```
;==============================================================================
;==============================================================================
; Macros
;==============================================================================
;==============================================================================

%assign SYS_exit                1
%assign SYS_read                3
%assign SYS_write               4
%assign SYS_open                5
%assign SYS_close               6
%assign SYS_brk                 45

%assign O_RDONLY        0
%assign O_WRONLY        1
%assign O_RDWR          2
%assign O_ACCMODE       3

;==============================================================================
; A convenience macro that pushes the registers given as arguments. It is used
; as a means of saving these values when I call procedures in my code. It is
; also used in other macros below. It pushes the last params first. %0 is the
; number of params passed. %1... are the registers to push.
;==============================================================================
%macro  Push_Regs 1-8

; Push these registers, last arg first.

   %if %0 >= 1
      %if %0 >= 2
         %if %0 >= 3
            %if %0 >= 4
               %if %0 >= 5
                  %if %0 >= 6
                     %if %0 >= 7
                        %if %0 = 8
                           push %8
                        %endif
                        push %7
                     %endif
                     push %6
                  %endif
                  push %5
               %endif
               push %4
            %endif
            push %3
         %endif
         push %2
      %endif
      push %1
   %endif
%endmacro

;==============================================================================
; The analog to Push_Regs. It pops the registers from first to last, which
; correctly restores the params as pushed by Push_Regs. %0 is the number of
; params passed. %1... are the registers to pop.
;==============================================================================
%macro  Pop_Regs 1-8

; Pop the registers, first arg first.

   %if %0 >= 1
```

```
      pop %1
      %if %0 >= 2
         pop %2
         %if %0 >= 3
            pop %3
            %if %0 >= 4
               pop %4
               %if %0 >= 5
                  pop %5
                  %if %0 >= 6
                     pop %6
                     %if %0 >= 7
                        pop %7
                        %if %0 = 8
                           pop %8
                        %endif
                     %endif
                  %endif
               %endif
            %endif
         %endif
      %endif
   %endif
%endmacro

;=============================================================================
; Linux_Sys_Call uses int 0x80 as the mechanism to access the low level system
; call interface. All params are passed in the registers, which are loaded from
; the parameters to the macro before the call. %0 is the number of params
; passed.  %1 is the syscall name, %2 is the number of registers to prepare,
; %3... are the values for the registers. All registers except eax are saved
; since the function returns a value in eax. It is safe to assume that esp and
; ebp are returned to their original values after the call.
;=============================================================================
%macro  Linux_Sys_Call 1-7

; Push these registers since its unpredictable which registers are destroyed
; in the system call.

   Push_Regs ebx, ecx, edx, esi, edi

; I'm not sure if there are system calls with 0 params, but just in case.

   %if %0 >= 2

; If the number of registers is given, use its value to determine the number
; of arguments to move into the registers. Note that the usage of the registers
; is well defined for the system calls.

      %if %2 >= 1
         %if %2 >= 2
            %if %2 >= 3
               %if %2 >= 4
                  %if %2 >= 5
                     mov edi, %7
                  %endif
                  mov esi, %6
               %endif
               mov edx, %5
            %endif
            mov ecx, %4
         %endif
         mov ebx, %3
      %endif
   %endif
```

```
; Make the system call using the assigns at the top of the file.

   mov   eax, SYS_%{1}
   int   0x80

; Restore the registers.

   Pop_Regs ebx, ecx, edx, esi, edi

%endmacro

;============================================================================
; The system calls that I use in this project (or may eventually use).
;============================================================================

;============================================================================
; The open system call is a low level call which returns an integer as a file
; handle. Note fopen returns a structure and is a higher level call -- this one
; is much simplier. The parameters are a string giving the file name (1st), the
; flags (see defines at the top of the file for a subset of these -- 2nd param)
; and the mode (used when a file is created, determines the permissions -- 3rd
; param).
;============================================================================

%macro Sys_Open 0-3
   Linux_Sys_Call open, %0, %1, %2, %3
%endmacro

%macro Sys_Close 0-1
   Linux_Sys_Call close, %0, %1
%endmacro

;============================================================================
; The read system call simply returns the number of bytes requested (3rd param)
; in the buffer space (2nd param) using the file handle (1st param) returned
; from the open system call. Write is the analog and has the same params.
;============================================================================

%macro Sys_Read 0-3
   Linux_Sys_Call read, %0, %1, %2, %3
%endmacro

%macro Sys_Write 0-3
   Linux_Sys_Call write, %0, %1, %2, %3
%endmacro

;============================================================================
; Not used yet -- I think this is the mechanism to allocate dynamic memory.
;============================================================================

%macro  Sys_Brk 0-1
   Linux_Sys_Call brk, %0, %1
%endmacro

;============================================================================
; Exit the program with the error code given as the only parameter.
;============================================================================

%macro Sys_Exit 0-1
   Linux_Sys_Call exit, %0, %1
%endmacro

;============================================================================
; C_Sys_Call prepares the stack and calls the clib function specified as the
```

```
; first parameter. %0 is the number of params passed. %1 is the syscall name,
; %2 is the number of parameters to push, %3... are the values for the stack.
; All registers except eax are saved since the function returns a value in eax.
; It is safe to assume that esp and ebp are returned to their original values
; after the call.
;============================================================================
%macro  C_Sys_Call 1-7

   Push_Regs ebx, ecx, edx, esi, edi

   %if %0 >= 2
      %if %2 >= 1
         %if %2 >= 2
            %if %2 >= 3
               %if %2 >= 4
                  %if %2 >= 5
                     push dword %7
                  %endif
                  push dword %6
               %endif
               push dword %5
            %endif
            push dword %4
         %endif
         push dword %3
      %endif
   %endif

extern %{1}
   call %{1}
   add esp, (%{2}) * 4

   Pop_Regs ebx, ecx, edx, esi, edi
%endmacro

;============================================================================
; Simple MASM equivalents to Enter and Leave. These generate a call frame and
; allocate/deallocate space for local parameters.
;============================================================================

%macro  Enter 1
   push ebp
   mov ebp, esp
   sub esp, %1
%endmacro

%macro  Leave 0
   mov esp, ebp
   pop ebp
%endmacro

;============================================================================
; A convenience macro for putting in DEBUG statements easily (at least at this
; point). This macro allocates data space for a double word, which will store
; the computed length of the string passed as the 3rd.... parameter(s). The
; first and second params are the labels used for these storage locations. Note
; the version given in the documentation doesn't work under linux -- this one
; does :)
;============================================================================

%macro Make_Local_Str 3+
            section .data
%1          dd 4
%2          db %3
%%endstr:
```

```
            section .text
            mov dword [%1], %%endstr - %2
%endmacro


;=============================================================================
; if-then-else macro retrieved from the user manual on NASM. These use context
; stack via %push context_name. Note that %-1 inverts the condition code, e.g.,
; e to ne, ge to le, etc. The ifctx context_name checks that the top context on
; the stack is context_name. repl replaces the context with the new name.
;=============================================================================

%macro if 1
    %push if
    j%-1 %$ifnot
%endmacro

%macro else 0
    %ifctx if
        %repl else
        jmp %$ifend
        %$ifnot:
    %else
        %error "expected `if' before `else'"
    %endif
%endmacro

%macro endif 0
    %ifctx if
        %$ifnot:
        %pop
    %elifctx else
        %$ifend:
        %pop
    %else
        %error "expected `if' or `else' before `endif'"
    %endif
%endmacro
```