CIS 505 - Spring 2007
Midterm Solutions

Mar 19, 2007
(Draft 1)

There are nine problems and each problem is worth 10 points. The maximum score you can get for this exam is **70**, roughly one point for one minute. Please pick any **seven** of them. If you answer more than seven problems, we will pick any seven of them for grading unless you state explicitly which ones should be graded.

Please write your answers **legibly** so that we can understand your answers. If a problem seems ambiguous or impossible, please feel free to state your assumptions explicitly and solve the problem. Obviously, your assumptions should be reasonable and should not trivialize the problem.

Please sign the **pledge** at the back of the answerbook. **Good luck!**

1. (10 points)
Consider the following predence constraints that must be sastified for the execution of processes P1, P2, P3, P4 and P5:

- P1 can start any time.
- P2 can start after P1 completes.
- P3 can start after P1 completes.
- P4 can start after both P2 and P3 complete.
- P5 can start after P3 completes.

(a) (3 points) Draw a precedence diagram that captures the above constraints, where a node represents a process and a directed edge from process A to process B means that process B should not start until process A is completed.

Answer:

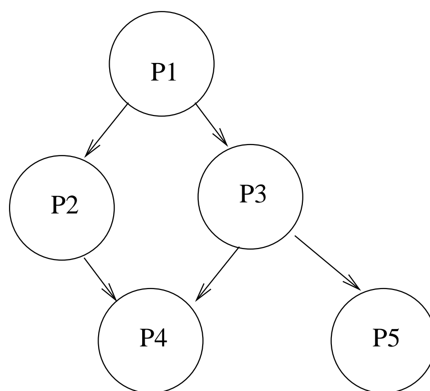

Figure 1: Precedence Diagram

(b) (3 points) Using semaphores, explain how these precedence constraints can be enforced. Make sure to explain what is the initial value of each semaphore. (Note: this solution need not use a mininum number of semaphores.)

Answer: We make use of three semaphores: S1 , S2 , S3. All semaphores are initialized to 0. Figure 2 illustrates how the semaphores can be used to enforce precedence constraints. Note that other solutions are possible.

P1	P2	P3	P4	P5
P1 code	down(S1)	down(S1)	down(S2)	down(S3)
up(S1)	P2 code	P3 code	down(S2)	P5 code
up(S1)	up(S2)	up(S2)	P4 code	
		up(S3)		

(c) (4 points) What is the minimum number of semaphores needed to solve the problem? Show your solution and justify your answer. (Note: this may be different than the one your provided for (b).)

Answer:

Here's a solution for two semaphores:

P1	P2	P3	P4	P5
P1 code up(S1) up(S1)	down(S1) P2 code up(S1) up(S1)	down(S1) P3 code up(S2) up(S2)	down(S2) down(S1) down(S1) P4 code	down(S2) P5 code

We gave partial credit (2 / 4) for solutions using three semaphores.

2. (10 points)

   (a) (4 points)
   What is the priority inversion problem? Also, describe using an example.

   Answer:

   Suppose there are three processes: high priority process P , low priority process Q and medium-priority process R. While the process P is blocked waiting for the low priority process Q, if the execution of Q is preempted by process R (which has higher priority than Q, but lower priority than P , then we say that a priority inversion is occured (because the high-priority process P is effectively being preempted by medium-priority process R).

   (b) (6 points) Consider the following four periodic real-time tasks (period, execution-time, deadline within a current period): (10, 3,10), (9, 2, 9), (8,1,4), (7,2,7). Assume that they are all released at time 0.

   (i) Show the actual schedule using RM (Rate-Monitonic).

   RM scheduling leads to a deadline miss. Figure 2 shows the details.

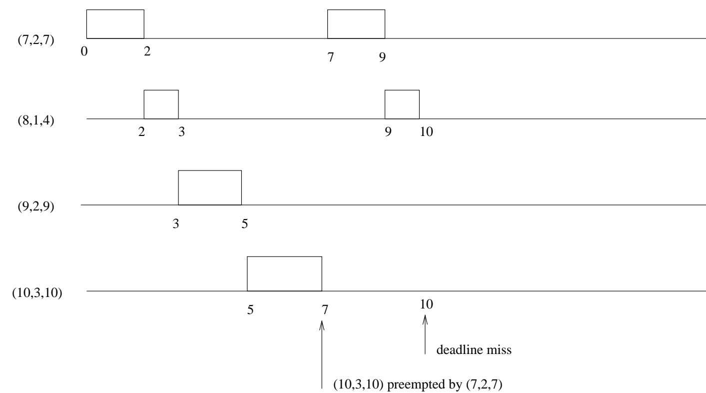

Figure 2: RM Schedule

   (ii) Show the actual schedule using EDF (Earliest-Deadline-First).

   See Figure 3.

3. (10 points) Answer True/False with short justification, or Don't know: 2 points for correct answer, .5 points for no answer, and -1 point for incorrect answer.

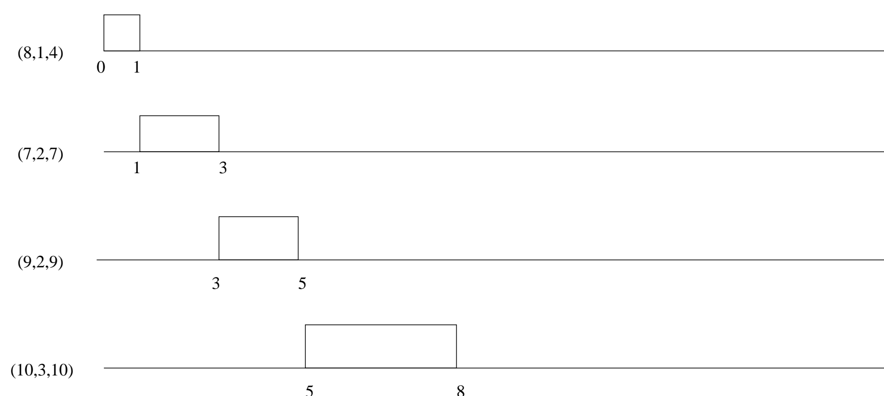

Figure 3: EDF Schedule

(a) User-level threads are faster to switch among than kernel-supported threads.
True because they do not involve kernel.

(b) One limitation of kernel-supported threads is that when a thread blocks, the kernel cannot schedule another thread from the same process.
False since each thread has its own stack and register set, no kernel boundary crossing.

(c) In a distributed system, it is difficult to detect deadlock since the deadlock can disappear while processes try to determine whether there is deadlock or not. This can happen due to unpredictable communication delays.
False because once deadlock occurs, the system is always deadlocked.

(d) The only cause of the semantic difference between RPC and local procedure call is due to communication delays since both are based on the same programming technique.
False because communication delay and processor failures cause differences for RPC semantics.

(e) Priority inversion cannot occur if processes (or tasks) can never be blocked.
True

4. (10 points) Answer True/False with short justification, or Don't know: 2 points for correct answer, .5 points for no answer, and -1 point for incorrect answer.

(a) User-level threads are more efficient since when a thread within a process is blocked waiting for I/O, another thread within the same process can execute.
False, user-level threads mean waiting on I/O blocks the whole process since OS does not know about threads.

(b) Priority inversion cannot happen if there are only two processes in the system.
True, since priority inversion needs at least three processes to occur.

(c) Consider The following two CSP processes
$\alpha P = \{up, down\}$
$\alpha Q = \{down\}$

$P = (up \rightarrow P | down \rightarrow P)$,
$Q = (down \rightarrow Q)$,

The following equation is true:
$(P \parallel Q) = (up \rightarrow (P \parallel Q)) | (down \rightarrow (P \parallel Q))$

True, since "up" need not be synchronized.

(d) Consider The following two CSP processes
$\alpha P = \{up, down\}$
$\alpha Q = \{up, down\}$
$P = (up \rightarrow P | down \rightarrow P)$
$Q = (down \rightarrow Q)$

The following equation is true:
$(P \parallel Q) = (up \rightarrow (P \parallel Q)) | (down \rightarrow (P \parallel Q))$

False, since "up" need to be synchronzied.

(e) Consider The following two CSP processes
$\alpha P = \{up, down\}$
$\alpha Q = \{down\}$
$P = (up \rightarrow P | down \rightarrow P)$
$Q = (down \rightarrow Q)$

The following equation is true:
$(P \parallel Q) = (down \rightarrow (P \parallel Q))$

True, since "up" need to synchronized.

5. (10 points)

   (a) (6 points)
   "Nested monitor call" refers to a situation in which a process executing inside a monitor makes a call to another monitor.

   (i) Suppose that the mutual-exclusion lock is released as it leaves the current monitor and is reacquired only when it returns from the called monitor. For example, a process P executes procedure A in monitor M, which calls procedure B in monitor N. Here, the lock of minitor M is released as it enters procedure B and the lock of monitor M is reaquired when the process P returns to the monitor M. Prove that deadlock cannot happen or provide a counterexample with deadlock.

   Answer: In the example above, suppose procedure B includes a call to procedure A. When process P runs, it starts with a call to A, aquiring a lock to M. When A calls B, P releases the lock to M and aquires the lock to N. Then when B calls A, the lock to N is released and the lock to M is aquired. P will continue to recurse infinitely, or in a real system, until resources are exhausted.

   (ii) Instead of releasing the lock, suppose the lock is retained while it leaves the current monitor. Again, prove that deadlock cannot happen or provide a counterexample with deadlock.

   Answer Suppose a process P executes procedure A in monitor M, which calls procedure B in monitor N. Now if procedure B calls procedure A, P will not be able to lock monitor M and the system will deadlock.

   (b) (4 points)
   Describe the difference between a system call and a library call? Also, provide one example for each.

   Answer: A library call is a user level function which may use system calls. A system call is a function which is not usually called directly by users and which directly accesses some low-level kernel functionality.

   sbrk(size) is a system call which increases the size of the heap space of the calling process by size.

   malloc(size) is a library call which allocates size bytes from the heap.

6. (10 points)

   (a) (6 points)
   A global state, GS, of a system is a collection of the local states of its sites. That is, $GS = \{LS_1, LS_2, \ldots, LS_n\}$ where n is the number of sites in the system and LS_i is a local state of site i.

   (i) Define what it means to say that a global state is inconsistent.

   Answer: A global state is said to be inconsistent if L_{Sp} and L_{Sq} in GS have the following states. Process P records that it received a message from Process Q, but Q does not have any record show that it ever sent this message to P.

   (ii) Give an example of such an inconsistent global state of the system consisting of three processes plus coordinator, where the coordinator is to detect the existence of deadlock using centralized algorithm.

Here is an example of such a state. The dashed line indicates the collected time of local states.

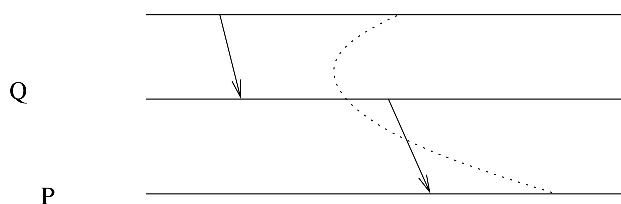

Figure 4:

(b) (4 points)
One of the reasons why it is difficult to develop distributed systems is due to communication, which can be unreliable and have unpredictable delays. Suppose it is possible to build an infinitely fast, ultra-reliable network that always delivers messages in zero time. How will this help to build distributed systems? What problems/difficulties will still remain?

Answer: An infinitely fast, ultra-reliable network will help build distributed systems. Specifically, protocols need not consider out-of-order delivery of packets as well as lost messages. Synchronization will be easier, since only processing delay needs to be considered (as opposed to processing delay and network delay). There are several problems/difficulties that will remain for the fast, reliable network. For example, failure transparency is particularly problematic for RPC calls, since the execution of the procedure occurs on a remote machine (i.e., detecting failures, exceptions, etc. remains difficult). In addition, scalability still poses a problem as processing power is finite.

7. (10 points)

(a) (5 points)
Describe Lamport's logical clock that preserves *happen-before* relations. Explain why it is not a total ordering. Explain how it can be extended to a total ordering. Provide an example where such a total ordering can be used.

Answer: Lamports logical clock preserves the happens-before relation. Each process has its own clock that ticks according to some rate. When a process transmits a message, it includes its current timestamp in the message. When a message arrives, the receiving process compares the timestamp in the message with its current timestamp. If the timestamp of the message is greater than its current timestamp, then it sets its current timestamp to the one specified in the message plus one. Since two processes can have the same timestamp value, Lamports logical clock does not provide a total ordering. A total ordering can trivially be accomplished by attaching the processor id (which we assume to be unique) to the low-order end of the time, separated by a decimal point. For example, if some event at time 40 occurs in processes 1 and 2, then the respective timestamps become 40.1 and 40.2.

(b) (5 points)

Show with an example that, with Lamport's logical clock, if $LC(e_1) < LC(e_2)$, it is not neccessarily true that e_1 happened before e_2. Vector clocks are developed to fix this problem. Explain how vector clocks work. Provide an example where a vector clock can be used.

Answer: Figure 7b shows that with Lamports logical clock, if $LC(e_1) < LC(e_2)$, it is not necessarily true that $e_1 \rightarrow e_2$. For example, let e_1 be the sole event for P3. Hence $LC(e_1) = 3$. Let e_2 be the last event for P 2. Hence $LC(e_2) = 4$. We note that although $LC(e_1) < LC(e_2)$, it is not true that $e1 \rightarrow e2$.

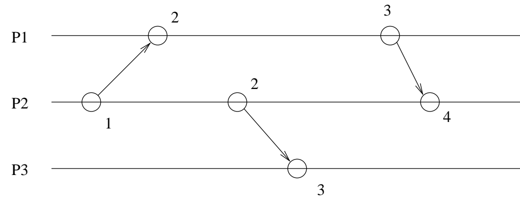

Figure 5: Logical Clocks

With vector clocks, each process attempts to keep track of the timestamp of all other processes. In a system with n processes, each process contains a vector of size n. Vector element $V_i[k]$ corresponds to process is knowledge of the Lamport clock operating on process k. When processes transmit messages, they include the contents of their vectors. When process j receives a message from process i, it computes $V_j[k] = max(V_j[k], V_i[k])$ for all $1kn, k \neq j$. Similar to Lamports clocks, when process i receives or sends a message, it increments $V_i[i]$.

8. (10 points)

   (a) (2 points)
   Prove or disprove that
   $P \parallel RUN_{\alpha P} = P$
   Answer:

$$
\begin{aligned}
\text{traces}(P \parallel RUN_{\alpha P}) &= \{t | t \uparrow \alpha P \in \text{traces}(P) \wedge t \uparrow \alpha P \in \text{traces } (RUN_{alphaP}) \wedge t \in (\alpha P)^*\} \\
&= \{t | t \uparrow \alpha P \in \text{traces}(P) \wedge t \in (\alpha P)^*\} \\
&= \{t | t \in \text{traces}(P)\} \\
&= \text{traces}(P)
\end{aligned}
$$

   (b) (2 points)
   Prove or disprove that
   $P \parallel P = P$
   Answer:

$$\begin{aligned}\text{traces}(P \parallel P) =& \{t | t \uparrow \alpha P \in \text{traces}(P) \wedge t \in (\alpha P)^*\} \\ =& \{t | t \in \text{traces}(P) \wedge t \in (\alpha P)^*\} \\ =& \text{traces}(P)\end{aligned}$$

Hence, $P \parallel P = P$

(c) (2 points)
Prove or disprove the following statement:
Suppose there are two processes Q and Q' such that, for every process, P, we have $P \parallel Q = P \parallel Q' = R$. Then, it must be the case that $Q = Q'$.
Answer:

$$\begin{aligned}&\text{Let } P = Q \\ &\qquad \Rightarrow P \parallel Q = Q = Q \parallel Q' \\ &\text{Let } P = Q' \\ &\qquad \Rightarrow P \parallel Q' = Q' = Q \parallel Q' \\ &\qquad \Rightarrow Q = Q'\end{aligned}$$

(d) (4 points)
Described advantages of iterative name resolution over recursive name resolution.
Answer:

- Reduced load on servers because they are not responsible for fetching IP addresses from other servers.
- If servers are not local with respect to each other, then communication between them will be as expensive as communication between the client and servers. In this case, recursive name resolution does not reduce communication overhead.
- In iterative name resolution, the client can cache IP addresses of individual servers. Hence, any names that use these servers can be resolved faster. In recursive name resolution, the client is only aware of the IP address of the entire name that was resolved. Hence, it may not have the IP addresses of intermediate servers.

9. (10 points)
Solve the Dinning Philosophers Problem with five philosophers using Hoare's Monitors or CSP. For CPS, you can use either CSP programming language or CSP algebra. Explain/justify why your solution (i) does not deadlock and (ii) also that it is fair in the sense that if a philosopher wants to eat, it will eventually be allowed to eat. For the latter property (ii), please be explicit about any assumptions that you are using in your argument.

Answer:

Consider the 5 philosophers and forks as shown in Figure 9, where P_i denotes a philosopher and S_i denotes a fork.

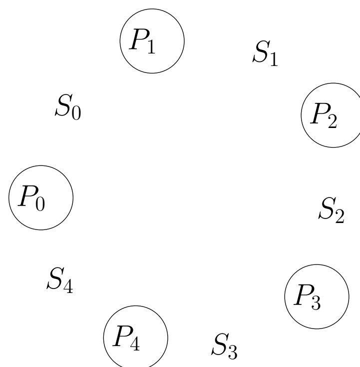

Figure 6: Dining Philosophers

- **Using CSP:**

  Let p_i denote the event that fork S_i has been picked, e denote the event that the philosopher is eating and k_i denote the event that fork S_i has been returned. Each fork process is as follows:

$$\begin{aligned} S_i =& P_i!0 \longrightarrow P_i?x \longrightarrow P_{(i+1) \bmod 5}!0 \longrightarrow P_{(i+1) \bmod 5}?x \longrightarrow S_i \\ & | P_{(i+1) \bmod 5}!0 \longrightarrow P_{(i+1) \bmod 5}?x \longrightarrow P_i!0 \longrightarrow P_i?x \longrightarrow S_i \end{aligned}$$

  Note, that each fork process alternates between the two philosophers that use it. This ensures fairness.

  Each even philosopher (P_0, P_2, P_4) is as follows:

$$\begin{aligned} P_i =& S_i?x \longrightarrow p_i \longrightarrow S_{(5+i-1) \bmod 5}?x \longrightarrow p_{(5+i-1) \bmod 5} \longrightarrow e \longrightarrow \\ & k_i \longrightarrow S_i!0 \longrightarrow k_{(5+i-1) \bmod 5} \longrightarrow S_{(5+i-1) \bmod 5}!0 \longrightarrow P_i \end{aligned}$$

  Each odd philosopher (P_1, P_3) is as follows:

$$\begin{aligned} P_i =& S_{i-1}?x \longrightarrow p_{i-1} \longrightarrow S_i?x \longrightarrow p_i \longrightarrow e \longrightarrow \\ & k_{i-1} \longrightarrow S_{i-1}!0 \longrightarrow k_i \longrightarrow S_i!0 \longrightarrow P_i \end{aligned}$$

  Note, that even philosophers first pick their left forks and odd philosophers first pick their right forks. Hence, they will never deadlock.

- **Using Monitors:**

  Each fork monitor is as follows:

  S_i: **monitor**

  begin

  busy: Boolean;

nonbusy: condition;
Procedure: acquire;
begin
if busy **then** nonbusy.wait;
busy = true;
end // End acquire
Procedure: release;
begin
busy = false;
nonbusy.signal;
end // End release
busy = false; // Initialization of monitor variable
end

Each even philosopher (P_0, P_2, P_4) is as follows:
P_i: **monitor**
begin
Procedure: eat;
begin
S_i.acquire();
$S_{(5+i-1) \text{ mod } 5}$.acquire();
eat sphagetti;
S_i.release();
$S_{(5+i-1) \text{ mod } 5}$.release();
end
end

Each odd philosopher (P_1, P_3) is as follows:
P_i: **monitor**
begin
Procedure: eat;
begin
S_{i-1}.acquire();
S_i.acquire();
eat sphagetti;
S_{i-1}.release();
S_i.release();
end
end
Deadlock is avoided because even philosophers first pick their left forks and odd philosophers first pick their right forks. Fairness is guaranteed because monitors ensure that

when a process signals another process after leaving a monitor, no third process can enter the monitor inbetween.