

20 a) O que este código faz? É possível prever qual será a saída do programa?

O processo-pai cria 2 filhos, que farão coisas análogas (ler do ~~buffer~~ ^{pipe} e então escrever no pipe ("tomato" no filho 1 e "turnip" no 2)), ele então escreve no pipe, printa uma mensagem em 'stdout', espera um pouco, lê do pipe e escreve isso em 'stdout' junto de outra mensagem, e mata seus filhos. Não dá para prever a saída do programa (o que o pai lerá do pipe) pois isso depende de qual foi o último filho a ler-e-escrever no pipe, e ambos estão em loop infinito sem ~~estados~~ condições de controle.

b) O que acontece se omitirmos o comando **write** do pai?

Os três processos entrarão em deadlock, pois todos tentarão ler de uma pipe vazia e ficarão travados nesse ponto, pois não há ninguém para escrever na pipe.

c) O que acontece se omitirmos o comando **write** de um dos filhos?

~~Quando o processo pai chamar o 'write' de um dos filhos seja removido, podemos garantir a saída do programa, pois só haverá uma coisa sendo escrita na pipe (isso porém, apenas se o processo sem write não for chamado entre o pai e o filho com write)~~

3) (3,0) Dado o problema dos Filósofos Glutões, complete o código a seguir com chamadas a semáforos. [RESPONDER NA FOLHA DE PROVA]

Dados Compartilhados

```
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N

#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[N];

typedef int semaphore;

semaphore mutex = 1;
semaphore philo[N] = {0, 0, ..., 0};
```

```
void philosopher(int i)
{ while (TRUE) {
    think();
    take_forks(i);
    eat();
    put_forks(i); }}
```

```
void take_forks(int i)
{ down(&mutex);
  state[i] = HUNGRY;
  test(i);
  down(&philo[i]);
  up(&mutex); }
```

```
void test(i)
{ if (state[i] == HUNGRY &&
    state[LEFT] != EATING &&
    state[RIGHT] != EATING) {
    state[i] = EATING;
    up(&philo[i]);
  } }
```

```
void put_forks(i)
{ down(&mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  up(&mutex); }
```

