Designing a Compositional Real-Time Operating System

Christoph Kirsch
Universität Salzburg

ARTIST Summer School Shanghai
July 2008

tiptoe.cs.uni-salzburg.at[#]

- Silviu Craciunas* (Programming Model)
- Hannes Payer* (Memory Management)
- Harald Röck (VM, Scheduling)
- Ana Sokolova* (Theoretical Foundation)
- Horst Stadler (I/O Subsystem)

[#]Supported by a 2007 IBM Faculty Award and the EU ArtistDesign Network of Excellence on Embedded Systems Design
*Supported by Austrian Science Fund Project P18913-N15

What We Want

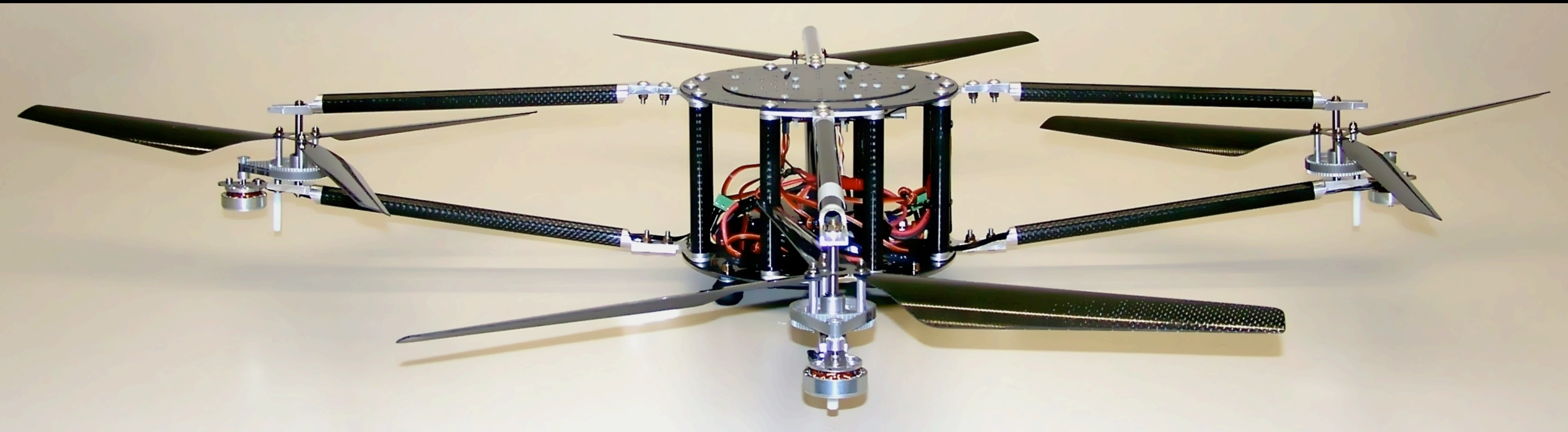
1. Focus on principled engineering of real-time and embedded software
2. Study the trade-off between temporal and spatial performance and predictability as well as compositionality of real-time programs
3. Design and implement a real-time operating system kernel from scratch to support higher levels of real-time programming abstractions

“Theorem”

- (Compositionality) The time and space a software process needs to execute is determined by the process, not the system and not other software processes.
- (Predictability) The system can tell how much time and space is available without looking at any existing software processes.

“Corollary”

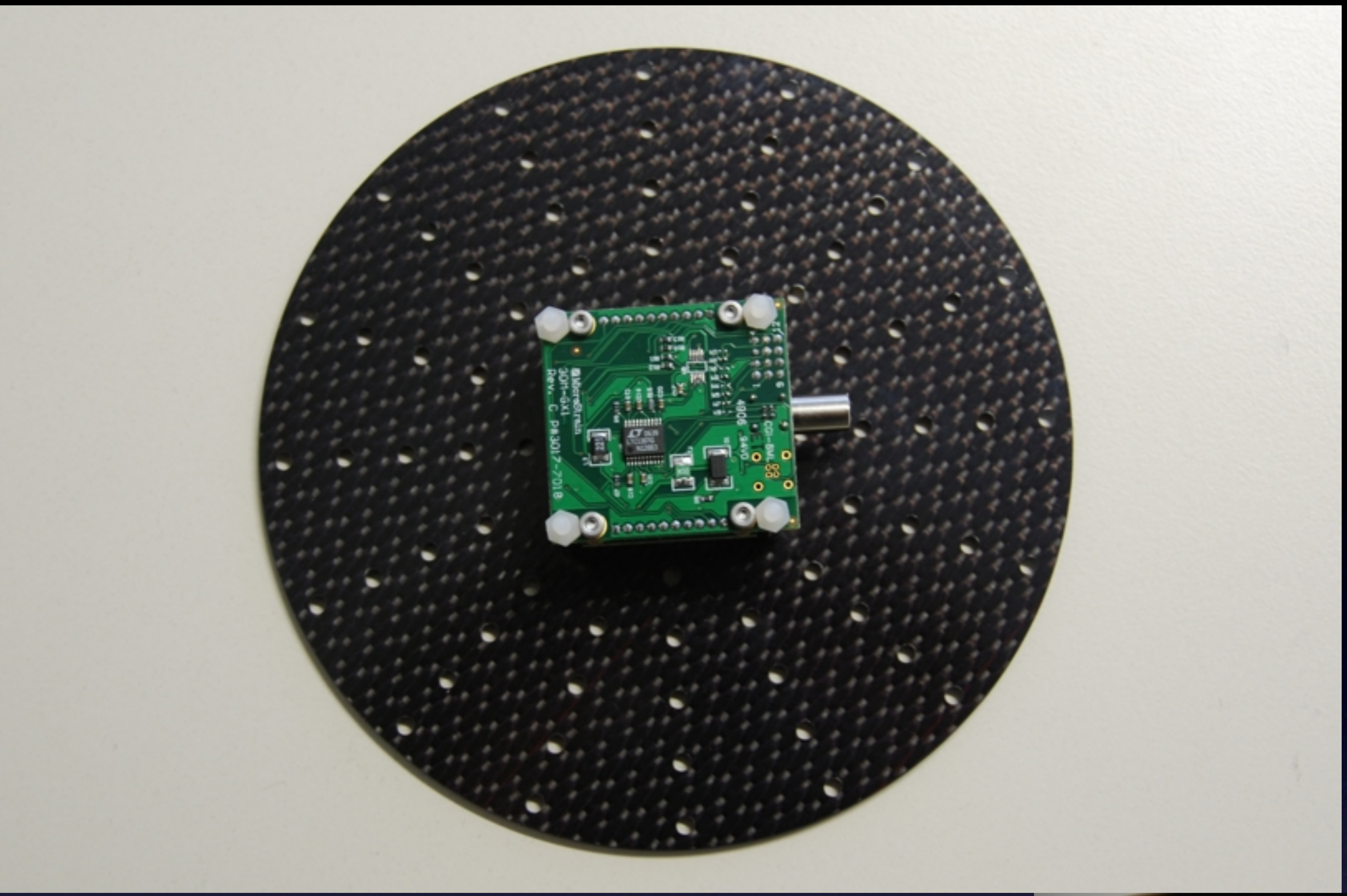
- (Memory) The time a software process takes to allocate and free a memory object is determined by the size of the object.
- (I/O) The time a software process takes to read input data and write output data is determined by the size of the data.



The JAviator

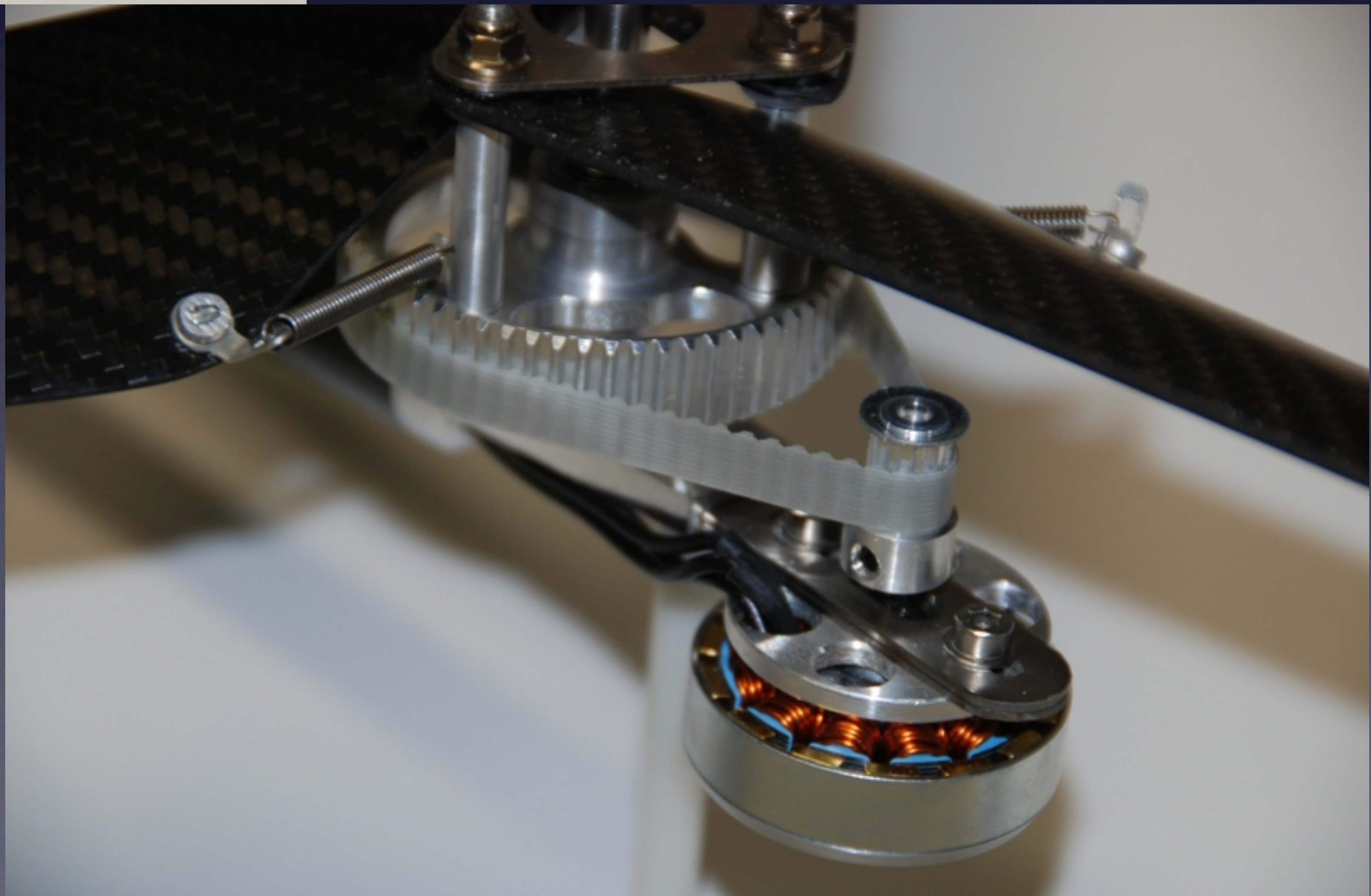
javiator.cs.uni-salzburg.at

Quad-Rotor Helicopter

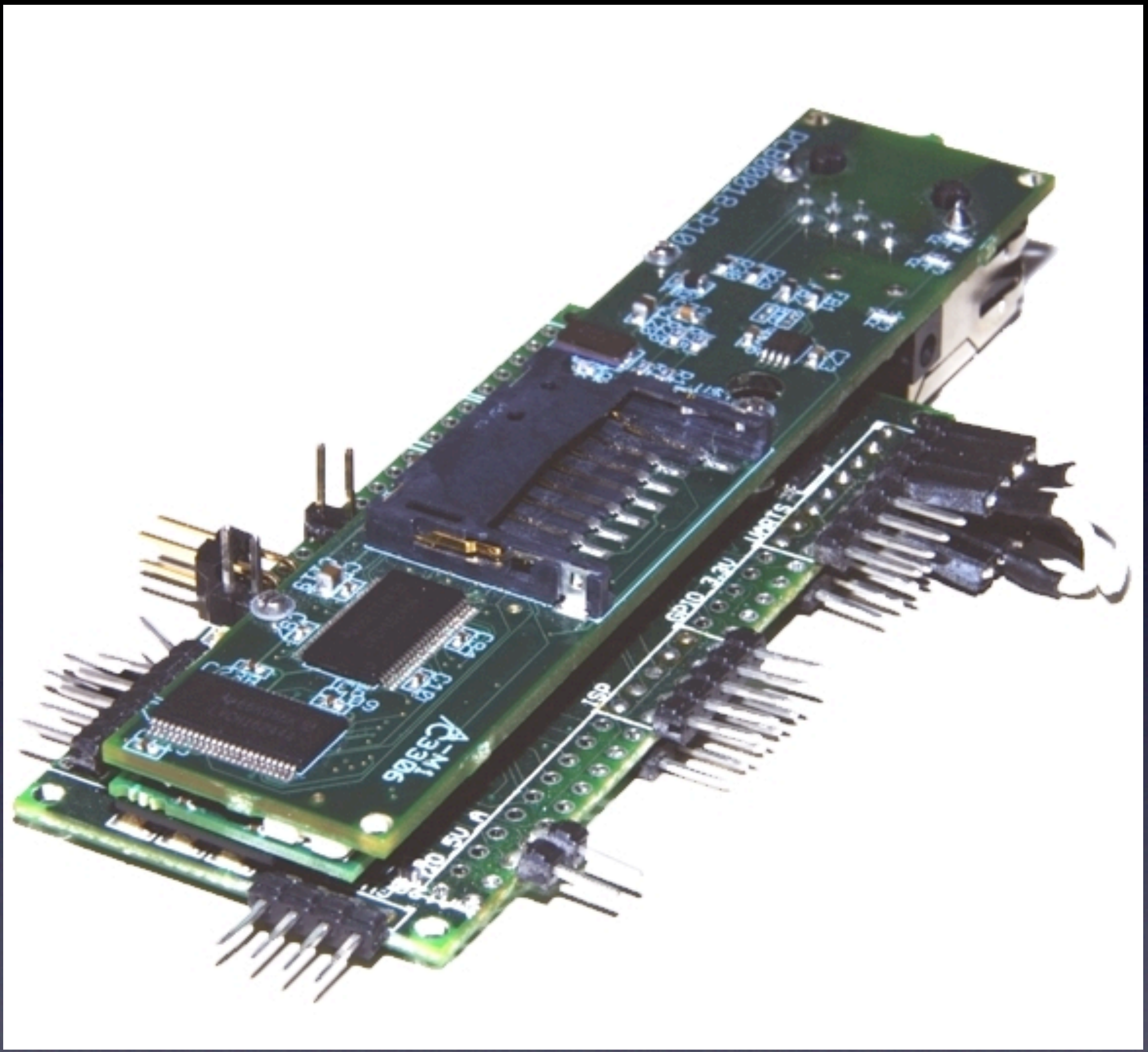


Gyro

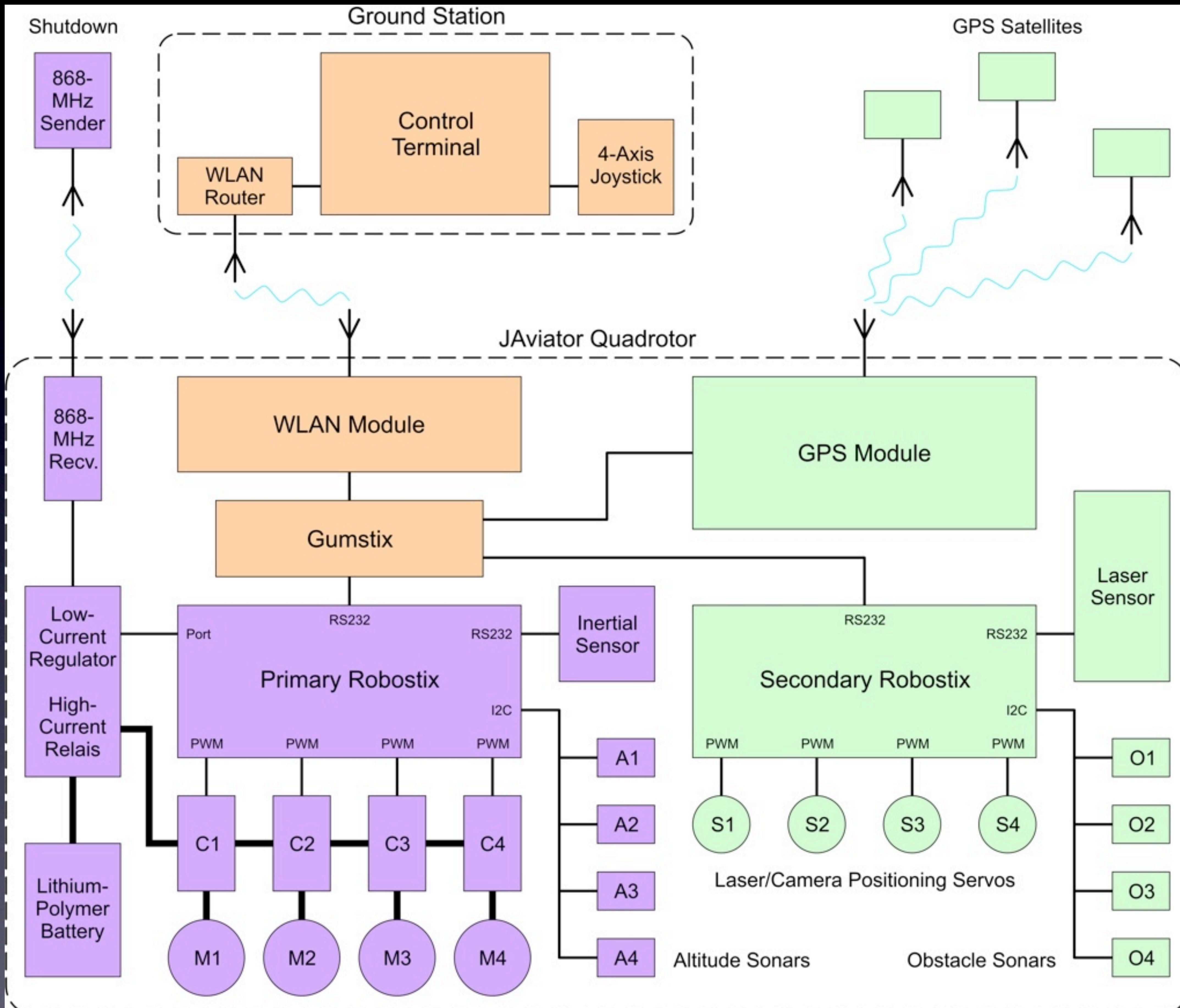
Propulsion



Gumstix

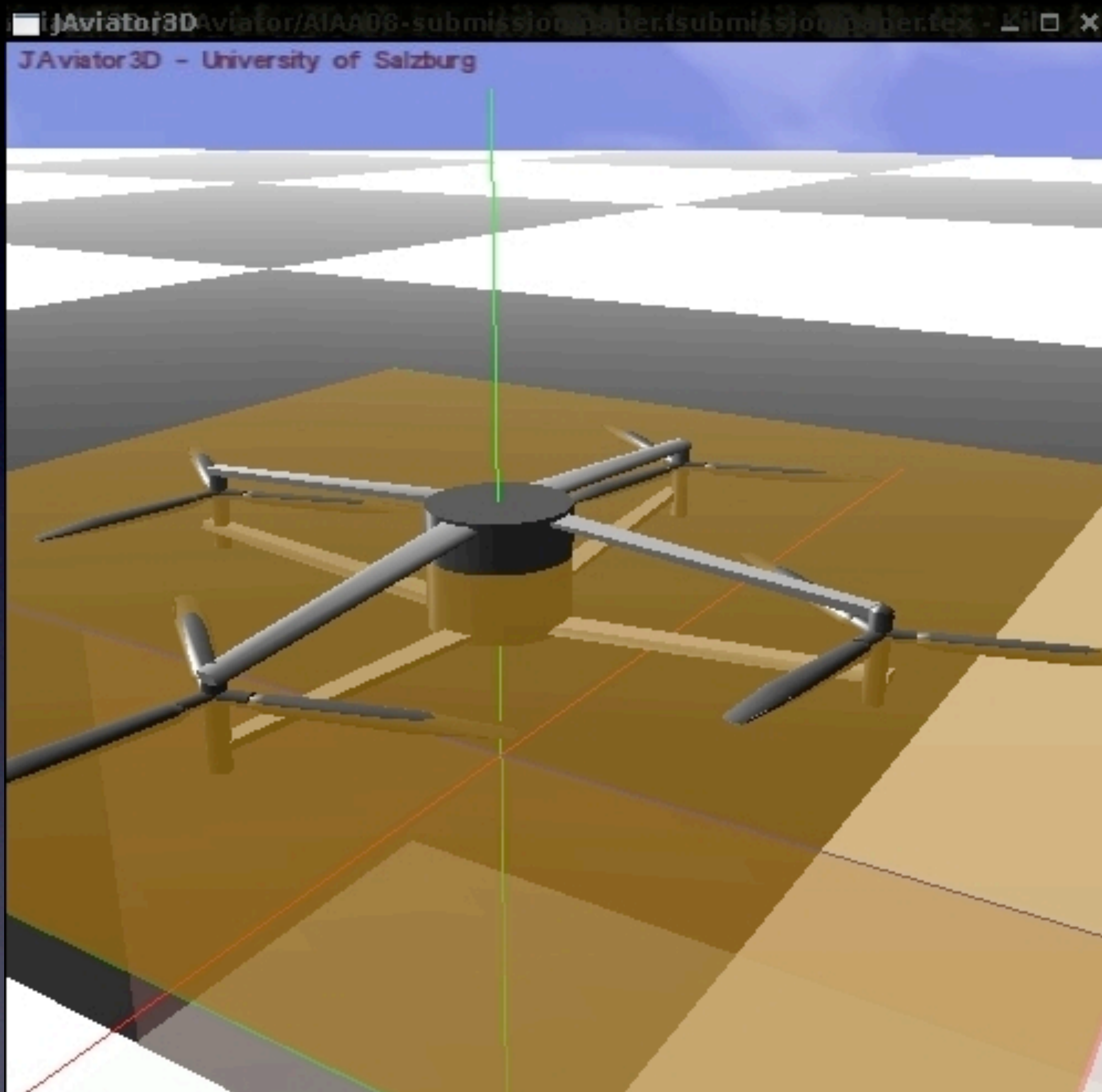


600MHz XScale, 128MB RAM, WLAN, Atmega uController



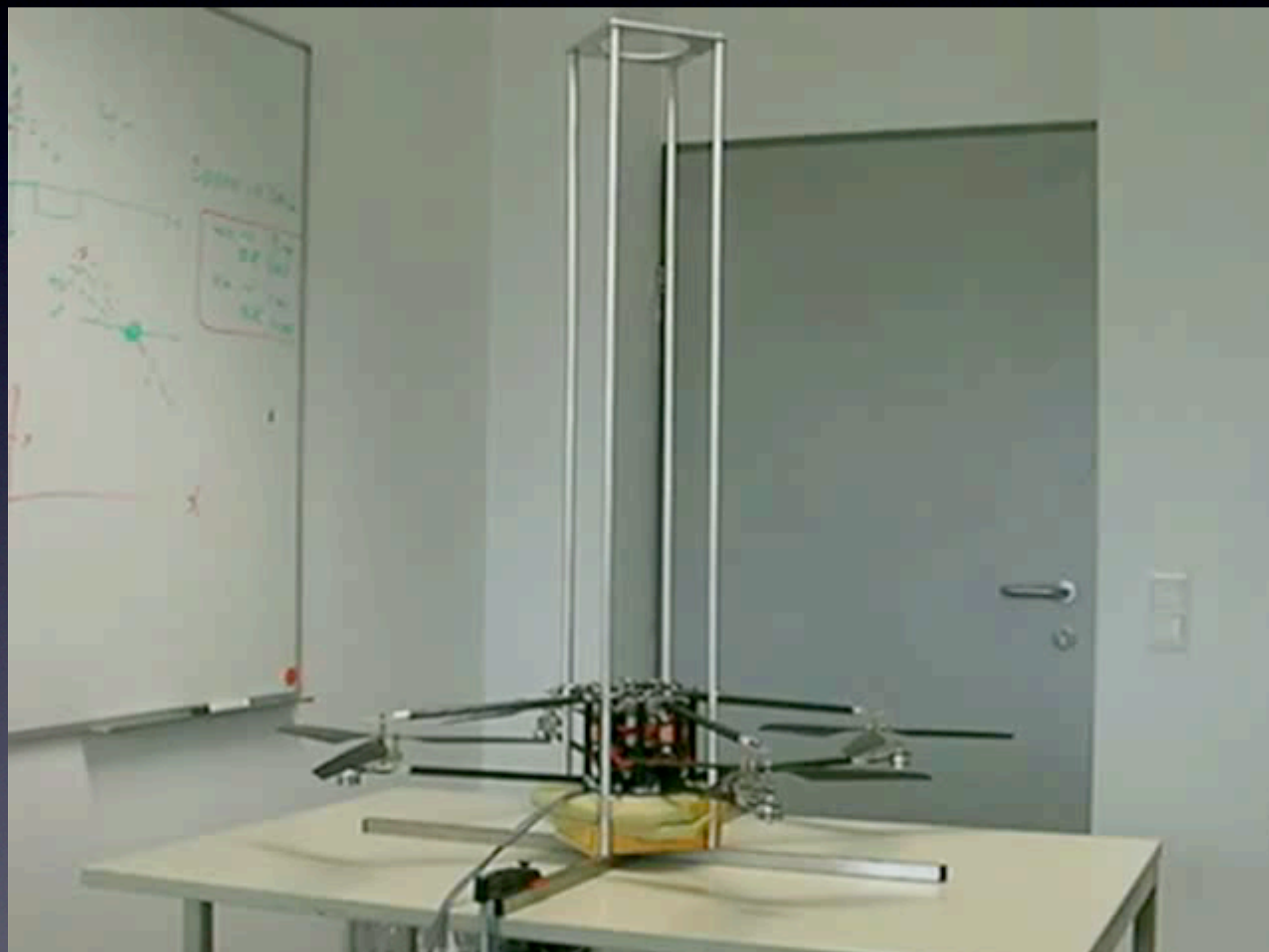
Shutdown
868-MHz Sender
Ground Station
Control Terminal
WLAN Router
4-Axis Joystick
GPS Satellites
JAviator Quadrotor
868-MHz Recv.
WLAN Module
GPS Module
Gumstix
Laser Sensor
Low-Current Regulator
High-Current Relais
Port
RS232
RS232
Primary Robostix
I2C
PWM
PWM
PWM
PWM
Inertial Sensor
RS232
RS232
Secondary Robostix
I2C
PWM
PWM
PWM
PWM
C1
C2
C3
C4
Lithium-Polymer Battery
M1
M2
M3
M4
A1
A2
A3
A4
Altitude Sonars
S1
S2
S3
S4
Laser/Camera Positioning Servos
O1
O2
O3
O4
Obstacle Sonars


JAviator Control Terminal
Yaw Limit
Joystick Mode
Direct Connection
Connect to Heli
Start Test Mode
Start Logging
Roll Limit
Yaw
deg
Front
Left
Hover
Right
Rear
Roll
deg
-1
-1
0
1
Offsets
1
33 Yaw Trim
Roll Trim 0
0
-1
0
Pitch
deg
Altitude
cm
351
353
Signals
352
351
Pitch Trim 0
Altitude Limit
Port Settings
Reset Needles
Shut down Heli
Key Assistance
About Terminal
Pitch Limit

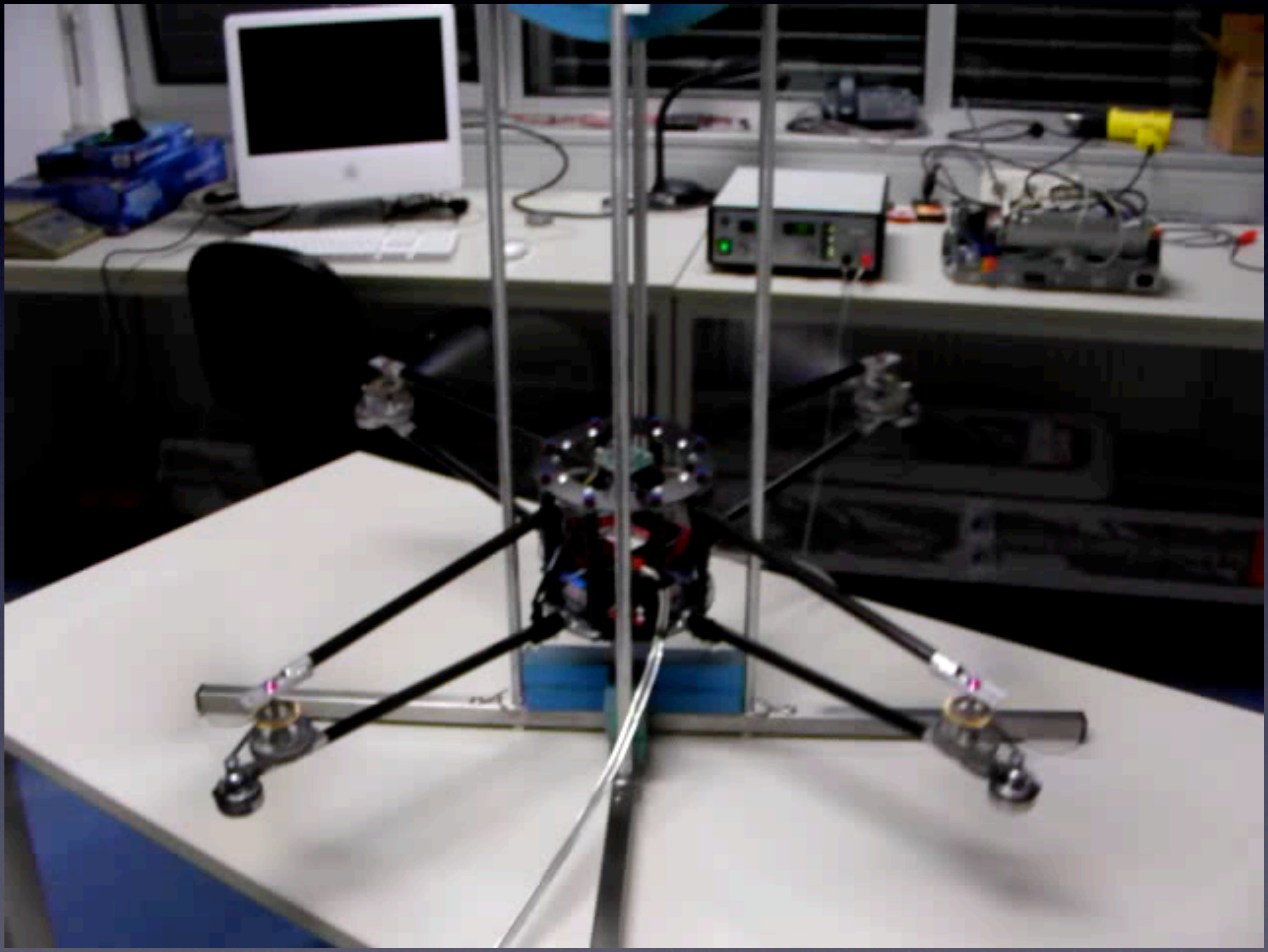

JAviator3D
JAviator3D - University of Salzburg


JAviator3D
JAviator3D - University of Salzburg

Oops

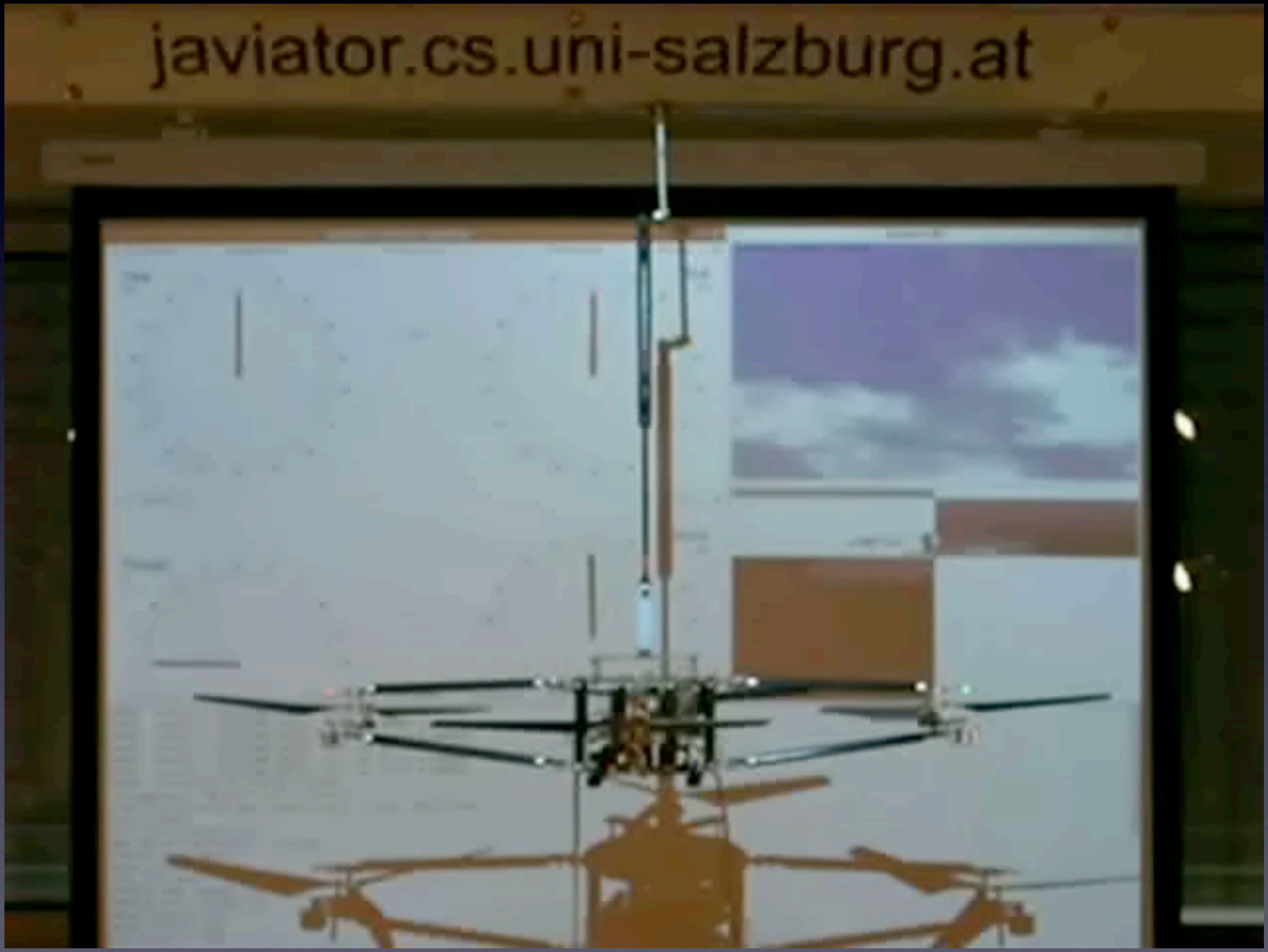


Flight Control



Free Flight

July 11, 2008



Outline

1. Introduction
2. Process Model
3. Concurrency Management
4. Memory Management
5. I/O Management

Applications

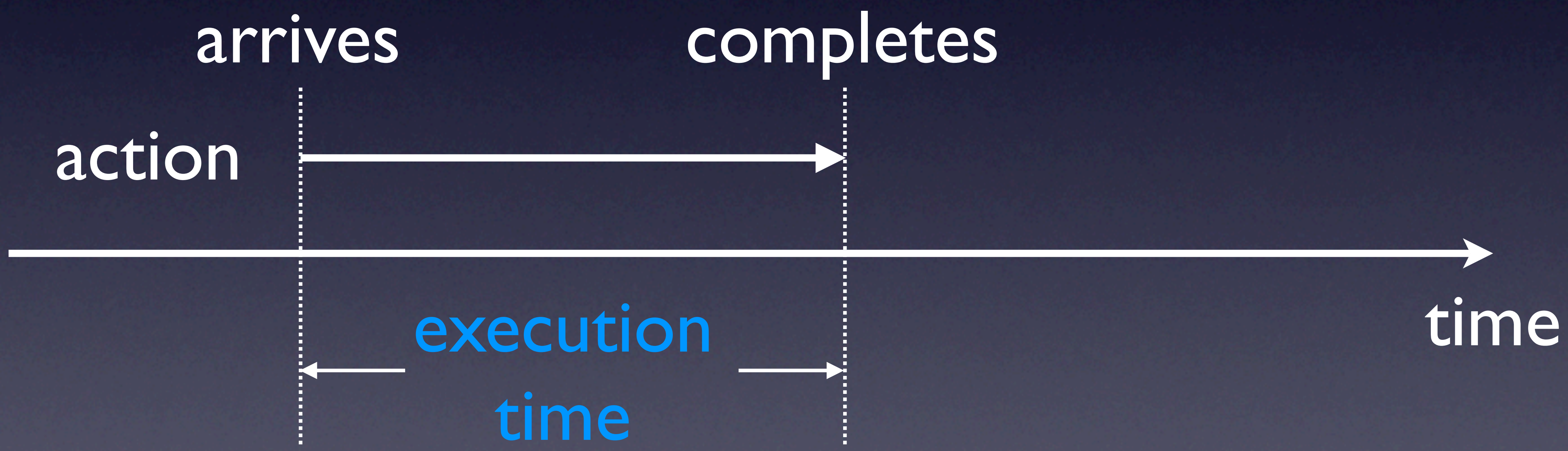
Operating System

Hardware

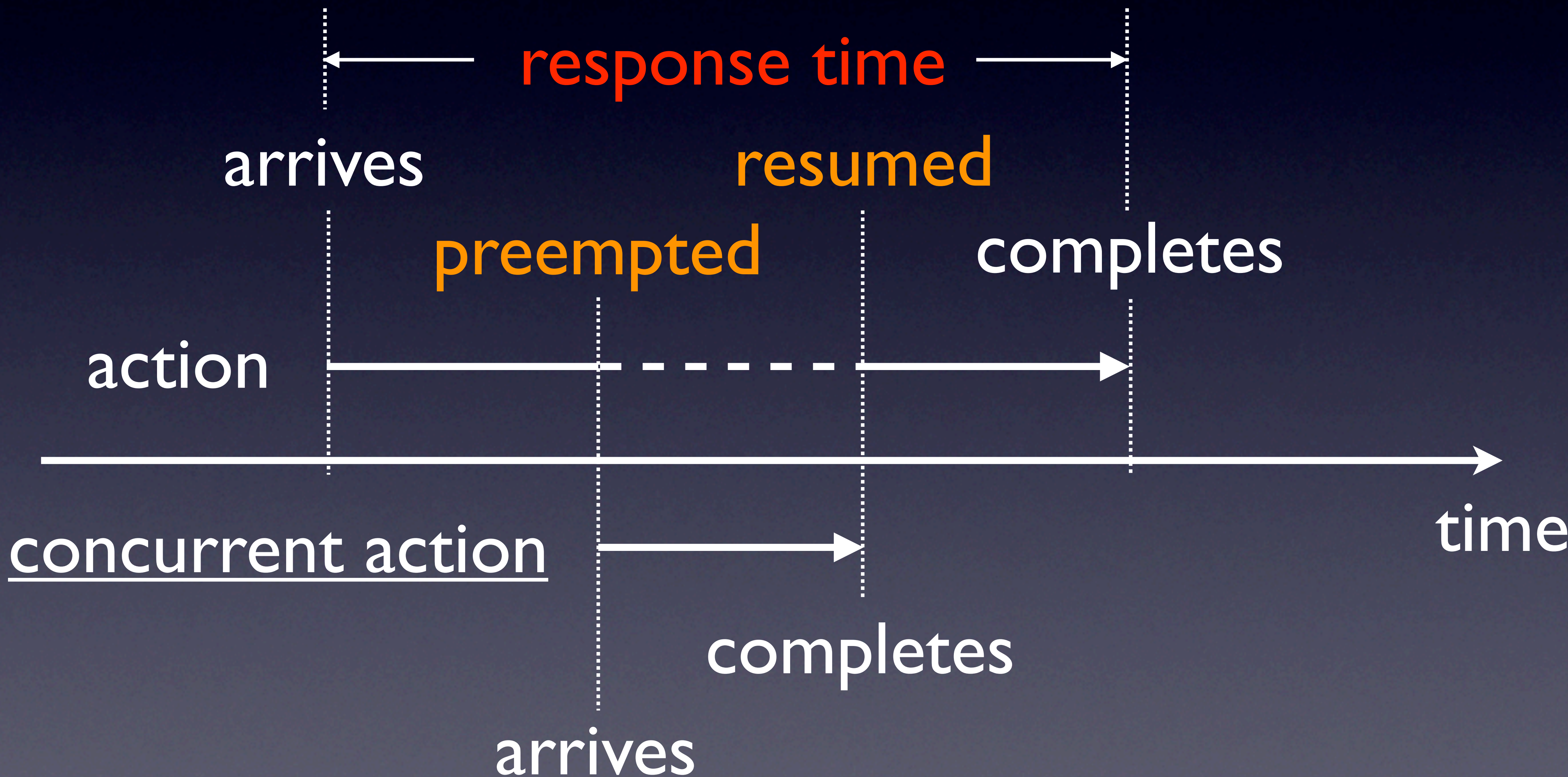
Application and Resources

application-oriented real-time programming	resource-oriented real-time programming
processes	processors/memory
concurrency	distribution/isolation
response times	execution times
frequencies	timers

Process Action



Concurrency

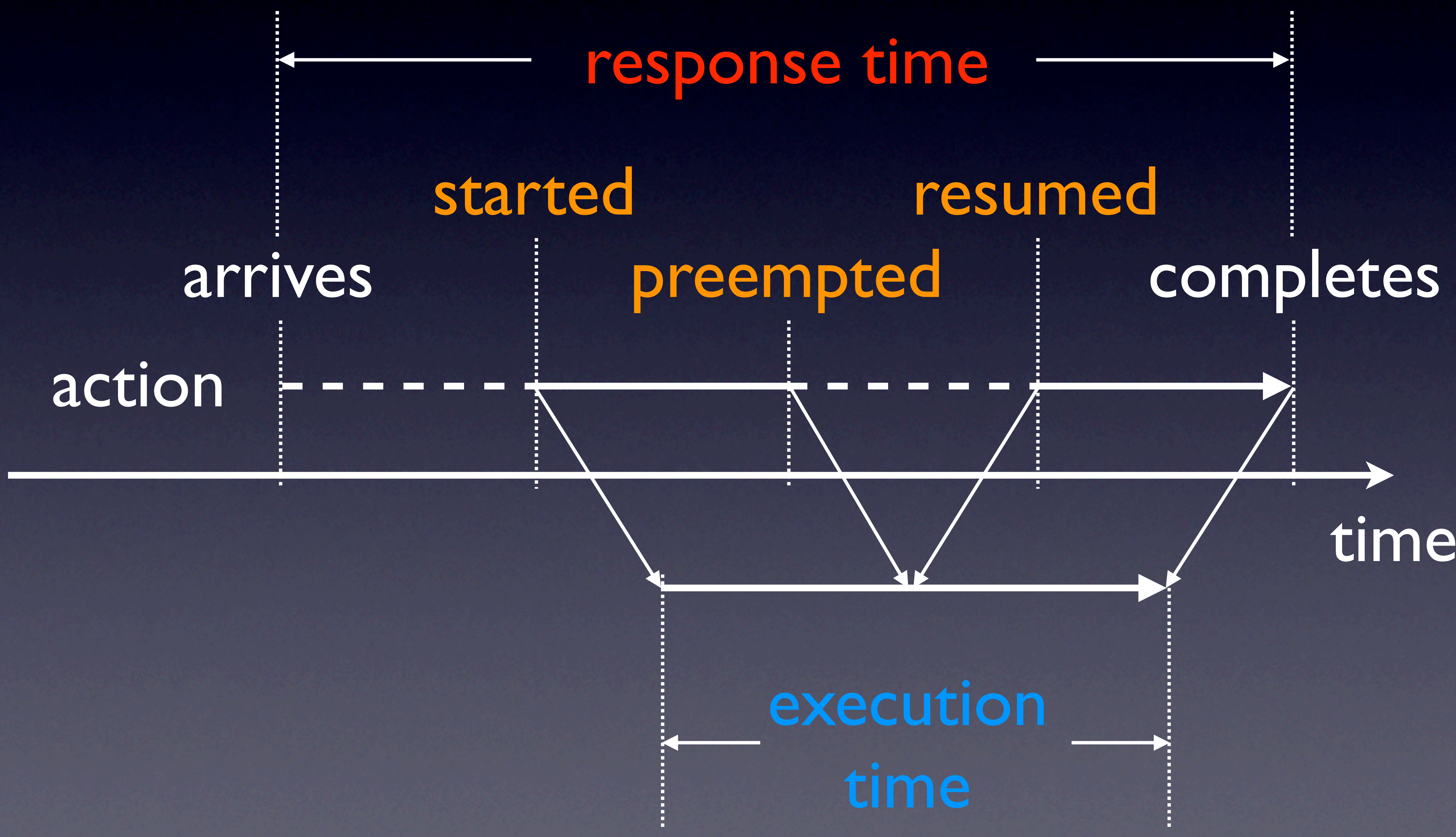


Process

vs.

System

Execution and Response



Time

- The temporal behavior of a process action is characterized by its execution time and its response time
- The execution time is the time it takes to execute the action in the absence of concurrent activities
- The response time is the time it takes to execute the action in the presence of concurrent activities

Analyses

1. The execution time of a process action is determined by the process action and the executing processor.
   - Worst-case execution time (WCET) analysis
2. The response time of a process action is determined by the entire system of processes executing on a processor.
   - Real-time scheduling theory

WCET

- The worst-case execution time (WCET) of a process action on a given processor is an upper bound on the execution times of the action on the processor on any possible input
- The challenge is to compute the least conservative WCET on the most up-to-date processor architectures with the least amount of programmer assistance

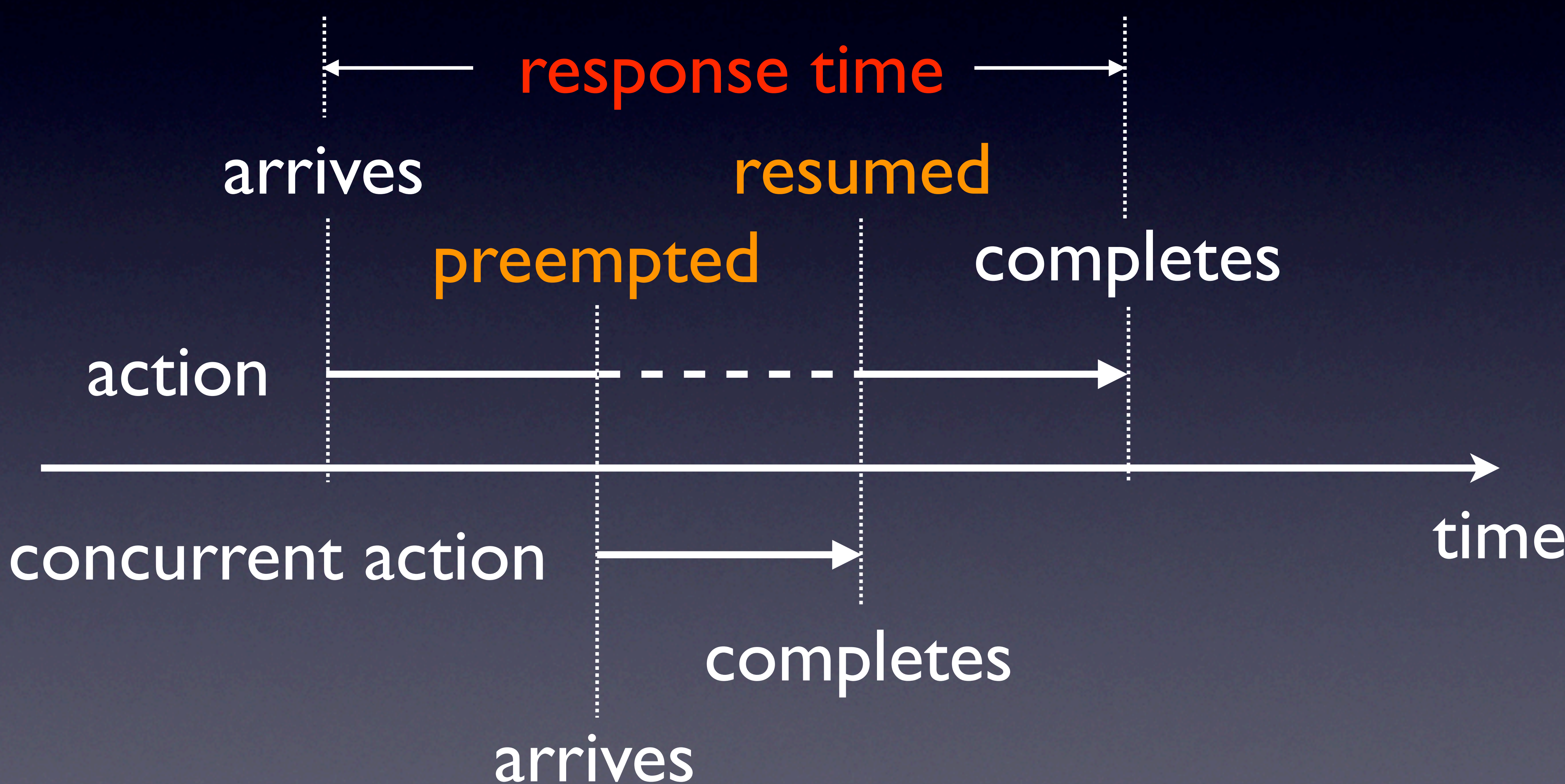
WCET Analysis

- The WCET analysis of a process action on a given processor involves the machine code implementation of the action and the machine code performance of the processor
- The less conservative a WCET bound is the more utilized a system may potentially be since WCETs constrain schedulability (in hard real-time applications)

Real-Time Scheduling

- The worst-case response time of a process action in a given context of other concurrent process actions is bounded by its WCET and the interference from the other actions
- The process model determines the context
- The scheduling algorithm determines the interference

Context & Interference



Context

- Standard model: a <u>process</u> *P* periodically invokes a process action (also called task or job) with a <u>WCET</u> λ_P and a <u>period</u> π_P
  - $P = (\lambda_P, \pi_P)$
- Advanced models: sporadic, aperiodic, conditional, logical, synchronous etc.
- Key advantage: <u>finite</u> description of temporal context of <u>non-terminating</u> processes

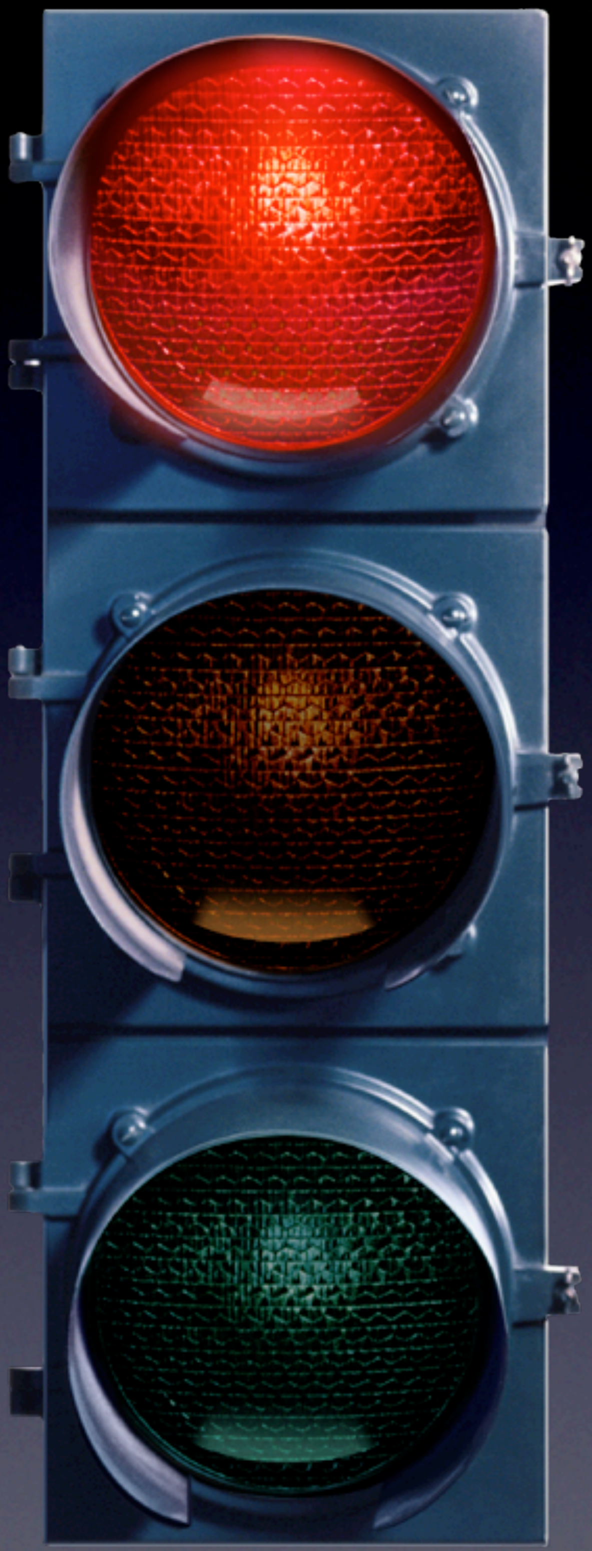
Interference

- A scheduling algorithm *A* determines for a given set of processes a schedule, i.e., for each time instant which process executes
- A schedulability test *T* for *A* determines whether a given set of processes can be scheduled by *A* (is schedulable or feasible) such that "timeliness" holds (e.g. deadlines are met)
- Schedulability involves matching application requirements and resource capabilities

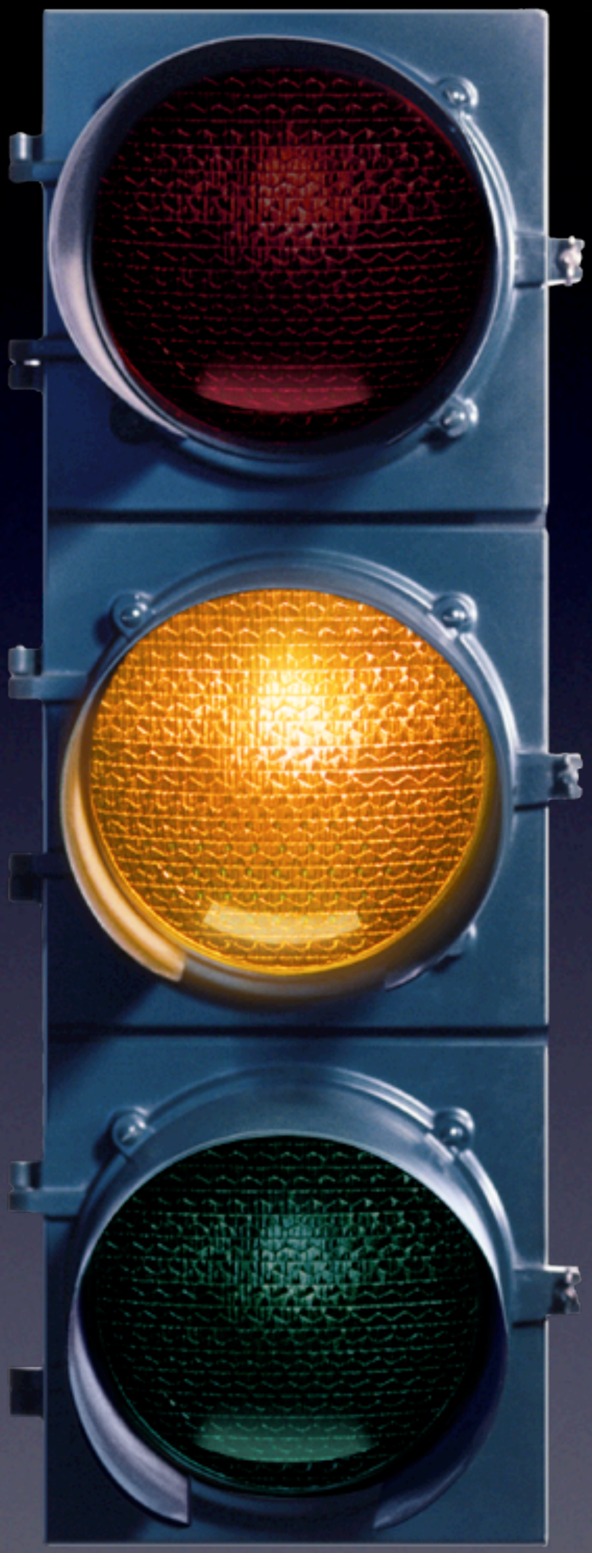
Process States

- A process (action) that has completed and not yet arrived is called *blocked*
- A blocked process (action) may also be called *waiting* (e.g. for some event to occur)
- A process (action) that has arrived and not yet completed is called *ready*
- A process (action) that is executing is called *running*

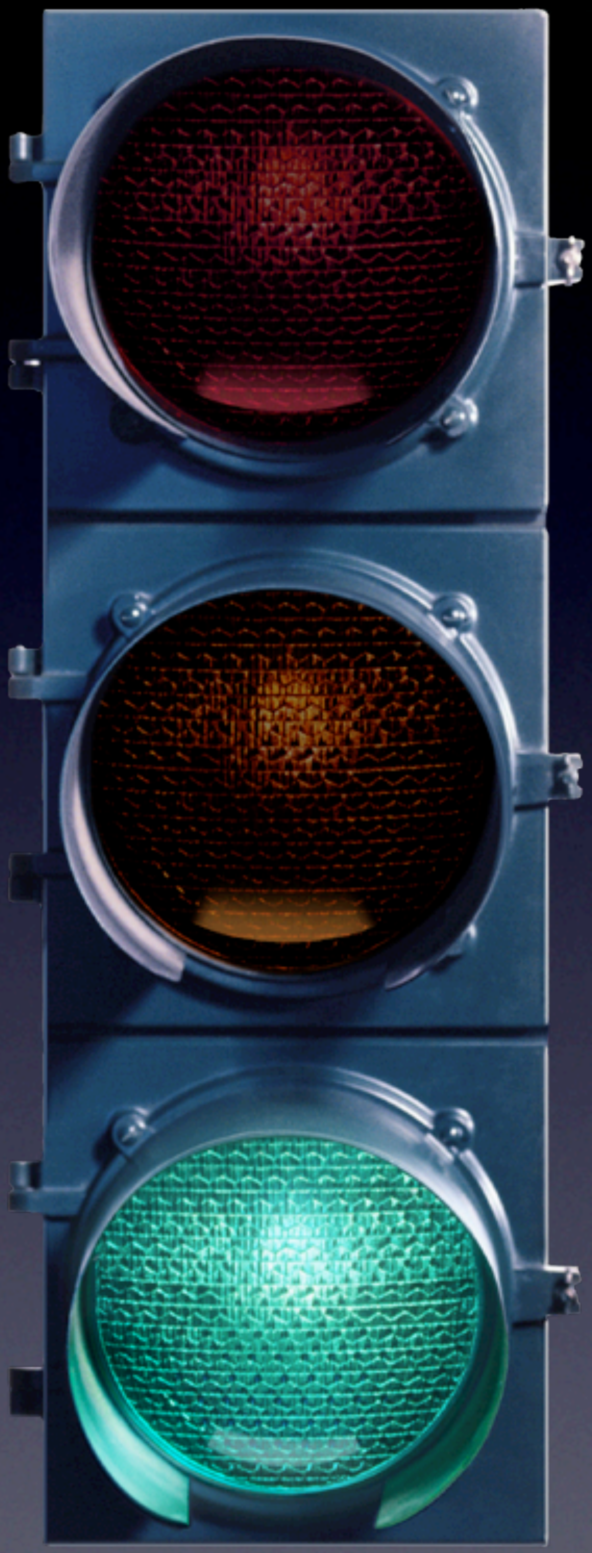
blocked process

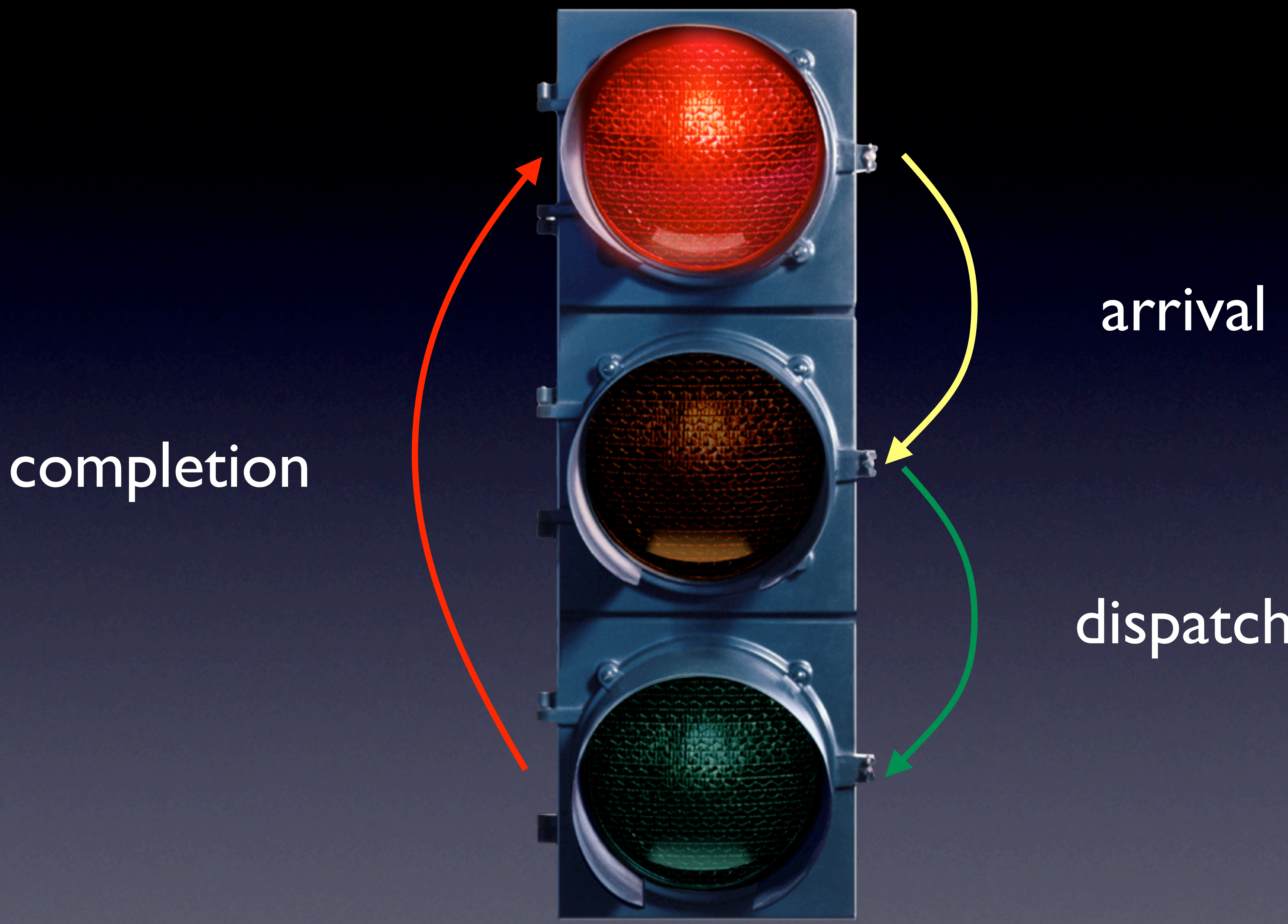


ready process

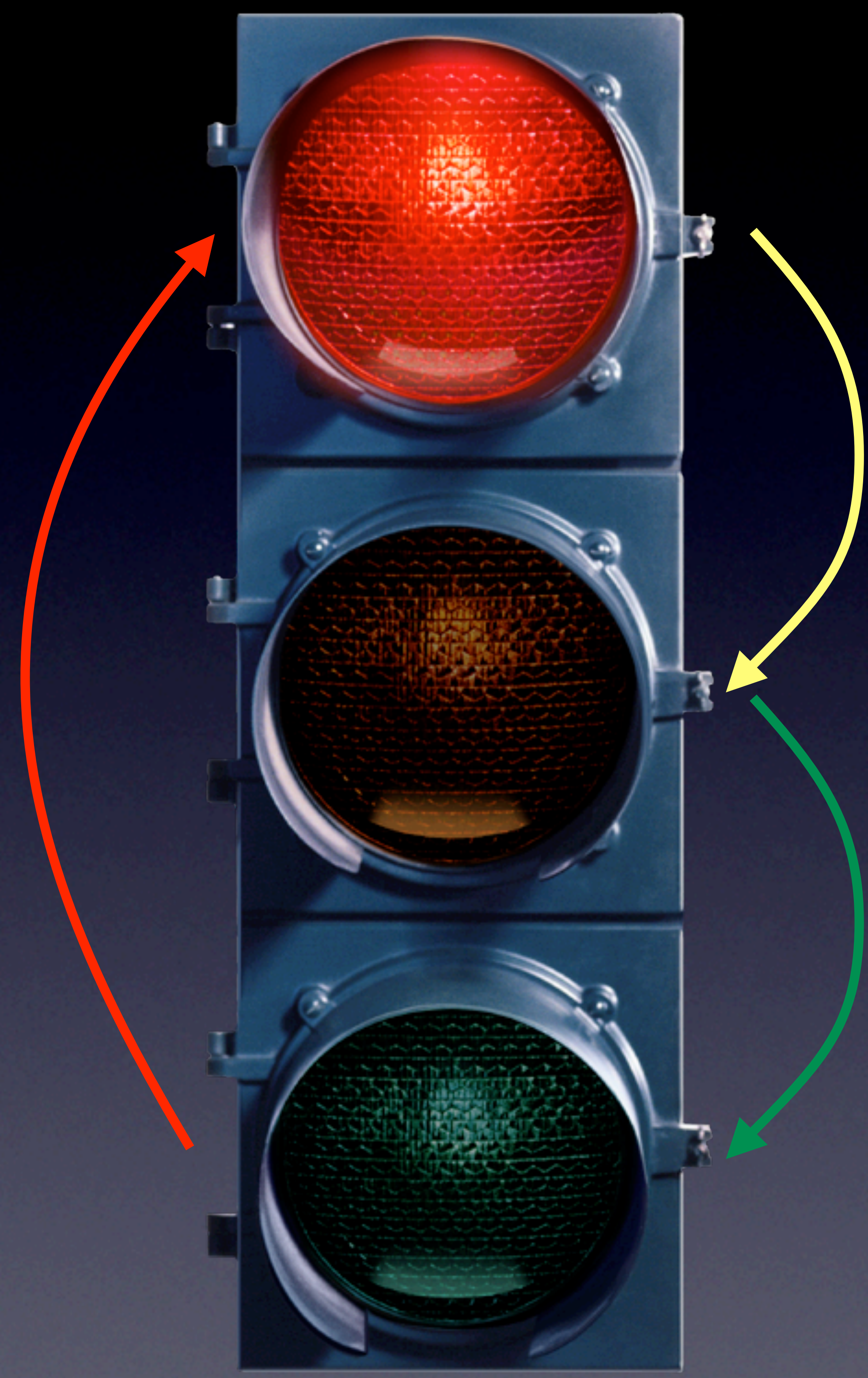


running process



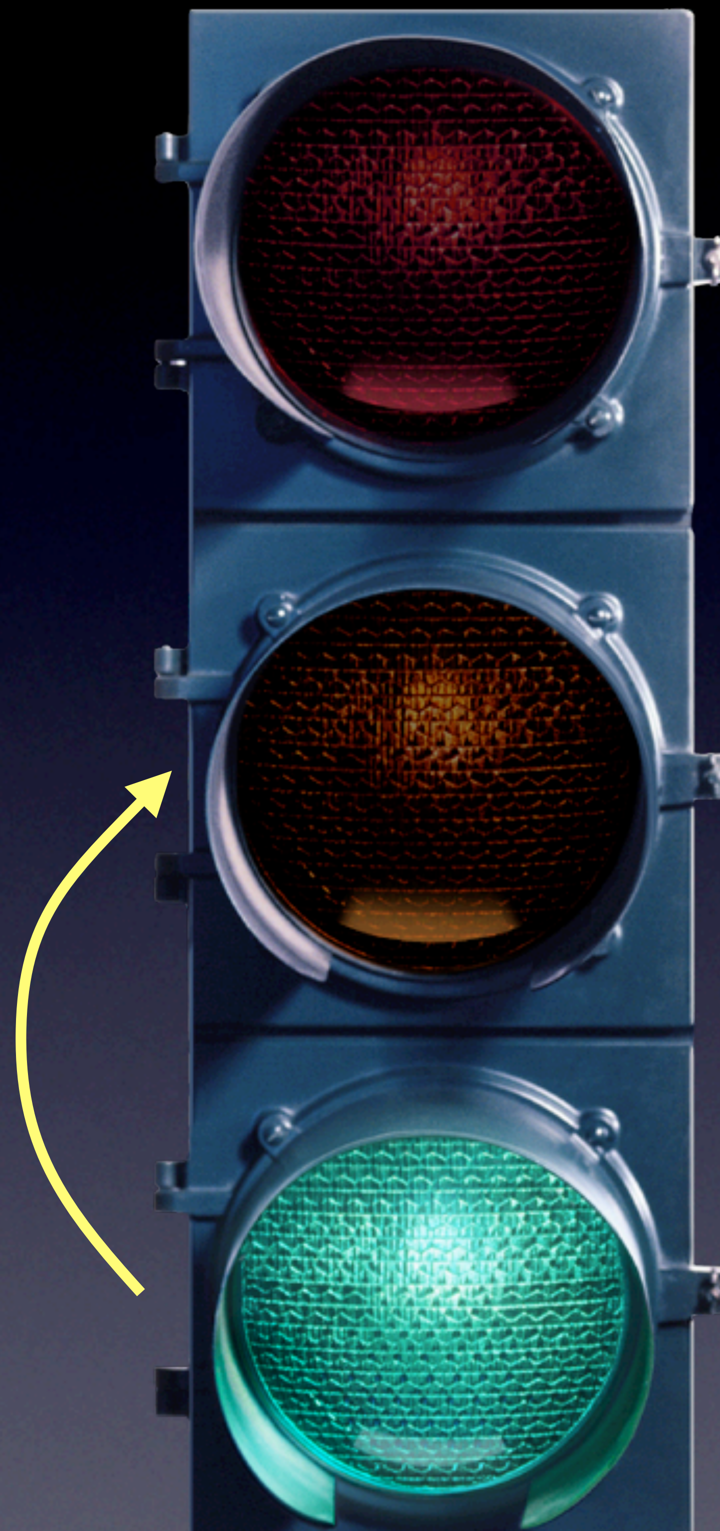

arrival
dispatch
completion

process
arrives



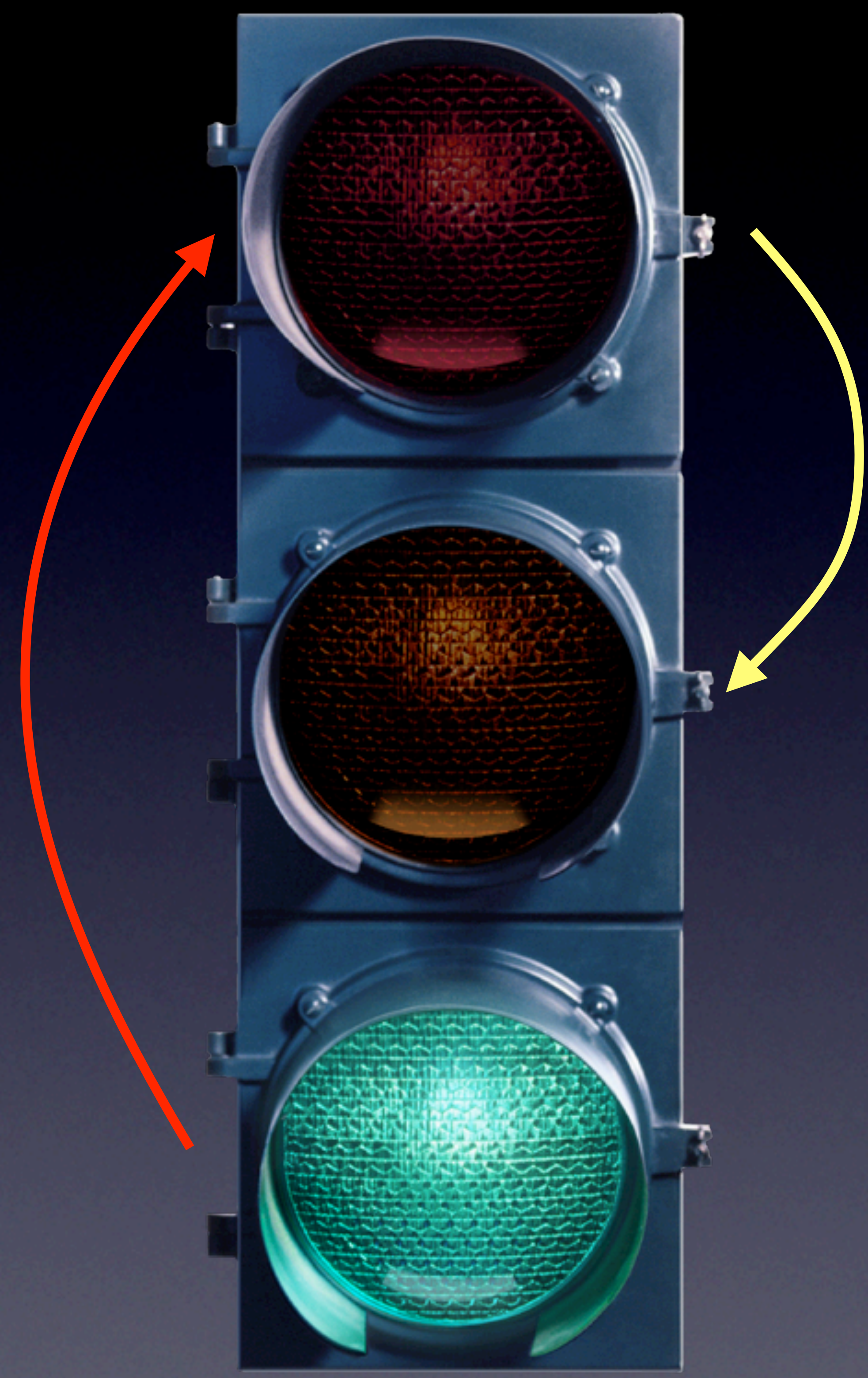
process
completes

process
is dispatched
by scheduler



preemption

process
suspended



process
resumed

EDF Algorithm

- The earliest-deadline-first (EDF) scheduling algorithm always dispatches at any time instant a ready process action with a relative deadline (e.g. process period) that is earlier than the relative deadline of any other ready process action
- EDF is a dynamic priority assignment algorithm

Optimality

- A scheduling algorithm *A* is <u>optimal</u> with respect to a property *S* (e.g. schedulability) if *A* always determines a schedule that satisfies *S* provided some schedule that satisfies *S* exists
- EDF is optimal with respect to schedulability but requires <u>preemption</u>

EDF Test

- The standard utilization-based schedulability test for EDF is:

$$\sum_P \lambda_P / \pi_P \leq 1$$

- The test returns true if and only if each process P may invoke, every π_P time instants, an EDF-dispatched process action with at most λ_P execution time within at most π_P response time

Precision

- A schedulability test is <u>sufficient</u> if a positive test result implies schedulability (required)
- A schedulability test is <u>necessary</u> if schedulability implies a positive test result (optional)
- The utilization-based schedulability test for EDF is sufficient and necessary but only works for <u>periodic</u> processes

Scheduling & Schedulability

- Scheduling algorithms control the <u>access</u> of processes to processors
  - Time and space complexity should be constant, or proportional to the number of processes (p) and distinct time instants (t)
- Schedulability tests control the <u>admission</u> of processes into the system
  - Complexity should be similar to above

Scheduling & Admission

- Scheduling requires queue management:
  - <u>insert</u> process into ready queue
  - <u>select</u> process from ready queue
- Admission requires resource management:
  - <u>admit</u> process into system

Complexity

	list	tree	array
insert	$O(n)$	$O(\log n)$	$O(1)$
select	$O(1)$	$O(\log n)$	$O(n)$
admit	$O(1)$		

process queue: $n = p$ (processes)
timeline queue: $n = t$ (time instants)

Performance vs. Predictability

- Frequency of scheduler invocations:
  - Conflict between throughput and latency
- Execution time of each scheduler invocation:
  - Upper bound, lower bound, variance (jitter)
  - Conflict between low variance and low bounds (optimizations that work for all inputs are difficult)

Predictability

1. A non-functional, quantifiable property of a process action (such as its response time) is **predictable** if its quantity can be bounded in terms of other, known quantities
2. Such a property is more predictable than another if the **prediction effort** is less and the **prediction accuracy** is higher than for the other property

Effort and Accuracy

1. The prediction effort should be proportional to the bounding quantities, or even constant
2. The prediction accuracy should be conservative, or even exact

Example

- Action response time is $(0, \pi_P]$ if

$$\sum_P \lambda_P / \pi_P \leq 1$$

- Constant-time effort for admission
- Actual response times may vary by at most π_P (bad for large π_P)

Compositionality

1. A component model is compositional with respect to some quantifiable, non-functional property (such as action response times) if, for any system composed in the model, the respective quantities in the system's components do not change when composed.
2. Such a model is more compositional than another if the composition effort is less and the composition accuracy is higher than for the other model.

Example

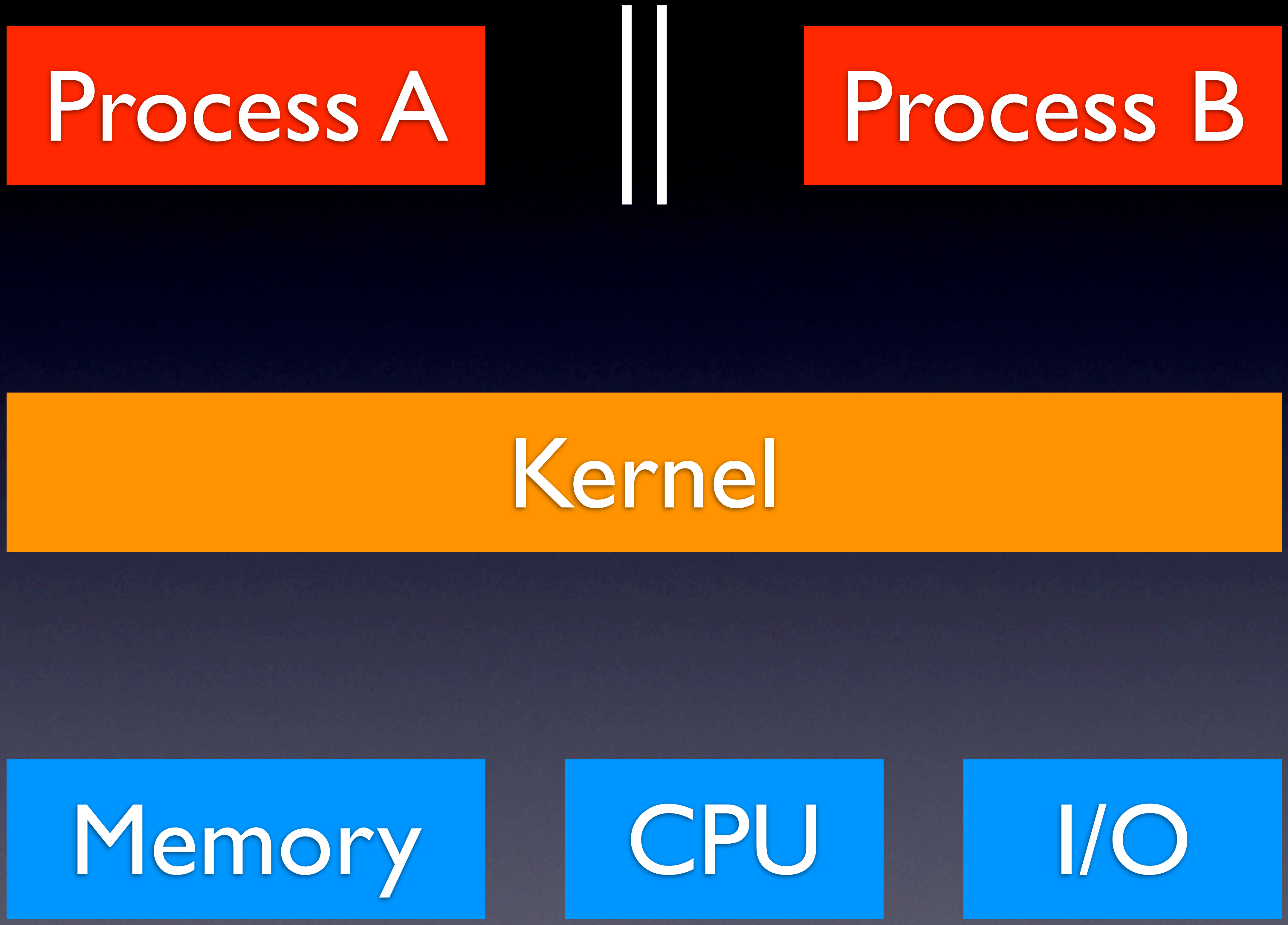
- Set of periodic processes:
  - Existing processes still meet deadlines even when adding/removing processes
- Giotto program:
  - Existing Giotto processes maintain input and output times even when adding/removing Giotto processes

Application and Resources

application	kernel	resource
processes	*compositionality*	processors/ memory
concurrency		distribution
response times	*predictability*	execution times
frequencies		timers

Outline

1. Introduction
2. Process Model
3. Concurrency Management
4. Memory Management
5. I/O Management


Process A
||
Process B
Kernel
Memory
CPU
I/O

Tiptoe Process Model

- Tiptoe processes invoke process actions
- Process actions are system calls and procedure calls but also just code, which may have optional workload parameters
- Workload parameters determine the amount of work involved in executing process actions

Example

- Consider a process that reads a video stream from a network connection, compresses it, and stores it on disk, all in real time
- The process periodically adapts the frame rate, allocates memory, receives frames, compresses them, writes the result to disk, and finally deallocates memory to prepare for the next iteration

Pseudo Code

```
loop {
  int number_of_frames = determine_rate();

  allocate_memory(number_of_frames);
  read_from_network(number_of_frames);

  compress_data(number_of_frames);

  write_to_disk(number_of_frames);
  deallocate_memory(number_of_frames);
} until (done);
```

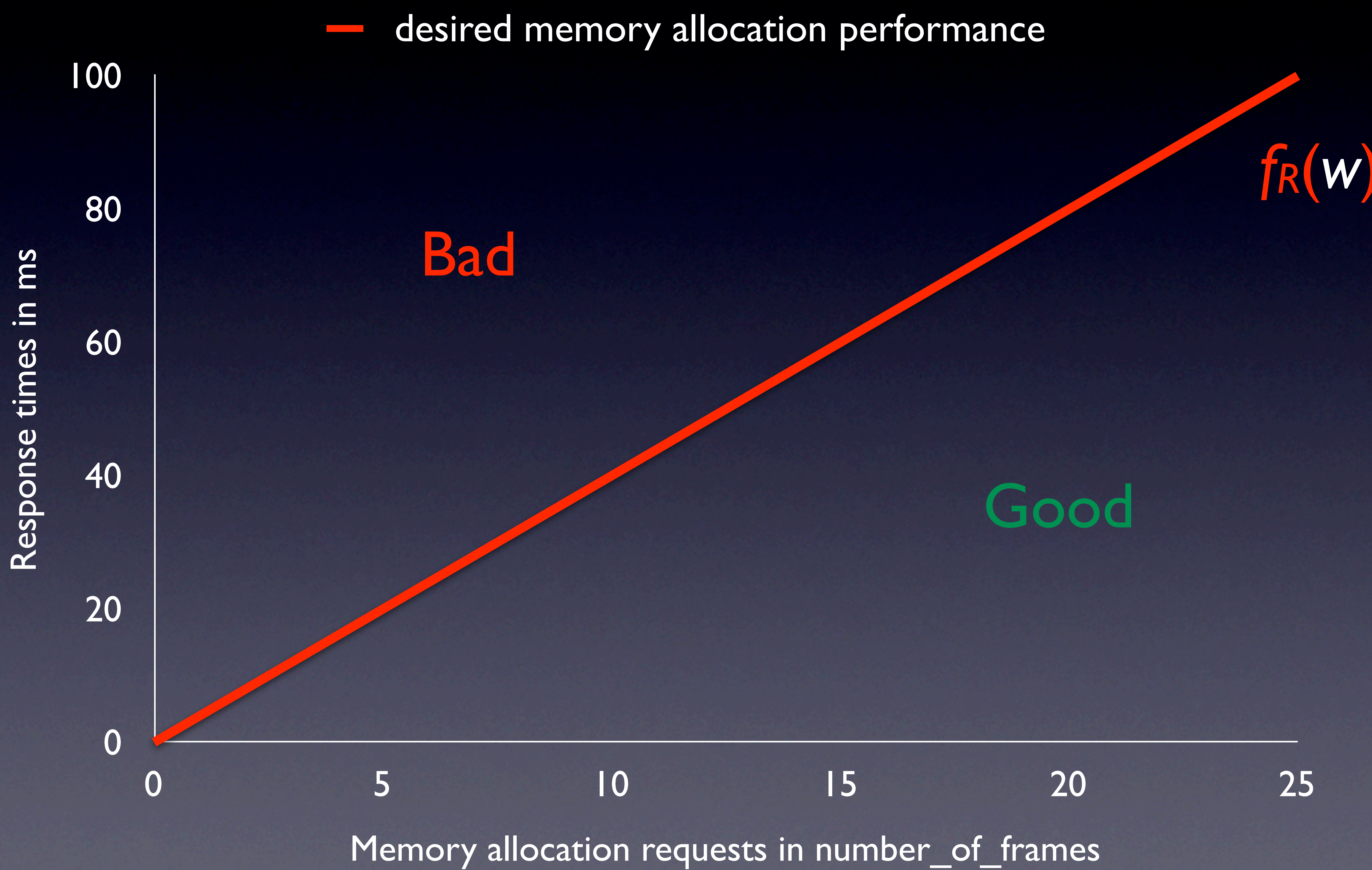
Tiptoe Programming Model

- Process actions are characterized by their execution time and response time in terms of their workload parameters
- The execution time is the time it takes to execute an action in the absence of concurrent activities
- The response time is the time it takes to execute an action in the presence of concurrent activities

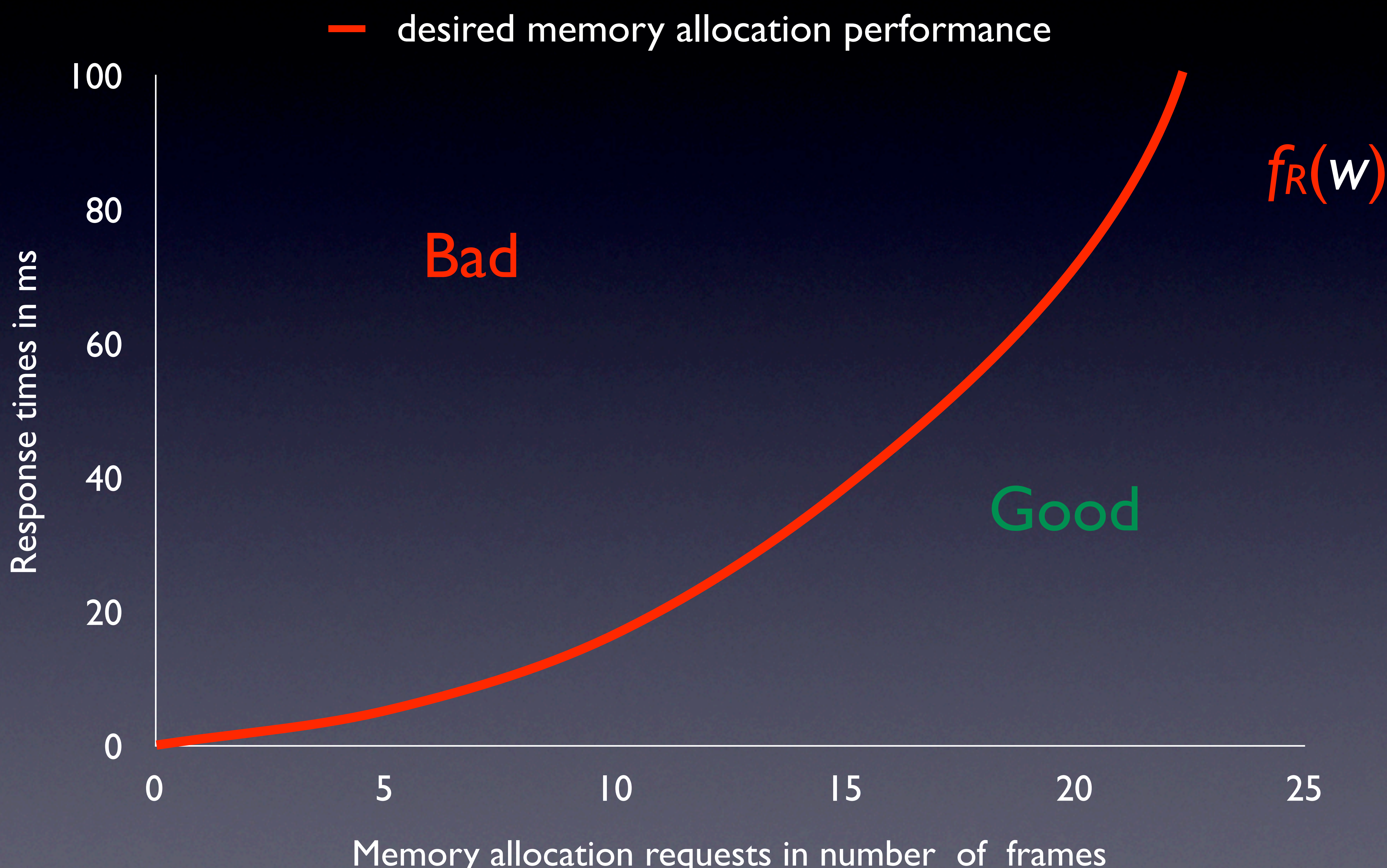
Compositionality

- System of Tiptoe processes:
  - The individual actions of running Tiptoe processes maintain their response times even when adding/removing processes

Response-Time Function



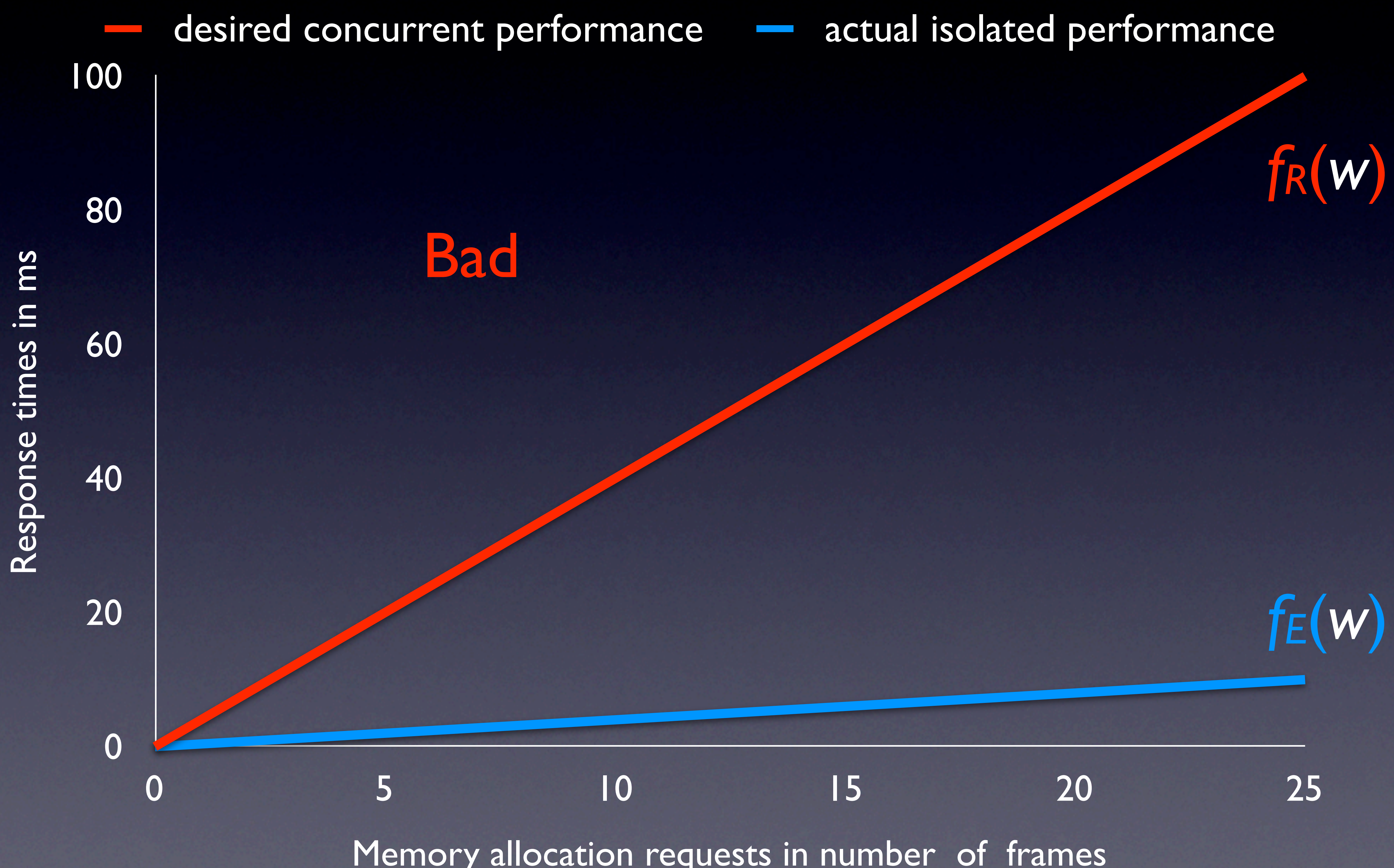
Compositional Response!



A response-time (RT) function is a discrete function

$$f_{\mathrm{R}} : N \rightarrow Q^{+}$$

Execution-Time Function



An execution-time (ET) function is a discrete function

$$f_E : E_D \rightarrow Q^+$$

with $E_D \subseteq N$

E_D is the action's execution domain

Utilization Function:

$$f_U(w) = \frac{f_E(w)}{f_R(w)}$$

With

$$f_R(w) = 4 * w \text{ (in ms)}$$
$$f_E(w) = 0.4 * w \text{ (in ms)}$$

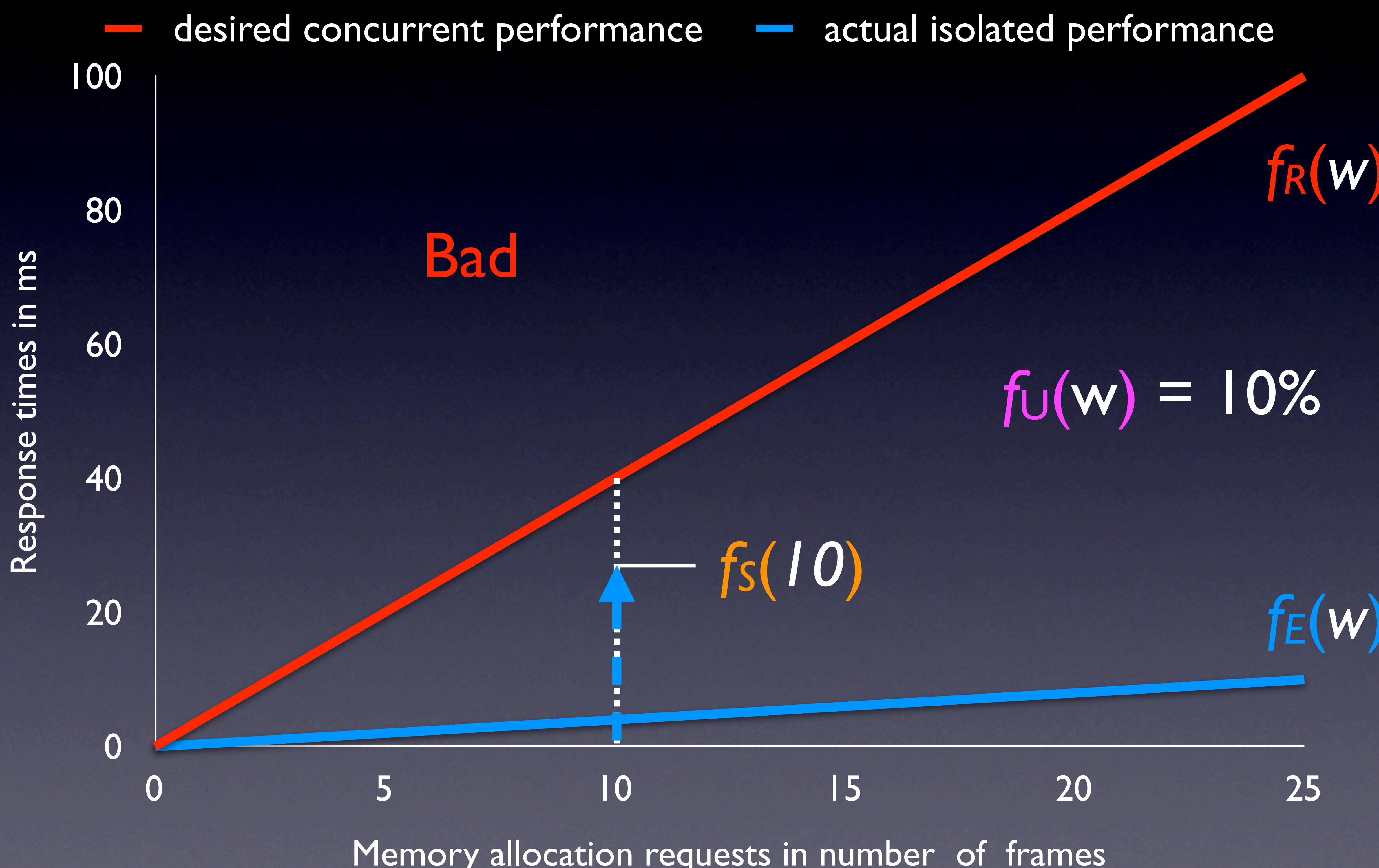
we have that

$$f_U(w) = 10\% \text{ (for } w>0\text{)}$$

Outline

1. Introduction
2. Process Model
3. Concurrency Management
4. Memory Management
5. I/O Management

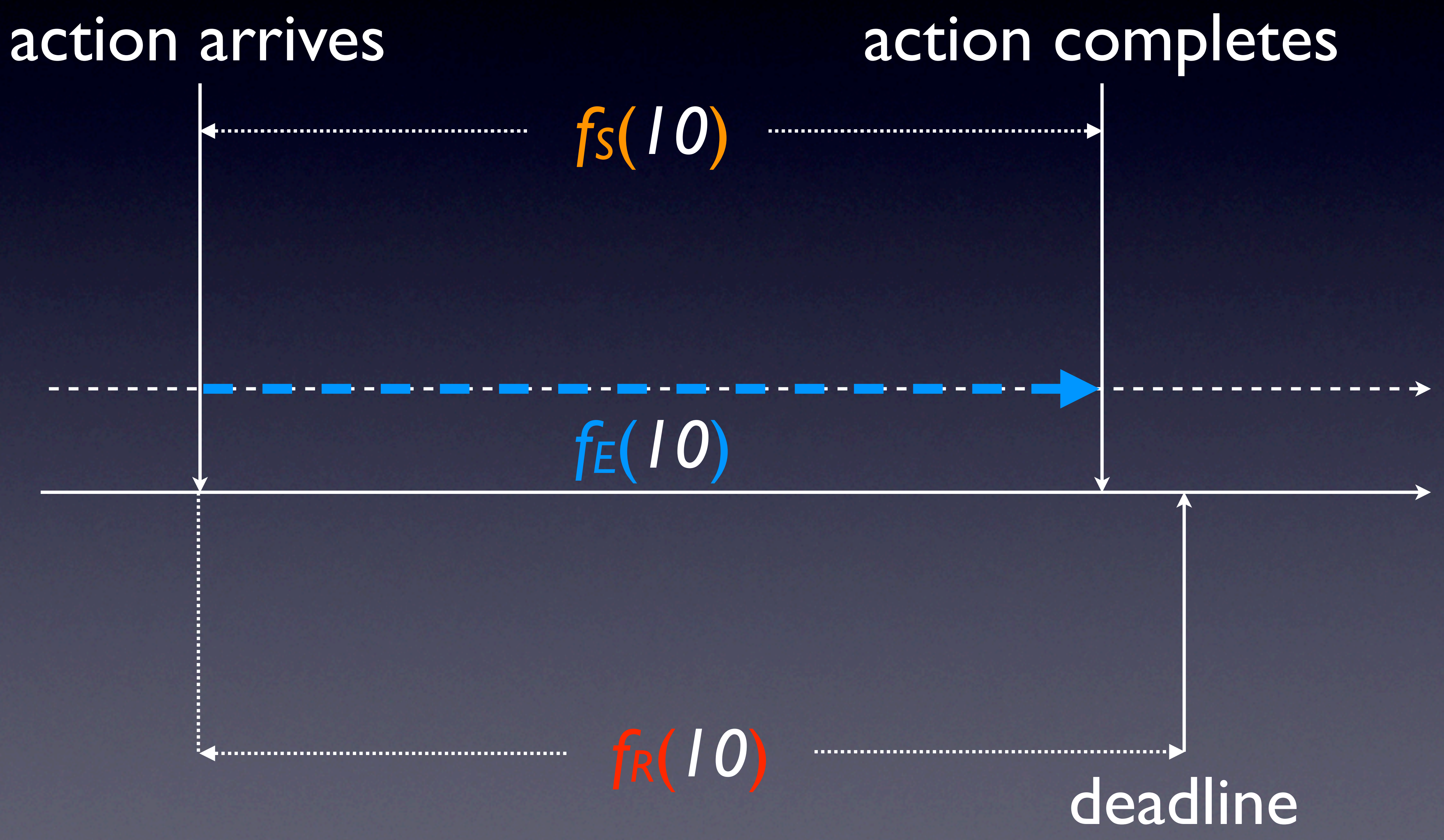
Scheduled Response Time



Scheduling and Admission

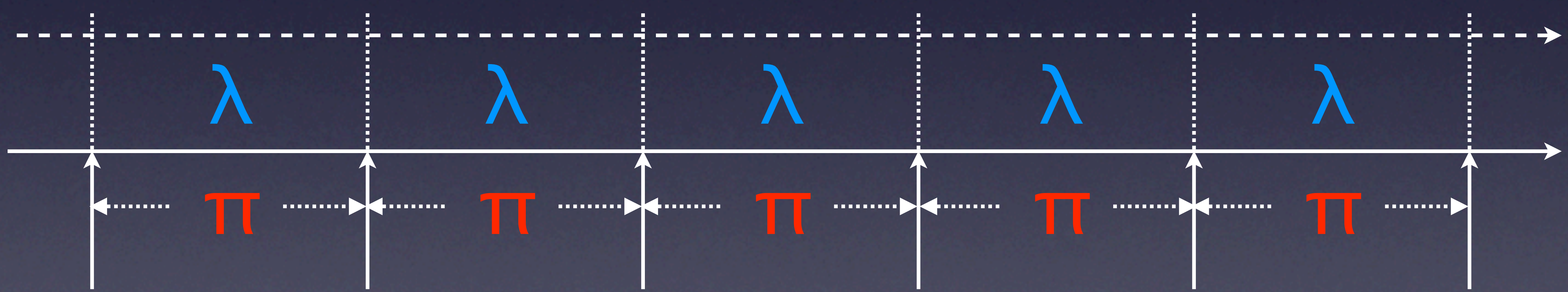
- Process scheduling:
  - How do we efficiently schedule processes on the level of individual process actions?
- Process admission:
  - How do we efficiently test schedulability of newly arriving processes

Just use EDF, or not?



Virtual Periodic Resource

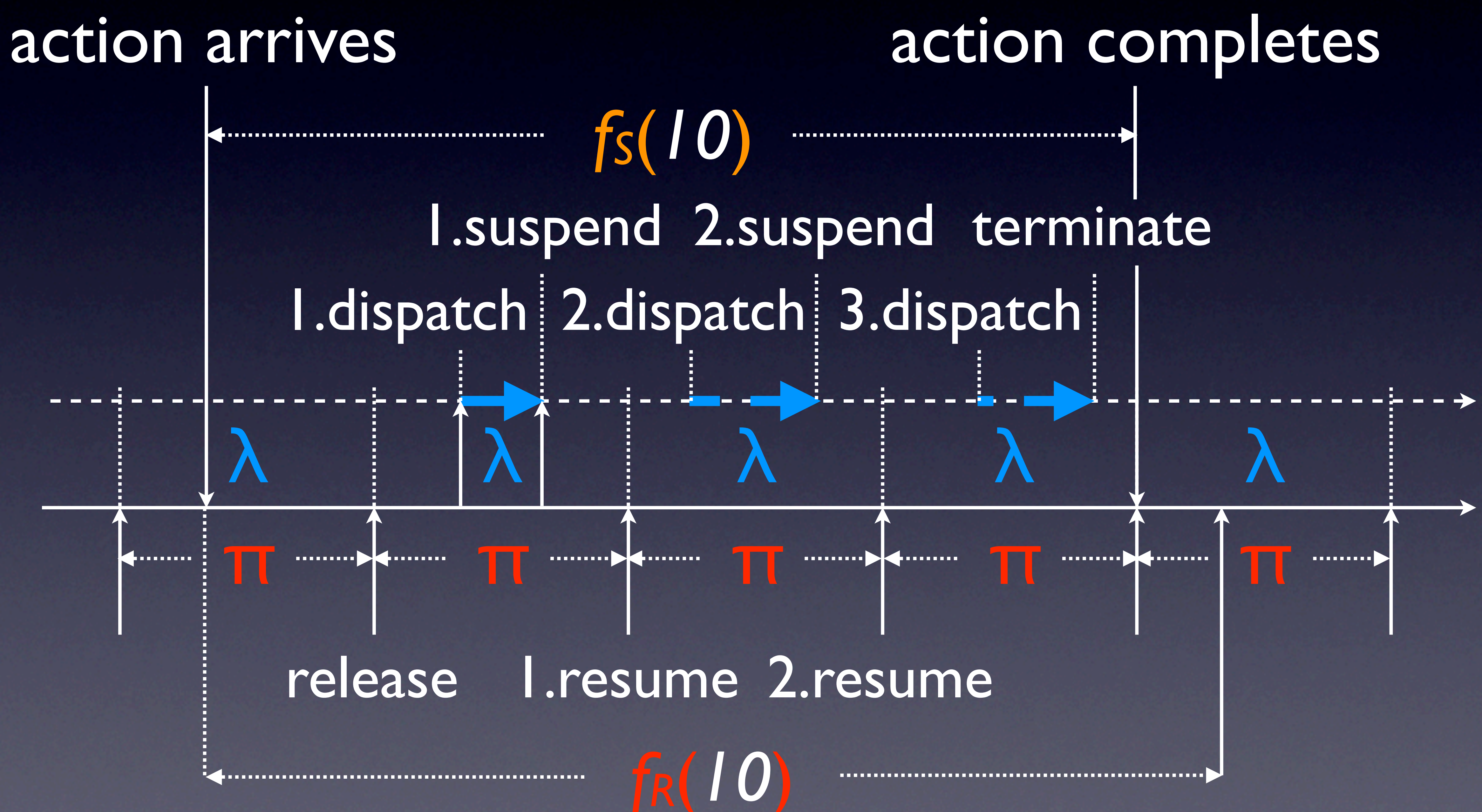
limit: λ
period: π
utilization: λ / π



Tiptoe Process Model

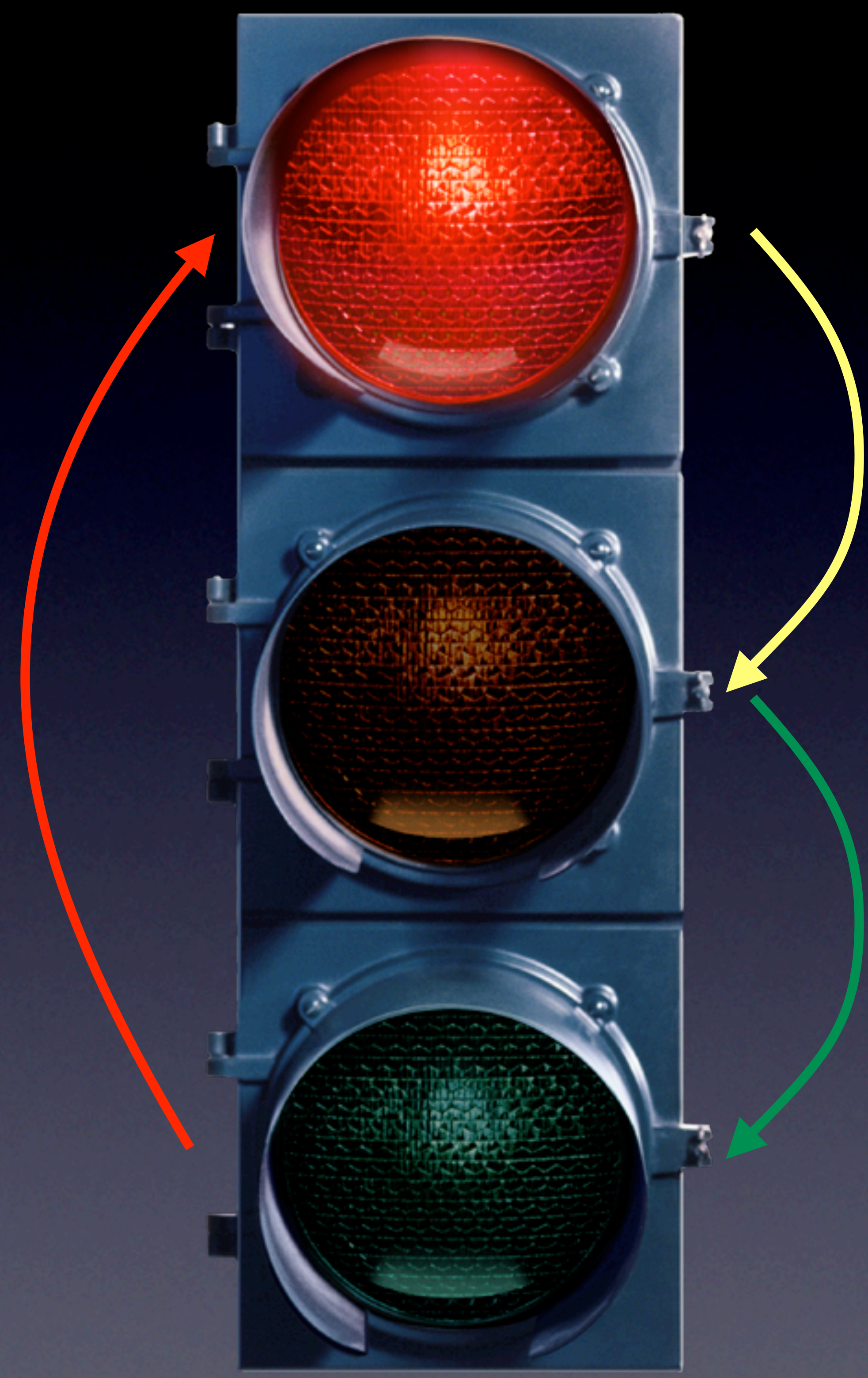
- Each Tiptoe process declares a finite set of virtual periodic resources
- Each process action of a Tiptoe process uses exactly one virtual periodic resource declared by the process

Release, Dispatch, Suspend, Resume, Terminate

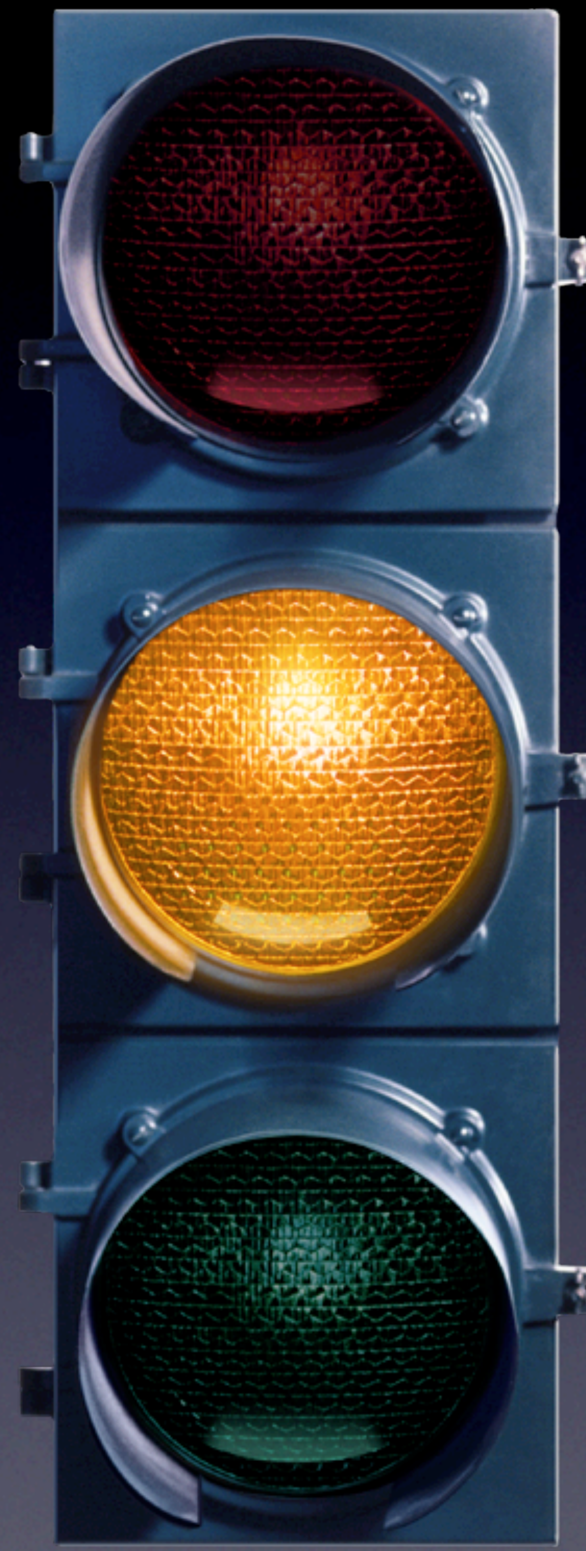


Scheduling Strategies

- release action upon arrival at the beginning of next period (release strategy)
- dispatch released actions in EDF order using periods as deadlines (dispatch strategy)
- suspend running actions when limit is exhausted and resume at beginning of next period (limit strategy)
- terminate completed actions at the end of next period (termination strategy)

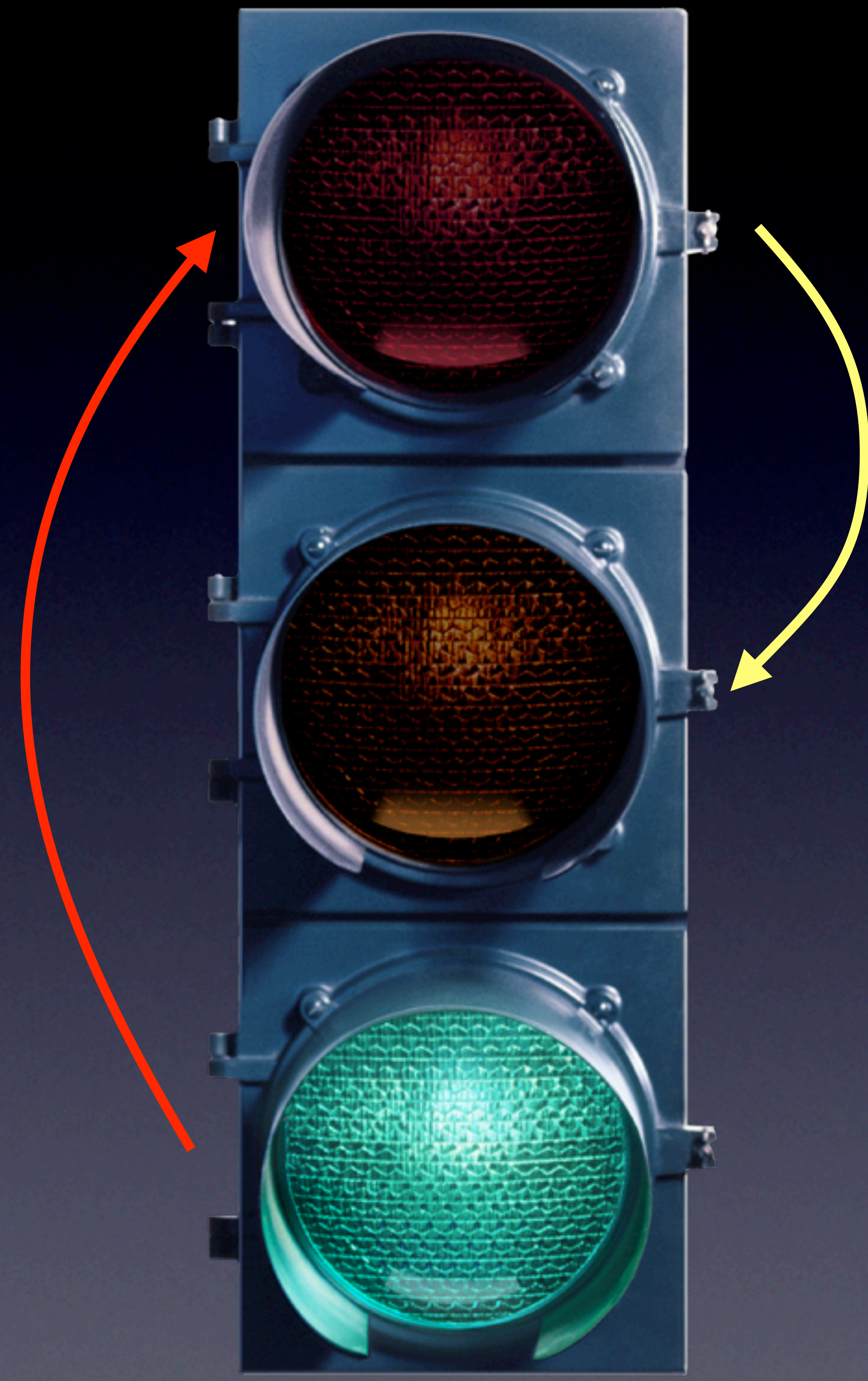

arrival
dispatch
completion

release strategy



dispatch
strategy

limit
strategy



limit
strategy

termination strategy

$$\forall w \in E_D.\ f_S(w) \leq f_R(w)\ ?$$

$$\forall w \in E_D.$$

$$\pi_a * \lceil f_E(w) / \lambda_a \rceil$$

$$\leq f_S(w) \leq$$

$$(\pi_a - 1) + \pi_a * \lceil f_E(w) / \lambda_a \rceil$$

if

$$\sum_P \max_R(\lambda_{PR}/\pi_{PR}) \leq 1$$

$$\forall w \in E_D.$$

$$0$$

$$\leq f_S(w) - \pi_a * \lceil f_E(w) / \lambda_a \rceil \leq$$

$$\pi_a - 1$$

if

$$\sum_P \max_R(\lambda_{PR}/\pi_{PR}) \leq 1$$

Tiptoe Compositionality

$$\forall fs, fs'. \forall w \in E_D.$$

$$0 \leq | fs(w) - fs'(w) | \leq \pi_a - 1$$

if

$$\sum_P \max_R(\lambda_{PR}/\pi_{PR}) \leq 1$$

$$\forall w \in E_D.\ f_S(w) \leq f_R(w)\ ?$$

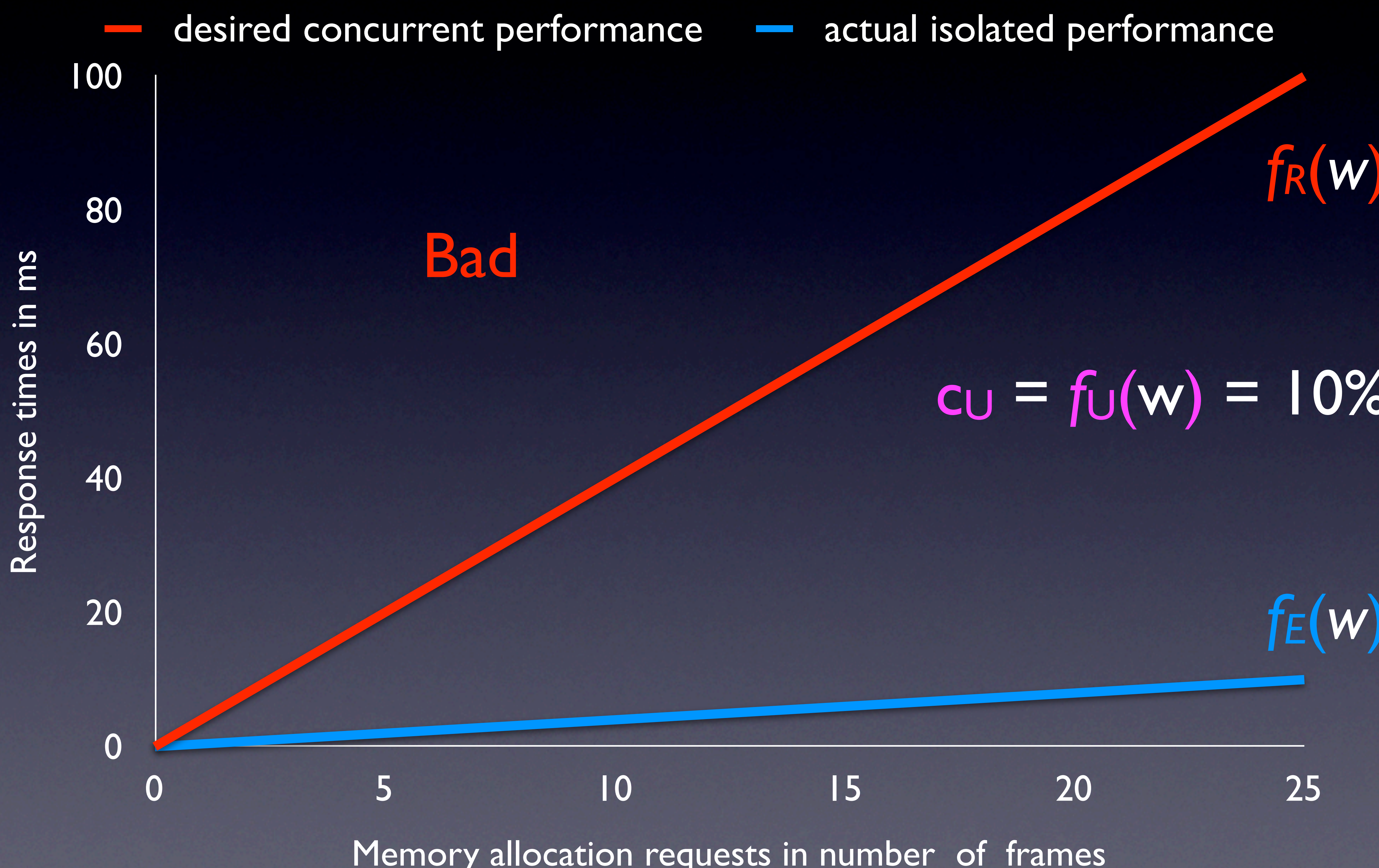
A set of workloads $U_D \subseteq E_D$ is a utilization domain if there is a constant $0 \leq c_U \leq 1$ s.t.

$$\forall w \in U_D.\ f_U(w) \leq c_U$$

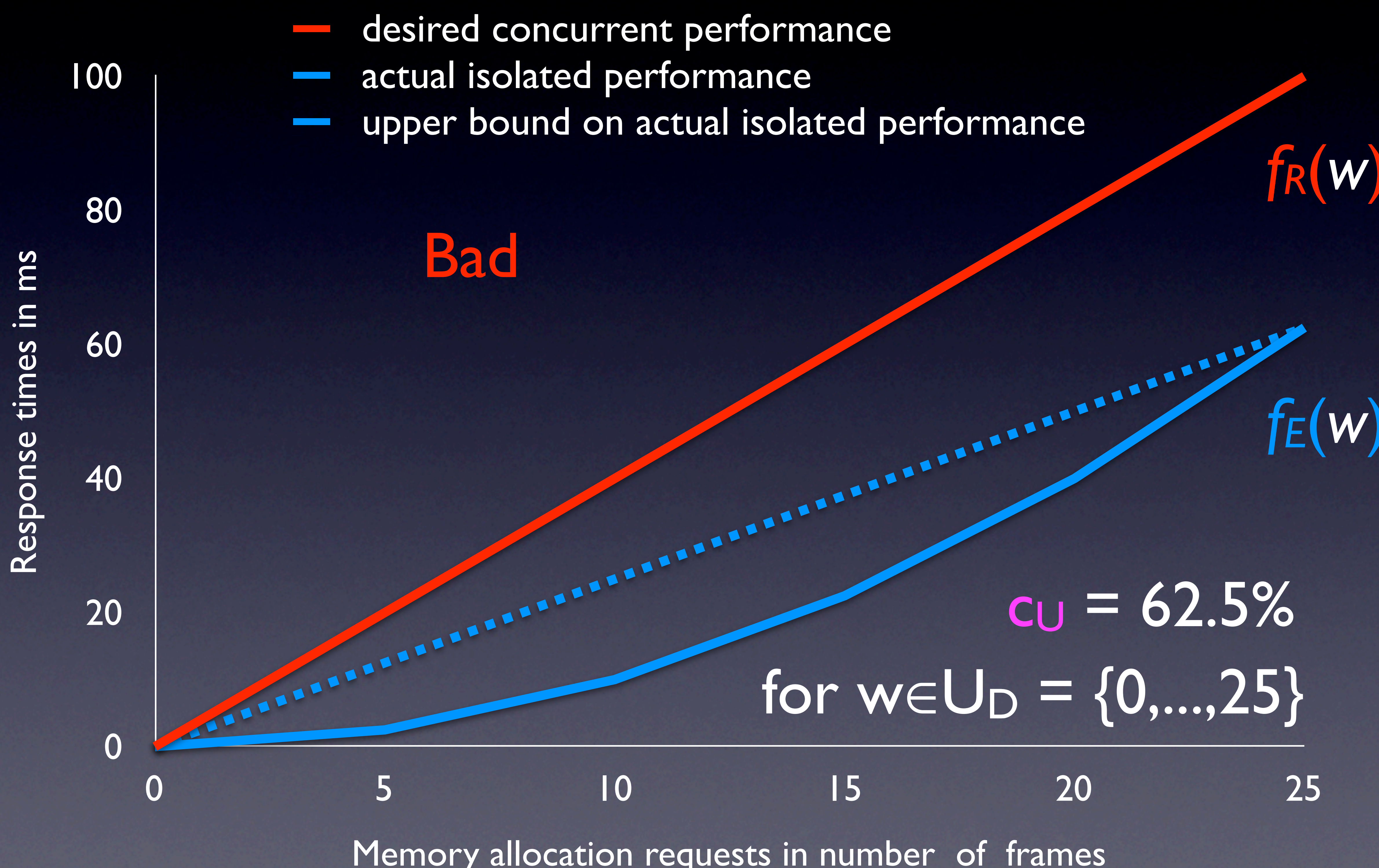
and

$$\forall c \leq c_U. \exists w \in U_D.\ c \leq f_U(w)$$

Infinite Utilization Domain



Finite Utilization Domain



With λ_a / π_a = c_U, we know that

$\forall w \in U_D.\ f_S(w) \leq f_R(w) + \pi_a$

if

π_a divides $f_R(w)$ evenly

and

$\sum_P \max_R(\lambda_{PR}/\pi_{PR}) \leq 1$

For example,
for linear discrete functions

$f(w) = n * w$

we have that

π_a divides $f(w)$ evenly

if and only if

π_a divides n evenly

$$\forall w \in U_D.\ f_S(w) \leq f_R(w) + \pi_a$$

For example, with

$f_R(w) = 4 * w + 4$ (in ms)

$f_E(w) = 0.4 * w + 0.2$ (in ms)

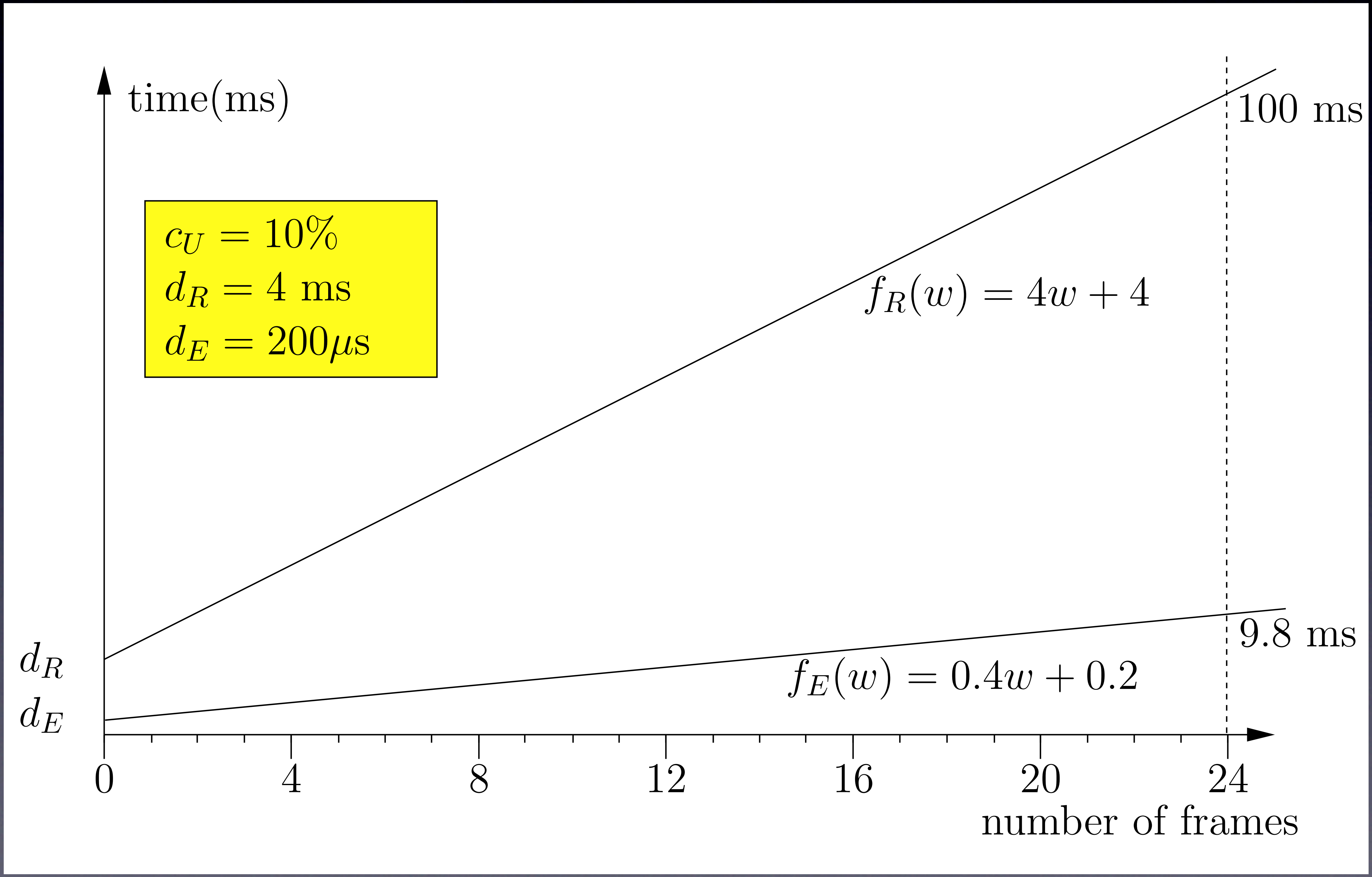
we have again

$f_U(w) = 10\%$ (for $w>0$)

$f_R(1) = 8$ms but only 125fps

$f_R(24) = 100$ms yet 240fps

Intrinsic Delay



Since

$$\forall w \in N.\ f_R(w) > 0$$

there is a unique $w_d \in N$ s.t.

$$\forall w \in N.\ f_R(w) \geq f_R(w_d)$$

$f_R(w_d)$ is the <u>intrinsic response delay</u> denoted by d_R

Since

$\forall w \in E_D.\ f_E(w) > 0$

there is a unique $w_d \in E_D$ s.t.

$\forall w \in E_D.\ f_E(w) \geq f_E(w_d)$

$f_E(w_d)$ is the intrinsic execution delay denoted by d_E

Utilization Function:

$$f_U(w) = \frac{f_E(w) - d_E}{f_R(w) - d_R}$$

(if $f_R(w) > d_R$)

With λ_a / π_a = c_U, we know that

$\forall w \in U_D.\ f_S(w) \leq f_R(w)$

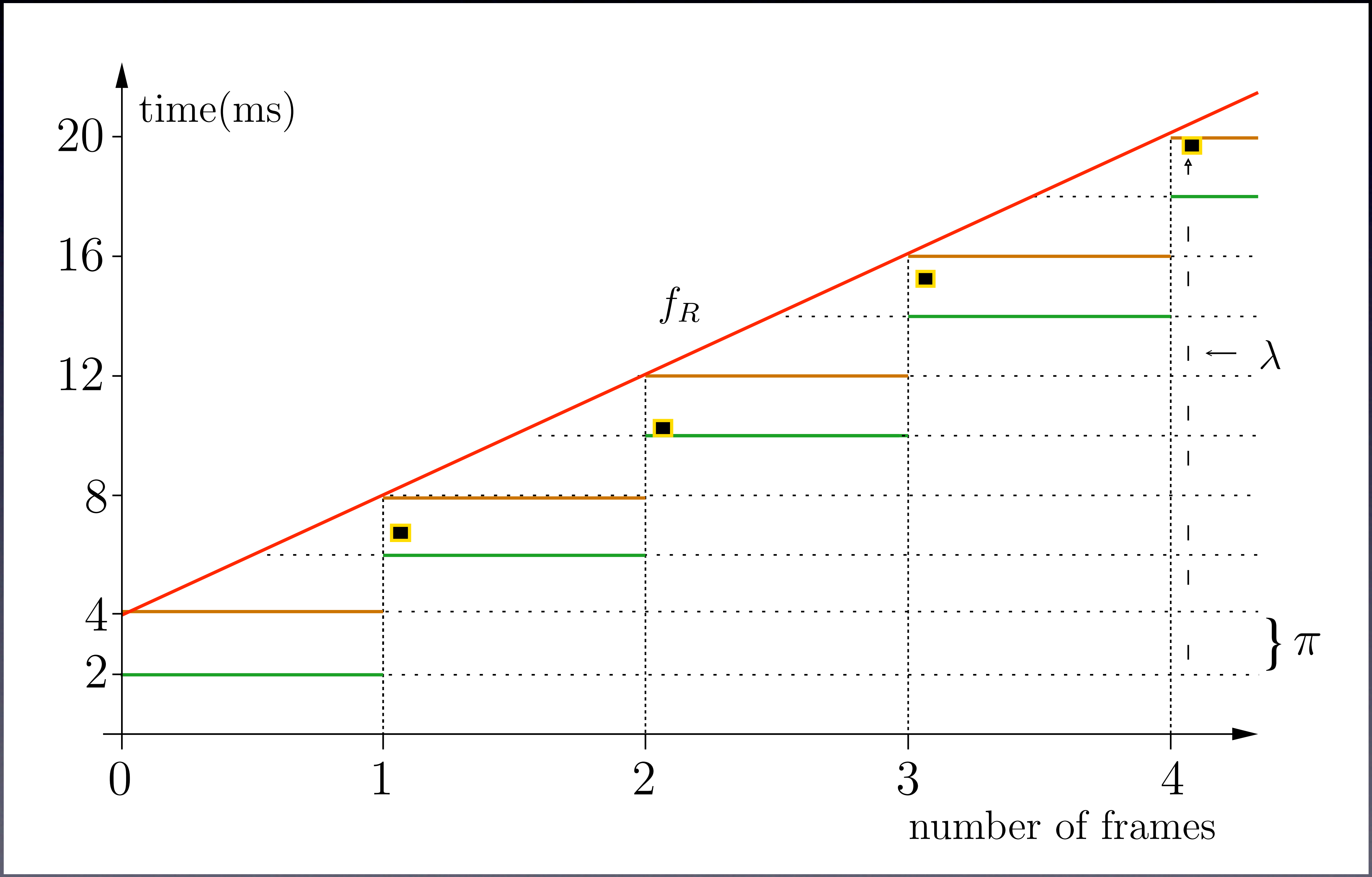
if

$0 < \pi_a \leq d_R - d_E / c_U$, and

π_a divides d_R and $f_R(w) - d_R$ evenly,

and $\sum_P \max_R(\lambda_{PR} / \pi_{PR}) \leq 1$

Scheduler



time(ms)
20
16
12
8
4
2
f_R
$\leftarrow \lambda$
$\} \pi$
0
1
2
3
4
number of frames

Scheduling Algorithm

- maintains a queue of ready processes ordered by deadline and a queue of blocked processes ordered by release times
- ordered-insert processes into queues
- select-first processes in queues
- release processes by moving and sorting them from one queue to another queue

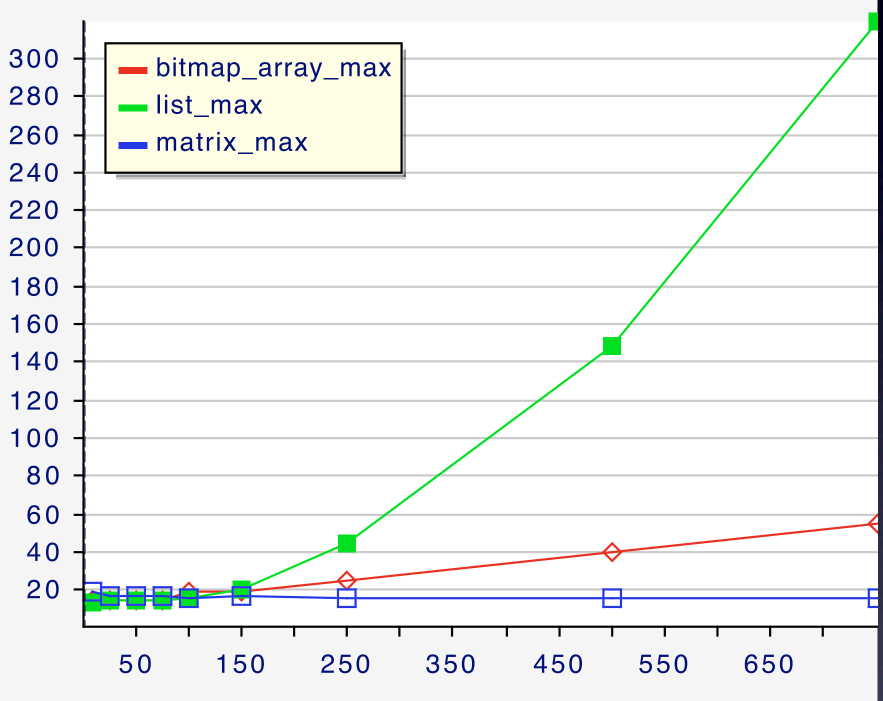
Time and Space

	list	array	matrix
ordered-insert	$O(n)$	$\Theta(\log(t))$	$\Theta(\log(t))$
select-first	$\Theta(1)$	$O(\log(t))$	$O(\log(t))$
release	$O(n^2)$	$O(\log(t) + n \cdot \log(t))$	$\Theta(t)$

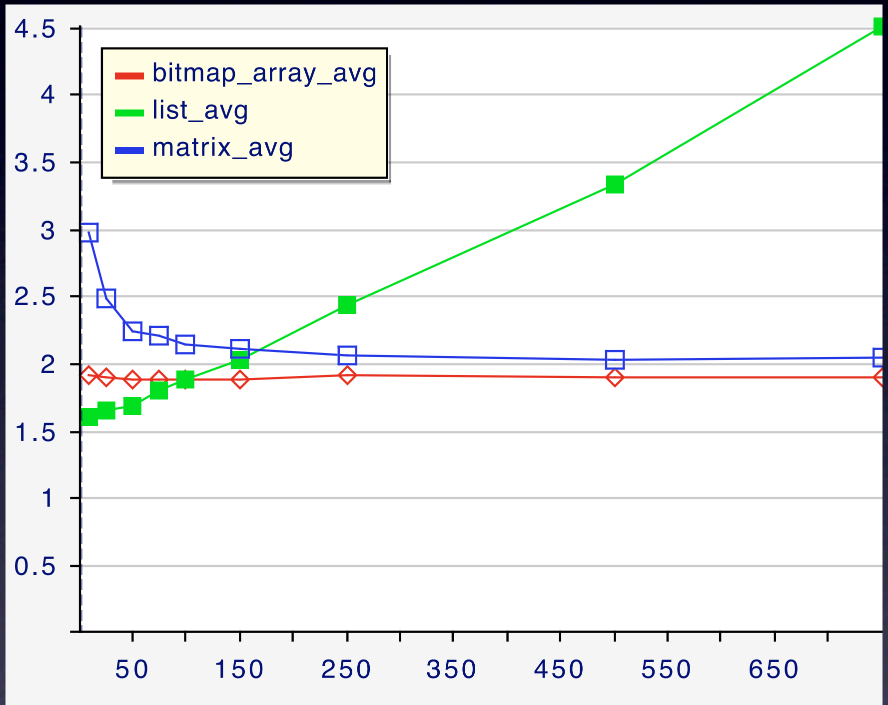
	list	array	matrix
time	$O(n^2)$	$O(\log(t) + n \cdot \log(t))$	$\Theta(t)$
space	$\Theta(n)$	$\Theta(t + n)$	$\Theta(t^2 + n)$

n: number of processes t: number of time instants

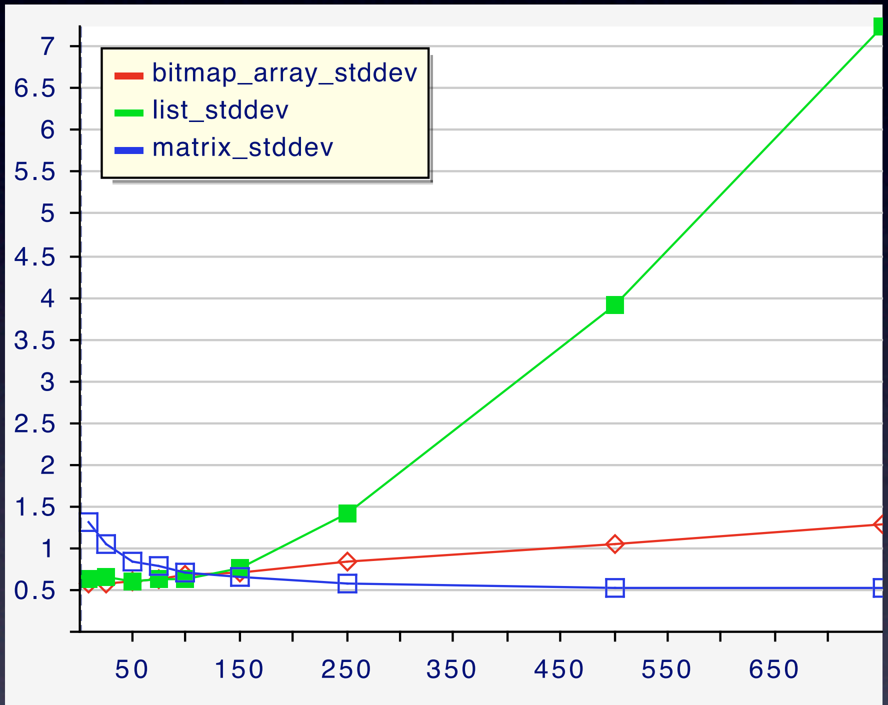
Scheduler Overhead



Max

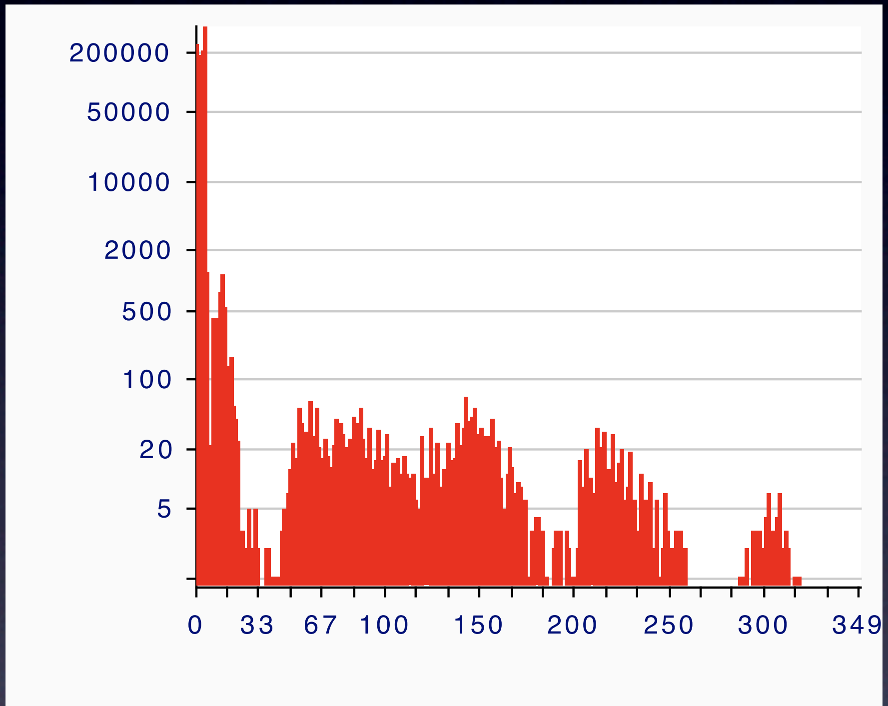


Average

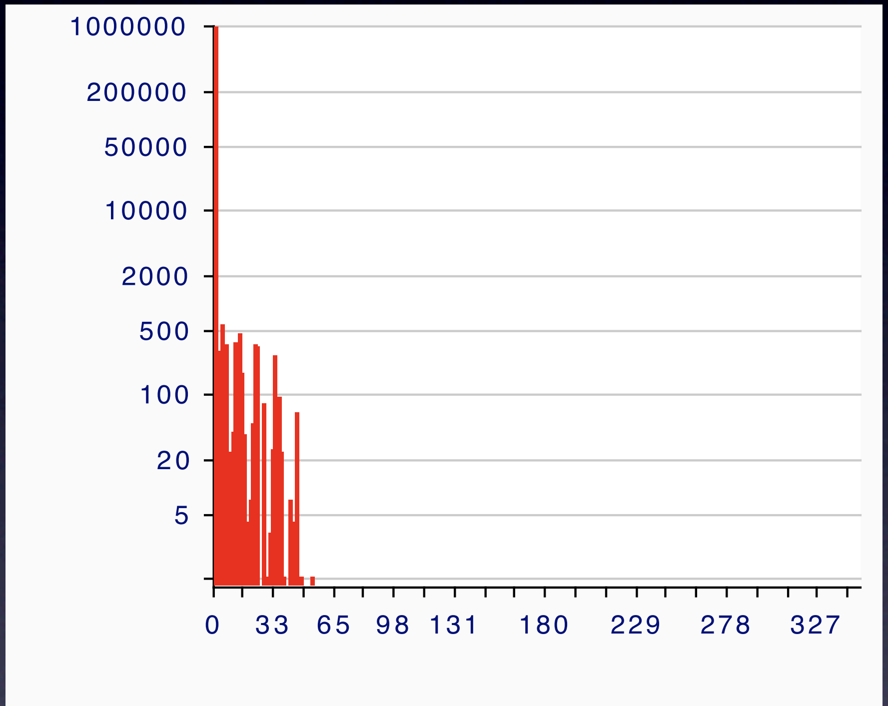


Jitter

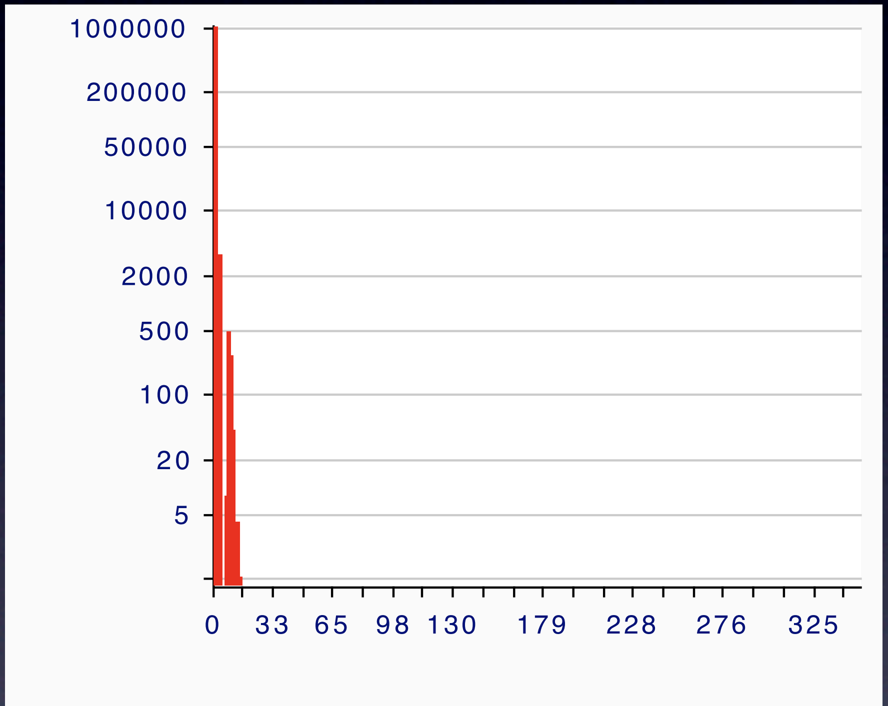
Execution Time Histograms



List

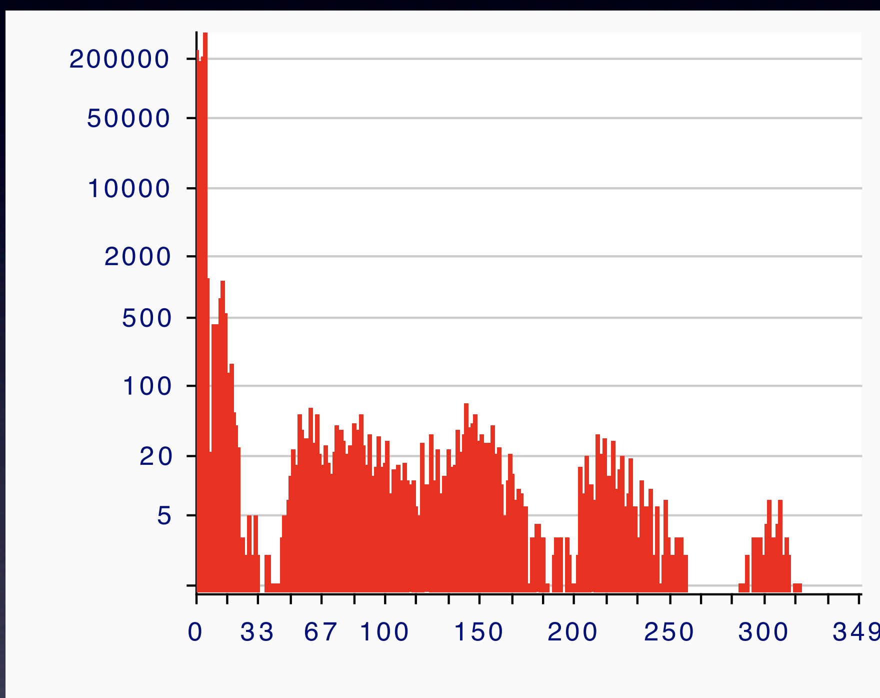


Array

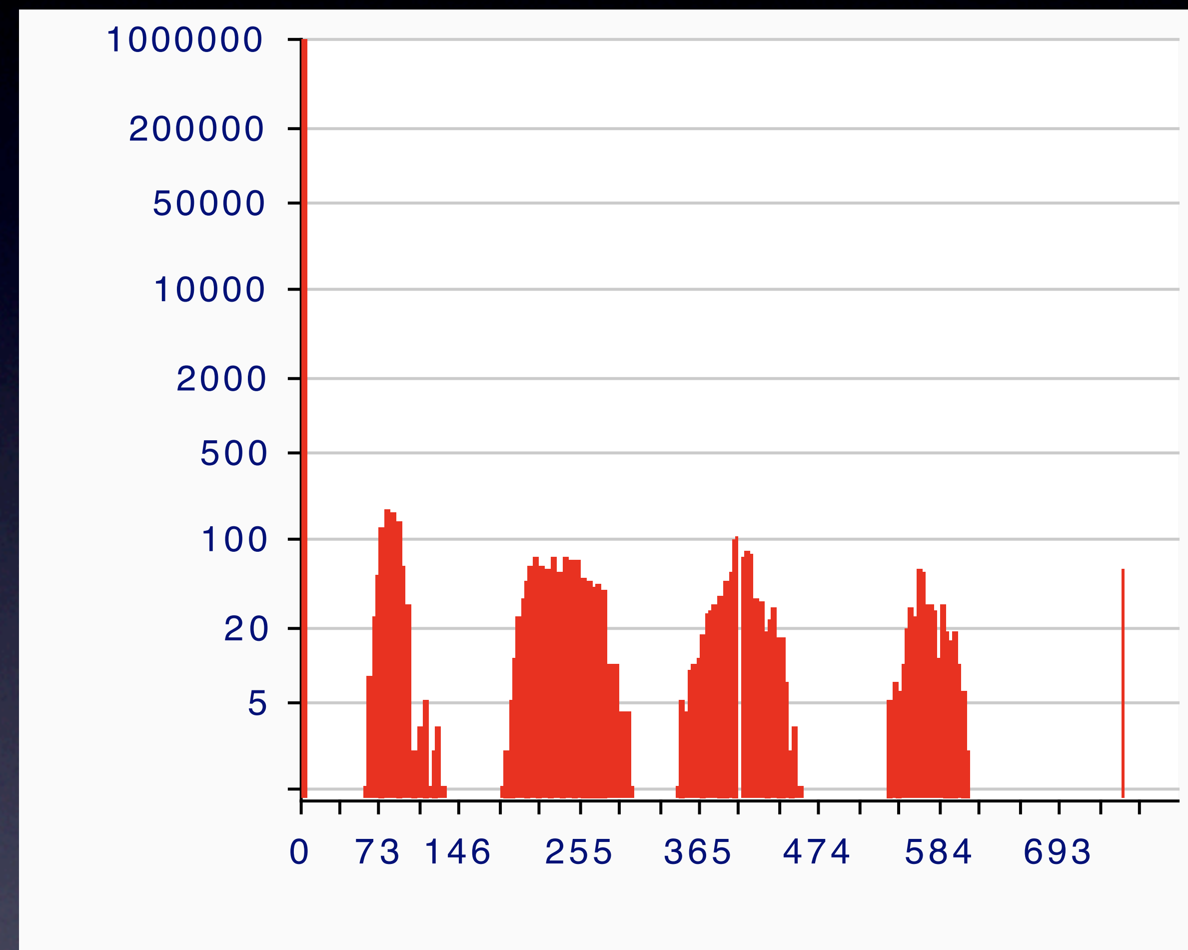


Matrix

Process Release Dominates

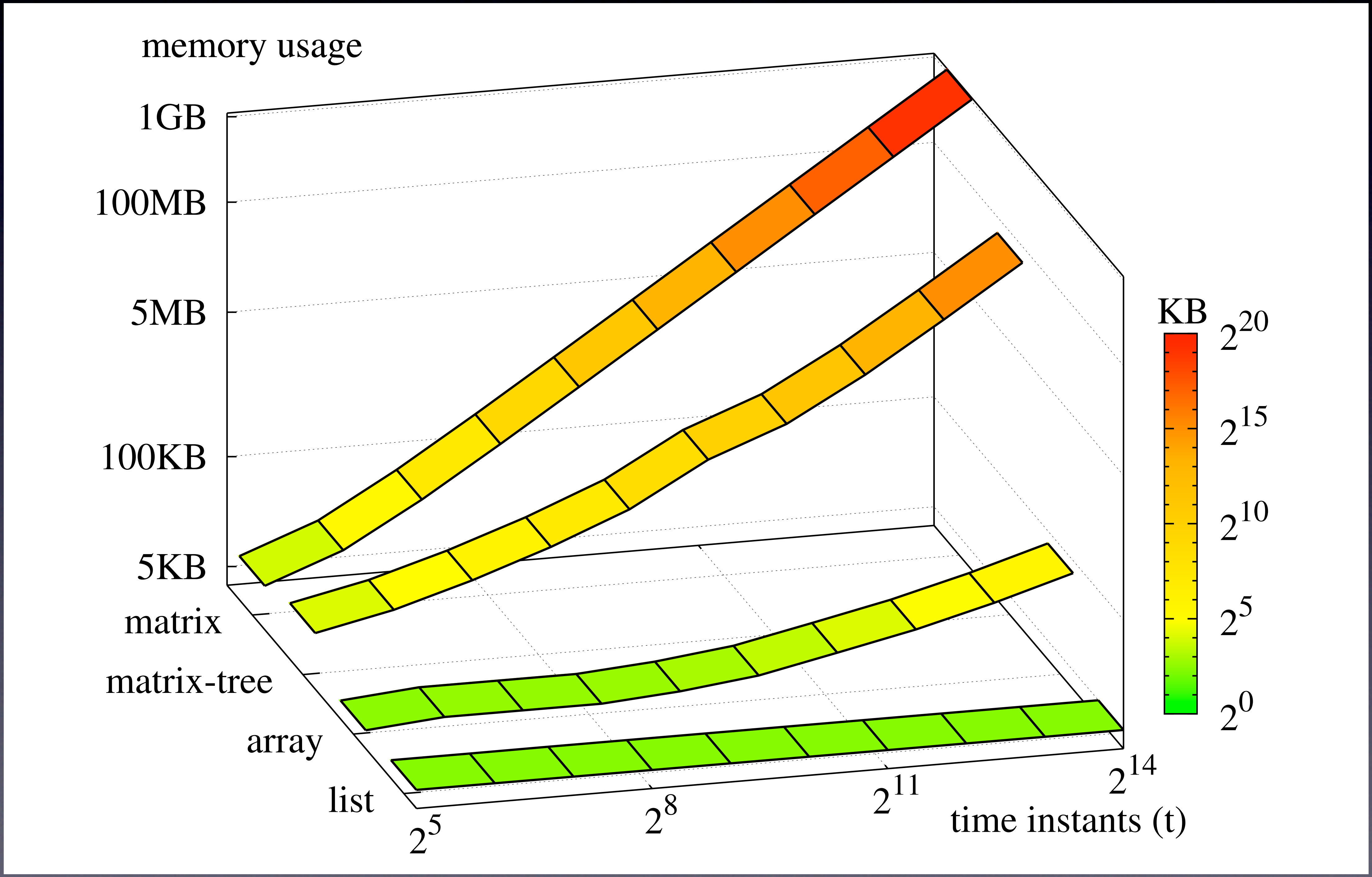


List

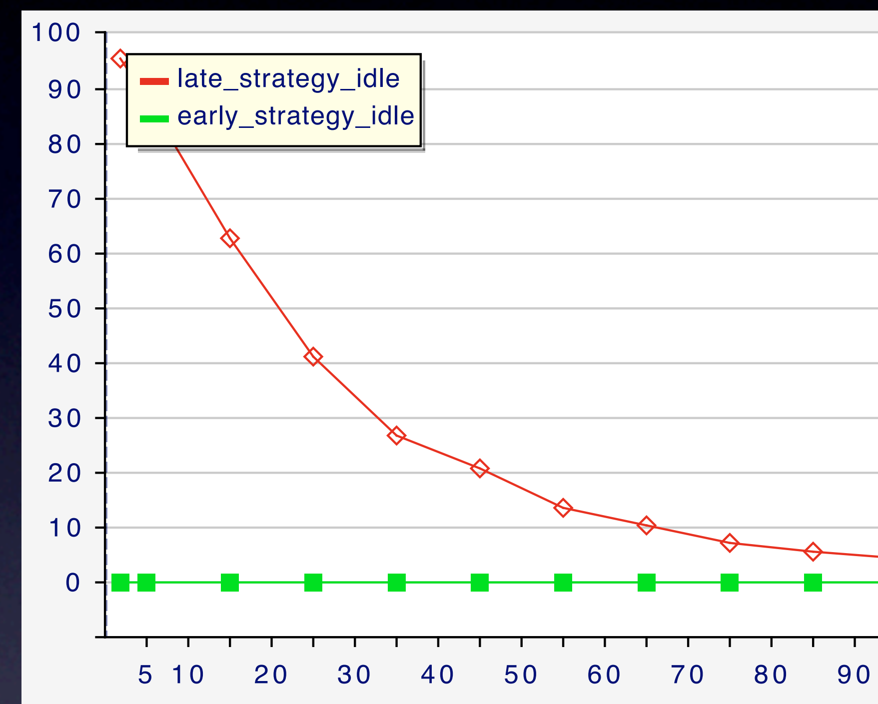


Releases per Instant

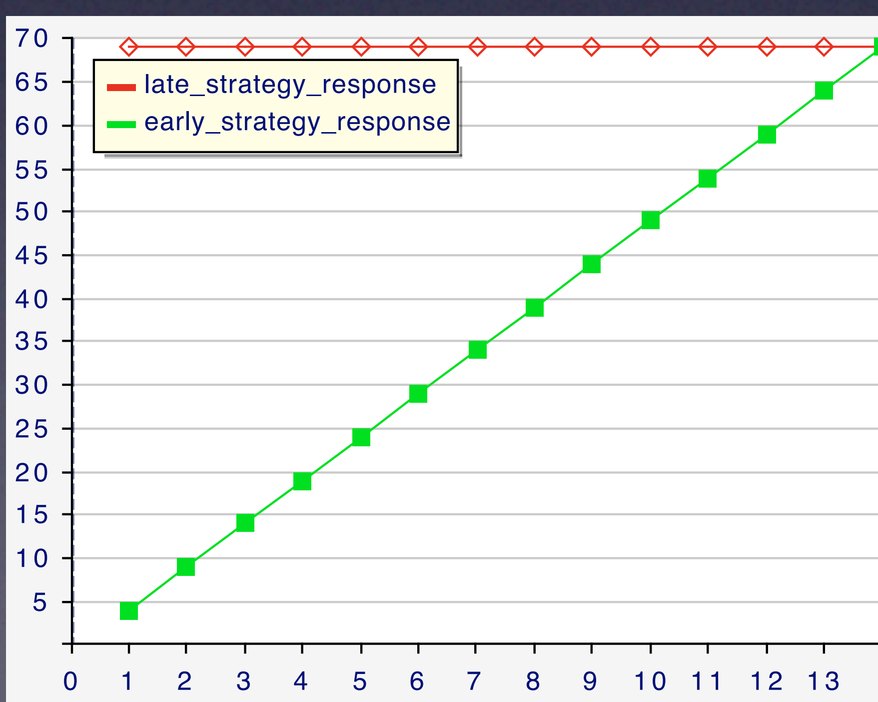
Memory Overhead



Release Strategies



Idle Time



Response Time

Outline

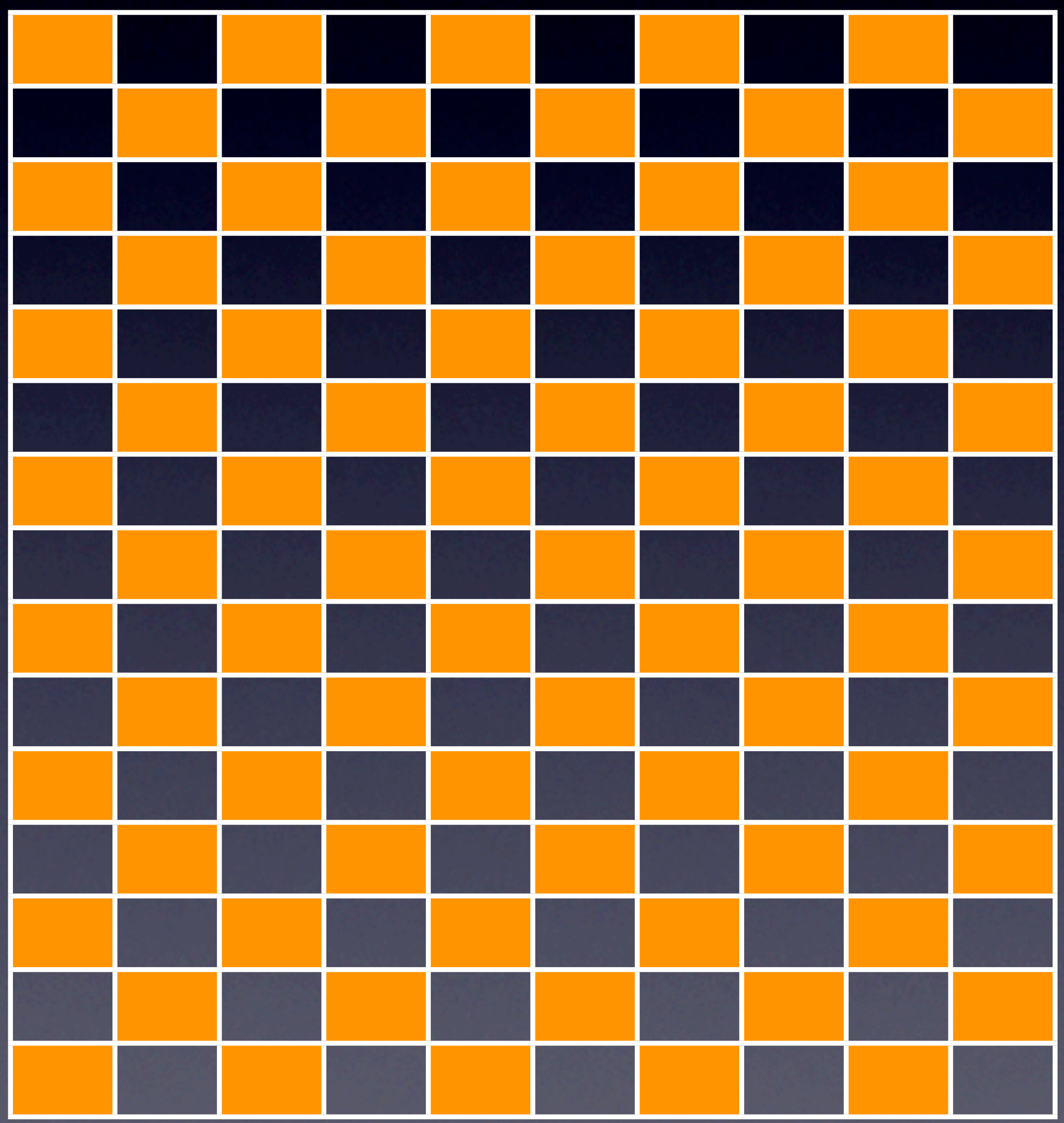
1. Introduction
2. Process Model
3. Concurrency Management
4. Memory Management
5. I/O Management

What We Want

- malloc(n) takes at most TIME(n)
- free(n) takes at most TIME(n)
- access takes small constant time

- small and predictable memory fragmentation bound

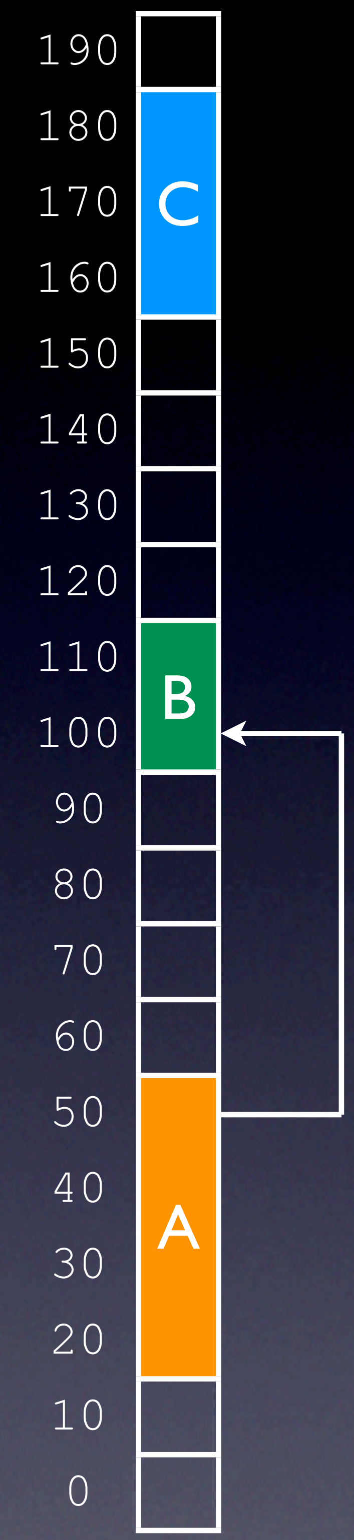
The Problem



- Fragmentation
  - Compaction
- References
  - Abstract Space

Example:

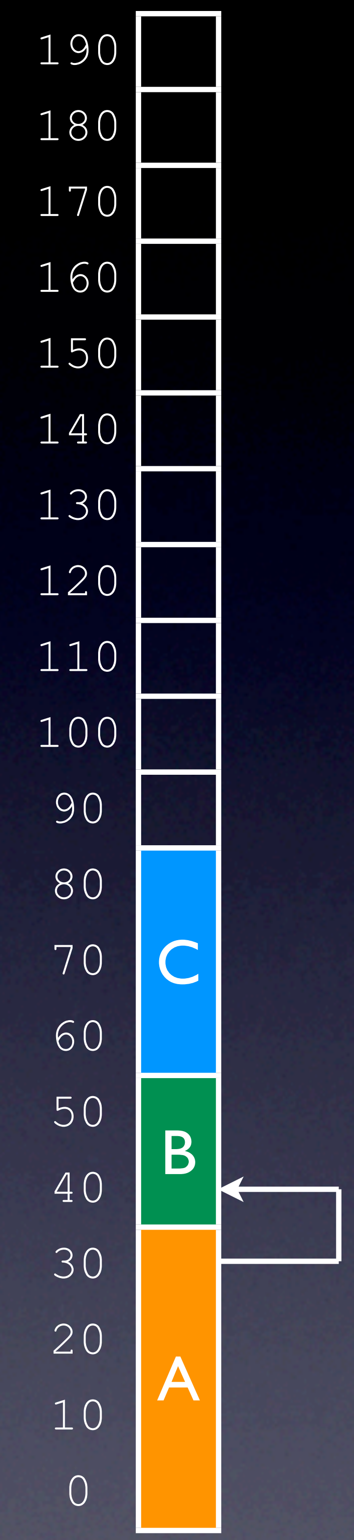
- There are three objects
- Object A starts at address 20
- Object A needs 40 bytes
- B starts at 100, needs 20 bytes
- C starts at 160, needs 30 bytes
- A contains a reference to B



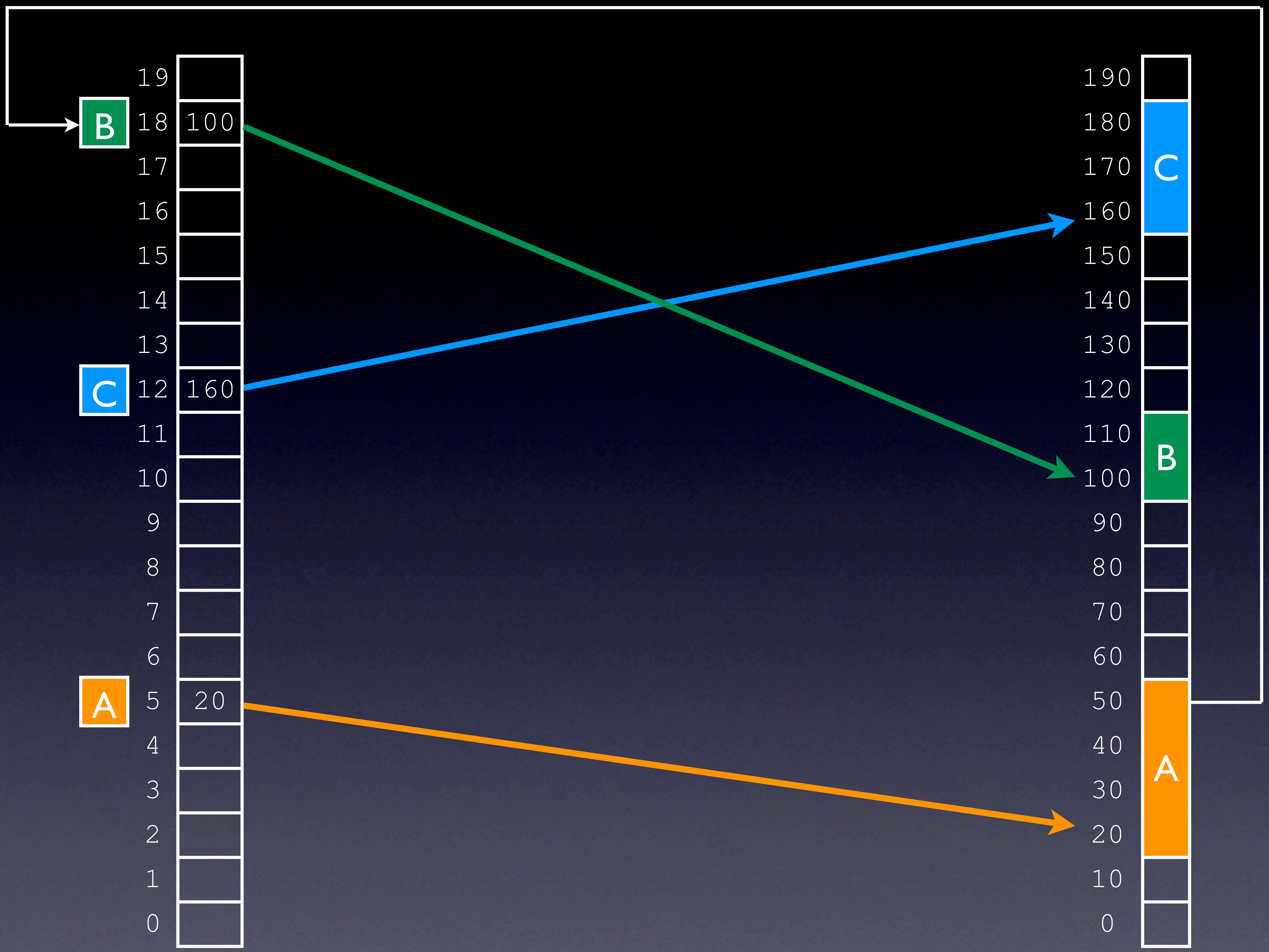
Memory

Problem:

- The addresses of objects change
- Now A starts at address 0
- B at address 40, C at address 60
- The reference to B requires update

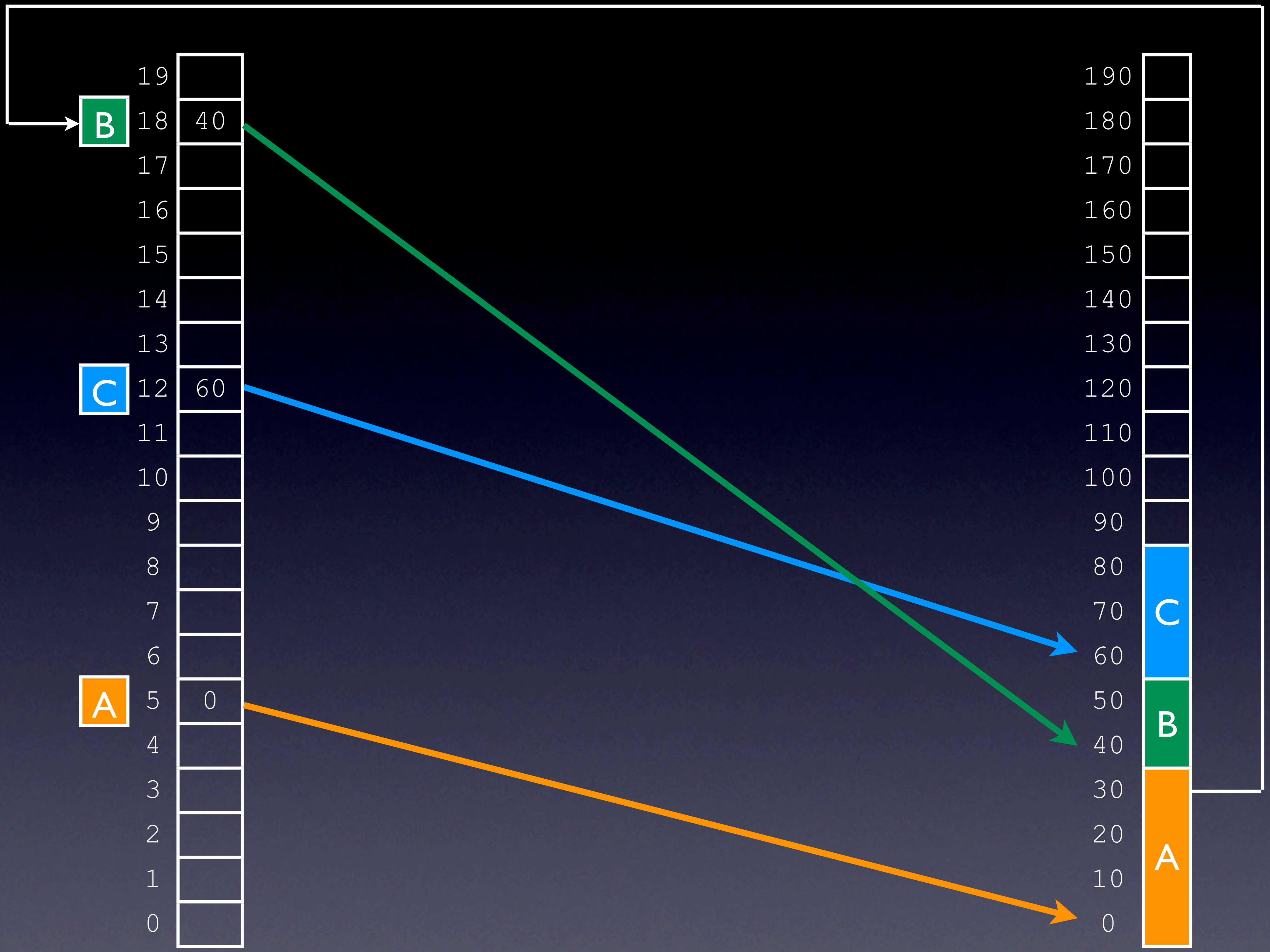


Memory



Abstract Space

Concrete Space



Abstract Space

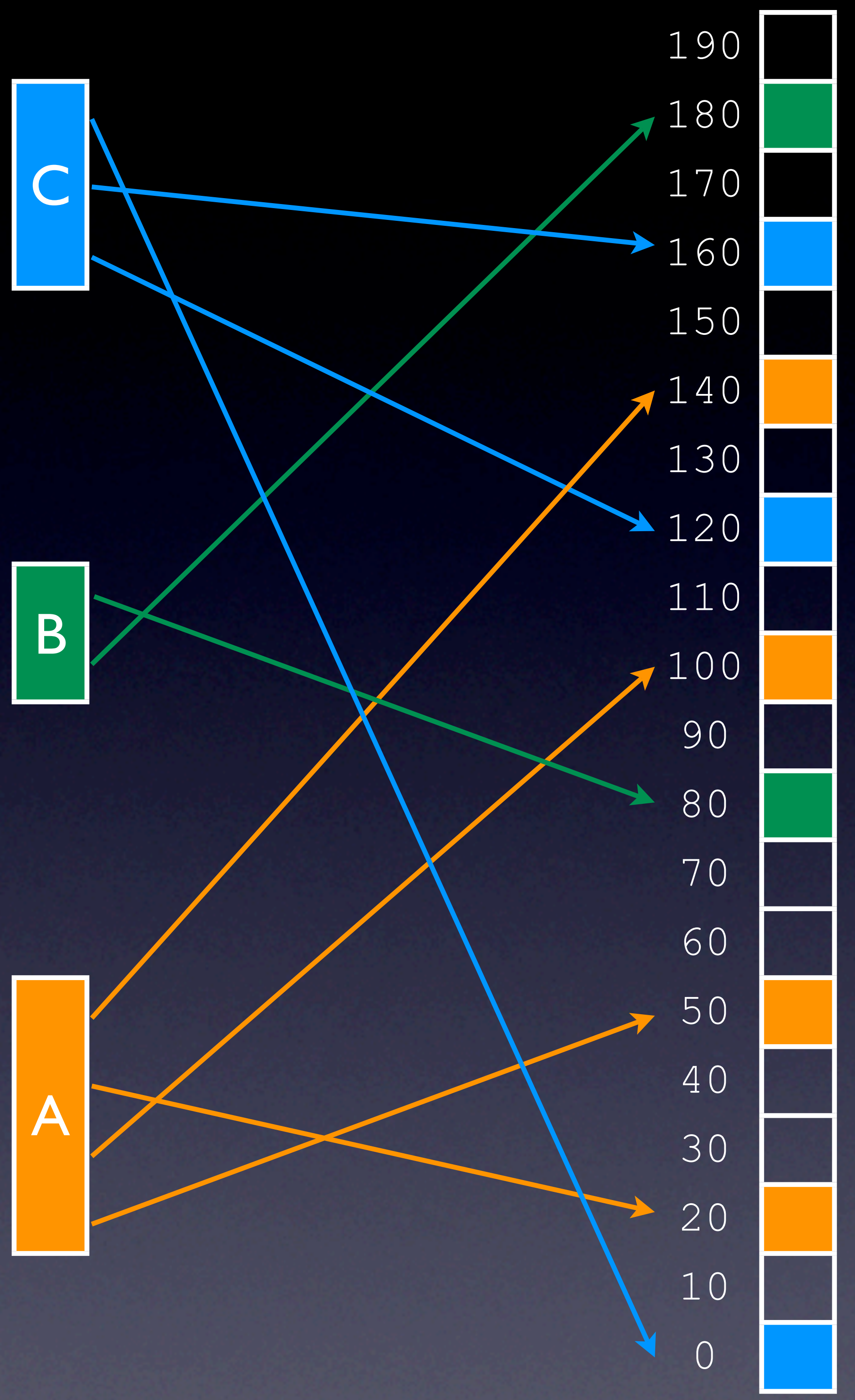
Concrete Space

Constant Access Time

- constant access times require contiguous space
- contiguous space gets fragmented over time
- non-contiguous space does not get fragmented but results in non-constant access times

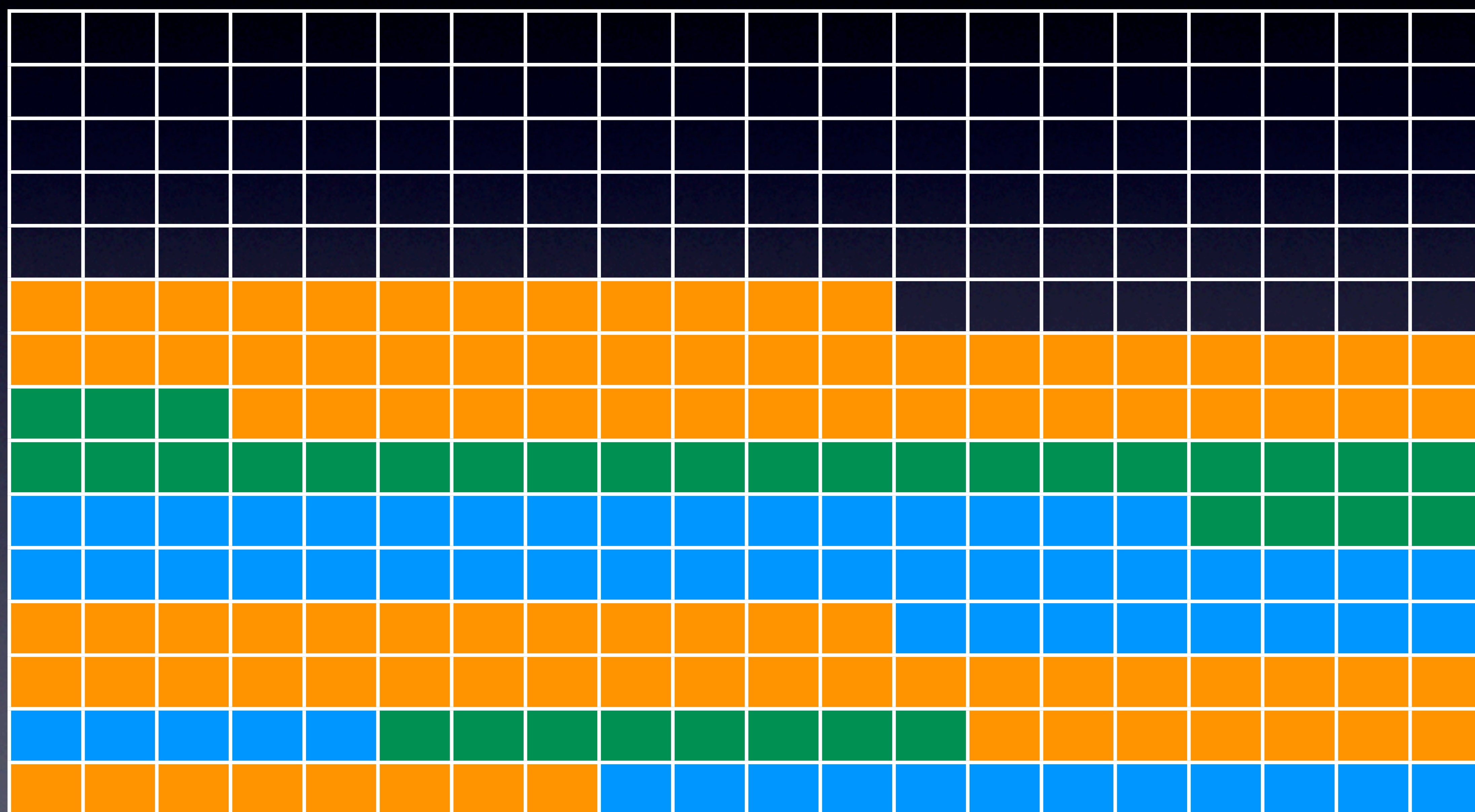
Problem:

- No fragmentation but
- <u>Lists</u>: linear access time
- <u>Trees</u>: log access time

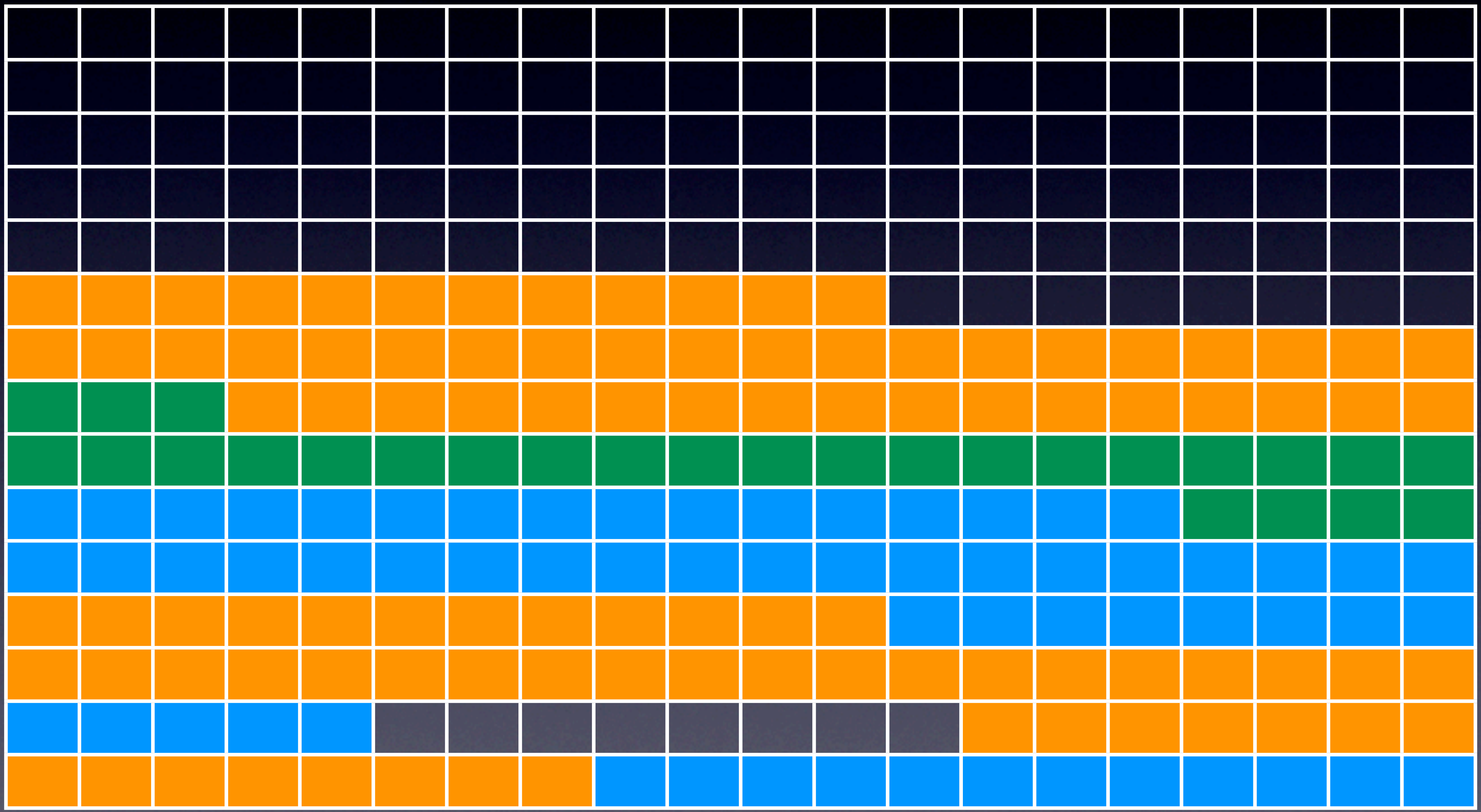


Lists/Trees Non-Contiguous

Keep It Compact?



Does Not Work!



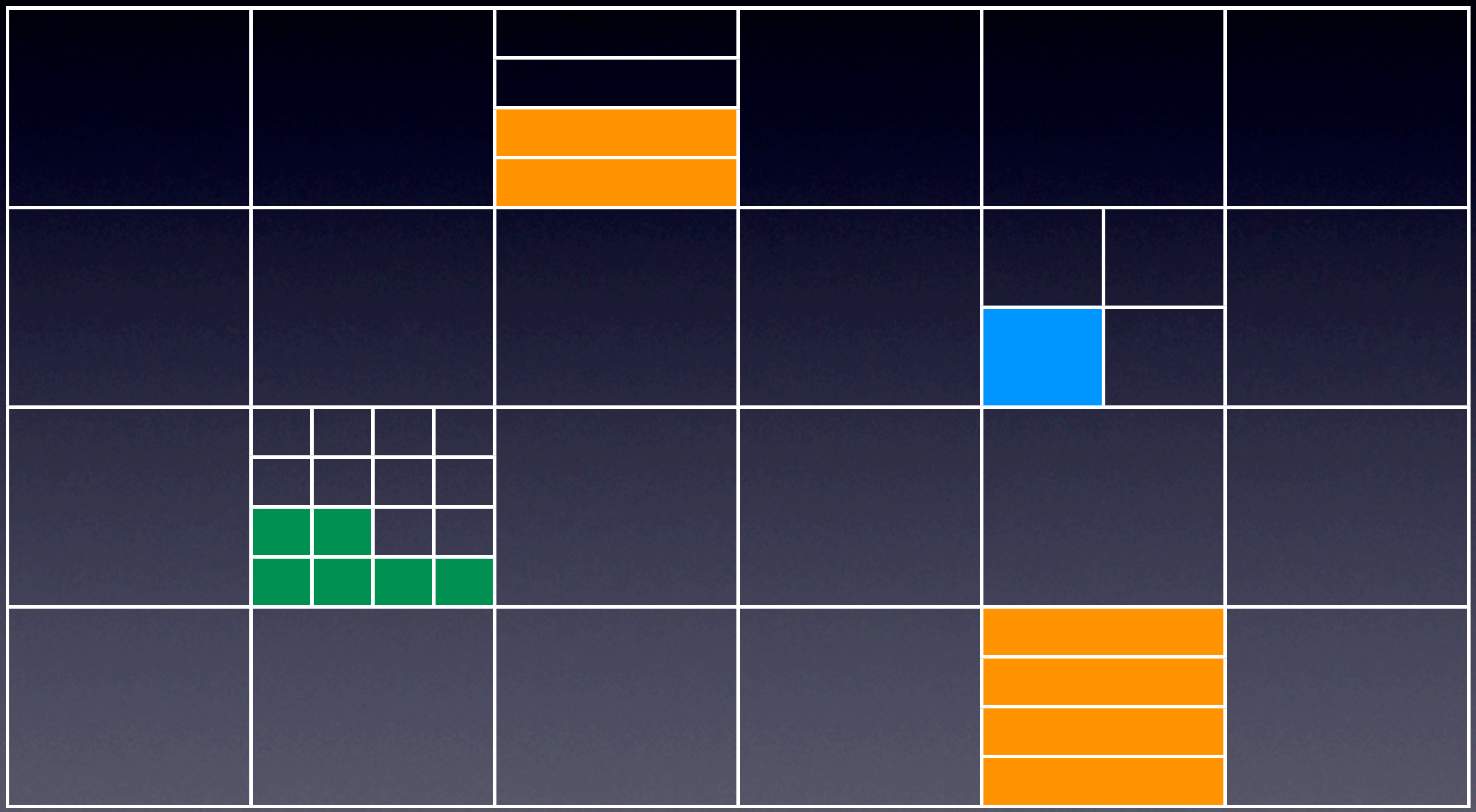
Trade-off Speed for
Memory Fragmentation

Keep Speed and
Memory Fragmentation
Bounded and Predictable

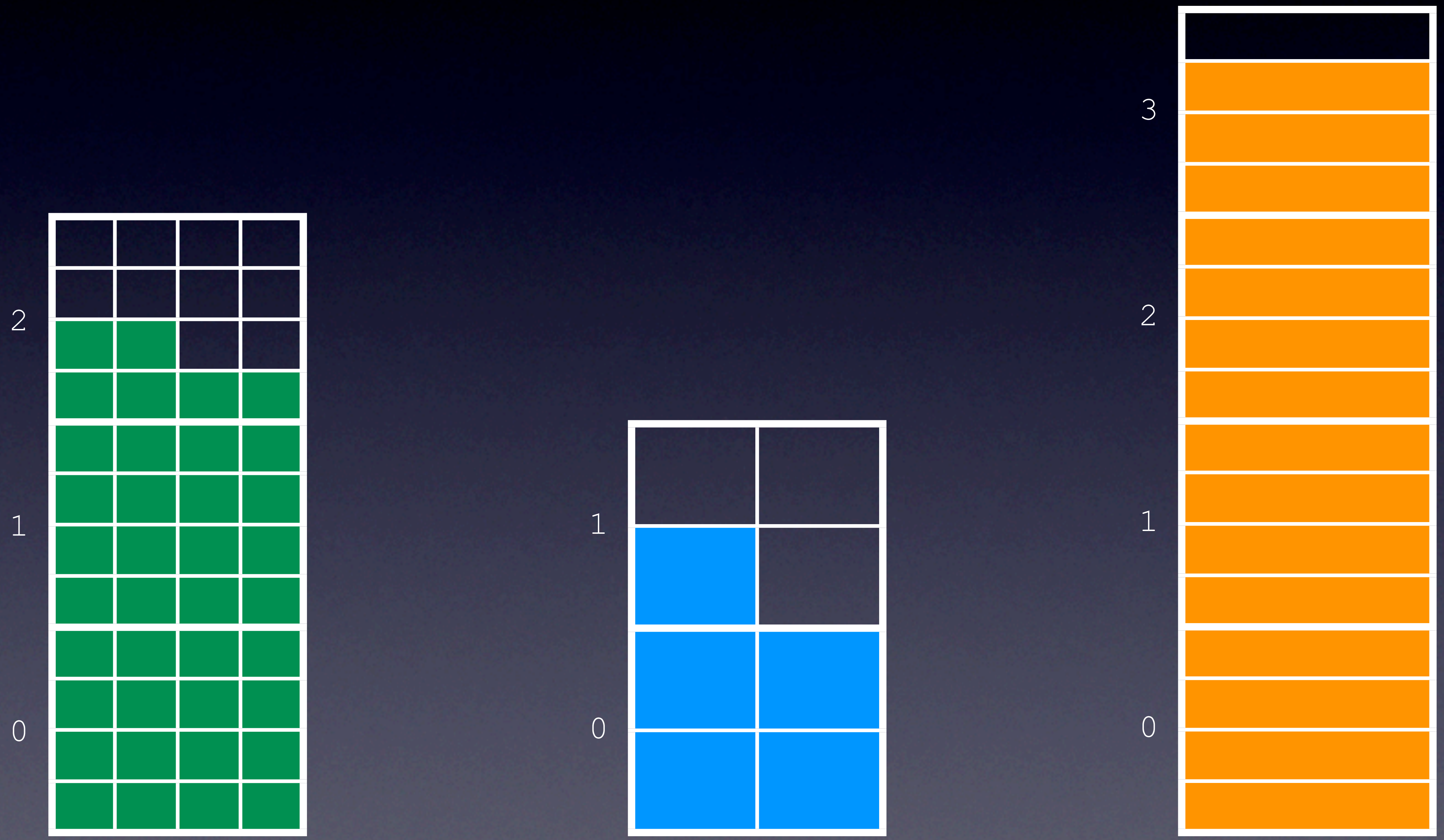
Partition Memory into Pages

16KB	16KB	16KB	16KB	16KB	16KB
16KB	16KB	16KB	16KB	16KB	16KB
16KB	16KB	16KB	16KB	16KB	16KB
16KB	16KB	16KB	16KB	16KB	16KB

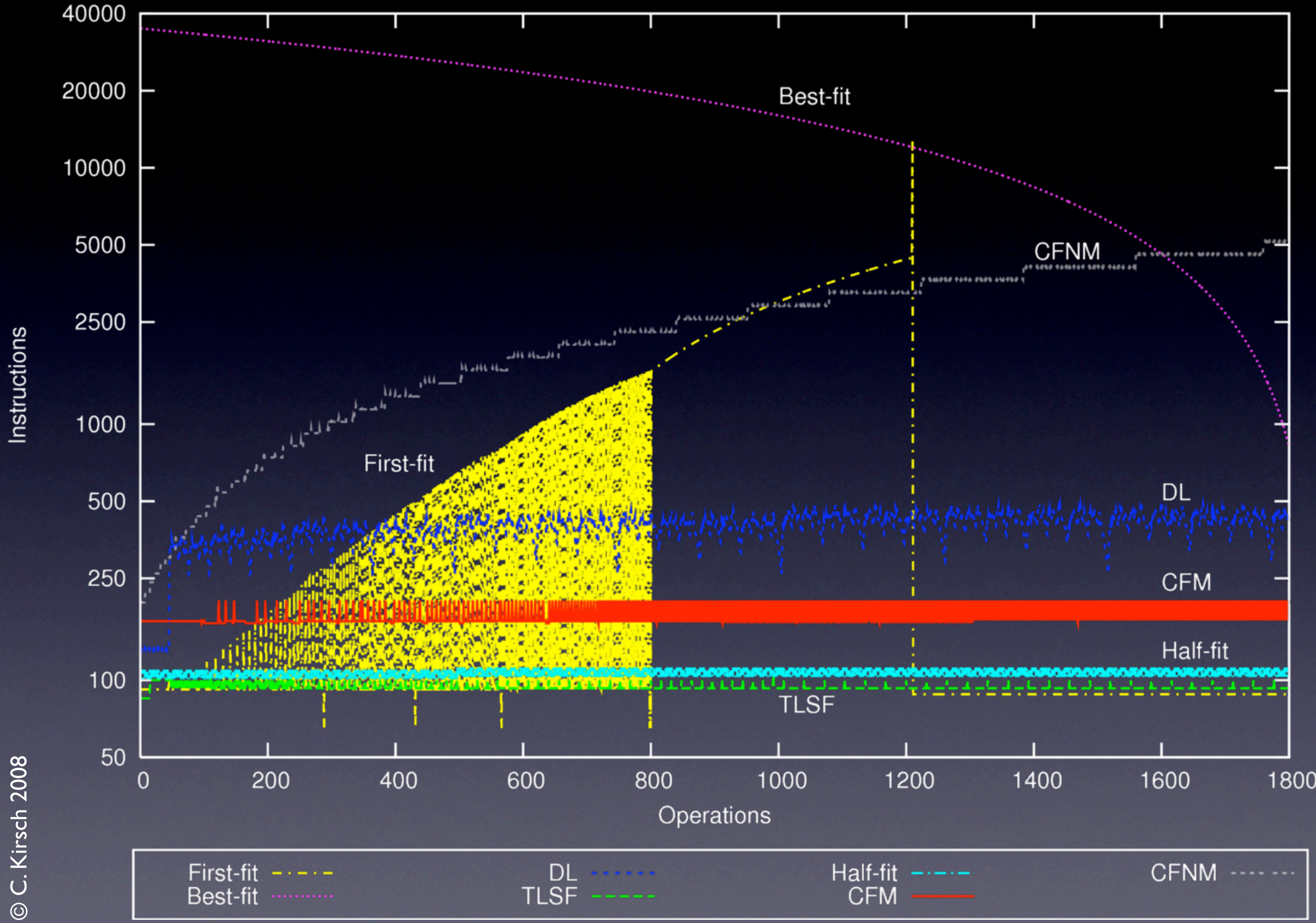
Partition Pages into Blocks



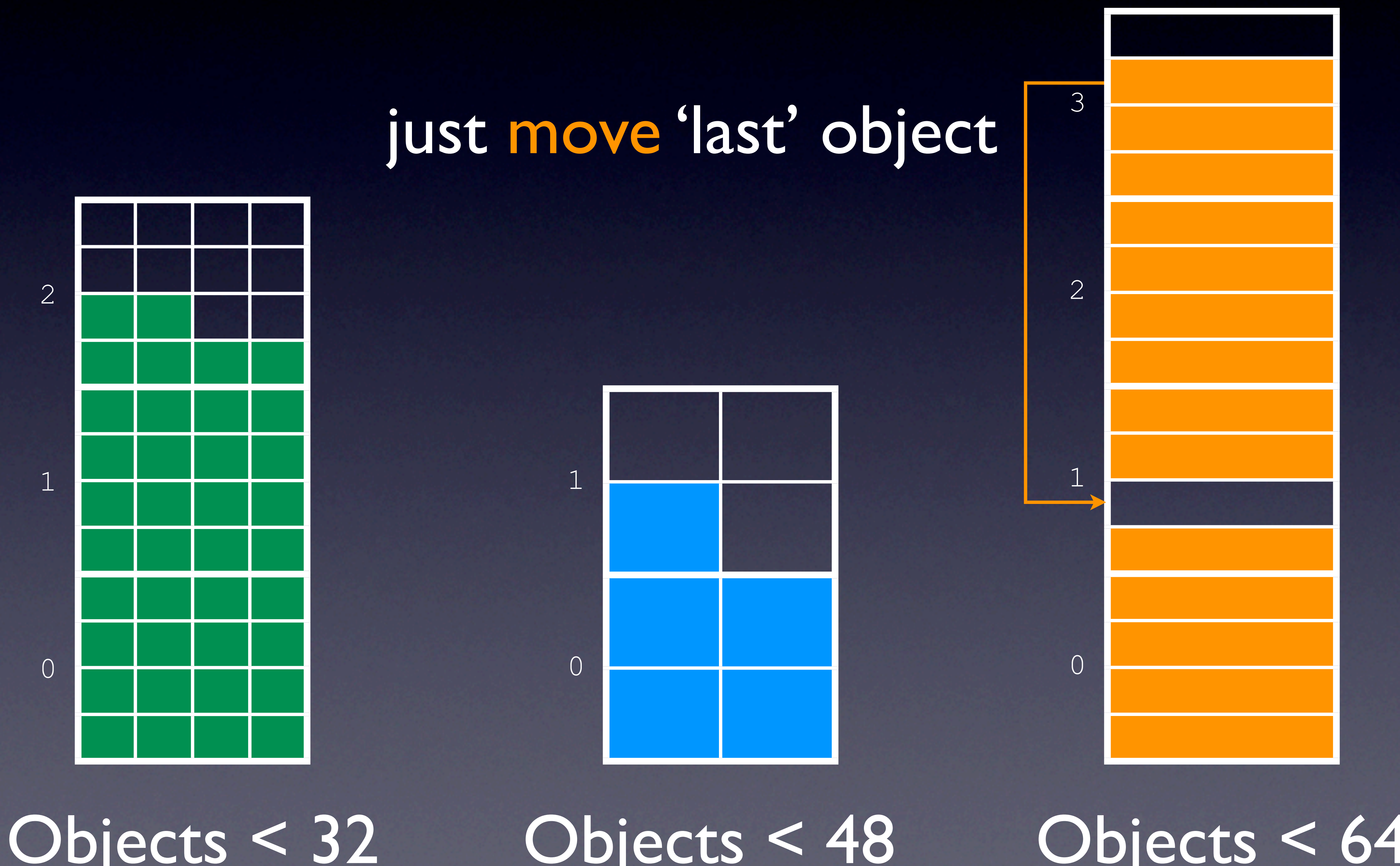
Size-Class Compact

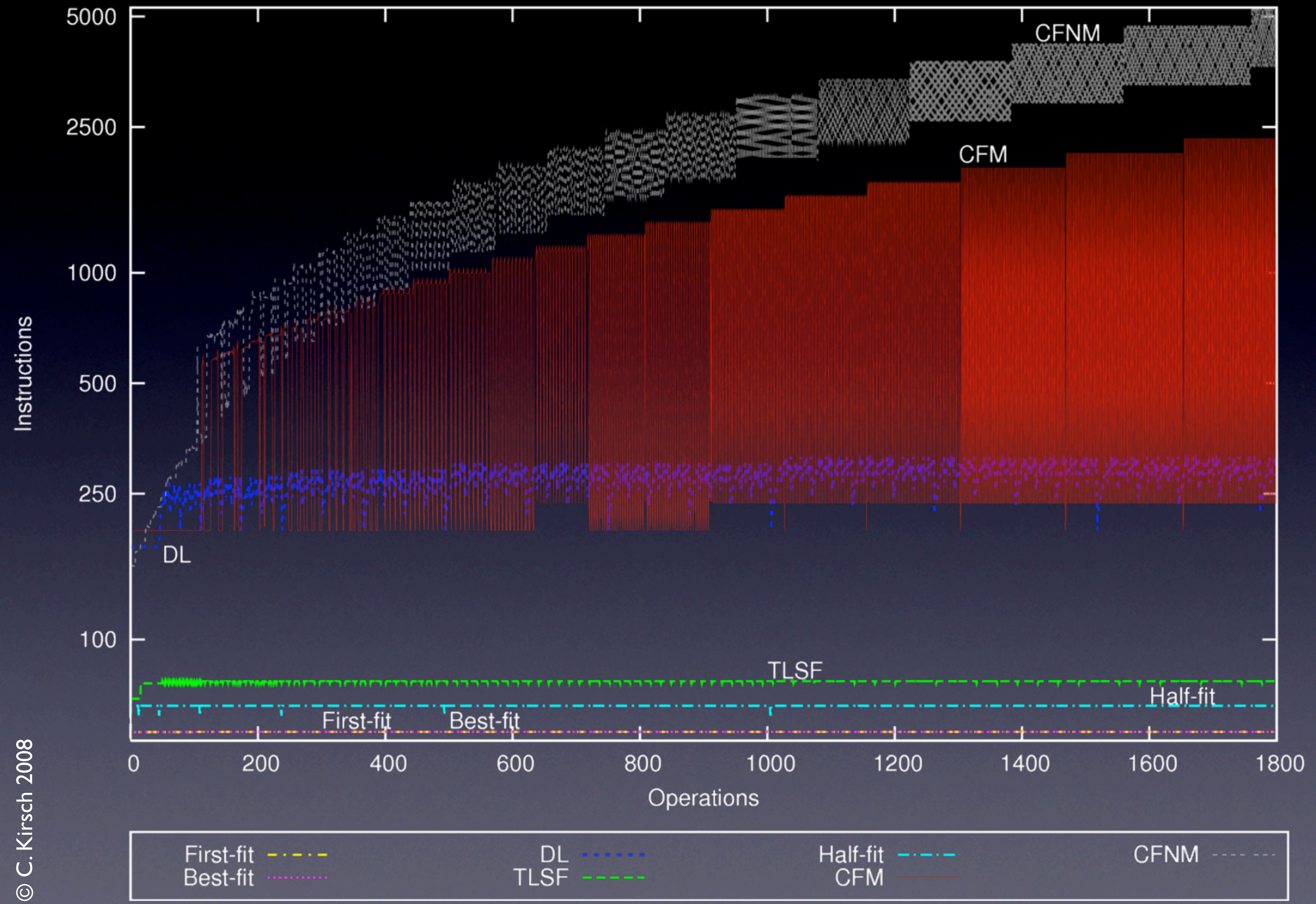


Objects < 32 Objects < 48 Objects < 64


40000
20000
10000
5000
2500
1000
500
250
100
50
Instructions
0
200
400
600
800
1000
1200
1400
1600
1800
Operations
Best-fit
CFNM
First-fit
DL
CFM
Half-fit
TLSF
First-fit
Best-fit
DL
TLSF
Half-fit
CFM
CFNM

"Compact-Fit" (Bounded Compaction)



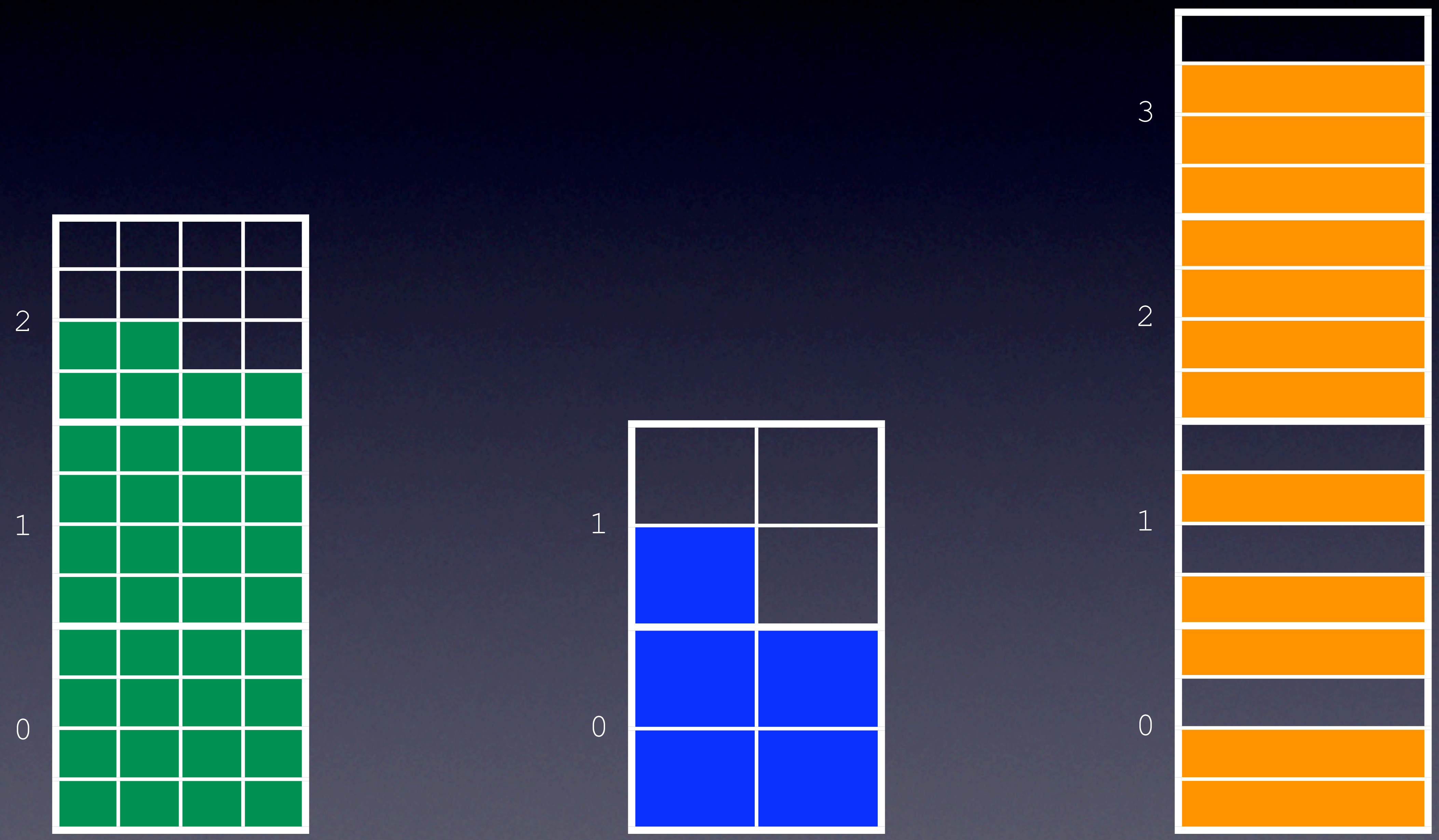


5000
2500
1000
500
250
100
Instructions
CFNM
CFM
DL
TLSF
Half-fit
First-fit
Best-fit
0
200
400
600
800
1000
1200
1400
1600
1800
Operations
First-fit
Best-fit
DL
TLSF
Half-fit
CFM
CFNM

Results I

- malloc(n) takes O(1)
- free(n) takes O(n)
  (because of compaction)
- access takes one indirection
  (because of abstract address space)
- memory fragmentation is bounded and predictable in constant time

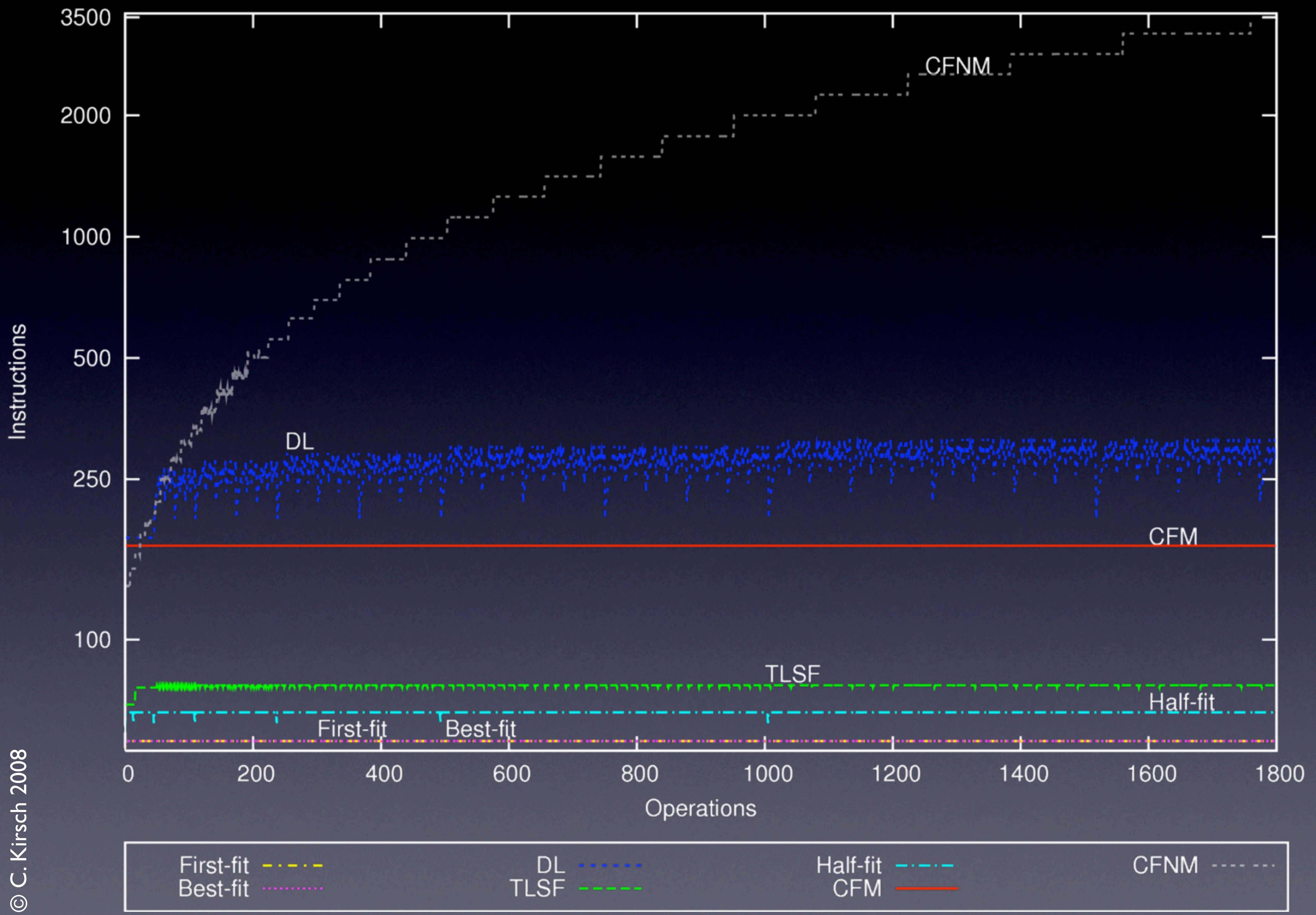
Partial Compaction



Objects < 32

Objects < 48

Objects < 64



3500
2000
1000
500
250
100
Instructions
0
200
400
600
800
1000
1200
1400
1600
1800
Operations
CFNM
DL
CFM
TLSF
Half-fit
First-fit
Best-fit
First-fit
Best-fit
DL
TLSF
Half-fit
CFM
CFNM

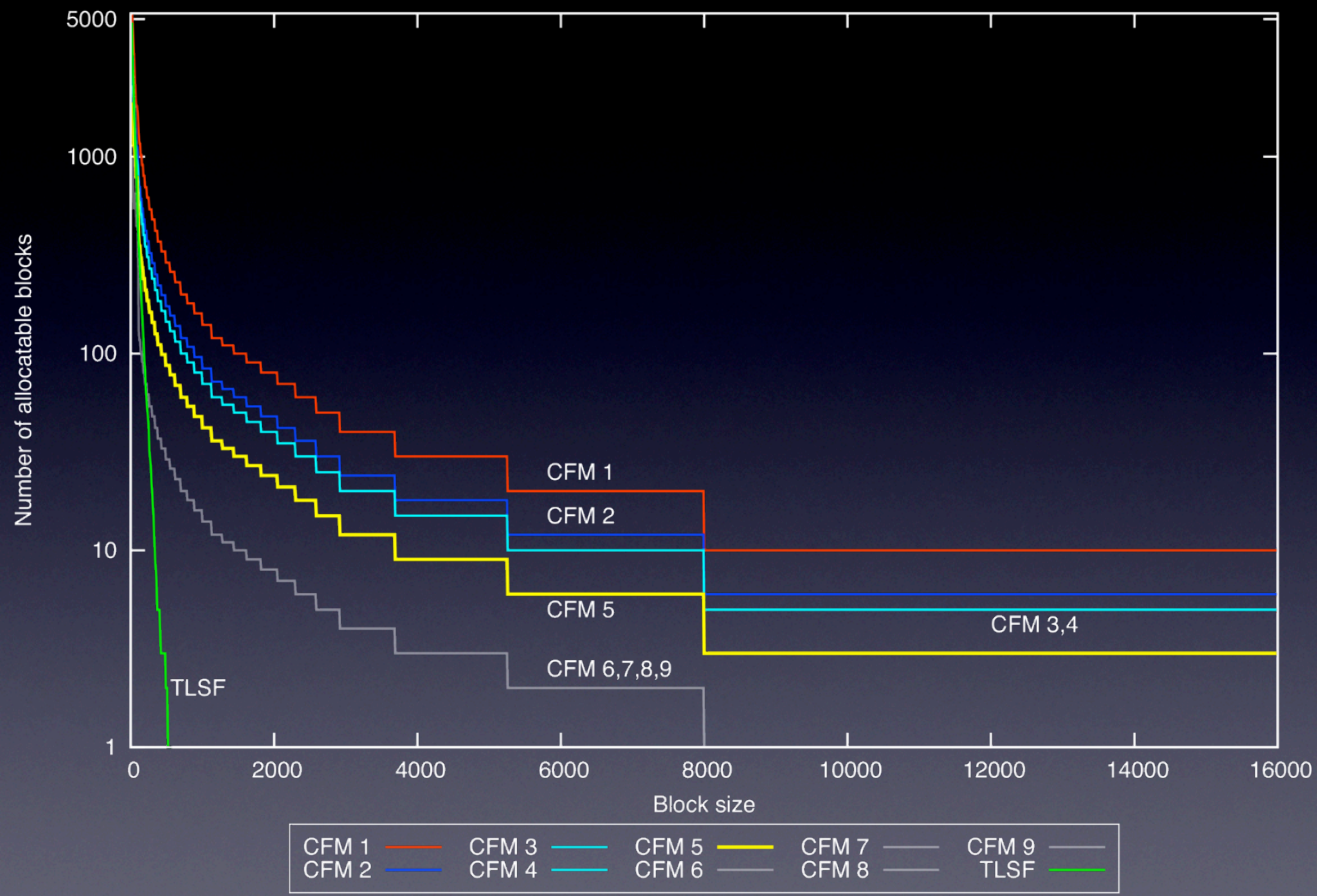
Program Analysis

Definition:
Let *k* count deallocations in a given size-class for which no subsequent allocation was done (“k-band mutator”).

Proposition:
Each deallocation that happens when *k* < max_number_of_non_full_pages takes constant time.

Results II

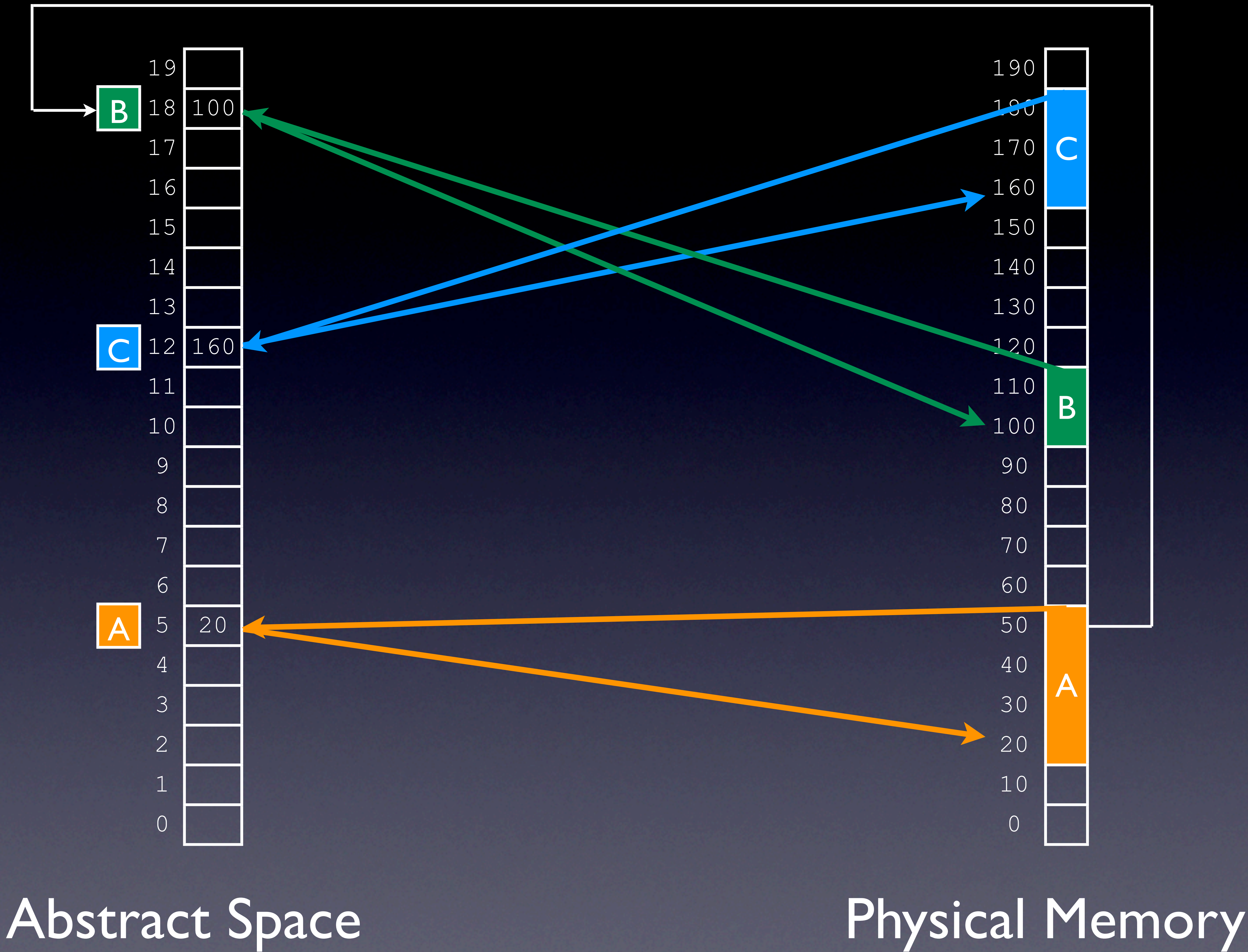
- if mutator stays within k-bands:
  - malloc(n) takes O(1)
  - free(n) takes O(1)
  - access takes one indirection
- memory fragmentation is bounded in k and predictable in constant time



Number of allocatable blocks
5000
1000
100
10
1
0
2000
4000
6000
8000
10000
12000
14000
16000
Block size
CFM 1
CFM 2
CFM 5
CFM 3,4
CFM 6,7,8,9
TLSF
CFM 1
CFM 2
CFM 3
CFM 4
CFM 5
CFM 6
CFM 7
CFM 8
CFM 9
TLSF

Two Implementations!

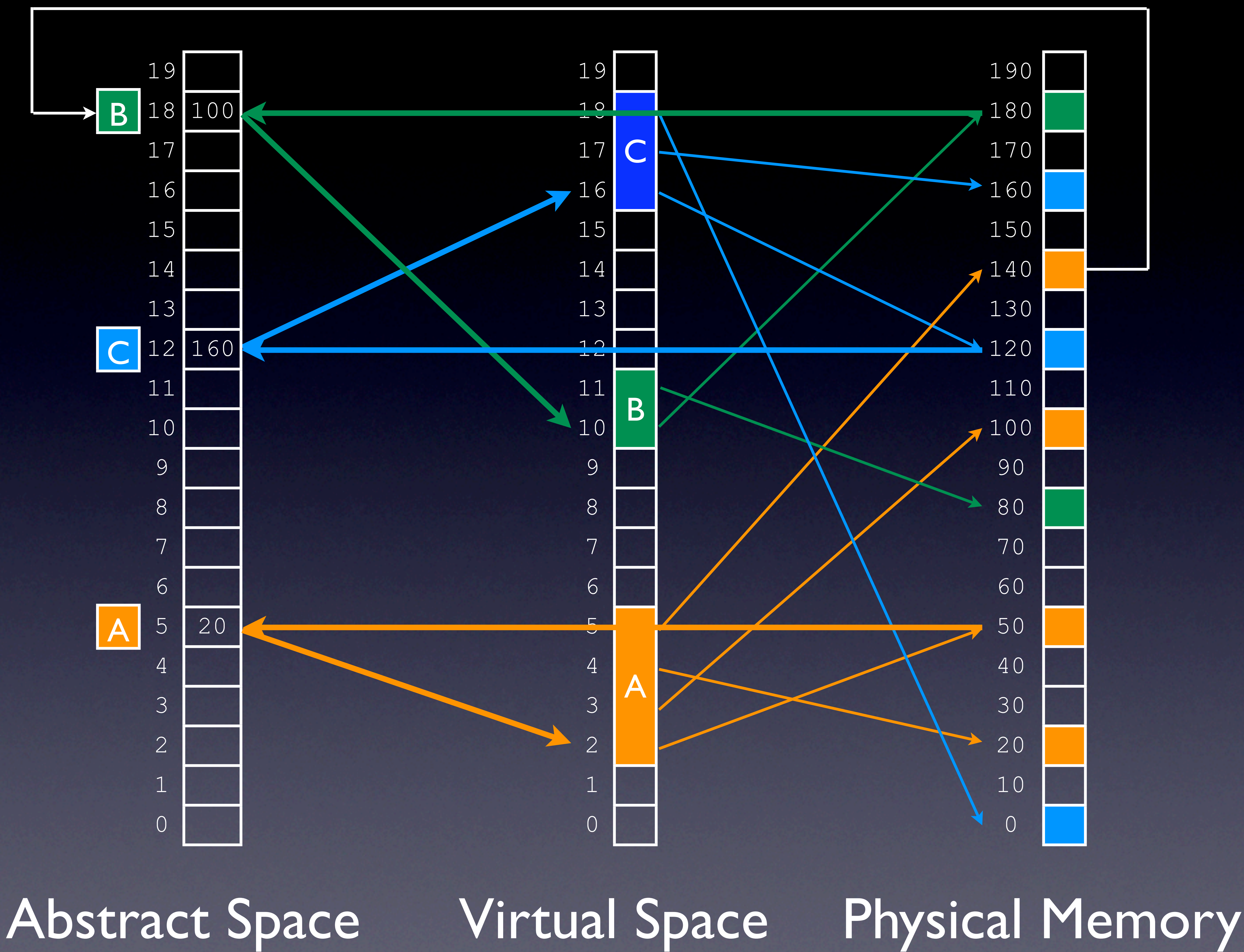
1. Concrete Space = Physical Memory
2. Concrete Space = Virtual Memory



B
C
A
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
100
160
20
190
180
170
160
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
Abstract Space
Physical Memory

Two Implementations!

1. Concrete Space = Physical Memory
2. Concrete Space = Virtual Memory

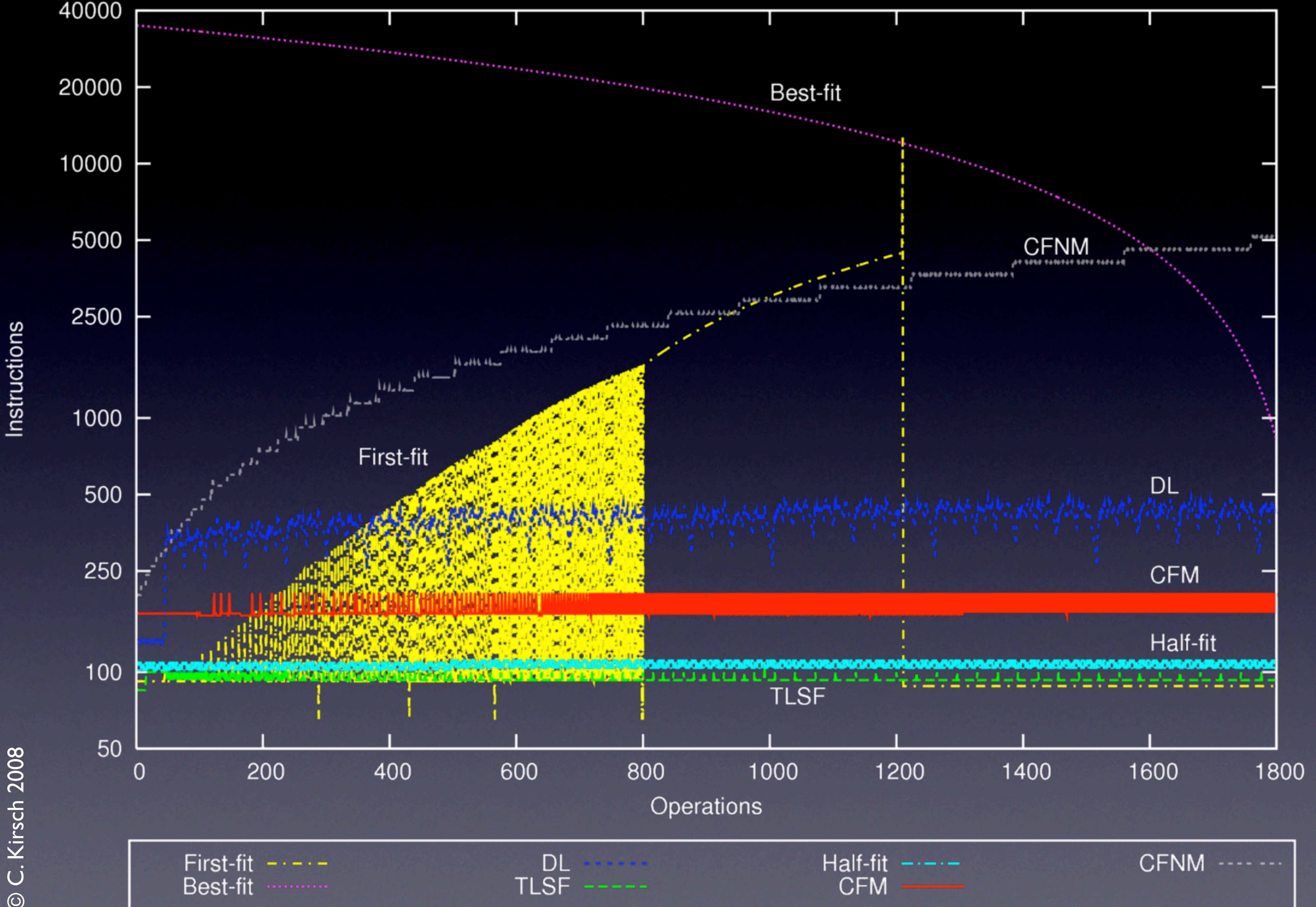


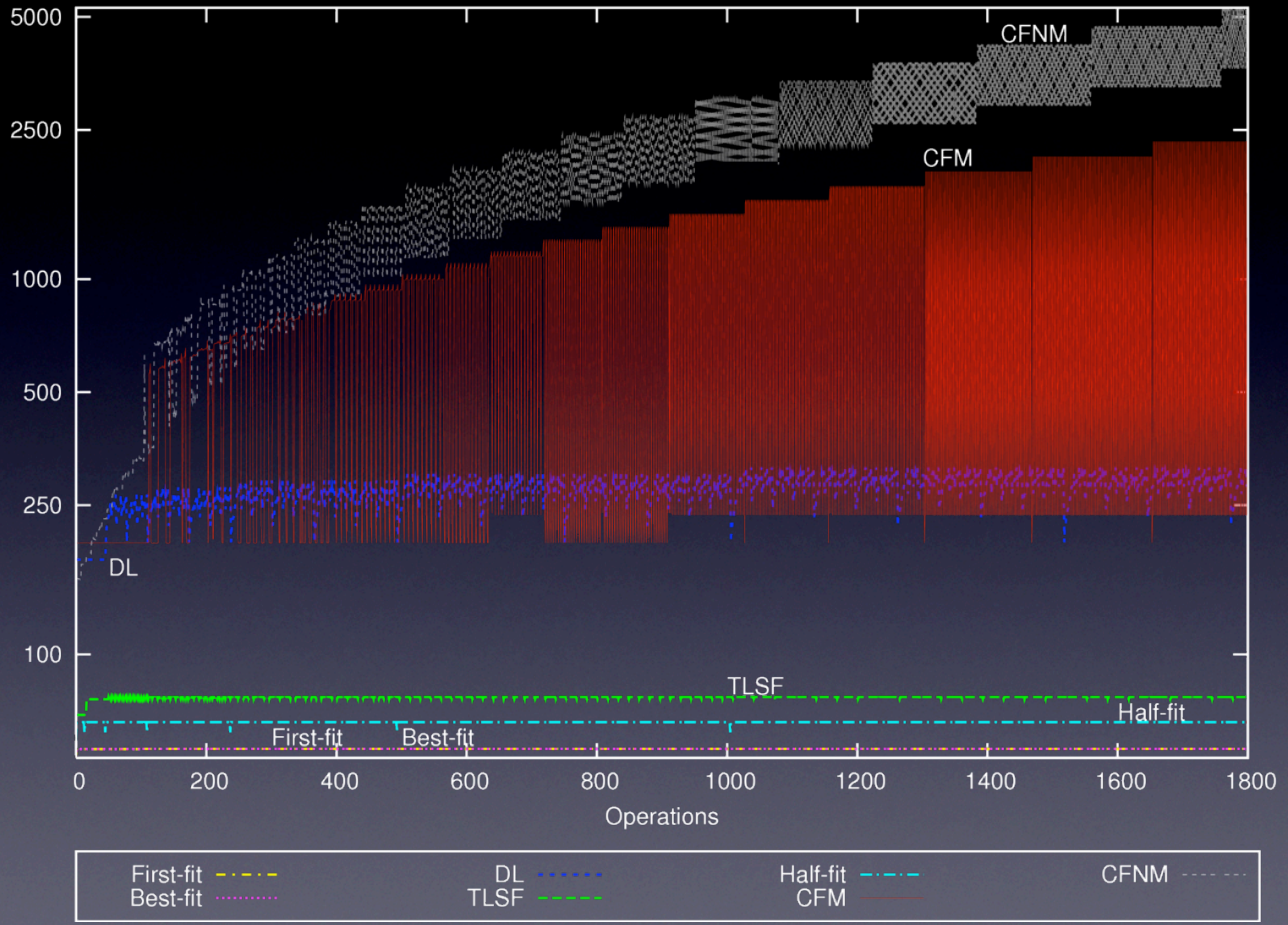
B
C
A
100
160
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
190
180
170
160
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
Abstract Space
Virtual Space
Physical Memory

Results III

- malloc(n) takes Θ(n) (because of block table)
- free(n) takes Θ(n)
  (because of block table and compaction)
- access takes two indirections
  (because of abstract/virtual address space)

- memory fragmentation is bounded in k and predictable in constant time

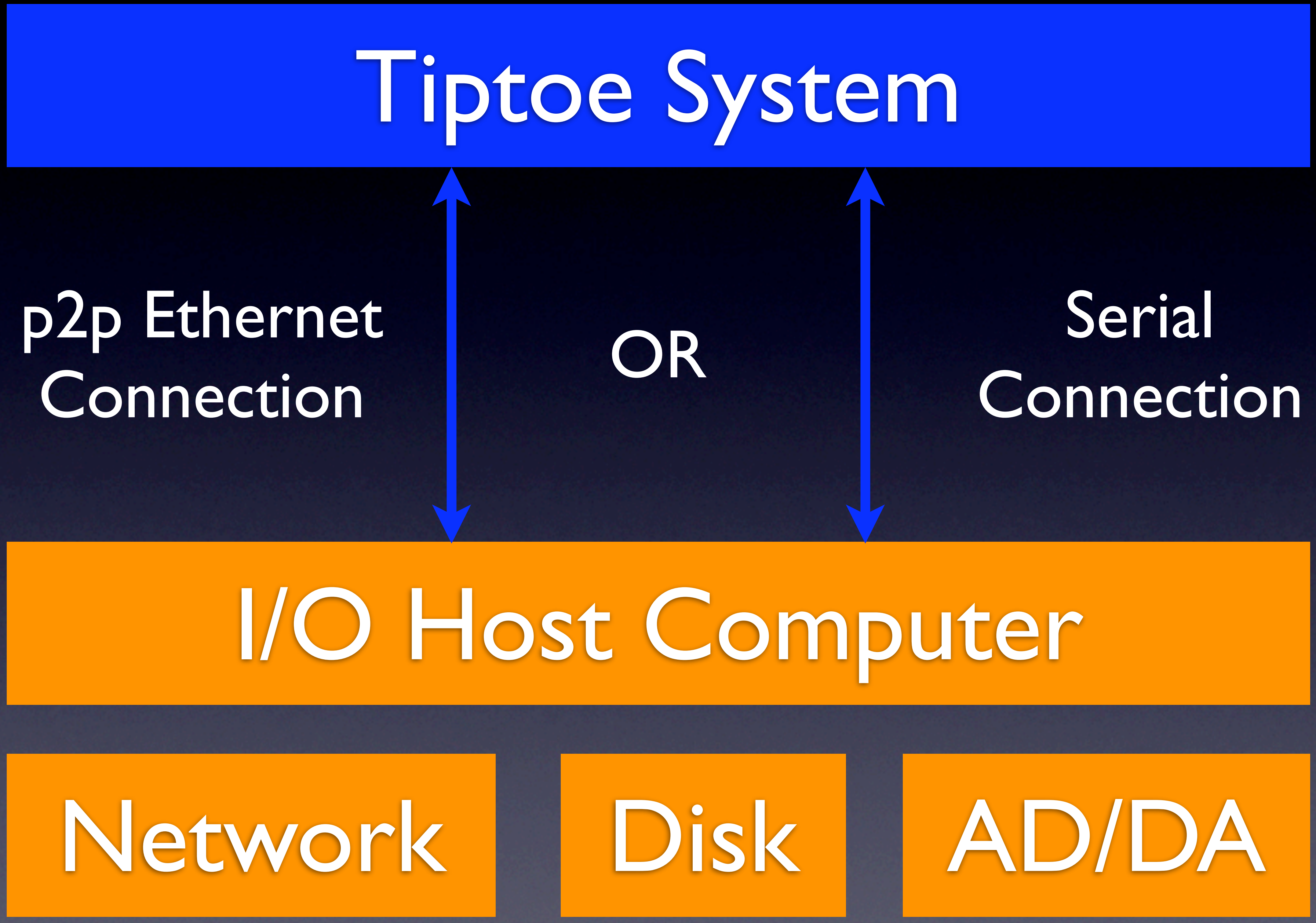

40000
20000
10000
5000
2500
1000
500
250
100
50
Instructions
0
200
400
600
800
1000
1200
1400
1600
1800
Operations
Best-fit
CFNM
First-fit
DL
CFM
Half-fit
TLSF
First-fit
Best-fit
DL
TLSF
Half-fit
CFM
CFNM



5000
2500
1000
500
250
100
Instructions
CFNM
CFM
DL
TLSF
Half-fit
First-fit
Best-fit
0
200
400
600
800
1000
1200
1400
1600
1800
Operations
First-fit
Best-fit
DL
TLSF
Half-fit
CFM
CFNM

Outline

1. Introduction
2. Process Model
3. Concurrency Management
4. Memory Management
5. I/O Management


Tiptoe System
p2p Ethernet Connection
OR
Serial Connection
I/O Host Computer
Network
Disk
AD/DA

Current/Future Work

- Concurrent memory management
- Process management
- I/O subsystem


Thank you