## Metrics

(1) Solve the following:
   A. How much is 2^13 in decimals?
   B. How much is roughly 1 billion in power of 2?
   C. How much is 2^64/2^21 in power of 2?
   D. How much is 128MB/4KB ?
   E. How much is log2(8192)?
   F. $2^{30}$bytes of storage equals?
   G. 1 Gbps network bandwidth equals how many bits per second (bps)?

(2) When measuring I/O throughput, what is the difference between the units
   A. MBps and Mbps
   B. KBps and Kbps?

(3) How is a Mebibyte different from Megabyte?

(4) How much are these units in decimals?
   A. Pico
   B. Nano
   C. Micro
   D. Milli
   E. Kilo
   F. Mega
   G. Giga
   H. Tera
   I. Peta

Hint:
For metric system: See http://www.chemteam.info/Metric/Metric-Prefixes.html
For size of information in computers see: https://web.stanford.edu/class/cs101/bits-gigabytes.html

(5) Replace "?" below with the correct answer
   A. 1 Nanosecond = ? seconds
   B. 500 Milliseconds = ? seconds
   C. 4 KB = ? bytes
   D. 4 Kilometers = ? meters
   E. 4Kbps = ? bits per second

## OS Definition, ISA

1. What is an Operating System? List its primary responsibilities.

2. What are the three (or four) different ways in which OS code can be invoked? Explain.

3. Explain the following interfaces in a computer system
    a. Instruction Set Architecture (ISA)
    b. User Instruction Set Architecture (User ISA),
    c. System ISA,
    d. Application Binary Interface (ABI).
    e. Application Programmers' Interface (API)

4. Why doesn't a program (executable binary) that is compiled on the linux machine execute on a Windows machine, even if the underlying CPU hardware is the same (say x86)?

5. What is meant by virtualization? Give examples of many(virtual)-to-one(physical, one-to-many, and many-to-many resource virtualization.

6. What was the first computer? First OS? First programmer? First language?

## Processes

(1) (a) What is a process? (b) How is a process different from a program?

(2) In the memory layout of a typical process, why do stack and heap grow towards each other (as opposed to growing in the same direction)?

(3) In terms of call-return behavior, how are the fork() and exec() system calls different from other system calls?

(4) Describe the process lifecycle illustrating the states and transitions.

(5) **[10 pts]** Which state transitions occur in a process lifecycle when a process

a. Makes a blocking read() system call
b. Exceeds its CPU timeslice
c. Is interrupted by a hardware interrupt
d. Dereferences a NULL pointer.
e. Attempts to acquire a blocking lock that is taken by another process?
f. Is pre-empted
g. Voluntarily yields the CPU

(6) During a process lifecycle, what events can cause the following transitions?
(a) Ready to Running state
(b) Running to Ready state
(c) Ready to Blocked state
(d) Blocked to Ready state

(7) What is a zombie process? Why does the Operating System maintain the state of zombie processes? List two ways in which a parent process can prevent a child process from becoming a zombie.

(8) Why are frequent context switches expensive in terms of system performance?

(9) What is cold-start penalty? What are some ways to reduce it?

(10) What are some key factors that affect application performance after a context switch?

# Threads

1. What are threads? How do they differ from processes? How are they similar?
2. What state do threads share? What state is different?
3. Why does context-switching between threads incur less overhead than between processes?
4. Briefly explain
   (a) User-level threads
   (b) Kernel-level threads
   (c) Local thread scheduling
   (d) Global thread scheduling

5. What are the benefits and disadvantages of using user-level and kernel-level threads?

6. What combinations or user/kernel threads and global/local scheduling are feasible and why?

7. What kind of applications benefit the most from kernel-level threads support? What kind of applications benefit most from user-level threads support? Explain why with examples?

8. Explain how a web server could use threads to improve concurrency when serving client requests.

9. What happens if a thread in a multi-threaded process crashes? How can you improve the robustness (fault-tolerance) of a multi-threaded application?

10. Event-driven programming
    (a) What is the "event-driven" programming model?
    (b) What does the structure of a typical event-driven program look like?
    (c) When would you prefer an event-driven programming model over a thread-based programming model?

11. What is the problem with long-running event handlers? How do threads solve this problem?

12. What type of applications would be more suitable for thread-based programming compared to event-driven programming?

13. What are callbacks and what problems can they cause when used with threads?

14. (a) How are threads better than processes?
    (b) How are processes better than threads?

15. Assume a single-CPU system. You are given three multi-threaded processes. P1 does a lot of computation, but little I/O. P2 does lots of I/O but little computation. P3 does a reasonable mix of both computation and I/O. What kind of threads would you prefer for each process? Explain why?

## Inter-Process Communication

1. List any five inter-process communication mechanisms, with a one line description for each?

2. When using a pipe for inter-process communication, why should a process promptly close any unused write descriptors to the pipe? Also give an example of what happens if it doesn't.

3. Let's say a **chain of filters** refers to a series of commands whose standard inputs and standard outputs are linked by pipes. For example,

   "ps -elf | grep bash | more"

   is a chain with three commands.

   In the general case,

   "command1 | command2 | command 3 | ... | command K"

   is a chain of filters with K commands.

   Suppose you were implementing a shell (e.g. csh, bash, tcsh, ksh, etc.), how would you go about supporting a chain of filters with ***arbitrary*** number of commands? Explain. Don't write actual code.

4. What's the difference between byte-stream vs. message oriented communication?

5. How would you perform input/output redirection from/to a file from a process?

# System Calls, Traps, Exceptions

1. What is the difference between a system call, hardware interrupt, a software interrupt (trap), and an exception? Give examples of each.

2. What is a system call? How is a system call different from a normal function call?

3. How are trap handlers, exception handlers, and interrupt handlers different from system calls?

4. What steps take place when a system call is invoked by a process?

5. What is a system call table? Why is it needed? OR What role does it play in OS security?

6. Explain the CPU-privilege transitions during a system call.

7. (a) Why do some operating systems, such as Linux, map themselves (i.e. the kernel code and data) into the address space of each process? (b) What is the alternative?

8. Assume a mainstream monolithic OS, such as Linux. When a process makes a system call, how can the system call mechanism avoid any context switching overhead between the calling process and the OS? (as opposed to the overhead seen when switching between two processes).

9. Rootkits (malicious code in the kernel) can intercept system calls made by processes (all processes or a specific process) and replace the original system call behavior with some other behavior.. How would you go about implementing such behavior? Describe the design but don't write any code.

10. How does the OS ensure that a user-level process does not jump to, and execute, arbitrary code in the OS memory?

11. Does a system call invocation by a process trigger a context switch? Why or why not?

## Kernel Modules

(1) Why are memory access errors (such as segmentation fault and null pointer dereferencing) more dangerous in kernel space than in user space?

(2) What are kernel modules?

(3) What are some advantages of kernel modules?

# Concurrency

1. Define Concurrency. How does it differ from parallelism?

2. Explain the differences between apparent concurrency and true concurrency.

3. What is "atomicity" and why is it important? (in concurrency, not physics!)

4. Briefly explain with examples
   **A.** Critical Section
   B. Race condition
   C. Deadlock

5. What's wrong with associating locks with code rather than shared resources?
6. Describe the behavior of (a) UP and DOWN operations on a semaphore, (b) WAIT and SIGNAL operations on a condition variable.

7. Under what situation would you use (a) Blocking locks, (b) Non-blocking locks, and (c) Spin locks. Which of these locks can be used in interrupt handlers and how?

8. When should you NOT use (a) blocking locks, (b) non-blocking locks, and (c) spin-locks?

9. (a) Consider (a) Blocking locks, (b) Non-blocking locks, and (c) Spin locks. Which of these locks can be used in interrupt handlers and how?

10. What is deadlock? How can you prevent it?

11. What is the main difference between a binary semaphore and a counting semaphore?

12. What is priority inversion? How can it be prevented?

13. Explain how a deadlock can occur in the operating system between code executing in the user-context and code executing in interrupt handlers. Also explain how you would prevent such a deadlock.

14. Multiple processes are concurrently acquiring and releasing a subset of locks from a set of N locks L1, L2, L3, ….., LN. A process may try to acquire ***any subset*** of the N locks. What is the convention that all processes must follow to guarantee that there would be no deadlocks? Explain with an example where two processes need to acquire ***different but intersecting subsets*** of the N locks above.

15. How does the ***Test-and-Set Lock (TSL)*** instruction work? Why can't we use separate LOAD and STORE instructions instead?

16. Explain how you can implement the UP and DOWN operations on a mutex (binary semaphore) using the TSL instruction.

17. How does the **compare-and-set instruction** work? (b) How can you implement a DOWN operation on a mutex (binary semaphore) using a compare-and-set instruction (such as CMPXCHG in x86)?

18. Consider the classical producer-consumer problem. Producers produce items and insert them in a common buffer. Consumers remove items from the common buffer and consume them. In the following skeleton of pseudo-code, ***demonstrate the use of SEMAPHORES and MUTEXES*** to complete the pseudo-code for producer and consumer functions. Your code should have ***no race conditions*** and ***no busy loops***.

You can assume that the following functions are available to you. You shouldn't need anything more than these functions in your pseudo-code.

```
produce_item() produces and returns an item
insert_item(item) inserts the item in the common buffer
remove_item() removes and returns an item at the head of the buffer
consume_item(item) consumes the item supplied
up(&semaphore) and down(&semaphore) have their usual meanings
```

==========Pseudo-code Skeleton ==========

```
#define N 100                   /* Number of slots in the buffer */
typedef int semaphore;          /* semaphores are a special kind of counter */
semaphore mutex = (initialize this);      /* figure out the role of mutex */
semaphore empty = (initialize this);      /* figure out the role of empty sem
*/
semaphore full = (initialize this);       /* figure out the role of full sem
*/

void producer(void)
{
      /* complete this function */
}

void consumer(void)
{
      /* complete this function too */
}
```

==========

19. Consider the classical producer-consumer problem. Producers produce items and insert them in a common buffer. Consumers remove items from the common buffer and consume them. Complete the following skeleton pseudo-code to explain how you can solve the producer-consumer problem using **a monitor and condition variables.**

```
procedure Producer
begin
        /* complete this procedure */
end

procedure Consumer
begin
        /* complete this procedure */
end

monitor ProducerConsumer
        condition /* declare the condition variables you need */
        integer /* declare any other variables you need */

        procedure  insert(item)
        begin
                /* complete this procedure */
        end

        procedure item *remove()
        begin
                /* complete this procedure */
        end
end monitor
```

20. Consider the "events vs threads" argument in the context of monilithic operating system kernels (like Linux or Windows). (a) Which model do these operating systems primarily use -- events or threads? Why? (b) Let's say you that have to design an operating system that uses the opposite model to what you just answered in (a). What would be the major design changes you would make to the kernel in terms of CPU scheduling, memory management, and I/O processing subsystems?

21. What are the tradeoffs in using semaphores versus monitors with condition variables?

22. You are given a function f() in the Linux kernel that constitutes a **critical section**, i.e. no two parts of the kernel should execute f() concurrently. Assume that when the function f() is invoked anywhere in kernel, you call it using the following convention.

        Do some form of locking;

        Invoke function f()

        Do some form of unlocking.

    Explain what type of locking/unlocking mechanism would you choose under each of the following situations and justify your answer:

a. Function f() executes for a very **short** time. It can be called concurrently from two or more threads within the kernel (meaning either processes or conventional threads currently in the kernel context, such as within a system call), but **NEVER** from the within an interrupt context. (Interrupt context refers to the code that is executed immediately as a result of a hardware interrupt to the kernel, i.e. interrupt service routine, and also to the code that executes immediately following an ISR, but just before resuming the interrupted thread.)
b. Function f() can execute for a very **long** time. Otherwise, just as in the previous case, it can be called concurrently from two or more threads within kernel, but **never** from the within an interrupt context.
c. Function f() executes for a very **short** time. It can be called concurrently from two or more threads within kernel, and **ALSO** from the within an interrupt context.

Justify your answers, keeping in mind that the system can have either just a single-processor or multiple processors. Try to give the best possible locking mechanism, not just something that works. If possible, you can support your answer with real examples from within Linux source code where each of the above types of locking/unlocking approaches are used.

23. Explain how you can implement the WAIT and SIGNAL operations on condition variable using the TSL instruction.
24. In the lecture titled "Why threads are a bad idea" ,
    a) what is the key reason that John Ousterhout discouraged the use of threads for most applications?
    b) What are the drawbacks of thread-based programming that event-driven programming doesn't have?
    c) When should you use threads, then?

25. When would you use a semaphore? When would you use a condition variable?