] == 0) { //caso o processo seja
terminado
             completed++;
             int finish_time = current_time + 1;
                                turnaround_time[menorProc] = finish_time
-arrival_time[menorProc];
                        wait_time[menorProc] = turnaround_time[menorProc] -
burst_time[menorProc];
               total_wait_time += wait_time[menorProc];
               total_turnaround_time += turnaround_time[menorProc];
            }
            current_time++;
        }
```

```
    }
    printf("Processes  Burst   Arrival   Waiting   Turnaround\n");
    for (int i = 0; i < n; i++) {
         printf("    %d\t     %d\t\t%d\t  %d\t\t%d\n", proc[i], burst_time[i],
arrival_time[i], wait_time[i], turnaround_time[i]);
    }
    printf("Average waiting time = %.2f\n", (float)total_wait_time / n);
    printf("Average turnaround time = %.2f\n", (float)total_turnaround_time /
n);
    return 0;
}
int main() {
    int proc[30]; //array para os ids dos processos
    int n = sizeof(proc) / sizeof(proc[0]);
    int burst_time[30]; // array para o brust dos 30 processos
    int arrival_time[30]; // array para o arrival de cada processo
    gera_processos(proc, burst_time, arrival_time, n);
    SRTF(proc, n, burst_time, arrival_time);
    return 0;
}
```

Resultado:

```
File  Edit  View  Search  Terminal  Help
felps@felps-VirtualBox:~/Trabalho4$ ./SRTF
Processes  Burst   Arrival   Waiting   Turnaround
   1        8          0         0          8
   2        5          3         5          10
   3        8          6         7          15
   4        8          9         12         20
   5        8          12        42         50
   6        8          15        47         55
   7        12         18        109        121
   8        12         21        118        130
   9        5          24        5          10
   10       5          27        7          12
   11       12         30        121        133
   12       12         33        130        142
   13       5          36        3          8
   14       5          39        5          10
   15       5          42        7          12
   16       8          45        50         58
   17       12         48        127        139
   18       8          51        52         60
   19       8          54        57         65
   20       12         57        130        142
   21       12         60        139        151
   22       8          63        56         64
   23       5          66        4          9
   24       5          69        6          11
   25       5          72        8          13
   26       12         75        136        148
   27       12         78        145        157
   28       12         81        154        166
   29       5          84        1          6
   30       5          87        3          8
Average waiting time = 56.20
Average turnaround time = 64.43
felps@felps-VirtualBox:~/Trabalho4$
```

3.4 Prioridades fixas

O escalonador baseado em prioridades é um mecanismo que leva em consideração prioridades associadas a cada processo. Nesse modelo, os processos com um valor de prioridade mais alto são agendados para serem executados primeiro, com o valor variando de 1 a 5, e não é preemptivo, ou seja, os processos não são interrompidos se já estiverem em execução.

A implementação desse escalonador ocorre dentro da função *avgtime()*, na qual é utilizado o algoritmo de Selection Sort como base para ordenar os processos com base em suas prioridades, e o teste de *arrival_time* <= *current_time* para garantir que o processo está pronto para execução, evitando que processos não prontos sejam escalonados.

Caso nenhum processo possa ser escalonado, o tempo atual é incrementado e o valor de i é decrementado para buscar novamente a partir do mesmo processo, evitando que qualquer um deles seja ignorado.

Código:

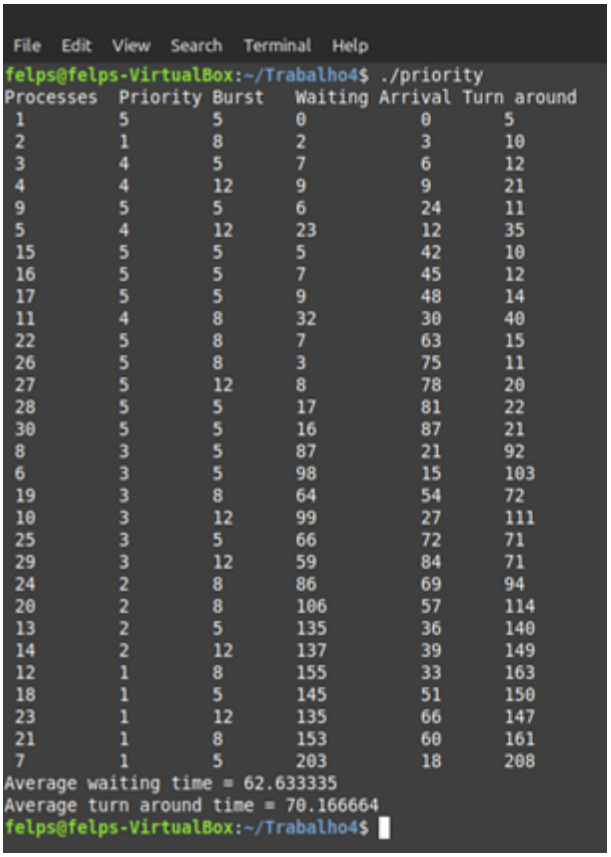
```
#include <stdio.h>
#include <time.h>
// Function to find the waiting time for all processes
void gera_processos(int proc[], int burst_time[], int arrival_time[], int
priority[], int n) {
    // Inicializa o gerador de números aleatórios com o tempo atual
    srand(time(NULL));
   int times[] = {5, 8, 12};
    for (int i = 0; i < n; i++) {
        proc[i] = i + 1;
        burst_time[i] = times[rand() % 3];  // Burst time 5,8 ou 12
         arrival_time[i] = i*3;   //Gera o arrival time de cada processo como
multiplo de 3
        priority[i] = rand() % 5 + 1;   // Prioridade entre 1 e 5
    }
}
int waitingtime(int proc[], int n,
int burst_time[], int wait_time[], int arrival_time[]) {
   // waiting time for first process is 0
   wait_time[0] = 0;
```

```
   // calculating waiting time
   for (int i = 1; i < n ; i++ )
      wait_time[i] = burst_time[i-1] + wait_time[i-1];
   for (int i = 1; i < n ; i++ )//ajusta o wait com base no arrival
      wait_time[i] = wait_time[i] - arrival_time[i];
   return 0;
}
// Function to calculate turn around time
int turnaroundtime( int proc[], int n,
int burst_time[], int wait_time[], int tat[]) {
   // calculating turnaround time by adding
   // burst_time[i] + wait_time[i]
   int i;
   for ( i = 0; i < n ; i++)
   tat[i] = burst_time[i] + wait_time[i];
   return 0;
}
//Function to calculate average time
int  avgtime(  int  proc[],  int  n,  int  burst_time[],int  priority[],  int
arrival_time[]) {
   int wait_time[n], tat[n], total_wt = 0, total_tat = 0;
   int i, j, idMax, current_time = 0;
   int maior_p = 0;
   //ordenação dos processos
   for (i = 0; i < n; i++) {
    idMax = -1;
    maior_p = 0;
    for (j = i; j < n; j++) {
        if (arrival_time[j] <= current_time && priority[j] > maior_p  ){
            idMax = j;
            maior_p = priority[j];
        }
    }
      if (idMax == -1) // se nenhum processo estiver pronto
         {current_time++;
         i--;}
      else{if (idMax != i) {//faz as trocas
            int temp;
            // Trocar os IDs dos processos
            temp = proc[i];
            proc[i] = proc[idMax];
```

```
            proc[idMax] = temp;
            // Trocar as prioridades
            temp = priority[i];
            priority[i] = priority[idMax];
            priority[idMax] = temp;
            // Trocar os burst times
            temp = burst_time[i];
            burst_time[i] = burst_time[idMax];
            burst_time[idMax] = temp;
            // Trocar os tempos de chegada
            temp = arrival_time[i];
            arrival_time[i] = arrival_time[idMax];
            arrival_time[idMax] = temp;
         }
   // Atualizar o current_time apos escalonar um processo
   current_time += burst_time[i];
   }
}
   waitingtime(proc, n, burst_time, wait_time, arrival_time);
   //Function to find turn around time for all processes
   turnaroundtime(proc, n, burst_time, wait_time, tat);
   //Display processes along with all details
   printf("Processes  Priority Burst   Waiting Arrival Turn around \n");
   // Calculate total waiting time and total turn
   // around time
   for ( i=0; i<n; i++) {
      total_wt = total_wt + wait_time[i];
      total_tat = total_tat + tat[i];
        printf(" %d\t    %d\t     %d\t     %d     \t%d     \t%d\n", proc[i],
priority[i], burst_time[i], wait_time[i], arrival_time[i], tat[i]);
   }
   printf("Average waiting time = %f\n", (float)total_wt / (float)n);
   printf("Average turn around time = %f\n", (float)total_tat / (float)n);
return 0;
}
// main function
int main() {
   //process id's
   int proc[30];
   int n = sizeof(proc) / sizeof(proc[0]);
   int burst_time[30];
```

```
    int arrival_time[30];
    int priority[30];
    gera_processos(proc, burst_time, arrival_time, priority, n);
    //Burst time of all processes
    avgtime(proc, n, burst_time, priority, arrival_time);
    return 0;
}
```

Resultado:



3.5 RR

Esse método de escalonamento atribui a cada processo um quantum (intervalo de tempo) seguindo uma fila circular, garantindo que todos os processos tenham acesso a CPU.

O escalonador foi aplicado dentro da função *waitingtime()* usando um loop que a cada iteração percorre a fila de processos dando o quantum de cada processo, se esse processo estiver pronto para a execução, o loop é finalizado

quando todos os processos forem concluídos. Nesse caso o quantum foi definido como 2, a execução termina quando todos os processos estiverem concluídos, ou seja, quando o remaining time de todos for 0.

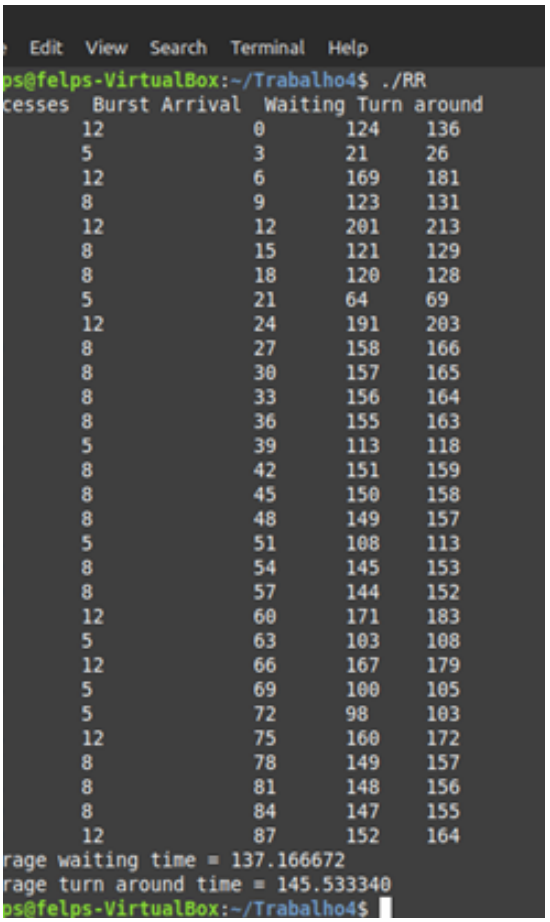
Código:

```
#include <stdio.h>
#include <time.h>

#define QUANTUM 2

void gera_processos(int proc[], int burst_time[],int arrival_time[], int n) {
   // Inicializa o gerador de números aleatórios com o tempo atual
   srand(time(NULL));
   int times[] = {5, 8, 12};
   for (int i = 0; i < n; i++) {
      proc[i] = i + 1;
      burst_time[i] = times[rand() % 3];  // Burst time 5,8 ou 12
      arrival_time[i] = i*3; // Gera arrival time com multiplos de 3
   }
}
// Function to find the waiting time for all processes
int waitingtime(int proc[], int n,
int burst_time[], int wait_time[], int arrival_time[]) {
   // Faça uma cópia do burst_time para modificar
   int remaining_time[n];
   for (int i = 0; i < n; i++)
      remaining_time[i] = burst_time[i];
   int current_time = 0, done;
   do{
      done = 1; // Suponha que todos os processos sejam concluídos
      // Atravesse todos os processos
      for (int i = 0; i < n; i++) {
         // Se o processo ainda não estiver concluído
        if (remaining_time[i] > 0) {
          done = 0; // Marcar como não concluído
            // Se o tempo restante for menor ou igual ao quantum
          if(arrival_time[i] <= current_time){
            if (remaining_time[i] <= QUANTUM) {
              current_time += remaining_time[i]; //finaliza o processo
```

```
                wait_time[i] = current_time-burst_time[i]-arrival_time[i] ; //
Calcular tempo de espera
                remaining_time[i] = 0; // Marcar processo como concluído
              } else {
                current_time += QUANTUM; //Adicionar quantum ao tempo atual
                  remaining_time[i] -= QUANTUM; // Subtraia o quantum do tempo
restante
               }
             }
          }
       }
     }while(done == 0);
   return 0;
}
// Function to calculate turn around time
int turnaroundtime( int proc[], int n,
int burst_time[], int wait_time[], int tat[]) {
   // calculating turnaround time by adding
   // burst_time[i] + wait_time[i]
   int i;
   for ( i = 0; i < n ; i++)
      tat[i] = burst_time[i] + wait_time[i];
   return 0;
}
//Function to calculate average time
int avgtime( int proc[], int n, int burst_time[], int arrival_time[]) {
   int wait_time[n], tat[n], total_wt = 0, total_tat = 0;
   int i;
   //Function to find waiting time of all processes
   waitingtime(proc, n, burst_time, wait_time, arrival_time);
   //Function to find turn around time for all processes
   turnaroundtime(proc, n, burst_time, wait_time, tat);
   //Display processes along with all details
   printf("Processes  Burst Arrival  Waiting Turn around \n");
   // Calculate total waiting time and total turn
   // around time
   for ( i=0; i<n; i++) {
      total_wt = total_wt + wait_time[i];
      total_tat = total_tat + tat[i];
            printf("  %d\t    %d\t\t  %d\t  %d  \t%d\n",  i+1,  burst_time[i],
arrival_time[i], wait_time[i], tat[i]);
```

```
    }
    printf("Average waiting time = %f\n", (float)total_wt / (float)n);
    printf("Average turn around time = %f\n", (float)total_tat / (float)n);
    return 0;
}
// main function
int main() {
    //process id's
    int proc[30];
    int n = sizeof(proc) / sizeof(proc[0]);
    int burst_time[30];
    int arrival_time[30];
    gera_processos(proc, burst_time,arrival_time, n);
    avgtime(proc, n, burst_time, arrival_time);
    return 0;
}
```

Resultado:



3.6 Comparação dos escalonadores

A escolha do algoritmo de escalonamento depende das características desejadas para o sistema. Baseado nos testes que fizemos os algoritmos SJF e SRTF tem os menores tempos médios de espera e resposta, já que priorizam os terminar os processos mais rapidamente, ficando abaixo de 60 u.t. de espera média. Já o algoritmo baseado em prioridades teve médias um pouco mais elevadas que os dois primeiros, porém é os processos de maiores prioridades foram executados quase que imediatamente. Já o RR tem a média de espera e de resposta mais altos, porém todos os processos tiverem acesso a CPU igualmente. A seguir uma tabela com as médias de cada escalonador.

	Tempo médio de espera	Tempo médio de resposta
SJF	55.3 u.t.	63.1 u.t.
SRTF	56.2 u.t.	64.4 u.t.
Prior. fixa	62.6 u.t.	70.1 u.t.
RR	137.1 u.t.	145.5 u.t.

Baseado nisso, é possível fazer a análise de suas vantagens e desvantagens, avaliando em que sistemas podem ser melhor aplicados.

1)SJF (Shortest Job First)

a) Vantagens:
   i) Tempo médio de espera é menor em comparação com os outros algoritmos.
   ii) Processos menores são favorecidos

b) Desvantagens:

   i) É necessário saber quanto tempo cada processo precisa, o que pode não estar disponível.
   ii) Se processos curtos continuarem chegando, os processos longos não serão executados(inanição).

c) Ideal em sistemas de batch(lotes), onde os processos geralmente têm tempos de execução conhecidos.

2)SRTF (Shortest Remaining Time First)

a) Vantagens:
   i) Tempo médio de espera é curto.
   ii) Tempo de resposta mais rápido para processos menores.

b) Desvantagens:
   i) É necessário a informação de quanto tempo de execução resta para cada processo.
   ii) Processos longos podem sofrem com inanição.
   iii) Pode causar sobrecarga nas trocas de contexto, pois os processos são interrompidos para dar preferência aos processos mais curtos

c) Útil em sistemas de tempo compartilhado.

3)Prioridade(fixa)

a) Vantagens:
   i) Garante que processos críticos(com alta prioridade) sejam executados imediatamente.

b) Desvantagens:
   i) Pode gerar inanição nos processos de baixa prioridade se os processos de alta prioridade forem longos e/ou muitos.
   ii) Requer a atribuição de prioridades de forma equilibrada.

c) Utilizado em sistemas de tempo real, onde certas tarefas precisam ser executadas de imediato.

4)RR(Round Robin)

a) Vantagens:
   i) Todos os processos têm acesso igual a CPU.
   ii) Não há inanição.

b) Desvantagens:
   i) Tempo médio de espera geralmente é alto.
   ii) Se o Quantum de tempo for muito pequeno, pode gerar sobrecarga nas trocas de contexto.
   iii) Tempo de resposta de processos longos é grande.

c) Utilizado em sistemas de tempo real e/ou ambientes multitarefas.

4. Semáforo

4.1. O Barbeiro Dorminhoco

Objetivo: Gerir a organização do barbeiro e seus clientes, alocando o barbeiro para dormir na cadeira quando não há clientes e alocando a cadeira a um cliente quando necessário, acordando assim o barbeiro dorminhoco e criando uma fila de espera limitada para o atendimento quando os demais clientes chegarem.

Programa Explicado:

Neste código usamos threads e semáforos para gerir o tráfego de clientes na barbearia. Usamos 4 semáforos para controlar o acesso dos recursos.

```
 7  #define NUM_CADEIRAS 20
 8
 9  sem_t clientes; // Semáforo para controlar os clientes na fila de espera
10  sem_t cadeira; // Semáforo para controlar a cadeira do barbeiro
11  sem_t travesseiro; // Semáforo para acordar o barbeiro
12  sem_t seatBelt; // Semáforo para sinalizar o término do corte de cabelo
13
```

Logo em seguida, definimos o número de clientes e iniciamos a contagem da fila em 0.

Seguido disso, adicionamos uma struct Queue para representar a fila de espera, uma função Enqueue para gerir a entrada de clientes na fila de espera e uma função Dequeue para gerir a saída de clientes da fila de espera.

```
19  struct Queue {
20      int data[NUM_CADEIRAS];
21      int front, rear;
22  } clienteQueue; // Estrutura de dados para representar a fila de espera dos clientes
23
24  void enqueue(int value) {
25      clienteQueue.rear = (clienteQueue.rear + 1) % NUM_CADEIRAS;
26      clienteQueue.data[clienteQueue.rear] = value;
27  }
28
29  int dequeue() {
30      int value = clienteQueue.data[clienteQueue.front];
31      clienteQueue.front = (clienteQueue.front + 1) % NUM_CADEIRAS;
32      return value;
33  }
```

Abre-se uma função denominada “barbeiro” para controlar o comportamento do barbeiro, através do uso dos semáforos que foram definidos no início do código, e outra função denominada “cliente” para controlar o comportamento dos clientes.

```c
void *cliente(void *arg) {
    int id = *((int *)arg);

    // Tenta entrar na fila de espera
    pthread_mutex_lock(&mutex);
    if (clientesEsperando < NUM_CADEIRAS) {
        clientesEsperando++;
        printf("Cliente %d chegou.\n", id);
        enqueue(id);
        pthread_mutex_unlock(&mutex);

        // Aguarda a vez na fila de espera
        sem_wait(&cadeira);
        printf("Cliente %d é o próximo da fila. \n", id);
        sleep(1);

        // Cliente acordou o barbeiro e está cortando o cabelo
        printf("Cliente %d acordou o barbeiro e está cortando o cabelo.\n", id);
        sleep(2);

        // Cliente terminou o corte de cabelo
        printf("Cliente %d terminou o corte de cabelo e foi embora.\n", id);
        sem_post(&clientes); // Libera o barbeiro
        sem_post(&seatBelt); // Sinaliza o término do corte
    } else {
        // A fila de espera está cheia, o cliente vai embora insatisfeito
        pthread_mutex_unlock(&mutex);
        printf("Cliente %d viu o tamanho da fila e foi embora.\n", id);
    }
}
```

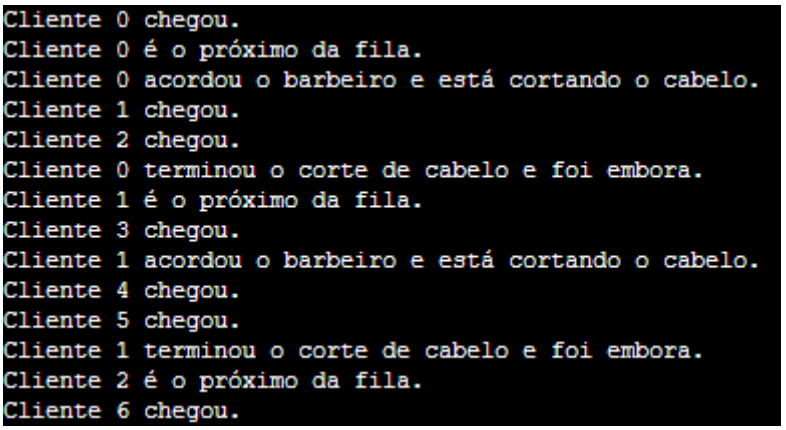
E a partir desse momento, abrimos a função “main” para dar início no processo dos semáforos e estruturas de dados, além de criarmos as Threads do barbeiro e do clientes para simular o atendimento e chegada/saída.

```c
    for (int i = 0; i < numClientes; i++) {
        clientesArgs[i] = i;
        pthread_create(&clienteThreads[i], NULL, cliente, &clientesArgs[i]);
        sleep(1); // Cria clientes com um pequeno atraso
    }

    for (int i = 0; i < numClientes; i++) {
        pthread_join(clienteThreads[i], NULL);
    }

    // Aguarda o barbeiro terminar
    pthread_cancel(barbeiroThread);
    pthread_join(barbeiroThread, NULL);
    printf("encerrou o horário de atendimento e o barbeiro dormiu na cadeira. \n");

    return 0;
}
```

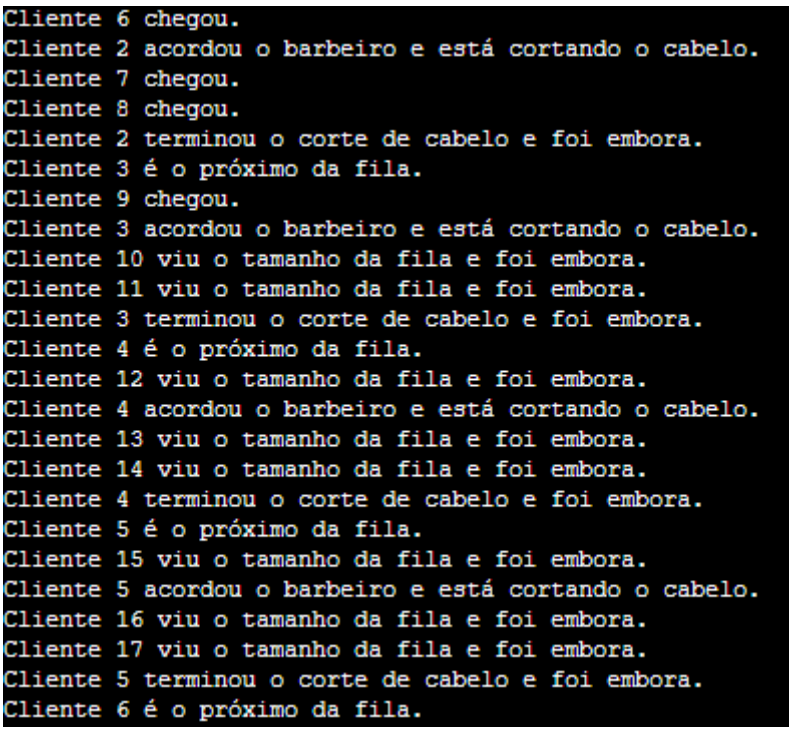
Em resumo, o programa implementa o problema do "Barbeiro Dorminhoco", onde múltiplos clientes aguardam na fila para serem atendidos por um único barbeiro. A

utilização de semáforos e mutexes garante que o comportamento das threads seja coordenado de forma segura, respeitando as regras do problema.

Capturas de tela do emulador rodando o programa:



Nas primeiras linhas exibidas, não ocorre congestionamento.



Aqui a maioria dos novos clientes é dispensada devido o número de saída (liberação de cadeiras) e menor que a vazão de entrada.