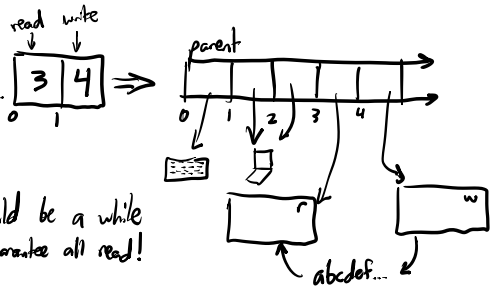
today: syscall to let processes pass messages back & forth

pipe

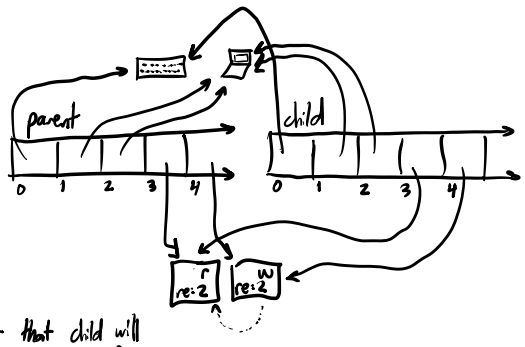
creates 2 file descriptors - 1 read 1 write

↳ writes to some location maintained by OS — read fd can read from it

```
int main(→) {
    int fds[2];
    pipe(fds); // finds 2 unused descriptors, e.g.
close(fds[1]); → write(fds[1], "hello", 6); // includes '\0'
    char buffer[6];
close(fds[0]); → read(fds[0], buffer, 6); // really should be a while
    printf("%s\n", buffer);                  loop to guarantee all read!
    return 0;
}
```



now across process boundaries!




```
int main(→) {
    int fds[2];
    pipe(fds);
    pid_t pid = fork();
    if (pid==0) {
        close(fds[1]); // visibly clear that child will
        char buffer[6];   not be writing - drops refcount to 1
        read(fds[0], buffer, sizeof(buffer)); // also blocking syscall! will wait until at least 1 byte available
        close(fds[0]);
        printf("%s\n", buffer);
        return 0;
    }
    close(fds[0]);
    write(fds[1], "hello", 6);
    close(fds[1]);
    waitpid(pid, NULL, 0); // will block if child not finished yet
    return 0;
}
```

shell pipe directive: $echo -e "peach\npear\napple" | sort | grep ea

↑ OS creates pipe bridging 2 peer processes (see picture in slides)

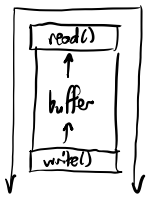
pipe created before child created w/ fork()

file descriptors 0,1,2 are closed by OS when process exits

don't have to close these

w/ pipe:

V-node table: (tries to make pipe look like regular file)



dup2: syscall gets descriptor to reference same resource as another descriptor

int dup2(int source, int target);

e.g. dup2(fds[0], STDIN_FILENO);
close(fds[0]);
//rewires fd 0 to pull bytes from read end of pipe instead of keyboard
↳ parse "standard input" to a process much more quickly