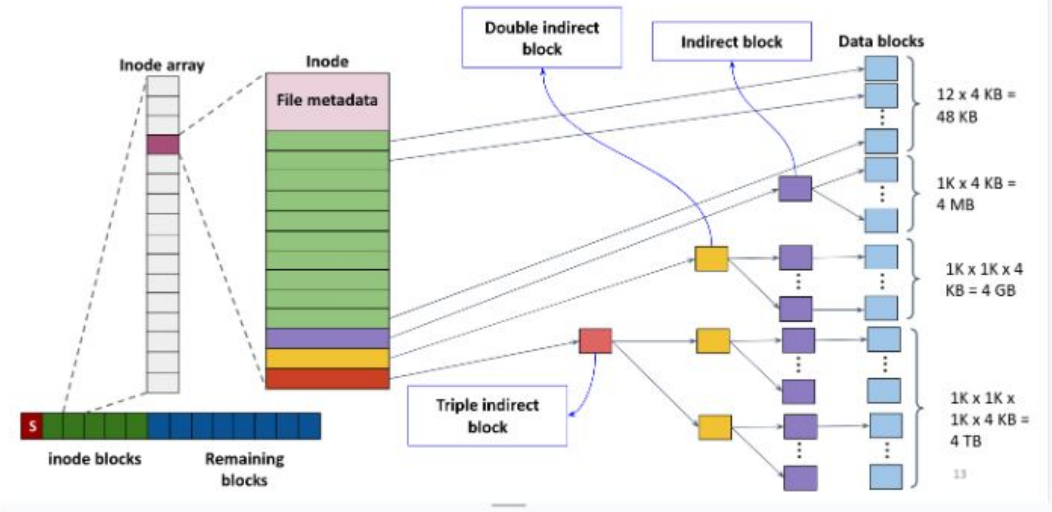
0: stdin
1: stdout
2: stderr

```
int add_all(void* tab, size_t el_size, size_t nb_el,
void* (*add_element)(void*, void*), void** result);
```



Ici nos tables intermédiaires stockent 1000 pointeurs vers des data blocks, donc on a 1,000 * 4Kb = 4Mb!

```
typedef struct _node {
        struct _node* left;
        struct _node* right;
        int*            shared;
} Node;
```

attention const vs non-const

ne pas oublier
if (ptr != NULL) {}

realloc(ptr, new_size)

node->prop

typedef double (*ma_func)(double);

```
double apply(ma_func fptr, double x)
{
        return fptr(x);
}
```

Under the simplifying assumption that there is no CPU cache, what must the CPU do in order to write value to the memory location that stores variable `x` ?

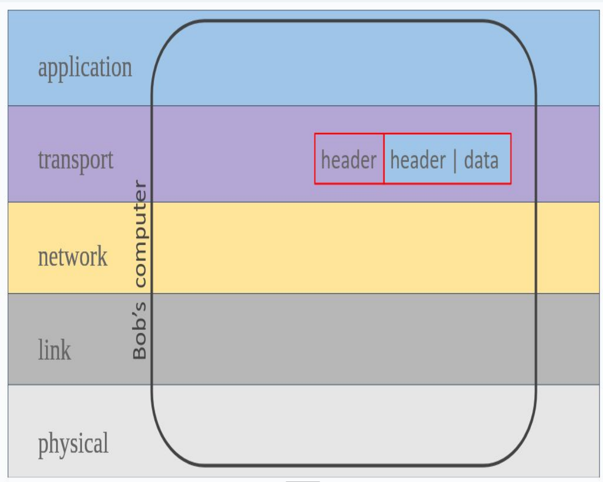
The CPU must perform the following operations:

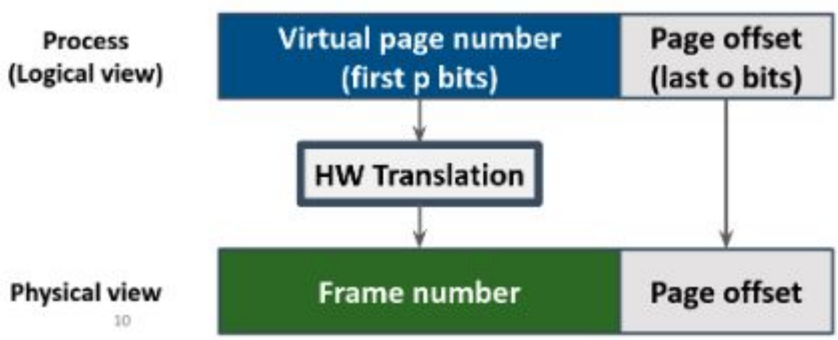
- Translate the virtual address (VA_x) to a physical address (PA_x) using the page table (specifically, MMU inside the CPU does this).
- Check access permissions (e.g., is the page writable?).
- Issue a memory write operation to store the new value at PA_x.
- If copy-on-write (COW) applies, allocate a new page (PA_x'), copy the old contents, update the page table for the process, and write to the new page.

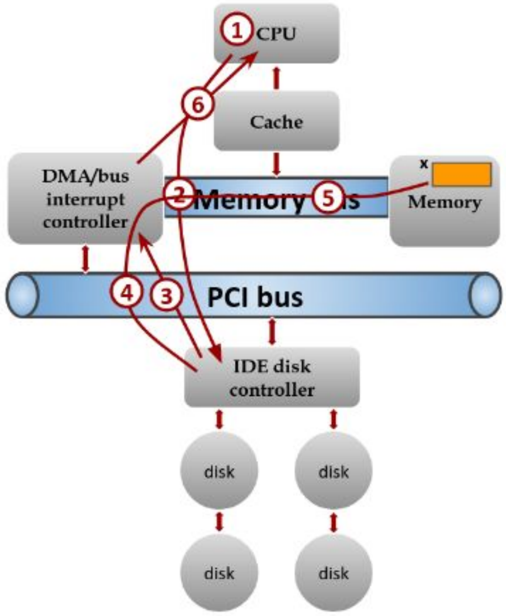
```
size_t vector_push(vector* v, type_el val) {
  if (v != NULL) {
    while (v->size >= v->allocated) {
      if (vector_enlarge(v) == NULL) {
        return 0;
      }
    }
    v->content[v->size] = val;
    ++(v->size);
    return v->size;
  }
  return 0;
}
```

- Suppose process P1 is running, then the timer interrupt occurs and the OS scheduler picks P2 to run.
- What information must be updated concerning memory accesses?
  - Update the Page Table Base Register (i.e., register cr3) to point to P2's page table. (The address of this page table is stored in a per process task struct in linux for P2).
  - Update CPU registers to restore P2's execution state, including its stack pointer, instruction pointer, and general-purpose registers.

```
vector* vector_enlarge(vector* v) {
  if (v != NULL) {
    vector result = *v;
    result.allocated += VECTOR_PADDING;
    if ((result.allocated > SIZE_MAX / sizeof(type_el)) ||
        ((result.content = realloc(result.content,
                              result.allocated * sizeof(type_el)))
         == NULL)) {
      return NULL; /* retourne NULL en cas d'échec ;
                    * v n'a pas été modifié.          */
    }
    // affectation finale, tout d'un coup
    *v = result;
  }
  return v;
}
```







- When a process calls "fopen," the fopen function makes a syscall, which causes the CPU to execute a syscall handler. If the syscall is file-related (open, read, write, close...), the syscall handler calls into the FS. So: the FS is touched because a file-related syscall is made.
- When the FS handles a file-related syscall, there are three cases:
    - The FS does not need to access any data or metadata that is stored on the disk. E.g., if the syscall is "lseek"; of if the syscall is "close," and the file was opened in read-only mode. In this case, the FS does not touch the block device interface (it does not call into the device driver to read from or write to the disk).
    - The FS does need to access data and/or metadata that is stored on the disk. E.g., if the syscall is "open," or "read," or "write." In this case, there are two sub-cases:
        - The FS needs to access a block that has been cached in the FS/block cache. In this sub-case, the FS does not touch the block device interface.
        - The FS needs to access a block that has *not* been cached. In this sub-case, the FS *does* touch the block device interface.

In summary: The FS touches the block device interface if and only if it needs to read from or write to the device.

Solutions

1. a. CPU tells device driver to transfer disk data to buffer at address X
2. e. Device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. f. Disk controller initiates DMA transfer
4. d. Disk controller sends each byte to DMA controller
5. c. DMA controller transfers bytes to buffer X, increasing memory address and decreasing until C = 0
6. b. When C = 0, DMA interrupts CPU to signal transfer completion

est-ce que le polling est parfois mieux que l'interrupt ? oui, par exemple si il y a énormément de paquets qui arrivent en même temps, par exemple plus vite que interrupt handling + context switching. **livelock** -> on ne traite plus rien
donc les systèmes réels utilisent les deux
on peut aussi utiliser du delay and batch pour éviter beaucoup de contexts switch (mais augmente la latence)

FSCK (file system checker)

Périodiquement, après un crash ou un certains nombres d'opérations, cet outil va vérifier la consistence du système.

Par exemple, le FSCK peut corriger le nombre de liens pointant vers un inode (quand on créé un lien symbolique ou un hard link).

Le FSCK peut déplacer TODO lost+found

Si deux inodes pointent vers le même block de données (ça ne devrait pas arriver), il peut copier le data block vers un nouveau et modifier le lien.

Problèmes :

- c'est lent
- c'est pas toujours correct (il essaye de retrouver un état consistent, mais est-ce que c'est le **bon** état ? consistency $\neq$ correctness)

0	1	2	3	4	5	6	7
	[... (inode 1: Location: 12 ...), (inode 2: Location: 14 ...)]	[....]	[... (inode 8: Location: 7 ...), ...]		"This block stores a text file ... "	"00101010 010101010 010101110 01011..."	"its.me": inode 4 "world.txt": inode 2 "baz": inode 12
8	9	10	11	12	13	14	15
	"etc": inode 4 "pwd": inode 5	"This block stores another text file ... "		"foo": inode 3 "bar": inode 7 "hello": inode 8		"!\nabcdefg ..."	

- Block 1, to read inode 1 and find out where the directory "/" is stored.
- Block 12, to look up the inode number for "/hello".
- Block 3, to read inode 8 and find out where the directory "/hello" is stored.
- Block 7, to lookup the inode number for "/hello/world.txt"
- Block 1, to read inode 2 and find out where the file "/hello/world.txt" is stored.
- Block 14, to read the first byte, which is "!".

Journaling (logs)

Objectifs :

- limiter le travail à faire to recover
- obtenir l'état **correct** et non plus seulement l'état consistent
- être plus rapide (plus besoin de scanner tout le disque)

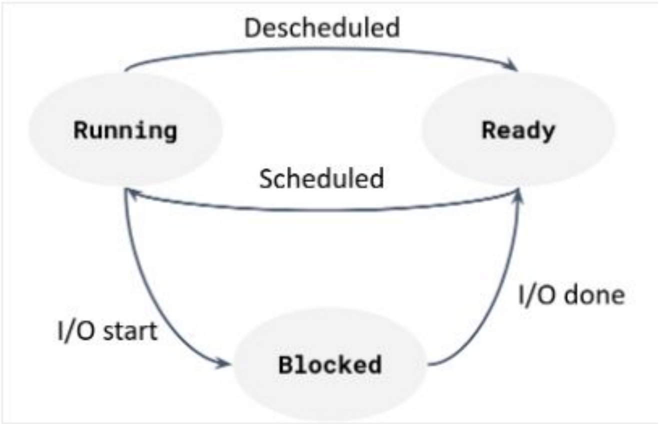
→ écrire dans le journal **avant** d'écrire dans le disque (**write-ahead** logs)
et écrire ce qu'il y avait avant d'overwrite le contenu d'un fichier

Transactions

Propriétés :

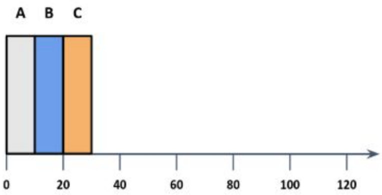
- **atomic** : soit tout fonctionne, soit rien
- **consistent** : amène toujours à un état correct
- **isolée** : opérations n'interagissent pas être elles
- **durable** : une fois qu'elle est complétée, les effets sont persistents

- 파일 조작: `open`, `read`, `write`, `close`, `lseek`
- 프로세스 관리: `fork`, `execve`, `exit`, `wait4`
- 메모리 관리: `mmap`, `mprotect`, `munmap`, `brk`
- 시스템 정보: `uname`, `getpid`, `getuid`
- 통신: `pipe`, `socket`, `connect`, `accept`



FIFO First in, first out

On a trois threads A, B, C qui prennent **chacun** 10 secondes. Ils arrivent à peu près en même temps.



ça marche bien quand on sait le temps que va mettre chaque thread, ce qui n'est généralement pas le cas.

$T_{\text{arrival}} = 0$
$T_{\text{completion A}} = 10$
$T_{\text{completion B}} = 20$
$T_{\text{completion C}} = 30$
average turnaround time is 20.

mais que se passe-t-il si A prend 100 secondes ?

Shorter time to completion first (STCF)

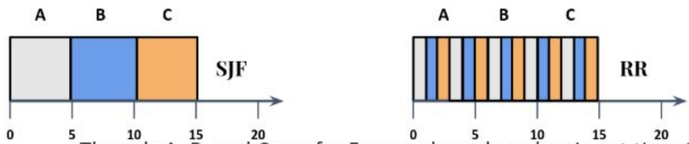
Il étend le shortest job first. à chaque fois qu'un thread est créé :
- il détermine lequel des jobs restants (**dont** celui en cours) a le temps restant le plus faible
- il le schedule

New metric : le temps de réponse

C'est le temps avant que le thread soit scheduled. Les utilisateurs veulent des réponses intéractives !
Ce n'est pas du tout pris en compte dans le STCF

Round Robin (RR)

Au lieu de faire tourner les threads jusqu'à ce qu'ils soient complétés, RR schedule un thread pour un intervalle fixe, et switch au prochain thread.



Threads A, B, and C run for 5 seconds each and arrive at time 0

Le temps de turnaround augmente.

Shortest job first (SJF)

On va choisir le thread le plus rapide à exécuter. Le turnaround baisse beaucoup! (approx. 50)

Mais qu'est-ce qu'il se passe si A arrive à t=0 et doit tourner pendant 100 secondes puis B et C arrive à t=10 et tournent pendant 10 secondes ? A est schedulé et on n'a pas prévu de l'arrêter !

Le tournaround est de approx 103.

Convoy effect : un certain nombre de clients potentiellement rapides se retrouvent derrière un client très long.

polite vs forced scheduling

- FIFO et SJF sont des **non-preemptive**. Ils ne switch que lorsque le thread en cours a fini son exécution.
- **Preemptive** schedulers arrêtent l'exécution du thread en cours et switch à un autre de façon forcée pour éviter que le CPU soit monopolisé.

Deux metrics utiles :
- **utilization** : quelle fraction du temps le CPU passe à exécuter un thread. Objectif : maximiser ce temps.
- **turnaround time** : le temps total que les threads mettent à compléter leur tâche. Objectif : minimiser ce temps. $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$.

TLDR

`const type * ptr` objet constant
`type * const ptr` pointeur constant

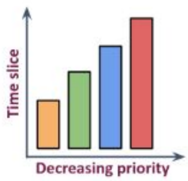
Déclaration	Pointeur externe constant ?	Pointeur intermédiaire constant ?	Objet final constant ?
`int **ptr`	❌	❌	❌
`const int **ptr`	❌	❌	✅
`int * const *ptr`	❌	✅	❌
`const int * const *ptr`	❌	✅	✅
`int ** const ptr`	✅	❌	❌
`const int ** const ptr`	✅	❌	✅
`int * const * const ptr`	✅	✅	❌
`const int * const * const ptr`	✅	✅	✅

Multi-level feedback queue (MLFQ)

Challenge : le scheduler doit pouvoir supporter de longues tâches dans le background (batch processing) et donner une réponse rapide pour les process interactifs.

Batch-based thread : le temps de réponse n'est pas important (on veut minimiser les context switch)
Interactive thread : le temps de réponse est critique, c'est des bursts courts

Pour cela, MLFQ utilise les past behaviors pour prédire les comportements futurs.



- Les threads de haut niveaux ont un temps de run courts, ceux plus bas un temps plus long.
- Les threads de haut niveaux vont toujours être traités en premier

Règles :
- si priority(A) > priority(B) alors A tourne
- si les deux sont égales, alors on fait du RR (et on change l'intervalle en fonction du niveau)
- les threads commencent tous à une haute priorité (on ne sait pas combien de temps ils vont tourner)
- puis, quand il utilise tout l'intervalle, le scheduler descend sa priorité
- périodiquement, tous les threads sont déplacés dans la topmost queue (priority boosting) pour éviter que les threads interactifs empêchent les threads du bas de ne jamais tourner (**starvation**)

```
typedef void (*policy_function)(set_t *set, size_t line, size_t index);
```

Creating a File (e.g., `open("/cs202/w07", O_CREAT | O_RDWR)`)

1. **Root Inode (read)**:
  - The root directory's inode is read to locate its data block pointers.

2. **Root Data (read)**:
  - Reads root directory data to find the entry for `cs202`.

3. **`cs202` Inode (read)**:
  - The inode for directory `cs202` is read to locate its data blocks and metadata.

4. **`cs202` Data (read)**:
  - Reads the directory data for `cs202` to check if the file `w07` already exists. Since we are creating `w07`, the system checks to ensure it doesn't already exist or identifies an available inode for use.

5. **Inode Bitmap (read/write)**:
  - The system reads the inode bitmap, which tracks inode availability.
  - It finds a free inode for the new file `w07` and marks this inode as allocated by writing back to the inode bitmap.

6. **Writing to the `cs202` Data**:
  - The directory data of `cs202` is updated to include a new entry for `w07`, linking it to the newly allocated inode. This ensures that `w07` is now part of the `cs202` directory

7. **w07 Inode (read + write)**:
- The new inode for `w07` (previously allocated) must now be **initialized**:
- Set owner, permissions, size (probably zero initially), data block pointers, timestamps, etc.
- So a **write()** happens to the **w07 inode**.

8. **cs202 Inode (write)**:
- last modification (creating a file)

9. **w07 Inode (read)**:
- Now that `w07` exists and has its inode allocated, **read()** the inode to prepare for writing actual file content.

10. **Data Bitmap (read + write)**:
  - **read()** the **data bitmap** to find a free **data block** to store the file contents for `w07`.
  - **write()** back to the data bitmap to **mark** that block as **allocated**.(Just like with the inode bitmap, but for **data blocks** now.)

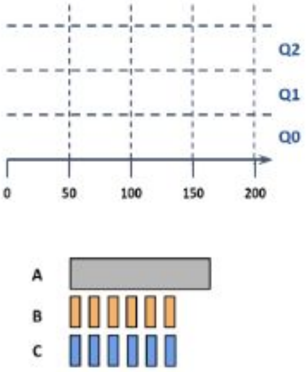
11. **w07 Data[0] (write)**:
- Finally, the **first actual write** to the data block(s) of the file `w07`:
  - Saving the file's first piece of content.
  - This is the **first data block** of the file.

12. **w07 inode** :
modify last access date, file size...

8:56 AM

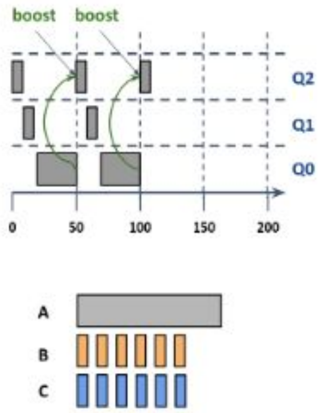
MLFQ example

- 3 priority queues (Q2 highest and Q0 lowest)
- Periodic boosting window: 50 ms (Rule 5)
- Time-slice: 10 ms
- 3 threads
  - A is long running thread
  - B and C are interactive threads issuing IO
- $T_{arrival(A)} = 0$
- $T_{arrival(B)} = T_{arrival(C)} = 100$



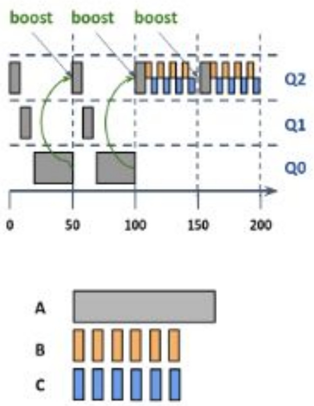
MLFQ example

- A runs for 10 ms in Q2 and gets demoted to Q1
- A then runs for 10 ms in Q1 and demoted to Q0
- A runs for 30 ms and gets **boosted** to Q2 (**R 5**)
- The same procedure happens until 100 ms



MLFQ example

- A runs for 10 ms in Q2 and gets demoted to Q1
- A then runs for 10 ms in Q1 and demoted to Q0
- A runs for 30 ms and gets **boosted** to Q2 (**R 5**)
- The same procedure happens until 100 ms
- Process B and C also join Q2
- A scheduled for 10 ms
- B is scheduled and then followed by C
  that are issuing IO requests as well



MLFQ does not starve long running jobs and gives equal time to all jobs

	data bitmap	inode bitmap	root inode	cs202 inode	w07 inode	root data	cs02 data	w07 data[0]
open(“cs202/w07”)			read()					
						read()		
				read()				
							read()	
		read()						
		write()						
							write()	
					read()			
					write()			
				write()				
write()					read()			
	read()							
	write()							
								write()
					write()			

inter-arrival time at B = amount of time that elapses from the moment the last bit of the first packet arrives until the moment the last bit of the second packet arrives

Tahoe vs Reno (timeout)

Événement	Tahoe	Reno
Timeout	cwnd → 1 MSS ssthresh → cwnd/2 state → slow start	même que Tahoe
3 duplicate ACKs	cwnd → 1 MSS ssthresh → cwnd/2 state → slow start	dès que les 3 acks dupliqués sont reçus: - sshtresh = cwnd/2 - cwnd = sstresh (+ 3 MSS) - on retransmet le paquet manque et on passe en **fast recovery** - on reste en **fast recovery** jusqu'à ce qu'on reçoive un nouvel ACK - pour tout ACK dupliqués reçus entre temps (en **fast recovery**), on augmente la window de 1 MSS - dès qu'on reçoit un nouvel ACK on reset la window à sstresh

sstresh : slow start threshold

Quand on a un timeout, on fait donc:

- sstresh = $\frac{\text{cwnd}}{2}$ puis `cwnd` = 1
- on reste en slow start
- quand on atteindra `sstresh` , on passera automatiquement en congestion avoidance

States

- **slow start** : augmenter la window de maniere agressive
  - aumente de 1 MSS par ACK. à chaque ACK :

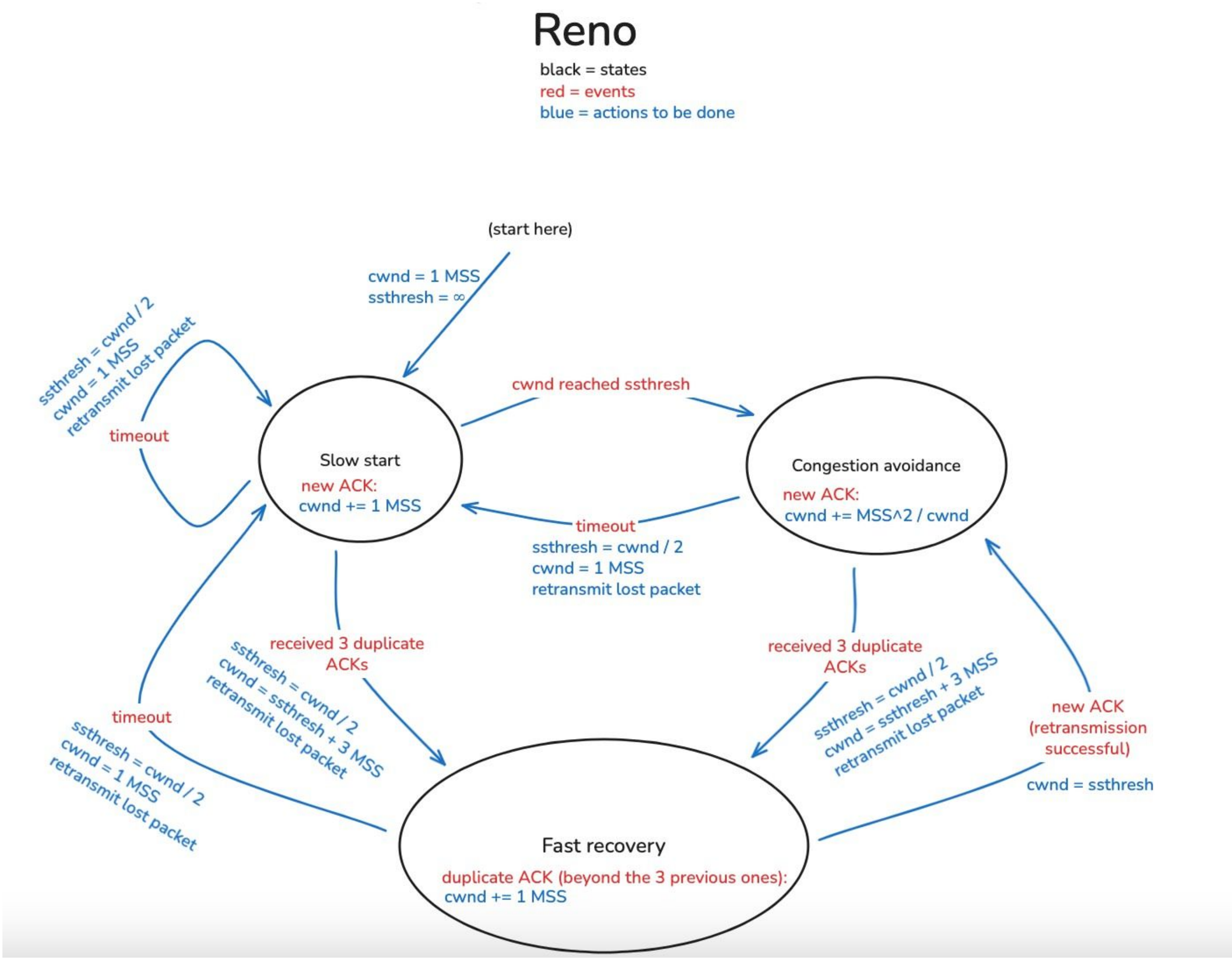
$$\text{cwnd} = \text{cwnd}_{t-1} + \text{MSS}$$

- **congestion avoidance** : augmenter la `cwnd` précautionneusement
  - augmente de 1 MSS par RTT. à chaque ACK :

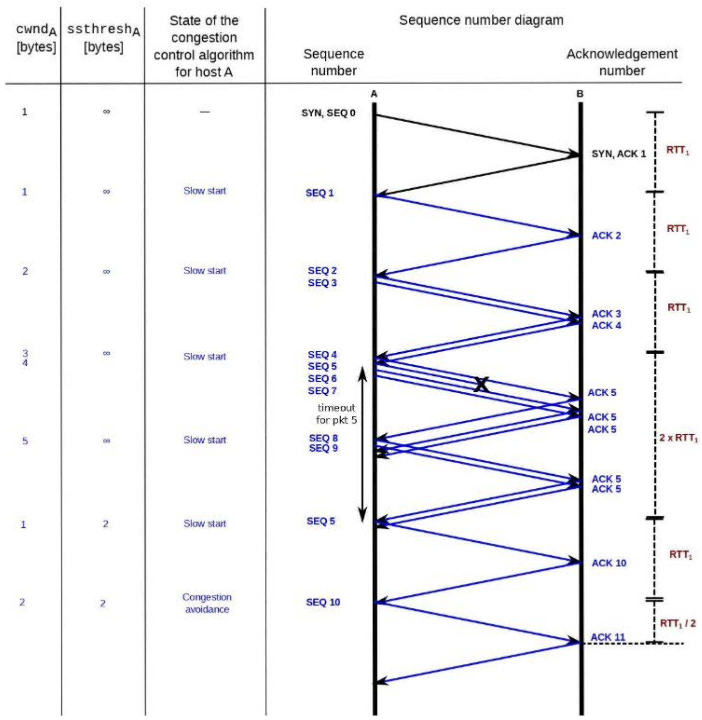
$$\text{cwnd} = \frac{\text{MSS}^2}{\text{cwnd}_{t-1}}$$

  - parce que :
    - on envoie cwnd bytes par RTT, soit $\frac{\text{cwnd}}{\text{MSS}}$ segments par RTT
    - on va donc augmenter, par RTT

$$\frac{\text{cwnd}}{\text{MSS}} \cdot \frac{\text{MSS}^2}{\text{cwnd}_{t-1}} \approx \text{MSS}$$



- Sender window = min{receiver window, congestion window



Timeout calculation

- EstimatedRTT = 0.875 EstimatedRTT + 0.125 SampleRTT
- DevRTT = function (RTT variance)
- Timeout = EstimatedRTT + 4 DevRTT
- Empirical, conservative RTT estimation

```c
#include <stdio.h>
#include "socket_layer.h"
#include <netinet/in.h>
#include <strings.h>

int main(int argc, char *argv[])
{

    int socket = udp_server_init(CS202_DEFAULT_IP, CS202_DEFAULT_PORT, 0);
    if (socket < 0) {
        fprintf(stderr, "Error creating socket\n");
        return 1;
    }
    struct sockaddr_in cli_addr;
    char buf[sizeof(unsigned int)];

    printf("Server listening on %s:%u\n", CS202_DEFAULT_IP, CS202_DEFAULT_PORT);

    while(udp_read(socket, buf, sizeof(buf), &cli_addr)) {
        printf("Received value: %u\n", *buf);

        // send back a 1
        char ack[sizeof(char)];
        char value = 1;
        ack[0] = value;
        udp_send(socket, ack, sizeof(ack), &cli_addr);
        printf("Sent acknowledgment: %u\n", value);
    }
}
```

Question 2 (9 points):

All link-layer switches have just been rebooted, and all end-system caches are initially empty. Then, the user of workstation E_1 visits web page `www.yyy.ch/index.html`, which contains no embedded objects (e.g., no images).

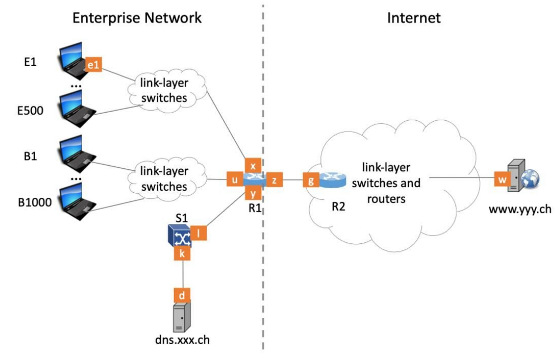
State all the packets that are **received, forwarded, or transmitted by router R_1 until E_1's user can view the web page**. For example, if a packet follows the path $E_1 \rightarrow R_1 \rightarrow$ ...`www.yyy.ch`, then you should state it 2 times: when it is received by R_1, and when it is forwarded by R_1.

Answer by filling in Table 2. To denote the IP address or the MAC address of interface s, write "s". If a field is not applicable, write "–". To repeat a field from the above cell, write "." To illustrate the format, we have provided a hypothetical example entry.

#	Source MAC	Dest MAC	Source IP	Dst IP	Transp. prot.	Src Port	Dst Port	Application & Purpose
1	e_1	broadcast	-	-	-	-	-	ARP request for x's MAC
2	x	e_1	-	-	-	-	-	ARP reply
3	e_1	x	e_1	d	UDP	2000	53	DNS request for w's IP
4	y	broadcast	-	-	-	-	-	ARP request for d's MAC
5	d	y	-	-	-	-	-	ARP reply
6	y	d	e_1	d	UDP	2000	53	DNS request for w's IP
7	d	y	d	root	UDP	2500	53	DNS request for w's IP
8	z	broadcast	-	-	-	-	-	ARP request for g's MAC
9	g	z	-	-	-	-	-	ARP response
10	z	g	d	root	UDP	2500	53	DNS request for w's IP
11	g	z	root	d	UDP	53	2500	DNS response
12	y	d	root	d	UDP	53	2500	DNS response
13	d	y	d	e_1	UDP	53	2000	DNS response
14	x	e_1	d	e_1	UDP	53	2000	DNS response
15	e_1	x	e_1	w	TCP	3000	80	TCP SYN
16	z	g	e_1	w	TCP	3000	80	TCP SYN
17	g	z	w	e_1	TCP	80	3000	TCP SYN ACK
18	x	e_1	w	e_1	TCP	80	3000	TCP SYN ACK
19	e_1	x	e_1	w	TCP	3000	80	HTTP GET index
20	z	g	e_1	w	TCP	3000	80	HTTP GET index
21	g	z	w	e_1	TCP	80	3000	HTTP OK
22	x	e_1	w	e_1	TCP	80	3000	HTTP OK

Table 2: Packets received, forwarded, or transmitted by router R_1 in Question 2.

You can find a copy of this network topology at the end of the exam. You can detach it so that you can look at the topology while solving the problem, without having to turn the pages back and forth.




```c
// ============================================================================
int udp_server_init(const char *ip, uint16_t port, time_t t)
{
    M_REQUIRE_NON_NULL(ip);

    // Create a socket
    int socket = get_socket(t);
    if (socket < 0) {
        return ERR_NETWORK;
    }

    // Bind the socket to the address
    int err = bind_server(socket, ip, port);
    if (err != ERR_NONE) {
        close(socket);
        return err;
    }

    return socket;
}
```

```c
/**************************************************
ssize_t udp_read(int socket, char *buf, size_t bu
                 struct sockaddr_in *cli_addr)
{
    M_REQUIRE_NON_NULL(buf);
    M_REQUIRE_NON_NULL(cli_addr);

    // receive data from the socket
    socklen_t addr_len = sizeof(*cli_addr);

    ssize_t res = recvfrom(
        socket, // the socket to read from
        buf, // the buffer to read into
        buflen, // the size of the buffer
        0, // flags (0 means no flags)
        (struct sockaddr *) cli_addr, // the address of the client
        &addr_len // the size of the address structure
    );

    return res < 0 ? ERR_NETWORK : res;
}
```

```c
// ============================================================================
int get_socket(time_t t)
{

    const int socketfd = socket(
        AF_INET, // use IP v4
        SOCK_DGRAM, // socket UDP
        0
    );

    if (socketfd < 0) {
        return ERR_NETWORK;
    }

    if (t > 0) {
        // Set receive timeout
        struct timeval timeout;
        zero_init_var(timeout);

        // this is the timeout in seconds
        timeout.tv_sec = t;
        timeout.tv_usec = 0;
        if (setsockopt(
            socketfd, // which socket to set the option to
            SOL_SOCKET, // which level/layer to set the option to (Socket Option Level)
            SO_RCVTIMEO, // the name of the value to set (the timeout)
            &timeout, // the value to set
            sizeof(timeout) // length of the value
        ) < 0
        ) {
            close(socketfd);
            return ERR_NETWORK;
        }
    }

    return socketfd;
}
```

```c
// ============================================================================
int get_server_addr(const char *ip, uint16_t port,
                    struct sockaddr_in *p_server_addr)
{
    M_REQUIRE_NON_NULL(ip);
    M_REQUIRE_NON_NULL(p_server_addr);

    struct sockaddr_in server_addr;
    zero_init_var(server_addr);

    server_addr.sin_family = AF_INET; // IPv4
    server_addr.sin_port = htons(port); // port in network byte order (interne
big endian)
    int success = inet_pton(AF_INET, ip, &server_addr.sin_addr.s_addr); // co
string to binary form
    if (success <= 0) {
        return ERR_ADDRESS;
    }

    *p_server_addr = server_addr;

    return ERR_NONE;
}
```

```c
// ============================================================================
int bind_server(int socket, const char *ip, uint16_t port)
{
    struct sockaddr_in server_addr;
    zero_init_var(server_addr);

    int err = ERR_NONE;

    err = get_server_addr(ip, port, &server_addr);
    if (err != ERR_NONE) {
        return err;
    }

    if (
        bind(
            socket, // the socket to bind
            (struct sockaddr *) &server_addr, // in the handout it's said: you can safely cast a struct
sockaddr_in* to a struct sockaddr*, and vice versa
            sizeof(server_addr) // the size of the address structure
        ) < 0
    ) {
        return ERR_NETWORK;
    }

    return err;
}
```

```c
ssize_t udp_send(int socket, const char *response, size_t response_len
                 const struct sockaddr_in *cli_addr)
{
    M_REQUIRE_NON_NULL(response);
    M_REQUIRE_NON_NULL(cli_addr);

    socklen_t addr_len = sizeof(*cli_addr);

    // send data to the socket
    ssize_t res = sendto(
        socket, // the socket to send to
        response, // the buffer to send
        response_len, // the size of the buffer
        0, // flags (0 means no flags)
        (const struct sockaddr *) cli_addr, // the address of the cli
        addr_len // the size of the address structure
    );

    return res < 0 ? ERR_NETWORK : res;
}
```

```c
int main(int argc, char *argv[])
{

    unsigned int value;
    printf("What unsigned int value do you want to send? ");
    scanf("%u", &value);

    // Create a socket
    int socket = get_socket(0);
    if (socket < 0) {
        fprintf(stderr, "Error creating socket\n");
        return 1;
    }

    struct sockaddr_in p_server_addr;
    get_server_addr(CS202_DEFAULT_IP, CS202_DEFAULT_PORT, &p_server_addr);
    udp_send(socket, (const char *)&value, sizeof(value), &p_server_addr);
    printf("Sent value: %u\n", value);

    // wait for acknowledgment
    udp_read(socket, (char *)&value, sizeof(value), &p_server_addr);
    if (value == 1) {
        printf("Received acknowledgment: %u\n", value);
    } else {
        fprintf(stderr, "Error: did not receive acknowledgment\n");
    }

    close(socket);

    return 0;

}
```

```c
#include <stdio.h>
#include <pthread.h>
int counter = 0;

void *incr(void *arg) {
  counter = counter + 1;
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t threads[5];
  // Create two threads T1 and T2
  for (int i=0; i < 5; i++) {
     pthread_create(&threads[i], NULL, incr, NULL);
  }
  for (int i=0; i < 5; i++) {
     pthread_join(threads[i], NULL);
  }
  printf("Counter: %d\n", counter);
  return 0;
}
```

pthread_mutex()

```c
#include<stdio.h>
#include<pthread.h>
pthread_mutex_t m =
        PTHREAD_MUTEX_INITIALIZER;


void critical section(int *counter) {
    pthread_mutex_lock(&m);
    *counter += 1;
    pthread_mutex_unlock(&m);
    return;
}
```

Standard POSIX-compliant API

- **Semantic of a mutex:**
  - **Thread blocks when lock is held**
- **Integration with other synchronisation primitives (e.g., condition variables)**
- **Implementation within the OS**

Note: Linux also supports `futex (fast user-level mutex)`, as a higher-performing alternative to `pthread_mutex` (with a more complex API)

Exercise 5: link-state routing

Consider the network in Figure 4. Execute the link-state (Dijkstra's) algorithm we saw in class to compute the least-cost path from each of *x*, *v*, and *t* to all the other routers.

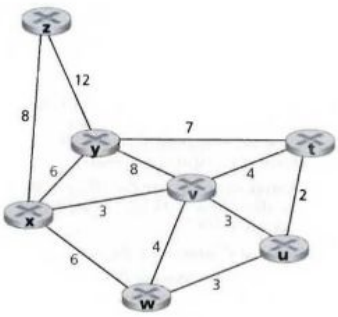


Figure 4: Network topology.

Exercise 6: distance-vector routing

Consider the network in Figure 5. Execute the distance-vector (Bellman-Ford) algorithm we saw in class and show the information that router *z* knows after each iteration.

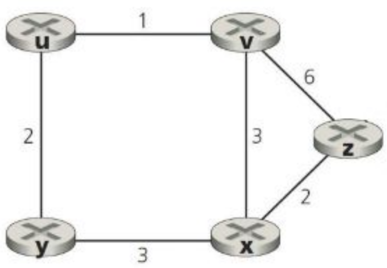


Figure 5: Network topology.

The least cost path from routers x, v, and t to all the other routers is displayed in the next table along with the execution steps of the link-state algorithm.

For each router i, C(i) stands for cost to i, and p(i) stands for predecessor to i.

1. Least-cost path from x to all network nodes.

step	nodes visited	C(t),p(t)	C(u),p(u)	C(v),p(v)	C(w),p(w)	C(y),p(y)	C(z),p(z)
0	x	∞	∞	3,x	6,x	6,x	8,x
1	x,v	7,v	6,v	3,x	6,x	6,x	8,x
2	x,v,u	7,v	6,v	3,x	6,x	6,x	8,x
3	x,v,u,w	7,v	6,v	3,x	6,x	6,x	8,x
4	x,v,u,w,y	7,v	6,v	3,x	6,x	6,x	8,x
5	x,v,u,w,y,t	7,v	6,v	3,x	6,x	6,x	8,x
6	x,v,u,w,y,t,z	7,v	6,v	3,x	6,x	6,x	8,x

2. Least-cost path from v to all network nodes.

step	nodes visited	C(t),p(t)	C(u),p(u)	C(w),p(w)	C(x),p(x)	C(y),p(y)	C(z),p(z)
0	v	4,v	3,v	4,v	3,v	8,v	∞
1	v,x	4,v	3,v	4,v	3,v	8,v	11,x
2	v,x,u	4,v	3,v	4,v	3,v	8,v	11,x
3	v,x,u,t	4,v	3,v	4,v	3,v	8,v	11,x
4	v,x,u,t,w	4,v	3,v	4,v	3,v	8,v	11,x
5	v,x,u,t,w,y	4,v	3,v	4,v	3,v	8,v	11,x
6	v,x,u,t,w,y,z	4,v	3,v	4,v	3,v	8,v	11,x

3. Least-cost path from t to all network nodes.

step	nodes visited	C(u),p(u)	C(v),p(v)	C(w),p(w)	C(x),p(x)	C(y),p(y)	C(z),p(z)
0	t	2,t	4,t	∞	∞	7,t	∞
1	t,u	2,t	4,t	5,u	∞	7,t	∞
2	t,u,v	2,t	4,t	5,u	7,v	7,t	∞
3	t,u,v,w	2,t	4,t	5,u	7,v	7,t	∞
4	t,u,v,w,x	2,t	4,t	5,u	7,v	7,t	15,x
5	t,u,v,w,x,y	2,t	4,t	5,u	7,v	7,t	15,x
6	t,u,v,w,x,y,z	2,t	4,t	5,u	7,v	7,t	15,x

Each node in the topology has its own view of the network, which is updated independently from other nodes at the end of each step. Therefore, for every step of the algorithm, you also need to update each of the other cost tables. Otherwise, your solution may be incorrect.

In our solution we only show the cost table for node z, as required by the question. The cost table at node z consists of 5 columns (all possible destinations) and 3 rows (all possible sources—one row for node z and one row for each neighbor). Each entry of the table denotes the cost between the associated source-destination nodes.

Initially (at step 0), node z has the following view of the network:

		To				
		u	v	x	y	z
From	v	∞	∞	∞	∞	∞
	x	∞	∞	∞	∞	∞
	z	∞	6	2	∞	0

At step 1:

		To				
		u	v	x	y	z
From	v	1	0	3	∞	6
	x	∞	3	0	3	2
	z	7	5	2	5	0

At step 2:

		To				
		u	v	x	y	z
From	v	1	0	3	3	5
	x	4	3	0	3	2
	z	6	5	2	5	0

At step 3:

		To				
		u	v	x	y	z
From	v	1	0	3	3	5
	x	4	3	0	3	2
	z	6	5	2	5	0

We see that from step 2 to step 3 the cost tables did not change, indicating that the algorithm has converged.