



# Monitores

Walter Fetter Lages

w.fetter@ieee.org

Universidade Federal do Rio Grande do Sul
Escola de Engenharia
Departamento de Engenharia Elétrica

ENG04008 Sistemas de Tempo Real

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL


# Introdução

- Regiões crítcas tendem a ser disperas pelo programa
- Monitores implementam um controle mais estruturado
- Sincronização implementada pelos monitores é mais eficiente
- Monitor é um módulo que encapsula as regiões críticas escritas como procedimentos
  - Variáveis acessadas em exclusão mútua são ocultas
  - Procedimentos executam em exclusão mútua

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL

# Exemplo

```
monitor buffer;
export append, take;
var (* declaração das variáveis *)

procedure append(I:integer);
    ...
end;

procedure take (var I:integer);
    ...
end;

begin
    (* inicialização *)
end.
```

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL


# Variáveis de Condição

- Dentro do monitor a exclusão mútua é garantida
- Ainda existe a necessidade de sincronização
- Variáveis de condição admitem apenas duas operações
  - `wait(c)`
    - Bloqueia o processo na variável c
    - Libera a exclusão mútua do monitor
  - `signal(c)`
    - Desbloqueia um processo que está bloqueado em c
    - Não tem efeito se não houver processo bloqueado

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL


# Exemplo

```
procedure append(I:integer);
begin
     If inbuffer=size then wait(space);
     buf[top]:=I;
     inbuffer:=inbuffer+1;
     top:=(top+1) mod size;
     signal(item);
end;
```

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL


# Problema de Exclusão Mútua

- Quando um processo executa um `signal()` e desbloqueia outro, existirão dois processo executando no monitor
- Modificação na semântica do `signal()`
  - `signal()` só é permitido como última instrução no monitor
  - `signal()` executa implicitamente um return, forçando a saída do monitor
  - Se o `sinal()` desbloquear outro processo, ele bloqueia a sí mesmo O processo desbloqueado roda penas após o processo que chamou `signal()` deixar o monitor

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL


# Variáveis de Condição POSIX

- `int pthread_cond_init(`
  `pthread_cond_t *cond,`
  `pthread_condattr_t *cond_attr)`
  - Variáveis do tipo `pthread_cont_t` podem ser inicializadas com a constante `PTHREAD_COND_INITIALIZER`
- `int pthread_condattr_init(`
  `pthread_condattr_t *attr)`
- `int pthread_condattr_destroy(`
  `pthread_condattr_t *attr)`
- `int pthread_cond_destroy(`
  `pthread_cond _t *cond)`

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL


# Variáveis de Condição POSIX

- `int pthread_cond_signal(`
  `pthread_cond_t *cond)`
- `int pthread_cond_broadcast(`
  `pthread_cond_t *cond)`
- `int pthread_cond_wait(`
  `pthread_cond_t *cond,`
  `pthread_mutex_t *mutex)`
- `int pthread_cond_timedwait(`
  `pthread_cond_t *cond,`
  `pthread_mutex_t *mutex,`
  `const struct timespec *abstime)`

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL


# Mutexes POSIX

- `int pthread_mutex_init(`
  `pthread_mutex_t *mutex,`
  `const pthread_mutexattr_t *`
  - Variáveis do tipo `pthread_mutex_t` podem ser inicializadas com a constante `PTHREAD_MUTEX_INITIALIZER`
- `int pthread_mutexattr_init(`
  `pthread_mutexattr_t *attr)`
- `int pthread_mutexattr_destroy(`
  `pthread_mutexattr_t *attr)`
- `int pthread_mutex_destroy(`
  `pthread_mutex_t *mutex)`

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL


# Mutexes POSIX

- `int pthread_mutex_lock(`
  `pthread_mutex_t *mutex)`
- `int pthread_mutex_trylock(`
  `pthread_mutex_t *mutex)`
- `int pthread_mutex_unlock(`
  `pthread_mutex_t *mutex)`

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL


# Implementação de Monitor

```
#include <pthreads.h>
#define SIZE 10
typedef struct
{
    pthread_mutex_t mutex;
    pthread_cond_t notfull;
    pthread_cond_t not empty;
    int count, first, last;
    int buf[SIZE];
} buffer;
```

# Implementação de Monitor

```
int append(int item, buffer *b)
{
    pthread_mutex_lock(&b->mutex);
    while(b->count==SIZE)
       pthread_cond_wait(&b->notfull,
                         &b->mutex);
    b->buf[b->first]=item;
    b->count++;
    b->first=(b->first+1) % SIZE;
    pthread_mutex_unlock(&b->mutex);
    pthread_cond_signal(&b->notempty);
    return 0;
}
```

# Implementação de Monitor

```
int take(int *item, buffer *b)
{
     pthread_mutex_lock(&b->mutex);
     while(b->count==0)
          pthread_cond_wait(&b->notempty
                            &b->mutex);
     item=b->bufb->[last];
     b->last=(b->last+1) % SIZE;
     b->count--;
     pthread_mutex_unlock(&b->mutex);
     pthread_cond_signal(&n->notfull);
     return 0;
}
```