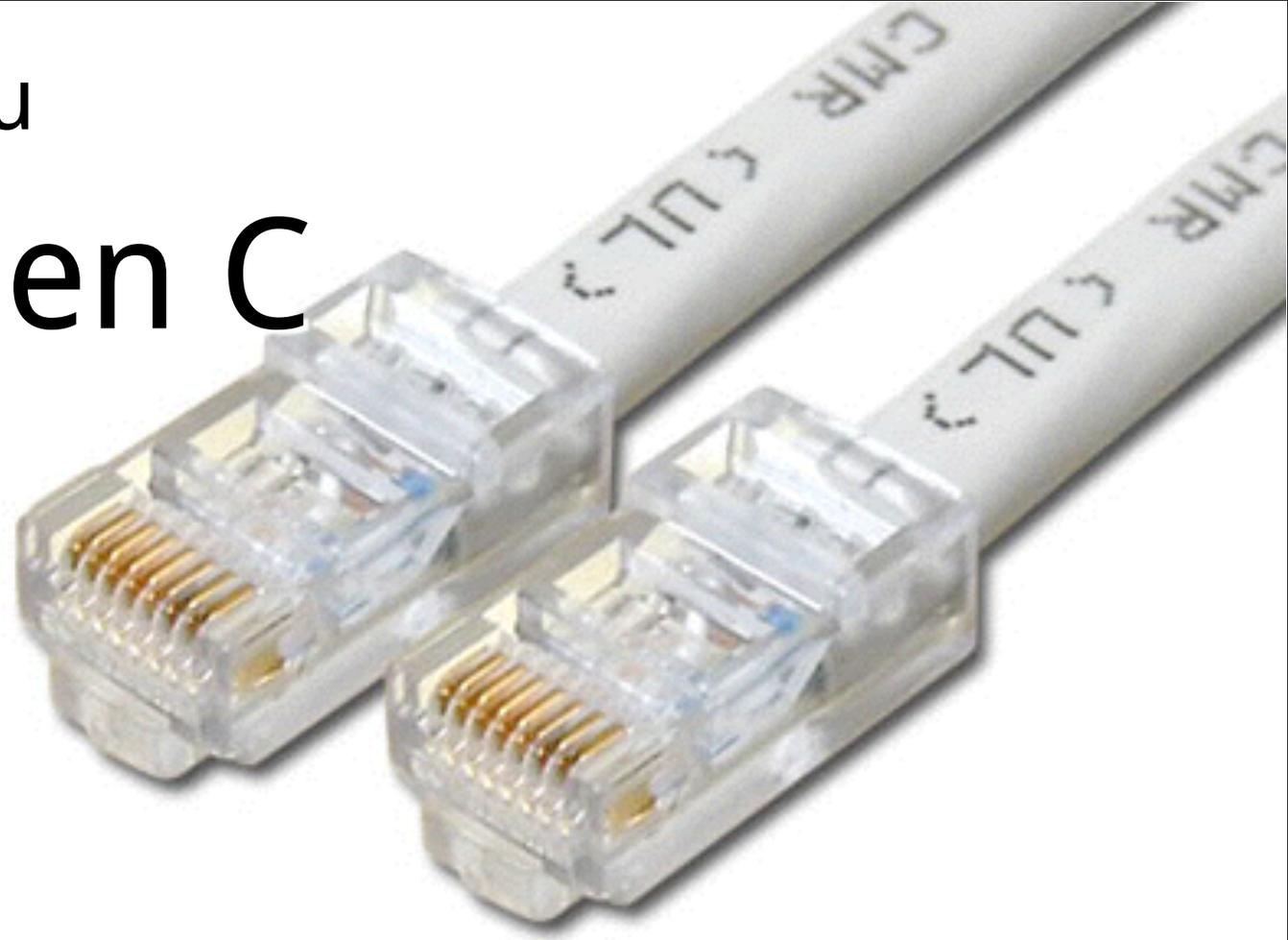
Programmation Réseau

Attente multiple en C

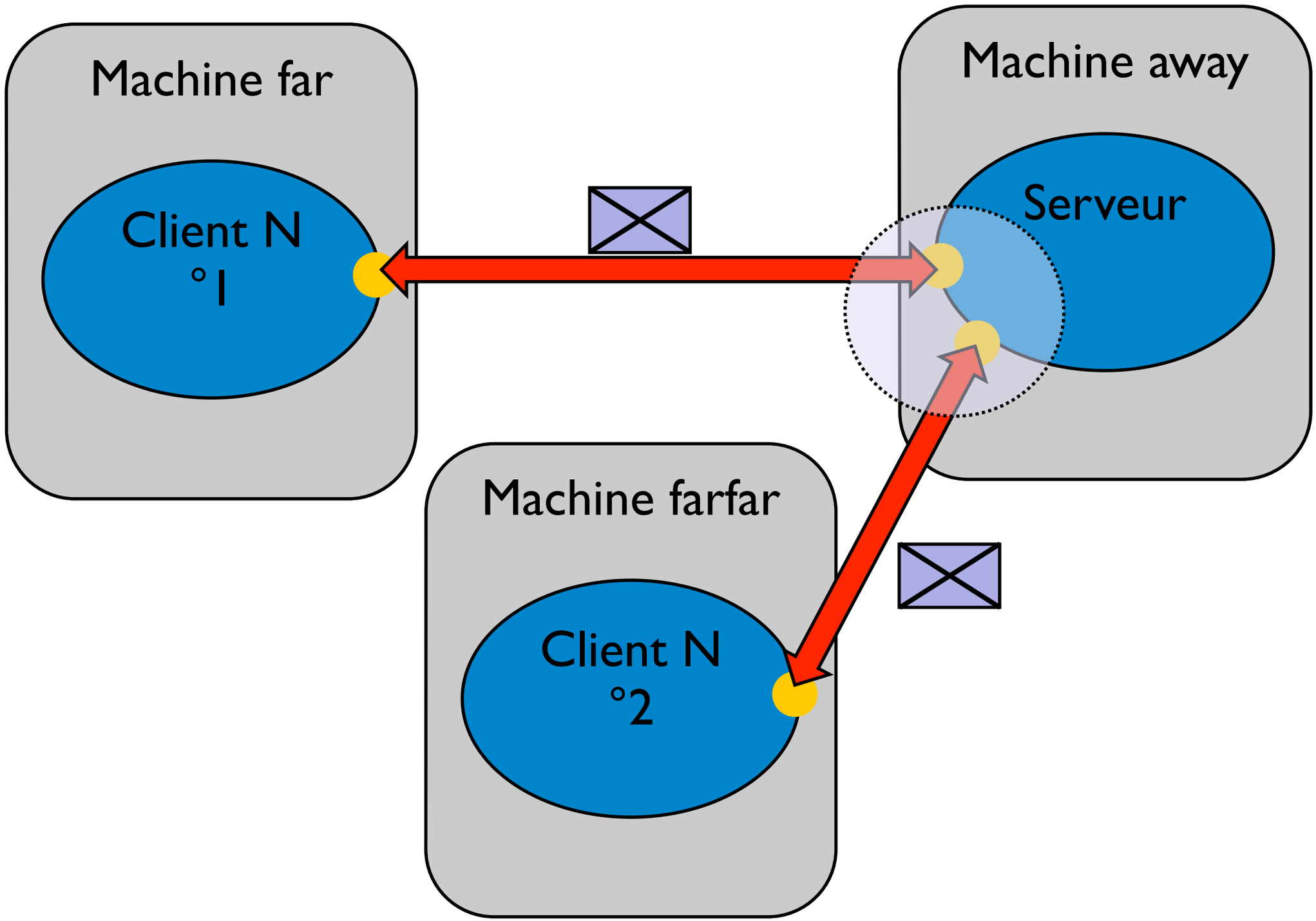


Jean-Baptiste.Yunes@univ-paris-diderot.fr
armand@informatique.univ-paris-diderot.fr

UFR Informatique

2014

Attente multiple



Attente multiple

- La lecture (attente de données) est bloquante:
  - Le programme ressort de l'appel de lecture:
    - Quand il y a des données
    - Ou une erreur
- Comment faire si un serveur veut s'occuper de plusieurs clients simultanément?
  - A tour de rôle?... Oui mais, si un client n'envoie pas de données, le serveur va rester bloqué!

Solutions possibles?

- `fcntl(desc, F_SETFL, O_NONBLOCK)`
  - Les opérations bloquantes rendront `-1` et `errno== EAGAIN`
- `ssize_t read(int desc, void *buf, size_t count);`

```
while(true) {
    if (read(sock1, buf1, lg1) != -1) {
      process(buf1);
    }
    if (read(sock2, buf2, lg2) != -1) {
      process(buf2);
    }
}
```

- Que se passe-t-il si aucun client n’envoie de données?

fcntl

- fcntl(d,cmd,value)
- File descriptor CoNTroL
- Appel système permettant de positionner des attributs associés à un descripteur
- d : descripteur
- cmd à réaliser (set, get, etc.)
- value (essentiellement pour set mais pas que…)

`select`

- Demander au système (Unix ou autre) de réveiller le processus (thread) dès qu'une opération sera possible sur un descripteur parmi N autres.

- Appel système `select`

- fournir la liste des descripteurs de fichiers sur lesquels on attend que quelque chose soit possible.

  - Notion d'ensemble de descripteur « file descriptor set »

    - en fait un simple « champ de bits »...

Select / poll

Conforme POSIX 1003.1

Terminal — less — 80×30

```
SELECT(2)                 BSD System Calls Manual                 SELECT(2)

NAME
     FD_CLR, FD_COPY, FD_ISSET, FD_SET, FD_ZERO, select -- synchronous I/O
     multiplexing

SYNOPSIS
     #include <sys/select.h>

     void
     FD_CLR(fd, fd_set *fdset);

     void
     FD_COPY(fd_set *fdset_orig, fd_set *fdset_copy);

     int
     FD_ISSET(fd, fd_set *fdset);

     void
     FD_SET(fd, fd_set *fdset);

     void
     FD_ZERO(fd_set *fdset);

     int
     select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds,
         fd_set *restrict errorfds, struct timeval *restrict timeout);

:
```

Select / poll

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

- On peut attendre sur tout type de fichiers
  - **sockets**, tubes (nommés ou non), tty, autres…
- Ce n'est donc pas un appel spécifique au réseau, mais en pratique très utilisé dans les applications réseau pour attendre
- des connexions ou des données (`readfds`)
- de la place pour écrire dans un tube, une socket (`writefds`)
- des conditions « exceptionnelles »
  - en pratique : données TCP Out-Of-Band (OOB)

Select / poll

- `nfds :`
  - numéro du plus grand descripteur valide +1
- `readfds :`
  - liste de descripteurs sur lesquels on attend des données en réception (données ou données de liaison — accept)
- `writefds :`
  - liste de descripteurs sur lesquels on attend de pouvoir écrire
- `exceptfds :`
  - liste de descripteurs surveillés pour conditions exceptionnelles
- `utimeout :`
  - temps maximum d'attente (où on reste bloqué dans select – Secondes + µsecondes)
    - NULL => attente bloquante
    - {0,0} => retour immédiat (polling)

`select`

- en sortie:
  - `select` renvoie le nombre de descripteurs sur lesquels des opérations sont possibles
  - les `fd_set` ont été modifiés! (il faudra les réinitialiser pour un prochain appel)
  - les bits à 1 indiquent quels fichiers correspondent aux attentes

Structure générale simplifiée.

```
/* Création et initialisation du fd_set primitif */
/* Expression des souhaits */
fd_set initial_set;
int fd_max=0;
FD_CLR(&initial_set);
FD_SET(d1,&initial_set);
fd_max = max(fd_max,d1);
…
FD_SET(dn,&initial_set);
fd_max = max(fd_max,dn);
…
```

Structure générale simplifiée.

```
for(;;)  {
   fd_set rdfs;
   /* Create « waiting » set */
   FD_COPY(&initial_set,&rdfs);
   /* Wait */
   res = select(fd_max+1,&rdfs,NULL,NULL,NULL);
  /* Find descriptors */
   for (; res >0; ) {
      // Loop on di
      if (FD_ISSET(di, &rdfs)) {
         /* do the job for di: read, accept… */
         res--;
      }
  }
}
```

Select « bonnes pratiques »

- Réinitialiser les `fd_sets` à chaque itération
- Éviter l'abus de délais de garde « timeout »
  - Si utilisé : les réinitialiser à chaque itération
- `nfds` doit être bien calculé (efficacité améliorée)
- Ne mettre dans les `fd_sets` que les descripteurs utiles
- Tester tous les descripteurs des `fd_sets` au retour

Select « bonnes pratiques »

- Attention
  - `read`, `write`, `send`, `recv`
    - peuvent retourner / envoyer moins de données que demandé
- Éviter les `read`/`write` sur des petites tailles (1 octet)
- Traiter correctement les erreurs
  - `EINTR` / `EAGAIN`
- Traiter les EOF (`read` qui renvoie 0)

utilitaire

```
int createServerSocket(uint16_t port) {
  struct sockaddr_in a;
  int s = socket(PF_INET,SOCK_STREAM,0);
  if (s==-1) {
    fprintf(stderr,"socket problem\n"); exit(EXIT_FAILURE);
  }
  bzero(&a,sizeof(a));
  a.sin_family = AF_INET;
  a.sin_port = htons(port);
  a.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(s,(struct sockaddr *)&a,sizeof(a))==-1) {
    fprintf(stderr,"bind problem\n"); close(s); exit(EXIT_FAILURE);
  }
  if (listen(s,0)==-1) {
    fprintf(stderr,"listen problem\n"); close(s); exit(EXIT_FAILURE);
  }
  if (fcntl(s,F_SETFL,O_NONBLOCK)<0) {
    fprintf(stderr,"problem setting non blocking state\n");
  }
  return s;
}
```

Select / poll

main

```
s1 = createServerSocket(61234);
s2 = createServerSocket(61235);
fd_set acceptSet;
FD_ZERO(&acceptSet); FD_SET(s1,&acceptSet); FD_SET(s2,&acceptSet);
do {
  memcpy(&readyToAcceptSet,&acceptSet,sizeof(fd_set));
  ready = select(MAX(s1,s2)+1,&readyToAcceptSet,NULL,NULL,NULL);
  if (ready==-1) break;
  while (ready--) {
    int s = FD_ISSET(s1,&readyToAcceptSet)?s1:s2;
    l = sizeof(c);
    if ((d=accept(s,(struct sockaddr *)&c,&l))==-1) {
      fprintf(stderr,"accept problem"); close(s);
      exit(EXIT_FAILURE);
    }
     FD_CLR(s,&readyToAcceptSet);
    if (getsockname(d,(struct sockaddr *)&local,&l)==-1) {
      fprintf(stderr,"getsockname problem\n");
    } else
      printf("Entered at %d",ntohs(local.sin_port));
    do_what_you_want_or_need(d);
  }
} while(1);
```

`poll`

- Demander au système (Unix ou autre) de réveiller le processus (thread) dès qu'une opération sera possible sur un descripteur parmi N autres.

- Appel système `poll`

  - fournir la liste des descripteurs de fichiers sur lesquels on attend que quelque chose soit possible.

    - structuration en descripteur+actions souhaitées et non pas en « champ de bits »

Select / poll

Conforme POSIX 1003.1

Terminal — less — 80×30

```
POLL(2)                    BSD System Calls Manual                    POLL(2)

NAME
     poll -- synchronous I/O multiplexing

SYNOPSIS
     #include <poll.h>

     int
     poll(struct pollfd fds[], nfds_t nfds, int timeout);

DESCRIPTION
     Poll() examines a set of file descriptors to see if some of them are
     ready for I/O or if certain events have occurred on them.  The fds argu-
     ment is a pointer to an array of pollfd structures, as defined in
     <poll.h> (shown below).  The nfds argument specifies the size of the fds
     array.

     struct pollfd {
         int    fd;       /* file descriptor */
         short  events;   /* events to look for */
         short  revents;  /* events returned */
     };

     The fields of struct pollfd are as follows:

     fd             File descriptor to poll.

:
```

`poll`

- les `events` possibles sont
  - `POLLIN` : lecture ou accept
  - `POLLOUT` : écriture
  - `POLLPRI` : lecture prioritaire (type OOB/URG)
  - `POLLHUP` : déconnexion
  - `POLLERR` : erreur

Select / poll

```
s1 = createServerSocket(61234);
s2 = createServerSocket(61235);

struct pollfd p[2];
p[0].fd = s1; p[0].events = POLLIN;
p[1].fd = s2; p[1].events = POLLIN;

int i; socklen_t l; struct sockaddr_in c, local;
char buffer[256]; int ready;
do {
  ready = poll(p,2,-1); // blocking on poll
  if (ready==-1) break;
  i = 0;
  while (ready--) {
    while (p[i].revents != POLLIN) i++;
    if (p[i].revents == POLLIN) do_the_job_with(p[i].fd);
  }
} while(1);
```

Asynchronous polling

- Demander au système (Unix ou autre) d'interrompre le processus dès qu'une opération sera possible sur un descripteur parmi N autres.

- réception du signal (en général SIGPOLL) nécessite :

  - association système du processus et du descripteur

  - installation d'un handler

  - activation du mécanisme

Association « système »

- par appel à `fcntl` avec
  - le descripteur
  - l'opération `F_SETOWN`
  - le processus (ou le groupe)

Choix du signal

- par appel à `fcntl` avec
  - le descripteur
  - l'opération `F_SETSIG`

cette opération n'existe pas dans TOUS les systèmes... (par défaut `SIGIO`)

  - le processus (ou le groupe)

Activation

- par appel à `fcntl` avec
  - le descripteur
  - l'opération F_SETFL
  - le mode contenant O_ASYNC