Giovanni Pietrangelo
Lucas Silvestre Kloss Teles
Turma S71

Relatório

Trocas de Contexto (*p01*)

O presente relatório refere-se a análise de um código presente no arquivo *contexts.c* (previamente fornecido pelo Professor) que aborda as seguintes funções de contexto:

a) **getcontext(&a)**: salva o contexto atual na variável a .
b) **setcontext(&a)**: restaura um contexto salvo anteriormente na variável a .
c) **swapcontext(&a,&b)**: salva o contexto atual em a e restaura o contexto salvo anteriormente em b .
d) **makecontext(&a, ...)**: ajusta alguns valores internos do contexto salvo em a .

As variáveis **a** e **b** são do tipo ***ucontext_t*** e armazenam contextos de execução.

---

1. Objetivo e Parâmetros

O objetivo de cada função encontra-se acima, na descrição deste relatório. Abaixo, analisando *The GNU Library*, os parâmetros utilizados são exemplificados. Assim, temos para cada uma das funções:

a) *int* **getcontext** *(ucontext_t *ucp):* A função ***getcontext*** inicializa a variável apontada pelo *ucp* (no caso a variável **&a**) com o contexto da *thread* chamada. O contexto contém o conteúdo dos registradores, a máscara de sinal e a pilha atual. A função retorna 0 se bem sucedida, e -1 no caso contrário.

b) *int* ***setcontext*** *(const ucontext_t *ucp)*: A função ***setcontext*** restaura o contexto descrito por ucp (no caso **&a**). O contexto não é modificado e pode ser reutilizado quantas vezes quiser. A função não retorna nada a menos que tenha ocorrido um erro, caso em que retornaria -1.

c) *int* ***swapcontext*** *(ucontext_t *restrict oucp, const ucontext_t *restrict ucp)*: A função ***swapcontext*** é semelhante ao ***setcontext***, mas ao invés de apenas substituir o contexto atual, o último é salvo no objeto apontado por *oucp* (no caso **&a**) como se fosse uma chamada para *getcontext*. O contexto salvo continuaria após a chamada para *swapcontext*.

d) *void* ***makecontext*** *(ucontext_t *ucp, void (*func) (void), int argc, ...)*: O parâmetro *ucp* passado para ***makecontext*** deve ser inicializado por uma chamada para *getcontext*. O contexto será modificado de forma que, se o contexto for retomado, ele começará chamando a função *func* que obtém os argumentos *argc integer* passados. Os argumentos inteiros que devem ser passados devem seguir o parâmetro *argc* na chamada para *makecontext*.

Giovanni Pietrangelo
Lucas Silvestre Kloss Teles
Turma S71

2. Campos da estrutura *ucontext_t*

Analisando um trecho de código, observamos alguns parâmetros relacionados ao ucontext_t:

```
64    ContextPing.uc_stack.ss_sp = stack;
65    ContextPing.uc_stack.ss_size = STACKSIZE;
66    ContextPing.uc_stack.ss_flags = 0;
67    ContextPing.uc_link = 0;
```

a) `uc_stack.ss_sp`: Primeiramente, uma ***uc_stack*** é a pilha usado para o contexto em questão. O valor não precisa ser (e normalmente não é) o ponteiro da pilha. Ela é do tipo ***stack_t*** que descreve uma pilha de sinal. Uma pilha de sinal é uma área especial de memória a ser utilizada como pilha de execução durante manipuladores de sinal. Agora, ***ss_sp*** aponta para a base da pilha de sinal.

b) `uc_stack.ss_size`: Este é o tamanho (em bytes) da pilha de sinais ao qual ***ss_sp*** aponta. É preciso definir isso para o espaço alocado para a pilha.

c) `uc_stack.ss_flags`: Com esta flag é possível informar ao sistema que ele deve ou não usar a pilha de sinais.

d) `uc_link`: Este é um ponteiro para a próxima estrutura de contexto que é usada se o contexto descrito na estrutura atual retornar.

3. Estudando o código

```
59    getcontext (&ContextPing);
```

Salva o contexto atual na variável &ContextPing (inicialização do parâmetro).

```
75    makecontext (&ContextPing, (void*)(*BodyPing), 1, "      Ping");
```

Modifica o contexto especificado por &ContextPing, inicializado pelo *getcontext()*; Quando este contexto é retomado usando *swapcontext()* ou *setcontext()*, a execução do programa continua pela chamada da função BodyPing (passando os argumentos que seguem *argc* na função *makecontext*).

```
77    getcontext (&ContextPong);
```

Salva o contexto atual na variável &ContextPong (inicialização do parâmetro).

```
93    makecontext (&ContextPong, (void*)(*BodyPong), 1, "      Pong");
```

Modifica o contexto especificado por &ContextPong, inicializado pelo *getcontext()*; Quando este contexto é retomado usando *swapcontext()* ou *setcontext()*, a execução do programa continua pela chamada da função BodyPong (passando os argumentos que seguem *argc* na função *makecontext*).

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

Salva o contexto em &ContextMain, e retoma o contexto salvo em &ContextPing (e &ContextPong). Assim, de acordo com o método *makecontext* comentado anteriormente, as funções BodyPing e BodyPong serão executadas.

```
26    swapcontext (&ContextPing, &ContextPong);
44    swapcontext (&ContextPong, &ContextPing);
```

Tendo em vista estas duas linhas, ambas vão alternar entre os contextos salvos em &ContextPing e &ContextPong. Assim, de acordo com o método *makecontext* comentado anteriormente, as funções BodyPing e BodyPong serão executadas toda vez que ocorrer esta troca (dependendo qual contexto for salvo e qual será retomado).

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

Tendo em vista estas duas linhas, ambas vão retornar para o contexto salvo em &ContextMain, encerrando, assim, a execução do programa.

4. Diagrama de tempo da execução