Advanced Programming in the UNIX Environment

Process Limits and Identifiers

Terminal — 80×24

```
apue$
```

getrlimit(2) / setrlimit(2)

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);
```

Returns: 0 on success; -1 on error

Changing resource limits follows these rules:

- a process may change its *soft limit* to a value less than or equal to its *hard limit*
- any process can lower its *hard limit* greater than or equal to its *soft limit*
- only superuser can raise *hard limits*
- changes are <u>per process</u> only (which is why ulimit must be a shell built-in)

Terminal — 80×24

```
apue$
```

Process Identifiers

```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Process ID’s are guaranteed to be unique and identify a particular executing process with a non-negative integer.

Certain processes have fixed, special identifiers. They are:

- *swapper, sched, idle* or *system*, process ID 0 – responsible for scheduling
- *init*, process ID 1 – bootstraps a Unix system, owns orphaned processes
- *pagedaemon*, process ID 2 – responsible for the VM system (someUnix systems)

Terminal — 80×24

```
apue$
```

Process Limits and Identifiers

Certain aspects of a process's execution are restricted via resource limits.

A resource limit is specified as a *soft* limit and a *hard* limit; only the superuser may raise a hard limit.

Resource limits are enforced *per process.*

A process further has (at least) a process ID (PID) and a parent process ID (PPID). More on these process relationships in our next videos.

Advanced Programming in the UNIX Environment

Process Control

Terminal — 80×24

```c
	while (getinput(buf, sizeof(buf))) {
		buf[strlen(buf) - 1] = '\0';

		if((pid=fork()) == -1) {
			fprintf(stderr, "shell: can't fork: %s\n",
					strerror(errno));
			continue;
		} else if (pid == 0) {   /* child */
			execlp(buf, buf, (char *)0);
			fprintf(stderr, "shell: couldn't exec %s: %s\n", buf,
					strerror(errno));
			exit(EX_UNAVAILABLE);
		}

		/* parent waits */
		if ((pid=waitpid(pid, &status, 0)) < 0) {
			fprintf(stderr, "shell: waitpid error: %s\n",
					strerror(errno));
		}
	}

	exit(EX_OK);
```

58,9	1134	95%

fork(2)

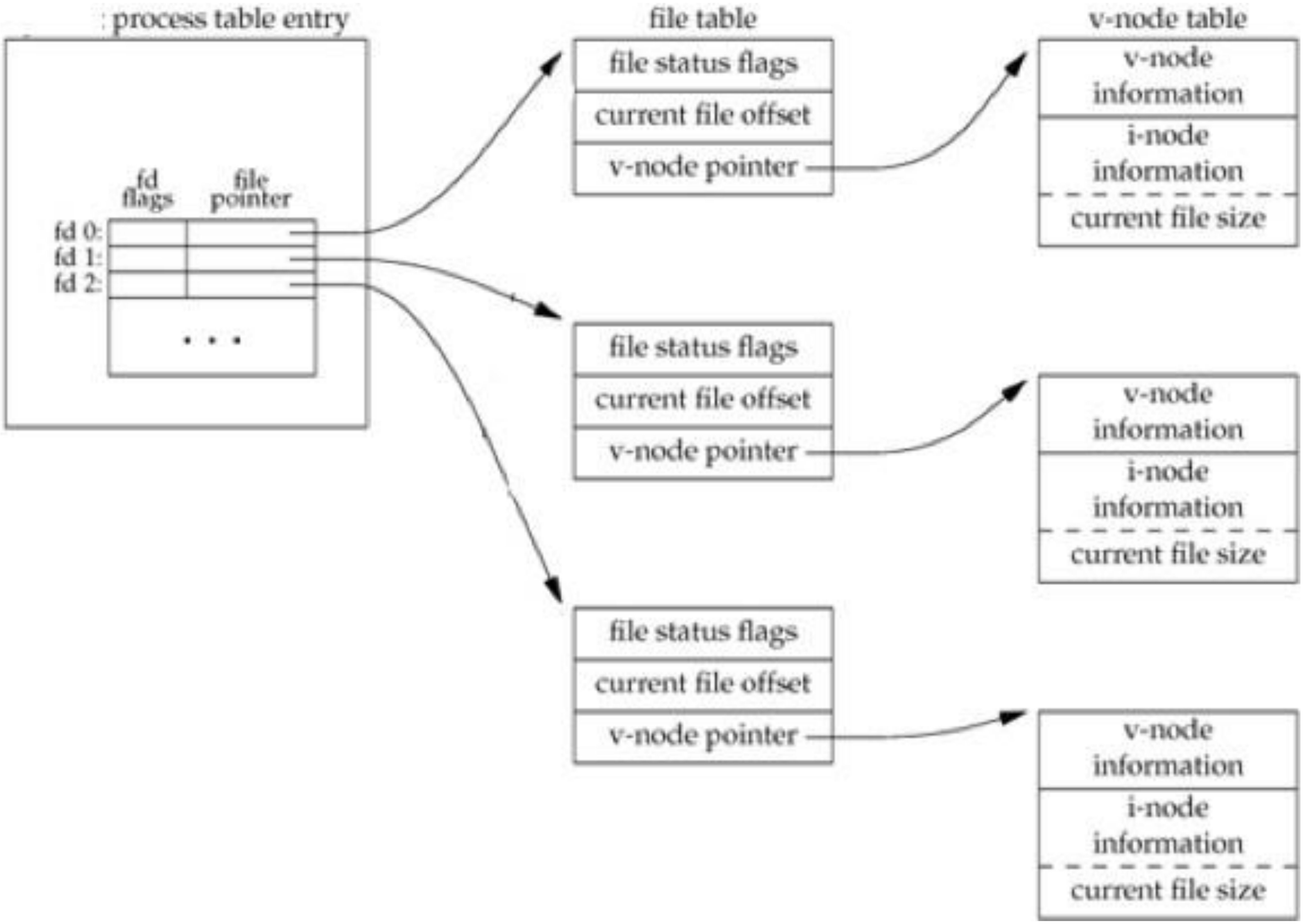
```
#include <unistd.h>

pid_t fork(void);
```

Returns: twice(!): 0 to the child, new pid to the parent; -1 on error

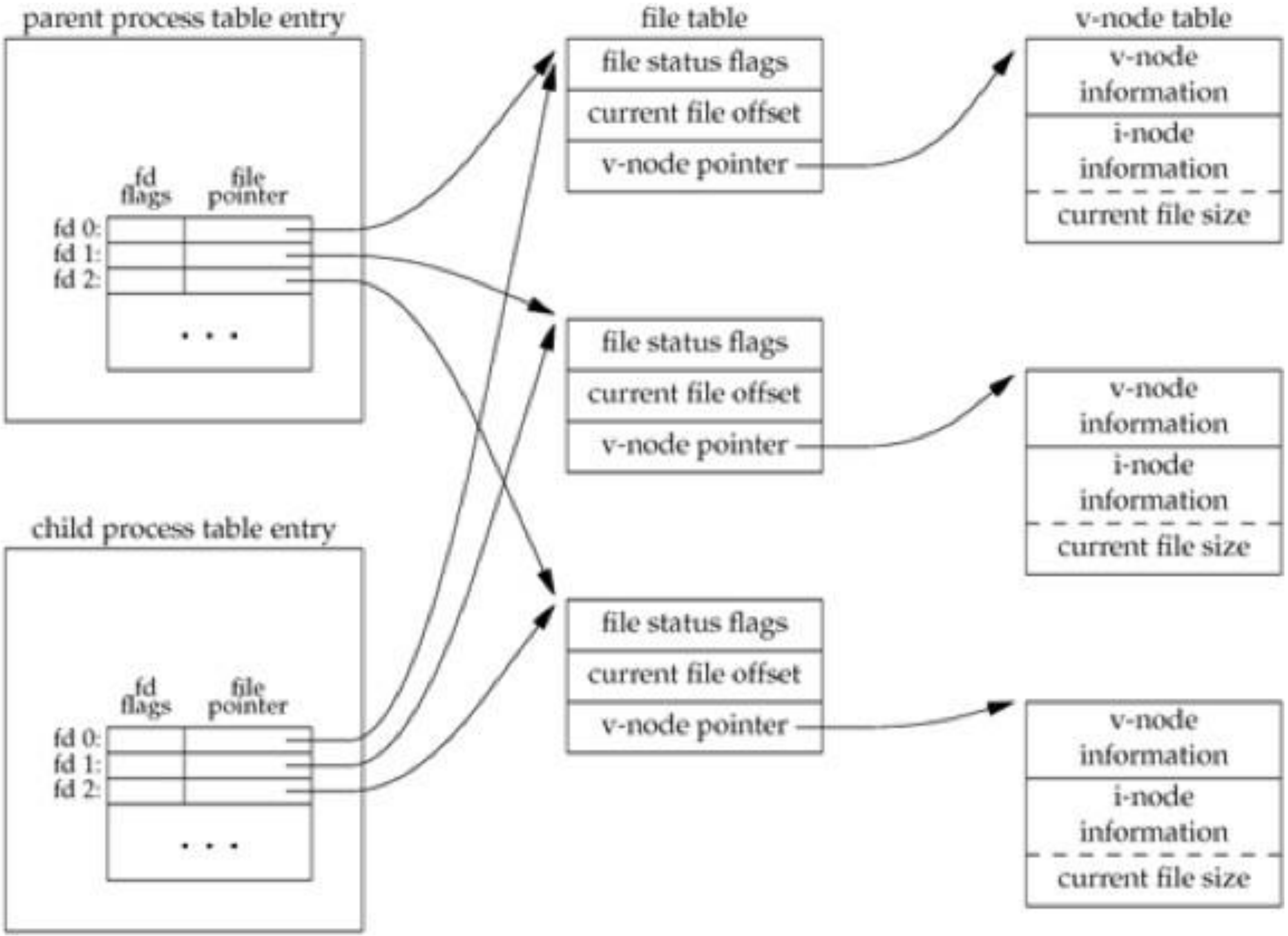
fork(2) causes creation of a new process. The new process (child) is an exact copy of the calling process (parent) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (*i.e.*, the processID of the parent process).
- The child process has its own copy of the parent's descriptors.
- The child process's resource utilizations are set to 0.

Note: no order of execution between child and parent is guaranteed!



process table entry
fd flags
file pointer
fd 0:
fd 1:
fd 2:
. . .
file table
file status flags
current file offset
v-node pointer
file status flags
current file offset
v-node pointer
file status flags
current file offset
v-node pointer
v-node table
v-node information
i-node information
current file size
v-node information
i-node information
current file size
v-node information
i-node information
current file size

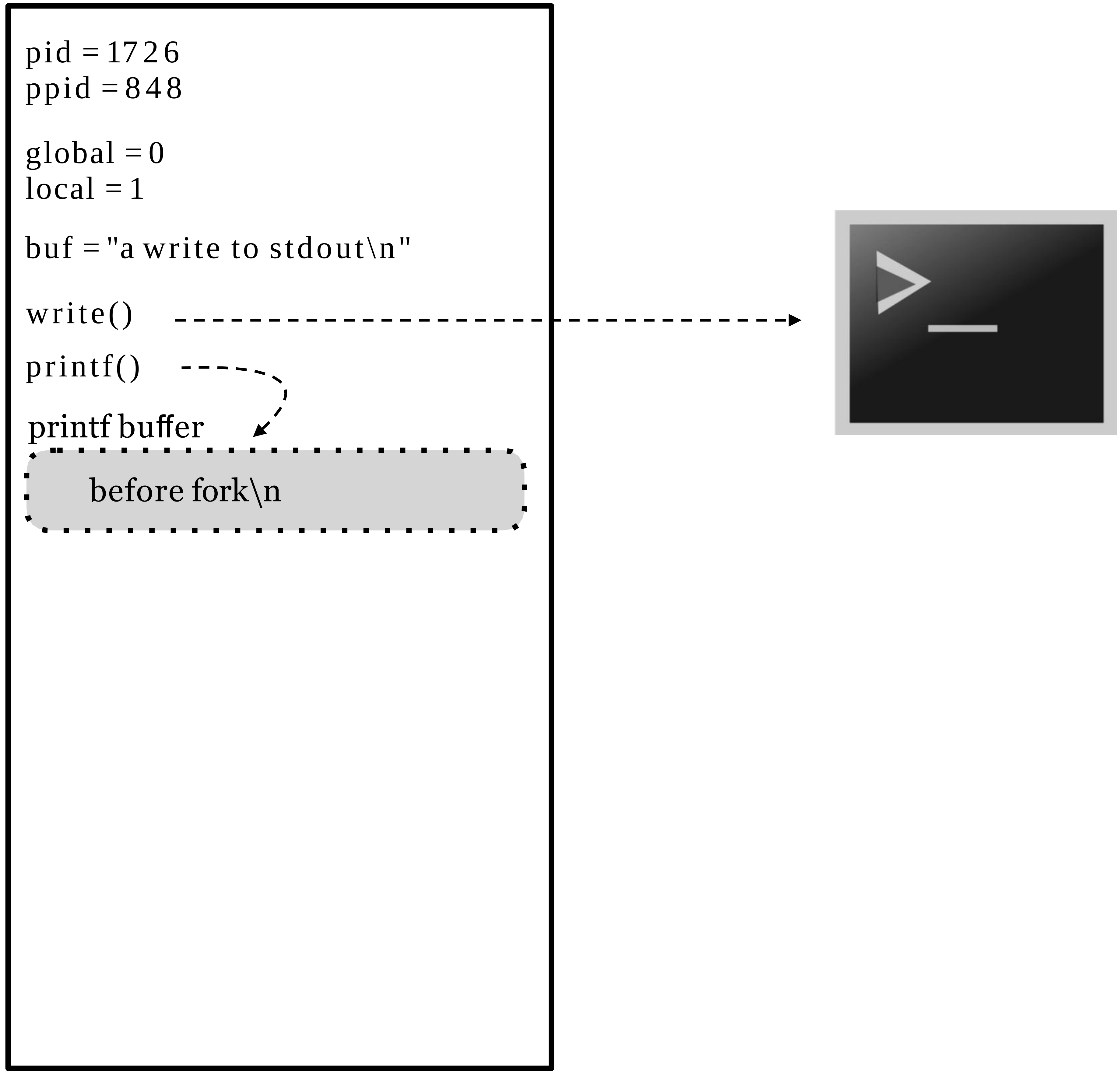

parent process table entry
fd flags
file pointer
fd 0:
fd 1:
fd 2:
. . .
child process table entry
fd flags
file pointer
fd 0:
fd 1:
fd 2:
. . .
file table
file status flags
current file offset
v-node pointer
file status flags
current file offset
v-node pointer
file status flags
current file offset
v-node pointer
v-node table
v-node information
i-node information
current file size
v-node information
i-node information
current file size
v-node information
i-node information
current file size

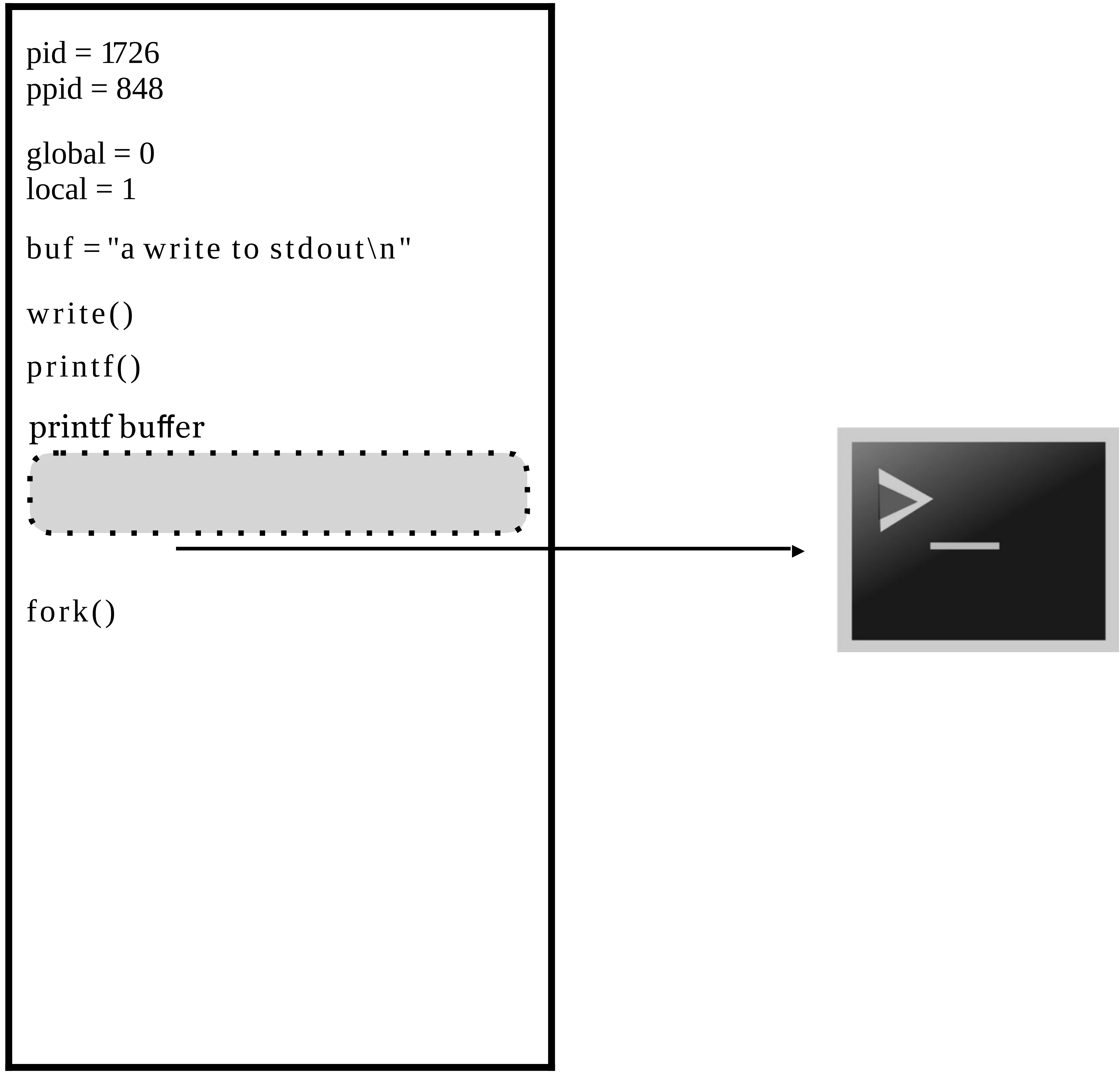
Terminal — 80×24

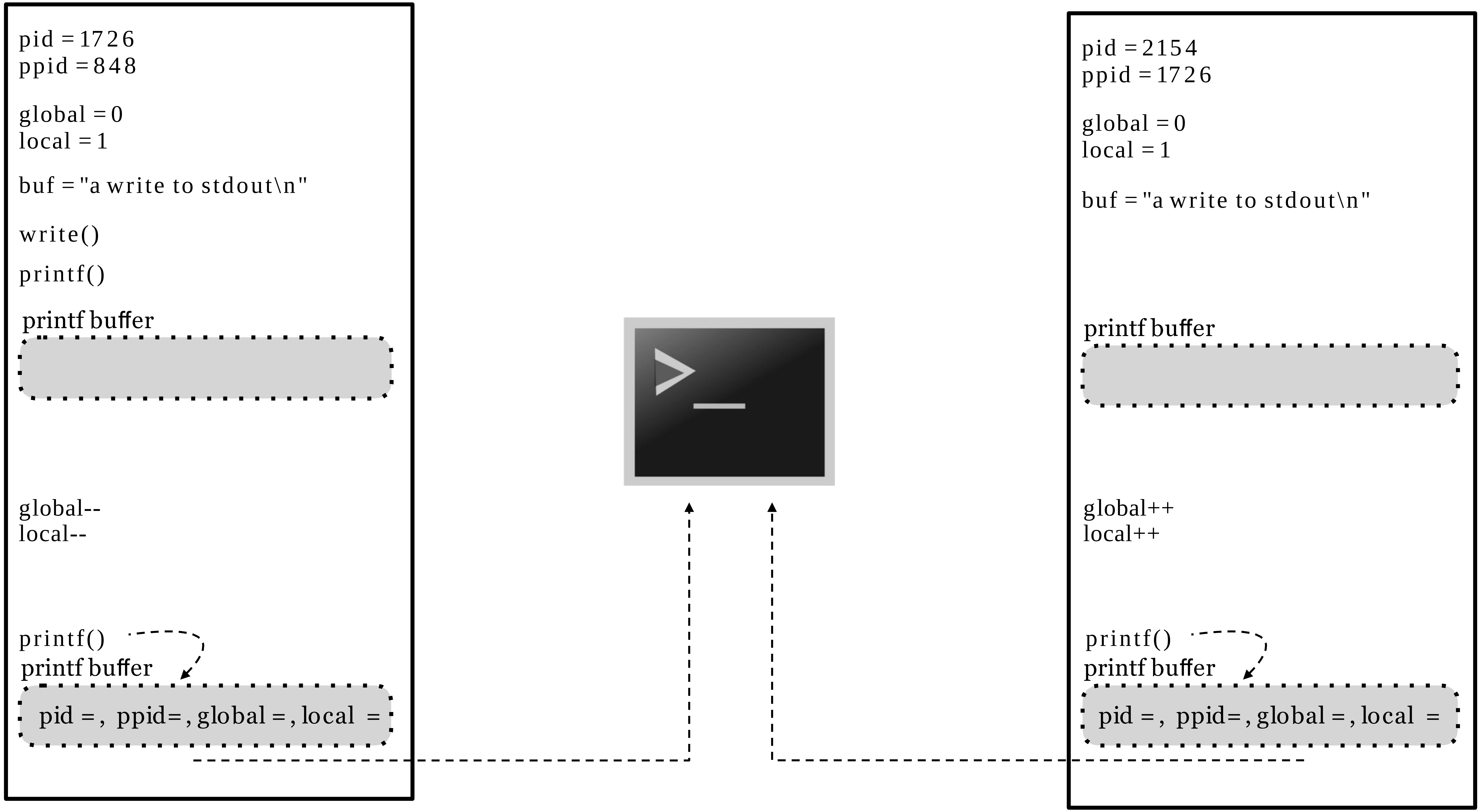
```
apue$ vim forkseek.c
apue$ cc -Wall -Werror -Wextra forkseek.c
apue$ ./a.out forkseek.c
Starting pid is: 361
361 offset is now: 0
child 999 done seeking
361 offset is now: 64
999 offset is now: 96
apue$
```

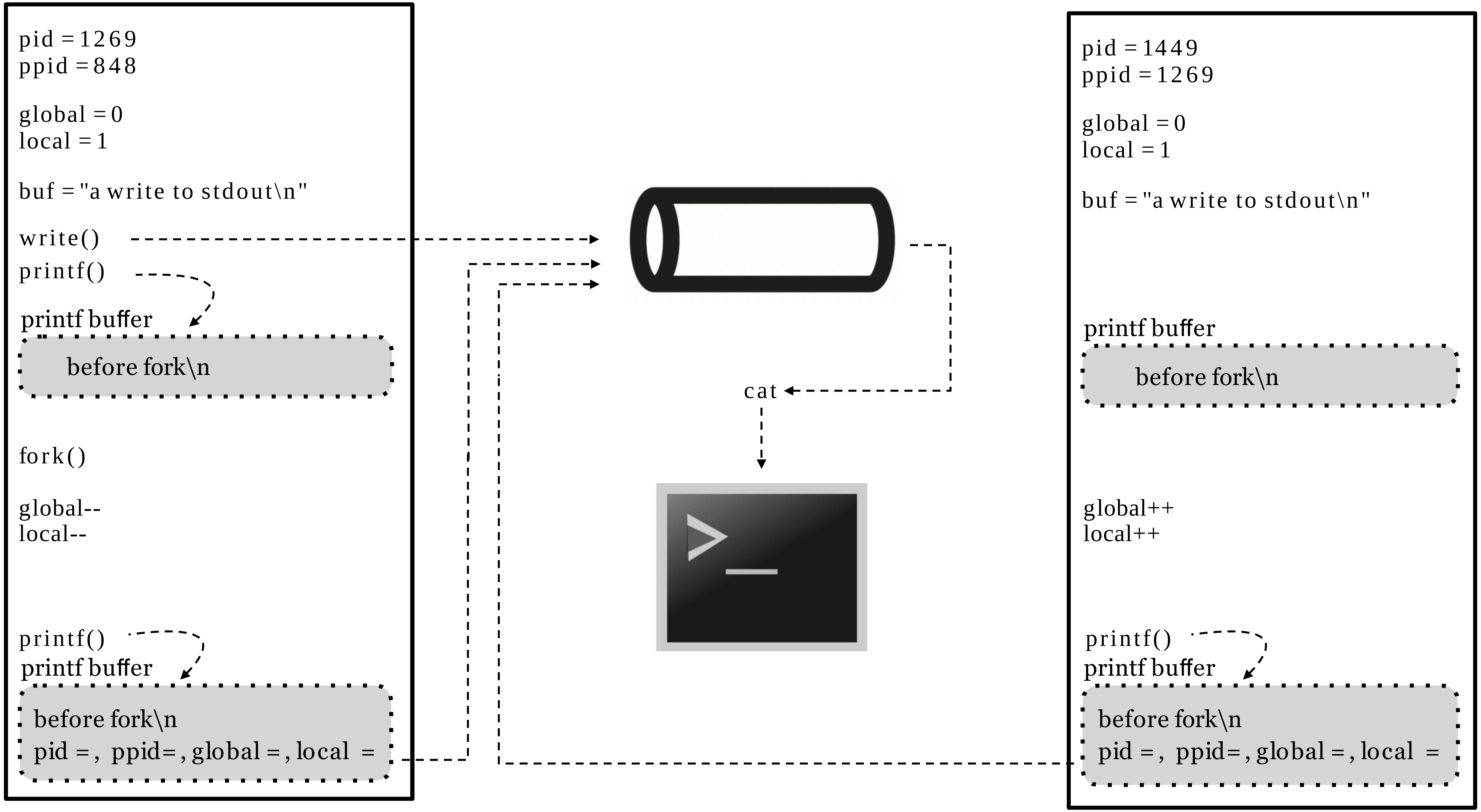
Terminal — 80×24

```
apue$ vi forkflush.c
apue$ cc -Wall -Werror -Wextra forkflush.c
apue$ ./a.out
a write to stdout
before fork
pid = 2154, ppid = 1726, global = 1, local = 2
pid = 1726, ppid = 848, global = -1, local = 0
apue$ echo $$
848
apue$ ./a.out | cat
a write to stdout
before fork
pid = 1449, ppid = 1269, global = 1, local = 2
before fork
pid = 1269, ppid = 848, global = -1, local = 0
apue$
```


pid = 1726
ppid = 848
global = 0
local = 1
buf = "a write to stdout\n"
write()
printf()
printf buffer
before fork\n


pid = 1726
ppid = 848
global = 0
local = 1
buf = "a write to stdout\n"
write()
printf()
printf buffer
fork()


pid = 1726
ppid = 848
global = 0
local = 1
buf = "a write to stdout\n"
write()
printf()
printf buffer
global--
local--
printf()
printf buffer
pid = , ppid= , global = , local =
pid = 2154
ppid = 1726
global = 0
local = 1
buf = "a write to stdout\n"
printf buffer
global++
local++
printf()
printf buffer
pid = , ppid= , global = , local =



pid = 1269
ppid = 848
global = 0
local = 1
buf = "a write to stdout\n"
write()
printf()
printf buffer
before fork\n
fork()
global--
local--
printf()
printf buffer
before fork\n
pid = , ppid= , global = , local =
cat
pid = 1449
ppid = 1269
global = 0
local = 1
buf = "a write to stdout\n"
printf buffer
before fork\n
global++
local++
printf()
printf buffer
before fork\n
pid = , ppid= , global = , local =

exec(3)

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *path, const char *arg, ...);
int execlpe(const char *path, const char *arg, ..., char *const envp[]);
int execle(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *path, char *const argv[]);
int execvpe(const char *path, char *const argv[], char *const envp[]);
```

Returns: doesn't; -1 on error

The `exec()` family of functions are used to completely replace a running process with a new process image. They all are front-ends for the `execve(2)` system call.

The `exec(3)` functions

- if it has a *v* in its name, argv's are a vector: `const * char argv[]`
- if it has an *l* in its name, argv's are a list: `const char *arg0, .../* (char *) 0 */`
- if it has an *e* in its name, it takes a `char * const envp[]` array of environment variables
- if it has a `p` in its name, it uses the `PATH` environment variable to search for the file

- open file descriptors are inherited, unless the close-on-exec file flag was set
- ignored signals in the calling process are ignored after exec, but caught signals are reset to default
- real UID/GID is inherited; effective UID/GID is inherited unless the executable was setuid/setgid

wait(2) and waitpid(2)

```
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t wpid, int *status, int options);


# include <sys/resource.h>
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);
```

Returns: child PID on success; -1 on error

`wait()` suspends execution of the calling process until status information is available for a terminated child process.

`waitpid()` / `wait4()` allow waiting for a specific process or process group; `wait3()` / `wait4()` allow inspection of resource usage.

wait(2) and waitpid(2)

Once we get a termination status back in status, we'd like to be able to determine how a child died. We do this with the following macros:

- WIFEXITED(status) – true if the child terminated normally; use WEXITSTATUS(status) to get the exit status
- WIFSIGNALED(status) – true if child terminated abnormally (by receiving a signal it didn't catch); use
  - WTERMSIG(status) to retrieve the signal number
  - WCOREDUMP(status) to see if the child left a core image
- WIFSTOPPED(status) – true if the child is currently stopped; use WSTOPSIG(status) to determine the signal that caused this

Additionally, wait(2) will block until a child terminates; pass WNOHANG to waitpid(2) / wait(4) to return immediately.

What happens if we don't `wait(2)`?

Terminal — 80×24

```
4175 pts/1 Z+   0:00.00 (a.out)
====
1102 pts/1 Z+   0:00.00 (a.out)
2668 pts/1 Z+   0:00.00 (a.out)
2974 pts/1 Z+   0:00.00 (a.out)
2996 pts/1 S+   0:00.00 ./a.out
4175 pts/1 Z+   0:00.00 (a.out)
====
1102 pts/1 Z+   0:00.00 (a.out)
2668 pts/1 Z+   0:00.00 (a.out)
2974 pts/1 Z+   0:00.00 (a.out)
2996 pts/1 S+   0:00.00 ./a.out
3723 pts/1 Z+   0:00.00 (a.out)
4175 pts/1 Z+   0:00.00 (a.out)
I'm going to sleep - try to kill my zombie children, if you like.
[1]   Terminated              ./a.out
apue$ ps
 PID TTY   STAT     TIME COMMAND
 701 pts/0 Is   0:00.09 -sh
2262 pts/0 S+   0:00.01 screen -r -d
2134 pts/1 O+   0:00.00 ps
4217 pts/1 Ss   0:00.01 /bin/sh
2388 pts/2 Is+  0:00.01 /bin/sh
2361 pts/3 Ss+  0:00.01 /bin/sh
```

Process Control

All processes not explicitly instantiated by the kernel were created via fork(2).

fork(2) creates a copy of the current process, including file descriptors and output buffers.

To replace the current process with a new process image, use the exec(3) family of function.

After creating a new process via fork(2), the parent process can wait(2) for the child process to reap its exit status and resource utilization; failure to do so will create a zombie process until the parent is terminated, at which point init will reap it.