EECS 482
Introduction to Operating Systems

Winter 2018

Baris Kasikci

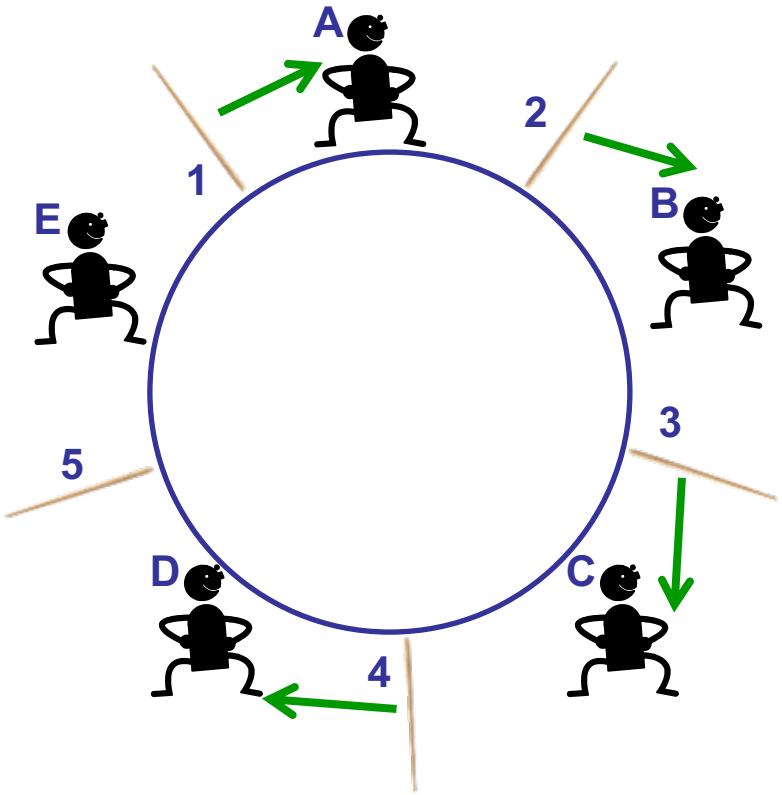
Slides by: Harsha V. Madhyastha

Four necessary conditions for deadlock

- Limited resources
  - Not enough to serve all threads simultaneously
- No preemption
  - Can't force threads to give up resources
- Hold and wait
  - Threads hold resources while waiting to acquire other resources
- Cyclical chain of requests

Eliminating circular chain

- Impose global ordering of resources



Eliminating hold-and-wait

- Two ways to avoid hold and wait:
  - Wait for all resources needed to be free; grab them all atomically
  - If cannot get a resource, release all and start over
- Move resource acquisition to beginning

```
Phase 1: acquire all resources
Phase 2: while (!done) {
             work
         }
Phase 3: release all resources
```

- Ensures working threads will complete

Banker's algorithm

- An alternative solution to eliminate hold-and-wait
  - Allows for more concurrency
- Declare resources at the beginning, but don't actually acquire them

```
Phase 1: declare all resources
Phase 2: while (!done) {
             acquire resource if safe
             work
         }
Phase 3: release all resources
```

Banker's algorithm

```
Phase 1: declare all resources
Phase 2: while (!done) {
             acquire resource if safe
             work
         }
Phase 3: release all resources
```

- Only grant resource if it's "safe", otherwise block
  - Safe means I can guarantee that all threads can finish

- Criterion: Can I grant the maximum resources of all threads in some sequential order?

The bank example

- A bank has $6000
- Customers establish credit limit, and then can borrow money (up to their credit limit)
  - Credit limit is “max resource usage”
- When their business is done, customers return the money

Bank solution #1

- *Bank gives money upon request, if it's available*

- Example:
  - Ann asks for credit of $2000
  - Bob asks for credit of $4000
  - Charlie asks for credit of $6000
- Can the bank approve all these credit lines?
- No, a deadlock can form:
  - Ann borrows $1000
  - Bob borrows $2000
  - Charlie borrows $3000

Bank solution #1

- *Bank gives money upon request, if it's available*

- Only works if the bank reserves the money when credit line is established. Customers may have to wait at the credit approval stage.
- Example:
  - Ann asks for credit of $2000. Approved.
  - Bob asks for credit of $4000. Approved.
  - Charlie asks for credit of $6000. Must wait until Ann and Bob drop their lines of credit.

Banker's algorithm

- *Bank approves **all** credit requests. Bank gives money upon request but **only if it's safe**.*
- Example:
  - Ann borrows $1000 (bank has $5000 left)
  - Bob borrows $2000 (bank has $3000 left)
  - Charlie wants to take out $2000. Is this allowed?
- It is allowed **iff** there is some sequential ordering of fulfilling everyone's max resources
  - Charlie leaves bank with $1000.
  - Ann can finish, leaving $2000
  - So Bob can finish, leaving $4000
  - So Charlie can finish.

Ann:	**$2000**
Bob:	**$4000**
Charlie:	**$6000**

Banker's algorithm

- *Bank approves **all** credit requests. Bank gives money upon request but **only if it's safe**.*
- Example #2:
  - Ann borrows $1000 (bank has $5000 left)
  - Bob borrows $2000 (bank has $3000 left)
  - Charlie wants to take out $2500. Is this allowed?
- It is allowed **iff** there is some sequential ordering of fulfilling everyone's max resources
- $500 left in bank. Can't guarantee that any single customer would finish.

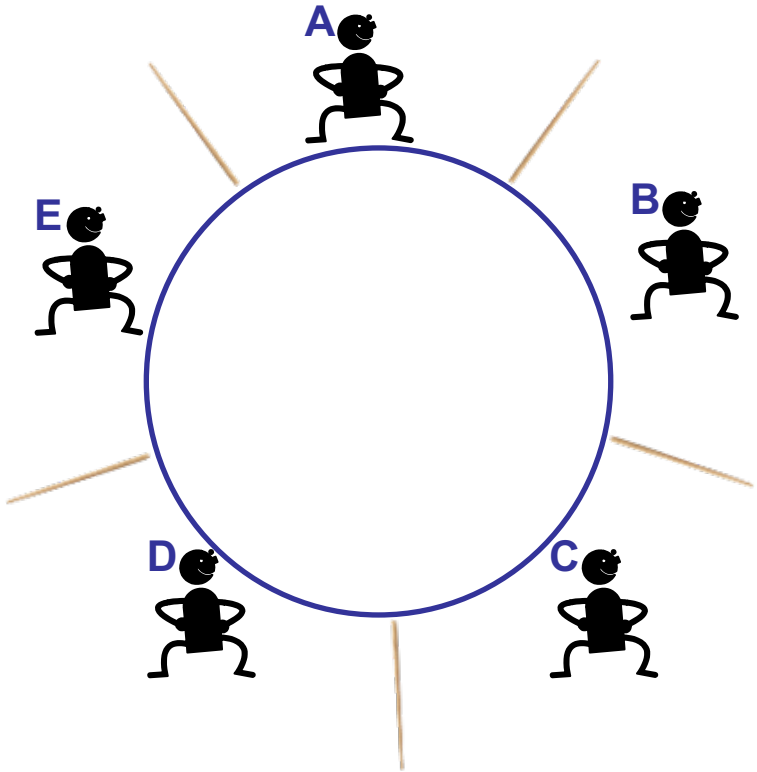
Ann:	**$2000**
Bob:	**$4000**
Charlie:	**$6000**

Banker's algorithm

- *Bank approves **all** credit requests. Bank gives money upon request but **only if it's safe**.*
- Example #3:
  - Ann borrows $1000 (bank has $5000 left)
  - Bob borrows $2000 (bank has $3000 left)
  - Charlie wants to take out $2000. Is this allowed?
- $1000 left in bank.
  - Ann can finish, leaving $2000 in bank
  - Bob might need $3000 more
  - Charlie might need $4000 more
  - Can't guarantee that either can finish

Ann:	$2000
Bob:	$5000
Charlie:	$6000

Banker's algorithm for the dining philosophers



- Max resource need for each philosopher is 2
- Grant chopstick unless it's the last chopstick and nobody has 2

Banker's algorithm

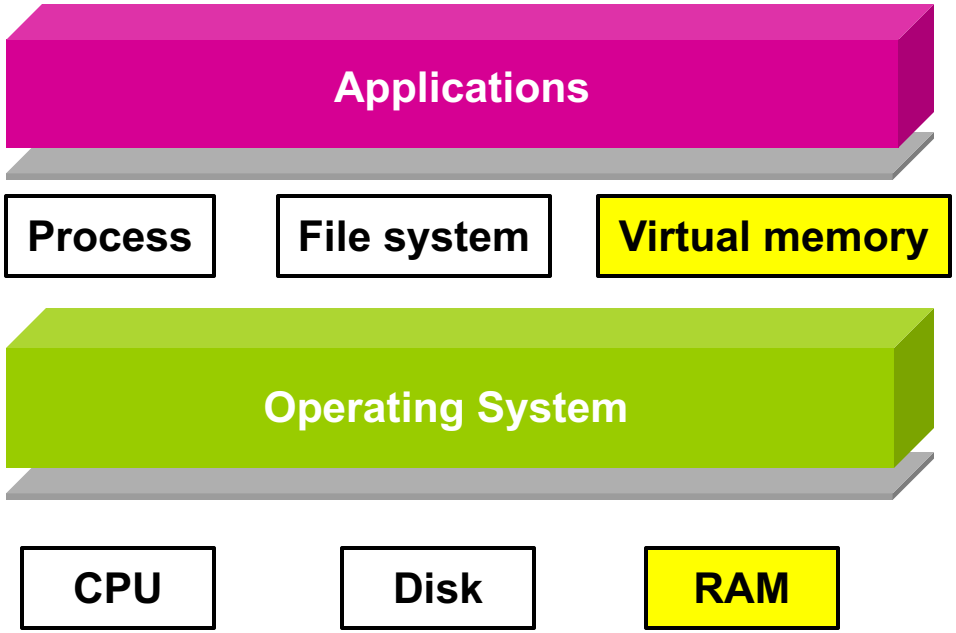
- Allows system to overcommit resources
  - Sum of max resources can be greater than total resources

- Elegant algorithm, when applicable
  - Sometimes it's hard to know what the max resource need is
  - Only applies to "quantitative" resources
    - » Not applicable to locks

Project 2

- Test, test, test!
- Test your code **while** writing it
- Write your test cases **before** the code
- Go through the spec, write a test case for **every** condition required by the spec

OS Abstractions



Memory management

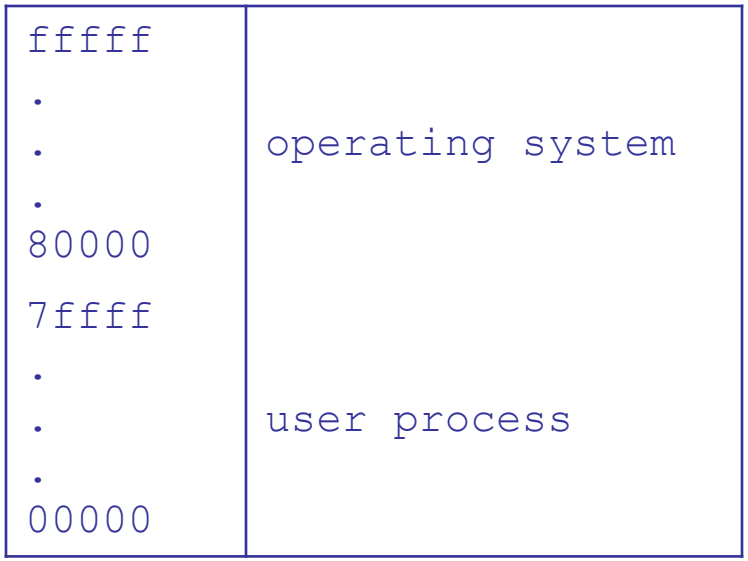
- Recall: Process = Set of threads + address space
- Address space
  - All the memory space the process can use as it runs
- Hardware interface: physical memory shared between processes
- Why not use physical memory addresses directly?

Address space abstraction provided by OS

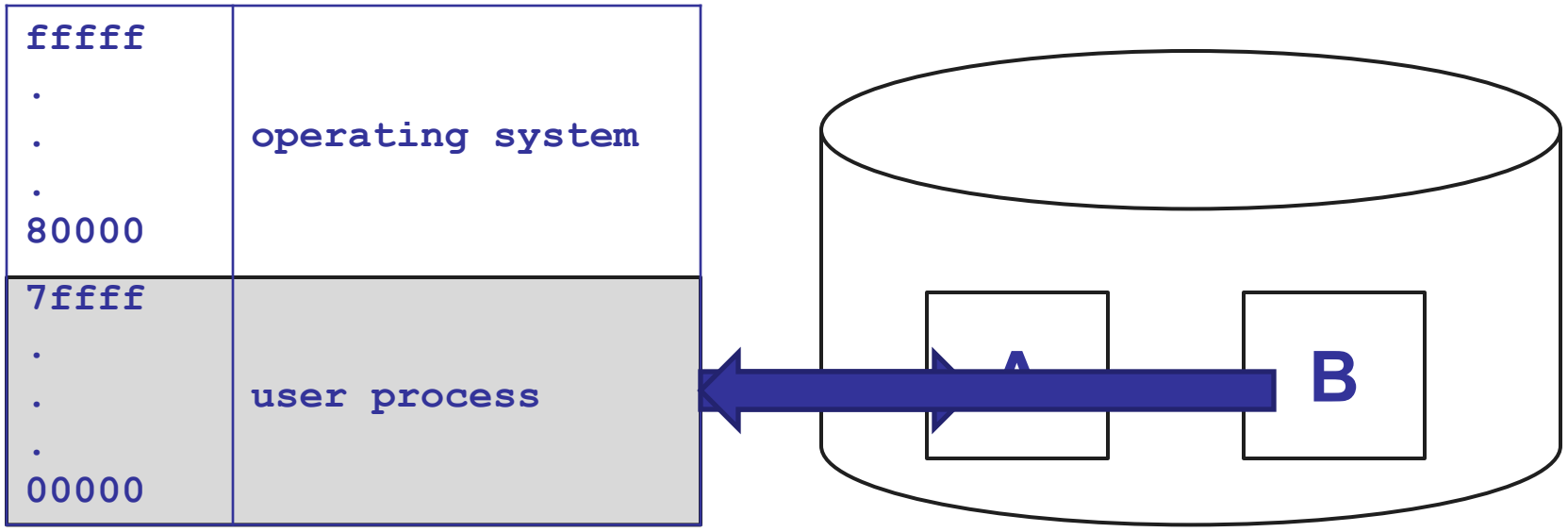
- Virtual memory: an address space can be larger than the machine's physical memory
- Address independence: same numeric address can be used in different address spaces (i.e., different processes), yet remain logically distinct
- Protection: one process can't access data in another process's address space (actually controlled sharing)

Uni-programming

- 1 process runs at a time
- Always load process into same spot in memory
- Reserve space for OS
- Virtual address = physical address
- How to swap between processes?



Swap between processes

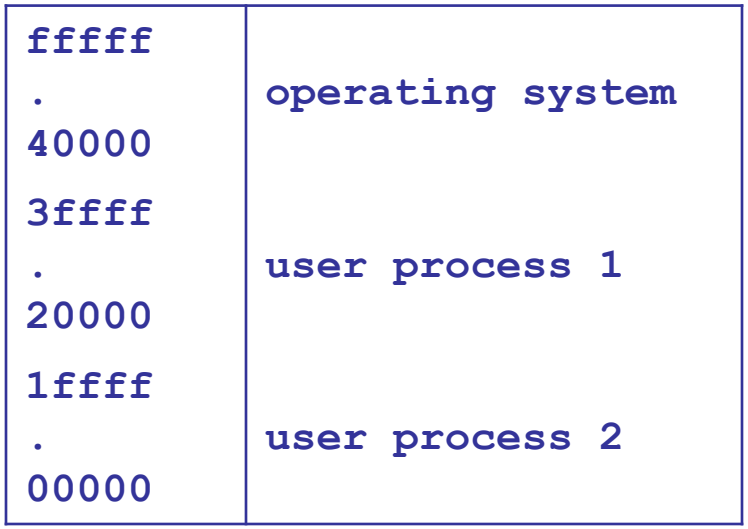


Problems?

Multi-programming

- Multi-programming
  - Allow >1 processes share physical memory
  - We still want address independence: programs written assuming address range starts at 0
  - Now, only 1 process can start at physical address 0
  - Implies **address translation**
    - » Provides address independence
    - » Provides protection (in some cases)
  - Two options:
    - » Static address translation (before execution)
    - » Dynamic address translation (during execution)

Static address translation



- Compiler generates addresses starting at 0
- **Linker-loader** adds offset to instructions
  - E.g., MOV 0x10, %eax becomes MOV 0x20010, %eax
- Any problems?

Dynamic address translation

- Problem is application gets “last move”
  - Compiler generates machine code (app)
  - Linker-loader translates addresses (OS)
  - Register values used to calculate addresses (app)

- Dynamic translation: system has the last move
  - Hardware (MMU) translates all memory references
  - Virtual address: address used by the process
  - Physical address: address in physical memory

Dynamic address translation

- Address Independence
  - Virtual addresses are scoped to 1 process
- Protection
  - One process can't refer to another's address space
- Virtual memory
  - VA only needs to be in phys. mem. when accessed
  - Allows changing translations on the fly

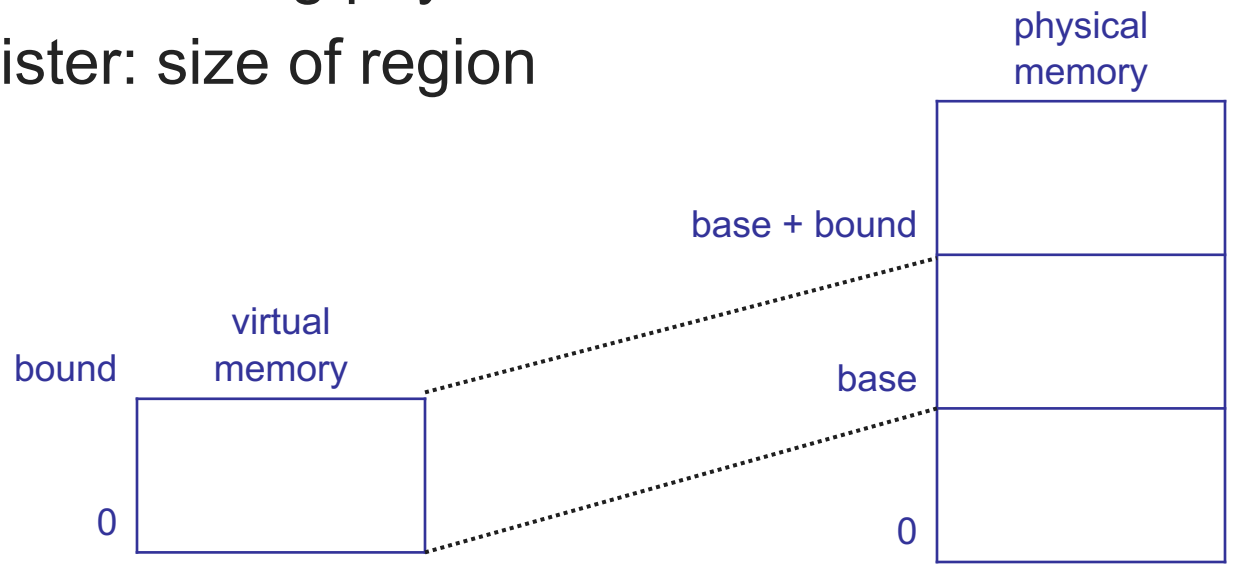
Address translation

- Many ways to implement translator

- Tradeoffs
  - Flexibility (sharing, growth, virtual memory)
  - Size of data needed to support translation
  - Speed of translation

Base and bounds

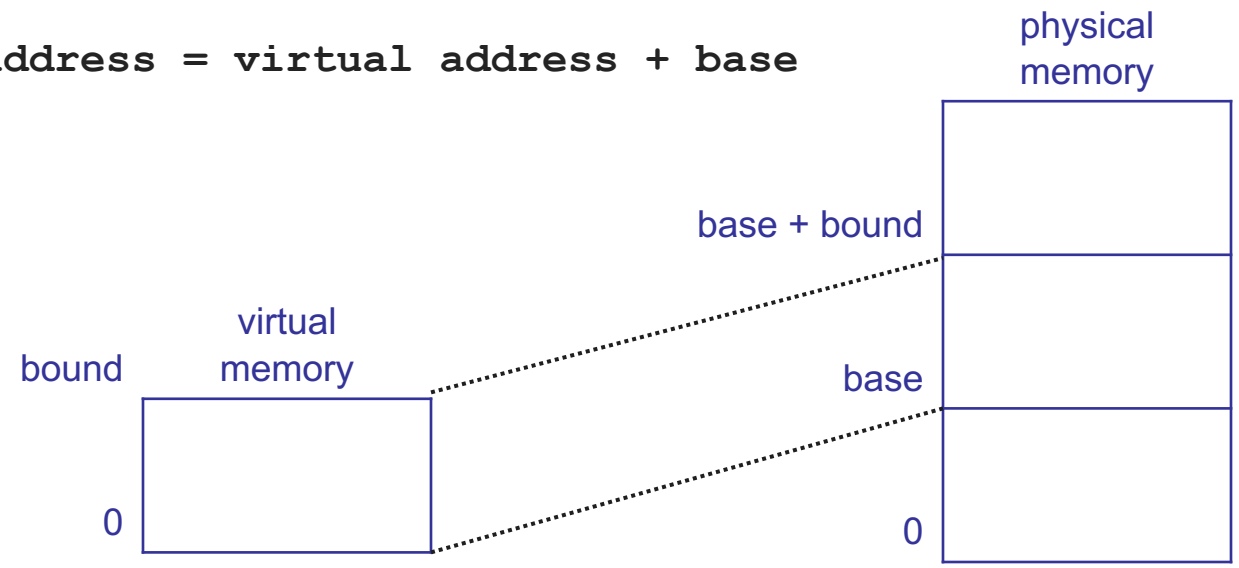
- Load each process into contiguous region of physical memory
  - Prevent process from accessing data outside its region
  - Base register: starting physical address
  - Bound register: size of region



Base and bounds

- MMU Translation:

```
if (virtual address > bound) {
        trap to kernel; kill process (core dump)
} else {
        physical address = virtual address + base
}
```



Base and bounds

- Similar to linker-loader, but also protects processes from each other
- Only kernel can change translation data (base and bounds)

- How to swap between processes?

- What to do when address space grows?

Base and bounds

- Pros?
  - Fast
  - Simple hardware support
- Cons?
  - Virtual address space limited by physical memory
  - No controlled sharing
  - External fragmentation