


<p><i>What is the difference between virtual and physical memory addresses? Which (if any) is the application programmer able to manipulate?</i></p> <p>1</p>	<p><i>How does a system call differ from an application method/function call?</i></p> <p>2</p>
<p><i>A is a background process, typically owned by root. Examples abound: page, cron, logging...</i></p> <p>3</p>	<p><i>If a page is pinned, what effect does this have (if any) on the page eviction policy? Give two different reasons for pinning a page.</i></p> <p>4</p>
<p><i>What are the three main states a process can be in?</i></p> <p>5</p>	<p><i>How can a state change?</i></p> <p>6</p>
<p><i>In a virtual memory system, under what circumstances might a page fault occur?</i></p> <p>7</p>	<p><i>What does the operating system need to do to classify and process a page fault?</i></p> <p>8</p>

<p><i>It changes mode to enter privileged code, having a fixed set of entry addresses in protected space. It is normally slower than a simple function call.</i></p> <p>2</p>	<p><i>Physical memory addresses refers to a specific location in memory, whereas virtual memory is a meaningless address that needs to be mapped to a physical address using a page table. A programmer will manipulate the virtual address space.</i></p> <p>1</p>
<p><i>A pinned page is not considered for page eviction; it is retained in RAM. A page may be pinned if the code or data it contains is likely to be wanted rapidly and unpredictably – e.g. interrupt handlers. If the page is currently subject to a DMA transfer it must be kept in RAM even if that process is not running in software.</i></p> <p>4</p>	<p><i>A daemon is a background process, typically owned by root. Examples abound: page, cron, logging...</i></p> <p>3</p>
<p><i>- Running: can become stuck (e.g. on I/O) and change to blocked; can be preempted and change to ready.</i></p> <p><i>- Ready: will be changed to running if chosen by the scheduler.</i></p> <p><i>- Blocked: will be changed to ready (or, possibly, running) when unblocked (e.g. by a system call or interrupt).</i></p> <p><i>Ways in which states can change.</i></p> <p>6</p>	<p><i>Ready, running and blocked.</i></p> <p>5</p>
<p><i>Identify the faulting address. Check validity against the process' segment definitions; permissions can be obtained and checked. If the address or the operation was illegal, a segmentation fault is indicated and the process is terminated. Else, if the operation was legitimate and the page is present but the page permissions have been artificially reduced, increase the permissions and return to the faulting operation (a minor page fault). Else, if the page is absent, set up a DMA transfer to fetch the faulting page from the disk.</i></p> <p>8</p>	<p><i>- an illegal virtual address (e.g. corrupted pointer)</i></p> <p><i>- a legal virtual address with the page not present in memory</i></p> <p><i>- a privilege violation</i></p> <p><i>- a legitimate page which has had its privileges temporarily reduced (for monitoring purposes)</i></p> <p><i>Reasons which may cause a page fault.</i></p> <p>7</p>

<p><i>Give an example of a hard real-time computer system.</i></p> <p>9</p>	<p><i>What is the basic difference in process scheduling in a hard real-time computer compared to a Windows or Unix system? What is the role of the process' priority in each case?</i></p> <p>10</p>
<p><i>A real-time system has three processes, at high, medium and low priority. The high and the low priority processes share a data structure which is protected by a mutex. Describe how it might be possible to reach a circumstance where the medium priority process is delaying the running of the high-priority process.</i></p> <p>11</p>	<p><i>What is the main difference between a process and a thread? In what way(s) are they similar?</i></p> <p>12</p>
<p><i>Why is thread-switching within a process usually much quicker than process-switching? Outline all the differences which you believe may have a significant impact on the time penalty for switching processes in a system using virtual memory and note which (if any) are unnecessary when thread switching.</i></p> <p>13</p>	<p><i>What implications are there for thread and process scheduling if the computer has multiple processors (using the same physical memory) rather than a single CPU?</i></p> <p>14</p>
<p><i>A is one which must be executed atomically.</i></p> <p>15</p>	<p><i>A critical section of code is used by a single system call. What are the simplest ways of guaranteeing its correct operation in a single-tasking uniprocessor system?</i></p> <p>16</p>

<p><i>A real-time system schedules to minimise latency, particularly preferring high-priority processes. It can assume that every process will block reasonable quickly and the processor(s) will spend some time in enforced idleness. Timeslicing is counterproductive and may not be used, or be confined to processes of equal priority. An interactive scheduler tries to balance general throughput; it will assume that any process may be running indefinitely and will timeslice running processes. All processes will progress: low-priority processes will receive a smaller share of (rather than no) processor time.</i></p> <p>10</p>	<p><i>Any decent example will do: flight control, in-car control system, Segway balancing system etc.</i></p> <p>9</p>
<p><i>The context which is private to a process includes the address space and the resources it is using. Threads within a process share access to these resources. Both processes and threads are independently schedulable code units. Each has some private data such as register values and a stack.</i></p> <p>12</p>	<ol style="list-style-type: none"> <i>1) Low priority process wins the mutex</i> <i>2) High priority process preempts</i> <i>3) Medium priority process becomes ready</i> <i>4) High priority process blocks on mutex</i> <i>5) Medium priority process runs, preventing low priority process from releasing the mutex.</i> <p><i>This is the classic priority inversion.</i></p> <p>11</p>
<p><i>For processes, really not much as each process has an independent context. For threads, the virtual address space is shared but will be managed independently on each processor. Threads can only be switched cheaply if they use the same processor, otherwise they will be similar to processes. A kernel-based thread scheduler may take this into account.</i></p> <p>14</p>	<p><i>Switching the processor context (registers) is common to both and takes a noticeable time. The major difference is the change of the memory view in process switching. Switching page tables is quite quick but the consequence of changing the virtual memory map is that the TLB and the (virtual) L1 cache(s) will need to be flushed. There is a penalty in writing back the dirty lines from the cache, plus an ongoing performance impact until the cache(s) and TLB have refilled for the new process.</i></p> <p>13</p>
<p><i>In a single-tasking system there can be no interruptions so there is no need for anything other than the code itself.</i></p> <p>16</p>	<p><i>A critical section is one which must be executed atomically.</i></p> <p>15</p>

<p><i>A critical section of code is used by a single system call. What are the simplest ways of guaranteeing its correct operation in a multi-tasking uniprocessor system?</i></p> <p>17</p>	<p><i>A critical section of code is used by a single system call. What are the simplest ways of guaranteeing its correct operation in a multi-tasking, shared memory multiprocessor system?</i></p> <p>18</p>
<p> is the keeping of copies of a subset of available data in a local, fast store. If the data is wanted in again it is available rapidly.</p> <p>19</p>	<p><i>What has happened if part of a cache is "dirty"?</i></p> <p>20</p>
<p><i>Give some examples of where the principle of caching is applied in computing, and how the kernel is involved (if it is).</i></p> <p>21</p>	<p><i>In the general context of caching, what is "LRU" and why is it generally effective?</i></p> <p>22</p>

<p><i>In a multi-processor system interference from other processors must be prevented; some form of lock (e.g. a "mutex") will be needed.]</i></p> <p>18</p>	<p><i>In a multi-tasking system, context switching needs to be avoided; this can be done by disabling interrupts over the critical section.</i></p> <p>17</p>
<p><i>There has been a write operation and the cache area is inconsistent with the original copy; it will need saving, later.</i></p> <p>20</p>	<p><i>Caching is the keeping of copies of a subset of available data in a local, fast store. If the data is wanted in again it is available rapidly.</i></p> <p>19</p>
<p><i>LRU is "Least Recently Used" - an algorithm which can be used to evict part of a cache contents. It selects the part which has not been used in the longest time. It works because, statistically, data which haven't been used for some time are typically not going to be used in the immediate future either.</i></p> <p>22</p>	<ul style="list-style-type: none"> - The memory data cache hierarchy. The kernel is responsible for managing cachability of different memory areas and flushing the virtual cache at context-switch time. - Virtual memory. The kernel is responsible for trapping cache misses, choosing pages for eviction, setting up the refill transfers, etc. - TLB. The kernel needs to flush the TLB on a context switch. - Web cache. Not O.S. related. <p><i>Examples of caching</i></p> <p>21</p>