INF01151 – Sistemas Operacionais II

Aula 07 - Da teoria à prática... Parte I

(Processos, threads, mutex, semáforos, variáveis de condição e monitores)

Prof. Alberto Egon Schaeffer Filho

INSTITUTO DE INFORMÁTICA UFRGS

Introdução

- Fluxos de controle para programação concorrente
  - Processos ou threads
- Processos/threads interativos
  - São concorrentes
  - Compartilham dados (sincronização de acesso – exclusão mútua) ou coordenam suas atividades entre si (sincronização de condição)
- Questões em aberto:
  - Como criar, *na prática*, fluxos de controle?
  - Como sincronizar, *na prática*, as atividades?

INSTITUTO DE INFORMÁTICA UFRGS

Processos no mundo UNIX

- *fork( )* cria um processo filho cópia do processo pai
  - Após o *fork( )* tem-se dois processos:
    - Processo pai: *fork* retorna o pid do filho
    - Processo filho: o *fork* retorna zero
  - Combinado com a chamada de sistema *exec( )* e similares
    - Substitui o espaço de endereçamento do processo filho
- *waitpid* (*pid_t pid*) faz o processo que a executa esperar pelo término do processo *pid*
- *exit(int status)* termina o processo

INSTITUTO DE INFORMÁTICA UFRGS

```
/* Orion Lawlor's Short UNIX Examples, olawlor@acm.org 2004/9/5
   Shows how to use fork() in a UNIX program.
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h> /* for pid_t */
#include <sys/wait.h> /* for wait */

void doWork(char *arg) {
    // snipped code
}

int main()
{
    /*Spawn a new process to run alongside us.*/
    pid_t pid = fork();
    if (pid == 0) {           /* child process */
        doWork("child");
        exit(0);
    }
    else {                    /* pid!=0; parent process */
        doWork("parent");
        waitpid(pid,0,0);     /* wait for child to exit */
    }
    return 0;
}
```

APIs para *threads*

- Principais APIs
  - **Microsoft Windows**
    - Microsoft Win32 e Microsoft Foundation Class (MFC)
    - Microsoft .Net e Commom Language Runtime (CLR)
  - **Java Threads**
    - Suporte a *threads* "embutido" na própria linguagem Java e suportado pela Java Virtual Machine
  - **POSIX Threads**
    - Padrão para plataformas UNIX
    - Possui uma versão open source para windows (*pthreads-win32*)
- Apresentam funcionalidades similares de formas diferentes
  - Criação, sincronização, término

Java threads

- Suporte a *threads* é integrado na linguagem
  - Permite criação e gerenciamento de threads de forma independente do sistema operacional ou de bibliotecas
  - Cada thread de execução tem sua própria *call stack*
- Todo programa java tem ao menos uma *thread*
  - *Main* thread, que contém método **`main`**
- Criação de threads
  - 1º método: Estendendo classe **`Thread`**
  - 2º método: Implementando interface **`Runnable`**
- Pacote **`java.lang`**

Criação de threads: 1º método

- Estendendo a classe **`Thread`**
  - Necessário sobrescrever o método **`run`** (the job to do)
  - Método **`start`** é usado para criar thread, e implicitamente chama **`run`**

```
class Simple extends Thread {
        public void run() {
           System.out.println("I'm a thread");
        }
}
```

```
public class First {
     public static void main(String args[]) {
          Simple s = new Simple();

          s.start();
           System.out.println("I'm the main thread");
      }
}
```


Criação de threads: 2º método

- Implementando interface **`Runnable`**
  - Necessário implementar o método **`run`** (the job to do)
  - Método **`start`** é usado para criar thread, e implicitamente chama **`run`**
  - Objeto *Runnable* passado como o *target* da thread

```
class Simple implements Runnable {
        public void run() {
           System.out.println("I'm a thread");
        }
}
```

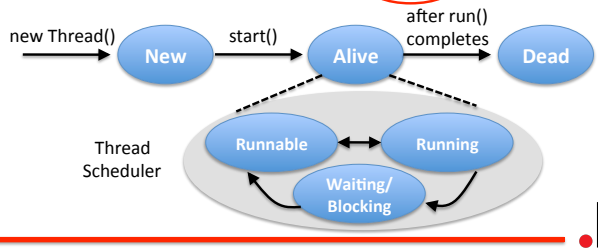
```
public class Second {
     public static void main(String args[]) {
          Runnable s = new Simple( );
          Thread thr = new Thread(s);

          thr.start();
           System.out.println("I'm the main thread");
      }
}
```

Considerações gerais: Java threads

- Etapas para uso de threads em Java
  1. Definir uma nova classe que:
     - Estenda a classe **`Thread`**; ou que
     - Implemente a interface **`Runnable`**
  2. Definir um novo método **`run`** (código da thread)
  3. Criar uma nova instância através de **`new`**
  4. Disparar a thread usando o método **`start`**

```
Thread t = new Thread(runObj);
t.run(); // Permitido, mas não
         // inicia nova thread!
```



Considerações gerais: Java threads

- Métodos permitem *"influenciar"* o escalonamento de threads
  - `public static void sleep(long millis) throws InterruptedException`
  - `public static void yield()`
  - `public final void join() throws InterruptedException`
  - `public final void setPriority(int newPriority)  // entre 1 e 10`
- Uma thread não pode dizer para outra bloquear:
  - Métodos `sleep()`, `yield()` e `join()` afetam a thread corrente que está executando!
  - Antigos `stop()`, `suspend()` e `resume()` são **deprecated**
- Comportamento do escalonador de threads
  - Depende da implementação da JVM, mas em geral…
  - Prioridade das threads em Runnable é menor ou igual à prioridade da thread em Running
    - Se prioridades forem as mesmas, qualquer coisa pode acontecer

POSIX Threads (*Pthreads*)

- Biblioteca de *threads* portável para várias plataformas
  - Conjunto de rotinas C para programação multithread
  - Padrão usado no Linux e em sistemas UNIX em geral
  - IEEE POSIX 1003.1c standard (1995)
    - Especificação, não implementação
- Possui um grande número de funções
  - Pthreads API contém em torno de 100 subrotinas
  - Foco principal é a criação, destruição e sincronização
  - Possui funções que permitem explorar características próprias de um sistema (e.g. prioridades)
    - Atenção: comportamento varia de plataforma para plataforma

Pthreads basics

- Incluir header para biblioteca de *Pthreads*

```
#include <pthread.h>
```

- Variáveis de atributos e descritores: inicializados antes de criar a thread, mas muitos podem ser alterados mais tarde

```
pthread_attr_t attr;       /* thread attributes */
pthread_t thr;             /* thread descriptor */
```

  - Atributos:
    - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
    - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
    - scheduling parameter
    - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent: PTHREAD_INHERIT_SCHED)
    - scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS)
    - guard size
    - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
    - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h)
- Inicialização dos atributos

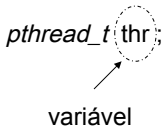
```
pthread_attr_init(&attr);
```

  - Geralmente os valores default são suficientes

Primitiva de criação

```
int pthread_create (
            pthread_t *thr,                  //descritor retornado
            pthread_attr_t *attr;            //atributos (NULL = none)
            void *(*start_routine)(void *), //ponteiro p/ função a ser executada
            void *arg);                      //ponteiro p/ parâmetro (único)
```

Retorna 0, se criou com sucesso;
diferente de 0, caso contrário.



- Identificador
- Atributos
- Função
- Ponteiro p/ parâmetros

- Uma *thread* é uma estrutura de dados → tipo *pthread_t*

Exemplo de criação de *Pthreads*

```
#include <pthread.h>
int g;

void *do_it_1(void *arg) {
    int i, n = *(int *) arg;
    for(i = 0; i < n; i++)
        g = i;
}
void *do_it_2(void *arg) {
    int i, n = *(int *) arg;
    for(i = 0; i < n; i++)
        printf("%d\n", g);
}
int main( int argc, char *argv) {
    pthread_t th1, th2;
    int n = 10;
    pthread_create(&th1, NULL, do_it_1, &n);
    pthread_create(&th2, NULL, do_it_2, &n);
            ....
}
```


Primitivas relacionadas ao término

- Thread termina sua própria execução chamando:

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

  - `status` é um valor de retorno (ou NULL)
  - Chamada implicitamente se thread retorna da função que havia iniciado
- Thread pode ser cancelada por outra thread

```
pthread_cancel(pthread_t thr);
```

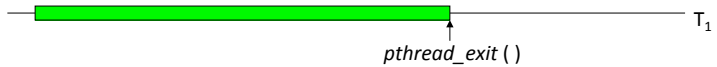
  - `thr` é a thread a ser cancelada
- Thread pai pode esperar por término da thread filha chamando:

```
pthread_join(pthread_t thr, void **res);
```

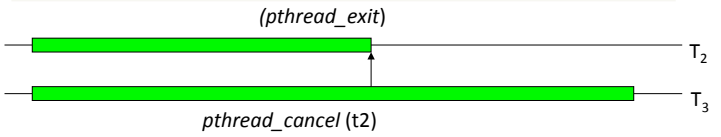
  - `thr` é o decritor da thread filha
  - `res` é endereço da localização do valor de retorno
    - Preenchido quando filha executa pthread_exit

Primitivas relacionadas ao término (destruição)

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




```
int pthread_cancel (pthread_t thread);
```

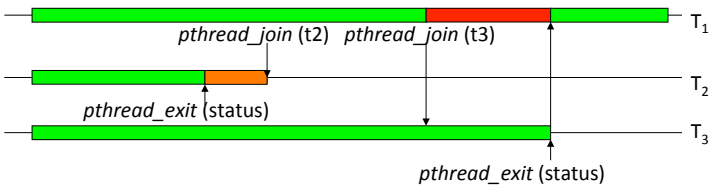


- Terminar = liberar a memória ocupada pela estrutura de dados *pthread_t*
- *exit( )* versus *pthread_exit( )*
  - Terminar um processo (*exit*) é terminar todas as threads (elas vivem "dentro")

Primitivas relacionadas ao término (sincronização)

- Possibilidade de uma *thread* esperar pelo término de outra
  - Identificador da thread pela qual se espera
  - Possibilidade de receber um valor de retorno

```
void pthread_join (pthread_t thread, void **res);
```



→ Válido apenas para *threads* com atributo `joinable` (ao invés de `detached`):
- É o **default**!

Exemplo de espera por *threads*

```c
#include <pthread.h>

int g;
void *do_it_1(void *arg) {
    int i, n = *(int *) arg;
    for(i = 0; i < n; i++)
        g = i;
}
void *do_it_2(void *arg) {
    int i, n = *(int *) arg;
    for(i = 0; i < n; i++)
        printf("%d\n", g);
}
```

```c
int main( int argc, char *argv) {

    pthread_t th1, th2;
    int n = 10;
    pthread_create(&th1, NULL, do_it_1, &n);
    pthread_create(&th2, NULL, do_it_2, &n);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
}
```

Possível usar **`pthread_join`** para recuperar valores da thread que se espera, mas programador deve ser bom em ponteiros em C.

Outro exemplo de espera por *threads*

```c
#include <pthread.h>
void *test_routine (void *arg) {
    double *value;
    value = (double*) malloc(sizeof(double));
    *value = 10.0;
    pthread_exit(value);
}
int main( int argc, char **argv) {
    pthread_t thr;
    double *res;
    int status;
    status = pthread_create( &thr, NULL, test_routine, NULL );
    if (status != 0)
        exit(1);
    status = pthread_join( thr, (void **) &res );
    if (status != 0)
        exit(1);
    printf("resultado retornado: %.2f\n", *res);
    free(res);
}
```


Leituras adicionais

- Andrews, G. "*Foundations of Multithreaded, Parallel and Distributed Programming*", Addison-Wesley, 2000.
  - Capítulo 4 (seção 4.6) e capítulo 5 (seção 5.4 e 5.5)
- Silberschatz, A. and Galvin, P. – *"Operating Systems Concepts"*. Wiley, 8th edition, 2009.
  - Capítulo 4 (seção 4.3)
- Butenhof, D. – "*Programming with POSIX Threads*". Addison-Wesley. 1997.