CS 4348/5348 Operating Systems, Homework #2

1. Implement a monitor solution for the bakery problem. Assume that you have N salesmen and customers arrive at arbitrary times. Note: neither the customers nor the salesmen should have a busy waiting.

```
Monitor
{  int sales_count, initialized to N
   int sales_wait, initialized to 0
   int cust_count, initialized to 0
   condition sales, cust;

  get_service()
  {  if (sales_count <= 0) cust.wait;
     cust_count = cust_count + 1;
     if (sales_count = sales_wait) sales.signal;
    sales_count = sales_count – 1;
  }

  release_service()
  { } -- this can be omitted

  prepare_service ()
  { if (cust_count <= 0) { sales_wait = sales_wait + 1;  sales.wait;  sales_wait = sales_wait – 1; }
    cust_count = cust_count – 1;
  }

  complete_service()
  { sales_count = sales_count + 1;
    cust.signal;
  }
}   // end monitor

customer
{ get_service ();
  receive the service
  release_service ();
}

salesman
{ prepare_service();
  provide the service
  complete_service();
}
```

2. Discuss the steps for implementing the Monitor construct using integer semaphores. Consider letting the signalee continue after condition signaling. You need to consider all steps in the implementation and give more implementation details than those shown in lecture slides (which are incomplete).

   The monitor should be protected by a semaphore “mutex”.

For each condition cond, we need to use a semaphore and a separate counter (let condsem and counter represent them). For cond.wait, we always set condsem.count to 0 (e.g., initialize semaphore again to force it to be set to 0) before wait(condsem), so that cond.wait will be forced to wait. For cond.signal, we should set condsem.count to negative of the counter value before signal(condsem) (so that a process in condsem.queue can be released correctly or when condsem.queue is empty, condsem.count = 0).

We also need to use a semaphore temp and maintain a counter tempcount for holding the signalers after conditional signals. When implementing cond.signal, after signal(condsem), execute

tempcount++; wait(temp);

When any process is exiting Monitor, execute:

if tempcount > 0 { tempcount– –; temp.signal; } else mutex.signal;

3. Given the system state, follow Banker's algorithm to determine whether to grant or deny the requests. You need to compute the "Needed" matrix (numbers of resources the processes still need) and the "Available" vector (the available resources currently in the system) first. Then you need to pretend to allocate and perform the safety analysis. You need to show your work for all the steps.

	Total	Allocated				MaxReq				Needed				Available
		P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	
R_1	3	1	0	0	0	1	1	2	2					
R_2	6	1	1	0	1	2	2	2	1					
R_3	4	0	0	1	1	2	1	1	2					

	Requested			
	P_1	P_2	P_3	P_4
R_1	0	1	0	1
R_2	0	0	0	0
R_3	1	1	0	0

	Total	Allocated				MaxReq				Needed				Available
		P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	
R_1	3	1	0	0	0	1	1	2	2	0	1	2	2	2
R_2	6	1	1	0	1	2	2	2	1	1	1	2	0	3
R_3	4	0	0	1	1	2	1	1	2	2	1	0	1	2

#Preliminary Checks

Request is Request[i, j] < Need[i,j] for any i,j ⇒ no problem

Request is $\sum_i$ Request[i, j] < Available[j] for any j ⇒ no problem

	Total	Allocated				MaxReq				Needed				Available
		P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	
R_1	3	1	1	0	1	1	1	2	2	0	0	2	1	0
R_2	6	1	1	0	1	2	2	2	1	1	1	2	0	3
R_3	4	1	1	1	1	2	1	1	2	1	0	0	1	0

P_2 can continue, choose P_2 to continue and P_2 returns all its resources

R_1	3	1	0	0	1	1	1	2	2	0	0	2	1	1
R_2	6	1	0	0	1	2	2	2	1	1	0	2	0	4
R_3	4	1	0	1	1	2	1	1	2	1	0	0	1	1

P_4 or P_1 can continue, choose P_4 to continue and P_4 returns all its resources

R_1	3	1	0	0	0	1	1	2	2	0	0	2	0	2
R_2	6	1	0	0	0	2	2	2	1	1	0	2	0	5

R_3	4	1 0 1 0	2 1 1 2	1 0 0 0	2

P_1 or P_3 can continue, choose P_1 to continue and P_1 returns all its resources

R_1	3	0 0 0 0	1 1 2 2	0 0 2 0	3
R_2	6	0 0 0 0	2 2 2 1	0 0 2 0	6
R_3	4	0 0 1 0	2 1 1 2	0 0 0 0	3

P_3 can continue and all processes can finish ⇒ So, it is safe, grant the request

4. Given the current system state, follow the deadlock detection algorithm to check whether there is a deadlock. You need to show your work clearly. If there is deadlock, you should also indicate which processes are involved in the deadlock.

	Total	Allocated	Requested
		P_1 P_2 P_3	P_1 P_2 P_3
R_1	1	1 0 0	0 0 1
R_2	1	0 1 0	1 0 0
R_3	1	0 0 0	0 1 1
R_4	1	0 0 1	1 0 0

	Total	Allocated	Avl	Requsted	
		P_1 P_2 P_3		P_1 P_2 P_3	
R_1	1	1 0 0	0	0 0 1	
R_2	1	0 1 0	0	1 0 0	
R_3	1	0 0 0	1	0 1 1	P_2 can get the resources
R_4	1	0 0 1	0	1 0 0	⇒ P_2 return resources

R_1	1	1 0 0	0	0 0 1	
R_2	1	0 0 0	1	1 0 0	No process can get the resources
R_3	1	0 0 0	1	0 0 1	⇒ Deadlock
R_4	1	0 0 1	0	1 0 0	P_1 and P_3 involved

5. Consider a memory system with 1GB space (=1024MB). Currently the memory free list contains ((128MB, 64MB), (320MB, 8MB), (512MB, 16MB), (768MB, 256MB)) where each item (x, y) in the list represents the starting location x and length y of a free partition. Four jobs J_1, J_2, J_3, and J_4 of sizes 14MB, 20MB, 160MB, and 64MB, respectively, are to be loaded to the memory. Consider the first fit allocation policy and answer the following questions.

   (a) Consider each job J_i, $1 \le i \le 4$. When J_i is running, what would be the value for the base register and bound register?

J_1 gets (128MB, 64MB) slot
⇒ base register 0x08000000, bound register 0x08dfffff
(or 0x08e00000 if we use A>=B as the circuit, but consider the above as the correct answer)

J_2 gets allocated after J_1 in (128MB, 64MB) slot
⇒ base register 0x08e00000, bound register 0x0a1fffff (starting address of next slot is 0x0a200000)

J_3 gets (768MB, 256MB) slot
⇒ base register 0x30000000, bound register 0x39ffffff (starting address of next slot is 0x3a000000)

J_4 gets allocated after J_3 in (768MB, 256MB) slot
⇒ base register 0x3a000000, bound register 0x3dffffff (starting address of next slot is 0x3e000000)

(b) Now J_4 is running and it accesses the logical address 0x04213a0c. Compute the corresponding physical address. Also, check and determine whether this causes access violation.

J_4 base register 0x3a000000 + logical address 0x04213a0c
⇒ physical address 0x3e213a0c > bound register 0x3dffffff
⇒ access violation

(c) Now J_4 is running and it accesses the logical address 0x02f3f12c. Compute the corresponding physical address. Also, check and determine whether this causes access violation.

J_4 base register 0x3a000000 + logical address 0x02f3f12c
⇒ physical address 0x3cf3f12c < bound register 0x3dffffff
⇒ good address, no access violation

6. Consider a 1GB memory system using buddy scheme. The system loads jobs J_1, J_2, J_3, J_4, and J_5 of size 40MB, 20MB, 160MB, 64MB, and 100MB, respectively.

(a) Compute the starting addresses of each job.

J_1 starting at 0x00000000 (0, 64MB), internal fragmentation = 24MB
J_2 starting at 0x04000000 (64MB, 32MB), internal fragmentation = 12MB
J_3 starting at 0x10000000 (256MB, 256MB), internal fragmentation = 96MB
J_4 starting at 0x08000000 (128MB, 64MB), internal fragmentation = 0
J_5 starting at 0x20000000 (512MB, 128MB), internal fragmentation = 28MB

(b) What is the total internal fragmentation?

Total fragmentation = 24+12+96+28 MB = 160MB

7. Consider a 256MB memory system using simple paging scheme. Each page is of size 16KB. The memory is byte addressable. The page table for Process *P* is as follows. Process *P*'s address space is only 5 pages. The numbers in the table are all in decimal.

0	11
1	25
2	40
3	3
4	13

(a) How many bits are required to address the entire memory?
(b) How many bits are required to address the offset within each page?
(c) What is the maximal possible internal fragmentation *P* can have?
(d) Given a logical address 0x0000ef5b, compute the corresponding physical address. Also, check and determine whether this causes access violation.
(e) Given a logical address 0x00017f5b, compute the corresponding physical address. Also, check and determine whether this causes access violation.

(a) 28 bits
(b) 14 bits
(c) 16KB – 1
(d) 0x0000ef5b = 1110 1111 0101 1011 ⇒ page 3, mapped to frame 3,
⇒ physical address is the same as the logical address
(e) 0x00017f5b = 0001 0111 1111 0101 1011 ⇒ page 5, does not exist ⇒ access violation

8. Consider a virtual memory system with cache, main memory, and disk. The cache size is 256KB and has an access time 15ns. The cache hit ratio is 90% (which is considered low). The main memory is 256MB and has an access time 150ns. 99.99% of the memory accesses would find the demanded pages in the main memory and 0.01% of them would generate page faults. The disk size is 20GB and has an access time 1ms. Compute the average memory access time.

   Average memory access time
   = 15 ns + (1–0.9) * 150 ns + (1–0.9)*(1–0.9999) * 10^6 ns = 15 ns + 15 ns + 10 ns = 40 ns

9. Consider a virtual memory system using 32-bit addressing. The physical memory is of size 4GB. Each page is of size 8KB. Assume that each entry of the page table fits in one word. A two-level page table is used and the root level contains 1K entries. A process P has the following page table.

Root level page table
0x00001000
0x00000100
null
……

Level 2 page table start at 0x00000000
frame 150
frame 21
disk
……

Level 2 page table start at 0x00000100
frame 33
disk
frame 120
……

Level 2 page table start at 0x00001000
frame 160
frame 512
……

   (a) How many bits are required to address the root page table?
   (b) How many bits are required to address each second level page table?
   (c) Convert the logical address 0000 0000 0100 0000 0101 0000 1111 0101 (binary) to physical address.

   (a) 10 bits
   (b) 9 bits
   (c) 0000 0000 0100 0000 0101 0000 1111 0101
   It is root table entry 1, secondary page table entry 2, which is mapped to frame 120 = 0x78
   Physical address is: 0000 0000 0000 1111 0001 0000 1111 0101 or 0x000f10f5