

7. Pipes

Pipes have two main benefits:

- they control same inter-process communication
- they are transparent (the program doesn't know they exist)

How are they implemented?

```
int pipe(int pipefd[2]);  
pipefd[0] = "read"  
pipefd[1] = "write"  
// these can be thought of like 0 = stdin and 1 = stdout
```

This builds a circular bounded buffer that can be used to communicate between processes

BUT that introduces the case of a full buffer; what do we do?

- a. write what you can and return the size
- b. wait
- c. throw error

Pipes therefore have a third benefit—flow control!

```
$ du | sort -n
```

How do we use pipes?

```
int pipefd[2];  
pid_t cpid;  
char buf;  
if (argc != 2) exit(EXIT_FAILURE);  
if (pipe(pipefd) == -1) {perror("pipe"); exit(1);}  
if ((cpid = fork()) == 0) {  
    close(pipefd[1];  
    while(read(pipefd[0], &buf, 1) > 0) write(STDOUT_FILENO, &buf, 1);  
    write(STDOUT_FILENO, "\n", 1);  
    close(pipefd[0];
```

```

    _exit(0);
}
else { // parent
    close(pipefd[0]);
    write(pipefd[1], argv[1], strlen(argv[1]));
    close(pipefd[1]);
    wait(NULL);
    exit(0);
}
// note: the order of execution is not guaranteed

```

Pipes, as mentioned above, are not perfect; we have 4 major categories of pipe failure:

1. Reading from an empty pipe where the writer never writes
 - a. hang indefinitely
2. Reading from an empty pipe with no writers
 - a. return 0
3. Writing to a full pipe where the reader never reads
 - a. suspend/hang indefinitely
4. Writing to a full pipe where the reader never reads
 - a. SIGPIPE since returning 0 would probably cause many people to retry
 - b. if SIGPIPE is ignored, write fails with EPIPE

Note that the big mistakes and performance errors occur when we do not close pipes

- a named pipe can cause an fd leak, since it could be referenced by any program

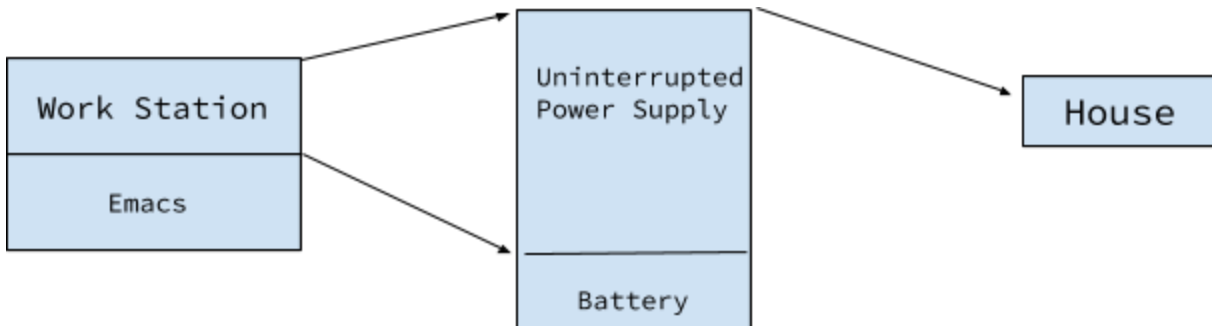
A common issue:

```

while (condition) printf("%d\n", number);
// since this would run forever on write failure,
// we have an asynchronous signal SIGPIPE
// which kills a program if no readers remain

```

BUT sometimes there is a reader, but they're not listening (plus closing has a delay)
Take the setup:



What do we do if the power in the house goes out?

1. do nothing and lose work
2. OS saves a stack of running processes and state in the stack
3. End-to-End (processes are notified & can figure it out for themselves)

Since #2 can make clock-based apps go haywire, we consider #3;

We have 3 ways of notifying processes that we have lost power

- i. poll /dev/power
 - this is annoying, since the burden is on the programmer
- ii. /dev/powering_off
 - the kernel blocks this unless we have lost power
 - this is too complicated
- iii. SIGNALS
 - grab the attention of a program not designed to receive it