[redacted]

All questions are of equal value. Most questions have multiple parts. You must answer every part of every question. Read each question CAREFULLY; Make sure you understand EXACTLY what question is being asked and what type of answer is expected, and make sure that your answer clearly and directly responds to the asked question.

Many students lose many points for answering questions other than the one I asked. Misunderstanding a question may be evidence that you have not mastered the underlying concepts. If you are unsure about what a question is asking for, raise your hand and ask. Spend more time thinking and less time writing. Short and clear answers get more credit than long, rambling or vague ones. Write carefully. I do not grade for penmanship, spelling or grammar, but if I cannot read or understand your answer, I can't give you credit for it.

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

SubTotal:

XC

Total:

python2 supports ( ) new OS doesn't support

1: (a) Consider an after-market application, developed and shipped separately from the OS. What could happen if the OS vendor relased a new OS version with a non upwards compatable API (and associated ABI) change?

Since the OS is not upward compatible, it might not support newer or future versions of OS API and ABI and thus it might break the application since it doesn't support newer versions of API.

(b) What would the application developer have to do to deal with this?

The application developer has to make sure that he or she doesn't use API or services of newer version of OS. That is, the developer shouldn't use API or services that is up-to-date and should stick with those old ones.

(c) Explain how/why interface specifications, designed and written independently from the current implementation, might have affected this situation.

If the specification is separate, the user doesn't have to worry about which implementation the API used and thus doesn't have to worry about the compatibility issue. Since the interface is independent and complete, the user can savely use the API.

2: (a) What is a resource contention convoy?

The resource contention convoy is the situation when threads compete for the same resource (for critical section) and thus, when a line will form and when the waiting time is far larger than the in critical section, the line will be forever and throughput drops.

(b) Under what circumstances is one likely to form?

There are cases when it's likely to happen. The first is when some threads compete for the same lock section. In this case, parallelism changes to sequence. Another situation is threads compete for limited resources (such as read from device, memory). Because resources is limited, each threads has to wait synchronously and thus form a convoy.

(c) Suggest two distinct approaches to eliminating (or SIGNIFICANTLY improving) the problem.

① Spread the request over many resources or locks. In other words, try to use fine grained locks so that a service or object doesn't depend only on one resource or lock to reduce contention

② Reduce the spend in critical section. In other words, try to move time consuming operations out of critical section (like I/O operation) or reduce the length of code in critical section. especially procedure call.

3: (a) List two different types of events that might cause a running process to be preempted.

① I/O interrupt

② timer interrupt

(b) List two different types of events that might cause a running process to become blocked.

① I/O operation.

② memory allocation.

4: (a) Identify three key criteria in terms of which mutual exclusion mechanisms should be evaluated.

correctness, fairness, performance.

(b) Evaluate spin-locks against these criteria.

① Spin-locks are correct in multiple-processor or preemption but are incorrect under conflict I/O operations.

② Spin-locks may result in unbounded wait time (starvation) and thus is not fair.

③ Spin-locks ~~wait~~ waste CPU and memory so it's not highly ~~performance~~ performed.

(c) Evaluate interrupt disables against these criteria.

① Interrupt disable is not correct against multiple-processor.

② Interrupt disable is good in fairness if the interrupt is brief.

③ Interrupt disable may delay important operation and one bug might disable the whole system.

(d) Evaluate mutexes against these criteria.

① Mutexes are correct in multiple-processor or preemption but are incorrect against conflict I/O.

② Mutexes may result in unbounded waiting time.

③ The performance of mutexes is generally good.

5: (a) Describe a major capability (not merely memory savings) of DLLs that cannot be achieved w/mere shared libraries.

(run-time)

DLL can load the routine when it's needed but shared library would map library at exec time.

(b) Describe a specific situation where and why this capability would be NECESSARY.

It's necessary when a process doesn't know which routine it needs to call at link-time and it can only be decided by process's behavior or input at run-time. DLL at this time is needed because it doesn't require shared object to be determined at link time and mapped at exec time. Whenever the process needs a routine, run time loader can load routines in DLL.

(c) Briefly describe a significant mechanism that is not required to support shared libraries, but is required to support DLLs ... and very briefly explain why this additional mechanism is necessary.

The mechanism we support system calls. Since we don't know which system call we need to execute before run-time, shared library is not needed. However, with DLL, we can use run-time loader to load the system call we want to have and thus be more flexible and memory efficient.

6: (a) What is the primary advantage of shared memory IPC over message communication?

Shared memory IPC is really fast in terms of data access and OS is not involved in data transfer but message communication is slow and requires OS.

(b) Why does it have this advantage?

Because OS doesn't involve in data transfer and the same memory is mapped into different virtual address spaces.

(c) List TWO advantages of message IPC over shared memory, and why each cannot reasonably be achieved with shared memory IPC.

① Message IPC can be used anywhere on the earth but shared memory IPC can only be used on the same system. It doesn't make sense for shared memory because shared memory doesn't require OS in data transfer and require the mapping of same memory.

② Message IPC has complex flow control and error handling so that when there is a problem, it's safer and more reliable. In other side, OS doesn't involve in shared memory IPC and cannot give protection.

7: (a) list two pieces of information that a variable partition free-list must keep track of that might not be needed with fixed partition memory allocation.

① The length of chunk

② Where is the next chunk on the free-list.

(b) list two operations that a variable-partition free list has to be designed to enable/optimize (zero points for "allocate and free").

① How to allocate free memory to prevent external fragmentation. Since there are many free memory, we want to find a way that we can quickly find desired free memory and minimize external

② Coalesce: Since memory are limited and there are external fragmentation, we need a way to combine free memory to form larger one or even move processes to re-compact and defragmentation and have free memory

(c) list an additional piece of information we might want to maintain in the free list descriptors to detect or prevent common errors, and briefly explain how the information would be maintained, and how it would be used to detect or prevent a problem.

We can add guard zone to prevent and detect buffer over flow. We would add two guard zones, one at the beginning of the chunk, one at the end of the chunk. We can associate special values with guard zones or even special access. When something ~~tries to~~ causes buffer over flow, it would change the special value we associate with the guard zones or it might not have the permission to write or read it and thus allow us to detect (if values change or access denied) and prevent.

8: Consider a server front-end that receives requests from the network, creates data structures to describe each request, and then queues them for a dozen server-back-end threads that do the real work. Sketch out server and worker-thread algorithms that use semaphores to distribute/await incoming requests, and protect the critical sections in queue updates.

~~The server could use first FIFO algorithm to distribute the requests.~~

The server could use dynamic multi-queue scheduling to distribute different requests. It's helpful so that we can give each one it's best time slice.

The ~~free~~ thread should lock the queue when someone reads it (nobody can write) but when it's writing, nobody can read.

9: Briefly list the sequence of (hint: 8-10) operations that happens, in a demand-paging system, from the page-fault (for a page not yet in main memory) through the (final, successful) resumption of execution.

(a) describe the hardware and low level fault handling.

~~- CPU tries to access not present page and generate a trap.~~

~~use~~

~~- the vector table to find~~

- CPU tries to access a not present page and generate a trap.
- ~~the~~ uses vector table to find the PS/PC of the page fault handler.
- load the PS, pushed PC/PS of process on stack, load PC.
- ~~1st level handler saves registers and go to find it's a see page fault and forward to 2nd level handler.~~

(b) describe the software lookup, selection, I/O, updates.

- 1st level handler saves registers (on stack) and find it's a page fault and go to 2nd level handler
- 2nd level handler find where that page is
- Allocate page free frame
- Block process and read in page
- Update the page table to point to page

(c) describe the return/resumption process.

- return to 1st level handler, restore registers
- Back up usermode and retry the fault instruction
- Unblock process

10: (a) Why is each Linux Condition Variable pared with a mutex? Briefly escribe the race condition that is being managed.

It's used to prevent the case that wake up is called before sleep and then a thread goes to sleep that will never be woken up. Since there is no protection of the lock, when the wait function tries to check the variable and right after that when this thread is about to put itself to sleep, it's preempted and another thread called signal to wake up. After that, preemption happens again and the thread puts itself to sleep.

(b) Write snipppets of signal and wait code, illustruatrating correct use of the condition variable to await a condition.

Note that lock is a mutex initialized by pthread. value is a variable initialized to be 0.

```
void wait() {
    pthread_lock(&lock);
    while (value)
        pthread_cond_wait(&c, &lock);
    value = 1;
    pthread_unlock(&lock);
}
```

```
void signal() {
    pthread_lock(&lock);
    value = 0;
    pthread_cond_signal(&c);
    pthread_unlock(&lock);
}
```

pthread_mutex. pthread_cond

(c) What will the operating system do with the mutex, during which system call(s)?

During pthread_cond_wait, the OS will check that (require) the mutex is hold and after that it will release the mutex. Just before the thread is woken up, the OS will make sure that the thread holds the mutex back.

(d) Could we do this for ourselves? If so, how? If not, why not?

~~We couldn't do this for~~
~~we could do this, Since after we~~

We couldn't do it because if we separate them into different operations, preemptions can happen at any time and race condition would occur.

XC: (a) Heap allocation is much more complex than stack allocation. What key capability do we gain by using heap allocation functions like malloc(3) rather than stack allocation?

- The allocation is managed by user-mode library
- We need explicit allocation rather than OS takes care of stack
- We need to explicit free element so we control its life rather than OS pop it
- Data segment grow is controlled by system call.

(b) Heap allocation is much more complex than direct data segment extension and contraction with sbrk(2). Ignoring the higher cost of system calls (vs subroutine calls), what key capability do we gain by using heap allocation functions like malloc(3) rather than sbrk(2)?

- We have a free list that we can maintain
- We need to explicit free element so we control their lives
- Not restricted by OS
- Have to delete resources ~~our~~ ourselves.

(c) It was briefly mentioned that mmap(2) could be used as an alternative to sbrk(2) to increase the usable data size in a process' virtual address space. What practical benefit/ability might we gain by using mmap(2) rather than sbrk(2) to augment the malloc arena?

By using mmap, we can map memory into our address space and there is only one copy of it so it's fast. We don't have to use sbrk to increase data segment and maintain a free list so less external fragmentation and we have less system call.