Betriebssysteme und Systemnahe Programmierung

Kapitel 12 • Process Control

Winter 2016/17

Marcel Waldvogel

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.

Figure 4-36. The steps required to carry out the fork system call.

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.

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

EXEC System Call (2)

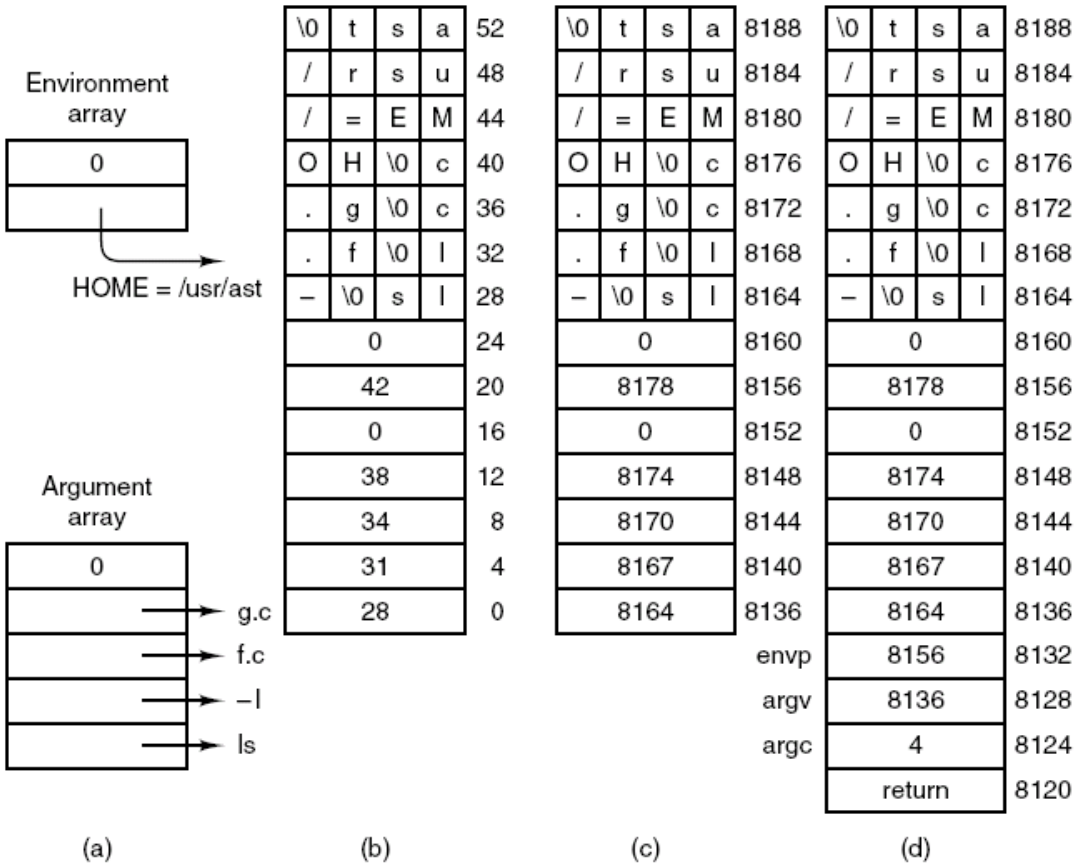

Figure 4-38. (a) The arrays passed to *execve*. (b) The stack built by *execve*. (c) The stack after relocation by the PM. (d) The stack as it appears to *main* at start of execution.

EXEC System Call (3)

```
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 4-39. The key part of *crtso*, the C run-time, start-off routine.

Signal Handling (1)

Preparation: program code prepares for possible signal.
Response: signal is received and action is taken.
Cleanup: restore normal operation of the process.

Figure 4-40. Three phases of dealing with signals.

Signal Handling (2)

```
struct sigaction {
    __sighandler_t sa_handler;   /* SIG_DFL, SIG_IGN, SIG_MESS,
                                    or pointer to function */
   sigset_t sa_mask;             /* signals to be blocked during handler */
   int sa_flags;                 /* special flags */
}
```

Figure 4-41. The sigaction structure.

Signal Handling (3)

Signal	Description	Generated by
SIGHUP	Hangup	KILL system call
SIGINT	Interrupt	TTY
SIGQUIT	Quit	TTY
SIGILL	Illegal instruction	Kernel (*)
SIGTRAP	Trace trap	Kernel (M)
SIGABRT	Abnormal termination	TTY
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	FS

Figure 4-42. Signals defined by POSIX and MINIX 3. Signals indicated by (*) depend on hardware support. Signals marked (M) not defined by POSIX, but are defined by MINIX 3 for compatibility with older programs. Signals kernel are MINIX 3 specific signals generated by the kernel, and used to inform system processes about system events. Several obsolete names and synonyms are not listed here.

Signal Handling (4)

Signal	Description	Generated by
SIGALRM	Alarm clock, timeout	PM
SIGTERM	Software termination signal from kill	KILL system call
SIGCHLD	Child process terminated or stopped	PM
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
SIGKMESS	Kernel message	Kernel
SIGKSIG	Kernel signal pending	Kernel
SIGKSTOP	Kernel shutting down	Kernel

Figure 4-42. Signals defined by POSIX and MINIX 3. Signals indicated by (*) depend on hardware support. Signals marked (M) not defined by POSIX, but are defined by MINIX 3 for compatibility with older programs. Signals kernel are MINIX 3 specific signals generated by the kernel, and used to inform system processes about system events. Several obsolete names and synonyms are not listed here.

Signal Handling (5)

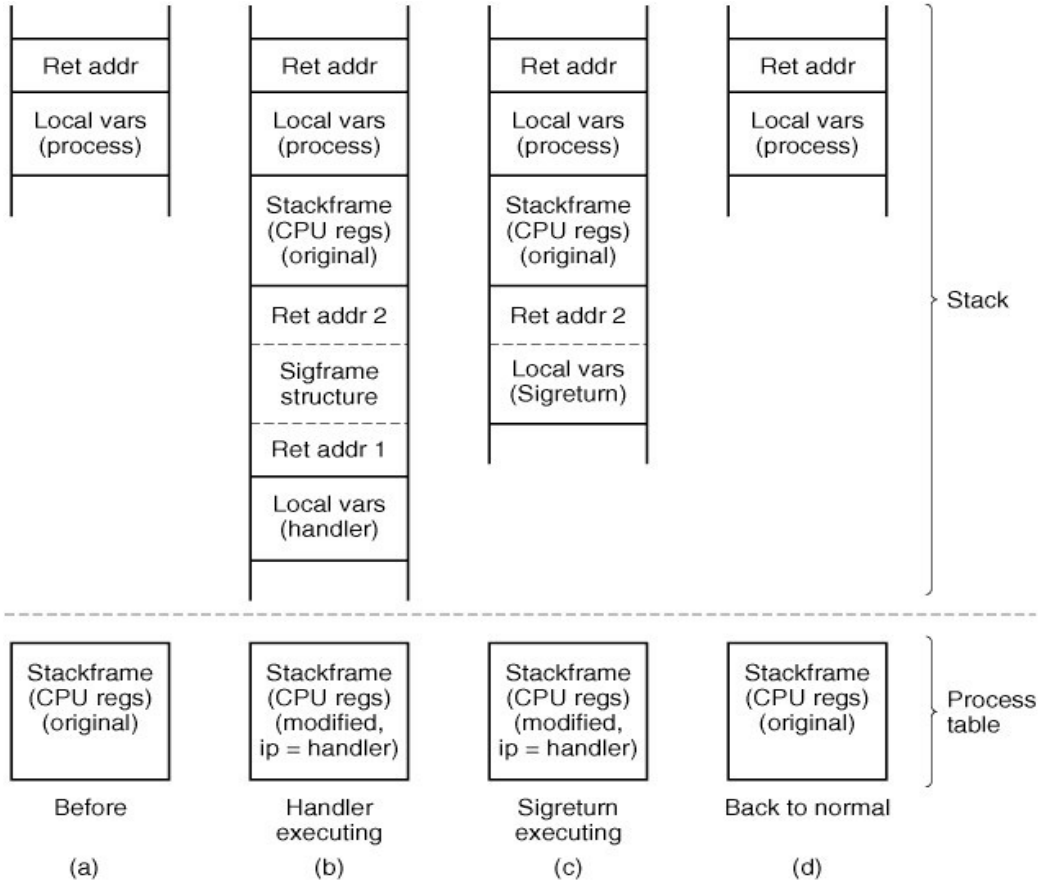

Figure 4-43. A process' stack (above) and its stackframe in the process table (below) corresponding to phases in handling a signal. (a) State as process is taken out of execution. (b) State as handler begins execution. (c) State while sigreturn is executing. (d) State after sigreturn completes execution.

Implementation of EXIT

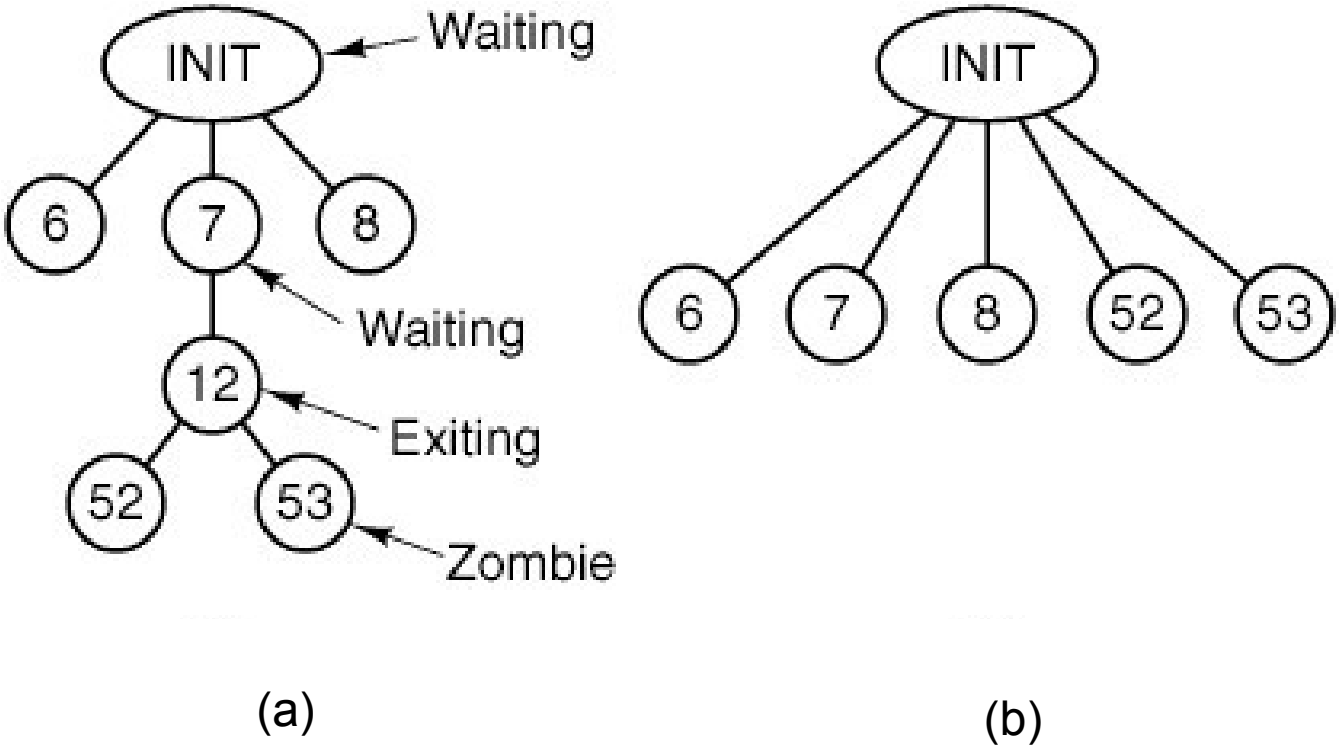

Figure 4-45. (a) The situation as process 12 is about to exit. (b) The situation after it has exited.

Implementation of EXEC

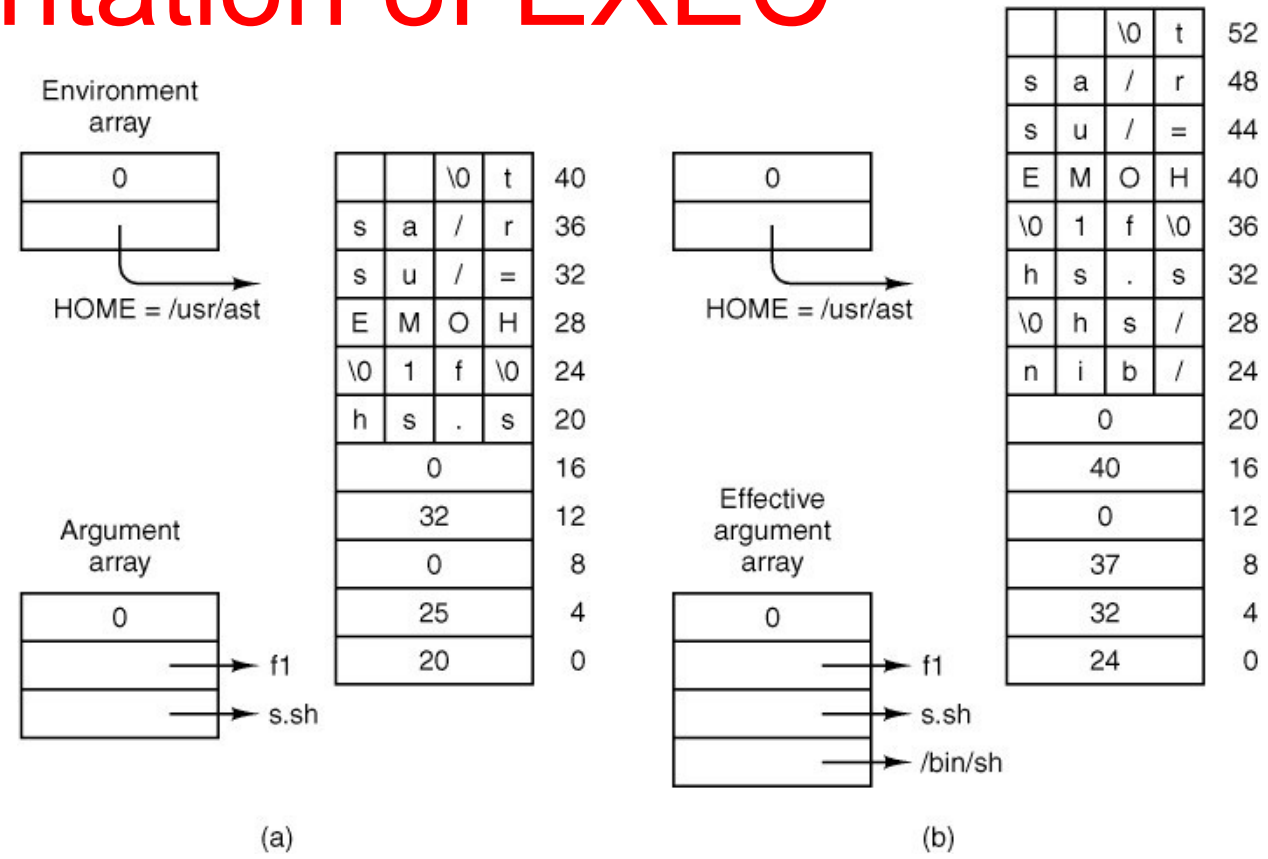

Figure 4-46. (a) Arrays passed to *execve* and the stack created when a script is executed. (b) After processing by *patch_stack*, the arrays and the stack look like this. The script name is passed to the program which interprets the script.

Signal Handling (1)

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 4-47. System calls relating to signals.

Signal Handling (2)

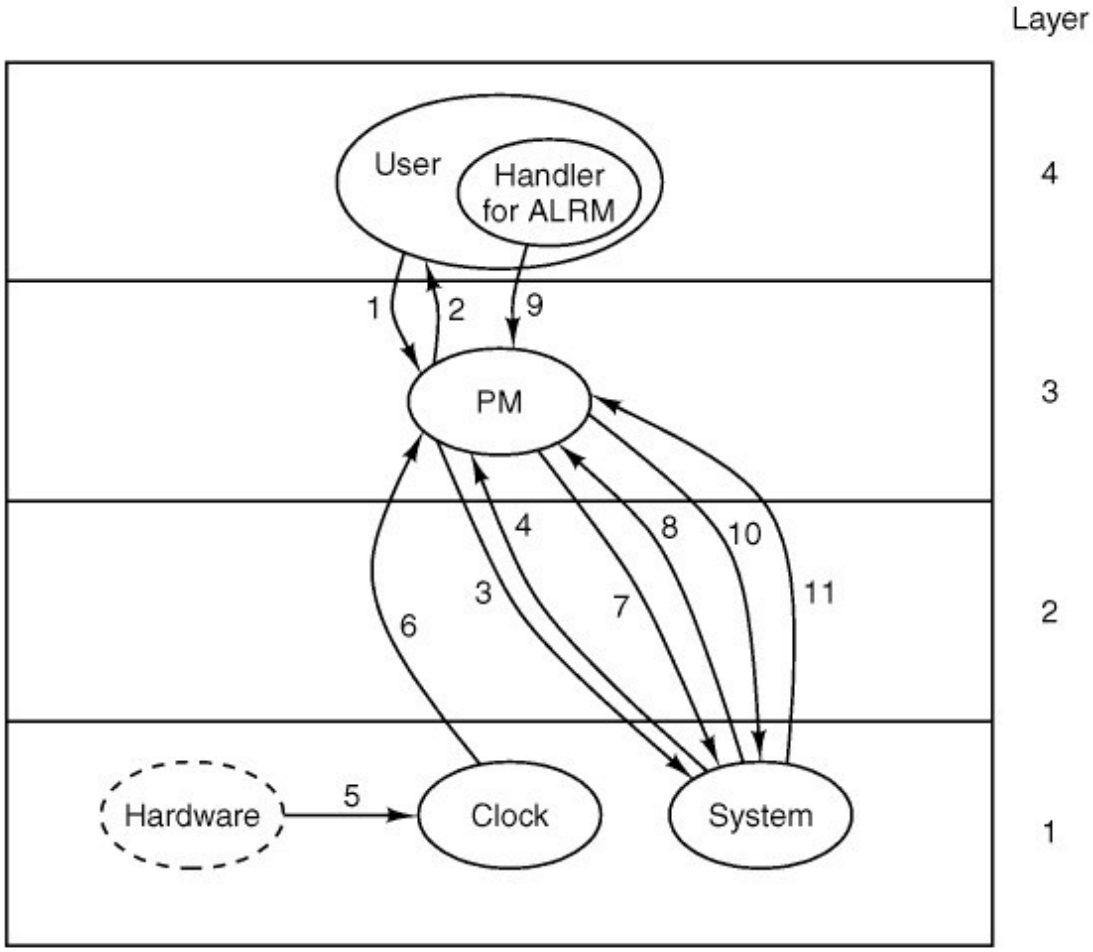

Figure 4-48. Messages for an alarm. The most important are: (1) User does alarm. (4) After the set time has elapsed, the signal arrives. (7) Handler terminates with call to sigreturn. See text for details.

Signal Handling (3)

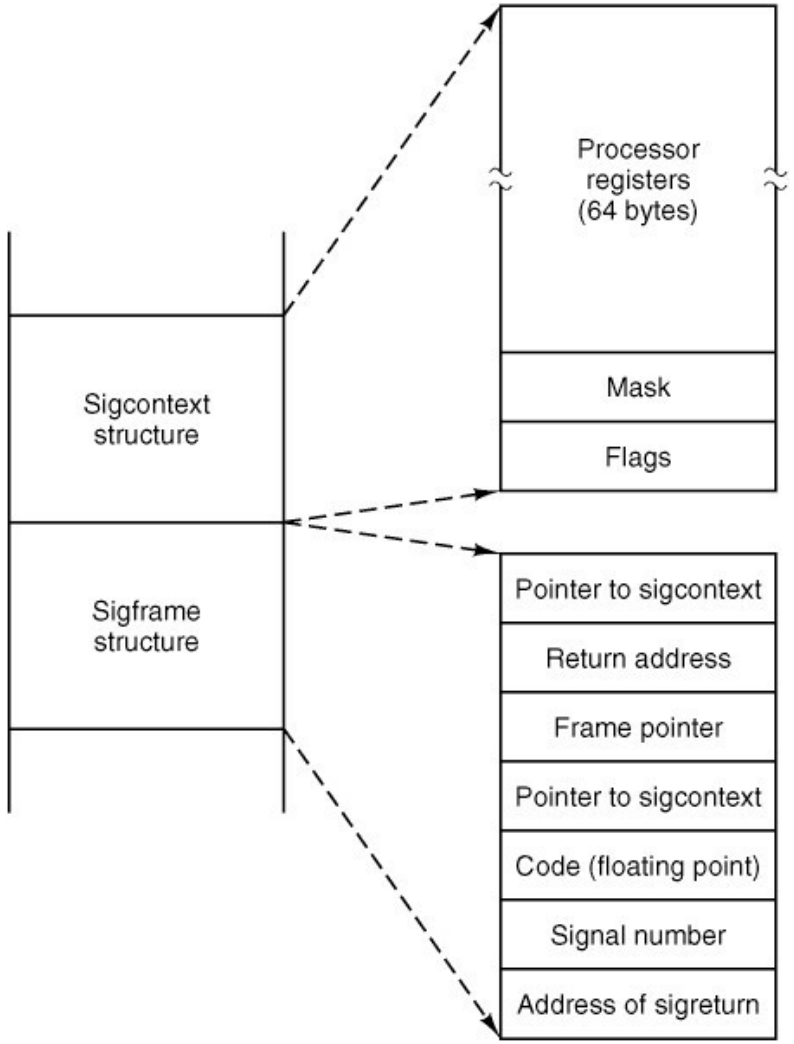

Figure 4-49. The sigcontext and sigframe structures pushed on the stack to prepare for a signal handler. The processor registers are a copy of the stackframe used during a context switch.

Other System Calls (1)

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

Figure 4-50. Three system calls involving time.

Other System Calls (2)

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

Other System Calls (3)

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

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

Other System Calls (4)

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 4-53. Debugging commands supported by *servers/pm/trace.c.*