Contexte d'exécution / coroutines

Philippe Quéinnec

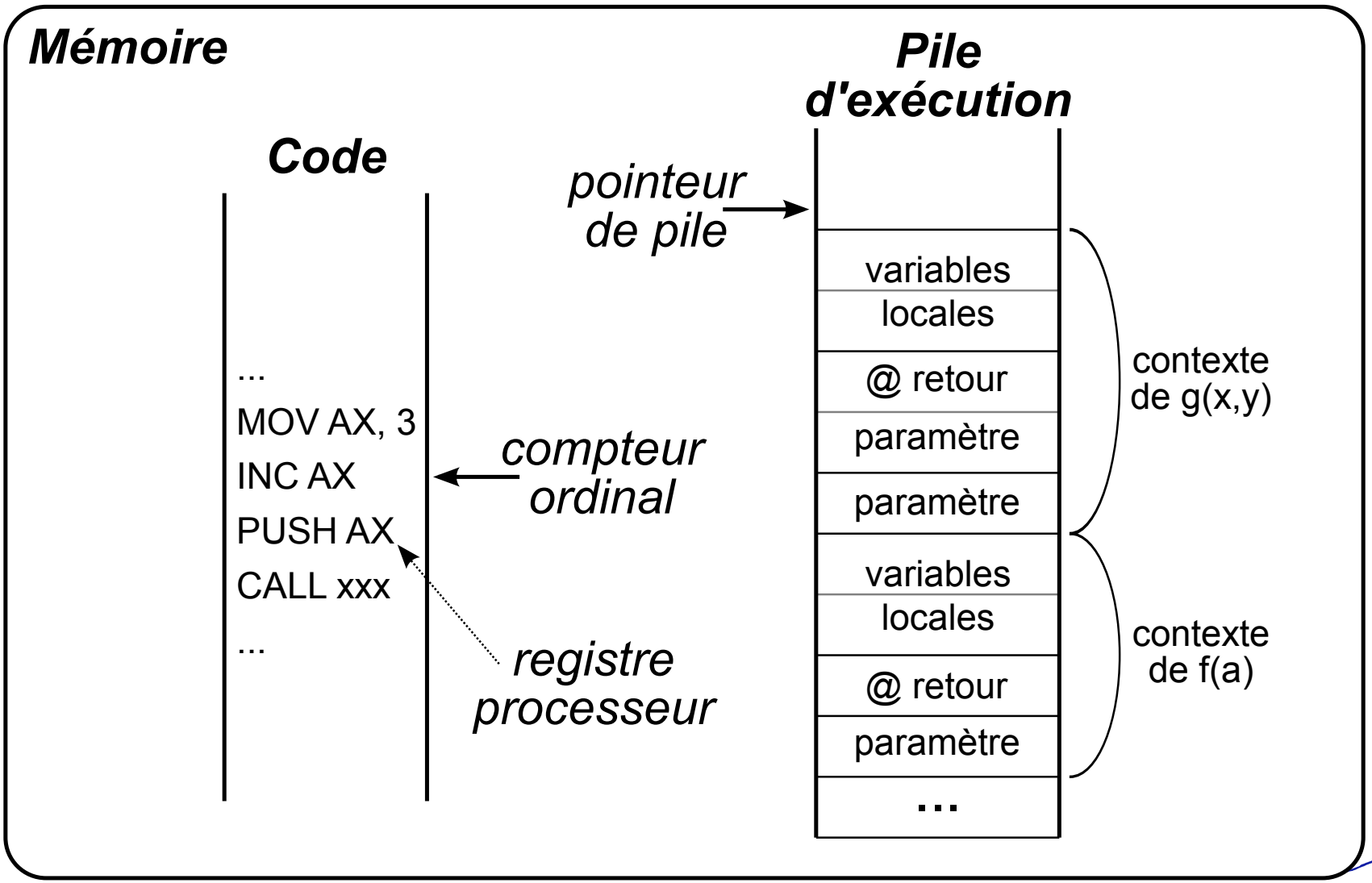
18 mai 2017

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



Exécution de code – Appel d'une fonction

Appel d'une fonction `x = foo(a,b,c)`

- Les paramètres sont empilés
- L'adresse de retour (@ de l'instruction suivante) est empilée
- Le compteur ordinal devient l'adresse de la première instruction de la fonction

Exécution de la fonction

- Les variables locales sont placées dans la pile (espace réservé soit à l'entrée, soit au fur et à mesure)

Sortie de la fonction

- Les variables locales sont dépilées (ptr de pile mis à jour)
- L'adresse de retour est récupérée
- Les paramètres sont dépilés (ou c'est l'appelant qui le fait)
- Le résultat est empilé
- Le compteur ordinal est restauré à l'adresse de retour

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 appelée 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 mis en place.
  Rq : `swapcontext(o,n)` $\neq$ `getcontext(o); setcontext(n);`

Implantation

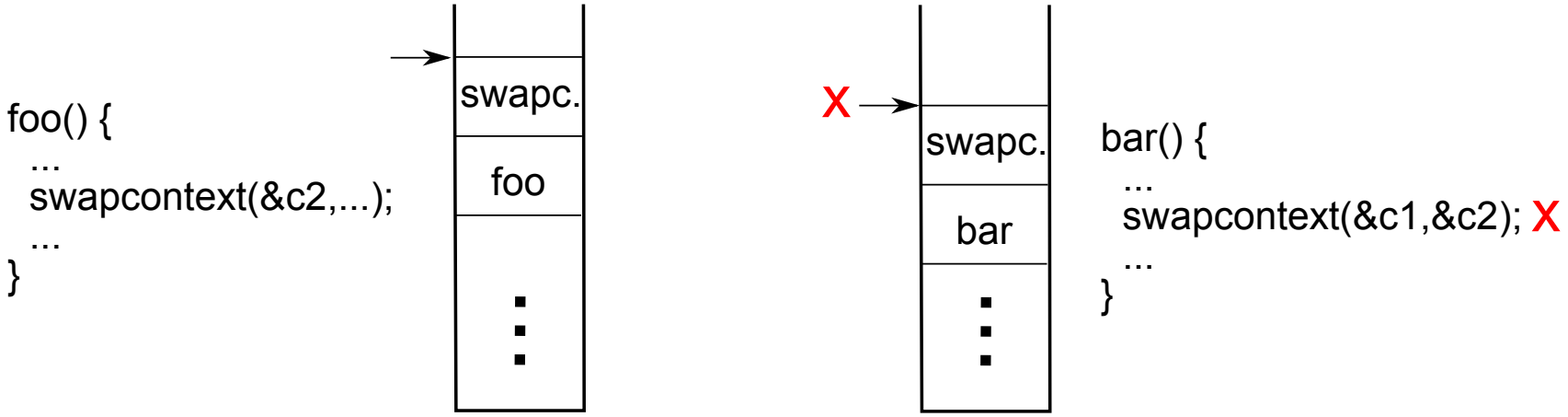
Implantation

- 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 le 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 nouvelle pile
}
```

Implantation de getcontext/setcontext

getcontext(c) = swapcontext(c, c)

setcontext(c) = swapcontext(unused, c)

Implantation de makecontext (1)

- Construire une pile d'appel qui ressemble à l'appel de swapcontext,
- Et un retour de cet "appel" exécute le code fourni.
- Difficulté supplémentaire : il faut travailler avec la pile du *contexte en cours de création* : pas d'accès aux paramètres situés dans le contexte de création.

Variables globales (hors pile) :

- Gstackptr, Gregisters : sauvegarde temporaire de l'état courant
- Gnewctx : le contexte en cours de création
- Gcode, Garg : variables temporaires pour transmettre les paramètres au nouveau contexte

État processeur :

- CPUstackptr, CPUregisters

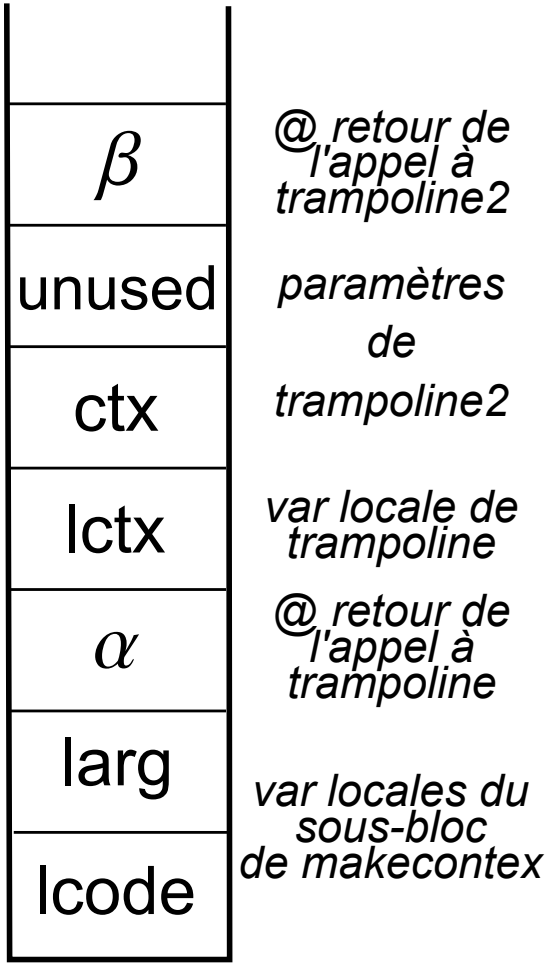
Implantation de makecontext (2)

```
/* En entrée, new doit contenir l'espace mémoire nécessaire.
 * En sortie, new est un contexte correctement initialisé
 *  tel qu'un transfert vers lui exécutera code(arg). */
void makecontext(ucontext_t *new, void (*code)(void*), void *arg)
{
    /* sauve l'état courant */
    Gstackptr = CPUstackptr; Gregisters = CPUregisters;
    new->effectivecall = 0;
    Gnewctx = new; Gcode = code; Garg = arg; /* en global */
    CPUstackptr = new->stackptr; /* Danger ! on change de pile */
    { /* variables locales dans la nouvelle pile */
        void (*lcode)(void*) = Gcode;
        void *larg = Garg;
        if (trampoline() α) { lcode(larg); γ }
    }
    /* restaure l'état initial */
    CPUstackptr = Gstackptr; CPUregisters = Gregisters;
    new->effectivecall = 1;
}
```

Implantation de makecontext (3)

```
int trampoline()
{
    ucontext_t *lctx = Gnewctx;
    trampoline2(lctx, lctx); β
    return lctx->effectivecall;
}

/* même signature que swapcontext */
void trampoline2(ucontext_t *ctx,
                 ucontext_t *unused)
{
    ctx->stackptr = CPUstackptr;
    ctx->registers = CPUregisters; X
}
```



Plan

Coroutines

Définition

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

Faible abstraction par rapport aux contextes d'exécution mais manipulation plus simple.

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 (pas de retour obligatoire).
- `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 ?

code initial	code1	code2
$c_1 =$ créer(_,code1,_) ;		
$c_2 =$ créer(_,code2,_) ;	transférer(c_4,c_3) ;	
transférer(c_3,c_1) ;	⋮	⋮
transférer(c_4,c_2) ;		

Exemple (2)

Que fait le code suivant ?

code initial	code1	code2
c_1 = créer(_,code1,_) ;	transférer(c_1,c_2) ;	transférer(c_2,c_1) ;
c_2 = créer(_,code2,_) ;	transférer(c_1,c_2) ;	transférer(c_1,c_1) ;
transférer(c_3, c_1) ;	transférer(c_3,c_2) ;	⋮
⋮	⋮	

Mise en œuvre

À peu de chose près :

- 1 coroutine = 1 contexte
- transférer = swapcontext

Terminaison

Le code d'une coroutine ne doit jamais se terminer (transfert à qui ?)
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;
    void (*code) (void *);
    void *arg;
    ucontext_t uc;
    void *stack;
};

typedef struct coroutine *coroutine_t;

static void enveloppe (void *cv)
{
    coroutine_t c = (coroutine_t)cv;
    (c->code) (c->arg);
    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);
    t->uc.uc_link = 0;/* process exits when this context returns */
    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, 2, t);
    return t;
}
```

Implantation (3)

```
void cor_transférer (coroutine_t suspendue,
                     coroutine_t activé)
{
    ucontext_t uc;
    uc = activé->uc;
    swapcontext (&(suspendue->uc), &uc);
}
```

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)

```
// Remonter (en dépilant) la pile d'exception
//   jusqu'à trouver traitant->e == e
// Panique si pas trouvée (exception non traitée)
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
- Coroutine : abstraction des contextes, notion de traitement, de transfert de contrôle
- Exceptions : restauration d'un contexte passé