Programmation concurrente

IS^{ia}3/IS^{ia}2A3

Thibault Liétard

2026

Notion de processus

Définitions

Processus
Un processus est un programme en cours d'exécution. Il exécute des instructions sur le processeur et peut utiliser de la mémoire vive ou de l'espace disque.

Processus parent
Chaque processus possède un processus parent puisqu'il est créé par un autre processus.

Le seul processus à ne pas avoir de parent est le premier à être lancé au démarrage : il s'agit du processus init, qui a pour PID 1, et qui démarre le système d'exploitation.

Identifiant

PID
identifiant unique d'un processus dans un système.

PPID
PID du processus parent d'un processus.

Afficher les processus en cours

commandes

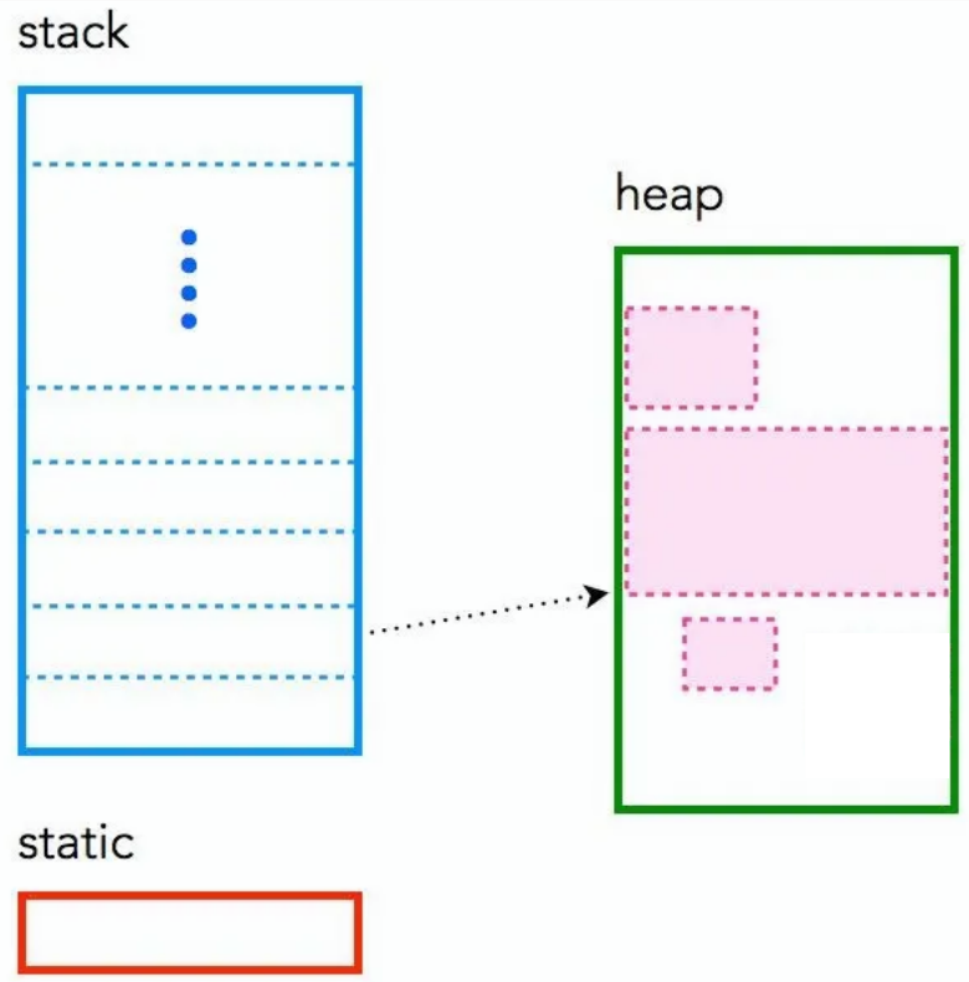
- *ps* : affiche tous les processus du terminal
- *ps -x* : affiche tous les processus
- *pstree* : affiche l'arboressence des processus
- *htop* : outil complet

Structure d’un processus

Un processus possède...

- ses instructions
- ses variables globales (mémoire statiques)
- ses variables dynamiques (pile et tas)

Mémoire d’un processus 1/2



Mémoire d'un processus 2/2

Allocations des variables

- **variables globales** : zone statique, pour la durée de vie du processus
- **paramètres de fonctions** : sur la pile, pour la durée d'exécution de la fonction
- **variable de bloc** : sur la pile, pour la durée d'exécution du bloc concerné
- **variables dynamiques** : sur le tas, allouée au *malloc*, désallouée au *free*.

Gestion de processus en C

à inclure

#include <sys/types.h>

#include <unistd.h>

#include <stdlib.h>

Accéder aux informations du processus

- *pid_t getpid(void);*
- *pid_t getppid(void);*
- *uid_t getuid(void);*

Gestion

- *int sleep(int duree);*
- *void exit(int code);*

fork / wait

Création d'un processus

fork

La fonction *fork* permet de créer un nouveau processus depuis un programme. Ce nouveau processus est :

- la copie exacte du processus parent
- un processus concurrent

syntaxe

pid_t fork(void);

Copie exacte

Le fils hérite

- le même code
- la même zone de données
- l'environnement du processus parent (répertoire de travail)
- la même priorité
- les propriétaires
- les descripteurs de fichiers ouverts
- les comportements vis à vis des signaux

Les différences

- le PID et le PPID
- la valeur de retour du *fork*

Distinguer le père et le fils

Valeur de retour du fork

- **dans le fils** : 0
- **dans le processus parent** : PID du fils créé

Exemple

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
    pid_t p;
    if ((p=fork())<0 ){
        perror("echec creation processus");
        exit(1);
    }

    if(p==0){
        printf("Dans le fils ! PID = %d\n", getpid());
        printf("Valeur de p dans le fils : %d\n", p);
    }
    else{
        printf("Dans le père ! PID = %d\n", getpid());
        printf("Valeur de p dans le père : %d\n", p);
    }

    return 0;
}
```

Bonne pratique

Attention !
On teste **toujours** la valeur de retour d'un appel système.

Ordonnancement

Attention
Les mécanismes de priorité et de préemption sur les processus ne donnent aucune garantie de l'ordre d'exécution. L'ordre des messages peut varier.

Adoption
Puisque tous les processus ont un parent, que se passe t-il si le père se termine avant le fils ?

Solution

synchronisation de processus

#include <sys/wait.h>

pid_t wait(int status)*

principe

- provoque la suspension du processus appelant jusqu'à ce que l'un de ses fils se termine
- retourne -1 si pas de fils
- retourne le PID du fils terminé sinon

Alternative

pid_t waitpid(int status, pid_t pid)*

Exemple parfait

```c
int main(){
    pid_t p;
    if ((p=fork())<0 ){
        perror("echec creation processus");exit(1);
    }

    if(p==0){
        printf("Dans le fils ! PID = %d\n", getpid());
        printf("Valeur de p dans le fils : %d\n", p);
    }
    else{
        pid_t pid;
        printf("Dans le père ! PID = %d\n", getpid());
        printf("Valeur de p dans le père : %d\n", p);
        if ((pid=wait(NULL))<0){
            perror("echec wait");exit(2);
        }
        printf("Fin du fils %d\n", pid);
    }

    return 0;
}
```

Exercices

1 - Petit-fils

Donnez le code d'un programme qui crée un fils, et dont le fils crée un petit fils. Attention à la concurrence et à la synchronisation.

2 - Famille nombreuse

Donnez le code d'un programme qui crée un 2 fils. Attention à la concurrence et à la synchronisation.

Appels systèmes d’entrée/sortie

Ouverture d'un fichier

Principe
Réserver une entrée libre dans la table des fichiers et récupérer son **descripteur** (index).

Pour tous les processus
Au moins trois descripteurs de bases :

- 0 : entrée standard (clavier)
- 1 : sortie standard (console)
- 2 : sortie erreur standard (console, mais en rouge)

Ouverture d’un fichier

syntaxe

int open(char ref, int mode)*

Détails

- Renvoie le descripteur du fichier
- mode : 0 pour lecture, 1 pour écriture, 2 pour lectur/écriture
- le fichier doit exister, sinon erreur (-1 retourné)
- les droits d’accès doivent être compatibles, sinon erreur
- pointeur de fichier sur le 1er caractère à l’ouverture
- ouverture en écriture non destructive

Création d'un fichier

syntaxe

int creat(char ref, int mode)*

Détails

- Renvoie le descripteur du fichier créé
- mode : définit le droit d'accès
- -1 si échec
- si le fichier existe, il est remis à 0 caractères, mais ses droits sont conservés

lecture d'un fichier

syntaxe

*int read(int desc, char *buf, int n)*

Détails

- *desc* : descripteur du fichier à lire
- *buf* : tableau de caractère où stocker ce qui est lu
- *n* : nombre de caractères à lire à partir du caractère pointé
- retourne -1 si échec
- retourne le nombre de caractères lus sinon (0 si fin de fichier)

écriture dans un fichier

syntaxe

*int write(int desc, char *buf, int n)*

Détails

- *desc* : descripteur du fichier dans lequel écrire
- *buf* : tableau de caractère contenant ce qu'il faut écrire
- *n* : nombre de caractères à écrire depuis *buf*
- retourne -1 si échec
- retourne le nombre de caractères écrits sinon

fermeture d'un fichier

syntaxe

int close(int desc)

Détails

- *desc* : descripteur du fichier à fermer
- tous les descripteurs sont automatiquement fermés à la fin d'un processus
- bonne pratique : fermer un fichier dès qu'on a fini de l'utiliser

Exercice

cp

Écrire un programme qui simule la commande *cp* et copie le premier fichier passé en paramètre du programma dans le deuxième.

Communications entre processus

Définition

tube
Moyen de communication entre deux processus. Il s'agit en fait d'un fichier utilisé comme file (FIFO).

Particularités

- utilise des descripteurs de fichier...
- ...mais qui n'ont pas de nom dans le système
- taille limitée mais gérée par le système
- gestion de type producteur/consommateur

Création d'un tube

syntaxe

int pipe(int p[2])

Détails

- Déclare deux descripteurs : p[0] pour la lecture, p[1] pour l'écriture
- retourne 0 si le tube est créé
- retourne -1 en cas d'échec

Lecture et écriture dans un pipe

Principe

- *read* pour lire dans un pipe
- *write* pour y écrire

Héritage

Pour que deux processus partagent le même tube, il faut qu'ils partagent les mêmes descripteurs pour lui.

- père/fils
- fils/fils

Attention

Il faut fermer les descripteurs manuellement.

Fonctionnement 1/2

Le système s'assure que

- si un processus tente d'écrire dans un tube alors que plus aucun processus n'est en mesure d'y lire, le processus reçoit le signal SIGPIPE (interruption du processus)
- le processus est suspendu si lecture dans un tube vide ou écriture dans un tube plein
- on peut faire la différence entre un tube provisoirement vide et un tube définitivement vide

Fonctionnement 2/2

read

- bloquant si des processus peuvent encore écrire (suspend le processus appelant)
- retourne 0 si plus aucun descripteur en écriture ouvert (non-bloquant)

Conséquences

Il faut **toujours** fermer les descripteurs quand on a fini de les utiliser.

Exemple

```c
int main(){
    pid_t p;
    int tube[2], i;

    if (pipe(tube)<0){
        perror("echec tube"); exit(1);
    }
    if ((p=fork())<0 ){
        perror("echec creation processus");exit(1);
    }

    if(p==0){
        if ((close(tube[0]))<0) {perror("fermeture tube");exit(1);}
        printf("Entrer un entier\n");
        scanf("%d", &i);
        if ((write(tube[1], &i, sizeof(int)))<0) exit(1);
        if ((close(tube[1]))<0) {perror("fermeture tube");exit(1);}
    }
    else{
        pid_t pid;
        if ((close(tube[1]))<0) {perror("fermeture tube");exit(1);}
        if ((read(tube[0], &i, sizeof(int)))<0) exit(1);
        printf("Entier reçu du fils : %d\n", i);
        if ((close(tube[0]))<0) {perror("fermeture tube");exit(1);}
        if ((pid=wait(NULL))<0){
            perror("echec wait");exit(2);
        }
    }
    return 0;
}
```

Exercice

Adaptation
Adapter l'exercice précédent pour que le fils lise et transmettent une chaîne de caractère de taille *maximum* 50 caractères.

Programmation multithreadée

Côté processeur

Deux notions différentes
Un processeur moderne peut être :

- **multi-cœur** : il peut exécuter plusieurs processus en parallèle car il possède plusieurs centre de calcul distincts (les cœurs)
- **multi-thread** : chaque cœur peut exécuter plusieurs *threads* en même temps, c'est à dire plusieurs flots d'éxécution.

Définitions

Thread

Un *thread* est un fil d'exécution interne à un processus. Il ne s'agit pas d'un processus différent, car tous les *threads* issus d'un même processus partagent les mêmes espaces d'adressage ou table de fichiers.

Différences

Un nouveau *thread* possède son propre compteur ordinal (pointeur d'instruction), et s'exécute de façon concurrente des autres *threads*.

Caractéristiques

Avantages

- permet de traiter des requêtes en parallèle
- augmente la réactivité
- libère du temps de traitement pour d'autre tâche

Inconvénients

- plus lourd à programmer
- consomme plus de mémoire

La bibliothèque *pthread*

Syntaxe

#include <pthread.h>

Généralités

- Les fonctions suivent la nomenclature *pthread_x*, avec *x* l'opération à réaliser (création, terminaison,...)
- Si la fonction porte sur un objet particulier, cela sera précisé entre *pthread* et l'opération
- les *thread* ont un identifiant : *pthread_t TID*

Création d'un thread 1/2

Principe

- La création d'un *thread* renvoie un TID
- Il faut donner un point d'entrée au *thread* : l'adresse d'une fonction
- Un *thread* possède un attribut :
  - **joignable** : le processus appelant peut se synchroniser à la terminaison du *thread*
  - **détaché** : pas de synchronisation possible

Création d'un thread 2/2

Syntaxe

*int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void*(start_routine)(void*), void* arg)*

Détails

- *pthread_t *thread* : pointeur pour récupérer le TID du *thread* qui sera créé
- *const pthread_attr_t *attr* : attribut du thread, ou NULL (par défaut)
- *void*(start_routine)(void*)* : fonction de type *void** prenant un seul paramètre de type *void**; ça sera le point d'entrée du *thread*
- *void* arg* : le paramètre pour la fonction *start_routine*

Attribut d'un thread

Valeurs possibles

- *PTHREAD_CREATE_JOIGNABLE* : synchronisation possible ; comportement par défaut
- *PTHREAD_CREATE_DETACHED* : pas de synchronisation possible

Synchronisation

syntaxe
*int pthread_join(pthread_t tid, void **status)*

Détails

- *tid* : identifiant du *thread* avec lequel on veut synchroniser la terminaison.
- *status* : valeur de retour du *thread* synchronisé (renvoyée par *pthread_exit*)

Attention
C'est le *pthread_join* qui cause la libération des ressources mémoire du *thread* joignable. Il doit donc impérativement y avoir un *join* pour chaque *thread* joignable pour éviter les fuites mémoire.

Terminaison

syntaxe

*int pthread_exit(void *status)*

Détails

- termine le *thread* appelant avec le code de retour spécifié
- différent du *exit* classique (qui termine un processus, et donc tous ses *threads* associés).

Exemple

```c
void* hello(void* arg){
    int num = *((int*) arg);
    printf("hello depuis le thread %d\n", num);
    return NULL;
}

int main(){
    pthread_t tid1, tid2;
    int arg1=1, arg2=2;

    pthread_create(&tid1, NULL, hello, (void*)&arg1);
    pthread_create(&tid2, NULL, hello, (void*)&arg2);

    printf("Attente de la fin du thread 1...\n");
    pthread_join(tid1, NULL);
    printf("Fin du thread 1.\n");
    printf("Attente de la fin du thread 2...\n");
    pthread_join(tid2, NULL);
    printf("Fin du thread 2.\n");

    return 0;
}
```

Attention

Espace partagé
Les *threads* partagent le même espace de variables. Si on utilisait une seule variable *arg* pour les deux, aucune garantie n'est donnée sur la valeur qui sera lu.

Exemple à ne pas faire

```c
void* hello(void* arg){
    int num = *((int*) arg);
    printf("hello depuis le thread %d\n", num);
    return NULL;
}

int main(){
    pthread_t tid1, tid2;

    int arg=1;
    pthread_create(&tid1, NULL, hello, (void*)&arg);
    arg = 2;
    pthread_create(&tid2, NULL, hello, (void*)&arg);

    printf("Attente de la fin du thread 1...\n");
    pthread_join(tid1, NULL);
    printf("Fin du thread 1.\n");
    printf("Attente de la fin du thread 2...\n");
    pthread_join(tid2, NULL);
    printf("Fin du thread 2.\n");

    return 0;
}
```

Threads détachés

Principe

On souhaite créé un *thread* qui ne sera pas synchronisé avec le *thread* principal. Cela implique d'être très vigilant sur l'utilisation de la mémoire.

Détails

- Il faut explicitement créer un attribut *pthread_attr* pour spécifier détaché
- Le *thread* principal peut se terminer avant les autres (absence de synchro) donc *pthread_exit* impératif dans le *thread* principal
- De fait, il faut s'assurer que les variables passées en arguments des *threads* existent toujours quand ceux-ci s'exécutent

Exemple 1/2

```c
int main(){
    pthread_t tid1, tid2;

    int *arg1 = (int*)malloc(sizeof(int));
    int *arg2 = (int*)malloc(sizeof(int));
    *arg1=1; *arg2=2;

    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    pthread_create(&tid1, &attr, hello, (void*)arg1);
    pthread_create(&tid2, &attr, hello, (void*)arg2);

    pthread_exit(NULL);

    return 0;
}
```

Exemple 2/2

```
 5 void* hello(void* arg){
 6     int num = *((int*) arg);
 7     free(arg);
 8     printf("hello depuis le thread %d\n", num);
 9     return NULL;
10 }
```

Gestion du partage des ressources

Principe

Cas pratique

- On dispose d'une ressource limitée
- Pour l'utiliser, un *thread* teste si elle est disponible
- Si oui, il l'utilise

Problème

Si le *thread* est préempté entre le test et l'utilisation par un autre *thread* qui utilise la ressource, le test n'est plus valable.

Solution

Exclusion mutuelle
On crée des **sections critiques** dans le code : accessible uniquement par un *thread* à la fois

Mise en œuvre
On utilise pour cela des **sémaphores** (*mutex* en anglais).

Principe d'un sémaphore

Deux opérations

- P(S) : on réserve le sémaphore juste avant d'accéder à la section critique, on attend s'il n'est pas disponible
- V(S) : on libère le sémaphore en sortie de section critique, et on débloque un *thread* en attente s'il y en a

Conditions nécessaires

Les primitives *P* et *V* doivent être atomiques : on ne peut pas préempter au milieu d'une réservation ou d'une libération.

Interblocage

Problème
Imaginons deux sections critiques gérées par des sémaphores *S1* et *S2* et deux *threads* concurrents.

Thread 1 :

```
P(S1);
P(S2);
...
V(S1);
V(S2);
```

Thread 2 :

```
P(S2);
P(S1);
...
V(S2);
V(S1);
```

Solutions à l'interblocage

- Opérations *P* dans le même ordre
- Limitation des sections critiques au minimum

Les sémaphores de la bibliothèque pthread

Syntaxe

pthread_mutex_t m ;

Fonctions

- *int pthread_mutex_init(pthread_mutex_t *m, const pthread_mutex_attr *attr)*
  - *m* : mutex à initialiser
  - *attr* : mode d'initialisation (NULL par défaut)
- *int pthread_mutex_lock(pthread_mutex_t *m)*
  - *m* : mutex à verrouiller si possible (suspend le processus sinon)
- *int pthread_mutex_unlock(pthread_mutex_t *m)*
  - *m* : mutex à déverrouiller

Exemple 1/2

Soit la fonction suivante pour réserver des places pour un événements.

```
 5 #define MAX 25
 6 int res=0;
 7
 8 void* reserver(void* arg){
 9     int demande = *((int*) arg);
10     if (res+demande<MAX)
11         res += demande;
12     printf("réservations totales : %d\n", res);
13     return NULL;
14 }
```

Exemple 2/2

```
16 int main(){
17     pthread_t tid1, tid2;
18 
19     int arg1, arg2;
20     printf("Entrer les demandes de réservations\n");
21     scanf("%d", &arg1); scanf("%d", &arg2);
22 
23     pthread_create(&tid1, NULL, reserver, (void*)&arg1);
24     pthread_create(&tid2, NULL, reserver, (void*)&arg2);
25 
26     printf("Attente de la fin du thread 1...\n");
27     pthread_join(tid1, NULL);
28     printf("Fin du thread 1.\n");
29     printf("Attente de la fin du thread 2...\n");
30     pthread_join(tid2, NULL);
31     printf("Fin du thread 2.\n");
32 
33     printf("Réservations à la fin du thread principal : %d\n", res);
34 
35     return 0;
36 }
```

Utiliser des mutex pour créer une section critique où cela est nécessaire.

Exercice

Producteur/consommateur
Écrire un programme qui va instancier une variable globale et créer deux *threads* :

- un **producteur** qui produit une valeur aléatoire entre 0 et 100
- un **consommateur** qui va lire cette valeur aléatoire et l'afficher *aussitôt* qu'elle est disponible

La variable globale doit être de type *int*, et affichée le plus tôt possible par le consommateur. Le processus production/consommation se répétera 10 fois. Le consommateur ne doit manquer aucune valeur produite.