

A decorative graphic on the left side of the slide, consisting of a dark brown wavy line with a yellow-orange line running parallel to it.

FORK – EXEMPEL

```
static int g = 10;
int main(void) {
    int v = 10;
    pid_t p;
    if ((p = fork()) < 0) {
        perror("fork error");
        return 1;
    } else if (p == 0) {
        v++;
        g++;
    } else {
        g--;
        v--;
        if (waitpid(p, NULL, 0) != p) {
            perror("waitpid error");
            return 1;
        }
    }
    printf("mypid = %d parentpid = %d p = %d v = %d g = %d\n",
           getpid(), getppid(), p, v, g);
    return 0;
}
```



PID: 100

g: 10

v: 10

p: ?



fork

PID: 100

g: 10

v: 10


p: 101

PID: 101

g: 10

v: 10

p: 0



The diagram illustrates a process state transition. It consists of two rectangular boxes. The top box has a green border and contains the text: PID: 100, g: 10, v: 10, p: 101. The bottom box has a yellow border and contains the text: PID: 101, g: 10, v: 10, p: 0. A black arrow points from the bottom of the green box to the top of the yellow box, indicating a transition from the state in the green box to the state in the yellow box.

PID: 100

g: 10

v: 10

p: 101

PID: 101

g: 10

v: 10

p: 0

g--

PID: 100
g: ~~10~~ 9
v: 10
p: 101

PID: 101
g: 10
v: 10
p: 0

v--

PID: 100
g: ~~10~~ 9
v: ~~10~~ 9
p: 101

PID: 101
g: 10
v: 10
p: 0

wait

PID: 100

g: ~~10~~ 9

v: ~~10~~ 9

p: 101

PID: 101

g: 10

v: 10

p: 0

wait

```
PID: 100  
g: 10 9  
v: 10 9  
p: 101
```

v++

```
PID: 101  
g: 10  
v: 10 11  
p: 0
```

wait

```
PID: 100  
g: 10 9  
v: 10 9  
p: 101
```

g++

```
PID: 101  
g: 10 11  
v: 10 11  
p: 0
```

wait

```
PID: 100  
g: 10 9  
v: 10 9  
p: 101
```

printf

```
PID: 101  
g: 10 11  
v: 10 11  
p: 0
```

mypid = 101 parentpid = 100 p = 0 v = 11 g = 11

wait

```
PID: 100  
g: 10 9  
v: 10 9  
p: 101
```

exit

```
mypid = 101   parentpid = 100   p =    0   v = 11   g = 11
```

printf

PID: 100

g: ~~10~~ 9

v: ~~10~~ 9

p: 101

mypid = 101 parentpid = 100 p = 0 v = 11 g = 11

mypid = 100 parentpid = 63 p = 101 v = 9 g = 9

exit

```
mypid = 101   parentpid = 100   p =    0   v = 11   g = 11  
mypid = 100   parentpid =   63   p = 101   v =  9   g =  9
```

PIPE

- Ett sätt kommunicera mellan olika processer är att använda **pipes**.
- Det finns flera sätt att kommunicera på, men pipes är den äldsta metoden och finns implementerat på alla system.
- Ett sätt att se på en pipe är just som ett enkelriktat rör, man skickar in på ena sidan och plockar ut från den andra. För att skapa en pipe så använder man funktionen

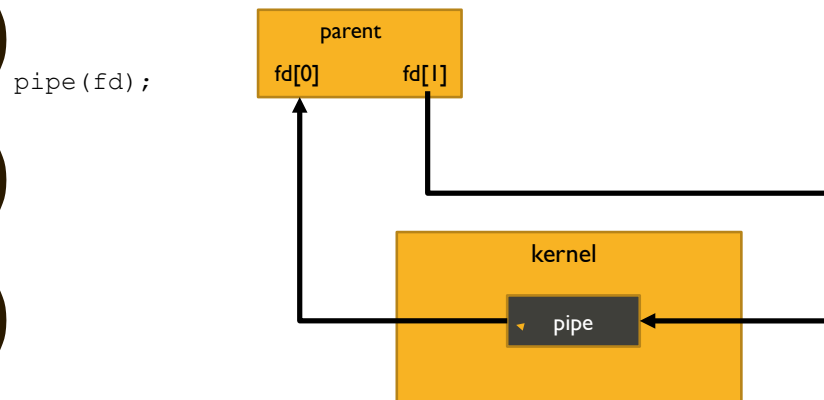
```
int pipe( int fildes[2] )
```

som returnerar två stycken 'file descriptors'.
- Den file descriptor som finns på index 0 är "läsändan" och den på 1 är "skrivändan".

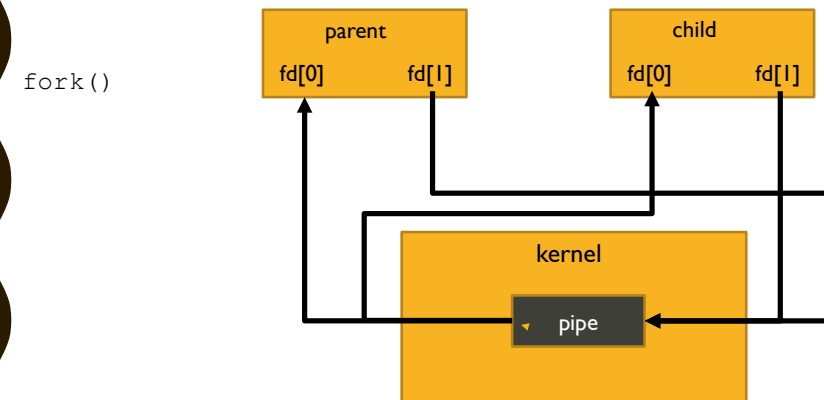
SKAPA EN PIPE

- Om jag vill skriva att ett program som ska skapa en barnprocess och sätta upp en pipe till den så att föräldern kan skicka information till barnet. Då kan man göra på följande sätt:
 1. Anropa `pipe` för att skapa själva pipen
 2. Gör en `fork` för att få barnprocessen. Kom ihåg att barnet får en kopia av förälderns variabler etc, dvs. barnet kommer att ha tillgång till de file descriptorer som pipen har.
 3. Nu bör barnet stänga sin "skrivända"
 4. Och föräldern bör stänga sin "läsända"

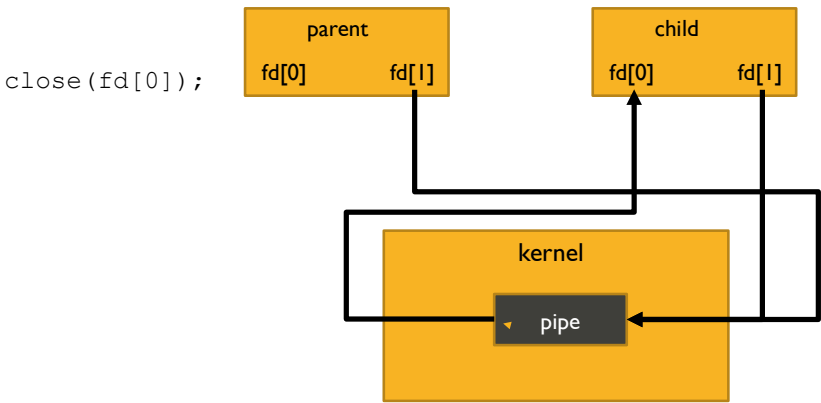
PIPE - ANVÄNDNING



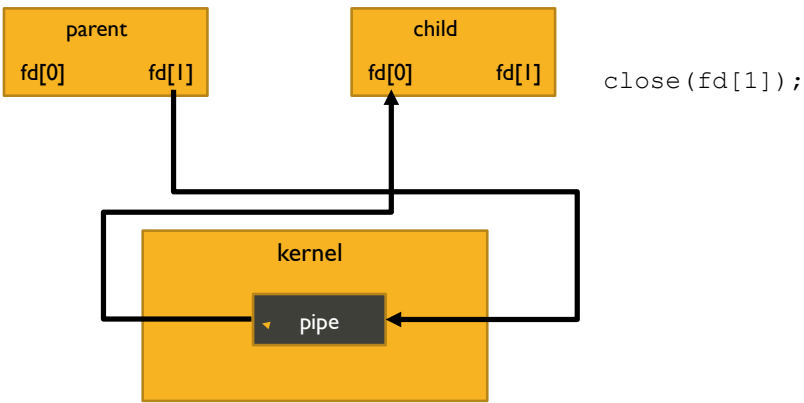
PIPE - ANVÄNDNING



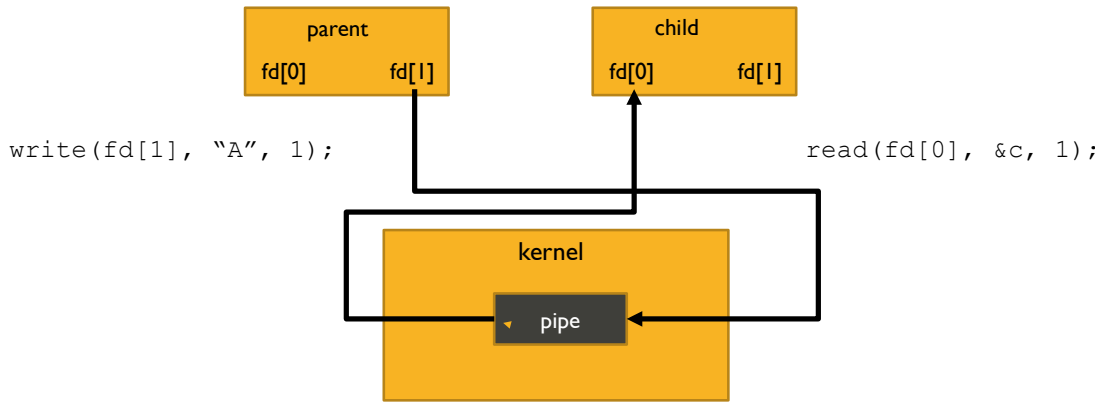
PIPE - ANVÄNDNING



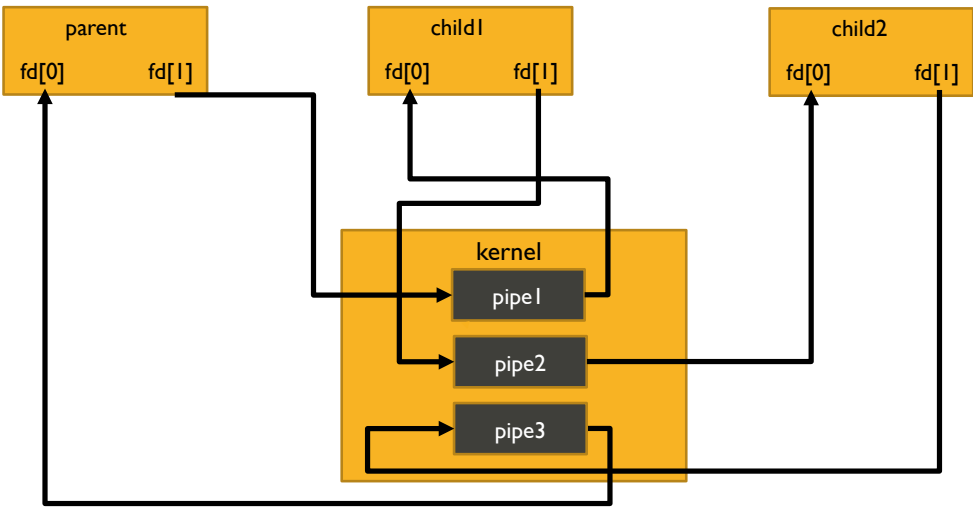
PIPE - ANVÄNDNING



PIPE - ANVÄNDNING

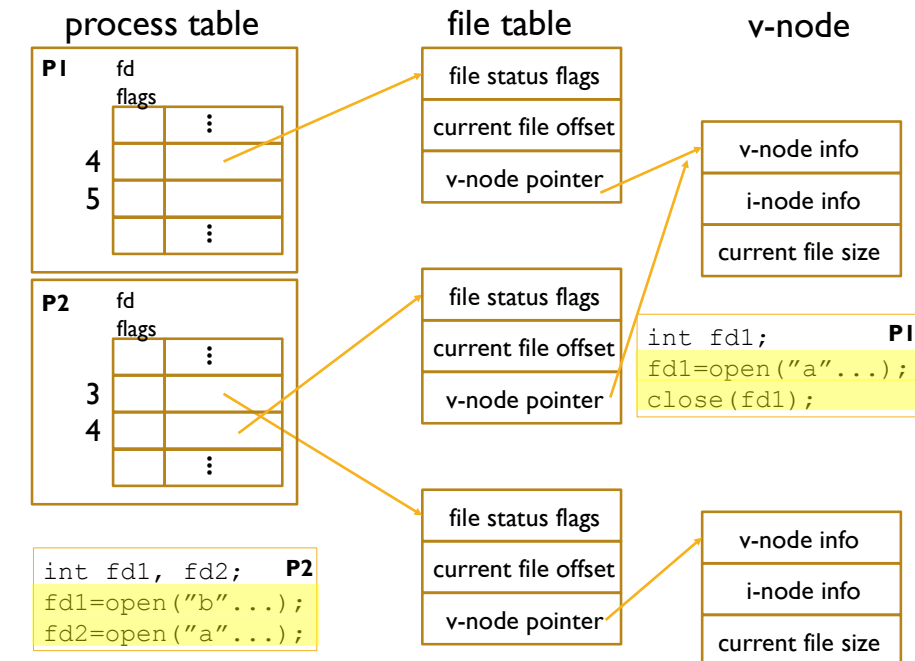
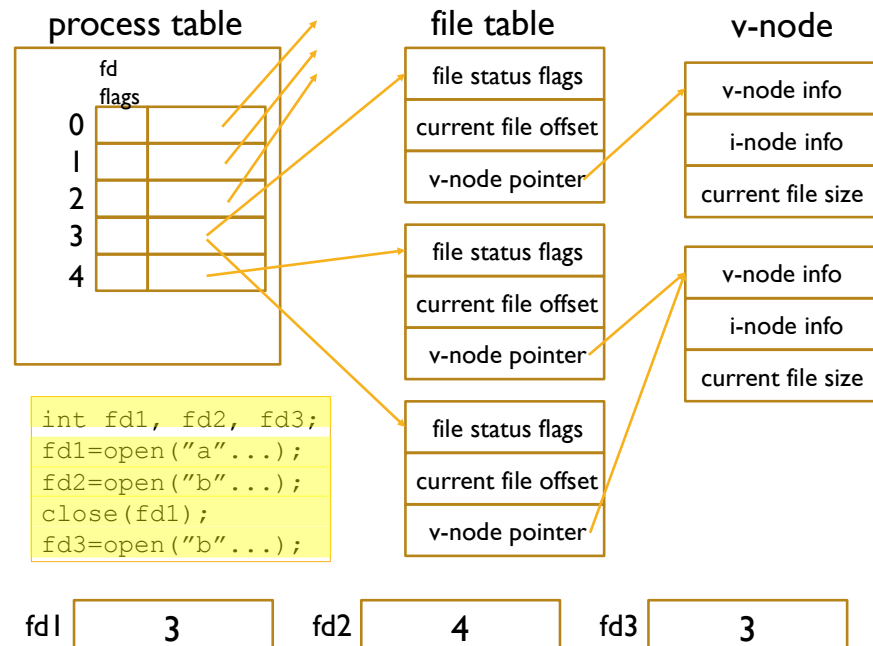
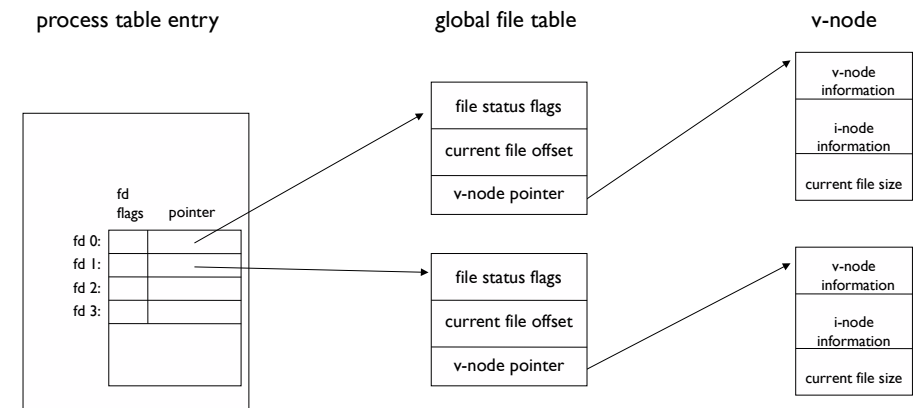


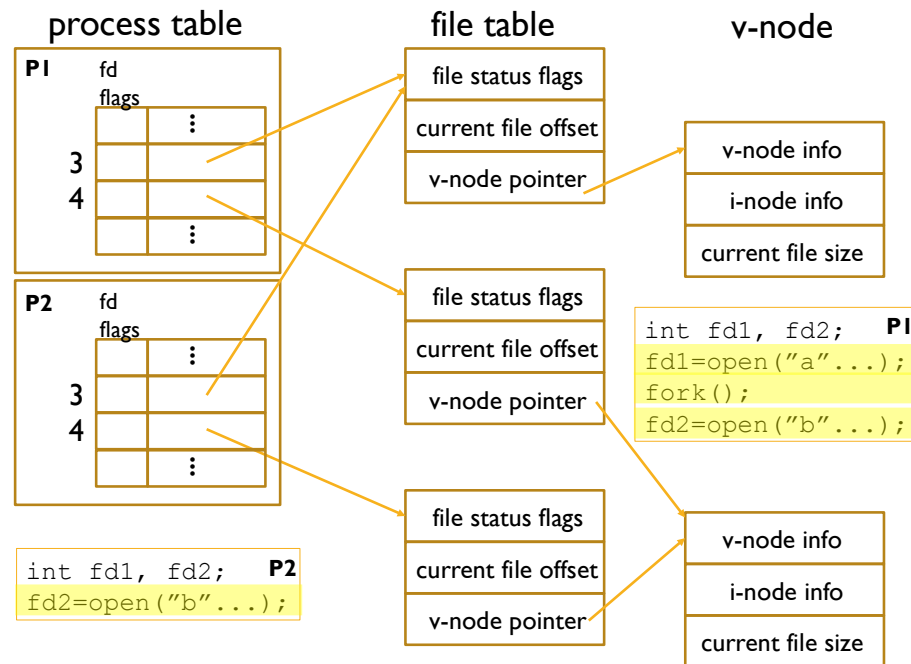
PIPE – EXTRA EXEMPEL



DELA FILER

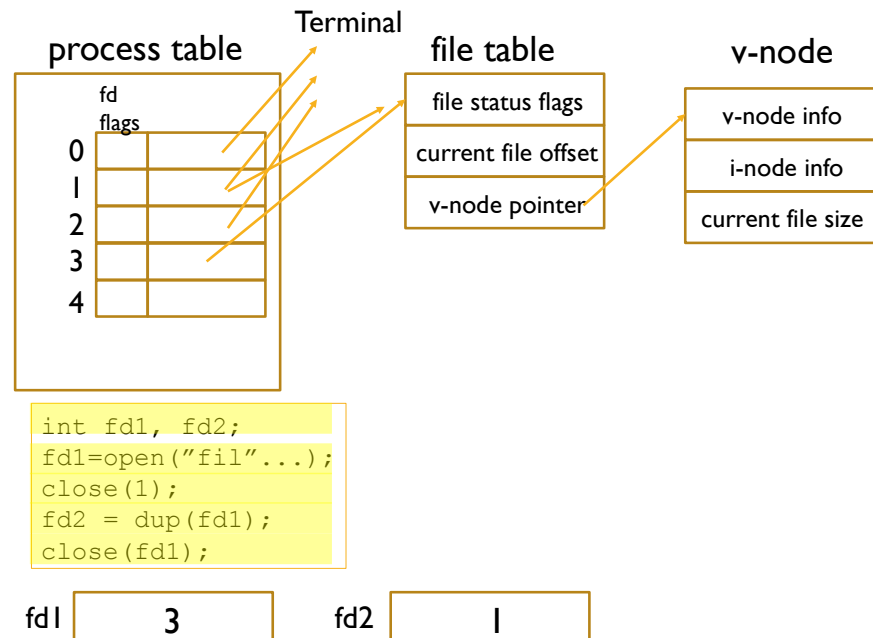
- En sak som kanske är lite oväntat är att flera processer kan dela på samma fil. Detta kan ju naturligtvis ge upphov till lite problem ...
- Process table
 - Varje process har en post i tabellen med en processbeskrivning som innehåller information om själva processen. Bland annat så finns alla "file descriptors" som processen använder där.
- File table
 - Kärnan har en tabell med alla öppna filer. Innehåller bla status flaggorna för filen (read, write etc), nuvarande offset för filen och en pekare till filens v-node
- V-node
 - Varje öppen fil har en v-node. Innehåller bl.a. info om typen av fil och pekare till funktioner som används på filen.



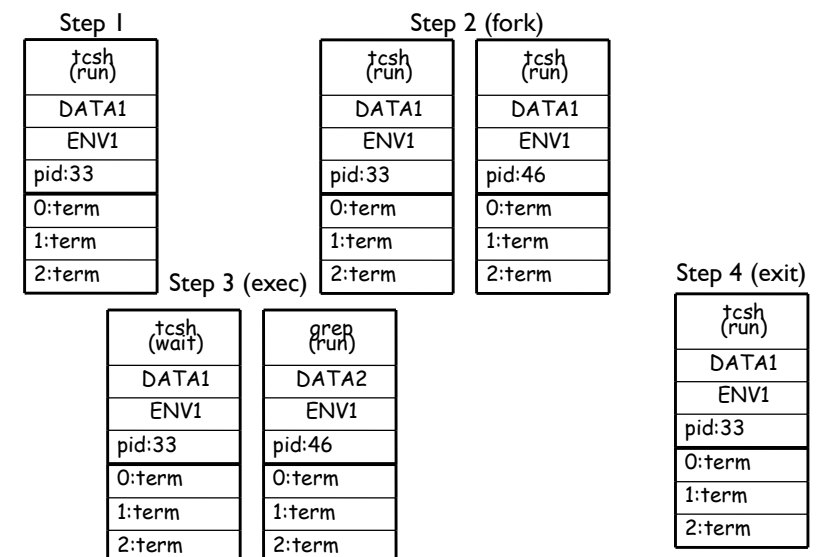


DUP & DUP2

- `int dup(int oldd);`
- `int dup2(int oldd, int newd);`
- Används för att duplicera en fildeskriptor
- Med `dup2` kan man välja vilket nummer den nya fildeskriptorn ska ha. Används denna redan kommer den först att stängas (som om `close` anropats)



grep maxprop *.c



tr "a-z" "[A-Z]" < srcfile > destfile

Step 1

tcsh (run)
DATA1
ENV1
pid:33
0:term
1:term
2:term

Step 2 (fork)

tcsh (wait)
DATA1
ENV1
pid:33
0:term
1:term
2:term

tcsh (run)
DATA1
ENV1
pid:46
0:term
1:term
2:term
3:srcfile
4:destfile

Step 3 (dup, close)

tcsh (wait)
DATA1
ENV1
pid:33
0:term
1:term
2:term

tcsh (run)
DATA1
ENV1
pid:46
0:srcfile
1:destfile
2:term

tr "a-z" "[A-Z]" < srcfile > destfile

Step 4 (exec)

tcsh (wait)
DATA1
ENV1
pid:33
0:term
1:term
2:term

tr (run)
DATA2
ENV1
pid:46
0:srcfile
1:destfile
2:term

Step 5 (exit)

tcsh (run)
DATA1
ENV1
pid:33
0:term
1:term
2:term

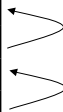
who | grep jacob | wc -l

Step 1

tcsh (run)
DATA1
ENV1
pid:33
0:term
1:term
2:term

Step 2 (pipe)

tcsh (run)
DATA1
ENV1
pid:33
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write



who | grep jacob | wc -l

Step 3 (fork)

tcsh (run)
DATA1
ENV1
pid:33
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write

tcsh (run)
DATA1
ENV1
pid:46
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write

tcsh (run)
DATA1
ENV1
pid:47
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write

tcsh (run)
DATA1
ENV1
pid:45
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write

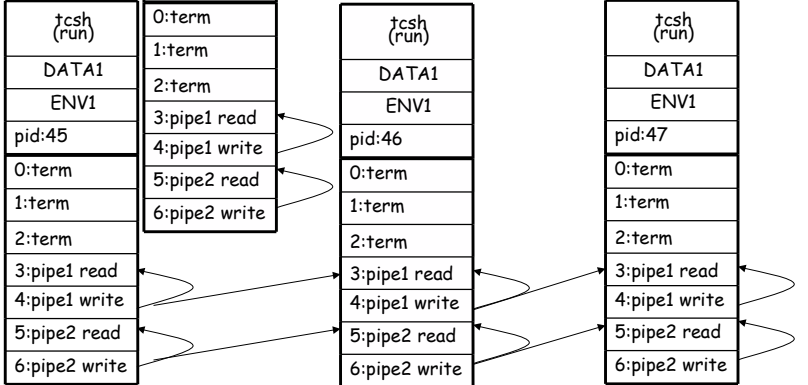
tcsh (run)
DATA1
ENV1
pid:46
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write

tcsh (run)
DATA1
ENV1
pid:47
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write

tcsh (run)
DATA1
ENV1
pid:48
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write

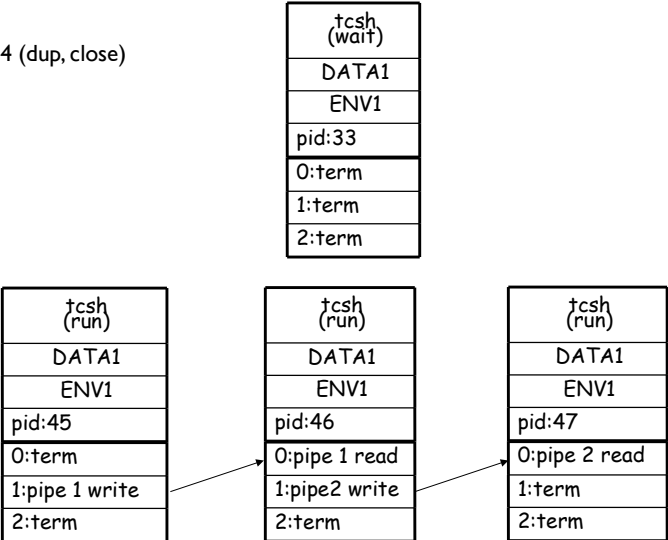
tcsh (run)
DATA1
ENV1
pid:49
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write

tcsh (run)
DATA1
ENV1
pid:50
0:term
1:term
2:term
3:pipe1 read
4:pipe1 write
5:pipe2 read
6:pipe2 write



who | grep jacob | wc -l

Step 4 (dup, close)



who | grep jacob | wc -l

Step 5 (exec)

