Chapter 6 - Deadlocks

Luis Tarrataca
luis.tarrataca@gmail.com

CEFET-RJ

Motivation

Certain resources can only be used by one process at a time, *e.g.:*

- printers, tapes, internal tables;

Having two processes simultaneously accessing the:

- Printer leads to gibberish;
- Same file-system table slot will lead to a corrupted file system;

Consequently:

- OS needs ability to grant a process exclusive access to certain resources.

A process may also need exclusive access to multiple resources, *e.g.:*

- Two processes each want to record a scanned document on a Blu-ray disc
  1. Process A requests permission to use the scanner and is granted it;
  2. Process B requests permission to use the Blu-ray driver and is granted it;
  3. Process A then asks for the Blu-ray recorder:
     - But request is suspended until B releases it;
  4. Unfortunately instead of releasing Blu-ray recorder:
     - Process B asks for the scanner;

Can you see any problem with this behaviour? Any ideas?

Can you see any problem with this behaviour? Any ideas?

This situation is called a **deadlock**, *i.e.:*

- No progress can be made!
- In portuguese: *impasse*

Can you think of any other type of situation where deadlocks occur?

Can you think of any other type of situation where deadlocks occur?

Database system, where program may have to lock several registers *e.g.:*

- Process A locks records R1;
- Process B locks record R2;
- If each process tries to lock each other one's record:
  - **Deadlock...**

Conclusion:

- Deadlocks can occur on hardware resources or on software resources.
- Deadlocks happen throughout computer science!

Resources

Deadlocks may involve **resources** requiring exclusive access:

- *E.g.:* devices, data records, files, and so forth;
- Resource can be:
  - Hardware device;
  - Piece of information;
- Many different resources that a process can acquire;
- If several instances of the resource are available:
  - Any one of them can be used to satisfy any request for the resource

In your opinion what is a resource? Any ideas?

In your opinion what is a resource? Any ideas?

Resource is anything that must be:

- Acquired, used, and released over the course of time.

Preemptable and Nonpreemptable Resources

Resources come in two types:

- **Preemptable** (in portuguese: preemptivo)
- **Nonpreemptable**

Lets have a look at these types =)

Preemptable Resource

Preemptable Resource:

- Can be taken away from the process owning it with no ill effects.

Can you think of any resource that is preemptable? Any ideas?

Example (1/4)

Can you think of any resource that is preemptable? Any ideas?

Memory is a preemptable resource, consider a system with:

- 1 GB of user memory
- One printer
- Two 1-GB processes that each want to print something.;

Example (2/4)

1. Process A requests and gets the printer:
   - Then starts to compute the values to print;
2. Before it has finished the computation:
   - OS scheduler changes to process B;
3. Process B now runs and tries, unsuccessfully to acquire printer;

Example (3/4)

Potential deadlock situation:

- Process A has the printer;
- Process B has the memory;
- Neither process can process with the resource held by the other;

Example (4/4)

But is this really a problem? Any ideas?

Not really! No deadlock occurs since:

- Possible to **preempt the memory** from B and swapping in process A;
- Now process A can:
  - Execute;
  - Print;
  - Release printer;

Nonpreemptable resource

Nonpreemptable resource:

- Cannot be taken away from its current owner without potentially causing failure

Can you think of any resource that is nonpreemptable? Any ideas?

Can you think of any resource that is nonpreemptable? Any ideas?

Example: If a process has begun to burn a Blu-ray

- Cannot simply give the Blu-ray drive to another process;
- This would simply result in a garbled Blu-ray;
- Blu-ray recorders are not preemptable at an arbitrary moment.

So how can we know wether a resource is preemptable or not? Any ideas?

So how can we know wether a resource is preemptable or not? Any ideas?

No simple answer!:

- Depends on the context!
- Memory is preemptable because:
  - Pages can always be swapped out to disk to recover it

So, how can we deal with nonpreemptable resources? Any ideas

So, how can we deal with nonpreemptable resources? Any ideas

Abstract sequence of events required to use a resource:

1. Request the resource.
2. Use the resource.
3. Release the resource.

If resource is **not available** when requested:

- Requesting process is forced to **wait;**
- In some OS process is automatically:
  - **blocked** when a resource request fails;
  - **awakened** when a resource available;
- In other OS:
  - Request fails with an error code;
  - Up to the calling process to wait a little while and try again;

When a resource request is **denied:**

- Process will **loop:**
  1. Requesting the resource;
  2. **Sleeping;**
  3. Trying again.

Although process is not blocked:

- it is as good as blocked, because it cannot do any useful work;
- From now on we will assume process always **sleeps**

Resource Acquisition

For some kinds of resources:

- Up to the process to manage resource usage themselves:
  - Rather than the OS;
  - *E.g.:* records in a database system;

Resource Acquisition

For some kinds of resources:

- Up to the process to manage resource usage themselves:
  - Rather than the OS;
  - *E.g.:* records in a database system;

But how can we manage resources? Any ideas?

Resource Acquisition

But how can we manage resources? Any ideas?

Associate a **semaphore** or a **mutex** with each resource:

- All initialized to 1;
- The three steps list above are then implemented as:
  1. **Down Operation:** Request the resource.
  2. Use the resource.
  3. **Up operation:** Release the resource.

This strategy can be illustrated as follows for **one resource:**

```
typedef int semaphore;
semaphore resource_1;


void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

Figure: Using a semaphore to protect one resource (Source: (Tanenbaum and Bos, 2015))

This strategy can be illustrated as follows for **two resources:**

```
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);
}
```

Figure: Using a semaphore to protect two resources (Source: (Tanenbaum and Bos, 2015))

Can you see any problem with the previous examples to protect resources? Any ideas?

Can you see any problem with the previous examples to protect resources? Any ideas?

- Only one process is involved!
  - Everything works fine;
  - No competition for resources;
  - No need to formally acquire resources;

Lets consider **two processes**, A and B, and **two resources**:

- **Scenario 1:** both processes ask for the resources in the **same** order;
- **Scenario 2:** both processes ask for the resources in **different** order;

Difference may seem minor but it is not... Lets have a look =)

Scenario 1: Same order resource acquisition

```
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);
        }
```

Figure: Deadlock-free code (Source: (Tanenbaum and Bos, 2015))

Scenario 1: Same order resource acquisition

From the previous figure:

1. Process A will acquire 1^{st} resource before Process B;
2. Process A will acquire 2^{nd} resource;
3. Process A will do its work.

If Process B attempts to acquire resource 1 before it has been released:

- Process B will block until resource becomes available.

Now lets have a look at what happens with a different order resource acquisition

Scenario 2: Different order resource acquisition

```
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);
}
```

Figure: Code with potential deadlock (Source: (Tanenbaum and Bos, 2015))

Scenario 2: Different order resource acquisition

From the previous figure (1/2):

- **Possibility 1:**
  1. Process A acquires both resources:
  2. Effectively blocks Process B until Process A is done;

Scenario 2: Different order resource acquisition

From the previous figure (2/2):

- **Possibility 2:**
  1. Process A acquires resource 1;
  2. Process B acquires resource 2:
  3. Each process will block trying to acquire the other resource;
  4. Neither process will ever run again... **(Deadlock)**

Introduction to Deadlocks

Based on the previous slides:

In your opinion what is a deadlock? Any ideas?

Introduction to Deadlocks

Based on the previous slides:

> In your opinion what is a deadlock? Any ideas?

Each process is waiting for an event that only another process can cause:

- Because all processes are waiting:
  - None will ever cause the event the other processes depend on;
  - None of the processes can run:
  - None can release the resources;
  - None can be awakened;

Conditions for Resource Deadlocks

Four conditions must hold for there to be a **deadlock:** (1/3)

- **Mutual Exclusion condition**
- **Hold-and-wait condition**
- **No-preemption condition**
- **Circular wait condition**

All four of these conditions must be present for a **deadlock** to occur:

- If one of them is absent, no resource deadlock is possible.

Conditions for Resource Deadlocks

Four conditions must hold for there to be a **deadlock:** (2/3)

- **Mutual Exclusion condition:**
  - Each resource is assigned to zero or one process;
- **Hold-and-wait condition:**
  - Processes holding acquired resources can request new resources;

Conditions for Resource Deadlocks

Four conditions must hold for there to be a **deadlock:** (3/3)

- **No-preemption condition:**
  - Resources previously granted cannot be forcibly taken away from a process.
  - They must be explicitly released by the process holding them.
- **Circular wait condition:**
  - There must be a circular list of two or more processes:
    - Each of which is waiting for a resource held by the next member of the chain.

Deadlock modeling

Directed graphs can be used to model the four conditions (1/3):

- With two kinds of nodes:
  - **Processes (Circles);**
  - **Resources (Rectangles);**

Deadlock modeling

A directed arc from a resource to a process means that:

- Resource has previously been requested by, granted to, and is currently held by that process.

Figure: Holding a resource (Source: (Tanenbaum and Bos, 2015))

A directed arc from a process to a resource means that:

- Process is blocked waiting for that resource;

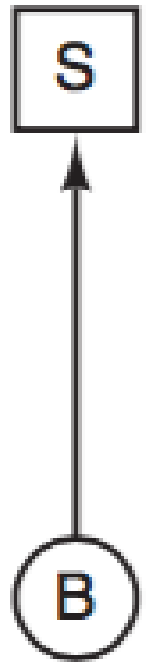

Figure: Holding a resource (Source: (Tanenbaum and Bos, 2015))

Example 1 (1/2)

Example of a deadlock in a graph:

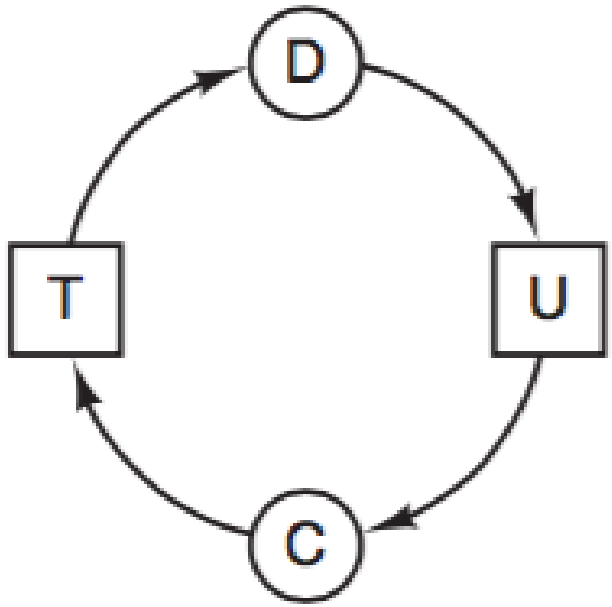

Figure: Deadlock (Source: (Tanenbaum and Bos, 2015))

Example 1 (2/2)

From the previous figure:

- Process C is waiting for resource T
- Resource T is held by process D;
- Process D is waiting for resource U;
- Resource U is held by process C;
- **Conclusion:** C - T - D - U - C, i.e. a cycle!
  - Both processes will wait forever: **Deadlock**

Example 2 (1/2)

Imagine that we have:

- Three processes: A, B, and C;
- Three resources: R, S, and T;
- Requests and releases of the three processes are:

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

Figure: (Source: (Tanenbaum and Bos, 2015))

Example 2 (2/2)

- OS is free to run any unblocked process at any instant, *e.g.*::
  1. Run process A until A finished all its work;
  2. Then run process B to completion;
  3. And finally run process C.

Example 2 (2/2)

- OS is free to run any unblocked process at any instant, *e.g.*::
  1. Run process A until A finished all its work;
  2. Then run process B to completion;
  3. And finally run process C.

But can you see any problem with running the processes in this way? Any ideas?

Example 2 (2/2)

- OS is free to run any unblocked process at any instant, *e.g.*::
  1. Run process A until A finished all its work;
  2. Then run process B to completion;
  3. And finally run process C.

But can you see any problem with running the processes in this way? Any ideas?

- This ordering **does not** lead to any deadlocks;
- There **does not** exist parallelism:
- Accordingly there is **no** competition for resources;

Example 3 (1/8)

Suppose processes do I/O and computing:

- Assume a round-robin scheduling

First things first:

What is round-robin scheduling? Any ideas?

Example 3 (1/8)

Suppose processes do I/O and computing:

- Assume a round-robin scheduling

First things first:

What is round-robin scheduling? Any ideas?

- Each process is given a certain time to execute;
- If process exceeds time:
  - Scheduler switched to another process;
- Procedure repeats in a circular fashion;

Example 3 (2/8)

Resource requests occur in the order:

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

Figure: (Source: (Tanenbaum and Bos, 2015))

Example 3 (3/8)

If requests are carried in order there are six resource graphs:

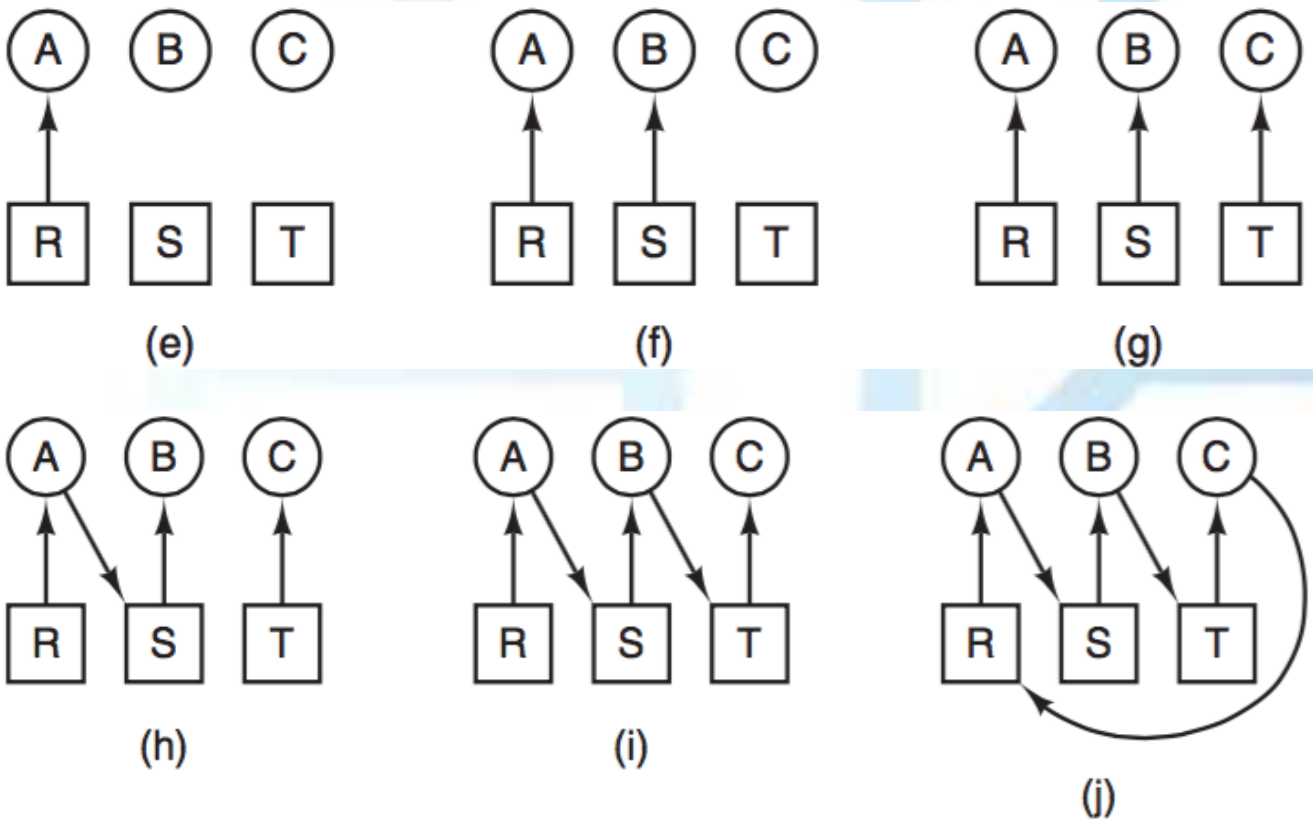

Figure: (Source: (Tanenbaum and Bos, 2015))

Example 3 (4/8)

Can you see where processes block in the previous picture? Any ideas?

Example 3 (4/8)

Can you see where processes block in the previous picture? Any ideas?

- After request 4 has been made:
  - A blocks waiting for S;
- In the next two steps B and C also block;
- Ultimately leading to a **deadlock**;

Example 3 (5/8)

Idea: OS is not required to run processes in any special order:

- If granting a particular request might lead to a deadlock:
  - OS suspends process without granting request;

Example (6/8)

If OS knew about impending deadlock:

- OS could suspend B instead of granting it S;
- By running only A and C, we would get the following requests and releases:

Figure: (Source: (Tanenbaum and Bos, 2015))

Example (7/8)

Previous sequence would lead to the following resource graphs:

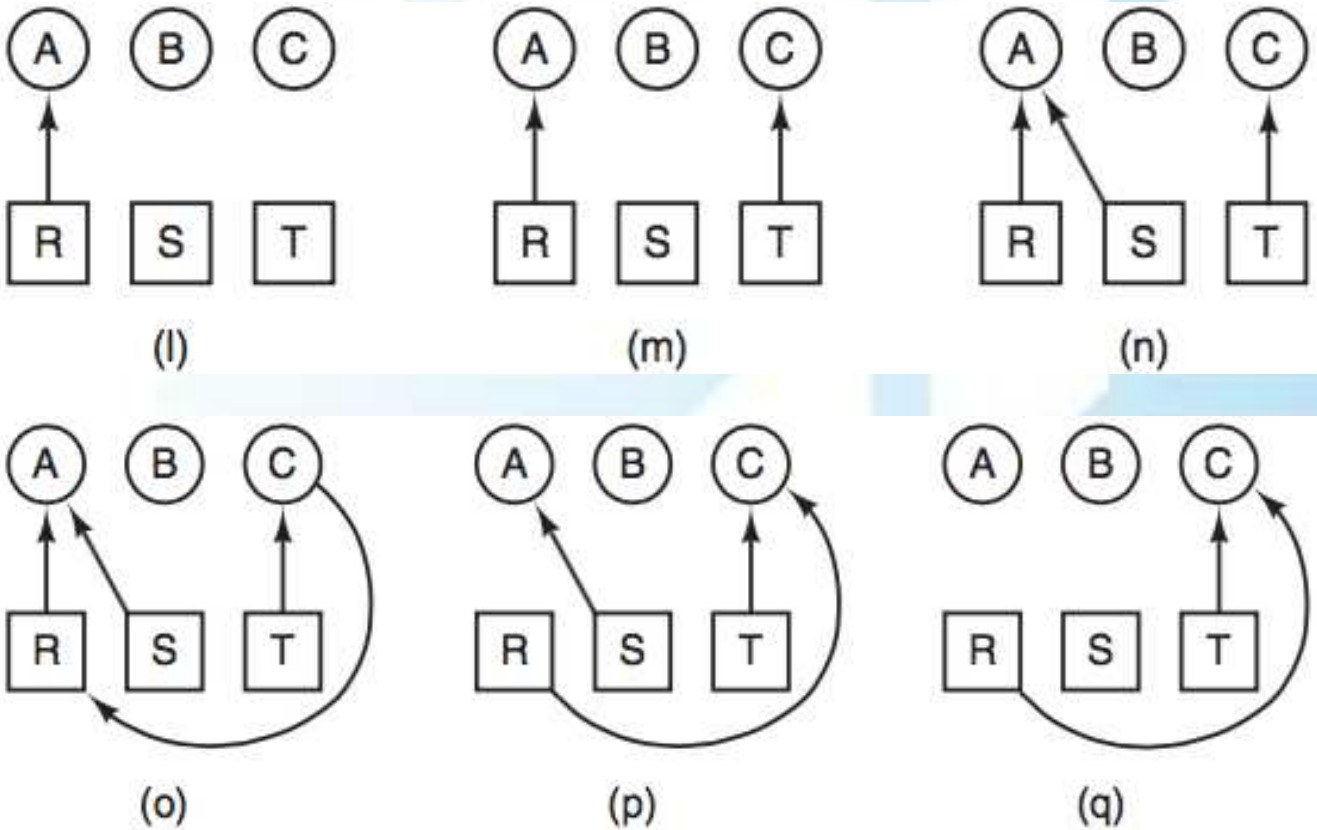

Figure: (Source: (Tanenbaum and Bos, 2015))

Conclusion: No deadlock!

Example (8/8)

After step (q) process B can be granted S because:

- Process A is finished and C has everything it needs;
- Even if B blocks when requesting T:
  - No deadlock can occur;
  - B will just wait until C is finished.

Conclusion:

- Resource graphs are a tool that lets us see if:
  - A given request/release sequence leads to deadlock
- We just carry out the requests and releases step by step:
  - After every step we check the graph to see if it contains any cycles;
  - If there are cycles: **deadlock**;
  - If there are no cycles: **no deadlock**

In general, four strategies are used for dealing with deadlocks:

1. Just ignore the problem:
   - Maybe if you ignore it, it will ignore you.
   - Good life philosophy... ;)
2. Detection and recovery:
   - Let them occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention:
   - by structurally negating one of the four conditions.

Ostrich Algorithm

First things first:

What is an ostrich? Any ideas?

Ostrich Algorithm

First things first:

What is an ostrich? Any ideas?

Figure: Who knows...

What are ostriches known for? Any ideas?

What are ostriches known for? Any ideas?

Figure: Who knows...

Ostrich algorithm:

- Stick your head in the sand and pretend there is no problem;
- People react to this strategy in different ways:
  - Mathematicians: prevent deadlocks at all costs;
  - Engineers: Maybe not worth to deal with deadlocks:
    - How often is the problem expected?
    - How often the system crashes for other reasons?
    - How serious a deadlock is?

Deadlock Detection and Recovery

When this technique is used:

- OS does not attempt to prevent deadlocks from occurring;
- Instead, OS :
  - Lets the deadlock occur;
  - Tries to detect deadlocks;
  - Takes some action to recover;

Lets have a look at some techniques to do this =)

Deadlock Detection with One Resource of Each Type

Let us begin with the simplest case:

- there is only one resource of each type, *e.g.:*
  - one scanner;
  - one Blu-ray recorder;
  - one plotter;
  - etc...
- *I.e.* no more than one of each class of resource;

Based on the previous slides:

How do you think we can detect deadlocks? Any ideas?

Based on the previous slides:

How do you think we can detect deadlocks? Any ideas?

"Strange" idea:

- Construct a resource - process graph;
- If graphs contains one or more cycles:
  - Deadlock exists!
  - Any process that is part of a cycle is deadlocked;
- If no cycles exist, the system is not dead-locked;

Example

Consider a system with:

- Seven processes (A though G);
- Six resources (R through W);
- Process - resource requests:
  1. Process A holds R and wants S;
  2. Process B holds nothing but wants T;
  3. Process C holds nothing but wants S;
  4. Process D holds U and wants S and T;
  5. Process E holds T and wants V;
  6. Process F holds W and wants S;
  7. Process G holds V and wants U;

Is this system deadlocked, and if so, which processes are involved? Any ideas?

- Lets try to construct the graph and see what happens =)

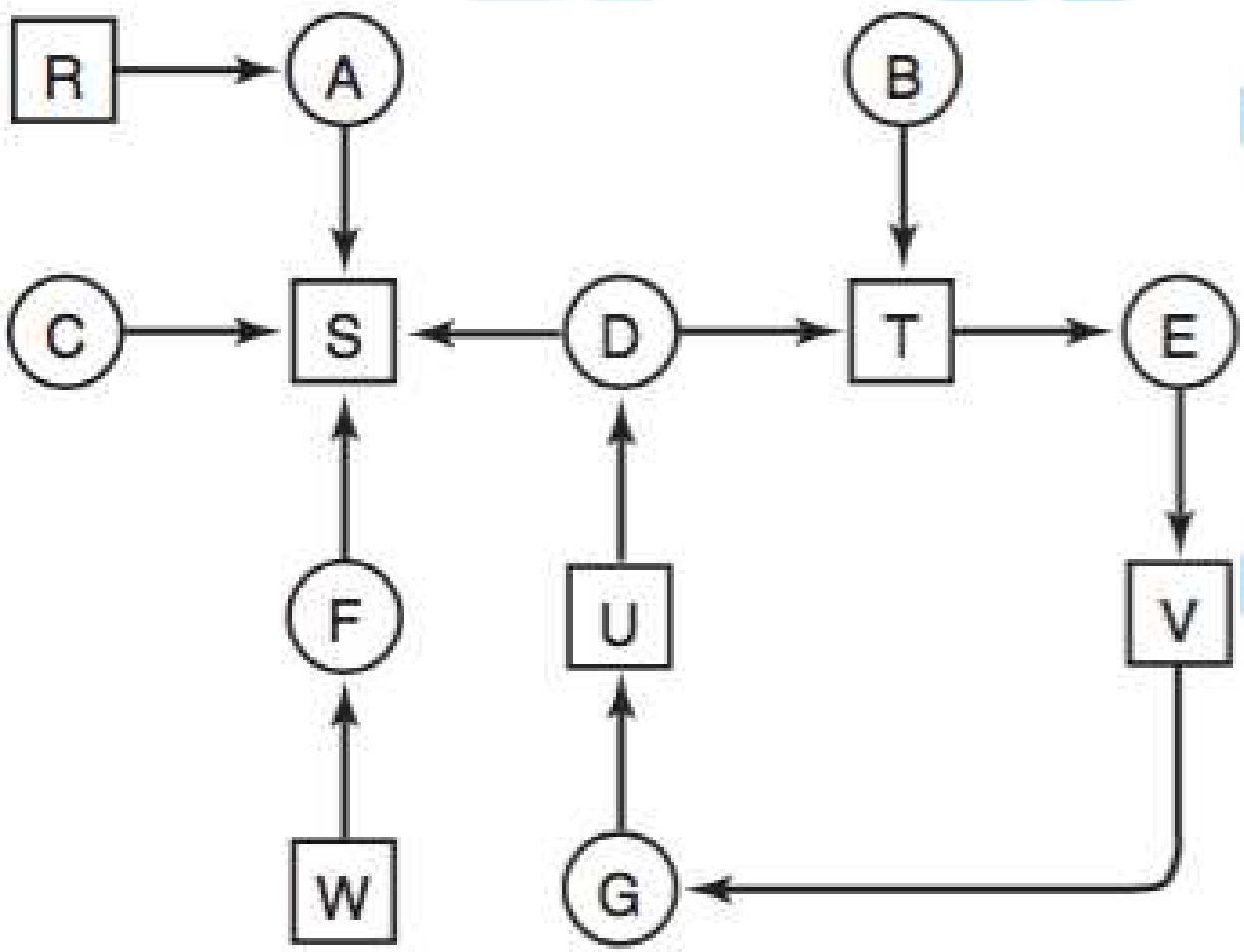

Figure: A resource graph. (Source: (Tanenbaum and Bos, 2015))

Can you see any problems with the resource - process graph? Any ideas?

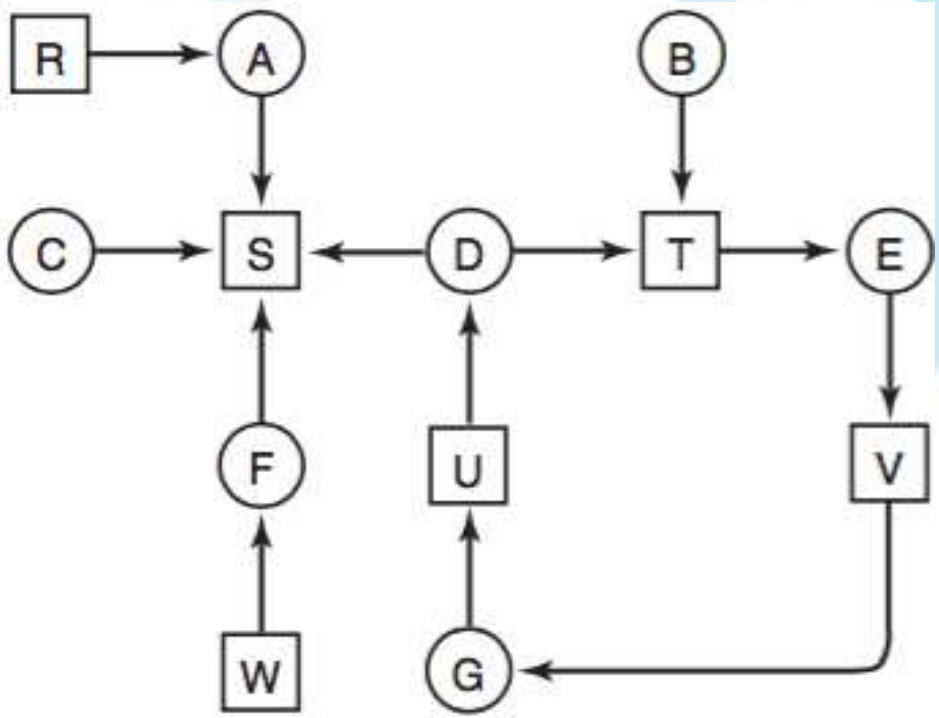

Figure: A resource graph. (Source: (Tanenbaum and Bos, 2015))

Graph contains one cycle, which can be seen by visual inspection (1/2):

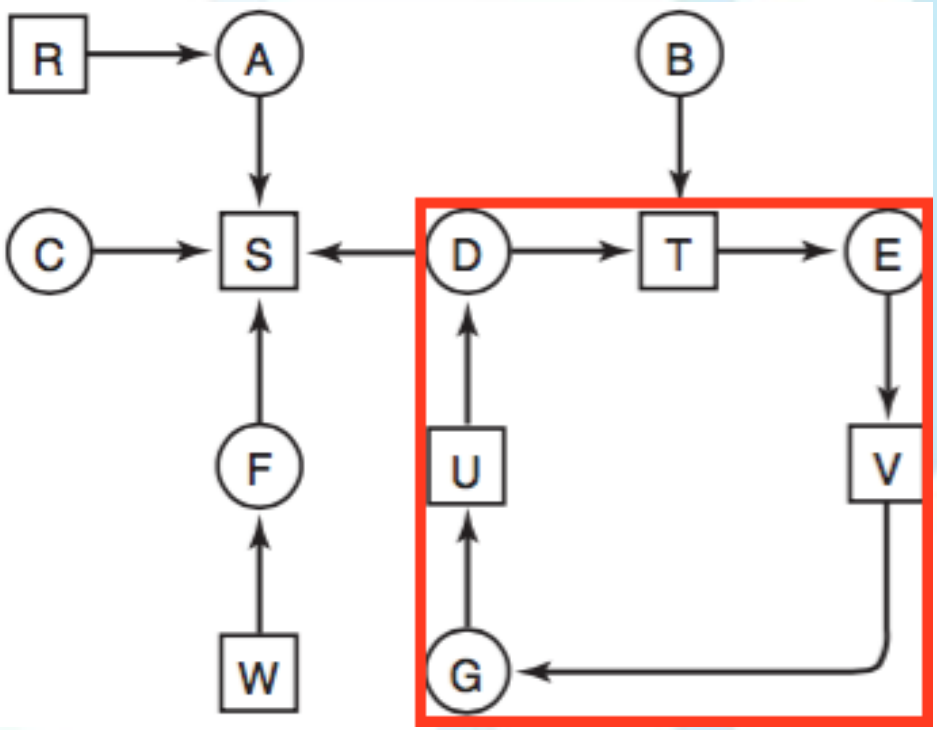

Figure: A resource graph. (Source: (Tanenbaum and Bos, 2015))

Graph contains one cycle, which can be seen by visual inspection (2/2):

- **Deadlocked:**
  - Processes D, E, and G;
- **Not deadlocked:**
  - Processes A, C, and F;
  - Resource S can be allocated to any one of them:
    - When process finishes resource is freed;
    - Other process can take resource in turn and also complete;

Visual inspection allows for cycle detection:

But how can this be done by an algorithm? Any ideas?

Visual inspection allows for cycle detection:

But how can this be done by an algorithm? Any ideas?

- Many algorithms for detecting cycles in directed graphs are known.
- Lets look at a simple one;

Cycle detection algorithm (1/2)

Algorithm employs the following data structures:

- Uses a list of nodes *L*;
- Uses a list of arcs;
  - To avoid repeated inspections:
    - Arcs are marked to indicate they have been inspected;

Cycle detection algorithm (1/3)

Algorithm idea:

- Take each node as the root of what it hopes will be a tree;
- Do a depth-first search on the tree, if search:
  - Goes back to a node it has already encountered: **cycle!**
  - Exhausts all arcs from any given node:
    - Search backtracks to the previous node;
    - If search backtracks to the root: **no cycles** for current node!
    - If search backtracks to the root for all nodes: **graph is cycle free!**

Cycle detection algorithm (2/3)

For each node, N, in the graph (1/2):

1. Use N as starting node;
2. Initiate L to empty list;
3. Designate all arcs as **unmarked**;
4. Try to add current node N to the end of L:
   - If node already $\in L$: **cycle!**, algorithm terminates;

Cycle detection algorithm (3/3)

For each node, N, in the graph (2/2):

5. From the current node:
   - See if there are any unmarked outgoing arcs:
     - Pick random unmarked arc;
     - Mark the random arc;
     - Follow random arc to new node;
   - Otherwise: If current node is the initial node:
     - Graph does not contain any cycles: algorithm terminates;
   - Otherwise: **dead end**
     - Remove node from L;
     - Go back to previous node;
     - Make previous node the current one and go to step 4;

Lets see how the algorithm works in practice:

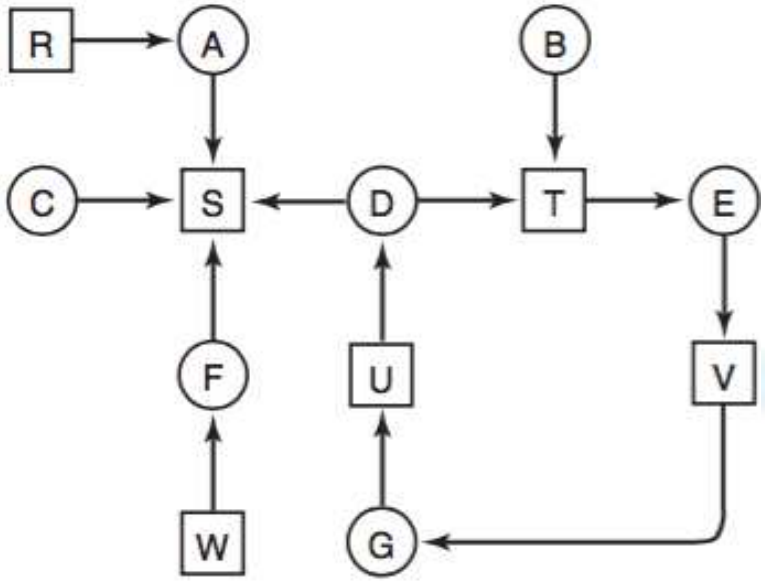

Figure: A resource graph. (Source: (Tanenbaum and Bos, 2015))

Processing with current node = R:

1. L = (R)
2. L = (R, A)
3. L = (R, A, S)
4. L = (R, A)
5. L = (R)

No cycle detected!

Lets see how the algorithm works in practice:

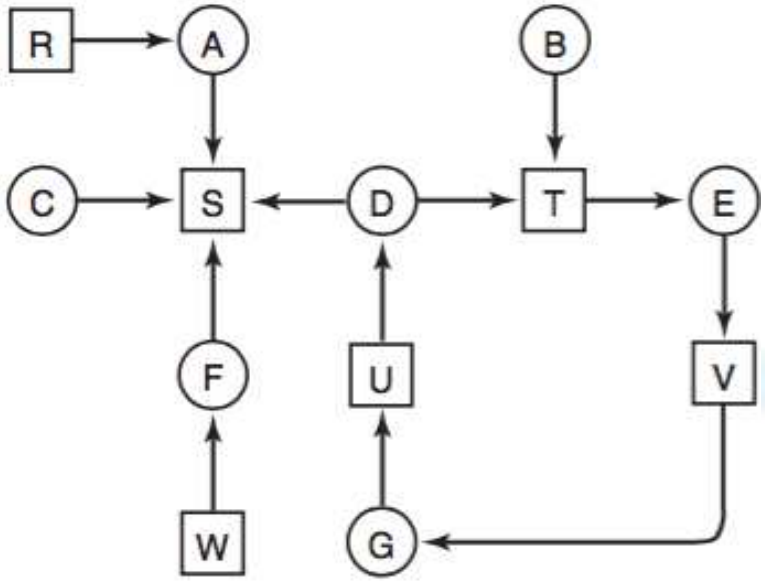

Figure: A resource graph. (Source: (Tanenbaum and Bos, 2015))

Processing with current node = A:

1. L = (A)
2. L = (A, S)
3. L = (A)

No cycle detected!

Lets see how the algorithm works in practice:

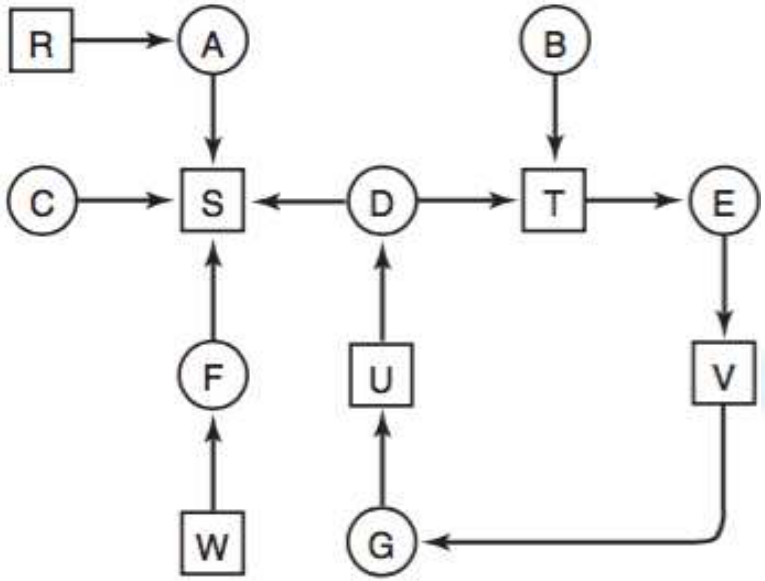

Figure: A resource graph. (Source: (Tanenbaum and Bos, 2015))

Processing with current node = B:

1. L = (B)
2. L = (B, T)
3. L = (B, T, E)
4. L = (B, T, E, V)
5. L = (B, T, E, V, G)
6. L = (B, T, E, V, G, U)
7. L = (B, T, E, V, G, U, D)
8. L = (B, T, E, V, G, U, D, T)

Cycle detected!

Deadlock Detection with Multiple Resources of Each Type

Previous section focused on:

- Deadlock detection with one resource of each type:

It is also possible to detect deadlocks when:

- Multiple resources of each type exist;
- Unfortunately: No time for that during this semester! =’(

Recovery from deadlock

Suppose that deadlock detection algorithm succeeds and detects a deadlock? What next?

Recovery from deadlock

Suppose that deadlock detection algorithm succeeds and detects a deadlock? What next?

- Some way is needed to recover!
- We will discuss various ways of recovering from deadlock:
  - Recovery through Preemption;
  - Recovery through Rollback;
  - Recovery through Killing Processes;

Recovery through Preemption

May be possible to temporarily take a resource away:

- From the current process to another process;
- Highly dependent on the nature of the resource;
- Recovering this way is frequently difficult or impossible!

Recovery through Rollback (1/2)

If developers know **deadlocks** are likely:

- Processes can be made to be **checkpointed** periodically:
- **Checkpointing** means that:
  - Process writes state to a file so that it can be restarted later, *i.e.*::
    - Process memory image;
    - Process resources' state;
  - New checkpoints should not overwrite old ones:
    - New files should be written;
    - As the process executes, a whole sequence accumulates.

Recovery through Rollback (2/2)

Do you have any idea how the checkpointing process can be used to do deadlock recovery? Any ideas?

Recovery through Rollback (2/2)

Do you have any idea how the checkpointing process can be used to do deadlock recovery? Any ideas?

Deadlock algorithm detects which resources are needed (1/2):

1. To perform recovery, a process that owns a needed resource is:
   - **Rolled back** to a point in time before it acquired resource;
   - This is done by starting at one of its earlier checkpoints;
   - All the work done since the checkpoint is **lost!**

Recovery through Rollback (2/2)

Do you have any idea how the checkpointing process can be used to do deadlock recovery? Any ideas?

Deadlock algorithm detects which resources are needed (2/2):

2. Process is reset to an earlier moment when it did not have resource;
3. Resource can then be assigned to one of the deadlocked processes;
4. If the restarted process tries to acquire the resource again:
   - Process will have to wait until resource becomes available.

Recovery through Killing Processes

Do you know of any crude but simple ways to break a deadlock? Any ideas?

Recovery through Killing Processes (1/3)

Do you know of any crude but simple ways to break a deadlock? Any ideas?

One possibility is to kill one or more processes **in the** cycle:

- With a little luck: other processes will be able to continue;
- If this does not help: repeat until the cycle is broken.

Recovery through Killing Processes (2/3)

Do you know of any crude but simple ways to break a deadlock? Any ideas?

Another possibility: kill a process **not in** the cycle:

- Choose process holding resources needed by other process in the cycle;

Recovery through Killing Processes (3/3)

Where possible:

- Best to kill a process that can be rerun from the beginning with no ill effects;
- Example 1: compilation **can** always be rerun;
- Example 2: updating a database **cannot** always be run a 2^{nd} time safely;

References I

Tanenbaum, A. and Bos, H. (2015).

Modern Operating Systems.

Pearson Education Limited.