UNIVERSITÉ DE
SHERBROOKE

Département d'informatique
Faculté des sciences

IFT 630 - Processus concurrents et parallélisme

Chapitre 2

Concurrence

Gabriel Girard[1]

Sherbrooke

10 novembre 2022

[1] Gabriel.Girard@usherooke.ca

Table des matières

Chapitre 2

Concurrence et Parallélisme

Ce chapitre est inspiré de [57, 68, 69].

Qu'est-ce que la concurrence ou le parallélisme ?

La concurrence ou parallélisme au niveau des processus se définit de la façon suivante : *deux processus sont concurrents s'ils existent (s'exécutent) en même temps*

Attention

Pour ce cours, nous ne faisons pas de distinction entre parallélisme et concurrence, car les deux impliquent des solutions relativement semblables. Toutefois, il est important de savoir que dans la littérature, ces deux termes ont des significations différentes, la concurrence étant un cas plus général que celui du parallélisme. Pour des définitions plus complètes qui permettent de distinguer les deux termes, veuillez consulter l'annexe A.

Si deux processus sont concurrents et que les variables de chacun des processus sont inaccessibles aux autres processus, ils sont dits **asynchrones**. Dans ce cas, on peut montrer qu'il n'y aura aucun problème lors de l'exécution, i.e. que la sortie finale devient une fonction appliquée sur l'entrée qui est indépendante du temps (vitesse d'exécution). Ainsi si e représente l'entrée d'une fonction et f la fonction elle-même, alors la sortie s est calculée par :

$$s = f(e)$$

Si, au contraire, les processus sont **synchrones**, i.e. qu'un processus peut changer les variables d'un ou de plusieurs autres processus, alors la sortie d'un processus devient une fonction qui dépend de la vitesse relative de l'exécution des autres processus. Ainsi si e représente l'entrée d'une fonction et f la fonction elle-même, alors la sortie s est calculée par :

$$s = f(e, t)$$

où t représente dans ce cas la notion de temps ou de vitesse d'exécution de la fonction f. Les figures 2.1 et 2.2 illustrent des cas d'exécution dont le résultat dépend du temps.

Que la sortie d'un fonction dépende de la vitesse d'exécution n'est généralement pas un comportement souhaitable. Toutefois, ce problème survient sur des systèmes multi-programmés où le

Exemple 1

Le compte en banque

Soit votre compte en banque A contenant initialement 500$.
Vous décidez de retirer 10$. Au même moment et dans le même compte, le gouvernement fait le dépôt automatique de votre bourse d'études de 1000$. Les deux transactions s'exécutent alors en parallèle. Soient :

- T1 : retrait de 10$;
- T2 : ajout de 1000$.

Toutes les transactions peuvent se subdiviser en plusieurs instructions machines. Ainsi pour chaque transaction, le détail des opérations à effectuer est :

- `T1 : (1) Load ; (2) sub 10;  (3) Store;`
- `T2 : (1) Load ; (2) Add 1000; (3) Store;`

La séquence d'exécution suivante des instructions peut alors se produire :

1. `T2.1 : Load` → 500$
2. `T2.2 : Sub 10` → 490$

   —
3. `T1.1 : Load` → 500$
4. `T1.2 : Add 1000` → 1500$
5. `T1.3 : Store 1500` → 1500$

   —
6. `T2.3 : Store 490` → 490$

À la fin de cette exécution, il restera dans le compte 490$ plutôt que 1490$.

Figure 2.1 – Exemple d'exécution synchrone

comportement d'un processus peut être influencé par les autres processus. Dans cette situation, **les processus doivent se synchroniser et coopérer**.

Définition : Condition de course

Une situation où plusieurs processus accèdent simultanément à la même ressource et dont le résultat final dépend de l'ordre d'exécution s'appelle une **condition de course**.

2.1 Types de concurrence

2.1.1 Parallélisme implicite vs explicite

Le parallélisme existe que nous en soyons conscient ou non. Le parallélisme implicite, lui, se fait toujours à notre insu. Le parallélisme au niveau matériel, abordé au chapitre précédent, est une forme de parallélisme implicite. Le parallélisme explicite, quant à lui, exige de l'utilisateur un

Exemple 2

Soit les deux programmes suivants :

Pgm 1	Pgm 2
(1.1) : R = 1	(2.1) : T = 1
(1.2) : X = T	(2.2) : Y = R

Le programme 1 initialise une variable R puis copie T dans sa variable X. De même, le programme 2 initialise la variable T puis copie R dans T. On suppose qu'initialement les variables R et T contiennent la valeur 0.
La sortie des deux programmes, soient les valeurs (X,Y), dépend en grande partie de la vitesse et de l'ordre d'exécution des instructions. Les sorties possibles selon la séquence d'exécution sont :

- X=0, Y=1 → séquence = 1.1 ; 1.2 ; 2.1 ; 2.2
- X=1, Y=0 → séquence = 2.1 ; 2.2 ; 1.1 ; 1.2
- X=1, Y=1 → séquence = 1.1 ; 2.1 ; 1.2 ; 2.2

Est-ce que la sortie X=0, Y=0 est possible ?

Oui, elle est possible. En fait, cette sortie est une conséquence de l'une des trois causes suivantes : le parallélisme au niveau matétiel (superscalaire), une incohérence des données au niveau de la cache ou un réordonnancement, pour des raisons d'optimisation, des instructions d'un même programme par le compilateur ou le processeur (des instructions qui, dans ce cas-ci, sont techniquement indépendantes).

Figure 2.2 – Exemple d'exécution synchrone

apprentissage adapté au niveau de complexité associé à ce type de programmation.

Le parallélisme implicite

Ce type de concurrence est celui implanté dans les ordinateurs, dans les systèmes multiprogrammés et dans certains environnements parallélisant (tels les compilateurs parallélisant). Dans cette situation, les processus ne sont généralement même pas conscients de la synchronisation et ignorent même qu'ils doivent être synchronisés avec d'autres processus (s'il y a synchronisation). Tous ces processus sont tout de même en concurrence pour l'accès à certaines ressources fournies par le système d'exploitation ou par l'ordinateur. Unix, Windows, Linux sont des exemples d'environnements dans lesquels la concurrence implicite est présente.

La concurrence implicite elle-même se subdivise en deux sous-types :

1. le parallélisme indépendant qui se caractérise par :
   - l'utilisation d'applications séparées ;
   - un besoin inexistant de synchronisation ;
   - une performance très élevée sur des machines indépendantes ou des multiprocesseurs ;

- aucun besoin de modifier les applications.

Comme il n'y a pas de synchronisation, le concept d'intervalle entre les besoins de se synchroniser ne s'applique pas.

2. le parallélisme de granularité très grossière et grossière qui se caractérise par :
   - un besoin de synchronisation très grossier entre les processus ;
   - une accélération moindre que le parallélisme indépendant mais encore très élevée ;
   - une bonne performance sur les multiprocesseurs et une performance limitée sur les systèmes distribués (ordinateurs multiples) directement liées aux besoins en communication ;
   - une quantité minimale ou nulle de modifications à apporter aux applications.

   Le parallélisme de granularité très grossière nécessite un synchronisation à des intervalles variants de 2000 à un million d'instructions.

Le parallélisme explicite

La concurrence explicite se produit généralement au niveau d'un même programme. On parle alors de programmation parallèle ou de traitement parallèle. Elle exige un langage ou des outils permettant de spécifier explicitement la concurrence. Des exemples d'environnements pour la concurrence explicite : les langages SR et JR, les bibliothèques de calcul parallèle (MPI, OpenCL, Cuda, ...) et multi-fils, et le support multi-fils dans certains langages (C++, Java, C#, Python, ...).

La concurrence explicite se subdivise aussi en deux catégories :

1. le parallélisme de granularité moyenne caractérisé par :
   - des applications comprenant plusieurs fils d'exécution ;
   - une spécification explicite du parallélisme et de la synchronisation par la personne qui développe l'application ;
   - un haut degré de communication et de synchronisation ;
   - la nécessité de repenser la planification des processus à cause des fréquentes interactions.

   Le parallélisme de granularité moyenne exige une synchronisation à des intervalles variants entre 20 et 200 instructions.
2. le parallélisme de granularité fine caractérisé par :
   - une programmation plus complexe que la simple utilisation de fils d'exécution ;
   - une intervention plus importante du matériel ;
   - plusieurs approches telles que le parallélisme par les données ou le parallélisme au niveau des instructions ;

   La parallélisme par les données est de type SIMD et se retrouve dans des cartes graphiques ou des processeurs vectoriels, dans la parallélisation automatique de boucles fournies par certains environnements ou compilateurs, et dans les ordinateurs à flux de données. Ces derniers (proposés dans les années 60 et oubliés dans les années 80), remplacent le contrôle de l'exécution des instructions, fait normalement par l'horloge, par le flux des données [85, 88].
   Nous avons déjà abordé le parallélisme au niveau matériel avec le pipeline, le VLIW et le SMT (multi-fils). Dans certains cas, c'est la personne en charge du développement ou le compilateur qui détermine ce qui doit se faire en parallèle.
   La parallélisme de granularité fine nécessite une synchronisation à des intervalles inférieures à 20 instructions.

2.1.2 Pseudo-parallélisme vs parallélisme

Le parallélisme est soit réel, soit émulé. On a recours au terme pseudo-parallélisme (parallélisme émulé) lorsque plusieurs processus s'exécutent simultanément sur un même processeur (ou cœur). Celui-ci s'obtient par émulation grâce aux concepts de multi-programmation, de tranches de temps et de changements de contexte. Le pseudo-parallélisme s'effectue par un entrelacement de l'exécution des différentes tâches.

Le parallélisme réel se produit lorsque l'on utilise des multiprocesseurs, multi-cœurs ou multi-ordinateurs. Dans ce cas, les processus s'exécutent réellement en même temps.

2.1.3 Taxonomie du parallélisme

Il existe une taxonomie du parallélisme créée par Flynn [29]. La taxonomie de Flynn est une classification proposée originalement pour les architectures des ordinateurs mais facilement élargie au parallélisme en général. Comme décrite à la figure 2.3, celle-ci divise le parallélisme en quatre catégories. La première, SISD (Single Instruction Single Data), consiste en un ordinateur séquentiel ou un programme séquentiel. Cette catégorie n'exploite aucun parallélisme et correspond à l'architecture de base de von Neumann.

La seconde catégorie, SIMD (Single Instruction Multiple Data), représente une organisation dans laquelle on exécute plusieurs fois le même programme, chacun travaillant sur des données différentes. Au niveau matériel, les processeurs vectoriels (Sparc VIS [83, 10, 77, 50], Intel MMX/SSE/AVX [89, 90, 92, 93], Power Altivec [84], Nvidia Cuda [96, 39], AMD Radeon [91], ...) fonctionnent selon ce principe puisque plusieurs unités travaillent en parallèle sur différentes données provenant d'un vecteur ou d'une matrice. Au niveau logiciel, on applique ce concept en calcul parallèle avec des outils tel MPI ou OpenCL. Le logiciel SETI@home [66] (ou Boinc [9]) est un autre exemple d'application du SIMD.

SETI@home

SETI@Home [66] sert à analyser les informations en provenance de l'espace, obtenues par des radiotélescopes. Il fonctionne sous la forme d'un sauve-écran que l'on installe sur notre ordinateur. Quand le sauve-écran entre en fonction, il communique avec le serveur du SETI pour indiquer que l'ordinateur sur lequel il s'exécute est actuellement inactif. Le serveur du SETI envoie alors à tous les ordinateurs disponibles (inactifs) reliés à son réseau, des segments différents d'informations provenant de l'espace. Tous les ordinateurs travaillant pour le SETI font exactement la même analyse mais sur des données distinctes. C'est là un bel exemple de SIMD. Le même principe s'applique pour folding@HOME (analyse génomique), SIDock@home (calcul pour le COVID 19), ... Veuillez consulter le site de «Boinc» [9] pour plus d'informations sur les applications en cours.

La troisième catégorie, MISD (Multiple Instruction Single Data), est une organisation dans laquelle une même donnée est traitée par plusieurs unités de calcul en parallèle. Des cas réels d'implantation du MISD sont plutôt rares. On la retrouve parfois dans les applications de filtrage numérique et dans les systèmes redondants tolérants aux pannes. Par exemple, SETI@HOME envoie la même copie des segments de données à un minimum de trois ordinateurs pour se prémunir contre d'éventuelles pannes. Il est aussi possible d'imaginer d'utiliser plusieurs algorithmes de décryptage

en parallèle sur le même message codé [8].

La dernière catégorie, le MIMD (Multiple Instruction Multiple Data), représente la situation où plusieurs unités indépendantes exécutent des calculs différents sur des données différentes. Il s'agit de l'architecture parallèle la plus utilisée. On retrouve dans cette catégorie les multi-processeurs, les multi-ordinateurs et les multi-cœurs. Certains divisent cette classe en deux sous-classes, les MIMD à mémoire partagée (multi-processeurs et multi-cœurs) et les MIMD à mémoire distribuée (réseau d'ordinateurs).

Taxonomie de Flynn

Taxonomie de Flynn

	Une instruction	Plusieurs instructions
Une donnée	SISD Architecture von Neumann	MISD (Rare) Processeur systolique, pipeline, tolérance aux fautes
Plusieurs données	SIMD CM-1/CM-1, Sparc VIS, Intel MMX, Power Altivec, Nvidia CUDA, ...	MIMD Grappes de calcul, multi-processeurs, multi-coeurs, , grid, ...

Figure 2.3 – Taxonomie de Flynn

2.2 Modélisation de la concurrence

Il existe de multiples approches pour modéliser/spécifier/valider les systèmes concurrents : la logique temporelle, les réseaux de Petri, les ordres partiels, les graphes d'événements et les graphes. Ces modèles sont basés sur un formalisme rigoureux qui requièrent un certain bagage mathématique.

Pour le moment, nous présentons une méthode plus simple et visuelle appelée les «graphes de précédence». Ceux-ci s'avèrent le moyen le plus utilisé pour «modéliser» la concurrence. Il sert en fait à spécifier la concurrence au niveau des énoncés.

Un graphe de précédence est un graphe acyclique dont les nœuds correspondent à des énoncés individuels et les arcs aux relations de précédence entre les énoncés. Ainsi, un arc allant de l'énoncé E_i à l'énoncé E_j signifie que l'énoncé E_j doit débuter son exécution seulement lorsque l'énoncé E_i a terminé la sienne.

Exemple

Soit le programme 2.1 qui lit deux entrées, calcule leur somme et emmagasine le résultat. Dans ce cas, les opérations de lecture, lire(a) et lire(b), peuvent s'exécuter en parallèle. Le graphe de précédence de la figure 2.4 représente le parallélisme possible pour cet exemple.

```
1 lire(a);
2 lire(b);
3 c := a + b;
4 ecrire(c);
```

Programme 2.1 – Programme simple

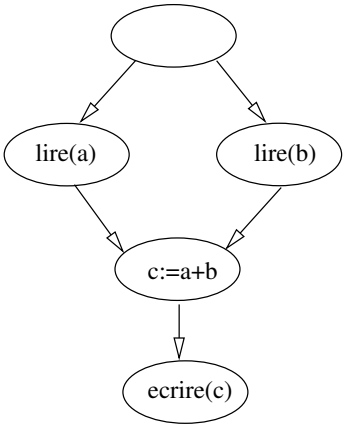


Figure 2.4 – Graphe de précédence pour le programme 2.1

La transformation d'un programme séquentiel en programme parallèle semble, à première vue, relativement simple (selon cet exemple). Elle doit cependant respecter certaines contraintes que nous abordons à la section 2.5.2.

2.3 Opérations pour le parallélisme

Les graphes de précédence sont d'une grande utilité pour visualiser le parallélisme possible dans un algorithme. Ils ne fournissent cependant aucun moyen d'exprimer des énoncés ou des programmes parallèles. Dans cette section, nous présentons plusieurs outils servant à écrire des programmes parallèles.

2.3.1 Fork/Join

Les instructions Fork et Join ont été introduites par Conway en 1963 [13] ainsi que par Dennis et Van Horn en 1966 [20]. Elles ont été les premières notations pour spécifier la concurrence.

Les instructions `Fork/Join` s'utilisent de deux façons qui diffèrent en peu de chose.

Fork/Join version 1

L'instruction **`Fork`** (Fork = embranchement) sépare un traitement unique en deux traitements parallèles. Cette première version de l'instruction **`Fork`** s'emploie de la façon suivante :

```
...
fork A;
...
```

Cette instruction démarre deux exécutions concurrentes, l'une débutant à l'énoncé étiqueté `A` et l'autre à l'instruction qui suit le `Fork`. Un exemple d'utilisation de cet énoncé est présenté au programme 2.2. La figure 2.5 illustre le graphe de précédence associé à ce programme.

```
1               E1
2               fork A
3               E2
4               ...
5       A:      E3
6               ...
```

Programme 2.2 – Programme utilisant le Fork

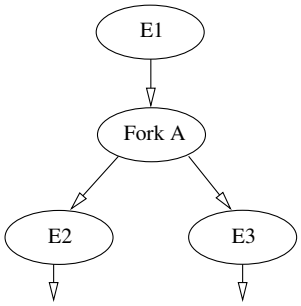


Figure 2.5 – Graphe de précédence pour le programme 2.2

Éventuellement, les deux traitements démarrés avec le `Fork` doivent se joindre en un seul. L'instruction `Join` fournit le moyen de **combiner deux ou plusieurs traitements parallèle en un seul**.

Soit deux traitements parallèles (figure 2.5). Comme ceux-ci vont à des vitesses différentes, l'un de ces traitements exécutera le `Join` avant l'autre. Dans ce cas, le premier à se rendre au `Join` termine son exécution alors que le second la poursuit. Le programme 2.3 complète le programme 2.2 avec un `Join`. La figure 2.6 illustre un comportement possible pour le `Join`.

```
1               E1
2               fork A
3               E2
4               ...
5               goto B
6       A:      E3
7               ...
8       B:      join
```

Programme 2.3 – Programme utilisant le Fork et Join

S'il y a plus de deux traitements parallèles, alors seul le dernier à exécuter le `Join` poursuivra son exécution tandis que les autres seront détruits. Cependant, lorsqu'il y a plus de deux traitements parallèles, l'instruction `Join` doit connaître le nombre de traitements à joindre et, pour ce faire, on lui ajoute un paramètre l'indiquant. Elle aura alors le format suivant :

```
...
join compte;
...
```

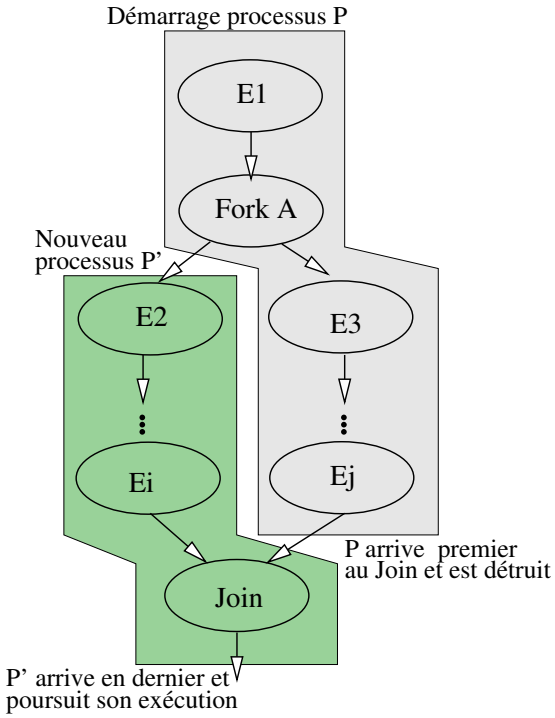


Figure 2.6 – Graphe de précédence pour le programme 2.3

L'effet de cette instruction est illustré au programme 2.4. Dans celui-ci la variable `compte` est un entier non signé, initialisé selon le nombre de traitements démarrés avec l'instruction `Fork`. S'il y a deux traitements parallèles, la variable `compte` sera alors initialisée à 2.

```
1             compte := compte -1;
2             if compte != 0 then exit();
```

Programme 2.4 – Effet d'un Join

L'instruction `Join` telle que décrite dans le programme 2.4 occasionne un important problème si plusieurs processus l'exécutent simultanément. La figure 2.7 présente une séquence d'énoncés illustrant ce problème. Dans cet exemple, `P1` commence par modifier la variable `compte` puis se fait interrompre (changement de contexte) avant d'être en mesure de la tester. Le processus `P2` prend alors le contrôle et modifie à son tour la variable `compte`. Comme sa valeur est 0, `P2` poursuit son exécution. À son retour, `P1` vérifie la valeur de la variable `compte` et celle-ci étant 0, il poursuit également son exécution. L'instruction `Join` a donc échoué.

Pour éviter cette conséquence néfaste, l'exécution de l'instruction `Join` se doit d'être atomique, c'est-à-dire que l'exécution concurrente de deux opérations «`Join`» devra avoir le même effet que leur exécution séquentielle.

compte := 2		
Processus P1	Processus P2	Valeur de compte
—-	——	2
`compte := compte -1;`		1

	`compte = compte -1;`
	`if compte != 0 then exit();`
	*// ** P2 poursuit son exécution*
		0
		0
		0

`if compte != 0 the exit();` *// ** P1 poursuit son exécution*		0
Problème : les deux processus poursuivent leur exécution !		

Figure 2.7 – Deux processus exécutant l'instruction `Join` en même temps

Atomicité du Join

L'exécution de l'instruction `Join` se doit d'être atomique. L'atomicité est une propriété utilisée en programmation concurrente pour désigner une opération ou un ensemble d'opérations d'un programme qui s'exécutent entièrement sans pouvoir être interrompues avant la fin de leur déroulement. Nous verrons plus tard comment rendre atomique l'exécution d'un ensemble d'opérations.

Le programme 2.5 présente un exemple d'utilisation des instructions `Fork/Join` et la figure 2.8, le graphe de précédence associé à ce programme.

```
1              compte := 2
2              fork  A
3              ...
4              E1
5              goto  B
6       A:     E2
7       B:     join  compte
8              ...
```

Programme 2.5 – Programme utilisant le Fork et Join

Le programme 2.6 reprend le programme 2.1 (lecture et somme de deux valeurs) pour y ajouter des `Fork/Join` et le transformer en programme parallèle. Les figures 2.9 et 2.10 présentent la traduction d'un graphe de précédence en un programme utilisant les `Fork/Join`.

Illustrons maintenant le parallélisme avec un exemple concret. Le programme 2.7 implante la copie d'un fichier. Pour paralléliser ce programme, il faut d'abord rendre indépendantes les instructions aux lignes 8 et 9. À cette fin, l'enregistrement lu à la ligne 9 (et à la ligne 5) est copié dans une autre variable avant son écriture à la ligne 8, comme l'indique le programme 2.8. Finalement, comme les deux instructions sont désormais indépendantes, leur exécution en parallèle est alors possible. Le programme 2.9 implante la version parallèle de la copie d'un fichier en utilisant les

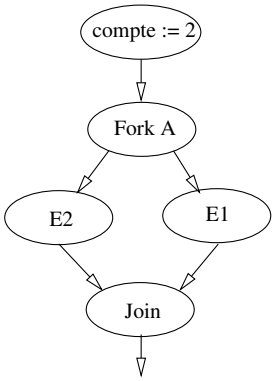
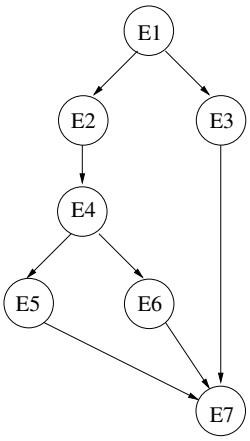


Figure 2.8 – Graphe de précédence pour le programme 2.5

```
1              compte := 2;
2              fork lec2;
3              lire(a);
4              goto join1;
5       lec2:  lire(b);
6       Join1: join compte;
7              c := a + b;
8              ecrire(c);
```

Programme 2.6 – Version parallèle de la somme de deux valeurs




```
     E1;
     compte := 3;
     fork L1;
     E2;
     E4;
     fork L2;
     E5;
     goto L3;
L2:  E6;
     goto L3;
L1:  E3;
L3:  join compte;
     E7;
```

Figure 2.9 – Graphe de précédence et son programme

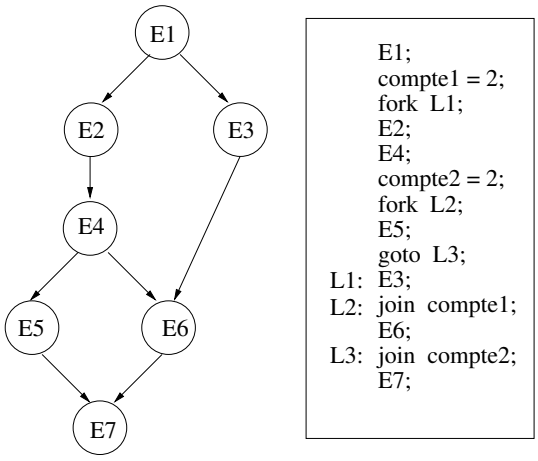


Figure 2.10 – Graphe de précédence et son programme

`Fork/Join`.

```
1     var  f,g : file of T;  // deux variable qui représentent des fichiers
2          r,s : T;
3     begin
4        reset(f);           // on se place au début du fichier
5        read(f,r);          // Lecture du premier enregistrement
6        while not eof(f)
7           begin
8              write(g,r);
9              read(f,r);
10          end;
11       write(g,r);         // écriture du dernier enregistrement
12    end.
```

Programme 2.7 – Programme qui copie un fichier

Évidemment, due à la présence obligatoire des «`goto`», l'utilisation des `Fork/Join` possède un travers fort important, celui de ne pas respecter les notions de base de la programmation structurée. En fait, l'instruction `Fork` elle-même est très similaire à une instruction `goto` étant donné son comportement.

Fork/Join : version 2

La deuxième implantation proposée pour les instructions `Fork` et `Join` ressemble à un appel de procédure [1], mais qui serait asynchrone (exactement comme une opération de création de fil d'exécution - `Tread_create`).

Avec cette version, l'instruction `Fork` reçoit maintenant un paramètre qui spécifie le nom de la procédure qui s'exécutera en concurrence avec le programme appelant. Pour se synchroniser, ce

1. Pour les besoins du cours, on ne fait pas ici de distinction entre les termes procédures, fonctions, routines et sous-routines. On les considère équivalents à quelques détails près.

```
1     var  f,g : file of T;
2          r,s : T;
3     begin
4        reset(f);
5        read(f,r);
6        while not eof(f)
7           begin
8              s := r;
9              write(g,s);
10             read(f,r);
11          end;
12       write(g,r);
13    end.
```

Programme 2.8 – Programme modifié pour faciliter la parallélisation

```
1   Var  f,g : file of T;
2        r,s : T;
3        cpt : integer;
4   begin
5     reset(f);
6     read(f,r);
7     while not eof(f)
8     begin
9              cpt := 2;
10             s := r;
11             fork L1;
12             write(g,s);
13             goto L2;
14       L1: read(f,r);
15       L2: join cpt;
16    end;
17    write(g,r);
18  end.
```

Programme 2.9 – Version parallèle de la copie d'un fichier

dernier exécutera un `Join`, en précisant comme paramètre le nom de la fonction à joindre. Suite à l'exécution du `join`, l'appelant se bloquera jusqu'à ce que la fonction appelée termine son exécution (si elle n'est pas déjà terminée). La figure 2.11 illustre le fonctionnement de cette version des `Fork` et `Join`. Dans cet exemple, l'exécution de la fonction `Fct2` démarre lorsque le `Fork` est appelé et termine à l'instruction `end`. Le programme principal (main) et la fonction `Fct2` s'exécutent concurremment à partir du `Fork` et ce, jusqu'à ce que le programme principal appelle le `Join` (ou que `Fct2` termine).

Les énoncés `Fork/Join` sont repris dans l'implantation de la plupart des environnements multi-fils (Posix, Windows, Solaris, etc). En effet, dans plusieurs cas, la création d'un nouveau fil d'exécution est très similaire à un `Fork` (appel de procédure asynchrone). Le système d'exploitation Unix fait appel à des instructions de type `Fork/Join` pour démarrer de nouveaux processus. Des énoncés semblables existent dans plusieurs langages de programmation.

Les instructions `Fork/Join` sont puissantes mais aussi dangereuses, car il est possible de les introduire «à n'importe quel endroit» dans un programme, y compris dans un énoncé d'itération ou de sélection. Une connaissance approfondie du programme à exécuter est primordiale pour déterminer les tâches qui se feront en parallèle. De plus, l'usage d'un `Fork` dans une boucle infinie entraîne bien des désagréments («fork bomb»). Toutefois, quand elles sont utilisées correctement

```
int main()                          int Fct2()
   ...                                 ...
   Fork Fct2();                        ...
   ...                                 end;
   Join Fct2()
   ...
```

Figure 2.11 – Fonctionnement des instructions `Fork` et `Join`

(discipline est le mot clé ici), ces instructions se révèlent très pratiques et très puissantes.

2.3.2 Coroutine

Les coroutines [82] sont semblables aux sous-routines (fonctions, procédures, méthodes, ...) mais elles permettent un transfert de contrôle symétrique plutôt que strictement hiérarchique. En fait, les sous-routines constituent un cas particulier des coroutines. Ainsi, la sortie d'une coroutine, contrairement à la sortie d'une routine normale terminant son exécution, s'avère possiblement le résultat d'une suspension de son exécution qui se poursuivrait éventuellement plus tard. Les coroutines ont été proposées pour la première fois en 1958 par Conway (selon Knuth [82]). Elles permettent de réaliser des traitements basés sur des algorithmes coopératifs tels que les itérateurs, les générateurs, les boucles d'événements («event loops) et la gestion des exceptions. Elles sont tombées dans l'oubli pendant plusieurs années mais sont de retour en force depuis la montée en popularité de la programmation événementielle. On retrouve maintenant le concept de coroutines dans la plupart des langages modernes (C++, Python, Java, ...).

Transfert de contrôle hiérarchique

Transfert de contrôle hiérarchique entre fonctions.

Il est bon de se rappeler qu'effectivement les transferts de contrôle entre les fonctions sont généralement strictement hiérarchiques. Cela signifie que le retour d'un appel se fait toujours directement vers l'appelant. La figure 2.12 illustre ce type de transfert de contrôle. Des instructions existent en C et C++ pour passer outre cette hiérarchie (`setjmp` et `Longjmp`), mais, due à leur similarité avec les `goto`s, elles sont rarement employées.

À l'intérieur des coroutines, le transfert de contrôle symétrique se fait exclusivement grâce à l'énoncé `resume` (remplacé aujourd'hui par `yield`). L'exécution d'un `resume` est semblable à l'exécution d'un appel de fonction. Ainsi lors de l'appel, le `resume` :

1. sauve suffisamment d'information sur l'état d'exécution de l'appelant pour permettre au contrôle de revenir plus tard à l'instruction qui suit le `resume` ;
2. transfère le contrôle à la routine nommée (en paramètre).
   - 1^{er} appel ⇒ transfert au début de la coroutine ;
   - appels suivants ⇒ transfert à la suite du `resume`.

Lorsqu'une coroutine est appelée pour la première fois, le contrôle lui est transféré à son point

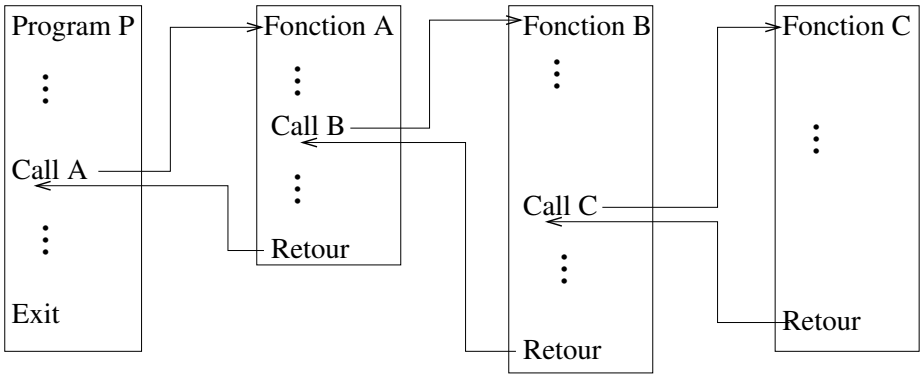


Figure 2.12 – Transfert de contrôle hiérarchique entre les sous-routines (fonctions)

d'entrée (comme une fonction normale). Une coroutine transfère ensuite le contrôle à une autre coroutine uniquement avec l'énoncé `resume`. Le contrôle est aussi retourné à une autre coroutine (possiblement l'appelante) grâce au `resume` plutôt qu'avec un énoncé `return`. L'énoncé `resume` constitue donc la seule technique pour transférer le contrôle entre les coroutines.

Un énoncé `resume` ne termine pas nécessairement l'exécution de la coroutine. Il peut donc être inséré n'importe où dans le code (pas seulement à la fin). La coroutine poursuivra son exécution plus tard et ce, à l'instruction qui suit le `resume`. De plus, comme le transfert n'est pas hiérarchique, n'importe laquelle des coroutines, pas seulement la première coroutine appelée, a la possibilité de retourner le contrôle à la routine originale (programme principal). Le mode de fonctionnement des coroutines est illustré à la figure 2.13. Dans cet exemple, un programme principal «`P`» démarre l'exécution des coroutines avec un premier appel grâce à l'énoncé `call` (le programme principal n'est pas une coroutine). De même, pour le retour au programme principal, la coroutine «`C`» recourt à un énoncé `return`. L'énoncé `resume` sert ensuite au transfert de contrôle symétrique entre les coroutines «`A`», «`B`» et «`C`».

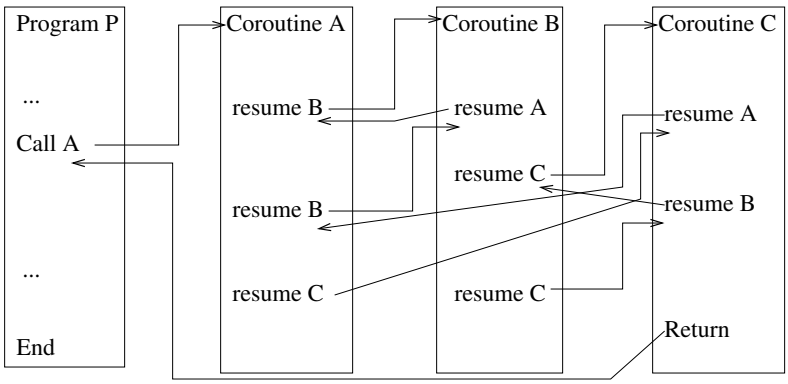


Figure 2.13 – Transfert de contrôle symétrique entre les coroutines

Les coroutines ne fournissent pas un réel parallélisme au sens où nous l'avons défini. Dans les

faits, la sémantique de leur exécution ne permet de n'en exécuter qu'une à la fois. C'est une forme de concurrence dans laquelle tous les transferts de contrôle sont entièrement spécifiés à l'avance plutôt que laissés à la discrétion du système sous-jacent. Il n'y a donc aucun non-déterminisme dans les coroutines et donc **aucun besoin de synchronisation**. En fait, il est courant de considérer chaque coroutine comme l'implantation d'un processus et l'exécution de l'énoncé `resume` comme la synchronisation entre ceux-ci. Manipulées avec soin, les coroutines constituent une technique très intéressante pour organiser une forme de concurrence. En fait, il est très possible d'implanter la multiprogrammation à l'aide des coroutines.

Les coroutines ont été introduites originalement dans les langages Simula, Bliss et Modula 2 (1960-70) et plus tard dans Boost (C++) [64]. Les coroutines de Boost sont originales en ce sens qu'elles sont soit symétriques, soit asymétriques [71] [2]. Aujourd'hui on les retrouve partout. Ainsi, les coroutines ressemblent aux fils d'exécution de niveau usager sauf qu'elles adoptent une forme de multi-tâches strictement coopérative. Les fibres de Windows (.Net 4.0) sont en fait très semblables à des coroutines. On retrouve aussi ce concept dans les langages de programmation moderne [82] tels que C# [54], Haskell [11, 100], Javascript [14, 70], Perl, Kotlin [45, 4], Python [21, 33] et C++ [19, 49, 32], ainsi que dans certains environnements tels Unity [48]. Les coroutines se retrouvent aujourd'hui au cœur de tous les systèmes à base d'événements.

2.3.3 Cobegin/Coend (Parbegin/parend)

Les énoncés `Cobegin/Coend` sont des énoncés structurés et de «haut niveau» (plus abstraits ou mois primitifs) pour spécifier l'exécution concurrente d'un ensemble d'opérations. C'est l'équivalent «parallèle» des énoncés de définition de bloc de code séquentiel, tel que `begin/end` ou «`{...}`», présents dans les langages de programmation dit «de haut niveau». Ils s'emploient de la façon suivante :

E_0;
`cobegin`
    E_1;
    E_2;
    E_3;
    ...;
    E_n;
`coend;`
E_{n+1}

Cet énoncé provoque l'exécution en parallèle des énoncés E_1 à E_n. Il est important de mentionner qu'il est possible de décomposer chaque énoncé E_i en une multitude d'énoncés variés, y compris des `cobegin/coend`. Voici une autre syntaxe courante pour ce type d'énoncés :

E_0;
$E_1 \parallel E_2 \parallel E_3 \parallel \ldots \parallel E_n$;
E_{n+1};

2. Un coroutine asymétrique connaît l'appelant, ce qui lui donne la possibilité de lui retourner le contrôle directement, ce qui n'est pas le cas avec des coroutines symétriques.

Le graphe de précédence associé à cet énoncé est illustré à la figure 2.14. Ainsi, l'exécution de l'énoncé E_0 doit terminer avant que les exécutions des énoncés E_1 à E_n puissent commencer. Ces dernières doivent terminer avant que l'exécution de l'énoncé E_{n+1} puisse débuter.

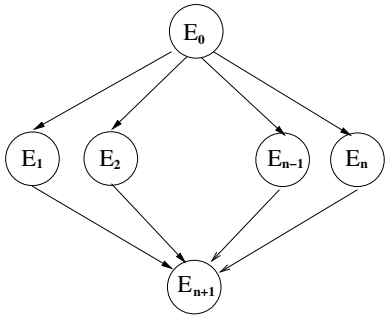


Figure 2.14 – Graphe représentant le concurrence induite par les énoncés cobegin/coend

Exemple 1

Pour ce premier exemple, reprenons le programme 2.1 qui lit deux entrées, en calcule la somme et affiche le résultat. Le programme 2.10 présente la version `cobegin/coend` de cet exemple.

```
1        cobegin
2           lire(a);
3           lire(b);
4        coend
5        c := a + b;
6        ecrire(c);
```

Programme 2.10 – Version parallèle du programme «somme»

Exemple 2

Ici, nous considérons le graphe de précédence de la figure 2.9 converti en programme parallèle à l'aide des énoncés `fork/join`. La conversion de ce graphe en programme parallèle à l'aide des `cobegin/coend` est illustrée à la figure2.15.

Exemple 3

Dans cet exemple, nous tentons de reprendre le graphe de précédence de la figure 2.10 converti en programme parallèle à l'aide des énoncés `fork/join`. La conversion de ce graphe en programme parallèle à l'aide des `cobegin/coend` se révèle alors impossible. Ceci met en évidence que les énoncés `fork/join` (version «`goto`») sont plus puissants que les énoncés `cobegin/coend`.

Exemple 4

Pour ce dernier exemple, nous reprenons le programme 2.8) qui copie un fichier. Le programme 2.11 illustre la version `cobegin/coend` de ce programme.

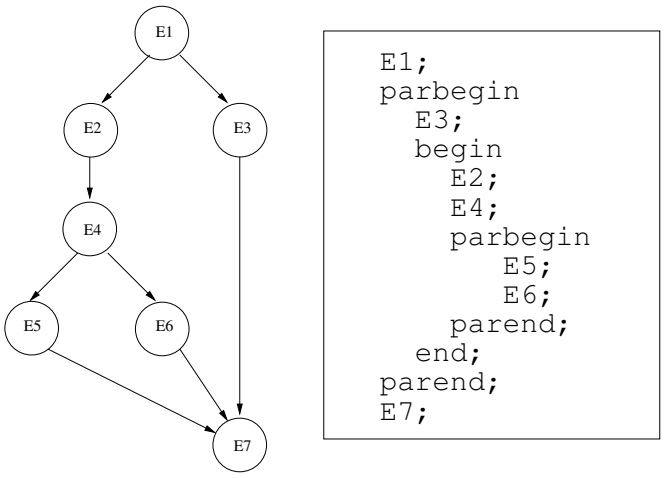


Figure 2.15 – Graphe de précédence et son programme

```
1    Var  f,g : file of T;
2         r,s : T;
3    begin
4       reset(f);
5       read(f,r);
6       while not eof(f)
7       begin
8          s := r;
9          parbegin
10            write(g,s);
11            read(f,r);
12         parend;
13     end;
14     write(g,r);
15   end.
```

Programme 2.11 – Version parallèle du programme «somme»

Conclusion

Les énoncés `cobegin/coend` s'ajoutent facilement à un langage moderne structuré (C++, Java, C#, ...). Ils ne possèdent toutefois pas la même capacité d'expression que le `fork/join` (version 1) comme nous avons pu le constater à l'exemple 3. **Mais est-ce un problème sérieux ?**

En fait non, car parmi tous les cas de concurrence modélisables par un graphe de précédence, plusieurs ne représentent aucune situation réelle nécessitant une implantation. Ainsi, l'exemple 3 est un exemple de graphe qui n'a guère de chance d'exister dans le «monde réel». Notre incapacité à l'exprimer n'est donc pas inquiétante. En fait, une ré-écriture du programme permettrait probablement de le représenter.

Donc les énoncés `cobegin/coend` sont moins puissants mais ils le sont suffisamment pour exprimer la plupart (sinon tous) des traitements concurrents.

De plus, la syntaxe des énoncés `cobegin/coend` rend explicite les routines qui seront exécutées en parallèle et fournit une structure de contrôle respectant les principes de la programmation

structurée (ce qui n'est pas le cas de la version 1 du `Fork/Join`).

Des variantes des énoncés `cobegin/coend` ont été implantées dans les langages Algol 68 [44, 94], CSP [87], Edison [65], Argus [47] et Chapel [22]. Peu de langages modernes et couramment utilisés fournissent l'équivalent des énoncés `cobegin/coend`.

2.3.4 Déclaration de processus

Des programmes imposants sont souvent construits sous la forme d'une collection de routines séquentielles (fonction, méthodes, ...) exécutées en parallèle. L'approche alors préconisée consiste à déclarer des fonctions «normales» et à les activer avec des `cobegin/coend`, des `Forks` ou autres énoncés similaires (`start`). Toutefois, la structure des programmes concurrents serait beaucoup plus lisible et compréhensible si la déclaration de la routine indiquait si elle sera oui ou non exécutée en parallèle.

Un énoncé de déclaration de processus a donc été proposé pour clarifier ces situations. Le programme 2.12 présente la syntaxe de cet énoncé. Ce type de déclaration indique clairement quelles routines seront exécutées en parallèle. De plus, il est possible d'ajouter des paramètres, semblables syntaxiquement à ceux d'un énoncé d'itération, indiquant le nombre de versions de ce processus appelées à être exécutées en parallèle. Le programme 2.13 en est un écrit dans le langage de programmation parallèle SR qui utilise l'énoncé «`process`» afin d'implanter une multiplication de matrices en parallèle. Le paramètre, fourni à l'énoncé processus, indique qu'il faut démarrer «`n`» processus, chacun d'eux recevant une valeur distincte de «`i`». Chaque processus est en charge de calculer une ligne de la matrice résultante (la ligne «i»). Quant à elles, les boucles internes (`fa` pour «`for all`») calculent, pour chaque élément de la ligne résultante (ligne 4), le produit scalaire (ligne 6).

```
1          process P1(...)
2          {
3              ....
4
5          }
```

Programme 2.12 – Déclaration de processus

```
1  process calcul(i := 1 to n )
2     var j,k:int
3     write("Processus ", i)
4     fa j := 1 to n ->
5         c[i,j]:=0
6         fa k := 1 to n ->
7             c[i,j]:= c[i,j] + a[i,k]* b[k,j]
8         af
9      af
10 end
```

Programme 2.13 – Multiplication de matrice en parallèle.

En lien avec ce type de notation, une question est soulevée. Comment lancer l'exécution des processus qui déjà déclarés ? Deux méthodes de démarrage s'imposent : implicite et explicite. Suivant l'approche implicite, aucun appel, ni instruction spéciale ne sont requises. Les processus sont automatiquement démarrés lorsque l'exécution du programme «arrive» dans la portée de déclaration des processus ou lorsqu'une instance de l'objet contenant la déclaration est créée. C'est le cas précisément lorsque des «processus» sont déclarés dans une classe : ils démarrent automatiquement lorsqu'une instance de la classe est créée. Cependant, selon l'approche explicite, il est nécessaire,

pour démarrer un processus, de l'appeler ou de recourir à une instruction spéciale («`start`» par exemple.)

Certains langages ou bibliothèques fournissent des déclarations explicites de processus. C'est le cas des langages DP [67], SR, JR et ADA. Les bibliothèques MPI et OpenCL offrent aussi des mécanismes similaires. Dans les cas de SR, JR, ADA le démarrage est implicite. Dans le cas de OpenCL, le démarrage des processus (kernel) est explicite. MPI est un cas particulier. La déclaration de processus est implicite (c'est tout le programme) mais le démarrage est explicite (MPI_INIT).

2.3.5 Analyse des différentes opérations

Nous venons de définir de multiples opérations de création de processus. Analysons maintenant ces opérations en lien avec les différents comportements que nous avons présentés à la section **??**.

1. Au niveau de l'exécution
   - `Fork/Join`
     Dans les deux versions, le parent et l'enfant poursuivent leur exécution en parallèle.
   - `cobegin/coend`
     Avec cet énoncé, le parent est bloqué jusqu'à ce que tous ses enfants terminent.
   - `process`
     Avec cet énoncé, les deux situations sont possibles. Dans SR et JR, le parent se bloque. Dans MPI et OpenCL, le parent poursuit son exécution.
2. Au niveau du partage
   - `Fork/Join`
     Dans la version `goto`, le parent et l'enfant partagent toutes leurs variables. Dans la version «appel de procédure», le parent et l'enfant ne partagent que les variables globales. Les variables locales, elles, ne le sont pas.
     Ainsi, les fonctions de création de fils (`thread_create`) ne permettent que le partage des variables globales. Cependant, le système Unix implante une version du `fork` qui a l'avantage d'offrir une option de partage partiel des variables entre le parent et l'enfant.
   - `cobegin/coend`
     Avec cet énoncé, le parent et l'enfant partagent toutes leurs variables.
   - `process`
     Avec cet énoncé, les deux situations sont possibles. Dans SR et JR, le partage est partiel (variables globales). Dans MPI et OpenCL, il n'y a aucun partage car chacun fonctionne avec sa copie des données.

2.4 Multi-fils ou événementiel

Il y a un véritable débat au sein de la «communauté informatique» sur le recours ou non, au multi-fils dans certains systèmes modernes. Il porte sur l'usage de l'approche multi-fils versus l'approche événementielle (basée sur une boucle d'événements). Certains sont d'avis que le multi-fils est trop coûteux et trop lourd dans les environnements où il y a un grand nombre de requêtes (sites

web par exemple). Dans ces situations, les systèmes génèrent trop de fils d'exécution, ce qui mène à une surcharge pour le système sous-jacent (changement de contexte, consommation de mémoire, ...) et à une grande complexité au niveau de la programmation (synchronisation, mise au point, ...). Les systèmes événementiels, eux, ne souffrent d'aucun de ces problèmes et sont généralement beaucoup plus performants sous une lourde charge. Toutefois, ces derniers ne profitent pas pleinement des systèmes multi-processeurs et multi-cœurs. L'annexe B fournit davantage d'informations sur ce sujet.

2.5 Problèmes dus à la concurrence

Le parallélisme apporte un niveau de performance des plus intéressant mais il génère plusieurs nouvelles situations problématiques. Cette section traite de celles-ci.

2.5.1 Programmation

Les individus ont en général de l'aisance et une bonne capacité à penser/regarder/produire une activité à la fois. Développer une habileté pour produire des solutions parallèles exigent une changement de paradigme et de l'entraînement. Nous verrons certaines solutions proposées afin de faciliter cette transition.

2.5.2 Conditions essentielles à la concurrence

Fréquemment, il s'avère ardu de déterminer quels énoncés peuvent ou non être exécutés en parallèle. En fait, pour y parvenir, il faut tout simplement que ceux-ci soient disjoints. **Mais qu'est-ce que des énoncés disjoints ?**

Présentons d'abord quelques notations.

- Soit

$$L(I_i) = \{a_1, a_2, ..., a_m\}$$

  l'ensemble de lecture de l'énoncé I_i (I pour instruction). Cet ensemble contient toutes les variables référées par l'énoncé I_i telle que la valeur de chaque a_j ne change pas pendant toute la durée de l'exécution de I_i.

- Soit

$$E(I_i) = \{b_1, b_2, ..., b_m\}$$

  l'ensemble d'écriture de l'énoncé I_i. Cet ensemble contient toutes les variables référées par l'énoncé I_i telle que la valeur de chaque b_j est modifiée pendant toute la durée de l'exécution de I_i.

Exemple : Ensembles de lecture et d'écriture pour le programme «somme»

Reprenons l'exemple du programme `somme` :

```
lire(a)
lire(b)
c := a + b
ecrire(c)
```

Les ensembles de lecture et d'écriture pour chacun des énoncés sont ainsi définis :

- L(lire(a)) = {}
- L(lire(b)) = {}
- L(c=a+b) = {a,b}
- L(ecrire(c)) = {c}
- E(lire(a)) = {a}
- E(lire(b)) = {b}
- E(c=a+b) = {c}
- E(ecrire(c)) = {}

Les trois conditions qui doivent obligatoirement être respectées pour que deux énoncés I_1 et I_2 exécutés concurremment produisent toujours le bon (le même) résultat sont :

1. $L(I_1) \cap E(I_2) = \{\}$
2. $E(I_1) \cap L(I_2) = \{\}$
3. $E(I_1) \cap E(I_2) = \{\}$

On appelle ces trois contraintes les conditions de Bernstein du nom de leur créateur. En fait, elles expriment simplement ce qui suit : des énoncés qui accèdent des variables communes sont en conflits si l'un d'entre eux effectue une écriture (modifie une des variables communes).

2.5.3 Difficulté de mise au point

Les programmes parallèles sont beaucoup plus difficiles à mettre au point («debugger») que les programmes séquentiels. La raison en est très simple, c'est qu'ils sont difficiles à tester.

Pourquoi sont-ils difficiles à tester ?

Lorsqu'une erreur provoquée par une mauvaise manipulation de données partagées (énoncés non disjoints) se produit sur un programme parallèle, il est très difficile de la retracer.

En effet, si on y réfléchit, quel est l'élément fondamental qui permet d'effectuer des tests, donc de faire la mise au point de nos programmes (séquentiels) ?

Les tests sont fondamentalement basés sur notre capacité à reproduire un traitement. Ainsi, lors de la mise au point, on s'attend à ce qu'un programme donne toujours les mêmes résultats lorsqu'on leur applique les mêmes tests. Dans un programme parallèle formé d'énoncés non disjoints, ce n'est pas toujours le cas. Il est fort possible que des exécutions répétées sur les mêmes données en entrée ne produisent jamais le même résultat.

Mise au point

La mise au point d'un programme est basée sur notre capacité à reproduire le traitement qui a conduit à l'erreur.
Dans un programme parallèle, cela peut s'avérer impossible.

Voici quelques exemples de situations problématiques.

Exemple 1 : incrémentation de la variable x en parallèle

Soit une variable `x` initialisée à `0`. Soit deux processus qui modifient en parallèle cette variable telle qu'illustrée par le programme 2.14. On constate que P_1 incrémente `x` de 1 et que P_2 incrémente `x` de 2.

```
1 cobegin
2    P1: x := x + 1;
3    P2: x := x + 2;
4 coend;
```

Programme 2.14 – Deux énoncés modifiant la variable x en parallèle.

Il est parfaitement raisonnable d'espérer que la valeur finale de `x`, après l'exécution de P_1 et P_2, soit 3. Malheureusement cela ne s'avère pas toujours.

Étudions donc le comportement de ces énoncés. Un énoncé d'assignation en étant un de haut niveau ne s'implante pas toujours comme une seule instruction au niveau de la machine (indivisible et atomique). Elle se divise généralement en trois instructions machines telles qu'illustrées par le programme 2.15. Selon l'ordre d'exécution des instructions, comme l'indique la figure 2.16, le résultat en sortie peut être 1, 2 ou 3.

```
1 assignation :     // x = x + 1
2        load  x    // Charge x dans un registre (R1)
3        Add   1    // Additionne 1 (ou 2)
4        Store x    // Emmagasine la nouvelle valeur de x en mémoire
```

Programme 2.15 – Implantation d'un énoncé d'assignation.

Exécution 1			Exécution 2			Exécution 3		
P1	P2	x	P1	P2	x	P1	P2	x
—-	——	0	—-	——	0	—-	——	0
`load x;`		0		`load x`	0	`load x;`		0
	`load x`	0	`load x;`		0	`add 1`		0
	`add 2`	0	`add 1`		0	`store x`		1
	`store x`	2	`store x`		1		`load x;`	2
`add 1`		2		`add 2`	1		`add 2`	2
`store x`		1		`store x`	2		`store x`	3
valeur finale : x=1			valeur finale : x=2			valeur finale : x=3		

Figure 2.16 – Résultats possibles des sommes en parallèle

Exemple 2 : programme qui copie un fichier

Ici, on reprend le programme parallèle qui consiste à copier un fichier (version `cobegin/coend`). Le programme 2.16 présente une implantation correcte tandis que le programme 2.17 en présente une erronée. En effet, la ligne 8 du programme 2.16 a été, par erreur, insérée dans les énoncés parallèles `parbegin/parend` du programme 2.17.

```
1     var  f,g : file of T;
2          r,s : T;
3     begin
4        reset(f);
5        read(f,r);
6        while not eof(f)
7           begin
8              s := r;
9              parbegin
10                write(g,s);
11                read(f,r);
12             parend
13          end;
14       write(g,r);
15    end.
```

Programme 2.16 – Programme qui copie un fichier (version correcte).

```
1     var  f,g : file of T;
2          r,s : T;
3     begin
4        reset(f);
5        read(f,r);
6        while not eof(f)
7           begin
8              parbegin
9                 s := r;
10                write(g,s);
11                read(f,r);
12             parend
13          end;
14       write(g,r);
15    end.
```

Programme 2.17 – Programme qui copie un fichier (version incorrecte).

Analysons les comportements possibles de ce programme incorrect. Soit l'état initial suivant pour nos fichiers «f» et «g» :

- $f = \{1, 2, 3, ...m\}$ où $1, 2, ..., m$ représentent les enregistrements contenus dans le fichier.
- $g = \{\}$ indiquant que le fichier est initialement vide.

Évidemment un comportement normal produirait à la fin de l'exécution :

- $f = \{1, 2, 3, ...m\}$
- $g = \{1, 2, 3, ...m\}$

Concentrons nous sur les lignes suivantes du programme incorrect :

9. `s := r`
10. `write(g,s)`
11. `read (f,r)`

Si le premier élément du fichier est copié correctement, on obtient

- $f = \{3, ...m\}$ // Les deux premiers enregistrements sont lus...
- $g = \{1\}$
- $r = 2$
- $s = 1$

Les possibilités de sortie à la prochaine itération sont données à la figure 2.17. La colonne 1 de la figure présente l'ordre d'exécution des instructions. En effet, on suppose que ces dernières, étant démarrées simultanément, s'exécutent dans n'importe quel ordre. Le tableau présente donc six possibilités d'ordonnancement pour le traitement de ces instructions. La colonne suivante fournie les sorties possibles après une seule itération dans la boucle, en considérant chaque énoncé comme indivisible (ce qui n'est pas toujours le cas). On remarque qu'il y a trois sorties distinctes pour le fichier «`g`» après cette unique itération. Le nombre de sorties possible pour m éléments devient donc énorme (si $m = 10000 \rightarrow$ on pourrait obtenir 3^{10000} sorties différentes). Il est donc très peu probable que la personne en charge de la mise au point obtienne deux fois le même résultat lors des tests.

Ces différentes sorties pour chacune des exécutions sont dues aux influences externes qui accélèrent ou ralentissent l'exécution de certains énoncés ou processus. La programmation parallèle est réellement plus hasardeuse que la programmation séquentielle.

Ordre d'exécution des instructions	contenu de g	r	s	contenu de f
9 ; 10 ; 11	$\{1, 2\}$	3	2	$\{4, ...m\}$
9 ; 11 ; 10	$\{1, 2\}$	3	2	$\{4, ...m\}$
10 ; 9 ; 11	$\{1, 1\}$	3	2	$\{4, ...m\}$
10 ; 11 ; 9	$\{1, 1\}$	3	3	$\{4, ...m\}$
11 ; 9 ; 10	$\{1, 3\}$	3	3	$\{4, ...m\}$
11 ; 10 ; 9	$\{1, 1\}$	3	3	$\{4, ...m\}$

Figure 2.17 – Différences séquences pour la copie d'un enregistrement

Parfois, le seul moyen pour la personne qui développe un programme parallèle de parvenir à identifier une erreur consiste à examiner le code. Il existe toutefois, et heureusement, de plus en plus d'outils permettant la mise au point de programme parallèle, des «debugger» par exemple, qui supportent la mise au point de programmes multi-fils.

2.5.4 Difficulté pour l'interaction

On se rappelle que la conception de programmes parallèles nécessite fréquemment des communications entre les différents processus afin d'accomplir une tâche commune.

Donc, en plus de tous les problèmes de conception et de mise au point, l'interaction entre les processus s'avère aussi très complexe à réaliser. Aussi, la conception des protocoles de communication,

qui permettent aux processus d'échanger de l'information de façon sûre et efficace, se doit d'être fort rigoureuse. Le cours «*IFT585 -Télématique*» présente une série de tels protocoles permettant à des systèmes parallèles de communiquer adéquatement.

Dans certains cas, afin de faciliter cette communication, celle-ci passe par un programme central (administrateur) qui se charge de diffuser l'information. Dans d'autres situations la communication passe directement par les intervenants (pair-à-pair).

Nous traitons de la communication inter-processus dans le prochain chapitre.

2.5.5 Preuve de bon fonctionnement

Il est important de noter que l'application de tests ne sert qu'à prouver l'existence d'erreurs dans un programme et non leur absence. Ainsi, même si on applique des tests systématiques, ceci n'assurera aucunement qu'un programme est exempt d'erreur. Toutefois cela nous amène un certain degré de confiance dans le bon fonctionnement du dit programme.

Lorsque l'on conçoit des programmes parallèles, il est compliqué d'appliquer des tests, mais, comme nous l'avons déjà mentionné, il est encore plus ardu de détecter et corriger les erreurs. Pour s'assurer de la validité des programmes parallèles, on recourt de plus en plus à des méthodes formelles. Celles-ci fournissent la preuve du bon fonctionnement du programme. Ces méthodes existent depuis longtemps déjà et sont appliqués aussi aux programmes séquentiels. Toutefois, et c'est sans surprise, il est beaucoup plus difficile de prouver le bon fonctionnement d'un programme parallèle que celui d'un programme séquentiel.

Lorsque l'on veut s'assurer du bon fonctionnement d'un programme avec des méthodes formelles, on doit vérifier certaines propriétés associées aux programmes. Ainsi démontrer l'exactitude ou la validité d'un programme concurrent se résume à valider :

- des propriétés de sûreté (safety properties) ;

  Ces propriétés indiquent que des événements indésirables ne doivent pas se produire (something bad must not happens), donc à s'assurer que le programme donnera le bon résultat. S'assurer que le programme ne contient pas d'interblocage et que l'exclusion mutuelle est respectée sont des exemples de telles propriétés.

- Les propriétés de vivacité (liveness properties) ;

  Les propriétés de vivacité certifient la bonne progression du programme, i.e. que des événements souhaitables se produisent (something good must happens). Parmi ces propriétés, on retrouve : la terminaison du programme, l'entrée éventuelle d'un processus en section critique, l'absence de famine, etc.

- Les propriétés d'équité ;

  Cette propriété n'est pas primordiale mais elle est recherchée.

Propriétés d'un programme

Toutes les proptiétés d'un programme s'expriment en termes de sureté et de vivacité.

Nous approfondissons les méthodes formelles dans un prochain chapitre.

Annexe A

Concurrence et parallélisme

A.1 Concurrence VS parallélisme

Au fil des années, maintes discussions eurent lieu [12, 30, 34, 37, 42, 58, 59, 60, 27, 73, 74] pour clarifier, différencier et mieux définir les concepts de concurrence et de parallélisme. Les références citées ne représentent qu'une mince fraction de toutes les opinions et discussions sur le sujet mais de celles-ci, on retient déjà plusieurs définitions distinctes (et contradictoires) pour ces deux concepts.

En premier lieu, mentionnons déjà que certains, dont Gustafson et Xu[28], ne font aucune distinction entre les deux, du moins, tant que cela n'est pas nécessaire.

Par contre, plusieurs spécialistes du sujet traitent la concurrence et le parallélisme comme deux concepts réellement distincts. De plus la définition donnée généralement au terme `concurrence` varie elle aussi d'un auteur à l'autre.

A.1.1 Parallélisme

La majorité des auteurs[30, 34, 42, 28, 26, 58, 59, 27, 73, 74] définissent par le terme parallélisme, un état à l'exécution qui nécessite la présence de plusieurs unités de calcul (processeurs ou cœurs). Le parallélisme concerne donc l'usage de plusieurs processeurs ou cœurs dans le but ultime d'améliorer la performance de applications.

Parallélisme

Le parallélisme est un état à l'exécution et requiert l'usage de multiples processeurs.

Parallélisme

Le parallélisme concerne «*faire plusieurs actions en même temps*».

A.1.2 Concurrence

Au moins deux définitions de la concurrence se confrontent à cette première définition du parallélisme. La principale, et la plus acceptée, est de définir la concurrence en terme de propriété conceptuelle liée à la structure d'un programme. Selon Rob Pike[58, 59], la concurrence concerne un problème plus général et plus large que le simple parallélisme. La concurrence est une technique permettant de structurer un programme par la composition de tâches indépendantes. La concurrence désignerait alors une façon de structurer un programme de telle manière qu'il puisse éventuellement profiter du parallélisme pour mieux performer. Que ces tâches soient exécutées réellement en même temps ou non est un détail relié à l'implantation. Elles ont autant la possibilité de l'être sur plusieurs processeurs, séquentiellement sur un seul processeur ou en pseudo-parallélisme, i.e. en alternance sur un seul processeur, et ce en utilisant les tranches de temps.

Cette définition de la concurrence est très large. Le parallélisme n'est pas le but ultime de la concurrence. Le but est plutôt d'obtenir une bonne structure. Cette définition de la concurrence permet le parallélisme et le rend facile.

Concurrence

La concurrence est une technique permettant de structurer un programme par la composition de tâches pouvant s'exécuter indépendamment. Que ces tâches s'exécutent réellement en même temps ou non est un détail relié à l'implantation.

Concurrence

La concurrence concerne la gestion de plusieurs tâches en même temps.

Ainsi, selon cette première définition, la concurrence est un concept général qui englobe le parallélisme. La figure A.1 illustre la relation entre ces deux concepts. La figure A.2 présente les liens entre ces concepts et les différentes notions présentées dans les cours de système d'exploitation et de système concurrent et parallélisme. Enfin, la figure A.3 fait le lien entre ces concepts et les constructions abordées dans ces mêmes cours.

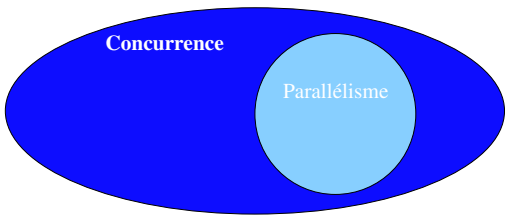


Figure A.1 – La concurrence est un concept plus général que le parallélisme

La seconde définition de la concurrence [12, 34, 28, 27, 74], celle adoptée par un certain nombre d'auteurs, concerne seulement l'exécution simultanée de plusieurs tâches par alternance sur un seul processeur (grâce au concept de tranches de temps). Cela donne l'impression de parallélisme à

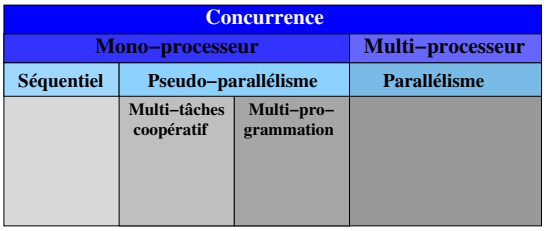


Figure A.2 – Liens entre concurrence et différents concepts abordés dans les cours

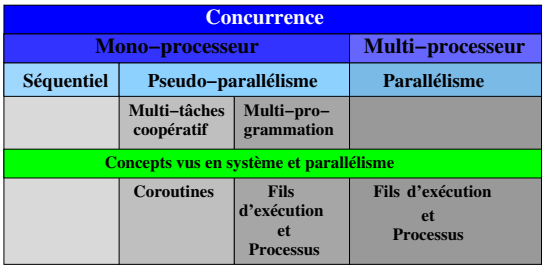


Figure A.3 – Liens entre concurrence et différentes constructions abordées dans les cours

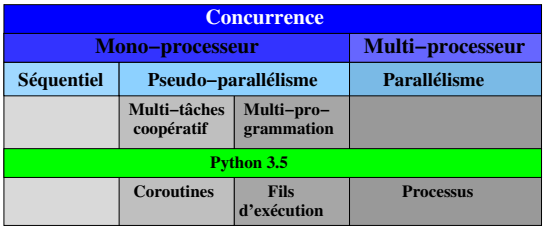


Figure A.4 – Concurrence et parallélisme en Python

l'usager (pseudo-parallélisme ou parallélisme virtuel).

Le tableau A.1 résume la distinction entre parallélisme et concurrence selon cette dernière définition.

	Concurrence # 1	Concurrence #2	Parallélisme
Base	Gérer et exécuter plusieurs tâches en même temps	Gérer et exécuter plusieurs tâches en même temps	Exécuter plusieurs tâches en même temps
Obtenu par	Dépend de l'implantation	Alternance des opérations	Plusieurs processeurs
Bénéfices	Conception modulaire	Plus de travail/unité de temps	Accélération du calcul et plus de rendement
Utilise	Dépend de l'implantation	Changement de contexte	Plusieurs processeurs
# de UCTs	Dépend de l'implantation	1	2 et plus

Table A.1 – Table comparant les définitions de la concurrence

A.1.3 Autres définitions

Plusieurs autres définitions pour ces deux termes sont en usage, mais celles-ci sont généralement moins acceptées.

Ainsi, Jenkov [42] définit la concurrence comme le fait d'exécuter des tâches en même temps et le parallélisme comme celui de diviser les tâches et d'exécuter les sous-tâches résultantes en parallèle.

Harper [37], quant à lui, suggère une définition qui diffère de toutes les autres. Pour lui, la concurrence concerne seulement la composition de tâches non déterministes et le parallélisme concerne seulement la performance asymptotique de programmes qui présentent un comportement déterministe. En gros, on parle de concurrence quand il n'y a aucune façon de savoir si un gain de performance est possible à cause du non déterministe. Il y aurait parallélisme quand on peut déterminer le gain de performance par un calcul de complexité.

D'autres [60] préfèrent définir la concurrence en terme de gestion des accès à un état partagé à partir de plusieurs processus, tandis que le parallélisme concerne l'usage de plusieurs processeurs pour améliorer la performance.

Une discussion de 2011 sur StackOverflow [26] regroupe les visions des auteurs sur le sujet. On y retrouve toutes les variantes proposées pour définir ces deux termes.

A.1.4 Autres notions reliées à la concurrence

Type de concurrence

Il existe selon [60], plusieurs niveaux de concurrence :

- le bas niveau qui utilise des instructions atomiques ;
- Le niveau intermédiaire qui utilise des verrous (sémaphores, ...) ;
- le haut niveau qui remplace les verrous par les «futures» (communication par messages).

Solution aux données partagées

Pour éviter les problèmes de synchronisation, il est proposé par [60] :

- de ne pas partager de données mutables ;
- d'utiliser des verrous ;
- de recourir aux des structures de données qui supportent la concurrence (*threadsafe*).

Support par les langages de programmation

Plusieurs langages de programmation fournissent des constructions pour implanter le parallélisme. Certains, tel C++, ne font qu'enrichir les constructions du système sous-jacent. D'autres, tel Python, implantent des constructions distinctives (voir figure A.4). Il s'agit des fils d'exécution (`threading`), des coroutines (`asyncio`) et des processus (`multiprocessing`). Les fils d'exécution de Python ne peuvent profiter des multiples cœurs ou processeurs à cause du `Gil` [2]. Ils constituent donc plutôt une forme de pseudo-parallélisme. Le «*multiprocessing*» en revanche permet d'utiliser les multiples processeurs.

A.2 Conclusion

Pour les besoins du cours IFT630, il n'est pas vraiment primordial de distinguer concurrence et parallélisme car les solutions apportées ne requièrent pas une telle distinction. Toutefois, lors de certaines situations particulières, il pourrait être important de nuancer la définition qui convient au contexte. Dans ce cas, une identification claire de la définition choisie est requise, universellement acceptée ou non.

Annexe B

Approche multi-fils ou événementielle

Maintes discussions ont eu lieu et ce, depuis plusieurs années [17, 24, 25, 35, 38, 40, 46, 55, 62, 75, 78] concernant la meilleure approche pour implanter des serveurs performants. En fait, l'approche choisie doit pouvoir supporter une mise à l'échelle adéquate (jusqu'à plusieurs milliers de requêtes par seconde). Les deux approches privilégiées actuellement sont celle basée sur le multi-fils et celle orientée événements.

Dans la littérature, on retrouve de nombreux documents faisant l'éloge d'une approche plutôt qu'une autre, mais depuis l'apparition d'environnements populaires tels que Nginx et Node.js, certains affirment la supériorité de l'approche orientée événements par rapport à celle basée sur le multi-fils.

Avant de comparer les deux approches, détaillons leur mode de fonctionnement respectif.

B.1 L'approche multi-fils

L'architecture multi-fils est conçue de telle façon qu'on associe un fil d'exécution à chacune des requêtes (ou connexions) faites au serveur. Chaque fil est ainsi en mesure de traiter une requête et exécuter les entrées/sorties requises par des appels synchrones (bloquants). Cette méthode est la plus naturelle pour traiter des entrées/sorties. Le fil retourne ensuite la réponse directement au client.

Cette architecture fournit un modèle de programmation connu et maîtrisé puisque chacune des tâches requises à la gestion d'une requête peut être codée séquentiellement. Elle offre ainsi une abstraction mentale simple en isolant chaque requête et en cachant la concurrence. Cette concurrence s'obtient en employant plusieurs fils ou processus simultanément. Le programme B.1 illustre la structure approximative d'un serveur fonctionnant selon ce principe.

Initialement, cette architecture fut basée sur une approche multi-processus, traditionnelle dans Unix, consistant à utiliser un processus par requête ou connexion. C'est ce modèle qui servi au premier serveur HTTP («Cern Httpd»). Les processus, ayant chacun leur propre espace d'adresses, traitent chaque requête isolément et n'exigent ainsi aucune synchronisation (pas de mémoire partagée).

```
1 ...
2 while (true)
3 {                                                    \
4     -> Lecture des informations sur la requête        |
5     -> Lecture du fichier (ou traitement de la requête)  | 3 opérations bloquantes
6     -> Envoie de la réponse                           |
7 }                                                    /
```

Programme B.1 – Structure approximative d'un serveur multi-fils

Dû au fait que les processus possèdent une structure lourde et que leur création est coûteuse, les serveurs démarraient plusieurs processus par anticipation au moment de leur initialisation («*pre-fork*»). Certains serveurs, par la suite, créaient de nouveaux processus au besoin. Tous les processus partageaient le port de communication (socket), ce qui leur permettaient d'attendre une nouvelle requête, de la traiter, de répondre au client, puis d'attendre la requête suivante. La figure B.1 illustre cette architecture.

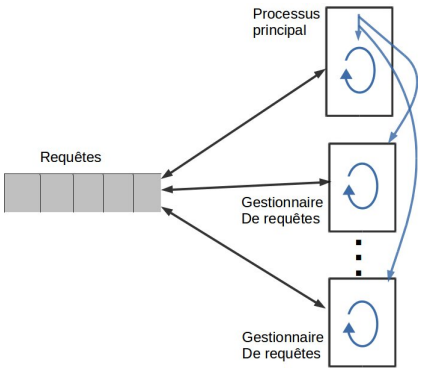


Figure B.1 – Architecture avec de multiples processus

À l'origine, Le choix de cette approche fut justifié par le fait que la plupart des bibliothèques associées aux langages n'étaient pas sécures par rapport aux multi-fils («*thread-safe*»). De plus, elle avait aussi l'avantage d'être très fiable car, dans l'éventualité d'un problème, un seul processus est affecté. Le parent démarrait alors rapidement un processus de remplacement.

Apache applique ce modèle avec le module «MPM prefork» [6].

La lourdeur de la structure d'un processus (consommation de mémoire, changements de contexte, etc) limite cependant le nombre de processus que l'on peut créer et, du même coup, le nombre de requêtes que l'on peut traiter simultanément. Cette architecture souffre donc de graves problèmes de mise à l'échelle et répond difficilement à une augmentation du nombre de requêtes.

Pour remédier à ces inconvénients, l'architecture multi-fils a été conçue. Rappelons qu'un fil d'exécution est un processus léger et que tous les fils d'un même processus s'exécutent dans le même espace d'adresses. La structure d'un fil est donc beaucoup moins lourde que celle d'un processus puisqu'elle nécessite moins de ressources, réduisant ainsi de façon significative les temps de démarrage et de changement de contexte.

Une architecture multi-fils respecte le même principe que celui de l'architecture multi-processus,

à savoir que l'on assigne un fil par requête. Il y a cependant quelques différences. Les multiples fils d'exécution partagent tous le même espace d'adresses, donc un même état global. L'empreinte en mémoire est moins grande car l'unique mémoire consommée par un nouveau fil est sa pile. La création et la destruction d'un fil s'en trouvent donc simplifiées. Une autre conséquence positive est de rendre possible l'implantation de facilités communes, telle la cache partagée. La synchronisation et la coordination des fils deviennent toutefois nécessaires.

Plusieurs architectures sont disponibles pour implanter des serveurs multi-fils. L'une d'entre elles emploie un fil en charge de la réception des requêtes qui, ensuite, les distribue aux fils travailleurs. Une seconde architecture propose que chaque fil «écoute» directement sur le port de communication afin de recevoir lui-même les requêtes à traiter. Ces deux architectures sont illustrées à la figure B.2

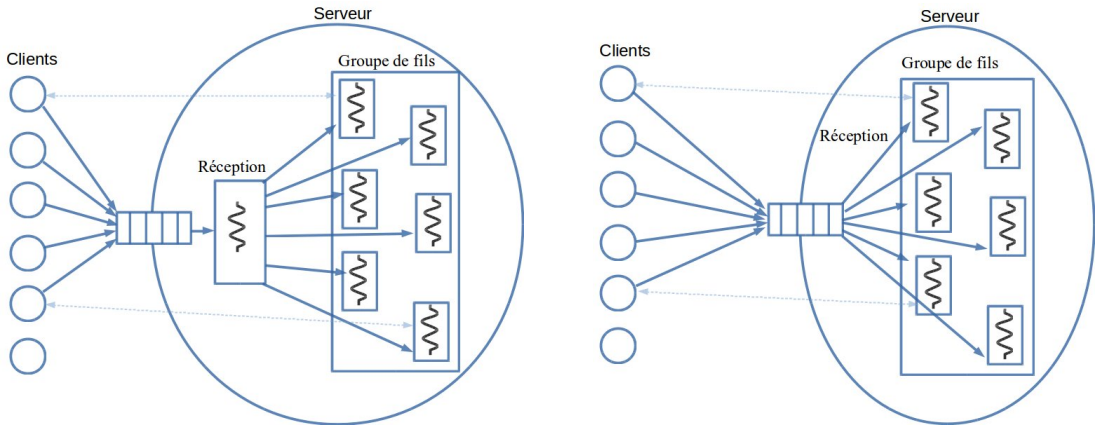


Figure B.2 – Architectures multi-fils

Les modules «Apache MPM worker» [7] et «Apache MPM event» [5, 23] combinent le multi-processus et le multi-fils. Tous deux créent plusieurs processus, chacun d'eux gérant un ensemble de fils d'exécution. Il existe aussi plusieurs bibliothèques permettant de créer des serveurs multi-fils dont TPL[76] pour C#.

B.2 L'approche événementielle

L'approche événementielle [36] est une alternative à celle du multi-fils. Selon cette approche, le serveur fonctionne à l'aide d'un seul fil d'exécution pour traiter toutes les requêtes. Pour éliminer les attentes lors d'opérations d'entrées/sorties, l'unique fil utilise des opérations non bloquantes, car du fait qu'il n'y ait aucune concurrence (il n'y a qu'un fil), aucun blocage ne peut advenir dû à la synchronisation ou à des demandes d'entrées/sorties.

Dans ce modèle, l'unique fil d'exécution boucle (*event loop*) sur la réception et le traitement des requêtes (qui arrivent sous la forme d'événements). Lorsqu'une requête survient, il la traite et, lors d'une entrée/sortie, il lance une opération asynchrone non bloquante. Il associe à cette opération une fonction de rappel (callback) et un «événement» en retour qui s'ajoutera, à la fin de l'exécution de l'entrée/sortie, à la file des requêtes à traiter par le serveur. Cette façon de procéder lui permettra de compléter la requête initiale dans le futur. Après le lancement de l'appel asynchrone, le serveur sauve l'environnement de la requête courante et passe à la suivante. L'environnement

ainsi préservé servira à terminer la requête plus tard. Ainsi, dans ce modèle, la file des requêtes à traiter par le serveur contient autant des nouvelles requêtes provenant de clients, que des événements initiés par le serveur pour compléter des requêtes en cours. Ce second type d'événements nécessite soit l'enregistrement de pilotes spéciaux pour les traiter, soit des fonctions de rappel qui leurs sont préalablement associées. La figure B.3 décrit le fonctionnent d'un serveur basé sur l'approche événementielle.

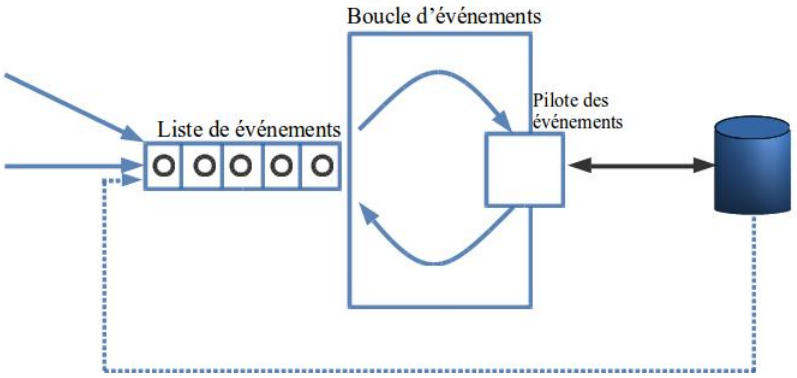


Figure B.3 – Architecture basée sur les événements

Lorsqu'une requête concerne la continuation d'une autre requête, le serveur doit être apte à retrouver l'état de la requête originale afin de la compléter. Les états des requêtes en cours doivent être organisés dans une structure de données appropriée, soit explicitement via une machine à états finis, soit implicitement via le concept de continuation ou de fermeture [1] (*closure* ou *callback*).

Le résultat du fonctionnement de cette approche est le suivant : le flux de contrôle d'une application «dirigé par les événements» se trouve d'une certaine façon inversé. Au lieu d'un flux séquentiel, ce type d'application utilise une cascade d'appels asynchrones et de fonctions de rappel exécutées sur des événements. Le flux de contrôle de l'exécution est alors moins évident à suivre et plus complexe à mettre au point.

L'usage d'une architecture dirigée vers les événements pour les serveurs dépend historiquement de la disponibilité d'opérations d'entrées/sorties asynchrones et non bloquantes au niveau du système d'exploitation. Elle dépend aussi d'une interface de notification d'événements.

1. Pour plus d'informations sur la continuation, voir annexe C.

Entrées/Sorties non bloquantes synchrones ou asynchrones

Les entrées/sorties sont dites synchrones ou asynchrones :

- Entrées/sorties non bloquantes synchrones ;

  Ces opérations retournent, soit le résultat s'il est disponible, soit un code d'erreur. Cela signifie que lorsqu'une opération d'entrée/sortie est lancée, elle retourne immédiatement le contrôle au demandeur, et cela même si l'opération n'est pas terminée. L'entrée/sortie s'exécutera alors en arrière plan et il faudra répéter l'opération autant de fois que nécessaire pour récupérer le résultat. C'est une forme d'attente active.

- Entrées/sorties non bloquantes asynchrones ;

  L'opération d'entrées/sorties retourne immédiatement le contrôle au demandeur sans aucune attente. Lorsque celle-ci termine, le résultat est envoyé à une fonction de rappel. L'entrée/sortie se fait en arrière plan.

Plusieurs systèmes sont basés sur cette approche dont Nginx[3, 31, 43], Node.js[41, 52, 53] et Redis[81]. Tous ces systèmes sont théoriquement à fil unique. Toutefois, ils utilisent tout de même le multi-fils pour profiter des multi-coeurs/multi-processeurs ou pour compenser le manque d'opérations asynchrones. Nginx[3, 31, 43] crée autant de processus travailleurs qu'il y a de processeurs disponibles. Chaque travailleur est orienté événements. Node.js[41, 52, 53] gère aussi un ensemble de fils (*threadpool*) pour exécuter certaines entrées/sorties pour lesquelles les opérations asynchrones ne fonctionnent pas. Node.js met cependant tout en œuvre pour exploiter en permanence les entrées/sorties asynchrones.

B.3 Comparaison entre les approches multi-fils et événementielles

Comment se comparent réellement les deux approches ? L'argument «fils versus événements» est très vieux mais revient régulièrement à la surface. Certains vantent la supériorité d'un modèle par rapport à l'autre [17, 25, 35, 38, 40, 55, 62, 75, 78] et d'autres affirment qu'ils sont équivalents [24, 46]. Dans cette section, nous allons étudier ces points de vue en comparant les deux approches selon plusieurs critères soit la performance, la facilité d'utilisation, la bonne utilisation des ressources et quelques autres que nous abordons dans les sections suivantes.

B.3.1 La performance

On reproche souvent à l'approche multi-fils sa faible performance, en particulier sur sa difficulté à supporter une très lourde charge. En effet, pour une telle charge, une multitude de fils d'exécution est nécessaire pour traiter les requêtes (une fil par requête). Certains de ces fils s'exécutent en parallèle, alors que certains autres sont bloqués (par exemple en attente de la fin d'une entrée/sortie). Si l'on considère des centaines sinon des milliers de fils concurrents, alors une forte consommation de mémoire et maints changements de contexte représentent en effet une charge des plus importantes pour l'UCT. Le problème de consommation de mémoire réside dans la fait que chaque fil requiert une pile. Cette pile est généralement de taille fixe et plutôt volumineuse (quelques mega-octets). Dans un

environnement traitant des milliers de requêtes, cette utilisation de l'espace devient problématique. Les nombreux changements de contexte, eux, sont provoqués par la présence de multiples fils de niveau noyau (*kernel thread*). Chaque fil qui se bloque (sur une entrée/sortie ou sur un verrou), ou qui subit un réquisition d'UCT (fin de sa tranche de temps ou processus de plus haute priorité) provoque un changement de contexte. Chacun de ceux-ci implique un travail de la part de l'UCT, entraîne le déplacement de données en mémoire (jusqu'à 2 mégaoctets) et provoque des défauts de cache pour le nouveau fil qui acquiert le contrôle. De plus, un fil qui se voit retirer le contrôle par réquisition, alors qu'il se situe au milieu de l'exécution d'une tâche critique ou à la limite de terminer une phase d'utilisation de l'UCT (avant un blocage ou la fin de son exécution), entraînera un changement de contexte inutile. Une telle perte de contrôle à un moment critique est une source potentielle d'erreurs. Cette surcharge devient vraiment lourde quand des milliers de fils s'exécutent simultanément. Devant ces difficultés, il est fort probable que cette approche ne soit pas celle à privilégier lorsque la charge de travail devient si lourde (des milliers de requêtes par seconde) [62].

L'approche événementielle, de son côté, consomme moins de mémoire et surcharge moins l'UCT. Il est important de noter que le traitement de chaque requête, dans une application orientée vers les événements, consomme aussi de la mémoire. En effet, ce traitement, divisé en plusieurs étapes, requiert de sauver explicitement l'état d'une requête avant de passer à une autre et cette sauvegarde consomme de la mémoire. Cette consommation de l'espace est toutefois moindre que celle du multi-fils car la taille de la structure de données requise est dynamique et s'adapte à la requête. De plus, le fil étant unique, aucun changement de contexte ne se produit. Notons que même si chaque nouvel événement génère un appel au système d'exploitation, c'est cependant beaucoup moins lourd qu'un changement de contexte. Tout compte fait, ceci devrait rendre cette approche plus efficace sur une seule UCT.

Des contre-arguments sont soulevés face à ces problèmes de performance du multi-fils. Ainsi la surcharge en mémoire, due à l'allocation d'une pile associée à chaque fil, pourrait être réduite en allouant des piles de plus petite taille ou de taille dynamique. De plus, certains affirment [17] que les changements de contexte sont maintenant tellement optimisés, que la surcharge provoquée n'est pas vraiment plus importante que celle due à un appel système destiné à gérer les événements.

Plusieurs auteurs [24] suggèrent de remplacer les fils de niveau noyau par des fils de niveau usager. Avec ce dernier type de fils, la planification devient coopérative (car les changements de contexte sont initiés par les fils eux-mêmes), ce qui élimine les changements de contexte inutiles. De plus, la surcharge induite par les changements de contexte est moindre pour l'UCT car le système d'exploitation n'est pas impliqué dans l'opération. Finalement, la création de fils de niveau usager est moins lourde et l'espace occupé par la pile s'ajuste au besoin. Les «*green threads*», les «*state threads*» et autres bibliothèques optimisées seront privilégiées dans ce cas.

Pour conclure, mentionnons que des recherches ont été réalisées dans le but de fournir des solutions combinant les deux approches. C'est le cas des *threadlet* [18, 51] et des *fibril* [15, 16].

B.3.2 La facilité de programmation

Le multi-fils

Les approches multi-fils et événementielles apportent chacune leur lot de difficultés lorsqu'il s'agit de les programmer.

L'approche multi-fils suit une stratégie simple et facile à implanter. Les différents fils emploient les entrées/sorties synchrones, c'est une façon naturelle d'exprimer les accès. Lorsqu'une entrée/sor-

tie synchrone est initiée par un fil, le contrôle est alloué à un autre fil par changement de contexte [2]. Ceci permet de produire des programmes plus petits, plus lisibles, mieux structurés, plus expressifs et plus simples à comprendre. Les fils sont, en effet, une extension naturelle de la programmation séquentielle et correspondent chacun à une tâche, ce qui permet de recourir aux algorithmes séquentiels existants. Ces derniers sont bien connus, bien compris et représente des outils génériques. Les fils d'exécution offrent une abstraction simple et puissante quand les tâches sont isolées et que le partage est limité.

Malheureusement, la programmation devient complexe avec le multi-fils lorsque des données sont partagées. Il est alors ardu de développer du code parallèle. La coordination et la synchronisation sont généralement des concepts difficiles à maîtriser et ce, autant par les personnes débutantes en programmation que par celles ayant une plus grande expérience/expertise [55]. Des erreurs se multiplient alors souvent sans moyens évidents de les identifier, ni de les corriger. Celles-ci se produisent principalement lors de l'accès à un état global partagé et lors des conditions de course, obligeant l'emploi de verrous (mutex, sémaphores, moniteurs, ...). L'utilisation de ceux-ci est en effet susceptible de provoquer des erreurs tels les interblocages, les interblocages actifs (« *livelocks*) et la famine, d'une part et de limiter les performances (blocage sur les verrous) d'autre part. De plus, l'usage des verrous ne facilite en rien la mise au point des programmes, bien au contraire.

L'obtention d'une bonne performance est parfois laborieuse à réaliser lorsqu'un état est partagé. En effet, choisir la bonne granularité de verrouillage est complexe. Le recours à une granularité trop grande ou trop fine entraîne des conséquences variées et non souhaitables. Ainsi, afin de limiter ou d'éviter les erreurs, un verrouillage relativement grossier est fréquemment choisi, ce qui diminue la concurrence et la performance. En revanche, un verrouillage trop fin augmente la complexité ainsi que les risques d'interblocages, et a le potentiel d'indirectement limiter la performance (changement de contexte et planification).

Finalement, il s'avère parfois risqué de combiner l'usage de plusieurs modules (programmes/bibliothèques) qui font appel à des verrous car, le résultat d'une telle composition n'est pas «sûr», dû notamment aux possibilités de dépendances circulaires.

L'événementiel

L'événementiel a l'avantage de n'impliquer qu'un seul fil pour gérer la boucle d'événements. Le traitement est entièrement réalisé à l'aide des entrées/sorties asynchrones, lesquelles ne génèrent aucune réquisition. En aucun cas, ni pour le gestionnaire ni pour les fonctions de rappel, il n'est nécessaire de se préoccuper des accès concurrents. Cette approche élimine toute la complexité associée à la synchronisation et toutes les erreurs qui lui sont associées.

L'ennui avec l'approche événementielle provient du type de programmation adopté. Elle offre l'illusion du parallélisme en brisant le modèle standard de programmation. Ce *nouveau* modèle présente deux défauts majeurs. Premièrement, selon cette approche, un programme se doit d'être écrit comme une machine à états finis, et le style de programmation qui leur est associé est plus complexe et lourd à saisir. D'ailleurs, dans maintes situations, déterminer une solution adéquate basée sur une telle machine se révèle quasi impossible.

En second lieu, la programmation événementielle amène une inversion du contrôle qui fragmente le code. Cela a pour effet de produire des programmes désordonnés, complexes, fastidieux et non structurés. Le problème vient du fait que c'est la boucle d'événements qui contrôle, et non pas le personne en charge du développement. Ainsi, lors d'une entrée/sortie, on ne décider de suspendre

2. Évidemment, seulement si le système d'exploitation fonctionne avec la réquisition.

l'exécution de l'initiateur afin d'attendre le résultat. Il faut absolument lancer une entrée/sortie et retourner le contrôle à la boucle d'événements. Lorsque les données sont disponibles, la boucle d'événements appellera la routine de complétion de l'entrée/sortie. Tout ceci force le découpage d'un programme simple en multiples «morceaux» où chacun d'eux constitue une fonction de rappel. Ce découpage s'effectue à chaque instruction d'entrée/sortie. La logique d'une tâche est donc divisée en plusieurs événements et fonctions de rappel. Le flux de contrôle est alors centré sur le retour à la boucle et sur l'attente pour l'exécution de la fonction de rappel. En ce sens, cette façon de faire amène un chaînage de fonctions, soit une forme de GOTO avec délai. En fait, on se doit alors de renoncer aux outils de base de la programmation structurée pour retourner le contrôle à la boucle.

Conclusion

En conclusion, le niveau d'abstraction est moins élevé avec l'approche événementielle et le programme résultant est plus complexe. Les algorithmes séquentiels existants sont inutilisables dans les systèmes événementiels. En terme de flux de contrôle, un programme dirigé par les événements ressemblent à un programme écrit en langage d'assemblage. Heureusement, plusieurs outils, patrons de conception (tel *reactor*[1, 56, 61, 63]) et langages de programmation (tel Javascript) ont émergé pour supporter la programmation événementielle. Ainsi un langage intégrant la fermeture et la continuation, contribuera largement à réduire la complexité de programmation associée à l'événementiel. Si le langage ne le prévoit pas (tel le langage C), alors la manipulation se fera manuellement dans l'architecture événementielle.

B.3.3 Le bon usage des ressources

Quand on réfère à une «bonne utilisation» des ressources, cela concerne principalement l'usage des multiples cœurs ou processeurs contenus dans un serveur. Le multi-fils exploite et profite naturellement des multiples cœurs et processeurs. Dans l'approche multi-fils, on associe chacune des connexions à un fil et les fils, eux, s'exécutent sur n'importe lequel des cœurs ou processeurs. L'emploi des capacités du matériel est donc maximisé.

On reproche souvent à l'approche événementielle de ne pouvoir profiter naturellement des multiples cœurs ou processeurs présents sur les architectures modernes [55]. La plupart des applications orientées événements ont cependant mis au point différentes techniques afin d'exploiter au mieux ces ressources. Parmi celles-ci, mentionnons ces deux possibilités :

- configurer plusieurs serveurs logiciels qui partageront le même port de communication («*socket*») sur la même machine (N-copy) ;
- démarrer plusieurs fils d'exécution qui traiteront en parallèle des fonctions de rappel indépendantes.

Ainsi, comme nous l'avons déjà mentionné, Nginx et Node.js offrent des solutions pour bénéficier des multiples cœurs. Malgré leur apparente simplicité, ces solutions sont perçues, par bon nombre, comme supérieures à celles basées sur l'approche multi-fils, puisqu'elles évitent la création d'un trop grand nombre de fils avec les conséquences négatives que cela entraîne.

B.3.4 Les blocages inévitables et les requêtes orientées UCT

Parmi toutes les requêtes reçues par un serveur, il est possible que certaines d'entre elles soient orientées UCT ou que certaines opérations soient obligatoirement bloquantes. L'approche multi-fils

offre une performance correcte et est appropriée pour traiter ces situations.

Toutefois, ce type de requêtes s'avère problématique pour l'approche événementielle. En effet, celle-ci, pour bien fonctionner, nécessite que chaque requête consomme très peu de temps UCT et évite toute forme de blocage. Ainsi, l'exécution d'une requête exigeante en UCT compromet le traitement de nouvelles requêtes. De même, le traitement d'une opération bloquante (sauf pour une l'attente d'une nouvelle requête) provoque une attente indue pour les nouveaux événements et nuit à la performance [17, 25]. L'idéal, soit aucun blocage ni exécution d'une opération à forte consommation d'UCT pour l'unique fil, se révèle souvent utopique. Certaines solutions sont envisageables, mais elles exigent d'avoir recours à de multiples fils pour réduire l'impact de telles requêtes, tel Node.js.

B.3.5 Le non déterminisme et la facilité de mise au point

Rappelons que le non-déterminisme rend le comportement d'un programme difficile à comprendre et à prédire. Il est donc plus que laborieux de faire la mise au point d'un tel programme. Le non-déterminisme est provoqué par le parallélisme et une planification basée sur la réquisition. L'exécution dans un environnement multi-fils est non-déterministe. L'exécution des fonctions de rappel est, quant à elle, déterministe (s'il n'y a pas de *yield*). Dans ce dernier cas, la planification est explicite et se produit dans l'application (et non dans le système d'exploitation). Il n'y a alors aucun changement de contexte inutile, ni pendant l'exécution d'une opération critique. On a même le loisir d'optimiser la planification pour tenir compte des besoins spécifiques de l'application.

Répétons cependant que cet inconvénient, associé au multi-fils, N'existe plus si l'utilisation de fils de niveau usager est privilégiée.

B.3.6 Portabilité

À l'époque des premières critiques envers l'approche basée multi-fils en 1996 [55], ce concept était mal supporté par plusieurs systèmes. La transportabilité du code multi-fils s'avérait ardu car, le support variait énormément d'un environnement à l'autre et plusieurs bibliothèques ne fournissaient pas des fonctionnalités «sécures vis-à-vis le multi-fils» (`Thread-safe`). Aujourd'hui, le support des fils d'exécution est généralisé et plutôt uniforme (Posix ou intégré dans des langages portables tels C++, Java, ...).

En revanche le support des opérations asynchrones a toujours été présent dans la plupart des environnements (même si l'uniformité est quant à elle généralement absente). Cela fournissait une meilleure portabilité aux applications basées sur les événements [55].

B.3.7 La preuve de bon fonctionnement du programme

L'explosion du nombre d'états dans les programmes multi-fils complexifie l'analyse de l'exécution. Puisque les programmes orientés événements suivent un mode d'exécution basé sur les machines à états, leur analyse en semble simplifiée.

B.3.8 Les approches multi-fils et événementielles sont équivalentes

Erb [24] affirme que les deux modèles sont équivalents et réfèrent à la comparaison faite par Lauer et Needham [46] en 1979. À l'époque, Lauer et Needham étudièrent les deux modèles de

programmation présents dans les systèmes d'exploitation (passage de messages vs appel de procédures) et conclurent à leur équivalence. L'architecture orientée vers le passage de messages est basé sur un faible nombre de processus communiquant par messages. L'architecture orientée vers l'appel de procédures, elle, suppose la présence d'un plus grand nombre de processus se partageant des données. Erb [24] considère que cette comparaison s'applique toujours à la situation actuelle car le modèle orienté vers le passage de messages correspondrait au modèle événementiel alors que celui orienté vers les appels de procédures correspondrait quant à lui au modèle multi-fils.

La comparaison des deux modèles conduit à ces constats :

- ces deux modèles sont logiquement équivalents : il est donc possible de convertir (mapped) un programme écrit selon un modèle en un programme équivalent dans l'autre modèle.
- les performances devraient être équivalentes si les stratégies de planification le sont également.

Erb conclut en mentionnant d'éventuelles causes aux difficultés précédemment énoncées, soit une mauvaise implantation de la bibliothèque qui gère les fils, soit l'essence même de la planification avec réquisition.

B.4 Conclusion

Que conclure ? Doit-on coûte que coûte déclarer l'une ou l'autre «meilleure approche» ? Ou encore affirmer que l'une est «mauvaise» et l'autre «bonne» ? Après tout, chacune de ces approches, soit le multi-fils ou l'événementielle, possède «ses» défauts et «ses» bons coups. Résumons-nous.

Les fils entraînent certains problèmes au niveau de la programmation, due à la synchronisation. Ils requièrent aussi un support uniforme du multi-fils, et ce, de tous les modules (bibliothèques) utilisés («thread-safe»). Ils n'atteignent pas toujours le niveau de performance escompté (à cause de la granularité de la synchronisation et des changements de contexte) et ne conviennent pas à certains types d'application (les interfaces usagers graphiques). Toutefois, les fils se révèlent des plus appropriés en ce qui concernent les applications exigeant des temps de traitement significatifs et nécessitant l'usage de plusieurs cœurs ou processeurs.

L'approche événementielle a aussi son lot d'inconvénients et le plus important est certainement, dû à son modèle de programmation, qui nous ramène aux «bons» vieux «GOTO». D'autres irritants de cette méthode, la gestion des contextes et la gestion des exceptions, s'avèrent souvent complexes. De plus, elle exploite moins bien les capacités multi-cœurs ou multi-processeurs des ordinateurs modernes. Du côté avantage, l'approche événementielle se distingue pour les environnements nécessitant une grande quantité d'entrées/sorties asynchrones, telles les interfaces usagers graphiques.

Côté performance, on admet que les deux approches atteignent des niveaux comparables. En effet, le multi-tâches coopératif (fils de niveau usager ou co-routines) apparaît adéquat pour les deux approches. En ce qui concerne la pile, il est aussi possible d'éviter les problèmes de gestion engendrés par l'événementiel autant que les conséquences d'un usage excessif de la mémoire engendré par le multi-fils.

Finalement, en ce qui concerne le flux de contrôle de l'exécution, chacune des approches est à même de le rendre obscur et de complexifier la mise au point. Ce n'est pas l'apanage d'une seule méthode.

En terminant, un fait est certain, certaines applications s'adaptent mieux à l'une des deux approches. Leurs forces et leurs faiblesses se doivent d'être reconnues afin de les utiliser à bon

escient. En fait, rien n'empêche ces deux approches de se côtoyer si cela est requis. À nous de faire preuve de jugement dans nos choix pour choisir judicieusement ces remarquables outils.

Annexe C

Continuation

C.1 Citoyen de première classe (First class citizen)

Dans les langages de programmation, on fait souvent référence, concernant les «objets» supportés par ceux-ci, à leur «niveau de citoyenneté» [98]. Il existe en effet différents «niveaux de citoyenneté» qui déterminent les propriétés des objets (ou les actions qu'il est possible d'appliquer sur eux).

Mentionnons dans un premier temps les principales actions qui s'appliquent à un objet. Chacun peut :

- être assigné à une variable ;
- être «passé» en paramètre à une fonction ;
- être transféré en valeur de retour d'une fonction (*return value*).

Dans certains cas, pour obtenir le niveau de citoyenneté le plus élevé, le langage doit aussi fournir la capacité de créer dynamiquement de nouveaux objets.

Un objet d'un langage sur lequel s'appliquent ces trois (ou quatre) opérations est considéré «citoyen de première classe» (*first class citizen*). Donc, des fonctions sont des citoyennes de première classe, s'il est possible de les assigner à une variable, de les «passer» en paramètres à une fonction et de les transmettre en valeur de retour d'une fonction. Dans quelques cas, il est aussi possible de créer dynamiquement (à l'exécution) de nouvelles fonctions (support pour les fonctions littérales ou anonymes).

Ainsi dans un langage supportant les fonctions comme citoyennes de première classe, celles-ci peuvent être manipulées comme des variables ordinaires. Le concept de fonctions *lambda* que l'on retrouve dans les langages fonctionnels et autres langages modernes non-fonctionnels en est l'exemple le plus courant.

Parfois, on emploie l'expression «citoyenne de seconde classe» pour qualifier une fonction qui ne peut qu'être «passée» en paramètre. Finalement, une fonction dite «citoyenne de troisième classe» en est une sur laquelle on ne peut effectuer aucune des opérations que nous venons de présenter.

La terminologie «fonction d'ordre supérieur» (*high order function*) désigne quant à elle, celle qui a la possibilité de recevoir une fonction en paramètre.

C.2 Continuation et fermeture (closure)

Dans un langage de programmation, une fermeture [97, 95, 72] (*closure*) définit un objet constitué d'une fonction et de son environnement (ou contexte) lexical de référence. Ce dernier est l'ensemble des variables non locales à la fonction incluses dans l'environnement de référence. Les variables de l'environnement sont capturées, soit par valeur, soit par référence.

Le concept de fermeture est particulièrement intéressant quand une fonction, F_1, est définie à l'intérieur d'une autre fonction, F_2. Les variables de cette dernière (F_2) sont incluses dans le contexte lexical de F_1.

De façon générale, une fermeture désigne un objet qui sera éventuellement transmis en argument d'une fonction. Les fermetures apparaissent donc dans les langages où les fonctions sont des citoyennes de première classe. Wikipedia [97, 95] présente maints exemples de langages modernes fournissant le concept de fermeture.

C.2.1 Exemples

En langage Python, les fonctions sont d'ordre supérieur et «citoyennes de première classe». De plus, il y est possible de créer des fonctions dynamiquement (fonctions *lambda* ou anonymes). Le programme C.1 fournit un exemple de manipulation de fonctions et de fermetures dans ce langage. Soit quatre fonctions :

- $f(x)$ et $h(x)$: ces deux fonctions reçoivent un paramètre x, définissent une fonction et la retourne.
- $g(y)$: cette fonction, définie à l'intérieur de la fonction f, retourne la somme des variables x et y. La variable y est reçue en paramètre, tandis que x est connue puisque déclarée par f (elle fait donc partie de la portée de g).
- La fonction *lambda* : la fonction *lambda*, une fonction anonyme définie dynamiquement à l'intérieur de la fonction h, implante la même fonctionnalité que g.

Quelques explications sur le code du programme :

- Aux lignes 1 à 5, on constate que les fonctions f et h renvoient des fonctions en «valeur de retour».
- À la ligne 7, on assigne la fermeture de la fonction g (le retour de f) à une variable. Il est question de fermeture car on retourne la fonction g et son environnement lexical. En effet, cela est nécessaire car pour exécuter g nous devons connaître toutes les variables qu'elle utilise (x et y). La variable y est connue car elle est définie dans g. Toutefois c'est différent dans le cas de la variable x. L'environnement lexical fournira l'information requise sur x.
- À la ligne 8, on assigne la fermeture de la fonction anonyme (*lambda*) retournée par h à la variable b.
- Aux lignes 9 et 10, on procède aux «appels» des variables a et b. Ces appels exécuteront les fonctions assignées à ces variables et afficheront les résultats, soit 3 et 4 respectivement.
- Aux lignes 14 et 15, on procède aux «appels» des fermetures anonymes des lignes 12 et 13, qui produiront respectivement 6 et 10.

```
1  def f ( x ):
2        def g ( y ): return x + y
3        return g
4  def h ( x ):
5        return lambda y : x + y
6
7  a = f(1)   # a est une variable contenant une fermeture
8  b = h(1)   # b est une variable contenant une fermeture
9  print(a(2))
10 print(b(3))
11
12 f(1)(5)    # fermeture "anonyme"
13 h(5)(5)    # fermeture "anonyme"
14 print (f (1)(5))
15 print (h (5)(5))
```

Programme C.1 – Exmeples de fermeture en Python.

Provenance du terme «fermeture»

D'où vient le terme «fermeture» ou pourquoi l'utilise-t-on pour identifier un objet incluant une fonction et son environnement lexical ?

Considérons le lien entre les fonctions et les expressions *lambda*. Soit :

- Fonction nommée : c'est une fonction régulière. On emploie son nom pour l'appeler.
- Fonction anonyme : c'est une fonction qui ne porte pas de nom. Elle est définie par une expression lambda (λ).

Les expressions λ (ou termes λ) proviennent du λ-calcul [86, 99], soit le langage de programmation considéré comme le plus simple jamais écrit. En effet, il ne fait qu'appliquer une expression à une autre expression disons f à x, noté $f\ x$ (une fonction f avec un paramètre x). De plus, selon le λ-calcul, «tout» est une fonction et toutes les fonctions sont obligatoirement anonymes et n'acceptent qu'un seul paramètre (x). Il est donc du domaine du possible que x soit aussi une fonction.

Ainsi dans le λ-calcul, on retrouve les expressions λ suivantes :

- une variable x ;
  En λ-calcul, une fonction non encore définie, contenue dans un terme λ, est remplacée par une variable.
- si f et x sont des expressions, alors ($f\ x$) est une expression appelée une *application*.
  Une application représente l'action d'appeler une fonction pour produire $f(x)$.
- si f est une expression et x une variable, alors $\lambda x.f$ est une expression appelée une *abstraction* ;
  Pour que le terme $f\ x$ soit convertit en fonction f qui prend un paramètre x, on doit établir une correspondance entre un symbole, ici x, et un contenant auquel on peut substituer une valeur (un paramètre). On effectue la correspondance par l'intermédiaire de la lettre λ. On écrit la lettre λ suivi du symbole, puis d'un "." juste avant l'expression. Cela a pour conséquence de convertir celle-ci en une fonction qui prend un seul paramètre.

Abstraction λ

Une abstraction $\lambda x.f$ définie une fonction anonyme qui accepte un seul paramètre x en entrée, effectue la substitution dans la fonction f et retourne f. L'abstraction (ou le λ) sert à lier la variable x au terme f.

Pour qu'une expression soit valide, elle doit nécessairement être obtenue par l'usage répété de ces trois règles (récursivement).

Soit l'abstraction :

$$\lambda x.x + 2.$$

Celle-ci lie la variable x à l'expression $x + 2$. On dit alors de x qu'elle est une variable «fermée» («*bounded*»), i.e. que l'on peut lui substituer une valeur passée en paramètre.

Puisqu'une fonction λ ne possède pas de nom, on ne pourra pas y référer ultérieurement, il faut donc y faire appel directement en lui fournissant une valeur de paramètre. Ainsi, «$(\lambda x.x + 2)\ 7$» devient 9. Cette expression définit une fonction f contenant un symbole fermé, le paramètre x, auquel on assigne une valeur (7).

En λ-calcul, le concept de déclaration de variable n'existe pas. Soit le cas suivant :

$$\lambda x.x/y + 2 \quad (ou f(x) = x/y + 2).$$

Ici, x est «fermée» mais la variable y, elle, ne l'est pas. Le λ-calcul traite y comme une variable «libre» (ou «ouverte»), i.e. qu'elle n'est pas encore définie (inconnue). Cela signifie qu'on ne sait pas ce qu'elle est, ni d'où elle vient. Il est donc impossible d'évaluer l'expression tant que l'on ne connaît pas la signification et la valeur de y [1].

Il existe donc deux types d'expressions λ :

- celles dites «fermées» : chaque symbole est fermé par une expression λ.
- celles dites «ouvertes» : certains symboles sont libres. Il est nécessaire d'obtenir des informations externes pour compléter et évaluer l'expression.

Les symboles «libres ou ouverts» doivent donc être définis à l'extérieur de la fonction soit dans l'environnement d'exécution de la fonction, appelé «environnement ou contexte lexical». Ce contexte provient généralement du langage, d'une bibliothèque ou de tout autre élément de l'environnement.

Il est possible de «fermer» une expression λ en fournissant le contexte lexical qui définit les symboles libres et leur assigne une valeur. Cela transforme une expression λ ouverte en une expression λ fermée, d'où le terme fermeture.

Ainsi $\lambda x.x/y + 2$ est une expression ouverte. Si on lui fournit l'environnement suivant $[y : 3, + : [op], 2 : [nbr], q : 42, w : 45]$, elle devient fermée et peut désormais être évaluée.

La fermeture, comprenant la fonction et son contexte lexical, s'applique sur une fonction nommée ou anonyme.

Continuation

La continuation [79] d'un programme est la suite des instructions qu'il lui reste à exécuter à un moment précis. Elle crée une structure de données qui représente l'état du programme à un point particulier de son exécution et le transforme en une variable ré-utilisable et accessible par celui-ci. Une continuation est donc une fermeture ou plutôt elle s'implante grâce à une fermeture car elle

1. En fait, les symboles 2 et + sont aussi libres. Toutefois ils ont une signification connue de tous (on doit cependant les définir pour l'ordinateur)

a besoin de connaître l'état courant du programme et son contexte lexical afin de poursuivre le traitement.

La «continuation courante» est celle dérivée de la position courante de l'exécution. Elle servira, dans le futur, à reprendre l'exécution d'une fonction à partir de ce point. Les opérations *resume* (pour les co-routines) et *yield* (pour les fils d'exécution) créent des continuations courantes.

Le terme continuation est aussi employé pour référer à la «continuation de première classe» (*first class continuation*) [80] qui est une construction donnant l'habileté à un langage de programmation de sauver son état d'exécution à tout moment et de retourner à ce point plus tard et cela à plusieurs reprises. La continuation de première classe permet à un langage de contrôler l'ordre d'exécution de ses instructions. On sauve l'état et on y retourne plus tard. Cela ne sauve pas les données, seulement le contexte d'exécution.

Il existe un paradigme de programmation dérivé de ce concept appelé «continuation passing style» [79]. Avec ce paradigme, un fonction reçoit un paramètre supplémentaire qui est une continuation explicite appelée «fonction de continuation». Ainsi, chaque fonction reçoit une autre fonction en paramètre qui représente l'excédent du calcul à traiter par rapport à l'appelant. Quand une fonction termine le traitement demandé, elle retourne le résultat en appelant la fonction de continuation avec la valeur du résultat en paramètre. Dans ce style de programmation, l'appelant doit toujours, au moment de l'appel, fournir la fonction de continuation. Il faut donc abandonner l'idée qu'une fonction doit nécessairement retourner le contrôle à celle qui l'a appelée.

Le principal inconvénient de la continuation est sa forte ressemblance, ou équivalence fonctionnelle, aux `GOTO`. C'est un `GOTO` glorifié. Leur utilisation est susceptible d'alourdir significativement la lecture et la compréhension du code.

On recourt aux continuations afin d'encoder des mécanismes de contrôle tels que les exceptions, les co-routines, les itérateurs, les générateurs, les fils de niveau usager (green threads), la programmation événementielle... On retrouve aussi ce principe dans les instructions `setjmp/longjmp` du langage C et dans quelques autres langages de programmation. Certains langages supportent la continuation d'une façon restrictive, i.e qu'elle est restreinte à une continuation de retour («return continuation») imposant des limites sur les paramètres que l'on peut passer. D'autres langages supportent la «continuation générale» dans laquelle la fonction peut accepter sans condition tout argument, comme toute fonction. La continuation est un concept qui devient particulièrement populaire dans les langages de programmation pour le WEB.

En voici un exemple :

```
fontion F1 (valeur, f2)
{
   // calcul du résultat.....

   f2(résultat)
}
```

Pour d'autres exemples de langages supportant la continuation, référez-vous à Wikipedia [80, 79].

Bibliographie

[1] Chris Adamson : Architecture of a highly scalable nio-based server blog. https://community.oracle.com/docs/DOC-983601, 2015.

[2] Abhinav Ajitsaria : What is the python global interpreter lock (gil) ? https ://realpython.com/python-gil/, 2021.

[3] Andrew Alexeev : Nginx. http://www.aosabook.org/en/nginx.html, 2012.

[4] Developpers Android : Kotlin coroutines on android. https://developer.android.com/kotlin/coroutines?gclid=Cj0KCQiA6NOPBhCPARIsAHAy2zAEnoMJNPuuqRYQ2HOdX9UWWPLGVu5wl8ZSYgYAbopsUpDUIsCtp-0aAj2XEALw_wcB&gclsrc=aw.ds, 2020.

[5] Apache : Apache mpm event. https://httpd.apache.org/docs/2.4/fr/mod/event.html, 2018.

[6] Apache : Apache mpm prefork. https://httpd.apache.org/docs/2.4/fr/mod/prefork.html, 2018.

[7] Apache : Apache mpm worker. https://httpd.apache.org/docs/2.4/fr/mod/worker.html, 2018.

[8] Blaise Barney et Donald Frederick : Introduction to parallel computing tutorial. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##MemoryArch, 2022.

[9] Boinc : Calculez pour la science. https://boinc.berkeley.edu/, 2022.

[10] Mike Burrel : An introduction to sparc's simd offerings. https://mikeburrell.wordpress.com/2007/12/14/an-introduction-to-sparcs-simd-offerings/, 2007.

[11] Dan Burton : Part 2 : Coroutines. https://www.schoolofhaskell.com/user/DanBurton/coroutines-for-streaming/part-2-coroutines, 2020.

[12] Sumouli ChoudHary : Concurrency vs parallelism. https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_concurrency_vs_parallelism.htm, 2017.

[13] Melvin E. Conway : A multiprocessor system design. *In Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), page 139–146, New York, NY, USA, 1963. Association for Computing Machinery.

[14] Harold Cooper : Coroutine event loops in javascript. `https://x.st/javascript-coroutines/`, 2012.

[15] Jonathan Corbet : Fibrils and asynchronous system calls. `https://lwn.net/Articles/219954/`, 2007.

[16] Jonathan Corbet : Kernel space : fibrils and asynchronous system calls. `https://www.networkworld.com/article/2294838/software/kernel-space--fibrils-and-asynchronous-system-calls.html`, 2007.

[17] Jonathan Corbet : Thread-based or event-based ? `https://lwn.net/Articles/223980/`, 2007.

[18] Jonathan Corbet : Threadlets. `https://lwn.net/Articles/223899/`, 2007.

[19] CPPreference : Coroutines (c++20). `https://en.cppreference.com/w/cpp/language/coroutines`, 2021.

[20] Jack B. Dennis et Earl C. Van Horn : Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, mars 1966.

[21] Python Docs : Coroutines and tasks. `https://docs.python.org/3/library/asyncio-task.html`, 2021.

[22] Chapel Documentation : cobegin statements : creating groups of tasks. `https://chapel-lang.org/docs/1.14/users-guide/taskpar/cobegin.html`, 2016.

[23] Joe Doherty : What exactly is a pre-fork web server model ? `https://stackoverflow.com/questions/25834333/what-exactly-is-a-pre-fork-web-server-model`, 2014.

[24] Benjamin Erb : Concurrent programming for scalable web architectures. Diploma thesis, Institute of Distributed Systems, Ulm University, April 2012.

[25] Erginduran : What are the differences between event-driven and thread-based server system ? `https://stackoverflow.com/questions/25280207/what-are-the-differences-between-event-driven-and-thread-based-server-system`, 2014.

[26] Adrian Mouat et al. : What is the difference between concurrency and parallelism ? `https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism`, 2011.

[27] StackExchange et al. : Difference between parallel and concurrent programming ? `https://cs.stackexchange.com/questions/19987/difference-between-parallel-and-concurrent-programming`, 2014.

[28] Vicky Katara et al. : What is the difference between concurrency and parallelism ? `https://www.quora.com/What-is-the-difference-between-concurrency-and-parallelism`, 2017.

[29] Michael J. Flynn : Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, septembre 1972.

[30] Deepshi Garg : Parallelism is not concurrency. https://medium.com/@deepshig/concurrency-vs-parallelism-4a99abe9efb8, 2017.

[31] Owen garret : Inside nginx : How we designed for performance & scale. https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/, 2015.

[32] GeeksForGeeks : Coroutines in c/c++. https://www.geeksforgeeks.org/coroutines-in-c-cpp/, 2021.

[33] GeeksForGeeks : Coroutines in python. https://www.geeksforgeeks.org/coroutine-in-python/, 2022.

[34] Lokesh Gupta : Concurrency vs parallelism. https://howtodoinjava.com/java/multi-threading/concurrency-vs-parallelism/, 2018.

[35] Andreas Gustafsson : Threads without the pain. *Queue*, 3(9):40 :34–40 :41, novembre 2005.

[36] Adron Hall : Understanding the node.js event loop. https://strongloop.com/strongblog/node-js-event-loop/, 2013.

[37] Robert Harper : Parallelism is not concurrency. https://existentialtype.wordpress.com/2011/03/17/parallelism-is-not-concurrency/, 2011.

[38] Stuart harris : Tasks and task parallel library (tpl) : Multi-threading made easy. https://www.codeproject.com/Articles/1083787/Tasks-and-Task-Parallel-Library-TPL-Multi-threadin, 2016.

[39] Martin Heller : What is cuda ? parallel programming for gpus. *InfoWorld*, August 2018 [Online]. doi :https ://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html.

[40] Petter Häggholm : Node.js : How is an event based model more efficient than thread based model for serving http requests ? https://www.quora.com/Node-js\-How\-is\-an\-event-based-model-more-efficient-than-thread-based-model-for-serving-http-requests, 2015.

[41] Deepal Jayasekara : Event loop and the big picture - nodejs event loop part 1. https://jsblog.insiderattack.net/event-loop-and-the-big-picture-nodejs-event-loop-part-1-1cb67a182810, 2017.

[42] Jakob Jenkov : Concurrency vs. parallelism. http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html, 2015.

[43] Daniel Khan : What you should know to really understand the node.js event loop. https://medium.com/the-node-js-collection/what-you-should-know-to-really-understand-the-node-js-event-loop-and-its-metrics-c4907b19da4c, 2017.

[44] Acervo Kima : Algol 68. https://wiki.acervolima.com/algol-68/, 2020.

[45] Kotlin : Coroutines. https://kotlinlang.org/docs/coroutines-overview.html, 2021.

[46] Hugh C. Lauer et Roger M. Needham : On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, 1979.

[47] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Robert Scheifler et William Weihl : Argus reference manual. https://www.researchgate.net/publication/2600187_Argus_Reference_Manual, 11 1987.

[48] Unity Manual : Coroutines. https://docs.unity3d.com/Manual/Coroutines.html, 2020.

[49] David Mazières : My tutorial and take on c++20 coroutines. https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html, 2021.

[50] Sun Microsystems : The vistm instruction set - a white paper. https://0x04.net/~mwk/doc/sparc/vis_whitepaper.pdf, 2002.

[51] Ingo Molnar : Syslets, threadlets, generic aio support. https://lwn.net/Articles/224241/, 2007.

[52] NodeJs : Node.js tutorial. https://https://www.tutorialspoint.com/nodejs/index.htm, 2018.

[53] Trevor Norris : Understanding the node.js event loop. https://nodesource.com/blog/understanding-the-nodejs-event-loop/, 2015.

[54] Andrew Nosenko : Asynchronous coroutines with c# and iasyncenumerable. https://dev.to/noseratio/asynchronous-coroutines-with-c-8-0-and-iasyncenumerable-2e04, 2020.

[55] John Ousterhout : Why threads are a bad idea (for most purposes). *In Usenix Winter technical Conference*, 1996.

[56] Tian Pan : Understanding reactor pattern for highly scalable i/o bound web server. https://www.puncsky.com/blog/2015-01-13-understanding-reactor-pattern-for-highly-scalable-i-o-bound-web-server, 2015.

[57] James L. Peterson et Abraham. Silberschatz : *Operating system concepts / James L. Peterson, Abraham Silberschatz.* Addison-Wesley Pub. Co Reading, Mass, 1983.

[58] Rob Pike : Concurrency is not parallelism. https://vimeo.com/49718712, 2014.

[59] Rob Pike : Concurrency is not parallelism. https://talks.golang.org/2012/waza.slide#1, 2014.

[60] Tutorial Points : Concurrency vs parallelism. https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_concurrency_vs_parallelism.htm, 2018.

[61] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt et Thomas D. Jordan : Proactor - an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events, 1997.

[62] Isacc Roth : What makes node.js faster than java ? https://strongloop.com/strongblog/node-js-is-faster-than-java/, 2014.

[63] Douglas C. Schmidt : Reactor : An object behavioral pattern for concurrent event demultiplexing and dispatching, 1995.

[64] Boris Schäling : *The boost C++ libraries (Chapter 51).* XML Press, 2014. https://theboostcpplibraries.com/boost.coroutine(visited2022-01-23).

[65] Terry A. Scott et R. A. McBride : A programmer's guide to the edison language. *SIGSMALL/PC Notes*, 16(1):14–23, feb 1990.

[66] SETI@home : Seti@home. https://setiathome.berkeley.edu/, 2022.

[67] Abraham Silberschatz : Priority and queuing specification in 'distributed processes'. *The Computer Journal*, 25(1):34–36, 1982.

[68] Abraham Silberschatz, Greg Gagne et Peter Baer Galvin : *Operating System Concepts.* Wiley, 6 édition, 2002.

[69] Abraham Silberschatz, Peter Baer Galvin et Greg Gagne : *Operating System Concepts, 10th Edition.* Wiley, 2018.

[70] Skymen : [js] making use of coroutines. https://www.construct.net/en/tutorials/js-making-coroutines-2255, 2019.

[71] StackOverflow : What is the difference between asymmetric and symmetric coroutines ? https://stackoverflow.com/questions/41891989/what-is-the-difference-between-asymmetric-and-symmetric-coroutines, 2020.

[72] Stringfixer : Fermeture (programmation informatique). https://stringfixer.com/fr/Closure_(computer_science), 2022.

[73] Dávid Szakállas : Concurrency and parallelism : Understanding i/o. https://blog.risingstack.com/concurrency-and-parallelism-understanding-i-o/, 2016.

[74] techDifferences : Difference between concurrency and parallelism. https://techdifferences.com/difference-between-concurrency-and-parallelism.html, 2017.

[75] Hendry Ten : What so different about node.js's event-driven ? can't we do that in asp.net's httpasynchandler ? https://stackoverflow.com/questions/5599024/what-so-different-about-node-jss-event-driven-cant-we-do-that-in-asp-nets-ht, 2011.

[76] Rasik Bihari Tiwari : Thread-based or event-based ? https://blog.red-badger.com/blog/2013/01/29/thread-based-or-event-based, 2013.

[77] M. Tremblay, J.M. O'Connor, V. Narayanan et Liang He : Vis speeds new media processing. *IEEE Micro*, 16(4):10–20, 1996.

[78] Rob von Behren, Jeremy Condit et Eric Brewer : Why events are a bad idea (for high-concurrency servers). *In Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[79] Wikipedia : Continuation. https://en.wikipedia.org/wiki/Continuation, 2018.

[80] Wikipedia : Continuation-passing style. https://en.wikipedia.org/wiki/Continuation-passing_style, 2018.

[81] Wikipedia : Redis. https://fr.wikipedia.org/wiki/Redis, 2018.

[82] Wikipedia : Coroutine. https://en.wikipedia.org/wiki/Coroutine, 2020.

[83] Wikipedia : Visual instruction set. https://en.wikipedia.org/wiki/Visual_Instruction_Set, 2020.

[84] Wikipedia : Altivec. https://fr.wikipedia.org/wiki/AltiVec, 2021.

[85] Wikipedia : Architecture dataflow. https://fr.wikipedia.org/wiki/Architecture_Dataflow, 2021.

[86] Wikipedia : Closure (computer science). https://fr.wikipedia.org/wiki/Lambda-calcul, 2021.

[87] Wikipedia : Communicating sequential processes. https://en.wikipedia.org/wiki/Communicating_sequential_processes, 2021.

[88] Wikipedia : Dataflow architecture. https://en.wikipedia.org/wiki/Dataflow_architecture, 2021.

[89] Wikipedia : Jeu d'instructions mmx. https://fr.wikipedia.org/wiki/Jeu_d%27instructions_MMX, 2021.

[90] Wikipedia : Mmx (instruction set). https://en.wikipedia.org/wiki/MMX_(instruction_set), 2021.

[91] Wikipedia : Radeon. https://fr.wikipedia.org/wiki/Radeon, 2021.

[92] Wikipedia : Streaming simd extensions. https://fr.wikipedia.org/wiki/Streaming_SIMD_Extensions, 2021.

[93] Wikipedia : Advanced vector extensions. https://en.wikipedia.org/wiki/Advanced_Vector_Extensions, 2022.

[94] Wikipedia : Algol 68. https://en.wikipedia.org/wiki/ALGOL_68, 2022.

[95] Wikipedia : Closure (computer science). https://en.wikipedia.org/wiki/Closure_(computer_programming), 2022.

[96] Wikipedia : Compute unified device architecture. https://fr.wikipedia.org/wiki/Compute_Unified_Device_Architecture, 2022.

[97] Wikipedia : Fermeture (informatique). https://fr.wikipedia.org/wiki/Fermeture_(informatique), 2022.

[98] Wikipedia : First class citizen. https://en.wikipedia.org/wiki/First-class_citizen, 2022.

[99] Wikipedia : Lambda calculus. https://en.wikipedia.org/wiki/Lambda_calculus, 2022.

[100] Ashley Yakeley : Coroutines in io. https://www.reddit.com/r/haskell/comments/autlww/coroutines_in_io/, 2018.