Trabalho Prático I – Entrega 28 de SETEMBRO de 2015

Implementação de Biblioteca de *Threads*

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 **Piccolo thread** (ou apenas *pithread*), deverá oferecer capacidades básicas para programação com *threads* como criação, execução, sincronização, término e trocas de contexto.

Ainda, a biblioteca *pithread* deverá ser implementada, OBRIGATORIAMENTE, na linguagem C e sem o uso de outras bibliotecas (além da *libc*, é claro). Além disso, a implementação deverá executar em ambiente GNU/Linux e será testada na máquina virtual *alunovm-sisop.ova*.

2. Descrição Geral

A biblioteca *pithread* deverá ser capaz de gerenciar uma quantidade variável de *threads* (potencialmente grande), limitada pela capacidade de memória RAM disponível na máquina. Cada *thread* deverá ser associada a um identificador único (*tid – thread identifier*) que será um número inteiro, 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 *picreate*); (ii) cedência voluntária (primitiva *piyield*); (iii) quando essa *thread* está bloqueada esperando para entrar em uma seção crítica (*pilock*) e outra *thread* libera essa seção crítica (primitiva *piunlock*) e (iv) quando essa *thread* estiver bloqueada pela primitiva *piwait*, esperando por uma outra *thread*, e essa outra *thread* terminar.

Executando: representa o estado em que a *thread* está usando o processador. Uma *thread* nesse estado pode passar para os estados *apto*, *bloqueado* ou *término*. Uma *thread* passa para *apto* sempre que executar uma primitiva *piyield*. Uma *thread* pode passar de *executando* para *bloqueado* através da execução das primitivas *piwait* ou *pilock*. 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* sempre que executar uma primitiva *piwait*, para esperar a conclusão de outra *thread*, ou ao tentar entrar em uma seção crítica – primitiva *pilock* – e a mesma já estiver sendo usada por outra *thread*.

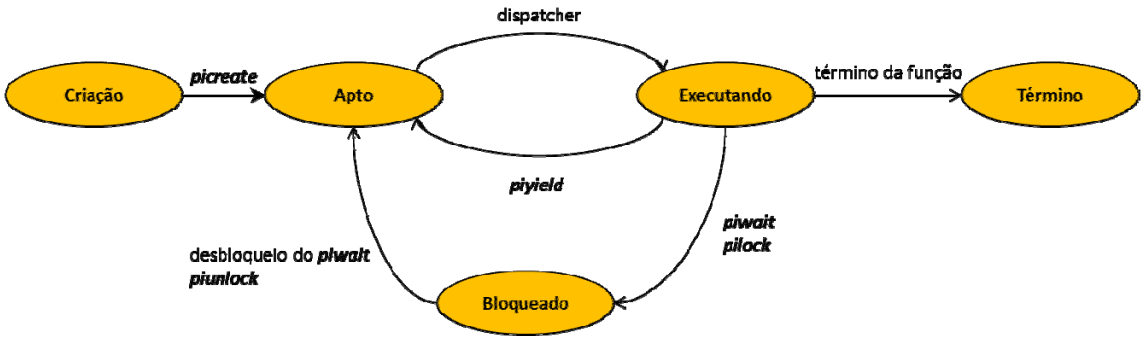

Figura 1 – Diagrama de estados e transições da *pithread*

O escalonador a ser implementado é do tipo **não preemptivo** e deve seguir uma **política de MULTIPLAS FILAS com prioridades dinâmicas**, onde as *threads* de mesma prioridade seguem uma política FIFO. As prioridades dinâmicas são implementadas com o auxílio de um sistema de créditos. Nessa política, as *thread*s são selecionadas para execução segundo sua prioridade, que é determinada pelos créditos que a mesma possui: quanto mais créditos, maior a prioridade. Ao ser criada, uma *thread* recebe uma certa quantidade de créditos, que corresponde à prioridade inicial da mesma.

Assim, quando a CPU ficar livre, o escalonador deverá selecionar a primeira *thread* da fila de aptos, de maior prioridade, que não estiver vazia, para receber a CPU (passar para o estado *executando*).

Sistema de duas filas de aptos

Para evitar postergação indefinida, o escalonador possuirá duas filas de *aptos*: uma fila denominada de *aptos-ativos* e outra de *aptos-expirados*. As *threads* passam de uma fila para a outra da seguinte forma:

- Quando as *threads* forem criadas, elas serão inseridas na fila de *aptos-ativos*, na posição (prioridade) correspondente aos seus créditos iniciais.
- Sempre que uma *thread* sair do estado de executando, está terá seus créditos reduzidos em 10 unidades. Notar que uma *thread* não pode ter um número negativo de créditos (se isso acontecer, deve-se atribuir zero ao número de créditos). Além disso, se a *thread* estiver retornando para o estado de *aptos-ativos*, ela será inserida na fila correspondente a sua nova prioridade.
- Quando uma *thread* não tiver mais créditos, ela será movida para a fila de *aptos-expirados*. Essa *thread* deverá receber uma quantidade de créditos igual aquela atribuída na sua criação e deverá ser inserida na fila de *aptos-expirados* correspondente a sua prioridade (determinada pela quantidade de créditos na criação).
- Quando não houver mais nenhuma *thread* na fila *aptos-ativos*, essa se tornará a nova fila *aptos-expirados*, e a fila *aptos-expirados* se tornará a nova fila *aptos-ativos*.
- Os processos que forem desbloqueados receberão 20 unidades de créditos, estando limitados a 100, e serão inseridos na fila atual de *aptos-ativos*, na posição correspondente aos seus créditos.

3. Interface de programação

A biblioteca *pithread* deve oferecer uma interface de programação (API) que permita o seu uso para o desenvolvimento de programas. O grupo deverá desenvolver as funções dessa API, conforme descrição a seguir, que deve ser RIGOROSAMENTE respeitada.

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-ativos* na posição correspondente a sua prioridade. A função da biblioteca responsável pela criação de uma *thread* é a *picreate*. Essa função recebe como parâmetro o número de créditos a serem atribuídos a *thread*. Esse valor pode variar entre 1 a 100 (créditos = 1,2,3...,100). 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.

```
int picreate (int credCreate, void *(*start)(void *), void *arg);
```

Parâmetros:

`credCreate`: número de créditos atribuídos a threads (mínimo: 1; máximo: 100)
`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*)

Retorno:

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

A estrutura de dados usada para definir o TCB (*Thread Control Block*) deverá ser, OBRIGATORIAMENTE, aquela fornecida abaixo. Os campos da estrutura foram especificados de maneira a possibilitar as funcionalidades solicitadas. Entretanto, em função de sua implementação, alguns deles podem não ser utilizados. Esse, por exemplo, é o caso dos campos *ant* e *next*: os campos *ant* e *next* só serão utilizados para listas duplamente encadeadas enquanto que, em listas simples, apenas o campo *next* será utilizado

```
typedef struct TCB {
      int           tid;          // identificador da thread
      int           state;        // estado em que a thread se encontra
                                  // 0: Criação; 1: Apto; 2: Execução; 3: Bloqueado e 4: Término
      int           credCreate;   // créditos atribuídos na criação (corresponde à prioridade inicial)
      int           credReal;     // créditos atuais da thread (usado para determinar a prioridade atual da thread)
      ucontext_t    context;      // contexto de execução da thread (SP, PC, GPRs e recursos)
      struct TCB    *prev;        // ponteiro para o TCB anterior da lista
      struct TCB    *next;        // ponteiro para o próximo TCB da lista
} TCB_t;
```

Liberando voluntariamente a CPU: uma *thread* pode liberar a CPU de forma voluntária com o auxílio da primitiva *piyield*. Se isso acontecer, a *thread* que executou *piyield* retorna ao estado *apto-ativos*, sendo reinserida na fila de *aptos-ativos*, segundo sua nova prioridade (lembrar que ela perda 10 créditos ao sair de *executando*). Então, o escalonador será chamado para selecionar a *thread* que receberá a CPU.

```
int piyield(void);
```

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

Sincronização de término: uma *thread* pode ser bloqueada até que outra termine sua execução usando a função *piwait*. A função *piwait* recebe como parâmetro o identificador da *thread* cujo término está sendo aguardado. Quando essa *thread* terminar, a função *piwait* 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 *piwait* para uma mesma *thread*, apenas uma delas, a primeira que realizou a chamada, será bloqueada. As outras chamadas retornarão imediatamente com um código de erro. Se *piwait* for feito para uma *thread* que não existe (não foi criada ou já terminou), a função retornará imediatamente com o código de erro. Observe que não há necessidade de um estado *zombie*, pois a *thread* que aguarda o término de outra (a que fez *piwait*) não recupera nenhuma informação de retorno proveniente da *thread* aguardada.

```
int piwait(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.

Exclusão mútua: o sistema prevê o emprego de semáforos binários para realizar a sincronização de acesso a recursos compartilhados (seção crítica). As primitivas existentes são *pimutex_init*, *pilock* e *piunlock*, e usam uma variável semáforo que recebe o nome específico de MUTEX (de *MUTtual EXclusion*). A primitiva *pimutex_init* é usada para inicializar a variável *pimutex_t* e deve ser chamada, obrigatoriamente, antes da variável ser usada com as primitivas *pilock* e *piunlock.*

A estrutura de dados abaixo deverá ser usada para as variáveis *mutex*.

```
typedef struct mutex {
      int           flag;          // indica se o mutex foi adquirido ou não (1: livre; 0: ocupado)
      struct TCB    *first, *last; // ponteiros para lista de threads bloqueadas no mutex
} pimutex_t;
```

A função *pimutex_init* inicializa uma variável do tipo *pimutex_t* e consiste em colocá-la no estado livre. Isso é, liberado para que uma *thread* possa adquiri-lo. Ainda, cada *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 pimutex_init (pimutex_t *mtx);
```

Parâmetros:
mtx: ponteiro para uma variável do tipo *pimutex_t*. Esse ponteiro aponta para uma estrutura de dados que representa o *mutex*.
Retorno:
Quando executada corretamente: retorna 0 (zero)
Caso contrário, retorna um valor negativo.

A primitiva *pilock* será usada para indicar a entrada na seção crítica. Se a seção crítica estiver livre, a entrada da *thread* corrente na seção crítica é autorizada, e o valor da variável do tipo *pimutex_t* deve passar para ocupado. Se, por outro lado, a seção crítica estiver ocupada, a *thread* será bloqueada (transição de *executando* para *bloqueado*) e seu *tid* deverá ser armazenado para, posteriormente, fazer sua liberação (desbloqueio).

```
int pilock (pimutex_t *mtx);
```

Parâmetros:
mtx: ponteiro para uma variável do tipo *pimutex_t*.
Retorno:
Quando executada corretamente: retorna 0 (zero)
Caso contrário, retorna um valor negativo.

Ao encerrar a execução da seção crítica, a *thread* deverá chamar a *piunlock* para liberar a variável de *mutex*. Se houver mais de uma *thread* bloqueada nesse *mutex*, a primeira delas, segundo uma política de FIFO, deverá passar para o estado *aptos-ativos* e as demais devem continuar no estado *bloqueado*.

```
int piunlock (pimutex_t *mtx);
```

Parâmetros:
mtx: ponteiro para uma variável do tipo *pimutex_t*.
Retorno:
Quando executada corretamente: retorna 0 (zero)
Caso contrário, retorna um valor negativo.

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 *pithread*, existirão dois tipos de *threads*: *thread main* (criada pelo sistema operacional) e as *threads* de usuário (criadas através das chamadas *picreate*). 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 *pithread* para, posteriormente, em trocas de contexto da *main* para as *threads* criadas pelo *picreate*, 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 com as *threads* criadas com a *picreate*.
- Como o escalonador é orientado a prioridades, é preciso definir uma prioridade para a *thread main*, uma vez que a definição da prioridade das *threads* é feita na chamada *picreate* e a *main* não é criada por essa chamada. Por escolha de projeto, definiu-se que a thread *main* deverá a maior prioridade possível. No caso, a *thread main* deve receber 100 créditos. Ainda, a *thread main* deverá ter um identificador único (*tid*) que será atribuído o valor ZERO. Dessa forma, a *thread main* possui um TCB como as demais *threads*.

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 *pithread.tar.gz* fornecido (conforme seção 6).

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:

- O arquivo *tar.gz* a ser gerado deve ter o nome formado pelos números de cartão dos componentes do grupo. Por exemplo, supondo que esses números são 123456 e 654321, o arquivo deverá ter o nome "123456_654321.tar.gz";
- Entregue o arquivo *.tar.gz* via Moodle.

ATENÇÃO:

- **NÃO** inclua, no *tar.gz*, cópia da Máquina Virtual;
- **NÃO** serão aceitos outros formatos de arquivos, tais como *.rar* ou *.zip*.

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.

6. Arquivo *.tar.gz*

Será fornecido pelo professor (disponível no Moodle) um arquivo *pithread.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 *pithread*) da estrutura de diretórios do arquivo *pithread.tar.gz* está disponibilizado um arquivo *Makefile* de referência, que deve ser completado de maneira a gerar a biblioteca (ver seção 8). Para a entrega, nesse diretório deve ser colocado o arquivo PDF de relatório (conforme seção 0). Os subdiretórios do diretório *pithread* são os seguintes:

- *bin*: local onde colocar todos os arquivos objetos (arquivos *.o*) gerados pela compilação da biblioteca;
- *exemplos*: local onde estão os programas fonte de exemplo fornecidos pelo professor e o *makefile* para geração dos executáveis. Os arquivos resultantes da compilação serão colocados nesse mesmo subdiretório.
- *include*: local onde colocar todos os seus arquivos de include (arquivos *.h*). Nesse subdiretório está disponível o arquivo *pithread.h* e *pidata.h* (seção 7), de uso obrigatório;
- *lib*: local onde colocar a biblioteca gerada (*libpithread.a*);
- *src*: local onde colocar todos os seus arquivos fonte (arquivos *.c*) da implementação da biblioteca;
- *testes*: diretório de trabalho para a geração dos 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).

Para criar programas de teste que utilizem a biblioteca *pithread* siga os procedimentos da seção 9.

7. Arquivo *pithread.h* e *pidata.h*

Os protótipos das funções da biblioteca que definem a API estão declarados no arquivo *pithread.h*, de uso obrigatório.

Esse arquivo estará no subdiretório *include* da estrutura de diretórios fornecida no arquivo *pithread.tar.gz* e **não pode ser alterado**.

Qualquer inclusão que seja necessária deve ser feita no arquivo denominado *pidata.h*, cujo conteúdo poderá ser definido pelo grupo, à exceção da *struct TCB* que deverá ser aquela definida nesta especificação. O arquivo *pidata.h* deverá ser colocado no subdiretório *include*.

8. Geração da *libpithread* (descrição do Makefile)

As funcionalidades da *pithread* deverão ser disponibilizadas através da biblioteca denominada *libpithread.a*. Uma biblioteca é um tipo especial de programa objeto em que suas funções são chamadas por outros programas. Para isso, o programa chamador deve 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*.

Notar que o programa fonte do chamador deve incluir o arquivo de cabeçalho (*header files*) *pithread.h* com os protótipos das funções disponibilizadas pelo arquivo *libpithread.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 *libpithread.a*, no diretório *lib*.
- Regra "*clean*": responsável por remover todos os arquivos dos subdiretórios *bin* e *lib*.

9. Utilizando a *pithread*: 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 C. Todas as funções da biblioteca podem ser chamadas pela *main*. Por exemplo, pode-se chamar a *piwait()* 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 *pithread* (ver ANEXO I sobre como ligar os programas à biblioteca). Então, o programa executável resultante poderá ser executado.

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

A biblioteca deve possuir todas as funções da API, mesmo que não tenham sido implementadas. Nesse caso, devem apenas retornar código de erro.

10. Relatório

Além da implementação, o grupo deve entregar um relatório– arquivo PDF – que consiste em responder as questões formuladas abaixo:

1. Nome dos componentes do grupo e número do cartão.
2. Indique, para cada uma das funções que formam a biblioteca *pithread*, (*picreate, piyield, piwait, pilock* e *piunlock*) se as mesmas estão funcionando corretamente ou não. Para o caso de não estarem funcionando adequadamente, descrever qual é a sua visão do porquê desse não funcionamento.
3. Descreva os testes realizados pelo grupo e se o resultado esperado se concretizou. Cada programa de teste elaborado e entregue pelo grupo deve ter uma descrição de operação, quais as saídas fornecidas e os resultados finais esperados. Observe que as primitivas da biblioteca devem ser testadas por, **pelo menos**, um programa de teste.
4. Quais as principais dificuldades encontradas na implementação do trabalho e quais as soluções empregadas para contorná-las.

11. *Road map* para a implementação

Algumas dicas do que precisará ser feito:

- É preciso entender o correto funcionamento das primitivas *u_context* (atividade experimental 2) e utilizar a estrutura de dados fornecida em *pidata.h* para representar uma *thread.* No TCB estão todas as informações relativas a uma *thread* (*tid*, estado, contexto, *etc*.);
- Devem ser implementadas rotinas para tratamento de listas encadeadas prevendo inserção e a retirada de elementos. Exemplo de elementos dessas listas são os TCBs e os MUTEXes;
- Deve ser implementado o escalonador, com a política solicitada, e o despachante (*dispatcher*);
- Deve ser elaborado um conjunto de programas de testes.

12. 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. Na Internet pode-se encontrar várias implementações de bibliotecas de *threads* similares ao que está sendo solicitado. ENTRETANTO, NÃO SE ILUDAM!! NÃO É SÓ COPIAR!! Esses códigos são muitos mais completos e complexos do que vocês precisam fazer. Eles servem como uma boa fonte de inspiração. A base para elaboração e manipulação das *pithread* são as chamadas de sistema providas pelo GNU/Linux: *makecontext*(), *setcontex*t(), *getcontext*() e *swapcontext*(). Estude o comportamento dessas funções.

13. Critérios de avaliação

A avaliação do trabalho considerará as seguintes condições:

- Entrega 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.*
- O relatório deverá estar completo.

Itens que serão avaliados e sua valoração:

- 10,0 pontos: clareza e organização do código, programação modular, *makefiles*, arquivos de inclusão bem feitos (sem código C dentro de um include!!) e comentários adequados;
- 20,0 pontos: respostas ao questionário, correta associação entre a implementação e os conceitos vistos em aula, e explicação e funcionamento (na prática) dos programas de teste desenvolvidos para verificar o funcionamento de cada primitiva da biblioteca *pithread*;
- 70,0 pontos: funcionamento da *pithread* de acordo com a especificação. Para essa verificação serão utilizados programas padronizados desenvolvidos pelo professor.

14. Data de entrega e avisos gerais – LEIA com MUITA ATENÇÃO

1. Data de entrega: **28 de SETEMBRO de 2015**.
2. Faz parte da avaliação a obediência RÍGIDA aos padrões de entrega definidos na seção 6 (arquivos *tar.gz*, estrutura de diretórios, *makefile*,etc).
3. O trabalho poderá ser desenvolvido INDIVIDUALMENTE ou em DUPLAS. Salienta-se que DUPLAS são formadas exclusivamente por DOIS alunos, NÃO será aceito nenhuma exceção. Da mesma forma, apesar do trabalho ser idêntico entre as duas turmas, NÃO serão permitidas duplas com alunos pertencentes a turmas diferentes.
4. O trabalho 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.
5. Admite-se a entrega do trabalho com até UMA semana de atraso (**05 de OUTUBRO de 2015**). Nesse caso, o trabalho será avaliado e, da nota alcançada (de um total de 100,0 pontos) será diminuído 20,0 pontos pelo atraso. Não serão aceitos trabalhos entregues além dos prazos estabelecidos.

15. 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.

O professor da disciplina reserva-se o direito de solicitar uma demonstração do programa com a presença de todo o grupo. A nota final será baseada nos parâmetros acima e na arguição sobre questões de projeto e de implementação feitas ao(s) aluno(s).

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
```