Piles multiples d'exécution

MU4IN501 – DLP : Développement d'un langage de programmation
Master STL, Sorbonne Université

Antoine Miné

Année 2020–2021

Cours 7 bis

(préparation au TME 7)

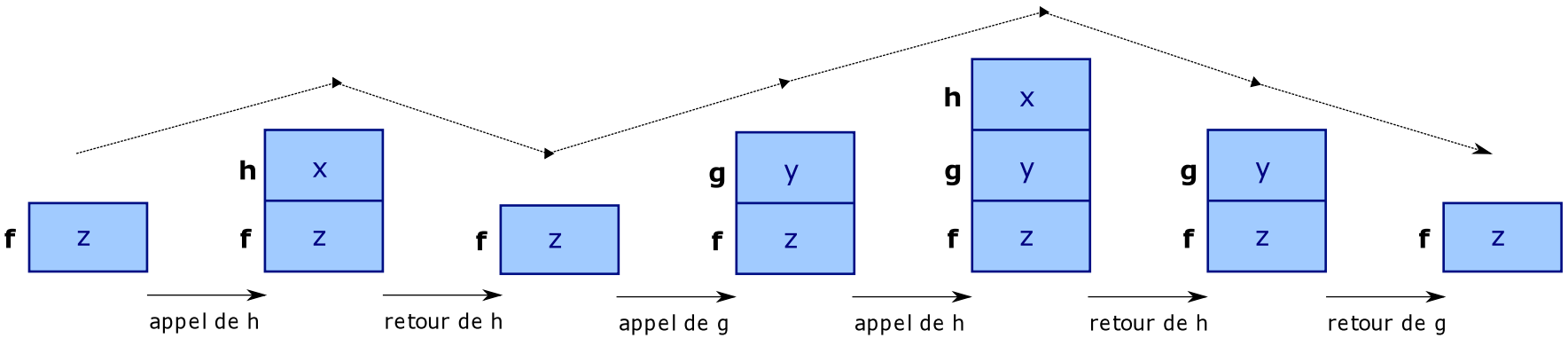
1er décembre 2020

Rappel : modèle d'exécution par pile (en C)

```
void h(int x) {
}
void g(int y) {
  h(y+1);
}
void f() {
  int z;
  h(2);
  g(3);
}
```

L'exécution des appels de fonction forme une pile de blocs d'activation.

- appel : empilage des arguments, variables locales, temporaires, adresse de retour ;
- retour : dépilage, saut à l'adresse de retour.



Rappel : retour de fonction

retour anticipé

```
void f() {
  while(...) {
    if (...) return;
    ...
  }
}
```

saut long

```
jmp_buf buf;
void f() {
  if (setjmp(buf)==0) {
    g();
  }
}
void g() {
  h();
  ...
}
void h() {
  if (...)
    longjmp(buf);
}
```

Il est facile de :

- sortir en tout point d'une fonction vers l'appelant direct ;
  pas uniquement en fin de fonction
- dépiler plusieurs blocs d'activation en une fois, saut calculé
  avec `setjmp` et `longjmp`, voir le cours sur les exceptions

Limitation du modèle à pile unique

appel de coroutine

```
void f() {
  c = call g(12);
  ...
  resume c;
  ...
  resume c;
  ...
}
```

coroutine

```
void g(int n) {
  for (int i=0; i<n; i++) {
    yield();
  }
}
```

Par contre, les coroutines sont impossibles, i.e. :

- créer un appel à `g` en fixant la valeur des arguments (`call`)
- puis exécuter un morceau de `g` et retourner avant la fin (`yield`)
- puis exécuter la suite de `f` , après l'appel à `g`
- puis reprendre l'exécution de `g` là où elle s'était arrêtée, en retrouvant les valeurs des variables locales (`restore`).

Raison : une fois le bloc d'activation dépilé, toutes ses variables locales disparaissent et ne peuvent pas être restaurées.

Au-delà du modèle à pile unique

But : pouvoir interrompre une exécution et la reprendre.

Applications :

- gestion des erreurs avec réessai ;
  (`try ...error ...catch { ...retry; }`)
- itérateurs de collection (e.g., les générateurs en Python) ;
- modèle producteur / consommateur (e.g., compression a volée) ;
- programmation réactive synchrone (e.g., systèmes embarqués, jeux vidéo) ;
- programmation concurrente.

Solutions : c.f. TME 7

- créer explicitement plusieurs piles (en C) ;
- ou utiliser les processus légers : les *threads* (en Java ou en C).

Piles multiples en C

Piles multiples en C

`ucontext` : interface de bas niveau en C de manipulation des piles.

- comme `longjmp`, permet de modifier le pointeur de pile ;
- permet également de créer des piles séparées,
  et de sauter d'une pile à l'autre.

Chaque pile comporte ses propres blocs d'activation et permet une séquence d'appels et de retours indépendante des autres piles.

C'est une unité d'exécution de programme indépendante.

- à la différence des *threads*, le changement de pile est contrôlé par le programme, pas par le système, et il n'y a pas d'exécution parallèle.

Interface `ucontext` (1/2)

L'interface est accessible par : `#include <ucontext.h>`

Elle contient un type :

- `ucontext_t`

  information de contexte (similaire à `jmpbuf_t`)
  contient un pointeur de pile et une copie des registres

et deux fonctions permettant de simuler `setjmp` et `longjmp` :

- `int getcontext (ucontext_t *ucp)`

  initialise `ucp` avec l'information de contexte courant
  similaire à `setjmp`
  sauter à `ucp` revient à sauter à l'instruction suivant le retour de `getcontext`

- `int setcontext (ucontext_t *ucp)`

  saute vers le contexte `ucp`
  la fonction ne retourne jamais, comme `longjmp`

Exemple : simulation de saut long

```
#include <ucontext.h>

ucontext_t uc;

void f() {
  if (getcontext (&uc) < 0) // erreur ...
  g();
}

void g() {
  h();
}

void h() {
  setcontext(&uc);
}
```

Très similaire à `setjmp` et `longjmp`, pour l'instant...

Interface `ucontext` (2/2)

Des champs de `ucontext_t` permettent de gérer des piles multiples allouées par le programmeur (pas par le système) :

- `uc_stack.ss_sp` : pointeur vers le début de la pile ;
- `uc_stack.ss_size` : taille de la pile ;
- `uc_link` : où sauter après un `return` quand la pile est vide.

Les fonctions suivantes sont alors utiles :

- `void makecontext (ucontext_t *ucp, void (*f) (void), int argc, ...)`

  change la pile et le point de saut d'un contexte

  la structure `ucp` doit être d'abord initialisée par `getcontext`,
  puis les champs `uc_stack`, `uc_link` renseignés avant l'appel ;
  sauter vers `ucp` revient alors à exécuter `f` dans la nouvelle pile

- `int swapcontext (ucontext_t* oucp, ucontext_t * ucp)`

  stocke le contexte courant dans `oucp` et saute vers le contexte `ucp`

  sauter vers `oucp` revient à sauter à l'instruction suiant le retour de `swapcontext`

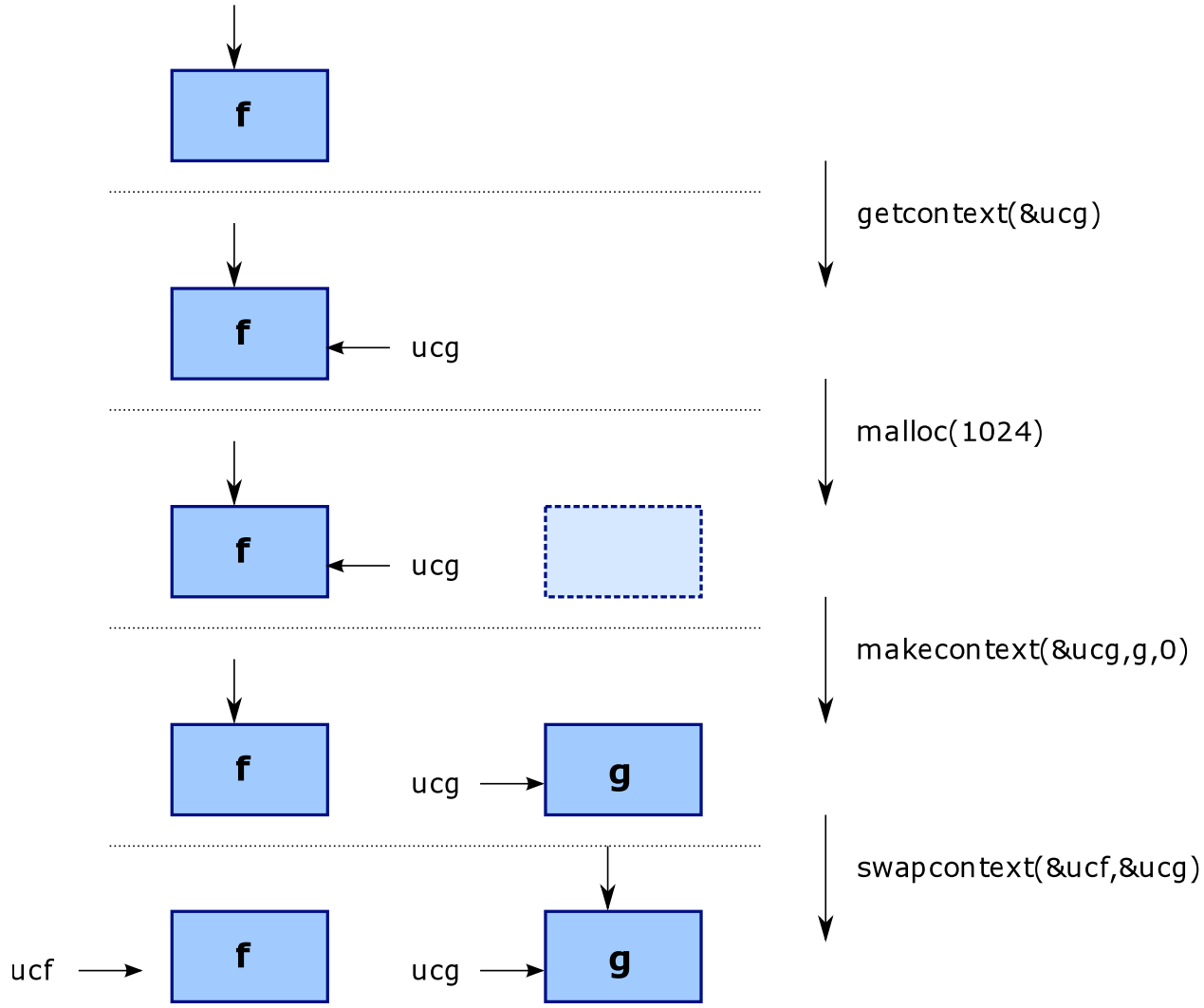
Exemple : navigation entre piles multiples

```
#include <ucontext.h>

ucontext_t ucf, ucg;

void f() {
  getcontext(&ucg);
  ucg.uc_stack.ss_sp = malloc(1024);
  ucg.uc_stack.ss_size = 1024;
  makecontext(&ucg, g, 0);
  // A ...
  swapcontext(&ucf, &ucg);
  // B ...
  swapcontext(&ucf, &ucg);
  // C ...
}

void g() {
  // D ...
  swapcontext(&ucg, &ucf);
  // E ...
  setcontext(&ucf);
  // F ...
}
```

Flot d'exécution : A, D, B, E, C.

- initialisation d'un contexte `ucg` ;
- une pile est allouée
  et `ucg` est attaché à la pile ;
- `ucg` est attaché à la fonction `g` ;
- échange de contrôle
  entre `ucg` et `ucf` ;
- puis entre `ucg` et `ucf`, etc. ;
- le contrôle revient à la fin à `ucf`,
  donc à `f`.

Illustration de la création de piles multiples



Processus légers en Java

Processus et processus légers

En système, un processus est une unité d'exécution indépendante, donc avec son propre flot de contrôle et sa propre pile.

- processus :
  un programme, avec son propre espace mémoire isolé des autres ;
- processus « légers » ou *threads* :
  unité d'exécution dans un programme
  créée dynamiquement
  partageant les variables globales et les ressources.

Les processus s'exécutent en concurrence :

- parallélisme réel : sur des cœurs différents ;
- parallélisme simulé : par changements de contextes rapides ;
- ou un mélange des deux.

Un ordonnanceur contrôle quel(s) processus s'exécute à chaque instant.

Processus léger en Java

Plusieurs manières de faire en Java.

Manière utilisée ici : hériter de la classe `java.lang.Thread`.

- la classe doit définir une méthode `void public run()` ;
  point d'entrée de la thread
- une fois l'instance de thread créée, appeler sa méthode `start`.

Exemple de thread en Java

```
public class Activity extends Thread
{
  private int x;
  public Activity(int x) {
    this.x = x;
    start();
  }
  public void run() {
    // ...
  }
}
...
new Activity(1);
new Activity(2);
new Activity(3);
```

Contrôle de l'ordonnancement

L'ordre d'exécution des threads est contrôlé par l'ordonnanceur ;
celui-ci choisit quelles threads non-bloquées s'exécutent effectivement.

Le contrôle sur les threads se fait par des primitives de synchronisation qui peuvent les bloquer !

Exemple : le sémaphore `java.util.concurrent.Semaphore`

- contient un compteur entier : nombre de resources ;
- `acquire` : consomme une ressource (décrémente le compteur) ;
- `release` : produit une ressource (incrémente le compteur) ;
- le compteur ne descend jamais en dessous de zéro ;
  `acquire` sur un compteur à zéro force la thread a attendre le prochain `release` !
  $\Longrightarrow$ contrôle de l'exécution des threads.

Interface très simple, mais très puissante...
de nombreuses utilisations en programmation concurrente.

Exemples d'utilisation de sémaphore

section critique

```
Semaphore x = new Semaphore(1);

void m1() {
  x.acquire();
  // section critique
  x.release();
}

void m2() {
  x.acquire();
  // section critique
  x.release();
}
```

signal

```
Semaphore x = new Semaphore(0);

void m1() {
  // initialisation
  x.release();
  // calcul, après initialisation
}

void m2() {
  x.acquire();
  // calcul, après initialisation
}
```

- section critique :
  une seule méthode parmi `m1` et `m2` s'exécute à un instant donné ;
- signal : `m1` signale à `m2` quand elle peut commencer son exécution.

Bonus : continuations en langages fonctionnels

Notion de continuation

Une continuation est une représentation du calcul restant à effectuer.
notion très abstraite, dépendant du modèle et du langage de programmation !

Utilité : continuations de première classe, réification

- voir les continuations comme des valeurs du langage ;
- les stocker, les passer en argument ;
- changer la continuation courante
  pour altérer le flot de contrôle du programme.

Très général : modélise tous les effets de contrôle, local ou non-local !

Dans un modèle d'exécution à pile (à la C ou Java), la continuation est :

- la position de la prochaine instruction à exécuter ;
- mais aussi tout le contenu de la pile !

Coûteux...
Peut-on éviter les piles multiples ou les copies de pile ?
Oui, avec les fonctions de première classe.

Exemple : coroutines encodées avec des fonctions (1/2)

appel de coroutine

```
function f() {
  let z = 12 in
  x = 1;
  print x;
  c = call g (2);
  print x+z;
  resume c;
  print x+z;
  resume c;
  print x+z
}
```

coroutine appelée

```
function g(y) {
  x = x+y;
  yield;
  x = x+y;
  yield;
  x = x+y
}
```

Exemple de couroutine :

- lors d'un `yield`, `f` doit retrouver son point d'exécution antérieur et la valeur de la variable locale `z` ;
- lors d'un `resume`, `g` doit retrouver son point d'exécution antérieur et la valeur de la variable locale `y`.

Exemple : coroutines encodées avec des fonctions (2/2)

appel de coroutine

```
function f() {
  let z = 12 in
  x = 1;
  print x;
  g (2, fun cont1 ->
  print x+z;
  cont1 (fun cont2 ->
  print x+z;
  cont2 (fun cont3 ->
  print x+z
  )))
}
```

coroutine appelée

```
function g(y,cont) {
  x = x+y;
  cont (fun cont4 ->
  x = x+y;
  cont4 (fun cont5 ->
  x = x+y;
  cont5 (fun () -> ())
  ))
}
```

Ressemble beaucoup au programme original.
Mais les `yield` et `resume` sont changés en des appels de fonctions !

Les fonctions anonymes `fun cont`*i* `-> ...` encodent les continuations.
Une continuation prend en argument une autre continuation `cont`*i*, exécute une instruction, puis appelle `cont`*i*.

Exécution de l'exemple (1/3)

appel de coroutine

```
function f() {
  let z = 12 in
  x = 1;
  print x;
  g (2, fun cont1 ->
  print x+z;
  cont1 (fun cont2 ->
  print x+z;
  cont2 (fun cont3 ->
  print x+z;
  )))
}
```

coroutine appelée

```
function g(y,cont) {
  x = x+y;
  ————————
  cont (fun cont4 ->
  x = x+y;
  cont4 (fun cont5 ->
  x = x+y
  cont5 (fun () -> ())
  ))
}
```

Exécution du début de `f` et de la première instruction de `g`.
`cont` dénote le code de `f` à exécuter après `print x`.

Exécution de l'exemple (2/3)

appel de coroutine

```
function f() {
  let z = 12 in
  x = 1;
  print x;
  g (2, fun cont1 ->
  print x+z;
  ---------
  cont1 (fun cont2 ->
  print x+z;
  cont2 (fun cont3 ->
  print x+z
  )))
}
```

coroutine appelée

```
function g(y,cont) {
  x = x+y;
  cont (fun cont4 ->
  x = x+y;
  cont4 (fun cont5 ->
  x = x+y;
  cont5 (fun () -> ())
  ))
}
```

L'exécution de `f` reprend, et exécute `print x+z`.
L'argument `cont1` de la fonction anonyme exécutée
est la continuation de `g`, a exécuter après le premier `x = x+y`.

Exécution de l'exemple (3/3)

appel de coroutine

```
function f() {
  let z = 12 in
  x = 1;
  print x;
  g (2, fun cont1 ->
  print x+z;
  cont1 (fun cont2 ->
  print x+z;
  cont2 (fun cont3 ->
  print x+z
  )))
}
```

coroutine appelée

```
function g(y,cont) {
  x = x+y;
  cont (fun cont4 ->
  x = x+y;
  ————————
  cont4 (fun cont5 ->
  x = x+y;
  cont5 (fun () -> ())
  ))
}
```

L'exécution de `g` reprend avec le deuxième `x = x+y`,
`cont4` dénote le code de `f` à exécuter après le `print x+z`,
et ainsi de suite...

Justification

appel de coroutine

```
function f() {
  let z = 12 in
  x = 1;
  print x;
  g (2, fun cont1 ->
  print x+z;
  cont1 (fun cont2 ->
  print x+z;
  ...
```

coroutine appelée

```
function g(y,cont) {
  x = x+y;
  cont (fun cont4 ->
  x = x+y;
  cont4 (fun cont5 ->
  x = x+y;
  cont5 (fun () -> ())
  ))
}
```

Cette technique fonctionne grâce à la portée lexicale.

La variable locale `z` reste accessible dans toutes les continuations de `f`.
$\Longrightarrow$ sa valeur est préservée malgré les appels à `g`.

C'est donc le mécanisme de fonctions locales et de clôture
qui se charge de copier et restaurer l'environnement depuis la pile,
en se limitant aux moreaux utiles.

Note : une implantation efficace doit quand même faire attention à optimiser les appels terminaux.

Pour aller plus loin. . .

- La transformation que nous avons faite s'appelle la méthode par passage de continuation (*continuation passing style*)
  - elle peut être appliquée à un programme entier ;
  - c'est aussi une forme intermédiaire de certains compilateurs.

- Peu de langages offrent un accès direct à la continuation :
  - `call/cc` introduit en Scheme ;
  - `callcc` offert en SML.

  D'autres ont des bibliothèques pour cela (OCaml).

Résumé :

- les continuations de première classe offrent un support général pour le contrôle avancé ;
  (exceptions, coroutines, générateurs, processus coopératifs, . . .)
- les fonctions de première classe permettent d'implanter les continuations de première classe.