Oct 30, 2006
Lecture 10: Deadlocks

November 6, 2006

1 ADMIN

Please read Chapter 8 on deadlocks.

2 REVIEW

- Q: As CPU's become faster (relative to I/O devices), this makes CPU scheduling more critical. Argue both sides.

  A: If CPU are so fast, no matter how we poorly we schedule, the jobs still get completed in reasonable time.

  As CPU becomes faster, each minor unnecessary delay translates into mega-cycles of CPU.

- Q: Give a method to give scheduling priority to shorter processes.

  A: Use Quantum Queues $Q_0, Q_1, Q_2, \ldots$: Jobs in Q_i will be given 2^i quanta. Initially, all jobs are put in Q_0. When a job from Q_i is preempted at the end of its allocated time, it is put into Q_{i+1}. Jobs in Q_i has higher priority than jobs in Q_{i+1}.

- Q: To give scheduling priority to IO-Bound processes, we need some way estimating the probability that the process is IO-Bound. How can we do this?

3 DEADLOCKS

- WHAT IS THE DEADLOCK PROBLEM?

  1. Deadlock Condition: a set of processes, each waiting on an event that only another in the set can cause.
  2. Example: Dining Philosophers.
  3. System Deadlock: no process in the system can may progress. (CONTRAST: a subset of processes may be locked out, but the system is not deadlocked)

- Process-Resource Model:

  1. Each process may need one or more of a resource.
  2. Physical Resource Examples: CPU, monitor, printer, scanner, CD recorder, plotter, tape drive, etc.
  3. Logical Resources Examples: files, semaphores, monitors, etc.
  4. Some resouces are exclusive.
  5. Some resources are preemptable but some are not.

6. There may be many copies of the same resource.
7. Some process may need several resources simultaneously.
8. PROTOCOL: a process must REQUEST a resource before use, and must RELEASE resource after use.

- 4 NECESSARY AND SUFFICIENT CONDITIONS FOR DEADLOCK (Coffman et all (1971):

    1. (MUTUAL EXCLUSION) Processes require EXCLUSIVE use of resources
    2. (HOLD AND WAIT) Processes requests more than ONE resources, and must WAIT until all requests have been granted.
    3. (NON-PREEMPTION) Use of resource is non-preemptable (e.g., printer, CD burner)
    4. (CIRCULARITY) Two or more processes are waiting in a cyclic fashion: process P_i waits for $P_{(i+1) \bmod n}$ for $i = 0, 1, \ldots, n-1$.

    REMARK: this is an if and only if list of conditions.

- Holt's RESOURCE GRAPHS are bipartitite dgraphs with 2 kinds of nodes:

    1. Circles= processes, Squares=resources
    2. (circle P $\rightarrow$square R): process P is blocking on resource R.
    3. (square R $\rightarrow$process P): resource R is currently used by process P.
    4. DEADLOCK: cycle
    5. How to draw bipartite graph.

    Note that this is a runtime graph.

- Example (Figure 3-4):

    1. A: Request R, Request S, Release R, Release S.
    2. B: Request S, Request T, Release S, Release T.
    3. C: Request T, Request R, Release T, Release S.
    4. SHOW THE HOLT GRAPH after each step of the following execution sequence:
    5. A request R, B request S, C request T,
    6. A request S, B request T, C request R.

- 4 APPROACHES TO DEADLOCKS

    1. Ostrich Algorithm: Ignore (Unix, Windows)
    2. Detect and Recover (reboot?)
    3. Dynamic avoidance: careful resource allocation
    4. Prevention: negate one of the 4 conditions for deadlock.

- Detection of Deadlock in the Simple Case:

    1. Simple Case means this: each resource is unique (single copy).
    2. In the simple case, we have deadlock IF and ONLY IF there is a cycle in the resource graph.
    3. ALGORITHM: use depth first search from EACH node of the graph.
    4. MORE PRECISELY: initially all nodes and edges are marked "unseen". A driver loop will run a dfs search from each unseen node.

        When a node or edge is first seen, it is marked "seen". However, when all the outgoing edges of a node have been marked "seen", the node is marked "done".

        When we first traverse an edge, we check to see if the node at the other end of the edge is "unseen", "seen" or "done". If unseen, we extend our recursive search to this node; if seen, we have detected a cycle; if "done", we back up from our DFS search.

5. REMARK: This algorithm is superior to the one in Tanenbaum text. It has running time $O(m+n)$ where the digraph has m edges and n nodes.

- Generalization to Multiple Copies of Resources.

  1. QUESTION: Why can't the general case of multiple copies be reduced to this one? E.g., if there are 5 copies of resource r1, just duplicate r1 five times. ANSWER: You could, but it does not truly reflect the fact that these 5 copies are interchangeable. So, you lose something in this reduction to the simple case.
  2. Assume n processes and m resources.
  3. Processes are $p_1, \ldots, p_n$, resources are $r_1, \ldots, r_m$.
  4. We know the **Maximum Request Matrix** M is a $n \times m$ matrix, where indicating that p_i never request more than M_{ij} copies of r_j.
  5. There is a total of $e_j \geq 0$ copies of r_j $(j = 1, \ldots, m)$ Let $E = (e_1, \ldots, e_m)$.
  6. Let $F = (f_1, \ldots, f_m) \leq E$ be the **free vector**. This can determined from A and E. In fact, $f_j = e_j - \sum i = 1^n A_{ij}$.
  7. The **Current Allocation matrix** A is a $n \times m$ matrix, where p_i holds A_{ij} copies of r_j.
  8. Define the **Claim Matrix** C which is simply $M - A$. Thus C_{ij} is the number of units of r_j that p_i has a claim on.
  9. Depending on the problem, we may also have a queue Q of **Pending Requests**.
  10. PROBLEM FORMULATION: the ith process can, at any time, release ALL of its allocated resources, or request a vector $B = (b_1, \ldots, b_m)$ of resource. We require $b_j \leq C_{ij}$ for $j = 1, \ldots, m$. Our algorithm has to respond to each request for resources, or release of resources. When a request is made, we either fulfil it (and update A by incrementing the ith row with vector B), or put it on hold (by placing the vector B in the Q). When resources are released, we check to see if any pending requests in Q can be fulfilled.
  11. STATE: this is defined as the the current allocation matrix A.
  12. ASSUMPTION: a process will terminate in finite time when all its requests are satisfied.
  13. A state is **safe** if eventually all requests currently holding resources can terminate.
  14. GENERALIZED RESOURCE GRAPHS: In the text (section 8.5.2), a graphical representation of the state is given by introducing a third kind of edge, a "claimed edge".

- Banker's Algorithm:

  1. Before trying to solve the general problem, suppose there is only ONE kind of resource (call it MONEY).
  2. In other words, the number of resources, m is equal to 1. All the above matrices $n \times m$ becomes n-vectors, and all vectors becomes a single number. Instead of writing M_{ij}, we just write M_i, etc. Assume the Bank has a total of e_1 dollars to be lent.
  3. The algorithm is called the BANKER.
  4. Each process is called a CUSTOMER.
  5. The Maximum Request Matrix is just a vector $M = (M_1, \ldots, M_n)$ where M_i is the credit limit for customer i. It is assumed that $M_i \leq e_1$ for all i.
  6. The Current Allocation Matrix is just a vector $A = (A_1, \ldots, A_n)$ and the Claimed Matrix C becomes $C = (C_1, \ldots, C_n)$ where $C_i = M_i - A_i$.
  7. Each customer can either ask to borrow more money (up to its credit limit) or return borrowed money.
  8. The banker has to decide to satisfy a request for money or to put it on hold.
  9. The Free Vector is just a number f_1, equal to $e_1 - \sum_{i=1}^{n} A_i$.

10. WHAT DOES IT MEAN for A to be SAFE? It means this: eventually, every customer who is holding resources will return all borrowed money.
11. ASSUMPTION: a customer may borrow up to its limit, and if this limit is reached, it will eventually return all the borrowed money (if not earlier).
12. We say the state A is **safe** if either $A = (0, \ldots, 0)$ or there exists some $i = 1, \ldots, n$ with $A_i > 0$ such that $f_1 \geq C_i$ and the updated state with $A_i = 0$ (so $f_1 = f_1 + A_i$) is safe.
13. EXAMPLE:
14. This easily leads to an $O(n^2)$ time algorithm to check for safety.

- Safety Checking Algorithm

  1. This algorithm checks if a state is safe. It depends only on the matrices A and M and the Free Vector F.
  2. Mark every process i corresponding to the ith row of A that is non-zero. The other processes are unmarked.
  3. Let G be initialized to the free vector F.
  4. Do as long as the following condition hold: (1) there is a marked process, p_i, and (2) $A_i + G \geq M_i$ where M_i and A_i are the ith rows of M and A, resp. When (1) and (2) holds, we add A_i to G and unmark p_i.
  5. At this point, (1) or (2) fails. If (1) fails, all processes are unmarked. We conclude that A is safe. If (2) fails, we have found an marked process that can cause a deadlock.
  6. What does unmarking a process p_i mean? It means that if the process p_i can complete its job if we wait long enough.