Suggested answers to Activities Week 1-6

Activity 1.1

- Yes.
- It’s an illusion – by switching the CPU quickly between processes, each process appears to be making continuous progress. This is similar to seeing people move in a projected movie.
- You need to save/restore all the CPU hardware state that keeps track of the process’s execution, e.g., registers like SP, PC, PSR, etc. To/from main memory.
- Timer interrupts periodically give control to the OS, which can then force rescheduling even if the process doesn’t give up the CPU voluntarily.
- Yes. Otherwise, by disabling say the timer interrupt, a process can hoard the CPU.

Activity 1.2

- A person can think (brain). She can interact with her environment (eyes, ears, nose, mouth, touch, etc). She remembers what she learned (short/long term memory).
- CPU (brain). Input devices like keyboard (eyes). Output devices like speaker (mouth). Cache and main memory (ST memory). Secondary storage like disks (LT memory).
- CPU management, IO management, memory management, storage management, etc.

Activity 3.1

- N.
- 1 / (x + (1-x)/N).
  - *Remark*. In a massively parallel computer, N is huge. But if say just 10% of the code is not parallelizable, then the speedup will be limited to 10, no matter how big N is. Hence, it's important to extract as much parallelism as possible from the task to take advantage of parallel hardware.
- Limited number of physical CPUs. Multiprocessing overheads like context switching, IPC, or inter-process synchronization.

Activity 3.2

- The kernel CPU scheduler will consider the runnable thread for using the CPU.
  - Subject to competition with other runnable threads in the system.
- The whole user process is blocked. Hence, the user thread scheduler in the process can’t run, and the runnable user thread has no chance of being scheduled even if it is runnable.
- Context switching between user threads doesn’t have to trap to the kernel (no system call overhead).

Activity 4.1

1. Could use hardware getAndSet instruction, software Peterson’s Algorithm, etc, to provide atomicity for acquire() and release().
2. The operations acquire() and release() (the non-busy wait versions on Slide 6.24) are short, so a busy waiting solution for them is acceptable (the busy wait won’t take long and so won’t waste many CPU cycles).

Activity 4.2

- **P1**. Ensure mutual exclusion in updating the shared buffer. **P2**. For producer, ensure there’s at least one empty buffer slot. **P3**. For consumer, ensure there’s at least one full buffer slot.
- Three semaphores, one for each of P1, P2, P3.
  - One binary semaphore for the *mutual exclusion* problem P1. Two counting semaphores for the two *condition synchronization* problems P2 & P3.
- Call the semaphores mutex, empty, full for P1, P2, P3, respectively. Initialize them to 1, N, 0.

Activity 4.2 (cont'd)

```
public void produce(Work item) {

            acquire(empty);
            acquire(mutex);

            buffer[in] = item;
            in = (in + 1) % BUFFER_SIZE;

            release(mutex);
            release(full);
      }
```

Code for consumer is exactly analogous (exchange the places of empty and full).

Activity 4.3

- The semaphore's integer state variable (value) will increase by 1, i.e., the effect of release() is *remembered* even if no producers are yet asking for an empty slot.
  - That's why the race condition illustrated on Slide 6.36 won't happen for semaphores.
- The notify() will have no effects (since there are no producer threads in the wait set that can be waked up).
  - Hence, the race condition on Slide 6.36 could happen if we called wait()/notify() without holding a lock.
- The number of full slots is kept with a counting semaphore itself. So there's no need for a separate count variable.

Homework 4.1

- Attempt 1 is incorrect. It does not satisfy mutual exclusion. Initially, wantEnter[0] == wantEnter[1] == false. Consider this interleaving of P0's and P1's execution:
  - P0 tests wantEnter[1] == false, exits while loop;
  - P1 tests wantEnter[0] == false, exits while loop;
  - P0 sets wantEnter[0] to true and enters CS;
  - P1 sets wantEnter[1] to true and enters CS;

- Attempt 2 is incorrect. It doesn't satisfy progress (initially, turn == 0):
  - P0 does *not* want to enter CS, turn == 0 indefinitely;
  - P1 wants to enter, indefinitely stuck in while loop (although CS is available) because turn is 0;

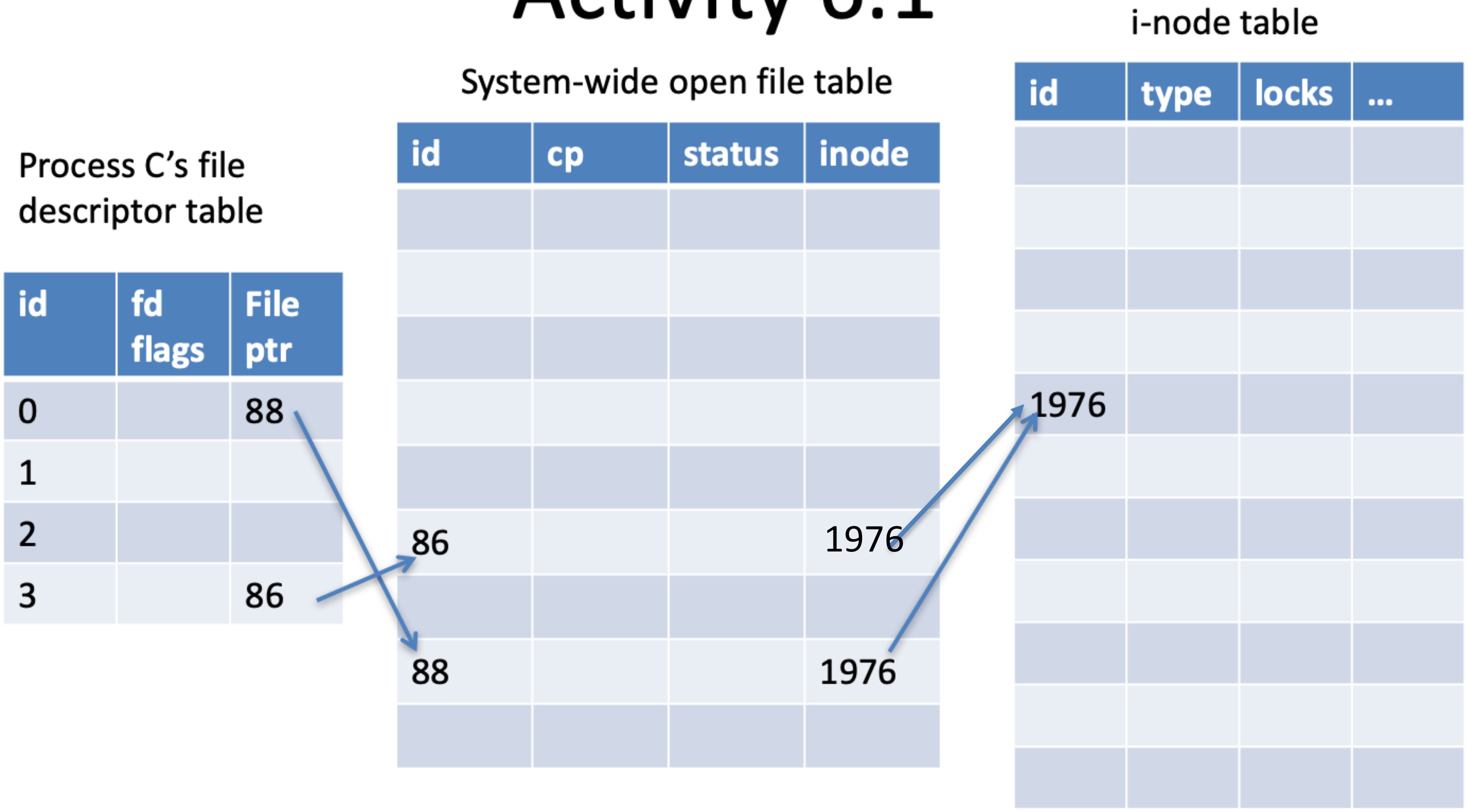
Homework 4.1 (cont’d)

- According to Peterson’s Algorithm, after *i* exits the critical section (CS), before it can enter again, it must set turn to *j*. So if *j* is waiting to enter (i.e., flag[*j*] == true), *i* must now wait at the while loop until after *j* got its own chance to enter the CS. Hence, *j* can’t be beaten twice in a row, and bounded waiting is satisfied with a bound of 1.

Activity 5.1

- Yes, the system is safe. P_1, P_3, P_4, P_0, P_2 is a safe sequence. P_1, P_3, P_4, P_2, P_0 is another one.
- Can’t grant (3, 3, 0) by P_4 because of insufficient resources.
- Can’t grant (0, 2, 0) by P_0 because the resulting resource allocation state isn’t safe.

Activity 6.1



Process C's file descriptor table

id	fd flags	File ptr
0		88
1		
2		
3		86

System-wide open file table

id	cp	status	inode
86			1976
88			1976

i-node table

id	type	locks	…
1976			

NB: entry number 88 in open file table can be replaced by any number not currently in use

Activity 6.2

1. **cat greet.txt** says file not found; **cat like.txt** displays the text “I love you”.
2. The symbolic link **home** is successfully created, showing that a cycle is allowed in the directory structure.
3. The **-L** option of **find** specifies whether the command should follow symbolic links. In our case, the .txt files can only be found by following the symbolic link **home**. Hence **find** with **-L** lists the .txt files, whereas **find** without **-L** doesn’t.
4. **find -L** lists the .txt files and terminates without getting into any infinite loops. The command is able to cope with a cyclic directory structure.
5. Ubuntu doesn’t allow hard links to directories. This restriction ensures that we can’t form cycles with hard links.