Trabalho Prático I – Entrega: 9 de maio de 2019
Implementação de Biblioteca de *Threads* cthreads 19.1

1. Descrição Geral

O objetivo deste trabalho é a aplicação dos conceitos de sistemas operacionais relacionados ao escalonamento e ao contexto de execução, o que inclui a criação, chaveamento e destruição de contextos. Esses conceitos serão empregados no desenvolvimento de uma biblioteca de *threads* em nível de usuário (modelo N:1). Essa biblioteca de *threads*, denominada de ***compact thread*** (ou apenas *cthread*), deverá oferecer capacidades básicas para programação com *threads* como criação, execução, sincronização, término e trocas de contexto.

A biblioteca *cthread* deverá ser implementada, OBRIGATORIAMENTE, na linguagem C e sem o uso de outras bibliotecas (além da *libc*, é claro). A criação de *threads* e o chaveamento de contexto deverão utilizar as chamadas de sistema existentes no GNU/Linux da família *makecontext(), setcontext(), getcontext()* e *swapcontext().* A implementação deverá executar em ambiente GNU/Linux e será testada na máquina virtual *alunovm-sisop.ova* disponível no moodle da disciplina.

Para facilitar o desenvolvimento do trabalho, sua organização e, posteriormente, a sua correção, os professores da disciplina disponibilizam um arquivo binário (*support.o*) contendo uma série de funções e definições a serem OBRIGATORIAMENTE usadas. Essencialmente, essas funções realizam a inserção e a retirada de elementos em filas, geração de números randômicos e controle de tempo (essas duas últimas classes de funções não são necessárias em 2019-01). Para ter acesso a essas funções, deve ser incluído o arquivo *support.h*, também fornecido pelos professores.

Ainda, no material disponibilizado no moodle existem dois arquivos de inclusão: *cthreads.h* e *cdata.h*. O arquivo *cthreads.h* NÃO pode ser alterado de forma alguma. O arquivo *cdata*.h possui a definição da estrutura de dados a ser utilizada para o *Thread Control Block* (TCB) e para a variável do tipo semáforo. Essas estruturas, se necessário, podem ser modificadas APENAS pela inclusão de novos campos e, mesmo assim, só após os campos já fornecidos. Todos esses arquivos disponibilizados são fornecidos no arquivo *cthreads.tar.gz*.

2. Descrição Geral

A biblioteca *cthread* deverá ser capaz de gerenciar uma quantidade variável de *threads* (potencialmente grande), limitada pela capacidade de memória RAM disponível na máquina virtual. Cada *thread* deverá ser associada a um identificador único (*tid – thread identifier*) que será um número inteiro positivo (com sinal), de 32 bits (*int*). Não há necessidade de se preocupar com o reaproveitamento do identificador da *thread* (*tid*), pois os testes não esgotarão essa capacidade.

O diagrama de transição de estados é o fornecido na figura 1 e seus estados estão descritos a seguir.

> **Apto**: estado que indica que uma *thread* está pronta para ser executada e que está apenas esperando a sua vez para ser selecionada pelo escalonador. Há quatro eventos que levam uma *thread* a entrar nesse estado: (i) criação da *thread* (primitiva *ccreate*); (ii) cedência voluntária (primitiva *cyield*); (iii) quando a *thread* estiver bloqueada esperando por um recurso (*cwait*) e outra *thread* liberar esse recurso (primitiva *csignal*); e, (iv) quando a *thread* estiver bloqueada pela primitiva *cjoin*, esperando por uma outra *thread*, e essa outra *thread* terminar.
>
> **Executando**: representa o estado em que a *thread* está usando o processador. Uma *thread* é colocada nesse estado sempre que o escalonador selecioná-la para execução. A partir desse estado, uma *thread* pode passar para os estados *apto*, *bloqueado* ou *término*. Uma *thread* passa para *apto* sempre que executar uma primitiva *cyield*. Uma *thread* **pode** passar para *bloqueado* através da execução das primitivas *cjoin* ou *cwait*. Finalmente, uma *thread* passa ao estado *término* quando efetuar o comando *return* ou quando chegar ao final da função que executava.

Bloqueado: uma *thread* passa para o estado *bloqueado* em duas situações: (i) quando uma *thread* executar uma primitiva *cjoin*, para esperar a conclusão de outra *thread*; (ii) ao tentar usar um recurso protegido por semáforo – primitiva *cwait* – e o mesmo já estiver sendo usado por outra *thread*.

O escalonador a ser utilizado deve ser do tipo **com prioridades não preemptivo** e seguir uma política de múltiplas filas sem realimentação, onde cada fila segue uma política FIFO. Há três níveis de prioridade: alta, média e baixa. Nessa política, sempre que as *threads* entrarem no estado *apto*, elas serão inseridas no final da fila corresponde a sua prioridade. Assim, quando a CPU ficar livre, o escalonador selecionará a primeira *thread* da fila de aptos de maior prioridade, que não estiver vazia, para receber a CPU (passar para o estado *executando*).

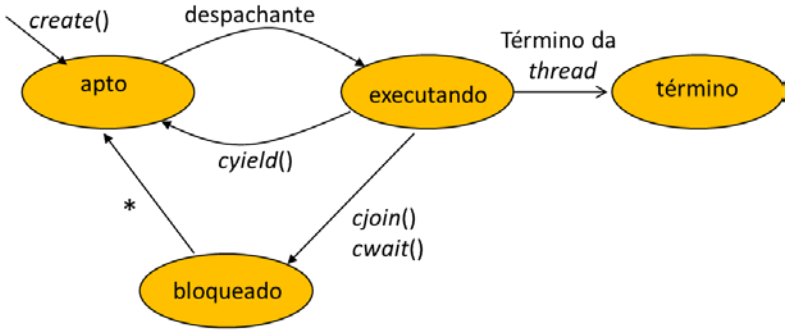

Figura 1 – Diagrama de estados e transições da *cthread*

3. Interface de programação

A biblioteca *cthread* deve oferecer uma interface de programação (API) para permitir seu uso para o desenvolvimento de programas. O grupo deverá desenvolver as funções dessa API, conforme descrição a seguir, que deverá ser RIGOROSAMENTE respeitada. Salienta-se que todas as funções previstas na interface devem ser implementadas. No caso de não haver uma implementação correta, ou mesmo inexistente, de uma determinada primitiva, essa primitiva deverá permitir a sua chamada e deverá ser retornado o código -9 como uma indicação de erro, ou seja, deve ser uma função apenas com o comando *return* com um código de erro.

Criação de uma *thread*: A criação de uma *thread* envolve a alocação das estruturas necessárias à gerência das mesmas (*TCB-Thread Control Blocks*, por exemplo) e a sua devida inicialização. Ao final do processo de criação, a *thread* deverá ser inserida na fila de *aptos*. A função da biblioteca responsável pela criação de uma *thread* é a *ccreate*. A *thread main*, por ser criada pelo próprio sistema operacional da máquina no momento da execução do programa, apresenta um comportamento diferenciado. Esse comportamento está descrito na seção 4.

Observe que o escalonador a ser implementado é do tipo NÃO PREEMPTIVO, ou seja, se uma *thread* em execução criar outra com prioridade maior que a sua, a mesma continuará em execução até que faça uma ação (chamada de função) que torne livre o processador.

```
int ccreate (void *(*start)(void *), void *arg, int prio);
```

Parâmetros:

`start`: ponteiro para a função que a *thread* executará.
`arg`: um parâmetro que pode ser passado para a *thread* na sua criação. (Obs.: é um único parâmetro. Se for necessário passar mais de um valor deve-se empregar um ponteiro para uma *struct*)
`prio`: Valores numéricos válidos 0, 1 e 2 (0 = alta prioridade, 1 = média prioridade, 2 = baixa prioridade). Demais valores são considerados inválidos e a função deve retornar com código de erro.

Retorno*:*

Quando executada corretamente: retorna um valor positivo, que representa o identificador da *thread* criada.
Caso contrário, retorna um valor negativo.

Cedência voluntária da CPU: uma *thread* pode liberar a CPU de forma voluntária com o auxílio da primitiva *cyield*. Se isso acontecer, a *thread* que executou *cyield* retorna ao estado *apto*. Então, o escalonador será chamado para selecionar a *thread* que receberá a CPU.

```
int cyield(void);
```

Retorno:

Quando executada corretamente: retorna 0 (zero)
Caso contrário, retorna um valor negativo.

Alterando a prioridade das *threads*: uma *thread* pode alterar a sua própria prioridade. Se essa operação resultar que exista no estado apto uma *thread* com prioridade maior do que a *thread* em execução (estado executando), a que criou deverá continuar sua execução normalmente SEM ser preemptada em favor daquela no estado apto.

```
int csetprio(int tid, int prio);
```

Parâmetros:

`tid`: identificador da *thread* cuja prioridade será alterada (Na versão 2019/01, deixar sempre esse campo como NULL)
`prio`: nova prioridade da *thread*.

Retorno:

Quando executada corretamente: retorna 0 (zero)
Caso contrário, retorna um valor negativo.

Sincronização de término: a função *cjoin* permite que uma *thread* fique bloqueada a espera pelo término de execução de outra *thread*. A função *cjoin* recebe como parâmetro o identificador da *thread* cujo término está sendo aguardado. Quando essa *thread* terminar, a função *cjoin* retorna com um valor inteiro indicando o sucesso ou não de sua execução. Uma determinada *thread* só pode ser esperada por uma única outra *thread*. Se duas ou mais *threads* fizerem *cjoin* para esperar o término de uma mesma *thread*, apenas a primeira que realizou a chamada será bloqueada. As outras chamadas retornarão imediatamente com um código de erro e seguirão sua execução. Se *cjoin* for feito para uma *thread* que não existe (não foi criada ou já terminou), a função retornará imediatamente com um código de erro. Observe que não há necessidade de um estado *zombie*, pois a *thread* que aguarda o término de outra (a que fez *cjoin*) não recupera nenhuma informação de retorno proveniente da *thread* aguardada.

Observe que se uma *thread* terminar e desbloquear uma outra *thread*, o escalonador deve levar em conta as prioridades para selecionar a próxima a executar.

```
int cjoin(int tid);
```

Parâmetros:

`tid`: identificador da *thread* cujo término está sendo aguardado.

Retorno:

Quando executada corretamente: retorna 0 (zero)
Caso contrário, retorna um valor negativo.

Sincronização de controle: o sistema prevê o emprego de uma variável especial para realizar a sincronização de acesso a recursos compartilhados (por exemplo, uma seção crítica). As primitivas existentes são *csem_init, cwait* e *csignal*, e usam uma variável especial *(csem_t*) que recebe o nome específico de *semáforo*. A primitiva *csem_init* é usada para inicializar a variável *csem_t* e deve ser chamada, obrigatoriamente, antes que essa variável possa ser usada com as primitivas *cwait* e *csignal*.

Inicialização de semáforo: a função *csem_init* inicializa uma variável do tipo *csem_t* e consiste em fornecer um valor inteiro (*count*), positivo ou negativo, que representa a quantidade existente do recurso controlado pelo semáforo. Para realizar *exclusão mútua*, o valor inicial da variável *semáforo* deve ser 1 (semáforo binário). Ainda, cada *variável semáforo* deve ter associado uma estrutura que registre as *threads* que estão bloqueadas, esperando por sua liberação. Na inicialização essa lista deve estar vazia.

```
int csem_init (csem_t *sem, int count);
```

Parâmetros:

`sem`: ponteiro para uma variável do tipo *csem_t*. Aponta para uma estrutura de dados que representa a variável semáforo.

`count`: valor a ser usado na inicialização do semáforo. Representa a quantidade de recursos controlados pelo semáforo.

Retorno:

Quando executada corretamente: retorna 0 (zero)
Caso contrário, retorna um valor negativo.

Solicitação (alocação) de recurso: a primitiva *cwait* será usada para solicitar um recurso. Se o recurso estiver livre, ele é atribuído a *thread*, que continuará a sua execução normalmente; caso contrário, a *thread* será bloqueada e posta na fila de espera desse recurso. Se na chamada da função o valor de *count* for menor ou igual a zero, a *thread* deverá ser posta no estado bloqueado e colocada na fila associada a variável *semáforo*. Para cada chamada a *cwait* a variável *count* da estrutura semáforo é decrementada de uma unidade.

```
int cwait (csem_t *sem);
```

Parâmetros:

`sem`: ponteiro para uma variável do tipo semáforo.

Retorno:

Quando executada corretamente: retorna 0 (zero)
Caso contrário, retorna um valor negativo.

Liberação de recurso: a chamada *csignal* serve para indicar que a *thread* está liberando o recurso associado ao semáforo dado por *csem_t*. Para cada chamada da primitiva *csignal*, a variável *count*, no interior da *struct s_sem*,deverá ser incrementada de uma unidade. Se houver mais de uma *thread* bloqueada a espera desse recurso, a primeira delas, segundo uma política de FIFO, deverá passar para o estado *apto* e as demais devem continuar no estado *bloqueado*.

Atente para o fato que o escalonar é NÃO preemptivo, assim, se a *thread* em executando realizar um *csignal* que libere (desbloqueie) uma *thread* com uma prioridade maior que a sua, a *thread* em execução continuará no estado *executando*..

Ainda, as *threads* que esperam por um mesmo semáforo devem ser liberadas de acordo com a sua prioridade, isso é, se houver uma *thread* de média prioridade e de alta prioridade esperando pelo mesmo semáforo, a de alta prioridade é que deve ser desbloqueada. As *threads* de mesma prioridade são desbloqueadas na ordem em que foram bloqueadas (fila).

```
int csignal (csem_t *sem);
```

Parâmetros:

`sem`: ponteiro para uma variável do tipo semáforo.

Retorno:

Quando executada corretamente: retorna 0 (zero)
Caso contrário, retorna um valor negativo.

Identificação do grupo: Além das funções de manipulação das *threads* e de sincronização, a biblioteca deverá prover a implementação de uma função que forneça o nome dos alunos integrantes do grupo que desenvolveu a biblioteca *chtread*. O protótipo dessa função é:

```
int cidentify (char *name, int size);
```

Parâmetros:

`name`: ponteiro para uma área de memória onde deve ser escrito um *string* que contém os nomes dos componentes do grupo e seus números de cartão. Deve ser uma linha por componente.

`size`: quantidade máxima de caracteres que podem ser copiados para o *string* de identificação dos componentes do grupo.

Retorno:

Quando executada corretamente: retorna 0 (zero)

Caso contrário, retorna um valor negativo.

ATENÇÃO: A biblioteca deve possuir todas as funções da API, mesmo que não tenham sido implementadas. Nesse caso, devem apenas retornar o código de erro (-9). A função *cidentify*, apesar de ser obrigatória sua implementação, NÃO vale pontos na avaliação final.

4. Comportamento da *thread main*

Ao lançar a execução de um programa, o sistema operacional cria um processo e associa a esse processo uma *thread* principal (*main*), pois todo processo tem pelo menos um fluxo de execução. Assim, na implementação da *cthread*, existirão dois tipos de *threads*: *thread main* (criada pelo sistema operacional) e as *threads* de usuário (criadas através das chamadas *ccreate*). Isso implica na observação dos seguintes aspectos, sobre o tratamento das *threads* e, em especial, da *thread main*:

- É necessário definir um contexto para a *thread main*. Esse contexto deve ser criado apenas na primeira chamada às funções da biblioteca *cthread* para, posteriormente, em trocas de contexto da *main* para as *threads* criadas pelo *ccreate*, ser possível salvar e recuperar o contexto de execução da *main*. Para a criação desse contexto devem ser utilizadas as mesmas chamadas *getcontext*() e *makecontext*(), usadas na criação de *threads* com a *ccreate*.
- A *thread main* deverá ter associado um identificador único (*tid*). Esse *tid* deverá ser o valor ZERO. Portanto, da mesma forma que as *threads* de usuário, a *thread main* também possui um TCB associado.

A thread *main* tem a menor prioridade (baixa).

5. Entregáveis: o que deve ser entregue?

A entrega do trabalho será realizada através da submissão pelo Moodle de um arquivo *tar.gz*, cuja estrutura de diretórios deverá seguir, OBRIGATORIAMENTE, a mesma estrutura de diretórios do arquivo *cthread.tar.gz* fornecido (conforme seção 6). Entregar apenas UM arquivo por grupo.

Utilize a estrutura de diretórios especificada para desenvolver seu trabalho. Assim, ao terminá-lo, basta gerar um novo arquivo *tar.gz*, conforme descrito no ANEXO II. Observe também o seguinte:

- **NÃO** inclua, no *tar.gz*, cópia da Máquina Virtual;
- **NÃO** serão aceitos outros formatos de arquivos, tais como *.tgz*, *.rar* ou *.zip*.
- **REMOVA** qualquer outro arquivo desnecessário como, por exemplo, aqueles gerados por gerenciadores de versões (*git*, entre outros).

O arquivo *tar.gz* deverá conter os arquivos fontes da implementação, os arquivos de *include*, a biblioteca, a documentação, os *makefiles* e os programas de testes.

Além do arquivo de entrega, serão disponibilizados no Moodle uma planilha de acompanhamento do projeto que deverá ser **atualizada semanalmente**, mais um formulário de entrega final. A planilha de acompanhamento deverá ser preenchida, UMA por grupo, pelo "gerente" do grupo, e ser entregue em

uma versão *pdf*, nas datas fornecidas no Moodle. O gerente é um dos componentes, designado pelo próprio grupo, e que será o responsável por todas as entregas.

6. Arquivo *.tar.gz*

Será fornecido pelo professor (disponível no Moodle) um arquivo *cthread.tar.gz*, que deve ser descompactado, conforme descrito no ANEXO II, de maneira a gerar, em seu disco, a estrutura de diretórios a ser utilizada, OBRIGATORIAMENTE, para a entrega do trabalho.

No diretório raiz (diretório *cthread*) da estrutura de diretórios do arquivo *cthread.tar.gz* está disponibilizado um arquivo *Makefile* de referência, que deve ser completado de maneira a gerar a biblioteca (ver seção 8). Os subdiretórios e arquivos do diretório *cthread* são os seguintes:

\cthread		
	bin	DIRETORIO: local estarão todos os arquivos objetos (arquivos *.o*) gerados pela compilação da biblioteca. Nesse subdiretório está disponível o arquivo *support.o*, de uso obrigatório. Esse arquivo possui a implementação das funções de gerenciamento de filas. A regra *clean* do *makefile* NÃO deve remover esse arquivo;
	exemplos	DIRETÓRIO: local onde estão os programas fonte de exemplos fornecidos pelos professores da disciplina e um *makefile* para geração dos executáveis dos exemplos. Os arquivos resultantes da compilação serão colocados nesse mesmo subdiretório.
	include	DIRETÓRIO: local onde deverão ser postos todos os seus arquivos de inclusão (arquivos *.h*). Nesse subdiretório estão disponíveis os arquivos *cthread.h* e *cdata.h* , de uso obrigatório. Também estará disponível nesse subdiretório o arquivo *support.h*, com os protótipos das funções de gerenciamento de filas;
	lib	DIRETÓRIO: local onde deve ser posta a biblioteca gerada (*libcthread.a*);
	src	DIRETÓRIO: local onde são postos todos os arquivos ".c" (códigos fonte) usados na implementação de *cthread* 19.1.
	teste	DIRETÓRIO: local para os programas de teste fornecidos pelo grupo. Nesse diretório deverão ser postos todos os arquivos usados na geração dos testes: fonte dos programas de teste, arquivos objeto, arquivos executáveis e o *makefile* para sua geração (ver seção 8).
	Makefile	ARQUIVO: arquivo *makefile* com regras para gerar a "*libcthread*". Deve possuir uma regra "*clean*", para limpar todos os arquivos binários (.o) e a biblioteca (.a) gerados pelo grupo (NÃO remover o arquivo *support.o*)
	support.pdf	ARQUIVO: arquivo com a documentação das funções da biblioteca de suporte.

Para criar programas de teste que utilizem a biblioteca *cthread* siga os procedimentos da seção 8.

7. Geração da *libcthread* (descrição do Makefile)

As funcionalidades da *cthread* deverão ser disponibilizadas através da biblioteca denominada *libcthread.a*. Uma biblioteca é um tipo especial de programa objeto em que suas funções são chamadas por outros programas. Para isso, o programa chamador deverá ser ligado com a biblioteca, formando um único executável. Portanto, uma biblioteca é um arquivo objeto, com formato específico, gerado a partir dos arquivos fontes que implementam as suas funções.

Para gerar uma biblioteca deve-se proceder da seguinte forma (vide detalhes no ANEXO I):

- Compilar os arquivos que implementam a biblioteca, usando o comando *gcc* e gerando os arquivos objeto correspondentes.
- Gerar o arquivo da biblioteca usando o comando *ar*. Devem ser colocados nessa biblioteca todos os arquivos ".o" gerados na compilação e o arquivo *support.o* fornecido.

Notar que o programa fonte do chamador deve incluir o arquivo de cabeçalho (*header files*) *cthread.h* com os protótipos das funções disponibilizadas pelo arquivo *libcthread.a*, de maneira a ser compilado sem erros.

Para gerar a biblioteca deverá ser criado um *makefile* com, pelo menos, duas regras:

- Regra "*all*": responsável por gerar o arquivo *libcthread.a*, no diretório *lib*. Isso inclui a compilação de todos os programas do grupo
- Regra "*clean*": responsável por remover todos os arquivos dos subdiretórios *bin* e *lib*, a exceção do arquivo *support.o*

De forma similar, o *makefile* do diretório *testes*, deverá ter a regra "all" (responsável por gerar todos os executáveis dos programas de teste fornecidos) e a regra "*clean*" para remover todos os arquivos binários (arquivos *.o*) e os executáveis dos programas de teste.

8. Utilizando a *cthread*: execução e programação (programas de teste)

A partir do *main* de um programa C poderão ser lançadas várias *threads* através da primitiva de criação de *threads*. Cada *thread* corresponderá, na verdade, a execução de uma função desse programa. Todas as funções da biblioteca podem ser chamadas pela *main*. Por exemplo, pode-se chamar a *cjoin()* para que a *thread main* aguarde que suas *threads* filhas terminem.

Após ter desenvolvido um programa em C, esse deve ser compilado e ligado com a biblioteca que implementa a *cthread* (ver ANEXO I sobre como ligar os programas à biblioteca). Então, o programa executável resultante poderá ser executado.

O arquivo *cthread.tar.gz*, fornecido no Moodle como parte dessa especificação, possui no diretório *exemplo* alguns programas exemplos do uso das primitivas da biblioteca *cthread*. Também está disponível, nesse mesmo diretório, um *makefile* exemplo para gerar esses programas.

IMPORTANTE: os programas exemplo são do semestre passado e precisam ser adaptados para a API e para o comportamente da biblioteca cthreads deste semestre.

9. Material suplementar de apoio

A biblioteca definida constitui o que se chama de *biblioteca de threads em nível de usuário* (modelo N:1). Na realidade, o que está sendo implementado é uma espécie de máquina virtual que realiza o escalonamento de *threads* sobre um processo do sistema operacional. A base para elaboração e manipulação das *cthreads* são as chamadas de sistema providas pelo GNU/Linux: *makecontext*(), *setcontex*t(), *getcontext*() e *swapcontext*(). Estude o comportamento dessas funções.

10. Critérios de avaliação

A avaliação do trabalho considerará as seguintes condições:

- Entrega das planilhas de acompanhamento parciais e final e do trabalho final dentro dos prazos estabelecidos;
- Obediência à especificação (formato e nome das funções);
- Compilação e geração da biblioteca **sem erros ou *warnings***;
- Fornecimento de todos os arquivos solicitados conforme organização de diretórios fornecidos na seção 8;
- Execução correta dentro da máquina virtual *alunovm-sisop.ova*.

Itens que serão avaliados e sua valoração:

- 20,0 pontos: relativos a entrega das planilhas de acompanhamento e de entrega final, em formato *pdf*, nos *links* disponibilizados no Moodle. Cada formulário vale 4 pontos: são quatro planilhas de acompanhamento e uma de entrega final. ATENÇÃO: respeite os prazos do Moodle. Não serão aceitos entregas após o prazo.
- 80,0 pontos: funcionamento da *cthread* de acordo com a especificação. Para essa verificação serão utilizados programas padronizados desenvolvidos pelos professores da disciplina. A nota será proporcional à quantidade de execuções corretas desses programas, considerando-se a dificuldade relativa de cada um.

11. Avisos gerais – LEIA com MUITA ATENÇÃO

1. Faz parte da avaliação a obediência RÍGIDA aos padrões de entrega definidos na seção 6 (arquivos *tar.gz*, estrutura de diretórios, *makefile*,etc).
2. O trabalho deverá ser desenvolvido em grupos de DOIS ou TRÊS componentes. NÃO poderá ser feito individualmente.
3. O trabalho final deverá ser entregue até a data prevista, conforme cronograma de entrega no ***Moodle***. Deverá ser entregue um arquivo *tar.gz* conforme descrito na seção 6. NÃO haverá extensão de prazos.

12. Observações

Recomenda-se a troca de ideias entre os alunos. Entretanto, a identificação de cópias de trabalhos acarretará na aplicação do Código Disciplinar Discente e a tomada das medidas cabíveis para essa situação.

ANEXO I – Compilação e Ligação

1. Compilação de arquivo fonte para arquivo objeto

Para compilar um arquivo fonte (*arquivo.c*, por exemplo) e gerar um arquivo objeto (*arquivo.o*, por exemplo), pode-se usar a seguinte linha de comando:

```
gcc –c arquivo.c –Wall
```

Notar que a opção *–Wall* solicita ao compilador que apresente todas as mensagens de alerta (*warnings*) sobre possíveis erros de atribuição de valores a variáveis e incompatibilidade na quantidade ou no tipo de argumentos em chamadas de função.

2. Compilação de arquivo fonte DIRETAMENTE para arquivo executável

A compilação pode ser feita de maneira a gerar, diretamente, o código executável, sem gerar o código objeto correspondente. Para isso, pode-se usar a seguinte linha de comando:

```
gcc –o arquivo arquivo.c –Wall
```

3. Geração de uma biblioteca estática

Para gerar um arquivo de biblioteca estática do tipo "*.a*", os arquivos fonte devem ser compilados, gerando-se arquivos objeto. Então, esses arquivos objeto serão agrupados na biblioteca. Por exemplo, para agrupar os arquivos "*arq1.o*" e "*arq2.o*", obtidos através de compilação, pode-se usar a seguinte linha de comando:

```
ar crs libexemplo.a arq1.o arq2.o
```

Nesse exemplo está sendo gerada uma biblioteca de nome "*exemplo*", que estará no arquivo *libexemplo.a*.

4. Utilização de uma biblioteca

Deseja-se utilizar uma biblioteca estática (chamar funções que compõem essa biblioteca) implementada no arquivo *libexemplo.a*. Essa biblioteca será usada por um programa de nome *myprog.c*.

Se a biblioteca estiver no mesmo diretório do programa, pode-se usar o seguinte comando:

```
gcc -o myprog myprog.c –lexemplo -Wall
```

Notar que, no exemplo, o programa foi compilado e ligado à biblioteca em um único passo, gerando um arquivo executável (arquivo *myprog*). Observar, ainda, que a opção *–l* indica o nome da biblioteca a ser ligada. Observe que o prefixo *lib* e o sufixo *.a* do arquivo não necessitam ser informados. Por isso, a menção apenas ao nome *exemplo*.

Caso a biblioteca esteja em um diretório diferente do programa, deve-se informar o caminho (*path* relativo ou absoluto) da biblioteca. Por exemplo, se a biblioteca está no diretório */user/lib*, caminho absoluto, pode-se usar o seguinte comando:

```
gcc -o myprog myprog.c –L/user/lib –lexemplo -Wall
```

A opção "*-L*" suporta caminhos relativos. Por exemplo, supondo que existam dois diretórios: *testes* e *lib*, que são subdiretórios do mesmo diretório pai. Então, caso a compilação esteja sendo realizada no diretório *testes* e a biblioteca desejada estiver no sudiretório *lib*, pode-se usar a opção *-L* com "*../lib*". Usando o exemplo anterior com essa nova localização das bibliotecas, o comando ficaria da seguinte forma:

```
gcc -o myprog myprog.c –L../lib –lexemplo -Wall
```

ANEXO II – Compilação e Ligação

1. Desmembramento e descompactação de arquivo *.tar.gz*

O arquivo *.tar.gz* pode ser desmembrado e descompactado de maneira a gerar, em seu disco, a mesma estrutura de diretórios original dos arquivos que o compõe. Supondo que o arquivo *tar.gz* chame-se "*file.tar.gz*", deve ser utilizado o seguinte comando:

```
tar -zxvf file.tar.gz
```

2. Geração de arquivo *.tar.gz*

Uma estrutura de diretórios existente no disco pode ser completamente copiada e compactada para um arquivo *tar.gz*. Supondo que se deseja copiar o conteúdo do diretório de nome "*dir*", incluindo seus arquivos e subdiretórios, para um único arquivo *tar.gz* de nome "*file.tar.gz*", deve-se, a partir do diretório pai do diretório "*dir*", usar o seguinte comando:

```
tar -zcvf file.tar.gz dir
```