Projeto - Sistemas Operacionais

Q1)

// Pai

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

int main()
    Pid_t pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Erro ao criar o processo filho \n");
        return 1;
    } else if (pid == 0) {
        execlp("./filho", "filho", NULL);
    } else {
        printf("Nome do aluno: Estêvão Sousa Vieira \n PID do pai: "
            "%d \n\n", getpid());
        wait(NULL);
        printf("\n\n Fim Processo Pai \n");
    }
    return 0;
}
```

// Filho

```
#include <stdio.h>
 int main() {
     printf("Eai bicho, qual a boa? \n");
     return 0;
 }
```

Q2)

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

int main(){
    int a, b, c, d;

    printf("Digite dois valores para a soma (a e b): ");
    scanf("%d %d", &a, &b);
    printf("Digite dois valores para multiplicação (c e d): ");
    scanf("%d %d", &c, &d);

    pid_t pid1 = fork();
    if (pid1 < 0){
        printf(stderr, "Erro ao criar o processo filho 1\n");
        return 1;
    } else if (pid1 == 0){
        int soma = a + b;
        printf("Filho 1 (pid: %d) - Soma: %d\n", getpid(), soma);
        exit(soma);
    }
    pid_t pid2 = fork();

    if (pid2 < 0)
        printf(stderr, "Erro ao criar o processo filho 2\n");
        return 1;
    } else if (pid2 == 0){
```

```
    int mult = c * d;
    printf("Filho 2 (Pid: %d) - Multiplicação: %d \n", getpid(),
           mult);
    exit(mult);
}

int status1, status2;
waitpid(pid1, &status1, 0);
waitpid(pid2, &status2, 0);

int resultado_soma = WEXITSTATUS(status1);
int resultado_mult = WEXITSTATUS(status2);
int resultado_final = resultado_mult / resultado_soma;
printf("Pai (PID: %d) - Resultado Final: %d \n", getpid(),
       resultado_final);
return 0;
}
```

Q3)

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

int main(){
    int a, b, c, d;
    int pipefd1[2], pipefd2[2];
    if (pipe(pipefd1) == -1 || pipe(pipefd2) == -1){
        perror("pipe");
        exit(EXIT_FAILURE);
    }
```

```
printf("Digite dois valores para soma (a e b): ");
scanf("%d %d", &a, &b);
Printf("Digite dois valores para multiplicação (c e d): ");
scanf("%d %d", &c, &d);

Pid_t pid1 = fork();
if (pid1 < 0){
    perror("fork");
    exit(EXIT_Failure);
} else if (pid1 == 0){
    close(pipefd1[0]);
    int soma = a+b;
    write(pipefd1[1], &soma, sizeof(soma));
    printf("Filho 1 (PID:%d) - Soma: %d\n", getpid(),
           soma);
    exit(EXIT_Success);
}
Pid_t pid2 = fork();
if(pid2 < 0){
    perror("fork");
    EXIT(EXIT_FAILURE);
} else if (pid2 == 0){
    close(pipefd2[0]);
    int mult = c*d;
    write(pipefd2[1], &mult, sizeof(mult));
    printf("Filho 2 (PID:%d) - Multiplicação: %d\n",
           getpid(), mult);
    close(pipefd2[1]);
    exit(Exit_Sucess);
}
```

```
close(pipefd1[1]);
close(pipefd2[1]);
int soma, mult;
read(pipefd1[0], &soma, sizeof(soma));
read(pipefd2[0], &mult, sizeof(mult));
close(pipefd1[0]);
close(pipefd2[0]);

int resultado_final = mult / soma;
printf("Pai (PID:%d) - Resultado Final: %d\n", getpid(), resultado_
    final);
wait(NULL);
wait(NULL);
return 0;
}
```

Q4)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/shm.h>
#include <sys/ipc.h>

int main() {
    int a, b, c, d;
    printf("Digite dois valores para soma (a e b): ");
    [redacted]
    scanf("%d %d", &a, &b);
    printf("Digite dois valores para multiplicação (c e d): ");
```

```
scanf("%d %d", &c, &d);

key_t key1 = ftok("shmfile1", 65);
key_t key2 = ftok("shmfile2", 75);

int shmid1 = shmget(key1, sizeof(int), 0666 | IPC_CREAT);
int shmid2 = shmget(key2, sizeof(int), 0666 | IPC_CREAT);

int * soma = (int *) shmat(shmid1, (void *) 0, 0);
int * mult = (int *) shmat(shmid2, (void *) 0, 0);

pid_t pid1 = fork();

if (pid1 < 0){
    perror("fork");
    exit(EXIT_FAILURE);
} else if (pid1 == 0) {
    *soma = a + b;
    printf("filho 1 (PID: %d) - Soma: %d\n", getpid(), *soma);
    exit(EXIT_SUCCESS);
}
pid_t pid2 = fork();
if (pid2 < 0){
    perror("fork");
    exit(EXIT_FAILURE);
} else if (pid2 == 0){
    *mult = c * d;
    printf("filho 2 (PID: %d) - Multiplicação: %d\n",
        getpid(), *mult);
        exit(EXIT_SUCESS);
}
```

```
wait (NULL);
wait (NULL);

int resultado_final = *mult / *soma;
printf("Pai (PID: %d) - Resultado Final: %d\n", getpid(),
    resultado_final);

shmdt (soma);
shmdt (mult);

shmctl (shmid1, IPC_RMID, NULL);
shmctl (shmid2, IPC_RMID, NULL);

return 0;
}
```

Q5)

Java: Em Java, a sincronização entre threads é essencial para evitar condições de corrida e garantir consistência dos dados compartilhados.

- O Java fornece várias ferramentas para a sincronização

1. synchronized:

- Utilizado para métodos ou blocos de código que precisam ser acessados por apenas uma thread por vez.
  Garante que a P uma thread pode acessar a seção de código ao mesmo tempo.

2. 'Lock' e 'ReentrantLock'

- Classes da biblioteca 'java.util.concurrent.locks' que oferecem mais controle sobre o bloqueio do que 'synchronized'.
Permite bloqueios mais flexíveis e pode ser usado para implementar bloqueios de tempo e tentativas de bloqueio.

3. 'Semaphore':

- Uma ferramenta de sincronização que controla o acesso a um recurso compartilhado através de contagem de permissões. Permite que um número fixo de Threads acesse o recurso ao mesmo tempo.

4. 'Volatile':

- Utilizado para variáveis que podem ser acessadas por várias Threads. Garante que a leitura e escrita no variável são sempre feitas a partir da memória principal, não do cache da Thread.

. 'Atomic Variables':

- Classes como 'AtomicInteger', 'AtomicBoolean', 'AtomicLong', etc., fornecem operações atômicas para variáveis primitivas. Permitem a leitura e escrita de forma atômica sem a necessidade de sincronização explícita.

GO

GO, projetado para concorrência e oferece várias ferramentas para sincronização entre goroutines:

1. Channels (Canais):

- Conductos de comunicação que permitem a passagem de dados entre goroutines.
Podem ser criados com ou sem buffer e são utilizados para sincronizar o acesso a dados compartilhados.
Um canal sem buffer, que garante que o goroutine remetente fique bloqueado até que o goroutine receptor leia os dados, sincronizando automaticamente as goroutines.

2. sync.Mutex e sync.RWMutex:

- Mutex é usado para criar exclusão mútua, garantindo que apenas uma goroutine pode acessar uma seção crítica de código por vez.

RWMutex permite múltiplas leituras simultâneas ou uma única escrita, oferecendo mais eficiência quando há mais operações de leitura do que escrita.

3. sync.WaitGroup:

- Facilita a espera pelo término de várias goroutines.
Utilizando para aguardar até que todas as goroutines em um grupo tenha finalizado sua execução.

4. 'Sync. Cond':

Permite que threads esperem até serem notificadas para continuar.
Usado em conjunto com 'Mutex' para criar condições complexas de sincronização.

5. Context

Usado para propagar deadlines, cancelamento e outros sinais de tempo entre goroutines.
Facilita o controle de tempo e cancelamento de operações concorrentes.