# Deadlock and starvation

Stefan D. Bruda

CS 464/564, Fall 2023

# CLIENT-SERVER SYSTEMS CAN FAIL

- Temporary blocking: a computation blocks until an event happens
  - This event will happen eventually
- Deadlock: a computation blocks until an event happens.
  - But the event never happens
  - Computation has no chance to proceed; this is a permanent failure
  - Typically, this happens when we have a set of processes (or threads) in which each component is blocked waiting for a resource that is held by another component in the set.
- Test for whether deadlock has occurred: will an external input allow computation to proceed?
- Deadlock can result from:
  - Ambiguous protocol specification
  - Programming errors and oversight

# DEADLOCK DETECTION

- Can we detect deadlock at runtime?
- For a system running on one machine: impossible
  - We have to distinguish between temporary blocking and deadlock
  - The programmer can invent new resources (e.g., mutexted variables), so the operating system has no way of knowing which program uses what resource
  - Deadlock can depend on the order in which events arrive. So even if we know everything about resources, we still have to do something similar to proving a theorem.
- For a distributed system: impossible
  - All of the above
  - We also have to inspect multiple programs running on multiple machines, possibly under different conditions (e.g., different operating systems)
  - Deadlocks can occur in a distributed system even if the individual programs are deadlock-free!
- No practical program can be designed to determine whether a set of clients and servers are deadlocked
- The only thing we can do is to minimize the possibility of deadlock
  - Recipe: care at all the levels (protocol, coding, installation)
- To avoid deadlock, one must first understand how can it occur
  - We therefore discuss (some of) the situations that can deadlock the system

# SINGLE INTERACTION DEADLOCK

- The simplest form of deadlock, and thus the easiest to prevent
- We use the request-response paradigm, and timeout values
- Request-response: one side (usually the client) sends a request, the other responds
  - The protocol must specify which side sends the request
  - A protocol that does not fully specify such kind of synchronization rules is prone to failure
  - Example of protocol featuring imprecise specifications:
    1. The client establishes a connection to the server
    2. Immediately after the connection is established, either the server of the client sends an initialization message, to which the peer responds
    3. Then the interaction happens normally (client sends requests, server responds)
  - The protocol allows flexibility in its implementation, but two implementations can easily collide and generate a deadlock
- All the interaction sequencing must be precisely specified and implemented
  - Typical approach to sequencing: describe the protocol using finite automata

# Single interaction deadlock (cont'd)

- Other problem: inherent unreliability of the communication medium
  - Typical manifestation: A message is lost, whomever is supposed to receive it deadlocks (and in turns deadlocks its peer)
  - Variant: An incomplete message is sent (e.g., a line without the terminating newline); if the peer expects the complete message, it deadlocks
- This problem is mostly manifest when using an unreliable transmission protocol (UDP), but it can happen anywhere
  - In no circumstance should you use TCP algorithms with UDP connections
- Solution:
  - Timeout: If too much time passes without any reaction from the peer, the program times out
    - The connection is considered dropped (TCP only), or
    - Mechanisms for retransmission are provided (does not make much sense in TCP applications)
  - Choosing the right timeout value is black magic; there is no recipe

# Single interaction deadlock (finale)

- If we use a concurrent server, single interaction deadlock is not critical in many cases
  - Often, only one thread/process and the corresponding client deadlocks
  - However, this eats up resources, possibly preventing other clients to connect!
  - Additionally, things get really hairy if a thread/process deadlocks within a critical region!
  - So even if single interaction deadlock is apparently unimportant, one must still pay attention to such a possibility
- A related problem: starvation
  - Some clients can access the service, while others cannot (i.e., some clients starve)
  - A real problem in iterative servers, but also an issue in concurrent ones
  - An iterative server must not permit arbitrarily long interactions; a concurrent server should in principle do the same (why?)
  - Timeout mechanisms are applicable here too (but are not panacea)

# Busy connections

- Clients that do not time out can still generate starvation
  - TCP ensures flow control; data is written by the program in a buffer and then transmitted at a pace the client can handle
  - A clients that refuses to read the responses (or reads them slowly) can delay or even prevent further transmission (machine-wide!)
  - Same goes for a client that overwhelms the server with data
- Avoiding blocking operations
  - During long operations, the server can poll periodically the input and read from the socket even if it has no use for the incoming data at that moment
  - A server can also avoid blocking operations
    - For instance, poll the socket, and send only if there is room
    - Also implement a timeout mechanism to take care of the case in which the client never reads the responses

- The first kind of starvation (client just staying there and doing nothing) is solved by not managing concurrency. . .
  - Then a malicious client will just block the thread that handles it, no problem, there are more where this one came from; the thread does not even get scheduled, so there is no overhead
  - . . . Apparently!
- Resources (including concurrency) are managed anyway at machine level
  - Managed resources include: processes, active sockets, total number of descriptors
  - It is also the case that each TCP connection uses buffer space
  - So if you do not manage your resources you just exchange one problem (a misbehaving server) to another with is much worse (a misbehaving machine)
  - Conclusion: in all but the most trivial cases concurrency should be managed

# Report problems

- The maximum amount of resources available is not the same between multiple operating systems—indeed it may not be the same even for identical machines running the same OS!
- In other words, you cannot anticipate whether your server runs out of resources
- So arrange that your server report problems
  - Check the values returned by all the system calls and generate appropriate log messages
    - Even if the error is not critical, do generate a message.
    - The system administrator can then examine the system logs and react to unusual conditions
  - Yes, you may want to consider even calls to `fork` or `new`
    - An error condition here should probably generate a `LOG_EMERG` syslog entry

# LIVELOCK

- Scenario:
  - Syslog obtains the timestamp of each log entry from a time server
  - The system administrator decides to debug the time server; debug information goes to syslog
  - A log entry generates a request from the time server, which in turn generates other log entries, which in turn generates fresh requests to the time server, which in turn. . .
- Something like a deadlock (i.e., caused by circular dependencies), but not really a deadlock
  - Indeed, the servers are not blocked; they all work like crazy; as soon as a message is processed, another one arrives
  - This situation is called livelock
- Solution: avoid circular dependencies by documenting dependencies

# UNDERSTAND AND DOCUMENT DEPENDENCIES

- Almost anything is a client-server application these days
- A programmer should
  - Understand the existing dependencies and avoid introducing cycles
  - Document the newly introduced dependencies
- Approaches in keeping dependency information:
  - Coarse-grained: services are the working entities
    - If syslog uses the time service then the time service cannot call syslog no matter what
    - Advantage: the resulting dependency graph is easy to manage
    - Disadvantage: may introduce stronger restrictions than necessary
  - Fine-grained: servers are the working entities
    - If syslog server $Y$ uses the time server $X$, then time server $X$ cannot use the syslog server $Y$ (but can use, say, syslog server $Z$)
    - Advantage: does not introduce unnecessary constraints
    - Disadvantage: the resulting dependency graph is nightmarish to manage

# A LAST WORD: NON-DISTRIBUTED DEADLOCK

- It is not necessary to have a client-server application to obtain deadlocks; a multithreaded program will do nicely
- The potential problem: critical regions (mutexes, semaphores, etc.)
- Simple rules to avoid deadlocks:
  - If you must acquire more than one critical region simultaneously, always acquire them in the same order
    - Do not rely on “this can never happen!” It can happen, and will do so at the worst possible moment
  - Make sure that you release the critical region eventually
    - Common problem: returning from within a critical region without releasing it; returning also means exiting with an error
    - Remember, a mutex is just an integer which is, say, incremented once acquired; if your thread returns/dies/is canceled without releasing the thing, nothing is there to release it for you;
    - Another common problem: acquiring critical regions in signal handlers
    - Signal handlers fire up asynchronously, so there is a decent chance that one will fire up while the main code is. . . in the critical region the handler is supposed to acquire