

## Trabalho Prático I

### Implementação de Biblioteca de *Threads* *cthread* 16.2

#### 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.

Ainda, a biblioteca *cthread* 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 *cthread* 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 *ccreate*); (ii) cedência voluntária (primitiva *cyield*); (iii) quando essa *thread* está bloqueada esperando por um recurso (*cwait*) e outra *thread* libera esse recurso (primitiva *csignal*) e (iv) quando essa *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* nesse estado pode passar para os estados *apto*, *bloqueado* ou *término*. Uma *thread* passa para *apto* sempre que executar uma primitiva *cyield*. Uma *thread* pode passar de *executando* 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* sempre que executar uma primitiva *cjoin*, para esperar a conclusão de outra *thread*, ou ao tentar entrar em usar um recurso – primitiva *cwait* – e o mesmo já estiver sendo usado por outra *thread*.

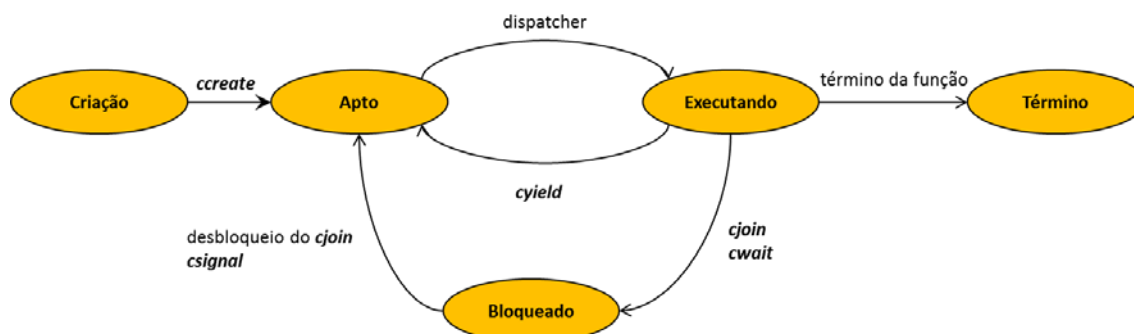


Figura 1 – Diagrama de estados e transições da *cthread*

O escalonador a ser implementado é do tipo **não preemptivo, sem prioridades**, e deve seguir uma **política de loteria**. A ideia básica é dar um “bilhete de loteria” para as *threads* no momento de sua criação. O bilhete de loteria corresponde a um número entre 0 e 255, gerado aleatoriamente. Sempre que a CPU ficar livre e, por consequência, for necessário tomar uma decisão de escalonamento, será feito o sorteio da CPU. Esse sorteio corresponde à geração de um novo

número aleatório, de forma semelhante a um sorteio de loteria: sorteia-se um número e a *thread* que tiver o bilhete com número mais próximo ao sorteado passará do estado *Apto* para *Executando*. Se duas ou mais *threads* tiverem bilhetes com o mesmo número e forem as escolhidas pelo sorteio, ganhará a CPU aquela que tiver o menor identificador de *thread* (*tid*).

A função para geração dos números randômicos será fornecida pelos professores da disciplina.

### 3. Interface de programação

A biblioteca *cthread* 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*. 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.

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

**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*)

**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.

**ATENÇÃO:** é obrigatório o uso da biblioteca *support* fornecida pelos professores da disciplina. Junto com o material fornecido para a realização deste trabalho você encontrará um binário (*support.o*) que implementa várias funções e o arquivo de inclusão (*support.h*) correspondente. As filas de processos devem utilizar essas primitivas e, portanto, os elementos a serem encadeados são instâncias da estrutura TCB. A função de geração randômica também faz parte do binário *support.o* e seu protótipo também está no arquivo *support.h*.

```
typedef struct s_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      ticket;        // "bilhete" de loteria da thread, para uso do escalonador
    ucontext_t context;     // contexto de execução da thread (SP, PC, GPRs e recursos)
} TCB_t;
```

**Liberando voluntariamente a 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*, sendo reinserida na fila de *aptos*. 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.

**Sincronização de término:** uma *thread* pode ser bloqueada até que outra termine sua execução usando a função *cjoin*. 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 uma mesma *thread*, apenas a primeira que realizou a chamada será bloqueada. As outras chamadas retornarão imediatamente com um código de erro. Se *cjoin* 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 *cjoin*) não recupera nenhuma informação de retorno proveniente da *thread* aguardada.

```
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 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 da variável ser usada com as primitivas *cwait* e *csignal*.

A estrutura de dados abaixo deverá ser usada, OBRIGATORIAMENTE, para as variáveis *semáforo*.

```
typedef struct s_sem{
    int count; // indica se recurso está ocupado ou não (livre > 0, ocupado ≤ 0)
    PFILA2 fila; // ponteiro para uma fila de threads bloqueadas no semáforo.
} csem_t;
```

**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*, esse valor inicial da variável *semáforo* deve ser 1. 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 a espera desse recurso na fila. 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. Para cada chamada a *csignal* a variável *count* 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*.

```
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 *cthread*. 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.

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

#### 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.

#### 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).

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:

**ATENÇÃO:**

- **NÃO** inclua, no *tar.gz*, cópia da Máquina Virtual;

- **NÃO** serão aceitos outros formatos de arquivos, tais como *.tgz*, *.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 *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). Para a entrega, nesse diretório deve ser colocado o arquivo PDF de relatório (conforme seção 10). Os subdiretórios do diretório *cthread* são os seguintes:

\cthread		
	bin	DIRETÓRIO: local estarão todos os arquivos objetos (arquivos <i>.o</i> ) gerados pela compilação da biblioteca. Nesse subdiretório está disponível o arquivo <i>support.o</i> , de uso obrigatório. Nesse arquivo está a implementação das funções de gerenciamento de filas, conforme especificação da primeira atividade experimental;
	include	DIRETÓRIO: local onde colocar todos os seus arquivos de include (arquivos <i>.h</i> ). Nesse subdiretório está disponível o arquivo <i>cthread.h</i> e <i>cdata.h</i> (seção 7), de uso obrigatório. Também estará disponível nesse subdiretório o arquivo <i>support.h</i> , com os protótipos das funções de gerenciamento de filas;
	lib	DIRETÓRIO: local onde colocar a biblioteca gerada ( <i>libcthread.a</i> );
	src	DIRETÓRIO: local onde são postos todos os arquivos ".c" (códigos fonte) usados na implementação de <i>cthread</i> 16.02.
	exemplos	DIRETÓRIO: local onde estão os programas fonte de exemplos fornecidos pelo professor e o <i>makefile</i> para geração dos executáveis. Os arquivos resultantes da compilação serão colocados nesse mesmo subdiretório
	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 <i>makefile</i> para sua geração (ver seção 8).
	makefile	ARQUIVO: arquivo <i>makefile</i> com regras para gerar a " <i>libcthread</i> ". Deve possuir uma regra " <i>clean</i> ", para limpar todos os arquivos gerados.
	relatorio.pdf	ARQUIVO: arquivo PDF com as respostas do questionário.

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

## 7. Arquivo *cthread.h* e *cdata.h*

Os protótipos das funções da biblioteca que definem a API estão declarados no arquivo *cthread.h*, de uso obrigatório. Esse arquivo estará no subdiretório *include* da estrutura de diretórios fornecida no arquivo *cthread.tar.gz* e **não pode ser alterado**.

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

## 8. 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 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*. 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*) *pthread.h* com os protótipos das funções disponibilizadas pelo arquivo *libpthread.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 *libpthread.a*, no diretório *lib*.
- Regra “*clean*”: responsável por remover todos os arquivos dos subdiretórios *bin* e *lib*.

## 9. Utilizando a *pthread*: 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 *pthread* (ver ANEXO I sobre como ligar os programas à biblioteca). Então, o programa executável resultante poderá ser executado.

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

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

## 10. 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 *pthread* são as chamadas de sistema providas pelo GNU/Linux: *makecontext()*, *setcontext()*, *getcontext()* e *swapcontext()*. Estude o comportamento dessas funções.

## 11. Critérios de avaliação

---

A avaliação do trabalho considerará as seguintes condições:

- Participação às reuniões de projeto;
- Entrega 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.o*.

Itens que serão avaliados e sua valoração:

- 50,0 pontos: reuniões de acompanhamento, onde serão verificados:
  - a correta associação entre a implementação e os conceitos vistos em aula
  - os programas de teste propostos e desenvolvidos pelo grupo para verificar o funcionamento de cada primitiva da biblioteca *cthread*;
  - as explicações de funcionamento do código, bem como a clareza e organização;
  - o uso de programação modular
  - correta utilização dos makefiles
  - correta utilização dos arquivos de header (sem código “C” dentro dos includes)
- 50,0 pontos: funcionamento da *cthread* de acordo com a especificação. Para essa verificação serão utilizados programas padronizados desenvolvidos pelos professores.

## 12. 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 com três componentes.
3. As reuniões de projeto serão avaliadas e as notas alcançadas farão parte da nota final do trabalho. A ausência nas reuniões de projeto implicam em nota 0 (zero) naquela avaliação.
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.

## 13. 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 subdiretório *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
```