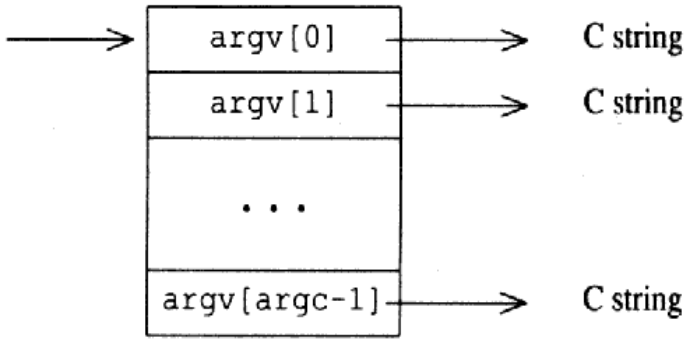
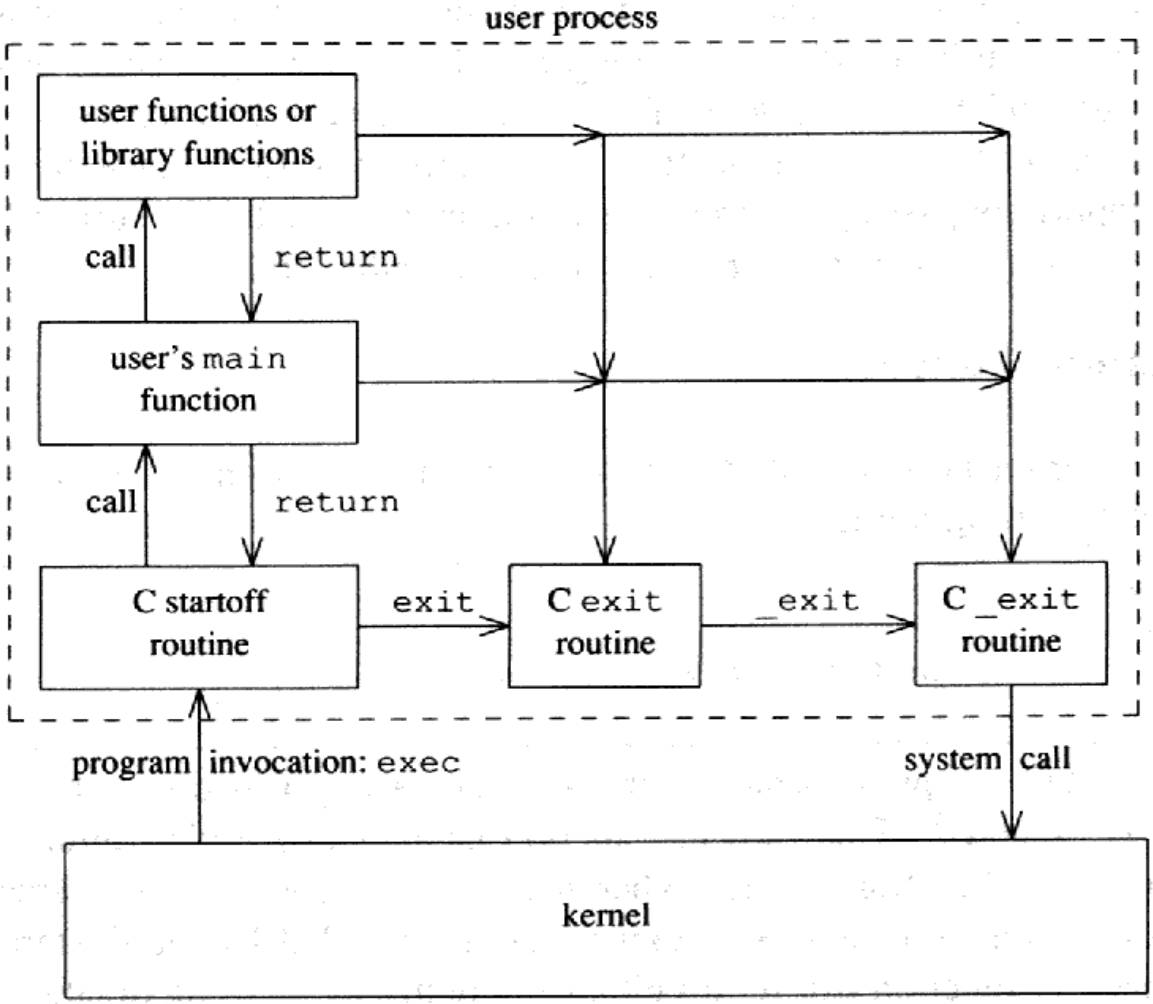
Outline

- Basics
  - Programs
  - Processes
  - File Model
  - Signal Model
  - Process Control
- Interprocess Communication(IPC)
  - File locking
  - Pipes
  - FIFOs
  - Shared Memory

Start-up Function






```
main(argc, argv, envp)
int argc;
char *argv[];
char *envp[];
{…
}
```

Environment List

- Example:

```
main(argc, argv, envp)
int     argc;
char    *argv[];
char    *envp[];
{
        int     i;

        for (i = 0; envp[i] != (char *) 0; i++)
                printf("%s\n", envp[i]);
        exit(0);
}
```

- Output:

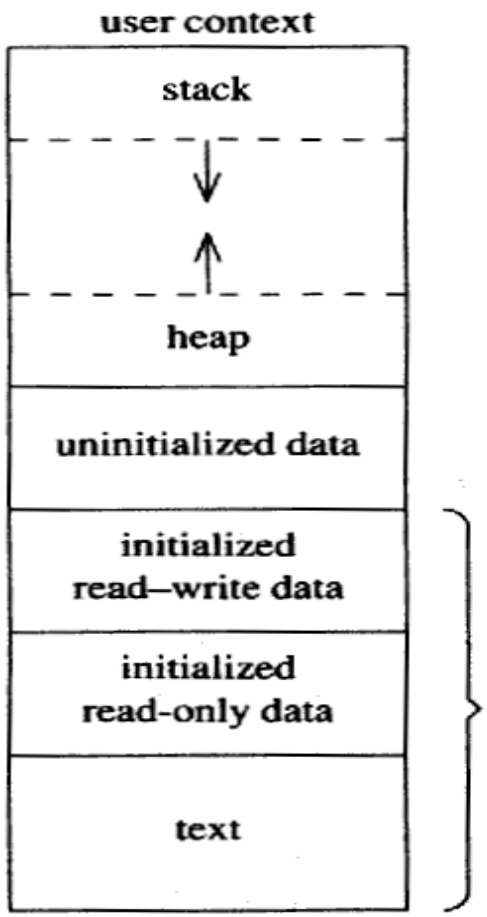
```
HOME=/usr1/stevens
SHELL=/bin/csh
TERM=att630
USER=stevens
PATH=/usr1/stevens/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin
EXINIT=set optimize redraw shell=/bin/csh
```

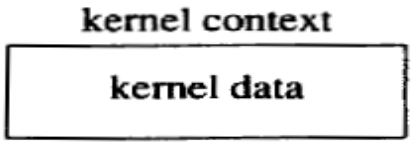
Environment List

- Use the external variable `envp` and `getenv()`:

```
main(argc, argv)
int     argc;
char    *argv[];
{
        int             i;
        extern char     **environ;

        for (i = 0; environ[i] != (char *) 0; i++)
                printf("%s\n", environ[i]);
        exit(0);
}
```

Process






```
int     debug = 1;
char    *progname;

main(argc, argv)
int     argc;
char    *argv[];
{
        int     i;
        char    *ptr, *malloc();

        progname = argv[0];
        printf("argc = %d\n", argc);
        for (i = 1; i < argc; i++) {
                ptr = malloc(strlen(argv[i]) + 1);
                strcpy(ptr, argv[i]);
                if (debug)
                        printf("%s\n", ptr);
        }
}
```

Kernel and Process

Contains information that the kernel needs to

- keep track of the process
- stop and restart the process

Process table contains a portion of kernel context.

Process ID(PID):

- 0: a special kernel process for “scheduler”.
- 1: a special process,called “init”.
- 2: a kernel process, called “pagedaemon”.
- 3 ~ (2^{32}-1) (~65000): no special meaning.

Parent Process ID:

Users and Groups

- Real User ID: (see `/etc/passwd`)
  - real user means the user executing the file.
  - Superuser: root, its user ID is 0.

```
unsigned short getuid();
```

- Real Group ID: (see `/etc/group`)

```
unsigned short getgid();
```

- Effective User ID:
  - effective user = real user
  - but it may also means the user owning the file.

```
unsigned short geteuid();
```

- Effective Group ID:

```
unsigned short getegid();
```

- Why Effective? ➔ Security Consideration
  - Consider a user to execute “cat > /etc/passwd” and “passwd”.

Password File

Password File

Each line in the `/etc/passwd` file has the following format:

login-name : *encrypted-password* : *user-ID* : *group-ID* : *miscellany* : *login-directory* : *shell*

There are seven fields, separated from each other by colons. The *login-name* is the name you enter in response to the `login:` prompt when logging on to the system. This field is sometimes called the *user name*. The *encrypted-password* field can be empty, in which case you are not prompted for a password. Since the passwords in this file are encrypted, this file is always readable by anyone. The *user-ID* and *group-ID* fields are the numeric values described above. The *login-directory* specifies your initial current working directory. The *shell* field specifies the pathname of a program that is invoked when you login. All these fields are described in more detail later in this chapter. Two typical entries are

```
root:x7f1mVqMxGl4g:0:10:The Superuser:/:/bin/ksh
stevens:u0ud5eOq2MpaZ:224:5:Richard Stevens:/usr1/stevens:/bin/ksh
```

The standard C library provides two functions to search the `/etc/passwd` file, looking for a matching user ID or login name.

```
#include <pwd.h>

struct passwd *getpwuid(int uid);

struct passwd *getpwnam(char *name);
```

File Types

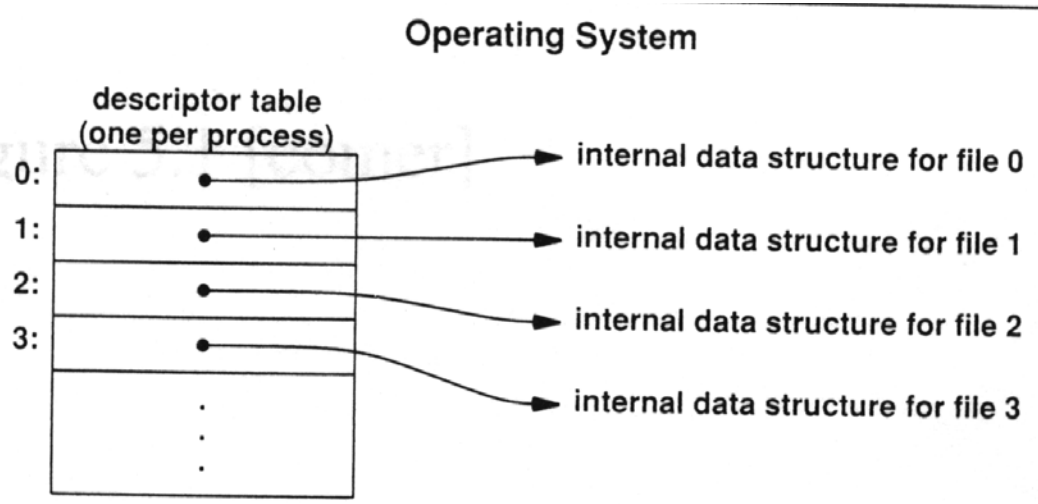
- Regular file
- Directory
- Special file: Character(TTY) / Block(Disk)
- Fifo
- Symbolic link
- Socket

- File access mode word (e.g., rwx etc, Figure 2.3)
- File mode creation mask (e.g., umask, file is 666 and directory is 777)

File Model in Unix

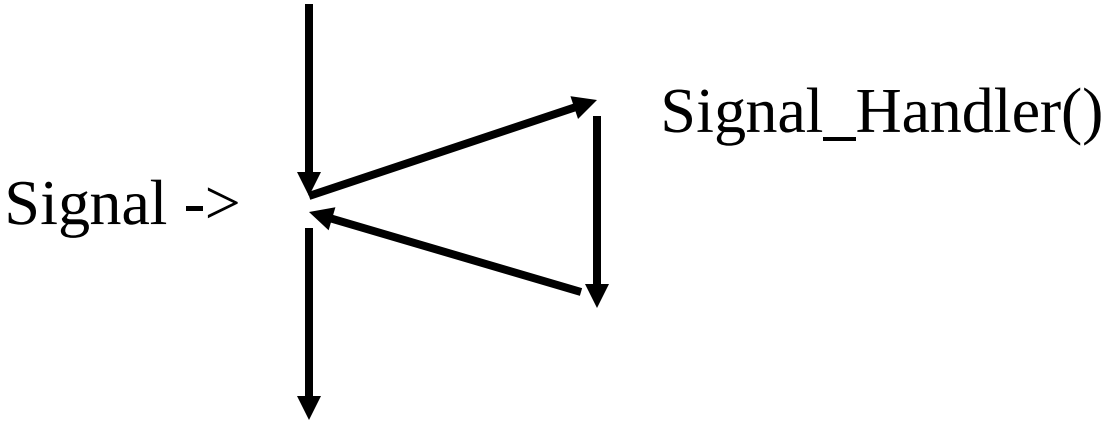
- File Descriptor Table:

0	stdin
1	stdout
2	stderr
3	file ...
4	file ...
..	...



- `fd = open("filename", O_RDWR/*flag*/, 0 /*mode*/).`
- `close(fd):`
- `read/write(fd,buffer,128/*size*/).`
- `fcntl`: control opened file's properties, e.g., nonblocking I/O, `ASYNC` to enable `SIGIO` signal.
- `ioctl`: control fd's behavior.(more for devices)
- `lseek/dup/dup2:`

Signal Model



Important:

- Signal event can happen at any time.

  ==>concurrent programming must be careful.

Basic Signal Functions

- Kill (int pid, int sig)
- signal (int sig, void (*func)(int))

Signal Tables

Name	Note	Description	Default action
SIGALRM		Alarm clock	Terminate
SIGBUS		Bus error	Terminate with core image
SIGCLD		Death of a child process	Discarded
SIGCONT	4.3BSD	Continue after SIGSTP	Discarded
SIGEMT		EMT instruction	Terminate with core image
SIGFPE		FPE instruction	Terminate with core image
SIGHUP		Hangup	Terminate
SIGILL		Illegal instruction	Terminate with core image
SIGINT		Interrupt character	Terminate
SIGIO	4.3BSD	I/O is possible on a file descriptor	Discarded
SIGIOT		IOT instruction	Terminate with core image
SIGKILL		Kill	Terminate
SIGPIPE		Write on a pipe with no one to read it	Terminate
SIGPOLL	System V	Selectable event on a streams device	Terminate
SIGPROF	4.3BSD	Profiling timer alarm	Terminate
SIGPWR	System V	Power fail	Terminate
SIGQUIT		Quit character	Terminate with core image
SIGSEGV		Segmentation violation	Terminate with core image
SIGSTOP	4.3BSD	Stop	Stop (suspend) process
SIGSYS		Bad argument to system call	Terminate with core image
SIGTERM		Software termination signal	Terminate
SIGTRAP		Trace trap	Terminate with core image
SIGTSTP	4.3BSD	Stop signal generated from keyboard	Stop (suspend) process
SIGTTIN	4.3BSD	Background read from control terminal	Stop (suspend) process
SIGTTOU	4.3BSD	Background write to control terminal	Stop (suspend) process
SIGURG	4.3BSD	Urgent condition present on socket	Discarded
SIGUSR1		User defined signal 1	Terminate
SIGUSR2		User defined signal 2	Terminate
SIGVTALRM	4.3BSD	Virtual time alarm	Terminate
SIGWINCH	4.3BSD	Window size change	Discarded
SIGXCPU	4.3BSD	CPU time limit exceeded	Terminate
SIGXFSZ	4.3BSD	File size limit exceeded	Terminate

Parameters

- PID:
  - pid = 0
    ==> sender's process group.
  - pid = -1 & sender = root
    ==> all processes except 0 and 1.
  - pid = -1 & sender not = root
    ==>all effective user ID is sender.
  - pid < -1 ==>process group -pid.

- Functions:
  - SIG_DFL: Default routine.
  - SIG_IGN: Ignore the signal.
  - user_defined routine.

Reliable Signals

- Signal handler remains installed after a signal occurs.
  - Older version will reset the handler to the default one before calling the handler. This may make message lost.
- Block the second signal handler while handling it.
- Avoid selected signals from occuring when desired, but do not discard it (blocking).
  - Older version has to use Signal to discard the signal. BSD provides some function.
  - E.g., Read some extra pages.

Sigmask & Sigblock

- Critical region:

```
int     oldmask;

oldmask = sigblock( sigmask(SIGQUIT) | sigmask(SIGINT) );

/* critical region */

sigsetmask(oldmask);     /* reset the mask to what it was */
```

Pause

- Wait for a signal to occur, but not safe…

```
int     flag = 0; /* global variable set when SIGINT occurs */

...
for ( ; ; ) {
        while (flag == 0)
                pause();    /* wait for a signal to occur */
        /* the signal has occurred, process it */
        ...
}
```

```
void intr_handler() {
   flag = 1;
}
...
signal(SIGINT, intr_handler);
...
```

Sigpause

- A Safe way. (why?)

```
int     flag = 0; /* global variable set when SIGINT occurs */

for ( ; ; ) {
        sigblock(sigmask(SIGINT));
        while (flag == 0)
                sigpause(0); /* wait for a signal to occur */
        /* the signal has occurred, process it */
        ...
}
```

- Notes:
  - sigblock: (on linux & freebsd)
    This interface is made obsolete by sigprocmask(2).
  - pause & sigpause: (on linux & freebsd)
    This interface is made obsolete by sigsuspend(2).

Problems of Concurrency

- Example I:

	Taipei	Hsinchu
Savings (S):	$30000	$30000
Program:	X=read(S)	X=read(S)
	X=X-10000	X=X-15000
	write X to S	write X to S

- Example Π

	Taipei	Hsinchu
Savings (S):	$30000	$30000
Program:	X=read(S)	X=read(S)
	X=X+10000	X=X+15000
	write X to S	write X to S

Problems of Concurrency

- Example I:

	Taipei	Hsinchu
Savings (S):	$30000	$30000
Program:	X=read(S)	X=read(S)
	X=S(30000)	X=S(20000)
	X=X-10000	X=X-15000
	X=20000	X=5000
	write X to S	write X to S
	S=20000	S=5000

Idea!

- Example Π

	Taipei	Hsinchu
Savings (S):	$30000	$30000
Program:	X=read(S)	X=read(S)
	X=X+10000	X=X+15000
	write X to S	write X to S

Problems of Concurrency

- Example I:

	Taipei	Hsinchu
Savings (S):	$30000	$30000
Program:	X=read(S)	X=read(S)
	X=S(30000)	X=S(30000)
	X=X-10000	X=X-15000
	X=20000	X=15000
	write X to S	write X to S
	S=20000	S=15000

But!

- Example Π

	Taipei	Hsinchu
Savings (S):	$30000	$30000
Program:	X=read(S)	X=read(S)
	X=X+10000	X=X+15000
	write X to S	write X to S

Problems of Concurrency

- Example I:

	Taipei	Hsinchu
Savings (S):	$30000	$30000
Program:	X=read(S)	X=read(S)
	X=X-10000	X=X-15000
	write X to S	write X to S

- Example Π

	Taipei	Hsinchu
Savings (S):	$30000	$30000
Program:	X=read(S)	X=read(S)
	X=S(30000)	X=S(30000)
	X=X+10000	X=X+15000
	X=40000	X=45000
	write X to S	write X to S
	S=40000	S=45000

But!

Process Control: Fork

- Fork: only copy data. test is shared.

```
main()
{
        int     childpid;

        if ( (childpid = fork()) == -1) {
                perror("can't fork");
                exit(1);
        } else if (childpid == 0) {
                /* child process */
                printf("child: child pid = %d, parent pid = %d\n",
                                        getpid(), getppid());
                exit(0);
        } else {
                /* parent process */
                printf("parent: child pid = %d, parent pid = %d\n",
                                        childpid, getpid());
                exit(0);
        }
}
```

Error Code

- In Unix, the system calls usually return -1, for errors.
  - To find out which error it is.
    - Access the global variable “errno”.
    - The meaning of “errno” can be found in Unix manual.

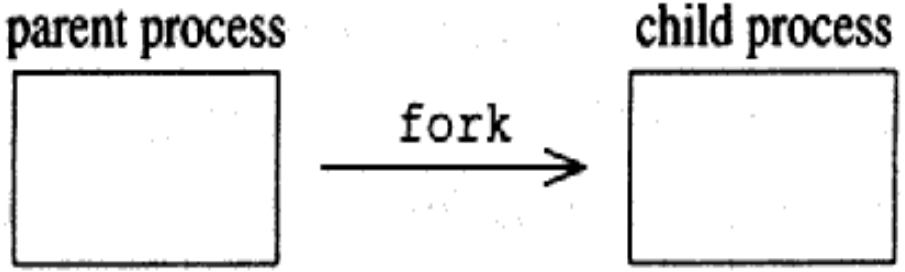
Differences Between Child and Parent Processes

Child process

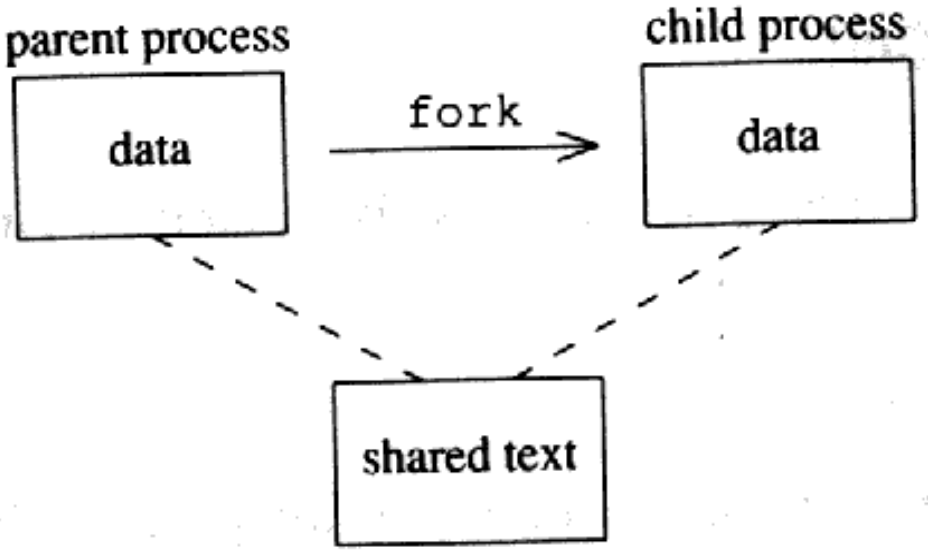
- has a new pid.
- has a different parent pid.
- has its own copies of the the parent’s file description table.
- resets the alarm clock, if it was set in the parent.

Consideration of Implementation

- Logically ➔

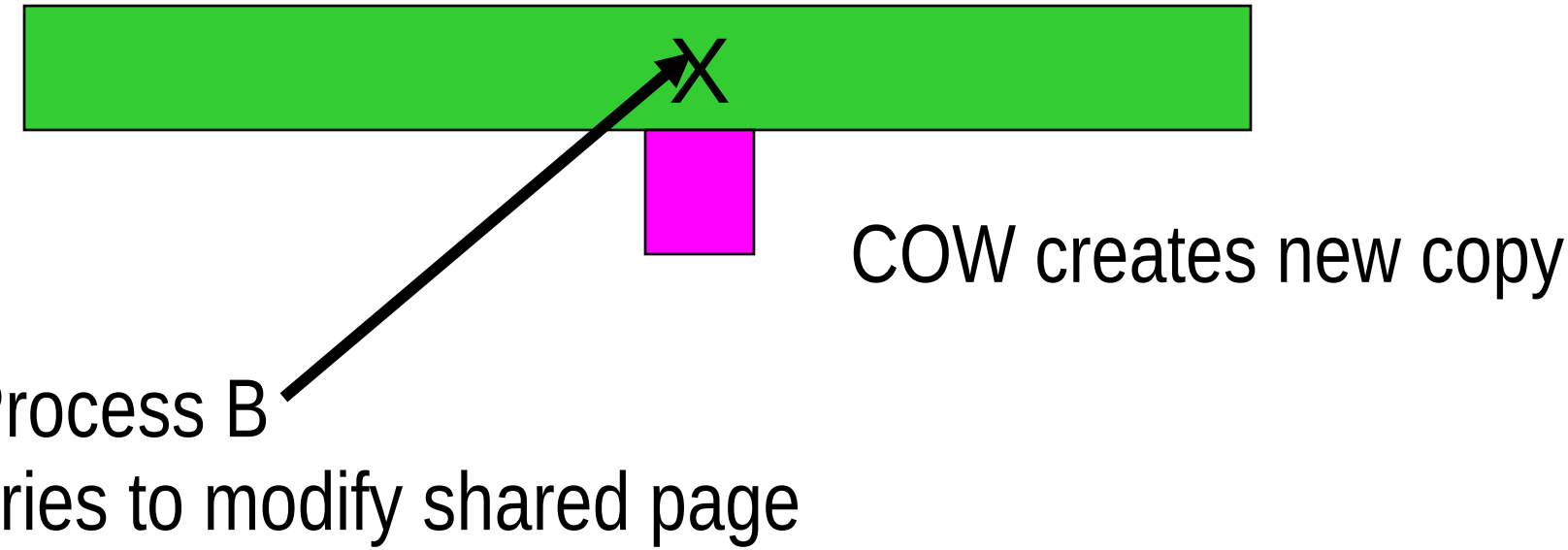


- Physically ➔
  - Shared memory
  - Copy-on-write



Copy-on-write (COW)

Process A and B share a range of pages



Process Control: Exec

- Exec: replace the program to the file.

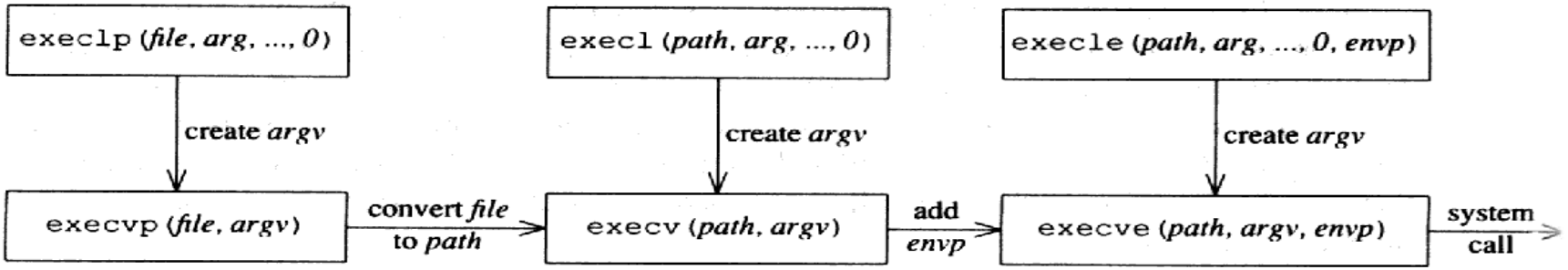
  - process ID
  - parent process ID
  - process group ID
  - terminal group ID
  - time left until an alarm clock signal
  - root directory
  - current working directory
  - file mode creation mask
  - real user ID
  - real group ID
  - file locks

  Two attributes that can change when a new program is `execed` are

  - effective user ID,
  - effective group ID.

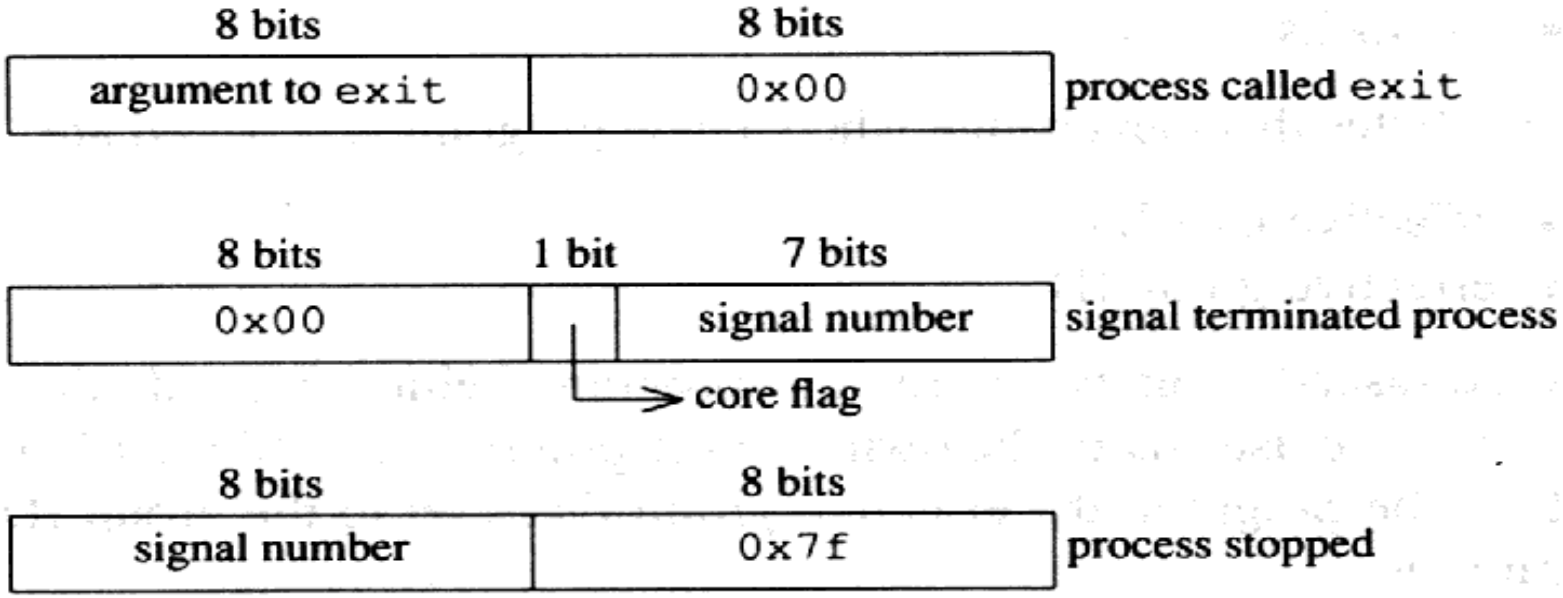
```
int execlp(char *filename, char *arg0, char *arg1, ..., char *argn,
           (char *) 0);

int execl(char *pathname, char *arg0, char *arg1, ..., char *argn,
          (char *) 0);

int execle(char *pathname, char *arg0, char *arg1, ..., char *argn,
           (char *) 0, char **envp);

int execvp(char *filename, char **argv);

int execv(char *pathname, char **argv);

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

The `exec` functions return to the caller only if an error occurs. Otherwise control passes to the start of the new program. The relationship between these six functions is shown in Figure 2.5.



Process Control: Wait

- `wait`: wait for one of its children finishing (exit).

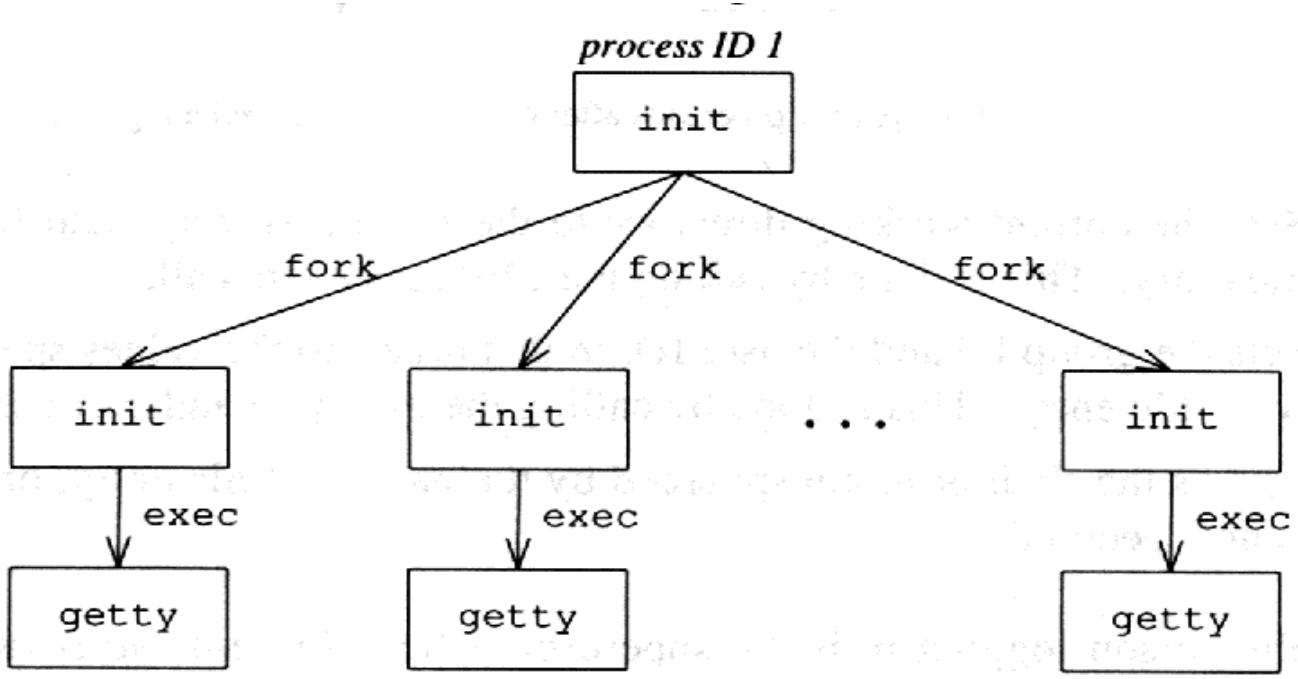


Wait Operation

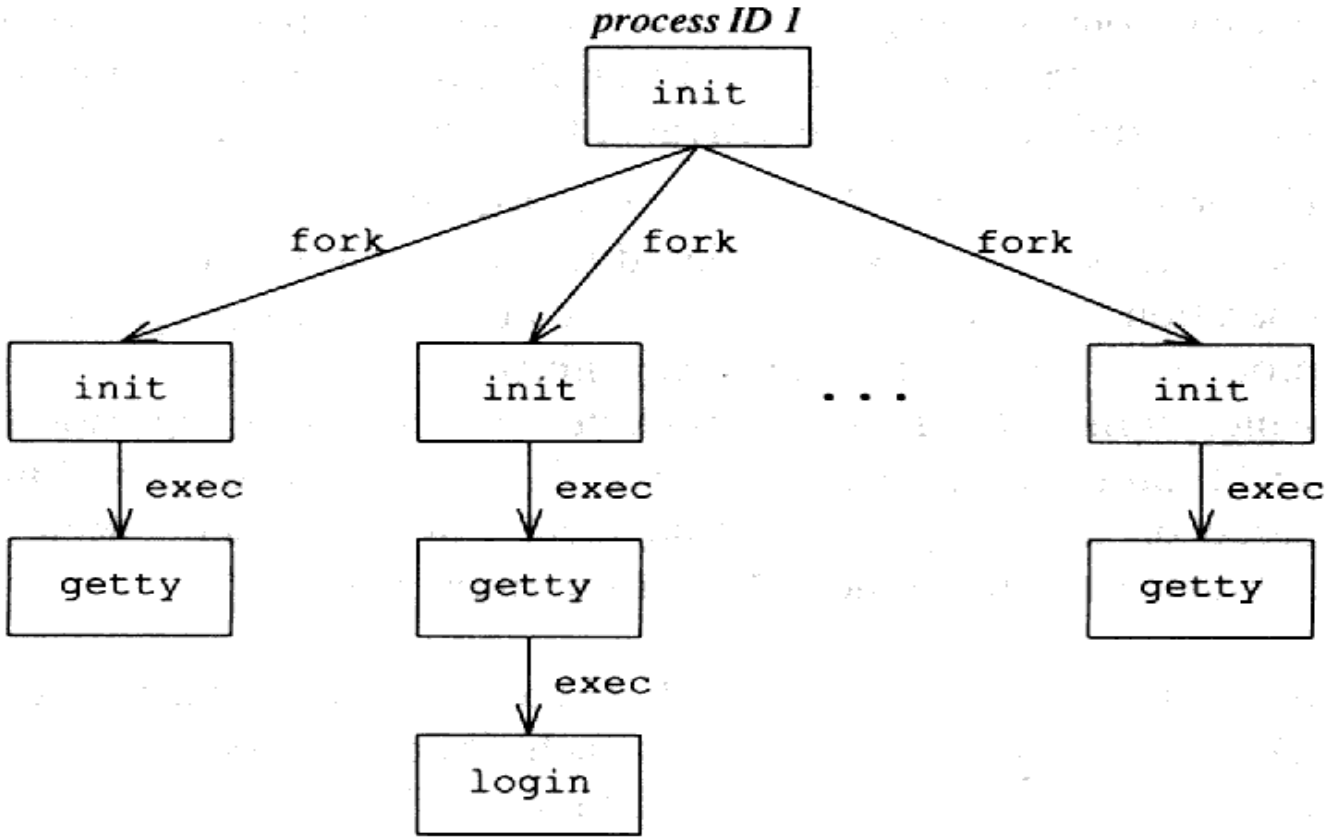
After process A calls `exit`,

- If A's parent has called wait, process A will terminates and parent get the exit status.
- If A's parent has not, process A is marked as zombie process and it will be released later.
- If A has some child processes, these processes are set their parent pid as 1 (init).

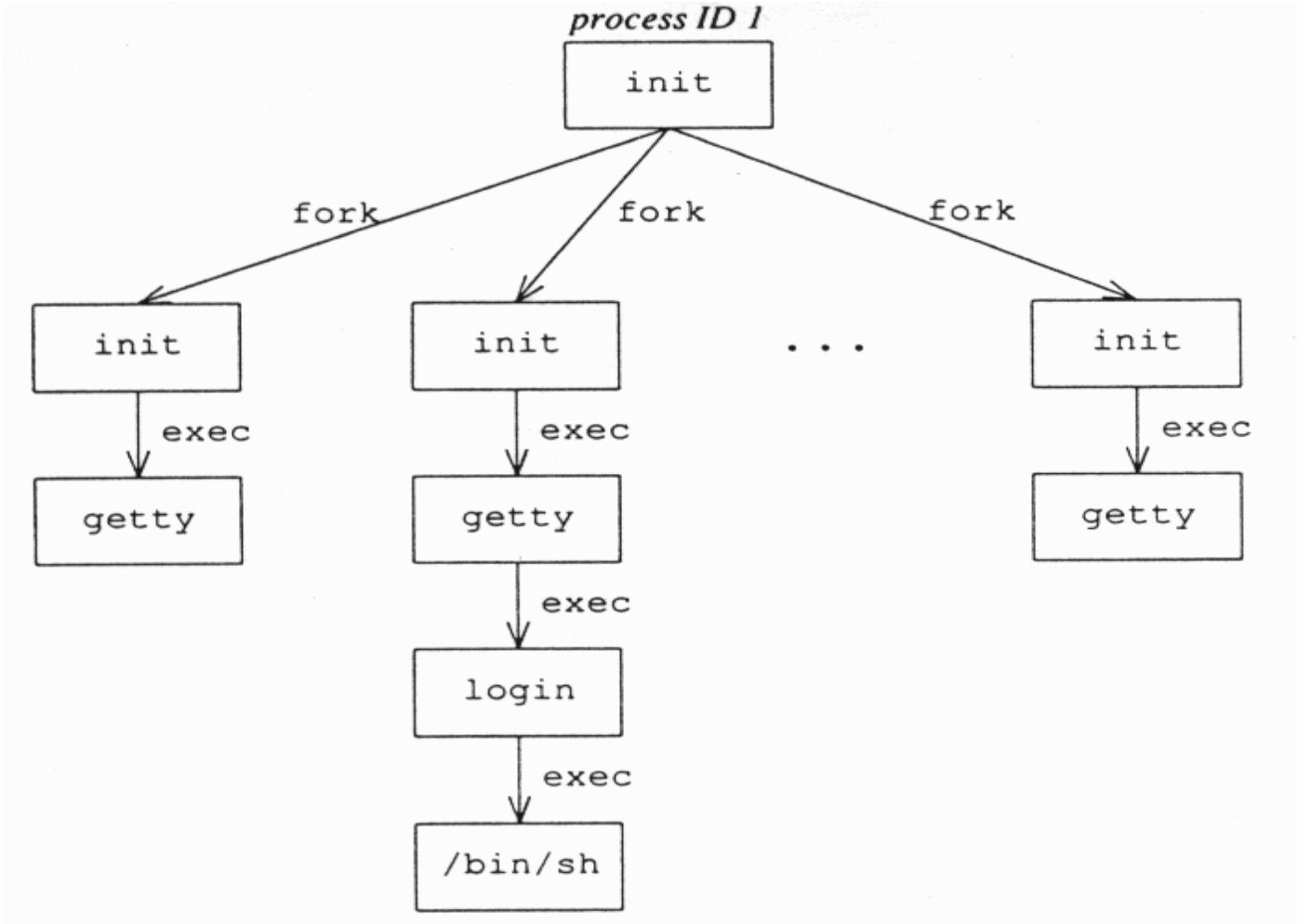
Process Relations

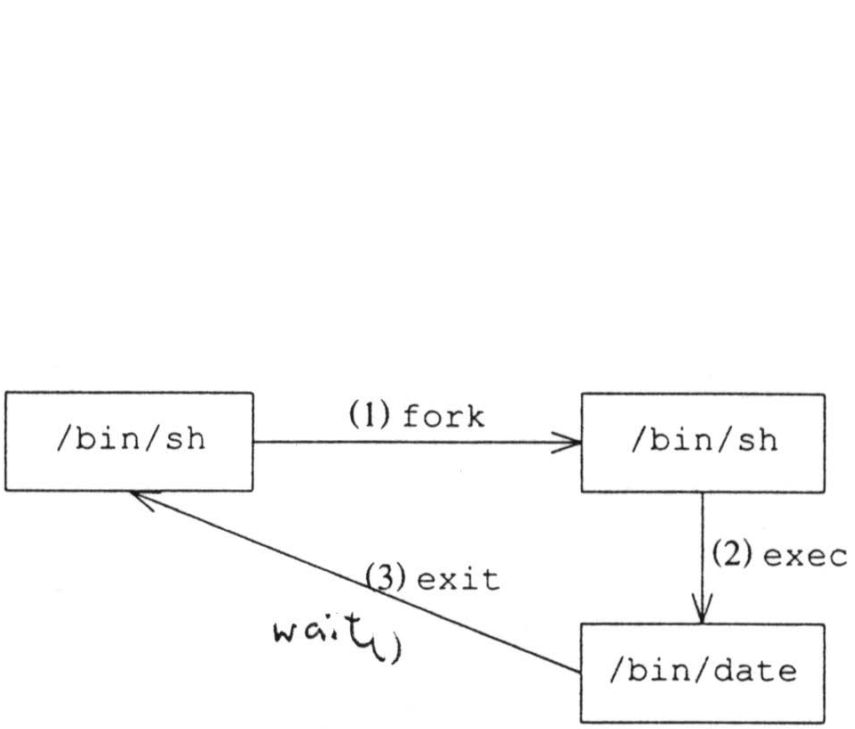


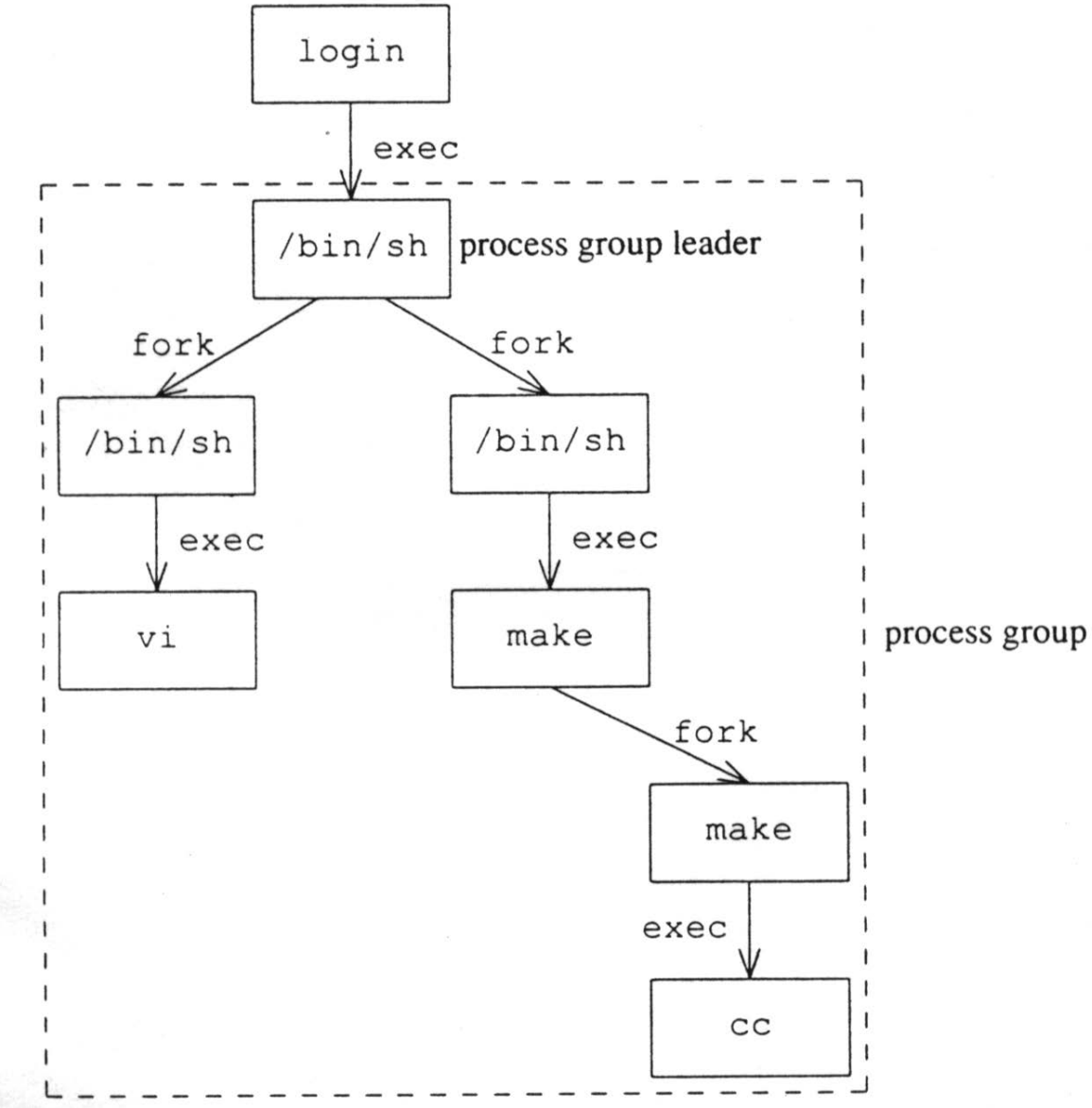
Process Relations

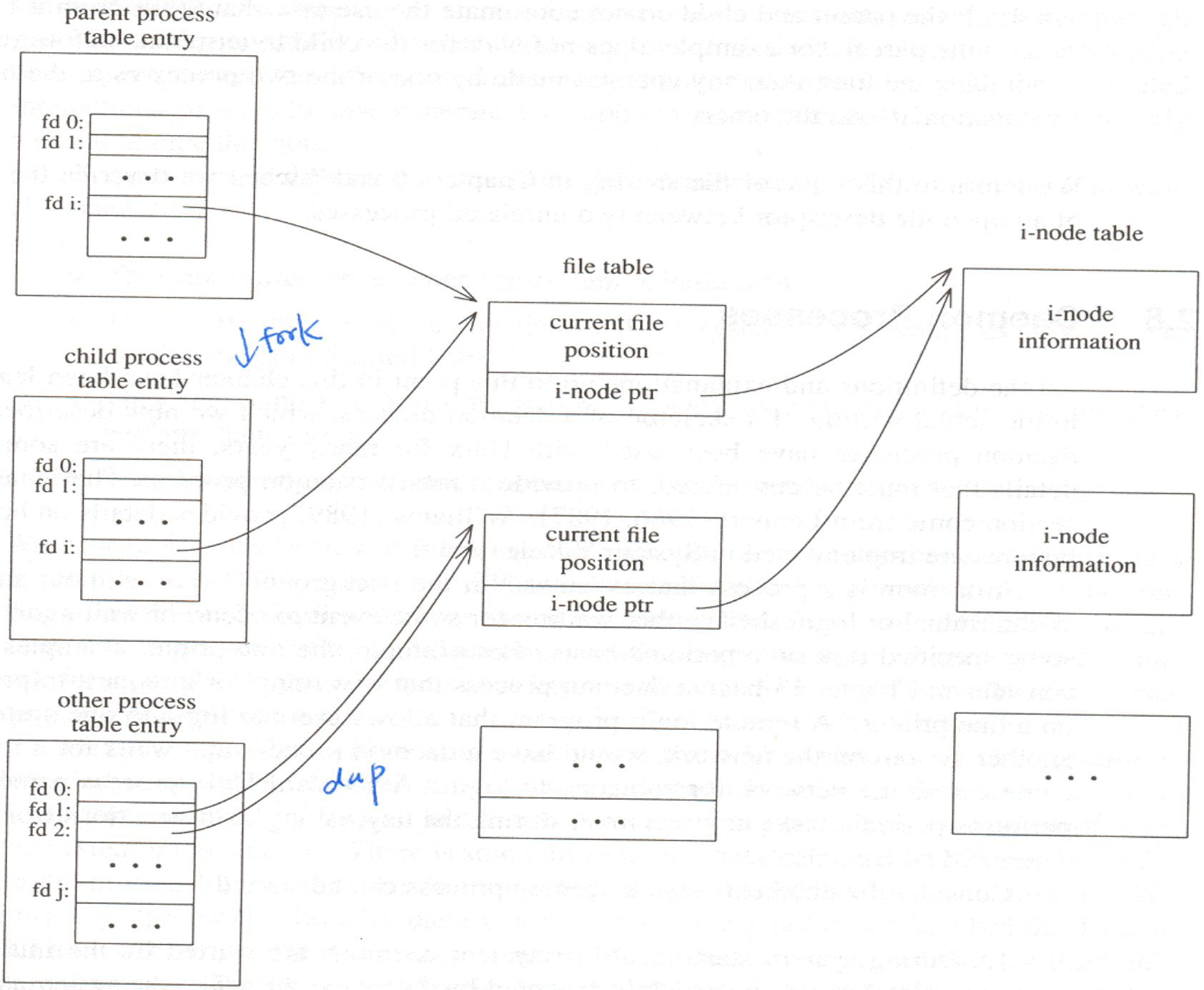


Process Relations




/bin/sh
(1) fork
/bin/sh
(2) exec
(3) exit
wait()
/bin/date


login
exec
/bin/sh
process group leader
fork
fork
/bin/sh
/bin/sh
exec
exec
vi
make
process group
fork
make
exec
cc


parent process
table entry
fd 0:
fd 1:
fd i:
child process
table entry
fd 0:
fd 1:
fd i:
other process
table entry
fd 0:
fd 1:
fd 2:
fd j:
fork
dup
file table
current file
position
i-node ptr
current file
position
i-node ptr
i-node table
i-node
information
i-node
information