Tully McDonald
220196038

1)

The first generation of computers were extremely bare-bones. They began as vacuum tubes and plug boards operated manually by a team of specially trained operators. These computers were comically large, horrendously slow, and inefficient. They typically occupied an entire room and some of the more advanced computers relied on punch cards for machine language instructions. This generation of computers mainly computed basic maths problems, and there was no real need (nor possibility) for an operating system in software form.

The second generation of computers utilised batch systems and replaced vacuum tubes with transistors. Transistors made computers significantly more efficient, quicker, and most importantly, cheaper. This is also when history began to see operating systems being used for more important aspects. What was particularly important for operating systems is human-readable assembly languages were developed which allowed for more advanced programming written in less time.

The third generation of computers integrated circuits and multi-programming. Computer transistors were further developed and created much smaller to fit on silicon chips, allowing for even greater efficiency, speed, etc. These systems also allowed multiple users to use the same machine at the same time. However, while doing so this introduced tedious problems like deadlocks which involve some very clever hardware and software solutions such as Peterson's solution and Strict Alternation.

The fourth generation of computers has essentially been large scale integration. The software was increasingly growing in capability as hardware was growing in performance by the miniaturisation of transistors onto smaller and smaller chips. Famously, Moore's Law states that the number of transistors on microchips consistently double every two years. As so, we saw incredible improvements over the years in efficiency and usability. Computer hardware was becoming small enough and cheap enough to grow a home-computer market. GUIs were developed which lead to today's trillion-dollar consumer market for personal computers. Microprocessors began to work their way into other markets as well, vehicles, televisions, house appliances, traffic lights. Today, you can hardly find a room without some form of computational hardware. This also evolved into computer networks across large distances, which were eventually united into what is now known as the internet. The internet grew and introduced instant communication across oceans and continents, previously only possible with radio waves and telephone lines. Eventually, social media platforms were born and people became more interconnected than ever before in history. Now nearly the whole of human knowledge is at the fingertips of billions.

Tully McDonald
220196038

2)

The von Neumann bottleneck refers to when the communication bus is unable to support the required communication speed between the CPU and main memory. It's the throughput of the bus that connects the CPU with the main memory which is limited in speed and can cause computers which are ever-growing in speed and performance to never truly utilise their full potential. This is a serious problem, it's like building a bigger car engine but not increasing the diameter of the exhaust pipe to support that amount of throughput.
Particularly recently, as CPU performance and main memory speed has improved and become much faster, there has been a lot of interest in this area. A couple of ways around the von Neumann bottleneck problem is:

- Branch predictor, which refers to predicting the logic result before computation. For example, with an if-else statement, a branch predictor will try to guess which path/branch will be taken. If computers did not have a branch predictor, they would need to wait for a conditional jump instruction to be computed before the following instruction. To mitigate this, a branch predictor will attempt to guess if the conditional jump would be made and then fetch that branch. Further down the line, the logic runs and the computer can see if it guessed correct. If it does not guess correct then it has to fetch the other branch which means there is a bit of a delay because of this.
- Another way to try to combat the von Neumann bottleneck problem is by caching frequently used data close to the processor instead of using the bus continuously for these bits of information. Many processing workloads tend to reuse various bits of data which can clutter up and increase the load on the bus communication system. So even small amounts of cache stored inside (or very close to) the CPU can make a huge difference and free up a lot of time spent waiting on the bus to complete a cycle of data collection.

3)

- Peterson's solution: Sometimes when a process is waiting to go critical, they will be constantly checking (busy wait), which means they are wasting CPU time. To prevent this, we need a way for processes to communicate and be woken once they can go critical. Peterson's solution addresses this by using strict alternation with a particular algorithm that allows dynamic critical time between processes. This is great because it's on the software level, however, it's complex as you add more processes to the equation.
- TSL instruction: Similar to Peterson's solution, in that, this also implement strict alternation on a hardware level. TSL instruction allows for systematic critical region control with as many processes as needed, it does not become more complex with more processes but it's inconvenient being on the hardware level.

4)

a) Yes, this does ensure mutual exclusion between processes 1 and 0 because process 0 is held in the while-loop until process 1 completes its critical section and sets the wait

variable to itself as its last operation. This effectively unlocks the waiting process once the running process has completed.

b) Technically it would work with a larger number of processes. But due to this being strict alternation, it's not a very good option.

5)

a) Ignoring deadlocks is simply that; ignore them, don't worry about them. An example of when this is appropriate is when deadlocks only occur when your machine has been running for a week, but your system is restarted every day.

b) Preventing deadlocks refers to breaking one of the four conditions that are required for deadlocks to occur. For example, breaking the mutual exclusion rule by using a remote service as a resource manager (i.e. printer daemon).

c) Avoiding deadlocks refers to checking if a particular action will result in a deadlock and not doing the action if it will result in a deadlock. For example, a system could use the banker's algorithm if it is aware of the resources at hand, the required resources per process, and the time required by each process.

d) Detecting and recovering from a deadlock refers to ignoring deadlocks until they happen, then perform an action to fix the deadlock. For example, if you have a machine that just has to perform basic non-time-sensitive calculations (doesn't matter how long each one takes), the system could just look for deadlocks and kill one of the processes in that deadlock.

6)

a) (where 1KB = 1024 bytes)
   i) 1023: Page 0, offset 1023
   ii) 9651: Page 9, offset 435
   iii) 11650: Page 11, offset 386
   iv) 84938: Page 82, offset 970

b)
   i) FIFO:
   Frame 4 is next to be replaced because it's currently the oldest (first in).

   ii) Least recently used
   Frame 2 would be replaced next because it's the least recently referenced.

   iii) Clock
   Frame 2 would be replaced next because it's the first in circular order with reference bit 0 (assuming we are not pointing at frame 3 to start with)

7)

a)

12 direct block pointers

Tully McDonald
220196038

4 indirect block pointers
2 double-indirect block pointers
Block size = 4KB
Pointers per block = 4K/4 = 1024

1) Smallest file too big for 12 direct block pointers
   Direct block pointer = 4KB = 4096B
   Answer: 12 * 4096B + 1 = 16385B

2) Smallest file too big for 4 indirect block pointers and 12 direct block pointers
   Indirect block pointer = 4096*1024 = 4194304B
   Answer: 12 * 4096 + 4 * 4194304 + 1 = 16826369B

3) Size of 12 direct block pointers, 4 indirect block pointers, 2 double-indirect block pointers
   Double-indirect block pointer = 1024*1024*4096
   Total storage: (12*4096)+4(1024*4096)+2(1024*1024*4096) = 8606760960B

4) Pointers per block would go from 1024 to 1024^2 (1048576)
   Answer: (12*4096)+4(1024^2*4096)+2(1024^2*1024^2*4096) = ~ 9.007216435x10^15

b) The file system block sizes are natural division points. If you don't align the partitions with these points then the drive is put in an awkward position and loses read/write performance, sometimes by an order of magnitude.

8)

a) One computer takes T*0.78*12 + T*0.22 = 9.58T
   Therefore, one computer takes 9.58 times longer than the cluster

b) One computer takes T*0.78*12 + T*0.13*4 + T*0.09 = 9.97T
   Therefore, one computer takes 9.97 times longer than the cluster

c) One computer takes T*0.26*12 + T*0.52*4 + T*0.22 = 5.42T
   Therefore, one computer takes 5.42 times longer than the cluster

9)

- Detect intrusion: Automatically monitor user activity and flag any abnormal behaviours which appear suspicious.
- Log events: Log user commands, actions, or other useful information for post-event investigation.
- Send alerts: If any suspicious behaviour is detected, notify an admin, so they can investigate further.