Recursos do Sistema Linux

Sistemas Embarcados

Conteúdo

1. Acesso a arquivos
2. Processos e Sinais
3. **Threads POSIX e biblioteca *pthreads***
4. Comunicação e Sincronismo entre Processos
5. Programação usando *sockets*
6. Device Drivers

Thread

- ***Threads*** são mecanismos que permitem um programa realizar mais de uma operação "simultaneamente".
- São executadas concorrentemente como processos e o *kernel* do Linux as organiza assincronamente, interrompendo cada *thread* de tempos em tempos de forma que todos tenham chance de ser executados.
- Porém, as *threads* são unidades concorrentes dentro dos processos.

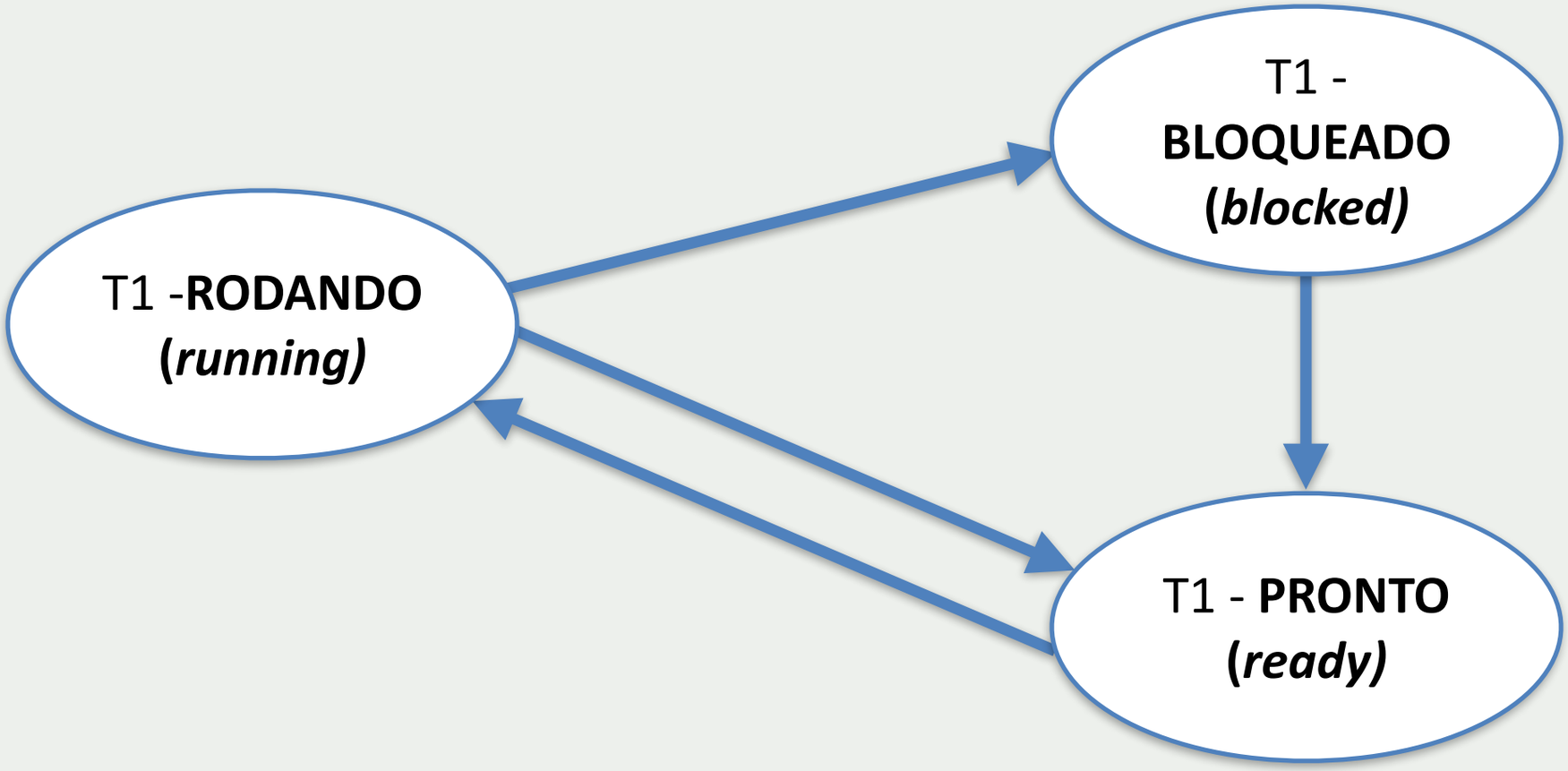
Thread vs. Processo

- Multi-**Processo**
  - A aplicação roda vários programas filho (*child processes*)
  - Cada filho executa sua própria tarefa
  - Cada filho tem seu espaço de memória **protegido**
  - A comunicação é feita por mecanismos (IPC): pipe, sinais, etc
  - Troca de Contexto (Heavyweight)
- Multi-**Thread**
  - Uma só aplicação pode rodar várias *threads*
  - Todas as *threads* **compartilham** o mesmo espaço de memória
  - A comunicação é mais simples por usar memória compartilhada
  - Risco de corrupção de dados
  - Troca de contexto (lightweight)

Processo vs Thread

Ambiente Processo	- espaço de endereçamento (memória) - processo pai / filho - proprietário - arquivos abertos - sinais - estatísticas de uso
Execução Thread	- contador de programa (***PC - program counter***) - apontador de pilha (***SP - stack pointer***) - registradores - estado (execução)

Estados das Threads



- A entidade que realmente se executa é a thread. O processo (ou tarefa) é só o ambiente.
- As threads compartilham as variáveis globais do programa, os descritores abertos, etc. Assim, há necessidade de mecanismos de sincronização

Thread (***pthreads***)

- O GNU/Linux implementa uma API (application program interface) conhecida como ***ptheads*** um padrão IEEE de *threads* denominado POSIX (Portable Operation System Interface).
- Todas as funções de *threads* e tipos de dados são declaradas no arquivo de header **<pthread.h>**. Contudo as funções do *pthread* não são incluídas nas bibliotecas padrões do C. Ao invés disso, deve ser incluído a implementação das funcionalidades do ***libpthread***, então é necessário adicionar à linha de comando o argumento -lpthread para conectar à compilação do código.

Thread

- Cada *thread* de um processo é identificada por um **thread ID** (C/C++tipo pthread_t)
- Cada *thread* criada executa uma "*thread function*", que é uma função ordinária que contém o código que a thread deve executar.
- A *thread* encerra sua execução quando retorna o valor da função.
- No GNU/Linux, as *threads* utilizam apenas um parâmetro do tipo ***void****, e seu retorno também é do tipo ***void****.

Thread

- Criação de *threads:*
  - Função ***pthread_create***:

```
#include <pthread.h>

    pthread_t thread_id;
    pthread_create (&thread_id, const pthread_attr_t *attr, &funcao, void *arg);
```

1) Um ponteiro para variável *pthread_t*, no qual o número de identificação (ID) da nova thread é armazenado.

2) Um ponteiro para o objeto atribuído à thread. Este objeto controla detalhes de como as *threads* interagem com o resto do programa.

3) Um ponteiro para a função da thread, que é uma função ordinária do tipo: void* (*) (void*)

4) Um valor de argumento para a thread do tipo void*; que é passado para a função somente quando a thread é iniciada.

Thread

- Uma chamada na função ***pthread_create*** retorna imediatamente, enquanto a thread original continua a execução do programa. Enquanto isso a nova thread começa executando a thread ***function***. Como o Linux agenda as *threads* assíncronamente, o programa não distingue a ordem de execução das instruções das *threads*.

Thread

- Exemplo: Criação de Threads

```c
#include <pthread.h>
#include <stdio.h>
/* Imprime 'x' em stderr. */
void* print_xs (void* unused)
{
    while (1)
        fputc ('1', stderr);
    return NULL;
}
int main ()
{
    pthread_t thread_id;
    /* Cria um novo thread. A nova therad irá chamar a função print_xs*/
    pthread_create (&thread_id, NULL, &print_xs, NULL);
    /* Imprime 'o' continuamente em stderr. */
    while (1)
        fputc (‘-', stderr);
    return 0;
}
```

Para compilar: **`gcc ExemploThread.c -o ExemploThread -lpthread`**

Thread

- Exemplo: Criação de Threads

```
#include <pthread.h>
#include <stdio.h>
/* Imprime 'x' em stderr. */
void* print_xs (void* unused)
{
    while (1)
        fputc ('x', stderr);
    return NULL;
}
int main ()
{
    pthread_t thread_id;
    /* Cria uma nova thread. */
    pthread_create (&thread_id, NULL, &print_xs, NULL);
    /* Imprime 'o' continuamente em stderr. */
    while (1)
        fputc ('-', stderr);
    return 0;
}
```

Para compilar: `gcc ExemploThread.c -o ExemploThread -lpthread`

Thread

- Passando dados para as thread
- O último argumento da função ***pthread_create*** é ***void *arg***
- É definido desta maneira para poder receber qualquer tipo de variável por ponteiro.
- Portanto é possível passar desde variáveis simples até estruturas de dados. Basta informar o ponteiro
- Para passar vários parâmetros, cria-se uma estrutura de dados e seu ponteiro é passado. Neste caso, a thread terá que reconhecer e saber acessar esta estrutura.
- Uma consequência disto é a possibilidade de criar uma thread com estrutura genérica e comportamento dependente do conteúdo da estrutura de dados passada.

Thread

- Exemplo: Passagem de Parâmetros para Threads

```c
#include <pthread.h>
#include <stdio.h>
struct char_print_parms
{
    char character;
    int count;
};
void* char_print (void* parameters)
{
    struct char_print_parms* p = (struct char_print_parms*) parameters;
    int i;
    for (i = 0; i < p->count; ++i)
        printf("%c\n", p->character);
    return NULL;
}
int main ()
{
    pthread_t thread1_id;
    pthread_t thread2_id;
    struct char_print_parms thread1_args;
    struct char_print_parms thread2_args;
    thread1_args.character = '1';
    thread1_args.count = 3000;
    pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
    thread2_args.character = '-';
    thread2_args.count = 2000;
    pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
    return 0;
}
```

Thread

- Existe um problema neste código.
- A thread principal (que roda a função main) cria as estruturas de parâmetro da *thread* (***thread1_args*** e ***thread2_args***) como variáveis locais, e depois passa esses ponteiros para a estrutura das *threads* que foram criadas.
- Caso a função main termine, simplesmente irá deslocar a memória e limpara a estrutura de dados usada na passagem de parâmetros.
- Se isso ocorrer a memória que contém o parâmetro das estruturas são desalocadas enquanto as outras threads ainda estão acessando.

Thread

- Solução: Juntar as Threads (aguardar o término delas)
- Para forçar que a função main fique ativa até que as outras *threads* do programa terminem utiliza-se uma função que possui funcionalidade equivalente ao ***wait***. Essa função é a ***pthread_join()***, que possui dois argumentos: a ***thread_ID*** da *thread* que irá esperar e um ponteiro para uma variável ***void*** que receberá o valor de retorno da thread.

Thread

- Exemplo: Passagem de Parâmetros para Threads com ***Join***

```c
#include <pthread.h>
#include <stdio.h>
struct char_print_parms{
    char character;
    int count; };
void* char_print (void* parameters) {
    struct char_print_parms* p = (struct char_print_parms*) parameters;
    int i;
    for (i = 0; i < p->count; ++i)
        fputc (p->character, stderr);
    return NULL;
}
int main (){
    pthread_t thread1_id;
    pthread_t thread2_id;
    struct char_print_parms thread1_args;
    struct char_print_parms thread2_args;
    thread1_args.character = '1';
    thread1_args.count = 3000;
    pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
    thread2_args.character = '-';
    thread2_args.count = 2000;
    pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
    pthread_join (thread1_id, NULL);
    pthread_join (thread2_id, NULL);
    return 0;
}
```

Thread

- **Cancelando threads**
- Em circunstâncias normais, uma thread termina quando sua execução finaliza, ou por meio do retorno da função thread ou pela chamada da ***pthread_exit()***. Mas é possível que uma *thread* requisite o término de outra, denomina-se a essa funcionalidade o cancelamento de thread.
- Para cancelar uma thread, deve-se chamar a função ***pthread_cancel()***, passando como parâmetro a *thread_ID* da *thread* a ser cancelada.

Thread

- Exemplo: Cancelamento de Threads - Parte 1/2

```c
# include <stdio.h>
# include <stdlib.h>
# include <pthread.h>
#include <unistd.h>
void *thread_function(void *arg);
int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Não foi possível criar a thread!");
        exit(EXIT_FAILURE);
    }
    sleep(5);
    printf("Cancelando a thread ... \n");
    res = pthread_cancel(a_thread);
    if (res != 0){
        perror("Não foi possível cancelar a thread!");
        exit(EXIT_FAILURE);
    }
    printf("Esperando o fim da execução da thread ... \n");
    res = pthread_join(a_thread,&thread_result);
    if (res != 0){
        perror("Não foi possível juntar as threads!");
        exit(EXIT_FAILURE);
    } exit(EXIT_SUCCESS);
}
```

Thread

- Exemplo: Cancelamento de Threads - Parte 2/2

```
void *thread_function(void *arg){
    int i, res;
    res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if (res != 0){
        perror("Falha na pthread_setcancelstate");
        exit(EXIT_FAILURE);
    }
    res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    if (res != 0){
        perror("Falha na pthread_setcanceltype");
        exit(EXIT_FAILURE);
    }
    printf("Função thread executando. \n");
    for (i = 0; i < 10; i++){
        printf("Thread em execução (%d) ... \n", i);
        sleep(1); }
    pthread_exit(0);
}
```