CMPT 300 — Operating Systems I

Summer 1999

Segment 7:

Deadlock, Livelock, Starvation & Determinacy

Melissa O'Neill

Deadlock

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

  For example,

  - System has a tape drive and a printer, protected by two mutexes, printer and tape (initially unlocked)
  - Two processes, *Alice* and *Bob* each need both devices and behave as follows:

Alice	*Bob*
printer.lock();	tape.lock();
… work …	*… work …*
tape.lock();	printer.lock();
… finish …	*… finish …*
tape.unlock();	printer.unlock();
printer.unlock();	tape.unlock();

Conditions for Deadlock

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Methods for Handling Deadlock

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem.

Deadlock Prevention

Restrain the ways resource requests can be made to prevent one of the four conditions for deadlock from occurring.

To employ deadlock prevention, processes should prevent *one* of the conditions for deadlock:

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Class Exercise: Develop ways to provide access to resources while preventing (at least one of) the above conditions.

Some answers:

- Mutual exclusion
  - Can the resource be shared?
  - Can we use spooling?
- Hold and wait
  - Allocate resources all at once at start
  - Prevent process from holding any resources while it is waiting
  - Allow non-blocking resource acquisition attempts
- No preemption
  - If a process holding resources requests another resource that cannot be immediately allocated to it, then the operating system releases all resources that process holds; process then waits for all the resources it now needs (old ones and new ones)
- Circular wait
  - Impose a total order on resource acquisition

Stochastic Locking

Example,

- Usual rule, lock the buffer, *then* lock the buffer list
- But, sometimes the program needs to find a free buffer in the list and use it (i.e., need to traverse the buffer list, find a buffer, then lock it)

  Solution,
  - Lock the buffer list
  - Move down the list of buffers, trying to find one we can lock. When we succeed in locking a buffer, that's the one we use.

Key idea

- When the program must break the rules for lock ordering, it can use a try_lock (or try_wait) primitive which won't block the process and won't risk deadlock. In this case, the programmer must think about issues like starvation and livelock instead of deadlock.

Deadlock Avoidance

Key idea

- Make the operating system worry about stopping deadlock from happening.

Requires that the system has some additional information available.

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
  - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe States

Requires that the system has some additional information available.

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe execution sequence of all processes.
- Execution sequence $<s_1, s_2, \ldots, s_3>$ is safe if for each s_i, the resources that process s_i can still request can be satisfied by currently available resources + resources held by all the s_j, with $j < i$.
  - If the resource needs of s_i are not immediately available, then it can wait until enough earlier processes have finished.

Safe States (contd.)

- If a system is in a safe state , no deadlocks are possible
- If a system is in an unsafe state, deadlock is possible.

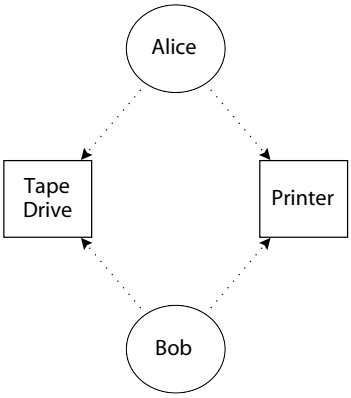
Thus,

- To avoid deadlock, ensure that a system will never enter an unsafe state.

Resource-Allocation Graph Algorithm

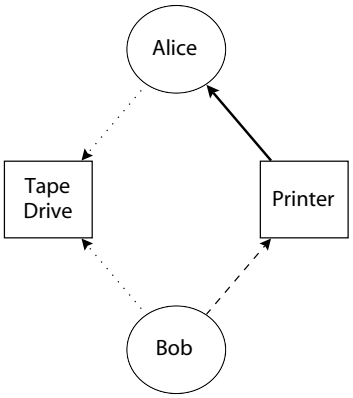
- Works for systems where there aren't multiple (equivalent) instances of resources
- Graph with vertices for resources and processes
- Three kinds of edges
  - *Claim edge* from process to resource
  - *Request edge* from process to resource
  - *Assignment edge* from resource to process
- Requesting process is blocked if converting its request edge to an assignment edge would cause a cycle

Resource-Allocation Graph Algorithm — Example



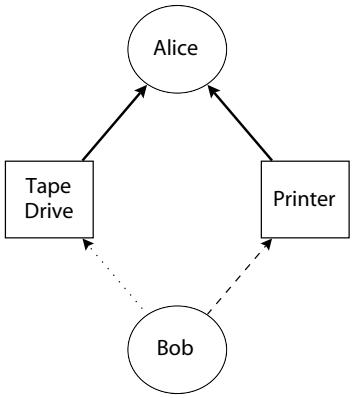
(Alice and Bob will both want to use the tape drive at some point.)

Resource-Allocation Graph Algorithm — Example



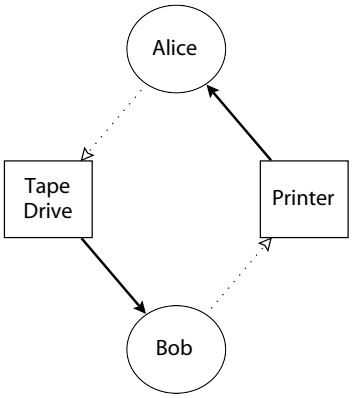
(Alice requests the printer and gets it, then Bob requests the printer also, but has to wait.)

Resource-Allocation Graph Algorithm — Example



(Alice then requests and gets the tape drive.)

Resource-Allocation Graph Algorithm — Another Example



(Alice requests the printer, then Bob requests the tape drive. Giving Bob the tape drive would result in the above graph.)

Banker's Algorithm

Sometimes we have multiple instances of a resource, for example:

- Ten direct lines to Toronto
- Five direct lines to Victoria
- Seven direct lines to Seattle

Resource-allocation graph algorithm does not work for multiple instances.

Solution — Banker's algorithm.

- Less efficient than resource-allocation graph algorithm
- But works for resources that have multiple instances
- Tasks must declare when they begin the maximum amount of each resource that they might ask for

Data Structures for the Banker's Algorithm

```
const int N;                          // total number of tasks
const int M;                          // total number of resource types

int available[M];                     // how much of each resource is available

int allocation[N][M];                 // current allocation of resources to tasks

int need[N][M];                       // how much more of each resource each task
                                      // could still require. For each task, this value is
                                      //   initially the task's declared maximum — its
                                      //   need decreases as its allocation increases.

bool has_resources(int task) {
    for ( int j = 0; j < M; ++j ) {
        if (allocation[task][j] > 0)
            return true;
    }
    return false;
}
```


Safety Algorithm

```
bool in_safe_state() {

    bool   done[N];
    int    current[M] = allocation;

    for (int i = 0; i < N; ++i)   // initialize elements of done to false
        done[i] = false;

    while (i < N) {               // see if we could finish all tasks
        for (i = 0; i < N; ++i) {
            if (done[i] == false && need[i] <= current) {
                done[i] = true;
                current += allocation[i];        // vector addition
                break;
            }
        }                                        // at loop exit, i == N
    }

        ⋮
```


Safety Algorithm (continued)

```
        ⋮

    bool safe = true;
    for (i = 0; i < n; ++i)
        if (done[i] == false) {
            safe = false;
            break;
    }
    return safe;
}
```

Class Exercise: Are either of the breaks necessary? What happens if they are removed?

Safety Algorithm (revised)

```
bool in_safe_state() {

    bool   done[N];
    int    current[M] = allocation;

    for (int i = 0; i < N; ++i)   // initialize elements of done to false
        done[i] = false;

    bool   changed;
    do {                          // see if we could finish all tasks
        changed = false
        for (i = 0; i < N; ++i) {
            if (done[i] == false && need[i] <= current) {
                done[i] = true;
                current += allocation[i];
                changed = true;
            }
        }
    } while (changed);

        ⋮
```

Revised Safety Algorithm (continued)

```
    ⋮
    bool safe = true;
    for (i = 0; i < n; ++i)
        if (done[i] == false) {
            safe = false;
            break;
    }
    return safe;
}
```


Resource-Request Algorithm

```
void request_resources(int task, int request[M]) {
    if (request > need[task])
        throw BadRequest();
    bool succeeded = false;
    while (succeeded == false) {
        if (request > available) {    // not enough resources — wait
            current_task.suspend();
        } else {                      // enough resources — is it safe?
            available -= request;
            allocation[task] += request;
            need[task] -= request;
            if (in_safe_state())
                succeeded = true;
            else {    // not safe — undo allocation attempt and wait
                available += request;
                allocation[task] -= request;
                need[task] += request;
                current_task.suspend();
            }
        }
    }
}
```


Banker's Algorithm — Example

- 5 processes, $P_0 \ldots P_4$
- 3 resource types
  - A — 10 instances
  - B — 5 instances
  - C — 7 instances
- Snapshot at time T_0

Available				Allocation			Max		
A	B	C		A	B	C	A	B	C
3	3	2	P_0	0	1	0	7	5	3
			P_1	2	0	0	3	2	2
			P_2	3	0	2	9	0	2
			P_3	2	1	1	2	2	2
			P_4	0	0	2	4	3	3

Class Exercise: What is the content of the need matrix?

Is the system in a safe state?

Banker's Algorithm — Example (contd.)

- Need matrix is:

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- The system is in a safe state since the sequence

  $<P_1, P_3, P_4, P_0, P_2>$

  satisfies safety requirement.
- Now, suppose P1 requests (1,0,2)

  Request <= Need[1]

  Request <= Available

Banker's Algorithm — Example (contd.)

- Configuration if request is granted

Available				Allocation			Need		
A	B	C		A	B	C	A	B	C
2	3	0	P_0	0	1	0	7	4	3
			P_1	3	0	2	0	2	0
			P_2	3	0	2	6	0	0
			P_3	2	1	1	0	1	1
			P_4	0	0	2	4	3	1

- The system is in a safe state since the sequence

  $<P_1, P_3, P_4, P_0, P_2>$

  satisfies safety requirement.

Class Exercise: P4 requests (3,3,0) — grantable?
P0 requests (0,2,0) — grantable?

Deadlock Detection

- Allow system to enter deadlock state
- Detection deadlock when it occurs
- Recover

Wait-for Graph Algorithm

- Works for systems where there aren't multiple (equivalent) instances of resources
- Graph with vertices for processes
- One kind of edge
  - *Wait-for edge* from process to process that has the desired resource
- Periodically invoke an algorithm that searches for a cycle in the graph.
  - An algorithm to detect a cycle in a graph requires an $O(n^2)$ operations, where n is the number of vertices in the graph.

Coffman's Algorithm

- Based on the banker's algorithm
  - banker's algorithm + optimism
- Less efficient than wait-for graph algorithm
- But, works for resources that have multiple instances

Data Structures for the Banker's Algorithm

```
const int N;                          // total number of tasks
const int M;                          // total number of resource types

int available[M];                     // how much of each resource is available

int allocation[N][M];                 // current allocation of resources to tasks

int request[N][M];                    // how much more of each resource each task
                                      //      each task is waiting for (all zeros if not
                                      //                                waiting at all).

bool has_resources(int task) {
    for ( int j = 0; j < M; ++j ) {
        if (allocation[task][j] > 0)
            return true;
    }
    return false;
}
```


Detection Algorithm

```
bool is_deadlocked() {

    bool   done[N];
    int    current[M] = allocation;

    for (int i = 0; i < N; ++i)          // initialize elements of done
        done[i] = ! has_resources(i);

    bool changed;
    do {
        changed = false;
        for (i = 0; i < N; ++i) {
            if (done[i] == false && request[i] <= current) {
                done[i] = true;
                current = current + allocation[i];
                changed = true;
            }
        }
    } while (changed);

        ⋮
```


Detection Algorithm (continued)

```
        ⋮

    bool deadlocked = false;
    for (i = 0; i < n; ++i)
        if (done[i] == false) {
            deadlocked = true;
            break;
    }
    return deadlocked;
}
```

Class Exercise: Can we remove the initial for loop?

Detection Algorithm — Example

- 5 processes, $P_0 \ldots P_4$
- 3 resource types
  - A — 7 instances
  - B — 2 instances
  - C — 6 instances
- Snapshot of system

Available				Allocation			Request			
A	B	C		A	B	C	A	B	C	
0	0	0	P_1	0	1	0	0	0	0	(runnable)
			P_2	2	0	0	2	0	2	(blocked)
			P_3	3	0	3	0	0	0	(runnable)
			P_4	2	1	1	1	0	0	(blocked)
			P_5	0	0	2	0	0	2	(blocked)

- The system is not deadlocked. The sequence

  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$

  could complete all processes.

Detection Algorithm — Example (contd.)

- What if P_2 now requests another instance of resource C?
- Snapshot of system

Available				Allocation			Request			
A	B	C		A	B	C	A	B	C	
0	0	0	P_0	0	1	0	0	0	0	(runnable)
			P_1	2	0	0	2	0	2	(blocked)
			P_2	3	0	3	0	0	1	(blocked)
			P_3	2	1	1	1	0	0	(blocked)
			P_4	0	0	2	0	0	2	(blocked)

Class Question: Is the system deadlocked?

Using Deadlock Detection

- When, and how often, to invoke depends on how often a deadlock is likely to occur.
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Class Exercise: Can we modify our deadlock detection algorithm so it identifies all the cycles? If so, how?

Recovery from Deadlock — Process Termination

Either:

- Abort all deadlocked processes, or
- Abort one process at a time until the deadlock cycle is eliminated.

But:

- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock — Resource Preemption

Two steps:

- Selecting a victim (minimize cost).
- Rollback (return to some safe state, restart process from that state.)

But:

- Starvation concerns
  - Same process may always be picked as victim, so include number of rollback in cost factor.

Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - Prevention
  - Avoidance
  - Detection

  to allow the use of the optimal approach for each class of resources in the system.
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.

Class Exercise: If we protect against deadlock, and protect shared data using locks, can we relax, confident our program will operate correctly?

If not, what issues should we worry about?

Starvation

If we escape deadlock, we may be prone to starvation:

- Deadlock — set of processes, none of which make progress
- Starvation— one process is delayed forever because other processes are always given preference

(remember the dining philosophers?)

Solution:

- Aging — processes that have been waiting a long time are given priority over those that have been waiting a short time.

Livelock

Attempting to use "back-off, retry later" or resource preemption approaches to preventing deadlock may cause *livelock*:

- Processes don't deadlock, but fail to make progress either.

  For example, suppose Wayne and Larry both want to listen to a *Lamest Z95 hits* CD on a personal CD player. Suppose:
  - Wayne has the CD player but wants the CD.
  - Larry has the CD but wants the CD player.
  - Wayne steals (i.e., preempts) the CD from Larry, meanwhile, Larry steals the CD player from Wayne.
  - Now, Wayne has the CD and Larry has the CD player.
  - Wayne steals the CD player from Larry, meanwhile, Larry steals the CD from Wayne.
  - Now, Wayne has the CD player and Larry has the CD.

  ⋮

  Livelock!

Determinacy Races

Class Exercise: Examine the programs below, and assume that they (implicitly) use locks to protect access to the variables.

Which ones always produce the same answer?

(a)

```
a := 2;
cobegin
   b := a;
   a := a + 1;
coend;
```

(b)

```
a := 1;
b := 2;
cobegin
   c := a + b;
   if a > b then
      d := a;
   else
      d := b;
coend;
cobegin
   a := c / 2;
   d := b + c + d;
coend;
```

(c)

```
a := 1;
b := 2;
cobegin
   begin
      c := a + 3;
      cobegin
         a := (a + b) / c;
         d := b / c;
      coend;
   end
   e = (b / 2) + 1;
coend;
```

(d)

```
cobegin
   a := -2;
   a := 2;
coend;
a := a * a;
```

Bernstein's Conditions

Bernstein derived useful conditions for avoiding determinacy races:

- Basic Conditions, for two tasks, t and t':

  $t \nleftrightarrow t' \quad \Leftrightarrow \quad (W_t \cap W_{t'} = \varnothing) \wedge (R_t \cap W_{t'} = \varnothing) \wedge (W_t \cap R_{t'} = \varnothing)$

  where
  - R_t = the set of memory locations read by t
  - $R_{t'}$ = the set of memory locations read by t'
  - W_t = the set of memory locations written by t
  - $W_{t'}$ = the set of memory locations written by t'

- For a set of tasks, $T = \{t_1, \ldots, t_n\}$:

  $\forall t_i, t_j \in T \text{ s.t. } t_i \ \not\trianglelefteq \ t_j \wedge t_j \ \not\trianglelefteq \ t_i \quad : \quad t_i \nleftrightarrow t_j$

  (assuming that parent thread *does not* run while children are running)

Segment Review

You should be able to:

- State the conditions necessary for deadlock
- Devise ways of preventing deadlock from occurring
- Describe, analyse and apply the resource allocation graph algorithm
- Describe, analyse and apply deadlock detection algorithms
- Describe approaches used when to deal with deadlock when it has been detected
- Explain the tradeoffs associated with deadlock handling
- Explain livelock and starvation
- Explain and apply Bernstein's conditions for determinacy