

Institut de recherche sur les lois  
fondamentales de l'Univers

Université  
Paris-Saclay

cea

irfu

# Conception et programmation multitâches

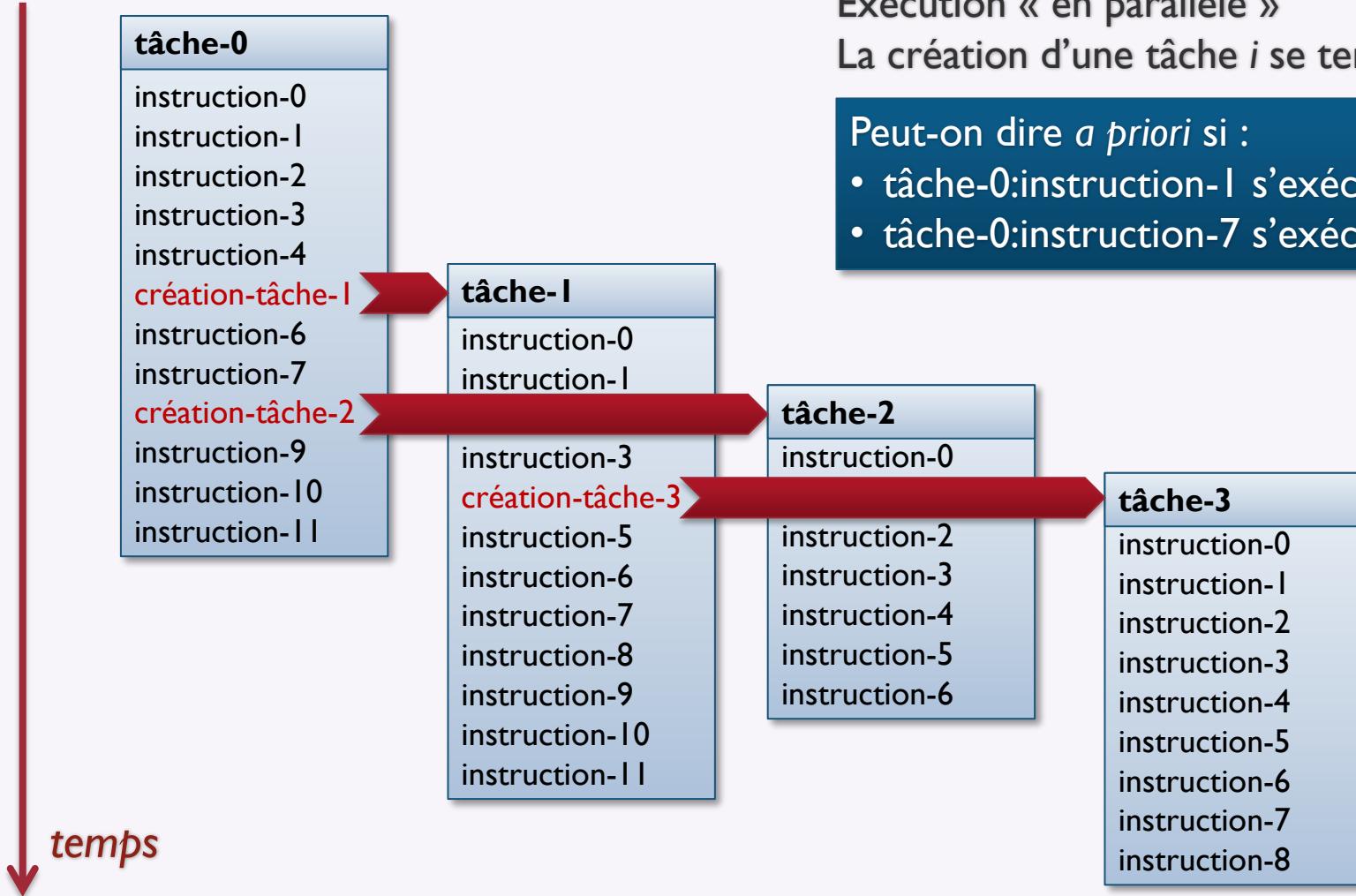


[shebli.anvar@cea.fr](mailto:shebli.anvar@cea.fr)  
+33 1 69 08 61 60  
+33 6 63 31 92 26

CEA Paris-Saclay  
91191 Gif-sur-Yvette  
France

cea  
irfu

# Caractéristiques d'une tâche



Exécution « en parallèle »

La création d'une tâche  $i$  se termine  $\cancel{\rightarrow}$  la tâche  $i$  a fini de s'exécuter

Peut-on dire *a priori* si :

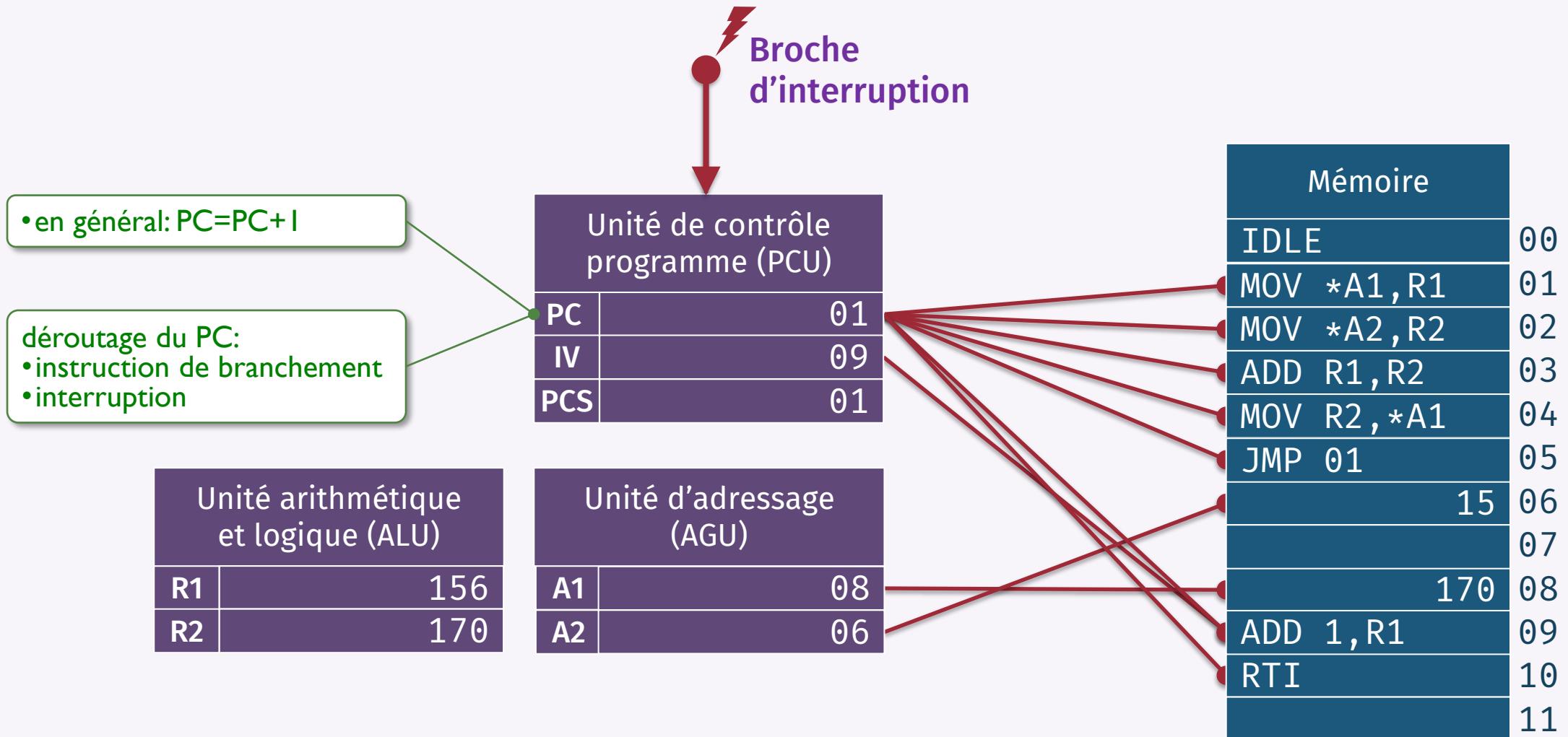
- tâche-0:instruction-1 s'exécute avant tâche-0:instruction-6 ?
- tâche-0:instruction-7 s'exécute avant tâche-1:instruction-6 ?

## OS MULTITÂCHES

Plusieurs tâches peuvent être exécutées sur *un seul core* en pseudo-parallèle.

Les tâches sont *préemptées* par une *interruption matérielle* dont la *routine d'ordonnancement* choisit à quelle tâche attribuer le CPU.

# Architecture électronique d'un processeur



## ■ **Appel régulier d'un ordonnanceur (scheduler)**

- Régularité assurée par une alarme électronique (hardware timer)
- TIC = période de cette alarme (typiquement quelques ms, en anglais: tick)

## ■ **Préemption: appel de l'ordonnanceur par interruption**

- L'alarme active un signal d'interruption
- La routine associée à l'alarme d'ordonnancement est le code de l'ordonnanceur

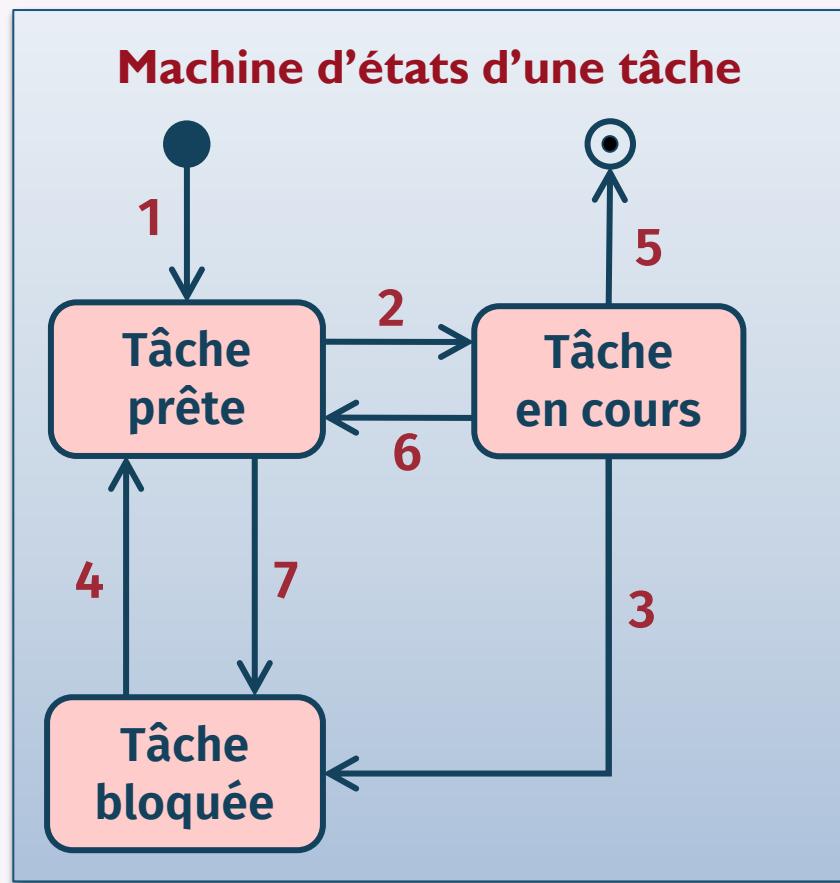
## ■ **Le code de l'ordonnanceur « active » une tâche (task, thread)**

- Sauvegarde le contexte de la tâche en cours
- Considère la liste des données liées à la gestion des tâches
- Décide par un algorithme de la tâche à activer
- Restaure le contexte de la tâche élue
- Contexte  $\supseteq$  valeurs des registres du processeur, notamment le PC (program counter)

# Ordonnanceur et états d'une tâche

Rôle de l'ordonnanceur : choix de la tâche à exécuter *parmi les tâches prêtes*

Différents algorithmes d'ordonnancement → différentes *politiques d'ordonnancement*



## Changements d'état d'une tâche

- Election
- Fin de la tâche
- Préemption
- Demande de ressource occupée, attente événement non présent, auto-suspension
- Création de la tâche
- Suspension
- Ressource libérée, événement arrivé, réactivation

# Séquence d'exécution – diagramme de Gantt

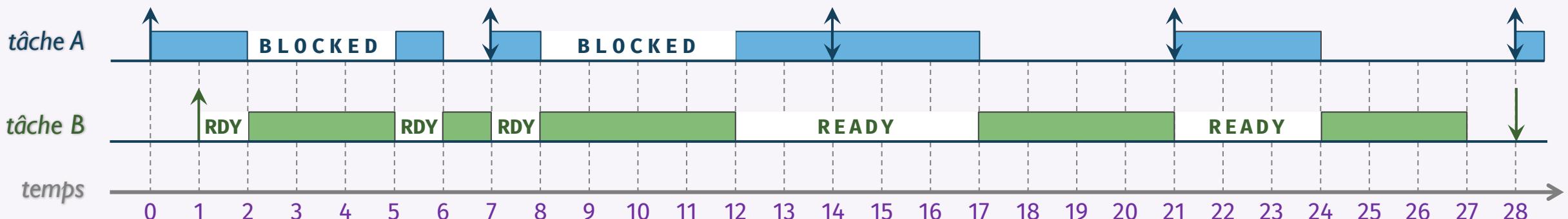
## ■ Séquence

- trace temporelle de l'exécution d'un ensemble de tâches (diagramme de Gantt)
- environnement de développement / débogage

## ■ Séquence valide

- lorsque chaque tâche est exécutée dans le respect de ses contraintes de temps (activation, délai)

ID tâche	$r_i$ date de départ	$c_i$ temps de départ	$R_i$ délai	$P_i$ période	$r_i + R_i$ échéance	$T_i$ temps d'exécution
A	0	0	7	7	7 ; 14 ; 21	3
B	1	2	27	-	28	15



Quelle est la tâche la plus prioritaire ? Pourquoi ?

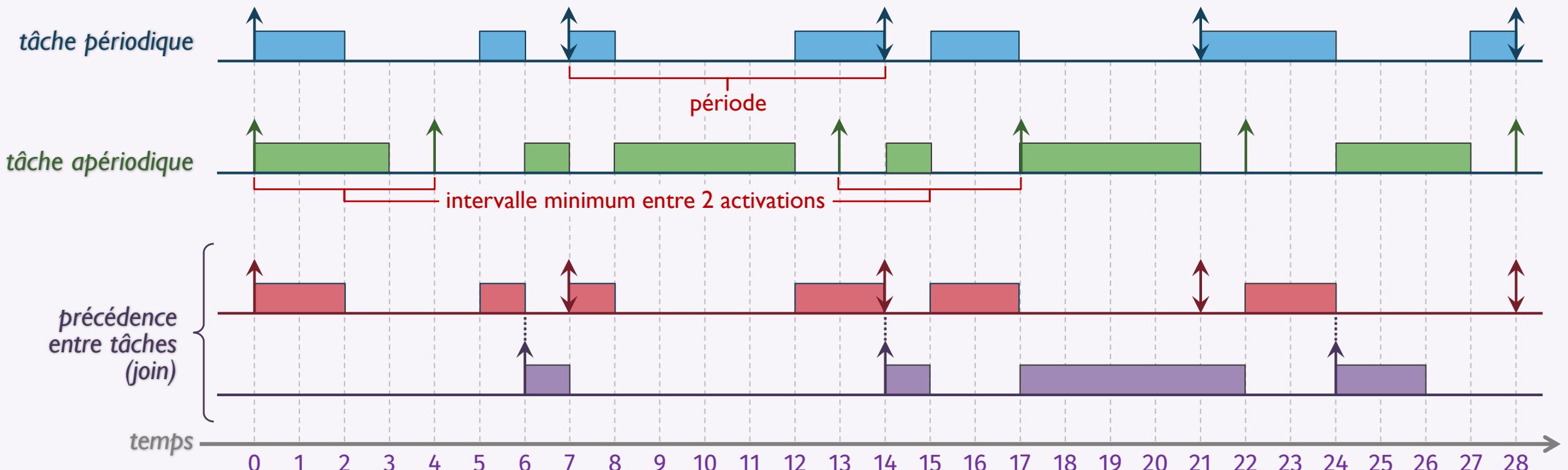
# Séquence d'exécution – diagramme de Gantt

## ■ Séquence

- trace temporelle de l'exécution d'un ensemble de tâches (diagramme de Gantt)
- environnement de développement / débogage

## ■ Séquence valide

- lorsque chaque tâche est exécutée dans le respect de ses contraintes de temps (activation, délai)



## ■ **Posix : norme pour API d'un OS multitâches**

- API de gestion du temps
  - Mesure du temps: `clock_gettime`
  - Timers: `timer_create`, `timer_settime`
  - `#include: <signal.h> <time.h>`
  - Librairie: `-lrt`
- API multitâches: librairie `pthread`
  - Création / destruction de tâche
  - Protection des accès concurrents par mutex
  - Activation / blocage des tâches par condition
  - `#include: <pthread.h>`
  - Librairie: `-lpthread`

## ■ **Aide sur le fonctions Posix sous Linux :**

- `man fonction`
- <http://man7.org/linux/man-pages/index.html>

## ■ **C'est une API en langage C**

- Programmation moderne dans un esprit « orienté objets »
- Mécanismes de « construction » et de « destruction »
- Complexité due à une programmation « orientée objets » en langage C
- Se prête très bien à une encapsulation C++

# Gestion du temps Posix

## Mesure du temps dans l'application

### ■ Temps présent absolu : `clock_gettime`

- temps absolu = temps écoulé depuis démarrage du processeur
- structure « timespec » à 2 champs : secondes et nanosecondes

Mesure de l'instant présent

```
struct timespec abstime;
clock_gettime(CLOCK_REALTIME, &abstime);
std::cout << "secondes: " << abstime.tv_sec << std::endl;
std::cout << "nanosecondes: ", abstime.tv_nsec << std::endl;
```

TD  
1

voir les autres possibilités avec `man clock_gettime`

Mesure d'une durée

```
struct timespec debut, fin, duree;
clock_gettime(CLOCK_REALTIME, &debut);
// . . . Section de code dont on veut mesurer la durée . . .
clock_gettime(CLOCK_REALTIME, &fin);
duree.tv_sec = fin.tv_sec - debut.tv_sec;
if (fin.tv_nsec < debut.tv_nsec) {
    duree.tv_sec -= 1;
    fin.tv_nsec += 1000000000;
}
duree.tv_nsec = fin.tv_nsec - debut.tv_nsec;
std::cout << "Measured duration is " << duree.tv_sec << " seconds and "
      << duree.tv_nsec << " nanoseconds equivalent to "
      << duree.tv_sec*1000. + duree.tv_nsec/1000000. << " milliseconds" << std::endl;
```

calcul de la différence de 2 instants

# Gestion du temps Posix

## Alarme à expiration unique ou périodique

### ■ Alarmes (Timers) POSIX

- émission d'un signal à l'expiration d'un délai temporel
- appel d'une fonction « handler » réagissant au signal

```
void myHandler(int sig, siginfo_t* si, void* ) { }
```

mon code avec accès à mes données dans `si->si_value`

```
struct sigaction sa;
sa.sa_flags = SA_SIGINFO;
sa.sa_sigaction = myHandler;
sigemptyset(&sa.sa_mask);
sigaction(SIGRTMIN, &sa, nullptr);
```

action à effectuer lorsque le timer arrive à échéance  
l'action est un appel de fonction avec paramètres  
pointeur sur la fonction à appeler (appelée *handler* ou *callback*)  
flags de blocage à 0 : aucun signal bloqué pendant l'exécution du *callback*  
action associée à un signal temps réel : entre SIGRTMIN et SIGRTMAX

```
struct sigevent sev;
sev.sigev_notify = SIGEV_SIGNAL;
sev.sigev_signo = SIGRTMIN;
sev.sigev_value.sival_ptr = (void*) &myData;
```

événement associé à l'expiration du timer  
l'événement sera de type « signal »  
numéro du signal émis : le même que celui associé à l'action  
données à transmettre au *handler* de type :  
`union sigval { int sival_int; void* sival_ptr; }`

```
timer_t tid;
timer_create(CLOCK_REALTIME, &sev, &tid);
itimerspec its;
its.it_value.tv_sec = 10;
its.it_value.tv_nsec = 0;
its.it_interval.tv_sec = 0;
its.it_interval.tv_nsec = 0;
timer_settime(tid, 0, &its, nullptr);
// . . . other instructions while waiting for the signal to arrive
timer_delete(tid);
```

identifiant du timer  
création du timer : prévoir destruction par `timer_delete`  
structure contenant la (les) durée(s) d'expiration du timer  
durée de la 1<sup>re</sup> expiration du timer : à zéro pour arrêter un timer déjà démarré  
durée de l'expiration périodique du timer : à zéro si timer non périodique  
démarrage du timer (on dit aussi « armement du timer ») ou arrêt si `it_value` est nul  
destruction du timer

TD  
2a, 2bCours OO  
S1→S4TD  
2c, 2d, 2e

## Tâches Posix

## pthread\_create : création d'une tâche

```
int pthread_create (pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg)
```

## main

instruction-0  
instruction-1  
instruction-2  
instruction-3  
instruction-4  
**création-tâche-1** → instruction-6  
instruction-7  
instruction-8  
instruction-9  
instruction-10  
instruction-11

temps

```
volatile bool stop = false;
pthread_t incrementThread;

pthread_create(&incrementThread, nullptr, incrementer, (void*) &stop);
```

→ **stop** = true;

```
void* incrementer(void* v_stop) {
    volatile bool* p_stop = (volatile bool*) v_stop;
    double counter = 0.0;
    while (not *p_stop) {
        counter += 1.0;
    }
    std::cout << "Counter value = " << counter << std::endl;
    return v_stop;
}
```

## incrementer

instruction-0  
instruction-1  
instruction-2  
instruction-3  
instruction-4  
instruction-5  
instruction-6  
instruction-7  
instruction-8  
instruction-9  
instruction-10  
instruction-11

# Tâche initiale

## ■ Initiale

- activée au lancement de l'application (typiquement la fonction main)
- crée l'environnement : tâches et ressources
- attends : fin des tâches ordinaires

► `int pthread_join (pthread_t th, void** retVal)`  
où `retVal` pointe sur la valeur de retour de la tâche (ou `nullptr`)

## ■ Ordinaire

- effectue un traitement
- liée à un événement externe ou interne
  - libération d'une ressource
  - libération d'une condition
  - périodique (activée par timer)
  - interruption
  - message

} Appel bloquant dans la boucle

## ■ Tâche de fond

- toujours active (pas en attente d'événement)
- exécutée lorsque toutes les autres tâches sont bloquées (priorité minimale)

## Tâches Posix

## pthread\_create : création d'une tâche

```
int pthread_create (pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg)
```

## main

instruction-0  
instruction-1  
instruction-2  
instruction-3  
instruction-4  
**création-tâche-1**  
instruction-6  
instruction-7  
instruction-8  
instruction-9  
instruction-10  
instruction-11

TD  
3a-1)

```
volatile bool stop = false;
pthread_t incrementThread;
```

creation et démarrage de la tâche

```
pthread_create(&incrementThread, nullptr, incrementer, (void*) &stop);
for (char cmd = 'r'; cmd != 's'; std::cin >> cmd)
    std::cout << "Type 's' to stop: " << std::flush;
stop = true;
```

attente de fin d'exécution de la tâche

```
void* incrementer(void* v_stop) {
    volatile bool* p_stop = (volatile bool*) v_stop;
    double counter = 0.0;
    while (not *p_stop) {
        counter += 1.0;
    }
    std::cout << "Counter value = " << counter << std::endl;
    return v_stop;
}
```

## incrementer

instruction-0  
instruction-1  
instruction-2  
instruction-3  
instruction-4  
instruction-5  
instruction-6  
instruction-7  
instruction-8  
instruction-9  
instruction-10  
instruction-11

temps

## pthread\_create : partage de données entre tâches

```
int pthread_create (pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg)
```

## main

instruction-0  
instruction-1  
instruction-2  
instruction-3  
instruction-4  
**création-tâche-1** → instruction-6  
instruction-7  
instruction-8  
instruction-9  
instruction-10  
instruction-11

temps

```
struct Data {
    volatile bool stop;
    volatile double counter;
};

Data data = { false, 0.0 };
pthread_t incrementThread;

pthread_create(&incrementThread, nullptr, incrementer, &data);
```

**data.stop** = true;  
~~pthread\_join(incrementThread, nullptr);~~  
 std::cout << "Counter value = " << data.counter << std::endl;

```
void* incrementer(void* v_data) {
    Data* p_data = (Data*) v_data;
    while (not p_data->stop) {

        p_data->counter += 1.0;

    }
    return v_data;
}
```

accès concurrents  
à la variable data

## incrementer

instruction-0  
instruction-1  
instruction-2  
instruction-3  
instruction-4  
instruction-5  
instruction-6  
instruction-7  
instruction-8  
instruction-9  
instruction-10  
instruction-11

## pthread\_create : partage de données entre tâches

```
int pthread_create (pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg)
```

main
instruction-0
instruction-1
instruction-2
création-tâche-1 →
création-tâche-2 →
création-tâche-3 →
instruction-6
instruction-7
instruction-8
instruction-9
instruction-10
instruction-11

temps



```
struct Data {
    volatile bool stop;
    volatile double counter;
};

Data data = { false, 0.0 };
pthread_t incrementThread[3];
pthread_create(&incrementThread[0], nullptr, incrementer, &data);
pthread_create(&incrementThread[1], nullptr, incrementer, &data);
pthread_create(&incrementThread[2], nullptr, incrementer, &data);

for (char cmd = 'r'; cmd != 's'; std::cin >> cmd)
    std::cout << "Type 's' to stop: " << std::flush;
data.stop = true;
for (int i=0; i<3; ++i) pthread_join(incrementThread[i], nullptr);
std::cout << "Counter value = " << data.counter << std::endl;

void* incrementer(void* v_data) {
    Data* p_data = (Data*) v_data;
    while (not p_data->stop) {
        p_data->counter += 1.0;
    }
    return v_data;
}
```

tableau de 3 tâches

les 3 tâches exécutent la même fonction et accèdent aux mêmes données

accès concurrents à la variable data

TD 3a-2)

incrementer
incrementer
incrementer
instruction-0
instruction-1
instruction-2
instruction-3
instruction-4
instruction-5
instruction-6
instruction-7
instruction-8
instruction-9
instruction-10
instruction-11

## pthread\_mutex : accès « thread safe » à des données

```
int pthread_mutex_init (pthread_mutex_t* mutex, const pthread_mutexattr_t* attr)
```

main
instruction-0
instruction-1
instruction-2
création-tâche-1 →
création-tâche-2 →
création-tâche-3 →
instruction-6
instruction-7
instruction-8
instruction-9
instruction-10
instruction-11

```
struct Data {
    volatile bool stop;
    volatile double counter;
    pthread_mutex_t mutex;
};

Data data = { false, 0.0 }; pthread_mutex_init(&data.mutex, nullptr);
pthread_t incrementThread[3];
pthread_create(&incrementThread[0], nullptr, incrementer, &data);
pthread_create(&incrementThread[1], nullptr, incrementer, &data);
pthread_create(&incrementThread[2], nullptr, incrementer, &data);
for (char cmd = 'r'; cmd != 's'; std::cin >> cmd)
    std::cout << "Type 's' to stop: " << std::flush;
data.stop = true; for (int i=0; i<3; ++i) pthread_join(incrementThread[i], nullptr);
pthread_mutex_destroy(&data.mutex);
std::cout << "Counter value = " << data.counter << std::endl;

void* incrementer(void* v_data) {
    Data* p_data = (Data*) v_data;
    while (not p_data->stop) {
        pthread_mutex_lock(&p_data->mutex);
        p_data->counter += 1.0;
        pthread_mutex_unlock(&p_data->mutex);
    }
    return v_data;
}
```

TD 3a-3)

ajout d'un mutex pour protéger la structure de données partagées

construction du mutex ; prévoir sa destruction après utilisation

verrouillage du mutex avant accès aux données partagées

section critique d'accès à data

déverrouillage du mutex après accès aux données partagées

temps



## pthread\_mutex : accès « thread safe » à des données

Toute ressource partagée par plus d'un thread doit être associée à un mutex

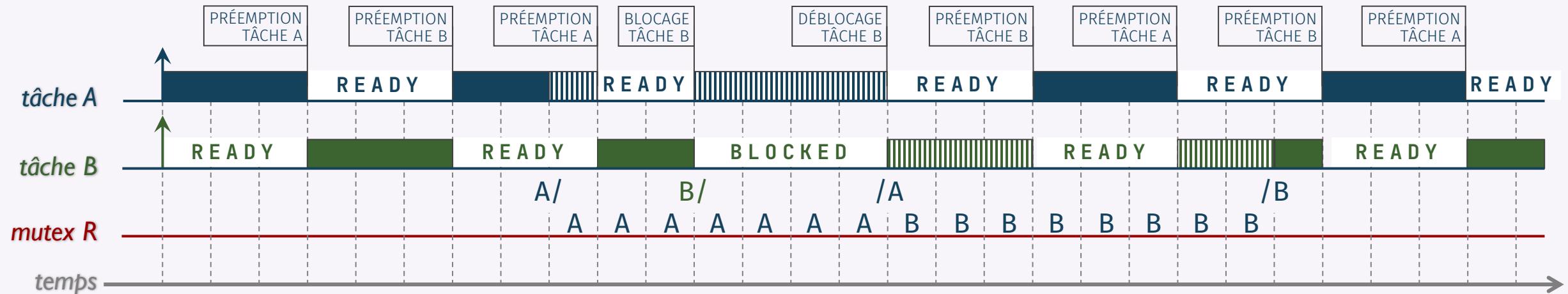
Ressource partagée = variable / donnée partagée (de n'importe quel type)

tâche A

```
void* incrementer(void* v_data)
{
    Data* p_data = (Data*) v_data;
    while (not *p_data->stop) {
        pthread_mutex_lock(&p_data->mutex);
        *p_data->counter += 1.0;
        pthread_mutex_unlock(&p_data->mutex);
    }
    return v_data;
}
```

tâche B

```
void* incrementer(void* v_data)
{
    Data* p_data = (Data*) v_data;
    while (not *p_data->stop) {
        pthread_mutex_lock(&p_data->mutex);
        *p_data->counter += 1.0;
        pthread_mutex_unlock(&p_data->mutex);
    }
    return v_data;
}
```



## pthread\_mutex : différentes sortes de mutex

Toute ressource partagée par plus d'un thread doit être associée à un mutex

Ressource partagée = variable / donnée partagée (de n'importe quel type)

type de mutex	2 <sup>e</sup> tentative de <i>lock</i> par une tâche détenant déjà le jeton	tentative de <i>unlock</i> par une tâche ne détenant pas le jeton
PTHREAD_MUTEX_DEFAULT	Undefined	Undefined
PTHREAD_MUTEX_NORMAL	Deadlock	Undefined
PTHREAD_MUTEX_ERRORCHECK	Error returned	Error returned
PTHREAD_MUTEX_RECURSIVE	Lock count	Error returned

```

pthread_mutexattr_t mutexAttribute;
pthread_mutexattr_init(&mutexAttribute);
pthread_mutexattr_settype(&mutexAttribute, PTHREAD_MUTEX_RECURSIVE); • initialisation d'un attribut de mutex de type récursif

pthread_mutex_t mutex; • initialisation d'un mutex avec l'attribut initialisé
pthread_mutex_init(&mutex, &mutexAttribute);

pthread_mutexattr_destroy(&mutexAttribute); • destruction de l'attribut une fois le mutex initialisé
// . . .
pthread_mutex_lock(&mutex); • utilisation du mutex
// . . .
pthread_mutex_unlock(&mutex); • destruction du mutex
// . . .
pthread_mutex_destroy(&mutex);

```

## pthread\_mutex : verrouillage avec timeout

**Toute ressource partagée par plus d'un thread doit être associée à un mutex**

Ressource partagée = variable / donnée partagée (de n'importe quel type)

### Le verrouillage de mutex étant potentiellement bloquant : verrouillage avec timeout

```
int pthread_mutex_timedlock (pthread_mutex_t* mutex, const struct timespec* timeout)
```

**ATTENTION** : la variable *timeout* est une **échéance** et non un délai ; il s'agit d'un temps processeur absolu (temps écoulé depuis le boot)

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, nullptr);
timespec deadline;
clock_gettime(CLOCK_REALTIME, &deadline);
deadline.tv_sec  += 2;
deadline.tv_nsec += 500000000;
if (deadline.tv_nsec >= 1000000000) {
    deadline.tv_sec  += 1;
    deadline.tv_nsec -= 1000000000;
}
if (pthread_mutex_timedlock(&mutex, & deadline) == ETIMEDOUT) {
    std::cout << "Timeout!" << std::endl;
} else {
    // ...
    pthread_mutex_unlock(&mutex);
}
pthread_mutex_destroy(&mutex);
```

Cours OO  
S5 → S10

TD  
3b

## Tâches Posix

## pthread\_create : tâche temps réel et priorité

modification de l'ordonnancement après création de la tâche

```
int pthread_setschedparam (pthread_t tid, int policy, const struct schedparam* p_schedParam)
int pthread_getschedparam (pthread_t tid, int* p_policy, struct schedparam* p_schedParam)
```

Politique d'ordonnancement (scheduling policy)	Type	Caractéristiques	Palette de priorités	Remarques
SCHED_OTHER	Statistique	Toutes tâches prêtes élues à tour de rôle	0	Ordonnancement par défaut
SCHED_RR	Temps réel	Tâches de même priorité : à tour de rôle	1 .. 99 (Linux)	Nécessite privilège root
SCHED_FIFO	Temps réel	Tâches de même priorité : 1 <sup>er</sup> arrivé, 1 <sup>er</sup> servi	1 .. 99 (Linux)	Nécessite privilège root

Posix minimum 32

En général, on choisit une seule politique d'ordonnancement pour toutes les tâches :

- la politique d'ordonnancement est fixée au niveau du main et toutes les tâches créées par la suite héritent de cette politique
- ordonnancement temps réel  $\Rightarrow$  respect strict des priorités : si  $prio(A) > prio(B)$ , B n'est pas élue tant que A n'est pas bloquée

```
int main()
{
    struct sched_param schedParam;
    schedParam.sched_priority = sched_get_priority_max(SCHED_RR);
    pthread_setschedparam(pthread_self(), SCHED_RR, &schedParam);
    // À partir d'ici, toute nouvelle tâche hérite de main (ici SCHED_RR)
    // . . .
    return 0;
}
```

Priorité max: pourquoi ?

**OS TEMPS RÉEL (RTOS)**  
=  
respect strict des priorités  
+  
latences garanties pour  
appel d'interruption et context switch

Description détaillée des politiques d'ordonnancement

man 7 sched

## pthread\_create : tâche temps réel et priorité

```
int pthread_create (pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg)
```

modification de l'ordonnancement *avant* création de la tâche

**main**

instruction-0  
instruction-1  
instruction-2  
instruction-3  
instruction-4  
**création-tâche-1** → instruction-6  
instruction-7  
instruction-8  
instruction-9  
instruction-10  
instruction-11

*temps*

```

volatile bool stop = false;
pthread_attr_t attr;
pthread_attr_init(&attr);
```

construction d'un attribut pthread  
(prévoir la destruction avec pthread\_attr\_destroy)

```

sched_param schedParams;
schedParams.sched_priority = 9;
pthread_attr_setschedparam(&attr, &schedParams);
pthread_t incrementThread;
pthread_create(&incrementThread, &attr, incrementer, (void*) &stop);
for (char cmd = 'r'; cmd != 's'; std::cin >> cmd)
    std::cout << "Type 's' to stop: " << std::flush;
stop = true;
pthread_attr_destroy(&attr);
```

application de paramètres d'ordonnancement (priorité)

```

pthread_join(incrementThread, NULL);

void* incrementer(void* v_stop) {
    volatile bool* p_stop = (volatile bool*) v_stop;
    double counter = 0.0;
    while (not *p_stop) {
        counter += 1.0;
    }
    std::cout << "Counter value = " << counter << std::endl;
    return v_stop;
}
```

destruction de l'attribut pthread

Cours OO  
S11

TD  
3c, 3d

**■ Soient A, B et C trois tâches, et R une ressource telles que**

- priorité(A) > priorité(B) > priorité(C)
- R est accédée par les tâches A et C
- $t$  est le temps en tics système,  $t \in \{0, 1, 2, \dots\}$

**■ Caractéristiques de la tâche A**

- la tâche A est activée à  $t = 30$
- le temps d'exécution de A est de 40 tics
- le délai de A est de 60 tics
- la tâche A demande l'accès à R après 10 tics d'exécution
- après avoir obtenu l'accès à R, A libère l'accès à R au bout de 10 tics d'exécution

**■ Caractéristiques de la tâche B**

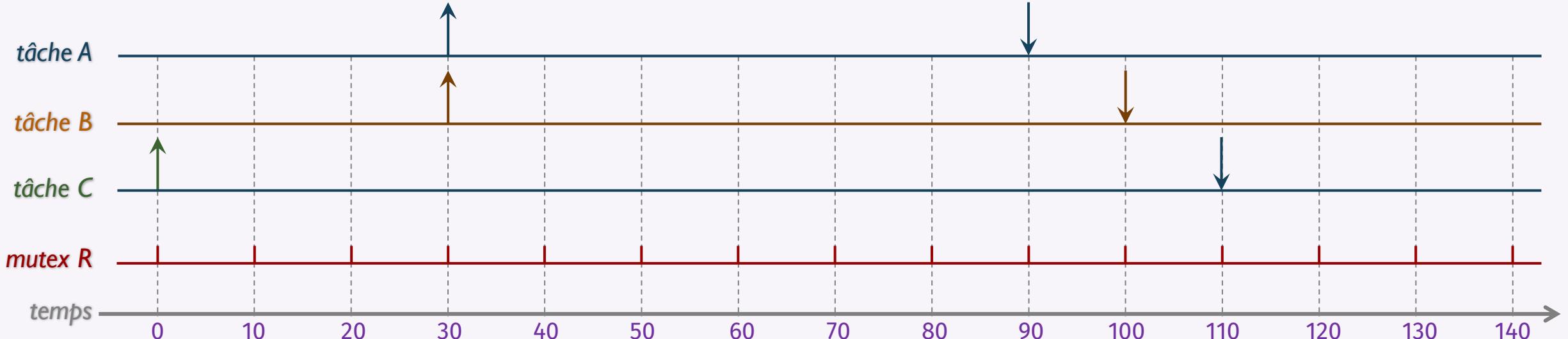
- la tâche B est activée à  $t = 30$
- le temps d'exécution de B est de 10 tics
- le délai de B est de 70 tics

**■ Caractéristiques de la tâche C**

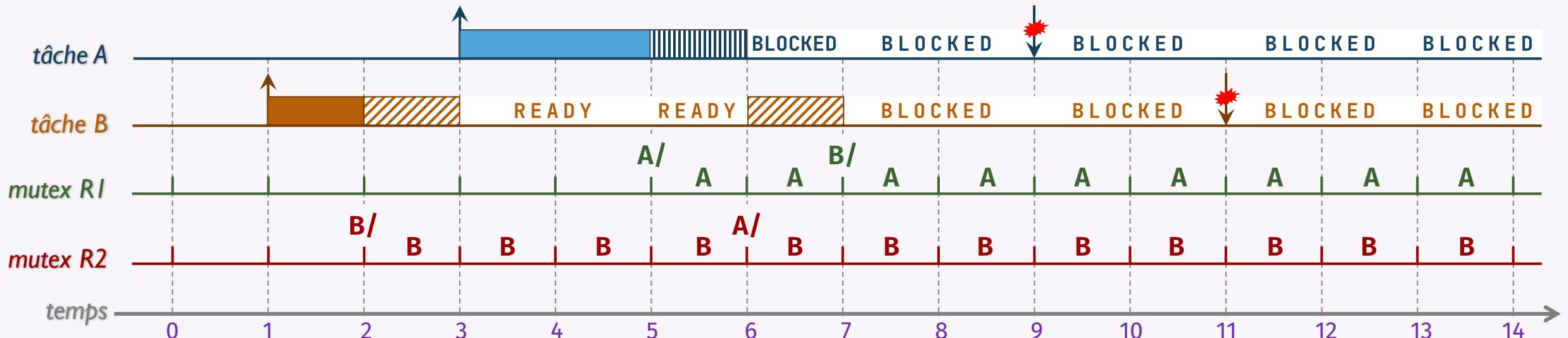
- la tâche C est activée à  $t = 0$
- le temps d'exécution de C est de 50 tics
- le délai de C est de 110 tics
- la tâche C demande l'accès à R après 20 tics d'exécution
- après avoir obtenu l'accès à R, C libère l'accès à R au bout de 20 tics d'exécution

# Inversion de priorité

## Établissez le diagramme de Gantt du scénario proposé



# Interblocage – Deadlock



solution : contraintes de programmation

- prise & libération *atomique* des ressources : *pas plus d'une ressource à la fois*
- prise et libération d'un ensemble de ressources dans un *ordre prédéterminé récursif*: R1/ → R2/ → R3/ → /R3 → /R2 → /R1
- dans tous les cas : protection contre le blocage par *timeout*

TD  
4

# Principaux types de tâche

## ■ Initiale

- activée au lancement de l'application (typiquement la fonction main)
- crée l'environnement : tâches et ressources
- attends : fin des tâches ordinaires

► `int pthread_join (pthread_t th, void** retVal)`  
où `retVal` pointe sur la valeur de retour de la tâche (ou `nullptr`)

## ■ Ordinaire

- effectue un traitement
- liée à un événement externe ou interne
  - libération d'une ressource
  - libération d'une condition
  - périodique (activée par timer)
  - interruption
  - message

} Appel bloquant dans la boucle

## ■ Tâche de fond

- toujours active (pas en attente d'événement)
- exécutée lorsque toutes les autres tâches sont bloquées (priorité minimale)

# Principaux types de tâche

## ■ Initiale

- activée au lancement de l'application (typiquement la fonction main)
- crée l'environnement : tâches et ressources
- attends : fin des tâches ordinaires

► `int pthread_join (pthread_t th, void** retVal)`  
où `retVal` pointe sur la valeur de retour de la tâche (ou `nullptr`)

## ■ Ordinaire

- effectue un traitement
- liée à un événement externe ou interne
  - ▶ libération d'une ressource
  - ▶ libération d'une condition
  - ▶ périodique (activée par timer)
  - ▶ interruption
  - ▶ message

} Appel bloquant dans la boucle

## ■ Tâche de fond

- toujours active (pas en attente d'événement)
- exécutée lorsque toutes les autres tâches sont bloquées (priorité minimale)

# Principaux types de tâche

## ■ Initiale

- activée au lancement de l'application (typiquement la fonction main)
- crée l'environnement : tâches et ressources
- attends : fin des tâches ordinaires

► `int pthread_join (pthread_t th, void** retVal)`  
où `retVal` pointe sur la valeur de retour de la tâche (ou NULL)

## ■ Ordinaire

- effectue un traitement
- liée à un événement externe ou interne
  - libération d'une ressource
  - libération d'une condition
  - périodique (activée par timer)
  - interruption
  - message

Appel bloquant dans la boucle

## ■ Tâche de fond

- toujours active (pas en attente d'événement)
- exécutée lorsque toutes les autres tâches sont bloquées (priorité minimale)

## pthread\_cond : attente &amp; notification sur condition

## ■ Principe

- Ressource R partagée par plusieurs tâches  $t_0, t_1, \dots, t_{n-1}$ 
  - Protégée par un mutex
- Condition C(R) sur la ressource
  - Tant que C(R) n'est pas réalisée,  $t_0$  reste dans l'état bloqué
  - Lorsqu'une tâche  $t_k$  modifie r permettant à C(R) d'être réalisée,  $t_k$  notifie  $t_0$  et  $t_0$  teste C(R)

```
struct Counter {
    volatile int value; • Pourquoi « volatile » ?
    pthread_mutex_t mutex; // Protection multitâches de la ressource Counter
    pthread_cond_t isEmpty; // Condition sur la ressource Counter
};

int main() {
    pthread_t th[3];
    Counter counter = {1000};
    pthread_mutex_init(&counter.mutex, nullptr);
    pthread_cond_init(&counter.isEmpty, nullptr); • initialisation de la condition
    pthread_create(&th[0], 0, monitor, &counter); // Message à l'écran quand compteur vide
    pthread_create(&th[1], 0, consumer, &counter); // Décrémente le compteur jusqu'à 0
    pthread_create(&th[2], 0, consumer, &counter); // Décrémente le compteur jusqu'à 0
    for (int i=0; i < 3; ++i) pthread_join(th[i], nullptr); // Attente de fin d'exécution des tâches
    pthread_mutex_destroy(&counter.mutex); • destruction de la condition
    pthread_cond_destroy(&counter.isEmpty); • et de son mutex associé
}
```

# **pthread\_cond : attente & notification sur condition**

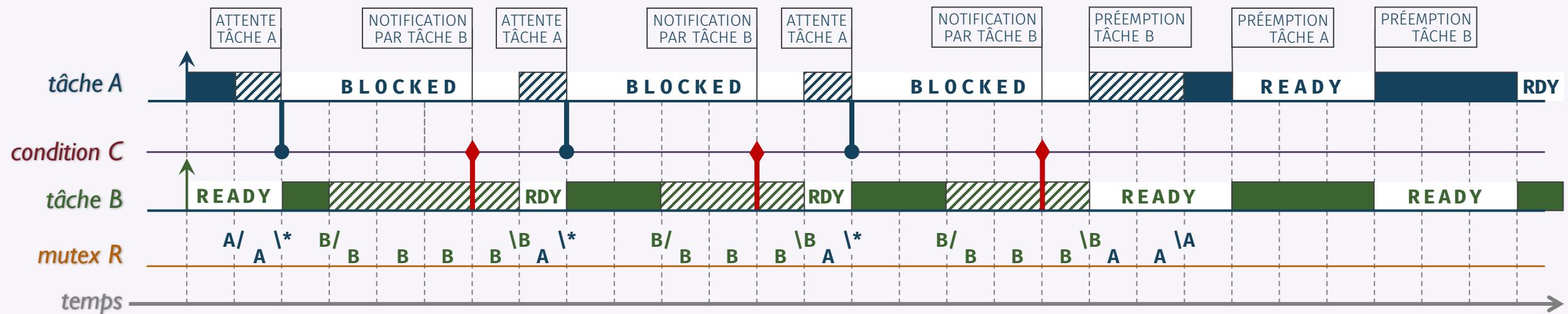
## tâches A1, A2, A3 ... Ap

```
void* monitor(void* v_counter) {
    Counter* p_counter = (Counter*) v_counter;
    pthread_mutex_lock(&p_counter->mutex);
    while (p_counter->value > 0) {
        pthread_cond_wait(&p_counter->isEmpty,
                           &p_counter->mutex);
    }
    pthread_mutex_unlock(&p_counter->mutex);
    std::cout<<"Counter is empty!"<<std::endl;
    return v_counter;
}
```

tâches B1, B2, B3 ... Bq

```
void* consumer(void* v_counter) {
    Counter* p_counter = (Counter*) v_counter;

    while (true) {
        pthread_mutex_lock(& p_counter->mutex);
        if (p_counter->value > 0) {
            p_counter->value -= 1; // Décrémentation
            pthread_cond_signal(&p_counter->isEmpty);
        }
        pthread_mutex_unlock(& p_counter->mutex);
    }
    return v_counter;
}
```



## Tâches Posix

## pthread\_cond : attente avec timeout &amp; notification « all »

tâches A1, A2, A3 ... Ap

```
void* monitor(void* v_counter) {
    Counter* p_counter = (Counter*) v_counter;

    timespec abstime;
    // Obtention du temps présent absolu
    clock_gettime(CLOCK_REALTIME, &abstime);
    // Timeout de 100 millisecondes
    abstime.tv_nsec += 100000000;
    if (abstime.tv_nsec > 1000000000) {
        abstime.tv_sec += 1;
        abstime.tv_nsec -= 1000000000;
    }
}
```

calcul de l'**échéance** en considérant  
un délai d'attente maximum de **0.1 seconde**

```
pthread_mutex_lock(&p_counter->mutex);
while (p_counter->value > 0) {
    pthread_cond_timedwait(&p_counter->isEmpty,
                           &p_counter->mutex,
                           &abstime);
}
```

```
pthread_mutex_unlock(&p_counter->mutex);
```

```
std::cout << "Counter is empty!" << std::endl;
return v_counter;
}
```

tâches B1, B2, B3 ... Bq

```
void* consumer(void* v_counter) {
    Counter* p_counter = (Counter*) v_counter;
```

```
while (true) {
    pthread_mutex_lock(& p_counter->mutex);
    if (p_counter->value > 0) {
        p_counter->value -= 1; // Décrémentation
        pthread_cond_broadcast(
            &p_counter->isEmpty);
    }
    pthread_mutex_unlock(& p_counter->mutex);
}

return v_counter;
}
```

exemple d'une tâche avec un délai de 5 tics et une période de 5 tics

