Practice Questions #1

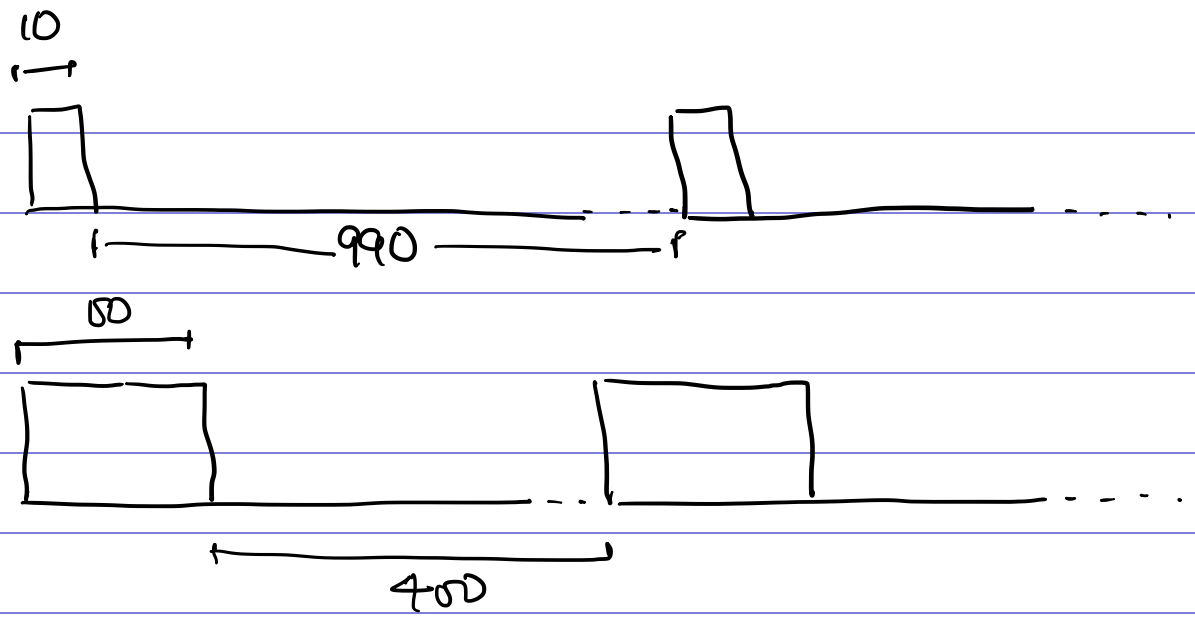
Answers.

(1) System call involves a mode switch from user to kernel. System call also runs procedures provided by the OS developer and not arbitrary functions written by the programmer.

(2) System calls involve the user-to-kernel switch not relevant for interrupts. Interrupt processing takes place even in microcontrollers that do not have an operating system. At interrupt, the interrupt service routine pointed to by the interrupt service vector is executed. In a modern OS, the interrupt service routines and vectors are managed by the kernel.

(3) Mode switching is an essential part of the system call. The purpose of the system call is to run procedures that need higher privileges. So a mode switch is necessary.

(4)



Utilization values we could get

$\frac{10}{500}$, $\frac{0}{500}$, $\frac{100}{500}$

(5) $\frac{110}{500}$, $\frac{10}{500}$, $\frac{100}{500}$

(6) Multi-programming brings multiple programs into memory. So they need to be protected from one another. Otherwise, to ensure correct execution, we need to save the memory contents, which would make multi-programing impractical.

(7) Although the total need is

300 + 200 + 300 = 800 MB
(A) (B) (C)

Each application is going to actually use much less. So we can fit them in memory much smaller than 800 MB.

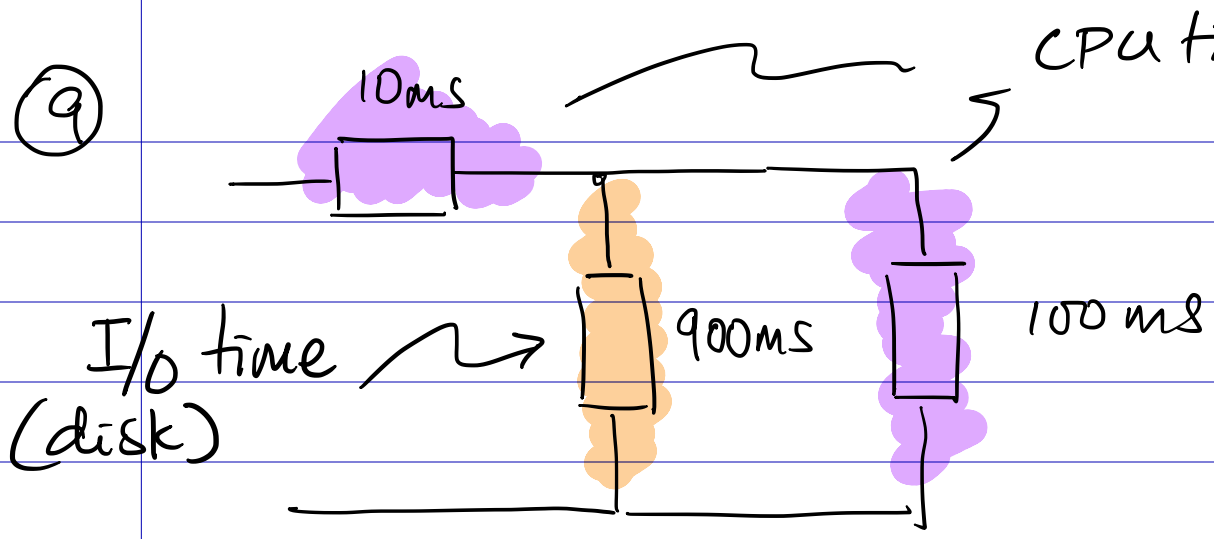
(8) Micro-kernel could be reliable because

(a) μkernel is very small so it can be reliably constructed.

(b) System does not crash unless μkernel crash. The services can crash and restart without affecting the system state as far is up or down

μkernel can make it less reliable because we need complex and evolving messaging interfaces.

(9)



Best scenario. is requests found in the cache.

One request served every 10 + 100 ms

Request rate = $1/0.110$ = 9.09 req/s

Worst case scenario, all requests go to disk.

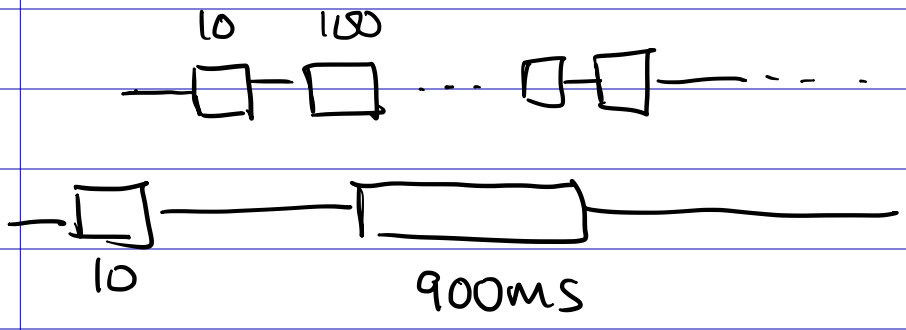
Request rate = $1/0.910$ = 1.09 rq/s

(10) There is only one processor (standard assumption unless specified otherwise).

A compute activity cannot be overlapped with another compute activity.

We can overlap a disk I/O activity with CPU processing.

So the best we could do is to do back-to-back cache access. With multiple (two) threads, we can do the following.



910 ms we can complete 1+ 8 jobs.

Request rate $9 / 0.910 = 9.89$ req/s

(11) 2 threads

(12) 16 processes

(13) yield() {

= thread = get a pointer to the currently active thread

= Save the execution state (e.g., registers) in thread storage.

= lookup the ready threads and pick up the next available one to run

= load the execution state of newly picked thread

}

(14) Process switch does not need a system call.

With preemptive multiprocessing, we have a clock interrupt that triggers a process switch.

It happens in the kernel. A system call is used to go from user to kernel.

Overhead is 500 ns (interrupt servicing)
+ 200 ns (saving)
+ 200 ns (restoring)

900 ns in total.

(15) Message from A
Message from B

or in reverse order.

Outputs from both processes are there in the file

(16) Insert a close(1) before

printf ("Message from A.");

(17)
```
fd = dup(1)          // save stdout
close (1)
  =
  =
  -
if (fork() == 0) {
        close (1)
        dup (fd)     // restore stdout
```

This ensures the proper restoration of stdout.

(18) No

(19) So the program can access services offered by the kernel using the system call interface.

(20) Refer to lectures

(21) Refer to lectures

(22) Speedup is $\frac{600}{170} = 3.52$

(23) $s + p = 600$

$s + p/4 = 170$

$3p/4 = 430$

$p = 573.3$ s

$s = 26.7$ s

$\frac{573.3}{600}$ portion of the app. is parallel

(24) On a 16 core machine

$$S + P/16 = 26.7 + \frac{573.3}{16}$$

$$= 35.8 + 26.7$$

$$= 62.5\ s$$

(25) When you parallelize an application, each machine (processor) is running a smaller instance of the problem.

For example, an array used in computations could be much smaller.

⇒ They fit in the cache.

Better performance.

(26)

$$\text{Speedup} = \frac{1}{S + \frac{(1-S)}{N}}$$

S → Serial fraction of the application.

Lets say the overhead is O expressed as a fraction of the overall runtime.

$$= \frac{1}{S + \frac{(1-S)}{n} + O}$$

(27) The application developer was using a user-level threading library

(28) Faster context switches

Can create very large number of threads

(29) Does not block with blocking system calls

Can leverage multiple processing cores

(30) Large-scale simulations.

Event-driven frameworks that want to be lightweight.