SCIENCES SORBONNE UNIVERSITÉ

Examen Réparti 2 PSCR
Master 1 Informatique Jan 2022

UE 4I400

Année 2021-2022

1 heure 30 – **Tout document papier autorisé sauf corrigés fournis de l'UE imprimés (annales, TD, TME)**
Tout appareil de communication électronique interdit (téléphones. . .)

Introduction

- Le barème est sur 21,5 points et est donné à titre indicatif et susceptible d'être modifié.
- Dans le code C++ demandé vous vous efforcerez d'ecrire du code correct et compilable, mais les petites typos ne seront pas sanctionnées.
- De même les problèmes d'include et de namespace ne sont pas pertinents (pas la peine de qualifier les `std::`, ni de citer tous les `#include`)
- On suppose également que **tous les appels système se passent bien**, on ne demande pas de contrôler les valeurs de retour (donc pas de `perror` ou `errno.h`).
- On vous fournit une annexe recensant la signature de certains appels système.

Le sujet est composé de sous sections indépendantes qu'on pourra traiter dans l'ordre qu'on souhaite. On va réaliser au cours de cet énoncé un même comportement de diverses manières, en utilisant d'abord les primitives C++, puis les primitives IPC Posix.

Une écriture pour N lectures

On considère une classe de synchronisation permettant à des producteurs d'envoyer un message à des consommateurs. Chaque message envoyé doit être lu exactement N fois. On ne peut pas envoyer de nouveau message tant que le message actuel n'a pas été lu N fois. Quand le message a été lu N fois, on peut de nouveau écrire un message, qui sera de même lu N fois.

Pour simplifier l'étude, on considère que les message sont des tableaux de 256 caractères. Nous aurons donc l'API suivante dans les questions 1 et 2 :

- On passe la valeur N à la construction; elle n'évoluera pas. Initialement le buffer est vide et donc prêt à traiter une écriture.
- `void write (const char * msg)` attend si nécessaire que le message précédent ait été lu N fois, puis recopie le message argument (de longueur 256 char) dans un attribut de la classe.
- `void read (char * msg)` lit le message s'il y en a un disponible qui n'a pas été lu N fois, sinon attend que ce soit le cas. La lecture va consister à recopier à l'adresse *msg* le contenu du buffer stocké. On doit donc garantir que *msg* est un pointeur vers une zone allouée d'au moins 256 octets.

Gestion avec des threads (4 points)

Question 1. (4 points) Ecrire la classe `BufferNRead` qui réalise ce comportement, en supposant que les producteurs et consommateurs sont des thread. On utilisera donc exclusivement des `mutex` et/ou des `condition_variable` pour réaliser la synchronisation. On demande d'utiliser deux conditions différentes pour réguler l'accès à read et à write. On recommande de comptabiliser les lectures ayant déjà eu lieu à l'aide d'un attribut de la classe. On recommande l'utilisation de `memcpy` sur 256 octets pour copier les messages depuis et vers le buffer.

Sémaphores et segments de mémoire partagée (7 points)

Question 2. (4 points) Ecrire la classe `BufferNReadProc` qui réalise ce comportement, en supposant que les producteurs et consommateurs sont des processus distincts, et que la classe est logée dans un segment de mémoire partagée. On utilisera exclusivement des `semaphores` POSIX pour réaliser la synchronisation. On ne s'appuiera que sur les compteurs internes à ces sémaphores; on demande de ne pas comptabiliser le nombre de lecteurs à l'aide d'une variable. On considère qu'il peut éventuellement y avoir plusieurs producteurs qui tentent d'accéder à `write` en concurrence; leurs messages ne doivent pas s'entrelacer.

Question 3. (1,75 points) Ecrire un programme *main* qui crée un `BufferNReadProc` (N étant passé comme premier argument au main) dans un segment de mémoire partagée accessible depuis n'importe quel processus de la machine à l'adresse "/mybuff".

Question 4. (1,75 points) Ecrire un programme *main* qui écrit dans le buffer existant à l'adresse "/mybuff" le message "Hello World".

Gestion à l'aide de `pipe` (5 points)

Dans cette partie nous allons utiliser un tube nommé "monpipe" ainsi qu'un sémaphore nommé "/monsem". L'objectif reste de réaliser la sémantique "une écriture suivie de N lectures".

Question 5. (1,75 point) Donnez les appels systèmes utiles pour engendrer ces objets nommés et pour s'en débarasser.

Question 6. (1,75 points) Ecrire un programme consommateur qui lit un message de 256 caractères depuis le tube et l'affiche.

Question 7. (2,25 points) Ecrire un programme qui écrit N fois (N lu dans le premier argument) un message (lu dans le deuxième argument et supposé contenir moins de 256 caractères) dans le tube. Même si on lance en concurrence ce programme on souhaite que les messages ne s'entrelacent pas. On peut par exemple invoquer `./writer 12 "Hello World"` pour envoyer 12 fois le message "Hello world" (paddé à 256 caractères).

Gestion à l'aide de `socket` TCP (4 points)

Dans cette partie nous allons utiliser des socket TCP pour réaliser la sémantique "une écriture suivie de N lectures". On ne traite que la partie serveur dans cet énoncé.

Ecrire un programme (serveur) qui engendre deux sockets TCP qui attendent des connexions respectivement sur le port 3000 et 4000. La socket 3000 est utilisée pour gérer les demandes d'écriture (write); quand un client s'y connecte, il doit nous transmettre un message de 256 caractères. Ce message est stocké dans le serveur. La socket 4000 est utilisée pour gérer les demandes de lecture (read); quand un client s'y connecte, on lui transmet (une fois) le message stocké. On fermera proprement les connections après usage, chaque connection n'est utilisée que pour transmettre un seul message,

Question 8. (4 points) Ecrire un programme qui lit la valeur de N dans son premier argument, instancie les sockets comme décrit, puis boucle indéfiniment sur le comportement suivant : traiter une demande client en écriture sur le port 3000, puis traiter N demandes en lecture sur le port 4000.

Annnexe

Signature par ordre alphabétique de certains appels système Posix utilisés dans l'UE.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
int close(int fd);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
pid_t fork(void);
int ftruncate(int fd, off_t length);
void freeaddrinfo(struct addrinfo *res);
int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints,
                struct addrinfo **res);
pid_t getpid(void);
pid_t getppid(void);
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
int kill(pid_t pid, int sig);
int listen(int sockfd, int backlog);
void *memcpy(void *dest, const void *src, size_t n);
void *memset(void *s, int c, size_t n);
int mkfifo(const char *pathname, mode_t mode);
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int pipe(int pipefd[2]);
ssize_t read(int fd, void *buf, size_t count);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
int sem_destroy(sem_t *sem);
int sem_init(sem_t *sem, int pshared, unsigned int value);
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
int shutdown(int socket, int how);
int sigsuspend(const sigset_t *mask);
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int socket(int domain, int type, int protocol);
int unlink(const char *pathname);
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
ssize_t write(int fd, const void *buf, size_t count);
```