RESOLUÇÃO DO CONCURSO 3

SISTEMAS OPERATIVOS 2024/2025

Feito por:
a83933 Rodrigo Linhas
2º Ano da Licenciatura de Engenharia Informática (LEI)
Regente da UC: Amine Berqia

Índice

Introdução

Este trabalho tem o intuito de demonstrar a minha resolução perante os exercícios propostos no Concurso 3, juntamente com a(s) referencia(s) bibliográfica(s) consultada(s) no final do documento. Claramente que existem várias maneiras de solucionar os exercícios, no entanto com este documento, demonstra como eu resolvi.

As soluções foram desenvolvidas em linguagem C, testadas em ambiente Linux Mint e validadas com capturas de ecrã para garantir conformidade com os requisitos. A estrutura do documento segue a numeração dos exercícios, com explicações concisas, exemplos de código e saídas geradas.

Palavras chave: threads, sinais, mutex, multithread, race condition, programa, output

EX 1

a)

Para o exercício proposto, fiz a seguinte implementação:

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

int i = 0;
int trys = 5;

void sighandler() {
    i++;
    if(i < 5)   printf("\nfalta %d tentativas\n", (trys-i));
    else        printf("\ndone\n");
}

int main (int argc, char* argv[]){
    while(1){
        if (signal(SIGINT,&sighandler) == SIG_ERR ) printf("NO  signal\n");
        if(i == 5)  break;
    }
    return 0;
}
```

Obtendo o seguinte output após ter compilado e executado o programa:

```
rodrigo@rodrigomint:~/UALGSO/C3$ gcc ex1-1.c -o ex1-1.out
rodrigo@rodrigomint:~/UALGSO/C3$ ./ex1-1.out
teste
ola
1
2
3
^X
^C
falta 4 tentativas
^C
falta 3 tentativas
^C
falta 2 tentativas
^C
falta 1 tentativas
^C
done
rodrigo@rodrigomint:~/UALGSO/C3$
```

No entanto tenho que salientar que mesmo com esta implementação existe um comando reservado que "termina" o programa quando é executado, que é o *ctrl+z*:

```
rodrigo@rodrigomint:~/UALGSO/C3$ gcc ex1-1.c -o ex1-1.out
rodrigo@rodrigomint:~/UALGSO/C3$ ./ex1-1.out
teste
ola
^X
^C
falta 4 tentativas
^Z
[2]+  Interrompido            ./ex1-1.out
rodrigo@rodrigomint:~/UALGSO/C3$
```

Isto acontece uma vez que escrevi este código só para detetar o SIGINT (sinal enviado pelo ctrl+c) e não para detetar também o SIGTSTP (sinal enviado pelo ctrl+z), contudo o mesmo é só usado para suspender processos ao enviar o sinal que não é interpretado pelo programa.
Basta introduzir no programa inicial um receptor do sinal SIGTSTP que resolve este problema:

```
while(1){
    if (signal(SIGINT,&sighandler) == SIG_ERR  ||
            signal(SIGTSTP, &sighandler) == SIG_ERR) printf("NO  signal\n");
    if(i == 5)  break;
}
```

```
rodrigo@rodrigomint:~/UALGSO/C3$ gcc ex1-1-ctrl+z.c -o ex1-1-ctrl+z.out
rodrigo@rodrigomint:~/UALGSO/C3$ ./ex1-1-ctrl+z.out
teste
ola
^Z
falta 4 tentativas
^C
falta 3 tentativas
^Z
falta 2 tentativas
^C
falta 1 tentativas
^Z
done
rodrigo@rodrigomint:~/UALGSO/C3$
```

b)

Para o exercício proposto, fiz a seguinte implementação:

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

#define NUM_THREADS 5

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

const char* msg = "Mensagem da thread ";


void* thread_task(void* arg) {

    int thread_id = *(int*)arg;  // ID único da thread

    pthread_mutex_lock(&mutex);  // Bloqueia o mutex para imprimir sem interrupção

    for (int i = 0; msg[i] != '\0'; i++) {
        putchar(msg[i]);  // Imprime a mensagem letra por letra
        fflush(stdout);  // Garante que a saída seja impressa imediatamente
        sleep(1);
    }

    printf("%d\n", thread_id);      // Finaliza com o ID
    pthread_mutex_unlock(&mutex);  // Libera o mutex

    return NULL;
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // Cria todas as threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i + 1;
        if(pthread_create(&threads[i], NULL, thread_task, &thread_ids[i])!= 0)  {
            perror("Erro ao criar thread");
            return 1;
        }
    }

    // Aguarda todas as threads terminarem
    for (int i = 0; i < NUM_THREADS; i++) {
        if(pthread_join(threads[i], NULL) != 0) {
            perror("Erro ao juntar thread");
            return 2;
        }
    }

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Obtendo o seguinte output após ter compilado, executado o programa e consequentemente ter aguardado a sua finalização uma vez que imprime a(s) mensagem(ns) letra a letra:

```
rodrigo@rodrigomint:~/UALGSO/C3$ gcc ex1-2.c -o ex1-2.out
rodrigo@rodrigomint:~/UALGSO/C3$ ./ex1-2.out
Mensagem da thread 1
Mensagem da thread 2
Mensagem da thread 3
Mensagem da thread 4
Mensagem da thread 5
rodrigo@rodrigomint:~/UALGSO/C3$
```

Também tenho que realçar que **para garantir que a biblioteca *pthread.h* tenha um comportamento devido temos que especificar a flag *-pthread* ao compilar o programa**, ficando desta forma:

```
rodrigo@rodrigomint:~/UALGSO/C3$ gcc ex1-2.c -o ex1-2.out -pthread
```

Para demonstrar que o programa tem o mesmo funcionamento:

```
rodrigo@rodrigomint:~/UALGSO/C3$ gcc ex1-2.c -o ex1-2.out -pthread
rodrigo@rodrigomint:~/UALGSO/C3$ ./ex1-2.out
Mensagem da thread 1
Mensagem da thread 2
Mensagem da thread 3
Mensagem da thread 4
Mensagem da thread 5
rodrigo@rodrigomint:~/UALGSO/C3$
```

EX 2

Para o problema proposto implementei a seguinte solução:

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

bool end = false;
char msg[30];

void *routine0(){
    printf("Thread 1: ");
    fflush(stdout);
    fgets(msg, sizeof(msg), stdin);
}

void *routine1(){
    if(msg[0] == 'E' && msg[1] == '\n'){
        end = true;
    }
    for(int i = 0; i < 30; i++){
        if(msg[i] == '\n')  break;
        printf("Thread 2: ");
        printf("%c\n", msg[i]);
    }
}

int main (int argc, char* argv[]){
    pthread_t th0, th1;
    while(true){
        if(pthread_create(&th0, NULL, &routine0, NULL) != 0){
            perror("Erro ao criar thread");
            return 1;
        }
        if (pthread_join(th0, NULL) != 0){
            perror("Erro ao juntar thread");
            return 2;
        }
        if(pthread_create(&th1, NULL, &routine1, NULL) != 0){
            perror("Erro ao criar thread");
            return 1;
        }
        if (pthread_join(th1, NULL) != 0){
            perror("Erro ao juntar thread");
            return 2;
        }
        if(end){
            break;
        }
    }
    printf("Prog finished\n");
    return 0;
}
```

Obtendo o seguinte output:

```
rodrigo@rodrigomint:~/UALGSO/C3$ gcc ex2.c -o ex2.out -pthread
rodrigo@rodrigomint:~/UALGSO/C3$ ./ex2.out
Thread 1: ghjhglo gh 12
Thread 2: g
Thread 2: h
Thread 2: j
Thread 2: h
Thread 2: g
Thread 2: l
Thread 2: o
Thread 2:
Thread 2: g
Thread 2: h
Thread 2:
Thread 2: 1
Thread 2: 2
Thread 1: E
Thread 2: E
Prog finished
rodrigo@rodrigomint:~/UALGSO/C3$
```

Notando que o programa só vai detetar pelo *E* sozinho e não por palavras que contenham a letra *E* mesmo que seja maiúscula, eis um exemplo:

```
rodrigo@rodrigomint:~/UALGSO/C3$ ./ex2.out
Thread 1: ola
Thread 2: o
Thread 2: l
Thread 2: a
Thread 1: testE
Thread 2: t
Thread 2: e
Thread 2: s
Thread 2: t
Thread 2: E
Thread 1: TESTE
Thread 2: T
Thread 2: E
Thread 2: S
Thread 2: T
Thread 2: E
Thread 1: E
Thread 2: E
Prog finished
rodrigo@rodrigomint:~/UALGSO/C3$
```

EX 3

a)

Após ter compilado e executado o programa dado no enunciado, deparamos com o seguinte output:

```
rodrigo@rodrigomint:~/UALGSO/C3$ gcc prog.c -o prog.out -pthread
rodrigo@rodrigomint:~/UALGSO/C3$ ./prog.out

 Job 1 has started

 Job 2 has started

 Job 2 has finished

 Job 2 has finished
rodrigo@rodrigomint:~/UALGSO/C3$
```

b)

O comportamento observado resulta de uma condição de corrida (*race condition*) na variável global *counter*, tema que já foi abordado nas aulas teóricas. **Sendo o counter acessado concorrentemente pelas threads sem sincronização.** Ambas as *threads* incrementam *counter* imediatamente após iniciarem sua execução. **Como não há controle sobre a ordem de execução, a 2ª *thread* pode alterar *counter* para 2 antes mesmo de a primeira *thread* conclua o loop.** Assim, ao imprimir a mensagem de finalização ("*Job has finished*"), ambas as *threads* leem o valor atualizado de *counter* (2), mesmo que a primeira *thread* tenha iniciado com *counter = 1*.

Isso ocorre porque *counter* é um recurso compartilhado sem proteção (como *mutex*), permitindo que as *threads* interfiram no estado umas das outras. O resultado é uma inconsistência no output, onde ambas as mensagens de término refletem o último valor da variável, e não o valor original associado a cada *thread*. **Este cenário ilustra claramente os riscos de acesso não sincronizado a dados compartilhados em ambientes *multithread*.**

c)

Como foi dito anteriormente, precisamos então que criar um *mutex* para conseguir sincronizar as *threads* e assim resolver o problema de compilação. Fazendo as alterações necessárias ao código base:

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg) {

    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;

    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++) ;

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void) {
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]", strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

Obtendo agora um output mais coerente:

```
rodrigo@rodrigomint:~/UALGSO/C3$ gcc prog-fixed.c -o prog-fixed.out -pthread
rodrigo@rodrigomint:~/UALGSO/C3$ ./prog-fixed.out

 Job 1 has started

 Job 1 has finished

 Job 2 has started

 Job 2 has finished
rodrigo@rodrigomint:~/UALGSO/C3$
```

Demonstrando assim precisamos de usar mais métodos complementares para sincronizar as *threads* para além do *pthread_join*, daí o uso do *mutex_lock* e *mutex_unlock* para evitar este problema de *race conditon*.

Eis uma tabela que demonstra as diferenças entro o uso de *mutex* e o não uso:

	COM MUTEX	SEM MUTEX
Race condition?	Não	Sim
Acesso a variável counter	Apenas uma *thread* por vez	Todas as *threads* simultaneamente
Output	Consistente	Inconsistente, uma vez que as *threads* imprimem o seu próprio counter
Desempenho	Leve redução devido à sincronização	Mais rápido, mas resultados imprevisíveis
Casos de uso	Essencial para operações com partilha de recursos entre *threads* (este caso)	Adequado para tarefas sem partilha de recursos entre *threads*

Referências bibliográficas

Eis todos os sites que eu consultei na resolução deste concurso:

- https://superuser.com/a/262948
- https://www.geeksforgeeks.org/
- https://stackoverflow.com/
- https://hpc-tutorials.llnl.gov/posix/
- https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html
- https://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html