NDIX Memory Layout.

Mike Bayliss

Operating Systems Group
Norsk Data Ltd.
Benham Valence
Newbury
Berkshire, RG16 8LU

(0635) 35544

ABSTRACT

This note outlines the layout of physical and logical memory established by the NDIX kernel.

1. Introduction.

The first actions of an NDIX kernel are concerned with changing its memory mapping, and establishing several new segments. This note describes these actions, and their intended result.

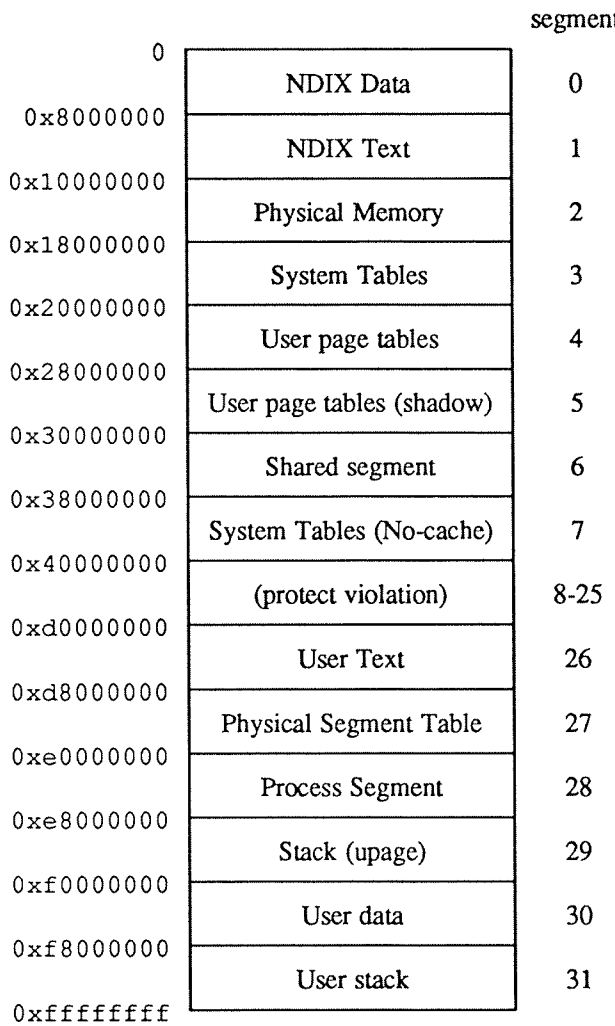
The contents of the segments are also described, together with details of how they are addressed.

2. Logical Memory Layout.

The layout of the kernel text segment is as described in the interface specification.

The layout of the kernel data segment is shown below, the segments in this mapping are:

- 0 The NDIX kernel data segment.
- 1 The NDIX kernel text segment.
- 2 A mapping of the physical ND500(0) memory.
- 3 A segment containing additional kernel data.
- 4 The page tables for NDIX user processes.
- 5 The shadow page tables for NDIX user processes.
- 6 A segment shared between the ND100 and NDIX, used for communication between the kernel and NDIX driver.
- 7 A segment containing additional kernel data that is accessed by the ND100 and which cannot be cached.
- 26 The current NDIX user process text segment.
- 27 The Physical Segment Table.
- 28 The Process Segment for the NDIX kernel.
- 29 The NDIX kernel stack segment.
- 30 The current NDIX user process data segment.
- 31 The current NDIX user process stack segment.



3. Detailed segment contents

3.1. NDIX Data

This segment is a standard UNIX† data segment, containing the kernel's data and bss segments.

It must be noted that for several dynamically allocated data structures, only pointers are found in this segment.

3.2. NDIX Text

This segment is the executable NDIX.

3.3. Physical Memory

This segment maps the whole of physical memory, including pages allocated to Sintran.

For some dynamically allocated data structures this is the only segment in which the structure actually appears and is addressable in.

† UNIX is a registered trademark of AT&T in the USA and other countries.

3.4. System Tables

This segment contains the following data structures:

usrpil — The first level indexes for the user process page tables. This area is 3 pages for each possible process.

forkutl — Extra *user* structure used during forking.

xswaputl

xswap2utl

swaputl — Extra *user* structures used during swapping.

pushutl — Extra *user* structure used during paging.

vmmap — A scratch page, used by the memory driver.

mbutl — Used by the IPC system. This area is *NMBCLUSTERS* pages long.

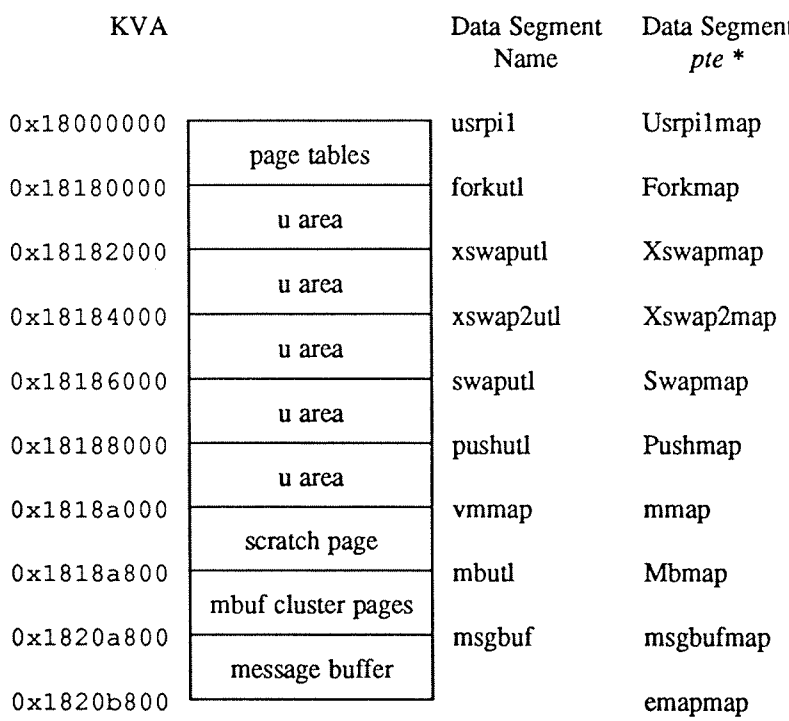
msgbuf — The message buffer. This is 2 pages.

Note that the contents of this segment are scattered throughout physical memory, and may change physical location while running.

Assuming:

256 processes
NMBCLUSTERS = 256
sizeof (struct user) = 8 pages

the segment layout and addresses are:



3.5. User page tables

This segment contains the page tables for user processes.

3.6. User page tables (shadow)

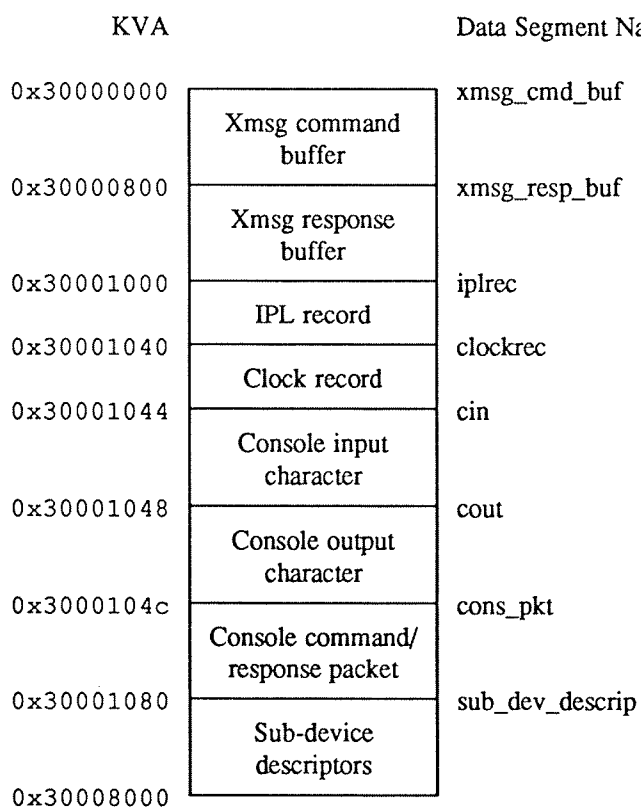
These are the shodows of the previous segment, and contain disk addresses if the page is invalid, or access and sharing information.

3.7. Shared segment

This segment consists of data shared between the NDIX kernel and the NDIX driver. It consists of the following data structures (which are described in the interface specification):

Xmsg command buffer
Xmsg response buffer
IPL record
Clock record
Console input character
Console output character
Console command/response packet
Sub-device descriptors

the segment layout and addresses are:



3.8. System Tables (No-cache)

This segment contains various kernel data structures that are frequently accessed by the ND100 and which are therefore not cached. The structures involved are:

xmsgdata Xmsg data buffers. The buffer is 2600 bytes long. There are two buffers for each xmsg channel.

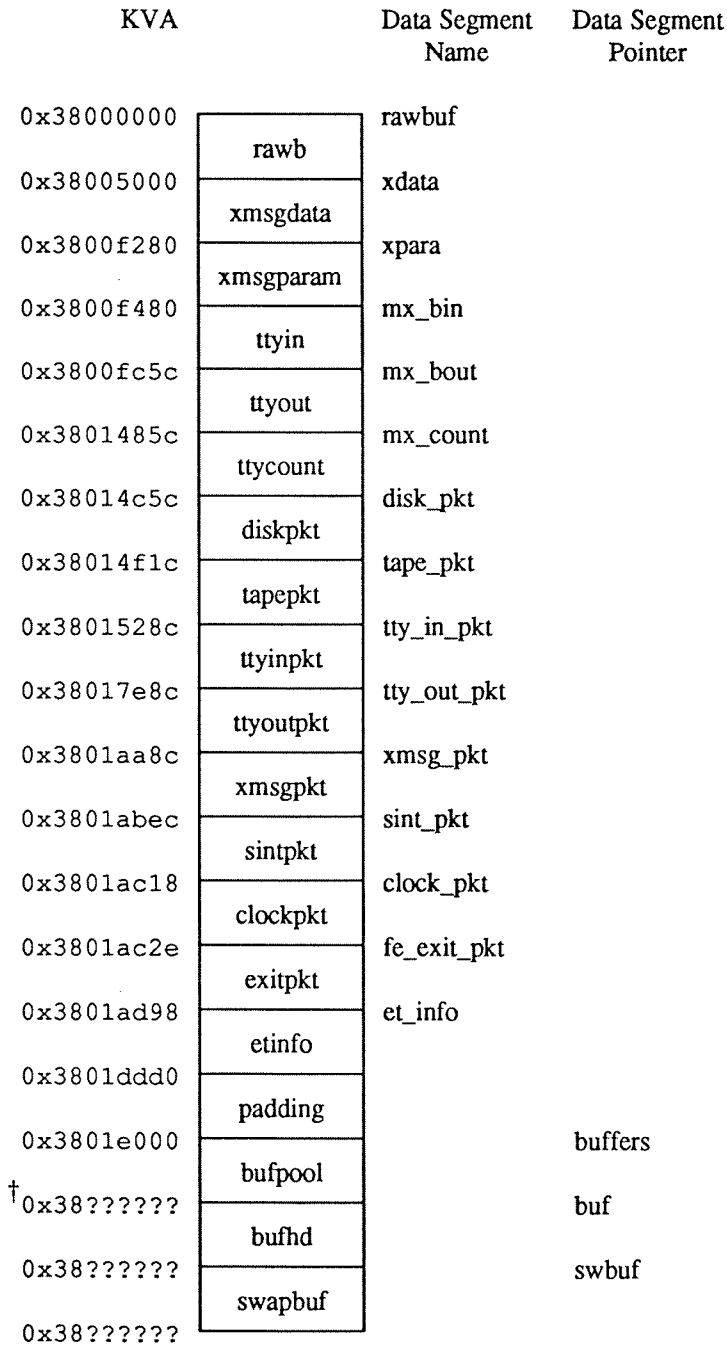
xmsgparam Parameter blocks for xmsg multicalls. These blocks are each 64 bytes long.

rawb	Raw I/O buffers. These buffers are 20480 bytes large.
ttyin	Terminal input queue. This is 2012 bytes long.
ttyout	Terminal output queue. There is one queue 76 bytes long for each possible terminal.
ttycount	This is a pair of unsigned shorts for each terminal, the first containing the number of characters read by the NDIX driver, and the second containing the number of characters read from the driver buffer by the kernel.
diskpkt	Disk command/response packets. There are 2 command/response packets (*union un_pkt*, size 22 bytes) for each possible disk.
tapepkt	Tape command/response packets. There are 2 command/response packets for each possible tape.
ttyinpkt	Terminal input command/response packets. There are 2 command/response packets for each possible terminal.
ttyoutpkt	Terminal output command/response packets. There are 2 command/response packets for each possible terminal.
xmsgpkt	Xmsg command/response packets. There is space for 2 command/response packets for each xmsg channel, although only one (total) is used.
sintpkt	Software interrupt command/response packets. There are 2 command/response packets for the software interrupt.
clockpkt	Clock command/response packet.
exitpkt	A single packet of type *struct exit_pkt* for the *feexit()* call. This is 362 bytes.
etinfo	Ethernet interface data and buffer. This area is 3086 bytes for each possible ethernet hardware interface.
bufpool	Disk buffer pool. This area is sized dynamically at startup.
bufhd	Disk buffer headers. This area is sized dynamically at startup.
swapbuf	Swap buffer headers. This area is sized dynamically at startup.

Assuming a maximum of:

256 terminals
16 disks
20 tapes
8 xmsg channels
4 ethernet controllers

the segment layout and addresses are:



3.9. User Text

This segment is mapped onto the current user process text segment (by copying domain information table entries).

† Sized dynamically, dependant on memory configuration.

3.10. Physical Segment Table

This is the ND500(0) PST. The following entries are used by the NDIX kernel:

PST INDEX	Segment
0†	5000 physical address of PST
1	Kernel text
2	Kernel data
3	Kernel stack
4	PST
5	System Tables
6	Physical Memory
7	NDIX Process Segment
8	User process page tables
9	User process shadow page tables
10	Shared segment
11	System Tables (not cached)

Additionally, for each user process, 5 PST slots are required. If n is the domain the user process is running in ($1 \leq n \leq 255$), then the following PST entries are used:

n * 5 + 28	u-area
n * 5 + 29	data
n * 5 + 30	stack
n * 5 + 31	text
n * 5 + 32	process segment (copy)

to map the segments of the user process.

3.11. Process Segment

This segment is the process segment for NDIX. It is set up to allow the mappings described in this document and the interface specification. For domain 0 (the NDIX kernel) these are:

† This entry is needed by the NDIX superstructure and is not usable as a valid PST entry.

data / program	domain	PST	use index
program	0	1	kernel text
program	31	0	other machine
data	0	2	kernel data
data	1	1	kernel text
data	2	6	physical memory
data	3	5	system tables
data	4	8	user page tables
data	5	9	shadow user page tables
data	6	10	shared segment
data	7	11	system tables (no-cache)
data	26	varies	user text
data	27	4	PST
data	28	7	process segment
data	29	varies	kernel stack
data	30	varies	user data
data	31	varies	user stack

in domain 0 only, the privledged instruction allowed bit is set. For other domains, the following capabilities are used:

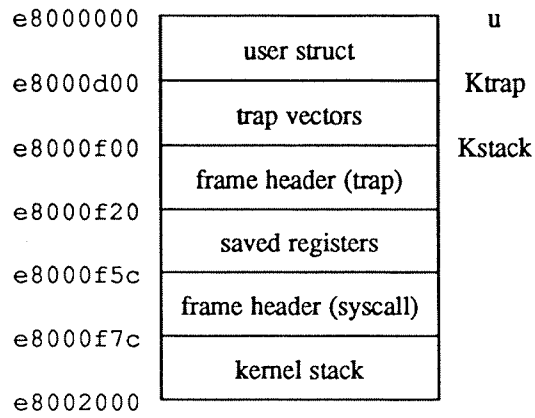
data / program	domain	use
program	0	user text
program	31	indirect domain 0
data	0	user data
data	1	user stack

3.12. Stack (upage)

This segment is used as the kernel's stack segment, and also includes the trap handler vectors and the current user process's *u-area*.

There is one copy of this segment for each process (kernel and user) and the correct segment is selected by copying PST entries.

The layout of the segment is:



3.13. User data

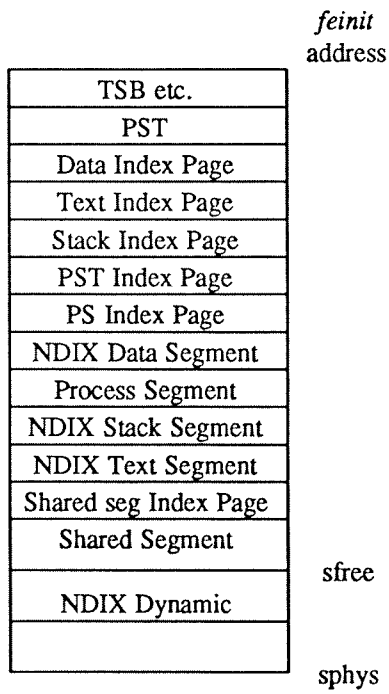
This segment is mapped onto the current user process data segment (by copying PST entries).

3.14. User stack

This segment is mapped onto the current user process stack segment (by copying PST entries).

4. Physical memory layout

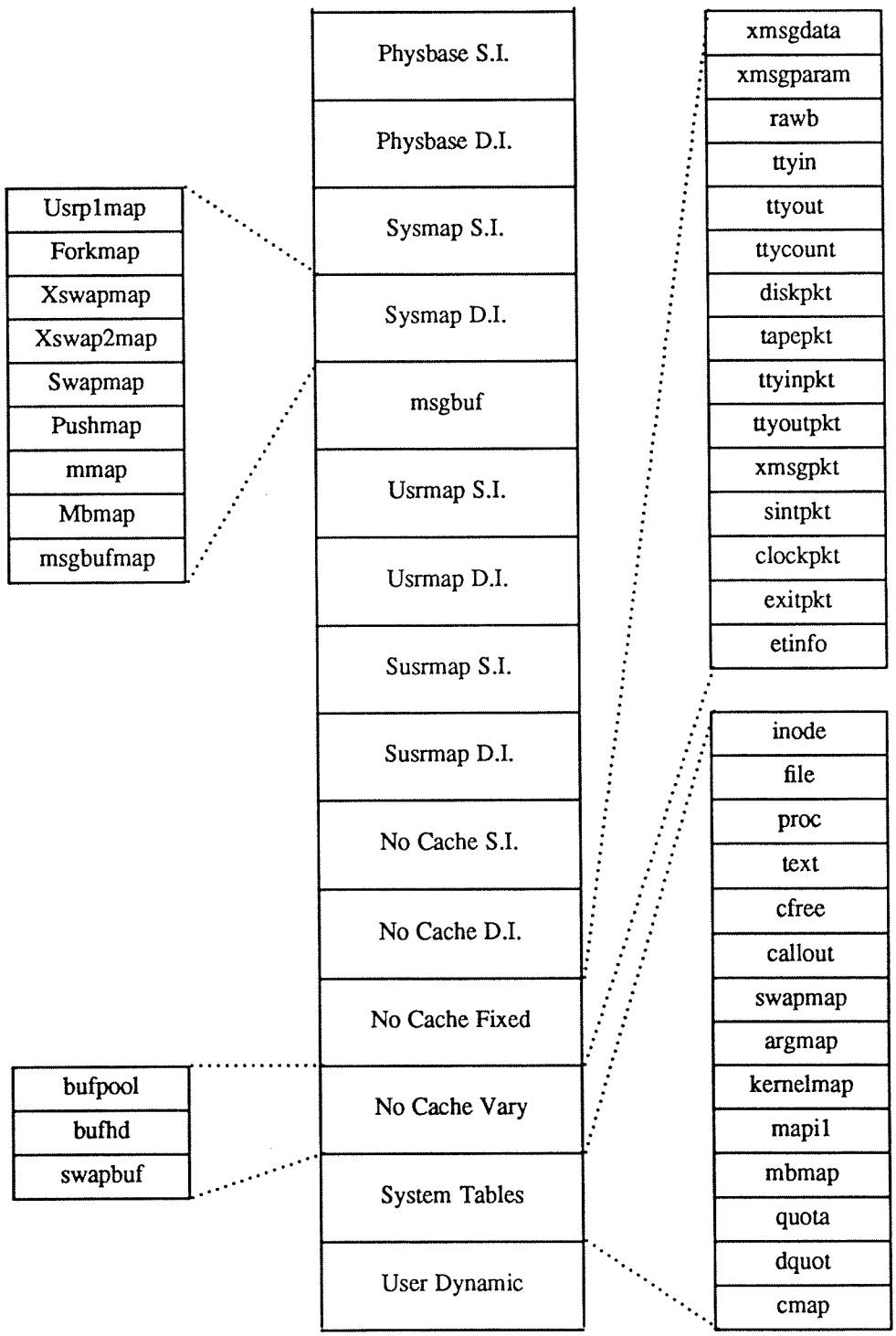
The initial physical memory layout is as described in the interface specification. The diagram is repeated here for convienence.



The area labelled NDIX Dynamic consists of two areas, *Kernel Dynamic* and *User Dynamic*.

5. The Kernel Dynamic area

This area consists of various structures sized and allocated by the NDIX kernel at boot time. The content of this area is:



6. User Dynamic Memory

The user dynamic memory is the area allocatable for user processes and is mapped by the NDIX core map (an array of *cmap* structures), one entry for each pageframe in the user dynamic area. The following types of entry exist in the core map:

6.1. NDIX free

Pages are marked as type *CSYS* in the core map and have the bit *c_free* set if they are not allocated to any NDIX process. Such pages are candidates for the NDIX memory allocators and will have their type changed to one of; *CTEXT*, *CSTACK*, *CDATA*, *CUPAGES* when allocated.

6.2. NDIX system

Pages are marked as type *CSYS* in the core map and have the bit *c_free* clear if they are in use by the NDIX kernel. Such pages are not candidates for the NDIX memory allocators.

6.3. U-area pages

Pages of type *CUPAGES* are not candidates for page-out by the NDIX page daemon and change to type *CSYS* when the associated process terminates or is swapped out.

6.4. NDIX allocated memory

Pages of type *CTEXT*, *CSTACK* and *CDATA* are candidates for page-out and if paged out or their associated process terminates or is swapped out† their type is changed to *CSYS* with *c_free* set.

7. Memory Handcrafting

This section is basically a walk through of *machine/machdep.c:startup()*. It differs from the code in two significant respects, firstly it is readable, and secondly it describes what should be done, not what is done.

When the kernel is started the memory layout in §4 is complete upto the address *sfree*. We make no assumptions about the contents or use of the area labelled *TSB*. There is however an assumption that all the areas loaded by Sintran are in the order shown and contiguous.

We will use the area between *sfree* and *sphys - 1* assuming that it is our memory exclusively, except for those areas whose addresses are given to the driver in *fecall*s. Although *sphys* is assumed to be the start of non-existant memory it is not fatal if this is not strictly true.

The call to *feinit()* gets the values of *sfree* and *sphys* as ND100 addresses so we need to correct them by *private* (the size of the ND100 + PIOC address spaces).

7.1. Physical Memory Mapping

We now have to go through various contortions to get a mapping of physical memory, starting from an empty PST slot.

The kernel domain information table is filled in to map data segment 2 through the PST slot (*PHYSINDEX*) reserved for physical memory mapping. We then set the PST slot to point to the first available page, treating it as a segment of page instead of as page tables. The TSB clear then allows us to address this page.

Physbase is set in *machine/locore.c* to be at `0x1000000`, the KVA of the start of physical memory. Since *Physbase* is declared as an array of characters (*h/systm.h*) the assignment to *pte* is setting *pte* to point at KVA `0x1000000`. This trick is used for the start addresses of many of the kernel segments, but this explanation will not be repeated!

Since physical memory is always more than 1Mb‡ we will always need two levels of indexing to

† all associated processes if *CTEXT*

‡ If it isn't, shoot the salesman.

address the whole of physical memory. Each second level page can map *NPTEPG* (*machine/param.h*) pages of memory so we need *maxmem/NPTEPG* pages of first level indexes, (rounded up of course!).

We fill in the page of memory at *Physbase* with *pte* structures, and the first *mappages* of these *ptes* contain the pageframe numbers of the following (currently unused) pages.

Now the PST entry for the physical memory segment is changed so that it will treat the page as an index page to a single indexed segment instead of as a direct segment. The TSB clear is once again needed so that the new mappings are seen. Now kernel segment 2 appears to be *mappages* long, but not yet initialized.

The next stage is to fill the segment with *ptes* addressing pageframes *0* to *maxmem*.

We then have the (by now familiar) modification to the PST entry so that the segment is now regarded as a double indexed segment, and the usual TSB clear.

The end result is that all the 5000 memory (except page 0) can now be accessed by the NDIX kernel, simply by adding `0x1000000` to the address.

7.2. Debugger Assist.

The kernel debuggers all need to be able to simulate the kernel address mappings and as a start to this process, they need to know where the PST starts in physical memory.

This address is worked out from the PST entry pointing to the PST itself and stored in the kernel variable *pstaddr*. This means that debuggers looking at a running system can find the information they require at a known offset in */dev/kmem* (the address of *pstaddr* since the kernel data segment starts at KVA 0) and then work on */dev/mem*. Alternatively they can work directly on */dev/kmem* and let the running system do the address mappings.

The start address of the PST is also placed in the first location in the PST (which is otherwise unusable). This is the start point of a crash dump, so the debuggers can find this one key value easily from a dump image.

7.3. System Table Mappings.

The system table is used to map various structures that can allocated at potentially changing memory locations. The handcrafting of segment 3 that we do here is only of the page tables that point to these structures, we do not allocate memory for the structures themselves yet.

We know that the page tables will need *SYSMAPPAGES* pages and thus that the segment must be double indexed. The first stage is therefore to take the next available page and fill it in with *ptes* pointing to the following *SYSMAPPAGES*.

The mapping must then be made valid by filling in the kernel domain information table and the PST index appropriately. Once again we flush the TSB so we can actually address the new segment.

Now we come to one of the (initially) incomprehensible parts. In the kernel data segment are a set of *struct pte* *:

Usrpi1map
Forkmap
Xswapmap
Xswap2map
Swapmap
Pushmap
mmap
Mbmap
msgbufmap

which are not initialized at compile time. These are now filled in with physical (in kernel segment 2) addresses of *ptes* in the pages just allocated and mapped. These *ptes* however, are still invalid, they all contain pageframe numbers of 0.

The structures which these *ptes* will point to are declared as *extern* in various header files, and assigned address in kernel segment 3 at the start of *machine/locore.c*. Thus other parts of the kernel can reference these structures via the segment 3 addresses, while memory is allocated as needed by other parts of the kernel which fill in the *ptes*.

An example of this allocation and filling in occurs immediately, where the last four pages of physical memory are allocated to the message buffer (*struct msgbuf msgbuf*) by filling the *ptes* pointed to by *msgbufmap*. The remainder of the kernel then accesses the message buffer via its kernel address (currently `0x1820a800`).

Actually this message buffer allocation is unnecessary, and in the wrong place - it would be far better at the start of physical memory, so it always makes it to disk when a crash dump is taken.

7.4. User Page Tables.

The next allocation is of page tables (and their shadows) for user processes. This is a *Matra* addition, and is only documented in one of their working papers.

Before the page tables can be allocated, they must be sized and since this size is dependant on swap space we must have run *machine/autoconf.c:configure()*. There is (inevitably) a complication. *Configure()* needs to know the root device, which might have to be supplied by the operator, hence we must print the version information to give the operator knowledge of which system he is actually booting.

Just to confuse you even more, the swap configuration is currently hardwired into each kernel, and is independant of the root device (this is a *bug*, *io/di.c:disize()* should behave correctly).

The allocation of the page tables is essentially the same as the allocation of the system table page tables above, except the size of the page tables is not known at compile time. The only problem is that the code for sizing the page tables does not agree with the comment.

7.5. Buffer Sizing

The buffer sizing must be done at this point, since the buffers (currently) reside in no-cache which we must be able to size before we allocate it.

Note that buffer sizing can be done by a user patching the kernel, a technique that might advisable when running heavy disk benchmarks.

7.6. No-Cache Segment.

Handcrafting of the no-cache segment is essentially the same as the user (or shadow page tables) except that we then fill in the page tables to address the pages after the page tables.

The segment consists of two parts, a fixed area of a size known before compilation and then the area needed for the buffer pool and headers. Note that the rounding factor here is only plus 2, since *MAXBSIZE* is a multiple of *NBPG*.

Given the segment size of *ncsize* pages, we will need *ncsize/NPTEPG* second level indexes, so we fill the next available page to point at that number of following pages.

The second level pages are then filled in to point at the following *ncsize* pages.

There is just one small complication left. The xmsg paramater and data areas (*xmsgdata* and *xmsgparam* in §3.8) must be in the same Sintran memory bank. A Sintran memory bank is 128K bytes, so all we do is swap (if necessary) *ptes* in the last level of indexing so that addresses `0x38005000` thru' `0x3800f47f` are on the same bank.

THIS CODE HAS NOT BEEN TESTED YET

7.7. System Data Structures.

A number of system tables are sized at runtime and are potentially user tunable by patching variables initialized in *GENERIC/param.c*.

Historical note: the use of sizing these tables by variables rather than constants was originally introduced to avoid excessive rebuilds of everything when say *NPROC* was changed and as a

consequence *ps*(1) broke. Patching to tune systems followed later, mainly by binary vendors.

Access to these tables is normally via pointers which point to addresses in the physical memory segment, and typically a start and end pointer are both set, although some structures do not require the end pointer.

The structures concerned are:

start	contains	size	end
inode	struct inode	ninode	inodeNINODE
file	struct file	nfile	fileNFILE
proc	struct proc	nproc	procNPROC
text	struct text	ntext	textNTEXT
cfree	struct cblock	nclist	
callout	struct callout	ncallout	
swapmap	struct map	nswapmap = nproc * 2	
argmap	struct map	ARGMAPSIZE	
kernelmap	struct map	nproc	
mapi1	struct map	nproc	
mbmap	struct map	nmbclusters/4	
quota	struct quota	nquota	quotaNQUOTA
dquot	struct dquot	ndquot	dquotNDQUOT
cmap	struct cmap	ncmap	ecmap

The start address of the first of these structures is set to the first available page after the pages used by the no-cache segment. The only tricky point is that before allocation of the *cmap* array, the value of *firstaddr* must be recalculated to prevent the core map mapping these tables. Note also that the core map includes itself in the pages it maps, a fact which is not understood by *analyze*(8).

After clearing all of the area we have just allocated the three pointers associated with the variable part of no-cache are initialized, to addresses in the no-cache segment instead of addresses in the physical segment.

We then make a check to ensure we have some memory left, even though we actually accept an incredibly low amount without panicing.

The callout table (*h/callout.h*) is initialized so that each entry points to the next in the table.

A check is made that we aren't trying to use too many processes (in this version each process has it's own domain) and we start to initialize the *proc* table (*h/proc.h*). The fields set here are an index that will be used later by *sys/vm_mem.c:memall()* when it is passed a pointer to a *proc* structure, the address of the processes pcb (in ND terminology, the domain information table) and an index into the PST where this processes u-area will be mapped.

We then call *sys/vm_mem.c:meminit()* to initialize the core map, and reset "maxmem". This latest resetting of "maxmem" is necessary because we are now working in page clusters rather than pages (however on the ND-5000 the cluster size is 1).

After telling the user how much memory he has to play with and how many buffers we've allocated, we initialize the resource maps.

8. Cache Handling.

The ND5000 has a variety of caches, enabled in differing combinations for different CPUs. This section attempts to explain (briefly) the caches and their impact on NDIX.

8.1. Theory.

8.1.1. Translation Speedup Buffer.

The Translation Speedup Buffer (TSB) is a cache of logical to physical address *mappings*. A TSB lookup is a function of the domain number, contents of the *PS* register and part of the logical address.

There are two *TSB*s, one for instruction space and one for data space. They are cleared by the instructions:

dctsb clear data TSB
pctsb clear text TSB

8.1.2. Memory Caches.

The caches hold logical address *contents* and are checked before the TSB. The cache lookup is a function the domain number, contents of the *PS* register and part of the logical address.

There are two caches, one instruction and one data. They are cleared by the instructions:

dcc clear data cache
pcc clear text cache

The data cache can function in one of two modes, either *write-through* or *write-in cache-only* (dirty cache). In write through mode all writes to the cache are also written to memory but in dirty mode the data is not written to memory until a special instruction is executed.

The instruction to flush a dirty cache to memory is:

ddirt

8.1.3. PS Register.

Since the cache and *TSB* lookups index on domain and PS register it is possible to avoid clears by careful use of the PS register.

The saving is not when executing a user process, since they each have unique domain numbers, but when the kernel is executing on behalf of a user process. In this case if the kernel uses a different value in the PS register depending on the domain it was called from then clears are not necessary as the kernel context switches in domain 0 between running for different user level processes.

The PS register contains the PST index of the page table for the *Process Segment* so just loading with arbitary values is not possible. Instead, a slot in the PST is reserved for use as a *process segment* for each domain.

THIS CODE HAS NOT BEEN TESTED YET.

8.2. Range Differences.

There are three different ND5000 processors, although only two are used for NDIX systems. The type 1 CPU is used only for the ND5200 and has no cache hardware. The type 2 CPU has cache hardware and is used for the ND5400 through ND5700. The type 3 CPU has an additional *booster card* for some address mode calculations, and used in the ND5800.

NDIX is not supplied to run on type 1 CPUs, and type 2 and 3 appear identical in terms of cache functions.

The range performance is controlled by use of different microcodes which enable different caches. The differences between CPU models are documented in the ND-5000 Hardware Description manual on page 13. The only point to note is that the *address cache* mentioned there is a subset of the instruction cache and is transparent above the microcode level.

8.3. In Practice.

The NDIX kernel requires to handle the caches for cases that are both obvious and non-obvious. The obvious cases are when user processes are loaded, unloaded and mapped into the kernel memory space†.

The non-obvious cases arise when the kernel is manipulating data in user segments, and also because of the inter-dependancy of the TSB and caches.

8.3.1. Cache Inter-depenancy.

Since a cache clear marks the whole of the cache invalid and forces future references to physical memory it is cleary necessary for a *dcc* to also carry out a *ddirt* to get the cache contents back to memory.

A *TSB* clear implies that new address mappings have come into effect. Since these new mappings are governed by page tables the *TSB* flushes also flush the caches.

Something not quite right here - a *pctsb* does a *pcc*, but the page tables are in **data**.

8.3.2. Why the Kernel Clears Caches.

Consider the following scenarios for examples of why the kernel needs to manipulate the caches. Remember that a user data segment is accessed via two sets of KVAs, one in the user's domain and one in the kernel domain.

8.3.2.1. Write-Through-Cache.

- The user writes to memory.
- The data is written into the cache and into physical memory.
- The kernel writes to the same location.
- The data is written into the cache and into physical memory.
- But, the two cache writes are at different cache locations since the cache address is a function of domain number.
- The user reads back the contents of the location.
- The read hits the cache and the value the user process wrote to memory is returned.
- Physical memory however contains the value the kernel wrote.

8.3.2.2. Dirty Cache.

- A user process writes to memory.
- The data is written to cache, but not to memory.
- The kernel reads the data.
- The kernel gets a cache miss and accesses physical memory.
- The kernel gets wrong data because the correct data is only in the cache.

8.3.3. What Is There Now.

The current cache handling is there because it works, not because it is what we believe should be there.

The basic starting point was the Release B cache handling which worked on systems with write-through cache. Initial Release C work was done with cache disabled but *TSB* enabled. We can therefore be reasonably confident that all the required *TSB* handling is in place.

---

† If this is not obvious you should not be reading this document. By the way I know of a tower in Paris they're thinking of junking that I can let you have for a song.

The next stage was to enable the instruction cache, and such systems continued to work. This left the remaining work as:

- handle data cache
- handle dirty cache
- remove redundant *TSB* and cache clears

Cache clears were added until such time as the system could run heavy user loads with a *write-through* cache.

Three types of failure were observed:

a) Kernel Page Faults

b) Invalid P registers

c) Invalid context blocks

The first type of fault was consistent and was initially cleared by place cache clears before the faulting instructions. This left the other two types of faults which appeared to be timing related.

Type b faults happened after an *splx()* call caused a delayed interrupt to be serviced. This fault stopped occuring after cache clears were placed at the start of the interrupt handler. *This fix is almost certainly wrong - see the code for more comments.*

Type c faults appeared to occur after *trap()* had slept and then returned. Various cache clears in the trap code reduced the ocurrance of this problem, a final (and sufficent) fix is a cache clear at the system call entry.

The cache clear at system call entry also allowed the removal of all the clears applied to fix type a problems.

9. No-cache Segment.

The use of cache as outlined above is not sensible for data used both by the ND100 and ND5000 since all accesses would have to preceeded by *ddirt*ies or *dccs* to ensure both CPUs and all possible KVAs saw consistent, correct data.

Such data is kept in segments that have their capabilities marked as shared, and are therefore never cached. The current system contains two such segments one handcrafted by the ND100, and one handcrafted by the ND5000. Allocation of structures to these segements is dependant on when the structures can be sized and if there addresses are needed before the kernel has hand crafted the second of these segments.

This section discusses the second of these segments, starting at KVA `0x38000000`.

Firstly, the fixed allocation of addresses is wrong, but a truely dynamic allocation is not possible because the pre-processor cannot use *sizeof* directives.

Secondly, the allocation of the XMSG areas, and the juggling to provide a single bank for these areas is confusing and potentially wrong. It will certainly be hard to clear addressing problems in this area. Additionally, the swapping of *ptes* could fragment a structure that would be better contiguous.

Thirdly, the presence of the buffer cache in un-cached memory will degrade the effectivness of this cache. Tests should be run to find out if it is better to have the disk buffers in cached memory and use *ddirt* instructions rather than the current layout.

Looking ahead to DMA to peripherals it is likely that transfers should be triggered out of their own contiguous address spaces rather than collected by copying. Such an approach would certainly pay for the *pagedaemon*.

10. .set Directives.

The use of the *.set* directive is potentially confusing, and is in the interests of completness, explained here.

The compiler can define a variable, which reserves space for the variable or declare a variable, which implies space will be allocated elsewhere.

However, the compiler (in conjunction with the linker) can only allocate contiguous areas starting at location zero. This is clearly not adequate for the kernel, where large numbers of structures must be allocated at fixed locations, and outside of the area the compiler would use.

The solution is for the variables to be declared *extern* wherever they are referenced, leaving the addresses to be found at link time. In *machine/locore.c* all such variables have a *.set* declaration which results in variable being defined as absolute with the value supplied in the directive.

At link time when an address is required for a variable that is always defined as external except for an absolute declaration, the address used is the value supplied in the *.set* directive.

The *.set* directive should therefore be ragarded as a method of telling the compiler the start address of a variable, instead of letting it asign an arbitary address.

Table of Contents