CPU Scheduling

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

UNIVERSITY OF CALGARY

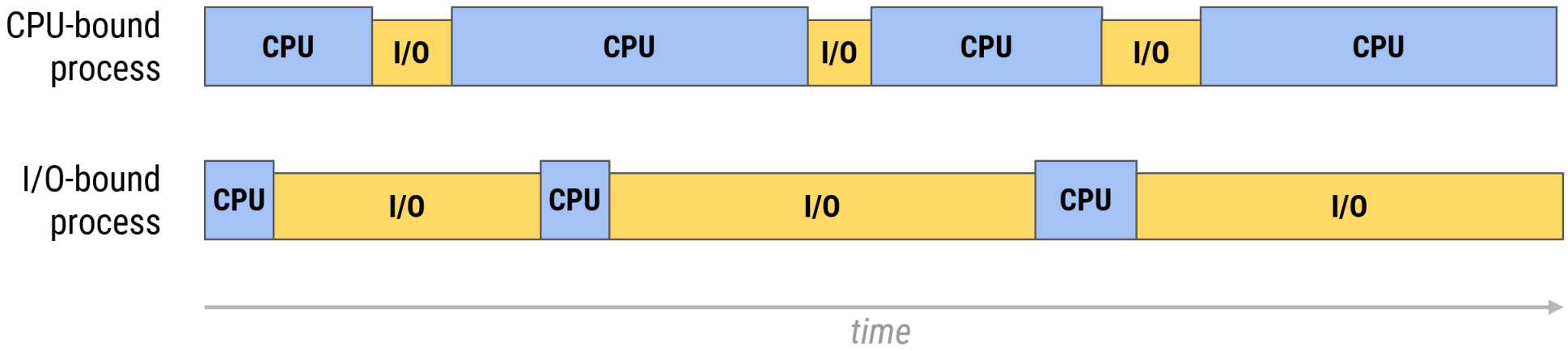
Topics

- objectives of scheduling
- cpu-bound vs io-bound processes
- when to schedule
- preemptive vs non-preemptive scheduling
- categories of CPU scheduling algorithms
  - batch, interactive, real-time
- metrics

Definition

Process behaviour

- most processes alternate bursts of CPU activity with bursts of I/O activity
- **CPU-bound** (or compute-bound) processes — have long CPU bursts and infrequent I/O waits
- **I/O-bound** processes — have short CPU burst and frequent I/O waits

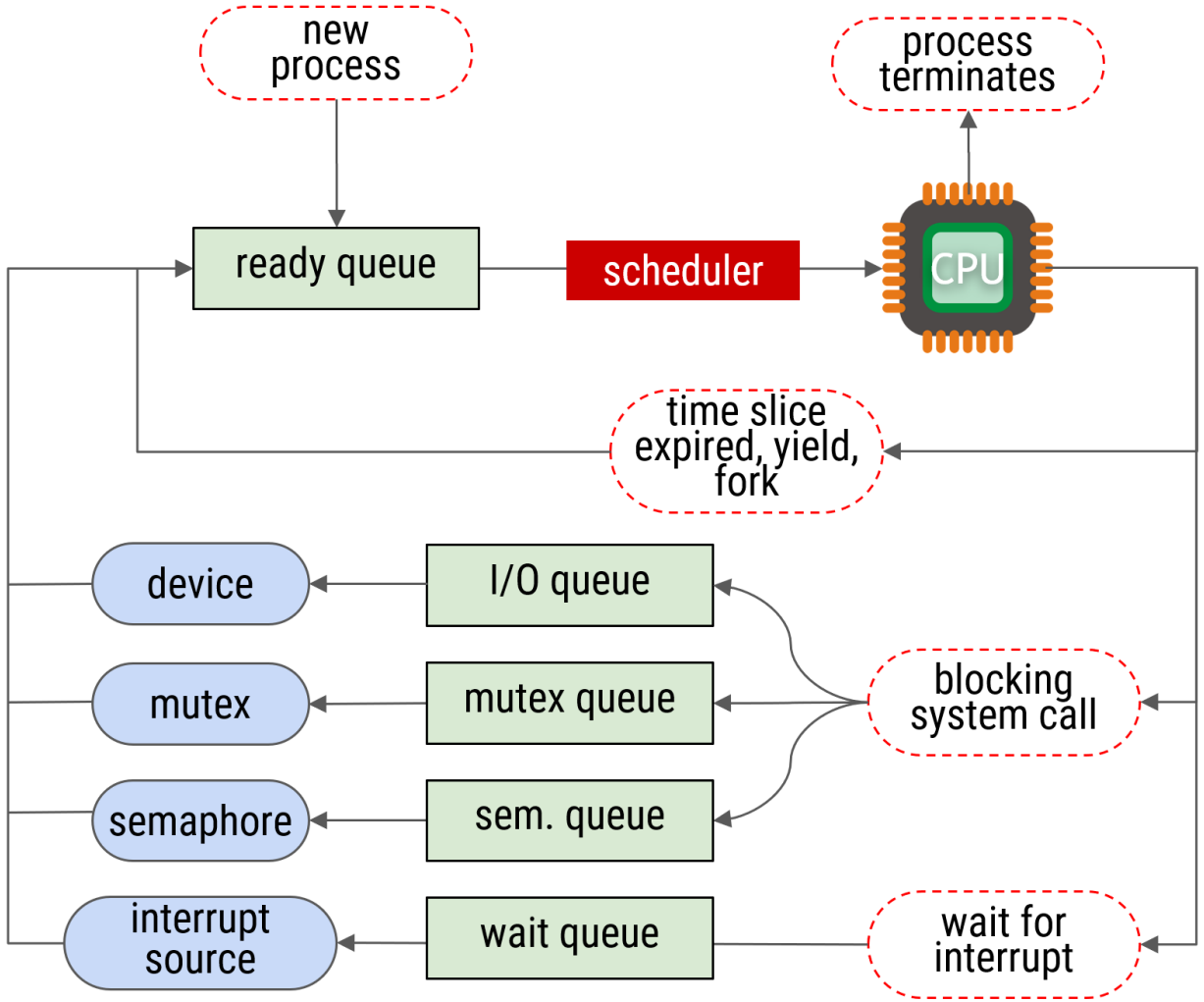


- a single CPU-bound process could keep CPU busy
- it would take several IO-bound processes to keep CPU occupied

CPU scheduling

- the piece of code that decides which process runs next is called a scheduler
  - usually part of a kernel
  - a scheduler implements some scheduling algorithm

When OS needs to invoke scheduler



- variety of reasons
- basically any time CPU becomes available
- examples:
  - process yields
  - thread calls mutex lock
  - time slice expires

Preemptive vs non-preemptive CPU scheduling

Preemptive vs non-preemptive CPU scheduling

- **non-preemptive** — context switch happens only voluntarily
  - multitasking is possible, but only through cooperation
  - process runs until it does a blocking syscall (e.g. I/O), terminates, or voluntarily yields CPU
  - example: FCFS
- **preemptive** — context switch can happen without thread's cooperation
  - usually as a direct or indirect result of some event, but not limited to clock interrupt

    e.g. new job is added, existing process is unblocked
  - example: SRTNtime-sharing

Preemptive vs non-preemptive CPU scheduling

- **preemptive time-sharing** — special case of preemptive
  - processes are context switched periodically to enforce time-slice policy
  - implemented through clock interrupts
  - without a clock, only cooperative multitasking (non-preemptive) is possible
  - example: RR
  - so common that 'preemptive' is often (mis)used to mean preemptive time-sharing

Categories/Metrics/Goals

Categories of scheduling algorithms

- batch systems
  - no impatient users
  - usually used on mainframes — e.g. processing payrolls, weather simulations
  - usually no interactivity is needed → no time-slice needed
  - preemption is rarely needed, except maybe when adding new jobs
- interactive systems
  - impatient users
  - running many tasks, usually with GUIs, many tasks must remain interactive
  - preemption (time-sharing) is essential to keep users happy
- real time systems
  - applications must be given guaranteed amount of CPU cycles
  - often tied closely to some hardware
    - robots, planes, cars, video/audio capture, games, streaming

Scheduling metrics

Statistics about each process/job:

- arrival time – the time a process arrives (e.g. when you double-click Firefox icon)
- start time – the time process first gets to run on CPU
  - different from arrival for batch systems, nearly identical to arrival on interactive systems
- finish time – when the process is done (time of the last instruction)
- response time – how long before you get first feedback, often response = start - arrival
- turnaround time – time from arrival to finish, turnaround = finish - arrival
- CPU time – how much time the process spent on CPU
- waiting time – total time spent it waiting queue, waiting = turnaround - CPU - I/O

Overall statistics:

- average turnaround time, average wait time ...
- throughput - number of jobs finished per unit of time

Scheduling algorithm goals examples

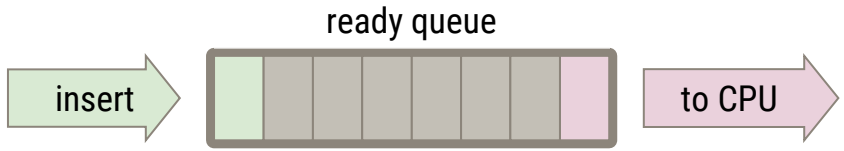
- All systems:
  - fairness: giving each process a fair share of the CPU
  - policy/priority enforcement: seeing that stated policy or priority is carried out
  - balance: keeping all parts of the system busy
- Batch systems:
  - throughput: maximize jobs per hour (or per minute)
  - turnaround time: minimize time between submission and termination
  - CPU utilization: keep the CPU busy all the time
  - waiting time: turnaround time - execution time

Scheduling algorithm goals examples

- Interactive systems:
  - response time: minimize time between submission and the first response
  - proportionality: meet users' expectations
- Real-time systems:
  - meeting deadlines: avoid losing data
  - predictability: avoid quality degradation in multimedia systems

First-come-first-served (FCFS) scheduling

First-come-first-served (FCFS) scheduling



- FCFS is the most basic scheduling algorithm

- it is non-preemptive
- common in batch environments

- CPU assigned in the order the processes request it, using a FIFO ready queue
- a running job keeps the CPU until it is either finished, or it blocks

- when running process blocks, next process from ready queue starts to execute
- when process is unblocked, it is appended at the end of the ready queue

- requires **minimum number of context switches** — only N switches for N processes

FCFS scheduling

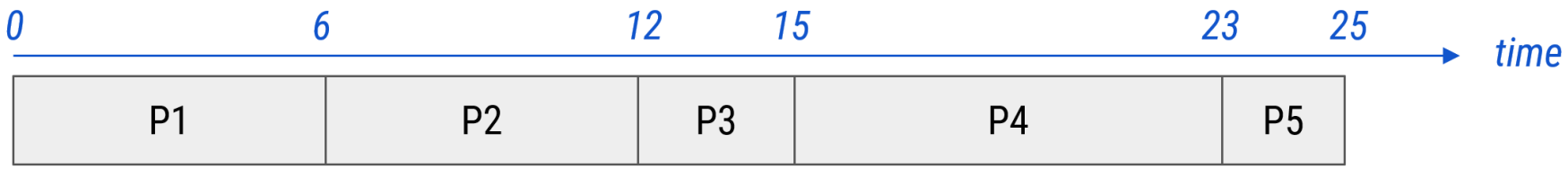
- let's construct a Gantt chart to visualize scheduling of 5 processes:

Process name	Arrival time	Burst time
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2

for simplicity we assume no I/O activity

the listed burst tells us how long the job will need to spend on CPU in order to finish

- Gantt chart:

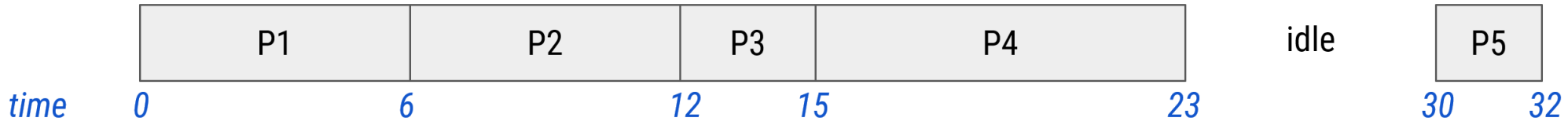


FCFS scheduling

- let's construct a Gantt chart to visualize scheduling of 5 processes:
- (for simplicity we assume no I/O activity)

Process name	Arrival time	Burst time
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	30	2

- Gantt chart:



FCFS - Simulating scheduling

Time	CPU	RQ	Comment/event
0	none	empty	simulation starts
0	none	1/6	P1 arrives
0	none	1/6,2/6	P2 arrives
0	1/6	2/6	P1 moves to CPU
1	1/5	2/6	P1 executes on CPU
1	1/5	2/6,3/3	P3 arrives
2	1/4	2/6,3/3	P1 executes on CPU
2	1/4	2/6,3/3,4/8	P4 arrives
3	1/3	2/6,3/3,4/8	P1 executes on CPU
3	1/3	2/6,3/3,4/8,5/2	P5 arrives
		...	
6	1/0	2/6,3/3,4/8,5/2	P1 executes last time
6	none	2/6,3/3,4/8,5/2	P1 is done
6	2/6	3/3,4/8,5/2	P2 moves to CPU
7	2/5	3/3,4/8,5/2	P2 executes
		...	
12	none	3/3,4/8,5/2	P2 is done
		...	
25	none	empty	P5 is done

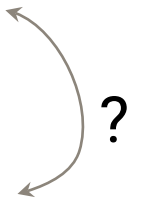
?

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2

Possible simulation loop structure

```
curr_time = 0
while(1) {
    if simulation done: break
    if thing_1 should happen at curr_time
        do thing_1
        continue // without incrementing time
    if thing_2 should happen at curr_time
        do thing_2
        continue
    ...
    if thing_n should happen at curr_time:
        do thing_n
        continue
    curr_time ++
}
```

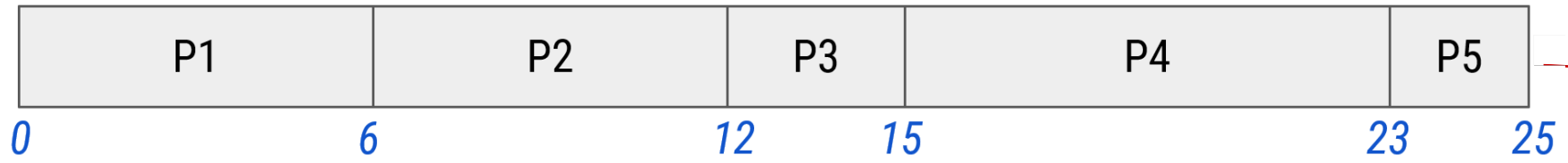
```
curr_time = 0
jobs_remaining = size of job queue
while(1) {
    if jobs_remaining == 0 break
    if process in cpu is done
        mark process done
        set CPU idle
        jobs_remaining --
        continue
    if a new process arriving
        add new process to RQ
        continue
    if cpu is idle and RQ not empty
        move process from RQ to CPU
        continue
    if CPU has job:
      execute one burst of job on CPU
    else:
      stay idle
    curr_time ++
}
```



FCFS scheduling

- let's calculate some statistics…

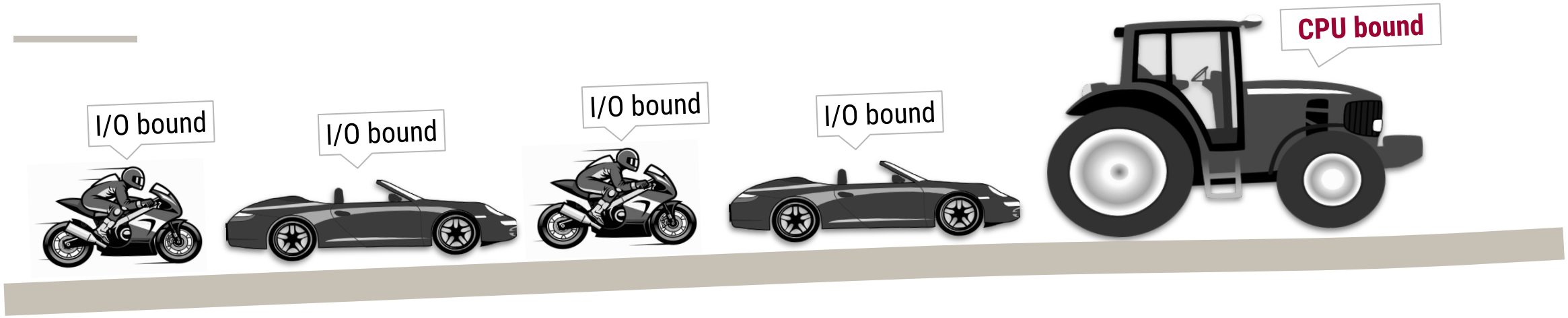
Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6	0	6	6	0
P2	0	6	6	12	12	6
P3	1	3	12	15	14	11
P4	2	8	15	23	21	13
P5	3	2	23	25	22	20



- **avg. wait time** = (0+6+11+13+20)/5 = 10 units
- **number of context switches**: 5

Convoy Effect

Convoy Effect



- big disadvantage of FCFS is the **convoy effect**
- convoy effect results in overall performance decrease of a system with mostly IO-bound processes when a CPU bound process is introduced
- a CPU-bound process will tie up the CPU, making the IO-bound processes progress at a much slower rate
- leads to long periods of idle I/O devices

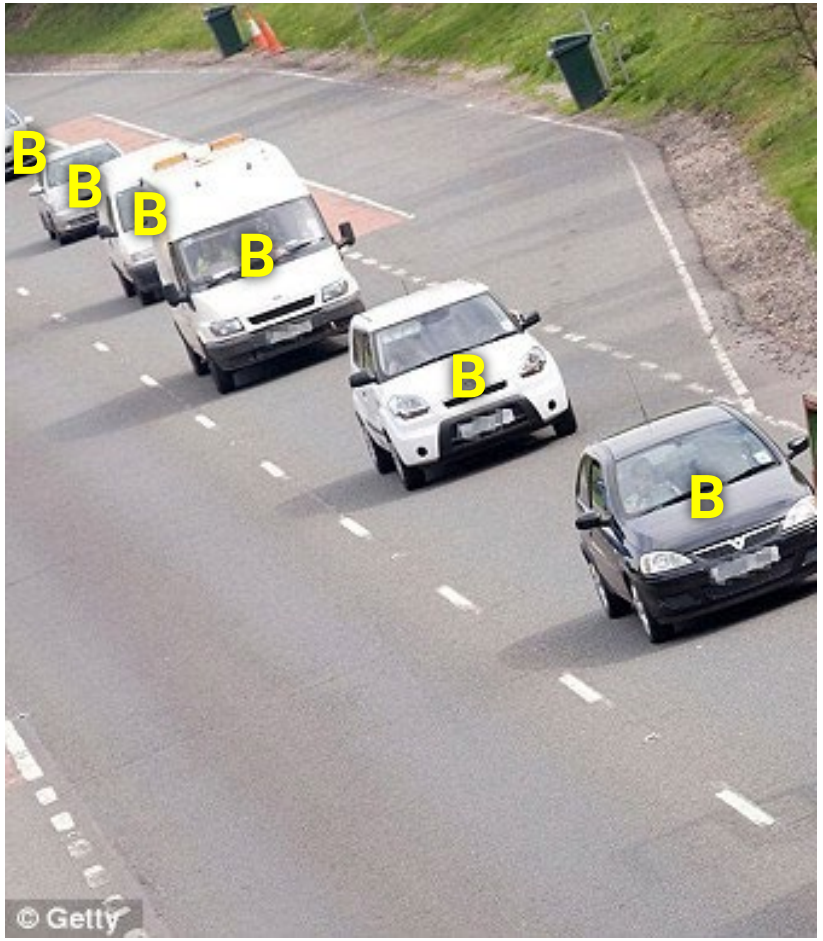
Convoy Effect Example

Example:

- many I/O-bound processes B
- assuming I/O operations can be done in parallel
- each B needs 1000 I/O operations, each 1/100s long, with negligible CPU bursts in between (e.g. 1/1000000s)

```
repeat 1000 times:
    do I/O for 0.01s
    use CPU for 0.000001s
```

- with only processes of type B in the system, each process B would finish in ~10 seconds



Convoy Effect Example

- let's introduce a CPU-bound process A, with 1s long CPU burst cycles

  repeat for long time:
  use CPU for 1.0s
  use I/O for 0.001s

- when A is present, each process B will take ~1000 seconds to execute
- a possible fix is to allow A to be preempted

Round-robin scheduling (RR)

Round-robin scheduling (RR)

- RR scheduler is a preemptive version of the FCFS scheduler
- each process is assigned a time interval, called a **time slice** (a.k.a. **quantum**)
  e.g., 10 msec, during which it is allowed to run
- if the process exceeds the quantum, the process is preempted (context switch),
  and CPU is given to the next process in ready queue
- preempted process goes at the back of the ready queue
- what if the process calls blocking system call?

RR scheduling

- construct a Gantt chart using quantum = 3 time units
- assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6				
P2	0	6				
P3	1	3				
P4	2	8				
P5	3	2				

RR - Simulating scheduling

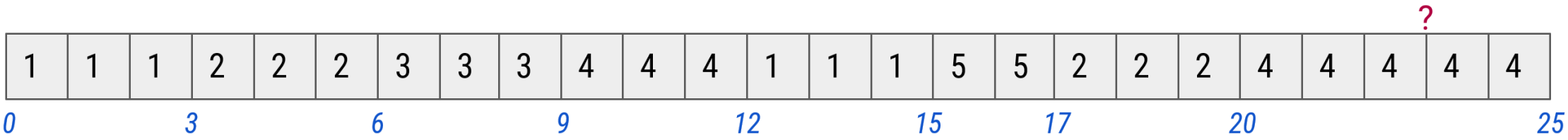
```
# Notation: <process #>/<bursts remaining>

T0  - CPU=[ ]   RQ=[]                    ; simulation start
T0  - CPU=[ ]   RQ=[1/6,2/6]             ; P1 arrives, P2 arrives
T0  - CPU=[1/6] RQ=[2/6]                 ; P1 starts executing
T1  - CPU=[1/5] RQ=[2/6,3/3]             ; P3 arrives
T2  - CPU=[1/4] RQ=[2/6,3/3,4/8]         ; P4 arrives
T3  - CPU=[2/6] RQ=[3/3,4/8,1/3]         ; P1 is preempted by P2
T3  - CPU=[2/6] RQ=[3/3,4/8,1/3,5/2]     ; P5 arrives
T6  - CPU=[3/3] RQ=[4/8,1/3,5/2,2/3]     ; P2 is preempted by P3
T9  - CPU=[ ]   RQ=[4/8,1/3,5/2,2/3]     ; P3 done
T9  - CPU=[4/8] RQ=[1/3,5/2,2/3]         ; P4 starts executing
T12 - CPU=[1/3] RQ=[5/2,2/3,4/5]         ; P4 is preempted by P1
T15 - CPU=[ ]   RQ=[5/2,2/3,4/5]         ; P1 done
T15 - CPU=[5/2] RQ=[2/3,4/5]             ; P5 executes
T17 - CPU=[ ]   RQ=[2/3,4/5]             ; P5 done
T17 - CPU=[2/3] RQ=[4/5]                 ; P2 executes
T20 - CPU=[ ]   RQ=[4/5]                 ; P2 done
T20 - CPU=[4/5] RQ=[]                    ; P4 executes
T25 - CPU=[ ]   RQ=[]                    ; P4 done, simulation ends
```

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2

RR scheduling

- construct a Gantt chart using quantum of 3 msec
- assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6	0	15	15	9
P2	0	6	3	20	20	14
P3	1	3	6	9	8	5
P4	2	8	9	25	23	15
P5	3	2	15	17	14	12



- **compressed execution order** (consecutive duplicates removed): P1, P2, P3, P4, P1, P5, P2, P4
- **average wait time**: 11 units, context switches: 8

Shortest-job-first scheduling (SJF)

Shortest-job-first scheduling (SJF)

- non-preemptive
- applicable to batch systems, where job length (expected execution time) is often known in advance

- when the CPU becomes available, it is assigned to the shortest job
  - shortest = shortest execution time
  - ties are resolved using FCFS

- SJF is similar to FCFS, but ready queue is sorted based on submitted estimate of execution time

SJF scheduling

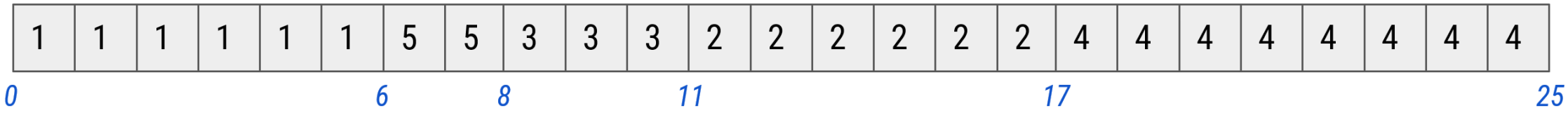
- construct a Gantt chart
- assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6				
P2	0	6				
P3	1	3				
P4	2	8				
P5	3	2				

SJF scheduling

- construct a Gantt chart
- assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6	0	6	6	0
P2	0	6	11	17	17	11
P3	1	3	8	11	10	7
P4	2	8	17	25	23	15
P5	3	2	6	8	5	3



- compressed execution order: P1, P5, P3, P2, P4
- average wait time: 7.2 units, context switches: 5

SJF scheduling

- advantages:
  - minimum number of context switches, just like FCFS
  - optimal turnaround time if all jobs arrive simultaneously
  - minimizes average waiting time

- disadvantages:
  - requires advanced knowledge of how long a job will execute
    - this is often known in batch job environments
  - has a potential for job starvation
    - long programs will never get to run if short programs are continuously added
    - could be solved by **aging** (increasing a job priority based on how long it has waited) and then sorting ready queue based on combination of priority and length

Shortest-remaining-time-next scheduling (SRTN)

Shortest-remaining-time-next scheduling (SRTN)

- preemptive improvement of SJF
- preemption can happen as a result of adding a new job
- next job is picked based on remaining time
  - remaining time = expected total execution time – time already spent on CPU

SRTN scheduling

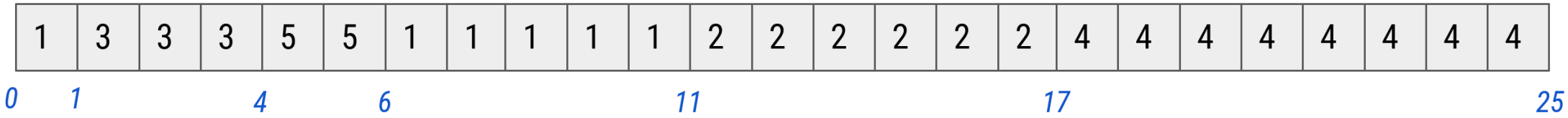
- fill out the table & construct a Gantt chart
- assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6				
P2	0	6				
P3	1	3				
P4	2	8				
P5	3	2				

SRTN scheduling

- fill out the table & construct a Gantt chart
- assume no I/O activity

Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	6	0	11	11	5
P2	0	6	11	17	17	11
P3	1	3	1	4	3	0
P4	2	8	17	25	23	15
P5	3	2	4	6	3	1



- condensed execution order: P1, P3, P5, P1, P2, P4
- average wait time: 6.4 units, context switches: 6

SRTN scheduling

- similar advantages to SJF:
  - optimal turnaround time even if jobs don't arrive at the same time

- similar disadvantages to SJF:
  - requires advanced knowledge of how long a job will execute
  - has a potential for job starvation
  - and also needs to consider cost of context switch

Multi-Level Queue

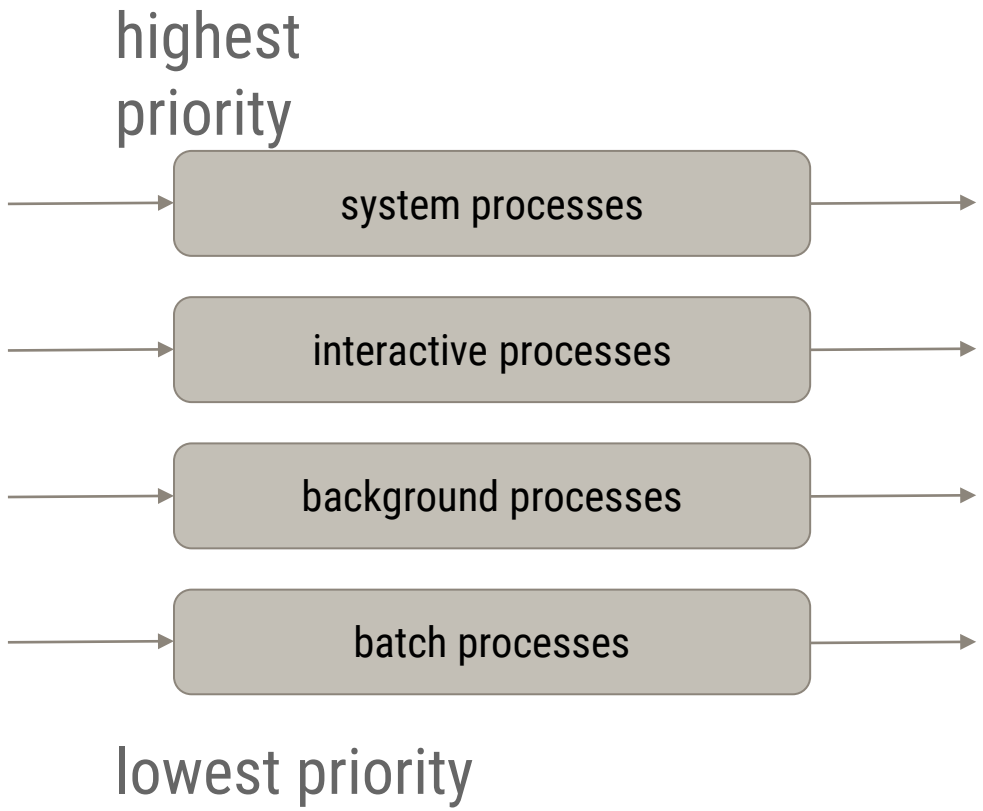
Multilevel queues

- preemptive time-sharing scheduling algorithm that supports process priorities
- ready queue is partitioned into separate queues, e.g.:
  - foreground queue – for interactive processes, such as browser, game
  - background queue – for non-interactive process, such as weather widget, web server
- each queue can have a different scheduling algorithm, e.g.:
  - foreground queue – using RR with time slice of 10ms
  - background queue – using RR with time slice of 100ms
- a process is permanently assigned to one of the queues
- scheduling is done based on queues …

Multilevel queues

option 1 – static priority scheduling:

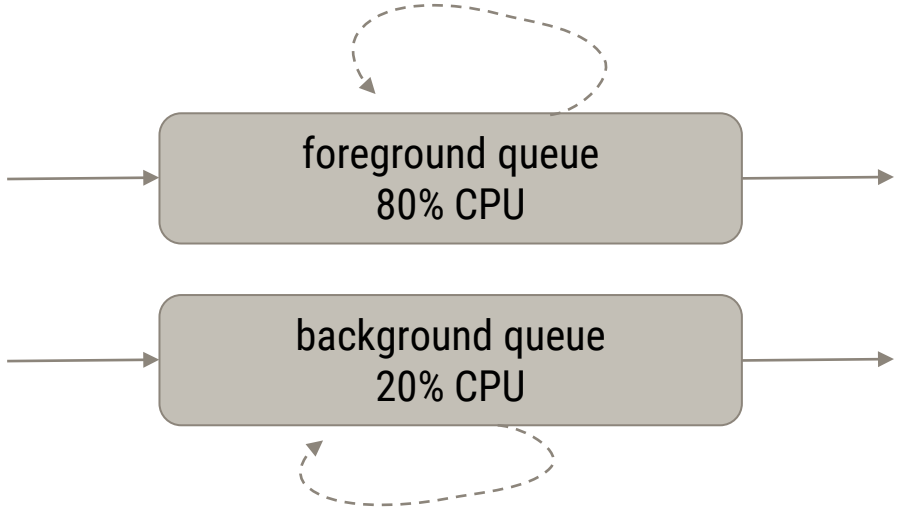
- each queue has a different, but fixed priority
- scheduler picks jobs from the highest priority queue, until it is empty
- only when empty, it switches to the next priority queue
- starvation is a problem
- not much different from single ready queue that is sorted based on priorities



Multilevel queues

option 2 – each queue gets a fixed CPU share

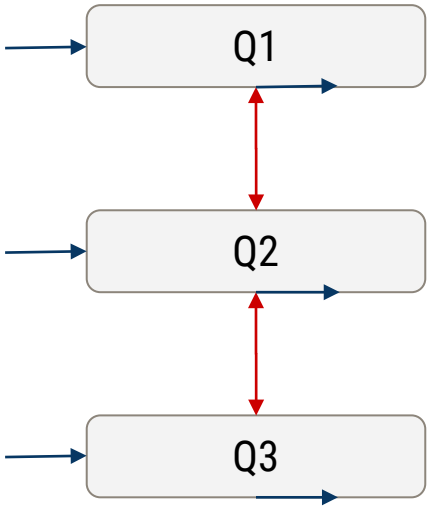
- example: 2 queues:
  - background queue gets 20% CPU
  - foreground queue gets 80% CPU
- processes in foreground queue would share 80% of CPU
- processes in background queue would share 20% of CPU
- not a very dynamic solution
- similar to fair share scheduling



Multi-Level Feedback Queue

Multilevel feedback queue scheduling

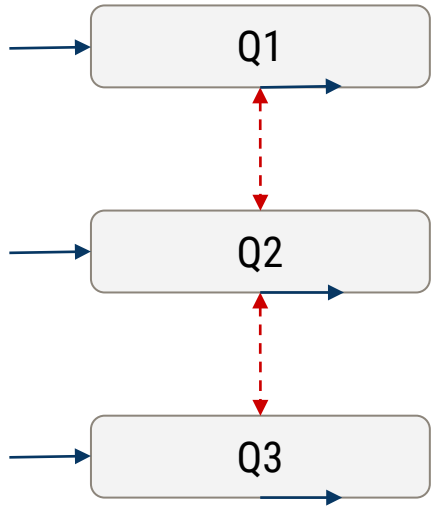
- similar to multilevel queues
  - multiple queues, each representing a different priority
  - scheduling is done based on queues
  - each process belongs to a queue
- but a process **can move** between queues (up or down in priority)



Multilevel feedback queue scheduling

- scheduler defined by
  - number of queues
  - different scheduling algorithms / quantums for each queue
  - when to move a process between queues
  - which queue a process will be assigned when the process needs service

- depending on the above:
  - can minimize starvation problem
  - can reduces convoy effect
  - can dynamically react to a job changing from CPU-bound to IO-bound
    - CPU-bound processes moved to low priority queue, IO-bound to high priority queue



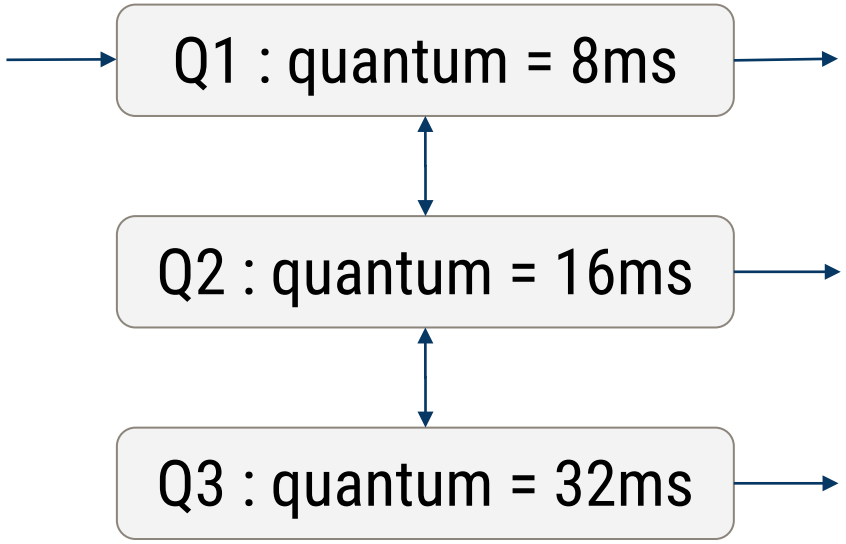
Multilevel feedback queue scheduling

Example configuration:

- three queues: Q1, Q2 and Q3
- all queues use RR and time-slices of 8ms, 16ms and 32ms
- jobs in Q1 are processed first, if empty then Q2, ... then Q3

- new jobs added to Q1

- if job in Q1 exceeds 8ms slice, it is demoted to Q2
- if job in Q2 exceeds 16ms slice, it is demoted to Q3
- if job in Q2 does I/O within 8ms, it is promoted to Q1
- if job in Q3 does I/O within 16ms, it is promoted to Q2



Shortest-process-next

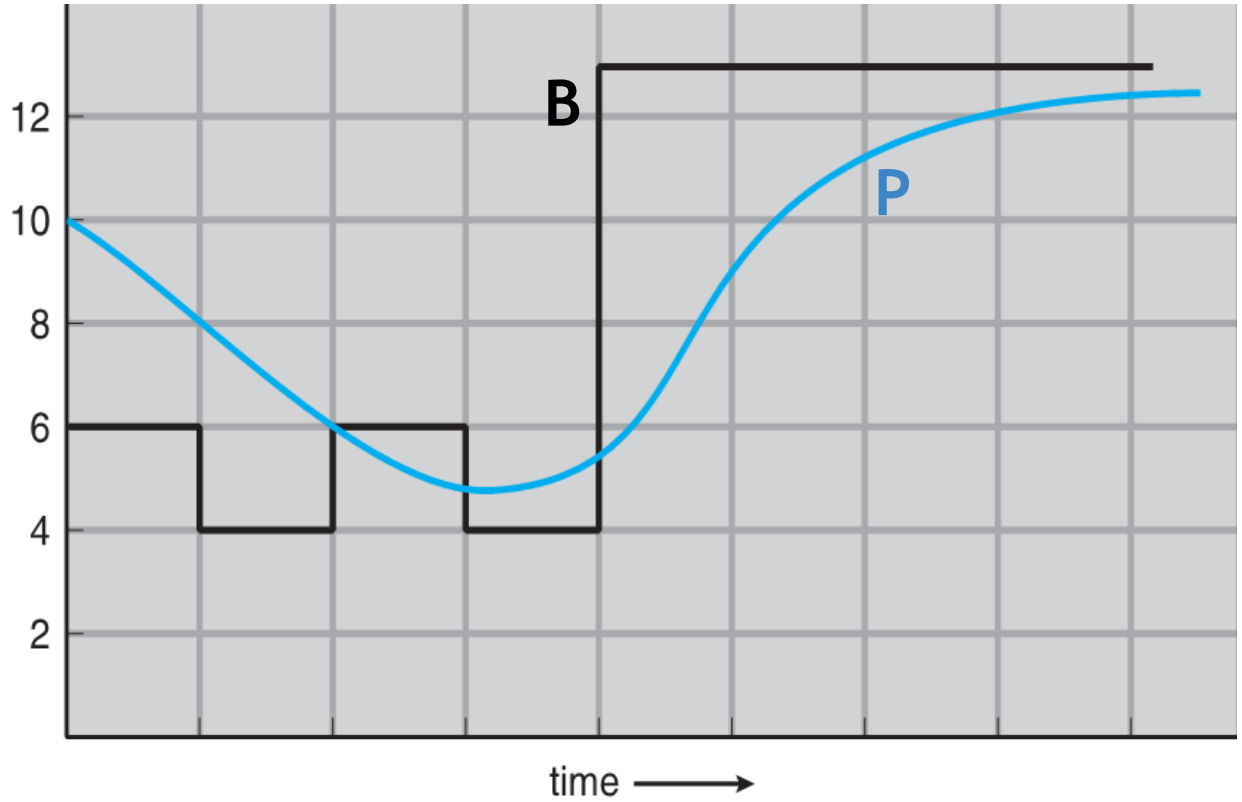
Shortest-process-next

- similar to SJF scheduling
  - uses time-sharing preemption (time slice)
  - ready queue is sorted by a **predicted** next CPU burst
  - prediction based on history of actual CPU bursts

- predicted burst can be calculated using **exponential smoothing**:
  after every burst **B**, we update the prediction **P** as follows:
  P = **a** x **B** + (1 - **a**) x **P'**
  where:
  P is the new prediction, initialized to some value (e.g. 0)
  P' is the previous prediction
  a is a smoothing factor, commonly set to ½

Predicting burst via exponential smoothing

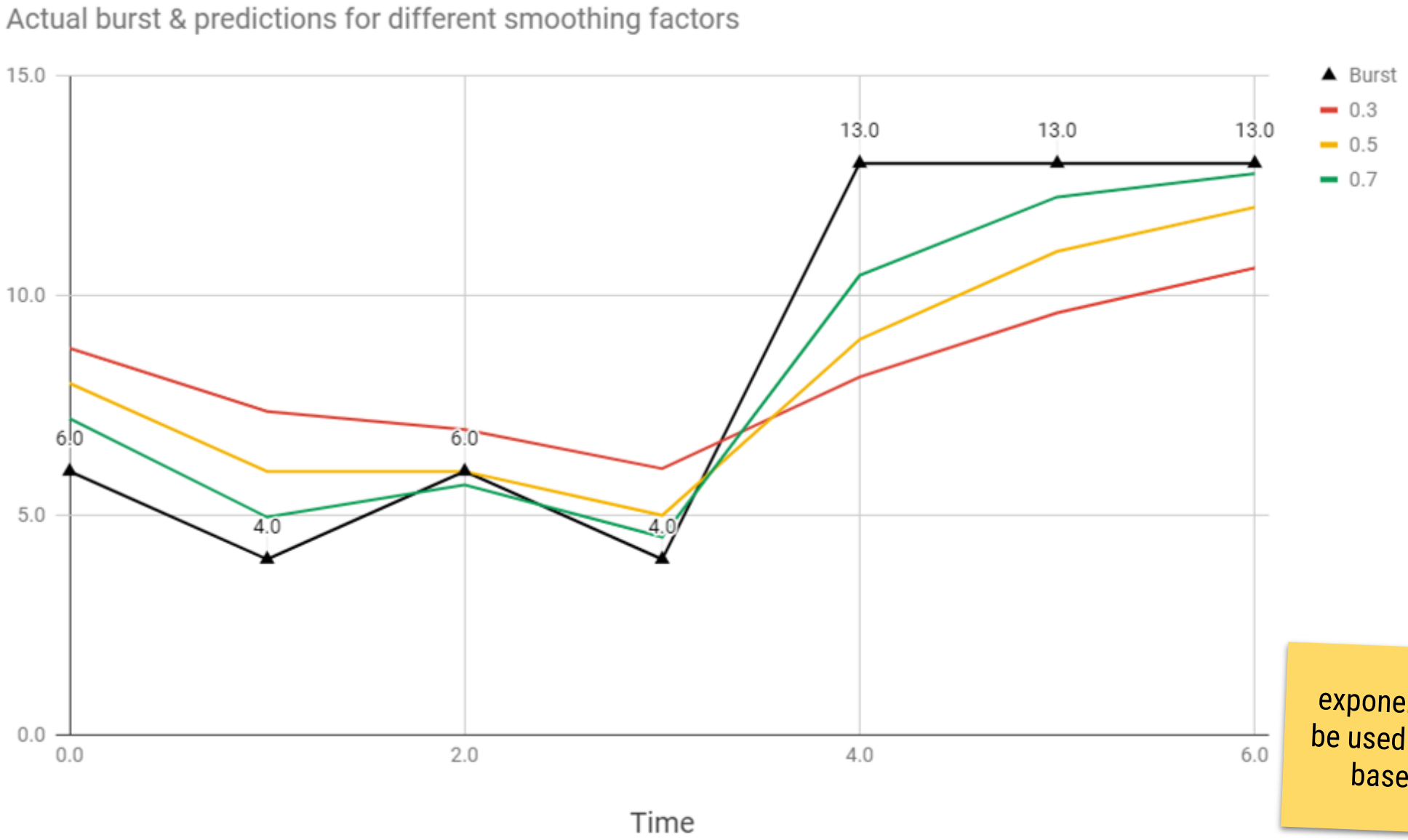
$$P = a \times B + (1 - a) \times P'$$



a=0.5

actual bursts	6	4	6	4	13	13	13	...
predicted burst P:	10	8	6	6	5	9	11	12	...

Exponential smoothing



exponential smoothing can be used to make noisy time-based data smoother

Fair-Share

Fair-share scheduling

- fair share scheduler takes into account the (importance of) owners of the processes
- ensures all users of a system get a fair share of the CPU
  by allocating CPU among users or groups, instead of processes
- solves problems like this:
  - user A runs 9 processes, user B runs a single process
  - user A would get 90% of the CPU, user B only gets 10%
- instead each user is allocated some fraction of the CPU
  - equal share: with **N** users, each user gets **1/N** of the CPU
  - unequal share: important users get a big chunk of CPU time,
    and even more important users get an even bigger chunk of CPU time
- then all processes belonging to an owner have to share the owner's CPU share

not related to Linux's completely-fair-scheduler

Fair-share scheduling

- example: two users, user #1 runs 5 processes: **A, B, C, D, E**, and user #2 runs 1 process: **F**

- scenario 1 – equal share, both users have 50% of the CPU
  possible scheduling sequences:
  A F B F C F D F E F A F B F C F D F E F A F B F C F D F E F …
  A B C D E F F F F F A B C D E F F F F F A B C D E F F F F F …

- scenario 2 – unequal share, user 1 has 75% CPU share, user 2 has 25% CPU share
  possible scheduling sequences:
  A B C F D E A F B C D F E A B F C D E F A B C F …
  A B C D E A B C D E A B C D E F F F F F …

Priority

Priority scheduling

- how do we handle processes with different priorities?

- why would some processes be more important than others?
  - e.g. priority could be determined by the VIP status of the owner
  - e.g. background service (spotify) could have lower priority than an interactive one (game)
  - … many other reasons

- priorities can be static or dynamic
  - static — e.g. bittorrent priority = 1, game priority = 10
  - dynamic — e.g. OS automatically increases I/O bound processes priority

- CPU allocation should reflect the priority of the process, higher priority ⟶ more CPU

Priority scheduling

- for batch systems, a simple approach to handle priorities could work:
  - modified SJF-like scheduling, where ready queue is sorted by priority
  - all jobs would eventually finish
  - but a new job with high priority might have to wait for already running lower priority job to finish first… could this be fixed? (maybe add preemption…)
  - starvation – low priority job might never start if higher priority jobs keep getting added, although unlikely to be a problem on batch systems in real world

- handling priorities with time-sharing preemptive scheduling is more complicated:
  - e.g. consider RR-like scheduling, where ready queue is sorted by priority
  - immediate problem is starvation
  - a high priority CPU-bound process would ensure no other processes get to run at all

Lottery

Lottery scheduling (probabilistic scheduling)

- preemptive time-sharing algorithm
- m
- each process gets some number of 'lottery tickets'
  - number of tickets based on process priority,
    higher priority → more tickets, but every process gets at least 1 ticket

- scheduler picks a random number and process with that ticket 'wins' a time slice
  - higher priority process has a higher chance of running
  - over long time, job's priority will determine the job's total CPU share

- cooperating processes may exchange tickets
  - allowing dynamic fine-tuning the priorities based on needs
  - e.g. in producer/consumer setting the producers could swap tickets with consumers

- starvation is not a problem (why?)

- there are some practical issues with implementing efficiently for large # tickets, # jobs, predictability

Real-Time Scheduling

Real-Time OS CPU scheduling

- programs are expressed as a set of **tasks** responding to events
- a task must respond within a fixed amount of time from time of event (**deadline**)
- correctness depends both on the logical result as well as the response time
- tasks deadline types:
  - **hard deadline**: must meet its deadline, miss will cause system failure, needs **Hard RTOS** e.g. missing interrupt in a pacemaker device
  - **soft deadline**: occasional miss is acceptable, needs **Soft RTOS**, e.g. game lag, LOD
  - also **firm deadline** - between hard/soft, infrequent miss is tolerable, but value of task completion is 0 after deadline, e.g. in automated manufacturing few bad products are OK
- task types:
  - **periodic**: occurring at regular interval/period
  - **aperiodic**: occurring unpredictably

Rate-monotonic scheduling algorithm (RM)

- RM is a preemptive scheduler, well suited for periodic tasks
- each task has a **static** priority, based on the inverse of the period of the task
  - shorter period = higher priority, longer period = lower priority
- when a task becomes runnable, task with highest priority preempts a task with lower priority
- a formula can be used to determine whether a set of tasks is schedulable
  - schedulable = a schedule exists where no deadlines are missed
  - a new task could be rejected if the system would become unschedulable

Earliest-deadline-first scheduling algorithm (EDF)

- EDF is another preemptive scheduler, with **dynamic** task priorities
- a task with a shorter deadline has a higher priority
  - scheduler picks a task with the earliest deadline
  - an event for higher priority task will preempt lower priority task

- optimal dynamic priority scheduling
  - if a schedule exists, EDF will find it
  - works better for aperiodic tasks than RM
  - can achieve 100% CPU utilization

- a formula can determine whether tasks can be scheduled without missing deadlines

EDF example

Task	Event time	Deadline	Burst
1	0	30	7
2	1	8	5
3	2	20	11
4	3	5	2

Gantt chart:

1	2	2	4	4	2	2	2	3	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1	1

Review

Recap

Environment	Scheduling Algorithms
Batch systems	▪ First come, first serve ▪ Shortest job first ▪ Shortest remaining time next
Interactive systems	▪ **Round robin** ▪ Shortest process next ▪ Fair share ▪ Lottery ▪ Multilevel queue ▪ **Multilevel feedback queue**
Real-time systems	▪ Rate-monotonic scheduling ▪ Earliest-deadline-first

Scheduling algorithms

Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized RR scheduling
FreeBSD	Yes	**Multilevel feedback queue**
Linux kernel before 2.6.0	Yes	**Multilevel feedback queue**
Linux kernel 2.6.0–2.6.23	Yes	O(1) scheduler
Linux kernel after 2.6.23	Yes	Completely Fair Scheduler
classic Mac OS pre-9	None	Cooperative scheduler
Mac OS 9	Some	Preemptive scheduler for MP tasks, and cooperative for processes and threads
macOS	Yes	**Multilevel feedback queue**
NetBSD	Yes	**Multilevel feedback queue**
Solaris	Yes	**Multilevel feedback queue**
Windows 3.1x	None	Cooperative scheduler
Windows 95, 98, Me	Half	Preemptive scheduler for 32-bit processes, and cooperative for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes	**Multilevel feedback queue**

Other schedulers

- **short-term scheduling** aka **CPU scheduling**
  - responsible for allocating CPU to running processes
    i.e. managing ready/blocking/waiting states
- **long-term scheduling**
  - important in batch systems
  - decides which programs to start running, which ones to delay
  - typical goal is to achieve good mix of CPU-bound and IO-bound processes
- **medium-term scheduling**
  - decides which programs to swap in/out when running low on resources, or when a process is idle for too long
- **I/O scheduling**
  - ordering the I/O queues to increase throughput

Onward to …
File Systems

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY