EDAF35: OPERATING SYSTEMS

MODULE 5.A

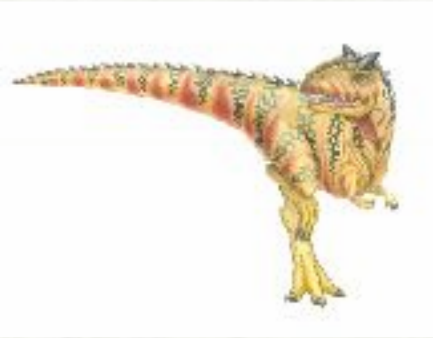
CPU SCHEDULING

CONTENTS

MODULE 5

- Scheduling concepts
- Criteria
- Algorithms
- Threads vs. process scheduling
- Multiprocessor and multicore issues
- Real-time scheduling

CHAPTER 5
(OR 6)

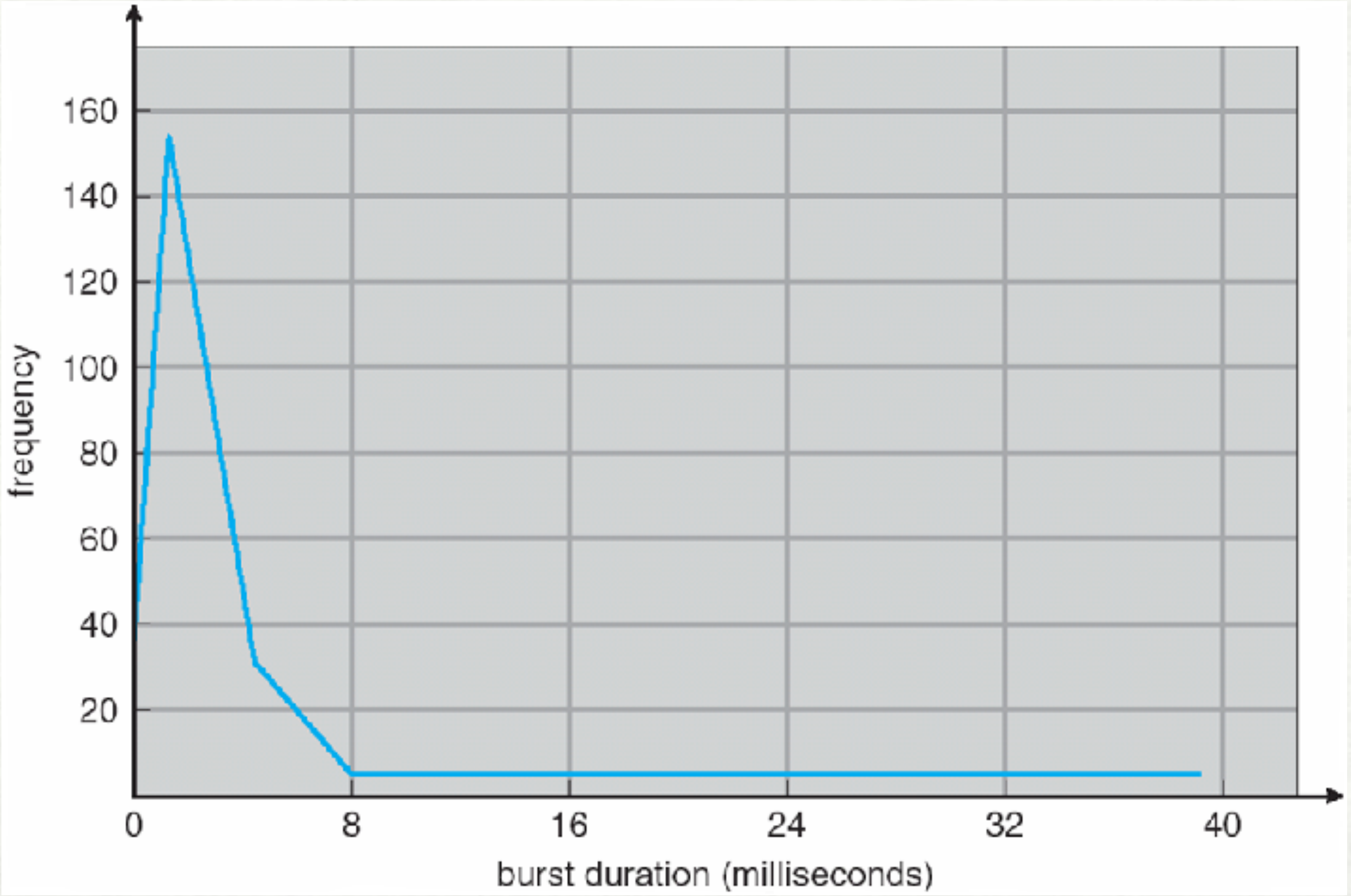


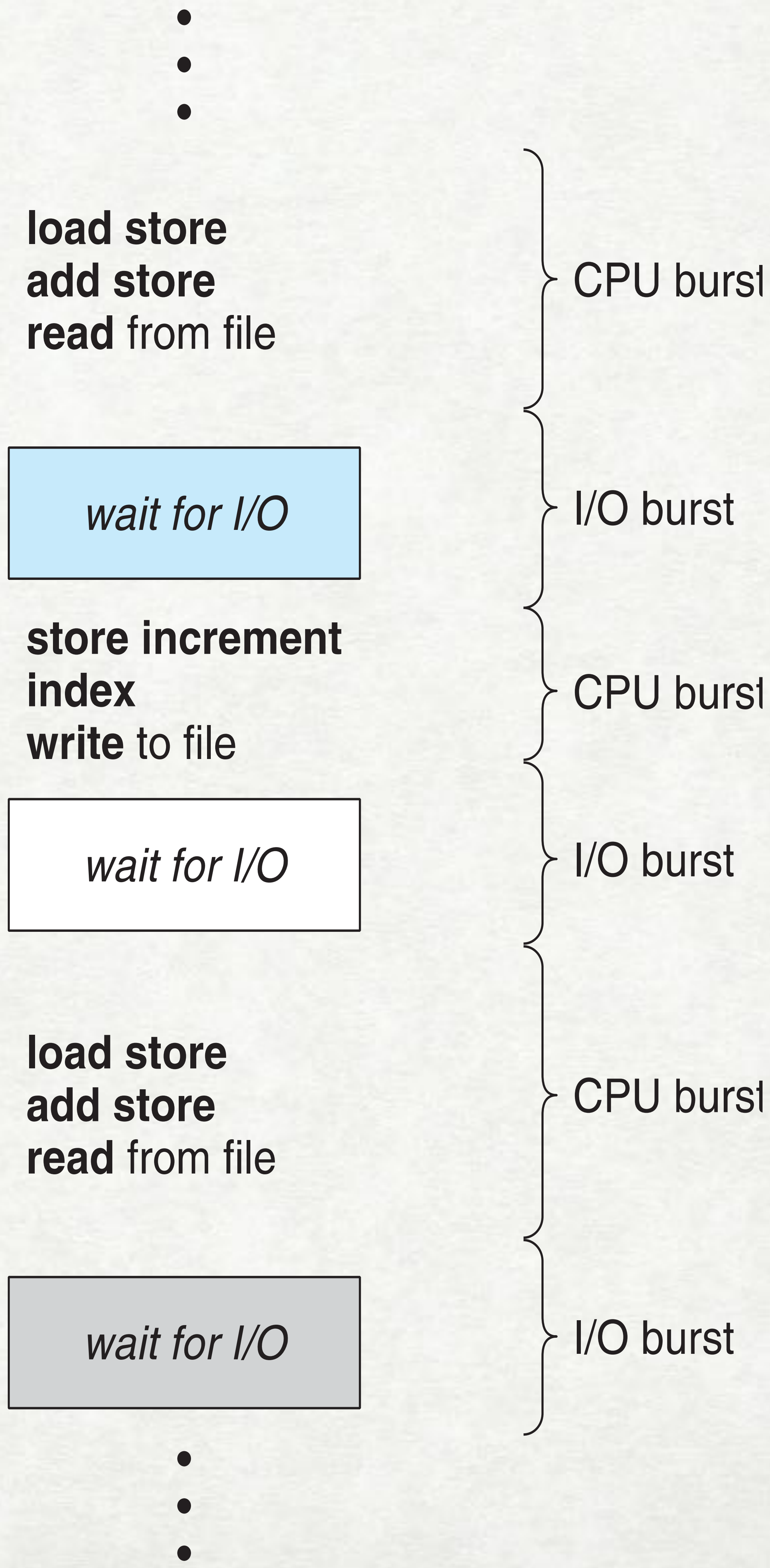
BASIC CONCEPTS

SCHEDULING

- "multiprogramming" — for maximizing CPU utilization
- typical program: sequence of CPU— I/O bursts
- CPU-bound vs. I/O-bound processes

e.g. CPU bursts histogram





load store
add store
read from file

CPU burst

wait for I/O

I/O burst

store increment
index
write to file

CPU burst

wait for I/O

I/O burst

load store
add store
read from file

CPU burst

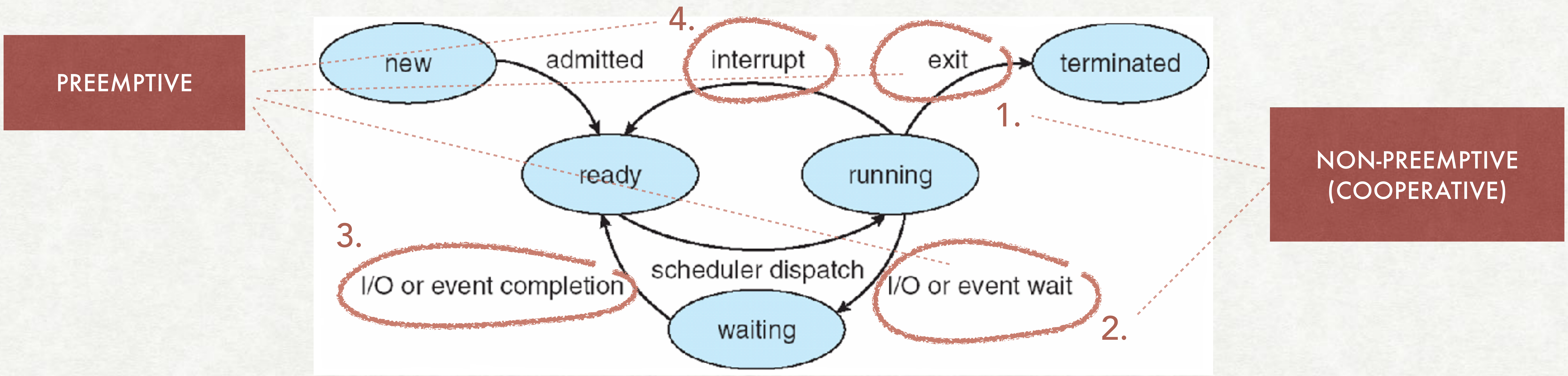
wait for I/O

I/O burst

BASIC CONCEPTS (II)

SCHEDULING

- **short-term scheduler** — choose the next to run from the “ready” queue. **When?**



PREEMPTIVE

NON-PREEMPTIVE (COOPERATIVE)

- **dispatcher** — gives control of the CPU to the selected process:
  - switch context, switch to user mode, jump to user PC (“dispatch latency”)

CRITERIA

SCHEDULING

- **CPU utilization** — busy ratio for the CPU
- **throughput** — # processes completed per time unit
- **turnaround time** — process submit to complete time
- **waiting time** — time spent as "ready"
- **response time** — submit to first output time

Are these independent?

Which to maximize and which to minimize?

SCHEDULING ALGORITHMS

- First-Come, First-Served (FCFS)
- Shortest-Job-First (SJF)
- Priority
- Round-Robin
- Multilevel Queue
- Multilevel Feedback Queue

FIRST-COME, FIRST-SERVED (FCFS)

SCHEDULING

Process	CPU Burst time
P_1	24
P_2	3
P_3	3

Arrive order: 1, 2, 3

Waiting time for 1, 2, 3?   Average waiting time?

Arrive order: 2, 3, 1

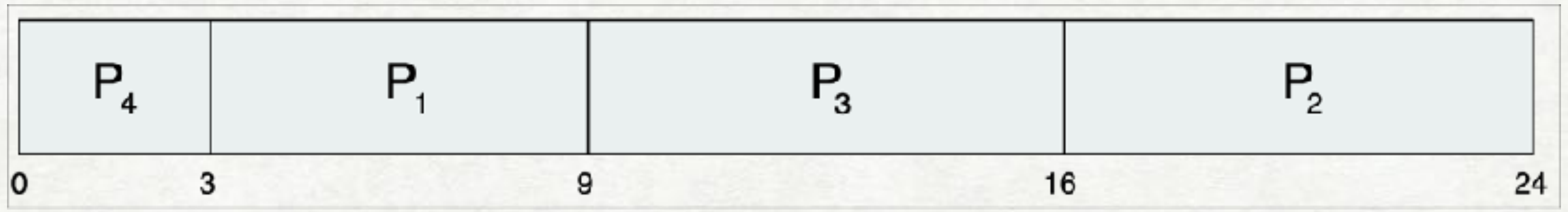
Convoy effect — short processes stuck after a long one (non-preemptive!)

SHORTEST-JOB-FIRST (SJF)

SCHEDULING

Process	CPU Burst time
P_1	6
P_2	8
P_3	7
P_4	3

Execute the shortest job first!

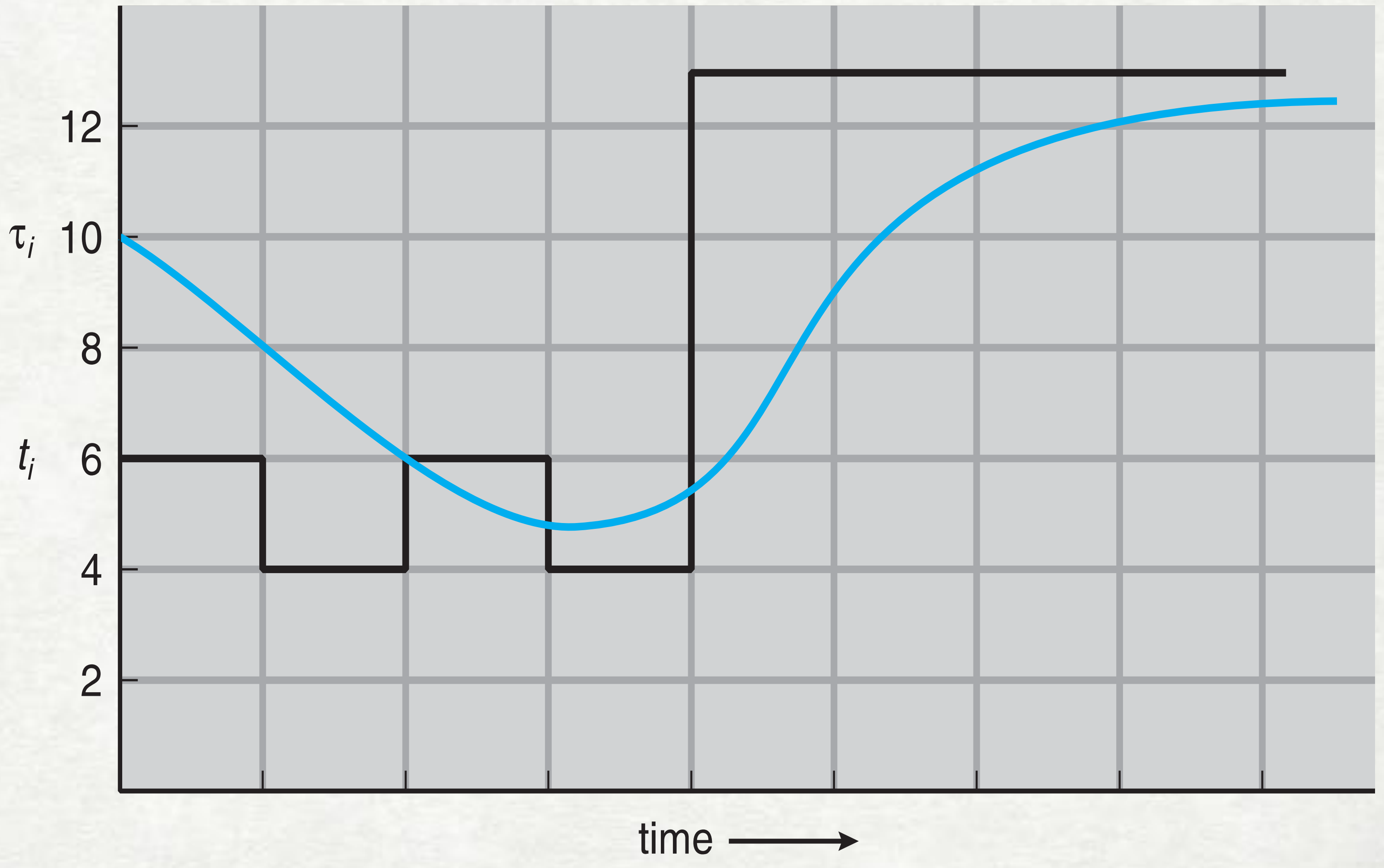


Waiting time for each? Average waiting time?

predict —
e.g exponential average:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

where $\alpha = 0..1$ (here 0.5)



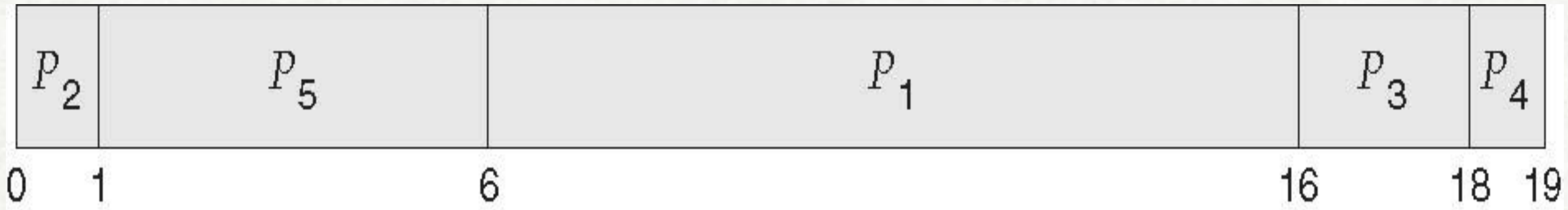
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Practical issue —
how to find out the burst times?

PRIORITY
SCHEDULING

Process	CPU Burst time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

ASSOCIATE A NUMBER (PRIORITY) WITH EACH –
here, smallest number means highest priority



Waiting time for each? Average waiting time?

☆ **reformulate SJF as priority scheduling — how?**

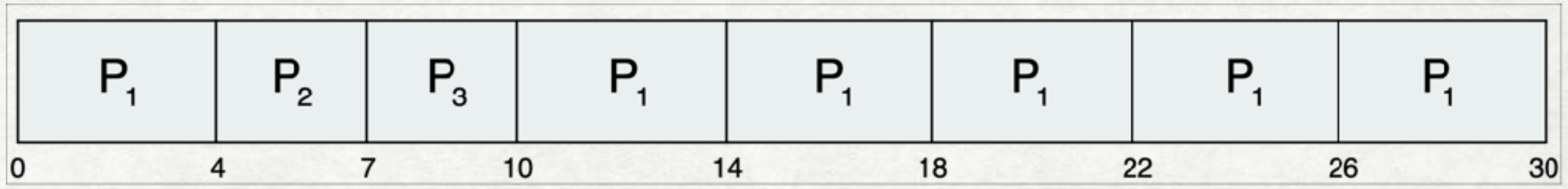
- preemptive or not
- problem: **starvation** (for low priority) — solution: **aging** (increase priority over time)

ROUND ROBIN (RR)

SCHEDULING

- time **quantum** or time slice (*q*) — execute, interrupt, **preempt**, repeat
- with *N* processes in ready queue
  - each gets *1/N* processor time
  - max wait is *(N-1)q*
- how to choose *q*?
  - very large — FIFO
  - very small — context switch overhead becomes high

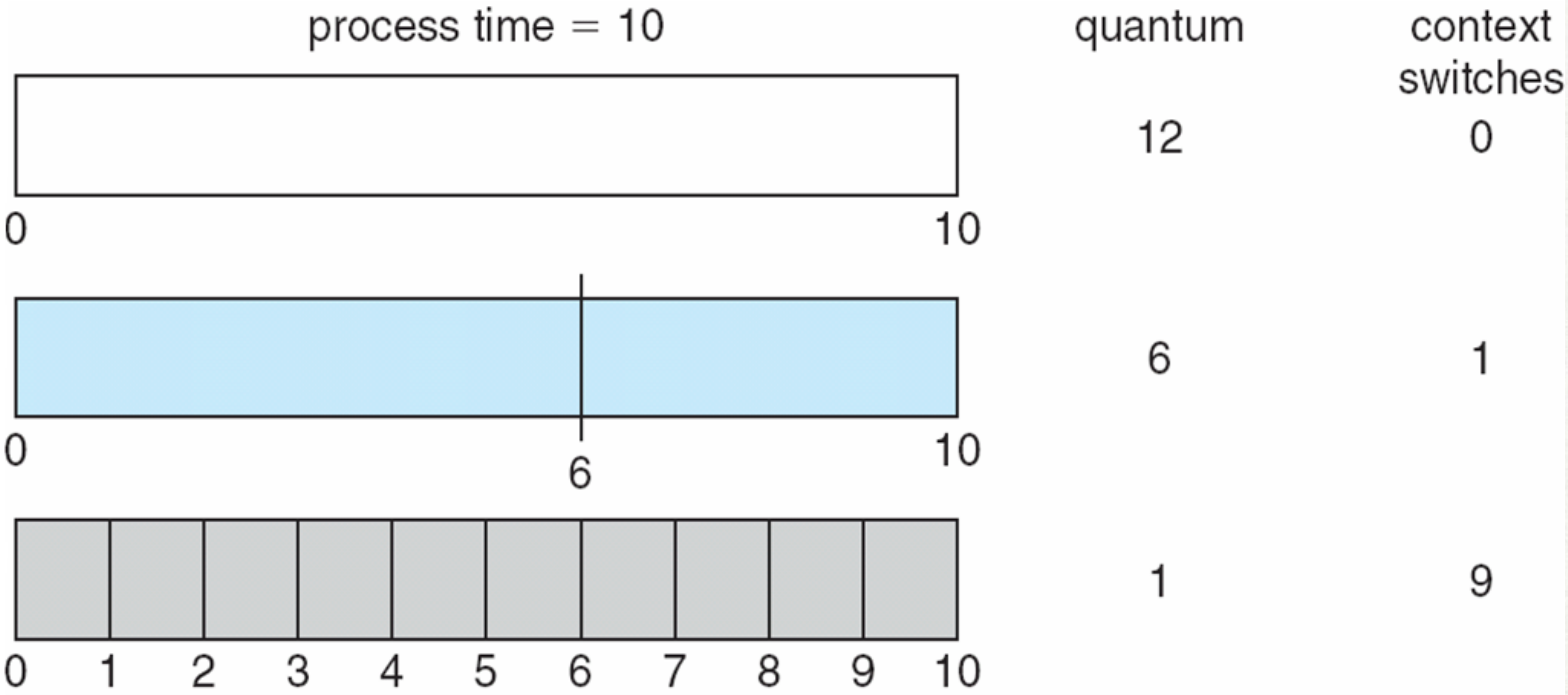
Process	CPU Burst time
P_1	24
P_2	3
P_3	3



E.g. compared to SJF: larger average turnaround (more wait), better response (starts fast).

TIME QUANTUM AND CONTEXT SWITCHES

SCHEDULING



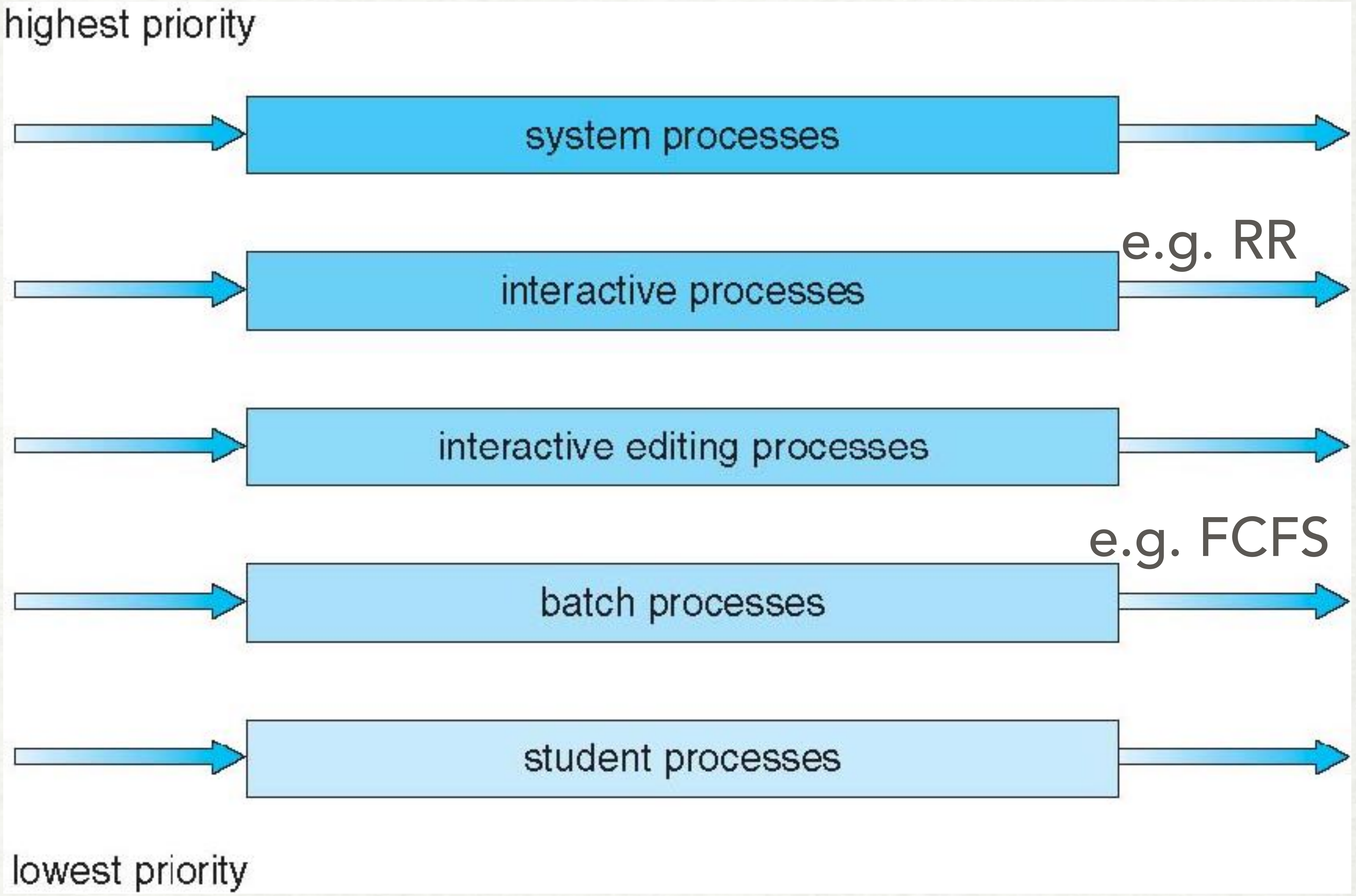
MULTILEVEL QUEUE

SCHEDULING

- several queues,
  different priorities,
  different policies

- permanently assign a process to a queue

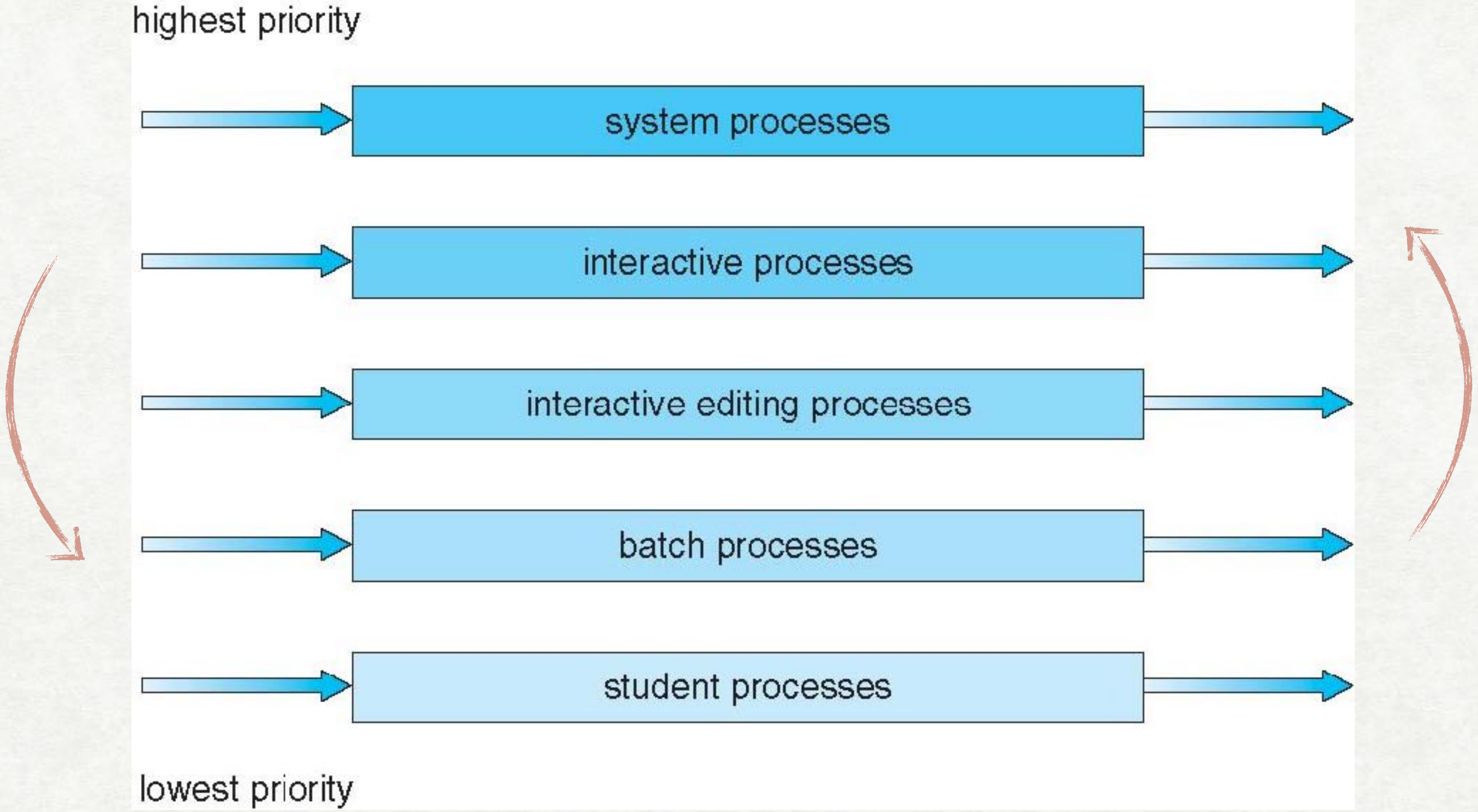
- to choose among queues:
  - fixed priority
  - time slice
    (e.g. 80% interactive, 20% batch)



MULTILEVEL FEEDBACK QUEUE

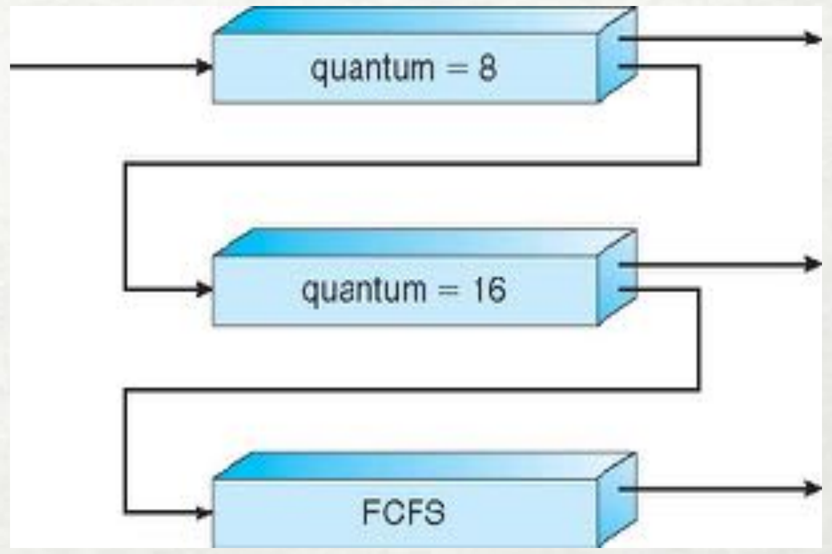
SCHEDULING

- processes can move between queues



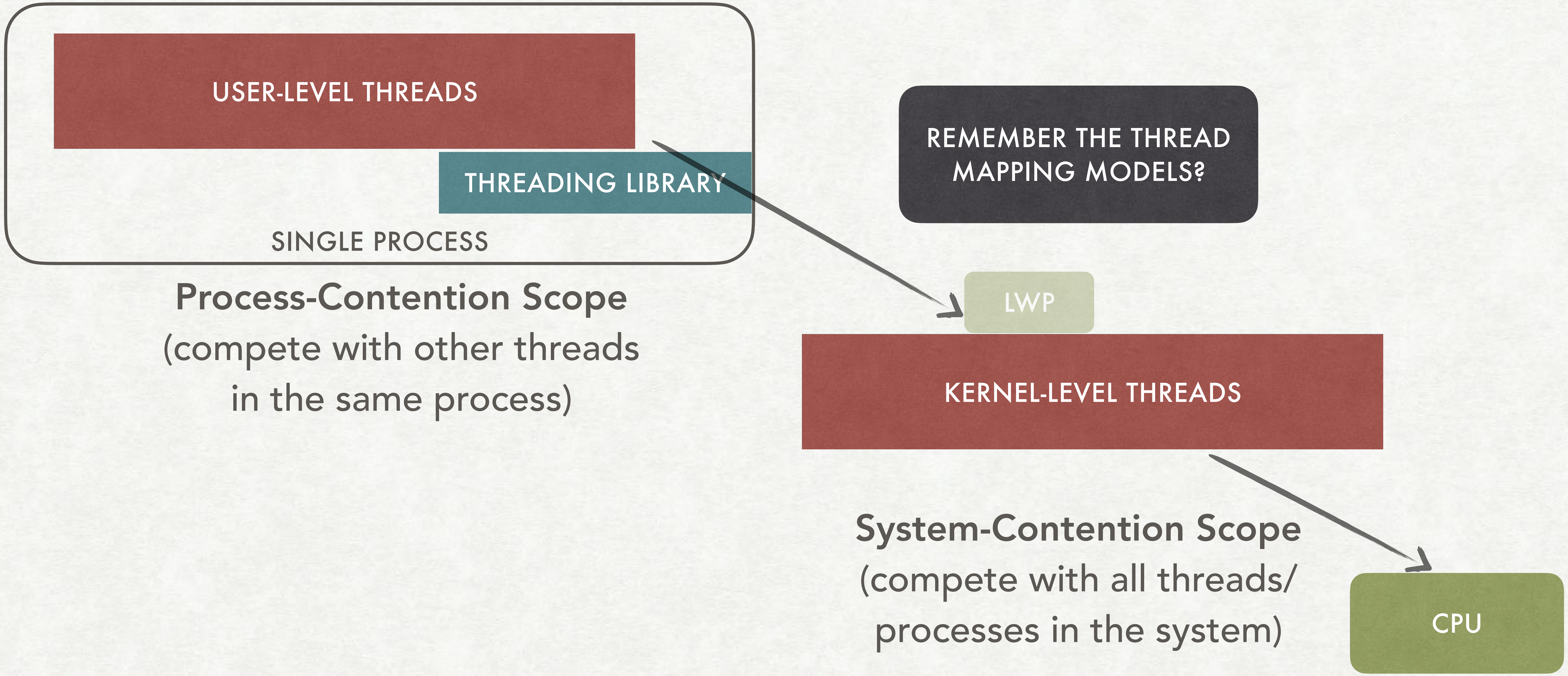
Demote process

Upgrade process



THREAD

SCHEDULING



USER-LEVEL THREADS

THREADING LIBRARY

SINGLE PROCESS

Process-Contention Scope
(compete with other threads
in the same process)

REMEMBER THE THREAD
MAPPING MODELS?

LWP

KERNEL-LEVEL THREADS

System-Contention Scope
(compete with all threads/
processes in the system)

CPU

SCHEDULING PTHREADS

EXAMPLE

- PTHREAD_SCOPE_PROCESS vs. PTHREAD_SCOPE_SYSTEM (limited by the OS)

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

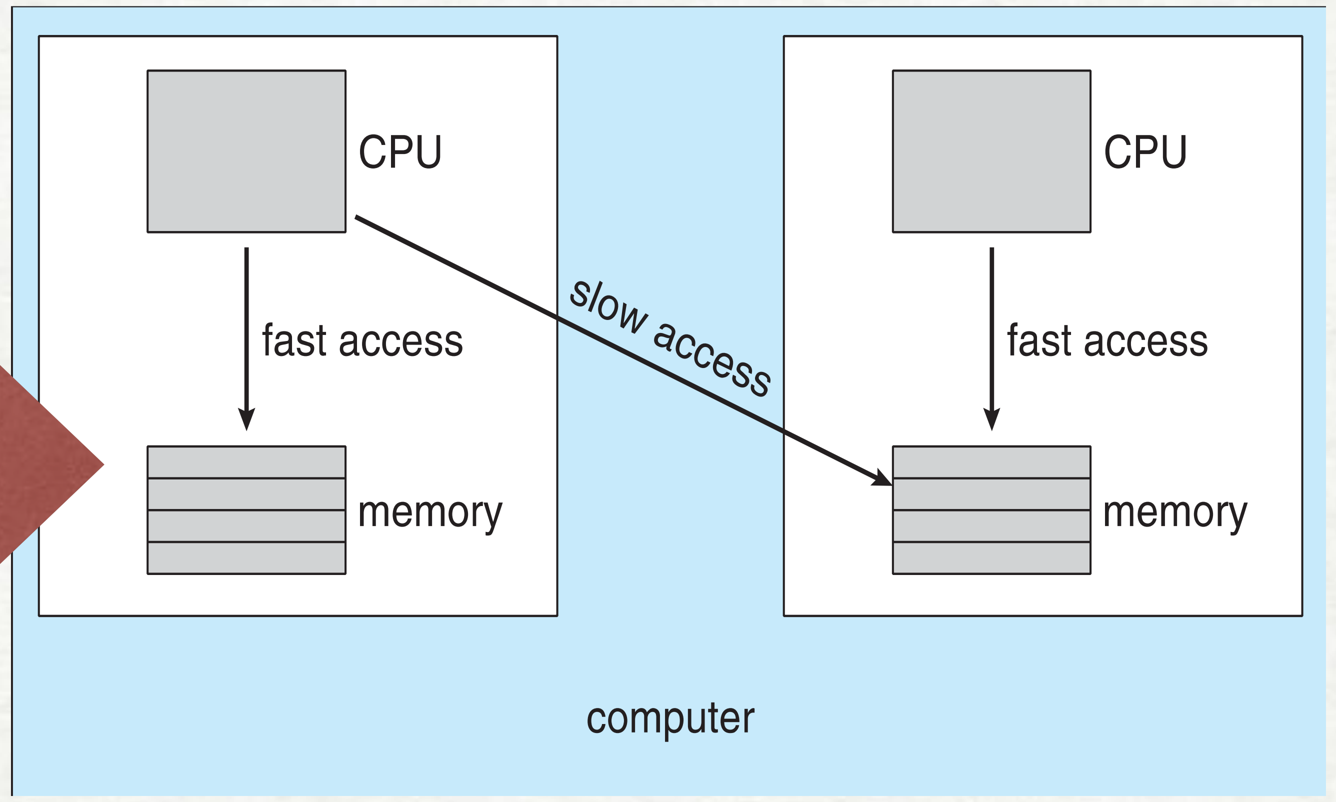
MULTI-PROCESSOR SYSTEMS

SCHEDULING

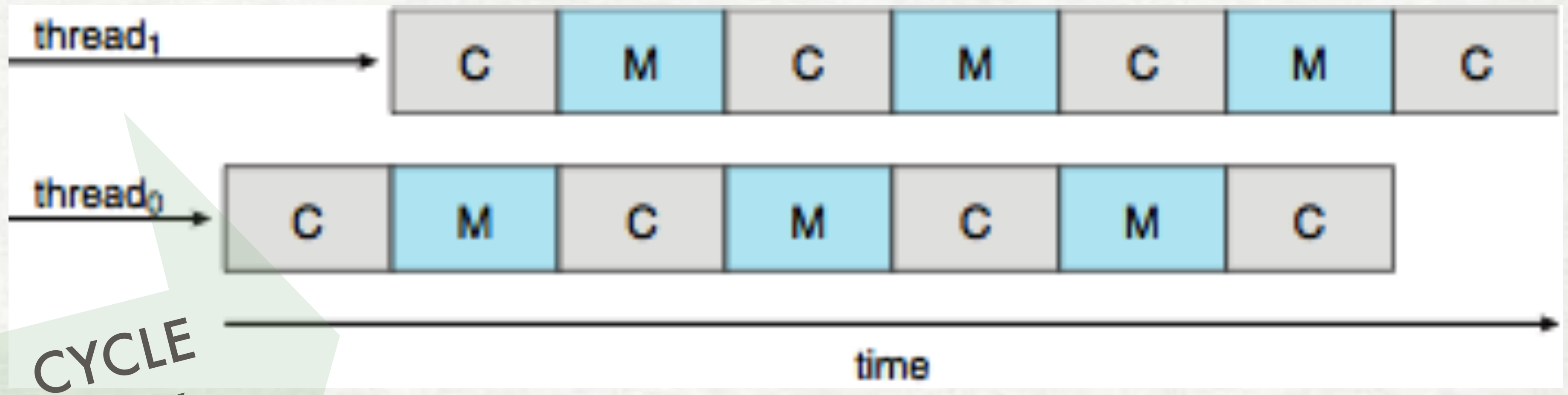
- homogeneous (similar processors)
- **symmetric vs. asymmetric** multiprocessing:
  who does the scheduling and other system activities?
- **processor affinity**: where to run a task?
  soft affinity (recommendation), hard affinity (rule)
- **load balancing**: distribute the workload evenly
  push vs. pull migration
- **multithreaded** multicore processors:
  coarse-grained vs. fine-grained multithreading

PLAYS A ROLE

NUMA



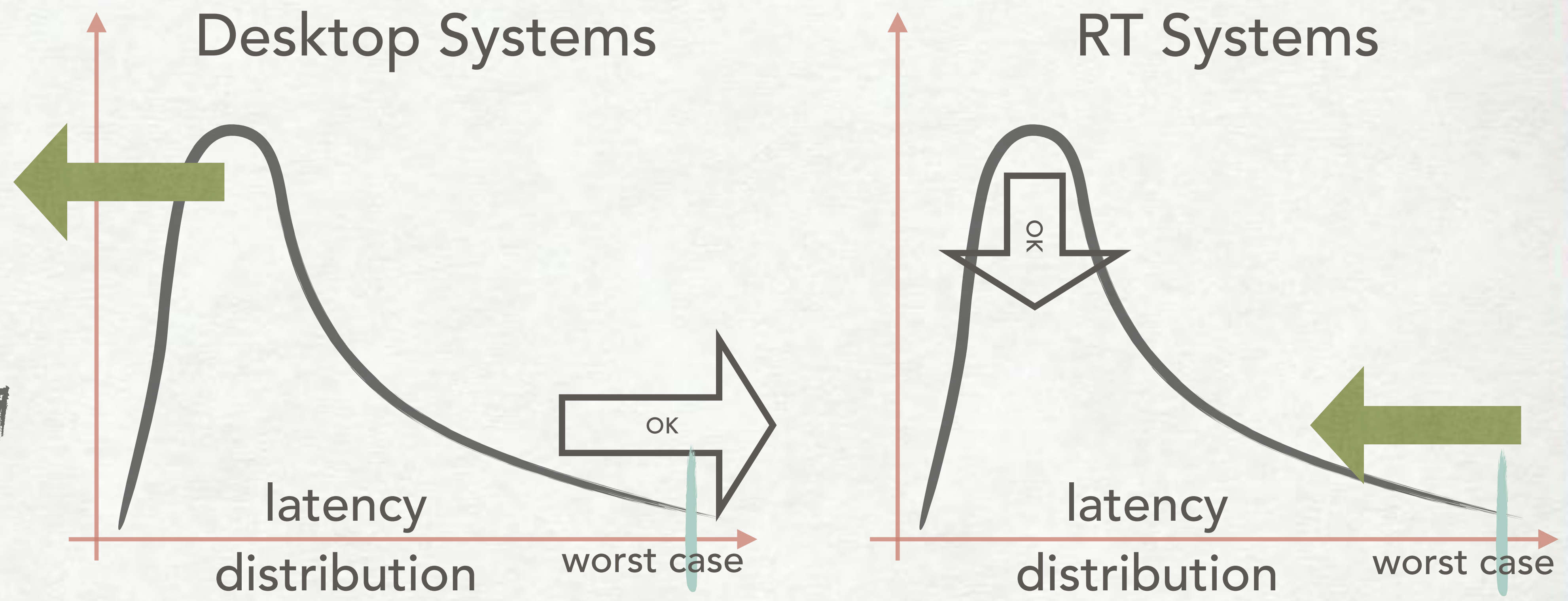
INSTR. CYCLE BOUNDARY

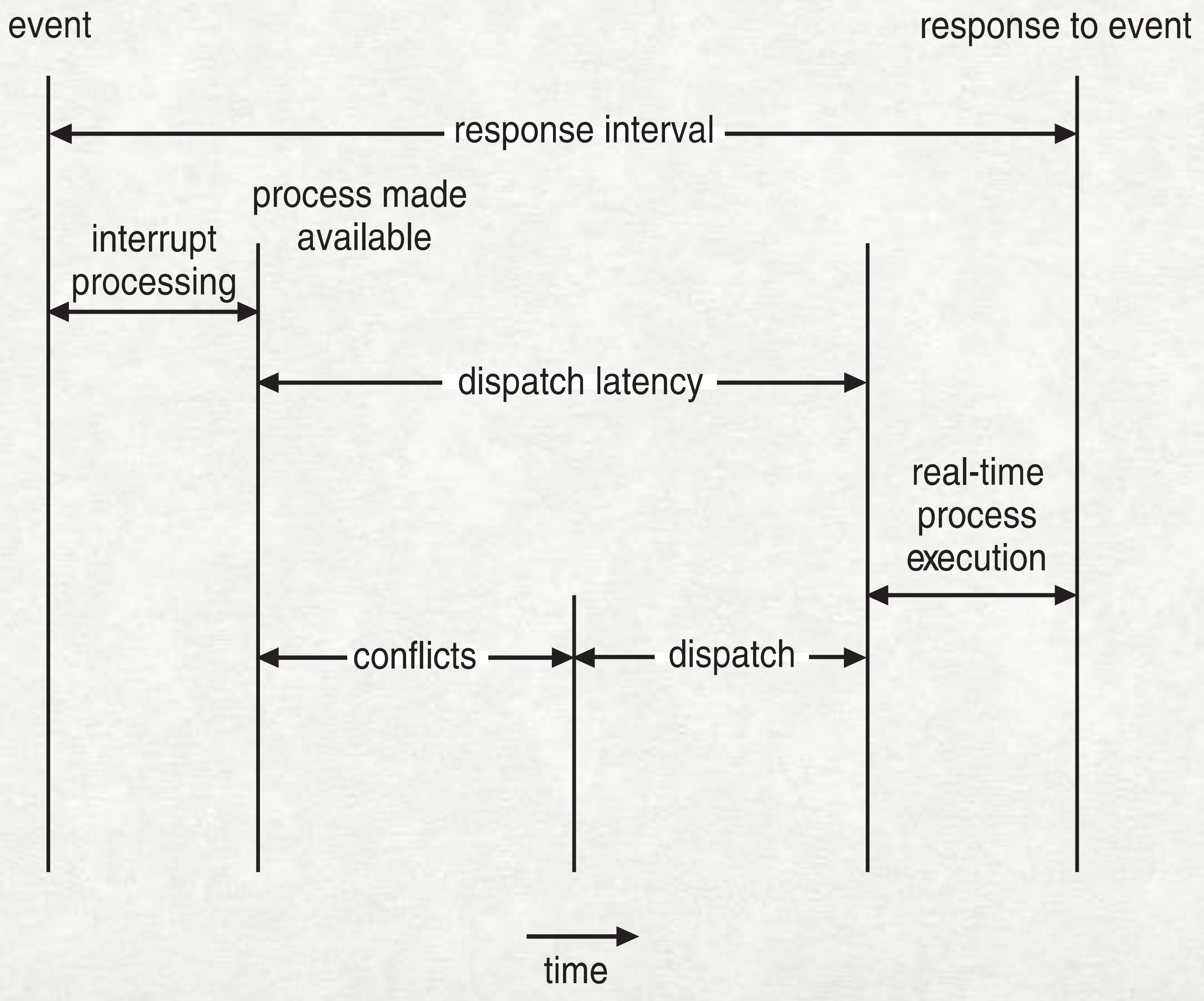


Hw managed threads

REAL-TIME SCHEDULING

- **soft vs. hard** real-time systems: <u>meeting deadlines</u>
- Time-predictability is key
- Optimize the worst case latency (rather than common case)
- Interrupt and dispatch latency are important
- Worst case response time guarantees

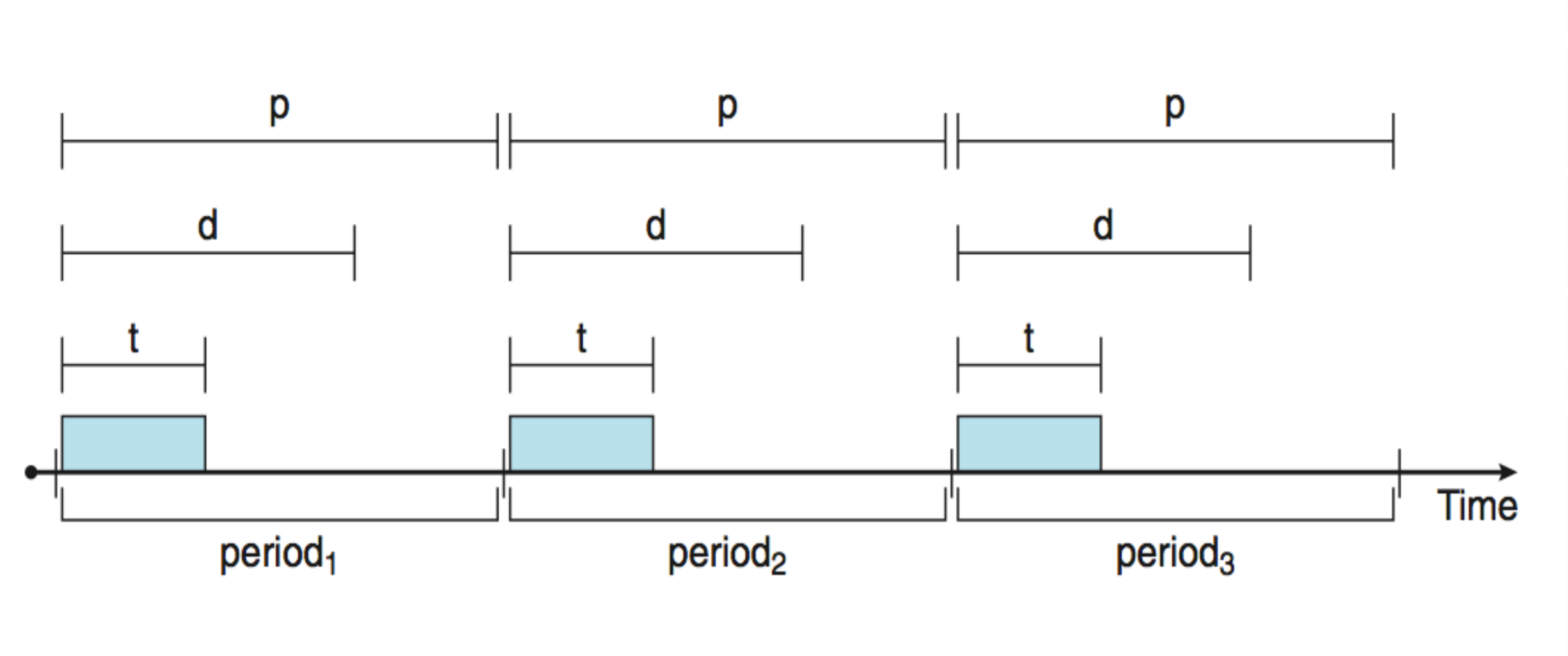




PRIORITY-BASED SCHEDULING

REAL-TIME SYSTEMS

- Special task model
  - periodic (p)
  - worst case execution time (t)
  - have deadlines (d)
    often d = p
- How to assign priorities?
  (fixed vs. dynamic, values)
- Analysis techniques? (guarantees)



assumption: independent tasks
— can be relaxed somewhat

RATE-MONOTONIC SCHEDULING (RMS)

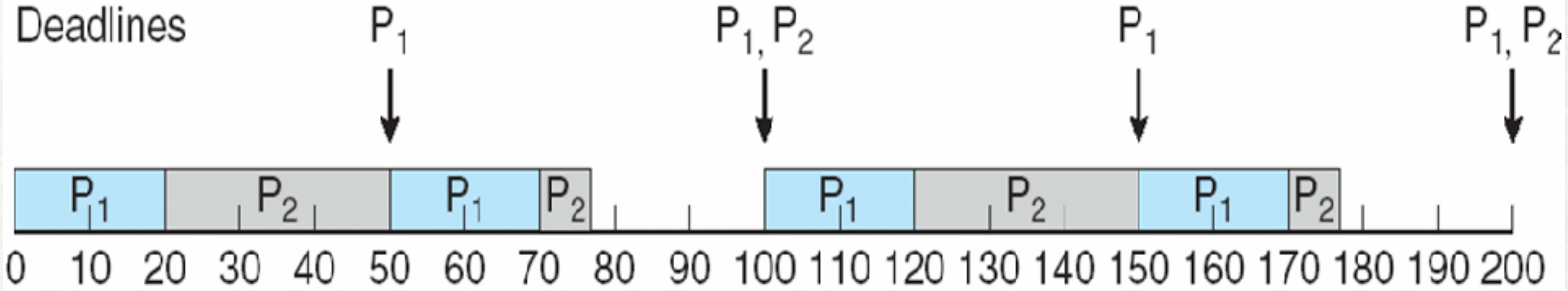
REAL-TIME

- fixed priorities (1/p)
- guarantees? static analysis
    - feasible for n tasks if CPU utilization is below a certain limit

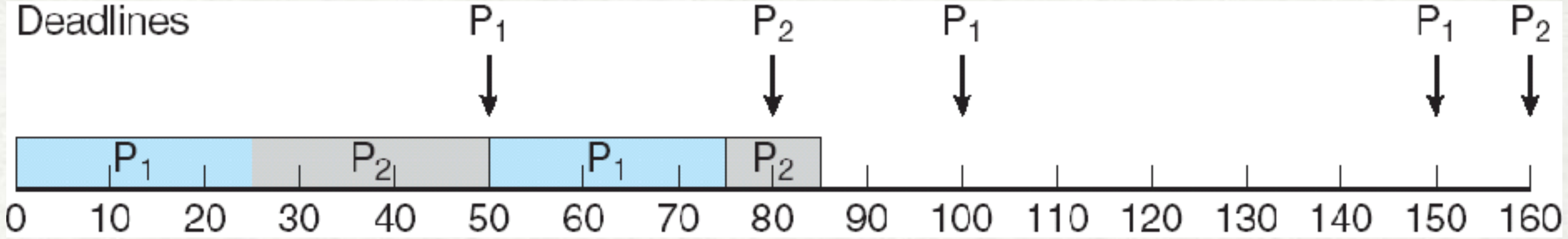
$$U = \sum_n U_n = \sum_n \frac{t_n}{p_n} \leq n(2^{\frac{1}{n}} - 1)$$

    - otherwise, may miss deadlines!
- optimal, **in its class!**

RMS for P_1 (t=20, p=50) and P_2 (t=35, p=100)



RMS for P_1 (t=20, p=50) and P_2 (t=35, p=80)

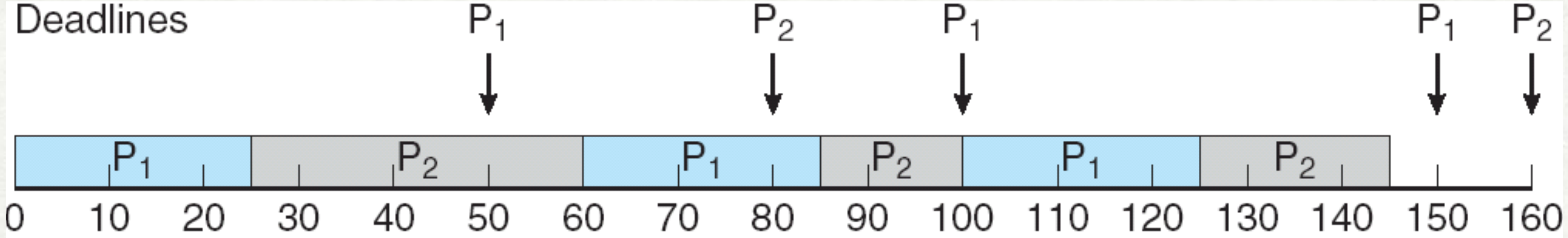


EARLIEST DEADLINE FIRST (EDF)

REAL-TIME SCHEDULING

- dynamic priorities
- guarantees: works if $U \leq 1$
- optimal: reaches 100% CPU utilization

RMS for P_1 (t=20, p=50) and P_2 (t=35, p=80)



PTHREADS SCHEDULING

POSIX 1.B STANDARD

- Two standard policies: **SCHED_FIFO**, **SCHED_RR** (same, but with time slice)
- Non-standard, OS-specific:
  SCHED_OTHER (default for the OS)
  SCHED_DEADLINE (EDF based, Linux),
  SCHED_SPORADIC (fixed budget, some RTOS)
- Reading and updating policy API:

```
pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)

pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)
```

EXAMPLE OF POSIX RT API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
```

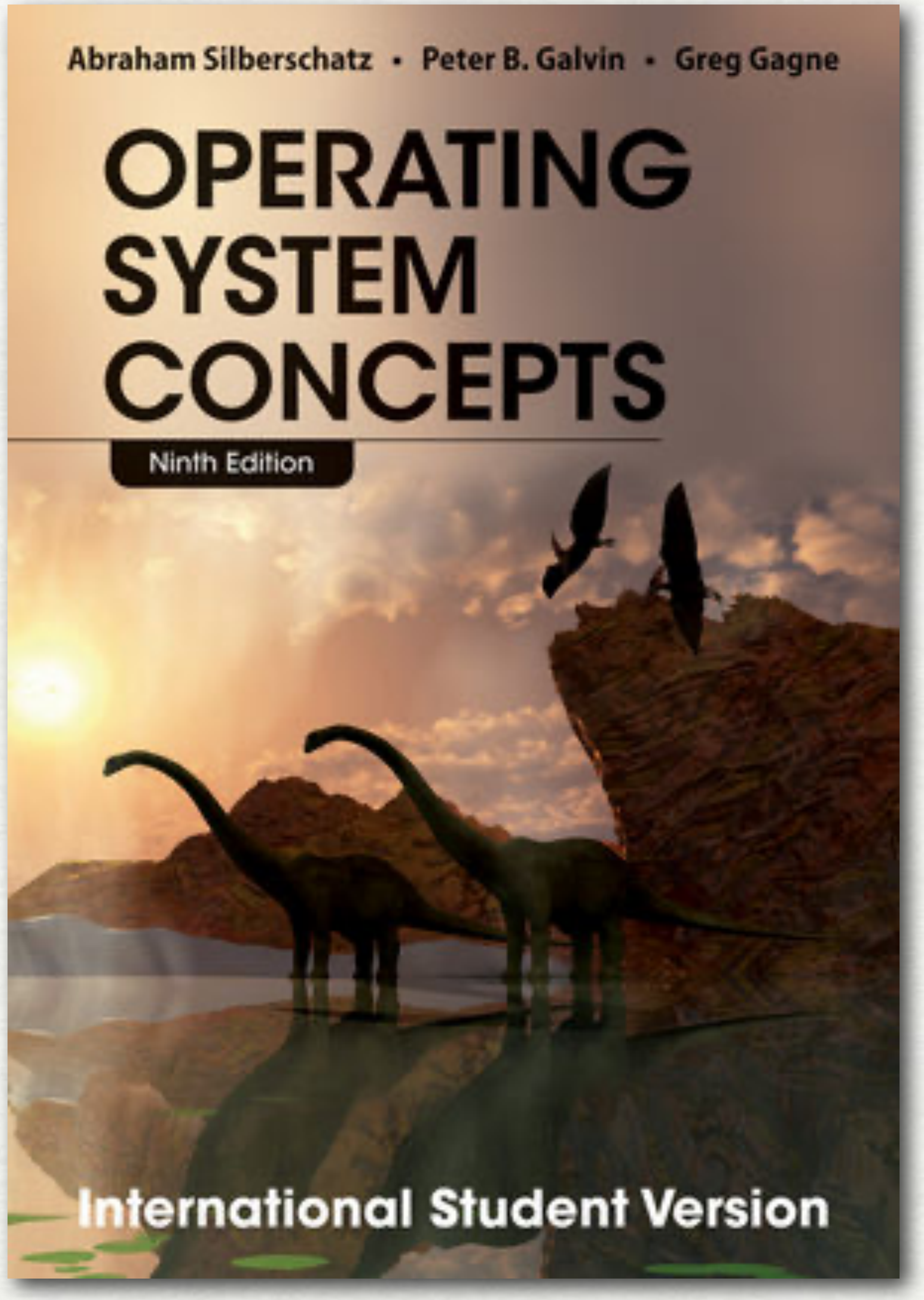
```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i], &attr, runner, NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

READ THE TEXTBOOK EXAMPLES FOR DIFFERENT OS

… AND COMPLEMENT WITH ONLINE INFORMATION

- Linux scheduling
- Windows scheduling
- Solaris scheduling



ALGORITHM EVALUATION

SCHEDULING

- How to choose a CPU-scheduling algorithm for an OS?

- Two phases:

  1. determine criteria (measure what?)
  2. evaluate according to the above (four different ways…)

1. DETERMINISTIC MODELING

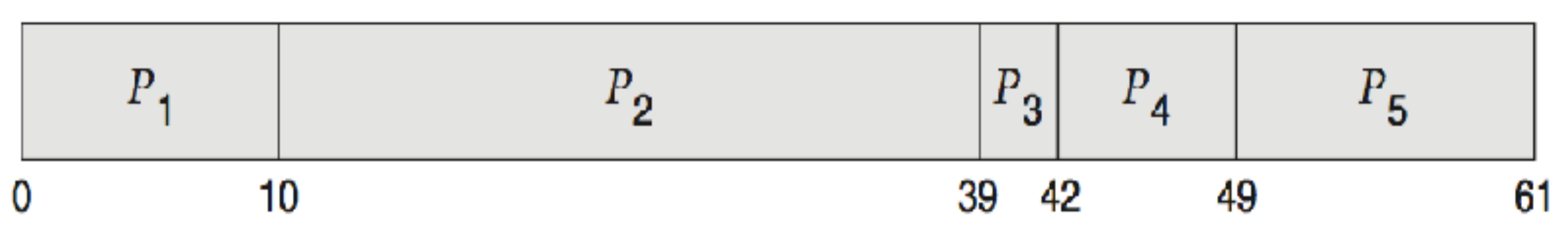
SCHEDULING EVALUATION

Process	CPU Burst time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

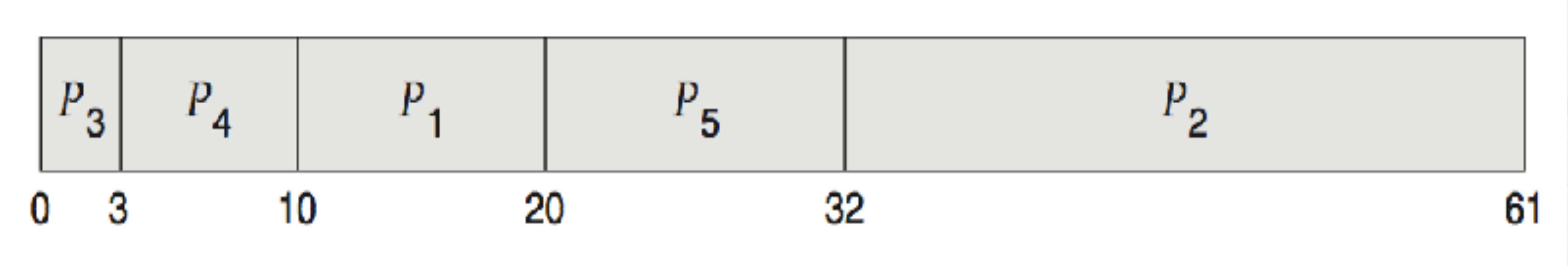
all arrive at t = 0

Criterium:
minimal average wait time?

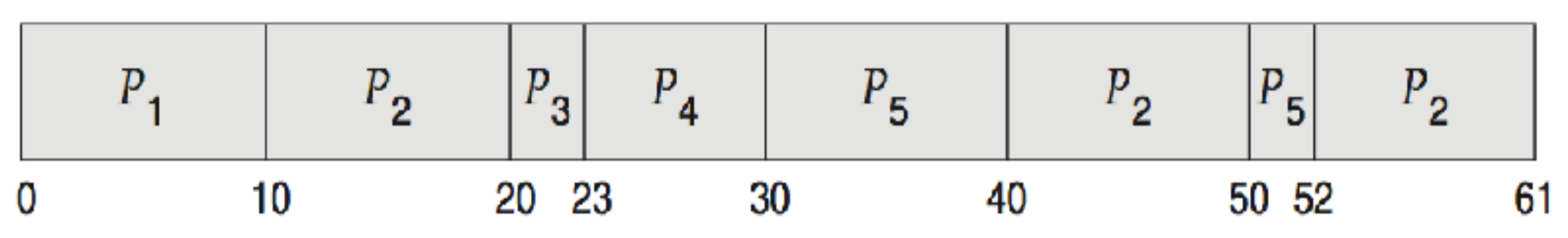
FCFS: **28**



SJF, non-preemptive: **13**



RR (q = 10): **23**



PROS: SIMPLE & FAST

CONS: REQUIRES EXACT DATA, GUARANTEES ONLY FOR THAT DATA

2. USE PROBABILISTIC MODELS

SCHEDULING EVALUATION

- queuing theory (alternatively, network calculus)
- process parameters (arrival times, duration, bursts) = probability distributions
- computing system = network of servers, each with own waiting queue
  - knows arrival rates, service rates
  - computes utilization, average waiting time, average queue length, throughput, etc.

PROS: CAN MODEL A RANGE OF DATA, KNOWN METHOD, FORMAL PROOF

CONS: SIMPLIFIED MODELS (UNREALISTIC), MAY DIVERGE (USELESS RESULTS)

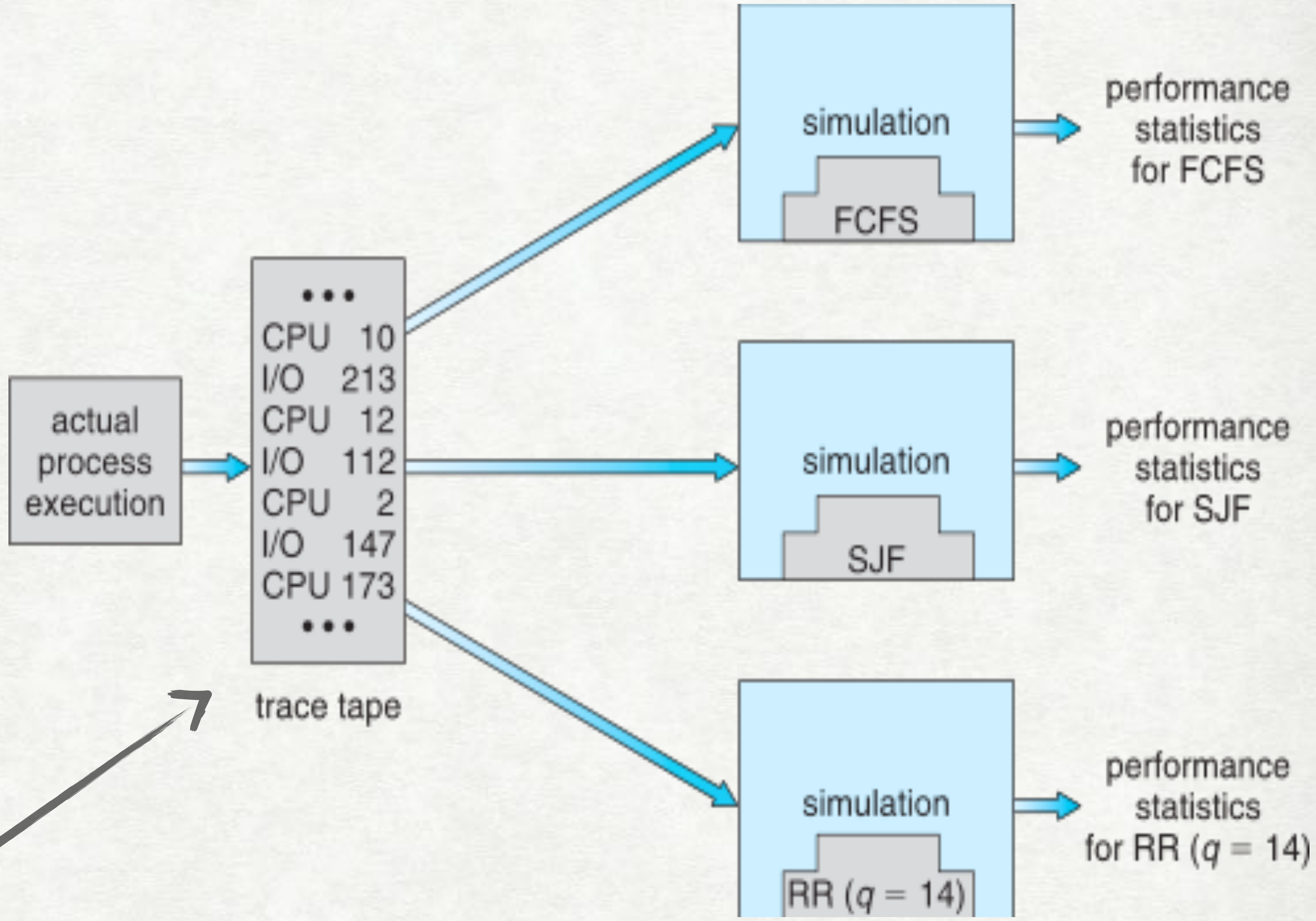
3. BY SIMULATION

SCHEDULING EVALUATION

- programmed model of the computer system
- may be at different levels of detail - thus accuracy
- may be stopped and resumed at any time
- data may come from:

  (a) randomly generated from given distributions

  (b) trace tapes recorded from real executions



PROS: MORE ACCURATE AND REALISTIC, ALLOWS FOR SOME EXPLORATION

CONS: MORE COMPLEX TO DEVELOP, DOES NOT CAPTURE ALL THE REAL DETAILS

4. BY IMPLEMENTATION

SCHEDULING EVALUATION

- Implement in a real system
- Test in real operation
- Obtain real measures in real environments
- Note: no formal proof, only “works for my tests”

PROS: MOST ACCURATE, SUITABLE FOR EXPLORATION, CATCHES THE UNEXPECTED

CONS: VERY HIGH COST, TIME-CONSUMING,
CANNOT ACCOUNT FOR ALL POSSIBLE VARIATIONS

END OF MODULE 5.A