Today: Minix Memory Management

•

Process and Memory Mgmt

- Memory management is tied to process management
  - Processes are allocated memory and use memory
- Minix functionality has evolved over time in terms of memory management support

Pre-3.1 Minix

- Process mgr (PM) included all memory management
  - One server handled processes and memory
- Segmented view of process
  - each process has text, stack, heap segment
- Segments are allocated contiguously in RAM
  - Memory is treated as a collection of holes and allocated segments
    - uses first-fit and best-fit policies for allocation
- No Paging, no virtual memory/demand paging

Pre-3.1 Minix

- Works well for older CPUs or embedded processors
  - CPUs lack support for paging
- Entire processes must be resident in RAM
- Fragmentation can occur

post-3.2 Minix

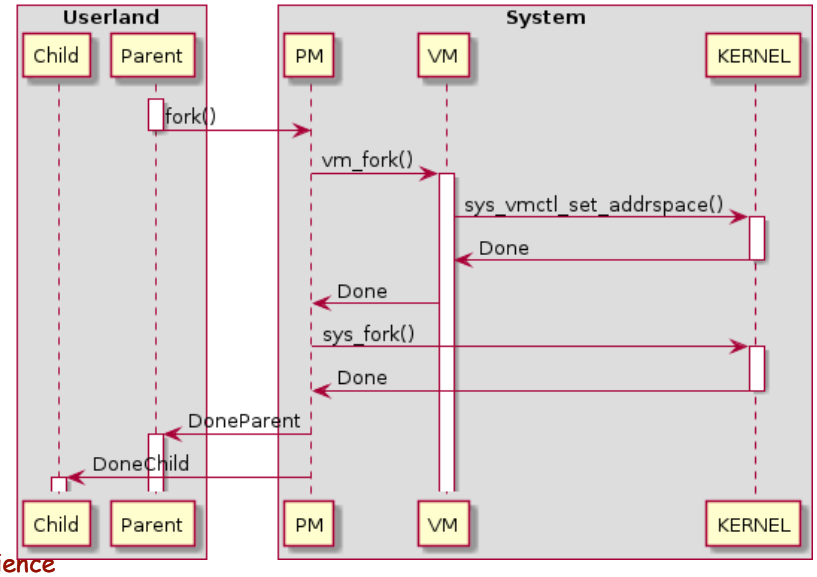
- Version 3.2.1 (used for our class), ver 3.3
- Introduced VM (virtual memory) server
  - Memory mgmt code moved from PM to VM
- Introduced support for paging and virtual memory
- Support modern CPU with support for paging
- Processes are a collection of segments
- Process segments laid out contiguously in virtual rem
  - No contiguous assumption for physical memory
  - Segments are paged; only subset of pages in physical RAM
- Segmented Paging with Virtual Memory

Proc and Memory Mgmt

- Discussion of VM (see Minix3 documentation)
- Discussion of PM as it related to memory management

Typical PM-VM Interaction

- Fork creates a new process, which requires memory to be allocated for the new process

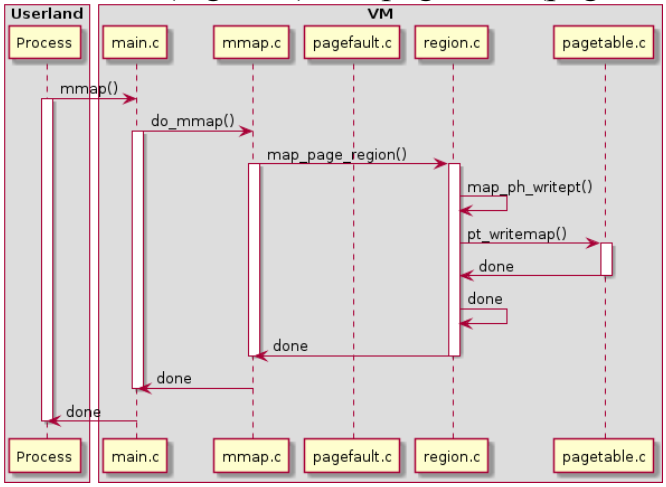


VM Server

- Goals: track used, unused memory, allocate memory to process, free memory.
  - Manage virtual memory
- Contains **arch dependent** and **independent** code
- Virtual Region (*struct vir_region* or *region_t*)
  - contiguous range of virtual address space
- Physical Region *struct phys_region*
  - physical blocks (pages) of memory
  - ref count, to track how many times the block is referenced
- Disk cache (holds pages of disk blocks)

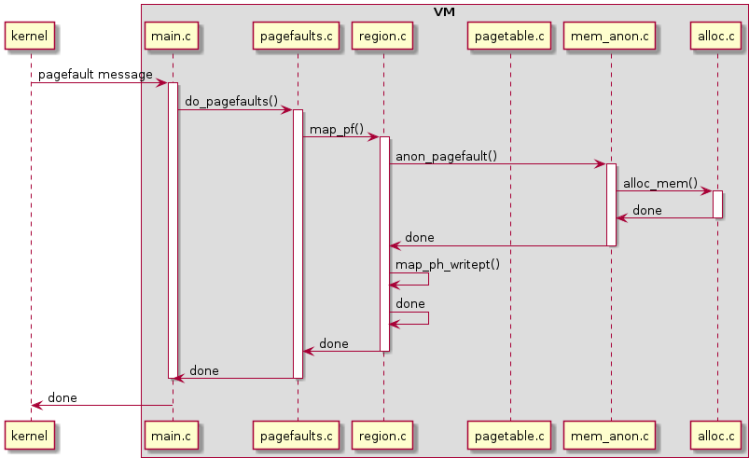
Typical Call Structure

- Calls to VM come from userland, PM, kernel
- Typical call handling
  - Receive call in main.c
  - Call-specific work in call-specific file: mmap.c, cache.c
  - Update data structures (region.c) and pagetable (pagetable.c)



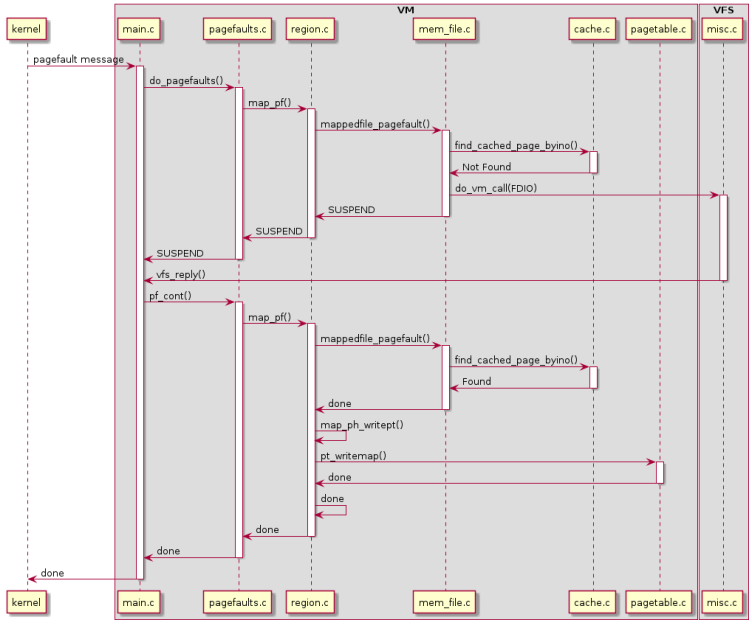
Handling Absent Memory

- Minix 3.2.1+ uses virtual memory: needed memory may not be present in RAM
- Pagefault handling for anonymous memory



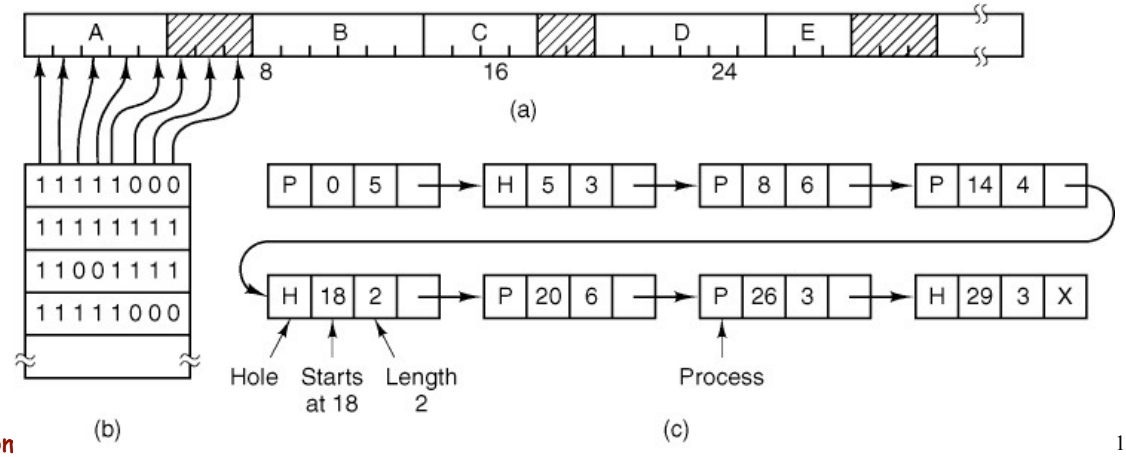
Page faults

- Page faults can occur for file-mapped region
  - Query cache else go to VFS



Using Bitmaps and Linked Lists

A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

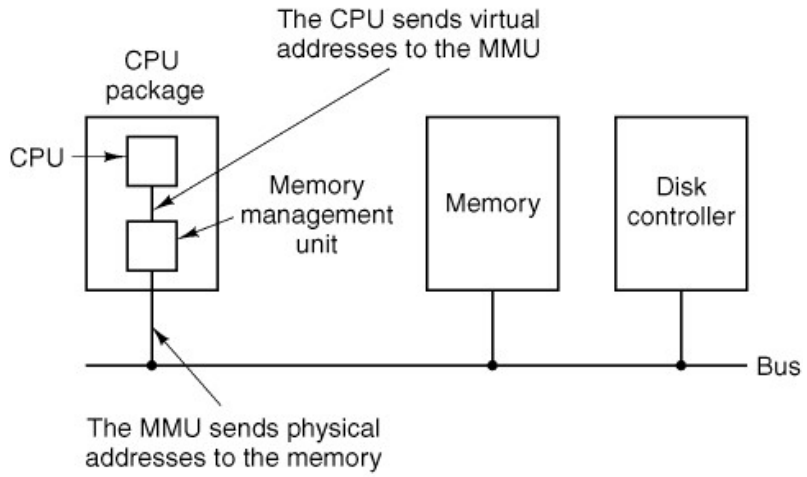


Memory Allocation Algorithms

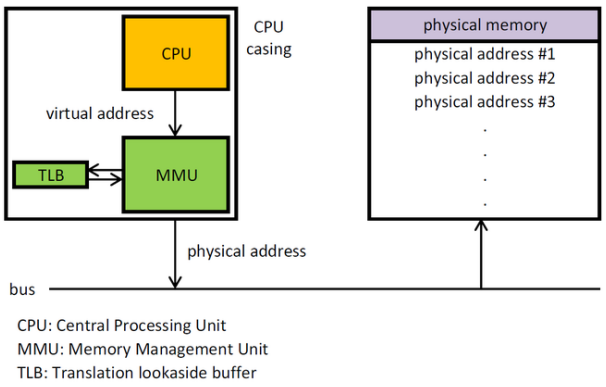
- First fit Use first hole big enough
- Next fit Use next hole big enough
- Best fit Search list for smallest hole big enough
- Worst fit Search list for largest hole available
- Quick fit Separate lists of commonly requested sizes

- Early Minix used these method for physical mem alloc
- Later Minix versions uses holes and allocation for allocating a process in virtual memory

Paging needs a MMU

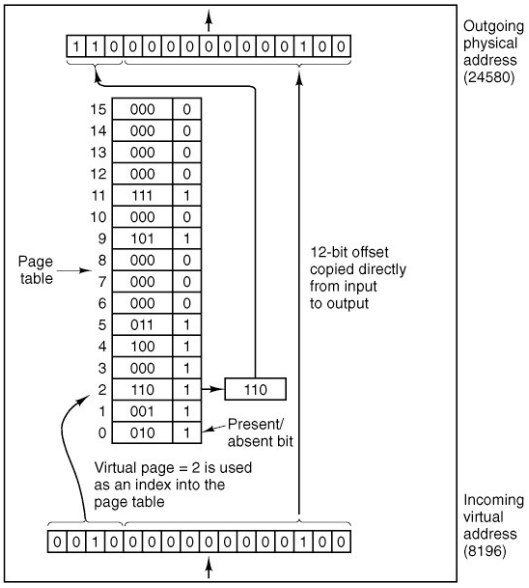


Example MMU



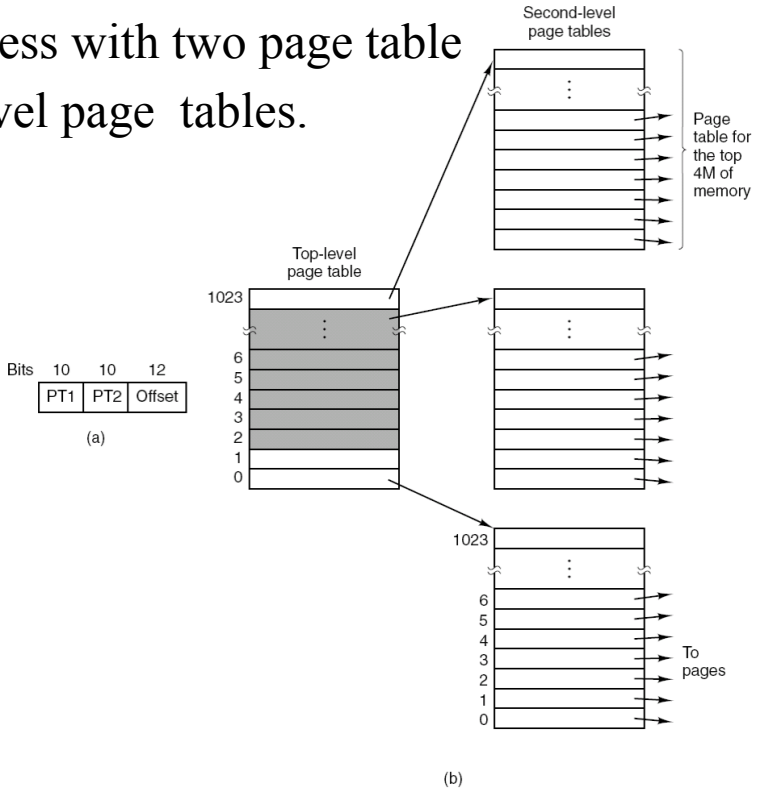
Paging using a MMU

The internal operation of the MMU with 16 4-KB pages.



Multilevel Page

(a) A 32-bit address with two page table fields. (b) Two-level page tables.

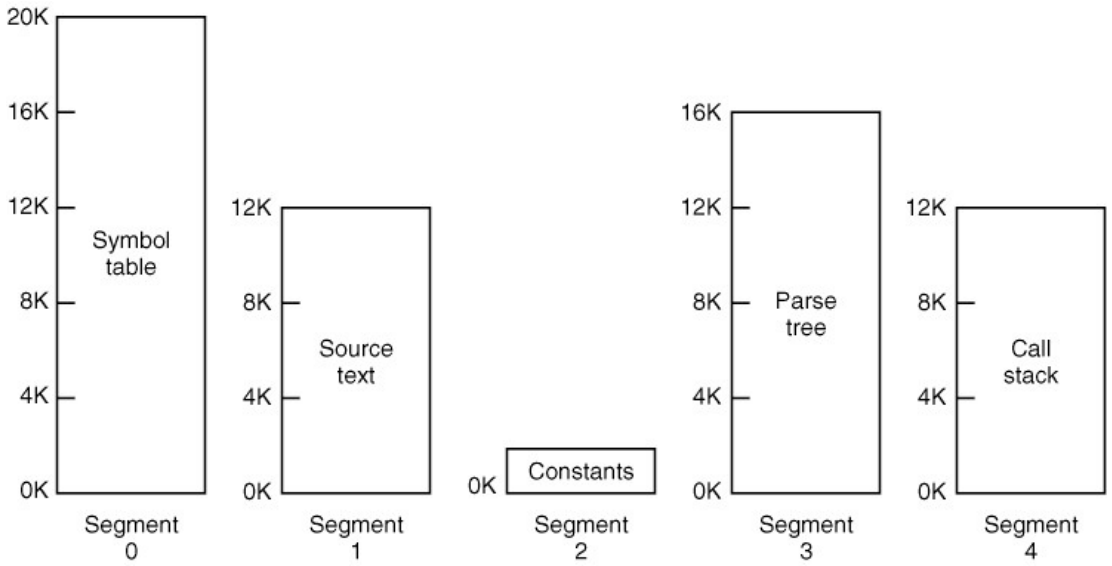


TLBs—Translation Lookaside Buffers

A TLB to speed up paging.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Segmentation



Segmentation vs Paging

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes

. . .

Segmentation vs Paging

Consideration	Paging	Segmentation
		. . .
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Segmentation with Paging:

Intel Pentium (and later) virtual memory support

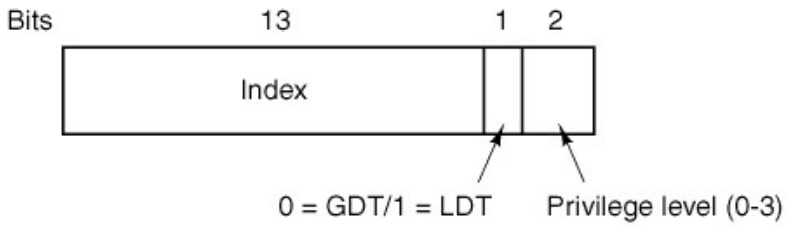
- Local Descriptor Table (LDT)
- Global Descriptor Table (GDT)

Each process has its own LDT; one GDT shared by all

LDT: segments local to a process

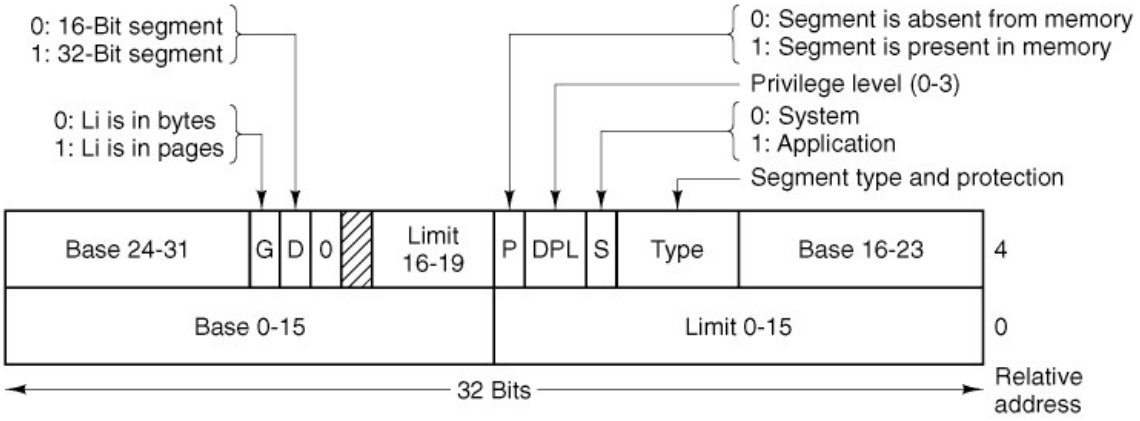
GDT: system segments + OS segments

Figure: Segment Selector



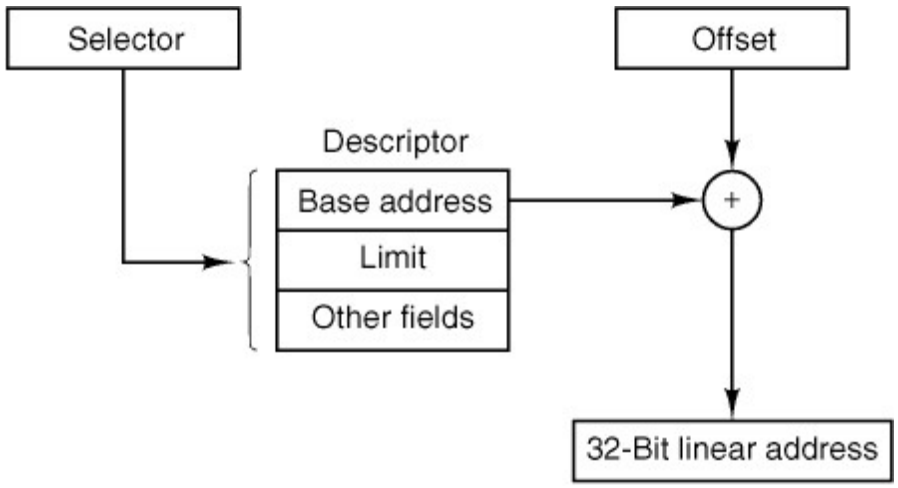
Segmentation with Paging:

Pentium code segment descriptor.
Data segments differ slightly.



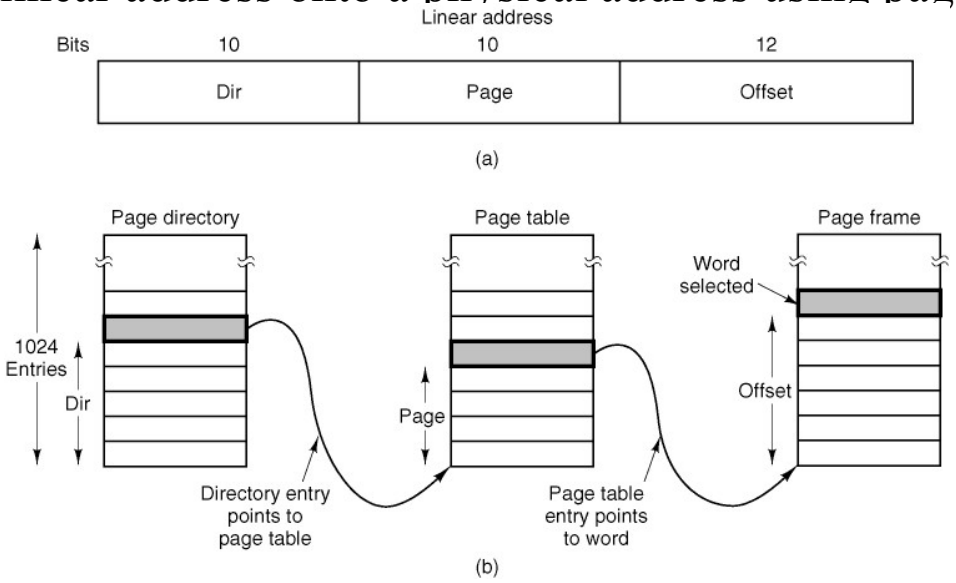
Segmentation with Paging:

Conversion of a (selector, offset)
pair to a linear address.



Segmentation with Paging:

Since segments are paged, next, we map the 32-bit linear address onto a physical address using page tables



Process Manager Data Structures

The message types, input parameters, and reply values used for communicating with the PM.

Message type	Input parameters	Reply value
fork	(none)	Child's PID, (to child: 0)
exit	Exit status	(No reply if successful)
wait	(none)	Status
waitpid	Process identifier and flags	Status
brk	New size	New size
exec	Pointer to initial stack	(No reply if successful)
kill	Process identifier and signal	Status
alarm	Number of seconds to wait	Residual time
pause	(none)	(No reply if successful)
sigaction	Signal number, action, old action	Status
sigsuspend	Signal mask	(No reply if successful)
sigpending	(none)	Status
sigprocmask	How, set, old set	Status
sigreturn	Context	Status
getuid	(none)	Uid, effective uid
getgid	(none)	Gid, effective gid
getpid	(none)	PID, parent PID

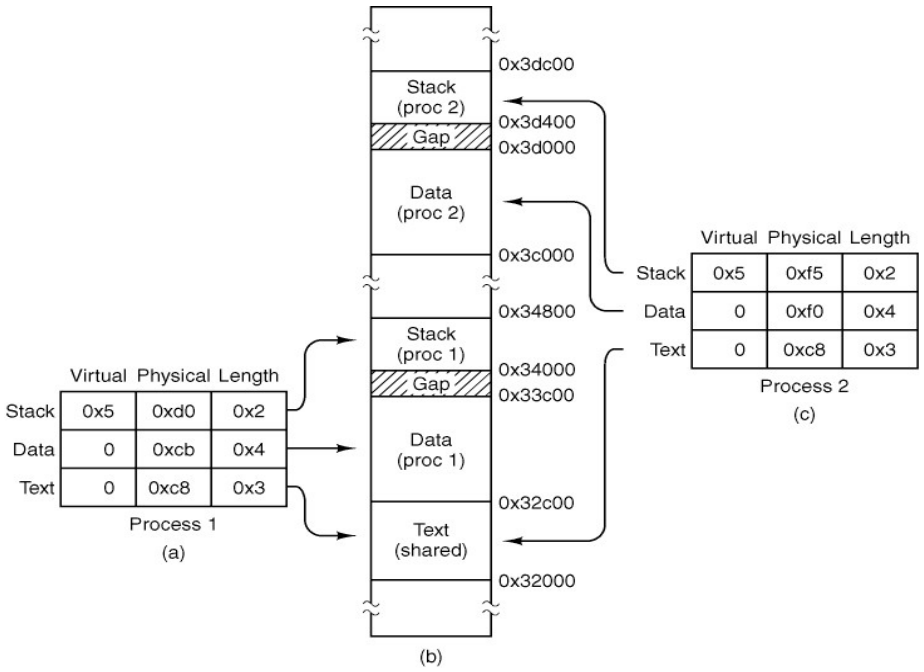
. . .

Process Manager Data Structures

The message types, input parameters, and reply values used for communicating with the PM.

Message type	Input parameters	Reply value
setuid	New uid	Status
setgid	New gid	Status
setsid	New sid	Process group
getpgrp	New gid	Process group
time	Pointer to place where current time goes	Status
stime	Pointer to current time	Status
times	Pointer to buffer for process and child times	Uptime since boot
ptrace	Request, PID, address, data	Status
reboot	How (halt, reboot, or panic)	(No reply if successful)
svrctl	Request, data (depends upon function)	Status
getsysinfo	Request, data (depends upon function)	Status
getprocnr	(none)	Proc number
memalloc	Size, pointer to address	Status
memfree	Size, address	Status
getpriority	Pid, type, value	Priority (nice value)
setpriority	Pid, type, value	Priority (nice value)
gettimeofday	(none)	Time, uptime

Sharing Text Segments



The Hole List

The hole list is an array of struct hole.

```
PRIVATE struct hole {
        struct hole *h_next;        /* pointer to next entry on the list */
        phys_clicks h_base;         /* where does the hole begin? */
        phys_clicks h_len;          /* how big is the hole? */
} hole[NR_HOLES];
```


FORK System Call

1. Check to see if process table is full.
2. Try to allocate memory for the child's data and stack.
3. Copy the parent's data and stack to the child's memory.
4. Find a free process slot and copy parent's slot to it.
5. Enter child's memory map in process table.
6. Choose a PID for the child.
7. Tell kernel and file system about child.
8. Report child's memory map to kernel.
9. Send reply messages to parent and child.

EXEC System Call (1)

1. Check permissions—is the file executable?
2. Read the header to get the segment and total sizes.
3. Fetch the arguments and environment from the caller.
4. Allocate new memory and release unneeded old memory.
5. Copy stack to new memory image.
6. Copy data (and possibly text) segment to new memory image.
7. Check for and handle setuid, setgid bits.
8. Fix up process table entry.
9. Tell kernel that process is now runnable.

Other System Calls in PM

Three system calls involving time.

Call	Function
time	Get current real time and uptime in seconds
stime	Set the real time clock
times	Get the process accounting times

Other System Calls in PM

Figure 4-51. The system calls supported in *servers/pm/getset.c.*

System Call	Description
getuid	Return real and effective UID
getgid	Return real and effective GID
getpid	Return PIDs of process and its parent
setuid	Set caller's real and effective UID
setgid	Set caller's real and effective GID
setsid	Create new session, return PID
getpgrp	Return ID of process group

Other System Calls in PM

Figure 4-52. Special-purpose MINIX 3 system calls in *servers/pm/misc.c.*

System Call	Description
do_allocmem	Allocate a chunk of memory
do_freemem	Deallocate a chunk of memory
do_getsysinfo	Get info about PM from kernel
do_getprocnr	Get index to proc table from PID or name
do_reboot	Kill all processes, tell FS and kernel
do_getsetpriority	Get or set system priority
do_svrctrl	Make a process into a server

Other System Calls (4)

Debugging commands supported by *servers/pm/trace.c.*

Command	Description
T_STOP	Stop the process
T_OK	Enable tracing by parent for this process
T_GETINS	Return value from text (instruction) space
T_GETDATA	Return value from data space
T_GETUSER	Return value from user process table
T_SETINS	Set value in instruction space
T_SETDATA	Set value in data space
T_SETUSER	Set value in user process table
T_RESUME	Resume execution
T_EXIT	Exit
T_STEP	Set trace bit

Memory Management Utilities

Three entry points of alloc.c

1. *alloc_mem* – request a block of memory of given size
2. *free_mem* – return memory that is no longer needed
3. *mem_init* – initialize free list when PM starts running