

Introdução ao Processamento Paralelo e ao Uso de Clusters de Workstations em Sistemas GNU/LINUX

Parte II: Processos e Threads*

Bueno A.D. (andre@lmpt.ufsc.br)
Laboratório de Meios Porosos e Propriedades Termofísicas - LMPT
Versão 0.2

12 de novembro de 2002

Resumo

Os conceitos básicos de processamento paralelo foram apresentados no artigo [*Introdução ao Processamento Paralelo e ao Uso de Clusters, Parte I: Filosofia*, Bueno A.D. 2002]. O termo threads, refere-se a tarefas leves, que consomem menos recursos computacionais que um processo, e que são utilizadas de forma eficiente para a execução de processamento paralelo em sistemas multi-processados. No presente artigo apresenta-se de forma mais detalhada os conceitos úteis ao aprendizado da programação paralela utilizando os mecanismos de multi-processing e multi-threading. A estrutura, as características e as diferenças entre processos e threads. Os estados assumidos pelos processos e threads. Que funções da linguagem C você deve conhecer para desenvolver programas usando threads, um roteiro básico e exemplos. Finalmente, apresenta-se a biblioteca common C++, uma biblioteca com classes prontas que facilitam o desenvolvimento de programas multi-threading.

Sumário

1	Introdução	2
2	Objetivo específico	2
3	Processos	2
3.1	Introdução aos processos	2
3.2	Estrutura e características de um processo	2
3.3	Estados de um processo	2
3.4	Processos sincronizados, recursos, dados	2
3.5	Comunicação entre processos	3
4	Exemplo de código comum	3
5	Exemplo de código usando múltiplos procesos	3

6	Threads	5
6.1	Introdução as threads	5
6.2	Estrutura e características de uma thread	5
6.3	Estados de uma thread	6
6.4	Tipos de threads	6
6.5	Semelhanças e diferenças entre threads e processos	6
6.6	Estado de um processo com threads	6
6.7	Prioridades	6
7	Cooperação e sincronização	6
7.1	Categorias de sincronização	6
7.2	Sincronização com mutex	6
7.3	Sincronização com semáforos	7
7.4	Sincronização com variáveis condicionais	7
8	Funções básicas para criar threads	7
8.1	Para criar uma thread	7
8.2	Para recuperar o resultado de uma thread	8
8.3	Para encerrar uma thread	8
8.4	Para deixar a thread destacada	8
8.5	Para cancelar uma thread	8
8.6	Para travar um bloco de código que acessa memória compartilhada (mutex)	9
8.7	Para realizar um delay em uma única thread	9
9	Roteiro básico para desenvolvimento de um programa com threads	9
10	Exemplo de código usando múltiplas threads	9
11	Common C++	10
11.1	O que é e onde encontrar ?	10
11.2	As classes Thread, Mutex e Conditional da biblioteca common C++	10
11.3	Exemplos de código com threads usando a biblioteca de classes common C++	11
12	Comparação dos tempos de processamento	12
13	Conclusões	13

*Este documento pode ser livremente copiado, segundo as regras GPL. Este artigo é uma seqüência do artigo *Introdução ao Processamento Paralelo e ao Uso de Clusters, Parte I: Filosofia*, Bueno A.D. 2002, <http://www.lmpt.ufsc.br/~protect~andre/Artigos/118-IntroducaoProgramacaoParalelaECluster-P1.pdf>

1 Introdução

As vantagens e formas de uso da programação paralela foram descritos no artigo [1]. Apresenta-se aqui os conceitos de processos e de threads, sua estrutura, característica e estados. Quais as características e diferenças entre as threads e os processos.

Apresenta-se exemplos práticos de uso de processos e threads em C e C++.

Segundo [10], o processamento paralelo com uso de threads é de mais alto nível que os mecanismos de troca de mensagens. Também tem um tempo de latência menor.

2 Objetivo específico

Descrever os conceitos básicos para programação utilizando-se processos e threads em sistemas GNU/Linux usando a linguagem C/C++.

3 Processos

3.1 Introdução aos processos

Os computadores e os sistemas operacionais modernos suportam multitasking (múltiplos processos sendo executados simultaneamente). Se o computador tem mais de um processador, o sistema operacional divide os processos entre os processadores.

No GNU/Linux e nas variantes do Unix, um processo pode ser clonado com a função `fork()`.

3.2 Estrutura e características de um processo

Um **processo** contém um programa sendo executado e um conjunto de recursos alocados. A estrutura de um processo é apresentada na Tabela 1.

Tabela 1: Estrutura de um processo e de uma thread.

Ítem	processo	thread
Nome	X	X
ID Identificador	X	X
PID- Identificador do pai (parent id)	X	
Group ID	X	
Real user ID	X	
Grupo suplementar ID	X	
Module_handle	X	
Máscara de sinais	X	
Linha de comando	X	
Pilha	X	X
Recursos	X	
Nome da cpu que esta rodando	X	
Prioridade	X	X
Tempo de processamento	X	
Máscara de arquivos (permissões)	X	
Variáveis do meio ambiente	X	

Pode-se listar as principais características de um processo:

- Um processo é autocontido (independente).
- Cada processo tem seu bloco de memória.
- Um processo pode criar outros processos, dito filhos. Quem cria é o pai (parent), quem é criado é o filho (child).
- Os processos pai e filho são independentes.
- O processo filha herda algumas propriedades do processo pai (nome do pai, prioridades, agendamentos, descritores de arquivo). Observe que o custo de criação de um processo é elevado, pois o sistema operacional deve clonar todas as informações do processo.
- O processo filho (criado no tempo t) pode usar os recursos do pai que já foram criados. Se depois de criar o processo filho, o processo pai criar novos recursos, estes não são compartilhados com o filho.
- O processo pai pode alterar a prioridade ou matar um processo filho.
- O processo filho é encerrado quando retorna o valor zero (0) para o pai.
- Se o processo filho apresenta problemas (termina mas não informa o pai), o mesmo se transforma em um zombie.
- Um processo destacado é um processo que não herda nada de seu pai e que é utilizado para realização de tarefas em background (ex: daemons de impressão).

3.3 Estados de um processo

Um processo pode estar num dos seguintes estados (veja Figura 1):

idle/new: Processo recém criado (sendo preparado).

ready: Processo sendo carregado.

standby: Cada processador tem um processo em standby (o próximo a ser executado).

running: O processo esta rodando (ativo).

blocked: Esperando evento externo (entrada do usuário) (inativo).

suspended-blocked: Processo suspenso.

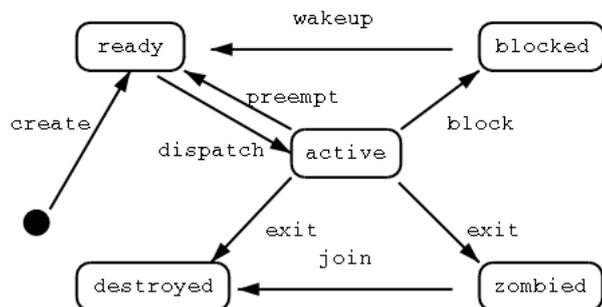
zombied: O processo acabou mas não informou seu pai.

done-terminated: O processo foi encerrado e os recursos liberados.

3.4 Processos sincronizados, recursos, dados

Processos sincronizados e não sincronizados: Quando o processo pai espera o encerramento do processo filho, ele tem uma execução sincronizada. Quando os processos rodam independentemente eles são não sincronizados.

Figura 1: Estados de um processo.



Recursos: Recursos são dispositivos utilizados pelo processo, pode-se citar: HD, CD, impressora, monitor, arquivos de disco, programas, bibliotecas, placa de som, placa de vídeo, entre outros.

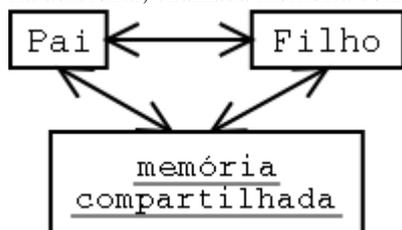
Dados: Arquivos, objetos, variáveis de ambiente, variáveis globais, memória compartilhada entre processos.

3.5 Comunicação entre processos

A comunicação entre processos pode ser realizada utilizando-se variáveis de ambiente, pipes, ou memória compartilhada (shared memory).

Quando um processo é clonado, o processo filho recebe cópias das variáveis do pai, podendo-se desta forma compartilhar as variáveis já criadas.

O processo pai e filho podem compartilhar um bloco de memória adicional, chamada memória compartilhada.



Comunicação entre processos usando pipes: Um pipe é um canal de comunicação entre processos, em que estruturas de dados podem ser acessadas num sistema sequencial. Um pipe pode ser anônimo ou nomeado. Um pipe anônimo é utilizado para comunicação entre o processo pai e filho, e é destruído quando o processo é destruído. Um pipe nomeado permanece vivo mesmo após a finalização do processo, e é utilizado para conectar processos não relacionados. O exemplo 5 mostra o uso de processos e pipes.

4 Exemplo de código comum¹

Apresenta-se a seguir um pequeno código em C, que determina o valor de pi e que será desenvolvido posteriormente utilizando processos e threads.

Listing 1: Determinação de pi.

```

//Inclue bibliotecas de C
#include <stdlib.h>
#include <stdio.h>

//Função main
int main(int argc, char *argv[])
{
    register double width, sum;
    register int intervals, i; //Número de intervalos
    intervals = atoi(argv[1]);
    width = 1.0 / intervals; //largura
    sum = 0;
    for (i=0; i<intervals; ++i)
    {
        register double x = (i + 0.5) * width;
        sum += 4.0 / (1.0 + x * x);
    }
    sum *= width;

    //Mostra o resultado
    printf ( "Estimation_of_pi_is_%f\n", sum);
    return(0);
}
  
```

5 Exemplo de código usando múltiplos procesos

Apresenta-se a seguir um pequeno código que cria um processo filho usando a função fork. A seguir troca dados entre os dois processos usando streams e pipes².

A Figura ilustra a clonagem de um processo com fork. Observe que após a clonagem, a única diferença entre os dois processos é o valor da variável retornada pela função fork. Para o processo pai o retorno (Pid) é igual a zero, e para o filho o Pid é diferente de 0.

Listing 2: Usando múltiplos processos com fork.

```

#include <unistd.h>
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <fstream>

using namespace std;

//Função main
int main (void)
{
    int Fd[2]; //identificador
    pipe (Fd); //cria pipe
    cout << "Fd[0]=_" << Fd[0] << endl;
    cout << "Fd[1]=_" << Fd[1] << endl;

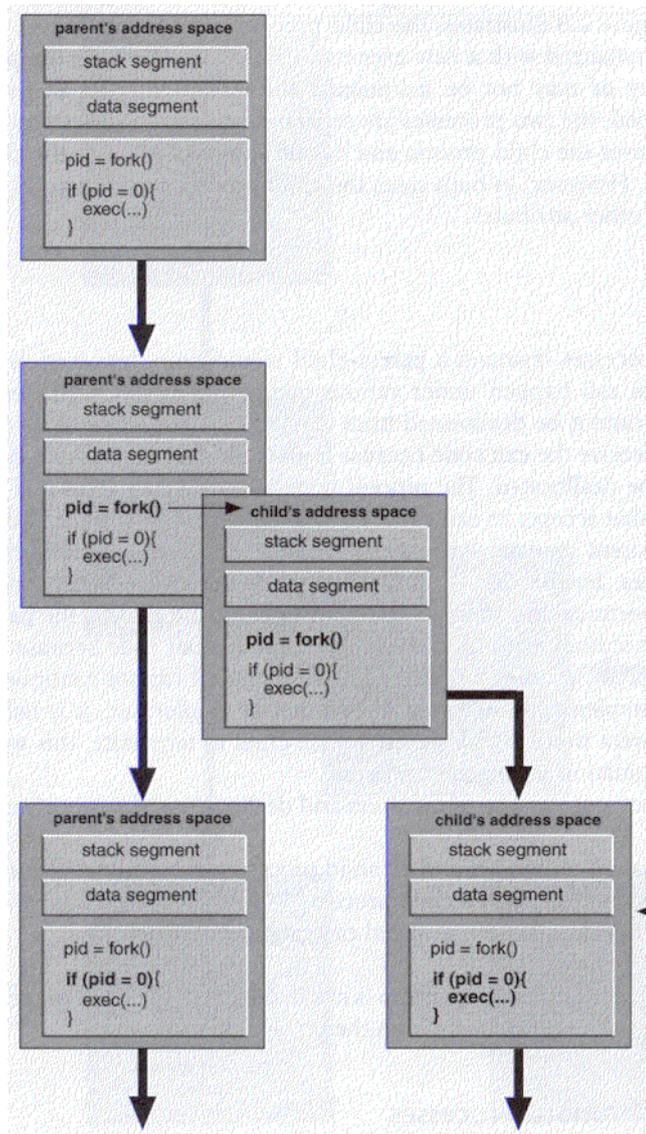
    ifstream In; //Cria stream leitura
    ofstream Out; //Cria stream escrita

    int Pid; //id do processo
    Pid = fork (); //cria processo filho
    //-----
    //se for o processo pai Pid == 0
    //-----
    if (Pid == 0)
    {
        cout << "pid_pai=_" << Pid << endl;
    }
  
```

¹Material extraído da referencia [C++ Programming - HOW-TO].

²Testado no RedHat 7.3, gcc 2.96.

Figura 2: Clonagem de um processo com fork, [5].



```

double entrada_double = 1.2;
int entrada_int = 1;
cout << "-----Valor_inicial_" << endl;
cout << "entrada_double=_ " << entrada_double <<
endl;
cout << "entrada_int=_ " << entrada_int << endl;

close (Fd[1]); //fecha pipe
In.attach (Fd[0]); //conecta para leitura
//lê valores
In >> entrada_double >> entrada_int;
cout << "-----Valor_final_" << endl;
cout << "entrada_double=_ " << entrada_double <<
endl;
cout << "entrada_int=_ " << entrada_int << endl;
In.close (); //fecha conexão
}
//-----
//se for o processo filho
//-----
else

```

```

{
cout << "pid_filho=_ " << Pid << endl;
double saida_double = 2.4;
int saida_int = 11;
close (Fd[0]); //fecha o pipe
Out.attach (Fd[1]); //conecta para escrita
Out << saida_double << " " << saida_int << endl
; //escreve
Out.close (); //fecha conexão
}

return 0;
}
/*
Para compilar:
=====
g++ processos2.cpp -o processos2
*/

```

Apresenta-se a seguir um pequeno código em C++, que determina o valor de pi usando multi-processing.

Listing 3: Determinação de pi usando multi-processos.

```

#include <unistd.h>
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <fstream>

using namespace std;

double process(int iproc,double intervalos,int nproc);

//-----Função main
int main (int argc, char* argv[])
{
//Obtém o intervalo
if( argc < 3 )
{
cout <<"Uso:_ " << argv[0] << " "
numero_de_intervalos_numero_de_processos"
<< endl ;
cout <<"Exemplo:_ " << argv[0] << " _1000_4_" << endl
;
return -1;
}
double intervalos = atof (argv[1]);
int nproc = atoi (argv[2]);
cout << argv[0] << " _intervalos_" << intervalos << " "
<< "nproc=" << nproc << endl;

//Criando o pipe e as streams
int Fd[2]; //identificador
pipe (Fd); //cria pipe
ifstream In;
ofstream Out;

//Criando processos
//id do processo
int iproc = 0 ;
//O processo pai ( Pid = 0 ) vai criar (nproc-1)
processos
int Pid = 0;
for (int n = 0; n < (nproc - 1); n++)
{
if( Pid == 0) //se for o processo pai
{
iproc++; //incrementa o iproc
Pid = fork (); //cria processo filho
}
}

```

```

//-----
//se for o processo pai Pid == 0
//-----
if (Pid == 0)
{
    double pi = 0;
    double extern_pi = 0;
    iproc = 0;
    pi = process ( iproc, intervalos, nproc);
    cout << "Pid_pai=_ " << Pid << " _iproc=_ "<<
        iproc << endl;
    cout << "pi_pai=_ " << pi << endl;

    close (Fd[1]); //fecha pipe
    In.attach (Fd[0]); //conecta para leitura
    for (int n = 0; n < (nproc - 1); n++)
    {
        In >> extern_pi; //lê valores
        pi += extern_pi;
    }
    In.close (); //fecha conexão

    cout << "Valor_estimado_de_pi=_ " << pi << endl;
}
//-----
//se for o processo filho
//-----
else
{
    double pi = 0;
    cout << "Pid_filho=_ " << Pid << " _iproc=_ "<<
        iproc << endl;
    pi = process ( iproc, intervalos, nproc);
    cout << "pi_filho=_ " << pi << endl;

    close (Fd[0]); //fecha o pipe
    Out.attach (Fd[1]); //conecta para escrita
    Out << pi << endl; //escreve
    Out.close (); //fecha conexão
}
return 0;
}

//-----Função process
double process (int iproc, double intervalos, int nproc
)
{
    register double width , localsum;
    width = 1.0 / intervalos;
    localsum = 0;
    for ( int i = iproc; i < intervalos; i += nproc )
    {
        register double x = (i + 0.5) * width;
        localsum += 4.0 / (1.0 + x * x);
    }

    localsum *= width;
    return *(new double(localsum));
}

/*
Informações:
=====
O processo pai tem Pid=0, setado na saída da função
fork.
Cada processo filho tem um Pid != 0.

Para identificar de forma mais clara os processos, foi
criada
a variavel iproc, que assume o valor 0 para o processo
pai
e 1,2,3... sucessivamente para os filhos.

```

```

Saída:
=====
[andre@mercurio Processos]$ ./a.out 1000 4
./a.out intervalos =1000 nproc=4
Fd[0]= 3
Fd[1]= 4
Pid filho = 12545 iproc = 1
pi filho = 0.785648
Pid filho = 12546 iproc = 2
Pid filho = 12547 iproc = 3
pi filho = 0.784648
Pid pai = 0 iproc = 0
pi pai = 0.786148
pi filho = 0.785148
Valor estimado de pi = 3.1416

Para compilar:
=====
g++ processos4.cpp -o processos4
*/

```

6 Threads

6.1 Introdução as threads

O que são threads ? Threads são múltiplos caminhos de execução que rodam concorrentemente na memória compartilhada e que compartilham os mesmos recursos e sinais do processo pai. Uma thread é um processo simplificado, mais leve ou “light”, custa pouco para o sistema operacional, sendo fácil de criar, manter e gerenciar.

Porque usar threads ? Em poucas palavras é o pacote definitivo para o desenvolvimento de programação em larga escala no Linux, [3]. O padrão é o POSIX 1003.1C, o mesmo é compatível com outros sistemas operacionais como o Windows.

Casos em que o uso de threads é interessante : Para salvar arquivos em disco. Quando a interface gráfica é pesada. Quando existem comunicações pela internet. Quando existem cálculos pesados.

O que você precisa ? De um sistema operacional que suporte *PThreads* e uma biblioteca de acesso as funções *PThreads*.

Veja uma introdução sobre threads em http://centaurus.cs.umass.edu/~wagner/threads_html/tutorial.html, e um conjunto de links em <http://pauillac.inria.fr/~xleroy/linuxthreads/>. Você encontra exemplos e descrições adicionais nas referências [2, 5, 6, 7, 8, 9, 4].

6.2 Estrutura e características de uma thread

A estrutura de uma thread é apresentada na Tabela 1. Veja que o número de itens de uma thread é bem menor que o de um processo. Dentre as características das threads cabe destacar:

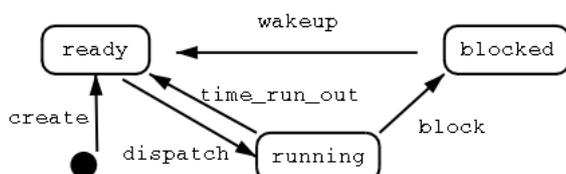
- Uma thread tem um contexto, um id, uma pilha, registradores e prioridades (veja Tabela 1).

- Todos os recursos são controlados pelo processo e compartilhados pelas diversas threads.
- A memória do processo é compartilhada pelas threads.
- Uma thread pode encerrar o processo matando a thread primária. A thread primária é na realidade o processo inicial, o processo que criou as threads, é criada automaticamente pelo sistema operacional. Isto significa que todo processo tem pelo menos uma thread
- Mudanças na thread primária afetam todas as threads.
- Ao mudar a prioridade de um processo, altera-se a prioridade de todas as threads.
- Uma thread destacada (detached) é semelhante a um processo destacado. Não precisa saber quem a criou e se torna independente.

6.3 Estados de uma thread

A Figura 3 mostra os possíveis estados de uma thread. Compare com a Figura 1 e veja que o número de estados é menor.

Figura 3: Estados de uma thread.



6.4 Tipos de threads

De uma maneira geral, existem diferentes tipos de threads, dentre as quais cabe destacar.

Dormir e disparar: Uma thread que fica dormindo, esperando o envio de uma mensagem. Quando recebe a mensagem disparada, isto é, executa a tarefa.

Antecipação de trabalho: Em alguns casos, o algoritmo deve optar entre duas opções e seguir o seu processamento. Algoritmos modernos tem utilizado threads para antecipar as execuções quando existe uma opção, assim, quando o programa chegar no ponto em que deve decidir qual caminho percorrer, o resultado já estará esperando.

6.5 Semelhanças e diferenças entre threads e processos

Como visto na Tabela 1, um processo contém um conjunto grande de dados, tendo um custo elevado para sua construção e destruição. Já uma thread é bem mais leve. Outra diferença importante é que um processo tem sua própria memória, já uma thread compartilha a memória com o processo que a criou e com as demais threads.

6.6 Estado de um processo com threads

Se o processo tem apenas a thread primária, o estado do processo é o estado da thread.

Se o processo tem mais de uma thread, então, se uma thread estiver ativa o processo esta ativo. Se todas as threads estiverem inativas o processo estará inativo.

6.7 Prioridades

A thread de mais alta prioridade é executada antes das demais. Se um conjunto de threads tem a mesma prioridade, o sistema operacional trata as mesmas utilizando um fila do tipo FIFO (first in first off).

Alguns sistemas usam sistemas dinâmicos para realizar este controle de prioridade.

Uma prioridade é dividida em classe e valor. A classe pode ser crítica, alta, regular e idle. O valor é um número entre 0 e 31.

Observe que se uma thread de prioridade alta, precisa de uma variável fornecida por uma thread de prioridade baixa, as prioridades podem ser invertidas.

7 Cooperação e sincronização

Em diversos momentos você vai ter de usar mecanismos de cooperação e de sincronização.

Um exemplo de cooperação ocorre quando duas ou mais threads tentam acessar um mesmo recurso, você vai precisar de um mecanismo de cooperação entre as duas threads.

A sincronização ocorre, por exemplo, quando duas threads precisam rodar simultaneamente para mostrar partes diferentes de uma imagem na tela.

7.1 Categorias de sincronização

Existem diferentes categorias de sincronização, dentre as quais pode-se destacar:

Sincronização de dados: Threads concorrendo no acesso de uma variável compartilhada ou a arquivos compartilhados.

Sincronização de hardware: Acesso compartilhado a tela, impressora.

Sincronização de tarefas: Condições lógicas.

7.2 Sincronização com mutex

Para que você possa impedir o acesso a uma variável compartilhada por mais de uma thread, foi desenvolvido o conceito de mutexes ou mutex (mutual exclusion), uma espécie de trava que é utilizada para impedir o acesso simultâneo a uma determinada porção do código. Ou seja, um mutex é um mecanismo simples de controle de recursos.

Uma variável mutex pode estar travada (lock) ou destravada (unlock). Pode ou não ter um dono (owned, unowned), pode ou não ter um nome (named, unnamed).

7.3 Sincronização com semáforos

Conjunto de variáveis introduzidas por E.W.Dijkstra. Um semáforo é um número inteiro positivo. É uma variável protegida.

Sobre o semáforo *s*, atuam duas funções. A função *P(s)* espera até que o valor de *s* seja maior que 0 e a seguir decrementa o valor de *s*. A função *V(s)* incrementa o valor de *s*.

Ex:

cria o semáforo *s* //valor=1.

P(s) //como $s > 0$, decrementa *s*, $s = 0$ e continua.

...aqui $s=0$ e o bloco de código esta travado para acesso externo.

V(s) //incrementa *s*, liberando acesso externo ao bloco.

Os semáforos podem ser:

mutex semaphore: Mecanismo usado para implementar exclusão mútua.

mutex event: Usado para suportar mecanismo de broadcasting.

esperar um pouco: O mesmo que “mutex event”, mas estendido para incluir múltiplas condições.

A *PThreads* oferece um conjunto de funções para uso de semáforos, são elas:

Semaphore_init: Inicializa o semáforo com valor 1.

Semaphore_down: Decrementa o valor do semáforo e bloqueia acesso ao recurso se a mesma for menor ou igual a 0.

Semaphore_up: Incrementa o semáforo.

Semaphore_decrement: Decrementa o semáforo sem bloquear.

Semaphore_destroy: Destrói o semáforo.

Dica: cuidado com deadlock, um deadlock ocorre quando o processo A espera o processo B e o processo B espera o processo A. Normalmente um deadlock ocorre quando a thread espera um recurso que não vai ser liberado, ou com o uso de wait circular. A dica é usar `timec_wait` no lugar de `wait`.

7.4 Sincronização com variáveis condicionais

Para gerenciamento das condições de processamento, as threads podem utilizar variáveis globais. Entretanto, uma das threads terá de ficar verificando continuamente o valor de determinada variável condicional, até sua liberação, ocupando tempo de processamento. Para um gerenciamento mais inteligente, a *PThreads*, fornece as variáveis condicionais.

Cria-se uma variável condicional que é manipulada pelas funções `wait`, `signal` e `status`, para sincronização das threads.

wait(condição); A thread corrente é suspensa, até que a condição ocorra. Observe que é criada uma lista com as threads que estão esperando esta condição.

timed_wait(); A thread corrente é suspensa, até que a condição ocorra ou até que o intervalo de tempo especificado seja espiado.

signal(condição); informa que a condição ocorreu, liberando a execução da thread que esta esperando.

status(condição); retorna o número de processos esperando a condição ocorrer.

Apresenta-se, nas linhas a seguir, o uso das variáveis condicionais de *Pthreads*.

```
//cria variável condicional
```

```
pthread_cond_t cond1;
```

```
//inicia variável condicional
```

```
pthread_cond_init ( cond1 , NULL);
```

```
..
```

```
//espera que a condição ocorra e desbloqueia o mutex1
```

```
pthread_cond_wait ( cond1 , mutex1 );
```

```
....
```

```
/*informa que a condição ocorreu, liberando a execução da thread que esperava a cond1. */
```

```
pthread_cond_signal ( cond1 );
```

```
...
```

```
//destrói a variável cond1.
```

```
pthread_cond_destroy ( cond1 );
```

No lugar de `wait`, pode-se utilizar `pthread_cond_timed_wait ()`; que espera um determinado intervalo de tempo para que a condição ocorra.

Também é possível enviar uma mensagem liberando a condição para todas as threads, `pthread_cond_broadcast()`. Entretanto, somente uma das threads terá liberado o mutex.

A Figura 1 ilustra os mecanismos de comunicação entre processos e threads.

A Figura 1 ilustra múltiplos processos e threads rodando simultaneamente.

8 Funções básicas para criar threads

Apresenta-se a seguir um conjunto de funções em C da biblioteca Posix Threads (*Pthreads*), e que são utilizadas para implementar threads.

8.1 Para criar uma thread

Primeiro você precisa criar um identificador para a thread

```
pthread_t thread_id;
```

Opcionalmente, pode criar uma estrutura de dados que serão passados para a thread.

```
pthread_attr_t thread_id_attribute;
```

A seguir, deve-se declarar e definir a função a ser executada pela thread. A função deve retornar `void` e ter como parâmetro um `void*`.

```
void thread_function(void *arg);
```

Finalmente, você pode executar a thread chamando a função `thread_create`. A thread que será criada levará em conta os atributos definidos em `thread_id_attribute`, retornará o id da thread no parâmetro `thread_id`, e executará a função `thread_function` usando o argumento `arg`. O retorno da função `pthread_create` pode ser 0 indicando sucesso, ou diferente de 0 indicando uma falha.

```
pthread_create( &thread_id, thread_id_attribute, (void *)&thread_function, (void *) &arg);
```

Figura 4: Mecanismos de comunicação entre processos e threads,[5].

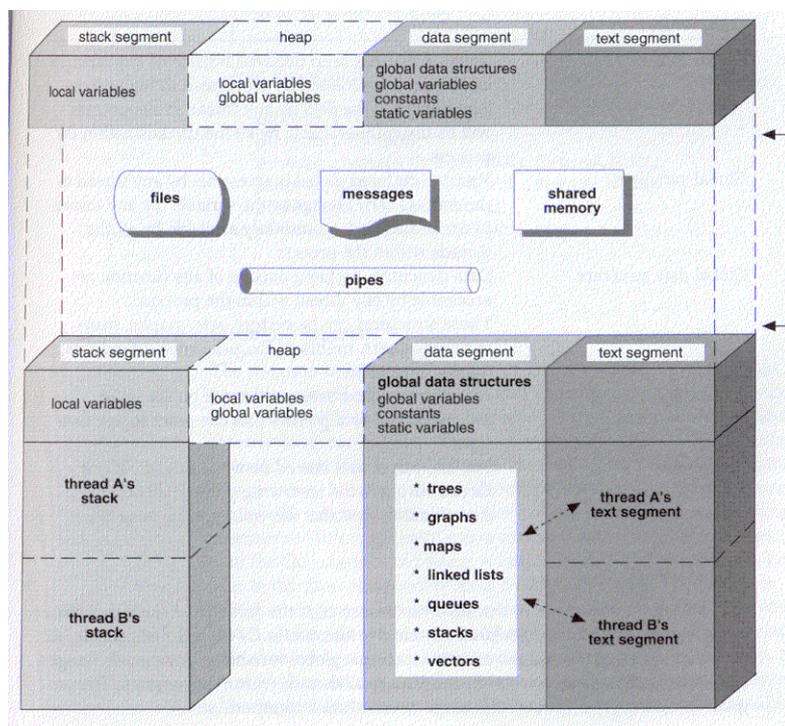
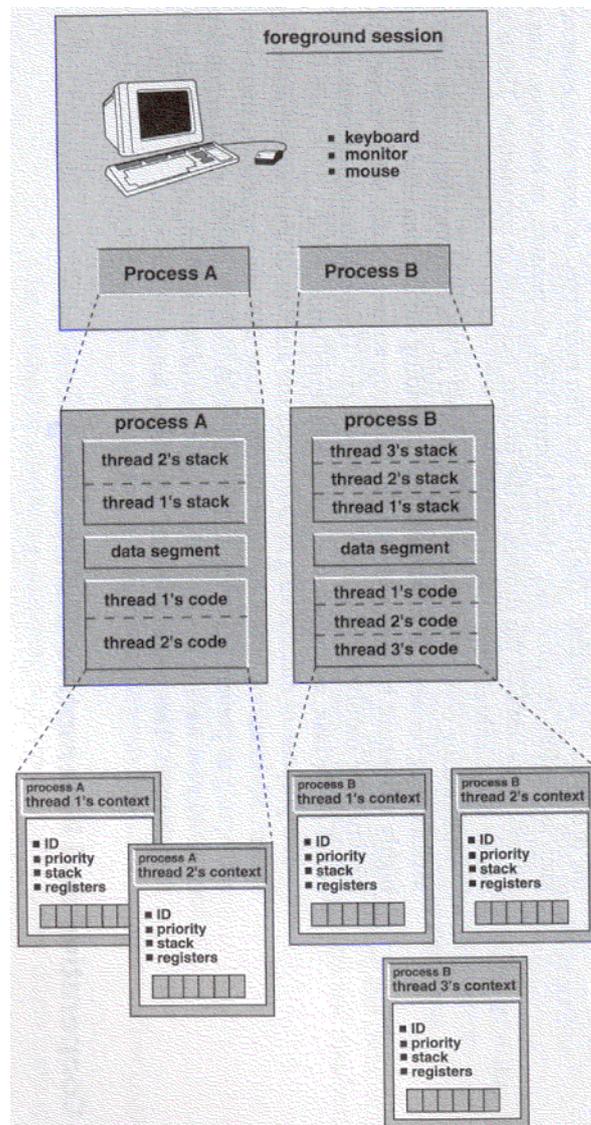


Figura 5: Múltiplos processos e threads rodando simultaneamente, [5].



Observe que pthread_create lança a thread e continua na execução da função atual.

Quando você cria a thread pode setar o tamanho da pilha ou usar NULL, indicando o uso do tamanho default.

O retorno da função thread_function pode ser obtido com a função join.

8.2 Para recuperar o resultado de uma thread

Para evitar que o processo principal seja encerrado antes da conclusão da thread, você deve esperar a conclusão da thread usando a função join. Veja ilustração .

```
pthread_join(thread id,&retval);
```

Observe que com join pode-se recuperar o valor de retorno da função executada pela thread.

8.3 Para encerrar uma thread

A thread é automaticamente encerrada quando a função thread_function é concluída. Entretanto, você pode solicitar explicitamente o encerramento da thread com pthread_exit.

```
void pthread_exit(void* status);
```

Quando uma thread é encerrada os recursos alocados pela thread são liberados.

8.4 Para deixar a thread destacada

Para deixar a thread independente do processo que a criou você deve usar a função detach, ou iniciar a thread com o atributo detach (de destacada).

```
int pthread_detach(pthread_t thread_id);
```

8.5 Para cancelar uma thread

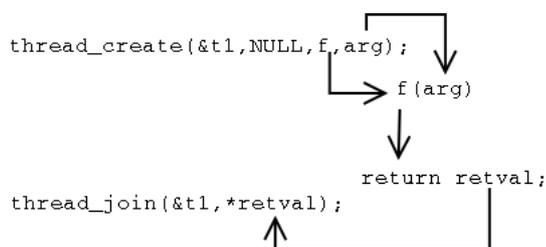
Você pode cancelar a thread usando a função pthread_cancel. Ou seja, a thread corrente cancela a thread com o id passado como argumento.

```
int pthread_cancel(pthread_id thread_id);
```

A função pthread_set_cancel_state é usada para definir se a thread pode ou não ser cancelada. E a função pthread_cancel_state informa se a thread pode ser cancelada.

```
pthread_set_cancel_state(PTHREAD_CANCEL_ENABLE,NULL);
```

Figura 6: Funcionamento de join.



```
pthread_set_cancel_state(PTHREAD_CANCEL_DISABLE, NULL);
```

8.6 Para travar um bloco de código que acessa memória compartilhada (mutex)

O funcionamento de um mutex é simples, você cria uma variável do tipo mutex, e a seguir inicializa seu uso³. Agora, sempre que for usar a variável que é compartilhada, você trava o bloco de código usando `pthread_mutex_lock`, podendo usar a variável compartilhada. Quando você encerrou o uso da variável compartilhada, pode liberar o bloco de código para as demais threads, usando `pthread_mutex_unlock`.

Exemplo:

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, pthread_mutexattr_default);
int volatile variável_compartilhada;
pthread_mutex_lock( &mutex );
variável_compartilhada = algo;
pthread_mutex_unlock( &mutex );
```

8.7 Para realizar um delay em uma única thread

A função `sleep(tempo)` é utilizada em um programa para executar uma pausa. Entretanto, em um programa com threads, todas as threads realizarão a pausa. Para que a pausa se aplique a uma única thread use a função `pthread_delay_np`.

```
pthread_delay_np( &delay );
```

9 Roteiro básico para desenvolvimento de um programa com threads

Agora que já descrevemos as principais funções utilizadas, podemos apresentar um pequeno roteiro para desenvolvimento de um programa com threads.

1. Cria o programa normalmente, roda o programa e testa seu funcionamento.
2. Identifica os pontos onde o programa apresenta processos concorrentes.
3. Inclue o header `<threads.h>` (ou `<pthread.h>`).

³Opcionalmente pode-se passar um conjunto de atributos para a variável mutex.

4. Monta uma função separada para a seção que tem código concorrente (que pode ser paralelizado). Por padrão, a função deve retornar `void*` e receber `void*` (**void* FunçãoThread(void*)**). Observe que o argumento pode ser uma estrutura de dados.
5. Cria variáveis adicionais (intermediárias) e variáveis que devem ser compartilhadas. As variáveis que serão compartilhadas devem ser declaradas com `volatile`.
6. Cria uma variável mutex (**pthread_mutex trava;**), que será utilizada para bloquear a parte do código que não pode ser executada ao mesmo tempo pelas duas (ou mais) threads.
7. Inicializa a variável mutex (**pthread_mutex_init(&trava, val)**).
8. Cria as diferentes threads (**pthread_t t1, t2, ..., tn;**).
9. Dispara a execução de cada thread usando chamadas a **pthread_create(&t1, NULL, FunçãoThread, &arg)**.
10. Observe que se as variáveis compartilhadas forem utilizadas, você deve incluir bloqueadores de acesso ao código usando **pthread_mutex_lock(&lock)** e **pthread_mutex_unlock(&lock)**.
11. Use a função **pthread_join(thread, &retval)** para esperar o encerramento da thread e obter o valor de retorno da thread. Observe que você deve usar `pthread_join` para evitar que o programa principal encerre sua execução antes das threads terem sido concluídas.
12. Na hora de compilar use a opção **-D_REENTRANT**.

Dica: Observe que se uma das threads chamar a função `exit()`, a mesma encerra a execução do programa sem que as demais threads tenham sido encerradas.

10 Exemplo de código usando múltiplas threads

Para compilar este exemplo use
`g++ -v exemplo-thread.cpp -lpthread -D_REENTRANT`.

Listing 4: Determinação de pi usando thread.

```
#include <stdio.h>
#include <stdlib.h>
#include "pthread.h"

volatile double pi = 0.0;
volatile double intervals;
pthread_mutex_t pi_lock;

//Função thread
void * process (void *arg)
{
  register double width, localsum;
  register int i;
  register int iproc = (*(char *) arg) - '0';

  width = 1.0 / intervals;
  localsum = 0;
```

```

for (i = iproc; i < intervals; i += 2)
{
    register double x = (i + 0.5) * width;
    localsum += 4.0 / (1.0 + x * x);
}

localsum *= width;

/*trava acesso a esta parte do código, altera pi, e
 destrava*/
pthread_mutex_lock (&pi_lock);
pi += localsum;
pthread_mutex_unlock (&pi_lock);
return (NULL);
}

int main (int argc, char *argv[])
{
    pthread_t thread0, thread1;
    void *retval;

    intervals = atoi (argv[1]);

    /* Inicializa a variável mutex*/
    pthread_mutex_init (&pi_lock, NULL);

    /* Executa duas threads */
    if(pthread_create(&thread0,NULL,process,(void*)"0")
        ||
        pthread_create(&thread1,NULL,process,(void*)"1"))
    {
        fprintf (stderr, "%s:_cannot_make_thread\n", argv
            [0]);
        exit (1);
    }

    /* Join espera as threads terminarem, o retorno é
    armazenado em retval */
    if ( pthread_join (thread0, &retval)
        || pthread_join (thread1, &retval))
    {
        fprintf (stderr, "%s:_thread_join_failed\n", argv
            [0]);
        exit (1);
    }

    printf ("Estimation_of_pi_is_%f\n", pi);
    return 0;
}

//g++ -v exemplo-thread.cpp -lpthread -o exemplo-thread

```

11 Common C++

Apresenta-se nesta seção a biblioteca common C++, o que é, onde encontrar e um exemplo.

11.1 O que é e onde encontrar ?

Uma biblioteca de programação da gnu que fornece um conjunto de classes e objetos para desenvolvimento de programas usando threads, funciona em GNU/Linux, Windows e em outras plataformas.

Disponível no site (<http://www.gnu.org/directory/GNU/commoncpp.html>).

11.2 As classes Thread, Mutex e Conditional da biblioteca common C++

Apresenta-se na listagem a seguir, de forma resumida, as classes Thread, Mutex e Conditional da biblioteca common C++. Para maiores detalhes consulte a documentação distribuída junto com a biblioteca. Os exemplos apresentados na seção 11.3 esclarecem o uso destas classes.

Listing 5: As classes Thread, Mutex e Conditional da biblioteca common C++.

```

//Uma classe Thread
class CCXX_CLASS_EXPORT Thread
{
public:
    typedef enum Cancel
    {
        cancelInitial=0,
        cancelDeferred=1,
        cancelImmediate,
        cancelDisabled,
        cancelManual,
        cancelDefault=cancelDeferred
    } Cancel;

    typedef enum Suspend
    {
        suspendEnable,
        suspendDisable
    } Suspend;

private:
    static Thread* _main;
    Thread *_parent;
    enum Cancel _cancel;
    Semaphore *_start;
    friend class ThreadImpl;
    class ThreadImpl* priv;
    static unsigned __stdcall Execute(Thread *th);

    void close();

protected:
    virtual void run(void) = 0;
    virtual void final(void){return;};
    virtual void initial(void){return;};
    virtual void* getExtended(void) {return NULL;};
    virtual void notify(Thread*){return;};
    void exit(void);
    bool testCancel(void);
    void setCancel(Cancel mode);
    void setSuspend(Suspend mode);
    void terminate(void);
    inline void clrParent(void){_parent = NULL;};

public:
    Thread(bool isMain);
    Thread(int pri = 0, size_t stack = 0);
    Thread(const Thread &th);
    virtual ~Thread();
    static void sleep(timeout_t msec);
    static void yield(void);
    int start(Semaphore *start = 0);
    int detach(Semaphore *start = 0);
    inline Thread *getParent(void){return _parent
        };
    void suspend(void);
    void resume(void);
    inline Cancel getCancel(void){return _cancel;};
    bool isRunning(void);
    bool isThread(void);
    friend CCXX_EXPORT(Throw) getException(void);
}

```

```

    friend CCXX_EXPORT(void) setException(Throw
        mode);
    friend inline void operator++(Thread &th)
        {if (th._start) th._start->post();};
    friend inline void operator--(Thread &th)
        {if (th._start) th._start->wait();};
};

//Uma classe Mutex
class CCXX_CLASS_EXPORT Mutex
{
    friend class Conditional;
    friend class Event;
private:
    volatile int _level;
    volatile Thread *_tid;
    pthread_mutex_t _mutex;
public:
    Mutex();
    virtual ~Mutex();
    void enterMutex(void);
    bool tryEnterMutex(void);
    void leaveMutex(void);
};

//Uma classe Conditional
class Conditional : public Mutex
{
private:
    pthread_cond_t _cond;
public:
    Conditional();
    virtual ~Conditional();
    void signal(bool broadcast);
    void wait(timeout_t timer = 0);
};

class CCXX_CLASS_EXPORT Semaphore
{
private:
    int _semaphore;
public:
    Semaphore(size_t resource = 0);
    virtual ~Semaphore();
    void wait(void);
    bool tryWait(void);
    void post(void);
    int getValue(void);
};

```

11.3 Exemplos de código com threads usando a biblioteca de classes common C++

O exemplo a seguir é distribuído juntamente com a biblioteca common C++.

Listing 6: Usando a biblioteca de classes common c++.

```

#include <cc++/thread.h>
#include <cstdio>
#include <cstring>
#include <iostream>

#ifdef CCXX_NAMESPACES
using namespace std;
using namespace ost;
#endif

//Classe thread filha
class Child:public Thread

```

```

{
public:
    Child () { };

    void run ()
    { cout << "child_start" << endl;
      Thread::sleep (3000);
      cout << "child_end" << endl;
    }

    void final ()
    { delete this;
    }
};

//Classe thread pai
class Father:public Thread
{
public:
    Father () { };

    void run ()
    { cout << "starting_child_thread" << endl;
      Thread *th = new Child ();
      th->start ();
      Thread::sleep (1000);
      cout << "father_end" << endl;
    }

    void final ()
    { delete this;
      //reset memory to test access violation
      memset (this, 0, sizeof (*this));
    }
};

int main (int argc, char *argv[])
{
    cout << "starting_father_thread" << endl;
    Father *th = new Father ();
    th->start ();
    Thread::sleep (10000);
    return 0;
}

/*
g++ thread2-adb.cpp -o thread2-adb -I/usr/local/
include/cc++2 -D_GNU_SOURCE
-pthread /usr/local/lib/libcgnu2.so -L/usr/local/lib
-ldl -Wl,--rpath -Wl,/usr/local/lib
*/

```

O exemplo a seguir utiliza a biblioteca de threads common C++ para determinar o valor de pi.

Listing 7: Determinação do valor de pi usando a biblioteca common c++.

```

#include <iostream>
#include <vector>

//Inclue a biblioteca de threads do common c++
#include <cc++/thread.h>

//Define o uso dos namespaces std e ost
#ifdef CCXX_NAMESPACES
using namespace std;
using namespace ost;
#endif

/*Cria classe thread com nome MinhaClasseThread.
Observe que os atributos comuns as threads são
declarados como estáticos. Observe o uso do objeto
Mutex*/

```

```

class MinhaClasseThread: public Thread
{
private:
    static Mutex pi_lock;
    int iproc;
public:
    static volatile double pi;
    static volatile int nproc;
    static volatile double intervals;
protected:
    virtual void run();
public:
    //Construtor
    MinhaClasseThread(int _iproc = 0):iproc(_iproc)
    { cout << "Construtor_iproc=_\n" << iproc << endl;
      };
};
//Inicializa atributos estáticos da classe
Mutex MinhaClasseThread::pi_lock;
volatile double MinhaClasseThread::pi = 0;
volatile double MinhaClasseThread::intervals = 10;
volatile int MinhaClasseThread::nproc;

//Define a função run
void MinhaClasseThread::run()
{
    cout << "iproc=_\n" << iproc << endl;
    register double width;
    register double localsum;

    width = 1.0 / intervals; //define a largura
    localsum = 0; //faz as contas locais
    for (register int i = iproc; i < intervals;
         i += nproc)
    {
        register double x = (i + 0.5) * width;
        localsum += 4.0 / (1.0 + x * x);
    }
    localsum *= width;

    //Trava o bloco de código com o mutex
    pi_lock.enterMutex();
    pi += localsum;
    pi_lock.leaveMutex();
};

//Função main
int main (int argc, char *argv[])
{
    //Obtem o intervalo
    if(argc <= 1)
    {
        cout <<"Usage:_\n" << argv[0] << "\n
        number_of_intervals_number_of_processor_\n" <<
        endl ;
        cout <<"Example:_\n" <<argv[0] << "_10_2_\n" << endl ;
        return -1;
    }

    MinhaClasseThread::intervals = atof (argv[1]);
    MinhaClasseThread::nproc = atoi (argv[2]);
    cout << "Entrada:_\n" << argv[0] << "_\n" << argv[1] << "\n"
    << argv[2] << endl;

    //Cria as threads
    vector< MinhaClasseThread*> vthread;
    for(int i = 0; i < MinhaClasseThread::nproc ; i++)
    {
        MinhaClasseThread* T= new MinhaClasseThread( i )
        ;
        vthread.push_back( T );
    }

    //Executa as threads
    for(int i = 0; i < MinhaClasseThread::nproc ; i++)
        vthread[i]->start();

    //Enquanto as threads estiverem rodando,
    //realizando uma pausa na thread primária
    for(int i = 0; i < MinhaClasseThread::nproc ; i++)
        while (vthread[i]->isRunning())
            Thread::sleep(10);

    //Mostra resultado
    cout <<"Estimation_of_pi_is_\n" << MinhaClasseThread::pi
    << endl;
    return 0;
}
/*
Para compilar:
g++ t6.cpp -o t6 -I/usr/local/include/cc++2 -
D_GNU_SOURCE -pthread /usr/local/lib/libcgnu2.so
-L/usr/local/lib -ldl -Wl,--rpath -Wl,/usr/local/
lib

Saída:
=====
[andre@mercurio threads]$ ./t6 10 2
Entrada: ./t6 10 2
Construtor iproc = 0
Construtor iproc = 1
iproc = 0
iproc = 1
Estimation of pi is 3.14243
*/

```

12 Comparação dos tempos de processamento

A Tabela 2 apresenta uma comparação do tempo de processamento utilizando multi-threadings e multi-processing.

Os computadores utilizados para os testes tem placa mãe ASUS-CUV4X-DLS com 2 processadores PIII de 1000MHz, 2GB de memória ram, placa de rede Intel EtherExpress Pro 100b de 100Mbps. E estão conectados a um switch 3com SuperStack III de 100Mbps usando cabos de par trançado. O sistema operacional utilizado é GNU/Linux com Mosix, kernel 2.4.19. O compilador é o gcc 2.96.

O cluster foi montado com 3 computadores (6 processadores).

Com 1 thread o tempo de processamento ficou em 1m4.850s, o uso de 2 threads é vantajoso (0m32.443s), e, como esperado, o uso de 4 threads implica em um tempo de processamento maior (0m32.602s). Todas as threads são executadas no mesmo computador, o mosix não distribuição processos que executem threads.

O uso de processos com mosix tem o comportamento esperado, o que pode ser visualizado através do programa de monitoramento (mon). A simulação com 2 processos, foi toda executada no mesmo computador, não houve redistribuição dos processos, e o tempo de processamento ficou em 0m22.963s. Quando se usa 4 processos, o mosix distribui os processos pelas máquinas do cluster, e o tempo de processamento ficou em 0m12.592s. Com seis processos o tempo de processamento ficou em 0m10.090s. De um modo geral, o tempo de processamento pode ser estimado pela relação tempoNormal/numeroProcessos com perdas pequenas em função da redistribuição dos processos.

Tabela 2: Comparação tempos processamento.

Comando	intervalos	número threads	tempo	
./threads	1000000000	1	1m4.850s	
./threads	1000000000	2	0m32.443s	
./threads	1000000000	4	0m32.602s	
Comando	intervalos	número processos	tempo	
./processos	1000000000	2	0m22.963s	
./processos	1000000000	4	0m12.592s	
./processos	1000000000	6	0m10.090s	

13 Conclusões

Apresentou-se as diferenças e semelhanças entre threads e processos, a estrutura de cada um, suas características e seus estados. As funções básicas para utilização de processos e threads são descritas de forma simples e com exemplos práticos. A seguir, apresentou-se a biblioteca common C++ que apresenta classes para manipulação de threads e mutexes, e alguns exemplos.

Verificou-se que a grande vantagem do uso de threads é o compartilhamento da memória pelas threads. Sendo a memória compartilhada a implementação de códigos paralelos é extremamente simplificada.

O uso de múltiplos processos, que são automaticamente distribuídos para as diferentes máquinas do cluster se mostrou eficiente e torna o uso de múltiplos processos com uso do Mosix uma opção de fácil implementação. Pois o uso das instruções fork e pipe é rapidamente compreendido pelos programadores de C/C++.

O uso de threads é relativamente simples, principalmente quando se utiliza uma biblioteca como a *common C++*.

Referências

- [1] André Duarte Bueno. *Introdução ao Processamento Paralelo e ao Uso de Clusters, Parte I: Filosofia*, 11 2002.
- [2] David R. Butenhof. *Programming with POSIX(R) Threads*. Addison-Wesley, 1987.
- [3] Hank Dietz. *Linux Parallel Processing HOWTO*. <http://yara.ecn.purdue.edu/pplinux/PPHOWTO/pphowto.html>, 1998.
- [4] GNU. *GNU CommonC++ Reference Manual*. GNU, <http://www.gnu.org/directory/GNU/commoncpp.html>, 2002.
- [5] Cameron Hughs and Tracey Hughes. *Object Oriented Multithreading using C++: architectures and components*, volume 1. John Wiley Sons, 2 edition, 1997.
- [6] Brian Masney. Introduction to multi-thread programming. *Linux Journal*, abril 1999.
- [7] LinuxThreads Programming. Matteo dell omodarme.
- [8] Bryan Sullivan. *Faq - threads* - <http://www.serpentine.com/bos/threads-faq/>, 1996.
- [9] Tom Wagner and Don Towsley. Getting started with posix threads. University of Massachusetts at Amherst, July 1995.
- [10] Barry Wilkinson and C. Michael Allen. *Parallel Programming: Techniques and Applications Using Workstation and Parallel Computers*. Prentice Hall, 1999.