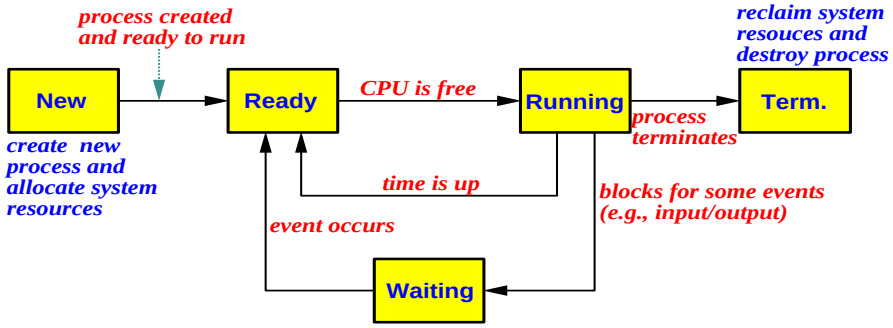
CS4411 Intro. to Operating Systems Exam 2 Solutions Fall 2009

1. **Recycled Problems**

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

Answer: The following state diagram is taken from my class note and was discussed in class. Fill in the elaboration for each state and transition by yourself.



See **p. 103** of our text and class notes. ∎

(b) **[7 points]** Define the meaning of a *race condition*? Answer the question first and use an execution sequence to illustrate your answer. **You will receive no credit if only an example is provided without an elaboration.**

Answer: A *race condition* is a situation in which *more than one* processes or threads access a shared resource *concurrently*, and the result depends on the order of execution.

The following is a simple counter updating example discussed in class. The value of `count` may be 9, 10 or 11, depending on the order of execution of the **machine instructions** of `count++` and `count--`.

```
int         count = 10;

Thread_1(...)              Thread_2(...)
{                          {
   // do something             // do something
   count++;                    count--;
}                          }
```

The following execution sequence shows a race condition. Two threads run concurrently (condition 1). Both threads access the shared variable `count` at the same time (condition 2). Finally, the computation result depends on the order of execution of the `SAVE` instructions (condition 3). The execution sequence below shows the result being 9; however, switching the two `SAVE` instructions yields 11. Since all conditions are met, we have a race condition.

`Thread_1`	`Thread_2`	*Comment*
do somthing	do somthing	`count = 10` initially
`LOAD count`		`Thread_1` executes `count++`
`ADD #1`		
	`LOAD count`	`Thread_2` executes `count--`
	`SUB #1`	
`SAVE count`		`count` is 11 in memory
	`SAVE count`	Now, `count` is 9 in memory

Stating that "`count++` followed by `count--`" or "`count--` followed by `count++`" would produce different results and hence a race condition is incorrect, because the threads do not access the shared variable `count` at the same time (*i.e.*, condition 2).

See **p. 193** of our text and class notes. ■

2. **Synchronization**

(a) **[8 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 has the details:

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

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

(b) **[8 points]** What are the differences between a Hoare type monitor and a Mesa type monitor? Vague answers and/or inaccurate or missing elaboration receive **no** credit.

Answer: The major difference is the way of releasing waiting threads. With a Hoare monitor, the signaling thread yields the monitor, allowing the released thread to execute immediately. With a Mesa monitor, the signaling thread continues and the released thread becomes *inactive*. The released thread will run sometime later when the monitor becomes empty. Because of this, Hoare type monitors require two context switches, while Mesa type monitors need only one. Moreover, context switching in a Hoare type monitor must be handled properly, making sure that the signaling thread is switched out and the released thread is switched in immediately to use the monitor.

See ThreadMentor web page and class notes. ■

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.*, I/O completion).

See **pp. 153–154** of our text. ■

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

	CPU Burst	*Priority*
A	4	2
B	6	4
C	2	1
D	5	3
E	3	5

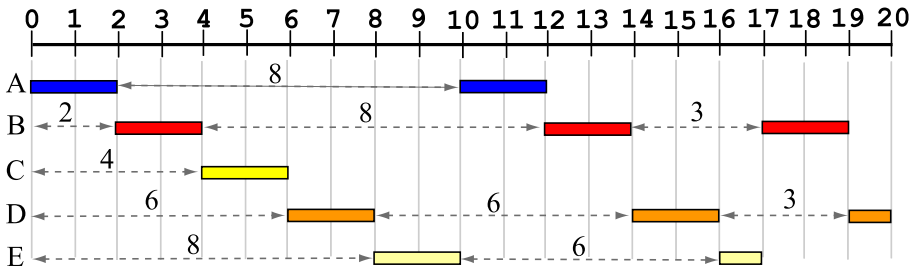
Fill the entries of the following table with waiting time and average waiting time for each indicated

scheduling policy and each process. Ignore context switching overhead.

Scheduling Policy	*Waiting Time*					*Average*
	A	*B*	*C*	*D*	*E*	*Waiting Time*
First-Come-First-Served						
Non-Preemptive Shortest-Job First						
Priority						
Round-Robin (time quantum=2)						

Answer:

Scheduling Policy	*Waiting Time*					*Average*
	A	*B*	*C*	*D*	*E*	*Waiting Time*
First-Come-First-Served	0	4	10	12	17	43/5 = 8.6
Non-Preemptive Shortest-Job First	5	14	0	9	2	30/5=6
Priority	2	11	0	6	17	36/5=7.2
Round-Robin (time quantum=2)	8	13	4	15	14	54/5=10.8



The above diagram shows the execution pattern of the round-robin algorithm with time quantum 2, where dashed arrows indicate waiting periods.

See class notes for the details. ∎

4. **Deadlocks**

(a) **[8 points]** What are the necessary conditions for a deadlock to occur? Name these conditions and provide an elaboration. Stating conditions without elaboration or stating 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, ..., P_n exits 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. 285–287** of our text. ∎

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

	Allocation				*Max*				*Need*				*Available*			
	U	V	W	X	U	V	W	X	U	V	W	X	U	V	W	X
A	1	1	0	2	1	1	0	3	0	0	0	1	3	4	0	0
B	0	1	2	3	1	2	4	5	1	1	2	2				
C	5	1	4	4	5	5	9	5	0	4	5	1				
D	0	0	1	1	5	0	1	1	5	0	0	0				
E	1	0	2	1	4	3	2	1	3	3	0	0				

Is this 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 (*i.e.*, banker's algorithm). Note that we always search for a candidate in the order of A, B, C, D and E.

- Since *Available* = [3, 4, 0, 0] is greater than E's *Need*=[3,3,0,0], E can run. After E completes, *Available* = [3,4,0,0]+[1,0,2,1]=[4,4,2,1].
- Since *Available*=[4.4.2.1] is greater than A's *Need*=[0,0,0,1], A can run. After A completes, *Available*=[4,4,2,1]+[1,1,0,2]=[5,5,2,3].
- Since *Available*=[5,5,2,3] is greater than B's *Need*=[1,1,2,2], B can run. After B completes, *Available*=[5,5,2,3]+[0,1,2,3]=[5,6,4,6].
- Since *Available*=[5,6,4,6] is greater than D's *Need*=[5,0,0,0], D can run. After D completes, *Available*=[5,6,4,6]+[0,0,1,1]=[5,6,5,7].
- Since *Available*=[5,6,5,7] is greater than C's *Need*=[0,4,5,1], C can run.

Therefore, if the five processes are run in the order of E, A, B, D and C, all of them can finish and the system is safe (*i.e.*, $< E, A, B, D, C >$ is a safe sequence). Note that safe sequence is not unique. See **pp. 256–260** of our text. ∎

(c) **[8 points]** Consider the following snapshot of a system with four resource types R_1, R_2, R_3 and R_4, and four processes A, B, C and D:

	Allocation				*Request*				*Available*			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
A	1	0	1	1	0	1	0	0	0	0	0	0
B	1	1	0	0	0	1	1	0				
C	1	0	1	1	0	0	0	0				
D	0	0	0	0	1	0	1	0				

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

Answer: Since *Available* = $[0, 0, 0, 0]$, only C can run as its request is $[0, 0, 0, 0]$. After C completes, it returns its allocation $[1, 0, 1, 1]$, making the new *Available* = $[1, 0, 1, 1] = [0, 0, 0, 0] + [1, 0, 1, 1]$. Now, D can run because its request is $[1, 0, 1, 0]$, which is smaller than *Available* = $[1, 0, 1, 1]$. After D's completion, it returns its allocation $[0, 0, 0, 0]$. As a result, the *Available* is still $[1, 0, 1, 1]$. At this time, neither A nor B can run, because both A and B require one R_2 which is not available. Therefore, the system is in a deadlock state, and the involved processes are A and B. ∎

5. **Memory Management**

(a) **[8 points]** Define *external* and *internal* fragments. Consider the following memory management schemes: fixed-size partitions, variable-size partitions, and paging. Which schemes have external fragments, and which schemes have internal fragments? Why? **Note that there are <u>three</u> questions. Elaborate your answer. Otherwise, you will receive <u>no</u> credit.**

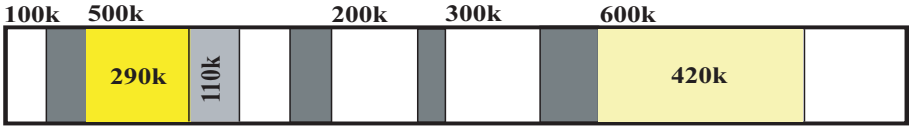
Answer: An *external* fragment is an unused memory block *between* two allocated (*i.e.*, used) ones. An *internal* fragment is an unused memory area *within* an allocated memory block. Fixed-size

partitions and paging do not have external fragments because all partitions and page frames are pre-allocated with fixed sizes. However, they may have internal fragments since a process may not use all the allocated space. Variable-size partitions do not have internal fragment; but, they have external fragments. Note that even though the variable-size partition scheme may allocate a bit more memory than requested, say to fit the boundary alignment requirement, we still consider its allocation being exact.

See **p. 327** of our text. ■

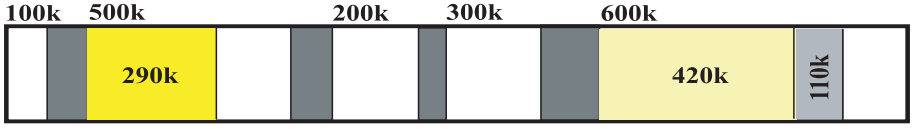
(b) **[16 points]** Given memory holes (*i.e.*, unused memory blocks) of 100K, 500K, 200K, 300K and 600K (in address order) as shown below, how would each of the *first-fit*, *next-fit*, *best-fit* and *worst-fit* algorithms allocate memory requests for 290K, 420K, 110K and 350K (in this order)? The shaded areas are used/allocated regions and are not available. Write your answer into the following diagrams. You should clearly write down the size of each memory block and indicate its status (*i.e.*, allocated or free). Otherwise, you will receive **no** credit for that part.

i. **First-fit:**



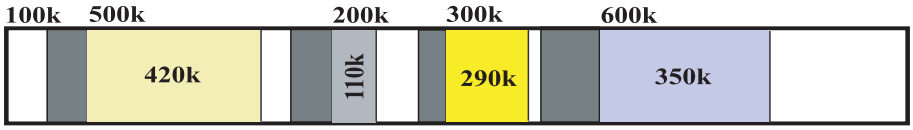
The 350K allocation request does not fit.

ii. **Next-fit:** The current area is the 100K region.

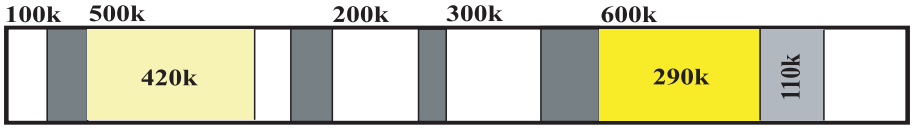


The 350K allocation request does not fit.

iii. **Best-fit:**



iv. **Worst-fit:**



The 350K allocation 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 addresses in the table below to their corresponding physical addresses, and fill the table entries with your answers.

	Process 1
0	3
1	7
2	1
3	5

	Process 2
0	2
1	0
2	6
3	4

Process	*Address*	*Page #*	*Offset*	*Physical Address*
Process 1	11,034			
Process 2	12,345			

Answer: Consider the logical address 11034 generated by process 1. Since page size is 4K = 4096, logical address 11034 is in page 2 = 11034/4096, and the offset is 11034 - 2× 4096 = 2842. From process 1's page table, page 2 is in page frame 1, and, hence, the corresponding physical address is 1× 4096 + 2842 = 6938. The second logical address 12345 is translated the same way with process 2's page table.

Process	*Address*	*Page #*	*Offset*	*Physical Address*
Process 1	11,034	2	2842	6,938
Process 2	12,345	3	57	16,441

(d) [**8 points**] Construct an inverted page table from the system snapshot of the page tables in Problem (5c).

Answer: Each entry in an inverted page table has two fields: the owner (*i.e.*, process ID) and the page number of the owner. From the two given page tables, the corresponding inverted page table is

	Process ID	*Page Number*
0	2	1
1	1	2
2	2	0
3	1	0
4	2	3
5	1	3
6	2	2
7	1	1

6. **Programming**

(a) [**25 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 until the condition will be met in the future. 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. **You must elaborate and justify your solution. Otherwise, you will receive low or even no grade.**

Answer: This is not a difficult problem; however, there *is* a hidden trap. The following is the class definition. Condition variable `block` blocks those threads that are not allowed to access the file, and `sum` is the sum of the `id`s 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 would immediately come up with the following code, which, unfortunately, is not a correct solution. After the caller is released by a signal from another thread, it does not go back and makes sure if the condition still holds, and, hence, 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 way would be replacing the `if` with `while` so that the caller loops back and checks the condition again as shown below. However, this is still incorrect. Suppose `Release(id)` signals 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
}
```

The following is a possible solution. If the condition is not met, the caller 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. Since this is a Hoare type monitor, the chain of events "check ⇒ `Signal()` ⇒ yield ⇒ `Wait()`" will occur to every thread blocked on condition variable `block`. 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 may switch the order of `block.Wait()` and `block.Signal()` like the one below. This is not a "good" solution. Since the thread that calls `Signal()` yields the monitor to the released thread, this version will release "all" threads blocked on condition variable `block` before they can do any testing. In other word, the chain of events "`Wait()`⇒`Signal()`⇒`yield` ⇒ check" will release every thread blocked on condition variable `block`. After the last blocked thread is released, executes `Signal()` and yields, the monitor becomes empty, and either a released thread runs or a new thread enters the monitor. In the latter case, the newcomer can cut in and change the condition, and, as a result, every released thread could be blocked again. Compared with the above solution, this one is certainly not good enough.

```
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
               block.Signal();   // no, let someone else to try
          }
          sum += id;             // Now, I can.  Update the sum
     MonitorEnd();               // exit monitor
}
```

Some used a counter 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). The first signal call forces the caller to yield the monitor to the released. 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
}
```

Some of you computed the value of `sum`, like the following, 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 checks 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 can easily become an incorrect one. ■