Introduction to Computer Systems

Final Exam

NAME: Matthew Schilling

Details: Monday, December 6 (Available online)
We have made accommodations for students unable to take it on Monday

See CampusWire and Email for how to upload your exam and when

To be done individually (honor system)

Covers all material from lectures 10-20 as well as associated reading/projects

All questions assume 64 bit x86 ("x64"), the architecture of the book
Show your work for partial credit
Answer all questions!
Please do not discuss the exam with others until everyone has taken it

*This exam is **watermarked with your name and the PDF also has your name cryptographically embedded in it.** Please do not leak it to anyone. If we see your copy in the wild, we can easily trace it back to you, and turn the case over to Northwestern for handling.*

Question	Score
TF	
SA1	
SA2	
SA3	
SA4	
SA5	
ID1	
ID2	
ID3	

True/False Questions (answer all of the following) (20% of total)

1. **T** (**F**) The TCP protocol provides a reliable byte stream communication service between processes on two machines even if the underlying network is unreliable. happened in 1 SETI Lab
2. **T** (**F**) A bug resulting from a race condition among threads appears nondeterministically, but appears every time the program is run. → Depending on bug, could be b/c threads race
3. (**T**) **F** Instruction fetches exhibit spatial locality, but not temporal locality.
4. **T** (**F**) An interrupt from a keyboard is synchronous.
5. **T** (**F**) Memory reads and writes of stack variables cannot exhibit spatial locality.
6. (**T**) **F** The IP protocol allows networks to interoperate regardless of their hardware technologies.
7. **T** (**F**) An illegal instruction exception is asynchronous.
8. **T** (**F**) Reading data from random addresses in a typical DRAM can be done with the same throughput as reading data from random addresses. → caches? (Have less throughput)
9. **T** (**F**) In a k-way set associative cache, each set contains k cache lines.
10. (**T**) **F** While the kernel handles an interrupt, it may decide to perform a context switch to another process or thread.
11. (**T**) **F** Page tables map virtual page numbers (VPNs) to physical page numbers (PPNs), and so each page table entry (PTE) contains both a PPN and metadata such as read, write, execute, kernel-mode, and present.
12. (**T**) **F** Each thread has a separate virtual address space.
13. (**T**) **F** The TLB caches virtual page number (VPN) to PTE (page table entry) mappings loaded from the page table hierarchy.
14. (**T**) **F** Unix files are uninterpreted streams of bytes.
15. (**T**) **F** In a malloc implementation that uses an implicit list with boundary tags, it is always possible to coalesce two blocks that are adjacent on the free list.
16. (**T**) **F** The two primary functions of a linker are symbol resolution and relocation.
17. **T** (**F**) Using the Unix system calls, it is possible to read byte `k` of a file without reading bytes `0..k-1` first.
18. **T** (**F**) A page fault in a user program is handled by code in the kernel and never has an effect that is visible to the user program.
19. (**T**) **F** The `mmap()` system call allows a thread to map a chunk of a file into a region of the process's virtual address space.
20. (**T**) **F** Multi-level page tables can compactly represent virtual address spaces that have many virtual pages that are not present.

Short Answer Questions (answer all of the following) (30% of total)

1. What is the difference between latency and throughput (bandwidth)? Give an example in your answer.

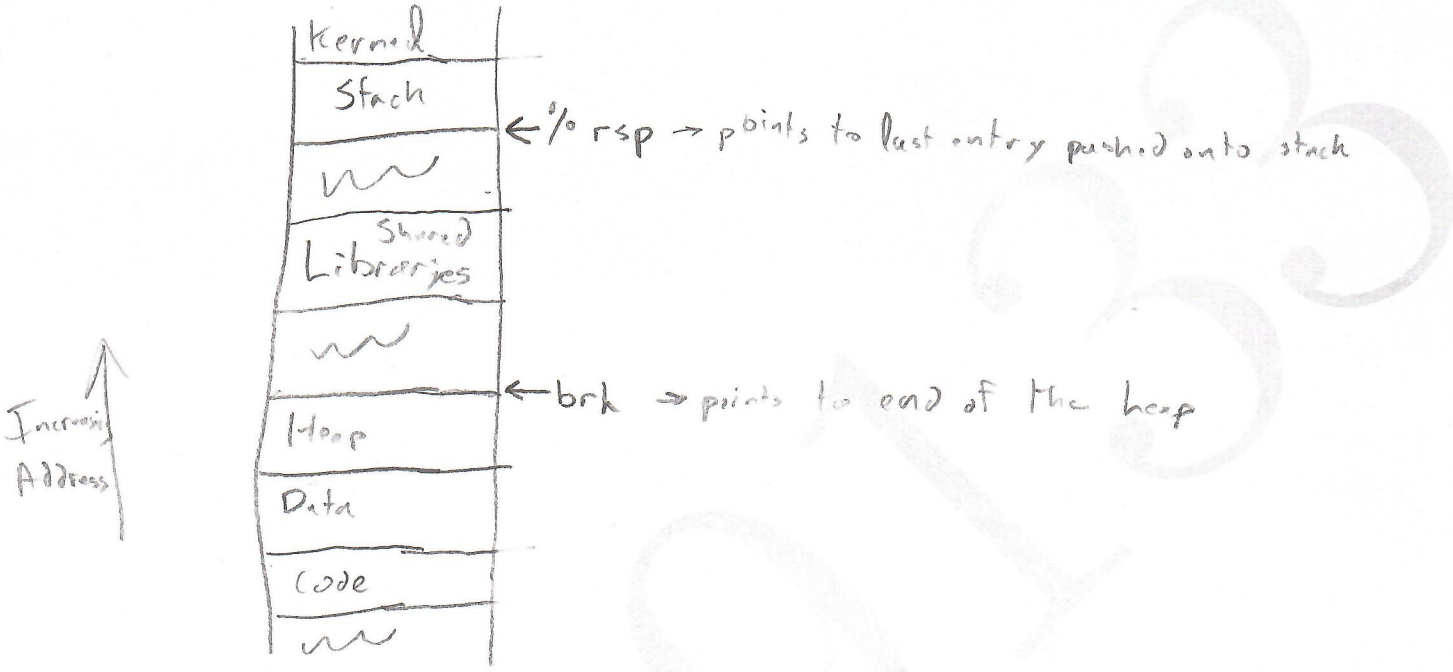
Latency - How long you wait for data to begin to load.
For example, this is how long it takes your computer to receive the first byte of data from a webpage, comparable to the first drop of water from a hose.

Throughput/Bandwidth - How much data can come through at once. In the web page example, this is how many bytes load a second after the first byte arrives, or in the hose example it's the flux of the water.

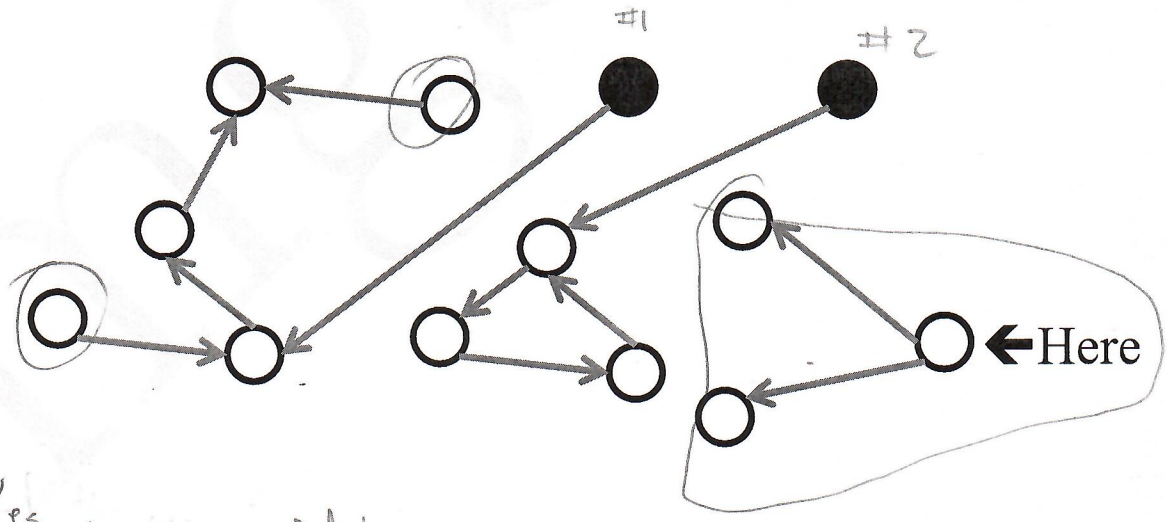
TL;DR: Latency is wait time, throughput is data flux

2. `fork()` and `exec()` questions. (Multiple choice questions, circle appropriate answer, assume no errors occur)
   a. A call to `fork()` returns (a) never, (b) once, (c) twice, once for the parent and once for the child.
   b. `fork()` returns (a) nothing, (b) the child pid to both processes, (c) the child pid to the parent and zero to the child.
   c. After a `fork()`, the parent and child processes execute (a) concurrently, (b) consecutively.
   d. After a `fork()`, the parent and child processes (a) share the same address space, (b) have duplicate but separate address spaces, (c) have different address spaces.
   e. After a `fork()`, the parent and child processes (a) have independent open files, (b) have shared open files, (c) the child has no open files.
   f. A call to `exec()` returns (a) never, (b) once, (c) twice.

3. Draw the virtual address space of a process, labeling the kernel, text/code, stack, data, heap, and shared libraries. Indicate where `%rsp` and `brk` point.



4. The black filled-in nodes in the following memory graph are root nodes. (a) Circle all the nodes that are garbage. (b) Suppose we are using *reference counting*. Consider the node marked "Here", and assume that it and the two nodes it points to have a reference count of 1. If the node marked "Here" is deleted, will we be able to correctly delete the other two nodes it points to? Why or why not?



Yes, you can delete the nodes here's node points to, as their references would fall to zero since there are no cycles. The same cannot be said if you delete the root node I labelled as #2.

5. In the following program, how many threads exist after 10 seconds? It might help to draw a timeline.

```
void *worker(void *x) {
   int tid;
   while (1) {
      sleep(1); // sleep for one second
      pthread_create(&tid, NULL, worker, NULL);
   }
}

int main() {
   worker(NULL);
}
```

2^n threads, so there will be 2^{10} threads @ ten seconds.

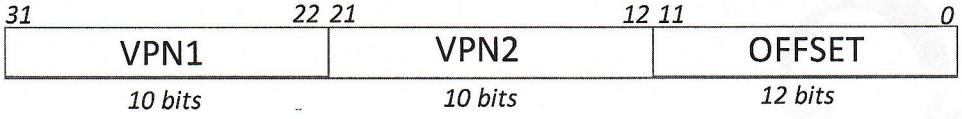
→ Know this b/c each second, each existing thread duplicates, i.e. exp. growth

$2^{10} = 4^5 = 1024$ threads

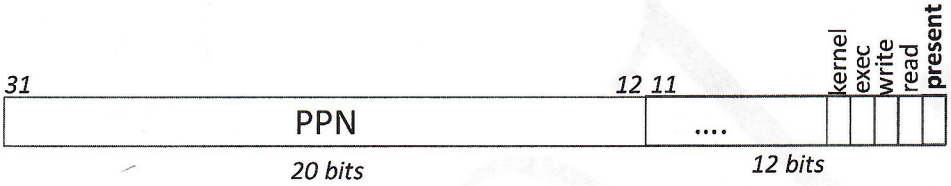
very clever, thank you for this question!

In Depth Questions (answer all of the following) (50 % of total)

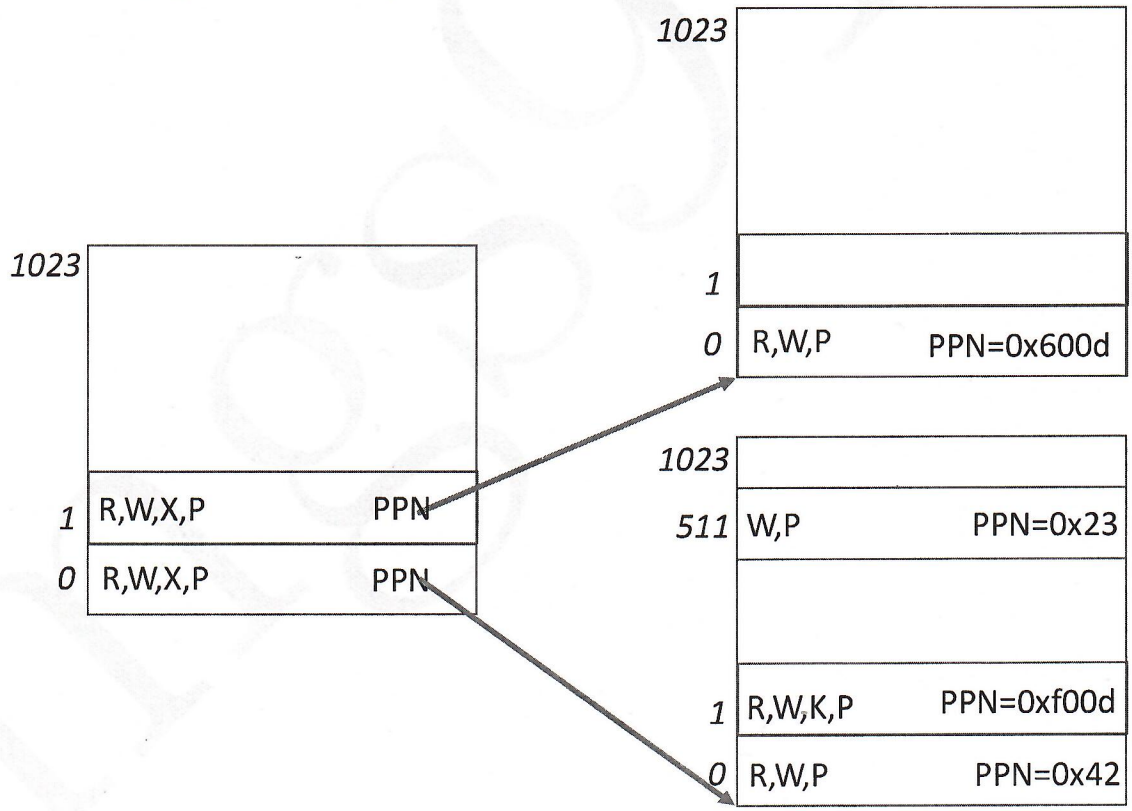
1. Suppose we have a machine with a 32-bit virtual address space and a 32-bit physical address space. This is the same machine as in homework 4, and paging is implemented using 2-level page tables. The virtual address looks like:



and a page table entry (PTE) looks like:



The current page table hierarchy/tree is as follows:



All unmarked entries are zeroed (and thus are **NotPresent**).

The problem continues on the next page.

Fill out the following table. Each READ, WRITE, or IFETCH (instruction read) is for a single byte. First, for each operation and virtual address, break the virtual address down into its VPN1, VPN2, and Offset fields. You can write these in binary or hex. Next, translate the virtual address given the page table hierarchy shown in the previous page.[1] Each operation involves only a single byte. Each operation is done in user mode. If the translation succeeds, give the resulting physical address. If the translation results in a page fault, give the reason why.

12 bits → 3 hex

10 bits → 2.5 hex

Operation and Virtual Address	VPN1	VPN2	Offset	Physical Address or Page Fault+Reason
WRITE 0x5	0x0	0x0	0x5	0x47
WRITE 0x1ff005	0x0	0x1FF	0x5	0x28
READ 0x1ddd	0x0	0x1	0xddd	Page Fault, user tried to access kernel
WRITE 0x400008	0x1	0x0	0x8	0x6016
READ 0x401033	0x1	0x1	0x33	Page fault, page not present
IFETCH 0x1ff006	0x0	0x1FF	0x6	Page Fault, not an instruction

d = 1101

y = 0100

$0x1FF = 16^2 + 15 \cdot 16 + 15$

$256 + 240 + 15$

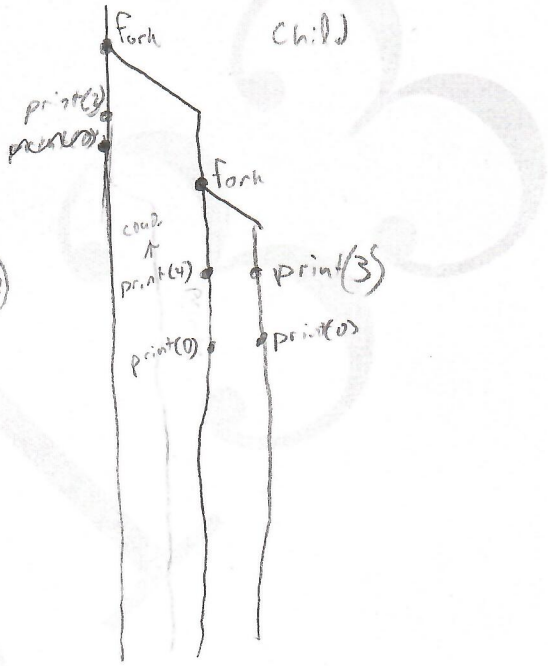
511

16
× 16
96
160
256

[1] VPN1, VPN2, and Offset will be helpful!

2. Consider the C program below. Assume that all functions return without error. Recall that `fork()` returns zero to the child process and nonzero to the parent process. It may be helpful to draw a process tree / timeline.

```
int main () {
  if (fork() == 0) {
    if (fork() == 0) {
      printf("3");
    } else {
      pid_t pid;
      int status;
      if ((pid = wait(&status)) > 0) {
        printf("4");
      }
    }
  } else {
    printf("2");
    exit(0);
  }
  printf("0");
  return 0;
}
```



For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program. If you're unsure, show your reasoning. It may be very helpful to draw a process tree and annotate it.

a. 32040	(Y)	N
b. 34002	Y	(N)
c. 30402	Y	(N)
d. 23040	(Y)	N
e. 40302	Y	(N)

Print(2) will be one of the 1st Ops.
to happen, b/c the children need to Fork still/at the same time as the print

definitely no, 4 is likely one of the last things printed

most likely sequence

3 more likely to come before 4

Exit of parent does not terminate children, just means return value of zero goes to garbage collection

3. You are running the following code on a machine with separate data and instruction caches.

```
int i, dotprod, n, a[n], b[n];

for (i=0;i<n;i++) {
   dotprod += a[i] * b[i];
}
```

You can assume that `i`, `n` and `dotprod` are allocated to registers, and thus there are only two memory references for data per loop iteration. Ignore instruction fetches. Assume that `n` is very large – there is much more data than will fit in the data cache. Assume the data cache is ***fully associative***.

a. Does the data reference pattern of this code exhibit spatial locality, temporal locality, or both? Explain.
b. The data cache has a block size of 64 bytes. `int`s are 4 bytes long on this machine. What is the miss rate?
c. Suppose that a data cache hit takes 1 ns while a cache miss takes 100 ns but loads an entire cache block. What is the average access time for the data references?
d. Now consider a ***k-way set-associative*** cache. What is the minimum value of ***k*** that would guarantee the same miss rate as the fully associative cache of the previous parts, regardless of the addresses of the `a` and `b` arrays.

a) Since i, n, + dotprod are stored in registers, only spatial locality is exhibited because the values of a[i] + b[i] are not used again. Spatial locality exists because you are accessing the next value in an array, which is the next value in memory. Though, depending on how many cache lines you have (if outrageous), this locality may not be used.

b) $\frac{60}{64} = \frac{15}{16} = 0.9375 \rightarrow 93.75\%$
→ wrong bytes that may be checked
→ total bytes to be checked

c) $t_{acc} = p_h \cdot t_h + (1-p_h) \cdot t_m = 0.0625 \cdot 1ns + 0.9375 \cdot 100 ns = 93.8125 ns$

d) 16