Instituto de Informática - UFRGS

Sistemas Operacionais

Trabalho Prático *Compact Threads* (*cthreads*)

Primitivas *context*

Compilação, ligação e makefile

Aula ``plus´´

Biblioteca de threads *cthreads 19.2*

Instituto de Informática - UFRGS
A. Carissimi -4-sept.-19

- Implementação de uma biblioteca de *threads* modelo N:1
  - Linguagem de programação C (não C++)
  - Executar em ambientes GNU/Linux (máquina virtual *alunovm-sisop.ova*)
    - Ferramenta VirtualBox (*https://www.**virtualbox**.org)*
- Principais tarefas
  - Desenvolver um escalonador não preemptivo com prioridades
  - Implementar primitivas para utilização de *threads* e programas de testes
  - Produto final: biblioteca *libcthread.a*
- Entregáveis e prazos
  - Arquivo *tar.gz* via Moodle com o nome e conteúdo conforme a especificação
  - Relatórios inicial, final e de acompanhamento
  - Datas de entrega: definidas no moodle

A ser feito em grupos de DUAS a TRÊS pessoas!

API primitivas *cthreads*

Instituto de Informática - UFRGS
A. Carissimi -4-sept.-19

Primitivas para criação, liberação voluntária, sincronização de término (*join*), sincronização de threads (semáforos) , suspensão e uma função genérica de identificação do grupo.

```
int ccreate (void *(*start)(void *), void *arg, int prio);
int cyield (void);
int cjoin (int tid);
int csem_init (csem_t *, int count);
int cwait (csem_t *sem);
int csignal (csem_t *sem);
int cidentify (char *name, int size);
```


Exemplo de uso de *cthreads*

Instituto de Informática - UFRGS
A. Carissimi -4-sept.-19

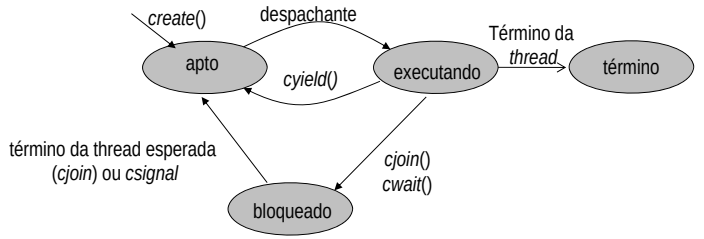
```
#include "../include/cthread.h"
#include <stdio.h>

void* func0(void *arg) {
        printf("Eu sou a thread ID0 imprimindo %d\n", *((int *)arg));
        return;
}

void* func1(void *arg) {
        printf("Eu sou a thread ID1 imprimindo %d\n", *((int *)arg));
}

int main(int argc, char *argv[]) {
        int     id0, id1, i;

        id0 = ccreate(func0, (void *)&i, 0);
        id1 = ccreate(func1, (void *)&i, 0);

        printf("Eu sou a main após a criação de ID0 e ID1\n");

        cjoin(id0);
        cjoin(id1);

        printf("Eu sou a main voltando para terminar o programa\n");
}
```

Supõe que esteja no diretório testes

Escalonador por prioridade (múltiplas filas) 2019-02

- Escalonador não preemptivo
  - Perde o processador apenas em chamada de sistema
    - Cedência voluntária (*cyield*), sincronização de espera (*cjoin*) por término de *thread*, primitivas de sincronização (*cwait*) ou término da função (*return*)
- Emprega uma política de prioridade dinâmica NÃO preemptivo
  - Prioridade é um valor correspondente ao último tempo no estado executando
  - Valores numericamente maiores, prioridade menor
- Escalonamento em dois níveis (multinível)
  - Seleciona a *thread* de maior prioridade (1º nível, escolha pela prioridade)
  - *Threads* com a mesma prioridade são escolhidas na base FCFS (2º nível)

Diagrama de estados e transições de *cthreads*



Fila* aptos (uma só)
Fila* de bloqueados (uma por semáforo)
Fila de executando (fila do "eu sozinho")

*Na verdade são listas encadeadas!!

support.h
support.o

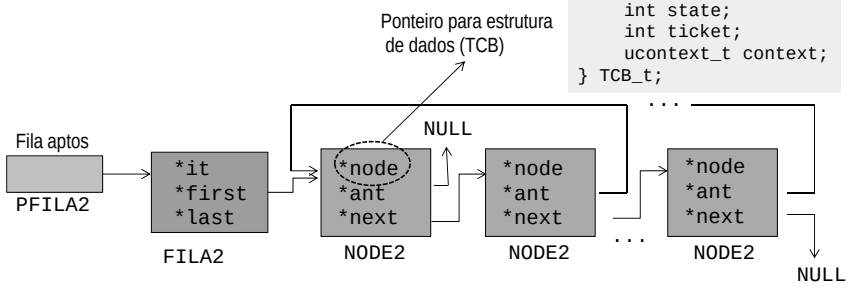
```
typedef struct s_TCB {
    int tid;
    int state;
    int prio;
    ucontext_t context;
} TCB_t;
```

cdata.h

Filas da *support.o*

(Fornecido no cdata.h)

```
typedef struct s_TCB {
    int tid;
    int state;
    int ticket;
    ucontext_t context;
} TCB_t;
```



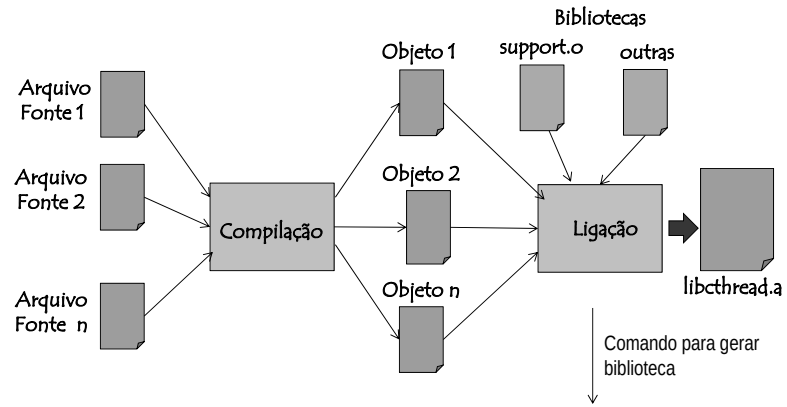

```
struct sFila2{
        struct sFilaNode2 *it;
        struct sFilaNode2 *first;
        struct sFilaNode2 *last;
} FILA2
```

```
struct sFilaNode2{
        void *node;
        struct sFilaNode2 *ant;
        struct sFilaNode2 *next;
} NODE2
```

```
PNODE2 = struct sFilaNode2
PFILA2 = struct sFila2
```

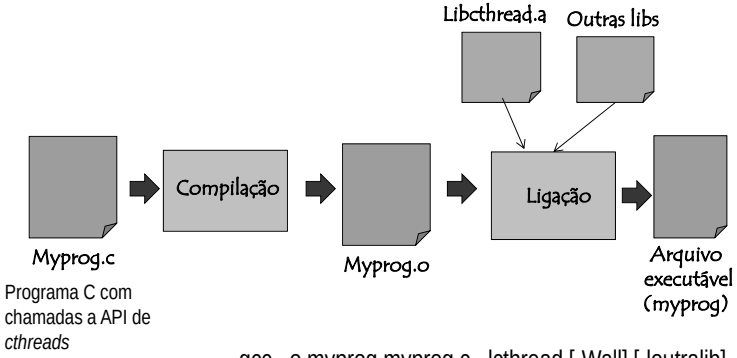
(Fornecidos no support.h)

No desenvolvimento do trabalho...



ar crs libcthread.a objeto1.o objeto2.o objeto3.o support.o

No uso da biblioteca (visão do usuário)...



Programa C com chamadas a API de *cthreads*

gcc –o myprog myprog.c –lcthread [-Wall] [-loutralib]

Biblioteca *libcthreads.a* versão 2019-02

- Entregáveis
  - Fontes da biblioteca *libcthreads.a*
  - *Makefile* para geração da biblioteca *libcthreads.a*
    - *Obrigatório* a definição das regras *clean* e *all*
  - Formulários de acompanhamento e de entrega final
- Forma de entrega
  - UM arquivo *tar.gz* com via Moodle
  - Planilhas de relatório final, em PDF, via Moodle, UMA entrega por grupo
  - Estrutura de diretório com estrutura rígida e fixa a ser seguida
    - Nome do diretório, subdiretórios, arquivos e organização

Prazos de entrega 2019-01

- Formulários de acompanhamento
  - Dia: 09/10/2019 (junto com o trabalho)
- Arquivo tar.gz com a implementação da *libcthread.a*
  - Dia: 09/05/2019
  - Respeito a estrutura de diretórios e arquivos

Entregas via Moodle

UMA entrega por grupo!

Encerramento automático de datas/prazos

Relatório final

RELATÓRIO FINAL DE IMPLEMENTAÇÃO FINAL CTHREADS 2018-02

Grupo

Nome:		**Cartão:**
Nome:		**Cartão:**
Nome:		**Cartão:**

Leia com atenção as instruções a seguir

1. Relatório a ser entregue apenas na entrega final do trabalho
2. Para cada primitiva marque com um "x" apenas uma coluna de status indicando o status da implementação. Se a primitiva está funcionado corretamente assinale a coluna "funciona", caso contrário, assinale "c/ erro". Se a função não foi implementada marque "Não feita"
3. Para as primitivas "c/ erro", descreva o erro que ocorre e, em havendo uma explicação provável, descreva sua hipótese.

	Status			
Primitiva	Funciona	C/ Erro	Não feita	Descrição do problema (apenas funções com erro)
ccreate()				
cyield()				
cjoin()				
csem_init()				
cwait()				
csignal()				
csetprio()				
cidentify()				

Material de apoio

- Biblioteca de funções *support*
  - Funções genéricas para tratamento de filas (criação, inserção, retirada, etc)
  - Outras funções de auxílio (nem todas são necessárias para implementação do trabalho neste semestre)
- Disponibilizada na forma de:
  - Arquivo binário *support.o* com o corpo das funções de fila e randomização
    - Deve ser ligado ao código desenvolvido
  - Arquivo de inclusão *support.h*
    - P*rototypes* das funções de fila e randômica e suas estruturas de dados
    - Usado em arquivos fontes com suas funções ou estruturas de dados
- Arquivos de inclusão

  - *cthread.h*: NÃO pode ser modificado (*prototypes* das funções para usuário)
  - *cdata.h:* estrutura de dados TCB e as definidas para a implementação

Família de funções *ucontext*

- Chamadas de sistema do Linux
  - Manipulação de contexto em modo usuário
  - Permitem a criação e chaveamento de contextos
- Funções existentes
  - *makecontext*(): criação
  - *getcontext*(): salva contexto de execução
  - *setcontext*(): recupera contexto de execução
  - *swapcontext*(): troca contexto de execução
    - Equivalente a fazer um *getcontext* seguido de um *setcontext*
- Estrutura de dados *u_context*
  - Define uma pilha, função associada ao contexto, informações do contexto

getcontext( )

```
getcontext(ucontext_t *ucp);
```

- Inicializa a estrutura apontada por *ucp* para o contexto do fluxo de execução que a executa
  - Registradores (PC, SP e GPRs), máscara de sinais e pilha
- O contexto *ucp* pode ser usado para:
  - Opção I: salvar contexto que pode ser retornar posteriormente via *setcontext*()
  - Opção II: servir para criar um molde de contexto, o qual será posteriomente modificado por uma chamada de *makecontext*( )
    - Análogo a criar um processo filho como cópia do pai e, posteriormente, trocar seu código.

makecontext( )

```
makecontext(ucontext_t *ucp, void (*func)(), int arg,...);
```

- Modifica o contexto especificado por *ucp* o qual foi inicializado por uma chamada prévia a *getcontext*( )
- Antes de chamar *makecontext* é preciso modificar *ucp* para
  - Definir uma pilha (área e tamanho)
  - Inicializar o campo *uc_link* para indicar o contexto (*ucp*) a ser executado quando o contexto definido por *makecontext* terminar sua execução.
- *makecontext* define
  - Uma função a ser executada pelo contexto (*func*)
  - Quantidade de argumentos (*argc*) e a lista de argumentos (todos inteiros) a serem passado para o contexto.

setcontext( )

```
setcontext(ucontext_t *ucp);
```

- Restaura o contexto apontado por *ucp*
  - Retorna a um ponto (contexto) previamente definido
  - Contexto deve ter sido criado previamente por *getcontext*( ) ou *makecontext*( )
- O contexto ucp foi definido por:
  - ***getcontext():*** execução continua na instrução seguinte após a chamada do local onde *getcontext* foi feito.
  - ***makecontext()***: execução continua na função definida pelo primeiro argumento de *makecontext*.

swapcontext( )

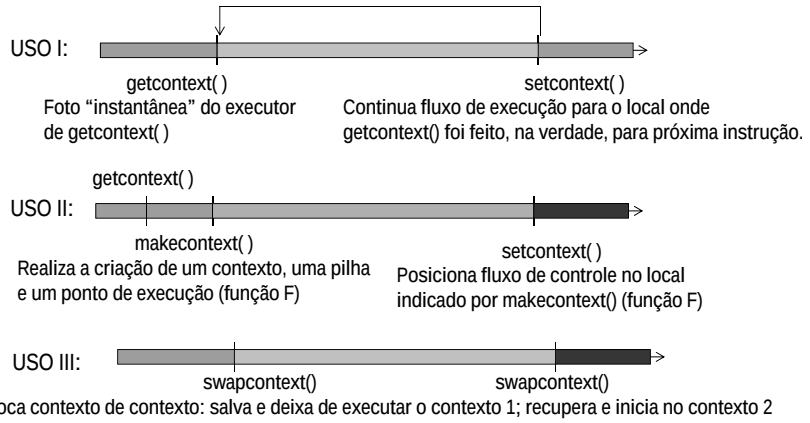
```
swapcontext(ucontext_t *oucp, ucontext_t ucp);
```

- Salva o atual contexto de execução em *oucp* e desvia para o contexto apontado por *ucp*
- Efetua a troca de dois contextos

Chamadas de sistema Linux para tratamento de contexto

FUNDAMENTAL para o trabalho prático da disciplina

- Funções makecontext( ), setcontext(), getcontext() e swapcontext()



Referências iniciais

- Roteiro experimental da atividade no moodle com exemplos
- Man page das funções
- http://en.wikipedia.org/wiki/makecontext

MATERIAL ADICIONAL

Revisão de programação em C

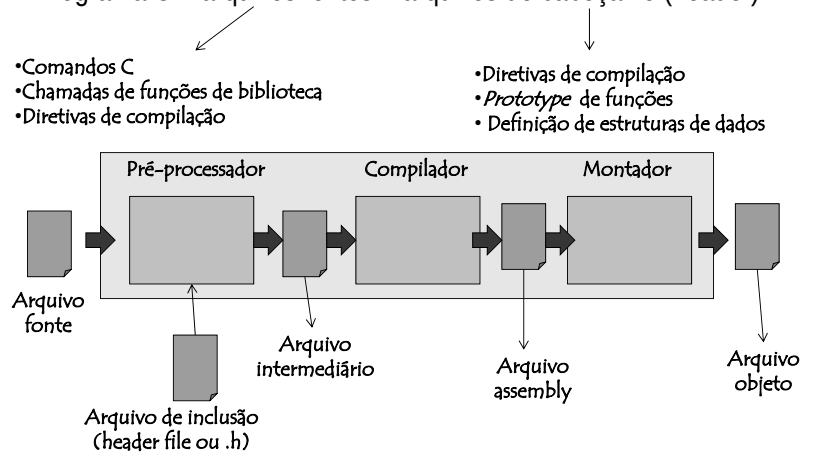
- Programas fontes C
  - Compostos por comandos C, tipos de dados, chamadas de função
  - Diretivas de compilação: ordens de como compilar um programa
    - Inclusão, definição de constantes, condicional
    - Iniciadas pelo caracter #
- Arquivos de cabeçalhos (*header files*)
  - São os arquivos .h
  - Definição de *prototypes* de função (nome, tipo de retorno, parâmetros, etc...)
  - Definição de tipos de dados
  - Adicionados no momento da compilação através da diretiva *include*

Arquivos objetos e executáveis

- A compilação correta de um arquivo fonte gera um arquivo objeto
- Arquivos objeto
  - NÃO podem ser executados diretamente é necessário gerar um executável
  - Vários objetos podem ser combinados para gerar um único executável
    - Situação comum: objeto do programa fonte e biblioteca
      - Ex.: programa "Hello World!" possui um objeto e só se torna executável após ser combinado com a *standard library* (*stdlib* ou *stdio*)
        - *hello.o* tem 1028 bytes e *hello* (executável) tem 7159 bytes (máquina virtual)
- A ligação é responsável por gerar um arquivo executável
  - Etapa obrigatória
  - Comportamento *default* dos compiladores (e.g. *gcc*)
    - Compilar e ligar já gerando um executável

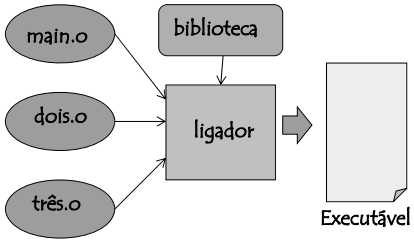
Desenvolvimento de programas em C (em Unix)

- Programa C = arquivos fontes + arquivos de cabeçalho (header)

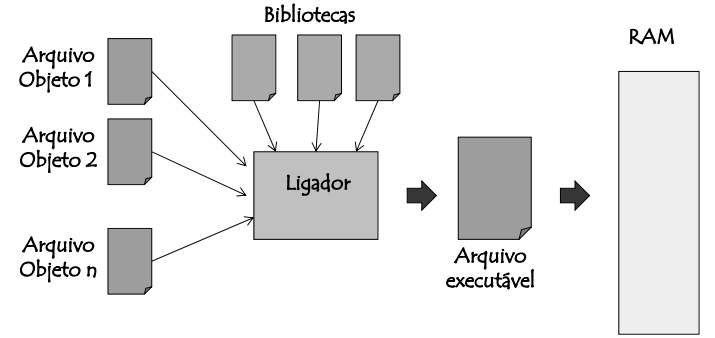


Ligador

- Combinação de vários arquivos objetos para gerar um executável
  - Arquivos desenvolvidos pelo próprio usuário ou por terceiros (bibliotecas)
  - Objetivo é resolver as referências externas



Desenvolvimento de programas em C (em Unix)




```
%ggc -c teste.o teste.c
%gcc -o teste teste.o -lmath
```

Compilação de teste.c gerando objeto teste.o

Ligação de teste.o com a biblioteca *libmath.a** gerando o executável teste

* A localização da biblioteca é dada por um caminho padrão e possuem nome que é prefixado por lib com a extensão .a (se estáticas)

Diretivas de compilação

- Extensão fornecida pelo compilador, não pela linguagem
- Possuem um símbolo identificador (#)
- Sintaxe independente do C
- Podem ser inseridos em qualquer parte do programa fonte
- Não possuem regras de escopo

Caminho dependente da instalação

```
#include <stdio.h>
#include "myheader.h"
```

Insere o arquivo .h no arquivo intermediário

Procura:
-diretório corrente ou definido por um *pathname*
-diretório do sistema (instalação)

Alguns outros exemplos de diretivas

```
#define  CONST   5
#udef    CONST
#define  swap(a,b)  ((a)>(b)?(a):(b))

#if MAX_PROC
    s=random(c);
#define MIN 10
#else
    s=random(2);
#endif

#ifnef PENTIUM
     [...comandos C...]
#endif

#if TAB_SIZE == 100
    int y=10;
#endif
```

Makefile

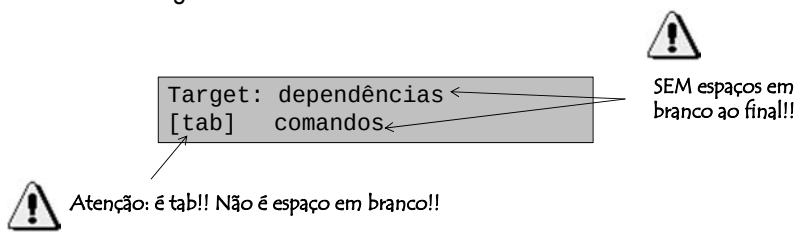
- Mecanismo para criar programas executáveis e gerenciar arquivos
  - Uso comum: realizar a compilação condicional de arquivos fontes e gerar executáveis
    - Verifica de acordo com um critério de data de modificação e dependência quais ações devem ser executadas
- Duas partes
  - Comando *make* e similares (*gmake, pmake*, etc)
  - Arquivo de makefile (*makefile* ou *Makefile*)
    - Regras com as dependências

Comando *make*

- Lê um arquivo de *makefile* e determina qual, ou quais, *targets* (arquivos) precisam ser gerados
  - Critério: comparação entre a data e hora de criação e alteração dos arquivos
  - Arquivos que dependem de arquivos com hora de modificação mais recente precisam ser regerados
- Princípio de funcionamento
  - O arquivo de *makefile* possui alvos (*targets*)
  - Inicia no alvo nomeado como ***all***, se não houver alvo ***all*** inicia:
    - no primeiro alvo do arquivo
    - no alvo fornecido como argumento do comando *make*

Arquivo de makefile

- Conjunto de dependências e regras
  - Basicamente, *target* (quem vai ser criado) e uma lista do conjunto de arquivos de quem ele depende (dependências)
- Formato geral:



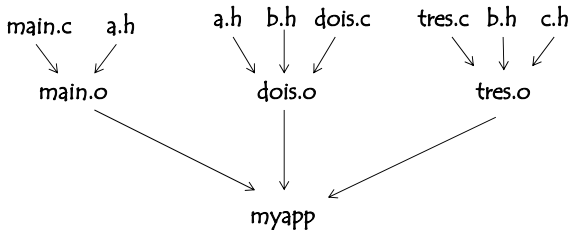
Exemplo (simples) de makefile e sua execução

```
myapp: main.o dois.o tres.o
       gcc –o myapp main.o dois.o tres.o

main.o: main.c a.h
       gcc –c main.c

dois.o: a.h b.h dois.c
       gcc –c dois.c

três.o: tres.c b.h c.h
       gcc –c tres.c

clean:
       rm *.o
```

Exemplos:

```
make  clean (executa a regra com nome clean)
make (executa a regra myapp, primeira do arquivo)
make dois.o (inicia pela regra dois.o)
```

Exemplo (simples): dependência



Situação exemplo:

Se o arquivo *b.h* for modificado, ele terá uma data mais recente que os demais. Assim, ao se executar *make* (ou *make myapp*), como *myapp* depende de *dois.o* e de *tres.o* e esses dois, por sua vez, dependem de *b.h*, será executado as regras *dois.o* e *tres.o.* Note que *main.o* não será regerado por independer de *b.h*.

Exemplo de makefile: alguns detalhes

```
#comentários iniciam o caracter sustenindo

#Possível definir macros (ex.: nome_da_macro = valor)

CC = gcc
LIBS = -L. –l/user

all: myapp  # por onde iniciar a análise de dependência
            # se não há all, inicia pela primeira regra

myapp:      # pode não haver dependência
       $(CC) –o myapp myapp.c $(LIBS)/lib1
```


Escalonador por loteria (versão 2016-02)

- Escalonador não preemptivo
  - Perde o processador por chamada de sistema
    - Cedência voluntária, sincronização de término de *thread*, sincronização de *thread* ou término da função
- Emprega uma política de sorteio (escalonador lotérico)
  - *Thread* recebe um bilhete de loteria na sua criação (valor de 0 a 255 aleatório)
  - Quando acionado, o escalonador gera um número aleatório (0 a 255)
    - A *thread* no estado apto que tiver o número mais próximo do número sorteado ganha o direito de executar na CPU
      - Em caso de empate, a que tem menor identificador (*tid*), ganha!

Escalonador por prioridade (múltiplas filas) 2017-02

- Escalonador não preemptivo
  - Perde o processador por chamada de sistema
    - Cedência voluntária, sincronização de espera por término de *thread*, primitivas de sincronização ou término da função
- Emprega uma política de prioridade dinâmica
  - O valor associado a prioridade é o tempo total que uma *thread* passou no estado executando
  - Quanto maior o tempo em executando, menor sua prioridade
- A fila de aptos é uma lista ordenada de threads (ordem crescente)

Escalonador preemptivo prioridade (2018-02)

- Escalonador preemptivo por prioridades
  - Perde o processador por chamada de sistema
    - Cedência voluntária, sincronização de espera por término de *thread*, primitivas de sincronização ou término da função
  - Perde o processador por prioridade
    - Sempre que existir no estado de “apto” um processo com prioridade MAIOR do que a prioridade do processo em estado “executando”
- Com prioridades
- A fila de aptos deve considerar, caso seja ordenada:
  - Prioridade das threads (primeiro critério)
  - Ordem de chegada (segundo critério)