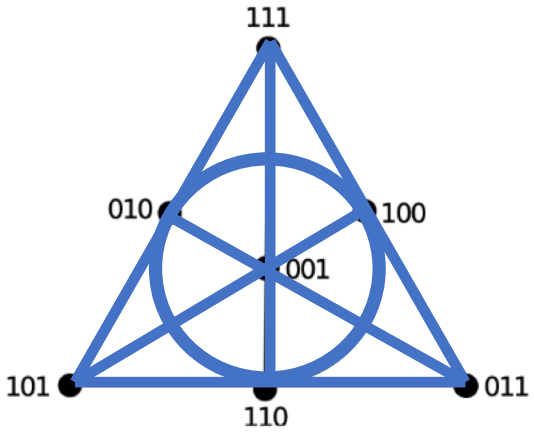
Computer Systems

Exercise 8

Last exercise

1.2 A Quorum System

Consider a quorum system with 7 nodes numbered from 001 to 111, in which each three nodes fulfilling $x \oplus y = z$ constitute a quorum. In the following picture this quorum system is represented: All nodes on a line (such as 111, 010, 101) and the nodes on the circle (010, 100, 110) form a quorum.



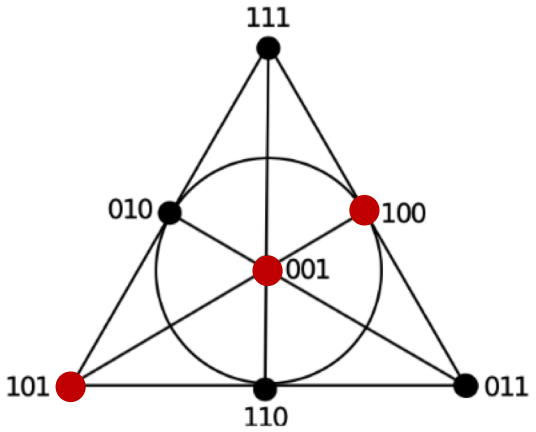
a) Of how many different quorums does this system consist and what are its work and its load?

Quorums: 7 Work: 3 Load: 3/7

Last exercise

1.2 A Quorum System

Consider a quorum system with 7 nodes numbered from 001 to 111, in which each three nodes fulfilling $x \oplus y = z$ constitute a quorum. In the following picture this quorum system is represented: All nodes on a line (such as 111, 010, 101) and the nodes on the circle (010, 100, 110) form a quorum.



a) Of how many different quorums does this system consist and what are its work and its load?

b) Calculate its resilience f. Give an example where this quorum system does not work anymore with $f + 1$ faulty nodes.

Quorums: 7 Work: 3 Load: 3/7

Resilience: 2, because every node is in 3 quorums, so two nodes can be contained in at most 2*3 quorums

Last exercise

1.3 Uniform Quorum Systems

Definitions:

s-Uniform: A quorum system $\mathcal{S}$ is *s-uniform* if every quorum in $\mathcal{S}$ has exactly s elements.
Balanced access strategy: An access strategy Z for a quorum system $\mathcal{S}$ is *balanced* if it satisfies $L_Z(v_i) = L$ for all $v_i \in V$ for some value L.

Claim: An s-uniform quorum system $\mathcal{S}$ reaches an optimal load with a balanced access strategy, if such a strategy exists.

a) Describe in your own words why this claim is true.

Idea: No matter which quorum gets accessed, exactly s nodes have to work. Therefore the sum of all loads should sum up to s (sum of access probabilities must sum to s if always s get accessed). To minimize the maximum element of a sum, set all elements to the average (balanced access strategy).

Last exercise

1.3 Uniform Quorum Systems

Definitions:

s-Uniform: A quorum system $\mathcal{S}$ is *s-uniform* if every quorum in $\mathcal{S}$ has exactly s elements.
Balanced access strategy: An access strategy Z for a quorum system $\mathcal{S}$ is *balanced* if it satisfies $L_Z(v_i) = L$ for all $v_i \in V$ for some value L.

Claim: An s-uniform quorum system $\mathcal{S}$ reaches an optimal load with a balanced access strategy, if such a strategy exists.

a) Describe in your own words why this claim is true.

b) Prove the optimality of a balanced access strategy on an s-uniform quorum system.

Last exercise

b) Let $V = \{v_1, v_2, ..., v_n\}$ be the set of servers and $\mathcal{S} = \{Q_1, Q_2, ..., Q_m\}$ an s-uniform quorum system on V. Let Z be an access strategy, thus it holds that: $\sum_{Q \in \mathcal{S}} P_Z(Q) = 1$. Furthermore let $L_Z(v_i) = \sum_{Q \in \mathcal{S}; v_i \in Q} P_Z(Q)$ be the load of server v_i induced by Z.

Then it holds that:

$$\sum_{v_i \in V} L_Z(v_i) = \sum_{v_i \in V} \sum_{Q \in \mathcal{S}; v_i \in Q} P_Z(Q) = \sum_{Q \in \mathcal{S}} \sum_{v_i \in Q} P_Z(Q)$$

$$= \sum_{Q \in \mathcal{S}} P_Z(Q) \sum_{v_i \in Q} 1 \overset{*}{=} \sum_{Q \in \mathcal{S}} P_Z(Q) \cdot s = s \cdot \sum_{Q \in \mathcal{S}} P_Z(Q) = s$$

The transformation marked with an asterisk uses the uniformity of the quorum system.

To minimize the maximal load on any server, the optimal strategy is to evenly distribute this load on all servers. Thus if a balanced access strategy exists, this leads to a system load of s/n.

Last exercise

2.2 Double Spending

Figure 1 represents the topology of a small Bitcoin network. Further assume that the two transactions T and T' of a doublespend are released simultaneously at the two nodes in the network and that forwarding is synchronous, i.e., after t rounds a transaction was forwarded t hops.

a) Once the transactions have fully propagated, which nodes know about which transactions?

b) Assuming that all nodes have the same computational power, i.e., same chances of finding a block, what is the probability that T will be confirmed?

$\frac{7}{12}$

c) Assuming the rightmost node, which sees T' first, has 20% of the computational power and all nodes have equal parts of the remaining 80%, what is the probability that T' will be confirmed?

$20+4*\frac{80}{11} = 49\%$

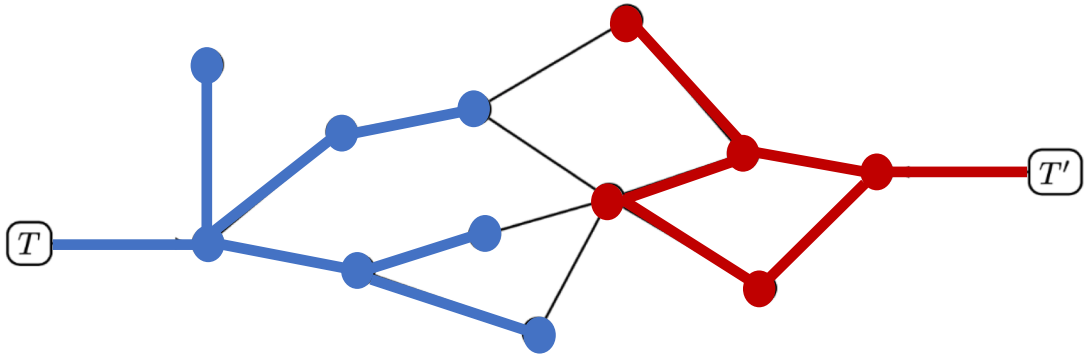


Figure 1: Random Bitcoin network

Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

(a) Address A mines 50 BTC.

(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.

Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

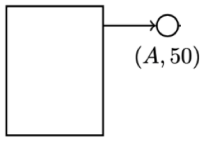
(a) Address A mines 50 BTC.

(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

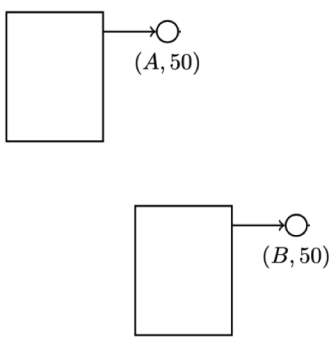
(a) Address A mines 50 BTC.

(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

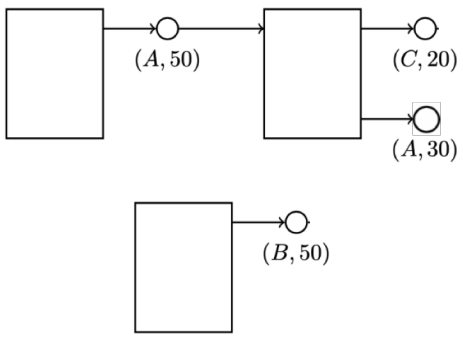
(a) Address A mines 50 BTC.

(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

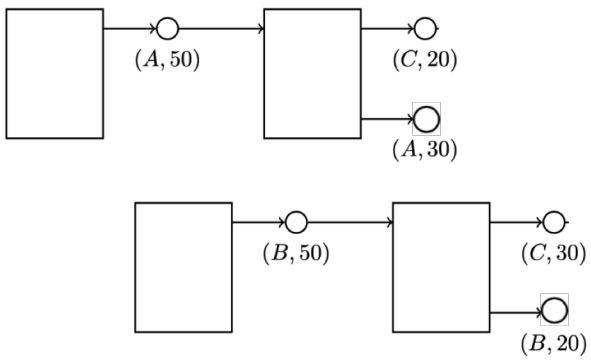
(a) Address A mines 50 BTC.

(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

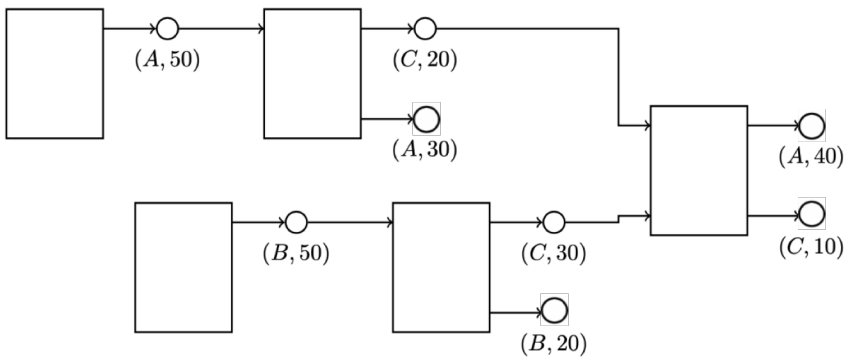
(a) Address A mines 50 BTC.

(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

(a) Address A mines 50 BTC.

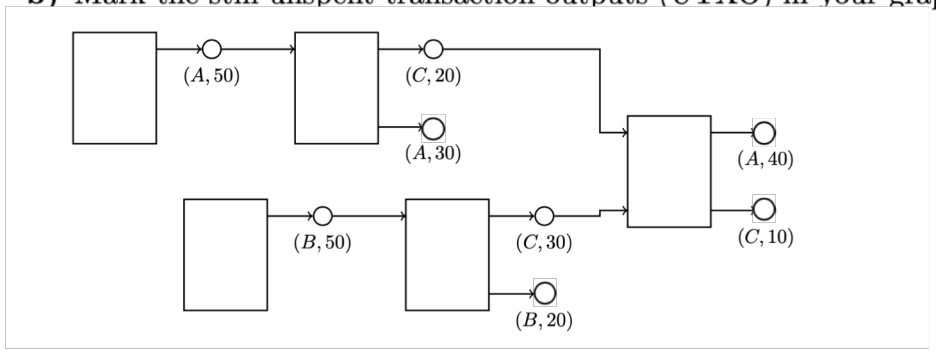
(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.

b) Mark the still unspent transaction outputs (UTXO) in your graph.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

(a) Address A mines 50 BTC.

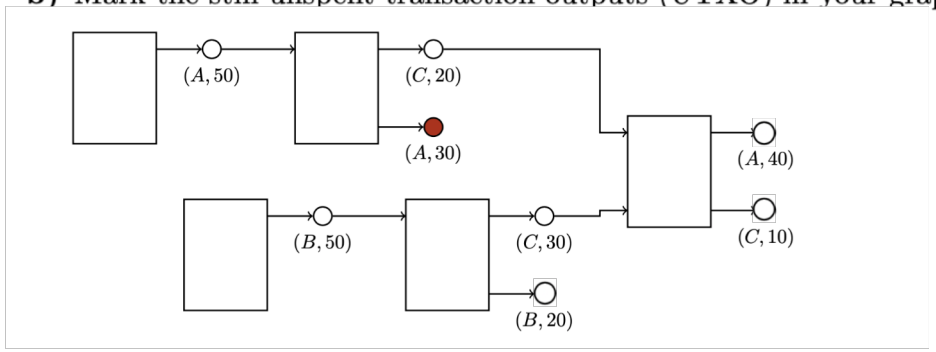
(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.

b) Mark the still unspent transaction outputs (UTXO) in your graph.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

(a) Address A mines 50 BTC.

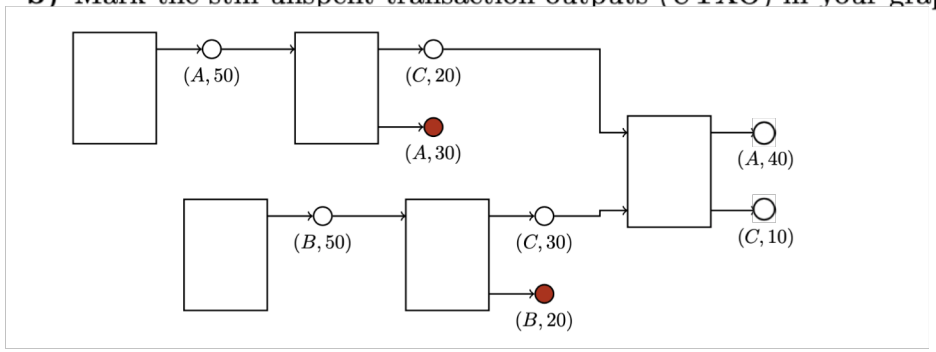
(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.

b) Mark the still unspent transaction outputs (UTXO) in your graph.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

(a) Address A mines 50 BTC.

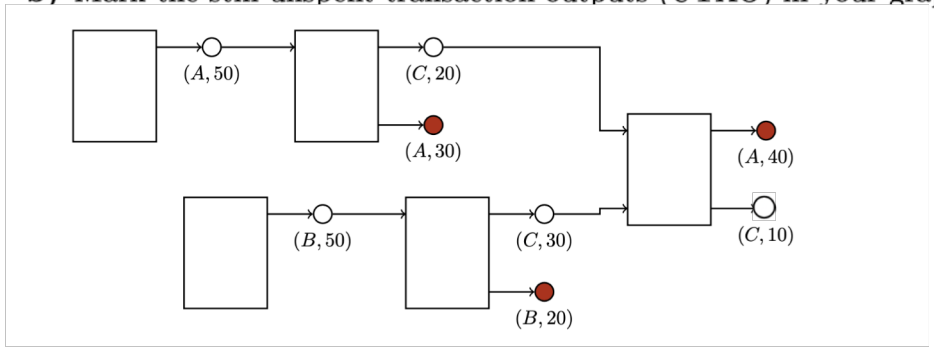
(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.

b) Mark the still unspent transaction outputs (UTXO) in your graph.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

(a) Address A mines 50 BTC.

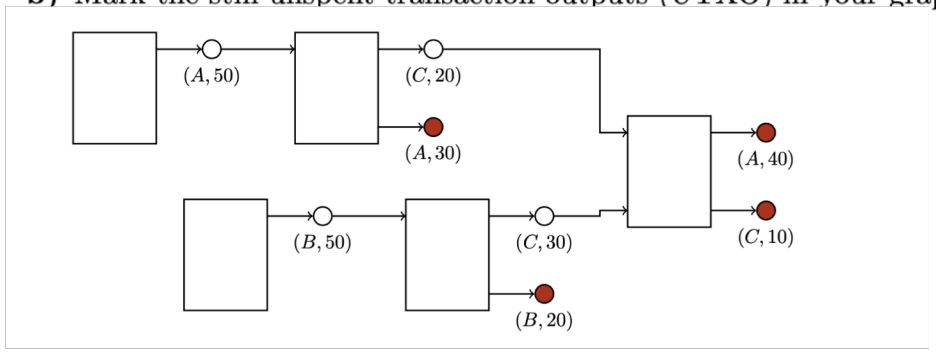
(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.

b) Mark the still unspent transaction outputs (UTXO) in your graph.



Last exercise

In Bitcoin existing money is stored as 'outputs'. An output is essentially a tuple (address, value). A transaction has a list of inputs, which reference existing outputs to destroy and a list of new outputs to create.

Because of this construction inputs claim the entire value associated with an output, even if the intended transfer is for a much smaller value than what the input references. If the input claims a larger value than needed for the transfer the user simply adds a *change output*, which returns the excess bitcoins to an address owned by the sender.

a) Draw the transaction graph created by the following transactions. Assume no fees are paid to the miners. Draw transactions as rectangles and outputs as circles. Arrows should point from outputs to the transactions spending them and from transactions to the outputs they are creating.

(a) Address A mines 50 BTC.

(b) Address B mines 50 BTC.

(c) A sends 20 BTC to C.

(d) B sends 30 BTC to C.

(e) C sends 40 BTC to A.

b) Mark the still unspent transaction outputs (UTXO) in your graph.

c) Why do inputs always spend the entire output value and not just the part that is needed for the transfer? Assume you can spend parts of an output and explain what would be needed to validate transactions and prevent the illegal generation of money.

For simplicity reasons, would need to handle a partially spent output.
For more details, refer to the master solution (interesting note on becoming rich)

Last exercise

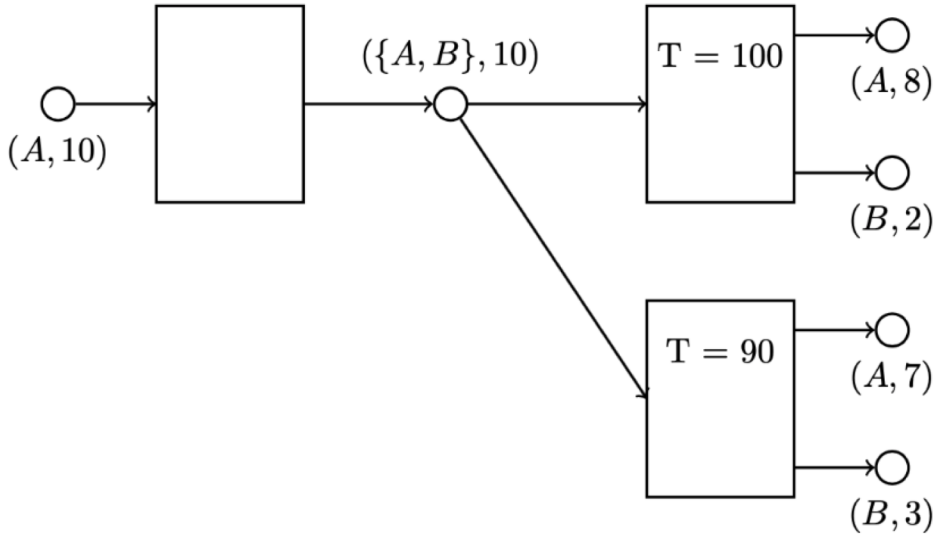


Figure 2: A "payment channel". A and B both have to sign to spend the output in the middle. The upper transaction can only be committed starting from blockheight 100, the lower one starting from blockheight 90.

Here first the left transaction is executed, this is called "opening" the channel. Then the transaction on the top right is prepared, but not executed. It can then be replaced by using lower timelocks.

a) What advantages does a payment channel have over regular Bitcoin transactions?

- Transactions are instantly finalized, so large confirmation delay of blockchain is irrelevant.
- Parties do not have to pay transaction fee for every transaction.

Last exercise

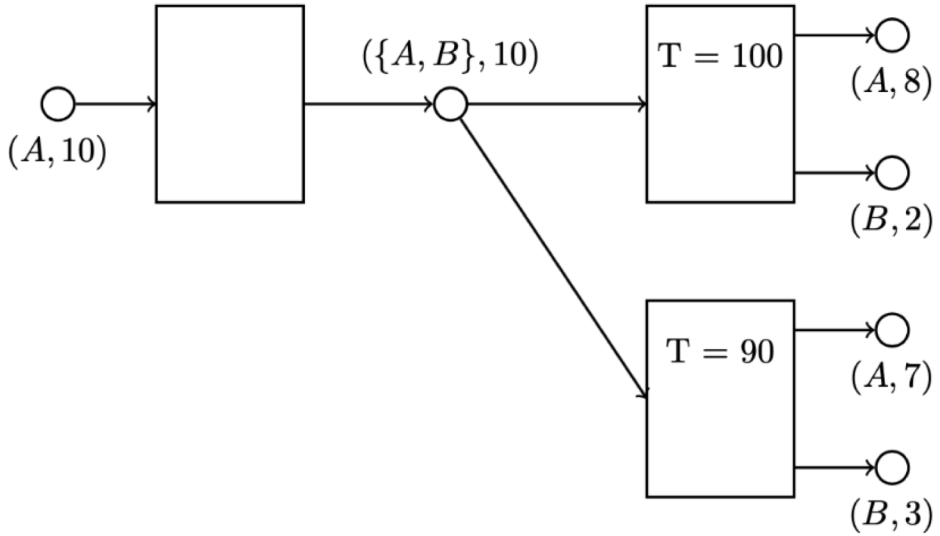


Figure 2: A "payment channel". A and B both have to sign to spend the output in the middle. The upper transaction can only be committed starting from blockheight 100, the lower one starting from blockheight 90.

Here first the left transaction is executed, this is called "opening" the channel. Then the transaction on the top right is prepared, but not executed. It can then be replaced by using lower timelocks.

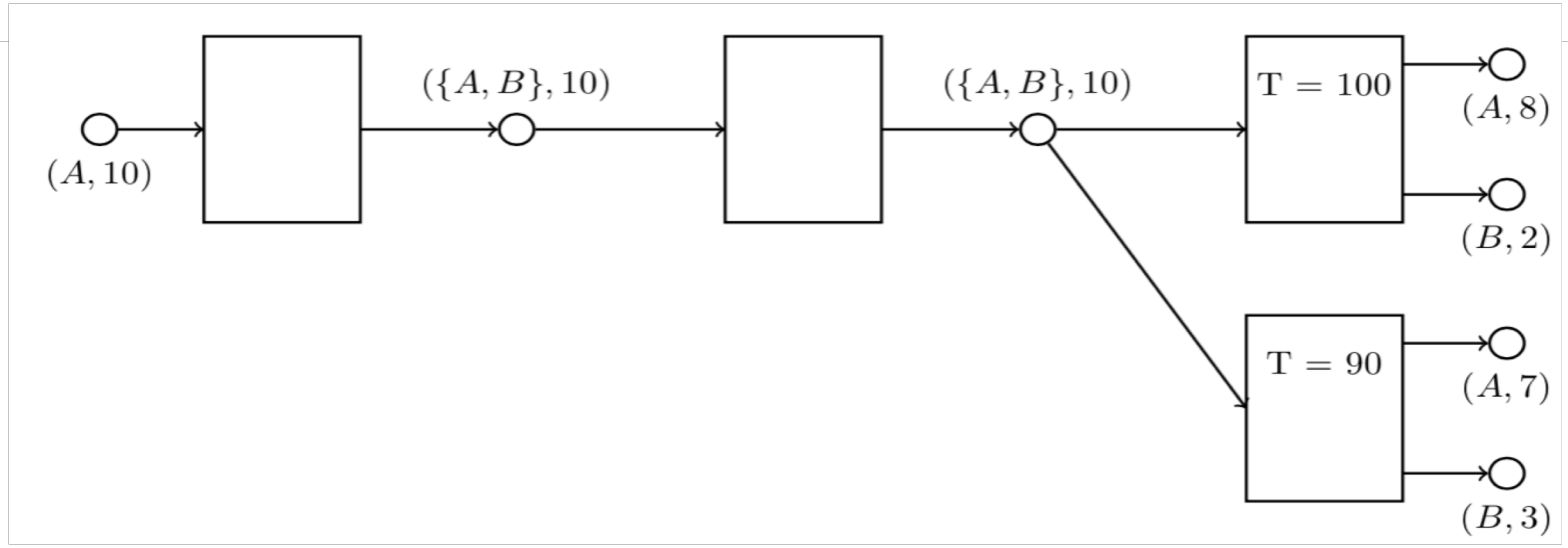
b) Why is the opening transaction needed? What could A do if the output being spent by the timelocked transactions would not require B's signature?

Without needing B's signature, A could just transfer the whole shared output to itself before the timelock expires

Last exercise

c) The channel cannot be used longer than the timeouts of the locktimes are. As soon as the first lock times out, the transaction needs to be executed, otherwise older replaced versions might become active as well. If someone wants to create a channel that he only uses occasionally, he needs to set the initial timelock far into the future.

Bitcoin also allows to define timelocks relative to the time the spent outputs were created. Can you think of a system that uses these relative timelocks to create channels that can be held open forever?

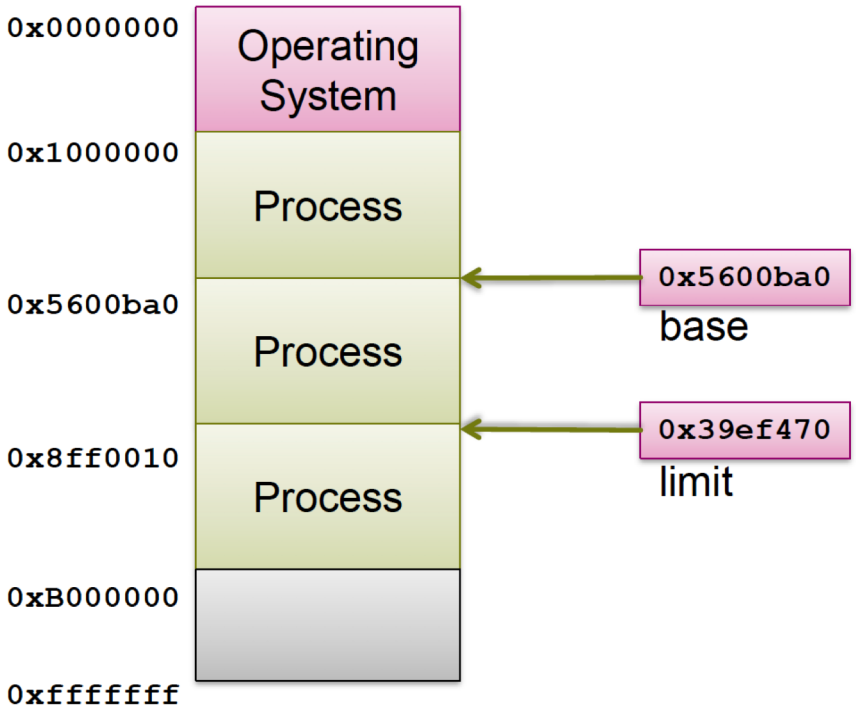


Memory management and virtual memory

- Goal: Allocate memory for every process. Memory allocation should be efficient and memory should be fast to access.

Partitioned Memory

A pair of base and limit registers define the logical address space



Issue:

- Addresses are not known until load time.

Solution:

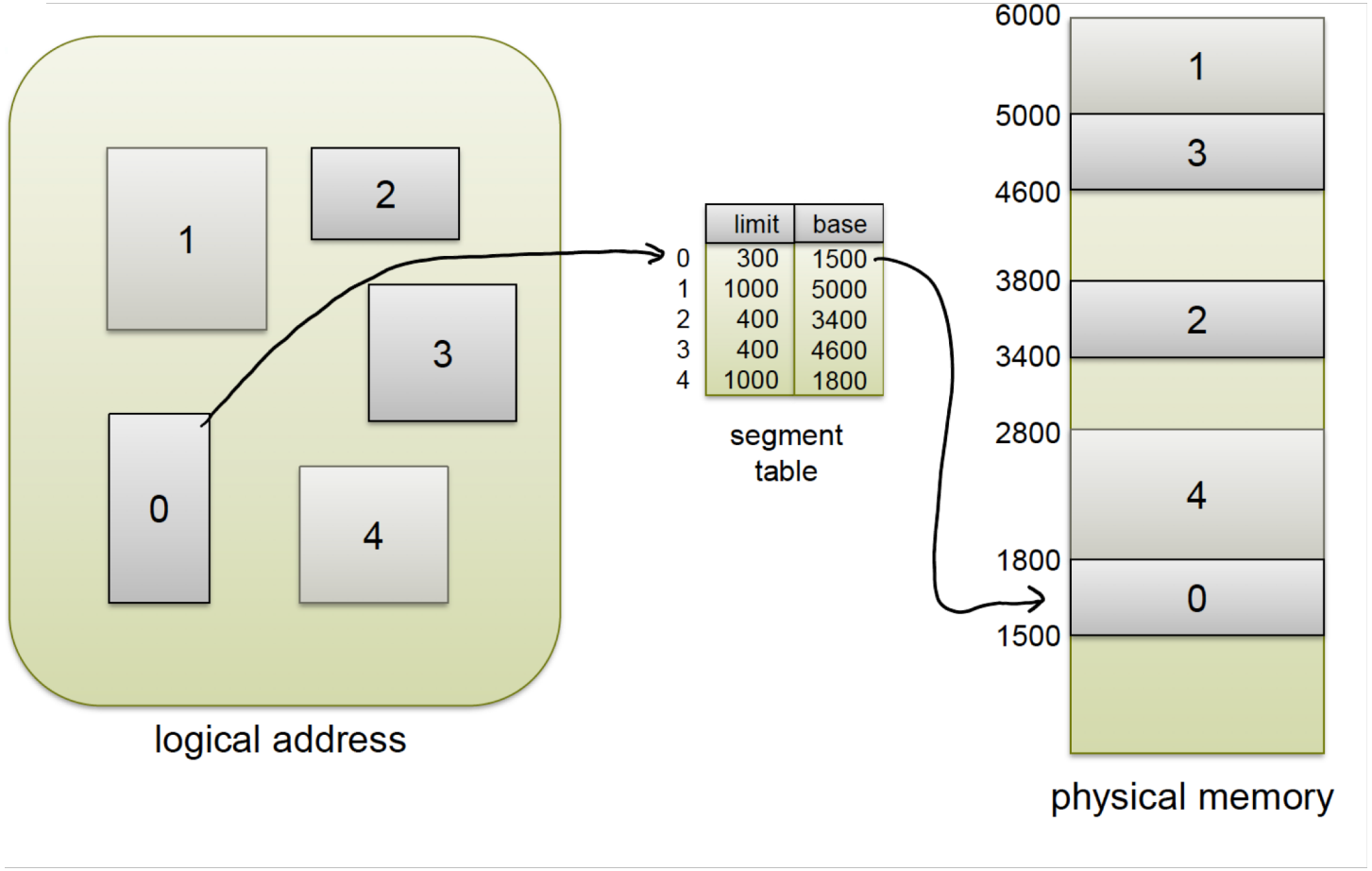
- Code is completely position-independent
- Relocation register maps compiled addresses to physical addresses

Issue:

- Only continuous memory regions per process
  - external fragmentation
  - no memory sharing between processes

Segmentation

- Idea: not only one address range per process, but multiple segments.



Segmentation Architecture

- Segment table – each entry has:
  - base
  - limit
- Segment-table base register
  - current segment table location in memory
- Segment-table length register
  - current size of segment table
- Summary:
  - Segments can be shared
  - Fast context switches (only reload segment table registers)
  - external fragmentation still a problem

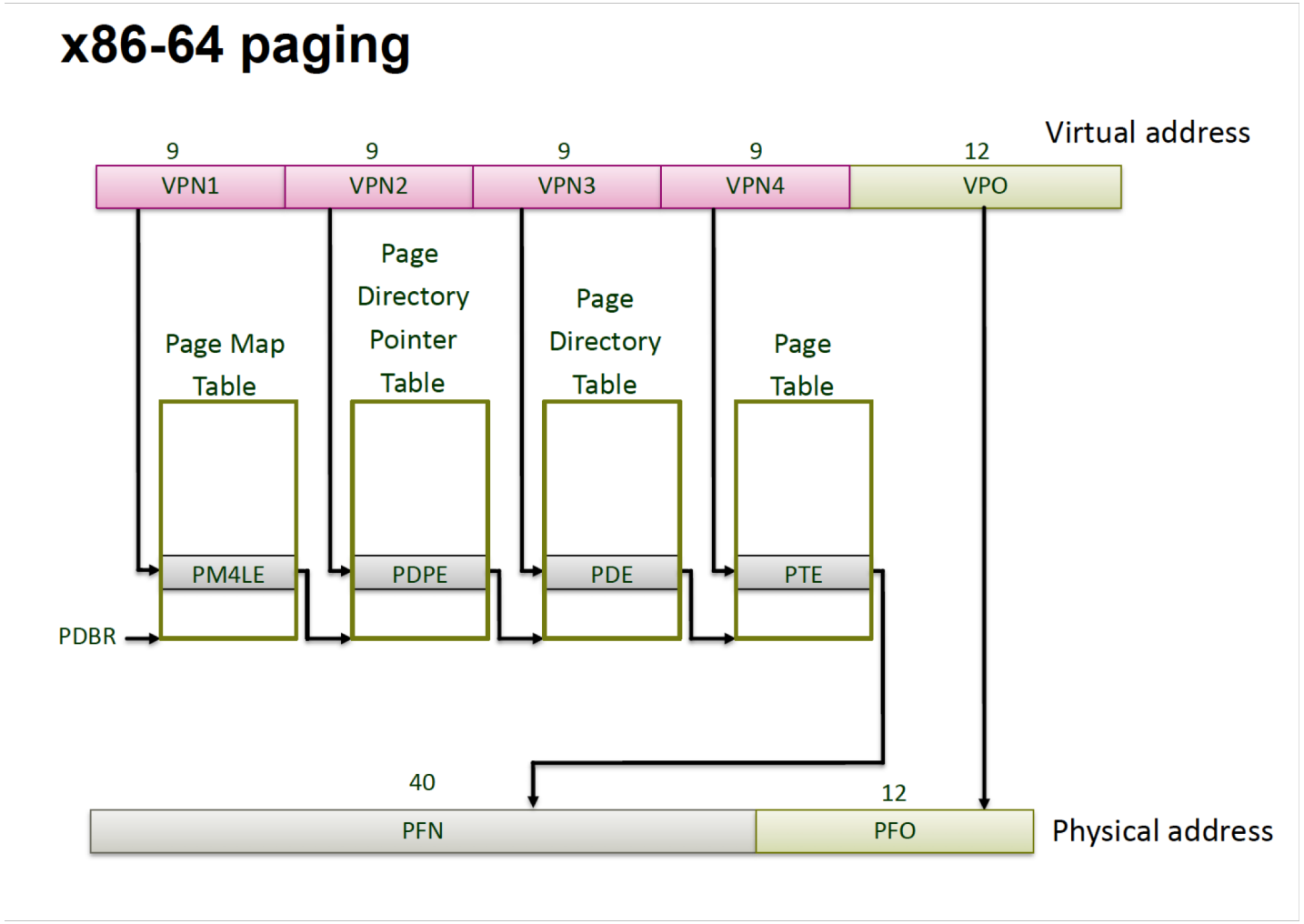
Paging

- Divides physical memory into fixed size frames
- Divides virtual memory into pages of same size
- uses page table to do mapping from virtual to physical memory
- Solves external fragmentation problem – process can always fit if there is available free memory

Different Page Tables

- Linear page table
  - array of physical page numbers indexed by virtual pages numbers
  - grows in size with memory
- Hierarchical page table
  - organized in multiple layers, each translating different part of virtual address

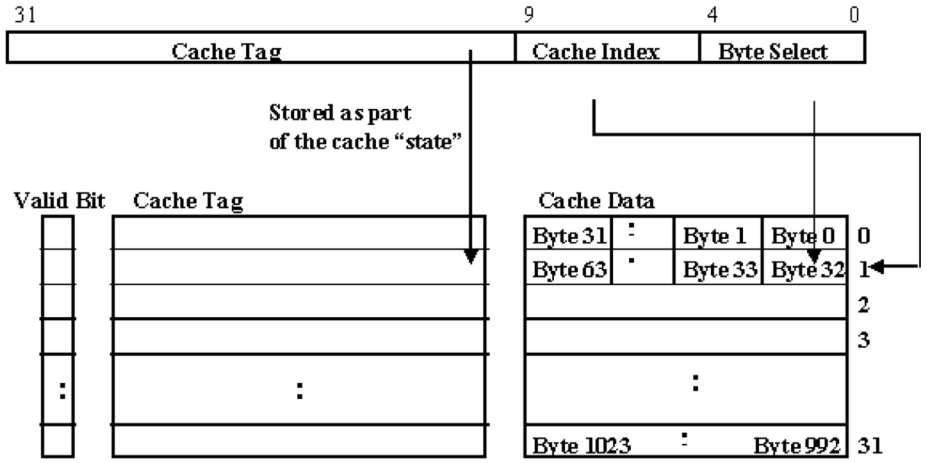
Example: x86


x86-64 paging
Virtual address
9
9
9
9
12
VPN1
VPN2
VPN3
VPN4
VPO
Page Map Table
Page Directory Pointer Table
Page Directory Table
Page Table
PM4LE
PDPE
PDE
PTE
PDBR
40
12
PFN
PFO
Physical address

Copy on write (COW)

- Have shared memory until one process writes to memory, then copy it
  - How:
  1. forking process, change rights of both processes to READONLY
  2. if one process tries write to memory, copy page and add it to childs page table as WRITEABLE
  3. mark page in parent's page table as WRITEABLE
- makes fork much faster, since you don't need to copy whole address space

Cache types



- virtually indexed, virtually tagged
  - suffers from homonyms, because same virtual address can mean different physical address in different processes
    - address-space tags can prevent that by adding to a cache line which address space it belongs to
  - fast
- physically indexed, physically tagged
  - good for context switches
  - slow, since address needs to be translated before lookup
- virtually indexed, physically tagged
  - good if it works
  - complicated
- physically indexed, virtually tagged
  - not really useful

Demand paging

- Idea: only page in page into main memory when needed.

Algorithm 18.2 Demand paging: page fault handling

On a page fault with faulting VPN v_{fault}:

```
1:  if there are free physical pages then
2:      p ← get_new_pfn()
3:  else
4:      p ← get_victim_pfn()
5:      v_old ← VPN mapped to p
6:      invalidate all TLB entries and page table mappings to p
7:      if p is dirty (modified) then
8:          write contents of p into v_old's area in storage
9:      end if
10: end if
11: read page v_fault in from disk into physical page p
12: install mapping from v_fault to p
13: return
```

Page replacement strategies

- How does `4: p ← get_victim_pfn()` actually work?
- optimal page replacement: take page that won't be used for the longest
  - gives optimal performance
  - not feasible in practice, since we do not know page accesses in advance
- FIFO page replacement
  - easy to implement
  - does not give good performance
  - Beladys anomaly

Page replacement strategies

- Least recently used
  - good performance
  - hard to implement, OS usually gets informed about page faults, but not every memory access
- 2nd chance replacement – approximation to least recently used
  - Idea: every time a page gets referenced, a “referenced” bit gets set to true (does not tell when exactly it got referenced, just that it got referenced recently). When looking for a victim page, iterate through all pages and set all true referenced bits to false. Take the first one that has “referenced = false”
  - approximation to LRU
  - hardware sometimes provides referenced bit
  - we can have better approximation

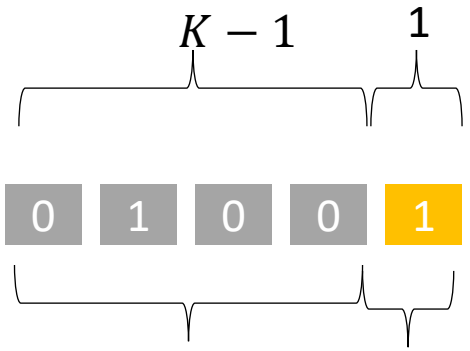
Allocating Pages between processes

- Global page allocation: OS selects page replacement from set of all pages
  - each process needs minimum number of pages
- Local page allocation: OS selects page replacement from current faulting process
  - works
  - Question here: how many pages get allocated per process?

Working set

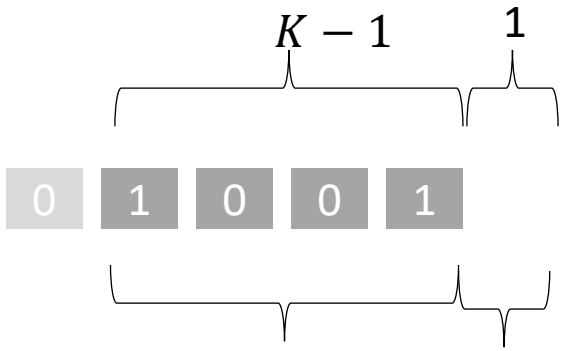
- Working set: set of pages accessed by a process during a fixed time window
- Working set size: size of working set ☺
- Thrashing: Working set size a lot bigger than size of physical memory

Working set estimation



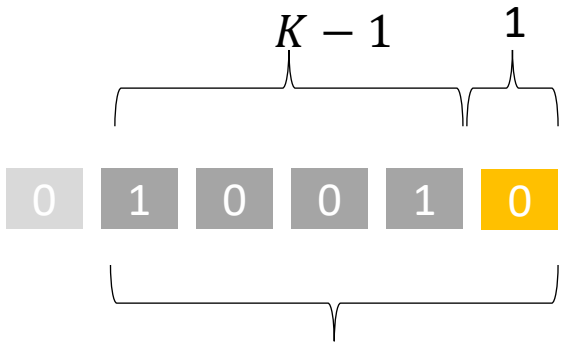
represents K*σ timesteps σ

Working set estimation



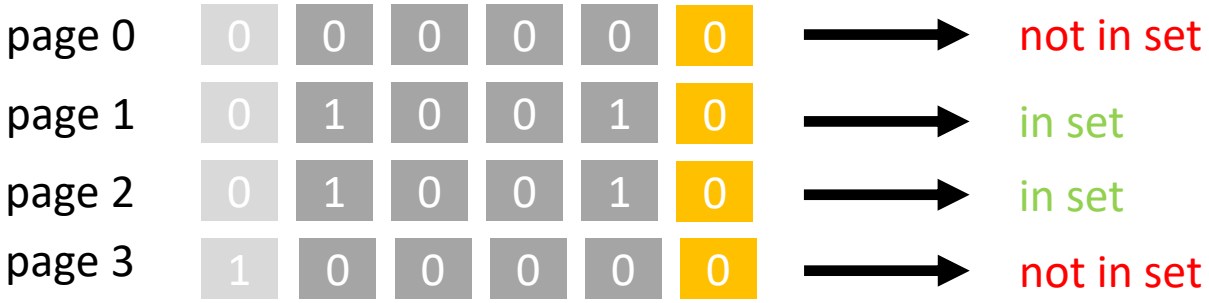
represents K*σ timesteps σ

Working set estimation



or over all those bits, if at least one is 1, then add to working set

Working set estimation



Working set = {page 1, page2}

Quiz

- Why doesn’t Beladys anomaly happen with stack-based replacement strategies?
  - Because with stack-based implementations, a set size of k is always a subset of a set size of k+n. So a page fault in k+n always happens in k too.
- What is a criteria for a good page replacement strategy?
  - The number of page faults should be as small as possible
- Is there a scenario when COW is slower then just copying the whole process?
  - Yes, if a large amount of pages gets written to. COW has a certain overhead to just copying everything, because of marking everything as readable/writable and the page faults it causes.

Next exercise