Computer Systems Tutorial

December 11th 2023

1 Week 7

Exercise 1. Determine the complexity of the following pieces of code in terms of big-$\mathcal{O}$ notation. For this, derive an expression to denote the number of steps which are needed to compute the pieces and identify the dominant term.

```
if (n == 1) {
    return;
}

for (int i = 0; i < n - 1; i++) {

    for (int j = 0; j < n - i - 1; j++) {

        if (arr[j] > arr[j + 1]) {
        // assume it takes
        // constant number of steps
            swap(arr[j], arr[j + 1]);
        }
}
```

(a)

```
for (int i = 0; i < n - 1; i++) {

    for (int j = 0; j < n - i - 1; j++) {

        if (arr[j] > arr[j + 1]) {
        // assume it takes
        // constant number of steps
            swap(arr[j], arr[j + 1]);
        }
    }
}

for (int i = 0; i < n; i++) {

    for (int j = 1; j < n; j = j*2) {

        k = k + j;

    }
}
```

(b)

```
for (int i = 0; i < n; i++) {

// pow(x,y) raises x to the power of y
    for (int j = 0; j < pow(2,i); j++) {

        k = k * j;

}
```

(c)

```
int fibonacci(n) {

    if (n <= 2) return 1;

    return fibonacci(n-1) + fibonacci(n-2);
}
```

(d)

Solution

a We need to compute the number of steps the program piece makes for the input depending on n. This will give us a formula that expresses the number of steps and by identifying the dominant term in the formula we will come up with the complexity in terms of big-$\mathcal{O}$. As we are interested in the worst-case complexity, we can skip lines 1-3 (or assume that only 1 step is needed to compute the *if* statement). Similarly, we can assume that only 1 step is needed to compute the expression in lines 9-13.

The number of steps is determined by the number of iterations of the innermost loop and by how many times the whole loop is invoked. We will proceed by looking at how many times the inner loop executes for each value of i (because the number of iterations is bounded by $n - i - 1$). And, as we assumed above the loop performs only one step at each iteration.

- For $i = 0$ the possible values of j are $[0, \ldots, n-1)$ (the right number is not included as we have a strict $<$). We got $n - 1$ by plugging $i = 0$ into $n - i - 1 = n - 0 - 1 = n - 1$. Hence, we have $n - 1$ iterations (for j from 0 to $n - 2$ because $n - 1$ is not included) of the inner loop for $i = 0$.
- For $i = 1$ the possible values of j are $[0, \ldots, (n-1-1)) = [0, \ldots, (n-2))$ and hence $n-2$ steps.
- Similarly, for $i = 2$ we have $n - 3$ steps and so on.
- We end when $i = n-2$ and then the possible values of j are $[0, \ldots, (n-(n-2)-1)) = [0, \ldots, 1)$ and we have only one step.

So far we found the number of steps for each value of i and to find the total number of steps we need to sum them for all i. We have a sum $1 + 2 + \ldots n - 2 + n - 1$ which is an arithmetic progression[1]. The sum is equal to $\frac{1+(n-1)}{2} \times (n-1) = \frac{n}{2} \times (n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$. This formula represents the total number of steps the two loops take. We can add an extra 1 to this formula to account for the first 3 lines and end up with $\frac{1}{2}n^2 - \frac{1}{2}n + 1$. The dominant term here is n^2 and hence the complexity is $\mathcal{O}(n^2)$.

b Recall that the number of steps for lines 1-11 is $\frac{1}{2}n^2 - \frac{1}{2}n$ and we only need to find the number of steps for lines 13-20. The number of iterations in the inner loops does not depend on i so the total number of steps would be the number of steps the inner loop does times the number of steps the inner loop is executed which is given by `i = 0; i < n`, i.e. it is executed n times. The inner loop does $log_2(n)$ iterations because j goes from 1 to $n - 1$ (actually we need to round this logarithm up, but it does not matter that much) and is doubled every iteration, and the number of times 2 should be doubled to reach n is exactly $log_2(n)$ as $2^{log_2(n)} = n$ (and since we start from 1 and reach $n - 1$ the number of iterations is the same). We can consider a couple of examples.

- $n = 4$ then $log_2(4) = 2$ and possible values of j are $1, 2$, i.e. exactly 2 of them.
- $n = 16$ then $log_2(16) = 4$ and possible values of j are $1, 2, 4, 8$, i.e. exactly 4 of them.

Hence, we have $log_2(n)$ iterations for each value of i of which there are n. And in total it gives us $nlog_2(n)$ steps. Recalling the number of steps for lines 1 - 11 we get the formula $\frac{1}{2}n^2 - \frac{1}{2}n + nlog_2(n)$ and n^2 is the dominant term (since $n^2 > nlog_2(n)$ as $log_2(n) < n$). Ultimately, the complexity is $\mathcal{O}(n^2)$.

[1]`https://en.wikipedia.org/wiki/Arithmetic_progression`

c To find the number of steps for this piece we need to essentially follow the same routine as in (a). That is, for each value of i we need to find the number of iterations the inner loop does.

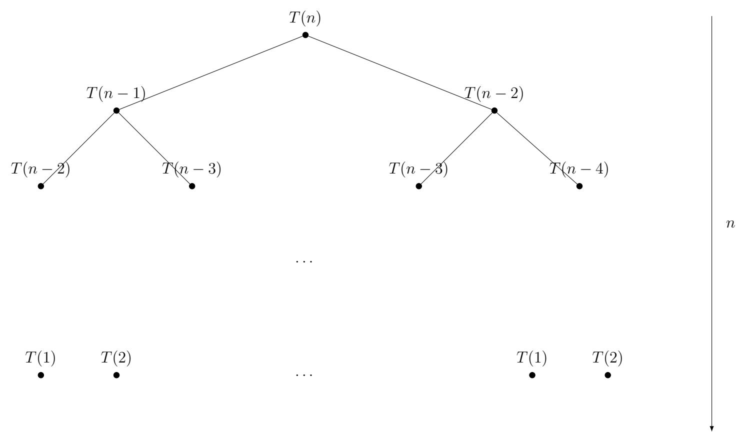
- $i = 0$. Then, j ranges in $[0, \ldots, 2^0) = [0, 1)$, i.e. the loop does only 1 iteration and 1 step (we assume the line 6 takes only 1 step as constants do not affect the overall complexity).
- $i = 1$. Then j ranges in $[0, \ldots, 2^1) = [0, 2)$, i.e. 2 steps.
- $i = 2$. Then j ranges in $[0, \ldots, 2^2) = [0, 4)$, i.e. 4 or 2^2 steps.
- $i = 3$. Then j ranges in $[0, \ldots, 2^3) = [0, 8)$, i.e. 8 or 2^3 steps.
- $i = n - 1$. Then j ranges in $[0, \ldots, 2^(n - 1))$, i.e. $2^(n - 1)$ steps.

Essentially, for each i the inner loop does 2^i steps. We need to find the sum of 2^i for all i from 0 to $n - 1$ to get the total number of steps both loops do. This is $2^0 + 2^1 + 2^2 + \ldots + 2^{n-1}$. The sum is equal to $2^n - 1$ because in binary the sum can be represented as n ones: $\underbrace{11\ldots11}_{n}$ (recall that number 7 in binary is 111 because $7 = 2^0 + 2^1 + 2^2 = 1 + 2 + 4$). Finally $\underbrace{11\ldots11}_{n} = 2^n - 1$, once again, $7 = 2^3 - 1 = 8 - 1$. Alternatively, one could use the formula for geometric series. Thus, the sum is equal to $2^n - 1$ where the dominant term is 2^n and the complexity is $\mathcal{O}(2^n)$. **There was a discussion at the tutorial that we can over approximate the number of iterations the inner loop does with 2^{n-1} for each i (as we saw above the number of iterations for each i is indeed always less than or equal to 2^{n-1}). And then argue that the total number of steps is $n2^{n-1}$. While this might be a reasonable thing to do under some circumstances, we are interested in the tightest upper bound and $2^n - 1$ is a more tight upper bound than $n2^{n-1}$ because for sufficiently large n, $n2^{n-1} > 2^n - 1$. You can see an example below in Figure 2.**

d This example is different from the others because it uses recursion rather than loops. So we need to somehow express the complexity recursively. Let's denote the number of steps the program does for n as input as $T(n)$. Then,

$$T(n) = T(n - 1) + T(n - 2) + 1$$

Where $+1$ is because we need to perform the addition and $T(n-1)+T(n-2)$ is because to compute $T(n)$ we first need to perform some steps in computing `fibonacci(n-1)` and `fibonacci(n-2)`. We also can note that $T(1) = T(2) = 1$ because we just return 1 in these cases. To find the number of steps for $T(n)$ we can unfold the recurrence. Unfolding the recurrence gives us the following tree (it essentially represents the tree of function calls).



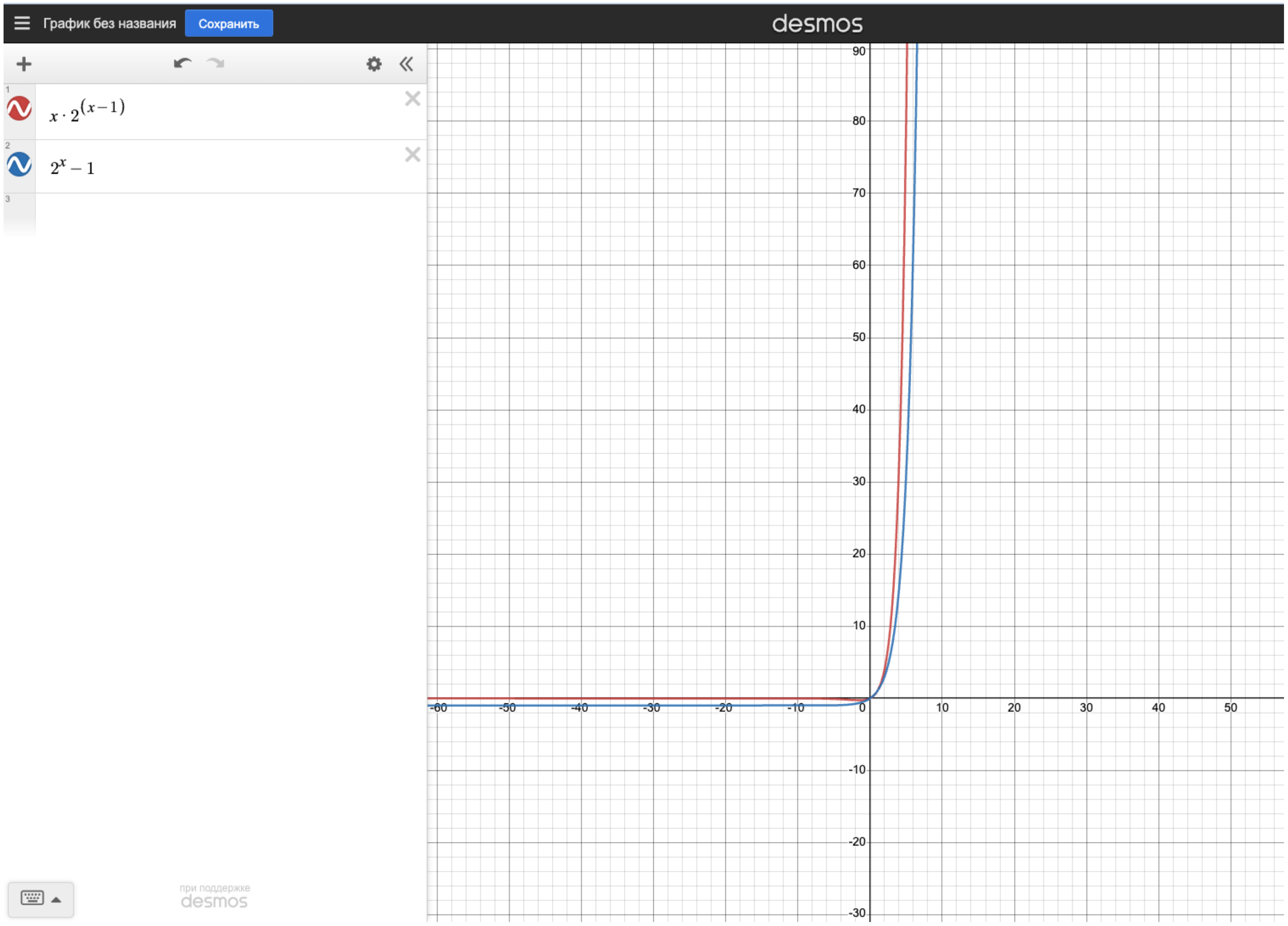


Figure 2: $n2^{n-1} > 2^n - 1$

The depth of the tree is n (you can check that by following the leftmost path from $T(n) to T(n)$) and at each level $i = [0, \dots, n-1]$ we have 2^i nodes. Each of the leaf nodes takes 1 step to compute. All the others take the number of steps their children do and another 1 extra step. So to calculate the total number of steps we need to find the number of nodes in this tree. Because at each level we have 2^i nodes and i goes from $0 \to n-1$ we have $2^n - 1$ steps and the complexity is $\mathcal{O}(2^n)$.

Exercise 2. Using Shunting yard algorithm build an RPN representation of the following arithmetic expression.

$$3 + 4 \times (2 - 1 \times 2)$$

Then, evaluate the expression using stack.

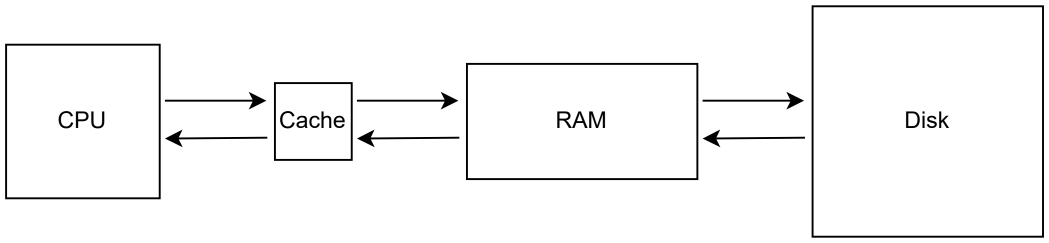
Solution The algorithm can be found here: `https://en.wikipedia.org/wiki/Shunting_yard_algorithm` and some other examples of its usage. I will present the trace of the algorithm in Table 1 for the example above.

You can try to evaluate the expression in the last row using stack to see if it evaluates to 3.

Step	Output	Stack	Input	Comment
1	Empty	Empty	$3 + 4 \times (2 - 1 \times 2)$	Initially both the stack and the output are empty
2	3	Empty	$+4 \times (2 - 1 \times 2)$	A number from the input goes right to the output
3	3	$+$	$4 \times (2 - 1 \times 2)$	Operand goes to the stack (the top of the stack is on the right)
4	3 4	$+$	$\times(2 - 1 \times 2)$	
5	3 4	$+ \times$	$(2 - 1 \times 2)$	
6	3 4	$+ \times ($	$2 - 1 \times 2)$	Note that left parenthesis also goes onto the stack
7	3 4 2	$+ \times ($	$-1 \times 2)$	
8	3 4 2	$+ \times (-$	$1 \times 2)$	
9	3 4 2 1	$+ \times (-$	$\times 2)$	
10	3 4 2 1	$+ \times (- \times$	$2)$	
11	3 4 2 1 2	$+ \times (- \times$	$)$	
12	$3\ 4\ 2\ 1\ 2 \times -$	$+ \times$	Empty	When a right parenthesis is encountered, the stack is popped to the output until the matching left parenthesis is found, which is discarded from the stack (not put to the output)
13	$3\ 4\ 2\ 1\ 2 \times - \times +$	Empty	Empty	The rest of the stack is popped to the output

Table 1: Run of Shunting yard algorithm

Exercise 3. A computer system has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20ns are required to access it. If it is in the main memory but not in cache, 60ns are needed to load it into cache (this includes the time to originally check the cache), and then the reference is started again. If the word is not in main memory, 12ms are required to fetch the word from disk, followed by 60ns to copy it to the cache, and then the reference is started again. The cache hit-ratio is 0.9 and the main memory hit-ratio is 0.6. What is the average time in ns required to access a referenced word on this system?



Solution. To find the average time we need to multiply the time to take some data from the particular memory by the probability that this particular memory is accessed. This is essentially the expected value of the access time: `https://en.wikipedia.org/wiki/Expected_value`. You can think of it as a dice with 6 sides. The expected value of a single dice roll is $1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \ldots + 6 \cdot \frac{1}{6} = 3.5$ i.e. the sum of values on the dice side multiplied by the probability to roll this value. The probabilities and times can be found in Table 4. The average is then $20 \cdot 0.9 + 80 \cdot 0.06 + 12000080 \cdot 0.04 = 480026$ ns.

Exercise 4. Predict the Output of the following program.

```c
int main(){
    int x = 1;
    if (fork() == 0) {
        printf("Child has: %d", ++x);
    } else {
        printf("Parent has: %d", --x);
    }
    wait(NULL);
    return 0;
}
```

Solution. `fork()` creates a child process that has its own copy of variable `x` and each process can modify only its copy. Thus, the possible outputs are

```
Child has: 2 //++1 = 2
Parent has: 0 // --1 = 0
```

and

```
Parent has: 0 //--1 = 0
Child has: 2 // ++1 = 2
```

Because processes can execute in any order, for example, the parent process time quantum could expire at line 3 and the child process will then print first.

`wait(NULL)` is needed for the parent process to wait until all its children exit. We are checking the output of `fork()` to check in which process we are at the moment: for the parent process the function returns the child's process id and for the child, it returns 0.

Location	Probability	Time to access in ns	Comment
Cache	0.9	20ns	
RAM	0.6×0.1	20ns + 60ns	We access RAM only if the data is not in Cache which has probability 1 - 0.9. It takes 60ns to fetch from RAM to Cache and another 20ns to fetch from Cache
Disk	0.4×0.1	20ns + 60ns + 12000000ns	We access Disk only if the data is not in Cache and not in RAM which has probability of 1 - 0.9 and 1 - 0.6 respectively. Also note that 1ms = 1000000ns.

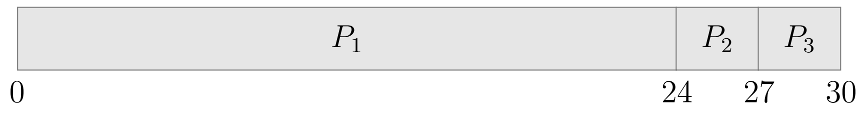
Table 2: Probabilities and times

2 Week 8

Preliminaries

Preemtive execution . Preemption is the act of temporarily interrupting an executing task, with the intention of resuming it at a later time. This interrupt is done by an external scheduler, for example, by setting a timer interrupt, or by using priorities, with no assistance or cooperation from the task.

Gantt chart A Gantt chart is a type of bar chart that illustrates a project schedule. This chart lists the tasks to be performed on the vertical axis, and time intervals on the horizontal axis. The width of the horizontal bars in the graph shows the duration of each activity. An example can be seen below.



Tasks to be performed are processes P_1, P_2, P_3, start and end time of each task are depicted on the horizontal axis. This chart shows the scheduling of these processes on a single CPU. When the number of CPUs exceeds 1, parallel processes are depicted on separate vertical lines.

Burst time A typical program consists of a sequence of interleaving I/O and CPU actions. To perform a CPU action a program needs to be allocated a CPU and the burst time refers to CPU time the current CPU action needs to execute before the next I/O action starts. In the exercises below we ignore I/O actions and hence burst time simply refers to the CPU time the process needs to complete its execution.

FCFS (first come first served) This is a simple scheduling strategy where processes are scheduled according to their position in the queue.

SJF (shortest job first) A process with the shortest burst time is scheduled first. If a program consists of several CPU actions and I/O actions the next burst time can be estimated by using the records of burst times for previous CPU actions.

Nonpreemtive priority A process with the highest priority is executed first and if another process with a higher priority arrives in the ready queue during the execution of the current process the current process keeps executing.

Preemtive priority A process with the highest priority is executed first and if another process with a higher priority arrives in the ready queue during the execution of the current process is stopped and the newly arrived high-priority process is executed next. The stopped process goes to the end of the queue.

Round Robin (RR) Processes are executed by utilising a time quantum. Once the quantum elapses, the process is stopped, and brought to the end of the queue and the next process in the queue is scheduled for the same time quantum.

Turnaround time (t/o time) The total amount of time to execute one process to its completion from the moment it first gets to the ready queue.

Waiting time The total amount of time a process has been waiting in the Ready queue. This is essentially t/o time − burst time.

CPU utilisation The percentage of time the CPU had work.

Throughput The number of processes that complete their execution per time unit.

Exercises

Exercise 1 A CPU scheduling algorithm determines an order for the execution of its scheduled processes. Given 5 processes to be scheduled on one processor, how many different schedules are possible? Assume that scheduling is nonpreemptive and no IO is made. How many different schedules are possible for n processes?

Solution We essentially need to find all the possible permutations of 5 processes (because once the process is scheduled it does not release the CPU until it finishes as we have no IO and scheduling is nonpreemptive). For the first process to schedule we have 5 options for the next one we have 4 (because we scheduled one already) and so on until we end up with only one process. This gives us $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ combinations to schedule 5 processes. For n processes the number of combinations is 5!. From a complexity point of view, we treat problems with exponential complexity to be non-tractable. But for scheduling the complexity is even worse as $\mathcal{O}(2^n) \subset \mathcal{O}(n!)$ meaning that problems with factorial complexity are harder than exponential ones. This means we need to come up with different heuristics and that a particular strategy works well only in some specific cases because it is not plausible to explore all the possible schedulings and pick the best one.

Exercise 2 Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to arrive in the order listed all at time 0.

a Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority(a larger priority number implies a higher priority), and RR(quantum=2ms).

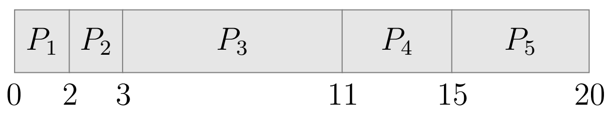
b What is the turnaround time of each process for each of the scheduling algorithms in part a?

c What is the waiting time of each process for each of these scheduling algorithms?

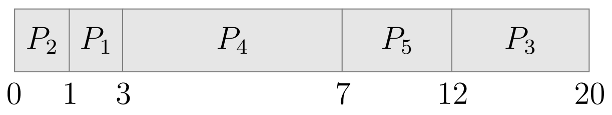
d Which of the algorithms results in the minimum average waiting time (over all processes)?

Solution

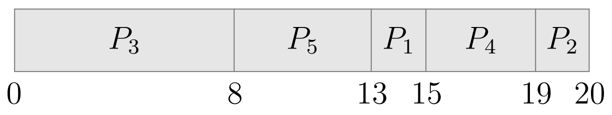
a *FCFS*



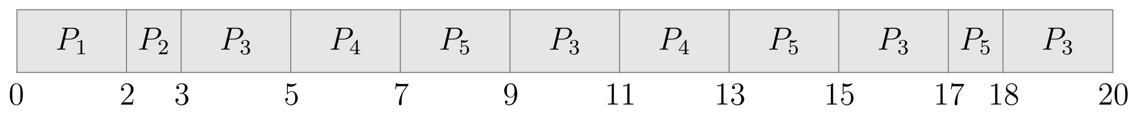
SJF



Nonpreemptive priority. Note that tie between P_1 and P_4 (they have the same priority) is broken according to FCFS.



RR



b Turnaround time

	FCFS	SJF	Priority	RR
P_1	2	3	15	2
P_2	3	1	20	3
P_3	11	20	8	20
P_4	15	7	19	13
P_5	20	12	13	18

Table 3: Turnaround time

c Waiting time

	FCFS	SJF	Priority	RR
P_1	0	1	13	0
P_2	2	0	19	2
P_3	3	12	0	12
P_4	11	3	15	9
P_5	15	7	8	13
Average	6.2	4.6	11	7.2

Table 4: Waiting time

d SJF has the smallest waiting time.

Exercise 3 The following processes are being scheduled using a preemptive, round-robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed above, the system also has an **idle task** (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue (first executes the process with the highest priority and processes with the same priority are scheduled by RR).

Process	Priority	Burst	Arrival
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

a Show the scheduling order of the processes using a Gantt chart.

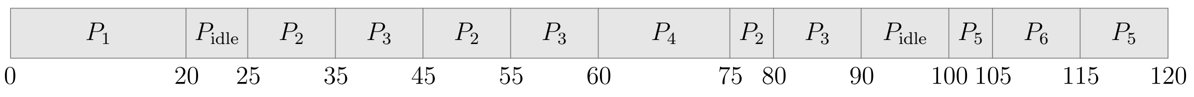
b What is the turnaround time for each process?

c What is the waiting time for each process?

d What is the CPU utilization rate (percentage of time the CPU is busy)?

Solution

a *FCFS*



b $P_1 : 20 - 0 = 20,\ P_2 : 80 - 25 = 55,\ P_3 : 90 - 30 = 60,\ P_4 : 75 - 60 = 15,\ P_5 : 120 - 100 = 20,\ P_6 : 115 - 105 = 10$

c $P_1 : 0,\ P_2 : 40,\ P_3 : 35,\ P_4 : 0,\ P_5 : 10,\ P_6 : 0$

d CPU utilisation is $\frac{105}{120} = 87.3\%$

Exercise 4 You and your friend John are working on a Parallel Programming project, which is an application that does not use IO functions. You have been told by your project manager that your software should have at least 6 times speedup as compared to its sequential implementation. You have also been told that your application will run on a system with 16 CPUs. Estimate the minimum percentage of the code that needs to be parallelized in your application, to meet the above performance requirement.

Solution Recall Amdahl's law

$$s = \frac{1}{(1-f) + \frac{f}{N}}$$

where f is a portion of a program that can be parallelized, N is the number of processes, and s is a theoretically achievable speed-up. Given $N = 16$ and $s \geq 6$ we have an inequality

$$\frac{1}{(1-f) + \frac{f}{16}} \geq 6$$

Solving it

$$1 \geq 6((1-f) + \frac{f}{16})$$

$$1 \geq 6 - 6f + \frac{6f}{16}$$

$$6f - \frac{6f}{16} \geq 5$$

$$5\frac{11}{16}f \geq 5$$

$$f \geq \frac{80}{91}$$

This requires f to be at least 0.87.

Exercise 5 Explain the difference between kernel threads and user threads.

Solution Kernel threads and user threads are two types of threads in the context of multitasking operating systems. They differ in terms of where they are managed and how they interact with the operating system.

Kernel Threads:

- Managed by the Kernel: Kernel threads are created and managed by the operating system's kernel. The kernel has full control over these threads.
- Operating System Support: Kernel threads are supported and scheduled by the operating system. The kernel is responsible for thread creation, scheduling, and termination.
- Concurrency: Since the kernel has direct control, kernel threads can run in parallel on multiple processors or cores, providing true parallelism.

User Threads:

- Managed by User-level Libraries: User threads are created and managed by user-level thread libraries or runtime environments. Examples include POSIX threads (pthread) and Win32 threads.
- Operating System Agnostic: The operating system may not be aware of user threads, and it may treat the entire process as a single entity. This means that user threads may not be scheduled in parallel on multiple processors.

- Concurrency: User threads rely on a single kernel thread for execution. If one user thread is blocked (e.g., waiting for I/O), it may block the entire process, including all user threads.

User threads typically use non-blocking calls to perform I/O in order to not block the process if the resource is unavailable. Please, see the following piece of Python code that uses user threads, and try to predict the output before running the program.

`https://replit.com/@andreybolkonskiy/ShySpicyKey#main.py`

3 Week 9

Preliminaries

There were some definitions of bandwidth and delay in the lecture but we will have to be a bit more specific in the exercises.

Bandwidth (digital) It is the data rate of a channel, a quantity measured in bits/sec, i.e. tells how many bits can arrive at a unit of time. That data rate is the end result of using the **analog** bandwidth (measured in Hertz) of a physical channel for digital transmission. Two ways that a digital bandwidth can be increased is either by increasing the analog bandwidth or by sending more than one bit "at once". The formula that relates analog and digital bandwidths is the following (assuming that we are in a noiseless environment)

$$B_{\text{digital}} = 2B_{\text{analog}} log_2 V \text{ bits/sec}$$

where V is the number of discrete levels in the signal (roughly how many bits arrive at once).

Propagation speed $s_{\textbf{prop}}$ This is the speed a signal can travel through a medium, e.g. a wire (Ethernet) or air (WiFi).

Propagation delay This is hence the distance divided by propagation speed. Roughly, it tells how soon a single bit will arrive at the destination (or more than a single bit depending on the encoding).

Bandwidth delay (Transmission delay) It is the time needed for a sender to get the packet onto the wire (note that it is not the time when the receiver receives the packet). This is simply the packet size divided by the bandwidth.

An important difference between bandwidth delay and propagation delay is that bandwidth delay is proportional to the amount of data sent while propagation delay is not. If we send two packets back-to-back, then the bandwidth delay is doubled but the propagation delay counts only once.

Total one-way transmission delay This is hence propagation delay plus bandwidth delay.

There are also queuing delay and **processing** delay that mean the obvious things.

Exercises

Exercise 1 How many edges are in a complete (each node is connected to any other node) graph with 5 nodes? In a graph with n nodes? Assume that only one edge between any two nodes is allowed.

Solution Let's enumerate nodes as a, b, c, d, e. Select node a, we need to connect it to 4 other nodes (no need to connect to itself as we do not allow loops). Then take node b, we do not need to connect it to a as they are already connected, so we have only 3 nodes to connect b to. Then we take c and similarly have only 2 nodes to connect c to. Then we take d and connect it to e. When we take e it is already connected to every node. So, in total, we have $4 + 3 + 2 + 1 = 10$ nodes.

Now consider n nodes. The first one needs to be connected to $n - 1$ nodes, the next one to $n - 2$ nodes and so on until we get to the penultimate node which needs to be connected only with one other node.

This gives us $(n-1)+(n-2)+\ldots+2+1$ nodes. Computing the sum (by using the formula for arithmetic series) we get $\frac{1+(n-1)}{2}(n-1)$ or $\frac{n}{2}(n-1) = \frac{n^2}{2} - \frac{n}{2}$.

Exercise 2

a Consider the following configuration

$$A \longrightarrow B$$

- Propagation delay is 40 µsec
- Bandwidth is 1 byte/µsec (1 mB/sec, 8 Mbit/sec)
- Packet size is 200 bytes (200 µsec bandwidth delay)

What is the total transmission delay to send one packet from A to B?

b Consider the following configuration

$$A \longrightarrow C \longrightarrow B$$

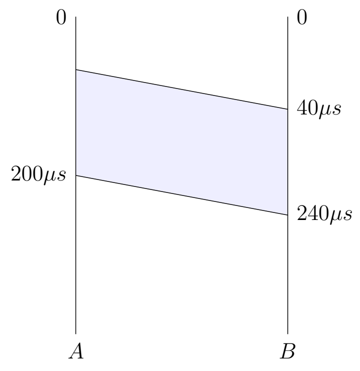
- Propagation delay is 40 µsec (for each wire)
- Bandwidth is 1 byte/µsec (1 mB/sec, 8 Mbit/sec)
- Packet size is 200 bytes (200 µsec bandwidth delay)

What is the total transmission delay to send one packet from A to B?

c The same as above, but with data sent as two 100-byte packets.

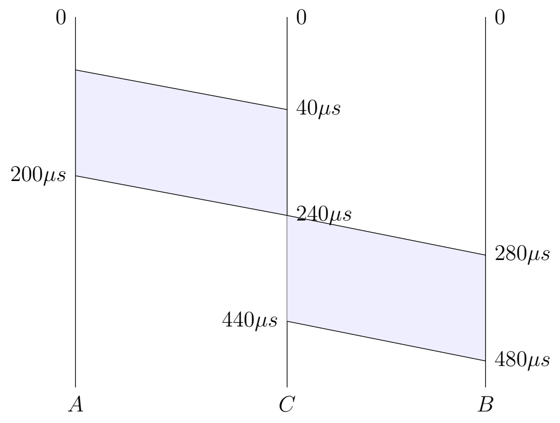
Solution

a Consider the diagram below.



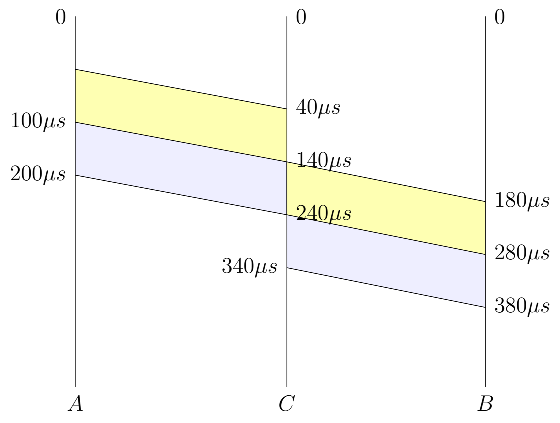
The first bit of the packet needs $40\mu s$ to reach B from A and as the bandwidth is 1 byte/μs we need $200\mu s$ to put the whole packet onto the network wire and the last bit will arrive at time $240\mu s$.

b Consider the diagram below.



The packet arrives from A to C like it did before in part (a). **Because we have store-forward packet-switching, C should wait for the whole packet to arrive before it can send it to B. Once C gets the whole packet, the first bit takes 40μs to arrive to B and the last bit gets sent at 440μs and arrives at 480μs. Once again, technically more than one bit can be sent at once but this is encoded inside the bandwidth. Note that at the tutorial I drew the diagram wrong: C sent the bit as soon as it received one. This is not the case in packet-switching.**

c Consider the next diagram



Now we are sending two 100 byte packets instead of one 200 byte packet. So, C wait for the first packet to arrive and sends it to B. The first bit arrives at 40μs and the last bit of the first packet arrives at 140μs. This is also the time when the first bit of the second packet arrives. Then it waits for the second packet to arrive and sends it to B. Ultimately, the last bit of the second packet arrives at 380μs to B. **Note that at the tutorial I also drew this one slightly wrong. In my drawing A sent the first bit of the second packet only after the last bit of the first packet arrived at C. This might be the case when C need to send some acknowledgement, but in this particular example it was wrong.**

Exercise 3 Now let us consider the situation when the propagation delay is the most significant component. Suppose we need to send a packet from A to B where the distance between A and B is 10000 km, propagation speed is 200 km/ms. Also, suppose that B has to respond to A with some acknowledgement signal. How many bits of data can we send from A to B before A receives the acknowledgement for the first received bit if the bandwidth is 1 Mbit/sec? 100Mbit/sec?

At most non-LAN scales, the delay is typically simplified to the round-trip time, or RTT: the time between sending a packet and receiving a response. Try executing `ping google.com` in your terminal.

Solution 10000 km with propagation speed of 200 km/ms gives us a propagation delay of 50 ms. Because we also need an acknowledgement from B this results in 100 ms. 1 Mbit/sec is the same as 1 Kbit / ms. Thus, we can manage to send $100 \cdot 1$ Kbit in 100 ms. This is about 12.5 Kbytes. So we can send quite a lot of data before hearing anything back if the propagation delay is significant.

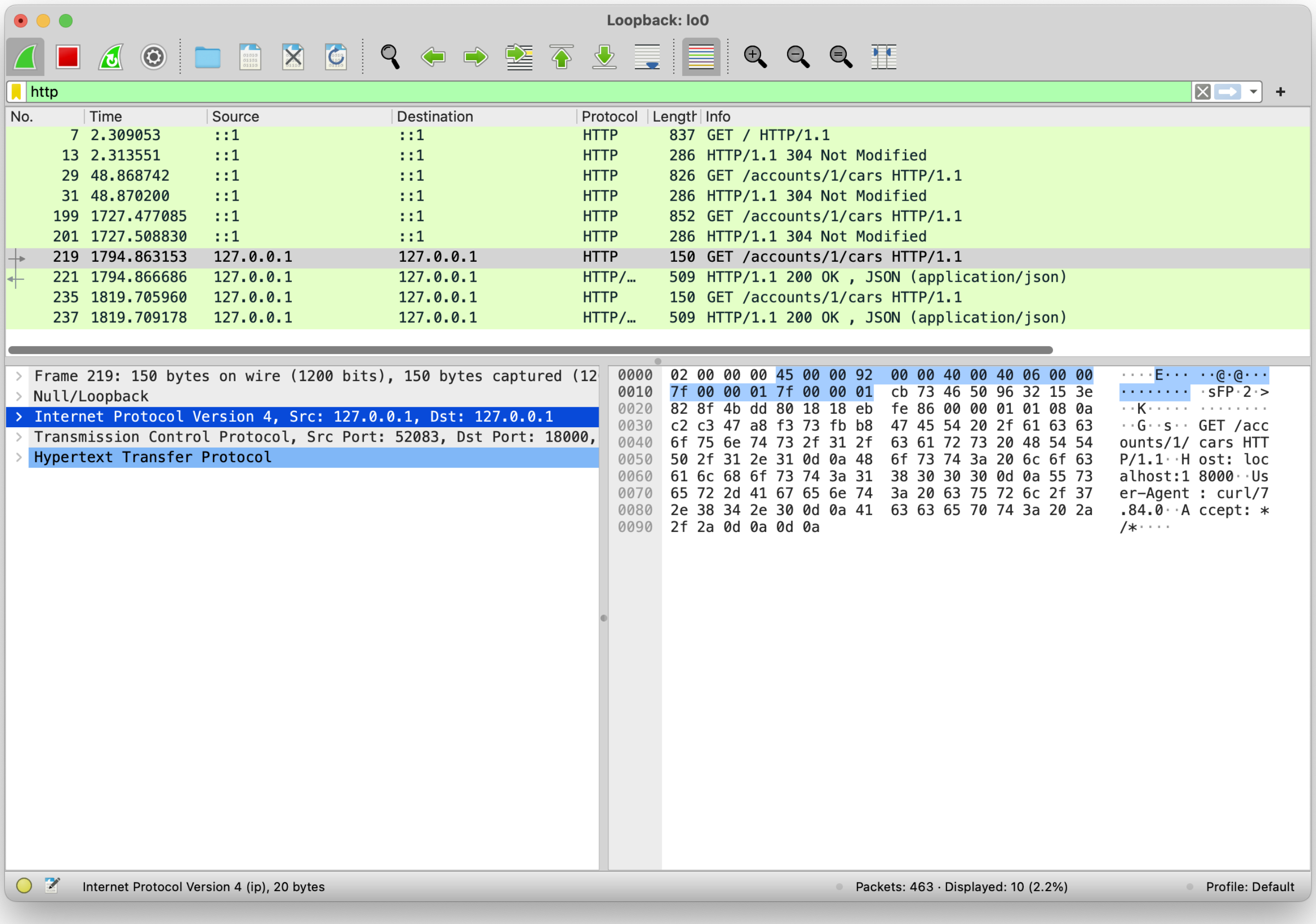
Executing `www.google.com` returns something like `round-trip avg = 22.177 ms` which is the propagation delay back and forth like above.

Exercise 4 This exercise is intended to encourage you to have a look at what various internet protocols look like. Below we will have a look at HTTP. We will need

- Wireshark traffic sniffer `https://www.wireshark.org/`
- Any HTTP server (with no encryption), for example `https://github.com/zowe/sample-node-api` Note that you need `npm` to run the server.

Clone the repository above, do `npm install;npm start` which will run the server on **localhost:18000**. Start Wireshark and select **loopback** interface for sniffing and start listening on this interface. Then do `curl localhost:18000/accounts/1/cars`. We use `curl` command line utility so that the responses do not get cached. Then stop listening and find the HTTP request that you've sent and HTTP response that you have received. Notice how your HTTP request gets wrapped into TCP and IP packets. The output should look like in Figure 4.

UPD: this will actually work with any web-server. For example, you can make an HTTP request to google by `curl http://www.google.com` and HTTPS request by `curl https://www.google.com`. Compare the corresponding packets in Wireshark (you need to select another interface to listen to, e.g. wifi or ethernet).



Loopback: lo0

http

```
No.    Time          Source       Destination  Protocol  Length Info
     7 2.309053      ::1          ::1          HTTP       837 GET / HTTP/1.1
    13 2.313551      ::1          ::1          HTTP       286 HTTP/1.1 304 Not Modified
    29 48.868742     ::1          ::1          HTTP       826 GET /accounts/1/cars HTTP/1.1
    31 48.870200     ::1          ::1          HTTP       286 HTTP/1.1 304 Not Modified
   199 1727.477085   ::1          ::1          HTTP       852 GET /accounts/1/cars HTTP/1.1
   201 1727.508830   ::1          ::1          HTTP       286 HTTP/1.1 304 Not Modified
   219 1794.863153   127.0.0.1    127.0.0.1    HTTP       150 GET /accounts/1/cars HTTP/1.1
   221 1794.866686   127.0.0.1    127.0.0.1    HTTP/…     509 HTTP/1.1 200 OK , JSON (application/json)
   235 1819.705960   127.0.0.1    127.0.0.1    HTTP       150 GET /accounts/1/cars HTTP/1.1
   237 1819.709178   127.0.0.1    127.0.0.1    HTTP/…     509 HTTP/1.1 200 OK , JSON (application/json)
```

```
> Frame 219: 150 bytes on wire (1200 bits), 150 bytes captured (12
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 52083, Dst Port: 18000,
> Hypertext Transfer Protocol
```

```
0000  02 00 00 00 45 00 00 92  00 00 40 00 40 06 00 00   ····E··· ··@·@···
0010  7f 00 00 01 7f 00 00 01  cb 73 46 50 96 32 15 3e   ········ ·sFP·2·>
0020  82 8f 4b dd 80 18 18 eb  fe 86 00 00 01 01 08 0a   ··K····· ········
0030  c2 c3 47 a8 f3 73 fb b8  47 45 54 20 2f 61 63 63   ··G··s·· GET /acc
0040  6f 75 6e 74 73 2f 31 2f  63 61 72 73 20 48 54 54   ounts/1/ cars HTT
0050  50 2f 31 2e 31 0d 0a 48  6f 73 74 3a 20 6c 6f 63   P/1.1··H ost: loc
0060  61 6c 68 6f 73 74 3a 31  38 30 30 30 0d 0a 55 73   alhost:1 8000··Us
0070  65 72 2d 41 67 65 6e 74  3a 20 63 75 72 6c 2f 37   er-Agent : curl/7
0080  2e 38 34 2e 30 0d 0a 41  63 63 65 70 74 3a 20 2a   .84.0··A ccept: *
0090  2f 2a 0d 0a 0d 0a                                  /*····
```

Internet Protocol Version 4 (ip), 20 bytes   Packets: 463 · Displayed: 10 (2.2%)   Profile: Default

Figure 4: Wireshark example

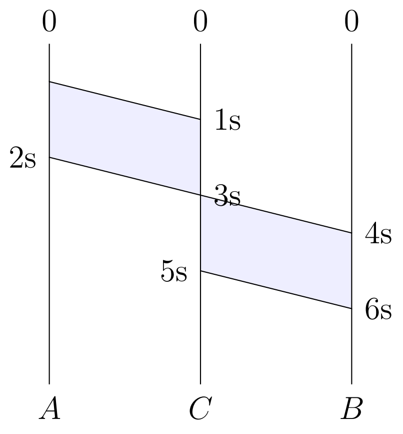
4 Week 10

4.1 Preliminaries

Let's recall how store-and-forward packet switching works. Suppose we have the following configuration and want to send a packet of size 40 KBytes from A to B and C is the intermediate router.

$$A \longrightarrow C \longrightarrow B$$

If the switching is store-and-forward, then C should receive the whole packet before it can start sending it to B. Suppose the data rate of channels $A \to C$ and $C \to B$ are the same and equal to 20 KBytes per second (160 Kbits/sec) and the propagation delay of both channels is 1 second. Then the following sequence diagram describes the time when the packet finally arrives at B.



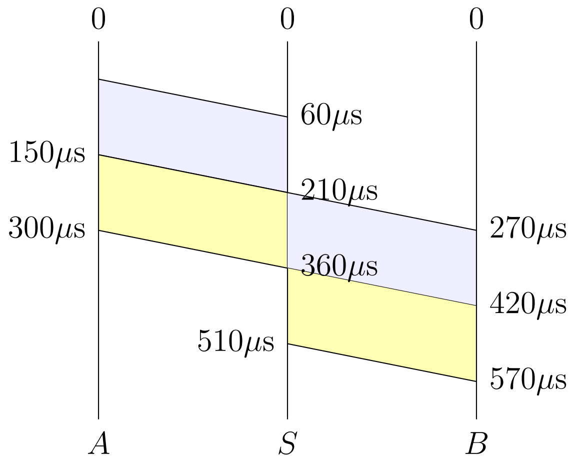
Because the bandwidth of the channel is 20 KBytes/sec, A needs 2 seconds to put all the data onto the wire and because the propagation delay is 1 second, the first bit arrives at C at 1s and the last bit at 3s. Similarly for the trip between C and B.

Recall that **throughput** is the actual amount of data that is transferred per time unit (the bandwidth is the maximal amount of data that can be transferred). Because the amount of data we sent above is more than the channel can get per second, we get that the throughput is equal to the bandwidth.

4.2 Exercises

Exercise 1 Suppose the path from A to B has a single switch S in between: $A \longrightarrow S \longrightarrow B$. Each link has a propagation delay of 60 µsec and a bandwidth of 2 bytes/µsec. How long would it take to send two back-to-back 300-byte packets from A to B?

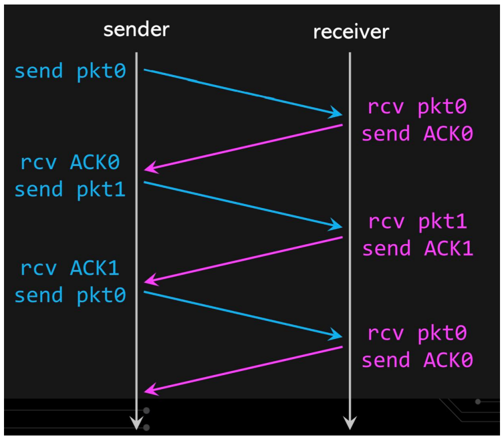
Solution Consider the diagram below.



The first bit of the first packet takes 60μs to arrive at S and 150μs are needed to put the whole packet onto the wire (300 bytes divided by 2 bytes/μsec) and hence the last bit of the first packet arrives at S at 210μs. Then the second packet can start being transmitted and the first packet can be forwarded by S (because we send back-to-back and store-and-forward). Repeating the same reasoning for the second packet and for the trip from S to B.

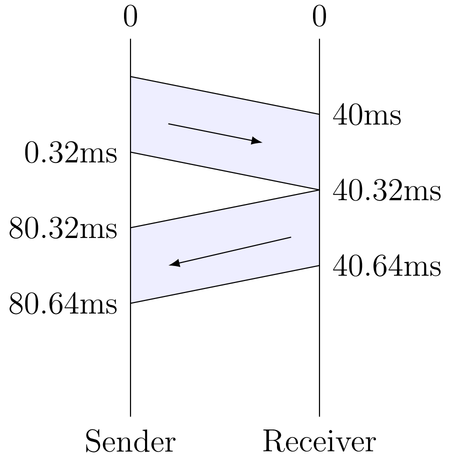
Exercise 2 We know that RDT3.0 is correct, but it suffers greatly in terms of performance due to being a Stop-and-Wait protocol. Let's assume that we have a 100 Mbps link between two devices and 40ms propagation delay, what will be the **throughput** on this link for a packet size of 4 KBytes when using RDT 3.0 protocol? Before we can send the next packet we need to wait for a packet with an acknowledgement and assume that this packet's size is also 4 KBytes and also assume that no packet is corrupted or lost.

Solution Recall how RDT works



That is, the receiver has to send an acknowledgement packet to the sender to confirm that it has received the packet. Only after the acknowledgement can the sender send another packet. In this question, we will assume that the receiver sends 4 KBytes acknowledgement response (although in practice the response contains only a header with no body).

Now consider the diagram below



The first bit takes 40 ms to arrive at the Receiver, but to put 4 Kbytes (32 KBits) of data onto the wire takes only 0.32 ms and therefore the last bit arrives at 40.32 ms. Then the sender should send these 4 Kbytes back following the same procedure and the last bit of acknowledgement gets to Sender at 80.64 ms. Ultimately, we've managed to transfer only 4 Kbytes in 80.64 ms, or $\frac{32000 \text{ bits}}{80.64 \text{ ms}} \equiv 397$ bits/ms, or, 397 Kbits/sec and hence this is our throughput (the actual amount of data transferred per unit of time). Recalling that the bandwidth was 100 Mbits/sec we get that the actual throughput is 250 times less than what we can achieve.

Exercise 3 For each IP network prefix given (with length), identify which of the subsequent IP addresses are part of the same subnet.

a **10.0.130.0/23:** 10.0.130.23, 10.0.129.1, 10.0.131.12, 10.0.132.7

b **10.0.132.0/22:** 10.0.130.23, 10.0.135.1, 10.0.134.12, 10.0.136.7

c **10.0.64.0/18:** 10.0.65.13, 10.0.32.4, 10.0.127.3, 10.0.128.4

Solution We will see the solution for (a) and the rest are similar. Recall, that the notation $x.y.z.w/m$ means that m first bits correspond to the subnet and the rest bits are available for hosts (and for the reserved subnet and broadcast IP addresses within the subnet). To see the range of available addresses we need to first convert the denary IP address into binary:

$$10.0.130.0/23 = \underline{00001010.00000000.1000001}0.00000000$$

where the subnet's part is underlined. The rest of the bits are available addresses within the subnet. Because the rest of the bits are all zeroes this is also the smallest address in the subnet (or the subnet's address). The largest available address is when all the bits are ones (this is also a broadcast address):

$$\underline{00001010.00000000.1000001}1.11111111$$

or, in denary,

$$10.0.131.255$$

All that is left is to check whether a given IP address falls in this range.

1. 10.0.130.0 < 10.0.130.23 < 10.0.131.255 and hence it is inside the range
2. 10.0.129 < 10.0.130.0 and hence it is not inside the range
3. 10.0.130.0 < 10.0.131.12 < 10.0.131.255 and hence it is inside the range
4. 10.0.132.7 > 10.0.131.255 and hence it is outside the range.

Also note that only the hosts within the same subnet (i.e. host 1 and 3) can communicate directly. If host 1 wants to communicate with host 2 it has to delegate this job to the router which connects two networks.

Exercise 4 By using Wireshark trace the TCP packets that arise when communicating with a web server over HTTP. In particular, have a look at the initial handshake and sequence and ACK numbers.

Solution You can check out this video: `https://youtu.be/3Zb_EebU22o?si=F9MnfE1mGEJv53nz`. This is pretty much the same thing I showed at the tutorial.

5 Week 11

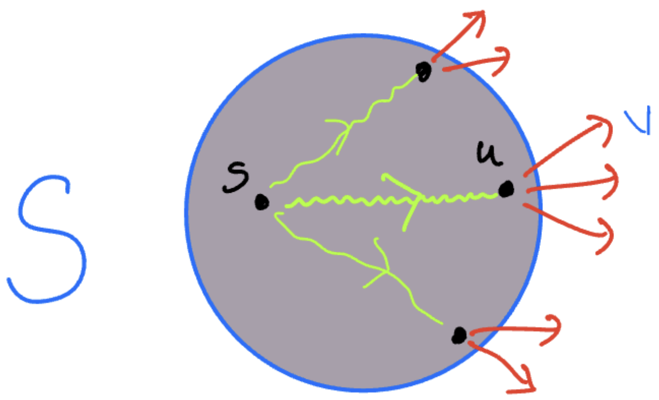
5.1 Preliminaries

Graph A graph is a pair of two sets: a set of *edges* and a set of *verticies* or nodes. We will denote it like (V, E) where elements of E are also pairs like (u, v) that stands for an edge from vertex u (the source) to vertex v (the target). We will then consider weighted graphs such that edges are of the form (u, w, v) where w denotes the weight of an edge, e.g. how 'costly' it is to follow that particular edge. A *path* from vertex u to vertex v in a graph is a sequence of nodes $(v_1, \ldots, v_n)$ such that the first vertex in the sequence $v_1 = u$ and the last vertex is $v_n = v$ and there is an edge between v_i and v_{i+1} for all $i < n$.

Dijkstra Algorithm This algorithm finds the distances of shortest paths from one source vertex v to all other vertices in a weighted graph such that all weights are positive. The pseudo-code is given below.

```
# s is the source vertex
    fun dijkstra(s):
    # D stores distances from s to all other vertices
        D = {}
        S = {s}

        for v in V:
            if (s,v) in E:
                D[v] = c(s,v) # c is the cost of the edge
            else:
                D[v] = float('inf')

        while S != V:
            v = min(D) and v not in S
            S.add(v)

            for (v,u) in E:
                D[u] = min(D[v] + c(v,u), D[u])

```

S is the set of currently visited vertices v' and it is guaranteed that the distance from s to $v' \in S$ is the shortest. Then the algorithm picks a vertex v not from S such that the distance to it from the source node is minimal across all candidates and updates the distances to all v's neighbours.



Note that if one also would like to retrieve the shortest paths themselves, it is needed to maintain the predecessor of each vertex, e.g. in an array *pred*, and then, to get the path from s to some given vertex v we need to trace $pred[v]$ until we reach s and reverse the sequence. For example, if $pred = [0, 0, 1, 2]$ (the value in the array for index i gives the predecessor of vertex i) and $v = 3$ and $s = 0$ it would mean that the predecessor of s is s, the predecessor of vertex 1 is s, the predecessor of vertex 2 is vertex 1 and the predecessor of vertex 3 is vertex 2. So the path from s to v is $(s, 1, 2, v)$. The changes in the code would be the following.

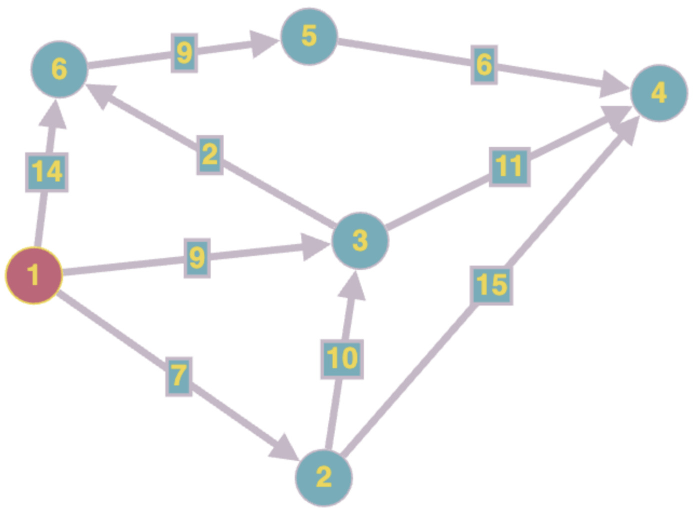
```
for v in V:
    if (s,v) in E:
        D[v] = c(s,v)
        pred[v] = s
```

and

```
for (v,u) in E:
    dist = D[v] + c(v,u)
    if dist < D[u]:
        D[u] = dist
        pred[u] = v
```

5.2 Exercises

Exercise 1 Find the distances of all shortest paths from source 1 in the graph below by using Dijkstra algorithm.



What is the set S at each step? What are the distances at each step?

Solution We will maintain the sets S and D across the iterations. First, according to lines 3-12 of the pseudo-code, we need to initialise D and S and hence $S = \{s\}$ and we will represent D as a table:

Node	Distance
1	0
2	7
3	9
4	∞
5	∞
6	14

Where Distance corresponds to the distance from the source node (1) to the Node in Node column. Then we go to lines 14-15 and pick the node with the minimal distance to it from s such that the node is not in S. This is node 2. So now $S = \{1, 2\}$ and we need to update the distances from node 2 to all its adjacent nodes, that is, to nodes 3 and 4:

Node	Distance
1	0
2	7
3	9
4	22
5	∞
6	14

The previous distance to node 3 was 9 which is shorter than the distance to 2 plus 10 (the cost of the edge between 2 and 3) and hence we keep the old distance for 3. In contrast, the distance to 4 was ∞ and we replace it with the distance to 2 plus the cost of the edge from 2 to 4 (15). We again need to pick the node with the shortest distance from s and this time it is node 9. Hence, $S = \{1, 2, 3\}$ and we update distances to $4, 6$.

The next node we pick is 6 and $S = \{1, 2, 3, 6\}$ and we update the distance to 5.

Node	Distance
1	0
2	7
3	9
4	20
5	∞
6	11

Node	Distance
1	0
2	7
3	9
4	20
5	20
6	11

Now we can pick either 4 or 5 as our next node. Let's pick 4. It has no adjacent nodes, so we proceed with $S = \{1, 2, 3, 6, 4\}$ and pick node 5. The previous distance to 4 is shorter than the distance to 5 plus the cost of the edge from 5 to 4 (6) and hence we keep the previous distance. Since $S = V$ we exit the `while` loop. The distances are the following.

Node	Distance
1	0
2	7
3	9
4	20
5	20
6	11

Exercise 2 What is the complexity of the above algorithm in terms of V and E?

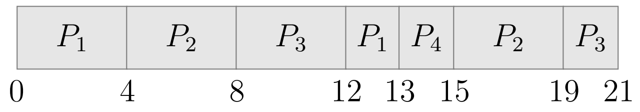
Solution The lines 4-12 take $\mathcal{O}(V)$ steps. The `while` loop performs at most V iterations. To find a minimum in an array D we need V steps. And the loop at line 17 takes no more than E steps. Hence, in total, we have $\mathcal{O}(V + V \times (V + E)) = \mathcal{O}(V^2 + VE)$. If we assume that there must be at most 1 edge between any two nodes, the number of steps in the loop at line 17 will be bounded by $V - 1$ (as any given node can be connected at most to every other node except itself) and the complexity will be $\mathcal{O}(V^2 + V^2) = \mathcal{O}(V^2)$. With a more sophisticated data structure for retrieving the minimum value, the complexity can be improved.

Exercise 3 Consider the following set of processes:

What is the average response time assuming Round Robin CPU scheduling policy with a Time Quantum (q = 4 ms). Also, note that newly arriving processes are added to the tail of the ready queue. The response time is the time it takes for the process to be first allocated a CPU after it arrives.

Process	Arrival Time	Burst time
P_1	0 ms	5 ms
P_2	2 ms	8 ms
P_3	3 ms	6 ms
P_4	5 ms	2 ms

Solution We will show the Gantt diagram and the ready queue at each timestamp.



Time	Queue (head to the right)
0	P_1
2	P_2
3	$P_3; P_2$
4	$P_1; P_3; P_2$
5	$P_4; P_1; P_3$
8	$P_2; P_4; P_1; P_3$
12	$P_3; P_2; P_4; P_1$
13	$P_3; P_2; P_4$
15	$P_3; P_2;$
19	P_3
23	

The response time for each process is given below (it is the time the process is first given the CPU minus its arrival time)

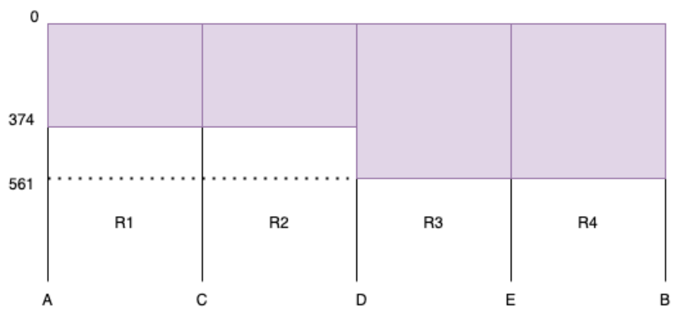
P_1	$0 - 0 = 0$
P_2 $4 - 2 = 2$	
P_3	$8 - 3 = 5$
P_4	$13 - 5 = 8$

And the average is $(0 + 2 + 5 + 8)/4 = \frac{15}{4} = 3.75$.

Exercise 4 Suppose Host A wants to send a large file to Host B. The path from Host A to Host B has four links, of rates R1 = 12 Mbps, R2 = 15 Mbps, R3 = 8 Mbps and R4 = 25 Mbps. What will be the minimum end-to-end delay of transferring a video clip of 535 MegaBytes? (We assume that there is no store-and-forward delay on the routers connecting the links)

Solution First thing we need to do is convert data size into bits because the bandwidth is measured in megabits. 535 MegaBytes is $535 \cdot 1024 \cdot 1024 \cdot 8 = 4\,487\,905\,280$ bits. We will also assume that once a bit of data has been sent it is immediately received by the receiver. Pictorially we can represent this scenario as follows.

As there is no propagation delay the first bit immediately arrives at B at time 0. Then, it takes 535 MegaBytes / R1's bandwidth seconds to send the data from A. In other words, the last bit sent from A



arrives at C at time 374 seconds. Since R2's bandwidth is higher than the one of R1 there is no delay between C receiving the data and forwarding it and R3 receives the last bit also at time 374 seconds. Then, R3's bandwidth is lower than R2's and thus D can not send the data over R3 at the same rate it receives it. It takes 535 MegaBytes / R3's bandwidth seconds to put all the data on channel R3, or 561 seconds (560.98816, to be precise). Because R4's bandwidth is higher than R3's there is no delay between E receiving and sending. Therefore, the transmission is bounded by the bandwidth of the slowest channel.

Exercise 5 In the **SYNACK** message of the TCP connection establishing handshake, the server sends a 1-byte TCP segment to the client. That segment has Sequence number = 900, SYN/ACKbits = 1. Which of the following TCP headers are correct for the corresponding **ACK** message from the client to the server?

a) Sequence number = 901, Acknowledgement number = 901, SYNbit = 0, ACKbit = 1

b) Sequence number = 500, Acknowledgement number = 901, SYNbit = 0, ACKbit = 1

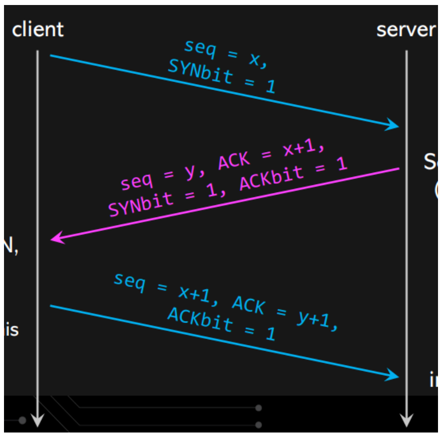
c) Sequence number = 901, Acknowledgement number = 483, SYNbit = 0, ACKbit = 1

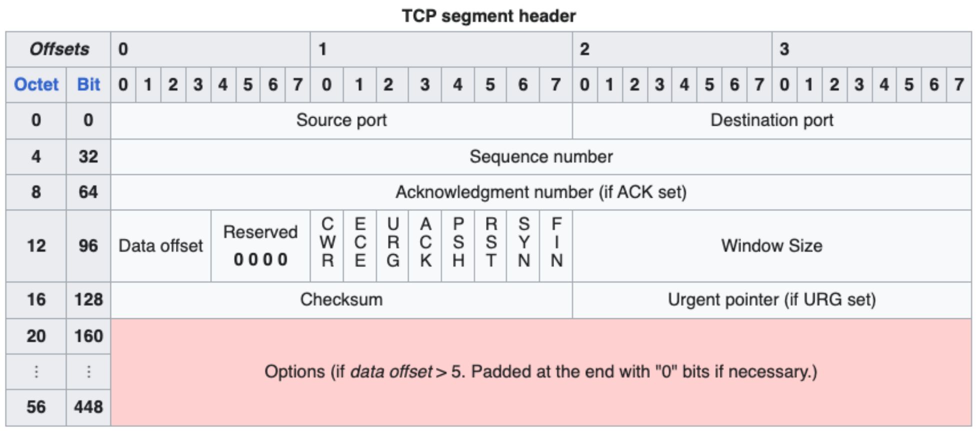
d) Sequence number = 483, Acknowledgement number = 901, SYNbit = 1, ACKbit = 1

e) Sequence number = 901, Acknowledgement number = 901, SYNbit = 1, ACKbit = 1

f) Sequence number = 901, Acknowledgement number = 483, SYNbit = 1, ACKbit = 1

Solution Recall TCP handshake diagram.





TCP segment header

Offsets		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0 0				CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if *data offset* > 5. Padded at the end with "0" bits if necessary.)																															
⋮	⋮																																
56	448																																

and the TCP header structure.

We are interested in the lowest arrow which goes from the client to the server. According to the protocol, ACK bit should be 1, SYNbit should be 0 and the ACK number should be sequence number + 1 where the sequence number is from the server's response (900). That is, ACKbit = 1, SYNbit = 0, ACK = 901. Only the first two options fit these constraints. I encourage you to check out how actual TCP packets look like in this video: `https://youtu.be/3Zb_EebU22o?si=F9MnfE1mGEJv53nz`