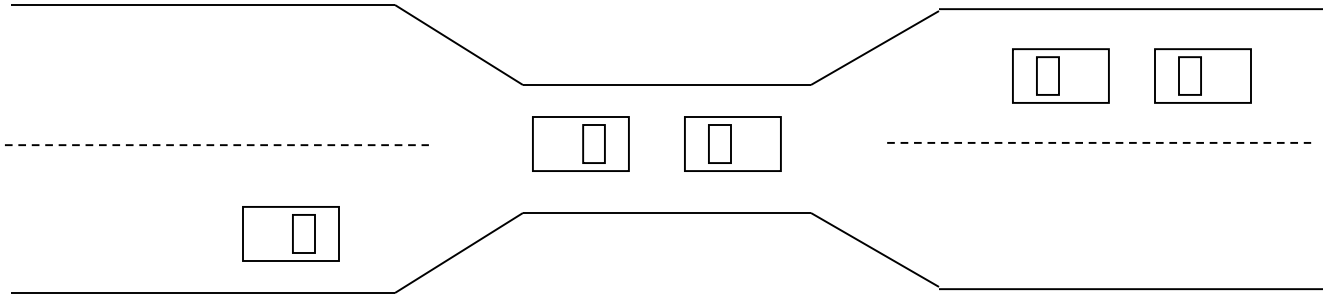
Agenda

- Assignment 3 due November 10
- Feedback for Assignment 2, Midterm on Brightspace
- Seminar on Deep Learning and Music
  - Dr. Sageev Oore, Google Visiting Research Scientist
  - Today: 11: 30 am – 1:00 pm
  - ME 107
- Today's lecture
  - Dealing with Deadlocks
    - Prevention
    - Avoidance
    - Detection and Action
  - IPC (Revisit)

Where we are…

- The OS provides three key abstractions
  - Process → CPU
  - Memory (Address) Space → RAM
  - Files → Secondary storage, Network, and Peripheral devices

Deadlock Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

Deadlock Strategies

Prevention

- Conservative
- Limit access to resources
- Impose restrictions on processes

Avoidance

- It is less restrictive than deadlock prevention
- Maximum resource requirement for each process must be stated in advance
- Fixed number of resources to allocate
- Process holding resources may not exit

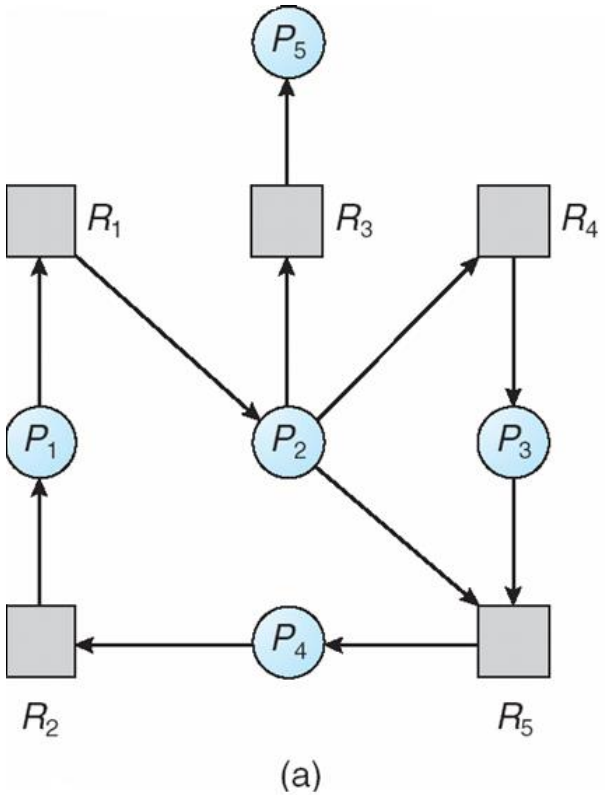
Deadlock Detection and Recovery

- Grant request whenever possible
- Periodically check for deadlock state
- Recover from deadlock
- Need to do two things:
  - Detect when deadlock has occurred
  - Recover from deadlock

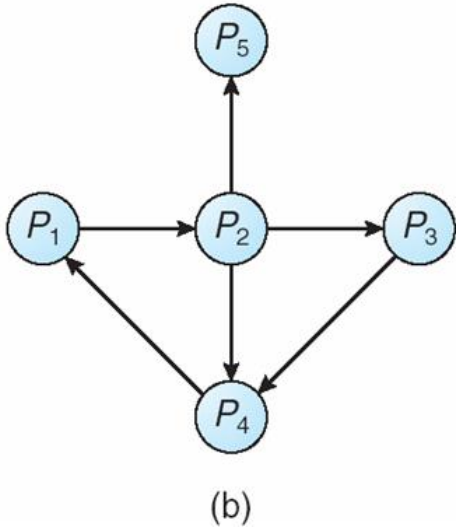
Deadlock Detection Algorithm

- Use variants of RAG or Banker's Algorithm (Single Instance Resource)
- Maintain *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Early detection
  - Periodically invoke an algorithm that searches for a cycle in the graph
  - If there is a cycle, there exists a deadlock
  - Simple to implement
- But consumes considerable processor time
  - requires an order of n^2 operations, where n is the number of vertices in the graph
- Trade-off between overhead and deadlock detection!

Example



Resource-Allocation Graph



Corresponding wait-for graph

Deadlock Detection: 1 Resource of Each Type

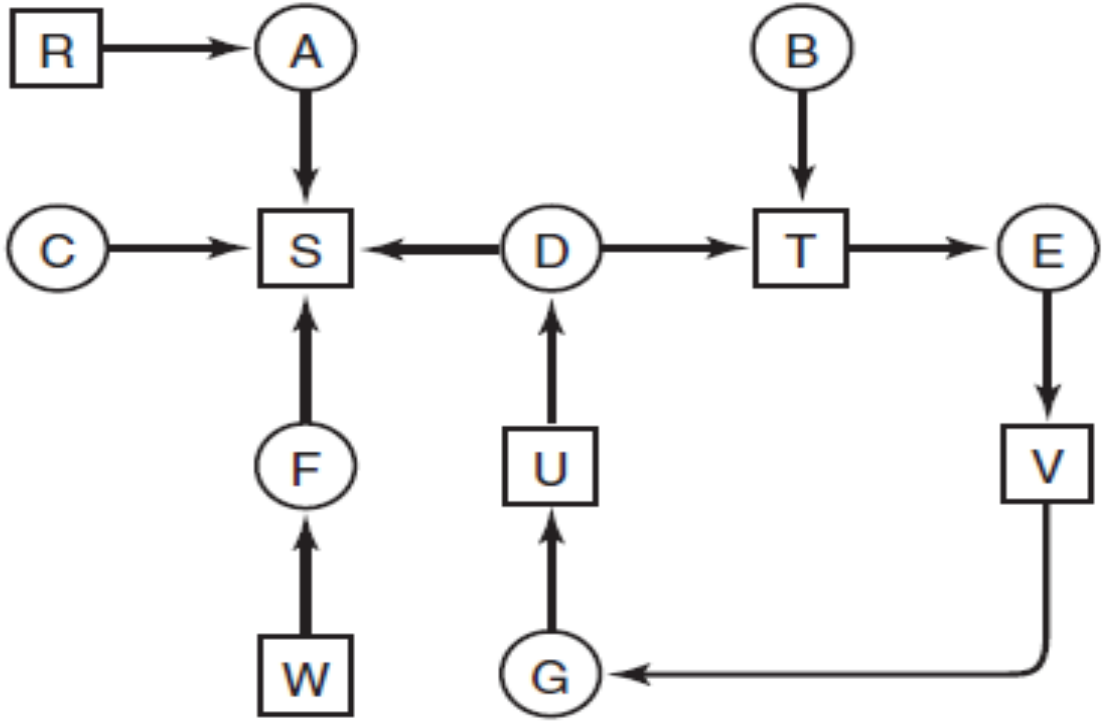
Example of a system – is it deadlocked?

1. Process A holds R, wants S
2. Process B holds nothing, wants T
3. Process C holds nothing, wants S
4. Process D holds U, wants S and T
5. Process E holds T, wants V
6. Process F holds W, wants S
7. Process G holds V, wants U

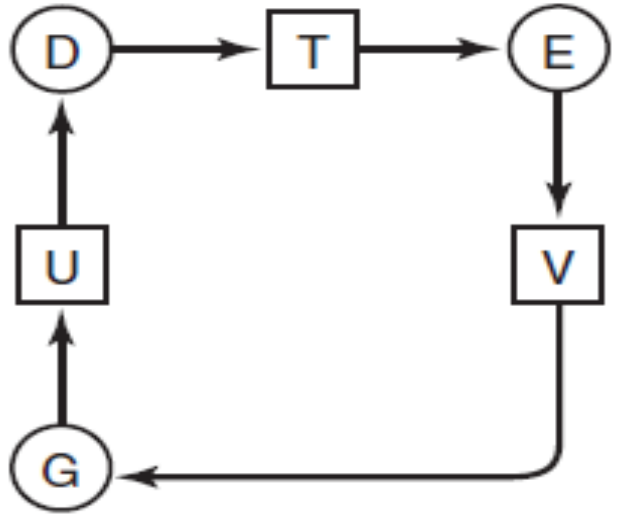
Deadlock Detection: 1 Resource of Each Type

- Process E holds T, wants V
- Process F holds W, wants S
- Process G holds V, wants U

- Process A holds R, wants S
- Process B holds nothing, wants T
- Process C holds nothing, wants S
- Process D holds U, wants S and T



(a) RAG.



(b) A cycle extracted from (a).

Deadlock Detection: Multiple Instances of Each Resource Type

- **Available***:* A vector of length *m* indicates the number of available resources of each type.
  - Available [*j*] = *k* means there are *k* instances of resource type R_j available
- **Allocation***:* An *n* x *m* matrix defines the current resource allocation to each process.
  - Allocation[*i,j*] = *k* means P_i is currently allocated *k* instances of R_j
- **Request***:* An *n* x *m* matrix indicates the current request of each process.
  - *Request* [i_j] = *k* means process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

- Step1: Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
  - *Work* = *Available*
  - For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*
- Step 2: Find an index *i* such that both:
  - *Finish*[*i*] == *false*
  - $Request_i \leq Work$
    If no such *i* exists, go to step 4
- Step 3:
  - $Work = Work + Allocation_i$
  - *Finish[i*] = *true*
    go to step 2
- Step 4: the system is in deadlock state
  - If *Finish*[*i*] == false, for some i, $1 \leq i \leq n$
  - Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Example

- Step 1:
  - Work = (00001)
  - P4 has no allocated resources → P4 is marked Finish4 = true
- Step 2 : the request for P3 is less than or equal to Work
- Step 3:
  - P3 is marked → Finish3 = true
  - Work = Work + Allocation3 = (00001) + (00010) = (00011)
- No other process satisfy step 2
- Step 4: P1 and P2 unmarked
  - P1 and P2 are deadlocked

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Recovery Strategies

- **Abort** all deadlocked processes
- **Rollback:** back up each deadlocked process to some previously defined checkpoint and restart all processes
- **Killing Processes:** successively abort deadlocked processes until deadlock no longer exists
- **Pre-emption:** successively pre-empt resources until deadlock no longer exists

Recovery Strategies

- How to choose from which processes to kill/pre-empt resources?
- Victim process selection criteria
  - Least amount of processor time consumed so far
  - Least amount of output produced so far
  - Least total resources allocated so far
  - Lowest priority

Discussion

- There is no user-friendly way to prevent or avoid deadlock
- The most common solutions in noncritical systems is to let deadlock happen and if they do, let the user decide which process to kill
- Ideally, we should write code that does not deadlock
- That is, our processes and threads should synchronize properly when necessary

Synchronization

- Idea: threads/processes work together to solve a problem
- Goals:
  - Ensure mutual exclusion
  - Avoid race conditions
  - Allow progress
    - Avoid starvation
    - Avoid deadlock
  - Avoid depending on the scheduler
  - Maximize concurrency
- Trade-off: simpler solutions reduce concurrency
- Question: How do we find good solutions?

Interprocess Communication (Revisited)

- Interprocess Communication (IPC) involves concurrent processes exchanging data among each other
- Several common forms of IPC
  - **Shared Memory**: processes communicate by reading/writing shared memory
  - **Message Passing**: processes communicate by sending messages to each other
  - **Signals**: processes can asynchronously notify another process
  - **Pipes**: special form of message passing

Shared Memory

- Model we have been discussing so far
- General purpose IPC system
- All processes/threads run on the same machine
- All processes/threads have access to the same memory
- Analogy: A black board in a classroom
- Most common type of IPC, because
  - Can be done locally
  - Natural form of concurrency
  - Little reliance on underlying system
  - Very efficient and fast (low overhead)

Challenges with Shared Memory

- Problem: consistency of the data being read/written
  - E.g. incrementing a counter
- Solution: Mutual Exclusion
  - Use locks, semaphores, or monitors to protect critical sections
- Other Problems:
  - Does not work across multiple machines
  - Hard to debug
  - Not always easy to reason about
  - Hard to get right
- We can solve many of these problems by using message passing

Message Passing

- Features:
  - Post-office or telephone model
  - Processes can run on same or different machines
  - Processes cannot read each others' memory
  - To communicate, processes ask the system to send a message on their behalf
  - Real life example: the Internet

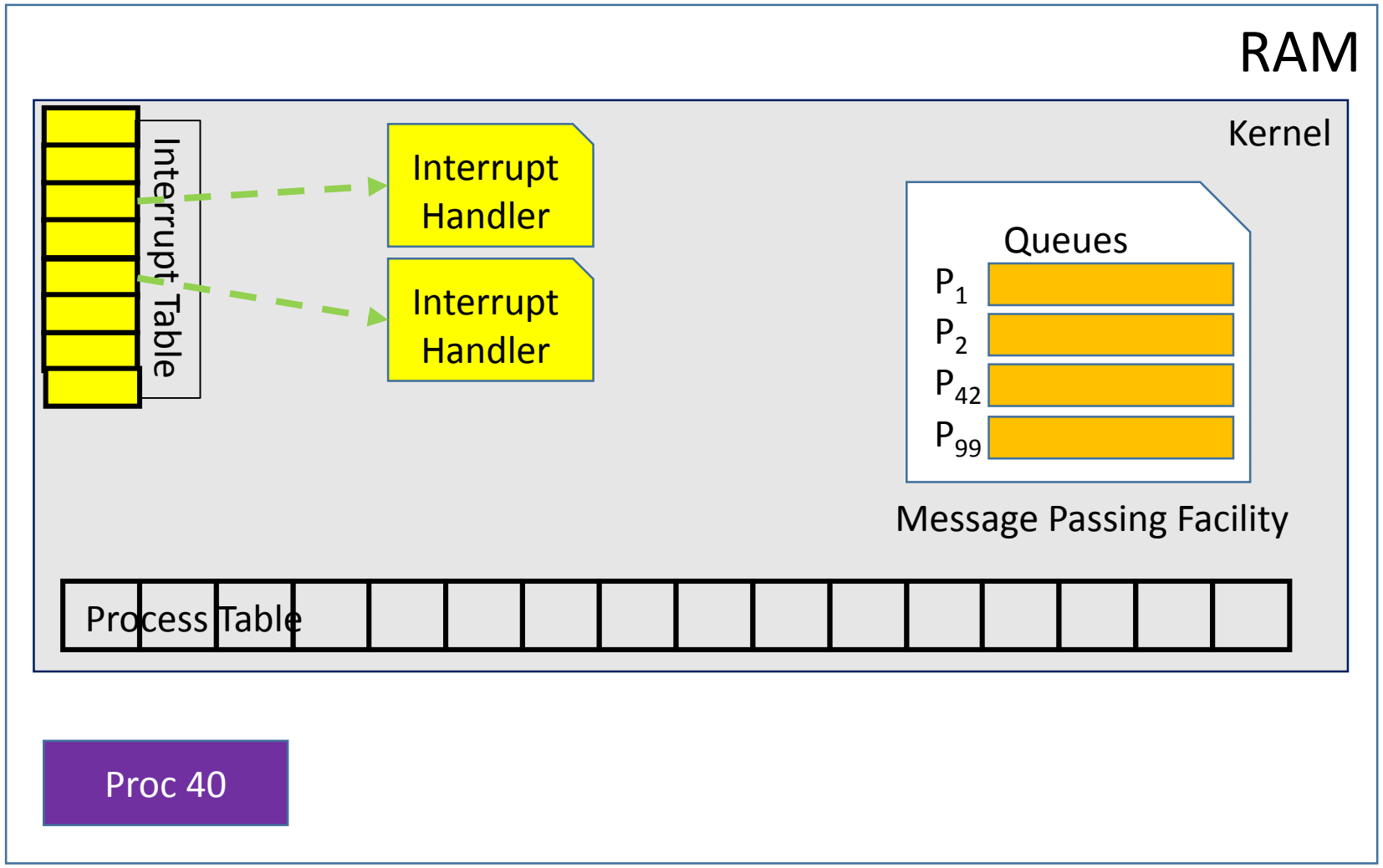
The Post-Office Analogy

- Sending a postcard (message) to someone
  - Create postcard (message)
  - Put destination and return (source) address on it
  - Drop it into a post office box
- Post office delivers the post card to receiver's mailbox
- The receiver then
  - Retrieves postcard (and other letters) from mailbox
  - Reads the postcard
- Processes (on different machines) can communicate with each other in a similar fashion

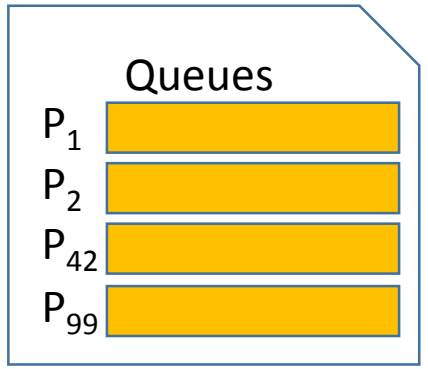
Message Passing Operation

- The system (OS) provides a “post-office” facility
- A process can “send” or “receive” a message via the facility
- Operations
  - `send(dst, msg)`: send message to process specified by dst
  - `recv(src, msg)`: receive message from process specified by src
  - `recvany( msg )`: receive message from any process
- Questions:
  - How do we identify processes?
  - How do processes learn about each other?
  - Should the sender block until message is received?
  - Should the receiver block until message is sent?

The Message Passing Facility



The Message Passing Facility



Message Passing Facility

- Queue P_i stores messages (or senders) to the process P_i
- The head of queue P_i is the next message to be received by P_i
- To send a message processes
  - must know the ID or name of the process they are sending to
  - ask the kernel to enqueue the message on the corresponding queue
- To receive a message, it is not (always) necessary for the process to explicitly specify the receiver
  - The process asks the kernel to remove a message from their queue and return it
- Analogy: A queue is the same as a mailbox

Process Naming

- How processes are named and how process names are discovered is a whole topic beyond scope of this course

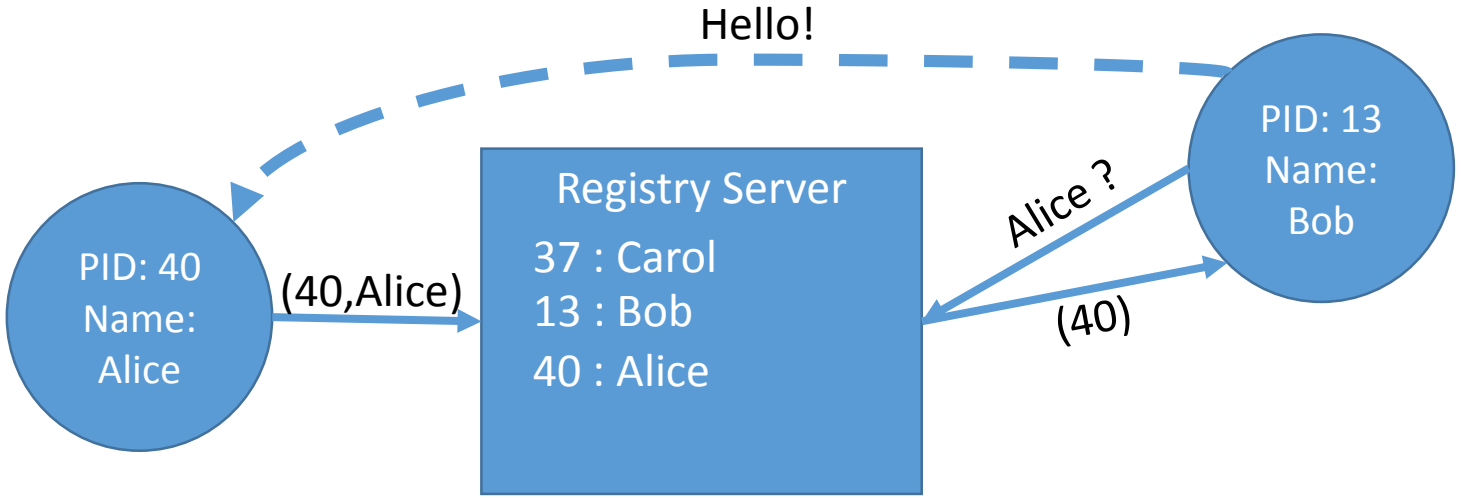
  Example: On the Internet

  - Processes are named by IP/port #
  - Use DNS to look up IP addresses and port #s of specific processes: e.g., 25 for e-mail (SMTP)
- Other Examples:
  - On a single system each process has unique process ID (pid) assigned by the OS
  - Each thread has a unique thread ID (tid), assigned by the OS or run-time system such as Java
- Another idea is to use a registry

Name Registry

- Analogy: A phone book
- Processes register their names on a central server
- Other processes can query server to find names

  E.g., this is what DNS is



Should Processes Block?

- When a process sends a message, should it block until the message is received?
- When a process receives a message, should it block until the message arrives?
- Answers depend on the system:
  - Typically, receivers will block until the message arrives, why?
  - Whether senders should block depends on what type of behaviour is desirable:
    - Synchronous sends
    - Asynchronous sends