

Sommaire

Les processus	2
Séquence n°1 : l'appel fork	2
Séquence n°2 : généalogie de processus	3
Séquence n°3 : les appels exec	3
Séquence n°4 : synchronisation des terminaisons	4
Séquence n°5 : les processus sous Windows	4
Les threads	7
Définition	7
Manuel du pogrammeur	8
Fonctionnalités des threads	8
Séquence n°1 : mise en oeuvre des threads sous Linux	9
Objectifs	9
Étape n°0 : création de deux fonctions	9
Étape n°1 : création de deux <i>threads</i>	10
Étape n°2 : compilation et édition de liens	11
Étape n°3 : exécution	12
Séquence n°2 : mise en oeuvre des threads sous Windows	13
Les threads dans différents environnements et langages	14

Les objectifs de ce tp sont de découvrir la programmation multitâche à base de processus et de *threads* dans différents systèmes d'exploitation.

Les processus

Séquence n°1 : l'appel fork

Question 1. Expliquer, si possible en utilisant un schéma, l'utilisation suivante de la primitive `fork` :

```
if (fork())
    printf("Je suis le père\n");
else
    printf("Je suis le fils\n");
```

processus.1a.c

Pour comprendre ce bout de code, vous devez consulter le manuel du programmeur :

```
$ man fork
```

```
...
```

En cas de succès, le PID du fils est renvoyé au processus parent, et 0 est renvoyé au processus fils. En cas d'échec -1 est renvoyé dans le contexte du parent, aucun processus fils n'est créé, et `errno` contient le code d'erreur.



Attention, l'affichage obtenu à l'exécution est le suivant :

```
$ ./a.out
```

```
Je suis le père
```

```
Je suis le fils
```

Question 2. Compléter l'affichage dans le programme suivant :

```
int main()
{
    int pid;
    pid = fork();
    if (pid == -1)
    {
        perror("fork"); exit(-1);
    }
    if (pid == 0) printf("Je suis le ...\n"); // père ou fils ?
    if (pid != 0) printf("Je suis le ...\n"); // père ou fils ?
}
```

processus.1b.c

Question 3. Dans le code source `processus.1c.c` fourni, améliorer le programme en affichant les PID/PPID de chaque processus (voir les appels `getpid()` et `getppid()`) avec les messages suivants : "Je suis le père, mon PID est x et j'ai créé un fils dont le PID est x" et "Je suis le fils, mon PID est x et mon père a le PID x". Donner l'affichage obtenu.

Séquence n°2 : généalogie de processus

Question 4. Quel affichage produit le code suivant ? Combien de processus sont créés ? Justifier (un schéma peut être utile;).

```
setbuf(stdout, NULL); // pour éviter la bufferisation sur l'affichage
printf("+");
fork();
printf("+"); // pour vous aider vous pouvez changer de symbole
fork();
printf("+"); // pour vous aider vous pouvez changer de symbole
```

processus.2a.c



Le `\n` force l'affichage sinon c'est bufférisé !

Séquence n°3 : les appels exec

À partir d'un programme en C, il est possible d'exécuter des processus de plusieurs manières avec soit l'appel `system` soit les appels `exec` (cf. `man exec`).

```
// Exemple 1 : l'appel system()
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Résultat de la commande ps fl :\n");
    system("ps fl");
    printf("Fin\n");
    return 0;
}
```

processus.3a.c

```
// Exemple 2 : l'appel execlp()
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("Résultat de la commande ps fl :\n");
    execlp("ps", "ps", "fl", NULL); // cf. man execlp
    printf("Fin\n");
}
```

processus.3b.c

Question 5. Quel est le rôle de la commande `ps` ?

Question 6. Comparer au niveau des PID/PPID l'exécution des deux exemples ci-dessus. Conclure.



Vous pouvez utiliser la commande `pstree` !

Question 7. Dans l'exemple 2, le message "Fin\n" ne s'affiche pas. Pourquoi ?

Question 8. Compléter le programme suivant pour qu'il lance une calculatrice (`kcalc` ou `gcalctool`) et un éditeur de texte (`kwrite` ou `gedit`) en utilisant `exec` :

```
if (! fork()) { printf("Fils 1 : je lance une calculatrice !\n"); ... }
if (! fork()) { printf("Fils 2 : je lance un éditeur de texte !\n"); ... }
```

processus.3b.c

Séquence n°4 : synchronisation des terminaisons

Question 9. Qui de ces 2 processus se termine le premier, le père ou le fils ? Essayez plusieurs fois.

```
if (fork())
{
    printf("Je suis le père\n");
    printf("Fin du père\n");
}
else
{
    printf("Je suis le fils\n");
    printf("Fin du fils\n");
}
```

processus.4a.c

Question 10. En utilisant la primitive `wait`, modifier le programme pour que le père se termine toujours après son fils.

Séquence n°5 : les processus sous Windows

Sous Windows, il y avait principalement trois options pour écrire des programmes multi-tâches :

- Utiliser la *Win32 Application Programming Interface* (API) en conjonction avec la *C run-time library* (CRT). Comme son nom l'indique, l'API est implémentée en C et elle intègre les fonctions d'appels systèmes de plus bas niveau accessibles au développeur d'applications windows actuel.
- Utiliser la *Microsoft Foundation Class* (MFC) en conjonction avec le C++. La MFC a été développée comme un ensemble de classes C++ qui agissent en tant que « *wrappers* » de l'API Win32, afin d'en faciliter l'utilisation et l'intégration dans le cadre d'un développement orienté-objet.
- Utiliser la plateforme .NET qui offre plusieurs options aussi pour décomposer une application en processus légers. Le « *namespace* » `System.Threading` en est une.



Évidemment, d'autres *framework* sont disponibles. On peut citer : Qt, CLX/VCL de Builder Borland, ...

L'appel `CreateProcess` de la Win32 permet la création et l'exécution d'un processus enfant.



Si on veut comparer avec Unix, l'appel `CreateProcess()` est l'équivalent des appels `fork()` et `exec()` regroupés.

Voici un exemple pour créer un nouveau processus sous Windows (msdn.microsoft.com) :

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

int main( VOID )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( ".\\Child.exe", //NULL then use command line
        NULL, //TEXT("Child"), // Command line.
        NULL, // Process handle not inheritable.
        NULL, // Thread handle not inheritable.
        FALSE, // Set handle inheritance to FALSE.
        0, // No creation flags.
        NULL, // Use parent's environment block.
        NULL, // Use parent's starting directory.
        &si, // Pointer to STARTUPINFO structure.
        &pi ) // Pointer to PROCESS_INFORMATION structure.
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

Exemple d'utilisation de CreateProcess

Le code source du processus père :

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

int main( VOID )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    DWORD status ; // address to receive termination status

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( !CreateProcess( ".\\Child.exe", NULL, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
```

```

{   printf( "CreateProcess failed (%d).\n", GetLastError() );
    ExitProcess(1);
}
WaitForSingleObject( pi.hProcess, INFINITE );
if(GetExitCodeProcess( pi.hProcess, &status))
    printf("I am the father. My child [%d,%d] has terminated with (%d). \n", pi.
        dwProcessId, pi.dwThreadId, status );
else printf( "GetExitCodeProcess failed (%d).\n", GetLastError() );

CloseHandle( pi.hProcess );
CloseHandle( pi.hThread );
printf( "I am the father. Going to terminate.\n");
ExitProcess(0);
}

```

father.c

Le code source du processus fils (Child.exe) :

```

#include <windows.h>
#include <stdio.h>
#include <tchar.h>
int main()
{
    printf("I am the child. My Pid and Tid: (%d, %d).\n", GetCurrentProcessId(),
        GetCurrentThreadId());
    printf( "I am the child. Going to sleep for 5 seconds. \n");
    Sleep(5000);
    ExitProcess(10); // exit code for all threads
}

```

Child.c



Sous Windows, tout processus dispose d'au moins un thread, de manière à exécuter le point d'entrée du programme. Ce thread ne pourra jamais être terminé sans que l'application ne soit elle-même terminée. Le processus ne doit en aucun cas être assimilé à ce thread. Le thread constitue simplement l'unité d'exécution de base du processus. Le système ne communiquera jamais avec le processus mais toujours avec l'un des threads de ce processus. En effet, le processus n'étant pas une unité d'exécution il ne réalise aucune tâche. Le premier thread est créé par le système. La création de nouveaux threads devra être explicite par l'appel `CreateThread`. Un processus peut donc posséder plusieurs threads.

Question 11. Compiler (avec Dev-Cpp ou MinGW) et tester. Vérifier les valeurs affichées en utilisant le gestionnaire de tâches de Windows (`taskmgr.exe`)

Les threads

Définition

Il existe plusieurs traductions du terme *thread* :

- **fil** d'exécution,
- activité, tâche
- *lightweight process (lwp)* ou processus léger

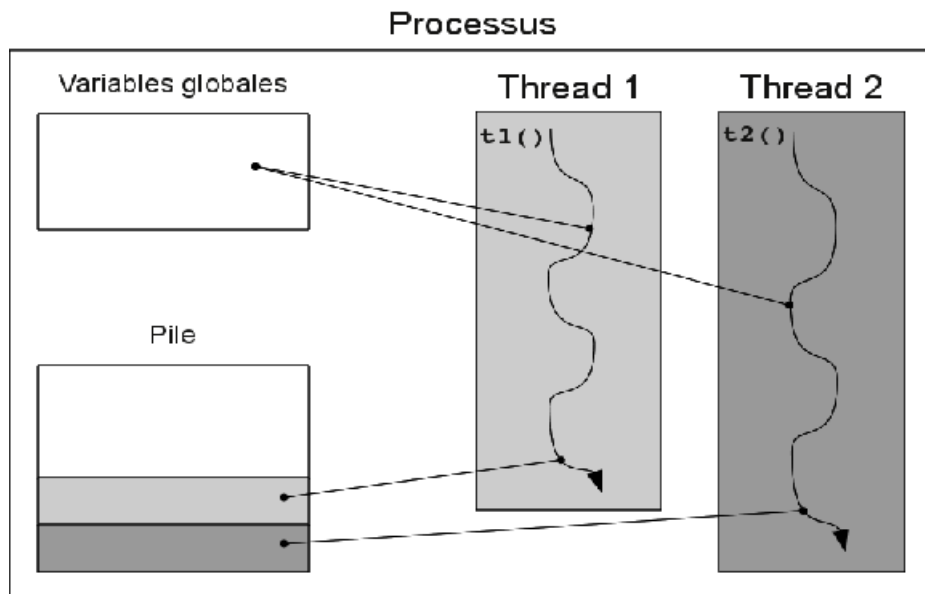


Par opposition, un processus créé par `fork()` sera qualifié de processus lourd.

Les *threads* permettent de dérouler plusieurs suites d'instructions, en **PARALLELE**, à l'intérieur d'un même processus. Un *thread* exécutera donc une **fonction**.



Besoin : on aura besoin de thread(s) dans une application lorsqu'on devra paralléliser des traitements.



Donc, un *thread* :

- est englobé par un processus
- dispose de sa propre pile pour implanter ses variables locales,
- partage les données globales avec les autres threads.



Il ne faut pas confondre la technologie *Hyperthreading* incluse dans certains processeurs Intel avec les *threads*. Cette technologie permet en effet aussi bien l'exécution simultanée de processus lourds que de *threads*.

Manuel du programmeur

Les systèmes d'exploitation mettent en oeuvre généralement les *threads*. C'est le cas des systèmes Unix/Linux et Microsoft ©Windows.

La plupart des langages (Java sur la plupart des systèmes d'exploitation, C++, C# .NET, ...) utilisent des extensions du langage ou des bibliothèques pour utiliser directement les services de *multithreading* du système d'exploitation.



En 1995, un standard d'interface pour l'utilisation des processus légers a vu le jour, ce standard est connu sous le nom de *pthread* (ou POSIX thread 1003.1c-1995).

Le développeur utilisera donc concrètement une interface pour programmer une application multi-tâche grâce par exemple :

- au **standard POSIX pthread** largement mis en oeuvre sur les systèmes UNIX/Linux
- à l'API **WIN32 threads** fournie par Microsoft ©Windows pour les processus légers

Les pages man sous Unix/Linux décrivent les appels de l'API **pthread** :

- `pthread_create(3)` : crée (et exécute) un nouveau *thread*
- ...
- `pthread(7)` : standard POSIX **pthread**



L'accès aux pages man se fera donc avec la commande `man`, par exemple : `man 3 pthread_create`

L'utilisation des *threads* POSIX est tout à fait possible en C++. Toutefois, il sera conseillé d'utiliser un *framework* C++ (Qt, Builder, commoncpp, boost, ACE, ...).

Fonctionnalités des threads

Les fonctionnalités suivantes sont présentes dans la norme **POSIX thread 1003.1c-1995** :

- gestion des processus légers (*thread management*) : initialisation, création, destruction ... et l'annulation (*cancellation*) ;
- gestion de la synchronisation : exclusion mutuelle (*mutex*), variables conditions ;
- données privées (*thread-specific data*) ;
- ordonnancement (*thread priority scheduling*) : gestion des priorités, ordonnancement préemptif ;
- signaux : traitant des signaux (*signal handler*), attente asynchrone, masquage des signaux, saut dans un programme (*long jumps*) ;

Les processus légers WIN32 sont implantés au niveau du noyau. L'unité d'exécution finale est le processus léger. L'ordonnancement est réalisé selon l'algorithme du tourniquet (*round-robin scheduling*). Tous les processus légers accèdent aux mêmes ressources du processus lourd, en particulier à son espace mémoire. Un processus lourd accède à une espace mémoire de 4 Gigaoctets (Go), découpé en 2 Go pour l'utilisateur et 2 Go pour le système d'exploitation. Chaque processus léger peut accéder à cette zone de 2 Go. Un processus léger d'un processus lourd ne peut pas accéder aux ressources d'un autre processus lourd.

Les fonctionnalités suivantes sont fournies avec les processus légers **WIN 32 threads** :

- gestion des processus légers : création, terminaison, priorités, suspension.
- gestion de la synchronisation : sections critiques, mutex, sémaphores, événements.

- communication : une file de message associée à chaque processus léger permet d'envoyer des messages à des processus légers du système.

La synchronisation sur un objet noyau est réalisée en utilisant une fonction commune (`WaitForSingleObject`). Une autre fonction permet également l'attente multiple sur une synchronisation (`WaitForMultipleObject`).

Séquence n°1 : mise en oeuvre des threads sous Linux

Objectifs

L'objectif de cette séquence est la mise en oeuvre simple d'une application à base de *threads*.

Étape n°0 : création de deux fonctions

On crée deux fonctions : une affichera des étoiles '*' et l'autre des dièses '#'.

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

// Fonctions :
void etoile(void);
void diese(void);

int main(void)
{
    setbuf(stdout, NULL); // pas de tampon sur stdout
    printf("Je vais lancer les 2 fonctions\n");
    etoile();
    diese();

    return EXIT_SUCCESS;
}

void etoile(void)
{
    int i;
    char c1 = '*';

    for(i=1;i<=200;i++)
    {
        write(1, &c1, 1); // écrit un caractère sur stdout (descripteur 1)
    }
    return;
}

void diese(void)
{
    int i;
    char c1 = '#';
```

```
for(i=1;i<=200;i++)
{
    write(1, &c1, 1);
}
return;
}
```

threads.0.c

L'exécution de ce programme montre la **séquentialité des traitements effectués** : tout d'abord la fonction `etoile()` s'exécute entièrement puis la fonction `diese()` est lancée et s'exécute à son tour.

```
$ ./threads.0
```

Je vais lancer les 2 fonctions

```
*****
*****
*****
*****#####
#####
#####
#####
#####
$
```

Cette application n'est donc pas multi-tâche car les traitements ne se sont pas effectués en parallèle mais l'un après l'autre.

Étape n°1 : création de deux *threads*

On crée maintenant deux tâches (*threads*) pour exécuter chacune des deux fonctions : une affichera des étoiles '*' et l'autre des dièses '#'.

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

// Fonctions correspondant au corps d'un thread (tache)
void *etoile(void *inutilise);
void *diese(void *inutilise);
// Remarque : le prototype d'une tâche doit être : void *(*start_routine)(void *)

int main(void)
{
    pthread_t thrEtoile, thrDiese; // les ID des de 2 threads

    setbuf(stdout, NULL); // pas de tampon sur stdout

    printf("Je vais creer et lancer 2 threads\n");
    pthread_create(&thrEtoile, NULL, etoile, NULL);
    pthread_create(&thrDiese, NULL, diese, NULL);

    //printf("J'attends la fin des 2 threads\n");
    pthread_join(thrEtoile, NULL);
    pthread_join(thrDiese, NULL);
    printf("\nLes 2 threads se sont termines\n");
}
```

```
printf("Fin du thread principal\n");
pthread_exit(NULL);

return EXIT_SUCCESS;
}

void *etoile(void *inutilise)
{
    int i;
    char c1 = '*';

    for(i=1;i<=200;i++)
    {
        write(1, &c1, 1); // écrit un caractère sur stdout (descripteur 1)
    }

    return NULL;
}

void *diese(void *inutilise)
{
    int i;
    char c1 = '#';

    for(i=1;i<=200;i++)
    {
        write(1, &c1, 1);
    }

    return NULL;
}
```

threads.1a.c



Ici le paramètre `inutilise` n'est pas utilisé mais requis pour respecter le prototype de `pthread_create()`.

Étape n°2 : compilation et édition de liens

Il faut compiler et faire l'édition des liens avec l'option `-pthread`.

```
$ gcc -o threads.1a threads.1a.c -D_REENTRANT -pthread
$
```


Séquence n°2 : mise en oeuvre des threads sous Windows

Sous Windows, la création d'un thread est réalisé par l'appel `CreateThread` de la Win32 :

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

DWORD WINAPI ThreadEtoile(LPVOID lpParam)
{
    int n;
    int *pData;

    pData = (int*)lpParam;
    n = *pData;
    printf("la tache%d affiche des * ...\n", n);

    int i;
    for(i=1;i<=200;i++)
    {
        printf("*");
    }

    return 0;
}

DWORD WINAPI ThreadDiese(LPVOID lpParam)
{
    int n;
    int *pData;

    pData = (int*)lpParam;
    n = *pData;
    printf("la tache%d affiche des # ...\n", n);

    int i;
    for(i=1;i<=200;i++)
    {
        printf("#");
    }

    return 0;
}

int main(void)
{
    DWORD dwThreadId1, dwThreadId2;
    HANDLE hThread1, hThread2;
    int n1 = 1, n2 = 2;

    // Création du thread (autre que le primaire)
    hThread1 = CreateThread(NULL, // default security attributes
```

```
        0, // use default stack size
        ThreadEtoile, // thread function
        &n1, // argument to thread function
        0, // use default creation flags
        &dwThreadId1); // returns the thread identifier

if (hThread1 == NULL)
{
    printf( "CreateThread failed (%d).\n", GetLastError());
    ExitProcess(1);
}

// Création du thread (autre que le primaire)
hThread2 = CreateThread(NULL, // default security attributes
        0, // use default stack size
        ThreadDiese, // thread function
        &n2, // argument to thread function
        0, // use default creation flags
        &dwThreadId2); // returns the thread identifier

if (hThread2 == NULL)
{
    printf( "CreateThread failed (%d).\n", GetLastError());
    ExitProcess(1);
}

// Attente fin des threads
WaitForSingleObject(hThread1, INFINITE);
WaitForSingleObject(hThread2, INFINITE);

CloseHandle(hThread1);
CloseHandle(hThread2);

printf("Fin ... \n");

return 0;
}
```

threads.2a.c

Question 12. Compiler (avec Dev-Cpp ou MinGW) et tester.

Les threads dans différents environnements et langages

La mise en oeuvre d'une application multi-tâche avec différents langages (C++ et Java) et/ou environnements de développement (*framework* Qt et Builder, API `commoncpp` et `boost`) est tout à fait possible et simplifie le travail du développeur :

- Sous **Qt**, on dérive une classe `QThread` et on écrit le code du *thread* dans la méthode `run()`. Ensuite, on appellera la méthode `start()` pour démarrer le *thread* et la méthode `stop()` pour l'arrêter.
- Sous **Builder**, on dérive une classe `TThread` et on écrit le code du *thread* dans la méthode `Execute()`. Ensuite, on appellera la méthode `Resume()` pour démarrer le *thread* et la méthode `Terminate()` pour l'arrêter.

- Avec l'API `commoncpp` (C++) : on dérive une classe `Thread` et on écrit le code du *thread* dans la méthode `run()`. Ensuite, on appellera la méthode `start()` pour démarrer le *thread* et la méthode `stop()` pour l'arrêter.
- Avec l'API `boost` (C++), on crée simplement un objet de type `thread` en passant au constructeur l'adresse d'une fonction.
- En Java : cela revient à dériver (`extends` en Java) une classe de base fournie par l'API (`Thread`) et à écrire le code du *thread* dans une méthode spécifique (`run()`). Ensuite, on appellera la méthode `start()` pour démarrer le *thread* et la méthode `stop()` pour l'arrêter. Il est aussi possible d'utiliser l'interface `Runnable` et instancier un objet `Thread` avec.



Le support des threads est basé sur des API écrites en C. Le portage en C++ pose le problème de passer l'adresse d'une méthode (représentant le code du *thread*) comme on le fait en C pour une fonction. Ce n'est pas possible car les méthodes d'une classe n'ont pas d'adresse au moment de l'édition de liens. La solution consiste à déclarer la méthode comme **statique**. Cette méthode `static` sera donc disponible en dehors de toute instance de la classe et pourra donc être passée en argument de `pthread_create()` par exemple. Par contre elle ne pourra pas accéder aux attributs de sa classe en dehors des autres membres statiques. Une solution est décrite ci-dessous.

En C++, le principe est de créer une classe **abstraite** `Thread` qui contiendra :

- une méthode virtuelle pure `run()` dans laquelle on placera le code du *thread* une fois qu'on aura dérivé la classe mère. Cette méthode pourra accéder aux attributs et méthodes de sa classe.
- une méthode `execute()` qui exécutera `run()`
- une méthode `start()` qui créera et lancera le *thread* (appel à `pthread_create()`) en lui passant l'adresse de la méthode statique `execute()`

