```
}
int Multiplica_Numeros(int A, int B){
  return A*B;
}
int Divide_Numeros(int A, int B){
  return A/B;
}
```

Tudo certo. Sua mini biblioteca está pronta. Para utilizá-la em outros programas, basta incluir o arquivo do cabeçalho ".h", conforme abaixo:

```
#include "operacoes.h"
int main(){
  int A=2,B=2;
  printf("%d",Soma_Numeros(A,B));
  ...
}
```

Você já sabe agora o básico para modularizar seus programas. O exemplo acima é didático mas, ao longo dos TPs, você sentirá necessidade de modularizar seu código. Por exemplo, um trabalho pode exigir que você use uma "Lista Encadeada". Para isso, você poderia criar uma "lista.h", que definirá as estruturas e as funções de operações sobre as listas e uma "lista.c", onde ficarão as implementações das operações.

6 Fazendo um Makefile

A primeira pergunta é: O que é um Makefile? É um arquivo que define as diretrizes de compilação do seu código. Muita das vezes, quando o código, em "C" está em um arquivo só, basta executar o comando abaixo:

```
gcc <flags> <arquivo.c> -o <binario>
```

O problema surge quando existem vários arquivos a serem mesclados para gerar o código final. Isso ocorre quando o código está dividido em diversos arquivos para uma melhor modularização (Direcionamento 4). Para isso, existe um programa "Make" que, quando invocado (*make*), chama um arquivo chamado "Makefile" dentro do diretório corrente. O arquivo "Makefile" contem as instruções de compilação do programa. A sintaxe básica de um arquivo "Makefile" é:

```
<nome da regra>: <dependencias>
<TAB> <comando>
```

Cada regra é lida e as dependências são arquivos ou regras. No exemplo da mini biblioteca no Direcionamento 4, com os arquivos "main.c", "operacoes.h" e "operacoes.c", poderíamos fazer criar o seguinte arquivo, com nome "Makefile", dentro do diretório onde estão os fontes do programa:

```
operacoes: operacoes.o main.o
<TAB> gcc operacoes.o main.o -o operacoes
main.o: main.c operacoes.h
```

```
4 <TAB> gcc -g -c main.c
5 operacoes.o: operacoes.h operacoes.c
6 <TAB> gcc -g -c operacoes.c
```

A primeira linha define o objetivo que queremos: o binário "operacoes". Entretanto, para alcançar esse objetivo (executar a regra de "operacoes"), é necessário os arquivos "operacoes.o" e "main.o", que são códigos já traduzidos para linguagem de máquina (binário) mas que ainda não foram ligados com as funções externas a ele. Por exemplo, main.o sabe que a função "Soma_Numeros" está definida em algum endereço de memória mas não sabe exatamente qual é o endereço. Chamados esse binário de "objeto(.o)". A pergunta é: porque tudo não é passado diretamente para código de máquina? A resposta é simples: eficiência. Quando a tradução do código para binário ocorre, isso gera um custo que, em algumas vezes, é caro (demora muito tempo). Dessa forma, não é eficiente traduzir todo o código se apenas uma pequena modificação for feita. Por isso, o programa é dividido em vários ".o" e, quando uma modificação é feita em apenas um deles, apenas o ".o" em questão é compilado novamente. Tudo que você precisa saber é isso. Para maiores detalhes, curse as disciplinas de "Software Básico" e "Compiladores".

Definimos, então, como criar os arquivos "operacoes.o" e "main.o" nas linhas 3 e 5. Observe que a regra "main.o" já tem suas dependências resolvidas e já pode ter o seu comando executado. O comando invoca o "gcc" para compilar seu programa. Entretanto, observe que não podemos compilar o programa inteiro pois "main.c" usa funções que não são implementadas dentro do arquivo (Soma_Numeros(int,int)) mas sabe que elas existem em um outro arquivo, através do "operacoes.h" que está incluído em seu código. Por isso, a compilação deve ser feita parcialmente, produzindo um binário tal que, quando todas as partes forem mescladas, as ligações sejam feitas corretamente. Para isso, utilizamos a flag "-c". O mesmo é feito para a regra "operacoes.o" (a flag "-g" é explicada no Direcionamento 6). Ao final, a regra "operacoes" passa a ter seus pré-requisitos preenchidos e executa o seu comando, que é ligar todos os arquivos ".o" (objetos). Para executar esse código, apenas digite "make". A figura 1 mostra um exemplo do make executado em um terminal Linux:

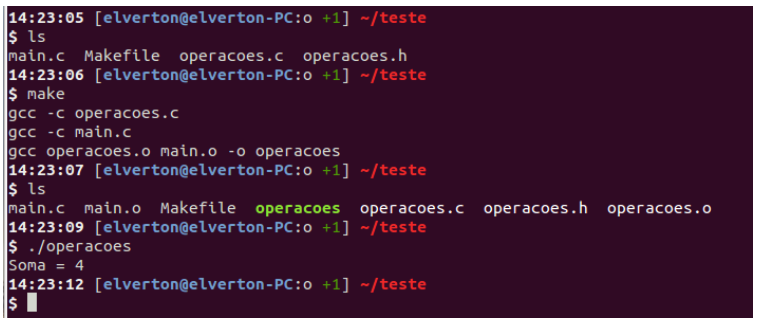

Figura 1: Executando um Makefile no terminal Linux.

Para mais detalhes, utilize o código do tutorial do "Make" no moodle da disciplina.