UNIVERSITE LIBRE DE BRUXELLES
Faculté des Sciences
Section Informatique

Année académique 2005–2006

INFO-F201 *Systèmes d'exploitation 1*, J. Goossens
Examen écrit : aide-mémoire

- `int open(const char *pathname, int flags)` : Ouvre le fichier `pathname` avec les permissions `flags` (`O_RDONLY`, `O_WRONLY`, ou `O_RDWR`) et renvoie le f.d. associé.
- `int close(int fd)` : Ferme le fichier de f.d. `fd`.
- `ssize_t read(int fd, void *buf, size_t count)` : Lit au plus `count` caractères sur `fd`, place le résultat dans `buf`, et renvoie le nombre de caractères lus.
- `ssize_t write(int fd, const void *buf, size_t count)` : Ecrit au plus les `count` premiers caractères de `buf` dans `fd` et renvoie le nombre de caractères écrits.
- `pid_t fork(void)` : Crée un processus fils et renvoie son PID.
- `pid_t getpid(void)` : Renvoie le PID du processus.
- `pid_t getppid(void)` : Renvoie le PID du processus père.
- `int execl(const char *path, const char *arg, ...)` : Exécute le programme `path` avec la liste d'aguments commençant par `arg` et terminée par NULL.
- `int execlp(const char *file, const char *arg, ...)` : Exécute le programme `file` (présent dans le `PATH`) avec la liste d'aguments commençant par `arg` et terminée par NULL.
- `int execv(const char *path, char *const argv[])` : Exécute le programme `path` avec comme paramètres `argv[]`.
- `int execvp(const char *file, char *const argv[])` : Exécute le programme `file` (présent dans le `PATH`) avec comme paramètres `argv[]`.
- `pid_t wait(int *status)` : Attend la fin d'un fils, stocke dans `status` son code de retour, et renvoie son PID.
- `pid_t waitpid(pid_t pid, int *status, 0)` : Attend la fin du processus fils de PID `pid`, stocke son code de retour dans `status`.
- `int pipe(int fd[2])` : crée un pipe anonyme et stocke dans `fd[0]` (resp. `fd[1]`) son f.d. de lecture (resp. d'écriture).
- `int mkfifo(const char *pathname, mode_t mode)` : crée un pipe nommé `pathname` avec `mode` comme permissions (`S_IRWXU, S_IRGRP, S_IROTH`, ...).
- `int dup(int oldfd)` : Renvoie un copie du f.d `oldfd`.
- `int dup2(int oldfd, int newfd)` : Copie le f.d. `oldfd` dans le f.d. `newfd`.

- Structures de données pour les sockets :

```
struct sockaddr_in {
   short int sin_family;        // Famille d'adresses (AF_INET)
   unsigned short int sin_port; // Numéro de port
   struct in_addr sin_addr;     // Adresse IP
   unsigned char sin_zero[8];   // Remplissage
};

struct in_addr {
   unsigned long s_addr; // 4 bytes IP address
};
```

- `uint32_t htonl(uint32_t hostlong)` : Renvoie la conversion *host to network* de `hostlong`.
- `uint16_t htons(uint16_t hostshort)` : Renvoie la conversion *host to network* de `hostshort`.
- `uint32_t ntohl(uint32_t netlong)` : Renvoie la conversion *network to host* de `netlong`.
- `uint16_t ntohs(uint16_t netshort)` : Renvoie la conversion *network to host* de `netshort`.
- `int inet_aton(const char *cp, struct in_addr *inp)` : Convertit l'adresse `cp` de notation pointée en notation entière et stocke le résultat dans `inp`.
- `char *inet_ntoa(struct in_addr in)` : Renvoie la conversion de notation entière en notation pointée de l'adresse `in`.
- `int socket(PF_INET, SOCK_STREAM, 0)` : Crée un socket et renvoie son f.d.
- `int bind(int sockfd, struct sockaddr *my_addr, int addrlen)` : Attache une paire adresse IP/numéro de port, stockée dans `sockaddr` (de taille `addrlen`), au socket `sockfd`.
- `int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)` : Se connecte à la machine distante d'adresse `serv_addr` (de taille `addrlen`) à l'aide du socket `sockfd`.
- `int listen(int sockfd, int backlog)` : Permet d'écouter des demandes de connection sur `sockfd` avec un file d'attente de taille `backlog`.
- `int accept(int fd, struct sockaddr *addr, int *addrlen)` : Permet de recevoir une connection sur le socket `fd` d'un ordinateur distant, dont l'adresse est stockée dans `addr` (de taille `addr_len`) et renvoie le nouveau f.d. de cette connection.
- `int getpeername(int fd, struct sockaddr *addr, int *addrlen)` : Stocke dans `addr` (de taille `addrlen`) l'adresse de la machine située à l'autre bout du socket `fd`.
- `int gethostname(char *hostname, size_t size)` : Stocke dans `hostname` de taille `size` le nom de la machine locale.

- Structure de données pour `gethostbyname` :

```
struct hostent {
        char    *h_name;
        char    **h_aliases;
        int     h_addrtype;
        int     h_length;
        char    **h_addr_list; //vecteur d'adresses
};
#define h_addr h_addr_list[0]
```

- `struct hostent *gethostbyname(const char *name)` : Renvoie un `hostent` contenant les adresses IP de la machine `name`.
- `FD_SET(int fd, fd_set *set)` : Ajoute `fd` à l'ensemble `set`.
- `FD_CLR(int fd, fd_set *set)` : Enlève `fd` à l'ensemble `set`.
- `FD_ZERO(fd_set *set)` : Vide l'ensemble `set`.
- `FD_ISSET(int fd, fd_set *set)` : Teste si `fd` appartient à l'ensemble `set`.
- `int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval* timeout)`

  Ecoute les ensembles de f.d. `readfds, writefds` et `exceptfds`, dont le maximum des f.d. + 1 est `n`, et stocke dans `readfds` les f.d. prêts à être lus (ou `accept()`és), et dans `writefds` ceux prêts à être écrits. Le dernier paramètre, `timeout`, permet de spécifier un délai maximal d'écoute, après lequel on sort automatiquement du `select` si aucun des file descriptors n'est devenu actif. Le paramètre `timeval` est du type suivant :

```
struct timeval{
    int tv_sec;   //secondes
    int tv_usec:  //millisecondes
};
```

  Si aucun délai n'est souhaité, le paramètre peut simplement être mis à `NULL`. La valeur de retour de `select` est le nombre de file descriptors actifs (donc `0` si on a dépassé le délai `timeout`).
- `int kill(pid_t pid, int sig)` : Envoi de signaux.
  -
```
SIGHUP 1   /* hangup */
SIGINT 2   /* interrupt */
SIGQUIT 3  /* quit */
SIGILL 4   /* illegal instruction */
SIGABRT 6  /* used by abort */
SIGKILL 9  /* hard kill */
SIGALRM 14 /* alarm clock */
SIGCONT 19 /* continue a stopped process */
SIGCHLD 20 /* to parent on child stop or exit */
```
  - Les signaux sont numérotés de 0 à 31.
  - Le signal `SIGKILL` ne peut être récupéré par une routine de gestion. Son effet est

toujours de terminer le processus à qui on l'envoie.
  - La combinaison de touches CTRL-C correspond à l'envoi du signal `SIGINT`.
- `sighandler_t signal(int signum, sighandler_t handler)` : Capture de signaux.
- `int shmget(key_t key, int size, int shmflg)` : Permet de créer une mémoire partagée.
- `void *shmat(int shmid, const void *shmaddr, int shmflg)` : Permet d'attacher la mémoire partagée à l'espace d'adressage d'un processus.
- `int shmdt(const void *shmaddr)` : Permet de détacher la mémoire partagée d'un processus.
- `int shmctl(int shmid, int cmd, struct shmid_ds *buf)` : Permet de contrôler la mémoire partagée (destruction...).
- `int semget(key_t key, int nsems, int semflg)` : Permet de créer un vecteur de sémaphores.
- `int semctl(int semid, int semnum, int cmd, ...)` : Permet de contrôler un vecteur de sémaphores :
  - `semctl(semid, n, SETVAL, value)` : Initialiser la valeur du n-iéme sémaphore.
  - `value = semctl(semid, n, GETVAL)` : Obtenir la valeur du n-iéme sémaphore.
  - `semctl(semid, 0, IPC_RMID)` : détruire un vecteur de semaphores.
- `int semop(int semid, struct sembuf *sops, unsigned nsops)` : Permet de réaliser des opérations sur le vecteur de sémaphores.
  - Le paramétre `sops` est un vecteur d'opérations à réaliser.
  - Le paramétre `nsops` est le nombre d'opérations à réaliser.
  - Les opérations sont codées dans un `struct sembuf` :
    ```
    struct sembuf {
        ushort_t sem_num;
        short sem_op;
        short sem_flg;
    }
    ```
  - `sem_num` est l'indice du sémaphore dans le vecteur de sémaphores sur lequel on veut faire l'opération.
  - Si `sem_op` est négatif, on demande `sem_op` exemplaires de la ressource (P). Si il est positif, on relâche `sem_op` exemplaires de la ressource (V)
  - L'appel à `semop()` est bloquant si **toutes** les opérations ne peuvent être effectuée en même temps.
  - `int pthread_create(pthread_t* thread, pthread_attr_t* attr, void* (*start_routine)(void*), void* arg)` : Permet de créer un *thread*.
  - `int pthread_join(pthread_t thread, void **thread_return)` : Permet de synchroniser un *thread* avec la fin d'un autre *thread*.
  - `void pthread_exit(void *retval)` : Permet de terminer un *thread*.
  - `int pthread_cancel(pthread_t thread)` : Permet de terminer ou couper un autre *thread*.
  - `int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);` : Permet d'initialiser un *mutex*.

- `int pthread_mutex_lock(pthread_mutex_t *mutex)` : Permet de verrouiller un *mutex* de manière bloquante.
- `int pthread_mutex_trylock(pthread_mutex_t *mutex)` : Permet de verrouiller un *mutex* de manière non bloquante.
- `int pthread_mutex_unlock(pthread_mutex_t *mutex)` : Permet de déverrouiller un *mutex*.