Operating Systems and Internetworking

OSI – M30233 (OS Theme)

Week 4- Synchronization and Deadlock

Dr Tamer Elboghdadly

Synchronization and Deadlock

- **Plan:**
  - General kinds of synchronization
  - Resource deadlocks, and
  - Strategies for dealing with them
- **There is an “appendix”**
  - Notification synchronization – detailed example

Synchronization

Mutual Exclusion Reprise

- ***ME* is one of the most important kinds of *synchronization* between threads.**
  - No thread can enter a critical section while another thread is inside its own critical section.
  - Can be implemented with the aid of various software devices, e.g. *semaphores*.

Beyond Mutual Exclusion

- **Many other kinds of synchronization, e.g.:**
  - Thread i *sends a message* to thread j
    - *Receive* operation in thread j can’t complete until *send* operation in thread i is reached.
    - Sometimes *send* can’t complete until thread j reaches *receive* – depends how much buffering.
  - *Join* synchronization between parent and child threads
    - *Join* operation in parent can complete when child terminates.
  - *Barrier* synchronization across a group of *N* processes.
    - No thread can complete its *barrier* operation until all *N* threads have reached their barrier operation.

UNIVERSITY OF PORTSMOUTH

General Synchronization

- Generally "synchronization" consists in a particular thread having to wait until some condition is created by one or more other threads.
- *Dijkstra's semaphores*, introduced last week, are one rather general mechanism for achieving different kinds of synchronization.
  - Recall semaphore S is an integer supporting two operations:
    - $P(S)$ decrease semaphore, *but not below zero*
    - $V(S)$ increase semaphore.

Other Uses of Semaphores

- In ME, semaphore is usually initialized to 1.
- In following example we assume instead semaphore *S* is initialized to *0*:

Thread i	*Thread j*
⋮	⋮
A	*P*(*S*)
V(*S*)	*B*

  - Execute *B* in thread j *only after A* executed in thread i.
  - *Notification* synchronization – compare with send/receive synchronization.
  - *See appendix to this week's lecture for more!*

RESOURCE DEADLOCK

Resources

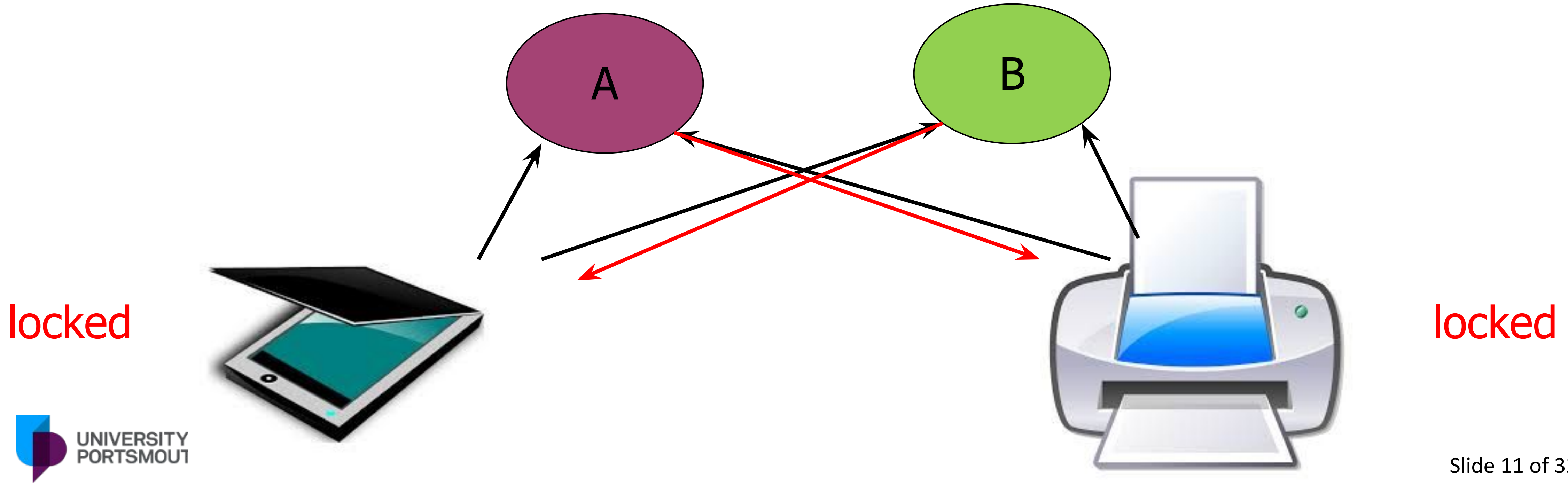
- Computer systems have many kinds of *resources*, that can only be *accessed by one process or thread at a time*.
- Examples include shared data structures in the operating system (mutual exclusion), and physical devices such as printers.
  - If two threads update a shared data structure at the same time, likely result is corruption of the data structure.
  - If two processes send data to a printer at the same time, likely output is gibberish.

Deadlocks

- *Resource deadlocks* can occur when processes (or threads) need to acquire access to more than one exclusive resource.
  - For example, a programme may need to use a scanner and a printer, and it acquires exclusive access to both these devices.
- Classic example involves two processes *A* and *B* and two resources *R* and *S*.
  - Both processes want to use both resources, but try to acquire them in different orders.

Deadlock Examples

- Example 1:
  - System has 1 scanner and 1 printer.
  - Processes *A* and *B* each hold one each of scanner and printer, and each needs the other one.



UNIVERSITY PORTSMOUTH

Deadlock Examples

- Example 2:
  - Semaphores *R* and *S*, initialized to 1.

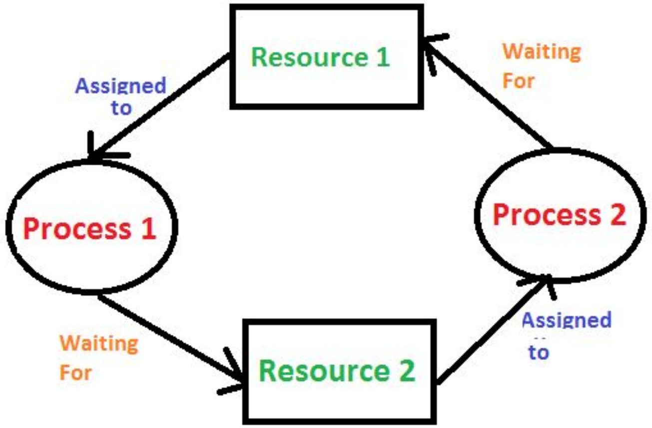
Thread i	*Thread j*
⋮	⋮
P(R)	*P(S)*
P(S)	*P(R)*
⋮	⋮

UNIVERSITY OF PORTSMOUTH

What is Deadlock?

- Deadlock is a situation where a process or a set of processes wait indefinitely for an event that can never occur.

- More specifically:
  - *A set of threads is in a resource deadlock state when every thread in the set is waiting on a resource which is being held by another thread in the set.*



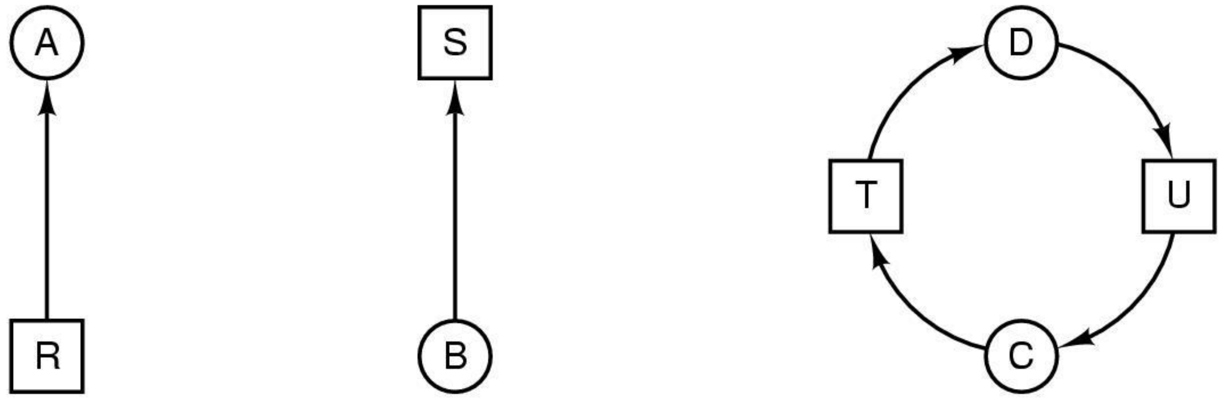
Deadlock Modelling

- Resource deadlock can be modelled using a *Resource Allocation Graph*, which shows:
  - Which processes are requesting which resources.
  - Which resources have been granted to which processes.
- If graph contains no cycles ➡ no deadlock.
- If graph contains a cycle ➡ deadlock.

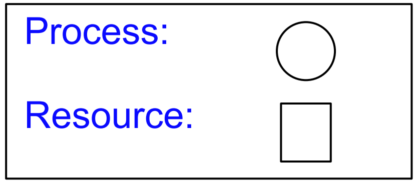
Resource Allocation Graphs

- Directed graphs†:

Note: Directions of the arrows are important.



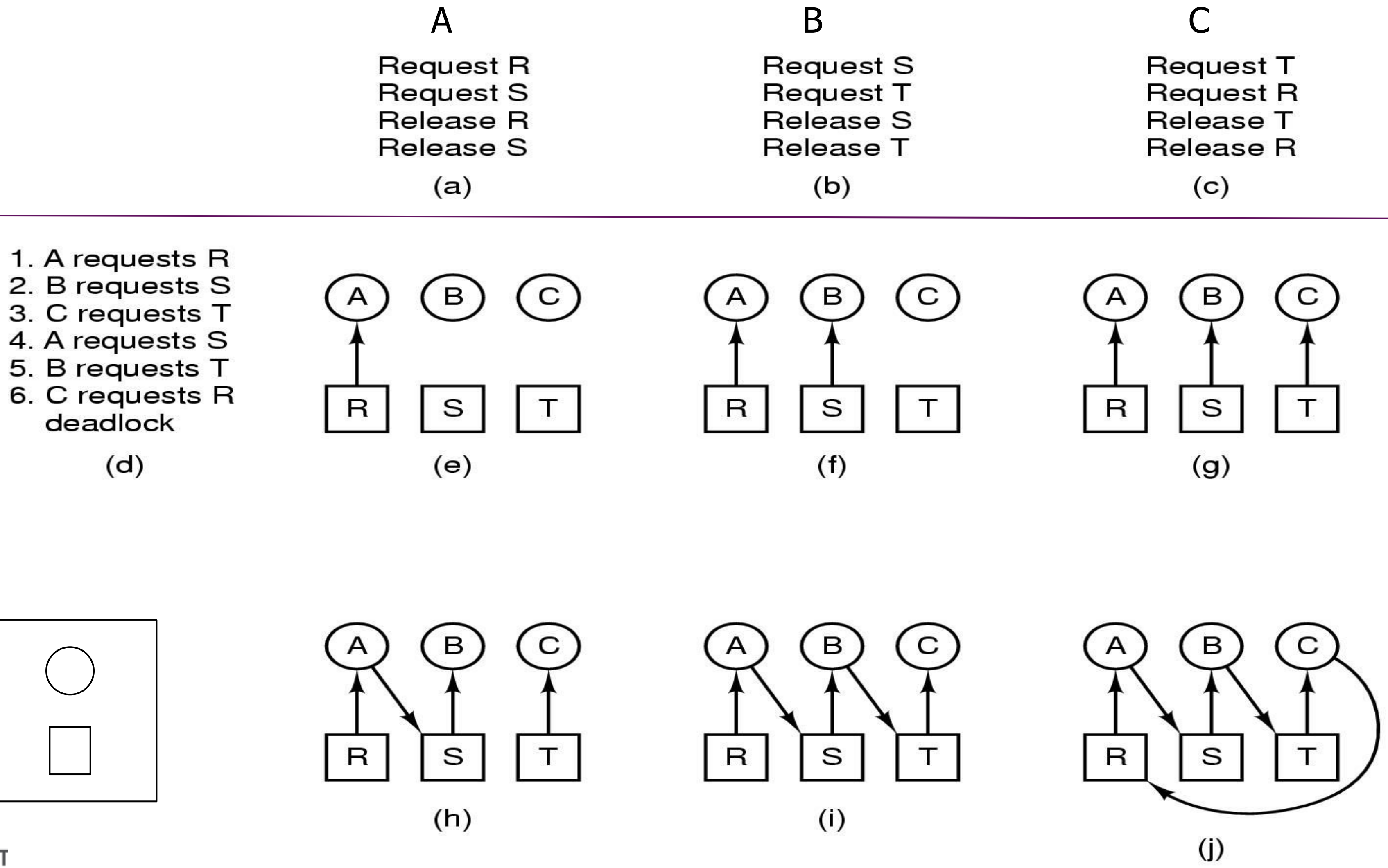
(a) (b) (c)



- resource *R* **assigned** to process *A*
- process *B* is **requesting/waiting** for resource *S*
- process *C* and *D* are in **deadlock** over resources *T* and *U*

†Tanenbaum, MOS, Fig 6-3

How A Deadlock May Occur†



No Deadlock

Deadlock

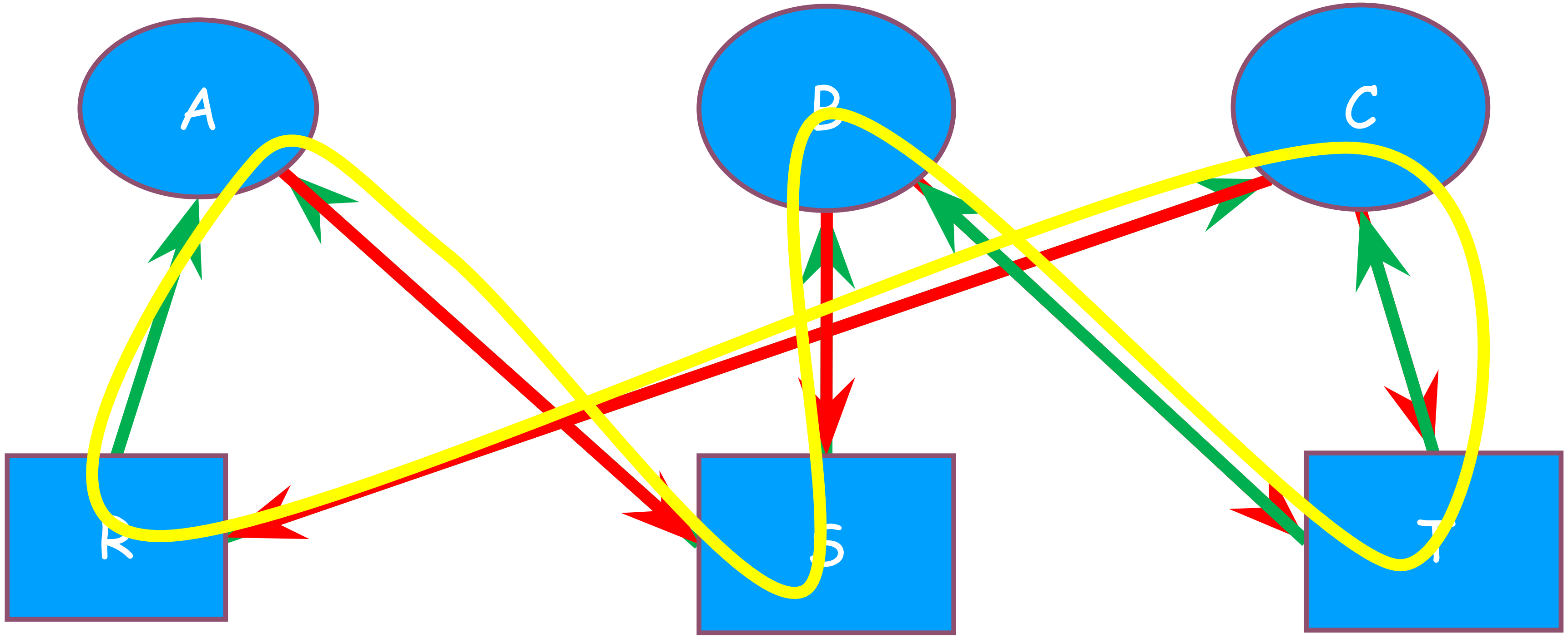
Process:

Resource:

†Tanenbaum, MOS, Fig 6-4

Example Program Trace

Thread B waiting for S
Thread B acquired S
Thread B waiting for T
Thread B acquired T
Thread A waiting for R
Thread A acquired R
Thread A waiting for S
Thread C waiting for T
Thread B released S
Thread B released T
Thread A acquired S
Thread C acquired T
Thread C waiting for R
Thread B waiting for S
Thread C acquired R
Thread A released R
Thread A released S
Thread B acquired S
Thread C released T
Thread C released R
Thread B waiting for T
Thread B acquired T
Thread C waiting for T
Thread B released S
Thread B released T
Thread C acquired T
Thread A waiting for R
Thread A acquired R
Thread B waiting for S
Thread B acquired S
Thread A waiting for S
Thread C waiting for R
Thread B waiting for T

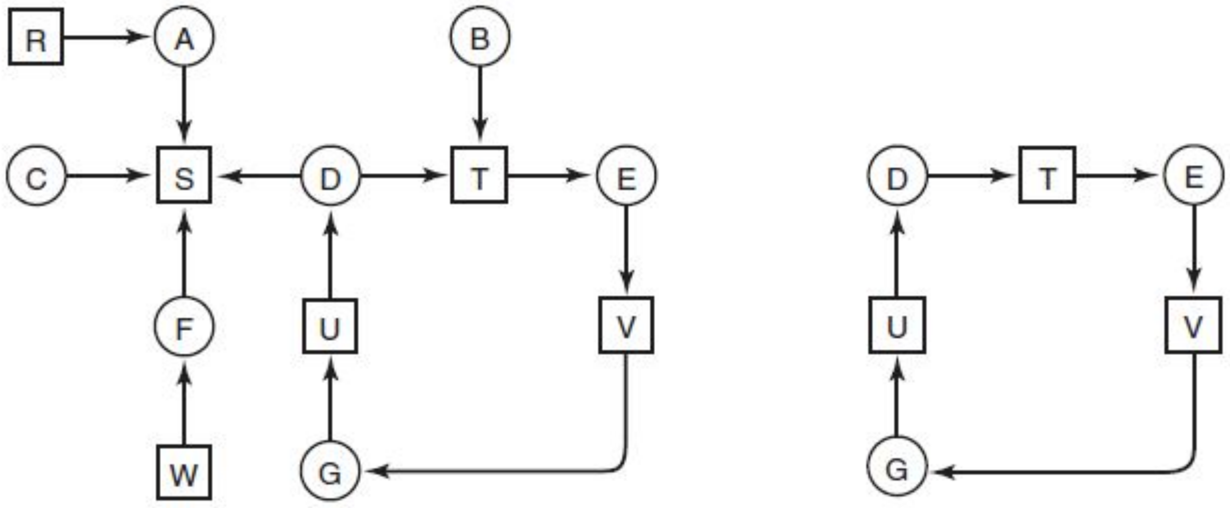


DEALING WITH DEADLOCK

Deadlock Detection and Recovery

- **Basic idea:**
  - Allows system to enter deadlock state.
  - Runs Detection algorithm periodically to check.
  - Performs Recovery scheme if deadlock.
- **To detect deadlock, search for a cycles in the Resource Allocation Graph.**

Deadlock Detection†



(a) (b)

- The graph a) contains the cycle b).
- This indicates deadlock

†Tanenbaum, MOS, Fig 6-5

Deadlock Recovery

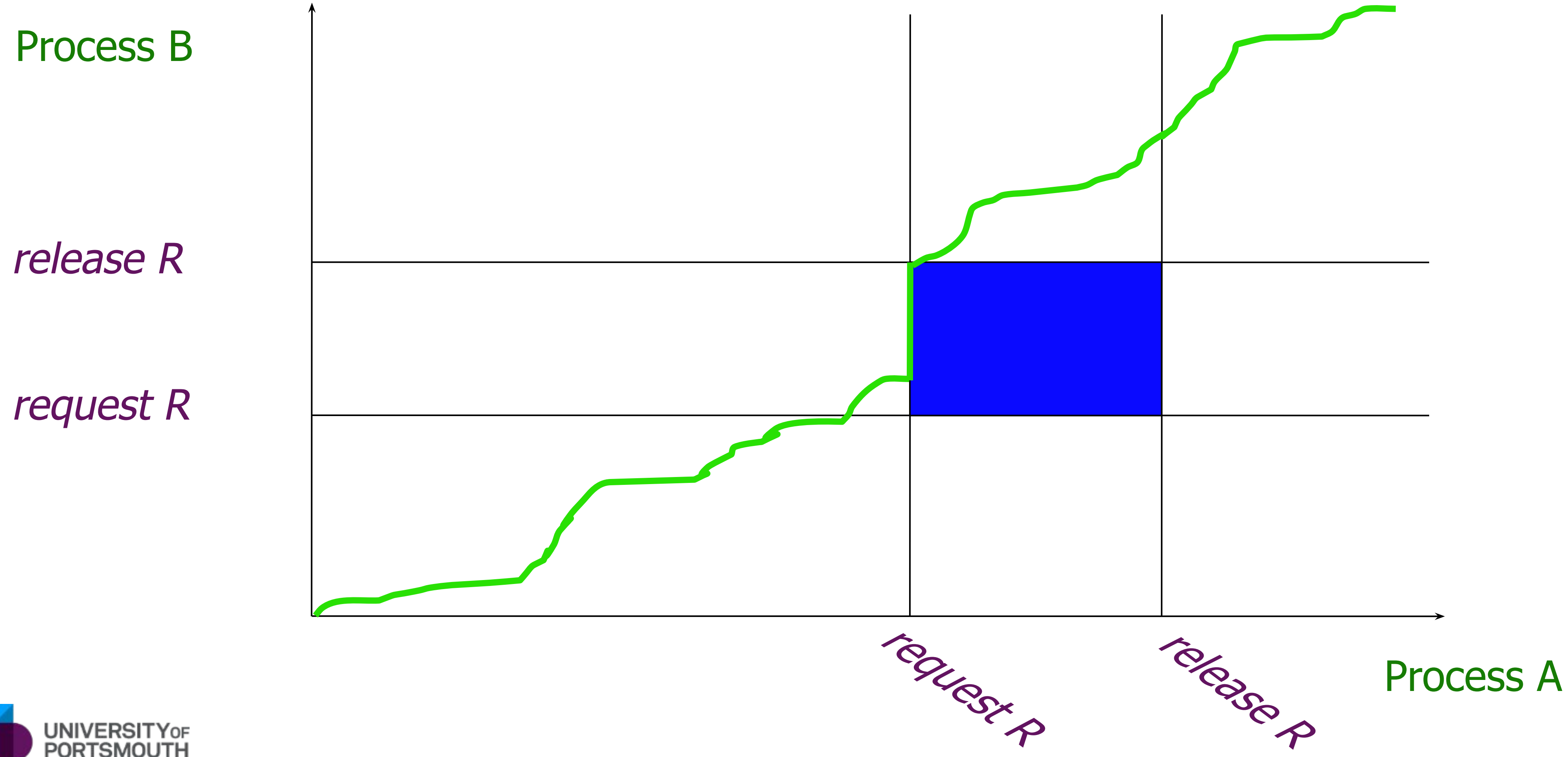
- Crudely - kill processes until the deadlock cycle is eliminated.
- “Survivors” get the resources.
- Commonly used in *Relational Database Management Systems*, where transactions identified as causing deadlock can be “rolled back”.

UNIVERSITY OF PORTSMOUTH

Deadlock Avoidance

- The idea here is to keep the system "safe" – avoid entering "unsafe" states which may later turn into a deadlock.
- Best illustrated by an example.
  - Consider the classic deadlock scenario of two processes sharing resources
  - Each process has its own "instruction counter".
  - If we plot these as two axes of a graph, behaviour of whole system is represented as a trajectory in this graph, moving upwards and to the right.

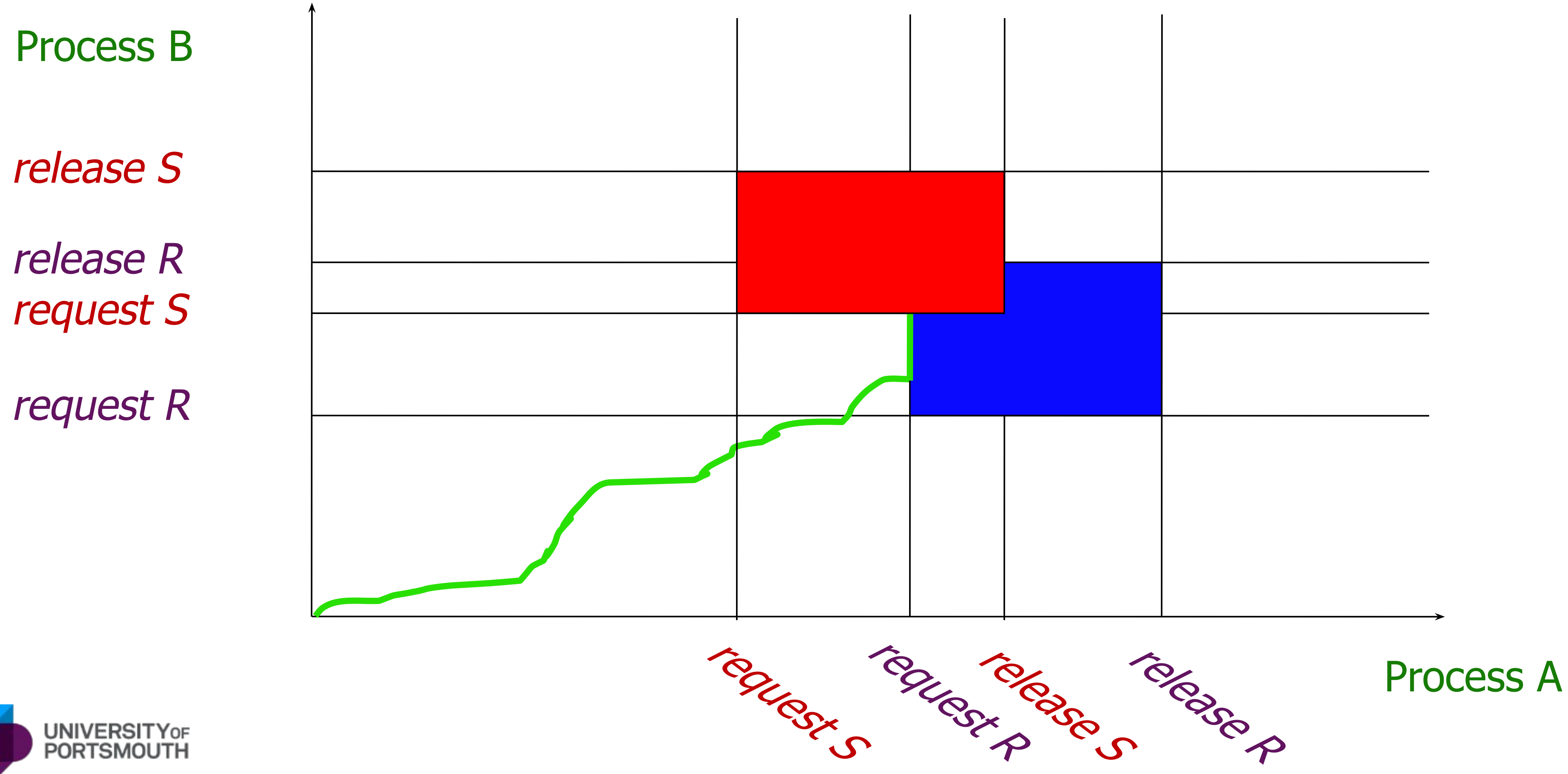
Two Processes and one Resource



Remarks

- Both processes request *R*, then later release it.
- The green trajectory is a possible evolution of the system as a whole – it is "moving" upwards and to the right (**can never go down or to the left**, because that implies process *A* or *B* is "going backwards").
- Blue region is forbidden by mutual exclusion (note *A* reaches this critical section last, and must wait).

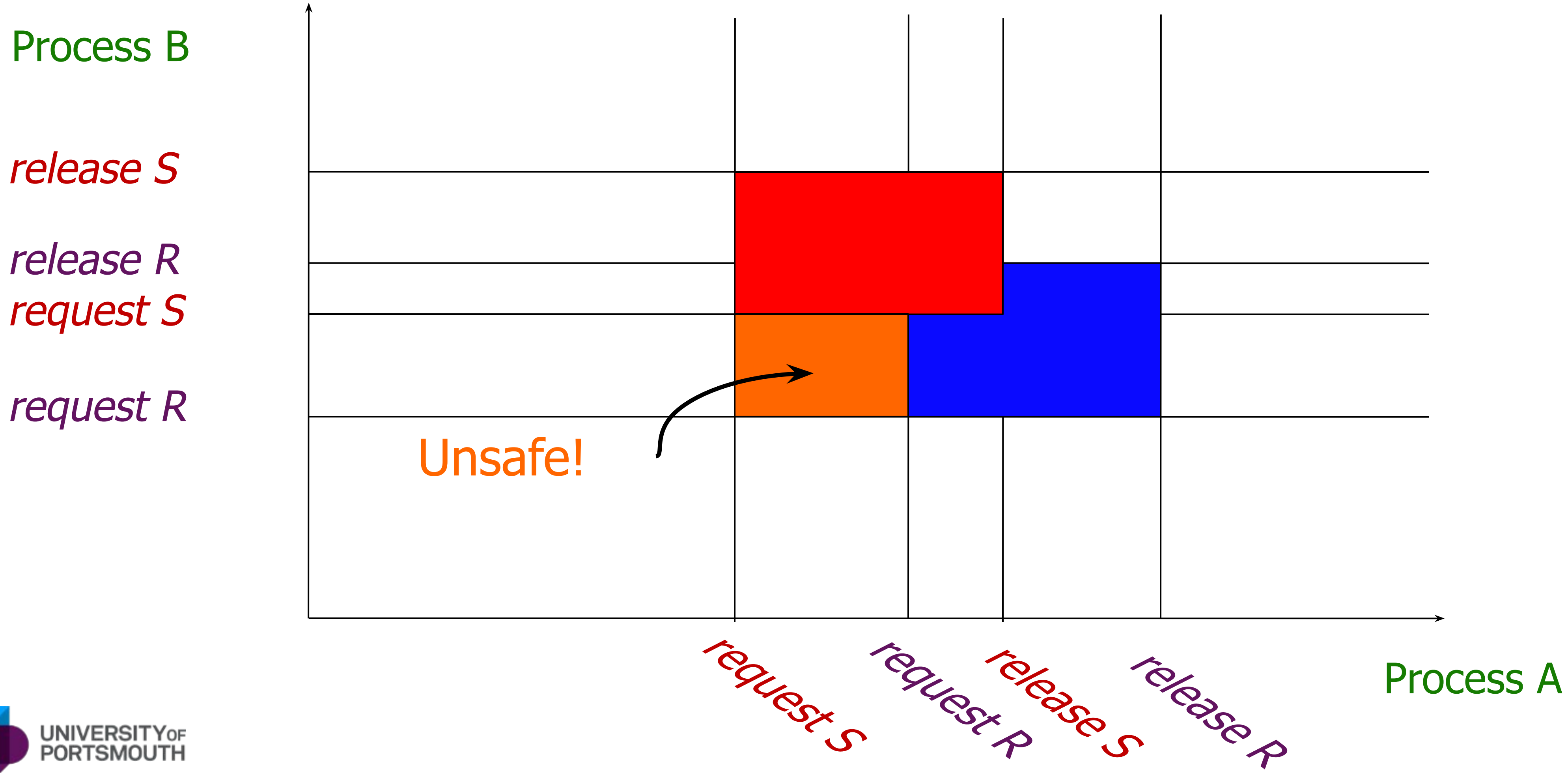
Two Processes and two Resources



Remarks

- Process *A* requests *S* then *R*; Process *B* requests *R* then *S*.
- Blue region is forbidden by mutual exclusion on *R*; Red region forbidden by mutual exclusion on *S*.
- The green trajectory is now blocked when it reaches the intersection of the two forbidden regions
  - system can only evolve *up* or *to right* - means it can't evolve at all from here!

Unsafe Region



Safe and Unsafe Regions

- The amber region in the previous graph is an *unsafe region*.
- If the system *ever* enters this region, it will *inevitably* be driven to the point of deadlock
  - because it can't move "down" or "to the left".
- A *deadlock avoidance* scheme should delay acquisition of any resource, if acquiring it would allow the system into an unsafe region.

Banker’s Algorithm (Dijkstra,1965)

- Basic idea:
  - Requires all processes to declare the max # of resource units that they may request.
  - Keeps track of current allocation for each process and their current “needs” (“max” – “has”).
  - When receives a request, “pretend” to honor the request, and
  - Try to fulfill the “needs” of all other processes in some order so as to check whether or not granting the request will lead to a safe state.
  - If safe, grant request; otherwise, deny request.
- Read Modern Operating Systems, sections 6.5.3 – 6.5.4

Summary

- General synchronization
  - Beyond mutual exclusion, and more on semaphores
  - Notification using monitors
- Resource deadlocks, and how to deal with them
  - Detect and recover (Detection algorithm).
  - Avoid by not entering an unsafe state (Banker’s algorithm),
- Next lecture: *Processes and scheduling*

UNIVERSITY OF PORTSMOUTH

Further Reading

- Andrew S. Tanenbaum, "*Modern Operating Systems*", 4th Edition, Pearson, 2014 (MOS), Chapter 6.

UNIVERSITY OF PORTSMOUTH

Questions?