50004 - Operating Systems - Lecture 4

Oliver Killane

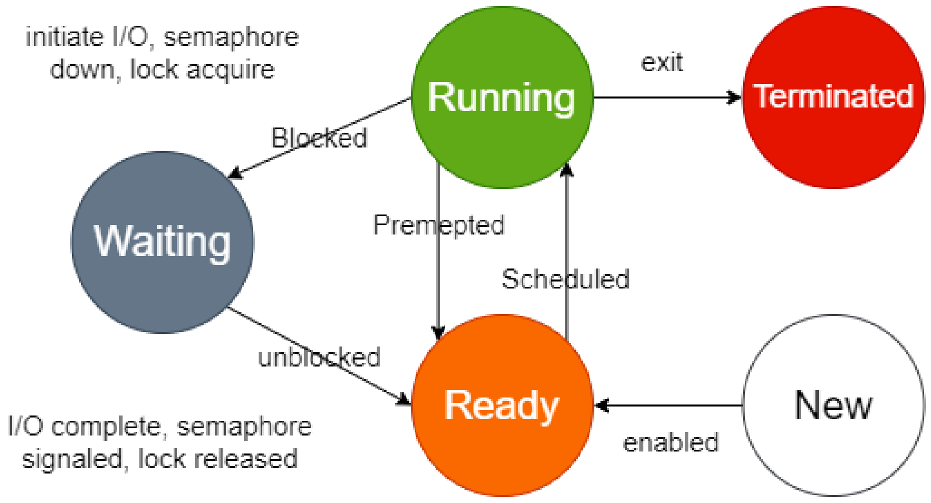
23/10/21

Lecture Recording

Lecture recording is available here

Scheduling

Process States


initiate I/O, semaphore down, lock acquire
Running
exit
Terminated
Blocked
Waiting
Premepted
Scheduled
unblocked
Ready
New
I/O complete, semaphore signaled, lock released
enabled

Scheduling concerns determining which ready process should be run, when and for how long.

Scheduling Goals

- **Fairness** Comparable processes get comparable resource allocation (e.g CPU time).
- **Avoid Indefinite Postponement** No process should starve (be left with no CPU time at all, effectively not running at all).
- **Enforce Policy** For example enforcing certain processes (e.g system processe) are prioritised.
- **Maximize Resource Utilisation** If there is a process that can be run, it should be run. Resources go beyond CPU time to resouces such as disk, or I/O.
- **Minimise Overhead** The scheduling system itself should use as little resources as possible. (e.g minimise scheduling decisions and frequency of context switches).

There are different priorities for different types of systems

Scheduler Types & Systems

- **Batch Systems** Optimise for throughput (jobs per unit time) or turnaround time (time from submission to completion).
- **Interactive Systems** Response Time is crucial (time from request issued to first response).

- **Real Time Systems** Need to meet deadlines. Soft: (recoverable) e.g reduced video quality.
  Hard: (irrecoverable) e.g factory robots collide.

There are two main types of scheduler

- **Non-Preemptive**
  - Processes are allowed to continue to run until they become blocked, or voluntarily yield the CPU.
  - Typically this requires software to be trusted to not monopolise the CPU maliciously.
  - Bad for interactivity as a process may be waiting for a long time.
  - Good for batch systems as sheduler overhead is typically small.
- **Preemptive**
  - Processes can run for a maximum amount of time (time slice) before being descheduled.
  - Requires a clock interrupt to allow the OS to take control from the process.
  - Used often in interactive systems as longest wait is guarenteed to be $(\text{processes} - 1) \times$ time slice.

CPU vs I/O Bound

CPU Bound

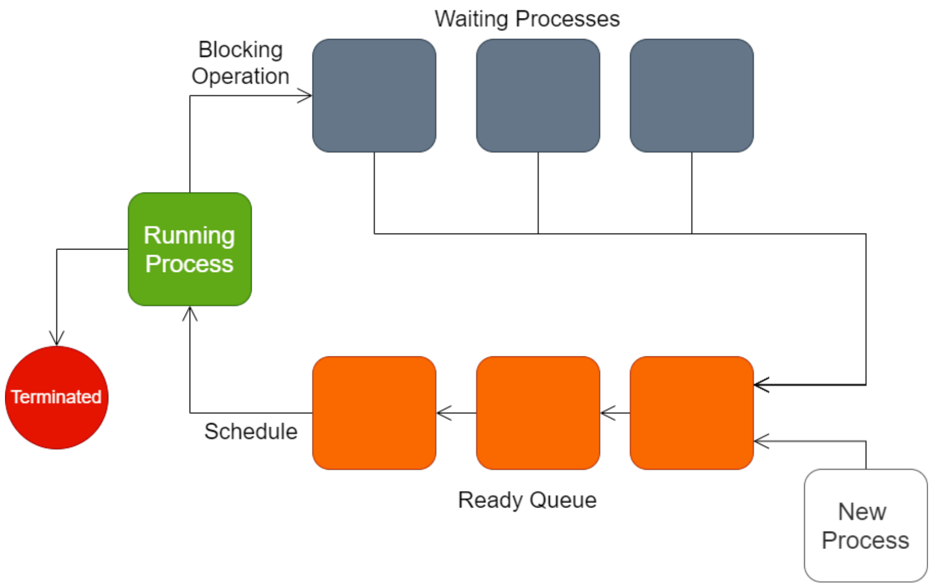
- Spend most of their time using the CPU (e.g data processing).

I/O Bound

- Spend most of their time waiting for I/O.
- Use the CPU briefly before issuing an I/O request.

Schedulers

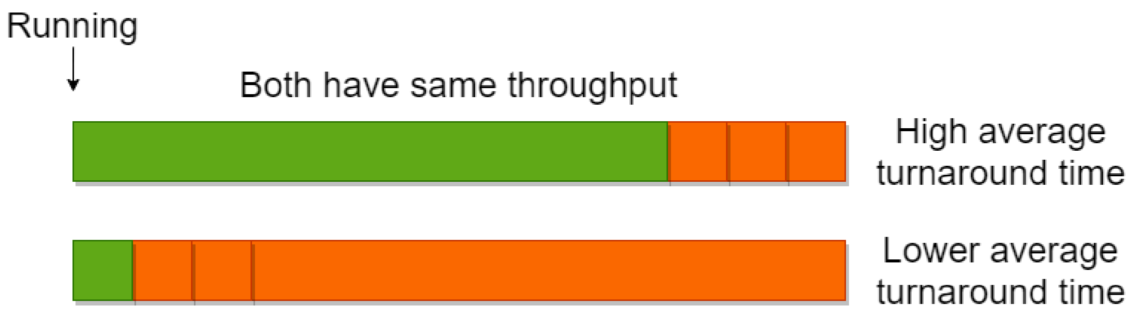
FCFS



Advantages

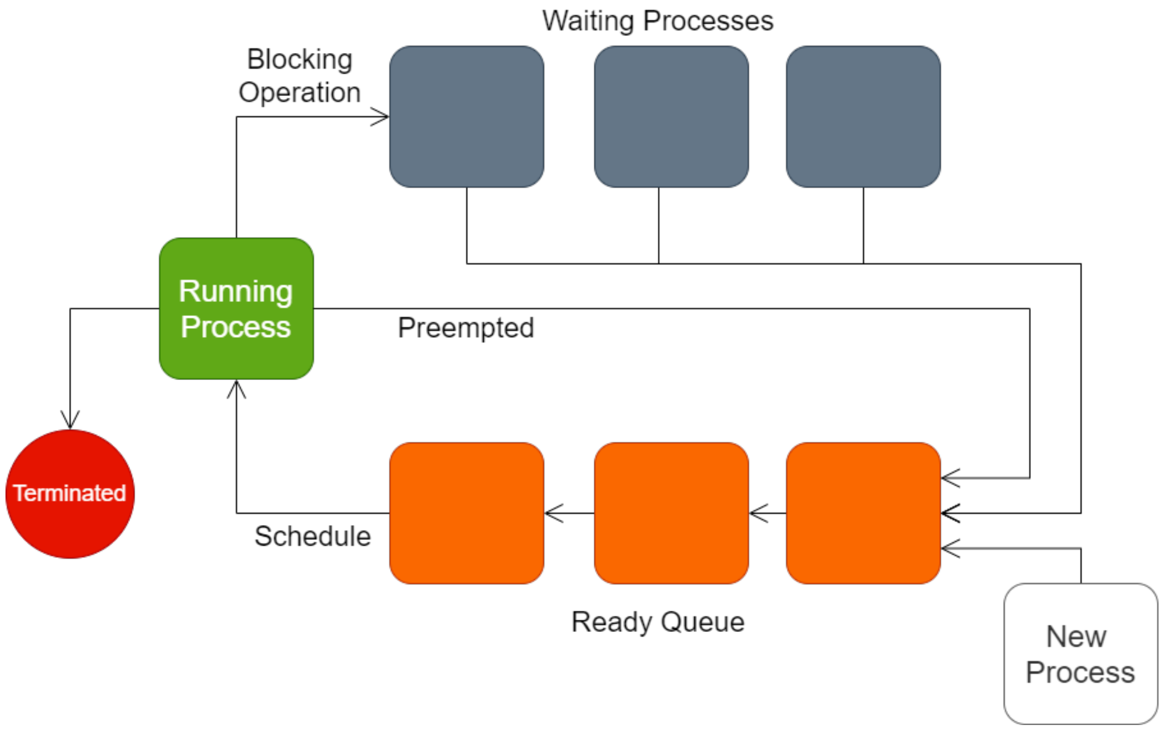
- All processes are eventually scheduled.
- Very simple & easy to implement.
- Very fast (minimal scheduling overhead).

Disadvantages
scheduler is unfair, for example:


Running
Both have same throughput
High average turnaround time
Lower average turnaround time

Round Robin Scheduling

Much like **FCFS**, however a process is preempted and added back to the ready queue when its time quantum expires (time slice).



- Fair (ready jobs get equal CPU time).

- Response time is good for a small number of jobs.
- Turnaround time is low when run times differ (smaller jobs can finish quickly), but poor for similar runtimes (constant interruption & sent to back of ready queue).

$$\text{overhead} = \frac{\text{context switch}}{\text{context switch} + \text{quantum}}$$

$$\text{worst response} = \text{number of processes} \times \text{quantum}$$

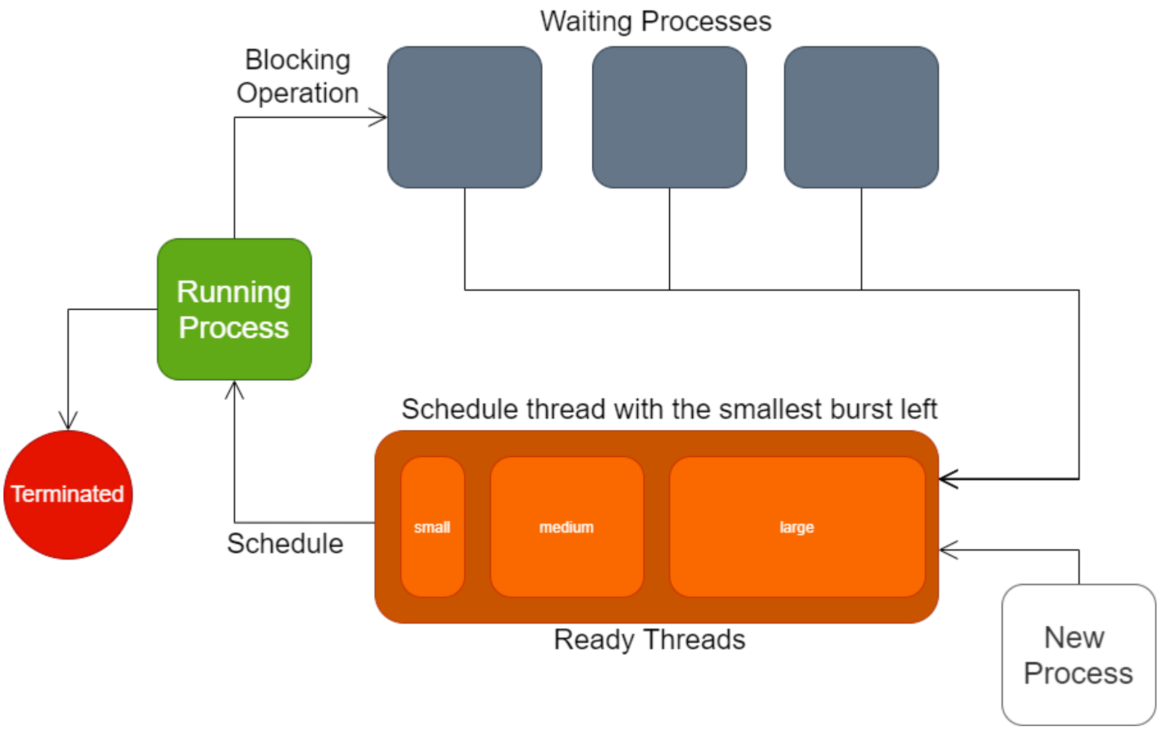
Hence the length of time quantum is a tradeoff between overhead and response time. It should be larger than the context switch time.

Typically from $10ms - 200ms$.

OS	Time Slice
Linux	$100ms$
Windows Client	$20ms$
WIndows Server	$180ms$

Shortest Job First

Schedule the ready thread with the shortest CPU burst remaining given we know the runtimes of each job (either provided, predicted or historical).



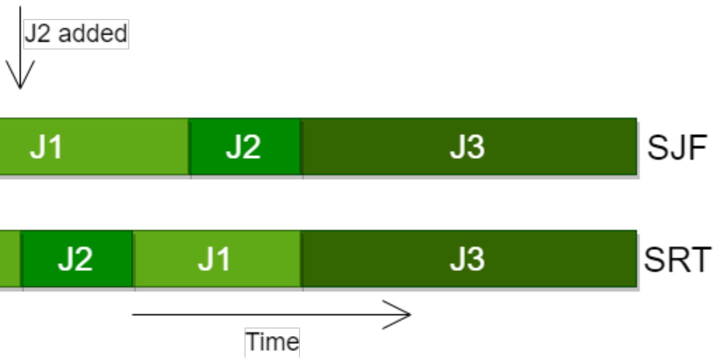
Average turnaround time is optimally low as we run the jobs that are quickest first.

Shortest Remaining Time

Premptive version of **SJF**, chooses the process with the shortest remaining time.

As **SJF** only makes scheduling decisions when a job is complete, when a new job is added while another job is running, it will have to wait. In **SRT** we can reschedule immediately upon receiving the shorter job. For example:

Job 1 & 2 added simultaneously, Job 3 added after $2s$:



One issue with **SRT** is that a long job could be heavily delayed if many short jobs are added.

Konowing Runtimes in Advance

Runtimes are normally not available in advance. However we can make estimates based on previous history (e.g past CPU time).

We can also use user estimates, however we may need to counteract programs cheating (by providing low estimates) by for example penalising processes that overrun their estimated time.

Fair Share Scheduling

On a multi-user system, we need to take into account which processes are being run by which users.

This can be achieved in **round robin** scheduling by having a ready queue per user, and round robining on which ready queue is used.

General Purpose Scheduling

- **Favour short & I/O Bound jobs**
  - Once scheduled they are quickly done with and can have resources freed.
  - I/O bound will likely be blocked and unscheduled quickly.
  - Keep reposnse times low.
- **Quickly Determine Job Nature**
  By quickly adapting to determine which jobs are **CPU** and **I/O** bound we can schedule more effectively.

Priority Scheduling

- Jobs run based on a priority. Always running the job with the highest priority.
- Priorities can be externally defined (user) or based on process-specific metrics (e.g expected CPU burst, previous CPU time, process parent).
- Can be static (unchanging) or dynamic (change during execution - e.g process aging).

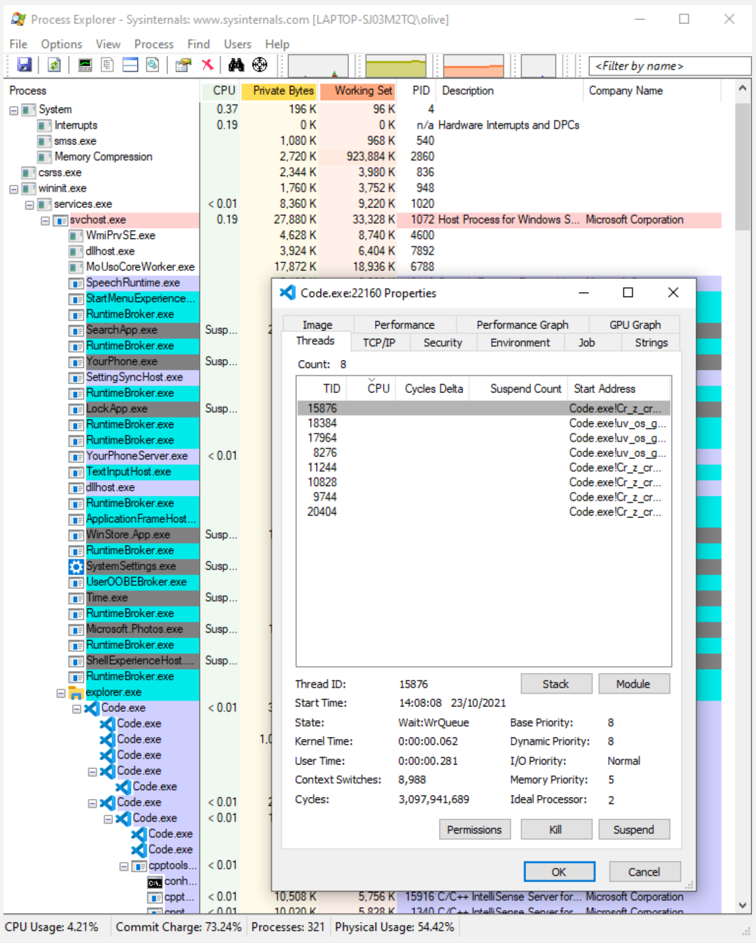
Note: Shortest remaining time is a form of priority scheduling.

These are also used in:

- Windows Vista, 7
- Mac OS X
- Linux 2.6 - 2.6.23
- Pintos!

Process explorer

If you are on windows, you can use the process explorer to see the priority system in action on any running process & its threads.
(https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer)


Process Explorer - Sysinternals: www.sysinternals.com [LAPTOP-SJ03M2TQ\olive]
File Options View Process Find Users Help
<Filter by name>
Code.exe:22160 Properties
Image Performance Performance Graph GPU Graph
Threads TCP/IP Security Environment Job Strings
Stack Module
Permissions Kill Suspend
OK Cancel

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
System	0.37	196 K	96 K	4		
Interrupts	0.19	0 K	0 K	n/a	Hardware Interrupts and DPCs	
smss.exe		1,080 K	968 K	540		
Memory Compression		2,720 K	923,884 K	2860		
csrss.exe		2,344 K	3,980 K	836		
wininit.exe		1,760 K	3,752 K	948		
services.exe	< 0.01	8,360 K	9,220 K	1020		
svchost.exe	0.19	27,880 K	33,328 K	1072	Host Process for Windows S...	Microsoft Corporation
WmiPrvSE.exe		4,628 K	8,740 K	4600		
dllhost.exe		3,924 K	6,404 K	7892		
MoUsoCoreWorker.exe		17,872 K	18,936 K	6788		
SpeechRuntime.exe						
StartMenuExperience...						
RuntimeBroker.exe						
SearchApp.exe	Susp...					
RuntimeBroker.exe						
YourPhone.exe	Susp...					
SettingSyncHost.exe						
RuntimeBroker.exe						
LockApp.exe	Susp...					
RuntimeBroker.exe						
RuntimeBroker.exe						
YourPhoneServer.exe	< 0.01					
TextInputHost.exe						
dllhost.exe						
RuntimeBroker.exe						
ApplicationFrameHost...						
WinStore.App.exe	Susp...					
RuntimeBroker.exe						
SystemSettings.exe	Susp...					
UserOOBEBroker.exe						
Time.exe	Susp...					
RuntimeBroker.exe						
Microsoft.Photos.exe	Susp...					
RuntimeBroker.exe						
ShellExperienceHost....	Susp...					
RuntimeBroker.exe						
explorer.exe						
Code.exe	< 0.01					
Code.exe						
Code.exe						
Code.exe						
Code.exe						
Code.exe						
Code.exe	< 0.01					
Code.exe	< 0.01					
Code.exe						
Code.exe						
cpptools...	< 0.01					
conh...						
cppt...	< 0.01	10,508 K	5,756 K	15916	C/C++ IntelliSense Server for...	Microsoft Corporation
cppt...	< 0.01	10,020 K	5,828 K	1340	C/C++ IntelliSense Server for...	Microsoft Corporation

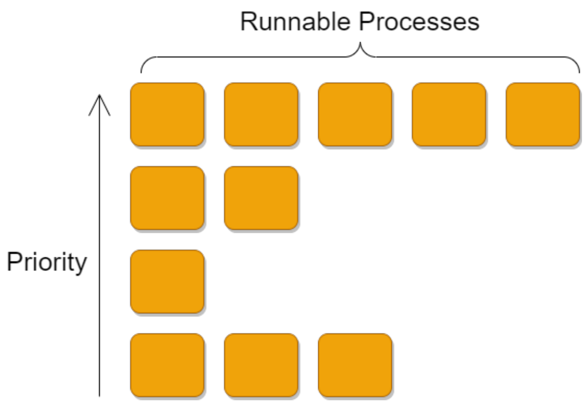
Count: 8

TID	CPU	Cycles Delta	Suspend Count	Start Address
15876				Code.exe!Cr_z_cr...
18384				Code.exe!uv_os_g...
17964				Code.exe!uv_os_g...
8276				Code.exe!uv_os_g...
11244				Code.exe!Cr_z_cr...
10828				Code.exe!Cr_z_cr...
9744				Code.exe!Cr_z_cr...
20404				Code.exe!Cr_z_cr...

Thread ID:	15876		
Start Time:	14:08:08 23/10/2021		
State:	Wait:WrQueue	Base Priority:	8
Kernel Time:	0:00:00.062	Dynamic Priority:	8
User Time:	0:00:00.281	I/O Priority:	Normal
Context Switches:	8,988	Memory Priority:	5
Cycles:	3,097,941,689	Ideal Processor:	2

CPU Usage: 4.21% | Commit Charge: 73.24% | Processes: 321 | Physical Usage: 54.42%

Multilevel Feedback queues



Each priority level has its own set of runnable threads, which can be scheduled (even by different algorithms).

We must be careful not to starve lower priority jobs. This can be done with the feedback mechanism that age a thread as it waits, and assign higher priority to threads that are waiting a long time.

We can also give I/O bound operations higher priority to ensure they are able to run as quickly they are done waiting (response time is important).

However there are some disadvantages:

- **Inflexible**
  Applications have no control over their priority, and priorities make no guarentees (e.g high priority does not matter, if all processes are running on high priority)
- **Cannot React Quickly**
  Often Needs a warm-up period (as initially priorities of processes may not relfect what bounds them/their behaviour). This also becomes a rpoblem for realtime systems that must react quickly.
- **Cheating**
  e.g Adding meaningless I/O to boost priority.

  As prity is based off 'feedback', priority can be manipulated.

Lottery Scheduling

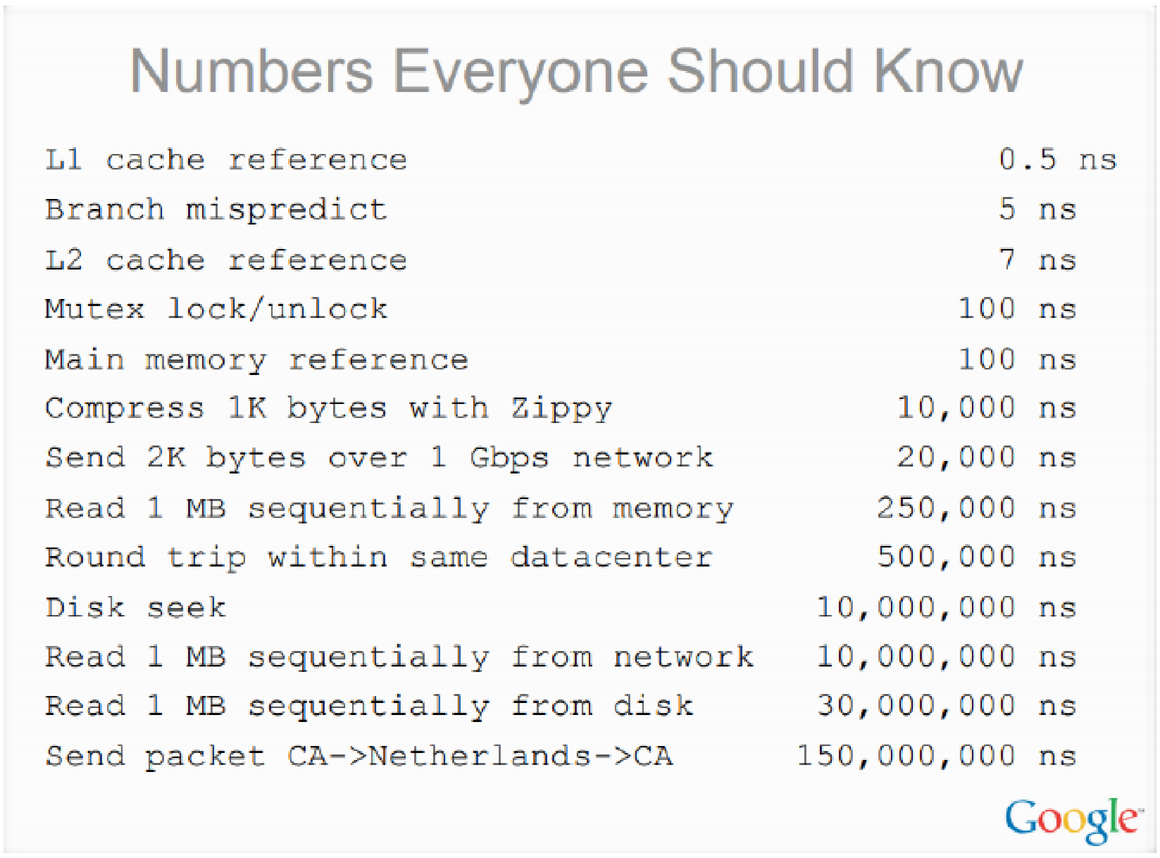
Next job to schedule is based on probability. At each scheduling decision a random process is chosen, with probability biased towards threads we that are higher priority.

- **tickets Meaningful**
  If a process has 20% of the tickets, it gets on average 20% of the time.
- **Highly responsive**
  We can provide a job more tickets to give it a higher chance of being run at the next scheduling decision.

- **No Starvation**
  As all threads have a chance of being scheduled at every decision.
- **Donation**
  Jobs can exchange tickets, this allows for easy priority donation, and allows cooperating jobs to achieve goals.
- **Adding a Job affects existing Jobs**
- **Unpredictable** A process can be *lucky* or *unlucky*, and this can impact response times (e.g interactive process is very unlucky, causes an issue).

Summary

- **Schedulers Balance Conflicting Goals** Fairness, enforce policy, maximise resource utilisation, minimise overhead.
- **Different Scheduling Algorithms are Appropriate in Different Contexts** Batch vs interactive vs real-time.
- **Well Studied Algorithms** FCFS, RR, SJF, SRT, MLFQs, Lottery



Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

Additions

A little python program for doing **RR**, **fcfs** calculations:

```python
from typing import List, Tuple

def fcfs(ps : List[Tuple[str,int]]):
    time = 0
    turnaround = 0
    for (p, t) in ps:
        time += t
        turnaround += t
        print("{} completed after {}".format(p, time))
    print("Avg turnaround: {}\nThroughput: {}", float(turnaround) / len(ps), float(
        ↪ time) / len(ps))

def rr(ps : List[Tuple[str,int]]):
    turnaround = 0
    ps = sorted(ps, key=lambda pt : pt[1])
    n = len(ps)
    lastt = ps[0][1]
    time = lastt * n
    print("{} completed after {}".format(ps[0][0], time))
    for (p, t) in ps[1:]:
        n -= 1
        time += (t - lastt) * n
        turnaround += time
        print("{} completed after {}".format(p, time))
        lastt = t
    print("Avg turnaround: {}\nThroughput: {}", float(turnaround) / len(ps), float(
        ↪ time) / len(ps))
```