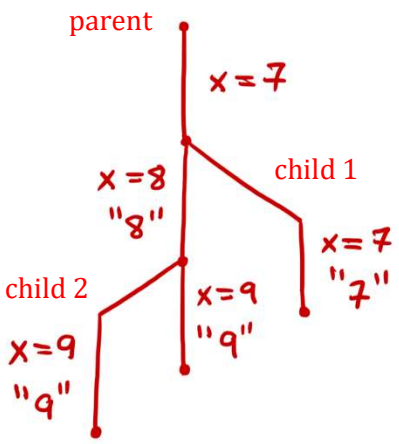
CSE 351 Section 9 – Virtual Memory

Fork and Concurrency:

Consider this code using Linux's `fork`:

```
int x = 7;
if( fork() ) {
    x++;
    printf(" %d ", x);
    fork();
    x++;
    printf(" %d ", x);
} else {
    printf(" %d ", x);
}
```



What are *all* the different possible outputs (i.e. order of things printed) for this code?
(Hint: there are four of them.)

Note: `fork()` returns 0 to the child, and the child's process ID (PID) to the parent.

From our first fork, we know child 1 will print "7", but since this print statement is not dependent on any other code (besides the initial `fork()`), it could be printed at any time.

We also know the parent will have to print "8" before the second call to `fork()`, meaning that the "8" is printed before the "9"s. Since the parent and child 2 both print out "9", even if the ordering of their prints changes, the output will not.

Possible orderings:

- 7 8 9 9
- 8 7 9 9
- 8 9 7 9
- 8 9 9 7

Exercises:

1) Name three specific benefits of using virtual memory:

Bridges memory and disk in memory hierarchy (treats Mem as a cache for Disk).
Simulates full address space for each process (simplifies memory management for programmers).
Enforces protection (and sharing) between processes.

2) What should happen to the TLB when a new entry is loaded into the page table base register?

Updating the PTBR means that we are switching processes. The page table entries currently in the TLB corresponded to the old page table & process, so none of them are valid once we context switch. **The valid bits of the TLB should all be set to 0 (invalidated).**

3) Fill in the formulas below using *descriptions*, not variables:

Page offset bits = $\log_2$(bytes in a page)

Virtual address bits = virtual page number bits + page offset bits

Physical address bits = physical page number bits + page offset bits

Virtual page number bits = $\log_2$(number of virtual pages)

Entries in a page table = number of virtual pages

4) Fill in the following table:

VA width (n)	PA width (m)	Page size (P)	VPN width	PPN width	Bits in PTE (assume V, D, R, W, X)
32	32	16 KiB	18	18	23
32	26	8 KiB	19	13	18
36	32	32 KiB	21	17	22
40	36	32 KiB	25	21	26
64	40	64 KiB	48	24	29

$p = \log_2 P$, VPN width $= n - p$, PPN width $= m - p$, bits in PTE = PPN width + 5

5) **Processor:** 16-bit addresses, 256-byte pages

TLB: 8-entry fully associative with LRU replacement

- Track LRU (shown in decimal) using 3 bits to encode the order in which pages were accessed, with 0 being the most recent

At some time instant, the TLB for the current process is in the initial state given below.

Assume that all page table entries that are not in the initial TLB have read and write permissions, but no execute permission (i.e. R = 1, W = 1, X = 0).

- OS will assign new pages starting at PPN 0x20, with read and write permissions but no execute permission (i.e. R = 1, W = 1, X = 0).

Fill in the final state of the TLB according to the access pattern below. For each access, indicate if it leads to a:

a) TLB hit? b) TLB miss? c) Page fault? d) Protection fault?

Initial TLB:

TLBT	PPN	Valid	R	W	X	Dirty	LRU
0x01	0x11	1	1	1	0	1	0
0x02	0x18	1	1	0	0	0	6
0x10	0x13	1	1	1	1	1	1
0x20	0x12	1	0	1	0	0	5
0x00	0x00	0	0	0	0	0	7
0x11	0x14	1	1	0	0	0	4
0xAC	0x15	1	1	0	0	0	2
0x34	0x16	1	1	1	0	1	3

Page Table (partial):

VPN	Valid	PPN
0x0	0	0x00
0x1	1	0x19
0x2	1	0x18
0x3	1	0x17
0x4	0	–
0x5	0	–
0x6	1	0x1A
0x7	0	–

VPN	Valid	PPN
0x8	1	0x1C
0x9	1	0x1D
0xA	0	0x1E
0xB	1	0x1F
0xC	0	–
0xD	1	0x09
0xE	0	–
0xF	1	0x1B

n = 16, p = 8, so VPN width = $n - p$ = 8 bits. TLB is fully associative, so S = 1 and TLBT = VPN.

1. Read 0x11F0 → TLBT = 0x11, TLB Hit

TLBT	PPN	Valid	R	W	X	Dirty	LRU
0x01	0x11	1	1	1	0	1	1
0x02	0x18	1	1	0	0	0	6
0x10	0x13	1	1	1	1	1	2
0x20	0x12	1	0	1	0	0	5
0x00	0x00	0	0	0	0	0	7
0x11	0x14	1	1	0	0	0	**0**
0xAC	0x15	1	1	0	0	0	3
0x34	0x16	1	1	1	0	1	4

2. Write 0x0301 → TLBT = 0x03, TLB Miss, no page fault
   Load PPN 0x17 into TLB and replace LRU block

TLBT	PPN	Valid	R	W	X	Dirty	LRU
0x01	0x11	1	1	1	0	1	2
0x02	0x18	1	1	0	0	0	7
0x10	0x13	1	1	1	1	1	3
0x20	0x12	1	0	1	0	0	6
0x03	**0x17**	**1**	**1**	**1**	**0**	**1**	**0**
0x11	0x14	1	1	0	0	0	1
0xAC	0x15	1	1	0	0	0	4
0x34	0x16	1	1	1	0	1	5

3. Write 0x20AE → TLBT = 0x20, TLB Hit

TLBT	PPN	Valid	R	W	X	Dirty	LRU
0x01	0x11	1	1	1	0	1	3
0x02	0x18	1	1	0	0	0	7
0x10	0x13	1	1	1	1	1	4
0x20	0x12	1	0	1	0	**1**	**0**
0x03	0x17	1	1	1	0	1	1
0x11	0x14	1	1	0	0	0	2
0xAC	0x15	1	1	0	0	0	5
0x34	0x16	1	1	1	0	1	6

4. Write 0x0532 → TLBT = 0x05, TLB Miss, Page fault
   Map 0x05 to PPN 0x20 and replace LRU

TLBT	PPN	Valid	R	W	X	Dirty	LRU
0x01	0x11	1	1	1	0	1	4
0x05	**0x20**	**1**	**1**	**1**	**0**	**1**	**0**
0x10	0x13	1	1	1	1	1	5
0x20	0x12	1	0	1	0	1	1
0x03	0x17	1	1	1	0	1	2
0x11	0x14	1	1	0	0	0	3
0xAC	0x15	1	1	0	0	0	6
0x34	0x16	1	1	1	0	1	7

5. Read 0x0E15 → TLBT = 0x0E, TLB Miss, Page fault
Map 0x0E to PPN 0x21 and replace LRU

TLBT	PPN	Valid	R	W	X	Dirty	LRU
0x01	0x11	1	1	1	0	1	5
0x05	0x20	1	1	1	0	1	1
0x10	0x13	1	1	1	1	1	6
0x20	0x12	1	0	1	0	1	2
0x03	0x17	1	1	1	0	1	3
0x11	0x14	1	1	0	0	0	4
0xAC	0x15	1	1	0	0	0	7
0x0E	**0x21**	**1**	**1**	**1**	**0**	**0**	**0**

6. Write 0xACFF → TLBT = 0xAC, TLB Hit, Protection fault

TLBT	PPN	Valid	R	W	X	Dirty	LRU
0x01	0x11	1	1	1	0	1	6
0x05	0x20	1	1	1	0	1	2
0x10	0x13	1	1	1	1	1	7
0x20	0x12	1	0	1	0	1	3
0x03	0x17	1	1	1	0	1	4
0x11	0x14	1	1	0	0	0	5
0xAC	0x15	1	1	0	0	0	**0**
0x0E	0x21	1	1	1	0	0	1

Final TLB:

TLBT	PPN	Valid	R	W	X	Dirty	LRU
0x01	**0x11**	**1**	**1**	**1**	**0**	**1**	**6**
0x05	**0x20**	**1**	**1**	**1**	**0**	**1**	**2**
0x10	**0x13**	**1**	**1**	**1**	**1**	**1**	**7**
0x20	**0x12**	**1**	**0**	**1**	**0**	**1**	**3**
0x03	**0x17**	**1**	**1**	**1**	**0**	**1**	**4**
0x11	**0x14**	**1**	**1**	**0**	**0**	**0**	**5**
0xAC	**0x15**	**1**	**1**	**0**	**0**	**0**	**0**
0x0E	**0x21**	**1**	**1**	**1**	**0**	**0**	**1**