Conception Multitâche

Shebli Anvar PhD.
CEA Irfu, Centre de Saclay
91191 Gif-sur-Yvette
France
shebli.anvar@cea.fr
+33 1 69 08 78 32
+33 6 63 31 92 26
+33 1 69 08 31 47

Mesure du temps : clock_gettime

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

voir les autres possibilités avec `man clock_gettime`

```
#include <time.h>

struct timespec abstime;
clock_gettime(CLOCK_REALTIME, &abstime);
printf("secondes: %d\n", abstime.tv_sec);
printf("nanosecondes: %d\n", abstime.tv_nsec);
```


© Shebli ANVAR, CEA Irfu

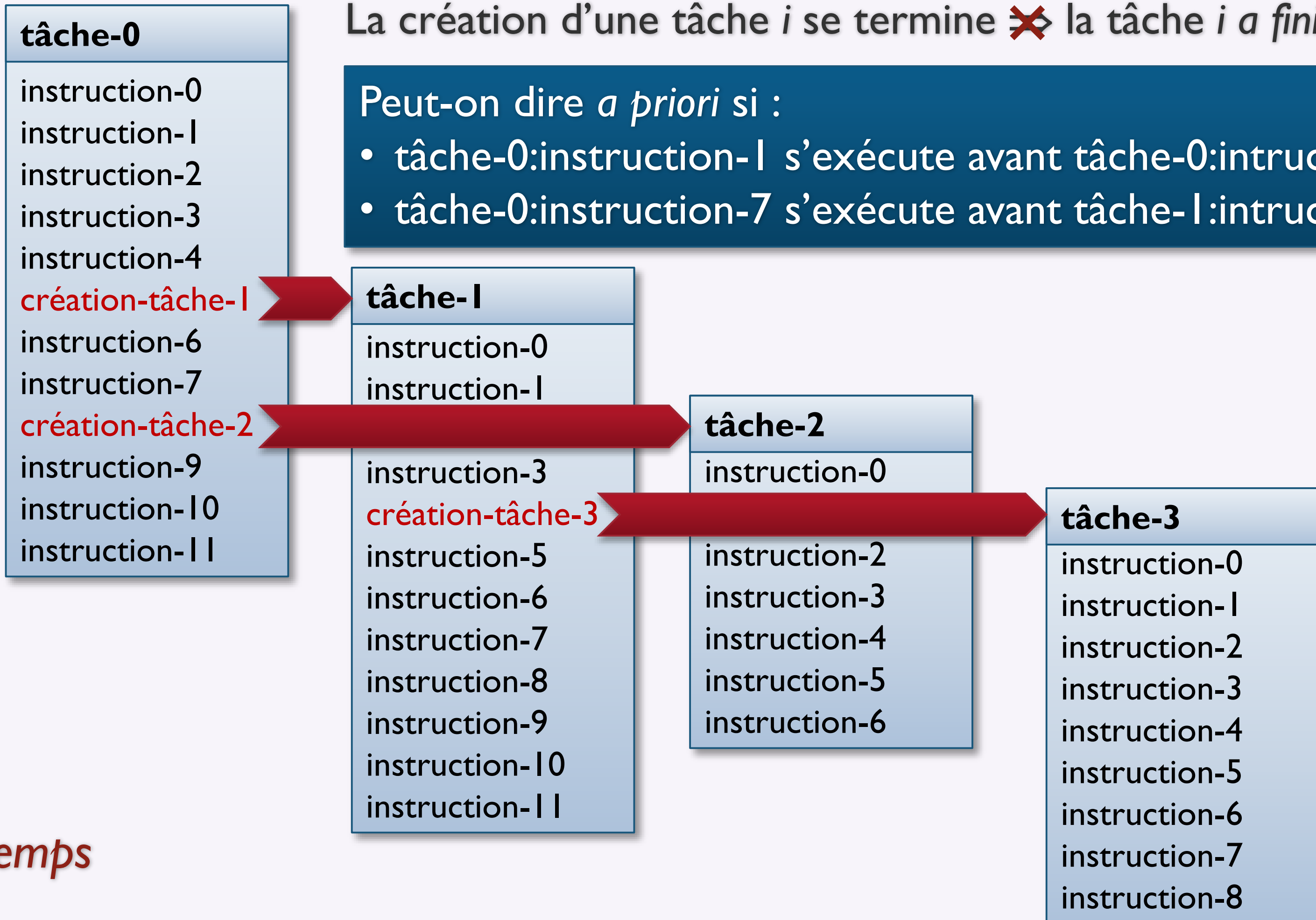
Caractéristiques d’une tâche

Exécution « en parallèle »

La création d’une tâche *i* se termine ⇏ la tâche *i a fini de s’exécuter*

Peut-on dire *a priori* si :

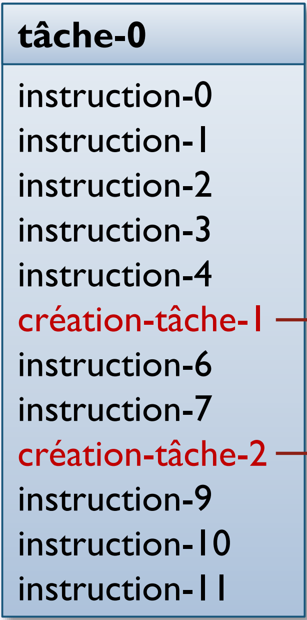
- tâche-0:instruction-1 s’exécute avant tâche-0:intruction-5 ? **oui**
- tâche-0:instruction-7 s’exécute avant tâche-1:intruction-6 ? **non**



© Shebli ANVAR, CEA Irfu

Posix
Création d'une tâche : pthread_create

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




```
pthread_t factorialThread_0, factorialThread_1;

int n = 5;
pthread_create(&factorialThread_0, NULL, factorial, &n);

n = 7;
pthread_create(&factorialThread_1, NULL, factorial, &n);
```

BUGGÉ !
Pourquoi ?

```
void* factorial(void* vn)
{
    int n = *((int*) vn);
    int r = n;
    while (--n > 1) r *= n;
    *((int*) vn) = r;
    return vn;
}
```

modification concurrente de la variable n

temps

Posix
Attributs d'une tâche : pthread_attr

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

tâche-0

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

```
pthread_t factorialThread_0, factorialThread_1;
pthread_attr_t attr;
pthread_attr_init(&attr); // Construction d'un attribut pthread

int n = 5;
pthread_create(&factorialThread_0, &attr, factorial, &n);

n = 7;
pthread_create(&factorialThread_1, &attr, factorial, &n);
pthread_attr_destroy(&attr); // Destruction d'un attribut pthread
```

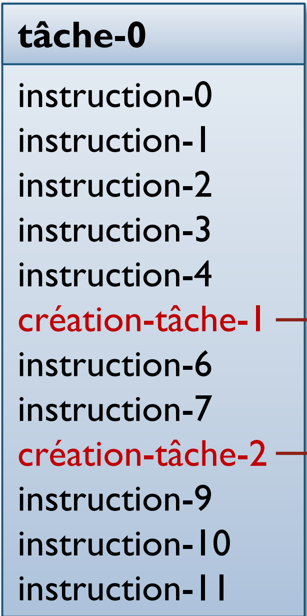
```
void* factorial(void* vn)
{
    int n = *((int*) vn);
    int r = n;
    while (--n > 1) r *= n;
    *((int*) vn) = r;
    return vn;
}
```

temps

Posix

Attributs d'une tâche : ordonnancement temps réel

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




```
pthread_t factorialThread_0, factorialThread_1;
pthread_attr_t attr;
pthread_attr_init(&attr); // Construction d'un attribut pthread
pthread_attr_setschedpolicy(&attr, SCHED_RR); // ou SCHED_FIFO
sched_param schedParams; // Paramètres d'ordonnancement
schedParams.sched_priority = 9;
pthread_attr_setschedparam(&attr, &schedParams);
int n = 5;
pthread_create(&factorialThread_0, &attr, factorial, &n);
schedParams.sched_priority = 23;
pthread_attr_setschedparam(&attr, &schedParams);
n = 7;
pthread_create(&factorialThread_1, &attr, factorial, &n);
pthread_attr_destroy(&attr); // Destruction d'un attribut pthread
```

```
void* factorial(void*
{
    int n = *((int*)
    int r = n;
    while (--n > 1) r
    *((int*) vn) = r;
    return vn;
}
```

OS Temps Réel (RTOS)
=
respect strict des priorités
+
latences garanties
(en particulier : context switch)

Posix
Accès concurrent à une ressource : pthread_mutex

Toute ressource partagée par plus d'un thread doit être associée à un mutex
Ressource partagée = variable partagée (de n'importe quel type)

```
pthread_t th[3];
pthread_attr_t attr;
pthread_attr_init(&attr); // Construction d'un attribut pthread
pthread_attr_setschedpolicy(&attr, SCHED_RR); // ou SCHED_FIFO
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED); // ordonnancement explicite
sched_param schedParams; // Paramètres d'ordonnancement
schedParams.sched_priority = 9;
pthread_attr_setschedparam(&attr, &schedParams);

double count = 0.0;

pthread_create(&th[0], &attr, countLoop, &count); // Accès non protégé à count
pthread_create(&th[1], &attr, countLoop, &count); // Accès non protégé à count
pthread_create(&th[2], &attr, countLoop, &count); // Accès non protégé à count
```

```
void* countLoop(void* vCount) {
    double* pCount = (double*) vCount;
    for (unsigned i=0u; i < 100000000u; ++i) {

        *pCount += 1.0;

    }
    return pCount;
}
```

Posix
Accès concurrent à une ressource : pthread_mutex

Toute ressource partagée par plus d'un thread doit être associée à un mutex
Ressource partagée = variable partagée (de n'importe quel type)

```
pthread_t th[3];
pthread_attr_t attr;
pthread_attr_init(&attr); // Construction d'un attribut pthread
pthread_attr_setschedpolicy(&attr, SCHED_RR); // ou SCHED_FIFO
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED); // ordonnancement explicite
sched_param schedParams; // Paramètres d'ordonnancement
schedParams.sched_priority = 9;
pthread_attr_setschedparam(&attr, &schedParams);

struct ProtCount {
    double count;
    pthread_mutex_t mutex;
} protCount;
protCount.count = 0.0;
pthread_mutex_init(&protCount.mutex, NULL);// Attention: prévoir pthread_mutex_destroy
pthread_create(&th[0], &attr, countLoop, &protCount); // Accès protégé à count
pthread_create(&th[1], &attr, countLoop, &protCount); // Accès protégé à count
pthread_create(&th[2], &attr, countLoop, &protCount); // Accès protégé à count
```

```
void* countLoop(void* vCount) {
    ProtCount* pCount = (ProtCount*) vCount;
    for (unsigned i=0u; i < 100000000u; ++i) {
        pthread_mutex_lock(&pCount->mutex); // Verrouillage du mutex (prise du jeton)
        pCount->count += 1.;
        pthread_mutex_unlock(&pCount->mutex); // Déverrouillage du mutex (rendu du jeton)
    }
    return pCount;
}
```

Posix
Accès concurrent à une ressource : pthread_mutex

Toute ressource partagée par plus d'un thread doit être associée à un mutex
Ressource partagée = variable partagée (de n'importe quel type)

Type de mutex	2e tentative de *lock* par une tâche détenant déjà le jeton	tentative de *unlock* 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

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 représente le temps absolu du processeur (depuis le boot)

```
timespec to;
clock_gettime(CLOCK_REALTIME, &to);
to.tv_sec  += 2;
to.tv_nsec += 500000000; //Timeout total de 2.5 secondes
if(pthread_mutex_timedlock(&mutex, &to) == ETIMEDOUT) {
    printf("timeout!\n");
}
```

Principaux types de tâche

- **Initiale**
  - activée au lancement de l'application
  - crée l'environnement : tâches et ressources
  - attends : fin des tâches ordinaires

```
int pthread_join (pthread_t th, void** retVal)
```
où retVal est l'adresse de la valeur de retour de la tâche

- **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
  - crée l'environnement : tâches et ressources
  - attends : fin des tâches ordinaires

int pthread_join (pthread_t *th*, void** *retVal*)
où retVal est l'adresse de la valeur de retour de la tâche

- **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 principale**

- **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
  - crée l'environnement : tâches et ressources
  - attends : fin des tâches ordinaires

int pthread_join (pthread_t *th*, void** *retVal*)
où retVal est l'adresse de la valeur de retour de la tâche

- **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 principale

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

Attente d'une tâche sur condition : pthread_cond

- **Principe**
  - Ressource *r* partagée par plusieurs tâches t_0, t_1, … t_{n-1}
    - Protégée par un mutex *m*
  - 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*)

Le code fourni (cette planche et suivante) est incomplet et contient des bugs : corrigez-les

```
struct Counter {
    Counter(int initVal = 0);
    int value; // Cette déclaration est-elle correcte ?
    pthread_mutex_t mutex; // Protection multitâche de la ressource Counter
    pthread_cond_t isEmpty; // Condition sur la ressource Counter
};

int main() {
    pthread_t th[3];
    Counter counter(1000); // Attention : prévoir initialisations et désallocations. Quoi ? Où ?
    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], 0);
}
```

`volatile int value;`

mutex et condition

constructeur

Attente d'une tâche sur condition : pthread_cond

- **Principe**
  - Ressource *r* partagée par plusieurs tâches t_0, t_1, … t_{n-1}
    - Protégée par un mutex *m*
  - Condition C(*r*) sur la ressource
    - Tant que C(*r*) n'est pas réalisée, t_0 reste dans l'état bloqué
    - Lorsqu'une tâche $t_{k,\ k>0}$ modifie *r* permettant à C(r) d'être réalisée, t_k notifie t_0 et t_0 teste C(*r*)

Peut sortir du wait sans que C(r) soit réalisée
Plus d'une tâche peuvent attendre C(r)

```
void* monitor(void* vCounter) {
    Counter* pCounter = (Counter*) vCounter;
    pthread_mutex_lock(&pCounter->mutex);
    while (counter->value > 0) // Pourquoi pas « if » ?
        pthread_cond_wait(&pCounter->isEmpty, &pCounter->mutex);
    std::cout << "Counter is empty! Bye!" << std::endl;
    return pCounter; // On n'a pas oublié quelque chose ?
}
```

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

```
void* consumer(void* vCounter) {
    Counter* pCounter = (Counter*) vCounter;
    while (true) {
        pthread_mutex_lock(&pCounter->mutex);
        pCounter->value -= 1; // Décrémentation
        pthread_cond_signal(&pCounter->isEmpty); // Signale une modification
        if (pCounter->value < 0) break; // Est-ce que cela pose un problème ?
        pthread_mutex_unlock(&pCounter->mutex);
    }
    return pCounter;
}
```

dans ce cas (break), le mutex n'est pas libéré

Attente sur condition avec timeout
Notification à toutes les tâches en attente

```
void* monitor(void* vCnt)
{
    Counter* pCnt = (Counter*) vCnt;
    timespec abstime;
    clock_gettime(CLOCK_REALTIME, &abstime); // Obtention du temps présent absolu
    abstime.tv_nsec += 100000000; // Timeout de 100 millisecondes
    abstime.tv_sec  += abstime.tv_nsec / 1000000000; // Renormalisation des secondes
    abstime.tv_nsec  = abstime.tv_nsec % 1000000000; // Renormalisation des nanosecondes
    pthread_mutex_lock(&pCnt->mutex);
    int status = 0;
    while (counter->value > 0)
    {
        status = pthread_cond_timedwait(&pCnt->isEmpty, &pCnt->mutex, &abstime);
        if (status == ETIMEDOUT) throw "Timeout waiting for condition!";
    }
    pthread_mutex_unlock(&pCnt->mutex);
    std::cout << "Counter is empty! Bye!" << std::endl;
    return pCnt;
}
```

```
void* consumer(void* vCnt) {
    Counter* pCnt = (Counter*) vCnt;
    while (true) {
        pthread_mutex_lock(&pCnt->mutex);
        pCnt->value -= 1; // Décrémentation
        pthread_cond_broadcast(&pCnt->isEmpty); // Signale à toutes les tâches en attente
        if (pCnt->value < 0) break;
    }
    pthread_mutex_unlock(&pCnt->mutex);
    return pCnt;
}
```

Alarme (timer)
Compte à rebours unique ou périodique

- **Timers POSIX**
  - émission d'un signal à l'issue d'un compte à rebours
  - appel d'une fonction « handler » réagissant au signal

```
void myHandler(int sig, siginfo_t* si, void*){/* mon code; mes données dans si->si_value*/}
```

```
struct sigaction sa; // action à effectuer lorsque survient le signal de fin de compte à rebours (CÀR)
sa.sa_flags = SA_SIGINFO; // l'action est un appel de fonction avec paramètres
sa.sa_sigaction = myHandler; // myHandler est la fonction à appeler
sigemptyset(&sa.sa_mask); // flags de blocage à 0 : aucun signal n'est bloqué pendant le traitement du signal
sigaction(SIGRTMIN, &sa, NULL); // installation de l'action associée au signal
```

```
sigevent_t sev; // événement associé à l'expiration du CÀR
sev.sigev_notify = SIGEV_SIGNAL; // l'événement sera de type « signal »
sev.sigev_signo = SIGRTMIN; // il s'agit d'un signal temps réel : entre SIGRTMIN et SIGRTMAX
sev.sigev_value = myData; // de type union sigval { int sival_int; void* sival_ptr; }
timer_t tid; // variable contenant l'identifiant du timer
timer_create(CLOCK_REALTIME, &sev, &tid); // création du timer: prévoir destruction (timer_delete)
itimerspec its; // structure contenant la (les) période(s) de CÀR du timer
its.it_value.tv_sec  = 10; // CÀR de 10 secondes
its.it_value.tv_nsec = 0; // pour recommencer le CÀR périodiquement, renseigner aussi its.it_interval
timer_settime(tid, 0, &its, NULL); // armement du timer (démarrage du CÀR)
// … autres instructions en attendant arrivée du signal
```

- **Fonctionnement périodique**
  - Réarmement périodique du timer
    - Relance du timer à l'intérieur du handler
    - Configuration du champ it_interval du timer

Tâche périodique
Mise en œuvre par alarme périodique et condition