FACULDADE: CENTRO UNIVERSITÁRIO DE BRASÍLIA – UniCEUB
CURSO: ENGENHARIA DE COMPUTAÇÃO
DISCIPLINA: SISTEMAS DE TEMPO REAL E EMBARCADOS
CARGA HORÁRIA: 60 H. A. **ANO/SEMESTRE**: 2020/02
PROFESSOR: ADERBAL BOTELHO
HORÁRIOS: SEGUNDAS E QUARTAS

LABORATÓRIO – SISTEMAS DE TEMPO REAL

RESUMO

Sistemas de Tempo Real são sistemas computacionais especializados capazes de reagir a estímulos oriundos de seu ambiente em prazos específicos, ou seja, para garantir a corretude do sistema é necessário atender as restrições temporais. O cálculo do tempo e sincronicidade das tarefas é realizado através da troca de mensagens, sinalizados por eventos. O laboratório vai trabalhar os conceitos de processos, tarefas e como ocorre a interação entre eles.

OBJETIVOS

Objetivo Geral
Compreender os elementos conceituais relacionados aos eventos e trocas de mensagem em sistemas de tempo real.

Objetivos Específicos

1. Entender o conceito de tarefas;
2. Entender o conceito de threads;
3. Entender o conceito de eventos;
4. Introduzir a linguagem de programação C.

EXERCÍCIO 1 – PROCESSOS

Para o exercício serão executadas rotinas na linguagem C. Considere o seguinte cabeçalho para seus programas:

```
1 #include <stdio.h>
2 int main() {
3     int i = 0;
4 }
```

O programa contém os elementos como se segue

1. Define as bibliotecas a serem utilizadas no programas;
2. Define a função main, que é executada ao chamar o script;
3. Declaração de variáveis;

EXERCÍCIO 1 – PROCESSOS

4. Fecha os parênteses da função main

- Crie um programa que execute uma tarefa na máquina e consuma o máximo de processador possível. Ex.: Conte todos os números até 1000.

EXERCÍCIO 2 – COMUNICAÇÃO ENTRE PROCESSOS

As seguintes funções estão disponíveis para serem utilizadas nos códigos da linguagem C:

fork()
Cria dois processos idênticos: pai e filho

wait()
Força o processo pai a esperar pela execução do filho

exit()
Finaliza o processo que chama a função e retorna o valor de saída exit.

Considere o seguinte trecho de código na linguagem C:

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4
5 int main()
6 {
7 pid_t pid;
8
9     /* fork a child process */
10     pid = fork();
11
12     if (pid < 0) { /* error occurred */
13         fprintf(stderr, "Fork Failed\n");
14         return 1;
15     }
16     else if (pid == 0) { /* child process */
17         printf("I am the child %d\n",pid);
18         execlp("/bin/ls","ls",NULL);
19     }
20     else { /* parent process */
21         /* parent will wait for the child to complete */
22         printf("I am the parent %d\n",pid);
23         wait(NULL);
24
25         printf("Child Complete\n");
26     }
27
28     return 0;
```

EXERCÍCIO 2 – COMUNICAÇÃO ENTRE PROCESSOS

```
29 }
```

Você é capaz de identificar os trechos executados por pai e filho?

Considere ainda o exemplo a seguir:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("%d\n",getpid());
6     fork();
7     printf("%d\n",getpid());
8
9
10     fork();
11     printf("%d\n",getpid());
12
13     fork();
14     printf("%d\n",getpid());
15
16     return 0;
17 }
```

Quantos processos serão criados?

EXERCÍCIO: Construa um exemplo, na linguagem C, onde os processos pai e filho trocam algum tipo de mensagem.

EXERCÍCIO 3 – THREADS

Na linguagem C, em sistemas Linux a criação de uma thread pode ser feita executando uma chamada à função clone()

clone()
Faz uma cópia compartilhada do processo (similar ao fork) mas mantém uma área de memória compartilhada.

Em sistemas Linux utilizamos o termo *task* para se referir a processos e threads. A principal diferença é o tipo de recurso que será compartilhado.

EXERCÍCIO 3 – THREADS

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Figura 1: Flags para criação de threads

O trecho abaixo apresenta uma criação de thread no Linux:

```
 1 #define _GNU_SOURCE
 2 #include <stdlib.h>
 3 #include <malloc.h>
 4 #include <sys/types.h>
 5 #include <sys/wait.h>
 6 #include <signal.h>
 7 #include <sched.h>
 8 #include <stdio.h>
 9
10 // 64kB stack
11 #define FIBER_STACK 1024*64
12
13 // The child thread will execute this function
14 int threadFunction( void* argument )
15 {
16     printf("child thread exiting\n");
17     return 0;
18 }
19
20 int main()
21 {
22     void* stack;
23     pid_t pid;
24
25     // Allocate the stack
26     stack = malloc( FIBER_STACK );
27     if ( stack == 0 )
28     {
29         perror("malloc: could not allocate stack");
30         exit(1);
31     }
32
33     printf( "Creating child thread\n" );
34
35     // Call the clone system call to create the child thread
36     pid = clone( &threadFunction, (char*) stack + FIBER_STACK,
37         SIGCHLD | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_VM, 0 );
38     if ( pid == -1 )
```

EXERCÍCIO 3 – THREADS

```
39      {
40      ¦   perror( "clone" );
41      ¦   exit(2);
42      }
43
44      // Wait for the child thread to exit
45      pid = waitpid( pid, 0, 0 );
46      if ( pid == -1 )
47      {
48          perror( "waitpid" );
49          exit(3);
50      }
51
52      // Free the stack
53      free( stack );
54      printf( "Child thread returned and stack freed.\n" );
55
56      return 0;
57  }
```

EXERCÍCIO: Incremente o exemplo acima para representar a troca de informações de contexto entre a thread e o processo pai.

EXERCÍCIO – PARA CASA

Considere o programa fatorial abaixo:

```
# include <stdio.h>

int main()
{
        int i, num, j;
        printf ("Enter the number: ");
        scanf ("%d", &num );

        for (i=1; i<num; i++)
                j=j*i;

        printf("The factorial of %d is %d\n",num,j);
}
```

1. Transforme o problema em um exemplo multiprocessado;
2. Compare o desempenho das chamadas *fork()* e *clone()*. Qual possui melhor desempenho? Por quê?

BIBLIOGRAFIA

SILBERSCHATZ, Abraham et al. **Operating system concepts**. Reading: Addison-Wesley, 1998.

BIBLIOGRAFIA

TANENBAUM, Andrew S.; MACHADO FILHO, Nery.**Sistemas operacionais modernos**. Prentice-Hall, 1995.