IN2140 exam – spring 2020

Candidate number: 15333

Task 1 – function calls vs System calls

How Function calls works

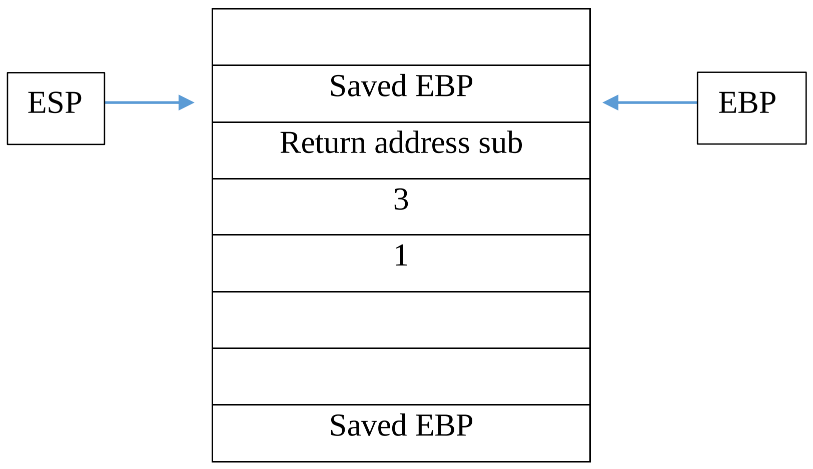
Function calls is a way for the program to pass the control to a function that performs a specific task. When we write programs, we often do not want all the code in a single main function, so it might be smart to split the code into several functions. That way, it is easier to keep the code in order, and the functions can be reused in several places. Let us say we have a function for subtracting two integers:

```
int sub(int x, int y){
        return x – y;
}

main(){
        sub(3, 1);
}
```

The stack will contain the arguments, variables and addresses for the function. When executing this program, a new frame will be allocated on the stack which the base pointer points to. First the return address for the main function will be pushed to the stack and saved.

Then, it pushes the arguments from the call on the sub() function in reverse order to the stack. The stack pointer will move upwards while variables are getting pushed. The return address for the called function sub() will then be pushed to the stack. After that, the frame pointer will be moves up to where the stack pointer is and grows the stack with another stack frame. The reason for this is that sub() could be needed the extra memory in case it has local variables to assign (does not happen in this case). The stack will now look like this:



When the sub() function return, the return value register gets the return value which in this case is 2. After done executing we want to go downwards in the stack. It pops the base pointer to make it go back to the previous frame. When returning the instruction pointer gets popped and the parameters are also popped in order to restore the stack for new functions.

How System calls works

System calls works as an interface between the user applications and the OS components. It is a way for the program to request a service from the kernel of the OS like memory or resources to be accessed. If we need a program to be able to read and write to a file, or fork and exit a process, then it means we need to use system calls in order to do that. System calls works pretty much the same as functions calls with registers and stack.

When executing a system call like exit(int status), the application will push the parameters on the stack. The library code will often be called to translate, and the system call number will be put in the register. Afterwards, the system call handler within the kernel will use this system call number to find and execute the requested operation which is sys.exit(). The instruction pointer will be increased, and the stack get cleaned up by removing the parameters. Since the instruction was sys.exit(), the process will be terminated (Lecture – kurs og OS intro, January 13th).

Differences between them

The difference is that system calls goes to the kernel to get access to services from the OS can provide. A function call is just a request from a function made by the user or pre-made in a library to perform a specific task.

Task 2 – Process and Threads

A process is the execution of a program that has a collection of instructions. In task two we have an example program, and a process will start once we execute that program. A process can also be in a different state from time to time, so this is something to pay attention to. The process may have just started, or it could be waiting for access for some particular data, or it could be interrupted by another process and so on. Therefore, the OS have to keep track of the state of these processes in a process table entry in order to ensure good productivity (lecture – OS: prosesser og CPU, January 20th). Also, we often want to switch from a running process to another one with context switching that stores the current state of the process. We can make processes for example with the system call fork() which makes a new child process that is in the same state as the one calling fork().

Switching between processes could be costly so threads could be a good solution. There could be multiple threads within a process which handles individual instructions. We avoid a lot of costs if we use thread to execute different task instead of using multiple processes. Threads can resemble small processes within a process that shares many of the same process resources, but also have their own local data (lecture – OS: prosesser og CPU, January 20th). Processes needs to switch between some resources like memory, address space and other global data within the process that threads do not, because threads share those things.

The difference between process and thread matter a lot for scheduling on servers like login.ifi.uio.no. Each user should get a process and within these processes there are many threads that are sharing common resources in the process. The threads can do tasks in parallel which is crucial for big servers.

Task 3 – Scheduling

A schedule is preemptive when the scheduler allows the tasks to be interrupted. The interruptions often occur because tasks have priorities. If a task got a higher priority than the one already running, then the highest priority task should be running instead (context switching). Preemptive schedules are schedules.

Non-preemptive scheduling is the opposite of preemptive. These schedules do not interrupt running processes in the middle of an executions. The tasks get to use their time and finish their job before the next one. Processes with higher priority also have to wait for their turn.

First, I thought that it could be smart to use a preemptive schedule because the system there are warning messages that the system is sending. The warning messages tells if there is something wrong with the system and that seems pretty crucial. But the assignment task also says that none of the status updates has priority and the system has no user interface, therefore it is better to use non-preemptive scheduling.

First in first out (FIFO)
Using the first in first out algorithm will complete the tasks in the order that they started. This is a non-preemptive schedule because the tasks are not prioritized executed and are executed in the order they arrived in the queue. Using this algorithm for this task could be a good solution because none of the status updates has priority, and I assume that the processes are endless so a dynamic algorithm like FIFO would be great. In addition, the implementation of FIFO is pretty simple. Usually, FIFO is not a great choice when it comes to time, but the tasks in this case is not that time consuming and they are all similar.

Shortest job first (SJF)
The shortest job first algorithm is also a simple algorithm like FIFO. SJF's main characteristics is that it calculates the runtime of every task and executes the ones that are shortest first. This way, the average amount of time the processes needs to wait will be minimized. Using SJF in this case could be smart, but it could also be hard to determine processing requirements since we get new processes all the time. In addition, SJF assumes that all processes are already ready in the queue.

Round-robin (RR)

The round-robin algorithm is similar to FIFO but unlike FIFO, RR is a preemptive scheduling algorithm. RR uses time slices to give all the tasks equal runtime in a circular order. When a process has ran in a certain amount of time, the scheduler will force the process to stop and give the allocation of CPU to another process. Using this algorithm in this case would not work because RR is preemptive, and the status updates are not prioritized.

Earliest deadline first (EDF)

The earliest deadline first algorithm is similar to SJF. Instead of shortest job, the highest priority tasks get executes first. This algorithm is also based on priority of tasks, so this will not be suited for this case.

For this case I would choose FIFO as the scheduling algorithm. The reason is that it is simple to implement, the task difference is minimal and there is no priority.

Task 4 – Virtual memory

Multi-level paging is a way to divide the pages into several levels of paging when the page table gets big to be able to loop up every page that we need. The tables got a tree structure with the smaller page tables linked to a main page table.

When we want to lookup memory in multi-level paging with for example 3 levels, we use the first bits in the virtual address as an index to find the element that we want. The entry that this pointer points to, points to another page table and the middle bits in the virtual address will point to the entry we want. Finally, this entry will point to the start of the page that we were looking for. The last bits in the virtual address is the offset that gives us the right element we are looking for within the page. To make memory lookups faster and to improve the overall performance, the OS uses unused parts of the RAM to keep page cache.

To increase performance even more, paging systems can predict which pages will be needed soon, preemptively loading them into RAM before a program look them up (Wikipedia, https://en.wikipedia.org/wiki/Paging, 2020). Page fault like a page frame being “dirty” can happen which means that a page in memory have changed from what is currently stored on disk. To handle a “dirty” page can be done by writing back to disk or just overwrite.

Offset is 10 bits because the page size is 2^{10}.

There are $\frac{2^{32}}{2^{10}} = 2^{22}$ pages.

22-bits are therefore needed to address a single page.

A single page can hold 1024 entries, so we need a total number of pages for each level:

First level: 2^22 / 2^10 = 4096.

Second level: 4096 / 2^10 = 4.

Third level fits on one page because 4 < 256.

The number of minimal requires levels are therefore 3.

Task 5 – Splitting an address block into networks

We have 1024 (2^{10}) addresses. Starting with netmask 255.255.255.255.

Netmask in binary: 11111111 11111111 11111111 11111111

Since we have 2^{10} addresses we need to remove 10 1’s for the host part.

The final netmask for the company will be: 11111111 11111111 11111100 00000000 = 255.255.252.0

CIDR notation: 9.239.16.0/22

I chose to split the 1024 addresses in 4 networks. The three networks netA, netB and netC get 256 (2^{8}) addresses each.

To find the netmask for these networks, we just do the same as the whole company.

First, we start with the netmask 255.255.255.255.

If we remove 8 1’s from the netmask for the host part, we get the netmask for each network:

11111111 11111111 11111111 00000000 = 255.255.255.0

CIDR notation: 9.239.16.0/24

The reason I chose to divide the addresses like this is that there is no good way of splitting 1024 addresses in 3. Each of the locations requires only 120 public addresses anyways, so 256 should be more than enough. The offices will have a possibility for future growth with some addresses in reserve, and the company also got a spare subnet in case they want expand their business with another office. If there are new employees coming or other network units, they will get an address from the reserve ones.

Task 6 – Packets over TCP

the statement “TCP provides a byte-stream service” means that “it doesn’t offer a packet service to the application layer” (lecture: Datacom: Forbindelsesorientert og forbindelsesløs kommunikasjon, March 30th , 2020). The sender writes the data it wants to send as bytes through a TCP connection, and the receiver will read the bytes out. TCP splits the byte stream into segments that fit into an L3 packet that IPv4 or IPv6 can use.

An application with a client and a server which communicate with a socket and uses functions like send() and recv() needs to make sure that all of the data get transferred. It can take several calls on send() and recv() to transfer all data, since TCP sends a byte-stream divided into segments, and stream sockets send and receive information in streams of data (IBM, https://www.ibm.com/support/knowledgecenter/SSLTBW_2.3.0/com.ibm.zos.v2r3.hala001/senrecconversation.htm). When using recv() the receiver needs to have a loop that runs as long as there are bytes that are not received yet in order to make sure that everything is transferred.

Things that programmers do differently when using TCP instead of packet-based services like UDP:

- Receiver uses a socket to listen for connecting requests and set up a connection between the sender and receiver because TCP is connection oriented.
- Handle connections.
- Handle receiving byte-streams. Receive as long as there is something to receive.
- Sender needs to establish a connection with the receiver.

- Needs to provide flow control in order to make the sender understand how much it can send.
- Send acknowledgment to sender after receiving.
- Make sure that the sequence of the packets is in order.

If an application uses TCP but needs to send packets, the developer must represent the packets like an array of bytes. The array of bytes splits into segments that fit into an L3 packet containing a header with protocol information like destination port and sequence number, and the payload which is the data that we want to send. When the receiver gets a package, it needs to extract the content from the package.

Let’s say the sender wants to send a struct “car” as a package containing model, model length, registration number and year. The sender sends this content in an array of bytes with a header. The receiver must extract those bytes and can put the extracted data in a struct on the receiver side to get the same packet format. After receiving a packet, it needs to send an ACK packet by sending a header without any payload, that informs the sender that the packet is received and with the right order.

PacketTP changes:

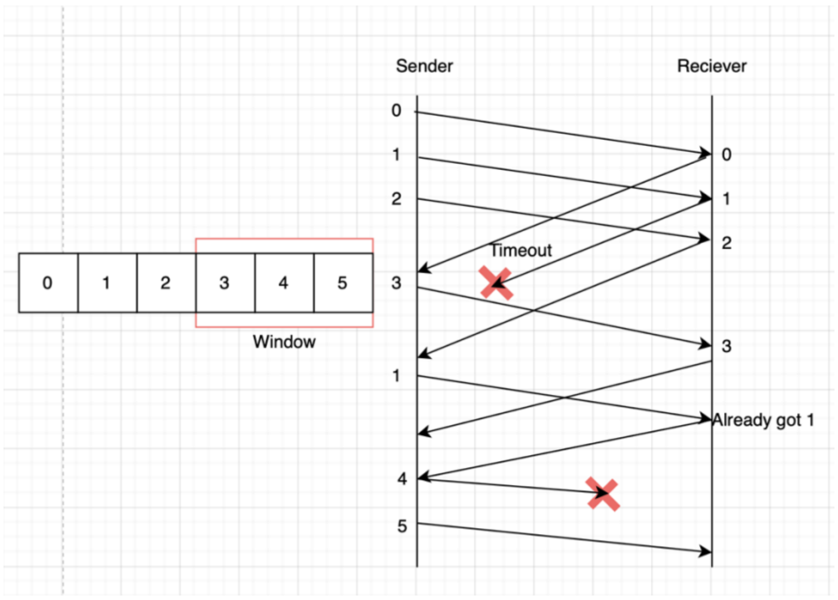
- Send datagram instead of byte-stream.
- Check if the number of bytes sent is equal to the number of bytes received.
- Change packet format.

Task 7 – Flow control

Selective repeat

The flow control mechanism selective repeat within the sliding window protocol is a way of handling the packets from they are sent from a client until they are received from the server. Packets are getting sent as a window with a size of equal or less than the number of sequence number divided by two. The main characteristic about selective repeat is that the sender does not have to send every single packet in the window if a packet gets lost, because the receiver stores the other packets that did not get lost in a static buffer. The receiver has to keep track of the sequence number of the packets that are not received in order to show the sender by sending ACKs that it needs to be retransmitted. If the sender does not receive an ACK for a packet, then it has to wait out the timeout before sending it again. The only packets that gets retransmitted from the sender are the ones that are damaged or missing (lecture - datacom: Flow control, April 20th, 2020). Here is an illustration of selective repeat with window size as 3, and 6 packets with sequence number 0 – 5:

Selective repeat example



- The first tree packets get sent, received and ACKed.
- Immediately after ACK for packet 0 is received, the sliding window will advance one packet and send the next packet which is 3.
- ACK for packet 1 got lost and the sender got a timeout. While Having a timeout, the ACK for packet 2 is received.
- Sender will now send packet 1 again. Since the receiver already has stores packet 1, it only sends a new ACK for this packet.
- ACK for packet 3 is received, followed by the ACK for packet 1. All The packets in the current window is now received after waiting for ACK for packet 1.

- The sliding window will now advance for the two-last packet, 4 and 5, and send these. What will happen now is that 5 will get received before 4, so the challenge with selective repeat is that packets are often not received in sorted order.

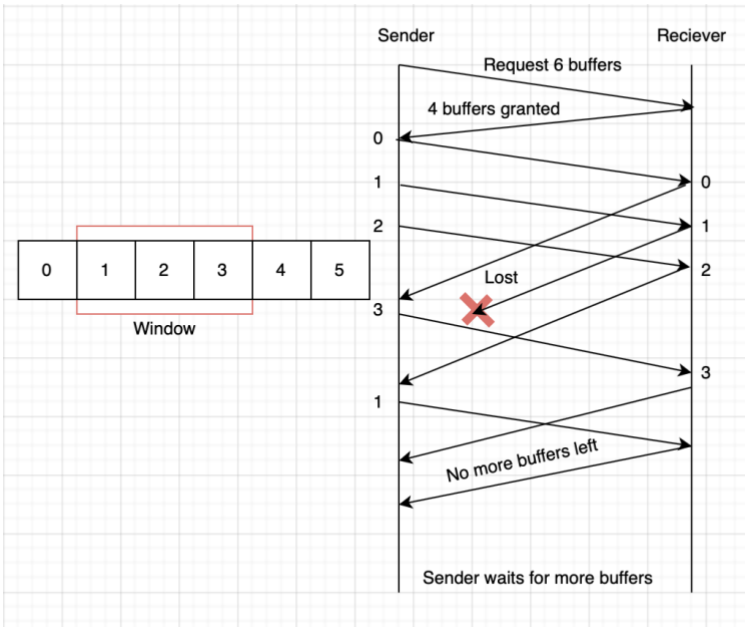
Selective repeat is an efficient mechanism because of the receiver keeps the packets in a buffer even if there are other packets that gets lost. However, it is a bit complex and it requires some memory for the buffer.

Credit mechanism

The flow control credit mechanism within the sliding window protocol is similar to selective repeat. Both of the flow control mechanisms got a buffer allocation for receiving packets at the receiver side.

The credit mechanism is a way of handling multiple sliding windows to multiple connections by allocating the buffers dynamically. The number of packet that a receiver can receive depends on the memory situation. The sender needs to request a specific amount that it needs from the buffer, and the receiver needs to reserve as many of the buffers that it has opportunity to reserve. The receiver notifies the sender with the amount of buffer-credit it got, and the sender can keep on sending packets as long as it has credits. ACKs have the same meaning in the credit mechanism but it will also include the amount of credits the sender has left (lecture - datacom: Flow control, April 20th, 2020).

Let’s say a sender asks the receiver to reserve 6 buffers, but the receiver only got 4 left because the others are already in use. The sender will then send the packet in the first window which has a size of 3.



The sender requests 6 buffers but only gets 4. The reason for this could be that there already are too many connections, or other connections require a lot of buffer. Buffer availability really depends on the current situation. Packet from the window get sent and it can keep on sending as long as it is within the granted buffers. If a packet gets lost it will be resent eventually, but the buffer limit could have already been reached.

Differences and loss of ACKs

The difference between selective repeat and credit mechanism is the buffer the receiver has. Selective repeat allocates a static buffer while credit mechanism allocates a dynamic buffer (lecture - datacom: Flow control, April 20th, 2020).

Selective repeat has a timeout for each packet and will wait for these packets to be received and ACKed, while the credit mechanism has a timeout once the credit is used up. Once the credit mechanism has used all the credit, we could end up losing or waiting for a long time for an important packet. Let's say the sender gets 3 buffers. If we send packet 0 to 2 and loose packet 1, the immediately send packet 3 after receiving ACK for packet 0. Since packet 1 got lost and the whole buffer filled up, then we have to wait for more space for this one.
A solution could be to keep track of how many sent packets there are compared to how much buffer it has. If there are 3 buffers and we want to send 4, then we have to make sure that it is

the first 3 that get those buffers. This will lead to more use of time, but we will not lose intermediate packets.

Task 8 – Building routing tables from Dijkstra’s algorithm

Steps for Dijkstra’s shortest path first algorithm (lecture: Datacom: Routing, May 11th, 2020) :

1. Node A labeled as permanent
2. Relabel all directly adjacent nodes with the distance to A (path length, nodes adjacent to source)
3. Examine all tentatively labeled nodes, make the node with the smallest label permanent
4. This node will be the new working node for the iterative procedure (i.e., continue with step 2.)

Explaining

Dijkstra’s algorithm has several versions, but I chose the one in the lecture. The goal is to find the shortest path from a starting node to every other node where every node has a certain distance between each other.

The first step is to label the starting node as permanent. Since the starting node is A, we have to label that as permanent to keep track of which nodes are visited or not, and the shortest path from A to A is 0 as well.

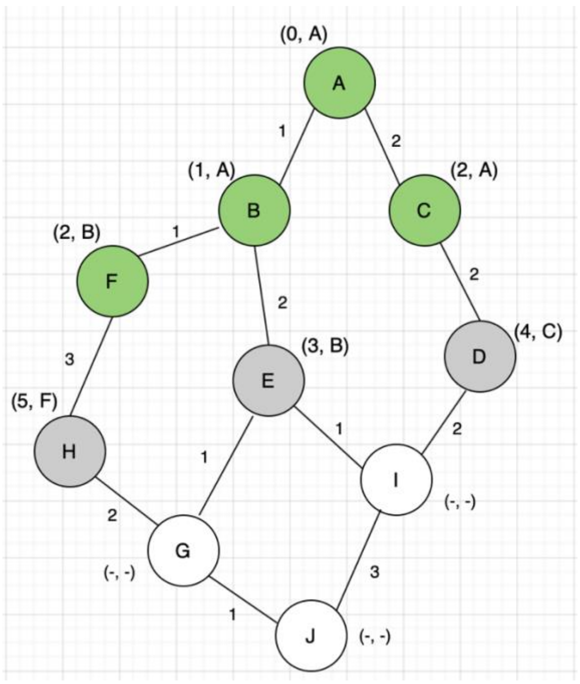
The second step is to relabel the connected node with the distance from A. For example, node D will be labeled right after C is permanent with the distance from A to C plus the distance from C to D, which is 4, and with the previous node we are working with. The reason the algorithm does this is to keep track of all tentatively labeled nodes and to be able to compare the nodes and pick the next move on the path.

Now, it makes the node with the smallest label permanent in order to work through the nodes and to make sure that it always chooses the smallest possible path. It does not matter if there are similar labels to choose from because the algorithm will find the shortest path anyway, but the intermediate paths may be a little different. If a node already has been labeled from another path, the smallest distance from these two paths is the one who wins.

Finally, it sets the permanently labeled node as the new working node and begins from the second step again. This last step makes a loop to make the algorithm visit every single node. Without this step it would just go to one node further.

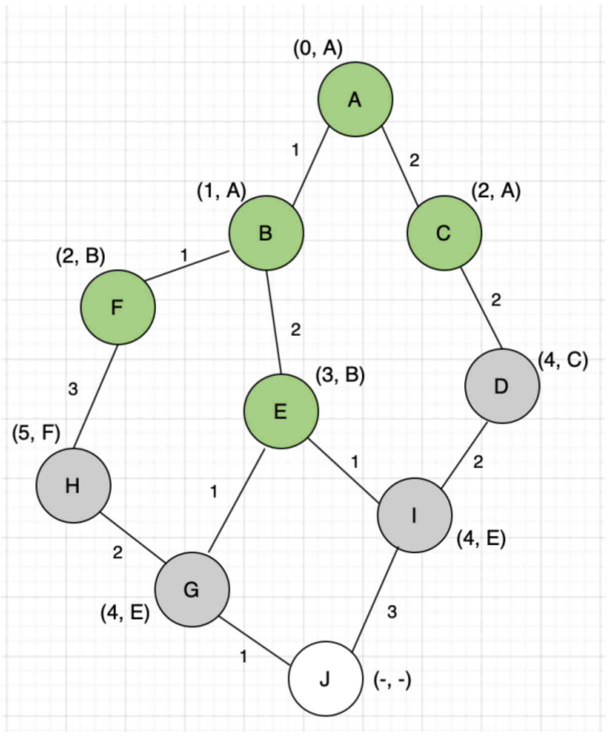
Green = permanent labeled nodes
Grey = labeled adjacent anodes

After F is permanent



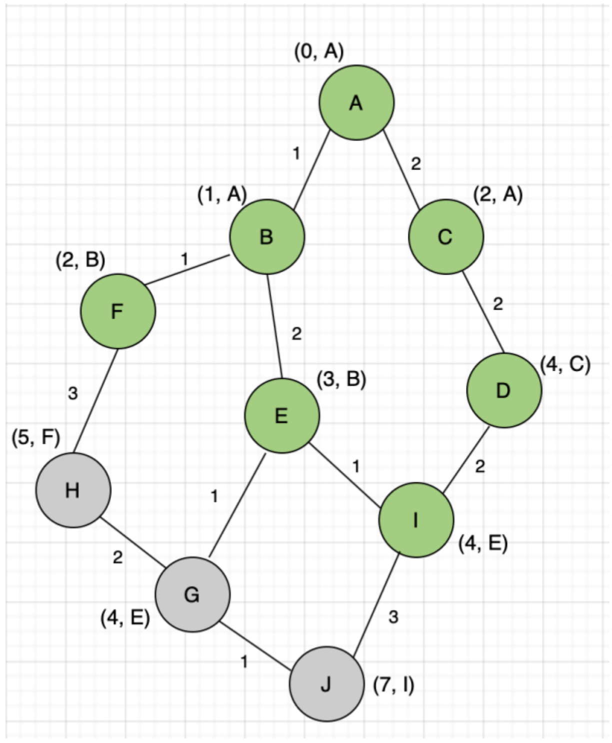
It starts with A as permanent. Labels both B and C with distance from A. Goes to B since it is the shortest one. Then labels F and E with distance from A. The distance from C and F is both 2, so it does not matter with one to choose. Goes to C and labels D with distance from A. Finally, it goes to F and labels H with distance from A.

After E is permanent



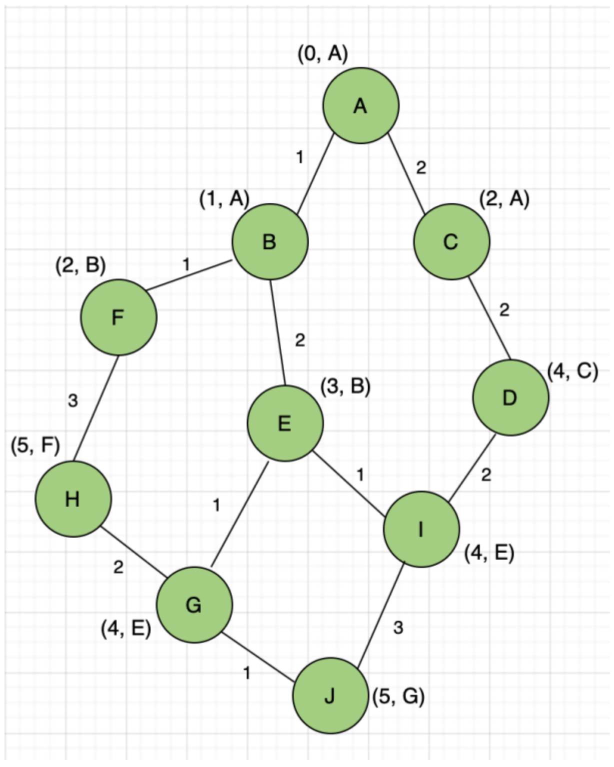
It goes to the smallest labeled node which is E. Labels G and I with distance from A.

After I is permanent



D and I got the same distance, but it chooses D. The path from A to I through E is smaller than the path through D, so I will keep the same labeled distance. J gets labeled with distance from A.

After all nodes are permanent



It goes to G because it got the smallest distance. J gets a new labeled distance because the distance through G is smaller than the one through I. At last, H gets permanently labeled, followed by J.

A’s routing table

Destination node	Next node on the path	Length of the shortest path
A	A	0
B	A	1
C	A	2
D	C	4
E	B	3

F	B	2
G	E	4
H	F	5
I	E	4
J	G	5