Nathan Pointer

4)
a:Even in a perfect world if we gave all processing power to one process at a time it would make some realtime applications like playing music cumbersome or impossible to write. Supposing that we did do this however I would prioritize smallest non consecutive origin work requests because this would encourage applications to small work requests and would allow multiple applications to do work.

b: perhaps applications could take turns inhabiting memory. Enactive applications could be pushed out to a nearby cache and cycled back in on their turn.

c: A user should only request disk space as necessary otherwise users might have to ‘steal’ storage capacity from other users to accomplish their tasks.

5)
a) If applications are allowed only to work within their own memory ‘sandboxes’ then we have no problem. This might require cooperation with the hardware to insure programs can only use os abstractions to interface with hardware.
b) this requires that user operating system has permissions in its data api. Hardware support would probably make things more secure as it could maintain order in a larger variety of contexts.
c) I imagine hardware design would be very important here to avoid allowing programs from the network compromise the system in the first place, for example if networks had a direct line into the state of the machine there would not be much you could do to avoid being compromised. software could be used to flag messages labeled as spam and prevent their transmission in the first place.

6)
I’m going to say **D,** all of the above as long as an acceptable level of security could be maintained because having multiple transmission methods allows application designers to choose the abstraction that best fits their use case. Transferring stuff via the file system seems a little bit hacky but potentially could have low overhead and be very portable. Messaging would be ideal for event based systems. Shared memory could be very fast and allow for complex application integration. If I had to chose one I would go with messaging.

---

1) Because the kernel process may need access to the application state, if the stack pointer did not move how could the state be saved to the kernel's stack?

7)

a: examples of trapping back into user space could include a return the result of a system call or the transfer of control associated with starting a new process
b: it would copy back any state of the user process, copy any results from system calls that were run and move the program and stack pointers

10) Kernel

12)

```
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>

int add(a,b){
	return(a+b);
}

float tinterval(struct timeval t0, struct timeval t1, int n ){
	return ((t1.tv_sec*1000000+t1.tv_usec)*1000 -
(t0.tv_sec*1000000+t0.tv_usec)*1000)/(n*1.0);
}

int main(){
	long n=10000000;
	int j;
	struct timeval t0, t1;

	gettimeofday(&t0, 0);
	for(int i=0;i<n;i++){
		add(2,5);
	}
	gettimeofday(&t1, 0);
	printf("Mean procedure call cost:%f\n", tinterval(t0,t1,n));

	gettimeofday(&t0, 0);
	for(int i=0;i<n;i++){
		j = getpid();
	}
	gettimeofday(&t1, 0);
	printf("Mean system call cost:%f\n", tinterval(t0,t1,n));
	return 0;
}
```

result:
Mean procedure call cost:9.833500
Mean system call cost:10.477800

Calling a system call like this calls two stubs one user one kernel side and there is overhead in switching contexts when working with protected processes such as copying and validating arguments, and copying return values back over to user program

15) when cpu receives interrupt it looks in the special interrupt vector to call the appropriate reaction as defined by the kernel. From there control may be passed back to the kernel so it can call other process etc. If the error is unresolvable the offending process can be terminated.

Q1: Why is the separation of mechanism and policy important and desirable?
separating mechanism and policy allow us to implement multiple policies without having to start from scratch, allows for micro kernel architectures. Because operating systems are so ‘basic’ or ‘general’ this separation is especially important

Q2: Describe the actions taken by a kernel to context-switch between processes.
kernel regains control of the processor through interrupt etc, stashes the state of the process in a process control block that is moved to the kernel stack for later use. Kernel then transfers control to the new process’s pcb, updating the position of the program and stack pointers to begin running the new process.

Q3: In what circumstances is the system-call sequence *fork(); exec()* most appropriate? When is *vfork()* preferable?
fork creates a new copy of the current process so if you need to insure that you don’t make any changes to the parent before exec starts a new program in the current process context then use fork-exec. vfork creates a new process without copying over data so potentially more efficient but also could introduce bugs if done incorrectly.

Q4: Suppose that you are developing a new computer and OS with significant resource constraints. The hardware is only allowed to support either clock interrupts or I/O interrupts, but not both.

- (a) On each IO interrupt OS passes input to appropriate process, scheduling hungry process in a que. I would also request that application designers write applications so they are frequently yielding to OS so that OS can service multiple processes for each IO interrupt. If a process is blocking the queue by taking multiple IO cycles, move it down in the que to allow other process to work.

- (b) clock interrupts would be used to create event loops which would poll I/O buffers to record input and trigger the appropriate processes, calling context switching if processes began to hog processor time. As long as these loops are sufficiently small user should not notice input lag.
- (c) Clocks are cheap and allow a lot of freedom in terms of process scheduling. IO interrupt only systems will have trouble with computationally heavy programs that do not do IO frequently as they will block other processes potentially forever actually until the user bangs the keyboard in frustration.