(1) [3] Released: CPU, Held: memory [and some other resources]

1½ points each.

Many suggested that “no resources are released:” if so, then what’s the point in blocking the process [review the reasons for introducing the states ‘blocked’ and ‘blocked and swapped’]

(2) [3] Essentially because once the PC is restored the next execution cycle will resume the suspended process: this means that the entire context that needs to be restored must have been restored. Also, the very act of doing the restoration will involve executing instructions, hence the use of the PC.

2 points.

Many answers only suggested that the PC was needed to point to the next instruction in order for the process to be resumed; this is true, but it doesn’t explain why that all important PC value needs to be the last value restored.

Several answers featured imprecise statements about the PC, saying it contains the next instruction: it does not. It contains the address of the next instruction.

(3) [5] First part: 3 points

\[
\frac{1 \text{ Kb}}{\text{sector}} \times \frac{20 \text{ sectors}}{\text{track}} \times \frac{100 \text{ tracks}}{\text{surface}} \times \frac{2 \text{ surfaces}}{\text{platter}} \times 3 \text{ platters} = 12000 \text{ Kb}
\]

Second part: 2 points

Reading one entire track takes one entire revolution of the disk. The disk spins at 7200 RPM, or, 120 revolutions per second, so the time for reading one track is \( \frac{1}{120} \) seconds.

Note that keeping units in your equations helps keeps things straight. Some of the answers for part 2 were interesting, from a few microseconds, to 416 minutes: the mistakes that led to these erroneous computations would have been spotted immediately if units had been tracked. For example, the one answer of 24 minutes was computed as \( \frac{7200}{3 \times 100} \) ... the answer was reportedly in minutes, but what was it really?

\[
\frac{7200 \text{ revolutions}}{\text{minute}} = \frac{24 \text{ revolutions}}{\text{minutes}} \times \frac{\text{disks}}{\text{platters}} \times \frac{\text{tracks}}{\text{surface}} \times \frac{\text{surfaces}}{\text{platters}} \times \frac{\text{platters}}{\text{disk}} \times \frac{\text{tracks}}{\text{surface}}
\]

By seeing that the units don’t come out making sense, you have to infer that the calculation is wrong. By arranging for the computation to produce the correct units, you can often deduce the calculation even if at first you don’t see how to.
(5) [3] The maximum amount of wasted space occurs when only one byte is put on a page: thus, if page size is $2^p$ then the upper bound on space wasted on a page due to fragmentation is $2^p - 1$.

The lower bound is either 1 or 0, depending on one’s interpretation. 0 means that the page is full, no space is wasted (and hence there is no fragmentation). 1 would mean that all bytes on the page except one are used, with only 1 byte lost to fragmentation. Both 0 and 1 were accepted.

2 points for the maximum, 1 point for the minimum.

(6) [12] Three operations, 4 points each.

initialize(s,n): s.count ← n, n ≥ 0;

wait(s): if (--s.count < 0) suspend process on s.queue, else return;

signal(s): if (++s.count <= 0) wake up some process on s.queue;

The variant form presented in SG&G was also accepted.

(7) [2] Yes, this can happen: each process resulting from the fork() continues executing the same code unless action is taken to alter that behaviour. If the programmer simply lets both the parent and the child keep running as they are, and does a read from a keyboard, then there are two separate threads (in separate processes) each trying to read from the keyboard.

2 points.

(8) [3] A frame is a partition of physical, real, memory. A page is a logical partition of a process (i.e., as distinct from real memory). Pages are loaded into frames.

3 points; imprecisions that might have cost some points were not making clear your understanding that frames are related to real memory and pages aren't.

(9) [7] 3 points (1 + 2): preferable to use an unmodified page because it does not have first to be written out to disk to be saved, further delaying page replacement

2 points: page buffering

2 points: because with page buffering, the frame whose page is being replaced is moved from one list to another, i.e., and so remains available in memory for some time before being written out and moved to the list of available frames. This means that irrespective of being modified or not, the page remains available and can be ‘re-loaded’ quickly.
Many students replied with LRU, or LFU, or any of a variety of other page replacement policies...but **all** of them suffer the same problem: upon encountering a modified frame to replace, they all require the process to wait first for the modified page to be saved, then the sought page to be loaded.

A serious misconception shown by several students was that an unmodified page must be a not recently used page: use and modification are unrelated. Think of pages holding instruction code for a process...such pages are never modified, yet are clearly often and, at any moment in the life of a program, recently used [the instruction-bearing pages near the current location, anyway].

(10) [5] A critical section is code that must be executed only by one thread/process at a time else there is the risk of race conditions and associated unpredicted behaviour.

5 points.

(11) [10] 3 points: the limitation is that not all signals can be caught and indicated by this process, notably signal 9, which will cause the process to terminate.

7 points: essential elements to show: a generic signal handler, some main function that first arms all handlers then waits to receive a signal. It was not necessary to be fastidious about how the actual signal number is known in the signal handler.

```c
main()
{
    /* at some point early in main: */
    void my_hdl();
    signal(*,my_hdl); /* make my_hdl handler for all signals */

    /* then, later, the main part: */
    while (1) loop; /* do something (or nothing) waiting for signals */
}

void my_hdl()
{
    printf("got signal %d\n",signal_number);
}
```

Many students took elaborate steps to prevent arming a signal handler for signal 9; there is no need to do this since the handler for signal 9 cannot be changed in any case. A common misconception was that signal 9 is called SIGTERM...it is, in fact, SIGKILL (SIGTERM is, usually, signal 15).
(12) [13] 1 point for the correct number of page faults, 9. If there was an error in the grid and the resulting page fault count was consistent with the error, then it was accepted as correct (but elements of the grid were penalized).

12 points for the grid: there are 36 squares and 3 separate things had to be reported for each one: at one point each, that’s 108 points. This number, divided by 9, gives the 12 points for this part. Point deductions were rounded up to the nearest integer multiple of 0.5; no number of deductions resulted in less than 0.5 points.

The solved grid below shows, for each column, the values of the clock hand and use bit at the end of that page reference:

<table>
<thead>
<tr>
<th>Clock</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>6</th>
<th>3</th>
<th>3</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6*</td>
<td>6*</td>
<td>&gt;6*</td>
<td>4*</td>
<td>4*</td>
<td>4*</td>
<td>4</td>
<td>&gt;4</td>
<td>3*</td>
<td>3*</td>
<td>3</td>
<td>&gt;3</td>
</tr>
<tr>
<td>1</td>
<td>&gt;</td>
<td>5*</td>
<td>5*</td>
<td>&gt;5*</td>
<td>&gt;5*</td>
<td>2*</td>
<td>2*</td>
<td>&gt;2*</td>
<td>&gt;2*</td>
<td>5*</td>
<td>5*</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>&gt;</td>
<td>3*</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>&gt;3</td>
<td>6*</td>
<td>6*</td>
<td>6*</td>
<td>&gt;6</td>
<td>2*</td>
<td>---</td>
</tr>
<tr>
<td>PF</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

One of the commonest misconceptions was that the clock hand is moved with every access to memory: it moves only when clock is searching for a page to replace; otherwise it stays pointing where it left off after the last page was loaded (i.e., at the frame after the one that has just been used to load the new page). The goal is that the clock hand is supposed to point at the place where the next search for a page to replace will begin.

(13) [10] 1 point each.

(i) F (vi) F
(ii) T (vii) F
(iii) T (viii) F
(iv) F (ix) T
(v) F (x) T

(14) [2] 1 point each: max size is Δ, min is 1.

(15) [4]

- asymmetric: one cpu, the master, is ‘privileged,’ does all system calls/kernel action; all cpus run unprivileged user work [common misconception: that the master cpu only runs kernel code; being the only one cpu that can do this, it may spend much of its time doing that, but it does, when not doing system work, regular user-level work just as all the other cpus are doing]

- symmetric: any cpu can do any system call
• symmetric is more robust because failure of any one cpu has minimal impact; loss of the master cpu in an asymmetric multiprocessors is fatal

• asymmetric is easier to build an OS for, since it is a ‘classic' single cpu kernel model; symmetric is much more complex.

1 point each

(16) [2] exit does not return a value and n ceases to exist when the process containing it exits;

2 points: right or wrong

(17) [4] portable: hardware-dependent code needs re-writing, but this is small and contained entirely within the microkernel, so only this part needs work; other services run as user-level processes so can simply be re-compiled

flexible: services can easily be added or removed at the user-level in order to tailor the micro-kernel-based OS to particular requirements.

2 points each

Saying simply that portability was enhanced because a microkernel is small was not enough: it is specifically that the part that needs true rewriting is the hardware dependent part and that part is small. Several people provided answers that simply provided a description of what portability is, without anything specific in the answer as to why micro-kernels were particularly good for portability.

(18) [4] ULT over KLT: speed of thread-switching; scheduling and thread-switching can be handled entirely within the process at user-level, without kernel intervention;

KLT over ULT: processes (all threads) don’t have to block just because one thread in task make system call and blocks

2 points each.

Several answers suggested only that ULTs were better because they were “faster.” Faster than what? they run on the same cpu; they generally execute no faster nor slower than a KLT. What is faster is switching between threads, mediated at the user level (i.e., with no kernel intervention), but that’s not being just “faster.”

(19) [12] Four conditions: 1½ points each: (1) mutual exclusion, (2) hold & wait, (3) no preemption, (4) circular wait.
Prevention is assured by breaking any one of the above four. Removing mutual exclusion is rejected as infeasible, leaving any two from:

1. disallow hold & wait: a process cannot hold resources and ask for more so either it asks for all of its resources at the start and starts when they can all be granted, or, must first release all of its resources before being granted the one requested; efficiency problem: process may not need all of its resources at the same time, so has resources it doesn’t need and isn’t using for part of its execution, i.e., resources other processes might need and have been able to use.

2. allow preemption: can take resources away from a process to satisfy a resource request; efficiency: not all resource types can reasonably be preempted, can needlessly slow a process’ progress

3. prevent circular wait: require resources to request their resources following a specific ordering; efficiency: same as (1)

3 points each (2 for the method, 1 for efficiency considerations).

A common mistake in answering this question was to provide solutions that were not prevention, but rather avoidance (process initiation denial, resource allocation denial).

(20) [4]
Disadvantage: a process may wait in the ‘best fit’ queue waiting to be loaded into memory while other, larger partitions are unused, so processes could be running there.

Alternative: use one single queue for all partitions: head-of-queue process sent to any partition big enough to hold it.

2 points each.

Some answers suggested the solution was to abandon fixed different-sized partitioning and go for variable sized; this was an acceptable suggestion. Others suggested that the use of a single queue would decrease fragmentation: actually, it can increase it (what happens when, using a single queue model, we allow a 4 Kb process to run in a 100 Kb partition? 96 Kb of space lost to internal fragmentation).

(21) [6]
2 points: page table: holds correspondence between a page of a process and the frame of real memory holding that page.

4 points: usage: isolate page number from virtual address, look up that page number in page table (e.g., using page number as index into table) to determine what frame holds the page (if it is resident; if not, a page fault occurs).

(22) [4]
2 points: more efficient memory utilization because the memory image of the process includes only code that is actually used since linking external objects only occurs as they are first invoked.

2 points: penalty for this improved memory utilization, process ‘stalls’ each first time a
function is invoked and needs to be linked.

Several suggested that it was either an advantage or disadvantage of dynamic linking at run time that a program would be linked with newest library versions. While possibly an advantage of dynamic linking at run time, it isn't really the explanation for the more efficient usage of memory.