# Relatório P5

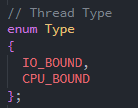
Alan Djon Ludke (17205070)

Pablo Daniel Riveros Strapasson (16100740)

Felipe de Campos Santos (17200441)

## (a) How did you intend to solve the problem?

Baseado na nossa entrega do P4, usaríamos o atributo *\_type* que adicionamos aos processos para decidir em qual core cada processo iria rodar.



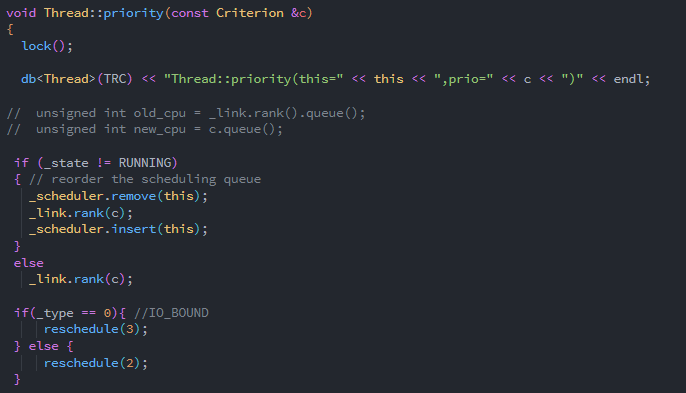
**Imagem 0:** Atributo \_type em process.h

Optamos por dividir os 4 cores em 2 grupos, um para CPU\_BOUND e um para IO\_BOUND. Levando isso em consideração, e sabendo que todo processo se inicia como CPU\_BOUND, todos processos criados começaram rodando nas CPUS 0 e 1, e seriam separados de forma a evitar uma sobrecarga da CPU, atribuindo os processos à CPU que estivesse com a fila mais vazia (menor \_size dentro da fila de processos do scheduler) ou ficando mais tempo em IDLE. Optamos por manter os processos CPU\_BOUND nos cores 0 e 1 porque, como todos os processos são criados como CPU\_BOUND e todos iniciam no core 0, fariamos a migração apenas depois da criaçao, pois a checagem e migração de processos pode ser custosa no sistema.

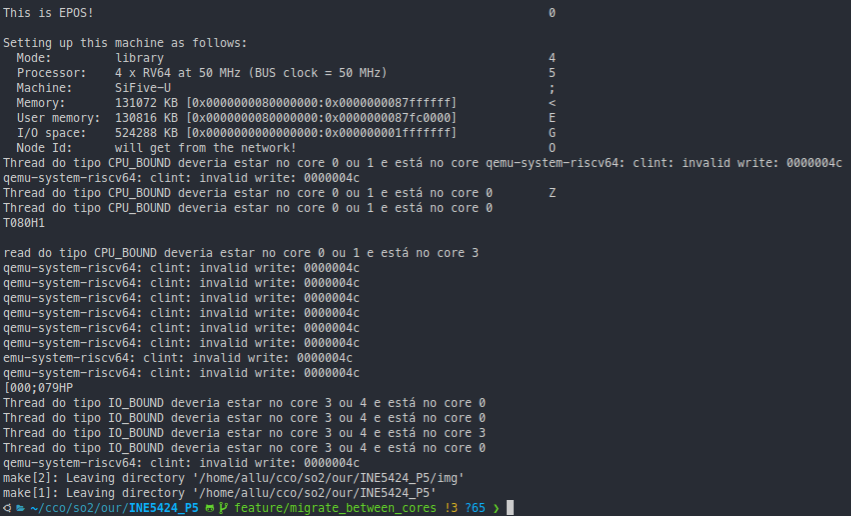
Sempre que ocorresse um *reschedule*, confirmaríamos primeiro qual o tipo do processo, e a partir disso selecionaríamos à qual *cluster* de CPUs o processo seria repassado. Com isso feito, uma verificação de qual CPU estaria “menos usada” entre as duas do cluster (fila de processos menor ou mais tempo em idle) seria feita e assim os processos seriam atribuídos à CPU com menor carga de trabalho.

## (b) Which parts of the solution have been implemented and how?

Atualmente temos implementadas a parte de atribuição dos processos para as CPU’s de acordo com o seu atributo *\_type*. Isso é feito dentro do método *priority* no arquivo Thread.cc, onde testamos o atributo *\_type* para IO\_BOUND, caso verdadeira a condição chamamos o método *reschedule*, também presente em Thread.cc, passando como parâmetro a id da CPU destino para a *thread.*

**Imagem 1:** Método priority() sendo usado para alternância entre cores.

Implementamos também a parte de migração de threads, usando o método *reschedule(id)*, que recebe o número da CPU para qual gostaríamos de migrar. Aqui encontramos nossa trava, pois recebemos *CLINT: invalid write.*

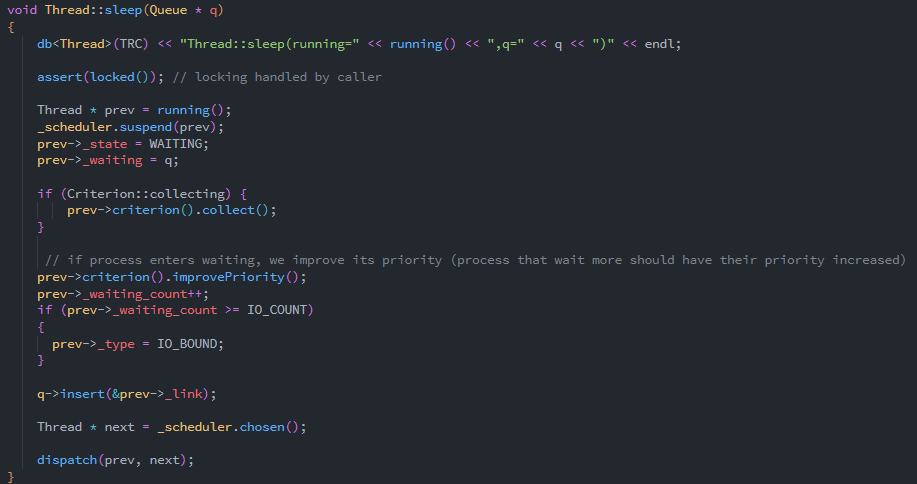


**Imagem 2:** Output de p5\_test com erro de clint.

Imaginamos que o erro seja causado por um endereço errado dentro do vetor de interrupts da IC::IP, mas não avançamos o suficiente para entender como corrigi-lo.

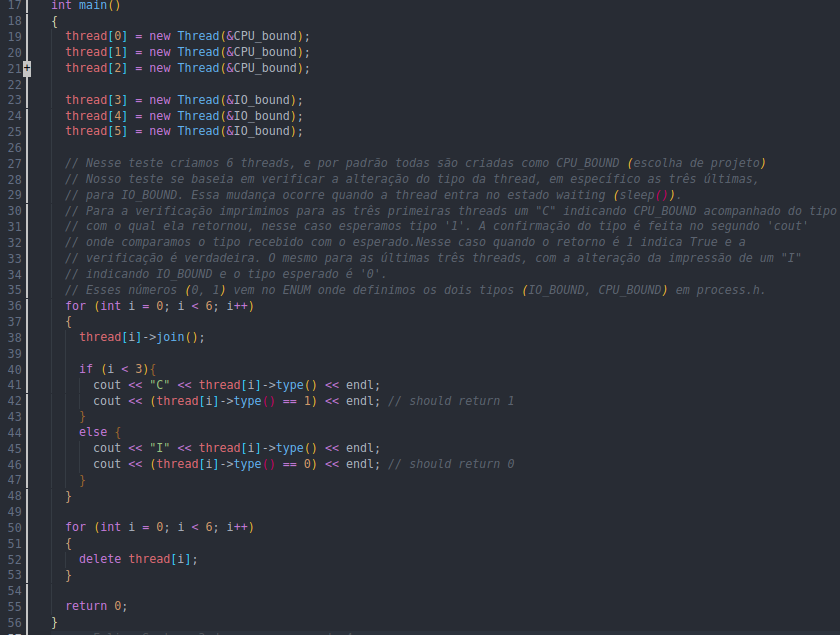
## (c) Which parts have been validated and how they have been validated?

No processo atual, cada vez que uma thread entra em *sleep* (indicando uma espera por IO), aumentamos em 1 seu contador de “quantas vezes entrou em sleep” e atribuímos a esse processo o tipo IO\_BOUND.

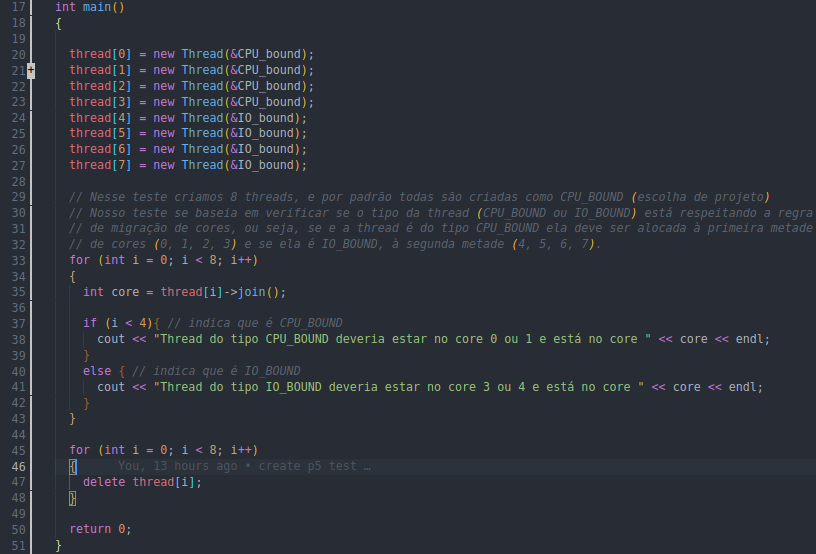


**Imagem 3**: método sleep em thread.cc com atribuição de \_type

Para validarmos isso, temos um teste (p4\_test) que cria 6 threads, 3 CPU\_BOUND e 3 IO\_BOUND, e espera que no final, as 3 últimas tenham o tipo trocado (lembrando que todas iniciam com tipo CPU\_BOUND). Já em outro teste (p5\_test), além do citado acima, retornamos junto do tipo da thread em qual *core* ela rodou, esperando que as threads IO\_BOUND tenham sido migradas para os *cores* 2 e 3.



**Imagem 4**: Definição de teste p4\_test



**Imagem 5:** Definição de teste p5\_test

## (d) How would you implement and validate the remaining parts in case you had more time?

Para podermos prosseguir com a implementação e testes, primeiro teríamos que corrigir o erro de endereço da CLINT. Usaríamos o manual para confirmar os endereços dos bits que estão sendo usados agora, e assim saberíamos se o erro é esse mesmo ou outra coisa. Acreditamos que, com isso corrigido, geraríamos corretamente uma interrupção no core destino, e assim nosso código faria a migração corretamente.

Precisaríamos também entender como acessar a métrica de IDLE de cada CPU e as filas de processo delas, para assim fazer a migração com métricas que deem suporte para a decisão de qual CPU escolher para cada processo.