💯

Recap

➡️ **QUESTIONS**

▼ Pipe fermée en read / write

- both sides opened
  `read` is blocking if the pipe is **empty**
  `write` is blocking if pipe is **full**
- closed **write** side
  `read` returns **0** → **EOF**
- closed **read** side
  `write` raises an **exception** → **"Broken pipe"** `SIG_PIPE`

▼ Différence entre stdout et STDOUT_FILENO

`stdout` is a `FILE *` pointer giving the standard output stream.

use `stdout` for `<stdio.h>` functions such as `fprintf()`, `fputs()` etc..

`STDOUT_FILENO` is an integer file descriptor. `#define STDOUT_FILENO 1` use it for `write` syscall `write(STDOUT_FILENO, *buff, size_t)`

The relation between the two is `STDOUT_FILENO == fileno(stdout)`

▼ Pourquoi un processus qui boucle inf ne monopolise pas le processeur

- Interrupt is a signal sent to a CPU
  Can be sent by **external hardware** (Keyboard, mouse, timers, ...)
  Or raised by the **CPU itself**
  Only information → `interrupt number`
- generaly CPU forced to handle interrupts instantly
  • Jump to a predefined routine address → `interrupt handler`

- Each interrupt can have its own interrupt handler
  An `interrupt vector table` must be setup in **RAM** →Done by the **Kernel**
- The interrupt handler calls `iret` to resume previous execution

- prevent processes running infinite periods, the kernel sets up a timer
  - `timer interrupt` periodically triggered → **~10ms**
  - This ensures that the associated **kernel routine** will be executed on a regular basis
  - `interrupt vector table` must be initialized beforehand

▼ **Signal Masqué ?**
Plusieurs occurrences d'un signal envoyées à un processeur qui a masqué ce signal ?

▼ Signal Masqué

signal auquel un processus a temporairement désactivé la réaction normale
signaux masqués sont mis en file d'attente et ne sont pas perdus

signaux masqués en utilisant la fonction système `sigprocmask + SIG_BLOCK` pour spécifier un ensemble de signaux qui doit être masqué
lever le masquage d'un signal : fonction `sigprocmask + SIG_UNBLOCK`

▼ Reception Signal Masqué

1. **chaque occurrence mise en file d'attente** → aucun signal n'est perdu.
2. **signaux pas immédiatement traités** → tant que le signal est masqué le processus ne réagit pas aux signaux reçus
   `handler` associés au signal pas exécutés + comportement normal du processus pas interrompu
3. **signaux traités quand masquage est levé** `sigprocmask` → les signaux en attente sont traités dans l'ordre où ils ont été reçus
   exécution du `handler` associées au signal

Sometimes, the delivery of a signal is not desirable
- During the update of a complex data structure
- When the process is not (yet) ready to catch signals

A process can block the delivery of signals

- The kernel maintains an **array of bits** → `current signal mask`
- Blocked signals will stay **pending** until delivery is allowed again

The **sigprocmask syscall** modifies the current signal mask

`int sigprocmask(int how, sigset_t *set, sigset_t *oset);`

1. Construct a mask (set) in user space
2. Call sigprocmask and tell **how** to combine the provided mask with the current signal mask:
   `SIG_SETMASK` → **replace** entire mask
   `SIG_BLOCK` → **block signals marked with 1** - keep others as is
   `SIG_UNBLOCK` → **unblock signals marked with 1** - keep others as is

---

▼ Primitives de la bibliothèque standard fread / fwrite → plus ou moins performantes que les syscall read / write ?
Que renferme le type FILE?

La "légère perte de performances" associée à l'utilisation des opérations de la bibliothèque standard par rapport aux appels système est principalement due à l'overhead supplémentaire induit par la couche de la bibliothèque standard. Voici quelques détails sur cet overhead :

1. **Buffering** : Les fonctions de la bibliothèque standard, comme `fwrite`, implémentent souvent des tampons de mémoire pour regrouper les opérations d'écriture. Cela signifie que les données sont d'abord écrites dans un tampon en mémoire, puis le tampon est vidé sur le disque lorsque certaines conditions sont remplies. Cela introduit un délai supplémentaire par rapport à un appel système direct, qui envoie immédiatement les données au noyau.
2. **Conversion de données** : Les fonctions de la bibliothèque standard peuvent effectuer des conversions de données, telles que la conversion de fins de lignes, la gestion des caractères spéciaux, etc. Cela peut ajouter un surcoût par rapport aux appels système qui fonctionnent avec des données brutes.
3. **Gestion des métadonnées** : Les opérations de la bibliothèque standard gèrent généralement les métadonnées associées aux fichiers, telles que les droits d'accès, les dates de modification, etc. Cela peut entraîner des opérations de lecture/écriture supplémentaires dans certaines circonstances.

4. **Surcharge de la pile** : Les appels aux fonctions de la bibliothèque standard ajoutent un peu de surcharge à la pile d'appels de la fonction, en passant par plusieurs couches de code avant d'atteindre l'implémentation du système sous-jacent. Les appels systèmes sont généralement plus directs et ont moins de surcharge de pile.

Il est important de noter que la perte de performances est généralement minime pour la plupart des applications, en particulier pour les petits volumes de données. Les avantages en termes de portabilité, de simplicité d'utilisation et de facilité de maintenance peuvent souvent l'emporter sur cette légère perte de performances.

Cependant, si les performances brutes sont essentielles pour votre application, ou si vous traitez de grands volumes de données, il peut être utile d'envisager des appels systèmes plus directs pour les opérations d'entrée/sortie critiques.

▼ **Dans quelles circonstances un processus peut-il se voir délivrer le signal SIGSEGV ?**
Qui lui envoie ce signal ?
Expliquez la succession d'événements ayant conduit à cet envoi. Le signal est-il forcément fatal au destinataire ?

Le signal SIGSEGV, également appelé "**Segmentation Fault**", est envoyé à un processus par le **noyau du système d'exploitation** lorsqu'il tente :

- Accès à une **zone mémoire non allouée**
- Accès à une **zone mémoire protégée**
- Accès à une **adresse mémoire invalide**
- **Dépassement de tampon** (buffer overflow)

Il est possible de **catch le signal** et d'appeler un **handler**. **SIGSEGV n'est pas fatal** si il est correctement géré par le processus.

➡️ **FILES**

- après `int ret = read(...)` → check `if (ret == 0) return -1;`

- pour read une valeur numérique → `read(fd, &val , size_t)` ne pas oublier **&**
- pour read un string → `read(fd, str , size_t)` ne pas déréférencer

```
// GESTION D'ERREUR
if (open(char *filename, O_WRONLY | O_CREATE | O_TRUNC, 0644) == -1)
{ perror("open"); exit(1); }

if (read(_,_,1) != 1) { fprintf(stderr, "_"); exit(1); }

if (atoi(_) < 1) { fprintf(stderr, "_"); exit(1); }
```

➡️ PROCESSUS / PIPES

- après `exec*` → check d'erreur `perror("exec*");`
- cmd1 arg1 | cmd2 arg2 | cmd3 arg3

```
int tube[2][2]; // one pipe between 2 children : CHILD 1 >⟹ CHILD 2 >=
⟹ CHILD 3
for (int i = 0; i < 2; i++) pipe(tube[i]);
pid_t child[3]; // one processus per child / command

child[0] = fork();
if (!child[0]) {
    close(tube[0][R]; close(tube[1][W]); close(tube[1][R]);
    dup2(tube[0][W], STDOUT_FILENO); close(tube[0][W]);
    execlp(cmd1, cmd1, arg1, NULL); perror(cmd1);
}
...

for (int i = 0; i < 2; i++) { close(tube[i][W]); close(tube[i][R]); }

// catch errors
for (int i = 0; i < 3; i++) {
    int status; pid_t p = wait(&satus);
```

```
    if (WIFEXITED(status) == 0 || WEXITSTATUS(status) != 0)
        perror("process terminated anormaly");
}
```

➡️ SIGNALS

```
int sigaction(int signum,const struct sigaction *restrict sa, struct sigaction *
restrict oldsa);

void handler (int sig) { }

struct sigaction sa;
sa.sa_flags = 0;
// SA_ONSTACK, SA_RESTART, SA_NODEFER (SA_NOMASK), SA_RESETHA
ND (SA_ONESHOT)
sigemptyset(&sa.sa_mask);
sa.sa_handler = handler;
// SIG_IGN
sigaction(SIGINT, &sa, NULL);
// SIGALRM, SIGUSR1, SIGUSR2, SIGCHLD, ...

/* ------------------------------------------------------------- */

sigset_t set, old;
sigemptyset(&set);
sigaddset(&set, SIGINT);

// BLOCK THE SIGNAL
int sigprocmask(SIG_BLOCK, set, old);
// previous set stored in old, can be NULL

// UNBLOCK
sigprocmask(SIG_SETMASK, &old, NULL);
sigprocmask(SIG_UNBLOCK, &set, &old);
```

```
sa.sa_handler = SIG_IGN;
sigaction(SIGINT, &sa, NULL);

#define SIG_BLOCK 0, SIG_UNBLOCK 1, SIG_SETMASK 2

/* ------------------------------------------------------------- */

#define SIGALRM 14
alarm(x) → sleep(x) + raise(SIGALRM)

/* ------------------------------------------------------------- */

sigjmp_buf env; // global
sigsetjmp(env, 1); // checkpoint
siglongjmp(env, 1); // go to checkoint
```

➡️ THREADS

- **STRUCTURE**

```
int NB_THREADS = 4;

struct thd {
    int id;
    // more data
} typedef THD;

void *start(void *p) {
    THD *thread = (THD *)arg;
    // thread→id is accessible

    // dive work by horizontal lines 1 px high
    for (int y = thread→id; y < HEIGHT; y += NB_THREADS)
      for (int x = 0; x < WIDTH; x++)
        test++; // printf("%lu\n", test);
```

```
    return NULL;
}

int main(int ac, char** av) {
    pthread_t tid[NB_THREADS];
  THD threads[NB_THREADS];

  for (int i = 0; i < NB_THREADS; i++) {
    threads[i].id = i;
    pthread_create(tid + i, NULL, start, (void *)&threads[i]);
  } // start at tid + 1 : 0 is main

  for (int i = 0; i < NB_THREADS; i++)
    pthread_join(tid[i], NULL);

    return 0;
}
```

- **BARRIER**

```
pthread_barrier_t barrier;

pthread_barrier_init(&barrier, NULL, NB_THREADS);

// wait for all the thread to reach this line
pthread_barrier_wait(&barrier);

pthread_barrier_destroy(&barrier);
```

- **MUTEX**

```
pthread_mutex_t mutex;

pthread_mutex_init(&mutex, NULL);

pthread_mutex_lock(&mutex);
// NO CONCURENT CODE
```

```
pthread_mutex_unlock(&mutex);

pthread_mutex_destroy(&mutex);
```