

Programmation Concurrente

Synchronisation

**Section Critique, Exclusion Mutuelle, Primitives de
Synchronisation**

Incrémantation concurrente d'un compteur

Section Critique

```
9 void *inc(void *arg) {  
10    unsigned int i;  
11    int k=atoi((char *)arg);  
12    printf("Thread %d lance\n",k);  
13    for (i=0;i<n;i++) compteur=compteur+1;  
14 }  
15 }
```

Thread 1

```
22    movl  $0, -20(%rbp)  
23    LBB0_1:  
24    movq  _n@GOTPCREL(%rip), %rax  
25    movl  -20(%rbp), %ecx  
26    cmpl  (%rax), %ecx  
27    jae  LBB0_4  
28    ## %bb.2:  
29    movl  _compteur(%rip), %eax  
30    addl  $1, %eax  
31    movl  %eax, _compteur(%rip)  
32    ## %bb.3:  
33    movl  -20(%rbp), %eax  
34    addl  $1, %eax  
35    movl  %eax, -20(%rbp)  
36    jmp  LBB0_1  
37    LBB0_4:  
38    movq  -8(%rbp), %rax
```

Thread 2

```
22    movl  $0, -20(%rbp)  
23    LBB0_1:  
24    movq  _n@GOTPCREL(%rip), %rax  
25    movl  -20(%rbp), %ecx  
26    cmpl  (%rax), %ecx  
27    jae  LBB0_4  
28    ## %bb.2:  
29    movl  _compteur(%rip), %eax  
30    addl  $1, %eax  
31    movl  %eax, _compteur(%rip)  
32    ## %bb.3:  
33    movl  -20(%rbp), %eax  
34    addl  $1, %eax  
35    movl  %eax, -20(%rbp)  
36    jmp  LBB0_1  
37    LBB0_4:  
38    movq  -8(%rbp), %rax
```

Exclusion Mutuelle

Solution Logicielle, Attente Active

```
6 int jeton=0;
7 unsigned int n;
8 unsigned int compteur=0;
9
10 void *inc(void *arg) {
11     unsigned int i=0;
12     int k=atoi((char *)arg);
13
14     printf("Thread %d lance\n",k);
15
16     while (i<n) {
17         while (jeton==1-k);           // E_k
18         compteur=compteur+1;        // C_k
19         jeton=1-k;                  // S_k
20         i=i+1;                      // R_k
21     }
22 }
```

Preuve par automate :

- Action : E_k, C_k, S_k, R_k
- Etat : <0/1,0/1,0/1>

Exclusion Mutuelle

Primitive de synchronisation, les mutex

```
pthread_mutex_t mutex;  
  
int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr);  
  
int pthread_mutex_lock(pthread_mutex_t *mutex);  
  
int pthread_mutex_unlock(pthread_mutex_t *mutex);  
  
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5
6 unsigned int n;
7 pthread_mutex_t m;
8 unsigned int compteur=0;
9
10 void *inc(void *arg) {
11     unsigned int i;
12     int k=atoi((char *)arg);
13
14     printf("Thread %d lance\n",k);
15
16     for (i=0;i<n;i++) {
17         pthread_mutex_lock(&m);
18         compteur=compteur+1;
19         pthread_mutex_unlock(&m);
20     }
21     return (void *) 0;
22 }
23
24 int main(int argc,char *argv[]) {
25     int i,j;
26     pthread_t t1, t2;
27     if (argc!=2) {
28         fprintf(stderr,"usage %s n (n>0)\n",*argv);
29         exit(2);
30     }
31     pthread_mutex_init(&m,NULL);
32
33     n=atoi(argv[1]);
34     printf("sizeof(int)=%lu\n",sizeof(int));
35     pthread_create(&t1,NULL,inc,"1");
36     pthread_create(&t2,NULL,inc,"2");
37     pthread_join(t1,NULL);
38     pthread_join(t2,NULL);
39     pthread_mutex_destroy(&m);
40     printf("compteur=%u\n",compteur);
41 }
42
```

Variables de condition

Mise en attente

```
pthread_cond_t cond;  
  
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);  
  
int pthread_cond_signal(pthread_cond_t *cond);  
  
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);  
  
int pthread_cond_destroy(pthread_cond_t *cond);
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

int done;
pthread_cond_t c;
pthread_mutex_t m;

void * foo() {
    sleep(rand()%5);
    pthread_mutex_lock(&m);
    printf("Je verifie si done est à 1\n");
    while (!done) {
        printf("J'attends\n");
        pthread_cond_wait(&c,&m);
        printf("Je verifie si done est à 1\n");
    }
    printf("foo\n");
    pthread_mutex_unlock(&m);
    return NULL;
}

void *bar() {
    sleep(rand()%5);
    pthread_mutex_lock(&m);
    done=1;
    printf("bar\n");
    pthread_cond_signal(&c);
    printf("J ai prévenu que done est modifié\n");
    sleep(5);
    pthread_mutex_unlock(&m);
    return NULL;
}

int main(int argc,char *argv[]) {
    pthread_t t1, t2;

    time_t t;

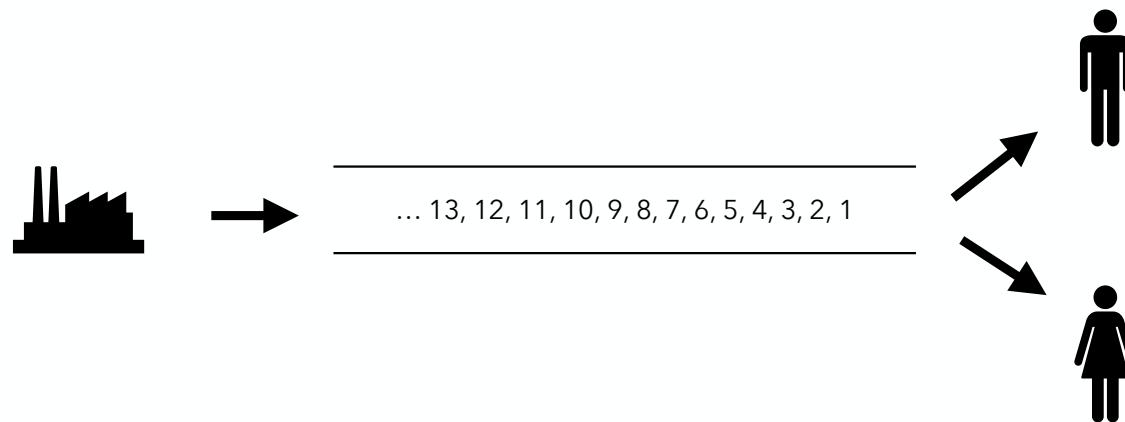
    srand((unsigned) time(&t));

    pthread_mutex_init(&m,NULL);
    pthread_cond_init(&c,NULL);

    pthread_create(&t1,NULL,foo,NULL);
    pthread_create(&t2,NULL,bar,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
}
```

Producteur → Consommateur

Problème classique de synchronisation



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5
6 #define N 10
7
8 int t[N]; //tableau pour la liste circulaire
9 int d, //début de la liste circulaire
10 | f; //fin de la liste circulaire
11 int n; // nombre d'éléments dans la liste circulaire
12
```

```
13 void * prod() {
14     int compteur=1;
15     while (1) {
16         while (n==N); // la liste est pleine
17         f=(f+1)%N;
18         t[f]=compteur++;
19         sleep(rand()%2);
20         n=n+1;
21     }
22 }
```

```
23 void *cons() {
24     while(1){
25         while (n==0); //la liste est vide
26         printf("%d ",t[d]);
27         d=(d+1)%N;
28         sleep(rand()%2);
29         n=n-1;
30     }
31 }
32 }
```

```
36 int main(int argc,char *argv[]) {
37     pthread_t t1, t2, t3;
38     time_t t;
39
40     srand((unsigned) time(&t));
41
42     setvbuf(stdout,NULL, _IONBF, 0);
43
44     pthread_create(&t1,NULL,prod,NULL);
45     pthread_create(&t2,NULL,cons,NULL);
46     pthread_create(&t3,NULL,cons,NULL);
47     pthread_join(t1,NULL);
48     pthread_join(t2,NULL);
49     printf("Fin\n");
50 }
```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5
6 #define N 10
7
8 int t[N]; //tableau pour la liste circulaire
9 int d, //début de la liste circulaire
10 f; //fin de la liste circulaire
11 int n; // nombre d'éléments dans la liste circulaire
12
13 pthread_cond_t c1, c2; // variable de condition pour les prod et les cons
14 pthread_mutex_t m; // mutex
15 int waiting_prod, waiting_cons; // nombre de prod et cons en attente

17 void * prod() {
18     int compteur=1, waiting=0;
19     while (1) {
20         pthread_mutex_lock(&m);
21         if(n != N){
22             t[f]=compteur++;
23             f=(f+1)%N;
24             sleep(rand()%2);
25             n=n+1;
26             if (waiting_cons)
27                 pthread_cond_signal(&c2);
28         }
29         else{
30             waiting=1; waiting_prod++;
31             pthread_cond_wait(&c1, &m);
32         }
33         waiting_prod-=waiting; waiting-=waiting;
34         pthread_mutex_unlock(&m);
35     }
36 }
37 }

39 void * cons() {
40     int waiting=0;
41     while(1){
42         pthread_mutex_lock(&m);
43         if(n != 0){
44             printf("%d ",t[d]);
45             d=(d+1)%N;
46             sleep(rand()%2);
47             n=n-1;
48             if(waiting_prod)
49                 pthread_cond_signal(&c1);
50         }
51         else{
52             waiting=1; waiting_cons++;
53             pthread_cond_wait(&c2, &m);
54         }
55         waiting_cons-=waiting; waiting-=waiting;
56     }
57 }
58 }

59

59
60 int main(int argc,char *argv[]){
61     pthread_t t1, t2, t3;
62     time_t t;
63
64     srand((unsigned) time(&t));
65
66     setvbuf(stdout,NULL, _IONBF, 0);
67
68     pthread_mutex_init(&m, NULL);
69     pthread_cond_init(&c1, NULL);
70     pthread_cond_init(&c2, NULL);
71
72     pthread_create(&t1,NULL,prod,NULL);
73     pthread_create(&t2,NULL,cons,NULL);
74     pthread_create(&t3,NULL,cons,NULL);
75     pthread_join(t1,NULL);
76     pthread_join(t2,NULL);
77     pthread_join(t3,NULL);
78     printf("Fin\n");
79 }

```

Les Sémaphores

Solution historique (Dijkstra) : compteur bloquant

```
#include <semaphore.h>

sem_t sem;

int sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_wait(sem_t *sem); // bloque si la valeur==0, sinon décrémenter la valeur

int sem_post(sem_t *sem); // incrémenter la valeur, et débloquer un thread

int sem_getvalue(sem_t *sem, int * value); // valeur du sémaphore

int sem_destroy(sem_t *sem);
```

Équivalence mutex et séaphore

pthread_mutex_t m;

sem_t m;

pthread_mutex_init(&m, NULL);

sem_init(&m, 0, 1);

pthread_mutex_lock(&m);

sem_wait(&m);

pthread_mutex_unlock(&m);

sem_post(&m);

pthread_mutex_destroy(&m);

sem_destroy(&m);

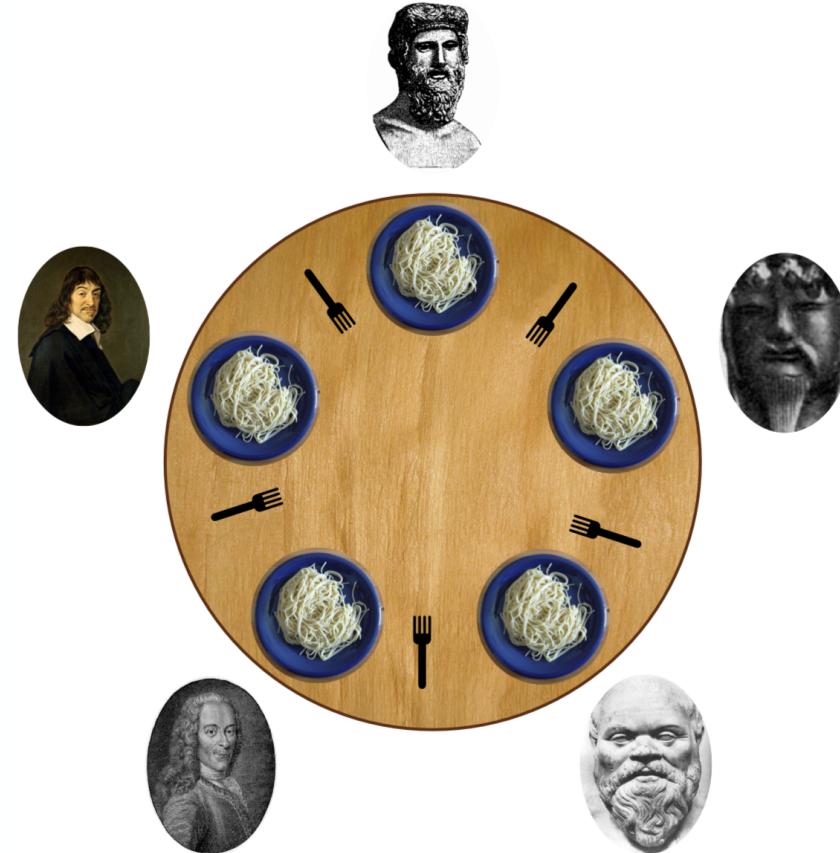
```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5 #include <semaphore.h>
6
7 #define N 10
8
9 int t[N]; //tableau pour la liste circulaire
10 int d, //début de la liste circulaire
11     f; //fin de la liste circulaire
12 int n; // nombre d'éléments dans la liste circulaire
13
14 sem_t m, // mutex
15     vide, // nombre de places libres
16     plein; // nombre de places occupées
17
18
19 void * prod() {
20     int compteur=1;
21     while (1) {
22         sem_wait(&vide); // la liste est pleine ?
23         sem_wait(&m);
24         t[f]=compteur++;
25         f=(f+1)%N;
26         sleep(rand()%2);
27         n=n+1;
28         sem_post(&m);
29         sem_post(&plein);
30     }
31 }
32
33 void *cons() {
34     while(1){
35         sem_wait(&plein); //la liste est vide ?
36         sem_wait(&m);
37         printf("%d ",t[d]);
38         d=(d+1)%N;
39         sleep(rand()%2);
40         n=n-1;
41         sem_post(&m);
42         sem_post(&vide);
43     }
44 }
45
46 int main(int argc,char *argv[]) {
47     pthread_t t1, t2, t3;
48     time_t t;
49
50     srand((unsigned) time(&t));
51
52     setvbuf(stdout,NULL, _IONBF, 0);
53
54     sem_init(&m,0,1);    // mutex pour l'accès aux variables globales
55     sem_init(&vide,0,N); // la liste est plein de vide !-
56     sem_init(&plein,0,0); // la liste est vide de plein !-
57
58     pthread_create(&t1,NULL,prod,NULL);
59     pthread_create(&t2,NULL,cons,NULL);
60     pthread_create(&t3,NULL,cons,NULL);
61
62     pthread_join(t1,NULL);
63     pthread_join(t2,NULL);
64     pthread_join(t3,NULL);
65     printf("Fin\n");
66 }
67

```

Le diner des philosophes

Notion d'interblocage (deadlock)



```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5
6
7 // b est un tableau qui indique si la baguette i (de 0 à 4) est libre (-1) ou non (>-1)
8 int b[]={-1,-1,-1,-1,-1};
9
10 void prendre_baguettes(int i) {
11     while (b[i]!=-1 && b[i]!=-1);
12     b[i]=-1;
13     sleep(rand()%2);
14     while (b[(i+1)%5]!=-1 && b[(i+1)%5]!=-1);
15     b[(i+1)%5]=-1;
16 }
17
18
19 void poser_baguettes(int i) {
20     if (b[i]==-1) b[i]=-1;
21     sleep(rand()%2);
22     if (b[(i+1)%5]==-1) b[(i+1)%5]=-1;
23 }
24
25
26 void *philo(void *arg) {
27     int i=*(int *)arg;
28     while (1) {
29         printf("(%)t je pense\n",i);
30         sleep(0);
31         printf("(%)v je vais prendre les baguettes\n",i);
32         prendre_baguettes(i);
33         printf("(%)p j'ai pris les baguettes\n",i);
34         printf("(%)m je mange\n",i);
35         sleep(0);
36         printf("(%)r j'ai fini de manger, je pose les baguettes\n",i);
37         poser_baguettes(i);
38     }
39     return NULL;
40 }
41

```

```

41
42 int main(int argc,char **argv, char **arge) {
43     pthread_t t[5];
44     int num[]={0,1,2,3,4};
45     time_t h;
46     int i;
47
48     srand((unsigned) time(&h));
49
50     setvbuf(stdout,NULL, _IONBF, 0);
51
52     for (i=0;i<5;i++)
53         pthread_create(t+i,NULL,philo,num+i);
54     for (i=0;i<5;i++)
55         pthread_join(t[i],NULL);
56
57 }
58

```

Un mutex/fourchette (deadlock)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5
6
7 // b est un tableau de mutex qui indique si la baguette i (de 0 à 4) est libre ou non
8 pthread_mutex_t b[5];
9
10 void prendre_baguettes(int i) {
11     pthread_mutex_lock(b+i);
12     sleep(rand()%2);
13     pthread_mutex_lock(b+(i+1)%5);
14 }
15
16 void poser_baguettes(int i) {
17     pthread_mutex_unlock(b+i);
18     sleep(rand()%2);
19     pthread_mutex_unlock(b+(i+1)%5);
20 }
21
22
23 void *philo(void *arg) {
24     int i=*((int *)arg);
25     while (1) {
26         printf("(%)t je pense\n",i);
27         sleep(0);
28         printf("(%)v je vais prendre les baguettes\n",i);
29         prendre_baguettes(i);
30         printf("(%)p j'ai pris les baguettes\n",i);
31         printf("(%)m je mange\n",i);
32         sleep(0);
33         printf("(%)r j'ai fini de manger, je pose les baguettes\n",i);
34         poser_baguettes(i);
35     }
36     return NULL;
37 }
38 }
```

```
40 int main(int argc,char **argv, char **arge) {
41     pthread_t t[5];
42     int num[]={0,1,2,3,4};
43     time_t h;
44     int i;
45
46     srand((unsigned) time(&h));
47
48     setvbuf(stdout,NULL, _IONBF, 0);
49
50     for (i=0;i<5;i++)
51         pthread_mutex_init(b+i,NULL);
52
53     for (i=0;i<5;i++)
54         pthread_create(t+i,NULL,philo,num+i);
55     for (i=0;i<5;i++)
56         pthread_join(t[i],NULL);
57
58 }
```

Un seul mutex

Granularité

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5
6
7 // b est un tableau qui indique si la baguette i (de 0 à 4) est libre (-1) ou non (>-1)
8 int b[]={-1,-1,-1,-1,-1};
9
10 pthread_mutex m;
11
12 void prendre_baguettes(int i) {
13     b[i]=i;
14     sleep(rand()%2);
15     b[(i+1)%5]=i;
16 }
17
18 void poser_baguettes(int i) {
19     b[i]=-1;
20     sleep(rand()%2);
21     b[(i+1)%5]=-1;
22 }
23
24
25
26 void *philo(void *arg) {
27     int i=*((int *)arg);
28     while (1) {
29         printf("(%)t je pense\n",i);
30         sleep(0);
31         printf("(%)v je vais prendre les baguettes\n",i);
32         pthread_mutex_lock(&m);
33         prendre_baguettes(i);
34         printf("(%)p j'ai pris les baguettes\n",i);
35         printf("(%)m je mange\n",i);
36         sleep(0);
37         printf("(%)r j'ai fini de manger, je pose les baguettes\n",i);
38         poser_baguettes(i);
39         pthread_mutex_unlock(&m);
40     }
41     return NULL;
42 }
43 }
```

```
45 int main(int argc,char **argv, char **arge) {
46     pthread_t t[5];
47     int num[]={0,1,2,3,4};
48     time_t h;
49     int i;
50
51     srand((unsigned) time(&h));
52
53     setvbuf(stdout,NULL, _IONBF, 0);
54     pthread_mutex_init(&m,NULL);
55     for (i=0;i<5;i++)
56         pthread_create(t+i,NULL,philo,num+i);
57     for (i=0;i<5;i++)
58         pthread_join(t[i],NULL);
59
60 }
61
```

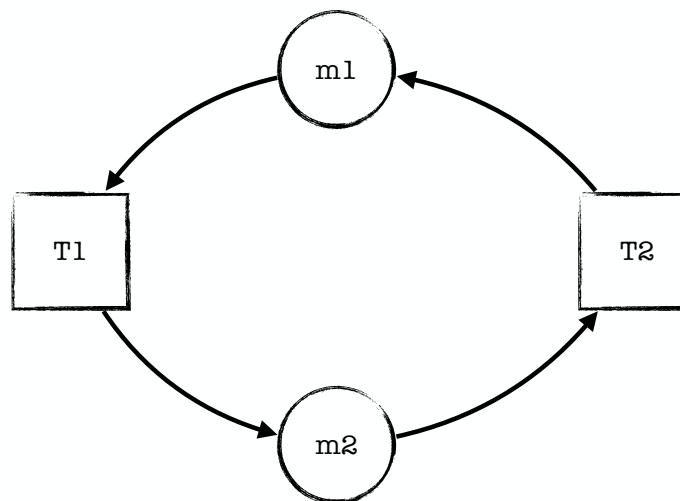
Deadlock – Interblockage

T1

Etreinte fatale

T2

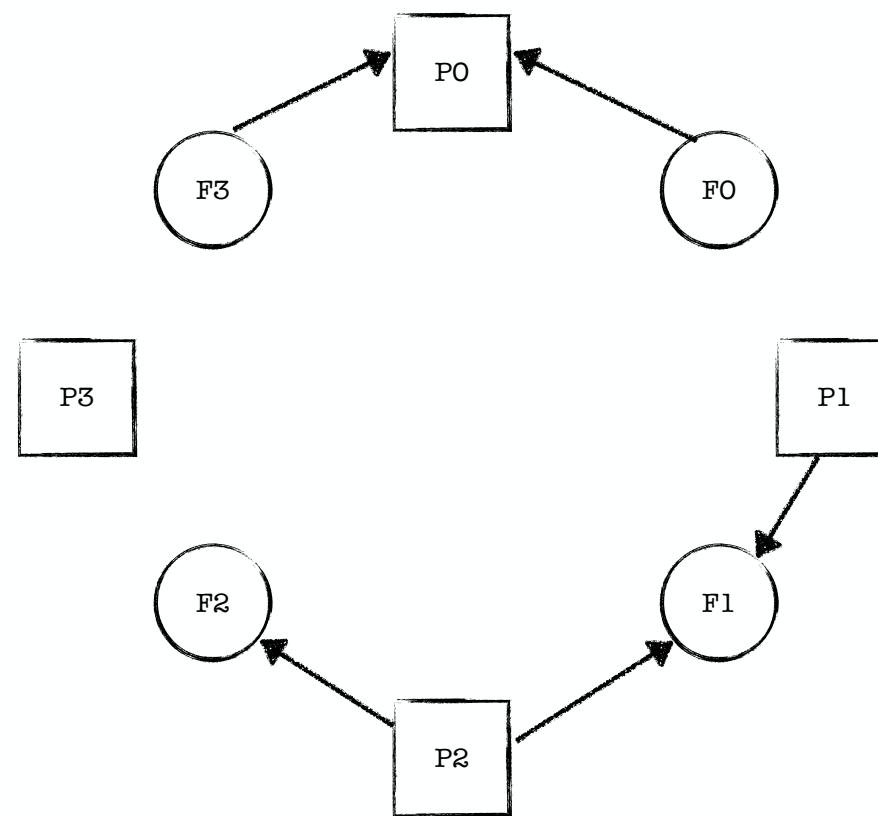
```
phtread_mutex_lock(&m1);          phtread_mutex_lock(&m2);  
.  
phtread_mutex_lock(&m2);          phtread_mutex_lock(&m1);  
.
```



Graphe d'allocation des ressources

Philosophes

Graphe d'allocation des ressources



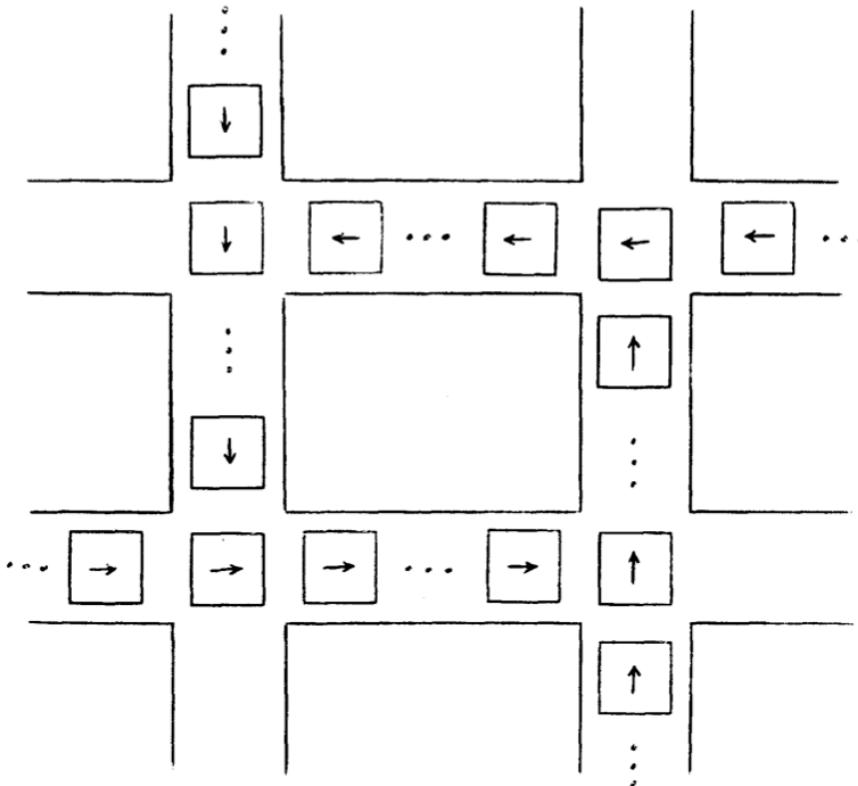
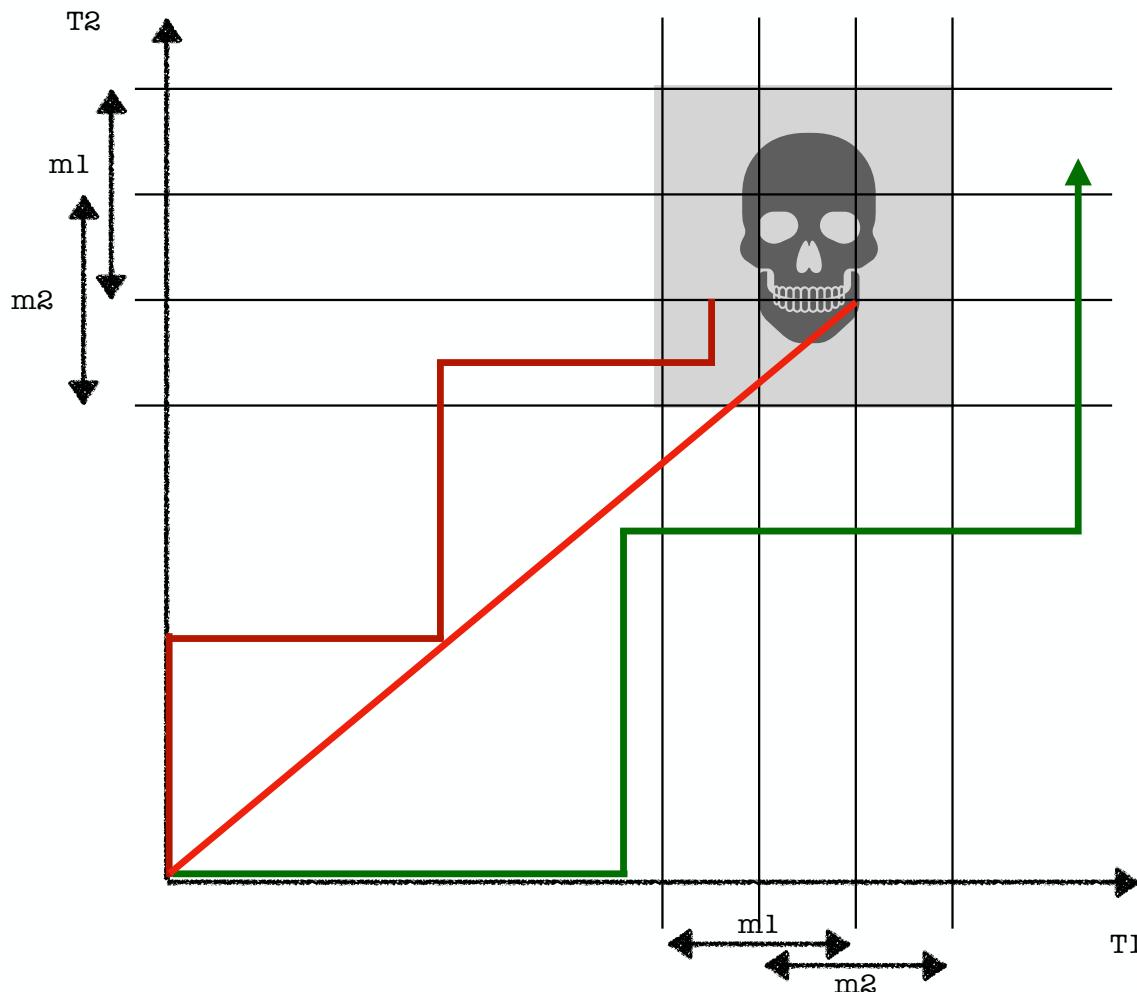


FIG. 1. Traffic deadlock.

« System Deadlock », Edward G. Coffman Jr, M. J. Elphick, Arie Shoshani, ACM Computing Survey, Volume 3, Number 1, 1971

Deadlock et ordonnancement



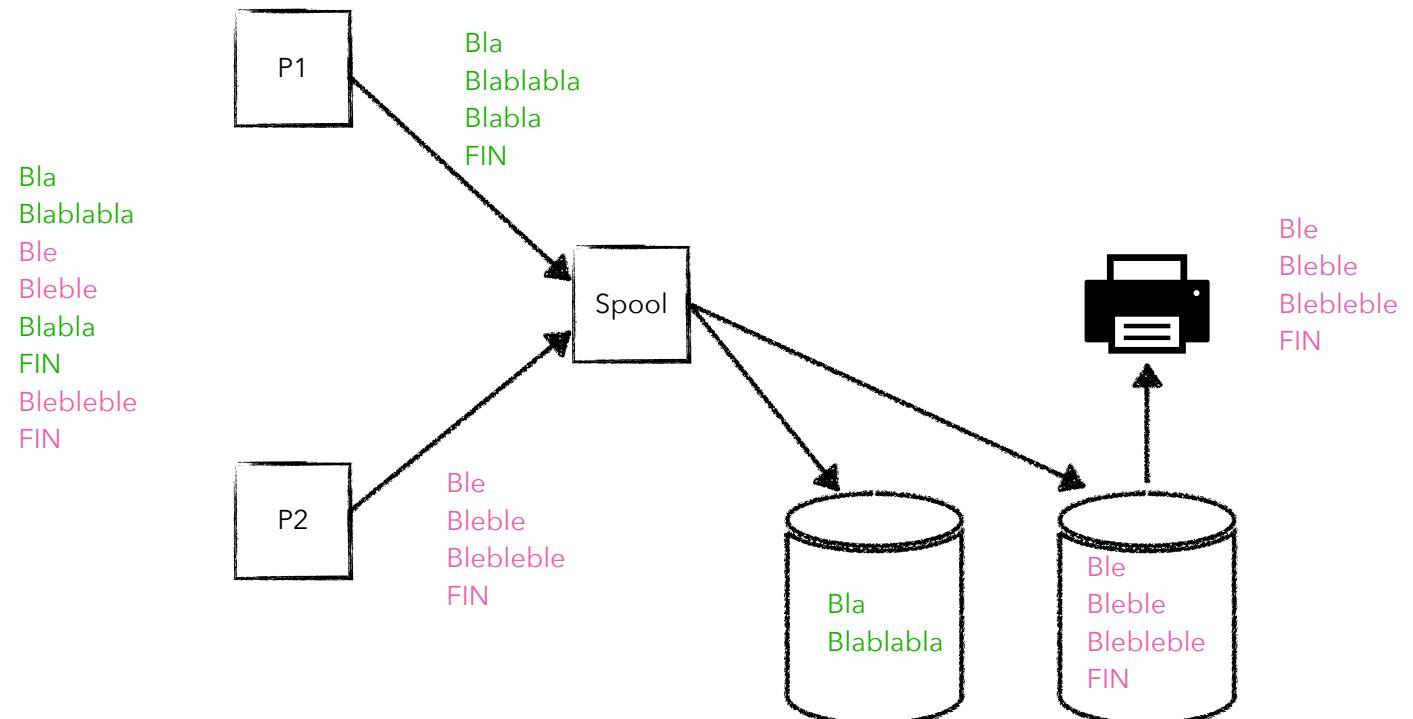
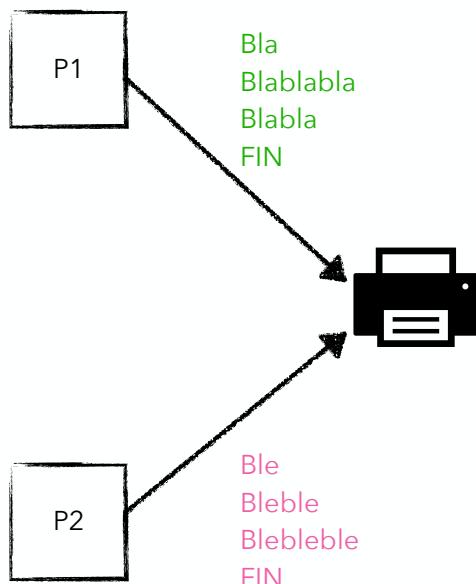
Interblocage : conditions nécessaires

1. Exclusion mutuelle : une ressource est détenue de manière exclusive
2. Détenir et attendre : les ressources sont prises une par une
3. Pas de préemption : on ne peut pas retirer une ressource allouée
4. Attente circulaire : le graphe d'allocation des ressources a un cycle

Ne pas respecter une condition => pas d'interblocage

Supprimer l'exclusion mutuelle

Exemple : le spooling (impression, fichier, IO).



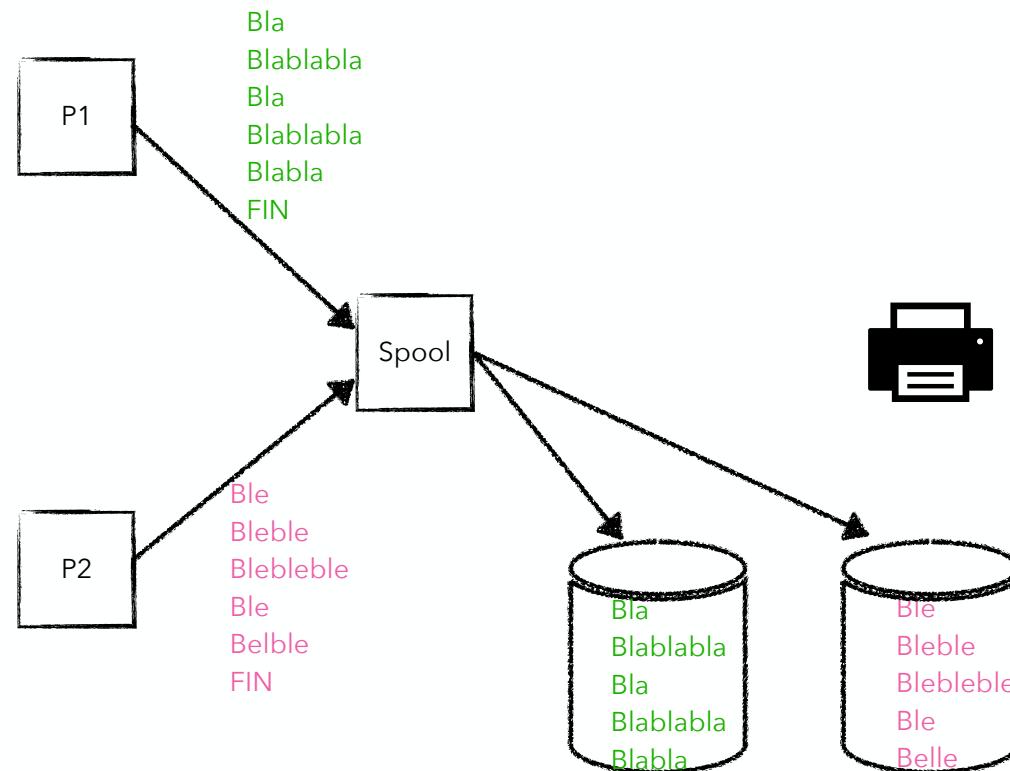
Exemple : spooler sp

Un thread dédié pour l'impression à l'écran

- Thread sp : spool qui regroupe les impressions des autres threads
- API :
 - Nouveau type `sp_t`
 - `sp_t * sp_open()` : ouverture d'une nouvelle impression, retourne un identifiant d'impression
 - `int sp_print(sp_t sp, char *s)` : chaîne qui sera à imprimer
 - `int sp_close(sp);`
- Le thread sp :
 - On enregistre pour chaque thread appelant, la liste des chaînes à imprimer à l'écran
 - À `sp_close` : le thread se réveille, affiche toutes les chaînes d'un coup, et libère la mémoire associée au thread appelant.

Spooling : temps -> espace

Deadlock possible



Ne plus tenir et attendre

1. Un mutex global : granularité
2. Relâcher si on doit attendre :
= préemption

PTHREAD_MUTEX_TRYLOCK(3) BSD Library Functions Manual PTHREAD_MUTEX_TRYLOCK(3)

NAME
`pthread_mutex_trylock` -- attempt to lock a mutex without blocking

SYNOPSIS

```
#include <pthread.h>

int
pthread_mutex_trylock(pthread_mutex_t *mutex);
```

DESCRIPTION
The `pthread_mutex_trylock()` function locks `mutex`. If the mutex is already locked, `pthread_mutex_trylock()` will not block waiting for the mutex, but will return an error condition.

RETURN VALUES
If successful, `pthread_mutex_trylock()` will return zero, otherwise an error number will be returned to indicate the error.

ERRORS
The `pthread_mutex_trylock()` function will fail if:

<code>[EINVAL]</code>	The value specified by <code>mutex</code> is invalid.
<code>[EBUSY]</code>	<code>Mutex</code> is already locked.

SEE ALSO
`pthread_mutex_destroy(3)`, `pthread_mutex_init(3)`, `pthread_mutex_lock(3)`, `pthread_mutex_unlock(3)`

STANDARDS
The `pthread_mutex_trylock()` function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

BSD July 30, 1998

Casser les cycles

Exemple : prendre les ressources dans l'ordre

```
11 // b est un tableau qui indique si la baguette i (de 0 à 4) est libre (-1) ou non (>-1)
12 pthread_mutex_t b[5];
13
14 void prendre_baguettes(int i) {
15
16     if (i<4) {
17         printf("(%)d prend la baguette de gauche\n",i);
18         pthread_mutex_lock(b+i);          // b[i]
19         printf("(%)d a pris la baguette de gauche\n",i);
20         sleep(rand()%2);
21         printf("(%)d prend la baguette de droite\n",i);
22         pthread_mutex_lock(b+(i+1)%5); // b[i+1]
23         printf("(%)d a pris la baguette de droite\n",i);
24     } else // i=4
25     {
26         printf("(%)d prend la baguette de droite\n",i);
27         pthread_mutex_lock(b+(i+1)%5); // b[0]
28         printf("(%)d a pris la baguette de droite\n",i);
29         sleep(rand()%2);
30         printf("(%)d prend la baguette de gauche\n",i);
31         pthread_mutex_lock(b+i);          // b[4]
32         printf("(%)d a pris la baguette de gauche\n",i);
33
34     }
35
36 }
37
```

Traiter les deadlocks

Systèmes de Gestion de BD (SGBD)

- Exemple : virement entre comptes bancaires
- Notion de transaction : assurer la cohérence des opérations
- Transactions concurrentes : cohérence des données