Laboratório 05: Threads de Aplicação no Linux

Manipulação de Contextos com `ucontext.h`

Objetivo

Este laboratório tem como objetivo introduzir o uso de contextos de execução com a biblioteca `ucontext.h`, permitindo a criação de múltiplas tarefas cooperativas (threads) dentro de um mesmo processo.

API de Contextos POSIX

A API POSIX oferece um conjunto de funções e tipos que permitem manipular diretamente contextos de execução:

- `getcontext(&a)`: salva o contexto atual na variável `a`.
- `setcontext(&a)`: carrega o contexto salvo em `a`.
- `swapcontext(&a,&b)`: salva o contexto atual em `a` e carrega o contexto `b`.
- `makecontext(&a, ...)`: define o ponto de entrada e argumentos de uma nova tarefa.
- `ucontext_t`: tipo de dado que representa um contexto de execução.

Consulte os manuais para mais detalhes: `man getcontext`.

Exemplo: Ping-Pong com Contextos

O código a seguir define duas funções (‘f_ping‘ e ‘f_pong‘) que alternam entre si usando `swapcontext`, simulando a troca de tarefas:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>

#define STACKSIZE 32768

ucontext_t cPing, cPong, cMain;

void f_ping(void * arg) {
   int i;
   printf("%s iniciada\n", (char *) arg);
   for (i = 0; i < 4; i++) {
      printf("%s %d\n", (char *) arg, i);
      swapcontext(&cPing, &cPong);
   }
   printf("%s FIM\n", (char *) arg);
   swapcontext(&cPing, &cMain);
}
```

```
19
20 void f_pong(void * arg) {
21    int i;
22    printf("%s iniciada\n", (char *) arg);
23    for (i = 0; i < 4; i++) {
24       printf("%s %d\n", (char *) arg, i);
25       swapcontext(&cPong, &cPing);
26    }
27    printf("%s FIM\n", (char *) arg);
28    swapcontext(&cPong, &cMain);
29 }
30
31 int main(int argc, char *argv[]) {
32    char *stack;
33
34    printf("Main INICIO\n");
35
36    getcontext(&cPing);
37    stack = malloc(STACKSIZE);
38    if (stack) {
39       cPing.uc_stack.ss_sp = stack;
40       cPing.uc_stack.ss_size = STACKSIZE;
41       cPing.uc_stack.ss_flags = 0;
42       cPing.uc_link = 0;
43    } else {
44       perror("Erro na criação da pilha: ");
45       exit(1);
46    }
47    makecontext(&cPing, (void*)(*f_ping), 1, "\tPing");
48
49    getcontext(&cPong);
50    stack = malloc(STACKSIZE);
51    if (stack) {
52       cPong.uc_stack.ss_sp = stack;
53       cPong.uc_stack.ss_size = STACKSIZE;
54       cPong.uc_stack.ss_flags = 0;
55       cPong.uc_link = 0;
56    } else {
57       perror("Erro na criação da pilha: ");
58       exit(1);
59    }
60    makecontext(&cPong, (void*)(*f_pong), 1, "\tPong");
61
62    swapcontext(&cMain, &cPing);
63    swapcontext(&cMain, &cPong);
64
65    printf("Main FIM\n");
66    exit(0);
67 }
```

Atividades

1. Estude o funcionamento do código acima. Qual o papel de cada chamada de contexto?
2. Modifique o exemplo para incluir uma terceira thread (ex.: ‘Ping2‘).
3. Adicione um mecanismo de escalonamento simples com uma variável global de controle.

4. Compare este modelo de threads com o modelo POSIX (`pthread`).