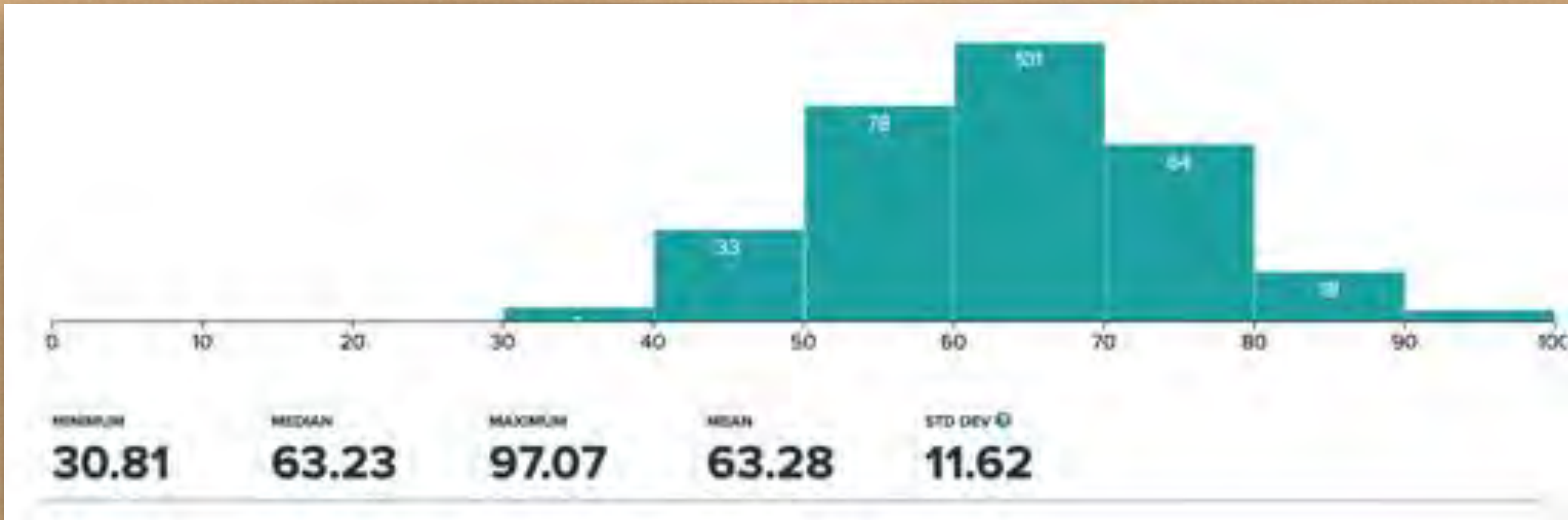
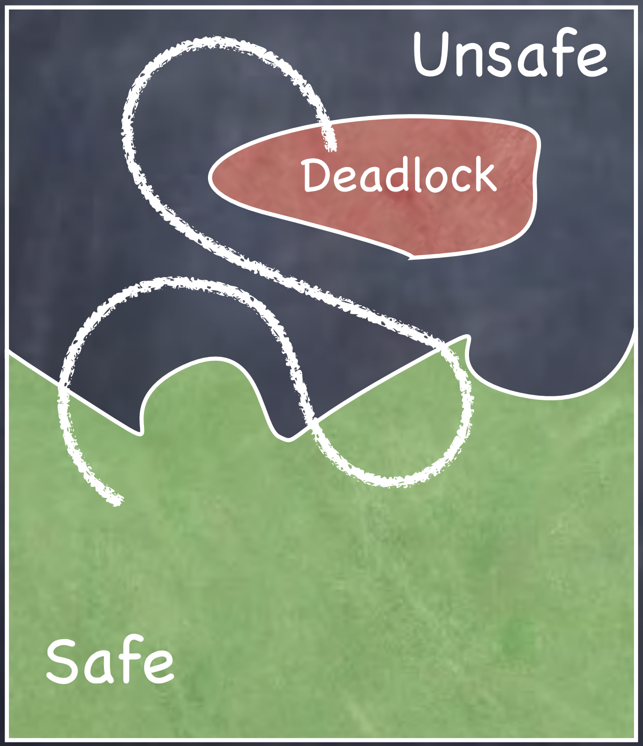


Prelim 1: Final results

Living dangerously: Safe, Unsafe, Deadlocked



A system's trajectory through its state space

- Safe: For any possible set of resource requests, there exists one safe schedule of processing requests that succeeds in granting all pending and future requests
  - no deadlock as long as system can enforce that safe schedule!
- Unsafe: There exists a set of (pending and future) resource requests that leads to a deadlock, independent of the schedule in which requests are processed
  - unlucky set of requests can force deadlock
- Deadlocked: The system has at least one deadlock

The Banker's books

- Assume n processes, m resources
- Max_{ij} = max amount of units of resource R_j needed by P_i
  - $MaxClaim_i$: Vector of size m such that $MaxClaim_i[j] = Max_{ij}$
- $Holds_{ij}$ = current allocation of R_j held by P_i
  - $HasNow_i$ = Vector of size m such that $HasNow_i[j] = Holds_{ij}$
- Available = Vector of size m such that Available[j] = units of R_j available
- A request by P_k is safe if, assuming the request is granted, there is a permutation of $P_1, P_2, \ldots, P_n$ such that, for all P_i in the permutation

$$Needs_i = MaxClaim_i - HasNow_i \leq Avail + \sum_{j=1}^{i-1} HasNow_j$$

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Holds

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

- Is this a safe state?

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

–

Holds

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

Needs

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	4	2

- Is this a safe state?

P_1, P_4, P_2, P_3, P_5

  - While safe permutation does not include all processes:
    - Is there a P_i such that $Needs_i \leq Avail$?
      - if no, exit with unsafe
      - if yes, add P_i to the sequence and set Avail = Avail + $HasNow_i$
  - Exit with safe

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Holds

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

Needs

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	4	2

- P_2 wants to change its holdings to 0 4 2 0

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Holds

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	0	4	2	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
2	1	0	0

Needs

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	1	3	3	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	4	2

- P_2 wants to change its holdings to 0 4 2 0
- Safe? Reduce P_1

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Holds

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	4	2	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
2	1	1	2

Needs

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	1	3	3	0
P_3	1	0	0	3
P_4	0	0	2	0
P_5	0	6	4	2

- P_2 wants to change its holdings to 0 4 2 0
- Safe? Reduce P_1; can't reduce any further

Unsafe!

If all processes were to ask together all the resources they may need, deadlock!

Reactive Responses to Deadlock

- Deadlock Detection
  - Track resource allocation (who has what)
  - Track pending requests (who's waiting for what)
- When should it run?
  - For each request?
  - After each unsatisfiable request?
  - Every hour?
  - Once CPU utilization drops below a threshold?

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	2
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	2
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	0	0	0
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
3	0	3

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	2
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	0	0	0
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
3	0	3

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	2
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	0
P_3	0	0	0
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
3	1	3

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	2
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	0
P_3	0	0	0
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
3	1	3

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	2
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	0
P_3	0	0	0
P_4	0	0	0
P_5	0	0	2

Available

R_1	R_2	R_3
5	2	4

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	0	0	0
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	0
P_3	0	0	0
P_4	0	0	0
P_5	0	0	2

Available

R_1	R_2	R_3
5	2	4

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	0	0	0
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	0	0	0
P_3	0	0	0
P_4	0	0	0
P_5	0	0	2

Available

R_1	R_2	R_3
7	2	4

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	0	0	0
P_3	0	0	0
P_4	0	0	0
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	0	0	0
P_3	0	0	0
P_4	0	0	0
→ P_5	0	0	2

Available

R_1	R_2	R_3
7	2	4

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	0	0	0
P_3	0	0	0
P_4	0	0	0
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	0	0	0
P_3	0	0	0
P_4	0	0	0
P_5	0	0	0

Available

R_1	R_2	R_3
7	2	6

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	0	0	0
P_3	0	0	0
P_4	0	0	0
P_5	0	0	0

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Yes, there is a safe schedule!

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	0
P_4	1	0	2
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Yes, there is a safe schedule!

but it is not a safe state!

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	1
P_4	1	0	2
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	0
P_2	2	0	0
P_3	3	0	3
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
0	0	0

Pending

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	2
P_3	0	0	1
P_4	1	0	2
P_5	0	0	2

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- Without Max, can we avoid deadlock by delaying granting requests?
  - NO! Deadlock triggered when request formulated, not granted!

Deadlock Recovery

- Blue screen & reboot
- Kill one/all deadlocked processes
  - Pick a victim (how?); Terminate; Repeat as needed
    - Can leave system in inconsistent state
- Proceed without the resource (if application permits)
  - Example: timeout on inventory check at Amazon
- Use transactions
  - Rollback & Restart
  - Need to pick a victim...

Summary

- Prevent
  - Negate one of the four necessary conditions
- Avoid
  - Schedule processes carefully
- Detect
  - Has a deadlock occurred?
- Recover
  - Kill or Rollback

Barrier Synchronization

And now for something completely different

- Remember Mutual Exclusion…
  - at most one process executing at the same time a given piece of code
- Barrier Synchronization is almost the opposite…
  - A set of processes that runs in rounds
  - All must complete the current round before any can start the next
  - Popular in HPC, simulations, graph processing, model checking…

Barrier Abstraction

- Barrier($\mathcal{N}$): barrier for $\mathcal{N}$ threads
- bwait(?barr): wait for every thread to catch up on barrier barr

Test Program for Barriers

```
import barrier

const NTHREADS = 3
const NROUNDS = 4

round = [0,] * NTHREADS
invariant (max(round) – min(round)) <= 1

barr = barrier.Barrier(NTHREADS)

def thread(self):
    for r in {0..NROUNDS–1}:
        barrier.bwait(?barr)
        round[self] += 1

for i in {0..NTHREADS–1}:
    spawn thread(i)
```

Threads can be at most one round apart

Waiting for everyone to reach the barrier

Barrier Implementation

```
from synch import *

def Barrier(required):
    result = {
        .mutex: Lock(), .cond: Condition(),
        .required: required, .left: required, .cycle: 0
    }

def bwait(b):
    acquire(?b→mutex)
    b→left −= 1
    if b→left == 0:
        b→cycle = (b→cycle + 1) % 2
        b→left = b→required
        notifyAll(?b→cond)
    else:
        let cycle = b→cycle:
            while b→cycle == cycle:
                wait(?b→cond, ?b→mutex)
    release(?b→mutex)
```

lock and condition (as you would expect)

no. of threads

threads that have not reached the barrier

incremented at each round to allow for barrier reuse

If repeat use were not an issue, a lock, a condition variable, and a counter initialized to the number of threads would suffice, but, to reuse the barrier, we must reset the counter!

Using Barriers

```
import barrier

const NTHREADS = 3
const NROUNDS = 4

round = [0,] * NTHREADS
invariant (max(round) – min(round)) <= 1

phase = 0
barr = barrier.Barrier(NTHREADS)

def thread(self):
    for r in {0..NROUNDS–1}:
        if self == 0: # coordinator prepares
            phase += 1
        barrier.bwait(?barr) # enter parallel work
        round[self] += 1
        assert round[self] == phase
        barrier.bwait(?barr) # exit parallel work

for i in {0..NTHREADS–1}:
    spawn thread(i)
```

everybody waits for the coordinator

everybody synchronizes at the end of round

Memory Management
(Ch. 12–17)

Abstraction is our Business

- What I have
  - A single (or a finite number) of CPUs
  - Many programs I would like to run
- What I want : a Thread
  - Each program has full control of one or more CPUs

Abstraction is our Business

- What I have
  - A certain amount of physical memory
  - Multiple programs I would like to run
    - together, they may need more than the available physical memory
- What I want : an Address Space
  - Each program has as much memory as the machine's architecture will allow to name
  - All for itself

Address Space

- Set of all names used to identify and manipulate unique instances of a given resource
  - memory locations (determined by the size of the machine's word)
    - for 32-bit-register machine, the address space goes from 0x00000000 to 0xFFFFFFFF
  - memory locations (determined by the number of memory banks mounted on the machine)
  - phone numbers (XXX) (YYY-YYYY)
  - colors: R (8 bits) + G (8 bits) + B (8 bits)

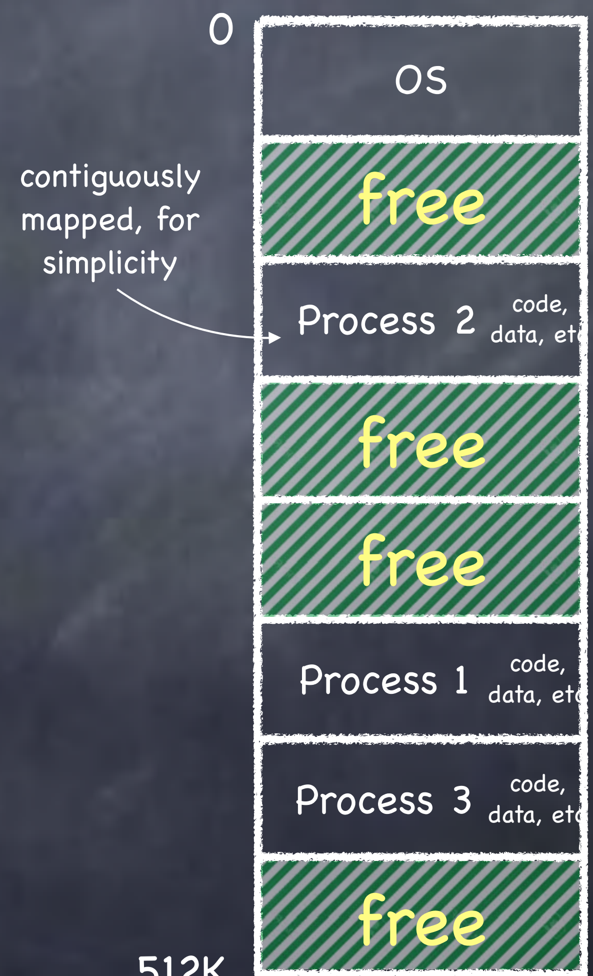
Virtual Address Space: An Abstraction for Memory

- Virtual addresses start at 0
- Heap and stack can be placed far away from each other, so they can nicely grow
- Addresses are all contiguous
- Size is independent of physical memory on the machine

Not at scale!

Physical Address Space: How memory actually looks

- Processes loaded in memory at some memory location
  - virtual address 0 is not loaded at physical address 0
- Multiple processes may be loaded in memory at the same time, and yet...
- ...physical memory may be too small to hold even a single virtual address space in its entirety
  - 64-bit, anyone?

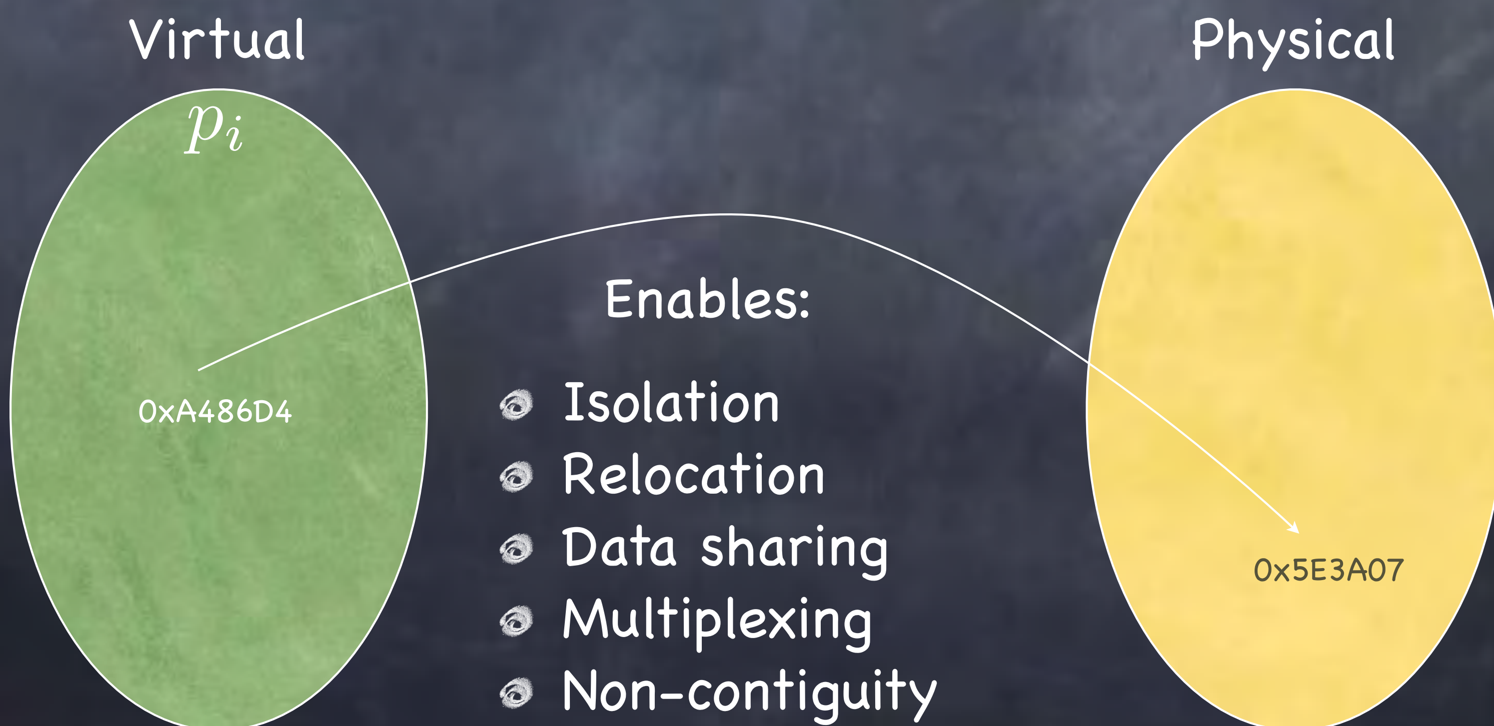


II. Memory Isolation

Step 2: Address Translation

- Implement a function mapping

$\langle pid, virtual\ address \rangle$ into $physical\ address$



Enables:

- Isolation
- Relocation
- Data sharing
- Multiplexing
- Non-contiguity

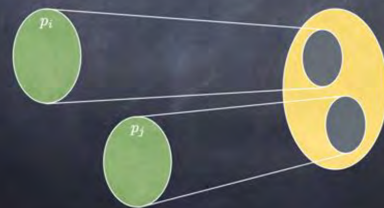
II. Memory Isolation

Step 2: Address Translation

- Implement a function mapping $\langle pid, virtual\ address \rangle$ into $physical\ address$

Isolation

- At all times, functions used by different processes map to disjoint ranges — aka "Stay in your room!"



Relocation

- The range of the function used by a process can change over time

Relocation

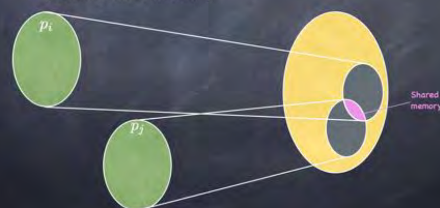
- The range of the function used by a process can change over time — "Move to a new room!"

Data Sharing

- Map different virtual addresses of distinct processes to the same physical address — ("Share the kitchen")

Data Sharing

- Map different virtual addresses of distinct processes to the same physical address — ("Share the kitchen")

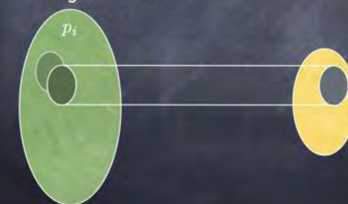


Multiplexing

- Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses — ever lived in a studio?

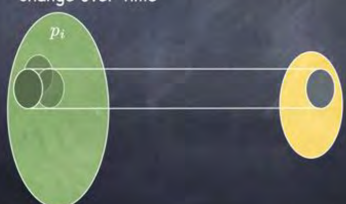
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



Multiplexing

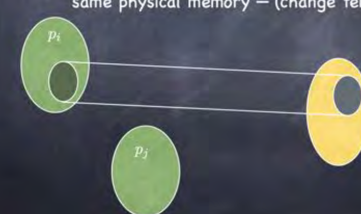
- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

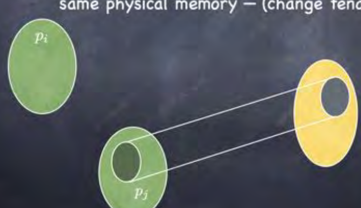
More Multiplexing

- At different times, different processes can map part of their virtual address space into the same physical memory — (change tenants)



More Multiplexing

- At different times, different processes can map part of their virtual address space into the same physical memory — (change tenants)



(Non) Contiguity

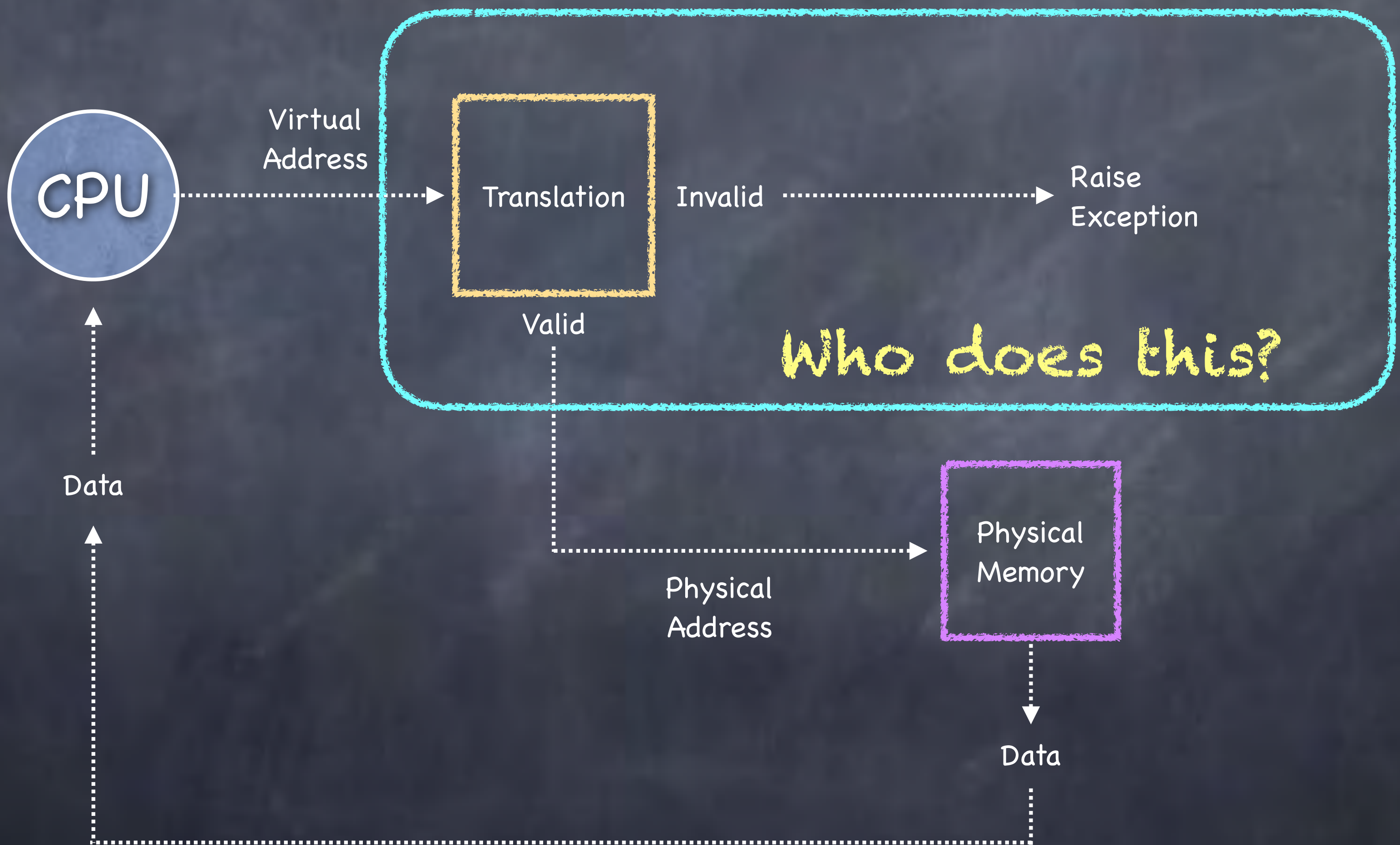
- Contiguous virtual addresses can be mapped to non-contiguous physical addresses…

(Non) Contiguity

- …and non-contiguous virtual addresses can be mapped to contiguous physical addresses

The Power of Mapping

Address Translation, Conceptually



Memory Management Unit (MMU)

- Hardware device
  - Maps virtual addresses to physical addresses

- User process
  - deals with virtual addresses
  - never sees the physical address
- Physical memory
  - deals with physical addresses
  - never sees the virtual address