The Clown Companion

For All the World Is not a Stage, but a Circus: Round and Funny

Dmitry Zinoviev

Boston

Special thanks to Praveena Salla, Lidia Vikhlyaeva, Tristam MacDonald, Arthur Barrett, Harish Srinivas, and a generation of Suffolk graduate and undergraduate students for their help in developing and debugging Clown, as well as to Vy Duong for proofreading and editing the manuscript.

TABLE OF CONTENTS

4. The Clown Companion

INTRODUCTION

An important part of a college-level operating system course agenda is to examine the interaction between an operating system and computer hardware. Assembly programming teaches students to think logically, waste no byte and no CPU cycle. Knowing the hardware helps students to understand the operation of such foundational mechanisms as memory protection, process dispatching, input/output, and file system organization. It also makes the motivation behind certain OS design decisions clearer. Last, but not least, from the practical point of view, exposing students to low-level programming prepares them for potential projects involving embedded systems and hand-held devices.

Elements of low-level assembly programming can also be found in computer architecture courses. Some universities continue to offer general assembly programming courses where students learn how to extract the ultimate performance from the computer hardware.

Traditionally, colleges have been using various RISC architectures (such as MIPS or RS6000) or Motorola 68x family as their primary hardware platforms. RISC cores are reasonably simple and regular. However, this trend seems to be rapidly disappearing in favor of the industrial mainstream Intel32 architecture. It should also be noted that from the OS development point of view, RISC cores lack many important features, such as segmentation (for superior memory protection) and non-trap-based system call support.

On the other hand, Intel32 CISC architecture is hard to learn. The instruction set is redundant, and the instruction format is highly irregular. This makes Intel32 system programming challenging, especially for undergraduate students. A need clearly exists for a good microprocessor simulator that could be used in an OS course (and possibly in other related courses).

Existing Simulators

Many microprocessor simulators have been developed, but most of them do not address the topic from an OS study point of view.

Some of them simulate RISC or otherwise "inappropriate" targets (e.g., Ant-32, MicSim, Microprocessor Trainer Simulator, and various Intel 8085 simulators).

Other simulators are too detailed (such as SID). They are simulating the computer hardware very closely, thus defeating the whole purpose of using a simulator in an undergraduate-level class. On the other hand, many simulators designed for educational purposes are oversimplified (MSFB). Being adequate for an introductory computer hardware course, they fail to provide substantial mechanisms for building advanced operating systems.

To summarize, existing simulators are either optimized to be used in the industry or in a hardware-oriented course, but not in a "classic" OS course, or they are intentionally hiding hardware from the upper OS layers. A "wish list" for an OS-optimized simulator includes the following requirements:

- Rich support for OS concepts.
- Little or no support for application-specific features, such as string operations and a floating-point unit (to reduce complexity and learning time).
- Reasonably detailed simulation (to make sure that the simulator could also be used in a computer architecture course).
- A collection of basic I/O devices, with a mechanism for adding more devices, if needed.
- Fast execution.
- A simple interface.
- A substantial set of development tools (such as assembler, linker, disk editor, debugger, C compiler).

Clown is an imaginary microprocessor-based computer, similar in architecture and functionality to an Intel®-based personal computer, that reasonably satisfies the listed requirements.

Availability

Clown is available free of charge for non-commercial use under the MIT license. It can be downloaded from Google Code:

```
http://code.google.com/p/clown/
```

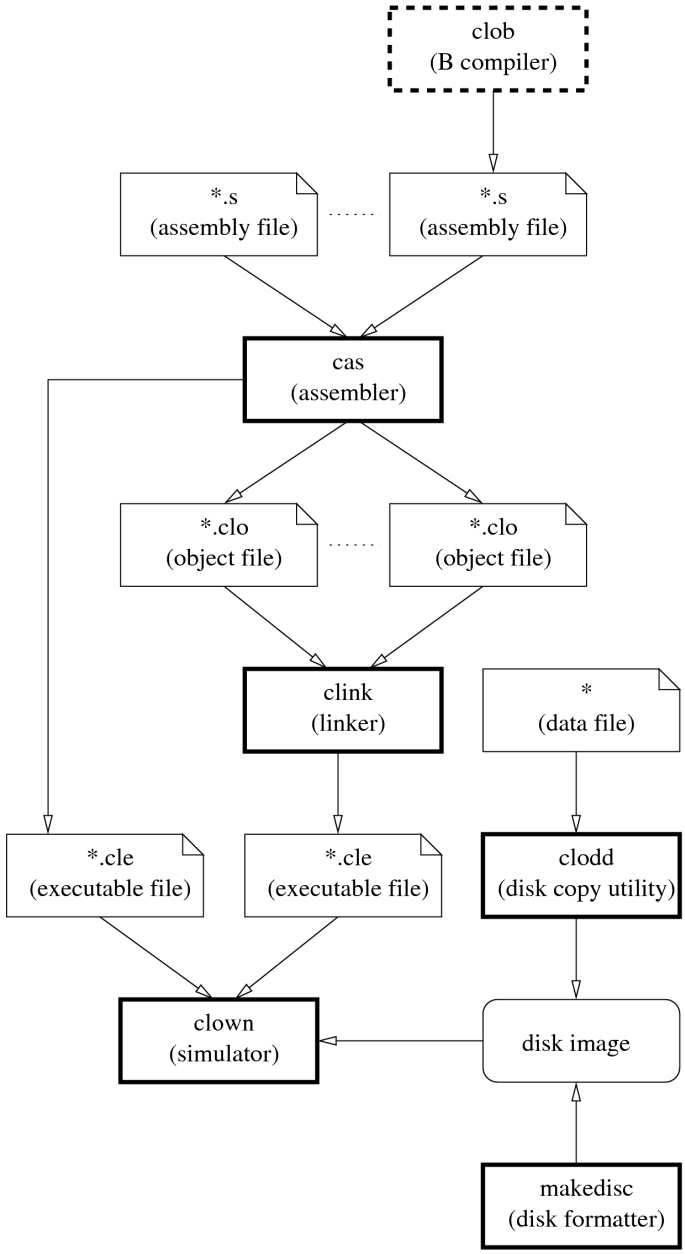
Clown is written in the ANSI C language. It is known to work on Linux and Mac OS X. To build Clown executables on Linux or Mac OS X, the following packages are required:

- flex
- bison
- an ANSI C compiler (such as gcc)
- make
- readline
- pthreads (optional)

Please e-mail your inquiries to Dmitry Zinoviev (dmitry@mcs.suffolk.edu).

8. The Clown Companion

SIMULATOR ORGANIZATION



The Clown package consists of *clown* (the simulator), *cas* (assembler), *clink* (linker), *makedisc* (disk formatting utility), and *clodd* (disk copy utility).

A typical Clown exercise consists of writing one or more assembly files, compiling and linking them, and running the simulator. A preformatted disk comes with the package. If necessary, other disks can be formatted and loaded with data and/or executable files.

Clob, the compiler of the Clown B-style high-level language, will be discussed in the later editions of this Companion.

SYSTEM ARCHITECTURE

System Architecture

The simulated system consists of the following components:

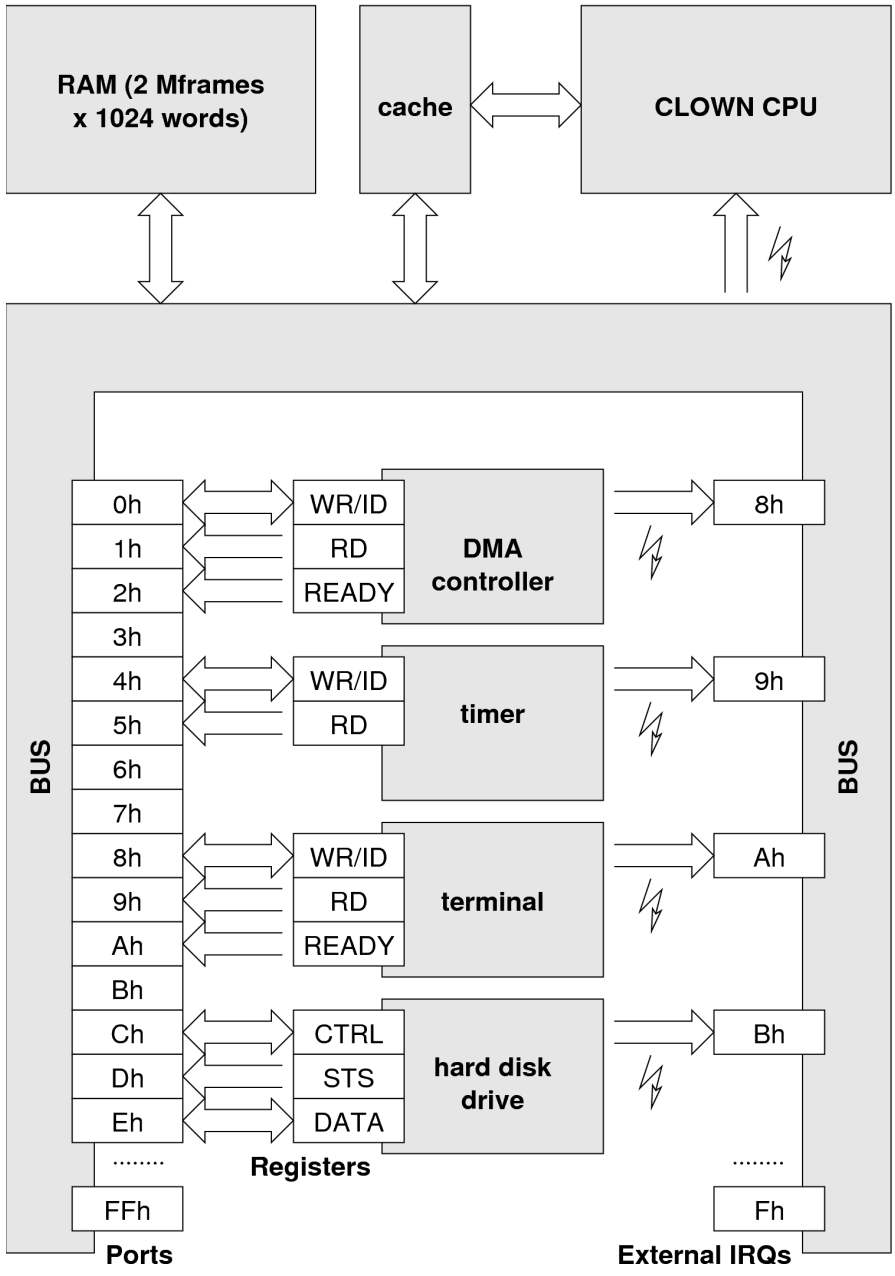
- a Clown CPU with a memory cache,
- a bank of 32-bit non-interleaved memory,
- 32-bit system and I/O buses with an implied bridge (the bridge is not simulated, and both buses are treated as one bus),
- 256 I/O ports,
- 16 interrupt request channels (IRQs),
- one direct memory access (DMA) channel,
- four basic I/O devices.

Clown uses a 32-line single-level direct-mapped write-back cache. Each cache line contains 16 words.

At most one Clown instruction is fetched, decoded, and executed at any given time. A number of memory accesses can happen during one CPU cycle. Each memory access takes a unit time.

A CPU cycle begins when fetch unit fetches the next word from the main memory or from the cache. If the next instruction is a two-word instruction, the next word is fetched as well. Then the instruction is decoded and executed. If an exception or an external interrupt occurs during the execute stage, the interrupt is recorded and delayed until the end of the stage.

The CPU has precedence over the DMA controller when accessing the memory bus. The memory bus is available for DMA transfers only if no CPU-initiated memory accesses (either instruction or data) happen during the CPU cycle.


RAM (2 Mframes
x 1024 words)
cache
CLOWN CPU
BUS
0h
1h
2h
3h
4h
5h
6h
7h
8h
9h
Ah
Bh
Ch
Dh
Eh
.....
FFh
Ports
WR/ID
RD
READY
DMA
controller
WR/ID
RD
timer
WR/ID
RD
READY
terminal
CTRL
STS
DATA
hard disk
drive
Registers
8h
9h
Ah
Bh
.....
Fh
External IRQs
BUS

CPU Architecture

The Clown CPU has sixteen 32-bit general-purpose registers (Intel: 8 GPR and two control registers, CR0 and CR3), eight 32-bit segment registers (Intel: 6 segment and 4 memory management registers), one 16-bit flag register, an instruction register, and a program counter.

The Clown MMU has a 16-entry direct-mapped Translation Lookaside Buffer (TLB; Intel 80386 has a 32-entry 4-way set-associative TLB). There is no dedicated stack pointer register, page table base register, or page fault address register. The functions of these registers are assigned to general-purpose registers, %r13 through %r15.

Clown supports only one data type: signed 32-bit words (for comparison, Intel supports at least 12 data types). This feature substantially simplifies system programming, while posing interesting challenges to compiler developers—such as type representations and conversions, and implementation of floating point arithmetics.

The Clown CPU supports both paging and segmentation. Either memory organization mechanism can be turned off (by disabling the page table or by declaring all memory to be one large segment, respectively).

Compared to Intel 30386, Clown has significantly fewer instructions (90 vs 200), which reduces the learning time. There are no data conversion instructions, decimal arithmetics, address manipulation, string and translation instructions, and high-level language support instructions.

A Clown instruction consists of either one or two words. The second word, if present (recognized by the MSB of the first word), is always the immediate operand.

The number of flags has also been minimized. There are only seven externally visible flags: four arithmetic flags (Carry, Zero, Sign, Overflow); Interrupt(s enabled) flag, a two-bit I/O Privilege Level flag, and a two-bit Current Privilege Level (CPL) flag (compared to 13 flags in Intel 30386).

The Interrupts flag can be controlled by the `sti` and `cli` instructions.

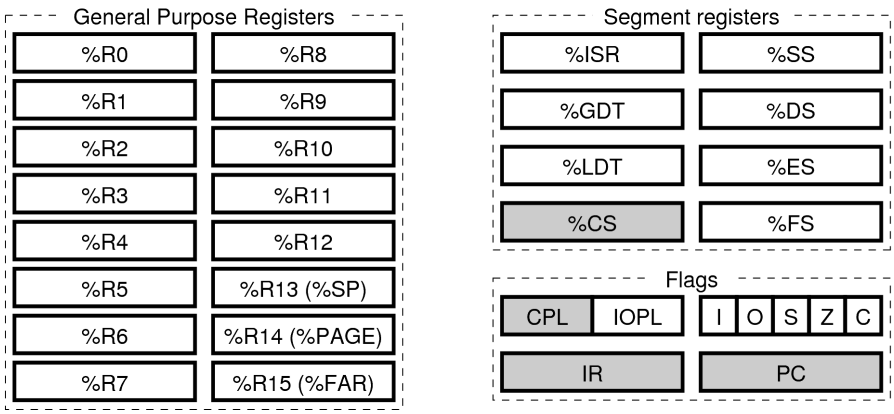
The IOPL flag can be controlled by the `chio` instruction.

The arithmetic flags are set and cleared by all arithmetic and logic instructions, as well as by the `getb` and `in` instructions. The Carry flag can be also cleared and set with `clc` and `stc` instructions. The state of all arithmetic flags can be implicitly checked with conditional control instructions.

The CPL flag can be modified or tested by pushing the entire *%flags* register on the stack, modifying and testing the value on the stack, and popping it back into the *%flags* register.

The *%pc* (program counter) register is modified implicitly by fetching the next instruction and explicitly by executing control instructions. Its value can be obtained by calling a procedure (the *%pc* will be at top of the stack).

The *%ir* (instruction register) cannot be observed or modified.


General Purpose Registers
%R0
%R1
%R2
%R3
%R4
%R5
%R6
%R7
%R8
%R9
%R10
%R11
%R12
%R13 (%SP)
%R14 (%PAGE)
%R15 (%FAR)
Segment registers
%ISR
%GDT
%LDT
%CS
%SS
%DS
%ES
%FS
Flags
CPL
IOPL
I
O
S
Z
C
IR
PC

Exceptions

Clown reports the following exceptional conditions:

#	Brief Description	Detailed Description
	Faults	
0	Invalid opcode	Instruction cannot be decoded
1	Page fault	Page not present in physical memory
2	Division by zero	Attempt to divide by zero
3	Stack overflow	Attempt to push into a full stack
4	Segmentation fault	Segment not present in physical memory
5	General protection fault	Attempt to execute a privileged instruction or otherwise violate hardware protection
6	Bus error	Bad address on the memory bus
7		Reserved
	Traps	
8	External interrupt	Timer
9	External interrupt	DMA controller
10	External interrupt	Terminal
11	External interrupt	Hard disk controller
12	External interrupt	Reserved
13	External interrupt	Reserved
14	External interrupt	Reserved
15	Software interrupt	Trap instruction

After returning from a fault handler, the value of the program counter is restored to the value it had before the execution of the instruction that caused the fault. After returning from a trap handler, the value of the program counter is restored to the value it had after the execution of the instruction that caused the trap.

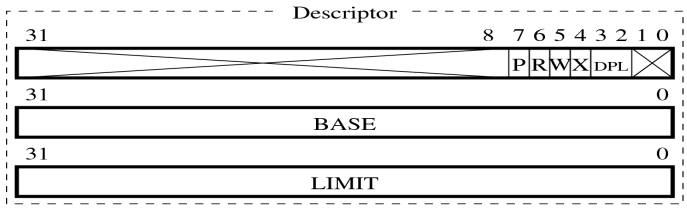
Clown supports nested interrupts. If during the execution of an interrupt handler another exception with a lower number (higher precedence) is signalled, the handler itself is interrupted, and the handler for the higher precedence interrupt is invoked.

If an exception is signalled during the execution of the handler for the same exception, a double fault exception occurs that stops the CPU. The double fault exception does not have a number and cannot be handled.

Segmentation

Clown provides full segmentation support, with the exception of doors.

A segment is a contiguous memory region (characterized by the base address and the limit) with certain access properties (readable, writable, executable) and privilege level. A segment is defined by a segment descriptor. The format of a segment descriptor is shown below:

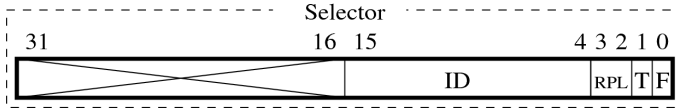


Segment descriptor flags and fields:

- BASE address,
- LIMIT,
- Present in memory,
- Readable,
- Writable,
- eXecutable,
- Descriptor PRivilege level (DPL).

The descriptors of all accessible segments are stored in two descriptor tables: Global Descriptor Table (GDT, assumed to be shared among all tasks) and Local Descriptor Table (LDT, assumed to be owned by a task). Each descriptor table is an indexed array of descriptors.

Segments are identified using segment selectors. The format of a segment selector is shown below:



A segment can be identified by more than one selector (the selectors may differ in the Requested Privilege Level).

Segment selector flags and fields:

- ID (offset in the corresponding descriptor table),
- Requested PRivilege level (RPL),
- Table selector (0 for GDT, 1 for LDT),
- This is a Fake segment descriptor (this flag is only used by the interrupt controller and equals 0 for far interrupt handlers and 1 for near interrupt handlers).

Each descriptor table is a segment itself.

Segment registers serve as caches for segment descriptors. Segment selectors

are used as cache tags. For a segment to be accessible, one of its selectors and the descriptor must be loaded in one of the segment registers, according to the table:

Segment	Segment Register
GDT, LDT	%gdt, %ldt
Code	%cs
Stack	%ss
Data	%ds (default), %es, %fs
Interrupt Vector	%isr

Logical Address Translation

To translate a logical address *A* into a linear address *L*, Clown executes the following algorithm:

1. Depending on the purpose of the translation, determine the corresponding segment register:

Purpose	Segment register	Access rights
Instruction fetch	%cs	RX
Stack operation, function call or return	%ss	R or W
Data manipulation	%ds, %es, %fs	R or W
Interrupt (including `trap`)	%isr	R
Explicit segment register manipulation	any	vary

2. Check if the “present” bit is set. If not, raise an exception.
3. Check if the address *A* is within the limits of the segment: A < LIMIT. If not, raise an exception.
4. Check the access rights (readable, writable, executable). If the rights do not match the request, raise an exception.
5. Check the permissions. A segment is accessible to a task if DPL < max(CPL, RPL). If the condition is false (see also page 24), raise a general protection fault.
6. Compute the logical address: L=A+BASE.

Handling Exceptions

The Clown interrupt vector *IV* is stored in the segment described by the %isr register. The base of the segment is initialized to 0, and the size is unlimited. The segment must be large enough to hold 16 segment selectors. Only the first 16 words of the segment are used.

To handle the exception number N, Clown inspects the value of the Nth word of the interrupt vector: `w=IV[N]`. If the least significant bit (LSB) of the word is 0, then the two least significant bytes of the word are treated as a segment selector `$seg (see page. 17)`, and the control is passed to that segment, as if a `CALL $seg` instruction were executed (a far interrupt).

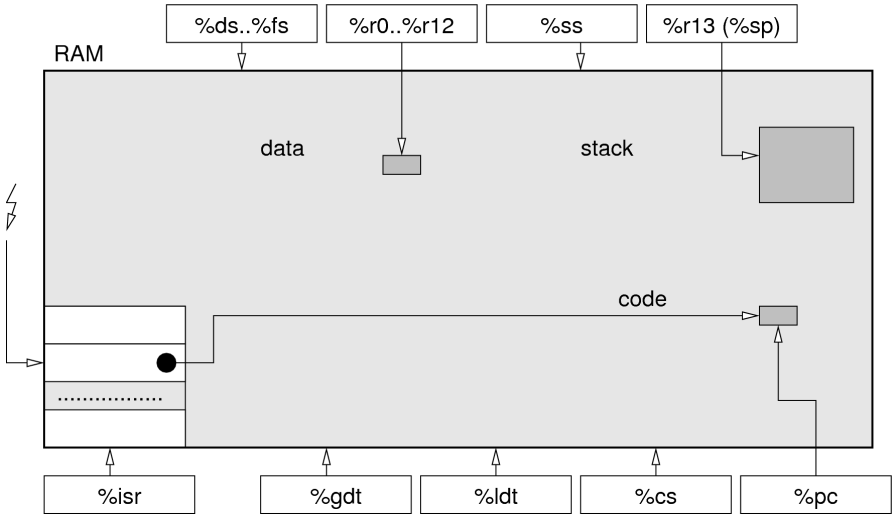
Otherwise, the next two LSBs of the word are treated as the new CPL, and the rest of the word is treated as the jump address, as if a `CALL(w & FFFFFFF8)` instruction were executed (a near interrupt). A near interrupt handler must be aligned at an 8 word boundary.

Both far and near interrupts can coexist in the interrupt vector.

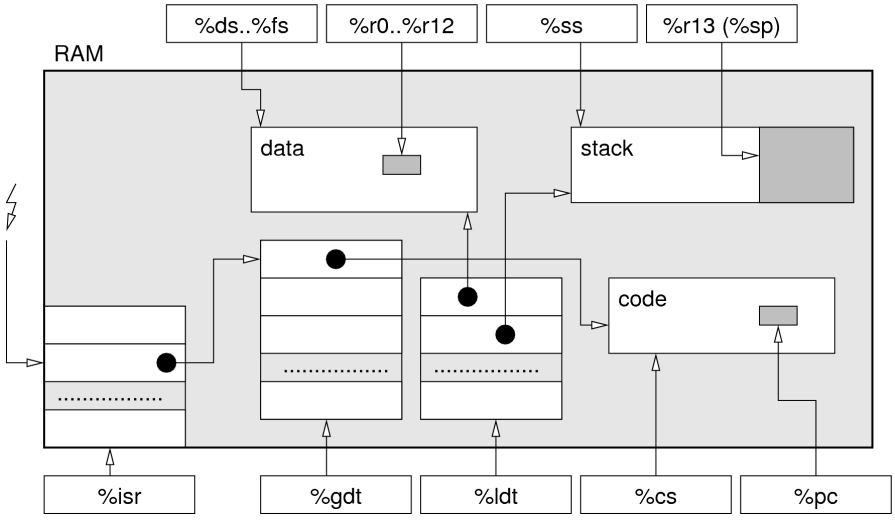
When programming interrupt handlers, programmers should use `RETFI` and `RETNI` instructions to return from far and near interrupts, respectively.

Memory Blueprint

In the flat memory model, the entire RAM is treated as one large segment that contains code, data, and stacks of all processes. All segment registers, except %ldt and %gdt, describe that segment. The interrupt handlers are linear pointers to 8-word aligned interrupt service routines:



In the segmented memory model, the RAM consists of segments that are addressed either directly through the segment registers or indirectly through the global and local descriptor tables. The interrupt handlers are segment selectors:



Paging

The Clown CPU supports hierarchical paging through page directories and page tables.

A linear address is treated as a combination of an index in the page directory `DIR`, an index in the page table `PAGE`, and the offset in the selected page `OFFSET:`

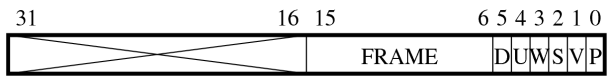
A Clown page consists of 1024 words.

A page table occupies one page.

A page directory occupies one page.

Each process can have no more than one page directory, no more than 1024 page tables, and no more than 1024×1024 pages.

Pages, page tables, and page directories are described using page descriptors. The format of a page descriptor is shown below:



- `FRAME` is the frame number in the Clown RAM where the page is currently stored.
- `D` is the "dirty" bit which is set if the contents of the page have been altered.
- `U` is the "used" bit which is set if the page has been accessed (either for reading or for writing).
- `W` is the "write" bit which is set if the page is writeable.
- `S` is the "superuser" bit. If the bit is clear, only the innermost ring tasks (`CPL=0`) are allowed to access the page.
- `V` is the "valid" bit which is set if the descriptor indeed describes a page (is a valid descriptor).
- `P` is the "present" bit which is set if the page is present in the RAM (not paged out).

A page table is an array of 1024 page descriptors of the pages registered in the table.

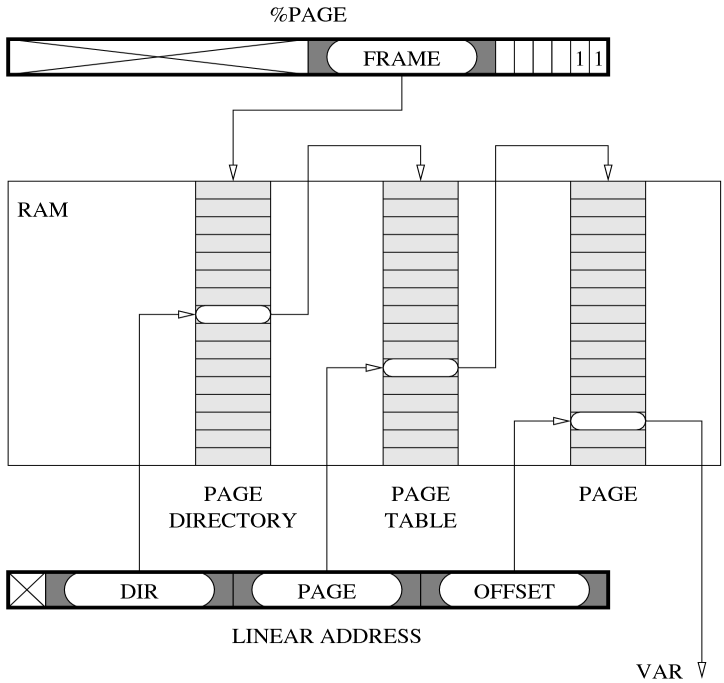
A directory table is an array of 1024 page descriptors of the page tables.

The page descriptor of the directory table is stored in the %page register. Only the innermost ring tasks can modify the contents of this register. The paging is on only if the "present" bit of the %page register is set.

Linear Address Translation

Let a⊕b ≡ (a<<N) + b and N=10 be the number of bits in the offset. To translate a linear address *L* into a physical address *P*, Clown executes the following algorithm:

1. Check if the “present” bit of the %page register is set. If not, the physical address equals the linear address: P=L.
2. Otherwise, locate the page table descriptor: ptd=RAM[%page.frame⊕L.dir].
3. Locate the page descriptor: pd=RAM[ptd.frame⊕L.page]. If, according to the page descriptor, the page is not present in the memory (pd.present==0), store the linear address *L* in the %far register and raise a page fault.
4. Otherwise, check the access rights. If CPL>pd.priv or the process attempts to write into a read-only page (pd.write==0), raise a general protection fault.
5. Mark the page “used” (pd.used=1).
6. If the purpose of the translation is to alter the contents of the frame, mark the page “dirty” (pd.dirty=1).
7. Compute the physical address: P=RAM[pd.frame⊕L.offset].



Protection Mechanisms

Clown provides hardware memory protection in the form of several access monitoring mechanisms. Violations of memory protection policies cause a General Protection Fault. Clown distinguishes the following five forms of memory protection violations:

%cs access or usage violation	Attempt to load a selector into %cs. %cs can be modified only implicitly, through `jmp` and `call` instructions.
CPL violation	Current privilege level violation: attempt by an outer-ring task to modify %isr, %gdt, %ldt, %page or load a segment with DPL<max(CPL, RPL) into any segment register; attempt by an outer-ring task to execute a privileged instruction (`cli`, `sti`, `chio`, `in`, `out`, `hlt`, `stop`).
Segmentation violation	Attempt to access a value beyond the segment boundary (including stack entries and entries in the local and global descriptor tables); attempt to load a non-readable selector into %isr, %gdt or %ldt, or a non-executable selector into %cs, or a non-writable selector into %ss, or a non-readable and non-writable selector into %ds, %es or %fs; attempt to write into a non-writable segment.
Invalid segment register	Attempt to load a selector into a nonexistent register.
ISR segment too small	Attempt to initialize %isr with a selector of a segment that is shorter than 16 words.

SIMULATOR

Interface Summary

clown is the Clown simulator. It can be run either in the batch or in the interactive mode. It supports the following command line options:

- -h, --help – show the help message and exit
- -q, --silent – work silently (do not show any diagnostic messages)
- -v, --version – print simulator version number and exit
- -r, --run – run simulator in the batch mode
- --cpl CPL – set the initial CPL of the Clown CPU to *CPL*
- -b, --bin FILE ADDR – load an executable image from file *FILE* into the RAM at address *ADDR*. This option can be repeated as needed to load more than one image file.

If *clown* is given any other command line argument, either not beginning with a dash or preceded by two dashes --, that argument is treated as the name of an image file, which will be loaded starting at the address 0.

The image from file *test.cle* can be loaded at the address 0 using either of the following command line expressions:

```
shell> clown test.cle
```

or

```
shell> clown --bin test.cle 0
```

The images from two files *test.cle* and *qwest.cle* can be loaded at addresses 0 and 2048, respectively, using either of the following command line expressions:

```
shell> clown --bin qwest.cle 2048 test.cle
```

or

```
shell> clown --bin test.cle 0 --bin qwest.cle 2048
```

The images will be loaded in the order they appear on the command line (with or without the *--bin* switch). This should normally not be a concern, unless the images overlap in the RAM.

If an executable file contains more than one segment, all segments will be loaded into the RAM sequentially. Additionally, *clown* creates a descriptor table and stores it in the main memory immediately after the last loaded segment. The selector of the newly created table is loaded into the %ldt. The selector of the first code segment is loaded into the %cs.

clown does not initialize the %gdt. If necessary, the %gdt register can be initialized by an application.

If more than one multi-segment file is loaded into the RAM, then only the selector of the first descriptor table is loaded into the %ldt.

Both %ldt and %cs are correctly reinitialized at reset.

Batch Mode

In the batch mode, the simulator is automatically reset, and the simulation is started. The simulation can be interrupted at any moment by pressing ^C (Ctrl+C) or sending signal SIGUSR1 to the *clown* process. If interrupted, the simulator enters the interactive mode. Otherwise, when the simulated program terminates, the simulator exits. Use the *-r* command line switch to enter the batch mode.

Interactive Mode

In the interactive mode (default), the simulator is automatically reset, and then displays the prompt `CLOWN>` and waits for the user's input. The following interactive commands are currently implemented (optional parameters and their default values are shown in {curly braces}):

Commands	**Description**
h, help	Display a concise help message.
l, load “file” {addr=0}	Load an executable image from *file* (in quotes) into the RAM, starting at address *addr*.
reset	Reset the simulator (including all flags and registers, but not the RAM) to its default state.
r, run {count=0}	If *count* is zero, then run the simulator until it stops normaly (with the STOP instruction) or abnormally (with an unhandled exception). Otherwise, execute *count* instructions. The simulation can be interrupted by pressing ^C (Ctrl+C) and later resumed with another identical *run* command. At the end of the simulation, the fetch unit is disabled and has to be enabled with the *reset* command.
s, step	Execute one instruction
%xx {format=\d}	Display the value(s) of register(s) *%xx*. *%xx* can be a general-purpose register designator *%r0* through *%r15*, *%sp* (stack pointer), *%far* (fault address register), *%page* (page table register), *%pc* (program counter), *%ir* (instruction register), *%flags* or *%fl (*flag register), or *%all* (all 16 general-purpose registers). The value(s) are displayed according to the *format*, which can be \d (decimal), \b (binary), \o (octal), \h (hexadecimal), \c (ASCII character), \p (page descriptor) or \a (linear address).
%xx=value	Change the value(s) of register(s) *%xx* (see the description of the *%xx* command above). The *value* is a number written in decimal, binary (0011b), octal (756o) or hexadecimal (0x67F or 67Fh) notation. Only decimal numbers can be negative.
[%xx] {'count=1} {format=\d}	Display the contents of *count* RAM locations pointed by the register(s) *%xx*, according to the *format* (see the description of the *%xx* command above). The pointer is treated as a physical address.

Commands	**Description**
[%xx]=value	Set the value in the RAM locations pointed by the register(s) *%xx* to *value.*The pointer is treated as a physical address.
[addr] {'count=1} {format=\d}	Display the contents of *count* RAM locations, starting at address *addr* (which is a number in any acceptable notation, see above). The pointer is treated as a physical address.
[addr]=value	Set the value of the RAM location at address *addr* (which is a number in any acceptable notation) to *value*. The pointer is treated as a physical address.
number {format=\d}	Display the *number* in a different *format*.
q, quit, exit	Exit the simulator.

30. The Clown Companion

ASSEMBLER

Interface Summary

cas is the Clown assembler. It converts source files written in the Clown assembly language into binary files that contain executable Clown instructions, symbol tables, and debug information. An output binary file can be immediately executable (a CLE file) if it contains no unresolved symbols and segments and no relocation information. Otherwise, the output file is an object file (a CLO file) and must be combined (linked) with other CLO files.

cas supports the following command line options:

- -h, --help – display the help message and exit
- -P, --nocpp – do not preprocess the source with the C preprocessor, *cpp*
- --pass-to-cpp "OPTIONS" – pass *OPTIONS* to the C preprocessor, *cpp*; if the shell variable *$CLOWN* is set, then option `-I($CLOWN/include/)` is passed to the *cpp* implicitly
- -b, --cle – generate a non-relocatable executable CLE module (default)
- -s, --clo – generate a relocatable CLO module (object file)
- -e ADDR, --entry ADDR – set the start address to *ADDR* (default 0)
- -o OFILE – set the output file name to *OFILE* (the default name is *IFILE*.cle for CLE modules and *IFILE*.clo for CLO modules, where *IFILE* is the name of the input assembly file, less the suffix .s, if any)
- -v, --version – print assembler version number and exit
- -l, --listing – produce code listing
- -ng, --nodebug – do not include file and line information for debugging
- -V – print assembler version number and continue
- -- – treat the next argument as a file name, even if it looks like yet another command line option.

Any other command line argument that either does not begin with a dash or is preceded by two dashes --, is treated as the name of the assembly file.

Program Listing

When invoked with the `-l` command line option, *cas* generates both the output binary file and the program listing. The listing contains the human-readable numerical value of each generated code and data word, as well as a list of all segments and symbols.

For each generated instruction or data word, the listing contains its offset in the binary file and the hexadecimal value. In a CLO module, if there is no offset in the first column, then the corresponding word is an escape symbol and will not be included in the final CLE module.

For the printable ASCII data fields, the corresponding ASCII characters are also displayed. For the data fields that numerically equal ASCII control and space characters, the corresponding standard ASCII names are displayed, such as STX, \b, etc.

The following partial listing has two segments, five instruction definitions, three auxiliary lines, and six data definitions:

```
$mycode
0x0000000D: 0xC9000000
            0xFF000008
            0x00000000 NUL
            0x00000003 ETX
0x0000000E: 0x00000002 STX
0x0000000F: 0xF0D00000
0x00000010: 0x00000001 SOH
0x00000011: 0x5E000000
$mydata
0x00000000: 0x00000048 'H'
0x00000001: 0x00000065 'e'
0x00000002: 0x0000006C 'l'
0x00000003: 0x0000006C 'l'
0x00000004: 0x0000006F 'o'
0x00000005: 0x00000020 SPACE
...
```

For each segment in the module, the listing contains the number of the segment (in a CLE file, that number corresponds to the index of the segment in the Local Descriptor Table), the name of the segment, whether the segment is code (**C**), data (**D**) or both (no letter), whether the segment is defined or not (**DEF** or **UNDEF**), and the size of the segment in words:

```
Segments:
--------------------------
  0. $code*              UNDEF
  1. $mycode           C DEF size=  8
  2. $mydata           D DEF size=  12
```

`$code*` is the default code segment that is always present, but not always defined.

For each symbol in the module, the listing contains the name of the symbol, whether the symbol is global or not (**G** or no **G**), whether the symbol is defined or not (**DEF** or **UNDEF**), and the address of the symbol definition (both the segment and the offset). The following listing describes three symbols: global defined

`main`, local defined `msg`, and undefined `gets` and `puts`. The defined symbols are in the corresponding code and data segments:

```
Symbols:

-------------------------------
main              G DEF $mycode:(0x00000000)
puts                UNDEF
gets                UNDEF
msg                 DEF $mydata:(0x00000000)
```

Source Debugging

Unless invoked with `-ng` or `--nodebug` option, *cas* generates line and file debugging information for every Clown instruction. This information is stored in the output CLO or CLE file. If the output file with the debugging information is older than any source file, *clown* notifies that the debugging information is stale:

```
--> Source file stdio.s is
--> newer than default.cle.
```

If the debugging information is available, then *clown* reports the source reference of every known memory location:

```
CLOWN> %pc
           %PC = +0000000000 [stdio.s:6]
CLOWN> [8]
           +1526728704 [stdio.s:9]
CLOWN> %sp
           %R13 = +0001048575  ; no information about the stack
```

The debugging information can be removed from a binary file by invoking the linker with `-ng` or `--nodebug` option:

```
CLOWN> clink --nodebug default.cle -o default-nodebug.cle
```

The debugging information is not available if paging is in use.

36. The Clown Companion

ASSEMBLY LANGUAGE

Program Structure

An assembly program consists of optional preprocessor directives, optional data declarations, and code. Code and data declarations can be optionally grouped into segments.

cas uses the C preprocessor, *cpp*, to pre-process source files **before** assembling them. See the documentation on *cpp* for more details on what preprocessor directives are available and how to use them. The most important directives are:

- #include
- #define
- #ifdef
- #ifndef
- #else
- #endif

The *--pass-to-cpp* command line option can be used to pass parameters (such as *-D*) to *cpp*, as in the following example:

```
shell> cas --pass-to-cpp "-D %ax=%r0" ...
```

In this example, all references to a fictitious register `%ax` are replaced with `%r0`.

The *--nocpp* option disables preprocessing.

The configuration of the Clown I/O devices is stored in the file *clown/config.h*. For consistency, include this file in any program that uses I/O devices.

Comments

cas supports C-style (/**/), C++-style (//), and assembly-style (;) comments. The C-style comments cannot be nested. The C++-style and assembly-style comments extend from the double forward slash or semicolon to the end of the line.

Expressions

In the spirit of Clown having only one data type, *cas* does not support floating point numbers. Only signed 4-byte integer numbers are supported.

A Clown expression is either an integer number, a symbol, or an arithmetic expression.

Integer numbers can be written in decimal, prefix and postfix octal or hexadecimal, and postfix binary notation:

- -45676 ; decimal
- +45676 ; decimal
- 0xB26C ; prefix hexadecimal
- B26Ch ; postfix hexadecimal
- 0131154 ; prefix octal
- 131154o ; postfix octal
- 1011001001101100b ; postfix binary

Small numbers can be entered as ASCII characters (in single quotes). *cas* supports some ASCII special characters: '\n', \t', '\r', '\b', '\f', and '\\'.

ASCII strings (in double quotes) can be used to initialize arrays of small numbers in data definitions. Remember that, despite the appearance, each character in a string is treated as a 4-byte number. Storing ASCII strings in a compact (one byte per character) form is the responsibility of the programmer or compiler, not the assembler.

The following operators can be used to form integer arithmetic expressions: + (add), - (subtract), * (multiply), / (divide), | (bitwise or), ~ (bitwise invert), & (bitwise and), ^ (bitwise exclusive or), ! (logical not), % (remainder). They have the same meaning, precedence, and associativity as in the C language. Parentheses () change the order of evaluation.

All expressions are evaluated during the assembly stage. No code is generated to evaluate constant expressions.

Symbols

Symbols identify memory locations. Symbol names are case-sensitive and follow the rules for identifiers in the C language. A symbol is defined by its identifier followed by a colon, as in the following code:

```
again:
          cmp %r1, 0
          jz again
```

Here, `again` is defined to be a symbol referring to the location of the `cmp` command. It is used by the `jz` command as the target of a near conditional jump.

A symbol evaluates to the address of the RAM location associated with the symbol. A symbol in square brackets evaluates to the content of the location. Compare:

```
          mov %r0, var   ; move the address of var into %r0
          mov %r0, [var] ; move the content of var into %r0
```

Symbols can be used in most numerical arithmetic expressions, e.g.:

```
          mov %r0, [var+1] ; move the content of the next word
                           ; after var into %r0
my_string:
          .string “Hello, world!\n”
end:
          mov %r0, end-my_string ; the size of the string
```

The scope of a symbol is the smallest of the segment or module where it is defined, unless the symbol is declared global. The scope of a global symbol is unlimited.

One memory location can be identified by more than one symbol:

```
start:
interrupt_vector:
          .word[16]
```

Modifiers

`.global` modifier makes a symbol global (visible in all modules):

```
.global _init:
           ...
```

`.align8` modifier mandates that the symbol immediately following the modifier is aligned at an 8-word boundary (the address of the symbol is evenly divisible by eight):

```
.align8 timerInterrupt:
           ...
```

Interrupt service routines must be aligned this way if used in flat (one-segment) mode.

`.page` modifier mandates that the symbol immediately following the modifier is aligned at a page boundary (the address of the symbol is evenly divisible by 1024):

```
.page _pageTable:
           ...
```

Modifiers `.align8` and `.page` can be combined with `.global`:

```
.align8 .global divideByZero:
           ...
.global .align8 segFault:
           ...
.global .page Segments:
           ...
.page .global _processes:
           ...
```

Neither `.page` nor `.align8` can be used in a multi-segment program.

Data Declarations

Data declarations allocate static memory for data and optionally fill it with data.

`.string` allocates memory for an array of small numbers represented as ASCII characters. One word of memory is allocated for each character, and the numerical value of the character is stored into the word. Memory is also allocated for the trailing zero character.

`.word` allocates memory for a single number or for an array of numbers, and optionally initializes this memory. The number of initializers must be equal to or less than the declared size of the array. The array locations that are not initialized explicitly, are filled with zeros. The initializers can be optionally enclosed in curly braces.

Some examples of data declarations (the curly braces are optional):

```
msg:
          .string “Clowns are in town!” ; a “string”
coeffs:
          .word[8] ; array of 8 numbers with initializers
          { C', 'l', 'o', 'w', 'n', 's', 255, 0x676FF }
```

Expressions can be used to initialize arrays and single variables (the curly braces are optional):

```
mix:
          .word[2] { (coeffs-msg)*2, (mix+7)/12 }
```

A segment name can be used as an initializer. It evaluates to a segment selector:

```
myisr:
          .word[16] { 0, 0, $divby0 }
```

Segment Declarations

A segment declaration consists of a segment qualifier and a segment name (segment selector). A segment qualifier is one of the `.data`, `.code` or `.const`. A segment name is a dollar sign "`$`" immediately followed by a C-style identifier. Segment names are case-sensitive. The scope of a segment name is unlimited.

The following code fragment defines two segments: code segment `$mycode` and data segment `$mystack`:

```
.code $mycode
          mov %ss, $mystack
          mov %sp, endstack
          .....

.data $mystack
mystack:
          .word[32]
endstack:
          .word
```

A segment declaration defines a segment that begins at the declaration and ends at the next segment declaration or at the end of the file, whatever comes first.

If no segments are defined in a module, all the module's contents are implicitly placed into the default segment `$code*`.

If segments are used, it is the responsibility of the programmer or the compiler (not the assembler) to declare, allocate, and initialize the stack segment and at least one code segment.

Registers

Valid general-purpose register names are %r0...%r15, %far (Fault Address Register; alias for %r15), %page (Page Table Register; alias for %r14), and %sp (Stack Pointer; alias for %r13).

Valid segment register names are %gdt (Global Descriptor Table), %ldt (Local Descriptor Table), %isr (Interrupt Service Routine Table), %cs (Code Base), %ss (Stack Base), %ds (Data Base 0—the default data segment), %es (alternative Data Base 1), and %fs (alternative Data Base 2).

Register names are case-insensitive.

A register name in square brackets stands for the contents of the RAM location pointed by the address stored in the register:

```
move  %r0 , 57      ; load 57 into %r0
move [%r0], 75      ; store 75 into the RAM location
                    ; pointed by %r0 (in location #57)
```

An attempt to store a value into the %page register results in a general protection fault, unless the task belongs to the innermost ring (CPL = 0).

46. The Clown Companion

ASSEMBLY COMMANDS

There are 55 Clown assembly commands and 93 modifications thereof. The command names are case-insensitive.

Arithmetic

Instructions generated by all arithmetic commands update all arithmetic flags (sign, zero, carry, and overflow). Instructions generated by the `div` command cause a division-by-zero exception if the divisor is zero.

`add %rx, %ry`	`%rx = %rx + %ry`
`add %rx, expr`	`%rx = %rx + expr`
`cmp %rx, %xy`	`%rx - %ry` (result is not stored, but flags are updated)
`cmp %rx, expr`	`%rx - expr`
`dec %rx`	`%rx = %rx - 1`
`div %rx, %ry`	`%rx = %rx / %ry`
`div %rx, expr`	`%rx = %rx / expr`
`div expr, %rx`	`%rx = expr / %rx`
`inc %rx`	`%rx = %rx + 1`
`mul %rx, %ry`	`%rx = %rx * %ry`
`mul %rx, expr`	`%rx = %rx * expr`
`neg %rx`	`%rx = -%rx`
`sal %rx, %ry`	`%rx = %rx << %ry` (arithmetic shift)
`sal %rx, expr`	`%rx = %rx << expr` (arithmetic shift)
`sar %rx, %ry`	`%rx = %rx >> %ry` (arithmetic shift)
`sar %rx, expr`	`%rx = %rx >> expr` (arithmetic shift)
`sub %rx, %ry`	`%rx = %rx - %ry`
`sub %rx, expr`	`%rx = %rx – expr`

The following code fragment computes the value of the fractional arithmetic expression $(a+b)/(a-1/b)$. We assume that the values of a and b are in the registers %r0 and %r1, respectively:

```
mov %r2, %r0 ; make a copy of a
add %r2, %r1 ; a+b
div 1, %r1   ; 1/b
sub %r0, %r1 ; a-1/b
div %r2, %r0 ; the answer is in %r2
```

Do not forget that Clown does not support floating point division.

Logical

Instructions generated by all logical commands update all arithmetic flags (sign, zero, carry, and overflow).

`and %rx, %ry`	`%rx = %rx & %ry`
`and %rx, expr`	`%rx = %rx & expr`
`not %rx`	`%rx = ~%rx`
`or %rx, %ry`	`%rx = %rx | %ry`
`or %rx, expr`	`%rx = %rx | expr`
`rol %rx, %ry`	`%rx = %rx << %ry` (logical shift)
`rol %rx, expr`	`%rx = %rx << expr` (logical shift)
`ror %rx, %ry`	`%rx = %rx >> %ry` (logical shift)
`ror %rx, expr`	`%rx = %rx >> expr` (logical shift)
`tst %rx, %xy`	`%rx & %ry` (result is not stored, but flags are updated)
`tst %rx, expr`	`%rx & expr`
`xor %rx, %ry`	`%rx = %rx ^ %ry`
`xor %rx, expr`	`%rx = %rx ^ expr`

The following code fragment computes the value of the logical expression (*a*&*b*)|(*a*^!*b*). We assume that the values of *a* and *b* are in the registers %r0 and %r1, respectively:

```
mov %r2, %r0 ; make a copy of a
and %r2, %r1 ; a&b
not %r1      ; !b
xor %r0, %r1 ; a^!b
or  %r2, %r0 ; the answer is in %r2
```

`xor`'ing a register with itself is an efficient way of zeroing it out (`xor` does not have an immediate part and occupies only one word, while `mov` occupies two words):

```
xor %r0, %r0
```

Data Manipulation

For all memory-referencing commands (`xchg` and some variants of `mov`), the memory address must be defined in the code segment whose descriptor is loaded in %ds, unless another segment is explicitly used.

`clrb %rx, %ry`	clear the `%ry`'th bit of `%rx: %rx[%ry]=0`
`clrb %rx, expr`	clear the *`expr`*'th bit of `%rx: %rx[`*`expr`*`]=0`
`getb %rx, %ry`	set the `Carry` flag if the `%ry`'th bit of `%rx` is set
`getb %rx, expr`	set the `Carry` flag if the *`expr`*'th bit of `%rx` is set
`mov %rx, [var]`	load the value of variable *`var`* into `%rx: %rx=`*`var`*
`mov %rx, var`	load the address of variable *`var`* into `%rx: %rx=&`*`var`*
`mov %rx, expr`	load the value of *`expr`* into `%rx: %rx=`*`expr`*
`mov [var], %rx`	store the value of `%rx` into variable *`var`*`: `*`var`*`=%rx`
`mov %rx, %ry`	copy `%ry` into `%rx:%rx=%ry; %ry` can be a segment register
`mov %rx, [%ry]`	load the value from the memory location defined by `%ry,` into `%rx: %rx=RAM[%ry]`
`mov [%rx], %ry`	store `%ry` into the memory location defined by `%rx: RAM[%rx]=%ry`
`mov %segr, %rx`	load `%rx` into segment register `%segr`; the value must be a valid segment selector; only the innermost ring tasks (CPL = 0) may modify registers `%gdt`, `%ldt`, and `%isr`; descriptors may not be explicitly stored in `%cs`
`mov %rx, %segr`	load the segment selector from `%segrx` into general purpose register `%rx`
`mov %segr, segd`	load segment descriptor *`segd`* into segment register `%segr`; only the innermost ring tasks (CPL = 0) may modify registers `%gdt`, `%ldt`, and `%isr`; descriptors may not be explicitly stored in `%cs`
`mov %rx, [%segr:var]`	load the value of variable *`var`* in segment `%`*`segr`* into `%rx: %rx=`*`var`*
`mov [%segr:var], %rx`	store `%rx` into variable *`var`* in segment `%`*`segr`*`: `*`var`*`=%rx`
`mov %rx, %ry(expr)`	load the *`expr`*'th byte of `%ry` into `%rx: %rx=%ry[`*`expr`*`]`
`mov %rx(expr), %ry`	replace the *`expr`*'th byte of `%rx` with the least significant byte of `%ry: %rx[`*`expr`*`]=%ry & 0xFF`

`setb %rx, %ry`	set the `%ry`'th bit of `%rx: %rx[%ry]=1`
`setb %rx, expr`	set the *`expr`*'th bit of `%rx: %rx[`*`expr`*`]=1`
`xchg %rx, [var]`	atomically exchange the values of `%rx` and variable *`var`*

The following code fragment loads the variable *dummy*, extracts the least significant byte of that variable, and atomically exchanges it with the content of the memory location that immediately follows *dummy*:

```
        mov %r0, [dummy]
        mov %r1, %r0(3)
        xchg %r1, [dummy+1]
        ...
dummy:
        .word 487634564
```

There must be a control command somewhere before the last line of this example that prevents Clown from executing the data declaration, as if it were an opcode. Otherwise, an Invalid Opcode exception will be raised.

Stack Manipulation

`peek %rx, offset`	load the value that is *`offset`* words deep in the stack (the top of the stack is 0 words deep) into `%rx`: `%rx=RAM[%sp+1+`*`offset`*`]`; the stack pointer is not affected by this command
`poke %rx, offset`	store `%rx` in the stack at the depth of *`offset`* words (the top of the stack is 0 words deep): `RAM[%sp+1+`*`offset`*`]=%rx`; the stack pointer is not affected by this command
`pop %rx`	pop a value from the stack into `%rx: %rx=RAM[++%sp]`
`push %rx`	push `%rx` into the stack: `RAM[%sp--]=%rx;` this command can cause a stack overflow exception
`push expr`	push *`expr`* into the stack: `RAM[%sp--]=%expr`; this command can cause a stack overflow exception

In the following example, the return address of a function is on the top of the stack, followed by two parameters, *a* and *b*. The fragment pushes the registers %r0 and %r1 onto the stack, then extracts *a* and *b* into %r0 and %r1, adds them up, stores the sum on the stack in the place of *a*, and restores the original values of %r0 and %r1:

```
push %r0
push %r1
peek %r0, 3  ; extract a
peek %r1, 4  ; extract b
add %r0, %r1
poke %r0, 3  ; save a+b
pop %r1      ; restore in the opposite order!
pop %r0
```

Unconditional Control

`call symbol`	call the code at *`symbol`* that must be defined in the same segment; `%pc` is pushed onto the stack
`call expr`	call the code at offset *`expr`*; the call target must be in the same segment; `%pc` is pushed onto the stack
`call [%rx]`	call the code at offset stored in `%rx`; the call target must be in the same segment; `%pc` is pushed onto the stack
`call $seg:symbol`	call the code at *`symbol`* defined in segment *`$seg`*; `%cs` and `%pc` are pushed onto the stack
`call $seg`	call the code at the beginning of segment *`$seg`*; `%cs` and `%pc` are pushed onto the stack
`jmp symbol`	jump to *`symbol`* that must be defined in the same segment
`jmp expr`	jump to offset *`expr`* that must be in the same segment
`jmp $seg:symbol`	jump to *`symbol`* defined in segment *`$seg`*
`jmp $seg`	jump to the beginning of segment *`$seg`*
`nop`	do nothing: skip a cycle
`retf`	return from a far procedure call (restore `%pc` and `%cs`)
`retfi`	return from a far interrupt call (restore `%flags`, `%pc`, and `%cs`)
`retn`	return from a near procedure call (restore `%pc`)
`retni`	return from a near interrupt call (restore `%flags` and `%pc`)
`trap`	trap the OS (raise interrupt #15)

The following code fragment pushes two parameters, *a* and b that have been stored in %r0 and %r1, respectively, for the near procedure *foo*; then it calls the procedure, saves the return value in %r0 and restores the stack:

```
push %r1      ; push b first!
push %r0      ; push a next!
call foo
pop %r0       ; the return value
inc %sp       ; two pushes vs one pop
```

The following code allows the Clown to have the interrupt vector at the beginning of the physical address space:

```
iv:
          jmp code     ; this instruction is executed once
          .word[15]    ; it can be later overwritten
code:
          ...          ; actual code
```

Conditional Control

The targets of all conditional control commands must be defined not further than 127 words away from the command itself.

`jc symbol`	jump to *`symbol`* if `Carry` flag is set
`jo symbol`	jump to *`symbol`* if `Overflow` flag is set
`js symbol`	jump to *`symbol`* if `Sign` flag is set
`jz symbol`	jump to *`symbol`* if `Zero` flag is set
`jnc symbol`	jump to *`symbol`* if `Carry` flag is clear
`jno symbol`	jump to *`symbol`* if `Overflow` flag is clear
`jns symbol`	jump to *`symbol`* if `Sign` flag is clear
`jnz symbol`	jump to *`symbol`* if `Zero` flag is clear

If a longer jump is required, use a combination of a short conditional jump and a long unconditional jump:

```
          ...                 ; arithmetic or logical operation
          jc carry_label
          jmp if_no_carry
carry_label:
          ...                 ; the code for the “carry” case
          jmp end_carry
if_no_carry:
          ...                 ; the code for the “no carry” case
end_carry:
          ...                 ; more code
```

System Control

Only the innermost ring tasks (CPL = 0) can execute these instructions without causing a General Protection Fault.

`chio expr`	change the `IOPL` of the current task to *`expr`*
`cli`	disable interrupts by clearing the `Interrupt` flag
`hlt`	suspend Clown and wait for external interrupts
`sti`	enable interrupts by setting the `Interrupt` flag
`stop`	stop Clown

Flag Control

`clc`	clear the `Carry` flag
`popf`	pop a value from the stack into the flag register: `%flags=RAM[++%sp]`; only the innermost ring tasks (CPL = 0) can modify flags CPL (current privilege level), IOPL (I/O privilege level), and I (interrupt)
`pushf`	push the flag register into the stack: `RAM[%sp--] =%flags`; this command can cause the stack overflow exception
`stc`	set the `Carry` flag

The following code fragment changes the CPL of the current process to 3 by manipulating the flags through the stack:

```
pushf         ; push the flags into the stack
pop %r0       ; pop the flags to %r0
or %r0,1100000b ; modify the flags
push %r0      ; push the flags from %r0
popf          ; pop the flags
```

Input and Output

Only the innermost ring tasks (CPL=0 or IOPL=0) can execute these instructions without causing a general protection fault. The instructions generated by the `in` command update all arithmetic flags (sign, zero, carry, and overflow).

`in %rx, port`	read a value from *`port`* and store it in `%rx`
`out %rx, port`	write `%rx` into the *`port`*
`out expr, port`	write *`expr`* into the *`port`*

The I/O *port* number is a constant (or any expression that can be evaluated to a constant at assembly time) preceded by a question mark, such as `?5`, `?IO_BASE` or `?(2*(IO_BASE+1))`.

58. The Clown Companion

PERIPHERAL DEVICES

Timer

The Clown timer alarms the CPU by raising external interrupts at certain moments of time. The timer has two I/O registers, R0 and R1, and two internal registers, counter and interval.

The Clown timer is an interval timer. If the value of the internal interval register is N, where N is a positive number greater than 1, then the timer raises an interrupt on IRQ8 every N cycles. If the value of the internal interval counter is 1 or 0, then upon expiration the timer raises an interrupt on IRQ8 and stops.

Register R0 is an R/W register. Reading from R0 returns `0x7F74696D` (the signature of the timer).

Writing 0 into R0 stops and resets the timer (both the internal counter and the internal interval register).

Writing 1 into R0 switches the timer into the “ready” state.

Writing a negative number into R0 sets the internal interval register to the absolute value of the number. If the timer is in the “ready” state, writing any positive value into R0 triggers the countdown, and the value written (less 1) becomes the initial value of the internal timer counter (the timer expires when the counter hits 0). If the timer is in the “idle” state, writing any value into R0 (except 0 or 1) does not cause any action.

Register R1 is an R register (read-only). Reading from R1 returns the current value of the internal counter (the time to expire). If the value is 0, the timer is in the “idle” state.

Programming Timer

The I/O base of the timer is at the port 0x04. The following code fragment causes the timer to expire periodically every 10,000 CPU cycles after the initial 1,000 cycle delay:

```
        #define IRQ_TIMER 8
        #define IOBASE_TIMER 0x04
        #define DR0 (IOBASE_TIMER+0)
        #define RESET 1

        out RESET, ?DR0     ; reset timer
        out -10000, ?DR0
        out 1000, ?DR0
        ...
loiter:
        hlt                 ; wait for an interrupt
        jmp loiter
```

The timer cannot be programmed in the polling mode. It is the programmer's responsibility to prepare an interrupt handler for the timer interrupt before the timer starts the countdown:

```
ivector:
        .word[16]           ; the interrupt vector begins here
        ...
        mov %r0, ivector
        add %r0, IRQ_TIMER  ; the timer slot
        mov %r1, handler
        or  %r1, 1          ; this is a near interrupt handler,
                            ; so set the LSB
        mov [%r0], %r1
        ...

.align8 handler:            ; here begins the interrupt handler
        ...                 ; do something useful!
        retni
```

Terminal

The Clown terminal is a combination of a text-oriented display and a keyboard. The terminal has three I/O registers: R0, R1, and R2. The keyboard can operate in the polling mode (default) or in the interrupt mode.

The keyboard has a built-in 16-word buffer that is enabled only in the interrupt mode. If the buffer overflows, further input characters are discarded without a notice.

Register R0 is an R/W register. Reading from R0 returns `0x7F747479` (the signature of the terminal).

Writing a number into R0 prints the corresponding ASCII character on the display.

Register R1 is an R/W register. Reading from R1 returns the ASCII code of the most recently pressed key on the keyboard, or 0 if no keys have been pressed.

Writing 0 into R1 switches the terminal keyboard into the polling mode. Writing any other value into R1 switches the terminal keyboard into the interrupt mode.

Register R2 is an R register (read-only). Reading from R2 returns the status of the keyboard: 1 if a key has been pressed, 0 if no keys have been pressed.

If the keyboard is in the interrupt mode, then an interrupt on IRQ10 is raised when a key is pressed. Otherwise, it's the programmer's responsibility to check if the keyboard is ready before reading the value from the register R1.

Programming Terminal

The I/O base of the terminal is at the port 0x08. The following code fragment prints a null-terminated string to the terminal:

```
            #define IOBASE_TTY 0x08
            #define DR0 (IOBASE_TTY+0)

            mov %r0, string
write:
            mov %r1, [%r0]      ; load the next character
            cmp %r1, 0          ; is it a zero?
            jz endw             ; finish printing
            out %r1, ?DR0       ; print the character
            inc %r0             ; increment the pointer
            jmp write           ; print more
endw:
            ...

string:
            .string “Hello, world!”
```

The following code fragment reads characters from the terminal into a null-terminated string. The input is terminated by a newline character `\n`:

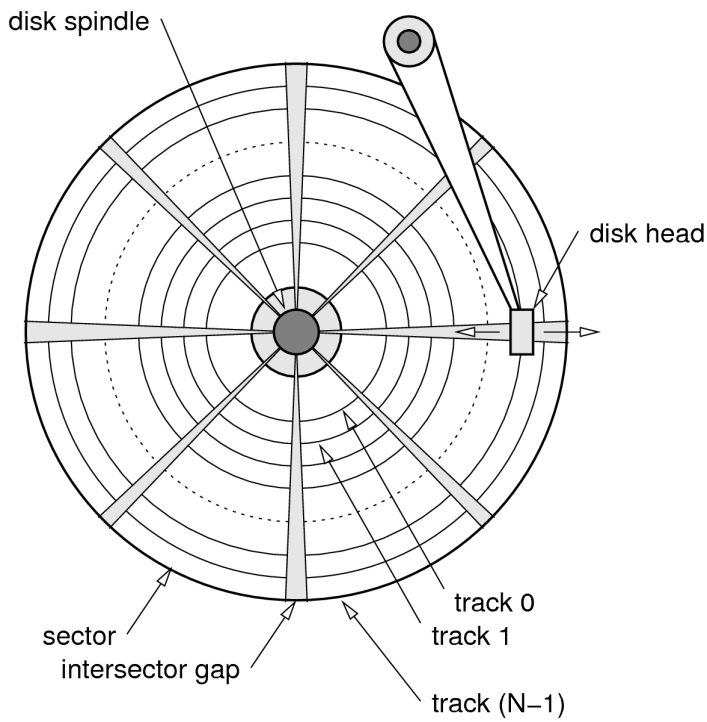
```
            #define IOBASE_TTY 0x08
            #define DR1 (IOBASE_TTY+1)
            #define DR2 (IOBASE_TTY+2)
            #define IDLE 0

            mov %r0, string
read:
            in %r1, ?DR2        ; read the keyboard status
            cmp %r1, IDLE       ; is the keyboard ready?
            jz read             ; not yet
            in %r1, ?DR1        ; read the next character
            cmp %r1,'\n'        ; is it the last one?
            jz done
            mov [%r0], %r1      ; store the next character
            inc %r0             ; increment the pointer
            jmp read            ; read more
done:
            mov [%r0], 0        ; the terminator
            ...
string:
            .word[1024]
```

Hard Disk

The organization of the Clown hard disk is shown in the picture. The default parameters of the disk are summarized in the table (scaled for the CPU speed of 10MIPS):

Parameter	**Value**
Number of tracks	64
Number of sectors per track	32
Number of words per sector	1024 (equals the memory frame size)
Number of words per gap	16
Spin rate	~6,000 rpm (100,000 CPU cycles/rev.)
Track-to-track seek latency	3 ms (30,000 CPU cycles)
Maximum seek latency	26 ms (260,000 CPU cycles)
Total disk size	2 mln words—or 8MB

The image of the hard disk is stored in the file *clown.dsk*. If shell variable



$CLOWN_DISC is set then its value is the path to the disk image file, less the file name. Otherwise, the file must be in the current working directory. Use the *makedisc* program to prepare an empty formatted disk image file.

Contiguous disk sectors on the same track are separated by an intersector gap

that does not contain data. The gap allows restarting the DMA controller in real time to support streaming input to or output from the disk.

Hard Disk Controller

The controller has three I/O registers: R0, R1, and R2—and an on-board buffer. The size of the buffer equals the size of a sector (1,024 words). Only one whole block can be read or written at a time.

Register R0 is an R/W register. Reading from R0 returns `0x7F686464` (the signature of the disk controller).

Writing a number to R0 initiates an operation, according to the table below. For some operations, another value (parameter) must be written at the next cycle into register R2:

Operation code in R0	Parameter in R2	Operation	Can raise interrupt?
0	Track number	Seek track	yes
1	Sector number	Read sector	yes
2	Sector number	Write sector	yes
3		Reset buffer pointer	no
4		Switch to interrupt mode	no
5		Switch to polling mode	no
Any other		Ignore	n/a

The controller can operate in the polling mode (default) or in the interrupt mode. If the controller is in the interrupt mode, then it raises an interrupt on IRQ11 upon completion of any of the first three operations.

Register R1 is an R register (read-only). Reading from R1 returns the status of the controller:

Status value	Meaning
0	Busy
1	Idle, new data available (only for the “read sector” operation)
2	Idle

Register R2 is an R/W register. Reading from R2 either returns the next word from the controller buffer (if there are new data available) and advances the buffer pointer to the next position, or returns a zero if there are no new data available. It is the responsibility of the programmer to check the value of register R1 to distinguish a zero datum from the zero flag.

Writing a word to R2 stores the word into the next position of the controller buffer (as specified by the pointer) and advances the pointer to the next position in the round-robin fashion (writing into the last position resets the pointer). A write to R2 that immediately follows a write to R0 may be interpreted in a special way (see the table of supported operations above).

Programming Disk Controller

The I/O base of the hard disk controller is at the port 0x0C. The following code fragment reads the 34th sector of the 7th track and loads the contents of the buffer into the RAM:

```
        #define IOBASE_HDD 0x0C
        #define DR0 (IOBASE_HDD+0)
        #define DR1 (IOBASE_HDD+1)
        #define DR2 (IOBASE_HDD+2)
        #define BUSY 0
        #define NEWDATA 1
        #define SEEK 0
        #define READ 1

        out SEEK, ?DR0      ; seek track
        mov %r0, (7-1)      ; track number
        out %r0, ?DR2
        mov %r1, array      ; initialize the array pointer
waits:
        in %r0, ?DR1
        cmp %r0, BUSY       ; check the disk status
        jz waits            ; wait for seek to finish
        out READ, ?DR0      ; read data
        mov %r0, (34-1)
        out %r0, ?DR0

waitn:
        in %r0, ?DR1
        cmp %r0, NEWDATA    ; check if new data are available
        jnz waitn           ; not ready yet

        in %r0, DR2         ; read a word
        mov [%r1], %r0      ; store the word
        inc %r1
        cmp %r1, end        ; is the buffer full?
        jnz waitn           ; not full yet
        ...
array:
        .word[1024]  ; RAM buffer
end:
        .word        ; “end of array” label
```

Note that when the controller performs the write sector operation, it never reports the “idle, new data available” status, because no new data can become available during a write operation.

DMA Controller

The direct memory access (DMA) controller supports direct memory access for the hard disk controller. Using the DMA controller, words can be transferred from the disk controller's buffer into the RAM and back bypassing the CPU (provided that the memory bus is not used by the CPU). The controller has three I/O registers: R0, R1, and R2.

Register R0 is an R/W register. Reading from R0 returns `0x7F646d61` (the signature of the controller).

Writing a number to R0 initiates an operation, according to the table below:

Operation code in R0	**Operation**
0	Reset controller
1	Trigger disk-to-memory transfer
2	Trigger memory-to-disk transfer
Any other	Ignore

Register R1 is an R register (read-only). Reading from R1 returns the status of the controller:

Status value	**Meaning**
0	Busy
1	Ready/Idle

Register R2 is a W register (write-only). If the controller is in the “idle” state, any word written into R2 is interpreted as the disk track number. The next written word is interpreted as the disk sector number. The next written word is interpreted as the physical memory address. All further writes to R2, as well as all words written to R2 when the controller is not in the “idle” state, are ignored. The actual transfer does not start until a corresponding value is written to register R0.

The controller always transfers data in blocks (1,024 words per block). Upon the completion of the transfer, the sector number is autoincremented, the memory address is increased by the block size, and an interrupt on IRQ9 is raised. Thus, the controller does not have to be reprogrammed when the data are read from or written to contiguous sectors into or from contiguous memory blocks.

Programming DMA Controller

The I/O base of the DMA controller is at the port 0x00. The following code fragment writes the contents of the buffer `block` into the sector 34 of the 7th track of the disk:

```
        #define IOBASE_DMA 0x00
        #define DR0 (IOBASE_DMA+0)
        #define DR1 (IOBASE_DMA+1)
        #define DR2 (IOBASE_DMA+2)
        #define RESET 0
        #define MEM2DSK 2


        mov %r1, block      ; initialize the array pointer
        out RESET, ?DR0     ; reset the controller
        out (7-1), ?DR2     ; store track number
        out (34-1), ?DR2    ; store sector number
        out %r1, ?DR2       ; store memory location
        out MEM2DSK, ?DR0   ; trigger memory-to-disk transfer
        hlt                 ; wait for the transfer to finish
        ...
block:
        .word[1024]      ; data area
```

70. The Clown Companion

DISK TOOLS

Makedisc

makedisc is the Clown disk formatting utility.

makedisc takes five command line parameters:

- the number of tracks
- the number of sectors per track
- track-to-track seek latency (in CPU cycles)
- maximum seek latency across the disk (in CPU cycles)
- disk rotation speed (in CPU cycles per revolution)

All parameters must be positive integer numbers. They do not have any default values. The new disk must have at least three tracks. The track-to-track seek latency must be less than the maximum seek latency.

Example:

```
shell> makedisc 64 32 30000 260000 100000
```

The new disk image is written into the file *clown.dsk*.

Incidentally, *makedisc* is the oldest component of the Clown system.

Clodd

clodd is the Clown disk copy utility (named after the Unix *dd*). It is used to copy host system files to the Clown disk image file and back.

The last parameter on the command line must be the name of the host system file *FILE*. All other command line parameters are optional.

clodd can take the following command line parameters:

- -h, --help – display the help message and exit
- -sector *SEC* – read from or write to sector *SEC* (default 0)
- -track *TR* – read from or write to track *TR* (default 0)
- -count *CNT* – read *CNT* blocks (default: read the entire disk)
- -unpack – record each 8-bit character as one Clown word while writing to the disk, or record each Clown word as an 8-bit character while reading from the disk
- -fromdisk – read from the disk into the *FILE* (the *FILE* will be overwritten)
- -todisk – write from the *FILE* to the disk (the *FILE* must exist)
- -q, --silent – work silently
- -v, --version – print program version number and exit

Exactly one of *-fromdisk* or *-todisk* options must be present: it determines the direction of data transfer.

The disk image is assumed to be in the file *clown.dsk*. If shell variable *$CLOWN_DISC* is set then its value is the path to the disk image file, less the file name. Otherwise, the file must be in the current working directory.

The default behavior of *clodd* is to copy data verbatim, so that each host word is recorded as a Clown word, and vice versa. If the *-unpack* option is used, then *clodd* records each host byte as a Clown word, and each Clown word as a host byte (the least significant byte of the Clown word is recorded, the other three bytes are discarded). This option is useful when copying text files: Clown has only one data type, and a printable character is internally represented as a word.

In this example, the 5th sector of the 12th track is copied verbatim into file *sector.dat*:

```
shell> clodd -track 11 -sector 4 [-count 1] -fromdisk sector.dat
```

In this example, text file *othello.txt* is copied to the Clown disk image, starting from the first sector of the second track, and unpacked:

```
shell> clodd -track 1 [-sector 0] -todisk -unpack othello.txt
```

In both examples, the parameters in brackets are optional.

74. The Clown Companion

LINKER

Interface Summary

clink is the Clown linker. It is used to combine multiple CLO (Clown object) files into one CLE (Clown executable) file or CLO (Clown object file or library).

The linker supports the following command line options:

- -h, --help – display the help message and exit
- -b, --cle – generate a non-relocatable executable CLE module (default)
- -s, --clo – generate a relocatable CLO module (a library)
- -o OFILE – set the output file name to *OFILE* (the default name is default.cle for CLE modules and default.clo for CLO modules)
- -v, --version – print linker version number and exit
- -l, --listing – produce code listing
- -q, --silent – operate silently, without reporting warnings and error messages
- -ng, --nodebug – do not include file and line information for debugging
- -V – print linker version number and continue
- -- – treat the next argument as a file name, even if it looks like yet another command line option.

Any other command line argument that either does not begin with a dash or is preceded by two dashes --, is treated as an object file name.

CLO and CLE modules

Clown supports two types of binary modules: CLO (Clown object file) and CLE (Clown executable file). Both types of modules are organized as XML files and can contain debugging information (source file names and line numbers), human-readable segment descriptors and symbol descriptors, and binary executable or object code in Base64 encoding.

A CLE module is self-contained. It does not have any undefined symbols or segments, or relocation information. A CLE module can be immediately executed by Clown. A CLE module cannot be combined with other modules.

To produce module *test.cle,* assemble the corresponding source file *test.*s using *cas*:

```
shell> cas test.s [-b -o test.cle]
```

The parameters in brackets are optional.

A CLO module cannot be immediately executed because it refers to undefined symbols or undefined segments, or contains relocation information. It must be linked with other modules to resolve dependencies or processed by the linker to remove the relocation hints.

To produce module *runme.clo* that consists of two object files, *getsputs.clo* and *stdio.clo*, combine the object files using *clink*:

```
shell> clink -o runme.cle getsputs.clo stdio.clo [-b]
```

The parameter in brackets is optional.

To produce a library module (a CLO module that consists of several CLO modules), use *clink* with the `-s` option:

```
shell> clink -s -o libc.clo stdio.clo math.clo ...
```

Once built, a library loses its internal modular structure and behaves as if it were a single indivisible object module.

Program Listing

When invoked with the `-l` command line option, *clink* generates both the output binary file and the program listing. The format of the listing is explained on page 33.

TYPICAL ASSIGNMENTS

String I/O

Write a program that has functions `gets()` and `puts()` and calls them repeatedly in an infinite loop, thus echoing the input to the output.

Boot loader

Write a “ROM boot loader.”

There is a compressed disk image in *examples/bootloader.dsk.gz*. Uncompress the file and rename it to *clown.dsk*.

There is a program (“operating system loader”) in the first sector of the first track of that image (track=0, sector=0) . The sector size is 1024 words. Your program will read the content of the sector into the Clown memory, starting at the location 0, and pass control to the first instruction of the OS loader. If your program is correct, you will see a meaningful message on the screen.

Advanced boot loader

Modify the boot loader from the previous assignment.

This time, use the DMA controller instead of direct polling. Develop two functions, `readSector (track, sector, &buffer)` and `writeSector (track, sector, &buffer)` (remember that all sectors have the same size). Each function programs the DMA controller, initiates the DMA transfer, and then blocks the CPU with the `hlt` instruction and waits until the DMA transfer completes. (This is not how a really smart operating system is supposed to behave, but good enough for the purpose of the exercise.) The DMA controller generates an interrupt when the transfer is complete that unblocks the CPU. For the purpose of this exercise one can have one single `retni` instruction in the interrupt handler.

Call the functions in the following order:

1. readSector (0,0,&buffer);
2. writeSector (0,1,&buffer); /* to test the write function */
3. readSector (0,1,&buffer);
4. Pass the control to the beginning of the buffer, as before.

Allocate the interrupt vector at the beginning of the RAM (the first 16 words). Make sure that your data and your code do not overwrite the interrupt vector. Use `jmp` instruction to "jump over" the vector.

The Clown DMA controller uses the interrupt request channel (IRQ) number 9. Initialize the corresponding slot of the vector. All other slots can be left uninitialized.

In the "flat" memory mode (the default Clown mode with no paging and no segmentation), a slot in the interrupt vector contains the address of the interrupt handler, followed by two mode bits that define the privilege level for the handler, followed by 1. This means that the three least significant bits of the address are not used and are assumed to be 001, and that an interrupt handler must be aligned at an 8-word boundary. Use `.align8` modifier to enforce this alignment.

System call interface

In the previous exercises, you implemented four basic kernel space I/O functions: `gets`, `puts`, `readSector`, and `writeSector`. In this exercise, implement a system call interface to these functions.

Rename the functions to `_gets`, `_puts`, `_readSector`, and `_writeSector`.

The interface consists of a system call handler (IRQ15 handler) and four wrapper functions.

The wrapper functions: `gets`, `puts`, `readSector`, and `writeSector`—push the parameters that are later used by their "underscored" counterparts, onto the stack. Then they push the system call number that is unique to each function (for example, 0 for `gets`, 1 for `puts`, etc.; use `#define` to avoid "magic numbers") and `trap` the OS.

The system call handler explores the third stack element that contains the system call number (the top of the stack and the second element contain the return address and the flags). Based on the number, the handler prepares the stack for the kernel function call and calls the corresponding "underscored" function. Upon the completion of the function execution, the handler saves the value returned by the function in the stack (`gets` and `puts` return values) and returns from the interrupt (`retni`).

The control returns to the wrapper function, which in turn returns to the caller.

The test program prepares the handlers (one for the DMA controller and one for the `trap`), switches the CPU into the protected mode by changing the CPL flag to 1 (see example on page 56), and then reads a string from the keyboard, writes it in a disk sector, reads it from the sector into a different buffer, and prints the buffer.

Multitasking

Write a collection of Clown programs that simulate multitasking.

Write two simple applications (printing programs): one that prints a star every N cycles, and another that prints a pound sign (#) every M cycles (M and N should be probably on the order of 10,000; you can adjust—calibrate—them to produce nice-looking output, not too fast and not too slow). Assemble these programs with different non-zero initial offsets O1 and O2 (use the `-e` option).

Write the main program that creates two PCBs for the applications. The process control block (PCB) should have space for the `%pc`, the `%flags`, and for the registers that you plan to save (probably one or two registers is enough, depending on whether the loop counter is stored in a variable or in a register). The main program also maintains the current process ID variable that equals 0 for the star-printing program and 1 for the pound-printing program.

The main program initializes the PCBs, the process ID variable, and the timer interrupt handler and starts the timer in the interval mode. The timer expires every K cycles, where K>>max(M,N). Remember that by default all CPU `%flags` are clear, except the interrupt flag. The PCB of the second application must have correct initial `%flags` (namely, the Interrupt flag), or else the timer interrupts may be disabled forever after the first context switch.

Then the main program starts one of the applications by `jmp`ing to its first instruction.

When the timer expires, the interrupt handler saves the `%pc`, the `%flags`, and some general-purpose registers in the current PCB (as defined by the PID variable), updates the PID variable, restores the `%pc`, the `%flags`, and some general-purpose registers from the next current PCB, and returns into the new process. Remember that when an interrupt happens, the interrupt controller pushes the `%pc` and the `%flags` onto the stack. The `retni` instruction expects to find them on the top of the stack exactly in the order they have been saved.

To test your system, load all three pieces into the Clown memory: the main program at address 0, and the applications at the addresses O1 and O2 (use the `-b` option). The addresses O1 and O2 are the same as at the assembly phase. Leave a plenty of space for each program, so that the programs do not overlap. Run the main program and enjoy the show.

Keyboard Buffer

Implement a keyboard buffer.

Allocate N words in the Clown memory for the bounded FIFO wrapped-around buffer. Switch the keyboard in the interrupt mode. Initialize the keyboard interrupt handler that, when invoked, collects the most recently pressed key code and stores it in the next available position in the buffer. If the buffer is full, the key code is ignored. There is no need to check if the keyboard is ready before collecting the character.

After initializing the handler, start an "application" that checks the buffer every M cycles (M>10000). If the buffer is not empty, then the "application" reads and prints the least recently inserted key code.

Since the head (start) and the tail (end) of the buffer are the variables shared among the interrupt handler and the "application," you may need to use the mutual exclusion mechanism based on the `xchg` instruction to access these variables, both in the "application" and in the handler.

Page table

Implement paging.

Write a simple application that prints a star every N cycles. Place the application code into the second page by assembling the program with the offset of 1024 (use the `-e` option). Remember that the first page, page #0, is reserved for the "operating system" and must be in the first frame, frame #0.

The main program sets up the Clown paging system. You will need a page directory (one page) and at least one page table (another page). Use `.page` modifier to align both data structures on page boundaries.

Enter at least two lines into the page table: one that maps page #0 to frame #0, and another that maps page #1 to any other frame (e.g., frame F=100). The main program turns paging on and passes control (`jmp`) to the application code at the beginning of the second page.

To test your system, load both pieces into the Clown memory: the main program into the first frame, and the application into the frame F (use the `-b` option). Run the main program. If it is correct, the application will print the stars, as if it were loaded in the second frame.

Files

There is a primitive file system on disk *examples/message.dsk.gz*. Uncompress the file and rename it to *clown.dsk*. The system has only one anonymous file. The file begins at sector 1, track 0. The first word of any block of the file is a printable character, and the last two words are the track number and the sector number of the next block. The last two words of the last block are zeros. Write a Clown assembly program that display the message stored in the file.

Segments

Place the Interrupt Vector and the stack in individual segments and test your program by successfully handling some interrupt (e.g., divide `%r0=0` by `%r0` and on exception change `%r0` to 1).

Declare four segments: `.code $mycode` for the application code, `.data $stack` for the stack, `.data $iv` for the interrupt vector, and `.code $handler` (the actual choice of name is yours).

The application code prepares the stack by initializing `%ss` and `%sp`, then registers the exception handler(s) in the interrupt vector, saves `$iv` to `%isr`, and finally causes the testing exception.

The handler code is stored in a separate segment `$handler`. There is no need to `.align8` it and to set the least significant bi to 1.

ALPHABETICAL INDEX

R

S

T

-

.

#

%