OS - IV

Memory Management

1. Concurrent Data Structures
2. Deadlocks
   - properties
   - tackling
3. Memory management
   - structured
   - contiguous
   - paging
   - Belady's anomaly

DBMS

- Indexes
- Query optimis.
- ~~Views~~
- window functions

Service | Product

Theory
↳ contiguous

Java
- GC

Code
- Hashmap

DSA
LLD

Thread sync.

① Mutex or locks

— .lock()

— — — } CS

.unlock()

Atomic Integer

↳ locking

andunlocking

Atomic Integer

a = 1
a = a+1 } .getAndAdd( )

↓

.lock( )

a =+1

.unlock( )

concurrent hash maps

10 { }

{ⓐ=1} mutex

locked

1 thread can access a hash map at one time.

<u>Hash Map</u> – acquire a mutex on a call

Slow performance

<u>Concurrent Hash maps</u>

{
1 – a
2 – b
3 – c
4 – d
}

1 – 10 – bucket 1

10 – 20 – 2

30 – 40 – 3

5 - e
6 - f

{ { } - bucket

lock for the bucket

range - 1 - 100
#
- %
- hashes

Lock = # of keys

a() - r()

contain( )

concurrent HM

- reduce the chances of threads waiting
- bucket size
- improving our bucketing algorithm

{1} {3} {5}

- Bucket size
- timeout

T1	T2
→ .lock(R1)	→ .lock(R2)
→ .lock(R2)	→ .lock(R1)
.....	....

.....

· Unlock (R2) · Unlock (R1)

· unlock (R1) · unlock (R2)

T1 LR1 W] .....

T2 LR2 W]

Stale mate

T1 & T2 – W

Deadlock

T1 → T2 } cycle

T2 → T1 } loop

① Mutual Exclusion – The resource can only be held by one process.

2. Hold and wait – T1 holds R1 and waits for R2

3. No preemption – T1 will only release R1 when complete

4. Cyclic waiting – P1 P2 P3

P1 → P2

P2 → P3

P3 → P1

Cycle

RAG

T1 → R1
T2 → R2
T1 → R2
T2 → R1

identify cycle
if RAG has
cycles, deadlock

Banker's algorithm

Tackling deadlocks

1. Prevention
2. Avoidance

③

④

R1, R2, R3

T1	T2
R2	R1
R1	R2

- resources are always allocated in order, T1 → R1 R2 ; released

T2 → blocked blocked R1 R2...

Deadlock avoidance

System does not go in an unsafe state.

R1 R2

T1	T2
lock R1	lock R2
lock R2	lock R1

OS - RAG

- next step in RAG
- deadlock can happen)

Banker's algorithm

Safety's algorithm

Avoidance

---

1. Prevention
2. Avoidance
3. Detect and Recover

(9)

(1) Release resources

(2) Aborting any or all processes

③ Resource scheduler,
preemption –

Ignorance

Windows | Linux

→ don't do anything

---

① Prevention – in-order resource allocation

② [illegible]

② Avoidance - Safe [illegible]

③ Detect & Recovery - abort
- resource preemption

④ Ignorance is bliss
→ most common OSes

jitter
% - P1 - 0.2
P2 - 0.4

TinyOS

How to handle deadlock at an application level?

① Timeouts

Server — application timeout

- 2s, all request should be killed

.lock   .tryAndlock

R1 T1 .lock
boolean - false

→ .lock(5)

lock with caution

6:18 | 6:22
10:48 | 10:52

Memory management

volatile or non-volatile

persisted or not

ROM ≠ Disk

1 TB

Applications

① App. stored on disk

② It gets loade into RAM

16/16

20

~~4 GB~~

Out of memory

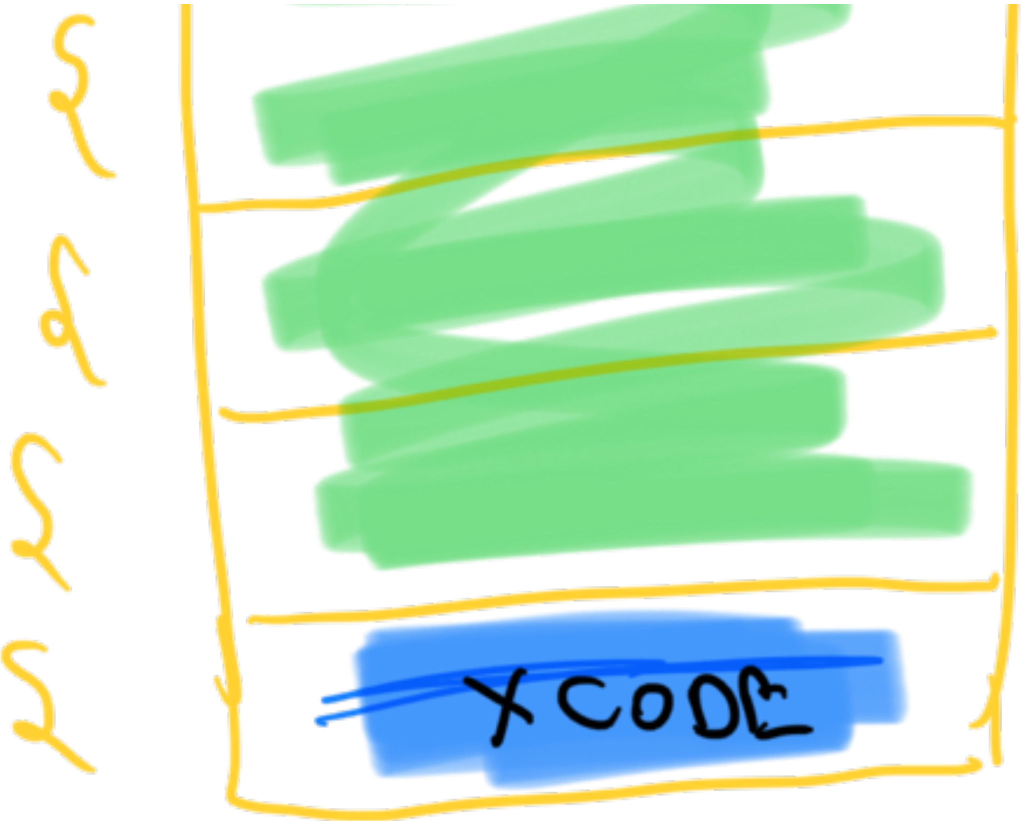
How are processes stored in memory?

contiguous memory allocation

fixed sized partition

1B

COD → 4 B



XCODE → 1 B

0.5 B

0.5 B → 1 B

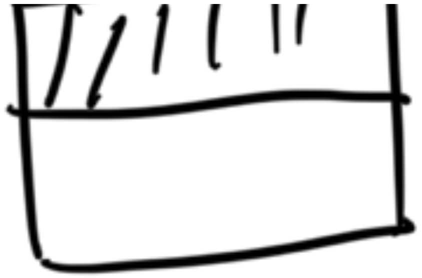
wasting resources

Fragmentation

XCODE

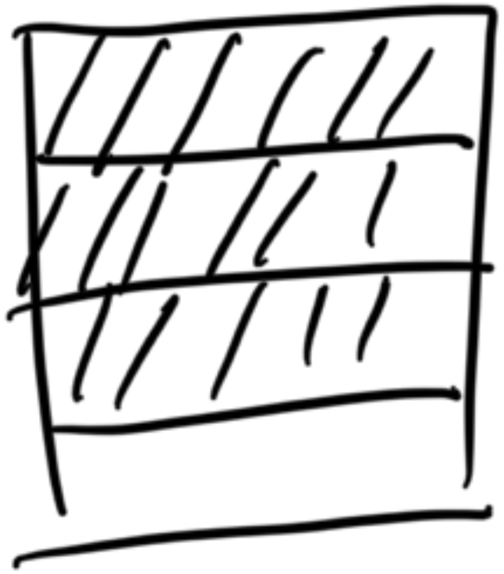
- 2 blocks

Internal fragmentation



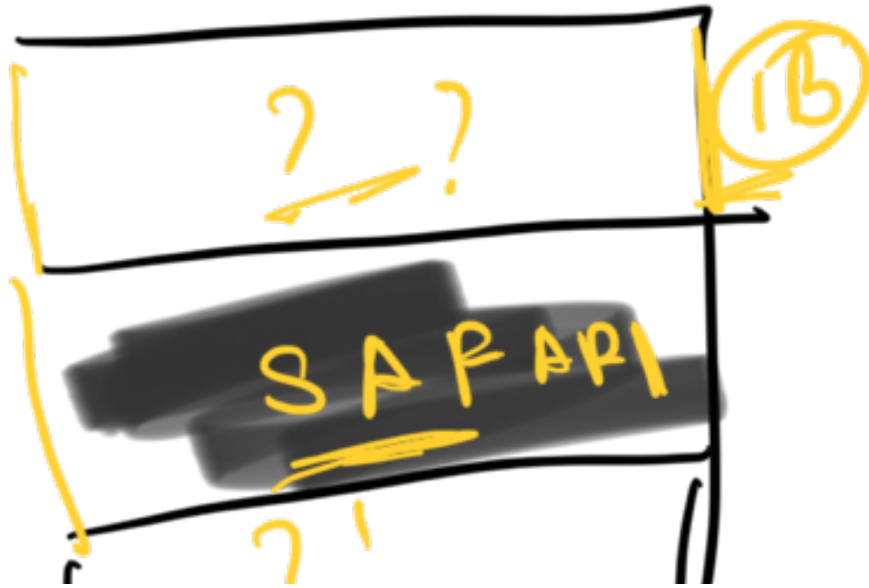
1 block

1 B



fixed

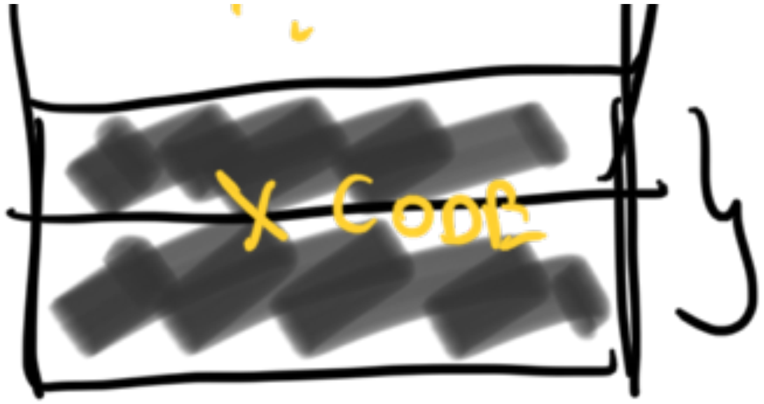
2B — ‡ -1 +1



XCODE — 2B

Safari — 1B

contiguous



COD - Lite - 2B

In are spaces that are not assignable

External fragmentation

All my data is together

Paging

1. memory into fixed size block

=> **pages**

1. storage into a fixed size blocks

=> **frames**

size(page) == size(frame)

x code {
x = 1
y = 2

Page Table

variable	logical addr
x	1
y	2

Page Table

RAM

process entity to logical addr

MEM → MMU

1. x code tells my OS variable x
2. x code goto page table x

   x = (1)

3. x code MMU get variable at (1)

MMU

LA	PA
1	abc

x = 1
y = 2
get
MMU
Storage
a
b
Page Table
x
LA
y
2
LA
PA
2
b

1. Check Page Table ⇒ LA
2. Get value from MMU → (LA)
3. Check s [LA ⇒ PA] table
4. If the data is in memory, return
   else, get data from disk

---

xcode
storage

Page replacement

page replacement

fifo, lifo

Page fault

Page Fault

- when the MMU does not find a page in memory

x : 1
y : 2
z : 19

page fault

PR algorithms - page faults

Thrashing

- excessive page faults

1 2 3 4 5 4 3 2 1

less no. of frames page.

Thrashing

increase ram

Belady's anomaly ⇒ II d c b a d c e d

P2 - 1

P3 - (2)

P2 - 2B

---

pages RAM

id to logical

logical — x 1

physical — y 2

MMU → LA

RAM

MMU — a

page replacement

MM (PA)

page table

id — LA

id — PA

MMU

LA — PA

page table — MMU

process

① DS | ML

② [illegible] — ECB