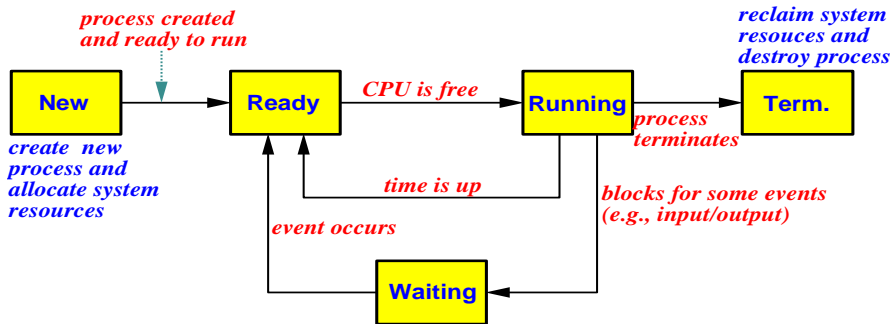


## CS4411 Intro. to Operating Systems Exam 2 Solutions Fall 2005

### 1. Recycled Problems

- (a) [10 points]\* Draw the state diagram of a process from its creation to termination, including all transitions, and briefly elaborate **each state** and **each transition**.

**Answer:** The following state diagram is taken from my overhead which was covered in class. Fill in the elaboration for each state and transition.



See p. 97 of our text for the details. ■

- (b) [5 points]\* The methods `Wait()` and `Signal()` must be atomic to ensure a correct implementation of mutual exclusion. Use an execution sequence to show that if `Wait()` is not atomic then mutual exclusion cannot be maintained. **You must use an execution sequence as we did many times in class to present your answer. Otherwise, you risk low or no credit.**

**Answer:** If `Wait()` is not atomic, its execution may be switched in the middle. If this happens, mutual exclusion will not be maintained. The following is a possible execution sequence, where `Count = 1` is the counter variable of the involved semaphore. This execution sequence clearly shows that mutual exclusion is not maintained if the `Wait()` method is not atomic.

Process A	Process B	Count	Comment
		1	Initial value
LOAD Count		1	A executes Count-- of Wait()
SUB #1		1	
	LOAD Count	1	B executes Count-- of Wait()
	SUB #1	1	
	SAVE Count	0	B finishes Count--
SAVE Count		0	A finishes Count--
if (Count < 0)		0	It is false from A
	if (Count < 0)	0	It is false from B
Both A and B enter the critical section			

This is an exercise in a reading list. ■

### 2. Synchronization

- (a) [10 points] Enumerate and elaborate all major differences between a semaphore wait/signal and a condition variable wait/signal. Vague answers and/or inaccurate or missing elaboration receive **no** credit.

**Answer:** The following table gives the details:

Semaphores	Condition Variables
Can be used anywhere, but not in a monitor	Can only be used in monitors
<code>wait()</code> does not always block its caller	<code>wait()</code> <b>always</b> blocks its caller
<code>signal()</code> increases the semaphore counter and may release a process	<code>signal()</code> either releases a process, or the signal is lost as it never occurs
If <code>signal()</code> releases a process, the caller and the released <b>both</b> continue	If <code>signal()</code> releases a process, either the caller or the released continues, but <b>not both</b>

This is part of the monitors slides used in class. ■

- (b) [5 points] Why is calling a monitor procedure from within another monitor (*i.e.*, nested monitor call) not a good practice?

**Answer:** Suppose thread  $T$  is in monitor  $M_1$  and calls monitor  $M_2$ . Due to the mutual exclusion property of a monitor, if thread  $T_1$  can enter monitor  $M_2$ , then thread  $T_1$  would be the only thread executing in monitors  $M_1$  and  $M_2$ . However, if thread  $T_1$  is blocked in monitor  $M_2$  (*e.g.*, a condition wait), then monitor  $M_1$  is not empty and no other threads can enter  $M_1$ . This is very inefficient. Worse, if while  $T_1$  is waiting on a condition variable in  $M_2$ , thread  $T_2$  enters  $M_2$  and calls  $M_1$ , then neither  $T_1$  nor  $T_2$  can continue because monitors  $M_1$  and  $M_2$  are both not empty. This is a circular waiting, and, hence, we have a deadlock.

This was discussed in class. ■

### 3. Process Scheduling

- (a) [8 points] What are *preemptive* and *non-preemptive* scheduling policies? Elaborate your answer.

**Answer:** With the *non-preemptive* scheduling policy, scheduling only occurs when a process enters the wait state or terminates. With the *preemptive* scheduling policy, scheduling also occurs when a process switches from running to ready due to an interrupt and from waiting to ready (*i.e.*, the event a thread has been waiting occurs – I/O completion).

See pp. 155–157 of our text. ■

- (b) [8 points] We discussed in class that the shortest job first (SJF) scheduling policy is “*provably optimal*.” What does this mean? And, show that the SJF scheduling is optimal for three jobs based on your answer to the previous question. **Note that are two questions. You have to answer each problem clearly and correctly.**

**Answer:** The “optimality” of the SJF algorithm means it always generates the smallest average waiting time, and, of course, the minimum waiting time. More precisely, the average waiting time under the SJF algorithm is smaller than the average waiting time of any known or unknown scheduling algorithm.

Suppose we have two jobs  $A$  and  $B$  with running time  $a$  and  $b$ . If  $A$  is run before  $B$ , the average waiting time is  $(0+a)/2 = a/2$ . On the other hand, if  $B$  is run before  $A$ , the average waiting time is  $(0+b)/2 = b$ . Thus, if a shorter job is run before a longer job, the average waiting time is reduced.

Since swapping the order of two consecutive out of order jobs reduces the average waiting time, we can scan the job array and swap out of order jobs. Eventually, when the job array is “sorted” by running time in ascending order, no average waiting time reduction is possible. As a result, if the job sequence is “sorted” by running time, the average waiting time reaches the minimum. However, if jobs are “sorted” based on running time in ascending order, it is exactly the SJF algorithm, and, hence, SJF is optimal.

This was discussed in class. ■

- (c) [20 points] Five processes *A*, *B*, *C*, *D* and *E* arrive in this order at the same time with the following CPU burst and priority values. A smaller value means a higher priority.

	<i>CPU Burst</i>	<i>Priority</i>
<i>A</i>	7	3
<i>B</i>	2	5
<i>C</i>	3	1
<i>D</i>	6	4
<i>E</i>	4	2

Fill the entries of the following table with waiting time and average waiting time for each indicated scheduling policy and each process. Ignore context switching overhead.

Answer:

<i>Scheduling Policy</i>	<i>Waiting Time</i>					<i>Average Waiting Time</i>
	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	
First-Come-First-Served	0	7	9	12	18	46/5=9.2
Non-Preemptive Shortest-Job First	15	0	2	9	5	31/5=6.2
Priority	7	20	0	14	3	44/5=8.8
Round-Robin (time quantum=2)	15	2	10	15	13	55/5=11

See the course slides for the details. ■

#### 4. Deadlocks

- (a) [10 points] What are the necessary conditions for a deadlock to occur? List these conditions and provide an elaboration. Providing conditions without an elaboration or providing a vague elaboration receives **no** credit.

Answer: There are four necessary conditions:

- **Mutual Exclusion:** Resources are not sharable. That is, the use of resources must be mutually exclusive.
- **Hold and Wait:** Processes hold some resources while waiting for additional ones.
- **No Preemption:** Resources can only be released by processes voluntarily. They cannot be preempted by the system.
- **Circular Waiting:** A set of processes  $P_1, P_2, \dots, P_n$  exists such that  $P_1$  is waiting for the resources that are being held by  $P_2$ ;  $P_2$  is waiting for the resources that are being held by  $P_3$ ; ...; and  $P_n$  is waiting for the resources that are being held by  $P_1$ .

See p. 247–249 of our text. ■

- (b) [10 points] Consider the following snapshot of a system:

	<i>Allocation</i>				<i>Max</i>				<i>Need</i>				<i>Available</i>			
	<i>U</i>	<i>V</i>	<i>W</i>	<i>X</i>	<i>U</i>	<i>V</i>	<i>W</i>	<i>X</i>	<i>U</i>	<i>V</i>	<i>W</i>	<i>X</i>	<i>U</i>	<i>V</i>	<i>W</i>	<i>X</i>
<i>A</i>	0	0	1	2	0	0	1	2	0	0	0	0	1	5	2	0
<i>B</i>	1	0	0	0	1	7	5	0	0	7	5	0				
<i>C</i>	1	3	5	4	2	3	5	6	1	0	0	2				
<i>D</i>	0	6	3	2	0	6	5	2	0	0	2	0				
<i>E</i>	0	0	1	4	0	6	5	6	0	6	4	2				

Is the system in a safe state? **Show your computation step-by-step; otherwise, you will receive no credit.**

**Answer:** The following shows the steps to find a safe sequence. Note that we always search for a candidate in the order of  $A, B, C, D$  and  $E$ .

- Since  $Available = [1,5,2,0]$  and is greater than  $A$ 's  $Need=[0,0,0,0]$ , we can run  $A$ . After  $A$  completes,  $Available = [1,5,2,0]+[0,0,1,2]=[1,5,3,2]$ .
- Since  $Available=[1,5,3,2]$  is greater than  $C$ 's  $Need=[1,0,0,2]$ , we can run  $C$ . After  $C$  completes,  $Available= [1,5,3,2]+[1,3,5,4]=[2,8,8,6]$ .
- Since  $Available=[2,8,8,6]$  is greater than  $B$ 's  $Need=[0,7,5,0]$ , we can run  $B$ . After  $B$  completes,  $Available= [2,8,8,6]+[1,0,0,0]=[3,8,8,6]$ .
- Since  $Available=[3,8,8,6]$  is greater than  $D$ 's  $Need=[0,0,2,0]$ , we can run  $D$ . After  $D$  completes,  $Available=[3,8,8,6]+[0,6,3,2]=[3,14,11,8]$ .
- Since  $Available=[3,14,11,8]$  is greater than  $E$ 's  $Need=[0,6,4,2]$ , we can run  $E$ .

Therefore, if the five processes are run in the order of  $A, C, B, D$  and  $E$ , all of them can finish and the system is safe (*i.e.*,  $\langle A, C, B, D, E \rangle$  is a safe sequence). Note that there are other safe sequences (*e.g.*,  $\langle A, D, C, B, E \rangle$ ). See **pp. 259–262** of our text. ■

(c) [8 points] Consider the following snapshot of a system:

	Allocation				Request				Available			
	U	V	W	X	U	V	W	X	U	V	W	X
A	0	0	1	0	2	0	0	1	2	1	0	0
B	2	0	0	1	1	0	1	0				
C	0	1	2	0	2	1	0	0				

Is the system in a deadlock state? If the system is in a deadlock state, what are the processes that involve in a deadlock. **Show your computation step-by-step; otherwise, you will receive no credit.**

**Answer:** Since only  $C$ 's  $Request = [2, 1, 0, 0]$  is less than or equal to  $Available = [2, 1, 0, 0]$ , we have to run  $C$  first. After  $C$  completes,  $Available = [2, 1, 0, 0] + [0, 1, 2, 0] = [2, 2, 2, 0]$ .

Since only  $B$ 's  $Request = [1, 0, 1, 0]$  is less than or equal to the current  $Available = [2, 2, 2, 0]$ , we run  $B$ . After  $B$  completes,  $Available = [2, 2, 2, 0] + [2, 0, 0, 1] = [4, 2, 2, 1]$ .

Finally, since  $A$ 's  $Request = [2, 0, 0, 1]$  is less than or equal to the current  $Available = [4, 2, 2, 1]$ ,  $A$  can run. Consequently, since an execution sequence  $C, B$  and  $A$  can be found, the system is not in a deadlock state.

See **pp. 262–265** of our text. ■

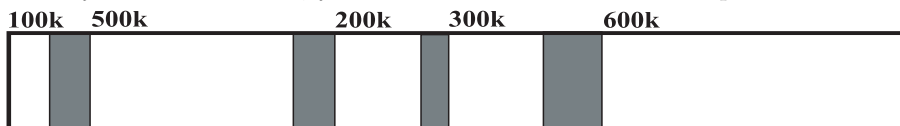
### 5. Memory Management

(a) [6 points] Define *external* and *internal* fragments. Which memory management scheme does not have external fragment? Why? **Note that there are three questions here.** ■

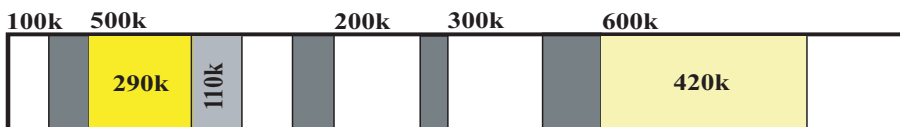
**Answer:** An external fragment is an unused memory block between two allocated memory blocks. An internal fragment is an unused memory area *within* an allocated memory block. The fixed partition scheme does not have external fragments because all partitions are pre-allocated with fixed sizes. However, the fixed partition scheme has internal fragments since a process may not use all the allocated space. Note that paging systems do not have external fragments either.

See **p. 287** of our text. ■

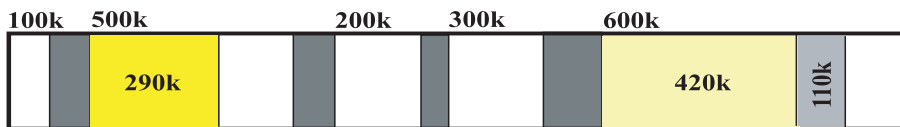
(b) [20 points] Given memory holes (*i.e.*, unused memory blocks) of 100K, 500K, 200K, 300K and 600K (in address order) as shown below, how would each of the *first-fit*, *next-fit*, *best-fit* and *worst-fit* algorithms allocate memory requests of 290K, 420K, 110K and 350K (in this order)? The shaded areas are used/allocated regions that are not available. Write your answer into the following figures. Use shaded areas to indicate unused memory blocks. You should write down the size of each allocated and unused memory block. Otherwise, you will receive **no** credit for that part.



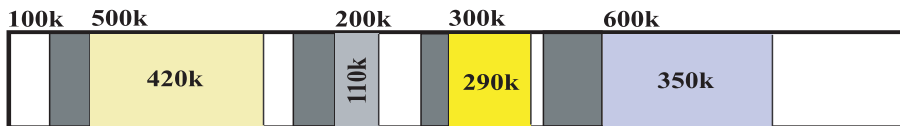
i. [5 points] **First-fit:** The 350K request does not fit.



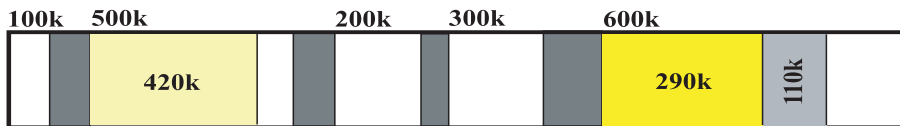
ii. [5 points] **Next-fit:** The 350K request does not fit.



iii. [5 points] **Best-fit:**



iv. [5 points] **Worst-fit:** The 350K request does not fit



(c) [6 points] A paging system uses 16-bit address and 4K pages. The following shows the page tables of two running processes, Process 1 and Process 2. Translate the logical address 16,000 of Process 1 and the logical address 9,000 of Process 2 to their physical addresses. Fill your answers into the table below.

Process 1		Process 2	
0	0	0	3
1	4	1	1
2	5	2	7
3	2	3	6
		4	8

<i>Process</i>	<i>Address</i>	<i>Page #</i>	<i>Offset</i>	<i>Physical Address</i>
Process 1	16,000	3	3,712	11,904
Process 2	9,000	2	808	29,480

- (d) [6 points] Suppose both processes ask for a shared memory of 4K, and suppose further that the system decides to allocate page frame 10 for this purpose. What virtual (or logical) addresses processes Process 1 and Process 2 will receive and what are the new page tables? **You should provide sufficient reasoning. A simple answer will receive no credit.**

**Answer:** Since the system assigns page frame 10 as the shared page, this page frame would go into the next available entry in each page table. Thus, Process 1 (*resp.*, Process 2) has a new page table entry 4 (*resp.*, 5) as shown below.

Process 1	
0	0
1	4
2	5
3	2
4	10

Process 2	
0	3
1	1
2	7
3	6
4	8
5	10

The physical address of this shared page is the beginning of page frame 10. From the point of view of Process 1, page frame 10 is page number 4, and its virtual address is  $4 \times 4096 = 16,384$ . By the same reason, the virtual address of page frame 10 from the point of view of Process 2 is  $5 \times 4096 = 20,480$ . ■

6. **Programming [20 points]**

Each thread in a system has a unique ID, which is a positive integer. The system also has a shared file that can be accessed by multiple threads simultaneously as long as the sum of the ID's of all threads that are currently accessing the file is less than a predefined value **MAXIMUM**.

Design a Hoare monitor **Strange** and monitor procedures **Access(id)** and **Release(id)**, where **id** is the ID of the calling thread. Monitor procedure **Access(id)** allows the caller to access the file if the sum of the all ID's and **id** is less than **MAXIMUM**. In this case, **Access(id)** returns. Otherwise, the caller is blocked. On the other hand, when a thread finishes accessing the shared file, it calls monitor procedure **Release(id)** to release the file.

Use **ThreadMentor** syntax to write the monitor code. It is required to have an explanation/elaboration of your design. Otherwise, you will risk low to very low grade. **Hint:** This problem looks easy; but, it has a tricky portion: what would happen if a blocked thread finds out it still cannot run after it is released? If you handle this situation improperly, the system may end up to have every thread blocked. This is the key issue when I will grade your paper.

**Answer:** This is not a difficult problem; however, as pointed out in the problem statement, there *is* a hidden trap. The following is the class definition. The condition variable **block** blocks those threads that are not allowed to access the file, and **sum** is the sum of the **ids** of those processes that are currently accessing the file.

```

class Strange : public Monitor {
public:
    Strange();           // constructor
    Access(int id);     // monitor procedure Access()
    Release(int id);    // monitor procedure Release()
private:
    Condition block;    // C.V. for blocking threads
    int sum;           // the current sum of PIDs
};

Strange::Strange(): Monitor(HOARE) // constructor
{
    sum = 0;
}

```

Monitor procedure `Access(id)` is the place where you have to pay special attention. The caller can have access if the sum of `sum` and `id` is less than or equal to `MAXIMUM`. Otherwise, the caller blocks. Based on this observation, many may immediately come up with the following code. However, this is not a correct solution. After the caller is released by a signal from another thread, the value that the released thread saw before calling `Wait()` could have been changed by other threads. Thus, this thread has the potential to access the file incorrectly.

```

void Strange::Access(int id)
{
    MonitorBegin();           // enter monitor
    if (sum + id > MAXIMUM) // can I access the file?
        block.Wait();       // block myself since I cannot access
    sum += id;                // Now, I can. Update the sum
    MonitorEnd();            // exit monitor
}

```

A natural solution would be replacing the `if` with `while` so that the caller can loop back and check the condition again as shown below. However, this is still incorrect. Suppose `Release(id)` calls `Signal()` to release a thread. What if the released thread loops back and finds out it cannot access the file? This thread blocks itself again and the monitor becomes empty (*i.e.*, no executing thread) even though one or more threads blocked on `block` can fulfill the condition.

```

void Strange::Access(int id)
{
    MonitorBegin();           // enter monitor
    while (sum + id > MAXIMUM) // can I access the file?
        block.Wait();       // block myself since I cannot access
    sum += id;                // Now, I can. Update the sum
    MonitorEnd();            // exit monitor
}

```

A correct way to solve this problem is to add a signal before the wait as shown below. Hence, if a thread sees the condition is not met, it releases a waiting thread and blocks itself. The released thread can verify the condition again, and access the file if the condition is met. Otherwise, the released thread releases another thread and blocks. In this way, a thread that finds out its condition cannot be met will wake up another thread, and, as a result, all threads will be released one by one to verify the condition. This is usually referred to as *cascaded signal* or *cascaded wake*. The monitor procedure `Release(id)` is easy.

```

void Strange::Access(int id)
{
    MonitorBegin();           // enter monitor
    while (sum + id > MAXIMUM) { // can I access the file?
        block.Signal();      // no, let someone else to try
        block.Wait();       // block myself since I cannot access
    }
    sum += id;               // Now, I can. Update the sum
    MonitorEnd();           // exit monitor
}

void Strange::Release(int id)
{
    MonitorBegin();           // enter monitor
    sum -= id;               // reduce the current sum
    block.Signal();          // allow one of the blocked to try
    MonitorEnd();           // exit monitor
}

```

Some used a counter `count` to count the number of waiting threads and signal that number of times in `Release(id)` as shown below. Unfortunately, this is still not correct. Keep in mind that this is a Hoare type monitor (*i.e.*, the signaler yields). After the first signal call, the caller yields the monitor to the signaled. This released thread goes back to check the condition, and waits if it cannot access the file. The control is then switched back for the second signal. However, since there is no scheduling policy for which thread should be released from condition variable `block`, the worse case could be the same thread released every time, and, as a result, no other threads blocked on `block` would have a chance to check for the condition. Of course, this is not a correct solution.

```

void Strange::Access(int id)
{
    MonitorBegin();           // enter monitor
    while (sum + id > MAXIMUM) { // can I access the file?
        count++;             // increase waiting count
        block.Wait();       // block myself since I cannot access
        count--;             // decrease waiting count
    }
    sum += id;               // Now, I can. Update the sum
    MonitorEnd();           // exit monitor
}

void Strange::Release(int id)
{
    int i;

    MonitorBegin();           // enter monitor
    sum -= id;               // reduce the current sum
    for (i = 1; i <= count; i++)
        block.Signal();      // release all waiting threads
    MonitorEnd();           // exit monitor
}

```

A few of you misinterpreted `MAXIMUM` as the maximum number of threads, and used an array of condition variables `block[]` for blocking threads and an array of flags `flag[]` indicating which thread is waiting, with `id` as an index (*i.e.*, thread `id` waits on condition variable `block[id]` if `flag[id]` is true). Then, `Release(id)` checks each and every element of `flag[]` and signals the corresponding condition variable to release the waiting thread. This is not very correct, because we really do not know the number of threads. In fact, `MAXIMUM` may be smaller than the number of threads, and some thread IDs may be larger than `MAXIMUM`.

Some of you computed the value `sum`, like the following one, before a thread is granted the access. This is also wrong. After the signal call in `Access(id)`, the caller yields the monitor to the released. This released thread will check the condition; however, since the value of `sum` was updated and became larger, the released thread will use an incorrect value of `sum` in its check. Consequently, as more threads are blocked, the value of `sum` increases and eventually many threads will use the wrong `sum` value in their condition tests until some threads start to decrease the value of `sum`.

```
void Strange::Access(int id)
{
    MonitorBegin();           // enter monitor
    sum += id;                // compute the new sum
    while (sum > MAXIMUM) { // can I access the file?
        block.Signal();      // no, let someone else to try
        sum -= id;           // restore the sum value
        block.Wait();        // block myself since I cannot access
    }
    MonitorEnd();            // exit monitor
}
```

This is a simple problem that can test if you have understood the basic merit of a monitor, its type, and the semantics of the signal and wait methods. If you are not careful, your program could be incorrect. ■