Bases de l'Architecture et des Systèmes

Programme, processeur, processus

Équipe pédagogique Système, Univ. Toulouse

Référence : *Operating System Concepts Tenth Edition*
par Avi Silberschatz, Peter Baer Galvin et Greg Gagne

- Livre : John Wiley & Sons, Inc. ISBN 978-1-118-06333-0
- Slides : `https://codex.cs.yale.edu/avi/os-book/OS10/slide-dir/index.html`

Objectif de ce cours

Un système d'exploitation (SE) a de nombreux rôles.

- Gérer et protéger le matériel d'un ordinateur
- Fournir des interfaces pratiques pour l'utilisateur
- Se servir efficacement du matériel

Objectif de ce cours

Un système d'exploitation (SE) a de nombreux rôles.

- Gérer et protéger le matériel d'un ordinateur
- Fournir des interfaces pratiques pour l'utilisateur
- Se servir efficacement du matériel

Ce cours est centré sur l'exécution de programmes sur un ordinateur et répond aux questions suivantes.

- Qu'est-ce qu'un programme ?
- Comment le SE gère la mémoire et le processeur ?
- Quelles interfaces sont fournies par le SE pour contrôler l'exécution d'un programme ?

Qu'est-ce qu'un programme ?

Qu'est-ce qu'un programme ?

```c
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Vision conception/développement

- code source
- algorithmes
- composants logiciels

Qu'est-ce qu'un programme ?

```c
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Vision conception/développement

- code source
- algorithmes
- composants logiciels

Vision système/architecture

Qu'est-ce qu'un programme ?

```c
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Vision conception/développement

- code source
- algorithmes
- composants logiciels

Problème : un processeur ne comprend aucun de ces langages !

Vision système/architecture

Qu'est-ce qu'un programme ?

```
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Vision conception/développement

- code source
- algorithmes
- composants logiciels

Problème : un processeur ne comprend aucun de ces langages !

Jetons un œil au fonctionnement d'un ordinateur pour éclaircir ça...

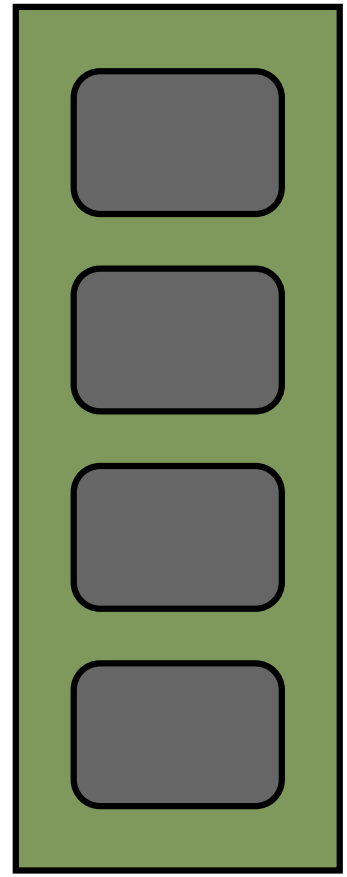
- La mémoire

- Le processeur

Vision système/architecture

La mémoire

C'est un composant essentiel qui sert à stocker des données.



La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : 8 bits

01101000
01101001
00101100
01110101
01100111
01110101
01100100
00111111

La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : un entier $0 \leq n \leq 255$

104
105
44
117
103
117
100
63

La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : 2 chiffres héxadécimaux

68
69
2C
75
67
75
64
3F

La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : 2 chiffres héxadécimaux
- Chaque case est identifiée par son **adresse**

Adresse	Valeur
0	68
1	69
2	2C
3	75
4	67
5	75
6	64
7	3F

La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : 2 chiffres héxadécimaux
- Chaque case est identifiée par son **adresse**

Deux opérations sont possibles sur la mémoire.

Adresse	Valeur
0	68
1	69
2	2C
3	75
4	67
5	75
6	64
7	3F

La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : 2 chiffres héxadécimaux
- Chaque case est identifiée par son **adresse**

Deux opérations sont possibles sur la mémoire.

- **Lire** la valeur d'une case : `read(0x02)` $\rightarrow$ `0x2C`

Adresse	Valeur
0	68
1	69
2	**2C**
3	75
4	67
5	75
6	64
7	3F

La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : 2 chiffres héxadécimaux
- Chaque case est identifiée par son **adresse**

Deux opérations sont possibles sur la mémoire.

- **Lire** la valeur d'une case : `read(0x02)` $\rightarrow$ `0x2C`

Adresse	Valeur
0	68
1	69
2	2C
3	75
4	67
5	75
6	64
7	3F

La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : 2 chiffres héxadécimaux
- Chaque case est identifiée par son **adresse**

Deux opérations sont possibles sur la mémoire.

- **Lire** la valeur d'une case : `read(0x02)` $\rightarrow$ `0x2C`
- **Écrire** la valeur d'une case : `write(0x04, 0x6D)`

Adresse	Valeur
0	68
1	69
2	2C
3	75
4	67
5	75
6	64
7	3F

La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : 2 chiffres héxadécimaux
- Chaque case est identifiée par son **adresse**

Deux opérations sont possibles sur la mémoire.

- **Lire** la valeur d'une case : `read(0x02)` $\rightarrow$ `0x2C`
- **Écrire** la valeur d'une case : `write(0x04, 0x6D)`
  (l'ancienne valeur de la case est perdue)

Adresse	Valeur
0	68
1	69
2	2C
3	75
4	**6D**
5	75
6	64
7	3F

La mémoire

C'est un composant essentiel qui sert à stocker des données.

Défini comme une suite finie de **cases** mémoires.

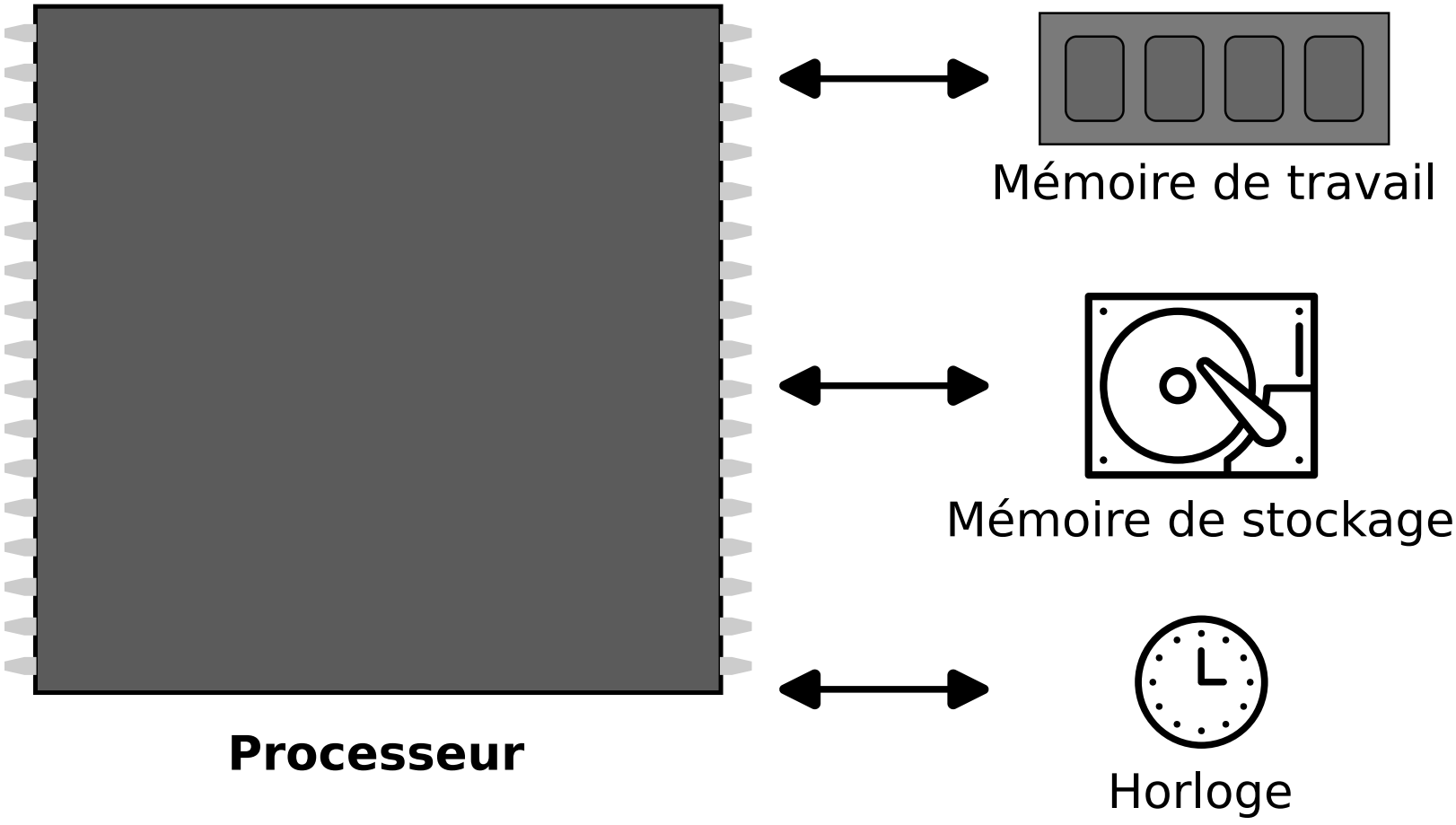
- L'exemple (à droite) comporte 8 cases
- Chaque case contient **1 octet** : 2 chiffres héxadécimaux
- Chaque case est identifiée par son **adresse**

Deux opérations sont possibles sur la mémoire.

- **Lire** la valeur d'une case : `read(0x02)` $\rightarrow$ `0x2C`
- **Écrire** la valeur d'une case : `write(0x04, 0x6D)`

Adresse	Valeur
0	68
1	69
2	2C
3	75
4	6D
5	75
6	64
7	3F

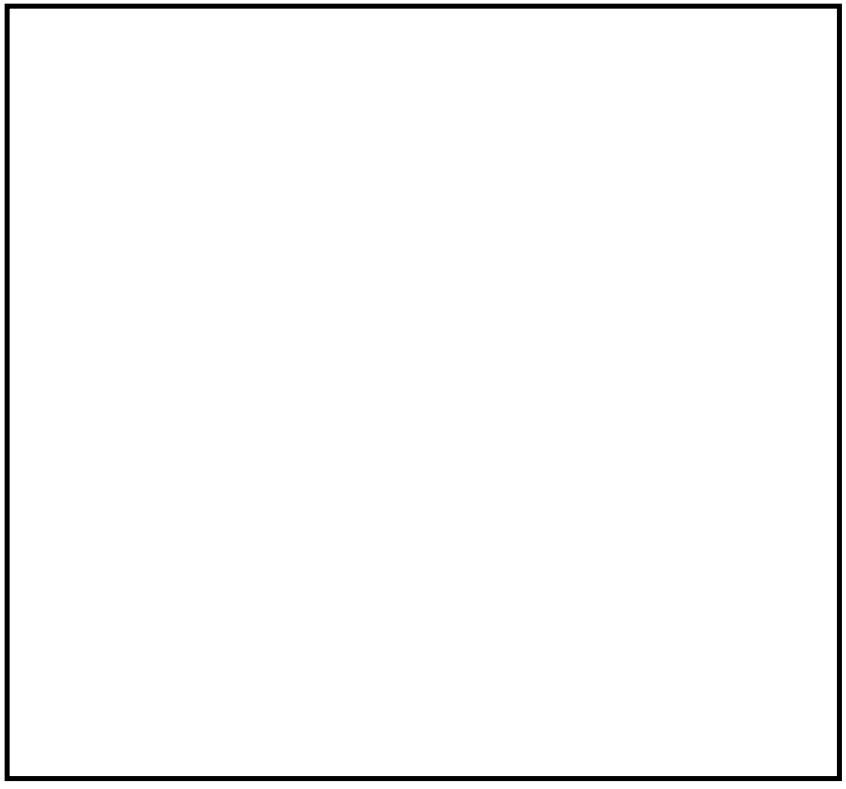
CPU : électronique



Le processeur (ou CPU pour *Central Processing Unit*)
est un **composant électronique** essentiel

Il est connecté à de nombreux autres composants
(directement ou non)

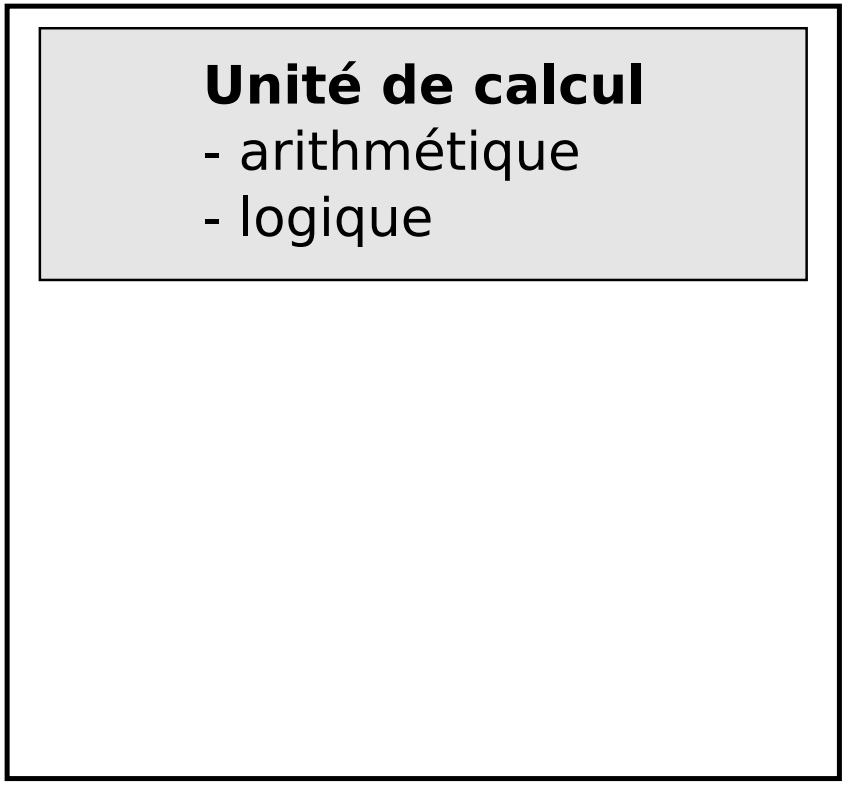
CPU : composants internes



Processeur

Le processeur contient lui-même plusieurs composants

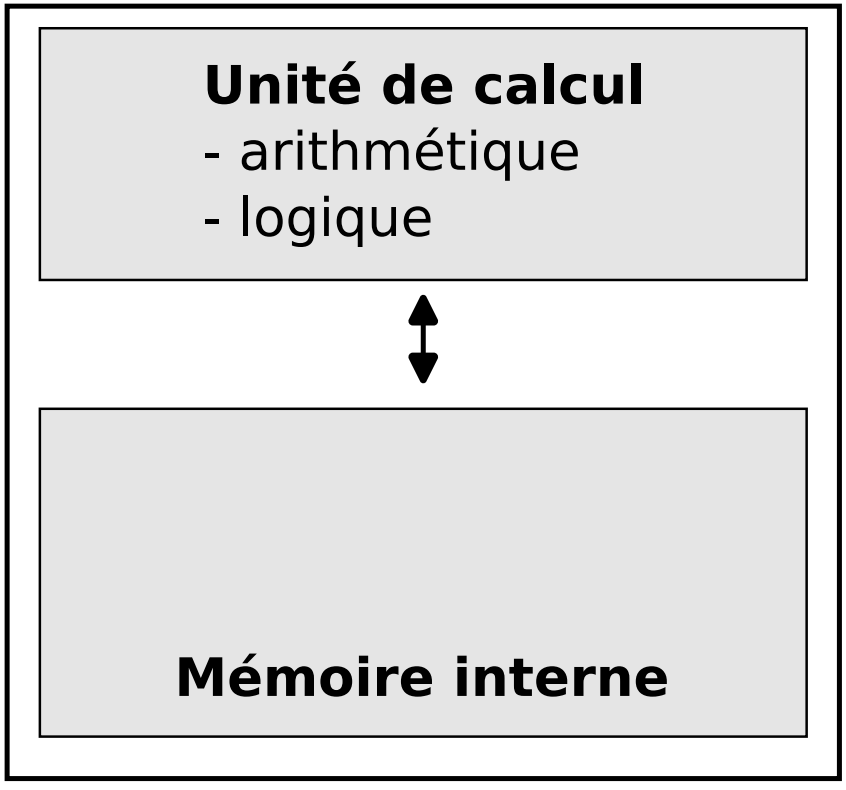
CPU : composants internes



Processeur

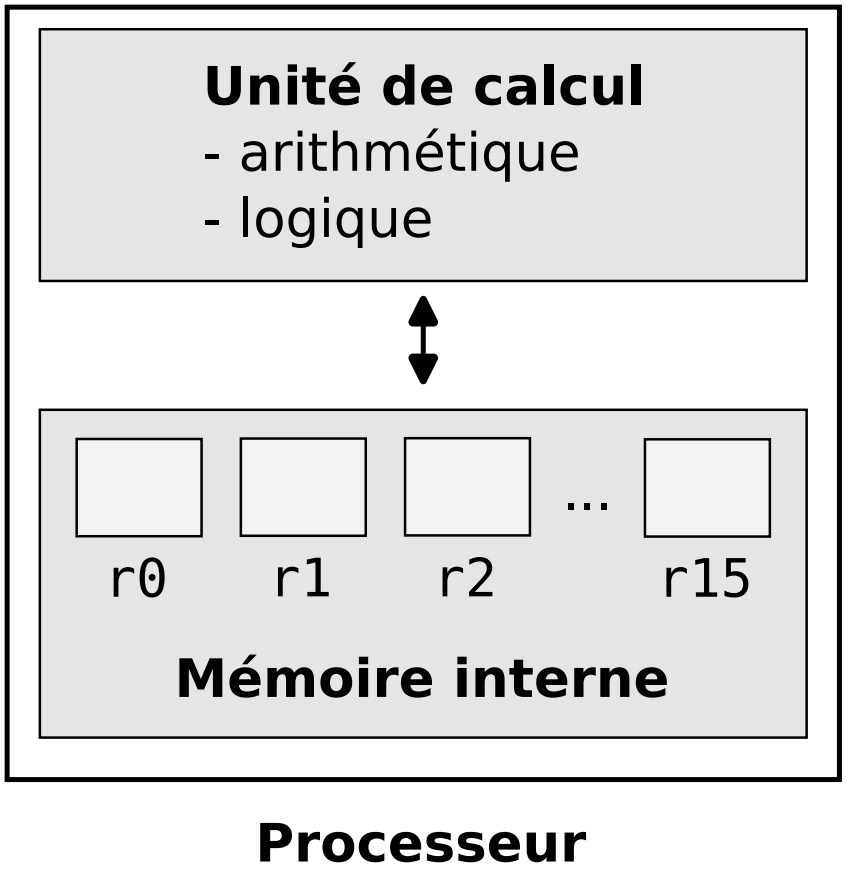
Le processeur contient lui-même plusieurs composants
- Une **ALU** (unité de calcul arithmétique et logique)

CPU : composants internes



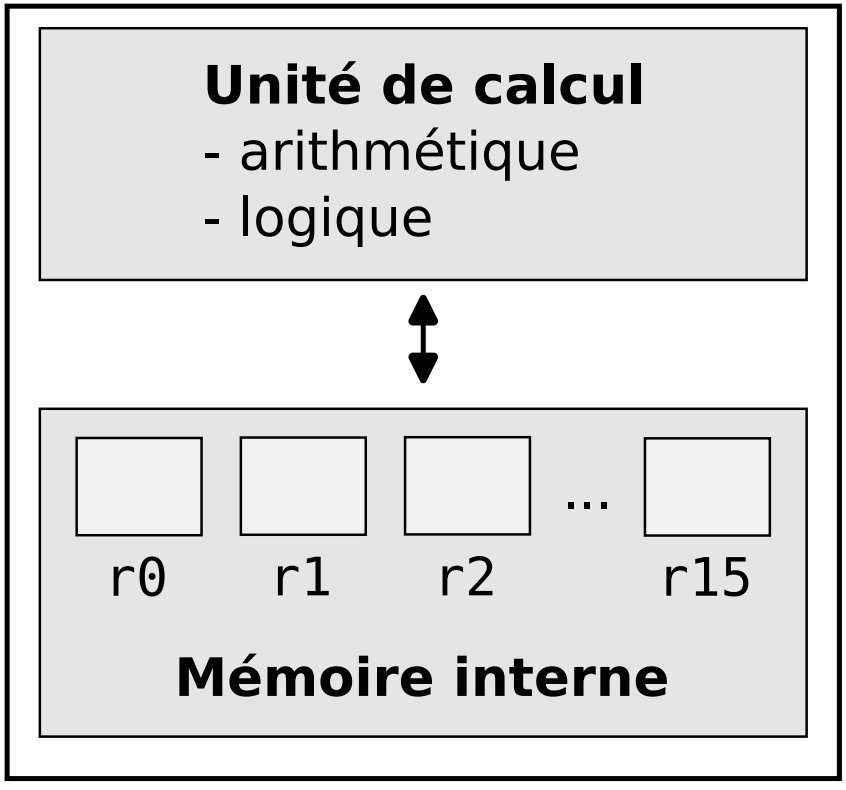
Le processeur contient lui-même plusieurs composants
- Une **ALU** (unité de calcul arithmétique et logique)
- Une **mémoire interne**

CPU : composants internes



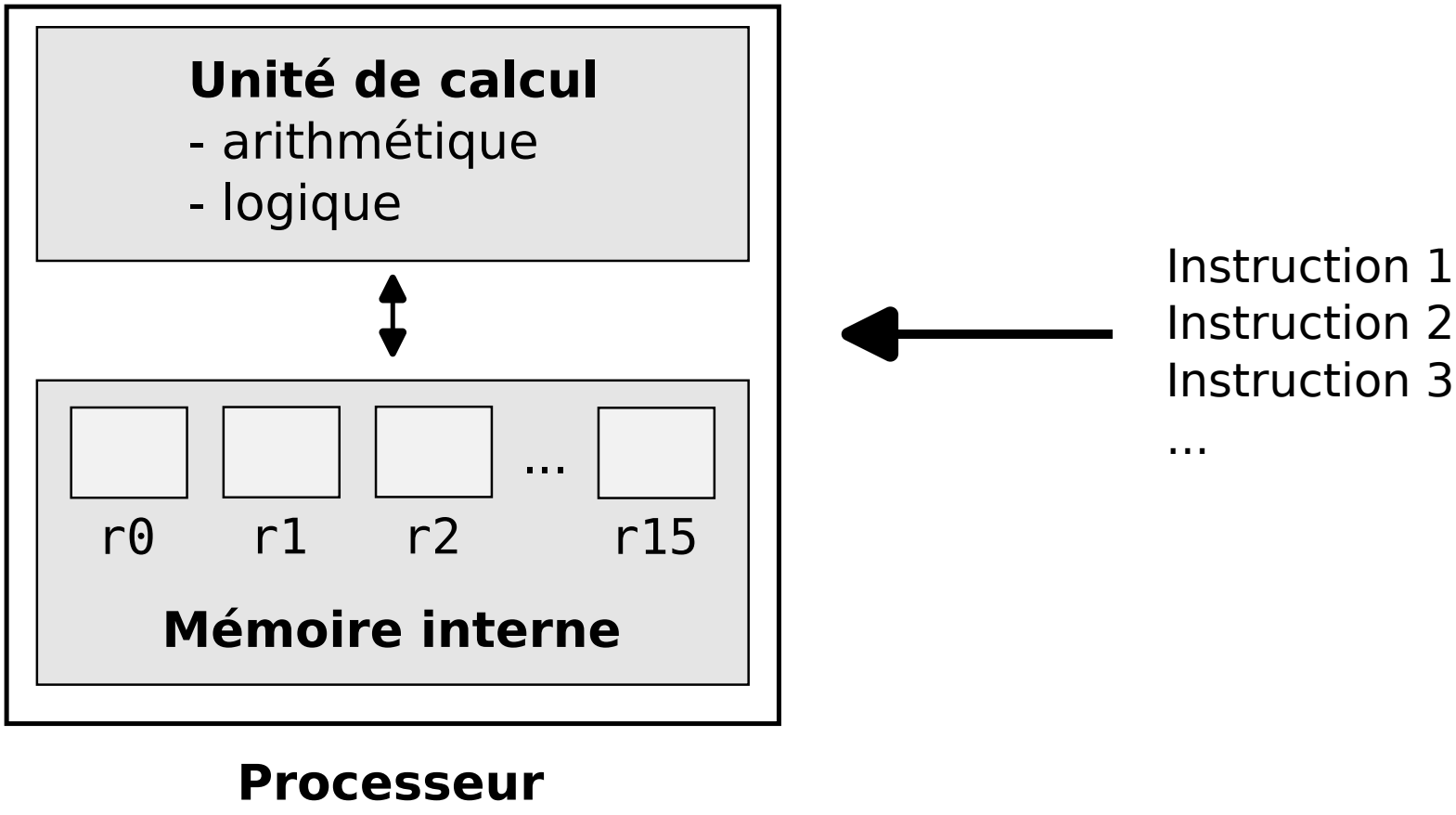
Le processeur contient lui-même plusieurs composants
- Une **ALU** (unité de calcul arithmétique et logique)
- Une **mémoire interne** qui contient notamment des **registres**
  - Un registre est une mémoire de **taille fixe** (*e.g.*, 32 bits)
  - Chaque registre a un **identifiant unique** (*e.g.*, `r0`)
  - D'autres données peuvent être stockées en mémoire interne (flags...)

CPU : rôle



Le processeur a pour rôle d'effectuer des **calculs**

CPU : rôle

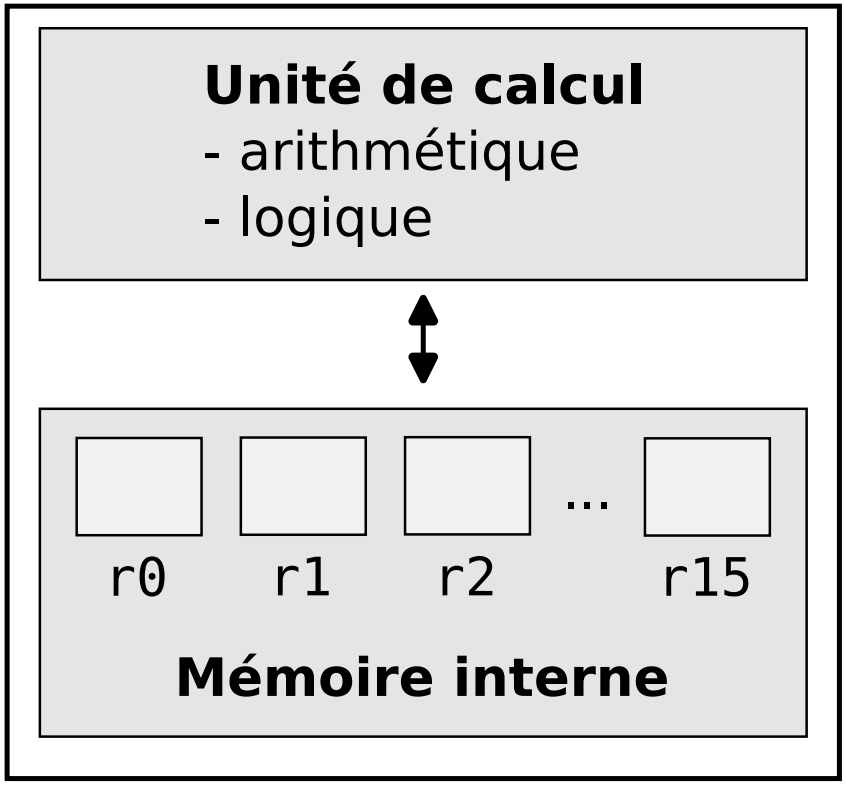


Le processeur a pour rôle d'effectuer des **calculs**, en exécutant des **opérations atomiques** appelées **instructions**, l'une après l'autre

Comportement :

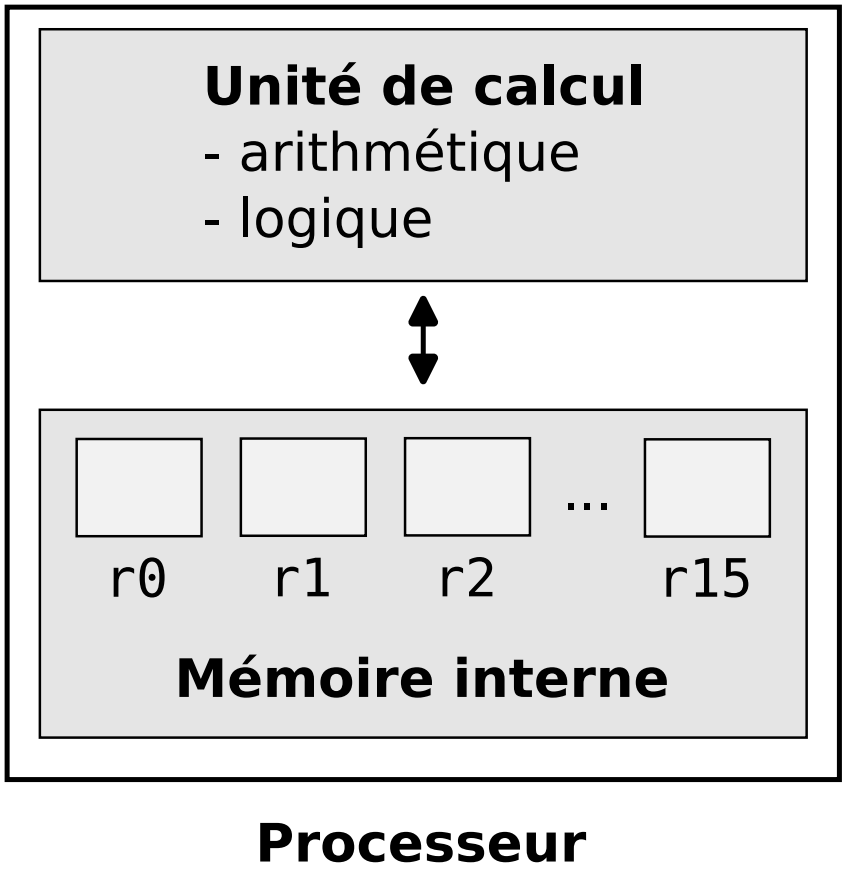
```
Boucle infinie :
| i = charger la prochaine instruction
| exécuter(i)
```

CPU : instructions



Les instructions sont les **opérations atomiques** qu'exécute le processeur

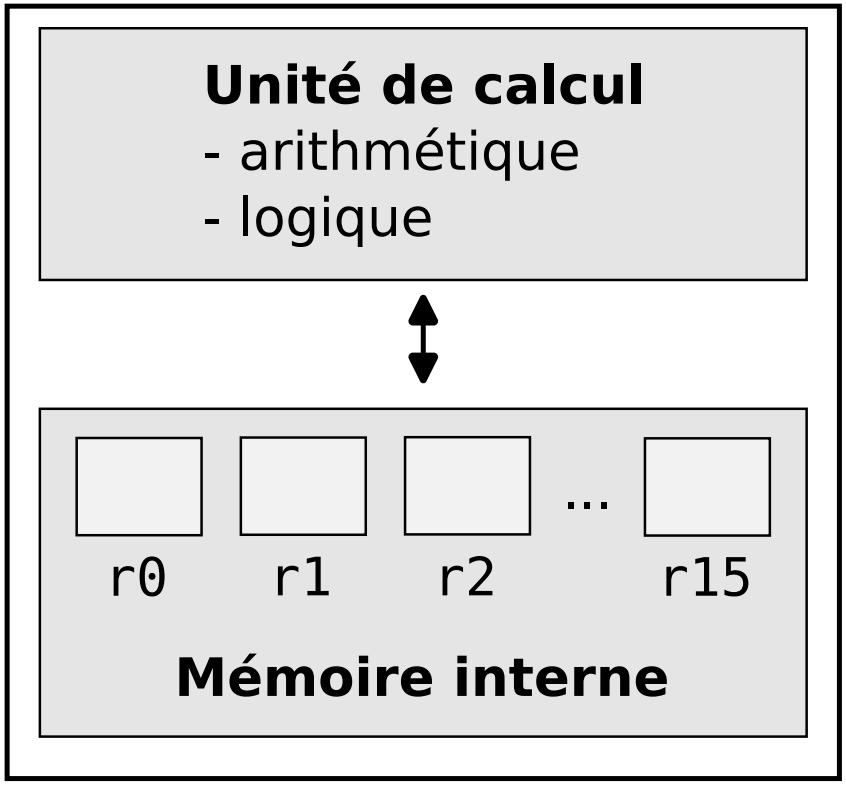
CPU : instructions



Les instructions sont les **opérations atomiques** qu'exécute le processeur

Pour montrer comment fonctionne un processeur, nous allons exécuter pas à pas une séquence d'instructions. Les instructions données ici et la manière de les écrire n'est pas importante, ce sont juste des exemples.
Important et à retenir : ce que peut faire ou non un processeur

CPU : instructions



nom de l'instruction

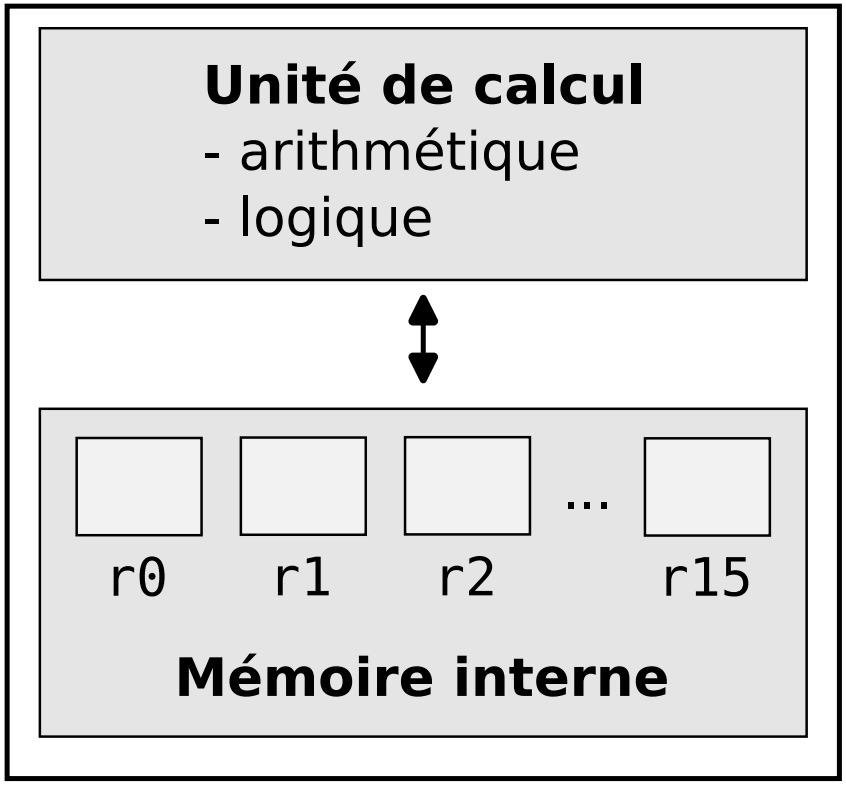
```
mov [0x01], %r0
```

paramètres de l'instruction

Les instructions sont les **opérations atomiques** qu'exécute le processeur

Voici un premier exemple d'instruction.
La syntaxe des instructions ne va pas nous intéresser ici,
mais chaque instruction sera représentée sur une ligne.

CPU : instructions



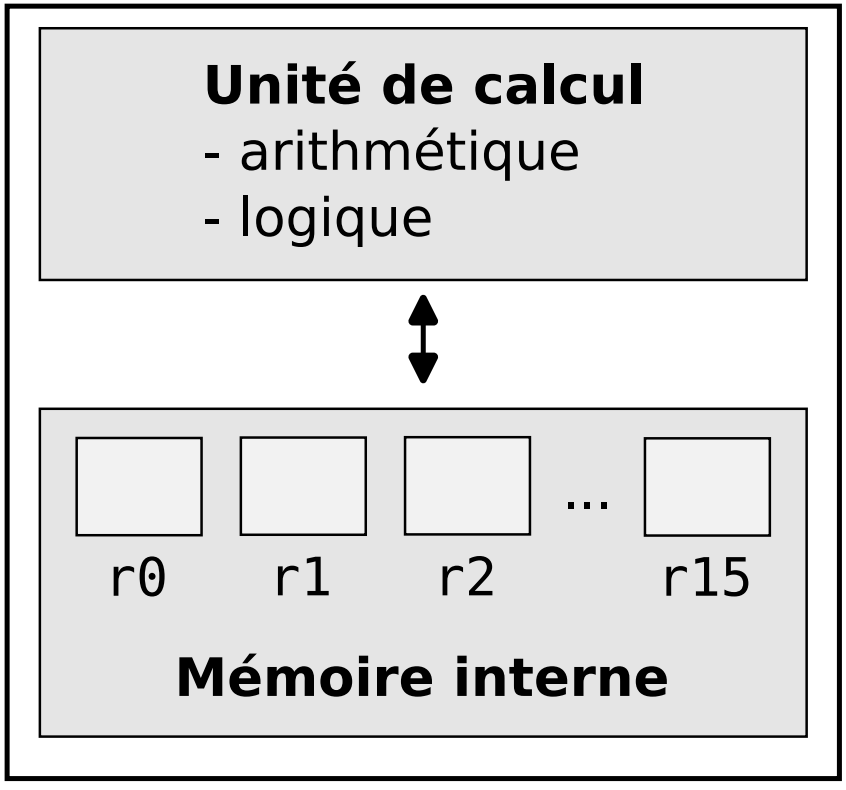
l'instruction en cours de traitement est indiquée par cette flèche

```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```

séquence d'instructions d'exemple que nous allons exécuter

Les instructions sont les **opérations atomiques** qu'exécute le processeur

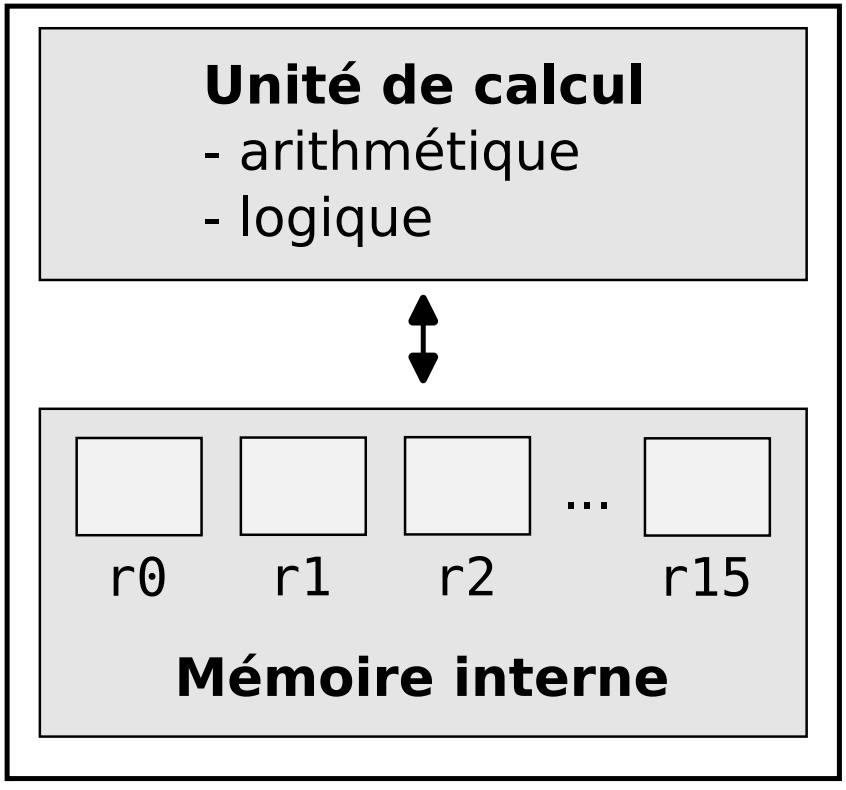
CPU : instructions




```
→0: mov [0x01], %r0
 1: mov %r0, %r1
 2: add $1, %r0
 3: cmp $2, %r0
 4: jnz 2
 5: syscall
```

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données**

CPU : instructions



Processeur

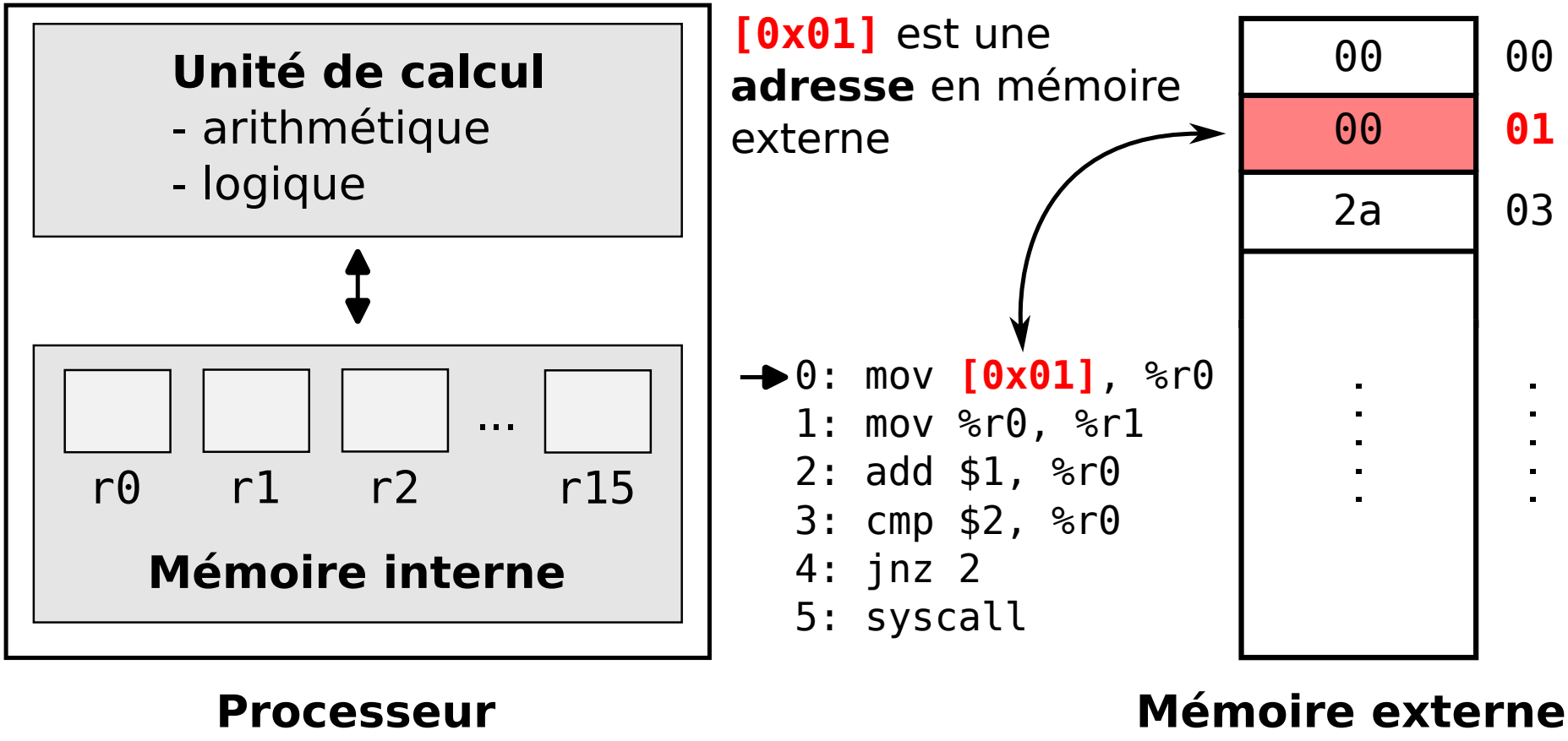
```
→0: mov [0x01], %r0
 1: mov %r0, %r1
 2: add $1, %r0
 3: cmp $2, %r0
 4: jnz 2
 5: syscall
```

00	00
00	01
2a	03
.	.
.	.
.	.

Mémoire externe

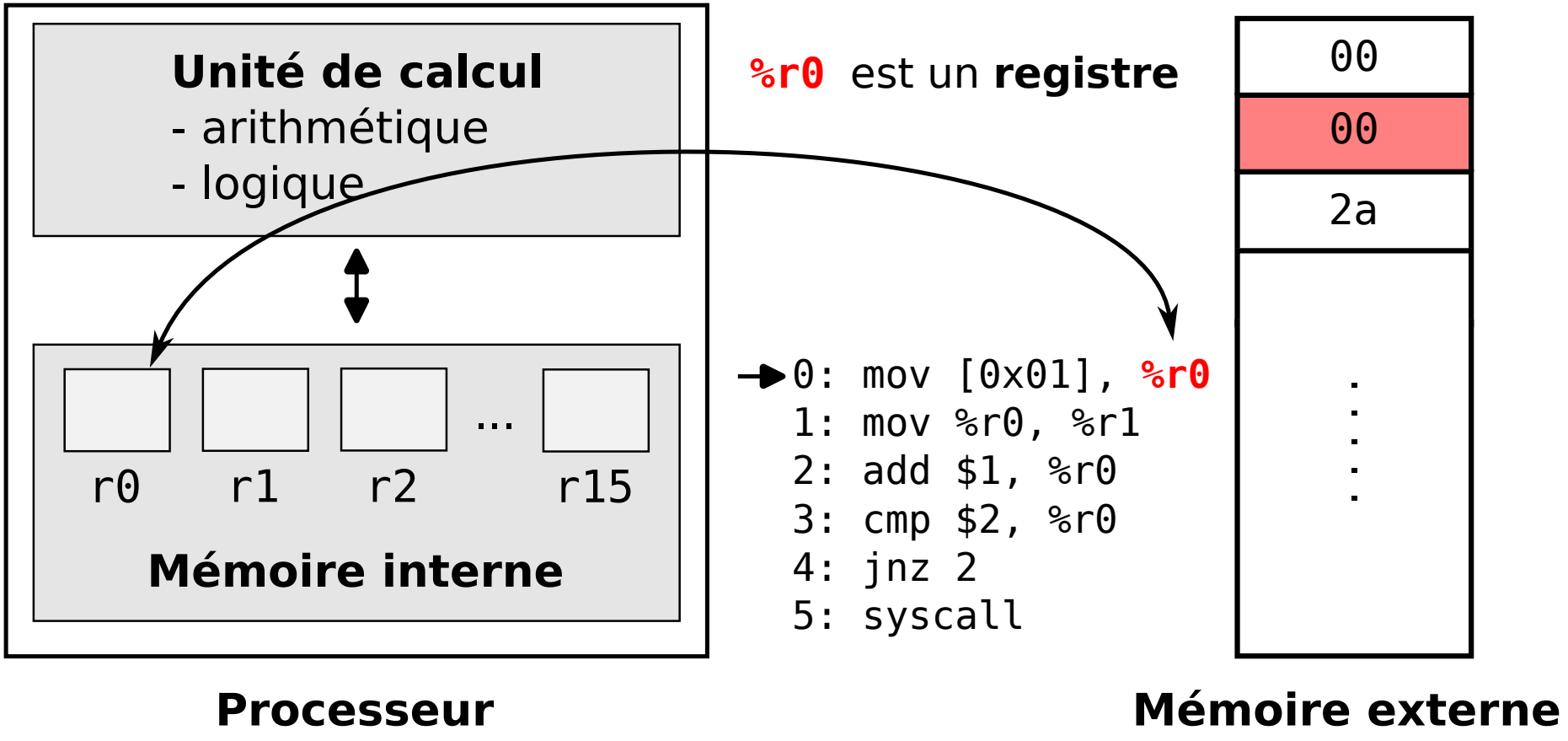
Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres)

CPU : instructions



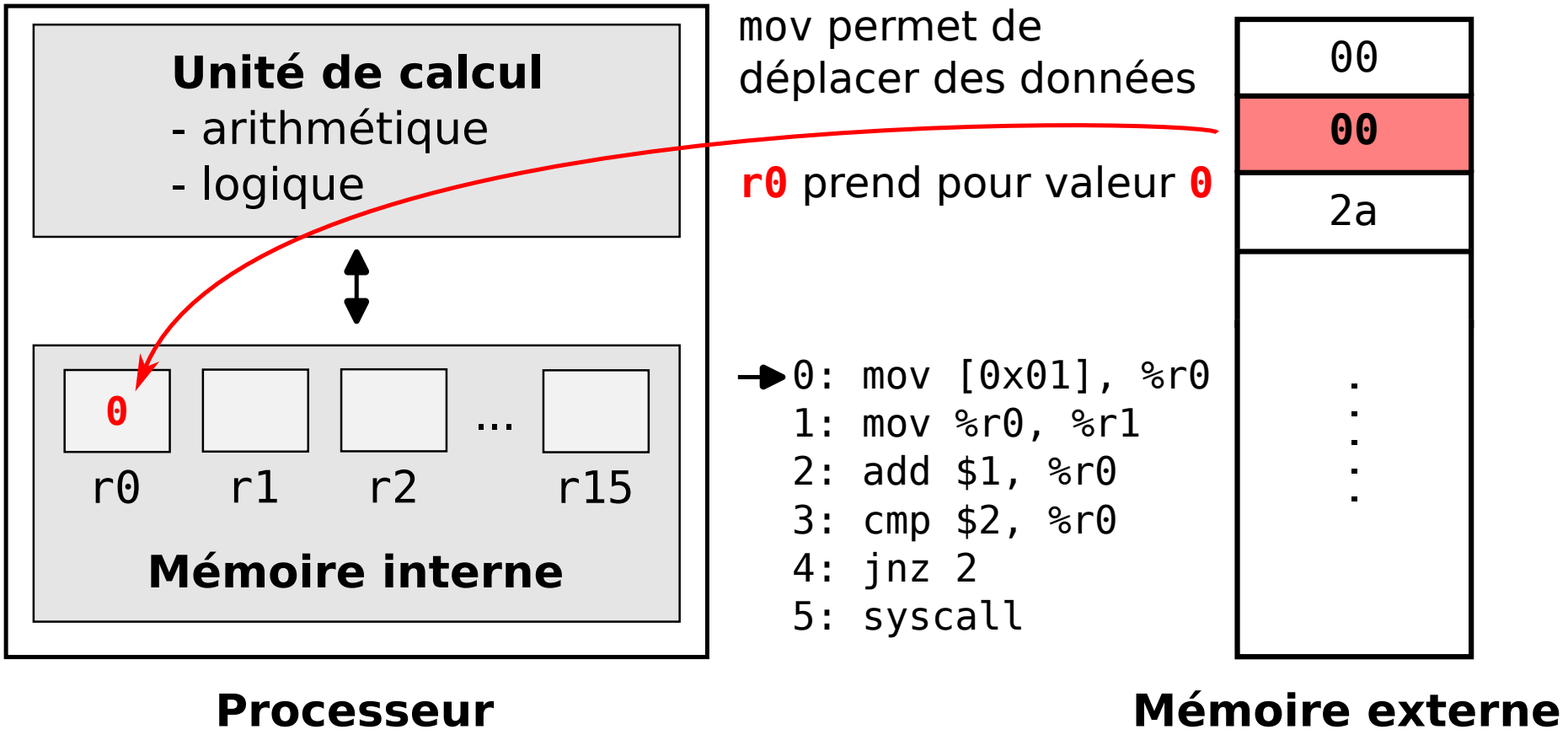
Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres)

CPU : instructions



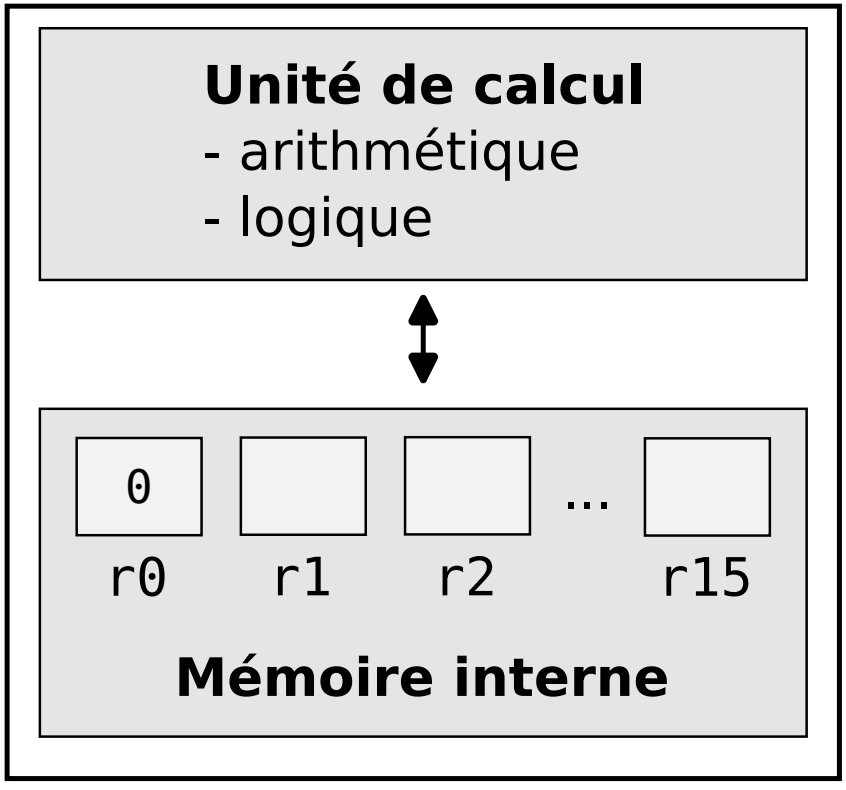
Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres)

CPU : instructions

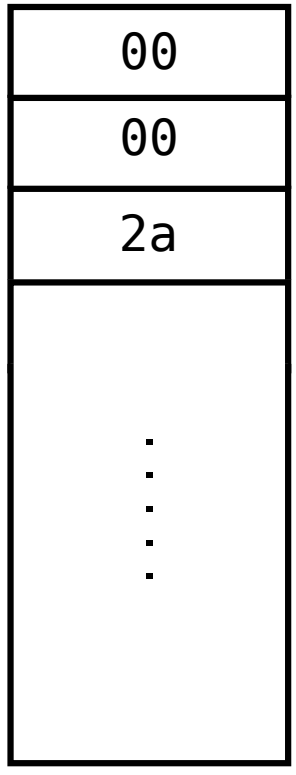


Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres)

CPU : instructions




```
  0: mov [0x01], %r0
→ 1: mov %r0, %r1
  2: add $1, %r0
  3: cmp $2, %r0
  4: jnz 2
  5: syscall
```

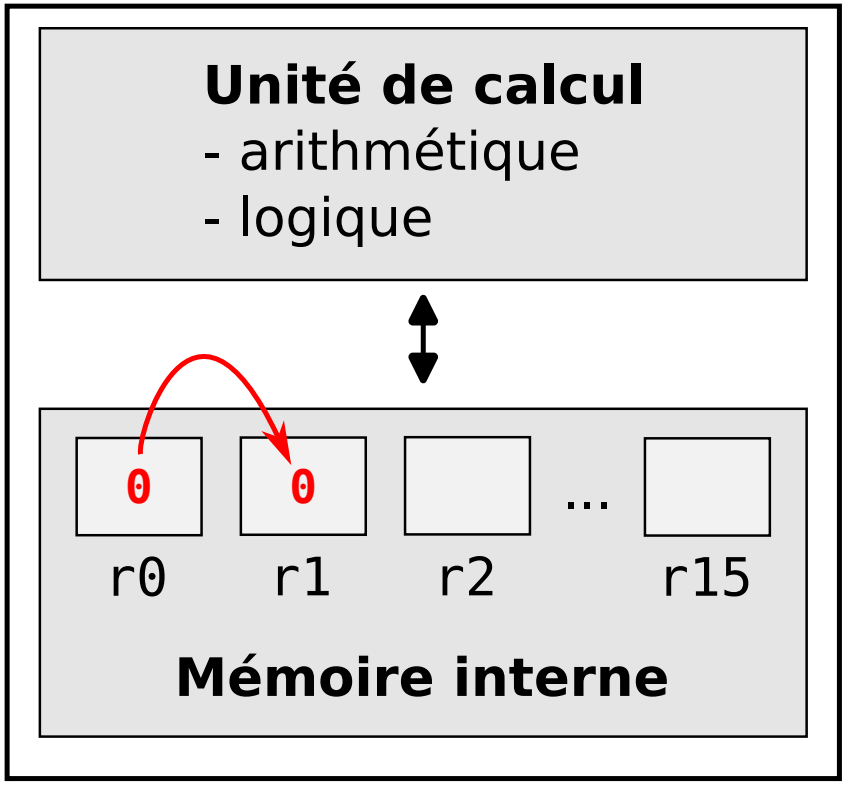


Processeur

Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)

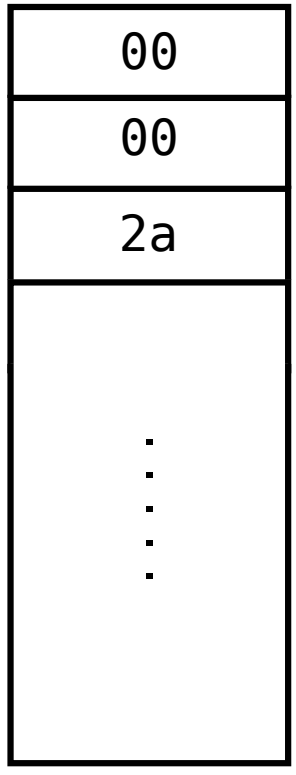
CPU : instructions



Processeur

r1 prend pour valeur **0**
(le contenu de **r0**)

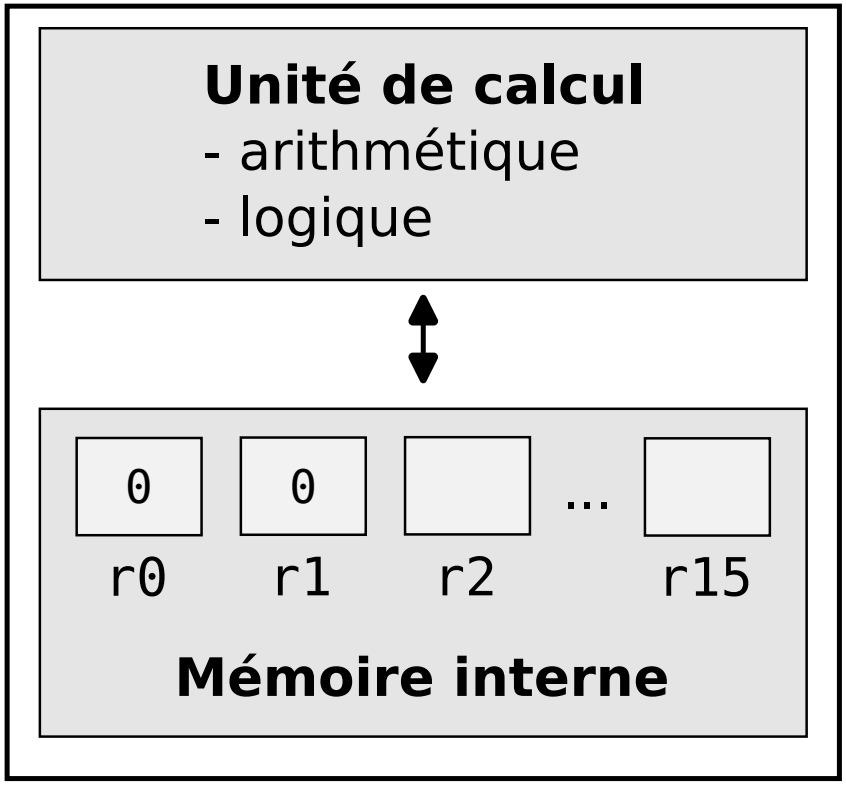
```
  0: mov [0x01], %r0
→ 1: mov %r0, %r1
  2: add $1, %r0
  3: cmp $2, %r0
  4: jnz 2
  5: syscall
```



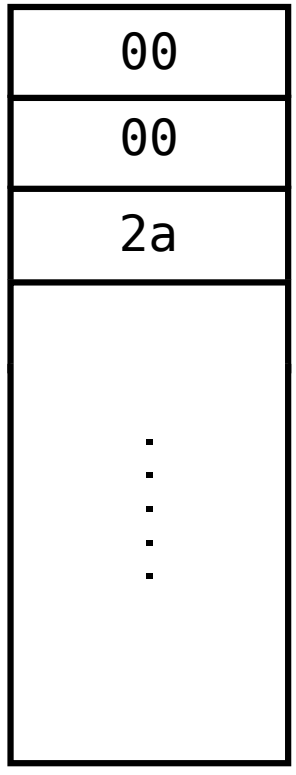
Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)

CPU : instructions



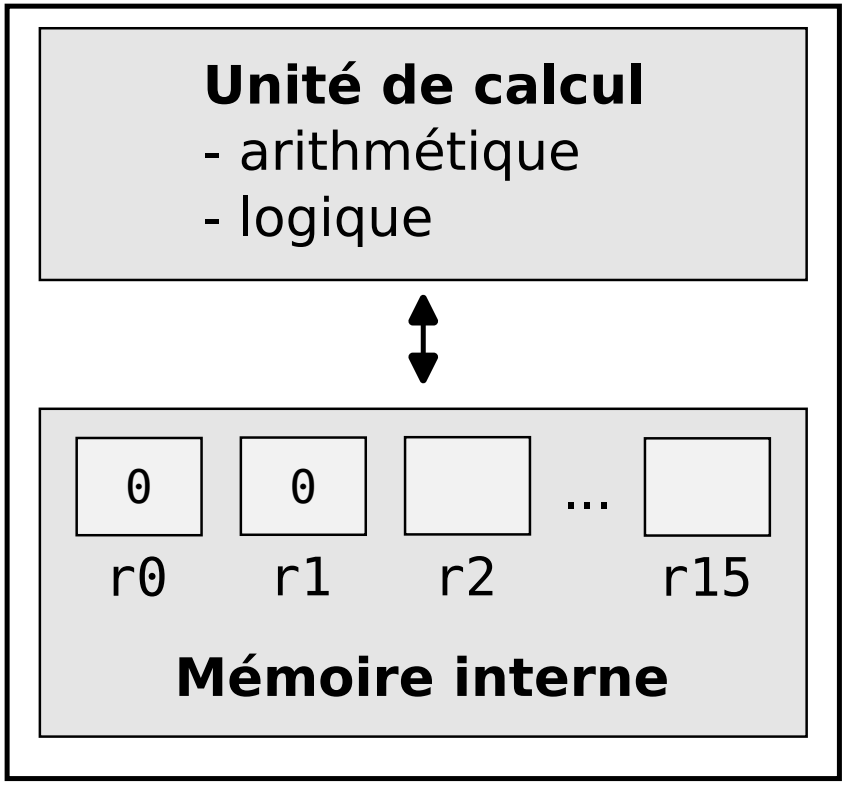

```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



Processeur **Mémoire externe**

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)

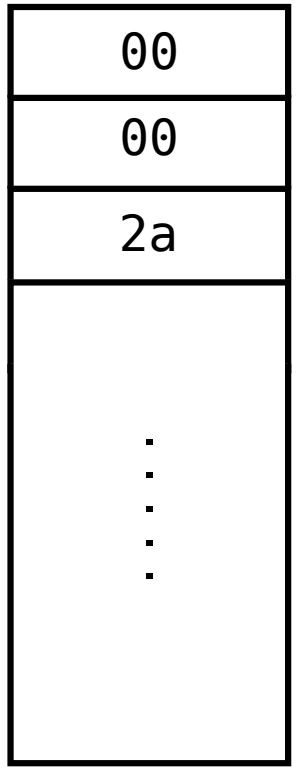
CPU : instructions



Processeur

add permet d'incrémenter la valeur contenue dans un registre (ici **r0**) par une valeur (ici **1**)

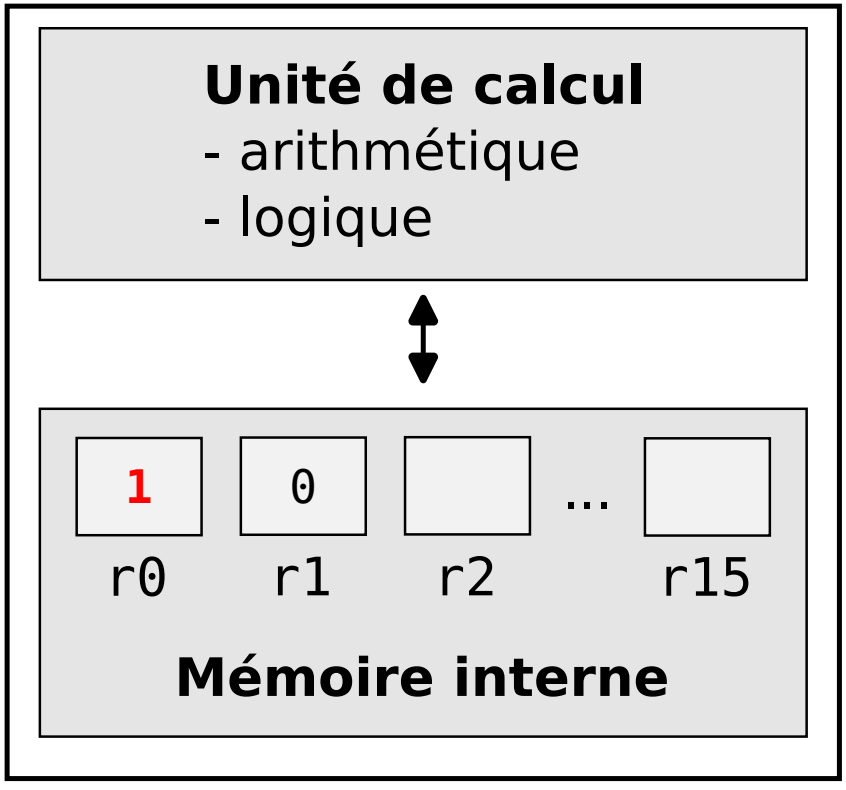
```
  0: mov [0x01], %r0
  1: mov %r0, %r1
→ 2: add $1, %r0
  3: cmp $2, %r0
  4: jnz 2
  5: syscall
```



Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)

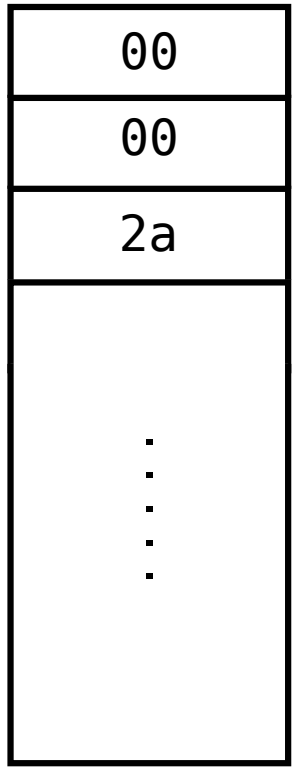
CPU : instructions



Processeur

0+1=1
donc **r0** prend pour nouvelle valeur **1**

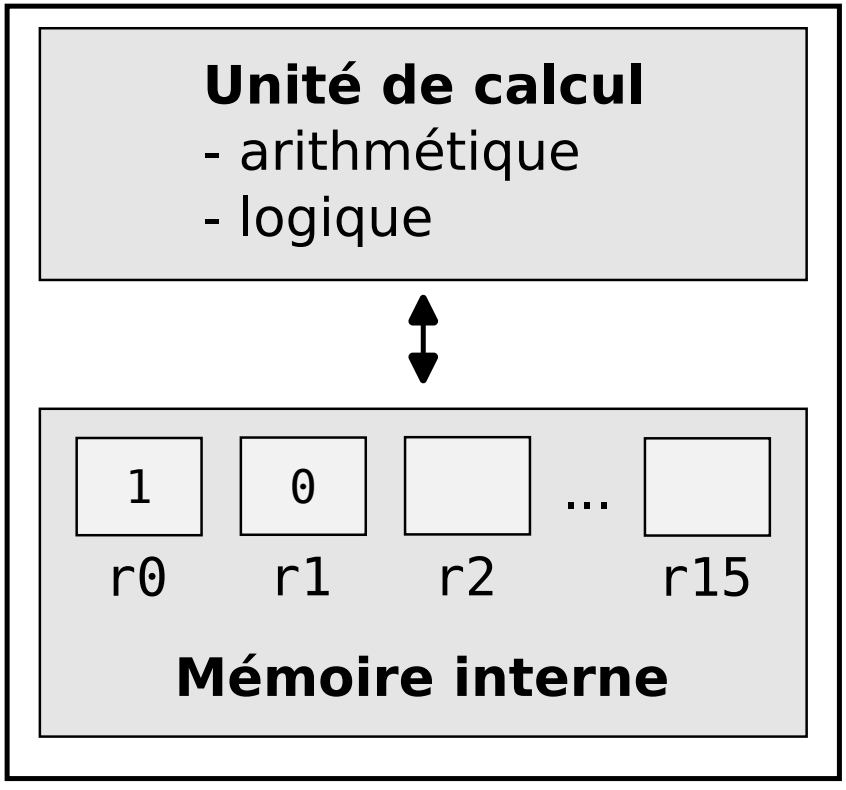
```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



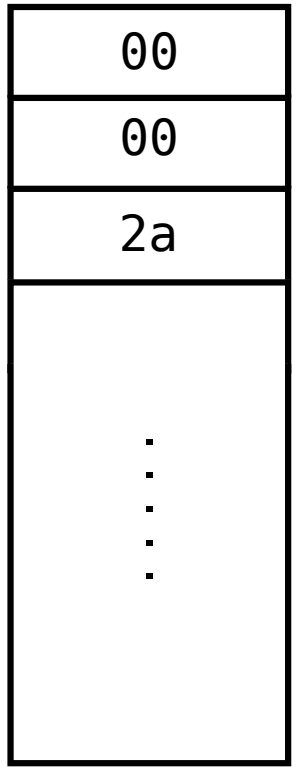
Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)

CPU : instructions



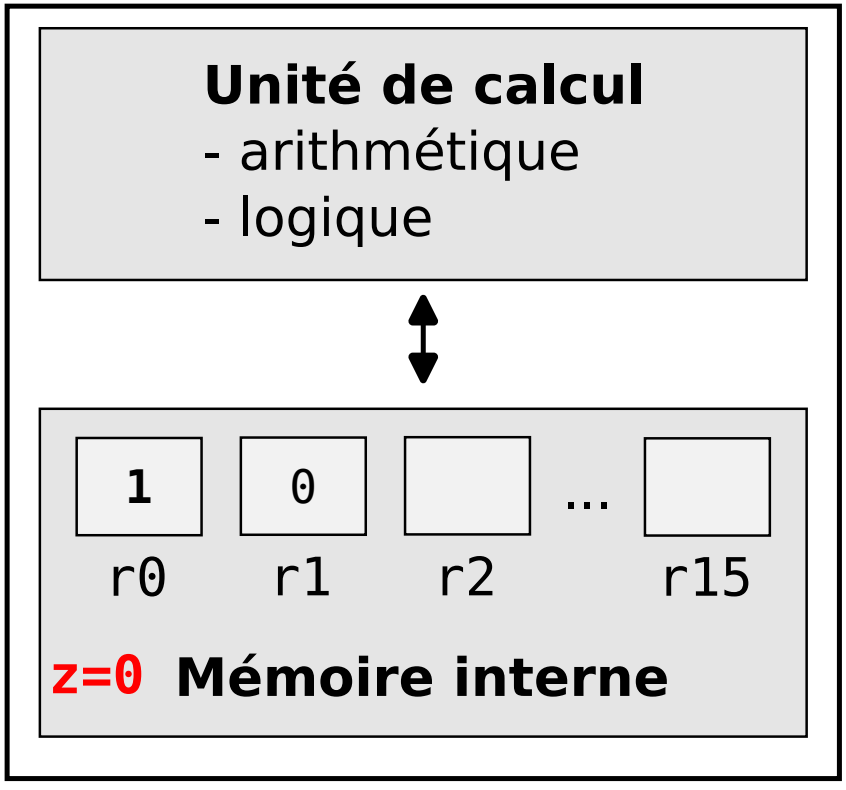

```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



Processeur **Mémoire externe**

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)

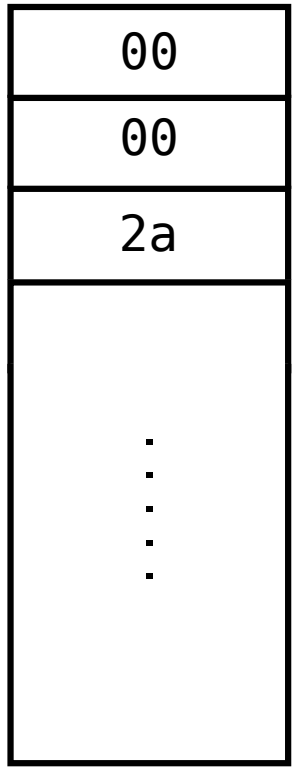
CPU : instructions



Processeur

cmp compare 2 valeurs,
le résultat est stocké
en mémoire interne
dans le flag Z
si =, Z est mis à 1
si ≠, Z est mis à 0

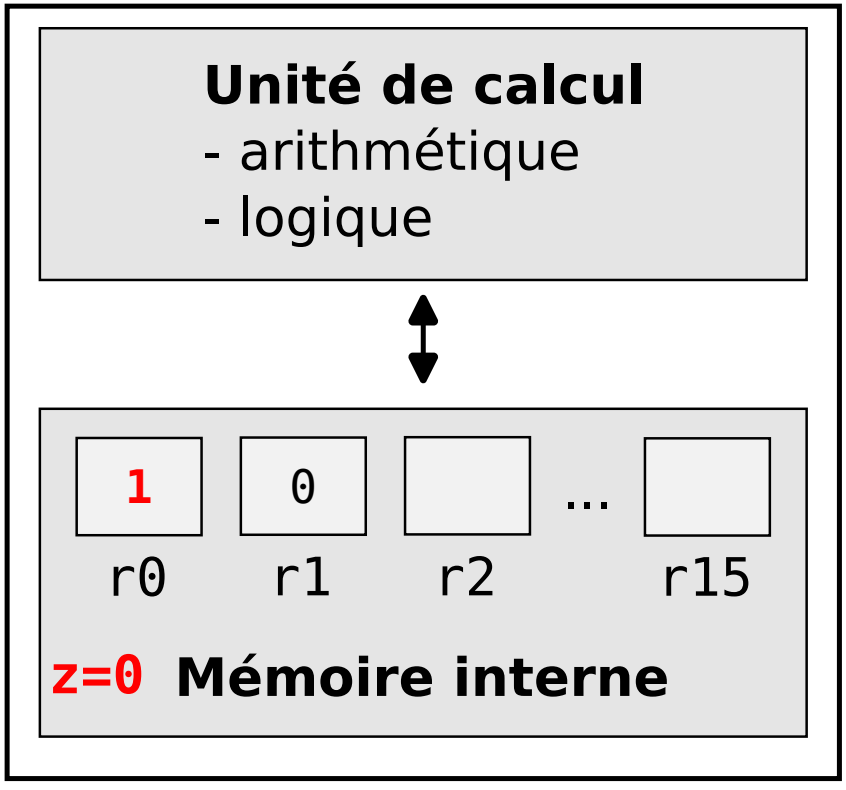
```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)

CPU : instructions

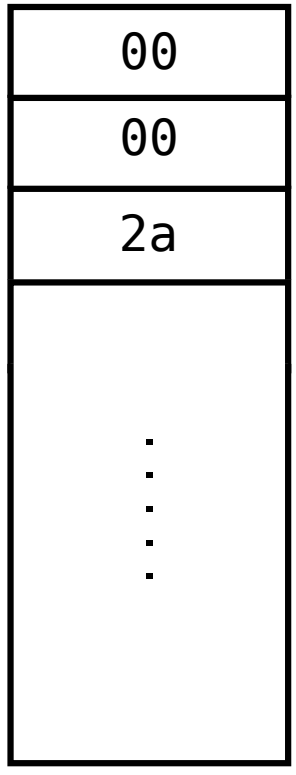


Processeur

ici on compare le
contenu de **r0**
et la valeur **2**

1≠2 donc
Z est mis à 0

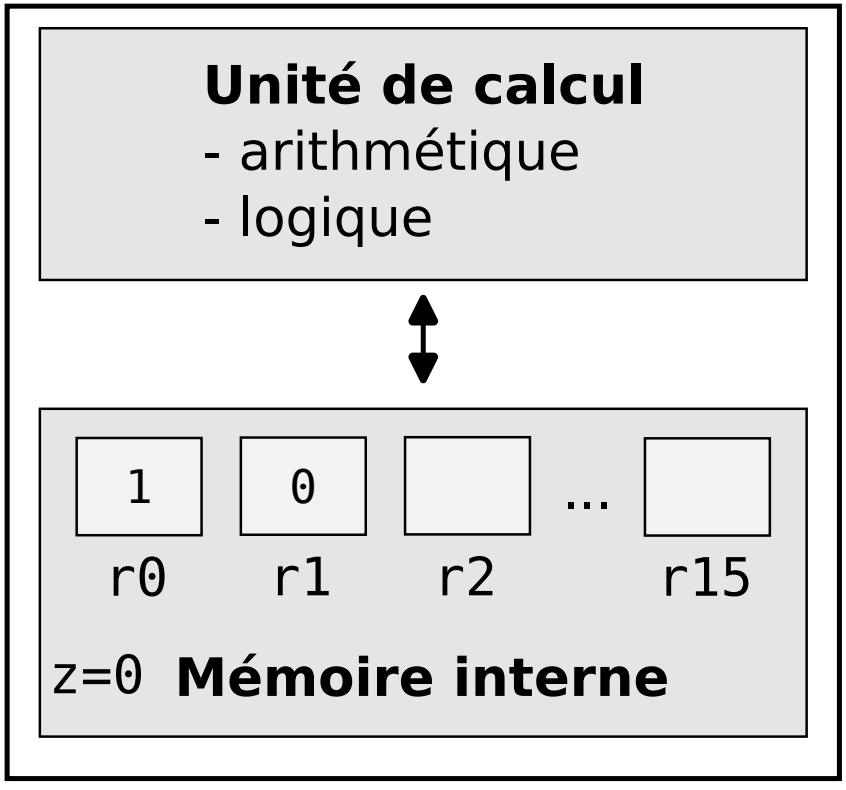
```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



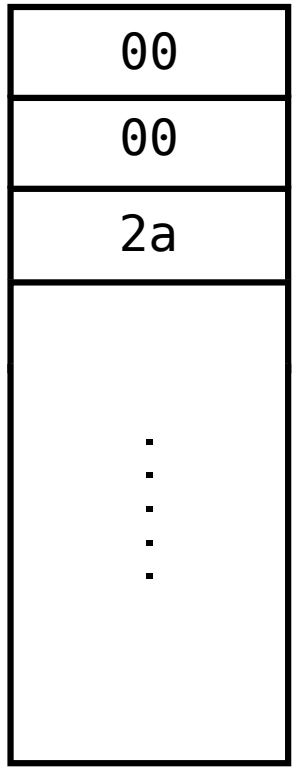
Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)

CPU : instructions



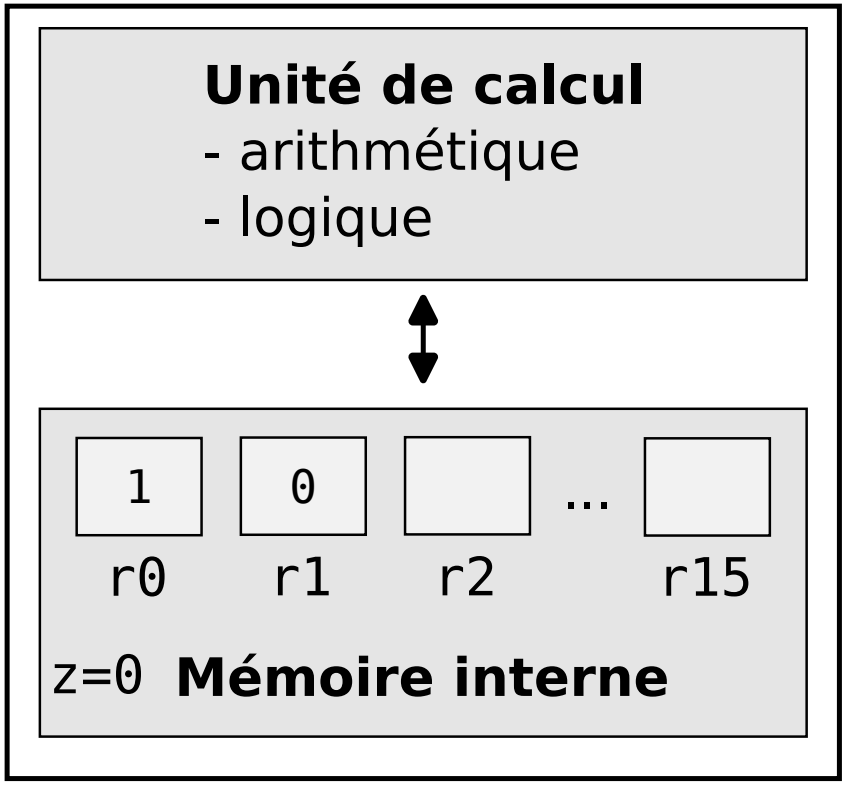

```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



Processeur **Mémoire externe**

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)
- **saut** conditionnel ou non : modifie quelle est la prochaine instruction

CPU : instructions



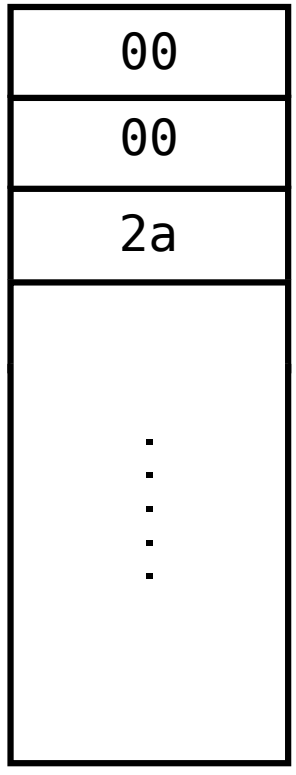
Processeur

`jnz` réalise un saut ou non selon la valeur de Z

Z=0 → saut
(change proch. instr.)

Z=1 → rien faire
(proch. instr. identique)

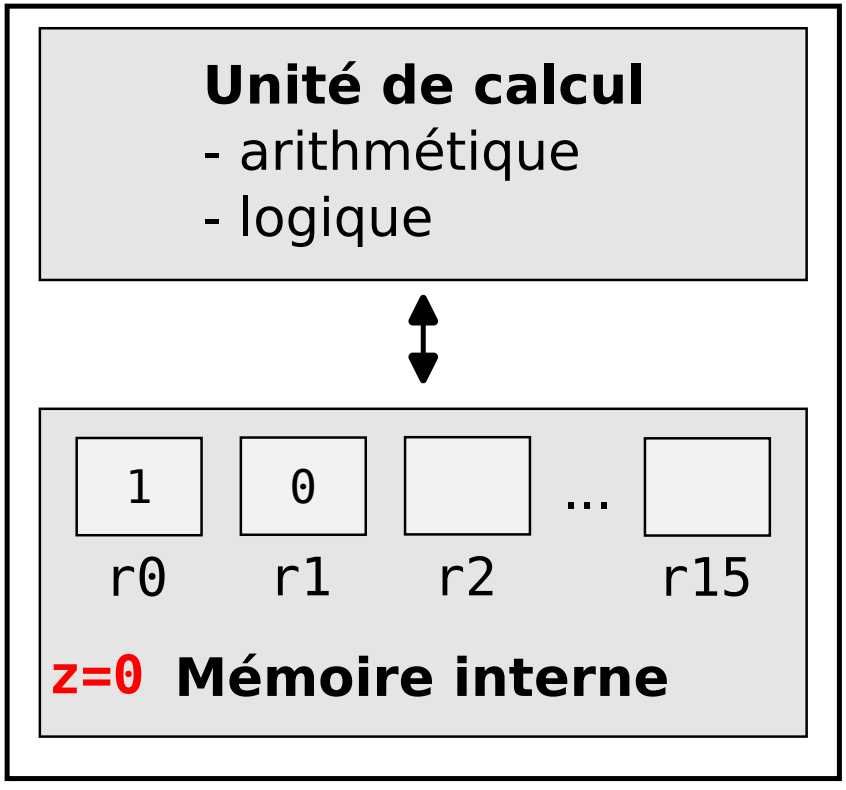
```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



Mémoire externe

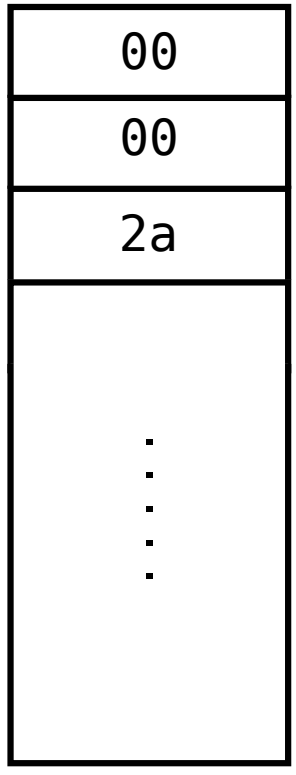
Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)
- **saut** conditionnel ou non : modifie quelle est la prochaine instruction

CPU : instructions



Z=0 donc la prochaine instruction devient l'instruction **2**

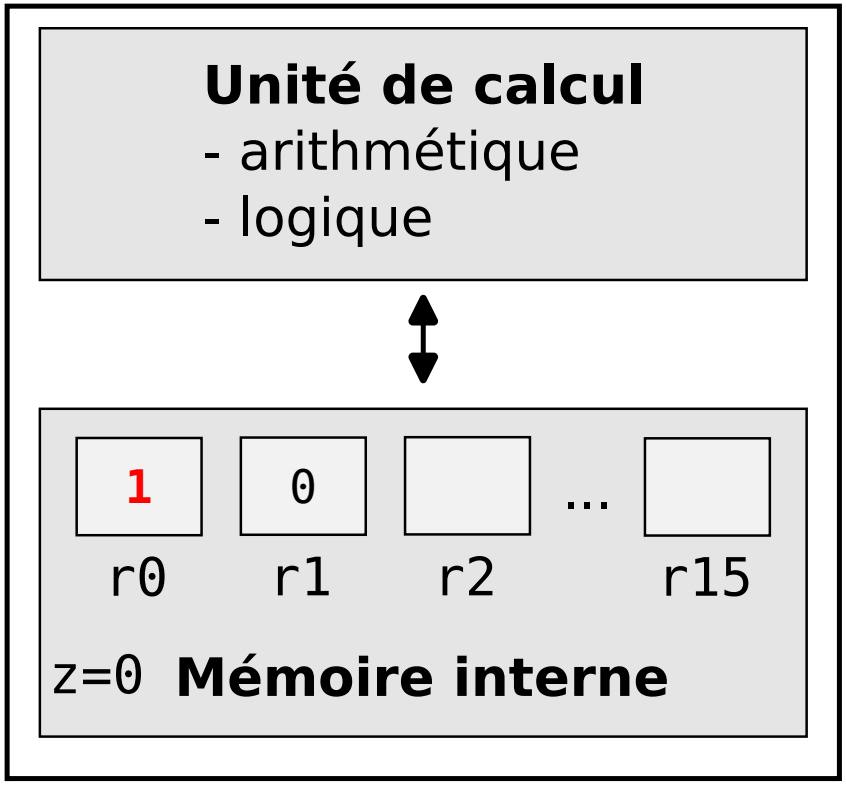
```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



Processeur **Mémoire externe**

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)
- **saut** conditionnel ou non : modifie quelle est la prochaine instruction

CPU : instructions

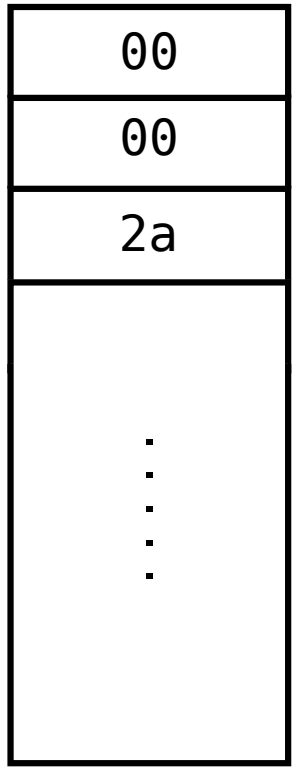


Processeur

on repasse à l'instruction 2

l'état n'est pas le même qu'avant !
(r0 vaut **1** et pas 0)

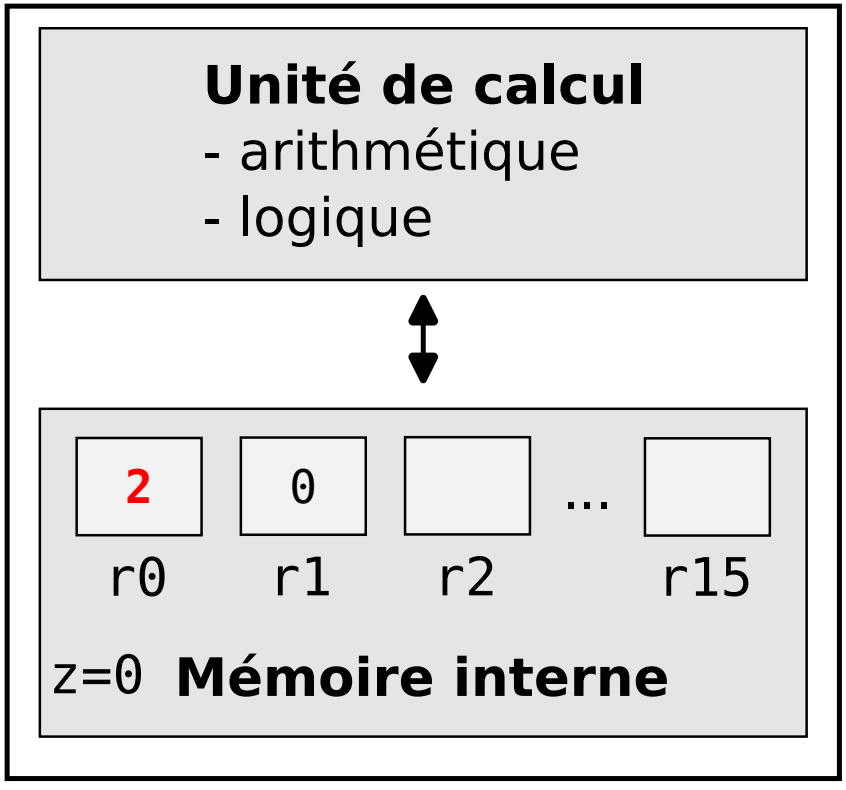
```
  0: mov [0x01], %r0
  1: mov %r0, %r1
→ 2: add $1, %r0
  3: cmp $2, %r0
  4: jnz 2
  5: syscall
```



Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)
- **saut** conditionnel ou non : modifie quelle est la prochaine instruction

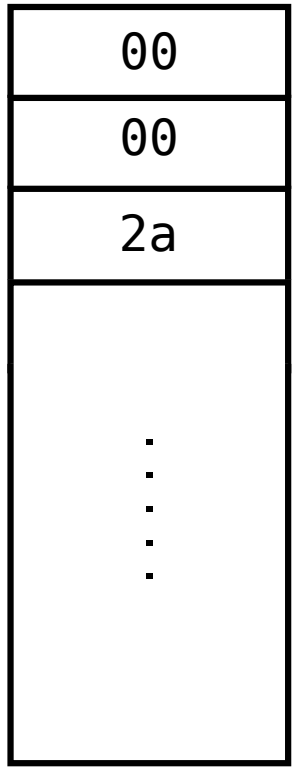
CPU : instructions



Processeur

1+1 vaut 2
donc r0 prend pour valeur 2

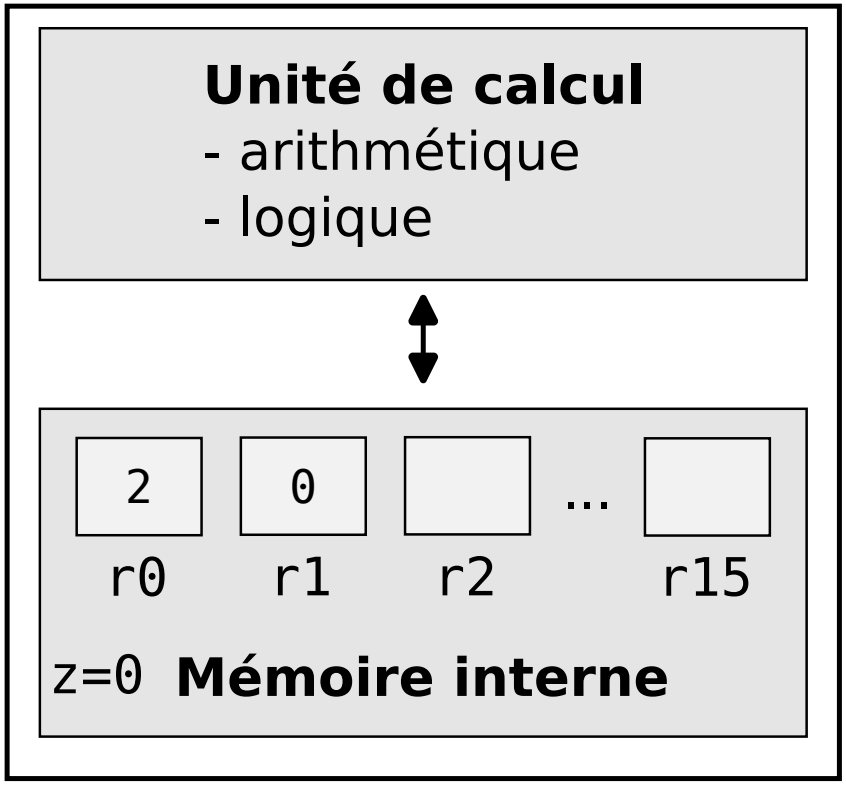
```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



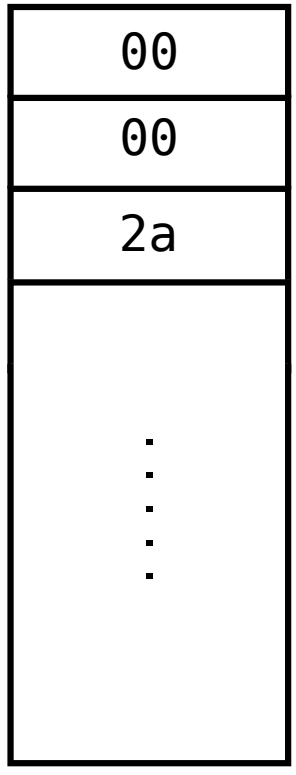
Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)
- **saut** conditionnel ou non : modifie quelle est la prochaine instruction

CPU : instructions



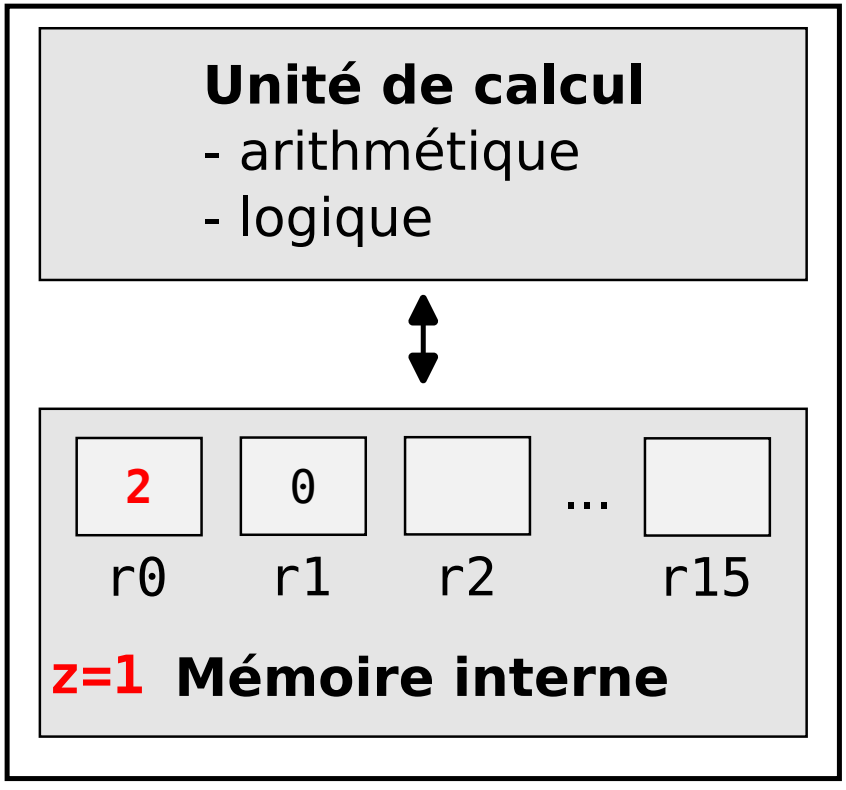

```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



Processeur **Mémoire externe**

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)
- **saut** conditionnel ou non : modifie quelle est la prochaine instruction

CPU : instructions

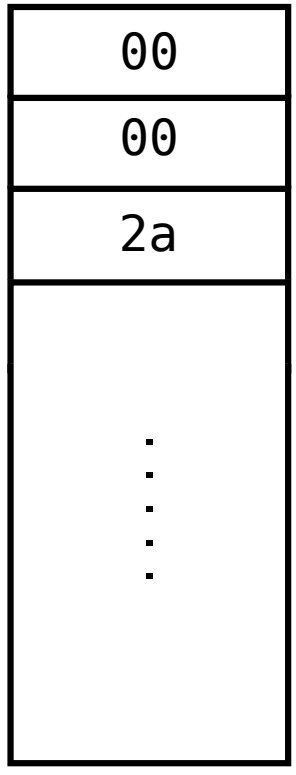


Processeur

on compare le contenu de **r0** (2) et la valeur **2**

2=2 donc
Z est mis à 1

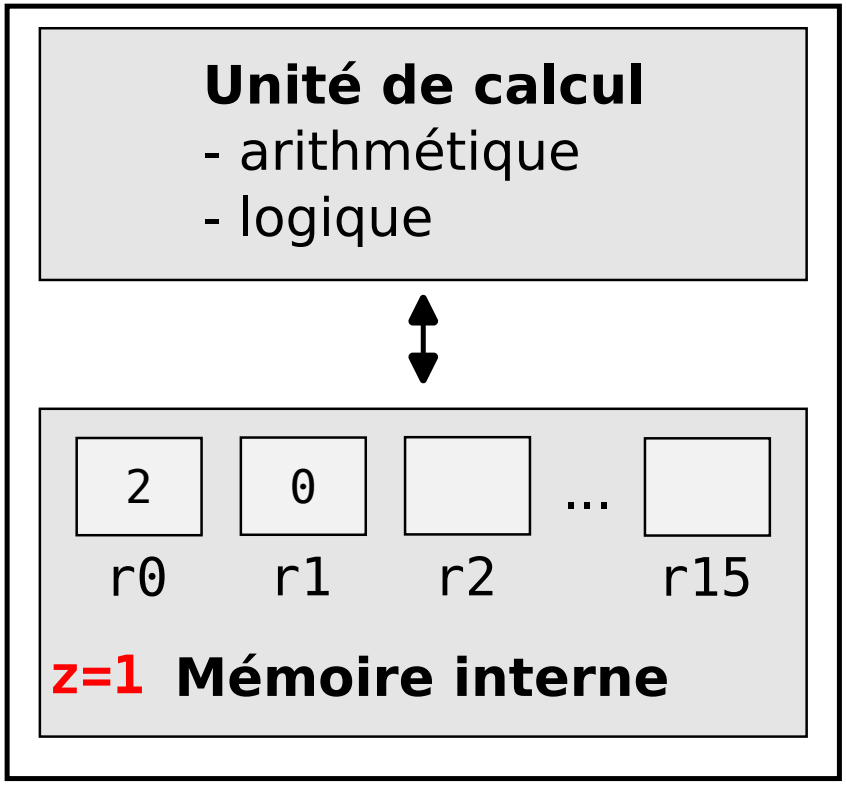
```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)
- **saut** conditionnel ou non : modifie quelle est la prochaine instruction

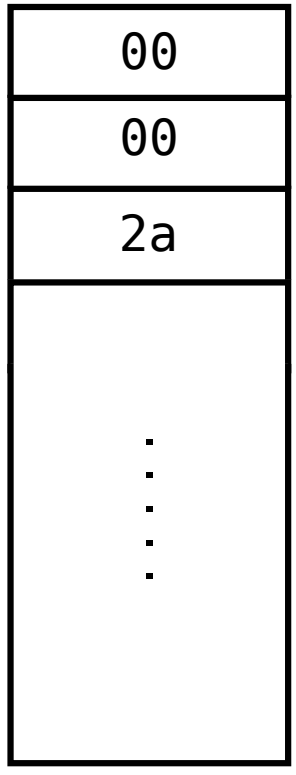
CPU : instructions



Processeur

Z=1 donc on ne fait **pas de saut**, (on ne fait rien puis on passe à l'instruction suivante : 5)

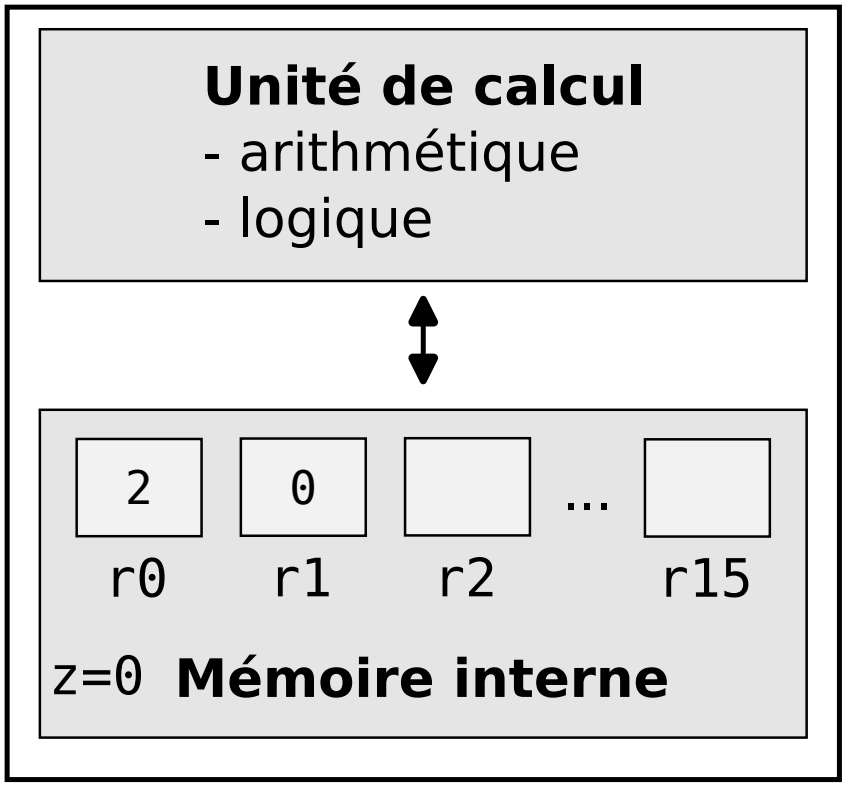
```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



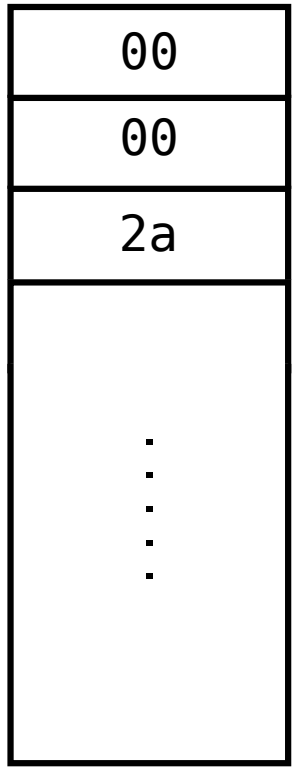
Mémoire externe

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)
- **saut** conditionnel ou non : modifie quelle est la prochaine instruction

CPU : instructions



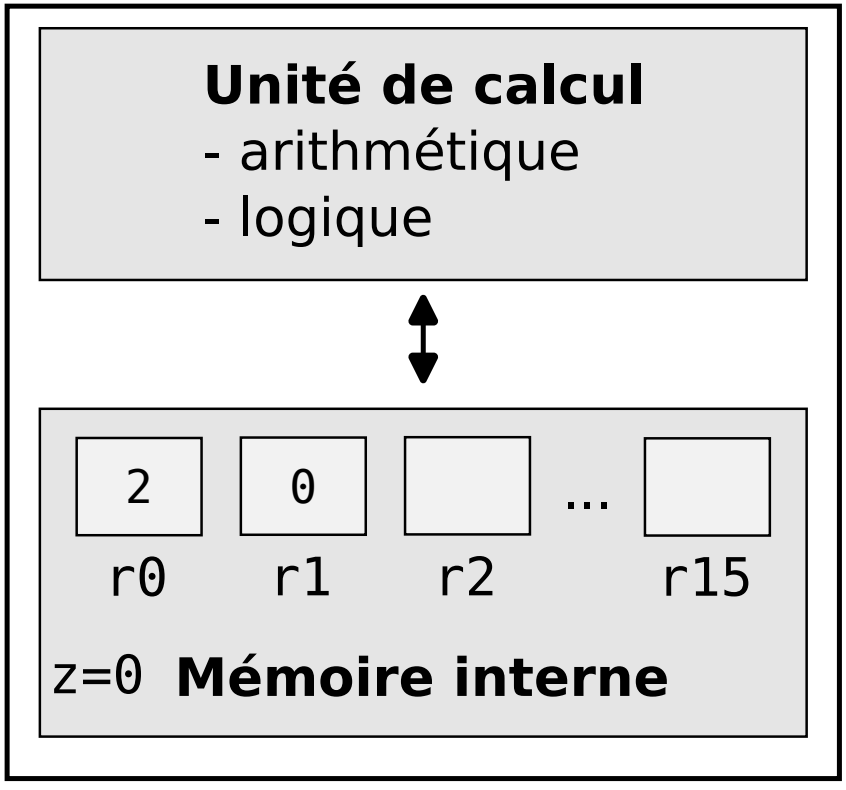

```
  0: mov [0x01], %r0
  1: mov %r0, %r1
  2: add $1, %r0
  3: cmp $2, %r0
  4: jnz 2
→ 5: syscall
```



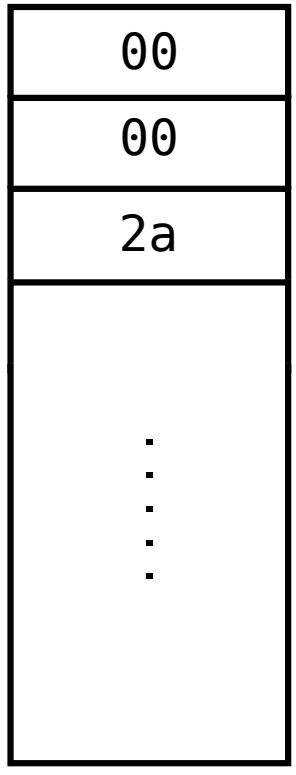
Processeur **Mémoire externe**

Les instructions sont les **opérations atomiques** qu'exécute le processeur
Elles peuvent être de plusieurs types :
- **déplacement de données** (mém. externe ↔ registres, entre registres)
- **calcul** (arithmétique, logique...)
- **saut** conditionnel ou non : modifie quelle est la prochaine instruction
- **autre** (on verra des exemples dans la suite de ce cours)

CPU : où sont les instructions ?



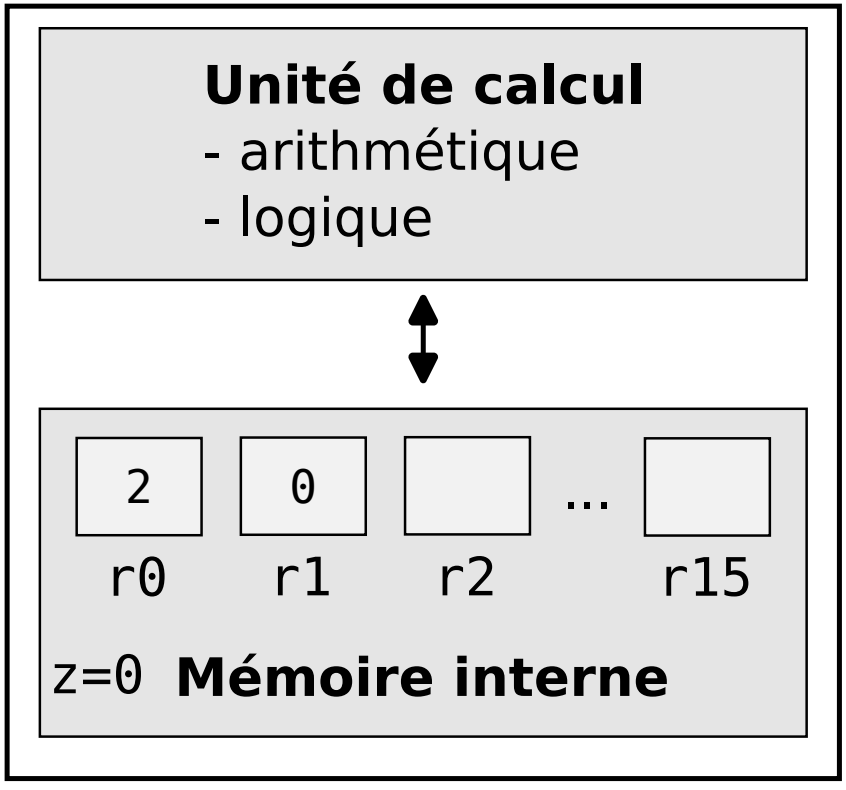

```
0: mov [0x01], %r0
1: mov %r0, %r1
2: add $1, %r0
3: cmp $2, %r0
4: jnz 2
5: syscall
```



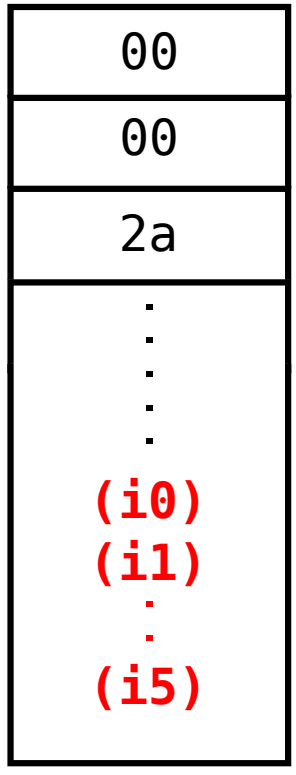
Processeur

Mémoire externe

CPU : où sont les instructions ?



Processeur

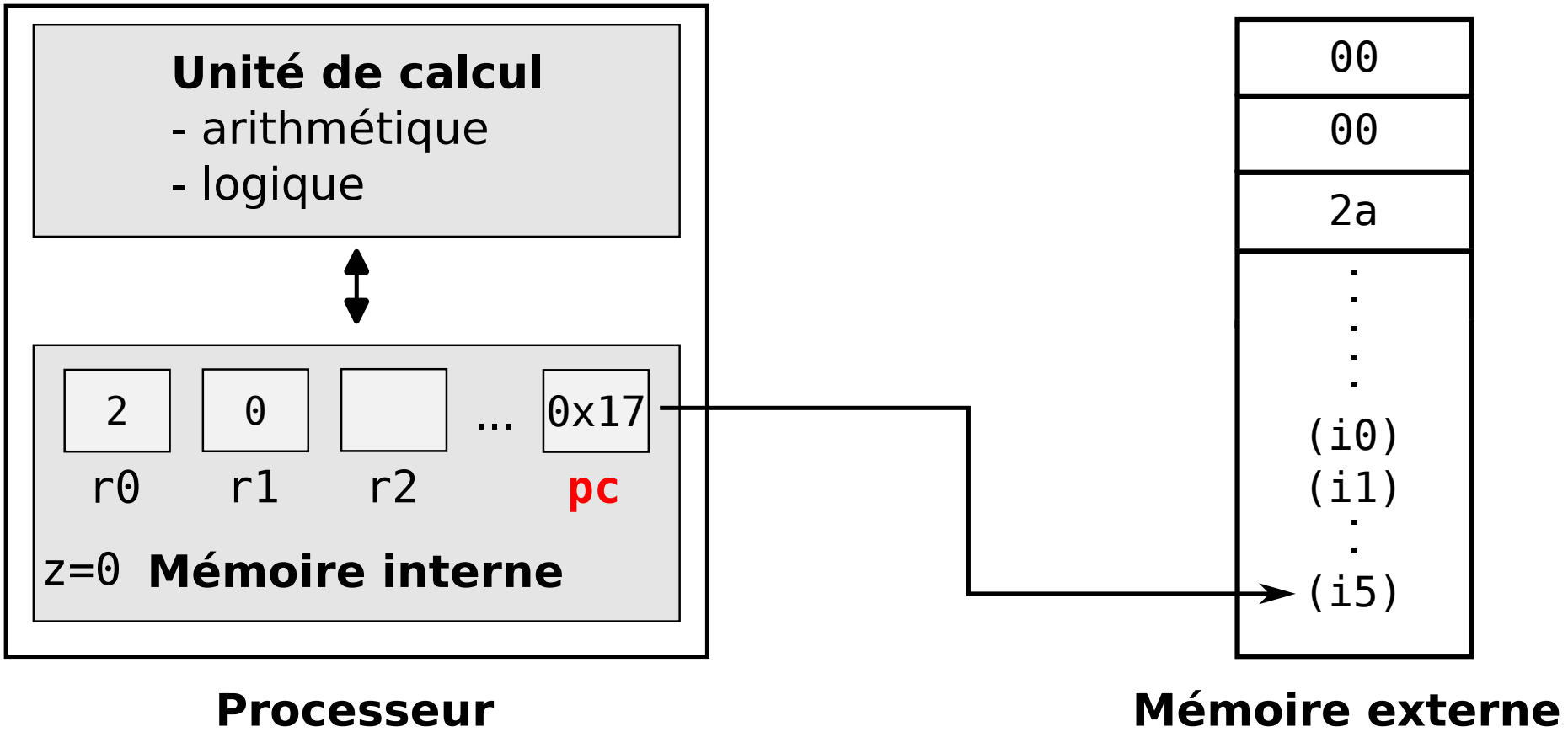


Mémoire externe

Les instructions sont elles-aussi en mémoire !

- Elles sont stockées sous forme de **langage machine**
- Le langage machine peut être vu comme un **encodage** des instructions généralement écrites en **langage d'assemblage**

CPU : où sont les instructions ?



Les instructions sont elles-aussi en mémoire !

- Elles sont stockées sous forme de **langage machine**
- Le langage machine peut être vu comme un **encodage** des instructions généralement écrites en **langage d'assemblage**
- L'adresse de l'instruction courante est stockée dans un **registre dédié** du processeur nommé *program counter* (**pc**)

Qu'est-ce qu'un programme ?

```c
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Vision conception/développement

- code source
- algorithmes
- composants logiciels

Problème : un processeur ne comprend aucun de ces langages !

Jetons un œil au fonctionnement d'un ordinateur pour éclaircir ça...

- La mémoire

- Le processeur

OK ! On peut revenir sur ce qu'est un programme :)

Vision système/architecture

Qu'est-ce qu'un programme ?

```
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Vision conception/développement

- code source
- algorithmes
- composants logiciels

Problème : un processeur ne comprend aucun de ces langages !

Un processeur ne parle que son **langage machine** : des intructions en binaire !

```
01010101010010001000100111100...10101
```

la fonction *index_of* est une suite de 74 octets

Vision système/architecture

Qu'est-ce qu'un programme ?

```c
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Vision conception/développement

- code source
- algorithmes
- composants logiciels

Problème : un processeur ne comprend aucun de ces langages !

Un processeur ne parle que son **langage machine** : des intructions en binaire !

```
01010101010010001000100111100...10101

554889e5897dec8975e8488955 ... fff5dc3
```

la fonction *index_of* est une suite de 74 octets ... que l'on peut représenter en base 16 (2 caractères = 1 octet)

Vision système/architecture

Qu'est-ce qu'un programme ?

```
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Vision conception/développement

- code source
- algorithmes
- composants logiciels

Problème : un processeur ne comprend aucun de ces langages !

Un processeur ne parle que son **langage machine** : des intructions en binaire !

```
01010101010010001000100111100...10101
```

la fonction *index_of* est une suite de 74 octets

```
554889e5897dec8975e8488955 ... fff5dc3
```

... que l'on peut représenter en base 16
(2 caractères = 1 octet)

```
554889e5897dec8975e8488955e0c745
fc00000000eb248b45fc4898488d1485
00000000488b45e04801d08b003945ec
75058b45fceb118345fc018b45fc3b45
e87cd4b8ffffffff5dc3
```

... que l'on peut représenter par bloc
(1 ligne = 16 octets)

Vision système/architecture

Qu'est-ce qu'un programme ?

```c
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Ce code machine = ce code C ?

```
554889e5897dec8975e8488955e0c745
fc00000000eb248b45fc4898488d1485
00000000488b45e04801d08b003945ec
75058b45fceb118345fc018b45fc3b45
e87cd4b8ffffffff5dc3
```

Qu'est-ce qu'un programme ?

```
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

Ce code machine = ce code C ?

Oui ! Chaque *bout* du code machine vient d'instructions/données du code C

```
554889e5897dec8975e8488955e0c745
fc00000000eb248b45fc4898488d1485
00000000488b45e04801d08b003945ec
75058b45fceb118345fc018b45fc3b45
e87cd4b8ffffffff5dc3
```

Qu'est-ce qu'un programme ?

```c
// Renvoie l'indice de la première occurrence
// de value dans le tableau array,
// ou -1 si value n'est pas dans array.
int index_of(int value, int size, int * array)
{
  for (int i = 0; i < size; ++i) {
    if (array[i] == value)
      return i;
  }
  return -1;
}
```

```
index_of:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    %esi, -24(%rbp)
        movq    %rdx, -32(%rbp)
        movl    $0, -4(%rbp)
        jmp     .L2
.L5:
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -32(%rbp), %rax
        addq    %rdx, %rax
        movl    (%rax), %eax
        cmpl    %eax, -20(%rbp)
        jne     .L3
        movl    -4(%rbp), %eax
        jmp     .L4
.L3:
        addl    $1, -4(%rbp)
.L2:
        movl    -4(%rbp), %eax
        cmpl    -24(%rbp), %eax
        jl      .L5
        movl    $-1, %eax
.L4:
        popq    %rbp
        ret
```

Ce code machine = ce code C ?

Oui ! Chaque *bout* du code machine vient d'instructions/données du code C

- Ce lien est plus clair en **assembleur** (code machine $\simeq$ assembleur)

```
554889e5897dec8975e8488955e0c745
fc00000000eb248b45fc4898488d1485
00000000488b45e04801d08b003945ec
75058b45fceb118345fc018b45fc3b45
e87cd4b8ffffffff5dc3
```

Langage de programmation VS langage machine

Il y a une différence de **niveau** entre ces deux langages.

Données

Programmation : variables et ensemble de variables structurées de haut niveau

Machine : registres et zones de mémoire entre deux adresses

Structuration du code

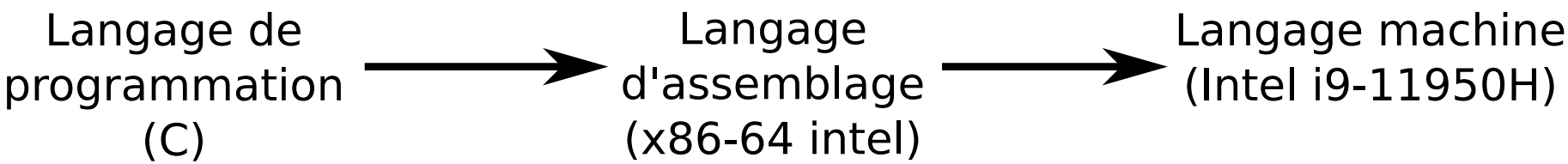
Programmation : instructions de haut niveau avec structures de contrôle (conditions, boucles, fonctions)

Machine : séquence d'instructions atomiques (calculs sur des registres très petits, déplacements de données en mémoire, saut conditionnel ou non vers une autre instruction atomique)

Langage de programmation $\rightarrow$ langage machine

Comment transformer un programme décrit dans un langage de programmation vers un programme machine est un domaine de l'informatique : **la compilation**

Comment faire ne sera pas détaillé ici
→ voir cours dédié d'une UE sur le sujet



Exécution directe, interprétation, JIT...

Le fonctionnement du CPU qu'on a vu s'applique à tout programme qui s'exécute, **quel que soit son langage de programmation** ! Il y a cependant plusieurs manières d'exécuter un programme.

Exécution directe, interprétation, JIT...

Le fonctionnement du CPU qu'on a vu s'applique à tout programme qui s'exécute, **quel que soit son langage de programmation** ! Il y a cependant plusieurs manières d'exécuter un programme.

Directe (celle qu'on a vue) : le programme est la séquence d'instructions qui s'exécute sur le processeur

- Courant pour programmes en C, C++, Rust...

Exécution directe, interprétation, JIT...

Le fonctionnement du CPU qu'on a vu s'applique à tout programme qui s'exécute, **quel que soit son langage de programmation** ! Il y a cependant plusieurs manières d'exécuter un programme.

Directe (celle qu'on a vue) : le programme est la séquence d'instructions qui s'exécute sur le processeur

- Courant pour programmes en C, C++, Rust...

Indirecte : le programme qui s'exécute sur le processeur n'est pas celui qu'on veut lancer, mais un programme intermédiaire chargé de transformer le code de votre programme en instructions à exécuter.

- Courant d'interpréter des programmes en Python, bash...
- Courant de compiler à la volée des programmes en Java, JavaScript...

SE et programme

Question

Quel rapport entre le SE et les programmes ?

SE et programme

Question

Quel rapport entre le SE et les programmes ?

Le SE est un programme qui s'exécute sur le CPU.

SE et programme

Question

Quel rapport entre le SE et les programmes ?

Le SE est ~~un~~ **LE** programme qui s'exécute sur le CPU !

SE et programme

Question

Quel rapport entre le SE et les programmes ?

Le SE est ~~un~~ **LE** programme qui s'exécute sur le CPU !

- C'est le **premier** programme à démarrer
- C'est le programme avec le **plus de privilèges** (accès direct total au matériel)
- C'est le programme qui **gère les autres programmes** (allocation de ressources, cycle de vie...)

Partage du CPU

Question

Si le SE s'exécute sur le CPU, comment est-il possible d'exécuter d'autres programmes sur le CPU ?

Partage du CPU

Question

Si le SE s'exécute sur le CPU, comment est-il possible d'exécuter d'autres programmes sur le CPU ?

Solution 1 : S'il y a plusieurs CPUs (cœurs), on peut réserver un CPU (cœur) au SE, et utiliser les autres pour lancer d'autres programmes.

Partage du CPU

Question

Si le SE s'exécute sur le CPU, comment est-il possible d'exécuter d'autres programmes sur le CPU ?

Solution 1 : S'il y a plusieurs CPUs (cœurs), on peut réserver un CPU (cœur) au SE, et utiliser les autres pour lancer d'autres programmes.

Problèmes/limites

- Gâchis de ressources : un CPU (cœur) est réservé alors qu'il ne fait rien la plupart du temps
- Ça ne marche pas quand il n'y a qu'un seul CPU (cœur)

Partage du CPU (2)

Question

Si le SE s'exécute sur le CPU, comment est-il possible d'exécuter d'autres programmes sur le CPU ?

Partage du CPU (2)

Question

Si le SE s'exécute sur le CPU, comment est-il possible d'exécuter d'autres programmes sur le CPU ?

Solution 2 : Le SE peut enregistrer son état courant d'exécution, exécuter un autre programme, puis quand l'autre programme a terminé, restaurer son état d'exécution précédent et continuer de s'exécuter.

Partage du CPU (2)

Question

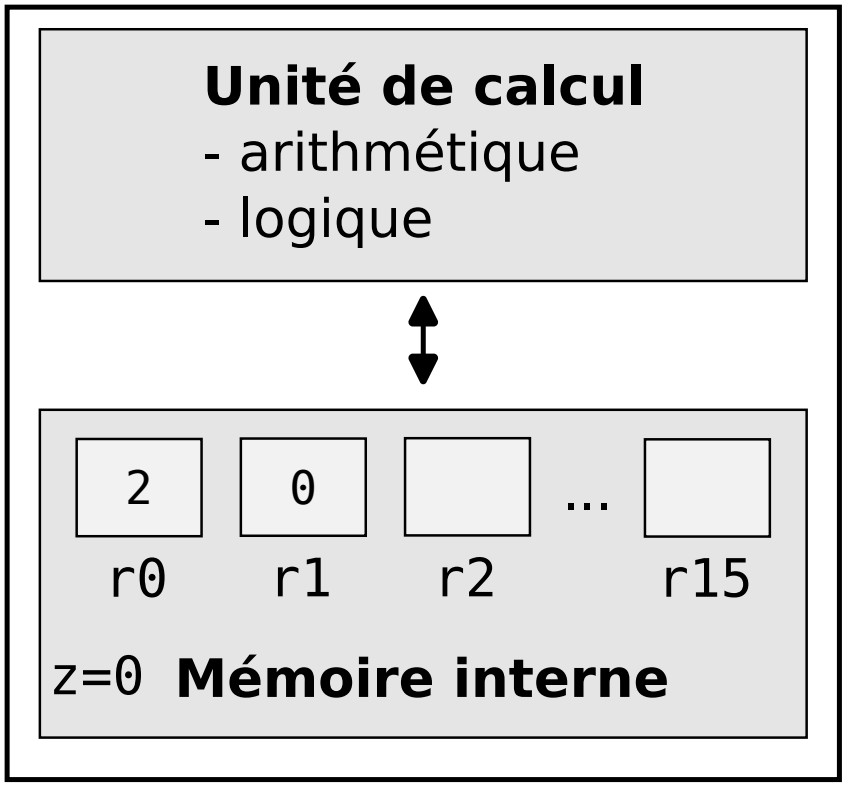
Si le SE s'exécute sur le CPU, comment est-il possible d'exécuter d'autres programmes sur le CPU ?

Solution 2 : Le SE peut enregistrer son état courant d'exécution, exécuter un autre programme, puis quand l'autre programme a terminé, restaurer son état d'exécution précédent et continuer de s'exécuter.

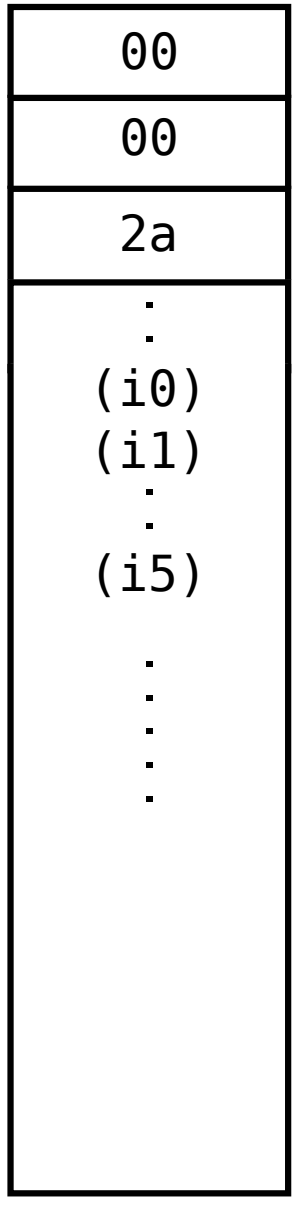
Ça peut marcher mais...

- Quel est cet "état courant d'exécution" ?
- Où enregistrer cet état ?

CPU : enregistrer et restaurer son état d'exécution



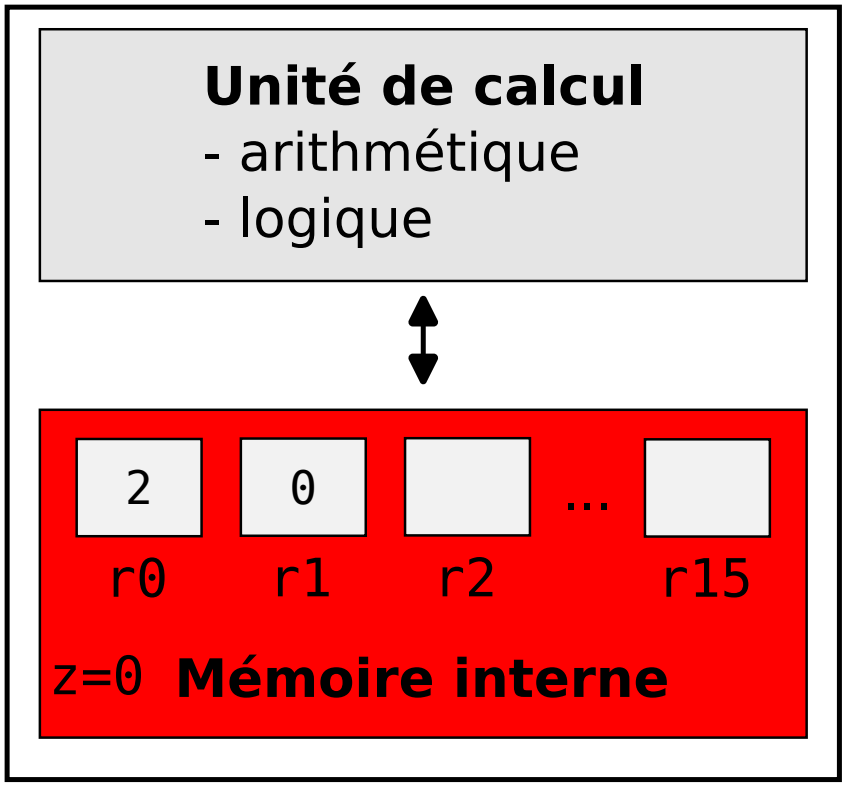
Processeur



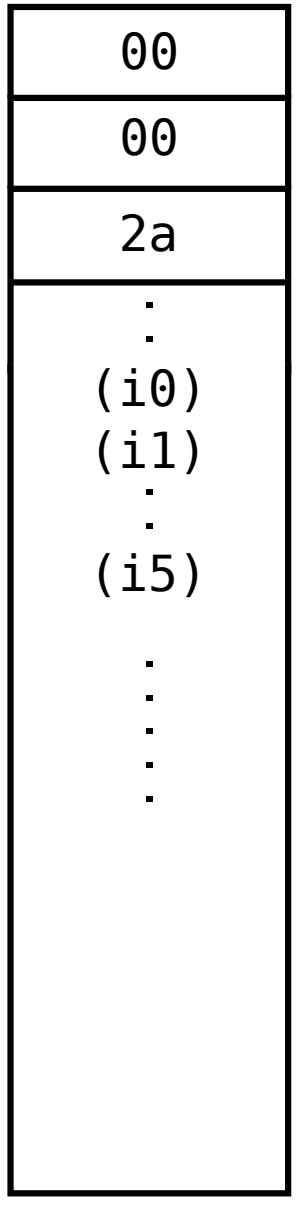
Mémoire externe

Question 1. Quel est l'état courant d'exécution du CPU ?

CPU : enregistrer et restaurer son état d'exécution



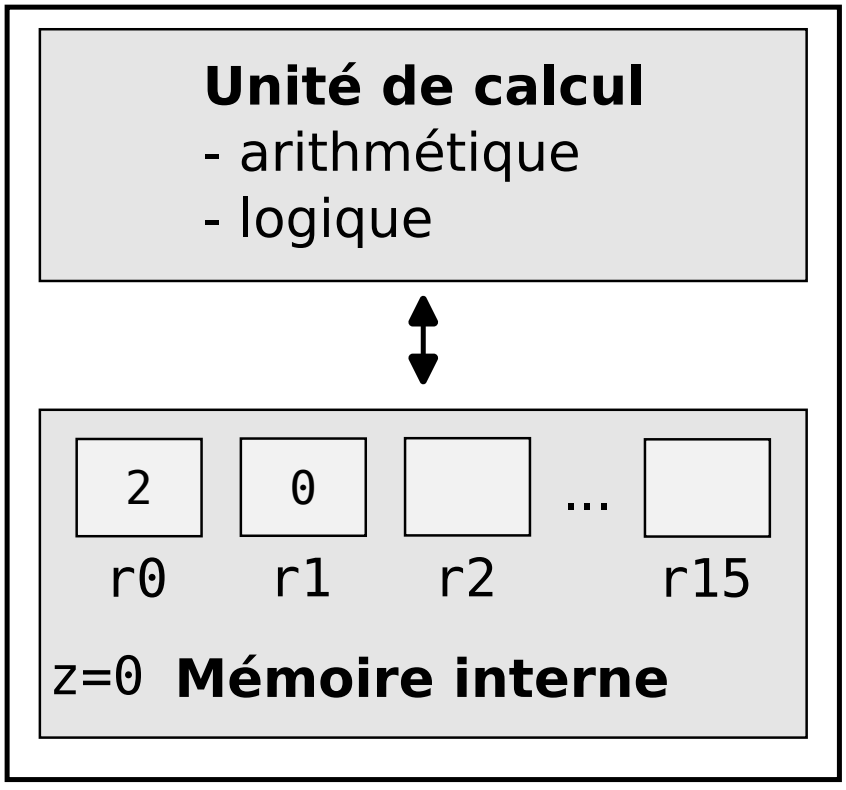
Processeur



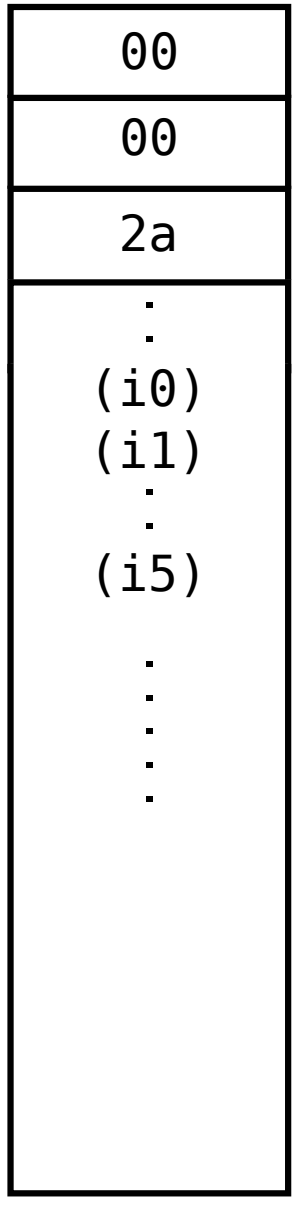
Mémoire externe

Question 1. Quel est l'état courant d'exécution du CPU ?
- La mémoire interne du CPU

CPU : enregistrer et restaurer son état d'exécution



Processeur

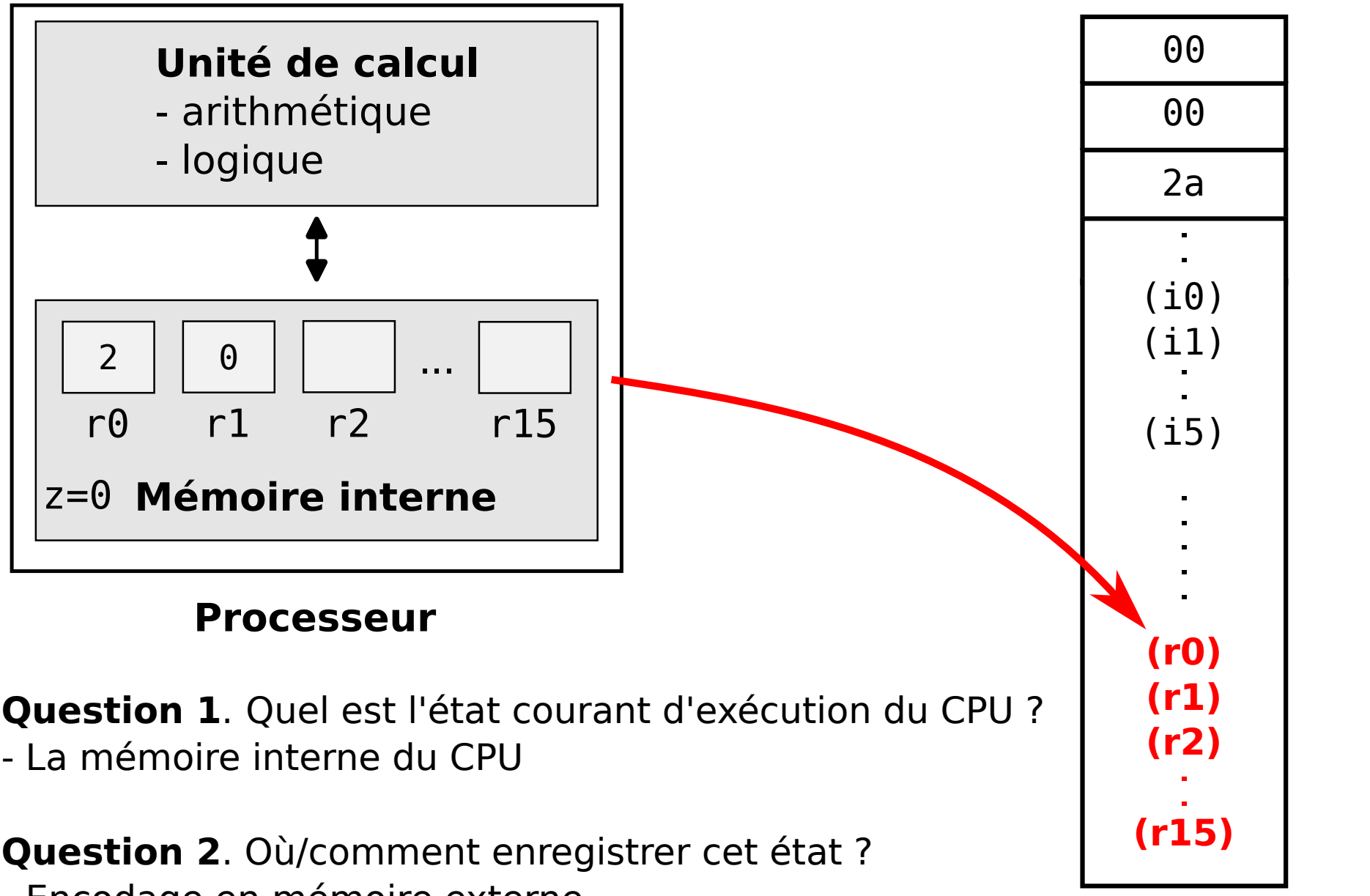


Mémoire externe

Question 1. Quel est l'état courant d'exécution du CPU ?
- La mémoire interne du CPU

Question 2. Où/comment enregistrer cet état ?

CPU : enregistrer et restaurer son état d'exécution



Question 1. Quel est l'état courant d'exécution du CPU ?
- La mémoire interne du CPU

Question 2. Où/comment enregistrer cet état ?
- Encodage en mémoire externe

Commutation de contexte (*context switch*)

Nom de ce mécanisme

Le mécanisme de sauvegarde de l'état d'un programme en cours d'exécution, de telle sorte qu'on puisse reprendre plus tard l'exécution du programme, s'appelle une **commutation de contexte** ou *context switch*.

Partage du CPU (3)

Ça peut marcher mais...

Si le programme peut écrire partout en mémoire externe, il peut rendre les autres programmes (dont le SE) invalides !

- Méchamment : en changeant leurs données
- Très méchamment : en changeant leurs instructions

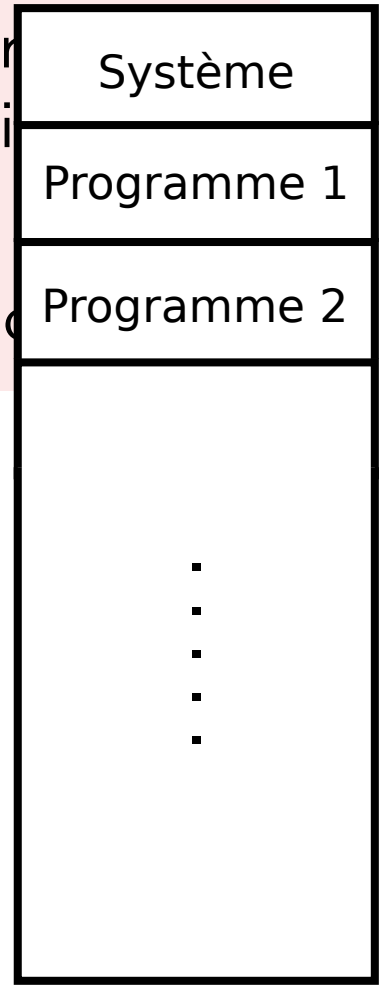
Partage du CPU (3)

Ça peut marcher mais...

Si le programme peut écrire partout en mémoire exter
peut rendre les autres programmes (dont le SE) invali

- Méchamment : en changeant leurs données
- Très méchamment : en changeant leurs instructi

- Le SE peut allouer une zone de mémoire externe à chaque programme qui n'est pas le SE
- Le SE peut réduire les privilèges des programmes (non SE) pour leur interdire d'accéder à la mémoire en dehors de leur zone autorisée



Partage du CPU (4)

Ça peut marcher mais...

Si un programme ne termine jamais, il va garder le CPU pour toujours !

Partage du CPU (4)

Ça peut marcher mais...

Si un programme ne termine jamais, il va garder le CPU pour toujours !

- En plus de sa boucle d'exécution normale (récupérer la prochaine instruction et l'exécuter), le CPU est **interrompu** quand il reçoit un signal d'autres composants électroniques. On parle d'**interruption matérielle**

Partage du CPU (4)

Ça peut marcher mais...

Si un programme ne termine jamais, il va garder le CPU pour toujours !

- En plus de sa boucle d'exécution normale (récupérer la prochaine instruction et l'exécuter), le CPU est **interrompu** quand il reçoit un signal d'autres composants électroniques. On parle d'**interruption matérielle**
- Le SE peut choisir en avance le code qui s'exécute lorsqu'une interruption matérielle est levée (c'est l'adresse d'une instruction en mémoire)

Partage du CPU (4)

Ça peut marcher mais...

Si un programme ne termine jamais, il va garder le CPU pour toujours !

- En plus de sa boucle d'exécution normale (récupérer la prochaine instruction et l'exécuter), le CPU est **interrompu** quand il reçoit un signal d'autres composants électroniques. On parle d'**interruption matérielle**
- Le SE peut choisir en avance le code qui s'exécute lorsqu'une interruption matérielle est levée (c'est l'adresse d'une instruction en mémoire)
- Le SE peut, juste avant d'exécuter un programme, dire à un composant électronique d'horloge de lever une interruption au temps qu'il souhaite.

Partage du CPU (4)

Ça peut marcher mais...

Si un programme ne termine jamais, il va garder le CPU pour toujours !

- En plus de sa boucle d'exécution normale (récupérer la prochaine instruction et l'exécuter), le CPU est **interrompu** quand il reçoit un signal d'autres composants électroniques. On parle d'**interruption matérielle**
- Le SE peut choisir en avance le code qui s'exécute lorsqu'une interruption matérielle est levée (c'est l'adresse d'une instruction en mémoire)
- Le SE peut, juste avant d'exécuter un programme, dire à un composant électronique d'horloge de lever une interruption au temps qu'il souhaite.

Ce mécanisme permet de garantir qu'un programme ne s'exécute jamais plus qu'un certain temps

Partage du CPU : ça marche :)

Implémenter l'ensemble des mécanismes qu'on vient de voir permet d'exécuter le SE et plusieurs programmes utilisateurs sur le même CPU

- Le SE fait une commutation de contexte pour passer d'un programme à un autre sur le CPU
- Le SE alloue une zone mémoire dédiée à chaque programme en cours d'exécution
- Le SE passe en mode privilégié sur le CPU quand il prend la main, et passe en mode restreint quand il donne la main à un programme utilisateur. Ce mode empêche au programme utilisateur d'accéder à certaines zones mémoire et de faire des accès directs au matériel
- Le SE est sûr qu'il reprendra la main après une commutation de contexte grâce à l'horloge matérielle

Programme et processus

Programme : définition

C'est un logiciel destiné à être exécuté sur un ordinateur

- Un programme *machine* est une séquence d'instructions (bas niveau, du CPU) et de données
- Un programme *source* est le code source du logiciel écrit dans un ou des langages de programmation

Processus : définition

C'est un programme en cours d'exécution

Un programme est une entité **passive** souvent stockée sur un SSD ou un disque dur. Un processus est une instanciation **active** d'un programme qui manipule la mémoire vive et les registres du CPU.

Données pour gérer les processus par le SE

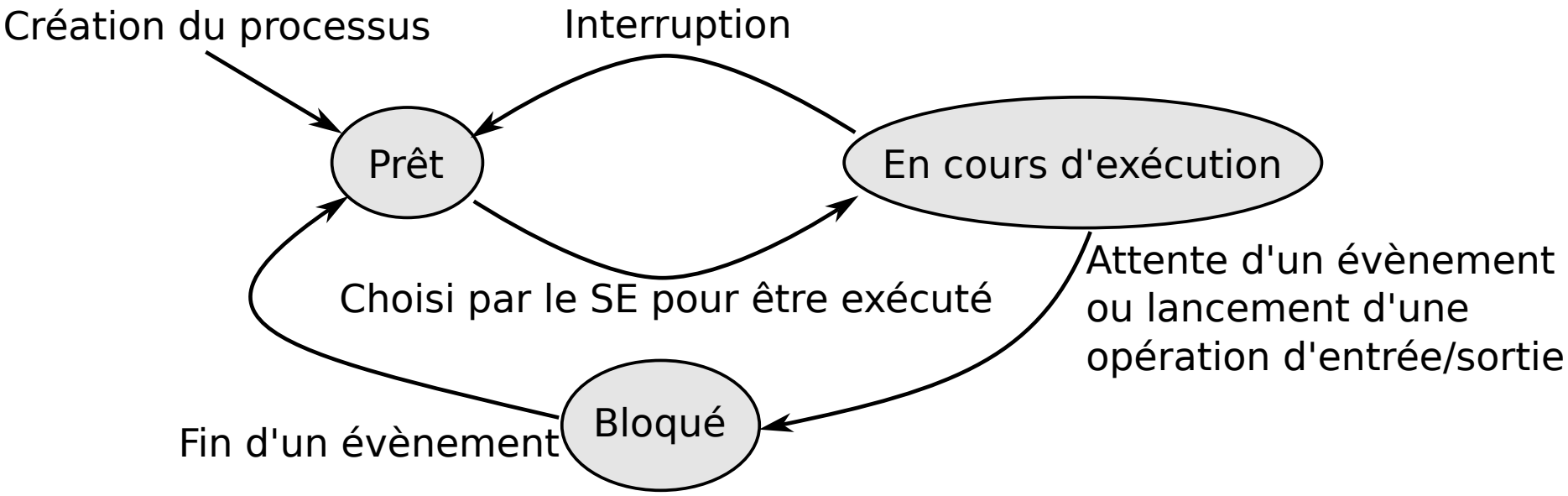
Le SE stocke toute l'information de gestion de l'exécution d'un programme dans une structure de données nommée *Process Control Block* (PCB). Cette structure contient notamment :

- L'identifiant unique du processus
- L'état du processus (en cours d'exécution, en attente...)
- Les privilèges du processus (processus SE ou utilisateur)
- De quoi enregistrer/charger le processus (registres CPU)
- Les zones mémoire auxquelles le processus peut accéder
- Des statistiques comptables (temps utilisé du CPU, depuis quand le processus existe...)
- Diverses informations sur les autres ressources allouées au processus (fichiers, cartes réseau, cartes graphique...)

États d'un processus

Un processus change d'état au cours de son exécution

- Prêt : peut être exécuté (attend d'être assigné à un CPU)
- En cours d'exécution : en train d'occuper un CPU
- Bloqué : en attente qu'un certain évènement se produise pour pouvoir s'exécuter (*e.g.*, libération d'une ressource autre que le CPU, fin d'une lecture disque, réception d'un message réseau...)



Le mécanisme d'appel système (*syscall*)

C'est la manière principale qu'utilise un processus pour demander une action ou une information au SE.

1. Le processus écrit en mémoire sa requête pour le SE
2. Le processus génère une **interruption logicielle**, pour que le SE reprenne la main sur le CPU (*e.g.*, via l'instruction dédiée `syscall` en x86-64)
3. Le SE prend la main et passe en mode privilégié
4. Le SE exécute son code de gestion de l'appel système. Le SE peut faire certaines actions puis (potentiellement plus tard) écrire une réponse en mémoire pour le processus.
5. (Peut-être plus tard) le SE décide d'assigner un CPU au processus, sauvegarde l'état du SE, passe le CPU en mode restreint puis restaure l'état du processus sur le CPU.
6. Le processus s'exécute et lit en mémoire la réponse du SE.

Cycle de vie d'un processus

Un processus peut demander la création d'un nouveau processus par un syscall (*e.g.*, `fork` ou `clone`). Chaque processus a un processus parent. Le processus `init` est lancé par le SE à la fin du démarrage.

Un processus peut faire de nombreux syscalls pour interagir avec son environnement, ou juste s'exécuter sans interagir avec le SE.

Pour se terminer, un processus fait un syscall `exit`. Le SE prend la main, récupère les ressources qui étaient allouées au processus puis détruit le processus.

Comportement du SE et ordonnancement

Le SE est un programme essentiellement **réactif** : il s'exécute quand une interruption matérielle ou logicielle est levée, puis **décide** à quel processus donner la main sur le CPU.

Le SE utilise à ce moment précis un **algorithme d'ordonnancement** pour choisir quel est le prochain processus qui doit s'exécuter.

Il existe de très nombreux algorithmes d'ordonnancement. Ceux utilisés à ce niveau se basent le plus souvent sur ces informations pour prendre leur décision :

- Quels processus sont *prêts* ? Depuis quand ?
- Quels processus ont le moins utilisé le CPU ?

Ordonnancement : propriétés recherchées

L'algorithme d'ordonnancement de processus utilisé par un SE est **critique** pour optimiser l'utilisation des ressources.

Plusieurs propriétés sont désirées :

- Performance : maximiser l'utilisation du CPU pour des opérations *utiles* (calculs des processus utilisateur). L'algorithme doit être très rapide car le temps qu'il prend sur le CPU est *inutile*
- Équité : on veut que tous les processus soient exécutés, avec une certaine équité, selon leurs priorités
- Réactivité : les processus interactifs ont besoin d'être souvent exécutés pour répondre à des évènements
- ... et bien d'autres selon le contexte dans lequel on est

Selon les propriétés précises que l'on cherche, on va choisir un algorithme plutôt qu'un autre.

Quantum de temps

Pour éviter qu'un processus utilisateur garde le CPU à tout jamais, le SE utilise presque toujours l'horloge matérielle pour être sûr de reprendre la main à une certaine date.

En terme d'algorithmes d'ordonnancement, on parle de **préemption** lorsqu'un processus peut être arrêté au cours de son exécution. On parle également de **quantum de temps** : c'est le temps maximum que le SE donne à une exécution consécutive du CPU à un processus. Si ce temps est dépassé, une interruption matérielle et levée et le SE reprend la main.

Politiques classiques d'ordonnancement

FCFS (pour *first-come, first-served*) est un algorithme sans préemption qui exécute les processus selon leur ordre d'arrivée.

Round Robin (tourniquet) exécute les processus avec préemption avec le même quantum de temps. Les processus prêts sont placés dans une file, ce qui permet de les exécuter les uns après les autres et de garantir une certaine équité.

On peut utiliser de nombreux critères (plus ou moins douteux) pour choisir quel processus exécuter parmi ceux prêts :

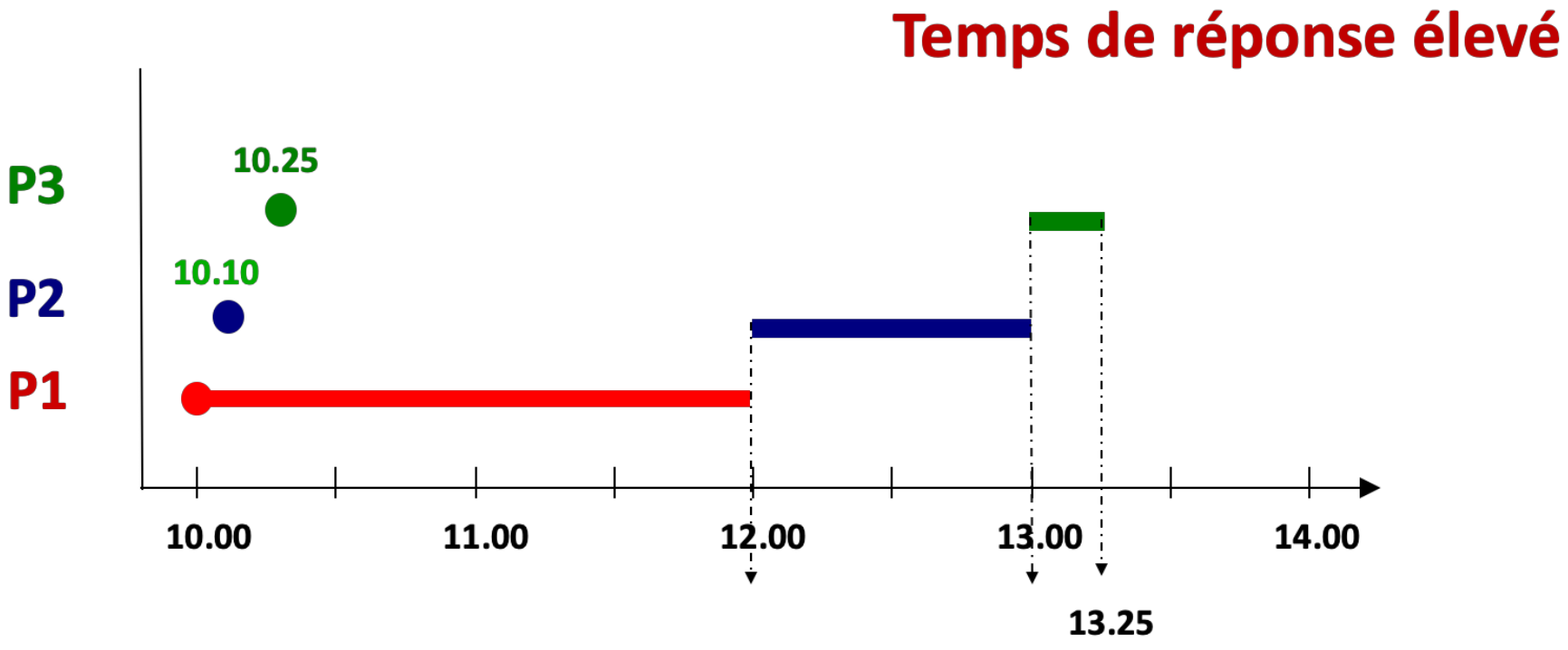
- Celui qui a le moins utilisé le CPU cette dernière minute
- Celui qui a le plus petit numéro de processus
- ...

Exemples d'ordonnancement

- 3 processus
  - P1 — Demande à s'exécuter à 10.00 — Durée : 2.00
  - P2 — Demande à s'exécuter à 10.10 — Durée : 1.00
  - P3 — Demande à s'exécuter à 10.25 — Durée : 0.25
- Hypothèse
  - Temps de changement de contexte `supposé` nul

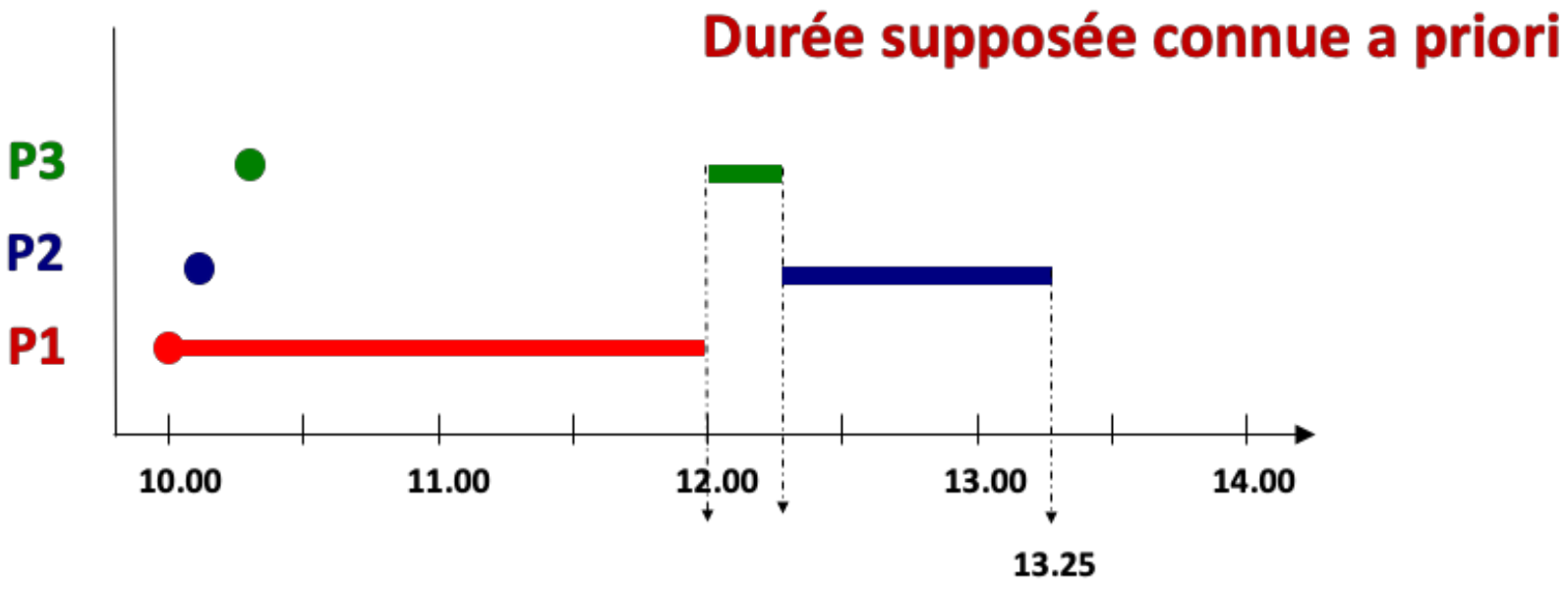
Politique d'ordonnancement FCFS ou FIFO (First Come, First Served)

- Le premier processus qui arrive est élu
- Pas de préemption



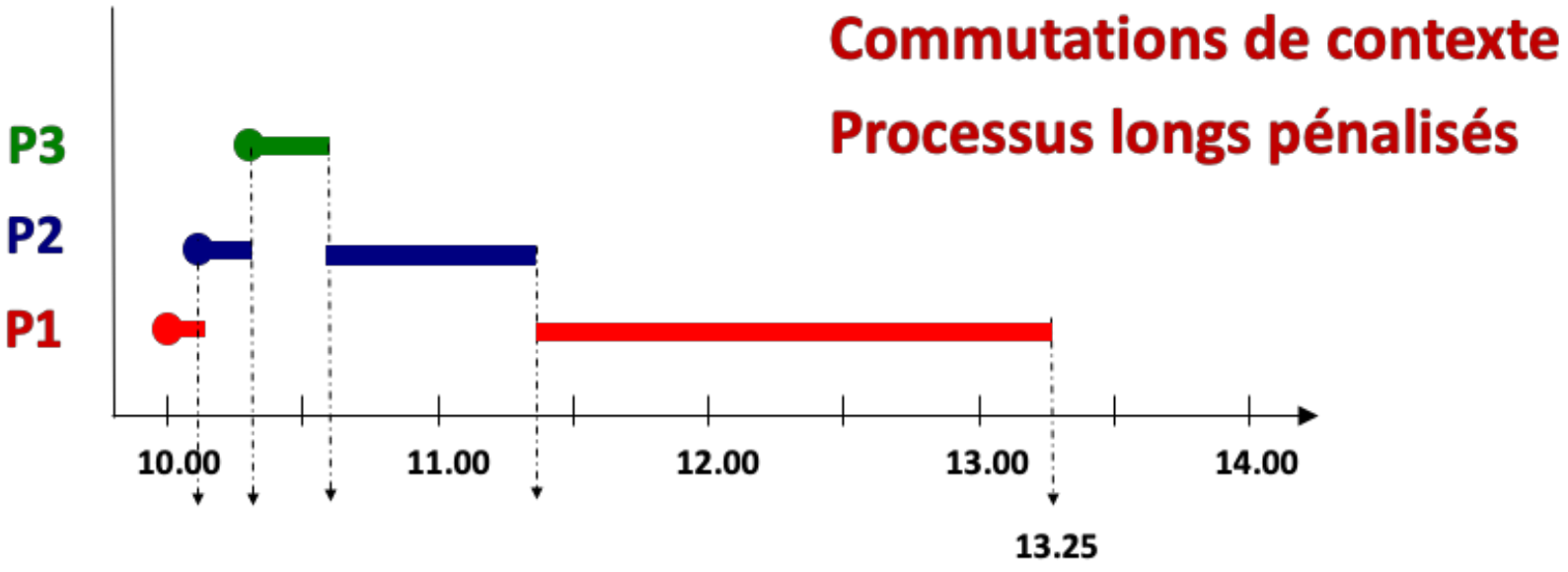
Politique d'ordonnancement SJF (Shortest Job First)

- Processus le plus court servi en premier
- Pas de préemption



Politique d'ordonnancement SRT (Shortest Remaining Time)

- Préemption
- Processus interrompu si une plus courte demande



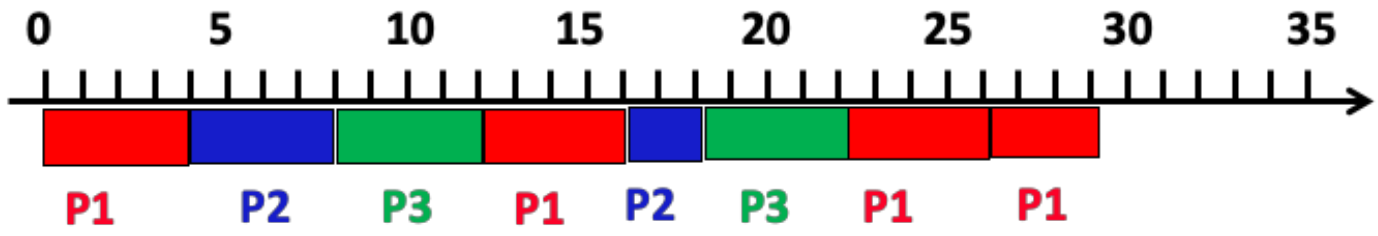
Politique d'ordonnancement Round Robin I

- Attribution d'un quantum de temps (entre 10 et 100 millisecondes)
- Préemption
- La libération de l'UC peut être due à :
  - la fin du quantum de temps
  - la fin du cycle d'UC

Politique d'ordonnancement Round Robin II

Processus	Durée de cycle
P1	15
P2	6
P3	8

Quantum de temps = 4 unités

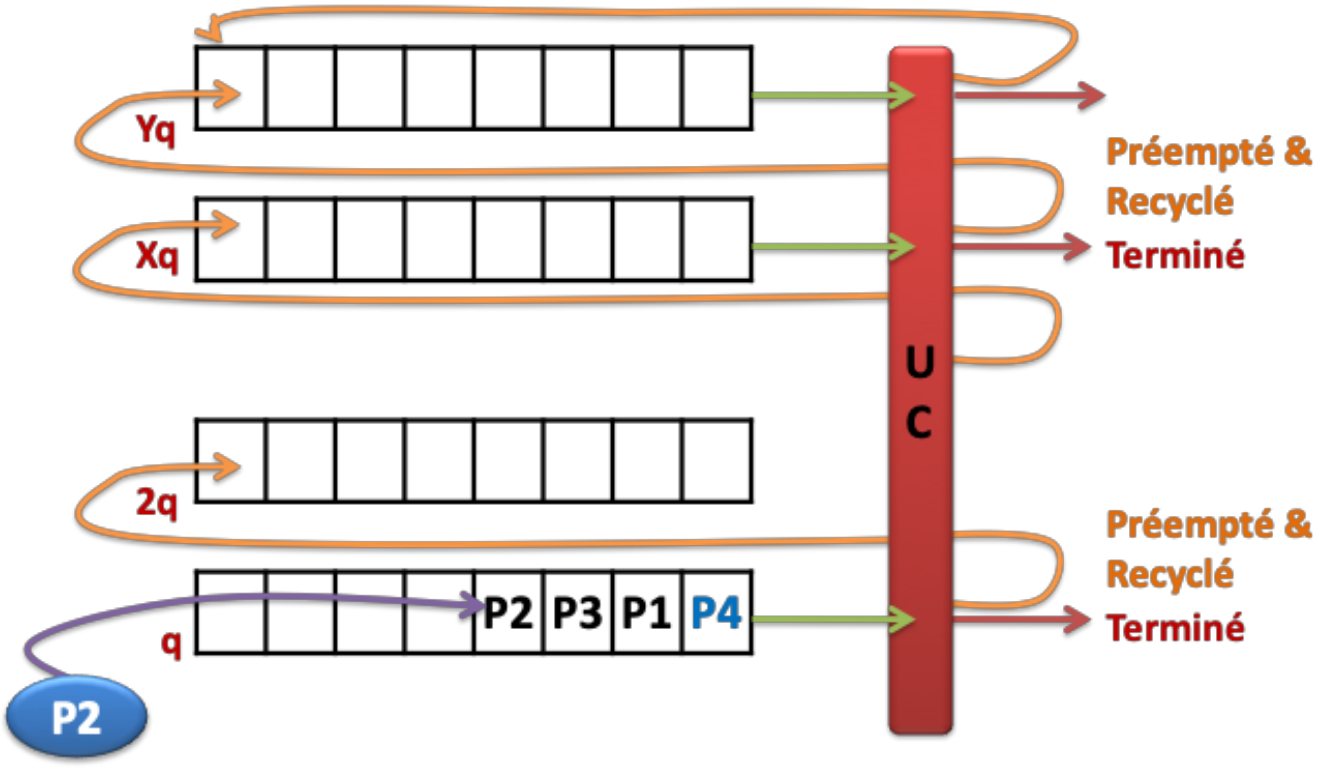


Politique d'ordonnancement Round Robin III

Évaluation

- Algorithme adapté à un système interactif
- Si le quantum de temps
  - est trop court : nombreuses commutations de contextes donc ralentissement des processus
  - est trop long : équivalent à FCFS
- Les mesures faites montrent que le quantum devrait être choisi de telle sorte que 80% des cycles d'UC se terminent avant la fin du quantum
- Favorise les processus courts

Files à priorités (UNIX)



- Quantum
- Les processus `prêts` sont mis dans des files de priorité, réévaluées périodiquement quand les processus ont eu du temps CPU (diminution de la priorité en fonction du passé).

Exemple I

- supposons 3 processus : P1, P2 et P3
- le quantum de temps est égal à 5 UT (unités de temps)
- au début, P1 a été élu et est exécuté par l'UC (unité centrale)

déroulement de P1

- utilisation de l'UC pendant 4 UT
- utilisation d'une E/S disque pendant 8 UT
- UC pd 8 UT
- fin

Exemple II

déroulement de P2

- UC pd 3 UT
- E/S réseau pendant 15 UT
- UC pendant 2 UT
- fin

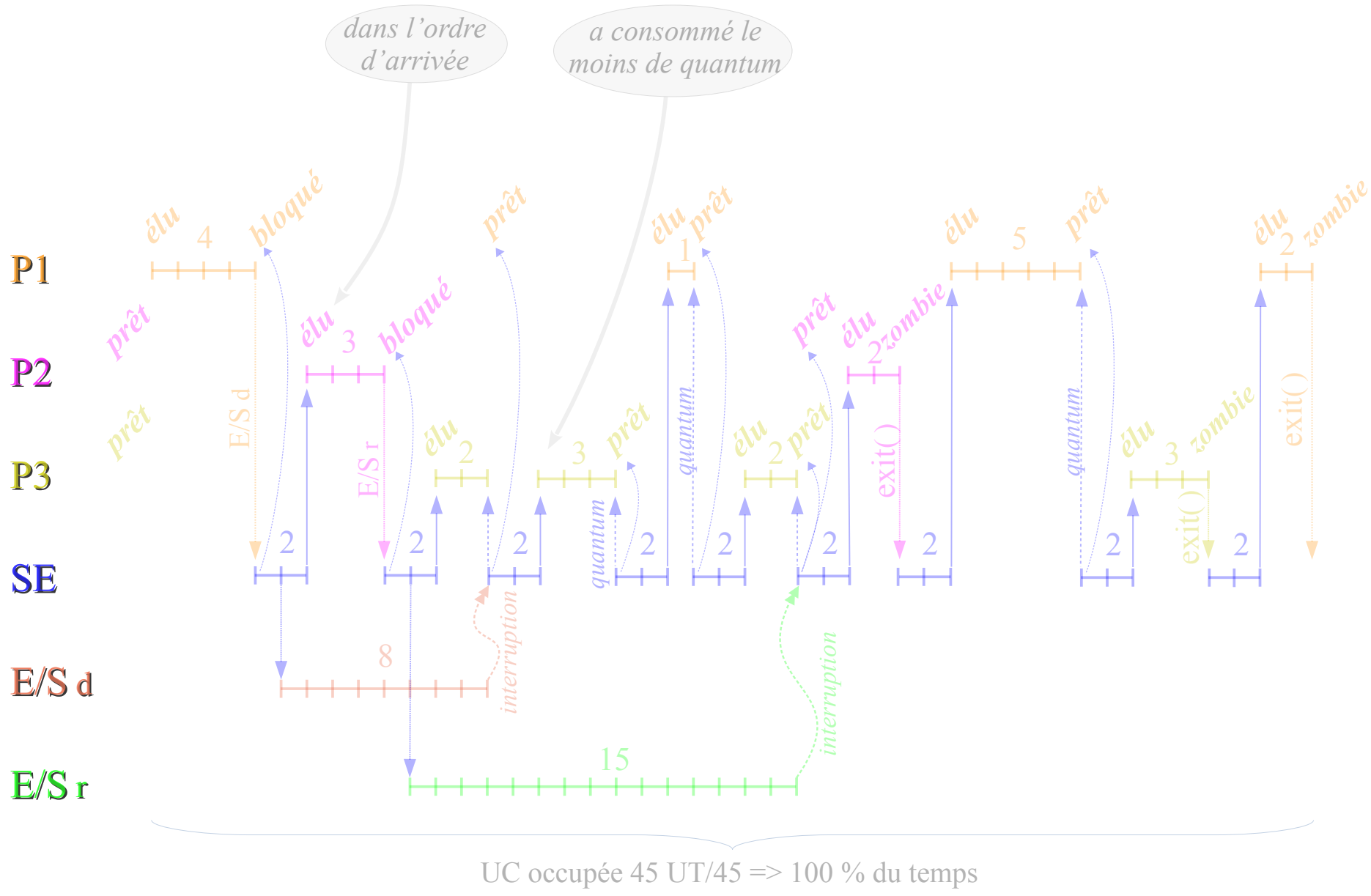
déroulement de P3

- UC pendant 10 UT
- fin

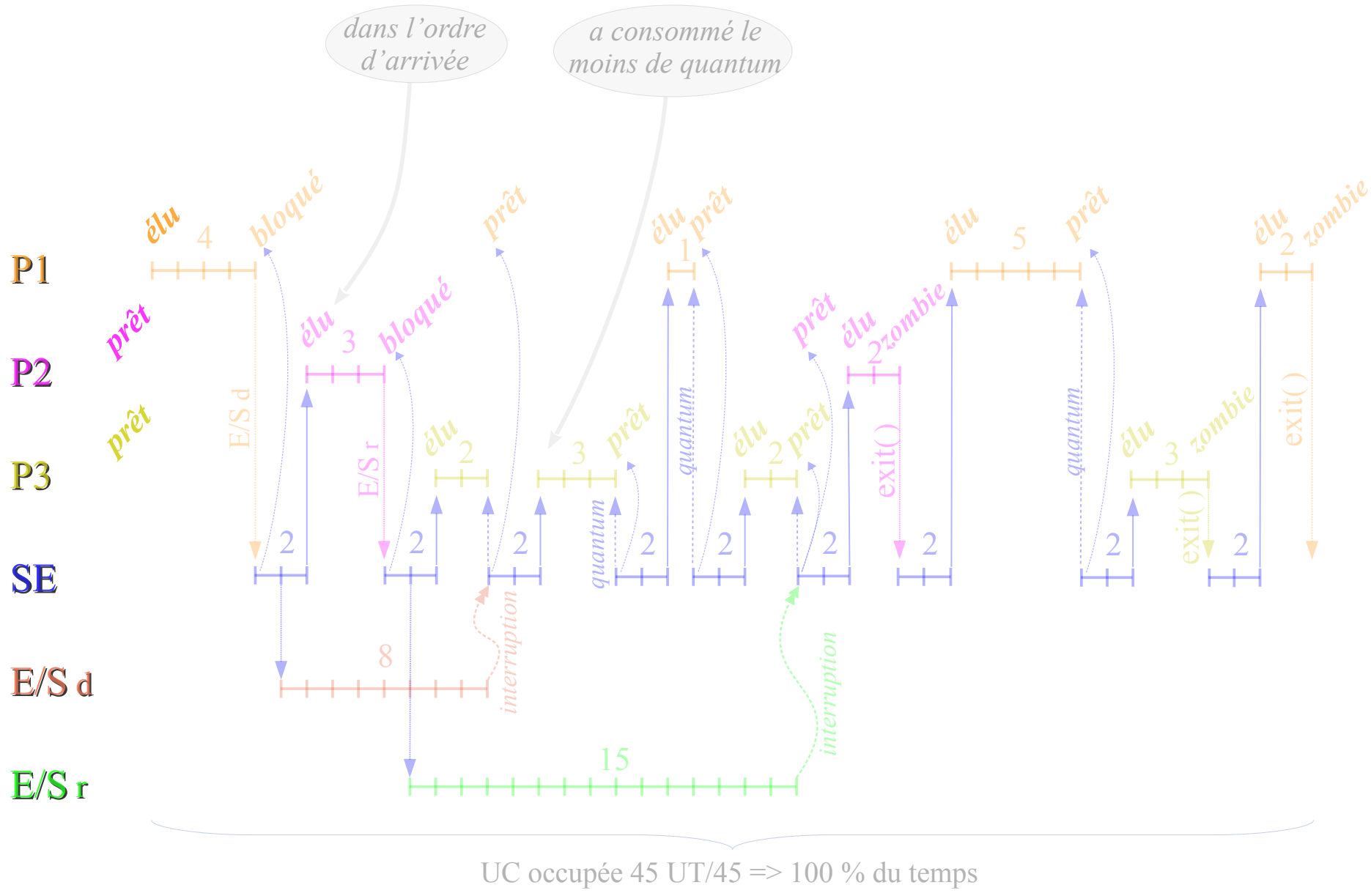
Exemple III

- on suppose que le SE prend 2 UT pour faire le changement de contexte :
  - dès qu'il y a une E/S (on suppose qu'elle commence après 1 UT)
  - dès qu'il doit décider qui sera le prochain processus élu (s'il y a un processus prêt)

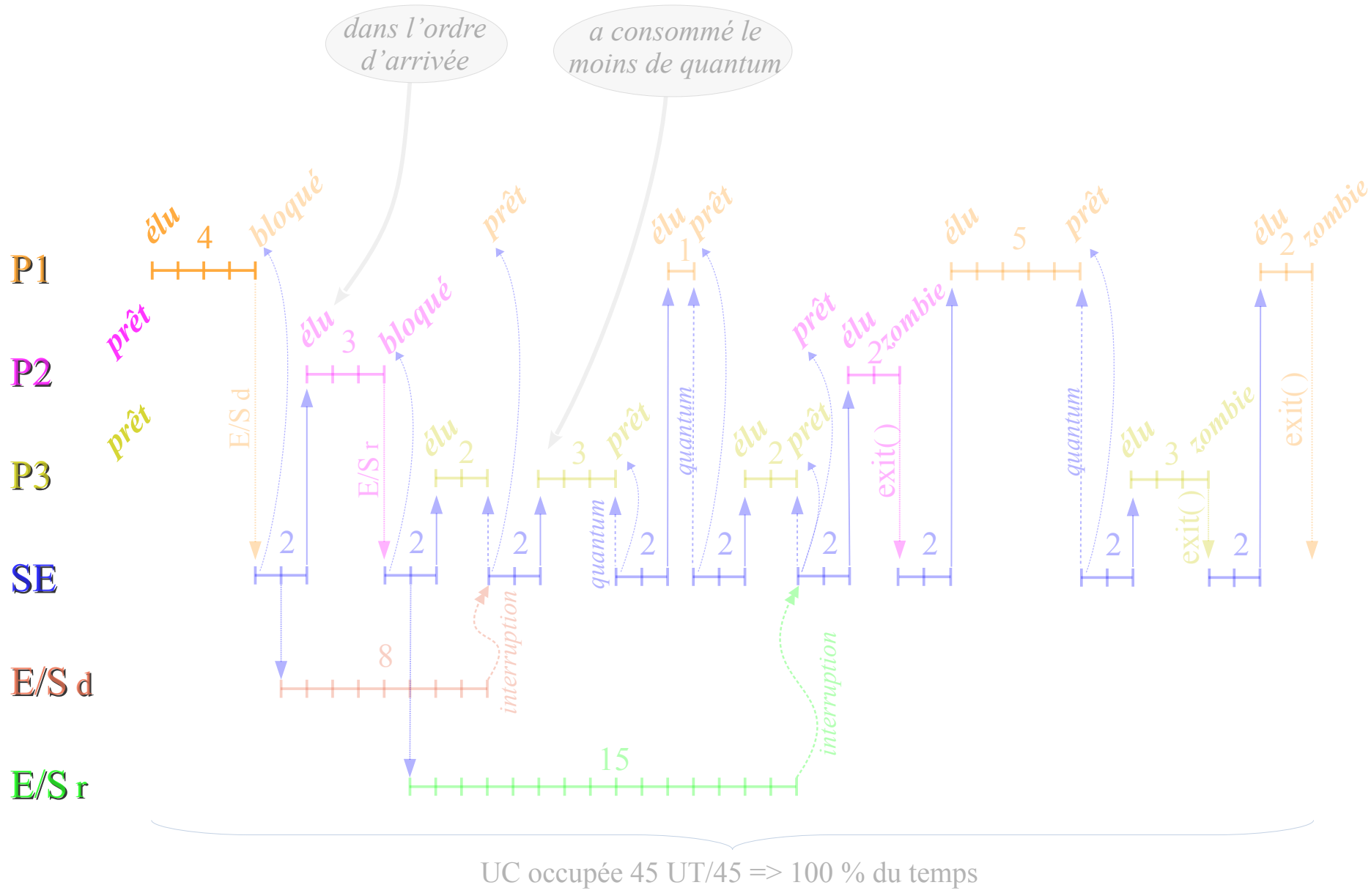
Chronogramme



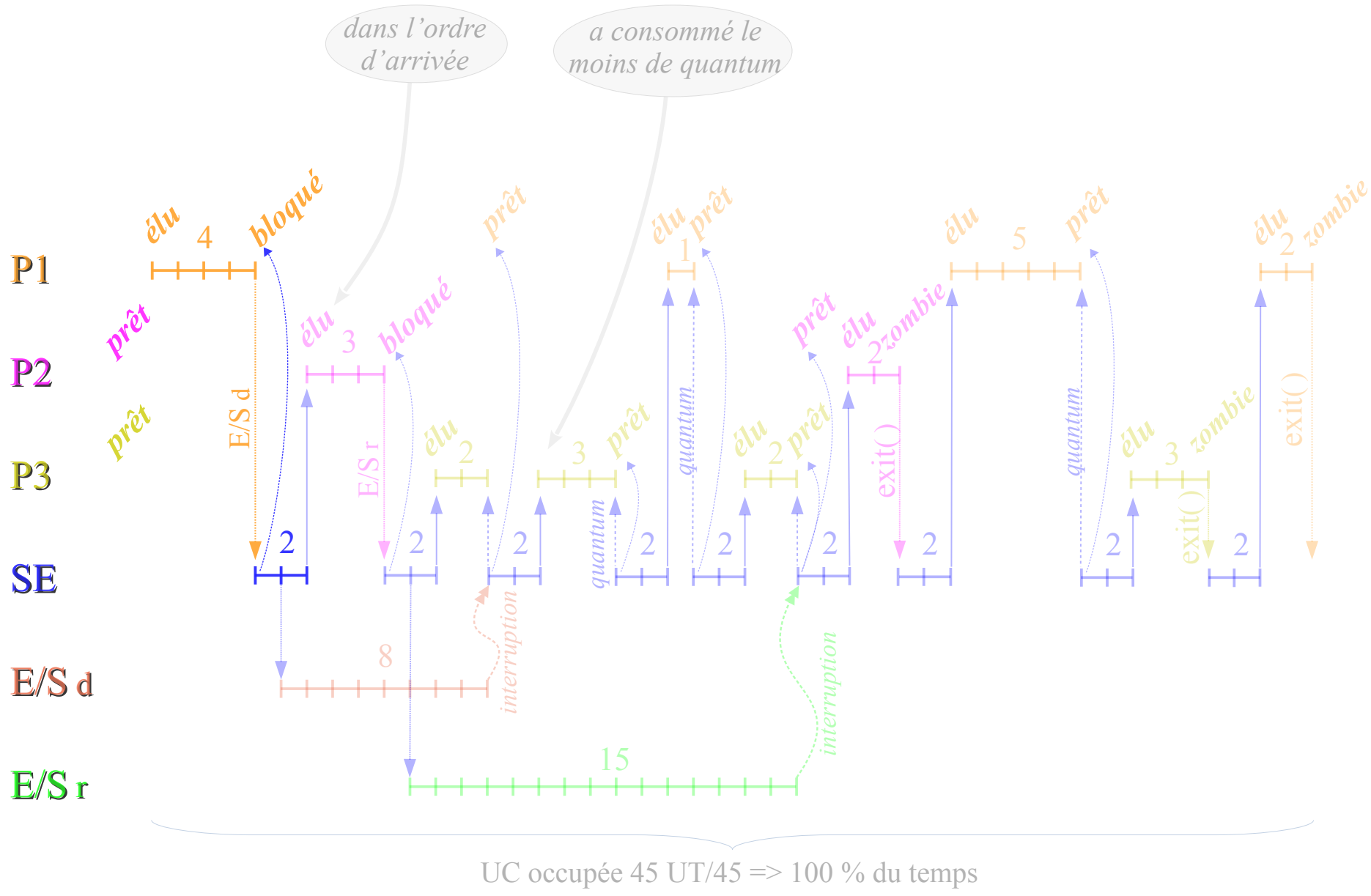
Chronogramme



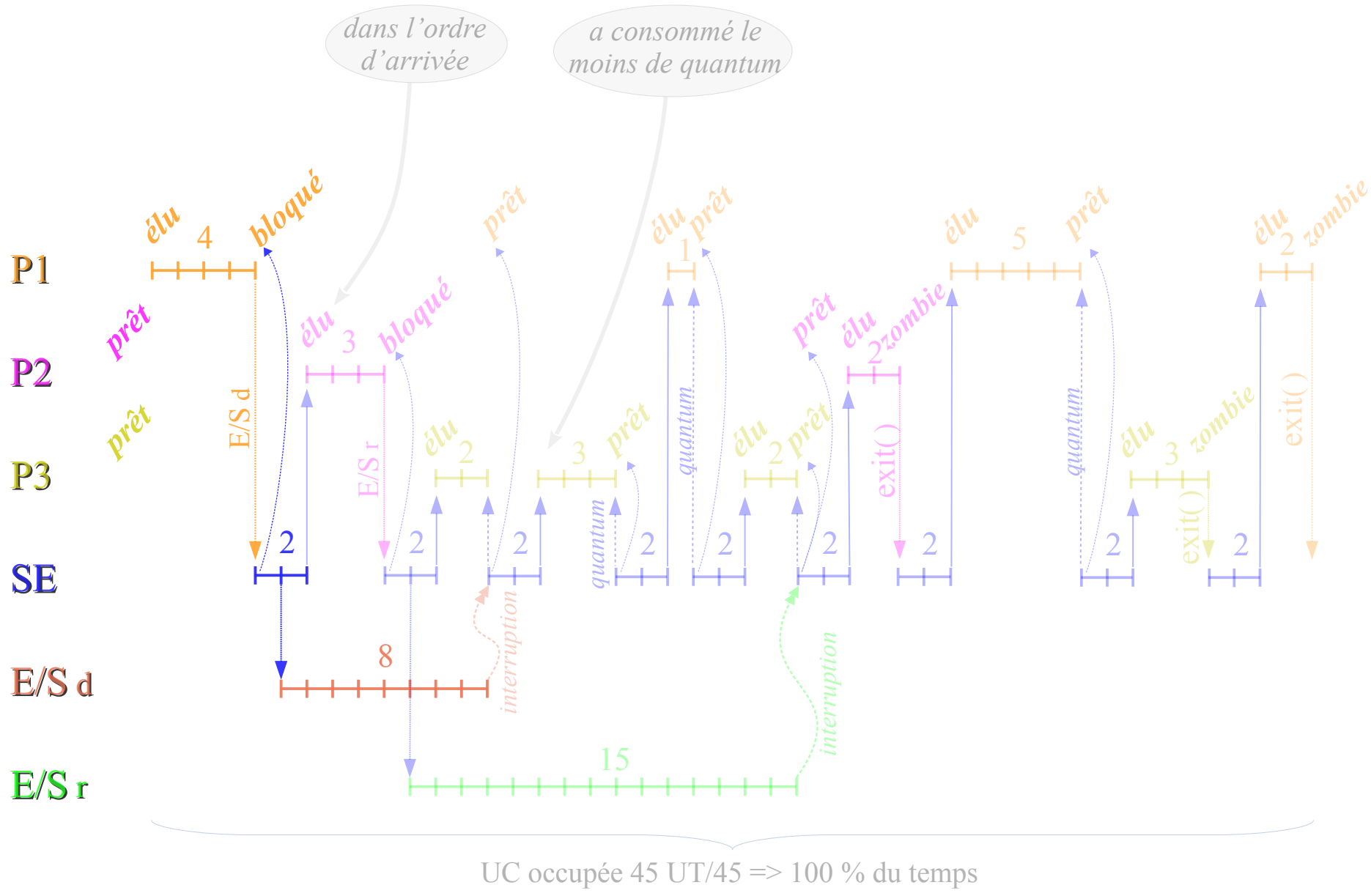
Chronogramme



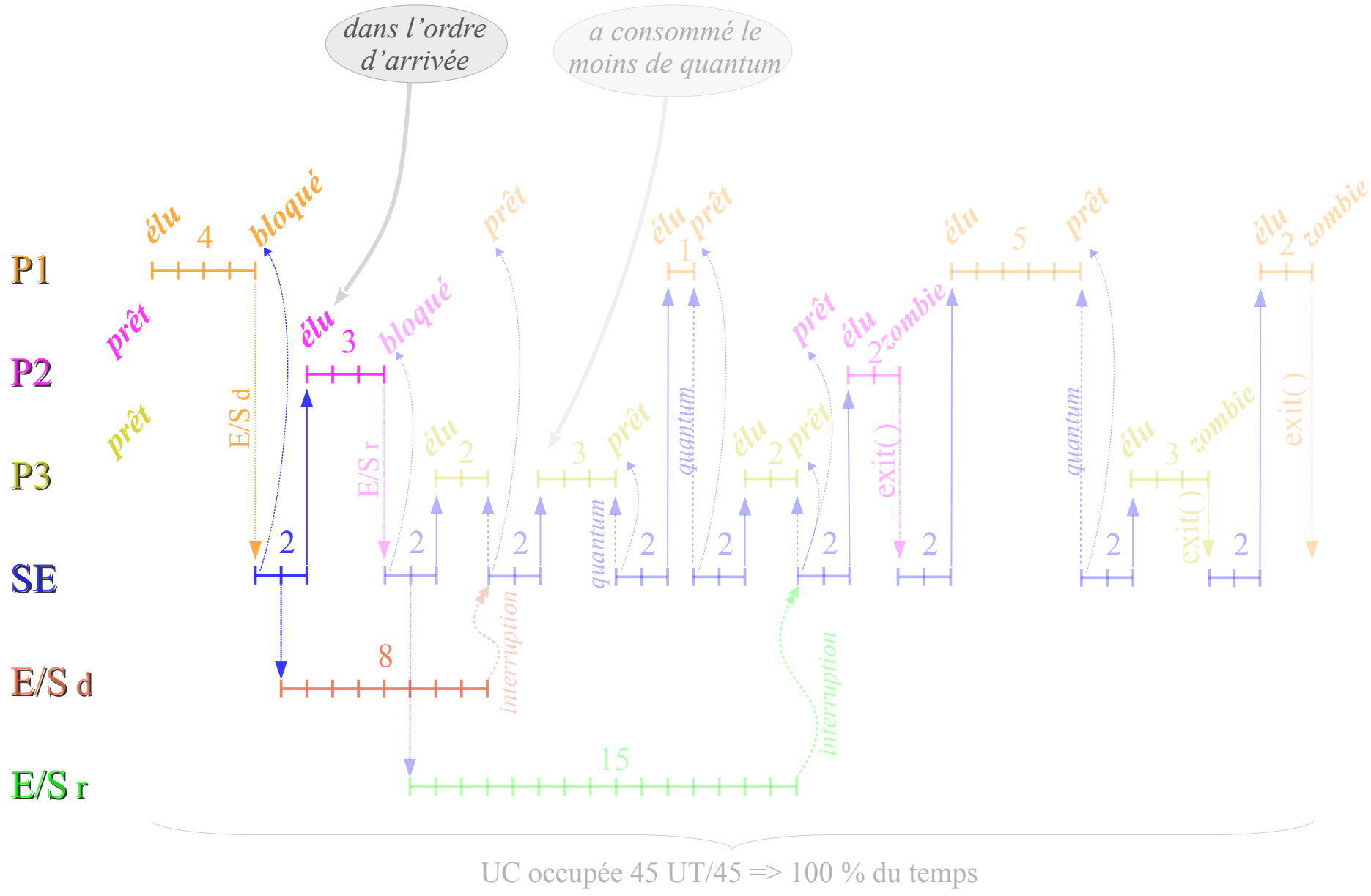
Chronogramme



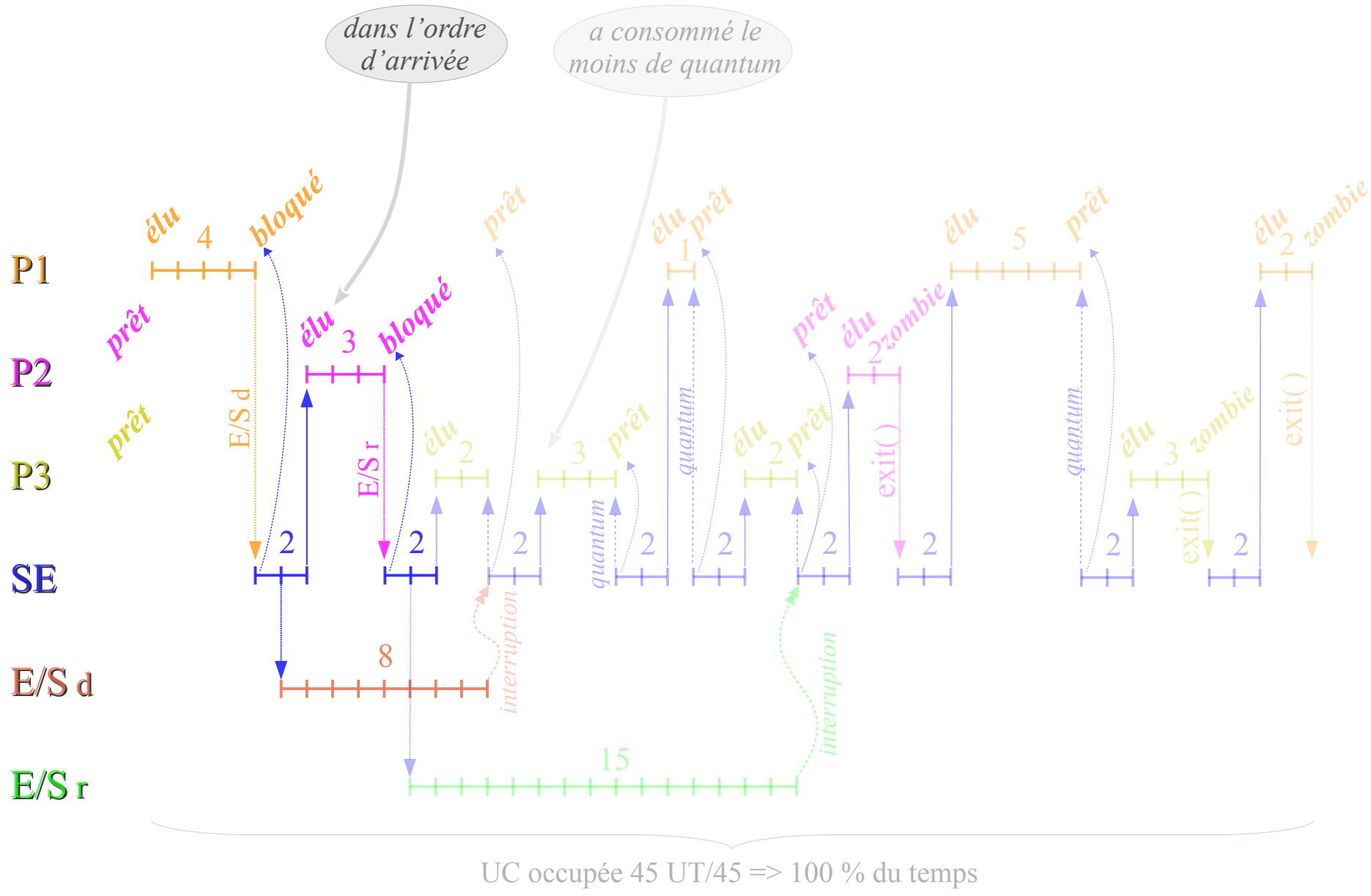
Chronogramme



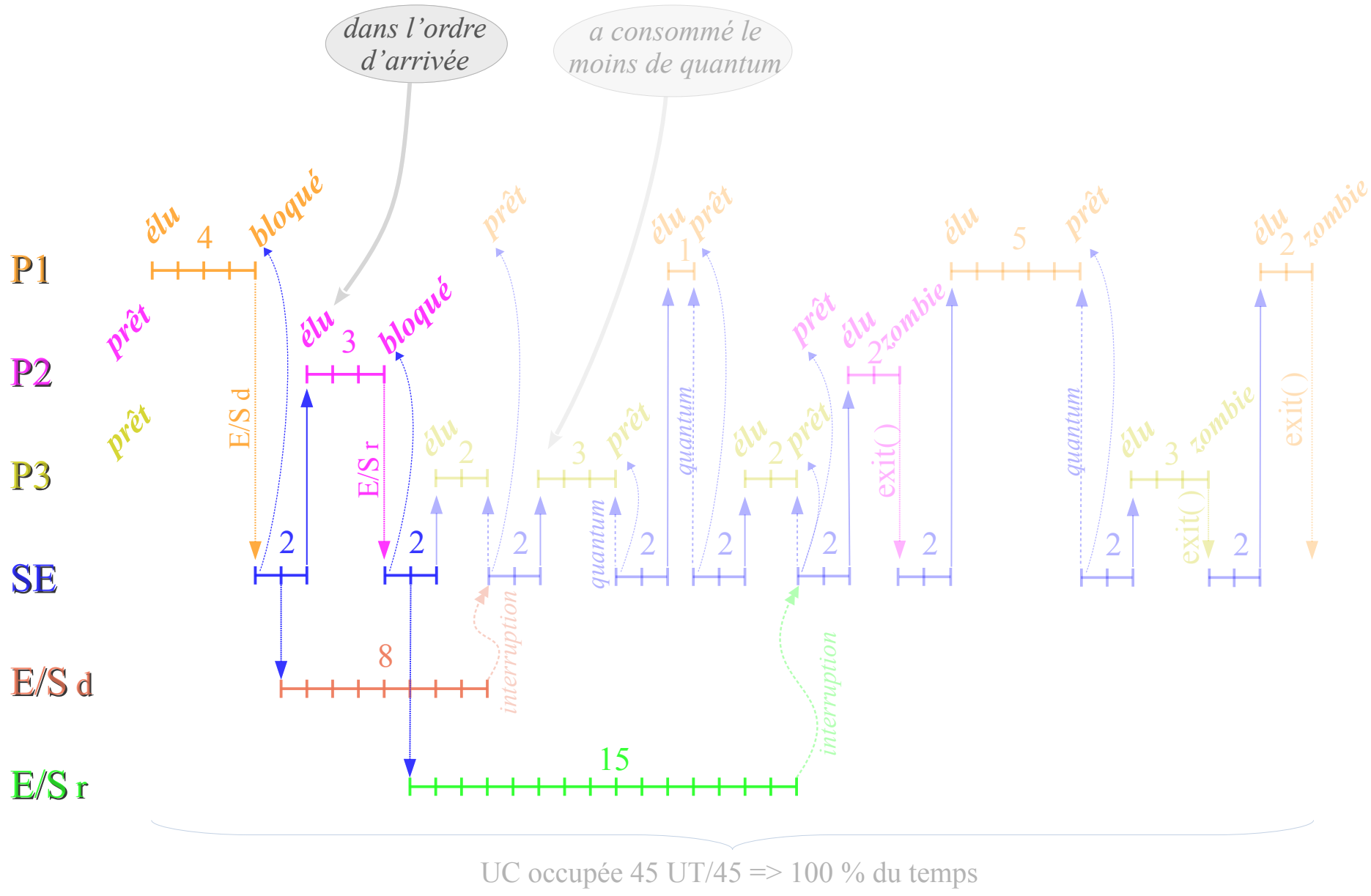
Chronogramme



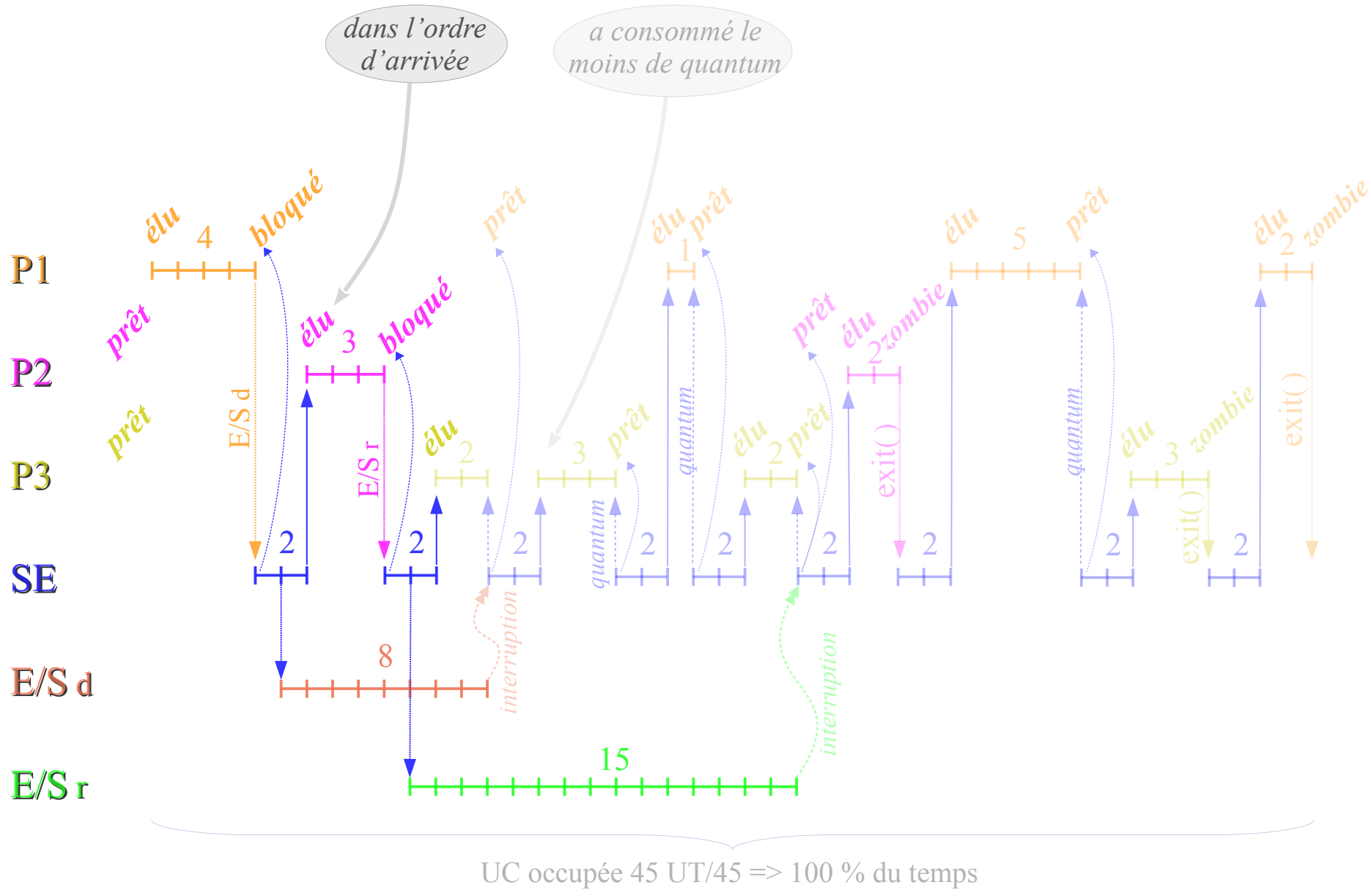
Chronogramme



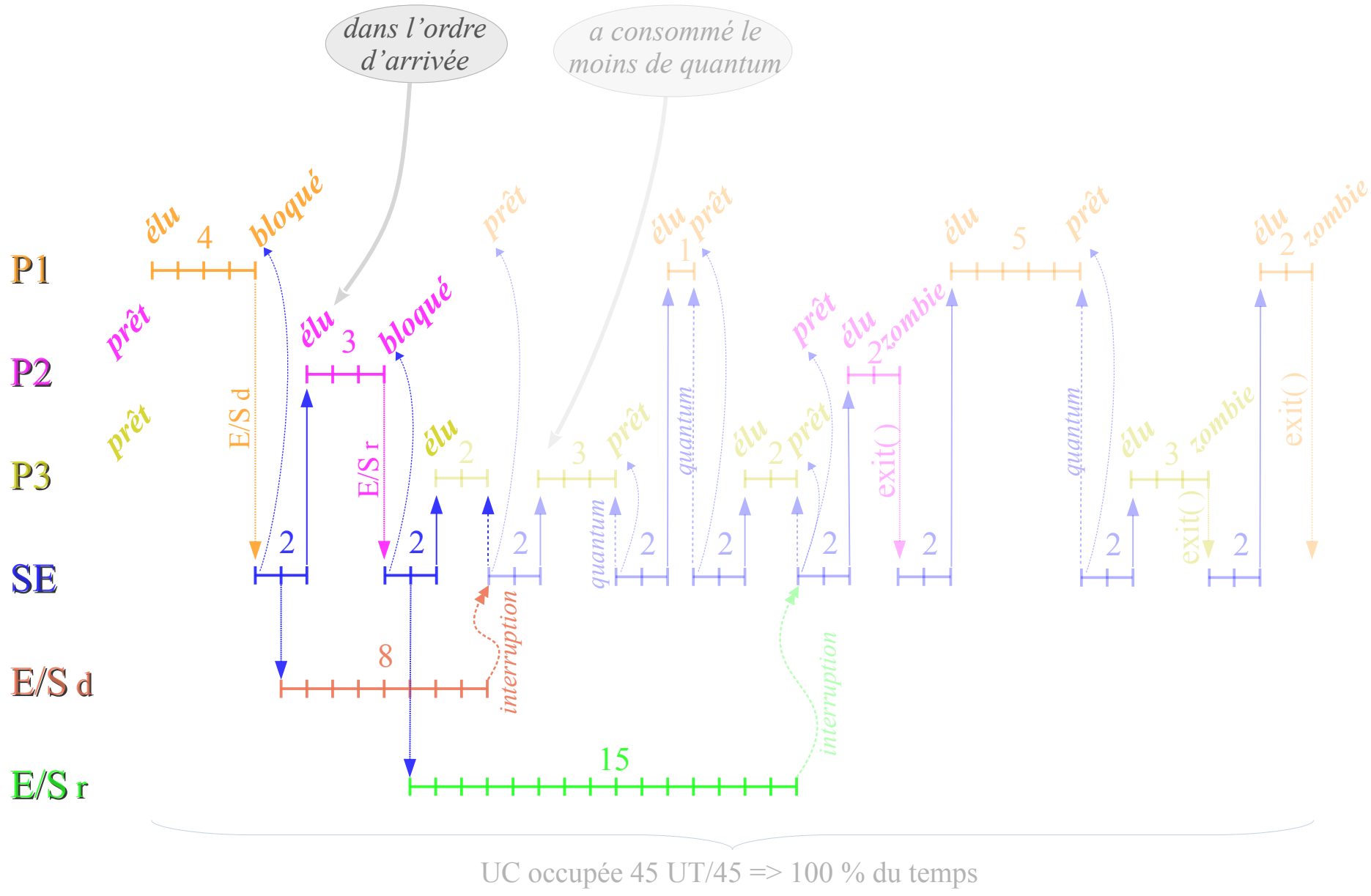
Chronogramme



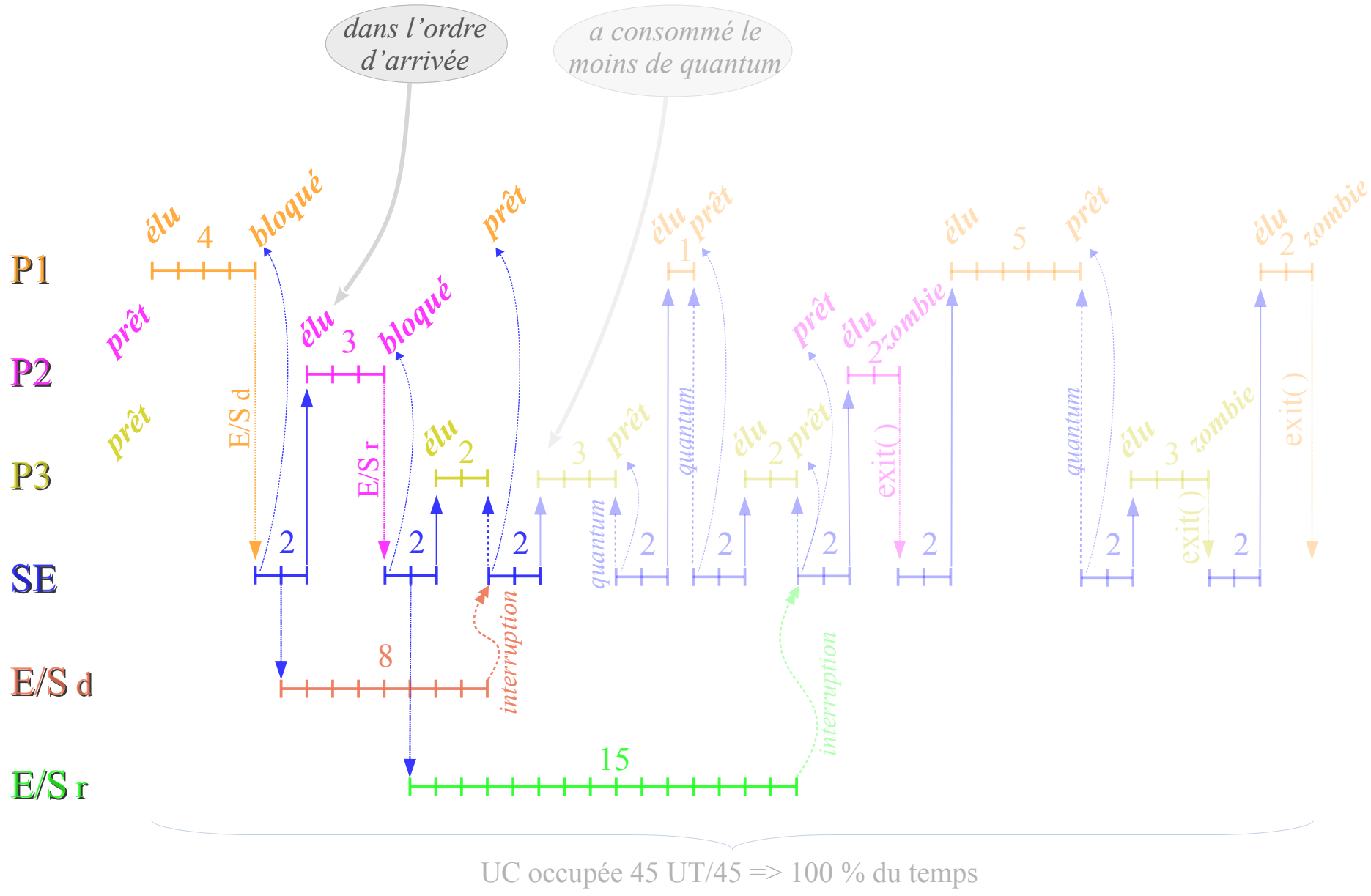
Chronogramme



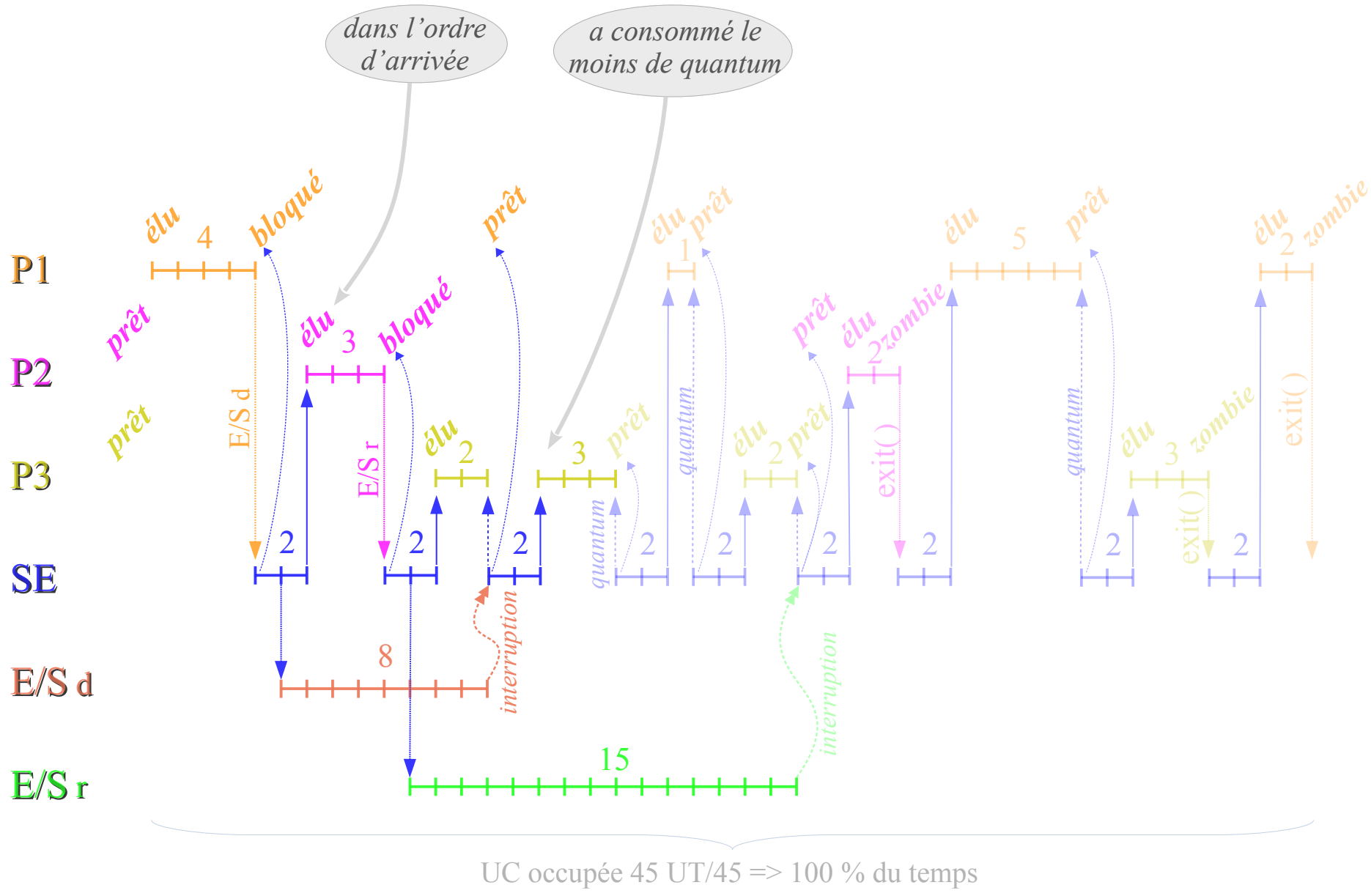
Chronogramme



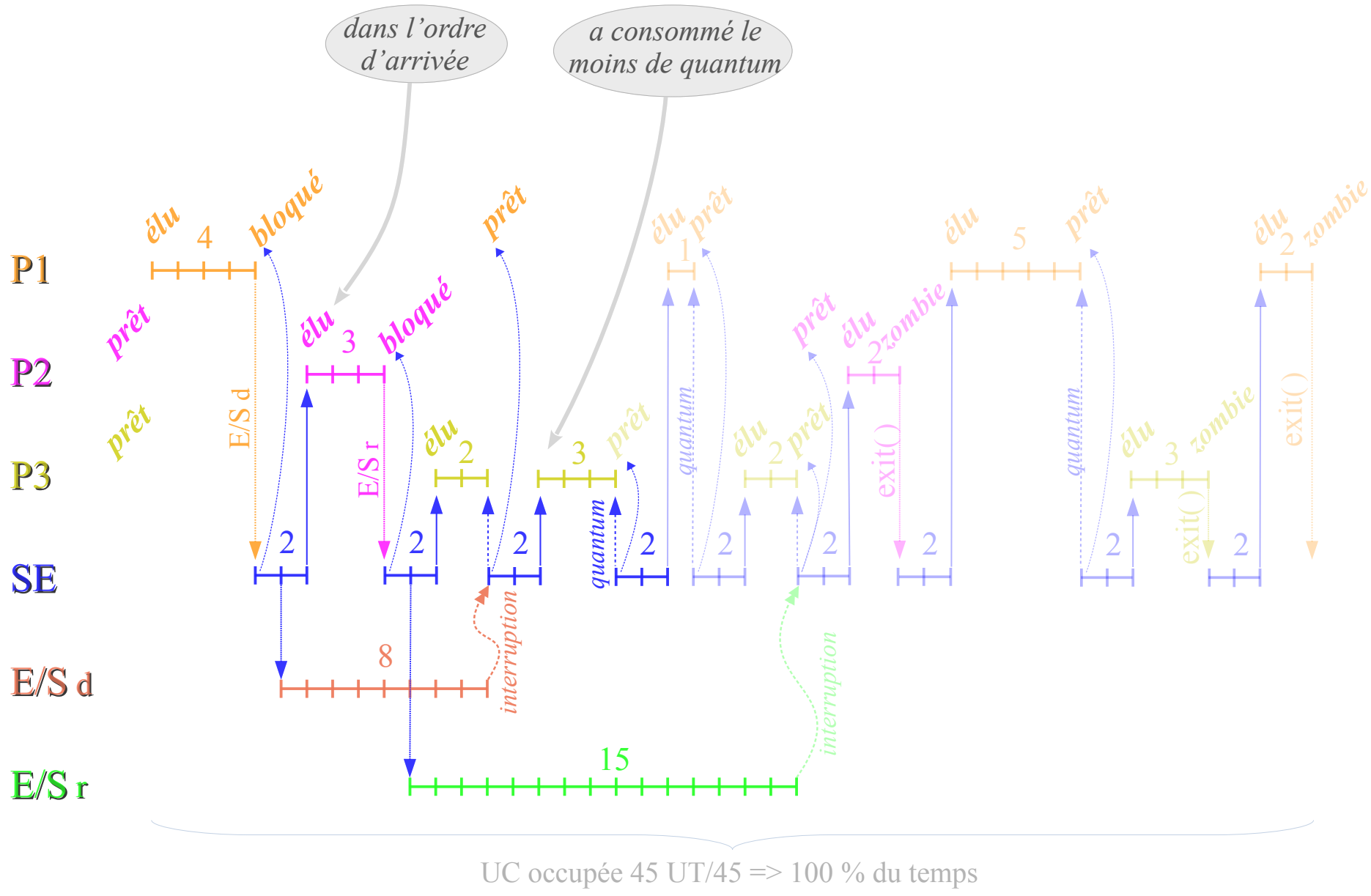
Chronogramme



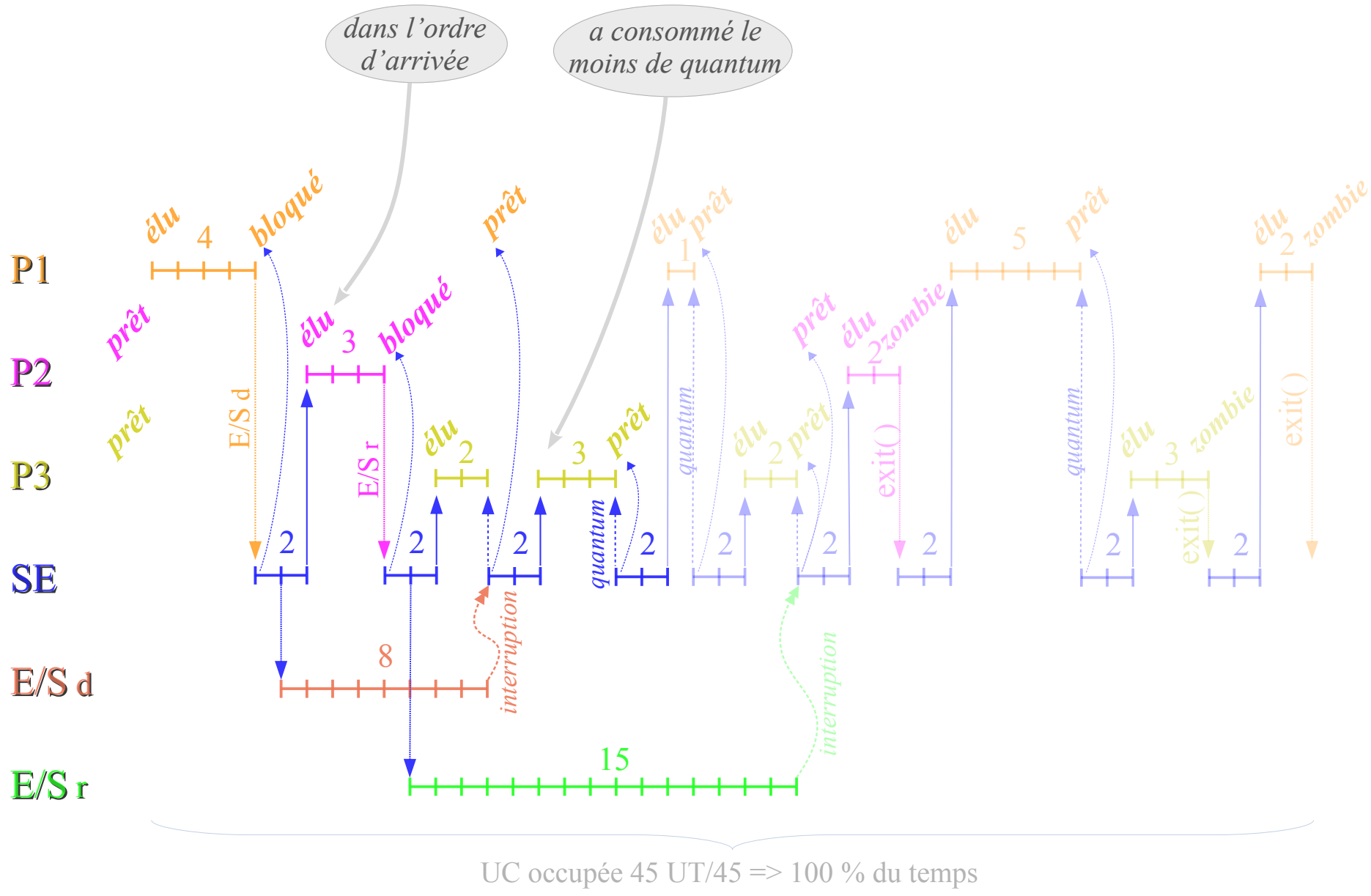
Chronogramme



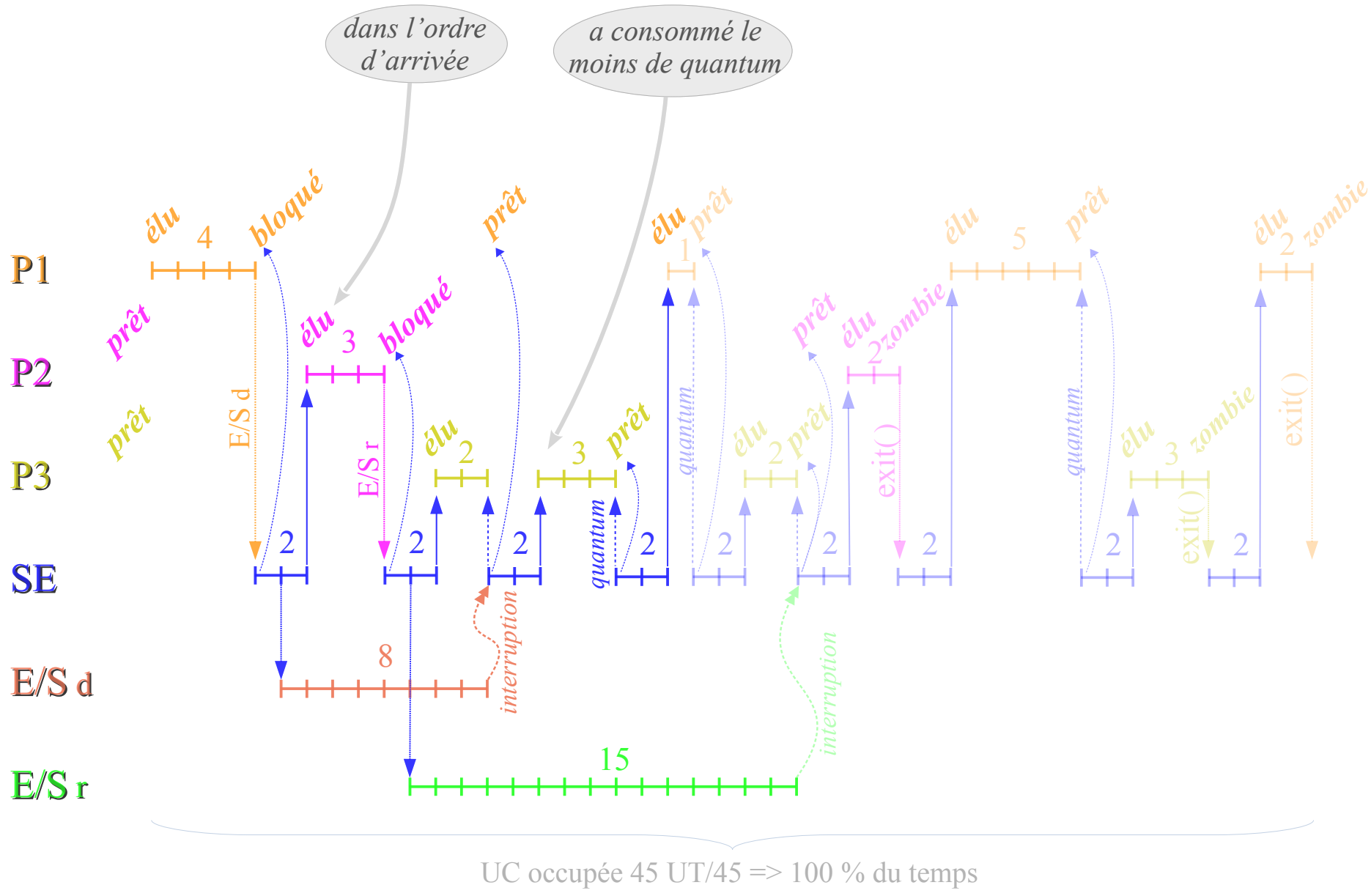
Chronogramme



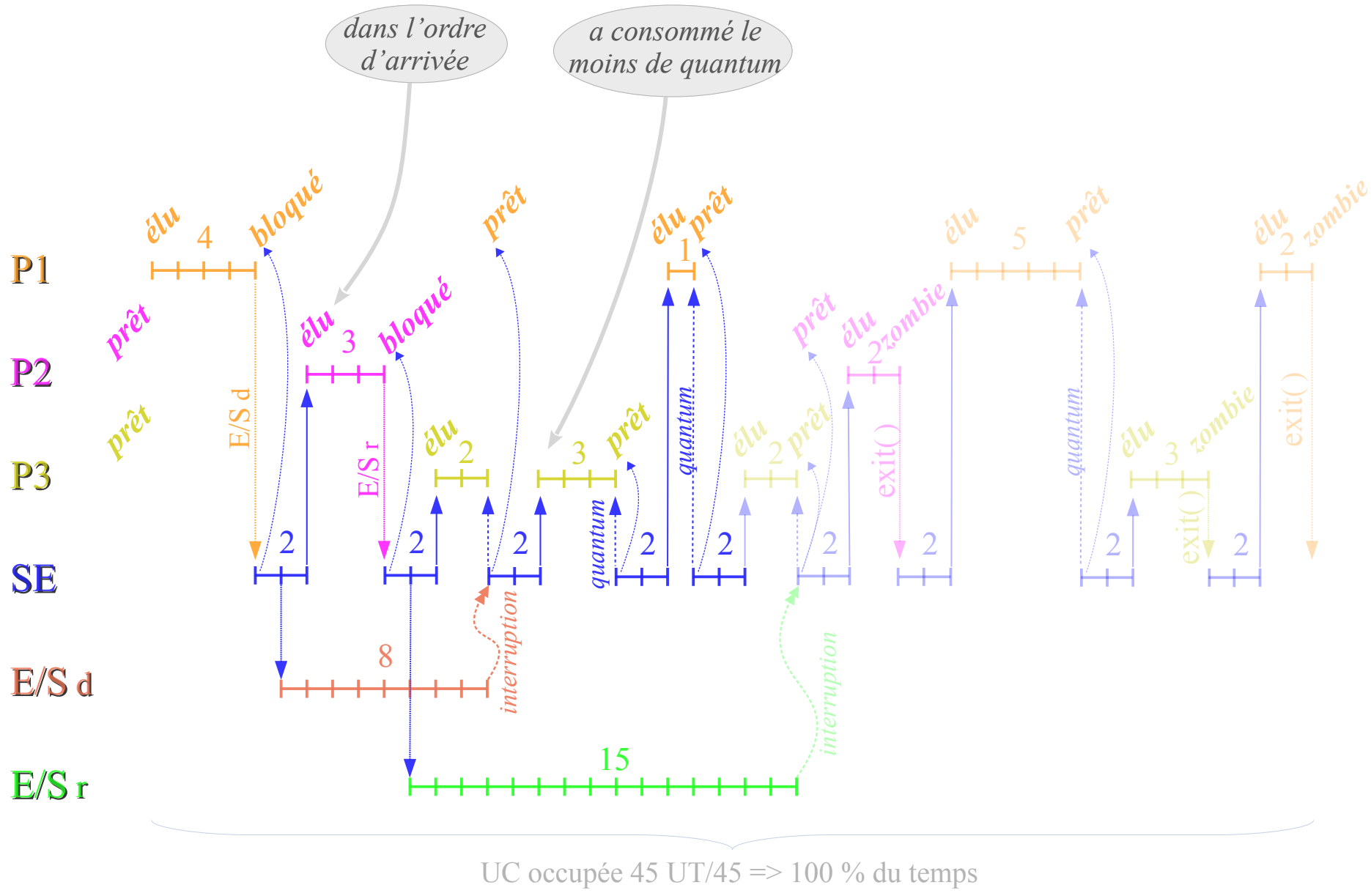
Chronogramme



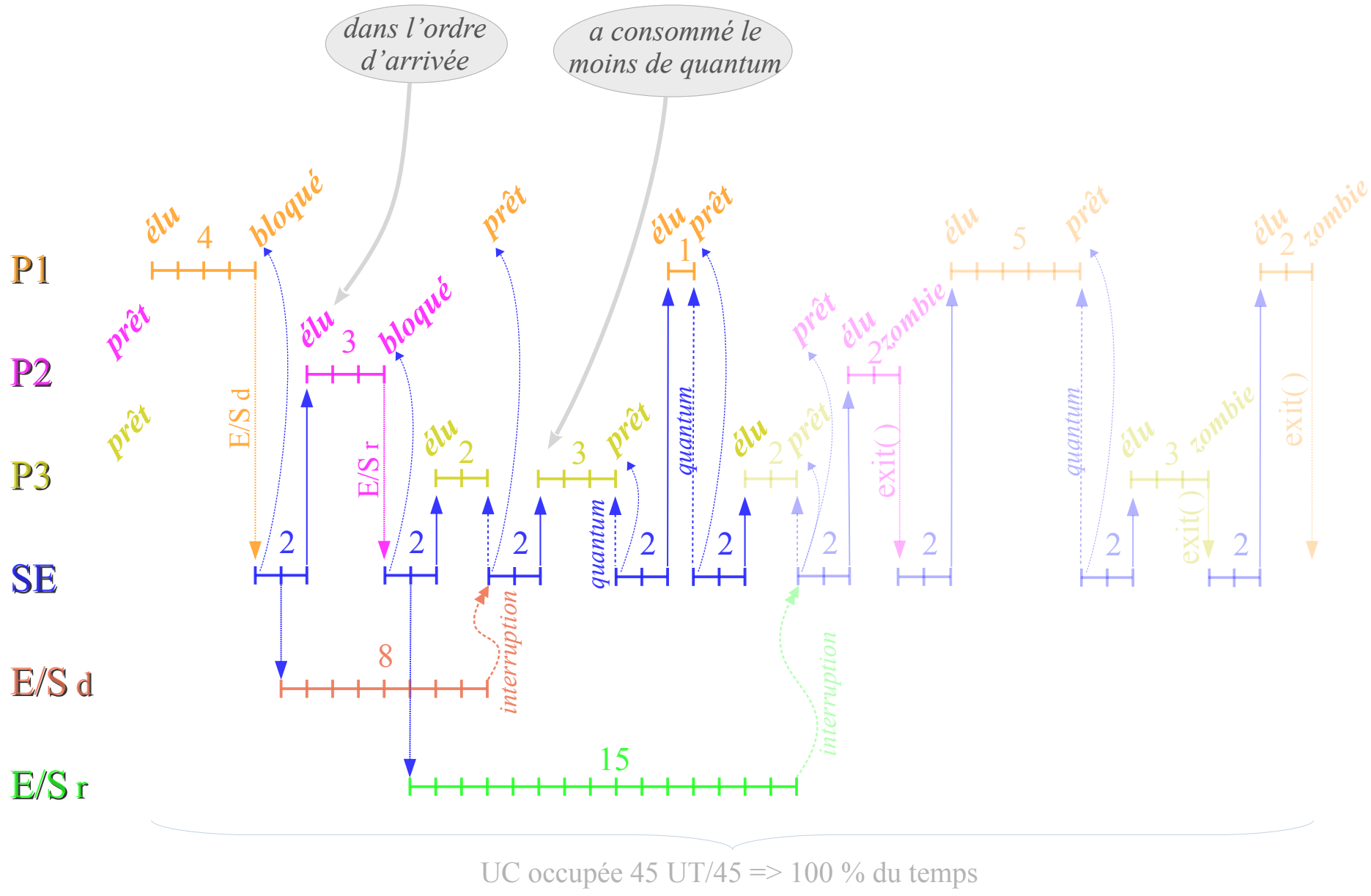
Chronogramme



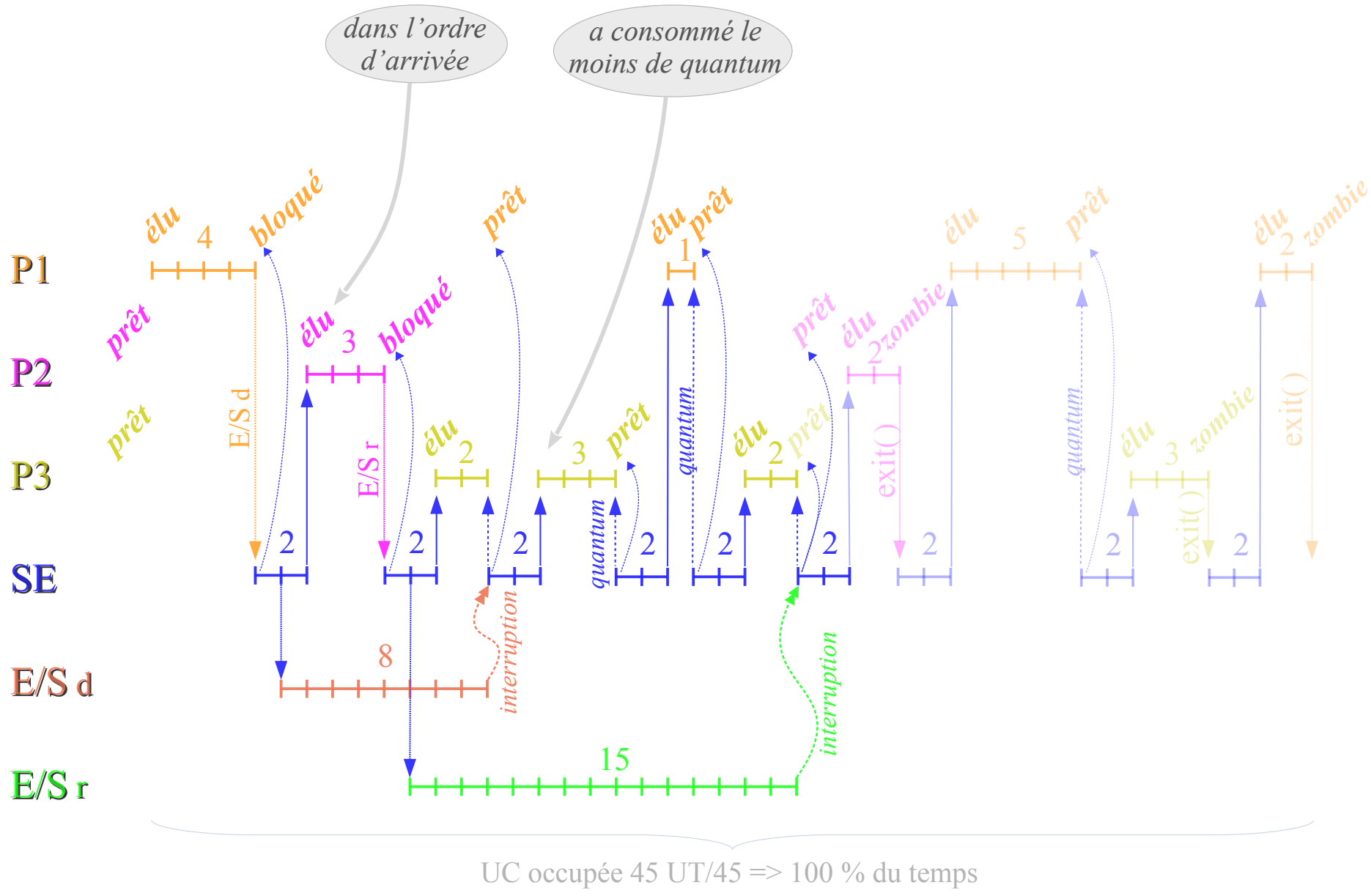
Chronogramme



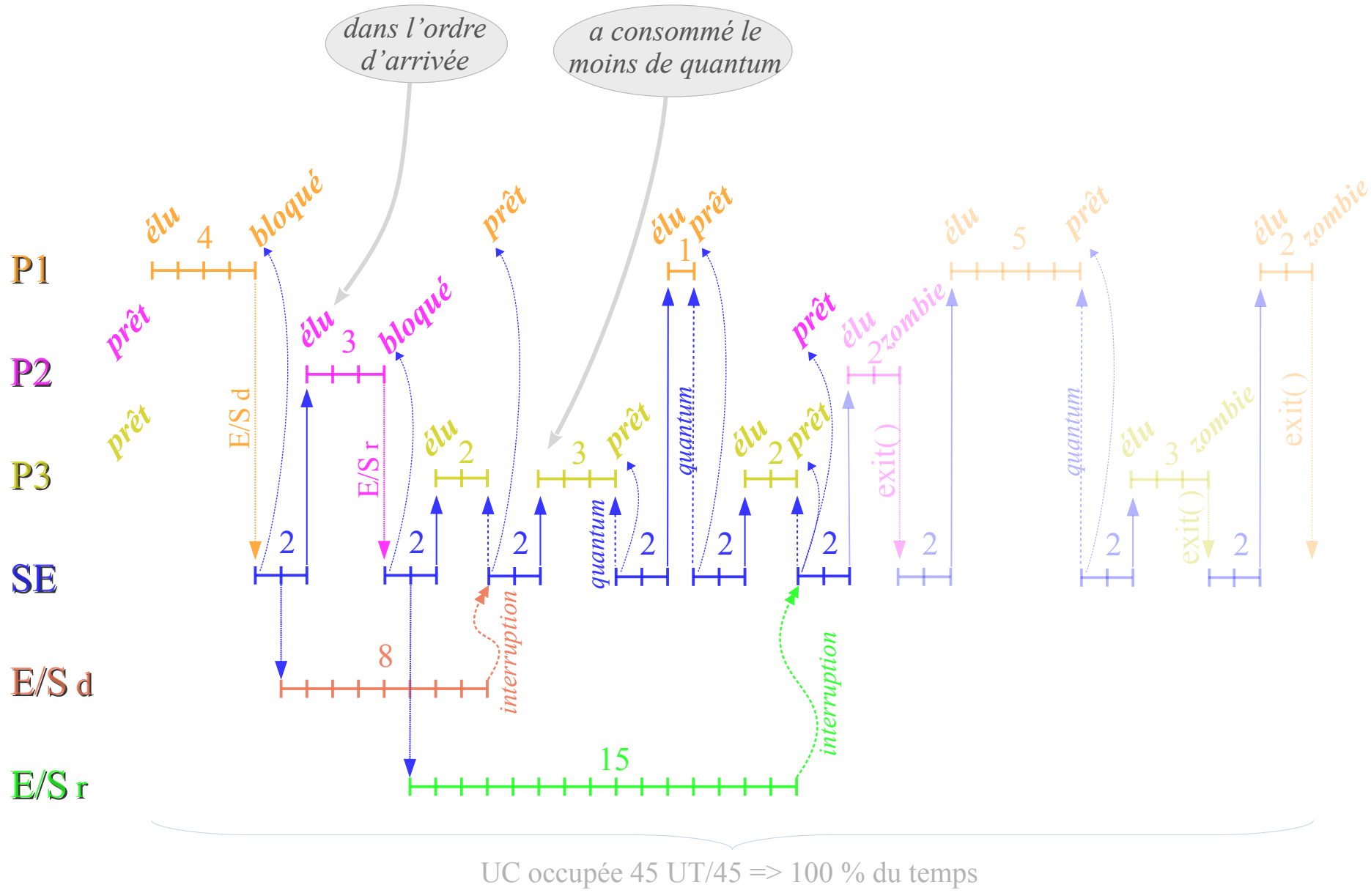
Chronogramme



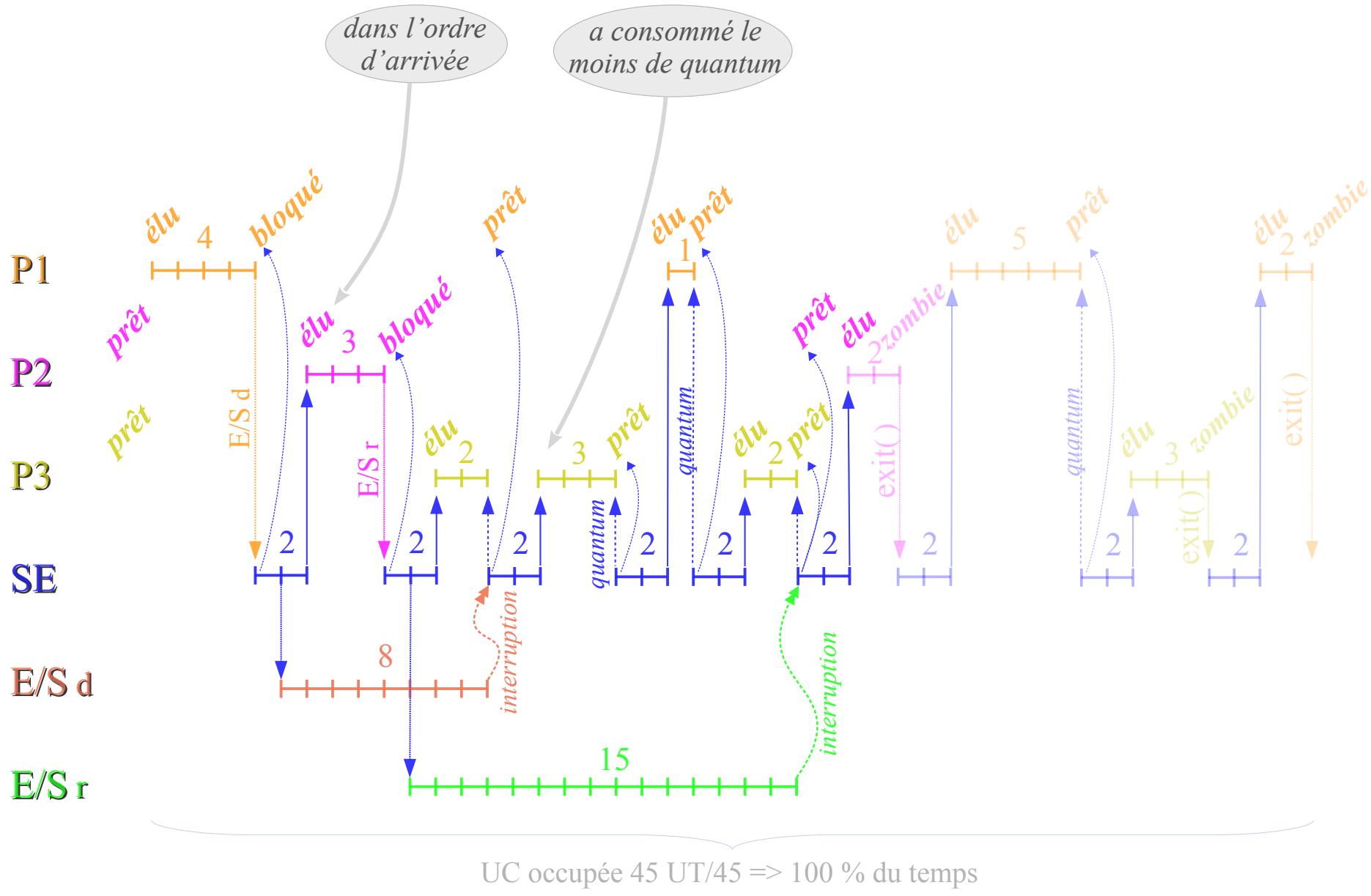
Chronogramme



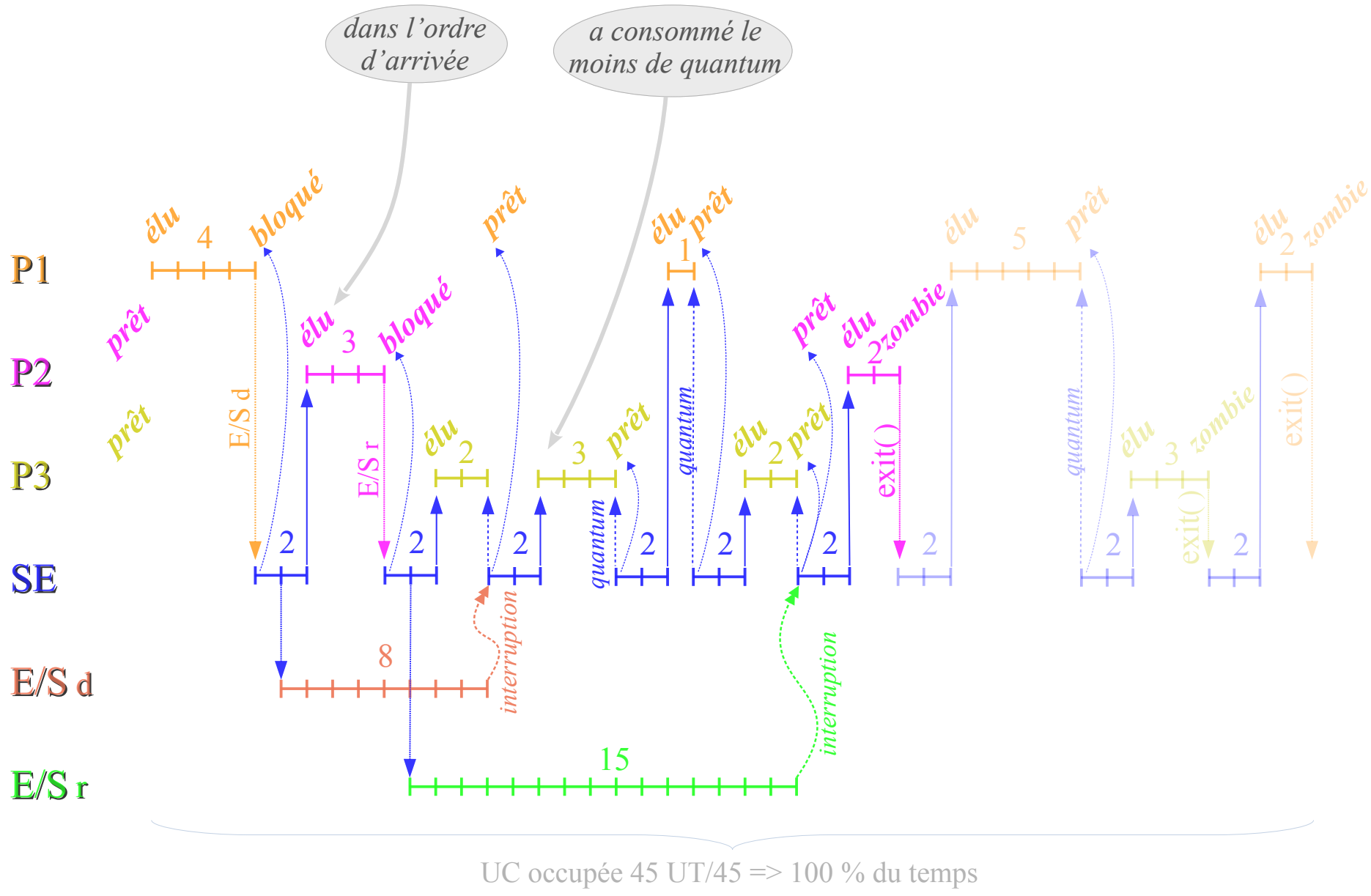
Chronogramme



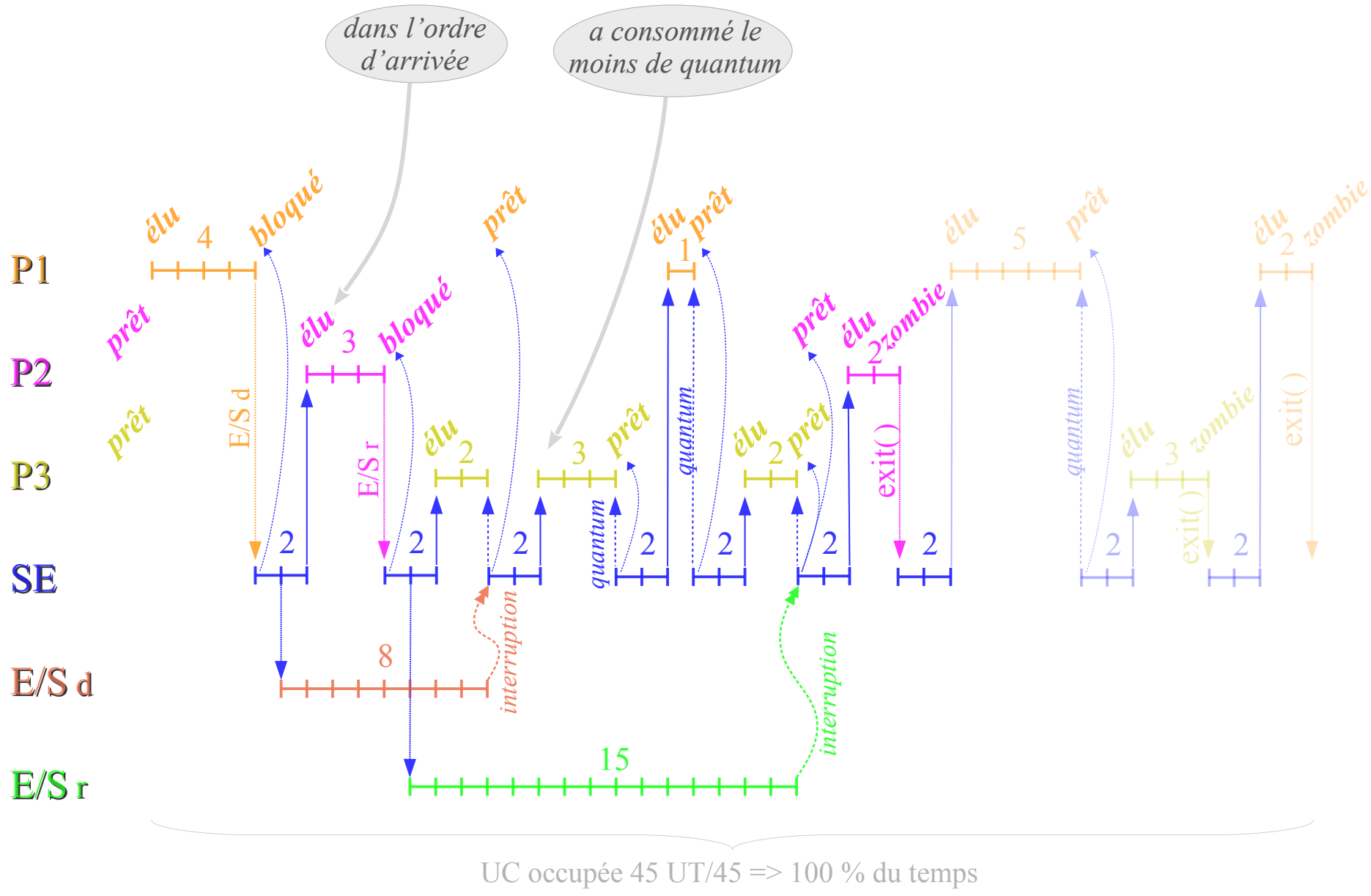
Chronogramme



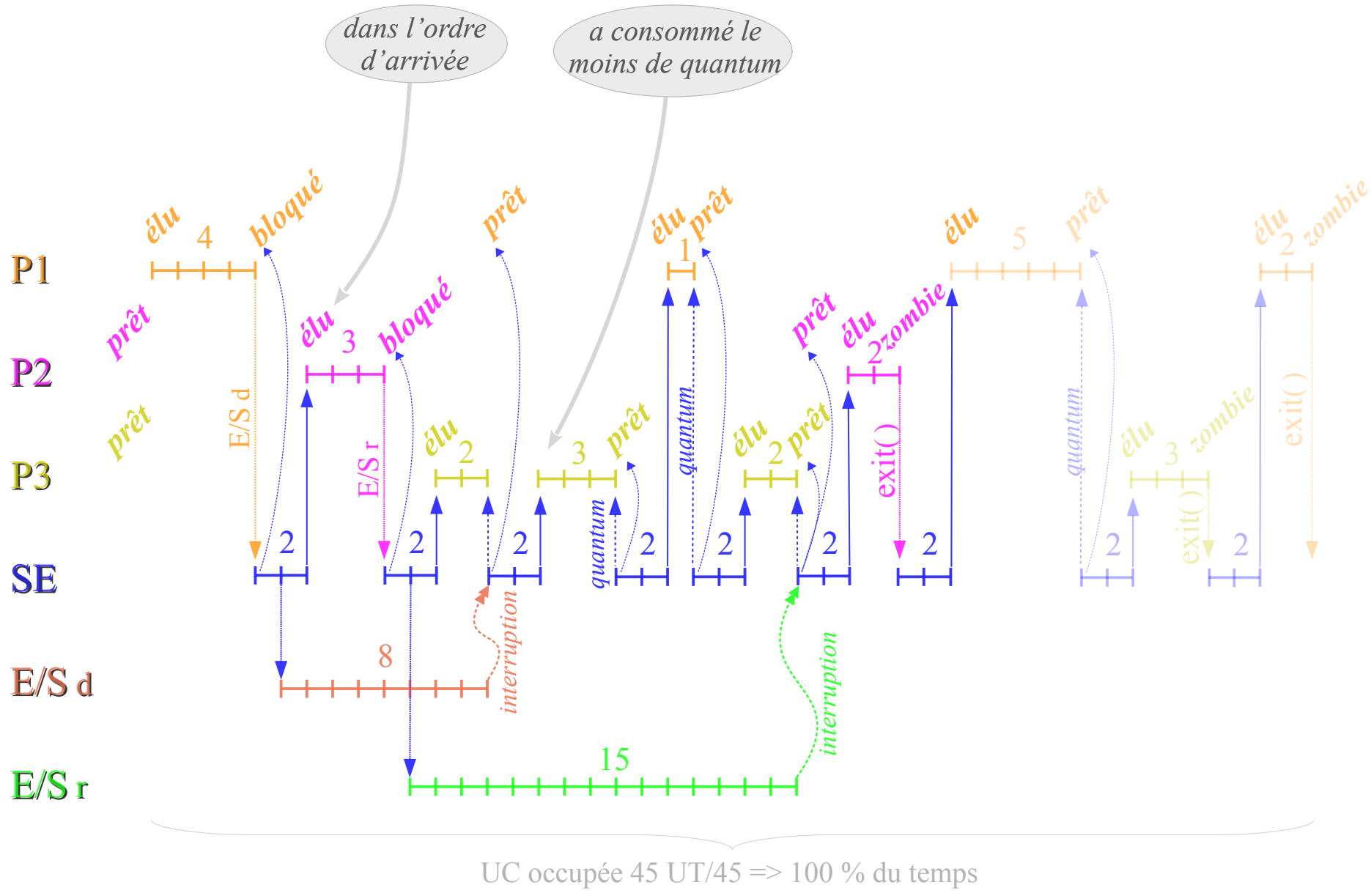
Chronogramme



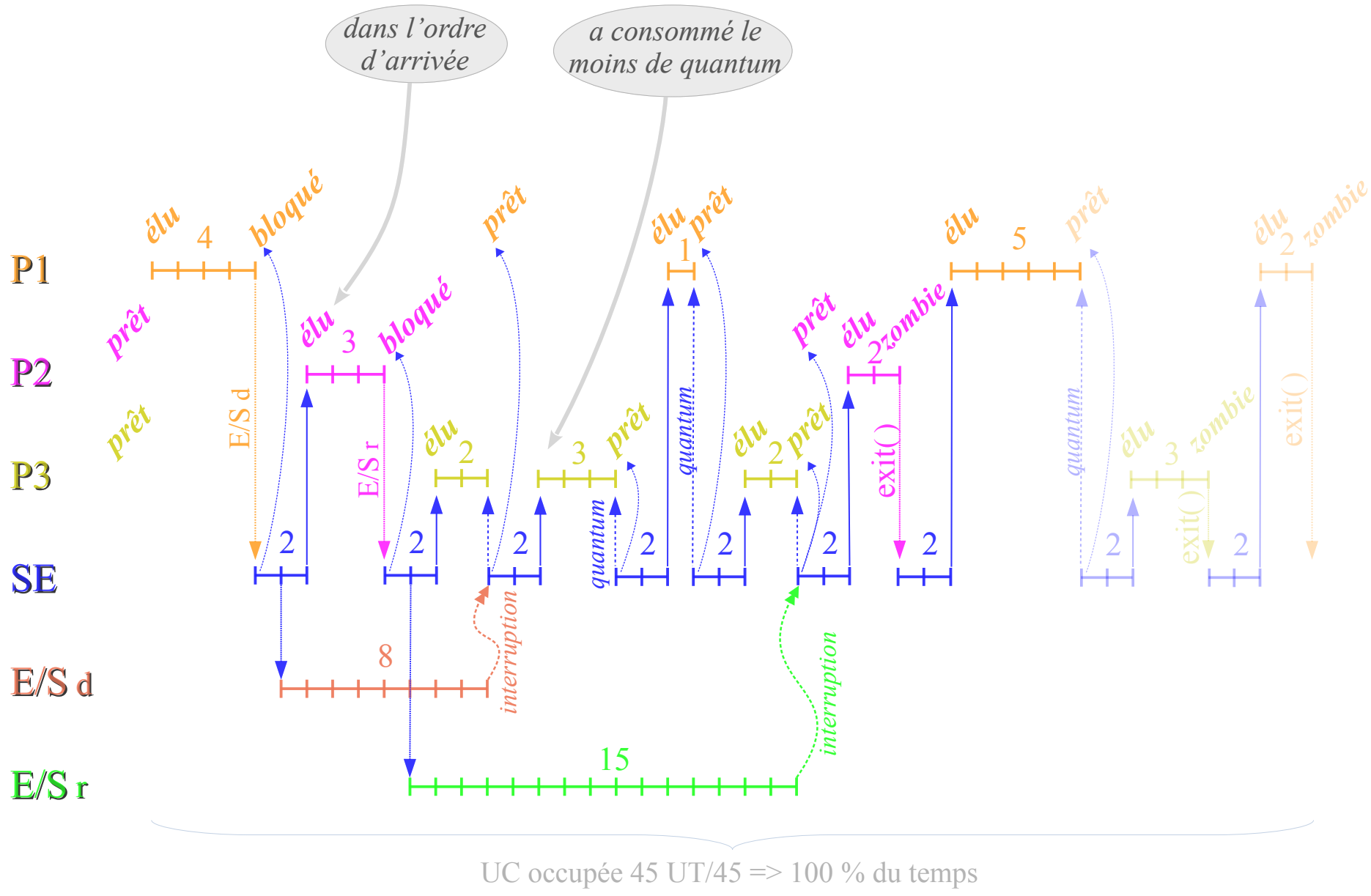
Chronogramme



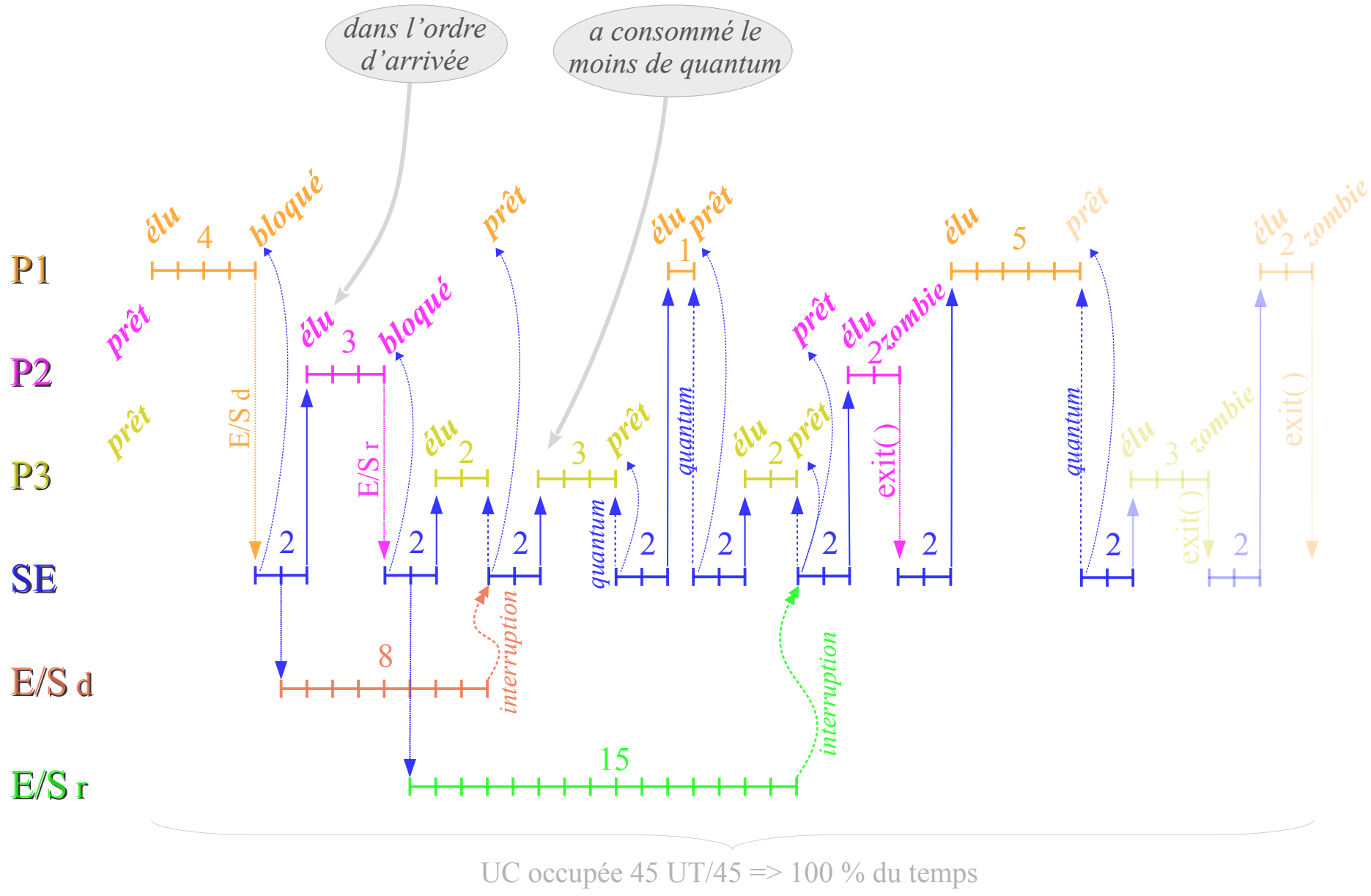
Chronogramme



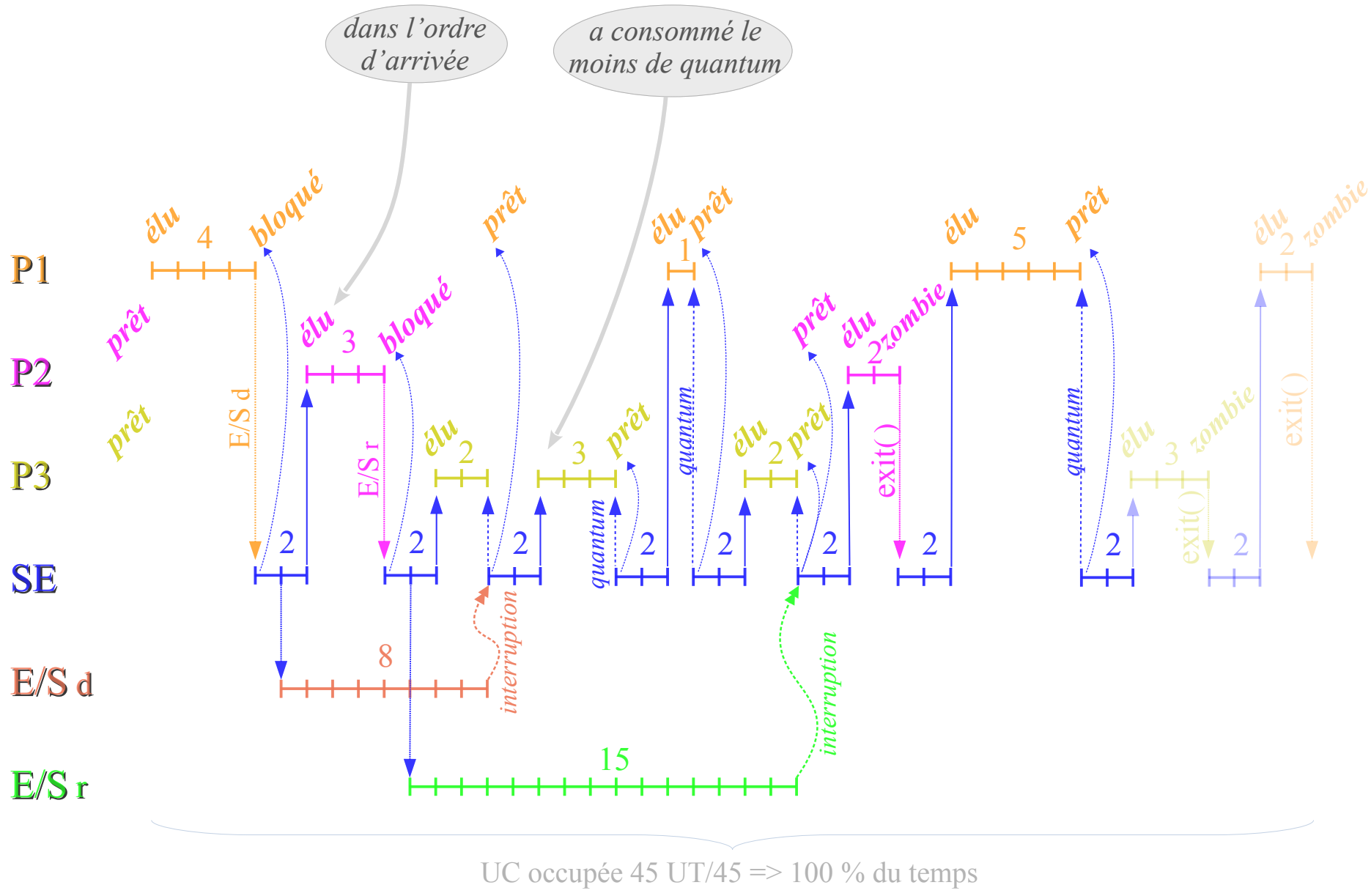
Chronogramme



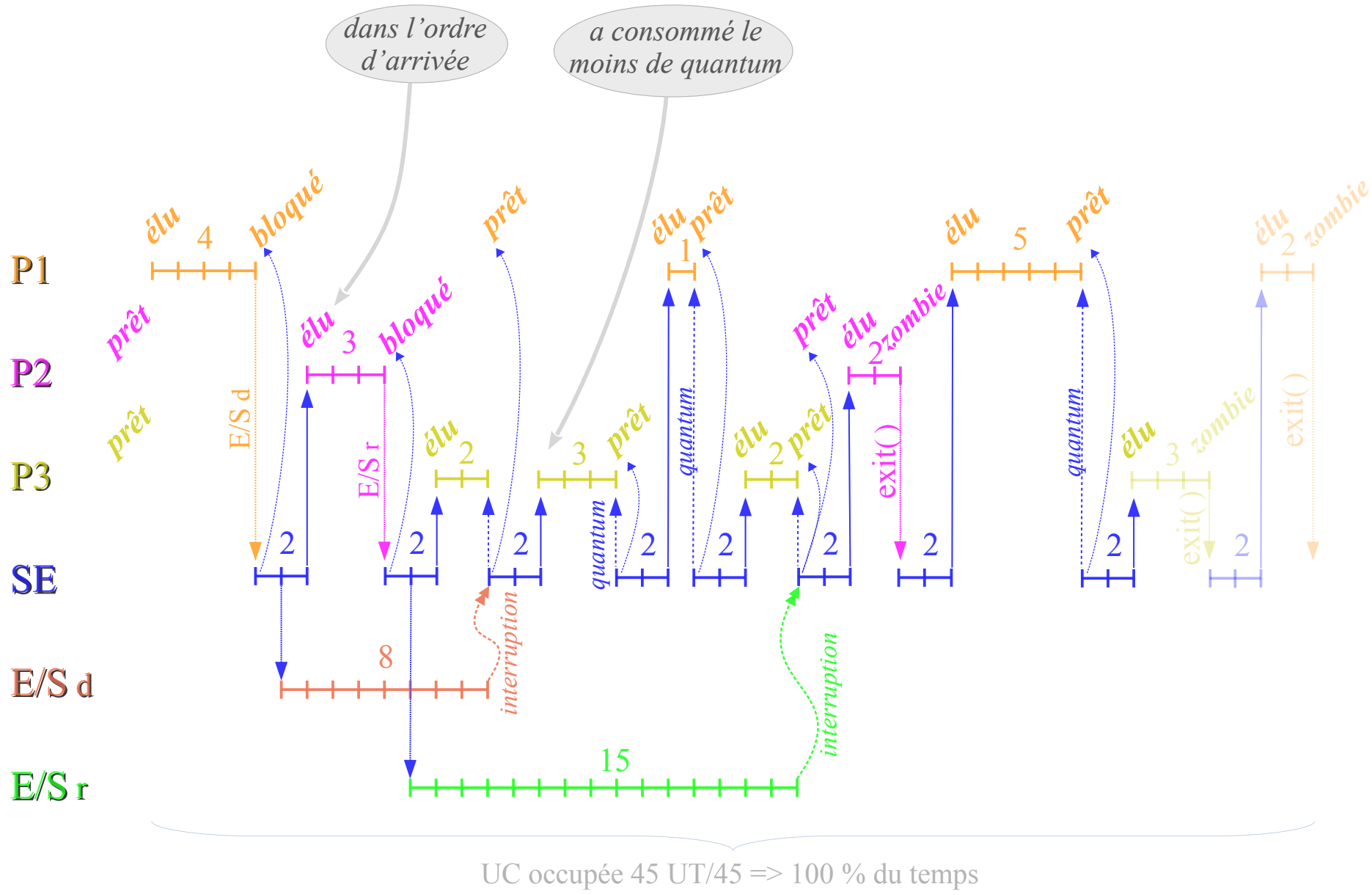
Chronogramme



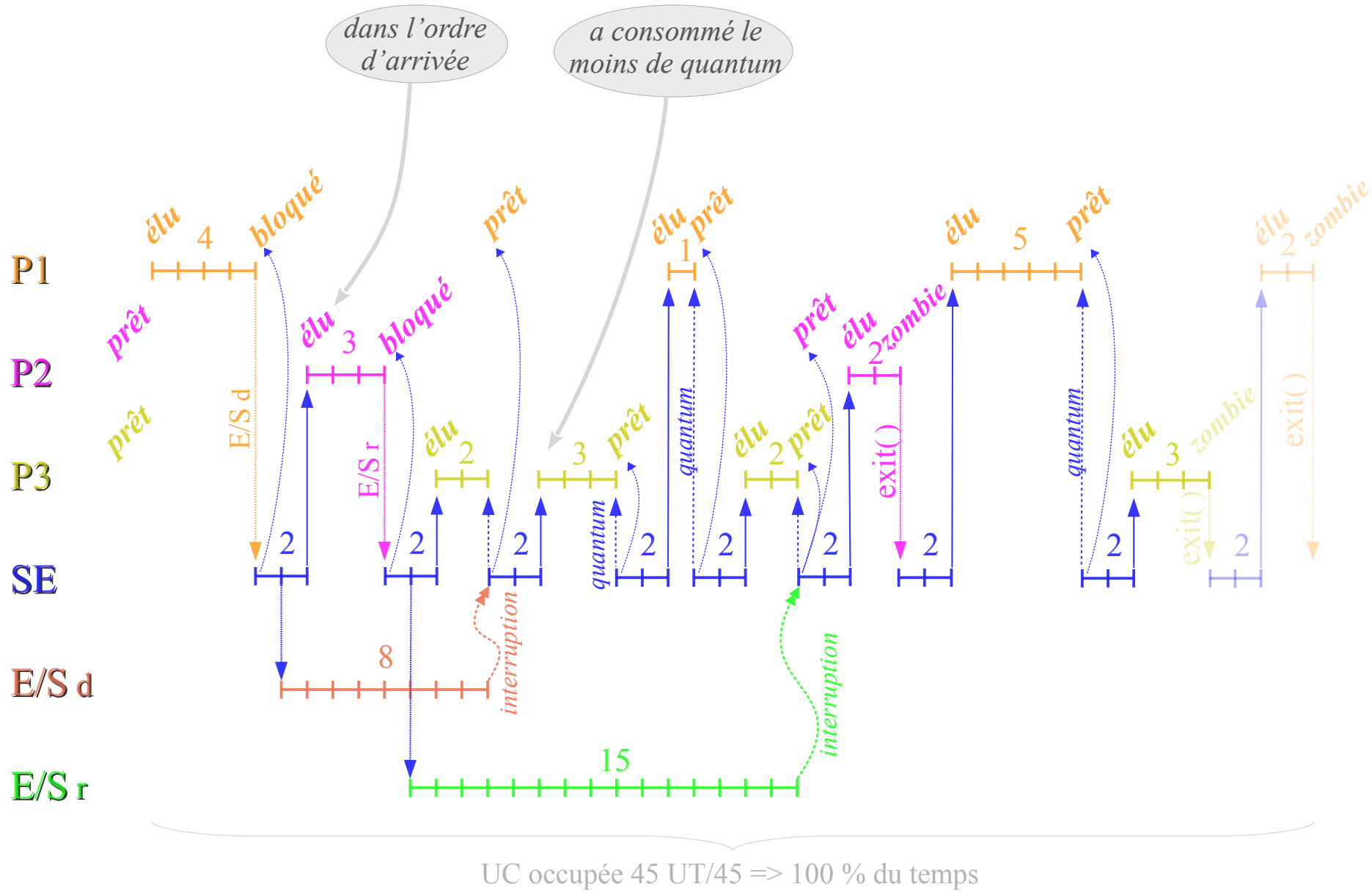
Chronogramme



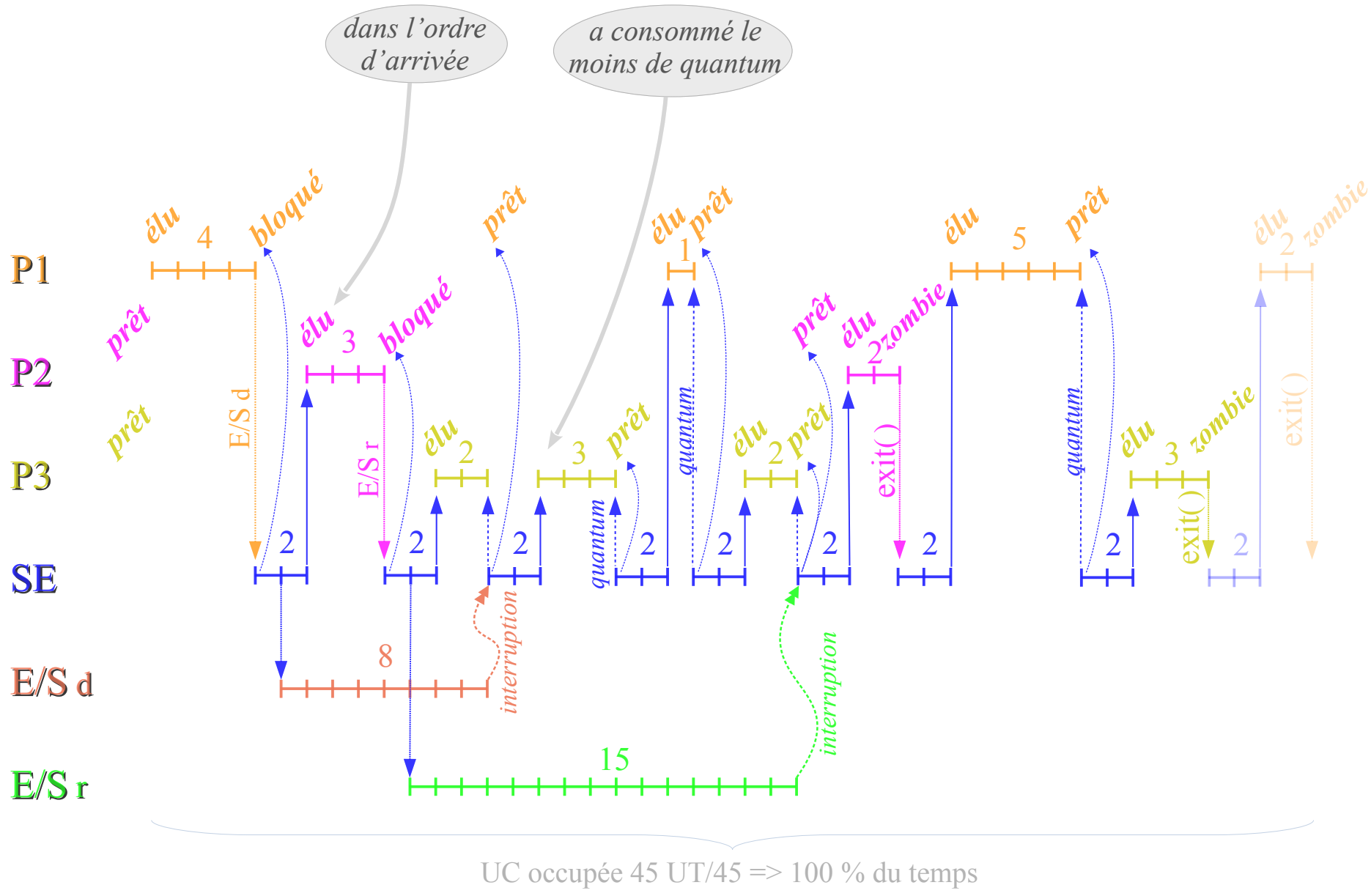
Chronogramme



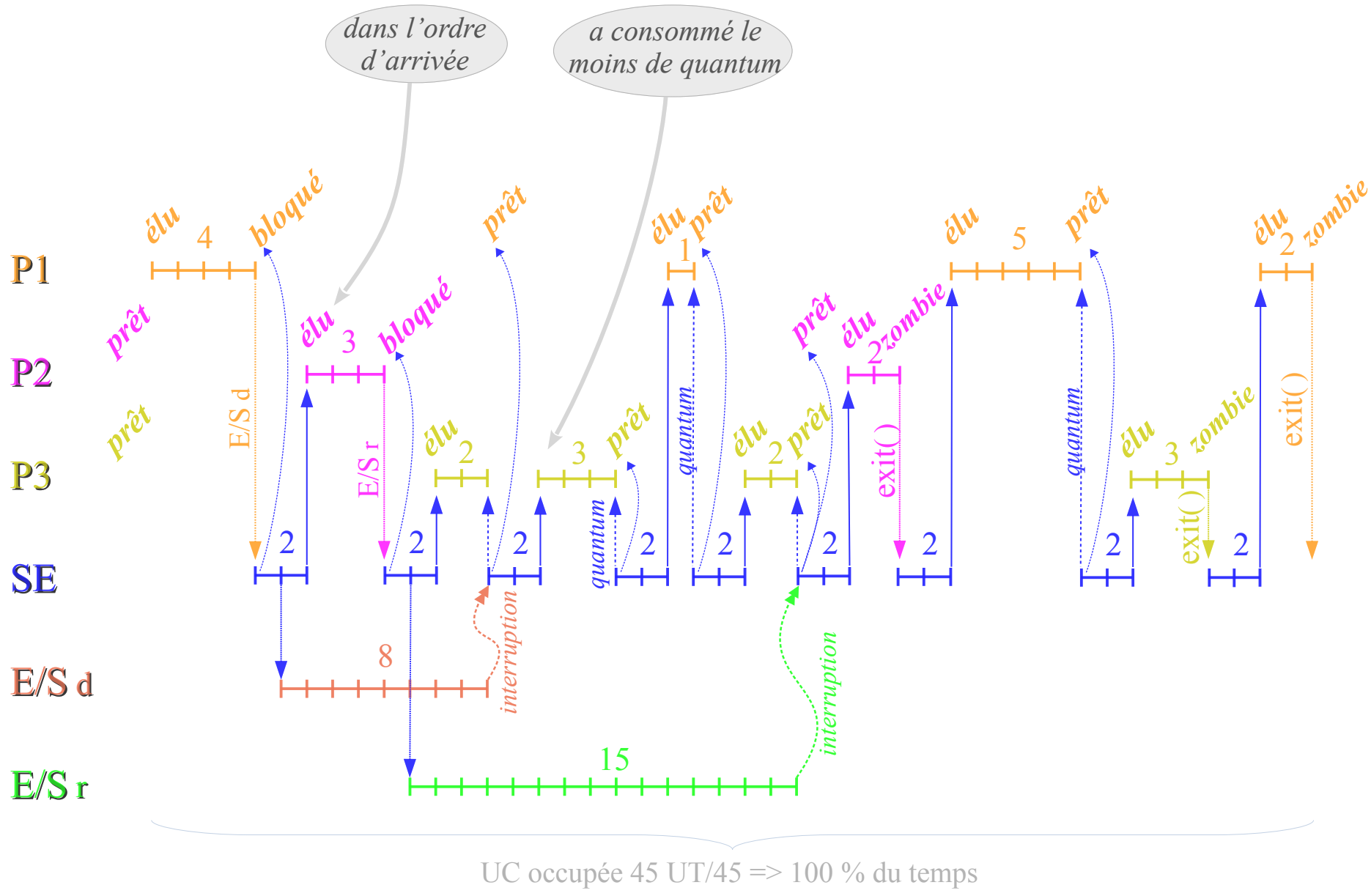
Chronogramme



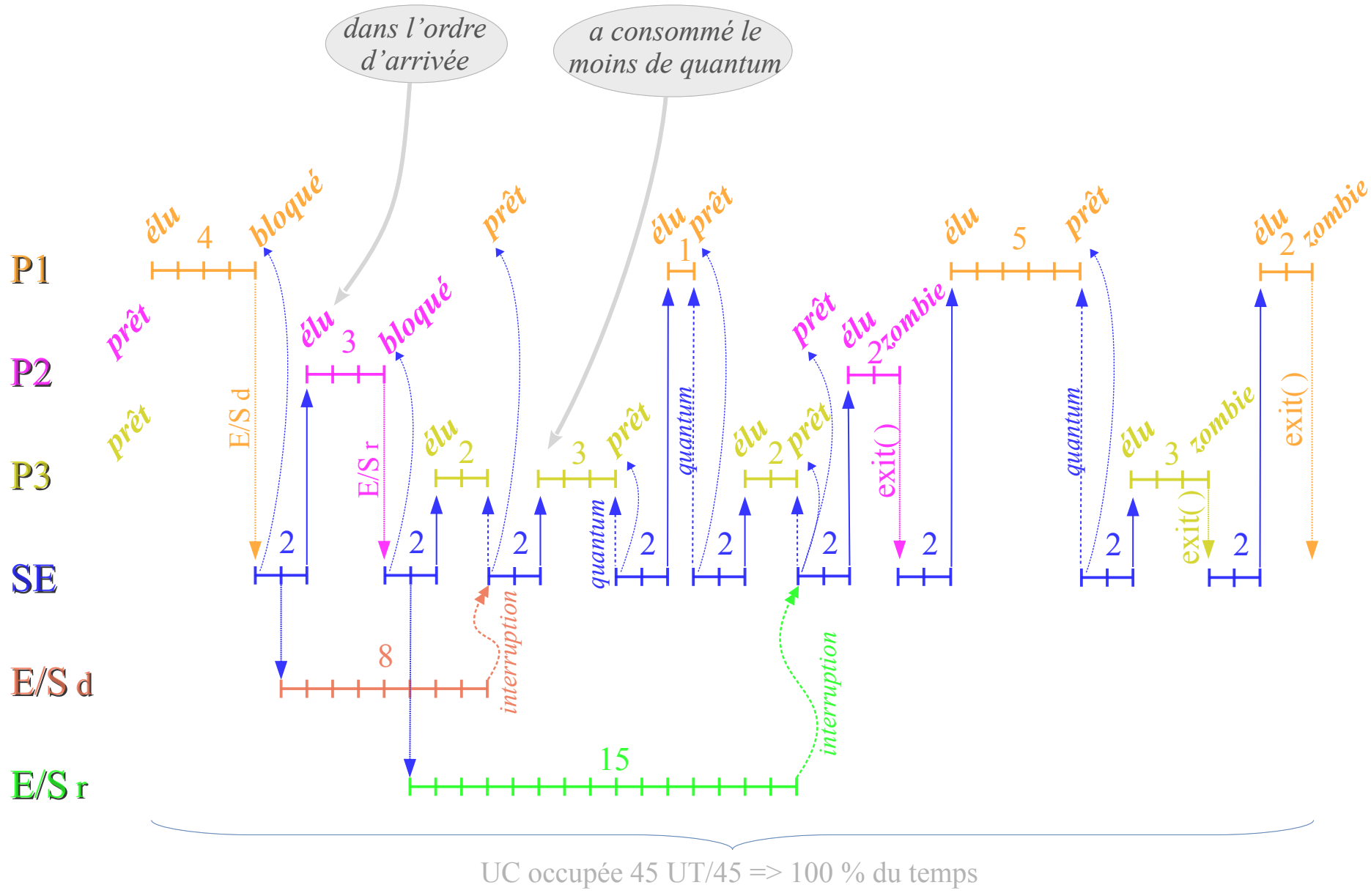
Chronogramme



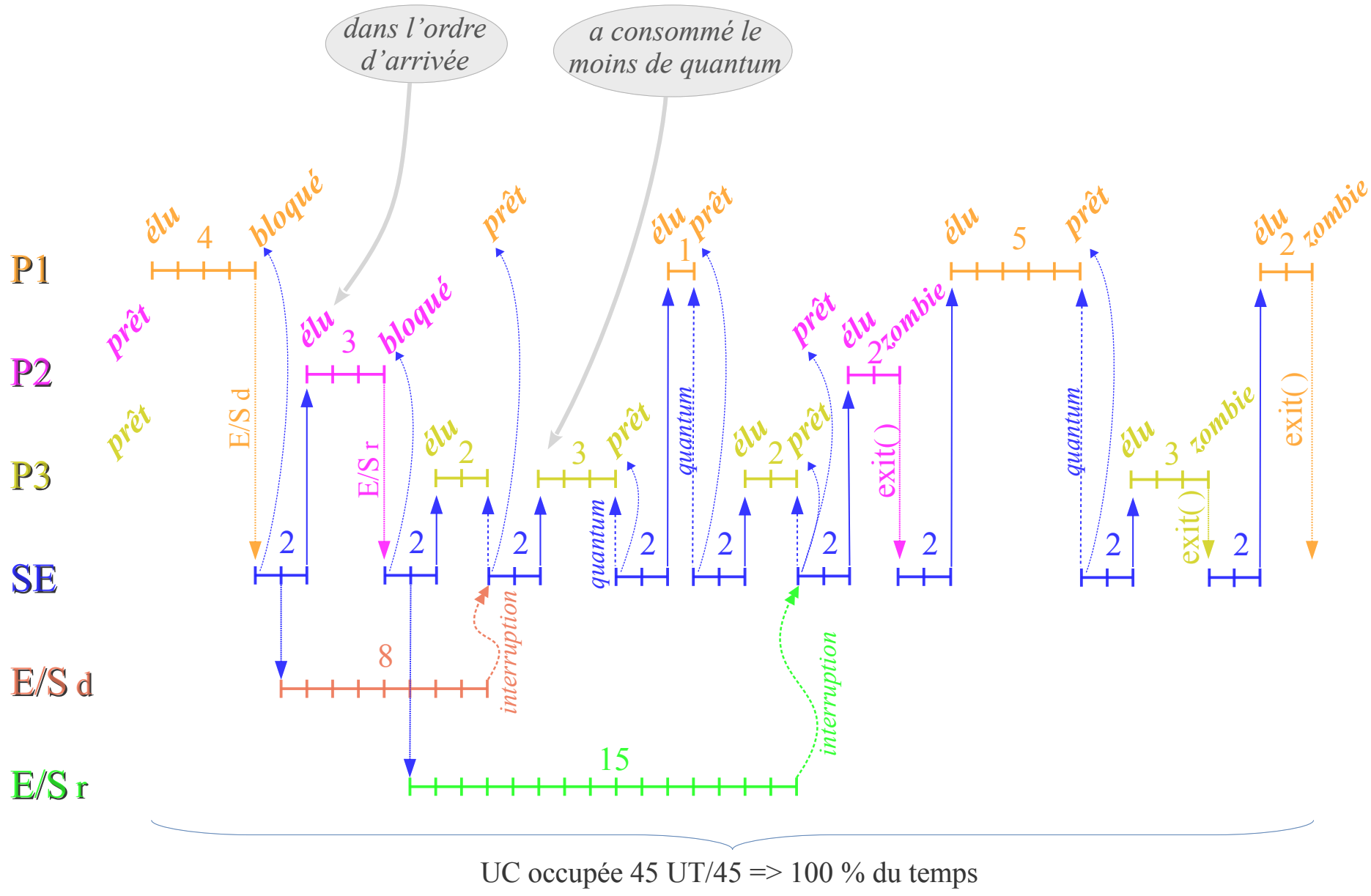
Chronogramme



Chronogramme

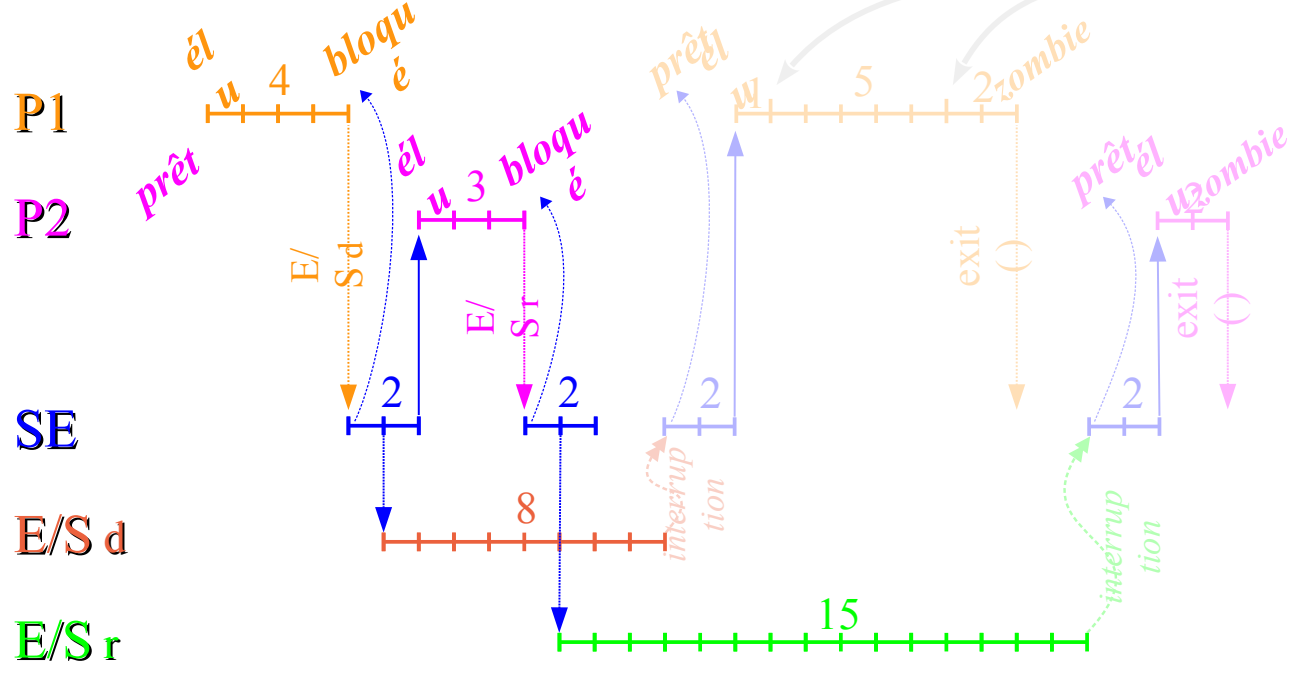


Chronogramme



Chronogramme 2

- que ce passe-t-il si P3 n'existe pas ?
- quel est alors le taux d'occupation de l'UC ?

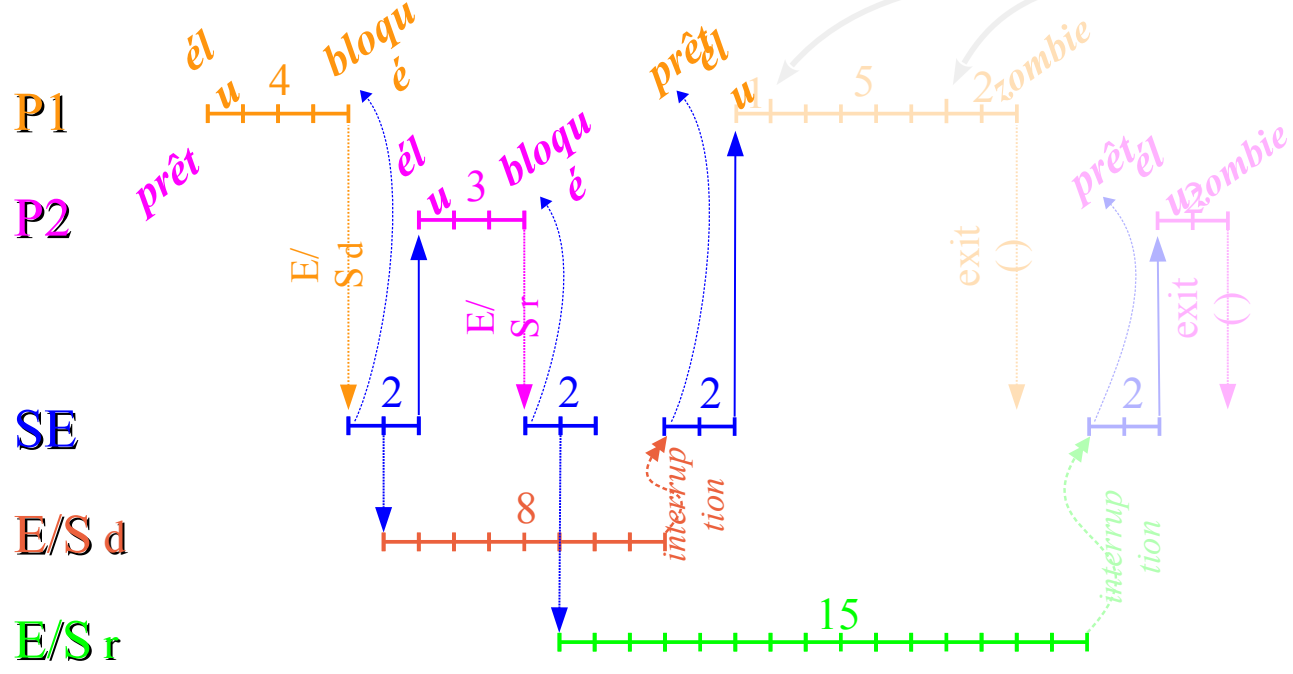


Taux d'occupation=temps où l'UC est occupée sur le temps total

UC occupée 25 UT/29 => 82 % du temps

Chronogramme 2

- que ce passe-t-il si P3 n'existe pas ?
- quel est alors le taux d'occupation de l'UC ?

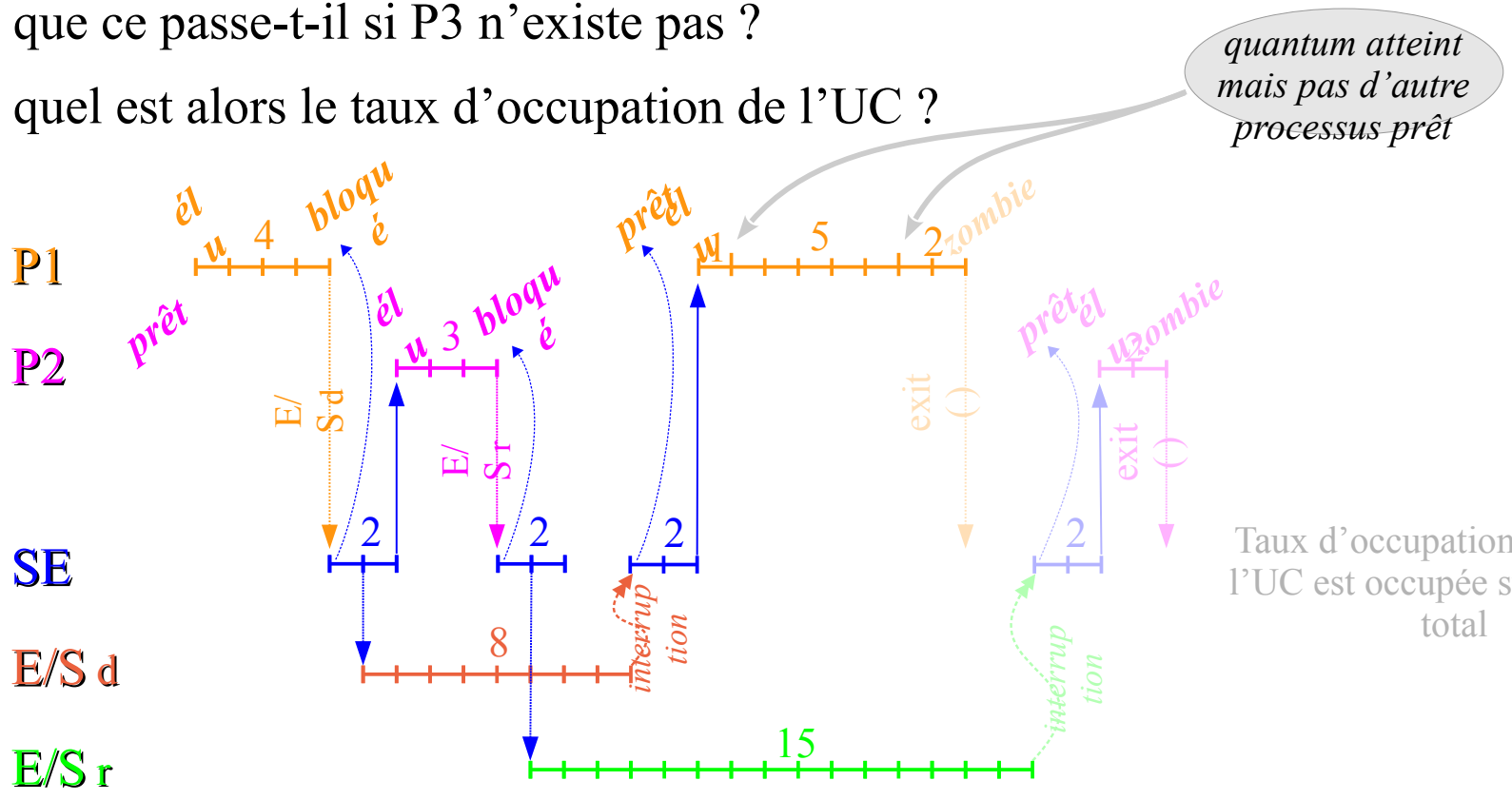


Taux d'occupation=temps où l'UC est occupée sur le temps total

UC occupée 25 UT/29 => 82 % du temps

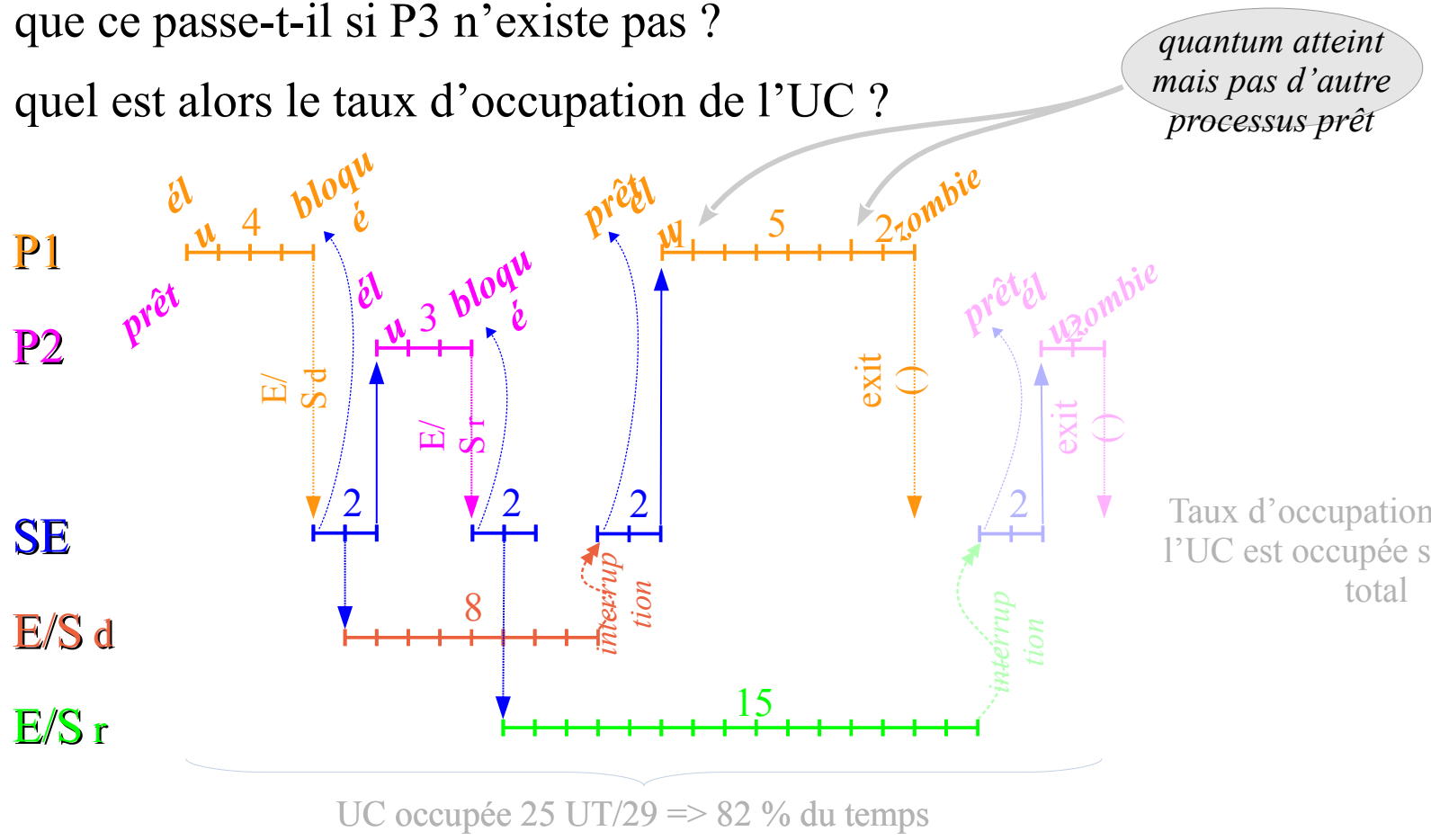
Chronogramme 2

- que ce passe-t-il si P3 n'existe pas ?
- quel est alors le taux d'occupation de l'UC ?



Chronogramme 2

- que ce passe-t-il si P3 n'existe pas ?
- quel est alors le taux d'occupation de l'UC ?

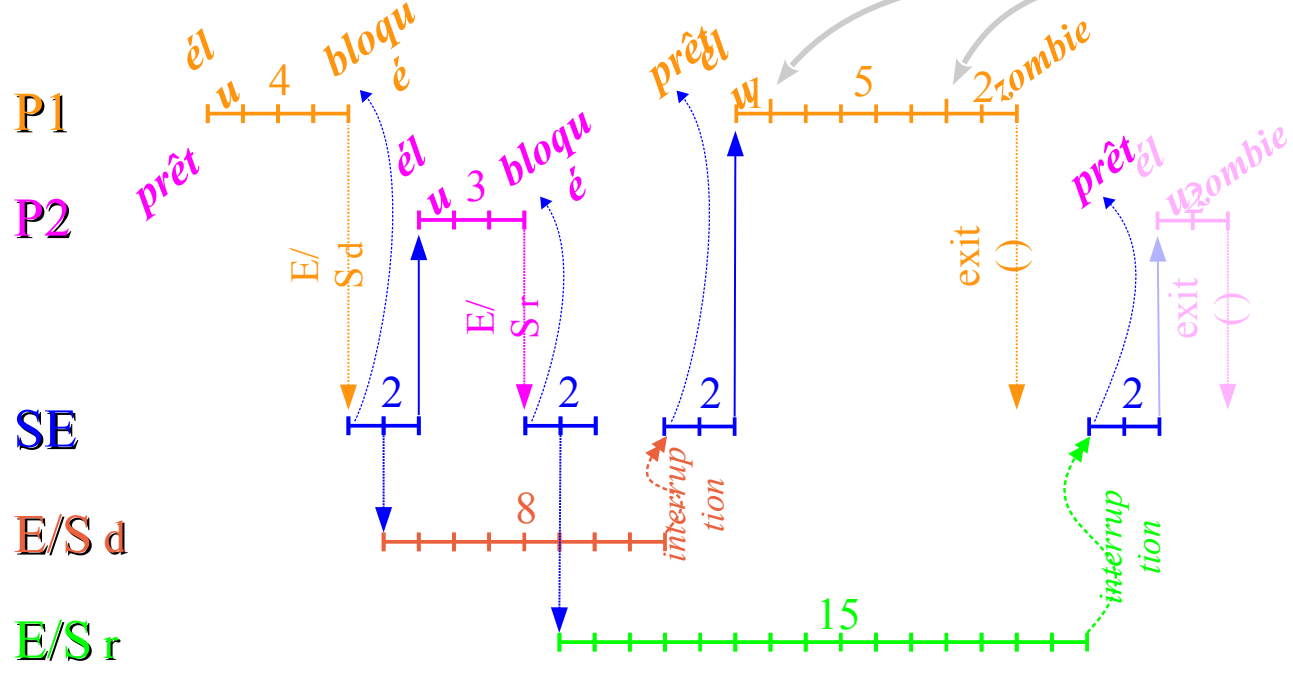


Taux d'occupation=temps où l'UC est occupée sur le temps total

UC occupée 25 UT/29 => 82 % du temps

Chronogramme 2

- que ce passe-t-il si P3 n'existe pas ?
- quel est alors le taux d'occupation de l'UC ?

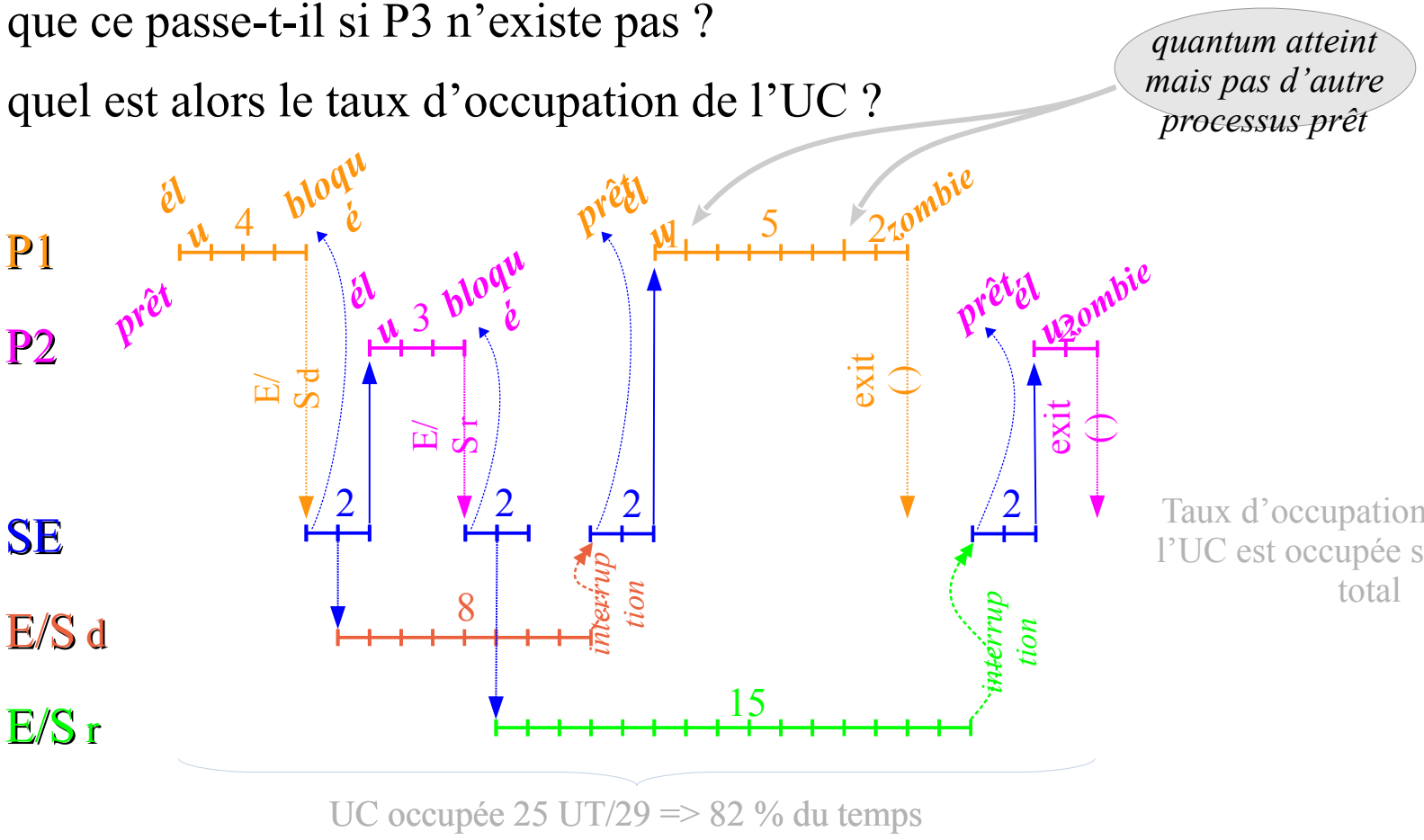


Taux d'occupation=temps où l'UC est occupée sur le temps total

UC occupée 25 UT/29 => 82 % du temps

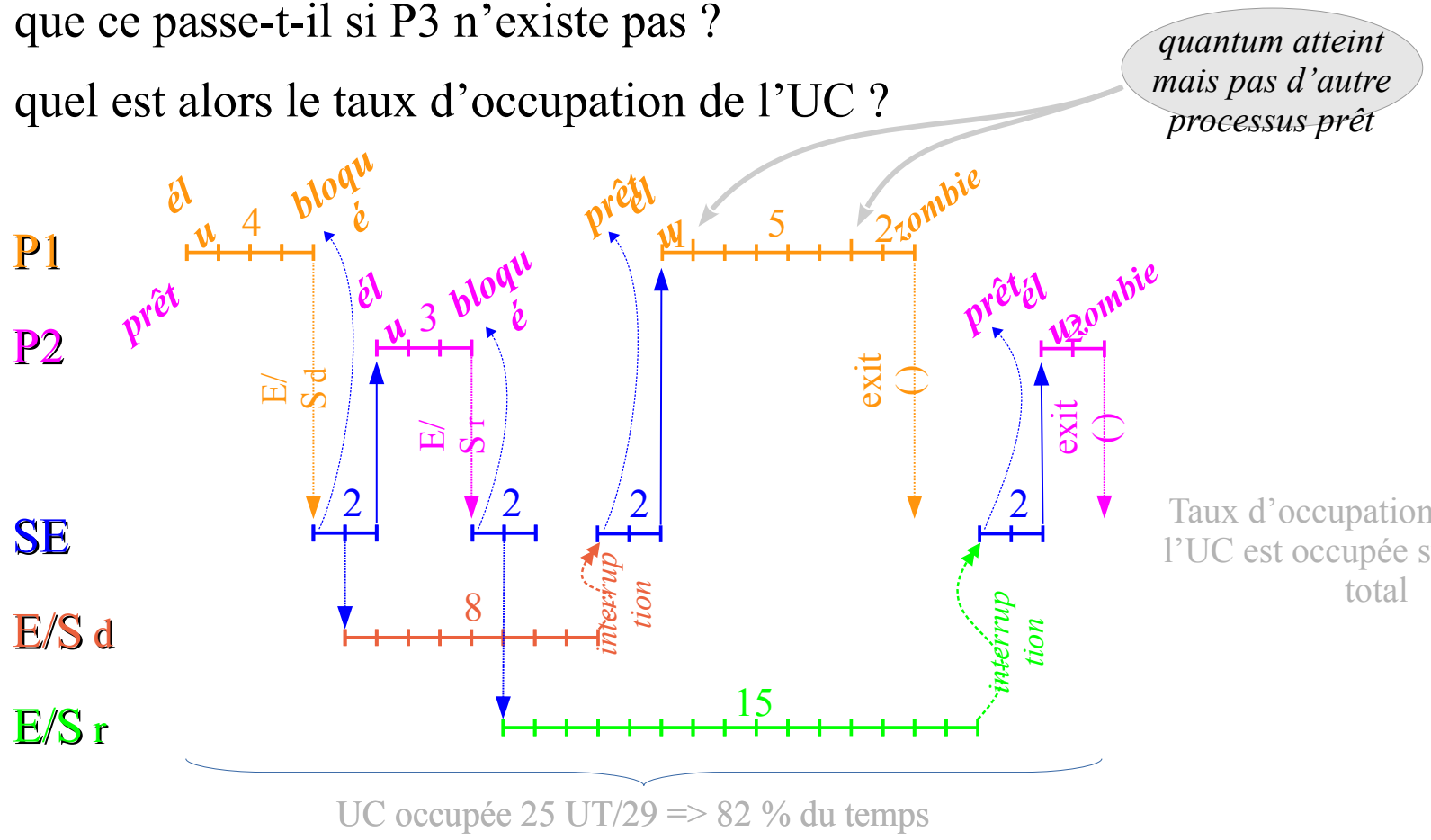
Chronogramme 2

- que ce passe-t-il si P3 n'existe pas ?
- quel est alors le taux d'occupation de l'UC ?



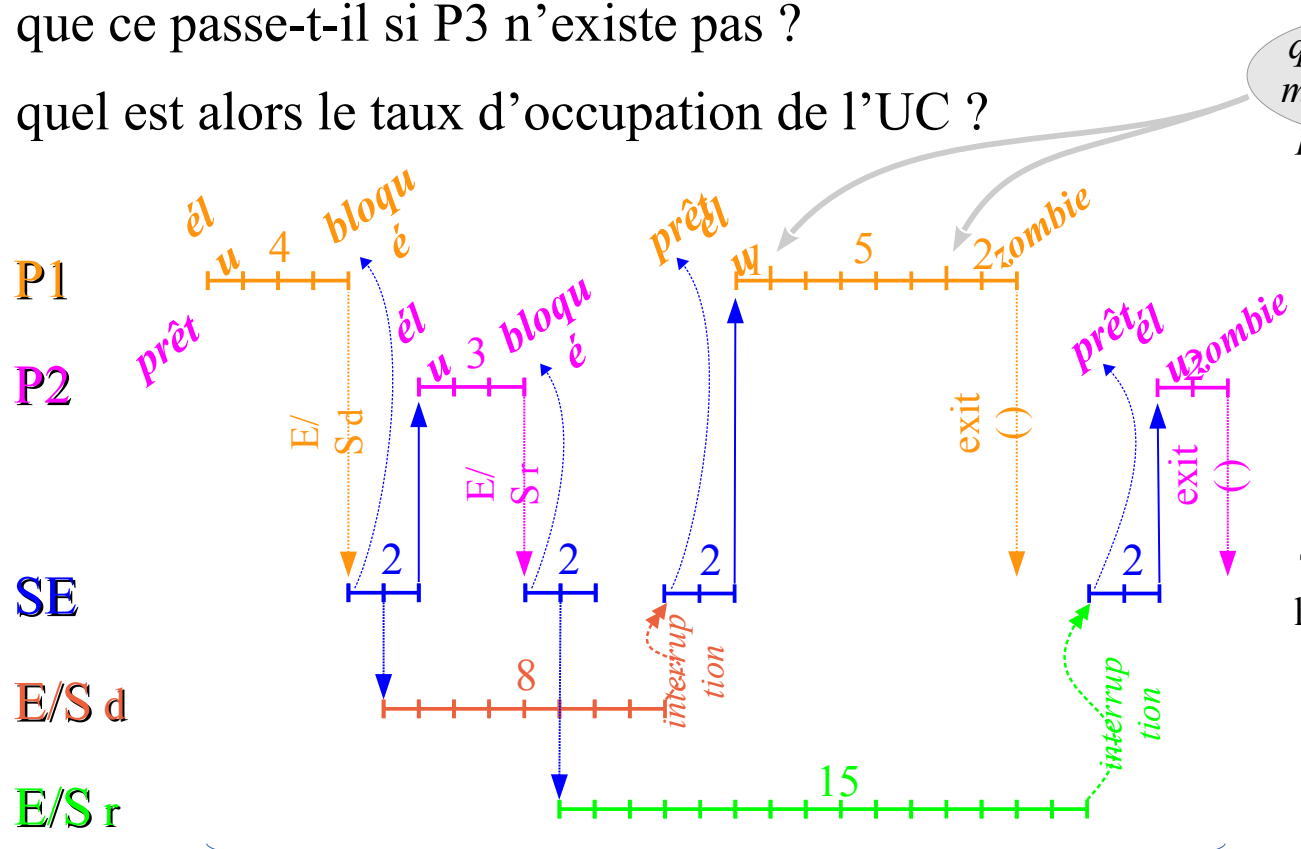
Chronogramme 2

- que ce passe-t-il si P3 n'existe pas ?
- quel est alors le taux d'occupation de l'UC ?



Chronogramme 2

- que ce passe-t-il si P3 n'existe pas ?
- quel est alors le taux d'occupation de l'UC ?



Taux d'occupation=temps où l'UC est occupée sur le temps total

UC occupée 25 UT/29 => 82 % du temps