Centro de Ciências Exatas e Tecnológicas Universidade Federal de Viçosa

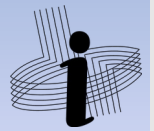
INF 310 – Programação Concorrente e Distribuída

Sincronizações Básicas

Professor: Vitor Barbosa Souza
vitor.souza@ufv.br

As três sincronizações básicas

- Possíveis sincronizações entre 2 *threads* executando operações distintas
  - Execução com exclusão mútua
  - Um processo deve executar após o outro
  - As operações devem iniciar "simultaneamente"



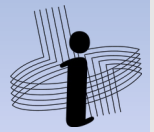
UFV

As três sincronizações básicas

- Operações primitivas
  - mutexbegin/mutexend
    - Define um bloco de código que será executado com exclusão mútua

```
...
mutexbegin;
Região Crítica;
mutexend;
...
```

  - block/wakeup
    - block – bloqueia o processo que executa esta operação
    - wakeup(T) – desbloqueia a *thread T*
      - Vamos considerar *wakeup* com memória (*stateful*)



UFV

As três sincronizações básicas

- Sincronização para compartilhamento
  - Processos compartilhando um recurso comum que deve ser usado com exclusão mútua

  - Thread T1:

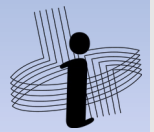
```
...
mutexbegin;
Região Crítica;
mutexend;
...
```

  - Thread T2:

```
...
mutexbegin;
Região Crítica;
mutexend;
...
```

  - Thread T3:

```
...
mutexbegin;
Região Crítica;
mutexend;
...
```



UFV

As três sincronizações básicas

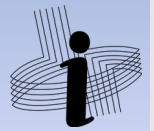
- Sincronização para comunicação
  - Define uma sequência de execução
  - Um processo espera pela sinalização do outro processo para avançar
    - Thread T1:

```
...
block;
A;
...
```

    - Thread T2:

```
...
B;
wakeup(T1);
...
```

  - Efeito: operação A será executada após a operação B



As três sincronizações básicas

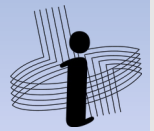
- Sincronização tipo barreira
  - Define uma execução “simultânea” de duas operações
    - Thread T1:

```
...
wakeup(T2);
block;
A;
...
```

    - Thread T2:

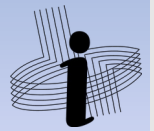
```
...
wakeup(T1);
block;
B;
...
```

  - Efeito: operação A só começa quando B também estiver pronta para começar (e vice-versa)



As três sincronizações básicas

- Sobre a necessidade de operações básicas
  - As operações mutexbegin/mutexend e block/wakeup(p) são consideradas “suficientes” para implementar qualquer sincronização
  - Estas operações podem ser implementadas a partir do “nada”
    - Os algoritmos de Dekker (para 2 processos) e Lamport (para *n* processos) mostram que é possível fazer a sincronização de exclusão mútua sem usar nenhuma operação especial



UFV

Mutexbegin/Mutexend em C++

- pthread_mutex_t <pthread.h>

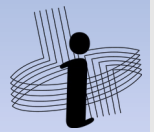
```
pthread_mutex_t m;              //define um mutex
pthread_mutex_init(&m,&attr); //inicializa o mutex
pthread_mutex_lock(&m);         //bloqueia até conseguir a trava (lock)
pthread_mutex_trylock(&m);      //retorna 0 se não obter o lock (não bloqueia)
pthread_mutex_timedlock(&m,&abstime); //bloqueia até determinado tempo
pthread_mutex_unlock(&m);       //libera a trava
pthread_mutex_destroy(&m);      //destrói o mutex
```

- std::mutex <mutex>

```
std::mutex m;                   //define um mutex
m.lock();                       //bloqueia até conseguir a trava
m.try_lock();                   //retorna false se não obter o lock
m.unlock();                     //destrava o mutex
```

- std::timed_mutex <mutex>
  - Além de lock, try_lock e unlock

```
std::timed_mutex tm;            //define um mutex com tempo
tm.try_lock_for(rtime);         //bloqueia durante um tempo
tm.try_lock_until(abstime);     //bloqueia até o determinado tempo
```



UFV

Mutexbegin/Mutexend em C++

- Exemplo 5a (*mutex* compartilhado)

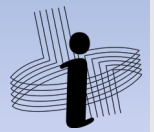
```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

int s=0;
mutex mux;

void incrementa(int n) {
    for (int i=0; i<n; ++i) {
        mux.lock();
        ++s;
        mux.unlock();
    }
}
```

```
void decrementa(int n) {
    for (int i=0; i<n; ++i) {
        mux.lock();
        --s;
        mux.unlock();
    }
}

int main() {
    thread t1(incrementa,1000);
    thread t2(decrementa,1000);
    t1.join();
    t2.join();
    cout<<s<<endl;
    return 0;
}
```



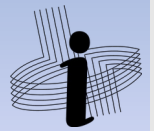
Mutexbegin/Mutexend em C++

- Gerenciamento da trava através do *std::unique_lock*
  - *mutex* tem proprietário único
  - trava é liberada no *destroy* do *unique_lock*

```
#include<mutex>
using namespace std;

mutex mux;

void f() {
    unique_lock<mutex> lck(mux);
    ...
}
```



UFV

Block/Wakeup()

- Implementando *block/wakeup()* com *mutexbegin/mutexend*

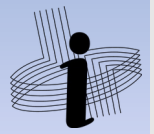
```
#include<pthread.h>
#include<map>

pthread_mutex_t mux=
        PTHREAD_MUTEX_INITIALIZER;
std::map<pthread_t,int> A;

void block() {
    bool sair=false;
    long eu=pthread_self();
    do {
        pthread_mutex_lock(&mux);
        if(A[eu] > 0) {
            A[eu]--;
            sair=true;
        }
        pthread_mutex_unlock(&mux);
    } while (!sair);
}
```

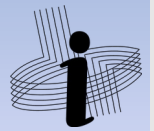
```
void wakeup(pthread_t t) {
   pthread_mutex_lock(&mux);
   A[t]++;
   pthread_mutex_unlock(&mux);
}

int main() {
  int num=10;
  for(int i=0;i<num;++i) {
     pthread_t t;
     pthread_create(&t,NULL,func,NULL);
     // t armazena o tid (thread id)
     A.insert(std::pair<pthread_t,int>
              (t,0));
  }
  for (auto &p:A)
     pthread_join(p.first,NULL);
  pthread_mutex_destroy(&mux);
}
```



Solução de problemas

- Solução de problemas com mutexbegin/mutexend e block/wakeup
  - Forma geral
    - Garantir exclusão mútua para manipular dados compartilhados
    - Examinar o estado atual do sistema
      - Se o estado permitir, então executar a função desejada e fazer *bloquear=false*
      - Caso contrário, fazer *bloquear=true* e colocar a sua identificação no fim da fila de bloqueados
    - Liberar a exclusão mútua
    - Se *bloquear=true* então bloquear a si mesmo. Caso contrário, prosseguir



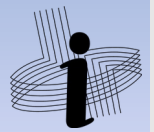
UFV

Solução para o problema do alocador de recursos

```
...
pthread_mutex_t muxrec;
int T=5;
int R[]={5,4,3,2,1};
queue<pthread_t> fila;

void libera(int u) {
    pthread_mutex_lock(&muxrec);
    R[T++]=u;
    if(!fila.empty()) {
        pthread_t id=fila.front();
        fila.pop();
        wakeup(id);
    }
    pthread_mutex_unlock(&muxrec);
}
```

```
int requisita() {
    int r;
    bool bloquear;
    do {
        bloquear=false;
        pthread_mutex_lock(&muxrec);
        if(T==0) {
            bloquear=true;
            fila.push(pthread_self());
        } else {
            r=R[--T];
        }
        pthread_mutex_unlock(&muxrec);
        if(bloquear) block();
    } while(bloquear);
    return r;
}
```



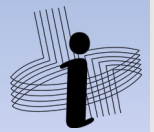
UFV

Solução para o problema do alocador de recursos

```
void* func(void* id){
    if(id==0) {
    /* código executado pela thread1 */
        sleep(1);
        int u;
        for (int i=0;i<5;++i) {
            u=requisita();
            cout<<"1 usando "<<u<<endl;
        }
    }
    else {
    /* código executado pela thread2 */
        int u;
        u=requisita();
        cout<<"2 usando "<<u<<endl;
        sleep(5);
        libera(u);

    }
}
```

```
int main() {
   pthread_mutex_init(&muxrec,NULL);
   int num=2;
   for(int i=0;i<num;++i) {
      pthread_t t;
      pthread_create(&t,NULL,func,
                     (void*)i);
      A.insert(pair<pthread_t,int>(t,0));
   }
   for (auto &p:A) {
      pthread_join(p.first,NULL);
   }
   pthread_mutex_destroy(&muxrec);
}
```



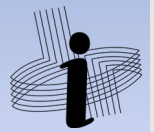
Variações do *mutex*

- *Mutex* parametrizado
  - Generalização das primitivas para acesso com exclusão mútua para diferentes conjuntos de dados
  - Uso aninhado pode provocar *deadlock*
    - P1:

```
…
mutexbegin(X);
"usa X";
mutexbegin(Y);
"usa X e Y";
mutexend(Y);
mutexend(X);
…
```

    - P2:

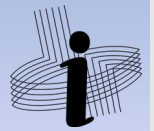
```
…
mutexbegin(Y);
"usa Y";
mutexbegin(X);
"usa X e Y";
mutexend(X);
mutexend(Y);
…
```



UFV

Variações do *mutex*

- *pthread_spin_lock* (espera ocupada) pode ser utilizado para fazer uma espera ocupada, ao invés do *mutex* quando a espera é muito pequena, por exemplo, incrementar um contador
  - Vantagem: no *unlock*, não é necessário ficar verificando se outras *threads* estão esperando
    - uma *thread* que não consegue obter o *lock* não é colocada para dormir e consegue executar a região crítica com latência menor assim que a trava estiver disponível
  - Desvantagem: existe o risco, mesmo que pequeno, da *thread* com a trava perder a CPU antes do *unlock* (melhor utilizar apenas em modo privilegiado, permitindo desabilitar *scheduling*)



UFV

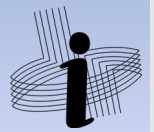
Barreiras

- C/C++

```
#include <pthread.h>
pthread_barrier_t b;                 //definição de uma barreira
pthread_barrier_init(&b,&attr,n); //inicialização para n threads
pthread_barrier_wait(&b)             //bloquear até as n threads chamarem
pthread_barrier_destroy(&b)          //destrói a barreira
```

  - O retorno de *pthread_barrier_wait* é a constante *PTHREAD_BARRIER_SERIAL_THREAD* para apenas uma das threads
- C++20 inclui o pacote *barrier*

Após liberar as threads o valor da barreira volta para seu valor definido na inicialização.
Útil para loops!



UFV