Operating Systems W6L1 - Deadlocks

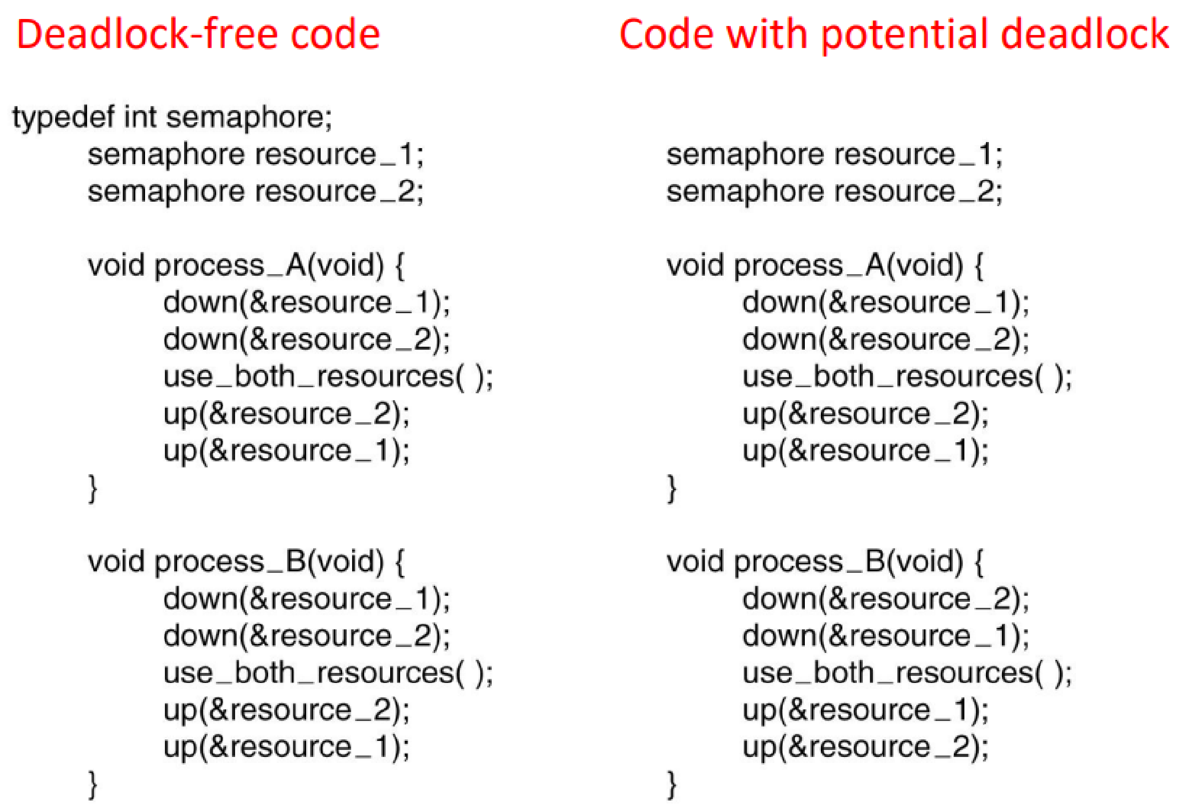
Class	Operating Systems
Created	@Oct 5, 2020 9:26 PM
Materials	
Reviewed	☐
Type	Lecture

Deadlocks

- Deadlock prevention exists everywhere, not just big machins but also your laptop and phones
- Resources can be *anything*, not just hardware
- ▼ What are preemptable and non-preemptable resources? What are some examples?
  - **Preemptable:** Can be taken away from the process with no ill-effect
  - **Non-preemptable:** Cannot be taken away from the process without causing the computation to fail. In other words (since negation can be confusing), *if you take the resource away from the process, the computation (and thus the process) will fail.*
- ▼ What are reusable and consumable resources? What are some examples?
  - ▼ Reusable
    - Can be safely used by only one process and is *not* depleted by that use

        - This includes processors, I/O channels, main and secondary memory, devices, and data structures (such as files, databases, and semaphores)
    - ▼ Consumable
        - Can be created (produced) and destroyed (consumed)
        - This includes interrupts, signals, messages, and information. Things within I/O buffers
- As a programmer, if you are not careful then you can cause a deadlock
- ▼ Deadlock-free vs. Deadlock-potential code

  Note the difference! If A acquires R1 and B acquires R2, then neither can proceed to the next step since they are both waiting on a taken resources. The code on the left ensures that A and B are waiting for the same resource, and thus not waiting for each other. **You want processes to compete for resources in the same order**



Deadlock-free code

```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

Code with potential deadlock

```
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_2);
                down(&resource_1);
                use_both_resources( );
                up(&resource_1);
                up(&resource_2);
        }
```

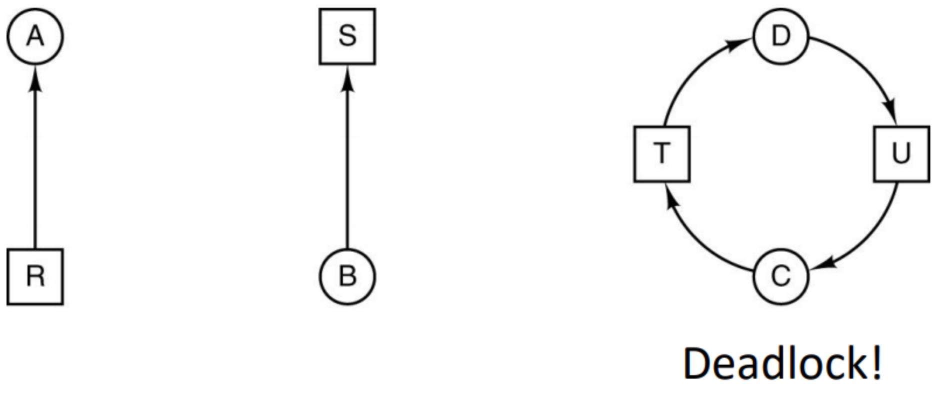
- Deadlocks do not waste resources → All processes deadlocked are in a blocked state

- A set of processes is deadlocked if each process in that set is waiting for an *event* that only another process in the set can cause
  - See chaining later in the notes
- ▼ Assumptions thus far
  - If a process is denied a resource, it is put to sleep
  - Only single-threaded processes
  - No interrupts possible to wake up a blocked process

Conditions for Resource Deadlocks

1. Each resource assigned exactly one process or is unavailable
2. A process using a resource can request anotehr resource
3. Resources must be explicitly released by process using them, not taken forcibly
4. A *circular chain* of two or more processes waiting for a resource already being held by the next process in the chain

▼ Resource Allocation Graph



▼ Deadlock Examples

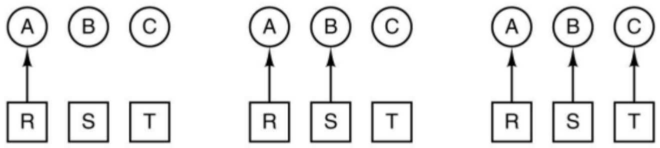
Walk through these and ensure you can explain where teh deadlocks come from!

Example 1:

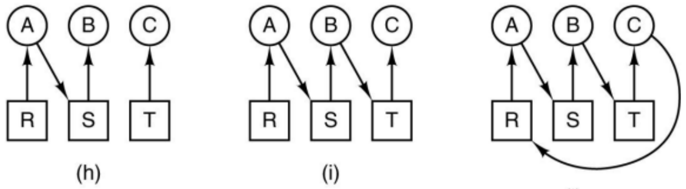
A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

(d)



(e) (f) (g)



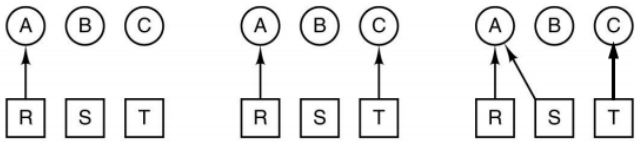
(h) (i) (j)

Example 2:

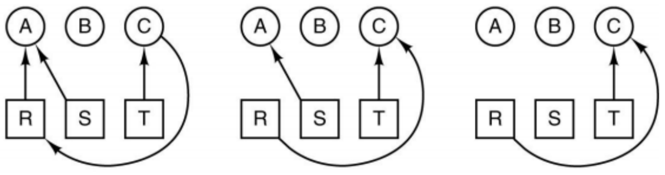
A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock

(k)



(l) (m) (n)



(o) (p) (q)

- Potential deadlock is non-deterministic, such as if the order works out. However, this entails we rely on luck and we don't want to do that

Dealing With Deadlocks

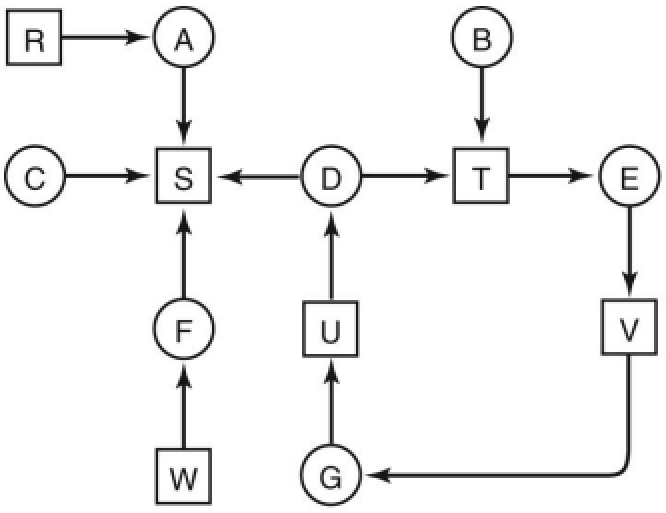
1. Ignore it and just be lucky → Put your head in the sand ("ostrich method")
2. Let them occur, detect them, and then take care of them
3. Avoid dynamically by careful resource allocation — Each time a process takes a resource, check if it can
4. Prevent statically by having a structure that negates 1 of the 4 conditions (in "conditions" section above)

Detection

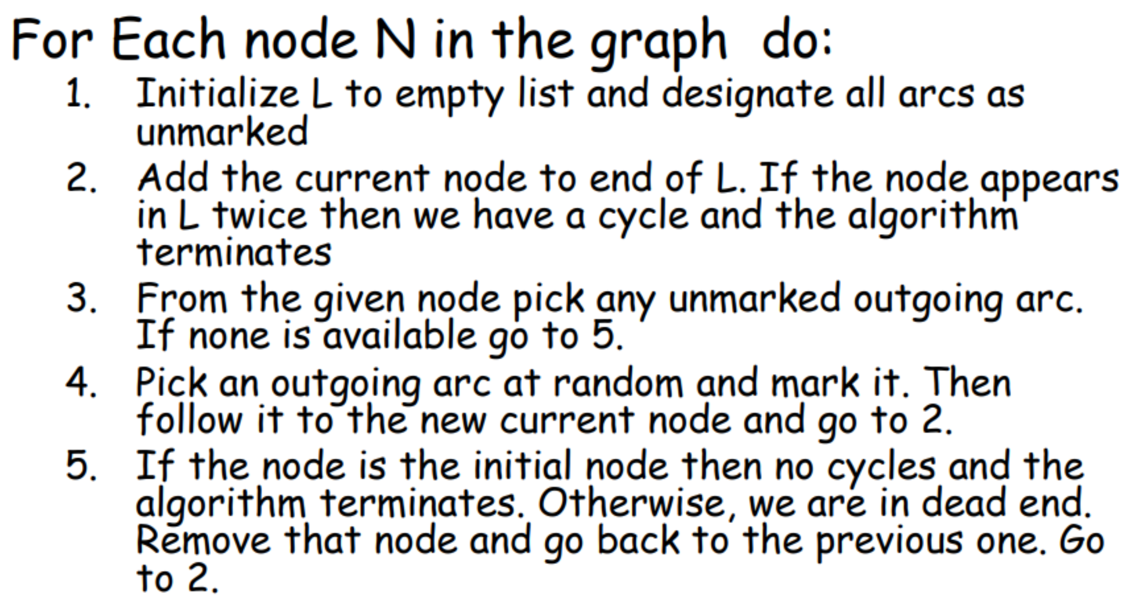
- The system does not prevent deadlocks, but recovers once it detects them
- The main issues here are...
  - Having to detect a resource of each type
  - Having to detect multiple resources of each type
  - Recovering from a deadlock

▼ Construct a graph and look for cucles, which upon being found entails a deadlock

Difficult to code, and would also need efficient searching



▼ Formal algorithm



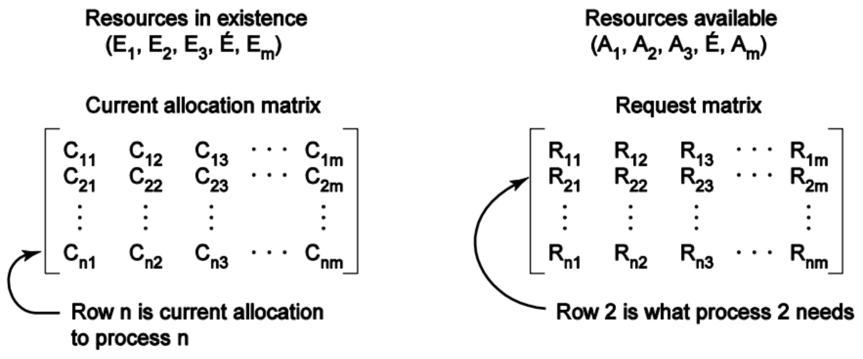
For Each node N in the graph do:

1. Initialize L to empty list and designate all arcs as unmarked
2. Add the current node to end of L. If the node appears in L twice then we have a cycle and the algorithm terminates
3. From the given node pick any unmarked outgoing arc. If none is available go to 5.
4. Pick an outgoing arc at random and mark it. Then follow it to the new current node and go to 2.
5. If the node is the initial node then no cycles and the algorithm terminates. Otherwise, we are in dead end. Remove that node and go back to the previous one. Go to 2.

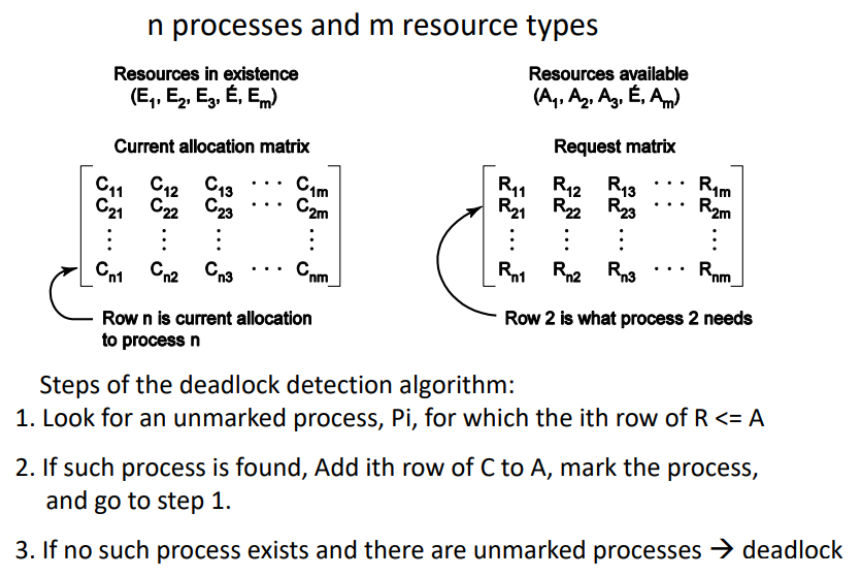
❗ Good exercise: Apply the algorithm to the above graph diagram

▼ Multiple Resources of Each Type

n processes and m resource types


Resources in existence
(E_1, E_2, E_3, É, E_m)
Resources available
(A_1, A_2, A_3, É, A_m)
Current allocation matrix
Request matrix
Row n is current allocation to process n
Row 2 is what process 2 needs

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix} \qquad \begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$



n processes and m resource types

Resources in existence
$(E_1, E_2, E_3, É, E_m)$

Resources available
$(A_1, A_2, A_3, É, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Steps of the deadlock detection algorithm:

1. Look for an unmarked process, Pi, for which the ith row of R <= A
2. If such process is found, Add ith row of C to A, mark the process, and go to step 1.
3. If no such process exists and there are unmarked processes → deadlock

▼ What does it mean for a process to be **marked**?

A process is said to be **marked** if it is able to complete, thus not becoming deadlocked. A marked process will never ask for other resources

▼ When should you check for deadlocks?

- Every time a resource request is made
- Every `k` minutes
- CPU utilization has dropped below a particular threshold (i.e. temperature) → Performance counters built into CUs make this quite easy for the OS, since the metrics are straightforward to obtain