1. How do paging and segmentation affect external fragmentation? - Paging eliminates external fragmentation and segmentation only reduces external fragmentation.
2. Which of the following is true about paging and segmentation? - In both segmentation and paging the address space of a process does not have to be contiguous.
3. Which of the following is the most time consuming step in handling a page fault? -Transferring data between disk and memory.
4. What is the difference between the C-Scan and the C-Look disk scheduling algorithms? - C-Scan always reaches the end of the disk, while C-Look does not reach a disk end unless there is a request at that end.
5. Which disk scheduling algorithm is the most likely algorithm to cause starvation? - SSTF, because if requests to locations that are adjacent to the current location keep arriving, an earlier request to a far location may wait indefinitely.
6. What’s the advantage of a File Allocation Table (FAT) over basic linked allocation? - Traversing the links in FAT is faster, because they are more localized. AND - The links in FAT are more cacheable.
7. How does indexed disk allocation compare with contiguous disk allocation? - Contiguous gives faster sequential access but indexed causes less external fragmentation.
8. How does indexed disk allocation compare with linked disk allocation? - Indexed gives faster random access but linked is less complex from implementation point of view.
9. Which of the following is (are) true about internal fragmentation in linked disk allocation? - Increasing the block size increases internal fragmentation. AND - Clustering increases internal fragmentation.
10. Which of the following is true about multilevel-feedback-queue scheduling? - If a process uses its entire time quantum, it is moved to a lower priority level. AND - If a process spends a lot of time in a low-priority level without getting the CPU, it is moved to a higher priority level.
11. Which of the following is **not** true about threads? - Parallel programming using threads can utilize more cores than parallel prog. using processes.
12. How does the page-fault frequency (PFF) technique prevent thrashing? - It takes frames from a process if its page fault rate falls below a certain lower bound. AND - It gives more frames to a process if its page fault rate exceeds a certain upper bound.
13. Which of the following is true about memory frame allocation? - With global frame allocation, the execution time of a process depends on other processes. AND - Global frame allocation does a better job at utilizing memory than local frame allocation.
14. Which of the following is **not** true about open-file tables? - A pointer to the last read location is stored in the system-wide table.
15. Which of the following is **not** true about virtual memory (VM) and physical memory (PM)? VM does not need any hardware support.
16. What’s the relationship between page size and fragmentation? - A larger page size increases internal fragmentation.
17. A logical address space of a process has 512 pages with an 8-KB page size. How many bits are required in the logical address? - 22
18. Which of the following is **not** true about paging and segmentation? FALSE: A single operand can never access more than one page. TRUE: Paging divides memory into equal blocks. Segmentation may divide memory into unequal blocks. A single instruction may access multiple pages. In both segmentation and paging the address space of a process does not have to be contiguous. A process can run even if only a small subset of its pages are leaded in physical memory.
19. Which of the following is true about a page fault? - Handling a page fault always involves transferring data from disk to memory. AND - Page faults occur infrequently due to locality of reference.
20. Which of the following is true about page replacement policies? - Optimal page replacement is theoretically the best but is impossible to implement in practice. AND - With FIFO page replacement, increasing the number of frames may increase page faults.
21. What is the difference between the Scan and the C-Scan disk scheduling algorithms? -Scan services requests when it is moving in both directions, while C-Scan services requests only when it is moving in one direction.
22. Which of the following is true about starvation in disk scheduling? - FCFS can never cause starvation. AND - SSTF is the only algorithm (among the ones we studied) that may cause starvation. AND - Scan doesn’t cause starvation, because the waiting time can be long but will always be bounded.
23. Which of the following is true about the difference between the Round Robin (RR) and the Shortest-Job-First (SJF) scheduling algorithms? - SJF minimizes the average waiting time, but RR does not.
24. What’s the difference between Rate-Monotonic Scheduling (RM) and Earliest-Deadline-First (EDF) Scheduling? - RM uses fixed priorities while EDF dynamically adjusts priorities.
25. How does contiguous disk allocation compare with linked disk allocation? - Contiguous provides faster sequential and random access but causes external fragmentation.
26. If a process uses 1100B of memory in a system with a page size of 512B, what’s the size of internal fragmentation? - 436B
27. How does indexed disk allocation compare with FAT-based linked allocation? - Indexed allocation may waste a substantial amount of disk space if there are many small files.
28. Which of the following is (are) true about the Working Set Size (WSS)? - A larger WSS increases the page fault rate. AND - If the WSSs of all processes exceeds the number of available frames, the system will thrash.
29. Which of the following is true about storage allocation algorithms? - First fit and best fit have been found to give better utilization, on average, than worst fit.
30. How do **processor affinity** and **load balancing** interact in a multi-processor environment? - Improving load balancing may conflict with the processor affinity requirement in some cases.

Q2: (a). Schedule each of the following inputs using the algorithm shown below it. First give the output in the form of a **Gantt chart**, and then compute the **average waiting time**. Show your work.

Input 1:

Process CPU Burst

P1 5 6

P2 2 4

P3 7 9

P4 3 3

Apply Round Robin (with a time quantum of 3 4). Assume that all processes arrive at time 0 but in the order shown in the above table (there are negligibly small differences in arrival times).

Intervals of 4 time quantum.

After First round: P1 needs 2 more, p3 needs 5 more

After Second round: p3 needs 1 more

P1 waited 0 +15 – 4

P2 waited 4

P3 waited 8 + (17-12) + 0

P4 waited 12

Input 2:

Process Arrival Time CPU Burst Priority

P1 0 7 2

P2 2 3 1

P3 4 2 3

P4 6 4 1

Apply Preemptive Priority Scheduling. Assume that a smaller number indicates higher priority. If a tie ever occurs, break it arbitrarily.

Input 3:

Process Arrival Time CPU Burst

P1 0 7

P2 2 3

P3 4 2

P4 5 6

Apply shortest remaining time first

P1 (0-2) > p2 (2-4,5) > p3 (5-7) > p1 (7-12) > p4 (12-18)

Average wait time

P1: 0 + (7-2)

P2: 2-2

P3: 5-4

P4: 12-5

Then divide sum by 4

(b) A major problem with priority scheduling is starvation.

1. Why is priority scheduling more likely than other scheduling algorithms to cause starvation? Describe how starvation happens in priority scheduling.

Starvation happens when high priority processes keep arriving and low priority processes never get a chance to execute.

1. Give one good solution that operating systems use to prevent starvation in priority scheduling. Give a brief description of the solution, not just its name.

Aging is a good solution because the longer the process waits in the queue, the higher the priority of this process becomes so it eventually gets the cpu

(b) Schedule the following two **periodic** processes using **Earliest-Deadline-First Scheduling**. Give the output in the form of a **Gantt chart**. Write a comment justifying the scheduling decision made at the arrival of every new CPU burst. If at some point, both processes have the same deadline, resolve the tie in favor of the process that currently has the CPU. Stop your scheduling as soon as a process misses its deadline or when you reach Time 60, whichever occurs first. Clearly indicate if a process misses its deadline.

P1: p1=30, t1=15, d1=30

P2: p2=20, t2=10, d2=20

Recall that p is the period, t is the length of the CPU burst, and d is the deadline. So, P1 will have CPU bursts of length 15 periodically arriving at times 0, 30, 60, …, and each burst must complete executing before the arrival of the next burst.

Q4: (A) Consider a virtual memory system with 8 pages (0 through 7) and 3 frames (0, 1, 2). Trace the state of the system for the sequence of page accesses shown below using the Least Recently Used (LRU) page replacement algorithm. Show the contents of the three frames after each page request (as done in class), and then give the number of page faults. (5 points)

7 3 5 7 4 5 1 2 5

Number of page faults:

1. A system has a Table-Lookaside Buffer (TLB) with a negligibly small access time compared to the memory access time. Calculate the TLB hit ratio that will keep the Effective Access Time (EAT) within 10% of the ideal EAT (ideal EAT is EAT with no TLB misses). Show your work **clearly**. (4 points)

1. A program has one loop that is executed 500 times. The code inside the loop sequentially accesses a large array that spans 100 pages. Each page has 200 words (array elements). The page replacement policy is LRU. Calculate the page fault rate for the execution of the entire loop in each of the following cases. Show your work **clearly**. (6 points)

1. The process has 120 frames allocated to it,

2. The process has 80 frames allocated to it.

int main(int argc, char \*argv[]){

1 pthread\_t tid[MAX\_THREADS];

2 pthread\_attr\_t attr[MAX\_THREADS];

3 int indices[MAX\_THREADS][3], i, arraySize, min;

// Code for parsing command-line arguments and generating input

4 CalculateIndices(arraySize, gThreadCount, indices);

5 InitSharedVars();

6 sem\_init(&sem\_1, 0, **0**); **X1: 0**

7 sem\_init(&sem\_2, 0, **1**); **X2:1**

8 for(i=0; i<gThreadCount; i++)

9 pthread\_create(tid+i, attr+i, Child, indices[i]);

10 sem\_wait(**X1**);**x3:sem\_1**

11 min = SearchThreadMin();

12 for(i=0; i<gThreadCount; i++)

13 pthread\_cancel(tid[i]);

14 printf(“Minimum is %d\n”, min);

}

void\* Child(void \*param) {

1 int threadNum = ((int\*)param)[0];

2 int start = ((int\*)param)[1];

3 int end = ((int\*)param)[2];

4 gThreadMin[threadNum] = MAX\_RANDOM\_NUMBER + 1;

5 for(int i=start; i<=end; i++) {

6 if(**X2**) {

7 gThreadMin[threadNum] = 0;

8 gThreadDone[threadNum] = true;

9 sem\_post(&sem\_1);

10 pthread\_exit(0);

}

11 if(gData[i] < gThreadMin[threadNum])

12 gThreadMin[threadNum] = gData[i];

}

13 gThreadDone[threadNum] = true;

14 sem\_wait(&sem\_2);

15 gDoneThreadCount++;

16 if(**X3**)

17 sem\_post(&sem\_1);

18 sem\_post(&sem\_2);

X1:sem\_1

X2:gData[i] ==0

X3:gDoneThdCnt == gThdCnt

Calculate the percentages shown below. Assume **pure demand paging** but that the pages of this process fit into physical memory and will not be knocked out of memory by other processes after they have been loaded. Also assume for simplicity that the pages accessed by the different code sections do not overlap with each other. First show the complete formula (don’t ignore anything), and then in doing the calculations, you may neglect insignificant terms.

1. The percentage of page accesses that will result in page faults.

Page faults = (# of page accesses resulting in faults)/(total number of page accesses) = (200 + 100 +6 + 400)/((200) + (100 x 1000) + (6x100,000) +400)) = 0.1 % or 1/1000

1. The percentage of memory accesses that will result in page faults, assuming that the program accesses 100 (using 50) memory locations per page.

Page faults = (# of page accesses resulting in faults)/(total number of page accesses) = (200 + 100 +6 + 400)/(50\*((200) + (100 x 1000) + (6x100,000) +400)) = 1/50,000