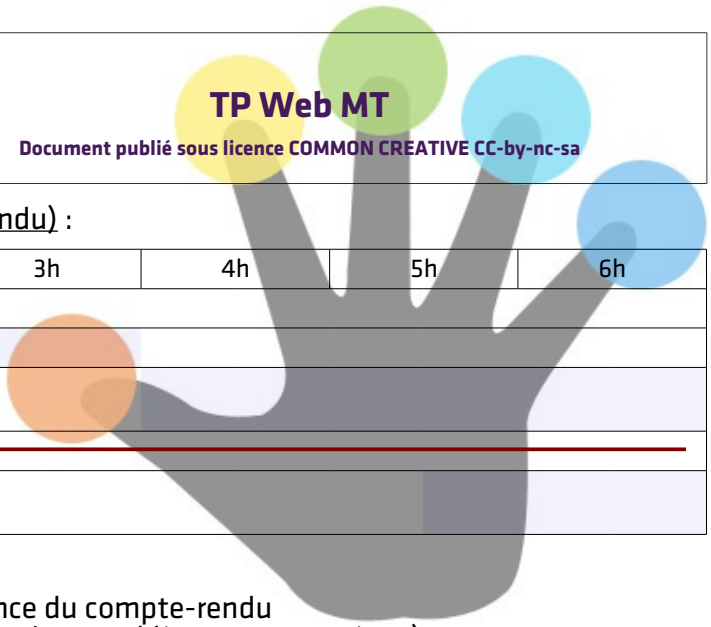
Lycée La Fayette
Champagne-sur-Seine • Fontaineroux

Responsable pédagogique	**AF**	AM	OP	PM
Période	Sem1	Sem2	Sem3	Sem4
Volume horaire	Cours/TD		TP	
			6+8	

TP Web MT



Indicateur temporel (hors rédaction du compte-rendu) :

questions	1h	2h	3h	4h	5h	6h
Etape 0						
Etape 1						
Etape 2 (3 à 6h)						
Etape 3 (le reste)						

Documents à rendre : qualité et pertinence du compte-rendu (analyse, commentaires, problèmes rencontrés,...).

Critères d'évaluation : Respect du Cahier des Charges et respect du plan du TP,

qualité des logiciels écrit.

Acquis de ce TP : Programmation système par thread, sémaphores, algorithme producteurs/consommateurs

Le Contexte

Les serveurs Web sont par nature multi-tâches, puisqu'ils permettent à plusieurs clients d'avoir accès à un contenu multimédia en même temps. Dans l'industrie, le Web, en tant que interface homme machine se généralise. Elle permet de configurer un routeur, un calculateur, un automate aisément. De même, les organes actifs d'un processus industriel peuvent produire des pages HTML faciles et agréables à lire.

Un Serveur Web est un simple intermédiaire entre les pages HTML et vos navigateurs, nous n'évoquerons pas ici les langages dynamiques tel que le PHP, JSP, servlet, et encore moins ASP.

Le protocole permettant le dialogue entre le navigateur et le serveur est HTTP.

Objectif

Nous allons étudier et modifier un serveur web de manière à le rendre multi-tâche. Cette démarche ne se fera pas en une fois, mais en une succession d'étapes vous permettant de comprendre les mécanismes de la programmation avec plusieurs processus.

Vous devrez conserver la trace de chaque étapes (sources inclus).

Pré-requis

Pour pouvoir effectuer ce TP dans les meilleurs condition vous devez avoir également des notion sur le protocole HTTP (lire l'annexe1).

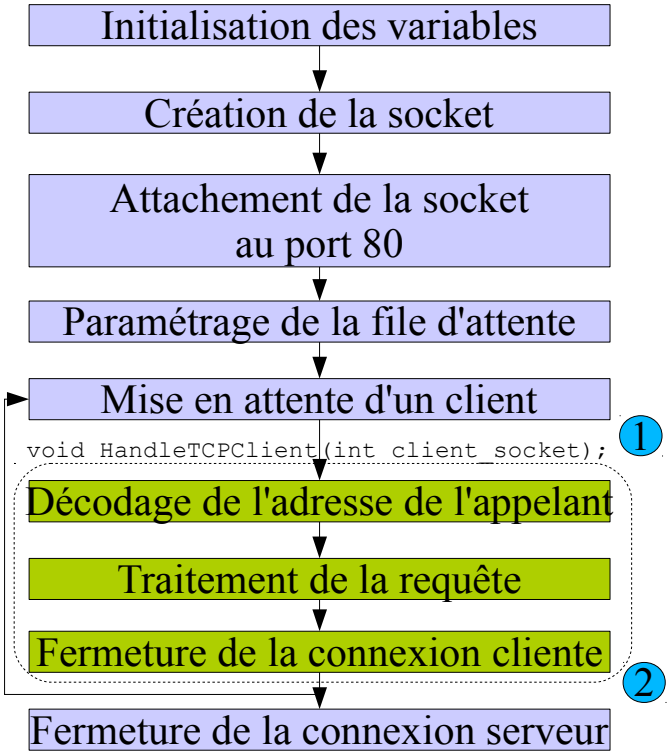
Vous devez également être capable de développer sur un système d'exploitation tel que Linux (appel système, compilation, recherche de documentation, ...).

Ah oui, j'oubliais, vous devez aussi lire tout le sujet, mais si vous lisez ces lignes, vous êtes déjà en bonne voie.

Présentation du travail à faire

Actuellement, il vous est fournit un code source fonctionnel qui implémente le protocole HTTP en version minimal (méthode GET). Ce code est mono-tâche. Lorsqu'il reçoit la requête d'un client, il la décode, ouvre le fichier recherché et renvoie ce fichier en ayant préalablement renvoyé un en-tête.

Voici l'algorithme du programme dans sa version 0.1 :



Votre travail va être de faire évoluer ce serveur vers une version multi-tâches. Les tâches bleu sont des tâches réseaux, ces étapes sont incontournables dans un serveur TCP. Votre travail s'insérera entre les étapes réseaux et les étapes dites « métiers » (1) et (2). Les tâches métiers sont le décodage d'une requête HTTP GET 1.0 et la réponse à cette demande. Ce fonctionnement est minimal, voir rustique. Ce code métier est contenu dans la fonction void HandleTCPClient(int client_socket);.

Etape0 : V0.1, Mise en Œuvre du serveur

Dans cette étape vous allez compiler ce serveur et vérifier son fonctionnement.

L'étape 0 consiste à comprendre le code source fournit, à identifier les différentes étapes de l'algorithme présenté au chapitre précédent.

Une fois votre programme exécutable et exécuté vous devez ouvrir un navigateur. Ce navigateur aura pour url quelque chose qui ressemble à : http://localhost:8080/index.html

le numéro 8080 est le port d'écoute du serveur (à passer en argument sur la ligne de commande). Le fichier index.html est le fichier à télécharger. Un fichier est fournit, avec 8 images, il est à placer dans « ./docRoot » .

Remarque : le fonctionnement ne vous parait probablement ralenti , pour simuler la charge de votre machine une pause est ajouté dans l'envoie du fichier.

Etape1 : V0.2, Fermeture propre du serveur

Présentation du problème

Lorsque vous souhaitez éteindre le serveur vous avez la possibilité de faire un kill sur le PID ou un CTR^C. Or lors de cette opération le serveur est arrêté brutalement, toutes les ressources ne sont pas libérées proprement.

Présentation de la solution

La solution consiste à dérouter le CTR^C de manière à fermer proprement la socket de travail, de mettre fin à la transmission, puis de déconnecter la socket d'écoute..

Mise en œuvre

Pour ce faire je vous propose de dérouter le signal SIGINT par la primitive signal (tel que présenté dans le cours) vers une fonction *« get_blody_signal(...) »* qui s'occupera du travail de fermeture, c'est à dire, appeler la fonction « void die() » déjà existante.

Travail à faire :

- écrire la fonction « void get_blody_signal(int sig) »
- modifier le main pour dérouter SIGINT vers celle-ci.
- tester la solution
- Remarque profitez en pour vous protéger de SIGPIPE, cela vous permettra de survivre au « send error »

Etape2 : V0.3 Serveur « Threadé »

Présentation du problème

Chaque traitement est fait en séquentiel, or le facteur limitant d'une connexion web n'est pas la puissance du processeur, mais c'est le temps de lecture du disque dur vers le serveur web et le temps de transfert réseau.

Les architectures mono client ne permettent pas d'optimiser les performances. Les architectures multi-processus lourd (fork) sont assez lourdes pour le système d'exploitation. Les threads, processus léger offrent une alternative intéressante.

Présentation de la solution

La solution consiste à paralléliser l'exécution du traitement de manière à optimiser l'utilisation des ressources disques (lectures des fichiers) et réseau.

Mise en oeuvre

Je vous propose d'utiliser la primitive pthread_create(...) de manière à paralléliser le traitement des requêtes.

(réfléchissez avant d'agir)

L'étape 1 est elle toujours acquise? Pourquoi ?

Travail à faire :

- réfléchir à une solution,
    - vous pouvez créer une nouvelle thread pour chaque nouveau client, sans garder trace de cette thread. Cette methode, n'est pas très ortodoxe et risque de vous poser des problème

par la suite. C'est peut être néanmoins une première étape à votre travail.

- vous pouvez plutôt posséder un « pool » de thread prêtent à traiter une demande, cette solution bien que élégante est un peu complexe à codé.
- Une solution mixte est aussi adaptée à notre situation, un tableau de N case contenant une thread plus l'état de cette « case » libre ou pas. Une nouvelle thread est crée pour chaque nouveau client, si il n'y a plus de case libre, il faut attendre qu'une se libère. L'attente peux se faire soit par une bête boucle, soit par un sémaphore initialisé au nombre de cases...

- concevoir sur papier les modifications à effectuer (papier original a rendre dans votre compte-rendu)
- mise en oeuvre de vos modifications
- test, mise au point.

Etape3 : V0.4 Log

Présentation du problème

Il est désagréable de voir s'afficher à l'écran beaucoup d'informations. Ces informations devraient plutôt être stocké dans un fichier.

Présentation de la solution

La solution pour ne plus voir s'afficher trop d'informations à l'écran est d'écrire dans un fichier de « log », ceci permettra la recherche à posteriori d'éléments contenu dans ce dernier.

Ce fichier sera rempli d'une part par la tache père et d'autre part par les threads fils. Il y a donc un gros risque de « collision » lors de l'écriture dans ce fichier.

La solution pour que cette écriture soit sécurisé consiste à en protéger son accès par un sémaphore.

Mise en oeuvre

Je vous propose dans un premier temps de coder la gestion du log, puis dans un second temps la sécurisation de celui ci, par exemple grâce à un sémaphore.

On pourrait imaginer une fonction **void log(int num_thread, char *message);** utilisée en lieu et place des appels à printf ou toutes autre fonctions à écriture direct sur stdout. Cette fonction écrira dans le fichier de log, puis dans un second temps elle se protégera elle même par un sémaphore.

RQ : l'écriture dans un log se fait en mode non bufferisé (pour que en cas de problème on puisse récupérer le maximum d'informations).

Travail à faire :

- écrire la fonction log() basique
- modifier tous les sources pour passer par log.
- test, mise au point
- évolution de log() avec un mutex
- test, mise au point

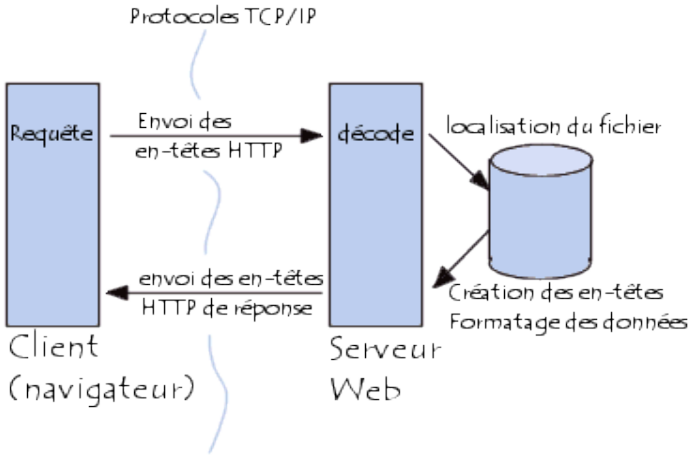
Annexe1 : Rappel sur HTTP

Le protocole HTTP (HyperText Transfer Protocol) est le protocole le plus utilisé sur Internet depuis 1990. La version 0.9 était uniquement destinée à transférer des données sur Internet (en particulier des pages Web écrites en HTML. La version 1.0 du protocole (la plus utilisée) permet désormais de transférer des messages avec des en-têtes décrivant le contenu du message en utilisant un codage de type MIME.

Le but du protocole HTTP est de permettre un transfert de fichiers (essentiellement au format HTML) localisé grâce à une chaîne de caractères appelée URL entre un navigateur (le client) et un serveur Web (appelé d'ailleurs *httpd* sur les machines UNIX)

- **Communication entre navigateur et serveur**

  La communication entre le navigateur et le serveur se fait en deux temps:



  - Le navigateur effectue une **requête HTTP**
  - Le serveur traite la requête puis envoie une **réponse HTTP**

- **Requête HTTP**

  Une requête HTTP est un ensemble de lignes envoyé au serveur par le navigateur. Elle comprend:

  - **une ligne de requête**: c'est une ligne précisant le type de document demandé, la méthode qui doit être appliquée, et la version du protocole utilisée. La ligne comprend trois éléments devant être séparé par un espace:
    - **La méthode**
    - **L'URL**
    - **La version du protocole utilisé par le client (généralement *HTTP/1.0*)**
  - **Les champs d'en-tête de la requête**: il s'agit d'un ensemble de lignes facultatives permettant de donner des informations supplémentaires sur la requête et/ou le client (Navigateur,système d'exploitation,...). Chacune de ces lignes est composée d'un nom qualifiant le type d'en-tête, suivi de deux points (:) et de la valeur de l'en-tête
  - **Le corps de la requête**: C'est un ensemble de ligne optionnel devant être séparé des lignes précédentes par une ligne vide et permettant par exemple un envoi de données par une commande POST lors de l'envoi de données au serveur par un formulaire

  Une requête HTTP a donc la syntaxe suivante (*<crlf>* signifie retour chariot ou saut de ligne):

```
METHODE URL VERSION<crlf>
EN-TETE : Valeur<crlf>
.
EN-TETE : Valeur<crlf>
Ligne vide<crlf>
CORPS DE LA REQUETE
```

Voici donc un exemple de requête HTTP:

```
GET http://www.commentcamarche.net HTTP/1.0
Accept : text/html
If-Modified-Since : Saturday, 15-January-2000 14:37:11 GMT
User-Agent : Mozilla/4.0 (compatible; MSIE 5.0; Windows 95)
```

- **Réponse HTTP**

Une réponse HTTP est un ensemble de lignes envoyé au navigateur par le serveur. Elle comprend:

- **une ligne de statut**: c'est une ligne précisant la version du protocole utilisé et l'état du traitement de la requête à l'aide d'un code et d'un texte explicatif. La ligne comprend trois éléments devant être séparé par un espace:
  - **La version du protocole utilisé**
  - **Le code de statut**
  - **La signification du code**
- **Les champs d'en-tête de la réponse**: il s'agit d'un ensemble de lignes facultatives permettant de donner des informations supplémentaires sur la réponse et/ou le serveur. Chacune de ces lignes est composé d'un nom qualifiant le type d'en-tête, suivi de deux points (:) et de la valeur de l'en-tête
- **Le corps de la réponse**: Il contient le document demandé

Une réponse HTTP a donc la syntaxe suivante (*<crlf>* signifie retour chariot ou saut de ligne):

```
VERSION-HTTP CODE EXPLICATION<crlf>
EN-TETE : Valeur<crlf>
.
EN-TETE : Valeur<crlf>
Ligne vide<crlf>
CORPS DE LA REPONSE
```

Voici donc un exemple de réponse HTTP:

```
HTTP/1.0 200 OK
Date : Sat, 15 Jan 2000 14:37:12 GMT
Server : Microsoft-IIS/2.0
Content-Type : text/HTML
Content-Lentgh : 1245
Last-Modified : Fri, 14 Jan 2000 08:25:13 GMT
```

Pour plus d'informations sur le protocole HTTP, le mieux est de se reporter à la RFC 1945 expliquant de manière détaillée le protocole :

- RFC 1945 traduite en français
- RFC 1945 originale