INE5645 — Programação Paralela e Distribuída

Trabalho II: Programação Distribuída

Pedro Santi Binotto (20200634)
Gabriel Lemos da Silva (18200628)
Bruno Alexandre Schmitt Costa (17105532)

Introdução do tema

Modelo de memória compartilhada entre processos

Escopo

O sistema desenvolvido implementa uma rede de processos que mantém um espaço de memória compartilhada. A comunicação entre estes processos acontece através de uma API que oferece operações de leitura e escrita, através da abstração de “blocos” de memória:

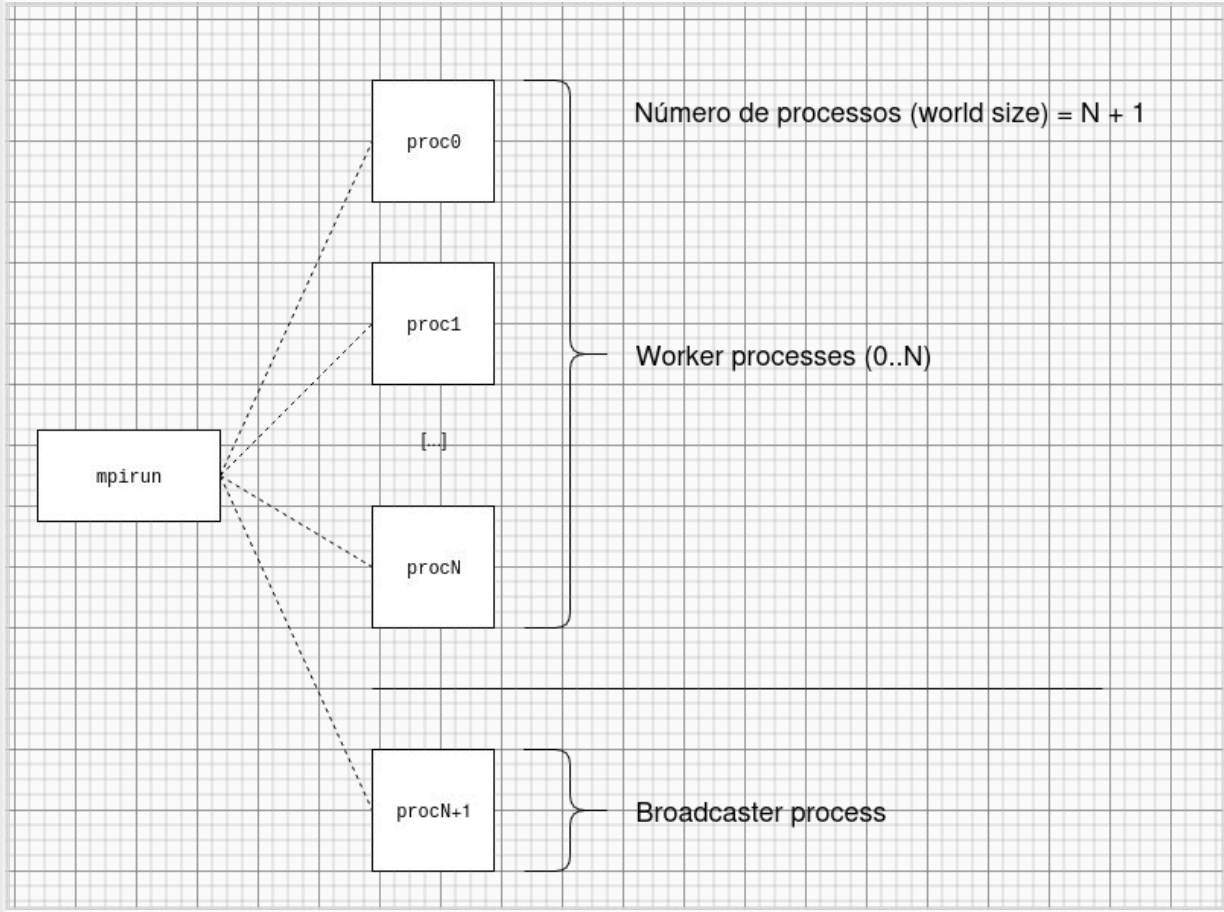
- Cada processo é responsável por “gerenciar” uma parcela dos blocos totais de memória;
- Cada processo acessa os blocos remotos através da API mantida pelo processo “responsável” pelo bloco;
- Os processos mantém *caches* locais representando os blocos remotos;

Organização e Arquitetura

Seja N informado pelo usuário e $(N \geq 0)$, o número total de processos instanciados é dado por $(N + 1)$.

Os processos $(0 \,..\, N)$ são *workers* e $(N + 1)$ é o *broadcaster* dedicado.

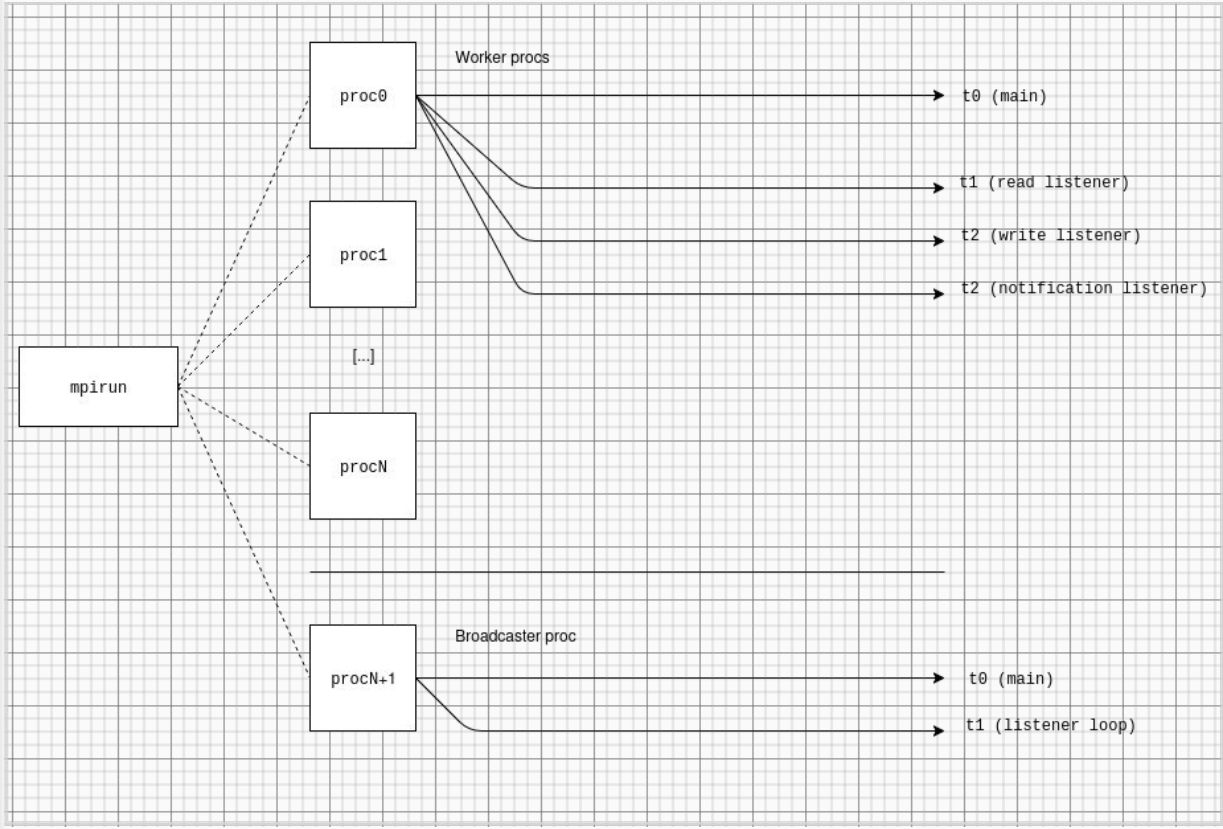
- ➢ *Processos "worker" mantém e gerenciam blocos da memória;*
- ➢ *Processos "worker" realizam computações sobre estes blocos;*
- ➢ *O "broadcaster" repassa notificações para os demais proessos.*



Organização e Arquitetura

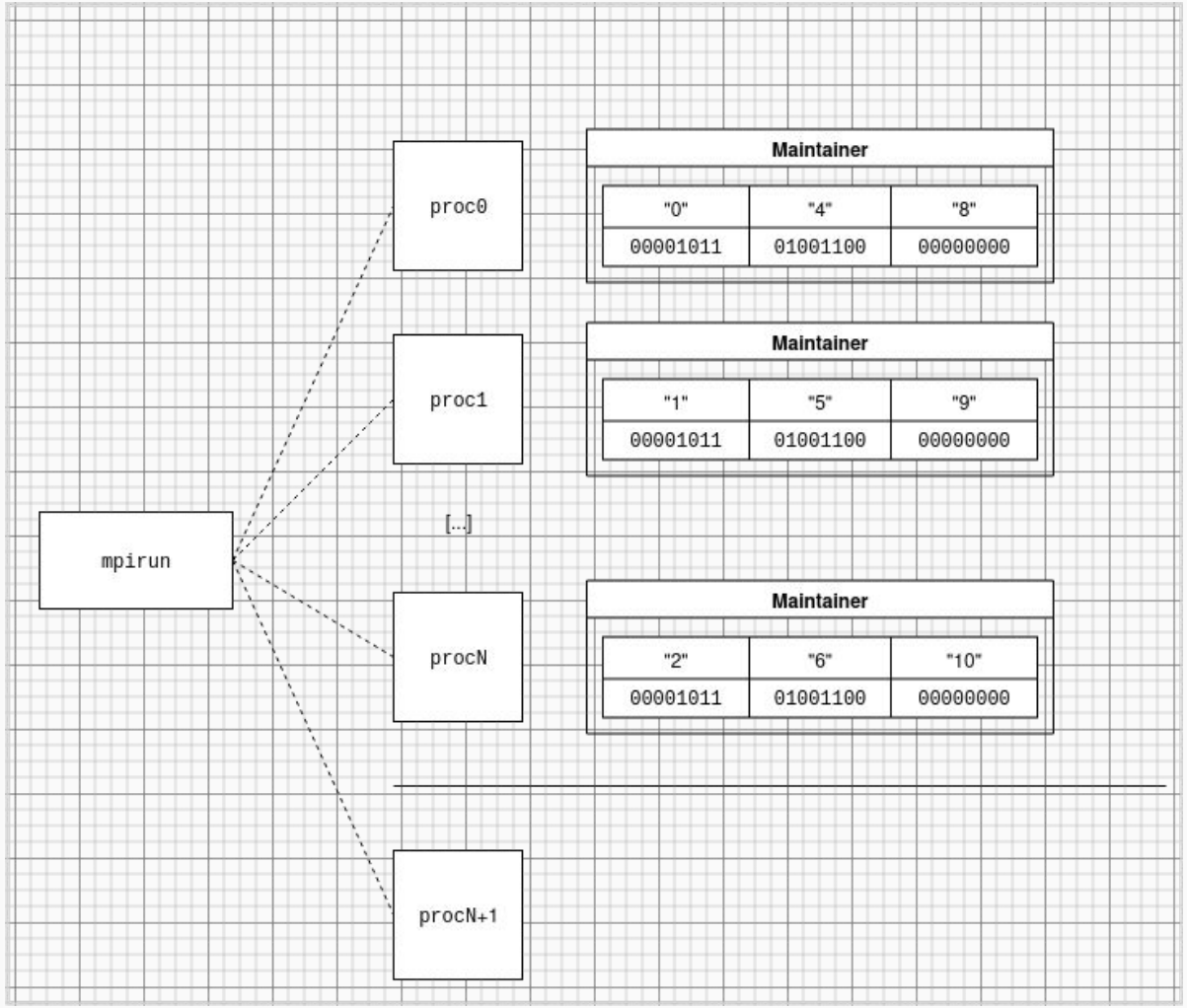
Cada processo *worker* apresenta quatro threads paralelas:

- Thread `t0 (main)` é a thread principal do processo; realiza leitura, computação e escrita sobre os blocos de memória;
- Thread `t1 (read)` é a thread que realiza o *listener loop* para atender requisições de leitura sobre os blocos gerenciados;
- Thread `t2 (write)` é a thread que realiza o *listener loop* para atender requisições de escrita;
- Thread `t3 (notification)` é a thread que receberá notificações so processo *broadcaster;*



Organização e Arquitetura

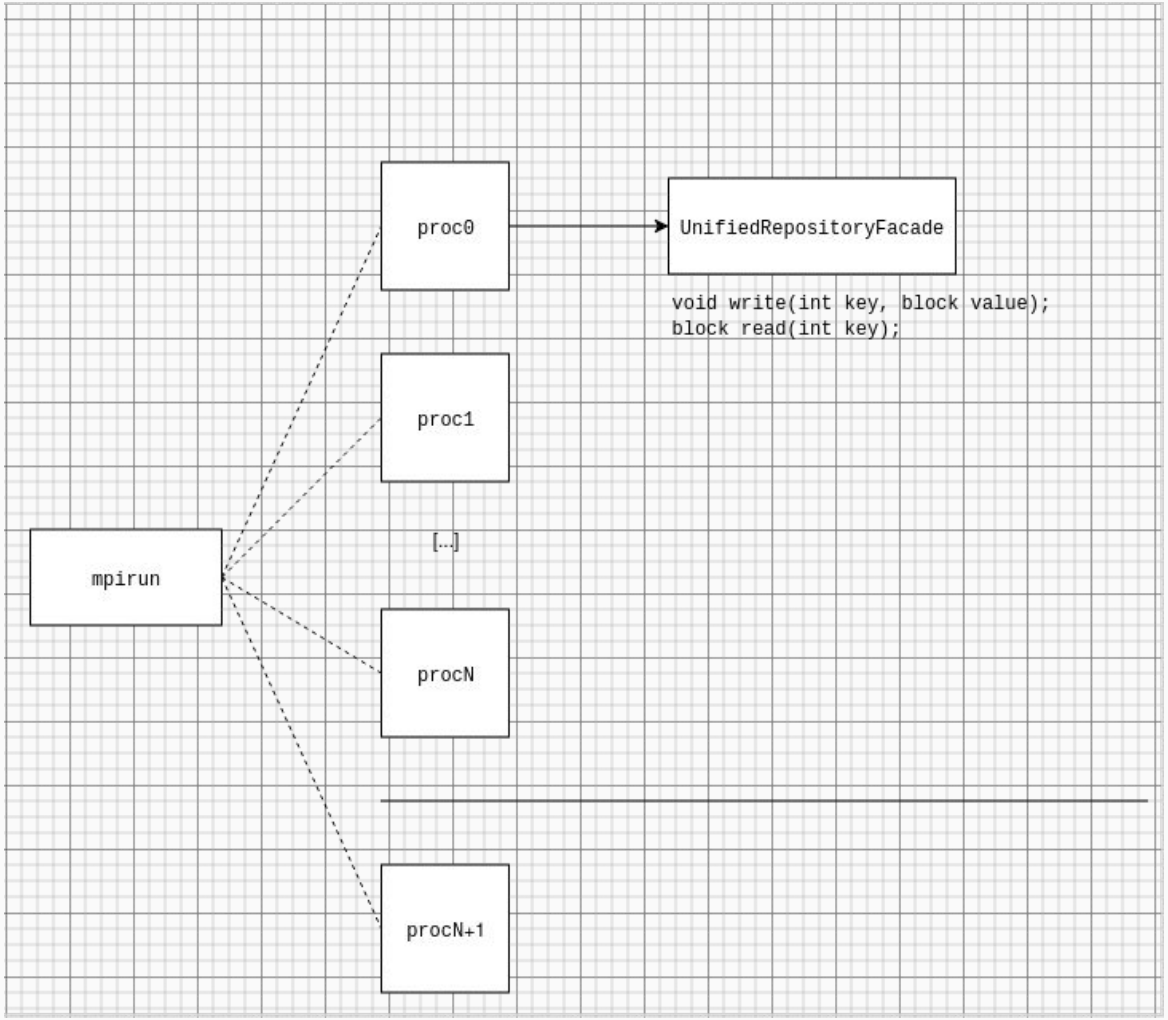
Ao iniciar a execução dos processos, os K blocos de memória serão distribuídos entre os N processos *worker* (round-robin, $0 \ldots N$), resultando em um “mapa de memória” conhecido por todos os processos, que associará os endereços de cada bloco com seu processo gerenciador.



Organização e Arquitetura

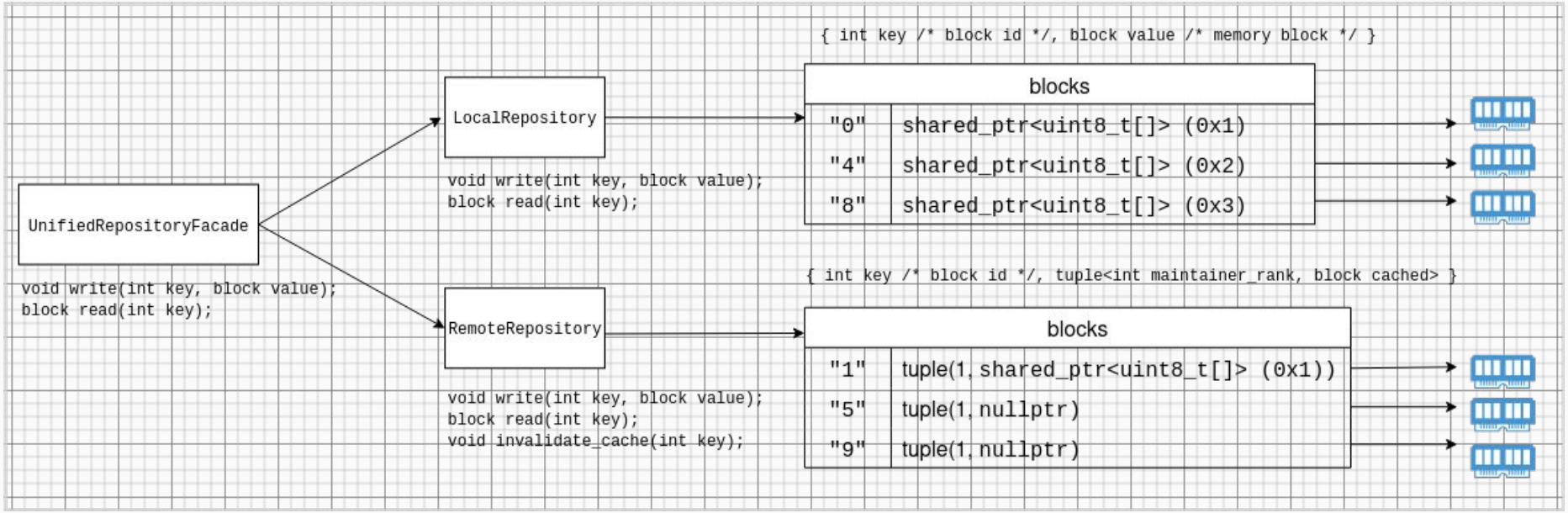
Ao receber o mapa de memória, cada processo *worker* inicializará uma cada de acesso de dados através da class `UnifiedRepositoryFacade;`

A abstração de acesso de dados permitirá à thread principal do processo realizar operações de leitura e escrita sobre os blocos de memória de forma transparente, sem que seja necessário implementar rotinas separadas para acessos remotos e locais.



Organização e Arquitetura

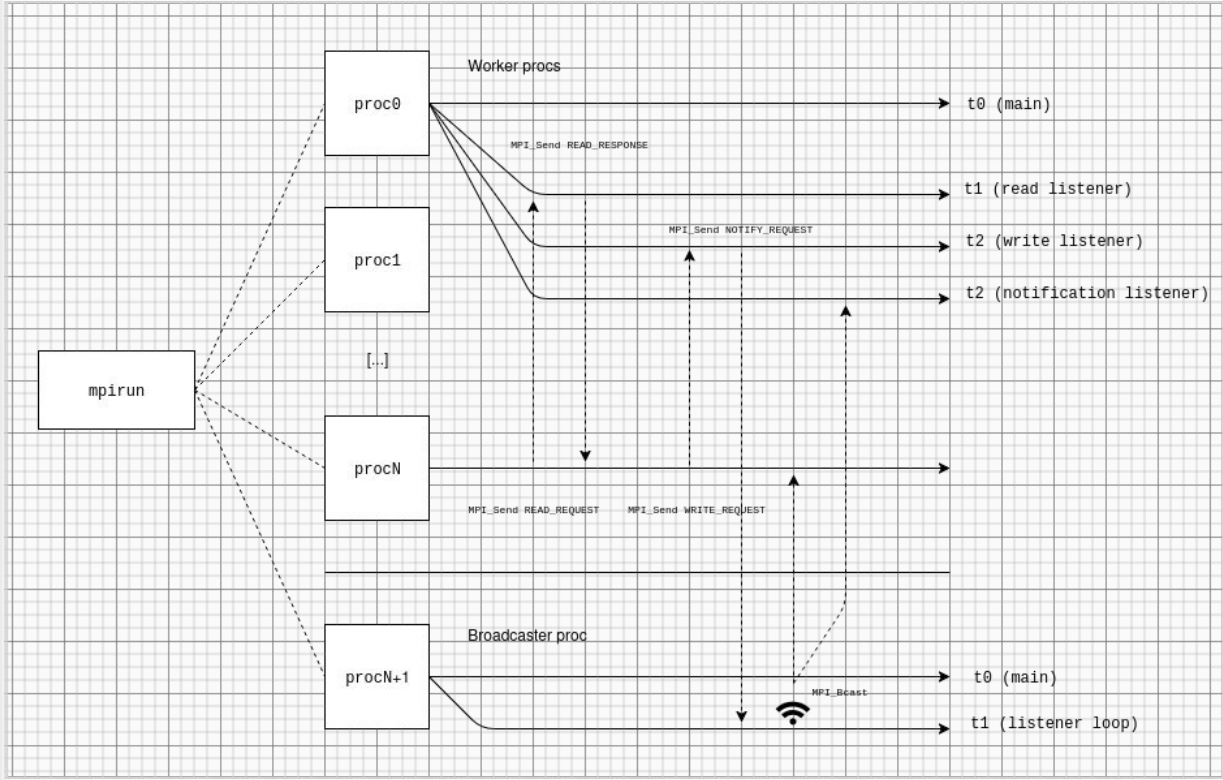
A nivel de implementação, a classe `UnifiedRepositoryFacade` oferece uma interface simples para acessar as estruturas de manutenção (`LocalRepository`) de dados local assim como de leitura/escreita remota (`RemoteRepository`)



Organização e Arquitetura

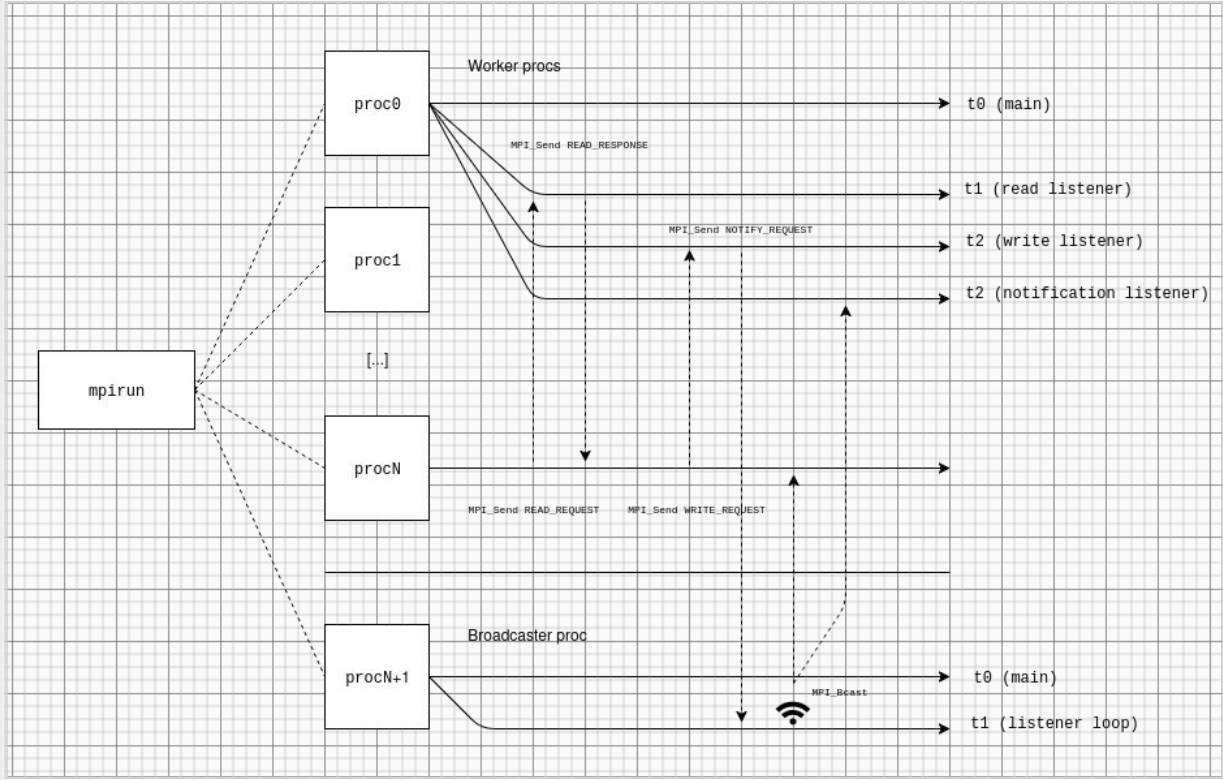
O modelo de *“life-cycle”* básico do processo então é o seguinte:

- A *thread* principal do processo *worker* realizará operações de leitura e escrita sobre os blocos;
- As *threads* auxiliares, por sua vez, executam um loop que aguarda a comunicação de eventos de leitura, escrita, e notificações de atualização vindos de outros processos;
- O processo *broadcaster* aguardará o evento de atualização de um bloco para realizar o broadcast do evento.



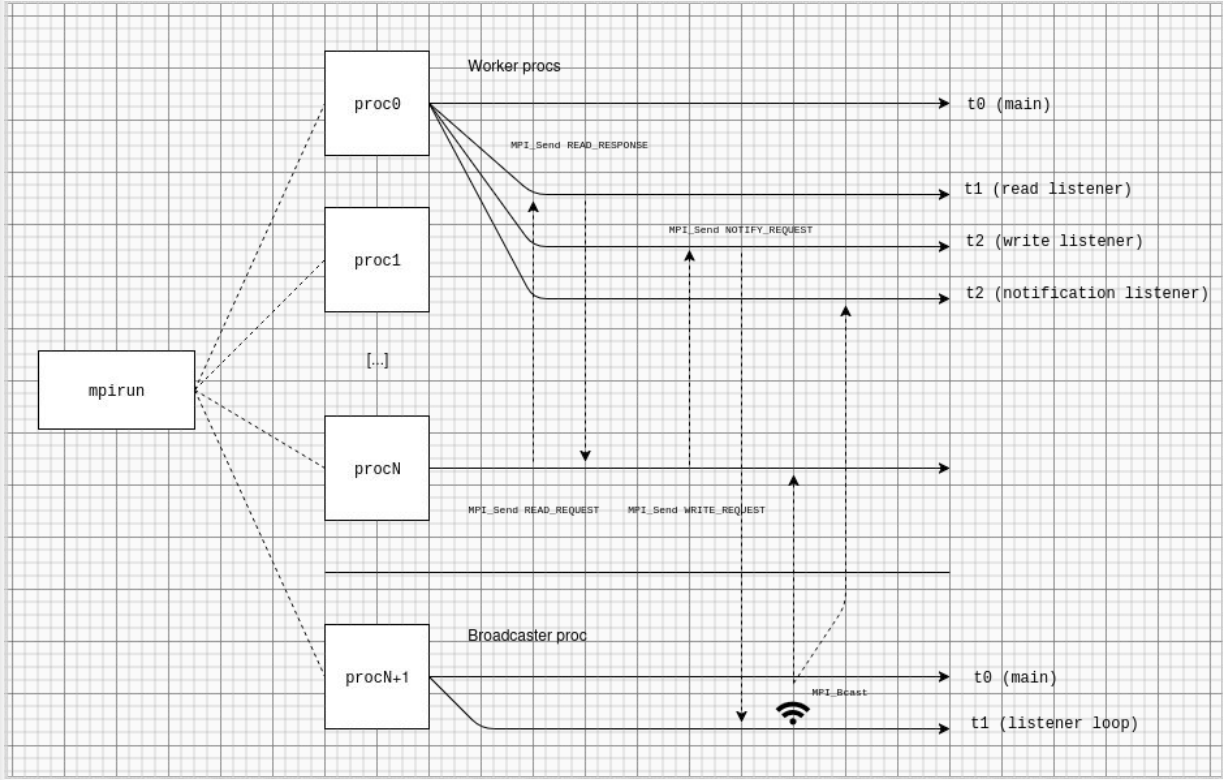
Organização e Arquitetura

- Quando uma operação de **escrita** é realizada sobre um bloco (localmente ou por um evento remoto), o processo gerenciador do(s) bloco(s) atualizado(s) enviará uma requisição de notificação para o processo *broadcaster*, contendo o *ID* do bloco atualizado, juntamente com o *timestamp* (UNIX) do momento da atualização;
- A notificação será enviada em **broadcast** para todos os *workers*, que invalidarão suas *caches* locais referentes àquele bloco, consultando diretamente do processo remoto na próxima operação de leitura.



Organização e Arquitetura

- A nível de processo, as operações de leitura e escrita sobre os blocos são protegidas de condições de corrida através do uso de mutexes (read/write) que permitem leituras concorrentes mas não escritas/invalidações;
- O evento de notificação enviado em **broadcast** contém também o *timestamp* da computação do bloco, para que, em um cenário “concreto”, possam ser realizadas invalidações de produções que fossem realizadas utilizando valores obsoletos, etc.



Parametrização e detalhes de implementação

O programa permite a parametrização de atributos para produzir comportamentos distintos sob circunstâncias diversas. Através no `Makefile`, podem ser parametrizados dos valores:

- Tamanho do bloco (em *bytes*);
- Número de blocos (total, será distribuído entre os processos);
- Número de *workers*;

Observabilidade

Para que seja possível observar e analisar o comportamento dos processos, o programa oferece diferentes níveis de log, assim como saída para `stdout` e escrita em arquivos de log (separados por rank do processo).

Observabilidade

Também é possível executar o programa no modo *debug*, que manterá uma instância do `gdb` (GNU Debugger) para cada processo, possibilitando o monitoramento mais precisa da execução do código.

Arquivos `.log`

- Ao executar a aplicação, serão produzidos no diretório `log/` arquivos que registram a atividade dos processos, permitindo que seja observado o que foi executado por cada instância, em ordem cronológica.

```
.
├── log
│   ├── 1752196454
│   ├── ...
│   └── 1752253576
│           ├── proc-0_output.log
│           ├── proc-1_output.log
│           ├── proc-2_output.log
│           ├── proc-3_output.log
│           └── proc-4_output.log
├── Makefile
├── README.md
└── src
     └── ...
```

Arquivos `.log`

- Em cada arquivo, encontra-se o que o processo submeteu ao `stdout` durante a execução. Dependendo do nível de `log` configurado, isso pode incluir apenas as transaçõs (recebeu dados para tal bloco, enviou requisição de leitura, etc.) como também um *dump* da memória representando o estado interno do processo a cada iteração.

```
Process assigned world rank 0 successfully initialized MPI
[BLOCK_SIZE, NUM_BLOCKS] não informados; usando parâmetros padrão:
[8, 4]
Process assigned world rank 0 successfully parsed program args
Process assigned world rank 0 successfully validated program args
[1752262034] (@proc 0) Started as worker process
[New Thread 0x7fffee1ff6c0 (LWP 124222)]
[1752262034] (@proc 0) Notification listener thread started
[1752262034] (@proc 0) Notification listener probing...
[New Thread 0x7fffed9fe6c0 (LWP 124224)]
[1752262034] (@proc 0) Write thread started
[1752262034] (@proc 0) Write listener probing...
[New Thread 0x7fffed1fd6c0 (LWP 124229)]
[1752262034] (@proc 0) Read thread started
[1752262034] (@proc 0) Started helper threads
[1752262034] (@proc 0) Read listener probing...
[1752262034] (@proc 0) Hello, World! from processor bridge, rank 0 out of 5
processors
[1752262034] (@proc 0) WRITE operation to block 2 called at remote repository
level
[1752262034] (@proc 0) Received method call to execute remote WRITE operation to
block 2 with value 01100111 11000110 01101001 01110011 01010001 11111111 01001010
00000000  on process ID 2
[1752262034] (@proc 0) Encoding write message from of key: 2, value: 01100111
11000110 01101001 01110011 01010001 11111111 01001010 00000000  as bytearray
buffer 00000010 00000000 00000000 00000000 01100111 11000110 01101001 01110011
01010001 11111111 01001010 00000000

[1752262034] (@proc 0) Sending WRITE request of key: 2, value: 01100111 11000110
01101001 01110011 01010001 11111111 01001010 00000000  serialized as 00000010
00000000 00000000 00000000 01100111 11000110 01101001 01110011 01010001 11111111
01001010 00000000  over MPI
```

Modo *debug*

- O modo *debug* permite também analisar a execução individual de cada processo em tempo real, através do GNU Debuger. Exemplo ao lado, com $N = 4$ (4 *workers* + 1 *broadcaster*).

Obrigado!

Por:
Bruno Alexandre Schmitt Costa [17105532]
Gabriel Lemos da Silva [18200628]
Pedro Santi Binotto [20200634]