3TC31 (ex. INF107)

Examen final

2025–2026

Instructions
— Durée : 1h30
— Document autorisé : 1 feuille A4 manuscrite recto-verso uniquement
— Dispositifs électroniques (calculatrice, ordinateur...) interdits
— Vous pouvez répondre en français ou en anglais
— Ce sujet contient trois parties indépendantes
— Répondez aux différentes questions sur une copie (pas sur le sujet)

Partie 1 (3 points / 15 minutes)

Question 1 (0.5 points)

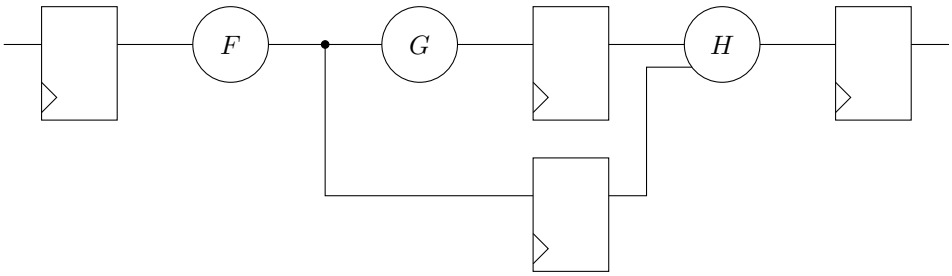
Quelle est la valeur, en décimal, du nombre représenté en complément à 2 sur 5 bits par la valeur 10000 ?

Question 2 (1 point)

À partir d'une bascule D (*D flip-flop*) et de portes logiques élémentaires, dessinez le schéma d'un circuit ayant trois entrées `D`, `clk` et `enable` et une sortie `Q` et le comportement suivant :
— Si `enable` vaut 1, lors d'un front montant de l'horloge `clk`, `Q` prend la valeur de `D`
— Le reste du temps, `Q` conserve sa valeur précédente

Question 3 (1 point)

Soit le circuit suivant :



On suppose que F, G et H sont des fonctions combinatoires et que toutes les bascules fonctionnent sur la même horloge.

On suppose les temps suivants :
— temps de propagation dans F : 4 ns
— temps de propagation dans G : 4 ns
— temps de propagation dans H : 3 ns
— temps de propagation dans les bascules (t_{co}) : 1 ns
— temps de pré-positionnement des bascules (t_{su}) : 1 ns

Quelle est la fréquence maximale de fonctionnement de ce circuit ?

Question 4 (0.5 points)

Soit l'instruction assembleur RISC-V `addi x1, x2, 42`.
— Que fait cette instruction ?
— Où est stockée la valeur 42 ?

Partie 2 (3 points / 15 minutes)

Question 5 (0.5 points)

Soit le programme suivant :

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  int * p;
  *p = 42;
  printf("*p␣=␣%d\n", *p);
  return EXIT_SUCCESS;
}
```

Que produit l'exécution de ce code ?

Question 6 (1 point)

Soit le programme suivant :

```
#include <stdlib.h>
#include <stdio.h>

void swap(int a, int b) {
  int c = a;
  a = b;
  b = c;
}

int main(int argc, char *argv[]) {
  int a = 1;
  int b = 2;
  swap(a, b);
  printf("a=%d␣b=%d\n", a, b);
  return EXIT_SUCCESS;
}
```

— Qu'affiche le programme ci-dessus ?
— Corrigez-le pour qu'il produise le résultat attendu (la fonction appelante doit voir la modification effectuée par la fonction `swap`), sans utiliser de variables globales.

Question 7 (1.5 points)

Soit une liste chaînée dont un nœud est représenté par la structure suivante :

```
struct node {
  int value;
  struct node *next;
};
```

Écrivez le code de la fonction `display` qui prend en unique argument une liste chaînée (à vous de trouver le type correct) et qui affiche la valeur (l'entier `value`) de tous les nœuds présents dans la liste, ou la chaîne `"Liste vide"` si la liste est vide.

Part 3 (14 points / 60 minutes)

Cette partie est composée de 4 exercices indépendants.

Exercice 1 : Entrées/Sorties bas niveau

Question 8 (4 points)

On souhaite implémenter deux fonctions, `read_object_at` et `write_object_at`, permettant respectivement de lire et d'écrire un objet de type `object_t` dans un fichier identifié par son descripteur `fd`.

Chaque objet est stocké séquentiellement dans le fichier. L'argument `index` indique le numéro de l'objet à lire ou à écrire (les objets sont numérotés à partir de zéro).

Les fonctions devront utiliser exclusivement les fonctions systèmes comme `open`, `close` etc. et *non pas* les fonctions de la bibliothèque standard C comme `fopen`, `fclose`, etc.

En cas d'erreurs, le processus doit s'arrêter.

```
/*****************************************************/

typedef struct {
  int id;
  char name[32];
  double value;
} object_t;

void read_object_at(int fd, size_t index, object_t *obj) {

  // TODO / À faire

}

void write_object_at(int fd, size_t index, object_t *obj) {

  // TODO / À faire

}
/*****************************************************/
```

Exercice 2 : Processus et parallélisme

Nous considérons un fichier comme un tableau d'octets non-signés. On suppose l'existence d'une fonction `partition_file` qui, à partir d'une partition entre l'index `first` et l'index `last` du fichier identifié par son descripteur `fd`, retourne un index `pivot`. Elle crée ainsi deux sous-partitions $[first, pivot]$ (gauche) et $[pivot + 1, last]$ (droite).

```
off_t partition_file(int fd, off_t start, off_t end);
```

Question 9 (2 points)

Dans la fonction `process_partition` (voir code juste après), créez *deux* processus qui se chargent de poursuivre le partitionnement respectivement des sous-partitions gauche et droite.

Question 10 (1 point)

Faites en sorte qu'après avoir créé les deux processus, le processus qui les a créés attende leur terminaison.

```
/*****************************************************/

void process_partition(int fd, off_t first, off_t last) {
    if (last - first <= 2)
        return;

    off_t pivot = partition_file(fd, start, end);

    // TODO / À faire (pour les deux questions)

}
```

```
int main(int argc, char *argv[]) {
    if (argc < 4) {
        fprintf(stderr, "Usage: %s fichier first last\n", argv[0]);
        exit(1);
    }

    char *filename = argv[1];
    off_t first = atoi(argv[2]);
    off_t last = atoi(argv[3]);

    int fd = open(filename, O_RDWR);
    if (fd < 0) { perror("open"); exit(1); }

    process_partition(fd, first, last);

    close(fd);
    return 0;
}
/*******************************************************/
```

Exercice 3 : Ordonnancement

Question 11 (3 points)

Considérez l'ensemble suivant de processus, avec la durée de l'intervalle CPU (*burst*) donnée en millisecondes :

Processus	***Burst***	**Priorité**
P_1	5	4 (plus basse)
P_2	3	1 (plus haute)
P_3	1	2
P_4	8	2
P_5	4	3

L'ordre d'arrivée est P_1, P_2, P_3, P_4, P_5, tous à l'instant 0 (environ). Considérons les algorithmes de planification : premier arrivé, premier servi (*first-come first-served*, FCFS), plus court d'abord (*shortest job first*, SJF), et tourniquet (*round robin*, RR) avec $q = 3$ ms. Quels sont les temps d'attente de chaque processus dans chaque cas ? (Rappel : le temps d'attente d'un processus est la quantité de temps totale passée en attente dans la file d'attente prête.) Recopiez le tableau suivant sur votre copie avec vos réponses :

	Temps d'attente (ms)				
Algorithme	P1	P2	P3	P4	P5
FCFS					
SJF					
RR, q=3					

Exercice 4 : Synchronisation

Question 12 (2 points)

Deux threads, A et B, s'amusent en exécutant respectivement les bouts de pseudo code suivants :

```
have_fun_A() {  // main function of thread A
    before_party();
    barrier_A();  // synchronization barrier
    after_party();
}

have_fun_B() {  // main function of thread B
    before_party();
    barrier_B();  // synchronization barrier
    after_party();
}
```

Ils souhaitent faire en sorte que seulement une fois que tous les deux sont arrivés à la "barrière" de synchronisation, le calcul continue. Par conséquence, aucun thread doit appeler `after_party()` avant que tous les deux aient terminé d'exécuter `before_party()`.

En utilisant les primitives de synchronisation vues en cours (et notamment : mutex lock et/ou sémaphores), donnez les implémentations en pseudo code des fonctions `barrier_A()` et `barrier_B()` qui implémentent correctement le comportement attendu. La fonction `init()`, dont l'implémentation est à fournir, sera exécutée avant la création des threads et peut être utilisée pour initialiser les outils de synchronisation utilisés par `barrier()`.

Vous trouverez en annexe les signatures des fonctions de manipulation des mutex et sémaphores.

```
init() {

  // TODO / À faire

}

barrier_A() {

  // TODO / À faire

}

barrier_B() {

  // TODO / À faire

}
```

Question 13 (2 points)

Nous souhaitons généraliser la solution au problème précèdent pour permettre à $N > 1$ threads de se synchroniser de la même manière. Chaque thread exécute maintenant le *même* pseudo code suivant :

```
have_fun() {  // main function of each of the N threads
    before_party()
    barrier()  // synchronization barrier
    after_party()
}
```

Le comportement attendu est maintenant le suivant : seulement une fois que *tous* les threads sont arrivés à la barrière de synchronisation (fonction `barrier`), le calcul peut continuer. Dit autrement : aucun thread doit appeler `after_party()` avant que *tous* les threads aient terminé l'exécution de `before_party()`.

Donnez les implémentations en pseudo code des fonctions `barrier()` et `init()` qui respectent ce comportement.

```
init() {

  // TODO / À faire

}

barrier() {

  // TODO / À faire

}
```

Annexe : signature des fonctions usuelles en C

- `void *memset(void *ptr, int value, size_t num);`
  Fills the first `num` bytes of the memory area pointed to by `ptr`, returns `ptr`.
- `int memcmp(const void *ptr1, const void *ptr2, size_t num);`
  Compares `num` bytes of the memory areas `prt1` and `ptr2`; returns `0` when all the bytes are the same.

— `void *memcpy(void *dst, const void *src, size_t num);`
Copies `num` bytes from memory area `src` to memory area `dst` ; returns `dst`.

— `char *strncpy(char *dst, const char *src, size_t sz);`
Copies up to `sz` characters from the string `src` to `dst`, stopping at the null character ; returns `dst`.
— `size_t strlen(const char *s);`
Calculates the length of the string pointed to by `s`.

— `void *malloc(size_t size);`
Allocates `size` bytes on the heap and returns a pointer to the allocated memory.
— `void *realloc(void *ptr, size_t size);`
Changes the size of the memory block pointed to by `ptr` to `size` bytes. If `ptr` is `NULL` behaves like `malloc`. May invalidate the pointer `ptr` and, in all cases, returns a pointer to the *reallocated* memory region. The **content** of the memory area originally pointed to by `ptr` is **preserved**.
— `void free(void *ptr);`
Frees the memory space pointed to by `ptr`, which has to be allocated by `malloc` or `realloc` beforehand.

— `void perror(const char *s);`
Produces a message on `stderr` describing the last error encountered during a call to a C library function.
— `void exit(int status);`
Causes normal process termination with an exit code `status`.

— `ssize_t read(int fildes, void *buf, size_t nbyte);`
Attempt to read `nbyte` bytes from the file associated with the file descriptor `fildes` into the buffer pointed to by `buf` ; returns the number of bytes actually read, `0` when the end-of-file (EOF) was reached, or `-1` in case of an error.
— `ssize_t write(int fildes, const void *buf, size_t nbyte);`
Attempt to write `nbyte` bytes from the buffer pointed to by `buf` to the file associated with the file descriptor `fildes` ; returns the number of bytes actually written or -1 in case of an error.
— `int pipe(int fildes[2]);`
Create a pipe and place two file descriptors into `fildes[0]` and `fildes[1]` ; return `0` on success or `-1` on error.
— `int open(const char *path, int oflag, ...);`
The file name specified by `path` is opened for reading and/or writing, as specified by the argument `oflag` ; the file descriptor is returned to the calling process.
— `int close(int fd);`
The close() call deletes the descriptor `fd` from the per-process object reference table.
— `off_t lseek(int fd, off_t offset, int whence);`
The lseek() function repositions the offset of the file descriptor `fd` to the argument `offset`, according to the directive `whence`. Upon successful completion, lseek() returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of `-1` is returned. lseek() repositions the file pointer as follows :
  — If `whence` is SEEK_SET, the offset is set to `offset` bytes.
  — If `whence` is SEEK_CUR, the offset is set to its current location plus `offset` bytes.
  — If `whence` is SEEK_END, the offset is set to the size of the file plus `offset` bytes.

— `pid_t fork(void);`
Create a new process by copying the current process ; returns `0` in the child process ; returns the process id of the child process or `-1` in case of an error in the parent process.
— `pid_t wait(int *status);`
Blocks the current process until one of its child processes terminates ; returns the process id of the child or `-1` in case of an error. Also returns the exit status of the child via the pointer `status`. If `status` is `NULL` no exit code is returned.
— `int execvp(const char *file, char *const argv[]);`
Replace the current process image with a new process image loaded from the executable file with name `file`, passing the table `argv` as command-line arguments to the new process image ; returns `-1` in case of an error.

Annexe : signature en pseudo-code des fonctions de synchronisation

```
/* functions available on a semaphore sem */
```

```
init_sem(semaphore sem, unsigned int n);
wait(semaphore sem);
signal(semaphore sem);

/* functions available on a mutex mtx */
init_mtx(mutex mtx);  // initially unlocked
lock(mutex mtx);
release(mutex mtx);
```