Avaliação do desempenho de *Threads* em *user level* utilizando sistema operacional *Linux*

Renato Ramos da Silva ¹
Roberto Sadao Yokoyama ¹

Resumo: A utilização de *threads* em sistemas computacionais é um recurso importante para possibilitar a execução simultânea de diversos trechos de código em um processo. A maneira como o suporte a *threads* é implementado nesses sistemas tem um impacto direto no desempenho. Neste trabalho, é apresentada uma avaliação do desempenho da utilização de *threads user-level* em Linux com as bibliotecas GNU Pth [2], Protothreads [1] e PM2 Marcel [7]. A avaliação de desempenho foi conduzida por meio da técnica de coleta de dados com monitores e as métricas utilizadas foram o tempo de resposta de funções de gerenciamento e sincronização de *threads* para cada biblioteca em operações *CPU bound*, *I/O bound* e simultâneas (*CPU bound* e *I/O bound*).

Abstract: The use of threads in computer systems is an important resource to enable simultaneous implementation of various portions of code in a process. The way that support the threads is implemented in such systems has a direct impact on performance. This work is a performance evaluation of the use of threads in Linux user-level with libraries GNU Pth [2], Protothreads [1] and PM2 Marcel [7]. The performance evaluation was conducted through the technique of collecting data on monitors, and was assessed the response time of service management and synchronization of threads for each library and in operations bound CPU, I/O bound and simultaneous (CPU bound and I / O bound).

1 Introdução

Uma importante característica de um Sistema Operacional (S.O.) é o tempo de resposta que ele retorna ao usuário, dando a sensação de executar diversos processos em paralelo (pseudo-parallelismo em sistemas com um processador), por isso surgiram os S.O. multitarefa, que se caracterizam por serem capazes de rodar múltiplos processos e tiveram sua origem na década de 70 [11].

¹Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - Campus São Carlos
Av. do Trabalhador Sancarlenso, 400 - Centro
Caixa Postal 668 - CEP 13560-970 São Carlos, SP)
{ramos,sadao@icmc.usp.br}

Ao final da década de 80, este conceito foi estendido para dentro do processo, permitindo a existência de múltiplos caminhos de execução dentro de um mesmo programa, este conceito é chamado de *threads*, que pode ser implementado em *user level* ou *kernel level*. Os primeiros S.O.s a implementarem esta característica eram baseados em *microkernel* [11] e até 1991 nenhum S.O. possuía uma biblioteca em *user level* para criação e uso de múltiplas *threads* [3]. Somente a partir de 1996, os principais S.O. passaram a suportar múltiplas *threads* dentro de um processo.

Os S.O.s foram alterados para suportarem as *threads*, passando a ser capaz de guardar o estado da *thread* e do processo em execução. As *threads* exigiram também que novas estruturas fossem adicionadas ao processo, por exemplo, contador do programa, registradores, estado da pilha, etc. [11]. Diversas implementações de *threads* são dependentes de características do S.O., isso possibilita maior ou menor compartilhamento de recursos.

Algumas bibliotecas *user level* tem sido propostas para a programação de *threads* em Linux, porém elas podem não ser adequadas, dependendo da característica da aplicação, mesmo que a utilização de *threads* seja inerente a ela. Pois, não são todas as bibliotecas que implementam as mesmas funções de gerenciamento e recursos de sincronização, elas diferem nos requisitos de *software* e *hardware*, e de maneira geral, comportam-se de forma distinta. E, nesse contexto, a escolha de uma biblioteca adequada torna-se um desafio.

Neste trabalho são definidas diversas métricas que sejam relevantes no desempenho de uma biblioteca de *threads* e avaliar seus tempos de resposta. Considerando que não existem restrições arquiteturais, é necessário efetuar uma análise cuidadosa das características do sistema em desenvolvimento para efetuar a escolha de uma determinada biblioteca. Essa escolha poderá ser baseada nas medidas descritas neste trabalho. Assim, verifica-se a importância de efetuar uma análise quantitativa do desempenho da utilização de cada uma das formas de implementação de *threads user level*.

2 *Threads*

Atualmente, o conceito de *threads* é amplamente pesquisado. Esse interesse por *threads* está associado com o advento de máquinas multiprocessadas e com a facilidade de exprimir tarefas concorrentes. Uma *thread* é usualmente definida como um fluxo de controle no interior de um processo [6]. Stallings [9] define como linhas de execução que, concorrentemente, dividem o mesmo contexto, ou seja, compartilham espaço de endereçamento.

Segundo Oliveira [6], quando um novo processo é criado, o *kernel* do Linux copia os atributos do processo corrente para o que está sendo criado. O Linux, entretanto, prevê uma segunda forma de criação de processos: a clonagem. Um processo clone compartilha os recursos (arquivos abertos, memória virtual, etc.) com o processo original. Quando dois ou

mais processos compartilham as mesmas estruturas, eles atuam como se fossem diferentes *threads* no interior de um único processo.

A principal vantagem do uso de *multithreads* em relação à de um processo advém da facilidade de comunicação e compartilhamento de dados entre as *threads*, uma vez que ambos existem no mesmo espaçamento de endereço de um único processo e compartilham variáveis globais [10]. Por isso, em alguns S.O.s, podem ser mais eficientes ao criar uma nova *thread* ao invés de um novo processo, pois o custo de mudança de contexto de execução de uma *thread* para outra é menor, se comparado à mudança de contexto entre processos [5]. Dessa forma, *Threads* oferecem maior desempenho em ambientes multiprocessados, aumenta a responsividade em interfaces gráficas e possuem um custo de gerenciamento menor que o de processos [9].

A utilização dos recursos de *threads*, normalmente é feita por meio de programação *multithreads*, que permite a programação concorrente e em sistemas com múltiplos processadores ou múltiplos núcleos (*multicore*), sendo possível um paralelismo real [4]. Quando se utiliza na programação *threads* em *user level*, as operações tem custo menor para o S.O., porém se uma *thread* está em uma operação bloqueante (e.x. fazendo Entrada/Saída), todas as *threads* do processo ficam bloqueadas. Por outro lado, a utilização de bibliotecas *kernel level* as chamadas ao *kernel* são mais caras para o S.O., a vantagem é que o *kernel* tem conhecimento das *threads*, e na situação anterior, pode passar a executar outra *thread* do mesmo processo [11].

Existem domínios de problemas que podem se beneficiar do uso de programação *multithread*, por exemplo, processamento de imagens, E/S assíncrono, bancos de dados, etc. Porém, em alguns domínios são discutíveis a utilização da programação *multithread*, por exemplo, nos cálculos seriais. É importante resaltar que, o uso de *threads* impõe diversas dificuldades, tais como, uma programação equivocada de *threads* pode parar a execução de toda a aplicação, pois elas co-existirem no mesmo endereço de um único processo, também podem aparecer questões como condições de disputa, gerando código que funciona na maioria das vezes, porém falha em certas condições. Adicionalmente, a depuração de um programa com *multithreads* é complexa se comparada a um programa convencional, e se o sistema não possui múltiplos processadores pode não haver uma diminuição no tempo de execução do programa ao usar *threads*. Portanto, a programação *multithread* é um recurso valioso, quando nas mãos de um bom programador e utilizada em um contexto apropriado.

As *threads* que executam dentro de um processo são chamados de *threads user-level*. Em linhas gerais, cada *threads user-level* corresponde a uma tarefa a ser executada. Estas *threads* são gerenciadas pela aplicação, sem conhecimento do *kernel* e seu escalonamento é feito por uma parte do código do próprio programa em execução.

3 Bibliotecas Utilizadas

3.1 PTH - GNU Portable Threads

A GNU pth (Pth)[2] é uma biblioteca *user-level* portátil para plataforma *Unix* baseada no padrão POSIX/ANSI-C que contém escalonamento não preemptivo para múltiplas *threads* de execução (*multithreading*) de eventos controlados na aplicações. Todas as *threads* são executadas no mesmo espaço de endereçamento do processo, mas cada *thread* tem seu próprio contador, pilha de execução, máscara de sinal e a variável *errno*(contém código de erro, ou seja, o número de erro do sistema).

O escalonamento de *threads* é feito de forma cooperativa, i.e., as *threads* são gerenciadas por uma prioridade e despachadas por um dispositivo de evento do escalonador não preemptivo. A intenção é que deste modo a portabilidade e a performance seja melhor que o escalonamento preemptivo [2].

3.2 PM2 Marcel

PM2 Marcel [7] é uma biblioteca *multithreading user level* compatível com o padrão POSIX. Está disponível em diversas formas de acordo com a plataforma e as necessidades do programador.

A arquitetura de Marcel foi desenhada para suportar um alto número de *threads* e explorar eficientemente arquiteturas hierárquicas (e.g. *chips* multi-core, máquinas NUMA).

Esta biblioteca atualmente é desenvolvida e mantida por Samuel Thibault. O software é livre e está disponível sob os termos do *GNU General Public License* versão 2. Os seguintes tipos do processador são suportados: X86, X86_64, IA64, Powerpc, Alpha, Sparc, MIPS. E os seguintes tipos de S.O.s são suportados: GNU/Linux, Mac OS X, AIX, OSF, Irix, Solaris, GNU/Hurd, Windows [7].

3.3 Protothreads

Protothreads (PT) [1] são *threads* ditas como "de pouco peso"(*lightweight*), *threads* sem pilhas que possuem um contexto de bloqueio no topo de um dispositivo de evento do sistema, sem o *overhead* de cada pilha por *thread*. Protothreads possuem um bloqueio condicional sobre a função da linguagem C.

A grande vantagem das PT sobre as outras, é que ela é *lightweight*: cada *thread* não precisa de sua própria pilha. Então, todas *threads* são executadas na mesma pilha e a alteração do contexto é feita pela "reempilhação"da pilha. Uma *thread* requer apenas dois *bytes* de memória. Com essas características a biblioteca é indicada para sistemas embarcados [1].

4 Coleta de Dados

As técnicas de aferição são os métodos que fornecem os resultados mais precisos. Dentre as técnicas de aferição, a técnica mais precisa é a coleta de dados, quando o objetivo for tão somente o de avaliação de desempenho através da obtenção direta dos dados no sistema computacional [8]. Entretanto, nem sempre uma coleta de dados em um sistema é uma tarefa trivial. Uma coleta sem certas precauções pode interferir na produção dos resultados, tornando-os imprecisos.

A técnica de coleta de dados pode ser realizada através de dois recursos: os monitores de *hardware* e os monitores de *software*. Neste trabalho foram utilizados os monitores de *software*, que são usados nos casos em que se deseja observar características específicas das *threads*. Informações obtidas pelos monitores de software podem ser muito úteis, por exemplo, para estudos de como algumas características dos S.O.s podem influenciar o desempenho de um sistema (políticas de escalonamento de CPU, de alocação de memória, entre outros) [8].

5 Planejamento de Testes

Foi efetuado um planejamento para a execução dos testes na avaliação. Os seguintes casos de testes foram utilizados:

- Gerenciamento de *Threads*
 - Verificar se as *threads* estão sendo criadas corretamente.
 - Verificar se a função *Join* está sendo executada corretamente.
 - Verificar se a função *Init* está sendo executada corretamente.
- Sincronização
 - Verificar se a função *Mutex lock* e *unlock* estão sendo executadas corretamente.
 - Verificar se as funções de semáforo estão sendo executadas corretamente.
- Módulo *CPU Bound*
 - Verificar a utilização do processador.
 - Verificar se o arquivo foi gravado corretamente.
- Módulo *I/O Bound*

- Verificar a utilização do disco.
 - Verificar se o arquivo foi gravado corretamente.
- Módulo *CPU e I/O Bound*
 - Verificar a utilização do processador e do disco.
 - Verificar se o arquivo foi gravado corretamente.

6 Planejamento da Avaliação

O objetivo da avaliação é verificar o desempenho da utilização das três bibliotecas escolhidas para suporte a *threads user-level*. A principal forma de analisar o desempenho das bibliotecas adotadas foi à análise do tempo despendido para executar determinadas tarefas.

Como a utilização de monitores provoca interferência no sistema, foi verificado as possíveis causas de interferência e as respectivas ações de minimização, como descritas na Tabela 1.

Tabela 1. Possíveis interferências e ações de minimização

Interferência	Ação para minimizar
- Versão do sistema operacional - Configuração do computador (hardware)	- Executar os monitores sempre no mesmo ambiente.
- Outros processos disputando os mesmos recursos.	- Executar em modo de segurança.
- Erros acumulados por truncamentos.	- Usar um tipo de variável para o contador com maior precisão possível.
- Forma de implementação	- Implementar sempre os mesmos algoritmos para cada biblioteca.

A abordagem definida para a avaliação é a execução de cada programa quarenta vezes, obtendo assim uma melhor confiabilidade estatística. Observando-se que cada biblioteca possui o seu código fonte para cada parâmetro avaliado.

O ambiente utilizado para os testes foi um PC com processador Athlon 64 3200+, 2GB de memória RAM, disco rígido IDE 1206 7200 RPM, rodando o sistema operacional Fedora core 5.

As funcionalidades podem ser diferenciadas para cada biblioteca, como funções para gerenciamento de *threads* e para gerenciamento de sincronismo. Por isso, tomou-se o cuidado de verificar a equivalência de cada funcionalidade avaliada.

Para medir o tempo gasto com a execução de cada uma das funções foram criados programas específicos. No gerenciamento de *threads* verificaremos as funções *init*, *create* e *join*. No gerenciamento de sincronização verificaremos as funções sobre *lock mutex*, *unlock mutex*, e *mutex init* para as bibliotecas GNU Pth [2] e PM2 Marcel [7]. A biblioteca PT [1] não possui a funcionalidade *mutex*, por isso foi verificado as funções de semáforos, pois a biblioteca PM2 Marcel oferece as mesmas funcionalidades.

Além das funcionalidades, foram utilizados monitores que visaram verificar o desempenho para operações CPU, IO e CPU/IO *Bound* com as diferentes bibliotecas. As operações *IO bound* são aquelas que acessam intensamente dispositivos de entrada e saída(disco). Já as operações *CPU bound* são aquelas que exigem processamento intenso. IO e CPU *bound* alternados intercalam estas duas classes de operações.

Para automatizar o processo, foi criado um programa utilizando a linguagem *bash script* para executar seqüencialmente cada um dos programas implementados, e variar as entradas de dados para os programas com diferentes números de interações.

7 Resultados

Neste capítulo são descritos os principais resultados obtidos com a execução dos testes com os programas de monitores implementados utilizando as bibliotecas GNU Pth [2], PT [1] e PM2 Marcel [7]. São apresentadas análises estatísticas sobre os dados, e as comparações entre os dados obtidos com os testes.

Na Seção 7.1 são discutidos os dados referentes aos tempos para execução de funções de gerenciamento de *threads*; na Seção 7.2 são discutidos os dados referentes aos tempos para execução de funções para sincronização de *threads*; na Seção 7.3 são discutidos os dados obtidos com a execução de monitores com threads CPU bound; na Seção 7.4 são discutidos os dados obtidos com a execução de monitores com threads I/O bound; na Seção 7.5 são discutidos os dados obtidos com a execução de monitores com threads CPU e I/O bound simultâneas; na Seção 7.6 são discutidos os dados obtidos com a execução de monitores para criação de diferentes números de threads simultâneas.

7.1 Funções de Gerenciamento de threads

Após efetuar a execução dos programas de monitores para verificação dos tempos para executar a função *init*, criação e *join* de *threads* a partir da obtenção dos tempos de execução

em uma amostra de 40 (quarenta) execuções, foi calculada a média e o desvio padrão, em microssegundos, e o intervalo de confiança para cada uma das bibliotecas.

7.1.1 Inicialização de Threads É mostrada na Tabela 2 os resultados utilizando a confiança de 99,95%, sendo possível observar a proporção entre o tempo de execução da função de inicialização pelas *threads* nas três bibliotecas.

Tabela 2. Tempo médio de execução da função *Init*

Biblioteca	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	118,25	4,436	$\pm 1,374$
PT	4,60	0,496	$\pm 0,154$
PM2	11257,15	39,589	$\pm 12,268$

Com a realização dos experimentos, verificou-se que a execução da função *init* da biblioteca PT é bem mais rápida entre as três, sendo em média 25,7 vezes mais rápida que a biblioteca GNU Pth que levava aproximadamente 95,19 vezes menos tempo para executar a função do que a biblioteca PM2.

7.1.2 Criação de Threads É mostrada na Tabela 3 os resultados utilizando a confiança de 99,95%. No gráfico da Figura 1 é possível observar a proporção entre o tempo de criação de *threads* nas três bibliotecas.

Tabela 3. Tempo médio de criação de *threads*

Biblioteca	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	40,73	0,751	$\pm 0,223$
PT	4,60	0,49	$\pm 0,152$
PM2	10,775	0,6197	$\pm 0,192$

Com a realização dos experimentos, verificou-se que a criação de uma *thread* com a biblioteca PT é a mais rápida, sendo em média um pouco mais de 2 (dois) vezes mais rápida que a biblioteca PM2, sendo essa aproximadamente 4 (quatro) vezes mais rápida para a criação de uma *thread* que a biblioteca GNU Pth.

7.1.3 Operação de Join de Threads É mostrado na Tabela 4 os resultados utilizando a confiança de 99,95%. No gráfico da Figura 2 é possível observar a proporção entre o tempo de execução da função *join* nas três bibliotecas.

Figura 1. Tempo médio para criação de threads

Figura 2. Tempo médio de execução da operação *join*

Tabela 4. Tempo médio de execução da operação *join*

Biblioteca	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	57,70	6,156	$\pm 1,908$
PT	4,68	0,474	$\pm 0,147$
PM2	8,1	0,632	$\pm 0,196$

Com a realização dos experimentos, verificou-se que a execução da função *join* de uma thread com a biblioteca PT é ligeiramente mais rápida que a biblioteca PM2, sendo as duas mais rápidas, aproximadamente 8 (oito) vezes mais rápida que a biblioteca GNU Pth.

7.2 Funções de Sincronização de *Threads*

As bibliotecas utilizadas não possuem as mesmas funções de sincronização, assim, somente as funções em comum foram feitos testes. O tratamento para a variável *mutex* existe nas bibliotecas GNU Pth e PM2 Marcel. Já a o tratamento de semáforos está nas bibliotecas PT e PM2 Marcel.

Após efetuar a execução dos programas de monitores para verificação dos tempos para inicialização, operação de *lock* e operação de *unlock* de *Mutex*, e inicialização, da função *wait* e da função *signal* de semáforo de *threads* a partir da obtenção dos tempos de execução em uma amostra de 40 (quarenta) execuções, foi calculada a média e o desvio padrão, em microssegundos e o intervalo de confiança para cada uma das bibliotecas.

7.2.1 Inicialização de *Mutex* É mostrado na Tabela 5 os resultados utilizando a confiança de 99,95%, sendo possível observar a proporção entre o tempo de execução da função de inicialização de *mutex* de *threads* nas duas bibliotecas.

Tabela 5. Tempo médio de execução da operação de inicialização do *mutex*

Biblioteca	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	6,225	0,423	$\pm 0,131$
PM2	4,70	0,464	$\pm 0,144$

Com a realização dos experimentos, verificou-se que a execução da função de inicialização da variável *mutex* com a biblioteca PM2 é a mais rápida que a biblioteca GNU Pth.

7.2.2 Operação de Lock em Mutex É mostrados na Tabela 6 os resultados utilizando a confiança de 99,95%, sendo possível observar a proporção entre o tempo de execução da função de *lock* de *mutex* de *threads* nas duas bibliotecas.

Tabela 6. Tempo médio de execução da operação de *lock* do *mutex*

Biblioteca	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	5,85	0,3613	$\pm 0,112$
PM2	4,825	0,385	$\pm 0,119$

Com a realização dos experimentos, verificou-se que a execução da função de *lock* da variável *mutex* com a biblioteca PM2 é a mais rápida que a biblioteca GNU Pth.

7.2.3 Operação de Unlock em Mutex É mostrada na Tabela 7 os resultados utilizando a confiança de 99,95%, sendo possível observar a proporção entre o tempo de execução da função de *unlock* de *mutex* de *threads* nas duas bibliotecas.

Tabela 7. Tempo médio de execução da operação de *unlock* do *mutex*

Biblioteca	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	5,625	0,49	$\pm 0,152$
PM2	4,75	0,438	$\pm 0,136$

Com a realização dos experimentos, verificou-se que a execução da função de *unlock* da variável *mutex* com a biblioteca PM2 é a mais rápida que a biblioteca GNU Pth.

7.2.4 Inicialização de semáforo É mostrada na Tabela 8 os resultados utilizando a confiança de 99,95%. No gráfico da Figura 3 é possível observar a proporção entre o tempo de execução da função de inicialização de semáforos de *threads* nas duas bibliotecas.

Tabela 8. Tempo médio para execução da função de inicialização do semáforo

Biblioteca	Média	Desvio Padrão	Intervalo de Confiança
PT	4,52	0,50	$\pm 0,156$
PM2	4,65	0,483	$\pm 0,149$

Figura 3. Tempo médio para execução da função de inicialização do semáforo

Com a realização dos experimentos, verificou-se que apesar da biblioteca PT ter uma média melhor onde poderia se considera-la mais rápida, pelo intervalo de confiança, as duas estão com um valor sobreposto. Podemos considerar que elas têm o mesmo desempenho.

7.2.5 Operação *Wait* de semáforo É mostrada na tabela 9 os resultados utilizando a confiança de 99,95%. No gráfico da Figura 4 é possível observar a proporção entre o tempo de execução da função de *wait* de semáforos de *threads* nas duas bibliotecas.

Tabela 9. Tempo médio de execução da operação de *wait* de semáforo

Biblioteca	Média	Desvio Padrão	Intervalo de Confiança
PT	4,55	0,503	$\pm 0,156$
PM2	4,725	0,452	$\pm 0,140$

Com a realização dos experimentos, verificou-se que apesar da biblioteca PT ter uma média um pouco melhor onde poderia se considera-la mais rápida, pelo intervalo de confiança, as duas estão com um valor sobreposto. Podemos considerar que elas tem o mesmo desempenho.

7.2.6 Operação *Signal* de semáforo É mostrada na Tabela 10 os resultados utilizando a confiança de 99,95%. No gráfico da Figura 5 é possível observar a proporção entre o tempo de execução da função de *signal* de semáforos de *threads* nas duas bibliotecas.

Figura 4. Tempo médio de execução da operação de *wait* de semáforo

Figura 5. Tempo médio de execução da operação de *signal* de semáforo

Tabela 10. Tempo médio de execução da operação de *signal* de semáforo

Biblioteca	Média	Desvio Padrão	Intervalo de Confiança
PT	4,575	0,50	$\pm 0,155$
PM2	6,1	0,441	$\pm 0,136$

Com a realização dos experimentos, verificou-se que em média, a execução da função de *signal* para semáforo com a biblioteca PT é a mais rápida que a biblioteca PM2.

7.3 Execução de *threads CPU bound*

Os testes com *threads CPU bound* envolveram a execução de duas *threads* simultâneas, em cada programa, que efetuavam diferentes números de iterações com operações envolvendo números de ponto flutuante. Foram obtidas amostras de 40 (quarenta) execuções, para 8 M, 16 M, 32 M, 64 M, 128 M e 256 M iterações, e cada uma delas foi calculada a média e o desvio padrão, em segundos, e o intervalo de confiança para cada uma das bibliotecas.

É mostrado na Tabela 11 os valores das médias de tempo, o desvio padrão e o intervalo de confiança, em segundos, para cada número de iterações nas três bibliotecas. Esses valores também são ilustrados pelo gráfico na Figura 6.

Figura 6. Tempo médio para execução de *threads CPU bound*

Com a realização dos experimentos, verificou-se que a execução do processo CPU

Tabela 11. Tempo de execução de *threads CPU bound*

Bibl.	Número de Interações	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	8 M	0,16331	0,00212	$\pm 0,00066$
	16 M	0,32551	0,00077	$\pm 0,00024$
	32 M	0,65068	0,00092	$\pm 0,00028$
	64 M	1,30138	0,00106	$\pm 0,00033$
	128 M	2,60238	0,00152	$\pm 0,00047$
	256 M	5,20454	0,00280	$\pm 0,00087$
PT	8 M	0,16405	0,02609	$\pm 0,00808$
	16 M	0,32748	0,05100	$\pm 0,01600$
	32 M	0,65046	0,07300	$\pm 0,02300$
	64 M	1,27746	0,00800	$\pm 0,00200$
	128 M	2,56067	0,01493	$\pm 0,00463$
	256 M	5,11235	0,02775	$\pm 0,00860$
PM2	8 M	0,06416	0,00050	$\pm 0,00016$
	16 M	0,12822	0,00052	$\pm 0,00016$
	32 M	0,25631	0,00050	$\pm 0,00020$
	64 M	0,51232	0,00058	$\pm 0,00018$
	128 M	1,02466	0,00088	$\pm 0,00027$
	256 M	2,04887	0,00100	$\pm 0,00031$

com duas *thread* utilizando a biblioteca PM2 é mais rápida que as demais bibliotecas. Pelo gráfico, pode-se notar uma curva de crescimento mais suave para a biblioteca PM2, denotando-se por valores menores para número de iterações elevadas.

7.4 Execução de *threads I/O bound*

Os testes com *threads I/O bound* envolveram a execução de duas *threads* simultâneas, em cada programa, que efetuavam diferentes números de iterações com operações escrita de dados em disco. Foram obtidas amostras de 40 (quarenta) execuções, para 8 KB, 16 KB, 32 KB, 64 KB, 128 KB e 256 KB de dados, e para cada uma delas foi calculada a média para cada uma das bibliotecas.

Na Tabela 12 são mostrados os valores das médias de tempo, o desvio padrão e o intervalo de confiança, em segundos, para cada número de iterações nas três bibliotecas. Esses valores também são ilustrados pelo gráfico na Figura 7.

Com a realização dos experimentos, verificou-se que a execução do processo CPU com duas *thread* utilizando a biblioteca PM2 é mais rápida que as demais bibliotecas. Pelo gráfico, pode-se notar uma curva de crescimento mais suave para a biblioteca PM2, denotando-se por valores menores para número de iterações elevadas.

Tabela 12. Tempo de execução de *threads I/O bound*

Bibl.	Número de Interações	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	8 M	0,05754	0,00406	$\pm 0,00126$
	16 M	0,09871	0,00427	$\pm 0,00133$
	32 M	0,19723	0,00756	$\pm 0,00235$
	64 M	0,39411	0,01484	$\pm 0,00460$
	128 M	0,78096	0,02821	$\pm 0,00875$
	256 M	1,57472	0,05734	$\pm 0,01777$
PT	8 M	0,05948	0,00666	$\pm 0,00206$
	16 M	0,09999	0,01366	$\pm 0,00423$
	32 M	0,20062	0,02752	$\pm 0,00853$
	64 M	0,39173	0,01421	$\pm 0,00440$
	128 M	0,78185	0,02452	$\pm 0,00760$
	256 M	1,56447	0,05152	$\pm 0,01597$
PM2	8 M	0,04929	0,00201	$\pm 0,00062$
	16 M	0,09844	0,00359	$\pm 0,00111$
	32 M	0,19757	0,00807	$\pm 0,00250$
	64 M	0,39584	0,01749	$\pm 0,00542$
	128 M	0,79057	0,03525	$\pm 0,01092$
	256 M	1,57519	0,07227	$\pm 0,02239$

Figura 7. Tempo médio para execução de *threads CPU bound*

7.5 Execução de *threads CPU* e *I/O bound*

Os testes com *threads CPU* e *I/O bound* simultaneamente envolveram a execução de duas *threads* simultâneas, em cada programa, de forma que uma *thread* executasse operações

de escrita em disco, e outra executasse operações de ponto flutuante. Para cada 1000 iterações de operações de ponto flutuante, era efetuada a escrita de um *byte*. Foram obtidas amostras de 40 (quarenta) execuções, para 8 KB, 16 KB, 32 KB, 64 KB, 128 KB e 256 KB de dados, e para cada uma delas foi calculada a média para cada uma das bibliotecas.

Na Tabela 13 são mostrados os valores das médias de tempo, o desvio padrão e o intervalo de confiança, em segundos, para cada número de iterações nas três bibliotecas. Esses valores também são ilustrados pelo gráfico na Figura 8.

Tabela 13. Tempo de execução de *threads CPU e I/O bound*

Bibl.	Número de Interações	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	8 M	0,10452	0,00135	$\pm 0,00042$
	16 M	0,20821	0,00222	$\pm 0,00688$
	32 M	0,41583	0,00416	$\pm 0,01291$
	64 M	0,83115	0,00783	$\pm 0,02426$
	128 M	1,63308	0,01586	$\pm 0,04916$
	256 M	3,32119	0,03148	$\pm 0,09756$
PT	8 M	0,02523	0,00325	$\pm 0,00101$
	16 M	0,04999	0,00673	$\pm 0,00209$
	32 M	0,09824	0,00376	$\pm 0,00117$
	64 M	0,19577	0,00676	$\pm 0,00210$
	128 M	0,39222	0,01410	$\pm 0,00437$
	256 M	0,78661	0,03080	$\pm 0,00957$
PM2	8 M	0,05648	0,00096	$\pm 0,00030$
	16 M	0,11317	0,00195	$\pm 0,00061$
	32 M	0,22701	0,00457	$\pm 0,00142$
	64 M	0,45353	0,00897	$\pm 0,00278$
	128 M	0,90578	0,01945	$\pm 0,00603$
	256 M	1,80863	0,03121	$\pm 0,00967$

Com a realização dos experimentos, verificou-se que o tempo para a execução do programa de *threads* de *CPU* e *I/O bound* com duas *threads* possui o mesmo desempenho para todas as três bibliotecas utilizadas.

7.6 Execução de *threads CPU e I/O bound*

Os testes com diferentes números de *threads* simultâneas envolveu a execução de programas equivalentes implementados nas três bibliotecas, variando o número de *threads* simultâneas para a criação de *threads*.

Na Tabela 14 são mostrados os valores das médias de tempo em microssegundos e o desvio padrão para cada número de iterações. Esses valores também são ilustrados pelo

Figura 8. Tempo médio para execução de *threads CPU bound*

gráfico na Figura 9.

Tabela 14. Tempo médio para criação de *N threads*

Bibl.	Número de threads	Média	Desvio Padrão	Intervalo de Confiança
GNU Pth	8	218,025	36,0637	± 11,17605
	16	405,875	11,6138	± 3,59910
	32	803,325	14,4743	± 4,48557
	64	1605,350	36,0758	± 11,17980
	128	3169,850	41,4361	± 12,84095
	256	6346,325	84,3355	± 26,13537
PT	8	37,825	7,6924	± 2,38387
	16	72,975	1,5104	± 0,46807
	32	146,250	4,5107	± 1,39785
	64	295,300	10,1708	± 3,15192
	128	587,325	13,5199	± 4,18980
	256	1183,000	26,4730	± 8,20392
PM2	8	44,025	1,8740	± 0,58077
	16	83,275	2,5317	± 0,78460
	32	158,750	3,90101	± 1,20891
	64	311,450	5,64755	± 1,75016
	128	619,100	5,45288	± 1,68982
	256	1229,925	7,10502	± 2,20183

Com a realização dos experimentos, verificou-se que o tempo para a execução do programa de *threads* de *CPU* e *I/O bound* com duas *threads* possui o mesmo desempenho para todas as três bibliotecas utilizadas.

Figura 9. Tempo médio para criação de *N threads*

8 Conclusão

A utilização de *threads* para permitir a execução simultânea de diferentes tarefas em um mesmo processo tem sido bastante difundida. A execução dessas *threads* pode ser gerenciada pelo *kernel* do sistema operacional ou pelo próprio processo.

A escolha pela abordagem de *threads user level* deve ser feita de forma cuidadosa, considerando as questões de desempenho envolvidas, a portabilidade e restrições arquiteturais.

Neste trabalho, foram apresentadas algumas das principais características das bibliotecas GNU Pth [2], PT [1] e PM2 Marcel [7], que oferecem suporte à utilização de *threads user-level*.

Uma avaliação de desempenho foi efetuada por meio da técnica de monitores visando verificar o desempenho de programas similares utilizando as três bibliotecas em tarefas de gerenciamento de *threads*, sincronização de *threads*, programas *CPU bound*, *I/O bound*, programas combinando tarefas *CPU bound* e *I/O bound* simultaneamente, e programas com diferentes números de *threads* simultâneas.

De acordo com os resultados da avaliação, de forma geral, as operações de gerenciamento de *threads*, que envolvem a execução da função *init*, criação e *join*, apresentou melhor desempenho com a biblioteca *PT*, como era esperado, devido ao fato da forma como essa biblioteca é implementada.

Quanto as operações de sincronização de *Pth* e *PM2*, que envolviam a inicialização, *lock* e *unlock* de mutexes, apresentou melhor desempenho com a biblioteca *PM2*.

Quanto as operações de sincronização PT e PM2, que envolviam a inicialização, *wait* e *signal* de semáforos, não apresentaram diferença sobre a função de inicialização. A biblioteca PT apresentou melhor desempenho sobre a função *signal*. Já sobre a função *wait* não é possível fazer uma definição sobre os resultados, pois o intervalo de confiança está sobreposto sem englobar o valor médio de ambos.

Nas avaliações com programas com *threads CPU bound*, a biblioteca PM2 apresentou melhor desempenho, considerando que as demais tiveram o mesmo desempenho.

Nas avaliações de programas com *threads I/O bound*, as bibliotecas mostraram, na média, o mesmo desempenho.

Nas avaliações de programas com *threads CPU e I/O bound*, a biblioteca PT apresentou melhor desempenho sobre as demais bibliotecas. A biblioteca PM2 mostrou um desempenho melhor que a GNU Pth.

Na avaliação do desempenho da execução de programas com diferentes números de *threads*, verificou-se que a PT obteve melhor desempenho. Com resultado um pouco pior está a biblioteca PM2, sendo a biblioteca GNU Pth com resultados bem abaixo das outras duas. Isso era esperado, pois reflete o que acontece para a criação de uma *thread*.

De forma geral, pode-se concluir que a escolha de uma biblioteca de *threads user level* deve ser feita de acordo com a aplicação que se deseja desenvolver. A biblioteca GNU Pth apresenta a vantagem por possuir diversas funções já implementadas. Se a aplicação requer operações alternadas de *CPU* e *IO bound* e/ou baixo consumo de memória, a indicada é a biblioteca PT. Para sistemas que utilizam muito mais a *CPU* do que operações de *IO*, ou seja, operações *CPU bound*, é indicado a utilização da biblioteca PM2.

Neste trabalho, não foi possível efetuar avaliações mais específicas quanto ao tempo para chaveamento de *threads*. Assim, como trabalho futuro, apontamos o estudo da arquitetura das bibliotecas apresentadas, bem como, a implementação de uma biblioteca que contenham as principais características das bibliotecas apresentadas.

9 Agradecimentos

Os autores gostariam de agradecer a Fapesp, CNPq e CAPES pelo apoio recebido.

Referências

- [1] A. Dunkels. Protothreads - lightweight, stackless threads in c, 2005. Disponível em: <http://www.sics.se/~adam/pt/>. Acesso em junho/2007.

- [2] GNU. Gnu portable threads, 2006. Disponível online em <http://www.gnu.org/software/pth/>. Acesso em junho/2007.
- [3] B. Lewis and D. J. Berg. Pthreads primer: A guide to multithreaded programming, 1996.
- [4] R. Love. Linux kernel development, 2005.
- [5] N. Matthew and R. Stones. Beginning linux programming, 1996.
- [6] A. S.; TOSCANI S. S. OLIVEIRA, R. S.; CARISSIMI. Sistemas operacionais e programação concorrente., 2003.
- [7] PM2. Marcel: A posix-compliant thread library for hierarchical multiprocessor machines, 2006. Disponível em: <http://runtime.futurs.inria.fr/marcel/index.php>. Acesso em junho/2007.
- [8] R. H. C; Francês C. L. Santana, M. J.; Santana. Avaliação e análise de desempenho de sistemas computacionais: Técnicas e ferramentas, 2007.
- [9] W. STALLINGS. Operating systems: Internals and design principles, 2005.
- [10] S. A. Stevens, W. R.; Rago. Advanced programming in the unix environment, 2005.
- [11] A. S. Tanenbaum and A. S. Woodhull. Operating systems: Design and implementation, 1997.