APOSTILA DE EXERCÍCIOS

EA876
Tiago Fernandes Tavares
tavares@dca.fee.unicamp.br

Revisão: 1º. Semestre - 2019

Índice

MÁQUINAS DE ESTADO

Exercício 1
Objetivo: projetar uma máquinas de estado capazes de reconhecer sequências específicas de caracteres

a) Desenhe uma máquina de estados capaz de reconhecer a cadeia de caraceres "FEEC". Marque o estado inicial e os estados que significam "aceitar cadeia".
b) Desenhe uma máquina de estados capaz de reconhecer a cadeia de caraceres "feec"
c) Desenhe uma máquina de estados capaz de reconhecer as cadeias "FEEC" e "feec"
d) Desenhe uma máquina de estados capaz de reconhecer as cadeia "FEEC" sem sensibilidade a maiúsculas e minúsculas, ou seja, "FeEc" também é uma cadeia válida.

Exercício 2
Objetivo: incorporar caracteres opcionais na máquina de estados

a) Projete uma máquina de estados capaz de reconhecer a cadeia "FEEC", mas de tal forma que o "C" seja opcional, ou seja, "FEE" também é uma cadeia válida.
b) Projete uma máquina de estados capaz de reconhecer todas as variações de "FEEC" com qualquer número de espaços entre as letras (ou seja: "F e E C" é uma cadeia válida)

Exercício 3
Objetivo: definir máquinas de estado usando categorias de caracteres

a) Projete uma máquina de estados capaz de reconhecer qualquer sequência de caracteres que represente um número inteiro.
b) Projete uma máquina de estados capaz de reconhecer qualquer sequência de caraceres que represente um número escrito como ponto flutuante.
c) Projete uma máquina de estados capaz de reconhecer qualquer sigla válida (não necessariamente existente) de disciplinas da Unicamp.

Exercício 4
Objetivo: aplicar máquinas de estado para reconhecimento em processos reais

Projete uma máquina de estados que deve identificar se uma sequência de caracteres corresponde a um e-mail da DAC de um aluno da Unicamp (verifique se o formato da sequência de caracteres é válido, não se o e-mail é existente).

Exercício 5
Objetivo: entender como implementar máquinas de estado em C

a) Leia o código state_machine/state_machine.c. Ele implementa uma máquina de estados. Identifique quais são os estados e quais são as condições de transição entre eles.

b) Modifique o código para que, ao invés de encontrar números inteiros, ele encontre vogais.

c) Leia o código state_machine/state_machine2.c. Quais são os estados e quais são as condições de transição entre eles?

EXPRESSÕES REGULARES

Exercício 1
Objetivo: identificar o paralelismo entre máquinas de estado e expressões regulares
a) Desenhe a máquina de estados correspondente à expressão regular (OLA)
b) Desenhea máquina de estados correspondente à expressão regular [Oo](LA)
c) Desenhe a máquina de estados correspondente à expressão regular [(OI)(OLA)]

Exercício 2
Objetivo: projetar expressões regulares capazes de reconhecer sequências específicas de caracteres

a) Escreva uma expressão regular capaz de reconhecer a cadeia de caraceres "FEEC".
b) Escreva uma expressão regular capaz de reconhecer a cadeia de caraceres "feec"
c) Escreva uma expressão regular capaz de reconhecer as cadeias "FEEC" e "feec"
d) Escreva uma expressão regular capaz de reconhecer as cadeia "FEEC" sem sensibilidade a maiúsculas e minúsculas, ou seja, "FeEc" também é uma cadeia válida.

Exercício 3
Objetivo: identificar o paralelismo entre máquinas de estado e expressões regulares com caracteres repetidos e opcionais
a) Desenhe a máquina de estados correspondente à expressão regular TE+STE
b) Desenhe a máquina de estados correspondente à expressão regular O?K
c) Desenhe a máquina de estados correspondente à expressão regular [0]*50
d) Desenhe a máquina de estados correspondente à expressão regular [01]+

Exercício 4
Objetivo: incorporar caracteres opcionais nas expressões regulares

a) Escreva uma expressão regular capaz de reconhecer a cadeia "FEEC", mas de tal forma que o "C" seja opcional, ou seja, "FEE" também é uma cadeia válida.
b) Escreva uma expressão regular capaz de reconhecer todas as variações de "FEEC" com qualquer número de espaços entre as letras (ou seja: "F e E C" é uma cadeia válida)

Exercício 5
Objetivo: definir expressões regulares usando categorias de caracteres

a) Escreva uma expressão regular capaz de reconhecer qualquer sequência de caracteres que represente um número inteiro.
b) Escreva uma expressão regular capaz de reconhecer qualquer sequência de caraceres que represente um número escrito como ponto flutuante.
c) Escreva uma expressão regular capaz de reconhecer qualquer sigla válida (não necessariamente existente) de disciplinas da Unicamp.

Exercício 6

Objetivo: aplicar expressões regulares para reconhecimento em busca por arquivos no computador

Troque a expressão regular em negrito abaixo de tal forma que a linha de comando fique equivalente a *"ls -R *.c"*:

> ls -R | grep -w -E "expressao_regular"

LEX

Exercício 1
Objetivo: entender a estrutura de um programa em lex.

Encontre o programa lex-demo/lexdemo.l. Primeiro, leia o código e marque todas as expressões ou marcações que parecem estranhas. Após:

a) Localize as três partes principais (definições, regras e subrotinas).

b) Encontre as expressões regulares relacionadas a inteiros positivos, inteiros negativos e palavras.

c) Compile (veja as instruções em leiame.txt) e teste o programa.

d) Remova as expressões [[:space:]]+;, [[:^space:]]+;, e [.\n]*; no fim da seção de regras. Desta vez, o que acontece com o programa? Qual é a função dessas regras?

e) Passe as expressões [[:space:]]+;, [[:^space:]]+;, e [.\n]*; para o começo do programa. Novamente, compile e teste. O que aconteceu? Por que?

f) Como um programa em Lex resolve possíveis coincidências entre regras que são ativadas?

Exercício 2
Objetivo: adicionar regras.

Modifique o programa lexdemo.l para que ele seja capaz, também, de reconhecer números em ponto flutuante. Teste o seu programa.

Exercício 3
Objetivo: usar variáveis em programas em Lex

Modifique novamente o programa para que ele retorne somente a soma dos números (tanto inteiros quanto de ponto flutuante) encontrados ao longo da entrada. Teste o programa.

Exercício 4
Objetivo: entender o problema do balanceamento de parênteses.

Faça um programa em Lex (pode usar o anterior como base) que identifica se uma string recebida como entrada tem parênteses balanceados. Teste o programa.

GRAMÁTICAS LIVRES DE CONTEXTO

Exercício 1
Objetivo: analisar expressões regulares como conjuntos de regras de formação

Ligue as regras de formação na coluna da esquerda às suas correspondentes expressões regulares na coluna da esquerda. Após, complete a coluna da direita com a expressão regular que corresponde à regra de formação que restou.

S → a S → Sa	a+
S → vazio S → a S → Sa	[ab]+
S → ab S → Sab	(ab)+
S → A S → AS A → b A → a	a*
S → vazio S → A S → AS A → ab	

Exercício 2
Objetivo: aplicar regras de formação em uma cadeia de caracteres

Considere as regras de formação:

S → vazio
S → ( ) S
S → S ( )
S → (S)

Verifique se as seguintes cadeias de caracteres podem ser geradas por essas regras. Mostre quais regras foram aplicadas (e em que sequência) para gerar a cadeia.
a) ( )
b) ( ) ( )
c) (( ))
d)) ( ) (
e) ( ) (( ))

Exercício 3
Objetivo: definir regras de uma gramática

Defina regras para a gramática capaz de gerar todas as expressões matemáticas que usam +, -, *, / e ( ). Assuma que um "inteiro qualquer" pode ser usado como um símbolo terminal.

Exercício 4
Objetivo: desenhar árvores sintáticas

Desenhe a árvore sintática que permite resolver a expressão matemática abaixo:

5 + 15 – (20 + 5) + 1 * 8 / 9

Para cada grau da árvore, anote o resultado intermediário que surge.

Exercício 5
Objetivo: executar o algoritmo do autômato com pilha
Um dos possíveis algoritmos que permitem resolver gramáticas livres de contexto é o algoritmo shift-reduce (SR). Ele toma como entrada a saída do tokenizador (gerado, por exemplo, por um programa escrito no Lex), e então executa um algoritmo baseado numa pilha (inicialmente vazia) que pode ser desenhado na forma de um fluxograma como mostra a Figura 1:

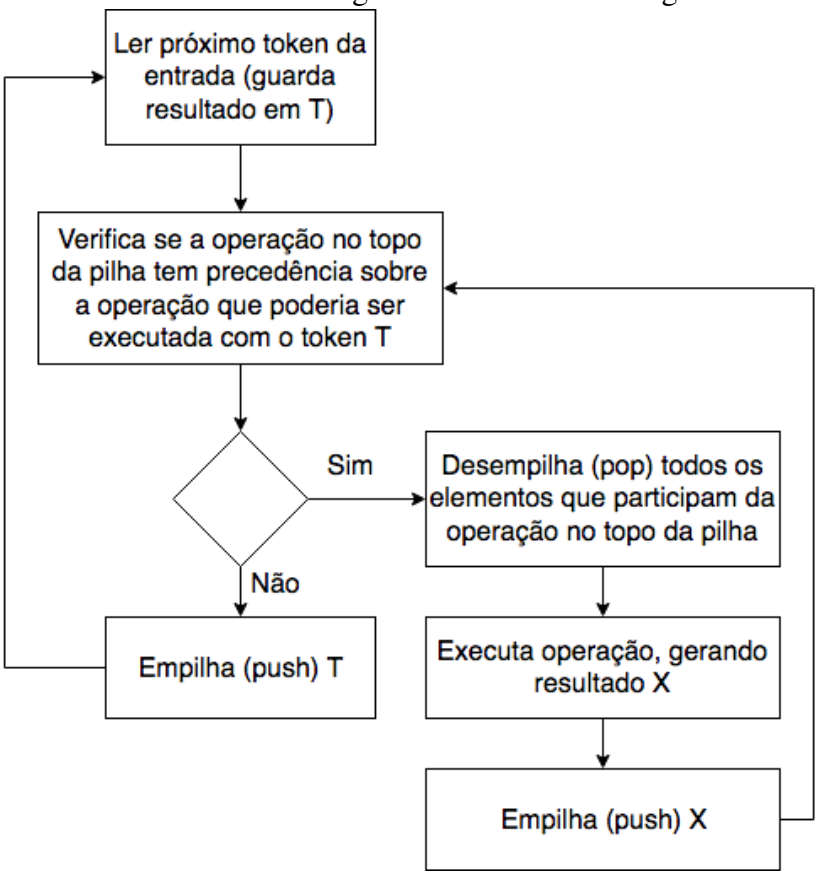


Figura 1: algoritmo shift-reduce

Aplique esse algoritmo, mostrando o conteúdo da pilha, para resolver as seguintes expressões:
a) 3 + 4 + 5 + 3 + 3
b) 3 + 4 * 5 * 3 + 3

YACC

Exercício 1
Objetivo: entender a estrutura básica de um programa em YACC

O programa ./yacc-demo/calc1.y define uma pequena calculadora. Ele é feito em Yacc, e acompanha o código calc1.l. Com base nos dois programas, descubra:

a) É possível dividir o código Yacc da mesma forma que o código Lex foi dividido?
b) Abra o Makefile e descubra em que ordem os programas são compilados.
c) Que códigos são gerados pelo Yacc/Bison?
d) O código Lex tem algumas anotações como return INT, return EOL e return SOMA. Onde esses símbolos (INT, EOL e SOMA) são definidos?
e) Quais regras de formação estão definidas no programa em Yacc? Escreva-as na forma de uma gramática livre de contexto.
f) O que é $$, $1, $2, etc.?
g) Teste o programa. Ele executa corretamente? O que acontece se o programa lê entradas inválidas?
h) O que acontece se a expressão na linha 12 (%left SOMA) for removida?

Exercício 2
Objetivo: propor novas regras para programas em Yacc

Modifique calc1.y de forma a adicionar os operadores de subtração, divisão e multiplicação. Teste seu programa: ele funciona corretamente? A precedência das operações está correta?

Exercício 3
Objetivo: usar símbolos não-terminais para propor regras em Yacc

Modifique novamente calc1.y de forma que a calculadora passe a aceitar parênteses nas expressões.

Dica: primeiro, escreva num papel a regra que você quer implementar e confirme que ela, de fato, ajuda a gerar expressões entre parênteses.

MAKEFILE E BIBLIOTECAS

Exercício 1
Objetivo: entender dependências

Considere o seguinte Makefile:

```
programa : main.c
  gcc -oprograma main.c -lm -ll -O3 -Wall
```

a) Como se chama o arquivo que contém o programa que será gerado pelo processo de compilação?
b) Qual é o arquivo que contém o código-fonte do programa?
c) Qual parte do Makefile indica que o programa deve ser compilado novamente caso seu código-fonte mude?

Exercício 2
Objetivo: entender dependências interligadas

Considere o seguinte Makefile:

```
main : lib0.o lib1.o main.c
  gcc -omain main.c lib0.o lib1.o

lib0.o : lib0.c
  gcc -c lib0.c

lib1.o : lib1.c
  gcc -c lib1.c
```

a) Quais são os arquivos-fonte necessários para compilar o programa inteiro?
b) Em que situações esse Makefile gera um comportamento diferente deste outro, mais compacto? Isso é uma vantagem ou uma desvantagem?

main : lib0.c lib1.c main.c
  gcc -omain main.c lib0.c lib1.c

Exercício 3
Objetivo: entender como funcionam variáveis em Makefiles

Considere o seguinte Makefile:

```
flags = -O3 -lm -ll
compiler = gcc
target = main
$(target) : main.c
```

```
  $(compiler) -o$(target) main.c $(flags)
```

Qual será o comando executado quando o usuário digita make?

Exercício 4
Objetivo: entender como usar Makefiles para acelerar seu processo produtivo

Considere o seguinte Makefile:

```
flags = -O3 -lm -ll
compiler = gcc
target = main
$(target) : main.c
  $(compiler) -o$(target) main.c $(flags)

clean :
  rm $(target)

submit : $(target)
  git add main.c
  git commit -am "Automatic commit"
  git push
```

O que deve acontecer quando o usuário usa os seguintes comandos, nesta ordem:

```
make
make clean
make submit
```

Exercício 5
Objetivo: corrigir um Makefile

O makefile abaixo tem problemas. Encontre todos os problemas dele e proponha as correções correspondentes.

```
flags = -O3 -lm -ll
compiler = gcc
target = main
$(target) : main.c
  $(compiler) -o$(target) main.c lib.o $(flags)

lib.o : lib.c
  $(compiler) -c lib.c

clean :
  rm $(target)
```

Exercício 6
Objetivo: aplicar pré-compilação de código para ocultar implementações

Veja o código que está em linking/estatico. Nele, há um programa principal e uma biblioteca. Suponha que somos responsáveis por fazer a biblioteca imprimir e entregá-la ao cliente. Porém, desejamos que o cliente não tenha acesso a nosso código-fonte. Como uma biblioteca estática (.a) permite fazer isso?

Exercício 7
Objetivo: entender aplicações de bibliotecas pré-compiladas dinâmicas

O código em linking/dinamico mostra uma biblioteca dinâmica. Analise todos os códigos e compile. Ele está usando uma das implementações do mylib.

a) Qual das implementações está sendo usada?
b) Que linha de comando permite re-compilar a biblioteca usando a outra implementação, mas sem re-compilar o programa principal?
b) Quando fazemos isso, o programa principal usa a versão da biblioteca que existia quando ele foi compilado, ou usa a nova versão?

Referências:

- Manual do Make: https://ftp.gnu.org/old-gnu/Manuals/make-3.79.1/html_chapter/make_toc.html
- Tutorial sobre bibliotecas compartilhadas: http://www.microhowto.info/howto/build_a_shared_library_using_gcc.html

MEMÓRIA VIRTUAL

Exercício 1
Objetivo: lembrar-se como variáveis e memória se relacionam

O que será impresso na tela ao fim do programa abaixo:

```
#include <stdio.h>
int main() {
  int a = 0;
  int b;
  int *c;

  a = 15;
  c = &a;
  (*c) = b;
  c = &b;
  (*c) = 10;
  printf("%d %d %d\n", a, b, *c);
  return 0;
}
```

Exercício 2
Objetivo: analisar um possível processo de administração de memória

Discuta com seu grupo a seguinte solução para um processo de administração de memória:

- O sistema de adminstração de memória recebe pedidos de acesso à memória (feitos por um programa) e administra esse acesso.
- Cada programa recebe um "offset" diferente. Pedidos de acesso a posições de memória receberão a adição desse "offset", que é caracerístitico de cada programa.
- Cada programa recebe um "limite superior". Pedidos de acesso a posições de memória acima desse "limite superior" são negados.

a) Essa proposta de administração de memória parece viável, ou seja, parece funcionar sem adicionar novos bugs a um programa? Por que?
b) Essa proposta de adminstração de memória é transparente para o programador (ou seja: o programador não precisa modificar seu código explicitamente para lidar com ela)? Por que?
c) Essa proposta de administração de memória garante que um programa não possa acessar variáveis de outro programa? Por que?

Exercício 3
Objetivo: entender o código assembly gerado para um programa simples e descobrir onde reside o sistema de administração de memória

Considere o seguinte código-fonte e o respectivo código assembly, gerado pelo GCC através do comando gcc -S -fno-asynchronous-unwind-tables, e então execute/responda:

a) Tente ligar as linhas de código C às linhas correspondentes do código assembly

b) O compilador GCC gera código assembly nativo da arquitetura da máquina ou gera um código específico com instruções do sistema operacional?

c) Há linhas específicas no código C ou no código assembly para lidar com a administração de memória?

d) Portanto, em que "parte" do computador está o sistema de administração de memória?

C	Assembly

`int main() {`
`  int a;`
`  int b;`
`  a = 1;`
`  b = 50;`
`  a += b;`
`  return 0;`
`}`
	`.file "simple.c"`
	`.text`
	`.globl main`
	`.type main, @function`
	`main:`
	`pushq %rbp`
	`movq %rsp, %rbp`
	`movl $1, -8(%rbp)`
	`movl $50, -4(%rbp)`
	`movl -4(%rbp), %eax`
	`addl %eax, -8(%rbp)`
	`movl $0, %eax`
	`popq %rbp`
	`ret`
	`.size main, .-main`
	`.ident "GCC: (Ubuntu 5.4.1-2ubuntu1~16.04) 5.4.1 20160904"`
	`.section .note.GNU-stack,"",@progbits`

Mais sobre a interpretação do código assembly:
https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

Sobre aflag -fno-asynchronous-unwind-tables:

https://stackoverflow.com/questions/2529185/what-are-cfi-directives-in-gnu-assembler-gas-used-for

Exercício 4
Objetivo: entender como é possível pausar e retomar um processo

Suponha que estamos emulando um microprocessador.

a) Quais partes do microprocessador devem ser representadas na máquina para que o emulador funcione?

b) Se quisermos "pausar" o emulador, fechar o programa e depois retomar a emulação do ponto em que paramos, quais elementos precisaremos guardar para fazer o nosso snapshot válido?

Exercício 5
Objetivo: entender como é possível comutar entre processos

a) No exercício 3, verificamos que cada programa, ou processo, executa diretamente instruções de máquina, ou seja, ele não é intermediado pelo sistema operacional. Então, como seria possível interrompê-lo para, hipoteticamente, pausá-lo? Em outras palavras, que tipo de procedimento de baixo nível (de máquina) poderia ser usado?

b) Usando o procedimento que você encontrou acima, proponha (conceitualmente: não precisa implementar!) uma maneira de pausar um processo que está executando e comutar para outro processo, que tenha sido pausado anteriormente.

Exercício 6
Objetivo: entender o impacto da comutação entre processos no desempenho de uma máquina

a) A comutação entre processos tem um "custo", chamado de *overhead*. Identifique os elementos que contribuem para esse *overhead*.

b) Em princípio, o programador do sistema operacional pode fazer com que seus processos comutem muito rapidamente ou muito lentamente. O que acontece com o desempenho da máquina (como um todo) se os programas comutam muito rapidamente? O que acontece com a concorrência entre os processos se os programas comumtam muito lentamente? Qual desses cenários é mais próximo de o computador congelar? Qual deles é mais próximo de uma interface que demora a responder?

MEMÓRIA VIRTUAL

Exercício 1
Objetivo: lembrar-se como variáveis e memória se relacionam

O que será impresso na tela ao fim do programa abaixo:

```
#include <stdio.h>
int main() {
  int a = 0;
  int b;
  int *c;

  a = 15;
  c = &a;
  (*c) = b;
  c = &b;
  (*c) = 10;
  printf("%d %d %d\n", a, b, *c);
  return 0;
}
```

Exercício 2
Objetivo: analisar um possível processo de administração de memória

Discuta com seu grupo a seguinte solução para um processo de administração de memória:

- O sistema de adminstração de memória recebe pedidos de acesso à memória (feitos por um programa) e administra esse acesso.
- Cada programa recebe um "offset" diferente. Pedidos de acesso a posições de memória receberão a adição desse "offset", que é caracerístitico de cada programa.
- Cada programa recebe um "limite superior". Pedidos de acesso a posições de memória acima desse "limite superior" são negados.

a) Essa proposta de administração de memória parece viável, ou seja, parece funcionar sem adicionar novos bugs a um programa? Por que?
b) Essa proposta de adminstração de memória é transparente para o programador (ou seja: o programador não precisa modificar seu código explicitamente para lidar com ela)? Por que?
c) Essa proposta de administração de memória garante que um programa não possa acessar variáveis de outro programa? Por que?

Exercício 3
Objetivo: entender o código assembly gerado para um programa simples e descobrir onde reside o sistema de administração de memória

Considere o seguinte código-fonte e o respectivo código assembly, gerado pelo GCC através do comando gcc -S -fno-asynchronous-unwind-tables, e então execute/responda:

a) Tente ligar as linhas de código C às linhas correspondentes do código assembly

b) O compilador GCC gera código assembly nativo da arquitetura da máquina ou gera um código específico com instruções do sistema operacional?

c) Há linhas específicas no código C ou no código assembly para lidar com a administração de memória?

d) Portanto, em que "parte" do computador está o sistema de administração de memória?

C	Assembly

`int main() {`
`  int a;`
`  int b;`
`  a = 1;`
`  b = 50;`
`  a += b;`
`  return 0;`
`}`
	`.file "simple.c"`
	`.text`
	`.globl main`
	`.type main, @function`
	`main:`
	`pushq %rbp`
	`movq %rsp, %rbp`
	`movl $1, -8(%rbp)`
	`movl $50, -4(%rbp)`
	`movl -4(%rbp), %eax`
	`addl %eax, -8(%rbp)`
	`movl $0, %eax`
	`popq %rbp`
	`ret`
	`.size main, .-main`
	`.ident "GCC: (Ubuntu 5.4.1-2ubuntu1~16.04) 5.4.1 20160904"`
	`.section .note.GNU-stack,"",@progbits`

Mais sobre a interpretação do código assembly:
https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

Sobre aflag -fno-asynchronous-unwind-tables:

https://stackoverflow.com/questions/2529185/what-are-cfi-directives-in-gnu-assembler-gas-used-for

Exercício 4
Objetivo: entender como é possível pausar e retomar um processo

Suponha que estamos emulando um microprocessador.

a) Quais partes do microprocessador devem ser representadas na máquina para que o emulador funcione?

b) Se quisermos "pausar" o emulador, fechar o programa e depois retomar a emulação do ponto em que paramos, quais elementos precisaremos guardar para fazer o nosso snapshot válido?

Exercício 5
Objetivo: entender como é possível comutar entre processos

a) No exercício 3, verificamos que cada programa, ou processo, executa diretamente instruções de máquina, ou seja, ele não é intermediado pelo sistema operacional. Então, como seria possível interrompê-lo para, hipoteticamente, pausá-lo? Em outras palavras, que tipo de procedimento de baixo nível (de máquina) poderia ser usado?

b) Usando o procedimento que você encontrou acima, proponha (conceitualmente: não precisa implementar!) uma maneira de pausar um processo que está executando e comutar para outro processo, que tenha sido pausado anteriormente.

Exercício 6
Objetivo: entender o impacto da comutação entre processos no desempenho de uma máquina

a) A comutação entre processos tem um "custo", chamado de *overhead*. Identifique os elementos que contribuem para esse *overhead*.

b) Em princípio, o programador do sistema operacional pode fazer com que seus processos comutem muito rapidamente ou muito lentamente. O que acontece com o desempenho da máquina (como um todo) se os programas comutam muito rapidamente? O que acontece com a concorrência entre os processos se os programas comumtam muito lentamente? Qual desses cenários é mais próximo de o computador congelar? Qual deles é mais próximo de uma interface que demora a responder?

A PILHA (*THE* STACK)

Objetivo: entender como funciona a pilha em um processo.

Exercício 1
Objetivo: lembrar do papel da pilha na execução de uma gramática livre de contexto

Considere a gramática livre de contexto com símbolo inicial S, não-terminal F e terminais *a* e *b*
S → vazio
S → aSa
S → *F*
F → *bbb*
Use o algoritmo shift-reduce para resolver a sequência:

abbba

Exercício 2
Objetivo: traçar uma árvore sintática de um código C

Trace a árvore sintática relativa à execução da linha em negrito no seguinte código:

```
int func(int a, int b) {
  return a + b;
}

int main() {
  int i, j, k;
  i = 0; j = 3;
  k = func(i, j);
  return 0;
}
```

Exercício 3
Objetivo: identificar como variáveis são "declaradas" em código assembly e entender a razão de usar tipos estáticos fortes.

A tabela abaixo mostra o código C de um pequeno programa e o código assembly gerado pelo compilador.

C	Assembly

`int main() {`
`  int a;`
`  int b;`
`  a = 1;`
`  b = 50;`
	`  .file "simple.c"`
	`  .text`
	`  .globl  main`
	`  .type  main, @function`
	`main:`

```
  a += b;
  return 0;
}
```

```
     pushq     %rbp
     movq %rsp, %rbp
     movl $1, -8(%rbp)
     movl $50, -4(%rbp)
     movl -4(%rbp), %eax
     addl %eax, -8(%rbp)
     movl $0, %eax
     popq %rbp
     ret
     .size     main, .-main
     .ident    "GCC: (Ubuntu 5.4.1-
2ubuntu1~16.04) 5.4.1 20160904"
     .section  .note.GNU-stack,"",@progbits
```

a) Qual é a posição de memória em que a variável *a* foi alocada, tomando como base o Base Pointer (rbp)?

b) A posição de memória que será alocada para uma variável é definida durante a compilação do programa ou durante sua execução?

c) Por que é necessário definir os tipos de variáveis no código-fonte de um programa em C?

Exercício 4
Objetivo: entender como C pode implementar o escopo local de variáveis usando a pilha.

No programa abaixo e no resultado de sua compilação em assembly:

a) Quando o programa chega na linha 10, qual é o valor da variável *a* declada na linha 08? Por que ela não se confunde com a variável *a* declarada na linha 02?

b) Desenhe a pilha do programa ao longo da execução do programa. Onde está a variável *a* declarada na linha 08? Onde está a variável *a* declarada na ilnha 02?

c) Como o GCC implementou o escopo local de variáveis?

C	Assembly

01: int func()
{
02: int a;
03: a = 1;
04: return
50;
05: }
06:
07: int main()
	.file "function.c"
	.text
	.globl func
	.type func, @function
	func:
	pushq %rbp
	movq %rsp, %rbp
	movl $1, -4(%rbp)
	movl $50, %eax

```
{                       popq %rbp
08:   int a;            ret
09:   a =               .size     func, .-func
func();                 .globl    main
10:   return 0;         .type     main, @function
11: }               main:
                        pushq     %rbp
                        movq %rsp, %rbp
                        subq $16, %rsp
                        movl $0, %eax
                        call func
                        movl %eax, -4(%rbp)
                        movl $0, %eax
                        leave
                        ret
                        .size     main, .-main
                        .ident    "GCC: (Ubuntu 5.4.1-2ubuntu1~16.04)
                    5.4.1 20160904"
                        .section  .note.GNU-stack,"",@progbits
```

Exercício 5
Objetivo: identificar o sentido de crescimento da pilha na memória do programa

O programa abaixo diz ser capaz de identificar o sentido de crescimento da pilha de um programa (isto é, se a pilha cresce em sentido "crescente" na memória ou se cresce em sentido "decrescente"). Esse programa está em pilha/ver_pilha.c, e pode ser testado, caso necessário.

a) Desenhe o estado da pilha na primeira chamada da função func(). Lembre-se de indicar o espaço reservado para as variáveis de escopo local.

b) Desenhe o estado da pilha após a primeira chamada recursiva da função func().

c) Você acredita que o programa, de fato, será capaz de demonstrar o sentido de crescimento da pilha?

```
void func(int N) {
  int a;
  printf("Stack em %p (%lu)\n", &a, (unsigned long int)&a);
  if (N<10) func(N+1);
}
```

```
int main() {
  func(0);
  printf("\n\n\n");
  return 0;
}
```

Exercício 6
Objetivo: explicitar uma falha de segurança de pilhas

Analise o código abaixo. Como a funcao_maliciosa() está fazendo para encontrar os dados que só deveriam ser acessíveis à partir da função main()? Se achar necessário, desenhe a pilha (ao longo da memória) e a posição que está sendo apontada por *char *d* a cada iteração do laço *for*. Após, execute o programa: ele está em pilha/seguranca_pilha.c.

```
#include <stdio.h>

void funcao_maliciosa() {
  char c, *d;
  d = &c;
  for (int i = 0; i < 150; i++) {
    printf("%c", *d);
    d++;
  }
  printf("\n");
  return;
}

int main() {
  char dado_sigiloso[] = "MINHA SENHA";
  funcao_maliciosa();
  return 0;
}
```

MEMÓRIA DINÂMICA E O HEAP

Exercício 1
Lembrar da diferença entre passagem de argumentos por parâmetro e por referência

O que será impresso na chamada printf()?

```
#include <stdio.h>

void func(int a, int *b) {
  (*b) = 10;
  a = 100;
}

int main() {
  int a, b;
  a = 0; b=0;
  func(a, &b);
  printf("%d %d\n", a, b);
  return 0;
}
```

Exercício 2
Entender o a relação da persistência de memória em pilhas e a função malloc()

Considere o programa abaixo (heap/factory.c)

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

typedef struct {
  int matricula;
  float salario;
} reg, *registro;

void novo_registro(registro *r) {
  (*r) = (registro) malloc(sizeof(reg));
}

void remover_registro(registro *r) {
  free (*r);
}

void imprimir_registro(registro *r) {
  printf("Imprimindo registro\n");
```

```
  printf("Matricula: %d\n", (*r)->matricula);
  printf("Salario: %f\n", (*r)->salario);
}

int main() {
  registro r;
  novo_registro(&r);
  r->matricula = 1324;
  r->salario = 1937.34;
  imprimir_registro(&r);
  remover_registro(&r);
  return 0;
}
```

a) Qual é o estado da pilha do programa assim que a função novo_registro() é chamada?
b) Qual é o estado da pilha do programa após o término da função novo_registro()?
c) Se a memória alocada com malloc() estivesse localizada na pilha do programa, o que aconteceria com ela quando a função novo_registro() retorna?

Exercício 3
Entender as interfaces de programação do processo de alocação dinâmica de memória e os problemas envolvidos.

Existem muitos algoritmos diferentes para implementar o processo de alocação dinâmica de memória. Neste exercício, lidaremos com um caso simples. Em nosso exemplo, a administração de memória é implementado como um vetor de ponteiros para blocos de memória. Cada bloco de memória é um trecho contínuo de tamanho conhecido. – ou seja, um vetor h[n] tal que h[n] é um ponteiro para o n-ésimo bloco da memória. Neste exercício, cada bloco da memória tem tamanho (arbitrariamente escolhido) 10. O algoritmo de alocação funciona da seguinte forma:

INICIALIZAÇÃO (quando programa é inicializado): marca todos os segmentos de memória como "livre".
PASSO 1: varre vetor h[n] até achar o primeiro bloco marcado como "livre"
PASSO 2: marca bloco h[n] como "usado"
PASSO 3: retorna ponteiro para a região de memória apontada por h[n]

O algoritmo de liberação funciona da seguinte forma:
PASSO 1: marca memória a ser liberada (h[n]) como "livre".

a) O algoritmo usa somente blocos de 10 bytes. Qual a fração da memória que seria desperdiçada se todas as variáveis alocadas fossem do tipo char (ou seja, de tamanho 1 byte)?
b) Qual seria uma possível solução para conseguir 100 bytes de memória usando o algoritmo proposto?

Exercício 4
Aplicar alocação dinâmica de memória e entender o problema de fragmentação de memória

Neste algoritmo, mais complicado que o anterior, h[n] é um vetor (grande) cuja posição n é um ponteiro para o primeiro byte alocado na n-ésima alocação de memória. Precisamos de um ponteiro auxiliar, P, que aponta para o primeiro byte livre da memória.

INICIALIZAÇÃO: P recebe 0
PASSO 1: procura próxima posição de h[] (m)que não foi associada a nenhum bloco de memória
PASSO 2: h[m] recebe P, e marca h[m] como "usada".
PASSO 3: variável RP guarda uma cópia de P (RP = P);
PASSO 4: soma o número N de bytes alocados a P (portanto, P passa a apontar para h[P+N])
PASSO 5: retorna RP

A liberação de memória funciona da seguinte forma:
PASSO 1: procura pelo ponteiro para a memória a ser liberada no vetor
PASSO 2: marca a memória a ser liberada como "livre"

a) Verifique se o algoritmo de alocação a seguir resolve satisfatoriamente os problemas encontrados no exercício anterior.
b) Qual é o estado do mapa de memórias após a seguinte sequência de alocações?
a = malloc(1); b = malloc(3); c = malloc(6); free(a); free(b);

Exercício 5

Objetivo: aplicar o conceito de segmentação para prever o comportamento de malloc() e free()

Leia atentamente o código abaixo (malloc/malloc_reuso.c) e tente acompanhar as saídas correspondentes. A última linha da saída foi removida. Após, responda às questões abaixo.

malloc_reuso.c	**SAÍDA (a última linha foi removida)**

`#include <stdlib.h>`
`#include <stdio.h>`
 	
`int main() {`
`  void *a;`
`  void *b;`
`  void *c;`
 	
`  printf("Reuso com Malloc e Free:\n");`
`  a = malloc(1000);`
`  printf("%p\n", a);`
`  free(a);`
`  a = malloc(1000);`
`  printf("%p\n", a);`
`  free(a);`
`  printf("Reuso e segmentacao:\n");`
	`Reuso com Malloc e Free:`
	`0x1d6f420`
	`0x1d6f420`
	`Reuso e segmentacao:`
	`0x1d6f420 0x1d6f810 0x1d6fc00`
	`0x1d6f420 0x1d6fff0 0x1d6fc00`

```
  a = malloc(1000);
  b = malloc(1000);
  c = malloc(1000);
  printf("%p %p %p\n", a, b, c);

  free(a);
  free(b);
  a = malloc(1500);
  b = malloc(1000);
  printf("%p %p %p\n", a, b, c);

  free(a);
  free(b);
  a = malloc(2500);
  printf("%p %p %p\n", a, b, c);

  free(a);
  free(c);
  return 0;
}
```

a) malloc() reaproveita a memória liberada por free()? Quais saídas evidenciam isso?
b) Em que parte do programa criamos fragmentação na memória? Que linha da saída evidencia isso?
c) Qual das hipóteses abaixo é válida:

1) malloc() retorna sempre um ponteiro para a primeira posição de memória livre.

2) malloc() retorna um ponteiro para a primeira posição de memória que tem espaço livre no mínimo do tamanho desejado

3) malloc() retorna um ponteiro para a posição de memória que tem mais espaço livre.

c) Qual deveria ser o conteúdo da última linha da saída?

Exercício 6
Entender a diferença entre as funções alloca() e malloc()

O texto abaixo foi extraído da saída do comando **man alloca** executado num terminal Linux:

```
NAME
       alloca - allocate memory that is automatically freed

SYNOPSIS
       #include <alloca.h>

       void *alloca(size_t size);
```

```
DESCRIPTION
       The alloca() function allocates size bytes of space
in the stack frame of the caller.  This temporary space is
automatically freed when the function that called alloca() returns
to its caller.

RETURN VALUE
       The alloca() function returns a pointer to the beginning of
the allocated space.  If the allocation causes stack overflow,
program behavior is undefined.
```

a) Onde a memória alocada usando alloca() é posicionada?
b) O que acontece com a memória alocada por alloca() quando a função que a chamou termina? Esse é o mesmo comportamento da memória alocada por malloc()?
c) É preciso usar free() para desalocar a memória alocada por alloca()?

Exercício 7
Entender um dos riscos de usar alloca() em um programa

Considere os dois programas abaixo:

```
/* Programa 1 */
void func() {
  char *a = (char*) alloca(10000000); /* Aloca 10 MB de memoria */
}

int main(){
  for (int i=0; i<100000; i++) {
    func();
  }
return 0;
}

/* Programa 2 */
int main() {
  char *a;
  for (int i=0; i<1000000; i++) {
    a = (char*) alloca(10000000);
  }
}
```

Por que o programa 2 causa segmentation fault e o programa 1 não?

TIPAGEM ESTÁTICA E TIPAGEM DINÂMICA

Objetivo: Entender as aplicações e overheads relacionados à implementação de tipagem dinâmica e sistemas de garbage collection

Algumas linguagens de programação usam tipos estáticos, isto é, uma variável é inerentemente associada ao tipo de dado que ela armazena. C e C++ são duas linguagens desse tipo. Por saber de antemão o tipo de dado que será armazenado, o compilador "sabe" quanto espaço deve ser alocado para a variável, e isso permite uma série de otimizações interessantes. Por outro lado, outras linguagens, como Python e MatLab, usam tipos dinâmicos, isto é, uma variável pode armazenar vários tipos de dados durante a execução do programa. Isso permite que o programador não tenha que explicitar a reserva de memória da variável. Nesta aula, estudaremos as vantagens e desvantagens de cada uma dessas abordagens.

Exercício 1
Objetivo: entender o que é tipagem estática e o que é tipagem dinâmica

Em algumas linguagens de programação, como Python ou MatLab, é possível realizar operações de atribuição conforme o seguinte programa:

A = 50
A = '70'
A = 'Ola!'
A = 50.0

a) Qual é o tipo de dado armazenado em A após cada uma das operações de atribuição?
b) Essa sequência de operações é permitida em Python e MatLab. Porém, em C ela é proibida. Por que é necessário que essa sequência de operações seja proibida em C?

Exercício 2
Objetivo: enumerar as informações que devem ser armazenas junto aos dados de variáveis de tipo dinâmico

Em Python, a operação soma, realizada em tipos inteiros, retorna a soma numérica dos operandos. Porém, a operação soma pode ser executada em strings, retornando a concatenação entre duas strings:

```
>>> a = 10
>>> b = 50
>>> a + b
60
>>> a = 'ola '
>>> b = 'mundo'
>>> a + b
'ola mundo'
```

a) Suponha que os tokens INT, VARIAVEL, STRING, SOMA e IGUAL foram definidos usando expressões regulares. Como seria a gramática livre de contexto que implementa a operação soma? Se precisar, crie mais tokens ou deixe de usar alguns tokens que não forem necessários.
b) A struct abaixo é adequada para representar uma variável de tipo dinâmico? Se faltarem informações, adicione mais linhas ao código:

```
typedef struct {
  void *dados;
  char *rotulo;
  int tipo;
} variavel_dinamica;
```

Exercício 3
Objetivo: entender o processo de criação de uma variável dinâmica no sistema

a) Ordene as instruções abaixo para implementar a operação de atribuição de uma variável dinâmica. Lembre-se que uma atribuição tem a forma A = X, onde A é o rótulo da variável que receberá o dado e X é o dado a ser atribuído. Se precisar adicione ou remova instruções:

1. Copia os dados contidos em X para o espaço de dados de Y
2. Aloca espaço para os dados de Y, considerando o tipo de X
3. Aloca espaço para nova struct Y do tipo variavel_dinamica
4. Aloca espaço para o rótulo de Y
5. Atribui valor 'A' ao rótulo de Y

b) Quais dessas instruções não são necessárias na operação de atribuição em linguagem C?

Exercício 4
Objetivo: argumentar sobre vantagens de linguagens com tipagem dinâmica

Linguagens como MatLab e Python têm tipos dinâmicos. Com base em sua experiência, enumere aspectos que tornam as linguagem com tipos dinâmicos interessantes. Enumere também a situação em que cada aspecto se aplica (por exemplo: "aspecto: não preciso pensar em como os dados são representados pela máquina. Situação: quando estou fazendo um código que é um protótipo matemático").

Exercício 5
Objetivo: argumentar sobre desvantagens de linguagens com tipagem dinâmica

Linguagens como MatLab e Python têm tipos dinâmicos. Com base em sua experiência, enumere aspectos que tornam as linguagem com tipos dinâmicos pouco interessantes. Enumere também a situação em que cada aspecto se aplica.

Exercício 6

Objetivo: argumentar sobre vantagens e desvantagens de linguagens com tipagem dinâmica

Usando os argumentos e situações que você construiu nos exercícios 4 e 5, identifique, para cada um dos casos abaixo, quais situações são relevantes. Usando esses argumentos, decida se cada um dos casos deveria ser implementado em Python ou C.

1) Um processador de efeitos de vídeo em tempo real
2) Um webserver
3) Um servidor de bancos de dados que acessa elementos no disco rígido
4) Um jogo de computador que emula o comportamento físico de todos os objetos em cena

GARBAGE COLLECTOR

Exercício 1
Objetivo: entender o processo de atribuição por referência

Quando lidamos com dados muito grandes (por exemplo, matrizes), operações de atribuição podem ser mais rápidas se a cópia de uma variável para outra for implementada usando somente a cópia do ponteiro para o espaço de dados. Em Python, isso pode levar a situações como esta:

```
>>> A = [1, 2, 3, 4, 5]
>>> B = A
>>> A[0] = 4
>>> B
[4, 2, 3, 4, 5]
```

Na segunda linha, a atribuição de A para B não causou a replicação da lista de valores. Ao invés disso, B passou a conter um ponteiro para os dados anteriormente representados por A.

Qual será o resultado da execução, em Python, de:

```
A = [1, 2, 3, 4, 5]
B = A
B[0] = 90
A
```

Exercício 2
Objetivo: entender o processo de contagem de referências

a) O código abaixo inicia com atribuição de uma lista para A. Nele, após cada linha, anote quantas variáveis contém ponteiros para o espaço de memória inicialmente alocado para os dados de A:

```
A = [1, 2, 3, 4, 5]          ← 1 referência
B = A
C = A
C = 100
A = 0
B = A
```

b) Após qual linha seria seguro desalocar a memória utilizada pela lista que foi inicialmente alocada em A? É possível justificar essa decisão usando a contagem de referências?

c) Qual é a complexidade computacional de determinar, usando contagem de referências após uma linha executada, se é possível desalocar um espaço de memória?

Exercício 3

Objetivo: entender quando o processo de contagem de referências falha

Considere o código Python abaixo e conte o número de referências aos dados de A em cada passo:

A = [1, 2, 3, 4, 5] ← 1 referência
A[3] = A
A = None

Exercício 4
Objetivo: entender referências circulares

Considere o código Python abaixo:

A = dict()
B = dict()
A[“proximo”] = B
B[“proximo”] = A

a) Construa um grafo mostrando as referências a A e B que existem nesse código

b) Após executar as linhas:
A = None
B = None

o sistema de contagem autorizaria a desalocação da memória relacionada a A e B?

c) Seria possível “descobrir” que A e B não são referenciadas se fosse possível caminhar por todas as variáveis dinâmicas do sistema, como um grafo, e determinar se A e B podem ser alcançadas à partir de alguma referência válida? Qual é a complexidade computacional dessa solução?

PROCESSOS

Objetivo: tonar-se capaz de criar e administrar processos em um programa

Observação: para estes exercícios, use os programas no diretório "processos" do repositório da disciplina.

Exercício 1
Objetivo: usar o comando ps em Linux

Execute o comando ps -a em um terminal Linux.
a) Qual é o significado da última coluna mostrada?
b) Qual é o significado da primeira coluna?

Exercício 2
Objetivo: encontrar um processo específico usando ps e grep

Veja o código-fonte de e01.c no diretório processos do repositório da disciplina.
a) O que ele deve fazer?
b) Compile e execute o programa, mas não dê nenhuma entrada. Em um outro terminal, use o comando ps -A | grep *algum_padrao* para encontrar o PID do seu programa. Troque *algum_padrao* para um padrão que você achar mais adequado.
c) Use o comando kill PID para encerrar o seu programa.

Exercício 3
Objetivo: criar um novo processo através da função fork()

Veja o código-fone de e02.c no diretório de processos do repositório da disciplina.
a) O que o comando fork() faz? Se precisar, use o man fork para ler o manual da função.
b) O que significa processo-pai? O que significa processo-filho?
c) Use o comando ps -A | grep (...) para encontrar o PID do processo-pai e do processo-filho. O PID do processo-filho mostrado pelo ps -A corresponde ao que é retornado pela função fork?
d) Você consegue identificar um comportamento estranho quando o programa termina? Qual é esse comportamento?

Exercício 4
Objetivo: usar o comando waitpid() para sincronizar dois processos

Veja o código-fonte de e03.c.
a) O que a função waitpid() faz? Se precisar, use o man waitpid para ler o manual da função.
b) O comportamento estranho observado anteriormente foi resolvido? Tente explicar o comportamento e também a solução.

Exercício 5
Objetivo: verificar se dois processos compartilham a mesma pilha ou o mesmo heap

a) Reflita com seu grupo: dois processos (pai e filho) devem compartilhar a mesma pilha e/ou o mesmo heap? Até esse momento, quais são as evidências que você pode usar para dar apoio a essa hipótese?
b) Faça um programa de computador (pode se basear nos anteriores) que demonstra se a pilha de dois processos está no mesmo endereço e se o heap de dois processos está no mesmo endereço.
c) Com base nas suas evidências, um processo pode acessar variáveis de outro processo? Isso também se aplica a variáveis declaradas antes do fork()?

Exercício 6
Objetivo: extrapolar conhecimentos sobre o compartilhamento de pilha e heap em processos

Veja o código-fonte de e04.c. Com base no que você inferiu sobre o comportamento de pilhas e heaps em sistemas multi-processo, o que você acha que será impresso ao fim do programa? Depois, execute o programa e verifique se sua hipótese se confirma ou não.

COMUNICAÇÃO INTER-PROCESSOS (IPC) – SIGNALS E MMAP

Objetivo: entender como diferentes processos podem se comunicar e compartilhar recursos
Nota: os códigos-fonte para estes exercícios estão o diretório ipc.

Exercício 1
Objetivo: entender como processos podem se comunicar enviando sinais.

Analise o código fonte de sinais.c.
a) Identifique todas as funções que você não tem certeza sobre o que fazem.
b) Compile e execute o programa. À partir das saídas que foram escritas, o que é possível inferir sobre essas funções?
c) A função sigquit() executa no processo pai ou no processo filho? Ela é iniciada por uma ação no processo pai ou no processo filho?

Exercício 2
Objetivo: entender o funcionamento de mmap

Analise o código fonte de memory_map.c
a) A variável (*b) do processo filho é a mesma (*b) do processo pai? Execute o programa e confirme ou rejeite a sua hipótese.
b) Use as manpages (man mmap) para entender o significado dos parâmetros de mmap().

Exercício 3
Objetivo: entender como processos revezam nos processadores da máquina

Analise o código fonte de divide_work.c.
a) Quantos processos-filho são gerados? Qual é a constante que define esse número?
b) O trabalho realizado pelos processos-filho é o mesmo realizado pelo processo-pai? Se não, qual é a diferença?
c) Faça uma hipótese sobre a ordem em que os comandos printf() serão executados. Depois, execute o programa algumas vezes. Sua hipótese se confirma? Qual é a ordem em que os processos são executados? Realize o mesmo teste, variando o valor de N.

Exercício 4
Objetivo: entender o que significam condições de corrida

Analise o código fonte de memory_map_divide_work.c.
a) Em que ordem os processos-filho deverão terminar?
b) O que deverá ser impresso na tela na última linha do programa?
c) O comportamento do programa é consistente, isto é, é o mesmo toda execução? Essa consistência ou inconsistência é responsabilidade do programador ou do sistema operacional?

Exercício 5

Objetivo: entender situações em que condições de corrida podem ser significativamente prejudiciais ao programa

Analise o código fonte de memory_map_2.c.
a) Antes de executar o programa, encontre o ponto em que há uma condição de corrida, isto é, em que a ordem de execução dos processos irá alterar o resultado do programa.
b) Execute o programa. O que acontece?

COMUNICAÇÃO INTER-PROCESSOS 2 - PIPES

Objetivo: entender como usar pipes e como funciona o padrão produtor-consumidor
Nota: os códigos-fonte para estes exercícios estão o diretório pipedemo.

Exercício 1
Objetivo: entender como usar pipes em POSIX

a) Olhe o código-fonte de contador.c. O que o programa faz?
b) Olhe o código-fonte de pipe_cont.sh. O que esse script faz?

Exercício 2
Objetivo: entender como usar pipes como entrada à partir de códigos C

Analise o código fonte de pipein.c.
a) O que a função popen() faz? O que ela retorna? Quais são os parâmetros da função?
b) O que a função pclose() faz?
c) Existem diferenças de comportamento entre pipe_cont.sh e pipein.c? Se sim, quais?

Exercício 3
Objetivo: entender como usar pipes como saída à partir de códigos C

Analise o código fonte de pipeout.c. Desenhe, para ele, um diagrama de blocos, no qual cada bloco representa um processo e as setas representam os dados que transitam entre os processos. Se precisar, use um símbolo de cilindro para representar dispositivos de armazenamento permanente.

Exercício 4
Objetivo: entender a importância de usar pipes para redirecionar entradas e saídas para dividir tarefas complexas

Analise o código fonte de ffmpegpipe.c
b) Quais são as entradas e saídas do programa ffmpeg? Se precisar, use a documentação de ffmpeg para entendê-lo melhor.
b) Quais são as entradas e saídas de ffmpegpipe.c?
c) Desenhe um diagrama de blocos usando as mesmas instruções do exercício 3.

Exercício 5

Objetivo: entender como usar pipes em processos-pai e processos-filho

Analise o código-fonte de fork_and_pipe.c.
a) Encontre todas as funções ou identificadores desconhecidos.
b) Faça um diagrama de blocos do sistema inteiro, usando as mesmas instruções do exercício 3.
c) O programa fork_and_pipe.c foi escrito usando um paradigma chamado produtor-consumidor. No seu diagrama de blocos, identifique o produtor e o consumidor.

Exercício 6
Objetivo: aplicar o paradigma produtor consumidor para resolver problemas em Linux

Usando as ferramentas wc, grep e ls, proponha uma linha de comando que permita contar o número de arquivos na pasta /home/ que têm números no nome. Desenhe um diagrama de blocos da sua solução.

THREADS

Objetivo: entender como threads funcionam num programa em C

Exercício 1
Objetivo: criar uma nova thread através da função pthread_create()

Veja o código-fone de 00-analysis.c no diretório *threading* do repositório da disciplina.
a) O que o comando pthread_create() faz? Se precisar, use o man para ler o manual da função.
b) Verifique no seu computador se a criação de uma nova thread leva ao aparecimento de um novo PID. Se necesário, modifique o programa 00-analysis.c para fazer essa investigação.

Exercício 2
Objetivo: analisar o comportamento de threads quando ao compartilhamento de pilha e heap

Inspirando-se no programa 00-analysis.c, faça um programa que permita identificar se uma thread comparilha, com o programa principal, a pilha e o heap.

Exercício 3
Objetivo: prever o comportamento de programas multithread

Verifique o código fonte de 01-intro.c.
a) Antes de executá-lo, tente inferir o que ele deve imprimir na tela.
b) Depois, compile e execute o programa e tente explicar o comportamento que você encontrou.

Exercício 4
Objetivo: resolver problemas de compartilhamento de memória em threads

Leia o código fonte de 02-memoria.c.
a) Antes de executá-lo, tente inferir qual problema (dentre os encontrados em 01-intro) são resolvidos neste código. Como esses problemas são resolvidos?
b) Qual problema ainda existe?
c) O que o programa deve imprimir na tela?

Exercício 5
Objetivo: entender a função pthread_join()

Leia o código fonte de 03-join.c.
a) Antes de executá-lo, tente inferir o que ele deve imprimir na tela.
b) O que a função pthread_join() faz? Qual seria a função equivalente quando pensamos em processos?

PROGRAMAÇÃO MULTITHREAD – THREAD POOL

Objetivo: entender como organizar um programa na forma de um "thread pool"

Exercício 1
Objetivo: entender o funcionamento de um mutex

Analise o código de mutex_demo.c.
a) O que acontece numa chamada a pthread_mutex_lock()?
b) O que acontece se pthread_mutex_lock() é chamada em uma thread e, logo em seguida, é chamada em outra thread?
c) O que acontece numa chamada a pthread_mutex_unlock()?
d) O que acontece se uma thread não executa a chamada pthread_mutex_unlock() depois de sua chamada a pthread_mutex_lock()?
e) Modifique o programa mutex_demo.c de forma que ele trave em uma chama a pthread_mutex_lock() e nunca seja encerrado. Teste sua solução.

Exercício 2
Objetivo: entender como organizar tarefas em uma equipe

Neste exercício, vamos comparar duas situações hipotéticas. Em ambas, há uma equipe de N trabalhadores que precisa realizar K tarefas (K > N). Cada tarefa tem uma duração desconhecida e só pode ser executada por um trabalhador de cada vez.

Forma 1 - Na primeira forma de organização, o trabalhador N recebe toda tarefa k tal que k%N==0. Cada trabalhor se ocupa somente das próprias tarefas.
Forma 2 - Na segunda forma de organização, os trabalhadores fazem inicialmente uma fila. O primeiro da fila pega a próxima tarefa que deve ser executada e dá lugar ao próximo. Quando termina de executá-la, volta ao fim da fila. O tempo para escolher a tarefa é muito menor que o tempo que demora para executar a tarefa.

Em ambos os casos, os trabalhadores só terminam seu turno quando todas as tarefas forem cumpridas. Todos os trabalhadores, supostamente, têm uma habilidade igual de exercutar tarefas.

a) Em qual das formas de organização há uma chance maior de haver trabalhadores que recebem muito mais tarefas que outros? Por que?
b) Qual das formas de organização levará mais rapidamente ao fim de todas as tarefas?

Exercício 3
Objetivo: analisar um código escrito usando a idéia de threadpool

Verifique o código fonte de patterns/threadpool.c.
a) Comece a análise do código pela função main(). O que a função main() faz?
b) Passe então para a função worker(). Qual das formas de organização do exercício anterior (Forma 1 ou Forma 2) parece ter sido implementada?

c) O funcionamento de um "worker" depende do funcionamento de outros workers?

Exercício 4
Objetivo: entender o conceito de "região crítica"

a) O que a função pthread_mutex_lock() faz?
b) Em patterns/threadpool.c, os resultados mostrados pelo programa seriam errados se a linha 48 (pthread_mutex_unlock()) fosse movida para a linha 52 (após a execução de fibo(), mas antes do printf())?
c) Nesse mesmo caso, o que aconteceria com o revezamento dos processos? Se precisar, compile o programa e verifique sua hipótese.
d) Identifique a região do código-fonte que deve obrigatoriamente estar entre um mutex_lock() e um mutex_unlock(), sob pena de, possivelmmente, gerar inconsistência de dados

Exercício 5
Objetivo: entender detalhes sobre um thread pool

Em patterns/threadpool.c:
a) As threads são criadas à medida que as tarefas chegam, ou elas são criadas em uma quantidade razoável e passam a "buscar" problemas?
b) As threads são instruídas a resolver uma tarefa específica, pré definida, ou cada thread pode executar várias instâncias diferentes das tarefas?
c) Cada tarefa depende do resultado de outras tarefas, ou cada tarefa é independente?

Exercício 6
Objetivo: comparar código computacional com um texto da literatura e identificar seus elementos.

O texto abaixo foi traduzido/adaptado da Wikipedia (https://en.wikipedia.org/wiki/Thread_pool):

"Em programação de computadores, uma thread pool é um padrão de design de software que visa conseguir concorrência de execução em um programa. Também chamado de replicated workers ou de modelo worker-crew, uma thread pool mantém multiplas **threads que executam tarefas** fornecidas por um **programa supervisor**. Mantendo todas a threads ativas, o modelo evita o overhead relacionado à criação e destruição de threads para tarefas muito curtas. Um método comum de scheduling de tarefas é uma fila sincronizada chamada "**fila de tarefas**"."

Encontre, no programa patterns/threadpool.c, como são implementadas:
a) As threads que executam tarefas
b) O programa supervisor
c) A fila de tarefas

PROGRAMAÇÃO MULTITHREAD – THREAD FARM

Objetivo: entender como organizar um programa na forma de uma "thread farm"

Exercício 1
Objetivo: entender como organizar tarefas em uma equipe

Neste exercício, vamos comparar duas situações hipotéticas. Em ambas, há uma equipe de trabalhadores que precisa realizar tarefas. Cada tarefa tem uma duração desconhecida e só pode ser executada por um trabalhador de cada vez. Não sabemos, em princípio, nem quantas tarefas existem e nem quando elas chegarão.

Forma 1 - Na primeira forma de organização, N trabalhadores esperam pela chegada da tarefa, recebendo, para isso, pelas horas de trabalho em espera. Quando uma tarefa chega, ela é realizada pelo próximo trabalhador disponível.
Forma 2 - Na segunda forma de organização, os trabalhadores ficam em suas casas. Um funcionário faz uma "triagem" da tarefa que chega e então chama um trabalhador para realizá-la. Após terminar, o funcionário vai para casa.

a) Qual das formas de organização leva a uma eficiência maior para cumprir tarefas?
b) Qual das formas de organização leva a um gasto maior com horas de trabalho?

Exercício 2
Objetivo: analisar um código escrito usando a idéia de thread farm

Verifique o código fonte de patterns/farm.c.
a) Comece a análise do código pela função worker(). O que a função worker() faz?
b) Passe então para a função main(). Qual das formas de organização do exercício anterior (Forma 1 ou Forma 2) parece ter sido implementada?
c) O funcionamento de um "worker" depende do funcionamento de outros workers?

Exercício 3
Objetivo: encontrar a "região crítica" em programas

Encontre a região crítica em patterns/farm.c.

Exercício 4
Objetivo: entender detalhes sobre uma thread farm

Em patterns/farm.c:
a) As threads são criadas à medida que as tarefas chegam, ou elas são criadas em uma quantidade razoável e passam a "buscar" problemas?
b) As threads são instruídas a resolver uma tarefa específica, pré definida, ou cada thread pode executar várias instâncias diferentes das tarefas?
c) Cada tarefa depende do resultado de outras tarefas, ou cada tarefa é independente?

Exercício 5
Objetivo: propor políticas para a criação de threads em um sistema

A criação ou destruição de uma thread gera um overhead computacional muito grande. Por outro lado, manter uma thread funcionando gera um overhead computacional (muito menor, porém não desprezível).

Em webservers, as requisições geralmente são esparsas, exceto pelos picos de acessos. Alguns desses picos acontecem em horários específicos, mas eles frequentemente ocorrem em horários imprevisíveis. Quando uma requisição não é atendida imediatamente, ela entra em uma fila e aguarda a sua vez.

Como seria uma boa forma de administrar (criar e destruir) threads em um webserver, visando minimizar tanto o custo computacional quanto o tempo de espera para requisições?

PROGRAMAÇÃO MULTITHREAD - PRODUTOR-CONSUMIDOR

Objetivo: encontrar e resolver problemas em sistemas produtor-consumidor.

Obs: Estes exercícios usam o programa threading/patterns/produtor-consumidor.c.

Exercício 1
Objetivo: entender os elementos que compõem um sistema produtor-consumidor

Analisando o código de produtor-consumidor.c, identifique:
a) A função/thread que produz elementos. Como eles são produzidos?
b) A função/thread que consome elementos. Como eles são consumidos?
c) A estrutura de dados que é usada para passar elementos da thread produtora para a consumidora.

Exercício 2
Objetivo: encontrar condições de corrida

No programa produtor-consumidor.c, o que acontece se:
a) Os elementos são produzidos e consumidos na mesma taxa, sincronizadamente (um elemento produzido para um elemento consumido)?
b) Os elementos são produzidos mais rápido que podem ser consumidos?
c) Os elementos são consumidos mais rápido que podem ser produzidos?

Após realizar essas inferências, execute o código. Se precisar, altere as instruções sleep() dentro das threads para realizar mais testes. Ainda não altere mais nada no programa.

Exercício 3
Objetivo: encontrar regiões críticas

Encontre as regiões críticas das threads e proteja-as com mutexes. Agora, o programa funciona como deveria? Por que?

Exercício 4
Objetivo: analisar uma proposta de alteração de programa

Uma possível proposta para resolver o problema encontrado é alterar as funções que controlam o buffer circular de forma que ela (a) ignore novas entradas caso o buffer esteja cheio e (b) retorne um erro na leitura caso o buffer esteja vazio. Quais seriam os problemas dessa solução?
(obs: se você preferir implementar a solução para testar, faça uma cópia de segurança do código-fonte original, pois ele será usado no exercício 5).

Exercício 5
Objetivo: propor uma solução para o problema de buffer circular do produtor-consumidor.

O objetivo deste exercício é propor uma solução para os problemas que encontramos no sistema, de tal forma que:
a) somente as funções produtor() e consumidor() sejam alteradas em relação ao código-fonte original,
b) nenhuma produção seja perdida, isto é, todas as produções possam ser consumidas,
c) funções sleep() não tenham que ser usadas (embora possam ser usadas durante fases de teste).

LEITURA COMPLEMENTAR

Após resolver o problema do sistema produtor-consumidor usando mutexes, leia o seguinte texto:

https://www.geeksforgeeks.org/semaphores-operating-system/

Se precisar, encontre mais leituras sobre semáforos. A documentação da biblioteca semaphore.h pode ser bastante útil:

http://pubs.opengroup.org/onlinepubs/009696799/basedefs/semaphore.h.html

Na biblioteca, encontre:
a) Qual é a função em C que implementa as funcionalidades de v(), que é descrita no texto recomendado?
b) Qual é a função em C que implementa as funcionalidades de p(), que é descrita no texto recomendado?

Depois disso, resolva novamente o Exercício 5, mas, desta vez, usando semáforos ao invés de mutexes. Exceto pela adição de variáveis globais relacionadas a semáforos, você não deve alterar o código-fonte original.

Esse exercício não será feito em aula, mas todos devem sentir-se livres para perguntar sobre ele em horários de monitoria e/ou atendimento extra-classe. Em especial, devem sentir-se encorajados a estudar esse problema em grupo.

Embora haja diversas soluções para isso online, é interessante passar algum tempo buscando a própria solução antes de compará-la com as soluções da literatura.

PROGRAMAÇÃO MULTITHREAD - DEADLOCK

Objetivo: entender as condições em que ocorre Deadlock

Obs: Estes exercícios usam os programas no diretório deadlock/

Exercício 1
Objetivo: entender um processo de travamento por deadlock
Analise o código-fonte do programa 01-trava_dupla.c. Antes de executá-lo, faça uma hipótese sobre o que ele deve mostrar na tela. Depois, compile o programa. Sua hipótese se confirma?

Exercício 2
Objetivo: entender as condições necessárias para ocorrer deadlock
Analisando o código 01-trava_dupla.c, argumente sobre a existência ou não das seguintes condições.
a) **Não-Preempção**: um recurso preemptivo é aquele que pode ser "retirado" de uma thread ou processo pelo sistema operacional.
b) **Exclusão mútua**: apenas um processo ou thread pode ter acesso a um recurso e ele não pode ser compartilhado.
c) **Espera com retenção de recursos (hold and wait)**: uma thread ou processo tem acesso a pelo menos um recurso e pede acesso a outros recursos.
d) **Espera circular**: um processo ou thread depende de um recurso que está sendo acessado por outra thread ou processo, que por sua vez depende de um recurso de outra thread ou processo, e assim por diante, até retornar à primeira thread ou processo.

Exercício 3
Objetivo: entender uma possível solução para deadlock
Analise o código-fonte do programa 02-altruista.c. Como ele propõe resolver o problema de travamento? Qual é o problema dessa solução? Que condição (dentre as discutidas no exercício 2) foi atenuada?

Exercício 4
Objetivo: entender uma possível solução para deadlock
Analise o código-fonte do programa 03-mutex_amplo.c. Como ele propõe resolver o problema de travamento? Qual é o problema dessa solução? Que condição (dentre as discutidas no exercício 2) foi atenuada?

Exercício 5
Objetivo: entender uma possível solução para deadlock
Analise o código-fonte do programa 04-random_sleep.c. Como ele propõe resolver o problema de travamento? Qual é o problema dessa solução? Que condição (dentre as discutidas no exercício 2) foi atenuada?

Exercício 6
Objetivo: entender o problema do jantar dos filósofos

O Problema do Jantar dos Filósofos é um problema clássico da computação. Nele, há um grupo de filósofos sentados ao redor de uma mesa redonda. Cada filósofo tem um palito à sua direita, e, por consequência, um palito à sua esquerda (cada palito é "compartilhado" por dois filósofos). Para comer do prato à sua frente, cada filósofo precisa dos dois palitos que estão próximos de si. Há um código computacional em 05-filosofos.c que implementa computacionalmente esse problema. Usando as condições discutidas no exercício 2, mostre que a situação do jantar dos filósofos pode gerar deadlock.

TEMPO DE EXECUÇÃO

Os programas para esta aula estão na pasta “bounding”

Exercício 1
Objetivo: entender como usar o programa time para medir tempo de execução

Veja o código do programa simple.c. O que ele faz? Compile o programa e então execute-o usando a instrução:
$ gcc -osimple simple.c
$ time ./simple

Que informações aparecem na tela?

Nesta aula, devemos entender e lidar com todas essas informações.

Exercício 2
Objetivo: entender o que é real time

Veja o código do programa waiting.c. O que ele faz? Compile o programa e então execute-o usando o programa time.

a) Qual dos tempos medidos aumentou?
b) O que esse tempo significa?

Exercício 3
Objetivo: entender o que é user time

Veja o código do programa fibo.c. O que ele faz? Compile o programa e então execute-o usando o programa time.

a) Qual dos tempos medidos aumentou?
b) O que esse tempo significa?

Exercício 4
Objetivo: entender o que é sys time

Veja o código do programa readfile.c. O que ele faz? Compile o programa e então execute-o usando o programa time.

a) Qual dos tempos medidos aumentou?
b) O que esse tempo significa?

Exercício 5

Objetivo: a relação entre user time, real time e threads

Veja o código do programa readfile.c. O que ele faz? Compile o programa e então execute-o usando o programa time.

a) Qual dos tempos medidos aumentou?
b) O que esse tempo significa?

Exercício 6
Objetivo: entender como medir real time e sys time à partir de um código C

Abra o código-fonte de cronometro.c.

a) Quais são os parâmetros da função medir_tempo()?

b) Quais funções são responsáveis por medir o tempo real de execução?

c) Quais funções são responsáveis por medir o tempo user de execução?

Exercício 7

Objetivo: reconhecer processos que são limitados por CPU (CPU-bounded) e limitados por I/O (I/O-bounded)

Ligue as observações na tabela abaixo às hipóteses que mais provavelmente às explicam. Depois, ligue as hipóteses às respectivas conclusões que podem ser tiradas delas. Por fim, identifique quais observações significam que o desempenho do programa é limitado por CPU e quais significam que é limitado por I/O.

Observação	Causa provável (hipótese)	Conclusão
Real time quase igual a user time	Programa passa muito tempo esperando por requisições de sistema	O programa provavelmente é multi-thread e tem ganho de desempenho com isso.
Real time muito maior que user time	Programa passa a maior parte do tempo executando instruções	Se o programa for single-thread, convertê-lo para multi-thread não terá impacto no desempenho.
Real time muito menor que user time	Várias threads executam simultaneamente, e passam a maior parte do tempo executando instruçoes	Se o programa for single-thread, ele poderia ser acelerado se fosse convertido para multi-thread.

COMUNICAÇÃO INTER-PROCESSOS 3 - SOCKETS

Objetivo: entender como usar sockets em C

Os programas para esta aula estão na pasta "sockets"

Exercício 1
Objetivo: lembrar o que é um cliente e o que é um servidor

Numa estrutura de rede computacional, as máquinas A, B e C conectam-se, independentemente, à máquina D. As máquinas A, B e C fazem requisições de documentos e a máquina D os fornece.

a) Faça um desenho que mostre essa topologia como um grafo. O que os nós representam? O que as arestas representam?

b) Identifique quais máquinas são clientes e quais máquinas são servidores.

Exercício 2
Objetivo: entender os conceitos de camadas de protocolo

a) Ligue os problemas na tabela abaixo a suas possíveis soluções. Cada problema pode ter mais de uma solução.
b) Enumere os problemas na ordem em que devem ser resolvidos para implementar um sistema de comunicação estilo *messenger*.

Problemas	**Soluções**
Interpretar um comando recebido por um programa	Sincronizar o envio de bits com um sinal auxiliar de clock
Enviar um byte de uma máquina para outra	Atribuir um identificador a cada programa, de tal forma que esse identificador é único dentro da máquina
Enviar um bit de uma máquina para outra	Definir que cada bit levará X microssegundos para ser transmitido
Identificar um programa dentro de uma máquina	Atribuir um identificador único a cada máquina
Identificar uma máquina em um barramento	Definir que 0V significa bit 0 e 5V significa bit 1
	Definir uma frequência de rádio para bit 0 e outra frequência para bit 1
	Definir uma linguagem (JSON, XML, etc.) para organizar dados.

Exercício 3
Objetivo: entender os passos necessários para criar um servidor usando socket

Veja o código-fonte de server.c.
a) Encontre as funções socket(), bind(), listen(), e accept(). O que elas fazem? O que elas retornam?
b) O que acontece se dois processos server tentarem executar simultaneamente? Por que?
c) Qual é a diferença entre o socket_fd e o connection_fd?

Exercício 4
Objetivo: entender os passos necessários para criar um cliente usando socket

Veja o código-fonte de client.c.
a) O que a função connect() faz? Quais são seus parâmetros? O que ela retorna?
b) No cliente, escrevemos e lemos diretamente do socket_fd, criado pela função socket(). No servidor, escrevemos e lemos de connection_fd, criado pela função accept. Por que isso é necessário?

Exercício 5
Objetivo: propor uma solução para servidores que suportam diversas conexões

Discuta possíveis soluções que permitam criar um servidor capaz de lidar com múltiplas conexões simultâneas. Que design patterns seriam mais indicados? Quais problemas aparecerão (race conditions, regiões críticas, riscos de deadlock)?

TAREFA DE CASA (opcional, não vou conferir):
Implemente e teste a solução que você propôs no exercício 5.

LEITURAS COMPLEMENTARES
Tutorial on sockets: http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
(outro) exemplo de cliente servidor: https://www.binarytides.com/server-client-example-c-sockets-linux/

O problema de fazer cliente-servidor em C é muito comum. Procure no GitHub por mais soluções!

SISTEMAS OPERACIONAIS DE TEMPO REAL

Objetivo: entender propriedades de um sistema operacional de tempo real

Exercício 1
Objetivo: comparar o uso de threads com o uso de interrupções em hardware

Assinale as alternativas corretas:
a) Um sistema de controle de um avião poderia ser implementado usando duas threads num esquema produtor-consumidor. Uma das threads é responsável por adquirir dados dos sensores do avião e a outra thread é responsável por calcular um sinal de controle.
b) Em um sistema multithread rodando Linux, é possível garantir que todas as threads serão executadas dentro de um intervalo arbitrário de tempo (por exemplo: 0.02s).
c) Uma interrupção de hardware, num microcontrolador, interrompe imediatamente todas as outras tarefas, e tem latência quase nula.
d) Uma vantagem de programar em threads de um sistema operacional ao invés de programar diretamente sobre o hardware (bare metal) é que é possível desenvolver software portável para diversas plataformas, o que tende a tornar o código mais duradouro e, ao longo do tempo, mais maduro.
e) Uma desvantagem de programar em threads é que não há controle absoluto da temporização de execução de cada tarefa, mas isso não é um problema para nenhuma aplicação.

Exercício 2
Objetivo: analisar um kernel de tempo real não-preemptivo

Veja o código-fonte de rt-kernel.c, no diretório rt-kernel. Esse código não compila (é um pseudo-código), mas pode ser analisado.

a) O que a função contar() faz?
b) Sabendo que a função rotina_de_interrupcao_periodica() é executada por uma chamada de interrupção de hardware que acontece a cada 10ms, qual é a frequência em que a função f_sensor() é chamada? E a função f_comunicador()?

Exercício 3
Objetivo: entender as vantagens de usar um sistema operacional em ambiente embarcado

FreeRTOS (https://www.freertos.org/) é um sistema operacional minimalista voltado a aplicações embarcadas e de tempo real. Em seu manual, ele diz que dá suporte a mais de 20 arquiteturas diferentes.
a) Quanto tempo um engenheiro leva para entender integralmente os bits/flags que devem ser configurados em um microcontrolador para fazer um programa de sensoreamento remoto (que lê

sensores e disponibiliza dados na Internet), assumindo o uso de C e um compilador, sem bibliotecas externas?
b) Quanto tempo um programa de sensoreamento remoto demoraria, aproximadamente, para ser portado para uma arquitetura mais nova?
c) Como um sistema operacional pode ajudar a reduzir esses dois tempos?
d) Estime a economia de recursos, em horas de trabalho, que pode ser conseguida usando um sistema operacional.

Exercício 4

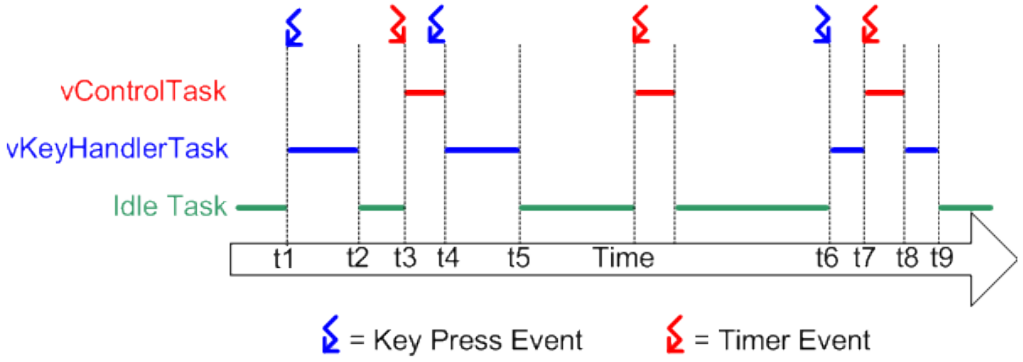
Objetivo: entender as desvantagens de usar um sistema operacional embarcado

Discuta como um sistema operacional afeta os seguintes aspectos:
a) Velocidade máxima de execução de um programa
b) Energia gasta por um programa
c) Espaço (memória EEPROM) usada pelo programa
d) Espaço (memória RAM) usada pelo programa

Exercício 5

Objetivo: entender a preemptividade de tasks em FreeRTOS

FreeRTOS implementa *tasks*, que funcionam de forma semelhante a threads. Tasks têm prioridades diferentes, e podem ser agendadas para executar em intervalos de tempo bem definidos, ou vinculadas a eventos do microcontrolador. O diagrama abaixo mostra o uso de recursos de CPU em duas tasks diferentes:



a) Qual é o evento que “dispara” cada task?
b) Qual task tem mais prioridade?
c) O que acontece quando chega o instante de executar uma task de prioridade mais alta enquanto outra, de prioridade mais baixa, executa?
d) O que acontece quando chega o instante de executar uma task de prioridade mais baixa enquanto outra, de prioridade mais alta, executa?
e) Como esse comportamento se compara ao escalonamento de threads em Linux?

LEITURA COMPLEMENTAR:
https://www.freertos.org -> Tudo sobre FreeRTOS

Sobre o kernel RT do exercício 2: ver vídeo no meu canal do Youtube.

MATERIAL DE APOIO: EXPRESSÕES REGULARES

TIPO	SÍMBOLOS	EXEMPLOS	ACEITAR	REJEITAR
Sequência explícita	( )	(abc)	abc	bac
União	[ ]	[abc]	a b c	ab ac abc
Grupos especiais	[a-z] [A-Z] [a-zA-Z] [0-9]	Minusculas Maiúsculas Todas as letras Todos os dígitos	a A a 4	A a 5 a
Negação	[^] (circunflexo no início do interior de [ ])	[^a-z]+	123	asd
Zero ou um	?	(teste)(ab)?	teste testeab	ab testeabab
Zero ou mais	*	(ab)*	[vazio] abab	dbca
Um ou mais	+	(ab)+ ab(ab)*	ab abab	[vazio] aaaaaabbbbb
N a M	{N,M}	(ab){2,4}	abab abababab	ab abab
N ou mais	{N,}	(ab){2,}	abab abab	ab
Qualquer coisa	.	.	(qualquer string)	[vazio]
Início de string	^	^(ab)+.*	ababteste	testeabab
Fim de string	$	.*(ab)+$	testeabab	ababteste
Caratere especial	[.]	[0-9]+[.][0-9]+	13.5	13@5

GREP

> cat arquivo.txt | grep -w -E “expressao_regular”
> ls -R | grep -w -E “”

-w = reconhecer somente palavras inteiras

-E = usar expressão regular

MATERIAL DE APOIO: ALGORITMO SHIFT-REDUCE