INDRA
Working
Paper

IEN 198
INDRA Note 1041
15th. September 1981

Extended Memory MOS for a UNIX Development Host

Robert Cole

ABSTRACT: This note describes a version of the RSRE Extended Memory MOS that can be used with a UNIX development host. The run-time MOS system design is explained. The linking and loading phases are described. Documentation is included in an appendix.

Department of Computer Science
University College, London

C O N T E N T S

1. Introduction

Recently a version of MOS (Micro Operating System) was developed at RSRE [1] which enabled the extra memory capacity of 11/23 and 11/34 computers to be used. The development was carefully designed to increase the buffer space available to all processes without adding significant 'context switching' overheads. A further development at UCL has used the same design, but for use on a UNIX development host. Standard UNIX software does not cater for extended memory systems to be built by users. As a result a new linker was produced, giving some flexibility to the resulting MOS system. It was decided to extend the EMMOS system of RSRE by using both USER and KERNEL addressing modes to provide extra protection and addressing space.

Figure 1 shows the two address mode maps that are used at runtime.

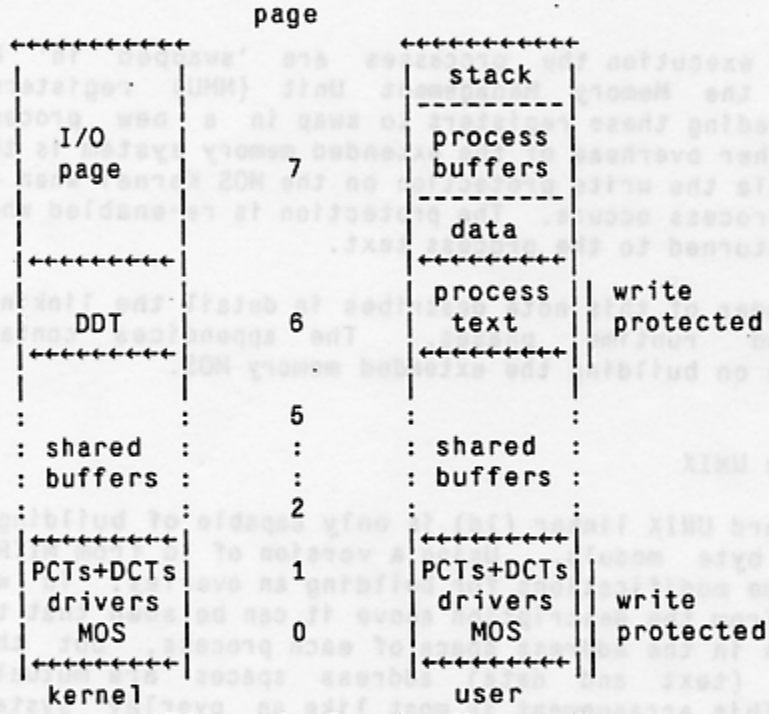

Figure 1. Runtime Memory Map

Here the I/O page is only available to the Kernel mode and both MOS and the process instructions are protected during process execution. The UNIX object format naturally separates text (instructions) from data (program variables) so it is possible to use this in our design. The last page of the user address space is loaded with the process data at the bottom and the process stack is created at the top. The full 8K bytes is allocated so any space left between can be used as

process private buffer space and managed by the system buffer management code.

Buffers, which are common to all processes, are set up above the MOS Kernel and below the processes. Normally the processes are all loaded in pages 6 and 7, however a very large process may force this boundary down to page 5. The size of the MOS Kernel will vary from system to system depending on the number of processes, the number and nature of devices, and any other code that is shared between processes. The MOS Kernel size is rounded up to the next page boundary. Figure 1 shows 2 pages being used for MOS. The Kernel mode (and MOS initialisation) stack is placed at the top of the MOS Kernel space. Thus the shared buffer area is squeezed by code requirements, however figure 1 shows 4 x 8K of buffer space (half the virtual address space) and is probably the worst case. In addition most processes will have some private buffer space.

During MOS execution the processes are 'swapped in' by manipulating the Memory Management Unit (MMU) registers. Apart from loading these registers to swap in a new process the only other overhead of the extended memory system is the need to disable the write protection on the MOS Kernel when an EMT from a process occurs. The protection is re-enabled when control is returned to the process text.

The remainder of this note describes in detail the linking, loading, and runtime phases. The appendices contain documentation on building the extended memory MOS.

2. Linking on UNIX

The standard UNIX linker (ld) is only capable of building a single 64K byte module. Using a version of ld from MITRE, which had some modifications for building an overlay, ld was extended. From the description above it can be seen that the MOS Kernel is in the address space of each process, but that the process (text and data) address spaces are mutually exclusive. This arrangement is most like an overlay system, with the MOS Kernel as the root. The program, lk (lk for link), was used to build the root and link in each overlay separately. Obviously each overlay should be able to access the root, but not vice versa.

The lk program uses two symbol tables, one for the root, and one for the current overlay. The overlay symbols are purged when the overlay definition is complete and the overlay has been linked. A mechanism was required for MOS to identify overlay code at runtime for swapping, as well as for them to

be easily indicated from the C configuration file. The mechanism chosen is to label each overlay by a tag symbol at link time. The symbol is given a unique value by the linker. The symbol itself can be used in C programs, and later in DDT, to identify a particular overlay (process).

An example command sequence to lk could be:

```
lk mos.o config.o \
{ ←tag1 -n -b 0140000 -e ←ent1 proc1.o lib.a }
```

where

```
{ ... } defines an overlay
←tag1 is the tag symbol, the leading underscore is
 required where the symbol is referenced in C code
-b 0140000 gives the base address at which the overlay
 code is to linked (page 6)
-e ←ent1 defines the entry address for this overlay
```

The -b flag is used to relocate the overlay code to the top end of the address space. The -n flag causes the data to be relocated to the next page boundary (if possible). At UCL we use a modified version of cc, called mcc, to build the MOS load file. Mcc will expand the "{ tagn" to become "{ tagn" -n -b 014000" and the "}" become "-l }", thus providing defaults.

The object code output is organised as a number of subfiles, one for the root, and one for each overlay. These subfiles are packed into a single output file using the ar (V) format. Size (I) and nm (I) have been modified to accept files in ar format and operate on the subfiles.

From the runtime map, fig. 1, it can be seen that DDT is in the Kernel address space, but at the opposite end to that of MOS. To achieve this lk recognises the tag "ddt" and marks it as being in the root (by the magic word) but treats the module as an overlay for linking purposes.

The normal a.out (V) header has been modified by using the unused word to include the tag symbol value and the base page into which the overlay should be swapped. The format is now:

```
typedef struct filhdr { /* a.out format header */
        int     magicw;
#define VMAGIC  0405
#define FMAGIC  0407
#define NMAGIC  0410
#define IMAGIC  0411
#define ROOTMAGIC 0515 /* root file in overlay system */
#define OVNMAGIC  0525 /* overlay with separate I and D */
#define OVFMAGIC  0535 /* overlay with contiguous I and D */
        int     tsize;
        int     dsize;
        int     bsize;
        int     ssize;
        int     entrypt;
        char    tag, page;      /* for MOS overlays */
        int     relflg;
} FILHDR;
```

3. Loading

Once the complete MOS system has been linked and is ready in the ar format file; it has to be loaded into the LSI or PDP-11. To manage the MMU and set up the overlays ready for swapping a special loader is used in the target computer. At UCL we load this as an ordinary program using a down-line loader. The program is then automatically started by the primary bootstrap and continues as the (secondary) bootstrap.

Each subfile is passed to the secondary bootstrap in two parts, first the 16 byte a.out header, followed by the text and data. The a.out header enables the bootstrap to see if the file contains root code or is an overlay.

If the subfile is a root file (magic word = 0515) it is loaded into the 1st 6 physical pages and the MMU is not used. Any symbols associated with the root are placed at the end of the last physical page (page 30), the amount of space left for the root symbols is an assembly constant shared between the virtual bootstrap, virtual MOS and DDT. The page byte in the a.out header allows loading to begin on any page boundary. Note that DDT is loaded into page 6 using this mechanism. The secondary boot operates from the very top of physical page 6 so that DDT should not overwrite this.

If the subfile is an overlay it is loaded above physical location 0160000. The text and data are loaded contiguously in 1K byte blocks from UNIX. When the text and data have been loaded (the a.out header contains the size, this is used as a check against the amount received) the physical address of the

top of the 8K process page boundary is calculated. The layout of a process is shown in Fig. 2.

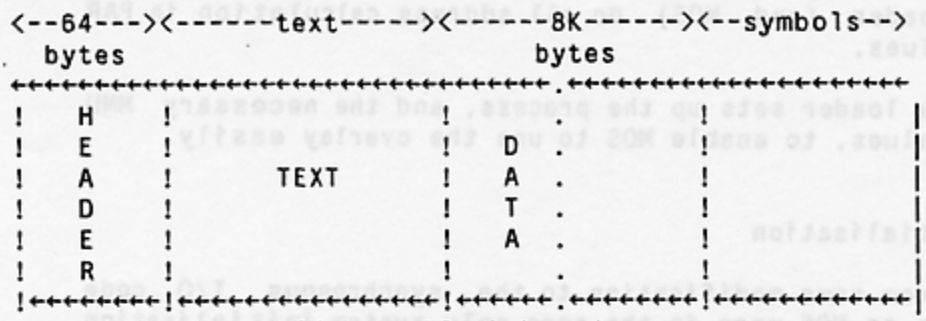

Figure 2. Overlay load map.

The a.out header is copied into the first 64 byte block, this block is used as a control block for the overlay. The format of this control block is:

```
typedef struct { /* overlay header block structure */
  int magic;      /* magic word */
  int tsize;
  int dsize;
  int bsize;
  int ssize;
  int entry; /* entry address for process */
  char tag;
  char page; /* same as a.out header up to here */
  int used;  /* indicates overlay already used */
  int memlo; /* address of first free loc in o/v */
  int next;  /* points to next overlay header */
  int par[16]; /* MMU register contents to */
  int pdr[16]; /* 'swap' this process in */
} OVERLAY←HEADER;
```

The base of the 8K page is, of course, the first data byte. This means there is a hole in the process address space between the end of the text and the first word of the data. There is no physical memory assigned to this area. If the subfile has any symbols these are loaded above the extended data page. When the data and text have been loaded the bss area is set to zero, this only occurs at load time. When the symbols have been loaded the virtual loader calculates the PAR and PDR register contents for the process by assuming all unloaded pages are allocated to the low physical memory and are full (all 8K in use). The loaded pages are given the PAR and PDR values needed to 'swap' in this process. The first location in the process private buffer pool is calculated (size of data + bss + 2 + 0160000) and the PAR contents

needed to access the next process control block. In this way the first overlay control block is always at 0160000 and subsequent overlays can be found by using the chain. The virtual loader (and MOS) do all address calculation in PAR content values.

Thus the loader sets up the process, and the necessary MMU register values, to enable MOS to use the overlay easily.

4. MOS Initialisation

Apart from some modification to the synchronous I/O code the changes to MOS were in the once only system initialisation code.

The MMU is enabled as soon as MOS loads the kernel mode registers to provide the same mapping as though the MMU was disabled. A window page is used to map into the overlays by loading a single PAR register with the values provided by the loader. Any spare physical memory between the last overlay and the MOS Kernel symbols is divided into 8K pieces and initialised with a control block. These may be used when two or more processes share the same text.

The control block used for each overlay means there was no need to increase the size of the Process Control Table (PCT), this was convenient as the size is currently 32 bytes.

5. MOS Runtime

As pointed out in [1] it is necessary to use the shared buffer area for all I/O and especially IORBs. This was the only change that needs to be made to MOS processes. However, some processes share routines and data space, these have to be loaded in the Kernel. All the facilities described in [2] are available.

The synchronous I/O routines (sin, sout, bin, bout) will copy data between the process data page and the shared buffers, if necessary.

6. DDT

The debugging program DDT is always loaded in the extended MOS. This has the same commands and features as the ordinary DDT. In addition the following command will 'swap in' a process, and its symbol table.

```
<tag>$1 - load an overlay
$1      - load the 'normal' address space
```

The <tag> should be a tag symbol used in linking. If no tag is given; DDT itself, and the I/O page are mapped in. Processes can be swapped randomly in DDT, this does not affect the running of the system.

The only (apparent) facility not readily converted is the use of single step over RTT and RTI instructions where the address mode is changed by that instruction.

Note, that when DDT is entered via a break (breakpoint or single step) PS and SP contain the values in use at the time of the break, thus are not necessarily the SP of the mode in which DDT is running. Also when the break occured in user mode the address space contains the user process, not the I/O page.

References

1. Wiseman, S.R., Davies, B.H., Memory Management Extensions to the SRI Micro Operating System for PDP- 11/23/34/40, IEN 136, May 1980

2. Cole & Treadwell, MOS User Guide, 2nd Ed., Indra Note 1042

APPENDIX

This appendix contains an example of building a virtual MOS system using the normal mechanisms, plus the system documentation for mcc and lk.

Example of Building a vMOS System

The vMOS system is defined in a cnf file in the same way as an ordinary system. The difference is that the lk 'tag' is used instead of the entry address. In our example a TCP test system is built:

```
extern   monitor, control, vcs, tcp, inet, monsl;

PCTE pcts[]     {
  setpcte("opcon", &control, &dt1i, &dt1o, 300, 0, 0),
  setpcte("tcp", &tcp, &cty1, 0, 200, 0, 0),
  setpcte("internet", &inet, 0, 0, 300, 0, 0),
  setpcte("cstest", &vcs, &cty2, 0, 200, 0, 0),
  setpcte("monitor",&monitor, 0, 0, 200, 0, 0),
  setpcte("monslave", &monsl, 0, 0, 200, 0, 0),
     };
```

Note that there is no mechanism for automatically loading complete processes so the Internet and monitoring processes are explicitly named. For this reason there is a special version of the TCP process that does not attempt to spawn those processes.

The commands to mcc must indicate all of the procceses. In this example a large number of routines are loaded in the kernel to enable them to be shared amongst processes. In fact this is to do with the monitoring and all that should be shared is the name of the routine.

```
mcc -o vtcptst tcpcnf.c vtcppnt.c monpid.c \
inprnt.c inaddr.c vprintf.o -l1 \
{ control -e opcon opcmds.c -l1 } \
{ vcs -e cstest cstest.o } \
{ monitor -e vmontr -lv } \
{ monsl -e monslave -lv } \
{ tcp -e vtcpm -lv } \
{ inet -e inetm -lv -l1 }
```

To show how mcc expands this command the actual command line to lk is shown below. Note that new-lines have been inserted to improve the clarity of the example.

```
lk -X /ucl/mos/lib/vmos.o -o vtcptst tcpcnf.o vtcppnt.o
monpid.o inprnt.o inaddr.o vprintf.o -l1
{ ←control -b 0140000 -n -e ←opcon opcmds.o -l1 -l }
{ ←vcs -b 0140000 -n -e ←cstest cstest.o -l }
{ ←monitor -b 0140000 -n -e ←vmontr -lv -l }
{ ←monsl -b 0140000 -n -e ←monslave -lv -l }
{ ←tcp -b 0140000 -n -e ←vtcpm -lv -l }
{ ←inet -b 0140000 -n -e ←inetm -lv -l1 -l }
ddtstk.o
{ ddt -b 0140000 -s /ucl/mos/lib/vddt.o }
-l
```

When converting from an ordinary MOS system to a vMOS system the 2 major problems are:

1. Different processes that share memory or routines, or even addresses.

2. Searching libraries so that only the required routines are built into an overlay, and not included in the kernel.

NAME

lk - link editor for Extended Memory MOS

SYNOPSIS

lk [{ tag] -sulxnebo name ... [}]

DESCRIPTION

Lk is a modification of ld(I) that can build an overlay system of the type used in the Extended Memory MOS system. All the features of ld are supported. If no overlays are defined lk behaves exactly the same as ld, only less efficiently.

Overlays are defined by enclosing switches and filename arguments between curly brackets "{}". Each overlay has to have a name, thus the first argument after every open curly bracket is taken as the symbol naming that overlay. The symbol is known as the overlay tag. Each tag is given a unique value by lk. The overlay definition syntax is:

{ tag -sulxneb name... }

Any arguments not bracketed are used to define the root. Root arguments may appear in any position between overlay definitions. All symbols defined in the root arguments are available to all overlays. Symbols defined in an overlay are purged when processing of the overlay is complete. Thus it is not possible to have references between overlays.

Lk produces output in ar(V) format. The root and each overlay are separate subfiles in the output. Each subfile is a complete a.out(V) format file with symbols if required. The default output name is "a.out".

Where symbols are used in the argument list (in -e, -u, and as the overlay tag) they must be preceded by an underscore if they are also defined in C programs.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries is important.

Lk understands several flag arguments which are written preceded by a '-'. Except for -l, they should appear before the file names.

The options suxXneb have a local effect when used within an overlay definition. If used outside an overlay their effect is global, but can be overridden by other flags or values.

-s 'squash' the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by strip.

-e The next argument is taken as a symbol defining the entry address of the overlay or object output.

-u take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-l This option is an abbreviation for a library name. -l alone stands for 'liba.a', which is the standard system library for MOS systems. -lx stands for 'libx.a' where x is any character. A library is searched when its name is encountered, so the placement of a -l is significant. The MOS libraries are kept in the MOS system directory and lk searches this directory for the libraries defined by the -l switch.

-x do not preserve local (non-.globl) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.

-X Save local symbols except for those whose names begin with 'L'. This option is used by mcc to discard internally generated labels while retaining symbols local to routines.

-n Arrange that when the output file is executed, the text portion will be read-only and shared among all users executing the file. This involves moving the data areas up the the first possible 4K word boundary following the end of the text. In an overlay this has the effect of relocating the data to the next memory page.

-o The next argument is taken the name of the output file, this overrides the a.out default.

-b The next argument is taken as a number giving the base address at which linking is to begin. This is used to relocate overlays above the root. An octal number

may be given by using a leading zero (0).

FILES

lib?.a	libraries
liba.a	MOS system library
lib1.a	TCP/IP/OPCON library
lib2.a	X25 library
libv.a	virtual MOS library
a.out	output file

SEE ALSO

as(I), ld(I), ar(I)

BUGS

ORIGIN

MITRE, then Robert Cole at UCL.

NAME

mcc - C compiler and MOS system builder

SYNOPSIS

mcc [{ tag] [-c] [-p] [-O] [-F] [-S] [-P] file ... [}]

DESCRIPTION

Mcc is the UNIX C compiler which is used to produce a MOS system load file. Mcc is very similar to cc(I) except that the command line to the loader (ld(I) or lk(I)) causes a MOS kernel to be included and MOS system libraries to be searched.

Arguments whose names end with '.c' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.c'. The '.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

If curly brackets are used in the argument list an extended memory system (vMOS) is assumed. For building vMOS systems mcc will call lk rather than ld for linking. Some expansion is done by mcc for certain arguments in a vMOS system,

"{ tag" becomes "{ ←tag -b 0140000 -n"
"}" becomes "-l }"

Arguments following the -e switch have an underscore prepended to convert C symbols to overlay symbols, the overlay tag is similarly treated. The virtual kernel is loaded, and the DDT segment added to the lk command.

Mcc will compare the modify dates on all source files ('.s' or '.c') with any object files ('.o') of the same name. If the object file has a later modify date then the source file is not compiled.

The following flags are interpreted by mcc. See ld(I) or lk(I) for load-time flags.

-c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

-p Causes a production version of the MOS kernel to be loaded (mosp.o). The default MOS kernel is mosr.o which includes DDT.

-O Invoke an object-code optimizer.

-F Force the compilation of a source file, even if there is a later object code file available.

-S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed '.s'.

-P Run only the macro preprocessor on the named C programs, and leave the output on corresponding files suffixed '.i'.

Other arguments are taken to be either loader flag arguments, or C-compatible object programs, typically produced by an earlier cc or mcc run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce a loadable MOS system with name a.out.

FILES

file.c	input file
file.o	object file
a.out	MOS system output
/tmp/ctm?	temporary
/lib/c[01]	compiler
/ucl/mos/lib/mos[pr].o	MOS kernels
/ucl/mos/lib/liba.a	MOS system library
/ucl/mos/lib/lib[12v].a	MOS utility libraries

SEE ALSO

''Programming in C- a tutorial,'' C Reference Manual, ld(I), lk(I), load(I)

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. Of these, the most mystifying are from the assembler, in particular ''m,'' which means a multiply-defined external symbol (function or data).

BUGS

ORIGIN

UCL - Robert Cole