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.

1. Modify the above code to solve the following search problem. Given an input array and a set of k keys, check if all the keys are found in the array. The program divides the array into gThreadCount divisions and creates a thread to search each division as in the above code. If all keys are found, the program prints “Found”; otherwise, it prints “Not Found”. Assume that the keys are already stored in a global array of size k named gKeys.

sem\_t chkSem, limitSem, mutexSem;

int gKeys[k];

bool gKeyFound[k];

Parent

Lines 1-5 stay the same

Lines 6, 7 replaced with:

sem\_init(&chkSem, 0, 0);

sem\_init(&limitSem, 0, X);

sem\_init(&mutexSem, 0, 1);

for (int j=0; j<k; j++)

gKeyFound[j] = false;

Lines 8, 9 stay the same

Line 10 replaced with:

bool done = false, allFound;

while (! done) {

sem\_wait(&chkSem);

allFound = true;

for (int j=0; j<k; j++)

if(gKeyFound[j]) {

allFound = false;

break;

}

if (allFound || gDoneThreadCount == gThreadCount)

done = true;

}

Line 11 deleted

Lines 12, 13 stay the same

Lines 14 replaced with

printf(“%s\n”, allFound? “All keys found”: “Some keys not found”);

Child

Lines 1-3 stay the same

Line 4 deleted

Lines 5-18 replaced with the following code:

sem\_wait(&limitSem);

int i, j;

for (i=start; i<=end; i++)

for(j=0; j<k; j++)

if(gData[i] == gKeys[j]) {

**gKeyFound[j] = true;**

sem\_post(&chkSem);

}

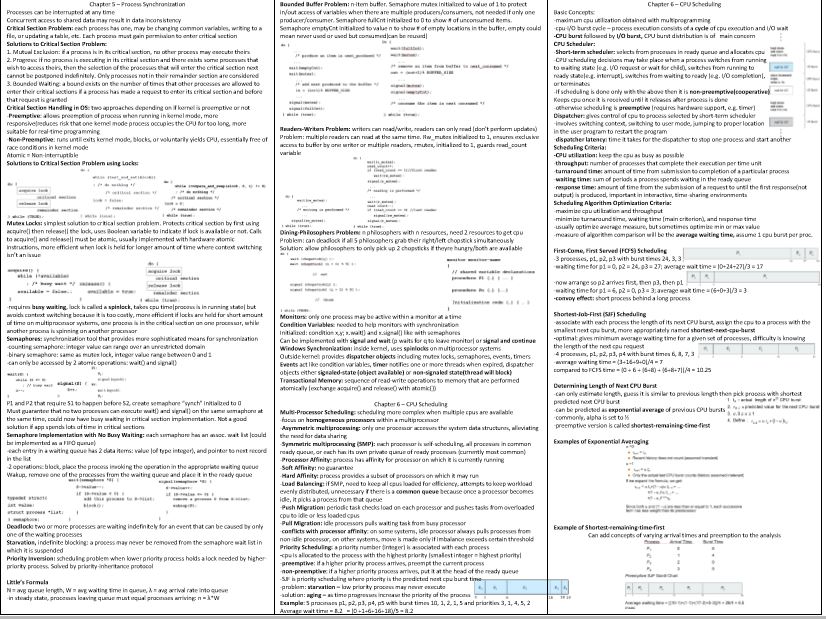
sem\_wait(&mutexSem);

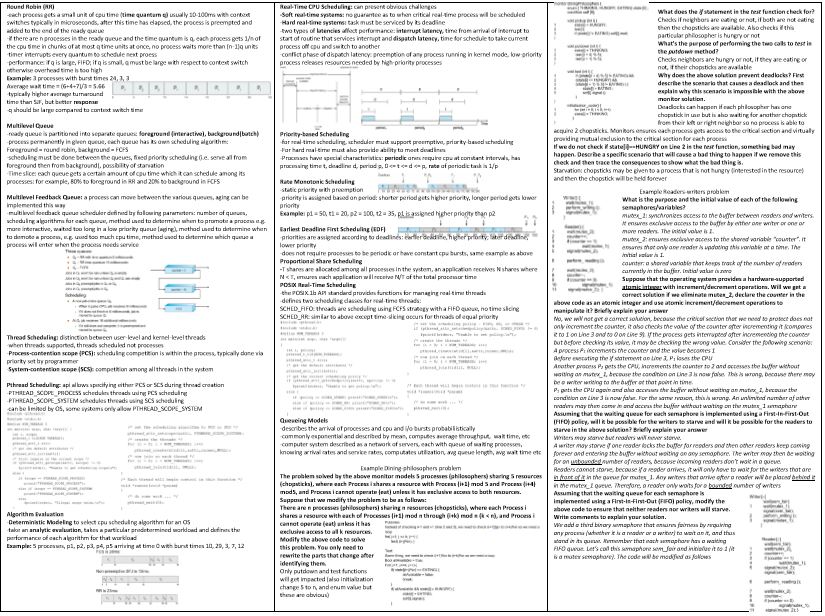
gDoneThreadCount++;

sem\_post(&mutexSem);

**sem\_post(&chkSem);**

sem\_post(&limitSem);





1. 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. Bankers algorithm.

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
2. 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.
3. What does each of the **Available** vector and the **Work** vector represent? - Available reflects the number of instances initially available

Work is initially equal to available, but continue to add to it the allocation of previous processes

1. How many times will the outer (while) loop get executed if the system is in a safe state? Why? - If process is in a safe state, the outer loop will execute n times depending on number of processes. If there are no processes that needs can be met, we are in an unsafe state and only execute < n times.
2. If we remove the clause “Finish[i]==false” from the condition of the if statement on Line 9 and only check the condition “Needi ≤ Work”, will that lead to a false positive (printing “Safe” for a system that is actually unsafe) or a false negative (printing “Unsafe” for a system that is actually safe)? - Finish[i]==false makes sure that the process being worked on has not been done before, otherwise the same process will execute over and over and lead to a false positive
3. Given the following contents for each of the TLB and the page table, compute the physical address for each of the logical addresses below. Assume that the frame (page) size is 4. Given the list of frames that are present in physical memory, indicate for each logical address if that memory access results in a page fault. Also indicate if the access results in a TLB miss.

(6 point)

TLB Page Table Frames in physical memory: 1, 2, 3, 7, 9

|  |  |  |  |
| --- | --- | --- | --- |
| Index | Value | Index | Value |
| 3 | 6 | 0 | 5 |
| 0 | 5 | 1 | 3 |
|  |  | 2 | 7 (using 1) |
|  |  | 3 | 6 |

|  |  |  |  |
| --- | --- | --- | --- |
| Logical Address | Physical Address Calculation | TLB miss? | Page Fault? |
| <0, 2>  Using <1,5>  First digit is page#, second digit is offset(displacement) | 3x8+5 = 29 | Tlb miss | no |
| <2, 1>  Using <3,2> | 6x8+2 = 50 | Tlb hit | yes |

1. It was stated in class that page faults do not occur frequently due to **locality of reference**. For a numerical example, consider the program below with a doubly nested loop:

Code Section A

for (i=0; i<1000; i++) {

Code Section B

for (j=0; j<100; j++)

Code Section C

}

Code Section D

Given the following page accesses (code and data) for each code section,

Code Section A accesses 200 pages

Code Section B accesses 100 pages

Code Section C accesses 6 pages

Code Section D accesses 400 pages

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. (3 points)

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. (2 points)

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