Chapter 5
CPU Scheduling

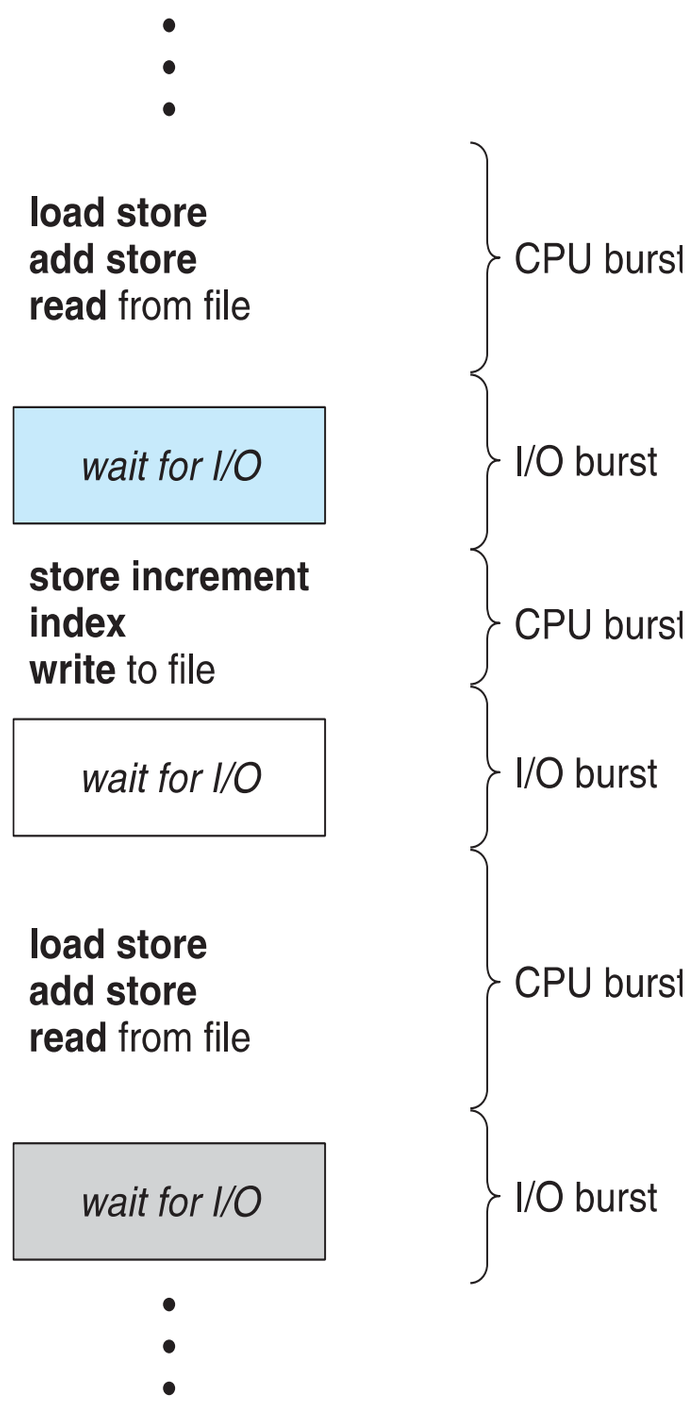
CS 3423 Operating Systems
Fall 2019
National Tsing Hua University

Overview

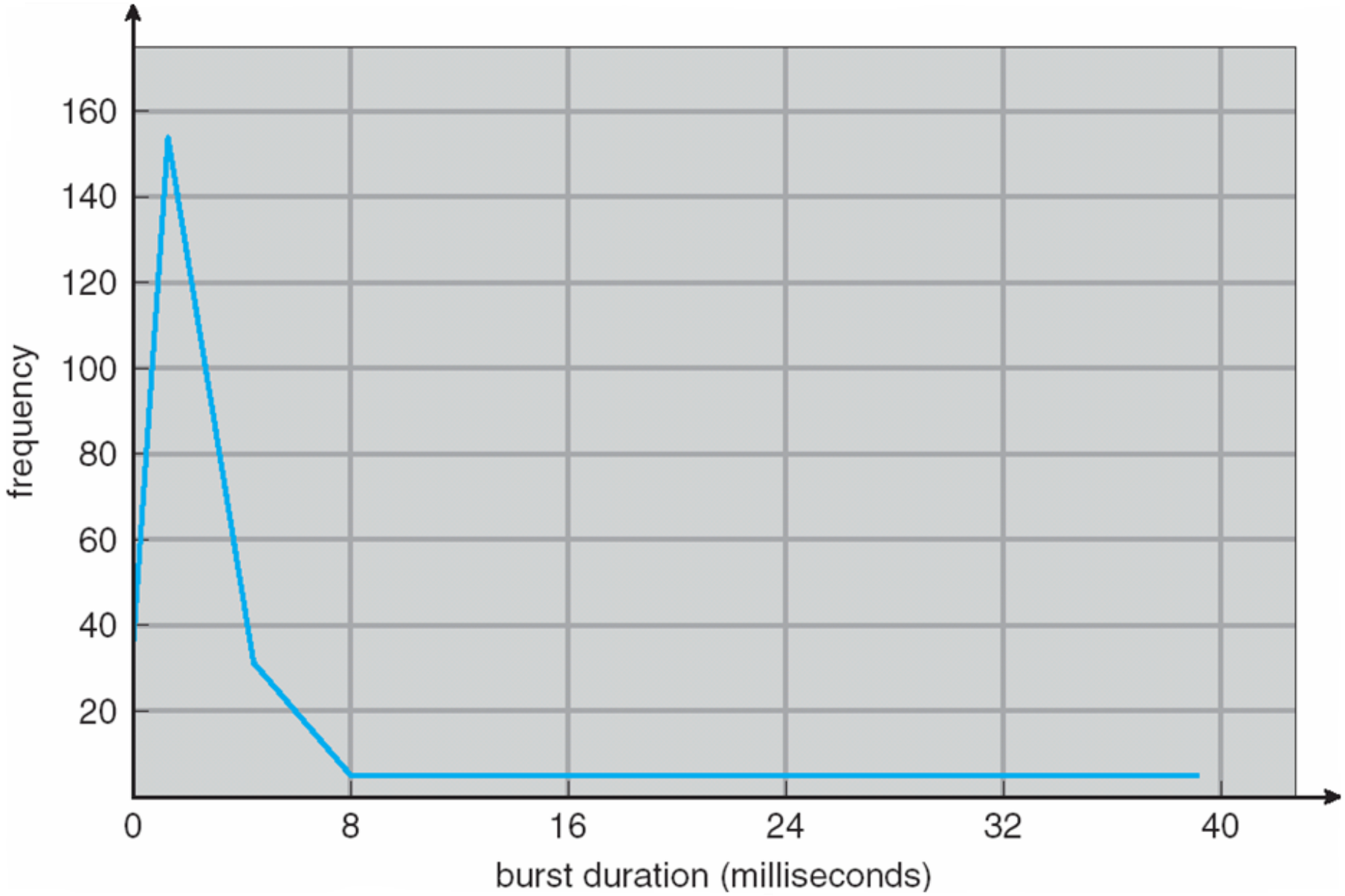
- Basic concepts
- Scheduling algorithms
- Special scheduling issues
- Case study

Basic Concepts

- Multiprogramming
  - keep several processes in memory
  - process waits => run another process on the CPU
- CPU-I/O burst cycle
  - process execution = interleaved CPU bursts and I/O bursts
  - I/O-bound program: more short CPU bursts
  - CPU-bound program: a few long CPU bursts



Histogram of CPU burst times



Preemptive vs Nonpreemptive

- CPU scheduling decision may take place when
  1. a process switches from RUNNING to WAITING state
  2. a process switches from RUNNING to READY state
  3. a process switches from WAITING to READY
  4. a process terminates
- Nonpreemptive (also called "cooperative")
  - a process keeps CPU until either #1 (waits) or #4 (terminates) or until it yields ("#5" not listed in the book)
- Preemptive
  - Scheduling under all 4 cases above

issues with Preemptive Scheduling

- Race condition:
  - inconsistent state of shared data as a result of context switching before data modification is complete
    - one process could overwrite another process's changes
- Solution: synchronization (Chapter 6)
  - incurs a cost associated with access to shared data

Preemptive vs Nonpreemptive Kernel Design

- Nonpreemptive kernel
  - waits for systems call to complete or process to block before context switch
  - simple, poor performance, not suitable for real-time
- Preemptive kernel (most modern OSs)
  - preemption could happen when making system call or when kernel is making critical changes
  - Needs extensive mutex locks

Interrupt issues

- temporarily disabling interrupt
  - needed to prevent race condition
  - may be needed to implement **atomic operations** (all or none)
- interrupts shouldn't be disabled for long
  - otherwise input may be lost,
  - otherwise output may be overwritten
  - interrupts should be disabled only for a few instructions
- In general, ISR shouldn't be long
  - higher-priority interrupts could mask lower-priority ones!

Dispatcher vs. scheduler

- Dispatcher
  - saving and restoring switching
  - jumping to the scheduled process
- Scheduler
  - Determines when to dispatch which process
- Latency breakdown: time it takes for dispatcher to stop one process and start another
  - scheduling time
  - interrupt-re-enabling time
  - context switching time

Scheduling Algorithms

Scheduling Criteria

- CPU Utilization
  - % time CPU is not idle. (0-100%) practical: 40% (light), 90% (heavy)
- Throughput
  - number of processes completed per time unit
- Turnaround time
  - time from submission to completion
- Waiting time
  - total time waiting in the Ready queue
- Response time
  - time from submission to the first response produced

Objectives of Scheduling Algorithm

- Maximize
  - CPU utilization
  - throughput
- Minimize
  - turnaround time
  - waiting time
  - response time

Scheduling Strategies

- First-come first-served (FCFS)
- Shortest job first (SJF)
- Priority-based scheduling
  - static, dynamic, ...
- Round-robin
- Multilevel queue scheduling
- Multilevel feedback queue scheduling

FCFS Scheduling

- Schedule a process in order of arrival
  - Effectively *nonpreemptive*

	exec time	ex1 arriva	ex2 arrival
P1	24	0	2
P2	3	1	1
P3	3	2	0

waiting time
= Ready but
not Running!

Example 1: Gantt chart

P0	P1 (24)				
P1	1	1	22	P2(3)	
P2	2		22	3	P3(3)

average waiting time: 48/3 = 16

Example 2: Gantt chart

P0			1	3	P1(24)
P1	1	2		P2(3)	
P2	P3(3)				

average waiting time: 6/3 = 2

FCFS evaluation

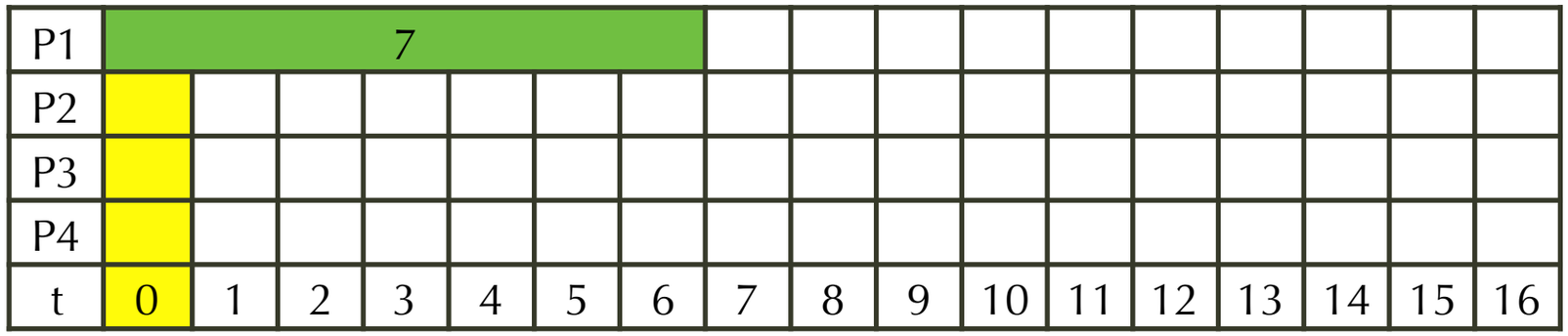
- (+) Simple
  - simple FIFO
- (-) CPU-bound and I/O-bound processes don't mix well
  - "Convoy effect" - shorter processes (I/O-bound) wait for one big process (CPU-bound) to finish => low CPU utilization
- (-) effectively nonpreemptive
  - bad for interactive systems

Shortest Job First (SJF) scheduling

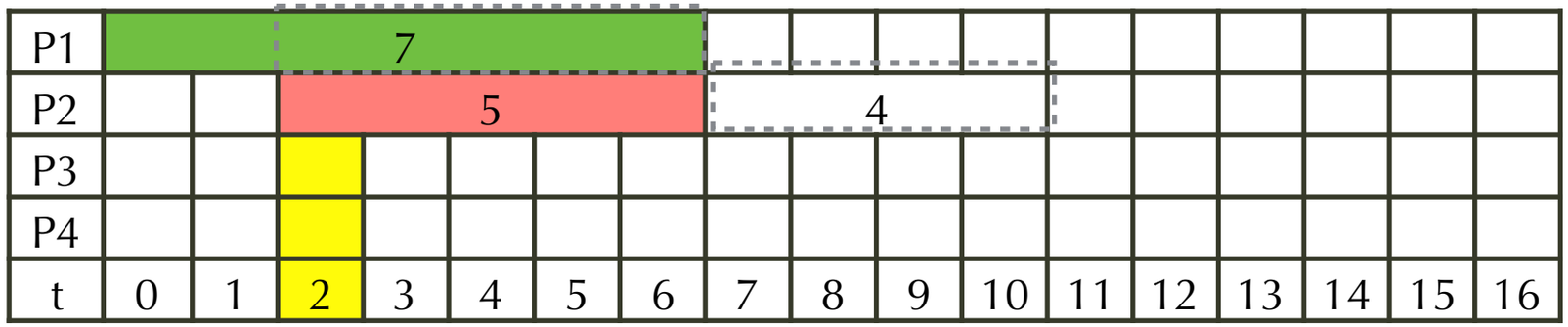
- Choose process with shortest **next CPU burst**
  - NOT process with the shortest total length!!
  - SJF is optimal for **minimum waiting time**
- Two schemes
  - preemptive: = **shortest remaining-time first**, when new job arrives, if remaining burst shorter than current process, preempt and run it
  - nonpreemptive: run a process till completion, even if a new job arrives with shorter remaining burst.
- Q: How do we know length of next CPU burst?
  - A: Can't know for sure (undecidable), but can use history to predict

Nonpreemptive SJF (1/3)

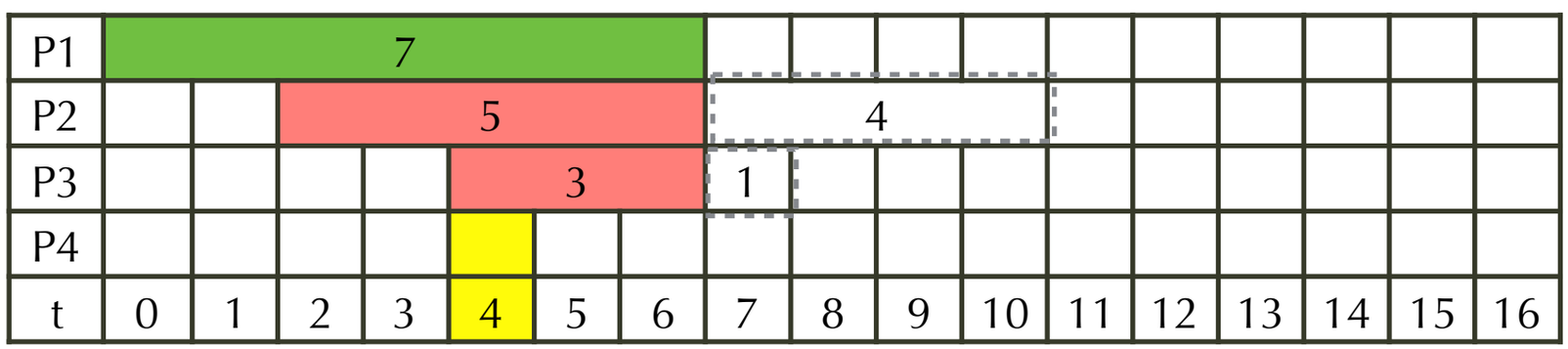
	burst time	arrv
P1	7	0
P2	4	2
P3	1	4
P4	4	5



t=0, P1 released. scheduled immediately



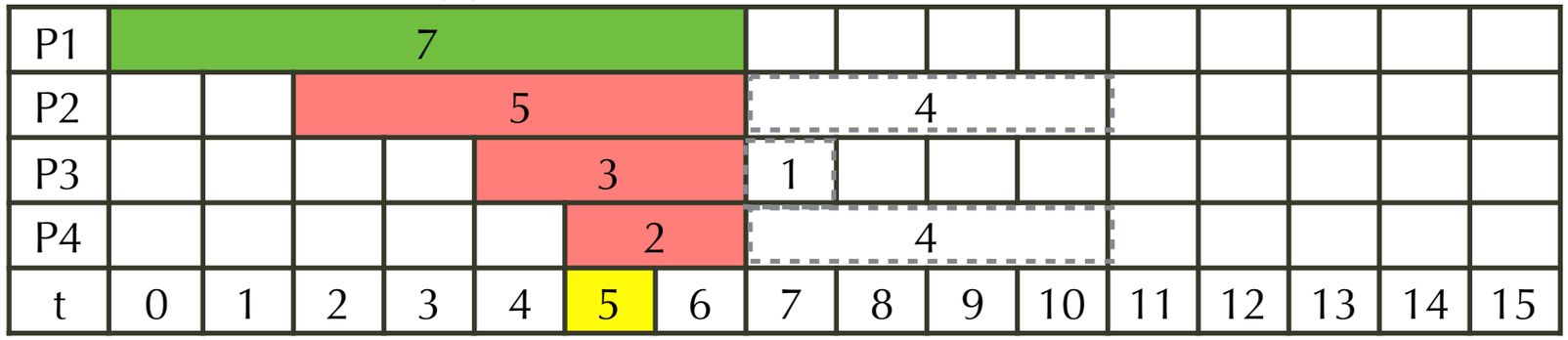
t=2, P2 released. Nonpreemptive => deferred till completion of P1



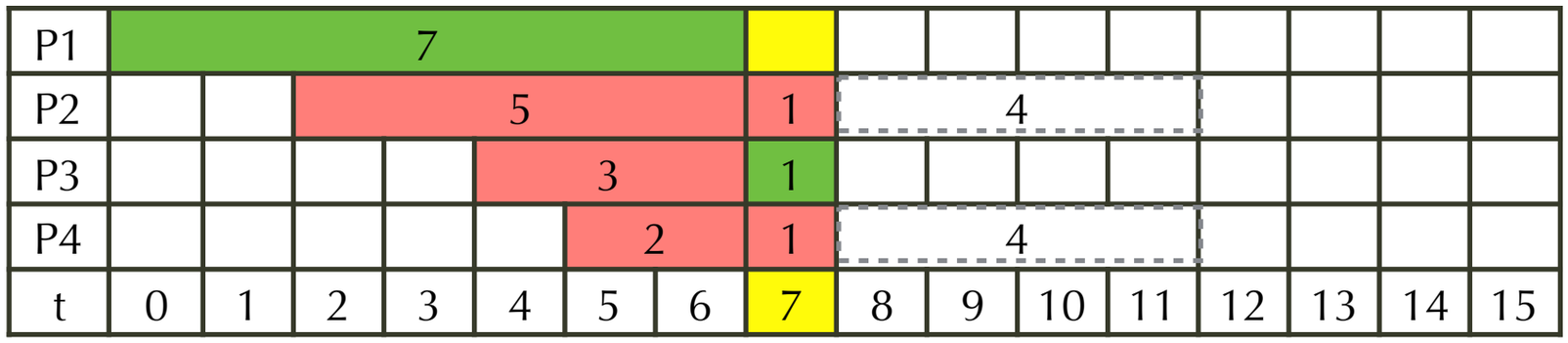
t=4, P3 released. deferred till completion of P1

Nonpreemptive SJF (2/3)

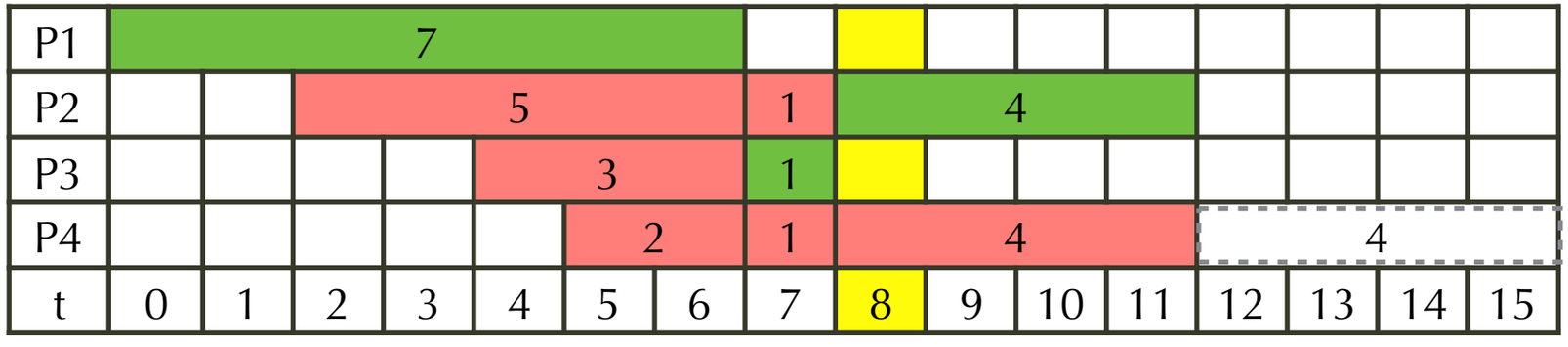
	burst time	arrv
P1	7	0
P2	4	2
P3	1	4
P4	4	5



t=5, P4 arrives but is blocked till P1 completes



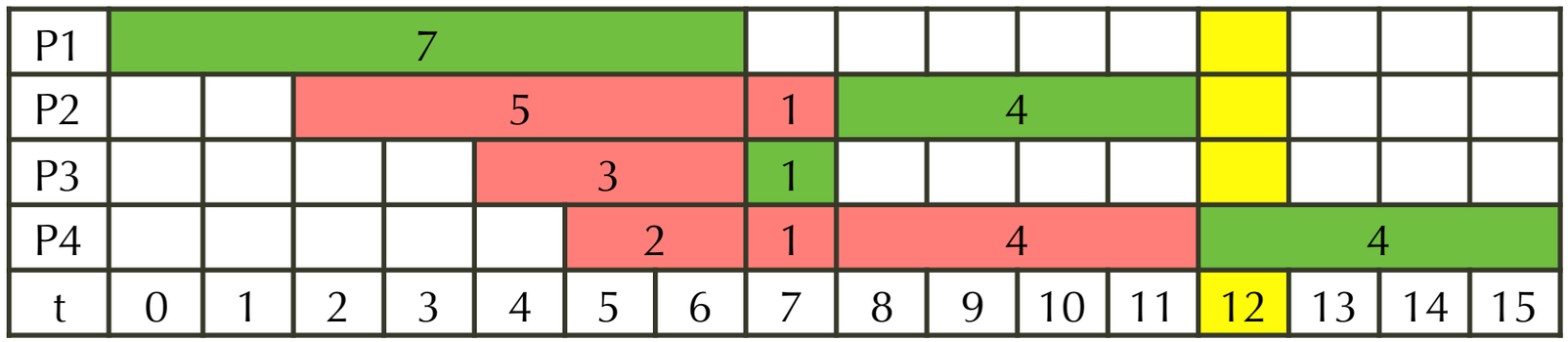
t=7, P1 completes. P3 is shortest and is scheduled.



t=8, P3 completes. P2 is shortest and is scheduled.

Nonpreemptive SJF (3/3)

	burst time	arrv
P1	7	0
P2	4	2
P3	1	4
P4	4	5



t=12, P2 completes. P4 is shortest (the only) and is scheduled.

at t=17, P4 completes.

"Response time": P1=0, P2=6, P3=3, P4=7
(waiting time before it can be run for the first time)

Total waiting time = 0 + (P1) + 6 (P2) + 3 (P3) + 7 (P4) = 16
Average waiting time = 16/4 = 4.

Preemptive SJF (1/3)

	burst time	arrv
P1	7	0
P2	4	2
P3	1	4
P4	4	5

P1	7															
P2																
P3																
P4																
t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

t=0, P1 released. scheduled immediately

P1	2/7		5													
P2			4													
P3																
P4																
t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

t=2, P2 released (4), shorter than remaining time of P1 (5), preempts P1

P1	2/7		2		5											
P2			2/4		2											
P3					1											
P4																
t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

t=4, P3 released (1). shorter than remaining P1 and remaining P2, preempts P2.

Preemptive SJF (2/3)

	burst time	arrv
P1	7	0
P2	4	2
P3	1	4
P4	4	5

P1	2/7		2		1	5										
P2			2/4		1	2/4										
P3					1/1											
P4						4										
t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

t=5, P4 arrives but is blocked because P2 (2 remaining) is shortest. P2 resumes.

P1	2/7		2		1	2	5									
P2			2/4		1	2/4										
P3					1/1											
P4						2	4/4									
t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

t=7, P2 completes. P4 (4) is shorter than P1's remaining 5 and is dispatched next

P1	2/7		2		1	2	4				5/7					
P2			2/4		1	2/4										
P3					1/1											
P4						2	4/4									
t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

t=11, P4 completes. P1 is the only remaining and is scheduled.

Preemptive SJF (3/3)

- Response time:
  - P1=0, P2 = 0, P3 = 0, P4 = 2
- Waiting time = completion time - arrival time - runtime (burst time)
  = (16-0-7)+(7-2-4)+(5-4-1)+(11-5-4)
  = 9+1+0+2=12
- Average wait time = 12/4 = 3
  - Compared 4 in nonpreemptive case!

Approximate Shortest-Job-First

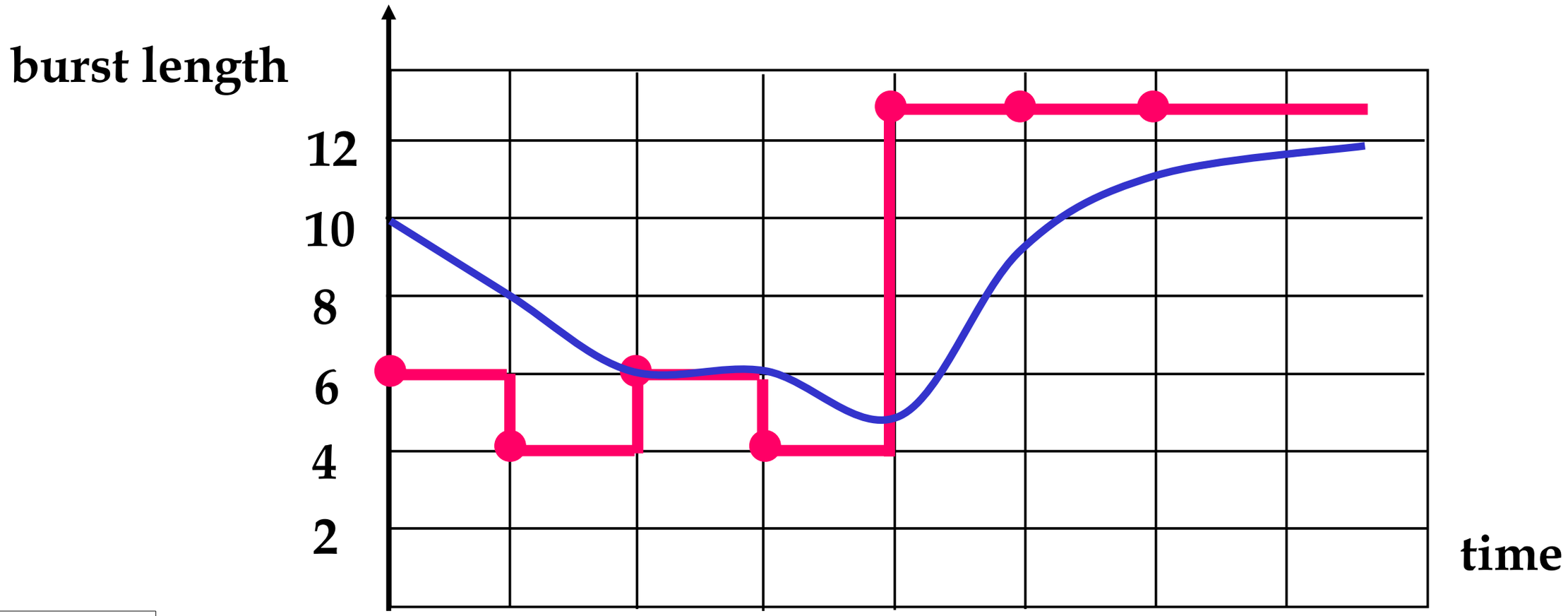
- Difficulty with SJF:
  - no knowledge of the length of the next CPU burst
- Solution: Approximate by prediction
  - Predict length of next burst as exponential average of previous CPU bursts

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \qquad \textit{// new one + history}$$

$$= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 t_{n-2} + \cdots$$

$$= \left(\frac{1}{2}\right) t_n + \left(\frac{1}{2}\right)^2 t_{n-1} + \left(\frac{1}{2}\right)^3 t_{n-2} + \cdots \qquad \textit{// commonly } \alpha = 1/2$$

Exponential Prediction of next CPU burst



	t0	*t1*	*t2*	*t3*	*t4*	*t5*	*t6*
CPU burst t_n	6	4	6	4	13	13	13
guess τ_n	10	8	6	6	5	9	11

Round-Robin (RR) Scheduling

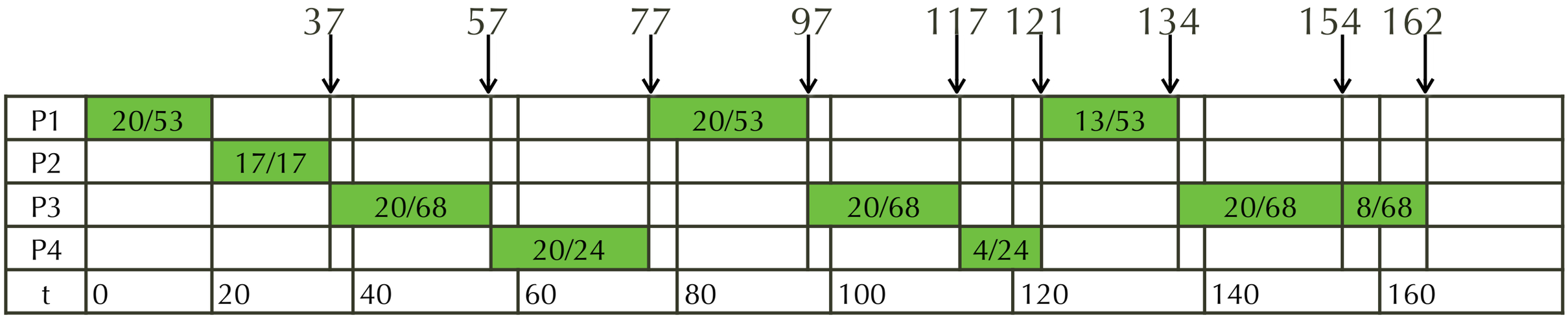
- Round-Robin = take turns on a regular basis
  - assumes preemption!!
  - if without preemption => essentially FCFS
- Concept of time quantum (TQ), aka time slice
  - Unit of CPU time, usually every 10-100 ms
  - after TQ elapsed, preempt process, put back to ready queue so next process gets to run
  - No starvation

Performance of Round-Robin

- if large time quantum
  - similar to FCFS
- if small time quantum
  - context-switching overhead dominates
- In practice:
  - 80% CPU bursts should be shorter than TQ

RR scheduling w/ TQ=20

	burst time
P1	53
P2	17
P3	68
P4	24



- Typically RR has higher turnaround than SJF, but better response

Priority-based Scheduling

- Pick process w/ highest priority to run next
  - could be preemptive or nonpreemptive
- Many possible ways to define priority
  - SJF is a priority scheme (i.e., inverse of length of next CPU burst)
  - FCFS: equal priority, no preemption
  - in real-time system, deadline could be priority
  - age of process
  - hybrid schemes

Issue with some priority schemes: Starvation

- Starvation = "indefinite blocking"
  - i.e., process is ready but not scheduled to run for long time
  - some processes get stuck with low priority and never get to execute!
    - example: IBM 7094 shutdown in 1973, when a process from 1967 never got to run
- Possible solution: aging
  - as time progresses, increase priority of processes
  - e.g., increase priority by 1 every 15 minutes (depends on number of priority level)

Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues
  - e.g., foreground (interactive) vs. background (batch)
  - Processes **stay in given queue**, does not move between queues
  - Each queue has its own scheduling algorithm (priority scheme) e.g., RR for foreground, FCFC for background
- Example: highest to lowest priority queues
  - Real-time, System, Interactive, Batch processes
- Scheduling is needed between queues
  - fixed priority => possible starvation
  - each queue could get a time slice

Multilevel Feedback Queue Scheduling

- A process can **move between queues**
  - aging can be implemented this way
- Idea: separate processes based on characteristic of CPU burst
  - I/O-bound and interactive processes in higher priority queue => short CPU burst
  - CPU-bound processes in lower priority queue => long CPU burst

Multilevel Feedback Queue Example

- Q_0, FCFS, TQ_0= 8 ms
  - Job enters Q_0, does not finish => move to Q_1.
- Q_1 also FCFS, TQ_1=16 ms
  - job does not finish => move to Q_2
- Q_i executes only if $Q_0 \ldots Q_{i-1}$ empty

Multilevel Feedback Queue

- Defined by
  - number of queues
  - scheduling policy for each queue
  - condition for “upgrading” a process
  - condition for “demoting” a process
  - condition for choosing which queue for a process to enter

Thread Scheduling: Scope of Contention

- Process-contention scope (PCS)
  - user-level scheduling competition is **within process**
  - for many-to-one and many-to-many models
  - Typically done via priority set by programmer
- System-contention scope (SCS)
  - Kernel thread scheduled onto available CPU
  - Competition among all kernel threads **in the system**

Pthread Scheduling

- API specifying PCS or SCS on thread creation
  - PTHREAD_SCOPE_PROCESS: PCS scheduling
  - PTHREAD_SCOPE_SYSTEM: SCS scheduling
- Can be limited by OS
  - Linux & macOS allow only SCS

Multi-Processor Scheduling, Multi-Core Processor Scheduling, and Real-time Scheduling

Multiprocessor Scheduling

- Asymmetric multiprocessing
  - all system activities are handled centrally by one processor ("master server")
  - other processors only execute user code
  - scheduler is simpler (no data structure sharing), but master server could become bottleneck
- Symmetric multiprocessing (SMP)
  - each processor schedules itself
  - Option1: each processor has own private queue
  - Option 2: all processors share ready queue [Most common]
    => need synchronization mechanism

Two trends in Multicore Processors

- Multiple cores on same physical chip
  - Faster and consumes less power by running at lower clock rate and therefore lower voltage
  - make up performance by parallelism
- Multiple threads per core
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
  - examples: Intel Hyperthreading, IBM (1968), Tera MTA (1988), CDC6600

Simultaneous Multithreading (SMT)

- aka hardware threads, chip multithreading
  - CPU maintains multiple PC, register set, etc ("logical processor")
- Coarse-grained
  - switch thread when memory stall occurs
  - high performance cost when flushing instruction pipeline
- Fine-grained
  - interleaving threads at instruction boundary
  - lower performance cost, since hardware supports thread interleaving

Multithreaded Scheduling on Multicore

- Thread scheduler needs to consider
  - cache behavior and resource sharing,
  - or else threads may run slower!
- Processor allocation
  - select which software thread to run on each hardware thread (processor)
- Intra-core threading
  - each core decides which hardware thread to run

Load Balancing

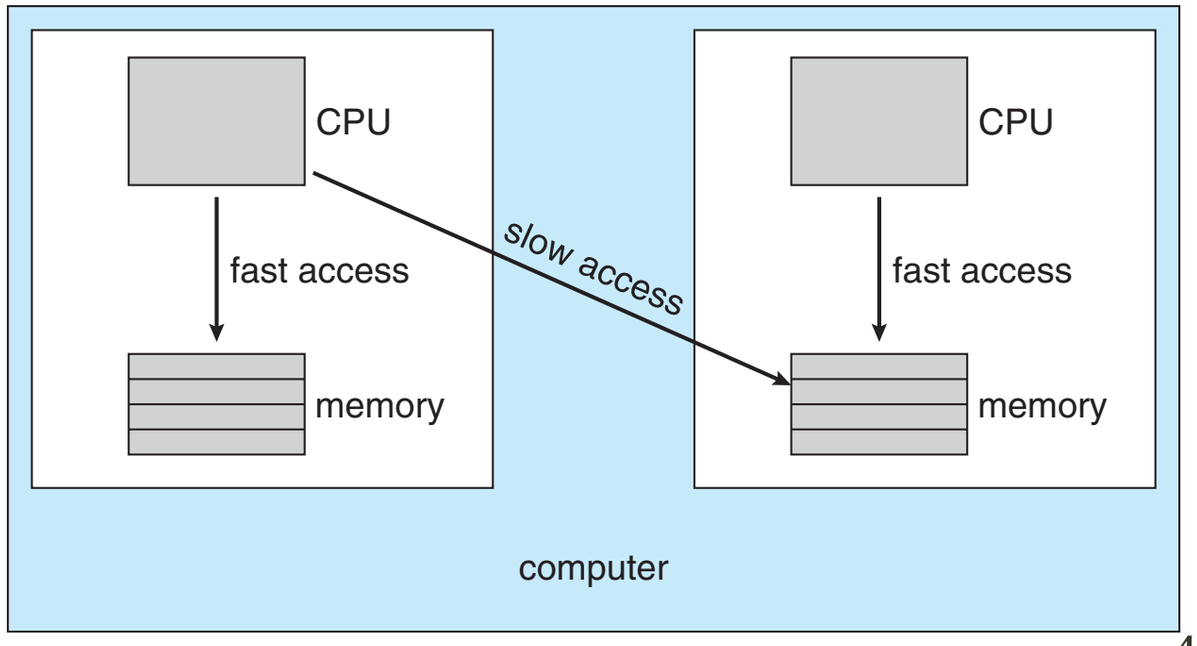
- Keep workload evenly distributed
  - For SMP (private queue), keep all CPUs loaded for efficiency
  - shared queue => automatically load balanced!
- Meaning of "load balanced"
  - private queues have same **number of threads**, or
  - equal distribution of **thread priorities** across all queues.
- Push migration vs Pull migration
  - Push: overloaded CPU pushes load to less-busy (or idle) CPUs
  - Pull: idle CPU pulls waiting task from busy CPU

Processor Affinity

- "closeness" between a process and the processor
  - data in cache memory of the processor
  - high cost to invalidate and repopulate cache when moving to another processor
  - nonuniform memory architecture (NUMA)
- Two kinds of affinity policy OS can set for migration
  - soft affinity: OS tries to keep process on same processor but allows migration between processors
  - hard affinity: restricts process to migrate to subset of processors
- Load balancing counteracts processor affinity

CPU Scheduling with NUMA

- NUMA = Nonuniform Memory Access
  - for systems built with combined CPU+memory boards
  - local access fast; cross-board access slow
- CPU scheduling + memory placement
  - high processor affinity for process whose memory is allocated to the CPU



Heterogeneous Multiprocessing (HMP)

- Motivation: mobile devices (smartphone)
  - Mixing high-performance and low-power multiprocessors
  - NOT the same as asymmetric multiprocessing!
- example: ARM big.LITTLE architecture
  - big = high performance, saves time
  - LITTLE = energy efficient, saves power
  - same instruction set, direct migration between them
  - together => saves energy

Real-Time Scheduling

Real-Time CPU Scheduling

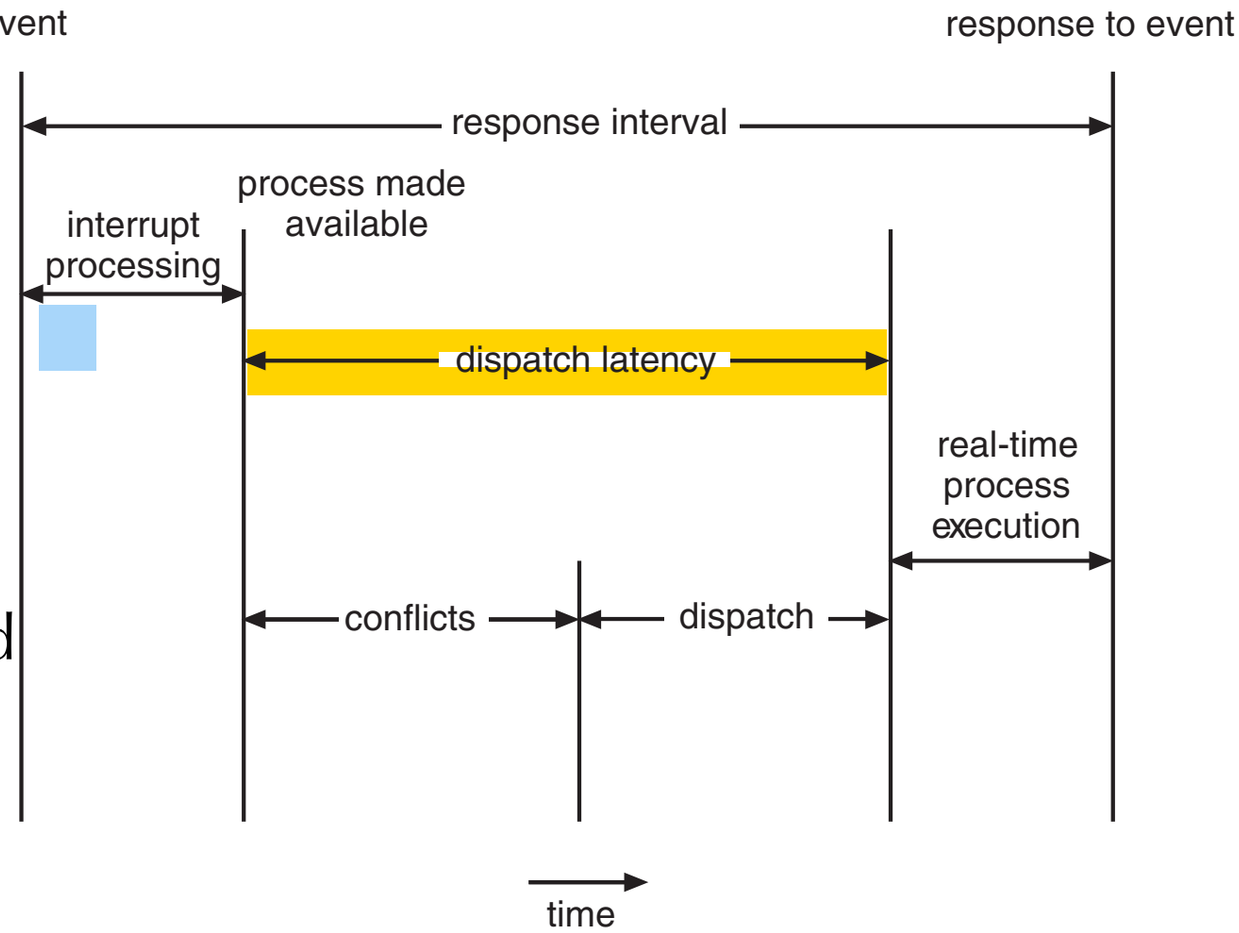
- Real-Time vs. Non-Real-Time
  - Non-Real-Time: time does not affect correctness
  - Real-Time: timing is a key part of correctness
- Real-Time vs. Speed
  - Real-time does not (always) mean "fast", though it could help
  - Real-Time means meeting *timing constraints*

Hard vs Soft Real-Time

- Soft real-time systems
  - provides *preferences* but *not guarantee* in scheduling real-time processes
  - missing deadline is undesirable but not critical
  - example: Multimedia streaming
- Hard real-time systems
  - provides guarantees in meeting deadline
    => once the task is accepted
  - missing deadline leads to fundamental failure
  - example: nuclear power plant controller

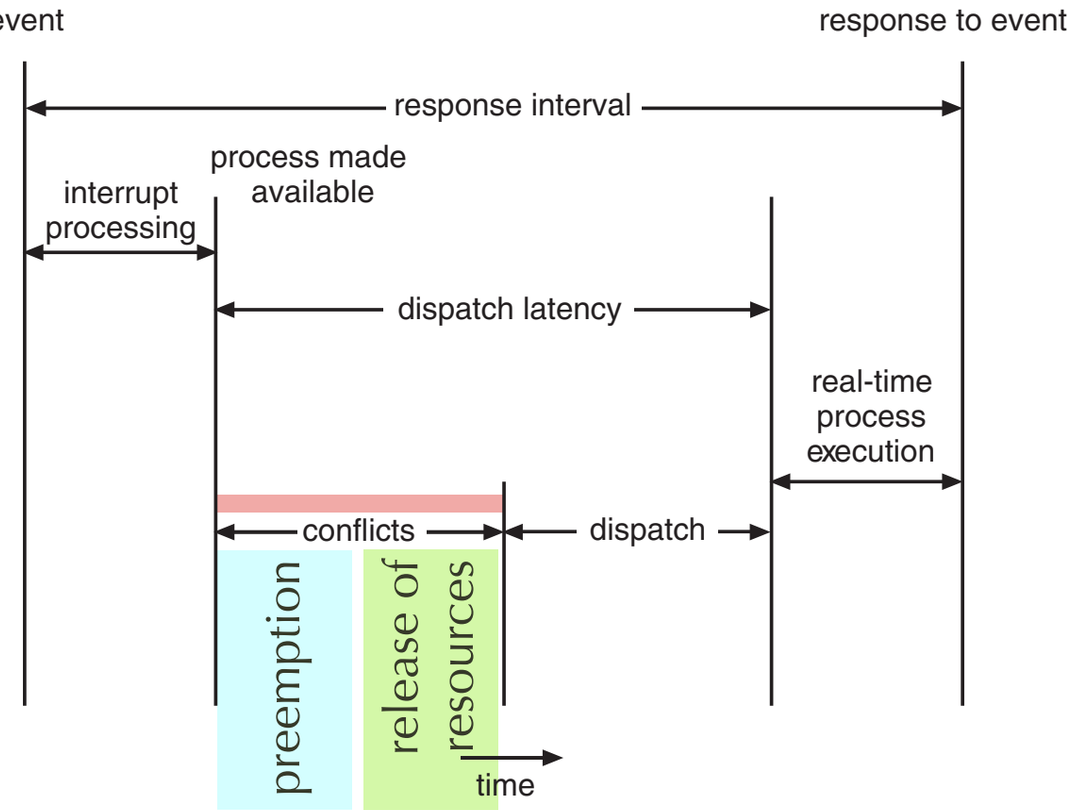
Latency

- Event latency
  - amount of time from event occurring to time the event is serviced
- Interrupt latency
  - amount of time from arrival of interrupt to start of ISR execution
  - must be bounded, not just minimized
- Dispatch latency
  - amount of time for the dispatcher to switch process
  - best minimized thru preemptive kernel



Conflict phase of Dispatch Latency

- Preemption
  - of any process running in kernel mode
- Release of resources
  - by low-priority process so high-priority processes can use them

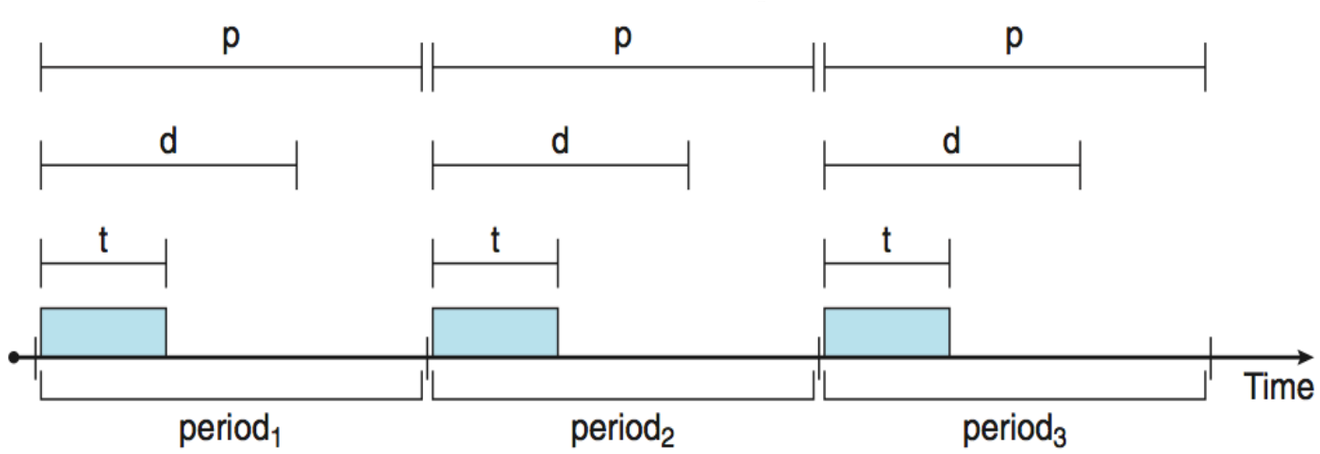


Real-Time Scheduling Algorithms

- Offline (pre-run-time)
  - workload is known or bounded before running
  - can perform schedulability analysis before running
  - could be done by separate algorithm; OS just dispatches
- Online (runtime, dynamic)
  - actual scheduling done as the system runs
  - need **admission control** -- accept or reject tasks based on ability to meeting timing constraints

Periodic vs Aperiodic Tasks

- Periodic
  - tasks is recurring
  - deadline d, could be same as period p
  - may have release time after start of period
- Aperiodic



  - task is nonrecurring, or may repeat but not before some minimum separation

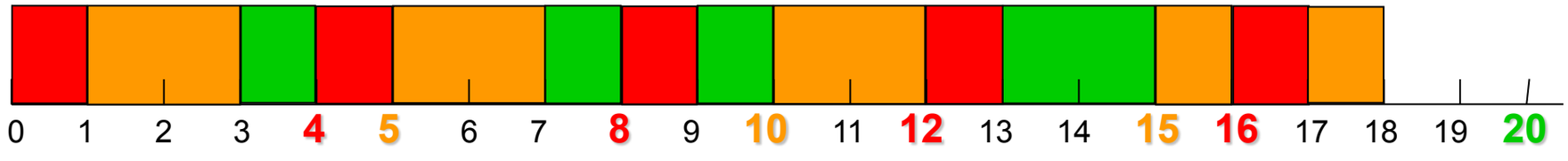
Priority-based Scheduling

- Rate-Monotonic (RM) scheduling
  - assumes periodic tasks, deadline = end of period
  - preemptive, **fixed priority** based on length of period
- Earliest-Deadline-First (EDF) scheduling
  - may be periodic or aperiodic
  - preemptive, **dynamic priority** based on time to deadline

Rate Monotonic (RM) Scheduling

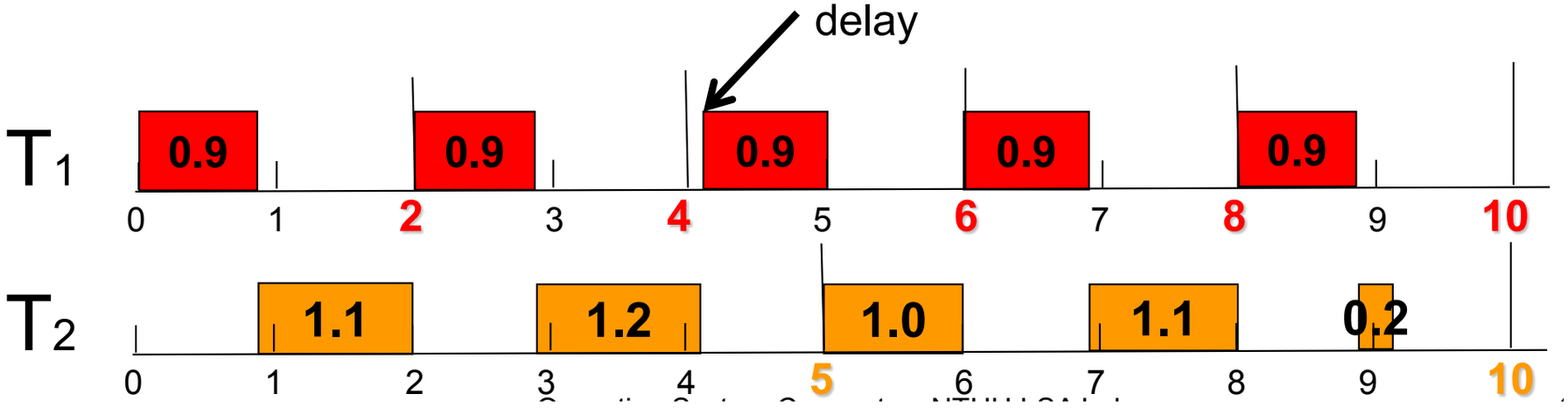
C.L. Liu (劉炯朗)
former President of NTHU, 1998-2002

- by Liu and Layland (1973)
- Shorter period => higher priority
  - assumption: period is fixed; preemptive
  - priority depends only on period, not CPU burst!
    (in contrast to shortest-job first)
- Example: tasks (period, burst)
  - (4, 1), (5, 2), (20, 5)



Earliest Deadline First (EDF) Scheduling

- Dynamic priority
  - the earlier the deadline, the higher the priority
  - could be periodic or aperiodic
    - Preemptive: highest-priority task gets to preempt others upon release (could be periodic)
- Example: (period, burst): (2, .9), (5, 2.3)



Schedulability Analysis

- RM
  - sufficient condition: Utilization ≤ about 69% = ln(2) (i.e., natural log of 2); actual bound may be higher
  - "optimal" for fixed-priority algorithms
- EDF
  - Utilization ≤ 100%
  - "optimal" for dynamic-priority algorithms
- "Optimal" for real-time scheduling
  - "if any other algorithm can schedule it, then it can schedule it"
  - different meaning of optimal for non-real-time objectives

Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  - Scheduler does not know it doesn't own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

Proportional Share Scheduling

- T shares are allocated among all processes in the system

- An application receives *N* shares where *N* < *T*

- This ensures each application will receive *N* / *T* of the total processor time

POSIX Real-Time Scheduling

- The POSIX.1b API for managing real-time threads
- Defines two scheduling classes for real-time threads:
  - `SCHED_FIFO` - threads scheduled by FCFS with a FIFO queue. No time-slicing for threads of equal priority
  - `SCHED_RR` - similar to `SCHED_FIFO` except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  - `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  - `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling 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);
}
```

Operating System Examples

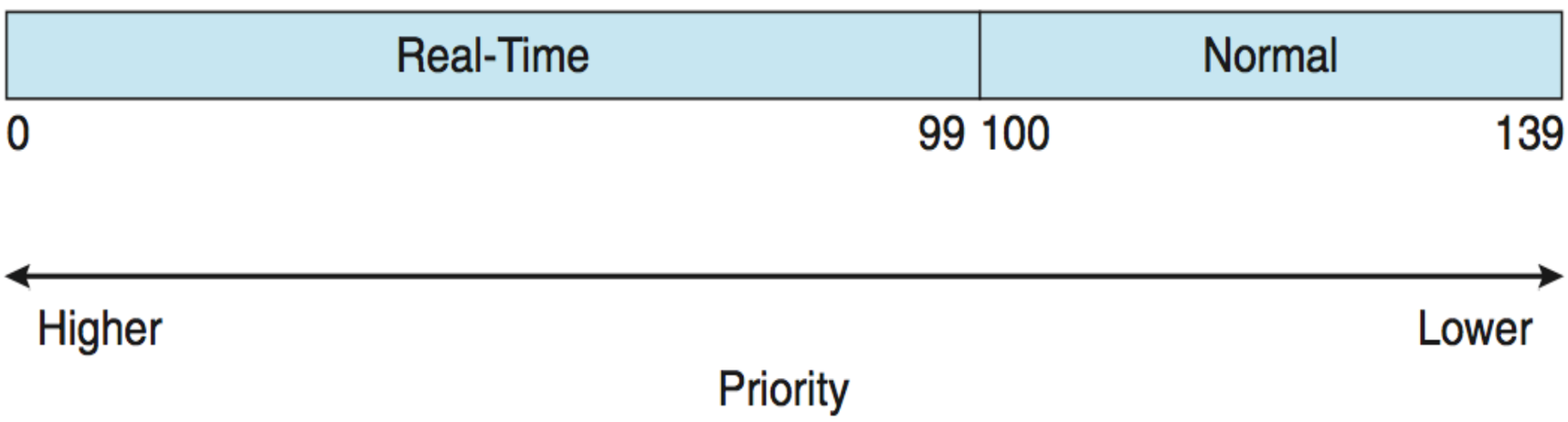
- Linux
- Windows
- Solaris

Linux Scheduling in Version 2.6.23 +

- Completely Fair Scheduler (CFS)
- Scheduling classes
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    - default
    - real-time

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Windows Scheduling

- Windows uses priority-based preemptive scheduling
  - Highest-priority thread runs next
  - Dispatcher is scheduler
  - Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
  - 32-level priority scheme
  - Variable class is 1-15, real-time class is 16-31
  - Priority 0 is memory-management thread
  - Queue for each priority
- If no runnable thread, runs idle thread

Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS), Interactive (IA), Real time (RT), System (SYS), Fair Share (FSS), Fixed priority (FP)
- Given thread can be in one class at a time
  - Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
- Runs until
  - (1) blocks,
  - (2) uses time slice,
  - (3) preempted by higher-priority thread
- Multiple threads at same priority selected via RR

Evaluation of Scheduling Policies (and Mechanisms)

- Deterministic modeling
  - evaluate performance for given workload
- Queuing model
  - by mathematical analysis
- Simulation
  - simulate either trace from real workload or synthetic workload (generated using random number)
- Implementation
  - implement the scheduler and run it for real