

## Troca de contextos

### 1. Descrição do problema

Contexto, no âmbito teórico de sistemas operacionais, é tratado como o estado de uma tarefa em um determinado instante de tempo. O código `contexts.c` gera uma simulação de jogo de ping pong e, faz isso justamente com a troca de contexto, colocando uma tarefa dentro do contexto de outra.

### 2. Pseudo-código

para as estruturas `ContextPing` e `ContextPong`

–criar uma pilha de tarefas

–inicializar a estrutura com o contexto ativo no momento

–alocar o tamanho desejado para a pilha

–caso a alocação seja bem sucedida, inicializar os atributos da estrutura

–modificar o contexto da estrutura

—salvar o contexto atual na estrutura em questão e ativar o contexto apontado pela outra

—salvar o contexto da estrutura atual e ativar o contexto apontado pela `ContextMain`

salvar o contexto atual na estrutura `ContextMain` e ativar o contexto apontado pela estrutura `ContextPing`

salvar o contexto atual na estrutura `ContextMain` e ativar o contexto apontado pela estrutura `Pong`

### 3. Saídas geradas

O código imprime o jogo de ping-pong, a cada iteração uma estrutura é responsável pela jogada.

### 4. Estrutura `ucontext_t`

```
typedef struct ucontext_t {
    struct ucontext_t *uc_link;
    sigset_t          uc_sigmask;
    stack_t           uc_stack;
    mcontext_t        uc_mcontext;
    ...
} ucontext_t;
```

O atributo `uc_link` aponta para o contexto que será retomado quando o contexto em curso terminar. Já o `uc_sigmask` representa os sinais que não podem ser acessado no contexto em questão. A pilha usada pelo contexto é armazenada no atributo `uc_stack`. Por fim, o atributo `uc_mcontext` guarda a representação específica do contexto salvo.

Ele também inclui a chamada das threads.

5. <ucontext\_t.h>

Esta biblioteca permite que o usuário faça trocas de contexto entre múltiplas threads de controle dentro de um mesmo processo. As funções disponibilizadas pela biblioteca são:

```
int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);
void makecontext(ucontext_t *ucp, void (*func)(), int argc ...);
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

6. Um pouco mais detalhado

**int getcontext(ucontext\_t \*ucp)**

Esta função inicializa a estrutura apontada por ucp com todo o contexto que está em vigência no momento. Quando queremos recolher o estado de uma tarefa naquele instante, podemos utilizá-la. Em casos de sucesso o retorno é 0.

**int setcontext(const ucontext\_t \*ucp)**

Esta função faz o contrário da anterior, isto é, ela permite que o usuário faça o contexto atual receber o contexto apontado por ucp. Para uma atribuição sem erros de retorno, é imprescindível que o contexto tenha sido recolhido com getcontext ou makecontext. Entretanto, existe uma pequena diferença em como o ambiente trata os contextos obtidos por cada um dos métodos.

Caso o contexto seja obtido por getcontext(), o sistema continua sua execução normalmente. No caso do contexto obtido por makecontext(), ele continua sua execução chamando a função passada como parâmetro deste método. Quando a função em discussão retorna, o sistema prossegue, agora tratando o atributo uc\_link como primeiro da chamada para makecontext(). Caso este atributo seja nulo, isso significa que a thread que precisa ser acessada existe e, portanto, o contexto existe. Em casos de sucesso este método não retorna nada.

**void makecontext(ucontext\_t \*ucp, void (\*func)(), int argc, ...)**

Este método modifica o contexto apontado por ucp e, esta modificação, é definida pela função func chamada no segundo argumento do método. Antes de tentar modificar o contexto, é necessário criar uma pilha que armazenará o futuro contexto. O endereço desta pilha precisa ser setado ao atributo uc\_stack do objeto ucp. Além disso, é necessário definir quem será o contexto sucessor e setar seu endereço no atributo uc\_link do objeto ucp.

**int swapcontext(ucontext\_t \*oucp, const ucontext\_t \*ucp)**

Este método salva o contexto atual na estrutura apontada por oucp e então ativa o contexto apontado por ucp. É muito útil quando precisamos fazer uma troca simultânea de contextos. Essa função foi muito utilizada no código estudado para este projeto, uma vez que a cada instante um jogador (elaborado como estrutura) precisa jogar.

7. Chamadas de métodos da biblioteca <ucontext\_t.h> dentro de contexts.c

Logo que a main é iniciada, uma pilha é declarada e então o seguinte código:

```
getcontext (&ContextPing);
stack = malloc (STACKSIZE) ;
if (stack) {
    ContextPing.uc_stack.ss_sp = stack ;
    ContextPing.uc_stack.ss_size = STACKSIZE;
    ContextPing.uc_stack.ss_flags = 0;
    ContextPing.uc_link = 0;
}
makecontext (&ContextPing, (void*)(*BodyPing), 1, "    Ping");
```

Inicialmente o usuário recolhe o contexto atual na estrutura ContextPing. Como o próximo passo é modificar o contexto recolhido, é preciso criar uma pilha e setar os atributos de ContextPing (conforme explicação do método na sessão 6). O método para modificação de contexto necessita de uma função que defina como essa mudança será concretizada. Neste caso, esta função é BodyPing() (ela foi suprimida para os fins deste relatório):

```
void BodyPing (void * arg) {
    ——— swapcontext (&ContextPing, &ContextPong); // 3 iteracoes
    swapcontext (&ContextPing, &ContextMain) ;
}
```

Este método faz com que o contexto das estruturas sejam invertidos 3 vezes e, finalmente, salva o contexto atual na variável ContextPing, ativando o contexto ContextMain. Esta estrutura de código é repetida para a estrutura ContextPong, da mesma forma, e com os mesmos significados para cada atributo.

Por fim, acontece as últimas trocas de contexto (inversão de jogadores do ping pong simulado) com base na chamada dos seguintes métodos:

```
swapcontext (&ContextMain, &ContextPing);
swapcontext (&ContextMain, &ContextPong);
```

A primeira chamada ativa o que simula o jogador do Ping, e a segunda ativa o jogador Pong.

8. Diagrama do tempo da execução

