# Relatório I de Sistemas Operacionais II - INE5424
# Implementação da fila de bloqueados para semáforo e mutex

**Bruno Izaias Bonotto - 16104059**
**Fabíola Maria Kretzer - 16100725**
**João Vicente Souto - 16105151**

## 1. INTRODUÇÃO

A proposta do primeiro exercício prático da disciplina consiste em alterar a implementação dos dispositivos de sincronização (semáforo e mutex) com o objetivo de melhorar o desempenho do sistema operacional EPOS e garantir a correta execução da aplicação *Philosopher's Dinner*. A abordagem utilizada para garantir a exclusão mútua dos recursos utilizados pela aplicação consistiu em impedir que uma *thread* voltasse a concorrer pelo uso do processador ao falhar no acesso a seção crítica, ao invés disso, a *thread* é inserida em uma fila do dispositivo de sincronização. Desta forma, suspendem-se as *threads* que não podem acessar a seção crítica até que essa esteja disponível. Garante-se, portanto, que as *threads* bloqueadas só sejam escalonadas segundo a ordem de chegada, que por sua vez é garantida pelo sincronizador de *threads.*

## 2. DESENVOLVIMENTO

### a. synchronizer.h

Em *synchronizer.h* foi adicionado uma fila de *threads* (*Synchronized_Queue*) denominada *_blocked* para armazenar as *threads* bloqueadas. Dessa maneira garante-se o ordenamento de forma que a primeira a entrar é a primeira a sair (FIFO). Além disso, foram modificadas as funções *sleep()*, *wakeup()* e *wakeup_all()*.

Ao chamar a função *sleep()*, a *thread* em execução é inserida na fila *_blocked* utilizando um link exclusivo (*_link2*). Em seguida é executada a função *suspend()*, onde ela será removida do processador, terá seu estado alterado para *SUSPENDED* e será inserida na fila *_suspend*, onde permanecerá até que seja sua vez de acessar a seção crítica solicitada. Por fim, outra *thread* será escalonada para uso do processador.

A função *wakeup()* é responsável por "acordar" a primeira *thread* da fila de bloqueadas, ou seja, ela é removida da fila *_blocked*. Posteriormente é chamada a função

*resume()*, responsável por alterar o estado da *thread* para *READY* e inseri-la novamente na fila *_ready,* permitindo que ela seja novamente escalonável.

Por fim, a função *wakeup_all()* executa o mesmo passo da função anterior, porém, ao invés de remover apenas a primeira *thread* da fila *_blocked*, esta "acorda" todas as *threads* bloqueadas, permitindo que todas elas concorram novamente para acessar a seção crítica.

**b. thread.h e thread.cc**

Para que o dispositivo de sincronização pudesse controlar a ordem de solicitação de acesso à seção crítica, foi necessário incluir na classe *thread.h* uma fila, denominada *Synchronized_Queue (typedef Queue<Thread>*) onde a prioridade levada em conta fosse unicamente a ordem de chegada. Além disso, um novo *link* foi criado, denominado *_sync_link*, viabilizando a inserção e remoção dos objetos na lista de bloqueio.

No entanto, com a finalidade de evitar qualquer conflito de tipos e, além disso, permitir melhor legibilidade do código, o nome do tipo das filas de escalonamento foi alterado de *Queue* para *Priority_Queue (typedef Ordered_Queue<Thread, Priority>*), o que demandou uma alteração na declaração das filas no arquivo *thread.cc*.

**c. semaphore.cc**

Para garantir a execução de forma "atômica" de algumas operações, o *synchronizer.h* fornece as operações *begin_atomic()* e *end_atomic()*, onde desabilita e habilita, respectivamente, as interrupções da CPU (funções utilizadas para garantir que as funções *p()* e *v()* executem sem condição de corrida).

Neste sentido, a *thread* que estiver executando a função *p()* primeiramente verifica (de forma "atômica") se o valor do semáforo é menor ou igual a zero antes de decrementá-lo, o que impede que a variável de controle do semáforo (*_value*) seja decrescido indiscriminadamente. Caso o valor seja menor que 1, a *thread* executará a função *sleep()* do sincronizador e, caso contrário, executará a função *fdec()* do mesmo, que por sua vez será responsável por decrementar a variável de controle do semáforo de forma apropriada.

Ao chamar a função *v(),* a *thread* verificará (através do sincronizador) se a fila de bloqueadas está vazia. Caso esteja ela executará a função *wakeup()* do sincronizador e, caso contrário  contrário, executará a função *finc()* do mesmo, que por sua vez será responsável por incrementar a variável de controle do semáforo de forma apropriada.

### d. mutex.cc

As funções *lock()* e *unlock()* do mutex seguem a mesmas ideias descritas no item anterior.

No entanto, ao entrar na função *lock()* do mutex, a *thread* executa a função *tsl()* do sincronizador, que faz *swap* entre o 1 e o valor da variável de controle do mutex (*_locked*). Caso o valor obtido seja 0 a *thread* continua seu fluxo de execução e, caso contrário, significa que a seção crítica está sendo acessado por outra *thread* e por conta disso está executará a função *sleep()*, onde permanecerá aguardando a liberação do mutex.

Já na função *unlock()* do mutex, a variável de controle é setada para 0. Ao final, a *thread* que está saindo da seção crítica executará a função *wakeup_all()* do sincronizador fazendo com que todas as *threads* bloqueadas concorram novamente pelo recurso solicitado (padrão *Posix*).

## 3. CÓDIGO FONTE

### 1. synchronizer.h

```
class Synchronizer_Common
{
    …
    typedef Thread::Synchronized_Queue Queue

    void sleep()
    {
        _blocked.insert(&(Thread::running()->_sync_link));
        Thread::running()->suspend();
    }

    void wakeup()
    {
        _blocked.remove()->object()->resume();
    }

    void wakeup_all()
    {
        while(!_blocked.empty())
            wakeup();
    }
```

```
bool has_any_blocked()
{
    return !_blocked.empty();
}

    Queue _blocked;
};
```

2. **thread.h**

```
class Thread
{
    …
    typedef Ordered_Queue<Thread, Priority> Priority_Queue;
    typedef Queue<Thread> Synchronized_Queue;
    ...
protected:
    ...
    Priority_Queue::Element _link;
    Synchronized_Queue::Element _sync_link;
    …
};

…

template<typename ... Tn>
inline Thread::Thread(int (* entry)(Tn ...), Tn ... an)
: _state(READY), _link(this, NORMAL), _sync_link(this)
{...}

template<typename ... Tn>
inline Thread::Thread(const Configuration & conf, int (* entry)(Tn ...), Tn ... an):
_state(conf.state), _link(this, conf.priority), _sync_link(this)
{...}

…
```

3. **thread.cc**

```
…
Thread::Priority_Queue Thread::_ready;
Thread::Priority_Queue Thread::_suspended;
...
```

4. **semaphore.c**

```
…
void Semaphore::p()
{
    ...
    begin_atomic();

    if (_value == 0)
        sleep();
    else
    {
        fdec(_value);
        end_atomic();
    }
}

void Semaphore::v()
{
    ...
    begin_atomic();

    if(has_any_blocked())
        wakeup();
    else
        finc(_value);

    end_atomic();
}
…
```

5. **mutex.cc**

```
…
void Mutex::lock()
{
    ...
    begin_atomic();

    if (tsl(_locked))
        sleep();

    end_atomic();
}
```

```
void Mutex::unlock()
{
    ...
    begin_atomic();

    _locked = false;
    wakeup_all();

    end_atomic();
}
```

## 4. CONCLUSÃO

A implementação à priori dos mecanismos de exclusão mútua do sistema operacional EPOS não estava completamente de acordo com o padrão *Posix*. No caso do semáforo, a variável de controle poderia ser decrementada indiscriminadamente de forma que, mesmo a variável sendo incrementada com a função *v()*, as demais *threads* não conseguiriam acessar a seção crítica e permaneceriam bloqueadas por tempo indeterminado. Para corrigir esse problema, as *threads* só decrementam a variável caso o valor dela seja maior que 0 (usando as funções *begin_atomic()* e *end_atomic()* para impedir condição de corrida). Outro problema que antes ocorria era o fato de que o semáforo não gerenciava a ordem de bloqueio das *threads* que solicitaram acesso à seção crítica, onde o único tratamento era retirá-las do processador. Para garantir a execução por ordem de chegada, caso não seja possível acessar a seção crítica solicitada, a thread é inserida no final da fila do processador e mantém-se no estado bloqueada, impedindo de ser novamente escalonada. Dessa forma, quando uma *thread* finaliza a execução da seção crítica caso a fila de bloqueadas esteja vazia incrementa a variável de controle e, caso contrário, desbloqueia a primeira da fila de bloqueadas, que por sua vez terá acesso ao recurso solicitado.

Para a implementação do mutex, embora não fosse necessário a criação de uma fila de *threads* bloqueadas, o fato de impedi-las de serem escalonadas quando não podem acessar o recurso solicitado pode aumentar o desempenho do sistema. No entanto, para confirmar esta hipótese, seria necessário uma série de experimentos os quais permitissem obter resultados de desempenho usando ambas as formas de implementação.