Contexte d'exécution / coroutines

Philippe Quéinnec

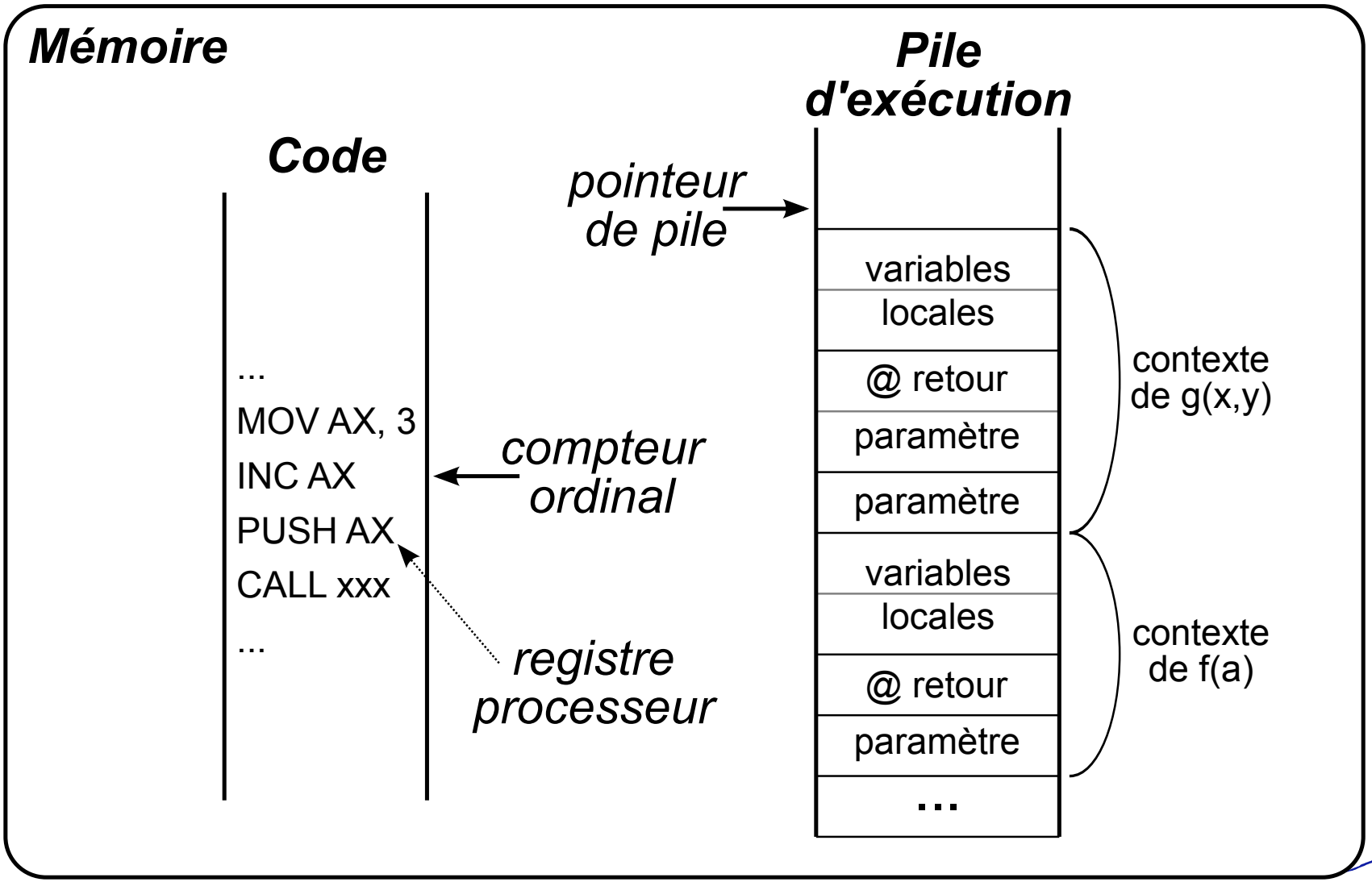
6 mai 2020

Machine support

- Notre machine d'exécution est... un processus Unix.
- Analogue à l'exécution dans une machine virtuelle.
- Nous allons développer un noyau d'exécution parallèle s'exécutant sur cette machine virtuelle.
- Pour autant, tout ce qu'on va voir pourrait se faire, sans guère de changements, sur un véritable support matériel, mais :
  - difficulté de développement
  - difficulté de mise au point
  - protection et isolation des bugs
  - accessibilité à un ordinateur nu

Plan

Exécution de code



Contexte d'exécution

Définition

Le contexte d'exécution est, idéalement, l'ensemble des informations (code, données...) qui capture l'état d'un calcul en court, i.e. tel que si un contexte d'exécution est sauvé puis ultérieurement remis en place, le calcul continue exactement au point où il en était.

Que met-on dans un contexte d'exécution ?

- code à exécuter $\rightarrow$ point courant du code $\rightarrow$ compteur ordinal
- calculs intermédiaires $\rightarrow$ $\begin{cases} \text{état de la pile} \rightarrow \text{ptr de pile} \\ \text{état du CPU} \rightarrow \text{registres} \end{cases}$
- autre contexte matériel $\rightarrow$ masque d'interruption

Interface

- Type opaque `ucontext_t`
- obtention du contexte courant de l'appelant :
  `getcontext(ucontext_t *ucp);`
- installation d'un contexte :
  `setcontext(ucontext_t *ucp);`
- initialisation d'un contexte :
  ```
  makecontext(ucontext_t *ucp, void (*f)(),
                                  int argc, ...);
  ```
  En cas d'installation de ce contexte, la fonction `f` est exécutée avec les arguments spécifiés.
- changement de contexte :
  `swapcontext(ucontext_t *old, ucontext_t *new);`
  Le contexte courant est sauvé dans `old` et un nouveau contexte est installé.
  Rq : `swapcontext(o,n)` $\neq$ `getcontext(o); setcontext(n);`

Exemple – ping/pong (1/2)

```
ucontext_t c_pp, c_ping, c_pong;

int main() {
  getcontext(&c_ping);
  c_ping.uc_stack.ss_sp = valloc(1024); // pile de ce contexte
  c_ping.uc_stack.ss_size = 1024;
  makecontext(&c_ping, ping, 0);

  getcontext(&c_pong);
  c_pong.uc_stack.ss_sp = valloc(1024); // pile de ce contexte
  c_pong.uc_stack.ss_size = 1024;
  makecontext(&c_pong, pong, 0);

  swapcontext(&c_pp, &c_ping);
  printf ("Fin");
  return 0;
}
```

Exemple – ping/pong (2/2)

```
void ping (void) {
    for (int i = 0; i < 10; i++) {
        printf ("ping %d\n", i);
        swapcontext(&c_ping, &c_pong);
    }
    printf ("Ping a fini\n");
    swapcontext(&c_ping, &c_pong);
}

void pong (void) {
    for (int i = 0; i < 10; i++) {
        printf ("     pong %d\n", i);
        swapcontext(&c_pong, &c_ping);
    }
    printf ("Pong a fini\n");
    setcontext(&c_pp);
}
```

Implantation

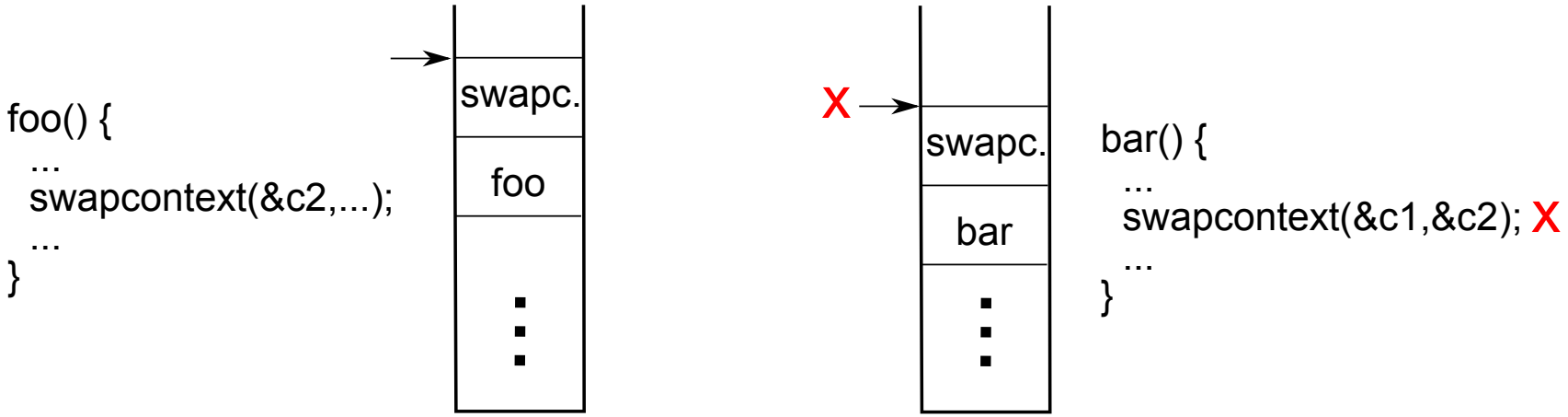
Implantation

- Fournis par le support système, implantés en assembleur
- mais implantable « à la main »

Hypothèses

- possibilité de lire et changer le pointeur de pile (CPUstackptr)
- possibilité de sauvegarder et restaurer les registres (CPUregisters)
- les variables locales sont rangées dans la pile ou les registres
- l'appel de sous-routine (JSR/CALL) range l'adresse de retour dans la pile
- Il n'est pas nécessaire de connaître la structure de la pile !

Implantation de swapcontext




```
swapcontext(old, new)
{
  old->registers = CPUregisters;
  old->stackptr = CPUstackptr;
  // X
  CPUregisters = new->registers
  CPUstackptr = new->stackptr
  return // dépile l'@ de retour dans la pile installée
}
```

Implantation de getcontext/setcontext

- getcontext(c) = swapcontext(c, c)
- setcontext(c) = swapcontext(unused, c)
- makecontext : construire une pile d'appel qui ressemble à un appel de swapcontext, et tel que le retour de cet "appel" exécute le code fourni.

Plan

Coroutines

Définition

Une coroutine = 1 traitement en cours = 1 contexte d'exécution

Faible abstraction par rapport au contexte d'exécution mais manipulation plus simple.

En interne, un contexte / une couroutine contient :

- une sauvegarde du compteur ordinal
  (quel code va-t-on exécuter ?)
- une sauvegarde du pointeur de pile et des autres registres
  (état de la pile d'appel de procédure et des variables locales)

Interface

- Type opaque `coroutine_t`
- Création d'une coroutine :
  ```
  coroutine_t cor_créer(char *nom,
                        void (*code)(void *arg), void *a);
  ```
  crée une nouvelle coroutine avec une pile vide et un contexte tel qu'un transfert causera l'exécution de `code(a)`.
- Transfert du contrôle :
  ```
  void cor_transférer(coroutine_t suspendue,
                      coroutine_t activée);
  ```
  Sauvegarde l'état courant du calcul dans suspendue et donne le contrôle à activée.
- Destruction :
  ```
  void cor_détruire(coroutine_t cor);
  ```
  Libère la coroutine et les ressources associées, qui ne doivent plus être accédées ensuite.

Remarques

- Une coroutine n'est pas une procédure, ni un processus, ni...
- `transférer` n'est pas une ≪ véritable ≫ procédure :
  Lors de `cor_transférer(c1,c2)`, `c2` est installé
  → le compteur ordinal change (≈ on part ailleurs) et on ne revient pas de l'appel.
  On n'exécutera le code qui suit l'appel à `transférer` que si, un jour, `c1` est installé.
- `transférer(c,c)` : l'état courant du calcul au moment de l'appel est sauvegardé dans `c`, et le contrôle est transférée à la coroutine qui était *contenue dans c à l'appel de transférer*.

Exemple (1)

Que fait le code suivant ?

c_1, c_2, c_3, c_4 sont des var. globales contenant des `coroutine_t`.

code initial	code1	code2
$c_1 =$ cor_créer(_,code1,_) ;		
$c_2 =$ cor_créer(_,code2,_) ;	δ :	
⋮	⋮	ϵ :
α : cor_transférer(c_3,c_1) ;	γ : cor_transférer(c_4,c_3) ;	⋮
⋮	⋮	θ :
β : cor_transférer(c_4,c_2) ;		

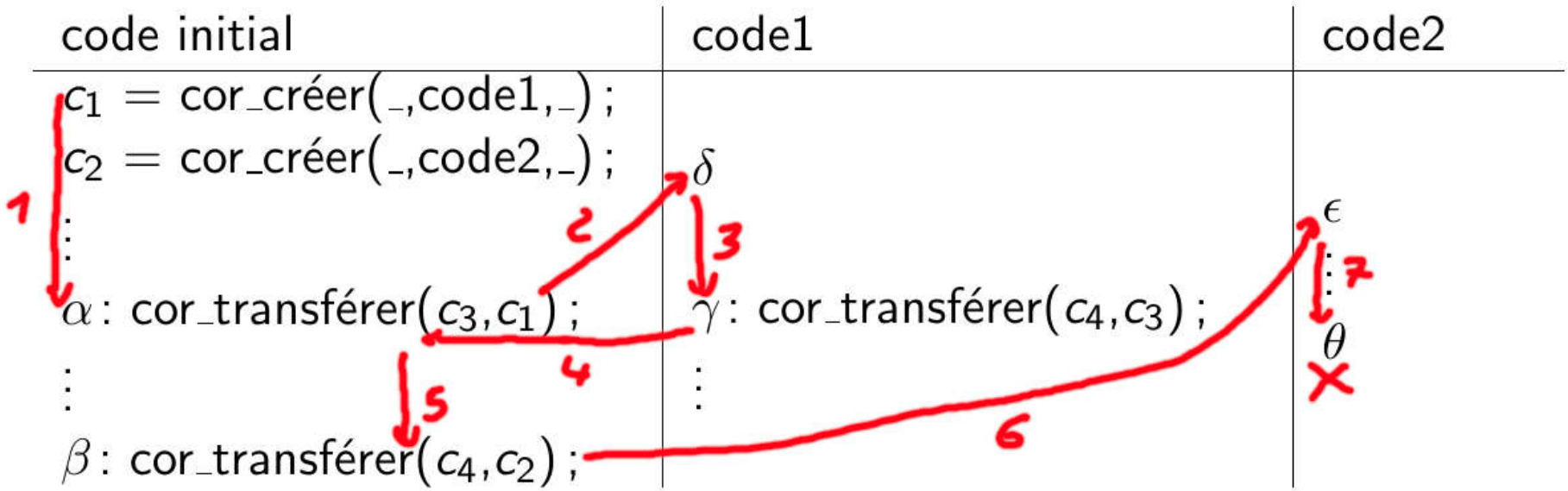
Exécution (1)

Une coroutine est un couple (cpt ordinal, pile).
Trois piles : P_0 (pour le code initial), P_1 et P_2 (créées par les appels à `cor_creer`)

	1	2	3	4	5	6	7
courant	α, P_0	δ, P_1	γ, P_1	α', P_0	β, P_0	ϵ, P_2	θ, P_2
c_1	$code1, P_1$						
c_2	$code2, P_2$						
c_3	–	α, P_0					
c_4	–			γ, P_1		β, P_0	

- une case blanche signifie que la valeur est inchangée
- une case bleue est le résultat d'un transfert.
- α' indique l'instruction qui suit α.
- Et une fois en θ, P_2 ? Retour d'un appel de procédure qui n'existe pas (la pile est vide) $\Rightarrow$ tout plante.

Exécution (1)


code initial
code1
code2
c_1 = cor_créer(_,code1,_);
c_2 = cor_créer(_,code2,_);
α : cor_transférer(c_3,c_1);
β : cor_transférer(c_4,c_2);
γ : cor_transférer(c_4,c_3);
δ
ϵ
θ
1
2
3
4
5
6
7

Exemple (2)

Que fait le code suivant ?

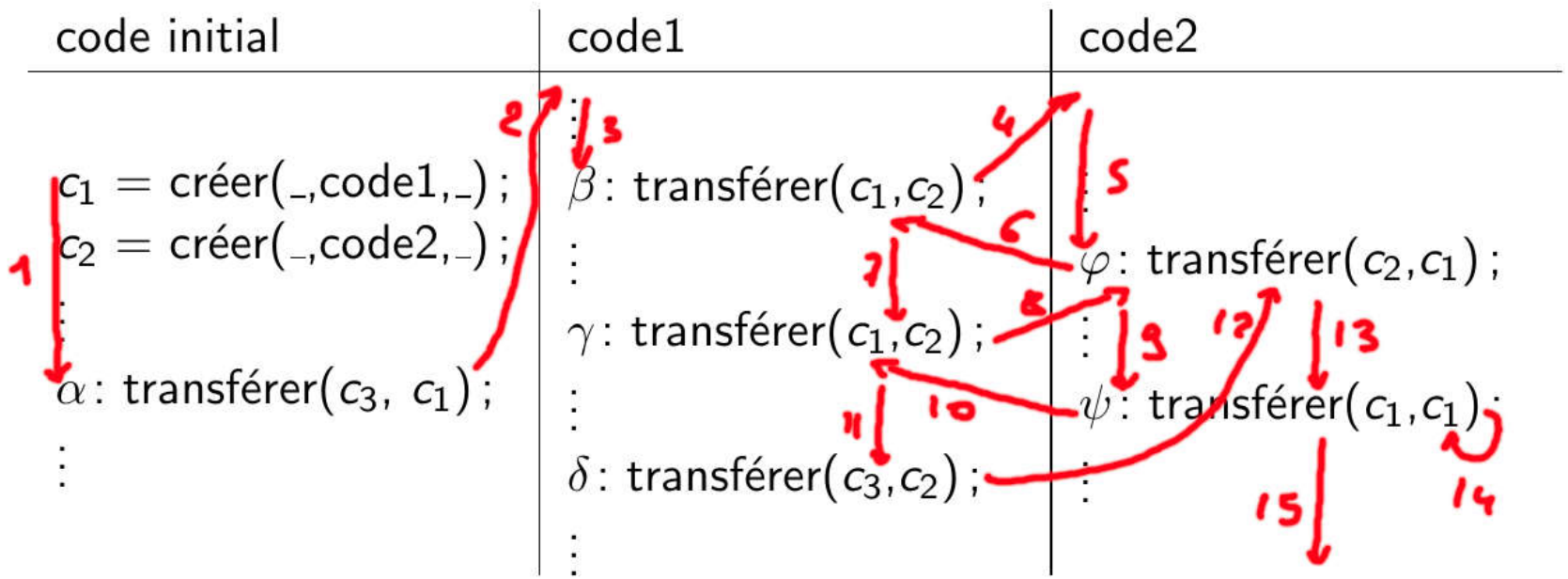
code initial	code1	code2
	⋮	
$c_1 =$ créer(_,code1,_) ;	β : transférer(c_1,c_2) ;	⋮
$c_2 =$ créer(_,code2,_) ;	⋮	φ : transférer(c_2,c_1) ;
⋮	γ : transférer(c_1,c_2) ;	⋮
α : transférer(c_3, c_1) ;	⋮	ψ : transférer(c_1,c_1) ;
⋮	δ : transférer(c_3,c_2) ;	⋮
	⋮	

Exécution (2)

Pour simplifier, on n'indique pas la pile (qui est sans utilité ici : pas de variable locale, pas d'appel de fonction).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
courant	α	$c1$	β	$c2$	φ	β'	γ	φ'	ψ	γ'	δ	φ'	ψ	ψ'	X
c_1	$c1$			β				γ						ψ	
c_2	$c2$					φ				ψ					
c_3	_	α										δ			

(c1 = entrée de code1, c2 = entrée de code2)

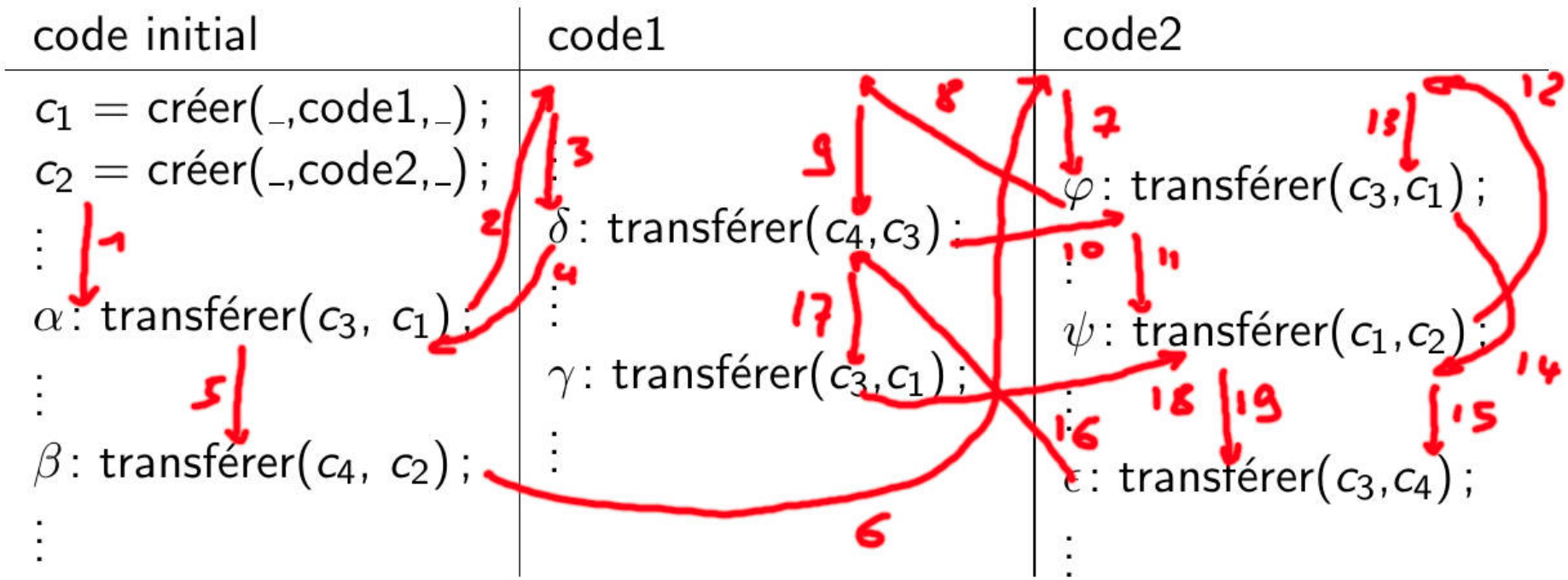


Exemple (3)

Que fait le code suivant ?

code initial	code1	code2
$c_1 =$ créer(_,code1,_) ;		⋮
$c_2 =$ créer(_,code2,_) ;	⋮	φ : transférer(c_3,c_1) ;
⋮	δ : transférer(c_4,c_3) ;	⋮
α : transférer(c_3, c_1) ;	⋮	ψ : transférer(c_1,c_2) ;
⋮	γ : transférer(c_3,c_1) ;	⋮
β : transférer(c_4, c_2) ;	⋮	ϵ : transférer(c_3,c_4) ;
⋮		⋮

Exécution (2)



Après 19, boucle infinie en suivant 16–17–18–19,
analogue à du temps partagé entre code1 (17) et code2 (19).

Exemple : importance de la pile

Variables globales : Gy (int), c_1, c_2, c_3 (coroutines).

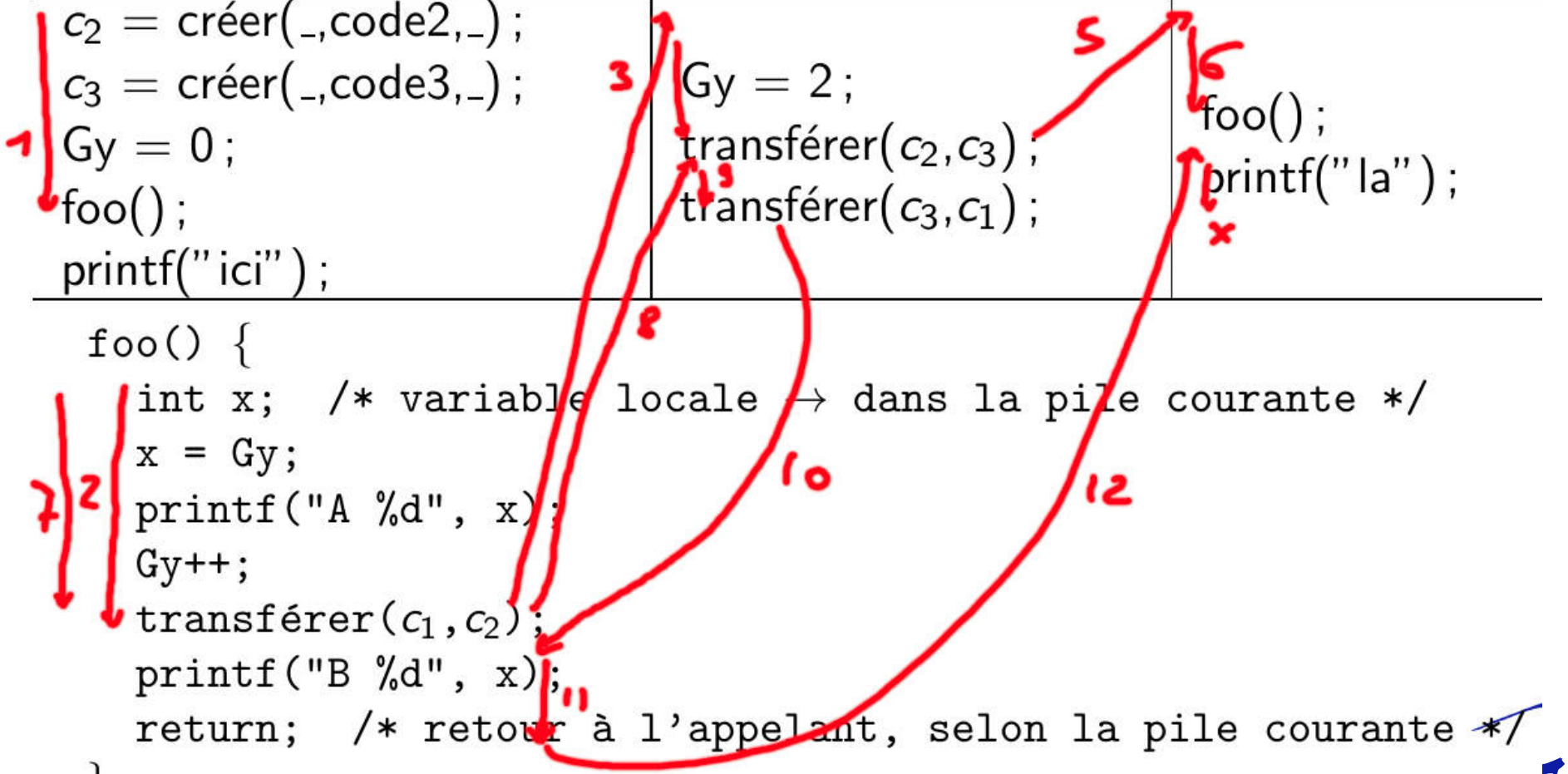
code initial	code2	code3
c_2 = créer(_,code2,_) ; c_3 = créer(_,code3,_) ; Gy = 0 ; foo() ; printf("ici") ;	Gy = 2 ; transférer(c_2,c_3) ; transférer(c_3,c_1) ;	foo() ; printf("là") ;

```
foo() {
  int x;  /* variable locale → dans la pile courante */
  x = Gy;
  printf("A %d", x);
  Gy++;
  transférer(c1,c2);
  printf("B %d", x);
  return;  /* retour à l'appelant, selon la pile courante */
}
```

Exécution (4)

code initial	code2	code3
c_2 = créer(_,code2,_) ; c_3 = créer(_,code3,_) ; Gy = 0 ; foo() ; printf("ici") ;	Gy = 2 ; transférer(c_2,c_3) ; transférer(c_3,c_1) ;	foo() ; printf("la") ;

```
foo() {
  int x;  /* variable locale → dans la pile courante */
  x = Gy;
  printf("A %d", x);
  Gy++;
  transférer(c1,c2);
  printf("B %d", x);
  return;  /* retour à l'appelant, selon la pile courante */
}
```



Exécution (4)

Affiche : A 0, A 2, B 2, là.
Suivre la pile courante, qui est la pile sauvegardée lors d'un transfert :

(1) pile courante P_1, piles créées P_2 (dans c_2) et P_3 (dans c_3).
(2) x dans P_1 vaut 0
(3) c_1 est écrasé avec la pile P_1
(5) installation de la pile P_3
(7) x dans P_3 vaut 2
(8) c_1 est écrasé avec P_3
(10) installation de la pile P_3 (depuis c_1)
(11) x dans la pile courante vaut 2
(12) return dans la pile courante, qui est P_3 : on revient après l'appel de foo() dans code3.

Mise en œuvre des coroutines

À peu de chose près :

- 1 coroutine $\approx$ 1 contexte
- transférer $\approx$ swapcontext

Différences :

- Mauvaise terminaison détectée
- `cor_transferer(c,c)` installe l'ancienne valeur de c, et écrase c avec le point de départ.
  `swapcontext(c,c)` écrase c avec le point de départ, puis installe c $\Rightarrow$ pas de transfert (équivalent à `getcontext(c)`).
- Destruction

Terminaison

Le code d'une coroutine ne doit jamais se terminer (pas d'appelant : retour où ?)
Solution = enveloppe

Enveloppe

L'enveloppe permet :

- de réaliser des actions d'initialisation *dans l'environnement de la nouvelle entité*,
- de réaliser des actions de terminaison, toujours dans cet environnement.

Implantation (1)

```
struct coroutine {
  char *nom;                  // pour débugger
  void (*code) (void *); // fonction à un paramètre
  void *arg;                  // le paramètre
  ucontext_t uc;              // le contexte
  void *stack;                // la pile (pour le ménage)
};

typedef struct coroutine *coroutine_t;

static void enveloppe (void *cv)
{
    coroutine_t c = (coroutine_t)cv;
    (c->code) (c->arg);  // appel de la fonction associée
    fprintf (stderr, "COROUTINES: fin sale.\n");
}
```

Implantation (2)

```
coroutine_t cor_créer (char *nom,
                       void (*code) (void *arg), void *arg)
{  coroutine_t t;
   t = malloc (sizeof (struct coroutine));
   t->nom = strdup (nom);
   t->code = code;
   t->arg = arg;
   getcontext (&(t->uc));
   t->stack = valloc (COR_STACKSIZE);   // ≈ malloc
   t->uc.uc_stack.ss_size = COR_STACKSIZE;
   t->uc.uc_stack.ss_flags = 0;
   t->uc.uc_stack.ss_sp = (char *)t->stack;
   makecontext (&(t->uc), enveloppe, 1, t);
   return t;
}
```

Implantation (3)

```
void cor_transférer (coroutine_t suspendue,
                     coroutine_t activé)
{
    ucontext_t uc = activé->uc;  // évite l'écrasement
    swapcontext (&(suspendue->uc), &uc);
}

// Note : une coroutine ne peut pas se détruire
// elle-même vu que la pile est libérée.
void cor_detruire (coroutine_t c)
{
    free (c->nom);
    free (c->stack);
    free (c);
}
```

Plan

Exceptions

- Bloc `try` bloc_1 `catch e` bloc_2
  + `raise e`
- `try` = sauvegarder le contexte puis exécuter le bloc_1
- `raise` = restaurer un contexte sauvegardé correspondant à l'exception `e`, et exécuter le bloc_2 correspondant
- plusieurs blocs imbriqués pour une même exception + plusieurs exceptions $\Rightarrow$ pile de sauvegarde de contexte

Implantation (1)

```
typedef struct exception {
    char *nom;
} *exception_t;

typedef struct exception_traitant {
    exception_t  exception;
    int          levée;
    ucontext_t   retour;
} exception_traitant_t;
```

Implantation (2)

try $bloc_1$ catch e $bloc_2$

```
{
    exception_traitant_t traitant;
    traitant.exception = e;
    traitant.levée = 0;
    push(pile_exception, &traitant);
    getcontext (&traitant.retour);
    if (! traitant.levée) { // code normal
        bloc_1
        pop(pile_exception);
    } else { // exception levée
        bloc_2
    }
```

Implantation (3)

raise(e)

```
exception_traitant_t traitant;
do {
   /* Panique si pile vide (exception non captée) */
   traitant = pop(pile_exception);
} while (traitant->exception != e);
traitant->levée = 1;
setcontext(&(traitant->retour));
```

Remarques :

- une pile spécifique n'est pas indispensable : la pile d'exécution peut être utilisée
- Avec la notion de processus (à suivre), la pile d'exception sera spécifique à chaque processus.

Résumé

- Contexte d'exécution : capture et changement
- Coroutine : abstraction des contextes, notion de traitement, de transfert de contrôle
- Exceptions : restauration d'un contexte passé