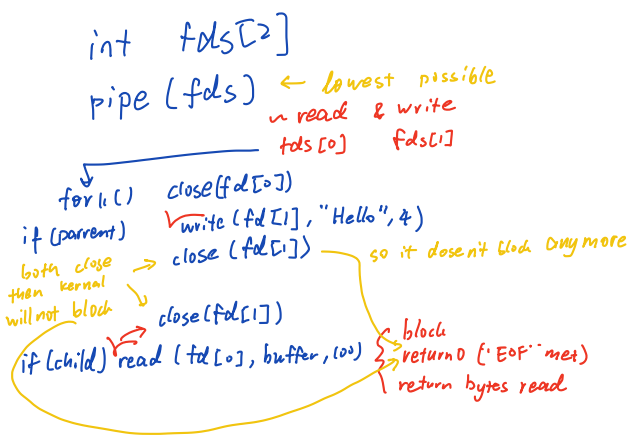
CS341 #22 — Pipes and file seeks

1 How do you use unnamed pipe to send a message from the parent to the child?



writing to a pipe that has no "reader" will get a SIGPIPE, which kills the process

Don't forget to fflush after using C lib function :

```
FILE f = fdopen(fd[1])
fprintf(f, ...)
fflush(f) ☆ → to immediately get result
```

read & write sync by default

these data are send to the kernal

---

2. What is fseek and ftell? How would you use them?

```
f = fopen("data", "f")
fseek(f, SEEK_CUR, -10) → go back 10 bytes
        SEEK_SET , 42
        SEEK_END, -10
fseek(f, SEEK_END, 0)
long posn = ftell(f)
```

↑ design failure long → 2GiB file

→ tells us how big f is

```
char* content = malloc(posn+1)
fseek(f, SEEK_SET, 0) → make sure future calls are good
        ||
      rewind
```

3. What happens to the other process if you fclose after forking?

fork → fclose → nothing change

4. What happens to the other process if you fseek before forking?

fseek → fork → both start from SEEK_CUR

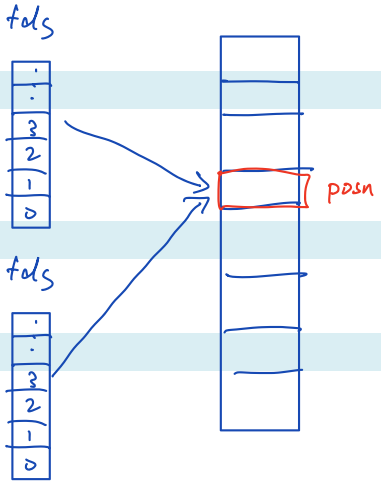
5. What happens to the other process if you fseek after forking?

fork → fseek → will affected

6. Why does pwrite exist? When would you use it?

```
pwrite(fd, void*, size_t, offset_t offset)
```

multi process can write at the same time



mkfifo ↓ pipe ↙

7. What is an named pipe and an unnamed pipe?

8. What signals can a pipe generate and when?

SIGPIPE

9. How would you modify your pipe code to send an integer value of a variable?

dprintf(fd[1], "%d", 42)

10. Why is it necessary to close the pipe's unused filedescriptors after forking?

AUTO GRADER ?

11. How would you fix/improve this code?

```
pthread_mutex_t m;
pthread_cond_t cv;
int in, out, count;
void* buffer[16]

void enqueue(void* ptr) {
  p_m_lock(&m);
  while(count < 16) {}
  pthread_mutex_unlock(&m);
  p_cond_broadcast(&cv);
  count ++;
  buffer[ (in++) % 16 ]  = ptr;
}
```

```
void* dequeue() {
  p_m_lock(&m);
  while(count  == 0) {}
  void* result = buffer[ (out++) % 16 ];
  p_cond_broadcast(&cv);
  pthread_mutex_unlock(&m);
  count --;
  return result;
}
```

```
void pipe_or_quit(int*result) {
  if( 0 == pipe(result) ) return; else quit("pipe");
}

void create_pipes(int* array6) {
  pipe_or_quit(array6);
  pipe_or_quit(array6 +2);
  pipe_or_quit(array6 +4);
}

void exec_or_quit(const char *program, const
char **args, int old_err_fd) {
  execv(program, (char*const*) args);
  dup2(old_err_fd, 2);
  quit("execv");
}
```

create 3 pipes

```
int run(const char *test, const char *prog, const char
**args, const char *input, char **output,
        char **erroroutput, int *waitresult) {
  if (test) printf("%s: Running %s\n", test, prog);
  int pipes[6];
  create_pipes(pipes);
  pid_t childid = fork_or_quit();
  if(childid ==0 ) {
//Child should close 'in'(input), out(output) err(output)
    // close unused end of pipes
    close(pipes[1]); close(pipes[2]);close(pipes[4]);
    int old_err_fd = dup(2);
    dup2_or_quit(pipes[0] /*read from */,0);
    dup2_or_quit(pipes[3] /*write to*/,  1);
    dup2_or_quit(pipes[5] /*write to*/  ,2);
    alarm(ALARM_TIMEOUT_SECONDS);
    exec_or_quit(prog, args, old_err_fd);
  }
```