M1 Informatique - Systèmes Répartis

Séance 2 - Multiplexage des entrées sorties

Benoît Darties

Université de Montpellier

13 octobre 2025

Problématique

Communication TCP et situations de blocage

Dans une communication TCP :

- Les opérations d'attente de connexion (`accept()`) sont bloquantes
- Les opération de réception de message (`recv()`) sont également bloquantes
- S'il y a plusieurs systèmes connectés - chacun par une socket - on ne sait pas lequel va nous envoyer un message en premier

Comment pouvoir travailler avec plusieurs sockets simultanément, en évitant les situations de blocage lorsqu'on ne connait pas l'ordre d'arrivée des messages ?

Comment éviter des situations de blocage

Stratégies possibles

- **Connaitre à l'avance l'ordre d'arrivée des messages** - ordonnancement - et organiser les communications en ce sens : non applicable pour des systèmes entièrement répartis
- **Modifier le comportement bloquant d'opérations d'entrée/sortie** via les options de `send/sendto/recv/recvfrom` ... : impose souvent de réaliser des boucles en attente active, lourd, couteux en temps, inadapté.
- **Utiliser des mécanismes avancés de surveillance des entrées sortie** :
  1. on surveille **simultanément** une liste de descripteurs (sockets)
  2. dès qu'un descripteur est dans un état exploitable (ie : arrivée d'un message, demande de nouvelle connexion) on le sélectionne et on effectue les traitements adéquats dessus,
  3. puis on retourne surveiller les les autres descripteurs (boucle de fonctionnement)

  C'est ce qu'on appelle du **multiplexage d'entrées/ sorties**

Plan

1. Présentation et fonctionnement
2. Exemple : code sans multiplexage vs code avec multuplexage
3. Configuration avancées

Définitions

Multiplexage des entrées/sorties

Un moyen de scruter plusieurs descripteurs de fichier ouverts, en attendant qu'un événement se produise sur au moins l'un de ces descripteurs.

Événement

- Un changement d'état à « **prêt** » pour effectuer une ou des opérations d'entrée/sortie.
- Lorsqu'un descripteur est à l'état **prêt** pour une opération, cette dernière peut être effectuée sans blocage.

Éléments scrutables

Tout élément d'un système, manipulable via un descripteur de fichier : **socket, tube, entrée standard, sortie standard, fichier**, etc.

Objectif de ce cours

Développer un serveur capable de gérer plusieurs clients simultanément sans devoir créer plusieurs processus. **En particulier**

- Utilisation du multiplexage des I/O
- Scruter les arrivées de messages sur des sockets créées par un serveur.

Appels systèmes et types vu dans ce cours

- type `fd_set` : type de données
- macros `FD_CLR()`, `FD_COPY()`, `FD_ISSET()`, `FD_SET()`, `FD_ZERO()`
- appel système `select()` : multiplexage des I/O

un problème d'anglais

en anglais

- un ensemble (d'éléments) se traduit par le mot : **set**
- l'action de fixer / définir une valeur à une variable se traduit par le mot : **set**

- Importance de comprendre le contexte
- Utilisation de majuscules / minuscules.

Mise en place du multiplexage

Déroulement global

1. On définit l'ensemble des sockets à scruter (écouter) en précisant :
   - quelles sockets sont à scruter en entrée (prêt en lecture suite à l'arrivée d'un message)
   - quelles sockets sont à scruter en écriture (prêt en écriture – quand ?)
2. On scrute les événements définis en une seule opération : opération bloquante
3. Si déblocage, on vérifie pour chaque socket si un événement s'est produit.
4. Si oui :
   - on sélectionne un descripteur pour le traiter.
   - on effectue l'opération associée (exemple : `recv` pour lire un message reçu).
   - on réinitialise le descripteur ; on le remet dans la liste à structer

Le type `fd_set`

Définition

- `fd_set` représente un **ensemble de descripteurs de fichiers** à surveiller.
- Il s'agit d'une **structure de bits** (bitfield) où chaque bit correspond à un descripteur
- Si le bit est à 1, le descripteur `fd` doit être pris en compte lors de la scrutation.

Représentation schématique

Indice (fd)	0	1	2	3	4	5	6	7	8	9	10	11
Valeur (bit)	0	0	0	0	0	1	0	0	1	0	1	0

Dans cet exemple, les descripteurs 5, 8 et 10 sont marqués comme actifs (valeur à 1).

Création et taille d'un ensemble `fd_set`

Taille maximale d'un `fd_set`

- Le nombre maximal de descripteurs pouvant être surveillés est défini par la constante `FD_SETSIZE` dont la valeur par défaut est 1024
- Cela signifie qu'un `fd_set` peut contenir jusqu'à **1024 descripteurs** différents (numérotés de 0 à 1023).
- Si un descripteur supérieur à `FD_SETSIZE - 1` est utilisé, le comportement est indéfini.
- La taille est fixée à la compilation.

Création d'un ensemble de descripteurs

```
1 #include <sys/select.h>
2 ...
3 fd_set ensemble;          // Déclare un ensemble de descripteurs
```

Opérations sur un ensemble `fd_set`

Une fois l'ensemble créé, les macro permettent de gérer facilement l'ajout / suppression de descripteurs à surveiller, ainsi que l'interrogation et la remise à zéro de l'ensemble fd_set

Macro disponibles

- `void FD_ZERO(fd_set *set);` Initialise à faux les éléments de l'ensemble `set`.
- `void FD_SET(int desc, fd_set *set);` Ajoute le descripteur `desc` à la liste des descripteurs de `*set` à scruter, i.e. positionne l'élément à l'indice `desc` à vrai.
- `select()` : se met en attente jusqu'à ce qu'un événement se produise sur au moins un descripteur scruté. Dans ce cas, elle **modifie** les ensembles passés en paramètres pour ne garder que les descripteurs passés à l'état "prêt" et retirer les autres.
- `int FD_ISSET(int desc, fd_set *set);` Teste si le descripteur `desc` est dans la liste des descripteurs de `*set`, et si l'indice `desc` est positionné à vrai.
- `void FD_CLR(int desc, fd_set *set);` Supprime le descripteur `desc` de la liste des descripteurs de `*set` à scruter, i.e. positionne l'élément à l'indice `desc` à faux.

La macro FD_ZERO

Prototype

```
1 void FD_ZERO(fd_set *set);
```

Rôle

- Initialise à zéro tous les bits de l'ensemble `fd_set` pointé par `set`.
- Autrement dit, `FD_ZERO()` vide la liste des descripteurs à scruter.
- Cette macro est toujours utilisée avant d'ajouter des descripteurs avec `FD_SET()`.
- Elle garantit que l'ensemble ne contient aucun descripteur résiduel d'un usage précédent.

Retour

- Aucune valeur de retour (fonction de type `void`).
- Le contenu mémoire de la structure est simplement remis à zéro.

Code : création et initialisation

```c
#include <sys/select.h>
...
    fd_set ensemble;      // Déclare un ensemble de descripteurs
    FD_ZERO(&ensemble);   // Initialise tous les bits à 0
```

Résultat : représentation mémoire simplifiée

Indice (fd)	0	1	2	3	4	5	6	7	8	9	10	11
Valeur (bit)	0	0	0	0	0	0	0	0	0	0	0	0

Tous les bits sont à 0 : aucun descripteur n'est encore surveillé.

La macro FD_SET

Prototype

```
1 void FD_SET(int fd, fd_set *set);
```

Rôle

- Ajoute le descripteur de fichier `fd` dans l'ensemble `fd_set` pointé par `set`.
- Marque ce descripteur comme devant être scruté par l'appel `select()`.
- Utilisé pour indiquer au noyau qu'on souhaite surveiller une socket ou un fichier particulier pour un type d'événement (lecture, écriture, exception).

Retour

- Aucune valeur de retour (fonction de type `void`).
- En cas de descripteur invalide (`fd` négatif ou supérieur à `FD_SETSIZE`), le comportement est indéfini — il faut donc vérifier les bornes avant l'appel.

Code : création, initialisation et ajout de descripteurs

```
1    fd_set ensemble;           // Déclare un ensemble de descripteurs
2    FD_ZERO(&ensemble);        // Initialise tous les bits à 0
3
4    int sock1 = 5, sock2 = 8; // fictif. à remplacer par descripteur socket
5    FD_SET(sock1, &ensemble); // Ajoute sock1 à l'ensemble
6    FD_SET(sock2, &ensemble); // Ajoute sock2 à l'ensemble
```

Résultat : représentation mémoire simplifiée

Indice (fd)	0	1	2	3	4	5	6	7	8	9	10	11
Valeur (bit)	0	0	0	0	0	1	0	0	1	0	0	0

Les descripteurs 5 et 8 ont été ajoutés à l'ensemble : seuls ces bits sont à 1, ils seront surveillés par `select()`.

La fonction `select()` — Prototype et arguments

Prototype

```
int select(
    int nfds,
    fd_set *readfds,  fd_set *writefds,  fd_set *exceptfds,
    struct timeval *timeout
);
```

Description des arguments

- `nfds` : valeur du plus grand descripteur à surveiller + 1.
- `readfds` : ensemble descripteurs à surveiller pour la lecture.
- `writefds` : ensemble descripteurs à surveiller pour l'écriture.
- `exceptfds` : ensemble descripteurs pour la détection d'exceptions.
- `timeout` : durée maximale d'attente, ou `NULL` pour un blocage illimité.

La fonction `select()` — Fonctionnement

Principe

- `select()` met le processus en attente jusqu'à ce qu'un ou plusieurs descripteurs deviennent prêts.
- Les ensembles (`readfds`, `writefds`, `exceptfds`) sont **modifiés** : seuls les descripteurs prêts sont conservés.
- Si `timeout` est `NULL`, l'attente est bloquante, sinon se débloque à expiration du délai.

Valeur de retour

- `>= 0` : nombre de descripteurs devenus prêts.
- `-1` : erreur (consulter `errno`).

Remarque

Après l'appel à `select()`, les ensembles doivent être reconstruits car ils ont été modifiés.

Représentation simplifiée : avant `select()`

fd	0	1	2	3	4	5	6	7	8	9
bit	0	0	0	1	0	1	0	0	1	0

Sur l'ensemble `readfds`, 3 descripteurs sont à surveillés pour lecture

Code simplifié

```
1    // [3, 5, 8] surveillés pour lecture
2    select(9, &readfds, NULL, NULL, NULL);
3    // appel bloquant. se débloque si un événement intervient
```

Représentation simplifiée

Après `select()` (un descripteur est prêt en lecture). Les autres ont été remis à 0.

fd	0	1	2	3	4	5	6	7	8	9
bit	0	0	0	0	0	1	0	0	0	0

La macro FD_ISSET

Prototype

```
1 int FD_ISSET(int fd, fd_set *set);
```

Rôle

- Vérifie si le descripteur de fichier `fd` appartient à l'ensemble `fd_set` pointé par `set`.
- Retourne une valeur non nulle si le descripteur est présent (bit à 1), 0 sinon.
- Après un appel à `select()`, cette macro permet d'identifier les descripteurs devenus prêts pour une opération (lecture, écriture, exception).

Retour

- `> 0` : le descripteur `fd` est actif dans l'ensemble.
- `0` : le descripteur `fd` n'est pas présent ou n'est pas prêt.
- Si argument invalide (`fd` négatif ou supérieur à `FD_SETSIZE`) : comportement indéfini.

Code : test de plusieurs descripteurs prêts (portion de code)

```c
fd_set readfds;
FD_ZERO(&readfds); // mise à 0 avant de scruter
FD_SET(3, &readfds); // descripteur 3 fictif (ex: sockets)
FD_SET(5, &readfds); // descripteur 5 fictif (ex: sockets)
int maxfd = 9;

select(maxfd + 1, &readfds, NULL, NULL, NULL);   // Attente  en lecture

// Boucle sur tous les descripteurs possibles
for (int fd = 0; fd <= maxfd; fd++) {
    if (FD_ISSET(fd, &readfds)) {
        printf("Des données sont disponibles sur fd=%d\n", fd);
        recv(fd, buffer, sizeof(buffer), 0);
    }
}
```

La macro FD_CLR

Prototype

```
1 void FD_CLR(int fd, fd_set *set);
```

Rôle

- Supprime le descripteur de fichier `fd` de l'ensemble `fd_set` pointé par `set`.
- Met le bit correspondant à `fd` à 0, indiquant que ce descripteur ne doit plus être scruté.
- Généralement utilisée pour retirer une socket fermée ou inactive d'un ensemble déjà géré.

Retour

- Aucune valeur de retour (fonction de type `void`).
- Si le descripteur est invalide (`fd` $<$ 0 ou $>$ à `FD_SETSIZE`) : comportement indéfini.

Plan

1 Présentation et fonctionnement

2 Exemple : code sans multiplexage vs code avec multuplexage

3 Configuration avancées

On reprend tout : exemple d'un serveur

Un serveur qui reçoit, en boucle, des requêtes de clients et retourne des réponses.

Serveur d'origine – itératif, sans multiplexage (extrait)

```
1  // ... socket(...), bind(...), listen(...)
2  while (1) {
3    sockTravailClient = accept(sockPrincipale, NULL, NULL);
4    char buffer[1000] // supposée fournie
5
6    int nbLus = recv(sockTravailClient, &buffer, 1000, 0);
7    send(sockTravailClient, &buffer, nbLus, 0);
8
9    close(sockTravailClient);
10   // ou alors : fork() /thread() pour maintenir la connexion
11 }
12 close(sockPrincipale); // non atteint ici
```

Serveur d'origine – itératif, avec multiplexage 1/2 (extrait)

```
// ... socket(...), bind(...), listen(...)
fd_set readfds; // set courant modifié par select
fd_set readfds_initial; // set initial, pour restaurer apres modification du select

FD_ZERO(&readfds_initial);
FD_SET(sockPrincipale, &readfds_initial);
readfds = readfds_initial;  // mise en place des descripteurs à scruter (copie)
int max = sockPrincipale; // mise a jour avec la plus grosse valeur de descripteur

while (1) { // boucle de fonctionnement
  select(max+1, &readfds, NULL, NULL, NULL); // on bloque jusqu'à un événement
  // on arrive ici : on a donc une socket sur laquelle quelque chose est à lire
  for (int df = 2; df <= max; df++) { // recherche du / des descripteurs pret(s)
    ... (voir page suivante)
    // recherche du descripteur actif, et traitement en conséquence
    // - ajout d'une nouvelle connexion si descripteur actif = sockPrincipale
    // - sinon : lecture sur une socket dédiée client
  }
  readfds = readfds_initial;  // remise en place des descripteurs à scruter
}
```

Serveur d'origine – itératif, avec multiplexage 2/2 (extrait)

```
1  for (int df = 2; df <= max; df++) { // recherche du / des descripteurs pret(s)
2    if (FD_ISSET(df, &readfds))  { // descripteur actif trouvé
3       // si la socket est sockPrincipale : nouvelle connexion
4      if (df == sockPrincipale) {
5        sockTravail = accept(sockPrincipale, NULL, NULL); //  à stocker
6        FD_SET(sockTravail, &readfds_initial); // ajout a la liste à surveiller
7        if (max < sockTravail) max = sockTravail; // mise a jour du nb max
8      }
9      else { // arrivée d'un message d'un client
10       struct requete req; // structure mise en place pour lecture de messages
11       if (recv(df, &req, sizeof(req), 0) <= 0) { // cas d'une fermeture de connexi
12         FD_CLR(df, &readfds_initial);  // suppresion de la requete a ecouter
13         close(df); // fermeture socket
14       }
15       else {
16         struct reponse  rep = taiter(req); // redaction d'une reponse
17         send(df, &rep, sizeof(rep), 0);
18       }
19     }
20   }
21 }
```

Remarques sur l'exemple

- Toutes les sockets ont été prises en compte, y compris la socket d'écoute des demandes de connexion.
- L'exemple est simplifié, il n'est qu'illustratif de l'utilisation des concepts vus.
- Il manque principalement :
  - le traitement des valeurs de retours des fonctions, au cas par cas,
  - la gestion des erreurs,
  - la prise en compte des écritures dans la scrutation (`send`),
  - éventuellement, la mise à jour de la valeur `max` des descripteurs (à quoi cela servirait ?)

Plan

1 Présentation et fonctionnement

2 Exemple : code sans multiplexage vs code avec multuplexage

3 Configuration avancées

Modifier la taille maximale d'un `fd_set`

Principe

- Par défaut, la taille maximale d'un ensemble `fd_set` est fixée à 1024.
- Cette limite est déterminée à la compilation
- Pour la modifier : redéfinir la constante `FD_SETSIZE` avant d'inclure le fichier d'en-tête `<sys/select.h>`.

Exemple de redéfinition

```
1 #define FD_SETSIZE 4096
2 #include <sys/select.h>
```

- Cette directive permet de gérer jusqu'à **4096 descripteurs** simultanément.
- À utiliser avec précaution : tous les programmes et bibliothèques compilés doivent utiliser la même valeur de `FD_SETSIZE` pour rester compatibles.

Scruter des événements durant un laps de temps défini

Délais d'attente

Si le paramètre `timeout` est `NULL`, la fonction attend jusqu'à l'occurrence d'un événement, sinon, un délai doit être défini :

```
1 struct timeval{
2   long tv_sec;   // secondes
3   long tv_usec;  // microsecondes
4 };
```

Valeur de retour

Le nombre de descripteurs scrutés et passés à l'état « prêt », 0 si le temps d'attente s'est écoulé sans occurrence d'un événement, -1 en cas d'erreur.

Pour finir

- Le multiplexage peut se faire en utilisant aussi la fonction : `poll(...)`.
- Penser à utiliser le multiplexage en incluant l'entrée standard dans les descripteurs à scruter, en particulier si votre programme peut réaliser des saisies et des réceptions en parallèle.
- Toujours tester les valeurs de retour et quitter proprement vos programmes (libérations des espaces alloués dynamiquement, fermer les sockets, etc).
- Favoriser le passage de paramètres à vos programmes (ou la saisie) et non le codage en dur des données supposées être modifiables.

L'appel système poll()

Prototype

```c
#include <poll.h>
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

Structures et indicateurs

```c
struct pollfd {
    int   fd;         // descripteur
    short events;     // événements surveillés
    short revents;    // événements survenus (rempli par poll)
};
/* events / revents courants */
POLLIN   /* pret en lecture  */
POLLOUT  /* pret en écriture */
POLLERR  /* erreur           */
POLLHUP  /* hang up/fermeture*/
POLLNVAL /* fd invalide       */
```

L'appel système `poll()`

Rôle

- Attend qu'au moins un descripteur devienne prêt selon `events`.
- Remplit `revents` pour chaque entrée active.
- `timeout` en millisecondes : `-1` bloque, `0` non bloquant.

Retour

- `> 0` : nombre d'entrées avec `revents` non nul.
- `0` : délai expiré sans événement.
- `-1` : erreur (`errno` renseigne la cause).