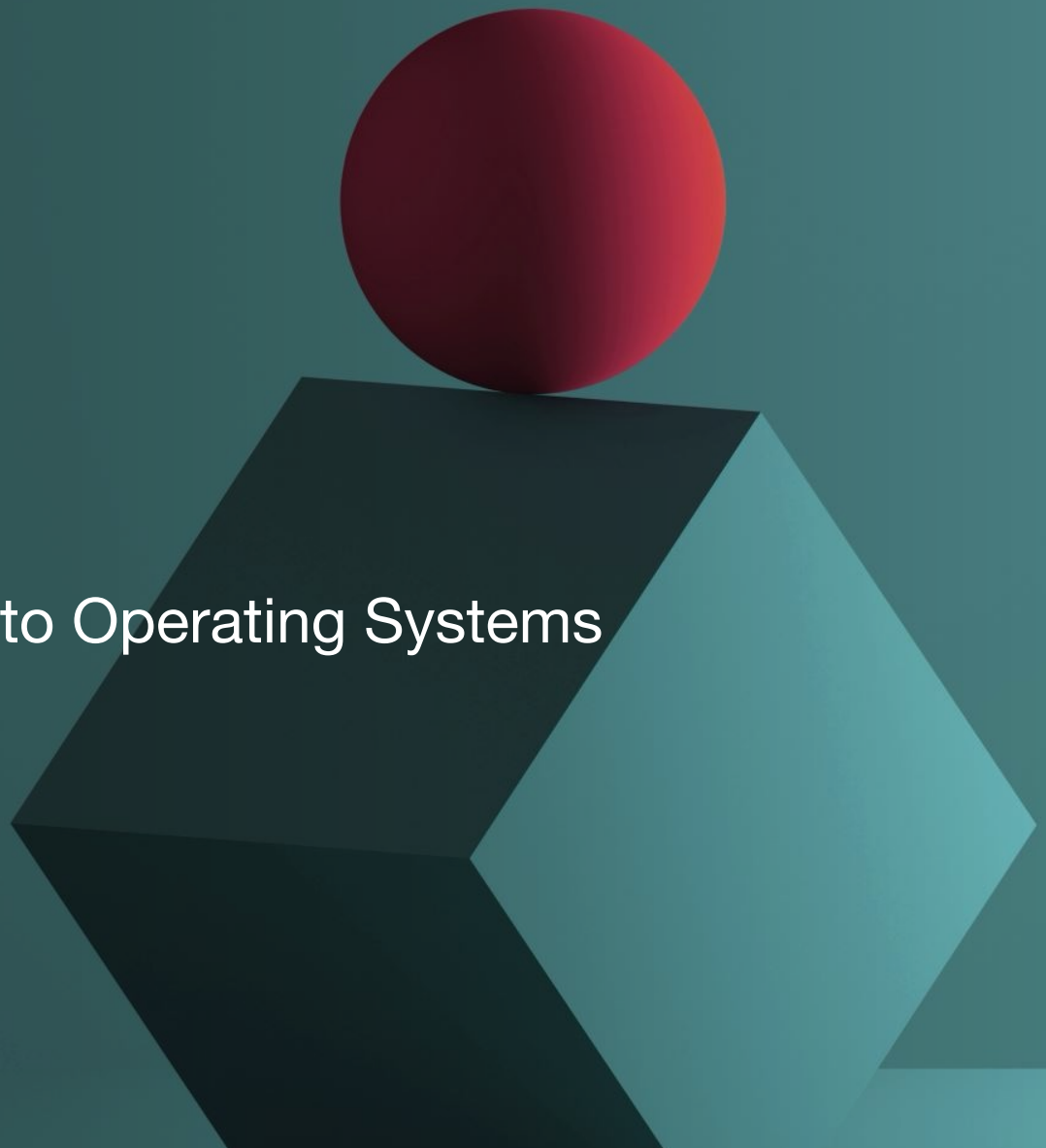


Tutorial 5

CS2106: Introduction to Operating Systems

Question 1

Question 1: Race Conditions

Consider the following two tasks, *A* and *B*, to be run concurrently and use a shared variable `x`. Assume that:

- load and store of *x* is atomic
- `x` is initialized to 0
- `x` must be loaded into a register before further computations can take place.

Task A	Task B
`x++;` `x++;`	`x = 2 * x`

Question 1: Race Conditions

Task A	Task B
`x++;` `x++;`	`x = 2 * x`

i. How many relevant interleaving scenarios are possible when the two threads are executed?

ii. What are all possible values of `x` after both tasks have terminated? Use a step-by-step execution sequence of the above tasks to show all possible results.

Question 1: Race Conditions

Task A	Task B
`x++;` `x++;`	`x = 2 * x`

Step 1: Break down the High-Level Language into Machine Instructions

	Task A	Task B
1	`lw $r1, 0(addr(x))`	`lw $r1, 0(addr(x))`
2	`addi $r1, $r1, 1`	`sll $r1, $r1, 1`
3	`sw $r1, 0(addr(x))`	`sw $r1, 0(addr(x))`
4	`lw $r1, 0(addr(x))`	
5	`addi $r1, $r1, 1`	
6	`sw $r1, 0(addr(x))`	

Question 1: Race Conditions

Task A	Task B
`x++;` `x++;`	`x = 2 * x`

Step 2: Identify which instructions, when interleaved can affect the value of x.

	Task A	Task B
1	`lw $r1, 0(addr(x))`	`lw $r1, 0(addr(x))`
2	`addi $r1, $r1, 1`	`sll $r1, $r1, 1`
3	`sw $r1, 0(addr(x))`	`sw $r1, 0(addr(x))`
4	`lw $r1, 0(addr(x))`	
5	`addi $r1, $r1, 1`	
6	`sw $r1, 0(addr(x))`	

The instructions that load and store the value of x into the register can affect the value of x when interleaved.

i.e. Lines 1, 3, 4 and 6

Question 1: Race Conditions

Task A	Task B
`x++;` `x++;`	`x = 2 * x`

We can simply the machine instructions into just loading and storing X for both tasks A and B for easier analysis of the interleaving scenarios later.

- When Store X happens for Task A, we are storing the value of x + 1 in the register back into memory.
- Similarly, when Store X happens for Task B, we are storing the value of 2 * x in the register back into memory

	Task A		Task B
A1	`Load x`	B1	`Load x`
A2	`Store x`	B2	`Store x`
A3	`Load x`		
A4	`Store x`		

Question 1: Race Conditions

Task A	Task B
`x++;` `x++;`	`x = 2 * x`

	Task A		Task B
A1	`Load x`	B1	`Load x`
A2	`Store x`	B2	`Store x`
A3	`Load x`		
A4	`Store x`		

- The order of the instructions A1 to A4 cannot change for Task A.
- Similarly, the order of the instructions B1, B2 cannot change for Task B.
- Let's see where we can insert B1 and B2 between A1 to A4 without affecting the respective order of instructions.
- Firstly, we can insert B1 into one of the **five slots** between A1 to A4. Note that B1 can be inserted before A1 and after A4.

Slot 1	A1	Slot 2	A2	Slot 3	A3	Slot 4	A4	Slot 5

Question 1: Race Conditions

Task A	Task B
`x++;` `x++;`	`x = 2 * x`

	Task A		Task B
A1	`Load x`	B1	`Load x`
A2	`Store x`	B2	`Store x`
A3	`Load x`		
A4	`Store x`		

- After inserting B1 into a slot, there are **six slots** where we can insert B2.
- However, there are only 5 + 4 + 3 + 2 + 1 = 15 where B2 can be inserted after B1.

B1	Slot 1	A1	Slot 2		A2	Slot 3		A3	Slot 4		A4	Slot 5	
		A1	B1	Slot 1	A2	Slot 2		A3	Slot 3		A4	Slot 4	
		A1			A2	B1	Slot 1	A3	Slot 2		A4	Slot 3	
		A1			A2			A3	B1	Slot 1	A4	Slot 2	
		A1			A2			A1			A4	B1	Slot 1

Question 1: Race Conditions

Task A	Task B
`x++;` `x++;`	`x = 2 * x`

	Task A		Task B
A1	`Load x`	B1	`Load x`
A2	`Store x`	B2	`Store x`
A3	`Load x`		
A4	`Store x`		

i. How many relevant interleaving scenarios are possible when the two threads are executed?

There are only 5 + 4 + 3 + 2 + 1 = 15 scenarios where B2 can be inserted after B1, hence there are 15 relevant interleaving scenarios.

B1	Slot 1	A1	Slot 2		A2	Slot 3		A3	Slot 4		A4	Slot 5					
		A1	B1	Slot 1	A2	Slot 2		A3	Slot 3		A4	Slot 4					
		A1			A2	B1	Slot 1	A3	Slot 2		A4	Slot 3					
		A1			A2			A3	B1	Slot 1	A4	Slot 2					
		A1			A2			A1			A4	B1	Slot 1				

Question 1: Race Conditions

Task A	Task B
`x++;` `x++;`	`x = 2 * x`

	Task A		Task B
A1	Load x	B1	Load x
A2	Store x	B2	Store x
A3	Load x		
A4	Store x		

ii. What are all possible values of **x** after both tasks have terminated? Use a step-by-step execution sequence of the above tasks to show all possible results.

Execution Sequence	Value
B1, B2, A1, A2, A3, A4	2
B1, A1, B2, A2, A3, A4	2
B1, A1, A2, B2, A3, A4	1
B1, A1, A2, A3, B2, A4	2
B1, A1, A2, A3, A4, B2	0
A1, B1, B2, A2, A3, A4	2
A1, B1, A2, B2, A3, A4	1
A1, B1, A2, A3, B2, A4	2

Execution Sequence	Value
A1, B1, A2, A3, A4, B2	0
A1, A2, B1, B2, A3, A4	3
A1, A2, B1, A3, B2, A4	2
A1, A2, B1, A3, A4, B2	2
A1, A2, A3, B1, B2, A4	2
A1, A2, A3, B1, A4, B2	2
A1, A2, A3, A4, B1, B2	4

Answer: Possible Results: 0, 1, 2, 3 and 4

Question 2

Question 2: Critical Section

- Can disabling interrupts avoid race conditions?
- If yes, would disabling interrupts be a good way of avoiding race conditions? Explain.

Question 2: Critical Section

Can disabling interrupts avoid race conditions?

- What happens when interrupts are disabled?
  - Timer Interrupts cannot happen.
  - Scheduling algorithms that uses time quantum will not be able to switch processes when a process has used up its time slice.
  - In theory, the interleaving between processes will not happen.
- So, if we disable interrupts when a process enters a critical section and then reenable interrupt after the process exits the critical section, it is similar to a process acquiring a universal lock and releasing it once exiting the critical section.

Refer to Lecture 6 Slide 18

Question 2: Critical Section

If yes, would disabling interrupts be a good way of avoiding race conditions? Explain.

- Disabling interrupts is not a good way to avoid race conditions.
- Reasons
  - Preemptive scheduling algorithms will not work properly, resulting in the operating system not being able to switch tasks and perform other things.
  - Many important wakeup signals are provided by interrupt service routines and these would be missed by the running process. A process can easily block on a semaphore and stay blocked indefinitely, because there is nobody to send a wakeup signal.
  - Not feasible as user code does not have privileges to disable interrupts.
  - Race conditions can still happen in a multi-core environment as another process can enter the critical section while running on a different core.

Question 3

Question 3

Multi-core platform X does not support semaphores or mutexes. However, platform X supports the following atomic function:

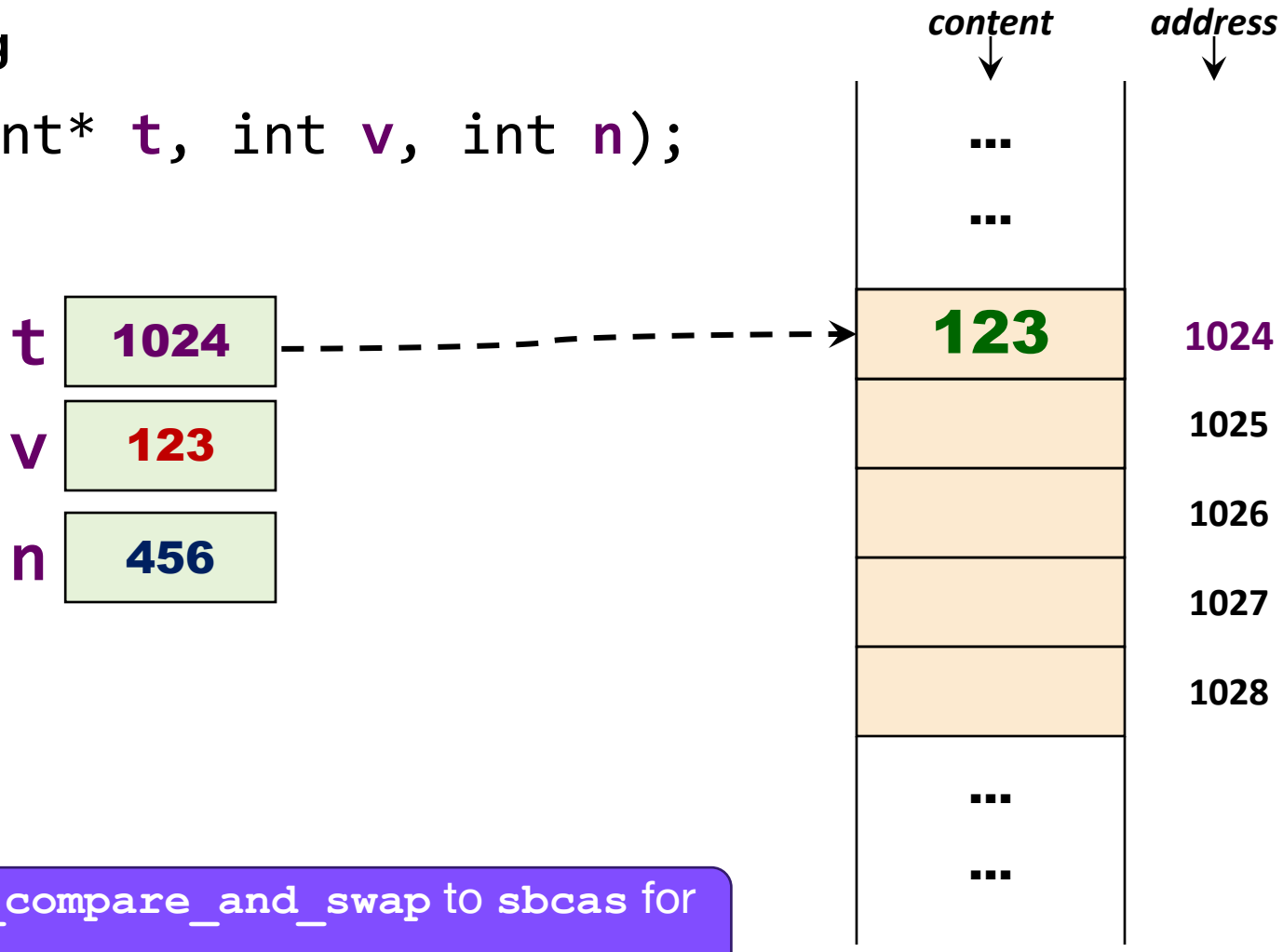
```
bool _sync_bool_compare_and_swap (int* t, int v, int n);
```

The above function atomically compares the value at location pointed by **`t`** with value **`v`**. If equal, the function will replace the content of the location with a new value **`n`**, and return 1 (true), otherwise return 0 (false).

Question 3

Before Swapping

```
bool sbcas(int* t, int v, int n);
```

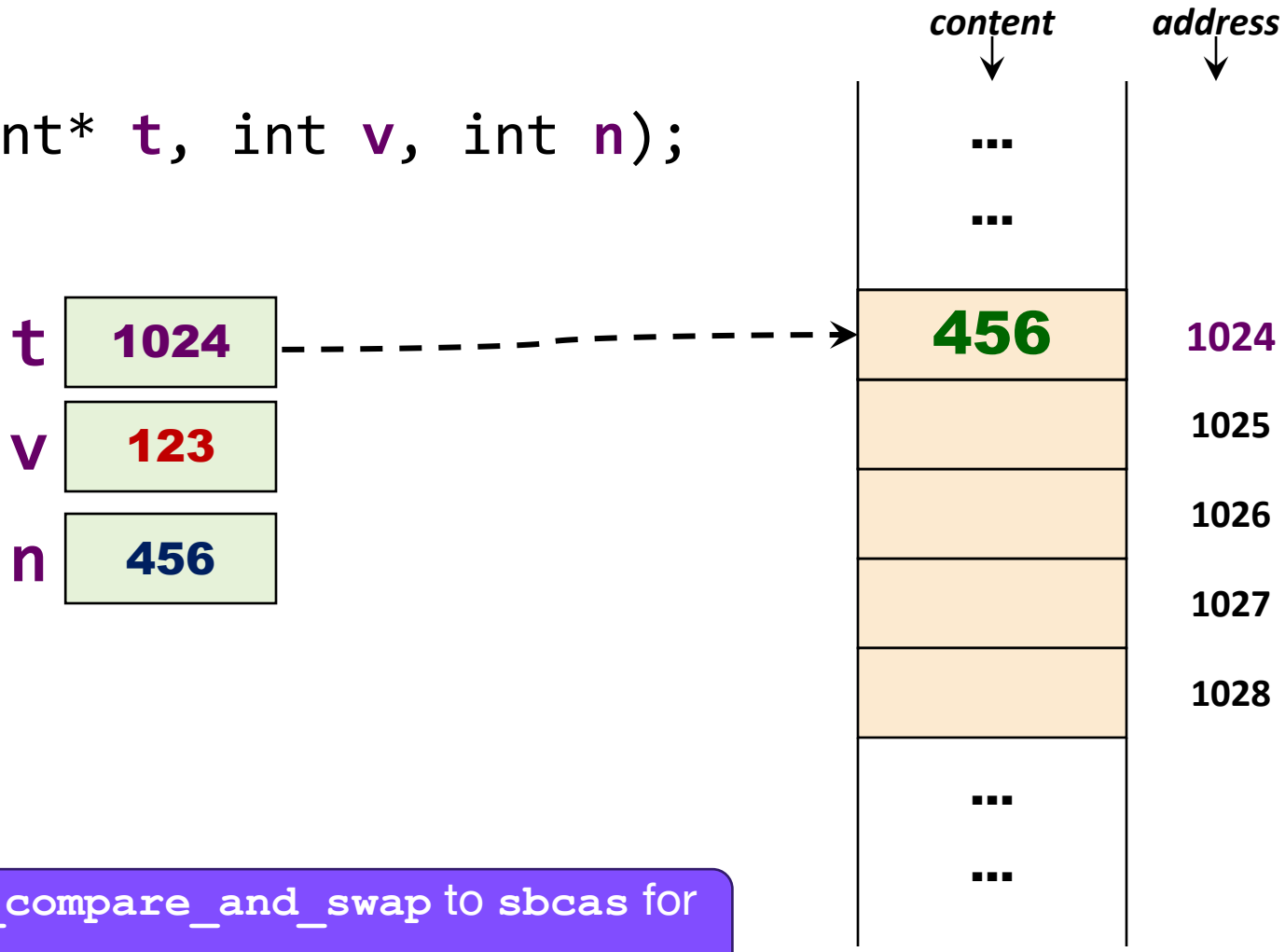


Note: Shorten `_sync_bool_compare_and_swap` to `sbcas` for readability.

Question 3

After Swapping

```
bool sbcas(int* t, int v, int n);
```



Note: Shorten `_sync_bool_compare_and_swap` to `sbcas` for readability.

Question 3

- Your task is to implement function **`atomic_increment`** on platform X.
- Your function should always return the incremented value of referenced location **`t`** and be free of race conditions. The use of busy waiting is allowed.

```
int atomic_increment( int* t ) {
    //your code here



}
```

Question 3

- Your task is to implement function **`atomic_increment`** on platform X.
- Your function should always return the incremented value of referenced location **`t`** and be free of race conditions. The use of busy waiting is allowed.

```
int atomic_increment( int* t ) {
    //your code here
    do {
        int temp = *t;
    } while (!_sync_bool_compare_and_swap(t, temp, temp+1));
    return temp+1;
}
```

Question 3

```
int atomic_increment( int* t ) {
    //your code here
    do {
        int temp = *t;
    } while (!_sync_bool_compare_and_swap(t, temp, temp+1));
    return temp+1;
}
```

- When another process or thread modifies the value at location pointed by **t**, the current process would be stuck at the while loop as it does not have a copy of the most updated value at location pointed by **t**.
- It would constantly fetch the most updated value at location pointed by **t** and only return the incremented value if has a copy of the most updated value at location pointed by **t**.

Question 4

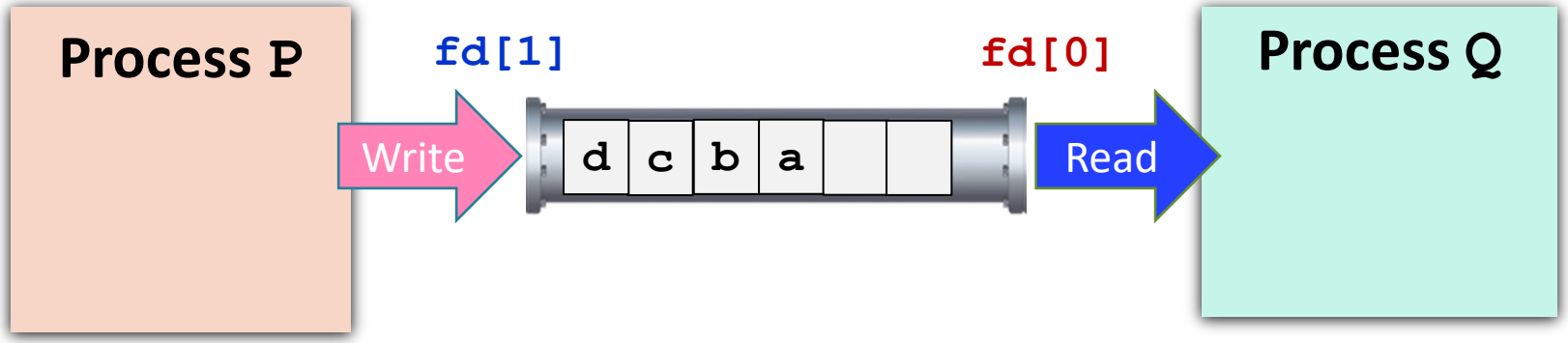
Question 4

You are required to implement an intra-process mutual exclusion mechanism (a lock) using Unix pipes. Your implementation should not use mutex (**`pthread_mutex`**) or semaphore (**`sem`**), or any other synchronization construct.

- Write your lock implementation below in the spaces provided.
- Definition of the pipe-based lock (**`struct pipelock`**) should be complete, but feel free to add any other elements you might need.
- You need to write code for **`lock_init`**, **`lock_acquire`**, and **`lock_release`**.

Line#	Code
1 2 3	**/* Define a pipe-based lock** ***/** struct pipelock { int fd[2]; };
11 12	**/* Initialize lock */** void lock_init(struct pipelock *lock) { }
21 22	**/* Function used to acquire lock */** void lock_acquire(struct pipelock *lock) { }
31 32	**/* Release lock */** void lock_release(struct pipelock * lock) { }

Recap: Pipe



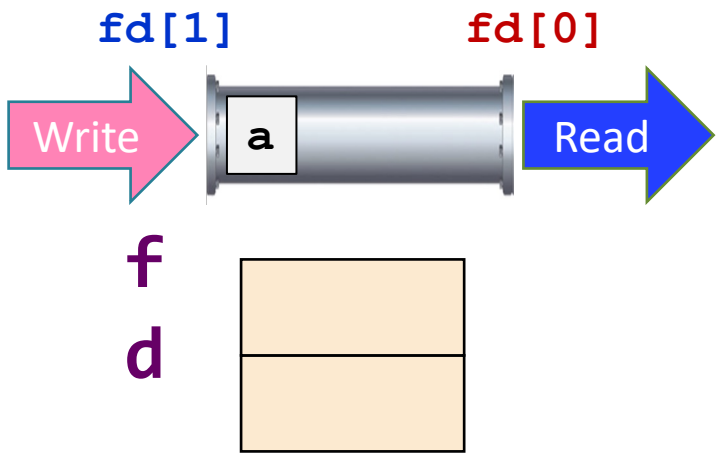
- **TL;DR:**
  - Can be shared between processes
  - FIFO, bounded buffer (size not an issue for us…)
  - Read blocks if no data
  - Read consumes data when available

Useful System Calls

```
int pipe(int pipefd[2]);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Question 4

Explanation: The first write is meant to initialize the lock such that exactly one thread can acquire the lock. This is done by writing a single character into a pipe.



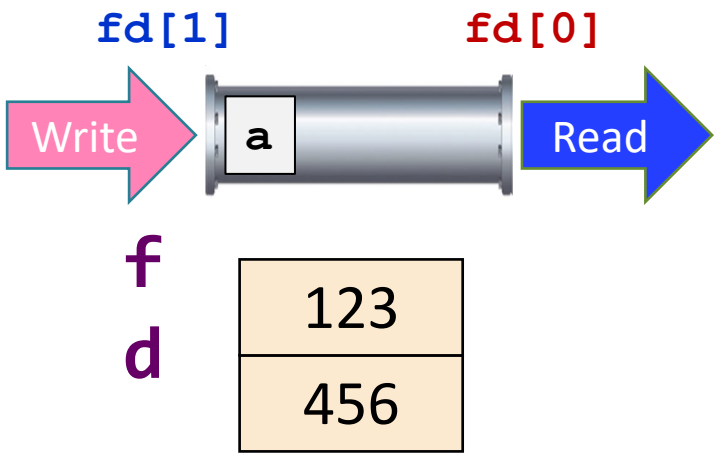

```
struct pipelock {
      int fd[2];
};

void lock_init(struct pipelock *lock) {

    pipe( lock->fd );
    write( lock->fd[1], "a", 1 );
}
```

Question 4

Explanation: Since there is only one character in the write end of the pipe, only one process can acquire the lock by reading the single character from the read end of the pipe.
Note: Read will block if there is no byte in the pipe.



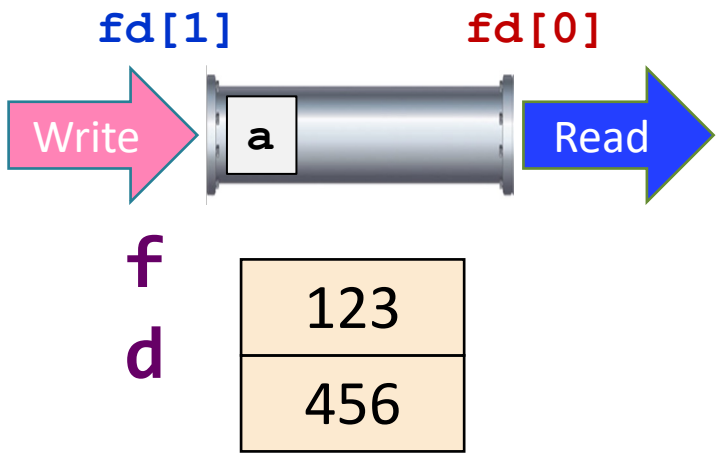

```
struct pipelock {
        int fd[2];
};

void lock_acquire(struct pipelock *lock) {

    char c;
    read(lock->fd[0], &c, 1);
}
```

Question 4

Explanation: A process can release the lock by writing exactly one byte or a single character back into the write end of the pipe to allow another process to acquire the lock.




```
struct pipelock {
      int fd[2];
};

void lock_release(struct pipelock *lock) {

    write( lock->fd[1], "a", 1 );

}
```

END OF TUTORIAL