4

MEMORY MANAGEMENT

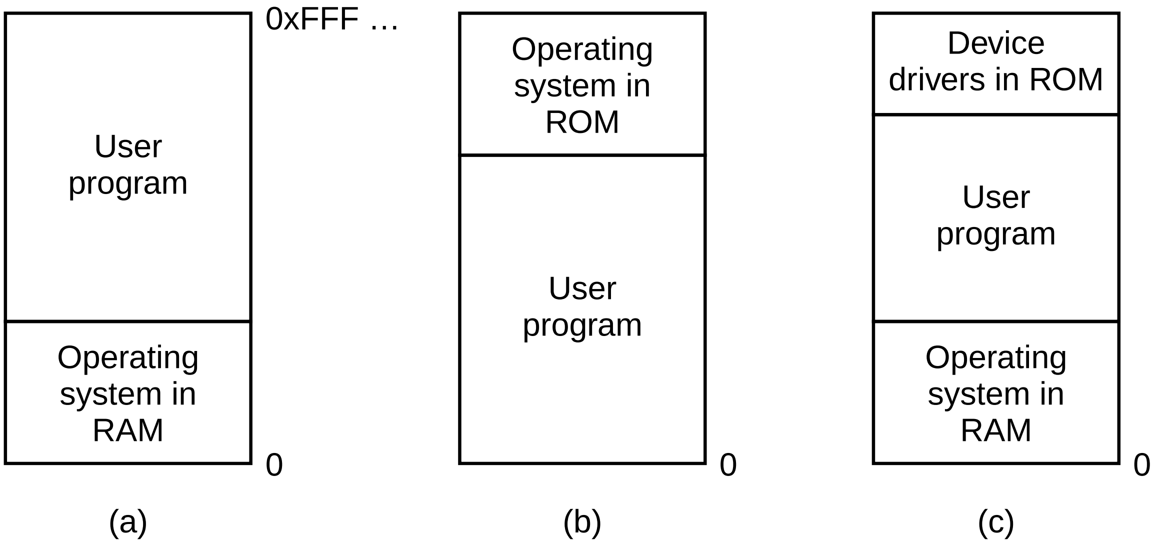


Figure 4-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

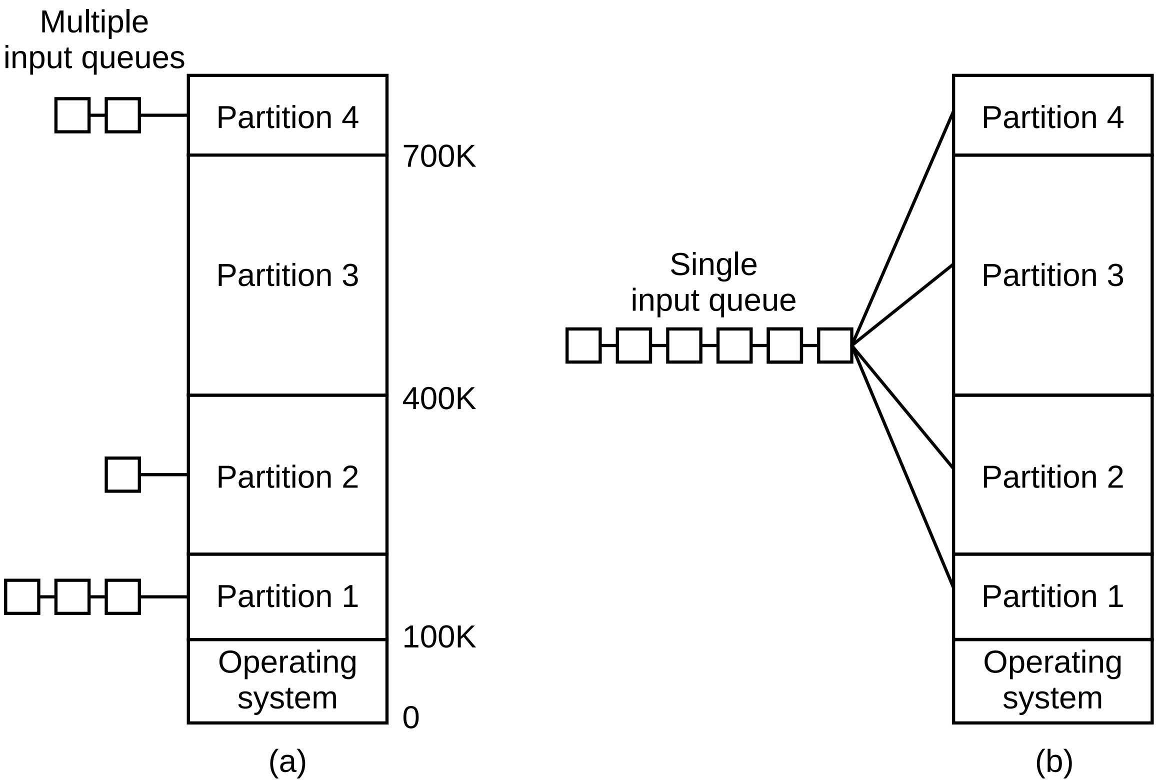


Figure 4-2. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

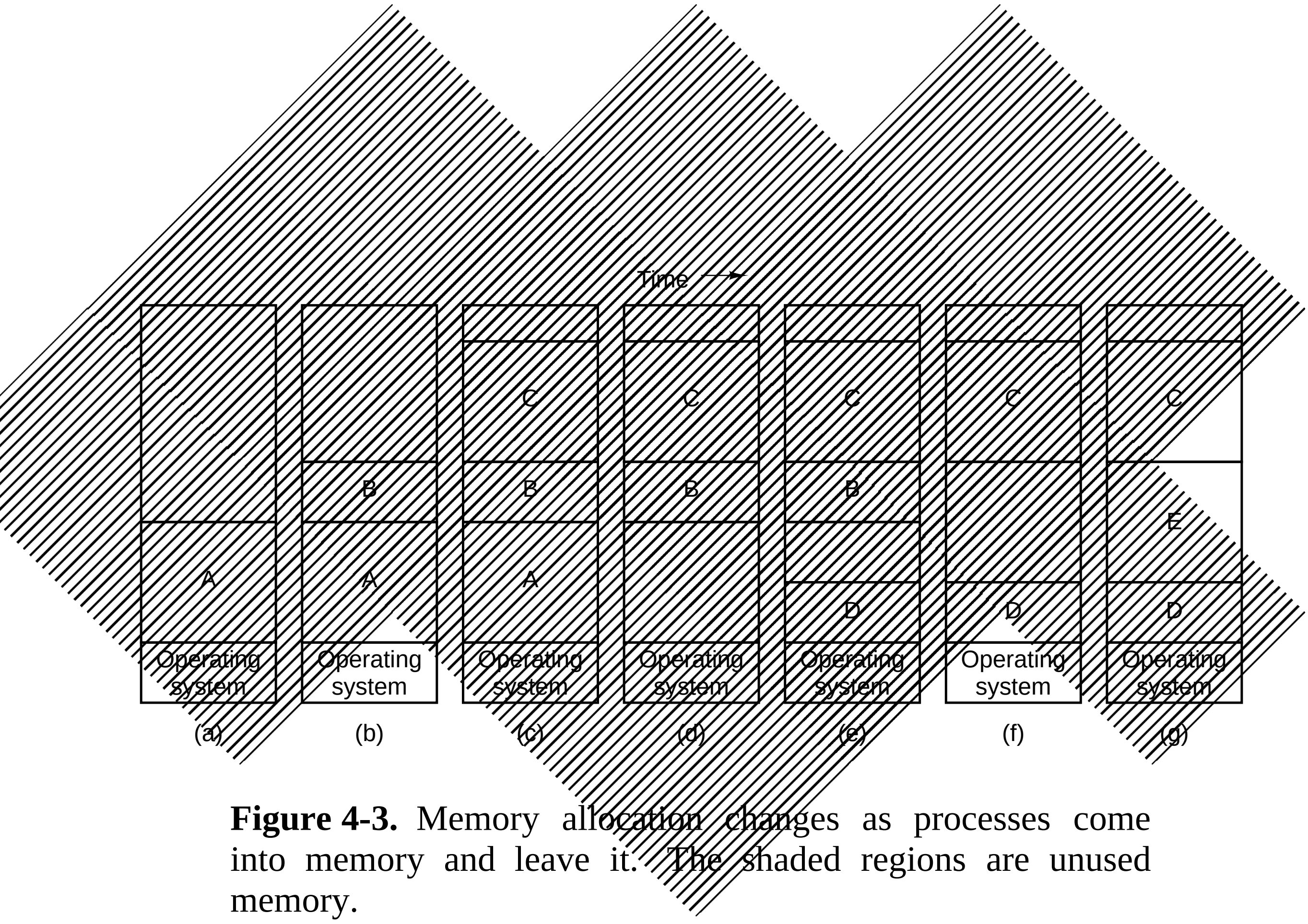


Figure 4-3. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

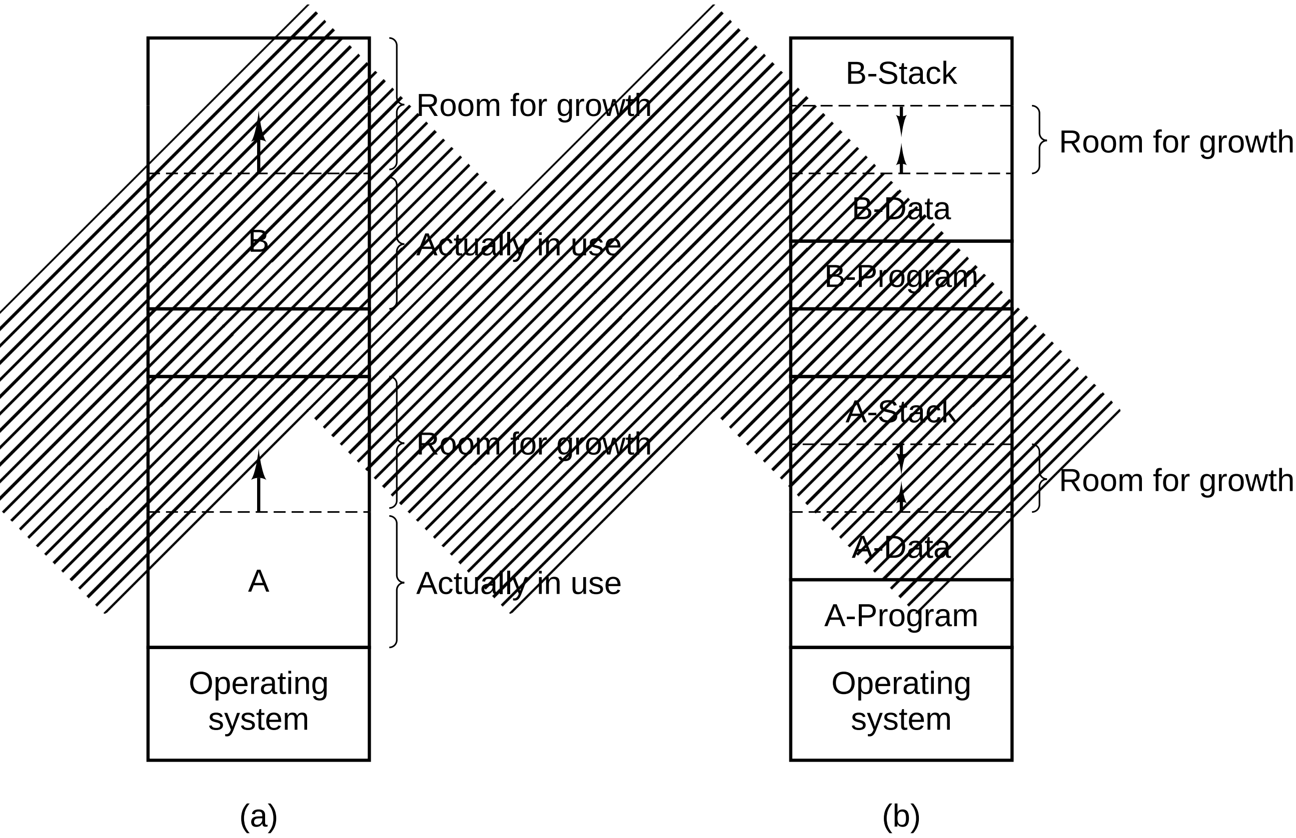


Figure 4-4. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

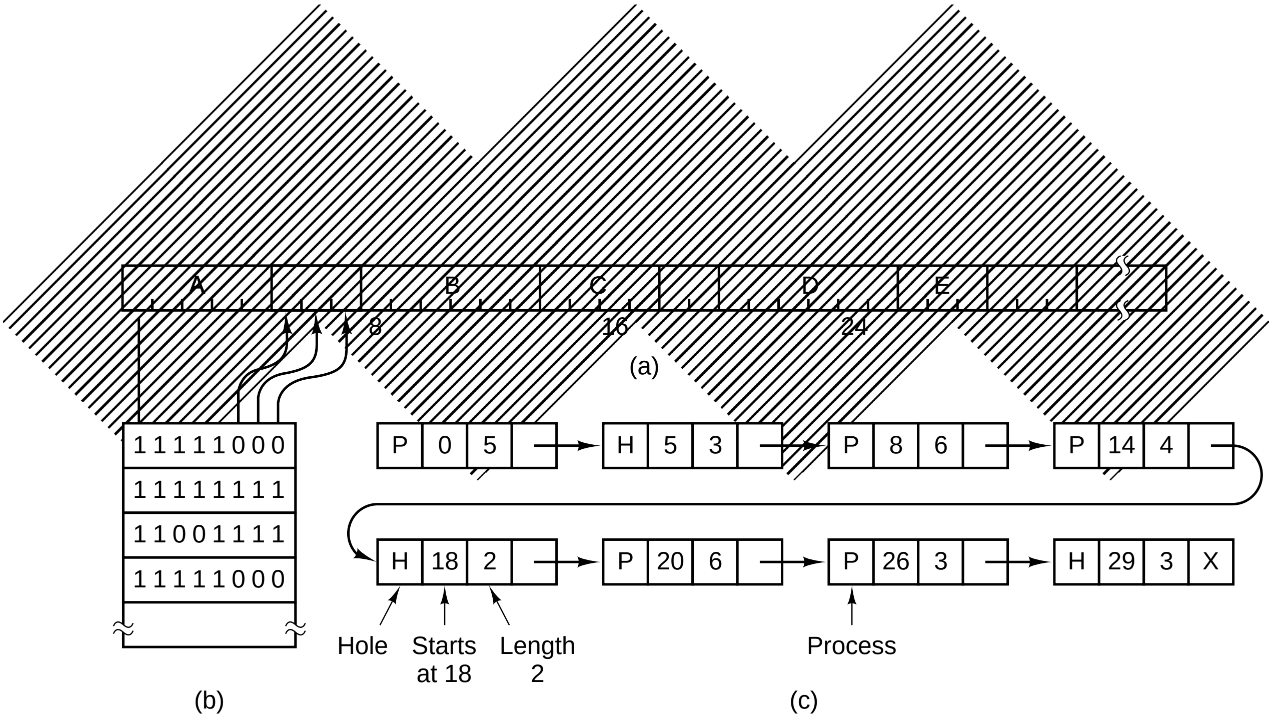


Figure 4-5. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bit map) are free. (b) The corresponding bit map. (c) The same information as a list.

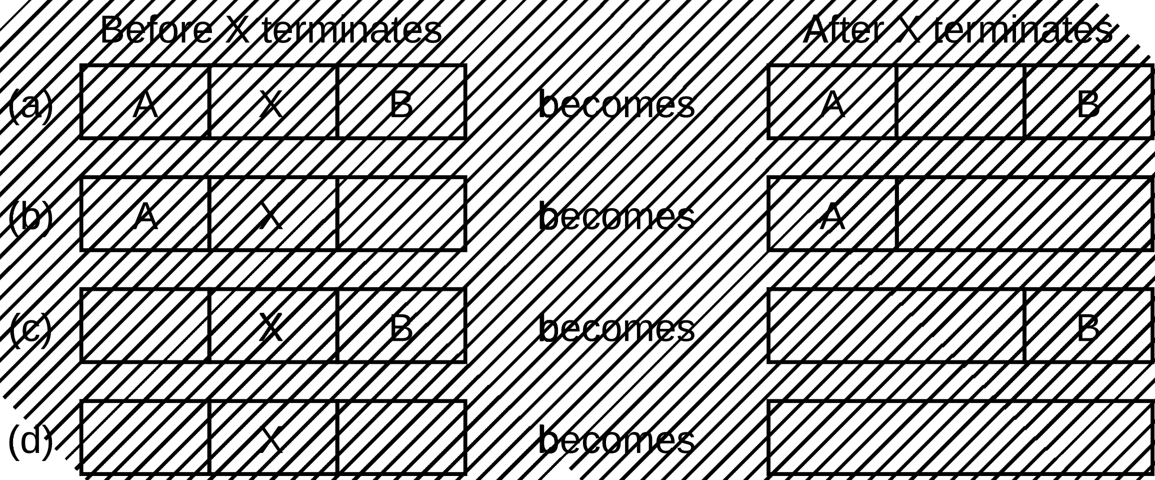


Figure 4-6. Four neighbor combinations for the terminating process, *X*.

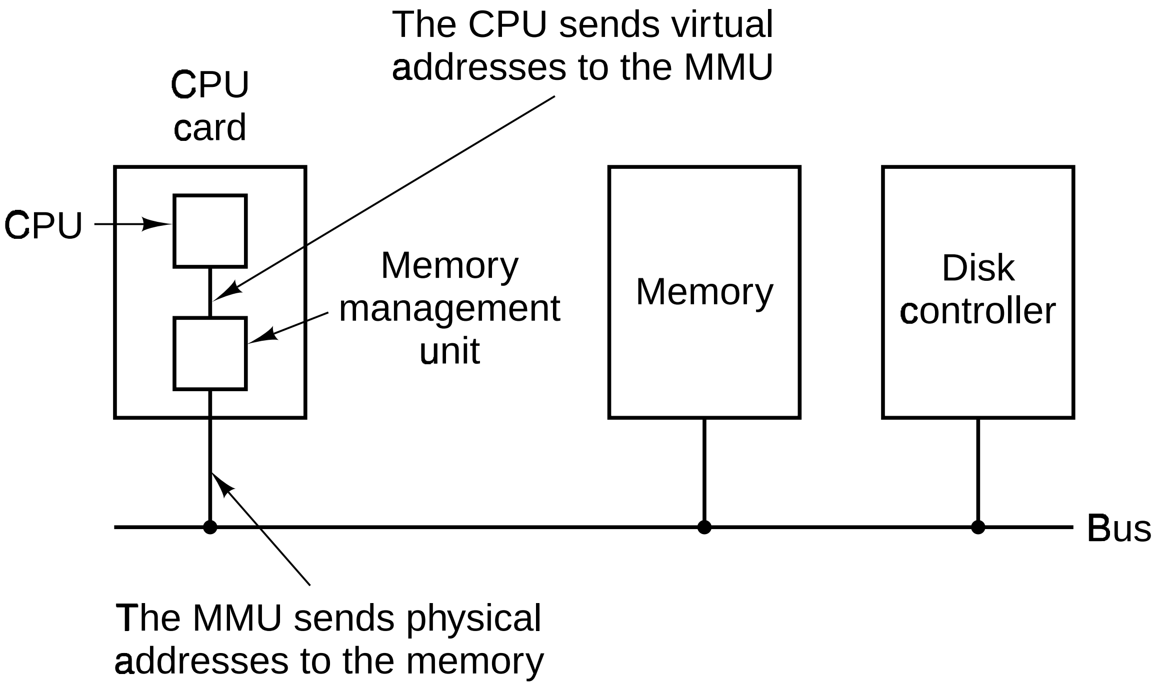


Figure 4-7. The position and function of the MMU.

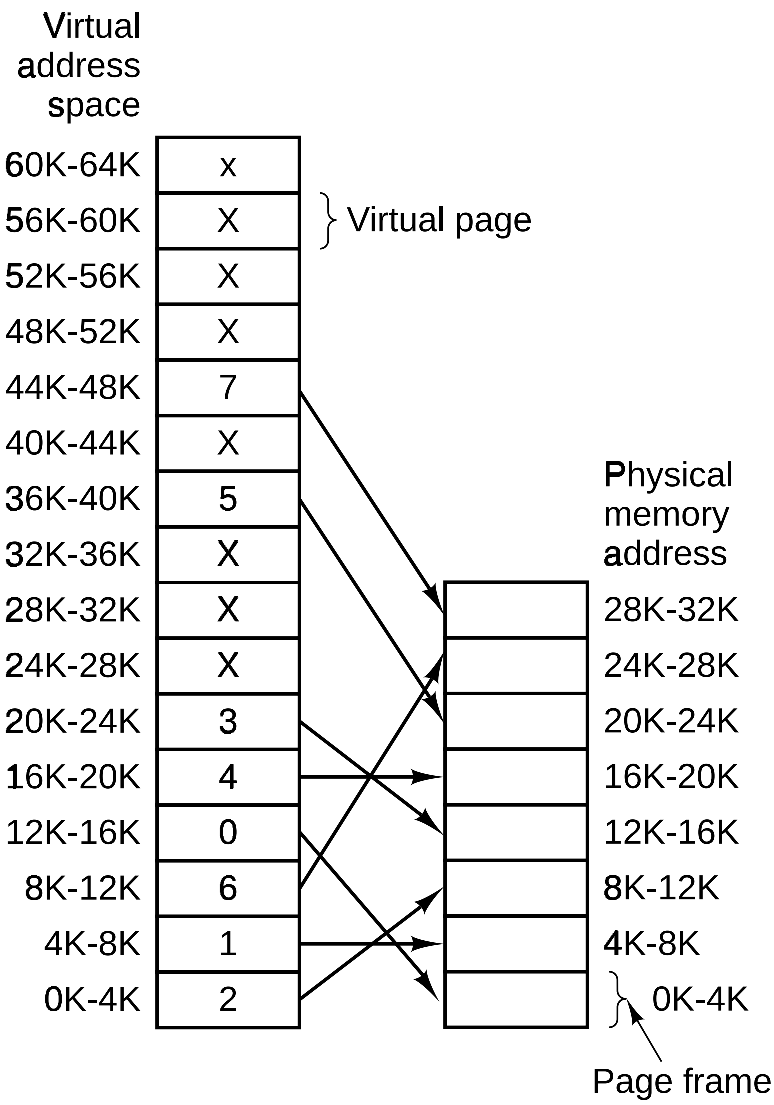


Figure 4-8. The relation between virtual addresses and physical memory addresses is given by the page table.

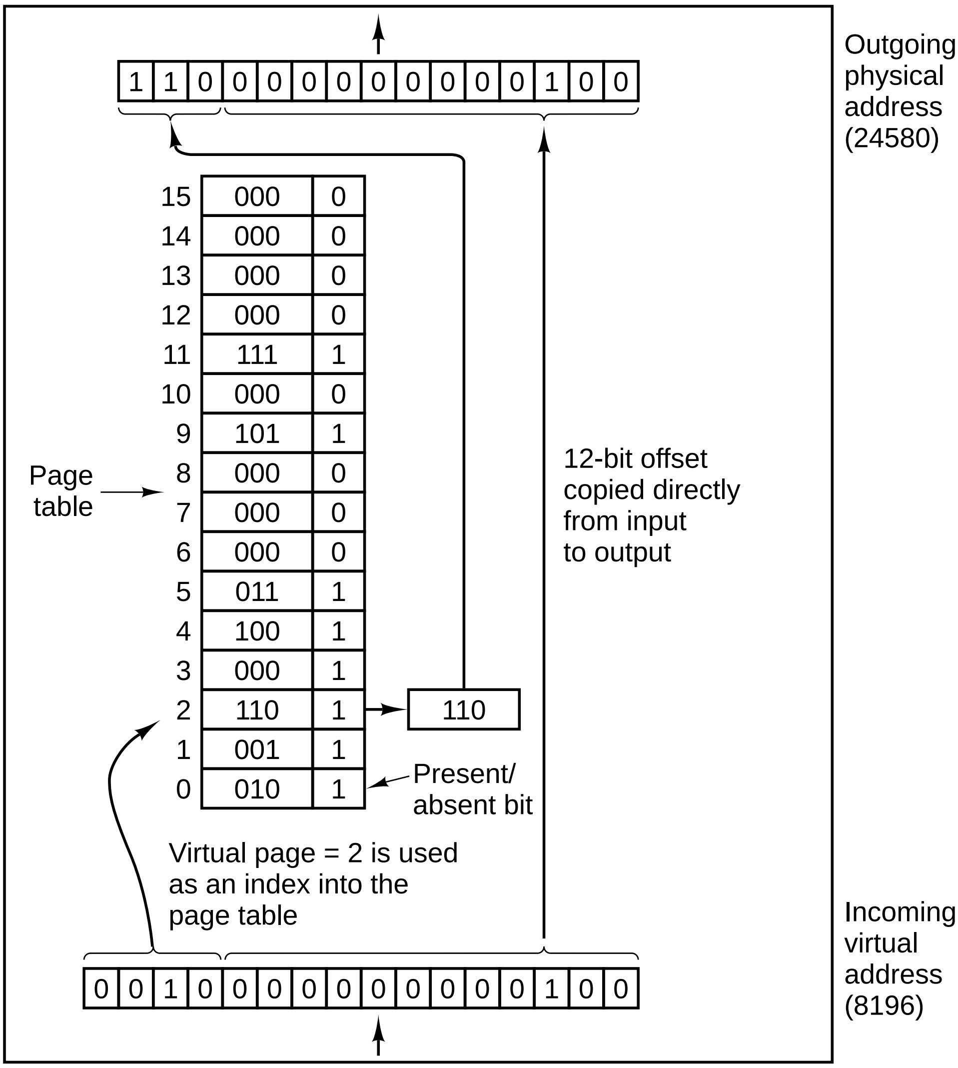


Figure 4-9. The internal operation of the MMU with 16 4K pages.

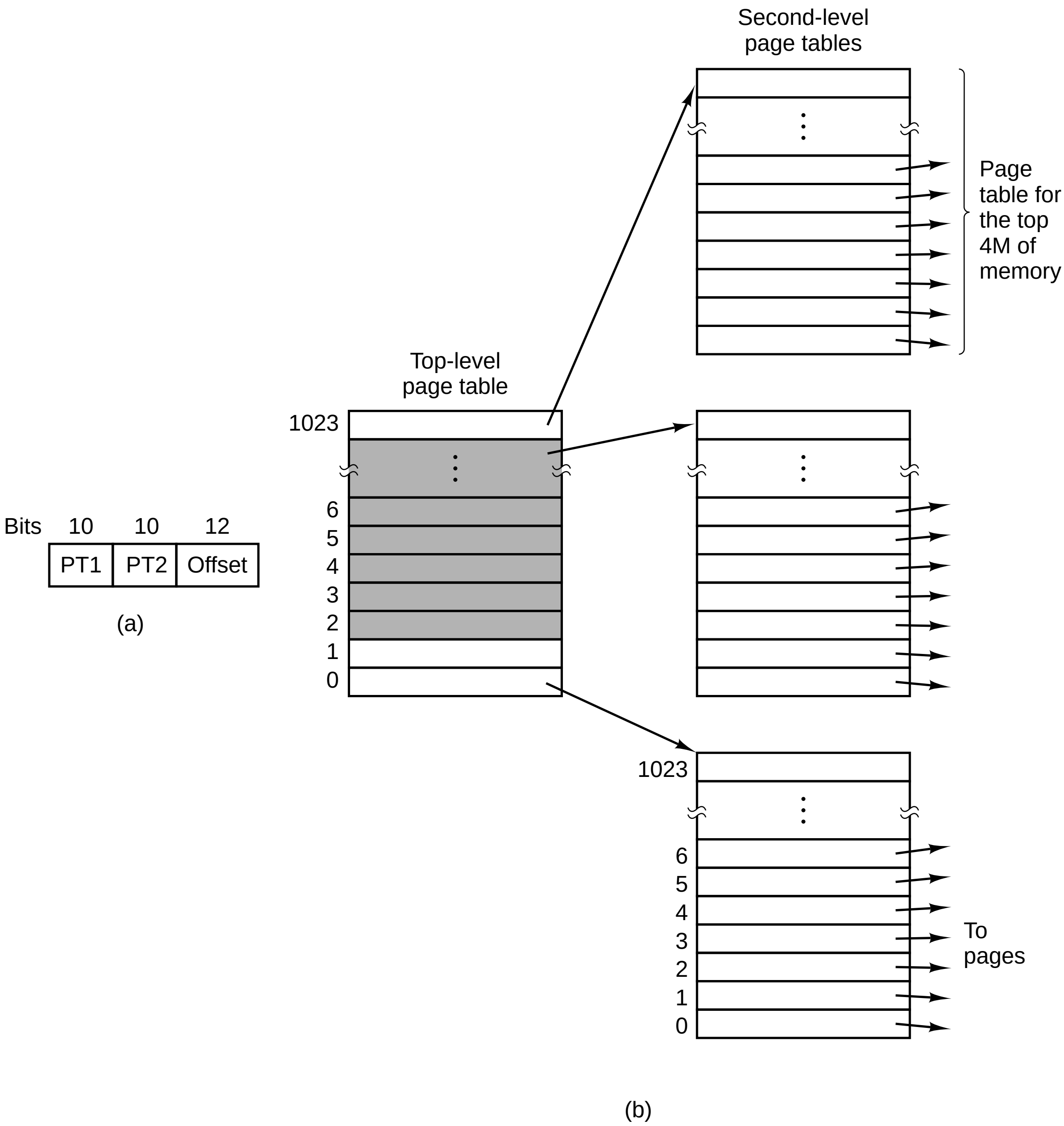


Figure 4-10. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

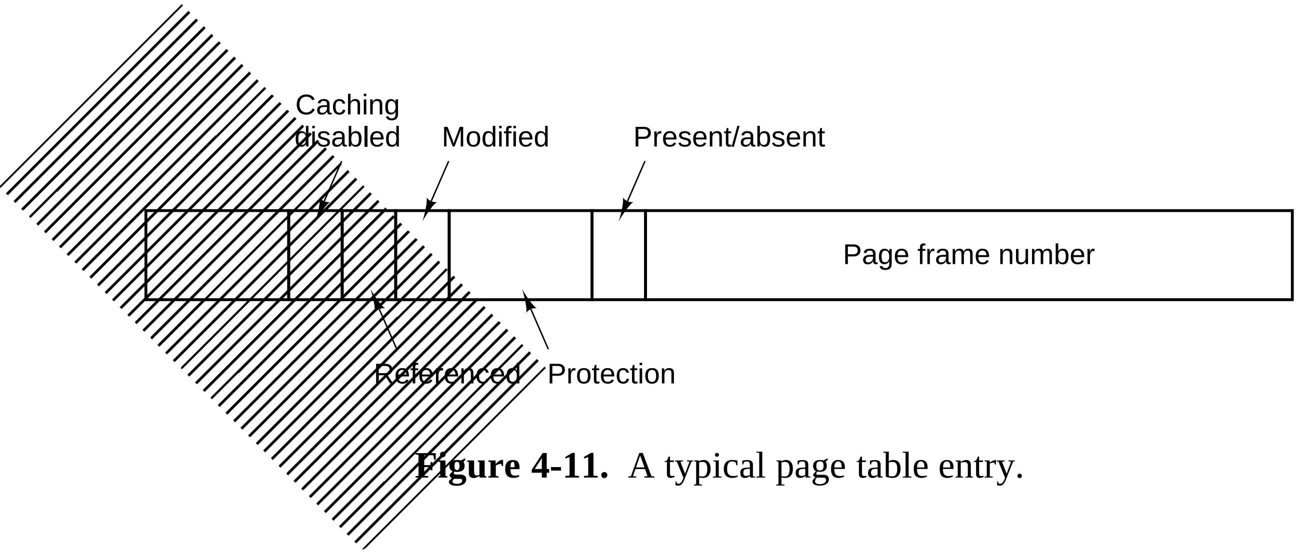


Figure 4-11. A typical page table entry.

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

Figure 4-12. A TLB to speed up paging.

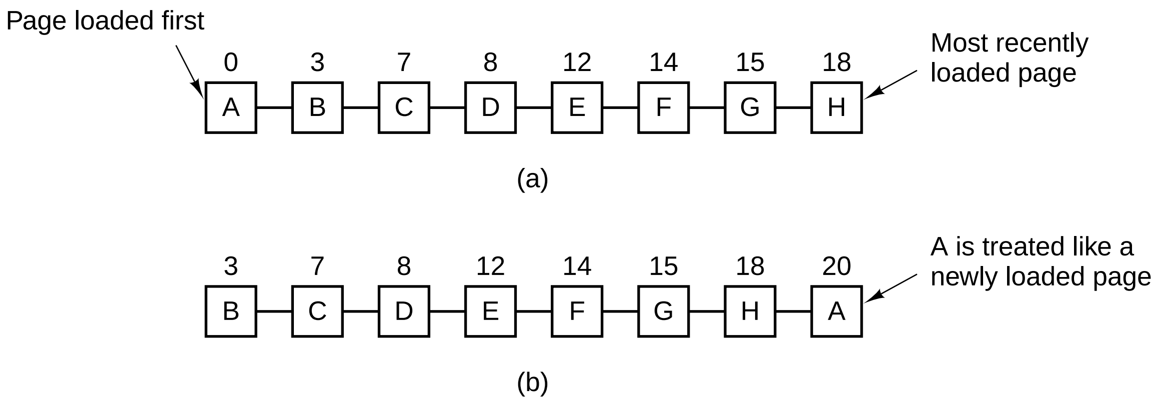


Figure 4-13. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set.

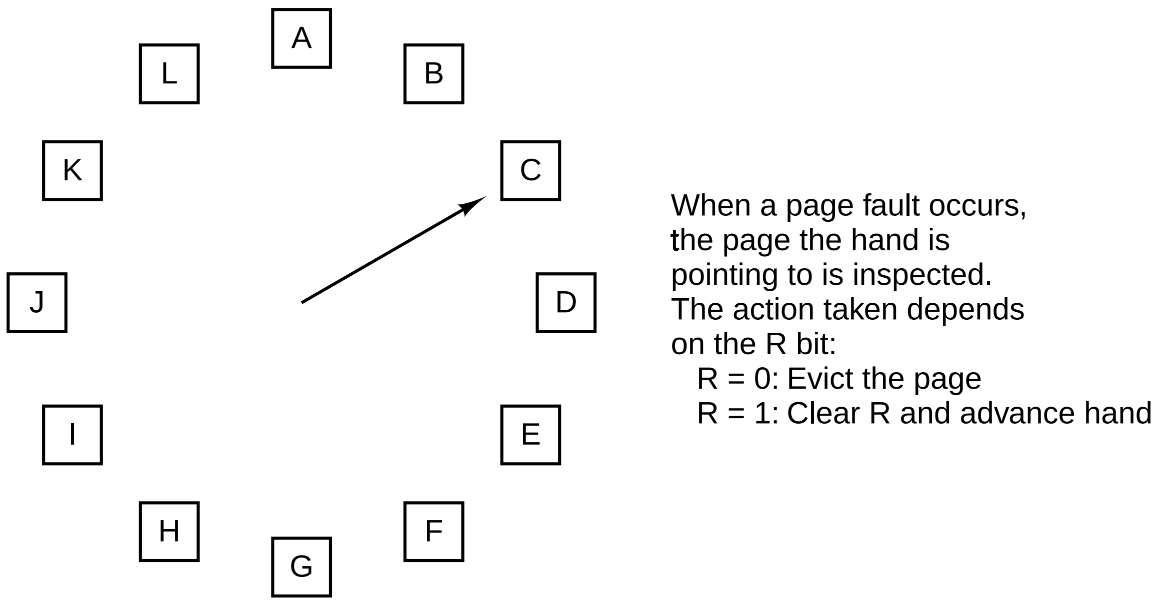


Figure 4-14. The clock page replacement algorithm.

Page

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

Page

0	1	2	3
0	0	1	1
1	0	1	1
0	0	0	0
0	0	0	0

(b)

Page

0	1	2	3
0	0	0	1
1	0	0	1
1	1	0	1
0	0	0	0

(c)

Page

0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0

(d)

Page

0	1	2	3
0	0	0	0
1	0	0	0
1	1	0	1
1	1	0	0

(e)

0	0	0	0
1	0	1	1
1	0	0	1
1	0	0	0

(f)

0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

(g)

0	1	1	0
0	0	1	0
0	0	0	0
1	1	1	0

(h)

0	1	0	0
0	0	0	0
1	1	0	1
1	1	0	0

(i)

0	1	0	0
0	0	0	0
1	1	0	0
1	1	1	0

(j)

Figure 4-15. LRU using a matrix.

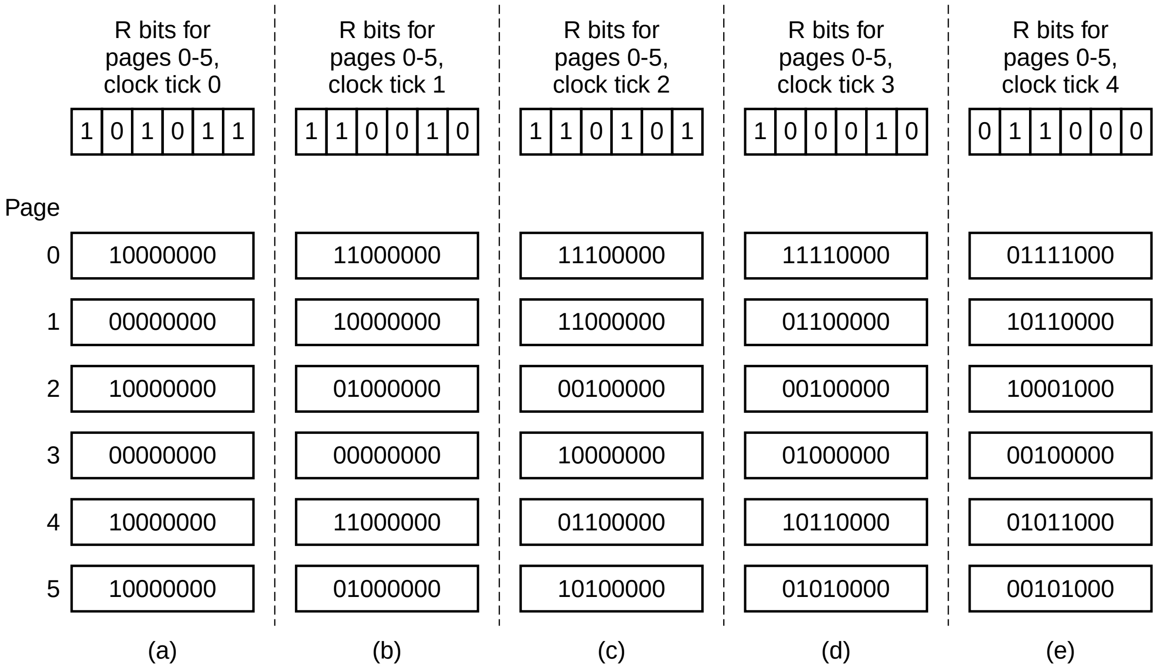


Figure 4-16. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

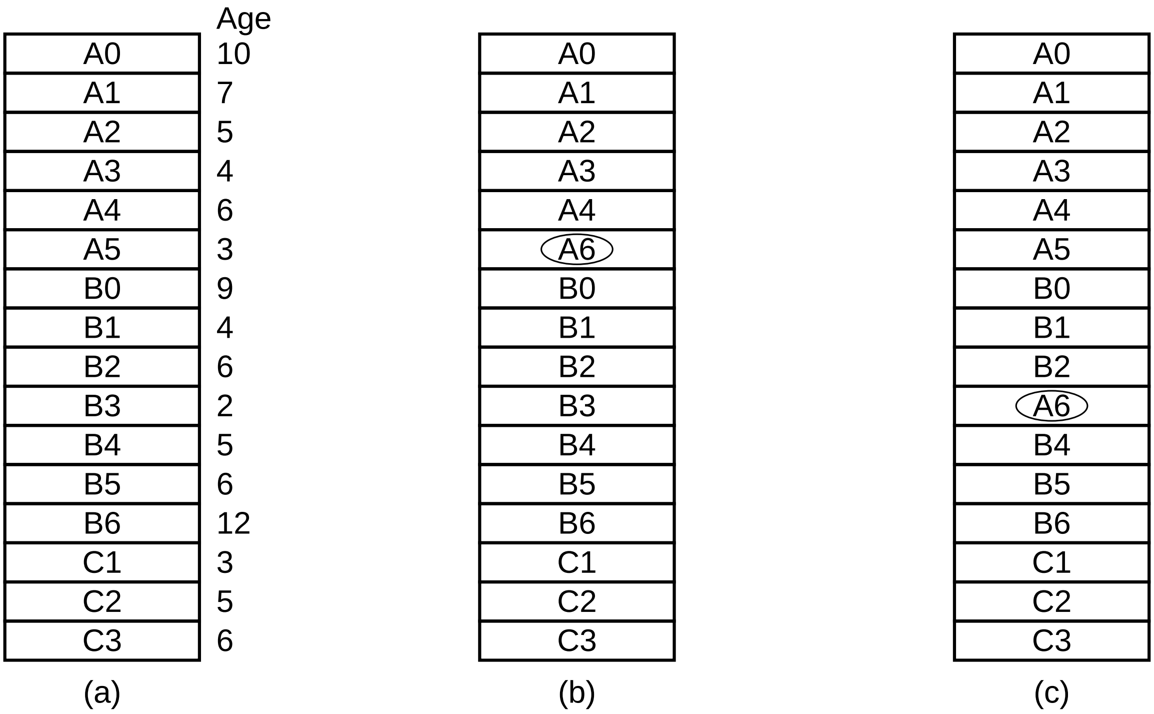


Figure 4-17. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

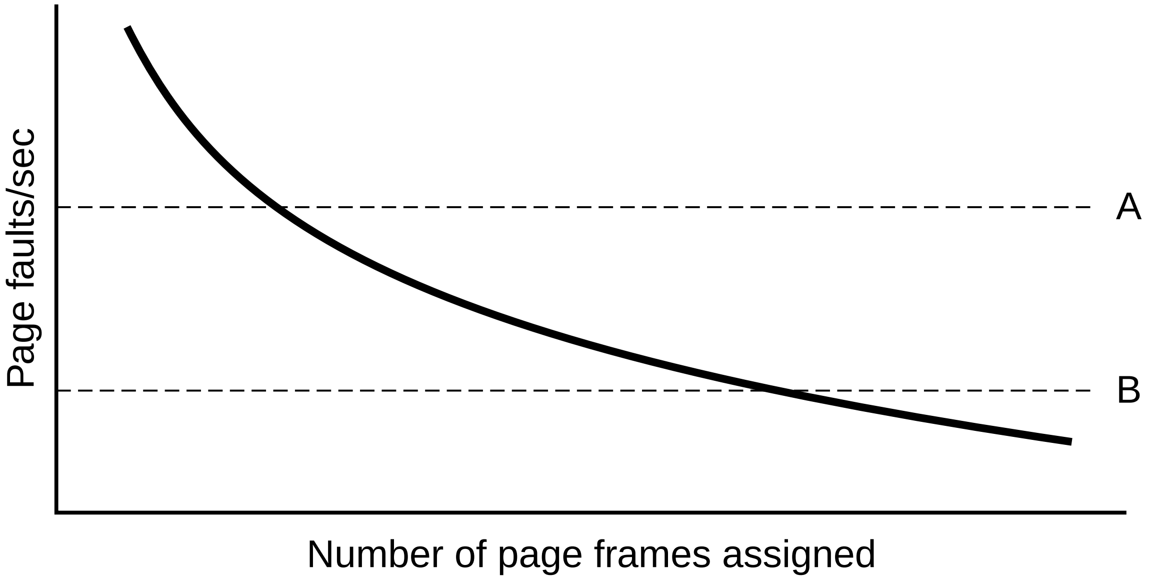


Figure 4-18. Page fault rate as a function of the number of page frames assigned.

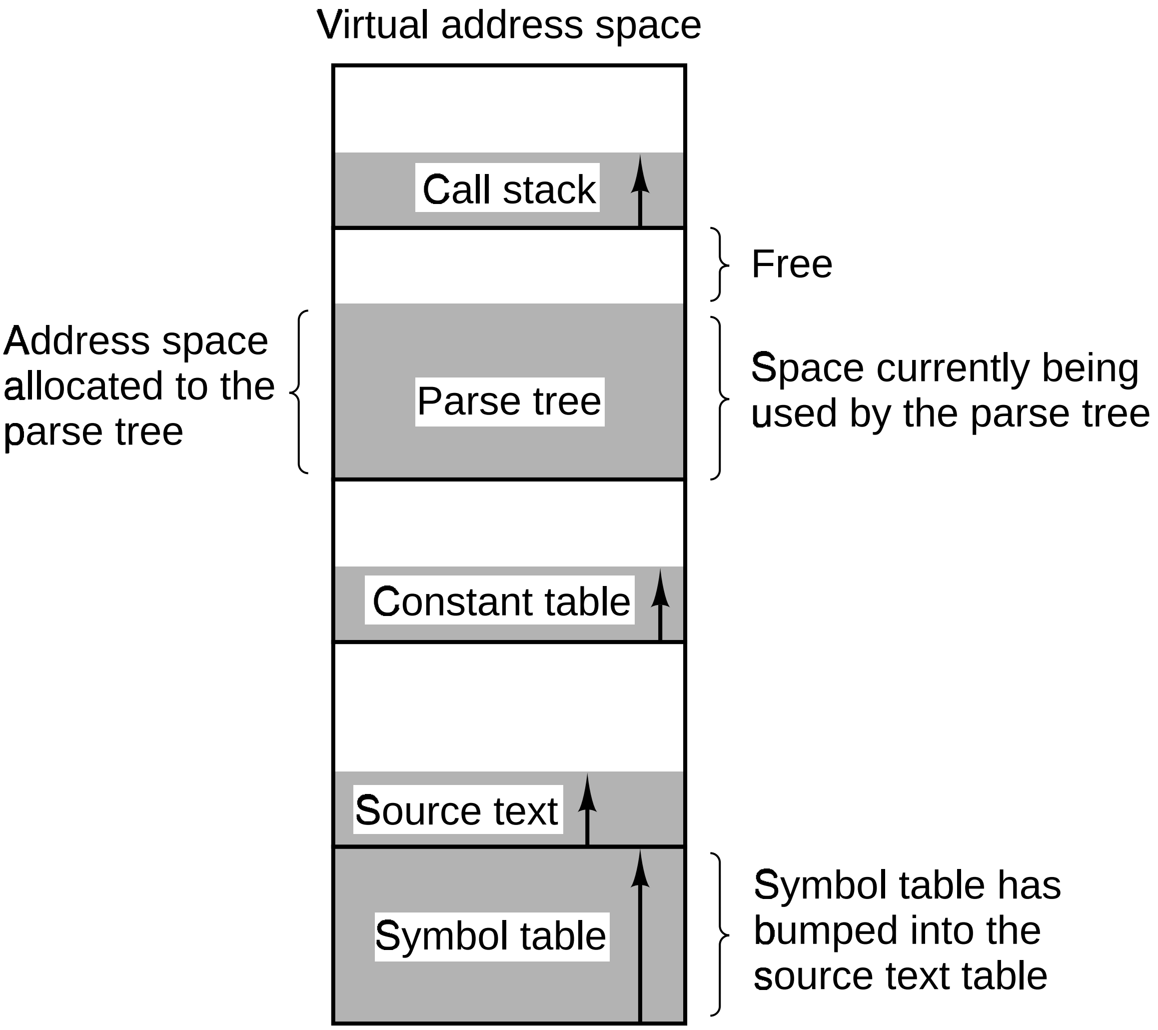


Figure 4-19. In a one-dimensional address space with growing tables, one table may bump into another.

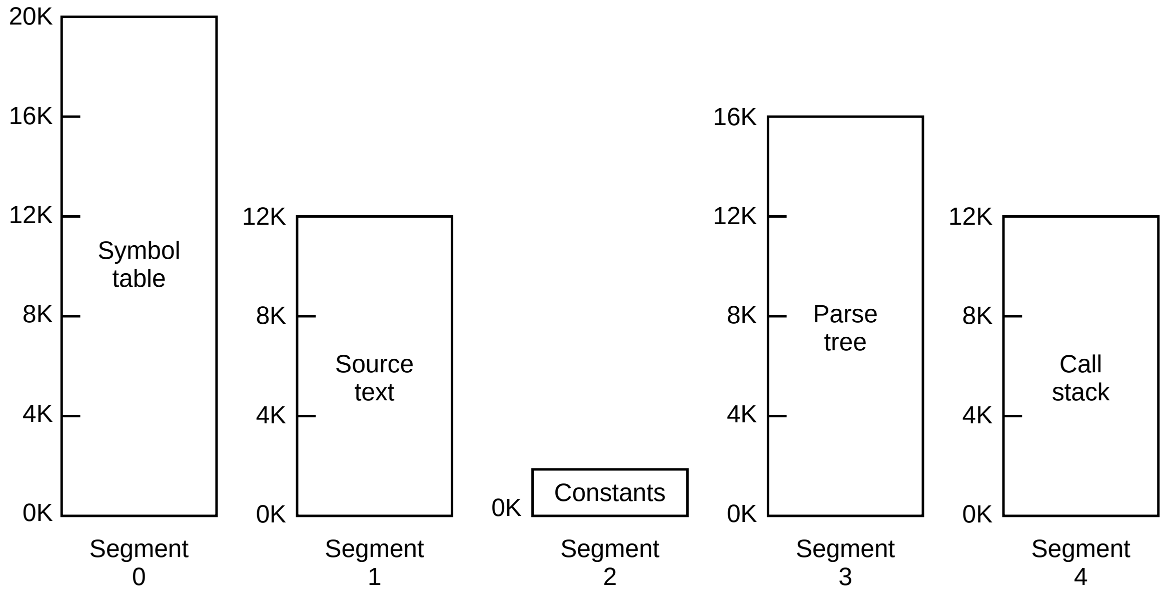


Figure 4-20. A segmented memory allows each table to grow or shrink independently of the other tables.

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
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

Figure 4-21. Comparison of paging and segmentation.

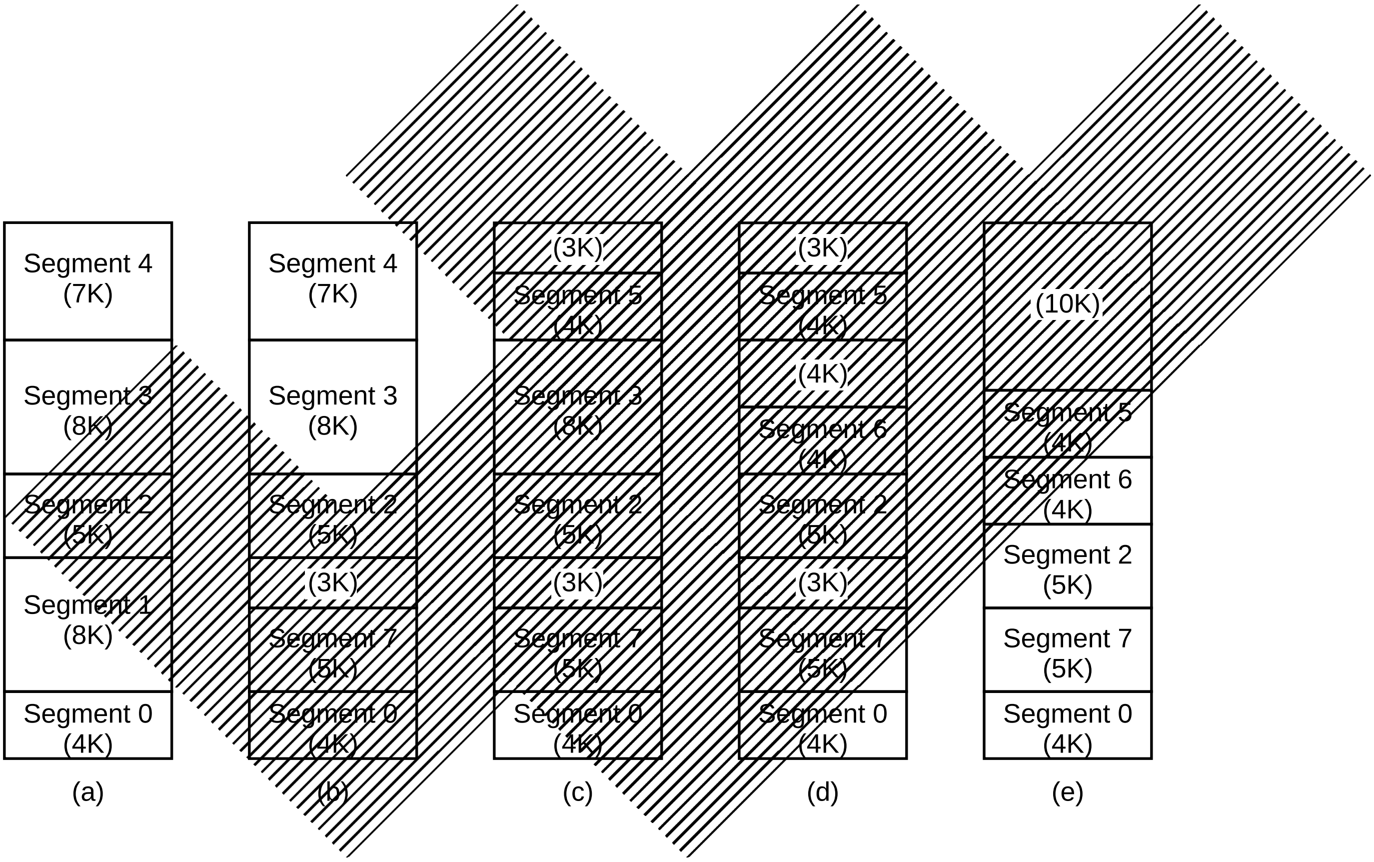


Figure 4-22. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

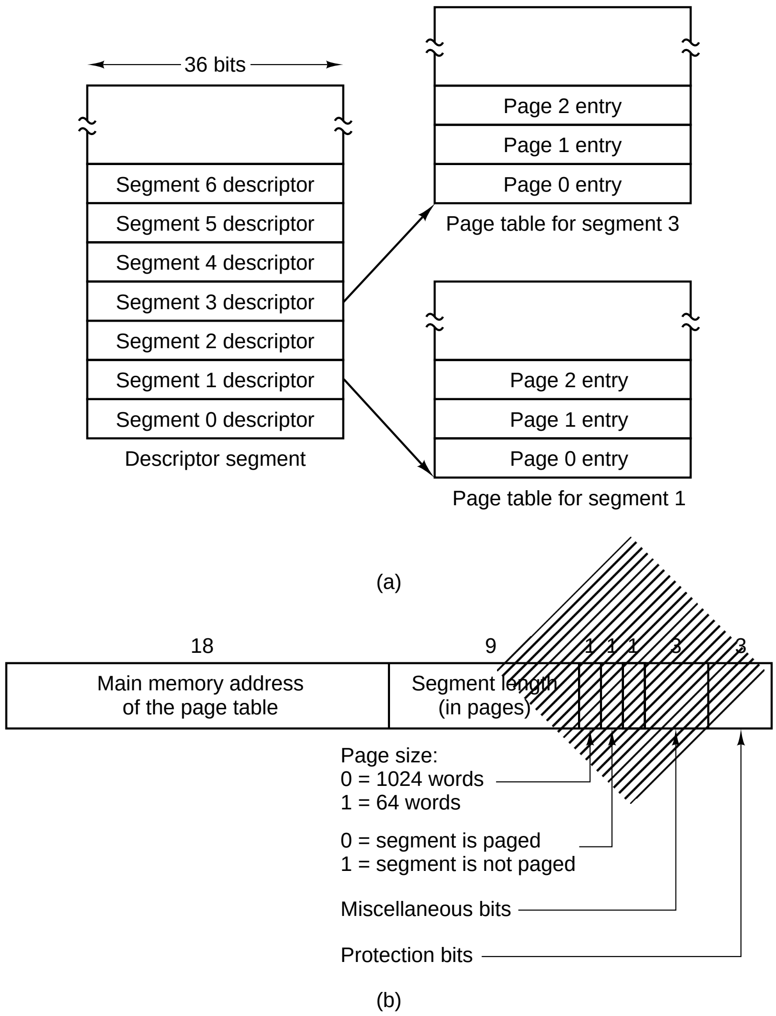


Figure 4-23. The MULTICS virtual memory. (a) The descriptor segment points to the page tables. (b) A segment descriptor. The numbers are the field lengths.

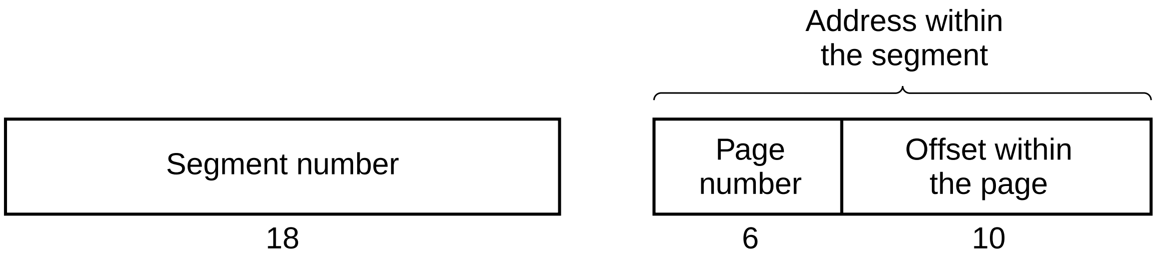


Figure 4-24. A 34-bit MULTICS virtual address.

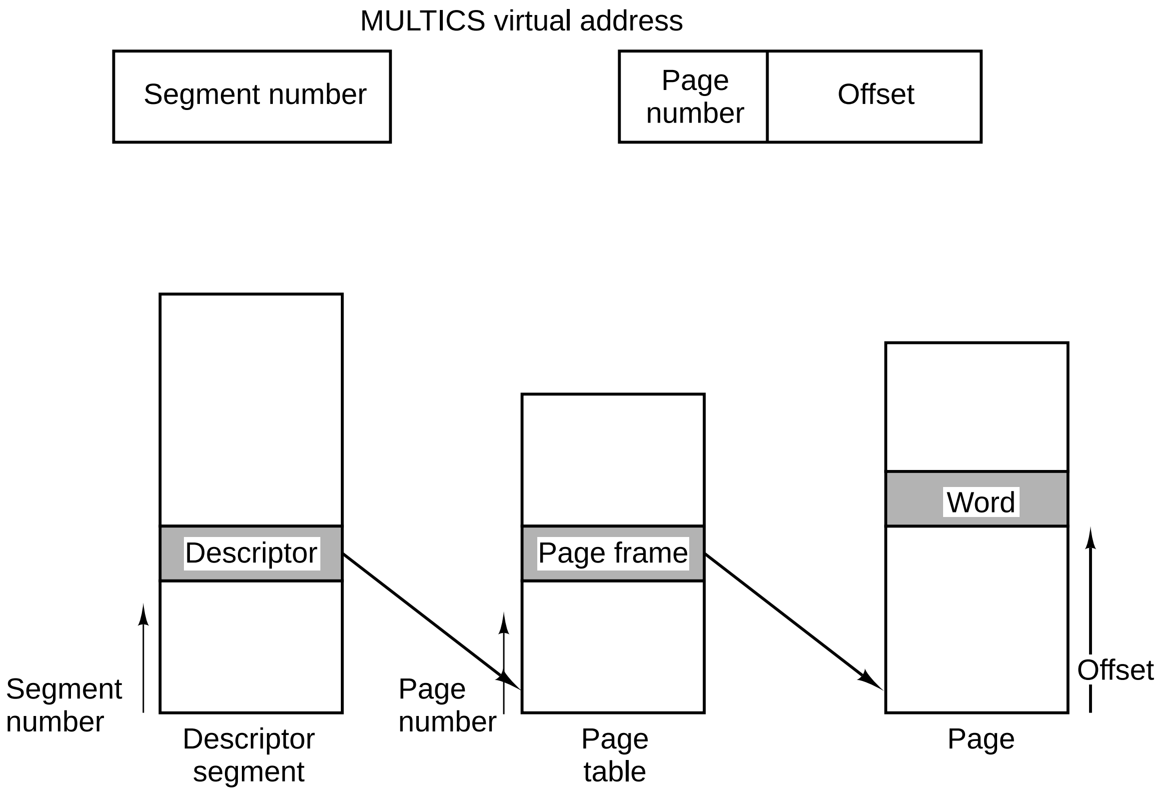


Figure 4-25. Conversion of a two-part MULTICS address into a main memory address.

Comparison field					
Segment number	Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figure 4-26. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

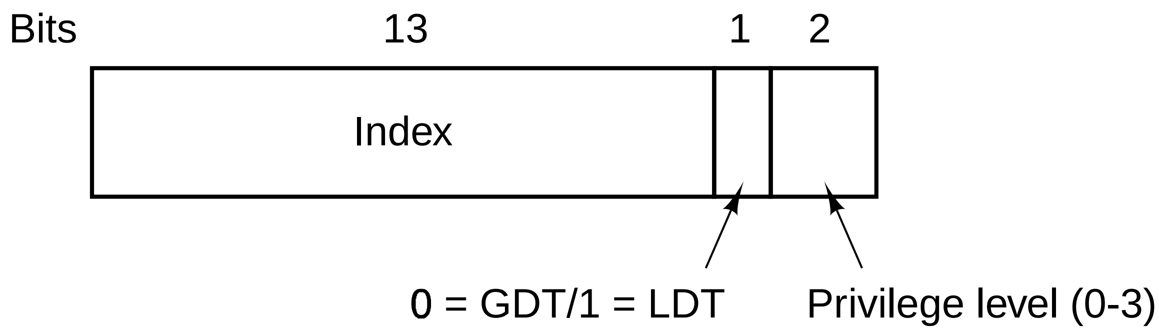


Figure 4-27. A Pentium selector.

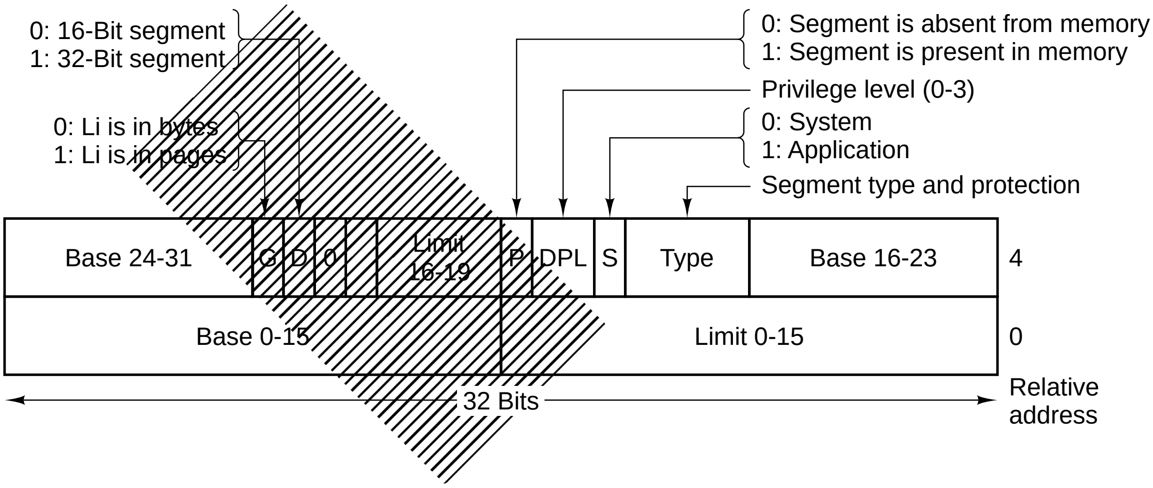


Figure 4-28. Pentium code segment descriptor. Data segments differ slightly.

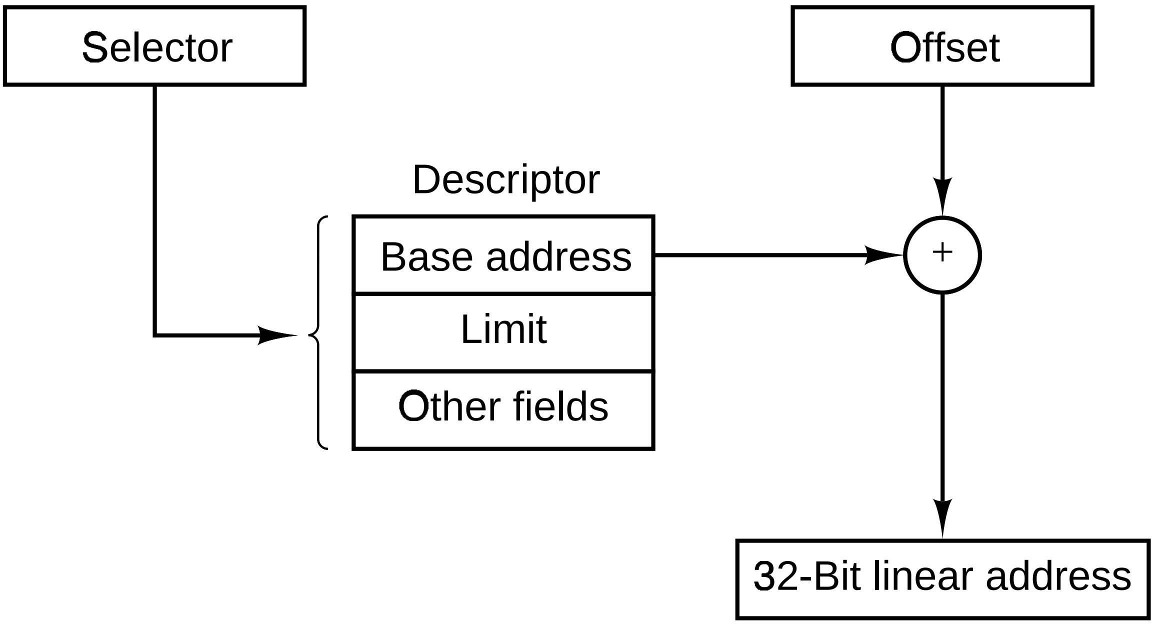


Figure 4-29. Conversion of a (selector, offset) pair to a linear address.

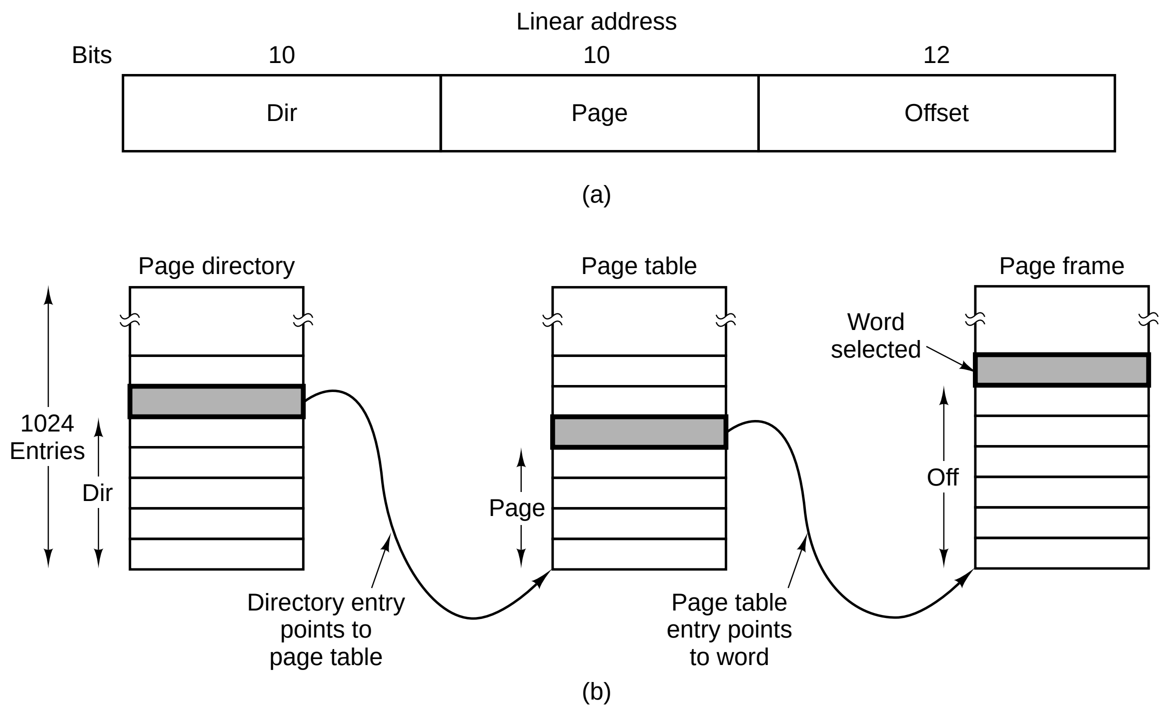


Figure 4-30. Mapping of a linear address onto a physical address.

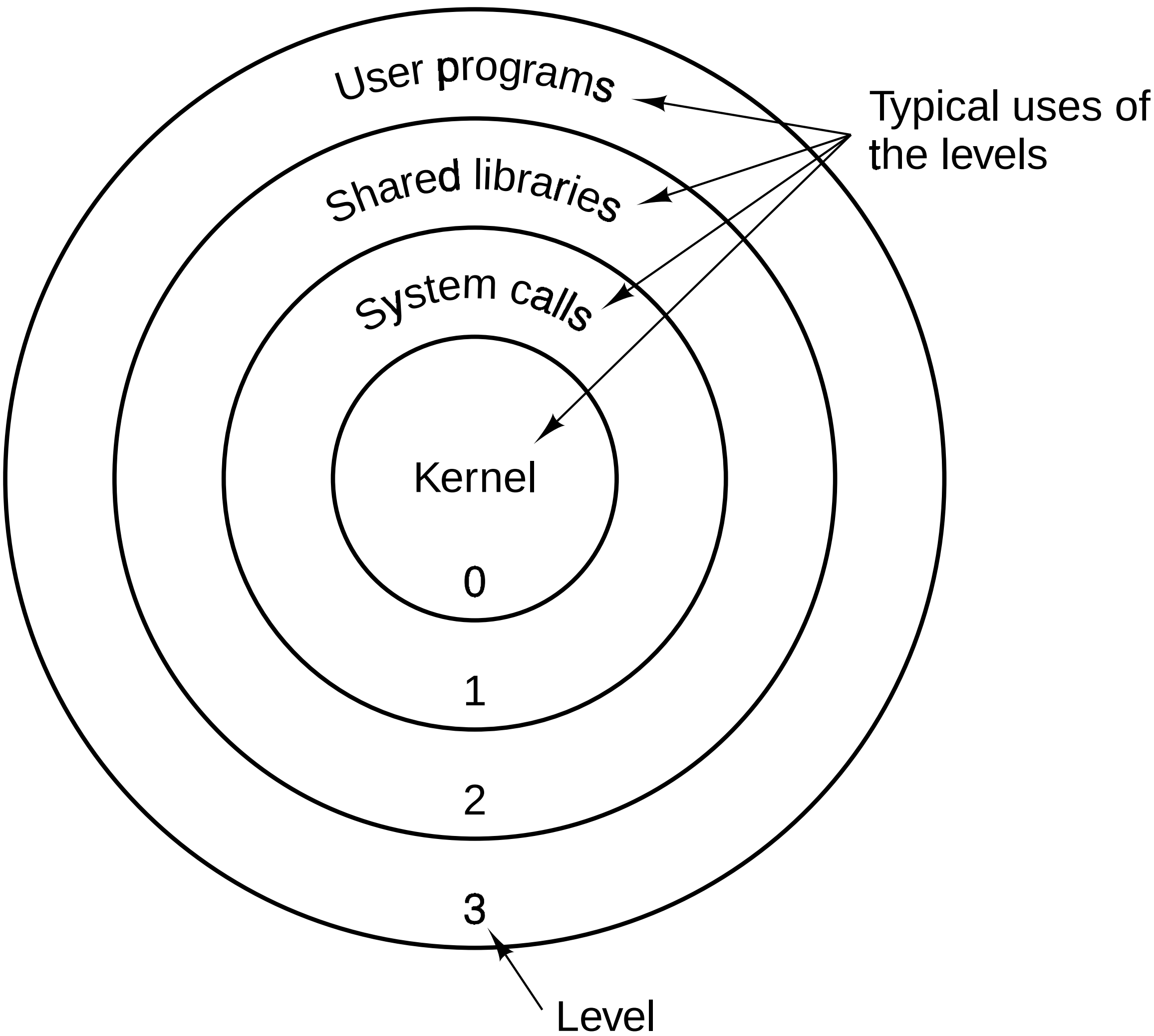


Figure 4-31. Protection on the Pentium.

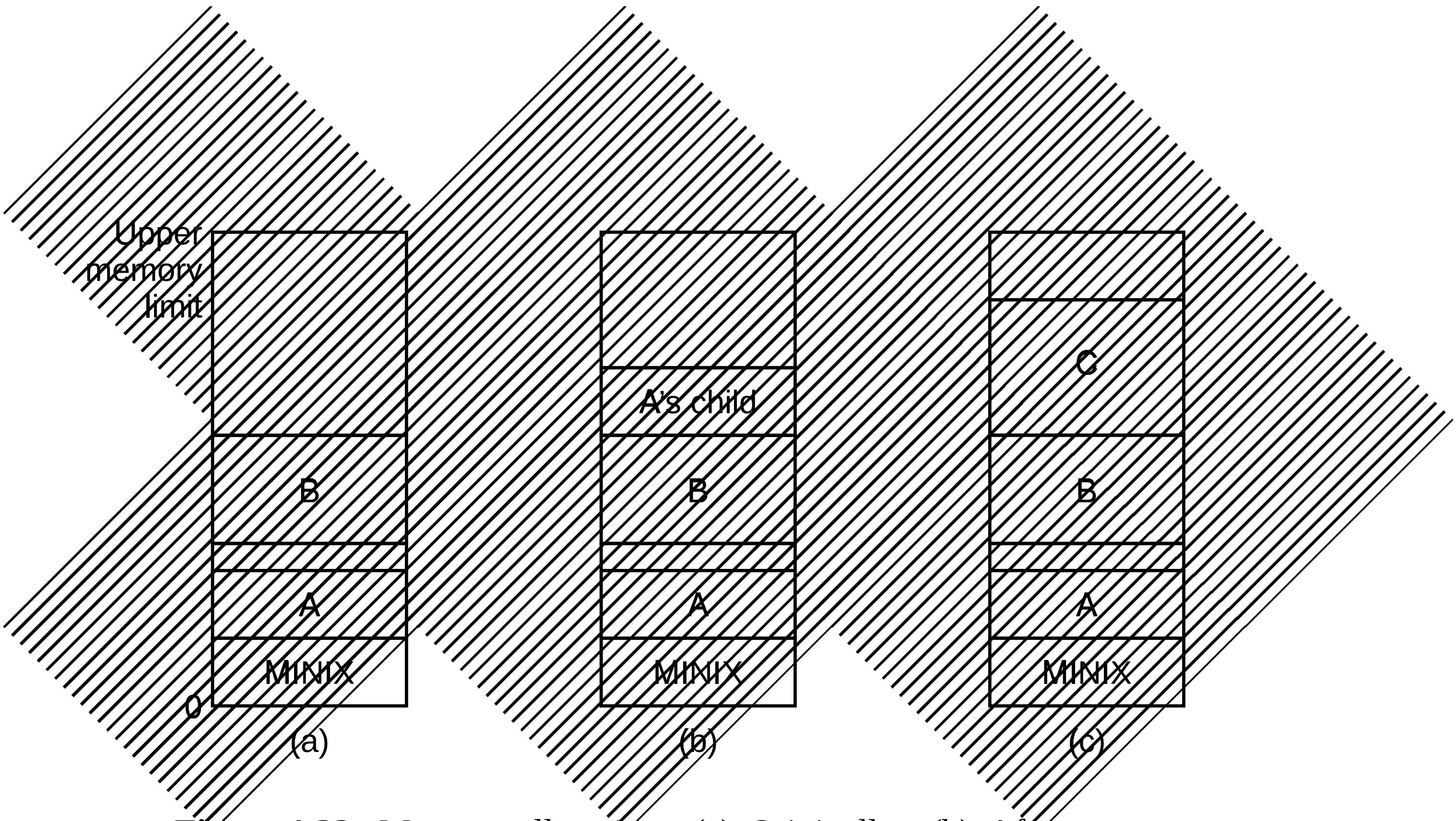


Figure 4-32. Memory allocation. (a) Originally. (b) After a SY FORK. (c) After the child does an EXEC . The shaded regions are unused memory. The process is a common I&D one.

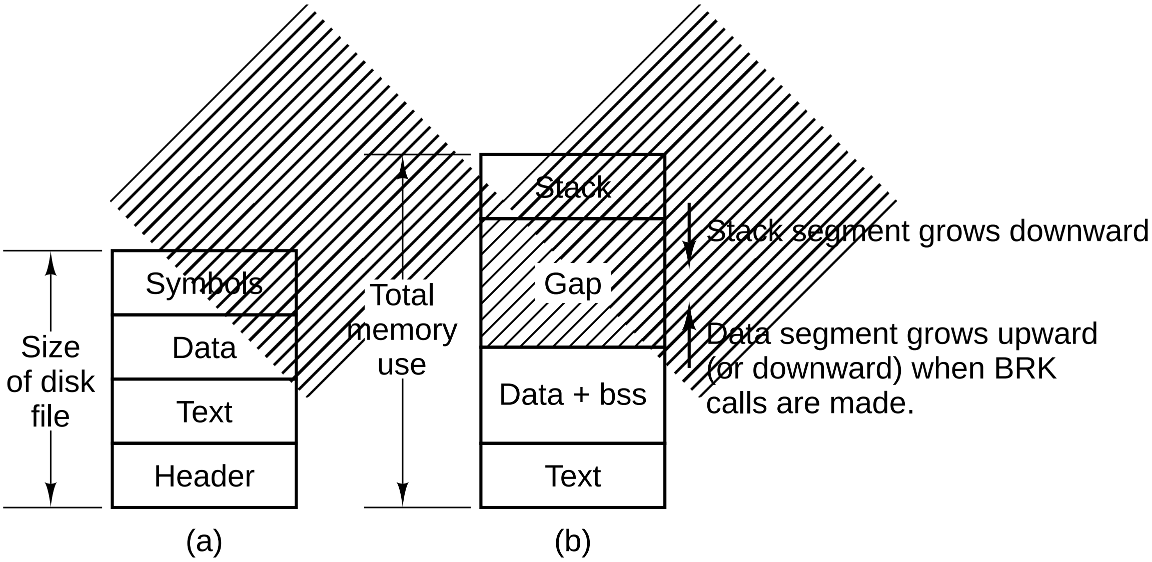
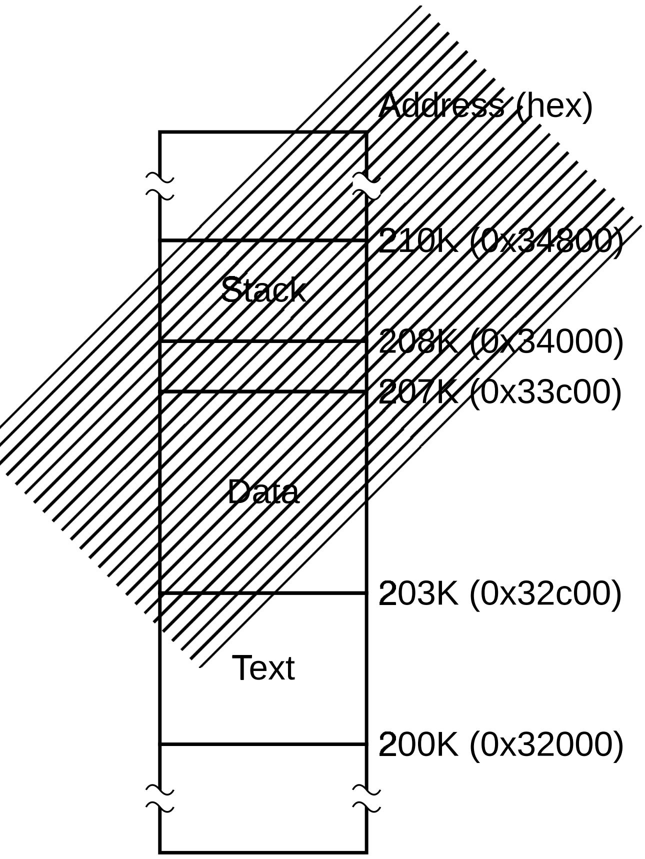


Figure 4-33. (a) A program as stored in a disk file. (b) Internal memory layout for a single process. In both parts of the figure the lowest disk or memory address is at the bottom and the highest address is at the top.

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	(none)	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	Sig. number, action, old action	Status
SIGSUSPEND	Signal mask	(No reply if successful)
SIGPENDING	(none)	Status
SIGMASK	How, set, old set	Status
SIGRETURN	Context	Status
GETUID	(none)	Uid, effective uid
GETGID	(none)	Gid, effective gid
GETPID	(none)	Pid, parent pid
SETUID	New uid	Status
SETGID	New gid	Status
SETSID	New sid	Process group
GETPGRP	New gid	Process group
PTRACE	Request, pid, address, data	Status
REBOOT	How (halt, reboot, or panic)	(No reply if successful)
KSIG	Process slot and signals	(No reply)

Figure 4-34. The message types, input parameters, and reply values used for communicating with the memory manager.



(a)

	Virtual	Physical	Length
Stack	0x20	0x340	0x8
Data	0	0x320	0x1c
Text	0	0x320	0

(b)

	Virtual	Physical	Length
Stack	0x14	0x340	0x8
Data	0	0x32c	0x10
Text	0	0x320	0xc

(c)

Figure 4-35. (a) A process in memory. (b) Its memory representation for nonseparate I and D space. (c) Its memory representation for separate I and D space.

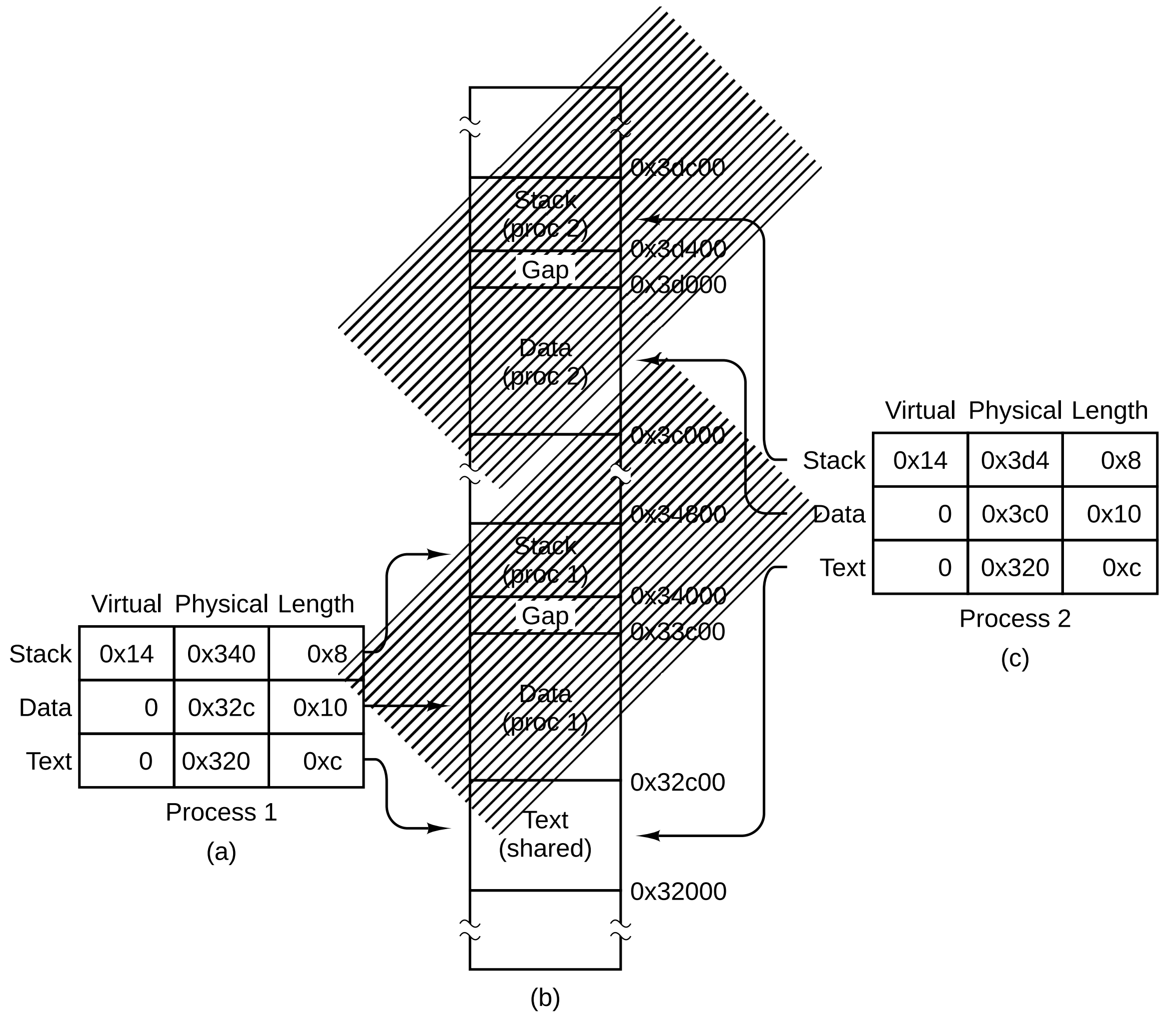


Figure 4-36. (a) The memory map of a separate I and D space process, as in the previous figure. (b) The layout in memory after a second process starts, executing the same program image with shared text. (c) The memory map of the second process.

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.

Figure 4-37. The steps required to carry out the FORK system call.

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.

Figure 4-38. The steps required to carry out the EXEC system call.

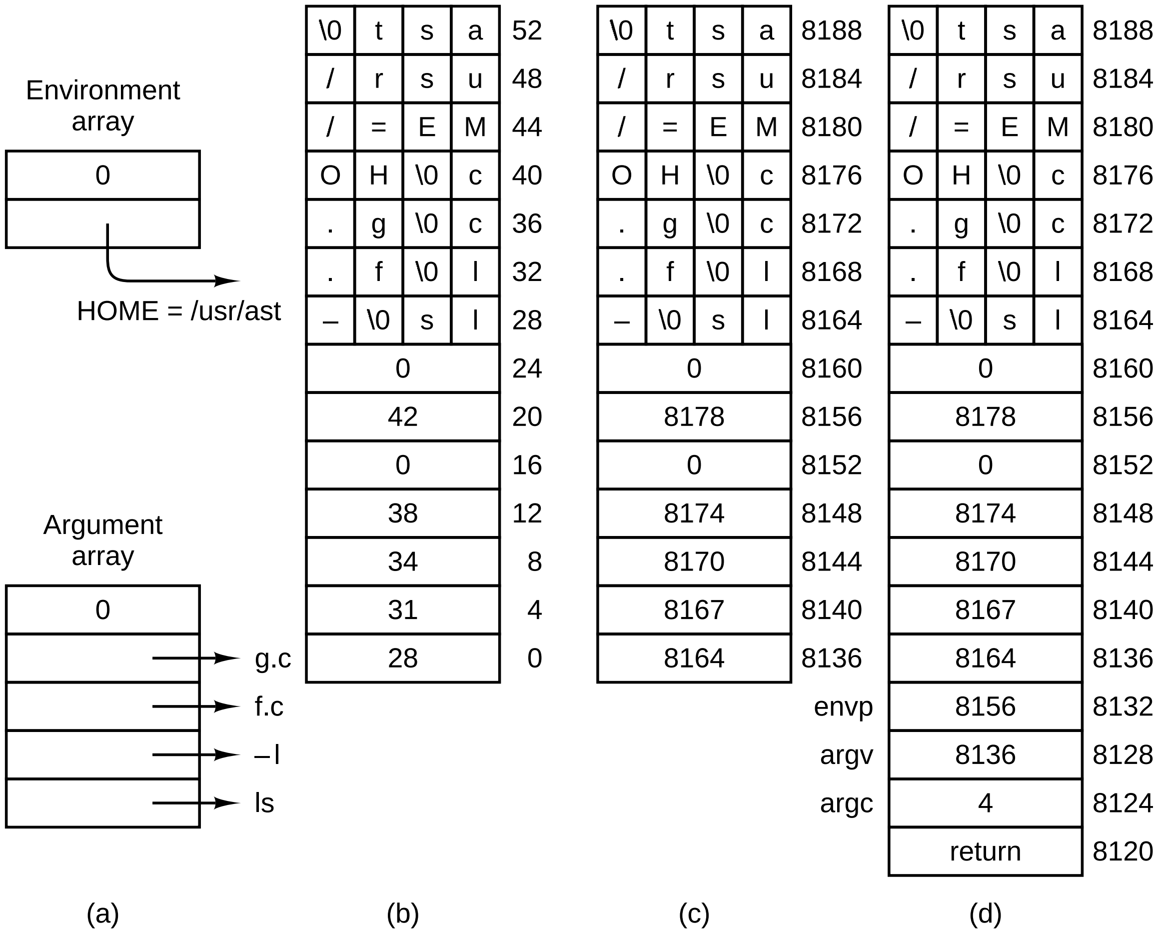


Figure 4-39. (a) The arrays passed to *execve*. (b) The stack built by *execve*. (c) The stack after relocation by the memory manager. (d) The stack as it appears to *main* at the start of execution.

```
push            ecx! push environ
push            edx! push argv
push            eax! push argc
call _main      ! main(argc, argv, envp)
push            eax! push exit status
call _exit
hlt             ! force a trap if exit fails
```

Figure 4-40. The key part of the C run-time, start-off routine.

Signal	Description	Generated by
SIGHUP	Hangup	KILL system call
SIGINT	Interrupt	Kernel
SIGQUIT	Quit	Kernel
SIGILL	Illegal instruction	Kernel (*)
SIGTRAP	Trace trap	Kernel (M)
SIGABRT	Abnormal termination	Kernel
SIGFPE	Floating point exception	Kernel (*)
SIGKILL	Kill (cannot be caught or ignored)	KILL system call
SIGUSR1	User-defined signal # 1	Not supported
SIGSEGV	Segmentation violation	Kernel (*)
SIGUSR2	User defined signal # 2	Not supported
SIGPIPE	Write on a pipe with no one to read it	Kernel
SIGALRM	Alarm clock, timeout	Kernel
SIGTERM	Software termination signal from kill	KILL system call
SIGCHLD	Child process terminated or stopped	Not supported
SIGCONT	Continue if stopped	Not supported
SIGSTOP	Stop signal	Not supported
SIGTSTP	Interactive stop signal	Not supported
SIGTTIN	Background process wants to read	Not supported
SIGTTOU	Background process wants to write	Not supported

Figure 4-41. Signals defined by POSIX and MINIX. Signals indicated by (*) depend upon hardware support. Signals marked (M) are not defined by POSIX, but are defined by MINIX for compatibility with older programs. Several obsolete names and synonyms are not listed here.

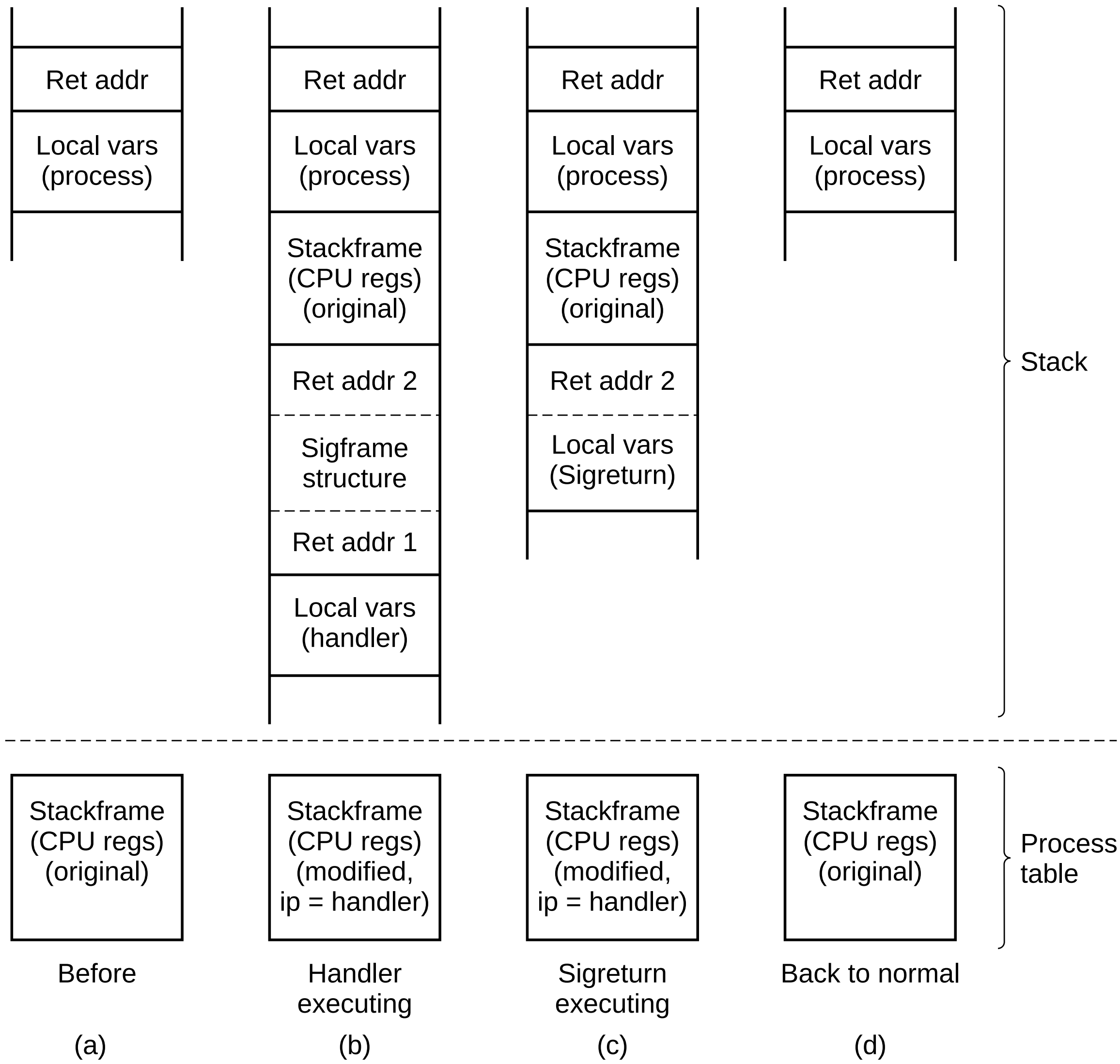


Figure 4-42. A process' stack (above) and its stackframe in the process table (below) corresponding to phases in handling a signal. (a) State as process is taken out of execution. (b) State as handler begins execution. (c) State while SIGRETURN is executing. (d) State after SIGRETURN completes execution.

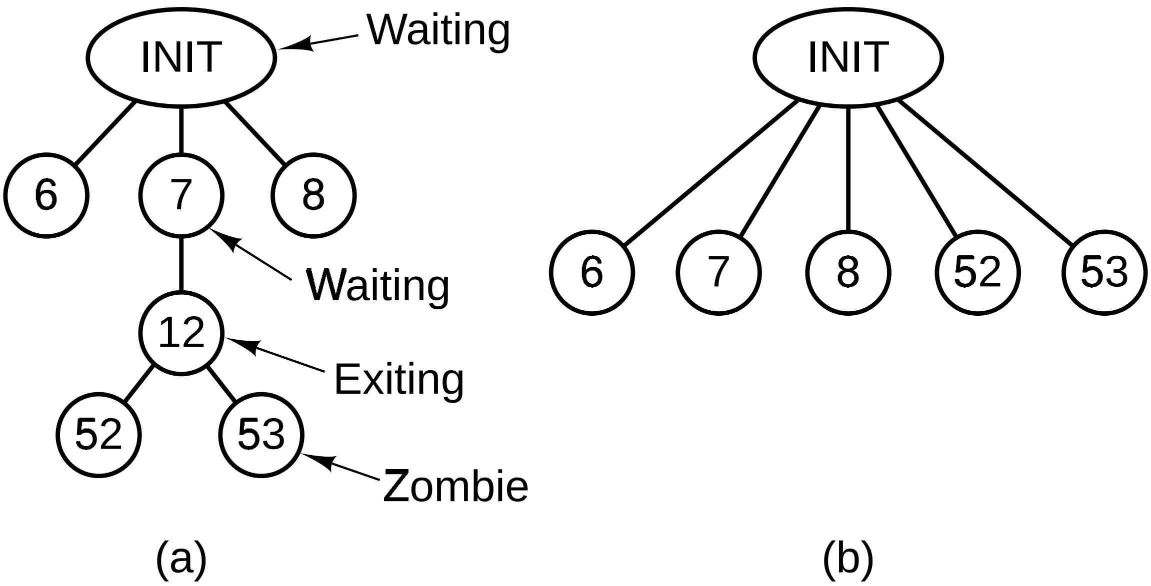


Figure 4-43. (a) The situation as process 12 is about to exit. (b) The situation after it has exited.

System call	Purpose
SIGACTION	Modify response to future signal
SIGPROCMASK	Change set of blocked signals
KILL	Send signal to another process
ALARM	Send ALRM signal to self after delay
PAUSE	Suspend self until future signal
SIGSUSPEND	Change set of blocked signals, then PAUSE
SIGPENDING	Examine set of pending (blocked) signals
SIGRETURN	Clean up after signal handler

Figure 4-44. System calls relating to signals.

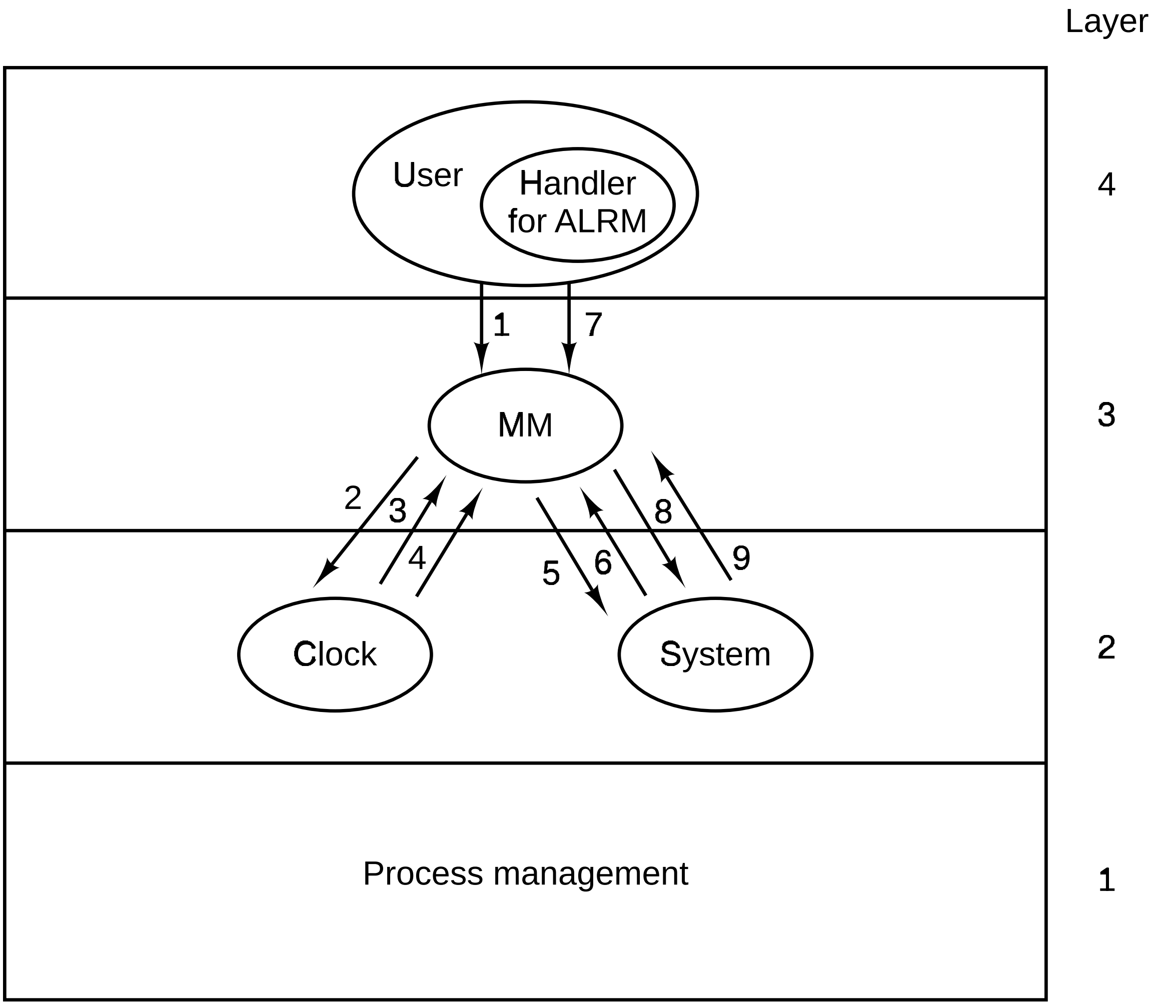


Figure 4-45. Messages for an alarm. The most important are: (1) User does ALARM. (4) After the set time has elapsed, the signal arrives. (7) Handler terminates with call to SIGRETURN. See text for details.

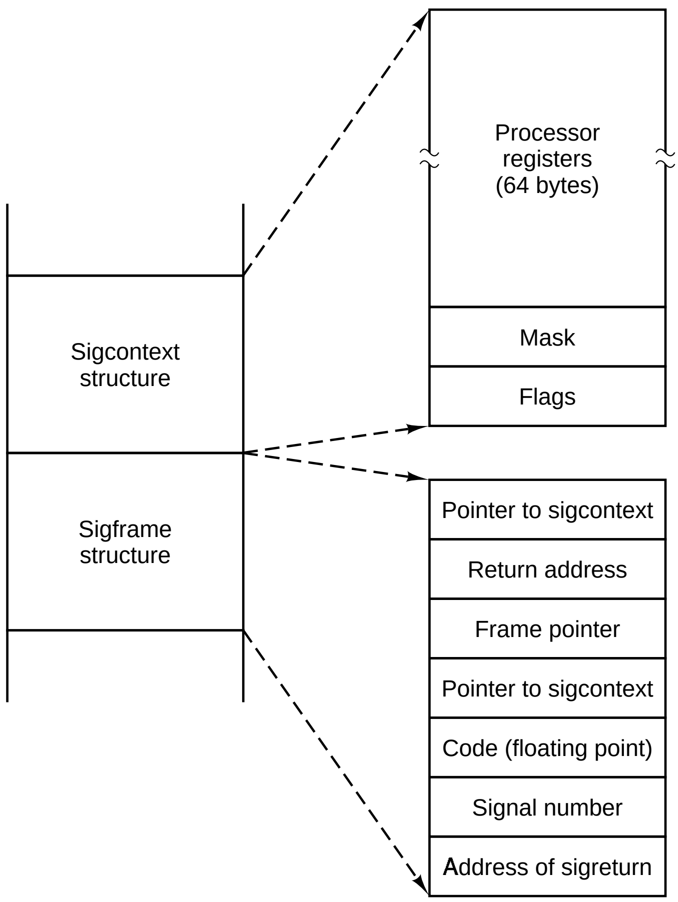


Figure 4-46. The sigcontext and sigframe structures pushed on the stack to prepare for a signal handler. The processor registers are a copy of the stackframe used during a context switch.

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

Figure 4-47. The system calls supported in *mm/getset.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

Figure 4-48. Debugging commands supported by *mm/trace.c*.