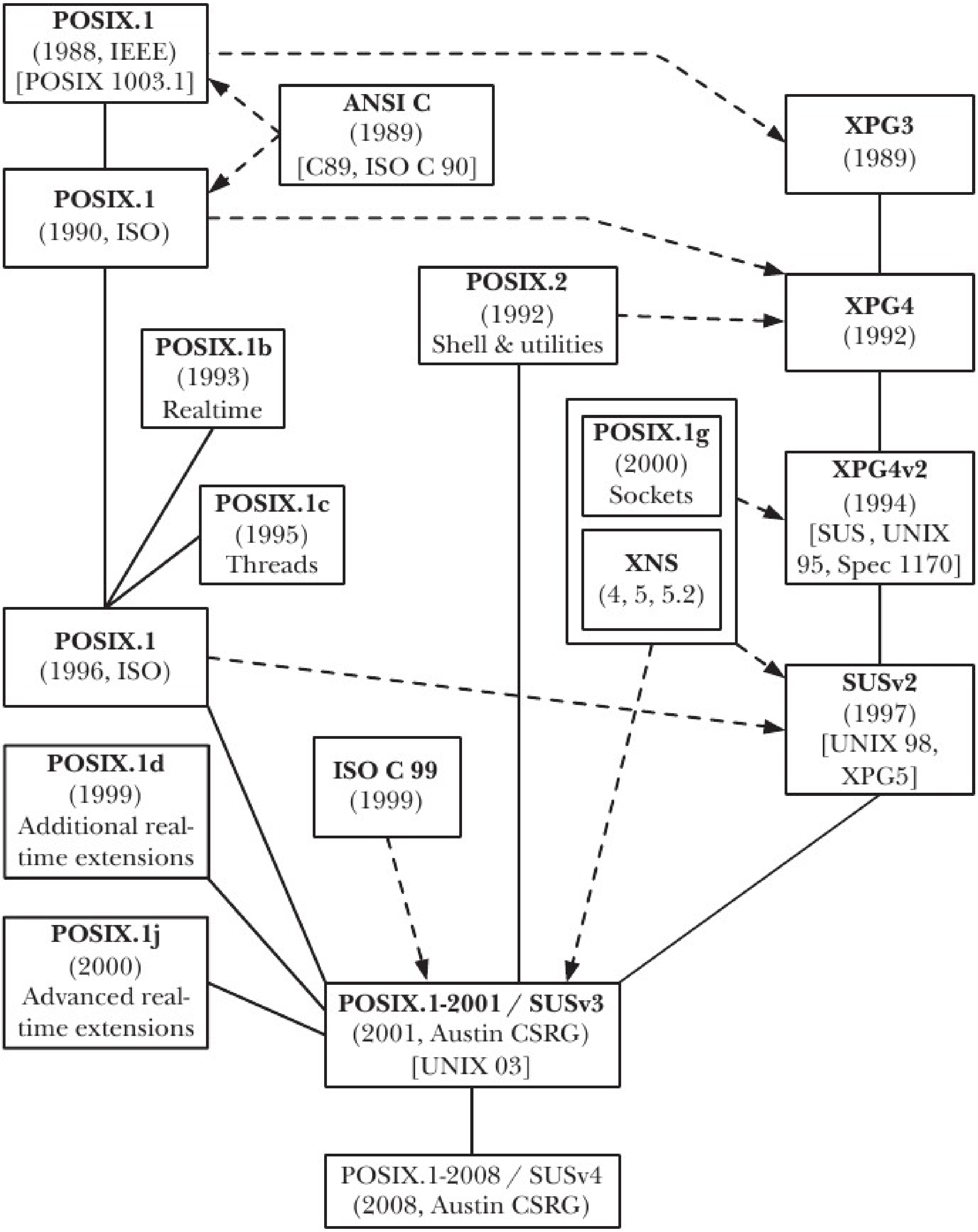
Figure 1.1 - Various standards



POSIX.1 (1988, IEEE) [POSIX 1003.1]
ANSI C (1989) [C89, ISO C 90]
XPG3 (1989)
POSIX.1 (1990, ISO)
POSIX.2 (1992) Shell & utilities
XPG4 (1992)
POSIX.1b (1993) Realtime
POSIX.1g (2000) Sockets
XPG4v2 (1994) [SUS, UNIX 95, Spec 1170]
POSIX.1c (1995) Threads
XNS (4, 5, 5.2)
POSIX.1 (1996, ISO)
SUSv2 (1997) [UNIX 98, XPG5]
POSIX.1d (1999) Additional real-time extensions
ISO C 99 (1999)
POSIX.1j (2000) Advanced real-time extensions
POSIX.1-2001 / SUSv3 (2001, Austin CSRG) [UNIX 03]
POSIX.1-2008 / SUSv4 (2008, Austin CSRG)

Figure 1.2 - system call flow

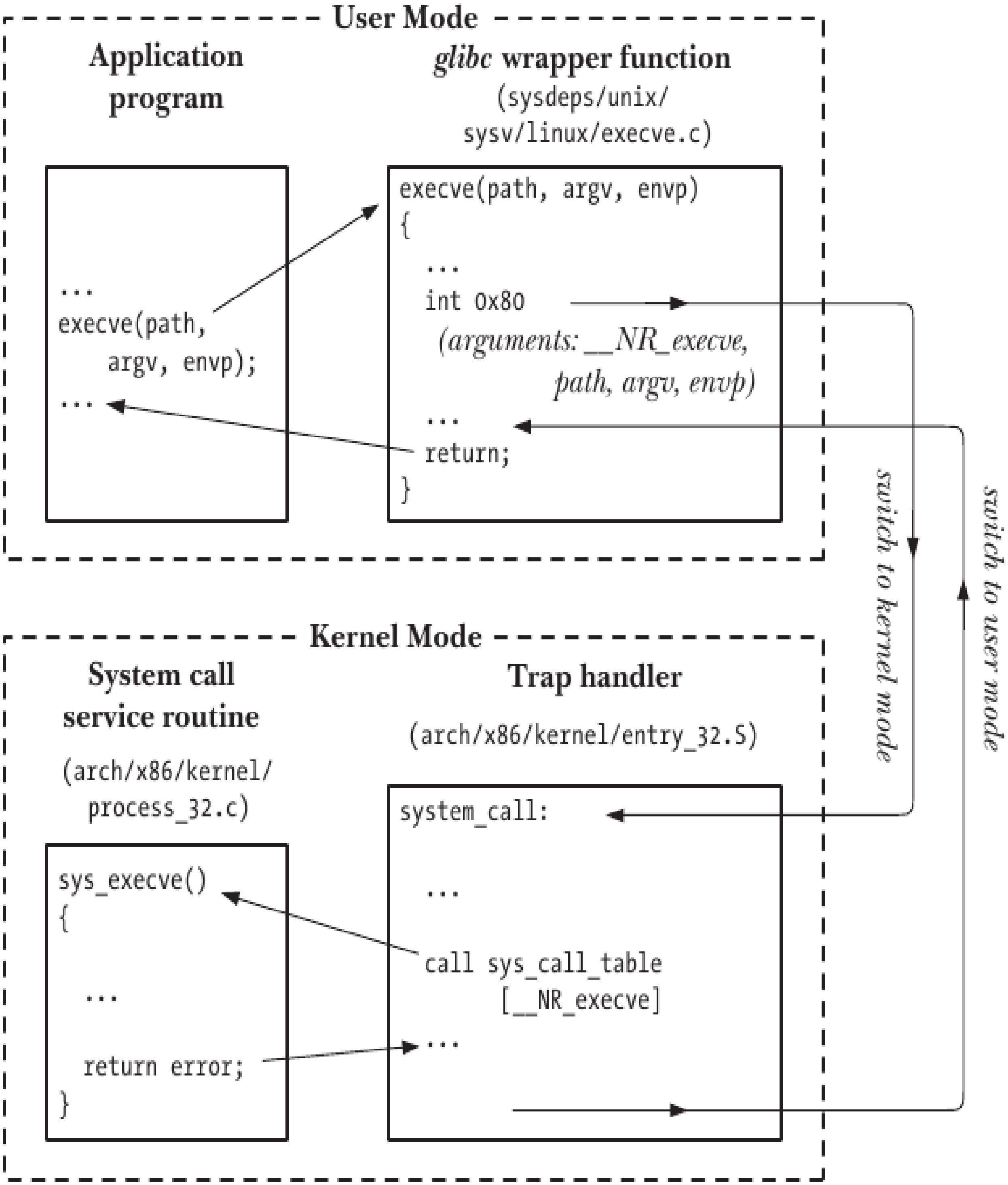


Figure 1.3 - process memory layout

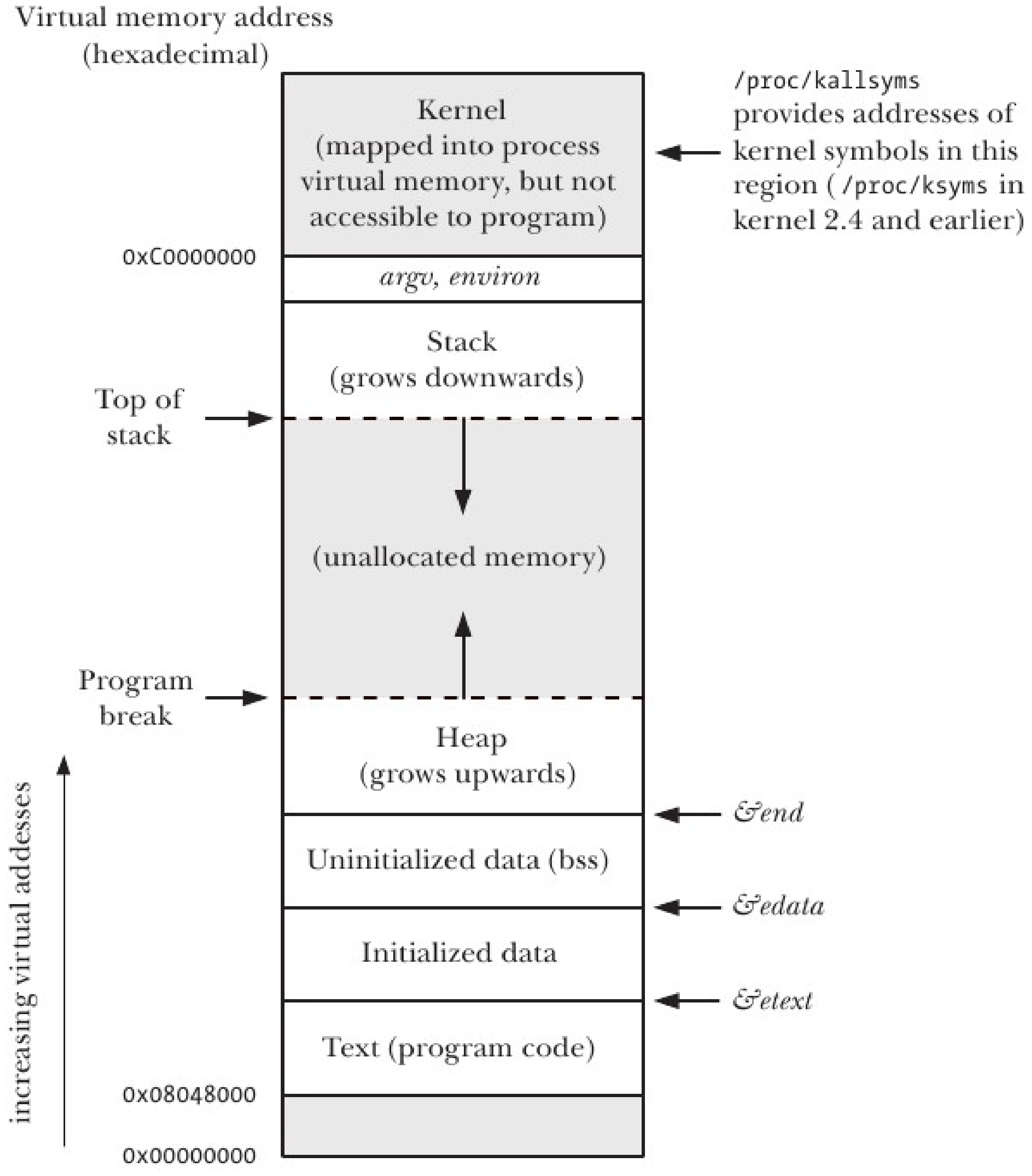

Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
increasing virtual addresses
0x08048000
0x00000000

Figure 1.4 - process address space to page-frames

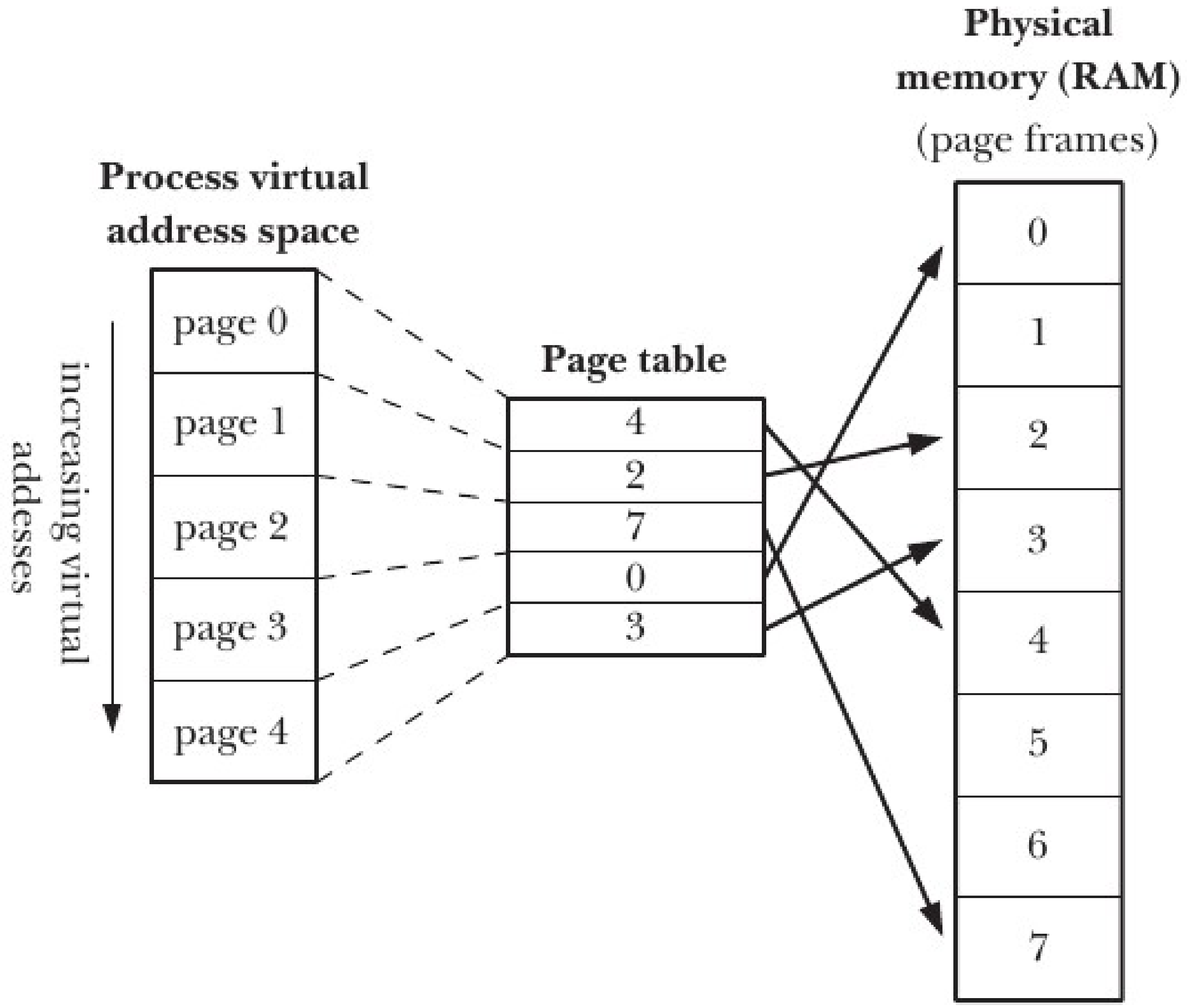
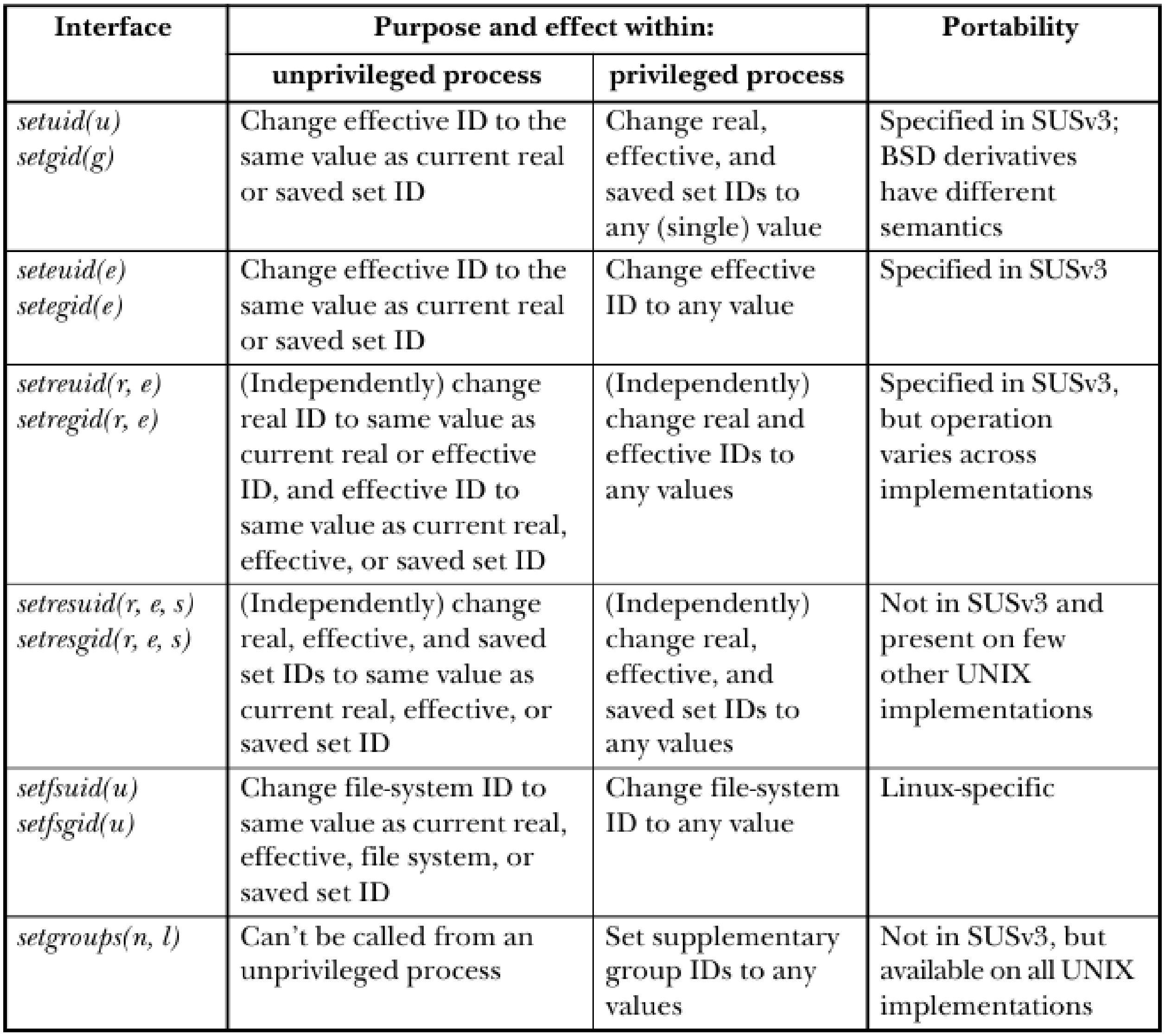


Figure 1.5 - changing process credentials



Interface	Purpose and effect within:		Portability
	unprivileged process	privileged process	

setuid(u)
setgid(g)
	Change effective ID to the same value as current real or saved set ID	Change real, effective, and saved set IDs to any (single) value	Specified in SUSv3; BSD derivatives have different semantics

seteuid(e)
setegid(e)
	Change effective ID to the same value as current real or saved set ID	Change effective ID to any value	Specified in SUSv3

setreuid(r, e)
setregid(r, e)
	(Independently) change real ID to same value as current real or effective ID, and effective ID to same value as current real, effective, or saved set ID	(Independently) change real and effective IDs to any values	Specified in SUSv3, but operation varies across implementations

setresuid(r, e, s)
setresgid(r, e, s)
	(Independently) change real, effective, and saved set IDs to same value as current real, effective, or saved set ID	(Independently) change real, effective, and saved set IDs to any values	Not in SUSv3 and present on few other UNIX implementations

setfsuid(u)
setfsgid(u)
	Change file-system ID to same value as current real, effective, file system, or saved set ID	Change file-system ID to any value	Linux-specific

setgroups(n, l)	Can't be called from an unprivileged process	Set supplementary group IDs to any values	Not in SUSv3, but available on all UNIX implementations

Figure 1.6 - process creation and related

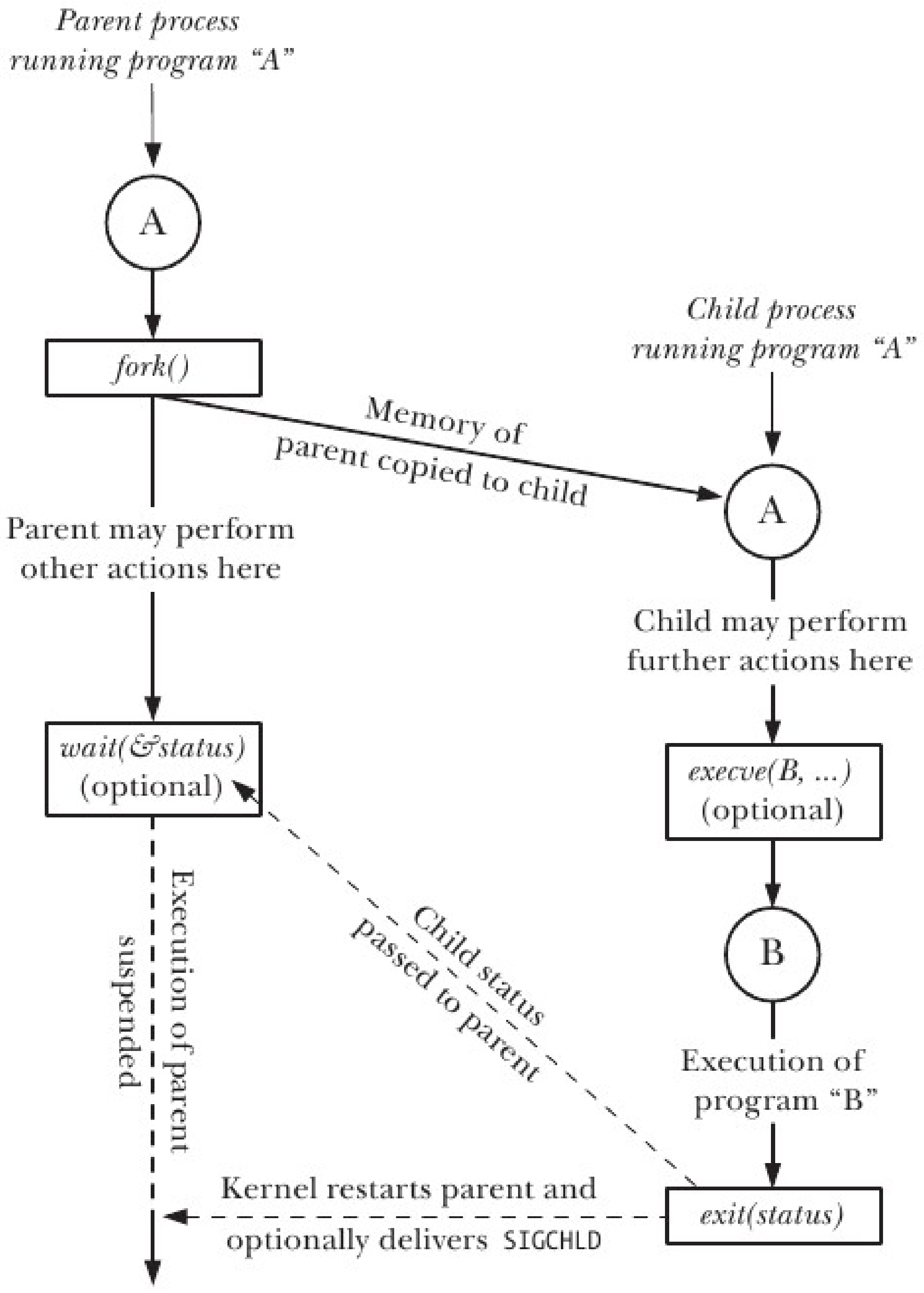

Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
B
Execution of parent suspended
Child status passed to parent
Execution of program "B"
Kernel restarts parent and optionally delivers SIGCHLD
exit(status)

Figure 1.7 - process creation and file descriptors

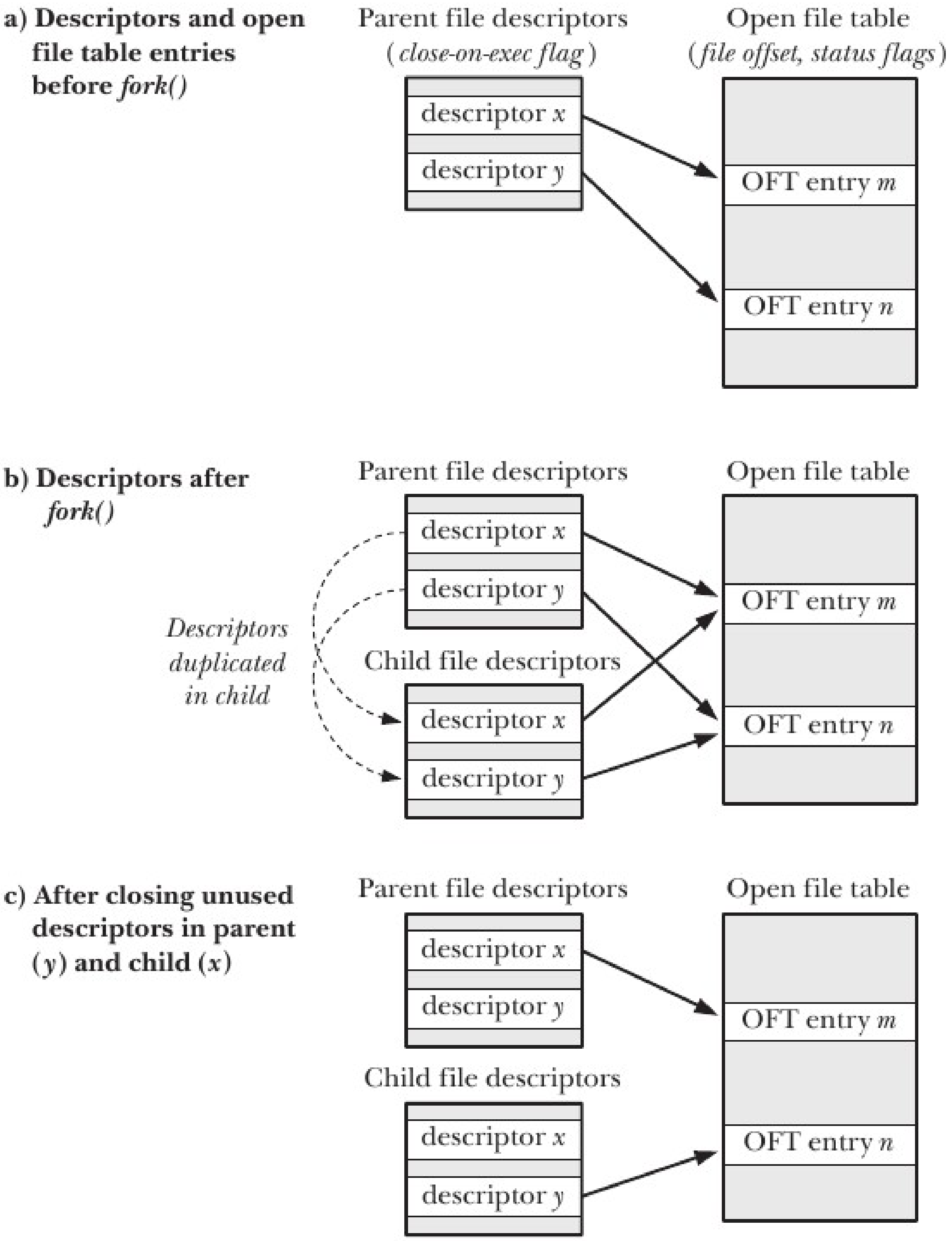


Figure 1.8 - copy – on – write

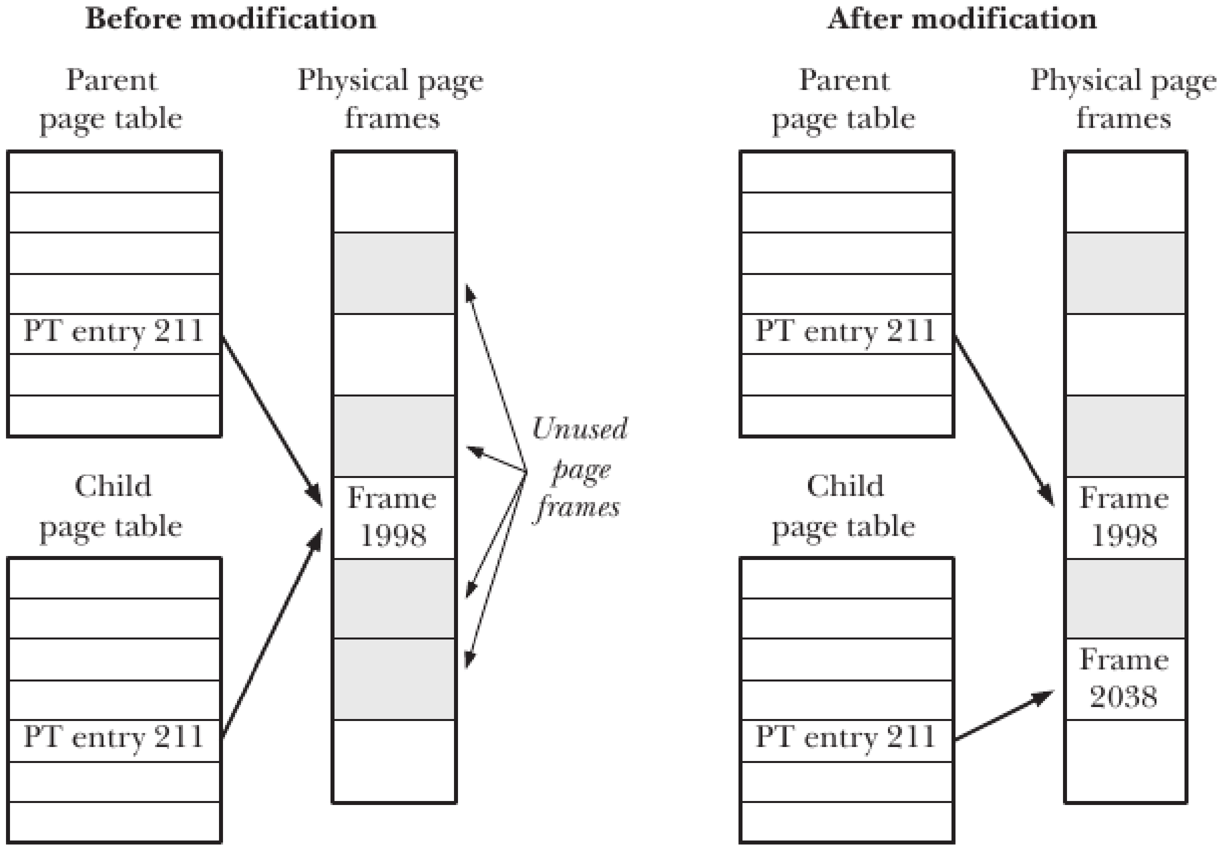


Figure 1.9 - status variable of wait() /waitpid()

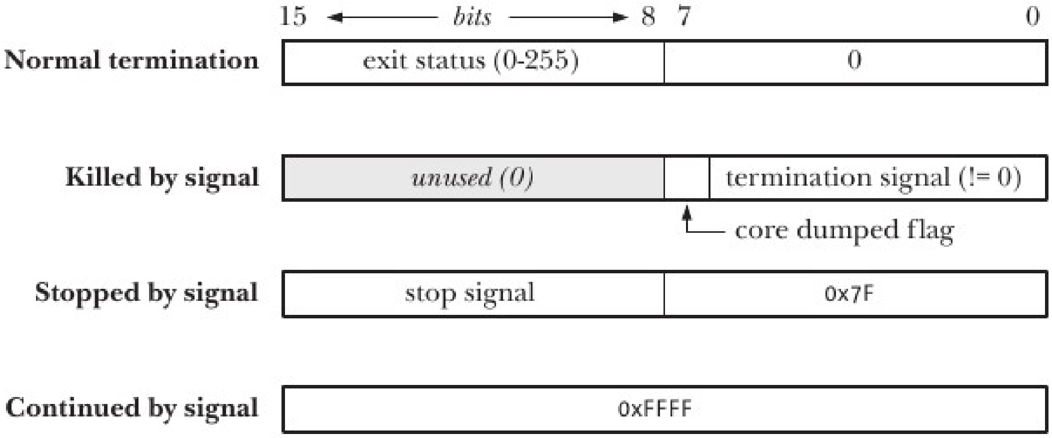
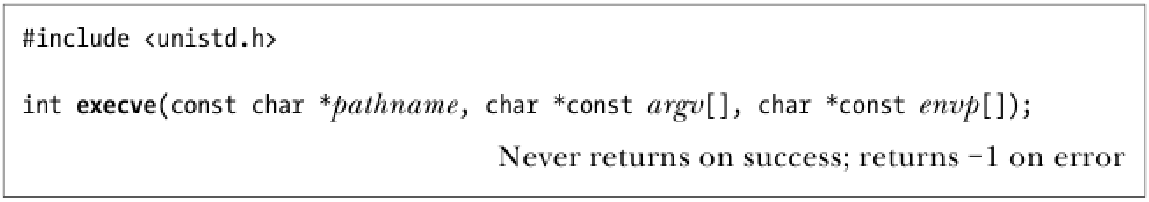


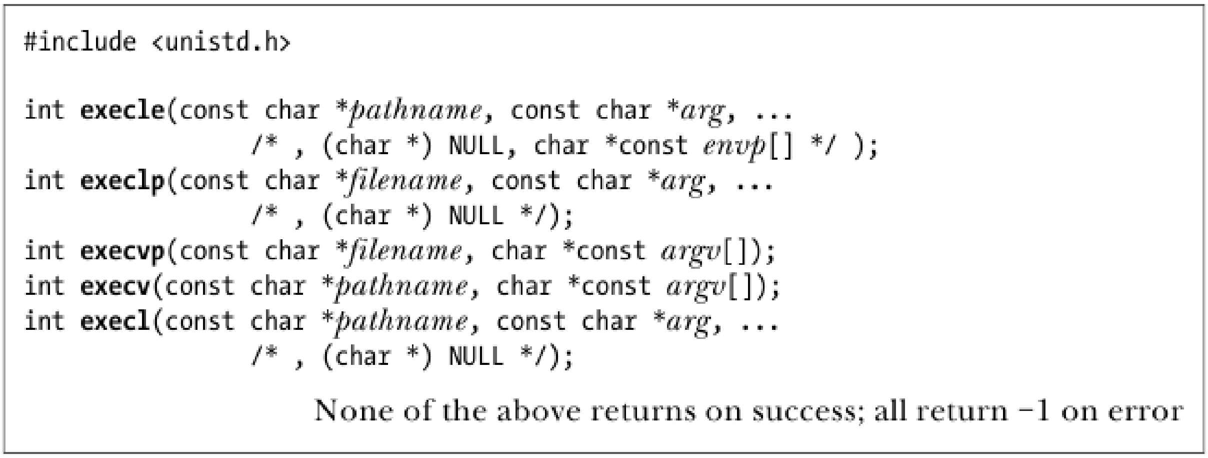
Figure 1.10 - execve() and related library APIs



```
#include <unistd.h>

int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Never returns on success; returns –1 on error

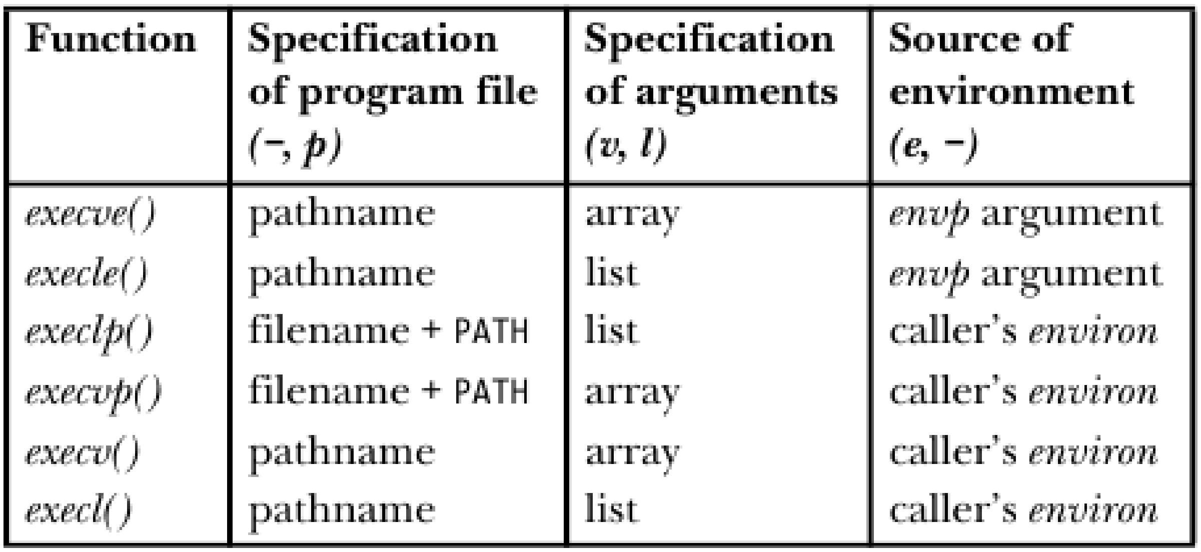


```
#include <unistd.h>

int execle(const char *pathname, const char *arg, ...
                /* , (char *) NULL, char *const envp[] */ );
int execlp(const char *filename, const char *arg, ...
                /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
                /* , (char *) NULL */);
```

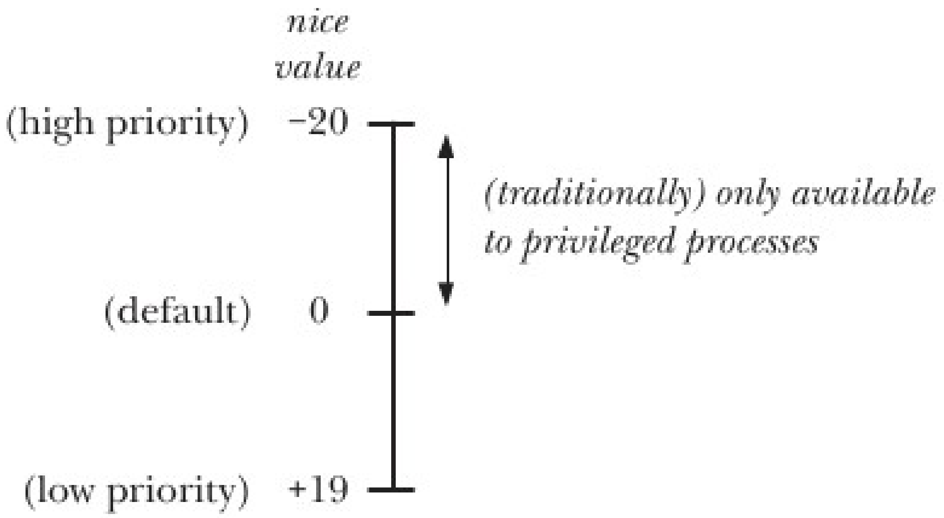
None of the above returns on success; all return –1 on error

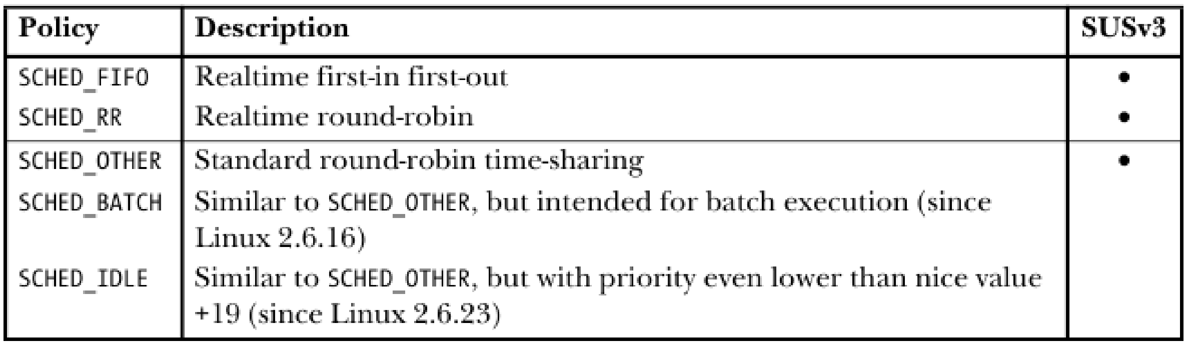
Figure 1.11 - execve() and library APIs continued...



Function	Specification of program file *(–, p)*	Specification of arguments *(v, l)*	Source of environment *(e, –)*
execve()	pathname	array	*envp* argument
execle()	pathname	list	*envp* argument
execlp()	filename + PATH	list	caller's *environ*
execvp()	filename + PATH	array	caller's *environ*
execv()	pathname	array	caller's *environ*
execl()	pathname	list	caller's *environ*

Figure 1.12 - nice priorities and scheduling policies





Policy	Description	SUSv3
SCHED_FIFO	Realtime first-in first-out	•
SCHED_RR	Realtime round-robin	•
SCHED_OTHER	Standard round-robin time-sharing	•
SCHED_BATCH	Similar to SCHED_OTHER, but intended for batch execution (since Linux 2.6.16)	
SCHED_IDLE	Similar to SCHED_OTHER, but with priority even lower than nice value +19 (since Linux 2.6.23)	

Figure 1.13 - various IPC mechanisms

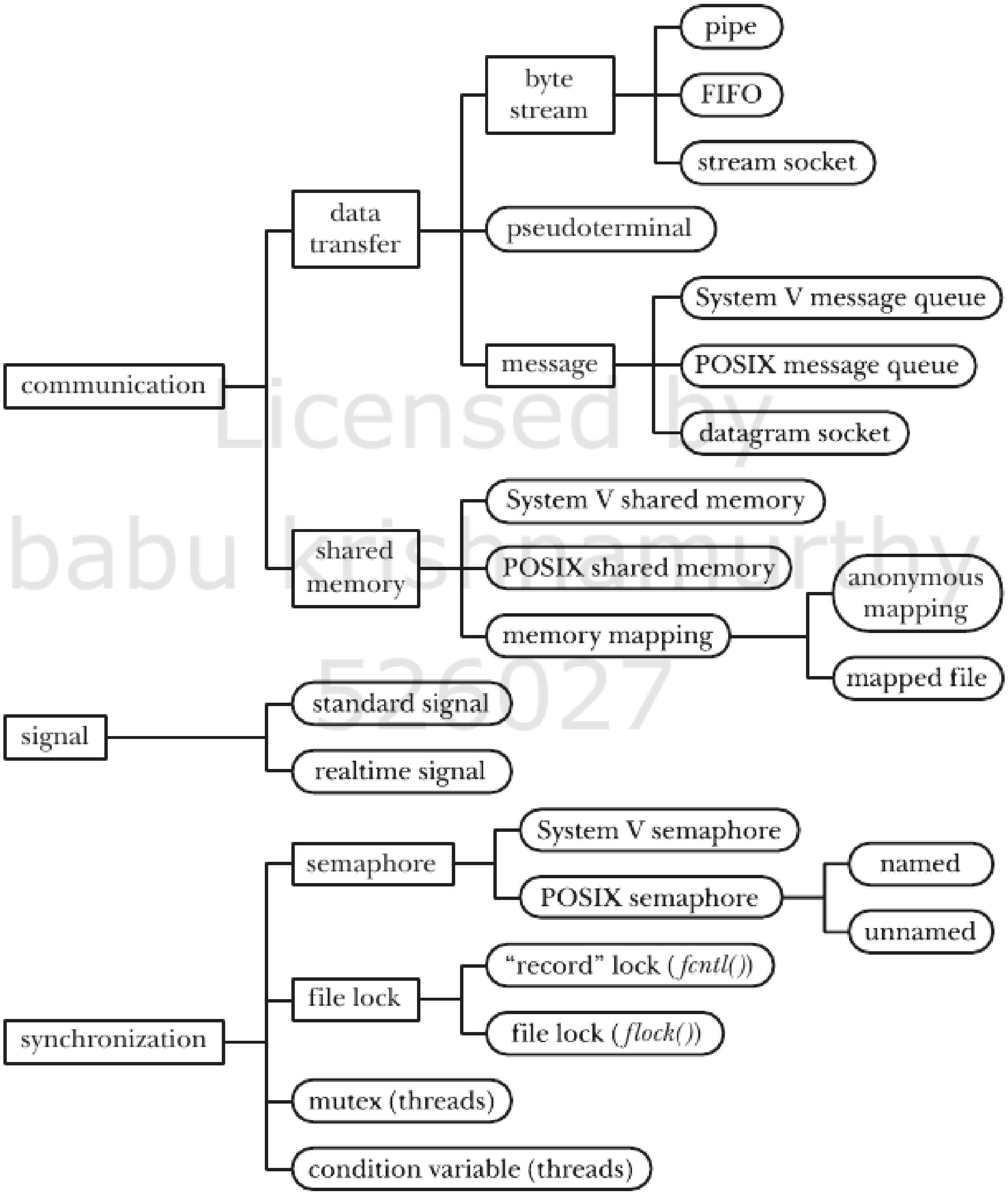
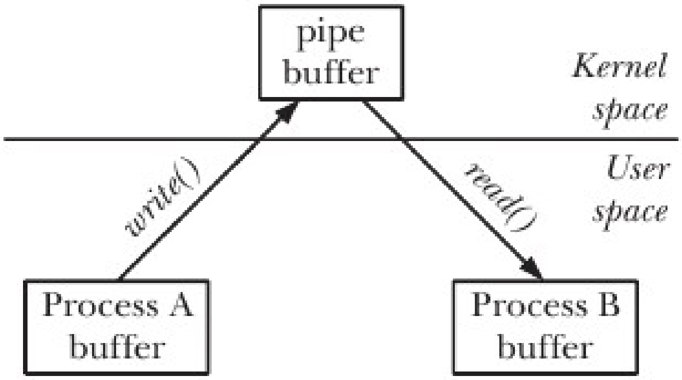


Figure 1.14 - pipe communication



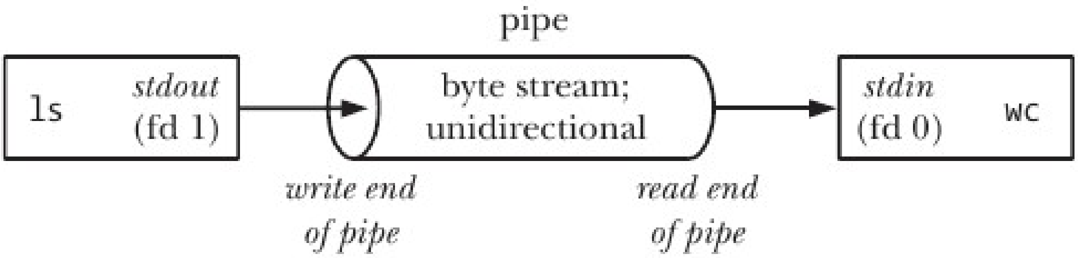
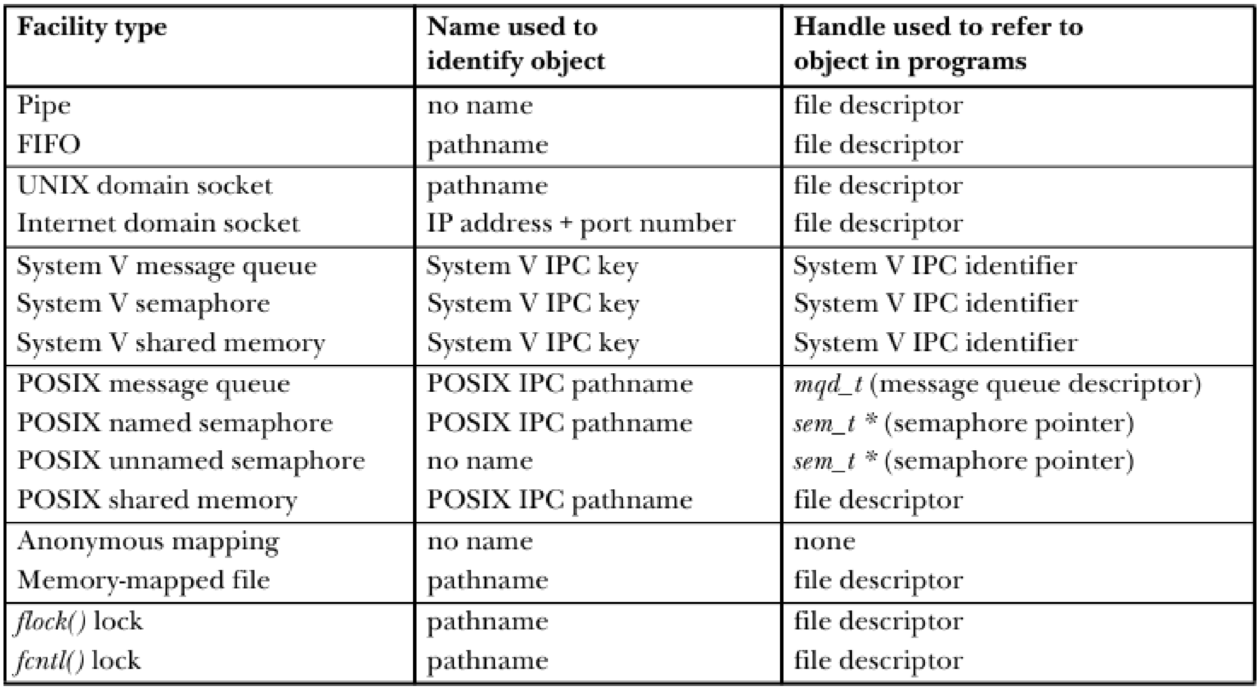
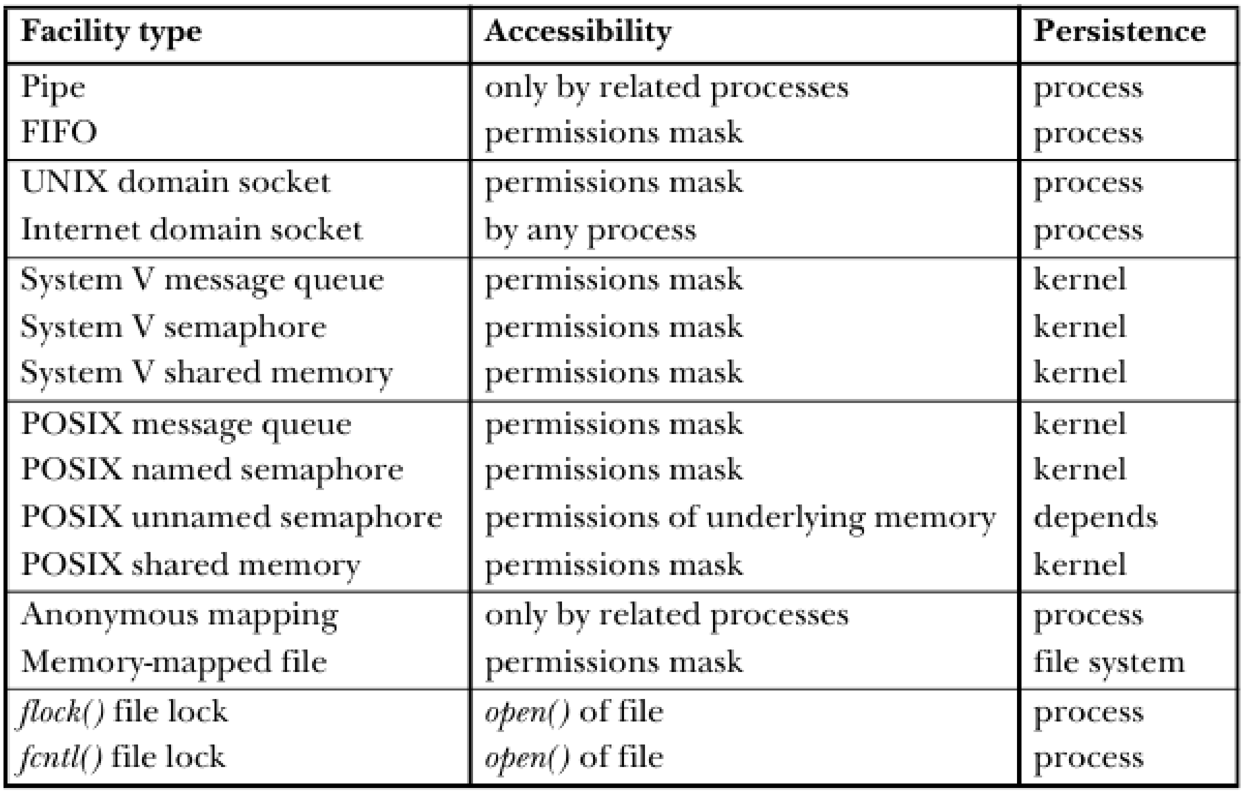


Figure 1.15 - summary of IPC mechanisms



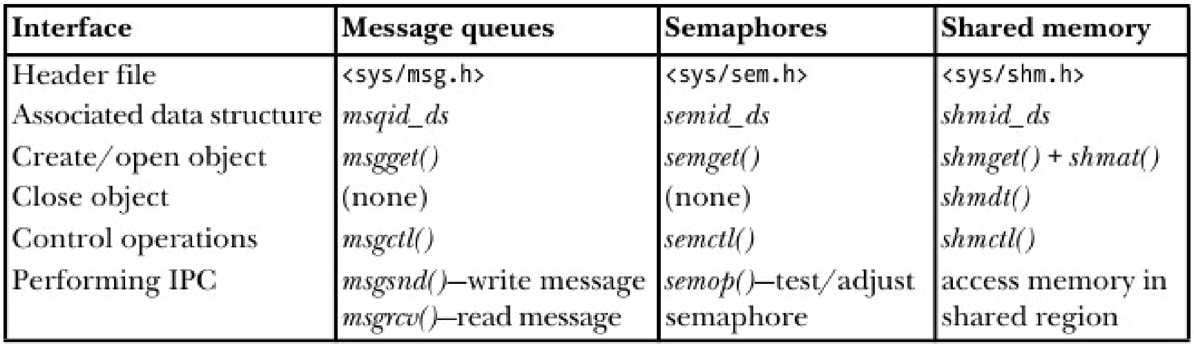
Facility type	Name used to identify object	Handle used to refer to object in programs
Pipe	no name	file descriptor
FIFO	pathname	file descriptor
UNIX domain socket	pathname	file descriptor
Internet domain socket	IP address + port number	file descriptor
System V message queue	System V IPC key	System V IPC identifier
System V semaphore	System V IPC key	System V IPC identifier
System V shared memory	System V IPC key	System V IPC identifier
POSIX message queue	POSIX IPC pathname	*mqd_t* (message queue descriptor)
POSIX named semaphore	POSIX IPC pathname	*sem_t* * (semaphore pointer)
POSIX unnamed semaphore	no name	*sem_t* * (semaphore pointer)
POSIX shared memory	POSIX IPC pathname	file descriptor
Anonymous mapping	no name	none
Memory-mapped file	pathname	file descriptor
flock() lock	pathname	file descriptor
fcntl() lock	pathname	file descriptor

Figure 1.16 - summary of IPC mechanisms



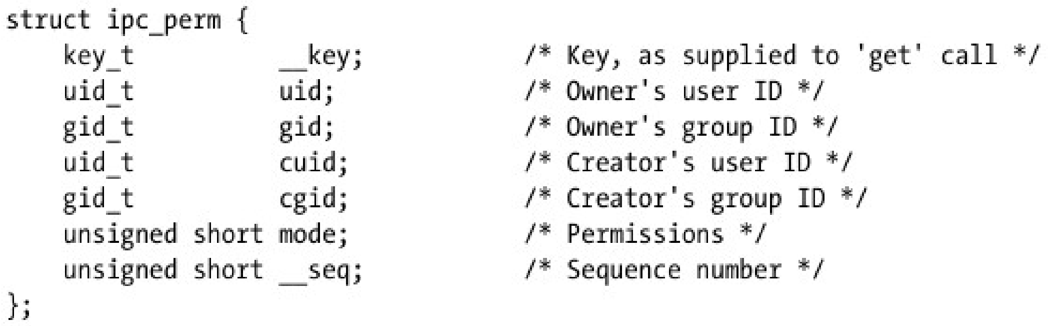
Facility type	Accessibility	Persistence
Pipe FIFO	only by related processes permissions mask	process process
UNIX domain socket Internet domain socket	permissions mask by any process	process process
System V message queue System V semaphore System V shared memory	permissions mask permissions mask permissions mask	kernel kernel kernel
POSIX message queue POSIX named semaphore POSIX unnamed semaphore POSIX shared memory	permissions mask permissions mask permissions of underlying memory permissions mask	kernel kernel depends kernel
Anonymous mapping Memory-mapped file	only by related processes permissions mask	process file system
flock() file lock *fcntl()* file lock	*open()* of file *open()* of file	process process

Figure 1.17 - POSIX SYS V IPC mechanisms



Interface	**Message queues**	**Semaphores**	**Shared memory**
Header file	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Associated data structure	*msqid_ds*	*semid_ds*	*shmid_ds*
Create/open object	*msgget()*	*semget()*	*shmget()* + *shmat()*
Close object	(none)	(none)	*shmdt()*
Control operations	*msgctl()*	*semctl()*	*shmctl()*
Performing IPC	*msgsnd()*—write message *msgrcv()*—read message	*semop()*—test/adjust semaphore	access memory in shared region

Figure 1.18 - struct ipc_perm { }, struct ipc_ids{ } and struct semid_ds { }



```
struct ipc_perm {
    key_t          __key;          /* Key, as supplied to 'get' call */
    uid_t          uid;            /* Owner's user ID */
    gid_t          gid;            /* Owner's group ID */
    uid_t          cuid;           /* Creator's user ID */
    gid_t          cgid;           /* Creator's group ID */
    unsigned short mode;           /* Permissions */
    unsigned short __seq;          /* Sequence number */
};
```

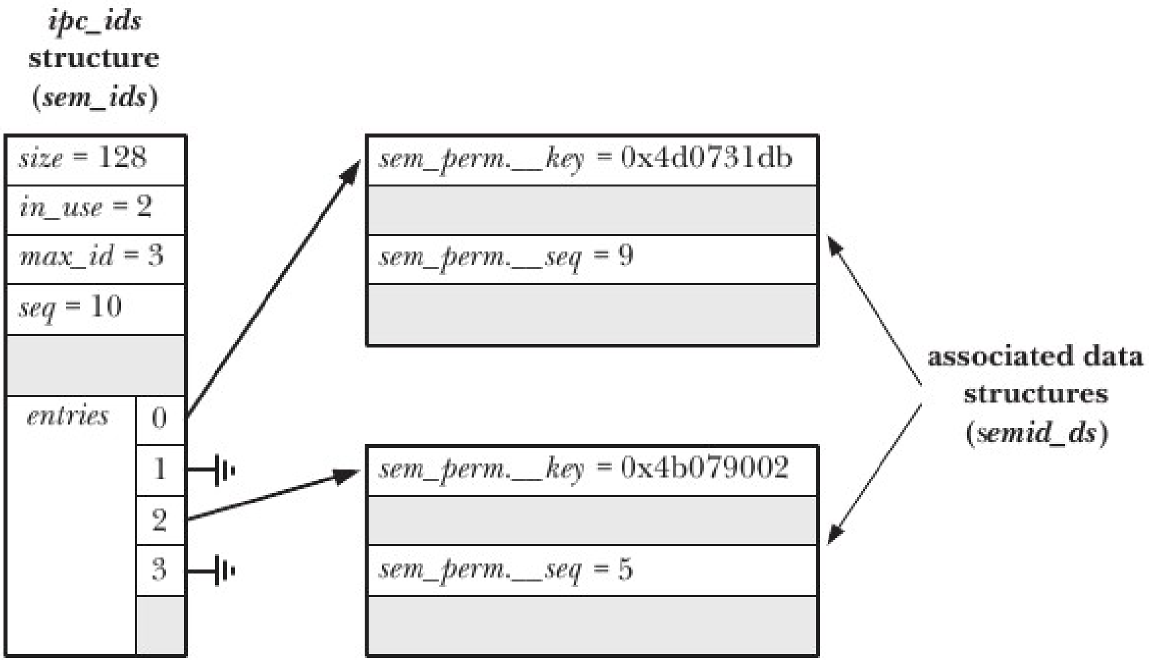
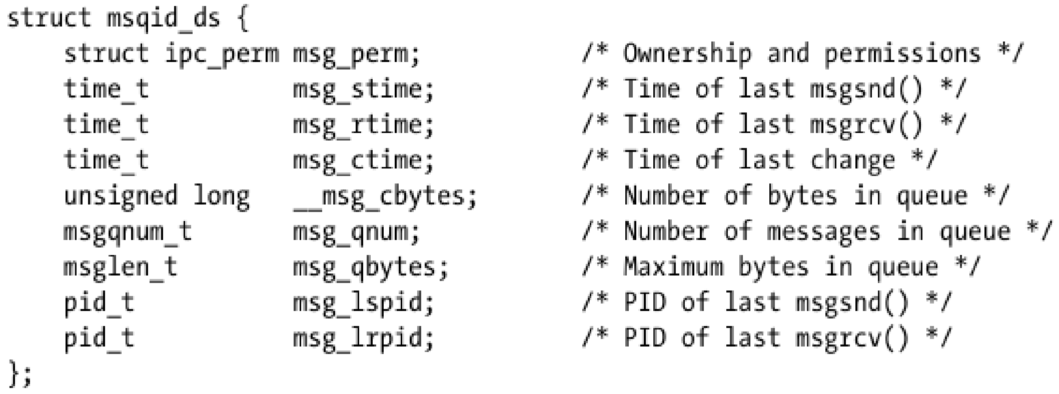


Figure 1.19 - POSIX SYS V message queue related objects



```
struct msqid_ds {
    struct ipc_perm msg_perm;           /* Ownership and permissions */
    time_t          msg_stime;          /* Time of last msgsnd() */
    time_t          msg_rtime;          /* Time of last msgrcv() */
    time_t          msg_ctime;          /* Time of last change */
    unsigned long   __msg_cbytes;       /* Number of bytes in queue */
    msgqnum_t       msg_qnum;           /* Number of messages in queue */
    msglen_t        msg_qbytes;         /* Maximum bytes in queue */
    pid_t           msg_lspid;          /* PID of last msgsnd() */
    pid_t           msg_lrpid;          /* PID of last msgrcv() */
};
```

Figure 1.20 - a POSIX SYS V message queue usage scenario

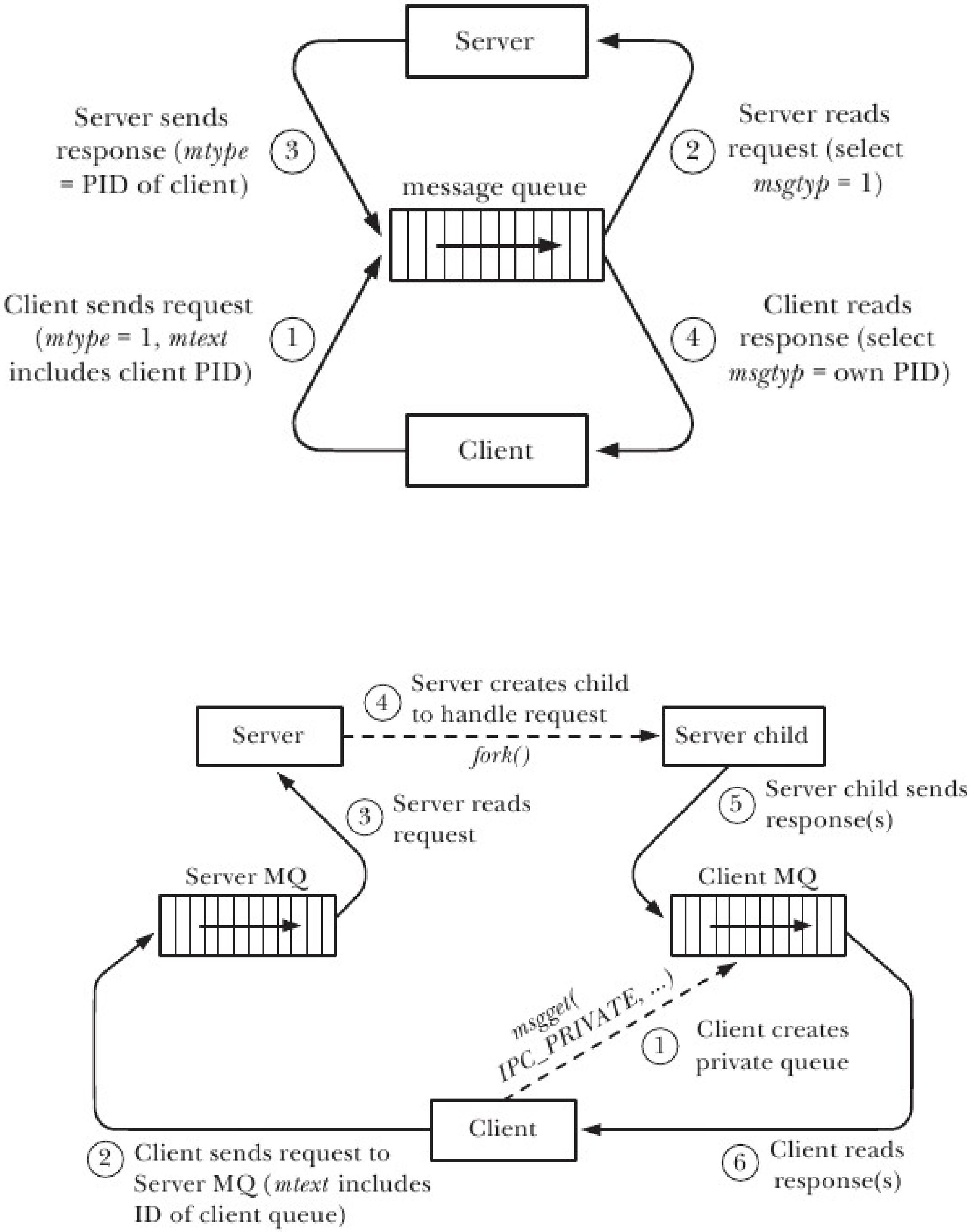
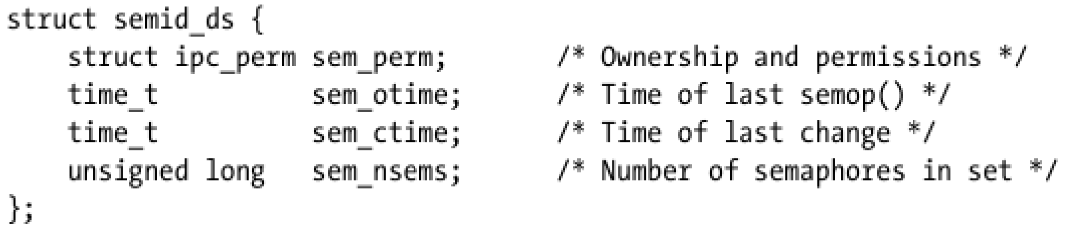

Server
Server sends response (mtype = PID of client)
3
Server reads request (select msgtyp = 1)
2
message queue
Client sends request (mtype = 1, mtext includes client PID)
1
Client reads response (select msgtyp = own PID)
4
Client
Server
4
Server creates child to handle request
fork()
Server child
3
Server reads request
5
Server child sends response(s)
Server MQ
Client MQ
msgget(IPC_PRIVATE, ...)
1
Client creates private queue
Client
2
Client sends request to Server MQ (mtext includes ID of client queue)
6
Client reads response(s)

Figure 1.21 - POSIX SYS V semaphore IPC OBJECT



```
struct semid_ds {
    struct ipc_perm sem_perm;       /* Ownership and permissions */
    time_t          sem_otime;      /* Time of last semop() */
    time_t          sem_ctime;      /* Time of last change */
    unsigned long   sem_nsems;      /* Number of semaphores in set */
};
```

Figure 1.22 - subtle SYS V semaphore initialization problem

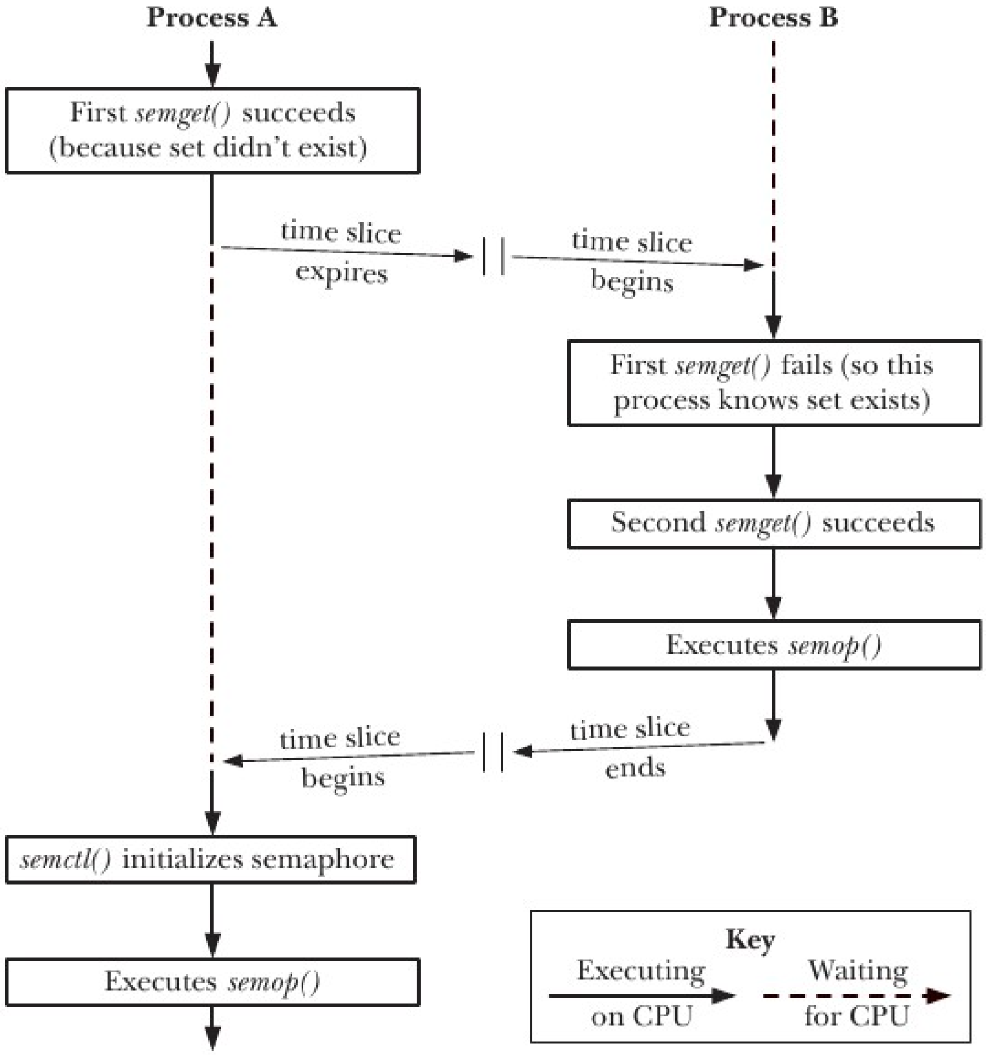

Process A
Process B
First *semget()* succeeds (because set didn't exist)
time slice expires
time slice begins
First *semget()* fails (so this process knows set exists)
Second *semget()* succeeds
Executes *semop()*
time slice begins
time slice ends
semctl() initializes semaphore
Executes *semop()*
Key
Executing on CPU
Waiting for CPU

Figure 1.23 - shared memory mappings

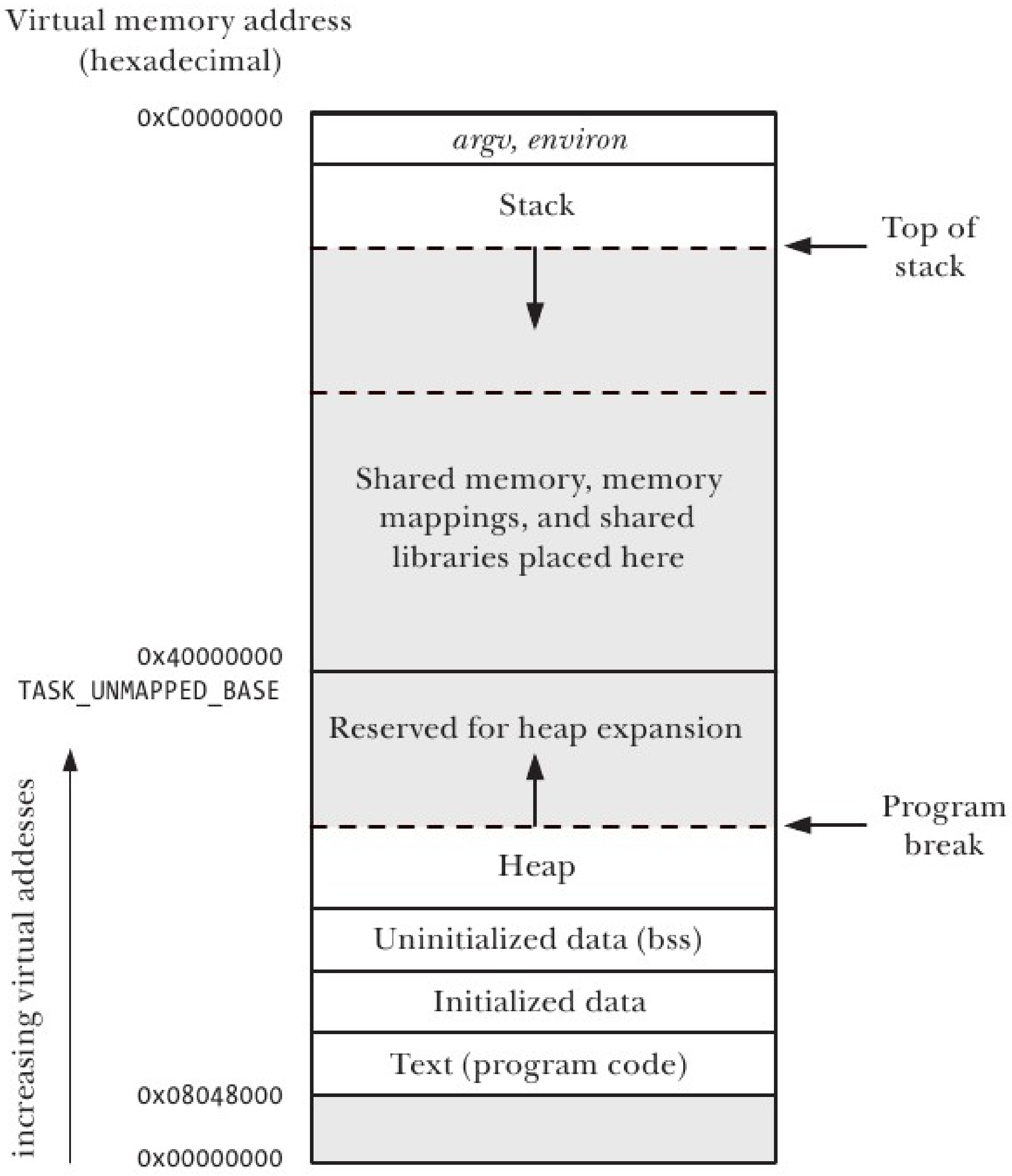
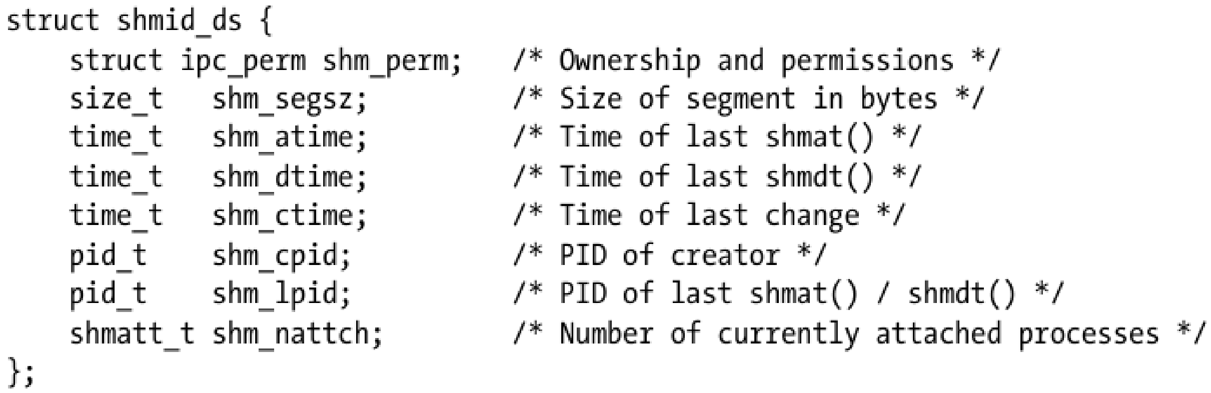
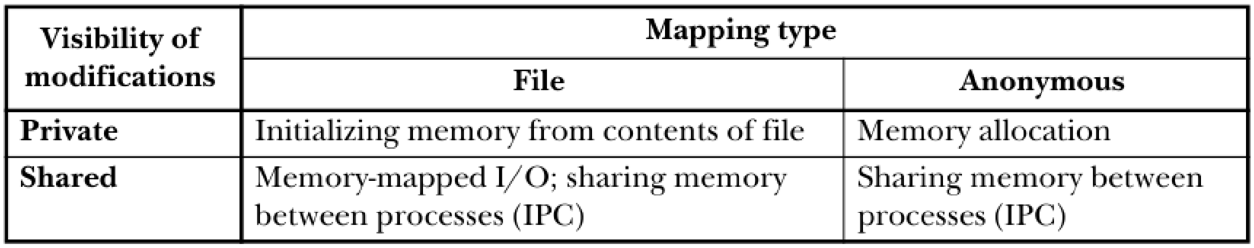


Figure 1.24 - SYSV shared memory IPC object



```
struct shmid_ds {
    struct ipc_perm shm_perm;   /* Ownership and permissions */
    size_t   shm_segsz;         /* Size of segment in bytes */
    time_t   shm_atime;         /* Time of last shmat() */
    time_t   shm_dtime;         /* Time of last shmdt() */
    time_t   shm_ctime;         /* Time of last change */
    pid_t    shm_cpid;          /* PID of creator */
    pid_t    shm_lpid;          /* PID of last shmat() / shmdt() */
    shmatt_t shm_nattch;        /* Number of currently attached processes */
};
```

Figure 1.25 - memory mappings



Visibility of modifications	Mapping type	
	File	**Anonymous**
Private	Initializing memory from contents of file	Memory allocation
Shared	Memory-mapped I/O; sharing memory between processes (IPC)	Sharing memory between processes (IPC)

Figure 1.26 - file memory – mapping

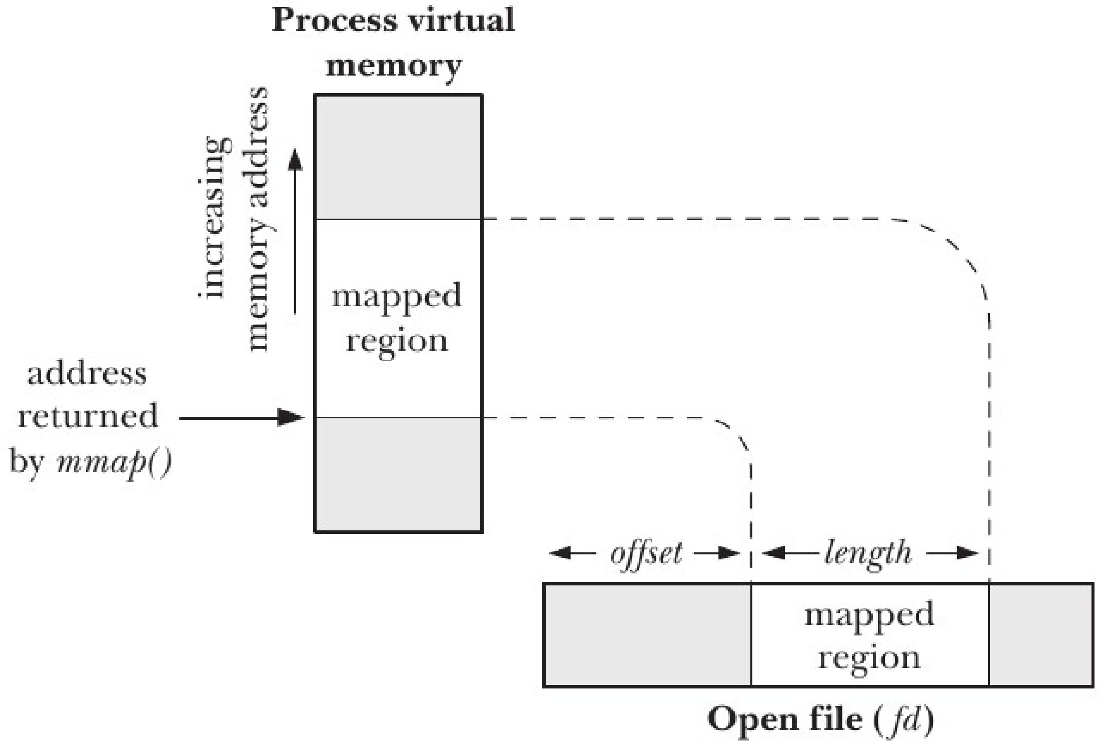


Figure 1.27 - shared file memory mapping

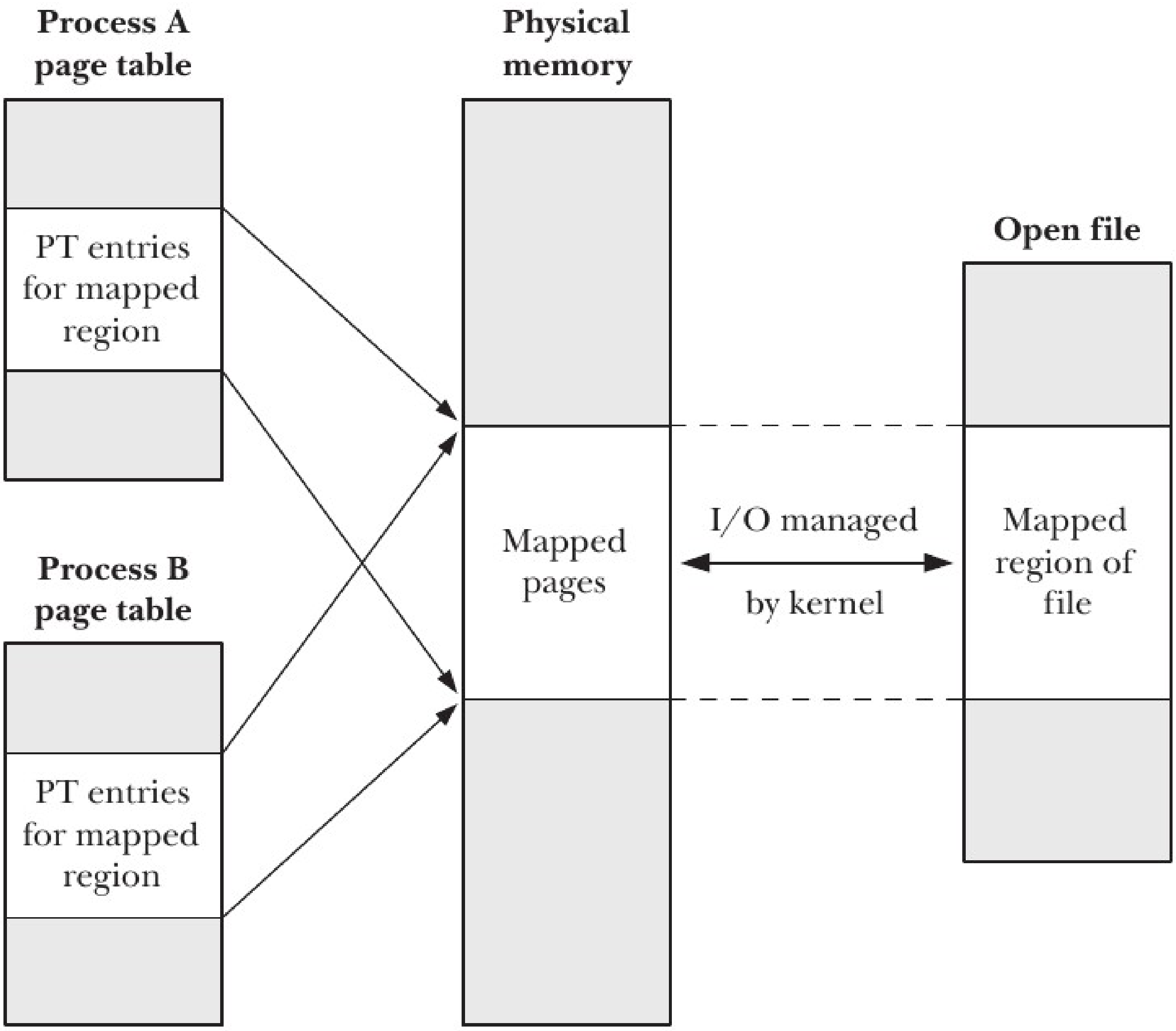


Figure 1.28 - file mapping access errors

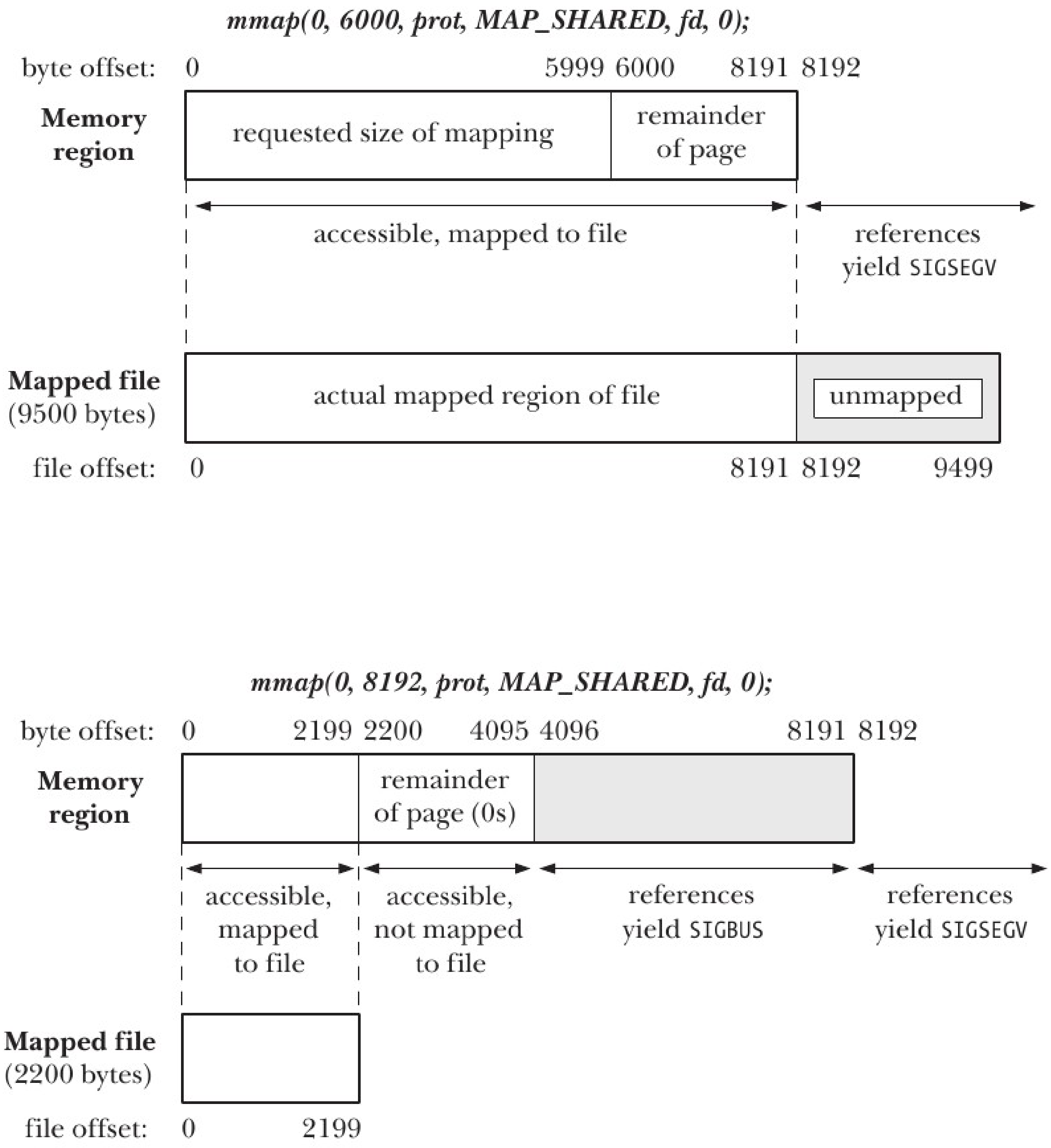
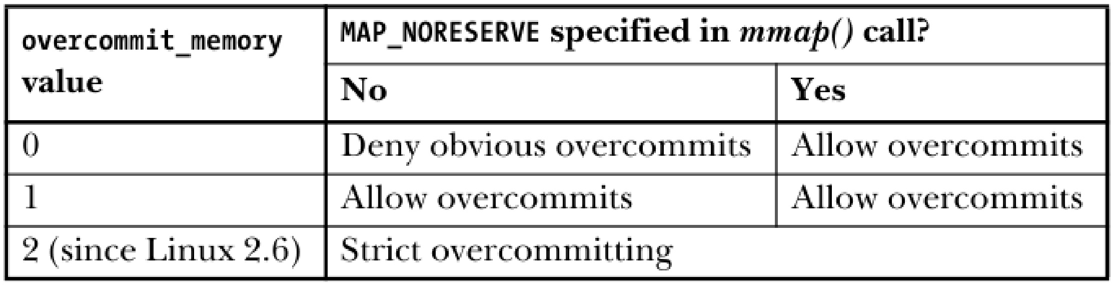
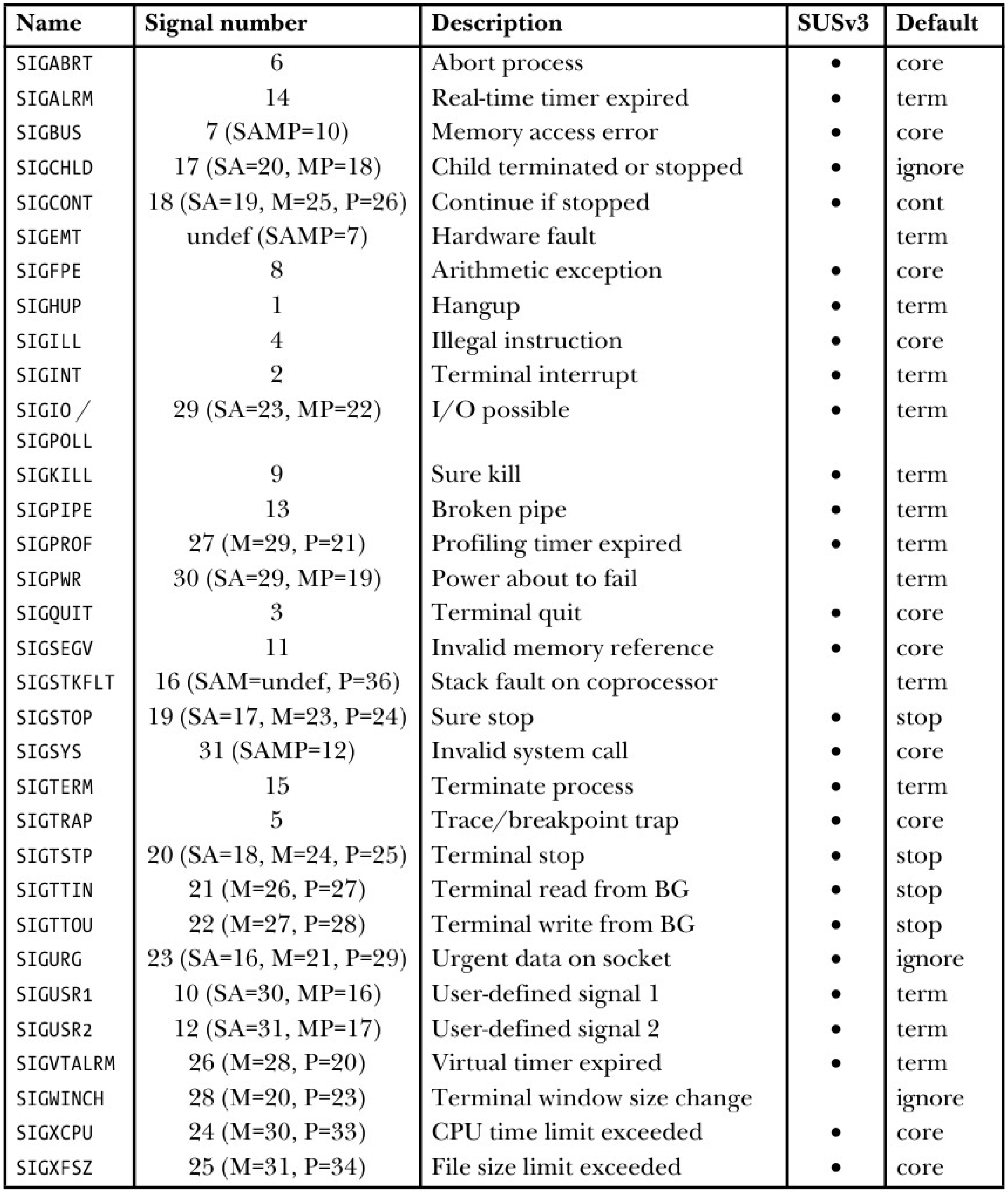


Figure 1.29 - mmap and virtual memory issues



`overcommit_memory` **value**	`MAP_NORESERVE` **specified in *mmap()* call?**	
	No	**Yes**
0	Deny obvious overcommits	Allow overcommits
1	Allow overcommits	Allow overcommits
2 (since Linux 2.6)	Strict overcommitting	

Figure 1.30 - different types of signals



Name	Signal number	Description	SUSv3	Default
SIGABRT	6	Abort process	•	core
SIGALRM	14	Real-time timer expired	•	term
SIGBUS	7 (SAMP=10)	Memory access error	•	core
SIGCHLD	17 (SA=20, MP=18)	Child terminated or stopped	•	ignore
SIGCONT	18 (SA=19, M=25, P=26)	Continue if stopped	•	cont
SIGEMT	undef (SAMP=7)	Hardware fault		term
SIGFPE	8	Arithmetic exception	•	core
SIGHUP	1	Hangup	•	term
SIGILL	4	Illegal instruction	•	core
SIGINT	2	Terminal interrupt	•	term
SIGIO / SIGPOLL	29 (SA=23, MP=22)	I/O possible	•	term
SIGKILL	9	Sure kill	•	term
SIGPIPE	13	Broken pipe	•	term
SIGPROF	27 (M=29, P=21)	Profiling timer expired	•	term
SIGPWR	30 (SA=29, MP=19)	Power about to fail		term
SIGQUIT	3	Terminal quit	•	core
SIGSEGV	11	Invalid memory reference	•	core
SIGSTKFLT	16 (SAM=undef, P=36)	Stack fault on coprocessor		term
SIGSTOP	19 (SA=17, M=23, P=24)	Sure stop	•	stop
SIGSYS	31 (SAMP=12)	Invalid system call	•	core
SIGTERM	15	Terminate process	•	term
SIGTRAP	5	Trace/breakpoint trap	•	core
SIGTSTP	20 (SA=18, M=24, P=25)	Terminal stop	•	stop
SIGTTIN	21 (M=26, P=27)	Terminal read from BG	•	stop
SIGTTOU	22 (M=27, P=28)	Terminal write from BG	•	stop
SIGURG	23 (SA=16, M=21, P=29)	Urgent data on socket	•	ignore
SIGUSR1	10 (SA=30, MP=16)	User-defined signal 1	•	term
SIGUSR2	12 (SA=31, MP=17)	User-defined signal 2	•	term
SIGVTALRM	26 (M=28, P=20)	Virtual timer expired	•	term
SIGWINCH	28 (M=20, P=23)	Terminal window size change		ignore
SIGXCPU	24 (M=30, P=33)	CPU time limit exceeded	•	core
SIGXFSZ	25 (M=31, P=34)	File size limit exceeded	•	core

Figure 1.31 - signal handling and process/program flow

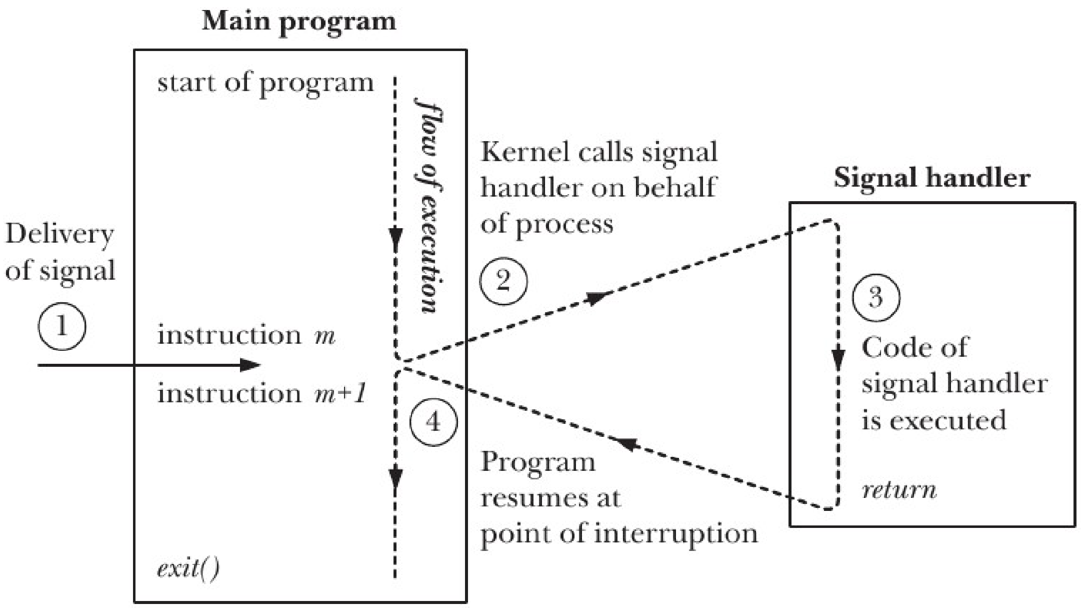

Main program
start of program
flow of execution
Delivery of signal
1
instruction m
instruction m+1
exit()
Kernel calls signal handler on behalf of process
2
Signal handler
3
Code of signal handler is executed
return
4
Program resumes at point of interruption

Figure 1.32 - permission to send a signal

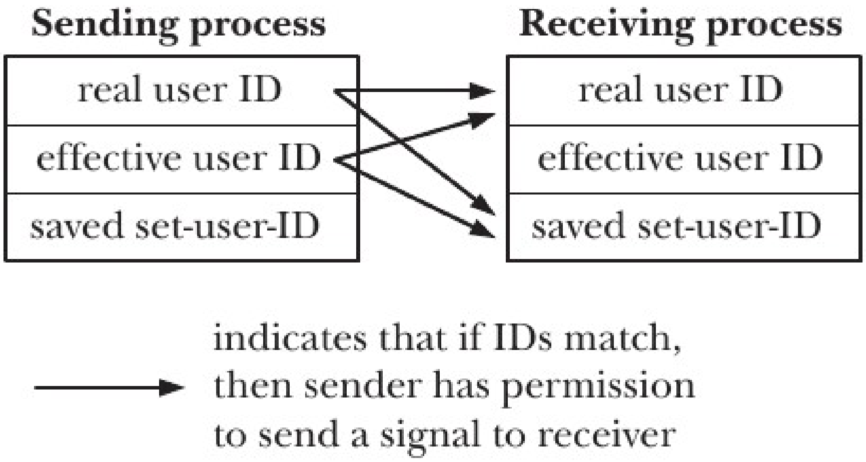
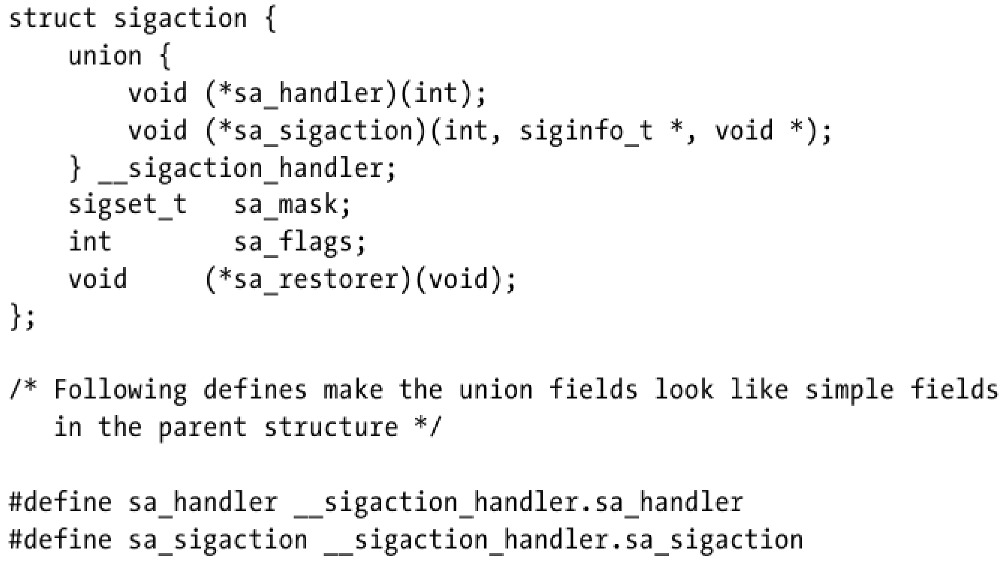


Figure 1.33 - async signal safe library functions

_Exit() (v3)
_exit()
abort() (v3)
accept() (v3)
access()
aio_error() (v2)
aio_return() (v2)
aio_suspend() (v2)
alarm()
bind() (v3)
cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
chdir()
chmod()
chown()
clock_gettime() (v2)
close()
connect() (v3)
creat()
dup()
dup2()
execle()
execve()
fchmod() (v3)
fchown() (v3)
fcntl()
fdatasync() (v2)
fork()
fpathconf() (v2)
fstat()
fsync() (v2)
ftruncate() (v3)
getegid()
geteuid()
getgid()
getgroups()
getpeername() (v3)
getpgrp()
getpid()
getppid()
getsockname() (v3)
getsockopt() (v3)
getuid()
kill()
link()
listen() (v3)
lseek()
lstat() (v3)
mkdir()
mkfifo()
open()
pathconf()
pause()
pipe()
poll() (v3)
posix_trace_event() (v3)
pselect() (v3)
raise() (v2)
read()
readlink() (v3)
recv() (v3)
recvfrom() (v3)
recvmsg() (v3)
rename()
rmdir()
select() (v3)
sem_post() (v2)
send() (v3)
sendmsg() (v3)
sendto() (v3)
setgid()
setpgid()
setsid()
setsockopt() (v3)
setuid()
shutdown() (v3)
sigaction()
sigaddset()
sigdelset()
sigemptyset()
sigfillset()
sigismember()
signal() (v2)
sigpause() (v2)
sigpending()
sigprocmask()
sigqueue() (v2)
sigset() (v2)
sigsuspend()
sleep()
socket() (v3)
sockatmark() (v3)
socketpair() (v3)
stat()
symlink() (v3)
sysconf()
tcdrain()
tcflow()
tcflush()
tcgetattr()
tcgetpgrp()
tcsendbreak()
tcsetattr()
tcsetpgrp()
time()
timer_getoverrun() (v2)
timer_gettime() (v2)
timer_settime() (v2)
times()
umask()
uname()
unlink()
utime()
wait()
waitpid()
write()

Figure 1.34 - struct sigaction { } and supported handlers



```
struct sigaction {
    union {
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
    } __sigaction_handler;
    sigset_t   sa_mask;
    int        sa_flags;
    void      (*sa_restorer)(void);
};

/* Following defines make the union fields look like simple fields
   in the parent structure */

#define sa_handler __sigaction_handler.sa_handler
#define sa_sigaction __sigaction_handler.sa_sigaction
```

Figure 1.35 - a special case of nested signal handling

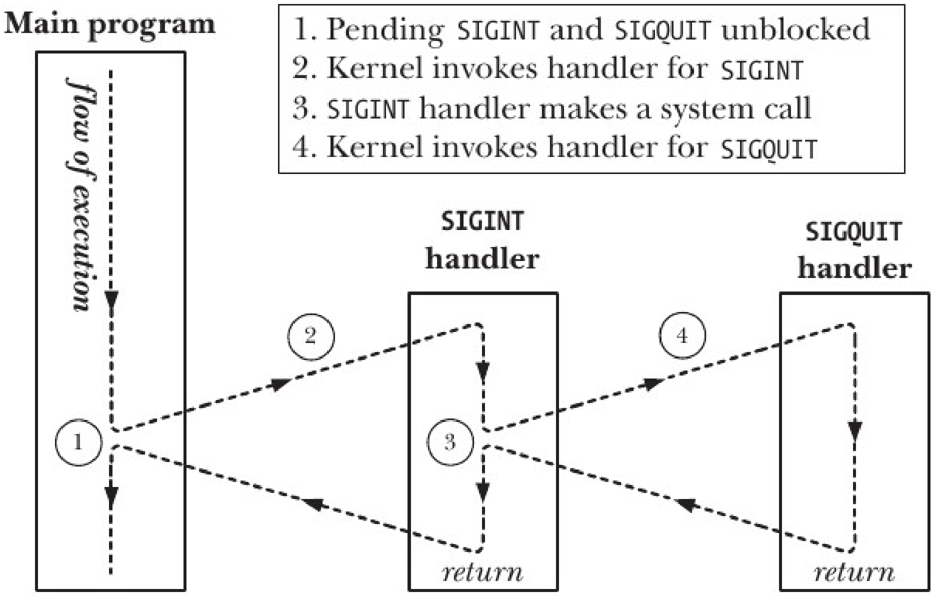

Main program
1. Pending SIGINT and SIGQUIT unblocked
2. Kernel invokes handler for SIGINT
3. SIGINT handler makes a system call
4. Kernel invokes handler for SIGQUIT
flow of execution
SIGINT handler
SIGQUIT handler
1
2
3
4
return
return

Figure 1.36 - threads in process address space

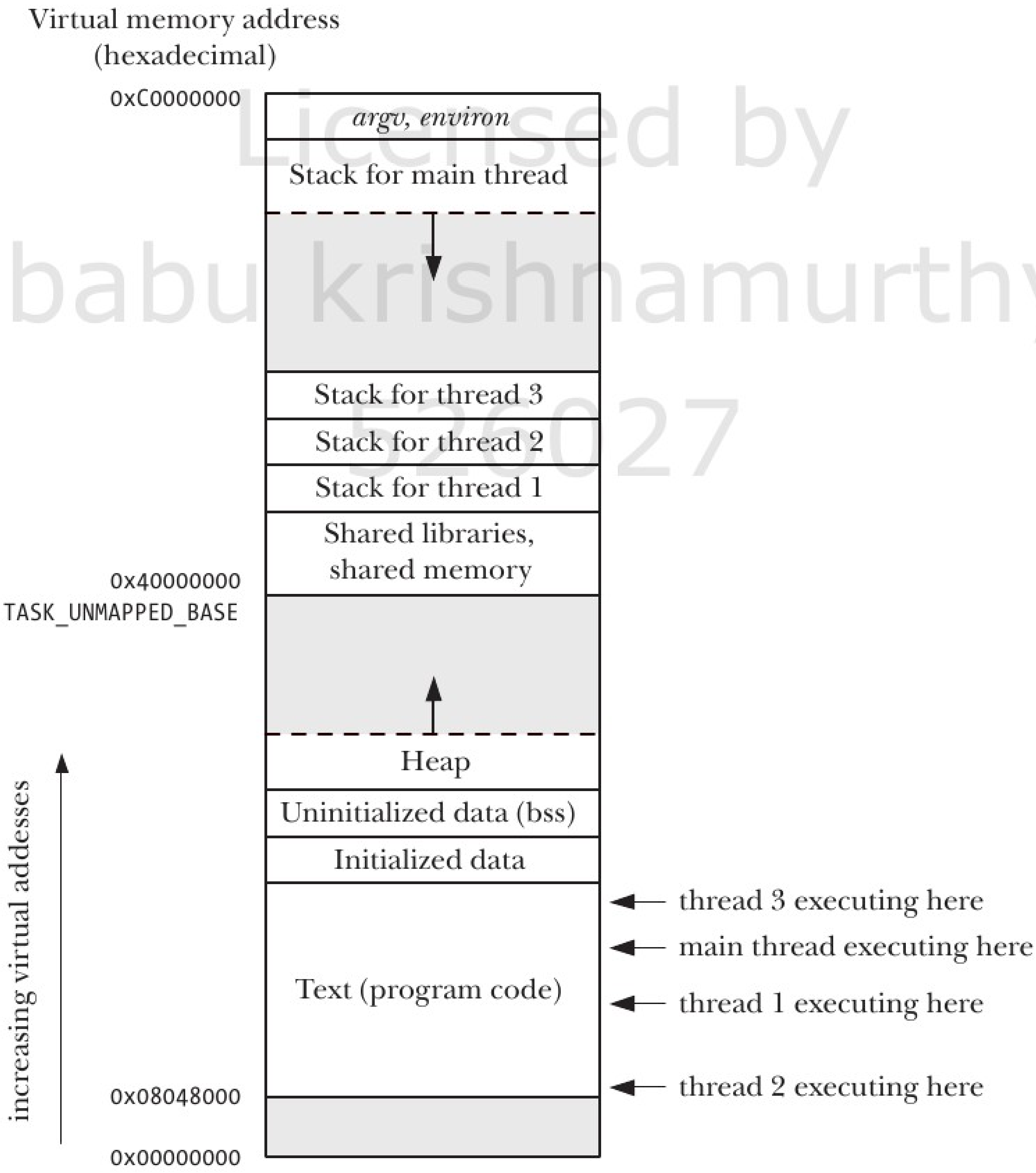
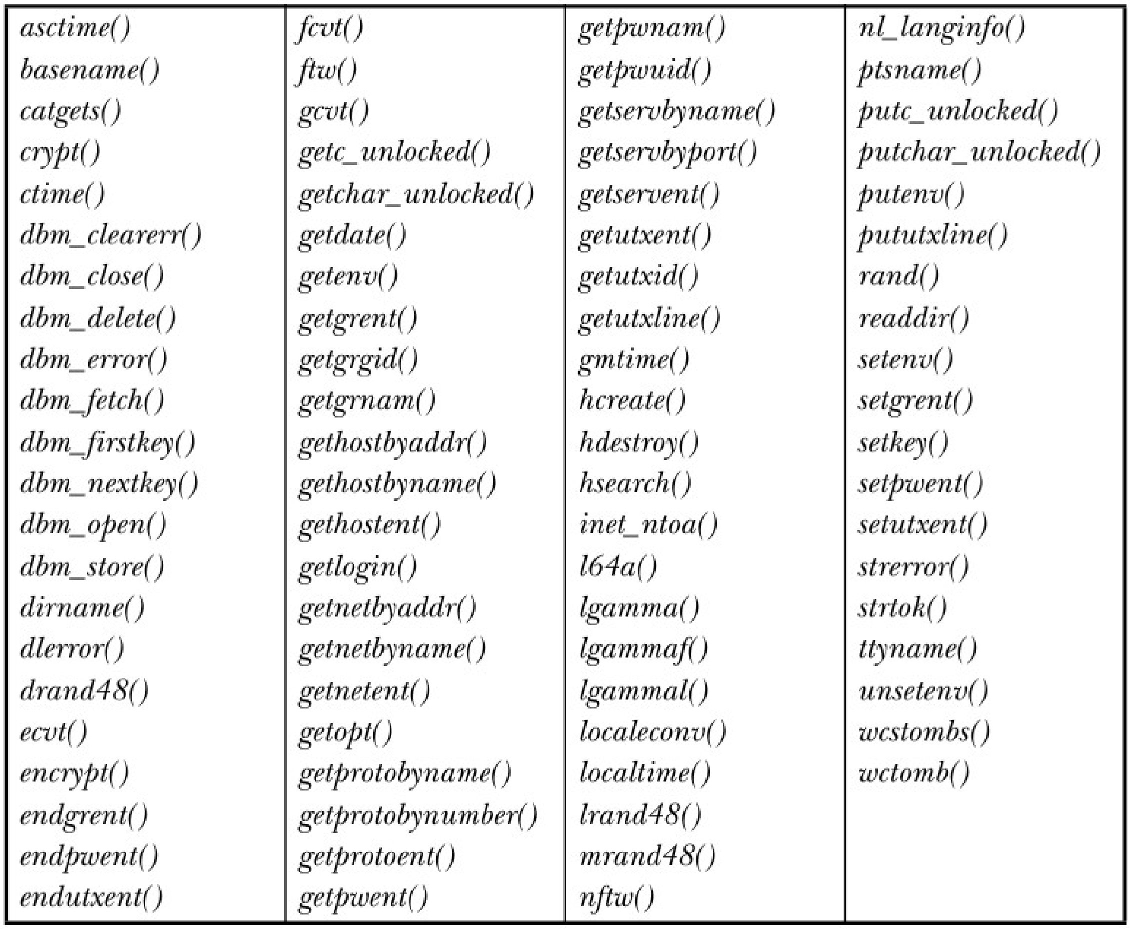
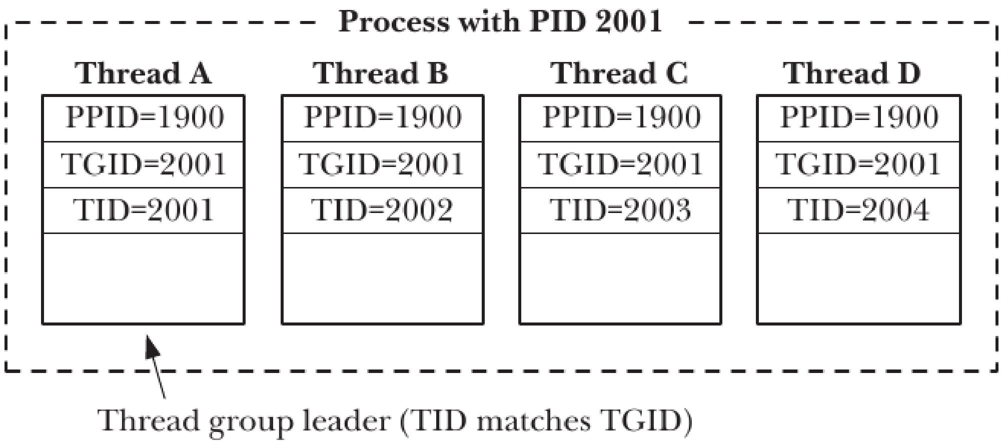


Figure 1.37 - non thread-safe library functions



- *asctime()*
- *basename()*
- *catgets()*
- *crypt()*
- *ctime()*
- *dbm_clearerr()*
- *dbm_close()*
- *dbm_delete()*
- *dbm_error()*
- *dbm_fetch()*
- *dbm_firstkey()*
- *dbm_nextkey()*
- *dbm_open()*
- *dbm_store()*
- *dirname()*
- *dlerror()*
- *drand48()*
- *ecvt()*
- *encrypt()*
- *endgrent()*
- *endpwent()*
- *endutxent()*
- *fcvt()*
- *ftw()*
- *gcvt()*
- *getc_unlocked()*
- *getchar_unlocked()*
- *getdate()*
- *getenv()*
- *getgrent()*
- *getgrgid()*
- *getgrnam()*
- *gethostbyaddr()*
- *gethostbyname()*
- *gethostent()*
- *getlogin()*
- *getnetbyaddr()*
- *getnetbyname()*
- *getnetent()*
- *getopt()*
- *getprotobyname()*
- *getprotobynumber()*
- *getprotoent()*
- *getpwent()*
- *getpwnam()*
- *getpwuid()*
- *getservbyname()*
- *getservbyport()*
- *getservent()*
- *getutxent()*
- *getutxid()*
- *getutxline()*
- *gmtime()*
- *hcreate()*
- *hdestroy()*
- *hsearch()*
- *inet_ntoa()*
- *l64a()*
- *lgamma()*
- *lgammaf()*
- *lgammal()*
- *localeconv()*
- *localtime()*
- *lrand48()*
- *mrand48()*
- *nftw()*
- *nl_langinfo()*
- *ptsname()*
- *putc_unlocked()*
- *putchar_unlocked()*
- *putenv()*
- *pututxline()*
- *rand()*
- *readdir()*
- *setenv()*
- *setgrent()*
- *setkey()*
- *setpwent()*
- *setutxent()*
- *strerror()*
- *strtok()*
- *ttyname()*
- *unsetenv()*
- *wcstombs()*
- *wctomb()*

Figure 1.38 - Threads in Linux and Ids



Process with PID 2001
Thread A
PPID=1900
TGID=2001
TID=2001
Thread B
PPID=1900
TGID=2001
TID=2002
Thread C
PPID=1900
TGID=2001
TID=2003
Thread D
PPID=1900
TGID=2001
TID=2004
Thread group leader (TID matches TGID)