Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programação Concorrente

Relatório de Entrega de Atividades

Aluno(s): Amanda Oliveira Alves e Fillype Alves do Nascimento

Matrícula: 15/0116276 e 16/0070431

Atividade: Aula Prática 03 - IPC

1.1.1

```
// autor: Amanda Oliveira Alves e Fillype Alves do Nascimento
// arquivo: 1.1.1.c
// atividade: 1.1.1

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

int contador = 0;

void *incrementos() {
    for(int i=0;i<100;i++){
        contador++;
    }
    pthread_exit(NULL);
}

int main() {

    const int NUMTHREADS = 10;
    pthread_t threads[NUMTHREADS];
    for (long int i = 0; i < NUMTHREADS; i++) {
        pthread_create(&threads[i], NULL, incrementos, NULL);
    }
```

```
    for (long int i = 0; i < NUMTHREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("%d\n", contador);

    return 0;
}
```

1.1.2 - Qual o problema do código anterior? Se há algum problema, ele acontece sempre? Por quê?

R: No código do exercício 1.1.1 temos várias threads manipulando a mesma variável global sem que haja verificação se há uma outra thread ou processo realizando alterações na variável. Como o único papel da função incrementos é incrementar a variável contador, não há prejuízos quanto à ordem de qual processo/thread incrementa a variável em um dado momento, já que o resultado final é avaliado na main após todos os processos/threads serem finalizadas.

1.1.3 - De que forma seria possível resolver o problema do código, utilizando os conhecimentos já apresentados na disciplina?

R: Utilizando alguma das abordagens de espera ocupada apresentadas, a fim de fazer com que um processo/thread não tente manipular a variável global enquanto outro processo/thread estiver fazendo alterações.

1.2 - Quais são as quatro condições para se evitar condições de corrida?

R: Para evitar condições de corrida, são necessárias a verificação de quatro condições:

a. Dois ou mais processos não podem estar simultaneamente dentro de suas regiões críticas.

b. Nenhuma consideração pode ser feita a respeito da velocidade relativa dos processos, ou a respeito dos processadores disponíveis.

c. Nenhum processo que esteja fora de sua região crítica pode bloquear a execução de outro processo.

d. Nenhum processo pode ser a obrigado a esperar indefinidamente para entrar em sua região crítica.

2.1.1

```
// autor: Amanda Oliveira Alves e Fillype Alves do Nascimento
// arquivo: 2.1.1.c
// atividade: 2.1.1

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

int contabancaria = 0;
int turn = 2;

void *deposito(){
    while(turn != 1){}
    contabancaria+=20;
    printf("Efeuado depósito. Saldo: %d\n", contabancaria);
    turn = 2;
    pthread_exit(NULL);
}

void *retirada() {
    while(turn != 2){}
    if(contabancaria >= 10){
        contabancaria-=10;
    }
    printf("Efetuada retirada. Saldo: %d\n", contabancaria);
    turn = 1;
    // pthread_exit(NULL);
```

```
}

int main() {

    pthread_t t1,t2;

    pthread_create(&t1, NULL, deposito, NULL);
    pthread_create(&t2, NULL, retirada, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

2.1.2. - O que acontece, com o funcionamento do algoritmo, como um todo, se a thread responsável pelo depósito terminar sua execução?

R: Assumindo que o valor de conta bancária é inicializado com 0 no momento da declaração, a função de retirada irá iniciar a execução do algoritmo sem fazer alterações e a função depósito irá finalizar depositando 20 unidades, sendo o valor final de contabancaria igual a 20 unidades.

2.1.3

```
// autor: Amanda Oliveira Alves e Fillype Alves do Nascimento
// arquivo: 2.1.3.c
// atividade: 2.1.3

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

int contabancaria = 0;
int turn = 2;

void *deposito(){
    int i = 0;
    while(i<10000){
```

```
        while(turn != 1){}
        contabancaria+=20;
        printf("Efeuado depósito. Saldo: %d\n", contabancaria);
        turn = 2;
    }
    pthread_exit(NULL);
}

void *retirada() {
    while(turn != 2){}
    if(contabancaria >= 10){
        contabancaria-=10;
    }
    printf("Efetuada retirada. Saldo: %d\n", contabancaria);
    turn = 1;
    pthread_exit(NULL);
}

int main() {

    pthread_t t1,t2;

    pthread_create(&t1, NULL, deposito, NULL);
    pthread_create(&t2, NULL, retirada, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

2.1.4

```
// autor: Amanda Oliveira Alves e Fillype Alves do Nascimento
// arquivo: 2.1.4.c
// atividade: 2.1.4

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

```
int contabancaria = 0;
int turn = 1;

void *deposito(){
    while(turn != 1){}
    contabancaria+=20;
    printf("Efeuado depósito. Saldo: %d\n", contabancaria);
    turn = 2;
    pthread_exit(NULL);
}

void *retirada() {
    while(turn != 2){}
    if(contabancaria >= 10){
        contabancaria-=10;
    }
    printf("Efetuada retirada. Saldo: %d\n", contabancaria);
    turn = 1;
    // pthread_exit(NULL);
}

int main() {

    pthread_t t1,t2,t3;

    pthread_create(&t1, NULL, deposito, NULL);
    pthread_create(&t2, NULL, retirada, NULL);
    pthread_create(&t3, NULL, deposito, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    return 0;
}
```

3.1.1

```c
// autor: Amanda Oliveira Alves e Fillype Alves do Nascimento
// arquivo: sync.c
// atividade: 3.1.1

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

int contador = 0;

void *incrementos() {
    for(int i=0;i<100;i++){
        __sync_fetch_and_add (&contador, 1);
    }
    pthread_exit(NULL);
}

int main() {

    const int NUMTHREADS = 10;
    pthread_t threads[NUMTHREADS];
    for (long int i = 0; i < NUMTHREADS; i++) {
        pthread_create(&threads[i], NULL, incrementos, NULL);
    }
    for (long int i = 0; i < NUMTHREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("%d\n", contador);

    return 0;
}
```

3.1.2 - Lendo o conjunto de instruções do binário gerado, em qual instrução função __sync_fetch_and_add foi convertida?

R:

com contador++:

```
<       movl    contador(%rip), %eax
<       addl    $1, %eax
<       movl    %eax, contador(%rip)
```

com __sync:

```
>       lock addl       $1, contador(%rip)
```

4.1

```c
// autor: Amanda Oliveira Alves e Fillype Alves do Nascimento
// arquivo: codigo.c
// atividade: 4.1

#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main(){

	char *fn = "/tmp/output";
	char conteudo[201];
	FILE *fp;
	scanf("%200s", conteudo);
	if (!access(fn, W_OK)) {
		fp = fopen(fn, "a+");
		fwrite(conteudo, sizeof(char), strlen(conteudo), fp);
		fclose(fp);
	}
	else {
		printf("Sem permissão de escrita no arquivo.\n");
	}
	return 0;
}
```

```
FROM gcc:4.9
RUN apt-get update
RUN apt-get install -y vim nano htop
COPY codigo.c /tmp
WORKDIR /tmp
RUN gcc -o prog codigo.c
RUN useradd aluno
RUN chown root prog
RUN chmod 4755 /tmp/prog
RUN rm codigo.c
USER aluno
ENTRYPOINT ["/bin/bash"]
```

4.1.1