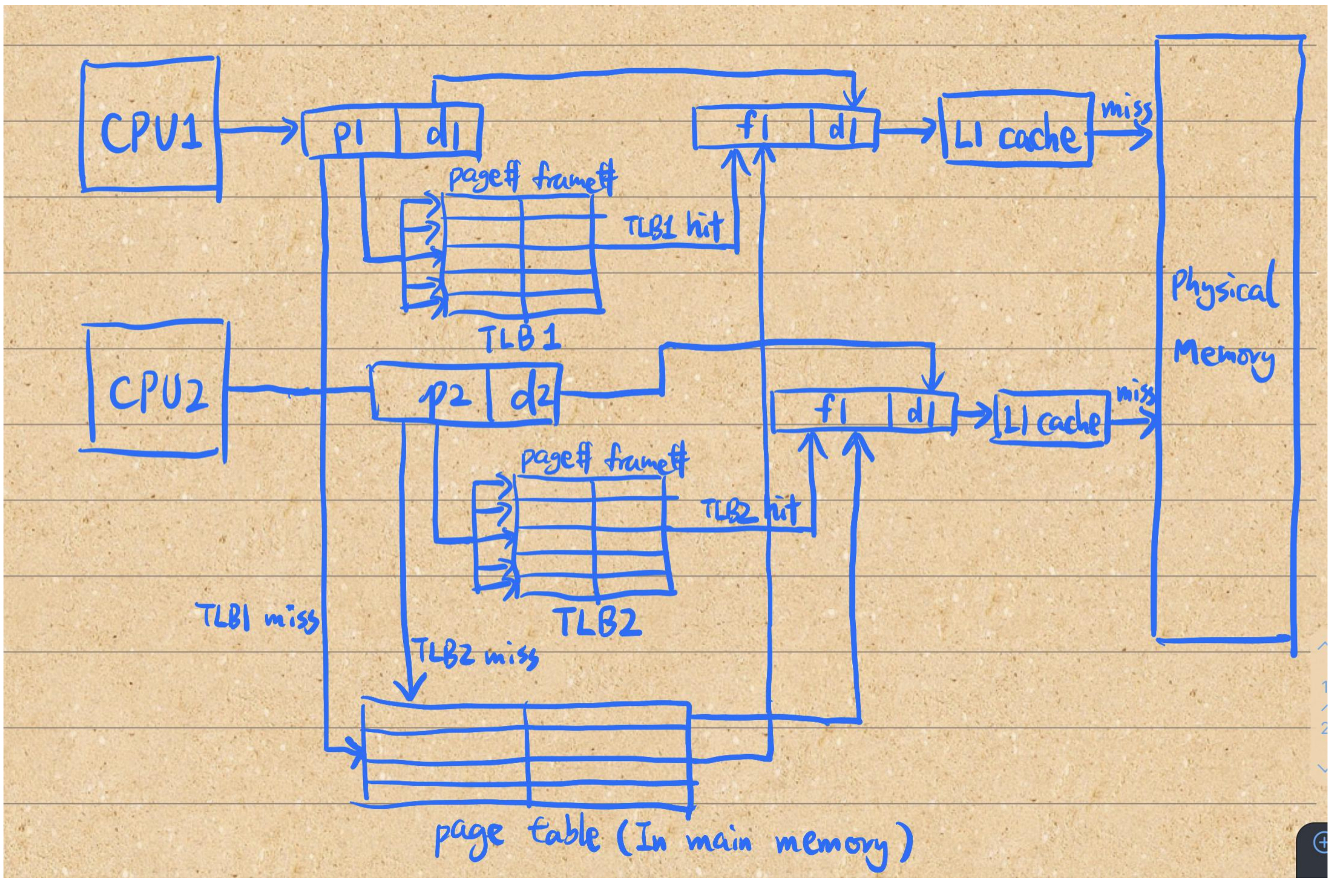
2021 Operating System Examination Solution

1. a. TLB is a hardware cache whose purpose is to enable fast lookup of physical address during address translation from virtual address to physical address. It's a cache of part of the page table.

   Each entry in the TLB consists of a page number and the associated frame number that stores the page.

   b. Something looks like this EXCEPT it's a 1️⃣ 4-level page table during look ups and not supposed to be only 1 level :). As long as you show 2️⃣ multiple cores in the diagram, 3️⃣ each core has its own TLB, 4️⃣ each core translates its own virtual address to the corresponding physical address (i.e. the address translation is per core per process), 5️⃣ what happens in address translation when there's a TLB hit/miss. 6️⃣ all cores use the same page table because the page table resides in the main memory, which is shared across different cores. Optionally, you can show L1/L2 caches if you want to. :)



   c. We know that the EAT is calculated as shown below when using a 4-level page table

$$\mathrm{EAT} = (b+c)\alpha + (b+5c)(1-\alpha) = b + 5c - 4\alpha c$$

   where α is the hit ratio, b is the TLB lookup time, and c is the memory access time.

   Using the given information, we know $b = 0.5$ ns, $c = 15$ ns and hence

$$\mathrm{EAT} = 0.5 + 5 \cdot 15 - 60\alpha = 75.5 - 60\alpha \text{ ns}$$

   In order for EAT to be 20 ns, the hit ratio has to be 0.925, or 92.5%

   d. i) In this case, there is a context switch happening on the same CPU core and hence the TLB of this core will be flushed. This will introduce a significant overhead of address translation for the new process because the page table information is not available in the TLB so the address translation process has to look up the page table in the main memory, which is slower.

ii) The TLB will not be flushed since two threads in the same process will share address space. This will have little impact in the address translation process as the other thread will most likely share the same entries that are stored in the TLB.

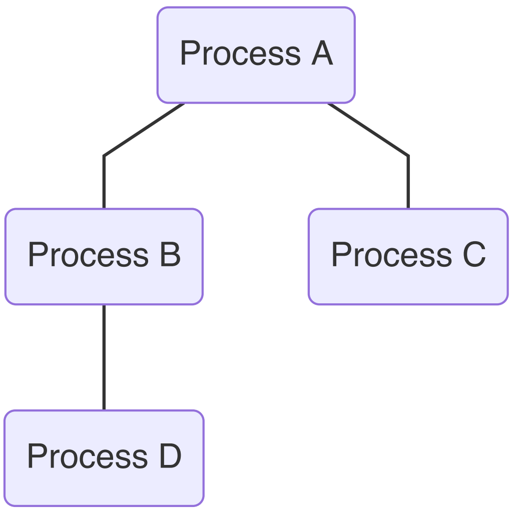
iii) The TLBs on both CPU cores will be flushed since context switch happens on both cores, assuming CPU1 will run a different process after switching the same process to CPU2. This will introduce significant overhead as TLBs on both CPUs are flushed, which means for the new process running on both cores, the address translation has to involve looking up the page table in the main memory, which is slower. (Similar or worse compared to i)

iv) The TLBs on both CPU cores will be flushed because this case is essentially equivalent to iii) where both cores are experiencing context switch, assuming CPU1 will run a different process after switching another thread of the same process to CPU2. The perforation implication is similar to the ones discussed in iii.

e. I would like to design the OS scheduler so that the running of one process on a CPU core will be not abruptly switched to another CPU core without significant reasons. Similarly, the OS scheduler should avoid switching a thread from one CPU core to another. Additionally, even on a single CPU core, we might want to reduce the frequency of context switch so that the impact of its overhead is low. (e.g. choose an appropriate time slice in the Round Robin Scheduler)

2. a. The kernel will first create a child process and then assign a process id to the child process. Next, it will copy the exact image/running state/memory state of the parent process to the child process. The child process will then be executed concurrently with parent process. The start line of child process is the line when the fork() system call happens. In the parent process, the fork() system call will return the pid of the child process. In the child process, the fork() system call will return 0 on success and -1 on failure. Failure to fork a new child process can happen when there is not enough memory to create another process. Optionally you can mention that the child process is added to OS process managing table or added to the group that also contains the parent proceses or things like this :)

   b. i) The process hierarchy looks like this



Process A executes line 1-9, 15, 16, 20, 21

Process B executes line 5-7, 10-13, 15, 16, 20, 21

Process C executes line 9, 15, 17, 18, 20, 21

Process D executes line 12, 15, 17, 18, 20, 21

You can either include the line with `fork()` or exclude it. Both are fine :)

ii) There are SO MANY possible answers. I'm too lazy to find them all, but here are 8 possible answers

```
ABCCDD
BACCDD
ABDCDC
BADCDC
ACDBCD
ACDBDC
ADCBCD
ADCBDC
```

iii) The program will NEVER output the string BCCADD. There are two letter 'C' in the output string. One of them is printed by process A (the parent process that starts running on line 5) and the other is printed by process B (the child process that starts running on line 5). The output from only running process A is 'AC' since it first prints 'A' on line 8 and then 'C' on line 16. Hence, at least one 'C' has to occur in the sequence after 'A', which makes the sequence 'BCCADD' impossible since both 'C' are before 'A'.

iv) We can add the line below between line 7 and line 8:

```
waitpid(pid, null, 0);
```

The result code looks like this

```
...
    if (pid != 0) {
        waitpid(pid, null, 0);  // wait for process B (the child) since the pid in process A (the parent) is the process id of B
        printf("A");
        pid = fork();
    } else {
...
```

c. The performance problem is caused by copying unnecessary new pages that are allocated to the child process during the fork() system call. Page copying is expensive, which will bring down the performance.

Some page creation is unnecessary. For instance, if a process that handles a heavy read request wants to create a child process, it is sufficient to let the child process share the same pages of its parent and marked it as read-only.

An extension to the fork() system call can be made similar to the discussion above: we can use the copy-on-write strategy to let fork() create a child process whose pages directly point to the frames of its parent.