Threads POSIX

Marcelo Johann

Nas aulas anteriores…

Processos

Plano da aula

- Introdução: do processo às **threads**
- Duas categorias de threads
  - Núcleo vs. usuário
- Mapeamento threads/processos
  - 1:1, N:1, N:M
- Exemplos de implementações
  - Solaris, Windows 2000, Linux.
- Programação com Threads
  - Posix Threads

A imagem do processo

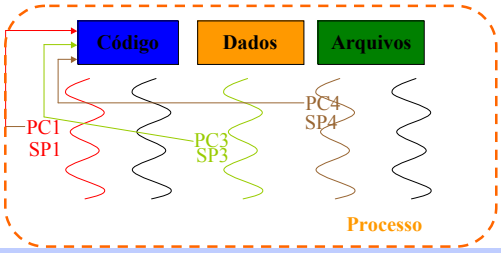
- **Texto**: contém as instruções binárias do código executável do processo (imagem).
- **Dados**: espaço para as variáveis do processo, declaradas como globais no programa.
- **Heap**: memória para alocação sob-demanda durante a execução
  - Alocação dinâmica (malloc/free)
- **Pilha**: memória para alocação:
  - De variáveis locais a sub-rotinas (vide chamada de funções);
  - Do endereço de retorno de uma sub-rotina.
- Área do usuário vs. área do sistema.

O processo é…

- Um programa em execução
  - Uma unidade de escalonamento
  - Um fluxo de execução
- Um conjunto de recursos (contexto) gerenciados pelo Sis. Op.
  - Registradores (PC, SP, …)
  - Memória
  - Descritores de arquivos
  - Etc…
- A troca de contexto é uma operação pesada
  - Deve acontecer cada vez que há decisão de escalonamento

A noção de *thread*

- Idéia simples: associar **mais de um fluxo de execução** a um processo:
  - Compartilhamento de recursos do PCB
  - O PCB deve incluir uma lista de threads!



Implementação de threads

- O Bloco descritor de Threads inclui:
  - Registradores privados
    - PC, SP, registradores de uso comum.
  - Uma pilha
    - Histórico da execução, com a várias chamadas a sub-rotinas que não completaram e suas variáveis locais.
    - Endereço de retorno após completar a chamada.
  - Thread ID
  - Ponteiros para outras threads
  - Ponteiro para o PCB em que se encontra.
    - Inclusive o espaço de endereçamento do processo pai!
  - Informação de escalonamento
    - Prioridade, estado, tipo de escalonamento…;

Vantagens das *threads*

- São processos **leves** (!= processos pesados).
  - Troca de contexto mais rápida;
    - Fator 5 no caso de SOLARIS.
  - Tempo de criação menor
    - Fator 30 no caso de SOLARIS.
  - Logo, diminui o tempo de resposta do sistema;
- Maior facilidade para mesclar threads I/O-bound com threads CPU-bound.
- Usa eficientemente as arquiteturas multi-processadas/multi-cores.
- O compartilhamento de recursos facilita a comunicação entre as *threads*:
  - Através da área de memória compartilhada (global, heap).
  - Necessita de sincronização.

Porque usar threads?

- Duas respostas a essa pergunta:
  - Em nível do projeto de Sistema Operacional:
    - Eficiência, aproveitamento de recursos…
    - Mascarar E/S com cálculo.
  - Em nível do desenvolvedor:
    - Melhor reatividade do software;
    - Melhor expressão das funcionalidades;
      - Assincronismo
    - Melhor organização do programa;
    - Obrigação de "programar bem"
      - Testes, depuração, isolamento de código crítico…
    - Linguagens dão suporte
      - Ada, Modula-3, Python, SmallTalk, Objective-C, Java

Dois tipos de *threads*

- **Thread "usuário"**
  - Uma biblioteca dá suporte a criação, escalonamento…, que executa em modo usuário.
  - O núcleo não interfere nas threads.
  - Vantagens: muito rápidas de gerenciar pois não necessitam do núcleo (chamada de sistema…)
  - Exemplo: bibliotecas Pthreads (POSIX), threads de SOLARIS.
- **Thread "núcleo"**
  - O núcleo oferece suporte a threads.
  - Mais lento…
  - O Sis. Op. pode escalonar mais eficientemente as threads, inclusive em máquinas multi-processadas.
  - Exemplo: Windows, Solaris, Linux.

Como conciliar os dois níveis?

- 3 soluções:
  - N *threads* usuário por *thread* de núcleo (N:1)
  - 1 *thread* usuário por *thread* de núcleo (1:1)
  - Meio termo entre os dois (N:M)

O modelo N:1

- N *threads* usuário estão fisicamente implementadas em uma *thread* de núcleo.
- Todo o gerenciamento das *threads* se faz em nível de usuário
  - Extremamente veloz.
- Grande limitação: o escalonamento.
  - O núcleo escalona as *threads* na CPU;
  - Ele só enxerga a *thread* de núcleo sem distinguir as *threads* nela implementadas.
  - Se uma *thread* de usuário tranca, toda a *thread* de núcleo estará bloqueada!
- Totalmente inapropriado para máquinas multi-processadas.
- Exemplo: NACHOS, MACH C-threads, POSIX Pthreads.

Exemplo do modelo N:1: Java *Threads*

- Java disponibiliza uma interface de programação com *threads*.
- Um programa Java executa dentro de uma máquina virtual
  - A JVM é executada, em geral, como um processo único.
  - A JVM usa várias *threads* para seu gerenciamento
    - Memória, coleta de lixo…
- O mapemanto das *threads* Java da JVM para as threads do Sis. Op. depende da implementação da JVM!
  - Caso seja para *threads* de usuário, pode ter ilusão de assincronismo!
  - No Windows, mapeamento 1:1 com *threads* de núcleo

O modelo 1:1

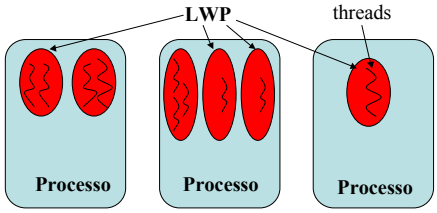
- Cada *thread* de usuário é mapeada em uma *thread* de núcleo.
  - Dessa forma, o Sis. Op. escalona as *threads* na CPU e não há blocagem.
- Adaptado a arquiteturas multi-processadas.
- Problema: maior custo de criação/manutenção das *threads*
  - Limita-se o número de *threads* disponíveis em tais sistemas.

O modelo N:M

- Várias *threads* de usuário são implementadas em várias *threads* de núcleo.
  - O número exato pode variar conforme for a arquitetura e/ou a aplicação.
- Junta as vantagens dos dois outros modelos.
- É uma solução herdada diretamente do sistema Solaris.

Caso de estudo: SUN Solaris

- Versão proprietária (SUN) do Sis. Op. UNIX.
  - Até 1992: Solaris 1.x suportava unicamente processos pesados.
- *Threads* de usuário (= *threads*)
- "Ligth Weight Process" (LWP): entidade intermediária entre o processo pesado e a *thread*.



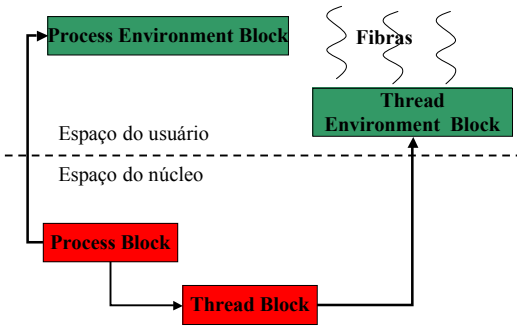
Mapeamentos threads
LWP – Processos no Solaris

- Um LWP é mapeado a uma thread de núcleo (1:1).
  - Ou seja, o núcleo escalona a entidade LWP.
- As *threads* (de usuário) podem ser:
  - *Limitadas :* sempre estão mapeadas em um dado LWP
  - *Ilimitadas*: podem ser mapeadas em qualquer momento em qualquer LWP.
    - o mapeamento é decidido em nível de usuário!
- Implementação:
  - Thread = thread-id, registradores (PC, SP), pilha e prioridade.
  - LWP = registradores, memória e contabilidade (no espaço do núcleo).
  - Thread de núcleo: prioridade, escalonamento, ponteiro para o LWP ativo.

Processo e *Thread* no Windows

- A partir de Windows 2000!
  - Muito inspirado do Solaris.
- O processo é descrito por um PCB
  - Parte no espaço do núcleo
  - Parte no espaço do usuário (PEB).
- Um Processo inclui no mínimo uma *thread*
  - *Thread* de núcleo
  - Pode ter mais de uma para máquinas multi-processadas.
  - Uma *thread* possui uma parte no espaço de núcleo, uma outra no espaço do usuário (2 pilhas!)
- As *threads* podem hospedar várias fibras.
  - = *thread* em nível de usuário.
  - Criadas por conversão de uma *thread*.
  - Também chamadas de "*lightweigth thread*".

Processo e Thread no Windows



Bibliotecas de Threads no Linux

Existe um bom número de bibliotecas de *threads* no Linux. Alguns exemplos:

- FSU Pthreads
  - Compatível POSIX
  - Usuário
- LinuxThreads
  - Usuário e núcleo
  - POSIX
- PCThreads
  - POSIX
  - Usuário
- NPGT (Next Generation POSIX Threads)
  - IBM. Terminado em 2003.
- NPTL
- ANAHY (Unisinos)

Linux Threads & NPTL

- Historicamente, **Linux Threads** foi a biblioteca "integrada" à libC, vindo com o núcleo Linux.
  - Mapeamento 1:1.
  - A troca de contexto é suficientemente rápida devido à implementação das *threads* de núcleo igual à dos processos (vide clone())
  - Número limitado de *threads* (8192).
  - Gerente de *threads* (criação/terminação).
  - Ausência de estruturas de sincronização no núcleo.
    - Usava sinais.
  - Foi descontinuada em 2002.
- NPTL (**Native Posix Thread Library**):
  - Desenvolvido com o núcleo 2.5 pela RedHat; liberado no 2.6.0 (Natal 2003)
  - Uma das principais novidades do núcleo 2.6.x
  - Ficou com o modelo 1:1
  - Acabou com o gerente de *threads*.
  - Sem limitação no número de *threads* (espaço de TID)

Do UNIX ao POSIX (1)

- Pre-história: MULTICS na Bell Telecom (AT&T)
  - Ken Thompson, Dennis Ritchie.
  - Desenvolvido em um DEC PDP-7
  - Terminado em março de 69.
- Continuado por Thompson – UNIX (1971)
  - Sistema de Arquivos;
  - Sub-sistema de gerenciamento de processos;
  - Shell (futuro Bourne-Shell);
  - 1o manual (UNIX Programmer's Manual).
- Porte numa PDP-11 (1972)
  - Editor ed;
  - Linguagem B.
- Unix, 4a edição (Bell Telecom)
  - Incluindo C;
  - Re-escrito em C.
  - Communications of the ACM, Jul. 1974.
- AT&T não podia comercializar UNIX
  - Deu a código para universidades:
    - Berkeley (1973), …
  - Interação Bell T. & Universidades -> versão 7 (1979).

Do UNIX ao POSIX (2)

- 1978: porte para **novas arquiteturas**
  - Unix + Intel 8086 = XENIX
  - Unix + VAX-11 = UNIX32V (32 bits)
    - Mandada para Berkeley = v. 3 BSD (1979)
- **Berkeley Software Distribution**: a versão de Berkeley
  - Desde 1974:
    - Ex, vi, Pascal
  - 3 BSD (1979), com paginação.
    - Decidiu o ministério da defesa a fomentar BSD!
    - Incorporação de TCP/IP
  - Versões 4.0 BSD (1980), …, 4.4 BSD (1993).
    - Sockets, signals, …
  - FreeBSD (dez. 1993)
- 1983: AT&T comercializa o **System V**
  - ICP,…
- 1982: Comercialização agressiva
  - A partir de System V ou de BSD.
  - 1982: SunOS / Solaris (nfs, vfs)
  - IBM/AIX
  - HP-UX
  - ULTRIX (dec) (multi-processadores)

Normalização: POSIX

- 1987: AT&T se une com a Sun para criar uma fusão SunOS/System V: SVR4
  - Tornou-se Solaris em 1994.
- Incompatibilidades incentivaram a procura de uma norma unificada.
  - System V Interface Definition (AT&T)
    - Baseado no Unix System V…
  - POSIX (IEEE)
    - Portable O. S. based on Unix
    - POSIX 1003.1 (1990)
    - POSIX.4 (1993)
    - POSIX.14 (1995) - Multi-tarefas / *threads*
- Define um sub-conjunto de chamadas de sistema que um sistema "*-IX" deve fornecer.
  - Portabilidade vs. eficiência.

Unix: por que tal sucesso?

- Distribuição do código fonte nos anos 70, de graça, possibilitou ciclo de desenvolvimento rápido.
  - Lembra de um outro Sis. Op.? :o)
- (No início) "*Small is beautifull*".
- Interface com periféricos E/S através de arquivos.
- Portabilidade através da linguagem C.

- Limitações:
  - Complexidade crescente;
  - Núcleo monolítico e complicado.
  - "*UNIX is simple and coherent, but it takes a genius (or, at any rate, a programmer) to understand its simplicity*" (D. Ritchie).

Usando *Threads* com POSIX

- POSIX define uma biblioteca de *threads* chamada pthread.
  - Em nível de usuário e/ou de núcleo;
  - Usada pelo programador de aplicações.

- O que tem em *Pthreads*?
  - Criação de *threads*;
    - Compartilham dados e arquivos!
  - Qual programa a *thread* deve executar;
  - Sincronização entre *threads*;
  - Escalonamento entre *threads*;
  - Terminação de *threads*.

Criação de threads

- int pthread_create(pthread_t * thread,
  const pthread_attr_t * attr,
  void * (*start_routine)(void *),
  void *arg);

- Argumentos:
  - **thread** – Thread ID.
  - **attr** – NULL para o default.
  - **start_routine**: é uma função que será executada pela thread. Pega em argumento um 'void*' e retorna um void*.
  - **arg** – ponteiro sobre o argumento da função. Para passar mais de um argumento, usar um vetor ou uma struct.

Os Atributos da Thread

- A struct 'pthread_attr_t' inclui:
  - A informação se a thread é "detachable" ou não.
  - O algoritmo de escalonamento
    - real-time?
    - SCHED_FIFO
    - SCHED_RR
    - SCHED_OTHER
  - O tipo de thread
    - Threads de núcleo: PTHREAD_SCOPE_SYSTEM
    - Threads de usuário: PTHREAD_SCOPE_PROCESS
  - Outros:
    - O endereço da pilha
    - O tamanho da pilha

Sincronização de threads

- **pthread_join**() garante que o processo pai espere até a thread em argumento terminar sua execução.

- **Mutexes**: Semafóros binários.
  - Garante que não haja corrida.
  - Deveria ser usado em cada dado que é compartilhado.
  - pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
  - pthread_mutex_lock(&mutex);
  - pthread_mutex_unlock(&mutex);

Conclusão sobre *threads*

- O recurso mais moderno dos Sistemas Operacionais
  - Vem embutido desde 1998, 2000…
- Existem várias bibliotecas de programação concorrente com threads:
  - Pthread, Java Threads, …
  - Em geral em nível de usuário.
- Tornou-se a unidade básica de escalonamento na CPU quando o Sis. Op. oferece *threads* de núcleo
  - Ex.: Windows
- Por isso, cada vez mais, as noções vistas com processos se aplicam, na realidade de hoje, a *threads*:
  - Algoritmos de escalonamento;
  - Sincronizações.
- Problemas:
  - **granularidade**
  - **sincronização**

Bibliografia complementar

- **Unix Internals**, Uresh Vahalia, Prentice Hall 1996.
- **Programming for the real world – POSIX.4**, Bill O. Gallmeister, O'Reilly 1995.
- **Programming with POSIX Threads**, David R. Butenhof, Addison-Wesley 1997.
- **Linux Internals**, Moshe Bar, McGraw-Hill 2000

- Linux Threads, http://pauillac.inria.fr/~xleroy/linuxthreads
- Linux Kernel: http://www.linuxhq.com/lkprogram.html

Próxima Aula

Escalonamento