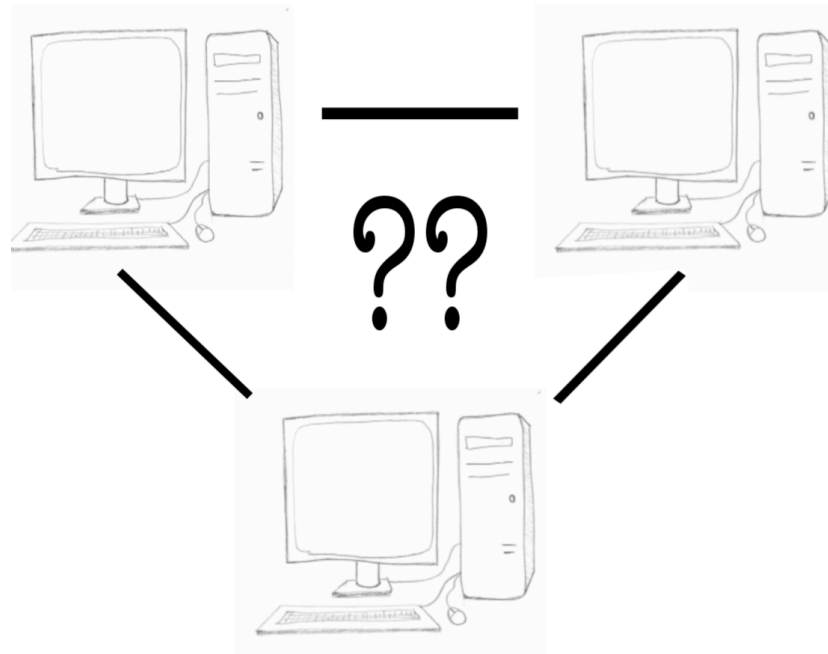


# PROGRAMMATION RÉSEAU

Arnaud Sangnier  
sangnier@irif.fr

## La concurrence en C



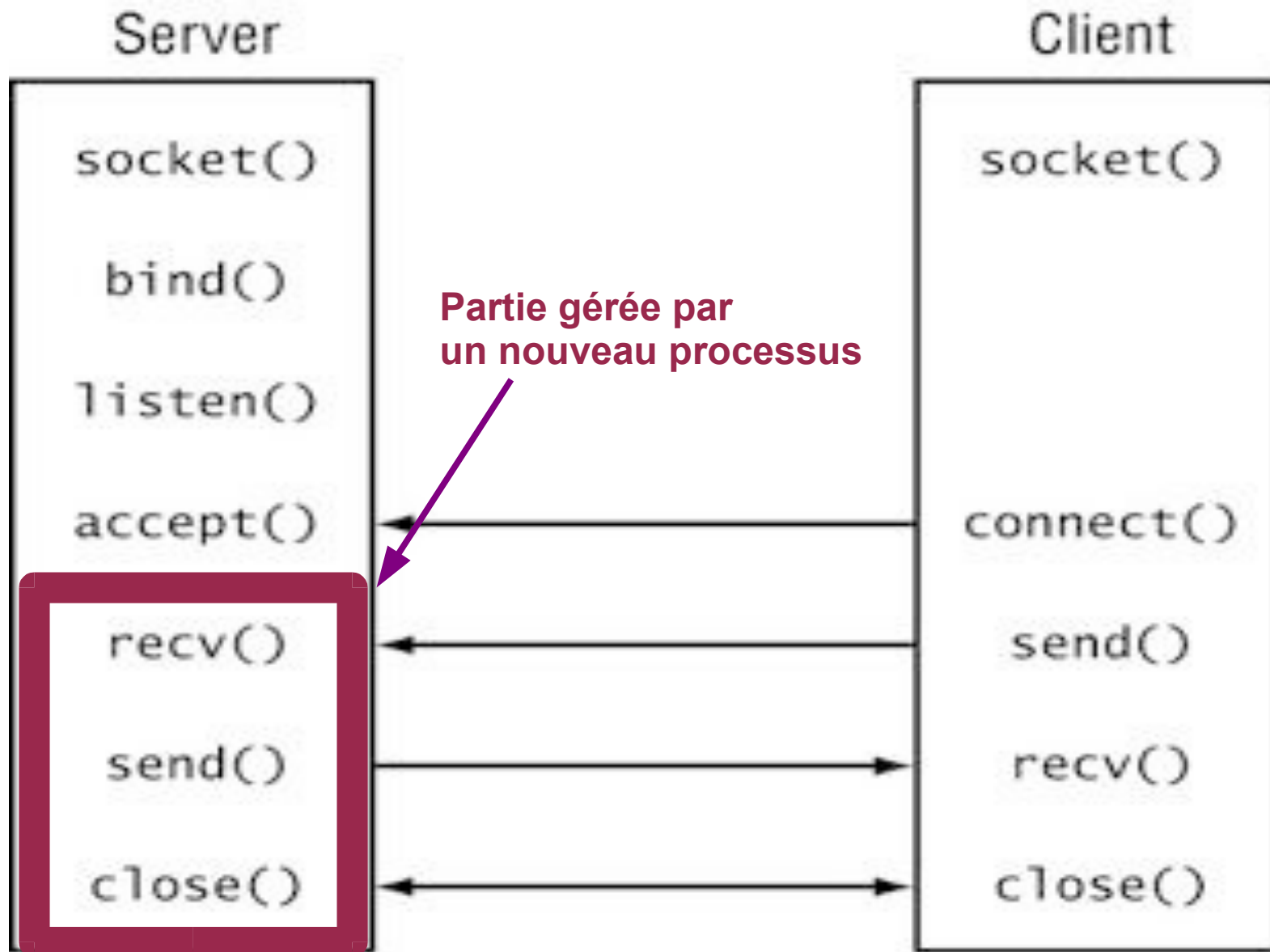
# Exemple de serveur en C

```
int main() {
    int sock=socket(PF_INET,SOCK_STREAM,0);
    struct sockaddr_in address_sock;
    address_sock.sin_family=AF_INET;
    address_sock.sin_port=htons(4242);
    address_sock.sin_addr.s_addr=htonl(INADDR_ANY);
    int r=bind(sock,(struct sockaddr *)&address_sock,sizeof(struct sockaddr_in));
    if(r==0){
        r=listen(sock,0);
        while(1){
            struct sockaddr_in caller;
            socklen_t size=sizeof(caller);
            int sock2=accept(sock,(struct sockaddr *)&caller,&size);
            if(sock2>=0){
                char *mess="Yeah!\n";
                write(sock2,mess,strlen(mess)*sizeof(char));
                char buff[100];
                int recu=read(sock2,buff,99*sizeof(char));
                buff[recu]='\0';
                printf("Message recu : %s\n",buff);
                close(sock2);
            }
        }
    }
    return 0;
}
```

# Problème du serveur séquentiel

- Le serveur précédent ne peut pas accepter plusieurs connexions simultanées
- Si il est en train de communiquer avec un client, un autre client qui fait **connect** sera mise en attente
- **Comment éviter cela :**
  - Faire un serveur concurrent
  - À chaque **accept**, créer un nouveau processus ou thread qui gère la communication
  - Il faut alors faire attention
    - aux données partagées entre les différents threads
    - à comment on gère l'ouverture et la fermeture de socket

# Schéma Client-Serveur en C



# À propos des processus

- Un processus est un **programme** (de nature statique) en cours d'**exécution** (de nature dynamique)
  - son exécution nécessite un environnement
    - espace d'adressage
    - objets entrées/sorties (par exemple sortie standard et entrée standard)
- Plusieurs processus peuvent s'exécuter sur une même machine de façon quasi-simultanée
  - Si le système d'exploitation est **à temps partagé** ou **multi-tâche**
  - Ce même sur une machine mono-processeur
  - Le système d'exploitation est chargé d'allouer les ressources
    - mémoire, temps processeur, entrées/sorties
  - *On a l'illusion du parallélisme*

# Création d'un nouveau processus

- En C, on peut créer un nouveau processus grâce à la fonction fork :
  - **pid\_t fork(void);**
- Cette fonction :
  - Crée un processus fils
  - Renvoie un identifiant de processus PID
  - On peut tester si on est le processus père ou le processus fils grâce au PID renvoyé
    - fork renvoie 0 pour le fils
    - fork renvoie l'identifiant du fils au père
- Autres fonctions utiles :
  - **pid\_t getpid(void);** pour avoir l'id du processus courant
  - **pid\_t getppid(void);** pour avoir l'id du processus père

# Création d'un nouveau processus (2)

- En pratique :
  - Tout se passe dans le même code
  - On teste le retour de fork pour savoir si on est le père ou le fils
    - `if(fork()==0) {...}`
  - Cela permet de distinguer ensuite les exécutions du processus fils et du père
  - Points sur lesquels il faut être attentif
    - Les variables ne sont pas partagées
    - À la création d'un processus fils, l'espace d'adressage du père est copiée

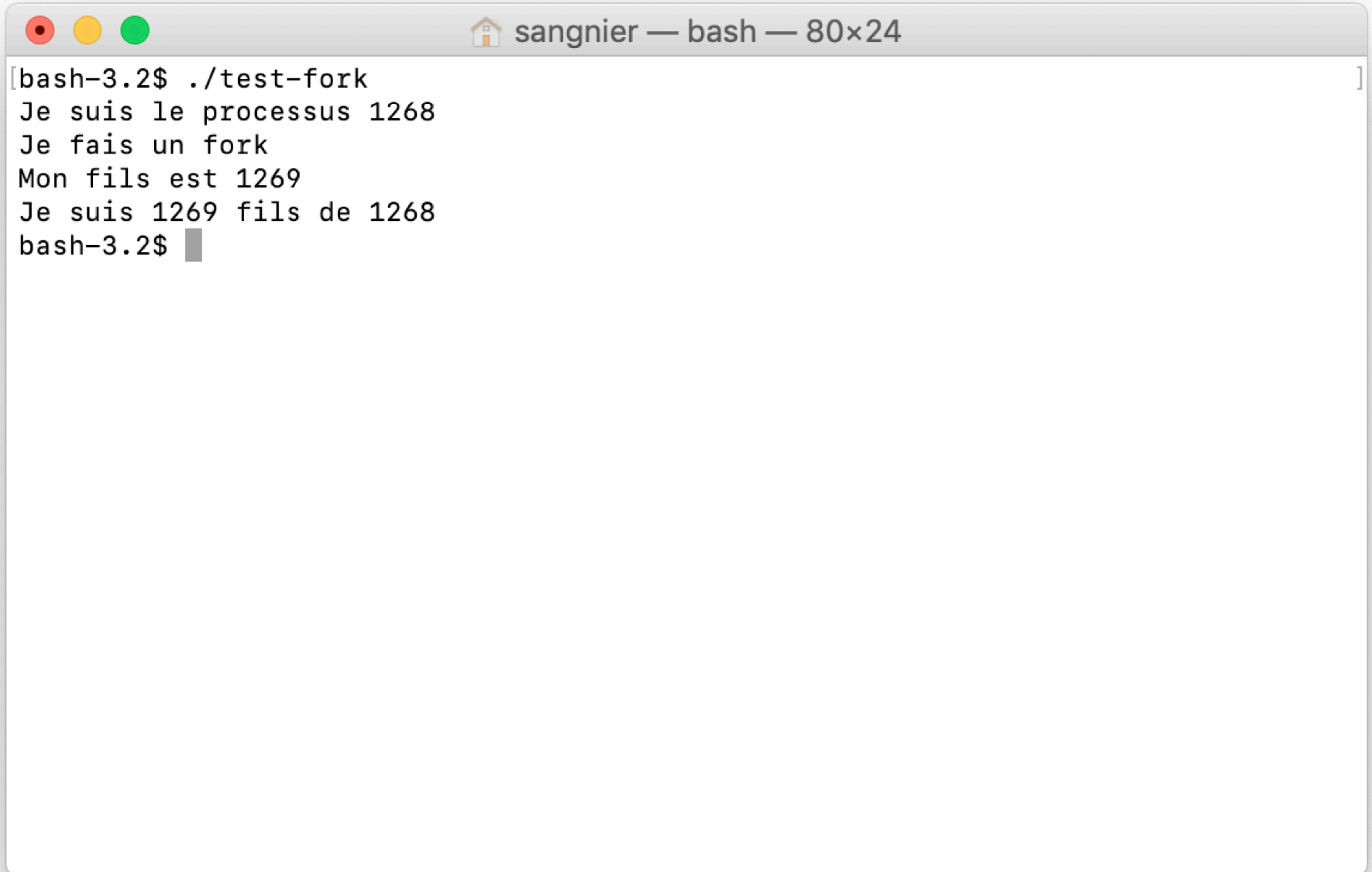
# Exemple

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

int main() {
    int id=getpid();
    printf("Je suis le processus %d\n",id);
    printf("Je fais un fork\n");
    int idfork=fork();
    if(idfork==0){
        int idfils=getpid();
        int idpere=getppid();
        printf("Je suis %d fils de %d\n",idfils,idpere);
    } else {
        printf("Mon fils est %d\n",idfork);
        int etat;
        waitpid(idfork, &etat, 0);
    }
    return 0;
}
```



# Résultat



```
[bash-3.2$ ./test-fork  
Je suis le processus 1268  
Je fais un fork  
Mon fils est 1269  
Je suis 1269 fils de 1268  
bash-3.2$ █
```

# Serveur concurrent avec fork

- Au niveau du serveur, après
  - **int sock2=accept(sock,(struct sockaddr \*)&caller,&size);**
- Le serveur créera un processus fils avec **fork**
- Le processus fils devra
  - Fermer son descripteur **sock**
  - Communiquer avec le client via son descripteur **sock2**
  - Fermer **sock2** à la fin de la communication
  - Faire **exit** pour terminer son exécution
- Le processus père devra
  - Fermer son descripteur **sock2**
  - Retourner sur l'**accept** pour attendre une nouvelle connexion
- **Vu que les espaces d'adressage sont copiés, si le père ferme son sock2, il ne ferme pas celui du fils**

# Exemple

```
int main() {
    int sock=socket(PF_INET,SOCK_STREAM,0);
    struct sockaddr_in sockaddress;
    sockaddress.sin_family=AF_INET;
    sockaddress.sin_port=htons(4244);
    sockaddress.sin_addr.s_addr=htonl(INADDR_ANY);
    int r=bind(sock,(struct sockaddr *)&sockaddress,sizeof(struct sockaddr_in));
    if(r==0){
        r=listen(sock,0);
        if(r==0){
            struct sockaddr_in caller;
            socklen_t si=sizeof(caller);
            while(1){
                int sock2=accept(sock,(struct sockaddr *)&caller,&si);
                if(sock2>=0){
                    int idfork=fork();
                    if(idfork==0){
                        close(sock);
                        /*Code de communication*/
                        close(sock2);
                        exit(0);
                    }
                    else{
                        close(sock2);
                    }
                }
            }
        }
    }
    return 0;
}
```

# Problème avec le fork

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

int main() {
    int pid = fork();
    int x=0;
    if ( pid == 0 ) {
        x=25;
        printf("Valeur de x pour le fils:%d\n",x);
        printf("fin du processus fils\n");
    }
    else {
        while(x!=25) {
            sleep(2);
            printf("Valeur de x pour le pere %d\n",x);
        }
        printf("fin du processus pere\n");
    }
    return 0;
}
```

# Résultat

```
sangnier — bash — 80x24
[bash-3.2$ ./test-fork2
Valeur de x pour le fils:25
fin du processus fils
Valeur de x pour le pere 0
Valeur de x pour le pere 0
Valeur de x pour le pere 0
Valeur de x pour le pere 0
Valeur de x pour le pere 0
^C
bash-3.2$ █
```

- La variable x du père n'est pas la même que la variable x du fils
- Quand le fils met son x à 25 le père ne le voit pas
- Le père reste donc bloquer dans sa boucle et la valeur de sa variable x reste inchangée

# Partage de variables



- On va avoir recours à des processus légers (**threads**)

# Threads vs Processus

- **Un thread** est un fil d'exécution dans un programme, qui est lui même exécuté par un processus
- Un processus peut avoir plusieurs threads
  - Il est alors **multi-threadé**
  - Au minimum il y a un thread
- Chaque fil d'exécution est distinct des autres et a pour attributs
  - Un point courant d'exécution (**pointeur d'instruction**) ou **PC (Program Counter)**
  - Une pile d'exécution (**stack**)
- Les threads sont plus difficiles à programmer en C MAIS :
  - Implémentation plus efficace
  - Partage des données plus faciles

# L'API POSIX

- En C, il existe une librairie classiquement utilisée pour la création et la manipulation de thread. La librairie **POSIX**
- Pour l'utiliser, il faut faire inclure le bon fichier :
  - **#include <pthread.h>**
- Mais il faut aussi préciser à la compilation que l'on souhaite utiliser cette librairie avec l'option `-pthread`
- Par exemple pour compiler un fichier `test.c` on pourra faire :
  - **gcc -pthread -Wall -o test test.c**



# Création de thread

- Pour créer un thread, on utilise la fonction de création de thread :
  - **int pthread\_create(  
pthread\_t \*thread, //On stockera les données du thread créé  
const pthread\_attr\_t \*attr, // Attributs du thread  
void \*(\*start\_routine) (void \*), // Fonctions à exécuter  
void \*arg); // Arguments de la fonction à exécuter**
- En pratique pour les attributs, on mettra **NULL** pour avoir les attributs par défauts
- Ces attributs permettent par exemple de donner une politique pour l'ordonnancer
- Le code qu'exécutera est donné par la fonction **start\_routine**
- Dès que la fonction **pthread\_create** a fini, le thread est créé et s'exécute

# Création de thread en pratique

- On crée une fonction qui contient le code à exécuté par chacun des threads, de signature
  - **void \*function(void \*)**
- Si on veut donner des arguments à cette fonction au moment de l'appel de thread
  - on les passe comme dernière argument de la fonction **pthread\_create**
- Il faut parfois attendre la fin d'exécution des thread créé :
  - si le programme principale termine avant, les threads sont détruits
  - on peut utiliser pour cela la fonction
    - **int pthread\_join(pthread\_t thread, void \*\*value\_ptr);**
  - le deuxième argument est celui créé par pthread\_create
  - value\_ptr est la valeur renvoyé par le thread en appelant
    - **void pthread\_exit(void \*value\_ptr);**
  - Attention : un appel à exit fait terminer le processus !!!!

# Exemple

- On commence par donner le code que va exécuter

```
void *affiche(void * arg) {  
    char *s=(char *) arg ;  
    printf("Mon message est :%s",s) ;  
    return NULL ;  
}
```

- Ensuite pour créer un thread qui exécute cette fonction, on procède ainsi

```
pthread_t th1;  
char *s1="Je suis thread 1\n";  
int r1=pthread_create(&th1,NULL,affiche,s1);
```

- Finalement on peut faire un join pour attendre la fin du processus

```
pthread_join(th1, NULL);
```

# Exemple

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

void *affiche(void *ptr);

int main(){
    pthread_t th1, th2;
    char *s1="Je suis thread 1\n";
    char *s2="Je suis thread 2\n";
    pthread_create(&th1,NULL,affiche,s1);
    pthread_create(&th2,NULL,affiche,s2);

    pthread_join(th1,NULL);
    pthread_join(th2,NULL);
    return 0;
}

void *affiche(void *ptr){
    char *s=(char *)ptr;
    printf("Mon message est : %s",s);
    return NULL;
}
```

# Exemple avec retour de fonction

```
int main(){
    pthread_t th1, th2;
    char *s1="Je suis thread 1\n";
    char *s2="Je suis thread 2\n";
    pthread_create(&th1,NULL,affiche,s1);
    pthread_create(&th2,NULL,affiche,s2);

    char *r1;
    char *r2;
    pthread_join(th1,(void **)&r1);
    pthread_join(th2,(void **)&r2);
    printf("%s",r1);
    printf("%s",r2);
    return 0;
}

void *affiche(void *ptr){
    char *s=(char *)ptr;
    printf("Mon message est : %s",s);
    char *mess=(char *)malloc(100*sizeof(char));
    strcat(mess,s);
    strcat(mess," Fini\n");
    return mess;
}
```

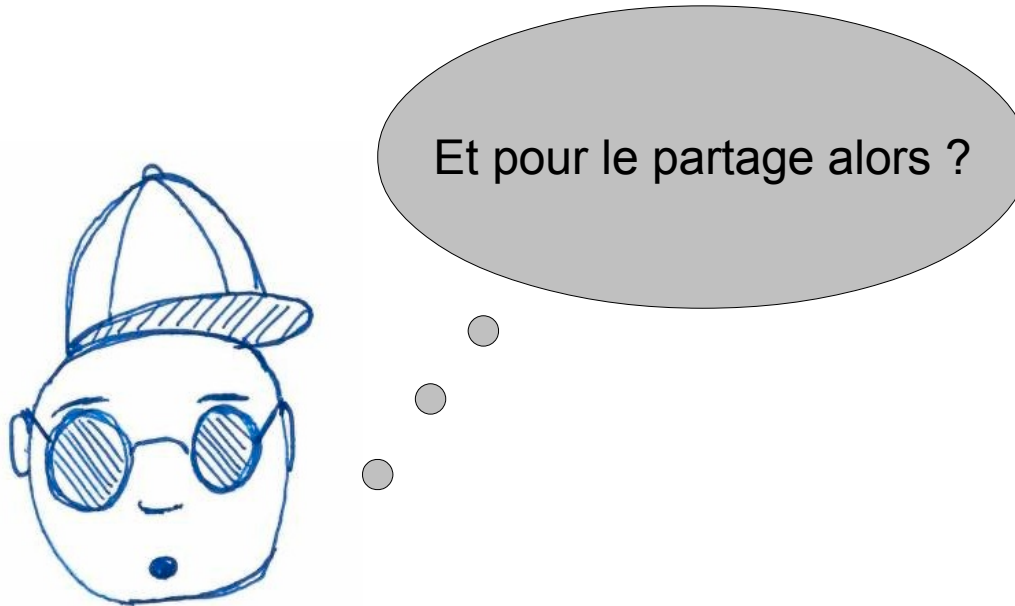
# Exemple avec pthread\_exit

```
int main() {
    pthread_t th1, th2;
    char *s1="Je suis thread 1\n";
    char *s2="Je suis thread 2\n";
    pthread_create(&th1,NULL,affiche,s1);
    pthread_create(&th2,NULL,affiche,s2);

    char *r1;
    char *r2;
    pthread_join(th1,(void **)&r1);
    pthread_join(th2,(void **)&r2);
    printf("%s",r1);
    printf("%s",r2);
    return 0;
}

void *affiche(void *ptr) {
    char *s=(char *)ptr;
    printf("Mon message est : %s",s);
    char *mess=(char *)malloc(100*sizeof(char));
    strcat(mess,s);
    strcat(mess," Fini\n");
    pthread_exit(mess);
}
```

# Partage de variables



- Soit on met ces variables en variables globales
- Soit on passe leur adresse en argument avec **pthread\_create**

# Exemple

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

void *inc(void *ptr);

int a=0;

int main() {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, inc, NULL);
    pthread_create(&th2, NULL, inc, NULL);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("a vaut %d\n", a);
    return 0;
}

void *inc(void *ptr) {
    a=a+1;
    return NULL;
}
```



# Retour sur notre serveur concurrent

- Pour programmer le serveur concurrent avec des threads, on va :
  - Créer un thread après chaque **accept**
  - La fonction qui sera exécuté par chaque thread sera en charge de la communication
  - Il faudra passer en argument de cette fonction, le descripteur de socket
    - En faisant attention à comment on le passe
  - Le thread devra uniquement fermer sa socket de communication lorsque sa conversation avec le client prendra fin
  - Le programme principal qui est censé ne jamais terminé (boucle **while(1){...}**) n'aura pas besoin de faire des **pthread\_join** pour attendre la fin d'exécution des thread communiquant avec les clients.

# Fonction de communication

```
void *comm(void *arg) {
    int sockcomm=*((int *)arg);
    char *mess="Hi\n";
    send(sockcomm,mess,strlen(mess)*sizeof(char),0);
    char buff[100];
    int recu=recv(sockcomm,buff,99*sizeof(char),0);
    buff[recu]='\0';
    printf("Message recu : %s\n",buff);
    close(sockcomm);
    return NULL;
}
```

# Exemple serveur

```
int main() {
    int sock=socket(PF_INET,SOCK_STREAM,0);
    struct sockaddr_in sockaddress;
    sockaddress.sin_family=AF_INET;
    sockaddress.sin_port=htons(4244);
    sockaddress.sin_addr.s_addr=htonl(INADDR_ANY);
    int r=bind(sock, (struct sockaddr *)&sockaddress, sizeof(struct
                                                                    sockaddr_in));

    if(r==0){
        r=listen(sock,0);
        if(r==0){
            struct sockaddr_in caller;
            socklen_t si=sizeof(caller);
            while(1){
                int sock2=accept(sock, (struct sockaddr *)&caller, &si);
                if(sock2>=0){
                    pthread_t th;
                    pthread_create(&th, NULL, comm, &sock2);
                }
            }
        }
    }
    return 0;
}
```

# Mauvais exemple serveur

```
int main() {
    int sock=socket(PF_INET,SOCK_STREAM,0);
    struct sockaddr_in sockaddress;
    sockaddress.sin_family=AF_INET;
    sockaddress.sin_port=htons(4244);
    sockaddress.sin_addr.s_addr=htonl(INADDR_ANY);
    int r=bind(sock, (struct sockaddr *)&sockaddress, sizeof(struct
                                                                    sockaddr_in));

    if(r==0) {
        r=listen(sock,0);
        if(r==0) {
            struct sockaddr_in caller;
            socklen_t si=sizeof(caller);
            int sock2 ; ← NON !!!!
            while(1) {
                sock2=accept(sock, (struct sockaddr *)&caller, &si);
                if(sock2>=0) {
                    pthread_t th;
                    pthread_create(&th, NULL, comm, &sock2);
                }
            }
        }
    }
    return 0;
}
```

# Explication

- Dans l'exemple précédent :
  - la variable sock2 sera partagée entre tous les threads de communication
- Si un nouveau client arrive et que le accept est exécuté avant que le thread précédent ait pu stocké la valeur de sock2 concernant sa communication, il y aura des problèmes
- Pour cela il suffit que le client soit lent avant de stocker la valeur de sock2 pour sa communication
- En fait le problème arrive aussi dans l'exemple précédent
- **Bonne pratique de programmation : allouez dynamiquement la mémoire pour stocker les descripteurs**
- Et bien entendu il faut faire les free correspondant.

# Exemple serveur correct

```
int main() {
    int sock=socket(PF_INET,SOCK_STREAM,0);
    struct sockaddr_in sockaddress;
    sockaddress.sin_family=AF_INET;
    sockaddress.sin_port=htons(4244);
    sockaddress.sin_addr.s_addr=htonl(INADDR_ANY);
    int r=bind(sock, (struct sockaddr *)&sockaddress, sizeof(struct
                                                                    sockaddr_in));

    if(r==0) {
        r=listen(sock,0);
        if(r==0) {
            struct sockaddr_in caller;
            socklen_t size=sizeof(caller);
            while(1){
                int *sock2=(int *)malloc(sizeof(int));
                *sock2=accept(sock, (struct sockaddr *)&caller, &si);
                if(sock2>=0) {
                    pthread_t th;
                    pthread_create(&th, NULL, comm, sock2);
                }
            }
        }
    }
    return 0;
}
```

# Attention au partage des données

- Les threads s'exécutent en parallèle
- Vous ne pouvez faire aucune hypothèse sur l'ordonnancement
- Si on n'est pas attentif sur la manipulation des variables partagées, on peut observer des comportements étranges
- Par exemple :
  - Un thread teste si une variable entière est positive avant de la décrémenter
  - Il est interrompu entre le test et la décrémentation
  - Un autre thread met la variable partagée à 0
  - Le premier thread met la variable à -1 ... ce qu'on ne voulait pas

# Exemple de partage de données

```
void *inc(void *ptr);
void *dec(void *ptr);

int a=0;

int main(){
    pthread_t th1, th2, th3;
    pthread_create(&th1, NULL, inc, NULL);
    pthread_create(&th2, NULL, dec, NULL);
    pthread_create(&th3, NULL, dec, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    printf("a vaut %d\n", a);
    return 0;
}

void *inc(void *ptr){
    a=a+1;
    return NULL;
}

void *dec(void *ptr){
    if(a>0){
        sleep(1);
        a=a-1;
    }
    return NULL;
}
```

**Ce code peut  
potentiellement  
afficher -1**



# Comment protéger les données

- L'accès aux données partagées doit donc être protégé
- En Java, on utilise le mot clef **synchronized**
- En C, on pourra utiliser des **verrous (mutex)**
- Le principe est suivant :
  - Un thread qui veut accéder à une donnée partagée demande le verrou
  - Si le verrou est libre, il continue son exécution
  - Si le verrou est pris, il bloque jusqu'à ce que le verrou soit libéré
  - Quand il a fini d'accéder aux variables partagées, il libère le verrou
- Ces verrous sont partagés entre les threads
- **Bonne pratique** : c'est un thread qui a pris le verrou qui doit le libérer

# Les verrous en C

- La librairie POSIX a des verrous de type :
  - **pthread\_mutex\_t verrou ;**
- La première chose à faire est initialisée le verrou
- Le plus simple consiste à faire :
  - **verrou=PTHREAD\_MUTEX\_INITIALIZER ;**
- Pour prendre le verrou, on utilise la fonction :
  - **int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);**
- Et pour libérer le verrou on utilise :
  - **int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);**
- Ces deux fonctions renvoient 0 si tout se passe bien

# Exemple utilisation de verrou

```
pthread_mutex_t verrou= PTHREAD_MUTEX_INITIALIZER;
int a=0;

int main(){
    pthread_t th1, th2, th3;
    pthread_create(&th1, NULL, inc, NULL);
    pthread_create(&th2, NULL, dec, NULL);
    pthread_create(&th3, NULL, dec, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    printf("a vaut %d\n", a);
    return 0;
}

void *inc(void *ptr){
    pthread_mutex_lock(&verrou);
    a=a+1;
    pthread_mutex_unlock(&verrou);
    return NULL;
}

void *dec(void *ptr){
    pthread_mutex_lock(&verrou);
    if(a>0){
        a=a-1;
    }
    pthread_mutex_unlock(&verrou);
    return NULL;
}
```