Section 1: Process

Q1: Which of the following components is responsible for loading the initial value in the program counter for an application program before it starts running:

- *Compiler*
- *Linker*
- *Loader*
- *Boot module or boot ROM*

Solution: The loader. The linker groups several modules and resolves external references, but the linker really does not perform the
actual loading of the program itself (except for dynamic linking, which we will study later in the course).

Q2: Given the following program that uses three memory segments in an address space as described in class (code segment, data segment and stack segment):

```
char a[100];
main(int argc, char ** argv)
{
   int d;
   static double b;
   char * s = "boo", * p;

   p = malloc(300);
   return 0;
}
```

Identify the segment in which each variable resides and indicate if the variable is private to a thread or shared across threads. Be careful.

The array a, the static variable b, and the string constant "boo" are all in the data segment and are shared across threads. The arguments argc and argv are in the stack segment and are private to the thread. The automatic variables d, s, and p are also in the stack segment and are private to the thread.

Note that the variable p itself is in the stack segment (private), but the object it points to is in the data segment which is a shared region of an address space(that's why the be careful warning). The contents of s consist of the address of the string "boo" which happens to be in the data segment (shared).

Q3: A hardware designer argues that there are enough transistors on the chip to provide 1024 integer registers and 512 floating point registers. You have been invited as the operating system guru to give opinion about the new design.

1. ***What is the effect of having such a large number of registers on the operating system?***
2. ***What additional hardware features you would recommend added to the design above.***
3. ***What happens if the hardware designer also wants to add a 16-station pipeline into the CPU. How would that affect the context switching overhead?***

1. The cost of a context switch or interrupt handling becomes very expensive. Also, the data structures that store the registers as part of a process context become large. This leads high overhead in memory and also may make the implementation of lightweight threads expensive.
2. A multithreaded architecture would be a good use of the abundance of registers. The idea is to partition the registers into a number of banks, for example, a set of 1024 registers could be divided into 32 banks of 32 registers each. Then, wewould be able to have the register sets of 32 processes without having to save it or restore it each time a context switchoccurs.
3. Since the pipeline has to be flushed on an interrupt, a deeper pipeline will take time to be flushed increasing latency ofserving the interrupt and overhead. The penalty of a context switch increases.

Q4: Given the following piece of code:

```
main(int argc, char ** argv)
{
        int child = fork();
        int c = 5;

        if(child == 0)
        {

                c += 5;
        }
        else
        {
                child = fork();
                c += 10;
                if(child)
                        c += 5;
        }
}
```

How many different copies of the variable c are there? What are their values?

The piece of code shown creates two processes. Therefore, we have a total of three processes, the parent, the first and second child. Each of these has its own private copy of the variable c. For the parent, the variable c be 20 before the end of the program. For the first child (the one created in the first program statement), the variable c will contain the value 10 before the end of the program. For the second child (the one created in the else clause), the variable c will contain the value 15 before the end of the program.

Q5: Given the following piece of code

```
main(int argc, char ** argv)
{
        forkthem(5)
}
void forkthem(int n)
{
        if(n > 0)
        {
                fork();
                forkthem(n-1);
        }
    }
```

How many processes are created if the above piece of code is run? Hint: It may be easier to solve this problem by induction.

To solve this problem, we compute the number of processes that get created by calling the function forkthem(). This can be given
by the following equation:

n > 0: T(n) = 2 T(n-1) + 1
n = 0: T(0) = 0

where T(n) is the number of processes created by the function. To see why this is the case, consider what happens when the function is called. The first statement calls the system call fork() which creates a child in addition to the caller. Both the caller and the child then execute a recursive call to forkthem() with an argument set to n-1. Therefore, a call to forkthem() creates one process of its own, and then is responsible for all the children that will get created by the function with n-1.

The solution to the recurrence equation is 2^n - 1.

Q6: What is the output of the following programs (inspect the manual for the system calls if you need more information, but please solve the problem without compiling and running the program).

Program 1:

```
main()
{
   val = 5;
   if(fork())
      wait(&val);
   val++;
   printf("%d\n", val);
```

```
    return val;
}
```

Program 2:

```
main()
{
    val = 5;
    if(fork())
        wait(&val);
    else
        exit(val);
    val++;
    printf("%d\n", val);
    return val;
}
```

In the first program, the parent process creates a child and then waits for the child to exit (through the system call "wait"). The child executes and prints out the value of val, which is "6" after the v++ statement. The child then returns the value of val to the parent, which receives it in the argument to "wait" (& val). The parent then prints out the value of val, which is now 7. Note that the parent and child have seperate copies of the variable "val".

Using similar reasoning, you can see that the parent in program 2 waits for the child to return, and the child exits immediately. In this case only one value gets printed out which is the number 6 (from the parent process.)

Section 2: Scheduling

Q1: Most round-robin schedulers use a fixed size quantum. Give an argument in favor of and against a small quantum.

Solution:

An argument against a small time quantum: Efficiency. A small time quantum requires the timer to generate interrupts with short intervals. Each interrupt causes a context switch, so overhead increases with a larger number of interrupts.

An argument for a small time quantum: Response time. A large time quantum will reduce the overhead of context switching since
interrupts will be generated with relatively long intervals, hence there will be fewer interrupts. However, a short job will have to wait
longer time on the ready queue before it can get to execute on the processor. With a short time quantum, such a short job will finish
quicker and produces the result to the end user faster than with a longer time quantum.

Q2: As a system administrator, you have noticed that usage peaks between 10:00AM to 5:00PM and between 7:00PM to 10:00PM. The company's CEO decided to call on you to design a system where during these peak hours there will be three levels of users. Users in level 1 are to enjoy better response time than users in level 2, who in turn will enjoy better response time than users in level 3. You are to design such a system so that all users will still get some progress, but with the indicated preferences in place.

1. **Will a fixed priority scheme with pre-emption and 3 fixed priorities work? Why, or why not?**
2. **Will a UNIX-style multi-feedback queue work? Why, or why not?**
3. **If none of the above works, could you design a scheduling scheme that meets the requirements?**

Solution:
1. No, it will not work. A fixed priority scheme can cause starvation. The required solution should enable all users to make progress. Fixed priority does not guarantee progress for processes with low priorities.

2. No, it will not work. The multi-feedback queuing system will cause processes in level 1 to get less time on the CPU if they stay in the system for very long. So, even though a level-1 process may start at the highest level in the feedback queue, its priority will degrade over time. So this solution does not satisfy the requirements.

3. A process of level-1 will have 3 entries in the ready queue, distributed evenly over the queue. A process of level-2 will have 2 entries, while a process of level 3 will have one entry. In a run, a process at level 1 will get 3 times as much as a process at level 3 on the CPU. Care should be taken such that when a process requests an I/O

operation, that all its entries would be removed from the ready queue simultaneously. Also, when a process is added to the ready queue, it has to be entered with all its entries even distributed over the entire queue. Other solutions such as a lottery scheduler are possible.

Q3: A periodic task is one that is characterized with a period T, and a CPU time C, such that for every period T of real time, the task executes C time on the CPU. Periodic tasks occur in real-time applications such as multimedia. For example, a video display process needs to draw a picture every 40 msec (outside the US, Japan and Canada). To do so, such a process will have a period of 40 msec, and within each, it will require some time to do the data copying and drawing the picture after getting it from the video stream.

Admission control is a policy implemented by the process manager in which the manager decides whether it will be able to run a task with given parameters and with the existing load. Of course, if the process manager over commits itself, a periodic task will not be able to meet its processing requirement every period. For the video display process example, the picture flow will slow down and a viewer will notice and get irritated.

Assume that a system has 5 processes, an editor, a mail checking program, a video display process, a network browser and a clock display. The requirements are such that the mail checking program has to execute for 10 msec every 1sec, the video display must execute 25 msec every 40 msec, and the clock display must execute for 10 msec every 1 sec.

1. **What scheduling policy would you use for such a system?**
2. **Two process schedulers are available for you. The first would consume about 5 msec of overhead in context switching every 40 msec, while the second consumes 0.5 msec of overhead in context switching every 40 msec. The first ensures that the**

periodic processes will get time on the CPU every 40 msec if they are ready, while the second uses simple round-robin scheduling. Which one is better suited for this application batch? Which one is more efficient?

3. **Assume that the clock display program actually underestimated the time it wanted to run, and instead has now to run for 20 msec every 40 msec. What should the system do? How could it do it? Explain your answers.**
   If an audio unit is to be installed with an audio playback processing requiring 15 msec every 40 msec, would you be able to admit the process into the system? If yes explain why, and if not, explain why and suggest what the system could do.

Solution:

1. A fixed priority scheme in which the video display would have the highest priority, followed by the clock, then followed by the rest.

2. While the second scheduling algorithm is more efficient than the first, it is practically useless for such a batch of applications because simple round-robin does not give the necessary guarantees that the video display program will run every 40-msec.

3. The system has to be able to stop the offending process. To do so, the system must set timers that would generate interrupts at the time limit by which a task has to finish. So, if the task has not finished after its allotted time has expired, then it is simply switched out and another application is run. This requires a very complex implementation to ensure that the timers are being handled properly.

4. The system cannot admit such a process, because it does not have the capacity to handle the workload and guarantee the time limits required for the other processes. In situations like this, the system may simply tell the user "sorry", or it may give her the ability to stop some of the existing applications so that the freed capacity be used to run the new application. Or, it may simply tell the user "we will not run this as it should, do you want to continue anyway?".

Q4: Comparison of FIFO, RR and SRTF Algorithms. Given the following mix of job, job lengths, and arrival times, assume a time slice of 10 and compute the completion for each job and average response time for the FIFO, RR, and SRTF algorithms.

Solution:

			Scheduling Algorithms					
Job	Length (secs)	Arrival time	FIFO		RR		SRTF	
			Completion time	Response time	Completion time	Response time	Completion time	Response time
0	85	0	85	85	220	220	220	220
1	30	10	115	105	80	70	40	30
2	35	10	150	140	125	115	75	65
3	20	80	170	90	145	65	100	20
4	50	85	220	135	215	130	150	65
Average Response Time				111		120		80

Note:
RR's scheduling is:

0:	0	
10:	1	* if scheduling 0, the result is listed below.
20:	2	
30:	0	
40:	1	
50:	2	
60:	0	
70:	1	
80:	2	-------->Job1 completes
90:	0	
100:	3	
110:	4	
120:	2	
125:	0	--------->Job2 completes
135:	3	
145:	4	--------->Job3 completes
155:	0	
165:	4	
175:	0	
185:	4	
195:	0	

205: 4
215: 0 --------->Job4 completes
220: --------->Job0 completes

RR can also be:

			Scheduling Algorithms					
Job	Length (secs)	Arrival time	FIFO		RR		SRTF	
			Completion time	Response time	Completion time	Response time	Completion time	Response time
0	85	0	85	85	220	220	220	220
1	30	10	115	105	90	80	40	30
2	35	10	150	140	135	125	75	65
3	20	80	170	90	155	75	100	20
4	50	85	220	135	220	135	150	65
Average Response Time				111		**125**		80

Q5: Many scheduling algorithms are parameterized. For instance, the round-robin algorithm requires a parameter to indicate the time quantum. The multi-level feedback (MLF) scheduling algorithm requires parameters to define the number of queues, the scheduling algorithm for each queue, and the criteria to move processes between queues (and perhaps others. . .).

> **Hence, each of these algorithms represents a set of algorithms (e.g., the set of round-robin algorithms with different quantum sizes). Further, one set of algorithms may *simulate* another (e.g., round-robin with infinite quantum duration is the same as first-come, first-served (FCFS)). For each of the following pairs of algorithms, answer the following questions:**

1. Priority scheduling and shortest job first (SJF)

a. State the parameters and behavior of priority scheduling
   Solution
   Parameter: Each job has a priority P
   Behavior: Choose the job with the lowest numerical priority
b. State the parameters and behavior of SJF
   Solution
   Parameter: Each job has a remaining time T
   Behavior: Choose the job with the lowest remaining T
c. Can SJF simulate priority scheduling for all possible parameters of priority scheduling? (How or why not: State how to set SJF scheduling parameters as a function of priority scheduling parameters or explain why this cannot be done.)
   Solution
   2 possible answers: (1) No - although both schedulers are priority queues, SJF is more restrictive than a general priority scheduler since a job's time may not correspond to its priority. (2) Yes - "spoof" the scheduler by passing priority in place of time.
d. Can priority scheduling simulate SJF for all possible parameters of SJF? (How or why not.)
   Solution
   Yes - set the priority P to the remaining time T

2. Multilevel feedback queues and first come first served (FCFS)
   a. State the parameters and behavior of multi-level queues
      Solution
      Parameters: N (# queues), scheduling algorithm for each queue, function that selects in which queue to place a job, criteria to interrupt a running job
      Behavior: Always schedule the job from the head of the lowest #'d non-empty queue according to that queue's scheduling algorithm; interrupt a running job when told to by the interrupt criteria; place newly arrived jobs or interrupted jobs in a queue based on the selection function
   b. State the parameters and behavior of FCFS
      Solution
      No parameters
      Behavior: Place arriving jobs at the back of a FIFO queue; when scheduling a new job, choose the one from the front of the FIFO queue; never interrupt a running job
   c. Can FCFS simulate multi-level feedback for all possible parameters of multi-level feedback? (How or why not?)
      Solution

No. Counterexample: make the top queue of MLF run RR with a short time quantum. If two jobs arrive and each is longer than the quantum, the MLF scheduler will allow both jobs to run for at least one quantum before either completes, but FCFS can never interrupt a running job, so one of the two will complete before the other has a chance to run.

d. Can multi-level feedback scheduling simulate FCFS for all possible parameters of FCFS? (How or why not?)

Solution

Yes. Make MLF have 1 queue and have that one queue run FCFS. Function to select which queue - always choose that queue. Policy to interrupt - Never.

3. Priority scheduling and first come first served (FCFS)

a. Can FCFS simulate priority scheduling for all possible parameters of priority scheduling? (How or why not?)

Solution

No. Counterexample: Suppose job 1 arrives at time 1 with priority 10 and length 1,000,000 and job 2 arrives at time 2 with priority 1 and length 1,000,000. FCFS will run job 1 to completion, then job 2 to completion. Priority will run job 1 for 1 unit, then job 2 to completion, then job 1 to completion.

b. Can priority scheduling simulate FCFS for all possible parameters of FCFS? (How or why not?)

Solution

Yes. Set all jobs to equal priority (we assume that the priority queue breaks ties with FCFS and infinite time quantum)

4. Round-robin and shortest job first (SJF)

a. State the parameters and behavior of round robin

Solution

Parameter: Quantum Q

Behavior: Run FIFO queue, but interrupt a running job Q units after it begins running and move it to the end of the FIFO queue

b. Can round robin simulate SJF for all possible parameters of SJF? (How or why not?)

Solution

No. Counterexample. 3 jobs arrive: job 1 at time 0 is 10 seconds long, job 2 arrives at time 1 and is 100 seconds long, job 3 at time 2 is 10 seconds long. SJF must run 1 to completion, then 3 to completion, and then 2 to completion. Claim: RR must run at least one quantum of job 2 before running any job 3.

c. Can SJF simulate round robin for all possible parameters of round robin? (How or why not?)
   Solution
   No. Same counterexample.

Section 3: Memory Management

Q1: In a 32-bit machine we subdivide the virtual address into 4 segments as follows:

10-bit	8-bit	6-bit	8 bit

We use a 3-level page table, such that the first 10-bit are for the first level and so on.

1. What is the page size in such a system?
2. What is the size of a page table for a process that has 256K of memory starting at address 0?
3. What is the size of a page table for a process that has a code segment of 48K starting at address 0x1000000, a data segment of 600K starting at address 0x80000000 and a stack segment of 64K starting at address 0xf0000000 and growing upward (like in the PA-RISC of HP)?

Solution:

1. The page field is 8-bit wide, then the page size is 256 bytes.
2. Using the subdivision above, the first level page table points to 1024 2nd level page tables, each pointing to 256 3rd page tables, each containing 64 entries. The program's address space consists of 1024 pages, thus we need 1024/64=16 third-level page tables (because each table has 64 entries). Therefore we need 16 entries in only one 2nd level page table (because 16 < 256 and hence one 2nd level page table is large enough to hold 16 entries), which, in turn, requires one entry in the first level page table (because 1 <1024). The size is thus: 1024 entries for the first table, 256 entries for the 2nd level page table, and 16 3rd level page table containing 64 entries each. Assuming 2 bytes per entry, the space required is 1024 * 2 + 256 * 2 (one second-level page table) + 16 * 64 * 2 (16 third-level page tables) = 4608 bytes.
3. First, the stack, data and code segments are at addresses that require having 3 page-table entries active in the first level page table (hint: think about the range

of virtual addresses that fit into each entry of the first-level page table). For 64K, you need 256 pages, or 4 third-level page tables. For 600K, you need 2400 pages, or 38 third-level page tables and for 48K you need 192 pages or 3 third-level page tables. Assuming 2 bytes per entry, the space required is 1024 * 2 + 256 * 3 * 2 (3 second-level page tables) + 64 * (38+4+3)* 2 (38 third-level page tables for data segment, 4 for stack and 3 for code segment) = 9344 bytes.

Q2: A computer system has a 36-bit virtual address space with a page size of 8K, and 4 bytes per page table entry.

1. How many pages are in the virtual address space?
2. What is the maximum size of addressable physical memory in this system?
3. If the average process size is 8GB, would you use a one-level, two-level, or three-level page table? Why?
4. Compute the average size of a page table in question 3 above.

Solution:

1. A 36 bit address can address 2^36 bytes in a byte addressable machine. Since the size of a page 8K bytes (2^13), the number of addressable pages is 2^36 / >2^13 = 2^23
2. With 4 byte entries in the page table we can reference 2^32 pages. Since each page is 2^13 B long, the maximum addressable physical memory size is 2^32 * 2^13 = 2^45 B (assuming no protection bits are used).
3. 8 GB = 2^33 B

   We need to analyze memory and time requirements of paging schemes in order to make a decision. Average process size is considered in the calculations below.

   1 Level Paging
   Since we have 2^23 pages in each virtual address space, and we use 4 bytes per page table entry, the size of the page table will be 2^23 * 2^2 = 2^25. This is 1/256 of the process' own memory space, so it is quite costly. (32 MB)

   2 Level Paging
   The address would be divided up as 12 | 11 | 13 since we want page table pages to fit into one page and we also want to divide the bits roughly equally.

   Since the process' size is 8GB = 2^33 B, I assume what this means is that the total size of all the distinct pages that the process accesses is 2^33 B. Hence, this process accesses 2^33 / 2^13 = 2^20 pages. The bottom level of the page table then holds 2^20 references. We know the size of each bottom level chunk of the page table is 2^11 entries. So we need 2^20 / 2^11 = 2^9 of those bottom level chunks.

The total size of the page table is then:

//size of the outer page table	//total size of the inner pages	
1 * 2^12 * 4	+ 2^9 * 2^11 * 4	= 2^20 * (2^-6 + 4) ~4MB

3 Level Paging
For 3 level paging we can divide up the address as follows:
8 | 8 | 7 | 13

Again using the same reasoning as above we need 2^20/2^7 = 2^13 level 3 page table chunks. Each level 2 page table chunk references 2^8 level 3 page table chunks. So we need 2^13/2^8 = 2^5 level-2 tables. And, of course, one level-1 table.

The total size of the page table is then:

//size of the outer page table	//total size of the level 2 tables	//total size of innermost tables	
1 * 2^8 * 4	2^5 * 2^8 *4	2^13 * 2^7 * 4	~4MB

As easily seen, 2-level and 3-level paging require much less space then level 1 paging scheme. And since our address space is not large enough, 3-level paging does not perform any better than 2 level paging. Due to the cost of memory accesses, choosing a 2 level paging scheme for this process is much more logical.

4. Calculations are done in answer no. 3.

Q3: In a 32-bit machine we subdivide the virtual address into 4 pieces as follows:

8-bit 4-bit 8-bit 12-bit

We use a 3-level page table, such that the first 8 bits are for the first level and so on. Physical addresses are 44 bits and there are 4 protection bits per page. Answer the following questions, showing all the steps you take to reach the answer. A simple number will not receive any credit.

1. What is the page size in such a system? Explain your answer (a number without justification will not get any credit).
2. How much memory is consumed by the page table and wasted by internal fragmentation for a process that has 64K of memory starting at address 0?
3. How much memory is consumed by the page table and wasted by internal fragmentation for a process that has a code segment of 48K starting at address 0x1000000, a data segment of 600K starting at address 0x80000000 and a stack segment of 64K starting at address 0xf0000000 and growing upward (towards higher addresses)?

Solution:

1. 4K. The last 12 bits of the virtual address are the offset in a page, varying from 0 to 4095. So the page size is 4096, that is, 4K.
2. 2912 or 4224 bytes for page tables, 0 bytes for internal fragmentation.
   Using the subdivision above, the 1st level page table contains 256 entries, each entry pointing to a 2nd level page table. A 2nd level page table contains 16 entries, each entry pointing to a 3rd page table. A 3rd page table contains 256 entries, each entry pointing to a page. The process's address space consists of 16 pages, thus we need 1 third-level page table. Therefore we need 1 entry in a 2nd level page table, and one entry in the first level page table. Therefore the size is: 256 entries for the first table, 16 entries for the 2nd level page table, and 1 3rd level page table containing 256 entries.

   Since physical addresses are 44 bits and page size is 4K, the page frame number occupies 32 bits. Taking the 4 protection bits into account, each entry of the level-3 page table takes (32+4) = 36 bits. Rounding up to make entries byte (word) aligned would make each entry consume 40 (64) bits or 5 (8) bytes. For a 256 entry table, we need 1280 (2048) bytes.

   The top-level page table should not assume that 2nd level page tables are page-aligned. So, we store full physical addresses there. Fortunately, we do not need control bits. So, each entry is at least 44 bits (6 bytes for byte-aligned, 8 bytes for word-aligned). Each top-level page table is therefore 256*6 = 1536 bytes (256 * 8 = 2048 bytes).

Trying to take advantage of the 256-entry alignment to reduce entry size is probably not worth the trouble. Doing so would be complex; you would need to write a new memory allocator that guarantees such alignment. Further, we cannot quite fit a table into a 1024-byte aligned region (44-10 = 34 bits per address, which would require more than 4 bytes per entry), and rounding the size up to the next power of 2 would not save us any size over just storing pointers and using the regular allocator.

Similarly, each entry in the 2nd level page table is a 44-bit physical pointer, 6 bytes (8 bytes) when aligned to byte (word) alignment. A 16 entry table is therefore 96 (128) bytes. So the space required is 1536 (2048) bytes for the top-level page table + 96 (128) bytes for one second-level page table + 1280 (2048) bytes for one third-level page table = 2912 (4224) bytes. Since the process can fit exactly into 16 pages, there is no memory wasted by internal fragmentation.

3. 5664 or 8576 bytes for page tables, 0 bytes.
   First, the stack, data and code segments are at addresses that require having 3 page table entries active in the first level page table, so we have 3 second-level page tables. For 48K, you need 12 pages or 1 third-level page table; for 600K, you need 150 pages, or 1 third-level page table and for 64K you need 16 pages or 1 third-level page table.

   So the space required is 1536 (2048) bytes for the top level page table + 3 * 96 (3 * 128) bytes for 3 second-level page tables + 3 * 1280 (3 * 2048) for 3 third-level page table = 5664 (8576) bytes.

   As the code, data, stack segment of the process fits exactly into 12, 150, 16 pages respectively, there is no memory wasted by internal fragmentation.

Q4: Describe the advantages of using a MMU that incorporates segmentation and paging over ones that are either pure paging or pure segmentation. Present your answer as separate lists of advantages over each of the pure schemes.

Q5: Consider the following piece of code which multiplies two matrices:

```
int a[1024][1024], b[1024][1024], c[1024][1024];
multiply()
{
   unsigned i, j, k;
```

```
    for(i = 0; i < 1024; i++)
        for(j = 0; j < 1024; j++)
            for(k = 0; k < 1024; k++)
                c[i][j] += a[i,k] * b[k,j];
}
```

Assume that the binary for executing this function fits in one page, and the stack also fits in one page. Assume further that an integer requires 4 bytes for storage. Compute the number of TLB misses if the page size is 4096 and the TLB has 8 entries with a replacement policy consisting of LRU.

Solution:
1024*(2+1024*1024) = 1073743872
The binary and the stack each fit in one page, thus each takes one entry in the TLB. While the function is running, it is accessing the binary page and the stack page all the time. So the two TLB entries for these two pages would reside in the TLB all the time and the data can only take the remaining 6 TLB entries.

We assume the two entries are already in TLB when the function begins to run. Then we need only consider those data pages.

Since an integer requires 4 bytes for storage and the page size is 4096 bytes, each array requires 1024 pages. Suppose each row of an array is stored in one page. Then these pages can be represented as a[0..1023], b[0..1023], c[0..1023]: Page a[0] contains the elements a[0][0..1023], page a[1] contains the elements a[1][0..1023], etc.

For a fixed value of i, say 0, the function loops over j and k, we have the following reference string:

a[0], b[0], c[0], a[0], b[1], c[0], ¡ a[0], b[1023], c[0]
¡
a[0], b[0], c[0], a[0], b[1], c[0], ¡ a[0], b[1023], c[0]

For the reference string (1024 rows in total), a[0], c[0] will contribute two TLB misses. Since a[0] and b[0] each will be accessed every four memory references, the two pages will not be replaced by the LRU algorithm. For each page in b[0..1023], it will incur one TLB miss every time it is accessed. So the number of TLB misses for the second inner loop is
2+1024*1024 = 1048578
So the total number of TLB misses is 1024*1048578 = 1073743872

Q6: A computer system has a page size of 1,024 bytes and maintains the page table for each process in main memory. The overhead required for doing a lookup in the page table is 500 ns. To reduce this overhead, the computer has a TLB that caches 32 virtual pages to physical frame mappings. A TLB lookup requires 100ns. What TLB hit-rate is required to ensure an average virtual address translation time of 200ns?

Q7: Discuss the issues involved in picking a page size for a virtual memory system.

- **a. Name one issue that argues for small page sizes? Name two that argue for large page sizes?**
- **b. How do the characteristics of disks influence the selection of a page size?**

Q8: Consider a system with a virtual address size of 64MB (2^26), a physical memory of size 2GB (2^31), and a page size of 1K (2^10). Under the target workload, 32 processes (2^5) are running; half of the processes are smaller than 8K (2^13) and half use the full 64MB virtual address space. Each page has 4 control bits.

1. What is the size of a single top-level page table entry (and why)?

2. What is the size of a single bottom-level page table entry (and why)?

3. If you had to choose between two arrangements of page table: 2-level and 3-level, which would you choose and why? Compute the expected space overhead for each variation: State the space overhead for each small process and each large process. Then compute the total space overhead for the entire system.

Q9: Suppose a program references pages in the following sequence:

ACBDBAEFBFAGEFA

Suppose the computer on which this program is running has 4 pages of physical memory.

1. Show how LRU-based demand paging would fault pages into the four frames of physical memory.
   Solution:

	A	C	B	D	B	A	E	F	B	F	A	G	E	F	A
1	A					+					+				+
2		C					E					G			
3			B		+				+				E		
4				D				F		+				+	

2. Show how optimal demand paging would fault pages into the four frames of physical memory.
   Solution:

	A	C	B	D	B	A	E	F	B	F	A	G	E	F	A
1	A					+					+				+
2		C					E						+		
3			B		+				+			G			
4				D				F		+				+	

3. Show how clock-based demand paging would fault pages into the four frames of physical memory.
   Solution:

	A	C	B	D	B	A	E	F	B	F	A	G	E	F	A
1	***A1***	***1***	***1***	***1***	***1***	***+1***	E1	1	1	1	***1***	0	+1	1	1
2		C1	1	1	1	1	***0***	F1	1	+1	1	0	0	+1	1
3			B1	1	+1	1	0	***0***	***+1***	***1***	0	G1	1	1	1

4				D1	1	1	0	0	0	0	A1	***1***	***1***	***1***	***+1***

"+" (plus sign) means cache hit
1/0 means the state of the "used" bit at the end of the specified step

bold italics is the position of the clock hand at the end of the specified step (this is what will be looked at first the next time the clock hand moves)