Anthony Ventresque

anthony.ventresque@ucd.ie

Operating Systems

COMP30640

Process Scheduling (2)

School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

Outline

- Understand the operating, benefits, and drawbacks of common scheduling algorithms
  - First Come First Served (FCFS) (FIFO) Scheduling
  - Round-Robin Scheduling (RR)
  - Priority Scheduling
  - Shortest Job First (SJF)
  - Shortest Remaining Time First (SRTF)
  - Multi-Level Feedback Scheduling

Scheduling Algorithms

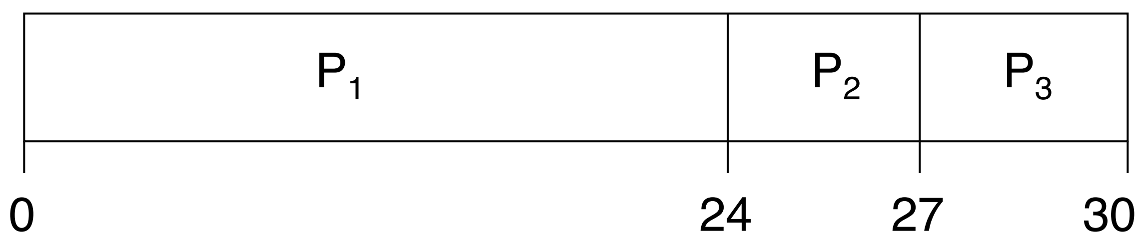
- The scheduling algorithm implements the system policies in order to achieve the scheduling objectives
- Its decisions may take into account
  - Preemptibility
  - Accountancy
  - Priorities
  - . . .

First-Come, First-Served

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - In early systems, FCFS meant one program scheduled until done (inc. I/O)
    - Now, means keep CPU until thread blocks
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

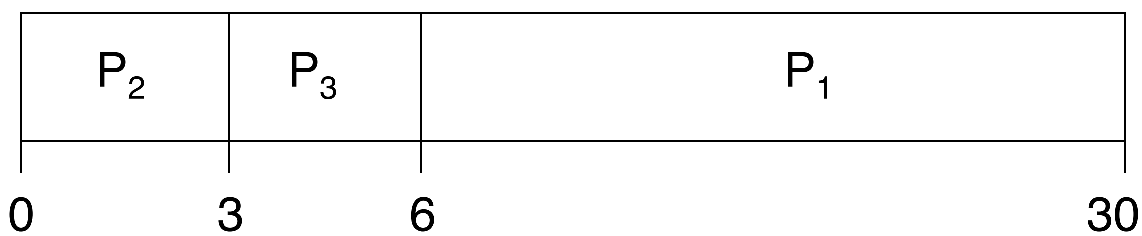
  - Suppose processes arrive in the order: P_1 , P_2 , P_3
    The Gantt Chart for the schedule is:



  - Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
  - Average waiting time: $(0 + 24 + 27)/3 = 17$
  - Average Completion time: $(24 + 27 + 30)/3 = 27$
- *Convoy effect:* short process behind long process
- FCFS favours CPU-bound processes

FCFS (cont'd)

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1
    Now, the Gantt chart for the schedule is:

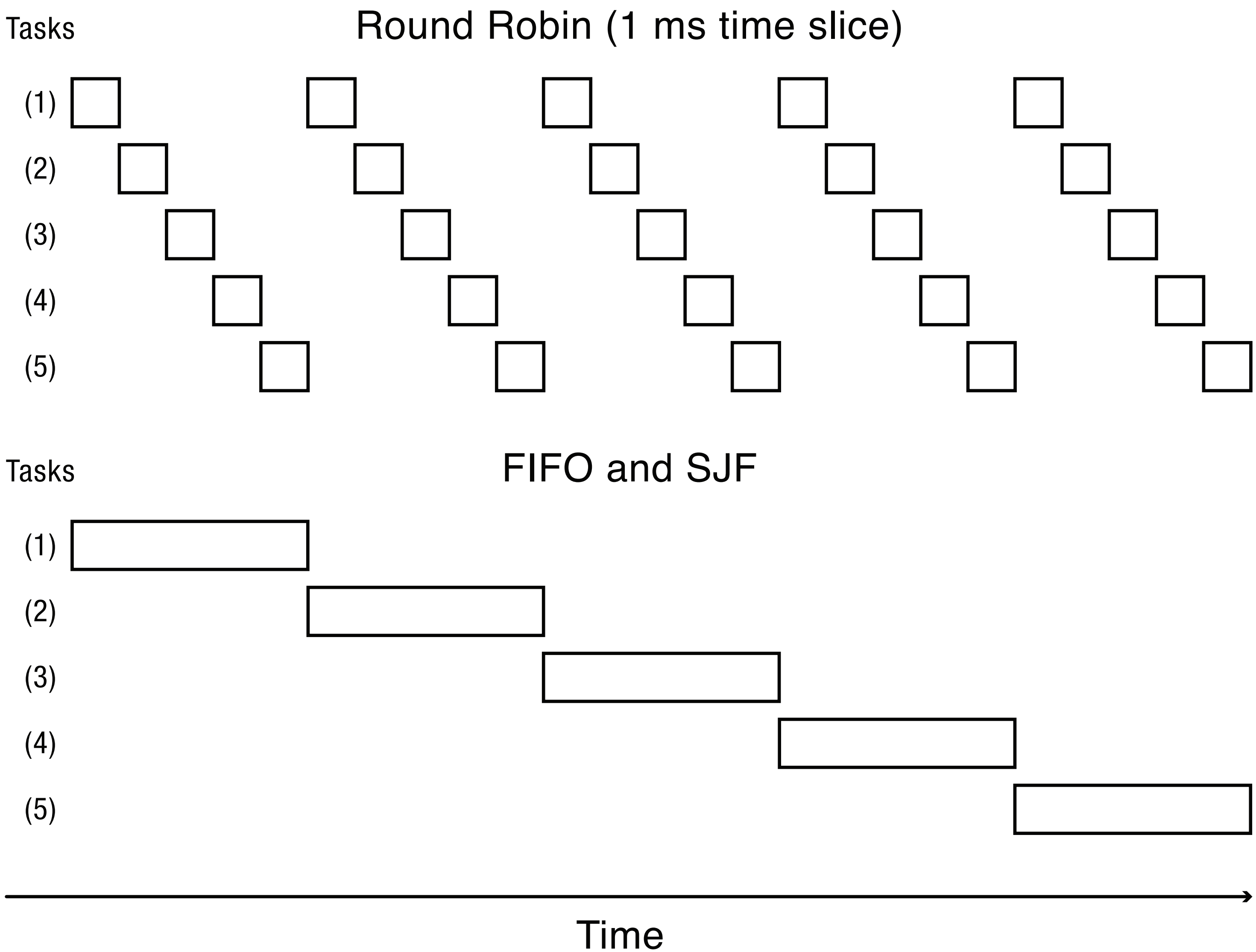


  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - Average waiting time: $(6 + 0 + 3)/3 = 3$
  - Average Completion time: $(3 + 6 + 30)/3 = 13$
- In second case:
  - average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - Stuck behind full trolley of small items at supermarket

Round Robin (RR)

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…
- Round Robin Scheme
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue
  - n processes in ready queue and time quantum is $q \Rightarrow$
    - Each process gets $1/n$ of the CPU time
    - In chunks of at most q time units
    - No process waits more than (n-1)q time units (Guaranteed)
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ Interleaved (really small $\Rightarrow$ hyperthreading?)
  - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

Round Robin vs. FCFS



Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

– The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 28 48 68 88 108 112 125 145 153

- Waiting time for:
  - P_1=(68-20)+(112-88)=72
  - P_2=(20-0)=20
  - P_3=(28-0)+(88-48)+(125-108)=85
    P_4=(48-0)+(108-68)=88
  - Average waiting time = (72+20+85+88)/4=66¼
  - Average completion time = (125+28+153+112)/4 = 104½
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Example with Different Time Quantum

Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]
0 – 8	8 – 32	32 – 85	85 – 153

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	$31\frac{1}{4}$
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	$61\frac{1}{4}$
	Q = 8	80	8	85	56	$57\frac{1}{4}$
	Q = 10	82	10	85	68	$61\frac{1}{4}$
	Q = 20	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$83\frac{1}{2}$
Completion Time	Best FCFS	85	8	153	32	$69\frac{1}{2}$
	Q = 1	137	30	153	81	$100\frac{1}{2}$
	Q = 5	135	28	153	82	$99\frac{1}{2}$
	Q = 8	133	16	153	80	$95\frac{1}{2}$
	Q = 10	135	18	153	92	$99\frac{1}{2}$
	Q = 20	125	28	153	112	$104\frac{1}{2}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

Round-Robin Discussion

- How do you choose time slice?
  - What if too big? Response time suffers
  - What if infinite (∞)? Get back to FIFO
  - What if time slice too small? Throughput suffers!

- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
  - In practice, need to balance short-job performance and long-job throughput:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

Comparison between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:
  - 10 jobs, each take 100s of CPU time; RR scheduler quantum of 1s; All jobs start at the same time
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average response time is much worse under RR!
    - Bad when all jobs same length

Priority Scheduling

- RR makes the implicit assumption that all processes are equally important, which is not reasonable in general
  - e.g. sending email vs real-time video playback
- Priority scheduling
  - Priority number (integer) is associated with each process
  - Priorities *quantify* the relative importance of processes
  - CPU allocated to the process with the highest priority
- Starvation issue: low priority processes might never execute
  - Solution: increase priority as time progresses (aging)

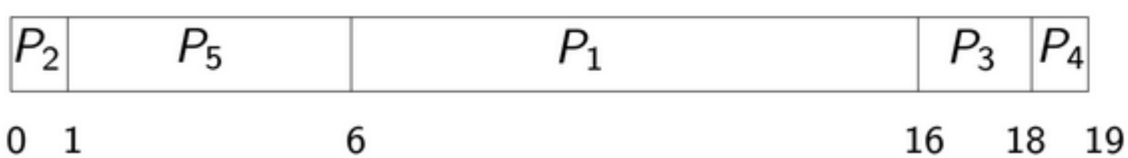
Priority Scheduling

- Priority scheduling can be either pre-emptive or non pre-emptive
- Adaptability:
  1. Static priorities:
     - Not responsive to environment changes, which could be exploited to increase throughput and reduce latency
     - easier to implement
  2. Dynamic priorities:
     - Responsive to change; e.g.: the OS may want to temporarily
     - Decrease the priority of a process holding a key resource needed by a higher-priority process
     - More complex to implement, overheads
- SJF can be seen as a type of priority scheduling, where priority is the predicted next CPU burst time (in this case a lower number means higher priority)

Example: Priority Scheduling

	P_1	P_2	P_3	P_4	P_5
burst time	10	1	2	1	5
priority	3	1	4	5	2

- (we assume here that a lower number means higher priority)

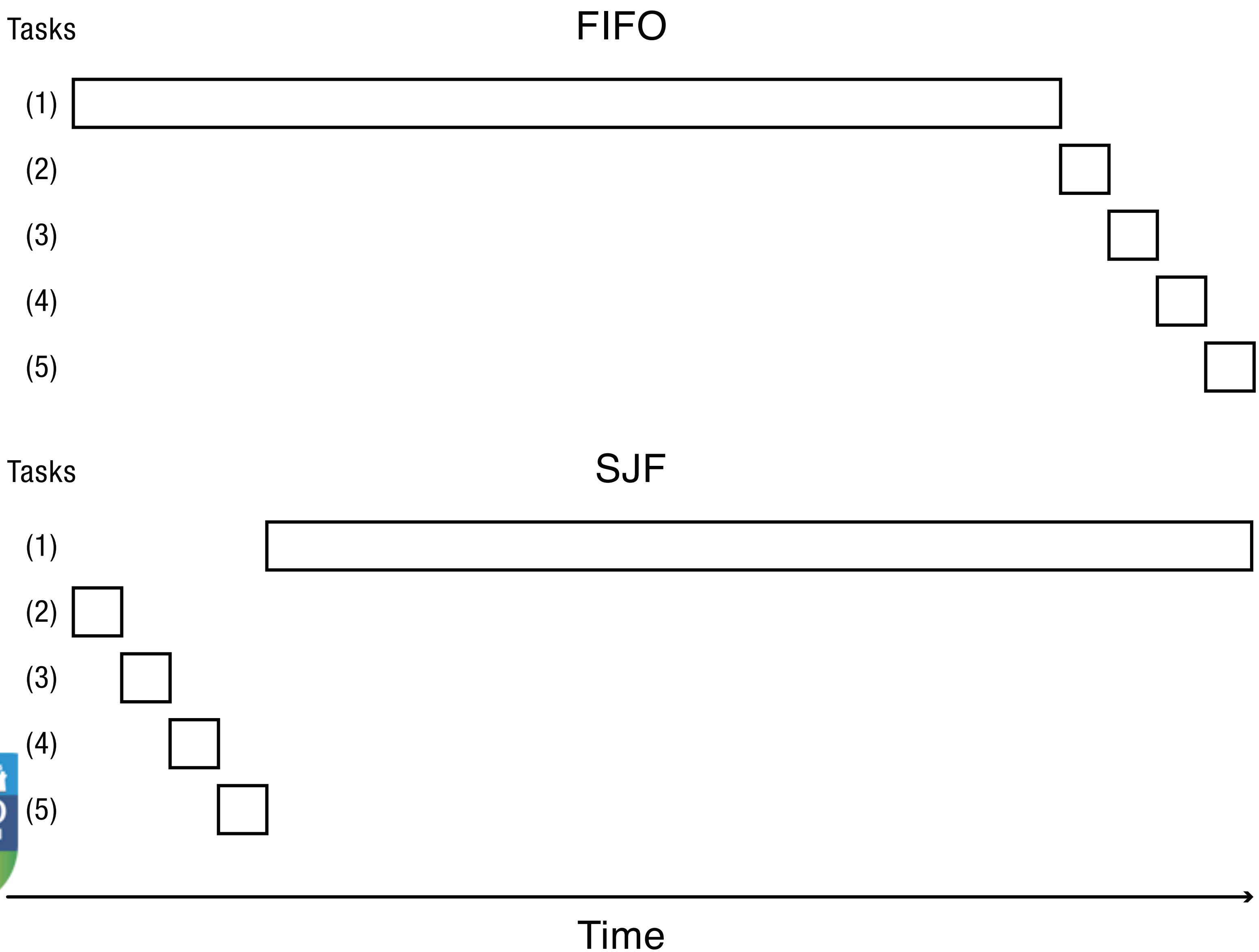


- Average waiting time: (0 + 1 + 6 + 16 + 18)/5 = 8.2 (assuming simultaneous arrival)

Shortest Job First (SJF) & Shortest Remaining Time First (SRTF)

- If we could predict CPU usage, could we always mirror best FCFS?
- **Shortest Job First (SJF)**:
  - Run whatever job has the least amount of computation to do
  - Sometimes called “Shortest Time to Completion First” (STCF)
- **Shortest Remaining Time First (SRTF):**
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

FIFO vs. SJF



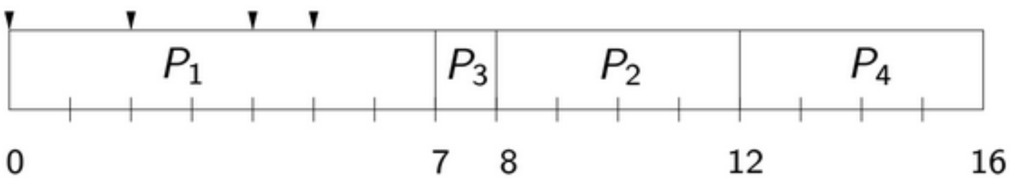
Discussion

- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
    - SJF gives the minimum average waiting time for a given set of processes (throughput is maximised)
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

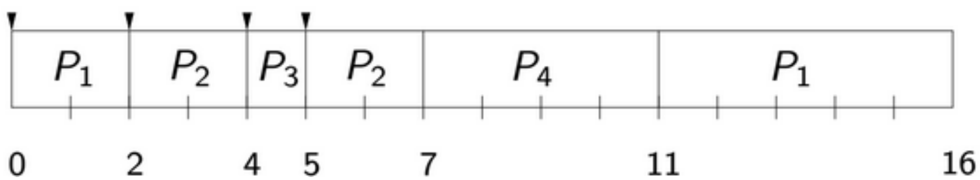
Example: Non pre-emptive and Pre-emptive SJF

	P_1	P_2	P_3	P_4
arrival time	0	2	4	5
burst time	7	4	1	4

- Non pre-emptive SJF

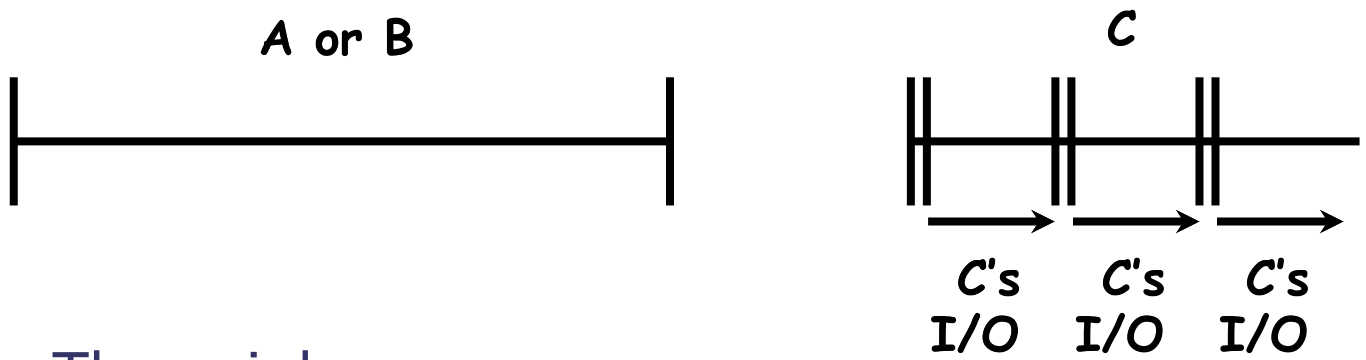


  - Average waiting time: (0 + 6 + 3 + 7)/4 = 4
- Pre-emptive SJF (SRTF)



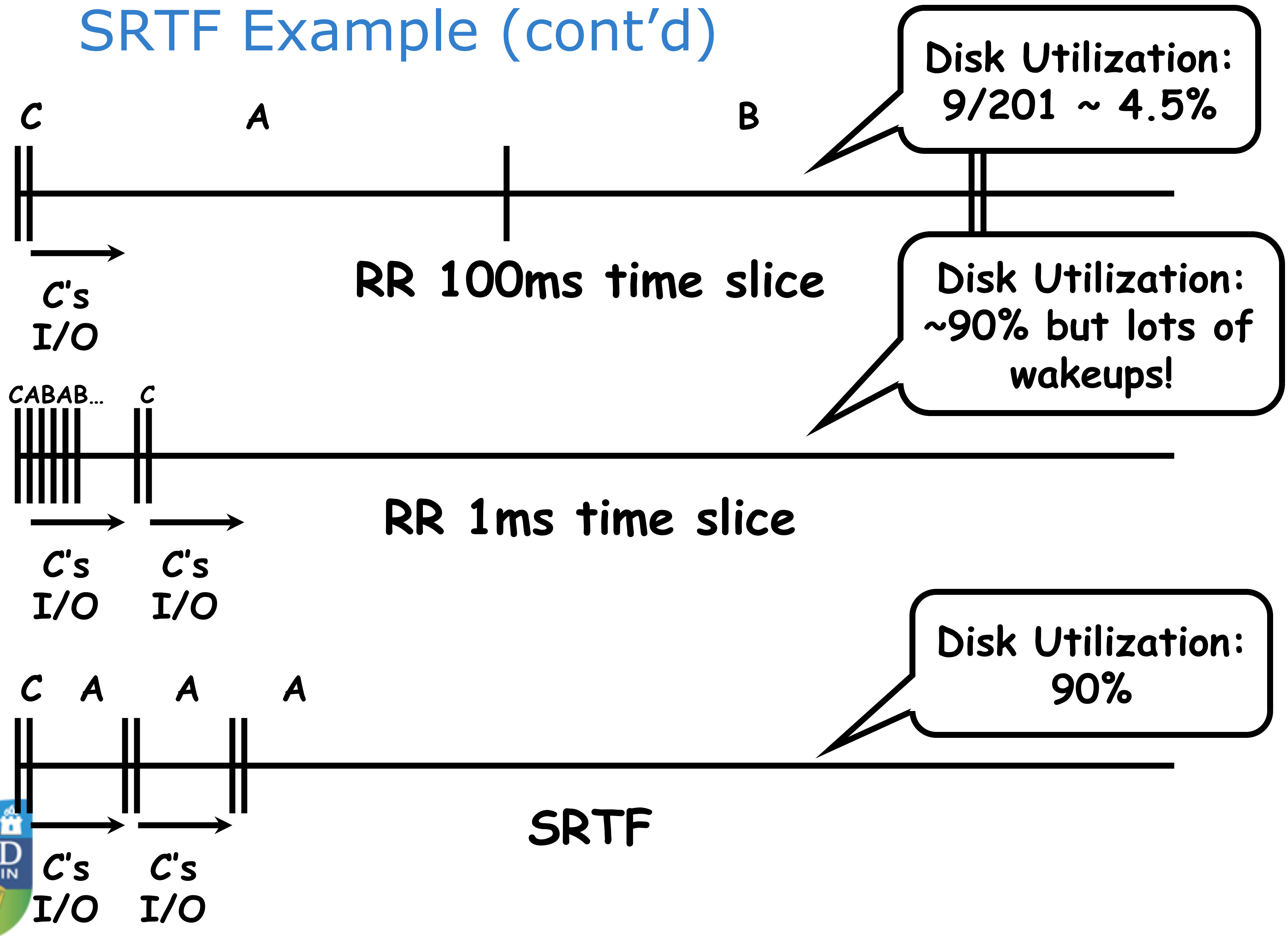
- Average waiting time: (9 + 1 + 0 + 2)/4 = 3

Example to Illustrate the Benefits of SRTF



- Three jobs:
  - A,B: both CPU bound, run for week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example (cont'd)



SRTF Further Discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if takes too long
  - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

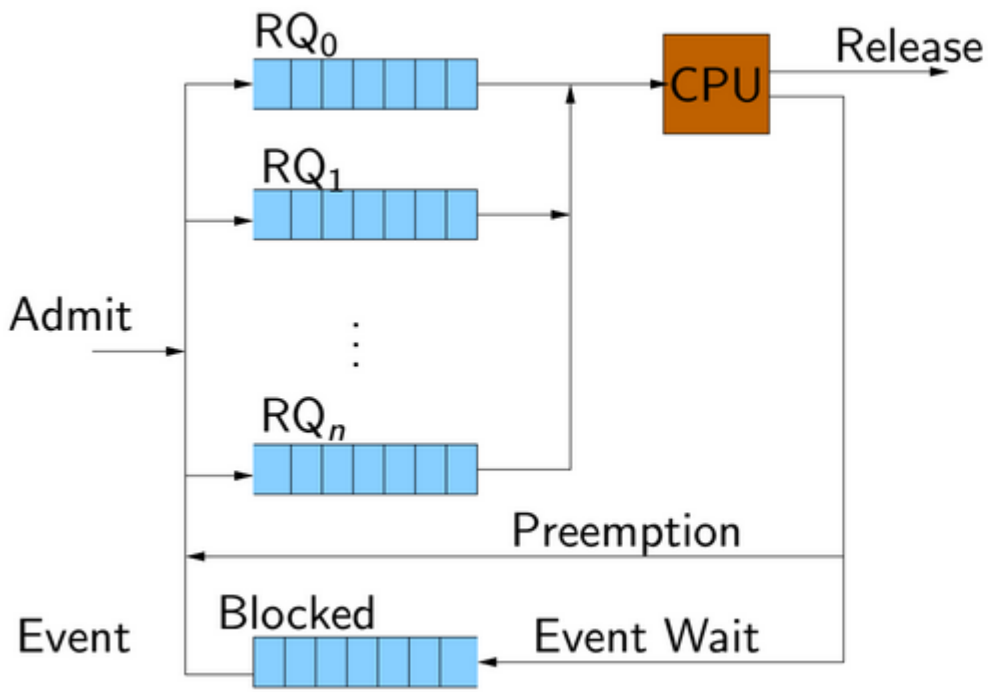
Multilevel Queue Scheduling

- The ready queue is partitioned into separate queues with their own scheduling algorithm
  - e.g.: Foreground queue (interactive) with RR, background queue (batch) with FCFS

- In multilevel queues there must be scheduling ***among the queues too***; common strategies are:
  1. Fixed priority scheduling
     - e.g.: serve first all processes from foreground queue, then all from background queue
     - Absolute precedence of higher-priority queues: possibility of starvation
  2. Time slice
     - Each queue gets a certain amount of CPU time which it can schedule among its processes
     - e.g.: 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue

- RQ0 system processes (highest priority)
- RQ1 interactive processes
- . . .
- RQn batch processes (lowest priority)

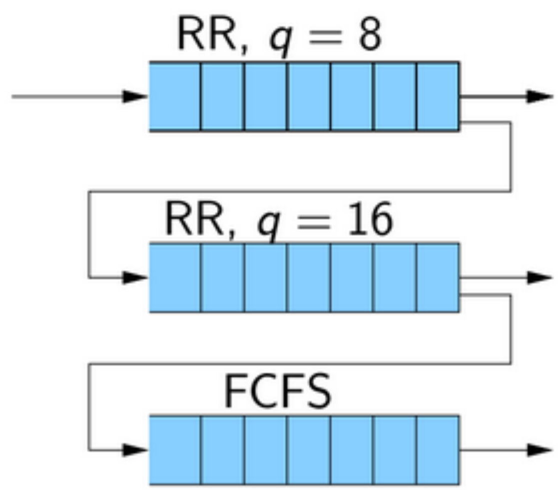


Multilevel Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)

Multilevel Feedback Queue

- Allows a process to move between queues
- Example:



- Longer processes are penalised and eventually moved to lower priority queues
- It affords <u>adaptability</u>: when a new process acquires the processor, the system does not know its burst pattern(CPU-bound, I/O-bound)

Traditional Unix Scheduling (SVR3, 4.3 BSD)

- Priority-based
- Multilevel queue feedback using RR within each queue
- 128 priorities in 32 queues (4 adjacent priorities); lower values, higher priority
- If a running process does not block or complete within one second, it is pre-empted
  - Kernel processes (priorities 0–49) cannot be pre-empted
  - Priorities are recomputed once per second, taking into account recent CPU usage
- A user can bias the priority of a process (towards less priority) using the nice command
  - Background processes (batch jobs) get higher nice values (i.e. less priority) automatically

Scheduling in Real-time Systems

- ***Real-time system***: meets the needs of processes that must produce correct output by a certain time (***deadlines*** or ***timing constraints***)
- We have seen that SJF is optimal considering the average waiting time, but it does not guarantee a fixed waiting time for any particular process
  - Therefore, it cannot be used in real-time scenarios
  - Consider CPU-bound processes verifying mission critical tasks
- If all processes can meet their deadlines regardless of their execution order, shortest deadlines first would be optimal

Categories of Real-time Scheduling

1. Soft real-time scheduling
   - Missing an occasional deadline is undesirable but tolerable
   - e.g.: multimedia playback
   - Priority scheduling required; real-time has highest priority
   - Small dispatch latency required (system calls should be pre-emptible)
2. Hard real-time scheduling
   - Absolute deadlines that always have to be met
   - e.g.: air traffic control
   - Special purpose software running on dedicated hardware

- Hard real-time: ***periodic*** or ***aperiodic*** (unpredictable) events
  - Assume *m* periodic events; event *i* occurs with period T_i times/second and requires t_i seconds of CPU to handle
  - These time constraints can only be met if
  - A real-time system that meets this criterion is ***schedulable***

Conclusion

- ***Scheduling***: selecting a waiting process from the ready queue and allocating the CPU to it
- ***FCFS (FIFO) Scheduling:***
  - Run threads to completion in order of submission
  - Pros: Simple and minimizes overhead
  - Cons: Short jobs get stuck behind long ones
  - If jobs are variable in size can have very poor average response time
  - If jobs are equal in size, FIFO is optimal in terms of average response time.
- ***Round-Robin Scheduling:***
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: If jobs are equal in size very poor average response time

Conclusion

- ***Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):***
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- ***Multi-Level Feedback Scheduling:***
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
  - Can achieve a balance between responsiveness, low overhead, and fairness

UCD DUBLIN