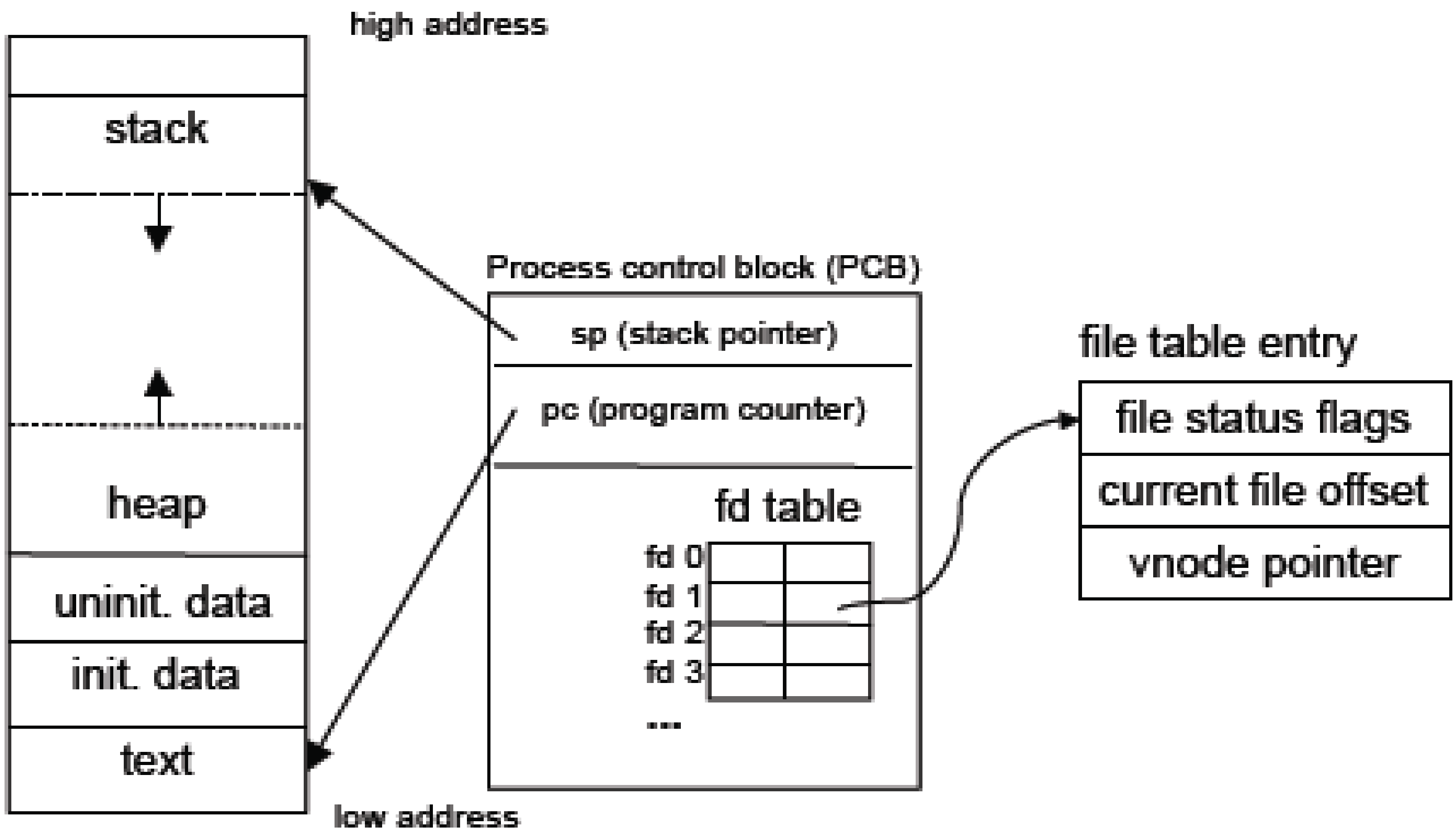
System Programming

Process Management

Processes in Unix

- Process: basic unit of execution
  - Executing instance of a program
  - Has a process ID (PID)
  - Occurs in a hierarchy of processes (parents and children)
  - Root is the `init` process
  - Each process has its own state/context/memory
- Shell commands dealing with processes: `ps, top, kill, nice,`…

Process state



File Objects and File Descriptors

- The stdio library provides FILE objects which handle buffering.
  - FILE *stdin, *stdout, *stderr;
- Why buffering? Efficiency.
- FILE objects are built on top of file descriptors. A file descriptor is an index into a per-process table of open file descriptors.
- We will use file descriptors for process management tasks.

Buffering

- un-buffered – output appears immediately
  - `stderr` is not buffered
- line buffered – output appears when a full line has been written.
  - `stdout` is line buffered when going to the screen
- block buffered – output appears when a buffer is filled or a buffer is flushed (on close or explicit flush).
  - normally output to a file is block buffered
  - `stdout` is block buffered when redirected to a file.

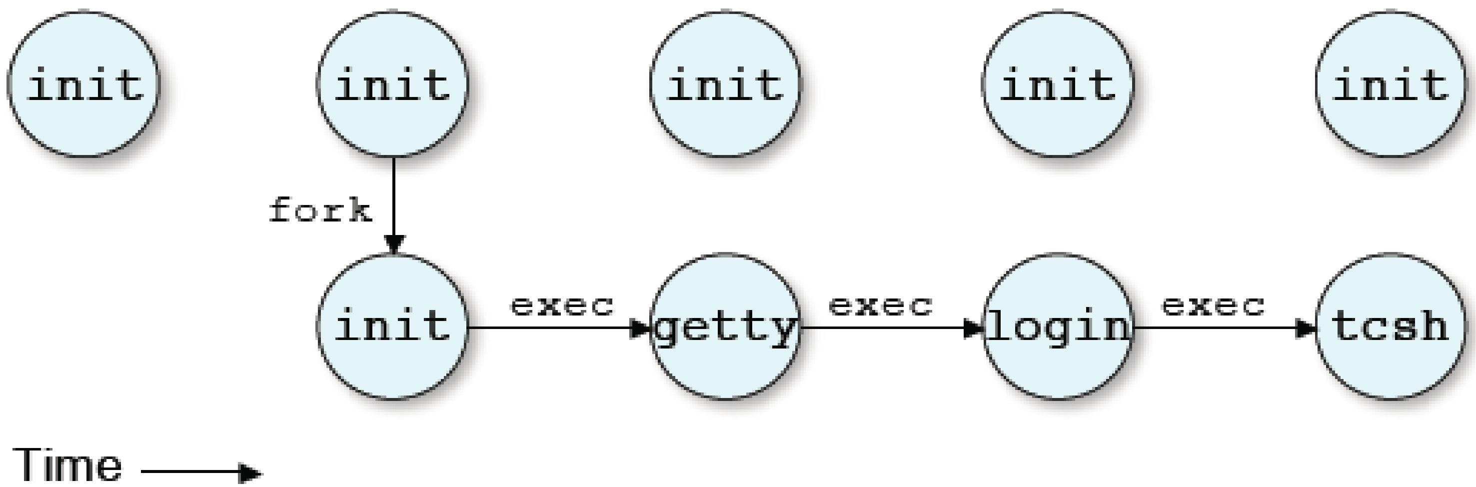
File Descriptors

- Used by low-level I/O
  - `open(), close(), read(), write()`
- declared as an integer
  - `int fd;`
- A useful system call to convert a FILE object to a fd
  - `int fileno( FILE *fp);`
- Of course it is possible to assign a stream interface to a file descriptor
  - `FILE *fdopen(int fd, const char *mode);`

Process Management Issues

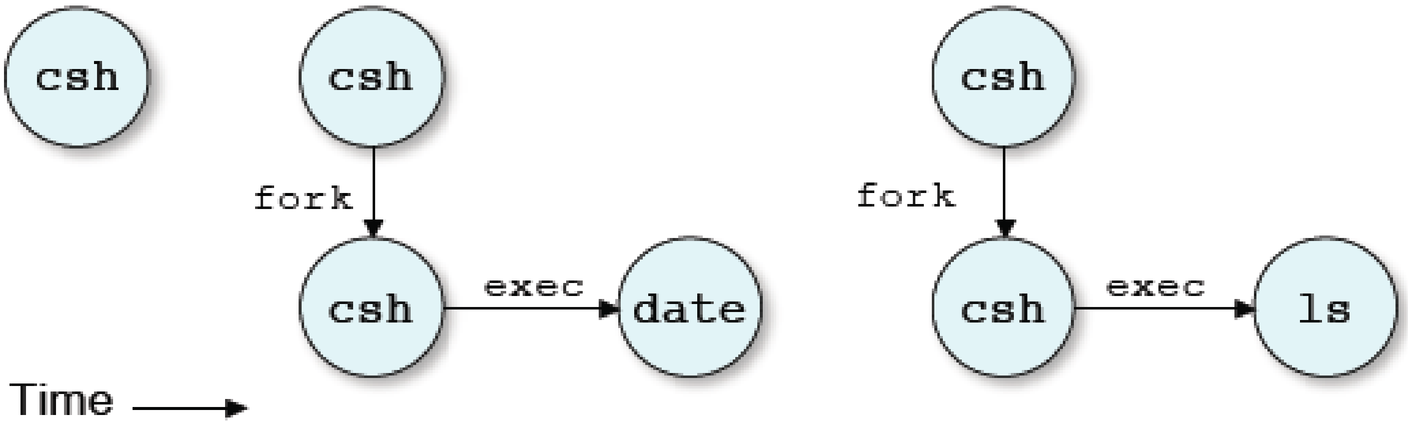
- System calls dealing with:
  - Creating a process
  - Setting the program a process executes
  - Waiting for a process to terminate
  - Terminating a process
  - Sending signals to a process
  - Communicating between processes

Initializing Unix



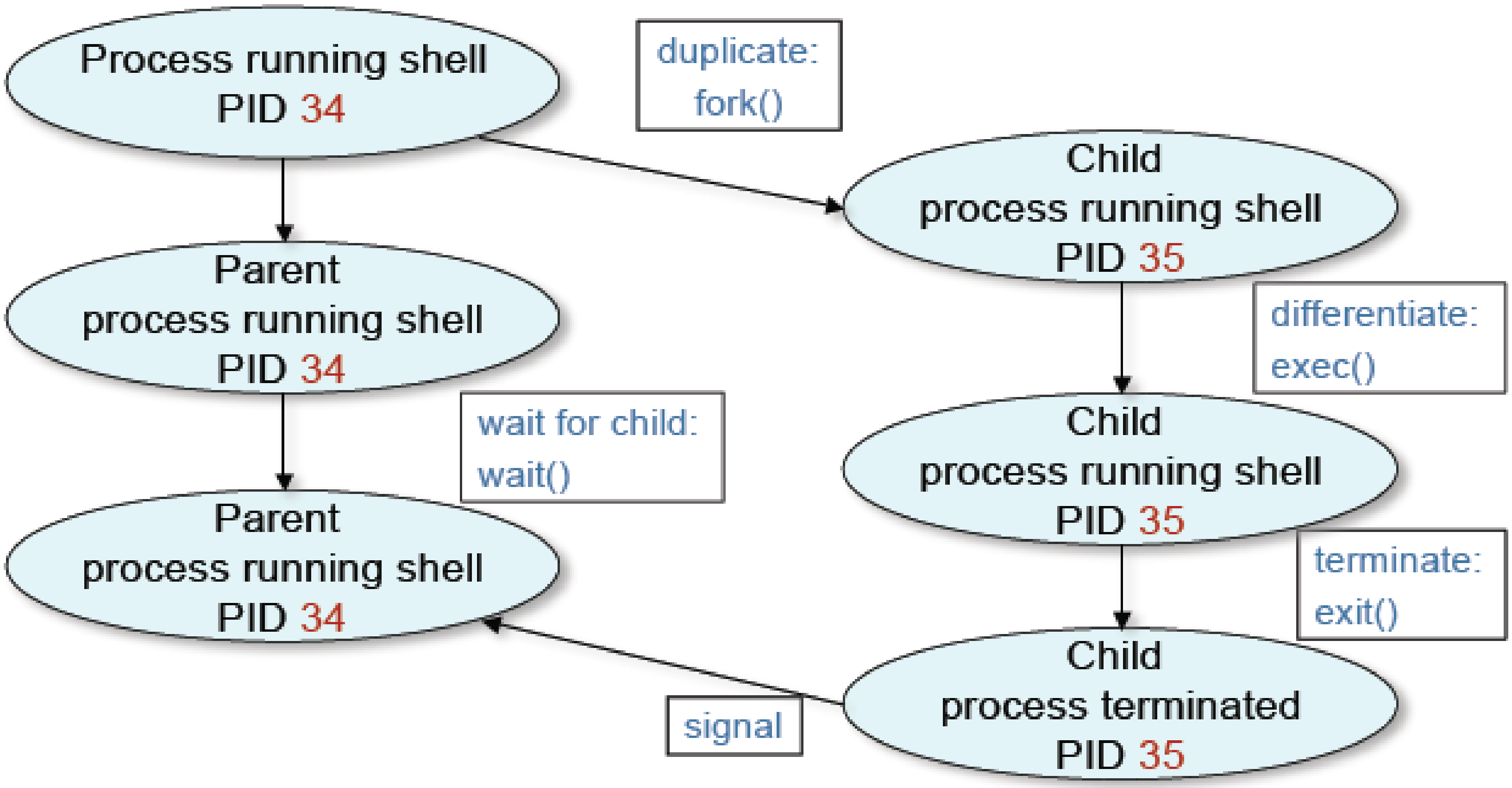
- See “top”, “ps –aux” to see what’s running
- The **only** way to create a new process is to duplicate an existing process. Therefore the ancestor of **all** processes is `init` with pid = 1

How csh runs commands



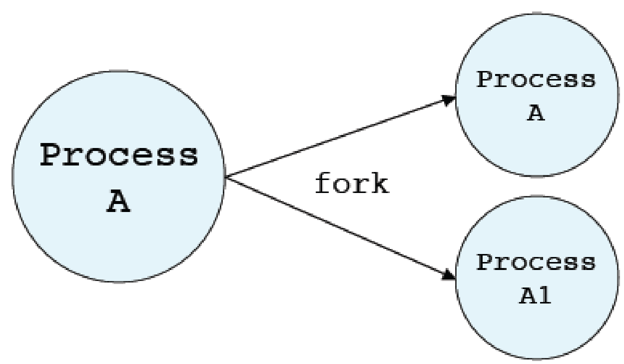
- When a command is typed, csh forks and then execs the typed command.
- After the fork, file descriptors 0, 1, and 2 still refer to stdin, stdout, and stderr in the new process.
- By convention, the executed program will use these descriptors appropriately.

How csh runs


Process running shell
PID 34
duplicate:
fork()
Child
process running shell
PID 35
Parent
process running shell
PID 34
differentiate:
exec()
wait for child:
wait()
Child
process running shell
PID 35
Parent
process running shell
PID 34
terminate:
exit()
Child
process terminated
PID 35
signal

Process Creation

- The `fork` system call creates a duplicate of the currently running program.
- The duplicate (child process) and the original (parent process) both proceed from the point of the `fork` with exactly the same data.
- The only difference is the return value from the `fork` call.



Fork: PIDs and PPIDs

- System call: `int fork()`
- If `fork()` succeeds it returns the child PID to the parent and returns 0 to the child;
- If `fork()` fails, it returns -1 to the parent (no child is created)
- Related system calls:
  - `int getpid()` – returns the PID of current process
  - `int getppid()` – returns the PID of parent process (ppid of 1 is 1)
  - `int getpgrp()` – returns the group ID of current process

When `fork()` fails

- Two possible reasons
- There is a limit to the maximum number of processes a user can create.
  - Once this limit is reached (i.e., process table full), subsequent calls to `fork()` return -1.
- The kernel allocates virtual memory to processes
  - When it runs out of memory, `fork` calls fails.

`fork()` properties

- Properties of parent inherited by child:
  - UID, GID
  - controlling terminal
  - CWD, root directory
  - signal mask, environment, resource limits
  - shared memory segments
- Differences between parent and child
  - PID, PPID, return value from fork()
  - pending alarms cleared for child
  - pending signals are cleared for child

fork() example

```
int i, pid;
i = 5;
printf(“%d\n”, i);
pid = fork();

if(pid != 0)
   i = 6; /* only parent gets here */
else
   i = 4; /* only child gets here */
printf(“%d\n”, i);
```

Fork Example

Original process (parent)

```
int i, pid;
i = 5;
printf("%d\n", i);
/* prints 5 */
pid = fork();
/* pid == 677 */
if(pid != 0)
   i = 6;
else
   i = 4;
printf("%d\n", i);
/* prints 6 */
```

Child process

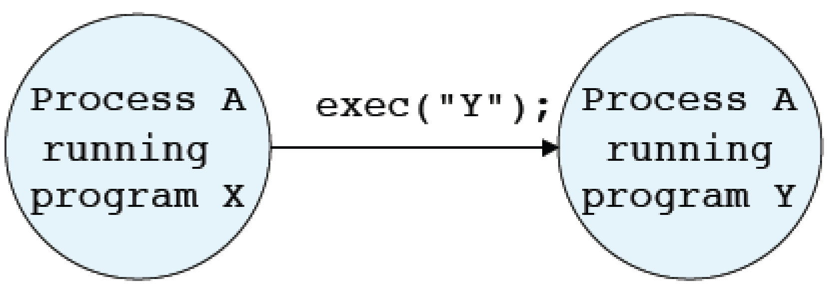
```
int i, pid;
i = 5;
printf("%d\n", i);

pid = fork();
/* pid == 0 */
if(pid != 0)
   i = 6;
else
   i = 4;
printf("%d\n", i);
/* prints 4 */
```

PID/PPID Example

```
#include <stdio.h>
int main(void) {
   int pid;
   printf("ORIG: PID=%d PPID=%d\n",
            getpid(), getppid());
   pid = fork();
   if(pid != 0)
      printf("PARENT: PID=%d PPID=%d\n",
            getpid(), getppid());
   else
      printf("CHILD: PID=%d PPID=%d\n",
            getpid(), getppid());
   return(1);
}
```

Executing a Program

- The `exec` system call replaces the program being run by a process by a different one.
- The new program starts executing from the beginning.
- On success, `exec` never returns, on failure, `exec` returns -1.



Exec example

<u>Program X</u>

```
int i = 5;
printf(“%d\n”, i);
exec(“Y”);
printf(“%d\n”, i);
```

<u>Program Y</u>

```
printf(“hello\n”);
```

exec( ) properties

- New process inherits from calling process:
  - PID and PPID, real UID, GID, session ID
  - controlling terminal
  - CWD, root directory, resource limits
  - process signal mask
  - pending signals
  - pending alarms
  - file mode creation mask (umask)
  - file locks

exec()

- Six versions exec():
  - `execl(char *path, char *arg0, …, (char *)0);`
  - `execv(char *path, char *argv[]);`
  - `execle(char *path, char *arg0, …, (char *)0, char *envp[]);`
  - `execve(char *pathname, char *argv[], char *envp[]);`
  - `execlp(char *file, char *arg0, …, (char *)0);`
  - `execvp(char *file, char *argv[]);`

Waiting for a Process to Terminate

- System call to wait for a child
  - `int wait(int *status)`
- A process that calls `wait()` can:
  - block (if all of its children are still running)
  - return immediately with the termination status of a child (if a child has terminated and is waiting for its termination status to be fetched)
  - return immediately with an error (if it doesn't have any child processes.
- Other `wait()` calls
  - wait3(), wait4()

Exit Status...macros

- WIFEXITED(status): true if process called _exit(2) or exit(3)
- WEXITSTATUS(status): The low-order 8 bits of the argument passed to _exit(2)
- WIFSIGNALED(status): True if the process terminated due to receipt of a signal.
- WTERMSIG(status): The number of the signal that caused the termination.
- WCOREDUMP(status): True if a core file was created.
- WIFSTOPPED(status): True if the process has not terminated, but has stopped and can be
- restarted.
- WSTOPSIG(status): Number of signal that stopped process.

Zombies

- A zombie process:
  - a process that is "waiting" for its parent to accept its return code
  - a parent accepts a child's return code by executing `wait()`
  - shows up as Z in ps –a
  - A terminating process may be a (multiple) parent; the kernel ensures all of its children are orphaned and adopted by init.

wait and waitpid

- `wait()` can
  - block
  - return with termination status
  - return with error
- If there is more than one child `wait()` returns on termination of any children
- `waitpid` can be used to wait for a specific child pid.
- `waitpid` also has an option to block or not to block

wait and waitpid

- `waitpid` has an option to block or not to block
- `pid_t waitpid(pid, &status, option);`
  - `if pid == -1` wait for any child process
  - `if pid == 0` wait for any child process in the process group of the caller
  - `if pid > 0` wait for the process with pid
  - `if option == WNOHANG` non-blocking
  - `if option == 0` blocking
- `waitpid(-1, &status, 0);` equivalent to `wait(&status);`

Example of wait

```
#include <sys/types.h>
#include <sys/wait.h>
   int main(void) {
   int status;
   if(fork() == 0) exit(7); /*normal*/
   wait(&status); prExit(status);
   if(fork() == 0) abort(); /*SIGABRT*/
   wait(&status); prExit(status);
   if(fork() == 0) status /= 0; /*FPE*/
   wait(&status) prExit(status);
}
```

prExit.c

```
#include <sys/types.h>
void prExit(int status) {
  if(WIFEXITED( status ) )
    printf(“normal termination\n”);
  else if(WIFSTOPPED( status ))
    printf(“child stopped, signal no.=
           %d\n”, WSTOPSIG(status));
  else if(WIFSIGNALLED( status ) )
    printf(“abnormal termination, signal
           no.= %d\n”,WTERMSIG(status));
}
```

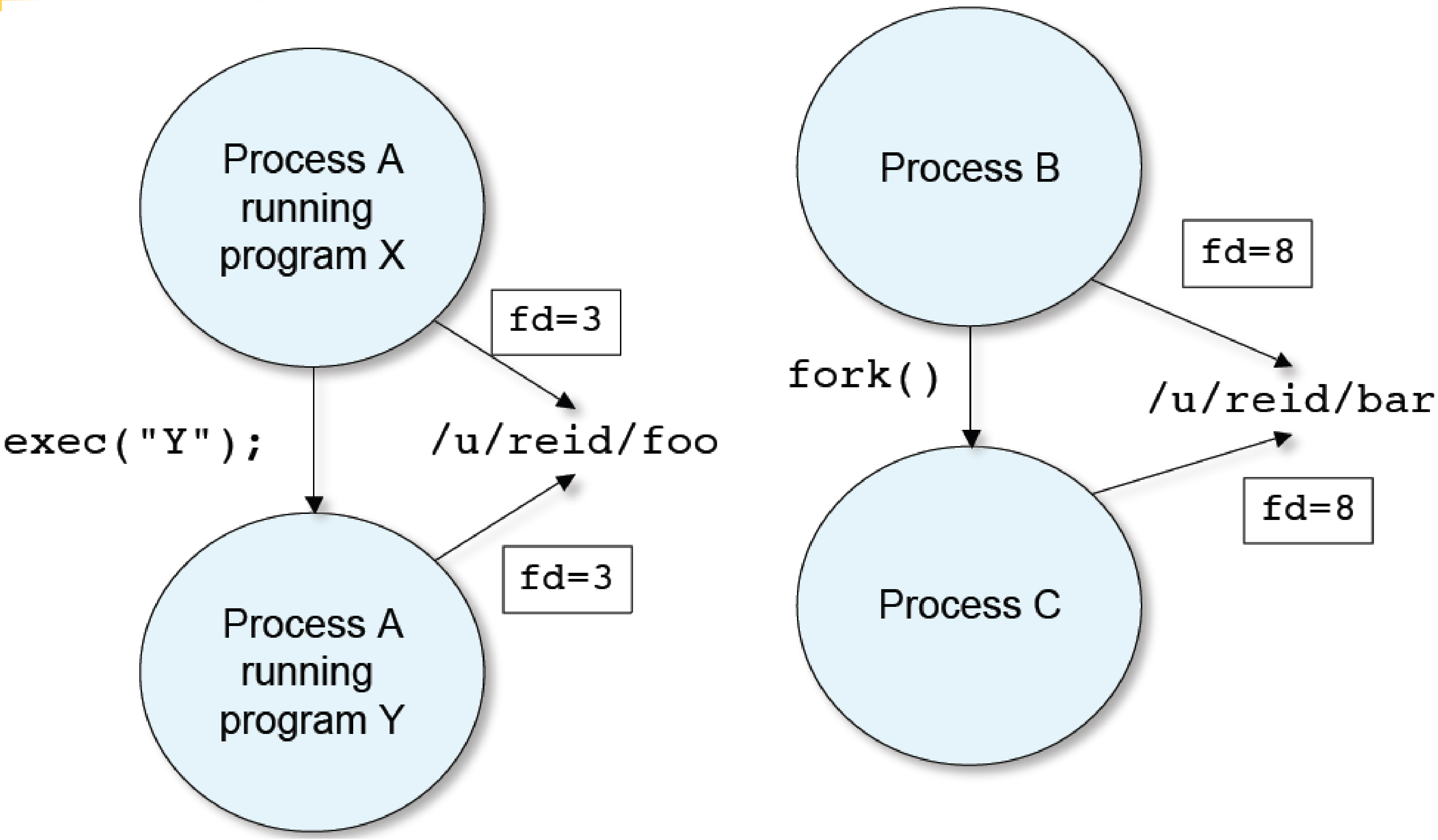
Process Termination

- Orphan process:
  - a process whose parent is the init process (PID 1) because its original parent died before it did.
- Terminating a process: `exit(int status)`
  - all open file descriptors are closed.
- Every normal process is a child of some parent, a terminating process sends its parent a SIGCHLD signal and waits for its termination status to be accepted.
- The Bourne shell stores the termination code of the last command in $?.

Processes and File Descriptors

- File descriptors are handles to open files.
- They belong to processes not programs.
- They are a process's link to the outside world.

FDs preserved across fork and exec



Signals

- Unexpected/unpredictable asynchronous events
  - floating point error
  - death of a child
  - interval timer expired (alarm clock)
  - control-C (termination request)
  - control-Z (suspend request)
- Events are called interrupts
- When the kernel recognizes an event, it sends a signal to the process.
- Normal processes may send signals.

What are signals for?

- When a program forks into two or more processes, rarely do they execute independently.
- The processes usually require some form of synchronization, often handled by signals.
- To transfer data between processes, we can use pipes and sockets.
- Signals are generated by
  - machine interrupts
  - the program itself, other programs or the user.

Software Interrupts

- <sys/signal.h> lists the signal types on CDF.
- "kill –l" gives a whole list of signals.
- "man 7 signal" ("man 5 signal" on Solaris) gives some description of various signals
  - SIGTERM, SIGABRT, SIGKILL
  - SIGSEGV, SIGBUS
  - SIGSTOP, SIGCONT
  - SIGCHLD
  - SIGPIPE

Signal table

- For each process, Unix maintains a table of actions that should be performed for each kind of signal.
- Here are a few.

Signal	**Default Action**	**Comment**
SIGINT	Terminate	Interrupt from keyboard
SIGSEGV	Terminate/Dump core	Invalid memory reference.
SIGKILL	Terminate/Dump core (cannot ignore)	Kill
SIGCHLD	Ignore	Child stopped or terminated.
SIGSTOP	Stop (cannot ignore)	Stop process.
SIGCONT		Continue if stopped.

Sending a signal

- From the command line use

  `kill [-signal] pid [pid]…`

  - sometimes built into shells (bash)
- If no signal is specified, `kill` sends the TERM signal to the process.
- signal can be specified by the number or name without the SIG.
- Examples:
  - `kill -QUIT 8883`
  - `kill -STOP 78911`
  - `kill -9 76433 (9 == KILL)`

Signaling between processes

- One process can send a signal to another process using the misleadingly named function call.

```
kill( int pid, int sig );
```

- This call sends the signal `sig` to the process pid
- Signaling between processes can be used for many purposes:
  - kill errant processes
  - temporarily suspend execution of a process
  - make a process aware of the passage of time
  - synchronize the actions of processes.

kill usage

`retval = kill(`*`pid, signal`*`)`

- pid>0 => to that process
- pid=0 => to process group of sender
- pid=-1 => all processes (but sender)
  - root -> all but system processes
  - !root -> all with same uid
- Example
  - `kill(1234, SIGINT);`
  - Sends an interrupt (signal) of type SIGINT
  - Does not block sending process
  - The process, whose ID is 1234, gets the signal.

What happens at "signal time"?

- Signal gets "Delivered" to the process
- Actions ...
  - Ignore the signal -- nothings happens
    - (Can’t ignore SIGKILL and SIGSTOP)
  - Catch the signal
    - Starts a designated function
    - (Can’t catch SIGKILL and SIGSTOP)
  - Default action
    - May ignore it
    - May terminate the process
    - May dump core and terminate process

Signal Handling

- A function can handle a delivered signal.

  void (*signal(int sig, void (*func)(int));)()

  - sig -> signal name
  - func -> function name, SIG_DFL or SIG_IGN
  - return -> previous function pointer (or SIG_DFL or SIG_IGN)
  - func is called outside of normal execution order.
- Example: `signal(SIGINT, func);`
  - A process which gets the signal `SIGINT` calls the function `func`
  - Some UNIXes set this only for the next signal delivery, requiring signal function to reset it again.

Timer signals

- Three interval timers are maintained for each process:
  - SIGALRM (real-time alarm, like a stopwatch)
  - SIGVTALRM (virtual-time alarm, measuring CPU time)
  - SIGPROF (used for profilers)
- Useful functions to set and get timer info:
  - sleep() – cause calling process to suspend.
  - usleep() – like sleep() but at a finer granularity.
  - alarm() – sets SIGALRM
  - pause() – suspend until next signal arrives
  - setitimer(), getitimer()
- sleep() and usleep() are interruptible by other signals.

"Advanced" signal interface

```
#include <signal.h>
struct sigaction {
      void (*sa_handler)(int);
      sigset_t sa_mask;
      int sa_flags;   };
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

- sa_mask -- a "set" of signals to "block" during handler running.
  - Routines to make signal sets:
    - sigemptyset, sigfillset, sigaddset, sigdelset, sigismember
- sa_flags -- Controls other things
  - SA_RESTART -- restart system calls that can be restarted

Inter-Process Communication (IPC)

- Exchanging data between processes
- We cannot use variables to communicate between processes since they each have separate address spaces, and separate memory.
- One easy way to communicate is to use files.
  - process A writes to a file and process B reads from it.
- Two basic paradigms
  - Message passing: processes send information back and forth in messages/ packets
  - Shared Memory: processes share a chunk of physical memory and read/ write data there to share that information

Pipes and File Descriptors

- After `fork()` is called we end up with two independent processes.
- A forked child inherits file descriptors from its parent
- `pipe()` creates an internal system buffer and two file descriptors, one for reading and one for writing.
- After the pipe call, the parent and child should close the file descriptors for the opposite direction. Leaving them open does not permit full-duplex communication.

Pipelines and Job Control

- Want entire pipeline as a single process group.

  ps aux | grep dhcli | grep -v grep | cut -c5-10
- fork a process to do entire pipeline and be group leader
- sh -> fork -> sh2

  sh2 -> fork -> exec ps

  -> fork -> exec grep

  -> fork -> exec grep

  -> exec cut

  OR

  -> fork -> exec cut

  -> wait for all children

Producer/Consumer Problem

- Simple example: `who | wc -l`
- Both the writing process (`who`) and the reading process (`wc`) of a pipeline execute concurrently.
- A pipe is usually implemented as an internal OS buffer.
- It is a resource that is concurrently accessed by the reader and the writer, so it must be managed carefully.

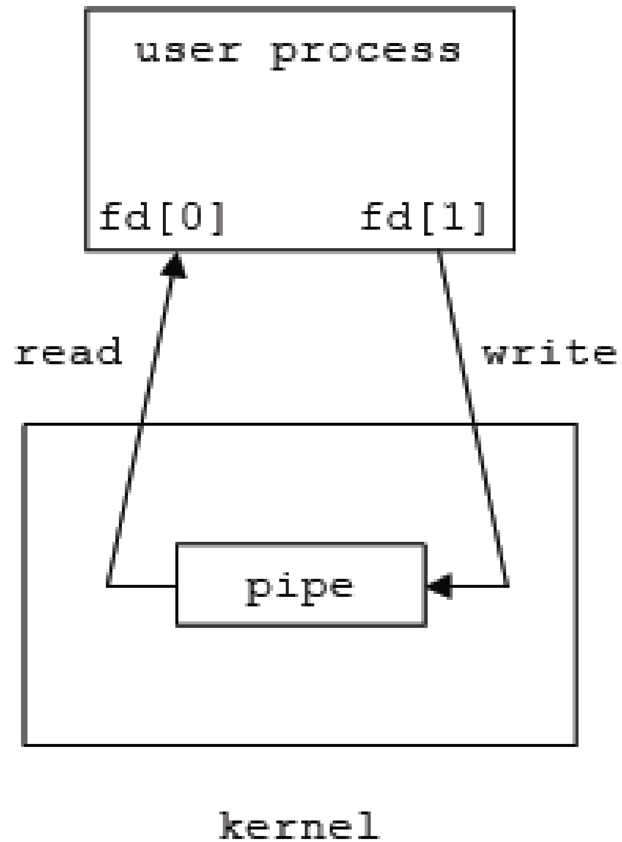
Producer/Consumer

- Consumer blocks when buffer is empty
- Producer blocks when buffer is full
- They should run independently as far as buffer capacity and contents permit
- They should never be updating the buffer at the same instant (otherwise data integrity cannot be guaranteed)
- Harder problem if there is more than one consumer and/or more than one producer.

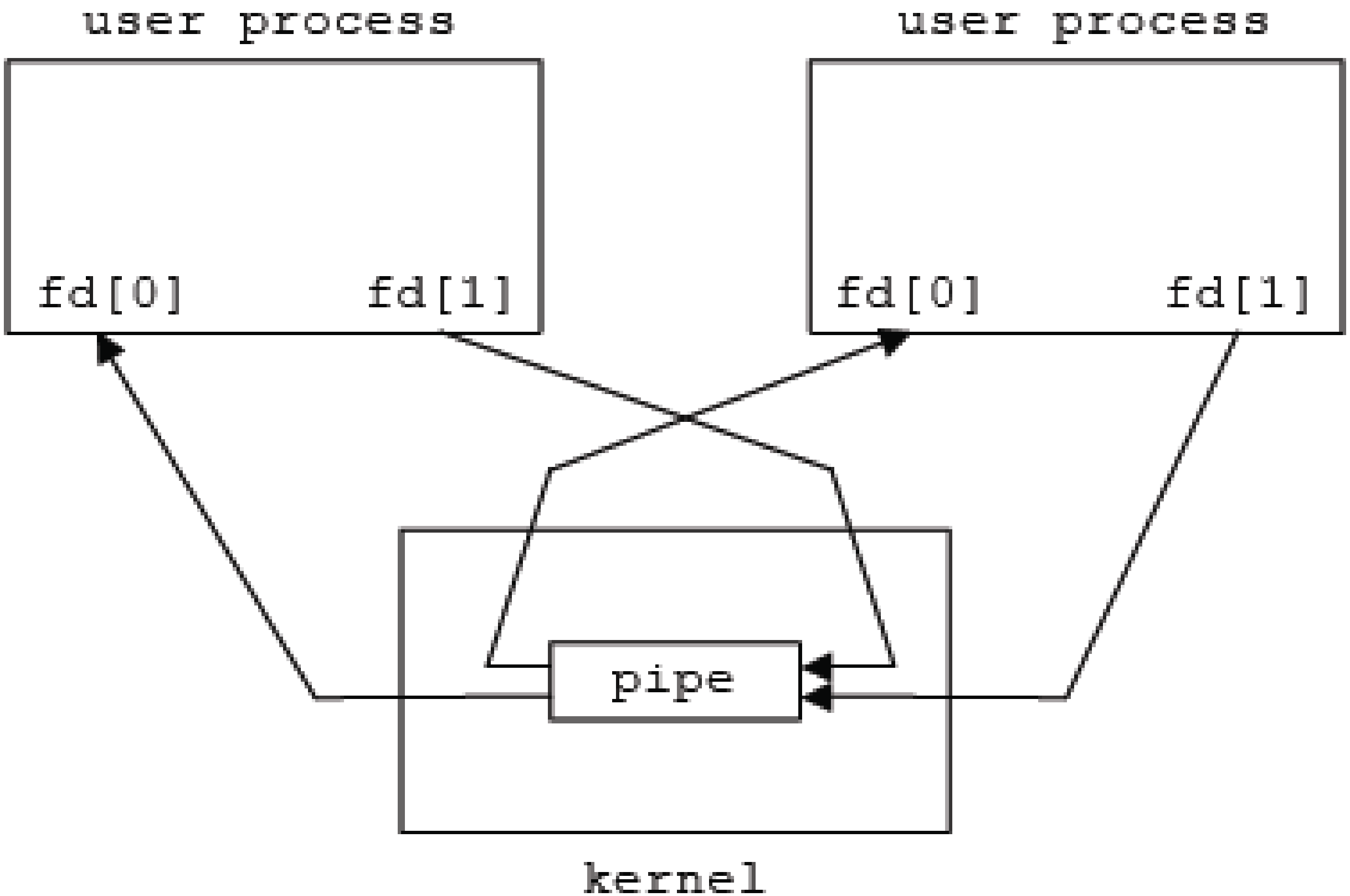
Case study

```
int pipe(int filedes[2])
```

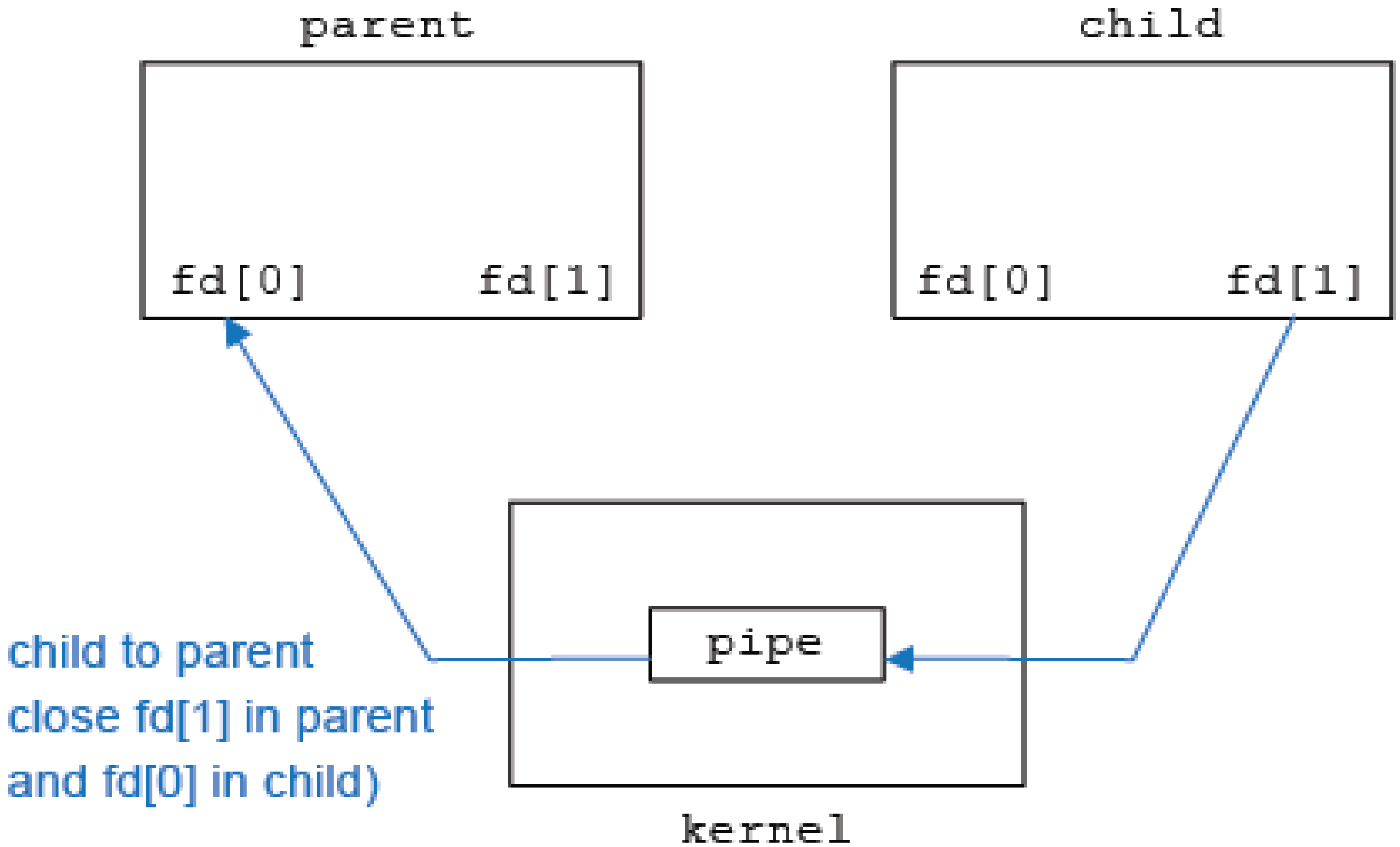
- half-duplex (one-way) communication



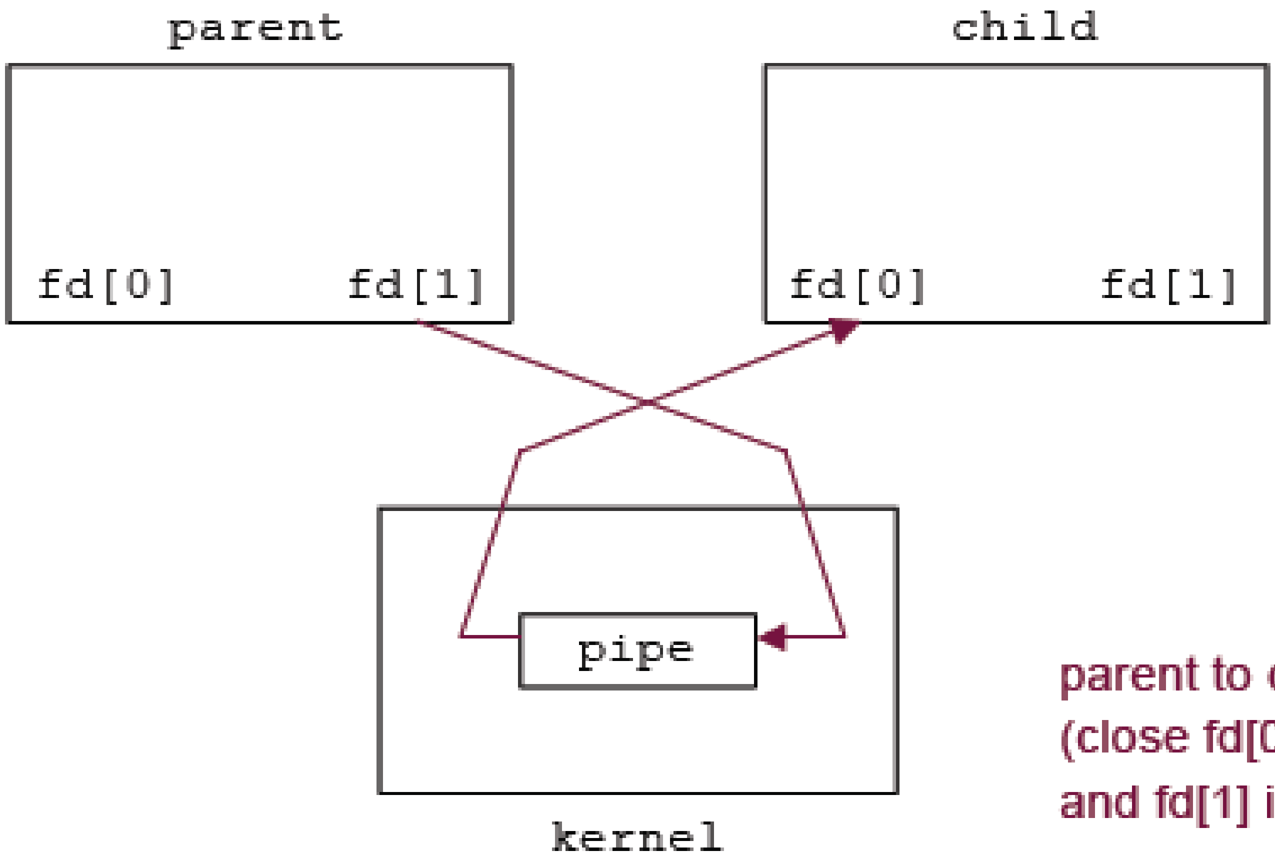
What happens after fork?



Direction of data flow?



Direction of data flow?



parent to child
(close fd[0] in parent
and fd[1] in child)

IPC with Pipes

- Example of message passing
- `int fds[2];`

  `retval = pipe(fds);`
- Creates two file descriptors (a pipe is a file), the first for reading, and the second for writing
- How does another process connect to this pipe?

dup and dup2()

- Often we want the stdout of one process to be connected to the stdin of another process.

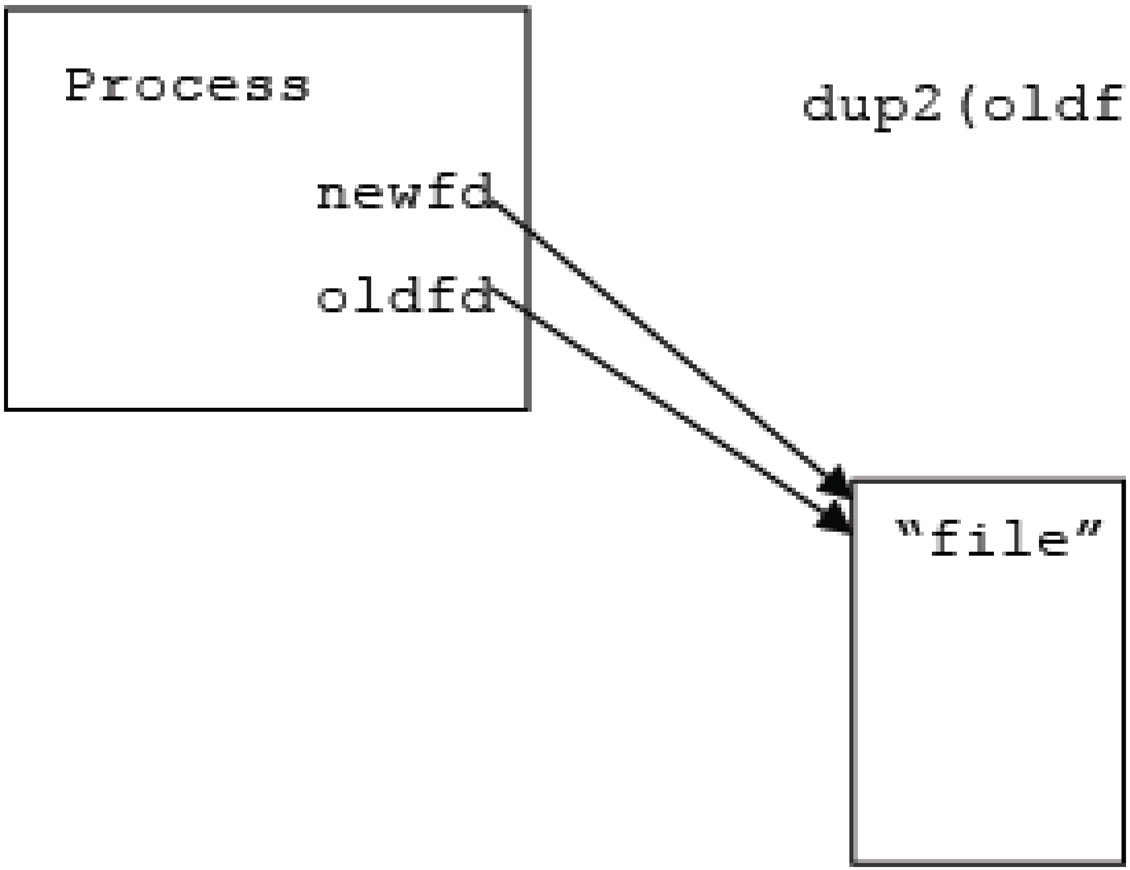
  int dup(int oldfd);

  int dup2(int oldfd, int newfd);

  - Copies oldfd pointer to newfd location.
  - Does not change open file table, just process fd table.
  - Two process fd entries point to the same open file table entry.
- dup: returns first unused fd in process table
- dup2: if newfd is open, it is first closed, then dup(oldfd) is called.
  - Note that `dup2()` refer to fds not streams

dup2()

```
oldfd = open("file");

dup2(oldfd, newfd);
```



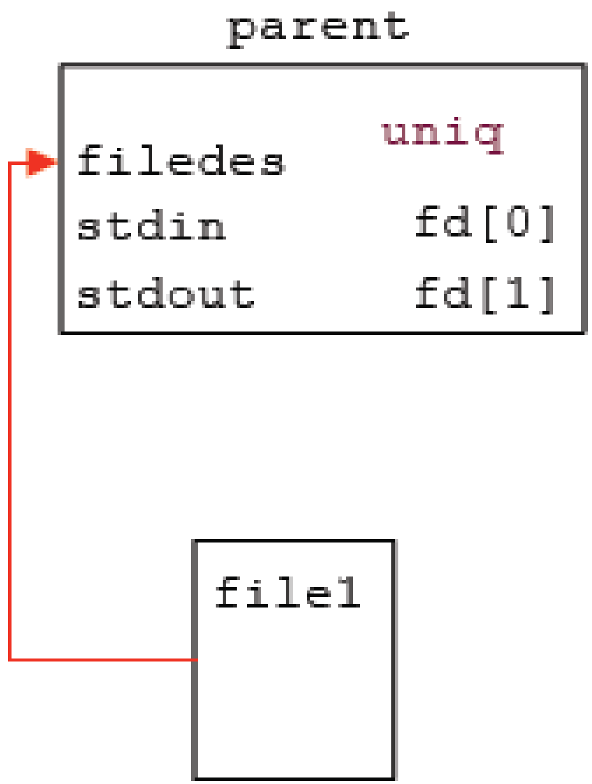
pipe()/dup2() example

```
/* equivalent to “sort < file1 |
  uniq” */
int fd[2], pid;
int filedes = open(“file1”,
  O_RDONLY);
dup2(filedes, fileno(stdin));
close(filedes);
pipe(fd);
```

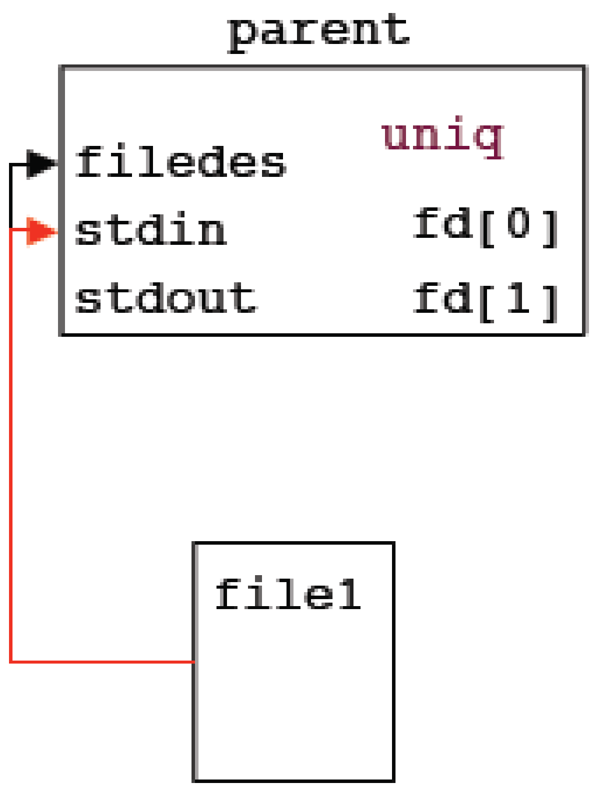
pipe()/dup2() example

```
if((pid = fork()) == 0) {/* child */
  dup2(fd[1], fileno(stdout));
  close(fd[0]); close(fd[1]);
  execl("/usr/bin/sort", "sort", (char *)
  0);
} else if(pid > 0){ /* parent */
  dup2(fd[0], fileno(stdin));
  close(fd[1]); close(fd[0]);
  execl("/usr/bin/uniq", "uniq", (char *)
  0);
} else {
  perror("fork"); exit(1);
}
```

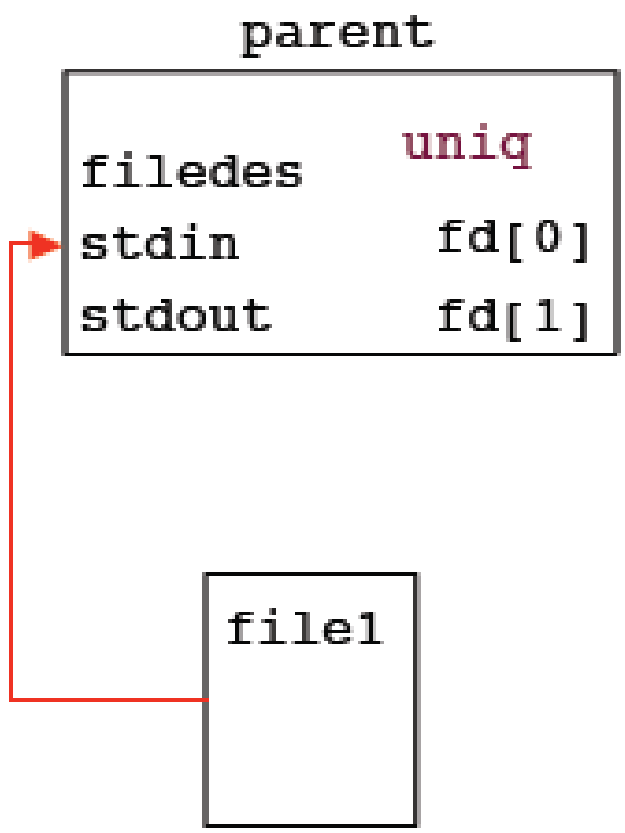
```
int filedes = open("file1", O_RDONLY);
```



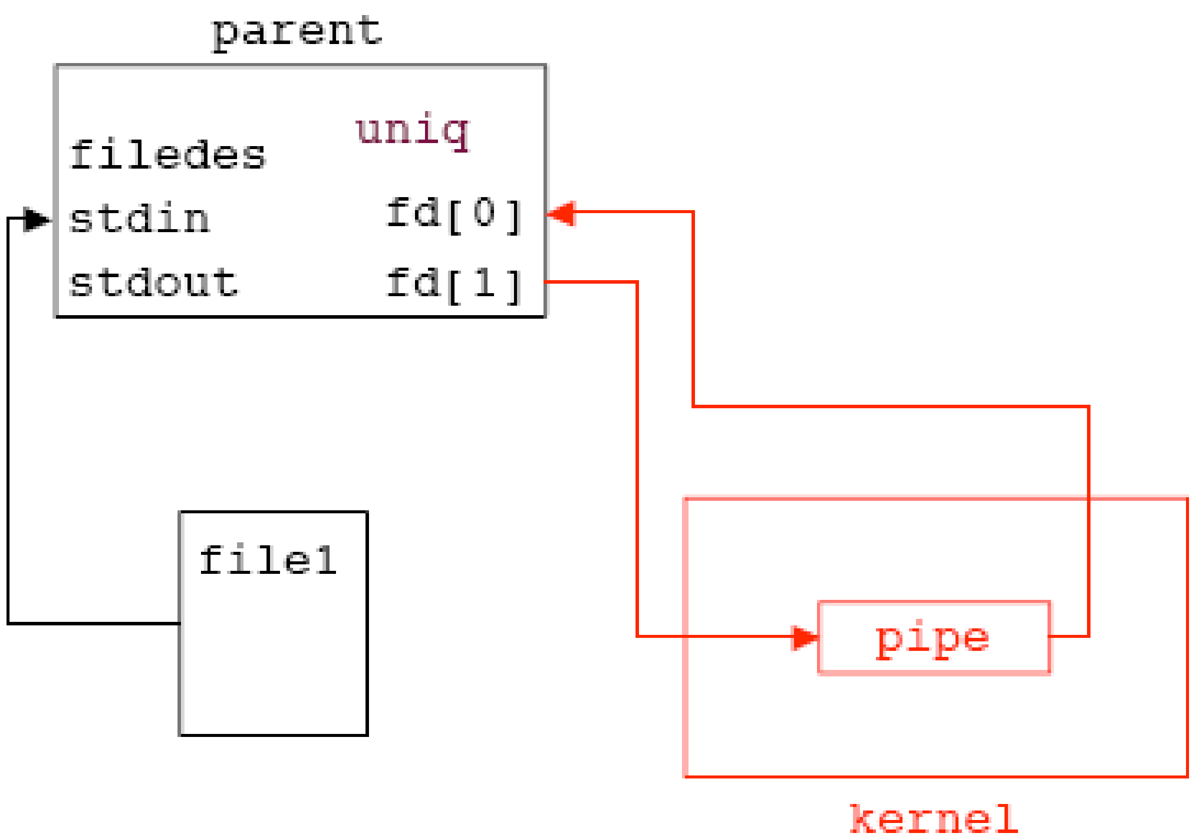
```
dup2(filedes, fileno(stdin));
```



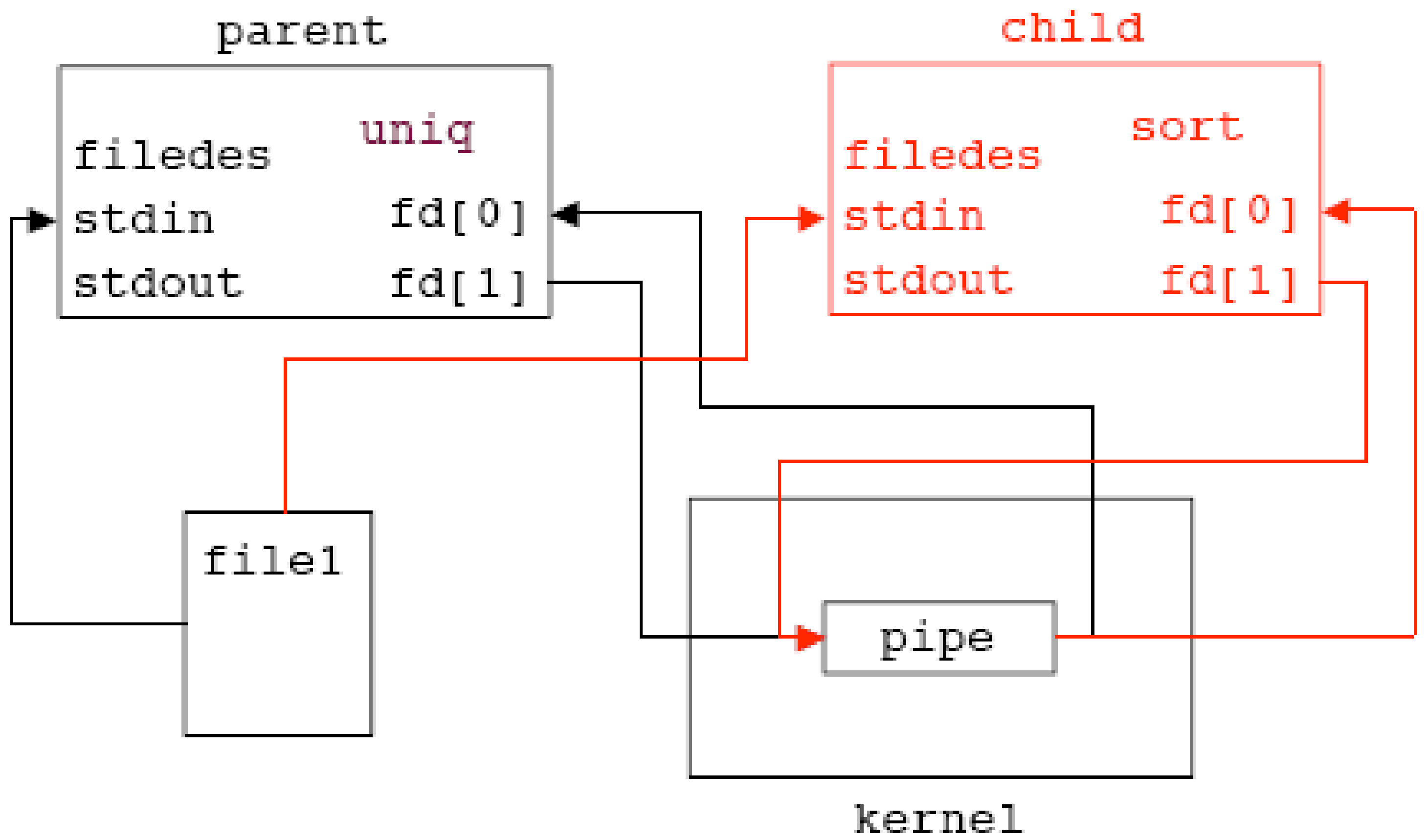
`close(filedes);`



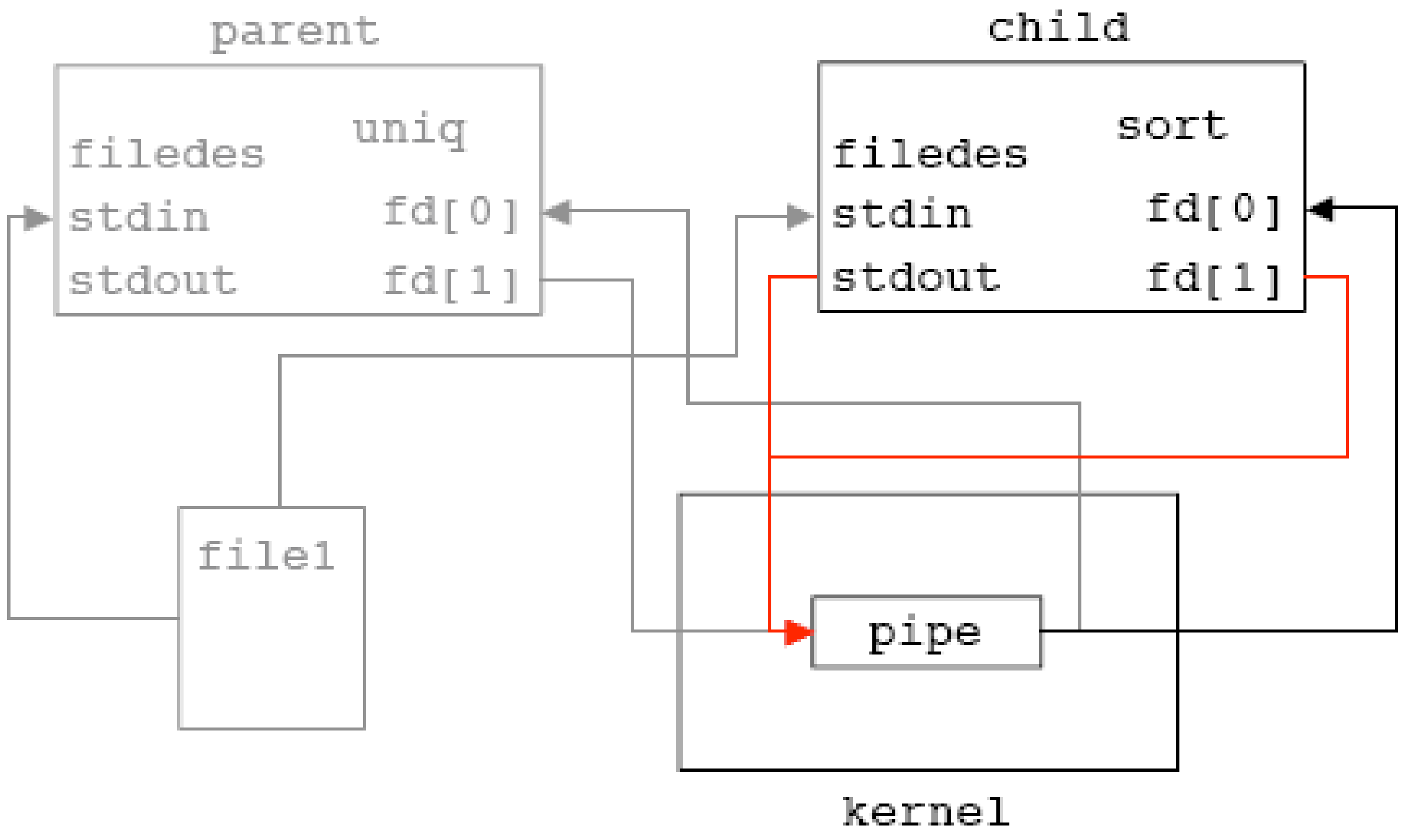
pipe(fd);



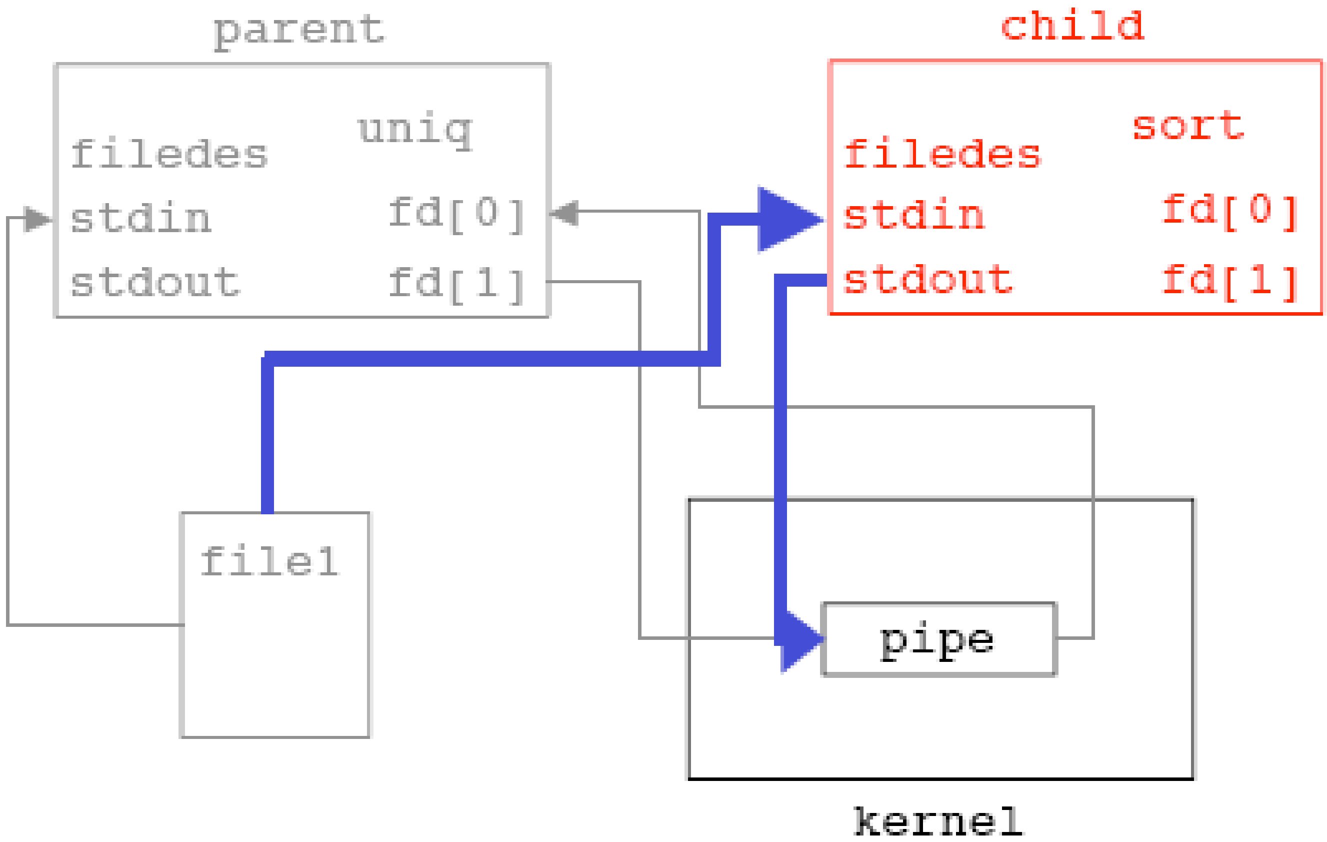
fork();


parent
filedes
uniq
stdin
fd[0]
stdout
fd[1]
child
filedes
sort
stdin
fd[0]
stdout
fd[1]
file1
pipe
kernel

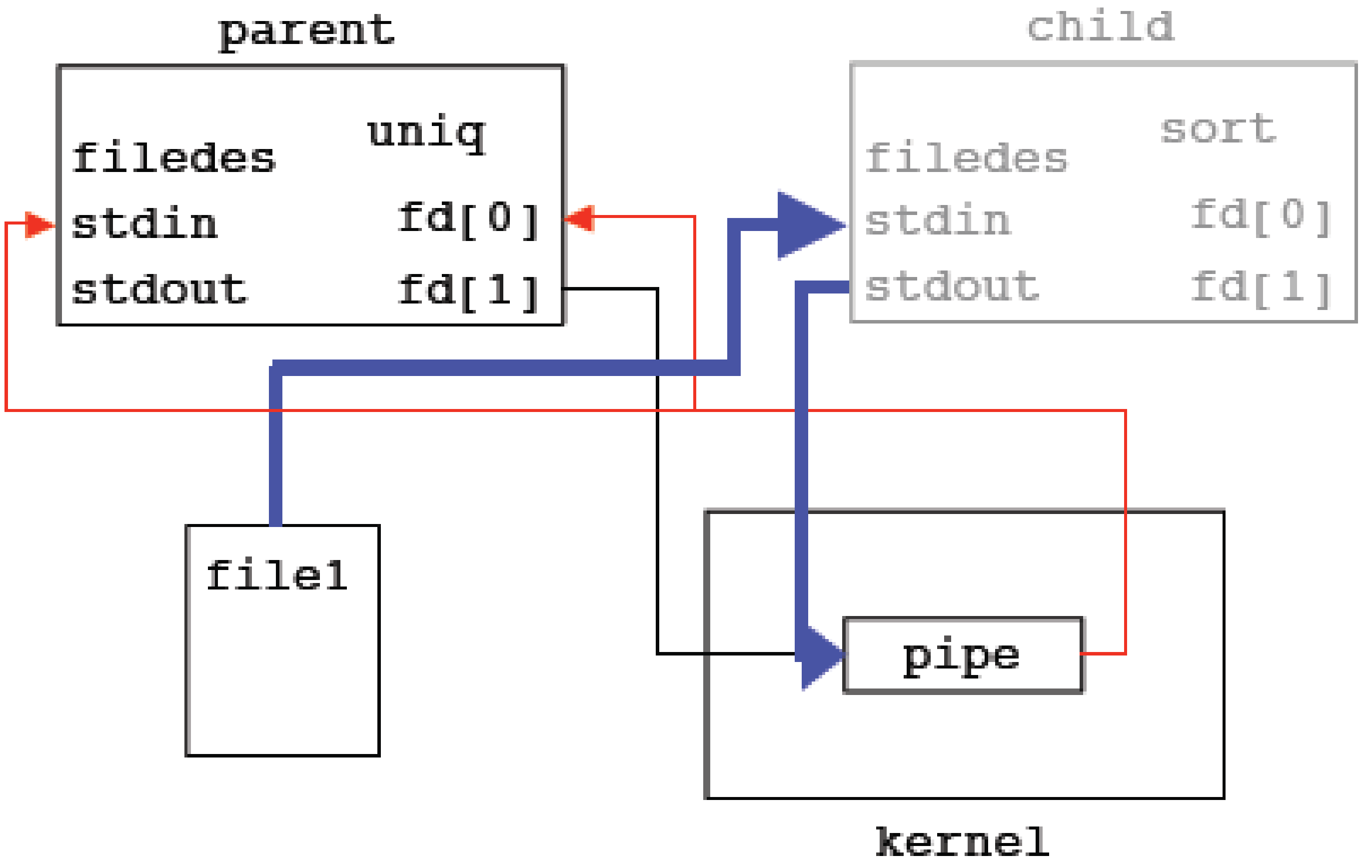
```
dup2(fd[1], fileno(stdout));
```



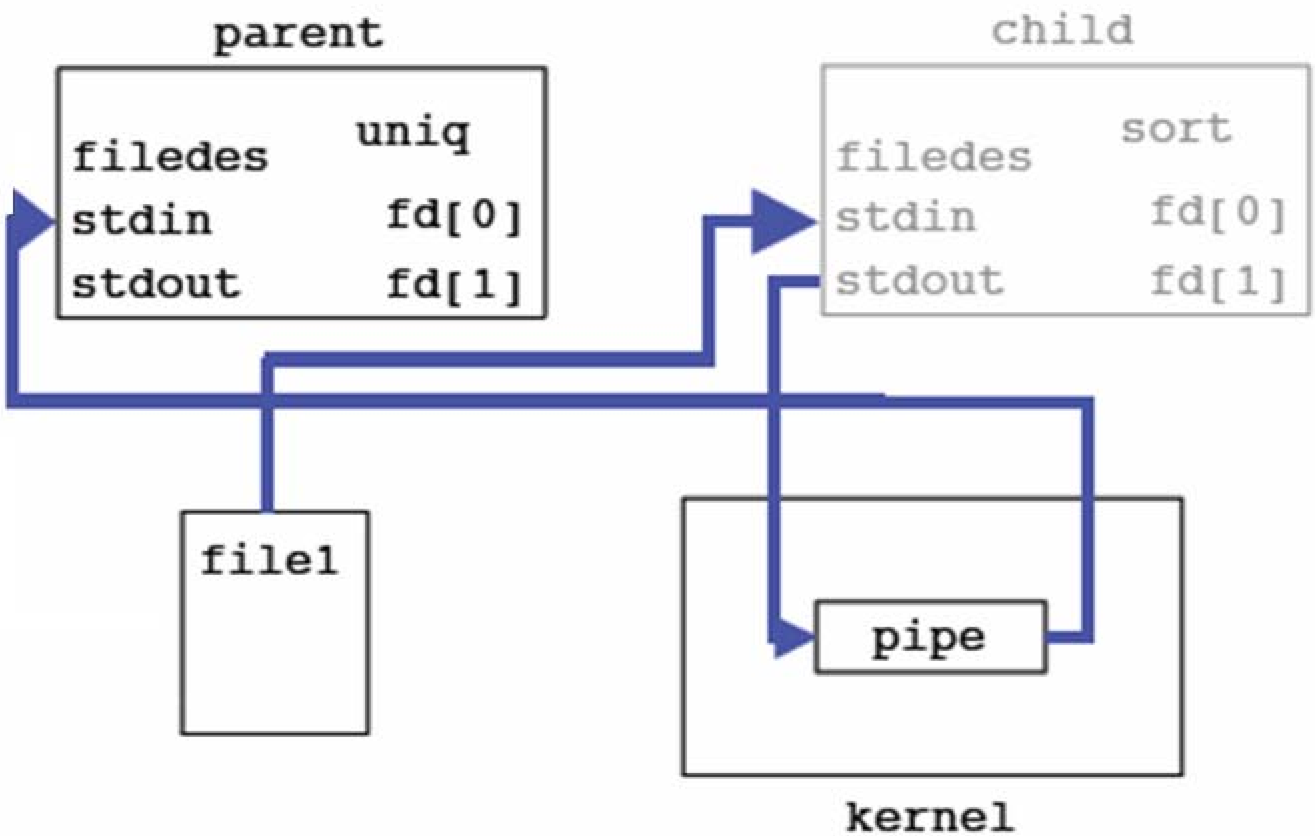
```
close(fd[0]); close(fd[1]);
```



dup2(fd[0], fileno(stdin));



`close(fd[1]); close(fd[0]);`



Reading and writing to a pipe

- A read on an empty pipe will block until there is something to read.
- A write on a full pipe will block until there is more space. (Pipes have a finite size.)
- Writing to a pipe that has been closed by the other end will result in a SIGPIPE or “Broken Pipe” message.
- Read will return 0 if the write end of the pipe is closed.