Sistemas Operativos

Trabalho Prático

Universidade do Minho

LCC 23/24

Elaborado por:

- a97554 Marco Silva
- a95917 Eduardo Soares
- a100113 David Gonçalves

INTRODUÇÃO

O enunciado do trabalho proposto descreve a implementação de um sistema de coordenação de tarefas. Este sistema consiste num servidor, o *orchestrator*, que recebe solicitações de tarefas de clientes e coordena a sua execução com base em políticas de escalonamento.

IMPLEMENTAÇÃO

Decidimos que a forma de comunicação entre os dois processos, por meio de *FIFOs* (First In, First Out), seria uma *struct* (apelidada de minfo).

```c
typedef struct minfo {
    int id;                     // Identificador único atribuído a cada mensagem.
    int tipo;                   // Tipo de mensagem: 0 para mensagem normal, 1 para mensagem de filho do servidor.
    int operacao;               // Tipo de operação: 0 para -p, 1 para -u, 2 para status.
    int time;                   // Tempo estimado de execução da tarefa em milissegundos.
    int pid;                    // ID do processo associado à mensagem para comunicação entre servidor e clientes.
    int custom;                 // Campo para implementação de uma política de escalonamento personalizada.
    struct timeval start;       // Tempo de início da execução da tarefa.
    struct timeval end;         // Tempo de término da execução da tarefa.
    char execucao[sizeExecute]; // Saída da execução dos comandos associados à tarefa.
    char nome[sizeExecute];     // Comando a ser executado (completo ou encadeado, dependendo da operação).
} *minfo;
```

Criámos o FIFO do cliente para servidor dizer que recebeu a tarefa.

```c
//Cria o FIFO do cliente para depois ler
char fifoc_name[30];
sprintf(fifoc_name, CLIENT "%d", mensagem->pid);

if(mkfifo(fifoc_name, 0666)==-1){
    perror("Erro ao criar o fifo");
    return -1;
}
```

```c
//Escreve a mensagem no FIFO do servidor
write(fifoserver_fd,mensagem, sizeof(struct minfo));
close(fifoserver_fd);
```

```c
//Abre o FIFO do cliente para leitura
int fifocliente_fd = open(fifoc_name, O_RDONLY, 0666);
```

Se o cliente mandou uma tarefa vai receber um aviso de esta ter sido recebida, caso peça um status vai receber o status.

```c
//Lê e processa a resposta do servidor
if(mensagem->operacao != 2){
    read(fifocliente_fd, mensagem, sizeof(struct minfo));
    char output[64];
    sprintf(output, "Task (id %d, pid %d) recieved\n", mensagem->id, mensagem->pid);

    write(1,output, strlen(output));

}else if(mensagem->operacao == 2){
    char statusoutput[4097];
    read(fifocliente_fd, &statusoutput, 4096);
    strcat(statusoutput, "\n");
    write(1,statusoutput, strlen(statusoutput));
}

unlink(fifoc_name);
return 0;
```

```c
// Inicializa variáveis
int tempotestes = 0;     // Tempo de execução total
int countID = 1001;      // id das mensagens
int countPT = 0;         // Quantos processos estão a decorrer neste momento
int countfila = 0;       // Quantos processos estão na fila
minfo filaEspera[10*N];
minfo mensagem = malloc(sizeof(struct minfo));
```

Quando uma mensagem é recebida, verifica se é uma tarefa já executada pronta para ser escrita no arquivo de output. Se sim, cria um processo filho para lidar com a escrita no arquivo. Se a mensagem indica o encerramento do servidor, o programa é encerrado.

```c
while(1){
    printf("começa o read\n");
    int read_bytes = read(fifoserver_fd, mensagem, sizeof(struct minfo));
    if(mensagem->pid == 0){
        unlink(SERVER);
        return 0;
    }
    // Processa a mensagem recebida
    printf("acaba o read\n");
    printf("quantos bytes %d\n", read_bytes);
    if(mensagem->tipo == 1){ // Se for uma tarefa a ser escrita no output
        countPT--;
        int pid = fork();
        if(pid==0){
            int fdoutput = open(OUTPUT, O_CREAT|O_APPEND|O_WRONLY, 0777);
            char *realoutput;
            realoutput = escritanooutput(mensagem);
            write(fdoutput,realoutput,strlen(realoutput));
            close(fdoutput);
            _exit(0);
        }
```

Se houver tarefas na fila e o número máximo de tarefas em execução ainda não foi atingido, o programa inicia um novo processo para lidar com a próxima tarefa. Ele seleciona a próxima tarefa da fila com base na política de escalonamento definida. De seguida, reduz o contador de tarefas na fila e aumenta o contador de tarefas em execução. Após isso, executa a tarefa de acordo com sua operação e atualiza os registos de tempo antes de enviar os resultados de volta ao servidor.

```
if((countfila>0) && (countPT<N)){ // Manda fazer um novo processo se tiver processos para fazer
    int pf=0;
    if(0 == strcmp(sp, "SJF")){
        pf = sc_SJF(&filaEspera, countfila);
    }
    if(0 == strcmp(sp, "CUSTOM")){
        pf = sc_CUSTOM(&filaEspera, countfila);
    }
    mensagem = filaEspera[pf];
    countfila--;
    countPT++;
    gettimeofday(&(mensagem->start), NULL);
    if(mensagem->operacao == 1){ // -u é 1 comando só
        int pid = fork();
        if(pid==0){
            int pipes[2];
            pipe(pipes);
            int pidfilho = fork();
            if(pidfilho==0){
                close(pipes[0]);
                dup2(pipes[1],1);
                close(pipes[1]);
                char *comando;
                comando = strdup(mensagem->nome);
                char *nome = strsep(&comando, " ");
                execlp(nome, nome, comando, NULL);
                _exit(0);
            }
            close(pipes[1]);
            read(pipes[0], mensagem->execucao, sizeExecute);
            mensagem->tipo = 1;
            gettimeofday(&(mensagem->end), NULL);
            mensagem->time = time_diff(&mensagem->start, &mensagem->end);

            int fifoserver_fd = open(SERVER, O_WRONLY, 0666);
            write(fifoserver_fd,mensagem, sizeof(struct minfo));
            close(fifoserver_fd);
            _exit(0);
```

O código abaixo trata da execução de tarefas múltiplas, representadas por comandos (separados pelo operador “|”). Ele cria processos filhos para cada comando, estabelece pipes para a comunicação entre eles, redireciona a entrada e saída padrão conforme necessário e executa os comandos usando a função *execlp*. No final, fecha os descritores de arquivo desnecessários.

```c
if(mensagem->operacao == 0){ // -p vários comandos
    int pidpai = fork();
    if(pidpai==0){
        char* exec_comandos[10];
        char *string, *cmd;
        int rN=0;
        cmd = strdup(mensagem->nome);
        while((string = strsep(&cmd,"|"))!=NULL){
            exec_comandos[rN]=string;
            rN++;
        }
        exec_comandos[rN]=NULL;
        int pipes[rN-1][2];
        for(int i=0;i<rN;i++){
            if(i==0){
                pipe(pipes[i]);
                int pid0 = fork();
                if(pid0==0){
                    close(pipes[i][0]);
                    dup2(pipes[i][1],1);
                    close(pipes[i][1]);
                    char *comando;
                    comando = strdup(exec_comandos[i]);
                    char *nome = strsep(&comando, " ");
                    execlp(nome, nome, comando, NULL);
                    _exit(0);
                }
                close(pipes[i][1]);
            }
```

```c
else if(i==rN-1){
    int pidfim = fork();
    if(pidfim == 0){
        dup2(pipes[i-1][0],0);
        close(pipes[i-1][0]);
        char *comando;
        comando = strdup(exec_comandos[i]);
        char *nome = strsep(&comando, " ");
        execlp(nome, nome, comando, NULL);
        _exit(0);
    }
    close(pipes[i-1][0]);
}else{
    pipe(pipes[i]);
    int pidmeio = fork();
    if(pidmeio == 0){
        close(pipes[i][0]);
        dup2(pipes[i-1][0],0);
        close(pipes[i-1][0]);
        dup2(pipes[i][1],1);
        close(pipes[i][1]);
        char *comando;
        comando = strdup(exec_comandos[i]);
        char *nome = strsep(&comando, " ");
        execlp(nome, nome, comando, NULL);
        _exit(0);
    }
    close(pipes[i-1][0]);
    close(pipes[i][1]);
}
```

```c
read(1,mensagem->execucao,sizeExecute);
mensagem->tipo = 1;
gettimeofday(&(mensagem->end), NULL);
mensagem->time = time_diff(&mensagem->start, &mensagem->end);

int fifoserver_fd = open(SERVER, O_WRONLY, 0666);
write(fifoserver_fd,mensagem, sizeof(struct minfo));
close(fifoserver_fd);
_exit(0);
```

O servidor lida com mensagens que não são tarefas a serem executadas imediatamente. Se a operação da mensagem for "status", ele executa o comando "cat" no arquivo de saída especificado. De seguida, envia a saída desse comando de volta ao cliente por meio de um FIFO.

```c
}else if(mensagem->tipo!=1){ // Se for uma tarefa a ser escalonada
    if(mensagem->tipo == 0){
        printf("--- mensagem lida ---\n");
        if(mensagem->operacao == 2){ // status -- cat do output
            int pid = fork();

            if(pid == 0){ // O filho do filho vai executar e o filho vai mandar o
                int pipes[2]; // output diretamente para o cliente
                pipe(pipes);
                int pidfilho = fork();
                if(pidfilho == 0){
                    close(pipes[0]);
                    dup2(pipes[1], 1);
                    close(pipes[1]);
                    int fdoutput = open(OUTPUT, O_CREAT|O_RDONLY, 0666);
                    execlp("cat", "cat", OUTPUT, NULL);
                    close(fdoutput);
                    _exit(0);
                }
                close(pipes[1]);
                char outputstatus[4096];
                read(pipes[0], &outputstatus, 4096);

                char fifoc_name[30];
                sprintf(fifoc_name, CLIENT "%d", mensagem->pid);
                int fifocliente_fd = open(fifoc_name, O_WRONLY, 0666);
                write(fifocliente_fd, &outputstatus, strlen(outputstatus));
                _exit(0);
```

Cada comando é executado em um processo filho, e os pipes são usados para redirecionar a saída do comando anterior para a entrada do próximo. Após a execução dos comandos, os resultados são enviados de volta ao servidor.

Se o cliente escrever "stop" no terminal, o processo filho enviará uma mensagem para o servidor, indicando que o programa deve ser interrompido. Isso é útil para permitir que o programa seja encerrado de forma controlada.

```c
// Filho que vai parar o programa se escrever stop no input
int pipedostop[2];
pipe(pipedostop);
int pidstop = fork();
if(pidstop==0){
    char stop[5] = "nstp";
    while(strcmp(stop,"stop")!=0){
        read(0, stop, 4);
    }
    minfo mensagemstop = malloc(sizeof(struct minfo));
    mensagemstop->pid = 0;
    write(fifoserver_fd, mensagemstop, sizeof(struct minfo));
    _exit(0);
}
```

FUNÇÕES IMPLEMENTADAS

- **swapminfo:** É definida para trocar duas estruturas *minfo* num array.

```c
void swapminfo(minfo a, minfo b){
    minfo tmp;
    *tmp = *a;
    *a = *b;
    *b = *tmp;
}
```

- t**ime_diff:** Calcula a diferença de tempo entre duas estruturas *timeval*.

```c
long int time_diff(struct timeval *start, struct timeval *end) {
    return (1e+3 *(end->tv_sec - start->tv_sec)) + (1e-3 * (end->tv_usec - start->tv_usec));
}
```

- **escritanooutput:** Formata a mensagem de saída para escrever no arquivo de saída.

```c
char* escritanooutput(minfo mensagem){
    // Formata a mensagem de saída para escrever no ficheiro de output
    char* realoutput = malloc(128 + sizeExecute * sizeof(char));
    sprintf(realoutput,
    "-----------------\n  TASK (id %d, pid %d)\n   TIME %d miliseconds\n   COMMAND %s\n   OUTPUT %s\n-----------------\n"
    , mensagem->id, mensagem->pid, mensagem->time, mensagem->nome, mensagem->execucao);
    return realoutput;
}
```

POLÍTICAS DE ESCALONAMENTO

- **FCFS (First-Come, First-Served):** As tarefas são executadas na ordem em que são recebidas, sem considerar nada.

- **SJF (Shortest Job First):** As tarefas são priorizadas com base no tempo de execução. Tarefas mais curtas têm prioridade sobre as mais longas.

```c
int sc_SJF(minfo (*fila)[], int N){
    int menor_tempo = (*fila)[0]->time;
    int indice_menor_tempo = 0;
    // Procura a tarefa com o menor tempo
    for (int i = 1; i< N; i++){
        if((*fila)[i]->time < menor_tempo){
            menor_tempo = (*fila)[i]->time;
            indice_menor_tempo = i;
        }
    }

    // Move a tarefa com menor tempo para o final da fila
    for(int i=indice_menor_tempo;i<N-1;i++){
        swapminfo((*fila)[i],(*fila)[i+1]);
    }
    indice_menor_tempo = N-1;
    return indice_menor_tempo;
}
```

- **CUSTOM:** Esta política permite uma customização adicional pensada pelo grupo, onde se usa o SJF. Se uma tarefa for ultrapassada 3 vezes, independentemente do tempo de execução da mesma, esta é executada de seguida.

```c
int sc_CUSTOM(minfo (*fila)[], int N){
    int menor_tempo = (*fila)[0]->time;
    int indice_escolhido = 0;

    // Procura a tarefa com o menor tempo
    for (int i = 0; i< N; i++){
        if((*fila)[i]->custom >=3){
            indice_escolhido = i;
            break;
        }else{
            if((*fila)[i]->time < menor_tempo){
                menor_tempo = (*fila)[i]->time;
                indice_escolhido = i;
            }
        }
    }

    // Move a tarefa com menor tempo para o final da fila
    for(int i=indice_escolhido;i<N-1;i++){
        swapminfo((*fila)[i],(*fila)[i+1]);
    }
    indice_escolhido = N-1;
    return indice_escolhido;
}
```

EXECUÇÃO DAS TAREFAS

Quando uma tarefa é recebida, o *orchestrator* coloca-a na fila de espera e verifica se há espaço para executar. Se for esse o caso, aplica a política de escalonamento para decidir qual executar primeiro.

O servidor inclui um mecanismo de monitoramento que permite encerrar o sistema de forma controlada. Um processo filho é criado para monitorar a entrada do usuário, e quando o comando "stop" é escrito, o servidor é encerrado e o FIFO é removido.

Após a conclusão de uma tarefa, o servidor formata os resultados e escreve-os no ficheiro de output. Isso permite que os clientes possam ver os resultados das tarefas concluídas com a operação status.

PROBLEMAS

Na parte de encadeação de programas, tivemos um problema onde não chegámos a nenhuma solução. O nosso objetivo após executar os programas seria guardar o output numa string e escrever a mesma no ficheiro de output, ou seja, a mesma coisa que fazemos para um programa só. Por algum motivo, ao executar, o output vai para o STD_OUT. Não percebemos o porquê, pois está redirecionado para um pipe.

TESTES

Fizemos testes para os 3 tipos de escalonamento, a quantidade de tarefas que podiam ser executadas ao mesmo tempo e a ordem das tarefas (tempo ascendente, descendente e misturado).

Para todos os tipos de teste, infelizmente, os tempos ficaram semelhantes, mostrando que as nossas políticas de escalonamento não estavam bem, desenvolvidas.

(Todos os tempos estão em milissegundos.)

1 TASK	FCFS	SJF	CUSTOM
ASC	4301	4295	4292
DES	4303	4300	4297
MIX	4291	4291	4298

2 TASKS	FCFS	SJF	CUSTOM
ASC	4298	4294	4296
DESC	4299	4300	4299
MIX	4295	4293	4295

3 TASKS	FCFS	SJF	CUSTOM
ASC	4296	4295	4298
DESC	4289	4300	4298
MIX	4294	4295	4294

CONCLUSÃO

O trabalho prático não foi fácil, porém fixe. Graças às aulas práticas foi ficando claro o que tínhamos de fazer, no final ficámos com uma ideia de como pode funcionar o gestor de processos de um computador.

Em relação às políticas de escalonamento, dá para perceber que depende sempre do caso, mas em princípio (quase sempre) as formas diretas são uma perda de tempo. A nossa CUSTOM não deixa de ser simples e o tempo total não varia muito do FCFS, mas dá uma volta a possíveis deadlocks.