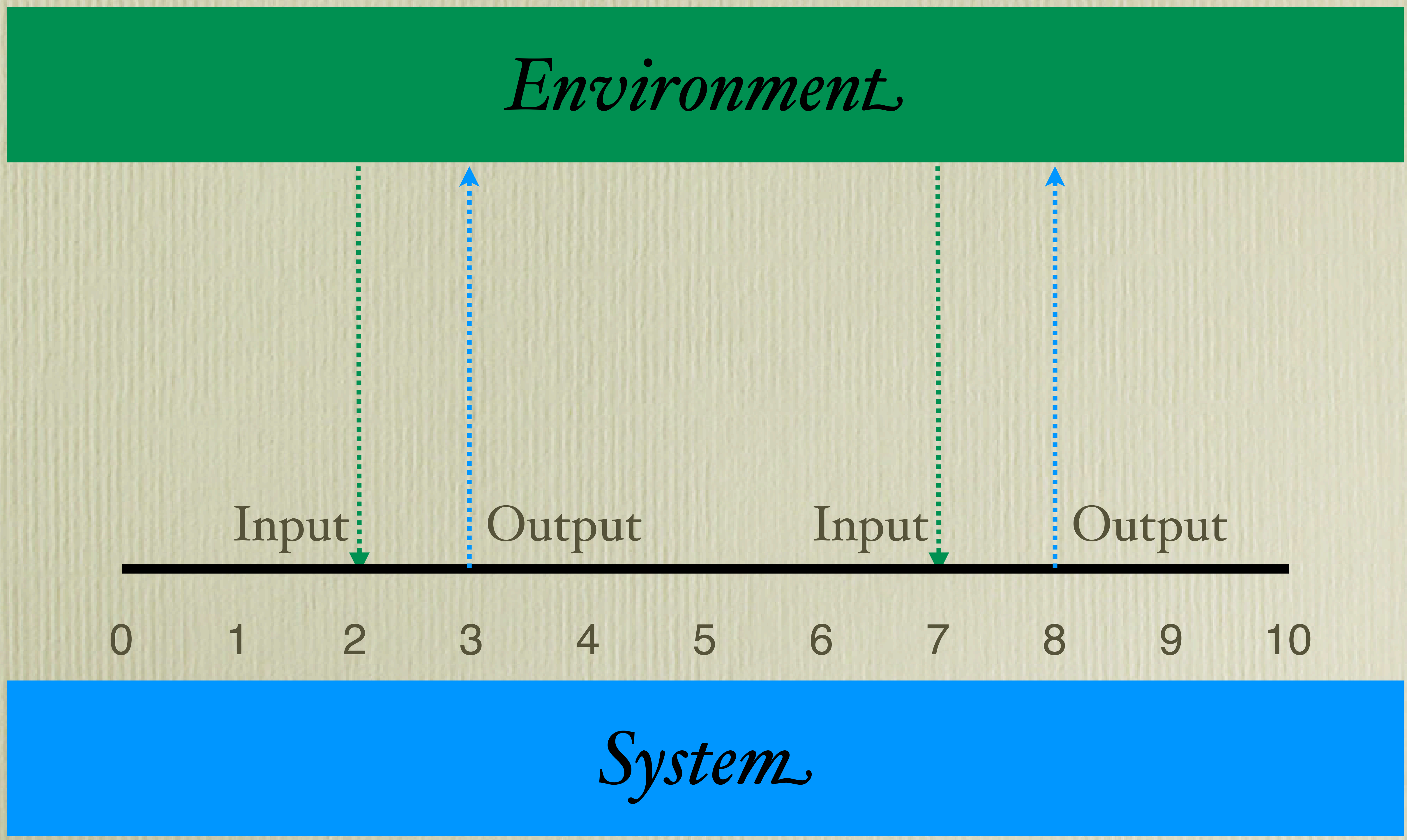
High-Level Programming of Real-Time and Concurrent Software Systems

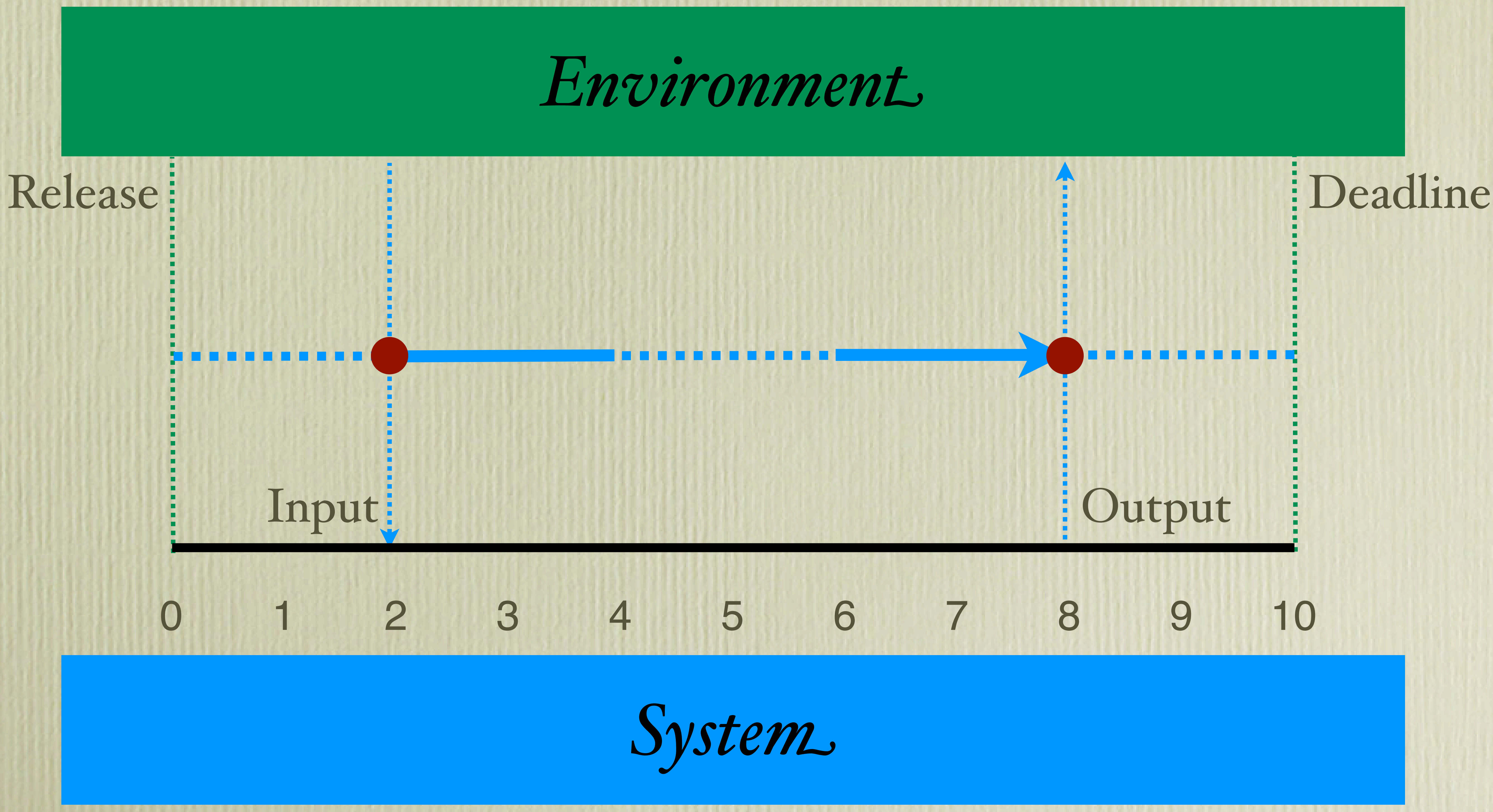
Christoph Kirsch
Universität Salzburg

Purdue University, December 2005

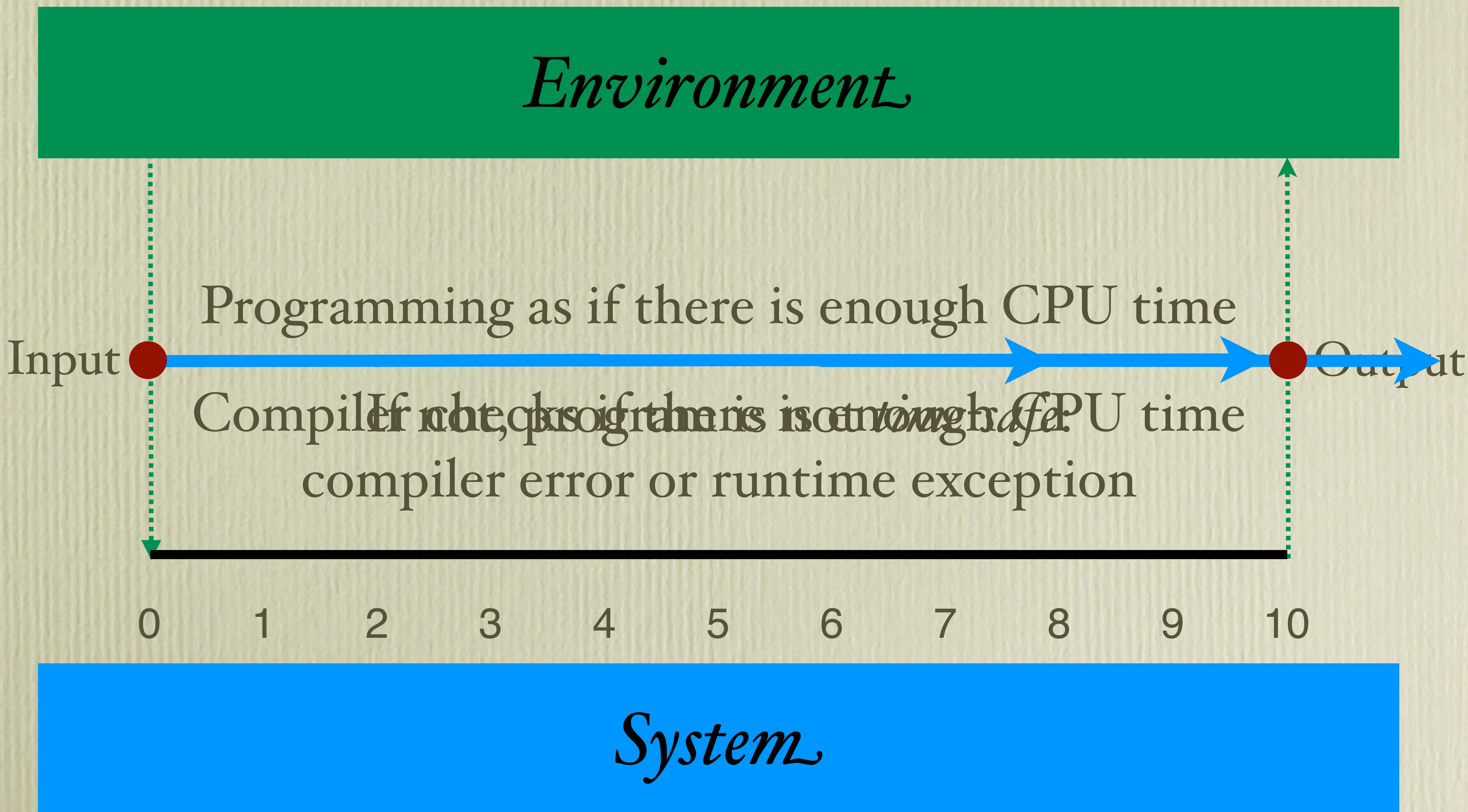
Real-Time Programming



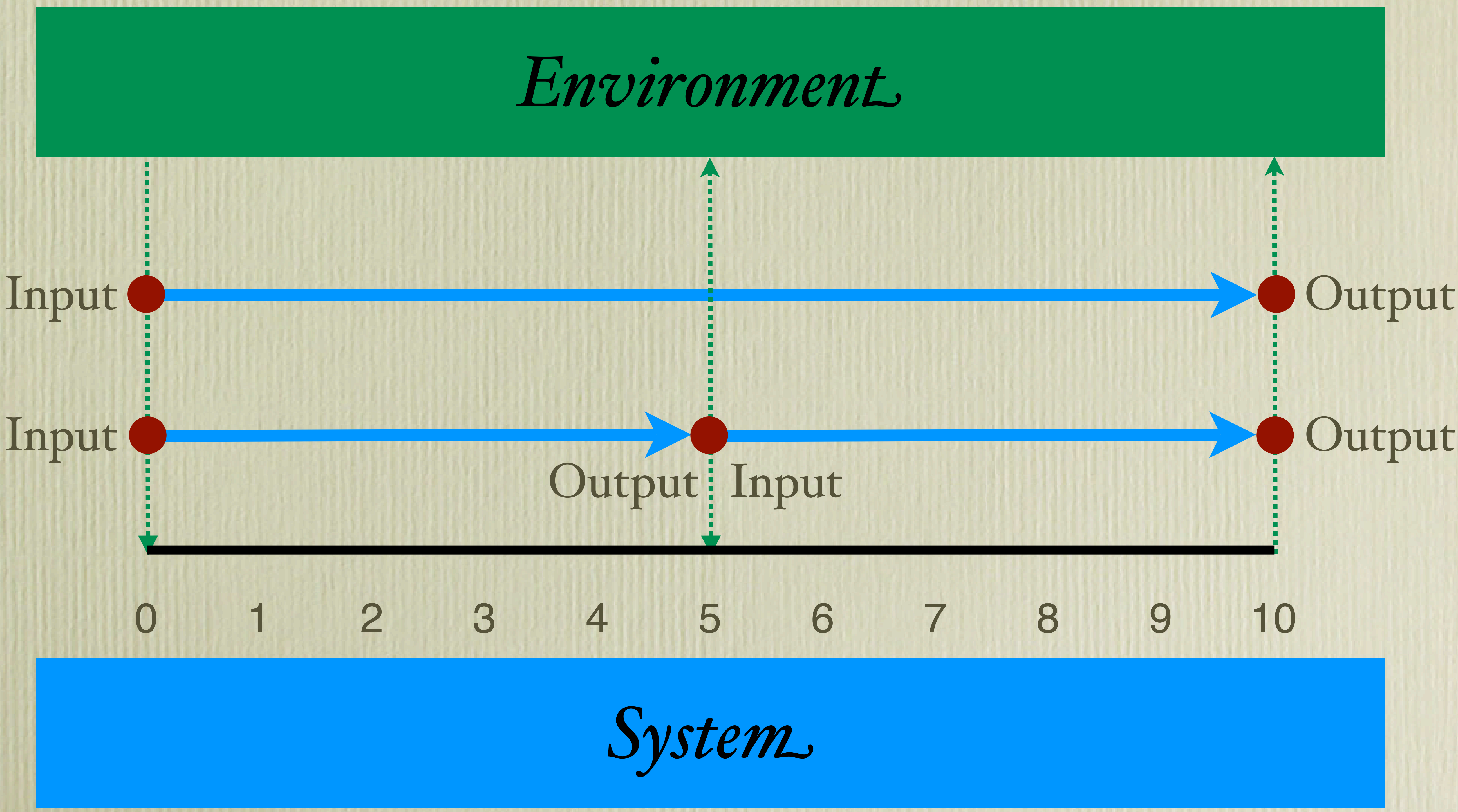
RT Programming Tradition



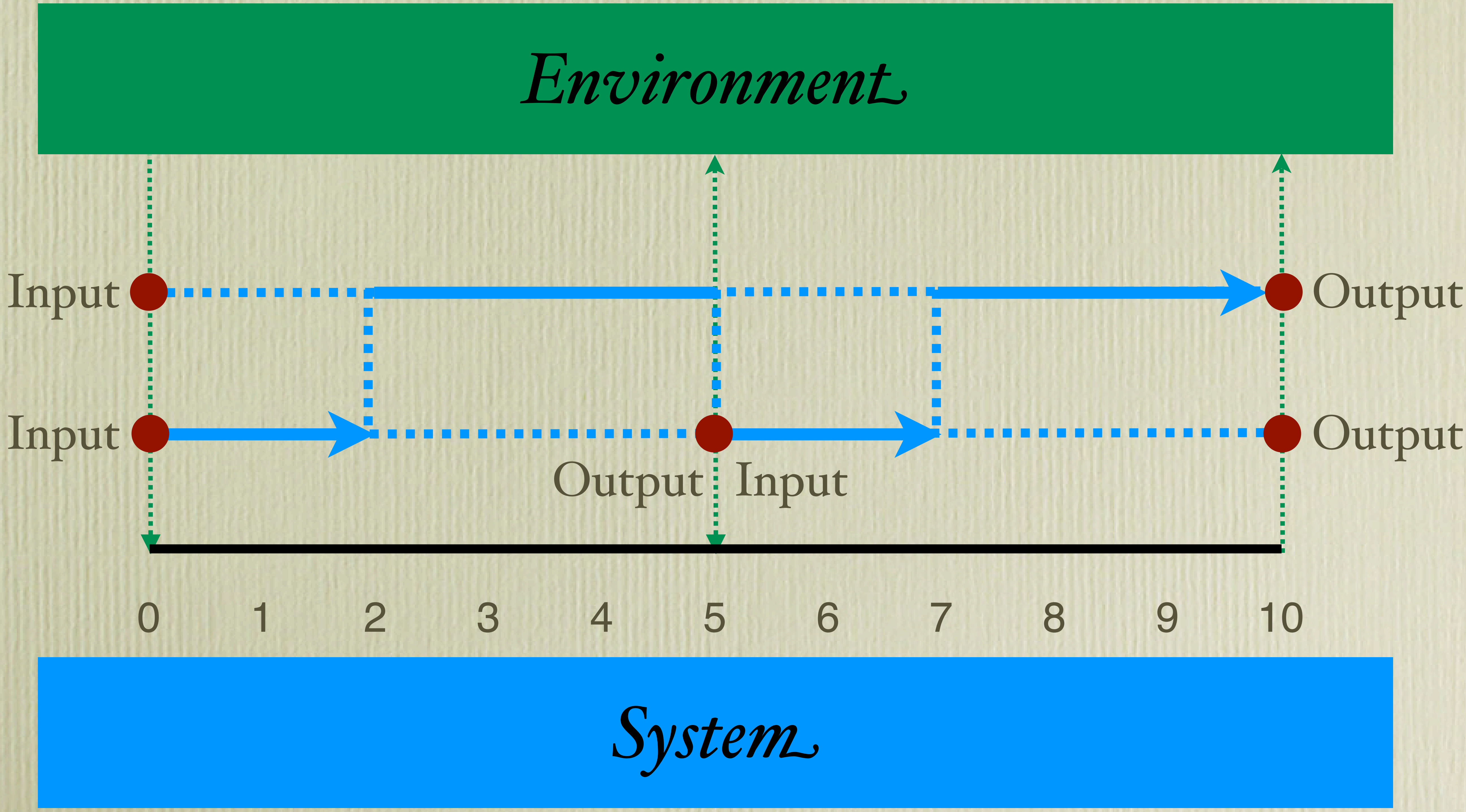
Logical Execution Time (LET)



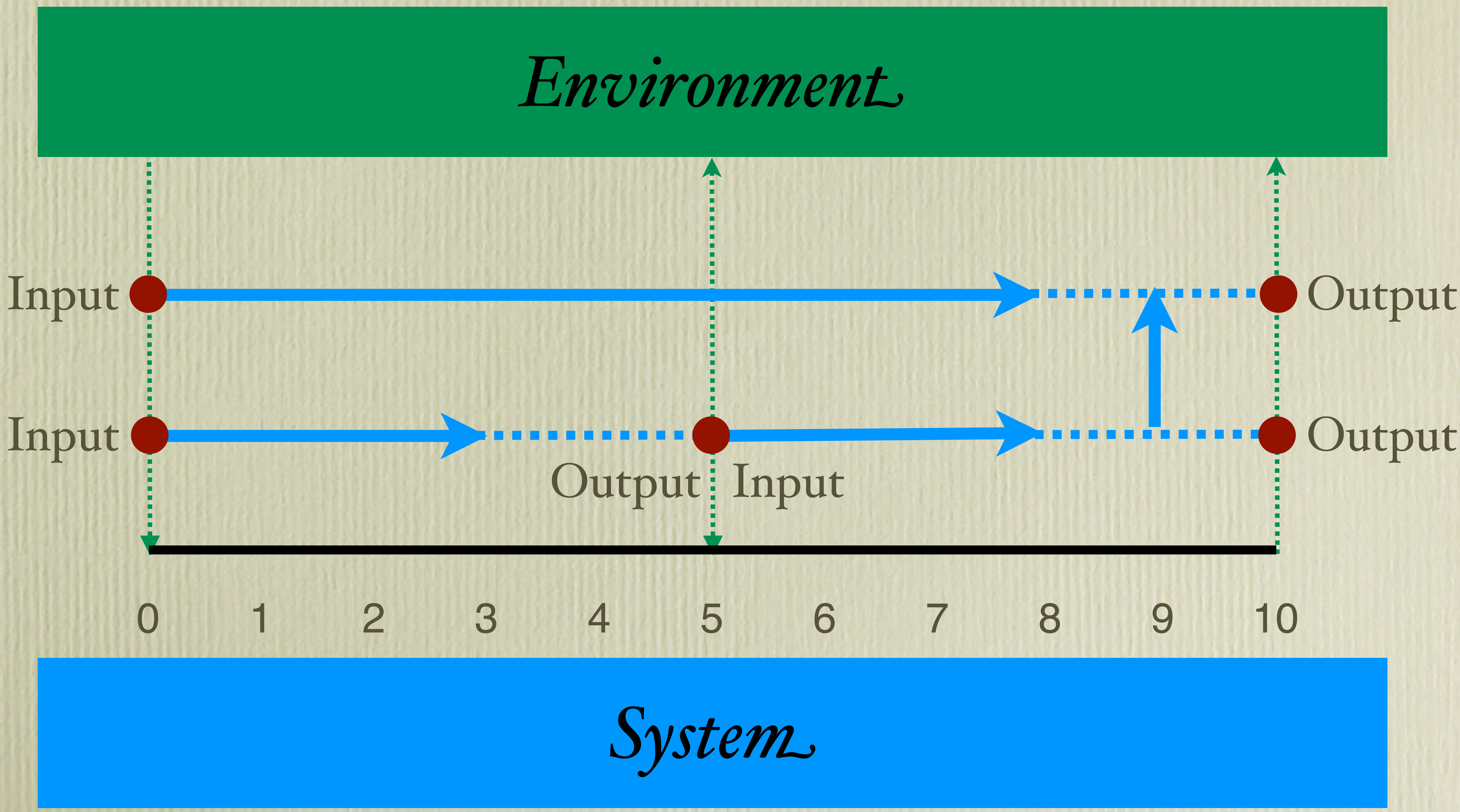
LET Programming



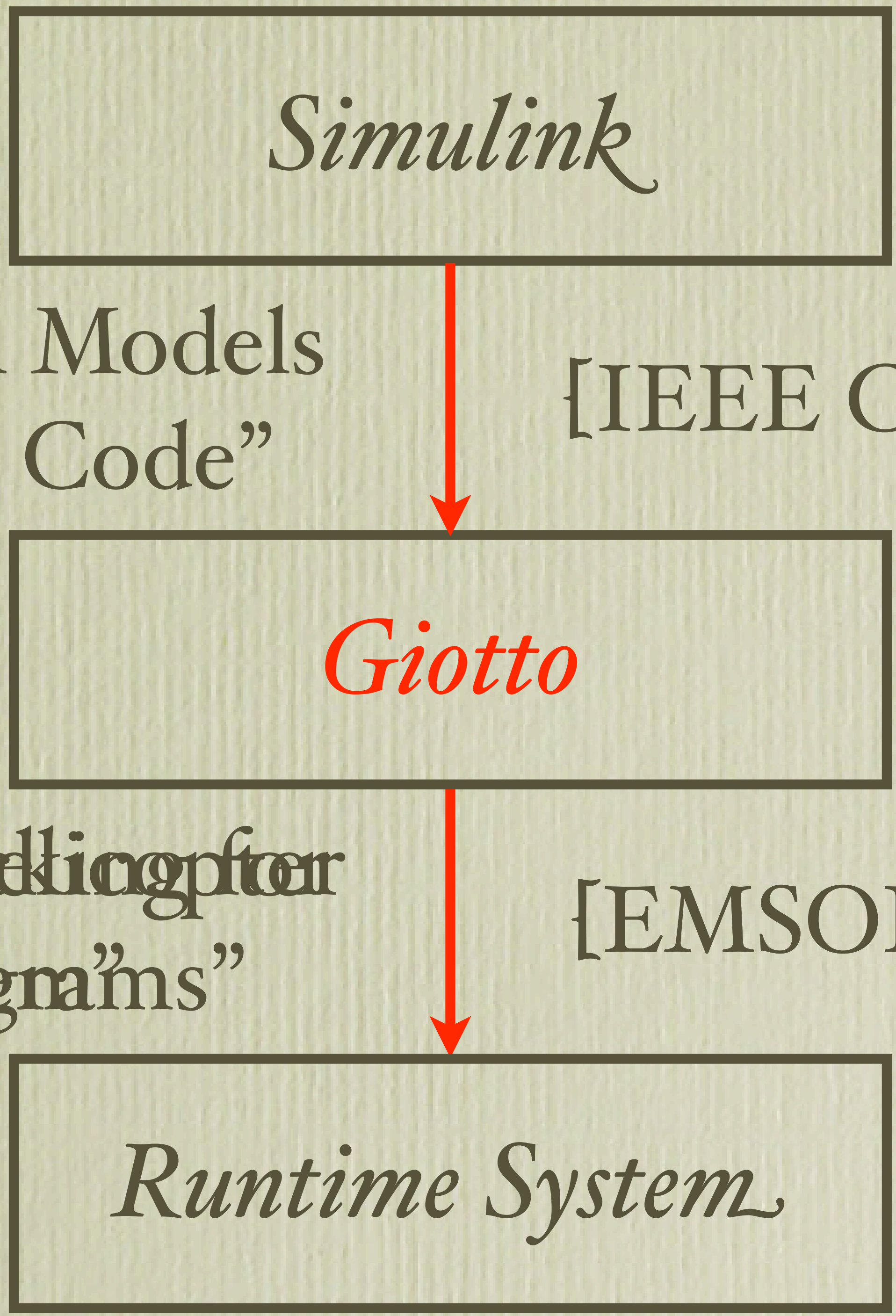
Single CPU, EDF Scheduler



Two CPUs, TDMA Network



Tool Chain



“From Control Models to Real-Time Code”

[IEEE CSM, 2003]

“Giotto: A Time Triggered Language for Embedded Programming”

[Proc. IEEE, 2003]
[EMSOFT, 2001]

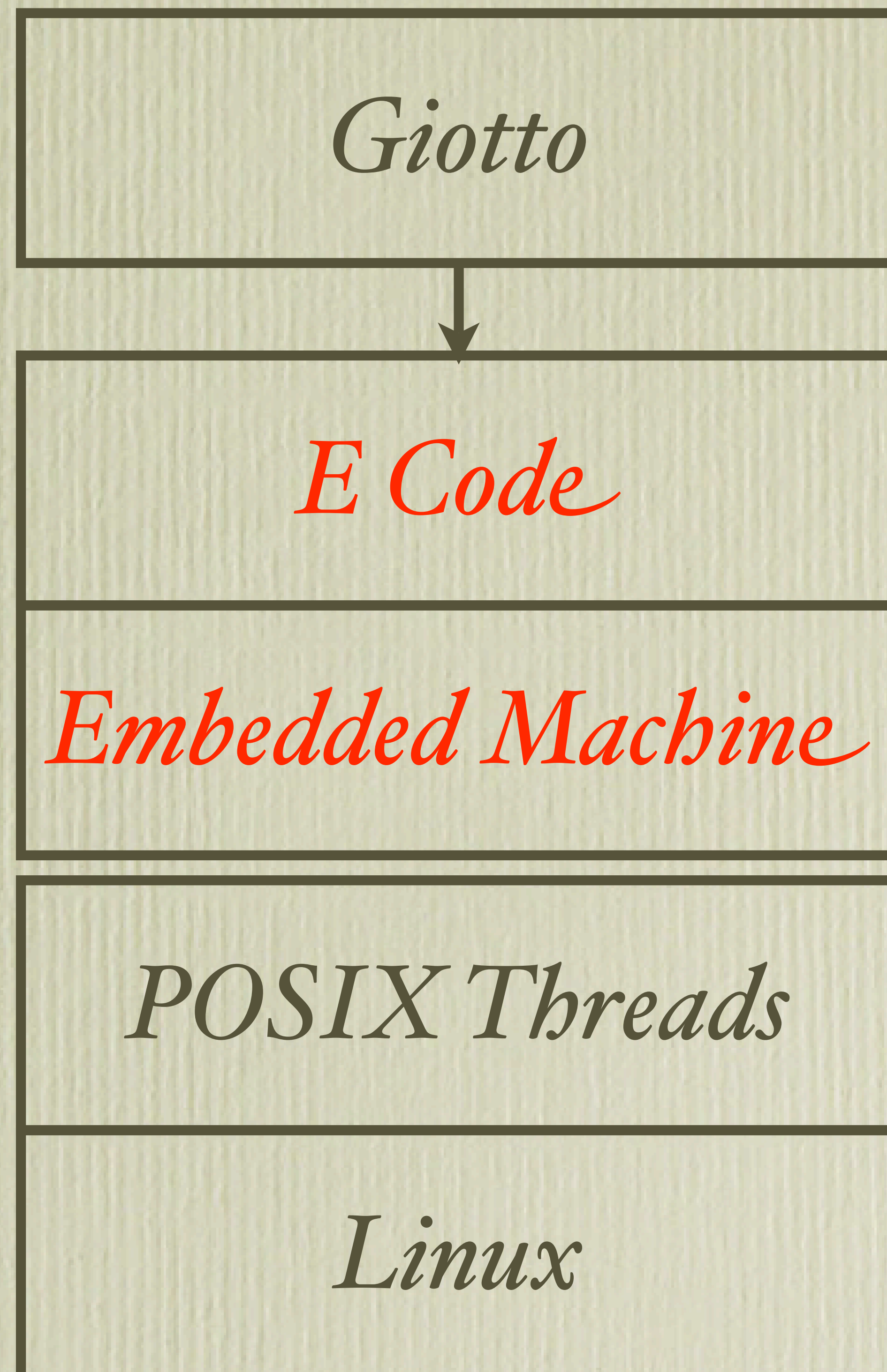
“A Giotto-Based Helicopter Control System”

“Time-Safety Checking for Embedded Programs”

[EMSOFT, 2002]

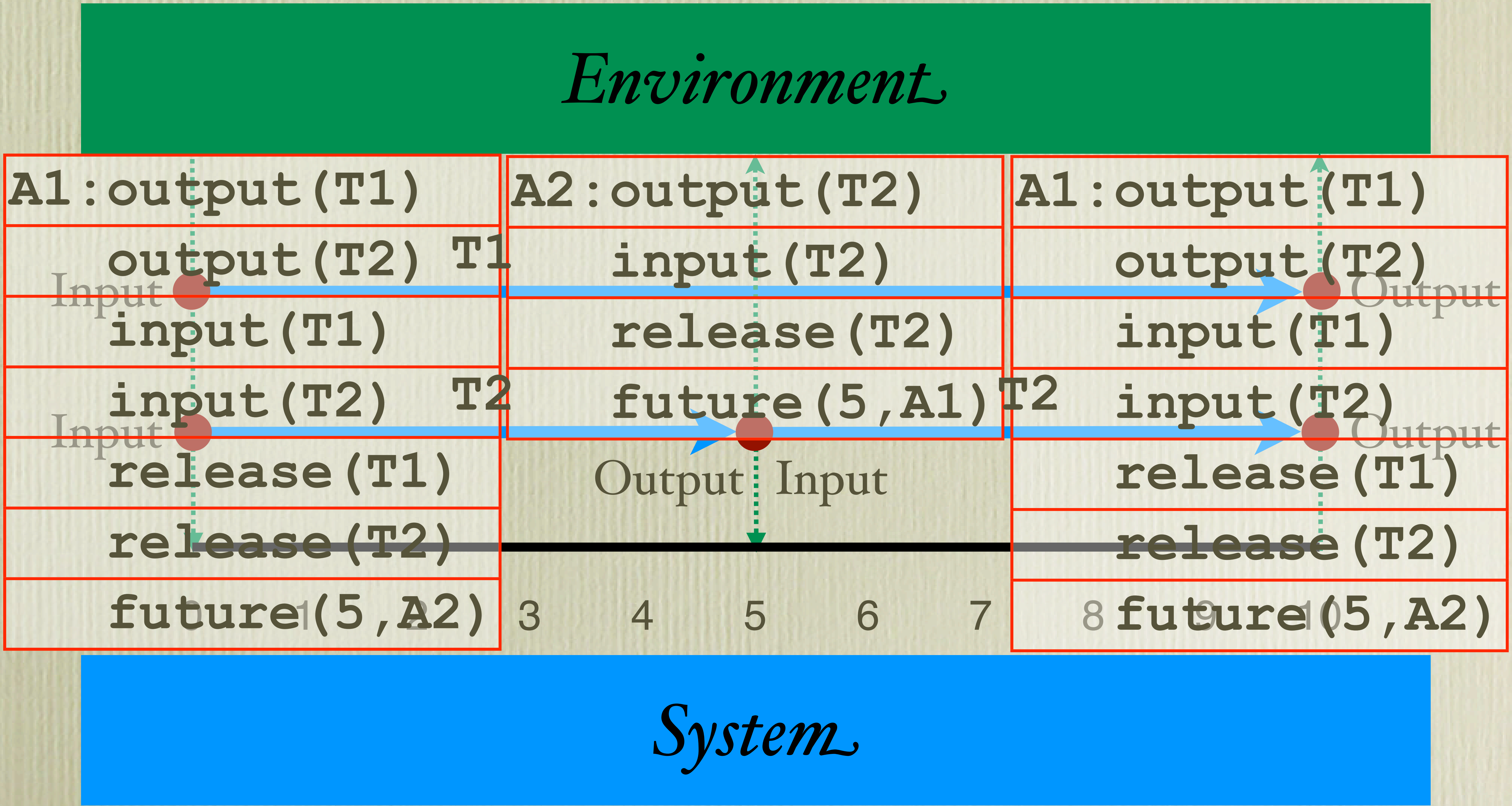
Runtime System

“The Embedded Machine:
Predictable, Portable
Real-Time Code”



[PLDI, 2002]

E Code

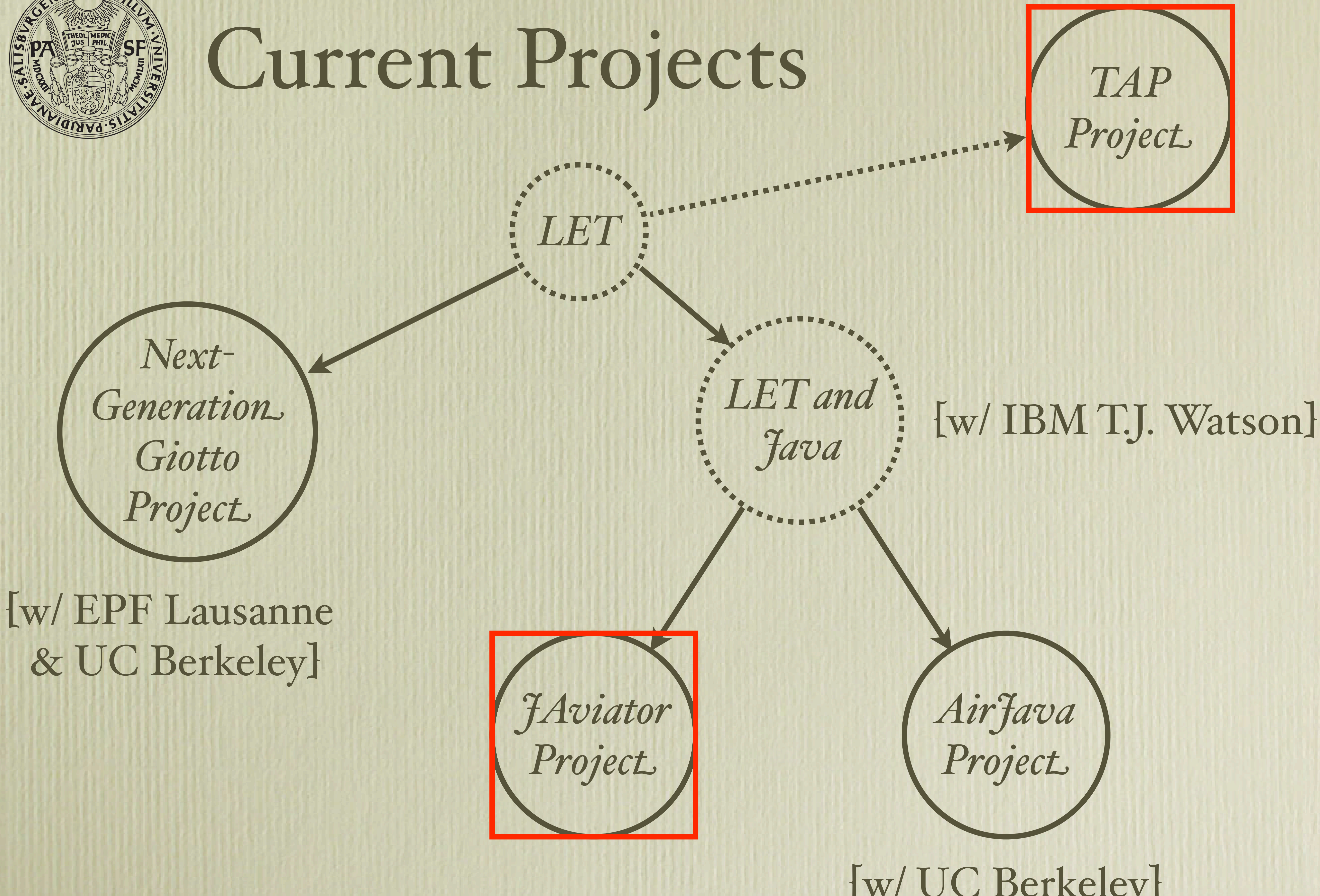


Environment
A1:output(T1)
output(T2)
input(T1)
input(T2)
release(T1)
release(T2)
future(5,A2)
A2:output(T2)
input(T2)
release(T2)
future(5,A1)
A1:output(T1)
output(T2)
input(T1)
input(T2)
release(T1)
release(T2)
future(5,A2)
T1
T2
T2
Input
Input
Output
Output
Output
Input
0 1 2 3 4 5 6 7 8 9 10
System

Schedule-Carrying Code

Schedule-Carrying Code	*Schedule-Carrying Code*	*Schedule-Carrying Code*
E+S Machine	*E+S Machine*	*E+S Machine*
POSIX Threads	*Microkernel*	*RT Ethernet*
Linux	*StrongARM*	*RT Linux*
[EMSOFT, 2003]	[VEE, 2005]	[LCTES, 2005]

Current Projects



The JAviator Project

javiator.cs.uni-salzburg.at

- Goal:
  - ➡ enable high-performance real-time code, e.g., flight control software, to be written *entirely* in Java
- Challenge:
  - ➡ enable *submillisecond, predictable* real-time behavior while maintaining as much *original* Java semantics as possible

The JAviator Platform

- the JAviator is a quadrotor UAV
- we are currently building our own prototype w/ 500g payload
- single XScale 400MHz CPU w/ Bluetooth onboard running RT Linux and IBM's J9 JVM
- 3 gyros, 1 3D compass, 5 ultrasonic sensors, 4 brushless motors, 1 LiPoly battery

Collaboration

see also [EMSOFT 2005]

- IBM (3 staff researchers lead by D.F. Bacon):
  - ➡ design and implementation of high-performance real-time garbage collection (Metronome)
- Our team (2 PhD students):
  - ➡ design and implementation of a LET-based concurrency model that extends Java's notion of "write-once-run-anywhere" to the temporal domain

Exotasks and Pods

- *exotasks* are individually garbage-collected software tasks that communicate by message passing through so-called *pods*

- exotasks may allocate memory and mutate their pointer structures

- exotasks may neither observe global mutable state nor their mutable state may be observed

- pods connect exotasks and “send-data-by-garbage-collection”

Implementation

- each exotask has its own private heap and fully preemptable garbage collector
- exotasks will be compiled into E code (the timing part) and dynamically scheduled and garbage collected (the functional part)
- exotasks with LETs may also be compiled into *G code* (schedule-carrying code extended by garbage-collecting instructions [M. Harringer, MSc Thesis, University of Salzburg, 2005])

The TAP Project

tap.cs.uni-salzburg.at

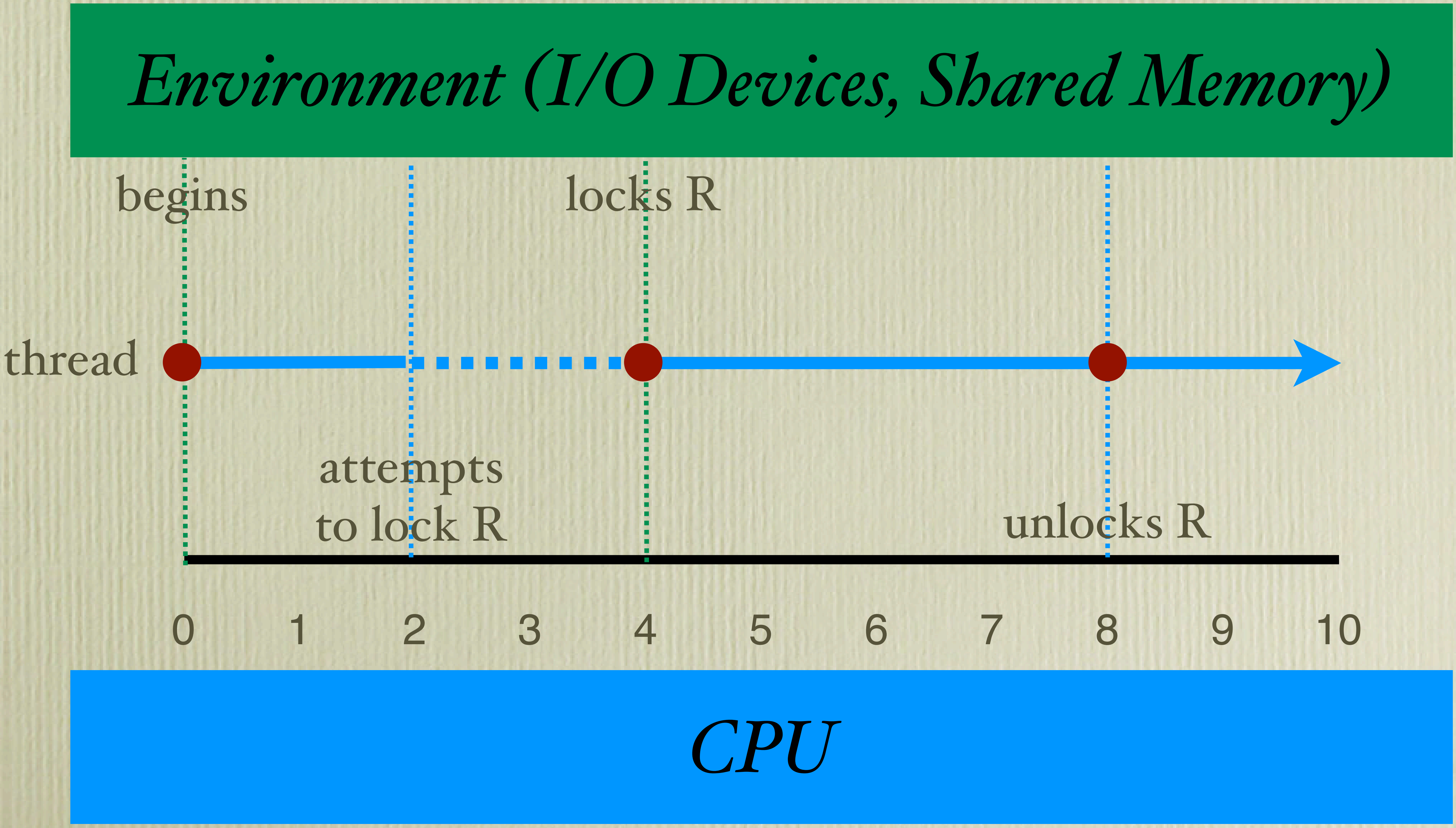
- Goal:
  - ➡ enable *efficient*, *predictable*, and *compositional* concurrent programming of high-performance servers such as file and web servers
- Approach: “Threading by Appointment”
  - ➡ separate I/O behavior from CPU scheduling, and control I/O behavior explicitly

Threading by Appointment

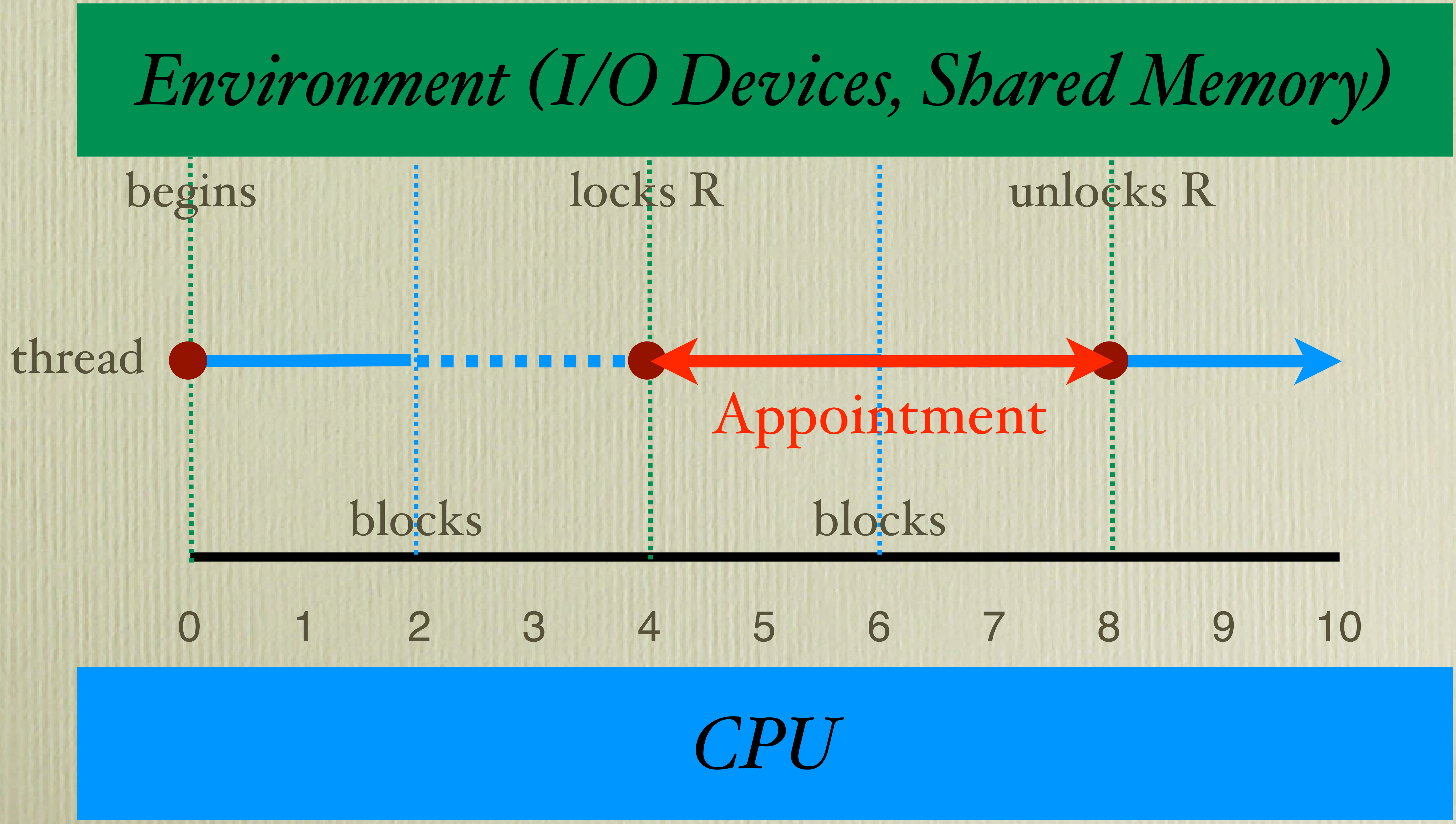
[Monterey Workshop, 2004]

- TAP threads must have *appointments* to "communicate", e.g., to invoke system calls
- Appointments determine the *order* and *time instant* when to "communicate", e.g., to execute system calls
- Appointments are made by the TAP runtime system transparently under a POSIX-compliant API according to a given *TAP policy*

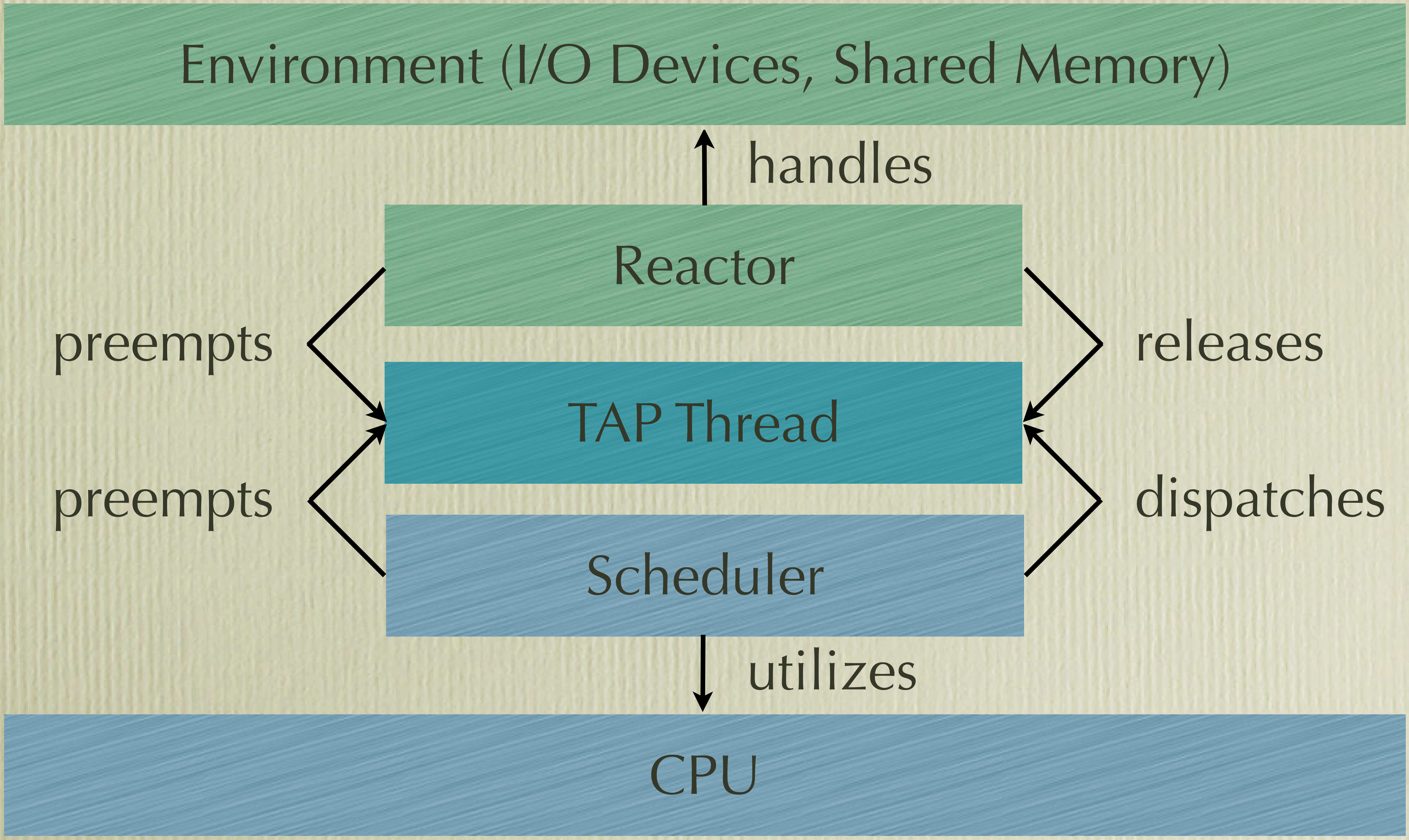
Example: Locking



Example: TAP Locking



Reactor vs. Scheduler



Traffic Shaping...

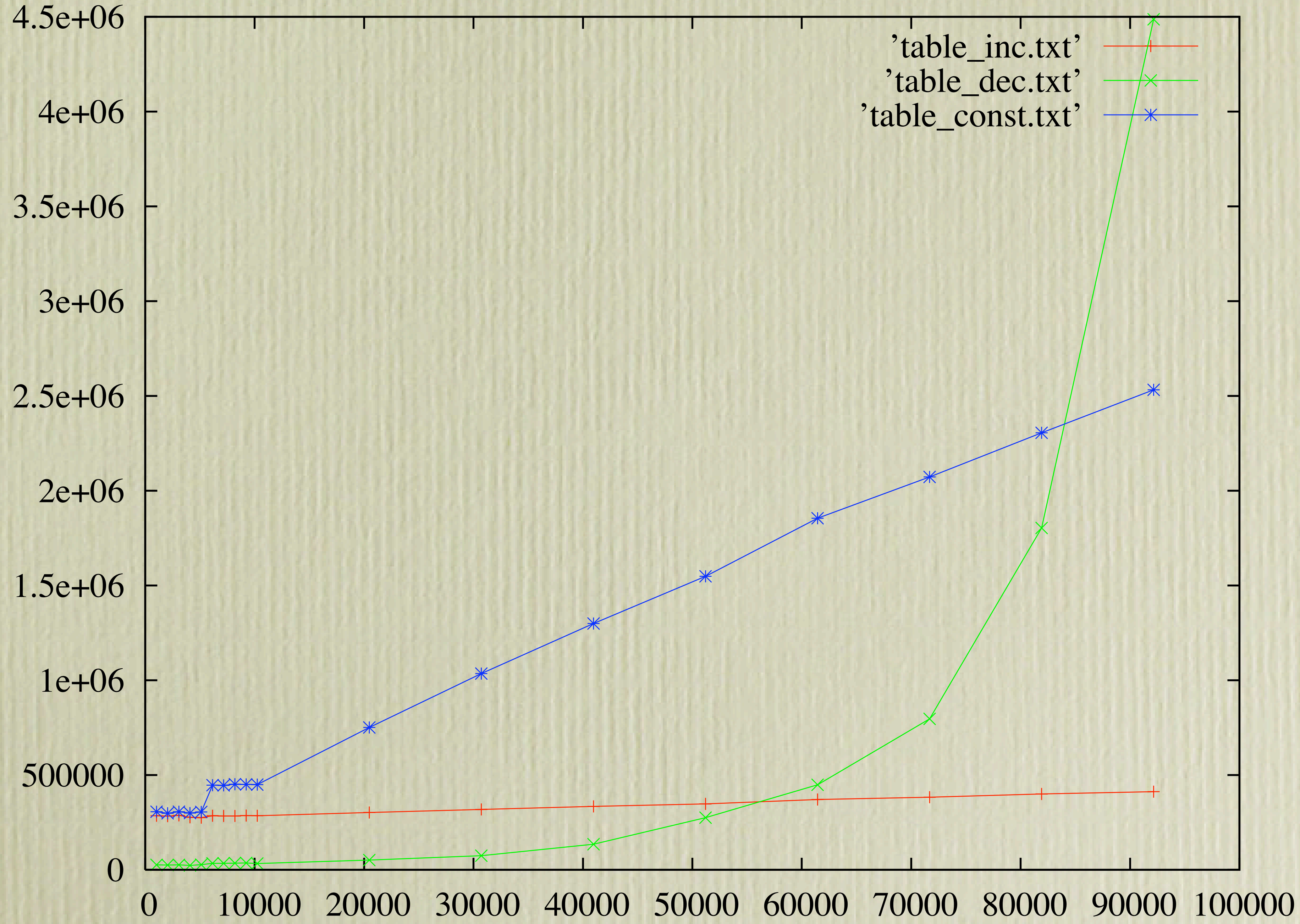
- ...controls volume, throughput, and latency of network traffic, using:
- queueing disciplines such as:
  - the *leaky-bucket* algorithm (creates fixed transmission rate on varying flows)
  - the *token bucket* algorithm (allows bursts while limiting average transmission rates)
- classification schemes: *interactive* vs. *bulk* traffic

Traffic Shaping System Calls

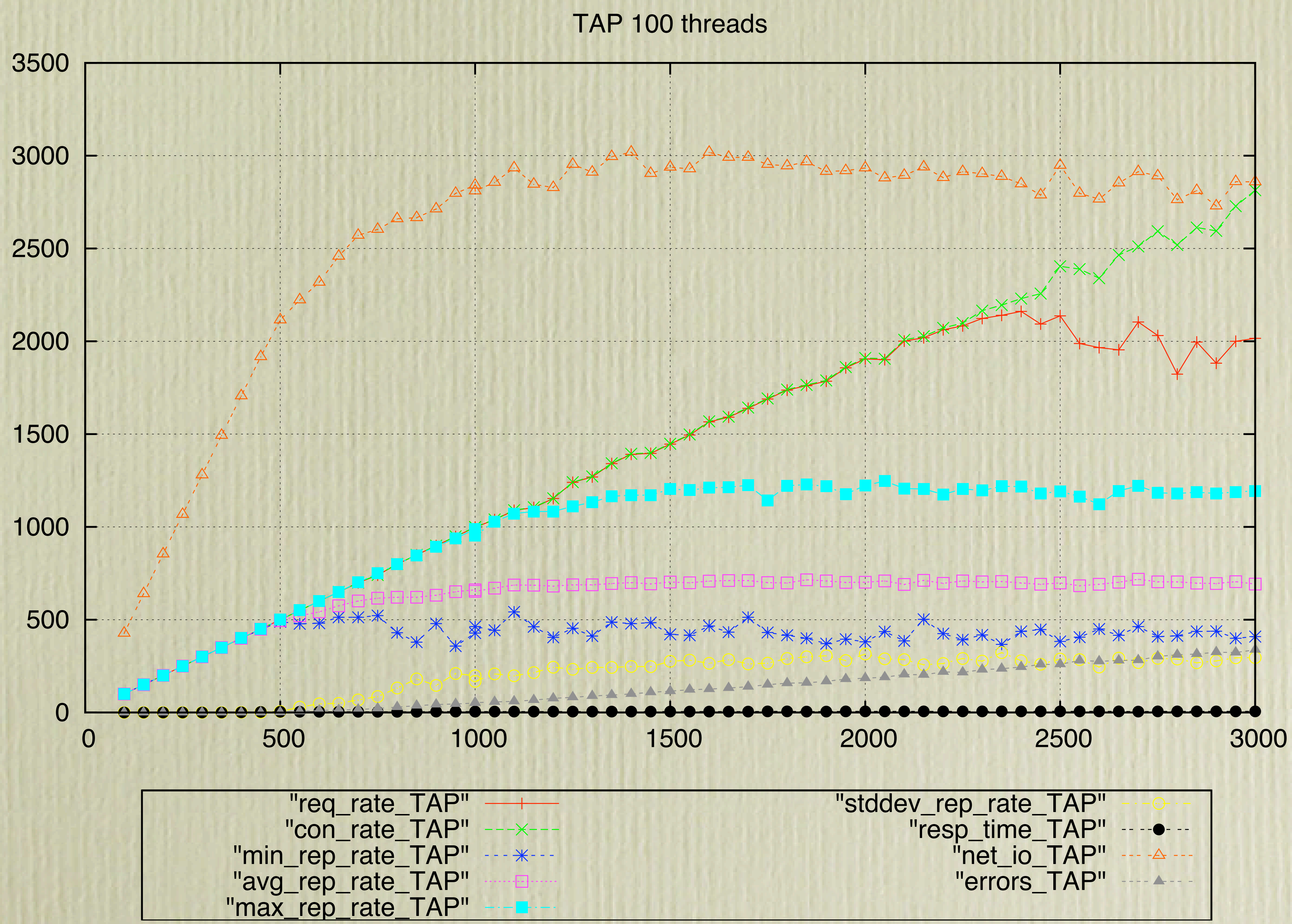
- system call = network packet
- appointment policy = queueing discipline
- thread behavior = classification scheme
  - e.g., "short-running" threads may have higher "appointment priority" than "long-running" threads

➡ improves latency of interactive threads

Latency

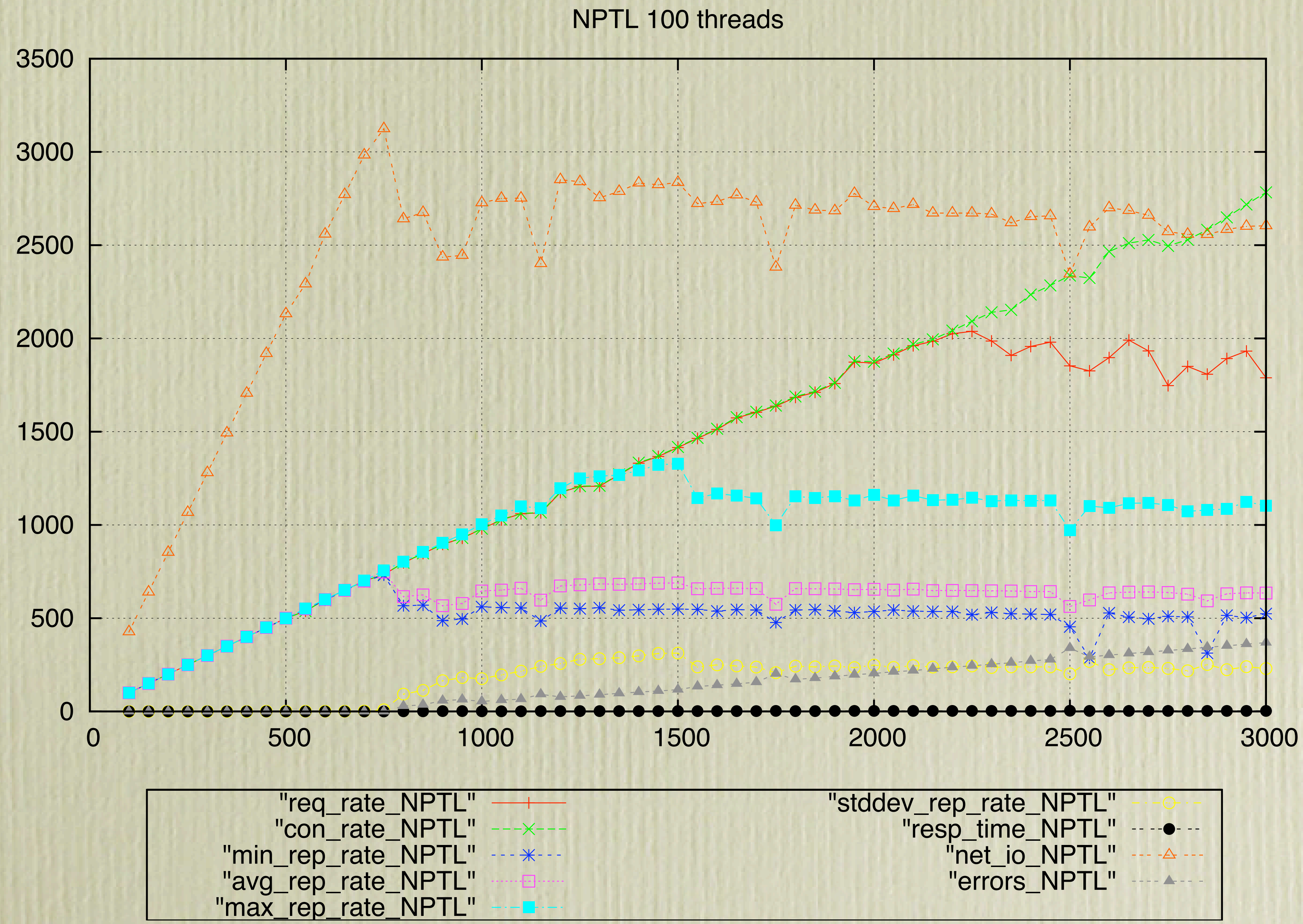

4.5e+06
4e+06
3.5e+06
3e+06
2.5e+06
2e+06
1.5e+06
1e+06
500000
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
'table_inc.txt'
'table_dec.txt'
'table_const.txt'

Throughput



TAP 100 threads
3500
3000
2500
2000
1500
1000
500
0
0
500
1000
1500
2000
2500
3000
"req_rate_TAP"
"con_rate_TAP"
"min_rep_rate_TAP"
"avg_rep_rate_TAP"
"max_rep_rate_TAP"
"stddev_rep_rate_TAP"
"resp_time_TAP"
"net_io_TAP"
"errors_TAP"

Throughput: NPTL



Thank you