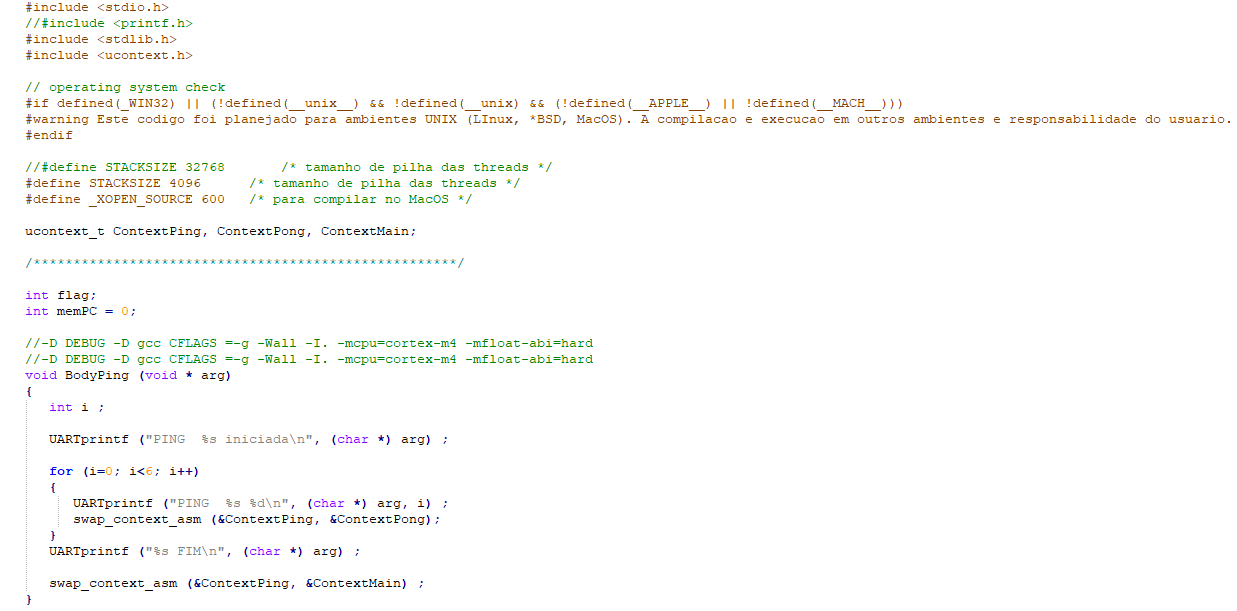
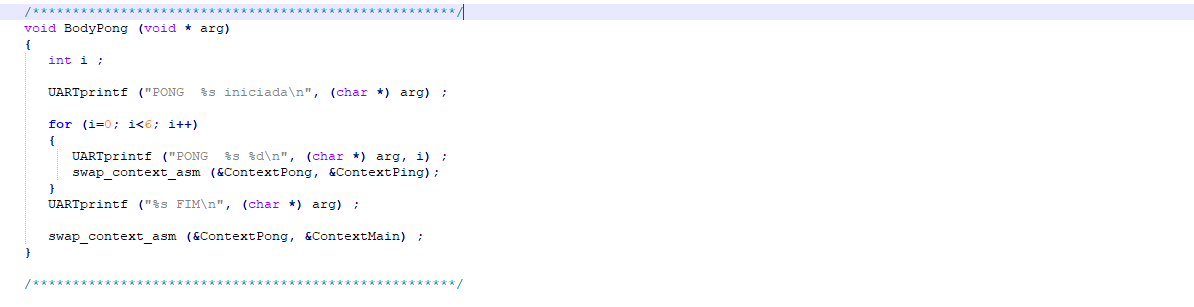
**P1 – Trocas de contexto**

**Henrique Luís Mazzuchetti**

Nesse exercício foi dado esse código fonte para analisar e responder algumas questões:





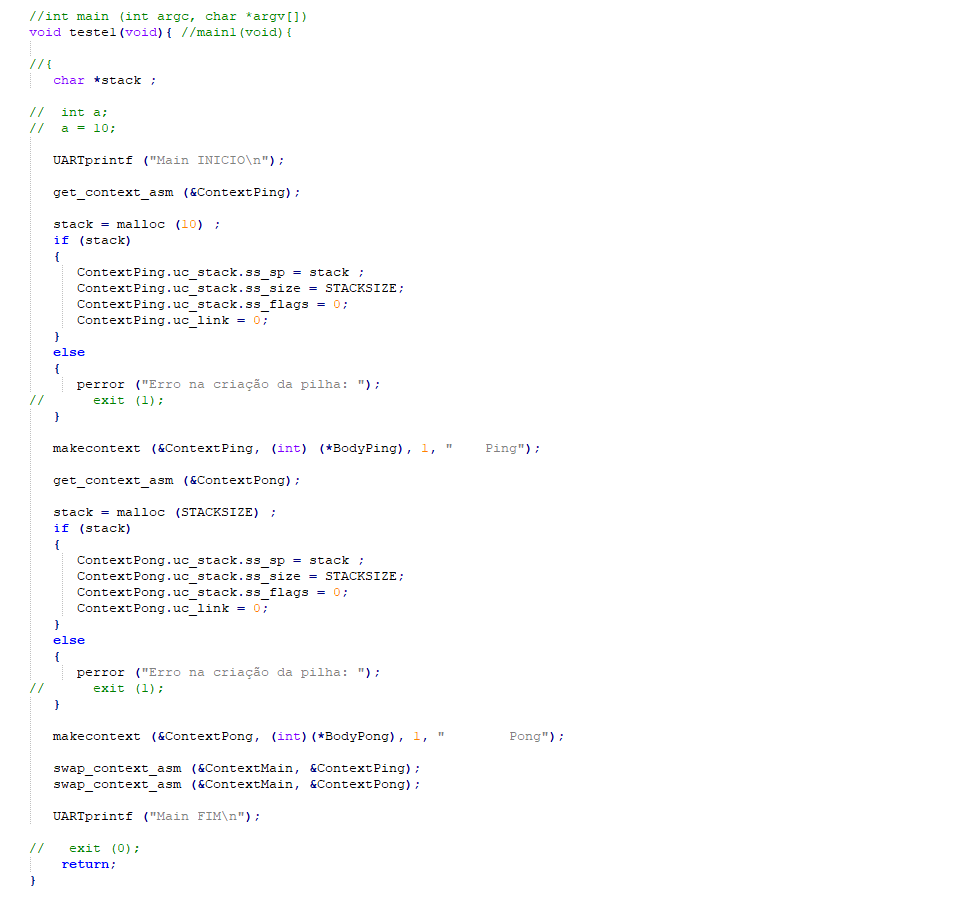


Figura 1 – Código fonte do context.c.

1. **Explique o objetivo e os parâmetros de cada uma das quatro funções abaixo:**

**get\_context\_asm(&a):** Função que trabalha com registradores em Assembly, que captura o contexto atual do projeto e salva esse contexto (valores dos registradores do processador) no endereço apontado pela variável “a”.

**setcontext(&a):** Restaura um contexto que foi salvo numa determinada variável “a” e restaura, ou seja, recoloca nos registradores da CPU. Isso inclui recolocar a posição que foi salva no *program counter* e no *stack pointer*. Significa que é possível pular para uma área de código completamente diferente após a execução dessa função.

**swap\_context\_asm(&a,&b):** Função que trabalha com registradores em Assembly, que salva o contexto atual na variável “a” e restaura o contexto da salvo anteriormente na variável “b”. Basicamente salva o contexto em “a” e salta para o contexto “b”.

**makecontext(&a, …) :** Esta função não cria um contexto. Ela utiliza o contexto salvo em “a” e ajusta alguns valores internos do contexto salvo em “a”.

As variáveis “a” e “b” são do tipo ucontext\_t e armazenam contextos de execução. Essas variáveis dependem da plataforma que está utilizando, quais registradores e flags existem no processador.

1. **Explique o significado dos campos da estrutura ucontext\_t que foram utilizados no código.**

**ucontext\_t:** as variáveis do tipo uconext\_t armazenam contextos de execução.

**uc\_stack.ss\_sp:** seta o StackPointer na posição passada por um ponteiro de caracteres.

**uc\_stack.ss\_size:** informa tamanho total da pilha.

**uc\_stack.ss\_flags:** inicializa as flags que serão utilizadas na pilha.

**uc\_link:** aponta para o contexto que será resumido quando este contexto retornar, ou seja informa o contexto sucessor.

1. **Explique cada linha do código de contexts.c que chame uma dessas funções ou que manipule estruturas do tipo ucontext\_t.**

O programa inicializa fazendo a inclusão da biblioteca de *user context* através de <ucontext.h> e adiciona 3 variáveis ucontext\_t para guardar os contextos ContextPing, ContextPong e ContextMain. São criadas as funções de thread BodyPing, BodyPong e o programa principal. O programa principal chama o getcontext, que armazena o contexto atual na variável ContextPing. Caso ele consiga alocar, o endereço da pilha será colocado no uc\_stack.ss\_sp, o tamanho da pilha no uc\_stack.ss\_size e flags serão ajustadas. Caso não consiga alocar irá retornar erro. Posteriormente a função makecontext é chamada, ajustando o contexto que foi salvo em getcontext e mexendo no valor do *program counter* para que ele aponte para a função BodyPing, e será colocado na pilha apontada pelo stackpointer a string “Ping”. Na sequência é feita a mesma coisa com o ContextPong, passando “Pong” para a pilha.

É feito o swapcontext onde é salvo o contexto atual de execução na variável ContexMain e ativa o contexto salvo em ContextPing. Isso irá acionar a função BodyPing que irá imprimir “Início” e irá fazer um for de 0 a 3 imprimindo a string que recebeu como parâmetro, adicionando “i”, que corresponde a: “Ping 0”, “Ping 1”, “Ping 2” e “Ping 3”. Porém, entre as passagens do laço *for* é feita a troca de contexto, saindo da variável ContextPing e entrando na variável ContextPong. Portando a execução altera entre “Ping” e “Pong”, e no final volta para Main, como pode ser visto na execução:

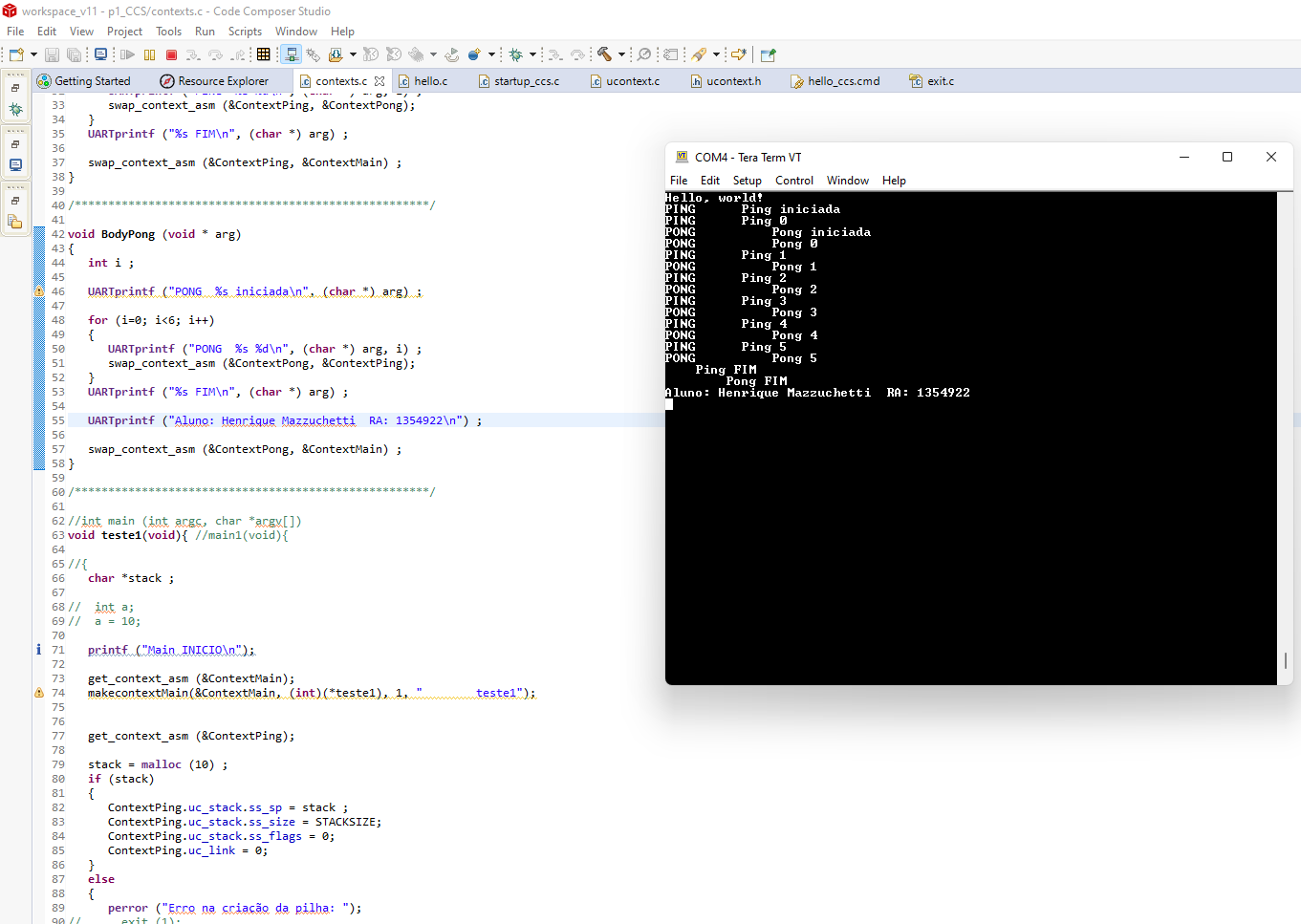


Figura 2 – Screenshot do Debug do CCS com o funcionamento no terminal TeraTerm.

1. **Para visualizar melhor as trocas de contexto, desenhe o diagrama de tempo dessa execução.**

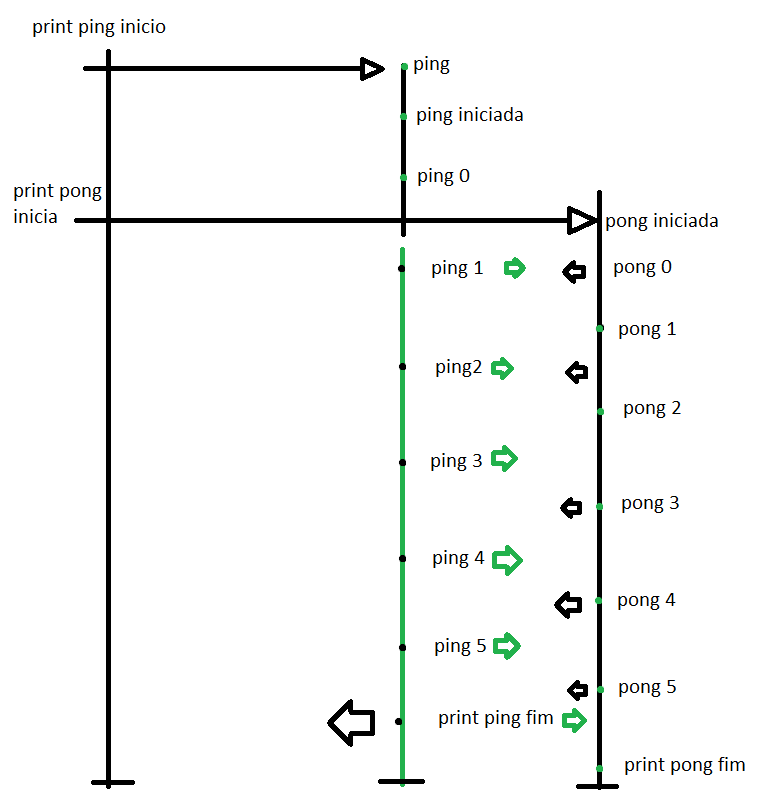


Figura 4 – Diagrama de tempo de execução.