CSCI [4|6] 730 Operating Systems

CPU Scheduling

Scheduling Plans

- Introductory Concepts
- Embellish on the introductory concepts
- Case studies
- Look at real time scheduling.
  - Practical system have some theory, and lots of tweaking (hacking).

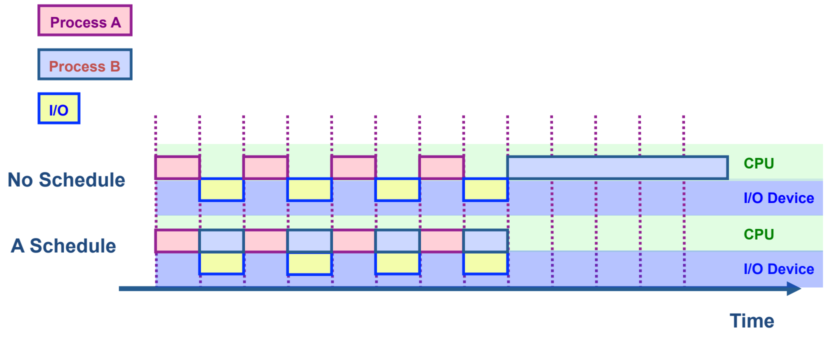
CPU Scheduling Questions?

- Why is scheduling needed?
- What is preemptive scheduling?
- What are scheduling criteria?
- What are disadvantages and advantages of different scheduling policies, including:
  - Fundamental Principles:
    - First-come-first-serve?
    - Shortest job first?
    - Preemptive scheduling?
  - Practical Scheduling (and case studies):
    - Hybrid schemes (Multilevel feedback scheduling?) that includes hybrids of SJF, FIFO, Fair Schedulers
      - Completely Fair Scheduling.
- How are scheduling policies evaluated?
  - What are important metrics?

Why Schedule? Management of Resources

- Resource: Anything that can be used by only a single [set] process(es) at any instant in time
  - Not just the CPU, what else?
- Hardware device or a piece of information
  - Examples:
    - CPU (time, time slice)
    - Tape drive, Disk space, Memory (spatial)
    - Locked record in a database (information, synchronization)
- Focus today managing the CPU, short term scheduling.

What is the Point? Can *Scheduling* make a difference?



- No Schedule vs A Schedule
- Schedule another waiting process while current CPU relinquish to CPU due to I/O.

Resource Classification

- Pre-emptable
  - Can forcibly removed the resource from a process (and possibly return it later) without ill effects.
- Non-preemptable
  - Cannot take a resource away from its current 'owner' without causing the computation to fail.

"*Resource*" Classification

- Preemptable (forcible removable)
  - Characteristics (desirable):
    - small state (so that it is not costly too preempt it).
    - only one resource
  - Examples:
    - CPU or Memory are typically a preemptable resources
- Non-preemptable (not forcible removable)
  - Characteristics:
    - Complicated state
    - May need many instances of this resource
  - Examples:
    - CD recorder - once starting to burn a CD needs to record to completion otherwise the end up with a garbled CD.
    - Blocks on disk

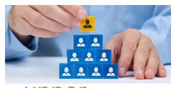
Resources Management Tasks

- Allocation (Space):
  - Space Sharing: Which process gets which resource (control access to resource)?
- Scheduling (Time):
  - Time Sharing: Which process gets resource and at what time ?
  - In which order should requests be serviced
  - Order and Time

Time and Space

The CPU Management Tea

- (How?)“The Dispatcher” (low level mechanism – the worker)
  - Context Switch
    - Save execution of old process in Process Control Block (PCB).
    - Add PCB to appropriate queue (ready or blocked)
    - Load state of next process from PCB to registers
    - Switch from kernel to user mode
    - Jump to instruction in user process



- (When?) “The Scheduler” (higher level mechanism - upper management) (schedules time).
  - Policy to determine when a specific process gets the CPU
- (Where?) Sometimes also “The Allocator” (space)
  - Policy to determine which processes compete for which CPU
  - Needed for multiprocessor, parallel, and distributed systems

Dispatch Mechanism

***Dispatcher** is the module that gives control of the CPU to the process selected by the scheduler.*

- OS runs dispatch loop:

```
while( forever )
{
   run process A for some time slice
   stop process A and save its context
   load context of another process B
   jump to proper location and restart program
}
```

- **Review: How does the dispatcher gain control?**

Recall: Entering System Mode

Same as - How does OS (scheduler) get control?

- Synchronous interrupts, or traps (produced by CPU).
  - Event internal to a process that gives control to OS
  - Examples: System calls, page faults (access page not in main memory), or errors (illegal instruction or divide by zero)
- Asynchronous interrupts: generated at arbitrary times.
  - Events external to a process, generated by other hardware
  - Examples: Characters typed, or completion of a disk transfer

How are interrupts handled?

- Each type of interrupt has corresponding routine (handler or interrupt service routine (ISR)
- Hardware saves current process and passes control to ISR

How does the dispatcher run?

Option 1: Cooperative Multi-tasking

- (internal events) Trust process to relinquish CPU through traps
  - Trap: Event internal to process that gives control to OS
  - Examples: System call, an explicit yield, page fault (access page not in main memory), or error (illegal instruction or divide by zero)
- Disadvantages: Processes can misbehave
  - By avoiding all traps and performing no I/O, can take over entire machine
  - Only solution: Reboot!
- Not performed in modern operating systems

How does dispatcher run?

Option 2: (external stimulus) True Multi-tasking

- Guarantee OS can obtain control periodically
- Enter OS by enabling periodic alarm clock
  - Hardware generates timer interrupt (CPU or separate chip)
  - Example: Every 10 ms
- User must not be able to mask timer interrupt
- Dispatcher counts interrupts between context switches
  - Example: Waiting 20 timer ticks gives the process 200 ms time slice
  - Common time slices range from 10 ms to 200 ms (Linux 2.6)

Scheduler Types

- Non-preemptive scheduler (cooperative multi-tasking)
  - Process remains scheduled until voluntarily relinquishes CPU (yields) – Mac OS 9.
  - Scheduler may switch in two cases:
    - When process exits
    - When process blocks (e.g. on I/O)
- Preemptive scheduler (Most modern OS, including most UNIX variants)
  - Process may be ‘de-scheduled’ at any time
  - Additional cases:
    - Process creation (another process with higher process enters system)
    - When an I/O interrupt occurs
    - When a clock interrupt occurs

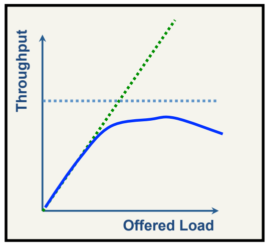
Scheduling Goals: Performance Metrics

- There is a tension between maximizing:
  - System's point of view: Overall efficiency (favoring the whole, the forest, the whole system).
  - User's point of view: Giving good service to individual processes (favoring the ‘individuals’, the trees).

Satisfy both : fast process response time (low latency) and high process throughput.

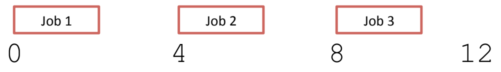
System View: Threshold - Overall Efficiency

- System Load (`uptime`):
  - The amount of work the system is doing
- Throughput:
  - Want many jobs to complete per unit time
- System Utilization:
  - Keep expensive devices busy
  - Jobs arrive infrequently and both throughput and system utilization is low
- Example: Lightly loaded system - jobs arrive infrequently - both throughput and system utilization is low.
- Scheduling Goal: Ensure that throughput increase linearly with load



Utilization / Throughput

- Problem type:
  - 3 jobs:
    - 1st job enters at time 0,
    - 2nd job at time 4, and
    - 3rd job at 8 second
  - Each job takes 2 seconds to process.
  - Each job is processed immediately – unless a job is on the CPU, then it waits



- Questions:
  - (1) What is the CPU utilization at time t = 12?
    - Consider the CPU utilization from t =0 to t=12.
    - Percentage used over a time period.
  - (2) What is the I/O device utilization at time t = 12?
  - (3) What is the throughput (jobs/sec)?

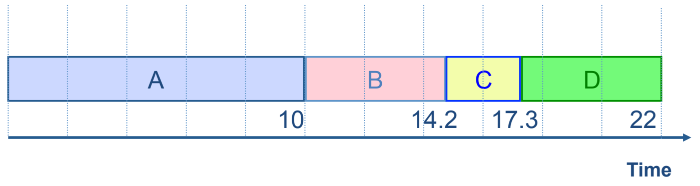
User View: Good Service (often measured as an *average*)

- Ensure that processes quickly start, run and completes.
- {average} **Turnaround time**: The time between
  - job arrival and
  - job completion.
- (average) **Response time**: The length of time when the job arrive and when if first start to produce output
  - e.g. interactive jobs, virtual reality (VR) games, click on mouse see VR change
- **Waiting time**: Time in ready queue - do not want to spend a lot of time in the ready queue
  - Better 'scheduling' quality metric than turn-around time since scheduler does not have control over blocking time or time a process does actual computing.
- **Fairness**: all jobs get the same amount of CPU over time
- **Overhead**: reduce number of context switches
- **Penalty Ratio**: Elapsed time / Required **Service** time
  - (normalizes according to the 'ideal' service time) - more on this next week
  - **Idle time** would expand elapsed time.
  - Service time is actual time on CPU (or/and IO device)

Which Criteria is Appropriate? Depends on *Expectation* of the System

- All Systems:
  - Fairness (give processes a fair shot to get the CPU).
  - Overall system utilization
  - Policy enforcement (priorities)
- Batch Systems (not interactive)
  - Throughput
  - Turn-around time
  - CPU utilization
- Real-time system (real time constraints)
  - Meeting deadlines (avoid losing data)
  - Predictability - avoid quality degradation in multimedia systems.

Gantt Chart (it has a name)



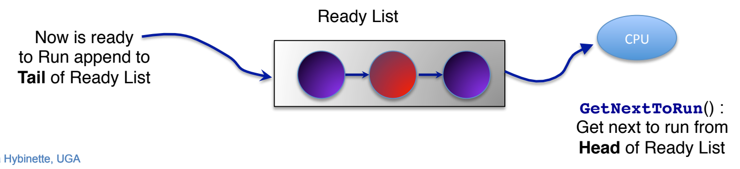
- Shows how jobs are scheduled over time on the CPU.

A Simple Policy: First-Come-First-Served (FCFC)

- The most basic scheduling policy is *first-come-first-served*, also called *first-in-first-out* (FIFO).
  - FCFS is just like the checkout line at the Publix.

  Maintain a queue ordered by time of arrival.

  GetNextToRun selects from the front of the queue.
- FCFS with pre-emptive time slicing is called *round robin* (more on that later)


Ready List
Now is ready to Run append to **Tail** of Ready List
CPU
GetNextToRun() : Get next to run from **Head** of Ready List

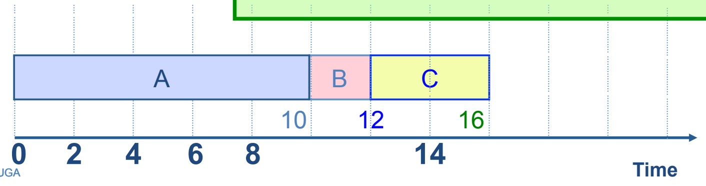
Evaluate: First-Come-First-Served (FCFS)

- Idea: Maintain FIFO list of jobs as they arrive
  - Non-preemptive policy
  - Allocate CPU to job at head of list (oldest job).

Job	Arrival	CPU burst
A	0	10
B	1	2
C	2	4

Average wait time:

Average turnaround time (enter/exit system):



First-Come-First-Served (FCFS)

- Idea: Maintain FIFO list of jobs as they arrive
  - Non-preemptive policy
  - Allocate CPU to job at head of list (oldest job).

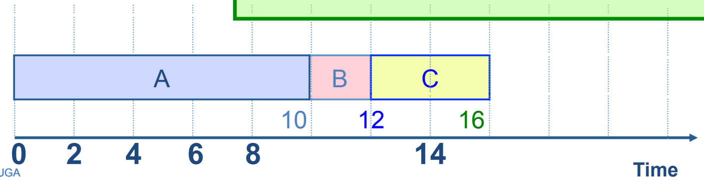
Job	Arrival	CPU burst
A	0	10
B	1	2
C	2	4

Average wait time:

```
  (0 +(10-1)+(12-2))/3   = 19/3 = 6.33
```

Average turnaround time (enter/exit system):

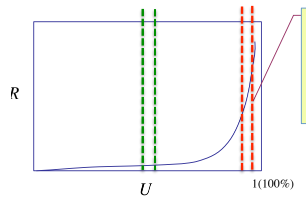
```
((10-0) +(12-1)+(16-2))/3 = 35/3 = 11.67
```



FCFS Discussion

- **Advantages:**
  - **Simple** implementation (less error prone)
  - Throughput is as good as any non-pre-emptive policy, if the CPU is the only schedulable resource in the system (rate of jobs completed).
  - Fairness – sort of – everybody eventually gets served (but not in terms of favoring long jobs – it is NOT FAIR!, positively bias towards long jobs, more common for short jobs to wait on a long job, not the other way around).
  - Intuitive
- **Disadvantages:**
  - Waiting time depends on arrival order
  - Response time: Tend to favor long bursts (CPU bound processes)
    - But : better to favor short bursts since they will finish quickly and "not crowd" the ready list.
  - Does not work on time-sharing systems (kind of… unless it is 'pre-emptive').

- **Response time** rises rapidly (degrades) with **load** and is unbounded.
  - At 50% utilization: a 10% load increase, increase response time by 10% (this is OK!)
  - At 90% utilization: a 10% load increase, increase response time by 10 times. (Oh my!).



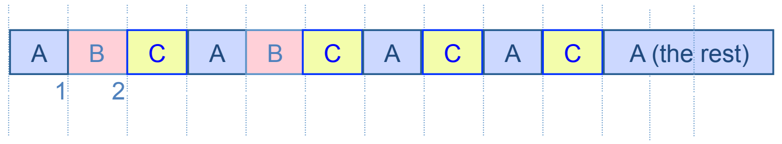
Here just a small increase in load causes a large increase in the expected response time

Pre-emptive FCFC: Round-Robin (RR)

- Idea: Run each job/burst for a *time-slice* or *quantum* (e.g., q=1) and then move to back of FIFO queue
  - Preempt job if still running at end of time-slice

Job	Arrival	CPU burst
A	0	10
B	1	2
C	1	4

Average wait:



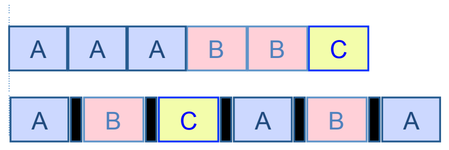
`Improves fairness – short jobs don't wait (as much) on long jobs anymore`

Non-Pre-emptive vs. Pre-emptive FCFS

- Example (quantum 1): Suppose jobs arrives at 'about' the same time (0), but slightly….
  - A is *before* B and
  - B is *before* C (time difference is insignificant, but not in terms of ordering)

Job	Arrival	CPU burst
A	0	3
B	0	2
C	0	1

Average turn-around time:



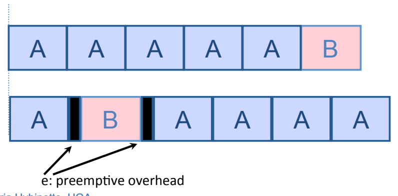
E: preemptive overhead

Pre-emptive FCFC: Round-Robin (RR)

- Another Example (quantum 1): Suppose jobs arrives at 'about' the same time (0), but A is before B (time difference is insignificant, but not in terms of ordering)

Job	Arrival	CPU burst
A	0	5
B	0	1

Average turn-around time:



Pre-emptive FCFC: Round-Robin (RR)

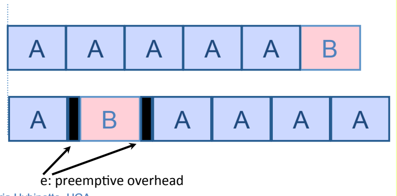
- Another Example (quantum 1): Suppose jobs arrives at 'about' the same time (0), but A is before B (time difference is insignificant, but not in terms of ordering)

Job	Arrival	CPU burst
A	0	5
B	0	1

Average turn-around time:

(5+6)/2 = 5.5

(2+6+2e)/2 = 4 + (1)*e



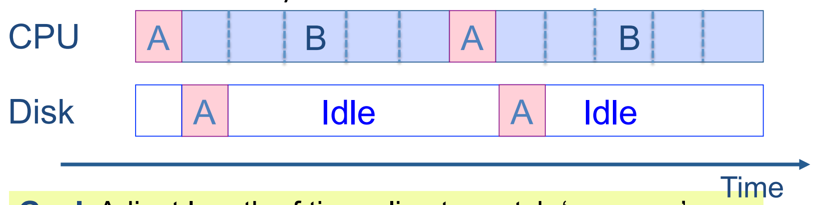
- **Response time**: RR reduces turn-around time for short jobs
- **Fairness**: RR reduces variance in wait time (but older jobs wait for newly arrived jobs)
- **Throughput**: context switch overhead (a quantum is 5-100 ms, **e** is on the order of micro seconds (us))

RR Discussion

- Advantages
  - Jobs get a fair share of CPU cycles
  - Shortest jobs finish relatively quickly
- Disadvantages
  - Poor average waiting time with similar job lengths
    - Example: 3 jobs that each requires 3 time slices
    - RR: All complete after about 9 time slices
    - FCFS performs better!
      - ABCABCABC = 4+5+6=15/3
      - AAABBBCCC = 0+3+6=9/3
  - **Observation**: Performance depends on **length** of time-slice
    - If time-**slice too short**, pay overhead of context switch (time slice matters, since it becomes more significant the smaller the time slice)
    - If time-**slice too long**, degenerate to FCFS (see next slide)

RR Time-Slice Consideratoins

- Time-slice too long, **degenerates** to problem of FCFS (short jobs wait on long jobs)
- Example:
  - Job A w/ 10 ms CPU burst/compute and 10 ms I/O burst
  - Job B always computes
  - Time-slice is 50 ms
- What about a really short time slices?



Goal: Adjust length of time-slice to match 'common' CPU bursts in job set

Shortest Jobs First (SFJ) - Minimizing …

- Shortest job first: optimal if the goal is to minimize response time or/and wait time.
  - Express lanes at Publix (fewer groceries, prioritize those customers).
- Idea: get short jobs out of the way quickly to minimize the number of jobs waiting for long jobs to finish.
- Examples of FCFS/FIFO service and see how SJF improves on FCFS/FIFO (hopefully!).

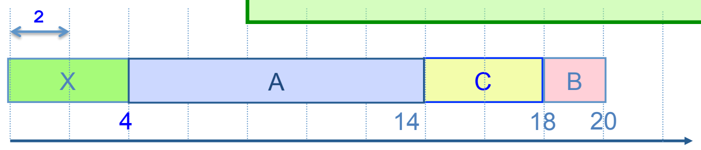
30/4 = 7.5
50/4 = 12.5

Another Example FIFO/FCFS

Job	Arrival	CPU burst
X	0	4
A	1	10
B	3	2
C	2	4

Average wait time:

Average turnaround time:



Time

Shortest-Job-First (SJF)

30/4 = 7.5
50/4 = 12.5

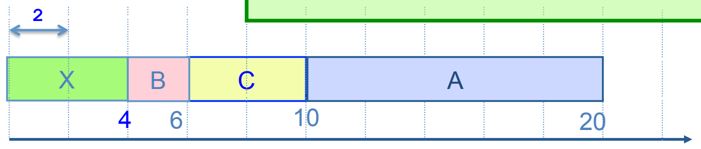
- Idea: Minimize average wait time by running shortest CPU-burst next
  - Non-preemptive policy
  - Use FCFS if jobs are of same length

14/4= 3.5
34/4= 8.5

Job	Arrival	CPU burst
X	0	4
A	1	10
B	3	2
C	2	4

Average wait time:

Average turnaround time:



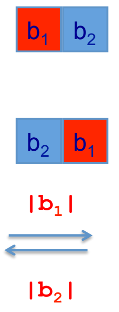
Time

Preliminary: Looking at an arbitrary scheduling policy

- Suppose we have a schedule S of average waiting time avg(W), and in this schedule we have two bursts: b_1 and b_2 , in the order b_1,b_2.
  - Lets denote | | as the length operator.
  - Assume $|b_1|$ = $|b_2|$ are of equal lengths.
- Question: IF we swap the order of b_1,b_2 to b_2, b_1 how does it impact avg(W)?

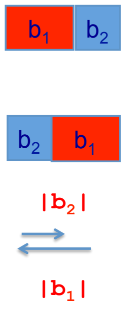
```
Increases b1 waiting time        +|b2|
Decreases b2 waiting time        -|b1|

Avg(W) +|b2|-|b1] = Avg(W) +|b1| -|b1] =  Avg(W)
```



Preliminary: Looking at an arbitrary scheduling policy

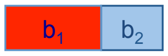
- What about if |b1| > |b2| and we swap them, i.e., we now change the schedule so that a shorter job, b2 , is scheduled ahead ahead of a longer job b1.
  - How does that impact avg(W)?
  - Is it the same, increased, or decreased?



```
Increases b1 waiting time        +|b2|
Decreases b2 waiting time        -|b1|

Avg(W) +|b2|-|b1] = Avg(W) +|b2| -|b1] =  Avg(W)
```

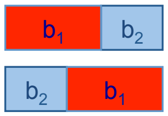
Proof: Optimality (Book)



```
OTHER=optimal
OTHER is not SJF
```

- Proof Outline: (by contraction) → Suppose SJF is not optimal, and there is some OTHER ordering that is optimal.
- In this scenario:
  - We have a set of bursts that are ready to run and we run them in some order that is OTHER than SJF.
    - AND we assume OTHER is an order that is Optimal
  - Then there must be some burst b_1 that is run before a shorter burst b_2 (because otherwise OTHER is SJF).
    - So, order is $b_1 b_2$, and $b_1 > b_2$
    - If we reversed the order we would:
      - increase the waiting time of b_1 by b_2 and (+b2)
      - decrease the waiting time of b_2 by b_1 (-b1)
  - Since **b1** is larger there is a net decrease (- is larger) in the total (waiting time)!
- Continuing in this manner to move shorter bursts ahead of longer ones, we eventually end up with the bursts sorted in increasing order of size (bubble sort). And now we are left with SJF schedule thus the contraction.

```
Find that burst
```



Optimality!!!

- SJF only optimal when all jobs are available simultaneously.
- See book for example why this is true.

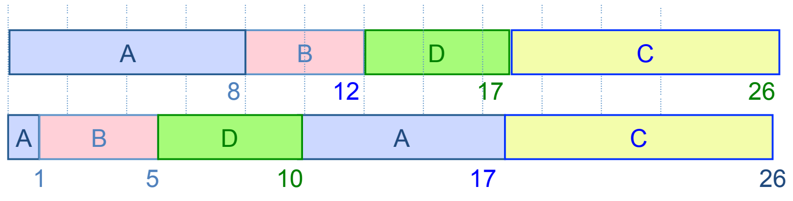
Shortest-Time-to-*Completion*-First (STCF/SCTF)

- Idea: Adds *preemption* to SJF
  - Schedule newly ready job if it has shorter than remaining burst for running job

Job	Arrival	CPU burst
A	0	8
B	1	4
C	2	9
D	3	5

SJF Average wait:

STCF Average wait:



SJF Discussion

- Advantages
  - Provably optimal for minimizing average wait time (with no preemption).
    - Moving shorter job before longer job improves waiting time of short job more than it harms waiting time of long job
  - Helps keep I/O devices busy
- Disadvantages
  - Problem: Cannot predict future CPU burst time
  - Approach: Make a good guess –
    - Use past behavior to predict future behavior
- Starvation: Long jobs may *never* be scheduled

$$G(n+1) = w * A(n) + (1-w)G(n)$$

Predicting Bursts in SJF

Guess = Actual Burst / 2 + Previous Guess / 2

- Key Idea: The past is a good predictor of the future (an optimistic idea) – 'habits'
  - Weighted averages of:
    - the most recent burst and the previous guesses (recursive) {history}
  - Approximate next CPU-burst duration from the durations of the previous burst and the previous guess). Average them.
  - Where we are going:
    - A recursive formula: accounts for entire past history, previous burst always important –
      - previous guesses and their importance drops of 'exponentially' with the time of their burst.

$$G(n+1) = w * A(n) + (1-w)G(n)$$

Example

- Suppose process p is given default expected burst length of 5 time units when it is initially run.
- Assume: The ACTUAL bursts length are:
  - 10, 10, 10, 1, 1,1
  - Note that these are of-course these are not known in advance.
- The predicted burst times for this process works as follows:
  - Let G(1) = 5 as default value
  - When process p runs, its first burst actually runs 10 time units (see above)
- so A(1) = 10.

- We could weigh the importance of the past with the most recent burst differently (but they need to add up to 1).

$$G(n+1) = w * A(n) + (1-w)G(n)$$

- w = 1 (past doesn't matter).
- How do we get started – no bursts before we start so what is the 'previous' burst G(1).
  - G(1) is a default burst size (e.g., 5).

$$guess = \frac{previous\ burst}{2} + \frac{previous\ guess}{2}$$

- Let b_1 be the most recent burst, b_2 the burst before that b_3 the burst before that b_4

$$guess = \frac{b_1}{2} + \frac{\frac{b_2}{2} + \frac{b_3 + \frac{b_4}{2}}{2}}{2}$$

$$guess = \frac{b_1}{2} + \frac{b_2}{4} + \frac{b_3}{8} + \frac{b_4}{16}$$

$$G(n+1) = w * A(n) + (1-w)G(n)$$

Example

- G(1) = 5 as default value
- A(1) = 10.

```
G(2) = 1/2 * G(1) + 1/2 A(1) = 1/2 * 5.00 + 1/2 * 10 = 7.5
G(3) = 1/2 * G(2) + 1/2 A(2) = 1/2 * 7.50 + 1/2 * 10 = 8.75
G(4) = 1/2 * G(3) + 1/2 A(3) = 1/2 * 8.75 + 1/2 * 10 = 9.38
```

Priority Based (typical in modern OSs)

- Idea: Each job is assigned a priority
  - Schedule highest priority ready job
  - May be preemptive or non-preemptive
  - Priority may be static or dynamic
- Advantages
  - Static priorities work well for real time systems
  - Dynamic priorities work well for general workloads
- Disadvantages
  - Low priority jobs can starve
  - How to choose priority of each job?
- Goal: Adjust priority of job to match CPU burst
  - Approximate SCTF by giving short jobs high priority

How Well do the Algorithms Stack UP

- Utilization
- Throughput
- Turnaround time: The time between job arrival and job completion.
- Response time: The length of time when the job arrive and when if first start to produce output
  - e.g. interactive jobs, virtual reality (VR) games, click on mouse see VR change
- Meeting Deadlines (not mentioned)
- Starvation

How to the Algorithms Stack Up?

	CPU Utilization	Through put	Turn Around Time	Response Time	Deadline Handling	Starvation Free
FIFO	Low	Low	High	High	No	Yes
Shortest Remaining Time	Medium	High	Medium	Medium	No	No
Fixed Priority Preemptive	Medium	Low	High	High	Yes	No
Round Robin	High	Medium	Medium	Low	No	Yes

Penalty Ratio (normalized to an ideal system)

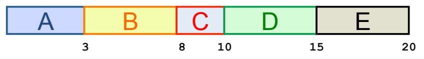
$$\text{Penalty ratio} = \frac{\text{Total elapsed time (actual)}}{\text{Service time: doing actual work (on CPU + doing I/O)}}$$

- Comparison to an ideal system: How much time worse is the turn-around time compared to an ideal system that would only consist of 'service time' (includes waiting)
  - Note this really measure of how well the scheduler is doing.
- Lower penalty ratio is better (actual elapsed time takes the same time as an idea system).
- Examples:
  - Value of "1" indicates 'no' penalty (the job never waits)
  - 2 indicates it takes twice as long than an ideal system.

Example using

Job	Arrival	CPU burst
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

- First Come First Serve
- Penalty Ratio – turn-around time (over ideal)

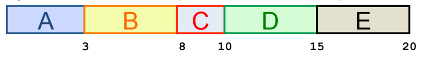


Job	Start Time	Finish Time	Waiting Time	Penalty Ratio
A				
B				
C				
D				
E				
avg				

Example using (CPU Only)

Job	Arrival	CPU burst
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

- First Come First Serve
- Penalty Ratio – turn-around time (over ideal – the burst itself)



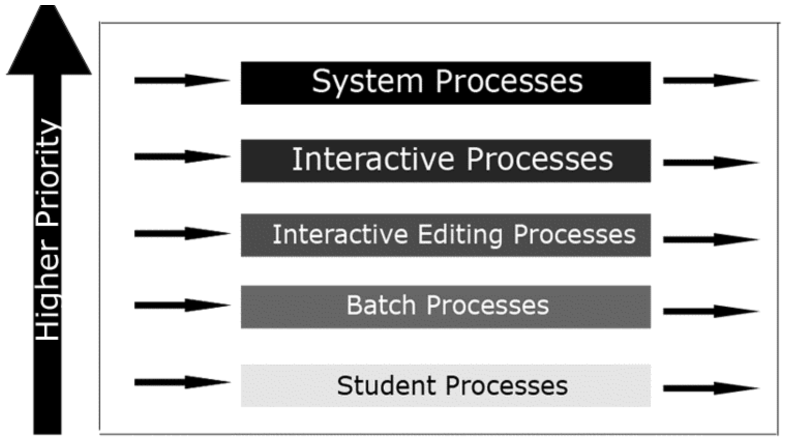
- **Shortest Burst worst PR.**
- **Even worse:**
  - **long burst at 0, takes 100 units**
  - **short burst at 1**
    - **Wait 99.**
    - **(101-1)/1 = 100**

Job	Start Time	Finish Time	Waiting Time	Penalty Ratio	
A	0	3	0	1.0	3/3
B	1	5	2	1.4	7/5
C	3	2	5	3.5	
D	9	5	1	1.2	
E	12	5	3	1.6	
avg			2.2	1.74	

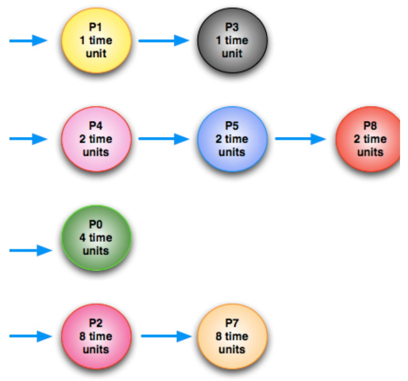
Multilevel Queue Scheduling

- Classify processes and put them in different scheduling queues
  - Interactive, batch, etc.
- Different scheduling priorities depending on process group priority
- Schedule processes with highest priority first, then lower priority processes.
- Other possibility : Time slice CPU time between the queues (higher priority queue gets more CPU time).

Multilevel Queue Scheduling



Multilevel *Feedback* Queue



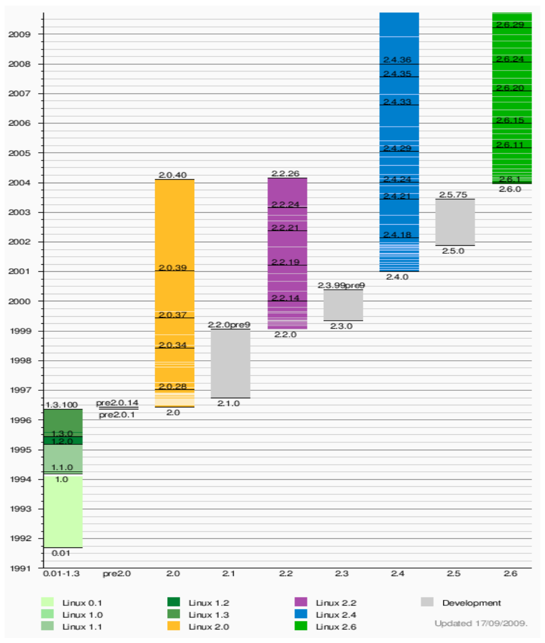
- Give new processes high priority and small time slice (preference to smaller jobs)
- If process doesn’t finish job bump it to the next lower level priority queue (with a larger time-slice).
- Common in interactive system

Case Studies: Early Scheduling Implementations

- Windows and Early MS-DOS
  - Non-Multitasking (so no scheduler needed)
- Mac OS 9
  - Kernel schedule processes:
    - A Round Robin Preemptive (fair, each process gets a fair share of CPU
  - Processes
    - schedules multiple (MACH) threads that use a cooperative thread schedule manager
      - each process has its own copy of the scheduler.

Case Studies: Modern Scheduling Implementations

- Multilevel Feedback Queue w/ Preemption:
  - FreeBSD, NetBSD Solaris, Linux pre 2.5
  - Example Linux: 0-99 real time tasks (200ms quanta), 100-140 nice tasks (10 ms quanta -> expired queue)
- Cooperative Scheduling (no preemption)
  - Windows 3.1x, Mac OS pre3 (thread level)
- O(1) Scheduling
  - time to schedule independent of number of tasks in system
  - Linux 2.5-2.6.24 ((v2.6.0 first version ~2003/2004)
- Completely Fair Scheduler (link read)
  - Maximizes CPU utilization while maximizing interactive performance / Red/Black Tree instead of Queue
  - Linux 2.6.23+


pre2.0
pre2.0.1
pre2.0.14
2.2.0pre9
2.3.99pre9
Linux 0.1
Linux 1.0
Linux 1.1
Linux 1.2
Linux 1.3
Linux 2.0
Linux 2.2
Linux 2.4
Linux 2.6
Development
Updated 17/09/2009.